

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)

AVA Documents
Document Sharing Hub

Contents	Page
Foreword	iv
Introduction	v
1 Scope	1
2 Normative reference	1
3 Subprograms provided	1
4 Instantiations	2
5 Implementations	2
6 Machine numbers and storable machine numbers	3
7 Denormalized numbers	4
8 Exceptions	4
9 Specifications of the subprograms	5
9.1 EXPONENT — Exponent of the Canonical Representation of a Floating-Point Machine Number	5
9.2 FRACTION — Signed Mantissa of the Canonical Represen- tation of a Floating-Point Machine Number	5
9.3 DECOMPOSE — Extract the Components of the Canonical Representation of a Floating-Point Machine Number	6
9.4 COMPOSE — Construct a Floating-Point Machine Number from the Components of its Canonical Representation . . .	6
9.5 SCALE — Increment/Decrement the Exponent of the Canonical Representation of a Floating-Point Machine Number	7
9.6 FLOOR — Greatest Integer Not Greater Than a Floating- Point Machine Number, as a Floating-Point Number . . .	7

9.7 CEILING — Least Integer Not Less Than a Floating-Point Machine Number, as a Floating-Point Number 7

9.8 ROUND — Integer Nearest to a Floating-Point Machine Number, as a Floating-Point Number 7

9.9 TRUNCATE — Integer Part of a Floating-Point Machine Number, as a Floating-Point Number 8

9.10 REMAINDER — Exact Remainder Upon Dividing One Floating-Point Machine Number by Another..... 8

9.11 ADJACENT — Floating-Point Machine Number Next to One Floating-Point Machine Number in the Direction of a Second 8

9.12 SUCCESSOR — Floating-Point Machine Number Next Above a Given Floating-Point Machine Number 9

9.13 PREDECESSOR — Floating-Point Machine Number Next Below a Given Floating-Point Machine Number 9

9.14 COPY_SIGN — Transfer of Sign from One Floating-Point Machine Number to Another 10

9.15 LEADING_PART — Floating-Point Machine Number with its Mantissa (in the Canonical Representation) Truncated to a Given Number of Radix-Digits 10

Annexes

A Ada specification for GENERIC_PRIMITIVE_FUNCTIONS 11

B Rationale 12

 B.1 Introduction and motivation 12

 B.2 History 12

 B.3 Packaging 13

 B.4 Implementation permissions 13

 B.5 Accuracy requirements 13

 B.6 Discussion of individual subprograms 15

 B.7 Relationship to other standards 18

 B.8 Influence on Ada 9X 18

C Bibliography 19

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.

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