SC22/WG11 N470$\alpha$2

# DRAFT INTERNATIONAL **ISO/IEC** STANDARD **WD 10967-3**

WD for the First edition
Date: 2000-11-26

# Information technology — Language independent arithmetic —

## Part 3: Complex floating point arithmetic and complex elementary numerical functions

*Technologies de l'information —*
*Arithmétique indépendante des languages —*

*Partie 3: Arithmétique des nombres en virgule flottante complexe et fonctions numériques élémentaires complexe*

**EDITOR'S WORKING DRAFT**
**November 27, 2000 9:52**

**Editor:**
**Kent Karlsson**
**IMI, Industri-Matematik International**
**Kungsgatan 12**
**SE-411 19 Göteborg**
**SWEDEN**
**Telephone: +46-31 10 22 44**
**Facsimile: +46-31 13 13 25**
**E-mail: keka@im.se**

Reference number
ISO/IEC WD 10967-3.1:2000(E)

## Copyright notice

# Contents

# Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialised system for world-wide 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 organisations, 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 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, *Implementation of information technology*. 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 10967-3 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Implementation of 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 floating point arithmetic and complex elementary numerical functions*

Additional parts will specify other arithmetic datatypes or arithmetic operations.

**Annex B is informative and is intended to be read in parallel with the main normative text of the standard.**

**Notes and Annexes B, C, and D as well as notes are informative.**

# 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.

Part 2 of ISO/IEC 10967 specifies a number of additional operations for integer and floating point types, in particular specifications for numerical approximations to elementary functions on reals.

### The content

The content of this part is based on part 1 and part 2, and extends part 1's and part 2's specifications to specifications for operations approximating integer complex (Gaussian integers) arithmetic, real complex arithmetic, and real complex elementary functions.

The numerical functions covered by this part are computer approximations to mathematical functions of one or more complex 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 extract apparently implementation dependent data (such as the exponent of a floating point number) in an implementation independent way. 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 3: Complex arithmetic and complex elementary numerical functions

## 1 Scope

This part of ISO/IEC 10967 defines the properties of numerical approximations for complex arithmetic operations and many of the complex 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 and part 2 are included by reference in this part.

This part provides specifications for numerical functions for which operand or result values are of complex integer or complex floating point datatypes constructed from integer and 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 provides specifications for

a) basic complex integer (Gaussian integer) operations,

b) non-transcendental Cartesian complex floating point operations,

c) exponentiations, logarithms, hyperbolics, and

d) trigonometric operations for Cartesian complex floating point.

This part also provides specifications for

e) the results produced by an included floating point operation when one or more operand values include IEC 60559 special values, and

f) program-visible parameters that characterise certain aspects of the operations.

### 1.2 Exclusions

This part provides no specifications for:

a) Datatypes and operations for polar complex floating point. This standard neither requires nor excludes the presence of such polar complex datatypes and operations.

b) Numerical functions whose operands are of more than one datatype. This standard neither requires nor excludes the presence of such "mixed operand" operations.

c) A complex interval datatype, or the operations on such data. This standard neither requires nor excludes such data or operations.

d) A complex fixed point datatype, or the operations on such data. This standard neither requires nor excludes such data or operations.

e) A complex rational datatype, or the operations on such data. This standard neither requires nor excludes such data or operations.

f) Matrix, statistical, or symbolic operations. This standard neither requires nor excludes such data or operations.

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

h) The properties of integer and floating point datatypes that properly belong in 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,
   4) the consequences of applying an operation to values of improper datatype, or to uninitialised data,
   5) the rules for assignment, parameter passing, and returning value,
   6) the presence or absence of automatic datatype coercions.

Furthermore, this part does not provide specifications for:

i) how numerical functions should be implemented,

j) 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 language standards and in language binding standards.

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

Conformity to this part is always with respect to a specified set of datatypes and set of operations. Conformity to this part implies conformity to part 1 and part 2 for the integer and floating point datatypes and operations used.

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 specifications of the binding standard takes precedence.

When a binding standard covers only a subset of the datatypes and operations defined in this part, an implementation remains free to conform to this part with respect to other datatypes or 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 datatypes and one or more operations that together satisfy all the requirements of clauses 5 through 8 that are relevant to those datatypes and operations. The implementation shall then document the binding.

An implementation is free to provide datatypes or 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 datatypes or 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.

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

3 It is not possible to conform to this part without specifying to which set of datatypes and set of operations conformity is claimed.

# 3 Normative references

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

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

ISO/IEC 10967-1:2002, *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic.*

ISO/IEC 10967-2:2000, *Information technology — Language independent arithmetic — Part 2: Elementary numerical functions.*

# 4 Symbols and definitions

## 4.1 Symbols

### 4.1.1 Sets and intervals

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

$[x, z]$ 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. The conventional notation for set definition and manipulation is also used. In particular this part uses

$\Rightarrow$ and $\Leftrightarrow$ for logical implication and equivalence
$+$, $-$, $/$, and $|x|$ on complex values
$\cdot$ for multiplication on complex values
$<$, $\leqslant$, $=$, $\neq$, $\geqslant$, and $>$ between reals
$\cup$, $\cap$, $\times$, $\in$, $\notin$, $\subset$, $\subseteq$, $\not\subseteq$, $\neq$, and $=$ with sets
$\times$ for the Cartesian product of sets
$\rightarrow$ for a mapping between sets

### 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* [43] ($e$ is the Napierian base):

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

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 $x \in [0, \infty[$.

### 4.1.4 Datatypes and exceptional values

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

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. The following symbols are defined in part 1 or part 2, and used in this part.

Exceptional values:
**overflow**, **underflow**, **invalid**, **infinitary**, and **absolute_precision_underflow**.
Integer operations:
$neg_I$, $add_I$, $sub_I$, and $mul_I$.
Floating point helper functions:
$e_F$, $u_F$, $result_F$, and $rnd_F$.
Floating point operations from part 1:
$sign_F$, $neg_F$, $add_F$, $sub_F$, $mul_F$, and $div_F$.
Floating point operations from part 2:
$sqrt_F$, $hypot_F$, $exp_F$, $power_F$, $ln_F$, $logbase_F$
$sinh_F$, $cosh_F$, $tanh_F$, $coth_F$, $sech_F$, $csch_F$,
$arcsinh_F$, $arccosh_F$, $arctanh_F$, $arccoth_F$, $arcsech_F$, $arccsch_F$,
$rad_F$, $sin_F$, $cos_F$, $tan_F$, $cot_F$, $sec_F$, $csc_F$,
$arcsin_F$, $arccos_F$, $arctan_F$, $arccot_F$, $arcsec_F$, $arccsc_F$, $arc_F$.

Floating point datatypes that conform to part 1 shall, for use with this part, have a value for the parameter $p_F$ such that $p_F \geqslant 2 \cdot \max\{1, \log_{r_F}(2 \cdot \pi)\}$, and have a value for the parameter $emin_F$ such that $emin_F \leqslant -p_F - 1$.

NOTES

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

2 These extra requirements, which do not limit the use of any existing floating point datatype, are made so that angles in radians are not too degenerate within the first two cycles, plus and minus, when represented in $F$.

3 $F$ should also be such that $p_F \geqslant 2 + \log_{r_F}(1000)$, to allow for a not too coarse angle resolution anywhere in the interval $[-big\_angle\_r_F, big\_angle\_r_F]$. See clause 5.3.9 of part 2.

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

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

These floating point values are not part of 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 4 – 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 **NaN**s 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 **+∞**, **−∞**, or **qNaN**, the actual result or continuation value is binding or implementation defined.

## 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 $\mathcal{Z}$, $\mathcal{G}$, $\mathcal{R}$, or $\mathcal{C}$.

**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 6.1.2 of part 1.)

**denormalisation loss:** A larger than normal rounding error caused by the fact that subnormal values have less than full precision. (See 5.2.5 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, or because the mathematical result cannot be represented with sufficient accuracy.

**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

1 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.

2 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.

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

**literal:** A syntactic entity denoting a constant value without having proper sub-entities that are expressions.

**monotonic approximation:** An floating point operation $op_F : ... \times F \times ... \to F$, for a floating point datatype with non-special value set $F$, where the other arguments are kept constant, is a monotonic approximation of a predetermined mathematical function $h : \mathcal{R} \to \mathcal{R}$ if, for every $a \in F$ and $b \in F$, $a < b$,

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

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

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

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

**normalised:** The non-zero values of a floating point type $F$ that provide the full precision allowed by that type. (See $F_N$ in 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 5.2.6 of part 1). (See also A.5.2.6 of part 1 for some examples.)

**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 $\mathcal{R}$,

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

Note that if $u \in \mathcal{R}$ 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.

**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.

The signature

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

states that the operation named $add_I$ shall accept any pair of $I$ values as input, and (when given such input) shall return either a single $I$ value as its output or the exceptional value **overflow**.

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 input. 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 5.2 of part 1 for a full definition.)

**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 $p$, and radix $r$, is $r^{t-p}$, and the ulp of a subnormal value is $fminD_F$. (See 5.2 of part 1.)

# 5 Specifications for complex datatypes and operations

This clause specifies Gussian integer (complex integer) datatypes, complex floating point datatypes and a number of helper functions and operations for complex integer and complex 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, in part 1, or in part 2), to mathematical functions, and to helper functions (specified in this part, in part 1, or in part 2). They also use special abstract values ($-\infty$, $+\infty$, $-\mathbf{0}$, **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 Complex integer datatypes and operations

Clause 5.1 of part 1 and clause 5.1 of part 2 specify integer datatypes and a number of operations on values of an integer datatype. In this clause complex integer (Gaussian integer) datatypes and operations on values of a complex integer datatype are specified.

A complex integer datatype is constructed from an integer datatype. For each integer datatype, there is one complex integer datatype.

$I$ 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.

$\mathrm{i}(I)$ is the set of non-special values in a imaginary integer datatype, constructed from the datatype corresponding to $I$.

$$\mathrm{i}(I) = \{\hat{\mathbf{i}} \cdot x' \mid x' \in I\}$$

$\mathrm{c}(I)$ is the set of non-special values in a complex integer datatype, constructed from the datatype corresponding to $I$.

$$\mathrm{c}(I) = \{x + \hat{\mathbf{i}} \cdot x' \mid x, x' \in I\}$$

### 5.1.1 The complex integer *result* helper function

The $result_{\mathrm{c}(I)}$ helper function:

$result_{\mathrm{c}(I)} : \mathcal{G} \to \mathrm{c}(I) \cup \{\textbf{overflow}\}$

$result_{\mathrm{c}(I)}(z) \quad = result_I(\mathfrak{Re}(z)) + \hat{\mathbf{i}} \cdot result_I(\mathfrak{Im}(z))$

NOTE – If one or both of the $result_I$ function applications on the right side returns **overflow**, then the $result_{\mathrm{c}(I)}$ application returns **overflow**. Similarly below for the specifications that do not use $result_{\mathrm{c}(I)}$ but specify the result parts directly.

### 5.1.2 Complex integer operations

$itimes_I : I \to \mathrm{i}(I) \cup \{\textbf{overflow}\}$

$itimes_I(x) \quad = \hat{\mathbf{i}} \cdot x$

$itimes_{\mathrm{c}(I)} : \mathrm{c}(I) \to \mathrm{c}(I) \cup \{\textbf{overflow}\}$

$itimes_{\mathrm{c}(I)}(x + \hat{\mathbf{i}} \cdot x') \quad = neg_I(x') + \hat{\mathbf{i}} \cdot x$

$re_I : I \to I$

$re_I(x) \quad = x$

$re_{\mathrm{i}(I)} : \mathrm{i}(I) \to I$

$re_{\mathrm{i}(I)}(\hat{\mathbf{i}} \cdot x') \quad = 0$

$re_{\mathrm{c}(I)} : \mathrm{c}(I) \to I$

$re_{\mathrm{c}(I)}(x + \hat{\mathbf{i}} \cdot x') \quad = x$

$im_I : I \to I$

$im_I(x) \quad = 0$

$im_{\mathrm{i}(I)} : \mathrm{i}(I) \to I$

$im_{\mathrm{i}(I)}(\hat{\mathbf{i}} \cdot x') \quad = x'$

$im_{\mathrm{c}(I)} : \mathrm{c}(I) \to I$

$im_{\mathrm{c}(I)}(x + \hat{\mathbf{i}} \cdot x') = x'$

$plusitimes_{\mathrm{c}(I)} : I \times I \to \mathrm{c}(I)$

$plusitimes_{\mathrm{c}(I)}(x, y) \quad = x + \hat{\mathbf{i}} \cdot y$

$neg_{\mathrm{i}(I)} : \mathrm{i}(I) \to \mathrm{i}(I) \cup \{\textbf{overflow}\}$

$neg_{\mathrm{i}(I)}(\hat{\mathbf{i}} \cdot x') \quad = \hat{\mathbf{i}} \cdot neg_I(x')$

$neg_{\mathrm{c}(I)} : \mathrm{c}(I) \to \mathrm{c}(I) \cup \{\textbf{overflow}\}$

$$neg_{\mathrm{c}(I)}(x + \hat{\mathrm{i}} \cdot x') = neg_I(x) + \hat{\mathrm{i}} \cdot neg_I(x')$$

$$conj_I : I \to I \cup \{\textbf{overflow}\}$$
$$conj_I(x) = x$$

$$conj_{\mathrm{i}(I)} : \mathrm{i}(I) \to \mathrm{i}(I) \cup \{\textbf{overflow}\}$$
$$conj_{\mathrm{i}(I)}(\hat{\mathrm{i}} \cdot x') = \hat{\mathrm{i}} \cdot neg_I(x')$$

$$conj_{\mathrm{c}(I)} : \mathrm{c}(I) \to \mathrm{c}(I) \cup \{\textbf{overflow}\}$$
$$conj_{\mathrm{c}(I)}(x + \hat{\mathrm{i}} \cdot x') = x + \hat{\mathrm{i}} \cdot neg_I(x')$$

$$add_{\mathrm{i}(I)} : \mathrm{i}(I) \times \mathrm{i}(I) \to \mathrm{i}(I) \cup \{\textbf{overflow}\}$$
$$add_{\mathrm{i}(I)}(\hat{\mathrm{i}} \cdot x', \hat{\mathrm{i}} \cdot y') = \hat{\mathrm{i}} \cdot add_I(x', y')$$

$$add_{\mathrm{c}(I)} : \mathrm{c}(I) \times \mathrm{c}(I) \to \mathrm{c}(I) \cup \{\textbf{overflow}\}$$
$$add_{\mathrm{c}(I)}(x + \hat{\mathrm{i}} \cdot x', y + \hat{\mathrm{i}} \cdot y') = add_F(x, y) + \hat{\mathrm{i}} \cdot add_F(x', y')$$

$$sub_{\mathrm{i}(I)} : \mathrm{i}(I) \times \mathrm{i}(I) \to \mathrm{i}(I) \cup \{\textbf{overflow}\}$$
$$sub_{\mathrm{i}(I)}(x, y) = add_{\mathrm{i}(I)}(x, neg_{\mathrm{i}(I)}(y))$$

$$sub_{\mathrm{c}(I)} : \mathrm{c}(I) \times \mathrm{c}(I) \to \mathrm{c}(I) \cup \{\textbf{overflow}\}$$
$$sub_{\mathrm{c}(I)}(x, y) = add_{\mathrm{c}(I)}(x, neg_{\mathrm{c}(I)}(y))$$

$$mul_{\mathrm{c}(I)} : \mathrm{c}(I) \times \mathrm{c}(I) \to \mathrm{c}(I) \cup \{\textbf{overflow}\}$$
$$mul_{\mathrm{c}(I)}(x + \hat{\mathrm{i}} \cdot x', y + \hat{\mathrm{i}} \cdot y') = result_{\mathrm{c}(I)}((x + \hat{\mathrm{i}} \cdot x') \cdot (y + \hat{\mathrm{i}} \cdot y')) \quad \text{if } x, y \in \mathrm{c}(I)$$

$$eq_{\mathrm{c}(I)} : \mathrm{c}(I) \times \mathrm{c}(I) \to \textbf{Boolean}$$
$$eq_{\mathrm{c}(I)}(x, y) = \textbf{true} \quad \text{if } x, y \in \mathrm{c}(I) \text{ and } x = y$$
$$= \textbf{false} \quad \text{if } x, y \in \mathrm{c}(I) \text{ and } x \neq y$$

$$neq_{\mathrm{c}(I)} : \mathrm{c}(I) \times \mathrm{c}(I) \to \textbf{Boolean}$$
$$neq_{\mathrm{c}(I)}(x, y) = \textbf{true} \quad \text{if } x, y \in \mathrm{c}(I) \text{ and } x \neq y$$
$$= \textbf{false} \quad \text{if } x, y \in \mathrm{c}(I) \text{ and } x = y$$

## 5.2 Complex floating point datatypes and operations

Clause 5.2 of part 1 and clause 5.2 of part 2 specify floating point datatypes and a number of operations on values of a floating point datatype. In this clause complex floating point datatypes and operations on values of a complex floating point datatype are specified.

NOTE – Further operations on values of a complex floating point datatype, for elementary complex 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.

$\mathrm{i}(F)$ is the set of non-special values in a imaginary floating point datatype, constructed from the datatype corresponding to $F$.

$$\mathrm{i}(F) = \{\hat{\mathbf{i}} \cdot y \mid y \in F\}$$

$\mathrm{c}(F)$ is the set of non-special values in a complex floating point datatype, constructed from the datatype corresponding to $F$.

$$\mathrm{c}(F) = \{x + \hat{\mathbf{i}} \cdot x' \mid x, x' \in F\}$$

### 5.2.1 The complex floating point *result* helper functions

$$result^*_{\mathrm{c}(F)} : \mathcal{C} \times (\mathcal{R} \to F^*) \to F \cup \{\mathbf{underflow}, \mathbf{overflow}\}$$

$$result^*_{\mathrm{c}(F)}(z, rnd) = result^*_F(\mathfrak{Re}(z), rnd) + \hat{\mathbf{i}} \cdot result^*_F(\mathfrak{Im}(z), rnd)$$

overflow, underflow, ... combination...!, (inexact)????

$result^*_F$ is defined in part 2.

Define the $no\_result_{\mathrm{c}(F)}$ and $no\_result2_{\mathrm{c}(F)}$ helper functions:

$$no\_result_{\mathrm{c}(F)} : \mathrm{c}(F) \to \{\mathbf{invalid}\}$$

$$no\_result_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x') =$$

$= \mathbf{qNaN} + \hat{\mathbf{i}} \cdot \mathbf{qNaN}$ if $x$ is a quiet NaN and $x'$ is not a signalling NaN

$= \mathbf{qNaN} + \hat{\mathbf{i}} \cdot \mathbf{qNaN}$ if $x'$ is a quiet NaN and $x$ is not a signalling NaN

$= \mathbf{invalid}(\mathbf{qNaN} + \hat{\mathbf{i}} \cdot \mathbf{qNaN})$ if $x, x' \in F \cup \{-\infty, -\mathbf{0}, +\infty\}$

$= \mathbf{invalid}(\mathbf{qNaN} + \hat{\mathbf{i}} \cdot \mathbf{qNaN})$ if $x$ is a signalling NaN or $x'$ is a signalling NaN

$$no\_result2_{\mathrm{c}(F)} : \mathrm{c}(F) \times \mathrm{c}(F) \to \{\mathbf{invalid}\}$$

$$no\_result2_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x', y + \hat{\mathbf{i}} \cdot y') =$$

$= \mathbf{qNaN} + \hat{\mathbf{i}} \cdot \mathbf{qNaN}$ if $x$ is a quiet NaN and neither $x'$, $y$, nor $y'$ is a signalling NaN

$= \mathbf{qNaN} + \hat{\mathbf{i}} \cdot \mathbf{qNaN}$ if $x'$ is a quiet NaN and neither $x$, $y$, nor $y'$ is not a signalling NaN

$= \mathbf{qNaN} + \hat{\mathbf{i}} \cdot \mathbf{qNaN}$ if $y$ is a quiet NaN and neither $x$, $x'$, nor $y'$ is not a signalling NaN

$= \mathbf{qNaN} + \hat{\mathbf{i}} \cdot \mathbf{qNaN}$ if $y'$ is a quiet NaN and neither $x$, $x'$, nor $y$ is not a signalling NaN

$= \mathbf{invalid}(\mathbf{qNaN} + \hat{\mathbf{i}} \cdot \mathbf{qNaN})$ if $x, x', y, y' \in F \cup \{-\infty, -\mathbf{0}, +\infty\}$

$= \mathbf{invalid}(\mathbf{qNaN} + \hat{\mathbf{i}} \cdot \mathbf{qNaN})$ if at least one of $x$, $x'$, $y$, or $y'$ is a signalling NaN

These helper functions are used to specify both NaN argument handling and to handle non-NaN-argument cases where $\mathbf{invalid}(\mathbf{qNaN} + \hat{\mathbf{i}} \cdot \mathbf{qNaN})$ is the appropriate result.

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

### 5.2.2 Basic arithmetic for complex floating point

$itimes_F : F \to \mathrm{i}(F) \cup \{\mathbf{overflow}\}$
$itimes_F(x) \quad = \hat{\mathbf{i}} \cdot x$

$itimes_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{overflow}\}$
$itimes_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$
$\quad = neg_F(x') + \hat{\mathbf{i}} \cdot x$

$re_F : F \to F$
$re_F(x) \quad = x$

$re_{\mathrm{i}(F)} : \mathrm{i}(F) \to F$
$re_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x') \quad = -\mathbf{0}$

$re_{\mathrm{c}(F)} : \mathrm{c}(F) \to F$
$re_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x') = x$

$im_F : F \to F$
$im_F(x) \quad = -\mathbf{0}$

$im_{\mathrm{i}(F)} : \mathrm{i}(F) \to F$
$im_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x') \quad = x'$

$im_{\mathrm{c}(F)} : \mathrm{c}(F) \to F$
$im_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$
$\quad = x'$

$plusitimes_{\mathrm{c}(F)} : F \times F \to \mathrm{c}(F)$
$plusitimes_{\mathrm{c}(F)}(x, y)$
$\quad = x + \hat{\mathbf{i}} \cdot y$

$neg_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F)$
$neg_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x') \quad = \hat{\mathbf{i}} \cdot neg_F(x')$

$neg_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F)$
$neg_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$
$\quad = neg_F(x) + \hat{\mathbf{i}} \cdot neg_F(x')$

$conj_F : F \to F$

$conj_F(x) \quad = x$

$conj_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F)$
$conj_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x') \quad = \hat{\mathbf{i}} \cdot neg_F(x')$

$conj_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F)$
$conj_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$
$\quad = x + \hat{\mathbf{i}} \cdot neg_F(x')$

$add_{\mathrm{i}(F)} : \mathrm{i}(F) \times \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{overflow}\}$
$add_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x', \hat{\mathbf{i}} \cdot y')$
$\quad = \hat{\mathbf{i}} \cdot add_F(x', y')$

$add_{\mathrm{c}(F)} : \mathrm{c}(F) \times \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{overflow}\}$
$add_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x', y + \hat{\mathbf{i}} \cdot y')$
$\quad = add_F(x, x') + \hat{\mathbf{i}} \cdot add_F(y, y')$

$sub_{\mathrm{i}(F)} : \mathrm{i}(F) \times \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{overflow}\}$
$sub_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x', \hat{\mathbf{i}} \cdot y')$
$\quad = \hat{\mathbf{i}} \cdot sub_F(x', y')$

$sub_{\mathrm{c}(F)} : \mathrm{c}(F) \times \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{overflow}\}$
$sub_{\mathrm{c}(F)}(x, y) \quad = add_{\mathrm{c}(F)}(x, neg_{\mathrm{c}(F)}(y))$

$mul_{\mathrm{i}(F)} : \mathrm{i}(F) \times \mathrm{i}(F) \to F \cup \{\mathbf{overflow}\}$
$mul_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x, \hat{\mathbf{i}} \cdot x')$
$\quad = neg_F(mul_F(x, x'))$
NOTE 1 – $mul_{\mathrm{c}(F)}$ is specified in clause 5.2.3

$div_{\mathrm{i}(F)} : \mathrm{i}(F) \times \mathrm{i}(F) \to F \cup \{\mathbf{overflow}\}$
$div_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x, \hat{\mathbf{i}} \cdot x')$
$\quad = div_F(x, x')$
NOTE 2 – $div_{\mathrm{c}(F)}$ is specified in clause 5.2.3

$eq_{\mathrm{c}(F)} : \mathrm{c}(F) \times \mathrm{c}(F) \to \mathbf{Boolean}$
$eq_{\mathrm{c}(F)}(x, y) \quad = \mathbf{true} \quad$ if $x, y \in \mathrm{c}(F)$ and $x = y$
$\quad = \mathbf{false} \quad$ if $x, y \in \mathrm{c}(F)$ and $x \neq y$

$neq_{\mathrm{c}(F)} : \mathrm{c}(F) \times \mathrm{c}(F) \to \mathbf{Boolean}$
$neq_{\mathrm{c}(F)}(x, y) \quad = \mathbf{true} \quad$ if $x, y \in \mathrm{c}(F)$ and $x \neq y$
$\quad = \mathbf{false} \quad$ if $x, y \in \mathrm{c}(F)$ and $x = y$

$abs_{\mathrm{i}(F)} : \mathrm{i}(F) \to F$
$abs_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x') \quad = abs_F(x')$

$abs_{\mathrm{c}(F)} : \mathrm{c}(F) \to F \cup \{\mathbf{underflow}, \mathbf{overflow}\}$
$abs_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')$
$\quad = hypot_F(x, x')$

$phase_F : F \to F$
$phase_F(x) \quad = arc_F(x, -\mathbf{0})$

$phase_{\mathrm{i}(F)} : \mathrm{i}(F) \to F$
$phase_{\mathrm{i}(F)}(\hat{\imath} \cdot x') \; = arc_F(-\mathbf{0}, x')$

$phase_{\mathrm{c}(F)} : \mathrm{c}(F) \to F \cup \{\mathbf{underflow}\}$
$phase_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')$
$\quad = arc_F(x, x')$

$sign_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F)$
$sign_{\mathrm{i}(F)}(\hat{\imath} \cdot x') \quad = \hat{\imath} \cdot sign_F(x')$

$sign_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}\}$
$sign_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')$
$\quad = sin_F(arc_F(x, x')) + \hat{\imath} \cdot cos_F(arc_F(x, x')$

### 5.2.3 Complex multiplication and division

There shall be two maximum error parameters for complex multiplication and division.

$max\_error\_mul_{\mathrm{c}(F)} \in F$
$max\_error\_div_{\mathrm{c}(F)} \in F$

no monotonicity requirements? no sign requirements? dependency on argument values?

The $mul^*_{\mathrm{c}(F)}$ approximation helper function:

$mul^*_{\mathrm{c}(F)} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$

$mul^*_{\mathrm{c}(F)}(x, y)$ returns a close approximation to $x \cdot y$ in $\mathcal{C}$ with maximum error $max\_error\_mul_{\mathrm{c}(F)}$.

Further requirement on the $mul^*_{\mathrm{c}(F)}$ approximation helper function are:

$mul^*_{\mathrm{c}(F)}(z, z') = mul^*_{\mathrm{c}(F)}(z', z)$ if $z, z' \in \mathcal{C}$
$mul^*_{\mathrm{c}(F)}(-z, z') = -mul^*_{\mathrm{c}(F)}(z, z')$ if $z, z' \in \mathcal{C}$
$mul^*_{\mathrm{c}(F)}(\mathrm{conj}(z), \mathrm{conj}(z')) = \mathrm{conj}(mul^*_{\mathrm{c}(F)}(z, z'))$
if $z, z' \in \mathcal{C}$

The $mul_{\mathrm{c}(F)}$ operation:

$mul_{\mathrm{c}(F)} : \mathrm{c}(F) \times \mathrm{c}(F) \to \mathrm{c}(F) \cup \{-\mathbf{0}..., \mathbf{underflow}, \mathbf{overflow}\}$
$mul_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y')$
$\quad = result^*_{\mathrm{c}(F)}(mul^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y'))$
if $x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y' \in \mathrm{c}(F)$ and $x + \hat{\imath} \cdot x' \neq 0$ and $y + \hat{\imath} \cdot y' \neq 0$
$\quad = sub_F(mul_F(x, y), mul_F(x', y')) + \hat{\imath} \cdot add_F(mul_F(x, y'), mul_F(x', y))$
otherwise

The $div^*_{\mathrm{c}(F)}$ approximation helper function:

$div^*_{\mathrm{c}(F)} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$

$div^*_{\mathrm{c}(F)}(x, y)$ returns a close approximation to $x/y$ in $\mathcal{C}$ with maximum error $max\_error\_div_{\mathrm{c}(F)}$.

Further requirement on the $div^*_{\mathrm{c}(F)}$ approximation helper function are:

$div^*_{\mathrm{c}(F)}(-z, z') = -div^*_{\mathrm{c}(F)}(z, z')$ if $z, z' \in \mathcal{C}$ and $z' \neq 0$
$div^*_{\mathrm{c}(F)}(z, -z') = -div^*_{\mathrm{c}(F)}(z, z')$ if $z, z' \in \mathcal{C}$ and $z' \neq 0$
$div^*_{\mathrm{c}(F)}(\mathrm{conj}(z), \mathrm{conj}(z')) = \mathrm{conj}(div^*_{\mathrm{c}(F)}(z, z'))$
if $z, z' \in \mathcal{C}$ and $z' \neq 0$

The $div_{\mathrm{c}(F)}$ operation:

$div_{\mathrm{c}(F)} : \mathrm{c}(F) \times \mathrm{c}(F) \to \mathrm{c}(F) \cup \{-\mathbf{0}..., \mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{invalid}\}$

$div_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y')$
$= result^*_{\mathrm{c}(F)}(div^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y'))$ if $x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y' \in \mathrm{c}(F)$ and $|y| \neq |y'|$
$=???$ otherwise

## 5.3 Elementary transcendental complex floating point operations

### 5.3.1 Operations for exponentiations and logarithms

There shall be two maximum error parameters for complex exponentiations and logarithms.

$max\_error\_exp_{\mathrm{c}(F)} \in F$
$max\_error\_power_{\mathrm{c}(F)} \in F$

no monotonicity requirements? no sign requirements? dependency on argument values?

#### 5.3.1.1 Natural exponentiation

The $exp^*_{\mathrm{c}(F)}$ approximation helper function:

$exp^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$exp^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $e^z$ in $\mathcal{C}$ with maximum error $max\_error\_exp_{\mathrm{c}(F)}$.

A further requirement on the $exp^*_{\mathrm{c}(F)}$ approximation helper function is:

$exp^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(exp^*_{\mathrm{c}(F)}(z))$ if $z \in \mathcal{C}$

The relationship to the $cos^*_F$, $sin^*_F$, and $exp^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$exp^*_{\mathrm{c}(F)}(\hat{\imath} \cdot x') = cos^*_F(x') + \hat{\imath} \cdot sin^*_F(x')$ if $x' \in \mathcal{R}$
$exp^*_{\mathrm{c}(F)}(x) = exp^*_F(x)$ if $x \in \mathcal{R}$
...cyclic rep. in general...

The $exp_{\mathrm{c}(F)}$ operation:

$exp_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{absolute\_precision\_underflow}\}$

$exp_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')$
$= result^*_{\mathrm{c}(F)}(exp^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x'), nearest_F)$ if $x + \hat{\imath} \cdot x' \in \mathrm{c}(F)$ and $|x'| \leqslant big\_angle\_r_F$
$= exp_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot x')$ if $x = -\mathbf{0}$
$= conj_{\mathrm{c}(F)}(exp_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0))$ if $x' = -\mathbf{0}$ and $x \neq -\mathbf{0}$
$= mul_F(0, cos_F(x')) + \hat{\imath} \cdot mul_F(0, sin_F(x'))$

$$
\begin{array}{ll}
 & \text{if } x = -\infty \text{ and } x' \in F \text{ and } |x'| \leqslant big\_angle\_r_F \\
= mul_F(+\infty, cos_F(x')) + \hat{\imath} \cdot mul_F(+\infty, sin_F(x')) & \\
 & \text{if } x = +\infty \text{ and } x' \in F \text{ and } |x'| \leqslant big\_angle\_r_F \text{ and } x' \neq 0 \\
= +\infty + \hat{\imath} \cdot 0 & \text{if } x = +\infty \text{ and } x' \in F \text{ and } x' = 0 \\
= radh_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') & \text{otherwise}
\end{array}
$$

NOTE – $radh_{\mathrm{c}(F)}$ is specified in clause 5.3.3.1.

#### 5.3.1.2 Complex exponentiation of argument base

The $power^*_{\mathrm{c}(F)}$ approximation helper function:

$power^*_{\mathrm{c}(F)} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$

$power^*_{\mathrm{c}(F)}(b, z)$ returns a close approximation to $b^z$ in $\mathcal{C}$ with maximum error $max\_error\_power_{\mathrm{c}(F)}$.

A further requirement on the $power^*_{\mathrm{c}(F)}$ approximation helper function is:

$power^*_{\mathrm{c}(F)}(\mathrm{conj}(b), \mathrm{conj}(z)) = \mathrm{conj}(power^*_{\mathrm{c}(F)}(b, z))$ if $b, z \in \mathcal{C}$

The $power_{\mathrm{c}(F)}$ operation:

$power_{\mathrm{c}(F)} : \mathrm{c}(F) \times \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\textbf{underflow}, \textbf{overflow}, \textbf{absolute\_precision\_underflow}, \textbf{invalid}\}$

$power_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y')$

$$
\begin{array}{ll}
= result^*_{\mathrm{c}(F)}(power^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y'), nearest_F) & \\
 & \text{if } x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y' \in \mathrm{c}(F) \text{ and } x \neq 0 \text{ and} \\
 & \quad |y' \cdot \ln(|x|)| \text{ is not too large...?} \\
= power_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y') & \\
 & \text{if } x = -\mathbf{0} \text{ (?)} \\
= ?????? & \text{if } x' = -\mathbf{0} \text{ and } x \neq -\mathbf{0} \\
= power_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x', 0 + \hat{\imath} \cdot y') & \\
 & \text{if } y = -\mathbf{0} \\
= conj_{\mathrm{c}(F)}(power_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot 0)) & \\
 & \text{if } y' = -\mathbf{0} \text{ (?)} \\
= exp_{\mathrm{c}(F)}(mul_{\mathrm{c}(F)}(ln_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x'), y + \hat{\imath} \cdot y')) & \\
 & \text{otherwise}
\end{array}
$$

NOTE – Complex raising to a power is multi-valued. The principal result is given by $b^q = e^{q \cdot \ln(b)}$. The $b^q$ function branch cuts at $\{x \mid x \in \mathcal{R} \text{ and } x < 0\} \times \mathcal{C}$ (except when $q$ is in $\mathcal{Z}$). Thus $power_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0, y) \neq power_{\mathrm{c}(F)}(x + \hat{\imath} \cdot (-\mathbf{0}), y)$.

#### 5.3.1.3 Complex square root

The $sqrt^*_{\mathrm{c}(F)}$ approximation helper function:

$sqrt^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$sqrt^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\sqrt{z}$ in $\mathcal{C}$ with maximum error $max\_error\_exp_{\mathrm{c}(F)}$.

Further requirements on the $sqrt^*_{\mathrm{c}(F)}$ approximation helper function are:

$$
\begin{array}{ll}
sqrt^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(sqrt^*_{\mathrm{c}(F)}(z)) & \text{if } z \in \mathcal{C} \\
sqrt^*_{\mathrm{c}(F)}(x) = \sqrt{x} & \text{if } x \in \mathcal{R} \text{ and } x \geqslant 0 \\
sqrt^*_{\mathrm{c}(F)}(x) = \hat{\imath} \cdot sqrt^*_{\mathrm{c}(F)}(-x) & \text{if } x \in \mathcal{R} \text{ and } x < 0 \\
\mathfrak{Re}(sqrt^*_{\mathrm{c}(F)}(\hat{\imath} \cdot x')) = \mathfrak{Im}(sqrt^*_{\mathrm{c}(F)}(\hat{\imath} \cdot x')) & \text{if } x' \in \mathcal{R} \text{ and } x' \geqslant 0
\end{array}
$$

The $sqrt_{\mathrm{c}(F)}$ operation:

$sqrt_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F)$

$sqrt_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')$

$= result^*_{\mathrm{c}(F)}(sqrt^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x'), nearest_F)$ if $x + \hat{\imath} \cdot x' \in \mathrm{c}(F)$

$= sqrt_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot x')$ if $x = -\mathbf{0}$ and $x' \in F \cup \{-\infty, +\infty\}$

$= -\mathbf{0} + \hat{\imath} \cdot (-\mathbf{0})$ if $x = negz$ and $x' = -\mathbf{0}$

$= conj_{\mathrm{c}(F)}(sqrt_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0))$ if $x \in F \cup \{-\infty, +\infty\}$ and $x' = -\mathbf{0}$

$= +\infty + \hat{\imath} \cdot (+\infty)$ if $x \in F \cup \{-\infty, -\mathbf{0}, +\infty\}$ and $x' = +\infty$

$= +\infty + \hat{\imath} \cdot 0$ if $x = +\infty$ and $x' \in F$ and $x \geqslant 0$

$= +\infty + \hat{\imath} \cdot (-\mathbf{0})$ if $x = +\infty$ and $((x' \in F$ and $x' < 0$ or $x' = -\mathbf{0})$

$= +\infty + \hat{\imath} \cdot (-\infty)$ if $x \in F \cup \{-\infty, -\mathbf{0}, +\infty\}$ and $x' = -\infty$

$= 0 + \hat{\imath} \cdot (+\infty)$ if $x = -\infty$ and $x' \in F$ and $x \geqslant 0$

$= 0 + \hat{\imath} \cdot (-\infty)$ if $x = -\infty$ and $((x' \in F$ and $x' < 0$ or $x' = -\mathbf{0})$

$= no\_result_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')$ otherwise

NOTE – The inverse of complex square is multi-valued. The principal result is given by $\sqrt{b} = e^{0.5 \cdot \ln(b)}$. The $\sqrt{\ }$ function branch cuts at $\{x \mid x \in \mathcal{R}$ and $x < 0\}$. Thus $sqrt_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0) \neq sqrt_{\mathrm{c}(F)}(x + \hat{\imath} \cdot (-\mathbf{0}))$ when $x < 0$.

#### 5.3.1.4 Natural logarithm

The $ln^*_{\mathrm{c}(F)}$ approximation helper function:

$ln^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$ln^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\ln(z)$ in $\mathcal{C}$ with maximum error $max\_error\_exp_{\mathrm{c}(F)}$.

A further requirement on the $ln^*_{\mathrm{c}(F)}$ approximation helper function is:

$ln^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(ln^*_{\mathrm{c}(F)}(z))$ if $z \in \mathcal{C}$

The relationship to the $arc^*_F$ and $ln^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$\mathfrak{Im}(ln^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')) = arc^*_F(x, x')$ if $x, x' \in \mathcal{R}$

$\mathfrak{Re}(ln^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')) = ln^*_F(|x + \hat{\imath} \cdot x'|)$ if $x, x' \in \mathcal{R}$ and $(x = 0$ or $x' = 0)$

The $ln_{\mathrm{c}(F)}$ operation:

$ln_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{infinitary}\}$

$ln_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') = result^*_{\mathrm{c}(F)}(ln^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x'), nearest_F)$ if $x + \hat{\imath} \cdot x' \in \mathrm{c}(F)$ and $(x \neq 0$ or $x' \neq 0)$

$= \mathbf{infinitary}(-\infty + \hat{\imath} \cdot arc_F(x, x'))$ if $x, x' \in \{-\mathbf{0}, 0\}$

$= conj_{\mathrm{c}(F)}(ln_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0))$ if $x' = -\mathbf{0}$

$= ln_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot x')$ if $x = -\mathbf{0}$ and $x \in F$ and $x' \neq 0$

$= +\infty + \hat{\imath} \cdot arc_F(x, x')$ if $x \in \{-\infty, +\infty\}$ and $x' \in F \cup \{-\infty, +\infty\}$

$= +\infty + \hat{\imath} \cdot arc_F(x, x')$ if $x \in F$ and $x' \in \{-\infty, +\infty\}$

$= no\_result_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')$ otherwise

NOTES

1 The inverse of natural exponentiation is multi-valued: the imaginary part may have any integer multiple of $2 \cdot \pi$ added to it, and the result is also in the solution set. The ln function (returning the principle value for the inverse) branch cuts at $\{x \mid x \in \mathcal{R} \text{ and } x < 0\}$, is continuous on the rest of $\mathcal{C}$, and $\ln(z) \in \mathcal{R}$ if $x \in \mathcal{R}$ and $x > 0$. Thus $ln_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0) \neq ln_{\mathrm{c}(F)}(x + \hat{\imath} \cdot (-\mathbf{0}))$ when $x < 0$.

2 $re_{\mathrm{c}(F)}(ln_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')) \approx ln_F(hypot_F(x, x'))$ and
$im_{\mathrm{c}(F)}(ln_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')) = arc_F(x, x')$ when there is no notification.

#### 5.3.1.5 Argument base logarithm

The $logbase^*_{\mathrm{c}(F)}$ approximation helper function:

$$logbase^*_{\mathrm{c}(F)} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$$

$logbase^*_{\mathrm{c}(F)}(b, z)$ returns a close approximation to $\log_b(z)$ in $\mathcal{C}$ with maximum error $max\_error\_power_{\mathrm{c}(F)}$.

A further requirement on the $logbase^*_{\mathrm{c}(F)}$ approximation helper function is:

$$logbase^*_{\mathrm{c}(F)}(\mathrm{conj}(b), \mathrm{conj}(z)) = \mathrm{conj}(logbase^*_{\mathrm{c}(F)}(b, z)) \quad \text{if } b, z \in \mathcal{C}$$

The $logbase_{\mathrm{c}(F)}$ operation:

$$logbase_{\mathrm{c}(F)} : \mathrm{c}(F) \times \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{infinitary}, \mathbf{invalid}\}$$

$$\begin{aligned} &logbase_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y') \\ &\quad = result^*_{\mathrm{c}(F)}(logbase^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y'), nearest_F) \\ &\qquad \text{if } x + \hat{\imath} \cdot x', y + \hat{\imath} \cdot y' \in \mathrm{c}(F) \text{ and } x \neq 0 \text{ and} \\ &\qquad\quad |\mathfrak{Re}(ln(x + \hat{\imath} \cdot x'))| \neq |\mathfrak{Im}(ln(x + \hat{\imath} \cdot x'))| \\ &\quad = div_{\mathrm{c}(F)}(ln_{\mathrm{c}(F)}(y + \hat{\imath} \cdot y'), ln_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')) \\ &\qquad \text{otherwise} \end{aligned}$$

NOTE – Complex logarithm with argument base is multi-valued. The principal result is given by $\log_b(q) = \ln(q)/\ln(b)$. Apart from the poles, the $\log_b(q)$ function branch cuts at $(\{x \mid x \in \mathcal{R} \text{ and } x < 0\} \times \mathcal{C}) \cup (\mathcal{C} \times \{x \mid x \in \mathcal{R} \text{ and } x < 0\})$.

### 5.3.2 Operations for radian trigonometric elementary functions

There shall be two maximum error parameters for complex trigonometric operations.

$$max\_error\_sin_{\mathrm{c}(F)} \in F$$
$$max\_error\_tan_{\mathrm{c}(F)} \in F$$

no monotonicity requirements? no sign requirements? dependency on argument values?

#### 5.3.2.1 Radian angle normalisation

$$rad_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{absolute\_precision\_underflow}\}$$
$$rad_{\mathrm{i}(F)}(\hat{\imath} \cdot x') = \hat{\imath} \cdot x'$$

$$rad_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{absolute\_precision\_underflow}\}$$

$$\begin{aligned} &rad_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \\ &\quad = rad_F(x) + \hat{\imath} \cdot x' \qquad \text{if } x' \in F \cup \{-\infty, -\mathbf{0}, +\infty\} \text{ and } rad_F(x) \in F \cup \{-\mathbf{0}\} \\ &\quad = \mathbf{absolute\_precision\_underflow}(\mathbf{qNaN} + \hat{\imath} \cdot \mathbf{qNaN}) \\ &\qquad \text{if } x' \in F \cup \{-\infty, -\mathbf{0}, +\infty\} \text{ and} \\ &\qquad\quad rad_F(x) = \mathbf{absolute\_precision\_underflow} \\ &\quad = no\_result_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \text{ otherwise} \end{aligned}$$

#### 5.3.2.2 Radian sine

The $sin_{\mathrm{i}(F)}$ operation:

$sin_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{overflow}\}$

$sin_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x') \quad = \hat{\mathbf{i}} \cdot sinh_F(x')$

The $sin^*_{\mathrm{c}(F)}$ approximation helper function:

$sin^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$sin^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\sin(z)$ in $\mathcal{C}$ with maximum error $max\_error\_sin_{\mathrm{c}(F)}$.

Further requirements on the $sin^*_{\mathrm{c}(F)}$ approximation helper function are:

$sin^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(sin^*_{\mathrm{c}(F)}(z)) \qquad$ if $z \in \mathcal{C}$

$sin^*_{\mathrm{c}(F)}(-z) = -sin^*_{\mathrm{c}(F)}(z) \qquad$ if $z \in \mathcal{C}$

The relationship to the $sin^*_F$ and $sinh^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$sin^*_{\mathrm{c}(F)}(x) = sin^*_F(x) \qquad$ if $x \in \mathcal{R}$

$sin^*_{\mathrm{c}(F)}(\hat{\mathrm{i}} \cdot x') = \hat{\mathrm{i}} \cdot sinh_F(x') \qquad$ if $x' \in \mathcal{R}$

The requirements implied by these relationships and the requirements from part 2 (GENERALISE?) remain even if there is no $sin_F$ or $sinh_F$ operations in any associated library for real-valued operations or there is no associated library for real-valued operations.

The $sin_{\mathrm{c}(F)}$ operation:

$sin_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{absolute\_precision\_underflow}\}$

$sin_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$

$= result^*_{\mathrm{c}(F)}(sin^*_{\mathrm{c}(F)}(x + \hat{\mathrm{i}} \cdot x'), nearest_F)$ if $x + \hat{\mathbf{i}} \cdot x' \in \mathrm{c}(F)$ and $|x| \leqslant big\_angle\_r_F$

$= conj_{\mathrm{c}(F)}(sin_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot 0))$ if $x' = -\mathbf{0}$

$= neg_{\mathrm{c}(F)}(sin_{\mathrm{c}(F)}(0 + \hat{\mathbf{i}} \cdot neg_F(x')))$ if $x = -\mathbf{0}$ and $x' \neq -\mathbf{0}$

$= 0 + \hat{\mathbf{i}} \cdot (+\infty)$ if $x = 0$ and $x' = +\infty$

$= 0 + \hat{\mathbf{i}} \cdot (-\infty)$ if $x = 0$ and $x' = -\infty$

$= mul_F(+\infty, sin_F(x)) + \hat{\mathbf{i}} \cdot mul_F(+\infty, cos_F(x))$ if $x' = +\infty$ and $x \notin \{-\mathbf{0}, 0\}$

$= mul_F(+\infty, sin_F(x)) + \hat{\mathbf{i}} \cdot mul_F(-\infty, cos_F(x))$ if $x' = -\infty$ and $x \notin \{-\mathbf{0}, 0\}$

$= rad_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$ otherwise

#### 5.3.2.3 Radian cosine

The $cos_{\mathrm{i}(F)}$ operation:

$cos_{\mathrm{i}(F)} : \mathrm{i}(F) \to F \cup \{\mathbf{overflow}\}$

$cos_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x') \quad = cosh_F(x')$

The $cos^*_{\mathrm{c}(F)}$ approximation helper function:

$cos^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$cos^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\cos(z)$ in $\mathcal{C}$ with maximum error $max\_error\_sin_{\mathrm{c}(F)}$.

Further requirements on the $cos^*_{\mathrm{c}(F)}$ approximation helper function are:

$cos^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(cos^*_{\mathrm{c}(F)}(z))$ if $z \in \mathcal{C}$
$cos^*_{\mathrm{c}(F)}(-z) = cos^*_{\mathrm{c}(F)}(z)$ if $z \in \mathcal{C}$

The relationship to the $cos^*_F$ and $cosh^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$cos^*_{\mathrm{c}(F)}(x) = cos^*_F(x)$ if $x \in \mathcal{R}$
$cos^*_{\mathrm{c}(F)}(\hat{\imath} \cdot x') = cosh_F(x')$ if $x' \in \mathcal{R}$

The $cos_{\mathrm{c}(F)}$ operation:

$cos_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{absolute\_precision\_underflow}\}$

$$
\begin{aligned}
&cos_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \\
&\quad = result^*_{\mathrm{c}(F)}(cos^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x'), nearest_F) && \text{if } x + \hat{\imath} \cdot x' \in \mathrm{c}(F) \text{ and } |x| \leqslant big\_angle\_r_F \\
&\quad = conj_{\mathrm{c}(F)}(cos_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0)) && \text{if } x' = -\mathbf{0} \\
&\quad = cos_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot neg_F(x')) && \text{if } x = -\mathbf{0} \text{ and } x' \neq -\mathbf{0} \\
&\quad = +\boldsymbol{\infty} + \hat{\imath} \cdot (-\mathbf{0}) && \text{if } x' = +\boldsymbol{\infty} \text{ and } x = 0 \\
&\quad = +\boldsymbol{\infty} + \hat{\imath} \cdot 0 && \text{if } x' = -\boldsymbol{\infty} \text{ and } x = 0 \\
&\quad = mul_F(+\boldsymbol{\infty}, cos_F(x)) + \hat{\imath} \cdot mul_F(-\boldsymbol{\infty}, sin_F(x)) && \text{if } x' = +\boldsymbol{\infty} \text{ and } x \notin \{-\mathbf{0}, 0\} \\
&\quad = mul_F(+\boldsymbol{\infty}, cos_F(x)) + \hat{\imath} \cdot mul_F(+\boldsymbol{\infty}, sin_F(x)) && \text{if } x' = -\boldsymbol{\infty} \text{ and } x \notin \{-\mathbf{0}, 0\} \\
&\quad = rad_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') && \text{otherwise}
\end{aligned}
$$

#### 5.3.2.4 Radian tangent

The $tan_{\mathrm{i}(F)}$ operation:

$tan_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{overflow}\}$
$tan_{\mathrm{i}(F)}(\hat{\imath} \cdot x') = \hat{\imath} \cdot tanh_F(x')$

The $tan^*_{\mathrm{c}(F)}$ approximation helper function:

$tan^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$tan^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\tan(z)$ in $\mathcal{C}$ with maximum error $max\_error\_tan_{\mathrm{c}(F)}$.

Further requirements on the $tan^*_{\mathrm{c}(F)}$ approximation helper function are:

$tan^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(tan^*_{\mathrm{c}(F)}(z))$ if $z \in \mathcal{C}$
$tan^*_{\mathrm{c}(F)}(-z) = -tan^*_{\mathrm{c}(F)}(z)$ if $z \in \mathcal{C}$

The relationship to the $tan^*_F$ and $tanh^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$tan^*_{\mathrm{c}(F)}(x) = tan^*_F(x)$ if $x \in \mathcal{R}$
$tan^*_{\mathrm{c}(F)}(\hat{\imath} \cdot x') = \hat{\imath} \cdot tanh_F(x')$ if $x' \in \mathcal{R}$

The $tan_{\mathrm{c}(F)}$ operation:

$tan_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{absolute\_precision\_underflow}\}$

$$
\begin{aligned}
&tan_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \\
&\quad = result^*_{\mathrm{c}(F)}(tan^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x'), nearest_F) && \text{if } x + \hat{\imath} \cdot x' \in \mathrm{c}(F) \text{ and } |x| \leqslant big\_angle\_r_F \\
&\quad = conj_{\mathrm{c}(F)}(tan_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0)) && \text{if } x' = -\mathbf{0}
\end{aligned}
$$

$$
\begin{aligned}
&= neg_F(tan_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot neg_F(x'))) \\
&\qquad\qquad \text{if } x = -\mathbf{0} \text{ and } x' \neq -\mathbf{0} \\
&= mul_F(0, tan_F(x)) + \hat{\imath} \cdot 1 \\
&\qquad\qquad \text{if } x' = +\infty \\
&= mul_F(0, tan_F(x)) + \hat{\imath} \cdot (-1) \\
&\qquad\qquad \text{if } x' = -\infty \\
&= rad_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \qquad \text{otherwise}
\end{aligned}
$$

#### 5.3.2.5 Radian cotangent

The $cot_{\mathrm{i}(F)}$ operation:

$cot_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$cot_{\mathrm{i}(F)}(\hat{\imath} \cdot x') \quad = neg_{\mathrm{i}(F)}(\hat{\imath} \cdot coth_F(x'))$

The $cot^*_{\mathrm{c}(F)}$ approximation helper function:

$cot^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$cot^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\mathrm{cot}(z)$ in $\mathcal{C}$ with maximum error $max\_error\_tan_{\mathrm{c}(F)}$.

Further requirements on the $cot^*_{\mathrm{c}(F)}$ approximation helper function are:

$cot^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(cot^*_{\mathrm{c}(F)}(z)) \qquad$ if $z \in \mathcal{C}$

$cot^*_{\mathrm{c}(F)}(-z) = -cot^*_{\mathrm{c}(F)}(z) \qquad$ if $z \in \mathcal{C}$

The relationship to the $cot^*_F$ and $coth^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$cot^*_{\mathrm{c}(F)}(x) = cot^*_F(x) \qquad$ if $x \in \mathcal{R}$

$cot^*_{\mathrm{c}(F)}(\hat{\imath} \cdot x') = -\hat{\imath} \cdot coth_F(x') \qquad$ if $x' \in \mathcal{R}$

The $cot_{\mathrm{c}(F)}$ operation:

$cot_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{absolute\_precision\_underflow}\}$

$$
\begin{aligned}
&cot_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \\
&\qquad = result^*_{\mathrm{c}(F)}(cot^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x'), nearest_F) \\
&\qquad\qquad \text{if } x + \hat{\imath} \cdot x' \in \mathrm{c}(F) \text{ and } |x| \leqslant big\_angle\_r_F \text{ and} \\
&\qquad\qquad\quad (x \neq 0 \text{ or } x' \neq 0) \\
&\qquad = conj_{\mathrm{c}(F)}(tan_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0)) \\
&\qquad\qquad \text{if } x' = -\mathbf{0} \\
&\qquad = neg_F(tan_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot neg_F(x'))) \\
&\qquad\qquad \text{if } x = -\mathbf{0} \text{ and } x' \neq -\mathbf{0} \\
&\qquad = mul_F(0, tan_F(x)) + \hat{\imath} \cdot (-1) \\
&\qquad\qquad \text{if } x' = +\infty \\
&\qquad = mul_F(0, tan_F(x)) + \hat{\imath} \cdot 1 \\
&\qquad\qquad \text{if } x' = -\infty \\
&\qquad = \mathbf{infinitary}(+\infty + \hat{\imath} \cdot (+\infty)) \\
&\qquad\qquad \text{if } x = 0 \text{ and } x' = 0 \\
&\qquad = rad_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \qquad \text{otherwise}
\end{aligned}
$$

#### 5.3.2.6 Radian secant

The $sec_{\mathrm{i}(F)}$ operation:

$sec_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{c}(F) \cup \{\mathbf{overflow}\}$

$sec_{\mathrm{i}(F)}(\hat{\imath} \cdot x') \quad = neg_{\mathrm{i}(F)}(\hat{\imath} \cdot sech_F(x'))$

The $sec^*_{\mathrm{c}(F)}$ approximation helper function:

$sec^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$sec^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\sec(z)$ in $\mathcal{C}$ with maximum error $max\_error\_tan_{\mathrm{c}(F)}$.

Further requirements on the $sec^*_{\mathrm{c}(F)}$ approximation helper function are:

$sec^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(sec^*_{\mathrm{c}(F)}(z))$ if $z \in \mathcal{C}$
$sec^*_{\mathrm{c}(F)}(-z) = -sec^*_{\mathrm{c}(F)}(z)$ if $z \in \mathcal{C}$

The relationship to the $sec^*_F$ and $sech^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$sec^*_{\mathrm{c}(F)}(x) = sec^*_F(x)$ if $x \in \mathcal{R}$
$sec^*_{\mathrm{c}(F)}(\hat{\imath} \cdot x') = -\hat{\imath} \cdot sech_F(x')$ if $x' \in \mathcal{R}$

The $sec_{\mathrm{c}(F)}$ operation:

$sec_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{absolute\_precision\_underflow}\}$

$$
\begin{aligned}
&sec_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \\
&\qquad = result^*_{\mathrm{c}(F)}(sec^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x'), nearest_F) && \text{if } x + \hat{\imath} \cdot x' \in \mathrm{c}(F) \text{ and } |x| \leqslant big\_angle\_r_F \\
&\qquad = conj_{\mathrm{c}(F)}(sec_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0)) && \text{if } x' = -\mathbf{0} \\
&\qquad = sec_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot neg_F(x')) && \text{if } x = -\mathbf{0} \text{ and } x' \neq -\mathbf{0} \\
&\qquad = mul_F(0, cos_F(x)) + \hat{\imath} \cdot mul_F(-\mathbf{0}, sin_F(x)) && \text{if } x' = +\boldsymbol{\infty} \\
&\qquad = mul_F(0, cos_F(x)) + \hat{\imath} \cdot mul_F(0, sin_F(x)) && \text{if } x' = -\boldsymbol{\infty} \\
&\qquad = rad_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') && \text{otherwise}
\end{aligned}
$$

#### 5.3.2.7 Radian cosecant

The $csc_{\mathrm{i}(F)}$ operation:

$csc_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{c}(F) \cup \{\mathbf{overflow}, \mathbf{infinitary}\}$

$csc_{\mathrm{i}(F)}() = neg_{\mathrm{i}(F)}(\hat{\imath} \cdot csch_F(x'))$

The $csc^*_{\mathrm{c}(F)}$ approximation helper function:

$csc^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$csc^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\csc(z)$ in $\mathcal{C}$ with maximum error $max\_error\_tan_{\mathrm{c}(F)}$.

Further requirements on the $csc^*_{\mathrm{c}(F)}$ approximation helper function are:

$csc^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(csc^*_{\mathrm{c}(F)}(z))$ if $z \in \mathcal{C}$
$csc^*_{\mathrm{c}(F)}(-z) = -csc^*_{\mathrm{c}(F)}(z)$ if $z \in \mathcal{C}$

The relationship to the $csc^*_F$ and $csch^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$csc^*_{\mathrm{c}(F)}(x) = csc^*_F(x)$ if $x \in \mathcal{R}$
$csc^*_{\mathrm{c}(F)}(\hat{\imath} \cdot x') = -\hat{\imath} \cdot csch_F(x')$ if $x' \in \mathcal{R}$

The $csc_{\mathrm{c}(F)}$ operation:

$csc_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{absolute\_precision\_underflow}\}$

$$
\begin{aligned}
&csc_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \\
&\qquad = result^*_{\mathrm{c}(F)}(csc^*_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x'), nearest_F) \\
&\qquad\qquad \text{if } x + \hat{\imath} \cdot x' \in \mathrm{c}(F) \text{ and } |x| \leqslant big\_angle\_r_F \text{ and} \\
&\qquad\qquad\quad (x \neq 0 \text{ or } x' \neq 0) \\
&\qquad = conj_{\mathrm{c}(F)}(csc_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0)) \\
&\qquad\qquad \text{if } x' = -\mathbf{0} \\
&\qquad = neg_F(csc_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot neg_F(x'))) \\
&\qquad\qquad \text{if } x = -\mathbf{0} \text{ and } x' \neq -\mathbf{0} \\
&\qquad = mul_F(-\mathbf{0}, sin_F(x)) + \hat{\imath} \cdot mul_F(-\mathbf{0}, cos_F(x)) \\
&\qquad\qquad \text{if } x' = +\infty \\
&\qquad = mul_F(-\mathbf{0}, sin_F(x)) + \hat{\imath} \cdot mul_F(0, cos_F(x)) \\
&\qquad\qquad \text{if } x' = -\infty \\
&\qquad = \mathbf{infinitary}(+\infty + \hat{\imath} \cdot (+\infty)) \\
&\qquad\qquad \text{if } x = 0 \text{ and } x' = 0 \\
&\qquad = rad_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \qquad \text{otherwise}
\end{aligned}
$$

#### 5.3.2.8 Radian arc sine

The $arcsin_{\mathrm{i}(F)}$ operation:

$arcsin_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F)$

$arcsin_{\mathrm{i}(F)}(\hat{\imath} \cdot x') = \hat{\imath} \cdot arcsinh_F(x')$

The $arcsin^*_{\mathrm{c}(F)}$ approximation helper function:

$arcsin^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$arcsin^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\arcsin(z)$ in $\mathcal{C}$ with maximum error $max\_error\_sin_{\mathrm{c}(F)}$.

Further requirements on the $arcsin^*_{\mathrm{c}(F)}$ approximation helper function are:

$$
\begin{aligned}
&arcsin^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(arcsin^*_{\mathrm{c}(F)}(z)) && \text{if } z \in \mathcal{C} \\
&arcsin^*_{\mathrm{c}(F)}(-z) = -arcsin^*_{\mathrm{c}(F)}(z) && \text{if } z \in \mathcal{C} \\
&\mathfrak{Re}(arcsin^*_{\mathrm{c}(F)}(x)) = -\pi/2 && \text{if } x \in \mathcal{R} \text{ and } x \leqslant -1 \\
&\mathfrak{Re}(arcsin^*_{\mathrm{c}(F)}(x)) = \pi/2 && \text{if } x \in \mathcal{R} \text{ and } x \geqslant 1
\end{aligned}
$$

The relationship to the $arcsin^*_F$ and $arcsinh^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$$
\begin{aligned}
&arcsin^*_{\mathrm{c}(F)}(x) = arcsin^*_F(x) && \text{if } x \in \mathcal{R} \text{ and } |x| \leqslant 1 \\
&arcsin^*_{\mathrm{c}(F)}(\hat{\imath} \cdot x') = \hat{\imath} \cdot arcsinh^*_F(x') && \text{if } x' \in \mathcal{R}
\end{aligned}
$$

The $arcsin^{\#}_{\mathrm{c}(F)}$ range limitation helper function:

$$arcsin^{\#}_{\mathrm{c}(F)}(z) = \max\{up_F(-\pi/2), \min\{\mathfrak{Re}(arcsin^*_F(z)), down_F(\pi/2)\}\} + \hat{\imath} \cdot \mathfrak{Im}(arcsin^*_F(z))$$

The $arcsin_{\mathrm{c}(F)}$ operation:

$arcsin_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}\}$

$$
\begin{aligned}
&arcsin_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \\
&\qquad = result^*_{\mathrm{c}(F)}(arcsin^{\#}_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x'), nearest_F) \\
&\qquad\qquad \text{if } x + \hat{\imath} \cdot x' \in \mathrm{c}(F) \\
&\qquad = neg_{\mathrm{c}(F)}(arcsin_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot neg_F(x')) \\
&\qquad\qquad \text{if } x = -\mathbf{0} \\
&\qquad = conj_{\mathrm{c}(F)}(arcsin_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0) \\
&\qquad\qquad \text{if } x' = -\mathbf{0} \text{ and } x \neq -\mathbf{0} \\
&\qquad = arc_F(x', x) + \hat{\imath} \cdot (+\infty) \quad \text{if } x' = +\infty \text{ and } x \in \{-\infty, -\mathbf{0}, +\infty\}
\end{aligned}
$$

$$
\begin{aligned}
&= arc_F(neg_F(x'), x) + \hat{\imath} \cdot (-\infty) \\
&\qquad\qquad \text{if } x' = -\infty \text{ and } x \in \{-\infty, -\mathbf{0}, +\infty\} \\
&= arc_F(x', x) + \hat{\imath} \cdot (+\infty) \quad \text{if } x \in \{-\infty, +\infty\} \text{ and } x' \in F \text{ and } x' \geqslant 0 \\
&= arc_F(neg_F(x'), x) + \hat{\imath} \cdot (-\infty) \\
&\qquad\qquad \text{if } x \in \{-\infty, +\infty\} \text{ and } x' \in F \text{ and } x' < 0 \\
&= no\_result_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \\
&\qquad\qquad \text{otherwise}
\end{aligned}
$$

NOTE – The inverse of sin is multi-valued, the real part may have any integer multiple of $2 \cdot \pi$ added to it, and the result is also in the solution set. The arcsin function (returning the principal value for the inverse) branch cuts at $\{x \mid x \in \mathcal{R} \text{ and } |x| > 1\}$. Thus $arcsin_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0) \neq arcsin_{\mathrm{c}(F)}(x + \hat{\imath} \cdot (-\mathbf{0}))$ when $|x| > 1$.

#### 5.3.2.9 Radian arc cosine

The $arccos^*_{\mathrm{c}(F)}$ approximation helper function:

$arccos^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$arccos^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\arccos(z)$ in $\mathcal{C}$ with maximum error $max\_error\_sin_{\mathrm{c}(F)}$.

Further requirements on the $arccos^*_{\mathrm{c}(F)}$ approximation helper function are:

$$
\begin{aligned}
&arccos^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(arccos^*_{\mathrm{c}(F)}(z)) && \text{if } z \in \mathcal{C} \\
&\mathfrak{Re}(arccos^*_{\mathrm{c}(F)}(x)) = \pi && \text{if } x \in \mathcal{R} \text{ and } x \leqslant -1
\end{aligned}
$$

The relationship to the $arccos^*_F$ and $arccosh^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$$
arccos^*_{\mathrm{c}(F)}(x) = arccos^*_F(x) \qquad \text{if } x \in \mathcal{R} \text{ and } |x| \leqslant 1
$$

The $arccos^\#_{\mathrm{c}(F)}$ range limitation helper function:

$$
arccos^\#_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') = \min\{\mathfrak{Re}(arccos^*_F(x + \hat{\imath} \cdot x')), down_F(\pi)\} + \hat{\imath} \cdot \mathfrak{Im}(arccos^*_F(x + \hat{\imath} \cdot x'))
$$

The $arccos_{\mathrm{c}(F)}$ operation:

$arccos_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F)$

$arccos_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')$

$$
\begin{aligned}
&= result^*_{\mathrm{c}(F)}(arccos^\#_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x'), nearest_F) \\
&\qquad\qquad \text{if } x + \hat{\imath} \cdot x' \in \mathrm{c}(F) \\
&= arccos_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot neg_F(x') \\
&\qquad\qquad \text{if } x = -\mathbf{0} \\
&= conj_{\mathrm{c}(F)}(arccos_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0) \\
&\qquad\qquad \text{if } x' = -\mathbf{0} \text{ and } x \neq -\mathbf{0} \\
&= arc_F(x, x') + \hat{\imath} \cdot (-\infty) \quad \text{if } x' = +\infty \text{ and } x \in \{-\infty, -\mathbf{0}, +\infty\} \\
&= arc_F(x, neg_F(x')) + \hat{\imath} \cdot (+\infty) \\
&\qquad\qquad \text{if } x' = -\infty \text{ and } x \in \{-\infty, -\mathbf{0}, +\infty\} \\
&= arc_F(x, x') + \hat{\imath} \cdot (-\infty) \quad \text{if } x \in \{-\infty, +\infty\} \text{ and } x' \in F \text{ and } x' \geqslant 0 \\
&= arc_F(x, neg_F(x')) + \hat{\imath} \cdot (+\infty) \\
&\qquad\qquad \text{if } x \in \{-\infty, +\infty\} \text{ and } x' \in F \text{ and } x' < 0 \\
&= no\_result_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \\
&\qquad\qquad \text{otherwise}
\end{aligned}
$$

NOTE – The inverse of cos is multi-valued, the real part may have any integer multiple of $2 \cdot \pi$ added to it, and the result is also in the solution set. The arccos function (returning the principal value for the inverse) branch cuts at $\{x \mid x \in \mathcal{R} \text{ and } |x| > 1\}$. Thus $arccos_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0) \neq arccos_{\mathrm{c}(F)}(x + \hat{\imath} \cdot (-\mathbf{0}))$ when $|x| > 1$.

#### 5.3.2.10 Radian arc tangent

The $arctan_{\mathrm{i}(F)}$ operation:

$arctan_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$arctan_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x') = \hat{\mathbf{i}} \cdot arctanh_F(x')$

The $arctan^*_{\mathrm{c}(F)}$ approximation helper function:

$arctan^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$arctan^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\arctan(z)$ in $\mathcal{C}$ with maximum error $max\_error\_tan_{\mathrm{c}(F)}$.

Further requirements on the $arctan^*_{\mathrm{c}(F)}$ approximation helper function are:

$arctan^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(arctan^*_{\mathrm{c}(F)}(z))$ if $z \in \mathcal{C}$

$arctan^*_{\mathrm{c}(F)}(-z) = -arctan^*_{\mathrm{c}(F)}(z)$ if $z \in \mathcal{C}$

$\mathfrak{Re}(arctan^*_{\mathrm{c}(F)}(\hat{\mathrm{i}} \cdot x') = \pi/2$ if $x' \in \mathcal{R}$ and $|x'| > 1$

The relationship to the $arctan^*_F$ and $arctanh^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$arctan^*_{\mathrm{c}(F)}(x) = arctan^*_F(x)$ if $x \in \mathcal{R}$

$arctan^*_{\mathrm{c}(F)}(\hat{\mathrm{i}} \cdot x') = \hat{\mathrm{i}} \cdot arctanh^*_F(x')$ if $x' \in \mathcal{R}$ and $|x'| < 1$

The $arctan^{\#}_{\mathrm{c}(F)}$ range limitation helper function:

$arctan^{\#}_{\mathrm{c}(F)}(z) = \max\{up_F(-\pi/2), \min\{\mathfrak{Re}(arctan^*_F(z)), down_F(\pi/2)\}\} + \hat{\mathrm{i}} \cdot \mathfrak{Im}(arctan^*_F(z))$

The $arctan_{\mathrm{c}(F)}$ operation:

$arctan_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$arctan_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$

$= result^*_{\mathrm{c}(F)}(arctan^{\#}_{\mathrm{c}(F)}(x + \hat{\mathrm{i}} \cdot x')), nearest_F)$ if $x + \hat{\mathbf{i}} \cdot x' \in \mathrm{c}(F)$ and $((x' \neq 1$ and $x' \neq -1)$ or $x \neq 0)$

$= neg_{\mathrm{c}(F)}(arctan_{\mathrm{c}(F)}(0 + \hat{\mathbf{i}} \cdot neg_F(x'))$ if $x = -\mathbf{0}$

$= conj_{\mathrm{c}(F)}(arctan_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot 0)$ if $x' = -\mathbf{0}$ and $x \neq -\mathbf{0}$

$= signb_F(x) \cdot down_F(\pi/2)$ if $(x \in \{-\infty, +\infty\}$ and $x' \in F \cup \{-\infty, +\infty\})$ or $(x \in F \cup \{-\infty, +\infty\}$ and $x' \in \{-\infty, +\infty\})$

$= \mathbf{infinitary}(down_F(\pi/2) + \hat{\mathbf{i}} \cdot (+\infty))$ if $x = 0$ and $x' = 1$

$= \mathbf{infinitary}(down_F(\pi/2) + \hat{\mathbf{i}} \cdot (-\infty))$ if $x = 0$ and $x' = -1$

$= no\_result_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$ otherwise

NOTE – The inverse of tan is multi-valued, the real part may have any integer multiple of $2 \cdot \pi$ (even any integer multiple of $\pi$) added to it, and the result is also in the solution set. The arctan function (returning the principal value for the inverse) branch cuts at $\{\hat{\mathbf{i}} \cdot x' \mid x' \in F$ and $|x'| > 1\}$. Thus $arctan_{\mathrm{c}(F)}(0 + \hat{\mathbf{i}} \cdot x') \neq arctan_{\mathrm{c}(F)}(-\mathbf{0} + \hat{\mathbf{i}} \cdot x')$ when $|x'| > 1$.

#### 5.3.2.11 Radian arc cotangent

The $arccot_{\mathrm{i}(F)}$ operation:

$arccot_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$arccot_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x') = \hat{\mathbf{i}} \cdot arccoth_F(x')$

The $arccot^*_{\mathrm{c}(F)}$ approximation helper function:

$arccot^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$

$arccot^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\mathrm{arccot}(z)$ in $\mathcal{C}$ with maximum error $max\_error\_tan_{\mathrm{c}(F)}$.

Further requirements on the $arccot^*_{\mathrm{c}(F)}$ approximation helper function are:

$arccot^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(arccot^*_{\mathrm{c}(F)}(z))$ if $z \in \mathcal{C}$
$arccot^*_{\mathrm{c}(F)}(-z) = -arccot^*_{\mathrm{c}(F)}(z)$ if $z \in \mathcal{C}$
$\mathfrak{Re}(arccot^*_{\mathrm{c}(F)}(\hat{\mathrm{i}} \cdot x')) = \pi/2$ if $x' \in \mathcal{R}$ and $|x'| < 1$

The relationship to the $arccot^*_F$ and $arccoth^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$arccot^*_{\mathrm{c}(F)}(x) = arccot^*_F(x)$ if $x \in \mathcal{R}$
$arccot^*_{\mathrm{c}(F)}(\hat{\mathrm{i}} \cdot x') = \hat{\mathrm{i}} \cdot arccoth^*_F(-x')$ if $x' \in \mathcal{R}$

The $arccot^\#_{\mathrm{c}(F)}$ range limitation helper function:

$arccot^\#_{\mathrm{c}(F)}(z) = \max\{up_F(-\pi/2), \min\{\mathfrak{Re}(arccot^*_F(z)), down_F(\pi/2)\}\} + \hat{\mathrm{i}} \cdot \mathfrak{Im}(arccot^*_F(z))$

The $arccot_{\mathrm{c}(F)}$ operation:

$arccot_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$arccot_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$

$= result^*_{\mathrm{c}(F)}(arccot^\#_{\mathrm{c}(F)}(x + \hat{\mathrm{i}} \cdot x')), nearest_F)$
if $x + \hat{\mathbf{i}} \cdot x' \in \mathrm{c}(F)$

$= neg_{\mathrm{c}(F)}(arccot_{\mathrm{c}(F)}(0 + \hat{\mathbf{i}} \cdot neg_F(x'))$
if $x = -\mathbf{0}$

$= conj_{\mathrm{c}(F)}(arccot_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot 0)$
if $x' = -\mathbf{0}$ and $x \neq -\mathbf{0}$

$= mul_F(signb_F(x), 0) + \hat{\mathbf{i}} \cdot mul_F(signb_F(x'), 0)$
if $(x \in \{-\infty, +\infty\}$ and $x' \in F \cup \{-\infty, +\infty\})$ or
$(x \in F \cup \{-\infty, +\infty\}$ and $x' \in \{-\infty, +\infty\})$

$= \mathbf{infinitary}(down_F(\pi/2), x) + \hat{\mathbf{i}} \cdot (+\infty)$
if $x' = -1$ and $x = 0$

$= \mathbf{infinitary}(down_F(\pi/2), x) + \hat{\mathbf{i}} \cdot (-\infty)$
if $x' = 1$ and $x = 0$

$= no\_result_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$
otherwise

NOTE – The inverse of cot is multi-valued, the real part may have any integer multiple of $2 \cdot \pi$ (even any integer multiple of $\pi$) added to it, and the result is also in the solution set. The arccot function (returning the principal value for the inverse) branch cuts at $\{\hat{\mathbf{i}} \cdot x' \mid x' \in \mathcal{R}$ and $|x'| < 1\}$. Thus $arccot_{\mathrm{c}(F)}(0 + \hat{\mathbf{i}} \cdot x') \neq arccot_{\mathrm{c}(F)}(-\mathbf{0} + \hat{\mathbf{i}} \cdot x')$ when $|x'| < 1$ or $x' = -\mathbf{0}$.

#### 5.3.2.12 Radian arc secant

The $arcsec^*_{\mathrm{c}(F)}$ approximation helper function:

$$arcsec^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$$

$arcsec^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\mathrm{arcsec}(z)$ in $\mathcal{C}$ with maximum error $max\_error\_tan_{\mathrm{c}(F)}$.

Further requirements on the $arcsec^*_{\mathrm{c}(F)}$ approximation helper function are:

$$arcsec^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(arcsec^*_{\mathrm{c}(F)}(z)) \quad \text{if } z \in \mathcal{C}$$

The relationship to the $arcsec^*_F$ and $arcsech^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$$arcsec^*_{\mathrm{c}(F)}(x) = arcsec^*_F(x) \qquad \text{if } x \in \mathcal{R}$$

The $arcsec^\#_{\mathrm{c}(F)}$ range limitation helper function:

$$\begin{aligned} arcsec^\#_{\mathrm{c}(F)}(z) &= \min\{\mathfrak{Re}(arcsec^*_F(z)), down_F(\pi/2)\} + \hat{\imath} \cdot \mathfrak{Im}(arcsec^*_F(z)) \quad \text{if } \mathfrak{Re}(z) \geqslant 1 \\ &= \min\{\mathfrak{Re}(arcsec^*_F(z)), down_F(\pi)\} + \hat{\imath} \cdot \mathfrak{Im}(arcsec^*_F(z)) \quad \text{if } \mathfrak{Re}(z) \leqslant -1 \\ &= arcsec_{\mathrm{c}(F)}(z) \qquad \text{otherwise} \end{aligned}$$

The $arcsec_{\mathrm{c}(F)}$ operation:

$$arcsec_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$$

$$\begin{aligned} &arcsec_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \\ &\quad = result^*_{\mathrm{c}(F)}(arcsec^\#_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')), nearest_F) \\ &\qquad \text{if } x + \hat{\imath} \cdot x' \in \mathrm{c}(F) \\ &\quad = arcsec_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot neg_F(x')) \\ &\qquad \text{if } x = -\mathbf{0} \\ &\quad = conj_{\mathrm{c}(F)}(arcsec_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0)) \\ &\qquad \text{if } x' = -\mathbf{0} \text{ and } x \neq -\mathbf{0} \\ &\quad = arccos_{\mathrm{c}(F)}(div_{\mathrm{c}(F)}(1, x + \hat{\imath} \cdot x')) \\ &\qquad \text{if } x \in \{-\infty, +\infty\} \text{ and } x' \in F \cup \{-\infty, +\infty\} \\ &\quad = arccos_{\mathrm{c}(F)}(div_{\mathrm{c}(F)}(1, x + \hat{\imath} \cdot x')) \\ &\qquad \text{if } x' \in \{-\infty, +\infty\} \text{ and } x \in F \cup \{-\infty, +\infty\} \\ &\quad = \mathbf{infinitary}(? + \hat{\imath} \cdot (+\infty)) \\ &\qquad \text{if } x' = 0 \text{ and } x = 0 \\ &\quad = no\_result_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \\ &\qquad \text{otherwise} \end{aligned}$$

NOTE – The inverse of sec is multi-valued, the real part may have any integer multiple of $2 \cdot \pi$ added to it, and the result is also in the solution set. The arcsec function (returning the principal value for the inverse) branch cuts at $\{x \mid x \in \mathcal{R} \text{ and } -1 < x < 1\}$. Thus $arcsec_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0) \neq arcsec_{\mathrm{c}(F)}(x + \hat{\imath} \cdot (-\mathbf{0}))$ when $-1 < x < 1$ or $x = -\mathbf{0}$.

#### 5.3.2.13 Radian arc cosecant

The $arccsc_{\mathrm{i}(F)}$ operation:

$$arccsc_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$$

$$arccsc_{\mathrm{i}(F)}(\hat{\imath} \cdot x') = \hat{\imath} \cdot arccsch_F(x')$$

The $arccsc^*_{\mathrm{c}(F)}$ approximation helper function:

$$arccsc^*_{\mathrm{c}(F)} : \mathcal{C} \to \mathcal{C}$$

$arccsc^*_{\mathrm{c}(F)}(z)$ returns a close approximation to $\mathrm{arccsc}(z)$ in $\mathcal{C}$ with maximum error $max\_error\_tan_{\mathrm{c}(F)}$.

Further requirements on the $arccsc^*_{\mathrm{c}(F)}$ approximation helper function are:

$arccsc^*_{\mathrm{c}(F)}(\mathrm{conj}(z)) = \mathrm{conj}(arccsc^*_{\mathrm{c}(F)}(z))$ if $z \in \mathcal{C}$
$arccsc^*_{\mathrm{c}(F)}(-z) = -arccsc^*_{\mathrm{c}(F)}(z)$ if $z \in \mathcal{C}$

The relationship to the $arccsc^*_F$ and $arccsch^*_F$ approximation helper functions in an associated library for real-valued operations shall be:

$arccsc^*_{\mathrm{c}(F)}(x) = arccsc^*_F(x)$ if $x \in \mathcal{R}$
$arccsc^*_{\mathrm{c}(F)}(\hat{\imath} \cdot x') = \hat{\imath} \cdot arccsch^*_F(-x')$ if $x' \in \mathcal{R}$

The $arccsc^{\#}_{\mathrm{c}(F)}$ range limitation helper function:

$$arccsc^{\#}_{\mathrm{c}(F)}(z) = \max\{up_F(-\pi/2), \min\{\mathfrak{Re}(arccsc^*_F(z)), down_F(\pi/2)\}\} + \hat{\imath} \cdot \mathfrak{Im}(arccsc^*_F(z))$$

The $arccsc_{\mathrm{c}(F)}$ operation:

$arccsc_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$$\begin{aligned}
&arccsc_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') \\
&\quad = result^*_{\mathrm{c}(F)}(arccsc^{\#}_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')), nearest_F) && \text{if } x + \hat{\imath} \cdot x' \in \mathrm{c}(F) \\
&\quad = neg_{\mathrm{c}(F)}(arccsc_{\mathrm{c}(F)}(0 + \hat{\imath} \cdot neg_F(x')) && \text{if } x = -\mathbf{0} \\
&\quad = conj_{\mathrm{c}(F)}(arccsc_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0) && \text{if } x' = -\mathbf{0} \text{ and } x \neq -\mathbf{0} \\
&\quad = arcsin_{\mathrm{c}(F)}(div_{\mathrm{c}(F)}(1, x + \hat{\imath} \cdot x')) && \text{if } x \in \{-\infty, +\infty\} \text{ and } x' \in F \cup \{-\infty, +\infty\} \\
&\quad = arcsin_{\mathrm{c}(F)}(div_{\mathrm{c}(F)}(1, x + \hat{\imath} \cdot x')) && \text{if } x' \in \{-\infty, +\infty\} \text{ and } x \in F \cup \{-\infty, +\infty\} \\
&\quad = \mathbf{infinitary}(? + \hat{\imath} \cdot (-\infty)) && \text{if } x = 0 \text{ and } x' = 0 \\
&\quad = no_result_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x') && \text{otherwise}
\end{aligned}$$

NOTE – The inverse of csc is multi-valued, the real part may have any integer multiple of $2 \cdot \pi$ added to it, and the result is also in the solution set. The arccsc function (returning the principal value for the inverse) branch cuts at $\{x \mid x \in \mathcal{R} \text{ and } -1 < x < 1\}$. Thus $arccsc_{\mathrm{c}(F)}(x + \hat{\imath} \cdot 0) \neq arccsc_{\mathrm{c}(F)}(x + \hat{\imath} \cdot (-\mathbf{0}))$ when $-1 < x < 1$ or $x = -\mathbf{0}$.

### 5.3.3 Operations for hyperbolic elementary functions

The complex trigonometric operations which are used in the specifications below are specified in clause 5.3.2. Note that the correspondences specified here are exact, not approximate.

#### 5.3.3.1 Hyperbolic normalisation

$radh_F : F \to F$
$radh_F(x) = x$

$radh_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{underflow}, \mathbf{absolute\_precision\_underflow}\}$
$radh_{\mathrm{i}(F)}(\hat{\imath} \cdot x') = \hat{\imath} \cdot rad_F(x')$

$radh_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{absolute\_precision\_underflow}\}$
$radh_{\mathrm{c}(F)}(x + \hat{\imath} \cdot x')$
$= \hat{\imath} \cdot rad_{\mathrm{c}(F)}(x' + \hat{\imath} \cdot neg_F(x))$

NOTE – $rad_{\mathrm{c}(F)}$ is defined in clause 5.3.2.1.

#### 5.3.3.2 Hyperbolic sine

$sinh_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{underflow}, \mathbf{absolute\_precision\_underflow}\}$
$sinh_{\mathrm{i}(F)}(\hat{\mathbf{1}} \cdot x') \quad = \hat{\mathbf{1}} \cdot (sin_F(x'))$

$sinh_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{absolute\_precision\_underflow}\}$
$sinh_{\mathrm{c}(F)}(x + \hat{\mathbf{1}} \cdot x')$
$\qquad = \hat{\mathbf{1}} \cdot (sin_{\mathrm{c}(F)}(x' + \hat{\mathbf{1}} \cdot neg_F(x)))$

#### 5.3.3.3 Hyperbolic cosine

$cosh_{\mathrm{i}(F)} : \mathrm{i}(F) \to F \cup \{\mathbf{underflow}, \mathbf{absolute\_precision\_underflow}\}$
$cosh_{\mathrm{i}(F)}(\hat{\mathbf{1}} \cdot x') \quad = cos_F(x')$

$cosh_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{absolute\_precision\_underflow}\}$
$cosh_{\mathrm{c}(F)}(x + \hat{\mathbf{1}} \cdot x')$
$\qquad = cos_{\mathrm{c}(F)}(x' + \hat{\mathbf{1}} \cdot neg_F(x))$

#### 5.3.3.4 Hyperbolic tangent

$tanh_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{absolute\_precision\_underflow}\}$
$tanh_{\mathrm{i}(F)}(\hat{\mathbf{1}} \cdot x') \quad = \hat{\mathbf{1}} \cdot (tan_F(x'))$

$tanh_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{absolute\_precision\_underflow}\}$
$tanh_{\mathrm{c}(F)}(x + \hat{\mathbf{1}} \cdot x')$
$\qquad = \hat{\mathbf{1}} \cdot (tan_{\mathrm{c}(F)}(x' + \hat{\mathbf{1}} \cdot neg_F(x)))$

#### 5.3.3.5 Hyperbolic cotangent

$coth_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{absolute\_precision\_underflow}\}$
$coth_{\mathrm{i}(F)}(\hat{\mathbf{1}} \cdot x') \quad = \hat{\mathbf{1}} \cdot (cot_F(neg_F(x')))$

$coth_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{absolute\_precision\_underflow}\}$
$coth_{\mathrm{c}(F)}(x + \hat{\mathbf{1}} \cdot x')$
$\qquad = \hat{\mathbf{1}} \cdot (cot_{\mathrm{c}(F)}(neg_F(x') + \hat{\mathbf{1}} \cdot x))$

#### 5.3.3.6 Hyperbolic secant

$$sech_{\mathrm{i}(F)} : \mathrm{i}(F) \to F \cup \{\textbf{underflow}, \textbf{overflow}, \textbf{absolute\_precision\_underflow}\}$$
$$sech_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x') = sec_F(neg_F(x'))$$

$$sech_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\textbf{underflow}, \textbf{overflow}, \textbf{absolute\_precision\_underflow}\}$$
$$sech_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x') = sec_{\mathrm{c}(F)}(neg_F(x') + \hat{\mathbf{i}} \cdot x)$$

#### 5.3.3.7 Hyperbolic cosecant

$$csch_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\textbf{underflow}, \textbf{overflow}, \textbf{infinitary}, \textbf{absolute\_precision\_underflow}\}$$
$$csch_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x') = \hat{\mathbf{i}} \cdot (csc_F(neg_F(x')))$$

$$csch_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\textbf{underflow}, \textbf{overflow}, \textbf{infinitary}, \textbf{absolute\_precision\_underflow}\}$$
$$csch_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x') = \hat{\mathbf{i}} \cdot (csc_{\mathrm{c}(F)}(neg_F(x') + \hat{\mathbf{i}} \cdot x))$$

#### 5.3.3.8 Inverse hyperbolic sine

$$arcsinh_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\textbf{underflow}\}$$
$$arcsinh_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x') = \hat{\mathbf{i}} \cdot (arcsin_F(x'))$$

$$arcsinh_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\textbf{underflow}\}$$
$$arcsinh_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x') = \hat{\mathbf{i}} \cdot (arcsin_{\mathrm{c}(F)}(x' + \hat{\mathbf{i}} \cdot neg_F(x)))$$

NOTE – The inverse of sinh is multi-valued, the imaginary part may have any integer multiple of $2 \cdot \pi$ added to it, and the result is also in the solution set. The arcsinh function (returning the principal value for the inverse) branch cuts at $\{\hat{\mathrm{i}} \cdot x' \mid x' \in \mathcal{R} \text{ and } |x'| > 1\}$. Thus $arcsinh_{\mathrm{c}(F)}(0 + \hat{\mathbf{i}} \cdot x') \neq arcsinh_{\mathrm{c}(F)}(-\mathbf{0} + \hat{\mathbf{i}} \cdot x')$ when $|x'| > 1$.

#### 5.3.3.9 Inverse hyperbolic cosine

$$arccosh_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F)$$
$$arccosh_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x') = \hat{\mathbf{i}} \cdot (arccos_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x'))$$
if $(x' \in F \text{ and } x' \geqslant 0)$ or $x' = +\infty$
$$= neg_{\mathrm{c}(F)}(\hat{\mathbf{i}} \cdot (arccos_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')))$$
if $(x' \in F \text{ and } x' < 0)$ or $x' \in \{-\infty, -\mathbf{0}\}$

NOTE – The inverse of cosh is multi-valued, the imaginary part may have any integer multiple of $2 \cdot \pi$ added to it, and the result is also in the solution set. The arccosh function (returning the principal value for the inverse) branch cuts at $\{x \mid x \in \mathcal{R} \text{ and } x < 1\}$. Thus $arccosh_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot 0) \neq arccosh_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot (-\mathbf{0}))$ when $x < 1$ or $x = -\mathbf{0}$.

#### 5.3.3.10 Inverse hyperbolic tangent

$arctanh_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$arctanh_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x')$
$= \hat{\mathbf{i}} \cdot (arctan_F(x'))$

$arctanh_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$arctanh_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$
$= \hat{\mathbf{i}} \cdot (arctan_{\mathrm{c}(F)}(x' + \hat{\mathbf{i}} \cdot neg_F(x)))$

NOTE – The inverse of tanh is multi-valued, the imaginary part may have any integer multiple of $2 \cdot \pi$ (even any integer multiple of $\pi$) added to it, and the result is also in the solution set. The arctanh function (returning the principal value for the inverse) branch cuts at $\{x \mid x \in \mathcal{R} \text{ and } |x| > 1\}$. Thus $arctanh_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot 0) \neq arctanh_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot (-\mathbf{0}))$ when $|x| > 1$.

#### 5.3.3.11 Inverse hyperbolic cotangent

$arccoth_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$arccoth_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x')$
$= \hat{\mathbf{i}} \cdot (arccot_F(neg_F(x')))$

$arccoth_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$arccoth_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$
$= \hat{\mathbf{i}} \cdot (arccot_{\mathrm{c}(F)}(neg_F(x') + \hat{\mathbf{i}} \cdot x))$

NOTE – The inverse of coth is multi-valued, the imaginary part may have any integer multiple of $2 \cdot \pi$ (even any integer multiple of $\pi$) added to it, and the result is also in the solution set. The arccoth function (returning the principal value for the inverse) branch cuts at $\{x \mid x \in \mathcal{R} \text{ and } |x| < 1\}$. Thus $arccoth_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot 0) \neq arccoth_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot (-\mathbf{0}))$ when $|x| < 1$ or $x = -\mathbf{0}$.

#### 5.3.3.12 Inverse hyperbolic secant

$arcsech_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$arcsech_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$
$= \hat{\mathbf{i}} \cdot (arcsec_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x'))$
if $(x' \in F \text{ and } x' \geqslant 0)$ or $x' = +\boldsymbol{\infty}$
$= neg_{\mathrm{c}(F)}(\hat{\mathbf{i}} \cdot (arcsec_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')))$
if $(x' \in F \text{ and } x' < 0)$ or $x' \in \{-\boldsymbol{\infty}, -\mathbf{0}\}$

NOTE – The inverse of sech is multi-valued, the imaginary part may have any integer multiple of $2 \cdot \pi$ added to it, and the result is also in the solution set. The arcsech function (returning the principal value for the inverse) branch cuts at $\{x \mid x \in \mathcal{R} \text{ and } x \leqslant 0 \text{ or } x > 1\}$. Thus $arcsech_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot 0) \neq arcsech_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot (-\mathbf{0}))$ when $x \leqslant 0$ or $x = -\mathbf{0}$ or $x > 1$.

#### 5.3.3.13 Inverse hyperbolic cosecant

$arccsch_{\mathrm{i}(F)} : \mathrm{i}(F) \to \mathrm{i}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$arccsch_{\mathrm{i}(F)}(\hat{\mathbf{i}} \cdot x')$
$= \hat{\mathbf{i}} \cdot (arccsc_F(neg_F(x')))$

$arccsch_{\mathrm{c}(F)} : \mathrm{c}(F) \to \mathrm{c}(F) \cup \{\mathbf{underflow}, \mathbf{infinitary}\}$

$arccsch_{\mathrm{c}(F)}(x + \hat{\mathbf{i}} \cdot x')$
$= \hat{\mathbf{i}} \cdot (arccsc_{\mathrm{c}(F)}(neg_F(x') + \hat{\mathbf{i}} \cdot x))$

NOTE – The inverse of csch is multi-valued, the imaginary part may have any integer multiple of $2 \cdot \pi$ added to it, and the result is also in the solution set. The arccsch function (returning the principal value for the inverse) branch cuts at $\{\hat{\mathrm{i}} \cdot x' \mid x' \in \mathcal{R} \text{ and } |x'| < 1\}$. Thus $arccsch_{\mathrm{c}(F)}(0 + \hat{\mathbf{i}} \cdot x') \neq arccsch_{\mathrm{c}(F)}(-\mathbf{0} + \hat{\mathbf{i}} \cdot x')$ when $-1 < x' < 1$ or $x' = -\mathbf{0}$.

## 5.4 Operations for conversion between numeric datatypes

### 5.4.1 Integer to complex integer conversions

Let $I$ be the non-special value set for an integer datatype.

$convert_{I \to c(I)} : I \to c(I)$

$convert_{I \to c(I)}(x)$
$= x + \hat{\mathbf{i}} \cdot 0$

$convert_{i(I) \to c(I)} : i(I) \to c(I)$

$convert_{i(I) \to c(I)}(\hat{\mathbf{i}} \cdot x)$
$= 0 + \hat{\mathbf{i}} \cdot x$

### 5.4.2 Floating point to complex floating point conversions

Let $F$ be the non-special value set for a floating point datatype.

The $convert_{F \to c(F)}$ operation:

$convert_{F \to c(F)} : F \to c(F$

$convert_{F \to c(F)}(x)$
$= x + \hat{\mathbf{i}} \cdot 0$

The $convert_{i(F) \to c(F)}$ operation:

$convert_{i(F) \to c(F)} : i(F) \to c(F)$

$convert_{i(F) \to c(F)}(\hat{\mathbf{i}} \cdot x)$
$= 0 + \hat{\mathbf{i}} \cdot x$

# 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 $i(I)$, $c(I)$, $i(F)$ or $c(F)$ (as appropriate), or be special values (where the real or imaginary component is $\mathbf{-0}$, $\mathbf{-\infty}$, $\mathbf{+\infty}$, 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: $\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, *signaling* **NaN**s (**sNaN**), and *quiet* **NaN**s (**qNaN**). Such values may be components of complex floating point datatypes, and may be included in values 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**.

## 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 complex radian arc sine operation for an argument $x$ ($arcsin_{c(F)}(x)$) might be invoked as
>
> | | |
> |---|---|
> | `arcsin(`$x$`)` | in Ada [7] |
> | `casin(`$x$`)` | in C [13] |
> | `asin(`$x$`)` | in Fortran [18] and C++ [14] |
> | `(asin` $x$`)` | in Common Lisp [38] |
>
> with a suitable expression of 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.

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).

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.

# 8 Documentation requirements

In order to conform to this Part, an implementation shall include documentation providing the following information to programmers.

NOTE 1 – 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.

NOTE 2 – Implementations that do not provide infinities or **NaN**s will have to document any continuation values used in place of such values.

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.

$max\_error\_mul_{c(F)} \in [0.5, ??]$
$max\_error\_div_{c(F)} \in [0.5, ??]$

$max\_error\_exp_{c(F)} \in [0.5, ??]$
$max\_error\_power_{c(F)} \in [0.5, ??]$

$max\_error\_sin_{c(F)} \in [0.5, ??]$
$max\_error\_tan_{c(F)} \in [0.5, ??]$

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.

A partially conforming implementation shall document which extra accuracy requirements are not fulfilled by the implementation.

## A.3 Partial conformity to part 1 or to part 2

...

# 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).

## B.1 Scope

### 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 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 standardisation 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 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* standardisation, if standardised.

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. 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.

## B.2 Conformity

Conformity to this standard is dependent on the existence of language binding standards. Each programming language committee (or other organisation 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.

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.

## 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

### 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 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 programme excerpts which are clearly marked, any programming language. E.g. $x/u$ stands for the mathematically exact result of dividing $x$ by $u$, 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 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 respecitive approximating function; `sin`, `cos`, etc., which are programming language names bound to LIA-2 operations. `sin` is thus very different from sin.

#### B.4.1.4 Datatypes and exceptional values

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 multi-sets (bags) and sets also qualify.

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 this distinction, if desired.

LIA allows for three methods for handing notifications: recording of indicators, change of control flow (returnable or not), and termination of program. The 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. E.g., **invalid** without a specified (by LIA-2 or binding) continuation value to cause change of control flow (like an Ada [7] exception), while **invalid** with a specified continuation value 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.

### 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 the LIA distinction between denormal and subnormal. Subnormal include zero values, while denormal does not.

## B.5 Specifications for the complex datatypes and operations

# 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-3. It contains suggestions for binding the "abstract" operations specified in LIA-3 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-3 operation name, or parameter name, bound to an identifier by the binding.

Portability of programs can be improved if two conforming LIA-3 systems using the same language agree in the manner with which they adhere to LIA-3. For instance, LIA-3 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.

The following clauses are suggestions rather than requirements because the areas covered are the responsibility of the various 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
C
C++
Fortran
Haskell
Java
Common Lisp
ISLisp
Modula-2
PL/I
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 $\star$.)

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* [7].

An implementation should follow all the requirements of LIA-3 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-3 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.

The LIA-3 complex integer operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $itimes_{i(I)}(x)$ | `..`$x$ | † |
| $itimes_{c(I)}(x)$ | `..`$x$ | † |
| $re_I(x)$ | `re(`$x$`)` | † |
| $re_{i(I)}(x)$ | `re(`$x$`)` | † |
| $re_{c(I)}(x)$ | `re(`$x$`)` | † |
| $im_I(x)$ | `im(`$x$`)` | † |
| $im_{i(I)}(x)$ | `im(`$x$`)` | † |
| $im_{c(I)}(x)$ | `im(`$x$`)` | † |
| $plusitimes_I(x, y)$ | `..`$x$ $y$ | † |
| $neg_{i(I)}(x)$ | `-`$x$ | † |
| $neg_{c(I)}(x)$ | `-`$x$ | † |
| $conj_I(x)$ | `conj(`$x$`)` | † |
| $conj_{i(I)}(x)$ | `conj(`$x$`)` | † |
| $conj_{c(I)}(x)$ | `conj(`$x$`)` | † |
| $add_{i(I)}(x, y)$ | $x$ `+` $y$ | † |

| | | |
|---|---|---|
| $add_{c(I)}(x, y)$ | $x$ `+` $y$ | † |
| $sub_{i(I)}(x, y)$ | $x$ `-` $y$ | † |
| $sub_{c(I)}(x, y)$ | $x$ `-` $y$ | † |
| $mul_{i(I)}(x, y)$ | $x$ `*` $y$ | † |
| $mul_{c(I)}(x, y)$ | $x$ `*` $y$ | † |
| $.eq_{i(I)}(x, y)$ | $x$ `=` $y$ | † |
| $eq_{c(I)}(x, y)$ | $x$ `=` $y$ | † |
| $.neq_{i(I)}(x, y)$ | $x$ `/=` $y$ | † |
| $neq_{c(I)}(x, y)$ | $x$ `/=` $y$ | † |
| $.abs_{i(I)}(x)$ | `abs(`$x$`)` | † |
| $.sign_{i(I)}(x)$ | `sign(`$x$`)` | † |

where $x$ and $y$ are expressions of type *CINT*.

The LIA-3 basic complex floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $itimes_F(x)$ | `i *` $x$ or `j *` $x$ | † |
| $re_F(x)$ | `Re(`$x$`)` | † |
| $re_{i(F)}(x)$ | `Re(`$x$`)` | † |
| $re_{c(F)}(x)$ | `Re(`$x$`)` | ⋆ |
| $im_F(x)$ | `Im(`$x$`)` | † |
| $im_{i(F)}(x)$ | `Im(`$x$`)` | † |
| $im_{c(F)}(x)$ | `Im(`$x$`)` | ⋆ |
| $plusitimes_F(x, y)$ | `Compose_From_Cartesian(`$x$`,`$y$`)` | ⋆ |
| $plusitimes_F(x, y)$ | $x$ `+ i *` $y$ or $x$ `+ j *` $y$ | ⋆ |
| $..._F(x, y)$ | `Compose_From_Polar(`$x$`,`$y$`)` | ⋆ |
| $..._F(u, x, y)$ | `Compose_From_Polar(`$x$`,`$y$`,`$u$`)` | ⋆ |
| $neg_{i(F)}(x)$ | `-` $x$ | ⋆ |
| $neg_{c(F)}(x)$ | `-` $x$ | ⋆ |
| $conj_F(x)$ | `Conjugate(`$x$`)` | † |
| $conj_{i(F)}(x)$ | `Conjugate(`$x$`)` | ⋆ |
| $conj_{c(F)}(x)$ | `Conjugate(`$x$`)` | ⋆ |
| $.add_{F,i(F)}(x, y)$ | $x$ `+` $y$ | ⋆ |
| $.add_{F,c(F)}(x, y)$ | $x$ `+` $y$ | ⋆ |
| $.add_{i(F),F}(x, y)$ | $x$ `+` $y$ | ⋆ |
| $.add_{c(F),F}(x, y)$ | $x$ `+` $y$ | ⋆ |
| $.add_{i(F),c(F)}(x, y)$ | $x$ `+` $y$ | ⋆ |
| $.add_{c(F),i(F)}(x, y)$ | $x$ `+` $y$ | ⋆ |
| $add_{i(F)}(x, y)$ | $x$ `+` $y$ | ⋆ |
| $add_{c(F)}(x, y)$ | $x$ `+` $y$ | ⋆ |
| $.sub_{F,i(F)}(x, y)$ | $x$ `-` $y$ | ⋆ |
| $.sub_{F,c(F)}(x, y)$ | $x$ `-` $y$ | ⋆ |
| $.sub_{i(F),F}(x, y)$ | $x$ `-` $y$ | ⋆ |
| $.sub_{c(F),F}(x, y)$ | $x$ `-` $y$ | ⋆ |
| $.sub_{i(F),c(F)}(x, y)$ | $x$ `-` $y$ | ⋆ |
| $.sub_{c(F),i(F)}(x, y)$ | $x$ `-` $y$ | ⋆ |
| $sub_{i(F)}(x, y)$ | $x$ `-` $y$ | ⋆ |
| $sub_{c(F)}(x, y)$ | $x$ `-` $y$ | ⋆ |
| $.mul_{F,i(F)}(x, y)$ | $x$ `*` $y$ | ⋆ |
| $.mul_{F,c(F)}(x, y)$ | $x$ `*` $y$ | ⋆ |
| $.mul_{i(F),F}(x, y)$ | $x$ `*` $y$ | ⋆ |
| $.mul_{c(F),F}(x, y)$ | $x$ `*` $y$ | ⋆ |

| | | |
|---|---|---|
| $.mul_{i(F),c(F)}(x,y)$ | $x$ `*` $y$ | $\star$ |
| $.mul_{c(F),i(F)}(x,y)$ | $x$ `*` $y$ | $\star$ |
| $mul_{i(F)}(x,y)$ | $x$ `*` $y$ | $\star$ |
| $.div_{F,i(F)}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $.div_{F,c(F)}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $.div_{i(F),F}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $.div_{c(F),F}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $.div_{i(F),c(F)}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $.div_{c(F),i(F)}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $div_{i(F)}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $eq_{i(F)}(x,y)$ | $x$ `=` $y$ | $\star$ |
| $eq_{c(F)}(x,y)$ | $x$ `=` $y$ | $\star$ |
| $neq_{i(F)}(x,y)$ | $x$ `/=` $y$ | $\star$ |
| $neq_{c(F)}(x,y)$ | $x$ `/=` $y$ | $\star$ |
| ...lt, gt, leq, geq on imaginary... | | $\star$ |
| $abs_{i(F)}(x)$ | `abs(`$x$`)` | $\star$ |
| $abs_{c(F)}(x)$ | `abs(`$x$`)` or `Modulus(`$x$`)` | $\star$ |
| $phase_F(x)$ | `Argument(`$x$`)` | † |
| $phase_{i(F)}(x)$ | `Argument(`$x$`)` | $\star$ |
| $phase_{c(F)}(x)$ | `Argument(`$x$`)` | $\star$ |
| $.phaseu_{c(F)}(u,x)$ | `Argument(`$x$`,`$u$`)` | $\star$ |
| $sign_{i(F)}(x)$ | `Sign(`$x$`)` | † |
| $sign_{c(F)}(x)$ | `Sign(`$x$`)` | † |

where $x$, $y$, and $z$ are expressions of type *CFLT*.

The parameters for LIA-3 operations approximating real complex valued functions can be accessed by the following syntax:

| | | |
|---|---|---|
| $max\_err\_mul_{c(F)}$ | `Err_Mul(`$x$`)` | † |
| $max\_err\_div_{c(F)}$ | `Err_Div(`$x$`)` | † |
| $max\_err\_exp_{c(F)}$ | `Err_Exp(`$x$`)` | † |
| $max\_err\_power_{c(F)}$ | `Err_Power(`$x$`)` | † |
| $max\_err\_sin_{c(F)}$ | `Err_Sin(`$x$`)` | † |
| $max\_err\_tan_{c(F)}$ | `Err_Tan(`$x$`)` | † |

where $x$ is an expression of type *CFLT*. 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-3 elementary floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $mul_{c(F)}(x,y)$ | $x$ `*` $y$ | $\star$ |
| $div_{c(F)}(x,y)$ | $x$ `/` $y$ | $\star$ |
| | | |
| $.exp_{i(F)}(x)$ | `Exp(`$x$`)` | $\star$ |
| $exp_{c(F)}(x)$ | `Exp(`$x$`)` | $\star$ |
| $.power_{i(F),I}(b,a)$ | $b$ `**` $a$ | $\star$ |
| $.power_{c(F),I}(x,a)$ | $x$ `**` $a$ | $\star$ |
| $.power_{c(F),F}(b,y)$ | $b$ `**` $y$ | $\star$ |
| $.power_{F,c(F)}(b,x)$ | $b$ `**` $x$ | $\star$ |
| $power_{c(F)}(x,y)$ | $x$ `**` $y$ | $\star$ |
| | | |
| $sqrt_{c(F)}(x)$ | `Sqrt(`$x$`)` | $\star$ |

| | | | |
|---|---|---|---|
| $ln_{c(F)}(x)$ | `Log(`$x$`)` | | $\star$ |
| $.logbase_{c(F)}(b, x)$ | `Log(`$x$`, `$b$`)` | (note parameter order) | † |
| $radh_F(x)$ | `RadH(`$x$`)` | | † |
| $radh_{i(F)}(x)$ | `RadH(`$x$`)` | | † |
| $radh_{c(F)}(x)$ | `RadH(`$x$`)` | | † |
| $sinh_{i(F)}(x)$ | `SinH(`$x$`)` | | † |
| $sinh_{c(F)}(x)$ | `SinH(`$x$`)` | | $\star$ |
| $cosh_{i(F)}(x)$ | `CosH(`$x$`)` | | † |
| $cosh_{c(F)}(x)$ | `CosH(`$x$`)` | | $\star$ |
| $tanh_{i(F)}(x)$ | `TanH(`$x$`)` | | † |
| $tanh_{c(F)}(x)$ | `TanH(`$x$`)` | | $\star$ |
| $coth_{i(F)}(x)$ | `CotH(`$x$`)` | | † |
| $coth_{c(F)}(x)$ | `CotH(`$x$`)` | | $\star$ |
| $sech_{i(F)}(x)$ | `SecH(`$x$`)` | | † |
| $sech_{c(F)}(x)$ | `SecH(`$x$`)` | | † |
| $csch_{i(F)}(x)$ | `CscH(`$x$`)` | | † |
| $csch_{c(F)}(x)$ | `CscH(`$x$`)` | | † |
| $arcsinh_{i(F)}(x)$ | `ArcSinH(`$x$`)` | | † |
| $arcsinh_{c(F)}(x)$ | `ArcSinH(`$x$`)` | | $\star$ |
| $arccosh_{c(F)}(x)$ | `ArcCosH(`$x$`)` | | $\star$ |
| $arctanh_{i(F)}(x)$ | `ArcTanH(`$x$`)` | | † |
| $arctanh_{c(F)}(x)$ | `ArcTanH(`$x$`)` | | $\star$ |
| $arccoth_{i(F)}(x)$ | `ArcCotH(`$x$`)` | | † |
| $arccoth_{c(F)}(x)$ | `ArcCotH(`$x$`)` | | $\star$ |
| $arcsech_{c(F)}(x)$ | `ArcSecH(`$x$`)` | | † |
| $arccsch_{i(F)}(x)$ | `ArcCscH(`$x$`)` | | † |
| $arccsch_{c(F)}(x)$ | `ArcCscH(`$x$`)` | | † |
| $rad_{i(F)}(x)$ | `Rad(`$x$`)` | | † |
| $rad_{c(F)}(x)$ | `Rad(`$x$`)` | | † |
| $sin_{i(F)}(x)$ | `Sin(`$x$`)` | | † |
| $sin_{c(F)}(x)$ | `Sin(`$x$`)` | | $\star$ |
| $cos_{i(F)}(x)$ | `Cos(`$x$`)` | | † |
| $cos_{c(F)}(x)$ | `Cos(`$x$`)` | | $\star$ |
| $tan_{i(F)}(x)$ | `Tan(`$x$`)` | | † |
| $tan_{c(F)}(x)$ | `Tan(`$x$`)` | | $\star$ |
| $cot_{i(F)}(x)$ | `Cot(`$x$`)` | | † |
| $cot_{c(F)}(x)$ | `Cot(`$x$`)` | | $\star$ |
| $sec_{i(F)}(x)$ | `Sec(`$x$`)` | | † |
| $sec_{c(F)}(x)$ | `Sec(`$x$`)` | | † |
| $csc_{i(F)}(x)$ | `Csc(`$x$`)` | | † |
| $csc_{c(F)}(x)$ | `Csc(`$x$`)` | | † |
| $arcsin_{i(F)}(x)$ | `ArcSin(`$x$`)` | | † |
| $arcsin_{c(F)}(x)$ | `ArcSin(`$x$`)` | | $\star$ |
| $arccos_{c(F)}(x)$ | `ArcCos(`$x$`)` | | † |
| $arctan_{i(F)}(x)$ | `ArcTan(`$x$`)` | | $\star$ |

| | | |
|---|---|---|
| $arctan_{c(F)}(x)$ | `ArcTan(`$x$`)` | † |
| $arccot_{i(F)}(x)$ | `ArcCot(`$x$`)` | ⋆ |
| $arccot_{c(F)}(x)$ | `ArcCot(`$x$`)` | † |
| $arcsec_{c(F)}(x)$ | `ArcSec(`$x$`)` | † |
| $arccsc_{i(F)}(x)$ | `ArcCsc(`$x$`)` | † |
| $arccsc_{c(F)}(x)$ | `ArcCsc(`$x$`)` | † |

where $x$ and $y$ are expressions of type *CFLT*.

Arithmetic value conversions in Ada are always explicit.

| | | |
|---|---|---|
| $convert_{I\to c(I)}(x)$ | `Compose_From_Cartesian(`$x$`)` | † |
| $convert_{i(I)\to c(I)}(x)$ | `Compose_From_Cartesian(`$x$`)` | † |
| $convert_{F\to c(F)}(x)$ | `Compose_From_Cartesian(`$x$`)` | ⋆ |
| $convert_{F\to c(F)}(x)$ | $x$ `- i * 0` or $x$ `- j * 0` | ⋆ |
| $convert_{i(F)\to c(F)}(x)$ | `Compose_From_Cartesian(`$x$`)` | ⋆ |
| $convert_{i(F)\to c(F)}(x)$ | `-0 +` $x$ | ⋆ |

complex IO...

Numerals...:

| | | |
|---|---|---|
| $imaginary\_unit_F$ | `i` or `j` | ⋆ |

## C.2 C

The programming language C is defined by ISO/IEC 9899:1999, *Information technology – Programming languages – C* [13].

An implementation should follow all the requirements of LIA-3 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-3 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 `_Bool` datatype.

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.

C99 has three complex floating point datatypes: `_Complex float`, `_Complex double`, and `_Complex long double`. *CFLT* is used below to designate one of the complex floating point datatypes. These datatypes are, however, not required for all C99 implementations.

The LIA-3 complex integer operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $itimes_{i(I)}(x)$ | `..`$x$ | † |
| $itimes_{c(I)}(x)$ | `..`$x$ | † |
| $re_I(x)$ | `re(`$x$`)` | † |
| $re_{i(I)}(x)$ | `re(`$x$`)` | † |
| $re_{c(I)}(x)$ | `re(`$x$`)` | † |
| $im_I(x)$ | `im(`$x$`)` | † |
| $im_{i(I)}(x)$ | `im(`$x$`)` | † |
| $im_{c(I)}(x)$ | `im(`$x$`)` | † |
| $plusitimes_I(x,y)$ | `..`$x$ $y$ | † |
| $neg_{i(I)}(x)$ | `-`$x$ | † |
| $neg_{c(I)}(x)$ | `-`$x$ | † |

| | | |
|---|---|---|
| $conj_I(x)$ | `conj(`$x$`)` | † |
| $conj_{i(I)}(x)$ | `conj(`$x$`)` | † |
| $conj_{c(I)}(x)$ | `conj(`$x$`)` | † |
| $add_{i(I)}(x,y)$ | $x$ `+` $y$ | † |
| $add_{c(I)}(x,y)$ | $x$ `+` $y$ | † |
| $sub_{i(I)}(x,y)$ | $x$ `-` $y$ | † |
| $sub_{c(I)}(x,y)$ | $x$ `-` $y$ | † |
| $mul_{i(I)}(x,y)$ | $x$ `*` $y$ | † |
| $mul_{c(I)}(x,y)$ | $x$ `*` $y$ | † |
| $.eq_{i(I)}(x,y)$ | $x$ `=` $y$ | † |
| $eq_{c(I)}(x,y)$ | $x$ `=` $y$ | † |
| $.neq_{i(I)}(x,y)$ | $x$ `/=` $y$ | † |
| $neq_{c(I)}(x,y)$ | $x$ `/=` $y$ | † |
| $.abs_{i(I)}(x)$ | `abs(`$x$`)` | † |
| $.sign_{i(I)}(x)$ | `sign(`$x$`)` | † |

where $x$ and $y$ are expressions of type *CINT*. The LIA-3 non-transcendental complex floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $itimes_F(x)$ | `i *` $x$ or `j *` $x$ | † |
| $re_F(x)$ | `Re(`$x$`)` | † |
| $re_{i(F)}(x)$ | `Re(`$x$`)` | † |
| $re_{c(F)}(x)$ | `Re(`$x$`)` | ⋆ |
| $im_F(x)$ | `Im(`$x$`)` | † |
| $im_{i(F)}(x)$ | `Im(`$x$`)` | † |
| $im_{c(F)}(x)$ | `Im(`$x$`)` | ⋆ |
| $plusitimes_F(x,y)$ | `Compose_From_Cartesian(`$x$`,`$y$`)` | ⋆ |
| $plusitimes_F(x,y)$ | $x$ `+ i *` $y$ or $x$ `+ j *` $y$ | ⋆ |
| $..._F(x,y)$ | `Compose_From_Polar(`$x$`,`$y$`)` | ⋆ |
| $..._F(u,x,y)$ | `Compose_From_Polar(`$x$`,`$y$`,`$u$`)` | ⋆ |
| $neg_{i(F)}(x)$ | `-` $x$ | ⋆ |
| $neg_{c(F)}(x)$ | `-` $x$ | ⋆ |
| $conj_F(x)$ | `Conjugate(`$x$`)` | † |
| $conj_{i(F)}(x)$ | `Conjugate(`$x$`)` | ⋆ |
| $conj_{c(F)}(x)$ | `Conjugate(`$x$`)` | ⋆ |
| $.add_{F,i(F)}(x,y)$ | $x$ `+` $y$ | ⋆ |
| $.add_{F,c(F)}(x,y)$ | $x$ `+` $y$ | ⋆ |
| $.add_{i(F),F}(x,y)$ | $x$ `+` $y$ | ⋆ |
| $.add_{c(F),F}(x,y)$ | $x$ `+` $y$ | ⋆ |
| $.add_{i(F),c(F)}(x,y)$ | $x$ `+` $y$ | ⋆ |
| $.add_{c(F),i(F)}(x,y)$ | $x$ `+` $y$ | ⋆ |
| $add_{i(F)}(x,y)$ | $x$ `+` $y$ | ⋆ |
| $add_{c(F)}(x,y)$ | $x$ `+` $y$ | ⋆ |
| $.sub_{F,i(F)}(x,y)$ | $x$ `-` $y$ | ⋆ |
| $.sub_{F,c(F)}(x,y)$ | $x$ `-` $y$ | ⋆ |
| $.sub_{i(F),F}(x,y)$ | $x$ `-` $y$ | ⋆ |
| $.sub_{c(F),F}(x,y)$ | $x$ `-` $y$ | ⋆ |
| $.sub_{i(F),c(F)}(x,y)$ | $x$ `-` $y$ | ⋆ |
| $.sub_{c(F),i(F)}(x,y)$ | $x$ `-` $y$ | ⋆ |
| $sub_{i(F)}(x,y)$ | $x$ `-` $y$ | ⋆ |
| $sub_{c(F)}(x,y)$ | $x$ `-` $y$ | ⋆ |
| $.mul_{F,i(F)}(x,y)$ | $x$ `*` $y$ | ⋆ |

| | | |
|---|---|---|
| $.mul_{F,c(F)}(x,y)$ | $x$ `*` $y$ | $\star$ |
| $.mul_{i(F),F}(x,y)$ | $x$ `*` $y$ | $\star$ |
| $.mul_{c(F),F}(x,y)$ | $x$ `*` $y$ | $\star$ |
| $.mul_{i(F),c(F)}(x,y)$ | $x$ `*` $y$ | $\star$ |
| $.mul_{c(F),i(F)}(x,y)$ | $x$ `*` $y$ | $\star$ |
| $mul_{i(F)}(x,y)$ | $x$ `*` $y$ | $\star$ |
| $.div_{F,i(F)}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $.div_{F,c(F)}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $.div_{i(F),F}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $.div_{c(F),F}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $.div_{i(F),c(F)}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $.div_{c(F),i(F)}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $div_{i(F)}(x,y)$ | $x$ `/` $y$ | $\star$ |
| $eq_{i(F)}(x,y)$ | $x$ `=` $y$ | $\star$ |
| $eq_{c(F)}(x,y)$ | $x$ `=` $y$ | $\star$ |
| $neq_{i(F)}(x,y)$ | $x$ `/=` $y$ | $\star$ |
| $neq_{c(F)}(x,y)$ | $x$ `/=` $y$ | $\star$ |
| ...lt, gt, leq, geq on imaginary... | | $\star$ |
| $abs_{i(F)}(x)$ | `abs(`$x$`)` | $\star$ |
| $abs_{c(F)}(x)$ | `abs(`$x$`)` or `Modulus(`$x$`)` | $\star$ |
| $phase_{F}(x)$ | `Argument(`$x$`)` | † |
| $phase_{i(F)}(x)$ | `Argument(`$x$`)` | $\star$ |
| $phase_{c(F)}(x)$ | `Argument(`$x$`)` | $\star$ |
| $.phaseu_{c(F)}(u,x)$ | `Argument(`$x$`,`$u$`)` | $\star$ |
| $sign_{i(F)}(x)$ | `Sign(`$x$`)` | † |
| $sign_{c(F)}(x)$ | `Sign(`$x$`)` | † |

where $x$, $y$ and $z$ are expressions of the same complex floating point type.

The LIA-3 parameters for operations approximating complex real valued transcendental functions can be accessed by the following syntax:

| | | |
|---|---|---|
| $max\_err\_mul_{c(F)}$ | `err_cmul`$t$ | † |
| $max\_err\_div_{c(F)}$ | `err_cdiv`$t$ | † |
| $max\_err\_exp_{c(F)}$ | `err_cexp`$t$ | † |
| $max\_err\_power_{c(F)}$ | `err_cpower`$t$ | † |
| $max\_err\_sin_{c(F)}$ | `err_csin`$t$ | † |
| $max\_err\_tan_{c(F)}$ | `err_ctan`$t$ | † |

where $t$ ...

The LIA-3 elementary complex floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $mul_{c(F)}(x,y)$ | $x$ `*` $y$ | $\star$ |
| $div_{c(F)}(x,y)$ | $x$ `/` $y$ | $\star$ |
| | | |
| $power_{c(F)I}(b,z)$ | `cpoweri`$t$`(`$b$`,` $z$`)` | † |
| $exp_{c(F)}(x)$ | `cexp`$t$`(`$x$`)` | $\star$ |
| $power_{c(F)}(b,y)$ | `cpower`$t$`(`$b$`,` $y$`)` | † |
| $pow_{c(F)}(b,y)$ | `cpow`$t$`(`$b$`,` $y$`)` | $\star$ Not LIA-3! |
| | | |
| $ln_{c(F)}(x)$ | `clog`$t$`(`$x$`)` | $\star$ |
| $.logbase_{c(F)}(b,x)$ | `clogbase`$t$`(`$b$`,` $x$`)` | † |

| | | |
|---|---|---|
| $sinh_{i(F)}(x)$ | sinh$t(x)$ | $\star$ |
| $sinh_{c(F)}(x)$ | sinh$t(x)$ | $\star$ |
| $cosh_{i(F)}(x)$ | cosh$t(x)$ | $\star$ |
| $cosh_{c(F)}(x)$ | cosh$t(x)$ | $\star$ |
| $tanh_{i(F)}(x)$ | tanh$t(x)$ | $\star$ |
| $tanh_{c(F)}(x)$ | tanh$t(x)$ | $\star$ |
| $coth_{i(F)}(x)$ | coth$t(x)$ | † |
| $coth_{c(F)}(x)$ | coth$t(x)$ | † |
| $sech_{i(F)}(x)$ | sech$t(x)$ | † |
| $sech_{c(F)}(x)$ | sech$t(x)$ | † |
| $csch_{i(F)}(x)$ | csch$t(x)$ | † |
| $csch_{c(F)}(x)$ | csch$t(x)$ | † |
| | | |
| $radh_{F}(x)$ | radianh$t(x)$ | † |
| $radh_{i(F)}(x)$ | radianh$t(x)$ | † |
| $radh_{c(F)}(x)$ | radianh$t(x)$ | † |
| | | |
| $arcsinh_{i(F)}(x)$ | asinh$t(x)$ | $\star$ |
| $arcsinh_{c(F)}(x)$ | asinh$t(x)$ | $\star$ |
| $arccosh_{c(F)}(x)$ | acosh$t(x)$ | $\star$ |
| $arctanh_{i(F)}(x)$ | atanh$t(x)$ | $\star$ |
| $arctanh_{c(F)}(x)$ | atanh$t(x)$ | $\star$ |
| $arccoth_{i(F)}(x)$ | acoth$t(x)$ | † |
| $arccoth_{c(F)}(x)$ | acoth$t(x)$ | † |
| $arcsech_{c(F)}(x)$ | asech$t(x)$ | † |
| $arccsch_{i(F)}(x)$ | acsch$t(x)$ | † |
| $arccsch_{c(F)}(x)$ | acsch$t(x)$ | † |
| | | |
| $rad_{i(F)}(x)$ | radian$t(x)$ | † |
| $rad_{c(F)}(x)$ | radian$t(x)$ | † |
| | | |
| $sin_{i(F)}(x)$ | sin$t(x)$ | $\star$ |
| $sin_{c(F)}(x)$ | sin$t(x)$ | $\star$ |
| $cos_{i(F)}(x)$ | cos$t(x)$ | $\star$ |
| $cos_{c(F)}(x)$ | cos$t(x)$ | $\star$ |
| $tan_{i(F)}(x)$ | tan$t(x)$ | $\star$ |
| $tan_{c(F)}(x)$ | tan$t(x)$ | $\star$ |
| $cot_{i(F)}(x)$ | cot$t(x)$ | † |
| $cot_{c(F)}(x)$ | cot$t(x)$ | † |
| $sec_{i(F)}(x)$ | sec$t(x)$ | † |
| $sec_{c(F)}(x)$ | sec$t(x)$ | † |
| $csc_{i(F)}(x)$ | csc$t(x)$ | † |
| $csc_{c(F)}(x)$ | csc$t(x)$ | † |
| | | |
| $arcsin_{i(F)}(x)$ | asin$t(x)$ | $\star$ |
| $arcsin_{c(F)}(x)$ | asin$t(x)$ | $\star$ |
| $arccos_{c(F)}(x)$ | acos$t(x)$ | $\star$ |
| $arctan_{i(F)}(x)$ | atan$t(x)$ | $\star$ |
| $arctan_{c(F)}(x)$ | atan$t(x)$ | $\star$ |
| $arccot_{i(F)}(x)$ | acot$t(x)$ | † |
| $arccot_{c(F)}(x)$ | acot$t(x)$ | † |

| | | |
|---|---|---|
| $arcsec_{c(F)}(x)$ | `asec`$t(x)$ | † |
| $arccsc_{i(F)}(x)$ | `acsc`$t(x)$ | † |
| $arccsc_{c(F)}(x)$ | `acsc`$t(x)$ | † |

where $b$, $x$, and $y$ are expressions of the same complex floating point type. $t$ is a string, part of the operation name, and is "`f`" for `_Complex float`, the empty string for `_Complex double`, and "`l`" for `_Complex long float`.

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 strings. The rules for when implicit conversions are applied is not repeated here, but work as if a cast had been applied.

| | | |
|---|---|---|
| $convert_{I \to c(I)}(x)$ | `(`$x$`)` | † |
| $convert_{i(I) \to c(I)}(x)$ | `(`$x$`)` | † |
| $convert_{F \to c(F)}(x)$ | $x$ `- I * 0` or $x$ `- _IMAGINARY_I * 0` | ⋆ |
| $convert_{i(F) \to c(F)}(x)$ | `-0 +` $x$ | ⋆ |
| complex IO... | | |

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 $I'$. A ? above indicates that the parameter is optional. $e$ is greater than 0.

Numerals...:

| | | |
|---|---|---|
| $imaginary\_unit_F$ | `I` or `_IMAGINARY_I` | ⋆ |

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-3, which can be used for floating point errors. For C99, the **absolute_precision_underflow** notification is ignored.

## C.3 C++

The programming language C++ is defined by ISO/IEC 14882:1998, *Programming languages – C++* [14].

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-1 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:

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:

where $x$, $y$ and $z$ are expressions of the same floating point type, and where $xs$ is an expression of type valarray of a floating point type.

The parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

| | | |
|---|---|---|
| $max\_err\_mul_{c(F)}$ | `err_mul<CFLT>()` | † |
| $max\_err\_div_{c(F)}$ | `err_div<CFLT>()` | † |
| $max\_err\_exp_{c(F)}$ | `err_exp<CFLT>()` | † |
| $max\_err\_power_{c(F)}$ | `err_power<CFLT>()` | † |
| $max\_err\_sin_{c(F)}$ | `err_sin<CFLT>()` | † |
| $max\_err\_tan_{c(F)}$ | `err_tan<CFLT>()` | † |

where $u$ is an expression of a floating point type. Several of the parameter functions are constant for each type (and library).

The LIA-2 elementary floating point operations are listed below, along with the syntax (type generic macros) used to invoke them:

| | | |
|---|---|---|
| $exp_{c(F)}(x)$ | `exp(`$x$`)` | ⋆ |
| $power_{c(F)}(b, y)$ | `power(`$b$`, `$y$`)` | † |
| $pow_{c(F)}(b, y)$ | `pow(`$b$`, `$y$`)` | ⋆ Not LIA-2! (See C.) |
| $ln_{c(F)}(x)$ | `log(`$x$`)` | ⋆ |
| $logbase_{c(F)}(b, x)$ | `logbase(`$b$`, `$x$`)` | † |
| $radh_{c(F)}(x)$ | `radh(`$x$`)` | † |
| $sinh_{c(F)}(x)$ | `sinh(`$x$`)` | ⋆ |
| $cosh_{c(F)}(x)$ | `cosh(`$x$`)` | ⋆ |
| $tanh_{c(F)}(x)$ | `tanh(`$x$`)` | ⋆ |
| $coth_{c(F)}(x)$ | `coth(`$x$`)` | † |
| $sech_{c(F)}(x)$ | `sech(`$x$`)` | † |
| $csch_{c(F)}(x)$ | `csch(`$x$`)` | † |
| $arcsinh_{c(F)}(x)$ | `asinh(`$x$`)` | ⋆ |
| $arccosh_{c(F)}(x)$ | `acosh(`$x$`)` | ⋆ |
| $arctanh_{c(F)}(x)$ | `atanh(`$x$`)` | ⋆ |
| $arccoth_{c(F)}(x)$ | `acoth(`$x$`)` | † |
| $arcsech_{c(F)}(x)$ | `asech(`$x$`)` | † |
| $arccsch_{c(F)}(x)$ | `acsch(`$x$`)` | † |
| $rad_{c(F)}(x)$ | `rad(`$x$`)` | † |

| | | |
|---|---|---|
| $sin_{c(F)}(x)$ | `sin`$(x)$ | $\star$ |
| $cos_{c(F)}(x)$ | `cos`$(x)$ | $\star$ |
| $tan_{c(F)}(x)$ | `tan`$(x)$ | $\star$ |
| $cot_{c(F)}(x)$ | `cot`$(x)$ | † |
| $sec_{c(F)}(x)$ | `sec`$(x)$ | † |
| $csc_{c(F)}(x)$ | `csc`$(x)$ | † |
| | | |
| $arcsin_{c(F)}(x)$ | `asin`$(x)$ | $\star$ |
| $arccos_{c(F)}(x)$ | `acos`$(x)$ | $\star$ |
| $arctan_{c(F)}(x)$ | `atan`$(x)$ | $\star$ |
| $arccot_{c(F)}(x)$ | `acot`$(x)$ | † |
| $arcctg_{c(F)}(x)$ | `actg`$(x)$ | † |
| $arcsec_{c(F)}(x)$ | `asec`$(x)$ | † |
| $arccsc_{c(F)}(x)$ | `acsc`$(x)$ | † |

where $b$, $x$, $y$, $u$, and $v$ are expressions of type *CFLT*.

Arithmetic value conversions in C++ are 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.

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 $I'$. A ? above indicates that the parameter is optional. $e$ is greater than 0.

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 '.' in them. The details are not repeated in this example binding, see ISO/IEC 14882, clause 2.9.1 Integer literals, and clause 2.9.4 Floating literals.

## C.4 Fortran

The programming language Fortran is defined by ISO/IEC 1539-1:1997, *Information technology – Programming languages – Fortran – Part 1: Base language* [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.

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 data type 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, parameterised with the `KIND` parameter.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(I)}(x, y)$ | $x$ `+` $y$ | † |
| $sub_{c(I)}(x, y)$ | $x$ `-` $y$ | † |
| $mul_{c(I)}(x, y)$ | $x$ `*` $y$ | † |

where $x$ and $y$ are expressions of type `CINTEGER` and where $xs$ is an expression of type array of `CINTEGER`.

The additional non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(F)}(x, y)$ | $x$ `+` $y$ | ⋆ |
| $sub_{c(F)}(x, y)$ | $x$ `-` $y$ | ⋆ |
| $mul_{c(F)}(x, y)$ | $x$ `*` $y$ | ⋆ |

where $x$, $y$ and $z$ are expressions of type *FLT*, and where $xs$ is an expression of type array of *FLT*.

The LIA-3 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

| | | |
|---|---|---|
| $max\_err\_mul_F$ | `ERR_MUL(`$x$`)` | † |
| $max\_err\_div_F$ | `ERR_DIV(`$x$`)` | † |
| $max\_err\_exp_F$ | `ERR_EXP(`$x$`)` | † |
| $max\_err\_power_F$ | `ERR_POWER(`$x$`)` | † |
| $max\_err\_sin_F$ | `ERR_SIN(`$x$`)` | † |
| $max\_err\_tan_F$ | `ERR_TAN(`$x$`)` | † |

where $b$, $x$ and $u$ are expressions of type *CFLT*. 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-3 elementary floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $div_{c(F)}(x, y)$ | $x$ `/` $y$ | ⋆ |
| $sqrt_{c(F)}(x)$ | `SQRT(`$x$`)` | ⋆ |
| $exp_{c(F)}(x)$ | `EXP(`$x$`)` | ⋆ |
| $power_{c(F)}(b, y)$ | $b$ `**` $y$ | ⋆ |
| $ln_{c(F)}(x)$ | `LOG(`$x$`)` | ⋆ |
| $logbase_{c(F)}(b, x)$ | `LOGBASE(`$b$`,` $x$`)` | † |
| $sinh_{c(F)}(x)$ | `SINH(`$x$`)` | ⋆ |
| $cosh_{c(F)}(x)$ | `COSH(`$x$`)` | ⋆ |
| $tanh_{c(F)}(x)$ | `TANH(`$x$`)` | ⋆ |
| $coth_{c(F)}(x)$ | `COTH(`$x$`)` | † |
| $sech_{c(F)}(x)$ | `SECH(`$x$`)` | † |
| $csch_{c(F)}(x)$ | `CSCH(`$x$`)` | † |
| $radh_{c(F)}(x)$ | `RADH(`$x$`)` | † |

| | | |
|---|---|---|
| $arcsinh_{c(F)}(x)$ | `ASINH(`$x$`)` | † |
| $arccosh_{c(F)}(x)$ | `ACOSH(`$x$`)` | † |
| $arctanh_{c(F)}(x)$ | `ATANH(`$x$`)` | † |
| $arccoth_{c(F)}(x)$ | `ACOTH(`$x$`)` | † |
| $arcsech_{c(F)}(x)$ | `ASECH(`$x$`)` | † |
| $arccsch_{c(F)}(x)$ | `ACSCH(`$x$`)` | † |
| | | |
| $rad_{c(F)}(x)$ | `RAD(`$x$`)` | † |
| | | |
| $sin_{c(F)}(x)$ | `SIN(`$x$`)` | ⋆ |
| $cos_{c(F)}(x)$ | `COS(`$x$`)` | ⋆ |
| $tan_{c(F)}(x)$ | `TAN(`$x$`)` | ⋆ |
| $cot_{c(F)}(x)$ | `COT(`$x$`)` | † |
| $sec_{c(F)}(x)$ | `SEC(`$x$`)` | † |
| $csc_{c(F)}(x)$ | `CSC(`$x$`)` | † |
| | | |
| $arcsin_{c(F)}(x)$ | `ASIN(`$x$`)` | ⋆ |
| $arccos_{c(F)}(x)$ | `ACOS(`$x$`)` | ⋆ |
| $arctan_{c(F)}(x)$ | `ATAN(`$x$`)` | ⋆ |
| $arccot_{c(F)}(x)$ | `ACOT(`$x$`)` | † |
| $arcctg_{c(F)}(x)$ | `ACTG(`$x$`)` | † |
| $arcsec_{c(F)}(x)$ | `ASEC(`$x$`)` | † |
| $arccsc_{c(F)}(x)$ | `ACSC(`$x$`)` | † |

where $b$, $x$, $y$, $u$, and $v$ are expressions of type *CFLT*.

Arithmetic value conversions in Fortran are always explicit, and the conversion function is named like the target type, except when converting to/from strings.

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 $I'$.

## C.5 Haskell

The programming language Haskell is defined by *Report on the programming language Haskell 98* [56], together with *Standard libraries for the Haskell 98 programming panguage* [57].

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-3 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 unlimited, 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.

The LIA-2 integer operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(I)}(x, y)$ | $x$ `+` $y$ or `(+)` $x$ $y$ | † |
| $sub_{c(I)}(x, y)$ | $x$ `-` $y$ or `(-)` $x$ $y$ | † |
| $mul_{c(I)}(x, y)$ | $x$ `*` $y$ or `(*)` $x$ $y$ | † |

where $x$ and $y$ are expressions of type *complex INT* [not in Haskell98].

The LIA-3 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(F)}(x, y)$ | $x$ `+` $y$ or `(+)` $x$ $y$ | ⋆ |
| $sub_{c(F)}(x, y)$ | $x$ `-` $y$ or `(-)` $x$ $y$ | ⋆ |

where $x$, $y$ and $z$ are expressions of type `complex` *FLT*.

The LIA-3 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

| | | |
|---|---|---|
| $max\_err\_mul_F$ | `err_mul` $x$ | † |
| $max\_err\_div_F$ | `err_div` $x$ | † |
| $max\_err\_exp_F$ | `err_exp` $x$ | † |
| $max\_err\_power_F$ | `err_power` $x$ | † |
| $max\_err\_sin_F$ | `err_sin` $x$ | † |
| $max\_err\_tan_F$ | `err_tan` $x$ | † |

where $x$ is an expression of type `complex` *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-3 elementary floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $mul_{c(F)}(x, y)$ | $x$ `*` $y$ or `(*)` $x$ $y$ | ⋆ |
| $sqrt_{c(F)}(x)$ | `sqrt` $x$ | ⋆ |
| $exp_{c(F)}(x)$ | `exp` $x$ | ⋆ |
| $power_{c(F)}(b, y)$ | $b$ `**` $y$ or `(**)` $b$ $y$ | ⋆ |
| $ln_{c(F)}(x)$ | `log` $x$ | ⋆ |
| $logbase_{c(F)}(b, x)$ | `logBase` $b$ $x$ or $b$ ‘`logBase`‘ $x$ | ⋆ |
| $radh_{c(F)}(x)$ | `hypradians` $x$ | † |
| $sinh_{c(F)}(x)$ | `sinh` $x$ | ⋆ |
| $cosh_{c(F)}(x)$ | `cosh` $x$ | ⋆ |
| $tanh_{c(F)}(x)$ | `tanh` $x$ | ⋆ |
| $coth_{c(F)}(x)$ | `coth` $x$ | † |
| $sech_{c(F)}(x)$ | `sech` $x$ | † |
| $csch_{c(F)}(x)$ | `csch` $x$ | † |
| $arcsinh_{c(F)}(x)$ | `asinh` $x$ | ⋆ |
| $arccosh_{c(F)}(x)$ | `acosh` $x$ | ⋆ |

| | | |
|---|---|---|
| $arctanh_{c(F)}(x)$ | `atanh` $x$ | $\star$ |
| $arccoth_{c(F)}(x)$ | `acoth` $x$ | † |
| $arcsech_{c(F)}(x)$ | `asech` $x$ | † |
| $arccsch_{c(F)}(x)$ | `acsch` $x$ | † |
| | | |
| $rad_{c(F)}(x)$ | `radians` $x$ | † |
| | | |
| $sin_{c(F)}(x)$ | `sin` $x$ | $\star$ |
| $cos_{c(F)}(x)$ | `cos` $x$ | $\star$ |
| $tan_{c(F)}(x)$ | `tan` $x$ | $\star$ |
| $cot_{c(F)}(x)$ | `cot` $x$ | † |
| $sec_{c(F)}(x)$ | `sec` $x$ | † |
| $csc_{c(F)}(x)$ | `csc` $x$ | † |
| | | |
| $arcsin_{c(F)}(x)$ | `asin` $x$ | $\star$ |
| $arccos_{c(F)}(x)$ | `acos` $x$ | $\star$ |
| $arctan_{c(F)}(x)$ | `atan` $x$ | $\star$ |
| $arccot_{c(F)}(x)$ | `acot` $x$ | † |
| $arcctg_{c(F)}(x)$ | `actg` $x$ | † |
| $arcsec_{c(F)}(x)$ | `asec` $x$ | † |
| $arccsc_{c(F)}(x)$ | `acsc` $x$ | † |

where $b$, $x$, $y$, $u$, and $v$ are expressions of type *complex FLT*.

Arithmetic value conversions in Haskell are always explicit. They are done with the overloaded `fromIntegral` and `fromFractional` operations.

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.

## C.6 Java

The programming language Java is defined by *The Java Language Specification* [55].

An implementation should follow all the requirements of LIA-3 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-3 for that operation. For each of the marked items a suggested identifier is provided. The LIA-3 operations that are provided in Java 2 (marked "$\star$" below) are in the final class `java.lang.Math` [not in Java 2...].

The Java datatype `boolean` corresponds to the LIA datatype **Boolean**.

Every implementation of Java has the integral datatypes `int`, and `long`.

Java has two floating point datatypes, `float` and `double`, which must conform to IEC 60559.

The LIA-3 integer operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(I)}(x, y)$ | `add(`$x$`, `$y$`)` | $\star$ |
| $sub_{c(I)}(x, y)$ | `sub(`$x$`, `$y$`)` | $\star$ |
| $mul_{c(I)}(x, y)$ | `mul(`$x$`, `$y$`)` | $\star$ |

where $x$ and $y$ are expressions of type *CINT*.

The LIA-3 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(F)}(x, y)$ | `add(`$x$`, `$y$`)` | $\star$ |
| $sub_{c(F)}(x, y)$ | `sub(`$x$`, `$y$`)` | $\star$ |

where $x$, $y$ and $z$ 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:

| | | |
|---|---|---|
| $max\_err\_mul_{c(F)}$ | `err_mul(`$x$`)` | † |
| $max\_err\_div_{c(F)}$ | `err_div(`$x$`)` | † |
| | | |
| $max\_err\_exp_{c(F)}$ | `err_exp(`$x$`)` | † |
| $max\_err\_power_{c(F)}(b, x)$ | `err_power(`$b$`, `$x$`)` | † |
| | | |
| $max\_err\_sin_{c(F)}$ | `err_sin(`$x$`)` | † |
| $max\_err\_tan_{c(F)}$ | `err_tan(`$x$`)` | † |

where $b$, $x$ 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`.

| | | |
|---|---|---|
| $mul_{c(F)}(x, y)$ | `mul(`$x$`, `$y$`)` | $\star$ |
| $div_{c(F)}(x, y)$ | `div(`$x$`, `$y$`)` | $\star$ |
| | | |
| $exp_{c(F)}(x)$ | `exp(`$x$`)` | $\star$ |
| $power_{c(F)}(b, y)$ | `power(`$b$`, `$y$`)` | † |
| $pow_{c(F)}(b, y)$ | `pow(`$b$`, `$y$`)` | $\star$ Not LIA-2! |
| | | |
| $ln_{c(F)}(x)$ | `log(`$x$`)` | $\star$ |
| $logbase_{c(F)}(b, x)$ | `log(`$b$`, `$x$`)` | † |
| | | |
| $radh_{c(F)}(x)$ | `hypradian(`$x$`)` | † |
| | | |
| $sinh_{c(F)}(x)$ | `sinh(`$x$`)` | † |
| $cosh_{c(F)}(x)$ | `cosh(`$x$`)` | † |
| $tanh_{c(F)}(x)$ | `tanh(`$x$`)` | † |
| $coth_{c(F)}(x)$ | `coth(`$x$`)` | † |
| $sech_{c(F)}(x)$ | `sech(`$x$`)` | † |
| $csch_{c(F)}(x)$ | `csch(`$x$`)` | † |
| | | |
| $arcsinh_{c(F)}(x)$ | `asinh(`$x$`)` | † |
| $arccosh_{c(F)}(x)$ | `acosh(`$x$`)` | † |
| $arctanh_{c(F)}(x)$ | `atanh(`$x$`)` | † |
| $arccoth_{c(F)}(x)$ | `acoth(`$x$`)` | † |
| $arcsech_{c(F)}(x)$ | `asech(`$x$`)` | † |

| | | |
|---|---|---|
| $arccsch_{c(F)}(x)$ | `acsch(`$x$`)` | † |
| $rad_{c(F)}(x)$ | `radian(`$x$`)` | † |
| $sin_{c(F)}(x)$ | `sin(`$x$`)` | ⋆ |
| $cos_{c(F)}(x)$ | `cos(`$x$`)` | ⋆ |
| $tan_{c(F)}(x)$ | `tan(`$x$`)` | ⋆ |
| $cot_{c(F)}(x)$ | `cot(`$x$`)` | † |
| $sec_{c(F)}(x)$ | `sec(`$x$`)` | † |
| $csc_{c(F)}(x)$ | `csc(`$x$`)` | † |
| $arcsin_{c(F)}(x)$ | `asin(`$x$`)` | ⋆ |
| $arccos_{c(F)}(x)$ | `acos(`$x$`)` | ⋆ |
| $arctan_{c(F)}(x)$ | `atan(`$x$`)` | ⋆ |
| $arccot_{c(F)}(x)$ | `acot(`$x$`)` | † |
| $arcctg_{c(F)}(x)$ | `actg(`$x$`)` | † |
| $arcsec_{c(F)}(x)$ | `asec(`$x$`)` | † |
| $arccsc_{c(F)}(x)$ | `acsc(`$x$`)` | † |

where $b$, $x$, $y$, $u$, and $v$ are expressions of type *CFLT*.

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.

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 $I'$. A ? above indicates that the parameter is optional. $e$ is greater than 0.

## C.7 Common Lisp

The programming language Common Lisp is defined by ANSI X3.226-1994, *Information Technology – Programming Language – Common Lisp* [38].

An implementation should follow all the requirements of LIA-3 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 types: `short-float`, `single-float`, `double-float`, and `long-float`. Not all of these floating point types must be distinct.

The additional integer operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(I)}(x, y)$ | `(+` $x$ $y$`)` | $\star$ |
| $sub_{c(I)}(x, y)$ | `(-` $x$ $y$`)` | $\star$ |
| $mul_{c(I)}(x, y)$ | `(*` $x$ $y$`)` | $\star$ |

where $x$ and $y$ are expressions of type *CINT*.

The LIA-3 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(F)}(x, y)$ | `(+` $x$ $y$`)` | $\star$ |
| $sub_{c(F)}(x, y)$ | `(-` $x$ $y$`)` | $\star$ |

where $x$, $y$ and $z$ are data objects of the same floating point type, and where $xs$ is a data objects that is a list of data objects of (the same, in this binding) floating point type. Note that Common Lisp allows mixed number types 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:

| | | |
|---|---|---|
| $max\_err\_mul_F$ | `(err-mul` $x$`)` | † |
| $max\_err\_div_F$ | `(err-div` $x$`)` | † |
| $max\_err\_exp_F$ | `(err-exp` $x$`)` | † |
| $max\_err\_power_F$ | `(err-power` $x$`)` | † |
| $max\_err\_sin_F$ | `(err-sin` $x$`)` | † |
| $max\_err\_tan_F$ | `(err-tan` $x$`)` | † |

where $b$, $x$ and $u$ are expressions of type *CFLT*. 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-3 elementary floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $mul_{c(F)}(x, y)$ | `(*` $x$ $y$`)` | $\star$ |
| $div_{c(F)}(x, y)$ | `(/` $x$ $y$`)` | $\star$ |
| $sqrt_{c(F)}(x)$ | `(sqrt` $x$`)` | $\star$ |
| $exp_{c(F)}(x)$ | `(exp` $x$`)` | $\star$ |
| $power_{c(F)}(b, y)$ | `(expt` $b$ $y$`)` (deviation: `(expt 0.0 0.0)` is 1) | $\star$ |
| $ln_{c(F)}(x)$ | `(log` $x$`)` | $\star$ |
| $logbase_{c(F)}(b, x)$ | `(log` $x$ $b$`)` (note parameter order) | $\star$ |
| $radh_{c(F)}(x)$ | `(hypradians` $x$`)` | † |
| $sinh_{c(F)}(x)$ | `(sinh` $x$`)` | $\star$ |
| $cosh_{c(F)}(x)$ | `(cosh` $x$`)` | $\star$ |
| $tanh_{c(F)}(x)$ | `(tanh` $x$`)` | $\star$ |
| $coth_F(x)$ | `(coth` $x$`)` | † |

| | | |
|---|---|---|
| $sech_F(x)$ | `(sech` $x$`)` | † |
| $csch_{c(F)}(x)$ | `(csch` $x$`)` | † |
| | | |
| $arcsinh_{c(F)}(x)$ | `(asinh` $x$`)` | ⋆ |
| $arccosh_{c(F)}(x)$ | `(acosh` $x$`)` | ⋆ |
| $arctanh_{c(F)}(x)$ | `(atanh` $x$`)` | ⋆ |
| $arccoth_{c(F)}(x)$ | `(acoth` $x$`)` | † |
| $arcsech_{c(F)}(x)$ | `(asech` $x$`)` | † |
| $arccsch_{c(F)}(x)$ | `(acsch` $x$`)` | † |
| | | |
| $rad_{c(F)}(x)$ | `(radians` $x$`)` | † |
| | | |
| $sin_{c(F)}(x)$ | `(sin` $x$`)` | ⋆ |
| $cos_{c(F)}(x)$ | `(cos` $x$`)` | ⋆ |
| $tan_{c(F)}(x)$ | `(tan` $x$`)` | ⋆ |
| $cot_{c(F)}(x)$ | `(cot` $x$`)` | † |
| $sec_{c(F)}(x)$ | `(sec` $x$`)` | † |
| $csc_{c(F)}(x)$ | `(csc` $x$`)` | † |
| | | |
| $arcsin_{c(F)}(x)$ | `(asin` $x$`)` | ⋆ |
| $arccos_{c(F)}(x)$ | `(acos` $x$`)` | ⋆ |
| $arctan_{c(F)}(x)$ | `(atan` $x$`)` | ⋆ |
| $arccot_{c(F)}(x)$ | `(acot` $x$`)` | † |
| $arcctg_{c(F)}(x)$ | `(actg` $x$`)` | † |
| $arcsec_{c(F)}(x)$ | `(asec` $x$`)` | † |
| $arccsc_{c(F)}(x)$ | `(acsc` $x$`)` | † |

where $b$, $x$, $y$, $u$, and $v$ are expressions of type *CFLT*.

Arithmetic value conversions in Common Lisp are can be explicit or implicit. The rules for when implicit conversions are done is implementation defined.

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. Convertion from string to numeric value is in Common Lisp done via a general read procedure, which reads Common Lisp 'S-expressions'.

## C.8 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* [20].

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 a ISLisp data object, subject only to total memory limitations.

ISLisp has one floating point type required to conform to IEC 60559.

The additional integer operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $min_I(x, y)$ | `(min` $x$ $y$`)` | $\star$ |
| $max_I(x, y)$ | `(max` $x$ $y$`)` | $\star$ |
| $min\_seq_I(xs)$ | `(min .` $xs$`)` or `(min` $x_1$ $x_2$ ... $x_n$`)` | $\star$ |
| $max\_seq_I(xs)$ | `(max .` $xs$`)` or `(max` $x_1$ $x_2$ ... $x_n$`)` | $\star$ |
| $dim_I(x, y)$ | `(dim` $x$ $y$`)` | † |
| $sqrt_I(x)$ | `(isqrt` $x$`)` | $\star$ |
| $power_I(x, y)$ | `(expt` $x$ $y$`)` (deviation: `(expt 0 0)` is 1) | $\star$ |
| $shift2_I(x, y)$ | `(shift2` $x$ $y$`)` | † |
| $shift10_I(x, y)$ | `(shift10` $x$ $y$`)` | † |
| $divides_I(x, y)$ | `(dividesp` $x$ $y$`)` | † |
| $even_I(x)$ | `(evenp` $x$`)` | † |
| $odd_I(x)$ | `(oddp` $x$`)` | † |
| $divf_I(x, y)$ | `(div` $x$ $y$`)` | $\star$ |
| $moda_I(x, y)$ | `(mod` $x$ $y$`)` | $\star$ |
| $group_I(x, y)$ | `(group` $x$ $y$`)` | † |
| $pad_I(x, y)$ | `(pad` $x$ $y$`)` | † |
| $quot_I(x, y)$ | `(quot` $x$ $y$`)` | † |
| $remr_I(x, y)$ | `(remainder` $x$ $y$`)` | † |
| $gcd_I(x, y)$ | `(gcd` $x$ $y$`)` (deviation: `(gcd 0 0)` is 0) | $\star$ |
| $lcm_I(x, y)$ | `(lcm` $x$ $y$`)` | $\star$ |
| $gcd\_seq_I(xs)$ | `(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 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:

| | | |
|---|---|---|
| $min_F(x, y)$ | `(min` $x$ $y$`)` | $\star$ |
| $max_F(x, y)$ | `(max` $x$ $y$`)` | $\star$ |
| $mmin_F(x, y)$ | `(mmin` $x$ $y$`)` | † |
| $mmax_F(x, y)$ | `(mmax` $x$ $y$`)` | † |
| $min\_seq_F(xs)$ | `(min .` $xs$`)` or `(min` $x_1$ $x_2$ ... $x_n$`)` | $\star$ |
| $max\_seq_F(xs)$ | `(max .` $xs$`)` or `(max` $x_1$ $x_2$ ... $x_n$`)` | $\star$ |
| $mmin\_seq_F(xs)$ | `(mmin .` $xs$`)` or `(mmin` $x_1$ $x_2$ ... $x_n$`)` | † |
| $mmax\_seq_F(xs)$ | `(mmax .` $xs$`)` or `(mmax` $x_1$ $x_2$ ... $x_n$`)` | † |

| | | |
|---|---|---|
| $floor_F(x)$ | `(float (floor` $x$`))` | $\star$ |
| $rounding_F(x)$ | `(float (round` $x$`))` | $\star$ |
| $ceiling_F(x)$ | `(float (ceiling` $x$`))` | $\star$ |
| | | |
| $dim_F(x, y)$ | `(dim` $x$ $y$`)` | † |
| $dprod_{F \to F'}(x, y)$ | `(prod` $x$ $y$`)` | † |
| $remr_F(x, y)$ | `(remainder` $x$ $y$`)` | † |
| $sqrt_F(x)$ | `(sqrt` $x$`)` | $\star$ |
| $rsqrt_F(x)$ | `(rsqrt` $x$`)` | † |
| | | |
| $add\_lo_F(x, y)$ | `(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 $x$, $y$ and $z$ are data objects of the same floating point type, and where $xs$ is an data objects that are lists of data objects of the same floating point type.

The LIA-2 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

| | | |
|---|---|---|
| $max\_err\_hypot_F$ | `(err-hypotenuse` $x$`)` | † |
| | | |
| $max\_err\_exp_F$ | `(err-exp` $x$`)` | † |
| $max\_err\_power_F$ | `(err-power` $x$`)` | † |
| | | |
| $max\_err\_sinh_F$ | `(err-sinh` $x$`)` | † |
| $max\_err\_tanh_F$ | `(err-tanh` $x$`)` | † |
| | | |
| $big\_radian\_angle_F$ | `(big-radian-angle` $x$`)` | † |
| $max\_err\_sin_F$ | `(err-sin` $x$`)` | † |
| $max\_err\_tan_F$ | `(err-tan` $x$`)` | † |
| | | |
| $min\_angular\_unit_F$ | `(minimum-angular-unit` $x$`)` | † |
| $big\_angle_F$ | `(big-angle` $x$`)` | † |
| $max\_err\_sinu_F(u)$ | `(err-sin-cycle` $u$`)` | † |
| $max\_err\_tanu_F(u)$ | `(err-tan-cycle` $u$`)` | † |
| | | |
| $max\_err\_convert_F$ | `err-convert-to-string` | † |
| $max\_err\_convert_D$ | `err-convert-to-string` | † |

where $b$, $x$ 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` $x$ $y$`)` | † |
| | | |
| $power_{FI}(b, z)$ | `(expt` $b$ $z$`)` | $\star$ |

| | | |
|---|---|---|
| $exp_F(x)$ | `(exp` $x$`)` | $\star$ |
| $expm1_F(x)$ | `(expm1` $x$`)` | † |
| $exp2_F(x)$ | `(exp2` $x$`)` | † |
| $exp10_F(x)$ | `(exp10` $x$`)` | † |
| $power_F(b, y)$ | `(expt` $b$ $y$`)` | $\star$ |
| $power1pm1_F(b, y)$ | `(expm1` $b$ $y$`)` | † |
| | | |
| $ln_F(x)$ | `(log` $x$`)` | $\star$ |
| $ln1p_F(x)$ | `(log1p` $x$`)` | † |
| $log2_F(x)$ | `(log2` $x$`)` | † |
| $log10_F(x)$ | `(log10` $x$`)` | † |
| $logbase_F(b, x)$ | `(logbase` $b$ $x$`)` | † |
| $logbase1p1p_F(b, x)$ | `(logbase1p` $b$ $x$`)` | † |
| | | |
| $sinh_F(x)$ | `(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` $x$`)` | † |
| | | |
| $arcsinh_F(x)$ | `(asinh` $x$`)` | † |
| $arccosh_F(x)$ | `(acosh` $x$`)` | † |
| $arctanh_F(x)$ | `(atanh` $x$`)` | $\star$ |
| $arccoth_F(x)$ | `(acoth` $x$`)` | † |
| $arcsech_F(x)$ | `(asech` $x$`)` | † |
| $arccsch_F(x)$ | `(acsch` $x$`)` | † |
| | | |
| $axis\_rad_F(x)$ | `(axis_rad` $x$`)` | † |
| $rad_F(x)$ | `(radians` $x$`)` | † |
| | | |
| $sin_F(x)$ | `(sin` $x$`)` | $\star$ |
| $cos_F(x)$ | `(cos` $x$`)` | $\star$ |
| $tan_F(x)$ | `(tan` $x$`)` | $\star$ |
| $cot_F(x)$ | `(cot` $x$`)` | † |
| $sec_F(x)$ | `(sec` $x$`)` | † |
| $csc_F(x)$ | `(csc` $x$`)` | † |
| | | |
| $arcsin_F(x)$ | `(asin` $x$`)` | $\star$ |
| $arccos_F(x)$ | `(acos` $x$`)` | $\star$ |
| $arctan_F(x)$ | `(atan` $x$`)` | $\star$ |
| $arccot_F(x)$ | `(acot` $x$`)` | † |
| $arcctg_F(x)$ | `(actg` $x$`)` | † |
| $arcsec_F(x)$ | `(asec` $x$`)` | † |
| $arccsc_F(x)$ | `(acsc` $x$`)` | † |
| $arc_F(x, y)$ | `(atan2` $y$ $x$`)` | $\star$ |
| | | |
| $axis\_cycle_F(u, x)$ | `(axis_cycle` $u$ $x$`)` | † |
| $cycle_F(u, x)$ | `(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` $u$ $x$ | † |
| $arcctgu_F(u, x)$ | `(actgU` $u$ $x$`)` | † |
| $arcsecu_F(u, x)$ | `(asecU` $u$ $x$`)` | † |
| $arccscu_F(u, x)$ | `(acscU` $u$ $x$`)` | † |
| $arcu_F(u, x, y)$ | `(atan2U` $u$ $y$ $x$`)` | † |
| | | |
| $rad\_to\_cycle_F(x, u)$ | `(rad_to_cycle` $x$ $u$`)` | † |
| $cycle\_to\_rad_F(u, x)$ | `(cycle_to_rad` $u$ $x$`)` | † |
| $cycle\_to\_cycle_F(u, x, v)$ | `(cycle_to_cycle` $u$ $x$ $v$`)` | † |

where $b$, $x$, $y$, $u$, and $v$ are expressions of type *FLT*, and $z$ is an expressions of type *INT*.

Arithmetic value conversions in ISLisp are 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 $r$) |
| | | |
| $rounding_{F\to I}(y)$ | `(round` $y$`)` | ⋆ |
| $floor_{F\to I}(y)$ | `(floor` $y$`)` | ⋆ |
| $ceiling_{F\to I}(y)$ | `(ceiling` $y$`)` | ⋆ |
| | | |
| $convert_{I\to F}(x)$ | `(float` $x$ *kind*`)` | ⋆ |
| | | |
| $convert_{F\to F'}(y)$ | `(float` $y$ *kind*`)` | ⋆ |
| $convert_{F\to F''}(y)$ | `(format` $g$ `"~G"` $y$`)` | ⋆ |
| $convert_{F\to F''}(y)$ | `(format-float` $g$ $y$`)` | ⋆ |

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. 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 types in base is 10.

ISLisp does not specify numerals for infinities and NaNs. Suggestion:

| | | |
|---|---|---|
| **+∞** | `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. ISLisp ignores **underflow** notifications. On **underflow** the continuation value (specified in LIA-2) is used directly without recording the **underflow** itself. 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.9 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* [21]. An implementation should follow all the requirements of LIA-3 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-3 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**.

The additional integer operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(I)}(x, y)$ | `add(`$x$`, `$y$`)` | † |

where $x$ and $y$ are expressions of type *INT* and where $xs$ is an expression of type `array [] of` *INT*.

The additional non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(F)}(x, y)$ | `add(`$x$`, `$y$`)` | † |
| $sub_{c(F)}(x, y)$ | `sub(`$x$`, `$y$`)` | † |

where $x$, $y$ and $z$ are expressions of type *FLT*, and where $xs$ is an expression of type `array [] of` *FLT*.

The LIA-3 parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

| | | |
|---|---|---|
| $max\_err\_mul_{c(F)}$ | `err_mul(`$x$`)` | † |
| $max\_err\_div_{c(F)}$ | `err_div(`$x$`)` | † |
| $max\_err\_exp_{c(F)}$ | `err_exp(`$x$`)` | † |
| $max\_err\_power_{c(F)}$ | `err_power(`$x$`)` | † |
| $max\_err\_sin_{c(F)}$ | `err_sin(`$x$`)` | † |
| $max\_err\_tan_{c(F)}$ | `err_tan(`$x$`)` | † |

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:

| | | |
|---|---|---|
| $mul_{c(F)}(x, y)$ | `mul(`$x$`, `$y$`)` | † |
| $div_{c(F)}(x, y)$ | `div(`$x$`, `$y$`)` | † |
| $sqrt_{c(F)}(x)$ | `sqrt(`$x$`)` | ⋆ |
| $exp_{c(F)}(x)$ | `exp(`$x$`)` | ⋆ |
| $power_{c(F)}(b, y)$ | `power(`$b$`, `$y$`)` | ⋆ |
| $ln_{c(F)}(x)$ | `ln(`$x$`)` | ⋆ |
| $logbase_{c(F)}(b, x)$ | `log(`$x$`, `$b$`)` | † |
| $radh_{c(F)}(x)$ | `hypradian(`$x$`)` | † |
| $sinh_{c(F)}(x)$ | `sinh(`$x$`)` | † |
| $cosh_{c(F)}(x)$ | `cosh(`$x$`)` | † |
| $tanh_{c(F)}(x)$ | `tanh(`$x$`)` | † |
| $coth_{c(F)}(x)$ | `coth(`$x$`)` | † |
| $sech_{c(F)}(x)$ | `sech(`$x$`)` | † |
| $csch_{c(F)}(x)$ | `csch(`$x$`)` | † |
| $arcsinh_{c(F)}(x)$ | `arcsinh(`$x$`)` | † |
| $arccosh_{c(F)}(x)$ | `arccosh(`$x$`)` | † |
| $arctanh_{c(F)}(x)$ | `arctanh(`$x$`)` | † |
| $arccoth_{c(F)}(x)$ | `arccoth(`$x$`)` | † |
| $arcsech_{c(F)}(x)$ | `arcsech(`$x$`)` | † |
| $arccsch_{c(F)}(x)$ | `arccsch(`$x$`)` | † |
| $rad_{c(F)}(x)$ | `radian(`$x$`)` | † |
| $sin_{c(F)}(x)$ | `sin(`$x$`)` | ⋆ |
| $cos_{c(F)}(x)$ | `cos(`$x$`)` | ⋆ |
| $tan_{c(F)}(x)$ | `tan(`$x$`)` | ⋆ |
| $cot_{c(F)}(x)$ | `cot(`$x$`)` | † |
| $sec_{c(F)}(x)$ | `sec(`$x$`)` | † |
| $csc_{c(F)}(x)$ | `csc(`$x$`)` | † |
| $arcsin_{c(F)}(x)$ | `arcsin(`$x$`)` | ⋆ |
| $arccos_{c(F)}(x)$ | `arccos(`$x$`)` | ⋆ |
| $arctan_{c(F)}(x)$ | `arctan(`$x$`)` | ⋆ |
| $arccot_{c(F)}(x)$ | `arccot(`$x$`)` | † |
| $arcctg_{c(F)}(x)$ | `arcctg(`$x$`)` | † |
| $arcsec_{c(F)}(x)$ | `arcsec(`$x$`)` | † |
| $arccsc_{c(F)}(x)$ | `arccsc(`$x$`)` | † |

where $b$, $x$, $y$, $u$, and $v$ are expressions of type *CFLT*.

Arithmetic value conversions in Modula-2 are 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.

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 $I'$. A ? above indicates that the parameter is optional. $e$ is greater than 0.

## C.10 PL/I

The programming language PL/I is defined by ANSI X3.53-l976 (R1998), *Programming languages – PL/I* [39], and endorsed by ISO 6160:1979, *Programming languages – PL/I* [25]. The programming language General Purpose PL/I is defined by ISO/IEC 6522:1992, *Information technology – Programming languages – PL/I general-purpose subset* [26], also: ANSI X3.74-1987 (R1998).

An implementation should follow all the requirements of LIA-3 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-3 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 data type, and at least one floating point data type. 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 LIA-3 integer operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(I)}(x, y)$ | $x$ `+` $y$ | † |
| $sub_{c(I)}(x, y)$ | $x$ `-` $y$ | † |
| $mul_{c(I)}(x, y)$ | $x$ `*` $y$ | † |

where $x$ and $y$ are expressions of type *CINT*.

The LIA-3 non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(F)}(x, y)$ | $x$ `+` $y$ | ⋆ |
| $sub_{c(F)}(x, y)$ | $x$ `-` $y$ | ⋆ |

where $x$, $y$ and $z$ 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:

| | | |
|---|---|---|
| $max\_err\_mul_{c(F)}$ | `err_mul(`$x$`)` | † |
| $max\_err\_div_{c(F)}$ | `err_div(`$x$`)` | † |
| $max\_err\_exp_{c(F)}$ | `err_exp(`$x$`)` | † |
| $max\_err\_power_{c(F)}$ | `err_power(`$x$`)` | † |

| | | |
|---|---|---|
| $max\_err\_sin_{c(F)}$ | `err_sin`$(x)$ | † |
| $max\_err\_tan_{c(F)}$ | `err_tan`$(x)$ | † |

where $x$ and $u$ are expressions of type *CFLT*. 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-3 elementary floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $mul_{c(F)}(x, y)$ | $x$ `*` $y$ | ⋆ |
| $div_{c(F)}(x, y)$ | $x$ `/` $y$ | ⋆ |
| $sqrt_{c(F)}(x)$ | `sqrt`$(x)$ | ⋆ |
| $exp_{c(F)}(x)$ | `exp`$(x)$ | ⋆ |
| $power_{c(F)}(b, y)$ | `power`$(b,\ y)$ | † |
| | | |
| $ln_{c(F)}(x)$ | `log`$(x)$ | ⋆ |
| $logbase_{c(F)}(b, x)$ | `log`$(b,\ x)$ | † |
| | | |
| $radh_{c(F)}(x)$ | `radh`$(x)$ | † |
| | | |
| $sinh_{c(F)}(x)$ | `sinh`$(x)$ | ⋆ |
| $cosh_{c(F)}(x)$ | `cosh`$(x)$ | ⋆ |
| $tanh_{c(F)}(x)$ | `tanh`$(x)$ | ⋆ |
| $coth_{c(F)}(x)$ | `coth`$(x)$ | † |
| $sech_{c(F)}(x)$ | `sech`$(x)$ | † |
| $csch_{c(F)}(x)$ | `csch`$(x)$ | † |
| | | |
| $arcsinh_{c(F)}(x)$ | `arcsinh`$(x)$ | ⋆ |
| $arccosh_{c(F)}(x)$ | `arccosh`$(x)$ | ⋆ |
| $arctanh_{c(F)}(x)$ | `arctanh`$(x)$ | ⋆ |
| $arccoth_{c(F)}(x)$ | `arccoth`$(x)$ | † |
| $arcsech_{c(F)}(x)$ | `arcsech`$(x)$ | † |
| $arccsch_{c(F)}(x)$ | `arccsch`$(x)$ | † |
| | | |
| $rad_{c(F)}(x)$ | `rad`$(x)$ | † |
| | | |
| $sin_{c(F)}(x)$ | `sin`$(x)$ | ⋆ |
| $cos_{c(F)}(x)$ | `cos`$(x)$ | ⋆ |
| $tan_{c(F)}(x)$ | `tan`$(x)$ | ⋆ |
| $cot_{c(F)}(x)$ | `cot`$(x)$ | ⋆ |
| $sec_{c(F)}(x)$ | `sec`$(x)$ | † |
| $csc_{c(F)}(x)$ | `csc`$(x)$ | † |
| | | |
| $arcsin_{c(F)}(x)$ | `arcsin`$(x)$ | ⋆ |
| $arccos_{c(F)}(x)$ | `arccos`$(x)$ | ⋆ |
| $arctan_{c(F)}(x)$ | `arctan`$(x)$ | ⋆ |
| $arccot_{c(F)}(x)$ | `arccot`$(x)$ | † |
| $arcctg_{c(F)}(x)$ | `arcctg`$(x)$ | † |
| $arcsec_{c(F)}(x)$ | `arcsec`$(x)$ | † |
| $arccsc_{c(F)}(x)$ | `arccsc`$(x)$ | † |

where $b$, $x$, $y$, $u$, and $v$ are expressions of type *CFLT*.

Arithmetic value conversions in PL/I are 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.

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 $I'$. A ? above indicates that the parameter is optional. $a$ is greater than 0.

## C.11 SML

The programming language SML is defined by *The Definition of Standard ML (Revised)* [58].

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-3 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:

| | | |
|---|---|---|
| $add_{c(I)}(x, y)$ | $x$ `+` $y$ or `op +` $(x,y)$ | $\star$ |
| $sub_{c(I)}(x, y)$ | $x$ `-` $y$ or `op -` $(x,y)$ | $\star$ |
| $mul_{c(I)}(x, y)$ | $x$ `*` $y$ or `op *` $(x,y)$ | $\star$ |

where $x$ and $y$ are expressions of type *CINT*.

The additional non-transcendental floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $add_{c(F)}(x, y)$ | $x$ `+` $y$ or `op +` $(x,y)$ | $\star$ |
| $sub_{c(F)}(x, y)$ | $x$ `-` $y$ or `op -` $(x,y)$ | $\star$ |

where $x$, $y$ and $z$ are expressions of type *CFLT*.

The binding for the floor, round, and ceiling operations here take advantage of the unlimited `Integer` type in SML, and that `Integer` is the default integer type.

The parameters for operations approximating real valued transcendental functions can be accessed by the following syntax:

| | | |
|---|---|---|
| $max\_err\_mul_F$ | `err_mul` $x$ | † |
| $max\_err\_div_F$ | `err_div` $x$ | † |
| $max\_err\_exp_F$ | `err_exp` $x$ | † |
| $max\_err\_power_F$ | `err_power` $x$ | † |
| $max\_err\_sin_F$ | `err_sin` $x$ | † |
| $max\_err\_tan_F$ | `err_tan` $x$ | † |

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-3 elementary floating point operations are listed below, along with the syntax used to invoke them:

| | | |
|---|---|---|
| $mul_{c(F)}(x, y)$ | $x$ `*` $y$ or `op *` $(x,y)$ | ⋆ |
| $div_{c(F)}(x, y)$ | $x$ `/` $y$ or `op /` $(x,y)$ | ⋆ |
| $sqrt_{c(F)}(x)$ | `sqrt` $x$ | ⋆ |
| | | |
| $exp_{c(F)}(x)$ | `exp` $x$ | ⋆ |
| $power_{c(F)}(b, y)$ | $b$ `**` $y$ | † |
| $pow_{c(F)}(b, y)$ | $b$ `pow` $y$ or `op pow` $(x,y)$ | ⋆ Not LIA-2! (See C.) |
| | | |
| $ln_{c(F)}(x)$ | `ln` $x$ | ⋆ |
| $logbase_{c(F)}(b, x)$ | `log_base` $(b,x)$ | † |
| | | |
| $radh_{c(F)}(x)$ | `hypradians` $x$ | † |
| | | |
| $sinh_{c(F)}(x)$ | `sinh` $x$ | ⋆ |
| $cosh_{c(F)}(x)$ | `cosh` $x$ | ⋆ |
| $tanh_{c(F)}(x)$ | `tanh` $x$ | ⋆ |
| $coth_{c(F)}(x)$ | `coth` $x$ | † |
| $sech_{c(F)}(x)$ | `sech` $x$ | † |
| $csch_{c(F)}(x)$ | `csch` $x$ | † |
| | | |
| $arcsinh_{c(F)}(x)$ | `arcsinh` $x$ | † |
| $arccosh_{c(F)}(x)$ | `arccosh` $x$ | † |
| $arctanh_{c(F)}(x)$ | `arctanh` $x$ | † |
| $arccoth_{c(F)}(x)$ | `arccoth` $x$ | † |
| $arcsech_{c(F)}(x)$ | `arcsech` $x$ | † |
| $arccsch_{c(F)}(x)$ | `arccsch` $x$ | † |
| | | |
| $rad_{c(F)}(x)$ | `radians` $x$ | † |
| | | |
| $sin_{c(F)}(x)$ | `sin` $x$ | ⋆ |
| $cos_{c(F)}(x)$ | `cos` $x$ | ⋆ |
| $tan_{c(F)}(x)$ | `tan` $x$ | ⋆ |
| $cot_{c(F)}(x)$ | `cot` $x$ | † |
| $sec_{c(F)}(x)$ | `sec` $x$ | † |
| $csc_{c(F)}(x)$ | `csc` $x$ | † |
| | | |
| $arcsin_{c(F)}(x)$ | `arcsin` $x$ | ⋆ |
| $arccos_{c(F)}(x)$ | `arccos` $x$ | ⋆ |
| $arctan_{c(F)}(x)$ | `arctan` $x$ | ⋆ |
| $arccot_{c(F)}(x)$ | `arccot` $x$ | † |
| $arcctg_{c(F)}(x)$ | `arcctg` $x$ | † |
| $arcsec_{c(F)}(x)$ | `arcsec` $x$ | † |
| $arccsc_{c(F)}(x)$ | `arccsc` $x$ | † |

where $b$, $x$, $y$, $u$, and $v$ are expressions of type *FLT*, and $z$ is an expressions of type *INT*

Type conversions in SML are always explicit.

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 $I'$.

# Annex D
(informative)

# Bibliography

This annex gives references to publications relevant to LIA-3.

### International standards documents

[1] ISO/IEC JTC1 Directives, Part 3: *Drafting and presentation of International Standards*, 1989.

[2] ISO 6093:1985, *Information processing – Representation of numerical values in character strings for information interchange.*

[3] ISO/IEC TR 10176:1998, *Information technology – Guidelines for the preparation of programming language standards.*

[4] ISO/IEC TR 10182:1993, *Information technology – Programming languages, their environments and system software interfaces – Guidelines for language bindings.*

[5] ISO/IEC 13886:1996, *Information technology – Language-Independent Procedure Calling*, (LIPC).

[6] ISO/IEC 11404:1996, *Information technology – Programming languages, their environments and system software interfaces – Language-independent datatypes*, (LID).

[7] ISO/IEC 8652:1995, *Information technology – Programming languages – Ada.*

[8] 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).*

[9] ISO/IEC 13814:1998, *Information technology – Programming languages – Generic package of complex elementary functions for Ada.*

[10] ISO 8485:1989, *Programming languages – APL.*

[11] ISO/IEC DIS 13751, *Information technology – Programming languages, their environments and system software interfaces – Programming language APL, extended*, 1999.

[12] ISO/IEC 10279:1991, *Information technology – Programming languages – Full BASIC.* (Essentially an endorsement of ANSI X3.113-1987 (R1998) [36].)

[13] ISO/IEC 9899:1999, *Programming languages – C.*

[14] ISO/IEC 14882:1998, *Programming languages – C++.*

[15] ISO 1989:1985, *Programming languages – COBOL.* (Endorsement of ANSI X3.23-l985 (R1991) [37].) Currently under revision (1998).

[16] ISO/IEC 16262:1998, *Information technology - ECMAScript language specification.*

[17] ISO/IEC 15145:1997, *Information technology – Programming languages – FORTH.* (Also: ANSI X3.215-1994.)

[18] ISO/IEC 1539-1:1997, *Information technology – Programming languages – Fortran - Part 1: Base language.*

[19] ISO/IEC TR 15580:1998, *Information technology – Programming languages – Fortran – Floating-point exception handling.*

[20] ISO/IEC 13816:1997, *Information technology – Programming languages, their environments and system software interfaces – Programming language ISLISP.*

[21] ISO/IEC 10514-1:1996, *Information technology – Programming languages – Part 1: Modula-2, Base Language.*

[22] ISO/IEC 10514-2:1998, *Information technology – Programming languages – Part 2: Generics Modula-2.*

[23] ISO 7185:1990, *Information technology – Programming languages – Pascal.*

[24] ISO/IEC 10206:1991, *Information technology – Programming languages – Extended Pascal.*

[25] ISO 6160:1979, *Programming languages – PL/I.* (Endorsement of ANSI X3.53-l976 (R1998) [39].)

[26] ISO/IEC 6522:1992, *Information technology – Programming languages – PL/I general-purpose subset.* (Also: ANSI X3.74-1987 (R1998).)

[27] ISO/IEC 13211-1:1995, *Information technology – Programming languages – Prolog – Part 1: General core.*

[28] ISO/IEC 9075:1992, *Information technology – Database languages – SQL.*

[29] ISO/IEC 8824-1:1995, *Information technology – Abstract Syntax Notation One (ASN.1) – Part 1: Specification of basic notation.*

[30] ISO/IEC 9001:1994, *Quality systems – Model for quality assurance in design, development, production, installation and servicing.*

[31] ISO/IEC 9126:1991, *Information technology – Software product evaluation – Quality characteristics and guidelines for their use.*

[32] ISO/IEC 12119:1994, *Information technology – Software packages – Quality requirements and testing.*

[33] ISO/IEC 14598-1:1999, *Information technology – Software product evaluation – Part 1: General overview.*

### National standards documents

[34] ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic.*

[35] ANSI/IEEE Standard 854-1987, *IEEE Standard for Radix-Independent Floating-Point Arithmetic.*

[36] ANSI X3.113-1987 (R1998), *Information technology – Programming languages – Full BASIC.*

[37] ANSI X3.23-l985 (R1991), *Programming languages – COBOL.*

[38] ANSI X3.226-1994, *Information Technology – Programming Language – Common Lisp.*

[39] ANSI X3.53-l976 (R1998), *Programming languages – PL/I.*

[40] ANSI/IEEE 1178-1990, *IEEE Standard for the Scheme Programming Language.*

[41] ANSI/NCITS 319-1998, *Information Technology – Programming Languages – Smalltalk.*

### Books, articles, and other documents

[42] J. S. Squire (ed), *Ada Letters*, vol. XI, No. 7, ACM Press (1991).

[43] 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.

[44] J. Du Croz and M. Pont, *The Development of a Floating-Point Validation Package*, *NAG Newsletter*, No. 3, 1984.

[45] 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.

[46] D. Goldberg, *What Every Computer Scientist Should Know about Floating-Point Arithmetic.* ACM Computing Surveys, Vol. 23, No. 1, March 1991.

[47] 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.

[48] 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.

[49] 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.

[50] U. Kulisch and W. L. Miranker, *Computer Arithmetic in Theory and Practice*, Academic Press, 1981.

[51] U. Kulisch and W. L. Miranker (eds), *A New Approach to Scientific Computation*, Academic Press, 1983.

[52] 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.

[53] *Floating-Point C Extensions* in Technical Report Numerical C Extensions Committee X3J11, April 1995, SC22/WG14 N403, X3J11/95-004.

[54] David M. Gay, *Correctly Rounded Binary-Decimal and Decimal-Binary Conversions*, AT&T Bell Laboratories, Numerical Analysis Manuscript 90-10, November 1990.

[55] James Gosling, Bill Joy, Guy Steele, *The Java Language Specification.*

[56] Simon Peyton Jones et al., *Report on the programming language Haskell 98*, February 1999.

[57] Simon Peyton Jones et al., *Standard libraries for the Haskell 98 programming language*, February 1999.

[58] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen, *The Definition of Standard ML (Revised)*, The MIT Press, 1997, ISBN: 0-262-63181-4.