SC22/WG11 N519

DRAFT INTERNATIONAL STANDARD **ISO/IEC FDIS 10967-1**

Final draft (FDIS) for the Second edition
2011-09-14

Information technology — Language independent arithmetic —

Part 1: Integer and floating point arithmetic

Technologies de l'information —
Arithmétique indépendante des languages —

Partie 1: Arithmétique des nombres entiers et en virgule flottante

FINAL DRAFT INTERNATIONAL STANDARD
September 14, 2011 15:48

Editor:
Kent Karlsson
E-mail: kent.karlsson14@telia.com

Reference number
ISO/IEC FDIS 10967-1:2011(E)

Contents

Foreword

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

International Standards are drafted in accordance with the rules in the ISO/IEC Directives, Part 2 [1].

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

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

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

This second edition cancels and replaces the first edition (ISO/IEC 10967-1:1994), which has been technically revised.

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

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

Introduction

The aims

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

It is the intent of this part of ISO/IEC 10967 to help to redress these shortcomings, by setting out precise definitions of integer and floating point datatypes, and requirements for documentation.

It is not claimed that this part of ISO/IEC 10967 will ensure complete certainty of arithmetic behaviour in all circumstances; the complexity of numeric software and the difficulties of analysing and proving algorithms are too great for that to be attempted.

The first aim of this part of ISO/IEC 10967 is to enhance the predictability and reliability of the behaviour of programs performing numeric processing.

The second aim, which helps to support the first, is to help programming language standards to express the semantics of arithmetic datatypes.

The third aim is to help enhance the portability of programs that perform numeric processing across a range of different platforms. Improved predictability of behaviour will aid programmers designing code intended to run on multiple platforms, and will help in predicting what will happen when such a program is moved from one conforming language processor to another.

Note that this part of ISO/IEC 10967 does not attempt to ensure bit-for-bit identical results when programs are transferred between language processors, or translated from one language into another. However, experience shows that diverse numeric environments can yield comparable results under most circumstances, and that with careful program design significant portability is actually achievable. In addition, the IEC 60559 (IEEE 754) standard goes a long way to ensure bit-for-bit identical results, and in this second edition of this part of ISO/IEC 10967 the requirements are tightened (compared to the first edition) to approach those of IEEE 754.

The content

This part of ISO/IEC 10967 defines the fundamental properties of integer and floating point datatypes. These properties are presented in terms of a parameterised model. The parameters allow enough variation in the model so that several integer and floating point datatypes are covered. In particular, the IEC 60559 (IEEE 754) floating point datatypes, both those of radix 2 and those of radix 10, are covered, as well as integer datatypes, both unlimited and limited, for the latter both signed or unsigned, are covered. But when a particular set of parameter values is selected, and all required documentation is supplied, the resulting information should be precise enough to permit careful numerical analysis.

The requirements of this part of ISO/IEC 10967 cover four areas. First, the programmer must be given runtime access to the specified operations on values of integer or floating point datatype. Second, the programmer must be given runtime access to the parameters (and parameter functions) that describe the arithmetic properties of an integer or floating point datatype. Third, the executing program must be notified when proper results cannot be returned (e.g., when a computed result may be out of range or undefined). Fourth, the numeric properties of conforming platforms must be publicly documented.

This part of ISO/IEC 10967 focuses on the classical integer and floating point datatypes. Subsequent parts considers common elementary numerical functions (Part 2), complex numerical numbers and complex elementary numerical functions (Part 3).

The benefits

Adoption and proper use of this part of ISO/IEC 10967 can lead to the following benefits.

For programming language standards it will be possible to define their arithmetic semantics more precisely without preventing the efficient implementation of the 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.

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

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

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

Information technology — Language independent arithmetic —

Part 1: Integer and floating point arithmetic

1 Scope

This part of ISO/IEC 10967 specifies properties of many of the integer and floating point datatypes available in a variety of programming languages in common use for mathematical and numerical applications.

It is not the purpose of this part of ISO/IEC 10967 to ensure that an arbitrary numerical function can be so encoded as to produce acceptable results on all conforming datatypes. Rather, the goal is to ensure that the properties of the arithmetic on a conforming datatype are made available to the programmer. Therefore, it is not reasonable to demand that a substantive piece of software run on every implementation that can claim conformity to this part of ISO/IEC 10967.

An implementor may choose any combination of hardware and software support to meet the specifications of this part of ISO/IEC 10967. It is the datatypes and operations on values of those datatypes, of the computing environment as seen by the programmer/user, that does or does not conform to the specifications.

The term *implementation* (of this part of ISO/IEC 10967) denotes the total computing environment pertinent to this part of ISO/IEC 10967, including hardware, language processors, subroutine libraries, exception handling facilities, other software, and documentation.

1.1 Inclusions

This part of ISO/IEC 10967 provides specifications for properties of integer and floating point datatypes as well as basic operations on values of these datatypes. Specifications are included for bounded and unbounded integer datatypes, as well as floating point datatypes. 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 (not-a-number), is prescribed for each specified floating point operation.

This part of ISO/IEC 10967 provides specifications for:

a) The set of required values of the arithmetic datatype.

b) A number of arithmetic operations, including:

    1) comparison operations on two operands of the same type,

    2) primitive operations (addition, subtraction, etc.) with operands of the same type,

    3) operations that access properties of individual values,

4) conversion operations of a value from one arithmetic datatype to another arithmetic datatype, where at least one of the datatypes is conforming to this part of ISO/IEC 10967, and
5) numerals for all values specified by this part of ISO/IEC 10967 for a conforming datatype.

This part of ISO/IEC 10967 also provides specifications for:

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

d) Program-visible parameters that characterise the values and certain aspects of the operations of an arithmetic datatype.

e) Methods for reporting arithmetic exceptions.

1.2 Exclusions

This part of ISO/IEC 10967 provides no specifications for:

a) Arithmetic and comparison operations whose operands are of more than one datatype. This part of ISO/IEC 10967 neither requires nor excludes the presence of such "mixed operand" operations.

b) An interval datatype, or the operations on such data. This part of ISO/IEC 10967 neither requires nor excludes such data or operations.

c) A fixed point datatype, or the operations on such data. This part of ISO/IEC 10967 neither requires nor excludes such data or operations.

d) A rational datatype, or the operations on such data. This part of ISO/IEC 10967 neither requires nor excludes such data or operations.

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

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

   1) the syntax of numerals and expressions in the programming language, including the precedence of operators 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 presence or absence of automatic datatype coercions, and the consequences of applying an operation to values of improper type, or to uninitialised data,
   4) the rules for assignment, parameter passing, and returning value.

   NOTE – See Clause 7 and Annex D for a discussion of language standards and language bindings.

The internal representation of values is beyond the scope of this standard. E.g., the value of the exponent bias, if any, is not specified, nor available as a parameter specified by this part of ISO/IEC 10967. Internal representations need not be unique, nor is there a requirement for identifiable fields (for sign, exponent, and so on).

Furthermore, this part of ISO/IEC 10967 does not provide specifications for how the operations should be implemented or which algorithms are to be used for the various operations.

2 Conformity

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

A binding standard specifies the correspondence between one or more of the arithmetic datatypes, parameters, and operations specified in this part of ISO/IEC 10967 and the concrete language syntax of some programming language. More generally, a binding standard specifies the correspondence between certain datatypes, parameters, and operations and the elements of some arbitrary computing entity. A language standard that explicitly provides such binding information can serve as a binding standard.

When a binding standard for a language exists, an implementation shall be said to conform to this part of ISO/IEC 10967 if and only if it conforms to the binding standard. In the case of conflict between a binding standard and this part of ISO/IEC 10967, the specifications of the binding standard takes precedence.

When a binding standard requires only a subset of the integer or floating point datatypes provided, an implementation remains free to conform to this part of ISO/IEC 10967 with respect to other datatypes independently of that binding standard.

When a binding standard requires only a subset of the operations specified in this part of ISO/IEC 10967, an implementation remains free to conform to this part of ISO/IEC 10967 with respect to other datatypes and operations, independently of that binding standard.

When no binding standard exists, an implementation conforms to this part of ISO/IEC 10967 if and only if it provides one or more datatypes and 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.

Conformity to this part of ISO/IEC 10967 is always with respect to a specified set of datatypes and set of operations. Under certain circumstances, conformity to IEC 60559 is implied by conformity to this part of ISO/IEC 10967.

An implementation is free to provide arithmetic datatypes and arithmetic operations that do not conform to this part of ISO/IEC 10967 or that are beyond the scope of this part of ISO/IEC 10967. The implementation shall not claim conformity to this part of ISO/IEC 10967 for such datatypes or operations.

An implementation is permitted to have modes of operation that do not conform to this part of ISO/IEC 10967. A conforming implementation shall specify how to select the modes of operation that ensure conformity. However, a mode of operation that conforms to this part of ISO/IEC 10967 should be the default mode of operation.

NOTES

1 Language bindings are essential. Clause 8 requires an implementation to supply a binding if no binding standard exists. See Annex C.7 for recommendations on the proper content of a binding standard, Annex E for an example of a conformity statement, and Annex D for suggested language bindings.

2 A complete binding for this part of ISO/IEC 10967 may include (explicitly or by reference) a binding for IEC 60559 as well. See 5.2.1 and Annex B.

3 It is not possible to conform to this part of ISO/IEC 10967 without specifying to which datatypes and set of operations, and modes of operation, conformity is claimed.

4 This part of ISO/IEC 10967 requires that certain integer operations are made available for a conforming integer datatype, and that certain floating point operations are made available for a conforming floating point datatype.

5 All the operations specified in this part of ISO/IEC 10967 for a datatype must be provided for a conforming datatype, in a conforming mode of operation for that datatype.

3 Normative references

The following referenced documents are indispensable for the application of this part of ISO/IEC 10967. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 60559, *Standard for floating-point arithmetic.*

4 Symbols and definitions

4.1 Symbols

For the purposes of this document, the following symbols are used.

4.1.1 Operators and relations

All prefix and infix operators have their conventional exact mathematical meaning. In particular, this document uses:

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

NOTE 1 – $\approx$ is used informally, in notes and the rationale.

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

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

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

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

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

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

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

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

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

NOTE 2 – $i|j$ is true exactly when j/i is defined and $j/i \in \mathcal{Z}$.

4.1.2 Sets and intervals

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

The conventional notation for set definition and for set operations are used.

The following notation for intervals is used in this document:

$[x, z]$ 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.

4.1.3 Exceptional values

The parts of ISO/IEC 10967 use the following six exceptional values:

a) **inexact**: the result is rounded and different from the exact result.

b) **underflow**: the absolute value of the unrounded result is less than the smallest normal value, and the rounded result may have lost accuracy due to the denormalisation (more than lost by ordinary rounding if the exponent range was unbounded).

c) **overflow**: the rounded result (when rounding as if the exponent range was unbounded) is larger than what can be represented in the result datatype.

d) **infinitary**: the corresponding mathematical function has a pole at the finite argument point, or the result is otherwise infinite from finite arguments.

NOTE – **infinitary** is a generalisation of **divide_by_zero**.

e) **invalid**: the operation is undefined but not infinitary, or the result is in $\mathcal{C}$ but not in $\mathcal{R}$, for the given arguments.

f) **absolute_precision_underflow**: indicates that at least one argument is such that the density of representable values is too low in the neighbourhood of the given argument value for a numeric result to be considered appropriate to return. This exceptional value is used for operations that approximate trigonometric functions (Part 2 and Part 3) and for operations that that approximate complex hyperbolic and exponentiation functions (Part 3).

For the exceptional values, a continuation value may be given in ISO/IEC 10967 in parenthesis after the exceptional value.

4.1.4 Special values

The following symbols represent special values defined in IEC 60559 and are used in ISO/IEC 10967:

$\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, **qNaN**, and **sNaN**.

These values are not part of I or F (see Clauses 5.1 and 5.2 for a definition of these datatypes), but if

$$hasinf_I$$

(see Clause 5.1) has the value **true**, the $\mathbf{+\infty}$, $\mathbf{-\infty}$ values are included in the integer datatype in the implementation that corresponds to I, and if iec_60559_F (see Clause 5.2.1) has the value **true**, all these special values are included in the floating point datatype in the implementation that corresponds to F.

NOTE – This document uses the above five special values for compatibility with IEC 60559. In particular, the symbol $\mathbf{-0}$ (in bold) is not the application of (mathematical) unary $-$ to the value 0, and is a value logically distinct from 0.

The specifications for floating point operations cover the results to be returned by an operation if given one or more of the IEC 60559 special values $\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, or NaNs as input values. These specifications apply only to systems which provide and support these special values.

If an implementation is not capable of representing a $\mathbf{-0}$ result or continuation value, 0 shall be used as the actual result or continuation value. If an implementation is not capable of representing a prescribed result or continuation value of the IEC 60559 special values $\mathbf{+\infty}$, $\mathbf{-\infty}$, or **qNaN**, the actual result or continuation value is binding or implementation defined.

4.1.5 The Boolean datatype

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

NOTE – Mathematical relations *are* true or false (or undefined, if an operand is undefined), which are abstract conditions, not values in a datatype. In contrast, **true** and **false** are values in **Boolean**.

4.1.6 Operation specification framework

Each of the operations are specified using a mathematical notation with cases. Each case condition is intended to be disjoint with the other cases, and encompass all non-special values as well as some of the special values.

Mathematically, each argument to an operation is a pair of a value and a set of exceptional values and likewise for the return value. However, in most cases only the first part of this pair is written out in the specifications. The set of exceptional values returned from an operation is at least the union of the set of exceptional values from the arguments. Any new exceptional value that the operation itself gives rise to is given in the form **exceptional_value**(*continuation_value*) indicating that the second (implicit) part of the mathematical return value not only is the union of the second (implicit) parts of the arguments, but in addition is unioned with the singleton set of the given exceptional value, or, in the case of **underflow** or **overflow**, the set of the given exceptional value and **inexact**.

In an implementation, the exceptional values usually do not accompany each argument and return value, but are instead handled as notifications. See Clause 6.

When not communicating values, notifications shall be internal to each computational thread, whether threads are explicit or implicit in the program as seen by the programmer.

When communicating values, if the value sending thread has notifications that may be relevant for a communicated values these notifications should be communicated to a receiving thread along with values (of any datatype, not just numeric ones). In such instances, the exceptional values are associated with the value, even though it may pick up notifications in the thread that arose for a different computation in that thread and were not cleared.

NOTES

1 If notifications were arbitrarily seen in other threads, it would be very difficult to know which computation (thread) it is that might have caused the notification, and thus may trigger notification handling when not appropriate in an unrelated thread. Therefore it is essential that notifications are internal to each computational thread, when not communicating a value.

2 If notifications (normally recorded in indicators) are trimmed away when communicating a value (of whatever type) to another thread, that can result in the failure to cause notification handling when that would have been appropriate. Not communicating notifications between communicating threads thus goes against a goal set out in the introduction, namely "the executing program must be notified when proper results cannot be returned (e.g., when a computed result may be out of range or undefined)".

However, many existing methods for remote procedure calling, or thread communication, do not communicate notifications (even when they are recorded in indicators).

4.2 Definitions of terms

For the purposes of this document, the following terms and definitions apply.

4.2.1
accuracy
closeness between the true mathematical result and a computed result

4.2.2
arithmetic datatype
datatype whose non-special values are members of $\mathcal{Z}$, $\mathcal{R}$, or $\mathcal{C}$

4.2.3
continuation value
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 be an IEC 60559 special value. (Contrast with *exceptional value.* See Clause 6.2.1.)

4.2.4
denormalisation
inclusion of lead zero digits, with corresponding adjustment of the exponent

Denormalisation is logically done before rounding (otherwise there may be double rounding, that is rounding done twice with slightly different rounding functions, and that would be nonconforming). It may be done in order to get the exponent (just) within representable range.

4.2.5
denormalisation loss
larger than normal rounding error caused by the fact that denormalisation plus rounding may lose precision more than only rounding would do if the target exponent range was unbounded

See Clause 5.2.4 for a full definition.

4.2.6
error
⟨in computed value⟩ difference between a computed value and the mathematically correct value

Used in phrases like "rounding error" or "error bound".

4.2.7
error
⟨computation gone awry⟩ exception

Used in phrases like "error message" or "error output". Error and exception are not synonyms in any other contexts.

4.2.8
exception
inability of an operation to return a suitable finite numeric result from finite arguments

This might arise because no such finite result exists mathematically (**infinitary** (e.g., at a pole), **invalid** (e.g., when the true result is in $\mathcal{C}$ but not in $\mathcal{R}$)), or because the mathematical result cannot, or might not, be representable with sufficient accuracy (**underflow**, **overflow**) or viability (**absolute_precision_underflow**).

NOTES

1 **absolute_precision_underflow** is not used in this document, but is used in Part 2 (and thereby also in Part 3).

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

4.2.9
exceptional value
non-numeric value produced (in the specification model) by an arithmetic operation to indicate the occurrence of an exception (or the inexactness of the result)

Exceptional values are not used in subsequent arithmetic processing. (See Clause 5.)

NOTES

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

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

4.2.10
helper function
function used solely to aid in the expression of a requirement

Helper functions are not accessible to the programmer, and are not required to be part of an implementation.

4.2.11
implementation (of this document)
total arithmetic environment presented to a programmer, including hardware, language processors, exception handling facilities, subroutine libraries, other software, and all pertinent documentation

4.2.12
literal
single syntactic entity denoting a constant value

4.2.13
normal value
non-special and non-subnormal value of a floating point datatype F

See F_{N} in Clause 5.2 for a full definition.

4.2.14
notification
process by which a program (or that program's user) is informed that an arithmetic exception has occurred

For example, dividing 2 by 0 results in a notification for **infinitary**. See Clause 6 for details.

4.2.15
numeral
numeric literal

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

4.2.16
operation
function that is intended to be made directly available to the programmer

As opposed to helper functions or theoretical mathematical functions.

4.2.17
pole
argument, x_0, where a given mathematical function, f, is defined, finite, monotone, and continuous in at least one one path of approach towards x_0, and where $\lim\limits_{x \to x_0} f(x)$ is infinite

4.2.18
precision
number of digits in the fraction of a floating point number

(See Clause 5.2.)

4.2.19
rounding
act of computing a result for an operation that is close to the exact result for that operation, but that does not have digits beyond what the target datatype can represent

Note that a suitable representable result may not exist (see Clause 5.2.5).

4.2.20
rounding function
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 is between two adjacent values in X, $rnd(u)$ selects one of those adjacent values.

4.2.21
round to nearest
rounding function, rnd, that when $u \in \mathcal{R}$ is strictly between two adjacent values in X, $rnd(u)$ selects the one nearest u, but if the adjacent values are equidistant from u, either value can be chosen deterministically but in such a way that sign symmetry is preserved ($rnd(-u) = -rnd(u)$)

4.2.22
round toward minus infinity
rounding function, rnd, that when $u \in \mathcal{R}$ is strictly between two adjacent values in X, $rnd(u)$ selects the one less than u

4.2.23
round toward plus infinity
rounding function, rnd, that when $u \in \mathcal{R}$ is strictly between two adjacent values in X, $rnd(u)$ selects the one greater than u

4.2.24
signature (of a function or operation)
argument and result 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 $add_I : I \times I \rightarrow I \cup \{\mathbf{overflow}\}$ states that the operation named add_I shall accept any pair of values in I as input, and when given such input shall return either a single value in I as its output or the exceptional value **overflow** possibly accompanied by a continuation value.

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

NOTE 5 – In particular, IEC 60559 special values are not in F, but must be accepted as arguments if iec_60559_F has the value **true**.

4.2.25
subnormal
denormal (obsolete)
value of a floating point datatype F, or $\mathbf{-0}$, whose absolute value is strictly less than the smallest positive normal value in F ($fminN_F$)

(See F_S in Clause 5.2 for a full definition.)

4.2.26
ulp
unit(s) in the last place (for a given real value and given floating point datatype)
for a value x in $\mathcal{R}$, that has a nearest-closer-to-zero normalised value in F extended to arbitrarily large values, where the normalised value's exponent is t, precision is p_F, and the radix is r_F: the unit is $r_F^{t-p_F}$; for a value x in $\mathcal{R}$, with a nearest-closer-to-zero subnormal value in F, as well as for $\mathbf{-0}$: the unit is $fminD_F$

This value depends on the exponent, the radix, and the precision used in representing the numbers in F. (See Clause 5.2.)

NOTE 6 – For a value that is exactly equal to an integer power of the radix, the ulp is the size of the gap between available values on the side *away* from zero.

5 Specifications for integer and floating point datatypes and operations

An arithmetic datatype consists of a set of values and is accompanied by operations that take values from an arithmetic datatype and return a value in an arithmetic datatype or a boolean value. For any particular arithmetic datatype, the set of non-special values is characterised by a small number of parameters. An exact definition of the value set will be given in terms of these parameters.

Each operation is given a signature and is further specified by a number of cases. These cases may refer to mathematical functions, to other operations, and to helper functions (specified in this document). They also use special values and exceptional values.

Given the datatype's non-special value set, V, the accompanying arithmetic operations will be specified as mathematical functions on V union certain special values that may be in the corresponding implementation datatype. These functions typically return values in V or a special value, but they may instead nominally return exceptional values (that have no arithmetic datatype, and are not to be confused with the special values) that are often specified along with a continuation value. Though nominally listed as a return value, an exceptional value is mathematically really part of a second component of the result, as explained in clause 4.1.6, and is to be handled as a notification as described in clause 6.

The exceptional values used in this document are **underflow**, **inexact**, **overflow**, **infinitary** (generalisation of division-by-zero), and **invalid**. Parts 2 and 3 will also use the exceptional value **absolute_precision_underflow** for the operations that correspond to cyclic functions. For many cases this document specifies which continuation value to use with a specified exceptional value. The continuation value is then expressed in parenthesis after the expression of the exceptional value. For example, **infinitary**($+\infty$) expresses that the exceptional value **infinitary** in that case is to be accompanied by a continuation value of $+\infty$ (unless the binding states differently). In case the notification is by recording in indicators (see Clause 6.2.1), the continuation value is used as the actual return value. This part of ISO/IEC 10967 sometimes leaves the continuation value unspecified, in which case the continuation value is implementation defined.

Whenever an arithmetic operation (as defined in this clause) returns an exceptional value (mathematically, that a non-empty exceptional value set is unioned with the union of exceptions from the arguments, as the exceptional values part of the result), notification of this shall occur as described in Clause 6.

An implementation of a conforming integer or floating point datatype shall include all non-special values defined for that datatype by this document. However, the implementing datatype is permitted to include additional values (for example, and in particular, IEC 60559 special values). This part of ISO/IEC 10967 specifies the behaviour of integer operations when applied to infinitary values, but not for other such additional values. This part of ISO/IEC 10967 specifies the behaviour of floating point operations when applied to IEC 60559 special values, but not for other such additional values.

An implementation of a conforming integer or floating point datatype shall be accompanied by all the operations specified for that datatype by this part of ISO/IEC 10967. Additional operations are explicitly permitted.

The datatype **Boolean** is used for parameters and the results of comparison operations. An implementation is not required by this document to provide a **Boolean** datatype, nor is it required by this part of ISO/IEC 10967 to provide operations on **Boolean** values. However, an implementation shall provide a method of distinguishing **true** from **false** as parameter values and as results of operations.

NOTE – This document requires an implementation to provide methods to access values, operations, and other facilities. Ideally, these methods are provided by a language or binding standard, and the implementation merely cites this standard. Only if a binding standard does not exist, must an individual implementation supply this information on its own. See Annex C.7.

5.1 Integer datatypes and operations

The non-special value set, I, for an integer datatype shall be a subset of $\mathcal{Z}$, characterised by the following parameters:

$bounded_I \in$ **Boolean**	(whether the set I is finite)
$minint_I \in I \cup \{-\infty\}$	(the smallest integer in I if $bounded_I =$ **true**)
$maxint_I \in I \cup \{+\infty\}$	(the largest integer in I if $bounded_I =$ **true**)

In addition, the following parameter characterises one aspect of the special values in the datatype corresponding to I in the implementation:

$hasinf_I \in$ **Boolean**	(whether the corresponding datatype has $-\infty$ and $+\infty$)

NOTE 1 – The first edition of this document also specified the parameter $modulo_I$. A binding may still have a parameter $modulo_I$, and for conformity to this second edition, that parameter is to have the value **false**. Part 2 includes specifications for operations add_wrap_I, sub_wrap_I, and mul_wrap_I. If the parameter $modulo_I$ has the value **true** (non-conforming case), that indicates that the binding binds the basic integer arithmetic operations, for bounded integer datatypes, to the corresponding wrapping operations instead of the add_I, sub_I, and mul_I operations of this document.

If $bounded_I$ is **false**, the set I shall satisfy

$$I = \mathcal{Z}$$

In this case, $hasinf_I$ shall be **true**, the value of $minint_I$ shall be $-\infty$, and the value of $maxint_I$ shall be $+\infty$.

If $bounded_I$ is **true**, then $minint_I \in \mathcal{Z}$ and $maxint_I \in \mathcal{Z}$ and the set I shall satisfy

$$I = \{x \in \mathcal{Z} \mid minint_I \leqslant x \leqslant maxint_I\}$$

and $minint_I$ and $maxint_I$ shall satisfy

$$maxint_I > 0$$

and one of:

$minint_I = 0$,
$minint_I = -maxint_I$, or
$minint_I = -(maxint_I + 1)$

A bounded integer datatype with $minint_I < 0$ is called *signed*. A bounded integer datatype with $minint_I = 0$ is called *unsigned*. An integer datatype in which $bounded_I$ is **false** is *signed*, due to the requirement above.

An implementation may provide more than one integer datatype. A method shall be provided for a program to obtain the values of the parameters $\mathit{bounded}_I$, hasinf_I, minint_I, and maxint_I, for each conforming integer datatype provided.

NOTES

2 The value of hasinf_I does not affect the values of minint_I and maxint_I for bounded integer datatypes.

3 Most traditional programming languages call for bounded integer datatypes. Others allow or require an integer datatype to have an unbounded range. A few languages permit the implementation to decide whether an integer datatype will be bounded or unbounded. (See C.5.1.0.1 for further discussion.)

4 Operations on unbounded integers will not overflow, but may fail due to exhaustion of resources.

5 Unbounded natural numbers are not covered by this document.

5.1.1 Integer result function

If $\mathit{bounded}_I$ is **true**, the mathematical operations $+$, $-$, and $\cdot$ can produce results that lie outside the set I even when given values in I. In such cases, the computational operations add_I, sub_I, neg_I, abs_I, and mul_I shall cause an overflow notification.

In the integer operation specifications below, the handling of overflow is specified via the result_I helper function:

$$\mathit{result}_I : \mathcal{Z} \to I \cup \{\textbf{overflow}\}$$

which is defined by:

$$\begin{aligned}
\mathit{result}_I(x) &= x && \text{if } x \in I \\
&= \textbf{overflow}(-\infty) && \text{if } x \in \mathcal{Z} \text{ and } x \notin I \text{ and } x < 0 \\
&= \textbf{overflow}(+\infty) && \text{if } x \in \mathcal{Z} \text{ and } x \notin I \text{ and } x > 0
\end{aligned}$$

NOTES

1 For integer operations, this document does not specify continuation values for **overflow** when $\mathit{hasinf}_I = \textbf{false}$ nor the continuation values for **invalid**. The binding or implementation must document the continuation value(s) used for such cases (see Clause 8).

2 For the floating point operations in Clause 5.2 a result_F helper function is used to consistently and succinctly express overflow and denormalisation loss cases.

5.1.2 Integer operations

5.1.2.1 Comparisons

For each provided conforming integer datatype, the following operations shall be provided.

$$\mathit{eq}_I : I \times I \to \textbf{Boolean}$$

$$\begin{aligned}
\mathit{eq}_I(x, y) &= \textbf{true} && \text{if } x, y \in I \cup \{-\infty, +\infty\} \text{ and } x = y \\
&= \textbf{false} && \text{if } x, y \in I \cup \{-\infty, +\infty\} \text{ and } x \neq y
\end{aligned}$$

$$\mathit{neq}_I : I \times I \to \textbf{Boolean}$$

$$\begin{aligned}
\mathit{neq}_I(x, y) &= \textbf{true} && \text{if } x, y \in I \cup \{-\infty, +\infty\} \text{ and } x \neq y \\
&= \textbf{false} && \text{if } x, y \in I \cup \{-\infty, +\infty\} \text{ and } x = y
\end{aligned}$$

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

$$
\begin{array}{lll}
lss_I(x,y) & = \mathbf{true} & \text{if } x, y \in I \text{ and } x < y \\
 & = \mathbf{false} & \text{if } x, y \in I \text{ and } x \geqslant y \\
 & = \mathbf{true} & \text{if } x \in I \cup \{-\infty\} \text{ and } y = +\infty \\
 & = \mathbf{true} & \text{if } x = -\infty \text{ and } y \in I \\
 & = \mathbf{false} & \text{if } x \in I \cup \{-\infty, +\infty\} \text{ and } y = -\infty \\
 & = \mathbf{false} & \text{if } x = +\infty \text{ and } y \in I \cup \{+\infty\}
\end{array}
$$

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

$$
\begin{array}{lll}
leq_I(x,y) & = \mathbf{true} & \text{if } x, y \in I \text{ and } x \leqslant y \\
 & = \mathbf{false} & \text{if } x, y \in I \text{ and } x > y \\
 & = \mathbf{true} & \text{if } x \in I \cup \{-\infty, +\infty\} \text{ and } y = +\infty \\
 & = \mathbf{true} & \text{if } x = -\infty \text{ and } y \in I \cup \{-\infty\} \\
 & = \mathbf{false} & \text{if } x \in I \cup \{+\infty\} \text{ and } y = -\infty \\
 & = \mathbf{false} & \text{if } x = +\infty \text{ and } y \in I
\end{array}
$$

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

$$gtr_I(x,y) = lss_I(y,x)$$

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

$$geq_I(x,y) = leq_I(y,x)$$

5.1.2.2 Basic arithmetic

For each provided conforming integer datatype, the following operations shall be provided. If I is unsigned, it is permissible to omit the operations neg_I, abs_I, and $signum_I$.

$neg_I : I \rightarrow I \cup \{\mathbf{overflow}\}$

$$
\begin{array}{lll}
neg_I(x) & = result_I(-x) & \text{if } x \in I \\
 & = +\infty & \text{if } x = -\infty \\
 & = -\infty & \text{if } x = +\infty
\end{array}
$$

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

$$
\begin{array}{lll}
add_I(x,y) & = result_I(x+y) & \text{if } x, y \in I \\
 & = -\infty & \text{if } x \in I \cup \{-\infty\} \text{ and } y = -\infty \\
 & = -\infty & \text{if } x = -\infty \text{ and } y \in I \\
 & = +\infty & \text{if } x \in I \cup \{+\infty\} \text{ and } y = +\infty \\
 & = +\infty & \text{if } x = +\infty \text{ and } y \in I \\
 & = \mathbf{invalid} & \text{if } x = +\infty \text{ and } y = -\infty \\
 & = \mathbf{invalid} & \text{if } x = -\infty \text{ and } y = +\infty
\end{array}
$$

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

$$\begin{array}{lll}
sub_I(x,y) & = result_I(x-y) & \text{if } x,y \in I \\
 & = -\infty & \text{if } x \in I \cup \{-\infty\} \text{ and } y = +\infty \\
 & = -\infty & \text{if } x = -\infty \text{ and } y \in I \\
 & = +\infty & \text{if } x \in I \cup \{+\infty\} \text{ and } y = -\infty \\
 & = +\infty & \text{if } x = +\infty \text{ and } y \in I \\
 & = \textbf{invalid} & \text{if } x = +\infty \text{ and } y = +\infty \\
 & = \textbf{invalid} & \text{if } x = -\infty \text{ and } y = -\infty
\end{array}$$

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

$$\begin{array}{lll}
mul_I(x,y) & = result_I(x \cdot y) & \text{if } x,y \in I \\
 & = +\infty & \text{if } x = +\infty \text{ and } (y = +\infty \text{ or } (y \in I \text{ and } y > 0)) \\
 & = -\infty & \text{if } x = +\infty \text{ and } (y = -\infty \text{ or } (y \in I \text{ and } y < 0)) \\
 & = -\infty & \text{if } x \in I \text{ and } x > 0 \text{ and } y = -\infty \\
 & = +\infty & \text{if } x \in I \text{ and } x < 0 \text{ and } y = -\infty \\
 & = +\infty & \text{if } x = -\infty \text{ and } (y = -\infty \text{ or } (y \in I \text{ and } y < 0)) \\
 & = -\infty & \text{if } x = -\infty \text{ and } (y = +\infty \text{ or } (y \in I \text{ and } y > 0)) \\
 & = -\infty & \text{if } x \in I \text{ and } x < 0 \text{ and } y = +\infty \\
 & = +\infty & \text{if } x \in I \text{ and } x > 0 \text{ and } y = +\infty \\
 & = \textbf{invalid} & \text{if } x \in \{-\infty, +\infty\} \text{ and } y = 0 \\
 & = \textbf{invalid} & \text{if } x = 0 \text{ and } y \in \{-\infty, +\infty\}
\end{array}$$

$abs_I : I \to I \cup \{\textbf{overflow}\}$

$$\begin{array}{lll}
abs_I(x) & = result_I(|x|) & \text{if } x \in I \\
 & = +\infty & \text{if } x \in \{-\infty, +\infty\}
\end{array}$$

$signum_I : I \to \{-1, 1\}$

$$\begin{array}{lll}
signum_I(x) & = 1 & \text{if } (x \in I \text{ and } x \geqslant 0) \text{ or } x = +\infty \\
 & = -1 & \text{if } (x \in I \text{ and } x < 0) \text{ or } x = -\infty
\end{array}$$

NOTE 1 – The first edition of this document specified a slightly different operation $sign_I$. $signum_I$ is consistent with $signum_F$, which in turn is consistent with the branch cuts for the complex trigonometric operations (Part 3).

Integer division with floor and its remainder:

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

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

NOTE 2 – $quot_I(minint_I, -1)$, for a bounded signed integer datatype where $minint_I = -maxint_I - 1$, is the only case where this operation will overflow.

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

$mod_I(x, y)$	$= x - (\lfloor x/y \rfloor \cdot y)$	if $x, y \in I$ and $y \neq 0$
	$= x$	if $x \in I$ and $y \in \{-\infty, +\infty\}$
	$=$ **invalid**	otherwise

NOTES

3 The first edition of this document specified the operations div_I^f, div_I^t, mod_I^a, mod_I^p, rem_I^f, and rem_I^t. However, $div_I^f = quot_I$, and $mod_I^a = rem_I^f = mod_I$. Further, div_I^t, mod_I^p, and rem_I^t are not recommended to be provided, as their use may give rise to late-discovered bugs.

4 Part 2 specifies the related operations $ratio_I$, $residue_I$, $group_I$, and pad_I.

5.2 Floating point datatypes and operations

A floating point datatype shall have a non-special value set F that is a finite subset of $\mathcal{R}$, characterized by the following parameters:

$r_F \in \mathcal{Z}$	(the radix of F)
$p_F \in \mathcal{Z}$	(the precision of F)
$emax_F \in \mathcal{Z}$	(the largest exponent of F)
$emin_F \in \mathcal{Z}$	(the smallest exponent of F)
$denorm_F \in$ **Boolean**	(whether F contains non-zero subnormal values)

In addition, the following parameter characterises the special values in the datatype corresponding to F in the implementation, and the operations in common for this document and IEC 60559:

$iec\_60559_F \in$ **Boolean** (whether the datatype and operations conform to IEC 60559)

NOTES

1 This standard does not advocate any particular representation for floating point values. However, concepts such as *radix*, *precision*, and *exponent* are derived from an abstract model of such values as discussed in Annex C.5.2.

2 The 2011 version of IEC 60559 also uses the parameters *emax* and *emin* (written as E_{max} and E_{min} in the 1989 version). However, those values are respectively one less than the $emax_F$ and $emin_F$ parameters of this document. The latter are, however, in line with the maximum and minimum exponent access variables in several programming languages.

The parameters r_F, p_F, and $denorm_F$ shall satisfy:

$r_F \geqslant 2$
$p_F \geqslant 2 \cdot \max\{1, \lceil \log_{r_F}(2 \cdot \pi) \rceil\}$
$denorm_F = \textbf{true}$

NOTE 3 – The first edition of this document only required for p_F that $p_F \geqslant 2$. The requirement in this edition allows for the use of any floating point type in widespread use and is made so that angles in radians are not too degenerate within the first two cycles, plus and minus, when represented in F.

Furthermore, r_F should be even, and p_F should be such that $p_F \geqslant 2 + \lceil \log_{r_F}(1000) \rceil$.

NOTE 4 – The recommendation that $p_F \geqslant 2 + \lceil \log_{r_F}(1000) \rceil$, which did not occur in the first edition of this document, allows for the use of any floating point type in widespread use and is made so as to allow for a not too coarse angle resolution, for operations in Part 2 and Part 3, anywhere in the interval $[-big\_angle\_r_F, big\_angle\_r_F]$ ($big\_angle\_r_F$ is a parameter introduced in Part 2).

The parameters $emin_F$ and $emax_F$ shall satisfy:

$$1 - r_F^{p_F} \leqslant emin_F \leqslant -1 - p_F$$
$$p_F \leqslant emax_F \leqslant r_F^{p_F} - 1$$

and should satisfy:

$$0 \leqslant emax_F + emin_F \leqslant 4$$

NOTE 5 – The first edition of this document had the wider range requirement $1 - r_F^{p_F} \leqslant emin_F \leqslant 2 - p_F$. The shorter range requirement in this edition of this document allows for the use of any floating point type in widespread use and is made so as to be able to avoid the underflow notification, that is, avoid denormalisation loss, in the specifications for the $expm1_F$ and $ln1p_F$ operations (Part 2) for subnormal arguments (though these operations are still inexact for non-zero subnormal arguments).

Given specific values for r_F, p_F, $emin_F$, $emax_F$, and $denorm_F$, the following sets are defined:

$$F_S = \{s \cdot m \cdot r_F^{e-p_F} \mid s \in \{-1, 1\},\ m, e \in \mathcal{Z},\ 0 \leqslant m < r_F^{p_F-1},\ e = emin_F\}$$

$$F_N = \{s \cdot m \cdot r_F^{e-p_F} \mid s \in \{-1, 1\},\ m, e \in \mathcal{Z},\ r_F^{p_F-1} \leqslant m < r_F^{p_F},\ emin_F \leqslant e \leqslant emax_F\}$$

$$F_E = \{s \cdot m \cdot r_F^{e-p_F} \mid s \in \{-1, 1\},\ m, e \in \mathcal{Z},\ r_F^{p_F-1} \leqslant m < r_F^{p_F},\ emax_F < e\}$$

$$F_L = \{s \cdot m \cdot r_F^{e-p_F} \mid s \in \{-1, 1\},\ m, e \in \mathcal{Z},\ r_F^{p_F-1} \leqslant m < r_F^{p_F},\ e < emin_F\}$$

$$F^\dagger = F_S \cup F_N \cup F_E$$

$$F^\ddagger = F_L \cup F^\dagger$$

$$\begin{aligned} F &= F_S \cup F_N && \text{if } denorm_F = \textbf{true} \\ &= \{0\} \cup F_N && \text{if } denorm_F = \textbf{false} \quad \text{(non-conforming case, see Annex A)} \end{aligned}$$

NOTES

6 $F^\dagger$ is the outwards unbounded extension of F, including in addition all subnormal values that would be in F if $denorm_F$ were **true**. $F^\dagger$ will be used in defining rounding for operations.

7 $F^\ddagger$ is the unbounded extension of F.

The elements of F_N are called *normal* floating point values. The elements of F_S, as well as the special value $\mathbf{-0}$, are called *subnormal* floating point values.

NOTE 8 – The terms *normal* and *subnormal* refer to the mathematical values involved, not to any method of representation.

An implementation may provide more than one floating point datatype.

For each of the parameters r_F, p_F, $emin_F$, $emax_F$, $denorm_F$, and iec_60559_F, and for each conforming floating point datatype provided, a method shall be provided for a program to obtain the value of the parameter.

NOTE 9 – The conditions placed upon the parameters r_F, p_F, $emin_F$, and $emax_F$ are sufficient to guarantee that the abstract model of F is well-defined and contains its own parameters, as well as enabling the avoidance of denormalisation loss (in particular for $expm1_F$ and $ln1p_F$ of Part 2). More stringent conditions are needed to produce a computationally useful floating point datatype. These are design decisions which are beyond the scope of this document. (See Annex C.5.2.)

5.2.1 Conformity to IEC 60559

The parameter iec_60559_F shall be true only when the datatype corresponding to F and the relevant operations completely conform to the requirements of IEC 60559. F may correspond to any of the floating point datatypes defined in IEC 60559.

When iec_60559_F has the value **true**, all the facilities required by IEC 60559 shall be provided. Methods shall be provided for a program to access each such facility. In addition, documentation shall be provided to describe these methods, and all implementation choices. When iec_60559_F has the value **true**, all operations and values common to this document and IEC 60559 shall satisfy the requirements of both standards.

NOTES

1 IEC 60559 is also known as IEEE 754 [34].

2 The IEC 60559 facilities include: values for infinities and NaNs, extended comparisons, rounding towards positive or negative infinity, an exceptions (including inexact) recorded in indicators. See annex B for more information.

3 IEC 60559, third edition, specifies $r_F = 2$ or $r_F = 10$, as well as values for maximum and minimum exponents and precision for the floating point datatypes it specifies.

4 If iec_60559_F is **true**, then $denorm_F$ must also be **true**. Note that $denorm_F =$ **false** is non-conforming also to this document.

5.2.2 Range and granularity constants

The range and granularity of F is characterized by the following derived constants:

$$fmax_F = \max\ F = (1 - r_F^{-p_F}) \cdot r_F^{emax_F}$$

$$fminN_F = \min\ \{z \in F_{\mathrm{N}} \mid z > 0\} = r^{emin_F - 1}$$

$$fminD_F = \min\ \{z \in F_{\mathrm{S}} \mid z > 0\} = r^{emin_F - p_F}$$

$$fmin_F = \min\ \{z \in F \mid z > 0\} \begin{array}{ll} = fminD_F & \text{if } denorm_F = \textbf{true} \\ = fminN_F & \text{if } denorm_F = \textbf{false} \quad \text{(non-conforming case)} \end{array}$$

$$epsilon_F = r_F^{1-p_F} \qquad \text{(the relative spacing in } F^{\ddagger} \text{ between adjacent values)}$$

For each of the derived constants $fmax_F$, $fminN_F$, $fmin_F$, and $epsilon_F$, and for each conforming floating point datatype provided, a method shall be provided for a program to obtain the value of the derived constant.

5.2.3 Approximate operations

The operations (specified below) add_F, sub_F, mul_F, div_F and, upon denormalisation loss, $scale_{F,I}$ are approximations of exact mathematical operations. They differ from their mathematical counterparts, not only in that they may accept special values as arguments, but also in that

a) they may produce "rounded" results,

b) they may produce a special value (even without notification, or for values in F as arguments), and

c) they may produce notifications (with values in F or special values as continuation values).

The approximate floating point operations are specified *as if* they were computed in three stages:

1) compute the exact mathematical answer (if there is any), or determine if an **infinitary** or **invalid** notification is required,

2) round the exact answer (if there is any) to p_F digits of precision in the radix r_F (the precision will be less if the rounded answer is subnormal), maybe producing a $\mathbf{-0}$ as the rounded result, and

3) determine if an **underflow**, **overflow** (using $\mathbf{+\infty}$ or $\mathbf{-\infty}$ as continuation value if available), or **inexact** notification is required.

These stages will be modelled by basic and elementary mathematical functions (stage 1) and two helper functions: $nearest_F$ (stage 2) and $result_F$ (stage 3). These helper functions are not visible to the programmer and are not required to be part of the implementation, just like exact mathematical functions are not required to be part of an implementation. An actual implementation need not perform the above stages at all, merely return a result (or produce a notification and a continuation value) as if it had.

5.2.4 Rounding and rounding constants

Define the helper function $e_F : \mathcal{R} \to \mathcal{Z}$ such that

$$\begin{aligned} e_F(x) &= \lfloor log_{r_F}(|x|) \rfloor + 1 && \text{if } |x| \geqslant fminN_F \\ &= emin_F && \text{if } |x| < fminN_F \end{aligned}$$

Define the helper function $u_F : \mathcal{R} \to F^\dagger$ such that

$$u_F(x) = r_F^{e_F(x) - p_F}$$

NOTES

1 The value $e_F(x)$ is the exponent for values in $F^\dagger$ in the immediate neighbourhood of x (which need not be in $F^\dagger$) When x is in $]-r_F \cdot fminN_F, r_F \cdot fminN_F[$, then $e_F(x)$ is $emin_F$ regardless of x.

2 The value $u_F(x)$ is the absolute distance between adjacent values in $F^\dagger$ in the immediate neighbourhood of x (which need not be in $F^\dagger$). When x is in $]-r_F \cdot fminN_F, r_F \cdot fminN_F[$, then $u_F(x)$ is $fminD_F$ regardless of x.

3 When x is on an exponent boundary of $F^\dagger$, the neighbourhood (mentioned in the previous two notes) is that which is away from zero.

For floating point operations, rounding is the process of taking an exact result in $\mathcal{R}$ and producing a p_F-digit approximation.

NOTE 4 – In Annex A of this document, and in Parts 2 and 3 of ISO/IEC 10967, the "exact result" may be a prerounding approximation, through approximation helper functions.

The $nearest_F$, $down_F$, and up_F helper functions are introduced to model the rounding process: The floating point helper function

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

is the rounding function that rounds to nearest, ties rounded to even last digit. The floating point helper function

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

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

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

is the rounding function that rounds towards positive infinity.

If, for some $x \in \mathcal{R}$ and some $i \in \mathcal{Z}$, such that $|x| < fminN_F$, $|x \cdot r_F^i| \geqslant fminN_F$, and rounding function $rnd : \mathcal{R} \to F^\dagger$, the formula

$$rnd(x \cdot r_F^i) = rnd(x) \cdot r_F^i$$

does not hold, then *rnd* is said to have a *denormalisation loss* at x.

NOTE 5 – If $nearest'_F : \mathcal{R} \to F^\ddagger$ is a round to nearest function to $F^\ddagger$, and $nearest'_F(x) \neq nearest_F(x)$, then $nearest_F$ has a denormalisation loss at x. Similarly for up_F and $down_F$.

5.2.5 Floating point result function

A floating point operation produces a rounded result or a notification. The decision is based on the computed result (either before or after rounding).

The $result_F$ helper function is introduced to model this decision. $result_F$ is partially implementation defined. $result_F$ has a signature:

$$result_F : \mathcal{R} \times (\mathcal{R} \to F^\dagger) \to F \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$$

NOTE 1 – The first input to $result_F$ is the computed result before rounding, and the second input is the rounding function to be used.

For the overflow cases for the three roundings $nearest_F$, up_F, and $down_F$, and for $x \in \mathcal{R}$, the following shall apply:

$$\begin{array}{lll}
result_F(x, nearest_F) & = \mathbf{overflow}(+\infty) & \text{if } nearest_F(x) > fmax_F \\
result_F(x, nearest_F) & = \mathbf{overflow}(fmax_F) \text{ or } \mathbf{inexact}(fmax_F) & \\
 & & \text{if } nearest_F(x) = fmax_F \text{ and } x > fmax_F \\
result_F(x, nearest_F) & = \mathbf{overflow}(-\infty) & \text{if } nearest_F(x) < -fmax_F \\
result_F(x, nearest_F) & = \mathbf{overflow}(-fmax_F) \text{ or } \mathbf{inexact}(-fmax_F) & \\
 & & \text{if } nearest_F(x) = -fmax_F \text{ and } x < -fmax_F \\
result_F(x, up_F) & = \mathbf{overflow}(+\infty) & \text{if } up_F(x) > fmax_F \\
result_F(x, up_F) & = \mathbf{overflow}(-fmax_F) & \text{if } up_F(x) < -fmax_F \\
result_F(x, up_F) & = \mathbf{overflow}(-fmax_F) \text{ or } \mathbf{inexact}(-fmax_F) & \\
 & & \text{if } up_F(x) = -fmax_F \text{ and } x < -fmax_F \\
result_F(x, down_F) & = \mathbf{overflow}(-\infty) & \text{if } down_F(x) < -fmax_F \\
result_F(x, down_F) & = \mathbf{overflow}(fmax_F) & \text{if } down_F(x) > fmax_F \\
result_F(x, down_F) & = \mathbf{overflow}(fmax_F) \text{ or } \mathbf{inexact}(fmax_F) & \\
 & & \text{if } down_F(x) = fmax_F \text{ and } x > fmax_F
\end{array}$$

For other cases for either of the thee rounding functions as *rnd*, and for $x \in [-fmax_F, fmax_F]$, the following shall apply:

$$\begin{array}{lll}
result_F(x, rnd) & = x & \text{if } x \in F \\
 & = \mathbf{inexact}(rnd(x)) & \text{if } x \neq rnd(x) \text{ and } fminN_F \leqslant |x| \leqslant fmax_F \\
 & = \mathbf{underflow}(\mathbf{-0}) & \text{if } denorm_F = \mathbf{false} \text{ and } -fminN_F < x < 0 \\
 & = \mathbf{underflow}(0) & \text{if } denorm_F = \mathbf{false} \text{ and } 0 < x < fminN_F \\
 & = \mathbf{underflow}(\mathbf{-0}) & \text{if } denorm_F = \mathbf{true} \text{ and } x < 0 \text{ and } rnd(x) = 0
\end{array}$$

$$
\begin{aligned}
&= \mathbf{inexact}(rnd(x)) \text{ or } \mathbf{underflow}(rnd(x)) \\
&\qquad\qquad \text{if } denorm_F = \mathbf{true} \text{ and } x \neq rnd(x) \text{ and } |x| < fminN_F \\
&\qquad\qquad\quad \text{and } rnd \text{ has no denormalisation loss at } x \\
&= \mathbf{underflow}(rnd(x)) \\
&\qquad\qquad \text{otherwise}
\end{aligned}
$$

NOTES

2 Overflow may be detected before or after rounding. If overflow is detected before rounding, the bounds for overflow are independent of rounding.

3 The notifications **underflow** and **overflow** implies inexact result. When **inexact** notifications are supported, and when recording notifications in indicators, recording an **underflow** or an **overflow** notification implies also recording an **inexact** notification.

4 An implementation is allowed to choose between $\mathbf{inexact}(rnd(x))$ and $\mathbf{underflow}(rnd(x))$ for values of x in the interval $]-fminN_F, fminN_F[$ where x is not in F and there is no denormalisation loss at x. However, a subnormal value without an underflow notification can be chosen only if $denorm_F$ is **true** and no denormalisation loss occurs at x.

5 In the non-conforming case $denorm_F = \mathbf{false}$, neither rounding is heeded in case of **underflow**.

An implementation shall document how the choice between $\mathbf{inexact}(rnd(x))$ and $\mathbf{underflow}(rnd(x))$ is made. Different floating point types may have different versions of $result_F$.

Define the $no\_result_{F\to F'}$ and $no\_result2_{F\to F'}$ helper functions:

$no\_result_{F\to F'} : F \to \{\mathbf{invalid}\}$

$$
\begin{aligned}
no\_result_{F\to F'}(x) \quad & \\
&= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
&= \mathbf{qNaN} && \text{if } x \text{ is a quiet NaN} \\
&= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \text{ is a signalling NaN}
\end{aligned}
$$

$no\_result2_{F\to F'} : F \times F \to \{\mathbf{invalid}\}$

$$
\begin{aligned}
no\_result2_{F\to F'}(x, y) \quad & \\
&= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x, y \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
&= \mathbf{qNaN} && \text{if at least one of } x \text{ and } y \text{ is a quiet NaN and} \\
& && \quad \text{neither a signalling NaN} \\
&= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \text{ is a signalling NaN or } y \text{ is a signalling NaN}
\end{aligned}
$$

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

NOTE 6 – The handling of special values, if available, other than NaNs, infinities, and $\mathbf{-0}$, is left unspecified by this document.

5.2.6 Floating point operations

5.2.6.1 Comparisons

For each provided conforming floating point datatype, the following operations shall be provided.

$eq_F : F \times F \to \mathbf{Boolean}$

$$\begin{array}{lll}
eq_F(x,y) & = \mathbf{true} & \text{if } x,y \in F \cup \{\mathbf{-\infty},\mathbf{+\infty}\} \text{ and } x = y \\
 & = \mathbf{false} & \text{if } x,y \in F \cup \{\mathbf{-\infty},\mathbf{+\infty}\} \text{ and } x \neq y \\
 & = eq_F(0,y) & \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty},\mathbf{-0},\mathbf{+\infty}\} \\
 & = eq_F(x,0) & \text{if } x \in F \cup \{\mathbf{-\infty},\mathbf{+\infty}\} \text{ and } y = \mathbf{-0} \\
 & = \mathbf{false} & \text{if } x \text{ is a quiet NaN and y is not a signalling NaN} \\
 & = \mathbf{false} & \text{if } y \text{ is a quiet NaN and x is not a signalling NaN} \\
 & = \mathbf{invalid}(\mathbf{false}) & \text{if } x \text{ is a signalling NaN or } y \text{ is a signalling NaN}
\end{array}$$

$neq_F : F \times F \to \mathbf{Boolean}$

$$\begin{array}{lll}
neq_F(x,y) & = \mathbf{true} & \text{if } x,y \in F \cup \{\mathbf{-\infty},\mathbf{+\infty}\} \text{ and } x \neq y \\
 & = \mathbf{false} & \text{if } x,y \in F \cup \{\mathbf{-\infty},\mathbf{+\infty}\} \text{ and } x = y \\
 & = neq_F(0,y) & \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty},\mathbf{-0},\mathbf{+\infty}\} \\
 & = neq_F(x,0) & \text{if } y = \mathbf{-0} \text{ and } x \in F \cup \{\mathbf{-\infty},\mathbf{+\infty}\} \\
 & = \mathbf{true} & \text{if } x \text{ is a quiet NaN and } y \text{ is not a signalling NaN} \\
 & = \mathbf{true} & \text{if } y \text{ is a quiet NaN and } x \text{ is not a signalling NaN} \\
 & = \mathbf{invalid}(\mathbf{true}) & \text{if } x \text{ is a signalling NaN or } y \text{ is a signalling NaN}
\end{array}$$

$lss_F : F \times F \to \mathbf{Boolean}$

$$\begin{array}{lll}
lss_F(x,y) & = \mathbf{true} & \text{if } x,y \in F \text{ and } x < y \\
 & = \mathbf{false} & \text{if } x,y \in F \text{ and } x \geqslant y \\
 & = lss_F(0,y) & \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty},\mathbf{-0},\mathbf{+\infty}\} \\
 & = lss_F(x,0) & \text{if } y = \mathbf{-0} \text{ and } x \in F \cup \{\mathbf{-\infty},\mathbf{+\infty}\} \\
 & = \mathbf{true} & \text{if } x = \mathbf{-\infty} \text{ and } y \in F \cup \{\mathbf{+\infty}\} \\
 & = \mathbf{false} & \text{if } x = \mathbf{+\infty} \text{ and } y \in F \cup \{\mathbf{-\infty},\mathbf{+\infty}\} \\
 & = \mathbf{false} & \text{if } x \in F \cup \{\mathbf{-\infty}\} \text{ and } y = \mathbf{-\infty} \\
 & = \mathbf{true} & \text{if } x \in F \text{ and } y = \mathbf{+\infty} \\
 & = \mathbf{invalid}(\mathbf{false}) & \text{if } x \text{ is a NaN or } y \text{ is a NaN}
\end{array}$$

$leq_F : F \times F \to \mathbf{Boolean}$

$$\begin{array}{lll}
leq_F(x,y) & = \mathbf{true} & \text{if } x,y \in F \text{ and } x \leqslant y \\
 & = \mathbf{false} & \text{if } x,y \in F \text{ and } x > y \\
 & = leq_F(0,y) & \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty},\mathbf{-0},\mathbf{+\infty}\} \\
 & = leq_F(x,0) & \text{if } y = \mathbf{-0} \text{ and } x \in F \cup \{\mathbf{-\infty},\mathbf{+\infty}\} \\
 & = \mathbf{true} & \text{if } x = \mathbf{-\infty} \text{ and } y \in F \cup \{\mathbf{-\infty},\mathbf{+\infty}\} \\
 & = \mathbf{false} & \text{if } x = \mathbf{+\infty} \text{ and } y \in F \cup \{\mathbf{-\infty}\} \\
 & = \mathbf{false} & \text{if } x \in F \text{ and } y = \mathbf{-\infty} \\
 & = \mathbf{true} & \text{if } x \in F \cup \{\mathbf{+\infty}\} \text{ and } y = \mathbf{+\infty} \\
 & = \mathbf{invalid}(\mathbf{false}) & \text{if } x \text{ is a NaN or } y \text{ is a NaN}
\end{array}$$

$gtr_F : F \times F \to \mathbf{Boolean}$

$gtr_F(x,y) = lss_F(y,x)$

$geq_F : F \times F \to \mathbf{Boolean}$

$geq_F(x,y) = leq_F(y,x)$

$isnegzero_F : F \rightarrow \mathbf{Boolean}$

$$\begin{array}{lll} isnegzero_F(x) & = \mathbf{true} & \text{if } x = \mathbf{-0} \\ & = \mathbf{false} & \text{if } x \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \\ & = \mathbf{invalid}(\mathbf{false}) & \text{if } x \text{ is a NaN} \end{array}$$

$istiny_F : F \rightarrow \mathbf{Boolean}$

$$\begin{array}{lll} istiny_F(x) & = \mathbf{true} & \text{if } (x \in F \text{ and } |x| < fminN_F) \text{ or } x = \mathbf{-0} \\ & = \mathbf{false} & \text{if } (x \in F \text{ and } |x| \geqslant fminN_F) \text{ or } x \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \\ & = \mathbf{invalid}(\mathbf{false}) & \text{if } x \text{ is a NaN} \end{array}$$

$isnan_F : F \rightarrow \mathbf{Boolean}$

$$\begin{array}{lll} isnan_F(x) & = \mathbf{false} & \text{if } x \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ & = \mathbf{true} & \text{if } x \text{ is a quiet NaN} \\ & = \mathbf{invalid}(\mathbf{true}) & \text{if } x \text{ is a signalling NaN} \end{array}$$

$issignan_F : F \rightarrow \mathbf{Boolean}$

$$\begin{array}{lll} issignan_F(x) & = \mathbf{false} & \text{if } x \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ & = \mathbf{false} & \text{if } x \text{ is a quiet NaN} \\ & = \mathbf{true} & \text{if } x \text{ is a signalling NaN} \end{array}$$

5.2.6.2 Basic arithmetic

For each provided conforming floating point datatype, the following round to nearest operations shall be provided, and the round towards negative and positive infinity operations should be provided. For the non-conforming case that $denorm_F = \mathbf{false}$, the operations below using up_F or $down_F$ as rounding shall *not* be provided.

NOTE 1 – If $denorm_F = \mathbf{false}$, then any result that is smaller than $fminN_F$ is replaced by zero. This implies that neither rounding direction (nearest, up, down) is heeded, doing "flush to zero" for would-be subnormal results. Thus if $denorm_F = \mathbf{false}$, the directed rounding operations would be unreliable for interval arithmetic, as well as other uses. That is why the directed rounding operations are not to be provided when $denorm_F = \mathbf{false}$.

The operations in this clause are specified only for the case that $r_F = r_{F'}$, $denorm_F = denorm_{F'}$, $iec\_60559_F = iec\_60559_{F'}$. If $iec\_60559_F = \mathbf{false}$ then the operations are required only if $F = F'$. The $add_{F \rightarrow F'}$ and $sub_{F \rightarrow F'}$ operations can underflow only if $denorm_{F'} = \mathbf{false}$ (non-conforming case) or $emin_F - p_F < emin_{F'} - p_{F'}$.

$neg_F : F \rightarrow F \cup \{\mathbf{-0}\}$

$$\begin{array}{lll} neg_F(x) & = -x & \text{if } x \in F \text{ and } x \neq 0 \\ & = \mathbf{-0} & \text{if } x = 0 \\ & = 0 & \text{if } x = \mathbf{-0} \\ & = \mathbf{-\infty} & \text{if } x = \mathbf{+\infty} \\ & = \mathbf{+\infty} & \text{if } x = \mathbf{-\infty} \\ & = no\_result_{F \rightarrow F}(x) & \text{otherwise} \end{array}$$

$add_{F \rightarrow F'} : F \times F \rightarrow F' \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$

$$\begin{array}{lll}
add_{F\to F'}(x,y) & = result_{F'}(x+y, nearest_{F'}) & \\
 & & \text{if } x,y \in F \\
 & = \mathbf{-0} & \text{if } x = \mathbf{-0} \text{ and } y = \mathbf{-0} \\
 & = add_{F\to F'}(0,y) & \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \\
 & = add_{F\to F'}(x,0) & \text{if } x \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \text{ and } y = \mathbf{-0} \\
 & = \mathbf{+\infty} & \text{if } x = \mathbf{+\infty} \text{ and } y \in F \cup \{\mathbf{+\infty}\} \\
 & = \mathbf{+\infty} & \text{if } x \in F \text{ and } y = \mathbf{+\infty} \\
 & = \mathbf{-\infty} & \text{if } x = \mathbf{-\infty} \text{ and } y \in F \cup \{\mathbf{-\infty}\} \\
 & = \mathbf{-\infty} & \text{if } x \in F \text{ and } y = \mathbf{-\infty} \\
 & = no\_result2_{F\to F'}(x,y) & \text{otherwise}
\end{array}$$

$add^{\uparrow}_{F\to F'} : F \times F \to F' \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$

$$\begin{array}{lll}
add^{\uparrow}_{F\to F'}(x,y) & = result_{F'}(x+y, up_{F'}) & \text{if } x,y \in F \\
 & = add_{F\to F'}(x,y) & \text{otherwise}
\end{array}$$

$add^{\downarrow}_{F\to F'} : F \times F \to F' \cup \{\mathbf{-0}, \mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$

$$\begin{array}{lll}
add^{\downarrow}_{F\to F'}(x,y) & = result_{F'}(x+y, down_{F'}) & \text{if } x,y \in F \text{ and } (x+y \neq 0 \text{ or } x = 0) \\
 & = \mathbf{-0} & \text{if } x,y \in F \text{ and } x+y = 0 \text{ and } x \neq 0 \\
 & = \mathbf{-0} & \text{if } add_{F\to F'}(x,y) = 0 \text{ and } (x = \mathbf{-0} \text{ or } y = \mathbf{-0}) \\
 & = add_{F\to F'}(x,y) & \text{otherwise}
\end{array}$$

$sub_{F\to F'} : F \times F \to F' \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$

$$sub_{F\to F'}(x,y) = add_{F\to F'}(x, neg_F(y))$$

$sub^{\uparrow}_{F\to F'} : F \times F \to F' \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$

$$sub^{\uparrow}_{F\to F'}(x,y) = add^{\uparrow}_{F\to F'}(x, neg_F(y))$$

$sub^{\downarrow}_{F\to F'} : F \times F \to F' \cup \{\mathbf{-0}, \mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$

$$sub^{\downarrow}_{F\to F'}(x,y) = add^{\downarrow}_{F\to F'}(x, neg_F(y))$$

$mul_{F\to F'} : F \times F \to F' \cup \{\mathbf{-0}, \mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$

$$\begin{array}{lll}
mul_{F\to F'}(x,y) & = result_{F'}(x \cdot y, nearest_{F'}) & \\
 & & \text{if } x,y \in F \text{ and } x \neq 0 \text{ and } y \neq 0 \\
 & = 0 & \text{if } x = 0 \text{ and } y \in F \text{ and } y \geqslant 0 \\
 & = \mathbf{-0} & \text{if } x = 0 \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-0}) \\
 & = \mathbf{-0} & \text{if } x = \mathbf{-0} \text{ and } y \in \text{F and } y \geqslant 0 \\
 & = 0 & \text{if } x = \mathbf{-0} \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-0}) \\
 & = 0 & \text{if } x \in F \text{ and } x > 0 \text{ and } y = 0 \\
 & = \mathbf{-0} & \text{if } x \in F \text{ and } x < 0 \text{ and } y = 0 \\
 & = \mathbf{-0} & \text{if } x \in F \text{ and } x > 0 \text{ and } y = \mathbf{-0} \\
 & = 0 & \text{if } x \in F \text{ and } x < 0 \text{ and } y = \mathbf{-0} \\
 & = \mathbf{+\infty} & \text{if } x = \mathbf{+\infty} \text{ and } ((y \in F \text{ and } y > 0) \text{ or } y = \mathbf{+\infty}) \\
 & = \mathbf{-\infty} & \text{if } x = \mathbf{+\infty} \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-\infty})
\end{array}$$

$$\begin{array}{lll}
 & = -\infty & \text{if } x = -\infty \text{ and } ((y \in F \text{ and } y > 0) \text{ or } y = +\infty) \\
 & = +\infty & \text{if } x = -\infty \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = -\infty) \\
 & = +\infty & \text{if } x \in F \text{ and } x > 0 \text{ and } y = +\infty \\
 & = -\infty & \text{if } x \in F \text{ and } x < 0 \text{ and } y = +\infty \\
 & = -\infty & \text{if } x \in F \text{ and } x > 0 \text{ and } y = -\infty \\
 & = +\infty & \text{if } x \in F \text{ and } x < 0 \text{ and } y = -\infty \\
 & = \mathit{no\_result2}_{F \to F'}(x, y) & \text{otherwise}
\end{array}$$

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

$$\begin{array}{lll}
\mathit{mul}^{\uparrow}_{F \to F'}(x, y) & = \mathit{result}_{F'}(x \cdot y, \mathit{up}_{F'}) & \text{if } x, y \in F \text{ and } x \neq 0 \text{ and } y \neq 0 \\
 & = \mathit{mul}_{F \to F'}(x, y) & \text{otherwise}
\end{array}$$

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

$$\begin{array}{lll}
\mathit{mul}^{\downarrow}_{F \to F'}(x, y) & = \mathit{result}_{F'}(x \cdot y, \mathit{down}_{F'}) & \text{if } x, y \in F \text{ and } x \neq 0 \text{ and } y \neq 0 \\
 & = \mathit{mul}_{F \to F'}(x, y) & \text{otherwise}
\end{array}$$

$$\mathit{div}_{F \to F'} : F \times F \to F' \cup \{\mathbf{-0}, \mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
\mathit{div}_{F \to F'}(x, y) & = \mathit{result}_{F'}(x/y, \mathit{nearest}_{F'}) & \\
 & & \text{if } x, y \in F \text{ and } x \neq 0 \text{ and } y \neq 0 \\
 & = 0 & \text{if } x = 0 \text{ and } y \in F \text{ and } y > 0 \\
 & = \mathbf{-0} & \text{if } x = 0 \text{ and } y \in F \text{ and } y < 0 \\
 & = \mathbf{-0} & \text{if } x = \mathbf{-0} \text{ and } y \in F \text{ and } y > 0 \\
 & = 0 & \text{if } x = \mathbf{-0} \text{ and } y \in F \text{ and } y < 0 \\
 & = \mathbf{infinitary}(+\infty) & \text{if } x \in F \text{ and } x > 0 \text{ and } y = 0 \\
 & = \mathbf{infinitary}(-\infty) & \text{if } x \in F \text{ and } x < 0 \text{ and } y = 0 \\
 & = \mathbf{infinitary}(-\infty) & \text{if } x \in F \text{ and } x > 0 \text{ and } y = \mathbf{-0} \\
 & = \mathbf{infinitary}(+\infty) & \text{if } x \in F \text{ and } x < 0 \text{ and } y = \mathbf{-0} \\
 & = 0 & \text{if } x \in F \text{ and } x \geqslant 0 \text{ and } y = +\infty \\
 & = \mathbf{-0} & \text{if } x \in F \text{ and } x \geqslant 0 \text{ and } y = -\infty \\
 & = \mathbf{-0} & \text{if } ((x \in F \text{ and } x < 0) \text{ or } x = \mathbf{-0}) \text{ and } y = +\infty \\
 & = 0 & \text{if } ((x \in F \text{ and } x < 0) \text{ or } x = \mathbf{-0}) \text{ and } y = -\infty \\
 & = +\infty & \text{if } x = +\infty \text{ and } y \in F \text{ and } y \geqslant 0 \\
 & = -\infty & \text{if } x = -\infty \text{ and } y \in F \text{ and } y \geqslant 0 \\
 & = -\infty & \text{if } x = +\infty \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-0}) \\
 & = +\infty & \text{if } x = -\infty \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-0}) \\
 & = \mathit{no\_result2}_{F \to F'}(x, y) & \text{otherwise}
\end{array}$$

$$\mathit{div}^{\uparrow}_{F \to F'} : F \times F \to F' \cup \{\mathbf{-0}, \mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
\mathit{div}^{\uparrow}_{F \to F'}(x, y) & = \mathit{result}_{F'}(x/y, \mathit{up}_{F'}) & \text{if } x, y \in F \text{ and } x \neq 0 \text{ and } y \neq 0 \\
 & = \mathit{div}_{F \to F'}(x, y) & \text{otherwise}
\end{array}$$

$$\mathit{div}^{\downarrow}_{F \to F'} : F \times F \to F' \cup \{\mathbf{-0}, \mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{invalid}\}$$

$$\begin{array}{lll}
\mathit{div}^{\downarrow}_{F \to F'}(x, y) & = \mathit{result}_{F'}(x/y, \mathit{down}_{F'}) & \text{if } x, y \in F \text{ and } x \neq 0 \text{ and } y \neq 0 \\
 & = \mathit{div}_{F \to F'}(x, y) & \text{otherwise}
\end{array}$$

$abs_F : F \to F$

$$\begin{array}{lll}
abs_F(x) & = |x| & \text{if } x \in F \\
 & = 0 & \text{if } x = \mathbf{-0} \\
 & = \mathbf{+\infty} & \text{if } x \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \\
 & = no\_result_{F \to F}(x) & \text{otherwise}
\end{array}$$

$signum_F : F \to F$

$$\begin{array}{lll}
signum_F(x) & = 1 & \text{if } (x \in F \text{ and } x \geqslant 0) \text{ or } x = \mathbf{+\infty} \\
 & = -1 & \text{if } (x \in F \text{ and } x < 0) \text{ or } x \in \{\mathbf{-0}, \mathbf{-\infty}\} \\
 & = no\_result_{F \to F}(x) & \text{otherwise}
\end{array}$$

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

$$\begin{array}{lll}
residue_F(x, y) & = result_F(x - (\mathrm{round}(x/y) \cdot y), nearest_F) & \\
 & & \text{if } x, y \in F \text{ and } y \neq 0 \text{ and} \\
 & & \quad (x \geqslant 0 \text{ or } x - (\mathrm{round}(x/y) \cdot y) \neq 0) \\
 & = \mathbf{-0} & \text{if } x, y \in F \text{ and } y \neq 0 \text{ and} \\
 & & \quad x < 0 \text{ and } x - (\mathrm{round}(x/y) \cdot y) = 0 \\
 & = \mathbf{-0} & \text{if } x = \mathbf{-0} \text{ and } y \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\} \text{ and } y \neq 0 \\
 & = x & \text{if } x \in F \text{ and } y \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \\
 & = no\_result2_{F \to F}(x, y) & \text{otherwise}
\end{array}$$

NOTE 2 – The $residue_F$ operation is informally known as "IEEE remainder".

$sqrt_{F \to F'} : F \to F' \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{invalid}\}$

$$\begin{array}{lll}
sqrt_{F \to F'}(x) & = result_{F'}(\sqrt{x}, nearest_{F'}) & \\
 & & \text{if } x \in F \text{ and } x \geqslant 0 \\
 & = x & \text{if } x \in \{\mathbf{-0}, \mathbf{+\infty}\} \\
 & = no\_result_{F \to F'}(x) & \text{otherwise}
\end{array}$$

$sqrt^{\uparrow}_{F \to F'} : F \to F' \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{invalid}\}$

$$\begin{array}{lll}
sqrt^{\uparrow}_{F \to F'}(x, y) & = result_{F'}(\sqrt{x}, up_{F'}) & \text{if } x \in F \text{ and } x \geqslant 0 \\
 & = sqrt_{F \to F'}(x) & \text{otherwise}
\end{array}$$

$sqrt^{\downarrow}_{F \to F'} : F \to F' \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{invalid}\}$

$$\begin{array}{lll}
sqrt^{\downarrow}_{F \to F'}(x, y) & = result_{F'}(\sqrt{x}, down_{F'}) & \text{if } x \in F \text{ and } x \geqslant 0 \\
 & = sqrt_{F \to F'}(x) & \text{otherwise}
\end{array}$$

5.2.6.3 Value dissection

For each provided floating point type, the following operations shall be provided. For the non-conforming case of $denorm_F = \mathbf{false}$, ulp_F may underflow, and the operations $succ_F$ and $pred_F$ shall *not* be provided.

The $exponent_{F \to I}$ and $scale_{F,I}$ operations are specified for an integer datatype I where $minint_I < emin_F - p_F$ and $maxint_I > emax_F$.

$$exponent_{F \to I} : F \to I \cup \{\mathbf{infinitary}\}$$

$$\begin{aligned}
exponent_{F \to I}(x) &= \lfloor \log_{r_F}(|x|) \rfloor + 1 && \text{if } x \in F \text{ and } x \neq 0 \\
&= \mathbf{infinitary}(-\infty) && \text{if } x \in \{\mathbf{-0}, 0\} \\
&= \mathbf{+\infty} && \text{if } x \in \{\mathbf{-\infty}, \mathbf{+\infty}\} \\
&= \mathbf{qNaN} && \text{if } x \text{ is a quiet NaN} \\
&= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \text{ is a signalling NaN}
\end{aligned}$$

NOTES

1 Since most integer datatypes cannot represent infinitary or NaN values, documented out of range finite integer values of the correct sign may be used instead of the infinities here.

2 The related IEC 60559 operation *logb* returns a floating point value, to guarantee the representability of the infinitary (and NaN) return values.

$$fraction_F : F \to F$$

$$\begin{aligned}
fraction_F(x) &= x / r_F^{exponent_{F \to \mathcal{Z}}(x)} && \text{if } x \in F \text{ and } x \neq 0 \\
&= x && \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, 0, \mathbf{+\infty}\} \\
&= no\_result_{F \to F}(x) && \text{otherwise}
\end{aligned}$$

$$scale_{F,I} : F \times I \to F \cup \{\mathbf{underflow}, \mathbf{overflow}\}$$

$$\begin{aligned}
scale_{F,I}(x, n) &= result_F(x \cdot r_F^n, nearest_F) \\
& && \text{if } x \in F \text{ and } n \in I \\
&= mul_{F \to F}(x, 0) && \text{if } n = \mathbf{-\infty} \\
&= x && \text{if } n = \mathbf{-0} \\
&= mul_{F \to F}(x, convert_{I \to F}(n)) \\
& && \text{otherwise}
\end{aligned}$$

$$succ_F : F \to F \cup \{\mathbf{overflow}\}$$

$$\begin{aligned}
succ_F(x) &= result_F(\min\ \{z \in F^{\dagger} \mid z > x\}, nearest_F) \\
& && \text{if } x \in F \text{ and } x \neq -fmin_F \text{ and } x \neq 0 \\
&= -fmax_F && \text{if } x = \mathbf{-\infty} \\
&= \mathbf{-0} && \text{if } x = -fmin_F \\
&= succ_F(0) && \text{if } x = \mathbf{-0} \\
&= fmin_F && \text{if } x = 0 \\
&= \mathbf{+\infty} && \text{if } x = \mathbf{+\infty} \\
&= no\_result_{F \to F}(x) && \text{otherwise}
\end{aligned}$$

$$pred_F : F \to F \cup \{\mathbf{overflow}\}$$

$$pred_F(x) = neg_F(succ_F(neg_F(x)))$$

$$ulp_F : F \to F \cup \{\mathbf{underflow}\}$$

$$\begin{aligned}
ulp_F(x) &= result_F(u_F(x), nearest_F) \\
& && \text{if } x \in F \\
&= ulp_F(0) && \text{if } x = \mathbf{-0} \\
&= no\_result_{F \to F}(x) && \text{otherwise}
\end{aligned}$$

5.2.6.4 Value splitting

For each provided floating point type, the following operations shall be provided. The $trunc_{F,I}$ and $round_{F,I}$ operations are specified for an integer type I where $maxint_I > p_F$.

$$intpart_F : F \to F \cup \{\mathbf{-0}\}$$

$$\begin{array}{lll} intpart_F(x) & = \lfloor x \rfloor & \text{if } x \in F \text{ and } x \geqslant 0 \\ & = neg_F(intpart_F(-x)) & \text{if } x \in F \text{ and } x < 0 \\ & = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ & = no\_result_{F \to F}(x) & \text{otherwise} \end{array}$$

$$fractpart_F : F \to F \cup \{\mathbf{-0}\}$$

$$\begin{array}{lll} fractpart_F(x) & = x - \lfloor x \rfloor & \text{if } x \in F \text{ and } x \geqslant 0 \\ & = neg_F(fractpart_F(-x)) & \text{if } x \in F \text{ and } x < 0 \\ & = x & \text{if } x = \mathbf{-0} \\ & = no\_result_{F \to F}(x) & \text{otherwise} \end{array}$$

$$trunc_{F,I} : F \times I \to F \cup \{\mathbf{-0}\}$$

$$\begin{array}{lll} trunc_{F,I}(x, n) & = \lfloor x / r_F^{e_F(x)-n} \rfloor \cdot r_F^{e_F(x)-n} & \text{if } x \in F \text{ and } x \geqslant 0 \text{ and } n \in I \\ & = neg_F(trunc_{F,I}(-x, n)) & \text{if } x \in F \text{ and } x < 0 \text{ and } n \in I \\ & = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ & = no\_result2_{F \to F}(x, n) & \text{otherwise} \end{array}$$

$$round_{F,I} : F \times I \to F \cup \{\mathbf{-0}, \mathbf{overflow}\}$$

$$\begin{array}{lll} round_{F,I}(x, n) & = result_F(\text{round}(x / r_F^{e_F(x)-n}) \cdot r_F^{e_F(x)-n}, nearest_F) & \\ & & \text{if } x \in F \text{ and } x \geqslant 0 \text{ and } n \in I \\ & = neg_F(round_{F,I}(-x, n)) & \text{if } x \in F \text{ and } x < 0 \text{ and } n \in I \\ & = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ & = no\_result2_{F \to F}(x, n) & \text{otherwise} \end{array}$$

5.3 Operations for conversion between numeric datatypes

Numeric conversion between different representation forms for integer and floating point values can take place under a number of different circumstances. E.g.:

a) explicit or implicit conversion between different numeric datatypes conforming to this part of ISO/IEC 10967;

b) explicit or implicit conversion between different numeric datatypes only one of which conforms to this part of ISO/IEC 10967;

c) explicit or implicit conversion between a character string and a numeric datatype.

The last case includes outputting a numeric value as a character string, inputting a numeric value from a character string source, and converting a numeral in the source program to a value in a numeric datatype (see clause 5.4). This part of ISO/IEC 10967 covers only the cases where at least one of the source and target of a conversion is a numeric datatype conforming to this part of ISO/IEC 10967.

When a character string is involved as either source or target of a conversion, this part of ISO/IEC 10967 does not specify the lexical syntax for the numerals parsed or formed. A binding standard should specify the lexical syntax or syntaxes for these numerals, and, when appropriate, how the lexical syntax for the numerals can be altered. This could include which set of digits to use in a numeral position system (Latin-Arabic digits, Arabic-Indic digits, traditional Thai digits, Devanagari digits, etc.). With the exception of the radix used in numerals expressing non-integer values, differences in lexical syntactic details that do not affect the value in $\mathcal{R}$ denoted by the numerals will not affect the result of a conforming conversion.

Output of floating point values is quite often to a fixed point format. Therefore conversion from a floating point datatype to a fixed point datatype is included in this clause.

Character string representations for integer values can include representations for $\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, and quiet NaNs. Character string representations for floating point and fixed point values should have formats for $\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, and quiet NaNs. For both integer and floating point values, character strings that are not numerals nor special values according to the lexical syntax used, shall be regarded as signalling NaNs when used as source of a numerical conversion.

For the cases where one of the datatypes involved in the conversion does not conform to this part of ISO/IEC 10967, the values of some numeric datatype parameters need to be inferred. For integers, one need to infer the value for *bounded*, and if that is **true** then also values for *maxint* and *minint*, and for string formats also the *radix*. For floating point values, one need to infer the values for r, p, and *emax* or *emin*. In case a precise determination is not possible, values that are 'safe' for that instance should be used. 'Safe' values for otherwise undetermined inferred parameters are such that

a) if the value resulting from the conversion is converted back to the source datatype by a conversion conforming to this part of ISO/IEC 10967 the original value, or a close approximation, should be regenerated if possible, and

b) overflow and underflow are avoided if possible.

If, and only if, a specified infinite special value result cannot be represented in the target datatype, the infinity result shall be interpreted as implying the **infinitary** notification. If, and only if, a specified NaN special value result cannot be represented in the target datatype, the NaN result shall be interpreted as implying the **invalid** notification. If, and only if, a specified $\mathbf{-0}$ special value result cannot be represented in the target datatype, the $\mathbf{-0}$ result shall be interpreted as 0.

5.3.1 Integer to integer conversions

Let I and I' be non-special value sets for integer datatypes. At least one of the datatypes corresponding to I and I' conforms to this part of ISO/IEC 10967.

The $convert_{I \to I'}$ operation:

$$convert_{I \to I'} : I \to I' \cup \{\mathbf{overflow}\}$$

$$\begin{aligned}
convert_{I \to I'}(x) &= result_{I'}(x) && \text{if } x \in I \\
&= x && \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
&= \mathbf{qNaN} && \text{if } x \text{ is a quiet NaN} \\
&= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \text{ is a signalling NaN}
\end{aligned}$$

5.3.2 Floating point to integer conversions

Let F be the non-special value set for a floating point datatype. Let I be the non-special value set for an integer datatype. At least one of the datatypes corresponding to F and I conforms to this part of ISO/IEC 10967.

The $floor_{F \to I}$, $rounding_{F \to I}$, and $ceiling_{F \to I}$ operations:

$floor_{F \to I} : F \to I \cup \{\mathbf{overflow}\}$

$$
\begin{array}{lll}
floor_{F \to I}(x) & = result_I(\lfloor x \rfloor) & \text{if } x \in F \\
 & = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
 & = \mathbf{qNaN} & \text{if } x \text{ is a quiet NaN} \\
 & = \mathbf{invalid}(\mathbf{qNaN}) & \text{if } x \text{ is a signalling NaN}
\end{array}
$$

$rounding_{F \to I} : F \to I \cup \{\mathbf{-0}, \mathbf{overflow}\}$

$rounding_{F \to I}(x)$

$$
\begin{array}{ll}
= result_I(\mathrm{round}(x)) & \text{if } x \in F \text{ and } (x \geqslant 0 \text{ or } \mathrm{round}(x) \neq 0) \\
= \mathbf{-0} & \text{if } x \in F \text{ and } x < 0 \text{ and } \mathrm{round}(x) = 0 \\
= x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
= \mathbf{qNaN} & \text{if } x \text{ is a quiet NaN} \\
= \mathbf{invalid}(\mathbf{qNaN}) & \text{if } x \text{ is a signalling NaN}
\end{array}
$$

$ceiling_{F \to I} : F \to I \cup \{\mathbf{-0}, \mathbf{overflow}\}$

$$
\begin{array}{lll}
ceiling_{F \to I}(x) & = result_I(\lceil x \rceil) & \text{if } x \in F \text{ and } (x \geqslant 0 \text{ or } \lceil x \rceil \neq 0) \\
 & = \mathbf{-0} & \text{if } x \in F \text{ and } x < 0 \text{ and } \lceil x \rceil = 0 \\
 & = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
 & = \mathbf{qNaN} & \text{if } x \text{ is a quiet NaN} \\
 & = \mathbf{invalid}(\mathbf{qNaN}) & \text{if } x \text{ is a signalling NaN}
\end{array}
$$

5.3.3 Integer to floating point conversions

Let I be the non-special value set for an integer datatype. Let F be the non-special value set for a floating point datatype. At least one of the datatypes corresponding to I and F conforms to this part of ISO/IEC 10967.

The $convert_{I \to F}$ operation:

$convert_{I \to F} : I \to F \cup \{\mathbf{inexact}, \mathbf{overflow}\}$

$$
\begin{array}{lll}
convert_{I \to F}(x) & = result_F(x, nearest_F) & \text{if } x \in I \\
 & = x & \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\
 & = \mathbf{qNaN} & \text{if } x \text{ is a quiet NaN} \\
 & = \mathbf{invalid}(\mathbf{qNaN}) & \text{if } x \text{ is a signalling NaN}
\end{array}
$$

The $convert^{\uparrow}_{I \to F}$ and $convert^{\downarrow}_{I \to F}$ operations:

$convert^{\uparrow}_{I \to F} : I \to F \cup \{\mathbf{inexact}, \mathbf{overflow}\}$

$$
\begin{array}{lll}
convert^{\uparrow}_{I \to F}(x) & = result_F(x, up_F) & \text{if } x \in I \\
 & = convert_{I \to F}(x) & \text{otherwise}
\end{array}
$$

$$convert^{\downarrow}_{I\to F} : I \to F \cup \{\mathbf{inexact}, \mathbf{overflow}\}$$

$$\begin{aligned}convert^{\downarrow}_{I\to F}(x) &= result_F(x, down_F) && \text{if } x \in I\\ &= convert_{I\to F}(x) && \text{otherwise}\end{aligned}$$

5.3.4 Floating point to floating point conversions

Let F and F' be non-special value sets for floating point datatypes. At least one of the datatypes corresponding to F and F' conforms to this part of ISO/IEC 10967.

The $convert_{F\to F'}$ operation:

$$convert_{F\to F'} : F \to F' \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$$

$$\begin{aligned}convert_{F\to F'}(x) &= result_{F'}(x, nearest_{F'}) && \text{if } x \in F\\ &= x && \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\}\\ &= \mathbf{qNaN} && \text{if } x \text{ is a quiet NaN}\\ &= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \text{ is a signalling NaN}\end{aligned}$$

NOTE – Modern techniques allow, on the average, efficient conversion with a maximum error of 0.5 ulp even when the radices differ. C99 [15], for instance, requires that all floating point value conversion is done with a maximum error of 0.5 ulp.

The $convert^{\uparrow}_{F\to F'}$ and $convert^{\downarrow}_{F\to F'}$ operations:

$$convert^{\uparrow}_{F\to F'} : F \to F' \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$$

$$\begin{aligned}convert^{\uparrow}_{F\to F'}(x) &= result_{F'}(x, up_{F'}) && \text{if } x \in F\\ &= convert_{F\to F'} && \text{otherwise}\end{aligned}$$

$$convert^{\downarrow}_{F\to F'} : F \to F' \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$$

$$\begin{aligned}convert^{\downarrow}_{F\to F'}(x) &= result_{F'}(x, down_{F'}) && \text{if } x \in F\\ &= convert_{F\to F'} && \text{otherwise}\end{aligned}$$

5.3.5 Floating point to fixed point conversions

Let F be the non-special value set for a floating point datatype conforming to this part of ISO/IEC 10967. Let D be the non-special value set for a fixed point datatype.

A fixed point datatype D is a subset of $\mathcal{R}$, characterised by a radix, $r_D \in \mathcal{Z}$ ($\geqslant 2$), a density, $d_D \in \mathcal{Z}$ ($\geqslant 0$), and if it is bounded, a maximum positive value, $dmax_D \geqslant 1$. Given these values, the following sets are defined:

$$D^* = \{n/(r_D^{d_D}) \ \mid\ n \in Z\}$$

$$\begin{aligned}&D = D^* && \text{if } D \text{ is not bounded}\\ &D = D^* \cap [-dmax_D, dmax_D] && \text{if } D \text{ is bounded}\end{aligned}$$

NOTE 1 – D corresponds to **scaled**(r_D, d_D) in ISO/IEC 11404 *Information technology – General-Purpose Datatypes (GPD)* [10]. However, that standard does not specify a $dmax_D$ parameter.

The fixed point rounding helper functions:

$$nearest_D : \mathcal{R} \to D^*$$

is the rounding function that rounds to nearest, ties round to even last digit.

$$up_D : \mathcal{R} \to D^*$$

is the rounding function that rounds towards positive infinity.

$$down_D : \mathcal{R} \to D^*$$

is the rounding function that rounds towards negative infinity.

The fixed point result helper function, $result_D$, is like $result_F$, but for a fixed point datatype. It will return **overflow** if the rounded result is not representable:

$$result_D : \mathcal{R} \times (\mathcal{R} \to D^*) \to D \cup \{\mathbf{inexact}, \mathbf{overflow}\}$$

$$\begin{aligned}
result_D(x, rnd) &= \mathbf{overflow}(\mathbf{+\infty}) && \text{if } x \in \mathcal{R} \text{ and } rnd(x) \notin D \text{ and } x > 0\\
&= \mathbf{overflow}(\mathbf{-\infty}) && \text{if } x \in \mathcal{R} \text{ and } rnd(x) \notin D \text{ and } x < 0\\
&= x && \text{if } rnd(x) \in D \text{ and } rnd(x) = x\\
&= \mathbf{inexact}(rnd(x)) && \text{if } rnd(x) \in D \text{ and } rnd(x) \neq x \text{ and}\\
& && \quad (rnd(x) \neq 0 \text{ or } 0 < x)\\
&= \mathbf{inexact}(\mathbf{-0}) && \text{if } rnd(x) = 0 \text{ and } x < 0
\end{aligned}$$

The $convert_{F \to D}$ operation:

$$convert_{F \to D} : F \to D \cup \{\mathbf{inexact}, \mathbf{overflow}\}$$

$$\begin{aligned}
convert_{F \to D}(x) &= result_D(x, nearest_D) && \text{if } x \in F\\
&= x && \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\}\\
&= \mathbf{qNaN} && \text{if } x \text{ is a quiet NaN}\\
&= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \text{ is a signalling NaN}
\end{aligned}$$

NOTES

2 The datatype D need not be visible in the programming language. D may be a subtype of strings, according to some format. Even so, no datatype for strings need be present in the programming language.

3 This covers, among other things, "output" of floating point datatype values, to fixed point string formats. E.g. a binding may say that `float_to_fixed_string(`x`, `m`, `n`)` is bound to $convert_{F \to S_{m,n}}(x)$ where $S_{m,n}$ is strings of length m, representing fixed point values in radix 10 with n decimals. The binding should also detail how NaNs, signed zeroes and infinities are represented in $S_{m,n}$, as well as the precise format of the strings representing ordinary values. (Note that if the length of the target string is limited, the conversion may overflow.)

The $convert^{\uparrow}_{F \to D}$ and $convert^{\downarrow}_{F \to D}$ operations:

$$convert^{\uparrow}_{F \to D} : F \to D \cup \{\mathbf{inexact}, \mathbf{overflow}\}$$

$$\begin{aligned}
convert^{\uparrow}_{F \to D}(x) &= result_D(x, up_D) && \text{if } x \in F\\
&= convert_{F \to D}(x) && \text{otherwise}
\end{aligned}$$

$$convert^{\downarrow}_{F \to D} : F \to D \cup \{\mathbf{inexact}, \mathbf{overflow}\}$$

$$\begin{aligned}
convert^{\downarrow}_{F \to D}(x) &= result_D(x, down_D) && \text{if } x \in F\\
&= convert_{F \to D}(x) && \text{otherwise}
\end{aligned}$$

5.3.6 Fixed point to floating point conversions

Let D be the non-special value set for a fixed point datatype. Let F be the non-special value set for a floating point datatype conforming to this part of ISO/IEC 10967.

The $convert_{D \to F}$ operation:

$$convert_{D \to F} : D \to F \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$$

$$\begin{aligned} convert_{D \to F}(x) &= result_F(x, nearest_F) && \text{if } x \in D \\ &= x && \text{if } x \in \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\} \\ &= \mathbf{qNaN} && \text{if } x \text{ is a quiet NaN} \\ &= \mathbf{invalid}(\mathbf{qNaN}) && \text{if } x \text{ is a signalling NaN} \end{aligned}$$

NOTE – This covers, among other things, "input" of floating point datatype values, from fixed point string formats. E.g. a binding may say that `string_to_float(`s`)` is bound to $convert_{S_{m,n} \to F}(s)$ where $S_{m,n}$ is strings of length m, where m is the length of s, and n is the number of digits after the "decimal symbol" represented in $S_{m,n}$, as well as the precise format of the strings representing ordinary values.

The $convert^{\uparrow}_{D \to F}$ and $convert^{\downarrow}_{D \to F}$ operations:

$$convert^{\uparrow}_{D \to F} : D \to F \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$$

$$\begin{aligned} convert_{D \to F}(x) &= result_F(x, up_F) && \text{if } x \in D \\ &= convert_{D \to F}(x) && \text{otherwise} \end{aligned}$$

$$convert^{\downarrow}_{D \to F} : D \to F \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$$

$$\begin{aligned} convert_{D \to F}(x) &= result_F(x, down_F) && \text{if } x \in D \\ &= convert_{D \to F}(x) && \text{otherwise} \end{aligned}$$

5.4 Numerals as operations in a programming language

NOTE – Numerals in strings, or input, is covered by the conversion operations in clause 5.3.

Each numeral is a parameterless operation. Thus, this clause introduces a very large number of operations, since the number of numerals is in principle infinite.

5.4.1 Numerals for integer datatypes

Let I' be a non-special value set for integer numerals for the datatype corresponding to I.

An integer numeral, denoting an abstract value n in $I' \cup \{\mathbf{-0}, \mathbf{+\infty}, \mathbf{-\infty}, \mathbf{qNaN}, \mathbf{sNaN}\}$, for an integer datatype with non-special value set I, shall result in

$$convert_{I' \to I}(n)$$

For each integer datatype conforming to this document and made directly available, there shall be integer numerals with radix 10.

For each radix for numerals made available for a bounded integer datatype with non-special value set I, there shall be integer numerals for all non-negative values of I. For each radix for numerals made available for an unbounded integer datatype, there shall be integer numerals for all non-negative integer values smaller than 10^{20}.

For each integer datatype made directly available and that may have special values:

a) There should be a numeral for positive infinity. There shall be a numeral for positive infinity if there is a positive infinity in the integer datatype.

b) There should be numerals for quiet and signalling NaNs.

5.4.2 Numerals for floating point datatypes

Let D be a non-special value set for fixed point numerals for the datatype corresponding to F. Let F' be a non-special value set for floating point numerals for the datatype corresponding to F.

A fixed point numeral, denoting an abstract value x in $D \cup \{\mathbf{-0}, \mathbf{+\infty}, \mathbf{-\infty}, \mathbf{qNaN}, \mathbf{sNaN}\}$, for a floating point datatype with non-special value set F, shall result in

$convert_{D \to F}(x)$

A floating point numeral, denoting an abstract value x in $F' \cup \{\mathbf{-0}, \mathbf{+\infty}, \mathbf{-\infty}, \mathbf{qNaN}, \mathbf{sNaN}\}$, for a floating point datatype with non-special value set F, shall result in

$convert_{F' \to F}(x)$

For each floating point datatype conforming to this document and made directly available, there should be radix 10 floating point numerals, and there shall be radix 10 fixed point numerals.

For each radix for fixed point numerals made available for a floating point datatype, there shall be numerals for all bounded precision and bounded range expressible non-negative values of $\mathcal{R}$. At least a precision (d_D) of 20 should be available. At least a range ($dmax_D$) of 10^{20} should be available.

For each radix for floating point numerals made available for a floating point datatype with non-special value set F, there shall be numerals for all bounded precision and bounded range expressible non-negative values of $\mathcal{R}$. The precision and range bounds for the numerals shall be large enough to allow all non-negative values of F to be reachable.

For each floating point datatype made directly available and that may have special values:

a) There should be a numeral for positive infinity. There shall be a numeral for positive infinity if there is a positive infinity in the floating point datatype.

b) There should be numerals for quiet and signalling NaNs.

The conversion operations used for numerals as operations should be the same as those used by default for converting strings to values in conforming integer or floating point datatypes.

6 Notification

6.1 Model for handling of notifications

Notification (as that term is used in ISO/IEC 10967) is the process by which a user or program is informed that an arithmetic operation, on given arguments, has some problem associated with it. Specifically, a notification shall occur when any arithmetic operation returns an exceptional value as defined in clause 5.

Logically there is a set of exceptional values associated with each value (not just arithmetic values). An operation returns a computed result together with the union of the arguments's sets of exceptional values and the set of exceptional values produced by the operation itself.

What in clause 5 is written as $add_I : I \times I \rightarrow I \cup \{\mathbf{overflow}\}$, should really be written as $add_I : (I \times \mathcal{P}(E)) \times (I \times \mathcal{P}(E)) \rightarrow (I \times \mathcal{P}(E))$, where E is the set of exception values, and $\mathcal{P}$ is powerset, and for each case of $add_I(\langle x, s_1\rangle, \langle y, s_2\rangle)$, return $s_1 \cup s_2$ as the second component and the first component is the computed value, except for the overflow case where the second component is $s_1 \cup s_2 \cup \{\mathbf{overflow}\}$ and the first component is then the continuation value. Since being explicit about this for every operation specification would clutter the specifications, the specifications in ISO/IEC 10967 are implicit about this handling of exceptional values.

Reproducing this nominal behaviour (a special case of recording in indicators, clause 6.2.1) may be prohibitively inefficient. Therefore the notification alternatives below relax this nominal behaviour. The maximum extension of the notification handling in these alternatives is a runtime thread (or similar construct), except when communicating values between threads, but may be more limited as specified in a binding standard.

6.2 Notification alternatives

Three alternatives for notification are provided in ISO/IEC 10967 The requirements are:

a) Notification by recording in indicators (see clause 6.2.1) shall be supplied, and should be the default way of handling notifications.

b) Notification by alteration of control flow (see clause 6.2.2) should be supplied in conjunction with any language which provides support for exception handling.

c) Notification by termination with message (see clause 6.2.3) (a special case of the second alternative) may be supplied.

NOTE – This is different from the first edition of this document, in which all implementations were required to supply the last alternative, but were given a choice between the first and the second based on whether the language supported exception handling or not.

6.2.1 Notification by recording in indicators

An implementation shall provide for support of this alternative, and it should be the default alternative for notification handling.

Notification consists of two elements: a prompt recording in the relevant indicators of the fact that an arithmetic operation returned an exceptional value, and means for the program or system to interrogate or modify the recording in those indicators at a subsequent time.

This notification alternative has indicators, which represent sets of exceptional values (which need not be just arithmetic ones). But the indicators need not be directly associated with values, but instead with determinable sets of values. However, computations that occur in parallel (logically or actually) must not interfere with each others's indicators. At the start of a computation the indicators for that computation are all clear (that is, the set of notifications recorded is empty).

NOTES

1 The "set of values" may thus be "all the values computed by a thread". Not just "output values", but all values computed by the thread. The "sets of values" may be subsets of the values computed by a thread, by the rules of the programming language or system.

2 When values (of any type) are communicated between threads, notification recordings are ideally communicated along with the values, and recorded in the receiving thread. This is preferably done automatically, needing no extra code from the programmer.

3 Computations that are completely ignored, e.g. speculative threads that are not taken, or timed out threads without output, will have their indicator recordings ignored too.

The set of indicators shall consist of at least six indicators, one for each of the exceptional values that may be returned by an arithmetic operation as defined in clause 5: **inexact**, **underflow**, **overflow**, **infinitary**, **invalid**, and **absolute_precision_underflow**.

NOTE 4 – Part 1 does not use **absolute_precision_underflow**, but it is used in Part 2 and Part 3.

Consider a set E including at least six elements corresponding to the six exceptional values used in this document: {**absolute_precision_underflow**, **inexact**, **underflow**, **overflow**, **infinitary**, **invalid**} $\subseteq E$. An implementation is permitted to expand the set E to include additional notification indicators beyond the six listed in this document.

Let *Ind* be a type whose values represent the subsets of E. An implementation shall provide an embedding of *Ind* into a programming language type. In addition, a method shall be provided for denoting each of the values of *Ind*, either as constants or via computation. The individual notifications as well as the set of all notifications shall be available as named constants.

Let *Ctx* (context) be a type whose values represent the subsets of E and possibly other contextual values (unspecified by this part of ISO/IEC 10967). A value of type *Ctx* holds the current indicators (and possibly other data) for the current thread or part of a thread. Multiple threads shall not share a *Ctx* value, but a thread shall have at least one and may have multiple *Ctx* values. Note that this is the model used in ISO/IEC 10967 for recording of notifications, and a type corresponding to *Ctx* need not be available, nor need such a type be present in an implementation.

The relevant indicators shall be set in the current *Ctx* value when any arithmetic operation returns exceptional values as defined in clause 5. When either of the exceptional values **underflow** or **overflow** is returned, not only shall the corresponding indicator be set, but also the indicator for **inexact** shall be set. Once set in a *Ctx*, an indicator shall be cleared only by explicit action (see *clear_indicators* below) of the program.

NOTE 5 – The status flags required by IEC 60559 are an example of this form of notification, *provided* that the status flags for different computations (microthreads, threads, programs, scripts, similar) are kept separate, and joined only when results of computations are joined.

When an arithmetic operation returns exceptional values as defined in clause 5, in addition to recording the event, an implementation shall provide a *continuation value* for the result of the failed arithmetic operation, and continue execution from that point. In many cases this document specifies the continuation value. The continuation value shall be implementation specified if this document does not specify one.

NOTE 6 – The infinities and NaNs produced by an IEC 60559 system are examples of values not in F which can be used as continuation values.

The following four operations shall be provided:

clear_indicators:	$Ctx \times Ind \rightarrow Ctx$
set_indicators:	$Ctx \times Ind \rightarrow Ctx$
current_indicators:	$Ctx \rightarrow Ind$
test_indicators:	$Ctx \times Ind \rightarrow Boolean$

The *Ctx* value result for the set and clear operations are used as replacement for the input *Ctx* value. That is, these two operations may change the state of the context. The *Ctx* value is usually the one for "the current thread" or similar construct, and is then an implicit argument to these operations.

For every value S in *Ind*, the above four operations shall behave as follows:

$clear\_indicators(C, S)$	in the context C clear each of the indicators referred in S
$set\_indicators(C, S)$	in the context C set each of the indicators referred in S
$current\_indicators(C)$	return the set of indicators that are set in the context C
$test\_indicators(C, S)$	**true** if any of the indicators referred in S are set in the context C

If either of the indicators for **underflow** or **overflow** is set by *set_indicators*, not only shall that indicator be set, but also the indicator for **inexact** shall be set. Indicators that are not referred to in S shall not be altered by *clear_indicators* nor *set_indicators* except as just mentioned, and no indicators are altered by *current_indicators* nor *test_indicators*.

NOTE 7 – No changes to the specifications of a language standard are required to implement this alternative for notification. The recordings can be implemented in system software. The operations for interrogating and manipulating the recording can be contained in a system library, and invoked as library routine calls.

The implementation shall not allow a program to complete successfully with an indicator that is set in the indicators of any context of the final thread. However, it is permissible for a binding to except **underflow** and **inexact** from hindering the successful completion of a program. Unsuccessful completion of a program shall be reported to the user of that program in an unambiguous and "hard to ignore" manner (see 6.2.3).

6.2.2 Notification by alteration of control flow

An implementation should provide this alternative for any language that provides a mechanism for handling of exceptions. This alternative is allowed (with system support) even in the absence of such a mechanism. This alternative can be applied to some notifications, while other notifications are dealt with by recording in indicators. In particular, notification by alteration of control flow should not be used for **inexact** nor for **underflow** (unless **underflow** implies "flush to zero").

Notification consists of prompt alteration of the control flow of the program to execute user provided exception handling code. The manner in which the exception handling code is specified and the capabilities of such exception handling code (including whether it is possible to resume the operation which caused the notification) is the province of the language standard, not this arithmetic standard.

If no exception handling code is provided for a particular occurrence of the return of an exceptional value as defined in clause 5, that fact shall be reported to the user of that program in an unambiguous and "hard to ignore" manner (see 6.2.3).

6.2.3 Notification by termination with message

An implementation may provide this alternative, which also serves as a back-up if the programmer has not provided the necessary code for handling of a notification by alteration of control flow.

Notification consists of prompt delivery of a "hard to ignore" message, followed by termination of execution of the program. Any such message should identify the cause of the notification and the operation responsible.

NOTE – The phrase "hard to ignore" is intended to discourage writing messages to log files (which are rarely read), or setting program variables (which disappear when the program completes).

6.3 Delays in notification

Notification may be momentarily delayed for performance reasons, but should take place as close as practical to the attempt to perform the responsible operation. When notification is delayed, it is permitted to merge notifications of different occurrences of the return of the same exceptional value into a single notification. However, it is not permissible to generate duplicate or spurious notifications.

In connection with notification, "prompt" means before the occurrence of a significant program event. For the recording in indicators in 6.2.1, a significant program event is an attempt by the program (or system) to access the indicators, or the termination of the program. For alteration of control flow described in 6.2.2, the definition of a significant event is language dependent, is likely to depend upon the scope or extent of the exception handling mechanisms, and must therefore be provided by language standards or by language binding standards. For termination with message described in 6.2.3, the definition of a significant event is again language dependent, but would include producing output visible to humans or other programs.

NOTE – Roughly speaking, "prompt" should at least imply "in time to prevent an erroneous response to the exception".

6.4 User selection of alternative for notification

A conforming implementation shall provide a means for a user or program to select among the alternate notification mechanisms provided. It should be possible to select different notification alternatives for different exceptional values. The choice of an appropriate means, such as compiler options, is left to the implementation.

The language or binding standard should specify the notification alternatives to be used in the absence of a user choice. The notification alternatives used in the absence of a user choice shall be documented.

7 Relationship with language standards

A computing system often provides arithmetic datatypes within the context of a standard programming language. The requirements of this document shall be in addition to those imposed by the relevant programming language standards.

This document does not define the syntax of arithmetic expressions. However, programmers need to know how to reliably access the operations defined in this document.

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 used to invoke each operation specified in Clause 5.

NOTE 2 – For example, integer equality ($eq_I(i, j)$) might be invoked as

`i = j`	in Pascal [25] and Ada [11]
`i == j`	in C [15], C++ [17], Java [62], and Fortran [21]
`i .EQ. j`	in Fortran [21]
`(= i j)`	in Common Lisp [37] and ISLisp [22]

An implementation shall document the semantics of arithmetic expressions in terms of compositions of the operations specified in clause 5.

NOTE 3 – For example, if x, y, and z are declared to be single precision (SP) reals, and calculation is done in single precision, then the expression

```
x + y < z
```

might translate to

$$lss_{SP}(add_{SP \to SP}(x, y), z)$$

If the language in question did all computations in double precision (DP), the above expression might translate to

$$lss_{DP}(add_{DP \to DP}(convert_{SP \to DP}(x), convert_{SP \to DP}(y)), convert_{SP \to DP}(z))$$

or

$$lss_{DP}(add_{SP \to DP}(x, y), convert_{SP \to DP}(z))$$

Alternatively, if x was declared to be an integer (J), and computations done in single precision, the above expression might translate to

$$lss_{SP}(add_{SP \to SP}(convert_{J \to SP}(x), y), z)$$

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 (including changes in precision).

b) Reordering of operations, such as the application of associative or distributive rules (note that these rules are not exact for floating point operations and may result in different notifications both for integer and floating point expressions).

c) Replacing operations (or entire subexpressions) with others, such as "`2*x`" $\to$ "`x+x`" or "`x/c`" $\to$ "`x*(1/c)`" (the latter is not exactly valid for floating point). (The example expressions here are expressed in some programming language, they are not mathematical expressions.)

d) Evaluating constant subexpressions at compile time.

e) Eliminating unneeded subexpressions.

Only transformations which alter the semantics of an expression (the values produced (including special values), and the notifications generated (including continuation values)) need be documented. Only the kinds of permitted transformations need be documented. It is not necessary to describe the specific choice of transformations that will be applied to a particular expression.

The textual scope of such transformations shall be documented, and any mechanisms that provide programmer control over this process should be documented as well.

NOTE 4 – It is highly desirable that programming languages intended for numerical use provide means for limiting the transformations applied to particular arithmetic expressions.

8 Documentation requirements

In order to conform to this part of ISO/IEC 10967, an implementation shall include documentation providing the following information to programmers.

a) A list of the provided integer and floating point types that conform to this part of ISO/IEC 10967.

b) For each conforming integer type, the values of the parameters: $bounded_I$, $minint_I$, $maxint_I$, and $hasinf_I$. (See 5.1.)

c) For each conforming floating point type, the values of the parameters: r_F, p_F, $emin_F$, $emax_F$, $denorm_F$, and iec_60559_F. (See 5.2.)

d) For each conforming unsigned integer type I, which (if any) of the operations neg_I, abs_I, and $signum_I$ are omitted for that type. (See 5.1.2.2.)

e) For each pair of types, a list of conversion operations provided.

f) For each conforming floating point type F, the full definition of $result_F$. (See 5.2.5.)

g) For each implementation defined continuation value, the value used.

h) The notation for invoking each operation provided by this part of ISO/IEC 10967. (See 5.1.2, 5.2.6, and 5.4.)

i) The translation of arithmetic expressions into combinations of operations provided by this part of ISO/IEC 10967, including any use made of higher precision. (See clause 7.)

j) For each conforming integer type, the method for a program to obtain the values of the parameters: $bounded_I$, $minint_I$, $maxint_I$, and $hasinf_I$. (See 5.1.)

k) For each conforming floating point type, the method for a program to obtain the values of the parameters: r_F, p_F, $emin_F$, $emax_F$, $denorm_F$, and iec_60559_F. (See 5.2.)

l) For each conforming floating point type, the method for a program to obtain the values of the derived constants $fmax_F$, $fmin_F$, $fminN_F$, $epsilon_F$.

m) The methods used for notification, and the information made available about the notification. (See clause 6.)

n) The means for selecting among the notification methods, and the notification method used in the absence of a user selection. (See 6.4.)

o) When "recording in indicators" is the method of notification, the type used to represent Ind, the method for denoting the values of Ind (the association of these values with the subsets of E must be clear), and the notation for invoking each of the four "indicator" operations, as well as the names of the empty indicator set and the set of all indicators. (See 6.2.1.)

p) When "recording in indicators" is the method of notification, the mechanism for transferring indicators along with values transferred when communicating values between different computations (such as threads).

q) For each conforming floating point type where iec_60559_F is **true**, and for each of the facilities required by IEC 60559, the method available to the programmer to exercise that facility. (See 5.2.1 and Annex B.)

NOTE – Much of the documentation required in this clause is properly the responsibility of programming language or binding standards. An individual implementation would only need to provide details if it could not cite an appropriate clause of the language or binding standard.

Annex A
(informative)

Partial conformity

The requirements of this part of ISO/IEC 10967 have been carefully chosen to be as beneficial as possible, yet be efficiently implemented on most existing or anticipated hardware architectures.

The bulk of requirements of this part of ISO/IEC 10967 are for documentation, or for parameters and functions that can be efficiently realized in software. However, the accuracy and notification requirements on the four basic floating point operations ($add_{F \to F'}$, $sub_{F \to F'}$, $mul_{F \to F'}$, and $div_{F \to F'}$) do have implications for the underlying hardware architecture.

A small number of computer systems may have some difficulties with some of the requirements of this part of ISO/IEC 10967 for floating point arithmetic:

a) The ability to represent subnormal values apart from zero.

b) The ability to record all notifications, including **underflow** and **inexact**.

c) The ability to distinguish **infinitary** and **invalid** notifications.

d) Strict 0.5-ulp accuracy for $add_{F \to F'}$, $sub_{F \to F'}$, $mul_{F \to F'}$, and $div_{F \to F'}$ (strictly interpreted this applies also for continuation values upon **underflow** which in turn requires subnormal values).

e) Strict 0.5-ulp accuracy for $convert_{F \to F'}$, $convert_{F \to D}$, and $convert_{D \to F}$.

f) $add_{F \to F}$ and $sub_{F \to F}$ never give a notification of **underflow**, but instead may return a subnormal value.

g) Round ties to even last digit.

h) A common rounding rule for $add_{F \to F'}$, $sub_{F \to F'}$, $mul_{F \to F'}$, and $div_{F \to F'}$.

i) The ability to do exact comparisons without spurious notifications.

j) A sign symmetric value set (all values can be negated exactly).

Language standards will want to adopt all the requirements of this part of ISO/IEC 10967 to provide programmers with the maximum benefit. However, if it is perceived that requiring full conformity to this part of ISO/IEC 10967 will exclude a significant portion of that language's user community from any benefit, then specifying partial conformity to this part of ISO/IEC 10967, as permitted in clause 2 and further specified in this Annex, may be a reasonable alternative.

Such partial conformity would relax one or more of the requirements listed above, but would retain the benefits of all other requirements of this part of ISO/IEC 10967. All deviations from conformity to this part of ISO/IEC 10967 shall be fully documented.

If a programming language (or binding) standard states that partial conformity is permitted, programmers will need to detect what degree of conformity is available. It would be helpful for the language standard to require parameters indicating whether or not conformity is complete, and if not, which of the requirements above are violated.

A.1 Integer overflow notification relaxation

Some programming languages specify a “wrapping” interpretation of addition, subtraction, and multiplication for bounded integer datatypes. These are in ISO/IEC 10967 modelled as different operations from the add_I, sub_I, and mul_I operations, and are specified in Part 2.

If a binding allows an implementation to interpret the ordinary (for that language) programming language syntax for integer addition, subtraction, and multiplication as wrapping operations, there shall be a Boolean parameter, available to programs:

$modulo_I$ — if **true** this indicates that the implementation uses the operations add_wrap_I, sub_wrap_I, and mul_wrap_I specified in Part 2 instead of add_I, sub_I, and mul_I specified in this part of ISO/IEC 10967 for the “ordinary” syntax for these operations.

NOTES

1 In the first edition of this document, this was modelled as a normative parameter and a change of interpretation of the add_I, sub_I, mul_I, div^f_I, and div^t_I operations. In this edition the parameter and the wrapping interpretation for the *ordinary* (in that programming language) addition, subtraction, and multiplication operations are accepted as only partially conforming.

2 The interpretation of integer division has been made stricter in this edition than in the first edition, and is no longer dependent on the the $modulo_I$ parameter even if integer overflow notification is otherwise relaxed.

3 add_wrap_I, sub_wrap_I, and mul_wrap_I can (and should) be provided as separate operations also in fully conforming implementations.

4 add_I, sub_I, and mul_I can (and should) be provided as separate operations (though perhaps will less appealing syntax) also in partially conforming implementations where integer overflow notification is relaxed for the ordinary syntax operations.

A.2 Infinitary notification relaxation

With an **infinitary** notification (as opposed to **overflow**) a continuation value that is an infinity is given as an exact value. It is therefore reasonable to have implementations or modes that suppress **infinitary** notifications.

If a binding allows infinitary notifications to go unrecorded, there shall be a Boolean parameter, available to programs:

$silent\_infinitary_I$ — **true** when infinitary notifications are suppressed, and not replaced by invalid notifications, for the integer datatype corresponding to I.

$invalid\_infinitary_I$ — **true** when infinitary notifications are replaced by invalid notifications for the integer datatype corresponding to I.

$silent\_infinitary_F$ — **true** when infinitary notifications are suppressed for the floating point datatype corresponding to F.

A.3 Inexact notification relaxation

Some architectures may not be able to efficiently detect inexact results.

If a binding allows inexact to go unrecorded, there shall be a Boolean parameter, available to programs:

$silent\_inexact_F$ — **true** when inexact notifications are suppressed for the datatype corresponding to F.

A.4 Underflow notification relaxation

Some architectures may not be able to efficiently detect underflow, even if there are no non-zero subnormal values.

If a binding allows underflow to go unrecorded, there shall be a Boolean parameter, available to programs:

$silent\_underflow_F$ — **true** when underflow notifications are suppressed for the datatype corresponding to F.

A.5 Subnormal values relaxation

If the parameter $denorm_F$ has a value **false**, and thus there are no subnormal values in F except 0 (and $\mathbf{-0}$, if available, in the corresponding datatype), then the corresponding datatype is not fully conforming to this part of ISO/IEC 10967. If a binding allows a floating point datatype in an implementation not to have subnormal values apart from 0 and $\mathbf{-0}$, the parameter $denorm_F$ shall be made available to programs, and the parameter $fminD_F$ shall not be available if $denorm_F$ has the value **false**. Further, how rounding is done in the interval $]-fminN_F, fminN_F[$ shall be documented for the case that $denorm_F$ has the value **false** (often "flush to zero" is used).

NOTE – If full conformity is required by the binding, the parameter $denorm_F$ is always **true** and need not be made available to programs.

A.6 Accuracy relaxation for add, subtract, multiply, and divide

Ideally, and conceptually, no information should be lost before the rounding step in the computational model of this part of ISO/IEC 10967. But some hardware implementations of arithmetic operations compute an approximation that loses information (conceptually) prior to rounding (to nearest). In some cases, it may even be so that $x + y = u + v$ may not imply $add'_{F\to F'}(x, y) = add'_{F\to F'}(u, v)$ (and similarly for subtract). ($add'_{F\to F'}$ is defined below.)

A maximum error parameter max_error is an element in F such that

$$|x - nearest_F(x')| \leqslant max\_error \cdot u_F(x)$$

for $x, x' \in \mathcal{R}$, where x is the mathematical result and x' is the result of a corresponding approximation helper function given the same arguments; max_error should be the minimal value for which the relation holds.

If this relaxation is allowed, there shall be a maximum error parameter, rnd_error_F parameter available to programs for the datatype corresponding to F. rnd_error_F shall be the maximum error for the addition, multiplication, and division operations. It cannot have a value that is less than 0.5 and shall have a value less than or equal to 1.

The add^*_F, mul^*_F, and div^*_F helper functions are introduced to model this pre-rounding approximation: $add^*_F : F^\ddagger \times F^\ddagger \to \mathcal{R}$, $mul^*_F : F^\ddagger \times F^\ddagger \to \mathcal{R}$, $div^*_F : F^\ddagger \times F^\ddagger \to \mathcal{R}$.

$add^*_F(x, y)$ returns a close approximation to $x + y$, satisfying
$$|(x + y) - nearest_F(add^*_F(x, y))| \leqslant rnd\_error_F \cdot u_F(x + y)$$
$mul^*_F(x, y)$ returns a close approximation to $x \cdot y$, satisfying
$$|(x \cdot y) - nearest_F(mul^*_F(x, y))| \leqslant rnd\_error_F \cdot u_F(x \cdot y)$$
$div^*_F(x, y)$ returns a close approximation to x/y, satisfying
$$|(x/y) - nearest_F(div^*_F(x, y))| \leqslant rnd\_error_F \cdot u_F(x/y)$$

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

$u + v \in F^\ddagger \;\Leftrightarrow\; add^*_F(u, v) = u + v$	if $u, v \in F^\ddagger$
$add^*_F(-u, -v) = -add^*_F(u, v)$	if $u, v \in F^\ddagger$
$add^*_F(u, v) = add^*_F(v, u)$	if $u, v \in F^\ddagger$
$add^*_F(u, x) \leqslant add^*_F(v, x)$	if $u, v, x \in F^\ddagger$ and $u < v$
$add^*_F(u \cdot r^i_F, v \cdot r^i_F) = add^*_F(u, v) \cdot r^i_F$	if $u, v \in F^\ddagger$ and $i \in \mathcal{Z}$

NOTES

1 The above requirements capture the following properties:

a) The result of add^*_F is exact iff the ‘true result’ is in $F^\ddagger$; and, by monotonicity, $add^*_F(u, v)$ and $u+v$, when not equal, are in the same range of values strictly between two adjacent values in $F^\ddagger$.

b) add^*_F is sign symmetric when changing both signs.

c) add^*_F is commutative.

d) add^*_F is monotonic for the left operand and, by commutativity, for the right operand.

e) For add^*_F a common exponent of its arguments can be factored out. But the approximation to +, and thus final rounding, may depend on the (absolute) difference of the exponents of the arguments.

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

$u \cdot v \in F^\ddagger \;\Leftrightarrow\; mul^*_F(u, v) = u \cdot v$	if $u, v \in F^\ddagger$
$mul^*_F(-u, v) = -mul^*_F(u, v)$	if $u, v \in F^\ddagger$
$mul^*_F(u, v) = mul^*_F(v, u)$	if $u, v \in F^\ddagger$
$mul^*_F(u, x) \leqslant mul^*_F(v, x)$	if $u, v, x \in F^\ddagger$ and $u < v$ and $0 < x$
$mul^*_F(u, x) \geqslant mul^*_F(v, x)$	if $u, v, x \in F^\ddagger$ and $u < v$ and $x < 0$
$mul^*_F(u \cdot r^i_F, v \cdot r^j_F) = mul^*_F(u, v) \cdot r^{i+j}_F$	if $u, v \in F^\ddagger$ and $i, j \in \mathcal{Z}$

NOTES

2 The above requirements capture the following properties:

a) The result of mul^*_F is exact iff the ‘true result’ is in $F^\ddagger$; and, by monotonicity, $mul^*_F(u, v)$ and $u \cdot v$, when not equal, are in the same range of values strictly between two adjacent values in $F^\ddagger$. Thus, if rnd_error_F is 1, the max error is strictly less than 1.

b) mul^*_F is sign symmetric for the left operand and, by commutativity, for the right operand.

c) mul^*_F is commutative.

d) mul^*_F is monotonic for the left operand and, by commutativity, for the right operand.

e) For mul^*_F the exponents of its arguments can be factored out.

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

$u/v \in F^\ddagger \;\Leftrightarrow\; div^*_F(u, v) = u/v$	if $u, v \in F^\ddagger$ and $v \neq 0$
$div^*_F(-u, v) = -div^*_F(u, v)$	if $u, v \in F^\ddagger$ and $v \neq 0$
$div^*_F(u, -v) = -div^*_F(u, v)$	if $u, v \in F^\ddagger$ and $v \neq 0$

$$\begin{array}{ll}
div^*_F(u,x) \leqslant div^*_F(v,x) & \text{if } u,v,x \in F^\ddagger \text{ and } u < v \text{ and } 0 < x \\
div^*_F(u,x) \geqslant div^*_F(v,x) & \text{if } u,v,x \in F^\ddagger \text{ and } u < v \text{ and } x < 0 \\
div^*_F(x,u) \geqslant div^*_F(x,v) & \text{if } u,v,x \in F^\ddagger \text{ and } (u < v < 0 \text{ or } 0 < u < v) \text{ and } 0 < x \\
div^*_F(x,u) \leqslant div^*_F(x,v) & \text{if } u,v,x \in F^\ddagger \text{ and } (u < v < 0 \text{ or } 0 < u < v) \text{ and } x < 0 \\
div^*_F(u \cdot r_F^i, v \cdot r_F^j) = div^*_F(u,v) \cdot r_F^{i-j} & \text{if } u,v \in F^\ddagger \text{ and } v \neq 0 \text{ and } i,j \in \mathcal{Z}
\end{array}$$

NOTES

3 The above requirements capture the following properties:

a) The result of div^*_F is exact iff the 'true result' is in $F^\ddagger$; and, by monotonicity, $div^*_F(u,v)$ and u/v, when not equal, are in the same range of values strictly between two adjacent values in $F^\ddagger$.

b) div^*_F is sign symmetric for the left operand and for the right operand.

c) div^*_F is monotonic for the left and right operands.

d) For div^*_F the exponents of its arguments can be factored out.

If this relaxation is permitted in a binding, $add'_{F\to F'}$, $sub'_{F\to F'}$, $mul'_{F\to F'}$, and $div'_{F\to F'}$, (replacing $add_{F\to F'}$, $sub_{F\to F'}$, $mul_{F\to F'}$, and $div_{F\to F'}$ in the binding) shall be defined as:

$$\begin{array}{lll}
add'_{F\to F'}(x,y) & = result_{F'}(add^*_{F'}(x,y), nearest_{F'}) & \text{if } x,y \in F \\
 & = add_{F\to F'}(x,y) & \text{otherwise}
\end{array}$$

$$sub'_{F\to F'}(x,y) = add'_{F\to F'}(x, neg_F(y))$$

$$\begin{array}{lll}
mul'_{F\to F'}(x,y) & = result_{F'}(mul^*_{F'}(x,y), nearest_{F'}) & \text{if } x,y \in F \\
 & = mul_{F\to F'}(x,y) & \text{otherwise}
\end{array}$$

$$\begin{array}{lll}
div'_{F\to F'}(x,y) & = result_{F'}(div^*_{F'}(x,y), nearest_{F'}) & \text{if } x,y \in F \text{ and } y \neq 0 \\
 & = div_{F\to F'}(x,y) & \text{otherwise}
\end{array}$$

This allows addition, subtraction, multiplication, and division that do not round ties to even last digit (when $rnd\_error_{F'}$ is 0.5), or rounds towards zero or away from zero (when $rnd\_error_{F'}$ is 1).

If this relaxation is allowed, there shall be a parameter rnd_style_F, available to programs for the datatype corresponding to F, having one of four constant values, which is defined by

$$\begin{array}{lll}
rnd\_style_F & = \textbf{nearesttiestoeven} & \text{if this relaxation is not engaged} \\
 & = \textbf{nearest} & \text{if this relaxation is engaged and } rnd\_error_F = 0.5 \\
 & = \textbf{truncate} & \text{if } nearest_F(|add^*_F(x,y)|) = down_F(|x+y|) \text{ and} \\
 & & \quad nearest_F(|mul^*_F(x,y)|) = down_F(|x \cdot y|) \text{ and} \\
 & & \quad nearest_F(|div^*_F(x,y)|) = down_F(|x/y|) \\
 & = \textbf{other} & \text{otherwise}
\end{array}$$

If $rnd\_style_F \neq$ **nearesttiestoeven**, the operations using up_F or $down_F$ as rounding shall *not* be provided.

A.7 Accuracy relaxation for floating point conversion

First define the least radix function, lb, defined for arguments that are greater than 0:

$$lb : \mathcal{Z} \to \mathcal{Z}$$

$$lb(r) = \min\{n \in \mathcal{Z} \mid n \geqslant 1 \text{ and there is an } m \in \mathcal{Z} \text{ such that } r = n^m\}$$

If this relaxation is allowed, there shall be a $max\_error\_convert_{F \to F'}$ parameter that gives the maximum error when converting from F to F' and $lb(r_F) \neq lb(r_{F'})$, a $max\_error\_convert_{F \to D}$ parameter that gives the maximum error when converting from F to D and $lb(r_F) \neq lb(r_D)$, and a $max\_error\_convert_{D \to F}$ parameter that gives the maximum error when converting from D to F and $lb(r_D) \neq lb(r_F)$. These parameters may be required to have the same value, and then only one parameter need be made available to programs.

$max\_error\_convert_{F \to F'}$, $max\_error\_convert_{F \to D}$, and $max\_error\_convert_{D \to F}$ should each have a value less than 1. When $lb(r_F) = lb(r_{F'})$, then $max\_error\_convert_{F \to F'}$, $max\_error\_convert_{F \to D}$, and $max\_error\_convert_{D \to F}$ shall all be 0.5.

The $convert^*_{F \to F'}$, $convert^*_{F \to D}$, and $convert^*_{D \to F}$ helper functions are introduced to model this pre-rounding approximation: $convert^*_{F \to F'} : F^\ddagger \to \mathcal{R}$, $convert^*_{F \to D} : F^\ddagger \to \mathcal{R}$, $convert^*_{D \to F} : D^* \to \mathcal{R}$.

$convert^*_{F \to F'}(x)$ returns a close approximation to x, satisfying
$$|x - nearest_F(convert^*_{F \to F'}(x))| \leqslant max\_error\_convert_{F \to F'} \cdot u_F(x)$$
$convert^*_{F \to D}(x)$ returns a close approximation to x, satisfying
$$|x - nearest_F(convert^*_{F \to D}(x))| \leqslant max\_error\_convert_{F \to D} \cdot u_F(x)$$
$convert^*_{D \to F}(x)$ returns a close approximation to x, satisfying
$$|x - nearest_F(convert^*_{D \to F}(x))| \leqslant max\_error\_convert_{D \to F} \cdot u_F(x)$$

Further requirements on the $convert^*_{F \to F'}$ approximation helper functions are:

$$\begin{array}{ll} convert^*_{F \to F'}(x) = x & \text{if } x \in \mathcal{Z} \cap F \\ convert^*_{F \to F'}(-x) = -convert^*_{F \to F'}(x) & \text{if } x \in F^\ddagger \\ convert^*_{F \to F'}(x) \leqslant convert^*_{F \to F'}(y) & \text{if } x, y \in F^\ddagger \text{ and } x < y \end{array}$$

Relationship to other floating point to floating point conversion approximation helper functions for conversion operations in the same library shall be:

$$convert^*_{F \to F'}(x) = convert^*_{F'' \to F'}(x) \qquad \text{if } lb(r_{F''}) = lb(r_F) \text{ and } x \in F \cap F''$$

The $convert'_{F \to F'}$ operation:

$$convert'_{F \to F'} : F \to F' \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$$

$$\begin{array}{ll} convert'_{F \to F'}(x) & = result_{F'}(convert^*_{F \to F'}(x), nearest_{F'}) \\ & \qquad \text{if } x \in F \\ & = convert_{F \to F'}(x) \qquad \text{otherwise} \end{array}$$

Further requirements on the $convert^*_{F \to D}$ approximation helper functions are:

$$\begin{array}{ll} convert^*_{F \to D}(x) = x & \text{if } x \in \mathcal{Z} \cap F \\ convert^*_{F \to D}(-x) = -convert^*_{F \to D}(x) & \text{if } x \in F \\ convert^*_{F \to D}(x) \leqslant convert^*_{F \to D}(y) & \text{if } x, y \in F \text{ and } x < y \end{array}$$

Relationship to other floating point to fixed point conversion approximation helper functions for conversion operations in the same library shall be:

$$convert^*_{F \to D}(x) = convert^*_{F'' \to D}(x) \qquad \text{if } lb(r_{F''}) = lb(r_F) \text{ and } x \in F \cap F''$$

The $convert'_{F \to D}$ operation:

$$convert'_{F \to D} : F \to D \cup \{\mathbf{inexact}, \mathbf{overflow}\}$$

$$\begin{array}{ll} convert'_{F \to D}(x) & = result_D(convert^*_{F \to D}(x), nearest_D) \\ & \qquad \text{if } x \in F \\ & = convert_{F \to F'}(x) \qquad \text{otherwise} \end{array}$$

Further requirements on the $convert^*_{D\to F}$ approximation helper functions are:

$$
\begin{array}{ll}
convert^*_{D\to F}(x) = x & \text{if } x \in \mathcal{Z} \cap D \\
convert^*_{D\to F}(-x) = -convert^*_{D\to F}(x) & \text{if } x \in D \\
convert^*_{D\to F}(x) \leqslant convert^*_{D\to F}(y) & \text{if } x, y \in D \text{ and } x < y
\end{array}
$$

Relationship to other floating point and fixed point to floating point conversion approximation helper functions for conversion operations in the same library shall be:

$$
\begin{array}{ll}
convert^*_{D\to F}(x) = convert^*_{D'\to F}(x) & \text{if } lb(r_{D'}) = lb(r_D) \text{ and } x \in D \cap D' \\
convert^*_{D\to F}(x) = convert^*_{F'\to F}(x) & \text{if } lb(r_{F'}) = lb(r_D) \text{ and } x \in D \cap F'
\end{array}
$$

The $convert'_{D\to F}$ operation:

$$convert'_{D\to F} : D \to F \cup \{\mathbf{inexact}, \mathbf{underflow}, \mathbf{overflow}\}$$

$$
\begin{array}{lll}
convert'_{D\to F}(x) & = result_F(convert^*_{D\to F}(x), nearest_F) & \\
& & \text{if } x \in D \\
& = convert_{D\to F}(x) & \text{otherwise}
\end{array}
$$

Annex B
(informative)

IEC 60559 bindings

When the parameter iec_60559_F is **true** for a floating point type F, all the facilities required by IEC 60559 shall be provided for that datatype. Methods shall be provided for a program to access each such facility. In addition, documentation shall be provided to describe these methods.

This means that a *complete* programming language binding for LIA-1 should provide a binding for all IEC 60559 facilities as well. Such a programming language binding must define syntax for all required facilities, and should define syntax for all optional facilities as well. Defining syntax for optional facilities does not make those facilities required. All it does is ensure that those implementations that choose to provide an optional facility will do so using a standardized syntax.

The normative listing of all IEC 60559 facilities (and their definitions) is given in IEC 60559. ISO/IEC 10967 does not alter or eliminate any of them. However, to assist the reader, the following summary is offered.

B.1 Summary

A binding of IEC 60559 to a programming language must provide the names of the programming language datatypes that correspond to:

a) binary32,

b) binary64,

c) binary128,

d) decimal64,

e) decimal128,

if any.

Note that the LIA-1 parameter values for each of the IEC 60559 datatypes the parameters $denorm_F$ and iec_60559_F are **true**. The remaining LIA-1 basic parameters for 'binary32' are:

$r_F = 2$
$p_F = 24$
$emin_F = -125$
$emax_F = 128$

For IEC 60559 'binary64' they are:

$r_F = 2$
$p_F = 53$
$emin_F = -1021$
$emax_F = 1024$

For IEC 60559 'binary128' they are:

$r_F = 2$
$p_F = 113$
$emin_F = -16381$
$emax_F = 16384$

For IEC 60559 'decimal64' they are:

$r_F = 10$
$p_F = 16$
$emin_F = -382$
$emax_F = 385$

For IEC 60559 'decimal128' they are:

$r_F = 10$
$p_F = 34$
$emin_F = -6142$
$emax_F = 6145$

IEC 60559 also specifies 'binary16' and 'decimal32' with only conversion operations specified, terming them "storage formats". These storage formats are *not* included when referring to IEC 60559 conforming datatype below. IEC 60559 also specifies *extended* formats, giving just maximum or minimum requirements on the parameters. These *are* included when referring to IEC 60559 conforming datatype below.

For each IEC 60559 conforming datatype, the binding must provide:

a) a method for denoting positive infinity,

b) a method for denoting at least one quiet NaN (not-a-number),

c) a method for denoting at least one signalling NaN (not-a-number).

For each IEC 60559 conforming datatype provided, the binding should provide the notation for invoking each of the following operations.

a) Homogeneous general-computational operations.

sourceFormat roundToIntegralTiesToEven(source)	$rounding_F$
sourceFormat roundToIntegralTiesToAway(source)	
sourceFormat roundToIntegralTowardZero(source)	
sourceFormat roundToIntegralTowardPositive(source)	$ceiling_F$
sourceFormat roundToIntegralTowardNegative(source)	$floor_F$
sourceFormat roundToIntegralExact(source)	
sourceFormat nextUp(source)	$succ_F$
sourceFormat nextDown(source)	$pred_F$
sourceFormat remainder(source, source)	$residue_F$
sourceFormat minNum(source, source)	$mmin_F$
sourceFormat maxNum(source, source)	$mmax_F$
sourceFormat minNumMag(source, source)	$mmin_F(abs_F(x), abs_F(y))$
sourceFormat maxNumMag(source, source)	$mmax_F(abs_F(x), abs_F(y))$
sourceFormat quantize(source, source)	
sourceFormat scaleB(source, logBFormat)	$scale_{F,I}$
logBFormat logB(source)	$exponent_{F \rightarrow I}(x) - 1$

b) formatOf general-computational operations. The basic arithmetic operations are also required by LIA-1, though not with built-in conversion of the arguments are of different type, and in LIA-1 not permitted to be rounding mode dependent, changing the rounding mode may change the implementation to be in a state not conforming to LIA-1.

formatOf-addition(source1, source2)	$add_{F\to F'}$, $add^{\uparrow}_{F\to F'}$, $add^{\downarrow}_{F\to F'}$
formatOf-subtraction(source1, source2)	$sub_{F\to F'}$, $sub^{\uparrow}_{F\to F'}$, $sub^{\downarrow}_{F\to F'}$
formatOf-multiplication(source1, source2)	$mul_{F\to F'}$, $mul^{\uparrow}_{F\to F'}$, $mul^{\downarrow}_{F\to F'}$
formatOf-division(source1, source2)	$div_{F\to F'}$, $div^{\uparrow}_{F\to F'}$, $div^{\downarrow}_{F\to F'}$
formatOf-squareRoot(source)	$sqrt_{F\to F'}$, $sqrt^{\uparrow}_{F\to F'}$, $sqrt^{\downarrow}_{F\to F'}$
formatOf-fusedMultiplyAdd(source1, source2, source3)	$mul\_add_{F\to F'}$, $mul\_add^{\uparrow}_{F\to F'}$, $mul\_add^{\downarrow}_{F\to F'}$
formatOf-convertFromInt(int)	$convert_{I\to F}$, $convert^{\uparrow}_{I\to F}$, $convert^{\downarrow}_{I\to F}$
intFormatOf-convertToIntegerTiesToEven(source)	$rounding_{F\to I}$
intFormatOf-convertToIntegerTowardZero(source)	
intFormatOf-convertToIntegerTowardPositive(source)	$ceiling_{F\to I}$
intFormatOf-convertToIntegerTowardNegative(source)	$floor_{F\to I}$
intFormatOf-convertToIntegerTiesToAway(source)	
intFormatOf-convertToIntegerExactTiesToEven(source)	
intFormatOf-convertToIntegerExactTowardZero(source)	
intFormatOf-convertToIntegerExactTowardPositive(source)	
intFormatOf-convertToIntegerExactTowardNegative(source)	
intFormatOf-convertToIntegerExactTiesToAway(source)	
formatOf-convertFormat(source)	$convert_{F\to F'}$, $convert^{\uparrow}_{F\to F'}$, $convert^{\downarrow}_{F\to F'}$
formatOf-convertFromDecimalCharacter(decimalCharacterSequence)	
$r_{F'} = 10$ and $r_D = 10$	
	$convert_{F'\to F}$, $convert_{D\to F}$
	$convert^{\uparrow}_{F'\to F}$, $convert^{\uparrow}_{D\to F}$
	$convert^{\downarrow}_{F'\to F}$, $convert^{\downarrow}_{D\to F}$
decimalCharacterSequence convertToDecimalCharacter(source, conversionSpecification)	
$r_{F'} = 10$ and $r_D = 10$	
	$convert_{F\to F'}$, $convert_{F\to D}$
	$convert^{\uparrow}_{F\to F'}$, $convert^{\uparrow}_{F\to D}$
	$convert^{\downarrow}_{F\to F'}$, $convert^{\downarrow}_{F\to D}$
formatOf-convertFromHexCharacter(hexCharacterSequence)	
$r_{F'} = 16$, $r_D = 16$, and $r_F = 2$	
	$convert_{F'\to F}$, $convert_{D\to F}$
	$convert^{\uparrow}_{F'\to F}$, $convert^{\uparrow}_{D\to F}$
	$convert^{\downarrow}_{F'\to F}$, $convert^{\downarrow}_{D\to F}$
hexCharacterSequence convertToHexCharacter(source, conversionSpecification)	
$r_{F'} = 16$, $r_D = 16$, and $r_F = 2$	

$convert_{F \to F'}$, $convert_{F \to D}$
$convert^{\uparrow}_{F \to F'}$, $convert^{\uparrow}_{F \to D}$
$convert^{\downarrow}_{F \to F'}$, $convert^{\downarrow}_{F \to D}$

NOTE – mul_add_F ($mul\_add_{F \to F'}$) is specified in LIA-2 (part 2 of ISO/IEC 10967).

c) Quiet-computational operations.

sourceFormat copy(source)	$convert_{F \to F}$, but non-signalling
sourceFormat negate(source)	similar to neg_F, but non-signalling
sourceFormat abs(source)	similar to abs_F, but non-signalling

decimalEncoding encodeDecimal(decimal)
decimal decodeDecimal(decimalEncoding)
binaryEncoding encodeBinary(decimal)
decimal decodeBinary(binaryEncoding)

d) Signaling-computational operations. The basic comparisons are also required by LIA-1, though not with built-in conversion if the arguments are of different type.

boolean compareQuietEqual(source1,source2)	eq_F
boolean compareQuietNotEqual(source1,source2)	neq_F
boolean compareSignallingGreater(source1,source2)	gtr_F
boolean compareSignallingGreaterEqual(source1,source2)	geq_F
boolean compareSignallingLess(source1,source2)	lss_F
boolean compareSignallingLessEqual(source1,source2)	leq_F
boolean compareSignalingEqual(source1, source2)	
boolean compareSignalingNotEqual(source1, source2)	
boolean compareSignalingNotGreater(source1,source2)	
boolean compareSignalingLessUnordered(source1,source2)	
boolean compareSignalingNotLess(source1,source2)	
boolean compareSignalingGreaterUnordered(source1,source2)	
boolean compareQuietGreater(source1,source2)	
boolean compareQuietGreaterEqual(source1,source2)	
boolean compareQuietLess(source1,source2)	
boolean compareQuietLessEqual(source1,source2)	
boolean compareQuietUnordered(source1,source2)	
boolean compareQuietNotGreater(source1,source2)	
boolean compareQuietLessUnordered(source1,source2)	
boolean compareQuietNotLess(source1,source2)	
boolean compareQuietGreaterUnordered(source1,source2)	
boolean compareQuietOrdered(source1,source2)	

e) Non-computational operations.

boolean is754version1985(void)	
boolean is754version2008(void)	
enum class(source)	
boolean isSignMinus(source)	similar to $eq_F(signum_F(x), -1)$

boolean isNormal(source)	similar to $geq_F(abs_F(x), fminN_F)$
boolean isFinite(source)	similar to $lss_F(abs_F(x), +\infty)$
boolean isZero(source)	$eq_F(x, 0)$
boolean isSubnormal(source)	similar to $istiny_F(x)$ && $neq_F(x, 0)$
boolean isInfinite(source)	$eq_F(abs_F(x), +\infty)$
boolean isNaN(source)	$isnan_F(x)$
boolean isSignaling(source)	$issignan_F(x)$
boolean isCanonical(source)	
enum radix(source)	r_F
boolean totalOrder(source, source)	
boolean totalOrderMag(source, source)	
boolean sameQuantum(source, source)	

f) Handling of notification recordings.

void lowerFlags(exceptionGroup)	*clear_indicators*
void raiseFlags(exceptionGroup)	*set_indicators*
boolean testFlags(exceptionGroup)	*test_indicators*
boolean testSavedFlags(flags, exceptionGroup)	
void restoreFlags(flags, exceptionGroup)	
flags saveAllFlags(void)	*current_indicators*

Note that several of the above facilities are already required by LIA-1 even for implementations that do not conform to IEC 60559.

IEC 60559 allows for dynamic change of rounding mode. Note though that LIA requires that the rounding is static for an operation (which may be simulated at lower level by setting and resetting a dynamic rounding mode, provided that the rounding mode is local to each thread). Thus explicitly using rounding mode setting at a “higher level” may change an LIA conforming implementation to (a mode in which it is) a non-conforming implementation.

IEC 60559 recommend a number of other operations (some of which are covered by LIA-2), but these are not listed here. See IEC 60559 for details.

B.2 Notification

The default notification handling in IEC 60559 is by recording in indicators, called exception flags in IEC 60559. One appropriate way to access the IEC 60559 exception flags is to use the functions defined in 6.2.1.

IEC 60559 also allows for alternate exception handling. In LIA this is modelled by notification by change of control.

Annex C
(informative)

Rationale

This annex explains and clarifies some of the ideas behind *Information technology – Language independent arithmetic – Part 1: Integer and floating point arithmetic* (LIA-1). This allows the standard itself to be more concise. Many of the major requirements are discussed in detail, including the merits of possible alternatives. The clause numbering matches that of the standard, although additional clauses have been added.

C.1 Scope

The scope of LIA-1 includes the traditional primitive arithmetic operations usually provided in hardware. The standard also includes several other useful primitive operations which could be provided in hardware or software. An important aspect of all of these primitive operations is that they are to be considered *atomic* (with respect to rounding and notifications, *not* necessarily with respect to thread parallelism, though they may not interfere with the results of LIA operations in other threads provided that arguments and results are thread separate) rather than noticeably implemented as a sequence of yet more primitive operations. Hence, each primitive floating point operation has a half *ulp* error bound when rounding to nearest, and is never interrupted by an intermediate notification. The latter is true also for all integer operations.

LIA-1 provides a parameterised model for arithmetic. Such a model is needed to make concepts such as “precision” or “exponent range” meaningful. However, there is no such thing as an “LIA-1 machine”. It makes no sense to write code intended to run on all machines describable with LIA-1 model – the model covers too wide a range for that. It does make sense to write code that uses LIA-1 facilities to determine whether the platform it is running on is suitable for its needs.

C.1.1 Inclusions

This standard is intended to define the meaning of an “integer datatype” and a “floating point datatype”, but not to preclude other arithmetic or related datatypes. The specifications for integer and floating point datatypes are given in sufficient detail to

a) support detailed and accurate numerical analysis of arithmetic algorithms,

b) serve as the first of a family of standards, as outlined in C.1.3,

c) enable a precise determination of conformity or non-conformity, and

d) prevent exceptions (like overflow) from going undetected.

C.1.2 Exclusions

There are many arithmetic systems, such as fixed point arithmetic, significance arithmetic, interval arithmetic, rational arithmetic, level-index arithmetic, slash arithmetic, and so on, which differ considerably from traditional integer and floating point arithmetic, as well as among themselves.

Some of these systems, like fixed point arithmetic, are in wide-spread use as datatypes in standard languages; most are not. A form of floating point is defined by Kulisch and Miranker [53, 54] which is compatible with the floating point model in LIA-1. For reasons of simplicity and clarity, these alternate arithmetic systems are not treated in LIA-1. They should be the subject of other parts of ISO/IEC 10967 if and when they become candidates for standardization.

The portability goal of LIA-1 is for programs, rather than data. LIA-1 does not specify the internal representation of data. However, portability of data is a subject of IEC 60559, which specifies internal representations that can also be used for data exchange.

Mixed mode operations in general, and other issues of expression semantics, are not addressed directly by LIA-1. However, suitable documentation is required (see clause 7).

C.1.3 Companion parts to this part

The following topics are the subject of a family of standard parts, of which LIA-1 is the first member:

a) Specifications for the usual elementary functions (LIA-2).

b) Specifications for complex and imaginary datatypes and operations (LIA-3).

This list is incomplete, and further parts may be created.

Each of these new sets of specifications is necessary to provide a *total numerical environment* for the support of portable robust numerical software. The properties of the primitive operations is used in the specifications of elementary and complex functions which

a) are realistic from an implementation point of view,

b) have acceptable performance, and

c) have adequate accuracy to support numerical analysis.

C.2 Conformity

A conforming system consists of an implementation (which obeys the requirements) together with documentation which shows how the implementation conforms to the standard. This documentation is vital since it gives crucial characteristics of the system, such as the range for integers, the range and precision for floating point, and the actions taken by the system on the occurrence of notifications.

The binding of LIA-1 facilities to a particular programming language should be as natural as possible. Existing language syntax and features should be used for operations, parameters, notification, and so on. For example, if a language expresses addition by “`x+y`,” then LIA-1 addition operations add_I and $add_{F \to F'}$ should be bound to the infix “`+`” operator.

Most current implementations of floating point can be expected to conform to the specifications in this standard. In particular, implementations of IEC 60559 (IEEE 754 [34]) in default mode will conform, provided that the user is made aware of any status flags that remain set upon exit from a program.

The documentation required by LIA-1 will highlight the differences between “almost IEEE” systems and fully IEEE conforming ones.

Note that a system can claim conformity for a single integer type, a single floating point type, or a collection of arithmetic types.

An implementation is free to provide arithmetic datatypes (e.g. fixed point) or arithmetic operations (e.g. exponentiation on integers) which may be required by a language standard but are not specified by LIA-1. Similarly, an implementation may have modes of operation (e.g. notifications disabled) that do not conform to LIA-1. The implementation must not claim conformity to LIA-1 for these arithmetic datatypes or modes of operation. Again, the documentation that distinguishes between conformity and non-conformity is critical. An example conformity statement (for a Fortran implementation) is given in Annex E.

C.2.1 Validation

This standard gives a very precise description of the properties of integer and floating point datatypes. This will expedite the construction of conformity tests. It is important that objective tests be available. Schryer [56] has shown that such testing is needed for floating point since two thirds of units tested by him contained serious design flaws. Another test suite is available for floating point [43], which includes enhancements based upon experience with Schryer's work [56].

LIA-1 does not define any process for validating conformity.

Independent assurance of conformity to LIA-1 could be by spot checks on products with a validation suite, as for language standards, or via vendors being registered under ISO/IEC 9001 *Model for quality assurance in production and installation* [31] enhanced with the requirement that their products claiming conformity are tested with the validation suite and checked to conform as part of the release process.

Alternatively, checking could be regarded as the responsibility of the vendor, who would then document the evidence supporting any claim to conformity.

C.3 Normative references

The referenced IEC 60559 standard is identical to IEEE 754. IEEE 754-2008 is a major revision of IEEE 754-1985, adding 124-bit floating point types, decimal radix floating point types (formerly in IEEE 854, now withdrawn), removing "trapping" exception handling, removing implementor options, adding operations, and more. IEC 60559 third edition (2011) will be identical to IEEE 754-2008.

C.4 Symbols and definitions

An arithmetic standard must be understood by numerous people with different backgrounds: numerical analysts, compiler-writers, programmers, microcoders, and hardware designers. This raises certain practical difficulties. If the standard were written entirely in a natural language, it might contain ambiguities. If it were written entirely in mathematical terms, it might be inaccessible to some readers. These problems were resolved by using mathematical notation for LIA-1, and providing this rationale in English to explain the notation.

C.4.1 Symbols

LIA-1 uses the conventional notation for sets and operations on sets. The set $\mathcal{Z}$ denotes the set of mathematical integers. This set is infinite, unlike the finite subset which a machine can conveniently handle. The set of real numbers is denoted by $\mathcal{R}$, which is also infinite. Numbers such as π, $1/3$ and $\sqrt{2}$ are in $\mathcal{R}$, but usually they cannot be represented exactly in a floating point type. (Some systems make provisions to represent a select few such numbers specially, and handle them 'symbolically' as much as possible.)

C.4.2 Definitions of terms

A vital definition is that of "notification". A notification is the report (to the program or user) that results from an error or exception as defined in ISO/IEC TR 10176 [7].

The principle behind notification is that such events in the execution of a program should not go unnoticed.

The preferred action is to use 'recording in indicators'. Another possibility is to invoke a change in the flow control of a program (for example, an Ada or Java "exception"), to allow the user to take corrective action. Changes of control flow are, however, harder to handle and recover from, especially if the notifications is not so serious and the computation may just continue. In particular, for **inexact** and **underflow** it is usually ill-advised to make a change in control flow, likewise for **infinitary** (usually divide-by-zero) notifications when infinity values are guaranteed to be available in the datatype. The practice in some older systems is that a notification consists of aborting execution with a suitable error message. This is hardly ever the proper action to take, and can be highly dangerous.

The various forms of notification are given names, such as **overflow**, so that they can be distinguished. However, bindings are not require to handle each named notification the same way everywhere. For example, **overflow** may be split into `integer-overflow` and `floating-overflow`, **infinitary** for integer results may result in a change of control flow, while **infinitary** on floating point results are handled by recording in indicators, only returning an infinitary continuation value while setting the indicator for **infinitary**.

One challenge is when computations are done in separate threads, and values communicated between the treads (or, in general, different programs maybe even running on different computers). When the method of notification is by change of control flow, the notification basically needs to be handled within the thread, otherwise the thread will terminate, and no further communication with it will take place. However, when the method of notification is by recording in indicators, the handling may be deferred. It may even be deferred to recipients of values in other threads (maybe even in other programs). For this to work, the indicators recording notifications need also be communicated along with the communicated value(s). Normally, this is not (yet) done by default, or automatically, so at present need to be done by explicit programming by application programmers.

Another important definition is that of a rounding function. A rounding function is a mapping from the real numbers onto a subset of the real numbers. Typically, the subset X is an "approximation" to $\mathcal{R}$, having unbounded range but limited precision. X is a discrete subset of $\mathcal{R}$, which allows precise identification of the elements of X which are closest to a given real number in $\mathcal{R}$. A rounding function rnd maps each real number y to an approximation of y that lies in X. If a real number y is in X, then clearly y is the best approximation for itself, so $rnd(y) = y$. If y

is between two adjacent values x_1 and x_2 in X, then one of these adjacent values must be the approximation for y:

$$x_1 < y < x_2 \;\Rightarrow\; (rnd(y) = x_1 \text{ or } rnd(y) = x_2)$$

Finally, if $rnd(y)$ is the approximation for y, and z is between y and $rnd(y)$, then $rnd(y)$ is the approximation for z also.

$$y < z < rnd(y) \;\Rightarrow\; rnd(z) = rnd(y)$$
$$rnd(y) < z < y \;\Rightarrow\; rnd(z) = rnd(y)$$

The last three rules are special cases of the monotonicity requirement

$$x < y \;\Rightarrow\; rnd(x) \leqslant rnd(y)$$

which appears in the definition of a rounding function.

Note that the value of $rnd(y)$ depends only on y and not on the arithmetic operation (or operands) that gave rise to y.

The graph of a rounding function looks like a series of steps. As y increases, the value of $rnd(y)$ is constant for a while (equal to some value in X) and then jumps abruptly to the next higher value in X.

Some examples may help clarify things. Consider a number of rounding functions from $\mathcal{R}$ to $\mathcal{Z}$. One possibility is to map each real number to the next lower integer:

$$rnd(u) = \lfloor u \rfloor$$

This gives $rnd(1) = 1$, $rnd(1.3) = 1$, $rnd(1.99\cdots) = 1$, and $rnd(2) = 2$. Another possibility would be to map each real number to the next higher integer. A third example maps each real number to the closest integer (with half-way cases rounding toward plus infinity):

$$rnd(u) = \lfloor u + 0.5 \rfloor$$

This gives $rnd(1) = 1$, $rnd(1.49\cdots) = 1$, $rnd(1.5) = 2$, and $rnd(2) = 2$. Each of these examples corresponds to rounding functions in actual use. For some floating point result examples, see C.5.2.4.

Note, the value $rnd(u)$ may not be representable in the target datatype. The absolute value of the rounded result may be too large. The $result_F$ function deals with this possibility. (See C.5.2.5 for further discussion.)

There is a precise distinction between *shall* and *should* as used in this standard: *shall* implies a requirement, while *should* implies a recommendation. One hopes that there is a good reason if the recommendation is not followed.

Additional definitions specific to particular types appear in the relevant clauses.

C.5 Specifications for integer and floating point datatypes and operations

Each arithmetic datatype conforming to LIA-1 consists of a subset of the real numbers characterized by a small number of parameters. Additional values may be included in an LIA-1 conforming datatype, especially infinities, negative zeroes, and NaNs ("not a number"). Two basic classes of types are specified: integer and floating point. A typical system could support several of each.

In general, the parameters of all arithmetic types must be accessible to an executing program.

The signature of each operation partially characterizes the possible input and output values. All operations are defined for all possible combinations of input values. Exceptions (like dividing

3 by 0) are modelled by the return of non-numeric exceptional values (like **invalid**, **infinitary**, etc.). The absence of an exceptional value in the result set of a signature does not indicate that that exception cannot occur, but that it cannot occur for values in the input set of the signature. Other exceptions, as well as other values, can be returned for inputs outside of the stated input values, e.g. infinities. The operation specifications (5.1.2, 5.2.6) state precisely when notifications must occur.

The philosophy of LIA-1 is that all operations either produce correct results or give a notification. For the operations specified in LIA, a notification must be based on the final result; there can be no spurious intermediate notifications. Arithmetic on bounded, non-modulo, integers must be correct if the mathematical result lies between $minint_I$ and $maxint_I$ and must produce a notification if the mathematically well-defined result lies outside this interval (**overflow**) or if there is no mathematically well-defined (and finite) result (**infinitary** or **invalid**). Arithmetic on floating point values must give a correctly rounded approximation if the approximation lies between $-fmax_F$ and $fmax_F$ and must produce a notification if the mathematically well-defined approximation lies outside this interval (**overflow**) or if there is no mathematically well-defined (and finite) approximation (**infinitary** or **invalid**).

C.5.1 Integer datatypes and operations

Most traditional computer programming languages assume the existence of bounds on the range of integers which can be values in integer types. Some languages place no limit on the range of integers, or even allow the boundedness of an integer type to be an implementation choice.

This standard uses the parameter $bounded_I$ to distinguish between implementations which place no restriction on the range of integer data values ($bounded_I$ = **false**) and those that do ($bounded_I$ = **true**). If the integer datatype (corresponding to) I is bounded, then two additional parameters are required, $minint_I$ and $maxint_I$. For unbounded integers, $minint_I$ and $maxint_I$ are required to have infinitary values. Infinitary values are required for unbounded integer types, and are allowed for bounded integer types.

For bounded integers, there are two approaches to out-of-range values: notification and "wrapping". In the latter case, all computation except comparisons is done *modulo* the cardinality of I (typically 2^n for some n), and no notification is required. The "wrapping" is modeled by operations specified in part 2.

C.5.1.0.1 Unbounded integers

Unbounded integers were introduced because there are languages which provide integers with no fixed upper limit. The value of the **Boolean** parameter $bounded_I$ must either be fixed in the language definition or must be available at run-time. Some languages permit the existence of an upper limit to be an implementation choice.

In an unbounded integer datatype implementation, every mathematical integer is potentially a data object in that datatype. The actual values computable depend on resource limitations, not on predefined bounds. Resource limitation problems are not modelled in LIA, but an implementation will need to make use of some notification to report the error back to the program (or program user). Note that also bounded integer datatypes may give rise to resource limitation errors, e.g. if the (intermediary) computed result cannot be stored.

LIA-1 does not specify how the unbounded integer datatype is implemented. Implementations will use a variable amount of storage for an integer, as needed. Indeed, if an implementation supplied a fixed amount of storage for each integer, this would establish a de facto $maxint_I$ and $minint_I$. It is important to note that this standard is not dependent upon hardware support for unbounded integers (which rarely, if ever, exists). In essence, LIA-1 requires a certain abstract functionality, and this can be implemented in hardware, software, or more typically, a combination of the two.

Operations on unbounded integers will never overflow. However, the storage required for unbounded integers can result in a program failing due to lack of memory. This is logically no different from failure through other resource limits, such as time.

The implementation may be able to determine that it will not be able to continue processing in the near future and may issue a warning. Some recovery may or may not be possible. It may be impossible for the system to identify the specific location of the fault. However, the implementation must not give false results without any indication of a problem. It may be impossible to give a definite "practical" value below which integer computation is guaranteed to be safe, because the largest representable integer at time t may depend on the machine state at that instant. Sustained computations with very large integers may lead to resource exhaustion.

Natural numbers (upwardly unbounded non-negative integers) are not modelled by LIA-1.

The signatures of the integer operations include **overflow** as a possible result because they refer to bounded integer operations as well.

C.5.1.0.2 Bounded non-modulo integers

For bounded non-modulo integers, it is necessary to define the range of representable values, and to ensure that notification occurs on any operation which would give a mathematical result outside that range. Different ranges result in different integer types. The values of the parameters $minint_I$ and $maxint_I$ must be accessible to an executing program.

The allowed ranges for integers fall into three classes:

a) $minint_I = 0$, corresponding to *unsigned* integers. The operation neg_I would always produce **overflow** (except on 0), and may be omitted.

   The operation abs_I is the identity mapping and may also be omitted. The operation div_I never produces **overflow**.

b) $minint_I = -maxint_I$, corresponding to *one's complement* or *sign-magnitude* integers. None of the operations neg_I, abs_I or div_I produces **overflow**.

c) $minint_I = -(maxint_I + 1)$, corresponding to *two's complement* integers. The operations neg_I and abs_I produce **overflow** only when applied to $minint_I$. The operation div_I produces **overflow** when $minint_I$ is divided by -1, since

$$minint_I/(-1) = -minint_I = maxint_I + 1 > maxint_I.$$

The Pascal, Modula-2, and Ada programming languages support subranges of integers. Such subranges typically do not satisfy the rules for $maxint_I$ and $minint_I$. However, that is not to say that these languages have non-conforming integer datatypes. Each subrange type can be viewed as a subset of an integer datatype that does conform to LIA-1. Integer operations are defined on those integer datatypes, and the subrange constraints only affect the legality of assignment and parameter passing.

C.5.1.0.3 Modulo integers

Modulo arithmetic operations on integers were introduced because there are languages that mandate wrapping operations for some integer types (e.g., C's **unsigned int** type), and make it optional for others (e.g., C's **signed int** type).

Modulo arithmetic operations on integer datatypes behave as above, but wrap rather than overflow when an operation would otherwise return a value outside of the range of the datatype. However, in this edition, this is modelled as separate operations from the add_I etc. operations. A binding may however use the same syntax for add_I and add_wrap_I (etc. for other operations), and let the datatypes of the arguments, or a $modulo_I$ parameter, imply which LIA operation is invoked.

Bounded modulo integers (in the limited form defined here) are definitely useful in certain applications. However, bounded integers are most commonly used as an efficient hardware approximation to true mathematical integers. In these latter cases, a wrapped result would be severely inaccurate, and should result in a notification. Unwary use of modulo integers can easily lead to undetected programming errors.

The developers of a programming language standard (or binding standard) should carefully consider which (if any) of the integral programming language types are bound to modulo integers. Since modulo integers are dangerous, programmers should always have the option of using non-modulo (overflow checking) integers instead.

Some languages, like Ada, allow programmers to declare new modulo integer datatypes, usually unsigned. Since the upper limit is then also programmer defined, the lower limit usually fixed at zero, these datatypes are more flexible, and very useful.

C.5.1.1 Integer result function

The integer result function, $result_I$, takes as argument the exact mathematical result of an integer operation, and checks that it is in the bounds of the integer datatype. If so, the value is returned. If not, the exceptional value **overflow** is returned.

The $result_I$ helper function is used in the specifications of the integer operations, and is used to consistently and succinctly express the overflow notification cases. The continuation value on overflow is binding or implementation defined.

The helper function $wrap_I$ (LIA-2), used to specify modulo operations (or "modulo integers") produces results in the range $[minint_I, maxint_I]$, and never produces an overflow notification. These results are positive for unsigned integer types, but may be negative for signed types.

C.5.1.2 Integer operations

C.5.1.2.1 Comparisons

The comparisons are always exact and never produce any notification.

C.5.1.2.2 Basic arithmetic

The ratio of two integers is not necessarily an integer. Thus, the result of an integer division may require rounding. Two rounding rules are in common use: *round toward minus infinity* ($quot_I$), and *round toward zero*. The latter is not specified by LIA-1, due to proneness for erroneous use, when the arguments are of different signs. For example,

$quot_I(-3, 2) = -2$	round toward minus infinity, specified in LIA-1
$div^t_I(-3, 2) = -1$	round toward zero, no longer specified by any part of LIA

$quot_I$ (called div^f_I in the first edition of LIA-1) as well as $ratio_I$ and $group_I$ (specified in LIA-2) all satisfy a broadly useful translation invariant:

$$quot_I(x + i * y, y) = quot_I(x, y) + i \qquad \text{if } y \neq 0 \text{, and no overflow occurs}$$

(and similarly for the $ratio_I$ and $group_I$). $quot_I$ is the form of integer division preferred by many mathematicians. div^t_I (no longer specified by LIA) is the form of division introduced by Fortran.

Integer division is frequently used for grouping. For example, if a series of indexed items are to be partitioned into groups of n items, it is natural to put item `i` into group `i/n`. This works fine if $quot_I$ is used for integer division. However if div^t_I (no longer specified by LIA) is used, and `i` can be negative, group 0 will get $2 \cdot n - 1$ items rather than the desired n. This uneven behaviour for negative `i` can cause subtle program errors, and is a strong reason against the use of div^t_I, and for the use of the other integer division operations.

mod_I gives the remainder after $quot_I$ division. It is coupled to $quot_I$ by the following identities:

$$\begin{array}{ll} x = quot_I(x, y) * y + mod_I(x, y) & \text{if } y \neq 0\text{, and no overflow occurs} \\ y < mod_I(x, y) \leqslant 0 & \text{if } y < 0 \\ 0 \leqslant mod_I(x, y) < y & \text{if } y > 0 \end{array}$$

Thus, $quot_I$ and mod_I form a logical pair. So do $ratio_I$ and $residue_I$, as well as $group_I$ and negated(!) pad_I (specified in LIA-2). Note that computing $mod_I(x, y)$ only as

$$sub_I(x, mul_I(quot_I(x, y), y))$$

is not correct for asymmetric bounded integer types, because $quot_I(x, y)$ can overflow but $mod_I(x, y)$ cannot.

C.5.2 Floating point datatypes and operations

Floating point values are traditionally represented as

$$X = \pm g * r_F^e = \pm 0.f_1 f_2 ... f_{p_F} * r_F^e$$

where $0.f_1 f_2 ... f_{p_F}$ is the p_F-digit *fraction* g (represented in base, or radix, r_F) and e is the exponent. This includes subnormal numbers, including zero and negative zero. (Usually, $f_1 = 0$ is permitted only for the minimal value for the exponent, in order to get unique representation of values. However, IEC 60559 does not make this restriction for base 10.)

The exponent e is an integer in $[emin_F, emax_F]$. The fraction digits are integers in $\{0, ..., r_F - 1\}$. If the floating point number is *normalized*, f_1 is not zero, and hence the minimum normalised value of the fraction g is $1/r_F$ and the maximum value is $1 - r_F^{-p_F}$. Subnormal values, including 0, cannot be represented in normalised form, since that would call for an exponent smaller than $emin_F$, indeed, for 0 the exponent would be negative infinity.

This description gives rise to five parameters that characterize the set of non-special values of a floating point datatype:

radix r_F: the "base" of the number system.

precision p_F: the number of radix r_F digits provided by the type.

$emin_F$ and $emax_F$: the smallest and largest exponent values. They define the range of the type.

$denorm_F$ (a **Boolean**): **true** if the datatype includes the *subnormal* values; **false** if the only subnormal values included are zeroes (non-conforming case).

For normalised values, the fraction g can also be represented as $i * r_F^{-p_F}$, where i is a p_F digit integer in the interval $[r_F^{p_F-1}, r_F^{p_F} - 1]$. Thus

$$X = \pm g * r_F^e = \pm(i * r_F^{-p_F}) * r_F^e = \pm i * r_F^{e-p_F}$$

This is the form of the floating point values used in defining the finite set F_N. The exponent e is often represented with a bias added to the true exponent, in order to avoid having to deal with negative integer representations for the exponent.

$denorm_F$ must be **true** for a fully conforming datatype. It is allowed to be **false** only for partially conforming datatypes.

The IEEE 754 [34] (same as IEC 60559) present a slightly different model for the floating point type. Normalized floating point numbers are represented as

$$\pm f_0.f_1...f_{p_F-1} * r_F^e$$

where $f_0.f_1...f_{p_F-1}$ is the p_F-digit *significand* (represented in radix r_F, where r_F is 2 or 10), and e is an integer exponent between $emin_F - 1$ (now, confusingly, called *emin* in IEEE 754) and $emax_F - 1$ (now, confusingly, called *emax* in IEEE 754). For $r_F = 2$ the minimum value of the significand for a normalised value is 1 and the maximum value is $r_F - 1/r_F^{p_F-1}$. For $r_F = 10$ the minimum value of the significand is 0 and the maximum value is $r_F - 1/r_F^{p_F-1}$, allowing for denormal values that need not be subnormal. LIA does not distinguish between normal and denormal representations of the same value. The IEEE significand is equivalent to $g * r_F$.

The fraction model and the significand model are equivalent in that they can generate precisely the same sets of floating point values. Currently, all ISO/IEC JTC1/SC22 programming language standards that present a model of floating point to the programmer use the fraction model rather than the significand one. LIA-1 has chosen to conform to this trend.

Note though, that the $fraction_F$ and $exponent_{F\to I}$ operations return values for subnormal (including 0) numbers, as if the exponent range was unlimited, and not tied to the integer representing the exponent part in the usual representations for floating point values. Thus the result of $exponent_{F\to I}$ is not limited to $[emin_F, emax_F]$.

C.5.2.0.1 Constraints on the floating point parameters

The constraints placed on the floating point parameters are intended to be close to the minimum necessary to have the model provide meaningful information. We will explain why each of these constraints is required, and then suggest some constraints which have proved to be characteristic of useful floating point datatypes.

LIA-1 requires that $r_F \geqslant 2$ and $p_F \geqslant 2 \cdot \max\{1, \lceil \log_{r_F}(2 \cdot \pi) \rceil\}$ in order to ensure that a meaningful set of values. At present, only 2, 8, 10, and 16 appear to be in use as values for r_F (except for string representations, where sometimes, as for Ada, a wider range of radices are permitted: like for Ada, from 2 to 36). The first edition of LIA-1 only required that $p_F \geqslant 2$, but such a low bound gives trouble in the specifications of some of the elementary function operations (LIA-2). Indeed, p_F should be such that $p_F \geqslant 2 + \lceil \log_{r_F}(1000) \rceil$, so that the trigonometric operations can be meaningful for more than just one cycle, but at 100 or so cycles. If the radix is 2, that means at least 12 binary digits in the fraction part. As a comparison, note that the IEEE 754 storage format 'binary16' has $p_F = 11$, and is thus not suitable for computations.

The requirement that $emin_F \leqslant 2 - p_F$ ensures that $epsilon_F$ is representable in F.

The requirement that $emax_F \geqslant p_F$ ensures that $1/epsilon_F$ is representable in F. It also implies that all integers from 1 to $r_F^{p_F} - 1$ are exactly representable.

The parameters r_F and p_F logically must be less than $r_F^{p_F}$, so they are automatically in F. The additional requirement that $emax_F$ and $-emin_F$ are at most $r_F^{p_F} - 1$ guarantees that $emax_F$ and $emin_F$ are in F as well.

A consequence of the above restrictions is that a language binding can choose to report r_F, p_F, $emin_F$, and $emax_F$ to the programmer either as integers or as floating point values without loss of accuracy.

Constraints designed to provide:

a) adequate precision for scientific applications,

b) "balance" between the overflow and underflow thresholds, and

c) "balance" between the range and precision parameters

are applied to safe model numbers in Ada [11]. No such constraints are included in LIA-1, since LIA-1 emphasizes descriptive, rather than prescriptive, specifications for arithmetic datatypes. However, the following restrictions have some useful properties:

a) r_F should be even

   An even value of r_F makes certain rounding rules easier to implement. In particular, rounding to nearest would pose a problem because with r_F odd and $d = \lfloor r_F/2 \rfloor$ we would have $\frac{1}{2} = .ddd\cdots$. Hence, for $x_1 < x < x_1 + ulp_F(x_1)$ a reliable test for x relative to $x_1 + \frac{1}{2}ulp_F(x_1)$ could require retention of many guard digits.

b) $r_F^{p_F - 1} \geqslant 10^6$

   This gives a maximum relative error ($epsilon_F$) of one in a million. This is easily accomplished by 24 binary or 6 hexadecimal digits.

c) $emin_F - 1 \leqslant -k * (p_F - 1)$ with $k \geqslant 2$ and k as large an integer as practical

   This guarantees that $epsilon_F^k$ is in F which makes it easier to simulate higher levels of precision than would be offered directly by the values in the datatype.

d) $emax_F > k * (p_F - 1)$

   This guarantees that $epsilon_F^{-k}$ is in F and is useful for the same reasons as given above.

e) $-2 \leqslant (emin_F - 1) + emax_F \leqslant 2$

   This guarantees that the geometric mean $\sqrt{fminN_F * fmax_F}$ of $fminN_F$ and $fmax_F$ lies between $1/r_F$ and r_F.

All of these restrictions are satisfied by most (if not all) implementations. A few implementations present a floating point model with the radix point in the middle or at the low end of the fraction. In this case, the exponent range given by the implementation must be adjusted to yield the LIA-1 $emin_F$ and $emax_F$. In particular, even if the minimum and maximum exponent given in the implementation's own model were negatives of one another, the adjusted $emin_F$ and $emax_F$ become asymmetric.

C.5.2.0.2 Radix complement floating point

LIA-1 presents an abstract model for a floating point datatype, defined in terms of parameters. An implementation is expected to be able to map its own floating point numbers to the elements in this model, but LIA-1 places no restrictions on the actual internal representation of the floating point values.

The floating point model presented in LIA-1 is sign-magnitude. A few implementations keep their floating point fraction in a radix-complement format. Several different patterns for radix-complement floating point have been used, but a common feature is the presence of one "extra" negative floating point number, that has no positive counterpart: the most negative. Its value is $-fmax_F - ulp_F(fmax_F)$. Some radix-complement implementations also omit the negative counterpart of $fmin_F$.

In order to accommodate radix-complement floating point, LIA-1 would have to make at least the following changes:

a) define additional derived constants which correspond to the negative counterparts of $fmin_F$ (the "least negative" floating point number) and $fmax_F$ (the "most negative" floating point number) and change the definitions of F_N, F_S, etc.;

b) negation and absolute value will also be **inexact** for some values close to exponent boundaries;

c) add **overflow** to the signature of neg_F (because neg_F evaluated on the most negative number will now overflow);

d) add **overflow** to the signature of abs_F (because abs_F will now overflow when evaluated on the most negative number);

e) add **underflow** to the signature of neg_F, if $-fmin_F$ is omitted;

f) expand the definitions of sub_F and $trunc_F$, and redefine these operations and also $round_F$ operations to ensure that these operations behave correctly.

g) redefine the $pred_F$ and $succ_F$ operations to treat the new most negative floating point number properly.

Because of this complexity, LIA-1 does not include radix-complement floating point.

Floating point implementations with sign-magnitude or (radix$-$1)-complement fractions can map the floating point numbers directly to LIA-1 model without these adjustments.

C.5.2.1 Conformity to IEC 60559

IEC 60559 is the international version of IEEE 754.

Note that "methods shall be provided for a program to access each [IEC 60559] facility". This means that a complete LIA-1 binding will include a binding for IEC 60559 as well.

C.5.2.1.1 Subnormal numbers

IEEE 754 (IEC 60559) datatypes and some non-IEEE floating point datatypes include subnormal numbers. 0 and $\mathbf{-0}$ are two of the subnormal numbers. Floating point formats can in general represent subnormal numbers, since the floating point representation must be able to represent 0, and that is usually done in a way that can readily be generalised to represent any subnormal number. LIA-1 models a subnormal floating point number as a real number of the form

$$X = \pm i * r_F^{emin_F - p_F}$$

where i is an integer in the interval $[0, r_F^{p_F-1} - 1]$. The corresponding fraction g lies in the interval $[0, 1/r_F - r_F^{-p_F}]$; its most significant digit is zero. Subnormal numbers partially fill the "underflow gap" in F: the interval $]-\mathit{fminN}_F, \mathit{fminN}_F[$ (that is: $]-r_F^{emin_F-1}, r_F^{emin_F-1}[$). Taken together, apart from $\mathbf{-0}$, they comprise the set F_S.

The values in F_S are linearly distributed with the same spacing as the values in the neighbouring ranges $]-r_F^{emin_F}, -r_F^{emin_F-1}]$ and $[r_F^{emin_F-1}, r_F^{emin_F}[$ in F_N. Thus they have a maximum *absolute* representation error of $r_F^{emin_F-p_F}$. However, since subnormal numbers have less than p_F digits of precision, the *relative* representation error varies widely. The relative error varies from $\mathit{epsilon}_F = r_F^{1-p_F}$ at the high ends of F_S to 1 at the low ends of F_S. Near 0, in $[-\mathit{fminD}_F, \mathit{fminD}_F]$, the relative error may be unboundedly large.

Whenever an addition or subtraction, resulting in the same datatype as the arguments, produces a result in F_S, that result is exact – the relative error is zero. Even for an "effective subtraction" no accuracy is lost, because the decrease in the number of significant digits is exactly the same as the number of digits cancelled in the subtraction. For multiplication, division, scaling, and some conversions, significant digits (and hence accuracy) may be lost if the result is in F_S. This loss of significant digits is called denormalisation loss.

The entire set of floating point numbers F is either $F_N \cup F_S$ (if subnormal numbers are provided), or $F_N \cup \{0\}$ (if all available non-special numbers, except 0, are normalized). For full conformity LIA-1 requires the use of subnormal numbers.

C.5.2.1.2 Signed zero

The IEEE standards define both 0 and $\mathbf{-0}$. Very few non-IEEE floating point datatypes provide the user with two "different" zeros. Even for an IEEE datatype, the two zeroes can only be distinguished with a few operations, not including comparisons, but e.g. using dividing by zero to obtain signed infinity, by (correctly) converting to a character string numeral, or by using operations that have a branch cut along an axis, like arc_F (LIA-2) or some complex inverse trigonometric operation (LIA-3). Programs that require that 0 and $\mathbf{-0}$ are distinct might not be portable to systems without IEEE floating point datatypes.

C.5.2.1.3 Infinities and NaNs

IEEE 754 [34] provides special values to represent *infinities* and *Not-a-Numbers. Infinity* represents an exact quantity (for instance from dividing a finite number by zero). Infinities are also used as an inexact continuation value for overflow. A *NaN* represents an indeterminate or unrepresentable quantity (e.g. from dividing zero by zero, or square root of -1 when that result cannot be represented).

Most non-IEEE floating point datatypes do not provide infinities or (quiet) NaNs. Thus, programs that make use of infinity or quiet NaNs will not be portable to systems that do not provide them.

Note also that LIA-1 requires the presence of both negative and positive infinity in unbounded integer datatypes. Also quiet NaNs should be provided. Such requirements are not made for bounded integer datatypes, since such datatypes are most often supported directly by hardware.

C.5.2.2 Range and granularity constants

The positive real numbers fmax_F, fmin_F, fminN_F, and fminD_F are interesting boundaries in the set F. fmax_F is the "overflow threshold" (though overflow may be tested before or after rounding). It is the largest value in F (and thereby also F_N). fmin_F is the value of smallest magnitude in F. fminN_F is the "subnormal threshold". It is the smallest normalized value in F: the point where the number of significant digits begins to decrease. Finally, fminD_F is the smallest strictly positive subnormal value, representable only if denorm_F is **true**.

This standard requires that the values of fmax_F, fmin_F, and fminN_F be accessible to an executing program. All non-zero floating point values fall in the ranges $[-\mathit{fmax}_F, -\mathit{fmin}_F]$ and $[\mathit{fmin}_F, \mathit{fmax}_F]$, and values in the ranges $[-\mathit{fmax}_F, -\mathit{fminN}_F]$ and $[\mathit{fminN}_F, \mathit{fmax}_F]$ can be represented with full precision.

The derived constant fminD_F need not be given as a run-time parameter. For a datatype in which subnormal numbers are provided (and enabled), the value of fminD_F is fmin_F. If subnormal numbers are not present, the constant fminD_F is not representable, and $\mathit{fmin}_F = \mathit{fminN}_F$.

The derived constant $\mathit{epsilon}_F$ must also be accessible to an executing program:

$$\mathit{epsilon}_F = r_F^{1-p_F}$$

It is defined as ratio of the weight of the least significant digit of the fraction g, $r_F^{-p_F}$, to the minimum value of a normalised g, $1/r_F$. So $\mathit{epsilon}_F$ can be described as the largest relative representation error for the set of normalised values in F_N.

An alternate definition of $\mathit{epsilon}_F$ currently in use is the smallest floating point number such that the expression `1 +` $\mathit{epsilon}_F$ yields a value greater than 1. This definition is flawed because it depends on the characteristics of the rounding function. For example, using round to nearest, $\mathit{epsilon}_F$ would be half of what it is defined to be in LIA. In the extreme, on an IEEE floating point datatype using round-to-positive-infinity, $\mathit{epsilon}_F$ would be fminD_F.

C.5.2.2.1 Relations among floating point datatypes

An implementation may provide more than one conforming floating point datatype, and most systems do. It is usually possible to order those with a given radix as $F_1, F_2, F_3, \cdots$ such that

$$p_{F_1} < p_{F_2} < p_{F_3} \cdots$$
$$\mathit{emin}_{F_1} \geqslant \mathit{emin}_{F_2} \geqslant \mathit{emin}_{F_3} \cdots$$
$$\mathit{emax}_{F_1} \leqslant \mathit{emax}_{F_2} \leqslant \mathit{emax}_{F_3} \cdots.$$

A number of systems do not increase the exponent range with precision. However, the following constraints

$$\begin{aligned}&2 \cdot p_{F_i} \leqslant p_{F_{i+1}}\\&2 \cdot (emin_{F_i} - 1) \geqslant (emin_{F_{i+1}} - 1)\\&2 \cdot emax_{F_i} \leqslant emax_{F_{i+1}}\end{aligned}$$

for each pair F_i and F_{i+1} would provide advantages to programmers of numerical software (for floating point datatypes not at the widest level of range-precision):

a) The constraint on the increase in precision expedites the accurate calculation of residuals in an iterative procedure. It also provides exact products for the calculation of an inner product or a Euclidean norm.

b) The constraints on the increase in the exponent range makes it easy to avoid the occurrence of an overflow or underflow in the intermediate steps of a calculation, for which the final result is in range.

C.5.2.3 Approximate operations

As an example, apply a three stage model to multiplication ($mul_{F\to F}(x, y)$) (for simplicity, assuming the target type is the same as the argument type, F):

a) First, compute the perfect result, $x * y$, as an element of $\mathcal{R}$.

b) Second, modify this to form a rounded result, $nearest_F(x * y)$, as an element of $F^\dagger$.

c) Finally, decide whether to accept the rounded result or to cause a notification.

Putting this all together, we get the defining case for multiplication when both arguments and the result are in F:

$$mul_{F\to F}(x, y) = result_F(x \cdot y, nearest_F) \quad \text{if } x, y \in F$$

The $result_F$ function is defined to compute the rounded result internally, since the result depend on the properties of the rounding function itself, as well as the exact value, not just the rounded result.

Note that in reality, step (a) only needs to compute enough of $x \cdot y$ to be able to complete steps (b) and (c), i.e., to produce a rounded result and to decide on overflow, inexact, and underflow.

The helper functions $nearest_F$, $result_F$, are the same for all the operations of a given floating point type.

The helper functions are not visible to the programmer, but they are included in the required documentation of the type. This is because these functions form the most concise description of the semantics of the approximate operations.

C.5.2.4 Rounding and rounding constants

Floating point operations are rarely exact. The true mathematical result seldom lies in F, so the mathematical result must be rounded to a nearby value that does lie in F. For convenience, this process is described in three steps: first the exact value is computed, then a determination is made about overflow, underflow, or inexact, finally the exact value is rounded to the appropriate precision and a continuation value is determined.

A rounding rule is specified by a rounding function which maps values in $\mathcal{R}$ onto values in $F^\dagger$. $F^\dagger$ is the set $F_S \cup F_N$ augmented with all values of the form $\pm i * r_F^{e-p_F}$ where $r_F^{p_F-1} \leqslant i \leqslant r_F^{p_F} - 1$ (as in F_N) but $e > emax_F$. The extra values in $F^\dagger$, i.e. F_E, are unbounded in range, but all have exactly p_F digits of precision. These are "helper values", and are not representable in the type F.

The round to nearest rule is "sign symmetric", $nearest_F(-x) = -nearest_F(x)$. This assures that the arithmetic operations $add_{F \to F'}$, $sub_{F \to F'}$, $mul_{F \to F'}$, and $div_{F \to F'}$ have the expected behaviour with respect to sign, as described in C.5.2.8.

In addition to being a rounding function (as defined in 4.2), $nearest_F$ does not depend upon the exponent of its input (except for subnormal values). This is captured by a "scaling rule":

$$nearest_F(x \cdot r_F^j) = nearest_F(x) \cdot r_F^j$$

which holds as long as x and $x \cdot r_F^j$ have magnitude greater than or equal to $fminN_F$.

Subnormal values have a wider relative spacing than 'normal' values. Thus, the scaling rule above does not hold for all x in the subnormal range. When the scaling rule fails, we say that $nearest_F$ has a *denormalization loss* at x, and the relative error

$$\left|\frac{x - nearest_F(x)}{x}\right|$$

is typically larger than for 'normal' values.

A constant may be provided, and must be provided if the operation variants specified in A.6 are allowed, to give the programmer access to some information about the rounding function in use. rnd_error_F describes the maximum rounding error (in ulps). Floating point datatypes that fully conform to LIA-1 have $rnd\_error_F = 0.5$. This is a value of the rounding error that is actually allowed, that is, the actual rounding error for any inexact LIA-1 operation is in the interval $[0, 0.5]$ ulp. Partially conforming floating point datatypes can have an $rnd\_error_F = 1$. This is a value of the (partially conforming) rounding error that is not actually allowed, that is, the actual rounding error for any inexact LIA-1 operation is in the interval $[0, 1[$ ulp.

IEEE 754 [34] defines five rounding methods. IEEE 754 allows for dynamic setting of a rounding mode to choose between them, but sometimes have different operations for different rounding methods.

a) *Round to nearest, ties to even last digit.* In the case of a value exactly half-way between two neighbouring values in $F^\dagger$, select the "even" result. That is, for non-negative x in $F^\dagger$

$$\begin{aligned} rnd(x + \tfrac{1}{2} \cdot u_F(x)) &= x + u_F(x) && \text{if } x/u_F(x) \text{ is odd} \\ &= x && \text{if } x/u_F(x) \text{ is even} \end{aligned}$$

This is the default rounding mode in IEEE 754.

b) *Round to nearest, ties away from zero.* In the case of a half-way value, round away from zero. That is, for non-negative x in $F^\dagger$

$$rnd(x + \tfrac{1}{2} \cdot u_F(x)) = x + u_F(x)$$

This rounding mode is in IEEE 754 specified only for radix 10 datatypes, though it has traditionally been used also for other radices (2, 8, 16).

c) *Round toward zero.*

d) *Round toward minus infinity.*

e) *Round toward plus infinity.*

The first three of these rounding rules are sign symmetric, but the last two are not. However, the last two are useful for getting tighter error bounds for interval arithmetic, provided that "flush to zero" on underflow is *not* used or is compensated for. The first two rounding methods give a half ulp error bound, so rnd_error_F is 0.5. The last three rounding methods give a one ulp error bound, so rnd_error_F is 1. Most non-IEEE implementations provide either the second or the third rule.

C.5.2.5 Floating point result function

The rounding function, like $nearest_F$, can return values that are not bounded by $fmax_F$. A result function is then used to check whether the result is within range, and to generate an exceptional value if required. The result function $result_F$ takes two arguments. The first one is a real value x (typically the mathematically correct result) and the second one is a rounding function rnd to be applied to x.

If F does not include subnormal numbers, and $rnd(x)$ is representable, then $result_F$ returns $rnd(x)$ (as a continuation value to **inexact** if $x \neq rnd(x)$). If $rnd(x)$ is too large or too small to be represented as a normal value, then $result_F$ returns **overflow** or **underflow**.

The only difference when F does contain subnormal values occurs when rnd returns a subnormal value. If there was a denormalization loss in computing the rounded value, then $result_F$ must return **underflow** with $rnd(x)$ as continuation value. On the other hand, if there was no denormalization loss, then the implementation is to return $rnd(x)$ as a continuation value either to **inexact** or **underflow** exceptional value return.

$result_F(x, rnd)$ takes rnd as its second argument (rather than taking $rnd(x)$) because one of the final parts of the definition of $result_F$ refers to denormalization loss. Denormalization loss is a property of the function rnd at x rather than of the individual value $rnd(x)$. In addition, the continuation value upon overflow, depends on the rounding function.

C.5.2.6 Floating point operations

This clause describes the floating point operations defined by the standard.

An implementation can easily provide any of these operations in software. However, portable versions of these operations will not be as efficient as those which an implementation provides and "tunes" to the architecture.

C.5.2.6.1 Comparisons

The comparison operations are atomic operations which never produce a notification when the arguments are in F, and then always return **true** or **false** in accordance with the exact mathematical result.

C.5.2.6.2 Basic arithmetic

a) The operations $add_{F \to F'}$, $sub_{F \to F'}$, $mul_{F \to F'}$, and $div_{F \to F'}$ carry out the usual basic arithmetic operations of addition, subtraction, multiplication, and division.

b) The operations neg_F and abs_F produce the negative and absolute value, respectively, of the input argument. They never overflow or underflow.

c) The operation $signum_F$ returns a floating point 1 or -1, depending on whether its argument is positive (including zero) or negative (including negative zero).

C.5.2.6.3 Value dissection

a) The operation $exponent_F$ gives the exponent of the floating point number in the model as presented in LIA-1, but as though the range of exponent values was unbounded. The value of $exponent_F$ can also be thought of as the "order of magnitude" of its argument, i.e., if n is an integer such that $r_F^{n-1} \leqslant x < r_F^n$, then $exponent_F(x) = n$. $exponent_F(0)$ is negative infinity (with a **infinitary** notification).

b) The operation $fraction_F$ scales its argument (by a power of r_F) until it is in the range $\pm[1/r_F, 1[$. Thus, for $x \neq 0$,

$$x = fraction_F(x) * r_F^{exponent_F(x)}$$

c) The operation $scale_F$ scales a floating point number by an integer power of the radix.

d) The operation $succ_F$ returns the closest element of F greater than the argument, the "successor" of the argument.

e) The operation $pred_F$ returns the closest element of F less than the argument, its "predecessor".

The $succ_F$ and $pred_F$ operations are useful for generating adjacent floating point numbers, e.g. in order to test an algorithm in the neighbourhood of a "sensitive" point.

f) The operation ulp_F gives the value of one unit in the last place, i.e., its value is the weight of the least significant digit. It will underflow catastrophically if $denorm_F =$ **false** in the range $]-r_F^{emin_F+p_F}, r_F^{emin_F+p_F}[$, but will never return an exceptional value if $denorm_F =$ **true**.

Standardizing functions such as $exponent_F$ and ulp_F helps shield programs from explicit dependence on the underlying format.

Note that the helper function e_F is not the same as the $exponent_F$ operation. They agree on 'normal' numbers in F, but differ if the argument is subnormal (including zero). $exponent_F(x)$ is chosen to be the exponent of x as though x were in normalized form and the range and precision were unbounded. For subnormal numbers, $e_F(x)$ is equal to $emin_F$. In addition, e_F is defined for an argument in $\mathcal{R}$, while $exponent_F$ is defined for arguments in F (plus special values).

C.5.2.6.4 Value splitting

a) The operation $trunc_F(x, n)$ zeros out the low $(p_F - n)$ digits of the first argument. When $n \leqslant 0$ then 0 is returned; and when $n \geqslant p_F$ the argument is returned.

b) The operation $round_F(x, n)$ rounds the first argument to n significant digits. That is, the nearest n-digit floating point value is returned. Values exactly half-way between two adjacent n-digit floating point numbers round away from zero. $round_F$ differs from $trunc_F$ by at most 1 in the n-th digit.

The $trunc_F$ and $round_F$ operations can be used to split a floating point number into a number of "shorter" parts in order to expedite the simulation of multiple precision operations without use of operations at a higher level of precision.

c) The operation $intpart_F$ isolates the integer part of the argument and returns this result in floating point form. Note that this is done by truncation, so it is sign symmetric bot not translation regular.

d) The operation $fractpart_F$ returns the value of the argument minus its integer part obtained by $intpart_F$. So this is sign symmetric but not invariant (apart from rounding) by integer translation (since the integer part is computed by truncation, and not by floor, ceiling, nor nearest (ties to even)). However, this way an exact result is obtained with no loss of accuracy due to rounding.

C.5.2.7 Levels of predictability

This clause explains why the method used to specify floating point types was chosen. The main question is, "How precise should the specifications be?" The possibilities range from completely prescriptive (specifying every last detail) to loosely descriptive (giving a few axioms which essentially every floating point system already satisfies).

IEEE 754 [34] takes the highly prescriptive approach, allowing relatively little latitude for variation. It even stipulates much of the representation. The Brown model [42] comes close to the other extreme, even permitting non-deterministic behaviour.

There are (at least) five interesting points on the range from a strong specification to a very weak one. These are

a) Specify the set of representable values exactly; define the operations exactly; but leave the representations unspecified.

b) Allow limited variation in the set of representable values, and limited variation in the operation semantics. The variation in the value set is provided by a small set of parameters.

c) Use parameters to define a "minimum" set of representable values, and an idealized set of operations. This is called a *model*. Implementations may provide more values (extra precision), and different operation semantics, as long as the implemented values and operations are sufficiently close to the model. The standard would have to define "sufficiently close".

d) Allow any set of values and operation semantics as long as the operations are deterministic and satisfy certain accuracy constraints. Accuracy constraints would typically be phrased as maximum relative errors.

e) Allow non-deterministic operations.

The IEEE model is close to (a). The Brown model is close to (e). LIA-1 selects the second approach because it permits conformity by most current systems, provides flexibility for high performance designs, and discourages increase in variation among future systems.

Note that the Brown model allows "parameter penalties" (reducing p_F or $emin_F$ or $emax_F$) to compensate for inaccurate hardware. The LIA-1 model does not permit parameter penalties.

A major reason for rejecting a standard based upon the Brown model is that the relational operations do not (necessarily) have the properties one expects. For instance, with the Brown model, $x < y$ and $y < z$ does not imply that $x < z$ (assuming $<$ here is in some programming language).

C.5.2.8 Identities

By choosing a relatively strong specification of floating point, certain useful identities are guaranteed to hold. The following is a sample list of such identities. These identities can be derived from the axioms defining the arithmetic operations.

The approximate operations ($add_{F \to F'}$, $sub_{F \to F'}$, $mul_{F \to F'}$, $div_{F \to F'}$, $sqrt_{F \to F'}$, ...) compute approximations to the ideal mathematical functions. Note that in this version of LIA-1 the approximate operations all have a source and a target type. This way double rounding is avoided when the target type has less precision than the argument type. IEC 60559 (IEEE 754), also in the version from 1989, requires this avoidance of double rounding for these operations. Some of the double rounding avoiding operations were previously (only) in LIA-2, but are now covered by LIA-1.

Since the approximate operations specified in LIA-1 are all so similar, it is convenient to give a series of rules that apply to all of them (with some qualifications). Let Φ be any of the LIA-1 approximate operations with arguments in F and result in F', and let ϕ be the corresponding ideal mathematical function (in the case of conversions, ϕ is the identity function). If ϕ is a single argument function, ignore the second argument.

When $\phi(x, y)$ is defined for $x, y \in F$, and no notification apart from **inexact** occurs (for **inexact**, consider the continuation value),

$$u \leqslant \phi(x, y) \leqslant v \;\Rightarrow\; u \leqslant \Phi(x, y) \leqslant v \qquad \text{(I)}$$

when $u, v \in F'$.

When $\phi(x, y)$ is defined for $x, y \in F$,

$$\phi(x, y) \in F' \;\Rightarrow\; \Phi(x, y) = \phi(x, y) \qquad \text{(II)}$$

When $\phi(u, x)$ and $\phi(v, y)$ are defined for $x, y, u, v \in F$, and no notification apart from **inexact** occurs (for **inexact**, consider the continuation value),

$$\phi(u, x) \leqslant \phi(v, y) \;\Rightarrow\; \Phi(u, x) \leqslant \Phi(v, y) \qquad \text{(III)}$$

When $\phi(x, y)$ is defined for $x, y \in F$, and no notification apart from **inexact** occurs (for **inexact**, consider the continuation value),

$$|\Phi(x, y) - \phi(x, y)| \leqslant u_F(\phi(x, y)) \leqslant u_F(\Phi(x, y)) \qquad \text{(IV)}$$

When $\phi(x, y)$ is defined for $x, y \in F$, non-zero, is in the range $[-fmax_{F'}, fmax_{F'}]$, and no notification apart from **inexact** occurs (for **inexact**, consider the continuation value),

$$\left| \frac{\Phi(x,y) - \phi(x,y)}{\phi(x,y)} \right| \leqslant ulp_{F'}(1) = epsilon_{F'} \qquad \text{(V)}$$

When $\phi(x, y)$ and $\phi(x \cdot r_F^j, y \cdot r_F^k)$ are defined for $x, y \in F$ and $j, k \in \mathcal{Z}$, are in the range $[-fmax_{F'}, fmax_{F'}]$, and no notification apart from **inexact** occurs (for **inexact**, consider the continuation value), for some $n \in \mathcal{Z}$

$$\phi(x \cdot r_F^j, y \cdot r_F^k) = \phi(x, y) \cdot r_F^n \;\Rightarrow\; \Phi(x \cdot r_F^j, y \cdot r_F^k) = \Phi(x, y) \cdot r_F^n \qquad \text{(VI)}$$

Rules (I) through (VI) apply to the approximate operations of LIA-1. This is true also with the relaxations in Annex A.6 with one exception. Rule (III) may then fail for add_F and sub_F when the approximate addition function is not equal to the true sum (i.e., $add^*_F(u, x) \neq u + x$, or $add^*_F(v, y) \neq v + y$). However, the following weaker rules always hold:

$$\begin{array}{l} u \leqslant v \;\Rightarrow\; add_{F\to F'}(u, x) \leqslant add_{F\to F'}(v, x) \\ u \leqslant v \;\Rightarrow\; add_{F\to F'}(x, u) \leqslant add_{F\to F'}(x, v) \\ u \leqslant v \;\Rightarrow\; sub_{F\to F'}(u, x) \leqslant sub_{F\to F'}(v, x) \\ u \leqslant v \;\Rightarrow\; sub_{F\to F'}(x, u) \geqslant sub_{F\to F'}(x, v) \end{array}$$

Rules (I) through (VI) also apply to the exact operations, but then they do not say anything of interest.

Here are some identities that apply to specific operations, when no notification apart from **inexact** occurs:

$$add_{F\to F'}(x, y) = add_{F\to F'}(y, x)$$

$$mul_{F\to F'}(x, y) = mul_{F\to F'}(y, x)$$

$$sub_{F\to F'}(x, y) = neg_{F'}(sub_{F\to F'}(y, x))$$

$$add_{F\to F'}(neg_F(x), neg_F(y)) = neg_{F'}(add_{F\to F'}(x, y))$$

$$sub_{F\to F'}(neg_F(x), neg_F(y)) = neg_{F'}(sub_{F\to F'}(x, y))$$

$$mul_{F\to F'}(neg_F x), y) = mul_{F\to F'}(x, neg_F(y)) = neg_{F'}(mul_{F\to F'}(x, y))$$

$$div_{F\to F'}(neg_F(x), y) = div_{F\to F'}(x, neg_F(y)) = neg_{F'}(div_{F\to F'}(x, y))$$

For $x \neq 0$,

$$x \in F_N \;\Rightarrow\; exponent_F(x) \in [emin_F, emax_F]$$

$$x \in F_S \;\Rightarrow\; exponent_F(x) \in [emin_F - p_F + 1, emin_F - 1]$$

$$r_F^{exponent_F(x)-1} \in F$$

$$r_F^{exponent_F(x)-1} \leqslant |x| < r_F^{exponent_F(x)}$$

$$|fraction_F(x)| \in [1/r_F, 1[$$

$$scale_F(fraction_F(x), exponent_F(x)) = x$$

$scale_F(x, n)$ is exact $(= x \cdot r_F^n)$ if $x \cdot r_F^n$ is in one of the ranges $[-fmax_F, -fminN_F]$ or $[fminN_F, fmax_F]$ or is 0, or if $n \geqslant 0$ and $|x \cdot r_F^n| \leqslant fmax_F$.

For $x \neq 0$ and $y \neq 0$,

$x = \pm i \cdot ulp_F(x)$ for some integer i which satisfies

$$\begin{array}{ll} r_F^{p_F-1} \leqslant i < r_F^{p_F} & \text{if } x \in F_N \\ 1 \leqslant i < r_F^{p_F-1} & \text{if } x \in F_S \end{array}$$

$$exponent_F(x) = exponent_F(y) \;\Rightarrow\; ulp_F(x) = ulp_F(y) \qquad \text{if } x, y \in F_N$$

$$x \in F_N \;\Rightarrow\; ulp_F(x) = epsilon_F * r_F^{exponent_F(x)-1}$$

Note that if $denorm_F =$ **true**, ulp_F is defined on all floating point values. If $denorm_F =$ **false** (not conforming to LIA-1), ulp_F underflows catastrophically (returning 0) on all values less than $fminN_F/epsilon_F$, i.e., on all values for which $e_F(x) < emin_F + p_F - 1$.

For $|x| \geqslant 1$,

$$intpart_F(x) = trunc_F(x, e_F(x)) = trunc_F(x, exponent_F(x))$$

For any $x \in F$, when no notification occurs,

$$succ_F(pred_F(x)) = x$$

$$pred_F(succ_F(x)) = x$$

$$succ_F(-x) = -pred_F(x)$$

$$pred_F(-x) = -succ_F(x)$$

For positive $x \in F$, when no notification occurs,

$$succ_F(x) = x + ulp_F(x)$$

$$\begin{aligned} pred_F(x) &= x - ulp_F(x) && \text{if } x \text{ is not } r_F^n \text{ for any integer } n \geqslant emin_F \\ &= x - ulp_F(x)/r_F && \text{if } x \text{ is } r_F^n \text{ for some integer } n \geqslant emin_F \end{aligned}$$

$$ulp_F(x) \cdot r_F^{p_F} = r_F^{e_F(x)}$$

For any x and any integer $n > 0$, when no notification occurs,

$$r_F^{exponent_F(x)-1} \leqslant |trunc_F(x, n)| \leqslant |x|$$

$$\begin{aligned} round_F(x, n) &= trunc_F(x, n), \quad \text{or} \\ &= trunc_F(x, n) + signum_F(x) \cdot ulp_F(x) \cdot r_F^{p_F - n} \end{aligned}$$

C.5.2.9 Precision, accuracy, and error

LIA-1 uses the term *precision* to mean the number of radix r_F digits in the fraction of a floating point datatype. All floating point numbers of a given type are assumed to have the same precision. (This does not hold for variable precision floating point datatypes. However, LIA-1 does not cover variable precision floating point datatypes.) A subnormal number has the same number of radix r_F digits, but the presence of leading zeros in its fraction means that fewer of these digits are significant.

In general, numbers of a given datatype will not have the same accuracy. Most will contain combinations of errors which can arise from many sources.

a) The error introduced by a single atomic arithmetic operation.

b) The error introduced by approximations in mathematical constants, such as π, $1/3$, or $\sqrt{2}$, used as program constants.

c) The errors incurred in converting data between external format (decimal text) and internal format.

d) The error introduced by use of a numerical library routine.

e) The errors arising from limited resolution in measurements.

f) Two types of modelling errors:

   1) Approximations made in the formulation of a mathematical model for the application at hand.

   2) Conversion of the mathematical model into a computational model, including approximations imposed by the discrete nature of numerical calculations.

g) The maximum possible accumulation of such errors in a calculation.

h) The true accumulation of such errors in a calculation.

i) The final difference between the computed result and the mathematically accurate result.

The last item is the goal of error analysis. To obtain this final difference, it is necessary to understand the other eight items, some of which are discussed below.

C.5.2.9.1 LIA-1 and error

LIA-1 interprets the error in a single atomic arithmetic operation to mean the error introduced into the result by the operation, without regard to any error which may have been present in the input operands.

The rounding function introduced in 5.2.4 produces the only source of error contributed by arithmetic operations. If the results of an arithmetic operation are exactly representable, they must be returned without error. Otherwise, LIA-1 requires that the error in the result of a conforming operation be bounded in magnitude by one half ulp, and bounded in magnitude by one ulp for partial conformity.

Rounding that results in a subnormal number may result in a loss of significant digits. A subnormal result is always exact for an $add_{F \to F}$ or $sub_{F \to F}$ operation (same type for arguments and result), provided $denorm_F = \mathbf{true}$. Such subnormal results will not give rise to underflow notifications. However, a subnormal result for a $mul_{F \to F}$ or $div_{F \to F}$ operation (same type for arguments and result) usually is not exact, which introduces an error of at most one half ulp (or one ulp, if the relaxations of A.6 are allowed), provided $denorm_F = \mathbf{true}$. When there is loss of significant digits in producing a subnormal result, the relative error due to rounding exceeds that for rounding a 'normal' result. Hence accuracy of a subnormal result for a $mul_{F \to F}$ or $div_{F \to F}$ operation is usually lower than that for a 'normal' result. Such loss of accuracy is required to be accompanied with an underflow notification.

Note that the error in the result of an operation on exact input operands becomes an "inherited" error if and when this result appears as input to a subsequent operation. The interaction between the intrinsic error in an operation and the inherited errors present in the input operands is discussed below in C.5.2.9.3.

C.5.2.9.2 Empirical and modelling errors

Empirical errors arise from data taken from sensors of limited resolution, uncertainties in the values of physical constants, and so on. Such errors can be incorporated as initial errors in the relevant input parameters or constants.

Modelling errors arise from a sequence of approximations:

a) Formulation of the problem in terms of the laws and principles relevant to the application. The underlying theory may be incompletely formulated or understood.

b) Formulation of a mathematical model for the underlying theory. At this stage approximations may enter from neglect of effects expected to be small.

c) Conversion of the mathematical model into a computer model by replacing infinite series by a finite number of terms, transforming continuous into discrete processes (e.g. numerical integration), and so on.

Estimates of the modelling errors can be incorporated as additions to the computational errors discussed in the next section. The complete error model will determine whether the final accuracy of the output of the program is adequate for the purposes at hand.

Finally, comparison of the output of the computer model with observations may shed insight on the validity of the various approximations made.

C.5.2.9.3 Propagation of errors

Let each variable in a program be given by the sum of its true value (denoted with subscript t) and its error (denoted with subscript e). That is, the program variable x

$$x = x_t + x_e$$

consists of the "true" value plus the accumulated "error". Note that the values taken on by x are "machine numbers" in the set F, while x_t and x_e are mathematical quantities in $\mathcal{R}$.

The following example illustrates how to estimate the total error contributed by the combination of errors in the input operands and the intrinsic error in addition. First, the result of an LIA-1 operation on approximate data can be described as the sum of the result of the true operation on that data and the "rounding error", where

$$rounding\_error = computed\_value - true\_value$$

Next, the true operation on approximate data is rewritten in terms of true operations on true data and errors in the data. Finally, the magnitude of the error in the result can be estimated from the errors in the data and the rounding error.

Consider the result, z, of LIA-1 addition operation on x and y:

$$z = add_{F\to F}(x, y) = (x + y) + rounding\_error$$

where the true mathematical sum of x and y is

$$(x + y) = x_t + x_e + y_t + y_e = (x_t + y_t) + (x_e + y_e)$$

By definition, the "true" part of z is

$$z_t = x_t + y_t$$

so that

$$z = z_t + (x_e + y_e) + rounding\_error$$

Hence

$$z_e = (x_e + y_e) + rounding\_error$$

The rounding error is bounded in magnitude by $0.5 \cdot ulp_F(z)$ ($ulp_F(z)$ if the relaxation in A.6 is allowed) (this can be a slight overestimate at exponent boundaries). If bounds on x_e and y_e are also known, then a bound on z_e can be calculated for use in subsequent operations for which z is an input operand.

Although it is a lengthy and tedious process, an analysis of an entire program can be carried out from the first operation through the last. At each stage the worst case combination of signs and magnitudes in the errors must be assumed. Thus, it is likely that the estimates for the final errors will be unduly pessimistic because the signs of the various errors are usually unknown. Under some circumstances it is possible to obtain a realistic estimate of the true accumulation of error instead of the maximum possible accumulation, e.g. in sums of terms with known characteristics.

C.5.2.10 Extra precision

The use of a higher level of range and/or precision is a time-honoured way of eliminating overflow and underflow problems and providing "guard digits" for the intermediate calculations of a problem. In fact, one of the reasons that programming languages have more than one floating point type is to permit programmers to control the precision of calculations.

Clearly, programmers should be able to control the precision of calculations whenever the accuracy of their algorithms require it. Conversely, programmers should not be bothered with such details in those parts of their programs that are not precision sensitive.

Some programming language implementations calculate intermediate values inside expressions to a higher precision than is called for by either the input variables or the result variable. This "extended intermediate precision" strategy has the following advantages:

a) The result value may be closer to the mathematically correct result than if "normal" precision had been used.

b) The programmer is not bothered with explicitly calling for higher precision calculations.

However, there are also some disadvantages:

a) Since the use of extended precision varies with implementation, programs become less portable.

b) It is difficult to predict the results of calculations and comparisons, even when all floating point parameters and rounding functions are known.

c) It is impossible to rely on techniques that depend on the number of digits in working precision.

d) Programmers lose the advantage of extra precision if they cannot reliably store parts of a long, complicated expression in a temporary variable at the higher precision.

e) Programmers cannot exercise precise control when needed.

f) Programmers cannot trade off accuracy against performance.

Assuming that a programming language designer or implementor wants to provide extended intermediate precision in a way consistent with the LIA-1, how can it be done? Implementations must follow the following rules detailed in clause 8:

a) Each floating point type, even those that are only used in extended intermediate precision calculations, must be documented.

b) The translation of expressions into LIA-1 operations must be documented. This includes any implicit conversions to or from extended precision types occurring inside expressions.

This documentation allows programmers to predict what each implementation will do. To the extent that a programming language standard constrains what implementations can do in this area, the programmer will be able to make predictions across all implementations. In addition, the implementation should also provide the user some explicit controls (perhaps with compiler directives or other declarations) to prevent or enable this “silent” widening of precision.

C.5.3 Operations for conversion between numeric datatypes

LIA-1 covers conversions from an integer type to another integer type and to a floating point type, as well as between floating point types. These conversions are often between two datatypes conforming to this document. But, unlike for other operations, the source or target datatype for a conversion may be a datatype that does not conform to LIA. An example of such a conversions are conversions to and from strings. Indeed it is common for string formats to include fixed-point formats. LIA-1 does not cover fixed-point datatypes otherwise, but does so for conversions, specifically due to the fixed-point string formats.

String formats also need to cover special values. For $\mathbf{-0}$, strings like “−0”, “−0.0”, “−0.00e−4”, “$-0.00 \cdot 10^{-4}$” are among the possible representations. $\mathbf{+\infty}$ can, for instance, be represented by strings like “$+\infty$”, “∞”, “+infinity”, or “positiva oändligheten”. In ordinary string formats for numerals, the string “Hello world!” is an example of a signalling NaN.

LIA-1 does not specify any string formats, not even for the special values $\mathbf{-0}$, $\mathbf{+\infty}$, $\mathbf{-\infty}$, and quiet NaN. The strings used may depend on preference settings. For instance, one may use different notation for the decimal separator character (like period, comma, Arabic comma, ...), use superscript digits for exponents in scientific notation, or use Arabic digits or traditional Thai digits. String formats for numerical values, and if and how they may depend on preference settings, is also an issue for bindings or programming language specifications, not for this part of LIA.

If the value converted is greater than those representable in the target datatype, or less than those representable in the target datatype, even after rounding, then an overflow will result. E.g., if the target is a character string of at most 3 digits, and the target radix is 10, then an integer source value of 1000 will result in an overflow. As for other operations, if the notification handling is by recording in indicators, a suitable continuation value must be used.

Most language standards contain (partial) format specifications for conversion to and from strings, usually for a decimal representation.

LIA-1 requires, like C [15], that all floating point conversion operations be such that the error is at most 0.5 ulp. This is now also required by IEC 60559 (IEEE 754) for the round-to-nearest conversions.

C.5.4 Numerals as operations in a programming language

C.5.4.1 Numerals for integer datatypes

Negative values (except $minint_I$ if $minint_I = -maxint_I - 1$) can be obtained by using the negation operation (neg_I).

Integer numerals in radix 10 are normally available in programming languages. Other radices may also be available for integer numerals, and the radix used may be part of determining the target integer datatype. E.g., radix 10 may be for signed integer datatypes, and radix 8 or 16 may be for unsigned integer datatypes.

Syntaxes for numerals for different integer datatypes need not be different, nor need they be the same. This part does not further specify the format for integer numerals. That is an issue for bindings.

Overflow for integer numerals can be detected at "compile time", and warned about. Likewise can notifications about invalid, e.g. for infinitary or NaN numerals that cannot be converted to the target type, be detected at "compile time" and be warned about.

C.5.4.2 Numerals for floating point datatypes

If the numerals used as operations in a program, and numerals read from other sources use the same radix, then "internal" numerals and "external" numerals (strings) denoting the same value in $\mathcal{R}$ and converted to the same target datatype should be converted to the same value. Indeed, the requirement on such conversions to round to nearest implies this.

Negative values (including negative 0, $\mathbf{-0}$, if avaliable) can be obtained by using the negation operation (neg_F).

Radices other than 10 may also be available for floating point numerals.

Integer numerals may also be floating point numerals, i.e. their syntaxes need not be different. Nor need syntaxes for numerals for different floating point datatypes be different, nor need they be the same. This part does not specify the syntax for numerals. That is an issue for bindings or programming language specifications.

Overflow or underflow for floating point numerals can be detected at "compile time", and warned about. Likewise can notifications about **infinitary** or **invalid**, e.g. for infinitary or NaN numerals that cannot be converted to the target type, be detected at "compile time" and be warned about.

C.6 Notification

The essential goal of the notification process is that it should be possible for a program (efficiently and at reasonable convenience to the programmer) to find out about relevant problems that have occurred when evaluating a set of values. For reasons of efficiency, it may not be possible to tie the reported problems to individual values, just a set of values. However, it must be possible to determine which set of values the reported problems are in relation to.

Given these notifications, a program can take action to recompute the result (using higher precision, or some other way of trying to avoid the problem), or report the problem to the user of the program in some suitable way.

C.6.1 Model handling of notifications

In the mathematical framework of LIA, every value is accompanied by a set of exceptional values, or actually, set of indicators. And in a floating point processor pipeline, this is often the actual case as well. At a higher level, it is usually not very efficient to let every value be individually accompanied by a set of indicators.

C.6.2 Notification alternatives

LIA-1 provides a choice of notification mechanisms to fit the requirements of various programming languages. The first alternative (recording in indicators) provides a standard notification handling mechanism for all programming languages. The second alternative (alteration of control flow) essentially says "if a programming language already provides an exception handling mechanism for some kinds of notification, it may be used for some of the arithmetic notifications too". The recording in indicators mechanism must be provided, and should be the default handling.

The third alternative (termination of program with message) is provided for use when the programmer has not (yet) programmed any exception handling code for the alteration of control flow alternative. It should be noted that this may terminate just an affected thread, while the rest of the program may continue.

Implementations are encouraged to provide additional mechanisms which would be useful for debugging. For example, pausing and dropping into a debugger, or continuing execution while writing to a log file.

In order to provide the full advantage of these notification capabilities, information describing the nature of the reason for the notification should be complete and available as close in time to the occurrence of the violation as possible.

C.6.2.1 Notification by recording in indicators

This alternative gives a programmer the primitives needed to obtain exception handling capabilities in cases where the programming language does not provide such a mechanism directly. An implementation of this alternative for notification should not need extensions to most programming languages. The status of the indicators is maintained by the system. The operations for testing and manipulating the indicators can be implemented as a library of callable routines.

This alternative can be implemented on any system with an "interrupt" capability, and on some without such a capability.

This alternative can be implemented on an IEEE system by making use of the required status flags. The mapping between the IEEE status flags and the LIA-1 indicators is as follows:

IEEE flag	LIA notification (can be recorded in indicator)
invalid	**invalid**
overflow	**overflow**
underflow	**underflow**
division by zero	**infinitary**
inexact	**inexact**
(no counterpart)	**absolute_precision_underflow** (LIA-2 and LIA-3)

Non-IEEE implementations are unlikely to detect inexactness of floating point results.

For a zero divisor, IEEE specifies an **invalid** exception if the dividend is zero, and a **division by zero** (**infinitary** in LIA) otherwise. Note that the continuation value also varies. Other architectures are not necessarily capable of making this distinction. In order to provide a reasonable mapping for an exception associated with a zero divisor, LIA allows that a binding may map both notification types to the same actual notification, like **invalid**.

Different notification types need not be handled the same. E.g. **inexact** and **underflow** should be handled by recording in indicators, or even be ignored if a binding so specifies, regardless of how other notifications are handled.

A program should check the set of indicators and handle indicators that are set in an appropriate way. Most programs are likely to ignore if the indicator for **inexact** is set, but other indicators should be dealt with by either recalculation in a different way, or by indicating the problem to the user in a suitable way, like an error message. For some computations it may be appropriate to ignore even notifications about **invalid**, for instance when using max/min operations that skip NaNs.

The mechanism of recording in indicators is general enough to be applied to a broad range of phenomena by simply extending the value set E to include indicators for other types of conditions. However, in order to maintain portability across implementations, such extensions should be made in conformity with other standards, such as language standards.

Notification indicators may be a form of thread global variable, but can be more local (but not more global). A single thread of computation must have at least one set of these indicators local to the thread, not interfering with other threads. However, care should be taken in designing systems with multiple threads or “interrupts” so that

a) logically asynchronous computations do not interfere with each other’s indicators, and

b) notifications do not get lost when threads are rejoined (unless the whole computation of the thread is ignored) or data exchanged between threads.

Similarly, any kind of evaluation “modes”, like rounding mode, or notification handling “modes” may be thread global modes, but can be more local (e.g. static per operation), but never more global. So the mode settings and changes in different threads do not interfere. The modes may be inherited from the logical parent of a thread, or be default if there is no logical parent to the thread. Note that LIA does not use a rounding mode model, different roundings have different operations. IEEE 754 has a partial rounding mode model, some operations are sensitive to the set rounding mode, while for other operations the rounding is fixed for the operation (as in LIA).

The details on how to do this is part of the design of the programming language, threads system, or hardware, and is not within the scope of LIA-1. Still, these details should be documented in a binding.

C.6.2.2 Notification by alteration of control flow

This alternative requires the programmer to provide application specific code which decides whether the computation should proceed, and if so how it should proceed. This alternative places the responsibility for the decision to proceed with the programmer who is presumed to have the best understanding of the needs of the application.

Ada and PL/I are examples of standard languages which include syntax that allows the programmer to describe this type of alteration of control flow.

Note, however, that a programmer may not have provided code for all trouble-spots in the program. This implies that program termination (or thread termination) must be an available alternative.

Designers of programming languages and binding standards should keep in mind the basic principle that a program should not be allowed to take significant irreversible action (for example, printing out apparently accurate results, or even terminating "normally") based on erroneous arithmetic computations.

C.6.2.3 Notification by termination with message

This alternative results in the termination of the program following a notification. It is intended mainly for use when a programmer has failed to exploit one of the other alternatives provided.

C.6.3 Delays in notification

Many modern floating point implementations are pipelined, or otherwise execute instructions in parallel. This can lead to an apparent delay in reporting violations, since an overflow in a multiply operation might be detected after a subsequent, but faster, add operation completes. The provisions for delayed notification are designed to accommodate these implementations.

Parallel implementations may also not be able to distinguish a single overflow from two or more "almost simultaneous" overflows. Hence, some merging of notifications is permitted.

Imprecise interrupts (where the offending instruction cannot be identified) can be accommodated as notification delays. Such interrupts may also result in not being able to report the kind of violation that occurred, or to report the order in which two or more violations occurred.

In general the longer the notification is delayed the greater the risk to the continued execution of the program.

C.6.4 User selection of alternative for notification

On some machine architectures, the notification alternative selected may influence code generation. In particular, the optimal code that can be generated for change of control flow (6.2.2) may differ substantially from the optimal code for recording in indicators (6.2.1). Because of this, it is unwise for a language or binding standard to require the ability to switch between notification alternatives during execution. Compile time selection should be sufficient.

An implementation can provide separate selection for each kind of notification (**overflow**, **underflow**, etc).

If a system had a mode of operation in which exceptions were totally ignored, then for this mode, the system would not conform to ISO/IEC 10967. However, modes of operation that ignore exceptions may have some uses, particularly if they are otherwise LIA-1 conforming. For example, a user may find it desirable to verify and debug a program's behaviour in a fully LIA-1 conforming mode (exception checking on), and then run the resulting "trusted" program with exception checking off.

In any case, it is essential for an implementation to provide documentation on how to select among the various LIA-1 conforming notification alternatives provided.

C.7 Relationship with language standards

Language standards vary in the degree to which the underlying datatypes are specified. For example, Pascal [25] merely gives the largest integer value ($maxint_I$), while Ada [11] gives a large number of attributes of the underlying integer and floating point types. LIA-1 provides a language independent framework for giving the same level of detail that Ada requires, specific to a particular implementation.

LIA-1 gives the meaning of individual operations on numeric values of particular type. It does not specify the semantics of expressions, since expressions are sequences of operations which could be mapped into individual operations in more than one way. LIA-1 does require documentation of the range of possible mappings.

The essential requirement is to document the semantics of expressions well enough so that a reasonable error analysis can be done. There is no requirement to document the specific optimisation technology in use.

An implementation might conform to the letter of LIA-1, but still violate its "spirit" – the principles behind LIA-1 – by providing, for example, a *sin* function that returned values greater than 1 or that was highly inaccurate for input values greater than one cycle. LIA-2 takes care of this particular example. Beyond this, implementors are encouraged to provide numerical facilities that

a) are highly accurate,

b) obey useful identities like those in C.5.2.0.1 or C.5.2.8,

c) notify the user whenever the mathematically correct result would be out of range, not accurately representable, or undefined,

d) are defined on as wide a range of input values as is consistent with the three items above.

LIA-1 does not cover programming language issues such as type errors or the effects of uninitialised variables. Implementors are encouraged to catch such errors – at compile time whenever possible, at run time if necessary. Uncaught programming errors of this kind can produce the very unpredictable and false results that this standard was designed to avoid.

A list of the information that every implementation of LIA-1 must document is given in clause 8. Some of this information, like the value of $emax_F$ for a particular programming language floating point type, may vary from implementation to implementation. However, due to the success of IEEE 754, this will be much less of an issue. Other information, like the syntax for accessing the value of $emax_F$, should be the same for all implementations of a particular programming language. See annex D for information on how this might be done.

To maximize the portability of programs, most of the information listed in clause 8 should be standardized for a given language – either by inclusion in the language standard itself, or by a language specific binding standard.

To further promote portability, the numeric datatype parameters should be standardised per programming language and datatype.

The allowed translations of expressions into combinations of LIA operations should allow reasonable flexibility for compiler optimisation. The programming language standard must determine what is reasonable. In particular, languages intended for the careful expression of numeric algorithms are urged to provide ways for programmers to control order of evaluation and intermediate precision within expressions. Note that programmers may wish to distinguish between such "controlled" evaluation of some expressions and "don't care" evaluation of others.

Developers of language standards or binding standards may find it convenient to reference LIA-1. For example, the specification method and the helper functions may prove useful in defining additional arithmetic operations.

C.8 Documentation requirements

To make good use of an implementation of this standard, programmers need to know not only that the implementation conforms, but *how* the implementation conforms. Clause 8 requires implementations to document the binding between LIA-1 types and operations and the total arithmetic environment provided by the implementation.

An example conformity statement (for a Fortran implementation) is given in annex E.

It is expected that an implementation will meet part of its documentation requirements by incorporation of the relevant language standard. However, there will be aspects of the implementation that the language standard does not specify in the required detail, and the implementation needs to document those details. For example, the language standard may specify a range of allowed parameter values, but the implementation must document the value actually used. The combination of the language standard and the implementation documentation together should meet all the requirements in clause 8.

Most of the documentation required can be provided easily. The only difficulties might be in defining helper functions like add^*_F (for partially conforming implementations, see annex A), or in specifying the translation of arithmetic expressions into combinations of LIA-1 operations.

Compilers often "optimise" code as part of the compilation process. Popular optimisations include moving code to less frequently executed spots, eliminating common subexpressions, and reduction in strength (replacing expensive operations with cheaper ones).

Compilers are always free to alter code in ways that preserve the semantics (the values computed and the notifications generated). However, when a code transformation may change the semantics of an expression, this must be documented by listing the alternative combinations of operations that might be generated. (There is no need to include semantically equivalent alternatives in this list.) This includes evaluations that are done at compile time instead of at runtime. For instance evaluating $quot_I(minint_I, -1)$ at compile time (as a constant expression) may yield an overflow which might not be visible at runtime (having replaced the expression with the continuation value alone, not triggering the overflow notification at runtime), if this expression is allowed to compile at all.

Annex D
(informative)

Example bindings for specific languages

This annex describes how a computing system can simultaneously conform to a language standard and to LIA-1. It contains suggestions for binding the "abstract" operations specified in LIA-1 to concrete language syntax.

Portability of programs can be improved if two conforming LIA-1 systems using the same language agree in the manner with which they adhere to LIA-1. For instance, LIA-1 requires that the derived constant $epsilon_F$ be provided, but if one system provides it by means of the identifier `EPS` and another by the identifier `EPSILON`, 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
Common Lisp

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-1.

In this annex, the datatypes, parameters, constants, operations, and exception behaviour of each language are examined to see how closely they fit the requirements of LIA-1. Where parameters, constants, or operations are not provided by the language, names and syntax are suggested. (Already provided syntax is marked with a $\star$.) Substantial additional suggestions to language developers are presented in C.7, but a few general suggestions are reiterated below.

This annex describes only the language-level support for LIA-1. An implementation that wishes to conform must ensure that the underlying hardware and software is also configured to conform to LIA-1 requirements.

A complete binding for LIA-1 will include a binding for IEC 60559. Such a joint LIA-1/IEC 60559 binding should be developed as a single binding standard. To avoid conflict with ongoing development, only LIA-1 specific portions of such a binding are presented in this annex.

Most language standards permit an implementation to provide, by some means, the parameters, constants and operations required by LIA-1 that are not already part of the language. The method for accessing these additional constants and operations depends on the implementation and language, and is not specified in LIA-1. 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.

A few parameters are completely determined by the language definition, e.g. whether the integer type is bounded. Such parameters have the same value in every implementation of the language, and therefore need not be provided as a run-time parameter.

During the development of standard language bindings, each language community should take care to minimise the impact of any newly introduced names on existing programs. Techniques such as "modules" or name prefixing may be suitable depending on the conventions of that language community.

LIA-1 treats only single operations on operands of a single datatype (in some cases with a different target type), but nearly all computational languages permit expressions that contain multiple operations involving operands of mixed types. The rules of the language specify how the operations and operands in an expression are mapped into the primitive operations described by LIA-1. In principle, the mapping could be completely specified in the language standard. However, the translator often has the freedom to depart from this precise specification: to reorder computations, widen datatypes, short-circuit evaluations, and perform other optimisations that yield "mathematically equivalent" results but remove the computation even further from the image presented by the programmer.

We suggest that each language standard committee require implementations to provide a means for the user to control, in a portable way, the order of evaluation of arithmetic expressions.

Some numerical analysts assert that user control of the precision of intermediate computations is desirable. We suggest that language standard committee consider requirements which would make such user control available in a portable way. (See C.5.2.10.)

Most language standards do not constrain the accuracy of floating point operations, or specify the subsequent behaviour after a serious arithmetic violation occurs.

We suggest that each language standard committee require that the arithmetic operations provided in the language satisfy LIA-1 requirements for accuracy and notification.

We also suggest that each language standard committee define a way of handling exceptions within the language, e.g. to allow the user to control the form of notification, and possibly to "fix up" the error and continue execution. The binding of the exception handling within the language syntax must also be specified.

In the event that there is a conflict between the requirements of the language standard and the requirements of LIA-1, the language binding standard should clearly identify the conflict and state its resolution of the conflict.

D.1 Ada

The programming language Ada is defined by ISO/IEC 8652:1995, *Information Technology – Programming Languages – Ada* [11].

An implementation should follow all the requirements of LIA-1 unless otherwise specified by this (example, and partial) language binding.

The operations or parameters marked "†" are not part of the language standard and must be provided by an implementation that wishes to conform to LIA-1. For each of the marked items a suggested identifier is provided. The additional facilities can be provided by means of an additional package, denoted by `LIA`.

The Ada datatype `Boolean` corresponds to the LIA-1 datatype **Boolean**.

Ada has a predefined signed integer type named `Integer`. It is a bounded integer type. Ada also has a predefined type `Natural`, which is an unsigned type with the same upper limit as `Integer`. `Natural` is a non-modulo (see below) datatype. A programmer can declare other subtypes of *root_integer* (an "anonymous" integer type). Those declared as "modulo" go from zero to a given upper limit. They can conform to LIA-1 integer datatypes in the value set. However, all arithmetic operations resulting in a "modulo" integer datatype (as determined by the Ada type system) have an implicit modulo with one plus the max value of the resulting datatype (see add_wrap_I, sub_wrap_I, and mul_wrap_I of LIA-2). Non-modulo integer subtypes may be signed. But they need not fulfill the requirements of LIA-1 on the minimum and maximum value. There may be other predefined signed integer types: `Short_Integer`, `Long_Integer`, `Short_Short_Integer`, `Long_Long_Integer`, etc. The notation INT is used to stand for the name of any one of these datatypes in what follows.

Ada exceptions do not distinguish between **infinitary**, **overflow**, and **invalid**, and the exception `Constraint_Error` is used by default for all three of these notifications. Integer operations that, for a non-modulo integer target type, mathematically result in a value outside the range $[min, max]$, i.e. results in an **overflow** notification, are required to raise the `Constraint_Error` exception.

The LIA-1 parameters for an integer datatype can be accessed by the following syntax:

$maxint_I$	INT`'Last`	⋆
$minint_I$	INT`'First`	⋆

The parameter $bounded_I$ is always **true**, and the parameter $hasinf_I$ is always **false**, and they need therefore not be provided to programs as named parameters. The parameter $modulo_I$ (see annex A) is always **false** for non-modulo integer datatypes, and always **true** for modulo integer datatypes (declared via the `modulo` keyword), and need not be provided for programs as a named parameter.

The LIA-1 integer operations are listed below, along with the syntax used to invoke them:

$eq_I(x, y)$	x `=` y	⋆
$neq_I(x, y)$	x `/=` y	⋆
$lss_I(x, y)$	x `<` y	⋆
$leq_I(x, y)$	x `<=` y	⋆
$gtr_I(x, y)$	x `>` y	⋆
$geq_I(x, y)$	x `>=` y	⋆
$neg_I(x)$	`-`x	⋆ (if $modulo_I =$ **false**)
$add_I(x, y)$	x `+` y	⋆ (if $modulo_I =$ **false**)
$add\_wrap_I(x, y)$	x `+` y	⋆ (if $modulo_I =$ **true**)
$sub_I(x, y)$	x `-` y	⋆ (if $modulo_I =$ **false**)
$sub\_wrap_I(x, y)$	x `-` y	⋆ (if $modulo_I =$ **true**)
$mul_I(x, y)$	x `*` y	⋆ (if $modulo_I =$ **false**)
$mul\_wrap_I(x, y)$	x `*` y	⋆ (if $modulo_I =$ **true**)
$abs_I(x)$	`abs` x	⋆
$signum_I(x)$	`Signum(`x`)`	†

$quot_I(x, y)$	`Quotient(`x`, `y`)`	†
$mod_I(x, y)$	x `mod` y	⋆

where x and y are expressions of type INT.

An implementation that wishes to conform to LIA-1 must provide all the LIA-1 integer operations for all the integer datatypes for which LIA-1 conformity is claimed.

Ada has a predefined floating point datatype named `Float`. There may be other predefined floating point types: `Short_Float`, `Long_Float`, `Short_Short_Float`, `Long_Long_Float`, etc. The notation FLT are used to stand for the name of any one of these datatypes in what follows.

Ada exceptions do not distinguish between **infinitary**, **overflow**, and **invalid**, and the exception `Constraint_Error` is used by default for **infinitary** and **invalid**. If FLT`'Machine_Overflows` is **true**, then floating point operations that mathematically result in a value outside the range $[min, max]$ (where min is, if conforming to LIA-1, the negative of max), i.e. results in an **overflow** notification, are required to raising the `Constraint_Error` exception. If FLT`'Machine_Overflows` is **false**, no exception is raised for **overflow** (but the notification should be recorded in an indicator).

The LIA-1 parameters and derived constants for a floating point datatype can be accessed by the following syntax:

r_F	FLT`'Machine_Radix`	⋆
p_F	FLT`'Machine_Mantissa`	⋆
$emax_F$	FLT`'Machine_Emax`	⋆
$emin_F$	FLT`'Machine_Emin`	⋆
$denorm_F$	FLT`'Denorm`	⋆
iec_60559_F	FLT`'IEC60559`	†
$hasnegzero_F$	FLT`'Signed_Zeroes`	⋆ (not LIA-1)
$fmax_F$	FLT`'Safe_Last`	⋆
$-fmax_F$	FLT`'Safe_First`	⋆
$fminN_F$	FLT`'Model_Small`	⋆
$fmin_F$	FLT`'Model_Smallest`	†
$epsilon_F$	FLT`'Model_Epsilon`	⋆
rnd_error_F	FLT`'Rnd_Error`	† (partial conf.)
rnd_style_F	FLT`'Rnd_Style`	† (partial conf.)

The value returned by FLT`'Rnd_Style` are from the enumeration type `Rnd_Styles`. Each enumeration literal corresponds as follows to an LIA-1 rounding style value:

nearesttiestoeven	`NearestTiesToEven`	†
nearest	`Nearest`	†
truncate	`Truncate`	†
other	`Other`	†

As currently written, Ada formally only allows truncate and nearest-with-ties-away (which is given by FLT`'Rounds`), not nearest-ties-to-even. This is not fully conforming to LIA-1, only partially conforming. Note, however, that nearest-with-ties-away is *not* readily available for IEEE 754 (IEC 60559) binary floating point datatypes.

There is no standard way of setting rounding mode (as per IEEE 754) in Ada. Note also that LIA recommends using separate operations for separate roundings, rather than using dynamic rounding modes. Separate operations are in this case more reliable and less error prone.

The LIA-1 floating point operations are listed below, along with the syntax used to invoke them:

$eq_F(x,y)$	`x = y`	⋆
$neq_F(x,y)$	`x /= y`	⋆
$lss_F(x,y)$	`x < y`	⋆
$leq_F(x,y)$	`x <= y`	⋆
$gtr_F(x,y)$	`x > y`	⋆
$geq_F(x,y)$	`x >= y`	⋆
$isnegzero_F(x)$	`isNegZero(x)`	†
$istiny_F(x)$	`isTiny(x)`	†
$isnan_F(x)$	`isNaN(x)`	†
$isnan_F(x)$	`x /= x`	⋆
$issignan_F(x)$	`isSigNaN(x)`	†
$neg_F(x)$	`-x`	⋆
$add_{F \to F'}(x,y)$	`x + y`	⋆
$add^{\uparrow}_{F \to F'}(x,y)$	`x +> y`	†
$add^{\downarrow}_{F \to F'}(x,y)$	`x <+ y`	†
$sub_{F \to F'}(x,y)$	`x - y`	⋆
$sub^{\uparrow}_{F \to F'}(x,y)$	`x -> y`	†
$sub^{\downarrow}_{F \to F'}(x,y)$	`x <- y`	†
$mul_{F \to F'}(x,y)$	`x * y`	⋆
$mul^{\uparrow}_{F \to F'}(x,y)$	`x *> y`	†
$mul^{\downarrow}_{F \to F'}(x,y)$	`x <* y`	†
$div_{F \to F'}(x,y)$	`x / y`	⋆
$div^{\uparrow}_{F \to F'}(x,y)$	`x /> y`	†
$div^{\downarrow}_{F \to F'}(x,y)$	`x </ y`	†
$abs_F(x)$	`abs x`	⋆
$signum_F(x)$	`Signum(x)`	†
$residue_F(x,y)$	`FLT'Remainder(x, y)`	⋆
$sqrt_{F \to F'}(x)$	`Sqrt(x)`	⋆
$sqrt^{\uparrow}_{F \to F'}(x)$	`SqrtUp(x)`	†
$sqrt^{\downarrow}_{F \to F'}(x)$	`SqrtDwn(x)`	†
$exponent_{F \to I}(x)$	`FLT'Exponent(x)`	⋆ (dev.: 0 if $eq_F(x, 0)$)
$fraction_F(x)$	`FLT'Fraction(x)`	⋆
$scale_{F,I}(x,n)$	`FLT'Scaling(x, n)`	⋆
$succ_F(x)$	`FLT'Adjacent(x, FLT'Safe_Last)`	⋆ (dev. at $fmax_F$)
$pred_F(x)$	`FLT'Adjacent(x, FLT'Safe_First)`	⋆ (dev. at $-fmax_F$)
$ulp_F(x)$	`FLT'Unit_Last_Place(x)`	†
$intpart_F(x)$	`FLT'Truncation(x)`	⋆

$fractpart_F(x)$	x - *FLT*'Truncation(x)	⋆
$trunc_{F,I}(x, n)$	*FLT*'Leading_Part(x, n)	⋆ (**invalid** for $n \leqslant 0$)
$round_{F,I}(x, n)$	*FLT*'Round_Places(x, n)	†

where x and y are expressions of type FLT and n is an expression of type INT.

An implementation that wishes to conform to LIA-1 must provide the LIA-1 floating point operations for all the floating point datatypes for which LIA-1 conformity is claimed.

Arithmetic value conversions in Ada are always explicit and usually use the destination datatype name as the name of the conversion function, except when converting to/from string formats.

$convert_{I \to I'}(x)$	*INT2*(x)	⋆
$convert_{I'' \to I}(s)$	Get(s, n, w);	⋆
$convert_{I'' \to I}(f)$	Get(f?, n, w?);	⋆
$convert_{I \to I''}(x)$	Put(s, x, *base*?);	⋆
$convert_{I \to I''}(x)$	Put(h?, x, w?, *base*?);	⋆
$floor_{F \to I}(y)$	*INT*(*FLT*'Floor(y))	⋆
$rounding_{F \to I}(y)$	*INT*(*FLT*'Unbiased_Rounding(y))	⋆
$ceiling_{F \to I}(y)$	*INT*(*FLT*'Ceiling(y))	⋆
$convert_{I \to F}(x)$	*FLT*(x)	⋆
$convert^{\uparrow}_{I \to F}(x)$	*FLT*'Up(x)	†
$convert^{\downarrow}_{I \to F}(x)$	*FLT*'Dwn(x)	†
$convert_{F \to F'}(y)$	*FLT2*(y)	⋆
$convert^{\uparrow}_{F \to F'}(x)$	*FLT2*'Up(x)	†
$convert^{\downarrow}_{F \to F'}(x)$	*FLT2*'Dwn(x)	†
$convert_{F'' \to F}(s)$	Get(s, n, w?);	⋆
$convert^{\uparrow}_{F'' \to F}(s)$	GetUp(s, n, w?);	†
$convert^{\downarrow}_{F'' \to F}(s)$	GetDwn(s, n, w?);	†
$convert_{F'' \to F}(f)$	Get(f?, n, w?);	⋆
$convert^{\uparrow}_{F'' \to F}(f)$	GetUp(f?, n, w?);	†
$convert^{\downarrow}_{F'' \to F}(f)$	GetDwn(f?, n, w?);	†
$convert_{F \to F''}(y)$	Put(s, y,Aft=>a?,Exp=>e?);	⋆
$convert^{\uparrow}_{F \to F''}(y)$	PutUp(s, y,Aft=>a?,Exp=>e?);	†
$convert^{\downarrow}_{F \to F''}(y)$	PutDwn(s, y,Aft=>a?,Exp=>e?);	†
$convert_{F \to F''}(y)$	Put(h?, y,Fore=>i?,Aft=>a?,Exp=>e?);	⋆
$convert^{\uparrow}_{F \to F''}(y)$	PutUp(h?, y,Fore=>i?,Aft=>a?,Exp=>e?);	†
$convert^{\downarrow}_{F \to F''}(y)$	PutDwn(h?, y,Fore=>i?,Aft=>a?,Exp=>e?);	†
$convert_{D \to F}(z)$	*FLT*(z)	⋆
$convert^{\uparrow}_{D \to F}(x)$	*FLT*'Up(x)	†
$convert^{\downarrow}_{D \to F}(x)$	*FLT*'Dwn(x)	†
$convert_{D' \to F}(s)$	Get(s, n, w?);	⋆
$convert^{\uparrow}_{D' \to F}(s)$	GetUp(s, n, w?);	†
$convert^{\downarrow}_{D' \to F}(s)$	GetDwn(s, n, w?);	†
$convert_{D' \to F}(f)$	Get(f?, n, w?);	⋆

$convert^{\uparrow}_{D'\rightarrow F}(f)$	`GetUp(`f`?, `n`, `w`?);`	†
$convert^{\downarrow}_{D'\rightarrow F}(f)$	`GetDwn(`f`?, `n`, `w`?);`	†
$convert_{F\rightarrow D}(y)$	*FXD*`(`y`)`	⋆
$convert^{\uparrow}_{F\rightarrow D}(x)$	*FXD*`'Up(`x`)`	†
$convert^{\downarrow}_{F\rightarrow D}(x)$	*FXD*`'Dwn(`x`)`	†
$convert_{F\rightarrow D'}(y)$	`Put(`s`, `y`,Aft=>`a`?,Exp=>0);`	⋆
$convert^{\uparrow}_{F\rightarrow D'}(y)$	`PutUp(`s`, `y`,Aft=>`a`?,Exp=>0);`	†
$convert^{\downarrow}_{F\rightarrow D'}(y)$	`PutDwn(`s`, `y`,Aft=>`a`?,Exp=>0);`	†
$convert_{F\rightarrow D'}(y)$	`Put(`h`?, `y`,Fore=>`i`?,Aft=>`a`?,Exp=>0);`	⋆
$convert^{\uparrow}_{F\rightarrow D'}(y)$	`PutUp(`h`?, `y`,Fore=>`i`?,Aft=>`a`?,Exp=>0);`	†
$convert^{\downarrow}_{F\rightarrow D'}(y)$	`PutDwn(`h`?, `y`,Fore=>`i`?,Aft=>`a`?,Exp=>0);`	†

where x is an expression of type *INT*, y is an expression of type *FLT*, and z is an expression of type *FXD*, where *FXD* is a fixed point type. *INT2* is the integer datatype that corresponds to I'. *FLT2* is the floating point datatype that corresponds to F'. A ? above indicates that the parameter is optional. f is an opened input text file (default is the default input file). h is an opened output text file (default is the default output file). s is of type `String` or `Wide_String`. For `Get` of a floating point or fixed point numeral, the base is indicated in the numeral (default 10). For `Put` of a floating point or fixed point numeral, only base 10 is required to be supported. For details on `Get` and `Put`, see clause A.10.8 Input-Output for Integer Types, A.10.9 Input-Output for Real Types, and A.11 Wide Text Input-Output, of ISO/IEC 8652:1995. *base*, n, w, i, a, and e are expressions for non-negative integers. e is greater than 0. *base* is greater than 1.

Ada provides non-negative numerals for all its integer and floating point types. The default base is 10, but any base from 2 to 16 can be used for a numeral. There is no differentiation between the numerals for different floating point types, nor between numerals for different integer types, but integer numerals (without a point) cannot be used for floating point types, and 'real' numerals (with a point) cannot be used for integer types. Integer numerals can have an exponent part though. Integer numerals are of the "anonymous" type *universal_integer*, and real numerals are of the "anonymous" type *universal_real*. The details are not repeated in this example binding, see ISO/IEC 8652:1995, clause 2.4 Numeric Literals, clause 3.5.4 Integer Types, and clause 3.5.6 Real Types.

The Ada standard does not specify any numerals for infinities and NaNs. The following syntax is suggested:

$+\infty$	*FLT*`'Infinity`	†
qNaN	*FLT*`'NaN`	†
sNaN	*FLT*`'NaNSignalling`	†

as well as string formats for reading and writing these values as character strings.

Ada has a notion of 'exception' that implies a non-returnable, but catchable, change of control flow. Ada uses its exception mechanism as its default means of notification for **overflow**, **infinitary**, and **invalid**. Ada uses the exception `Constraint_Error` for **infinitary** and **overflow** notifications, and the exceptions `Numerics.Argument_Error`, `IO_Exceptions.Data_Error`, and `IO_Exceptions.End_Error` for **invalid** notifications. **inexact** and **underflow** does not cause any exception in Ada, and the continuation value is used directly, since an Ada exception is inappropriate for these notifications. However, for LIA conformity, numeric notifications that do not cause Ada exceptions must be recorded in indicators per Ada task.

An implementation that wishes to follow LIA must provide recording in indicators as an alternative means of handling numeric notifications also for the notifications where the Ada standard requires alternation of control flow (Ada exceptions). (See 6.2.1.) Recording of indicators is the LIA preferred means of handling numeric notifications. In this suggested binding non-negative integer values in the datatype `Natural`, are used to represent values in *Ind*. The datatype *Ind* is identified with the datatype `Natural`. The values representing individual indicators are distinct non-negative powers of two. Indicators can be accessed by the following syntax:

inexact	`lia_inexact`	†
underflow	`lia_undeflow`	†
overflow	`lia_overflow`	†
infinitary	`lia_infinitary`	†
invalid	`lia_invalid`	†
absolute_precision_underflow		
	`lia_density_too_sparse`	† (LIA-2, -3)
union of all indicators	`lia_all_indicators`	†

The empty set can be denoted by `0`. Other indicator subsets can be named by combining individual indicators using bit-wise or, or just addition, or by subtracting from `lia_all_indicators`.

The indicator interrogation and manipulation operations are listed below, along with the syntax used to invoke them:

$clear\_indicators(C, S)$	`Clear_Indicators(`S`)`	†
$set\_indicators(C, S)$	`Set_Indicators(`S`)`	†
$current\_indicators(C)$	`Current_Indicators()`	†
$test\_indicators(C, S)$	`Test_Indicators(`S`)`	†

where S is an expression compatible with the datatype `Natural`. C is the Ada task the call to the Ada function is made in.

It is vital that indicators are managed separately for separate Ada tasks (as required by LIA). Likewise that dynamically set rounding modes (which LIA-1 does *not* recommend) are also managed separately for separate tasks in such an environment.

In order not to lose notification indicators within an Ada program when the computation is divided into several Ada tasks (threads), any in-parameter for a rendezvous must set in the accepting task (when the call is accepted) the indicators that are set in the caller, and any out-parameter will set in the caller (when the rendezvous finishes) the indicators that are then set in the accepting task.

D.2 C

The programming language C is defined by ISO/IEC 9899:1999, *Information technology – Programming languages – C* [15]. Some additions relevant for LIA are made in the technical report ISO/IEC TR 24732:2009, *Information technology – Programming languages, their environments and system software interfaces – Extension for the programming language C to support decimal floating-point arithmetic* [16].

An implementation should follow all the requirements of LIA-1 unless otherwise specified by this (example, and partial) language binding.

The operations or parameters marked "†" are not part of the language and must be provided by an implementation that wishes to conform to LIA-1. For each of the marked items a suggested identifier is provided. An implementation that wishes to conform to LIA-1 must supply declarations of these items in a header `<lia1.h>`. Integer valued parameters and derived constants can be used in preprocessor expressions.

The LIA-1 datatype **Boolean** is implemented as the C datatype `_bool` or in the C datatype `int` (**false** = 0 and any other value (usually 1) represents **true**).

C names several integer datatypes: `(signed) int`, `(signed) long (int)`, `(signed) long long (int)`, `unsigned (int)`, `unsigned long (int)`, and `unsigned long long (int)`. The here parenthesised part(s) of a name may be omitted when using the name in programs. Signed integer datatypes use 2's complement for representation for negative values. The notation INT is used to stand for the name of any one of these datatypes in what follows.

The conformity to LIA of `short int` and `char` (signed or unsigned), and similar "short" integer types are not relevant since values of these types are promoted to `int` (signed or unsigned as appropriate) before arithmetic computations are done.

However, the basic integer datatypes, listed above, have portability issues. They may have different limits in different implementations. Therefore, the C standard specifies a number of additional integer datatypes, defined for programs in the headers `<stdint.h>` and `<stddef.h>`. Similar portable integer datatypes have been defined in portable libraries. They are aliased, by typedefs, to the basic integer datatypes, but the aliases are made in an implementation defined way. The description here is not complete, see the C standard or the documentation for a portable library that implement these aliases. Some of the integer datatypes have a predetermined bit width, and the `int`*n*`_t` and `uint`*n*`_t`, where *n* is the bit width expressed as a decimal numeral. Some bit widths are required by the C standard. There are also minimum width, fastest minimum width, and special purpose integer datatypes (like `size_t`). Finally there are the integer datatypes `intmax_t` and `uintmax_t` that are the largest provided signed and unsigned integer datatypes.

NOTES

1 The overflow behaviour for arithmetic operations on signed integer datatypes is unspecified in the C standard. For the signed datatypes `signed int`, `signed long int`, `signed long long int`, and similar types (such as `int64_t`), for conformity with LIA the integer operations must notify overflow upon overflow, by default via recording in indicators.

2 The unsigned datatypes `unsigned int`, `unsigned long int`, `unsigned long long int`, and similar types (such as `uint64_t`), can partially conform if operations that properly notify overflow are provided. The operations named `+`, (binary) `-`, and `*` are in the case of the unsigned integer types bound to add_wrap_I, sub_wrap_I, and mul_wrap_I (specified in LIA-2). For (unary) `-`, and integer `/` similar wrapping operations for negation and integer division are accessed. The latter operations are not specified by LIA.

3 For portability reasons, it is common to use the size specified integer datatypes (like `int32_t`, etc. either the standard ones or such datatypes defined in portable libraries).

The LIA-1 parameters for an integer datatype can be accessed by the following syntax (those in the standard are in the header `<limits.h>`):

$maxint_I$	*T*`_MAX`	⋆
$minint_I$	*T*`_MIN`	⋆ (for signed ints)
$modulo_I$	*T*`_MODULO`	† (for signed ints)

where T is `INT` for `signed int`, `LONG` for `signed long int`, `LLONG` for `signed long long int`, `UINT` for `unsigned int`, `ULONG` for `unsigned long int`, and `ULLONG` for `unsigned long long int`.

For the bit size specified integer datatypes the limits are fixed and need not have explicit parameters accessible to programs. For other integer datatypes, such as `size_t` and `int_least32_t`, a complete binding must list how to access their parameters in a portable manner.

The parameter $hasinf_I$ is always **false**, and the parameter $bounded_I$ is always **true** for C integer types, and need not be provided to programs as named parameters. The parameter $minint_I$ is always 0 for the unsigned types, and need not provided for those types. The parameter $modulo_I$ is always **true** for the unsigned types, and need not be provided for those types.

The LIA-1 integer operations are either operators, or macros declared in the header `<stdlib.h>`. The integer operations are listed below, along with the syntax used to invoke them:

$eq_I(x, y)$	`x == y`	$\star$
$neq_I(x, y)$	`x != y`	$\star$
$lss_I(x, y)$	`x < y`	$\star$
$leq_I(x, y)$	`x <= y`	$\star$
$gtr_I(x, y)$	`x > y`	$\star$
$geq_I(x, y)$	`x >= y`	$\star$
$neg_I(x)$	`-x`	$\star$ (if $modulo_I$ = **false**)
$add_I(x, y)$	`x + y`	$\star$ (if $modulo_I$ = **false**)
$add\_wrap_I(x, y)$	`x + y`	$\star$ (if $modulo_I$ = **true**)
$sub_I(x, y)$	`x - y`	$\star$ (if $modulo_I$ = **false**)
$sub\_wrap_I(x, y)$	`x - y`	$\star$ (if $modulo_I$ = **true**)
$mul_I(x, y)$	`x * y`	$\star$ (if $modulo_I$ = **false**)
$mul\_wrap_I(x, y)$	`x * y`	$\star$ (if $modulo_I$ = **true**)
$abs_I(x)$	`tabs(x)`	$\star$ (for signed ints)
$signum_I(x)$	`tsgn(x)`	† (for signed ints)
$quot_I(x, y)$	`tquot(x, y)`	†
$mod_I(x, y)$	`tmod(x, y)`	†

where x and y are expressions of type `signed int`, `signed long int`, `signed long long int`, `unsigned int`, `unsigned long int`, or `unsigned long long int`, as appropriate, t is the empty string for `int`, `l` for `long int`, `ll` for `long long int`, `u` for `unsigned int`, `ul` for `unsigned long int`, and `ull` for `unsigned long long int`. The size determined integer datatypes do not have special prefixes, nor are there type generic names for the operations that are not denoted by operators. This may be an issue for portability.

Note that C requires a "modulo" interpretation for the ordinary addition, subtraction, and multiplication operations for unsigned integer datatypes in C (i.e. $modulo_I$ = **true** for unsigned integer datatypes), and is thus only partially conforming to LIA-1 for the unsigned integer datatypes. For signed integer datatypes, the value of $modulo_I$ is implementation defined. An implementation that wishes to conform to LIA-1 must provide all the LIA-1 integer operations for all the integer datatypes for which LIA-1 conformity is claimed.

C names three floating point datatypes: `float`, `double`, and `long double`. In implementations supporting IEC 60559 (IEEE 754) these datatypes are in practice expected to be **binary32**, **binary64**, and **binary128**, respectively.

ISO/IEC TR 24732:2009 [16] suggest adding the new floating point datatypes `_Decimal32`, `_Decimal64`, and `_Decimal128`. These are intended for the IEC 60559 (IEEE 754) datatypes **decimal32**, **decimal64**, and **decimal128**, respectively. Note that **decimal32** is specified as a storage format only in IEC 60559 (IEEE 754), while ISO/IEC TR 24732:2009 suggests doing computation also directly with `_Decimal32` values, not requiring (but allowing) conversion to a wider decimal type.

The notation *FLT* is used to stand for the name of any one of these datatypes in what follows.

The LIA-1 parameters and derived constants for a floating point datatype can be accessed by the following syntax:

r_F	`FLT_RADIX`	⋆ (`float`, `double`)
p_F	*T*`_MANT_DIG`	⋆
$emax_F$	*T*`_MAX_EXP`	⋆
$emin_F$	*T*`_MIN_EXP`	⋆
$denorm_F$	*T*`_DENORM`	†
iec_60559_F	`__STDC_IEC_559__`	⋆ (`float`, `double`)
$fmax_F$	*T*`_MAX`	⋆
$fminN_F$	*T*`_MIN`	⋆
$fmin_F$	*T*`_DEN`	⋆ (proposed)
$epsilon_F$	*T*`_EPSILON`	⋆
rnd_error_F	*T*`_RND_ERR`	† (partial conf.)
rnd_style_F	`FLT_ROUNDS`	⋆ (partial conf.)

where *T* is `FLT` for `float`, `DBL` for `double`, `LDBL` for `long double`, `DEC32` for `_Decimal32`, `DEC64` for `_Decimal64`, and `DEC128` for `_Decimal128`. The decimal types are not yet part of the C standard, just proposed in a TR.

Note that `FLT_RADIX` (header `float.h`) gives the radix for all of `float`, `double`, and `long double`, not for the decimal datatypes. Also note that `FLT_ROUNDS` (header `float.h`) gives the rounding style for all of `float`, `double`, and `long double`, not for the decimal datatypes.

The C standard specifies that the values of the parameter `FLT_ROUNDS` are `int` values with the following meaning in terms of the LIA-1 rounding styles.

truncate	`FLT_ROUNDS` $= 0$	⋆
nearest	`FLT_ROUNDS` $= 1$	⋆
other	`FLT_ROUNDS` $= 2$	⋆(towards positive infinity)
other	`FLT_ROUNDS` $= 3$	⋆(towards negative infinity)
nearesttiestoeven	`FLT_ROUNDS` $= 4$	†

The value returned from `fegetround()` (header `fenv.h`, and the names below are defined only if the rounding mode can be dynamically controlled) is one of:

truncate	`FE_TOWARDZERO`	⋆
other	`FE_UPWARD`	⋆
other	`FE_DOWNWARD`	⋆
nearesttiestoeven	`FE_TONEAREST`	⋆ (default)

Only the rounding mode `FE_TONEAREST` (with ties to even last digit) conforms to LIA. LIA recommends using separate operations for other roundings, rather than using dynamic rounding modes. Separate operations are in this case more reliable and less error prone.

The LIA-1 floating point operations are bound either to operators, or to macros declared in the header `<math.h>`. The operations are listed below, along with the syntax used to invoke them:

$eq_F(x, y)$	*x* `==` *y*	⋆
$neq_F(x, y)$	*x* `!=` *y*	⋆

$lss_F(x,y)$	x `<` y	⋆
$leq_F(x,y)$	x `<=` y	⋆
$gtr_F(x,y)$	x `>` y	⋆
$geq_F(x,y)$	x `>=` y	⋆
$isnegzero_F(x)$	`isNegZero(`x`)`	†
$istiny_F(x)$	`isTiny(`x`)`	†
$istiny_F(x)$	`-`T`_MIN <` x `&&` x `<` T`_MIN`	⋆
$isnan_F(x)$	`isNaN(`x`)`	†
$isnan_F(x)$	x `!=` x	⋆
$issignan_F(x)$	`isSigNaN(`x`)`	†
$neg_F(x)$	`-`x	(⋆) no invalid notification
$add_{F\to F'}(x,y)$	x `+` y	⋆
$add^{\uparrow}_{F\to F'}(x,y)$	x `+>` y	†
$add^{\downarrow}_{F\to F'}(x,y)$	x `<+` y	†
$sub_{F\to F'}(x,y)$	x `-` y	⋆
$sub^{\uparrow}_{F\to F'}(x,y)$	x `->` y	†
$sub^{\downarrow}_{F\to F'}(x,y)$	x `<-` y	†
$mul_{F\to F'}(x,y)$	x `*` y	⋆
$mul^{\uparrow}_{F\to F'}(x,y)$	x `*>` y	†
$mul^{\downarrow}_{F\to F'}(x,y)$	x `<*` y	†
$div_{F\to F'}(x,y)$	x `/` y	⋆
$div^{\uparrow}_{F\to F'}(x,y)$	x `/>` y	†
$div^{\downarrow}_{F\to F'}(x,y)$	x `</` y	†
$abs_F(x)$	`fabs`t`(`x`)`	⋆
$signum_F(x)$	`fsgn`t`(`x`)`	†
$residue_F(x,y)$	`remainder`t`(`x`,` y`)` or `remainder(`x`,` y`)`	⋆
$sqrt_{F\to F'}(x)$	`sqrt`t`(`x`)` or `sqrt(`x`)`	⋆
$sqrt^{\uparrow}_{F\to F'}(x)$	`sqrtUp`t`(`x`)`	†
$sqrt^{\downarrow}_{F\to F'}(x)$	`sqrtDwn`t`(`x`)`	†
$exponent_{F\to I}(x)$	`(int)(logb`t`(`x`)) + 1`	⋆, (or `(long)`)
$fraction_F(x)$	`fract`t`(`x`)`	†
$scale_{F,I}(x,n)$	`scalbn`t`(`x`,` n`)`	⋆
$scale_{F,I'}(x,m)$	`scalbln`t`(`x`,` m`)`	⋆
$succ_F(x)$	`nexttoward`t`(`x`, INFINITY)`	⋆
$pred_F(x)$	`nexttoward`t`(`x`, -INFINITY)`	⋆
$ulp_F(x)$	`ulp`t`(`x`)`	†
$intpart_F(x)$	`intpart`t`(`x`)`	†
$fractpart_F(x)$	`frcpart`t`(`x`)`	†
$trunc_{F,I}(x,n)$	`trunc`t`(`x`,` n`)`	†
$round_{F,I}(x,n)$	`round`t`(`x`,` n`)`	†

where x and y are expressions of type `float`, `double`, or `long double`, n is of type `int`, and m is of type `long int`, t is `f` for `float`, the empty string for `double`, `l` for `long double`, `d64` for `_Decimal64`, and `d128` for `_Decimal128`. The operations on values of decimal types are not yet part of the C standard, just proposed in a TR.

An implementation that wishes to conform to LIA-1 must provide the LIA-1 floating point operations for all the floating point datatypes for which LIA-1 conformity is claimed.

Arithmetic value conversions in C can be explicit or implicit. The explicit arithmetic value conversions are usually expressed as 'casts', except when converting to/from string formats. The rules for when implicit conversions are applied is not repeated here, but work as if a cast had been applied.

When converting to/from string formats, format strings are used. The format string is used as a pattern for the string format generated or parsed. The description of format strings here is not complete. Please see the C standard for a full description.

In the format strings `%` is used to indicate the start of a format pattern. After the `%`, optionally a string field width (w below) may be given as a positive decimal integer numeral.

For the floating and fixed point format patterns, there may then optionally be a '`.`' followed by a positive integer numeral (d below) indicating the number of fractional digits in the string. The C operations below use HYPHEN-MINUS rather than MINUS (which would have been typographically better), and only digits that are in ASCII, independently of so-called locale. For generating or parsing other kinds of digits, say Arabic digits or Thai digits, another API must be used, that is not standardised in C. For the floating and fixed point formats, $\mathbf{+\infty}$ may be represented as either `inf` or `infinity`, $\mathbf{-\infty}$ may be represented as either `-inf` or `-infinity`, and a **NaN** may be represented as `NaN`; all independently of so-called locale. For language dependent representations, or use of non-ASCII characters like ∞, of these values another API must be used, that is not standardised in C.

For the integer formats then follows an internal type indicator. Not all C integer types have internal type indicators, in particular the portable size fixed types do not have special type indicators (which is an issue for portability). For t below, `hh` indicates `char`, `h` indicates `short int`, the empty string indicates `int`, `l` (the letter l) indicates `long int`, `ll` (the letters ll) indicates `long long int`, and `j` indicates `intmax_t` or `uintmax_t`. Two more of the `..._t` integer datatypes have formatting letters: `z` indicates `size_t` and `t` indicates `ptrdiff_t` in the format. Finally, there are radix and signedness format letters (r below): `d` for signed decimal string; `o`, `u`, `x`, `X` for octal, decimal, hexadecimal with small letters, and hexadecimal with capital letters, all unsigned. E.g., `%jd` indicates decimal numeral string for `intmax_t`, `%2hhx` indicates hexadecimal numeral string for `unsigned char`, with a two character field width, and `%lu` indicates decimal numeral string for `unsigned long int`.

For the floating point formats instead follows another internal type indicator. Not all C floating point types have standard internal type indicators for the format strings. For u below the empty string indicates `double` and `L` indicates `long double`; and there is a proposal to use `H` for `_Decimal32`, `D` for `_Decimal64`, and `DD` for `_Decimal128`. Finally, there is a radix (for the string side) format letter: `e` or `E` for decimal, `a` or `A` for hexadecimal. E.g., `%15.8LA` indicates hexadecimal floating point numeral string for `long double`, with capital letters for the letter components, a field width of 15 characters, and 8 hexadecimal fractional digits.

For the fixed point formats also follows the internal type indicator as for the floating point formats. But for the final part of the pattern, there is another radix (for the string side) format letter (p below), only two are standardised, both for the decimal radix: `f` or `F`. E.g., `%Lf` indicates decimal fixed point numeral string for `long double`, with a small letter for the letter component. (There is also a combined floating/fixed point string format: `g`.)

$convert_{I\rightarrow I'}(x)$ $(INT2)\,x$ $\star$

$convert_{I''\to I}(s)$	`sscanf(s, "%wtr", &i)`		⋆
$convert_{I''\to I}(f)$	`fscanf(f, "%wtr", &i)`		⋆
$convert_{I\to I''}(x)$	`sprintf(s, "%wtr", x)`		⋆
$convert_{I\to I''}(x)$	`fprintf(h, "%wtr", x)`		⋆
$floor_{F\to I}(y)$	`(INT)floort(y)`		⋆
$floor_{F\to I}(y)$	`(INT)nearbyintt(y)`	(when in round towards $-\infty$ mode)	⋆
$rounding_{F\to I}(y)$	`(INT)nearbyintt(y)`	(when in round to nearest mode)	⋆
$ceiling_{F\to I}(y)$	`(INT)nearbyintt(y)`	(when in round towards $+\infty$ mode)	⋆
$ceiling_{F\to I}(y)$	`(INT)ceilt(y)`		⋆
$convert_{I\to F}(x)$	`(FLT)x`		⋆
$convert^{\uparrow}_{I\to F}(x)$	`(FLT>)x`		⋆
$convert^{\downarrow}_{I\to F}(x)$	`(<FLT)x`		⋆
$convert_{F\to F'}(y)$	`(FLT2)y`		⋆
$convert^{\uparrow}_{F\to F'}(y)$	`(FLT2>)y`		⋆
$convert^{\downarrow}_{F\to F'}(y)$	`(<FLT2)y`		⋆
$convert_{F''\to F}(s)$	`sscanf(s, "%w.duv", &r)`		⋆
$convert^{\uparrow}_{F''\to F}(s)$	`sscanf(s, "%>w.duv", &r)`		†
$convert^{\downarrow}_{F''\to F}(s)$	`sscanf(s, "%<w.duv", &r)`		†
$convert_{F''\to F}(f)$	`fscanf(f, "%w.duv", &r)`		⋆
$convert^{\uparrow}_{F''\to F}(f)$	`fscanf(f, "%>w.duv", &r)`		†
$convert^{\downarrow}_{F''\to F}(f)$	`fscanf(f, "%<w.duv", &r)`		†
$convert_{F\to F''}(y)$	`sprintf(s, "%w.duv", y)`		⋆
$convert^{\uparrow}_{F\to F''}(y)$	`sprintf(s, "%>w.duv", y)`		†
$convert^{\downarrow}_{F\to F''}(y)$	`sprintf(s, "%<w.duv", y)`		†
$convert_{F\to F''}(y)$	`fprintf(h, "%w.duv", y)`		⋆
$convert^{\uparrow}_{F\to F''}(y)$	`fprintf(h, "%>w.duv", y)`		†
$convert^{\downarrow}_{F\to F''}(y)$	`fprintf(h, "%<w.duv", y)`		†
$convert_{D'\to F}(s)$	`sscanf(s, "%w.dup", &g)`		⋆
$convert^{\uparrow}_{D'\to F}(s)$	`sscanf(s, "%>w.dup", &g)`		†
$convert^{\downarrow}_{D'\to F}(s)$	`sscanf(s, "%<w.dup", &g)`		†
$convert_{D'\to F}(f)$	`fscanf(f, "%w.dup", &g)`		⋆
$convert^{\uparrow}_{D'\to F}(f)$	`fscanf(f, "%>w.dup", &g)`		†
$convert^{\downarrow}_{D'\to F}(f)$	`fscanf(f, "%<w.dup", &g)`		†
$convert_{F\to D'}(y)$	`sprintf(s, "%w.dup", y)`		⋆
$convert^{\uparrow}_{F\to D'}(y)$	`sprintf(s, "%>w.dup", y)`		†
$convert^{\downarrow}_{F\to D'}(y)$	`sprintf(s, "%<w.dup", y)`		†
$convert_{F\to D'}(y)$	`fprintf(h, "%w.dup", y)`		⋆
$convert^{\uparrow}_{F\to D'}(y)$	`fprintf(h, "%>w.dup", y)`		†
$convert^{\downarrow}_{F\to D'}(y)$	`fprintf(h, "%<w.dup", y)`		†

where s is an expression of type `char*`, f is an expression of type `FILE*`, i is an lvalue expression of type `int`, g is an lvalue expression of type `double`, x is an expression of type *INT*, y is an expression of type *FLT*, *INT2* is the integer datatype that corresponds to I', and *FLT2* is the floating point datatype that corresponds to F'.

C provides non-negative numerals for all its integer and floating point types. The default base is 10, but base 8 (for integers) and 16 (both integer and floating point) can be used too. Numerals for different integer types are distinguished by suffixes: no suffix for `long int`, and `L` for `long long int`. Numerals for different floating point types are distinguished by suffix: `f` for `float`, no suffix for `double`, `l` for `long double`. There is a proposal to use the suffixes `DF` for `_Decimal32`, `DD` for `_Decimal64`, and `DL` for `_Decimal128`. Numerals for floating point types must have a '.' or an exponent in them. The details are not repeated in this example binding, see ISO/IEC 9899:1999, clause 6.4.4.1 Integer constants, and clause 6.4.4.2 Floating constants.

C specifies numerals (as macros) for infinities and NaNs for `float` (header `math.h`):

$+\infty$	`INFINITY`	⋆
qNaN	`NAN`	⋆
sNaN	`NANSIGNALLING`	†

as well as string formats for reading and writing these values as character strings.

C 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, which can be used for floating point errors. For C, the **absolute_precision_underflow** notification is ignored. The behaviour when integer operations initiate a notification is, however, not defined by C.

An implementation that wishes to conform to LIA-1 must provide recording in indicators (for all of the LIA notifications) as one method of notification. (See 6.2.1.) The datatype *Ind* is identified with the datatype `int`. The values representing individual indicators are distinct non-negative powers of two. Indicators can be accessed (header `fenv.h`) by the following syntax:

inexact	`FE_INEXACT`	⋆
underflow	`FE_UNDERFLOW`	⋆
overflow	`FE_OVERFLOW`	⋆ (integers †)
infinitary	`FE_DIVBYZERO`	⋆ (integers †)
invalid	`FE_INVALID`	⋆ (integers †)
absolute_precision_underflow	`FE_ARGUMENT_TOO_IMPRECISE`	†, LIA-2, -3
union of all indicators	`FE_ALL_EXCEPT`	⋆

The empty set can be denoted by `0`. Other indicator subsets can be named by combining individual indicators using bit-wise or. For example, the indicator subset

{**overflow**, **underflow**, **infinitary**}

can be denoted by the expression

```
FE_OVERFLOW | FE_UNDERFLOW | FE_DIVBYZERO
```

The indicator interrogation and manipulation operations (header `fenv.h`) are listed below, along with the syntax used to invoke them:

$clear\_indicators(C, S)$	`feclearexcept(`S`)`	⋆
$set\_indicators(C, S)$	`feraiseexcept(`S`)`	⋆
$current\_indicators(C)$	`fegetexceptflag(returnvalue, FE_ALL_EXCEPT)`	⋆
$test\_indicators(C, S)$	`fetestexcept(`S`)`	⋆

where S is an expression of type `int` representing an indicator subset.

It is vital that indicators are managed separately for separate threads (as required by LIA), in an environment where it is possible to have several threads within a C program. Likewise that dynamically set rounding modes (which LIA-1 does *not* recommend) are also managed separately for separate threads in such an environment.

In order not to lose notification indicators within a C program when the computation is divided into several threads, any in-parameter for thread communication must set in the accepting thread (when the call is accepted) the indicators that are set in the caller, and any out-parameter or result will set in the caller (when the communication call finishes) the indicators that are then set in the accepting thread.

D.3 C++

The programming language C++ is defined by ISO/IEC 14882:2011, *Programming languages – C++* [17].

An implementation should follow all the requirements of LIA-1 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and must be provided by an implementation that wishes to conform to LIA-1. For each of the marked items a suggested identifier is provided. Integer valued parameters and derived constants can be used in preprocessor expressions.

This example binding recommends that all identifiers suggested here be defined in the namespace `std::math`.

The LIA-1 datatype **Boolean** is implemented in the C++ datatype `bool`.

C++ names several integer datatypes: `(signed) int`, `(signed) long (int)`, `(signed) long long (int)`, `unsigned (int)`, and `unsigned long (int)`, and `unsigned long long (int)`. The here parenthesised part of a name may be omitted when using the name in programs. Signed integer datatypes use 2's complement for representation for negative values. The notation INT is used to stand for the name of any one of these datatypes in what follows.

The conformity to LIA of `short int` and `char` (signed or unsigned), and similar "short" integer types are not relevant since values of these types are promoted to `int` (signed or unsigned as appropriate) before arithmetic computations are done.

However, the basic integer datatypes, listed above, have portability issues. They may have different limits in different implementations. Therefore, the C++ standard specifies a number of additional integer datatypes, defined for programs in the headers `<stdint.h>` and `<stddef.h>`. Similar portable integer datatypes have been defined in portable libraries. They are aliased, by typedefs, to the basic integer datatypes, but the aliases are made in an implementation defined way. The description here is not complete, see the C++ standard or the documentation for a portable library that implement these aliases. Some of the integer datatypes have a predetermined bit width, and the `int`n`_t` and `uint`n`_t`, where n is the bit width expressed as a decimal numeral. Some bit widths are required by the C++ standard. There are also minimum width, fastest minimum width, and special purpose integer datatypes (like `size_t`). Finally there are the integer datatypes `intmax_t` and `uintmax_t` that are the largest provided signed and unsigned integer datatypes.

NOTES

1 The overflow behaviour for arithmetic operations on signed integer datatypes is unspecified in the C++ standard. For the signed datatypes `signed int`, `signed long int`, `signed long long int`, and similar types (such as `int64_t`), for conformity with LIA the integer operations must notify overflow upon overflow, by default via recording in indicators.

2 The unsigned datatypes `unsigned int`, `unsigned long int`, `unsigned long long int`, and similar types (such as `uint64_t`), can partially conform if operations that properly notify overflow are provided. The operations named `+`, (binary) `-`, and `*` are in the case of

the unsigned integer types bound to add_wrap_I, sub_wrap_I, and mul_wrap_I (specified in LIA-2). For (unary) `-`, and integer `/` similar wrapping operations for negation and integer division are accessed. The latter operations are not specified by LIA.

3 For portability reasons, it is common to use the size specified integer datatypes (like `int32_t`, etc. either the standard ones or such datatypes defined in portable libraries).

The LIA-1 parameters for an integer datatype can be accessed by the following syntax:

$maxint_I$	`numeric_limits<`INT`>::max()`	$\star$
$minint_I$	`numeric_limits<`INT`>::min()`	$\star$
$hasinf_I$	`numeric_limits<`INT`>::has_infinity`	$\star$
$signed_I$	`numeric_limits<`INT`>::is_signed`	$\star$ (not LIA-1)
$bounded_I$	`numeric_limits<`INT`>::is_bounded`	$\star$
$modulo_I$	`numeric_limits<`INT`>::is_modulo`	$\star$ (partial conf.)

For the bit size specified integer datatypes the limits are fixed and need not have explicit parameters accessible to programs. For other integer datatypes, such as `size_t` and `int_least32_t`, a complete binding must list how to access their parameters in a portable manner.

The parameter $hasinf_I$ is always **false**, and the parameter $bounded_I$ is always **true** for C++ integer types, and need not be provided to programs as named parameters. The parameter $minint_I$ is always 0 for the unsigned types, and need not provided for those types. The parameter $modulo_I$ is always **true** for the unsigned types, and need not be provided for those types.

The LIA-1 integer operations are either operators, or declared in the header `<stdlib.h>`. The integer operations are listed below, along with the syntax used to invoke them:

$eq_I(x, y)$	x `==` y	$\star$
$neq_I(x, y)$	x `!=` y	$\star$
$lss_I(x, y)$	x `<` y	$\star$
$leq_I(x, y)$	x `<=` y	$\star$
$gtr_I(x, y)$	x `>` y	$\star$
$geq_I(x, y)$	x `>=` y	$\star$
$neg_I(x)$	`-`x	$(\star)$
$add_I(x, y)$	x `+` y	$\star$ (if $modulo_I =$ **false**)
$add\_wrap_I(x, y)$	x `+` y	$\star$ (if $modulo_I =$ **true**)
$sub_I(x, y)$	x `-` y	$\star$ (if $modulo_I =$ **false**)
$sub\_wrap_I(x, y)$	x `-` y	$\star$ (if $modulo_I =$ **true**)
$mul_I(x, y)$	x `*` y	$\star$ (if $modulo_I =$ **false**)
$mul\_wrap_I(x, y)$	x `*` y	$\star$ (if $modulo_I =$ **true**)
$abs_I(x)$	`abs(`x`)`	$\star$
$signum_I(x)$	`sgn(`x`)`	†
$quot_I(x, y)$	`quot(`x`, `y`)`	†
$mod_I(x, y)$	`mod(`x`, `y`)`	†

where x and y are expressions of type `signed int`, `signed long int`, `signed long long int`, `unsigned int`, or `unsigned long int`, or `unsigned long long int`, as appropriate.

Note that C++ requires a "modulo" interpretation for the ordinary addition, subtraction, and multiplication operations for unsigned integer datatypes in C (i.e. $modulo_I =$ **true** for unsigned integer datatypes), and is thus only partially conforming to LIA-1 for the unsigned integer datatypes. For signed integer datatypes, the value of $modulo_I$ is implementation defined.

An implementation that wishes to conform to LIA-1 must provide all the LIA-1 integer operations for all the integer datatypes for which LIA-1 conformity is claimed.

C++ names three floating point datatypes: `float`, `double`, and `long double`. In implementations supporting IEC 60559 (IEEE 754) these datatypes are in practice expected to be **binary32**, **binary64**, and **binary128**, respectively.

The notation *FLT* is used to stand for the name of any one of these datatypes in what follows.

The LIA-1 parameters and derived constants for a floating point datatype can be accessed by the following syntax:

r_F	`numeric_limits<`*FLT*`>::radix`	⋆
p_F	`numeric_limits<`*FLT*`>::digits`	⋆
$emax_F$	`numeric_limits<`*FLT*`>::max_exponent`	⋆
$emin_F$	`numeric_limits<`*FLT*`>::min_exponent`	⋆
$denorm_F$	`numeric_limits<`*FLT*`>::has_denorm`	⋆
$hasinf_F$	`numeric_limits<`*FLT*`>::has_infinity`	⋆ (not LIA-1)
$hasqnan_F$	`numeric_limits<`*FLT*`>::has_quiet_nan`	⋆ (not LIA-1)
$hassnan_F$	`numeric_limits<`*FLT*`>::has_signalling_nan`	⋆ (not LIA-1)
iec_60559_F	`numeric_limits<`*FLT*`>::is_iec559`	⋆
$traps_F$	`numeric_limits<`*FLT*`>::traps`	⋆ (not LIA-1)
$tinyness\_before_F$	`numeric_limits<`*FLT*`>::tinyness_before`	⋆ (not LIA-1)
$fmax_F$	`numeric_limits<`*FLT*`>::max()`	⋆
$fminN_F$	`numeric_limits<`*FLT*`>::min()`	⋆
$fmin_F$	`numeric_limits<`*FLT*`>::denorm_min()`	⋆
$epsilon_F$	`numeric_limits<`*FLT*`>::epsilon()`	⋆
rnd_error_F	`numeric_limits<`*FLT*`>::round_error()`	⋆ (partial conf.)
rnd_style_F	`numeric_limits<`*FLT*`>::round_style`	⋆ (partial conf.)
$approx\_p\_10_F$	`numeric_limits<`*FLT*`>::digits10`	⋆ (not LIA-1)
$approx\_emax\_10_F$	`numeric_limits<`*FLT*`>::max_exponent10`	⋆ (not LIA-1)
$approx\_emin\_10_F$	`numeric_limits<`*FLT*`>::min_exponent10`	⋆ (not LIA-1)

The C++ standard specifies that the values of the parameter `round_style` are from the enumeration type `float_round_style`.

	`enum float_round_style`	
	`{`	
	`round_indeterminate = -1,`	⋆
truncate	`round_toward_zero = 0,`	⋆
nearest	`round_to_nearest = 1,`	⋆
other	`round_toward_infinity = 2,`	⋆
other	`round_toward_neg_infinity = 3`	⋆
nearesttiestoeven	`round_to_nearest_even = 4`	†
	`};`	

The value returned from `fegetround()` (header `fenv.h`, and the names below are defined only if the rounding mode can be dynamically controlled) is one of:

truncate	`FE_TOWARDZERO`	⋆
other	`FE_UPWARD`	⋆
other	`FE_DOWNWARD`	⋆
nearesttiestoeven	`FE_TONEAREST`	⋆ (default)

Only the rounding mode `FE_TONEAREST` (with ties to even last digit) conforms to LIA. LIA recommends using separate operations for other roundings, rather than using dynamic rounding modes. Separate operations are in this case more reliable and less error prone.

The LIA-1 floating point operations are either operators, or declared in the header `<math.h>`. The operations are listed below, along with the syntax used to invoke them:

$eq_F(x, y)$	x `==` y	⋆
$neq_F(x, y)$	x `!=` y	⋆
$lss_F(x, y)$	x `<` y	⋆
$leq_F(x, y)$	x `<=` y	⋆
$gtr_F(x, y)$	x `>` y	⋆
$geq_F(x, y)$	x `>=` y	⋆
$isnegzero_F(x)$	`isNegZero(`x`)`	†
$istiny_F(x)$	`isTiny(`x`)`	†
$istiny_F(x)$	`-numeric_limits<`FLT`>::min() <` x `&&` x `< numeric_limits<`FLT`>::min())`	⋆
$isnan_F(x)$	`isNaN(`x`)`	†
$isnan_F(x)$	x `!=` x	⋆
$issignan_F(x)$	`isSigNaN(`x`)`	†
$neg_F(x)$	`-`x	(⋆) no invalid notification
$add_{F \to F'}(x, y)$	x `+` y	⋆
$add^{\uparrow}_{F \to F'}(x, y)$	x `+>` y	†
$add^{\downarrow}_{F \to F'}(x, y)$	x `<+` y	†
$sub_{F \to F'}(x, y)$	x `-` y	⋆
$sub^{\uparrow}_{F \to F'}(x, y)$	x `->` y	†
$sub^{\downarrow}_{F \to F'}(x, y)$	x `<-` y	†
$mul_{F \to F'}(x, y)$	x `*` y	⋆
$mul^{\uparrow}_{F \to F'}(x, y)$	x `*>` y	†
$mul^{\downarrow}_{F \to F'}(x, y)$	x `<*` y	†
$div_{F \to F'}(x, y)$	x `/` y	⋆
$div^{\uparrow}_{F \to F'}(x, y)$	x `/>` y	†
$div^{\downarrow}_{F \to F'}(x, y)$	x `</` y	†
$abs_F(x)$	`fabs`t`(`x`)`	⋆
$signum_F(x)$	`fsgn`t`(`x`)`	†
$residue_F(x, y)$	`remainder`t`(`x`,` y`)` or `remainder(`x`,` y`)`	⋆
$sqrt_{F \to F'}(x)$	`sqrt`t`(`x`)` or `sqrt(`x`)`	⋆
$sqrt^{\uparrow}_{F \to F'}(x)$	`sqrtUp`t`(`x`)`	†
$sqrt^{\downarrow}_{F \to F'}(x)$	`sqrtDwn`t`(`x`)`	†
$exponent_{F \to I}(x)$	`expon(`x`)`	†
$fraction_F(x)$	`fract(`x`)`	†
$scale_{F,I}(x, n)$	`scale(`x`,` n`)`	†
$succ_F(x)$	`succ(`x`)`	†
$pred_F(x)$	`pred(`x`)`	†
$ulp_F(x)$	`ulp(`x`)`	†

$intpart_F(x)$	`intpart(`x`)`	†
$fractpart_F(x)$	`frcpart(`x`)`	†
$trunc_{F,I}(x, n)$	`trunc(`x`, `n`)`	†
$round_{F,I}(x, n)$	`round(`x`, `n`)`	†

where x and y are expressions of type `float`, `double`, or `long double`, and n is of type `int`.

An implementation that wishes to conform to LIA-1 must provide the LIA-1 floating point operations for all the floating point datatypes for which LIA-1 conformity is claimed.

Arithmetic value conversions in C++ can be explicit or implicit. The explicit arithmetic value conversions are usually expressed as 'casts', except when converting to/from string formats. The rules for when implicit conversions are applied is not repeated here, but work as if a cast had been applied.

When converting to/from string formats, format strings are used. The format string is used as a pattern for the string format generated or parsed. The description of format strings here is not complete. Please see the C++ standard for a full description.

In the format strings `%` is used to indicate the start of a format pattern. After the `%`, optionally a string field width (w below) may be given as a positive decimal integer numeral.

For the floating and fixed point format patterns, there may then optionally be a '`.`' followed by a positive integer numeral (d below) indicating the number of fractional digits in the string. The C++ operations below use HYPHEN-MINUS rather than MINUS (which would have been typographically better), and only digits that are in ASCII, independently of so-called locale. For generating or parsing other kinds of digits, say Arabic digits or Thai digits, another API must be used, that is not standardised in C++. For the floating and fixed point formats, $+\infty$ may be represented as either `inf` or `infinity`, $-\infty$ may be represented as either `-inf` or `-infinity`, and a **NaN** may be represented as `NaN`; all independently of so-called locale. For language dependent representations of these values another API must be used, that is not standardised in C++.

For the integer formats then follows an internal type indicator. Not all C integer types have internal type indicators, in particular the portable size fixed types do not have special type indicators (which is an issue for portability). For t below, the empty string indicates `int`, `l` (the letter l) indicates `long int`. Finally, there are radix and signedness format letters (r below): `d` for signed decimal; `o`, `u`, `x`, `X` for octal, decimal, hexadecimal with small letters, and hexadecimal with capital letters, all unsigned. E.g., `%d` indicates decimal numeral string for `int` and `%lu` indicates decimal numeral string for `unsigned long int`.

For the floating point formats instead follows another internal type indicator. Not all C++ floating point types have standard internal type indicators for the format strings. For u below the empty string indicates `double` and `L` indicates `long double`. Finally, there is a radix (for the string side) format letter: `e` or `E` for decimal. E.g., `%15.8LE` indicates hexadecimal floating point numeral string for `long double`, with a capital letter for the letter component, a field width of 15 characters, and 8 hexadecimal fractional digits.

For the fixed point formats also follows the internal type indicator as for the floating point formats. But for the final part of the pattern, there is another radix (for the string side) format letter (p below), only two are standardised, both for the decimal radix: `f` or `F`. E.g., `%Lf` indicates decimal fixed point numeral string for `long double`, with a small letter for the letter component. (There is also a combined floating/fixed point string format: `g`.)

$convert_{I \rightarrow I'}(x)$	`static_cast<INT2>(x)`	$\star$
$convert_{I'' \rightarrow I}(s)$	`sscanf(s, "%wtr", &i)`	$\star$
$convert_{I'' \rightarrow I}(f)$	`fscanf(f, "%wtr", &i)`	$\star$
$convert_{I \rightarrow I''}(x)$	`sprintf(s, "%wtr", x)`	$\star$
$convert_{I \rightarrow I''}(x)$	`fprintf(h, "%wtr", x)`	$\star$
$floor_{F \rightarrow I}(y)$	`static_cast<INT>(floor(y))`	$\star$
$rounding_{F \rightarrow I}(y)$	`static_cast<INT>(round(y))`	$\dagger$
$ceiling_{F \rightarrow I}(y)$	`static_cast<INT>(ceil(y))`	$\star$
$convert_{I \rightarrow F}(x)$	`static_cast<FLT>(x)`	$\star$
$convert^{\uparrow}_{I \rightarrow F}(x)$	`(FLT>)x`	$\star$
$convert^{\downarrow}_{I \rightarrow F}(x)$	`(<FLT)x`	$\star$
$convert_{F \rightarrow F'}(y)$	`static_cast<FLT2>y`	$\star$
$convert^{\uparrow}_{F \rightarrow F'}(y)$	`(FLT2>)y`	$\star$
$convert^{\downarrow}_{F \rightarrow F'}(y)$	`(<FLT2)y`	$\star$
$convert_{F'' \rightarrow F}(s)$	`sscanf(s, "%w.duv", &r)`	$\star$
$convert^{\uparrow}_{F'' \rightarrow F}(s)$	`sscanf(s, "%>w.duv", &r)`	$\dagger$
$convert^{\downarrow}_{F'' \rightarrow F}(s)$	`sscanf(s, "%<w.duv", &r)`	$\dagger$
$convert_{F'' \rightarrow F}(f)$	`fscanf(f, "%w.duv", &r)`	$\star$
$convert^{\uparrow}_{F'' \rightarrow F}(f)$	`fscanf(f, "%>w.duv", &r)`	$\dagger$
$convert^{\downarrow}_{F'' \rightarrow F}(f)$	`fscanf(f, "%<w.duv", &r)`	$\dagger$
$convert_{F \rightarrow F''}(y)$	`sprintf(s, "%w.duv", y)`	$\star$
$convert^{\uparrow}_{F \rightarrow F''}(y)$	`sprintf(s, "%>w.duv", y)`	$\dagger$
$convert^{\downarrow}_{F \rightarrow F''}(y)$	`sprintf(s, "%<w.duv", y)`	$\dagger$
$convert_{F \rightarrow F''}(y)$	`fprintf(h, "%w.duv", y)`	$\star$
$convert^{\uparrow}_{F \rightarrow F''}(y)$	`fprintf(h, "%>w.duv", y)`	$\dagger$
$convert^{\downarrow}_{F \rightarrow F''}(y)$	`fprintf(h, "%<w.duv", y)`	$\dagger$
$convert_{D' \rightarrow F}(s)$	`sscanf(s, "%wup", &g)`	$\star$
$convert^{\uparrow}_{D' \rightarrow F}(s)$	`sscanf(s, "%>w.dup", &g)`	$\dagger$
$convert^{\downarrow}_{D' \rightarrow F}(s)$	`sscanf(s, "%<w.dup", &g)`	$\dagger$
$convert_{D' \rightarrow F}(f)$	`fscanf(f, "%wup", &g)`	$\star$
$convert^{\uparrow}_{D' \rightarrow F}(f)$	`fscanf(f, "%>w.dup", &g)`	$\dagger$
$convert^{\downarrow}_{D' \rightarrow F}(f)$	`fscanf(f, "%<w.dup", &g)`	$\dagger$
$convert_{F \rightarrow D'}(y)$	`sprintf(s, "%w.dup", y)`	$\star$
$convert^{\uparrow}_{F \rightarrow D'}(y)$	`sprintf(s, "%>w.dup", y)`	$\dagger$
$convert^{\downarrow}_{F \rightarrow D'}(y)$	`sprintf(s, "%<w.dup", y)`	$\dagger$
$convert_{F \rightarrow D'}(y)$	`fprintf(h, "%w.dup", y)`	$\star$
$convert^{\uparrow}_{F \rightarrow D'}(y)$	`fprintf(h, "%>w.dup", y)`	$\dagger$
$convert^{\downarrow}_{F \rightarrow D'}(y)$	`fprintf(h, "%<w.dup", y)`	$\dagger$

where s is an expression of type `char*`, f is an expression of type `FILE*`, i is an lvalue expression of type `int`, g is an lvalue expression of type `double`, x is an expression of type *INT*, y is an expression of type *FLT*, *INT2* is the integer datatype that corresponds to I', and *FLT2* is the floating point datatype that corresponds to F'.

C++ provides non-negative numerals for all its integer and floating point types in base 10. The default base is 10, but base 8 (for integers) and 16 (both integer and floating point) can be used too. Numerals for different integer types are distinguished by suffixes: no suffix for `long int`, and L for `long long int`. Numerals for different floating point types are distinguished by suffix: `f` for `float`, no suffix for `double`, `l` for `long double`. Numerals for floating point types must have a '.' or an exponent in them. The details are not repeated in this example binding, see ISO/IEC 14882:2011.

C++ specifies numerals for infinities and NaNs (header `limits`):

+∞	`numeric_limits<`*FLT*`>::infinity()`	⋆
qNaN	`numeric_limits<`*FLT*`>::quiet_NaN()`	⋆
sNaN	`numeric_limits<`*FLT*`>::signaling_NaN()`	⋆

as well as string formats for reading and writing these values as character strings.

C++ has completely undefined behaviour on arithmetic notification.

An implementation that wishes to conform to LIA-1 must provide recording in indicators (for all of the LIA notifications) as one method of notification. (See 6.2.1.) The datatype *Ind* is identified with the datatype `int`. The values representing individual indicators are distinct non-negative powers of two. Indicators can be accessed by the following syntax:

inexact	`FE_INEXACT`	†
underflow	`FE_UNDERFLOW`	†
overflow	`FE_OVERFLOW`	† (also for integers)
infinitary	`FE_DIVBYZERO`	† (also for integers)
invalid	`FE_INVALID`	† (also for integers)
absolute_precision_underflow	`FE_ARGUMENT_TOO_IMPRECISE`	†, LIA-2, -3
union of all indicators	`FE_ALL_EXCEPT`	†

The empty set can be denoted by 0. Other indicator subsets can be named by combining individual indicators using bit-wise or. For example, the indicator subset

{**overflow**, **underflow**, **infinitary**}

would be denoted by the expression

`FE_OVERFLOW | FE_UNDERFLOW | FE_DIVBYZERO`

The indicator interrogation and manipulation operations (header `fenv.h`) are listed below, along with the syntax used to invoke them:

$clear\_indicators(C, S)$	`feclearexcept(`S`)`	⋆
$set\_indicators(C, S)$	`feraiseexcept(`S`)`	⋆
$current\_indicators(C)$	`fegetexceptflag(returnvalue, FE_ALL_EXCEPT)`	⋆
$test\_indicators(C, S)$	`fetestexcept(`S`)`	⋆

where S is an expression of type `int` representing an indicator subset.

It is vital that indicators are managed separately for separate threads (as required by LIA), in an environment where it is possible to have several threads within a C++ program. Likewise that dynamically set rounding modes (which LIA-1 does *not* recommend) are also managed separately for separate threads in such an environment.

In order not to lose notification indicators within a C++ program when the computation is divided into several threads, any in-parameter for thread communication must set in the accepting thread (when the call is accepted) the indicators that are set in the caller, and any out-parameter or result will set in the caller (when the communication call finishes) the indicators that are then set in the accepting thread.

D.4 Fortran

The programming language Fortran is defined by ISO/IEC 1539-1:2010, *Information technology – Programming languages – Fortran – Part 1: Base language*[21].

An implementation should follow all the requirements of LIA-1 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and must be provided by an implementation that wishes to conform to LIA-1. For each of the marked items a suggested identifier is provided.

The Fortran datatype `logical` corresponds to LIA-1 datatype **Boolean**.

Every implementation of Fortran has at least one integer datatype, named `integer`. An implementation is permitted to offer additional integer types (such as `integer(kind=8)`) with a different range, parameterized with the `kind` parameter.

The LIA-1 parameters for an integer datatype can be accessed by the following syntax:

$maxint_I$	`huge(`x`)`	⋆
$minint_I$	`minint(`x`)`	†
$modulo_I$	`modint(`x`)`	† (partial conf.)

where x is an expression of the appropriate integer type, and the result returned is appropriate for the type of x. The parameter $bounded_I$ is always **true**, and need not be provided. The parameter $hasinf_I$ is always **false**, and need not be provided.

The LIA-1 integer operations are listed below, along with the syntax used to invoke them:

$eq_I(x, y)$	x `.eq.` y or x `==` y	⋆
$neq_I(x, y)$	x `.ne.` y or x `/=` y	⋆
$lss_I(x, y)$	x `.lt.` y or x `<` y	⋆
$leq_I(x, y)$	x `.le.` y or x `<=` y	⋆
$gtr_I(x, y)$	x `.gt.` y or x `>` y	⋆
$geq_I(x, y)$	x `.ge.` y or x `>=` y	⋆
$neg_I(x)$	`-`x	⋆
$add_I(x, y)$	x `+` y	⋆
$sub_I(x, y)$	x `-` y	⋆
$mul_I(x, y)$	x `*` y	⋆
$abs_I(x)$	`abs(`x`)`	⋆
$signum_I(x)$	`sign(1, `x`)`	⋆
$mul_I(signum_I(x), abs_I(y))$	`sign(`y`, `x`)`	⋆
$quot_I(x, y)$	`quotient(`x`, `y`)`	†
$mod_I(x, y)$	`modulo(`x`, `y`)`	⋆

where x and y are expressions involving integers of the same `kind`.

An implementation that wishes to conform to LIA-1 must provide all the LIA-1 integer operations for all the integer datatypes for which LIA-1 conformity is claimed.

Implementation of Fortran have at least two floating point datatypes, denoted as `real` (single precision) and `real(kind=kind(0.0d0))` (`double precision`). In implementations supporting IEC 60559 (IEEE 754) these datatypes are in practice expected to be **binary32** and **binary64**, respectively.

An implementation is permitted to offer additional `real` types with different precision or range, parameterized with the `kind` parameter.

The LIA-1 parameters and derived constants for a floating point datatype can be accessed by the following syntax:

r_F	`radix(`x`)`	⋆
p_F	`digits(`x`)`	⋆
$emax_F$	`maxexponent(`x`)`	⋆
$emin_F$	`minexponent(`x`)`	⋆
$denorm_F$	`ieee_support_denormal(`x`)`	(⋆)
iec_60559_F	`ieee_support_standard(`x`)`	(⋆)
$fmax_F$	`huge(`x`)`	⋆
$fminN_F$	`tiny(`x`)`	⋆
$fmin_F$	`tiniest(`x`)`	†
$epsilon_F$	`epsilon(`x`)`	⋆
rnd_error_F	`rnd_error(`x`)`	† (partial conf.)
rnd_style_F	`ieee_get_rounding_mode(`x`)`	(⋆) (partial conf.)

where x is an expression of the appropriate `real(kind=`k`)` datatype.

The allowed values returned from `ieee_get_rounding_mode` are:

truncate	`ieee_to_zero`	⋆
nearest	`ieee_nearest`	⋆ (default)
other	`ieee_up`	⋆
other	`ieee_down`	⋆

Only the rounding mode `ieee_nearest` conforms to LIA. LIA recommends using separate operations for other roundings, rather than using dynamic rounding modes. Separate operations are in this case more reliable and less error prone.

The LIA-1 floating point operations are listed below, along with the syntax used to invoke them:

$eq_F(x, y)$	x `.EQ.` y or x `==` y	⋆
$neq_F(x, y)$	x `.NE.` y or x `/=` y	⋆
$lss_F(x, y)$	x `.LT.` y or x `<` y	⋆
$leq_F(x, y)$	x `.LE.` y or x `<=` y	⋆
$gtr_F(x, y)$	x `.GT.` y or x `>` y	⋆
$geq_F(x, y)$	x `.GE.` y or x `>=` y	⋆
$isnegzero_F(x)$	x `== 0.0 .and. ieee_is_negative(`x`)`	(⋆)
$istiny_F(x)$	`isTiny(`x`)`	†
$istiny_F(x)$	`-tiny(x) <` x `.and.` x `< tiny(x)`	⋆
$isnan_F(x)$	`ieee_is_nan(`x`)`	(⋆)
$issignan_F(x)$	`ieee_is_snan(`x`)`	†

$neg_F(x)$	`-` x	⋆
$add_{F\to F'}(x,y)$	x `+` y	⋆
$add^{\uparrow}_{F\to F'}(x,y)$	x `+>` y	†
$add^{\downarrow}_{F\to F'}(x,y)$	x `<+` y	†
$sub_{F\to F'}(x,y)$	x `-` y	⋆
$sub^{\uparrow}_{F\to F'}(x,y)$	x `->` y	†
$sub^{\downarrow}_{F\to F'}(x,y)$	x `<-` y	†
$mul_{F\to F'}(x,y)$	x `*` y	⋆
$mul^{\uparrow}_{F\to F'}(x,y)$	x `*>` y	†
$mul^{\downarrow}_{F\to F'}(x,y)$	x `<*` y	†
$div_{F\to F'}(x,y)$	x `/` y	⋆
$div^{\uparrow}_{F\to F'}(x,y)$	x `/>` y	†
$div^{\downarrow}_{F\to F'}(x,y)$	x `</` y	†
$abs_F(x)$	`abs(`x`)`	⋆
$signum_F(x)$	`sign(1.0, `x`)`	⋆
$mul_F(signum_F(x), abs_F(y))$	`sign(`y`, `x`)`	⋆
$residue_F(x,y)$	`ieee_rem(`x`, `y`)`	⋆
$sqrt_{F\to F'}(x)$	`sqrt(`x`)`	⋆
$sqrt^{\uparrow}_{F\to F'}(x)$	`sqrtUp`t`(`x`)`	†
$sqrt^{\downarrow}_{F\to F'}(x)$	`sqrtDwn`t`(`x`)`	†
$exponent_{F\to I}(x)$	`exponent(`x`)`	⋆ (dev.: 0 if $x = 0$)
$exponent_{F,I}(x)$	`floor(ieee_logb(`x`)) + 1`	⋆
$fraction_F(x)$	`fraction(`x`)`	⋆
$scale_{F,I}(x,n)$	`scale(`x`, `n`)`	⋆
$scale_{F,I}(x,n)$	`ieee_scalb(`x`, `n`)`	⋆
$succ_F(x)$	`nearest(`x`, 1.0)`	⋆
$succ_F(x)$	`ieee_next_after(`x`, ieee_positive_inf)`	⋆
$pred_F(x)$	`nearest(`x`, -1.0)`	⋆
$succ_F(x)$	`ieee_next_after(`x`, ieee_negative_inf)`	⋆
$ulp_F(x)$	`spacing(`x`)`	⋆
$intpart_F(x)$	`aint(`x`)`	⋆
$fractpart_F(x)$	x `- aint(`x`)`	⋆
$trunc_{F,I}(x,n)$	`trunc(`x,n`)`	†
$round_{F,I}(x,n)$	`round(`x,n`)`	†

where x and y are of a floating point datatype of the same ***kind***, and n is of an integer type.

An implementation that wishes to conform to LIA-1 must provide the LIA-1 floating point operations for all the floating point datatypes for which LIA-1 conformity is claimed.

Arithmetic value conversions in Fortran can be explicit or implicit. Where they are explicit, the conversion function is named like the target type, except when converting to and from string formats. Conversion between numeric and string formats is achieved by using read and write statements.

$convert_{I\to I'}(x)$	`int(`x`, `$kindi2$`)`	⋆

$convert_{I''\to I}(f)$	`read (`s`, '(B`n`)') `r	⋆(binary)
$convert_{I\to I''}(x)$	`write (`s`, '(B`n`)') `x	⋆(binary)
$convert_{I''\to I}(f)$	`read (`s`, '(O`n`)') `r	⋆(octal)
$convert_{I\to I''}(x)$	`write (`s`, '(O`n`)') `x	⋆(octal)
$convert_{I''\to I}(f)$	`read (`s`, '(I`n`)') `r	⋆(decimal)
$convert_{I\to I''}(x)$	`write (`s`, '(I`n`)') `x	⋆(decimal)
$convert_{I''\to I}(f)$	`read (`s`, '(Z`n`)') `r	⋆(hexadecimal)
$convert_{I\to I''}(x)$	`write (`s`, '(Z`n`)') `x	⋆(hexadecimal)
$floor_{F\to I}(y)$	`floor(`y`, `$kindi$`?)`	⋆
$rounding_{F\to I}(y)$	`rounding(`y`, `$kindi$`?)`	†
$ceiling_{F\to I}(y)$	`ceiling(`y`, `$kindi$`?)`	⋆
$convert_{I\to F}(x)$	`real(`x`, `$kind$`)` or for dbl. prec. `dble(`x`)`	⋆
$convert_{F\to F'}(y)$	`real(`y`, `$kind2$`)` or or for dbl. p. `dble(`y`)`	⋆
$convert_{F''\to F}(f)$	`read (`s`, `fmt`) `t	⋆
$convert_{F\to F''}(y)$	`write (`s`, `fmt`) `y	⋆
$convert_{D'\to F}(f)$	`read (`s`, fmt=`lbl_x`) `t	⋆

where x is an expression of type `integer(kind=`***`kindi`***`)`, y is an expression of type `real(kind=`***`kind`***`)`, s is an string variable, w, d, and e are literal digit (0-9) sequences, giving total, decimals, and exponent widths, and *fmt* is one of `'(F`w`.`d`)'`, `'(D`w`.`d`)'`, `'(E`w`.`d`)'`, `'(E`w`.`d`E`e`)'`, `'(EN`w`.`d`)'`, `'(EN`w`.`d`E`e`)'`, `'(ES`w`.`d`)'`, or `'(ES`w`.`d`E`e`)'` (see the Fortran standard for details).

Fortran provides non-negative numerals for all its integer and floating point types in base 10. Numerals for floating point types must have a '.' in them. The *kind* of a numeral is indicated by a suffix (`d` for `double precision`). The details are not repeated in this example binding, see ISO/IEC 1539-1:2010.

Numerals for infinities and NaNs:

$-\infty$	`ieee_negative_inf`	⋆
$+\infty$	`ieee_positive_inf`	⋆
qNaN	`ieee_quiet_nan`	⋆
sNaN	`ieee_signaling_nan`	⋆

as well as string formats for reading and writing these values as character strings (which should be detailed in a real binding).

Fortran implementations can provide recording of indicators for floating point arithmetic notifications, the LIA preferred method. An implementation that wishes to conform to LIA-1 must provide recording in indicators (for all of the LIA notifications) as one method of notification. (See 6.2.1.) The datatype *Ind* is identified with the datatype `integer`. The values representing individual indicators are distinct non-negative powers of two. Indicators can be accessed by the following syntax:

inexact	`ieee_inexact`	⋆
underflow	`ieee_underflow`	⋆
overflow	`ieee_overflow`	⋆ (integers †)
infinitary	`ieee_divide_by_zero`	⋆ (integers †)

invalid	`ieee_invalid`	⋆ (integers †)
absolute_precision_underflow		
	`lia_density_too_sparse`	†, LIA-2, -3
union of all indicators	`ieee_all`	⋆

The empty set if indicators can be denoted by 0. Other indicator subsets can be named by adding together individual indicators. For example, the indicator subset

{**overflow**, **underflow**}

would be denoted by the expression

`ieee_overflow` + `ieee_underflow`

The Fortran standard has subroutines that set or get the status of one exception flag (notification indicator in LIA terminology) at the time:

`ieee_set_flag(`*f*`, .false.)`	clear a single notification indicator
`ieee_set_flag(`*f*`, .true.)`	set (raise) a single notification indicator
`ieee_get_flag(`*f*`, `*out*`)`	get the current value of a single notification indicator

where *f* is one of the values listed above, and *out* is a boolean variable.

However, handling arithmetic exception flags one by one is tedious and error-prone, and also not conforming to LIA-1 nor to IEEE 754-2008.

The indicator interrogation and manipulation operations are listed below, along with the syntax used to invoke them:

$clear\_indicators(C, S)$	`lia_clear_indicators(`S`)`	†
$set\_indicators(C, S)$	`lia_raise_indicators(`S`)`	†
$current\_indicators(C)$	`lia_get_indicators()`	†
$test\_indicators(C, S)$	`lia_test_indicators(`S`)`	†

where S is an integer representing the set of indicators.

It is vital that indicators are managed separately for separate threads (as required by LIA), in an environment where it is possible to have several threads within a Fortran program. Likewise that dynamically set rounding modes (which LIA-1 does *not* recommend) are also managed separately for separate threads in such an environment.

In order not to lose notification indicators within a Fortran program when the computation is divided into several threads, any in-parameter for thread communication must set in the accepting thread (when the call is accepted) the indicators that are set in the caller, and any out-parameter or result will set in the caller (when the communication call finishes) the indicators that are then set in the accepting thread.

D.5 Common Lisp

The programming language Common Lisp is defined by ANSI X3.226-1994, *Information Technology – Programming Language – Common Lisp* [37].

An implementation should follow all the requirements of LIA-1 unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and must be provided by an implementation that wishes to conform to LIA-1. 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. Thus, the parameters $bounded_I$ (and $modulo_I$) are always **false**, $maxint_I$ is positive infinity, and $minint_I$ is negative infinity. LIA-1 requires for unbounded integer types that $hasinf_I$ shall be **true**, and thus for there to be representations of positive and negative infinity. Since these parameters have fixed values, the same for all implementations, they need not be provided as program accessible parameters.

The LIA-1 integer operations are listed below, along with the syntax used to invoke them:

$eq_I(x, y)$	`(=` x y`)`	⋆
$neq_I(x, y)$	`(/=` x y`)`	⋆
$lss_I(x, y)$	`(<` x y`)`	⋆
$leq_I(x, y)$	`(<=` x y`)`	⋆
$gtr_I(x, y)$	`(>` x y`)`	⋆
$geq_I(x, y)$	`(>=` x y`)`	⋆
$neg_I(x)$	`(-` x`)`	⋆
$add_I(x, y)$	`(+` x y`)`	⋆
$sub_I(x, y)$	`(-` x y`)`	⋆
$mul_I(x, y)$	`(*` x y`)`	⋆
$abs_I(x)$	`(abs` x`)`	⋆
$signum_I(x)$	`(sign` x`)`	†

(the `floor`, `ceiling`, `round`, and `truncate` can also accept floating point arguments)

	`(multiple-value-bind (flr md) (floor` x y`))`	⋆
$quot_I(x, y)$	`flr` or `(floor` x y`)`	⋆
$mod_I(x, y)$	`md` or `(mod` x y`)`	⋆
	`(multiple-value-bind (rnd rm) (round` x y`))`	⋆
$ratio_I(x, y)$	`rnd` or `(round` x y`)`	⋆
$residue_I(x, y)$	`rm`	
	`(multiple-value-bind (ceil pd) (ceiling` x y`))`	⋆
$group_I(x, y)$	`ceil` or `(ceiling` x y`)`	⋆
$pad_I(x, y)$	`(- pd)`	
	`(multiple-value-bind (trunc rest) (ceiling` x y`))`	⋆

where x and y are expressions of type `integer`.

An implementation that wishes to conform to LIA-1 must provide all the LIA-1 integer operations for the `integer` datatype.

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, though in light of the 2008 version of IEEE 754, it may be recommendable to map them, respectively, to **binary16**, **binary32**, **binary64**, and **binary128**.

The LIA-1 parameters and derived constants for a floating point datatype can be accessed by the following syntax:

r_F	`(float-radix` x`)`	⋆

p_F	`(float-digits` x`)`	⋆
$emax_F$	`maxexp-`T	⋆
$emin_F$	`minexp-`T	⋆
$denorm_F$	`denorm-`T	†
iec_60559_F	`ieee-754-`T	†
$fmax_F$	`most-positive-`T	⋆
$fminN_F$	`least-positive-normalized-`T	⋆
$fmin_F$	`least-positive-`T	⋆
$epsilon_F$	T`-epsilon`	⋆
rnd_error_F	T`-rounding-error`	† (partial conf.)
rnd_style_F	`rounding-style`	† (partial conf.)

where x is of type `short-float`, `single-float`, `double-float` or `long-float`, and T is the string `short-float`, `single-float`, `double-float`, or `long-float` as appropriate.

NOTE – LIA-1 requires sign symmetry in the range of floating point numbers. Thus the Common Lisp constants of the form `*-negative-*` are not needed since they are simply the negatives of their `*-positive-*` counterparts.

The value of the parameter `rounding-style` is an object of type `rounding-styles`. The values of `rounding-styles` have the following names corresponding to LIA-1 rnd_style_F values:

nearesttiestoeven	`nearesttiestoeven`	†
nearest	`nearest`	†
truncate	`truncate`	†
other	`other`	†

There is no standard way of setting rounding mode (as per IEEE 754) in Common Lisp. Note also that LIA recommends using separate operations for separate roundings, rather than using dynamic rounding modes. Separate operations are in this case more reliable and less error prone.

The LIA-1 floating point operations are listed below, along with the syntax used to invoke them:

$eq_F(x, y)$	`(=` x y`)`	⋆
$neq_F(x, y)$	`(/=` x y`)`	⋆
$lss_F(x, y)$	`(<` x y`)`	⋆
$leq_F(x, y)$	`(<=` x y`)`	⋆
$gtr_F(x, y)$	`(>` x y`)`	⋆
$geq_F(x, y)$	`(>=` x y`)`	⋆
$isnegzero_F(x)$	`(isNegativeZero` x`)`	†
$istiny_F(x)$	`(isTiny` x`)`	†
$isnan_F(x)$	`(isNaN` x`)`	†
$isnan_F(x)$	`(/=` x x`)`	⋆
$issignan_F(x)$	`(isSigNaN` x`)`	†
$neg_F(x)$	`(-` x`)`	⋆
$add_{F \to F'}(x, y)$	`(+` x y`)`	⋆
$add^{\uparrow}_{F \to F'}(x, y)$	`(+>` x y`)`	†
$add^{\downarrow}_{F \to F'}(x, y)$	`(<+` x y`)`	†
$sub_{F \to F'}(x, y)$	`(-` x y`)`	⋆
$sub^{\uparrow}_{F \to F'}(x, y)$	`(->` x y`)`	†

$sub^{\downarrow}_{F\to F'}(x,y)$	`(<- x y)`	†
$mul_{F\to F'}(x,y)$	`(* x y)`	⋆
$mul^{\uparrow}_{F\to F'}(x,y)$	`(*> x y)`	†
$mul^{\downarrow}_{F\to F'}(x,y)$	`(<* x y)`	†
$div_{F\to F'}(x,y)$	`(/ x y)`	⋆
$div^{\uparrow}_{F\to F'}(x,y)$	`(/> x y)`	†
$div^{\downarrow}_{F\to F'}(x,y)$	`(</ x y)`	†
$abs_F(x)$	`(abs x)`	⋆
$signum_F(x)$	`(float-sign x)`	⋆
	`(multiple-value-bind (rnd rm) (fround x y))`	⋆
$residue_F(x,y)$	`rm`	⋆
$sqrt_{F\to F'}(x,y)$	`(sqrt x)`	⋆
$sqrt^{\uparrow}_{F\to F'}(x)$	`(sqrtUp x)`	†
$sqrt^{\downarrow}_{F\to F'}(x)$	`(sqrtDwn x)`	†

	`(multiple-value-bind (frc xpn sg) (decode-float x))`	⋆
$exponent_{F\to I}(x)$	`xpn`	⋆
$fraction_F(x)$	`frc`	⋆
$scale_{F,I}(x,n)$	`(scale-float x n)`	⋆
$succ_F(x)$	`(succ x)`	†
$pred_F(x)$	`(pred x)`	†
$ulp_F(x)$	`(ulp x)`	†

	`(multiple-value-bind (int fract) (ftruncate x))`	⋆
$intpart_F(x)$	`int`	⋆
$fractpart_F(x)$	`fract`	⋆
$trunc_{F,I}(x,n)$	`(truncate-float x n)`	†
$round_{F,I}(x,n)$	`(round-float x n)`	†

where x and y are data objects of the same floating point type, and n is of integer type.

An implementation that wishes to conform to LIA-1 must provide the LIA-1 floating point operations for all the floating point datatypes for which LIA-1 conformity is claimed.

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

$convert_{I\to I''}(x)$	`(format nil "~wB" x)`	⋆(binary)
$convert_{I\to I''}(x)$	`(format nil "~wO" x)`	⋆(octal)
$convert_{I\to I''}(x)$	`(format nil "~wD" x)`	⋆(decimal)
$convert_{I\to I''}(x)$	`(format nil "~wX" x)`	⋆(hexadecimal)
$convert_{I\to I''}(x)$	`(format nil "~r,wR" x)`	⋆(radix r)
$convert_{I\to I''}(x)$	`(format nil "~@R" x)`	⋆(roman numeral)

$floor_{F\to I}(y)$	`(floor y)`	⋆
$rounding_{F\to I}(y)$	`(round y)`	⋆
$ceiling_{F\to I}(y)$	`(ceiling y)`	⋆

$convert_{I\to F}(x)$	`(float x kind)`	⋆

$convert_{F \to F'}(y)$	`(float` y $kind2$`)`	⋆
$convert_{F \to F''}(y)$	`(format nil "~`w`F"` y`)`	⋆
$convert_{F \to F''}(y)$	`(format nil "~`w, e, k, c`E"` y`)`	⋆
$convert_{F \to F''}(y)$	`(format nil "~`w, e, k, c`G"` y`)`	⋆
$convert_{F \to D'}(y)$	`(format nil "~`r, w`,0,#F"` y`)`	⋆

where x is an expression of type *INT*, y is an expression of type *FLT*, and *kind* and *kind*2 are objects of the target floating point type. These functions should be complemented with `floatUp`, `floatDwn`, `formatUp`, and `formatDwn` (all †) for directed roundings.

Conversion from string to numeric value is in Common Lisp done via a general read procedure, which reads Common Lisp 'S-expressions'. That function should be complemented with `readUp` and `readDwn` (†) for directed roundings when encountering floating point values.

Common Lisp provides non-negative numerals for all its integer and floating point datatypes in base 10.

There is no differentiation between the numerals for different floating point datatypes, nor between numerals for different integer datatypes, and integer numerals can be used for floating point values.

Common Lisp does not specify numerals for infinities and NaNs. Suggestion:

$+\infty$	`infinity-integer`	†
$+\infty$	`infinity-`*FLT*	†
qNaN	`nan-`*FLT*	†
sNaN	`signan-`*FLT*	†

as well as string formats for reading and writing these values as character strings.

Common Lisp has a notion of 'exception', alternation of control flow. However, Common Lisp has no notion of compile time type checking, and an operation can return differently typed values for different arguments. When justifiable, Common Lisp arithmetic operations return a rational or a complex floating point value rather than giving a notification, even if the argument(s) to the operation were not complex. For instance, `(sqrt -1)` (quietly) returns a representation of $0 + i$.

The notification method required by Common Lisp is alteration of control flow as described in 6.2.2. Notification is accomplished by signaling a condition of the appropriate type. LIA-1 exceptional values (except inexact, which is unsuitable to handle via change of control flow) are represented by the following Common Lisp condition types:

overflow	`floating-point-overflow`	⋆
underflow	`floating-point-underflow`	⋆
invalid	`arithmetic-error`	⋆
infinitary	`division-by-zero`	⋆
absolute_precision_underflow	`floating-point-density-too-sparse`	†, LIA-2, -3

Note that there is no integer overflow notification, since the integer datatype in Common Lisp is required to be unbounded. These condition types are subtypes of the `serious-condition` error condition type, that is errors which are "serious enough to require interactive intervention if not handled". That includes `floating-point-underflow`, but treating underflow by change of control flow is usually inappropriate. For LIA conformity, numeric notifications that do not cause Common Lisp exceptions must be recorded in indicators.

An implementation that wishes to follow LIA must provide recording in indicators as an alternative means of handling numeric notifications also for the notifications where the Common Lisp standard requires alternation of control flow. (See 6.2.1.) Recording of indicators is the LIA preferred means of handling numeric notifications. In this suggested binding non-negative integer values in the datatype `integer`, are used to represent values in *Ind*. The datatype *Ind* is identified with the datatype `integer`. The values representing individual indicators are distinct non-negative powers of two. Indicators can be accessed by the following syntax:

inexact	`lia-inexact`	†
underflow	`lia-undeflow`	†
overflow	`lia-overflow`	†
infinitary	`lia-infinitary`	†
invalid	`lia-invalid`	†
absolute_precision_underflow		
	`lia-density-too-sparse`	† (LIA-2, -3)
union of all indicators	`lia-all-indicators`	†

The empty set can be denoted by `0`. Other indicator subsets can be named by combining individual indicators using bit-wise or, or just addition, or by subtracting from `lia_all_indicators`.

The indicator interrogation and manipulation operations are listed below, along with the syntax used to invoke them:

clear_indicators(C, S)	`(clear-indicators` S`)`	†
set_indicators(C, S)	`(raise-indicators` S`)`	†
current_indicators(C)	`(current-indicators)`	†
test_indicators(C, S)	`(test-indicators` S`)`	†

where S is an expression of type `integer` representing an indicator subset.

It is vital that indicators are managed separately for separate threads (as required by LIA), in an environment where it is possible to have several threads within a Common Lisp program. Likewise that dynamically set rounding modes (which LIA-1 does *not* recommend) are also managed separately for separate threads in such an environment.

In order not to lose notification indicators within a Common Lisp program when the computation is divided into several threads, any in-parameter for thread communication must set in the accepting thread (when the call is accepted) the indicators that are set in the caller, and any out-parameter or result will set in the caller (when the communication call finishes) the indicators that are then set in the accepting thread.

Annex E
(informative)

Example of a conformity statement

This annex presents an example of a conformity statement for a hypothetical implementation of Fortran. The underlying hardware is assumed to provide 32-bit two's complement integers, and 32- and 64-bit floating point numbers that conform to the IEEE 754 (IEC 60559) standard.

The sample conformity statement follows.

This implementation of Fortran conforms to the following standards:

ISO/IEC 1539-1:2010, *Information technology – Programming languages – Fortran – Part 1: Base language*

IEEE Standard 754-2008, *Standard for floating-point arithmetic*

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

It also conforms to the suggested Fortran binding standard in D.4 of LIA-1.

Only implementation dependent information is directly provided here. The information in the suggested language binding standard for Fortran (see D.4) is provided by reference. Together, these two items satisfy the LIA-1 documentation requirement.

E.1 Types

There is one integer type, called `integer`. There are two floating point types, called `real` and `double precision` (or `real(kind=kind(0.0d0))`).

E.2 Integer parameters

The following table gives the parameters for `integer`, the names of the intrinsic inquiry functions with which they can be accessed at run-time, and their values.

Parameters for `integer`		
parameter	*inquiry function*	*value*
$hasinf_I$	*(none)*	**false**
$maxint_I$	`huge(`x`)`	$2^{31} - 1$
$minint_i$	`minint(`x`)` or `-huge(`x`) - 1`	$-(2^{31})$
$modulo_I$	`modint(`x`)`	**false**

where x is an expression of type `integer`.

E.3 Floating point parameters

The following table gives the parameters for `real(` and `real(kind=kind(0.0d0))`, the names of the intrinsic inquiry functions with which they can be accessed at run-time, and their values.

Parameters for floating point			
parameters	*inquiry function*	`real`	`real(kind=kind(0.0d0))`
r_F	`radix(`x`)`	2	2
p_F	`digits(`x`)`	24	53
$emax_F$	`maxexponent(`x`)`	128	1024
$emin_F$	`minexponent(`x`)`	−125	−1021
denorm	`ieee_support_denormal(`x`)`	**true**	**true**
iec_60559	`ieee_support_standard(`x`)`	**true**	**true**

where x is an expression of type `real` or `real(kind=kind(0.0d0))`, as appropriate.

The third table gives the derived constants, the names of the intrinsic inquiry functions with which they can be accessed at run-time, and the (approximate) values for `real` and `real(kind=kind(0.0d0))`. The inquiry functions return exact values for the derived constants.

Derived constants for floating point (default values for rnd_error_F and rnd_style_F)			
constants	*inquiry function*	`real`	`real(kind=kind(0.0d0))`
$fmax_F$	`huge(`x`)`	3.402823466 e+38	1.7976931349 e+308
$fminN_F$	`tiny(`x`)`	1.175494351 e−38	2.2250738585 e−308
$fmin_F$	`tiniest(`x`)`	1.401298464 e−45	4.9406564584 e−324
$epsilon_F$	`epsilon(`x`)`	1.192092896 e−07	2.2204460493 e−016
rnd_error_F	`rnd_error(`x`)`	0.5	0.5
rnd_style_F	`ieee_get_rounding_mode(`x`)`	`ieee_nearest`	`ieee_nearest`

where x is an expression of type `real` or `real(kind=kind(0.0d0))`, as appropriate.

The Fortran standard function `ieee_set_rounding_mode` can be used to dynamically change the rounding mode, but only round-to-neares-ties-to-even conforms with LIA-1 (other roundings are separate operations in LIA-1).

E.4 Expressions

Expressions that contain more than one LIA-1 arithmetic operation or that contain operands of mixed precisions or types are evaluated strictly according to the rules of Fortran (see clause 7.1.7 of the Fortran standard).

E.5 Notification

Notifications are raised under all circumstances specified by the LIA-1 standard. The programmer selects the method of notification by using a compiler directive. The relevant directives are:

```
!LIA$  NOTIFICATION=RECORDING              (default)
!LIA$  NOTIFICATION=TERMINATE              does not apply to inexact nor underflow
```

If a notification (which is not **inexact** nor **underflow**) occurs when termination is the notification method, execution of the program will be stopped and a termination message written on the standard error output.

If an exception occurs when recording of indicators is the selected method of notification, the value specified by IEEE 754 is used as the value of the operation and execution continues. If any indicator remains set when execution of the program is complete, an abbreviated termination message will be written on the standard error output.

A full termination message, given only if the source has been compiled for debugging, provides the following information:

a) name of the exceptional value (**infinitary**, **overflow**, **invalid**, or **absolute_precision_underflow**),

b) kind of operation whose execution caused the notification,

c) values of the arguments to that operation, and

d) point in the program where the failing operation was invoked (i.e. the name of the source file and the line number within the source file).

An abbreviated termination message only gives the names of the indicators that remain set.

Annex F
(informative)

Example programs

This annex presents a few examples of how various LIA-1 features might be used. The program fragments given here are written in Fortran, C, or Ada, and assume the bindings suggested in D.4, D.2, and D.1, respectively.

F.1 Verifying platform acceptability

A particular numeric program may not be able to function if the floating point type available has insufficient accuracy or range. Other programs may have other constraints.

Whenever the characteristics of the arithmetic are crucial to a program, that program should check those characteristics early on.

Assume that an algorithm needs a representation precision of at least 1 part in a million. Such an algorithm should be protected (in Fortran) by

```
if (1/epsilon(x) < 1.0e6) then
    print 3, 'The actual floating point type used has insufficient precision.'
    stop
end if
```

A range test might look like

```
if ((huge(x) < 1.0e30) .or. (tiny(x) > 1.0e-10)) ...
```

A check for $\frac{1}{2}$-ulp rounding would be

```
if (rnd_error(x) /= 0.5) ...
```

A program that only run on IEC 60559 (IEEE 754) platforms would test

```
if (.not. ieee_support_standard(x)) ...
```

F.2 Selecting alternate code

Sometimes the ability to control rounding behaviour is very useful. This ability is provided by IEC 60559 platforms. An example (in C) is

```
if (__STDC_IEC_559__)
{
    fesetround(FE_UPWARD);
    ... calculate using round toward plus infinity ...
    fesetround(FE_DOWNWARD);
    ... calculate using round toward minus infinity ...
    fesetround(FE_NEAREST);
    ... combine the results ...
}
else
{
        ... perform more costly (or less accurate) calculations ...
    }
```

LIA-1 recommends using separate operations for different roundings, rather than using a rounding mode setting.

F.3 Terminating a loop

Here's an example of an iterative approximation algorithm. We choose to terminate the iteration when two successive approximations are within N ulps of one another. In Ada, this is

```
    Approx, Prev_Approx:  Float;
    N: constant Float := 6.0;          -- max ulp difference for loop termination

    Prev_Approx := First_Guess(input);
    Approx := Next_Guess(input, Prev_Approx);
    while abs(Approx - Prev_Approx) > N * Float'Unit_Last_Place(Approx) loop
        Prev_Approx := Approx;
        Approx := Next_Guess(input, Prev_Approx);
    end loop;
```

This example ignores exceptions and the possibility of non-convergence.

F.4 Estimating error

The following is a Fortran algorithm for dot product that makes an estimate of its own accuracy. Again, we ignore exceptions to keep the example simple.

```
    real A(100), B(100), dot, dotmax
    integer I, loss
    ...
    dot = 0.0
    dotmax = 0.0
    do I = 1, 100
        dot = dot + A(I) * B(I)
        dotmax = max (abs(dot), dotmax)
    end do

    loss = exponent(dotmax) - exponent(dot)
    if (loss > digits(dot)/2) then
        print 3, 'Half the precision may be lost.'
    end if
```

F.5 Saving exception state

Sometimes a section of code needs to manipulate the notification indicators without losing notifications pertinent to the surrounding program. The following code (in C) saves and restores indicator settings around such a section of code.

```
int saved_flags;

saved_flags = fetestexcept(FE_ALL_EXCEPT);
feclearexcept(FE_ALL_EXCEPT);
...  run desired code ...
...  examine indicators and take appropriate action ...
...  clear any indicators that were compensated for ...
feraiseexcept(saved_flags);     /* merge-in previous state */
```

The net effect of this is that the nested code sets only those indicators that denote exceptions that could not be compensated for. Previously set indicators stay set.

F.6 Fast versus accurate

Consider a problem which has two methods of solution. The first solution method is a fast algorithm that works most of the time. However, it occasionally gives incorrect answers because of internal floating point overflows. The second solution method is much more reliable, but is a lot slower.

The following Fortran code tries the fast solution first, and, if that fails (detected via indicator recorded notification(s)), uses the slow but reliable one.

```
saved_flags = lia_get_indicators()

call lia_clear_indicators(lia_all_indicators)
result = FAST_SOLUTION(input)
if (lia_test_indicators(ieee_overflow)) then
    call lia_clear_indicators(lia_all_indicators)
    result = RELIABLE_SOLUTION(input)
end if

call lia_raise_indicators(saved_flags)
```

Demmel and Li discuss a number of similar algorithms in [44].

F.7 High-precision multiply

In general, the exact product of two p-digit numbers requires about $2 \cdot p$ digits to represent. Various algorithms are designed to use such an exact product represented as the sum of two p-digit numbers. That is, given X and Y, we must compute U and V such that

```
U + V = X * Y
```

using only p-digit operations.

Sorenson and Tang [55] present an algorithm to compute U and V. They assume that X and Y are of moderate size, so that no exceptions will occur. The Sorensen and Tang algorithm starts out (in C) as

```
X1 = (double) (float) X ;
X2 = X - X1;

Y1 = (double) (float) Y;
```

```
Y2 = Y - Y1;

A1 = X1*Y1;
A2 = X1*Y2;
A3 = X2*Y1;
A4 = X2*Y2;
```

where all values and operations are in double precision. The conversion to single precision and back to double is intended to chop X and Y roughly in half. Unfortunately, this doesn't always work accurately, and as a result the calculation of one or more of the As is inexact.

Using LIA-1's $round_F$ operation, we can make all these calculations exact. This is done by replacing the first four lines with

```
X1 = round (X, DBL_MANT_DIG/2);
X2 = X - X1;

Y1 = round (Y, DBL_MANT_DIG/2);
Y2 = Y - Y1;
```

LIA-2 specifies the operations add_low_F, sub_low_F, mul_low_F, and other operations to support higher precision calculations, or higher precision datatypes.

Bibliography

This section gives references to publications relevant to LIA-1.

International standards documents

[1] ISO/IEC Directives, Part 2 – *Rules for the structure and drafting of International Standards*, 6th edition, 2011.

[2] IEC 60559:2011, *Standard for floating-point arithmetic.* (Also: IEEE Standard 754-2008, *Standard for floating-point arithmetic.*)

[3] ISO/IEC 10967-2, *Information technology – Language independent arithmetic – Part 2: Elementary numerical functions*, (LIA-2).

[4] ISO/IEC 10967-3, *Information technology – Language independent arithmetic – Part 3: Complex integer and floating point arithmetic and complex elementary numerical functions*, (LIA-3).

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

[6] ISO/IEC 10646, *Information technology – Universal multiple-octet coded character set (UCS).*

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

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

[9] ISO/IEC 13886:1996, *Information technology – Language-independent procedure calling*, (LIPC).

[10] ISO/IEC 11404:2007 *Information technology – General-purpose datatypes (GPD)*, second edition.

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

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

[13] ISO/IEC 13751:2001, *Information technology – Programming languages, their environments and system software interfaces – Programming language extended APL.*

[14] ISO/IEC 10279:1991, *Information technology – Programming languages – Full BASIC.*

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

[16] ISO/IEC TR 24732:2009, *Information technology – Programming languages, their environments and system software interfaces – Extension for the programming language C to support decimal floating-point arithmetic.*

[17] ISO/IEC 14882:2011, *Programming languages – C++.*

[18] ISO/IEC 1989:2002, *Information technology – Programming languages – COBOL .*

[19] ISO/IEC 16262:2011, *Information technology – Programming languages, their environments and system software interfaces – ECMAScript language specification.*

[20] ISO/IEC 15145:1997, *Information technology – Programming languages – FORTH.* (Also: ANSI INCITS 215-1994 (R2006).)

[21] ISO/IEC 1539-1:2010, *Information technology – Programming languages – Fortran – Part 1: Base language.*

[22] ISO/IEC 13816:2007, *Information technology – Programming languages, their environments and system software interfaces – Programming language ISLISP.*

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

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

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

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

[27] ISO 6160:1979, *Programming languages – PL/I.*

[28] ISO/IEC 6522:1992, *Information technology – Programming languages – PL/I general-purpose subset.*

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

[30] ISO/IEC 8824-1:2002, *Information technology – Abstract syntax notation one (ASN.1) – Part 1: Specification of basic notation.*

[31] ISO 9001:2000, *Quality management systems – Requirements.*

[32] ISO/IEC 9126:2001, *Software engineering – Product quality – Part 1: Quality model.*

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

National and other standards documents

[34] IEEE 745-2008, *Standard for floating-point arithmetic.*

[35] The Unicode Standard, version 5.0, 2007.

[36] ANSI INCITS 113-1987 (R2003), *Information systems – Programming language – Full BASIC.*

[37] ANSI INCITS 226-1994 (R1999), *Information technology – Programming language – Common Lisp.*

[38] ANSI INCITS 53-1976 (R1998), *Programming languages – PL/I.*

[39] ANSI/IEEE 1178-1990, *Standard for the scheme programming language.*

[40] ANSI INCITS 319-1998 (R2007), *Information technology – Programming language – Smalltalk.*

Books, articles, and other documents

[41] M. Abramowitz and I. Stegun (eds), *Handbook of mathematical functions with formulas, graphs, and mathematical tables*, tenth printing, 1972, U.S. Government Printing Office, Washington, D.C. 20402.

[42] W. S. Brown, *A simple but realistic model of floating-point computation*, ACM Transactions on Mathematical Software, Vol. 7, 1981, pp.445-480.

[43] J. Du Croz and M. Pont, *The Development of a floating-point validation package*, NAG Newsletter, No. 3, 1984.

[44] J. W. Demmel and Xiaoye Li, *Faster numerical algorithms via exception handling*, 11th International symposium on computer arithmetic, Winsor, Ontario, June 29 - July 2, 1993.

[45] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys, vol. 23, no. 1, March 1991.

[46] J. R. Hauser, *Handling floating-point exceptions in numeric programs*, ACM Transactions on Programming Languages and Systems, vol. 18, no. 2, March 1986.

[47] J. E. Holm, *Floating point arithmetic and program correctness proofs*, Cornell University TR 80-436, 1980.

[48] C. B. Jones, *Systematic software development using VDM*, Prentice-Hall, 1986.

[49] W. Kahan and J. Palmer, *On a proposed floating-point standard*, SIGNUM Newsletter, vol. 14, nr. si-2, October 1979, pp.13-21.

[50] 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, Oxford, 1987.

[51] 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.

[52] D. E. Knuth, *Semi-numerical algorithms*, Addison-Wesley, 1969, section 4.4.

[53] U. Kulisch and W. L. Miranker, *Computer arithmetic in theory and practice*, Academic Press, 1981.

[54] U. Kulisch and W. L. Miranker (eds), *A new approach to scientific computation*, Academic Press, 1983.

[55] 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.

[56] N. L. Schryer, *A test of a computer's floating-point unit*, Computer Science Technical Report no. 89, AT&T Bell Laboratories, Murray Hill, NJ, 1981.

[57] G. Bohlender, W. Walter, P Kornerup, D. W. Matula, *Semantics for exact floating point operations*, Proceedings of the 10th IEEE Symposium on Computer Arithmetic, 1991.

[58] W. Walter et al., *Proposal for accurate floating-point vector arithmetic*, Mathematics and Computers in Simulation, vol. 35, no. 4, pp. 375-382, IMACS, 1993.

[59] B. A. Wichmann, *Floating-point interval arithmetic for validation*, NPL Report DITC 76/86, 1986.

[60] B. A. Wichmann, *Towards a formal specification of floating point*, Computer Journal, vol. 32, October 1989, pp.432-436.

[61] B. A. Wichmann, *Getting the correct answers*, NPL Report DITC 167/90, June 1990.

[62] J. Gosling, B. Joy, G. Steele, *The Java language specification*, second edition, 2000.

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

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

[65] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*, The MIT Press, 1997, ISBN: 0-262-63181-4.