SC22/WG11 N507

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

Working draft for the Second edition
2007-07-17

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

EDITOR'S WORKING DRAFT
July 18, 2007 11:07

Editor:
Kent Karlsson
E-mail: kent.karlsson14@comhem.se

Reference number
ISO/IEC WD 10967-1.1:2007(E)

Contents

Foreword **vii**

Introduction **viii**

1 Scope **1**
1.1 Inclusions . . . 1
1.2 Exclusions . . . 2

2 Conformity **3**

3 Normative references **4**

4 Symbols and definitions **4**
4.1 Symbols . . . 4
4.1.1 Sets and intervals . . . 4
4.1.2 Operators and relations . . . 4
4.1.3 Exceptional values . . . 5
4.1.4 Datatypes . . . 6
4.1.5 Special values . . . 6
4.1.6 Operation specification framework . . . 6
4.2 Definitions of terms . . . 7

5 Specifications for integer and floating point datatypes and operations **10**
5.1 Integer datatypes and operations . . . 12
5.1.1 Integer result function . . . 13
5.1.2 Integer operations . . . 13
5.1.2.1 Comparisons . . . 13
5.1.2.2 Basic arithmetic . . . 14
5.2 Floating point datatypes and operations . . . 15
5.2.1 Conformity to IEC 60559 . . . 17
5.2.2 Range and granularity constants . . . 17
5.2.3 Approximate operations . . . 17
5.2.4 Rounding and rounding constants . . . 18
5.2.5 Floating point result function . . . 19
5.2.6 Floating point operations . . . 20
5.2.6.1 Comparisons . . . 20
5.2.6.2 Basic arithmetic . . . 22
5.2.6.3 Value dissection . . . 24
5.2.6.4 Value splitting . . . 25
5.3 Operations for conversion between numeric datatypes . . . 26
5.4 Numerals as operations in a programming language . . . 26

6 Notification **26**
6.1 Model handling of notifications . . . 26
6.2 Notification alternatives . . . 27
6.2.1 Recording in indicators . . . 27
6.2.2 Alteration of control flow . . . 29

6.2.3 Termination with message . . . 29
6.3 Delays in notification . . . 29
6.4 User selection of alternative for notification . . . 30

7 Relationship with language standards **30**

8 Documentation requirements **31**

Annex A (normative) **Partial conformity** **33**
A.1 Integer overflow notification relaxation . . . 33
A.2 Infinitary notification relaxation . . . 34
A.3 Denormalisation loss notification relaxations . . . 34
A.4 Subnormal values relaxation . . . 35
A.5 Accuracy relaxation for add, subtract, multiply, and divide . . . 35
A.6 Comparison operations relaxation . . . 38
A.7 Sign symmetric value set relaxation . . . 38

Annex B (informative) **IEC 60559 bindings** **39**
B.1 Summary . . . 39
B.2 Notification . . . 40
B.3 Rounding . . . 41

Annex C (informative) **Requirements beyond IEC 60559** **43**

Annex D (informative) **Rationale** **45**
D.1 Scope . . . 45
D.1.1 Inclusions . . . 45
D.1.2 Exclusions . . . 45
D.1.3 Companion parts to this part . . . 46
D.2 Conformity . . . 46
D.2.1 Validation . . . 47
D.3 Normative references . . . 47
D.4 Symbols and definitions . . . 47
D.4.1 Symbols . . . 48
D.4.2 Definitions of terms . . . 48
D.5 Specifications for integer and floating point datatypes and operations . . . 49
D.5.1 Integer datatypes and operations . . . 50
D.5.1.0.1 Unbounded integers . . . 50
D.5.1.0.2 Bounded non-modulo integers . . . 51
D.5.1.0.3 Modulo integers . . . 51
D.5.1.0.4 Modulo integers versus overflow . . . 52
D.5.1.1 Integer result function . . . 52
D.5.1.2 Integer operations . . . 52
D.5.1.2.1 Comparisons . . . 52
D.5.1.2.2 Basic arithmetic . . . 52
D.5.2 Floating point datatypes and operations . . . 53
D.5.2.0.1 Constraints on the floating point parameters . . . 54
D.5.2.0.2 Radix complement floating point . . . 55
D.5.2.1 Conformity to IEC 60559 . . . 56

D.5.2.1.1 Subnormal numbers . . . 56
D.5.2.1.2 Signed zero . . . 57
D.5.2.1.3 Infinities and NaNs . . . 57
D.5.2.2 Range and granularity constants . . . 57
D.5.2.2.1 Relations among floating point datatypes . . . 58
D.5.2.3 Approximate operations . . . 58
D.5.2.4 Rounding and rounding constants . . . 59
D.5.2.5 Floating point result function . . . 60
D.5.2.6 Floating point operations . . . 60
D.5.2.6.1 Comparisons . . . 61
D.5.2.6.2 Basic arithmetic . . . 61
D.5.2.6.3 Value dissection . . . 61
D.5.2.6.4 Value splitting . . . 62
D.5.2.7 Levels of predictability . . . 62
D.5.2.8 Identities . . . 63
D.5.2.9 Precision, accuracy, and error . . . 65
D.5.2.9.1 LIA-1 and error . . . 66
D.5.2.9.2 Empirical and modelling errors . . . 66
D.5.2.9.3 Propagation of errors . . . 67
D.5.2.10 Extra precision . . . 68
D.5.3 Conversion operations . . . 68
D.6 Notification . . . 69
D.6.1 Model handling of notifications . . . 69
D.6.2 Notification alternatives . . . 69
D.6.2.1 Recording of indicators . . . 69
D.6.2.2 Alteration of control flow . . . 70
D.6.2.3 Termination with message . . . 71
D.6.3 Delays in notification . . . 71
D.6.4 User selection of alternative for notification . . . 71
D.7 Relationship with language standards . . . 72
D.8 Documentation requirements . . . 73

Annex E (informative) **Example bindings for specific languages** **75**
E.1 Ada . . . 76
E.2 C . . . 80
E.3 C++ . . . 86
E.4 Fortran . . . 91
E.5 Common Lisp . . . 95

Annex F (informative) **Example of a conformity statement** **101**
F.1 Types . . . 101
F.2 Integer parameters . . . 101
F.3 Floating point parameters . . . 102
F.4 Definitions . . . 102
F.5 Expressions . . . 103
F.6 Notification . . . 103

Annex G (informative) **Example programs** **105**

G.1 Verifying platform acceptability 105
G.2 Selecting alternate code 105
G.3 Terminating a loop 106
G.4 Estimating error 106
G.5 Saving exception state 106
G.6 Fast versus accurate 107
G.7 High-precision multiply 107

Annex H (informative) **Bibliography** **109**

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) are worldwide federations of national bodies (member bodies). The work of preparing International standards is normally carried out through ISO or IEC technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. ISO collaborates closely with the IEC on all matters of electrotechnical standardization.

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

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees 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.

ISO/IEC 10967-1 was prepared by 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 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*

Additional parts will specify other arithmetic datatypes or arithmetic operations.

Introduction

EDITOR'S NOTE – This needs to be rewritten. Especially considering the success of IEEE 754, and its revision.

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 document 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 document 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 document 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. These semantics need to be precise enough for numerical analysis, but not so restrictive as to prevent efficient implementation of the language on a wide range of platforms.

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 document does not attempt to ensure bit-for-bit identical results when programs are transferred between language processors, or translated from one language into another. Programming languages and platforms are too diverse to make that a sensible goal. 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.

The content

This document 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 on several platforms are covered. In particular, the IEEE 754 datatypes are covered, both those of radix 2 and those of radix 10, and both limited and unlimited integer datatypes are covered.

The requirements of this document 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

is out of range or undefined). Fourth, the numeric properties of conforming platforms must be publicly documented.

This document focuses on the classical integer and floating point datatypes. Subsequent parts considers common elementary numerical functions (part 2), complex numerical numbers (part 3), and possibly additional arithmetic types such as interval arithmetic and fixed point.

Relationship to hardware

This document is not a hardware architecture standard. It makes no sense to talk about an "LIA machine". Future platforms are expected either to duplicate existing architectures, or to satisfy high quality architecture standards such as IEC 60559 (also known as IEEE 754). The floating point requirements of this document are compatible with (and enhance) IEC 60559.

This document provides a bridge between the abstract view provided by a programming language standard and the precise details of the actual arithmetic implementation.

The benefits

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

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

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

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

Programs will be able to extract data such as the exponent of a floating point number in an implementation independent way.

Programs will be able to detect (and possibly correct for) exceptions in arithmetic processing.

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

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

Information technology — Language independent arithmetic —

Part 1: Integer and floating point arithmetic

1 Scope

This document specifies the 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 document 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 document.

An implementor may choose any combination of hardware and software support to meet the specifications of this document. 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 document) denotes the total computing environment pertinent to this document, including hardware, language processors, subroutine libraries, exception handling facilities, other software, and documentation.

1.1 Inclusions

This document 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 document 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, at least one of the datatypes conforming to this document, and

5) numerals for all values specified in this document in a conforming datatype.

This document 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.

e) Methods for reporting arithmetic exceptions.

Some operations specified in part 2 (ISO/IEC 10967-2) are included by reference in this document, making them required rather than optional. **EDITOR'S NOTE** – TO DO: move them completely to this document

1.2 Exclusions

This document provides no specifications for:

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

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

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

d) A rational datatype, or the operations on such data. This document 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. 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 uninitialized data,

4) the rules for assignment, parameter passing, and returning value.

NOTE – See Clause 7 and Annex E 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 document. Internal representations need not be unique, nor is there a requirement for identifiable fields (for sign, exponent, and so on).

Furthermore, this document 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 document 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 document 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 document if and only if it conforms to the binding standard. In the case of conflict between a binding standard and this document, 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 document with respect to other datatypes independently of that binding standard.

When no binding standard for a language exists, an implementation conforms to this document if and only if it provides one or more datatypes and operations that together satisfy all the requirements of clauses 5 through 8 relevant to that datatype.

Conformity to this document is always with respect to a specified set of datatypes. Under certain circumstances, conformity to IEC 60559 is implied by conformity to this document.

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

An implementation is permitted to have modes of operation that do not conform to this document. A conforming implementation shall specify how to select the modes of operation that ensure conformity. However, a mode of operation that conforms to this document 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 D.7 for recommendations on the proper content of a binding standard. See Annex F for an example of a conformity statement, and Annex E for suggested language bindings.

2 A complete binding for this document may include (explicitly or by reference) a binding for IEC 60559 as well. See 5.2.1 and annex B.

3 This document 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.

4 It is not possible to conform to this document without specifying to which datatypes (and modes of operation) conformity is claimed.

5 All the operations specified in this document 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 document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

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

ISO/IEC 10967-2:2001, *Information technology – Language independent arithmetic – Part 2: Elementary numerical functions.*

ISO/IEC 10967-3:2006, *Information technology – Language independent arithmetic – Part 3: Complex integer and floating point arithmetic and complex elementary numerical functions.*

4 Symbols and definitions

4.1 Symbols

4.1.1 Sets and intervals

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

The conventional notation for set definition and manipulation is used.

The following notation for intervals is used:

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

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

4.1.2 Operators and relations

All prefix and infix operators have their conventional (exact) mathematical meaning. The conventional notation for set definition and manipulation is also used. In particular:

$\Rightarrow$ 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
$\cup$, $\cap$, $\in$, $\notin$, $\subset$, $\subseteq$, $\not\subseteq$, $\neq$, and $=$ with sets
$\times$ for the Cartesian product of sets
$\rightarrow$ for a mapping between sets
$|$ for the divides relation between integer values
x^y, $\sqrt{x}$, $\log_b$ 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:

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

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

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

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

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

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

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

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

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

4.1.3 Exceptional values

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

a) **underflow**: the result has a denormalised representation and may have lost accuracy due to the denormalisation (more than lost by ordinary rounding if the exponent range was unbounded).

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

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

NOTE 1 – **infinitary** is a generalisation of **divide_by_zero**.

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

e) **absolute_precision_underflow**: indicates that the argument is such that the density of representable argument values is too small in the neighbourhood of the given argument value for a numeric result to be considered appropriate to return. Used for operations that approximate trigonometric functions (part 2 and part 3), and hyperbolic and exponentiation functions (part 3).

NOTE 2 – The exceptional value **inexact** is not specified in ISO/IEC 10967, but IEC 60559 conforming implementations will provide it. It should then be used also for operations approximating transcendental functions, when the returned result may be approximate. This part of ISO/IEC 10967 does not specify when it is appropriate to return this exceptional value, but does specify an appropriate continuation value. Thus, v is specified by ISO/IEC 10967

when v or **inexact**(v) should be returned by implementations that are based on IEC 60559. **underflow** and **overflow** implies **inexact**, implicitly or explicitly.

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

4.1.4 Datatypes

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

NOTE 1 – Mathematical relations *are* true or false (or undefined, if an operand is undefined), which are abstract conditions, not values in a datatype. In contrast, **true** and **false** are values in **Boolean**.

4.1.5 Special values

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

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

These floating point values are not part of the set F (see clause 5.2), but if iec_559_F (see clause 5.2.1) has the value **true**, these values are included in the floating point datatype in the implementation that corresponds to F.

NOTE 1 – This document uses the above five special values for compatibility with IEC 60559. In particular, the symbol **−0** (in bold) is not the application of (mathematical) unary − to the value 0, and is a value logically distinct from 0.

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

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 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. 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.

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

4.2 Definitions of terms

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

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

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

4.2.3
continuation value
A computational value used as the result of an arithmetic operation when an exception occurs. Continuation values are intended to be used in subsequent arithmetic processing. A continuation value can be a (in the datatype representable) value in $\mathcal{R}$ or an IEC 60559 special value. (Contrast with *exceptional value*. See 6.2.1.)

4.2.4
denormalisation loss
A larger than normal rounding error caused by the fact that subnormal values (including zeros) have less than full precision. (See 5.2.4 for a full definition.)

4.2.5
error
⟨in computed value⟩ The difference between a computed value and the mathematically correct value. Used in phrases like "rounding error" or "error bound".

4.2.6
error
⟨computation gone awry⟩ A synonym for *exception* in phrases like "error message" or "error output". Error and exception are not synonyms in any other contexts.

4.2.7
exception
The inability of an operation to return a suitable finite numeric result from finite arguments. This might arise because no such finite result exists mathematically (**infinitary** (e.g., at a pole), **invalid** (e.g., when the true result is in $\mathcal{C}$ but not in $\mathcal{R}$)), or because the mathematical result cannot, or might not, be representable with sufficient accuracy (**underflow**, **overflow**) or viability (**absolute_precision_underflow**).

NOTES

1 **absolute_precision_underflow** is not used in this document, but 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 E.

4.2.8
exceptional value
A non-numeric value produced by an arithmetic operation to indicate the occurrence of an exception. Exceptional values are not used in subsequent arithmetic processing. (See clause 5.)

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.

4.2.9
helper function
A function used solely to aid in the expression of a requirement. Helper functions are not visible to the programmer, and are not required to be part of an implementation. However, some implementation defined helper functions are required to be documented.

4.2.10
implementation (of this document)
The total arithmetic environment presented to a programmer, including hardware, language processors, exception handling facilities, subroutine libraries, other software, and documentation pertinent to this document.

4.2.11
literal
A syntactic entity, that does not have any proper sub-entity that is an expression, denoting a constant value.

4.2.12
normalised
Those values of a floating point type F that provide the full precision allowed by that type. (See F_{N} in 5.2 for a full definition.)

4.2.13
notification
The 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. (See clause 6 for details.)

4.2.14
numeral
A numeric literal. It may denote a value in $\mathcal{Z}$ or $\mathcal{R}$, $\mathbf{-0}$, an infinity, or a NaN.

4.2.15
operation
A function directly available to the programmer, as opposed to helper functions or theoretical mathematical functions.

4.2.16
precision
The number of digits in the fraction of a floating point number. (See clause 5.2.)

4.2.17
rounding
The act of computing a representable final result for an operation that is close to the exact (but unrepresentable) result for that operation. Note that a suitable representable result may not exist (see 5.2.5). (See also D.5.2.4 for some examples.)

4.2.18
rounding function
Any function $rnd : \mathcal{R} \to X$ (where X is a discrete and unlimited subset of $\mathcal{R}$) that maps each element of X to itself, and is monotonic non-decreasing. Formally, if x and y are in $\mathcal{R}$,

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

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

4.2.19
round to nearest
The property of a rounding function rnd that when $u \in \mathcal{R}$ is between two adjacent values in X, $rnd(u)$ selects the one nearest u. If the adjacent values are equidistant from u, either value can be chosen deterministically, but shall be such that $rnd(-u) = -rnd(u)$.

4.2.20
round toward minus infinity
The property of a rounding function rnd that when $u \in \mathcal{R}$ is between two adjacent values in X, $rnd(u)$ selects the one less than u.

4.2.21
round toward plus infinity
The property of a rounding function rnd that when $u \in \mathcal{R}$ is between two adjacent values in X, $rnd(u)$ selects the one greater than u.

4.2.22
shall
A verbal form used to indicate requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted. (Quoted from the directives [1].)

4.2.23
should
A verbal form used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that (in the negative form) a certain possibility is deprecated but not prohibited. (Quoted from the directives [1].)

4.2.24
signature (of a function or operation)
A summary of information about an operation or function. A signature includes the function or operation name; a subset of allowed argument values to the operation; and a superset of results from the function or operation (including exceptional values if any), if the argument is in the subset of argument values given in the signature.

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

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

> NOTE 5 – Operations are permitted to accept argument values not listed in the signature of the operation. In particular, IEC 60559 special values are not in F, but must be accepted as arguments if iec_559_F has the value **true**.

4.2.25
subnormal
Values of a floating point datatype F, including 0 and also the special value $\textbf{−0}$ (not in F), that provide less than the full precision allowed by that type. (See F_{S} in 5.2 for a full definition.)

4.2.26
ulp
The value of one "unit in the last place" of a floating point number. This value depends on the exponent, the radix, and the precision used in representing the number. Thus, the ulp of a normalised value x (in F), with exponent t, precision p_F, and radix r_F, is $r_F^{t-p_F}$, and the ulp of a subnormal value is $fminD_F$. (See clause 5.2.)

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 (usually the same as for the arguments, but there are exceptions, like for the conversion operations) or a boolean value. For any particular arithmetic datatype, the set of non-special values is characterized by a small number of parameters. An exact definition of the value set will be given in terms of these parameters.

Given the datatype's non-special value set, V, the accompanying arithmetic operations will be specified as mathematical functions on V union special values (the special values are not written out in the argument part of the signature). These functions typically return values in V or a special value (only the ones that can arise from non-special-value arguments), but they may instead nominally return certain exceptional values (not to be confused with the special values) that are not in any arithmetic datatype. Though nominally listed as a return value, mathematically it is really part of a second component of the result, as explained in clause 4.1.6, and to be handled in an implementation as described in clause 6.

The exceptional values used in this document are **underflow**(generalisation of underflow), **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 corresponding to cyclic functions. For many cases this document specifies which continuation value, that is actual 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**(**+∞**) expresses that the exceptional value **infinitary** in that case is to be accompanied by a continuation value of **+∞** (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 document 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.

Each operation specified in this document is given with a signature. Each operation is further specified by a specification in cases. The definition may use helper functions that may in part be defined by a binding or an implementation.

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 document specifies the behaviour of integer operations when applied to infinitary values, but not for other such additional values. This document 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 document. 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 document 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 1 – This document requires an implementation to provide methods to access values, operations, and other facilities. Ideally, these methods are provided by a 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 D.7.

5.1 Integer datatypes and operations

The non-special value set, I, for an integer datatype shall be a subset of $\mathcal{Z}$, characterized 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 $modulo_I$. For conformity to this second edition, that parameter shall be regarded as having the value **false**. Part 2 includes specifications for operations add_wrap_I, sub_wrap_I, and mul_wrap_I. A binding may still have a parameter $modulo_I$, and it having the value **true** (non-conforming case) indicates that the binding binds the basic integer arithmetic operations to the corresponding wrapping operation instead of the add_I, sub_I, and mul_I operations of this second edition of this document.

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

$$I = \mathcal{Z}$$

In this case, $hasinf_I$ shall be **true**, and 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 $bounded_I$, $hasinf_I$, $minint_I$, 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 D.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.

6 If the value of a parameter (like $bounded_I$) is dictated by a language standard, implementations of that language need not provide program access to that parameter explicitly. But for programmer convenience, $minint_I$ should anyway be provided for all signed integer datatypes, and $maxint_I$ should anyway be provided for all integer datatypes.

5.1.1 Integer result function

If $bounded_I$ 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:

$result_I : \mathcal{Z} \to I \cup \{\textbf{overflow}\}$

is defined by

$$
\begin{array}{lll}
result_I(x) & = x & \text{if } x \in I \\
 & = \textbf{overflow}(-\infty) & \text{if } x \in \mathcal{Z} \text{ and } x \notin I \text{ and } x < 0 \\
 & = \textbf{overflow}(+\infty) & \text{if } x \in \mathcal{Z} \text{ and } x \notin I \text{ and } x > 0
\end{array}
$$

NOTES

1 For integer operations, this document does not specify continuation values for **overflow** when $hasinf_I = \textbf{false}$ 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

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

5.1.2.1 Comparisons

$eq_I : I \times I \to \textbf{Boolean}$

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

$neq_I : I \times I \to \textbf{Boolean}$

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

$lss_I : I \times I \to \textbf{Boolean}$

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

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

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

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

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

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

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

5.1.2.2 Basic arithmetic

If I is unsigned, it is permissible to omit the operations neg_I, abs_I, and $signum_I$.

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

$add_I(x, y)$ $= result_I(x + y)$ if $x, y \in I$
$= \mathbf{-\infty}$ if $x \in I \cup \{\mathbf{-\infty}\}$ and $y = \mathbf{-\infty}$
$= \mathbf{-\infty}$ if $x = \mathbf{-\infty}$ and $y \in I$
$= \mathbf{+\infty}$ if $x \in I \cup \{\mathbf{+\infty}\}$ and $y = \mathbf{+\infty}$
$= \mathbf{+\infty}$ if $x = \mathbf{+\infty}$ and $y \in I$
$= \mathbf{invalid}$ if $x = \mathbf{+\infty}$ and $y = \mathbf{-\infty}$
$= \mathbf{invalid}$ if $x = \mathbf{-\infty}$ and $y = \mathbf{+\infty}$

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

$neg_I(x)$ $= result_I(-x)$ if $x \in I$
$= \mathbf{+\infty}$ if $x = \mathbf{-\infty}$
$= \mathbf{-\infty}$ if $x = \mathbf{+\infty}$

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

$sub_I(x, y) = add_I(x, neg_I(y))$

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

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

The operation $signum_I$, specified in clause 5.1.2 of part 3, shall be supplied.

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

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

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

The operations $quot_I$, mod_I, $ratio_I$, $residue_I$, $group_I$, and pad_I, are specified in clause 5.1.7 of part 2. The operations $quot_I$ and mod_I shall be provided. The operations $ratio_I$, $residue_I$, $group_I$, and pad_I should be provided.

NOTE 2 – 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 $mod_I^a = rem_I^f = mod_I$. div_I^t, mod_I^p, and rem_I^t are not recommended and should not be provided as their use may give rise to late-discovered bugs.

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:

$iec\_559_F \in$ **Boolean** (whether the datatype and operations conform to IEC 60559)

NOTE 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 D.5.2.

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

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

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

NOTE 2 – The first edition of this standard only required $p_F \geqslant 2$. The requirement in this edition guarantees that radian trigonometric operations never has denormalisation loss for angles close to zero for any conforming datatype.

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

$0 \leqslant emax_F + emin_F \leqslant 4$

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

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

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

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

$$F^* = F_{\mathrm{S}} \cup F_{\mathrm{N}} \cup F_{\mathrm{E}}$$

$$\begin{aligned} F &= F_{\mathrm{S}} \cup F_{\mathrm{N}} && \text{if } denorm_F = \textbf{true} \\ &= \{0\} \cup F_{\mathrm{N}} && \text{if } denorm_F = \textbf{false} \quad \text{(non-conforming case, see annex A)} \end{aligned}$$

F^* is the unbounded extension of F, including (in addition) all subnormal values that would be in F for $denorm_F$ being **true**.

NOTE 3 – The set F^* contains values of magnitude larger than those that are representable in the type F. F^* will be used in defining rounding.

The elements of F_{N} are called *normal* floating point values because of the constraint $r_F^{p_F-1} \leqslant i \leqslant r_F^{p_F} - 1$. The elements of F_{S}, including 0, as well as $-\mathbf{0}$ are called *subnormal* floating point values.

NOTE 4 – 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_559_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 5 – 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 D.5.2.)

5.2.1 Conformity to IEC 60559

The parameter iec_559_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_559_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_559_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 [37].

2 The IEC 60559 facilities include values for infinities and NaNs, extended comparisons, rounding towards positive or negative infinity, an inexact exception flag, and so on. See annex B for more information.

3 IEC 60559 only specifies an r_F of 2. An extended interpretation of iec_559_F for the case of an r_F of 10 can be, if the binding allows it, that the datatype and relevant operations satisfy the requirements of IEEE 854 [38].

4 If iec_559_F is **true**, then $denorm_F$ must also be **true**. Note that $denorm_F$ = **false** is non-conforming also to this standard.

5.2.2 Range and granularity constants

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

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

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

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

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

$$epsilon_F = r_F^{1-p_F} \qquad \text{(the maximum relative spacing in } F_{\mathrm{N}} \cup F_{\mathrm{E}} \text{ on either side of 0)}$$

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).

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

a) compute the exact mathematical answer (if there is any),

b) determine if notification is required,

c) round the exact answer 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 special value as the rounded result.

These stages will be modelled by two helper functions: $nearest_F$ (part of stage c) and $result_F$ (stages b and c). These helper functions are not visible to the programmer, and are not required to be part of the 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 $u_F : \mathcal{R} \to \mathcal{R}$ such that

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

NOTES

1 The value $e_F(x)$ is that of the e in the definitions of F_{S}, F_{N}, and F_{E}. When x is in $]-2 \cdot fminN_F, 2 \cdot fminN_F[$, then $e_F(x)$ is $emin_F$ regardless of x.

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

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

NOTE 3 – 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

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

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

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

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

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

is the rounding function that rounds towards positive infinity.

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 $rnd : \mathcal{R} \to F^*$, 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.

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:

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

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:

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

For other cases for either of the thee rounding functions as rnd, and for $x \in \mathcal{R}$, the following shall apply:

$$
\begin{array}{lll}
result_F(x, rnd) = rnd(x) & & \text{if } fminN_F \leqslant |x| \text{ and } |x| \leqslant fmax_F \\
 = rnd(x) & & \text{if } |x| < fminN_F \text{ and} \\
 & & \quad rnd \text{ has no denormalisation loss at } x \\
 = \textbf{underflow}(\textbf{-0}) & & \\
 & & \text{if } x < 0 \text{ and } rnd(x) = 0 \\
 = \textbf{underflow}(rnd(x)) & & \\
 & & \text{otherwise}
\end{array}
$$

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 There is no notion of underflow, but **underflow** coincides with underflow when underflow is detected as denormalisation loss.

4 When **inexact** notifications are supported, the notifications **underflow** and **overflow** implies **inexact** just as $x \neq rnd(x)$ implies **inexact**.

5 Approximation helper functions (parts 2 and 3) are to return a value in F only if that result is exact.

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

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

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

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

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

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

NOTE 6 – The handling of other special values, if available, is left unspecified by this document.

5.2.6 Floating point operations

5.2.6.1 Comparisons

For each provided floating point type, the following operations shall be provided:

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

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

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

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

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

$lss_F(x, y)$ $= \mathbf{true}$ if $x, y \in F$ and $x < y$
$= \mathbf{false}$ if $x, y \in F$ and $x \geqslant y$
$= lss_F(0, y)$ if $x = -\mathbf{0}$ and $y \in F \cup \{-\infty, -\mathbf{0}, +\infty\}$
$= lss_F(x, 0)$ if $y = -\mathbf{0}$ and $x \in F \cup \{-\infty, +\infty\}$
$= \mathbf{true}$ if $x = -\infty$ and $y \in F \cup \{+\infty\}$

	$=$ **false**	if $x = \mathbf{+\infty}$ and $y \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\}$
	$=$ **false**	if $x \in F \cup \{\mathbf{-\infty}\}$ and $y = \mathbf{-\infty}$
	$=$ **true**	if $x \in F$ and $y = \mathbf{+\infty}$
	$=$ **invalid**(**false**)	if x is a NaN or y is a NaN

$leq_F : F \times F \rightarrow$ **Boolean**

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

$gtr_F : F \times F \rightarrow$ **Boolean**

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

$geq_F : F \times F \rightarrow$ **Boolean**

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

$isnegzero_F : F \rightarrow$ **Boolean**

$isnegzero_F(x)$	$=$ **true**	if $x = \mathbf{-0}$
	$=$ **false**	if $x \in F \cup \{\mathbf{-\infty}, \mathbf{+\infty}\}$
	$=$ **invalid**(**false**)	if x is a NaN

$istiny_F : F \rightarrow$ **Boolean**

$istinyF(x)$	$=$ **true**	if ($x \in F$ and $\lvert x \rvert < fminN_F$) or $x = \mathbf{-0}$
	$=$ **false**	if ($x \in F$ and $\lvert x \rvert \geqslant fminN_F$) or $x \in \{\mathbf{-\infty}, \mathbf{+\infty}\}$
	$=$ **invalid**(**false**)	if x is a NaN

$isnan_F : F \rightarrow$ **Boolean**

$isnan_F(x)$	$=$ **false**	if $x \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\}$
	$=$ **true**	if x is a quiet NaN
	$=$ **invalid**(**true**)	if x is a signalling NaN

$issignan_F : F \rightarrow$ **Boolean**

$issignan_F(x)$	$=$ **false**	if $x \in F \cup \{\mathbf{-\infty}, \mathbf{-0}, \mathbf{+\infty}\}$
	$=$ **false**	if x is a quiet NaN
	$=$ **true**	if x is a signalling NaN

5.2.6.2 Basic arithmetic

For each provided floating point datatype, the following round to nearest operations shall be provided, and the round towards negative and positive infinity should be provided:

$add_F : F \times F \to F \cup \{\mathbf{overflow}\}$

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

$add_F^{\uparrow} : F \times F \to F \cup \{\mathbf{overflow}\}$

$$
\begin{array}{lll}
add_F^{\uparrow}(x,y) & = result_F(x+y, up_F) & \text{if } x,y \in F \\
 & = add_F(x,y) & \text{otherwise}
\end{array}
$$

$add_F^{\downarrow} : F \times F \to F \cup \{\mathbf{overflow}\}$

$$
\begin{array}{lll}
add_F^{\downarrow}(x,y) & = result_F(x+y, down_F) & \text{if } x,y \in F \\
 & = add_F(x,y) & \text{otherwise}
\end{array}
$$

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

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

$sub_F : F \times F \to F \cup \{\mathbf{overflow}\}$

$$sub_F(x,y) = add_F(x, neg_F(y))$$

$sub_F^{\uparrow} : F \times F \to F \cup \{\mathbf{overflow}\}$

$$sub_F^{\uparrow}(x,y) = add_F^{\uparrow}(x, neg_F(y))$$

$sub_F^{\downarrow} : F \times F \to F \cup \{\mathbf{overflow}\}$

$$sub_F^{\downarrow}(x,y) = add_F^{\downarrow}(x, neg_F(y))$$

NOTE 1 – $neg_F(x)$ is the same as $sub_F(\mathbf{-0}, x)$.

$abs_F : F \to F$

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

$signum_F : F \to F$

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

NOTE 2 – The first edition of this document specified the slightly different operation $sign_F$.

$mul_F : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{overflow}\}$

$$
\begin{array}{lll}
mul_F(x,y) & = result_F(x \cdot y, nearest_F) & \\
 & & \text{if } x, y \in F \text{ and } x \neq 0 \text{ and } y \neq 0 \\
 & = 0 & \text{if } x = 0 \text{ and } y \in F \text{ and } y \geqslant 0 \\
 & = \mathbf{-0} & \text{if } x = 0 \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-0}) \\
 & = \mathbf{-0} & \text{if } x = \mathbf{-0} \text{ and } y \in \text{F and } y \geqslant 0 \\
 & = 0 & \text{if } x = \mathbf{-0} \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-0}) \\
 & = 0 & \text{if } x \in F \text{ and } x > 0 \text{ and } y = 0 \\
 & = \mathbf{-0} & \text{if } x \in F \text{ and } x < 0 \text{ and } y = 0 \\
 & = \mathbf{-0} & \text{if } x \in F \text{ and } x > 0 \text{ and } y = \mathbf{-0} \\
 & = 0 & \text{if } x \in F \text{ and } x < 0 \text{ and } y = \mathbf{-0} \\
 & = \mathbf{+\infty} & \text{if } x = \mathbf{+\infty} \text{ and } ((y \in F \text{ and } y > 0) \text{ or } y = \mathbf{+\infty}) \\
 & = \mathbf{-\infty} & \text{if } x = \mathbf{+\infty} \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-\infty}) \\
 & = -\infty & \text{if } x = -\infty \text{ and } ((y \in F \text{ and } y > 0) \text{ or } y = \mathbf{+\infty}) \\
 & = \mathbf{+\infty} & \text{if } x = \mathbf{-\infty} \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-\infty}) \\
 & = \mathbf{+\infty} & \text{if } x \in F \text{ and } x > 0 \text{ and } y = \mathbf{+\infty} \\
 & = \mathbf{-\infty} & \text{if } x \in F \text{ and } x < 0 \text{ and } y = \mathbf{+\infty} \\
 & = \mathbf{-\infty} & \text{if } x \in F \text{ and } x > 0 \text{ and } y = \mathbf{-\infty} \\
 & = \mathbf{+\infty} & \text{if } x \in F \text{ and } x < 0 \text{ and } y = \mathbf{-\infty} \\
 & = no\_result2_F(x,y) & \text{otherwise}
\end{array}
$$

$mul_F^{\uparrow} : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{overflow}\}$

$$
\begin{array}{lll}
mul_F^{\uparrow}(x,y) & = result_F(x \cdot y, up_F) & \text{if } x, y \in F \text{ and } x \neq 0 \text{ and } y \neq 0 \\
 & = mul_F(x,y) & \text{otherwise}
\end{array}
$$

$mul_F^{\downarrow} : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{overflow}\}$

$$
\begin{array}{lll}
mul_F^{\downarrow}(x,y) & = result_F(x \cdot y, down_F) & \text{if } x, y \in F \text{ and } x \neq 0 \text{ and } y \neq 0 \\
 & = mul_F(x,y) & \text{otherwise}
\end{array}
$$

The operations $residue_F$ and $sqrt_F$, specified in clauses 5.2.5 and 5.2.6 of part 2, shall be provided.

NOTE 3 – The $residue_F$ operation is also known as "IEEE remainder".

$$
\begin{array}{lll}
div_F : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{invalid}\} & & \\
div_F(x,y) & = result_F(x/y, nearest_F) & \\
 & & \text{if } x,y \in F \text{ and } x \neq 0 \text{ and } y \neq 0 \\
 & = 0 & \text{if } x = 0 \text{ and } y \in F \text{ and } y > 0 \\
 & = \mathbf{-0} & \text{if } x = 0 \text{ and } y \in F \text{ and } y < 0 \\
 & = \mathbf{-0} & \text{if } x = \mathbf{-0} \text{ and } y \in F \text{ and } y > 0 \\
 & = 0 & \text{if } x = \mathbf{-0} \text{ and } y \in F \text{ and } y < 0 \\
 & = \mathbf{infinitary}(\mathbf{+\infty}) & \text{if } x \in F \text{ and } x > 0 \text{ and } y = 0 \\
 & = \mathbf{infinitary}(\mathbf{-\infty}) & \text{if } x \in F \text{ and } x < 0 \text{ and } y = 0 \\
 & = \mathbf{infinitary}(\mathbf{-\infty}) & \text{if } x \in F \text{ and } x > 0 \text{ and } y = \mathbf{-0} \\
 & = \mathbf{infinitary}(\mathbf{+\infty}) & \text{if } x \in F \text{ and } x < 0 \text{ and } y = \mathbf{-0} \\
 & = 0 & \text{if } x \in F \text{ and } x \geqslant 0 \text{ and } y = \mathbf{+\infty} \\
 & = \mathbf{-0} & \text{if } x \in F \text{ and } x \geqslant 0 \text{ and } y = \mathbf{-\infty} \\
 & = \mathbf{-0} & \text{if } ((x \in F \text{ and } x < 0) \text{ or } x = \mathbf{-0}) \text{ and } y = \mathbf{+\infty} \\
 & = 0 & \text{if } ((x \in F \text{ and } x < 0) \text{ or } x = \mathbf{-0}) \text{ and } y = \mathbf{-\infty} \\
 & = \mathbf{+\infty} & \text{if } x = \mathbf{+\infty} \text{ and } y \in F \text{ and } y \geqslant 0 \\
 & = \mathbf{-\infty} & \text{if } x = \mathbf{-\infty} \text{ and } y \in F \text{ and } y \geqslant 0 \\
 & = \mathbf{-\infty} & \text{if } x = \mathbf{+\infty} \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-0}) \\
 & = \mathbf{+\infty} & \text{if } x = \mathbf{-\infty} \text{ and } ((y \in F \text{ and } y < 0) \text{ or } y = \mathbf{-0}) \\
 & = no\_result2_F(x,y) & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
div_F^{\uparrow} : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{invalid}\} & & \\
div_F^{\uparrow}(x,y) & = result_F(x/y, up_F) & \text{if } x,y \in F \text{ and } x \neq 0 \text{ and } y \neq 0 \\
 & = div_F(x,y) & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
div_F^{\downarrow} : F \times F \to F \cup \{\mathbf{-0}, \mathbf{underflow}, \mathbf{overflow}, \mathbf{infinitary}, \mathbf{invalid}\} & & \\
div_F^{\downarrow}(x,y) & = result_F(x/y, down_F) & \text{if } x,y \in F \text{ and } x \neq 0 \text{ and } y \neq 0 \\
 & = div_F(x,y) & \text{otherwise}
\end{array}
$$

5.2.6.3 Value dissection

For each provided floating point type, the following operations shall be provided:

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

NOTES

1 Since most integer datatypes cannot represent any infinitaty (or NaN) values, documented "well out of range" finite integer values of the correct sign may here be used instead of the infinities.

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

$fraction_F : F \to F$

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

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

$$
\begin{array}{lll}
scale_{F,I}(x,n) & = result_F(x \cdot r_F^n, nearest_F) & \\
& & \text{if } x \in F \text{ and } n \in I \\
& = x & \text{if } x \in \{-\infty, -\mathbf{0}, +\infty\} \\
& = no\_result2_F(x, convert_{I \to F}(n)) & \\
& & \text{otherwise}
\end{array}
$$

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

$$
\begin{array}{lll}
succ_F(x) & = result_F(\min\ \{z \in F^* \mid z > x\}, nearest_F) & \\
& & \text{if } x \in F \\
& = succ_F(0) & \text{if } x = -\mathbf{0} \\
& = no\_result_F(x) & \text{otherwise}
\end{array}
$$

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

$$
\begin{array}{lll}
pred_F(x) & = result_F(\max\ \{z \in F^* \mid z < x\}, nearest_F) & \\
& & \text{if } x \in F \\
& = pred_F(0) & \text{if } x = -\mathbf{0} \\
& = no\_result_F(x) & \text{otherwise}
\end{array}
$$

$ulp_F : F \to F$

$$
\begin{array}{lll}
ulp_F(x) & = u_F(x) & \text{if } x \in F \\
& = ulp_F(0) & \text{if } x = -\mathbf{0} \\
& = no\_result_F(x) & \text{otherwise}
\end{array}
$$

5.2.6.4 Value splitting

For each provided floating point type, the following operations shall be provided:

$intpart_F : F \to F$

$$
\begin{array}{lll}
intpart_F(x) & = signum_F(x) \cdot \lfloor |x| \rfloor & \text{if } x \in F \\
& = x & \text{if } x \in \{-\infty, -\mathbf{0}, +\infty\} \\
& = no\_result_F(x) & \text{otherwise}
\end{array}
$$

$fractpart_F : F \to F$

$$
\begin{array}{lll}
fractpart_F(x) & = x - intpart_F(x) & \text{if } x \in F \\
& = x & \text{if } x = -\mathbf{0} \\
& = no\_result_F(x) & \text{otherwise}
\end{array}
$$

$trunc_{F,I} : F \times I \to F$

$$
\begin{array}{lll}
trunc_{F,I}(x,n) & = \lfloor x/r_F^{e_F(x)-n} \rfloor \cdot r_F^{e_F(x)-n} & \text{if } x \in F \text{ and } x \geqslant 0 \text{ and } n \in I \\
& = -trunc_F(-x,n) & \text{if } x \in F \text{ and } x < 0 \text{ and } n \in I \\
& = x & \text{if } x \in \{-\infty, -\mathbf{0}, +\infty\} \\
& = no\_result2_F(x,n) & \text{otherwise}
\end{array}
$$

$round_{F,I} : F \times I \to F \cup \{\textbf{overflow}\}$

$$
\begin{array}{lll}
round_{F,I}(x,n) & = result_F(\text{round}(x/r_F^{e_F(x)-n}) \cdot r_F^{e_F(x)-n}, nearest_F) & \\
& & \text{if } x \in F \text{ and } n \in I \\
& = x & \text{if } x \in \{-\infty, -\mathbf{0}, +\infty\} \\
& = no\_result2_F(x,n) & \text{otherwise}
\end{array}
$$

5.3 Operations for conversion between numeric datatypes

The operations specified in clause 5.4 of part 2 shall be provided for all provided conforming datatypes as well as between any conforming datatype and string formats for numerical values (regarded as integer, floating point and fixed point datatypes).

5.4 Numerals as operations in a programming language

The numerals specified in clause 5.5 of part 2 shall be supplied for all provided conforming datatypes.

6 Notification

6.1 Model handling of notifications

Notification 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 ones). A (strict) 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 above is written as $add_I : I \times I \to I \cup \{\textbf{overflow}\}$, should really be written as $add_I : (I \times \mathcal{P}(E)) \times (I \times \mathcal{P}(E)) \to (I \times \mathcal{P}(E))$, 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 $s_1 \cup s_2 \cup \{\textbf{overflow}\}$ and the first component is then the continuation value. Since being explicit about this for every specification would clutter the specifications, the specifications in ISO/IEC 10967 are implicit about this handling of exception 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), but may be more limited as specified in a binding standard.

6.2 Notification alternatives

Three alternatives for notification are provided in ISO/IEC 10967-1. The requirements are:

a) The alternative in clause 6.2.1 shall be supplied, and should be the default way of handling notifications.

b) The alternative in clause 6.2.2 should be supplied in conjunction with any language which provides support for exception handling.

c) The alternative in 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 Recording in indicators

An implementation shall provide this alternative.

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.

NOTES

1 The "set of values" may thus be "all the values computed by a thread". Not just (e.g.) "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 computations are joined, the common continuing computation must have the union (in practice often: bitwise or) of the joined computations indicators. This should be automatic, 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.

4 New threads (or smaller constructs) begin with cleared indicators. **EDITOR'S NOTE** – make normative

5 Any kind of modes or similar (e.g. the value of $big\_angle\_r_F$, specified in part 2) that affect the computations, should be propagated to new threads in a binding defined manner.

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

NOTES

6 Part 2 introduces one more notification type, **absolute_precision_underflow**, which must have its own indicator.

7 IEC 60559 requires that **inexact**ness is also recorded. **inexact** must have its own indicator.

Consider a set E including at least four elements corresponding to the four exceptional values used in this document: $\{\mathbf{invalid}, \mathbf{infinitary}, \mathbf{overflow}, \mathbf{underflow}\} \subseteq E$. Let Ind be be a type

whose values represent the subsets of E. An implementation shall provide an embedding of Ind into an existing programming language type. In addition, a method shall be provided for denoting each of the values of Ind (either as constants or via computation). An implementation is permitted to expand the set E to include additional notification indicators beyond the four listed in this document of ISO/IEC 10967.

The relevant indicators shall be set when any arithmetic operation returns exceptional values as defined in clause 5. Once set, an indicator shall be cleared only by explicit action of the program.

NOTE 8 – 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 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 ISO/IEC 10967 specifies the continuation value. The continuation value shall be implementation specified if ISO/IEC 10967 does not specify one.

NOTE 9 – The infinities and NaNs produced by an IEC 60559 system are examples of values not in F which can be used as continuation values. If the iec_559_F parameter is **true**, the continuation values must be precisely those stipulated in IEC 60559.

The following four operations shall be provided for the current indicators as well as for the indicators of accessible computations:

$clear\_indicators$: $Ind \rightarrow$
$set\_indicators$: $Ind \rightarrow$
$test\_indicators$: $Ind \rightarrow Boolean$
$current\_indicators$: $\rightarrow Ind$

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

$clear\_indicators(S)$ clear each of the indicators named in S

$set\_indicators(S)$ set each of the indicators named in S

$test\_indicators(S)$ return **true** if any of the indicators named in S is set

$current\_indicators()$ return the names of all indicators that are currently set

Indicators whose names are not in S shall not be altered.

NOTES

10 The argument to *test_indicators* tells which indicators to look at, it is not the indicator set in the argument that is tested for non-emptyness.

11 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. These calls may take additional arguments in order to refer to another accessible computation's indicators.

The implementation shall not allow a program(??) to complete successfully with an indicator that is set in the indicators of —???. 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.) It is permissible for a binding to except **underflow** (and **inexact**) from hindering the successful completion of a program(??).

6.2.2 Alteration of control flow

An implementation should provide this alternative for any language that provides a mechanism for the handling of exceptions. It is allowed (with system support) even in the absence of such a mechanism. It may be applied only to some notifications, while others are dealt with by recording in indicators.

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 Termination with message

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

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.

NOTES

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

2 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).

6.4 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. The choice of an appropriate means, such as compiler options, is left to the implementation.

The language or binding standard should specify the notification alternative to be used in the absence of a user choice. The notification alternative 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 [27] and Ada [11]
i == j      in C [17] and Fortran [22]
i .EQ. j    in Fortran [22]
(= i j)     in Common Lisp [42] and ISLisp [24]
```

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

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

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

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

Alternatively, if x was declared to be an integer, the above expression might translate to

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

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 or changes in precision.

b) Reordering of operations, such as the application of associative or distributive laws.

c) Replacing operations (or entire subexpressions) with others, such as "$2 * x$" → "$x + x$" or "x/c" → "$x * (1/c)$".

d) Evaluating constant subexpressions.

e) Eliminating unneeded subexpressions.

Only transformations which alter the semantics of an expression (the values produced, and the notifications generated) 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. Control over changes of precision is particularly useful.

8 Documentation requirements

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

> NOTE – Much of the documentation required in this clause is properly the responsibility of programming language or binding standards. An individual implementation would only need to provide details if it could not cite an appropriate clause of the language or binding standard.
>
> Some of the following items should *not* be overall standardized. See D.7 for a discussion of this topic.

a) A list of the provided integer and floating point types that conform to this document.

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

c) For each floating point type, the values of the parameters: r_F, p_F, $emin_F$, $emax_F$, $denorm_F$, and iec_559_F. (See 5.2.)

d) For each unsigned integer type I, which (if any) of the operations neg_I, abs_I, and $sign_I$ are omitted for that type. (See 5.1.2.)

e) For each floating point type F, the full definition of $result_F$. (See 5.2.4, and 5.2.5.)

f) The notation for invoking each operation provided by this document. (See 5.1.2 and 5.2.6.)

g) The translation of arithmetic expressions into combinations of operations provided by this document, including any use made of higher precision. (See clause 7.)

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

i) For each 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_559_F. (See 5.2.)

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

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

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

m) 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. (See 6.2.1.)

n) For each floating point type where iec_559_F is **true**, and for each "implementor choice" permitted by IEC 60559, the exact choice made. (See 5.2.1.)

o) For each floating point type where iec_559_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.)

,.,.,threads.,.,,.

Annex A
(normative)

Partial conformity

The requirements of ISO/IEC 10967-1 have been carefully chosen to be as beneficial as possible, yet be efficiently implemented on most existing or anticipated hardware architectures.

The bulk of ISO/IEC 10967-1 requirements 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, sub_F, mul_F, and div_F) do have implications for the underlying hardware architecture.

A small number of computer systems will have difficulty with some of the ISO/IEC 10967-1 requirements for floating point. The requirements in question are:

a) The ability to record all notifications, particularly denormalisation loss.

b) Strict 0.5-ulp accuracy of add_F, sub_F, mul_F, and div_F.

c) Round ties to even last digit.

d) A common rounding rule for add_F, sub_F, mul_F, and div_F.

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

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

As an example, the Cray family of supercomputers cannot satisfy the first five requirements above without a significant loss in performance. Machines with two's-complement floating point formats (quite rare) have difficulty with the last requirement.

Language standards will want to adopt all the requirements of ISO/IEC 10967-1 to provide programmers with the maximum benefit. However, if it is perceived that requiring full conformity to ISO/IEC 10967-1 will exclude a significant portion of that language's user community from any benefit, then specifying partial ISO/IEC 10967-1 conformity, 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 ISO/IEC 10967-1 requirements. All deviations from ISO/IEC 10967-1 conformity must 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 is 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 specified in this part.

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 for the "ordinary" syntax for these operations.

NOTES

1 In the first edition of this part, this was modelled as a normative parameter and a change of interpretation of the add_I, sub_I, mul_I, div_I^f, and div_I^t 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:

$silent\_infinitary_F$ – **true** when infinitary notifications are suppressed

NOTE 1 – Infinitary notification should only be handled by recording of indicators, if not suppressed, since other methods of notification are too disruptive.

A.3 Denormalisation loss notification relaxations

Some architectures may not be able to efficiently detect denormalisation loss, even if there are no subnormal values. Further, many architecture detect underflow instead of detecting denormalisation loss.

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

$silent\_denormalisation\_loss_F$ – **true** when denormalisation notifications are suppressed.

Underflow is when the result has an absolute value less than $fminN_F$. The "result" here may be the result before rounding or the result after an idealised rounding with not only subnormal values but also where values with smaller exponents (at least $emin_F - 1$) are included. In some cases underflow is not notified if the subnormal final result is exact, in particular an exact 0 or an exact $\mathbf{-0}$. If a binding allows detecting underflow instead of detecting denormalisation loss, there shall be a Boolean parameter available to programs:

underflow_notified$_F$ – **true** when underflow is notified. If *underflow_notified*$_F$ is **true** then *silent_denormalisation_loss*$_F$ shall be **true** unless underflow and denormalisation loss may both be notified for the same operation.

A.4 Subnormal values relaxation

If the parameter *denorm*$_F$ has a value of **false**, and thus there are no subnormal values in F except 0, then the corresponding datatype is not fully conforming to ISO/IEC 10967-1. 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.

NOTE 1 – If full conformity is required by the binding, the parameter *denorm*$_F$ is always **true** and need not be made available to programs.

The *nearest*$_F$ rounding then (exceptionally) shall return 0 for all values strictly between $-fminN_F$ and $fminN_F$, even though that is not round to nearest.

NOTES

2 The *result*$_F$ helper function will then return 0 for 0, **underflow**($\mathbf{-0}$) for values strictly between $-fminN_F$ and 0, and **underflow**(0) for values strictly between 0 and $fminN_F$.

3 If *underflow_notified*$_F$ is **true**, then **underflow** is notified instead of **underflow**.

A.5 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 ISO/IEC 10967-1). But some hardware implementations of arithmetic operations compute an approximation that loses information prior to rounding (to nearest). In some cases, it may even be so that $x + y = u + v$ may not imply $add_F(x, y) = add_F(u, v)$ (and similarly for subtract).

If this relaxation is allowed, the *rnd_error*$_F$ parameter shall have a value is $\leqslant 1$, and cannot have a value that is less than 0.5.

NOTE 1 – The *rnd_error*$_F$ parameter also signifies the maximum rounding error for multiplication and division.

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

$add^*_F(x, y)$ 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:

$add^*_F(u, v) \in F$ only if $add^*_F(u, v) = u + v$
$add^*_F(-u, -v) = -add^*_F(u, v)$ if $u, v \in F$
$add^*_F(u, v) = add^*_F(v, u)$ if $u, v \in F$
$add^*_F(u, x) \leqslant add^*_F(v, x)$ if $u, v, x \in F$ and $u < v$
$add^*_F(u, v) = u + v$ if $u, v \in F$ and $u + v \in F^*$
$add^*_F(u, v) = 0 \quad \Leftrightarrow \quad u + v = 0$
$add^*_F(u \cdot r^i_F, v \cdot r^i_F) = add^*_F(u, v) \cdot r^i_F$ if $i \in \mathcal{Z}$ and $u, v, u \cdot r^i_F, v \cdot r^i_F \in F_N$

NOTES

2 The above requirements capture the following properties:

a) add^*_F is sign symmetric.
b) add^*_F is commutative.
c) add^*_F is monotonic for the left operand (and, by commutativity, the right operand).
d) add^*_F is exact when the 'true result' is in F^*; and, by monotonicity, $add^*_F(u, v)$ is in the same "basic interval" as $u+v$. (A basic interval is the range between two adjacent F^* values.) Thus, if $max\_error\_add_F$ is 1, the max error is actuallt strictly less than 1.
e) add^*_F returns 0 exactly only when + returns 0 (and, with monotonicity, by implication add^*_F returns a correctly signed result, not avoiding underflow in and by itself.
f) add^*_F does not depend on the exponents of its arguments (but may depend on the difference of the exponents), for 'normal' arguments and results.

3 True addition (+) fulfills the requirements on add^*_F.

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

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

NOTES

4 The above requirements capture the following properties:

a) mul^*_F is sign symmetric.
b) mul^*_F is commutative.
c) mul^*_F is monotonic for the left operand (and, by commutativity, the right operand).
d) mul^*_F is exact when the 'true result' is in F^*; and, by monotonicity, $mul^*_F(u, v)$ is in the same "basic interval" as $u \cdot v$. (A basic interval is the range between two adjacent F^* values.) Thus, if $max\_error\_mul_F$ is 1, the max error is actuallt strictly less than 1.
e) mul^*_F returns 0 exactly only when $\cdot$ returns 0 (and, with monotonicity, by implication mul^*_F returns a correctly signed result, not avoiding underflow in and by itself.
f) mul^*_F does not depend on the exponents of its arguments, not even the difference in exponents, for 'normal' arguments and results.

5 True multiplication ($\cdot$) fulfills the requirements on mul^*_F.

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

$div^*_F(u,v) \in F$ only if $div^*_F(u,v) = u/v$
$div^*_F(-u,v) = -div^*_F(u,v)$ if $u,v \in F$ and $v \neq 0$
$div^*_F(u,-v) = -div^*_F(u,v)$ if $u,v \in F$ and $v \neq 0$
$div^*_F(u,x) \leqslant div^*_F(v,x)$ if $u,v,x \in F$ and $u < v$ and $x > 0$
$div^*_F(u,x) \geqslant div^*_F(v,x)$ if $u,v,x \in F$ and $u < v$ and $0 < x$
$div^*_F(x,u) \geqslant div^*_F(x,v)$ if $u,v,x \in F$ and $(u < v < 0$ or $0 < u < v)$ and $x > 0$
$div^*_F(x,u) \leqslant div^*_F(x,v)$ if $u,v,x \in F$ and $(u < v < 0$ or $0 < u < v)$ and $0 < x$
$div^*_F(u,v) = u/v$ if $u,v \in F$ and $v \neq 0$ and $u/v \in F^*$
$div^*_F(u,v) = 0 \quad \Leftrightarrow \quad u/v = 0$
$div^*_F(u \cdot r^i_F, v \cdot r^j_F) = div^*_F(u,v) \cdot r^{i-j}_F$ if $i,j \in \mathcal{Z}$ and $u,v,u \cdot r^i_F, v \cdot r^j_F \in F_N$

NOTES

6 The above requirements capture the following properties:

a) div^*_F is sign symmetric.

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

c) div^*_F is exact when the 'true result' is in F^*; and, by monotonicity, $div^*_F(u,v)$ is in the same "basic interval" as u/v. (A basic interval is the range between two adjacent F^* values.) Thus, if $max\_error\_mul_F$ is 1, the max error is actuallt strictly less than 1.

d) div^*_F returns 0 exactly only when / returns 0 (and, with monotonicity, by implication div^*_F returns a correctly signed result, not avoiding underflow in and by itself.

e) div^*_F does not depend on the exponents of its arguments, not even the difference in exponents, for 'normal' arguments and results.

7 True division (/) fulfills the requirements on div^*_F.

If this relaxation is permitted in a binding, add'_F, sub'_F, mul'_F and div'_F, (replacing add_F, sub_F, mul_F, and div_F in the binding) shall be defined as

$add'_F(x,y) = result_F(add^*_F(x,y), nearest_F)$ if $x,y \in F$
$= add_F(x,y)$ otherwise

$sub'_F(x,y) = add'_F(x, neg_F(y))$

$mul'_F(x,y) = result_F(mul^*_F(x,y), nearest_F)$ if $x,y \in F$
$= mul_F(x,y)$ otherwise

$div'_F(x,y) = result_F(div^*_F(x,y), nearest_F)$ if $x,y \in F$
$= div_F(x,y)$ otherwise

This allows addition (and subtraction and diminish) that does not round ties to even last digit (when rnd_error_F is 0.5), rounds towards zero or even rounds haphazardly (when rnd_error_F is 1), as well as allows multiplication and division that does not round ties to even last digit (when rnd_error_F is 0.5), or rounds towards zero (when rnd_error_F is 1).

If this relaxation is allowed, there shall be a parameter rnd_style_F, available to programs, having one of four constant values, is defined by

rnd_style_F	= **nearesttiestoeven**	if the relaxation is not engaged
	= **nearest**	if relaxation engaged and $rnd\_error_F = 0.5$
	= **truncate**	if $rnd\_error_F = 1$ and $\lvert add^*_F(x,y)\rvert \leqslant \lvert x+y\rvert$ and $\lvert mul^*_F(x,y)\rvert \leqslant \lvert x\cdot y\rvert$ and $\lvert div^*_F(x,y)\rvert \leqslant \lvert x/y\rvert$
	= **other**	otherwise

A.6 Comparison operations relaxation

$comparison\_via\_subtract_F$ – a Boolean parameter that is **true** when comparisons may overflow and underflow like subtraction. A binding or implementation shall document the applicable specification of the comparisons.

A.7 Sign symmetric value set relaxation

$negate\_may\_fail_F$ – a Boolean parameter that is **true** when the set of floating point values is not sign symmetric, and thus neg_F may underflow or overflow. If this relaxation is allowed, a binding or an implementation shall document the exact mathematical value set.

Annex B
(informative)

IEC 60559 bindings

When the parameter iec_559_F is **true** for a floating point type F, all the facilities required by IEC 60559 shall be provided for that type. 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.

This means that a *complete* programming language binding for LIA-1 should provide a binding for all IEC 60559 facilities as well. A programming language binding for a standard such as IEC 60559 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 should provide:

a) The name of the programming language type that corresponds to single format.

b) The name of the programming language type that corresponds to double format, if any.

c) The names of the programming language types that correspond to extended formats, if any.

For each IEC 60559 conforming type, the binding should provide:

a) A method for denoting positive infinity. (Negative infinity can be derived from positive infinity by negation).

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).

Note that the LIA-1 parameter values for IEC 60559 'single' are:

$r_F = 2$
$p_F = 24$
$emin_F = -125$
$emax_F = 128$
$denorm_F =$ **true**
$iec\_559_F =$ **true**

and for IEC 60559 'double' are:

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

$emax_F = 1024$
$denorm_F =$ **true**
$iec\_559_F =$ **true**

it just specifies, directly or indirectly, the values):

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

a) add_F, sub_F, mul_F, and div_F. (Also required by LIA-1.)

b) Remainder ($residue_F$), square-root ($sqrt_F$), and round-to-integral-value ($rounding_F$). (Also required by LIA-1, except for $rounding_F$, but specified in LIA-2.)

c) The type conversions $convert_{F_a \to F_b}$, $convert_{F \to I}$, $convert_{I \to F}$. (Also required by LIA-1, but are specified in LIA-2.)

d) Type conversions between the floating point values and decimal strings (both ways, required by LIA-1, but are specified in LIA-2).

e) The comparisons eq_F, neq_F, lss_F, leq_F, gtr_F, and geq_F. (Also required by LIA-1.)

f) The comparison "unordered". (Optional in IEC 60559.)

g) The other 19 comparison operations. (Optional in IEC 60559.)

h) The "recommended functions" *copysign* (can be expressed as $mul_F(abs_F(x), signum_F(y))$), *negate* ($neg_F$), *scaleb* ($scale_{F,I}$), *logb* ($exponent_{F,I}$), *nextafter* (LIA-1 instead specifies $succ_F$ and $pred_F$), *finite*, *isnan* ($isnan_F$), <>, and *class*. (Each is optional in IEC 60559. Negate, scaleb, logb, and nextafter are redundant with existing LIA-1 operations.)

The binding should provide the ability to read and write the following components of the floating point environment (modes or flags):

a) The rounding mode.

b) The five exception flags: inexact, underflow, overflow, divide_by_zero (in LIA called infinitary), and invalid.

c) The disable/enable flags for each of the five exceptions. (Optional in IEC 60559.)

d) The handlers for each of the exceptions. (Optional in IEC 60559.)

The binding should provide Boolean parameters for each implementor choice allowed by IEC 60559:

a) Whether trapping is implemented.

b) Whether tinyness (underflow) is detected "before rounding" or "after rounding".

c) Whether loss-of-accuracy is detected as a denormalization loss or as an inexact result.

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

B.2 Notification

One appropriate way to access the five IEC 60559 exception flags is to use the functions defined in 6.2.1. This requires extending the set E with one new value: **inexact**. (Such an extension is expressly permitted by 6.2.1.)

Designing a binding for the optional "trapping" facility should be done in harmony with the exception handling features already present in the programming language. It is possible that existing language features are sufficient to meet programmer's needs.

B.3 Rounding

The two directed roundings of IEC 60559, round-toward-positive infinity and round-toward-negative-infinity, do not satisfy the sign symmetry requirement of 5.2.4. However, the default IEC 60559 rounding does satisfy LIA-1 requirements.

To use the directed roundings, a programmer would have to take explicit action to change the current rounding mode. At that point, the program is operating under the IEC 60559 rules, not the LIA-1 rules. Such non-conforming modes are expressly permitted by clause 2.

The directed roundings are useful for implementing interval arithmetic. However, at a higher level it is better to use a special datatype for intervals, and arithmetic on intervals. (In case such a datatype will be specified by LIA, in a new part, the $result_F$ helper function is prepared for directed roundings.)

Annex C
(informative)

Requirements beyond IEC 60559

Any computing system providing floating point datatypes conforming to the requirements of IEC 60559 can economically let those datatypes conform to LIA-1 as well. This annex outlines the LIA-1 requirements that go beyond the requirements of IEC 60559.

For each floating point type F, the following parameters or derived constants must be provided to the program:

> p_F, r_F, $emin_F$, $emax_F$, $denorm_F$ (if it is allowed to have the value **false**), iec_559_F, $fmax_F$, $fmin_F$, $fminN_F$, $epsilon_F$, rnd_error_F (if it is allowed to have a value other than 0.5), and rnd_style_F (if it is allowed to have a value other than **nearesttiestoeven**; see annex A)

The following operations must be provided (typically in software):

> neg_F, abs_F, $signum_F$ (specified in LIA-3), $exponent_F$, $fraction_F$, $scale_F$, $succ_F$, $pred_F$, ulp_F, $intpart_F$, $fractpart_F$, $trunc_F$, and $round_F$

A method for notification must be provided that conforms to the applicable programming language standard. (This is independent of LIA-1 per se, since any implementation of a standard language must conform to that language's standard.)

When the language (or binding) standard does not specify a notification method, 6.2.1 requires that notification be done by setting "indicators" which reflect the status flags required by IEC 60559. (See annex B as well.)

Subclause 6.2.3 specifies a way for the programmer to demand prompt program termination on the occurrence of an LIA-1 notification. This is typically implemented using IEC 60559 trapping, or (if trapping is unavailable) by compiler generated code.

> NOTE – LIA-1 notifications correspond to the IEC 60559 exceptions overflow, underflow, divide_by_zero, and invalid.

If any status flags are set at program termination, this fact must be reported to the user of the program.

Thorough documentation must be provided as outlined in clause 8. Citing IEC 60559 will be sufficient for several of the documentation requirements, including requirements (c) and (e). Note that the implementor choices permitted by IEC 60559 must be documented.

Annex D
(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.

D.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* 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's running on is suitable for its needs.

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

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

D.1.2 Exclusions

There are many arithmetic systems, such as fixed point arithmetic, significance arithmetic, interval arithmetic [68], 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 Kulish and Miranker [60, 61] 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, and other issues of expression semantics, are not addressed directly by LIA-1. However, suitable documentation is required (see clause 7).

D.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 and conversion routines which

a) are realistic from an implementation point of view,

b) have acceptable performance, and

c) have adequate accuracy to support numerical analysis.

For operations on complex number datatypes, accuracy specifications comparable to those in LIA-1 are certainly feasible, but may have unacceptable performance penalties.

D.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 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 [37]) 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 F.

D.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 [65] 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 [50], which includes enhancements based upon experience with Schryer's work [65].

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* [33] 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.

D.3 Normative references

The referenced IEC 60559 standard is identical to the IEEE 754 standard and the former IEC 559 standard.

D.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.

There are various notations for giving a formal definition of arithmetic. In [69] a formal definition is given in terms of the Brown model [48]. Since the LIA-1 model differs from the Brown model, the definition in [69] is not appropriate for LIA-1.

D.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. Hence numbers such as π, $1/3$ and $\sqrt{2}$ are in $\mathcal{R}$, but usually they cannot be represented exactly in a computer.

D.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 of indicators'. Another possibility is to invoke a change in the flow control of a program (for example, an Ada "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 **underflow** it is usually ill-advised to make a change in control flow, likewise for **infinitary** 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 tt integer-overflow and tt floating-overflow, **infinitary** for integer results may result in an actual notification, while **infinitary** on floating results are handled quietly, only returning the infinitary continuation value while setting the indicator for **infinitary**.

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 $\mathcal{R}$. The rounding function rnd maps each real number u to an approximation of u that lies in X. If a real number u is in X, then clearly u is the best approximation for itself, so $rnd(u) = u$. If u is between two adjacent values x_1 and x_2 in X, then one of these adjacent values must be the approximation for u:

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

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

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

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 $rnd(u)$ depends only on u and not on the arithmetic operation (or operands) that gave rise to u.

The graph of a rounding function looks like a series of steps. As u increases, the value of $rnd(u)$ 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 D.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 D.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.

D.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. 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. However, sometimes a language standard requires that a type parameter has a known value (for example, that an integer type is bounded). In this case, the parameter must have the same value in every implementation of that language and therefore need not be provided as a run-time parameter.

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. 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 $-fmax_I$ and $fmax_I$ 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**).

D.5.1 Integer datatypes and operations

Most traditional computer programming languages assume the existence of bounds on the range of integers which can be data values. Some languages place no limit on the range of integers, or even allow the boundedness of the 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.

D.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 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.

D.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, 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.

D.5.1.0.3 Modulo integers

Modulo integers were introduced as a partially conforming case because there are languages that mandate wrapping 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 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 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.

D.5.1.0.4 Modulo integers versus overflow

$wrap_I$ (LIA-2) produces results in the range $[minint_I, maxint_I]$. These results are positive for unsigned integer types, but may be negative for signed types.

D.5.1.1 Integer result function

Integer result functions, $result_I$, 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.

D.5.1.2 Integer operations

D.5.1.2.1 Comparisons

The comparisons are always exact and never produce any notification.

D.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-2
$div^t_I(-3, 2) = -1$	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 $group_I$ all satisfy a broadly useful translation invariant:

$quot_I(x + i * y, y) = quot_I(x, y) + i$ if $y \neq 0$, 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_I^t (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_I^t (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 for against the use of div_I^t, and for the use of the other integer division operations.

mod_I (specified in LIA-2) gives the remainder after division. It is coupled to division by the following identities:

$x = quot_I(x, y) * y + mod_I(x, y)$	if $y \neq 0$, and no overflow occurs
$y < mod_I(x, y) \leqslant 0$	if $y < 0$
$0 \leqslant mod_I(x, y) < y$	if $y > 0$

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. Note that computing $mod_I(x, y)$ 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.

D.5.2 Floating point datatypes and operations

Floating point values are traditionally represented as either zero or

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

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.

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 value of the fraction g is $1/r_F$ and the maximum value is $1 - r_F^{-p_F}$.

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 *subnormal* values; **false** if not.

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.

$denorm_F$ must be **true** for a fully conforming datatype. It is allowed to be **false** only for partially conforming datatypes.

The IEEE standards 754 [37] and 854 [38] 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), $f_0 \neq 0$, and e is an integer exponent between $emin_F - 1$ and $emax_F - 1$. The minimum value of the significand is 1; the maximum value is $r_F - 1/r_F^{p_F-1}$. 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.

D.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. 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.

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 specified in IEEE 854 [38] and also 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.

D.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 $fminN_F$.

In order to accommodate radix-complement floating point, LIA-1 would have to

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);

b) add **overflow** to the signature of neg_F (because neg_F evaluated on the most negative number will now overflow);

c) add **overflow** to the signature of abs_F (because abs_F will now overflow when evaluated on the most negative number);

d) perhaps add **underflow** to the signature of neg_F, if $-fminN_F$ is omitted;

e) expand the definitions of sub_F and $trunc_F$ to ensure that these operations behave correctly.

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.

D.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.

IEC 60559 contains an annex listing a number of recommended functions. While not required, implementations of LIA-1 are encouraged to provide those functions.

D.5.2.1.1 Subnormal numbers

The IEEE standards 754 and 854 datatypes and some non-IEEE floating point datatypes include subnormal numbers. Logically, also 0 and $\mathbf{-0}$, are subnormal numbers, though not formally included. 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 $[1, r_F^{p_F - 1} - 1]$. The corresponding fraction g lies in the interval $[r_F^{-p_F}, 1/r_F - r_F^{-p_F}]$; its most significant digit is zero. Subnormal numbers partially fill the "underflow gaps" in $fminN_F$ that occur between $\pm r_F^{emin_F - 1}$ and 0. Taken together, and with 0, they comprise the set F_D.

The values in F_D 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 can vary widely. This relative error varies from $epsilon_F = r_F^{1-p_F}$ at the high ends of F_D to 1 at the low ends of F_D. Near 0, in $[-fminD_F, fminD_F]$, the relative error may be unboundedly large.

Whenever an addition or subtraction produces a result in F_D, 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_D.

The entire set of floating point numbers F is either $F_N \cup F_D$ (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. See Coonen [49] for a detailed discussion of the properties of subnormal numbers.

D.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. use of the IEEE *copysign* function, 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.

D.5.2.1.3 Infinities and NaNs

The IEEE standards 754 [37] and 854 [38] provide special values to represent *infinities* and *Not-a-Numbers*. *Infinity* represents a large value beyond measure, either as an exact quantity (from dividing a finite number by zero) or as the result of untrapped overflow. A *NaN* represents an indeterminate, and hence invalid, quantity (e.g. from dividing zero by zero).

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.

D.5.2.2 Range and granularity constants

The positive real numbers $fmax_F$, $fmin_F$, and $fminN_F$ are interesting boundaries in the set F. $fmax_F$ is the "overflow threshold". It is the largest value in F (and thereby also F_N). $fminN_F$ is the value of smallest magnitude in F. $fminN_F$ is the "subnormal threshold" or the "underflow 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 $[-fmax_F, -fmin_F]$ and $[fmin_F, fmax_F]$, and values in the ranges $[-fmax_F, -fminN_F]$ and $[fminN_F, fmax_F]$ 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 $fmin_F = fminN_F$.

The derived constant $epsilon_F$ must also be accessible to an executing program:

$$epsilon_F = r_F^{1-p_F}$$

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 g, $1/r_F$. So $epsilon_F$ can be described as the largest relative representation error for the set of normalized values in F_N.

An alternate definition of $epsilon_F$ currently in use is the smallest floating point number such that the expression `1 +` $epsilon_F$ yields a value greater than 1. This definition is flawed because it

depends on the characteristics of the rounding function. For example, on an IEEE floating point datatype with round-to-positive-infinity, $epsilon_F$ would be $fminD_F$.

D.5.2.2.1 Relations among floating point datatypes

An implementation may provide more than one floating point datatype, and most current systems do. It is usually possible to order those with a given radix as $F_1, F_2, F_3, \cdots$ such that

$$p_{F_1} < p_{F_2} < p_{F_3} \cdots$$
$$emin_{F_1} \geqslant emin_{F_2} \geqslant emin_{F_3} \cdots$$
$$emax_{F_1} \leqslant emax_{F_2} \leqslant emax_{F_3} \cdots.$$

A number of current systems do not increase the exponent range with precision. However, the following constraints

$$2 \cdot p_{F_i} \leqslant p_{F_{i+1}}$$
$$2 \cdot (emin_{F_i} - 1) \geqslant (emin_{F_{i+1}} - 1)$$
$$2 \cdot emax_{F_i} \leqslant emax_{F_{i+1}}$$

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.

D.5.2.3 Approximate operations

Let's apply a three stage model to multiplication ($mul_F(x, y)$):

a) First, compute the perfect result, $x * y$, as an element of $\mathcal{R}$.

b) Second, modify this to form a rounded result, $rnd_F(x * y)$, as an element of F^*.

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 are in F:

$$mul_F(x, y) = result_F(x \cdot y, rnd_F) \quad \text{if } x, y \in F$$

The $result_F$ function is defined to compute $rnd_F(x \cdot y)$ internally, since the result depend on the properties of the rounding function itself, 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 and denormalisation loss (or underflow).

The helper functions rnd_F, $result_F$, are the same for all the operations of a given floating point type. Similarly, the constant rnd_error_F does not differ between operations.

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.

D.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 or denormalisation loss (or underflow), finally the exact value is rounded to the appropriate precision and a continuation value is determined.

The round to nearest rule is specified by a rounding function $nearest_F$, which maps values in $\mathcal{R}$ onto values in F^*. F^* is the set $F_N \cup F_D$ 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^*, 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 requirement of "sign symmetry", $nearest_F(-x) = -nearest_F(x)$, is needed to assure the arithmetic operations add_F, sub_F, mul_F, and div_F have the expected behaviour with respect to sign, as described in D.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 constants are provided 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.

What are the most common rounding rules for existing floating point datatypes and operations?

IEEE 754 [37] and 854 [38] define four rounding rules. In addition, a fifth rounding rule is in common use. Hence, a useful list is as follows:

a) *Round toward minus infinity*

b) *Round toward plus infinity*

c) *Round toward zero*

d) *IEEE round to nearest*: In the case of a value exactly half-way between two neighbouring values in F_N, select the "even" result. That is, for x in F_N and $u = r_F^{e_F(x)-p_F}$

$$\begin{aligned} rnd(x + \tfrac{1}{2}u) &= x + u && \text{if } x/u \text{ is odd} \\ &= x && \text{if } x/u \text{ is even} \end{aligned}$$

This is the default rounding mode in the IEEE standards.

e) *"Traditional" round to nearest*: In the case of a half-way value, round away from zero. That is, if x and u are as above, then

$$rnd(x + \tfrac{1}{2}u) = x + u$$

The first two of these rounding rules do not have sign symmetry, but the last three do.

The first three rules give a one ulp error bound. The last two give a half ulp bound. Most non-IEEE implementations provide either the third rule or the last rule.

D.5.2.5 Floating point result function

The rounding function $nearest_F$ produces unbounded values. A result function is then used to check whether this 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 $rnd(x)$. If $rnd(x)$ is too large or too small to be represented, then $result_F$ returns **overflow** or **underflow** respectively.

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**. On the other hand, if there was no denormalization loss, then the implementation is free to return either **underflow** (causing a notification) or $rnd(x)$. Note that IEEE 754 allows some implementation flexibility in precisely this case. See the discussion of "continuation value" in 6.2.1.

$result_F(x, rnd)$ 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 rather than the individual value $rnd(x)$. (In addition, the continuation value upon overflow, depends on the rounding function.)

D.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. See [70] for a sample portable implementation in Pascal. However, portable versions of these operations will not be as efficient as those which an implementation provides and "tunes" to the architecture.

D.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.

D.5.2.6.2 Basic arithmetic

a) The operations add_F, sub_F, mul_F and div_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).

D.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, 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".

Together, the $succ_F$ and $pred_F$ operations correspond to the IEEE 754 recommended function *nextafter*. These 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 of a non-zero argument. The operation is undefined if the argument is zero.

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 on subnormal and zero ones. $exponent_F(x)$ 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 $emin_F$.

D.5.2.6.4 Value splitting

a) The operation $trunc_F$ 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 $round_F$ 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.

   $round_F$ is not intended to provide access to machine rounding.

c) The operation $intpart_F$ isolates the integer part of the argument, and returns this result in floating point form.

d) The operation $fractpart_F$ returns the value of the argument minus its integer part (obtained by $intpart_F$).

D.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 [37] takes the highly prescriptive approach, allowing relatively little latitude for variation. It even stipulates much of the representation. The Brown model [48] 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 $x < z$.

D.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.

In the following discussion, let u, v, x, and y be elements of F, and let j, k, and n be integers.

The seven operations add_F, sub_F, mul_F, div_F, $scale_F$, $convert_{F' \to F}$, and $convert_{I \to F}$ compute approximations to the ideal mathematical functions. All the other operations defined in LIA-1 produce exact results in the absence of notifications.

Since the seven approximate operations are all so similar, it is convenient to give a series of rules that apply to all of the seven (with some qualifications). Let Φ be any of the given operations, and let ϕ be the corresponding ideal mathematical function. In what follows, 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 occurs,

$$u \leqslant \phi(x, y) \leqslant v \;\Rightarrow\; u \leqslant \Phi(x, y) \leqslant v \tag{I}$$

when $u, v \in F$.

When $\phi(x, y)$ is defined for $x, y \in F$, and no notification occurs,

$$\phi(x, y) \in F \;\Rightarrow\; \Phi(x, y) = \phi(x, y) \tag{II}$$

When $\phi(u, x)$ and $\phi(v, y)$ are defined for $x, y, u, v \in F$, and no notification occurs,

$$\phi(u, x) \leqslant \phi(v, y) \;\Rightarrow\; \Phi(u, x) \leqslant \Phi(v, y) \tag{III}$$

When $\phi(x, y)$ is defined for $x, y \in F$, non-zero, and no notification occurs,

$$|\Phi(x, y) - \phi(x, y)| \leqslant u_F(\phi(x, y)) \leqslant u_F(\Phi(x, y)) \tag{IV}$$

When $\phi(x, y)$ is defined for $x, y \in F$, is in one of the ranges $[-fmax_F, -fminN_F]$ or $[fminN_F, fmax_F]$, and no notification occurs,

$$\left| \frac{\Phi(x,y) - \phi(x,y)}{\phi(x,y)} \right| \leqslant ulp_F(1) = epsilon_F \tag{V}$$

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 one of the ranges $[-fmax_F, -fminN_F]$ or $[fminN_F, fmax_F]$ or is 0, and no notification occurs, for some $n \in \mathcal{Z}$

$$\phi(x \cdot r_F^j, y \cdot r_F^k) = \phi(x, y) \cdot r_F^n \;\Rightarrow\; \Phi(x \cdot r_F^j, y \cdot r_F^k) = \Phi(x, y) \cdot r_F^n \tag{VI}$$

Rules (I) through (VI) apply to the seven approximate operations add_F, sub_F, mul_F, div_F, $scale_F$, $convert_{F' \to F}$, and $convert_{I \to F}$.

Rules (I) through (VI) also apply to the "exact" operations, but they don't say anything of interest.

Here are some identities that apply to specific operations, when no notification occurs:

$add_F(x, y) = add_F(y, x)$

$mul_F(x, y) = mul_F(y, x)$

$sub_F(x, y) = neg_F(sub_F(y, x))$

$add_F(neg_F(x), neg_F(y)) = neg_F(add_F(x, y))$

$sub_F(neg_F(x), neg_F(y)) = neg_F(sub_F(x, y))$

$mul_F(neg_F x), y) = mul_F(x, neg_F(y)) = neg_F(mul_F(x, y))$

$div_F(neg_F(x), y) = div_F(x, neg_F(y)) = neg_F(div_F(x, y))$

For $x \neq 0$,

$x \in F_N \;\Rightarrow\; exponent_F(x) \in [emin_F, emax_F]$

$x \in F_D \;\Rightarrow\; exponent_F(x) \in [emin_F - p_F + 1, emin_F - 1]$

$r_F^{exponent_F(x)-1} \in F$

$r_F^{exponent_F(x)-1} \leqslant |x| < r_F^{exponent_F(x)}$

$fraction_F(x) \in [1/r_F, 1[$

$scale_F(fraction_F(x), exponent_F(x)) = x$

$scale_F(x, n)$ 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

$$\begin{array}{ll} r_F^{p_F-1} \leqslant i < r_F^{p_F} & \text{if } x \in F_N \\ 1 \leqslant i < r_F^{p_F-1} & \text{if } x \in F_D \end{array}$$

$exponent_F(x) = exponent_F(y) \;\Rightarrow\; ulp_F(x) = ulp_F(y)$

$x \in F_N \;\Rightarrow\; ulp_F(x) = epsilon_F * r_F^{exponent_F(x)-1}$

Note that if $denorm_F =$ **true**, ulp_F is defined on all floating point values. If $denorm_F =$ **false** (not fully conforming to LIA-1), ulp_F underflows 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,

$$succ_F(x) = x + ulp_F(x)$$

$$\begin{aligned} pred_F(x) &= x - ulp_F(x) && \text{if } x \text{ is not } r_F^n \text{ for any integer } n \geqslant emin_F \\ &= x - ulp_F(x)/r_F && \text{if } x \text{ is } r_F^n \text{ for some integer } n \geqslant emin_F \end{aligned}$$

$$ulp_F(x) \cdot r_F^{p_F - n} = r_F^{e_F(x) - n} \quad \text{for any integer } n$$

For any x and any integer $n > 0$, when no notification occurs,

$$r_F^{exponent_F(x)-1} \leqslant |trunc_F(x, n)| \leqslant |x|$$

$$\begin{aligned} round_F(x, n) &= trunc_F(x, n), \quad \text{or} \\ &= trunc_F(x, n) + signum_F(x) \cdot ulp_F(x) \cdot r_F^{p_F - n} \end{aligned}$$

D.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. 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. Another part of this standard, *Information technology – Language independent arithmetic – Part 2: Elementary numerical functions*, deals with items (c), and (d).

D.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 or sub_F operation. However, a subnormal result for a mul_F or div_F operation usually is not exact, which introduces an error of at most one half ulp. Because of the loss of significant digits, the relative error due to rounding exceeds that for rounding a 'normal' result. Hence accuracy of a subnormal result for a mul_F or div_F operation is usually lower than that for a 'normal' result.

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 D.5.2.9.3.

D.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.

D.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:

$$z = add_F(x, y) = (x + y) + rounding\_error$$

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 $0.5 \cdot ulp_F(z)$. 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. It is likely that the estimates for the final errors will be unduly pessimistic because the signs of the various errors are usually unknown. Thus, at each stage the worst case combination of signs and magnitudes in the errors must be assumed.

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.

D.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.

D.5.3 Conversion operations

These are borrowed directly from LIA-2.

D.6 Notification

The essential goal of the notification process is that it should not be possible for a program(???) to terminate with an unresolved arithmetic violation unless the user has been informed of that fact, since the results of such a program may be unreliable.

D.6.1 Model handling of notifications

D.6.2 Notification alternatives

LIA-1 provides a choice of notification mechanisms to fit the requirements of various programming languages. The first alternative (recording of 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 such notifications". Language or binding standards are expected to choose one of these two as their primary notification mechanism. The recording of indicators mechanism must be provided, and should be the default handling.

The third alternative (termination of program(??) with message) is provided for use in two situations: (a) when the programmer has not (yet) programmed any exception handling code for the alteration of control flow alternative, and (b) when a user wants to be immediately informed of any exception.

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 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.

D.6.2.1 Recording of 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 indicator
invalid	**invalid**
overflow	**overflow**
underflow	**underflow**
division by zero	**infinitary**
inexact	(no counterpart in LIA)
(no counterpart)	**absolute_precision_underflow** (LIA-2 and LIA-3)

LIA-1 does not include notification for **inexact** because non-IEEE implementations are unlikely to detect inexactness of floating point results. However, if $ieee_F$ is **true** the notification **inexact** must be provided. This one should be handled by recording of indicators by default, regardless of how other notifications are handled.

For a zero divisor, IEEE specifies an **invalid** exception if the dividend is zero, and a **division by zero** (**infinitary**in LIA) otherwise. 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, a binding may map both notification types to the same actual notification.

Different notification types need not be handled the same. E.g. **inexact** and **underflow** should be handled by recording of indicators, or even be ignored if a binding so specifies, regardless of how other notifications are handled.

An implementation must check the recording before successfully terminating the program(??). Merely setting a status flag is not regarded as adequate notification, since this action is too easily ignored by the user and could thus damage the integrity of a program by leaving the user unaware that an unresolved arithmetic notification occurred. Hence LIA-1 prohibits successful completion of a program(??) if any status flag is set. Implementations can provide system software to test all status flags at completion of a program(??), and if any flag is set, provide a message.

The mechanism of recording of indicators proposed here 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 its own set of these indicators, 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).

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.

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.

D.6.2.2 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 which allows the user 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 must be an available alternative.

Although this alternative is expressed in terms of control flow, clause 2 gives binding standards the power to select the exception handling mechanisms most natural for the programming language in question. For example, a functional programming language might extend each of its types with special "error" values. In such a language, the natural notification mechanism would be to produce error values rather than to alter control flow.

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.

D.6.2.3 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.

The message must be "hard to ignore". It must be delivered in such a way that there is no possibility that the user will be unaware that the program was terminated because of an unresolved exception. For example, the message could be printed on the standard error output device, such as the user's terminal if the program is run in an interactive environment.

D.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.

D.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 6.2.2 may differ substantially from the optimal code for 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. Another non-conforming mode could be one in which the final check on the notification indicators was suppressed.

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.

D.7 Relationship with language standards

Language standards vary in the degree to which the underlying datatypes are specified. For example, Pascal [27] 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 large input values. 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 D.5.2.0.1 or D.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 floating point type, will frequently vary from implementation to implementation. 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 E 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. On the other hand to allow freedom in the implementation, we recommend that the following information not be standardized, but should be documented by the implementation:

a) The values of $maxint_I$ and $minint_I$ should not be standardized.

   However, it is reasonable to standardize whether a particular integer type is signed, and to give a lower bound on the value of $maxint_I$.

b) The values of r_F, p_F, $emin_F$, $emax_F$, and iec_559_F should not be standardized.

   However, it is reasonable to give upper bounds on $epsilon_F$ ($r_F^{1-p_F}$), and bounds on the values of $emin_F$ and $emax_F$. Certain languages provide decimal floating point types which require $r_F = 10$.

c) The semantics of rnd_F and $result_F$ should not be further standardized.

   That is, no further standardization beyond what is already required by LIA-1, since this would limit the range of hardware platforms that could support efficient implementations of the language.

d) The behaviour of $nearest_F$ on ties should be standardized.

e) The IEC 60559 implementor choices should not be limited (except by future revisions of IEC 60559).

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 functions rnd_F, $result_F$, e_F, and u_F may prove useful in defining additional arithmetic operations.

D.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 F.

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 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.)

Annex E
(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 D.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, 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 D.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.

If a language or binding standard wishes to make the selection of the notification method portable, but has no syntax for specifying such a selection, we suggest the use of one of the commonly used methods for extending the language such as special comment statements in Fortran or `pragma`s in C and Ada.

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.

E.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 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 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**.

Every implementation of Ada has at least one integer datatype. The notation INT is used to stand for the name of any one of these datatypes in what follows.

The LIA-1 parameters for an integer datatype can be accessed by the following syntax:

$maxint_I$	INT`'Last`	$\star$
$minint_I$	INT`'First`	$\star$

The parameter $bounded_I$ is always **true**, and the parameter $hasinf_I$ is always **false**, and they need therefore not be provided to programs. 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.

The LIA-1 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	$\star$
$add_I(x, y)$	x `+` y	$\star$
$neg_I(x)$	`-` x	$\star$
$sub_I(x, y)$	x `-` y	$\star$
$abs_I(x)$	`abs` x	$\star$
$signum_I(x)$	`Signum(`x`)`	†
$mul_I(x, y)$	x `*` y	$\star$
$quot_I(x, y)$	`Quotient(`x`,` y`)`	†
$mod_I(x, y)$	x `mod` y	$\star$
$truncdiv_I(x, y)$	x `/` y (dangerous syntax)	$\star$ (bad sem., not LIA-1!)
$truncrem_I(x, y)$	x `rem` y	$\star$ (bad sem., not LIA-1!)

where x and y are expressions of type INT.

Every implementation of Ada has at least one floating point datatype. The notation FLT are used to stand for the name of any one of these datatypes in what follows.

The LIA-1 parameters for a floating point datatype can be accessed by the following syntax:

r_F	FLT`'Machine_Radix`	$\star$
p_F	FLT`'Machine_Mantissa`	$\star$
$emax_F$	FLT`'Machine_Emax`	$\star$
$emin_F$	FLT`'Machine_Emin`	$\star$
$denorm_F$	FLT`'Denorm`	$\star$
$hasnegzero_F$	FLT`'Signed_Zeroes`	$\star$ (not LIA-1)
$hasinf_F$	FLT`'Has_Infinities`	† (not LIA-1)

iec_559_F	FLT'`IEC60559`	†

The LIA-1 derived constants for a floating point datatype can be accessed by the following syntax:

$fmax_F$	FLT'`Last`	⋆
$fminN_F$	FLT'`Fmin_Norm`	†
$fmin_F$	FLT'`Fmin`	†
$epsilon_F$	FLT'`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`	†

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`)`	†
$add_F(x, y)$	x `+` y	⋆
$neg_F(x)$	`-` x	⋆
$sub_F(x, y)$	x `-` y	⋆
$abs_F(x)$	`abs` x	⋆
$signum_F(x)$	`Signum(`x`)`	†
$mul_F(x, y)$	x `*` y	⋆
$sqrt_F(x)$	`Sqrt(`x`)`	⋆
$residue_F(x, y)$	FLT'`Remainder(`x`, `y`)`	⋆
$div_F(x, y)$	x `/` y	⋆
$exponent_{F,I}(x)$	FLT'`Exponent(`x`)`	⋆ (dev.: 0 if $x = 0$)
$fraction_F(x)$	FLT'`Fraction(`x`)`	⋆
$scale_{F,I}(x, n)$	FLT'`Scaling(`x`, `n`)`	⋆
$succ_F(x)$	FLT'`Adjacent(`x`, `FLT'`Last)`	⋆(dev. at $fmax_F$)
$pred_F(x)$	FLT'`Adjacent(`x`, -`FLT'`Last)`	⋆(dev. at $-fmax_F$)
$ulp_F(x)$	FLT'`Unit_Last_Place(`x`)`	†

$intpart_F(x)$	FLT'`Truncation(`x`)`	$\star$
$fractpart_F(x)$	x `-` FLT'`Truncation(`x`)`	$\star$
$trunc_{F,I}(x,n)$	FLT'`Leading_Part(`x`,` n`)`	$\star$(**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.

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.

$convert_{I\rightarrow I'}(x)$	*INT2*`(`x`)`	$\star$
$convert_{I''\rightarrow I}(s)$	`Get(`s`,` n`,` w`);`	$\star$
$convert_{I''\rightarrow I}(f)$	`Get(`f`?,` n`,` w`?);`	$\star$
$convert_{I\rightarrow I''}(x)$	`Put(`s`,` x`,` *base*`?);`	$\star$
$convert_{I\rightarrow I''}(x)$	`Put(`h`?,` x`,` w`?,` *base*`?);`	$\star$
$floor_{F\rightarrow I}(y)$	INT`(`FLT'`Floor(`y`))`	$\star$
$rounding_{F\rightarrow I}(y)$	INT`(`FLT'`Unbiased_Rounding(`y`))`	$\star$
$ceiling_{F\rightarrow I}(y)$	INT`(`FLT'`Ceiling(`y`))`	$\star$
$convert_{I\rightarrow F}(x)$	FLT`(`x`)`	$\star$
$convert_{F\rightarrow F'}(y)$	*FLT2*`(`y`)`	$\star$
$convert_{F''\rightarrow F}(s)$	`Get(`s`,` n`,` w`?);`	$\star$
$convert_{F''\rightarrow F}(f)$	`Get(`f`?,` n`,` w`?);`	$\star$
$convert_{F\rightarrow F''}(y)$	`Put(`s`,` y`, Aft=>`a`?, Exp=>`e`?);`	$\star$
$convert_{F\rightarrow F''}(y)$	`Put(`h`?,` y`, Fore=>`i`?, Aft=>`a`?, Exp=>`e`?);`	$\star$
$convert_{D\rightarrow F}(z)$	FLT`(`z`)`	$\star$
$convert_{D'\rightarrow F}(s)$	`Get(`s`,` n`,` w`?);`	$\star$
$convert_{D'\rightarrow F}(f)$	`Get(`f`?,` n`,` w`?);`	$\star$
$convert_{F\rightarrow D}(y)$	FXD`(`y`)`	$\star$
$convert_{F\rightarrow D'}(y)$	`Put(`s`,` y`, Aft=>`a`?, Exp=>0);`	$\star$
$convert_{F\rightarrow D'}(y)$	`Put(`h`?,` y`, Fore=>`i`?, Aft=>`a`?, Exp=>0);`	$\star$

where x is an expression of type INT, y is an expression of type FLT, and z is an expression of type FXD, where FXD is a fixed point type. *INT2* is the integer datatype that corresponds to I'. *FLT2* is the floating point datatype that corresponds to F'. A ? above indicates that the parameter is optional. f is an opened input file (default is the default input file). h is an opened output file (default is the default output file). s is of type `String` or `Wide_String`. For `Get` of a floating point or fixed point numeral, the base is indicated in the numeral (default 10). For `Put` of a floating point or fixed point numeral, only base 10 is required to be supported. For details on `Get` and `Put`, see clause A.10.8 Input-Output for Integer Types, A.10.9 Input-Output for Real Types, and A.11 Wide Text Input-Output, of ISO/IEC 8652:1995. *base*, n, w, i, a, and e are expressions for non-negative integers. e is greater than 0. *base* is greater than 1.

Ada provides non-negative numerals for all its integer and floating point types. The default base is 10, but all bases from 2 to 16 can be used. There is no differentiation between the numerals for different floating point types, nor between numerals for different integer types, but integer numerals (without a point) cannot be used for floating point types, and 'real' numerals (with a

point) cannot be used for integer types. Integer numerals can have an exponent part though. The details are not repeated in this example binding, see ISO/IEC 8652:1995, clause 2.4 Numeric Literals, clause 3.5.4 Integer Types, and clause 3.5.6 Real Types.

The Ada standard does not specify any numerals for infinities and NaNs. The following syntax is suggested:

+∞	*FLT*'`Infinity`	†
qNaN	*FLT*'`NaN`	†
sNaN	*FLT*'`NaNSignalling`	†

as well as string formats for reading and writing these values as character strings.

Ada has a notion of 'exception' that implies a non-returnable, but catchable, change of control flow. Ada uses its exception mechanism as its default means of notification. **underflow** does not cause any notification in Ada, and the continuation value to the **underflow** is used directly, since an Ada exception is inappropriate for an **underflow** notification. On **underflow** the continuation value (specified in LIA) is used directly without recording the **underflow** itself. Ada uses the exception `Constraint_Error` for **infinitary** and **overflow** notifications, and the exceptions `Numerics.Argument_Error`, `IO_Exceptions.Data_Error`, and `IO_Exceptions.End_Error` for **invalid** notifications.

Since Ada exceptions are non-returnable changes of control flow, no continuation value is provided for these notifications.

An implementation that wishes to follow LIA must provide recording of indicators as an alternative means of handling numeric notifications. 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*.

overflow	1	†
underflow	2	†
invalid	4	†
infinitary	8	†
absolute_precision_underflow	16	† (LIA-2, -3)
inexact	32	† (IEC 60559)
clear_indicators(S)	`Clear_Indicators(`S`)`	†
set_indicators(S)	`Set_Indicators(`S`)`	†
test_indicators(S)	`Test_Indicators(`S`)`	†
current_indicators()	`Current_Indicators()`	†

where S is an expression compatible with the datatype `Natural`.

E.2 C

The programming language C is defined by ISO/IEC 9899:1999, *Information technology – 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. 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` (1 = **true** and 0 = **false**).

C defines numerous integer datatypes. They may be aliases of each other in an implementation defined way. The description here is not complete. See the C99 standard. Some of the integer datatypes have a predetermined bit width, and the signed ones use 2's complement for representation of negative values: `int`n`_t` and `uint`n`_t`, where n is the bit width expressed as a decimal numeral. Some bit widths are required. There are also minimum width, fastest minimum width, and special purpose integer datatypes (like `size_t`). Also provided are the more well-known integer datatypes `char`, `short int`, `int`, `long int`, `long long int` (new in C99), `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int` (new in C99). Finally there are the integer datatypes `intmax_t` and `uintmax_t` (both new in C99) that are the largest provided signed and unsigned integer datatypes. `intmax_t` and `uintmax_t` may even be unbounded with a negative integer infinity as `INTMAX_MIN` and a positive integer infinity as `INTMAX_MAX` and `UINTMAX_MAX`. *INT* is used below to designate one of the integer datatypes.

NOTES

1 The conformity of `short` and `char` (signed or unsigned) is not relevant since values of these types are promoted to `int` (signed or unsigned) before computations are done.

2 `unsigned int`, `unsigned long int`, and `unsigned long long int` can 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.

The LIA-1 parameters for an integer datatype can be accessed by the following syntax:

$maxint_I$	T`_MAX`	$\star$
$minint_I$	T`_MIN`	$\star$(for signed ints)
$modulo_I$	T`_MODULO`	†(for signed ints)

where T is `INT` for `int`, `LONG` for `long int`, `LLONG` for `long long int`, `UINT` for `unsigned int`, `ULONG` for `unsigned long int`, and `ULLONG` for `unsigned long long int`.

The parameter $bounded_I$ is always **true**, and is not provided. The parameter $minint_I$ is always 0 for the unsigned types, and is not 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	$\star$
$add_I(x, y)$	x `+` y	($\star$) (if $modulo_I$ = **false**)
$add'_I(x, y)$	x `+` y	$\star$ (if $modulo_I$ = **true**)
$neg_I(x)$	`-` x	$\star$

$sub_I(x, y)$	x `-` y	($\star$) (if $modulo_I =$ **false**)
$sub'_I(x, y)$	x `-` y	$\star$ (if $modulo_I =$ **true**)
$abs_I(x)$	t`abs(`x`)`	$\star$ (for signed ints)
$signum_I(x)$	t`sgn(`x`)`	† (for signed ints)
$mul_I(x, y)$	x `*` y	($\star$) (if $modulo_I =$ **false**)
$mul'_I(x, y)$	x `*` y	$\star$ (if $modulo_I =$ **true**)
$quot_I(x, y)$	t`quot(`x`, `y`)`	†
$mod_I(x, y)$	t`mod(`x`, `y`)`	†
$truncdiv_I(x, y)$	x `/` y (dangerous syntax)	$\star$ (bad sem., not LIA-1!)
$truncrem_I(x, y)$	x `%` y	$\star$ (bad sem., not LIA-1!)

where x and y are expressions of type `int`, `long int`, or `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`.

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`. *FLT* is used below to designate one of the floating point datatypes.

The LIA-1 parameters for a floating point datatype can be accessed by the following syntax:

r_F	`FLT_RADIX`	$\star$
p_F	T`_MANT_DIG`	$\star$
$emax_F$	T`_MAX_EXP`	$\star$
$emin_F$	T`_MIN_EXP`	$\star$
$denorm_F$	T`_DENORM`	†
iec_559_F	T`_IEC_60559`	†

where T is `FLT` for `float`, `DBL` for `double`, and `LDBL` for `long double`. Note that `FLT_RADIX` gives the radix for all of `float`, `double`, and `long double`.

The C language standard presumes that all floating point precisions use the same radix and rounding style, so that only one identifier for each is provided in the language.

The LIA-1 derived constants for the floating point datatype can be accessed by the following syntax:

$fmax_F$	T`_MAX`	$\star$
$fminN_F$	T`_MIN`	$\star$
$fmin_F$	T`_TRUE_MIN`	†
$epsilon_F$	T`_EPSILON`	$\star$
rnd_error_F	T`_RND_ERR`	† (partial conf.)
rnd_style_F	`FLT_ROUNDS`	$\star$ (partial conf.)

where T is `FLT` for `float`, `DBL` for `double`, and `LDBL` for `long double`. Note that `FLT_ROUNDS` gives the rounding style for all of `float`, `double`, and `long double`.

The C standard specifies that the values of the parameter `FLT_ROUNDS` are from `int` with the following meaning in terms of the LIA-1 rounding styles.

nearesttiestoeven	`FLT_ROUNDS` $= 2$	†
nearest	`FLT_ROUNDS` $= 1$	
truncate	`FLT_ROUNDS` $= 0$	
other	`FLT_ROUNDS` $\neq 0$ or 1 or 2	

NOTE 3 – The definition of `FLT_ROUNDS` has been extended to cover the rounding style used in all LIA-1 operations, not just addition.

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`)`	†
$add_F(x, y)$	x `+` y	⋆
$neg_F(x)$	`-` x	⋆
$sub_F(x, y)$	x `-` y	⋆
$abs_F(x)$	`fabs`t`(`x`)`	⋆
$signum_F(x)$	`signbit(`x`)`	⋆
$mul_F(x, y)$	x `*` y	⋆
$sqrt_F(x)$	`sqrt`t`(`x`)` or `sqrt(`x`)`	⋆
$residue_F(x, y)$	`remainder`t`(`x`,` y`)` or `remainder(`x`,` y`)`	⋆
$div_F(x, y)$	x `/` y	⋆
$exponent_{F,I}(x)$	`(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`)`	⋆
$succ_F(x)$	`succ`t`(`x`)`	†
$pred_F(x)$	`pred`t`(`x`)`	†
$ulp_F(x)$	`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`, and `l` for `long double`.

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 C99 standard for a full description. In the format strings `%` is used to indicate the start of a format pattern. After the `%`, optionally a string field width (w below) may be given as a positive decimal integer numeral.

For the floating and fixed point format patterns, there may then optionally be a '`.`' followed by a positive integer numeral (d below) indicating the number of fractional digits in the string. The C operations below use HYPHEN-MINUS rather than MINUS (which would have been typographically better), and only digits that are in ASCII, independently of so-called locale. For generating or parsing other kinds of digits, say Arabic digits or Thai digits, another API must be used, that is not standardised in C. For the floating and fixed point formats, $+\infty$ may be represented as either `inf` or `infinity`, $-\infty$ may be represented as either `-inf` or `-infinity`, and a **NaN** may be represented as `NaN`; all independently of so-called locale. For language dependent representations of these values another API must be used, that is not standardised in C.

For the integer formats then follows an internal type indicator, of which some are new to C99. Not all C99 integer types have internal type indicators. However, for t below, `hh` indicates `char`, `h` indicates `short int`, the empty string indicates `int`, `l` (the letter l) indicates `long int`, `ll` (the letters ll) indicates `long long int`, and `j` indicates `intmax_t` or `uintmax_t`. (For system purposes there are also special type names like `size_t`, and `z` indicates `size_t` and `t` indicates `ptrdiff_t` as type format letters.) Finally, there is a radix (for the string side) and signedness (both sides) format letter (r below): `d` for signed decimal; `o`, `u`, `x`, `X` for octal, decimal, hexadecimal with small letters, and hexadecimal with capital letters, all unsigned. E.g., `%jd` indicates decimal numeral string for `intmax_t`, `%2hhx` indicates hexadecimal numeral string for `unsigned char`, with a two character field width, and `%lu` indicates decimal numeral string for `unsigned long int`.

For the floating point formats instead follows another internal type indicator. Not all C99 floating point types have standard internal type indicators for the format strings. However, for u below the empty string indicates `double` and `L` indicates `long double`. Finally, there is a radix (for the string side) format letter: `e` or `E` for decimal, `a` or `A` for hexadecimal. E.g., `%15.8LA` indicates hexadecimal floating point numeral string for `long double`, with capital letters for the letter components, a field width of 15 characters, and 8 hexadecimal fractional digits.

For the fixed point formats also follows the internal type indicator as for the floating point formats. But for the final part of the pattern, there is another radix (for the string side) format letter (p below), only two are standardised, both for the decimal radix: `f` or `F`. E.g., `%Lf` indicates decimal fixed point numeral string for `long double`, with a small letter for the letter component. (There is also a combined floating/fixed point string format: `g`.)

$convert_{I\to I'}(x)$	`(`*INT2*`)`x	$\star$
$convert_{I''\to I}(s)$	`sscanf(`s`, "%`wtr`", &`i`)`	$\star$

$convert_{I''\to I}(f)$	`fscanf(`f`, "%`wtr`", &`i`)`	$\star$
$convert_{I\to I''}(x)$	`sprintf(`s`, "%`wtr`", `x`)`	$\star$
$convert_{I\to I''}(x)$	`fprintf(`h`, "%`wtr`", `x`)`	$\star$
$floor_{F\to I}(y)$	`(`*INT*`)floor`t`(`y`)`	$\star$
$floor_{F\to I}(y)$	`(`*INT*`)nearbyint`t`(`y`)` (when in round towards $-\infty$ mode)	$\star$(C99)
$rounding_{F\to I}(y)$	`(`*INT*`)nearbyint`t`(`y`)` (when in round to nearest mode)	$\star$(C99)
$ceiling_{F\to I}(y)$	`(`*INT*`)nearbyint`t`(`y`)` (when in round towards $+\infty$ mode)	$\star$(C99)
$ceiling_{F\to I}(y)$	`(`*INT*`)ceil`t`(`y`)`	$\star$
$convert_{I\to F}(x)$	`(`*FLT*`)`x	$\star$
$convert_{F\to F'}(y)$	`(`*FLT2*`)`y	$\star$
$convert_{F''\to F}(s)$	`sscanf(`s`, "%`$w.duv$`", &`r`)`	$\star$
$convert_{F''\to F}(f)$	`fscanf(`f`, "%`$w.duv$`", &`r`)`	$\star$
$convert_{F\to F''}(y)$	`sprintf(`s`, "%`$w.duv$`", `y`)`	$\star$
$convert_{F\to F''}(y)$	`fprintf(`h`, "%`$w.duv$`", `y`)`	$\star$
$convert_{D'\to F}(s)$	`sscanf(`s`, "%`wup`", &`g`)`	$\star$
$convert_{D'\to F}(f)$	`fscanf(`f`, "%`wup`", &`g`)`	$\star$
$convert_{F\to D'}(y)$	`sprintf(`s`, "%`$w.dup$`", `y`)`	$\star$
$convert_{F\to D'}(y)$	`fprintf(`h`, "%`$w.dup$`", `y`)`	$\star$

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 float) can be used too. Numerals for different integer types are distinguished by suffixes. Numerals for different floating point types are distinguished by suffix: `f` for `float`, no suffix for `double`, `l` for `long double`. Numerals for floating point types must have a '.' or an exponent in them. The details are not repeated in this example binding, see ISO/IEC 9899:1999, clause 6.4.4.1 Integer constants, and clause 6.4.4.2 Floating constants.

C specifies numerals (as macros) for infinities and NaNs for `float`:

$+\infty$	`INFINITY`	$\star$
qNaN	`NAN`	$\star$
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 of indicators as one method of notification. (See 6.2.1.) The datatype *Ind* is identified with the datatype `int`. The values representing individual indicators should be distinct non-negative powers of two and can be accessed by the following syntax:

overflow	`FE_OVERFLOW`	(⋆)
underflow	`FE_UNDERFLOW`	(⋆)
invalid	`FE_INVALID`	(⋆)
infinitary	`FE_DIVBYZERO`	(⋆)
absolute_precision_underflow	`FE_ARGUMENT_TOO_IMPRECISE`	†, LIA-2, -3
inexact	`FE_INEXACT`	(⋆), IEC 60559

The empty set can be denoted by 0. Other indicator subsets can be named by combining individual indicators using bit-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 are listed below, along with the syntax used to invoke them:

clear_indicators	`feclearexcept(`i`)`	⋆
set_indicators	`feraiseexcept(`i`)`	⋆
test_indicators	`fetestexcept(`i`)`	⋆
current_indicators	`fetestexcept(FE_ALL_EXCEPT)`	⋆

where i is an expression of type `int` representing an indicator subset.

E.3 C++

The programming language C++ is defined by ISO/IEC 14882:1998, *Programming languages – C++* [18].

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

Every implementation of C++ has integral datatypes `int`, `long int`, `unsigned int`, and `unsigned long int`. *INT* is used below to designate one of the integer datatypes.

NOTES

1 The conformity of `short` and `char` (signed or unsigned) is not relevant since values of these types are promoted to `int` (signed or unsigned) before computations are done.

2 `unsigned int`, `unsigned long int`, and `unsigned long long int` can 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.

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.)

The parameter $minint_I$ is always 0 for the unsigned types. The parameter $modulo_I$ is always **true** for the unsigned 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	$\star$
$add_I(x, y)$	x `+` y	($\star$) (if $modulo_I =$ **false**)
$add'_I(x, y)$	x `+` y	$\star$(if $modulo_I =$ **true**)
$neg_I(x)$	`-` x	$\star$
$sub_I(x, y)$	x `-` y	($\star$) (if $modulo_I =$ **false**)
$sub'_I(x, y)$	x `-` y	$\star$(if $modulo_I =$ **true**)
$abs_I(x)$	`abs(`x`)`	$\star$
$signum_I(x)$	`sgn(`x`)`	†
$mul_I(x, y)$	x `*` y	($\star$) (if $modulo_I =$ **false**)
$mul'_I(x, y)$	x `*` y	$\star$(if $modulo_I =$ **true**)
$quot_I(x, y)$	`quot(`x`,` y`)`	†
$mod_I(x, y)$	`mod(`x`,` y`)`	†
$loosediv_I(x, y)$	x `/` y (dangerous syntax)	$\star$ (bad sem., not LIA-1!)
$loosерem_I(x, y)$	x `%` y	$\star$ (bad sem., not LIA-1!)

where x and y are expressions of type `int` or `long int` as appropriate.

C++ has three floating point datatypes: `float`, `double`, and `long double`. *FLT* is used below to designate one of the floating point datatypes.

The LIA-1 parameters for a floating point datatype can be accessed by the following syntax:

r_F	`numeric_limits<`FLT`>::radix`	$\star$
p_F	`numeric_limits<`FLT`>::digits`	$\star$
$emax_F$	`numeric_limits<`FLT`>::max_exponent`	$\star$
$emin_F$	`numeric_limits<`FLT`>::min_exponent`	$\star$
$denorm_F$	`numeric_limits<`FLT`>::has_denorm`	$\star$
$hasinf_F$	`numeric_limits<`FLT`>::has_infinity`	$\star$ (not LIA-1)
$hasqnan_F$	`numeric_limits<`FLT`>::has_quiet_nan`	$\star$ (not LIA-1)
$hassnan_F$	`numeric_limits<`FLT`>::has_signalling_nan`	$\star$ (not LIA-1)
iec_559_F	`numeric_limits<`FLT`>::is_iec559`	$\star$
$traps_F$	`numeric_limits<`FLT`>::traps`	$\star$ (not LIA-1)
$tinyness\_before_F$	`numeric_limits<`FLT`>::tinyness_before`	$\star$ (LIA-1 extra)

The C++ language standard presumes that all floating point precisions use the same radix and rounding style, so that only one identifier for each is provided in the language.

The LIA-1 derived constants for the floating point datatype can be accessed by the following syntax:

$fmax_F$	`numeric_limits<`FLT`>::max()`	$\star$
$fminN_F$	`numeric_limits<`FLT`>::min()`	$\star$
$fmin_F$	`numeric_limits<`FLT`>::denorm_min`	$\star$
$epsilon_F$	`numeric_limits<`FLT`>::epsilon()`	$\star$
rnd_error_F	`numeric_limits<`FLT`>::round_error()`	$\star$ (partial conf.)
rnd_style_F	`numeric_limits<`FLT`>::round_style`	$\star$ (partial conf.)
$approx\_p\_10_F$	`numeric_limits<`FLT`>::digits10`	$\star$ (not LIA-1)
$approx\_emax\_10_F$	`numeric_limits<`FLT`>::max_exponent10`	$\star$ (not LIA-1)
$approx\_emin\_10_F$	`numeric_limits<`FLT`>::min_exponent10`	$\star$ (not LIA-1)

The C++ standard specifies that the values of the parameter `round_style` are from `float_round_style`.

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	$\star$
$neq_F(x, y)$	x `!=` y	$\star$
$lss_F(x, y)$	x `<` y	$\star$
$leq_F(x, y)$	x `<=` y	$\star$
$gtr_F(x, y)$	x `>` y	$\star$
$geq_F(x, y)$	x `>=` y	$\star$
$isnegzero_F(x)$	`isNegZero(`x`)`	†
$istiny_F(x)$	`isTiny(`x`)`	†
$istiny_F(x)$	`-numeric_limits<`FLT`>::min()` $<$ x `&&` x `<` `numeric_limits<`FLT`>::min())`	$\star$
$isnan_F(x)$	`isNaN(`x`)`	†
$isnan_F(x)$	x `!=` x	$\star$
$issignan_F(x)$	`isSigNaN(`x`)`	†
$add_F(x, y)$	x `+` y	$\star$
$neg_F(x)$	`-` x	$\star$
$sub_F(x, y)$	x `-` y	$\star$
$abs_F(x)$	`abs(`x`)`	$\star$
$signum_F(x)$	`sgn(`x`)`	†
$mul_F(x, y)$	x `*` y	$\star$
$sqrt_F(x)$	`sqrt(`x`)`	$\star$
$residue_F(x, y)$	`remainder(`x`,` y`)`	($\star$)
$div_F(x, y)$	x `/` y	$\star$
$exponent_{F,I}(x)$	`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 all of the LIA-1 operations in all floating point precisions supported.

Arithmetic value conversions in C++ can be explicit or implicit. The rules for when implicit conversions are applied are not repeated here. C++ also deals with stream input/output in other ways, see clause 22.2.2 of ISO/IEC 14882:1998, 'Locale and facets'. The explicit arithmetic value conversions are usually expressed as 'casts', except when converting to/from string formats.

When converting to/from string formats, format strings are used. The format string is used as a pattern for the string format generated or parsed. The description of format strings here is not complete. Please see the C++ standard for a full description.

In the format strings `%` is used to indicate the start of a format pattern. After the `%`, optionally a string field width (w below) may be given as a positive decimal integer numeral. For the floating and fixed point format patterns, there may then optionally be a '`.`' followed by a positive integer numeral (d below) indicating the number of fractional digits in the string. The C++ operations below use HYPHEN-MINUS rather than MINUS (which would have been typographically better), and only digits that are in ASCII, independently of so-called locale. For generating or parsing other kinds of digits, say Arabic digits or Thai digits, another API must be used, that is not standardised in C++. For the floating and fixed point formats, $+\infty$ may be represented as either `inf` or `infinity`, $-\infty$ may be represented as either `-inf` or `-infinity`, and a **NaN** may be represented as `NaN`; all independently of so-called locale. For language dependent representations of these values another API must be used, that is not standardised in C.

For the integer formats then follows an internal type indicator. For t below, the empty string indicates `int`, `l` (the letter l) indicates `long int`. Finally, there is a radix (for the string side) and signedness (both sides) format letter (r below): `d` for signed decimal; `o`, `u`, `x`, `X` for octal, decimal, hexadecimal with small letters, and hexadecimal with capital letters, all unsigned. E.g., `%d` indicates decimal numeral string for `int` and `%lu` indicates decimal numeral string for `unsigned long int`.

For the floating point formats instead follows another internal type indicator. For u below the empty string indicates `double` and `L` indicates `long double`. Finally, there is a radix (for the string side) format letter: `e` or `E` for decimal. E.g., `%15.8LE` indicates hexadecimal floating point numeral string for `long double`, with a capital letter for the letter component, a field width of 15 characters, and 8 hexadecimal fractional digits.

For the fixed point formats also follows the internal type indicator as for the floating point formats. But for the final part of the pattern, there is another radix (for the string side) format letter (p below), only two are standardised, both for the decimal radix: `f` or `F`. E.g., `%Lf` indicates decimal fixed point numeral string for `long double`, with a small letter for the letter component. (There is also a combined floating/fixed point string format: `g`.)

$convert_{I\to I'}(x)$	`static_cast<`$INT2$`>(`x`)`	⋆
$convert_{I''\to I}(s)$	`sscanf(`s`, "%`wtr`", &`i`)`	⋆

$convert_{I''\to I}(f)$	`fscanf(`f`, "%`wtr`", &`i`)`	⋆
$convert_{I\to I''}(x)$	`sprintf(`s`, "%`wtr`", `x`)`	⋆
$convert_{I\to I''}(x)$	`fprintf(`h`, "%`wtr`", `x`)`	⋆
$floor_{F\to I}(y)$	`static_cast<`*INT*`>(floor(`y`))`	⋆
$rounding_{F\to I}(y)$	`static_cast<`*INT*`>(round(`y`))`	†
$ceiling_{F\to I}(y)$	`static_cast<`*INT*`>(ceil(`y`))`	⋆
$convert_{I\to F}(x)$	`static_cast<`*FLT*`>(`x`)`	⋆
$convert_{F\to F'}(y)$	`(`*FLT2*`)`y	⋆
$convert_{F''\to F}(s)$	`sscanf(`s`, "%`$w.duv$`", &`r`)`	⋆
$convert_{F''\to F}(f)$	`fscanf(`f`, "%`$w.duv$`", &`r`)`	⋆
$convert_{F\to F''}(y)$	`sprintf(`s`, "%`$w.duv$`", `y`)`	⋆
$convert_{F\to F''}(y)$	`fprintf(`h`, "%`$w.duv$`", `y`)`	⋆
$convert_{D'\to F}(s)$	`sscanf(`s`, "%`wup`", &`g`)`	⋆
$convert_{D'\to F}(f)$	`fscanf(`f`, "%`wup`", &`g`)`	⋆
$convert_{F\to D'}(y)$	`sprintf(`s`, "%`$w.dup$`", `y`)`	⋆
$convert_{F\to D'}(y)$	`fprintf(`h`, "%`$w.dup$`", `y`)`	⋆

where s is an expression of type `char*`, f is an expression of type `FILE*`, i is an lvalue expression of type `int`, g is an lvalue expression of type `double`, x is an expression of type *INT*, y is an expression of type *FLT*, *INT2* is the integer datatype that corresponds to I', and *FLT2* is the floating point datatype that corresponds to F'.

C++ provides non-negative numerals for all its integer and floating point types in base 10. Numerals for different integer types are distinguished by suffixes. Numerals for different floating point types are distinguished by suffix: `f` for `float`, no suffix for `double`, `l` for `long double`. Numerals for floating point types must have a '.' or an exponent in them. The details are not repeated in this example binding, see ISO/IEC 14882:1998, clause 2.9.1 Integer literals, and clause 2.9.4 Floating literals.

C++ specifies numerals for infinities and NaNs:

+∞	`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 of indicators as one method of notification. (See 6.2.1.) The datatype *Ind* is identified with the datatype `int`. The values representing individual indicators should be distinct non-negative powers of two and can be accessed by the following syntax:

overflow	`FE_OVERFLOW`	†
underflow	`FE_UNDERFLOW`	†
invalid	`FE_INVALID`	†
infinitary	`FE_DIVBYZERO`	†
absolute_precision_underflow	`FE_ARGUMENT_TOO_IMPRECISE`	†, LIA-2, -3

inexact	`FE_INEXACT`	†, IEC 60559

The empty set can be denoted by 0. Other indicator subsets can be named by combining individual indicators using bit-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 are listed below, along with the syntax used to invoke them:

clear_indicators	`feclearexcept(`i`)`	†
set_indicators	`feraiseexcept(`i`)`	†
test_indicators	`fetestexcept(`i`)`	†
current_indicators	`fetestexcept(FE_ALL_EXCEPT)`	†

where i is an expression of type `int` representing an indicator subset.

E.4 Fortran

The programming language Fortran is defined by ISO/IEC 1539-1:1997, *Information technology – Programming languages – Fortran – Part 1: Base language* [22]. It is complemented with ISO/IEC TR 15580:1998, *Information technology – Programming languages – Fortran – Floating-point exception handling* [23].

An implementation should follow all the requirements of LIA-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 one integer datatype, denoted as `INTEGER`. An implementation is permitted to offer additional `INTEGER` types 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`)`	†

where x is an expression of type `INTEGER`, and the result returned is appropriate for the `KIND` 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	$\star$
$add_I(x, y)$	x `+` y	$\star$
$neg_I(x)$	`-` x	$\star$
$sub_I(x, y)$	x `-` y	$\star$
$abs_I(x)$	`ABS(`x`)`	$\star$
$signum_I(x)$	`SIGN(1,` x`)`	$\star$
$mul_I(signum_I(x), abs_I(y))$	`SIGN(`y`,` x`)`	$\star$
$mul_I(x, y)$	x `*` y	$\star$
$quot_I(x, y)$	`QUOTIENT(`x`,` y`)`	†
$mod_I(x, y)$	`MODULO(`x`,` y`)`	$\star$
$truncdiv_I(x, y)$	x `/` y (dangerous syntax)	$\star$ (bad sem., not LIA-1!)
$truncrem_I(x, y)$	`MOD(`x`,` y`)` (dangerous syntax)	$\star$ (bad sem., not LIA-1!)

where x and y are expressions involving integers of the same `KIND`.

Every implementation of Fortran has two floating point datatypes, denoted as `REAL` (single precision) and `DOUBLE PRECISION`. An implementation is permitted to offer additional `REAL` types with different precision or range, parameterized with the ***kind*** parameter.

The LIA-1 parameters for a `REAL` datatype can be accessed by the following syntax:

r_F	`RADIX(`x`)`	$\star$
p_F	`DIGITS(`x`)`	$\star$
$emax_F$	`MAXEXPONENT(`x`)`	$\star$
$emin_F$	`MINEXPONENT(`x`)`	$\star$
$denorm_F$	`IEEE_SUPPORT_DENORMAL(`x`)`	($\star$)
$hasinf_F$	`IEEE_SUPPORT_INF(`x`)`	($\star$) (not LIA-1)
$hasqnan_F$	`IEEE_SUPPORT_NAN(`x`)`	($\star$) (not LIA-1)
iec_559_F	`IEEE_SUPPORT_STANDARD(`x`)`	($\star$)

where x is an expression of the appropriate `KIND REAL` datatype.

The LIA-1 derived constants for `REAL` datatypes can be accessed by the following syntax:

$fmax_F$	`HUGE(`x`)`	$\star$
$fminN_F$	`TINY(`x`)`	$\star$
$fmin_F$	`TINIEST(`x`)`	†
$epsilon_F$	`EPSILON(`x`)`	$\star$
rnd_error_F	`RND_ERROR(`x`)`	† (partial conf.)
rnd_style_F	`RND_STYLE`	† (partial conf.)

where x is an expression of type `KIND REAL`.

The allowed values of the parameter `RND_STYLE` are from the datatype `INTEGER` and can be accessed by the following syntax:

nearesttiestoeven	`RND_NEAREST...`	†
nearest	`RND_NEAREST`	†
truncate	`RND_TRUNCATE`	†
other	`RND_OTHER`	†

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	$\star$
$neq_F(x,y)$	x `.NE.` y or x `/=` y	$\star$
$lss_F(x,y)$	x `.LT.` y or x `<` y	$\star$
$leq_F(x,y)$	x `.LE.` y or x `<=` y	$\star$
$gtr_F(x,y)$	x `.GT.` y or x `>` y	$\star$
$geq_F(x,y)$	x `.GE.` y or x `>=` y	$\star$
$unordered_F(x,y)$	`IEEE_UNORDERED(`x`,` y`)`	($\star$), IEC 60559
$isnegzero_F(x)$	x `== 0.0 .AND. IEEE_IS_NEGATIVE(`x`)`	($\star$)
$istiny_F(x)$	`-TINY <` x `.AND.` x `< TINY`	$\star$
$isnan_F(x)$	`IEEE_IS_NAN(`x`)`	($\star$)
$issignan_F(x)$	`isSigNaN(`x`)`	†
$add_F(x,y)$	x `+` y	$\star$
$neg_F(x)$	`-` x	$\star$
$sub_F(x,y)$	x `-` y	$\star$
$abs_F(x)$	`ABS(`x`)`	$\star$
$signum_I(x)$	`SIGN(1.0,` x`)`	$\star$
$mul_I(signum_I(x), abs_I(y))$	`SIGN(`y`,` x`)`	$\star$
$mul_F(x,y)$	x `*` y	$\star$
$sqrt_F(x)$	`SQRT(`x`)`	$\star$
$residue_F(x,y)$	`RESIDUE(`x`,` y`)`	†
$div_F(x,y)$	x `/` y	$\star$
$exponent_{F,I}(x)$	`EXPONENT(`x`)`	$\star$ (dev.: 0 if $x = 0$)
$exponent_{F,I}(x)$	`FLOOR(IEEE_LOGB(`x`)) + 1`	($\star$)
$fraction_F(x)$	`FRACTION(`x`)`	$\star$
$scale_{F,I}(x,n)$	`SCALE(`x`,` n`)`	$\star$
$scale_{F,I}(x,n)$	`IEEE_SCALB(`x`,` n`)`	($\star$)
$succ_F(x)$	`NEAREST(`x`, 1.0)`	$\star$
$succ_F(x)$	`IEEE_NEXT_AFTER(`x`, HUGE(`x`))`	($\star$) (dev. at $fmax_F$)
$pred_F(x)$	`NEAREST(`x`, -1.0)`	$\star$
$succ_F(x)$	`IEEE_NEXT_AFTER(`x`, -HUGE(`x`))`	($\star$) (dev. at $-fmax_F$)
$ulp_F(x)$	`SPACING(`x`)`	$\star$
$intpart_F(x)$	`AINT(`x`)`	$\star$
$fractpart_F(x)$	x `- AINT(`x`)`	$\star$
$trunc_{F,I}(x,n)$	`TRUNC(`x,n`)`	†
$round_{F,I}(x,n)$	`ROUND(`x,n`)`	†

where x and y are reals of the same ***kind***, and n is of integer type.

An implementation that wishes to conform to LIA-1 for all its integer and floating point datatypes must provideLIA-1 operations and parameters for any additional `INTEGER` or `REAL` types provided.

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

$convert_{I \to I'}(x)$		`INT(`x`,` *kindi*`2)`	$\star$
	lbl_a	`FORMAT (B`n`)`	$\star$(binary)

$convert_{I'' \to I}(f)$		READ (UNIT=#f,FMT=lbl_a) r	⋆
$convert_{I \to I''}(x)$		WRITE (UNIT=#h, FMT=lbl_a) x	⋆
	lbl_b	FORMAT (On)	⋆(octal)
$convert_{I'' \to I}(f)$		READ (UNIT=#f,FMT=lbl_b) r	⋆
$convert_{I \to I''}(x)$		WRITE (UNIT=#h, FMT=lbl_b) x	⋆
	lbl_c	FORMAT (In)	⋆(decimal)
$convert_{I'' \to I}(f)$		READ (UNIT=#f,FMT=lbl_c) r	⋆
$convert_{I \to I''}(x)$		WRITE (UNIT=#h, FMT=lbl_c) x	⋆
	lbl_d	FORMAT (Zn)	⋆(hexadecimal)
$convert_{I'' \to I}(f)$		READ (UNIT=#f,FMT=lbl_d) r	⋆
$convert_{I \to I''}(x)$		WRITE (UNIT=#h, FMT=lbl_d) x	⋆
$floor_{F \to I}(y)$		FLOOR(y, *kindi*?)	⋆
$rounding_{F \to I}(y)$		ROUND(y, *kindi*?)	†
$ceiling_{F \to I}(y)$		CEILING(y, *kindi*?)	⋆
$convert_{I \to F}(x)$		REAL(x, *kind*) or sometimes DBLE(x)	⋆
$convert_{F \to F'}(y)$		REAL(y, *kind2*) or sometimes DBLE(y)	⋆
	lbl_e	FORMAT (Fw.d)	⋆
	lbl_f	FORMAT (Dw.d)	⋆
	lbl_g	FORMAT (Ew.d)	⋆
	lbl_h	FORMAT (Ew.dEe)	⋆
	lbl_i	FORMAT (ENw.d)	⋆
	lbl_j	FORMAT (ENw.dEe)	⋆
	lbl_k	FORMAT (ESw.d)	⋆
	lbl_l	FORMAT (ESw.dEe)	⋆
$convert_{F'' \to F}(f)$		READ (UNIT=#f, FMT=lbl_x) t	⋆
$convert_{F \to F''}(y)$		WRITE (UNIT=#h, FMT=lbl_x) y	⋆
$convert_{D' \to F}(f)$		READ (UNIT=#f, FMT=lbl_x) t	⋆

where x is an expression of type INTEGER(*kindi*), y is an expression of type REAL(*kind*), f is an input file with unit number #f, and h is an output file with unit number #h. w, d, and e are literal digit (0-9) sequences, giving total, decimals, and exponent widths. lbl_x is one of lbl_e to lbl_l; all of the $lbl_$s are labels for formats.

Fortran provides base 10 non-negative numerals for all of its integer and floating point types. Numerals for floating point types must have a '.' in them. The details are not repeated in this example binding, see ISO/IEC 1539-1:1997, clause 4.3.1.1 Integer type, and clause 4.3.1.2 Real type.

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

+∞	INFINITY	†
qNaN	NAN	†
sNaN	NANSIGNALLING	†

as well as string formats for reading and writing these values as character strings.

Fortran implementations can provide recording of indicators for floating point arithmetic notifications, the LIA preferred method. See ISO/IEC TR 15580:1998, *Information technology – Programming languages – Fortran – Floating-point exception handling* [23]. **absolute_precision_underflow** notifications are however ignored.

An implementation that wishes to conform to LIA-1 must provide recording of indicators 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 and can be accessed by the following syntax:

overflow	`IEEE_OVERFLOW`	($\star$)
underflow	`IEEE_UNDERFLOW`	($\star$)
invalid	`IEEE_INVALID`	($\star$)
infinitary	`IEEE_DIVIDE_BY_ZERO`	($\star$)
inexact	`IEEE_INEXACT`	($\star$), IEC 60559
absolute_precision_underflow	`ABS_PRECISION_UNDERFLOW`	†, LIA-2, -3

... The indicator interrogation and manipulation operations are listed below, along with the syntax used to invoke them:

set_indicators(i)	`IEEE_SET_FLAG(`i`, .TRUE.)`	($\star$), only one flag
clear_indicators(i)	`IEEE_SET_FLAG(`i`, .FALSE.)`	($\star$), only one flag
test_indicators(i)	`CALL IEEE_GET_STATUS(STATUS_VALUE)`	
	.,.,	($\star$)
current_indicators()	`CALL IEEE_GET_STATUS(STATUS_VALUE)`	($\star$)

where i is an expression of type `IEEE_STATUS_TYPE` representing an indicator.

E.5 Common Lisp

The programming language Common Lisp is defined by ANSI X3.226-1994, *Information Technology – Programming Language – Common Lisp* [42].

An implementation should follow all the requirements of LIA-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**, and the parameters $bounded_I$, $modulo_I$, $maxint_I$, and $minint_I$ 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 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`)`	⋆
$add_I(x, y)$	`(+` x y`)`	⋆
$neg_I(x)$	`(-` x`)`	⋆
$sub_I(x, y)$	`(-` x y`)`	⋆
$abs_I(x)$	`(abs` x`)`	⋆
$signum_I(x)$	`(sign` x`)`	†
$mul_I(x, y)$	`(*` x y`)`	⋆

(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`))`	⋆
$truncdiv_I(x, y)$	`trunc` or `(truncate` x y`)`	⋆ (bad sem., not LIA-1!)
$truncrem_I(x, y)$	`rest` or `(rem` x y`)`	⋆ (bad sem., not LIA-1!)

where x and y are expressions of type `integer`.

Common Lisp has four floating point types: `short-float`, `single-float`, `double-float`, and `long-float`. Not all of these floating point types must be distinct.

The LIA-1 parameters for the floating point types can be accessed by the following constants and inquiry functions.

r_F	`(float-radix` x`)`	⋆
p_F	`(float-digits` x`)`	⋆
$emax_F$	`maxexp-`T	⋆
$emin_F$	`minexp-`T	⋆
$denorm_F$	`denorm-`T	†
iec_559_F	`iec-559-`T	†

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 approriate.

The LIA-1 derived constants for the floating point datatype can be accessed by the following syntax:

$fmax_F$	`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.)

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`	†

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)`	†
$add_F(x, y)$	`(+ x y)`	⋆
$neg_F(x)$	`(- x)`	⋆
$sub_F(x, y)$	`(- x y)`	⋆
$abs_F(x)$	`(abs x)`	⋆
$signum_F(x)$	`(sign x)`	†
$mul_F(x, y)$	`(* x y)`	⋆
$sqrt_F(x, y)$	`(sqrt x)`	⋆
	`(multiple-value-bind (rnd rm) (round x y))`	⋆
$residue_F(x, y)$	`rm`	
$div_F(x, y)$	`(/ x y)`	⋆
$exponent_{F,I}(x)$	`(float-exponent x)`	†
$fraction_F(x)$	`(decode-float x)`	⋆
$scale_{F,I}(x, n)$	`(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.

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

$convert_{I\to I''}(x)$	`(format nil "~`w`B" `x`)`	⋆(binary)
$convert_{I\to I''}(x)$	`(format nil "~`w`O" `x`)`	⋆(octal)
$convert_{I\to I''}(x)$	`(format nil "~`w`D" `x`)`	⋆(decimal)
$convert_{I\to I''}(x)$	`(format nil "~`w`X" `x`)`	⋆(hexadecimal)
$convert_{I\to I''}(x)$	`(format nil "~`r, w`R" `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 *kind*`2)`	⋆
$convert_{F\to F''}(y)$	`(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*. Conversion from string to numeric value is in Common Lisp done via a general read procedure, which reads Common Lisp 'S-expressions'.

Common Lisp provides non-negative numerals for all its integer and floating point datatypes in base 10.

There is no differentiation between the numerals for different floating point datatypes, nor between numerals for different integer datatypes, and integer numerals can be used for floating point values.

Common Lisp does not specify numerals for infinities and NaNs. Suggestion:

+∞	`infinity-`*FLT*	†
qNaN	`nan-`*FLT*	†
sNaN	`signan-`*FLT*	†

as well as string formats for reading and writing these values as character strings.

Common Lisp has a notion of 'exception'.

However, Common Lisp has no notion of compile time type checking, and an operation can return differently typed values for different arguments. When justifiable, Common Lisp arithmetic operations return a rational or a complex floating point value rather than giving a notification, even if the argument(s) to the operation were not complex. For instance, `(sqrt -1)` (quietly) returns a representation of $0 + i$.

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 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	`loss-precision-underflow`	†, LIA-2, -3
inexact	`inexact`	†, IEC 60559

An implementation that wishes to conform to LIA-1 must signal the appropriate condition type whenever an LIA-1 exceptional value would be returned, and must provide a default handler for use in the event that the programmer has not supplied a condition handler.,,,,,,

Annex F
(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. The hardware floating point conforms to the IEEE 754 (IEC 60559) standard.

The sample conformity statement follows.

This implementation of Fortran conforms to the following standards:

ISO/IEC 1539:1991, *Information technology – Programming languages – FORTRAN*

ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic* (also known as IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems*)

ISO/IEC 10967-1:1994, *Language independent arithmetic – Part: 1 Integer and floating point arithmetic* (LIA-1)

It also conforms to the suggested Fortran binding standard in E.4 of LIA-1.

Only implementation dependent information is directly provided here. The information in the suggested language binding standard for Fortran (see E.4) is provided by reference. Together, these two items satisfy the LIA-1 documentation requirement.

F.1 Types

There is one integer type, called `integer`. There are two floating point types, called `real` and `double`.

F.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*
$maxint_I$	`HUGE`(x)	$2^{31} - 1$
$minint_i$	`MININT`(x)	-2^{31}
$hasinf_I$	(none)	**false**

where x is an expression of type `integer`.

F.3 Floating point parameters

The following table gives the parameters for `real` and `double`, 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`	`DOUBLE`
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	`DENORM`(x)	**true**	**true**
iec_559	`IEC_559`(x)	**true**	**true**

where x is an expression of the appropriate floating point type.

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 `double`. The inquiry functions return exact values for the derived constants.

Derived constants			
constants	*inquiry function*	`REAL`	`DOUBLE`
$fmax_F$	`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

where x is an expression of type `real` or `double`.

F.4 Definitions

In this implementation of Fortran, the programmer selects among the rounding functions by using a compiler directive, a comment line of the form

```
!LIA$  directive
```

The relevant directives (and the rounding functions they select) are

```
!LIA$  SELECT ROUND TO NEAREST              (default)
!LIA$  SELECT ROUND TO PLUS INFINITY        (does not conform to LIA-1)
!LIA$  SELECT ROUND TO MINUS INFINITY       (does not conform to LIA-1)
!LIA$  SELECT ROUND TO ZERO                 (does not conform to LIA-1)
```

These compiler directives affect all floating point operations that occur (textually) between the directive itself and the end of the smallest enclosing block or scoping unit, unless superseded by a subsequent directive.

The above directives select the rounding function for both `real` and `double`. In the absence of an applicable directive, the default is round to nearest. The round to nearest style rounds halfway cases such that the last bit of the fraction is 0.

The choice between $nearest_F(x)$ and **underflow**$(nearest_F(x))$ for the subnormal range is made in accordance with clause 7.4 of the IEEE 754 standard. In IEEE terms, this implementation chooses to detect tinyness after rounding, and loss of accuracy as an inexact result.

F.5 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).

F.6 Notification

Notifications are raised under all circumstances specified by the LIA-1. The programmer selects the method of notification by using a compiler directive. The relevant directives are:

```
!LIA$  NOTIFICATION=RECORDING                (default)
!LIA$  NOTIFICATION=TERMINATE
```

If an exception 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 provides the following information:

a) name of the exceptional value (**infinitary**, **overflow**, **underflow**, or **invalid**),

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 G
(informative)

Example programs

This annex presents a few examples of how various LIA-1 features might be used. The program fragments given here are all written in Fortran, C, or Ada, and assume the bindings suggested in E.4, E.2, and E.1, respectively.

G.1 Verifying platform acceptability

A 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, 'This platform 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 ran on IEC 559 platforms would test

```
if (.not. IEC_559(x)) ...
```

G.2 Selecting alternate code

Sometimes the ability to control rounding behavior is very useful. This ability is provided by IEC 60559 platforms. An example (in C) is

```
if (FLT_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 ...
    }
```

G.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 * LIA1.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.

G.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 = expon(dotmax) - expon(dot)
if (loss > digits(dot)/2) then
    print 3, 'Half the precision may be lost.'
end if
```

G.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.

```
#define ALL_INDICATORS  (~0)     /* all ones */
int saved_flags;

saved_flags = save_indicators();
clear_indicators(ALL_INDICATORS);
...  run desired code ...
...  examine indicators and take appropriate action ...
```

```
... clear any indicators that were compensated for ...
set_indicators(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.

G.6 Fast versus accurate

Consider a problem which has two solutions. The first solution is a fast algorithm that works most of the time. However, it occasionally gives incorrect answers because of internal floating point overflows. The second is completely reliable, but is known to be 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 = ...save_indicators()
call ...clear_indicators(ALL_INDICATORS)
result = FAST_SOLUTION(input)
if (..test_indicators(FLT_OVERFLOW)) then
    call ...clear_indicators(ALL_INDICATORS)
    result = RELIABLE_SOLUTION(input)
end if
call ...set_indicators(saved_flags)
```

Demmel and Li discuss a number of similar algorithms in [51].

G.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 [62] 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.

Annex H
(informative)

Bibliography

This annex gives references to publications relevant to LIA-1.

International standards documents

[1] ISO/IEC Directives, Part 3: *Rules for the structure and drafting of International Standards*, 1997.

[2] IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems.* (Also: ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic.*)

[3] ISO/IEC 10967-3, *Information technology – Language independent arithmetic – Part 3: Complex integer and floating point arithmetic and complex elementary numerical functions*, (LIA-3).

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

[5] ISO/IEC 10646-1:2000, *Information technology – Universal multi-octet character set (UCS) – Part 1: Architecture and Basic Multilingual plane*, second edition.

[6] ISO/IEC 10646-2:2001, *Information technology – Universal multi-octet character set (UCS) – Part 2: Supplementary planes.*

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

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

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

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

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

[12] ISO/IEC 13813:1998, *Information technology – Programming languages – Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types).*

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

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

[15] ISO/IEC 13751:2001, *Information technology – Programming languages, their environments and system software interfaces – Programming language extended APL.*

[16] ISO/IEC 10279:1991, *Information technology – Programming languages – Full BASIC.* (Essentially an endorsement of ANSI X3.113-1987 (R1998) [40].)

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

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

[19] ISO 1989:1985, *Programming languages – COBOL.* (Endorsement of ANSI X3.23-1985 (R1991) [41].) Currently (2001) under revision.

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

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

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

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

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

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

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

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

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

[29] ISO 6160:1979, *Programming languages – PL/I.* (Endorsement of ANSI X3.53-1976 (R1998) [43].)

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

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

[32] ISO/IEC 8824-1:1998, *Information technology – Abstract Syntax Notation One (ASN.1) – Part 1: Specification of basic notation.*

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

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

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

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

Nationaland other standards documents

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

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

[39] The Unicode Standard, version 3.0, 2000. Note that version 3.0 the encoded character repertoire is exactly the same as for ISO/IEC 10646-1:2000.

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

[41] ANSI X3.23-1985 (R1991), *Programming languages – COBOL.*

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

[43] ANSI X3.53-1976 (R1998), *Programming languages – PL/I.*

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

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

Books, articles, and other documents

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

[47] M. Abramowitz and I. Stegun (eds), *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, Tenth Printing, 1972, Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

[48] W. S. Brown, A Simple but Realistic Model of Floating-Point Computation, ACM Transactions on Mathematical Software, Vol. 7, 1981, pp.445-480

[49] J. T. Coonen, An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic, Computer, January 1980

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

[51] J. W. Demmel and X. Li, *Faster Numerical Algorithms via Exception Handling*, 11th International Symposium on Computer Arithmetic, Winsor, Ontario, June 29 - July 2, 1993.

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

[53] J. R. Hauser, *Handling Floating-Point Exceptions in Numeric Programs.* ACM Transactions on Programming Languages and Systems, Vol. 18, No. 2, March 1986, Pages 139-174.

[54] J. E. Holm, Floating Point Arithmetic and Program Correctness Proofs, Cornell University TR 80-436, 1980.

[55] C. B. Jones, Systematic Software Development Using VDM, Prentice-Hall, 1986.

[56] W. Kahan and J. Palmer, On a Proposed Floating-Point Standard, SIGNUM Newsletter, October 1979, pp.13-21.

[57] W. Kahan, *Branch Cuts for Complex Elementary Functions, or Much Ado about Nothing's Sign Bit*, Chapter 7 in *The State of the Art in Numerical Analysis* ed. by M. Powell and A. Iserles (1987) Oxford.

[58] 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.

[59] D. E. Knuth, Semi-Numerical Algorithms, Addison-Wesley, 1969, section 4.4

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

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

[62] 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.

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

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

[65] 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.

[66] G. Bohlender, W. Walter, P Kornerup, D. W. Matula, *Semantics for Exact Floating Point Operations*, IEEE Arithmetic 10, 1992.

[67] 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.

[68] B. A Wichmann, Floating-Point Interval Arithmetic for Validation, NPL Report DITC 76/86, 1986.

[69] B. A. Wichmann, Towards a Formal Definition of Floating Point, Computer Journal, Vol. 32, October 1989, pp.432-436.

[70] B. A. Wichmann, Getting the Correct Answers, NPL Report DITC 167/90, June 1990.

[71] J. Gosling, B. Joy, G. Steele, *The Java Language Specification.*

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

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

[74] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*, The MIT Press, 1997, ISBN: 0-262-63181-4.