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$^{(\text{FLOAT_TYPE'MACHINE_EMIN}-1)}$; 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;
```

9.2.2 Definition

a) FRACTION(0.0) = 0.0

b) For X ≠ 0.0, FRACTION(X) = X · β^{-k}, where k is the unique integer such that $\beta^{k-1} \leq |X| < \beta^k$

9.3 DECOMPOSE — Extract the Components of a Floating-Point Machine Number

9.3.1 Specification

```
procedure DECOMPOSE (X        : in  FLOAT_TYPE;
                     FRACTION : out FLOAT_TYPE;
                     EXPONENT : out EXPONENT_TYPE);
```

9.3.2 Definition

a) FRACTION = 0.0 and EXPONENT = 0.0 upon return from an invocation of DECOMPOSE(0.0, FRACTION, EXPONENT)

b) For X ≠ 0.0, FRACTION = X · β^{-k} and EXPONENT = k, where k is the unique integer such that $\beta^{k-1} \leq |X| < \beta^k$, upon return from an invocation of DECOMPOSE(X, FRACTION, EXPONENT)

NOTE — When X is a denormalized number, EXPONENT < FLOAT_TYPE'MACHINE_EMIN upon return from an invocation of DECOMPOSE(X, FRACTION, EXPONENT).

9.4 COMPOSE — Construct a Floating-Point Machine Number from the Components of its Canonical Representation

9.4.1 Specification

```
function COMPOSE (FRACTION : FLOAT_TYPE;
                  EXPONENT : EXPONENT_TYPE) return FLOAT_TYPE
```

9.4.2 Definition

a) COMPOSE(0.0, EXPONENT) = 0.0 for any EXPONENT

b) For FRACTION ≠ 0.0, let α = FRACTION · $\beta^{\text{EXPONENT}-k}$, where k is the unique integer such that $\beta^{k-1} \leq |X| < \beta^k$. If α is exactly representable as a floating-point machine number (see clause 6), COMPOSE(FRACTION, EXPONENT) = α; otherwise, COMPOSE(FRACTION, EXPONENT) yields either one of the machine numbers neighboring α, provided that FLOAT_TYPE'BASE'FIRST < α < FLOAT_TYPE'BASE'LAST.

9.4.3 Exceptions

When α as defined above is outside the range of machine numbers, COMPOSE raises the exception specified by Ada for signaling overflow (see clause 8) instead of delivering a result.

NOTES

1 For FRACTION ≠ 0.0, this function can deliver an approximation (possibly zero) to the exact mathematical result α only when EXPONENT is sufficiently negative to force α to be in the denormalized range, and either the implementation does not recognize denormalized numbers, or α is not exactly representable as a denormalized number (see clause 7).

2 The name FRACTION is not meant to suggest that the first argument is restricted to fractional values; rather, it is meant to suggest that the first argument supplies (via its fractional part in the canonical form) the fractional part of the result.

9.5 SCALE — Increment/Decrement the Exponent of the Canonical Representation of a Floating-Point Machine Number

9.5.1 Specification

```
function SCALE (X          : FLOAT_TYPE;
                ADJUSTMENT : EXPONENT_TYPE) return FLOAT_TYPE;
```

9.5.2 Definition

Let $\alpha = X \cdot \beta^{ADJUSTMENT}$. If α is exactly representable as a floating-point machine number (see clause 6), SCALE(X, ADJUSTMENT) = α; otherwise, SCALE(X, ADJUSTMENT) yields either one of the machine numbers neighboring α, provided that FLOAT_TYPE'BASE'FIRST $< \alpha <$ FLOAT_TYPE'BASE'LAST.

9.5.3 Exceptions

When α as defined above is outside the range of machine numbers, SCALE raises the exception specified by Ada for signaling overflow (see clause 8), instead of delivering a result.

NOTE — This function can deliver an approximation (possibly zero) to the exact mathematical result α only when ADJUSTMENT is sufficiently negative to force α to be in the denormalized range, and either the implementation does not recognize denormalized numbers, or α is not exactly representable as a denormalized number (see clause 7).

9.6 FLOOR — Greatest Integer Not Greater Than a Floating-Point Machine Number, as a Floating-Point Number

9.6.1 Specification

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

9.6.2 Definition

FLOOR(X) = $\lfloor X \rfloor$

NOTE — For sufficiently large |X|, this function merely returns its argument.

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

9.7.1 Specification

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

9.7.2 Definition

CEILING(X) = $\lceil X \rceil$

NOTE — For sufficiently large |X|, this function merely returns its argument.

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

9.8.1 Specification

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

9.8.2 Definition

`ROUND(X)` yields the integer nearest to `X`; if `X` is equidistant from two integers, then the even integer is chosen.

NOTE — For sufficiently large $|X|$, this function merely returns its argument.

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

9.9.1 Specification

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

9.9.2 Definition

$$\texttt{TRUNCATE(X)} = \begin{cases} \lfloor X \rfloor, & X \geq 0.0 \\ \lceil X \rceil, & X < 0.0 \end{cases}$$

NOTE — For sufficiently large $|X|$, this function merely returns its argument.

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

9.10.1 Specification

```
function REMAINDER (X, Y : FLOAT_TYPE) return FLOAT_TYPE;
```

9.10.2 Definition

For $Y \neq 0.0$, let $\alpha = X - (Y \cdot n)$, where n is the integer nearest to the exact value of X/Y; if $|n - (X/Y)| = 1/2$, then n is chosen to be even. If α is exactly representable as a floating-point machine number (see clause 6), `REMAINDER(X, Y)` $= \alpha$; otherwise, `REMAINDER(X, Y)` $= 0.0$.

9.10.3 Exceptions

For any `X`, `REMAINDER(X, 0.0)` raises the exception specified by Ada for signaling division by zero (see clause 8) instead of delivering a result.

NOTES

1 This function can deliver an approximation (namely, zero) to the exact mathematical result α only when `Y` is in the neighborhood of zero, `X` is sufficiently close to a multiple of `Y` to force α to be in the denormalized range, and the implementation does not recognize denormalized numbers (see clause 7).

2 The magnitude of the result is less than or equal to $|Y/2|$.

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

9.11.1 Specification

```
function ADJACENT (X, TOWARDS : FLOAT_TYPE) return FLOAT_TYPE;
```

9.11.2 Definition

a) `ADJACENT(X, X) = X`

b) For `TOWARDS` $\neq$ `X`, `ADJACENT(X, TOWARDS)` yields the floating-point machine number (see clause 6) neighboring `X` in the direction toward `TOWARDS`; in an implementation exploiting signed zeros (see clause 5), a zero result has the sign of `X`.

NOTES

1 Unlike `SUCCESSOR` and `PREDECESSOR`, to which it is related, `ADJACENT` never raises an exception.

2 For certain normalized arguments, the numerical value of the result depends on whether or not the implementation recognizes denormalized numbers (see clause 7). For example, for `TOWARDS` $\neq$ `0.0`, `ADJACENT(0.0, TOWARDS)` yields a denormalized number if the implementation recognizes denormalized numbers, and a normalized number otherwise. Similarly, `ADJACENT(`$\pm\sigma$`, 0.0)`, where σ is the smallest positive normalized number, yields a denormalized number if the implementation recognizes denormalized numbers, and zero otherwise.

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

9.12.1 Specification

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

9.12.2 Definition

`SUCCESSOR(X)` yields the floating-point machine number (see clause 6) neighboring `X` in the positive direction, provided that `X` $\neq$ `FLOAT_TYPE'BASE'LAST`; in an implementation exploiting signed zeros (see clause 5), a zero result yields a negatively signed zero.

9.12.3 Exceptions

Since there is no floating-point machine number neighboring `FLOAT_TYPE'BASE'LAST` in the positive direction (a consequence of the assumption and restriction in clause 6), `SUCCESSOR` raises the exception specified by Ada for signaling overflow (see clause 8), instead of delivering a result, when `X` = `FLOAT_TYPE'BASE'LAST`.

NOTE — For certain arguments, the numerical value of the result depends on whether or not the implementation recognizes denormalized numbers (see clause 7). For example, `SUCCESSOR(0.0)` yields a denormalized number if the implementation recognizes denormalized numbers, and a normalized number otherwise. Similarly, `SUCCESSOR(`$-\sigma$`)`, where σ is the smallest positive normalized number, yields a denormalized number if the implementation recognizes denormalized numbers, and zero otherwise.

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

9.13.1 Specification

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

9.13.2 Definition

`PREDECESSOR(X)` yields the floating-point machine number (see clause 6) neighboring `X` in the negative direction, provided that `X` $\neq$ `FLOAT_TYPE'BASE'FIRST`; in an implementation exploiting signed zeros (see clause 5), a zero result yields a positively signed zero.

9.13.3 Exceptions

Since there is no floating-point machine number neighboring FLOAT_TYPE'BASE'FIRST in the negative direction (a consequence of the assumption and restriction in clause 6), PREDECESSOR raises the exception specified by Ada for signaling overflow (see clause 8), instead of delivering a result, when X = FLOAT_TYPE'BASE'FIRST.

NOTE — For certain arguments, the numerical value of the result depends on whether or not the implementation recognizes denormalized numbers (see clause 7). For example, PREDECESSOR(0.0) yields a denormalized number if the implementation recognizes denormalized numbers, and a normalized number otherwise. Similarly, PREDECESSOR(σ), where σ is the smallest positive normalized number, yields a denormalized number if the implementation recognizes denormalized numbers, and zero otherwise.

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

9.14.1 Specification

```
function COPY_SIGN (VALUE, SIGN : FLOAT_TYPE) return FLOAT_TYPE;
```

9.14.2 Definition

a) In an implementation exploiting signed zeros (see clause 5), COPY_SIGN(VALUE, SIGN) delivers a result having the magnitude of VALUE and the sign of SIGN.

b) In an implementation not exploiting signed zeros,

$$\text{COPY\_SIGN(VALUE, SIGN)} = \begin{cases} 0.0, & \text{VALUE} = 0.0 \\ |\text{VALUE}|, & \text{VALUE} \neq 0.0 \text{ and SIGN} \geq 0.0 \\ -|\text{VALUE}|, & \text{VALUE} \neq 0.0 \text{ and SIGN} < 0.0 \end{cases}$$

9.14.3 Exceptions

Since the negation of some representable values causes overflow on some hardware (e.g., when 2's-complement representation is used for floating-point), COPY_SIGN raises the exception specified by Ada for signaling overflow (see clause 8), instead of delivering a value, in that case.

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

9.15.1 Specification

```
function LEADING_PART (X            : FLOAT_TYPE;
                       RADIX_DIGITS : POSITIVE) return FLOAT_TYPE;
```

9.15.2 Definition

a) LEADING_PART(0.0, RADIX_DIGITS) = 0.0 for any RADIX_DIGITS

b) For X > 0.0, LEADING_PART(X, RADIX_DIGITS) = $\lfloor \text{X}/\beta^{k-\text{RADIX\_DIGITS}} \rfloor \cdot \beta^{k-\text{RADIX\_DIGITS}}$, where k is the unique integer such that $\beta^{k-1} \leq |\text{X}| < \beta^k$

c) For X < 0.0, LEADING_PART(X, RADIX_DIGITS) = $\lceil \text{X}/\beta^{k-\text{RADIX\_DIGITS}} \rceil \cdot \beta^{k-\text{RADIX\_DIGITS}}$, where k is the unique integer such that $\beta^{k-1} \leq |\text{X}| < \beta^k$

NOTE — For RADIX_DIGITS ≥ FLOAT_TYPE'MACHINE_MANTISSA, this function merely returns its first argument.

Annex A
(normative)
Ada specification for GENERIC_PRIMITIVE_FUNCTIONS

```
generic
   type FLOAT_TYPE    is digits <>;
   type EXPONENT_TYPE is range  <>;
package GENERIC_PRIMITIVE_FUNCTIONS is

   function  EXPONENT     (X            :     FLOAT_TYPE)    return EXPONENT_TYPE;
   function  FRACTION     (X            :     FLOAT_TYPE)    return FLOAT_TYPE;
   procedure DECOMPOSE    (X            : in  FLOAT_TYPE;
                           FRACTION     : out FLOAT_TYPE;
                           EXPONENT     : out EXPONENT_TYPE);
   function  COMPOSE      (FRACTION     :     FLOAT_TYPE;
                           EXPONENT     :     EXPONENT_TYPE) return FLOAT_TYPE;
   function  SCALE        (X            :     FLOAT_TYPE;
                           ADJUSTMENT   :     EXPONENT_TYPE) return FLOAT_TYPE;

   function  FLOOR        (X            :     FLOAT_TYPE)    return FLOAT_TYPE;
   function  CEILING      (X            :     FLOAT_TYPE)    return FLOAT_TYPE;
   function  ROUND        (X            :     FLOAT_TYPE)    return FLOAT_TYPE;
   function  TRUNCATE     (X            :     FLOAT_TYPE)    return FLOAT_TYPE;
   function  REMAINDER    (X, Y         :     FLOAT_TYPE)    return FLOAT_TYPE;

   function  ADJACENT     (X, TOWARDS   :     FLOAT_TYPE)    return FLOAT_TYPE;
   function  SUCCESSOR    (X            :     FLOAT_TYPE)    return FLOAT_TYPE;
   function  PREDECESSOR  (X            :     FLOAT_TYPE)    return FLOAT_TYPE;

   function  COPY_SIGN    (VALUE, SIGN  :     FLOAT_TYPE)    return FLOAT_TYPE;
   function  LEADING_PART (X            :     FLOAT_TYPE;
                           RADIX_DIGITS :     POSITIVE)      return FLOAT_TYPE;

end GENERIC_PRIMITIVE_FUNCTIONS;
```

Annex B
(informative)
Rationale

B.1 Introduction and motivation

At about the time that work on a proposed Ada standard for the elementary functions began in 1986, early efforts to implement the elementary functions—square root, logarithm, trigonometric functions, and the like—underscored the need to be able to perform certain steps in their computation with extreme accuracy. These functions are typically implemented by transforming the argument so that it lies within a reduced range, computing the desired function on the transformed argument by a polynomial or rational approximation (designed to be sufficiently accurate over the relatively narrow reduced argument range) to obtain an intermediate result, and then constructing the final result by appropriately transforming the intermediate result. Accuracy is controlled in the middle step by the choice of approximation method, which bounds the approximation error. However, the final result can be extremely sensitive to errors (such as roundoff errors) made in the argument reduction step. Unnecessary error can also enter in the final step if the transformation it represents is not carried out carefully.

Details of the transformations needed in the argument reduction and result construction steps depend, of course, on the function being implemented. In the case of the periodic functions, the essential requirement is to compute an accurate remainder when the argument is divided by the period, if specified; when the period is allowed to default to the irrational 2π, a technique other than a simple division is required to obtain a suitably accurate remainder. In other cases, especially `SQRT` and `LOG`, decomposition of the argument into its exponent and fraction parts is the starting point, with the fraction part (or a simple function of it) becoming the transformed argument; the result construction step in these cases usually involves a simple modification—often just a scaling—of the intermediate result by a simple function of the exponent part.

If one is interested in implementing the elementary functions in a portable fashion, how does one go about computing accurate floating-point remainders and decomposing floating-point numbers into their constituent parts portably? Two problems arise if one tries to do these things entirely in portable Ada: the result is inefficient, often involving loops that require many traversals; and it cannot be proven fully accurate with Ada's model of floating-point arithmetic, since the model caters to the weaknesses of the weakest conforming implementation of Ada. (On machines manifesting them, such weaknesses—for example, lack of a guard digit—can introduce errors in the argument reduction step that become amplified as the loops are traversed.) The efficiency and accuracy problems can be solved, of course, by judicious use of representation clauses or interface programming in assembler language or even machine language insertions, given knowledge of the host machine, but that obviously destroys portability.

Exact floating-point remainder and decomposition of a floating-point number into its constituent parts are two examples of *primitive functions*—low-level floating-point functions having the property that they cannot be coded in Ada so as to be simultaneously accurate, efficient and portable. Since the accuracy and efficiency problems vanish when details of the underlying machine are taken into account (indeed, some of the primitive functions are directly available as hardware operations on specific machines), all that is really lacking is a standardized interface to the functions. That is what this International Standard provides.

Portable implementations of the generic elementary functions standard (ISO/IEC 11430:1994 [9]) will be the first beneficiary of this International Standard; others will follow. However, this International Standard will always have a specialized clientele: experts, probably highly trained numerical analysts, concerned with the development of high-quality, portable mathematical software. It is not for the average application programmer.

B.2 History

This International Standard has been developed by the ACM SIGAda Numerics Working Group (reporting to the WG9 Numerics Rapporteur Group) in collaboration with the Ada-Europe Numerics Working Group. The standardization effort has been supported and encouraged in the United States by the Ada Joint Program Office of the U.S. Department of Defense, and in Europe by the Commission of the European Communities.

Although work on ISO/IEC 11430:1994 and this International Standard began at about the same time, the former was essentially completed, except for some late refinements, about a year and a half earlier. Earliest drafts of this International Standard drew heavily from recommendations made many years earlier in [3]; other works influencing the Ada primitive functions at an early date were [5, 10, 12, 11]. Later versions of the primitive functions were influenced by the IEEE floating-point standards [6, 2] and by Part 1 of the proposed Language Independent Arithmetic Standard (LIA-1) [8]. One reason for the delay in completing the primitive functions standard, relative to the elementary functions standard, was a series of late additions to the former as the result of evolving implementation experience with the latter. Another was the recognition that software intending to exploit IEEE arithmetic had to pay particular attention to some of its more subtle features, such as denormalized numbers and signed zeros. It took considerable effort to describe the primitive functions so that they could be implemented in either IEEE or non-IEEE environments. This issue also had ramifications for the elementary functions standard, resulting in minor revisions late in the approval process.

B.3 Packaging

This International Standard defines the specification of a generic package called `GENERIC_PRIMITIVE_FUNCTIONS`. It is a package because that is the accepted way to collect together several related subprograms; it is generic, with generic formal parameters for the two types used for the arguments and results of the subprograms, in view of the rules for parameter associations and the inability to anticipate the types used in applications. The generic formal parameter `FLOAT_TYPE` gives the type to be used for the floating-point arguments and results of subprograms in `GENERIC_PRIMITIVE_FUNCTIONS`, while the generic formal parameter `EXPONENT_TYPE` gives the type to be used for the few integer arguments and results that, with one exception,[1)] deal with exponents of the canonical machine representation. When an instantiation of `GENERIC_PRIMITIVE_FUNCTIONS` is used in an implementation of the elementary functions (e.g., in the body of `GENERIC_ELEMENTARY_FUNCTIONS`), the `FLOAT_TYPE` of the latter should be passed through to the former, and a sufficiently wide integer type should be associated with `EXPONENT_TYPE`. The predefined type `INTEGER` probably suffices for the latter, but if one is worried about sufficient range, then an integer type whose range covers `SYSTEM.MIN_INT .. SYSTEM.MAX_INT` can be defined and used instead.

B.4 Implementation permissions

Like the elementary functions standard, the primitive functions standard permits implementations to impose a restriction that the generic actual type associated with `FLOAT_TYPE` in an instantiation shall not contain a range constraint that reduces the range of allowable values. Implementations choosing not to impose the restriction must be designed to be immune from the avoidable effects of such range constraints; in general, this means that variables of type `FLOAT_TYPE` cannot safely be used for intermediate results within an implementation of `GENERIC_PRIMITIVE_FUNCTIONS`. Those imposing the restriction must document it; they can safely use such variables, but they must behave in one of several stated ways (i.e., predictably) if the restriction is violated. (For a detailed discussion of the genesis of this optional restriction and its implications, see Annex C (Rationale) of [9].) Implementations are not allowed to impose a similar restriction on the generic actual types that can be associated with `EXPONENT_TYPE` during instantiation; it is not difficult to implement `GENERIC_PRIMITIVE_FUNCTIONS` to be efficient while limiting the consequences of insufficient range in that generic actual type to the unavoidable raising of `CONSTRAINT_ERROR` during a subprogram call or return.

B.5 Accuracy requirements

Perhaps the most significant difference between the two standards, other than subject area, is their respective handling of accuracy requirements. The elementary functions standard allows implementations to approximate the exact mathematical result but constrains the approximation error by requiring implementations to satisfy "maximum relative error" bounds. In contrast, the primitive functions standard requires implementations to deliver the exact mathematical result defined for each function, whenever that result is representable; approximations are permitted only when the mathematical result is not representable and is smaller in magnitude than the smallest normalized positive floating-point number; and even then, the result is constrained to be one of the adjacent representable numbers. This

[1)] One of the subprograms takes an argument that is a nonzero count of the number of digits to be retained in a particular computation; the predefined integer subtype `POSITIVE` is used for the corresponding parameter.

level of accuracy is an essential aspect of the definition of the functions as operations on machine numbers yielding related machine numbers, without which their utility in argument reduction, etc., would be compromised. Achieving the required accuracy is *not* a feat that can be accomplished portably in Ada, at least not without making assumptions about the performance of the hardware that go well beyond the requirements imposed by the Ada model of floating-point arithmetic. On the other hand, the required accuracy can be easily and efficiently achieved by targeting implementations for specific environments and by utilizing knowledge of the machine representations in conjunction with appropriate operations (often integer or bit operations), accessed, if necessary, through low-level interfaces. A precedent for the accuracy required of the primitive functions can be found in the Ada attribute `T'BASE'LAST` for a floating-point type `T`: by definition, it has full machine-number accuracy, which, in general, exceeds model-number and safe-number accuracy.

Because the primitive functions transform machine numbers into other well-defined machine numbers, this International Standard includes a discussion of exactly what is meant by "floating-point machine number" within the context of the subprograms' definitions. What numbers are in the set of machine numbers? Does that set include the extra-precise numbers that some Ada implementations generate as a consequence of using extended registers for intermediate results? The answer to the latter question must be no, for otherwise the precise mathematical formulas used to specify the results of some of the functions would imply that the output from a function must be extra-precise if its input is, and yet the implementor may have no means to ensure that that will be the case. Thus, it is clearly stated that the "machine numbers" referred to throughout this International Standard are the *storable* machine numbers—the ones that can be (a) stored; (b) propagated by assignment, parameter association and function returns; and (c) characterized by the representation attributes `FLOAT_TYPE'MACHINE_MANTISSA`, `FLOAT_TYPE'BASE'FIRST` and `FLOAT_TYPE'BASE'LAST`. Implementations of the primitive functions are entitled to assume that only storable machine numbers will be seen as arguments, and implementations of Ada must uphold that assumption (by forcing storage, if necessary, before calling a primitive function) in order for implementations of the primitive functions to have any hope of conforming to this International Standard.

Furthermore, because some hardware (e.g., that implementing IEEE arithmetic) has the capability of representing denormalized numbers—those with the minimum exponent and an unnormalized fraction part—one must also be precise about whether the set of machine numbers includes them. This International Standard says that it does if the hardware has the capability of representing them and the Ada implementation uses the hardware in such a way that it actually generates them; otherwise, it does not. This is especially significant when talking about "adjacent machine numbers," since the machine number adjacent to the smallest positive normalized number, in the direction toward zero, will be a denormalized number if the hardware and Ada implementation recognize denormalized numbers, and zero otherwise. It is also germane to the approximations that are permitted when a defined result falls in the denormalized range and is not exactly representable.

Some hardware (again, typically hardware conforming to IEEE arithmetic) has the capability of representing both positive and negative zeros (i.e., the sign of zero is relevant in some contexts). Like the elementary functions standard, the primitive functions standard allows signed zeros to be exploited if they are present in the hardware, but does not require them to be exploited. And like the elementary functions standard, the primitive functions standard does not give the required sign of each zero result (when signed zeros are being exploited), but leaves that to other standards or interpretations.[2)] The behavior of one of the primitive functions, `COPY_SIGN`, does depend on the sign of a zero argument (when signed zeros are being exploited), as is also true of `ARCTAN` and `ARCCOT` in the elementary functions. This International Standard also clarifies that plus and minus zero are *not* to be considered "adjacent" (and therefore different) machine numbers, in any context where adjacency is relevant; thus, the "neighbors" of zero do not depend on the sign of zero.

Early working drafts of this International Standard did not permit *any* approximations: when the exact mathematical result was not representable, they called for the raising of an exception to signal that fact. Indeed, this applied not just to underflow situations,[3)] but to overflow as well. An exception called `REPRESENTATION_ERROR` was reserved for that purpose. Commenting on an early draft, an observer convinced the committee that it would be better to signal overflow in the usual way (i.e., by raising the predefined exception provided by Ada for that contingency) and that it would also be better to provide a result conforming to the Ada standard in cases of underflow (including flushing

[2)] There are four exceptions, however. The required signs of zero results from `ADJACENT`, `SUCCESSOR`, `PREDECESSOR` and `COPY_SIGN` *are* spelled out in this International Standard because those functions are intimately concerned with representations.

[3)] For simplicity, this is understood to mean either actual underflow or merely denormalization, which is also known as "gradual underflow."

to zero, if nothing better could be done) instead of raising an exception. An overflow or underflow in the result of a primitive function is most likely to occur when the primitive function is used for scaling purposes in the final step of some other computation, such as that of an elementary function. In such a case, the elementary function would overflow or underflow as well, and it would be undesirable to force the latter to intercept a REPRESENTATION_ERROR exception arising in the former just so that it could substitute some other behavior. As this International Standard is now written, an overflow or underflow occurring in the result of a primitive function called to perform scaling in the final step of the computation of an elementary function can simply be propagated from the primitive function through the elementary function to the latter's caller, which will thus satisfy the requirements of the elementary functions standard in a most efficient way.

With underflows reported by approximations and overflows signaled by the appropriate predefined exception, there was no longer any need for the REPRESENTATION_ERROR exception, which was accordingly eliminated. No exceptions are declared by GENERIC_PRIMITIVE_FUNCTIONS. Only predefined exceptions may be raised by implementations of the primitive functions, and even those are restricted (as they were in the elementary functions standard) to specific cases where they are unavoidable.

B.6 Discussion of individual subprograms

The subprograms (fourteen functions and one procedure) in GENERIC_PRIMITIVE_FUNCTIONS can be organized into four groups for presentation purposes. In the discussion that follows, arguments and results are of the floating-point type FLOAT_TYPE except where noted, and β stands for the value of FLOAT_TYPE'MACHINE_RADIX.

The first group comprises basic decomposition, composition and scaling subprograms for floating-point numbers. These are the EXPONENT, FRACTION, COMPOSE and SCALE functions and the DECOMPOSE procedure.

EXPONENT is primarily useful in argument reduction steps, where it gives a coarse indication of the magnitude of its argument. Except when X = 0.0, the function EXPONENT(X) delivers—as a value of the integer type EXPONENT_TYPE—the unique integer k such that $\beta^{k-1} \leq |X| < \beta^k$. This definition is entirely mathematical and not related to the representation of X on the machine. Thus, as a positive X decreases through the normalized range and into the denormalized range, EXPONENT(X) continues to decrease, even though the exponent part of the machine representation of X stops decreasing when the smallest normalized number is reached. In fact, on the assumption that FLOAT_TYPE'MACHINE_EMIN is the value of that minimum exponent,[4] EXPONENT(X) can yield a value that is less than FLOAT_TYPE'MACHINE_EMIN (e.g., when X is denormalized). Finally, EXPONENT(0.0) is defined in this International Standard to deliver 0.

The EXPONENT function can be computed on machines lacking a direct hardware implementation by extracting and unbiasing the exponent field of the representation, with a special case for X = 0.0 and with additional steps required when X is denormalized.[5] EXPONENT corresponds closely to the IEEE recommended function logb, which is usually available in hardware, except that the result of EXPONENT is of an integer type instead of a floating-point type.

Some observers contended that EXPONENT(0.0) should not be 0; the most mathematically sensible alternative, $-\infty$, which can be represented on IEEE hardware at least, is ruled out by the integer-type result of EXPONENT. The committee staunchly preferred to stick with an integer type for the representation of the integer values delivered by this function, especially when it concluded that a result of zero for a zero argument is often a "don't care" case anyway (in the sense that the potential caller of EXPONENT will avoid the call and take a different path, when X = 0.0), and is probably harmless when not. Another alternative, raising an exception to signal an illegal argument when X = 0.0, was ruled out because it is unnecessarily harsh when a zero result is harmless.

The companion function FRACTION is also useful in argument reduction steps. For nonzero X, FRACTION(X) is defined to yield $X \cdot \beta^{-k}$, where k is as defined above for EXPONENT; FRACTION(0.0) yields 0.0. Thus, FRACTION(X) is the fraction part of the canonical form of the floating-point number X (normalized, however, when X is denormalized).

[4] This is a reasonable assumption, without which some numbers expressible in the canonical form would not be representable. It requires, however, that the definition of canonical form be relaxed to allow unnormalized fraction parts.

[5] When X is denormalized, the simplest strategy is to multiply X by a fixed power β^k of the hardware radix sufficient to normalize it, extract the exponent field, and then reduce the exponent by k.

This function can be computed on machines lacking a direct hardware implementation by extracting the fraction field of the representation, with a special case for `X = 0.0` and with additional steps required when `X` is denormalized.[6)]

Often, both the exponent part and the fraction part of a floating-point number are needed in argument reduction. For such occasions, the procedure `DECOMPOSE`, which computes and delivers both simultaneously through a pair of arguments of mode **out**, is provided.

The function `COMPOSE` is essentially the inverse of `DECOMPOSE`; it constructs a floating-point value from a given fraction and exponent part. Except when `FRACTION = 0.0`, `COMPOSE(FRACTION, EXPONENT)`—for arguments of the floating-point type `FLOAT_TYPE` and the integer type `EXPONENT_TYPE`, respectively—delivers the value $\texttt{FRACTION} \cdot \beta^{\texttt{EXPONENT}-k}$ (if it is representable), where k is the unique integer such that $\beta^{k-1} \leq |\texttt{FRACTION}| < \beta^k$; `COMPOSE(0.0, EXPONENT)` delivers `0.0` for any `EXPONENT`. If the defined result is not representable, then the appropriate predefined exception is raised in overflow situations, and one of the adjacent representable numbers is delivered in underflow situations. Note that the `FRACTION` argument is not required to be a pure fraction, with a zero exponent part (as if it had been obtained from the `FRACTION` function previously); rather, the fraction part of `FRACTION` is obtained and used to construct the result. It should be obvious that this function can be computed, on typical hardware, by appropriate manipulations of the fraction and exponent parts of floating-point quantities, as for the previous functions. `COMPOSE` finds representative uses in the result construction step of mathematical functions.

The remaining function of the first group, `SCALE`, is similar to `COMPOSE`; it has uses both in result construction steps and in argument conditioning (for Euclidean norms, complex arithmetic, and some matrix computations, for example). It takes arguments `X` and `ADJUSTMENT` and returns $\texttt{X} \cdot \beta^{\texttt{ADJUSTMENT}}$ (with the same provisions for dealing with overflow and underflow as exhibited by `COMPOSE`). `SCALE` is analogous to the IEEE recommended function `scalb`. When implemented by directly manipulating the exponent part of a floating-point number, it is potentially more efficient than multiplying or dividing by a power of the hardware radix, and by definition it retains full accuracy (multiplication and division, even by a power of the hardware radix, can lose accuracy on systems lacking guard digits for these operations). The function is sometimes available as a hardware operation.

The functions of the first group are not all independent. In theory, it is sufficient to have just `EXPONENT` and `SCALE`, or alternatively `EXPONENT` and `COMPOSE`; the others can be obtained in terms of these two. For greater efficiency, however, implementations should code each independently using the most direct interface to low-level representations and operations available.

The second group of subprograms comprises directed rounding functions (`ROUND`, `TRUNCATE`, `FLOOR` and `CEILING`, all of which yield an integer value in the floating-point type `FLOAT_TYPE`) and an exact remainder function (`REMAINDER`).

`ROUND`, of course, delivers the value of its argument, rounded to the nearest integer, with ties being broken by choosing the even integer; this corresponds to IEEE unbiased rounding. Ada already has something comparable in its predefined conversion between floating-point and integer types. The `ROUND` function differs in having a floating-point result type and in specifying that ties will be broken by choosing the even integer. `ROUND` and the other directed rounding functions are supposed to produce their floating-point results without going through an intermediate conversion to an integer type, which could raise an exception (often the higher-precision floating-point types can accommodate larger integer values than can be represented in the available integer type of widest range). `TRUNCATE` simply discards the fractional part, thereby rounding in the direction of zero. `FLOOR` and `CEILING` round in the negative and positive directions, respectively. All of these functions can be programmed efficiently at a low level and might even exist as hardware operations.

The `REMAINDER` function delivers the exact remainder upon dividing its first floating-point argument by its second floating-point argument. More precisely, `REMAINDER(X, Y)` finds an integer quotient q and a remainder r such that $r = \texttt{X} - \texttt{Y} \cdot q$; it delivers r. Algorithms exist for computing the result exactly, and reasonably efficiently, regardless of the relative magnitudes of the dividend and divisor; the operation is available as a hardware instruction on some machines.

There are two customary ways of defining the quotient q, which determines the corresponding remainder r. One way defines q as the integer obtained by rounding the exact value of `X/Y` towards zero. This gives r the sign of the dividend

[6)] When `X` is denormalized, the simplest strategy is to multiply `X` by a fixed power β^k of the hardware radix sufficient to normalize it, prior to extracting the fraction field.