INTERNATIONAL STANDARD

ISO/IEC 11729

First edition
1994-12-15

Information technology — Programming languages — Generic package of primitive functions for Ada

Technologies de l'information — Langages de programmation, leurs environnements et interfaces de logiciel de système — Ensemble générique de fonctions primitives pour l'Ada

Reference number
ISO/IEC 11729:1994(E)

Contents

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 11729 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee 22, *Programming languages, their environments and system software interfaces.*

Annex A forms an integral part of this International Standard. Annexes B and C are for information only.

Introduction

The generic package described here is intended to provide primitive operations that are required to endow mathematical software, such as implementations of the elementary functions, with the qualities of accuracy, efficiency and portability. With this International Standard, such mathematical software can achieve all of these qualities simultaneously; without it, one or more of them typically must be sacrificed.

The generic package specification included in this International Standard is presented as a compilable Ada specification in annex A with explanatory text in numbered clauses in the main body of text. The explanatory text is normative, with the exception of notes.

The word "may" as used in this International Standard consistently means "is allowed to" (or "are allowed to"). It is used only to express permission, as in the commonly occurring phrase "an implementation may"; other words (such as "can," "could" or "might") are used to express ability, possibility, capacity or consequentiality.

In formulas, $\lfloor v \rfloor$ and $\lceil v \rceil$ mean the greatest integer less than or equal to v and the least integer greater than or equal to v, respectively, and other notations have their customary meaning.

This page intentionally left blank

Information technology — Programming languages — Generic package of primitive functions for Ada

1 Scope

This International Standard specifies primitive functions and procedures for manipulating the fraction part and the exponent part of machine numbers (see clause 6) of the generic floating-point type. Additional functions are provided for directed rounding to a nearby integer, for computing an exact remainder, for determining the immediate neighbors of a floating-point machine number, for transferring the sign from one floating-point machine number to another and for shortening a floating-point machine number to a specified number of leading radix-digits. Some subprograms are redundant in that they are combinations of other subprograms. This is intentional so that convenient calls and fast execution can be provided to the user.

These subprograms are intended to augment standard Ada operations and to be useful in portably implementing such packages as those providing real and complex elementary functions, where (for example) the steps of argument reduction and result construction demand fast, error-free scaling and remaindering operations.

This International Standard is applicable to programming environments conforming to ISO 8652:1987.

2 Normative reference

The following standard contains provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the edition indicated was valid. All standards are subject to revision, and parties to agreements based on this International Standard 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.

ISO 8652:1987, *Programming languages — Ada* (Endorsement of ANSI Standard 1815A-1983)

3 Subprograms provided

The following fifteen subprograms are provided:

```
EXPONENT    FRACTION      DECOMPOSE     COMPOSE    SCALE
FLOOR       CEILING       ROUND         TRUNCATE   REMAINDER
ADJACENT    SUCCESSOR     PREDECESSOR
COPY_SIGN   LEADING_PART
```

The `EXPONENT` and `FRACTION` functions and the `DECOMPOSE` procedure decompose a floating-point machine number into its constituent parts, whereas the `COMPOSE` function constructs a floating-point machine number from those parts. The `SCALE` function scales a floating-point machine number accurately by a power of the hardware radix. The `FLOOR`, `CEILING`, `ROUND` and `TRUNCATE` functions all yield an integer value (in floating-point format) "near" the given floating-point argument, using distinct methods of rounding. The `REMAINDER` function provides an accurate remainder for floating-point operands, using the semantics of the IEEE `REM` operation. The `ADJACENT`, `SUCCESSOR` and `PREDECESSOR`

functions yield floating-point machine numbers in the immediate vicinity of other floating-point machine numbers. The COPY_SIGN function transfers the sign of one floating-point machine number to another. The LEADING_PART function retains only the specified number of high-order radix-digits of a floating-point number, effectively replacing the remaining (low-order) radix-digits by zeros.

4 Instantiations

This International Standard describes a generic package, GENERIC_PRIMITIVE_FUNCTIONS, which the user must instantiate to obtain a package. It has two generic formal parameters: a generic formal type named FLOAT_TYPE and a generic formal type named EXPONENT_TYPE. At instantiation, the user must specify a floating-point subtype as the generic actual parameter to be associated with FLOAT_TYPE and an integer subtype as the generic actual parameter to be associated with EXPONENT_TYPE. These are referred to below as the "generic actual types." These types are used as the parameter and, where applicable, the result types of the subprograms contained in the generic package.

Depending on the implementation, the user may or may not be allowed to associate, with FLOAT_TYPE, a generic actual type having a range constraint (see clause 5). If allowed, such a range constraint will have the usual effect of causing CONSTRAINT_ERROR to be raised when a floating-point argument outside the user's range is passed in a call to one of the subprograms, or when one of the subprograms attempts to return a floating-point value (either as a function result or as a formal parameter of mode out) outside the user's range. Allowing the generic actual type associated with FLOAT_TYPE to have a range constraint also has some implications for implementors.

The user is allowed to associate any integer-type generic actual type with EXPONENT_TYPE. However, insufficient range in the generic actual type will have the usual effect of causing CONSTRAINT_ERROR to be raised when an integer-type argument outside the user's range is passed in a call to one of the subprograms, or when one of the subprograms attempts to return an integer-type value (either as a function result or as a formal parameter of mode out) outside the user's range. Further considerations are discussed in clause 5.

In addition to the body of the generic package itself, implementors may provide (non-generic) library packages that can be used just like instantiations of the generic package for the predefined floating-point types (in combination with INTEGER for EXPONENT_TYPE). The name of a package serving as a replacement for an instantiation of GENERIC_PRIMITIVE_FUNCTIONS in which FLOAT_TYPE is equated with FLOAT (and EXPONENT_TYPE with INTEGER) should be PRIMITIVE_FUNCTIONS; for LONG_FLOAT and SHORT_FLOAT, the names should be LONG_PRIMITIVE_FUNCTIONS and SHORT_PRIMITIVE_FUNCTIONS, respectively; etc. When such a package is used in an application in lieu of an instantiation of GENERIC_PRIMITIVE_FUNCTIONS, it shall have the semantics implied by this International Standard for an instantiation of the generic package. This International Standard does not prescribe names for implementor-supplied non-generic library packages serving as pre-instantiations of GENERIC_PRIMITIVE_FUNCTIONS for exponent types other than INTEGER.

5 Implementations

For the most part, the results specified for the subprograms in clause 9 do not permit the kinds of approximations allowed by Ada's model of floating-point arithmetic. For this reason, portable implementations of the body of GENERIC_PRIMITIVE_FUNCTIONS are not believed to be possible. An implementation of this International Standard in Ada may use pragma INTERFACE or other pragmas, unchecked conversion, machine-code insertions, representation clauses or other machine-dependent techniques as desired.

An implementor is assumed to have knowledge of the underlying hardware environment and is expected to utilize that knowledge to produce the exact results (or, in a few cases, highly constrained approximations) specified by this International Standard; for example, implementations may directly manipulate the exponent field and fraction field of floating-point numbers.

An implementation may impose a restriction that the generic actual type associated with FLOAT_TYPE shall not have a range constraint that reduces the range of allowable values. If it does impose this restriction, then the restriction shall be documented, and the effects of violating the restriction shall be one of the following:

— Compilation of a unit containing an instantiation of GENERIC_PRIMITIVE_FUNCTIONS is rejected.

— CONSTRAINT_ERROR or PROGRAM_ERROR is raised during the elaboration of an instantiation of GENERIC_PRIMITIVE_FUNCTIONS.

Conversely, if an implementation does not impose the restriction, then such a range constraint shall not be allowed, when included with the user's generic actual type, to interfere with the internal computations of the subprograms; that is, if the floating-point argument and result are within the range of the type, then the implementation shall return the result and shall not raise an exception (such as CONSTRAINT_ERROR).

An implementation shall not allow insufficient range in the user's generic actual type associated with EXPONENT_TYPE to interfere with the internal computations of a subprogram when the range is sufficient to accommodate the integer-type arguments and integer-type results of the subprogram.

An implementation shall function properly in a tasking environment. Apart from the obvious restriction that an implementation of GENERIC_PRIMITIVE_FUNCTIONS shall avoid declaring variables that are global to the subprograms, no special constraints are imposed on implementations. Nothing in this International Standard requires the use of such global variables.

Some hardware and their accompanying Ada implementations have the capability of representing and discriminating between positively and negatively signed zeros as a means (for example) of preserving the sign of an infinitesimal quantity that has underflowed to zero. Implementations of GENERIC_PRIMITIVE_FUNCTIONS may exploit that capability, when available, in appropriate ways. At the same time, implementations in which that capability is unavailable are also allowed. Because a definition of what comprises the capability of representing and distinguishing signed zeros is beyond the scope of this International Standard, implementations are allowed the freedom not to exploit the capability, even when it is available. This International Standard leaves unspecified in some cases the sign that an implementation exploiting signed zeros shall give to a zero result; it does, however, specify that an implementation exploiting signed zeros shall yield a result for COPY_SIGN that depends on the sign of a zero argument. An implementation shall exercise its choice consistently, either exploiting signed-zero behavior everywhere or nowhere in this package. In addition, an implementation shall document its behavior with respect to signed zeros.

NOTE — It is intended that implementations of FLOOR, CEILING, ROUND and TRUNCATE determine the result without an intermediate conversion to an integer type, which might raise an exception.

6 Machine numbers and storable machine numbers

In the broad sense, a floating-point "machine number" of type FLOAT_TYPE is any number that can arise in the course of computing with operands and operations of that type. The set of such numbers depends on the implementation of Ada. Some implementations hold intermediate results in extended registers having a longer fraction part and/or wider exponent range than the storage cells that hold the values of variables. Thus, in the broad sense, there can be two or more different representations of floating-point machine numbers of type FLOAT_TYPE.

One such representation is that of the set of "storable" floating-point machine numbers. This representation is assumed to be the one characterized by the representation attributes of FLOAT_TYPE—for example (and in particular), FLOAT_TYPE'MACHINE_MANTISSA, FLOAT_TYPE'BASE'FIRST and FLOAT_TYPE'BASE'LAST. The significance of the storable floating-point machine numbers is that they can be assumed to be propagated by assignment, parameter association and function returns; because of the limited lifetime of values held in extended registers, there is no guarantee that a floating-point machine number outside this subset, once generated, can be so propagated.

The machine numbers referred to subsequently in this International Standard are to be understood to be storable floating-point machine numbers. An implementation of GENERIC_PRIMITIVE_FUNCTIONS is thus entitled to assume that the arguments of all of its subprograms are always storable floating-point machine numbers; furthermore, to support this International Standard, an implementation of Ada shall guarantee that only storable floating-point machine numbers are received as arguments by these subprograms. Without the assumption and the restriction, the exact results specified by this International Standard would be unrealistic (because, for example, they would imply that

extra-precise results must be delivered when extra-precise arguments are received), and those specified for ADJACENT, SUCCESSOR and PREDECESSOR would not even be well-defined.

The storability of a subprogram's arguments does not always guarantee that the desired mathematical result is representable as a storable floating-point machine number. In the few subprograms where the desired mathematical result can sometimes be unrepresentable, the actual result is permitted to be a specified approximation of the mathematical result, or it is omitted and replaced by the raising of an exception (see clause 8).

The term "neighboring machine number" is used in two contexts in this International Standard.

— When a desired mathematical result α is not representable but lies within the range of machine numbers, it necessarily falls between two adjacent machine numbers, the one immediately above and the one immediately below; those two numbers are referred to as the "machine numbers neighboring α."

— Every machine number X except the most positive (FLOAT_TYPE'BASE'LAST) has a nearest neighbor in the positive direction, and every one except the most negative (FLOAT_TYPE'BASE'FIRST) has a nearest neighbor in the negative direction; each is referred to as the "machine number neighboring X" in the given direction.

In both cases, the identity of the neighboring machine numbers is uniquely (if here only informally) determined by the fact that the set of machine numbers is understood to be the set of storable machine numbers (having FLOAT_TYPE'MACHINE_MANTISSA radix-digits in the fractional part of their canonical form) and is totally ordered.

7 Denormalized numbers

On machines fully or partially obeying IEEE arithmetic, the denormalized numbers are included in the set of machine numbers if the implementation of Ada uses the hardware in such a way that they can arise from normal Ada arithmetic operations (such implementations are said in this International Standard to "recognize denormalized numbers"); otherwise, they are not. Whether an implementation recognizes denormalized numbers determines whether the results of some subprograms, for particular arguments, are exact or approximate; it is also taken into account in determining the results that can be produced by the ADJACENT, SUCCESSOR and PREDECESSOR functions.

As used in this International Standard, a nonzero quantity α is said to be "in the denormalized range" when $|\alpha| <$ FLOAT_TYPE'MACHINE_RADIX$^{\texttt{(FLOAT_TYPE'MACHINE_EMIN}-1)}$; the term "canonical form of a floating-point number" is taken from the Ada Reference Manual, but its applicability is here extended to denormalized numbers by allowing the leading digit of the fractional part to be zero when the exponent part is equal to FLOAT_TYPE'MACHINE_EMIN.

8 Exceptions

Various conditions can make it impossible for a subprogram in GENERIC_PRIMITIVE_FUNCTIONS to deliver a result. Whenever this occurs, the subprogram raises an exception instead. No exceptions are declared in GENERIC_PRIMITIVE_FUNCTIONS; thus, only predefined exceptions are raised, as described below.

The REMAINDER function performs an operation related to division. When its second argument is zero, it raises the exception specified by Ada for signaling division by zero (this is NUMERIC_ERROR in the Ada Reference Manual, but it is changed to CONSTRAINT_ERROR by AI-00387).

The result defined for the SCALE, COMPOSE, SUCCESSOR, PREDECESSOR and, on some hardware, COPY_SIGN functions can exceed the overflow threshold of the hardware. When this occurs (or, more precisely, when the defined result falls outside the range FLOAT_TYPE'BASE'FIRST to FLOAT_TYPE'BASE'LAST), the function raises the exception specified by Ada for signaling overflow (this is NUMERIC_ERROR in the Ada Reference Manual, but it is changed to CONSTRAINT_ERROR by AI-00387).

All of the subprograms, as stated in clause 4, are subject to raising CONSTRAINT_ERROR when an integer-type value outside the bounds of the user's generic actual type associated with EXPONENT_TYPE is passed as an argument, or

when one of the subprograms attempts to return such an integer-type value. Similarly, if the implementation allows range constraints in the generic actual type associated with FLOAT_TYPE, then CONSTRAINT_ERROR will be raised when the value of a floating-point argument lies outside the range of that generic actual type, or when a subprogram in GENERIC_PRIMITIVE_FUNCTIONS attempts to return a value outside that range. Additionally, all of the subprograms are subject to raising STORAGE_ERROR when they cannot obtain the storage they require.

Whereas a result that is too large to be represented causes the signaling of overflow, a result that is too small to be represented exactly does *not* raise an exception; such a result, which can be computed by SCALE, COMPOSE and REMAINDER, is instead approximated (possibly by zero), as specified separately for each of these subprograms.

The only exceptions allowed during an instantiation of GENERIC_PRIMITIVE_FUNCTIONS, including the execution of the optional sequence of statements in the body of the instance, are CONSTRAINT_ERROR, PROGRAM_ERROR and STORAGE_ERROR, and then only for the reasons given in this paragraph. The raising of CONSTRAINT_ERROR during instantiation is only allowed when the implementation imposes the restriction that the generic actual type associated with FLOAT_TYPE shall not have a range constraint, and the user violates that restriction (it can, in fact, be an inescapable consequence of the violation). The raising of PROGRAM_ERROR during instantiation is only allowed for the purpose of signaling errors made by the user—for example, violation of this same restriction. The raising of STORAGE_ERROR during instantiation is only allowed for the purpose of signaling the exhaustion of storage.

9 Specifications of the subprograms

Except where an approximation is explicitly allowed and defined, the formulas given below under the heading *Definition* specify precise mathematical results. In a few cases, these formulas leave a subprogram undefined for certain arguments; in those cases, the subprogram will raise an exception, as stated under the heading *Exceptions*, instead of delivering a result.

In the specifications of EXPONENT, FRACTION, DECOMPOSE, COMPOSE, SCALE and LEADING_PART, the symbol β stands for the value of FLOAT_TYPE'MACHINE_RADIX.

9.1 EXPONENT — Exponent of the Canonical Representation of a Floating-Point Machine Number

9.1.1 Specification

```
function EXPONENT (X : FLOAT_TYPE) return EXPONENT_TYPE;
```

9.1.2 Definition

a) EXPONENT(0.0) = 0.0

b) For X ≠ 0.0, EXPONENT(X) yields the unique integer k such that $\beta^{k-1} \leq |X| < \beta^k$

NOTE — When X is a denormalized number, EXPONENT(X) < FLOAT_TYPE'MACHINE_EMIN.

9.2 FRACTION — Signed Mantissa of the Canonical Representation of a Floating-Point Machine Number

9.2.1 Specification

```
function FRACTION (X : FLOAT_TYPE) return FLOAT_TYPE;
```