INTERNATIONAL STANDARD

ISO/IEC 10967-1

First Edition
1994-12-15

Information technology – Language independent arithmetic –

Part 1:
Integer and floating point arithmetic

Technologies de l'information –
Arithmétique indépendante de langage –

Partie 1: Arithmétique de nombres entiers et en virgule flottante

Reference number
ISO/IEC 10967-1:1994(E)

Contents

ISO/IEC Copyright Office • Case Postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

Annexes

Foreword

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

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

International Standard ISO/IEC 10967-1 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Sub-Committee SC 22, *Programming languages*.

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: Mathematical procedures*
- *Part 3: Complex arithmetic and procedures*

Additional parts will specify other arithmetic types or operations.

Annexes A to J of this part of ISO/IEC 10967 are for information only.

Introduction

The aims

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

It is the intent of this part of ISO/IEC 10967 to help to redress these shortcomings, by setting out precise definitions of integer and floating point data types, and requirements for documentation. This is done in a way that makes as few presumptions as possible about the underlying machine architecture.

It is not claimed that this part of ISO/IEC 10967 will ensure complete certainty of arithmetic behavior in all circumstances; the complexity of numeric software and the difficulties of analysing and proving algorithms are too great for that to be attempted. Rather, the requirements set forth here will provide a firmer basis than hitherto for attempting such analysis.

Hence the first aim of this part of ISO/IEC 10967 is to enhance the predictability and reliability of the behavior 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 data types. 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 behavior will aid programmers designing code intended to run on multiple platforms, and will help in predicting what will happen when such a program is moved from one conforming language processor to another.

Note that this part of ISO/IEC 10967 does not attempt to ensure bit-for-bit identical results when programs are transferred between language processors, or translated from one language into another. 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 part of ISO/IEC 10967 defines the fundamental properties of integer and floating point numbers. These properties are presented in terms of a parameterized model. The parameters allow enough variation in the model so that most platforms are covered, but when a particular set of parameter values is selected, and all required documentation is supplied, the resulting information should be precise enough to permit careful numerical analysis.

The requirements of this part of ISO/IEC 10967 cover three areas. First, the programmer must be given runtime access to the parameters and functions that describe the arithmetic properties of the platform. Second, the executing program must be notified when proper results cannot be returned (e.g., when a computed result is out of range or undefined). Third, the numeric properties of conforming platforms must be publicly documented.

This part of ISO/IEC 10967 focuses on the classical integer and floating point data types. Later parts will consider common mathematical procedures (part 2), complex numbers (part 3), and possibly additional arithmetic types such as fixed point.

Relationship to hardware

ISO/IEC 10967 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 559 (also known as IEEE 754). The floating point requirements of this part of ISO/IEC 10967 are compatible with (and enhance) IEC 559.

This part of ISO/IEC 10967 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 part of ISO/IEC 10967 can lead to the following benefits.

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

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

Programs will be able to determine (at run time) the crucial numeric properties of the implementation. They will be able to reject unsuitable implementations, and (possibly) to correctly characterize the accuracy of their own results. Programs will be able to extract apparently implementation dependent data (such as the exponent of a floating point number) in an implementation independent way. Programs will be able to detect (and possibly correct for) exceptions in arithmetic processing.

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

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

Annex A is intended to be read in parallel with the standard.

Information technology – Language independent arithmetic – Part 1: Integer and floating point arithmetic

1 Scope

This part of ISO/IEC 10967 defines the properties of integer and floating point data types on computer systems to ensure that the processing of arithmetic data can be undertaken in a reliable and predictable manner. Emphasis is placed on documenting the existing variation between systems, not on the elimination of such variation. The requirements of this part of ISO/IEC 10967 shall be in addition to those that may be specified in other standards, such as those for programming languages (See clause 7).

It is not the purpose of this part of ISO/IEC 10967 to ensure that an arbitrary numerical function can be so encoded as to produce acceptable results on all conforming systems. Rather, the goal is to ensure that the properties of arithmetic on a conforming system are made available to the programmer.

Therefore, it is not reasonable to demand that a substantive piece of software run on every implementation that can claim conformity to this part of ISO/IEC 10967.

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

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

1.1 Specifications included in this part of ISO/IEC 10967

This part of ISO/IEC 10967 defines integer and floating point data types. Definitions are included for bounded, unbounded, and modulo integer types, as well as both normalized and denormalized floating point types.

The specification for an arithmetic type includes

a) The set of computable values.

b) The set of computational operations provided, including

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

2) comparison operations on two operands of the same type,
3) conversion operations from any arithmetic type to any other arithmetic type, and
4) operations that access properties of individual values.

c) Program-visible parameters that characterize the values and operations.

d) Procedures for reporting arithmetic exceptions.

NOTE – A.1.3 describes planned future work in this area.

1.2 Specifications not within the scope of this part of ISO/IEC 10967

This part of ISO/IEC 10967 provides no specifications for

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

b) A general unnormalized floating point data type, or the operations on such data. This part of ISO/IEC 10967 neither requires nor excludes such data or operations.

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

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

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

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

g) Floating point values that represent infinity or non-numeric results. However, specifications for such values are given in IEC 559.

h) The properties of integer and floating point data types that properly belong in language standards. Examples include

1) The syntax of literals and expressions.
2) The precedence of operators.
3) The rules of assignment and parameter passing.
4) The presence or absence of automatic type coercions.
5) The consequences of applying an operation to values of improper type, or to uninitialized data.

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 part of ISO/IEC 10967. Internal representations need not be unique, nor is there a requirement for identifiable fields (for sign, exponent, and so on). The value of the exponent bias, if any, is not specified.

2 Conformity

It is expected that the provisions of this part of ISO/IEC 10967 will be incorporated by reference and further defined in other International Standards; specifically in language standards and in language binding standards. Binding standards specify the correspondence between the abstract data types and operations of this part of ISO/IEC 10967 and the concrete language syntax of the language standard. A language standard that explicitly provides such binding information can serve as a binding standard.

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

When no binding standard for a language exists, an implementation conforms to this part of ISO/IEC 10967 if and only if it provides one or more data types that together satisfy all the requirements of clauses 5 through 8. Conformity is relative to that designated set of data types.

NOTES

1 Language bindings are essential. Clause 8 requires an implementation to supply a binding if no binding standard exists. See clause A.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 part of ISO/IEC 10967 will include a binding for IEC 559 as well. See 5.2.9 and annex C.

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

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

3 Normative reference

The following standard contains provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 10967. At the time of publication, the edition indicated was valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 10967 are encouraged to investigate the possibility of applying the most recent edition of the standard indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

IEC 559:1989, *Binary floating-point arithmetic for microprocessor systems.*

4 Symbols and definitions

4.1 Symbols

In this part of ISO/IEC 10967, $\mathcal{Z}$ denotes the set of mathematical integers, $\mathcal{R}$ denotes the set of real numbers, and $\mathcal{C}$ denotes the set of complex numbers. Note that $\mathcal{Z} \subset \mathcal{R} \subset \mathcal{C}$.

All prefix and infix operators have their conventional (exact) mathematical meaning. The conventional notation for set definition and manipulation is also used. In particular this part of ISO/IEC 10967 uses

$\Rightarrow$ and $\Longleftrightarrow$ for logical implication and equivalence
$+$, $-$, $*$, $/$, x^y, $log_x y$, $\sqrt{x}$, $|x|$, $\lfloor x \rfloor$, and $tr(x)$ on reals
$<$, $\leq$, $=$, $\neq$, $\geq$, and $>$ on reals
$\times$, $\cup$, $\in$, $\subset$, and $=$ on sets of integers and reals
max and min on non-empty sets of integers and reals
$\rightarrow$ for a mapping between sets

This part of ISO/IEC 10967 uses $*$ for multiplication, and $\times$ for the Cartesian product of sets.

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

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

and $tr(x)$ designates the integer part of x (truncated toward 0):

$$\begin{aligned} tr(x) &= \lfloor x \rfloor && \text{if } x \geq 0 \\ &= -\lfloor -x \rfloor && \text{if } x < 0 \end{aligned}$$

The type *Boolean* consists of the two values **true** and **false**. Predicates (like $<$ and $=$) produce values of type Boolean.

4.2 Definitions

For the purposes of this part of ISO/IEC 10967, the following definitions apply:

arithmetic data type: A data type whose values are members of $\mathcal{Z}$, $\mathcal{R}$, or $\mathcal{C}$.

NOTE 1 – This part of ISO/IEC 10967 specifies requirements for integer and floating point data types. Complex numbers are not covered here, but will be included in a subsequent part of ISO/IEC 10967 [15].

axiom: A general rule satisfied by an operation and all values of the data type to which the operation belongs. As used in the specifications of operations, axioms are requirements.

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. (Contrast with *exceptional value*. See 6.1.2.)

NOTE 2 – The infinities and NaNs produced by an IEC 559 system are examples of continuation values.

data type: A set of values and a set of operations that manipulate those values.

denormalization loss: A larger than normal rounding error caused by the fact that denormalized values have less than full precision. (See 5.2.5 for a full definition.)

denormalized: Those values of a floating point type F that provide less than the full precision allowed by that type. (See F_D in 5.2 for a full definition.)

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

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

exception: The inability of an operation to return a suitable numeric result. This might arise because no such result exists mathematically, or because the mathematical result cannot be represented with sufficient accuracy.

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

NOTES

3 Exceptional values are used as part of the defining formalism only. With respect to this part of ISO/IEC 10967, they do not represent values of any of the data types 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 559. Contrast this definition with that of *continuation value* above.

exponent bias: A number added to the exponent of a floating point number, usually to transform the exponent to an unsigned integer.

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

implementation (of this part of ISO/IEC 10967)**:** The total arithmetic environment presented to a programmer, including hardware, language processors, exception handling facilities, subroutine libraries, other software, and all pertinent documentation.

normalized: 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.)

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.)

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

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

rounding: The act of computing a representable final result for an operation that is close to the exact (but unrepresentable) result for that operation. Note that a suitable representable result may not exist (see 5.2.6). (See also A.5.2.5 for some examples.)

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

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

Note that if $u \in \mathcal{R}$ is between two adjacent values in X, $rnd(u)$ selects one of those adjacent values.

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

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

round toward zero: 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 0.

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 [2].)

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 [2].)

signature (of a function or operation)**:** A summary of information about an operation or function. A signature includes the operation name, the minimum set of inputs to the operation, and the maximum set of outputs from the operation (including exceptional values if any). The signature

$$add_I : I \times I \rightarrow I \cup \{\mathbf{integer\_overflow}\}$$

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

A signature for an operation or function does not forbid the operation from accepting a wider range of inputs, nor does it guarantee that every value in the output range will actually be returned for some input. An operation given inputs outside the stipulated input range may produce results outside the stipulated output range.

5 The arithmetic types

A type consists of a set of values and a set of operations that manipulate these values. For any particular type, the set of 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 type's value set (V), the type's operations will be specified as a collection of mathematical functions on V. These functions typically return values in V, but they may instead return certain "exceptional" values that are not in any arithmetic type. The exceptional values are **integer_overflow**, **floating_overflow**, **underflow**, and **undefined**.

NOTES

1 Exceptional values are used as part of the defining formalism only. With respect to this part of ISO/IEC 10967, they do not represent values of any of the data types described. There is no requirement that they be represented or stored in the computing system. They are not used in subsequent arithmetic operations.

2 The values *Not-a-Number* and *infinity* introduced in IEC 559 (also known as IEEE 754) are not considered exceptional values for the purposes of this part of ISO/IEC 10967. They are "continuation values" as defined in 6.1.2.

Whenever an arithmetic operation (as defined in this clause) returns an exceptional value, notification of this shall occur as described in clause 6.

Each operation has a signature which describes its inputs and outputs (including exceptional values). Each operation is further defined by one or more axioms.

An implementation of a conforming integer or floating point type shall include all the values defined for that type in this part of ISO/IEC 10967. Additional numeric values shall not be included in such a type. However, an implemented type is permitted to include additional non-numeric values (for example, continuation values representing exceptions).

NOTE 3 – This part of ISO/IEC 10967 does not define the behavior of operations when applied to such additional non-numeric values. Other standards (such as IEC 559) do define such behavior.

An implementation of a conforming integer or floating point type shall include all the operations defined for that type in this part of ISO/IEC 10967. Additional operations are explicitly permitted.

The type Boolean is used to specify parameter values and the results of comparison operations. An implementation is not required to provide a Boolean type, nor is it required to provide operations on boolean values. However, an implementation shall provide a means of distinguishing **true** from **false** as parameter values and as results of operations.

NOTE 4 – This part of ISO/IEC 10967 requires an implementation to provide "means" or "methods" to access values, operations, or other facilities. Ideally, these methods are provided by a language or binding standard, and the implementation merely cites these standards. Only if a binding standard does not exist, must an individual implementation supply this information on its own. See clause A.7.

5.1 Integer types

An integer type I shall be a subset of $\mathcal{Z}$, characterized by four parameters:

$bounded \in Boolean$	(whether the set I is finite)
$modulo \in Boolean$	(whether out-of-bounds results "wrap")
$minint \in I$	(the smallest integer in I)
$maxint \in I$	(the largest integer in I)

If *bounded* is **false**, the set I satisfies

$$I = \mathcal{Z}$$

In this case, *modulo* shall be **false**, and the values of *minint* and *maxint* are not meaningful.

If *bounded* is **true**, the set I satisfies

$$I = \{x \in \mathcal{Z} \mid minint \le x \le maxint\}$$

and *minint* and *maxint* shall satisfy

$$maxint > 0$$

and one of:
$$minint = 0$$
$$minint = -(maxint)$$
$$minint = -(maxint + 1)$$

An integer type with $minint < 0$ is called *signed*. An integer type with $minint = 0$ is called *unsigned*. An integer type in which *bounded* is **false** is *signed*.

NOTES

1 Most traditional programming languages call for bounded integers. Others allow an integer type to have an unbounded range. A few languages permit the implementation to decide whether an integer type will be bounded or unbounded. (See A.5.1.0.3 for further discussion.)

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

An implementation may provide more than one integer type. A method shall be provided for a program to obtain the values of the parameters *bounded*, *modulo*, *minint*, and *maxint*, for each integer type provided.

NOTE 3 – If the value of a parameter (like *bounded*) is dictated by a language standard, implementations of that language need not provide program access to that parameter explicitly.

5.1.1 Operations

For each integer type, the following operations shall be provided:

add_I: $I \times I \to I \cup \{\mathbf{integer\_overflow}\}$
sub_I: $I \times I \to I \cup \{\mathbf{integer\_overflow}\}$
mul_I: $I \times I \to I \cup \{\mathbf{integer\_overflow}\}$
div_I: $I \times I \to I \cup \{\mathbf{integer\_overflow}, \mathbf{undefined}\}$
rem_I: $I \times I \to I \cup \{\mathbf{undefined}\}$
mod_I: $I \times I \to I \cup \{\mathbf{undefined}\}$
neg_I: $I \to I \cup \{\mathbf{integer\_overflow}\}$
abs_I: $I \to I \cup \{\mathbf{integer\_overflow}\}$
$sign_I$: $I \to I$
eq_I: $I \times I \to Boolean$
neq_I: $I \times I \to Boolean$
lss_I: $I \times I \to Boolean$
leq_I: $I \times I \to Boolean$
gtr_I: $I \times I \to Boolean$
geq_I: $I \times I \to Boolean$

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

5.1.2 Modulo integers versus overflow

If *bounded* is **true**, the mathematical operations $+$, $-$, $*$, and $/$ (after rounding) can produce results that lie outside the set I. In such cases, the computational operations add_I, sub_I, mul_I, and div_I shall either cause a notification (if $modulo = \mathbf{false}$), or return a "wrapped" result (if $modulo = \mathbf{true}$).

The helper function

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

(which produces the wrapped result) is defined as follows:

$$wrap_I(x) = x + j * (maxint - minint + 1) \qquad \text{for some } j \text{ in } \mathcal{Z}$$

$$wrap_I(x) \in I$$

5.1.3 Axioms

For all values x and y in I, the following shall apply:

$$
\begin{aligned}
add_I(x,y) &= x+y && \text{if } x+y \in I\\
&= wrap_I(x+y) && \text{if } x+y \notin I \text{ and } modulo = \mathbf{true}\\
&= \mathbf{integer\_overflow} && \text{if } x+y \notin I \text{ and } modulo = \mathbf{false}
\end{aligned}
$$

$$
\begin{aligned}
sub_I(x,y) &= x-y && \text{if } x-y \in I\\
&= wrap_I(x-y) && \text{if } x-y \notin I \text{ and } modulo = \mathbf{true}\\
&= \mathbf{integer\_overflow} && \text{if } x-y \notin I \text{ and } modulo = \mathbf{false}
\end{aligned}
$$

$$
\begin{aligned}
mul_I(x,y) &= x*y && \text{if } x*y \in I\\
&= wrap_I(x*y) && \text{if } x*y \notin I \text{ and } modulo = \mathbf{true}\\
&= \mathbf{integer\_overflow} && \text{if } x*y \notin I \text{ and } modulo = \mathbf{false}
\end{aligned}
$$

An implementation shall provide one or both of the operation pairs div_I^f/rem_I^f and div_I^t/rem_I^t. (The functions $\lfloor\,\rfloor$ and $tr()$ are defined in 4.1.)

$$
\begin{aligned}
div_I^f(x,y) &= \lfloor x/y \rfloor && \text{if } y \neq 0 \text{ and } \lfloor x/y \rfloor \in I\\
&= wrap_I(\lfloor x/y \rfloor) && \text{if } y \neq 0 \text{ and } \lfloor x/y \rfloor \notin I \text{ and } modulo = \mathbf{true}\\
&= \mathbf{integer\_overflow} && \text{if } y \neq 0 \text{ and } \lfloor x/y \rfloor \notin I \text{ and } modulo = \mathbf{false}\\
&= \mathbf{undefined} && \text{if } y = 0
\end{aligned}
$$

$$
\begin{aligned}
rem_I^f(x,y) &= x - (\lfloor x/y \rfloor * y) && \text{if } y \neq 0\\
&= \mathbf{undefined} && \text{if } y = 0
\end{aligned}
$$

$$
\begin{aligned}
div_I^t(x,y) &= tr(x/y) && \text{if } y \neq 0 \text{ and } tr(x/y) \in I\\
&= wrap_I(tr(x/y)) && \text{if } y \neq 0 \text{ and } tr(x/y) \notin I \text{ and } modulo = \mathbf{true}\\
&= \mathbf{integer\_overflow} && \text{if } y \neq 0 \text{ and } tr(x/y) \notin I \text{ and } modulo = \mathbf{false}\\
&= \mathbf{undefined} && \text{if } y = 0
\end{aligned}
$$

$$
\begin{aligned}
rem_I^t(x,y) &= x - (tr(x/y) * y) && \text{if } y \neq 0\\
&= \mathbf{undefined} && \text{if } y = 0
\end{aligned}
$$

An implementation shall provide one or both of mod_I^a and mod_I^p.

$$
\begin{aligned}
mod_I^a(x,y) &= x - (\lfloor x/y \rfloor * y) && \text{if } y \neq 0\\
&= \mathbf{undefined} && \text{if } y = 0
\end{aligned}
$$

$$
\begin{aligned}
mod_I^p(x,y) &= x - (\lfloor x/y \rfloor * y) && \text{if } y > 0\\
&= \mathbf{undefined} && \text{if } y \leq 0
\end{aligned}
$$

$$
\begin{aligned}
neg_I(x) &= -x && \text{if } -x \in I\\
&= wrap_I(-x) && \text{if } -x \notin I \text{ and } modulo = \mathbf{true}\\
&= \mathbf{integer\_overflow} && \text{if } -x \notin I \text{ and } modulo = \mathbf{false}
\end{aligned}
$$

$$
\begin{aligned}
abs_I(x) &= |x| && \text{if } |x| \in I\\
&= wrap_I(|x|) && \text{if } |x| \notin I \text{ and } modulo = \mathbf{true}\\
&= \mathbf{integer\_overflow} && \text{if } |x| \notin I \text{ and } modulo = \mathbf{false}
\end{aligned}
$$

$$
\begin{aligned}
sign_I(x) &= \ \ 1 && \text{if } x > 0\\
&= \ \ 0 && \text{if } x = 0\\
&= -1 && \text{if } x < 0
\end{aligned}
$$

$$eq_I(x,y) = \mathbf{true} \iff x = y$$

$$neq_I(x,y) = \mathbf{true} \iff x \neq y$$

$$lss_I(x,y) = \textbf{true} \iff x < y$$
$$leq_I(x,y) = \textbf{true} \iff x \le y$$
$$gtr_I(x,y) = \textbf{true} \iff x > y$$
$$geq_I(x,y) = \textbf{true} \iff x \ge y$$

5.2 Floating point types

NOTE 1 – This part of ISO/IEC 10967 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 A.5.2.

A floating point type F shall be a finite subset of $\mathcal{R}$, characterized by five parameters:

$r \in \mathcal{Z}$	(the radix of F)
$p \in \mathcal{Z}$	(the precision of F)
$emin \in \mathcal{Z}$	(the smallest exponent of F)
$emax \in \mathcal{Z}$	(the largest exponent of F)
$denorm \in Boolean$	(whether F contains denormalized values)

The parameters r and p shall satisfy

$$r \ge 2 \quad \text{and} \quad p \ge 2$$

and r should be even.

The parameters $emin$, $emax$, and p shall satisfy

$$p - 2 \le -emin \le r^p - 1$$
$$p \le emax \le r^p - 1$$

Given specific values for r, p, $emin$, $emax$, and $denorm$, the following sets are defined:

$$F_N = \{0,\ \pm i * r^{e-p} \mid i, e \in \mathcal{Z},\ r^{p-1} \le i \le r^p - 1,\ emin \le e \le emax\}$$
$$F_D = \{\pm i * r^{e-p} \mid i, e \in \mathcal{Z},\ 1 \le i \le r^{p-1} - 1,\ e = emin\}$$
$$\begin{aligned} F &= F_N \cup F_D && \text{if } denorm = \textbf{true} \\ &= F_N && \text{if } denorm = \textbf{false} \end{aligned}$$

The elements of F_N are called *normalized* floating point values because of the constraint $r^{p-1} \le i \le r^p - 1$. The elements of F_D are called *denormalized* floating point values.

NOTE 2 – The terms *normalized* and *denormalized* refer to the mathematical values involved, not to any method of representation.

The type F is called *normalized* if it contains only normalized values, and called *denormalized* if it contains denormalized values as well.

An implementation may provide more than one floating point data type. A method shall be provided for a program to obtain the values of the parameters r, p, $emin$, $emax$, and $denorm$, for each floating point type provided.

NOTE 3 – The conditions placed upon the parameters r, p, $emin$, and $emax$ are sufficient to guarantee that the abstract model of F is well-defined and contains its own parameters. More stringent conditions are needed to produce a computationally useful floating point type. These are design decisions which are beyond the scope of this part of ISO/IEC 10967. (See A.5.2.)

The following set is an unbounded extension of F:

$$F^* = F_N \cup F_D \cup \{\ \pm i * r^{e-p} \mid i, e \in \mathcal{Z},\ r^{p-1} \le i \le r^p - 1,\ e > emax\}$$

NOTE 4 – The set F^* contains values of magnitude larger than those that are representable in the type F. F^* will be used in defining rounding.

5.2.1 Range and granularity constants

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

$$fmax = max\ \{z \in F \mid z > 0\} = (1 - r^{-p}) * r^{emax}$$

$$fmin_N = min\ \{z \in F_N \mid z > 0\} = r^{emin-1}$$

$$fmin_D = min\ \{z \in F_D \mid z > 0\} = r^{emin-p}$$

$$\begin{aligned} fmin = min\ \{z \in F \mid z > 0\} &= fmin_D \quad \text{if } denorm = \mathbf{true} \\ &= fmin_N \quad \text{if } denorm = \mathbf{false} \end{aligned}$$

$$epsilon = r^{1-p} \qquad \text{(the maximum relative spacing in } F_N\text{)}$$

A method shall be provided for a program to obtain the values of the derived constants $fmax$, $fmin$, $fmin_N$, and $epsilon$, for each floating point data type provided.

5.2.2 Operations

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

add_F: $F \times F \to F \cup \{\mathbf{floating\_overflow}, \mathbf{underflow}\}$
sub_F: $F \times F \to F \cup \{\mathbf{floating\_overflow}, \mathbf{underflow}\}$
mul_F: $F \times F \to F \cup \{\mathbf{floating\_overflow}, \mathbf{underflow}\}$
div_F: $F \times F \to F \cup \{\mathbf{floating\_overflow}, \mathbf{underflow}, \mathbf{undefined}\}$
neg_F: $F \to F$
abs_F: $F \to F$
$sign_F$: $F \to F$
$exponent_F$: $F \to J \cup \{\mathbf{undefined}\}$
$fraction_F$: $F \to F$
$scale_F$: $F \times J \to F \cup \{\mathbf{floating\_overflow}, \mathbf{underflow}\}$
$succ_F$: $F \to F \cup \{\mathbf{floating\_overflow}\}$
$pred_F$: $F \to F \cup \{\mathbf{floating\_overflow}\}$
ulp_F: $F \to F \cup \{\mathbf{underflow}, \mathbf{undefined}\}$
$trunc_F$: $F \times J \to F$
$round_F$: $F \times J \to F \cup \{\mathbf{floating\_overflow}\}$
$intpart_F$: $F \to F$
$fractpart_F$: $F \to F$
eq_F: $F \times F \to Boolean$
neq_F: $F \times F \to Boolean$
lss_F: $F \times F \to Boolean$
leq_F: $F \times F \to Boolean$
gtr_F: $F \times F \to Boolean$
geq_F: $F \times F \to Boolean$

J shall be a data type containing all integer values from $-(emax - emin + p - 1)$ to $(emax - emin + p - 1)$ inclusive.

NOTES

1 J should be a conforming integer type (if practical). However, a floating point type will suffice.

2 Operations are permitted to accept inputs not listed above. In particular, IEC 559 requires floating point operations to accept infinities and NaNs as inputs. Such values are not in F.

5.2.3 Approximate operations

The operations add_F, sub_F, mul_F, div_F, and $scale_F$ are approximations of exact mathematical operations. They differ from their exact counterparts in that

a) they produce "rounded" results, and

b) they produce notifications.

The axioms for floating point define these approximate operations *as if* they were computed in three stages:

a) Compute the exact mathematical answer. (For addition and subtraction, a close approximation to the exact answer may be used.)

b) Round this answer to p digits of precision. (The precision will be less if the answer is in the denormalized range.)

c) Determine if notification is required.

These stages will be modelled by three helper functions: add_F^* (stage a, for addition only), rnd_F (stage b), and $result_F$ (stage 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) as if it had.

Different floating point types may have different versions of add_F^*, rnd_F, and $result_F$.

5.2.4 Approximate addition

Some hardware implementations of addition compute an approximation to addition that loses information prior to rounding. As a consequence, $x + y = u + v$ may not imply $add_F(x, y) = add_F(u, v)$.

The add_F^* helper function is introduced to model this pre-rounding approximation:

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

For all values u, v, x, and y in F, and i in $\mathcal{Z}$, the following axioms are satisfied by add_F^*:

$$add_F^*(u, v) = add_F^*(v, u)$$

$$add_F^*(-u, -v) = -add_F^*(u, v)$$

$$x \leq u + v \leq y \;\Rightarrow\; x \leq add_F^*(u, v) \leq y$$

$$u \leq v \;\Rightarrow\; add_F^*(u, x) \leq add_F^*(v, x)$$

If u, v, $u * r^i$, and $v * r^i$ are all in F_N,

$add_F^*(u * r^i, v * r^i) = add_F^*(u, v) * r^i$

NOTE – The above five axioms capture the following properties:

a) Add_F^* is commutative.

b) Add_F^* is sign symmetric.

c) Add_F^* is in the same "basic interval" as $u+v$, and is exact if $u+v$ is exactly representable. (A basic interval is the range between two adjacent F values.)

d) Add_F^* is monotonic.

e) Add_F^* does not depend on the exponents of its arguments (only on the difference of the exponents).

Ideally, no information should be lost before rounding. Thus, add_F^* should satisfy

$add_F^*(x, y) = x + y$

5.2.5 Rounding

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

The rnd_F helper function is introduced to model this process:

$rnd_F \colon \mathcal{R} \to F^*$

Rnd_F is a rounding function as defined in 4.2. Rnd_F is sign symmetric. That is, for $x \in \mathcal{R}$,

$rnd_F(-x) = -rnd_F(x)$

For $x \in \mathcal{R}$ and $i \in \mathcal{Z}$, such that $|x| \geq fmin_N$ and $|x * r^i| \geq fmin_N$,

$rnd_F(x * r^i) = rnd_F(x) * r^i$

NOTE – This rule means that the rounding function does not depend on the "exponent" part of the real number *except* when denormalization occurs.

If, for some $x \in \mathcal{R}$ and some $i \in \mathcal{Z}$, such that $|x| < fmin_N$ and $|x * r^i| \geq fmin_N$, the formula

$rnd_F(x * r^i) = rnd_F(x) * r^i$

does *not* hold, then rnd_F is said to have a *denormalization loss* at x.

5.2.6 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:

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

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

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

For $x = 0$ or $fmin_N \leq |x| \leq fmax$:

$result_F(x, rnd) \; = rnd(x)$

For $|x| > fmax$:

$$
\begin{aligned}
result_F(x, rnd) &= rnd(x) && \text{if } |rnd(x)| = fmax \\
&= \textbf{floating\_overflow} && \text{otherwise}
\end{aligned}
$$

For $0 < |x| < fmin_N$:

$$
\begin{aligned}
result_F(x, rnd) &= rnd(x) \text{ or } \textbf{underflow} && \text{if } |rnd(x)| = fmin_N \\
&= rnd(x) \text{ or } \textbf{underflow} && \text{if } |rnd(x)| \in F_D,\ denorm = \textbf{true}, \text{ and} \\
& && \quad rnd \text{ has no denormalization loss at } x \\
&= \textbf{underflow} && \text{otherwise}
\end{aligned}
$$

An implementation is allowed to choose between $rnd(x)$ and **underflow** in the region between 0 and $fmin_N$. However, a denormalized value for $rnd(x)$ can be chosen only if $denorm$ is **true** and no denormalization loss occurs at x. An implementation shall document how the choice between $rnd(x)$ and **underflow** is made.

5.2.7 Axioms

For convenience, define two helper functions: e_F and rn_F.

Define e_F: $\mathcal{R} \rightarrow \mathcal{Z}$ such that

$$
\begin{aligned}
e_F(x) &= \lfloor log_r |x| \rfloor + 1 && \text{if } |x| \geq fmin_N \\
&= emin && \text{if } |x| < fmin_N
\end{aligned}
$$

NOTE 1 – The value $e_F(x)$ is that of the e in the definitions of F_N, F_D, and F^*, and has been defined to have the value $emin$ at 0. When x is in F_D, $e_F(x)$ is $emin$ regardless of x.

Define $rn_F : F \times \mathcal{Z} \rightarrow F^*$ such that

$$
rn_F(x, n) = sign_F(x) * \lfloor |x| / r^{e_F(x)-n} + 1/2 \rfloor * r^{e_F(x)-n}
$$

NOTE 2 – The value $rn_F(x)$ is x rounded to n digits of precision (using traditional round to nearest in which ties round away from zero).

For all values x and y in F, and n an integer in J the following shall apply:

$$
\begin{aligned}
add_F(x, y) &= result_F(add_F^*(x, y), rnd_F) \\
sub_F(x, y) &= add_F(x, -y) \\
mul_F(x, y) &= result_F(x * y, rnd_F) \\
div_F(x, y) &= result_F(x/y, rnd_F) && \text{if } y \neq 0 \\
&= \textbf{undefined} && \text{if } y = 0 \\
neg_F(x) &= -x \\
abs_F(x) &= |x| \\
sign_F(x) &= 1 && \text{if } x > 0 \\
&= 0 && \text{if } x = 0 \\
&= -1 && \text{if } x < 0 \\
exponent_F(x) &= \lfloor log_r |x| \rfloor + 1 && \text{if } x \neq 0 \\
&= \textbf{undefined} && \text{if } x = 0 \\
fraction_F(x) &= x / r^{exponent_F(x)} && \text{if } x \neq 0 \\
&= 0 && \text{if } x = 0
\end{aligned}
$$

$$
\begin{array}{lll}
scale_F(x,n) & = result_F(x * r^n, rnd_F) & \\
succ_F(x) & = min\ \{z \in F \mid z > x\} & \text{if } x \neq \mathit{fmax} \\
 & = \textbf{floating\_overflow} & \text{if } x = \mathit{fmax} \\
pred_F(x) & = max\ \{z \in F \mid z < x\} & \text{if } x \neq -\mathit{fmax} \\
 & = \textbf{floating\_overflow} & \text{if } x = -\mathit{fmax} \\
ulp_F(x) & = r^{e_F(x)-p} & \text{if } x \neq 0 \text{ and } r^{e_F(x)-p} \in F \\
 & = \textbf{underflow} & \text{if } x \neq 0 \text{ and } r^{e_F(x)-p} \notin F \\
 & = \textbf{undefined} & \text{if } x = 0 \\
trunc_F(x,n) & = \lfloor x/r^{e_F(x)-n} \rfloor * r^{e_F(x)-n} & \text{if } x \geq 0 \\
 & = -trunc_F(-x,n) & \text{if } x < 0 \\
round_F(x,n) & = rn_F(x,n) & \text{if } |rn_F(x,n)| \leq \mathit{fmax} \\
 & = \textbf{floating\_overflow} & \text{if } |rn_F(x,n)| > \mathit{fmax} \\
intpart_F(x) & = sign_F(x) * \lfloor |x| \rfloor & \\
\mathit{fractpart}_F(x) & = x - intpart_F(x) &
\end{array}
$$

$$eq_F(x,y) = \textbf{true} \iff x = y$$

$$neq_F(x,y) = \textbf{true} \iff x \neq y$$

$$lss_F(x,y) = \textbf{true} \iff x < y$$

$$leq_F(x,y) = \textbf{true} \iff x \leq y$$

$$gtr_F(x,y) = \textbf{true} \iff x > y$$

$$geq_F(x,y) = \textbf{true} \iff x \geq y$$

5.2.8 Rounding constants

Two derived constants shall be provided to characterize the rounding function: rnd_error and rnd_style.

Define the derived constant rnd_error to be the smallest element of F such that

$$|x - rnd_F(x)| \leq rnd\_error * r^{e_F(rnd_F(x))-p}$$

for all $x \in \mathcal{R}$. If $add_F^*(x,y)$ is not identically equal to $x+y$ for all $x, y \in F$, then rnd_error shall be defined as 1. (See add_F^* in 5.2.4.)

NOTE – The requirement that rnd_F be a rounding function implies that $rnd\_error \leq 1$. However, some definitions for rnd_F may yield smaller values for rnd_error. (See A.5.2.8.)

A method shall be provided for a program to obtain the value of the derived constant rnd_error for each floating point data type provided.

Rnd_F has the *round toward zero* property if for $x \in \mathcal{R}$

$$|rnd_F(x)| \leq |x|$$

Rnd_F has the *round to nearest* property if for $x \in \mathcal{R}$

$$|rnd_F(x) - x| \leq \tfrac{1}{2} * r^{e_F(x)-p}$$

Note that the behavior when x is exactly halfway between values in F is not specified by this definition.

The derived constant *rnd_style*, having one of three allowed constant values, is defined by

rnd_style	= **nearest**	if rnd_F has the round to nearest property
	= **truncate**	if rnd_F has the round toward zero property
	= **other**	otherwise

If $add_F^*(x, y)$ is not identically equal to $x + y$ for all $x, y \in F$, then *rnd_style* shall be defined as **other**.

A method shall be provided for a program to obtain the value of the derived constant *rnd_style* for each floating point data type provided. In addition, a notation for each of the values **nearest**, **truncate**, and **other** shall be provided such that the value of *rnd_style* can be compared to the constants.

5.2.9 Conformity to IEC 559

One further behavioral parameter shall be provided for each floating point type F:

$iec\_559 \in Boolean$ (whether F conforms to IEC 559)

The parameter *iec_559* shall be true only when the type F completely conforms to the requirements of IEC 559 (also known as IEEE 754). F may correspond to any of the floating point types defined in IEC 559.

When *iec_559* is **true**, all the facilities required by IEC 559 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.

NOTE 1 – The IEC 559 facilities include values for infinities and NaNs, extended comparisons, program control of rounding, an inexact exception flag, and so on. See annex C for more information.

When *iec_559* is **true**, all operations and values common to this part of ISO/IEC 10967 and IEC 559 shall satisfy the requirements of both standards. Values present only in IEC 559 (-0, $+\infty$, $-\infty$, and the NaNs) need only satisfy the requirements of IEC 559. The value set F as used in the definitions and axioms of 5.2 does not contain these extra values – it only contains the common values. Thus, the axioms in 5.2.7 apply only to the common values. Nevertheless, the operations in 5.2.2 shall not distinguish -0 from $+0$.

NOTE 2 – An implementation of IEC 559 can distinguish -0 from $+0$ only by the use of operations not in 5.2.2, or by the generation or use of values not in F (infinities).

A method shall be provided for a program to obtain the value of the parameter *iec_559* for each floating point data type provided.

5.3 Conversion operations

A conversion operation is a function from one arithmetic type to another arithmetic type. Conversion operations shall be provided

a) between any two distinct integer types,

b) between any two distinct floating point types of the same radix,

c) from any integer type to any floating point type, and

d) from any floating point type to any integer type.

NOTE 1 – This part of ISO/IEC 10967 does not define conversion operations between floating point types with different radices.

Let I_a and I_b be two integer types. The conversion operation

$$cvt_{I_a \to I_b} : I_a \to I_b \cup \{\mathbf{integer\_overflow}\}$$

shall be defined by

$$\begin{array}{lll} cvt_{I_a \to I_b}(x) & = x & \text{if } x \in I_b \\ & = wrap_{I_b}(x) & \text{if } x \notin I_b \text{ and } modulo_{I_b} = \mathbf{true} \\ & = \mathbf{integer\_overflow} & \text{if } x \notin I_b \text{ and } modulo_{I_b} = \mathbf{false} \end{array}$$

where $modulo_{I_b}$ is the *modulo* parameter for type I_b.

Let $nearest_X$ be a helper rounding function from $\mathcal{R}$ to X satisfying the round to nearest property.

Let F_a and F_b be two floating point types with the same radix. The conversion operation

$$cvt_{F_a \to F_b} : F_a \to F_b \cup \{\mathbf{floating\_overflow}, \mathbf{underflow}\}$$

shall be defined by

$$cvt_{F_a \to F_b}(x) = result_{F_b}(x, nearest_{F_b})$$

Let I be an integer type, and F be a floating point type. The conversion operation

$$cvt_{I \to F} : I \to F \cup \{\mathbf{floating\_overflow}\}$$

shall be defined by

$$cvt_{I \to F}(x) = result_F(x, nearest_F)$$

Let F be a floating point type, and I be an integer type. The conversion operation

$$cvt_{F \to I} : F \to I \cup \{\mathbf{integer\_overflow}\}$$

shall be defined by

$$\begin{array}{lll} cvt_{F \to I}(x) & = rnd_{F \to I}(x) & \text{if } rnd_{F \to I}(x) \in I \\ & = wrap_I(rnd_{F \to I}(x)) & \text{if } rnd_{F \to I}(x) \notin I \text{ and } modulo_I = \mathbf{true} \\ & = \mathbf{integer\_overflow} & \text{if } rnd_{F \to I}(x) \notin I \text{ and } modulo_I = \mathbf{false} \end{array}$$

where $modulo_I$ is the *modulo* parameter for type I and $rnd_{F \to I}$ is a helper rounding function from $\mathcal{R}$ to $\mathcal{Z}$.

NOTE 2 – With proper choice of $rnd_{F \to I}$, the function $cvt_{F \to I}$ could be identical with the function *floor*, *truncate*, *round*, or *ceiling*. These functions will be described in more detail in Part 2 of ISO/IEC 10967.

An implementation may provide more than one conversion operation for a given pair of types. In particular, different choices for $rnd_{F \to I}$ or $nearest_F$ can produce different conversion operations.

6 Notification

Notification is the process by which a user or program is informed that an arithmetic operation cannot be performed. Specifically, a notification shall occur when any arithmetic operation returns an exceptional value as defined in clause 5.

6.1 Notification alternatives

Three alternatives for notification are provided here. The requirements are:

a) The alternative in 6.1.1 shall be supplied in conjunction with any language which provides support for exception handling.

b) The alternative in 6.1.2 shall be supplied in the absence of language support for exception handling.

c) The alternative in 6.1.3 shall be supplied by all implementations.

6.1.1 Language defined notification

If the programming language in use defines an exception handling mechanism that can

a) detect the occurrence of arithmetic exceptions,

b) report such exceptions to the executing program,

c) permit the programmer to specify code to compensate for such exceptions, and then

d) continue program execution,

then notifications shall be handled by that language defined mechanism.

Such a mechanism may be defined as part of the programming language standard itself or by a separate binding standard.

> NOTE – The exception handling mechanisms of Ada and PL/I are examples of language defined notification. In these languages, an exception causes a prompt alteration of control flow to execute user provided exception handling code. Other notification mechanisms, such as continued execution with special non-numeric "error values," may be appropriate for other languages.

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 6.1.3.)

6.1.2 Recording of indicators

An implementation shall provide this alternative for any language that does not provide a mechanism for the handling of exceptions. It is allowed (with system support) even in the presence of such a mechanism.

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

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

These indicators shall be clear at the start of the program. They are set when any arithmetic operation returns an exceptional value as defined in clause 5. Once set, an indicator shall be cleared only by explicit action of the program. The implementation shall not allow a program to complete successfully with an indicator that is set. Unsuccessful completion of a program shall be reported to the user of that program in an unambiguous and "hard to ignore" manner. (See 6.1.3.)

NOTE 1 – The status flags required by IEC 559 are an example of this form of notification, *provided* that the program is not allowed to terminate successfully with any status flags still set.

Consider a set E including at least four elements corresponding to the four exceptional values: **integer_overflow**, **floating_overflow**, **underflow**, and **undefined**. Let *Ind* be be a type whose values represent the subsets of E.

An implementation shall provide an embedding of *Ind* into a programming language type. In addition, a method shall be provided for denoting each of the values of *Ind* (either as constants or via computation).

The following four operations shall be provided:

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

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.

An implementation is permitted to expand the set E to include additional notification indicators beyond the four listed above.

When any arithmetic operation returns an exceptional value 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:

a) In the case of **underflow** (that is, when $result_F(x, rnd) =$ **underflow**), the continuation value shall be $rnd(x)$ when $denorm =$ **true**, and 0 when $denorm =$ **false**.

b) In the case of **integer_overflow**, **floating_overflow**, and **undefined**, the continuation value shall be implementation defined. There are no restrictions on this continuation value. It is not required to be a valid value of the type I or F.

NOTES

2 The infinities and NaNs produced by an IEC 559 system are examples of values not in F which might be used as continuation values. If the *iec_559* parameter is **true**, the continuation values must be precisely those stipulated in IEC 559.

3 It is not specified by this part of ISO/IEC 10967 what happens when an operation is applied to a value that is not in its input domain (as defined by the operation signature). Thus, for example, the behavior of add_F on a NaN is not in the scope of this part of ISO/IEC 10967.

4 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.

6.1.3 Termination with message

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

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

6.2 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 recording of indicators as described in 6.1.2, a significant program event is an attempt by the program (or system) to access the indicators, or the termination of the program. For language defined notification as described in 6.1.1, 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 as described in 6.1.3, 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).

6.3 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 data types within the context of a standard programming language. The requirements of this part of ISO/IEC 10967 shall be in addition to those imposed by the relevant programming language standards.

This part of ISO/IEC 10967 does not define the syntax of arithmetic expressions. However, programmers need to know how to reliably access the operations defined in this part of ISO/IEC 10967.

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 [5] and Ada [6]
`i == j`	in C [9] and Fortran [3]
`i .EQ. j`	in Fortran [3]
`(= i j)`	in Common Lisp [32]

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

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

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

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

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 data type 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 "$2 * x$" $\rightarrow$ "$x + x$" or "x/c" $\rightarrow$ "$x * (1/c)$."

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 range of permitted transformations need be documented. It is not necessary to describe the specific choice of transformations that will be applied to a particular expression. (See the Fortran standard [3], particularly clauses 7.1.2 and 7.1.7, for an example of documentation in this area.)

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

NOTES

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

2 Some of the following items should *not* be standardized. See clause A.7 for a discussion of this topic.

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

b) For each integer type, the values of the parameters: *bounded*, *modulo*, *minint*, and *maxint*. (See 5.1.)

c) For each floating point type, the values of the parameters: r, p, *emin*, *emax*, *denorm*, and *iec_559*. (See 5.2.)

d) For each integer type I, which (or both) of the two permitted div_I and rem_I pairs are provided for that type, and which (or both) of the two mod_I functions are provided for that type. (See 5.1.3.)

e) 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 5.1.3.)

f) For each floating point type F, the full definitions of rnd_F, $result_F$, and add^*_F. (See 5.2.5, 5.2.6, and 5.2.4.) (This should include values for *rnd_error* and *rnd_style*.)

g) For each floating point type F, the type J used with the four operations $exponent_F$, $scale_F$, $trunc_F$, and $round_F$. (See 5.2.2.)

h) For each pair of types, a list of conversion operations provided including the semantics of each $rnd_{F \to I}$ and $nearest_F$ function. (See 5.3.)

i) The notation for invoking each operation provided by this part of ISO/IEC 10967. (See 5.1.1 and 5.2.2.)

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

k) For each integer type, the method for a program to obtain the values of the parameters: *bounded*, *modulo*, *minint*, and *maxint*. (See 5.1.)

l) For each floating point type, the method for a program to obtain the values of the parameters: *r*, *p*, *emin*, *emax*, *denorm*, and *iec_559*. (See 5.2.)

m) For each floating point type, the method for a program to obtain the values of the derived constants *fmax*, *fmin*, $fmin_N$, *epsilon*, *rnd_error*, and *rnd_style*, and the notation for the three values of *rnd_style*. (See 5.2.1 and 5.2.8.)

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

o) The means for selecting among the notification methods, and the notification method used in the absence of a user selection. (See 6.3.)

p) When "recording of 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 6.1.2.)

q) For each floating point type where *iec_559* is **true**, and for each "implementor choice" permitted by IEC 559, the exact choice made. (See 5.2.9.)

r) For each floating point type where *iec_559* is **true**, and for each of the facilities required by IEC 559, the method available to the programmer to exercise that facility. (See 5.2.9 and annex C.)

Annex A
(informative)

Rationale

This annex explains and clarifies some of the ideas behind ISO/IEC 10967-1, *Information technology – Language independent arithmetic – Part 1: Integer and floating point arithmetic* (LIA-1). This allows the standard itself to be 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. As a consequence, the clause numbering in this annex deviates from conventional ISO usage.

A.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 easily be provided in hardware or software. An important aspect of all of these primitive operations is that they are to be regarded as *atomic* rather than implemented as a sequence of yet more primitive operations. Hence, each primitive floating point operation has a one *ulp* error bound, and primitive operations are never interrupted by an intermediate notification.

LIA-1 provides a parameterized 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 the LIA-1 model – the model covers too wide a range for that. It does make sense to write code that uses the LIA-1 facilities to determine whether the platform it's running on is suitable for its needs.

A.1.1 Specifications included in this part of ISO/IEC 10967

This part of ISO/IEC 10967 is intended to define the meaning of an "integer type" and a "floating point type," but not to preclude other arithmetic or related types.

The specifications for integer and floating point types are given in sufficient detail to

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

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

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

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

A.1.2 Specifications not within the scope of this part of ISO/IEC 10967

There are many arithmetic systems, such as fixed point arithmetic, significance arithmetic, interval arithmetic [33], 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 data types in standard languages; most are not. A form of floating point is defined by Kulisch and Miranker [28, 29] which is compatible with (but considerably stricter than) 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 the subject of another standard, ASN.1 [7].

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

A.1.3 Proposed follow-ons to this part of ISO/IEC 10967

It is planned that the following topics be the subject of a family of standards, of which LIA-1 is the first member:

a) Specifications for the usual elementary functions [14].

b) Specifications for converting arithmetic values to and from text strings, particularly for I/O [14].

c) Specifications for converting between floating point types of different radix [14].

d) Specifications for complex data types [15].

This list is incomplete, and no ordering should be inferred.

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 will be used in the development 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.

In connection with the third point, the accuracy properties of the primitive operations will be used to arrive at accuracy specifications for the more advanced operations. For radix conversion and operations on complex number types, accuracy specifications comparable to those in LIA-1 are certainly feasible, but may have unacceptable performance penalties. Complete verification of the accuracy of an elementary function may not be possible.

A.2 Conformity

A conforming system consists of an implementation (which obeys the requirements) together with documentation which shows how the implementation conforms to this part of ISO/IEC 10967. 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 the LIA-1 addition operation *add* should be bound to the infix "+" operator.

Most integer arithmetic implementations are expected to conform to the specifications in this part of ISO/IEC 10967.

Most current implementations of floating point can be expected to conform to the specifications in this part of ISO/IEC 10967. In particular, implementations of IEEE 754 [1] 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 types (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 types 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.

A.2.1 Validation

This part of ISO/IEC 10967 gives a very precise description of the properties of integer and floating point types. This will expedite the construction of conformity tests. It is important that objective tests be available. Schryer [30] 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 [24], which includes enhancements based upon experience with Schryer's work [30], but progress here is inhibited by the lack of a standard against which to test.

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* [8], 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.

A.3 Normative references

A.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 [36] a formal definition is given in terms of the Brown model [22]. Since the current proposal differs from the Brown model, the definition in [36] is not appropriate for LIA-1. The production of a formal definition using VDM [27] would nevertheless be useful.

A.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.

This annex uses the conventional notation for open and closed intervals of real numbers, e.g. the interval $[a, b)$ denotes the set $\{x | a \leq x < b\}$.

A.4.2 Definitions

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 [10].

The principle behind notification is that such events in the execution of a program should not go unnoticed. The preferred action 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. Traditional practice is that a notification consists of aborting execution with a suitable error message. The various forms of notification are given names, such as **floating_overflow**, so that they can be distinguished.

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:

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

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.

$$\begin{aligned} u < z < rnd(u) &\;\Rightarrow\; rnd(z) = rnd(u) \\ rnd(u) < z < u &\;\Rightarrow\; rnd(z) = rnd(u) \end{aligned}$$

The last three rules are special cases of the monotonicity requirement

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

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. However, see A.5.2.4 for a discussion of the subtle interaction between $add_F^*(x, y)$ and the rounding function.

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 examples, see A.5.2.5.

Note, the value $rnd(u)$ may not be representable. The $result_F$ function deals with this possibility. (See A.5.2.6 for further discussion.)

There is a precise distinction between *shall* and *should* as used in this part of ISO/IEC 10967: *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.

A.5 The arithmetic types

Each arithmetic type is a subset of the real numbers characterized by a small number of parameters. 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 lists 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 **undefined**).

The presence of an exceptional value in a signature says that the notification may occur in some implementations, but not necessarily in all implementations. For example, integer arithmetic will not overflow in an implementation with unbounded integers. The axioms (5.1.3, 5.2.7) 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 result lies between *minint* and *maxint* and must produce a notification if the mathematically well-defined result lies outside this interval (**integer_overflow**) or if there is no mathematically well-defined result (**undefined**).

A.5.1 Integer types

Most traditional computer 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.

LIA-1 uses the parameter *bounded* to distinguish between implementations which place no restriction on the range of integer data values (*bounded* = **false**) and those that do (*bounded* = **true**). If the integer type I is bounded, then two additional parameters are required, $minint$ and $maxint$. For unbounded integers, $minint$ and $maxint$ would have no meaning, so they are not provided.

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.

A.5.1.0.1 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 $minint$ and $maxint$ must be accessible to an executing program.

The allowed ranges for integers fall into three classes:

a) $minint = 0$, corresponding to *unsigned* integers. The operation neg_I would always produce **integer_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 **integer_overflow**.

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

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

$$minint/(-1) = -minint = maxint + 1 > maxint.$$

The Pascal, Modula-2 and Ada programming languages support subranges of integers. Such subranges typically do not satisfy the rules for $maxint$ and $minint$. However, we do not intend to say that these languages are non-conforming. Each subrange type can be viewed as a subset of an ideal integer type that does conform to our rules. Integer operations are defined on these ideal types, and the subrange constraints only affect the legality of assignment and parameter passing.

A.5.1.0.2 Modulo integers

Modulo integers were introduced 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 integers behave as above, but wrap rather than overflow.

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.

A.5.1.0.3 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* must either be fixed in the language definition or must be available at run-time. Some languages, like Prolog, permit the existence of an upper limit to be an implementation choice.

In an unbounded integer implementation, every mathematical integer is potentially a data object. The actual values computable depend on resource limitations, not on predefined bounds.

LIA-1 does not specify how the unbounded type is implemented. Typical implementations 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 $maxint$ and $minint$. It is important to note that LIA-1 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.

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

A.5.1.1 Operations

A.5.1.2 Modulo integers versus overflow

$Wrap_I$ produces results in the range $[minint, maxint]$. These results are positive for unsigned integer types, but may be negative for signed types.

A.5.1.3 Axioms

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* (div_I^f), and *round toward zero* (div_I^t). Both are allowed by LIA-1. These rounding rules give identical results for $div_I(x, y)$ when x and y have the same sign, but produce different results when the signs differ. For example,

$$div_I^f(-3, 2) = -2 \qquad \text{(flooring division: round toward minus infinity)}$$
$$div_I^t(-3, 2) = -1 \qquad \text{(truncating division: round toward zero)}$$

Div_I^f satisfies an broadly useful recurrence relation:

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

and is the form of division preferred by many mathematicians. Div_I^t is the traditional 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 $div_I(i, N)$. This works fine if div_I^f is used for div_I. However if div_I^t is used, and i can be negative, group 0 will get $2N - 1$ items rather than the desired N. This uneven behavior for negative i can cause subtle program errors, and is a strong reason for preferring the use of div_I^f.

$Rem_I(x, y)$ gives the remainder after division. It is coupled to division by the following identities:

$$x = div_I(x, y) * y + rem_I(x, y) \qquad \text{if } y \neq 0\text{, and no overflow occurs}$$
$$0 \leq |rem_I(x, y)| < |y| \qquad \text{if } y \neq 0$$

Thus, div_I^f and rem_I^f form a logical pair, as do div_I^t and rem_I^t. Note that computing $rem_I(x, y)$ as

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

is not correct because $div_I(x, y)$ can overflow but $rem_I(x, y)$ cannot.

The modulus operation and the remainder operation are quite similar (in fact, mod_I^a is identical to rem_I^f), but they have been selected to satisfy somewhat different identities. In addition, various languages have chosen to extend the definition of the modulus operation, $mod_I(x, y)$, to permit negative values for the second argument, while others (like Pascal) choose to forbid it. LIA-1 introduces two versions of the modulus operation: mod_I^a, which extends the definition of modulus, and mod_I^p, which is undefined when $y < 0$. The modulus operations satisfy the following identities (in the absence of notification):

$$x = mod_I(x, y) + i * y \qquad \text{for some integer } i$$

$$0 \leq mod_I(x, y) < y \qquad \text{if } y > 0$$
$$y < mod_I^a(x, y) \leq 0 \qquad \text{if } y < 0$$

A.5.2 Floating point types

Floating point values are traditionally represented as either zero or

$$X = \pm g * r^e = \pm 0.f_1 f_2 ... f_p * r^e$$

where $0.f_1 f_2 ... f_p$ is the p-digit *fraction* g (represented in base, or radix, r) and e is the exponent.

The exponent e is an integer in $[emin, emax]$. The fraction digits are integers in $[0, r-1]$. If the floating point number is *normalized*, f_1 is not zero, and hence the minimum value of the fraction g is $1/r$ and the maximum value is $1 - r^{-p}$.

This description gives rise to five parameters that completely characterize the values of a floating point type:

> *radix* r: the "base" of the number system.
>
> *precision* p: the number of radix r digits provided by the type.
>
> *emin* and *emax*: the smallest and largest exponent values. They define the range of the type.
>
> *denorm*: (a Boolean) **true** if the type includes *denormalized* values; **false** if not.

The fraction g can also be represented as $i * r^{-p}$, where i is a p-digit integer in the interval $[r^{p-1}, r^p - 1]$. Thus

$$X = \pm g * r^e = \pm (i * r^{-p}) * r^e = \pm i * r^{e-p}$$

This is the form of the floating point values used in defining the finite set F_N.

Note that in some implementations, the exponent e is encoded with a bias added to the true exponent. LIA-1 uses the unbiased, true exponent.

The IEEE standards 754 [1] and 854 [20] present a slightly different model for the floating point type. Normalized floating point numbers are represented as

$$\pm f_0.f_1 ... f_{p-1} * r^e$$

where $f_0.f_1 ... f_{p-1}$ is the p-digit *significand* (represented in radix r, where r is 2 or 10), $f_0 \neq 0$, and e is an integer exponent between a given E_{min} and E_{max}. The minimum value of the significand is 1; the maximum value is $r - 1/r^{p-1}$.

The IEEE significand is equivalent to $g * r$. Consequently, the IEEE E_{max} and E_{min} are one smaller than the *emax* and *emin* given in the LIA-1 model.

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.

A.5.2.0.1 Denormalized numbers

The IEEE standards 754 and 854 and a few non-IEEE implementations include denormalized numbers. LIA-1 models a denormalized floating point number as a real number of the form

$$X = \pm i * r^{emin-p}$$

where i is an integer in the interval $[1, r^{p-1} - 1]$. The corresponding fraction g lies in the interval $[r^{-p}, 1/r - r^{-p}]$; its most significant digit is zero. Denormalized numbers partially fill the "underflow gaps" in $fmin_N$ that occur between $\pm r^{emin-1}$ and 0. Taken together, they comprise the set F_D.

The values in F_D are linearly distributed with the same spacing as the values in the range $[r^{emin-1}, r^{emin})$ in F_N. Thus they have a maximum absolute representation error of r^{emin-p}. However, since denormalized numbers have less than p digits of precision, the relative representation error can vary widely. This relative error varies from $epsilon = r^{1-p}$ at the high end of F_D to 1 at the low end of F_D. Near 0, the relative error increases without bound.

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 canceled 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 denormalized numbers are provided), or F_N (if all numbers are normalized). Thus LIA-1 allows, but does not require, the use of denormalized numbers. See Coonen [23] for a detailed discussion of the properties of denormalized numbers.

A.5.2.0.2 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 data types.

We require that $r \geq 2$ and $p \geq 2$ in order to ensure that we have a meaningful set of values. At present, only 2, 8, 10, and 16 appear to be in use as values for r.

The requirement that $emin \leq 2 - p$ ensures that $epsilon$ is representable in F.

The requirement that $emax \geq p$ ensures that $1/epsilon$ is representable in F. It also implies that all integers from 1 to $r^p - 1$ are exactly representable.

The parameters r and p logically must be less than r^p, so they are automatically in F. The additional requirement that $emax$ and $-emin$ are at most $r^p - 1$ guarantees that $emax$ and $emin$ are in F as well.

A consequence of the above restrictions is that a language binding can choose to report r, p, $emin$, and $emax$ 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 [20] and also are applied to the model and safe numbers of Ada [6]. No such constraints are included in LIA-1, which emphasizes descriptive, rather than prescriptive, specifications for arithmetic. However, the following restrictions have some useful properties:

a) r should be even.

   The most accurate rounding rule (round to nearest) is significantly more expensive to implement when r is odd, and commonly occuring values (like 0.5) cannot be represented exactly.

b) $r^{p-1} \geq 10^6$

   This gives a maximum relative error ($epsilon$) of one in a million. This is easily accomplished by 24 binary or 6 hexadecimal digits.

c) $emin - 1 \leq -k * (p - 1)$ with $k \geq 2$ and k as large an integer as practical.

This guarantees that $epsilon^k$ is in F which makes it easier to simulate higher levels of precision than would be offered directly by the values in the data type.

d) $emax > k * (p - 1)$

This guarantees that $epsilon^{-k}$ is in F and is useful for the same reasons as given above.

e) $-2 \leq (emin - 1) + emax \leq 2$

This guarantees that the geometric mean $\sqrt{fmin_N * fmax}$ of $fmin_N$ and $fmax$ lies between $1/r$ and r. This also means that for "most" x in F_N the reciprocal $1/x$ is also in F_N. One would like to be able to guarantee this for all x. Unfortunately this cannot be done. Consider the reciprocals of $fmin_N$ and $fmax$:

$$1/fmin_N \text{ in } F_N \quad \text{implies} \quad 1/fmin_N \leq fmax$$
$$1/fmax \text{ in } F_N \quad \text{implies} \quad 1/fmax \geq fmin_N$$

Since $fmin_N$ is a power of r and $fmax$ is not, neither equality can hold in the above. Further, with both equalities removed, only one of the remaining inequalities can hold.

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* and *emax*. In particular, even if the minimum and maximum exponent given in the implementation's own model were negatives of one another, the adjusted *emin* and *emax* become asymmetric.

A.5.2.0.3 Radix complement floating point

LIA-1 presents an abstract model for the floating point type, 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 current 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: the most negative. This "most negative" floating point number has no positive counterpart. It belongs to F^*, and its value is $-fmax - ulp_F(fmax)$. Some radix-complement implementations also omit the negative counterpart of $fmin_N$.

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* (the "least negative" floating point number) and *fmax* (the "most negative" floating point number);

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

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

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

e) remove $-x$ from the definitions of sub_F and $trunc_F$, and redefine these operations and also $round_F$ operations to ensure that every floating point number behaves correctly;

f) redefine the $pred_F$ and $succ_F$ operations to treat the "most negative" floating point number properly.

Because of this complexity, LIA-1 does not currently include radix-complement floating point.

Floating point implementations with sign-magnitude or (radix$-$1)-complement fractions can map the floating point numbers directly to the LIA-1 model without these adjustments.

A.5.2.0.4 Infinity and NaNs

The IEEE standards 754 [1] and 854 [20] provide non-numeric values to represent *infinity* and *Not-a-Number*. *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 implementations do not provide infinity or NaNs. Thus, programs that make use of infinity or NaNs will not be portable to systems that do not provide them. Non-portable programs are not in the scope of LIA-1. Therefore, LIA-1 makes no provision for infinity or NaNs. The behavior of operations with an infinity or a NaN as input is not defined by LIA-1. However, be sure to read 5.2.9 and clause C.1.

The handling of arithmetic exceptions by testing results for infinity or NaN is not portable. Therefore, programmers desiring portability to both IEEE and non-IEEE systems should use the notification methods described in clause 6.

A.5.2.0.5 Signed zero

The IEEE standards define both $+0$ and -0. Very few non-IEEE implementations provide the user with two "different" zeros. Even in an IEEE implementation, the two encodings of zero can only be distinguished with operations that are not provided in LIA-1, e.g. use of the IEEE *copysign* function, dividing by zero to obtain signed infinity, or (possibly) converting to a decimal string. Programs that assume $+0$ and -0 are distinct will not be portable to non-IEEE systems. Therefore, LIA-1 makes no distinction between $+0$ and -0.

A.5.2.1 Range and granularity constants

The positive real numbers $fmax$, $fmin$, and $fmin_N$ are interesting boundaries in the set F. $fmax$ is the "overflow threshold." It is the largest value in both F and F_N. $fmin$ is the "underflow threshold." It is the value of smallest magnitude in F. $fmin_N$ is the "denormalization threshold." It is the smallest normalized value in F: the point where the number of significant digits begins to decrease. Finally, $fmin_D$ is the smallest denormalized value, representable only if *denorm* is **true**.

LIA-1 requires that the values of $fmax$, $fmin$, and $fmin_N$ be accessible to an executing program. All non-zero floating point values fall in the range $\pm[fmin, fmax]$, and values in the range $\pm[fmin_N, fmax]$ can be represented with full precision.

The derived constant $fmin_D$ need not be given as a run-time parameter. On an implementation in which denormalized numbers are provided and enabled, the value of $fmin_D$ is $fmin$. If denormalized numbers are not present, the constant $fmin_D$ is not representable, and $fmin = fmin_N$.

The derived constant *epsilon* must also be accessible to an executing program:

$$epsilon = r^{1-p}$$

It is defined as ratio of the weight of the least significant digit of the fraction g, r^{-p}, to the minimum value of g, $1/r$. So *epsilon* can be described as the largest relative representation error for the set of normalized values in F_N.

An alternate definition of *epsilon* currently in use is the smallest floating point number such that the expression "`1+epsilon`" 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 implementation with round-to-positive-infinity, *epsilon* would be $fmin_D$.

A.5.2.2 Operations

This clause describes the floating point operations defined by LIA-1.

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 $sign_F$ returns a floating point $+1$, 0, or -1, depending on whether its argument is positive, zero, or negative.

d) 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 $r^{n-1} \leq x < r^n$, then $exponent_F(x) = n$. $Exponent_F(0)$ is undefined.

e) The operation $fraction_F$ scales its argument (by a power of r) until it is in the range $\pm[1/r, 1)$. Thus, for $x \neq 0$,

$$x = fraction_F(x) * r^{exponent_F(x)}$$

f) The operation $scale_F$ scales a floating point number by an integer power of the radix.

g) The operation $succ_F$ returns the closest element of F greater than the argument, the "successor" of the argument.

h) 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 neighborhood of a "sensitive" point.

i) 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.

j) The operation $trunc_F$ zeros out the low $(p - n)$ digits of the first argument. When $n \leq 0$ then 0 is returned; and when $n \geq p$ the argument is returned.

k) 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. Note that $round_F$ is distinct from the function rnd_F. $Round_F$ is not intended to provide access to machine rounding.

The $trunc_F$ and $round_F$ operations can be used to split a floating point number into a number of "shorter" parts in order to expedite the simulation of multiple precision operations without use of operations at a higher level of precision.

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

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

n) The Boolean operations are atomic operations which never produce a notification, and always return **true** or **false** in accordance with the exact mathematical result.

An implementation can easily provide any of these operations in software. See [34] 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.

Standardizing functions such as $exponent_F$ and ulp_F helps shield programs from explicit dependence on the underlying format.

The requirement that J contains all the integer values in the range $\pm(emax - emin + p - 1)$ means that the integer argument to the function $scale_F$ is in range whenever scaling between values in F. This is a reasonably weak condition which most arithmetic systems easily satisfy.

A.5.2.3 Approximate operations

Let's apply the 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 axiom for multiplication:

$$mul_F(x, y) = result_F(x * y,\ rnd_F)$$

(For technical reasons, the $result_F$ function is defined to compute $rnd_F(x * y)$ internally.)

Note that in reality, step (a) only needs to compute enough of $x * y$ to be able to complete steps (b) and (c), i.e., to produce a rounded result and to decide on overflow and underflow.

The helper functions rnd_F, $result_F$, and add^*_F are the same for all the operations of a given floating point type. Similarly, the constants rnd_error and rnd_style do 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.

A.5.2.4 Approximate addition

The definition of floating point addition and subtraction given in LIA-1 is more complex than for any other arithmetic operation. This is because some highly-optimized machines modify one or both operands *before* computing the sum.

The typical addition/subtraction implementation described below shows the interaction between alignment, negation, and guard digits.

Floating point additions and subtractions form two cases, depending on the signs of the operands: implied additions (addition of operands of the same sign or subtraction of operands with different signs) and implied subtractions (addition with opposite signs, subtraction with like signs). In both implied addition and subtraction, the radix points of the operands are aligned by right-shifting the smaller operand's fraction. In an implied subtraction, the smaller operand must also be negated, but there is a performance cost associated with negating before the alignment shift. To negate after alignment, enough guard digits must be maintained at the right to propagate the borrow correctly even if digits were lost in the alignment shift.

If the design goal is merely 1-ulp accuracy, then a single guard digit is sufficient. This is for implied subtraction – implied addition doesn't need it.

If round toward zero is desired, implied subtraction requires a single guard digit *plus* a "sticky bit" (which records whether any information was lost during alignment). Again, implied addition doesn't need these.

Finally, if round to nearest is desired (either version), both implied addition and implied subtraction need an additional "rounding bit." This bit records the size of the next lower guard digit relative to $r/2$.

Some implementations omit both the rounding bit and the sticky bit. This increases the execution speed and simplifies the hardware design. However, the smaller operand may have lost some precision during the alignment and negation. This slight loss of precision can become visible to the programmer when the computed result unexpectedly rounds to the other one of the two elements of F most closely bracketing the true result. In fact, in such an implementation, the computed result cannot be predicted from the true sum alone, but depends on the exact operands given.

LIA-1 introduces a helper function add^*_F to model such performance optimizations. $Add^*_F(x, y)$ is an approximate sum of x and y. It represents an intermediate stage in the computation of $add_F(x, y)$ – one which occurs *after* x and y have been combined into a single value, but (possibly) *before* all rounding steps have been completed.

Thus, the defining axiom for addition will be

$$add_F \; = result_F(add^*_F(x, y),\ rnd_F) \qquad \text{(the LIA-1 axiom)}$$

rather than

$$add_F \; = result_F(x + y,\ rnd_F) \qquad \text{(a more stringent axiom)}$$

The ideal definition of $add^*_F(x, y)$ is $x + y$. However, as noted above, some implementations of addition are less than ideal.

Add^*_F is constrained by five axioms. These are designed to ensure that $add^*_F(x, y)$ behaves enough like $x + y$ so that the final computed sum $add_F(x, y)$ will satisfy a reasonable set of identities – identities that most real machines actually satisfy. These axioms guarantee that the approximation process does not force the result too far from the mathematical result, does not depend on the order of operands, is monotonic non-decreasing in each operand independently, and behaves well under change of sign. In general, add_F^* will depend on the alignment shift (the difference in exponents), but not on the magnitude of the exponents.

A.5.2.5 Rounding

Floating point operations are rarely exact. The true mathematical result seldom lies in F, so this 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 the exact value is rounded to the appropriate precision, finally a determination is made about overflow or underflow.

The rounding rule is specified by a rounding function rnd_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 $\pm i * r^{e-p}$ where $r^{p-1} \leq i \leq r^p - 1$ (as in F_N) but $e > emax$. The extra values in F^* are unbounded in range, but all have exactly p digits of precision. These are "helper values," and are not representable in the type F.

The requirement of "sign symmetry," $rnd_F(-x) = -rnd_F(x)$, is needed to ensure the arithmetic operations add_F, sub_F, mul_F, and div_F have the expected behavior with respect to sign, as described in A.5.2.12.

In addition to being a rounding function (as defined in 4.2), rnd_F must not depend upon the exponent of its input (except for denormalized values). This is captured by a "scaling rule:"

$$rnd_F(x * r^j) = rnd_F(x) * r^j$$

which holds as long as x and $x * r^j$ have magnitude greater than (or equal to) $fmin_N$.

Denormalized values have a wider relative spacing than normalized values. Thus, the scaling rule above does not hold for all x in the denormalized range. When the scaling rule fails, we say that rnd_F has a *denormalization loss* at x, and the relative error

$$\left|\frac{x - rnd_F(x)}{x}\right|$$

is typically larger than for normalized values.

Within a single exponent range, the rounding function is not further constrained. In fact, an implementation that conforms to LIA-1 could provide a number of rounding rules. Each such rule would give rise to a logically distinct set of floating point operations (or types).

Information about the rounding function is available to the programmer via a pair of derived constants: *rnd_error* and *rnd_style*. See 5.2.8 and A.5.2.8 for an explanation of these constants and a further discussion of rounding.

A.5.2.6 Result function

The rounding function rnd_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 denormalized 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 **floating_overflow** or **underflow** respectively.

The only difference when F does contain denormalized values occurs when rnd returns a denormalized 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 6.1.2.

$Result_F(x, rnd)$ 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 $rnd(x)$.

A.5.2.7 Axioms

Note that the helper function e_F is not the same as the $exponent_F$ operation. They agree on normalized numbers, but differ on denormalized 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 denormalized numbers, $e_F(x)$ is equal to $emin$.

The helper function $rn_F(x, n)$ rounds a floating point number x to n digits of precision (radix r). Values that are exactly half-way between two adjacent n-digit floating point numbers round away from zero.

A.5.2.8 Rounding constants

Two constants are provided to give the programmer access to some information about the rounding function in use. *Rnd_error* describes the maximum rounding error (in ulps), and *rnd_style* places the rounding function into one of three major classes: truncate, round to nearest, and other.

What are the most common rounding rules?

IEEE 754 [1] and 854 [20] 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 neighboring values in F, select the "even" result. That is, for $x \geq 0$ in F and $u = r^{e_F(x)-p}$

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

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, and are possible candidates for rnd_F. The *round toward zero* rule has a *rnd_style* of **truncate**. The two *round to nearest* rules have a *rnd_style* of **nearest**. Rounding rules not listed here have a *round_style* of **other**.

The first three rules give a one-ulp error bound. That is, *rnd_error* is 1. The last two give a half-ulp bound, so *rnd_error* is $\frac{1}{2}$. However, one cannot conclude that *rnd_style* is **truncate** when *rnd_error* is 1, nor that *rnd_style* is **nearest** if *rnd_error* is $\frac{1}{2}$. Most current non-IEEE implementations provide either the third rule or the last rule.

A.5.2.9 Conformity to IEC 559

IEC 559 is the international version of IEEE 754.

Note that "methods shall be provided ... to access each [IEC 559] facility." This means that a complete LIA-1 binding will include a binding for IEC 559 as well.

IEC 559 contains an annex listing a number of recommended functions. While not required, implementations of LIA-1 are encouraged to provide those functions.

A.5.2.10 Relations among floating point types

An implementation may provide more than one floating point type, 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

$$p_1 \le p_2 \le p_3 \cdots$$
$$emin_1 \ge emin_2 \ge emin_3 \cdots$$
$$emax_1 \le emax_2 \le emax_3 \cdots.$$

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

$$2 * p_i \le p_{i+1}$$
$$2 * (emin_i - 1) \ge (emin_{i+1} - 1)$$
$$2 * emax_i \le emax_{i+1}$$

for each pair F_i and F_{i+1} would provide advantages to programmers of numerical software (for floating point types 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.

A.5.2.11 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 [1] takes the highly prescriptive approach, allowing relatively little latitude for variation. It even stipulates much of the representation. The Brown model [22] comes close to the other extreme, even permitting non-deterministic behavior.

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, and the variation in the operation semantics is provided by permitting different rounding functions and small differences in overflow and underflow checking.

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 or $emin$ or $emax$) 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 $x < z$.

A.5.2.12 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, $scale_F$, $cvt_{F' \rightarrow F}$, and $cvt_{I \rightarrow F}$ 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, and no notification occurs,

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

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

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

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

$$|\Phi(x,y) - \phi(x,y)| \le ulp_F^*(\phi(x,y)) \le ulp_F(\Phi(x,y)) \tag{IV}$$

where $ulp_F^*(x) = r^{e_F(x)-p}$ is ulp_F extended to all of $\mathcal{R}$ (not just $F - \{0\}$).

When $\phi(x,y)$ is defined, is in the range $\pm[fmin_N, fmax]$, and no notification occurs,

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

When $\phi(x,y)$ and $\phi(x * r^j, y * r^k)$ are defined, are in the range $\pm[fmin_N, fmax] \cup \{0\}$, and no notification occurs,

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

Rules (I) through (VI) apply to the seven approximate operations add_F, sub_F, mul_F, div_F, $scale_F$, $cvt_{F' \to F}$, and $cvt_{I \to F}$ with one exception. Rule III may fail for add_F and sub_F when the approximate addition function is not equal to the true sum (i.e., $add_F^*(u,x) \ne u + x$, or $add_F^*(v,y) \ne v + y$). Fortunately, the following weaker rules always hold:

$$u \le v \;\Rightarrow\; add_F(u,x) \le add_F(v,x)$$
$$u \le v \;\Rightarrow\; sub_F(u,x) \le sub_F(v,x)$$
$$u \le v \;\Rightarrow\; sub_F(x,u) \ge sub_F(x,v)$$

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):

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

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

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

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

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

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

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

For $x \ne 0$,

$$x \in F_N \;\Rightarrow\; exponent_F(x) \in [emin, emax]$$

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

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

$$r^{exponent_F(x)-1} \le |x| < r^{exponent_F(x)}$$

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

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

$Scale_F(x, n)$ is exact $(= x * r^n)$ if $x * r^n$ is in the range $\pm[fmin_N, fmax] \cup \{0\}$, or if $n \geq 0$ and $|x * r^n| \leq fmax$.

For $x \neq 0$ and $y \neq 0$,

$x = \pm i * ulp_F(x)$ for some integer i which satisfies

$$\begin{array}{ll} r^{p-1} \leq i < r^p & \text{if } x \in F_N \\ 1 \leq i < r^{p-1} & \text{if } x \in F_D \end{array}$$

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

$$x \in F_N \ \Rightarrow\ ulp_F(x) = epsilon * r^{exponent_F(x)-1}$$

Note that if $denorm = \mathbf{true}$, ulp_F is defined on all non-zero floating point values. If $denorm = \mathbf{false}$, ulp_F underflows on all values less than $fmin_N/epsilon$, i.e., on all values for which $e_F(x) < emin + p - 1$.

For $|x| \geq 1$,

$$intpart_F(x) = trunc_F(x, e_F(x)) = trunc_F(x, exponent_F(x))$$

For any x, 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, when no notification occurs,

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

$$\begin{array}{lll} pred_F(x) & = x - ulp_F(x) & \text{if } x \text{ is not } r^n \text{ for any integer } n \geq emin \\ & = x - ulp_F(x)/r & \text{if } x \text{ is } r^n \text{ for some integer } n \geq emin \end{array}$$

$$ulp_F(x) * r^{p-n} = r^{e_F(x)-n} \qquad \text{for any integer } n$$

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

$$r^{exponent_F(x)-1} \leq |trunc_F(x, n)| \leq |x|$$

$$\begin{array}{ll} round_F(x, n) & = trunc_F(x, n), \quad \text{or} \\ & = trunc_F(x, n) + sign_F(x) * ulp_F(x) * r^{p-n} \end{array}$$

A.5.2.13 Precision, accuracy, and error

LIA-1 uses the term *precision* to mean the number of radix r digits in the fraction of a floating point data type. All floating point numbers of a given type are assumed to have the same precision. A denormalized number has the same number of radix r digits, but the presence of leading zeros in its fraction means that fewer of these digits are significant.

In general, numbers of a given data type 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 mathematical 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 computers.

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 "answer" and the "truth."

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. A future part of this International Standard, *Information technology – Language independent arithmetic – Part 2: Mathematical procedures* [14], will deal with items (b), (c), and (d).

A.5.2.13.1 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 5.2.5 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 ulp.

Rounding that results in a denormalized number triggers a loss of significant digits. The result is always exact for an add_F or sub_F operation. However, a denormalized result for a mul_F or div_F operation usually is not exact, which introduces an error of at most one ulp. Because of the loss of significant digits, the relative error due to rounding exceeds that for rounding a normalized result. Hence accuracy of a denormalized result for a mul_F or div_F operation is usually lower than that for a normalized 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 A.5.2.13.3.

A.5.2.13.2 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 – one might even identify a "new" planet!

A.5.2.13.3 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 the 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 $ulp_F(z)$. 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.

A.5.2.14 Extra precision

The use of a higher level of range and/or precision is a time-honored 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 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.

A.5.3 Conversion operations

The conversion operations are easily defined. Four cases arise according to the source and destination types. When both are integer types, the conversion operation preserves the value if it is within the range of the destination type, otherwise the operation gives an overflow notification.

For the conversion to a floating point type, the value is computed by applying the result function with a round-to-nearest rounding rule for the destination type, and the implementation must document which of the possible round-to-nearest rules is being used. Rules (I) through (VI) of A.5.2.12 hold for these conversions. In particular, conversion to a higher precision (or wider range) type is always exact, and never produces a notification.

For the floating point to integer conversions, a special purpose rounding function is applied, which may depend upon both the source and destination types. The function chosen will differ from language to language: Ada chooses round-to-nearest, Pascal provides both round-to-nearest and round-to-zero.

A.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.

A.6.1 Notification alternatives

LIA-1 provides a choice of notification mechanisms to fit the requirements of various programming languages. The first alternative (language defined notification) essentially says "if a programming language already provides an exception handling mechanism, use it." The second alternative (recording of indicators) provides a standard exception handling mechanism for languages that do not already have one. Language or binding standards are expected to choose one of these two as their primary notification mechanism.

The third alternative (termination with message) is provided for use in two situations: (a) when the programmer has not (yet) programmed any exception handling code, 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 violation should be complete and available as close in time to the occurrence of the violation as possible.

A.6.1.1 Language defined notification

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.

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.

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.

Notification mechanisms that automatically alter control flow encourage programmers to consider and compensate for all arithmetic exceptions. Other mechanisms should be designed to encourage this as well. Any suppression of notification should be done only on explicit orders from the programmer.

A.6.1.2 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 any language. 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
overflow	**floating_overflow**
underflow	**underflow**
invalid	**undefined**
division by zero	**undefined**
inexact	(no counterpart)
(no counterpart)	**integer_overflow**

LIA-1 does not include notification for **inexact** because non-IEEE implementations are unlikely to detect inexactness of floating point results.

For a zero divisor, IEEE specifies an **invalid** exception if the dividend is zero, and a **division by zero** 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, LIA-1 specifies **undefined**, regardless of the value of the dividend.

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 violation 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, 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 are a form of global variable. A single thread of computation should see only one copy of these indicators. 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.

The proper way to do this is part of the design of the programming language or threads system, and is not within the scope of LIA-1.

A.6.1.3 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.

A.6.2 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.

A.6.3 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 6.1.2 may differ substantially from the optimal code for 6.1.3. 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 (**floating_overflow**, **underflow**, etc), but this is not required.

If a system had a mode of operation in which exceptions were totally ignored, then for this mode, the system would not conform to LIA-1. 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 behavior 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.

A.7 Relationship with language standards

Language standards vary in the degree to which the underlying data types are specified. For example, Pascal [5] merely gives the largest integer value (*maxint*), while Ada [6] 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 optimization 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. Another part of ISO/IEC 10967 will take 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 A.5.2.0.2 or A.5.2.12,

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 uninitialized 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 LIA-1 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* for a particular floating point type, will frequently vary from implementation to implementation. Other information, like the syntax for accessing the value of *emax*, 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* and *minint* 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 size of *maxint*.

b) The values of r, p, *emin*, *emax*, *denorm*, and (for now) *iec_559* should not be standardized.

   However, it is reasonable to give upper bounds on *epsilon* (r^{1-p}), and bounds on the values of *emin* and *emax*. Certain languages provide decimal floating point types which require $r = 10$.

c) The semantics of rnd_F, $result_F$, and add_F^* should not be 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 behavior of $nearest_F$ on ties should probably not be standardized.

e) The IEC 559 implementor choices should not be limited (except by future revisions of IEC 559).

The allowed translations of expressions into combinations of LIA operations should allow reasonable flexibility for compiler optimization. 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, $rnd_{F\to I}$, $result_F$, add_F^*, e_F, and rn_F may prove useful in defining additional arithmetic operations.

A.8 Documentation requirements

To make good use of an implementation of LIA-1, programmers need to know not only that the implementation conforms, but *how* the implementation conforms. Clause 8 requires implementations to document the binding between the 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^*, or in specifying the translation of arithmetic expressions into combinations of LIA-1 operations.

Compilers often "optimize" code as part of the compilation process. Popular optimizations 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 B
(informative)

Partial conformity

The requirements of LIA-1 have been carefully chosen to be as beneficial as possible, yet be efficiently implemented on almost all existing or anticipated hardware architectures. The bulk of LIA-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 LIA-1 requirements for floating point. The requirements in question are:

a) Strict 1-ulp accuracy of add_F, sub_F, mul_F, and div_F.

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

c) The ability to catch all exceptions, particularly underflow.

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

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

As an example, the Cray family of supercomputers cannot satisfy the first four 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 LIA-1 to provide programmers with the maximum benefit. However, if it is perceived that requiring full conformity to LIA-1 will exclude a significant portion of that language's user community from any benefit, then specifying partial LIA-1 conformity, as permitted in clause 2, may be a reasonable alternative.

Such partial conformity would relax one or more of the five requirements listed above, but would retain the benefits of all other LIA-1 requirements. All deviations from LIA-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 five requirements above is violated.

The following four boolean parameters might be a suitable set:

a) *Strict* – false when any of the LIA-1 requirements on rounding and accuracy are violated. (See 5.2.4 and 5.2.5.)

b) *Silent_underflow* – true when underflow notification is suppressed.

c) *Comparison_via_subtract* – true when comparisons may overflow and underflow like subtraction.

d) *Negate_may_fail* – true when the set of floating point values is not sign symmetric.

Finally, *rnd_error* may be greater than 1 in non-strict implementations.

Annex C
(informative)

IEC 559 bindings

When the parameter *iec_559* is **true** for a floating point type F, all the facilities required by IEC 559 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 559 facilities as well. A programming language binding for a standard such as IEC 559 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 559 facilities (and their definitions) is given in IEC 559. This part of ISO/IEC 10967 does not alter or eliminate any of them. However, to assist the reader, the following summary is offered.

C.1 Summary

A binding of IEC 559 (and thus LIA-1) 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 559 conforming type, the binding should provide:

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

e) A method for denoting at least one quiet NaN (not-a-number).

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

For each IEC 559 conforming type, the binding should provide the notation for invoking each of the following operations:

g) Add_F, sub_F, mul_F, and div_F. (Already required by LIA-1.)

h) Remainder, square-root, and round-to-integral-value.

i) The type conversions $cvt_{F_a \rightarrow F_b}$, $cvt_{F \rightarrow I}$, $cvt_{I \rightarrow F}$. (Already required by LIA-1.)

j) Type conversions between the floating point values and decimal strings (both ways).

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

l) The comparison "unordered." (Optional in IEC 559, but highly desirable.)

m) The other 19 comparison operations. (Optional in IEC 559.)

n) The "recommended functions" copysign, negate, scaleb, logb, nextafter, finite, isnan, <>, and class. (Each is optional in IEC 559. 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):

o) The rounding mode.

p) The five exception flags: inexact, underflow, (floating_)overflow, divide_by_zero, and invalid.

q) The disable/enable flags for each of the five exceptions. (Optional in IEC 559.)

r) The handlers for each of the exceptions. (Optional in IEC 559.)

The binding should provide boolean parameters for each implementor choice allowed by IEC 559:

s) Whether trapping is implemented.

t) Whether tininess is detected "before rounding" or "after rounding."

u) 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 559.

C.2 Notification

One appropriate way to access the five IEC 559 exception flags is to use the functions defined in 6.1.2. This requires extending the set E with three new values: **inexact**, **divide_by_zero**, and **invalid**. (Such an extension is expressly permitted by 6.1.2.) Whenever **divide_by_zero** or **invalid** is set (whether by the system or explicit programmer action), the LIA-1 indicator **undefined** is set as well. Whenever the LIA-1 indicator **undefined** is cleared, **divide_by_zero** and **invalid** are cleared as well.

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.

C.3 Rounding

The two directed roundings of IEC 559, round-toward-positive infinity and round-toward-negative-infinity, do not satisfy the sign symmetry requirement of 5.2.5. However, the default IEC 559 rounding does satisfy the 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 559 rules, not the LIA-1 rules. Such non-conforming modes are expressly permitted by clause 2.

Annex D
(informative)

Requirements beyond IEC 559

Any computing system conforming to the requirements of IEC 559 can economically conform to LIA-1 as well. This annex outlines the LIA-1 requirements that go beyond the requirements of IEC 559.

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

> p, r, *emin*, *emax*, *denorm*, *iec_559*, *fmax*, *fmin*, $fmin_N$, *epsilon*, *rnd_error*, and *rnd_style*

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

> neg_F, abs_F, $sign_F$, $exponent_F$, $fraction_F$, $scale_F$, $succ_F$, $pred_F$, ulp_F, $trunc_F$, $round_F$, $intpart_F$, and $fractpart_F$

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, 6.1.2 requires that notification be done by setting "indicators" which reflect the status flags required by IEC 559. (See annex C as well.)

6.1.3 requires that the programmer can demand prompt program termination on the occurrence of an LIA-1 notification. This is typically implemented using IEC 559 trapping, or (if trapping is unavailable) by compiler generated code.

> NOTE – The LIA-1 notifications correspond to the IEC 559 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 559 will be sufficient for several of the documentation requirements, including requirements (c), (f), and (h). Note that the implementor choices permitted by IEC 559 must be documented.

Annex E
(informative)

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* 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
Basic
C
Common Lisp
Fortran
Modula-2
Pascal and Extended Pascal
PL/I

This list is not exhaustive. Other languages are suitable for conformity to LIA-1.

In this annex, the data types, parameters, constants, operations, and exception behavior 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. Substantial additional suggestions to language developers are presented in clause A.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 the LIA-1 requirements.

A complete binding for LIA-1 will include a binding for IEC 559. Such a joint LIA-1 / IEC 559 binding should be developed as a single binding standard. To avoid conflict with ongoing development, only the LIA-1 specific portions of such a binding are presented in this annex.

E.1 General comments

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.

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 minimize 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 data type, but nearly all computational languages permit expressions that contain multiple operations involving operands of mixed types. The rules of the language specify how the operations and operands in an expression are mapped into the primitive operations described by LIA-1. In principle, the mapping could be completely specified in the language standard. However, the translator often has the freedom to depart from this precise specification: to reorder computations, widen data types, short-circuit evaluations, and perform other optimizations 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 A.5.2.14.)

Most language standards do not constrain the accuracy of floating point operations, or specify the subsequent behavior after a serious arithmetic violation occurs. We suggest that each language standard committee require that the arithmetic operations provided in the language satisfy the 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 `pragmas` 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.2 Ada

The programming language Ada is defined by ISO/IEC 8652:1986, *Information technology – Programming languages – Ada* [6].

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 a package named LIA.

The Ada data type BOOLEAN corresponds to the LIA-1 data type *Boolean*.

Every implementation of Ada has at least one integer data type, and at least one floating point data type. The notations *INT* and *FLT* are used to stand for the names of one of these data types in what follows.

The parameters for an integer data type can be accessed by the following syntax:

maxint	*INT*'LAST
minint	*INT*'FIRST

The parameter *bounded* is always **true**, and need not be provided. The parameter *modulo* is always **false**, and need not be provided.

The parameters for a floating point data type can be accessed by the following syntax:

r	*FLT*'MACHINE_RADIX	
p	*FLT*'MACHINE_MANTISSA	
emax	*FLT*'MACHINE_EMAX	
emin	*FLT*'MACHINE_EMIN	
denorm	*FLT*'DENORM	†
iec_559	*FLT*'IEC_559	†

The derived constants for the floating point data type can be accessed by the following syntax:

fmax	*FLT*'LAST	
$fmin_N$	*FLT*'MIN_NORM	†
fmin	*FLT*'MIN	†
epsilon	*FLT*'EPSILON	†
rnd_error	*FLT*'RND_ERROR	†
rnd_style	*FLT*'RND_STYLE	†

The value returned by the function *FLT*'RND_STYLE are from the enumeration type RND_STYLES. Each enumeration literal should correspond as follows to an LIA-1 rounding style value:

nearest	NEAREST	†
truncate	TRUNCATE	†
other	OTHER	†

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

add_I	`x + y`
sub_I	`x - y`
mul_I	`x * y`
div_I^f	no binding
div_I^t	`x / y`
rem_I^f	`x mod y`
rem_I^t	`x rem y`
mod_I^a	`x mod y`
mod_I^p	no binding

$sign_I$	`SIGNUM(`x`)`	†
neg_I	`-` x	
abs_I	`abs` x	
eq_I	x `=` y	
neq_I	x `/=` y	
lss_I	x `<` y	
leq_I	x `<=` y	
gtr_I	x `>` y	
geq_I	x `>=` y	

where x and y are expressions of type *INT*.

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

add_F	x `+` y	
sub_F	x `-` y	
mul_F	x `*` y	
div_F	x `/` y	
neg_F	`-` x	
abs_F	`abs` x	
$sign_F$	`SIGNUM(`x`)`	†
$exponent_F$	`EXPONENT(`x`)`	†
$fraction_F$	`FRACTION(`x`)`	†
$scale_F$	`SCALE(`x, n`)`	†
$succ_F$	`SUCCESSOR(`x`)`	†
$pred_F$	`PREDECESSOR(`x`)`	†
ulp_F	`UNIT_LAST_PLACE(`x`)`	†
$trunc_F$	`TRUNCATE_TO(`x, n`)`	†
$round_F$	`ROUND_TO(`x, n`)`	†
$intpart_F$	`INT_PART(`x`)`	†
$fractpart_F$	`FRACT_PART(`x`)`	†
eq_F	x `=` y	
neq_F	x `/=` y	
lss_F	x `<` y	
leq_F	x `<=` y	
gtr_F	x `>` y	
geq_F	x `>=` y	

where x and y are expressions of type *FLT* and n is of type *INT*.

Type conversions in Ada are always explicit and use the destination type name as the name of the conversion function. Hence:

$cvt_{I \rightarrow F}$, $cvt_{F' \rightarrow F}$	*FLT*`(`x`)`
$cvt_{F \rightarrow I}$, $cvt_{I' \rightarrow I}$	*INT*`(`x`)`

where x is an expression of the appropriate type. An implementation that wishes to conform to LIA-1 must use a round to nearest style for all conversions to floating point.

Ada defines its own method of exception handling based on alteration of control flow. Notification is accomplished by raising the exception `CONSTRAINT_ERROR`. An implementation that wishes to conform to LIA-1 must provide a default exception handler which terminates the program if no handler for the exception has been supplied by the programmer.

In addition, an implementation that wishes to conform to LIA-1 shall provide the alternative of notification through termination with a message as described in 6.1.3.

NOTE – A more comprehensive discussion of the relationship between LIA-1 and the Ada language can be found in [35]. In particular, this covers the relationship between the package `LIA` and packages for the elementary functions [16] and for primitive functions [17] being proposed to ISO.

E.3 BASIC

The programming language BASIC is defined by ISO/IEC 10279:1991, *Information technology – Programming languages – Full BASIC* [12].

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.

There is no user accessible BASIC data type corresponding to the LIA-1 data type *Boolean*. Any of the LIA-1 operations that return a *Boolean* value correspond to "relational-expressions" in BASIC which appear within control structures.

BASIC has one primitive computational data type, `numeric`. The model presented by the BASIC language is that of a real number with decimal radix and a specified (minimum) number of decimal digits. Numeric data is not declared directly, but any special characteristics are inferred from how they are used and from any `OPTIONS` that are in force.

The BASIC statement `OPTION ARITHMETIC NATIVE` ties the `numeric` type more closely to the underlying implementation. The precision and type of `NATIVE` numeric data is implementation dependent.

Since the BASIC numeric data type does not match the integer type required by LIA-1, an implementation is not required to supply any of the LIA-1 parameters or operations for integer data types.

The BASIC numeric type is used for the integer valued type "J" introduced in 5.2.2.

The parameters for the `numeric` data type can be accessed by the following syntax:

r	`RADIX`	†
p	`PLACES`	†
$emax$	`MAXEXP`	†
$emin$	`MINEXP`	†
$denorm$	`DENORM`	†
iec_559	`IEC_559`	†

The derived constants for the `numeric` data type can be accessed by the following syntax:

$fmax$	`MAXNUM`	†
$fmin_N$	`FMINN`	†
$fmin$	`EPS(0)`	
$epsilon$	`EPSILON`	†
rnd_error	`RNDERROR`	†
rnd_style	`RNDSTYLE`	†

The allowed values of the parameter *rnd_style* are `numeric` and can be accessed by the following syntax:

nearest	`NEAREST`	†
truncate	`TRUNCATE`	†
other	`OTHER`	†

The LIA-1 floating point operations are listed below, along with the syntax used to invoke them:

add_F	`x + y`	
sub_F	`x - y`	
mul_F	`x * y`	
div_F	`x / y`	
neg_F	`- x`	
abs_F	`ABS(x)`	
$sign_F$	`SGN(x)`	
$exponent_F$	`EXPON(x)`	†
$fraction_F$	`FRACTION(x)`	†
$scale_F$	`SCALE(x, n)`	†
$succ_F$	`SUCC(x)`	†
$pred_F$	`PRED(x)`	†
ulp_F	`ULP(x)`	†
$trunc_F$	`TRUNCTO(x, n)`	†
$round_F$	`ROUNDTO(x, n)`	†
$intpart_F$	`IP(x)`	
$fractpart_F$	`FP(x)`	
eq_F	`x = y`	
neq_F	`x >< y` or `x <> y`	
lss_F	`x < y`	
leq_F	`x =< y` or `x <= y`	
gtr_F	`x > y`	
geq_F	`x => y` or `x >= y`	

where x and y are `numeric` expressions and n is integral.

NOTES

1 The BASIC `EPS(x)` function differs from ulp_F, in that $ulp_F(x)$ raises a notification when $x = 0$ and can underflow when x is very close to zero, but `EPS(x)` returns *fmin* for these values of x.

2 The BASIC functions `ROUND` and `TRUNCATE` differ from $round_F$ and $trunc_F$ in that n refers to the number of digits retained after the decimal point, rather than the total number of digits retained.

The notification method required by BASIC is through alteration of control flow. Notification is accomplished through the exception handling facilities required by BASIC. An implementation that wishes to conform to LIA-1 must create a BASIC exception in any case where an LIA-1 operation would return an exceptional value. BASIC requires that a program substitute zero and continue execution when underflow occurs and no programmer-specified recovery procedure has been provided. This does not meet the notification requirements of LIA-1, but is explicitly permitted by this binding standard.

The exception codes returned by the function `EXTYPE` include refinements of the LIA-1 exceptional values along with values characterizing non-numeric exceptions as well. The following lists the BASIC exception code along with a description and corresponding LIA-1 exceptional value.

Code	*Description*	*LIA-1 value*	
1002	Numeric expression overflow	**floating_overflow**	
1003	Numeric supplied function overflow	**floating_overflow**	
1502	Numeric expression underflow	**underflow**	
1503	Numeric supplied function underflow	**underflow**	
3001	Division by zero	**undefined**	
3010	Attempt to evaluate `EXPON(0)`	**undefined**	†
3011	Attempt to evaluate `ULP(0)`	**undefined**	†

In addition, an implementation that wishes to conform to LIA-1 shall provide the alternative of notification through termination with a message as described in 6.1.3.

E.4 C

The programming language C is defined by ISO/IEC 9899:1990, *Information technology – Programming languages – C* [9].

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 `<lia.h>`.

Integer valued parameters and derived constants can be used in preprocessor expressions.

The LIA-1 data type *Boolean* is implemented in the C data type `int` (1 = **true** and 0 = **false**).

Every implementation of C has integral types `int`, `long int`, `unsigned int`, and `unsigned long int` which conform to LIA-1.

NOTE 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.

C has three floating point types that conform to this part of ISO/IEC 10967: `float`, `double`, and `long double`.

The parameters for an integer data type can be accessed by the following syntax:

maxint	`INT_MAX`	`LONG_MAX`	`UINT_MAX`	`ULONG_MAX`	
minint	`INT_MIN`	`LONG_MIN`			
modulo	`INT_MODULO`	`LONG_MODULO`			†

The parameter *bounded* is always **true**, and need not be provided. The parameter *minint* is always 0 for the unsigned types, and need not be provided for those types. The parameter *modulo* is always **true** for the unsigned types, and need not be provided for those types.

The parameters for a floating point data type can be accessed by the following syntax:

r	`FLT_RADIX`			
p	`FLT_MANT_DIG`	`DBL_MANT_DIG`	`LDBL_MANT_DIG`	
emax	`FLT_MAX_EXP`	`DBL_MAX_EXP`	`LDBL_MAX_EXP`	
emin	`FLT_MIN_EXP`	`DBL_MIN_EXP`	`LDBL_MIN_EXP`	
denorm	`FLT_DENORM`	`DBL_DENORM`	`LDBL_DENORM`	†
iec_559	`FLT_IEC_559`	`DBL_IEC_559`	`LDBL_IEC_559`	†

The *_DENORM macros and *_IEC_559 macros represent booleans and have values 1 or 0.

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 derived constants for the floating point data type can be accessed by the following syntax:

fmax	`FLT_MAX`	`DBL_MAX`	`LDBL_MAX`	
$fmin_N$	`FLT_MIN`	`DBL_MIN`	`LDBL_MIN`	
fmin	`FLT_TRUE_MIN`	`DBL_TRUE_MIN`	`LDBL_TRUE_MIN`	†
epsilon	`FLT_EPSILON`	`DBL_EPSILON`	`LDBL_EPSILON`	
rnd_error	`FLT_RND_ERR`	`DBL_RND_ERR`	`LDBL_RND_ERR`	†
rnd_style	`FLT_ROUNDS`			

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.

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

NOTE 2 – The definition of `FLT_ROUNDS` has been extended to cover the rounding style used in all LIA-1 operations, not just addition.

The 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:

add_I	x `+` y		
sub_I	x `-` y		
mul_I	x `*` y		
div_I	x `/` y		
rem_I	x `%` y		
mod_I^a	`modulo(`x, y`)`	`lmodulo(`x, y`)`	†
mod_I^p	no binding		
neg_I	`-` x		
abs_I	`abs(`x`)`	`labs(`x`)`	
$sign_I$	`sgn(`x`)`	`lsgn(`x`)`	†
eq_I	x `==` y		
neq_I	x `!=` y		
lss_I	x `<` y		
leq_I	x `<=` y		
gtr_I	x `>` y		
geq_I	x `>=` y		

where x and y are expressions of the same integer type.

The C standard permits div_I and rem_I (`/` and `%`) to be implemented using either round toward minus infinity (div_I^f/rem_I^f) or toward zero (div_I^t/rem_I^t). An implementation that wishes to conform to LIA-1 must choose the same rounding for both and document the choice.

The 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:

add_F	x `+` y
sub_F	x `-` y
mul_F	x `*` y

div_F	`x / y`			
neg_F	`- x`			
abs_F	`fabsf(x)`†	`fabs(x)`	`fabsl(x)`†	
$sign_F$	`fsgnf(x)`	`fsgn(x)`	`fsgnl(x)`	†
$exponent_F$	`exponf(x)`	`expon(x)`	`exponl(x)`	†
$fraction_F$	`fractf(x)`	`fract(x)`	`fractl(x)`	†
$scale_F$	`scalef(x,n)`	`scale(x,n)`	`scalel(x,n)`	†
$succ_F$	`succf(x)`	`succ(x)`	`succl(x)`	†
$pred_F$	`predf(x)`	`pred(x)`	`predl(x)`	†
ulp_F	`ulpf(x)`	`ulp(x)`	`ulpl(x)`	†
$trunc_F$	`trunctof(x,n)`	`truncto(x,n)`	`trunctol(x,n)`	†
$round_F$	`roundtof(x,n)`	`roundto(x,n)`	`roundtol(x,n)`	†
$intpart_F$	`intprtf(x)`	`intprt(x)`	`intprtl(x)`	†
$fractpart_F$	`frcprtf(x)`	`frcprt(x)`	`frcprtl(x)`	†
eq_F	`x == y`			
neq_F	`x != y`			
lss_F	`x < y`			
leq_F	`x <= y`			
gtr_F	`x > y`			
geq_F	`x >= y`			

where x and y are expressions of the same floating point type, and n is of type `int`.

NOTES

3 $Scale_F$ can be computed using the `ldexp` library function, only if `FLT_RADIX` = 2.

4 The standard C function `frexp` differs from $exponent_F$ in that no notification is raised when the argument is 0.

An implementation that wishes to conform to LIA-1 must provide the LIA-1 operations in all floating point precisions supported.

C provides the required type conversion operations with explicit cast operators:

$cvt_{F\to I}$, $cvt_{I'\to I}$	`(int) x`, `(long) x`, `(unsigned int) x`, `(unsigned long) x`
$cvt_{I\to F}$, $cvt_{F'\to F}$	`(float) x`, `(double) x`, `(long double) x`

The C standard requires that float to integer conversions round toward zero. An implementation that wishes to conform to LIA-1 must use round to nearest for conversions to a floating point type.

An implementation that wishes to conform to LIA-1 must provide recording of indicators as one method of notification. (See 6.1.2.) The data type *Ind* is identified with the data type `unsigned int`. The values representing individual indicators should be distinct non-negative powers of two and can be accessed by the following syntax:

integer_overflow	`INT_OVERFLOW`	†
floating_overflow	`FLT_OVERFLOW`	†
underflow	`UNDERFLOW`	†
undefined	`UNDEFINED`	†

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

{**floating_overflow**, **underflow**, **integer_overflow**} would be denoted by the expression

`FLT_OVERFLOW | UNDERFLOW | INT_OVERFLOW`

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

set_indicators	`set_indicators(`i`)`	†
clear_indicators	`clear_indicators(`i`)`	†
test_indicators	`test_indicators(`i`)`	†
current_indicators	`current_indicators()`	†

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

In addition, an implementation that wishes to conform to LIA-1 shall provide the alternative of notification through termination with a message as described in 6.1.3.

E.5 Common Lisp

The programming language Common Lisp is under development by ANSI X3J13 [21]. The standard will be based on the definition contained in *Common Lisp: the Language* [32].

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 data type that corresponds to the LIA-1 data type *Boolean*. Rather, `NIL` corresponds to **false** and `T` corresponds to **true**.

Every implementation of Common Lisp has one unbounded integer data type. 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* and *modulo* are always **false**, and the parameters *bounded*, *modulo*, *maxint*, and *minint* need not be provided.

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

r	`(float-radix `x`)`		
p	`(float-digits `x`)`		
emax	`maxexp-short-float`,	`maxexp-single-float`,	
	`maxexp-double-float`,	`maxexp-long-float`	
emin	`minexp-short-float`,	`minexp-single-float`,	
	`minexp-double-float`,	`minexp-long-float`	
denorm	`denorm-short-float`,	`denorm-single-float`,	†
	`denorm-double-float`,	`denorm-long-float`	†
iec_559	`iec-559-short-float`,	`iec-559-single-float`,	†
	`iec-559-double-float`,	`iec-559-long-float`	†

where x is of type `short-float`, `single-float`, `double-float` or `long-float`.

The derived constants for the floating point data type can be accessed by the following syntax:

fmax	`most-positive-short-float`	
	`most-positive-single-float`	
	`most-positive-double-float`	
	`most-positive-long-float`	
$fmin_N$	`least-positive-normalized-short-float`	
	`least-positive-normalized-single-float`	
	`least-positive-normalized-double-float`	
	`least-positive-normalized-long-float`	
fmin	`least-positive-short-float`	
	`least-positive-single-float`	
	`least-positive-double-float`	
	`least-positive-long-float`	
epsilon	`short-float-epsilon`	
	`single-float-epsilon`	
	`double-float-epsilon`	
	`long-float-epsilon`	
rnd_error	`short-float-rounding-error`	†
	`single-float-rounding-error`	†
	`double-float-rounding-error`	†
	`long-float-rounding-error`	†
rnd_style	`rounding`	†

NOTE 1 – 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` is an object of type `rounding-style`. The subtypes of `rounding-style` have the following names corresponding to the LIA-1 *rnd_style* values:

nearest	`nearest`	†
truncate	`truncate`	†
other	`other`	†

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

add_I	`(+` x y`)`
sub_I	`(-` x y`)`
mul_I	`(*` x y`)`
div_I^f	`(floor` x y`)`
div_I^t	`(truncate` x y`)`
rem_I^f	`(mod` x y`)`
rem_I^t	`(rem` x y`)`
mod_I^a	`(mod` x y`)`
mod_I^p	no binding
neg_I	`(-` x`)`
abs_I	`(abs` x`)`
$sign_I$	`(signum` x`)`

eq_I	`(= x y)`	
neq_I	`(/= x y)`	
lss_I	`(< x y)`	
leq_I	`(<= x y)`	
gtr_I	`(> x y)`	
geq_I	`(>= x y)`	

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

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

add_F	`(+ x y)`	
sub_F	`(- x y)`	
mul_F	`(* x y)`	
div_F	`(/ x y)`	
neg_F	`(- x)`	
abs_F	`(abs x)`	
$sign_F$	`(signum x)`	
$exponent_F$	`(float-exponent x)`	†
$fraction_F$	`(decode-float x)`	
$scale_F$	`(scale-float x n)`	
$succ_F$	`(succ x)`	†
$pred_F$	`(pred x)`	†
ulp_F	`(ulp x)`	†
$trunc_F$	`(truncate-float x n)`	†
$round_F$	`(round-float x n)`	†

`(multiple-value-bind (int fract) (ftruncate x))`

$intpart_F$	`int`
$fractpart_F$	`fract`

eq_F	`(= x y)`
neq_F	`(/= x y)`
lss_F	`(< x y)`
leq_F	`(<= x y)`
gtr_F	`(> x y)`
geq_F	`(>= x y)`

where x and y are data objects of the same floating point type, and n is of integer type.

NOTES

2 Only `signum` returns 0 when applied to 0 as is required of $sign_F$ by LIA-1. Neither `float-sign` nor `decode-float` do.

3 The function `float-exponent` differs from `decode-float` in that it generates a notification if the argument is zero, while `decode-float` does not.

Type conversions in Common Lisp are explicit through conversion functions. The programmer may choose the rounding in $cvt_{F \rightarrow I}$. Conversions to floating point type yield a result of default `float` type, unless there is a second operand y, in which case the result is of the same type as y.

$cvt_{I \rightarrow F}$, $cvt_{F' \rightarrow F}$	`(float x)`, `(float x y)`	
$cvt_{F \rightarrow I}$	`(floor x)`	(round toward minus infinity)
	`(ceiling x)`	(round toward positive infinity)
	`(truncate x)`	(round toward zero)

`(round x)`	(round to nearest)

An implementation of Common Lisp that wishes to conform to LIA-1 must use round to nearest when converting to a floating point type.

Common Lisp defines its own method of exception handling based on alteration of control flow. Notification is accomplished by signalling a condition of the appropriate type. The LIA-1 exceptional values are represented by the following Common Lisp condition types:

integer_overflow	(not needed, the integer type is unbounded)
floating_overflow	`floating-point-overflow`
underflow	`floating-point-underflow`
undefined	`division-by-zero`, or `arithmetic-error`

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.

In addition, an implementation that wishes to conform to LIA-1 shall provide the alternative of notification through termination with a message as described in 6.1.3.

E.6 Fortran

The programming language Fortran is defined by ISO/IEC 1539:1991, *Information technology – Programming languages – FORTRAN* [3].

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 a module named `LIA`.

The Fortran data type `LOGICAL` corresponds to the LIA-1 data type *Boolean*.

Every implementation of Fortran has one integer data type, denoted as `INTEGER`, and two floating point data types denoted as `REAL` (single precision) and `DOUBLE PRECISION`. (In Fortran terminology, `REAL` and `DOUBLE PRECISION` denote the same "type" (`REAL`) but have different `KIND` parameters. The rest of this clause will use the Fortran terminology.)

An implementation is permitted to offer additional `INTEGER` types with a different range and additional `REAL` types with different precision or range, parameterized with the `KIND` parameter.

The parameters for `INTEGER` types can be accessed by the following syntax:

maxint	`HUGE(x)`	
minint	`MININT(x)`	†
modulo	`MODINT(x)`	†

where x is an expression of the appropriate `INTEGER` type.

The parameter *bounded* is always **true**, and need not be provided.

The parameters for the `REAL` types can be accessed by the following syntax:

r	`RADIX(`x`)`	
p	`DIGITS(`x`)`	
$emax$	`MAXEXPONENT(`x`)`	
$emin$	`MINEXPONENT(`x`)`	
$denorm$	`DENORM(`x`)`	†
iec_559	`IEC_559(`x`)`	†

where x is an expression of the appropriate `REAL` type.

The derived constants for `REAL` data types can be accessed by the following syntax:

$fmax$	`HUGE(`x`)`	
$fmin_N$	`TINY(`x`)`	
$fmin$	`TINIEST(`x`)`	†
$epsilon$	`EPSILON(`x`)`	
rnd_error	`RND_ERROR(`x`)`	†
rnd_style	`RND_STYLE(`x`)`	†

where x is an expression of the appropriate `REAL` type.

The function `RND_STYLE` returns one of the following values of type `CHARACTER*8`:

$nearest$	`'nearest'`
$truncate$	`'truncate'`
$other$	`'other'`

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

add_I	x `+` y	
sub_I	x `-` y	
mul_I	x `*` y	
div_I^f	no binding	
div_I^t	x `/` y	
rem_I^f	`MODULO(`x, y`)`	
rem_I^t	`MOD(`x, y`)`	
mod_I^a	`MODULO(`x, y`)`	
mod_I^p	no binding	
neg_I	`-` x	
abs_I	`ABS(`x`)`	
$sign_I$	`SIGNUM(`x`)`	†
eq_I	x `.EQ.` y or x `==` y	
neq_I	x `.NE.` y or x `/=` y	
lss_I	x `.LT.` y or x `<` y	
leq_I	x `.LE.` y or x `<=` y	
gtr_I	x `.GT.` y or x `>` y	
geq_I	x `.GE.` y or x `>=` y	

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

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

add_F	x `+` y
sub_F	x `-` y
mul_F	x `*` y
div_F	x `/` y

neg_F	`- x`	
abs_F	`ABS(x)`	
$sign_F$	`SIGNUM(x)`	†
$exponent_F$	`EXPON(x)`	†
$fraction_F$	`FRACTION(x)`	
$scale_F$	`SCALE(x, n)`	
$succ_F$	`NEAREST(x, 1.0)`	
$pred_F$	`NEAREST(x, -1.0)`	
ulp_F	`ULP(x)`	†
$trunc_F$	`TRUNCTO(x, n)`	†
$round_F$	`ROUNDTO(x, n)`	†
$intpart_F$	`AINT(x)`	
$fractpart_F$	`x - AINT(x)`	
eq_F	`x .EQ. y` or `x == y`	
neq_F	`x .NE. y` or `x /= y`	
lss_F	`x .LT. y` or `x < y`	
leq_F	`x .LE. y` or `x <= y`	
gtr_F	`x .GT. y` or `x > y`	
geq_F	`x .GE. y` or `x >= y`	

where x and y are reals of the same KIND, and n is of type INTEGER.

NOTES

1 The Fortran function SIGN(1,x) is different from $sign_F$ because it returns 1 instead of 0 for $x = 0$.

2 The intrinsic function EXPONENT(x) differs from $exponent_F$ at $x = 0$ where it returns 0 instead of a notification.

3 The Fortran function SPACING differs from ulp_F in that it does not raise a notification on either underflow or an input of 0.

An implementation that wishes to conform to LIA-1 must provide the LIA-1 operations and parameters for any additional INTEGER or REAL types provided.

Type conversions in Fortran are either explicit through conversion functions, or implicit through expressions or assignment statements. For explicit conversions, an optional *kind* argument indicates the KIND of the destination. Conversions to a REAL type are required by LIA-1 to use round to nearest. The programmer may select the rounding of the REAL to INTEGER conversion by using one of the explicit conversion functions invoked with the following syntax:

$cvt_{I \rightarrow F}$	REAL(x, *kind*), DBLE(x)	(round to nearest)
$cvt_{F' \rightarrow F}$	REAL(x, *kind*), DBLE(x)	(round to nearest)
$cvt_{F \rightarrow I}$	INT(x, *kind*)	(round toward zero)
	NINT(x, *kind*)	(round to nearest)
	CEILING(x)	(round toward plus infinity)
	FLOOR(x)	(round toward minus infinity)
$cvt_{I' \rightarrow I}$	INT(x, *kind*)	

An implementation that wishes to conform to LIA-1 must provide recording of indicators as one method of notification. (See 6.1.2.) The data type *Ind* is identified with the data type INTEGER. The values representing individual indicators are distinct non-negative powers of two and can be accessed by the following syntax:

integer_overflow	`INT_OVERFLOW`	†
floating_overflow	`FLT_OVERFLOW`	†
underflow	`UNDERFLOW`	†
undefined	`UNDEFINED`	†

The empty set can be denoted by 0. Other indicator subsets can be named by adding together individual indicators. For example, the indicator subset

{**floating_overflow**, **underflow**, **integer_overflow**}

would be denoted by the expression

`FLT_OVERFLOW` + `UNDERFLOW` + `INT_OVERFLOW`

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

set_indicators	`call SET_INDICATORS(`i`)`	†
clear_indicators	`call CLR_INDICATORS(`i`)`	†
test_indicators	`TEST_INDICATORS(`i`)`	†
current_indicators	`CURR_INDICATORS()`	†

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

In addition, an implementation that wishes to conform to LIA-1 shall provide the alternative of notification through termination with a message as described in 6.1.3.

NOTE 4 – Implementations of Fortran 77 were not required to support names longer than six characters. However, they may still wish to provide the parameters and functions of LIA-1. To achieve consistency, it is suggested that the names given here be used after truncating to the first six alphabetic characters.

E.7 Modula-2

The International Standard [13] defining the programming language Modula-2 is under development by ISO/IEC JTC1/SC22/WG13. The Standard will be based on the definition in Wirth [37]. A binding for LIA-1 has been included in an early draft [18]. Therefore a suggested binding is not included here.

E.8 Pascal and Extended Pascal

The programming language Extended Pascal is defined in ISO/IEC 10206:1991, *Information technology – Programming languages – Extended Pascal* [11]. The programming language ISO Pascal is defined by ISO/IEC 7185:1990, *Information technology – Programming languages – Pascal* [5]. The programming language ANSI/IEEE Pascal is defined in ANSI/IEEE 770/X3.97-1983 [19].

ANSI/IEEE Pascal and ISO Pascal are so close that for the purposes of this annex they are treated as a single language, "Pascal." Where Extended Pascal differs from Pascal as regards this annex, the differences are noted.

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 LIA-1 data type *Boolean* is implemented as the Pascal data type `boolean`.

Pascal and Extended Pascal have an `integer` data type and a `real` data type.

The parameters for the `integer` data type can be accessed by the following syntax:

maxint	`maxint`	
minint	`minint`	†
modulo	`modulo`	†

The parameter *bounded* is always **true**, and need not be provided.

The parameters for the `real` data type can be accessed by the following syntax:

r	`radix`	†
p	`places`	†
emax	`maxexp`	†
emin	`minexp`	†
denorm	`denorm`	†
iec_559	`iec559`	†

The derived constants for the `real` data type can be accessed by the following syntax:

fmax	`maxreal`	
$fmin_N$	`minrealn`	†
fmin	`minreal`	
epsilon	`epsreal`	
rnd_error	`rnderror`	†
rnd_style	`rndstyle`	†

NOTE – Extended Pascal requires the constant-identifiers **maxreal**, **minreal**, and **epsreal**, but Pascal does not.

The allowed values of the parameter `rndstyle` are from the enumerated data type

```
Rndstyles = (nearest, truncate, other);
```

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

add_I	x `+` y	
sub_I	x `-` y	
mul_I	x `*` y	
div_I^f	no binding	
div_I^t	x `div` y	
rem_I^f	no binding	
rem_I^t	`rem(`x, y`)`	†
mod_I^a	no binding	
mod_I^p	x `mod` y	
neg_I	`-` x	
abs_I	`abs(`x`)`	
$sign_I$	`signi(`x`)`	†
eq_I	x `=` y	
neq_I	x `<>` y	

lss_I	`x < y`
leq_I	`x <= y`
gtr_I	`x > y`
geq_I	`x >= y`

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

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

add_F	`x + y`	
sub_F	`x - y`	
mul_F	`x * y`	
div_F	`x / y`	
neg_F	`- x`	
abs_F	`abs(x)`	
$sign_F$	`signf(x)`	†
$exponent_F$	`expon(x)`	†
$fraction_F$	`fraction(x)`	†
$scale_F$	`scale(x,n)`	†
$succ_F$	`succf(x)`	†
$pred_F$	`predf(x)`	†
ulp_F	`ulp(x)`	†
$trunc_F$	`truncto(x,n)`	†
$round_F$	`roundto(x,n)`	†
$intpart_F$	`intpart(x)`	†
$fractpart_F$	`fractpart(x)`	†
eq_F	`x = y`	
neq_F	`x <> y`	
lss_F	`x < y`	
leq_F	`x <= y`	
gtr_F	`x > y`	
geq_F	`x >= y`	

where x and y are expressions of type `real` and n is of type `integer`.

Pascal and Extended Pascal provide explicit type conversion functions from the `real` type to the `integer` type. The programmer can select between round toward zero and round to nearest.

$cvt_{F \rightarrow I}$	`trunc(x)`	(round toward zero)
	`round(x)`	(round to nearest)

The conversion from `integer` type to `real` type is done implicitly with assignment and should round to nearest.

$cvt_{I \rightarrow F}$	`x := n`	(round to nearest)

The error cases in LIA-1 which require a notification are either formal *error* cases in Pascal and Extended Pascal, or else violate the requirement for arithmetic to approximate the true mathematical operations. An implementation of Pascal and Extended Pascal that wishes to conform to LIA-1 must detect errors and provide recording of indicators as one method of notification. (See 6.1.2.) The data type *Ind* is identified with the data type `Indicators`:

```
Indicator = (integeroverflow, realoverflow, underflow, undefined);

Indicators = set of indicator;
```

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

set_indicators	`setindicators(`i`)`	†
clear_indicators	`clearindicators(`i`)`	†
test_indicators	`testindicators(`i`)`	†
current_indicators	`currentindicators`	†

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

In addition, an implementation that wishes to conform to LIA-1 shall provide the alternative of notification through termination with a message as described in 6.1.3.

E.9 PL/I

The programming language General Purpose PL/I is defined by ISO/IEC 6522:1992, *Information technology – Programming languages – PL/I general purpose subset* [4].

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 LIA-1 data type *Boolean* is implemented in the PL/I data type `BIT(1)` (1 = **true** and 0 = **false**).

An implementation of PL/I provides at least one integer data type, and at least one floating point data type. The attribute `FIXED(`n`,0)` selects a signed integer type with at least n (decimal or binary) digits of storage. The attribute `FLOAT(`k`)` selects a floating point type with at least k (decimal or binary) digits of precision.

The parameters for an integer data type can be accessed by the following syntax:

maxint	`maxint(`x`)`	†
minint	`minint(`x`)`	†
bounded	`bounded(`x`)`	†

where x is an expression of the appropriate integer type.

The parameter *modulo* is always **false**, and need not be provided.

The parameters for a floating point data type can be accessed by the following syntax:

r	`radix(`x`)`	
p	`places(`x`)`	†
emax	`maxexp(`x`)`	†
emin	`minexp(`x`)`	†
denorm	`denorm(`x`)`	†
iec_559	`iec_559(`x`)`	†

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

The derived constants for a floating point type can be accessed by the following syntax:

$fmax$	`maxval(x)`	
$fmin_N$	`minval(x)`	
$fmin$	`trueminval(x)`	†
$epsilon$	`epsilon(x)`	
rnd_error	`rnderror(x)`	†
rnd_style	`rndstyle(x)`	†

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

The allowed values of the parameter `RNDSTYLE` are of type `BINARY FIXED(2,0)` and can be accessed by the following syntax:

$nearest$	`NEAREST`	†
$truncate$	`TRUNCATE`	†
$other$	`OTHER`	†

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

add_I	`x + y`	
sub_I	`x - y`	
mul_I	`x * y`	
div_I^f	no binding	
div_I^t	`x / y`	
rem_I^f	`mod(x,y)`	
rem_I^t	`rem(x,y)`	†
mod_I^a	`mod(x,y)`	
mod_I^p	no binding	
neg_I	`- x`	
abs_I	`abs(x)`	
$sign_I$	`sign(x)`	
eq_I	`x = y`	
neq_I	`x ¬= y`	
lss_I	`x < y`	
leq_I	`x <= y` or `x ¬> y`	
gtr_I	`x > y`	
geq_I	`x >= y` or `x ¬< y`	

where x and y are expressions of the same `FIXED` type.

The floating point operations are listed below, along with the syntax used to invoke them.

add_F	`x + y`	
sub_F	`x - y`	
mul_F	`x * y`	
div_F	`x / y`	
neg_F	`- x`	
abs_F	`abs(x)`	
$sign_F$	`sign(x)`	
$exponent_F$	`exponent(x)`	
$fraction_F$	`fraction(x)`	†
$scale_F$	`scale(x,n)`	†
$succ_F$	`succ(x)`	†
$pred_F$	`pred(x)`	†
ulp_F	`ulp(x)`	†

$trunc_F$	`truncto(`x, n`)`	†
$round_F$	`round(`x, n`)`	
$intpart_F$	`trunc(`x`)`	
$fractpart_F$	x `- trunc(`x`)`	
eq_F	x `==` y	
neq_F	x `¬=` y	
lss_F	x `<` y	
leq_F	x `<=` y or x `¬>` y	
gtr_F	x `>` y	
geq_F	x `>=` y or x `¬<` y	

where x and y are expressions of the same `FLOAT` type, and n is an integer.

Type conversions in PL/I are either explicit through conversion functions, or implicit through assignment statements. The explicit conversion operation to a target type `FIXED(`n`, 0)` is invoked with the following syntax:

$cvt_{I' \to I}$, $cvt_{F \to I}$	`FIXED(`$x, n,$ `0)`

The explicit conversion operation to a target type `FLOAT(`k`)` is invoked with the following syntax:

$cvt_{I \to F}$, $cvt_{F' \to F}$	`FLOAT(`x, k`)`

An implementation that wishes to conform to LIA-1 must use round to nearest for the conversions to floating point types.

The notification method required by PL/I is through alteration of control flow. A condition is raised which leads to invocation of an interrupt operation through an `ON`-unit. The conditions raised by PL/I include some refinements of the exceptional values returned by LIA-1. The following lists the PL/I conditions along with the corresponding LIA-1 exceptional values.

PL/I condition	*LIA-1 value*	
`FIXEDOVERFLOW`	**integer_overflow**	
`SIZE`	**integer_overflow**	
`OVERFLOW`	**floating_overflow**	
`UNDERFLOW`	**underflow**	
`ZERODIVIDE`	**undefined**	
`UNDEFINED`	**undefined**	†

An implementation that wishes to conform to LIA-1 must raise the appropriate condition whenever an LIA-1 exceptional value would result. The `SIZE` condition is raised only in the case of overflow on conversion to an integer type; otherwise `FIXEDOVERFLOW` is raised. The condition `ZERODIVIDE` is raised in the case of division by zero. All other cases in which the LIA-1 exceptional value **undefined** is returned shall raise the condition `UNDEFINED`. An implementation that wishes to conform to LIA-1 must provide a default `ON`-unit which terminates the program with a message, if no `ON`-unit for the condition has been supplied by the programmer.

In addition, an implementation that wishes to conform to LIA-1 shall provide the alternative of notification through termination with a message as described in 6.1.3.

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 standard.

This example concentrates on conformity with LIA-1. Details concerning conformity to IEEE 754, while important, have been omitted. 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 IEC 559:1989, *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 clause E.6 of LIA-1.

Only implementation dependent information is directly provided here. The information in the suggested language binding standard for Fortran (see clause E.6 of LIA-1) 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 PRECISION`.

F.2 Integer parameters

The following table gives the parameters for `INTEGER`, the names of the generic functions with which they can be accessed at run-time, and their values.

Parameters for `INTEGER`		
parameter	*function*	*value*
$maxint$	`HUGE(`x`)`	$2^{31}-1$
$minint$	`MININT(`x`)`	-2^{31}
$modulo$	`MODINT(`x`)`	**false**

where x is an expression of type `INTEGER`.

F.3 Floating point parameters

The following table gives the parameters for `REAL` and `DOUBLE PRECISION`, the names of the generic functions with which they can be accessed at run-time, and their values.

Parameters for Floating Point			
parameter	*function*	`REAL`	`DOUBLE`
r	`RADIX(`x`)`	2	2
p	`DIGITS(`x`)`	24	53
$emax$	`MAXEXPONENT(`x`)`	128	1024
$emin$	`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 generic functions with which they can be accessed at run-time, and the (approximate) values for `REAL` and `DOUBLE PRECISION`. The functions return exact values for the derived constants.

Derived constants			
constant	*function*	`REAL`	`DOUBLE`
$fmax$	`HUGE(`x`)`	3.402823466 e+38	1.7976931349 e+308
$fmin_N$	`TINY(`x`)`	1.175494351 e−38	2.2250738585 e−308
$fmin$	`TINIEST(`x`)`	1.401298464 e−45	4.9406564584 e−324
$epsilon$	`EPSILON(`x`)`	1.192092896 e−07	2.2204460493 e−016
rnd_error	`RND_ERROR(`x`)`	0.5	0.5
rnd_style	`RND_STYLE(`x`)`	`'nearest'`	`'nearest'`

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

F.4 Definitions

The approximate addition function is defined to be true addition:

$$add^*_F(x, y) = x + y$$

The rounding function rnd_F is one of the four rounding functions implied by IEEE 754-1985 (clause 4) only two of which conform to LIA-1. 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 ZERO
!LIA$  SELECT ROUND TO PLUS INFINITY    (does not conform to LIA-1)
!LIA$  SELECT ROUND TO MINUS INFINITY   (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 PRECISION`. 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 result function $result_F$ is defined to use the selected rounding function rnd_F. The choice between $rnd_F(x)$ and **underflow** for the denormalized range is made in accordance with clause 7.4 of IEEE 754-1985. In IEEE terms, this implementation chooses to detect tininess 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).

All computed numeric results are of type `INTEGER`, `REAL`, or `DOUBLE PRECISION`. There are no "extended precision" intermediate results, or hidden higher precision types. No automatic conversions to higher precision (e.g., `REAL` to `DOUBLE PRECISION`) are made unless required by the Fortran standard.

F.6 Notification

Notifications are raised under all circumstances specified by 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 full 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 (**integer_overflow**, **floating_overflow**, **underflow**, or **undefined**),

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 clauses E.6, E.4, and E.2, 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 559 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);      /* restore the default rounding */
    ... combine the differently rounded 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;            -- an arbitrary constant value

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 Fast versus reliable

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, uses the slow but reliable one.

```
call clr_indicators (FLT_OVERFLOW)
result = FAST_SOLUTION (input)
if (test_indicators (FLT_OVERFLOW)) then
    result = RELIABLE_SOLUTION (input)
end if
```

Demmel and Li discuss a number of similar algorithms in [25].

Code that explicitly manipulates indicators may find it necessary to save and restore them as well. See clause G.7.

G.5 High-precision multiply

In general, the exact product of two p-digit numbers requires about $2p$ 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 [31] 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;
```

G.6 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.7 Saving and restoring indicators

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 */
unsigned int saved_flags;

saved_flags = current_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.

Annex H
(informative)

Bibliography

This annex gives references to publications relevant to the annexes of this part of ISO/IEC 10967.

International Documents

[1] IEC 559:1989, *Binary floating-point arithmetic for microprocessor systems.* (Also: ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*)

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

[3] ISO/IEC 1539:1991, *Information technology – Programming languages – FORTRAN.* (Also: ANSI X3.198-1991, *American National Standard Programming Language Fortran*)

[4] ISO/IEC 6522:1992, *Information technology – Programming languages – PL/I general purpose subset.* (Endorsement of ANSI X3.74-1987, *American National Standard Programming Language General Purpose PL/I*)

[5] ISO/IEC 7185:1990, *Information technology – Programming languages – Pascal.*

[6] ISO/IEC 8652:1987, *Information technology – Programming languages – Ada.* (Endorsement of ANSI/MIL-STD-1815A-1983, *Reference Manual for the Ada Programming Language*)

[7] ISO/IEC 8825:1990, *Information Processing Systems – Open Systems Interconnection – Specification of Basic Encoding Rules for Abstract Syntax Notation One* (ASN.1).

[8] ISO/IEC 9001:1987, *Quality systems – Model for quality assurance in production and installation.*

[9] ISO/IEC 9899:1990, *Information technology – Programming languages – C.* (Also: ANSI X3.159-1989, *American National Standard for Information Systems – Programming Language – C*)

[10] ISO/IEC TR 10176:1991, *Information technology – Guidelines for the preparation of programming language standards.*

[11] ISO/IEC 10206:1991, *Information technology – Programming languages – Extended Pascal.* (Also: ANSI/IEEE 770/X3.160-1989, *Standard for the programming language Extended Pascal*)

[12] ISO/IEC 10279:1991, *Information technology – Programming languages – Full BASIC.* (Also: ANSI X3.113-1987 *American National Standard for Information Systems – Programming Languages – Full BASIC*)

[13] ISO/IEC 10514:—, *Information technology – Programming languages – Modula-2.* (to be published)

[14] ISO/IEC 10967-2:—, *Information technology – Language independent arithmetic – Part 2: Mathematical procedures.* (to be published)

[15] ISO/IEC 10967-3:—, *Information technology – Language independent arithmetic – Part 3: Complex arithmetic and procedures.* (to be published)

[16] ISO/IEC JTC1/SC22/WG9 N102, *Proposed Standard for a Generic Package of Elementary Functions for Ada*, Draft 1.2, 12 Dec 1990.

[17] ISO/IEC JCT1/SC22/WG9 N103, *Proposed Standard for a Generic Package of Primitive Functions for Ada*, Draft 1.0, 13 Dec 1990.

[18] ISO/IEC JTC1/SC22/WG13 N738, *Modula-2 Standard*, Third Working Draft, BSI, 29 October 1989.

National and Other Documents

[19] ANSI/IEEE 770/X3.97-1983, *American National Standard / IEEE Standard / Pascal Computer Programming Language.*

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

[21] ANSI X3.226, *American National Standard for Information Systems – Programming Language – Common Lisp* (Draft 12.24).

Books and Articles

[22] W S Brown, A Simple but Realistic Model of Floating-Point Computation, *ACM Transactions on Mathematical Software*, Vol. 7, 1981, pp.445-480.

[23] J T Coonen, An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic, *Computer*, January 1980.

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

[25] 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.

[26] J E Holm, *Floating Point Arithmetic and Program Correctness Proofs*, Cornell University TR 80-436, 1980.

[27] C B Jones, *Systematic Software Development Using VDM*, Prentice-Hall, 1986.

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

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

[30] N L Schryer, *A Test of a Computer's Floating-Point Unit*, Computer Science Technical Report No. 89, AT&T Bell Laboratories, Murray Hill, N.J., 1981.

[31] 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.

[32] G L Steele Jr, *Common Lisp: The Language*, Digital Press, 1984.

[33] B A Wichmann, *Floating-Point Interval Arithmetic for Validation*, NPL Report DITC 76/86, 1986.

[34] B A Wichmann, *Getting the Correct Answers*, NPL Report DITC 167/90, June 1990.

[35] B A Wichmann, *The Language Compatible Arithmetic Standard and Ada*, NPL Report DITC 173/91, 1991.

[36] B A Wichmann, Towards a Formal Definition of Floating Point, *Computer Journal*, Vol. 32, October 1989, pp.432-436.

[37] N Wirth, *Programming in Modula-2*, Springer Verlag, 1988.

Annex J
(informative)

Glossary

This annex is provided as an aid to the reader who may not be familiar with the terms used in the other annexes. All definitions from 4.2 are repeated verbatim.

accuracy: This term applies to floating point only and gives a measure of the agreement between a computed result and the corresponding true mathematical result.

arithmetic data type: A data type whose values are members of $\mathcal{Z}$, $\mathcal{R}$, or $\mathcal{C}$.

> NOTE 1 – This part of ISO/IEC 10967 specifies requirements for integer and floating point data types. Complex numbers are not covered here, but will be included in a subsequent part of ISO/IEC 10967 [15].

axiom: A general rule satisfied by an operation and all values of the data type to which the operation belongs. As used in the specifications of operations, axioms are requirements.

boolean: A logical or truth value: **true** or **false**.

complex number: Numbers which include the real numbers as a special case and are often represented as $x + iy$ where $i = \sqrt{-1}$.

> NOTE 2 – Complex numbers are not covered by this part of ISO/IEC 10967, but will be included in a subsequent part of ISO/IEC 10967 [15].

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. (Contrast with *exceptional value*. See 6.1.2.)

> NOTE 3 – The infinities and NaNs produced by an IEC 559 system are examples of continuation values.

data type: A set of values and a set of operations that manipulate those values.

denormalization loss: A larger than normal rounding error caused by the fact that denormalized values have less than full precision. (See 5.2.5 for a full definition.)

denormalized: Those values of a floating point type F that provide less than the full precision allowed by that type. (See F_D in 5.2 for a full definition.)

elementary function: A function such as *sin*. These functions are usually evaluated as a sequence of operations and therefore may have lower accuracy than the basic operations.

> NOTE 4 – This part of ISO/IEC 10967 does not include specifications for elementary functions, which will be included in a subsequent part of ISO/IEC 10967, presently under development. [14]

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

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

exception: The inability of an operation to return a suitable numeric result. This might arise because no such result exists mathematically, or because the mathematical result cannot be represented with sufficient accuracy.

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

NOTES

5 Exceptional values are used as part of the defining formalism only. With respect to this part of ISO/IEC 10967, they do not represent values of any of the data types described. There is no requirement that they be represented or stored in the computing system.

6 Exceptional values are not to be confused with the NaNs and infinities defined in IEC 559. Contrast this definition with that of *continuation value* above.

exponent: The integer power to which the radix is raised in the representation of a floating point number. See the definition of *floating point number* below.

exponent bias: A number added to the exponent of a floating point number, usually to transform the exponent to an unsigned integer.

floating point: The arithmetic data type used in this part of ISO/IEC 10967 to approximate the real numbers $\mathcal{R}$.

floating point number: A member of a subset of $\mathcal{R}$, whose value is either zero or can be given in the form

$$\pm 0.f_1 f_2 ... f_p * r^e$$

where the radix r is the base associated with its data type, the exponent e is an integer between $emin$ and $emax$, and f_1, f_2, ... f_p are radix r digits.

fraction: The fractional part $0.f_1 f_2 ... f_p$ of a floating point number.

gradual underflow: The use of denormalized floating point format to decrease the chance that floating point calculations will underflow.

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.

identity: A relation among two or more operations of the same data type which holds for all values of the data type. An identity is derivable from the axioms satisfied by the operations involved.

implementation (of this part of ISO/IEC 10967)**:** The total arithmetic environment presented to a programmer, including hardware, language processors, exception handling facilities, subroutine libraries, other software, and all pertinent documentation.

integer: An element of $\mathcal{Z}$.

LIA-1: A reference to this part of ISO/IEC 10967.

mantissa: See the definition of *fraction* above, which is the term used in this part of ISO/IEC 10967.

NaN: "Not a Number," a non-numeric value used in some systems to represent the result of a numeric operation which has no result representable in the numeric data type.

normalized: 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.)

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.)

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

overflow: Integer overflow occurs for bounded integers if the integer result of an operation is greater than *maxint* or is less than *minint*.

Floating overflow occurs if the magnitude of the floating point result of an operation is greater than *fmax*, the maximum floating point number in the specified data type.

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

radix: The base associated with a floating point data type. In current practice, the radix is 2, 8, 10 or 16.

representable: A term used to describe a real number which is exactly equal to a floating point number.

rounding: The act of computing a representable final result for an operation that is close to the exact (but unrepresentable) result for that operation. Note that a suitable representable result may not exist (see 5.2.6). (See also A.5.2.5 for some examples.)

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

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

Note that if $u \in \mathcal{R}$ is between two adjacent values in X, $rnd(u)$ selects one of those adjacent values.

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

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

round toward zero: 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 0.

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 [2].)

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 [2].)

signature (of a function or operation)**:** A summary of information about an operation or function. A signature includes the operation name, the minimum set of inputs to the operation, and the maximum set of outputs from the operation (including exceptional values if any). The signature

$$add_I : I \times I \rightarrow I \cup \{\textbf{integer\_overflow}\}$$

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

A signature for an operation or function does not forbid the operation from accepting a wider range of inputs, nor does it guarantee that every value in the output range will actually be returned for some input. An operation given inputs outside the stipulated input range may produce results outside the stipulated output range.

significand: A term used in the IEEE standards 754 and 854 to denote the counterpart of the word *fraction* used in this part of ISO/IEC 10967. It has the value $f_0.f_1f_2...f_{p-1}$, where $f_0 \neq 0$ for normalized numbers and $f_0 = 0$ for denormalized numbers.

underflow: Underflow occurs if a floating point result has a value less in magnitude than $fmin_N$, the minimum normalized floating point number in the specified data type.

unnormalized: A non-zero floating point value for which $f_1 = 0$ in its fraction part $0.f_1f_2...f_p$. This part of ISO/IEC 10967 does not specify the properties of unnormalized numbers, except for the special case of denormalized numbers.

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 normalized value x, with exponent e, precision p, and radix r, is r^{e-p}.