General Decimal Arithmetic Specification

25th March 2009

Mike Cowlishaw

IBM Fellow
IBM UK Laboratories

mfc@uk.ibm.com

Version 1.70

Table of Contents

Introduction 5

Scope 7
- Objectives 7
- Inclusions 7
- Exclusions 7
- Restrictions 8

The Arithmetic Model 9
- Abstract representation of numbers 9
- Abstract representation of operations 12
- Abstract representation of context 13
- Default contexts 16

Conversions 17
- Numeric string syntax 17
- to-scientific-string – conversion to numeric string 19
- to-engineering-string – conversion to numeric string 20
- to-number – conversion from numeric string 21

Arithmetic operations 23
- abs 26
- add and subtract 26
- compare 27
- compare-signal 27
- divide 27
- divide-integer 29
- exp 29
- fused-multiply-add 30
- ln 31
- log10 31
- max 32
- max-magnitude 32
- min 32
- min-magnitude 33
- minus and plus 33
- multiply 33
- next-minus 34
- next-plus 34

next-toward 34
power 35
quantize 36
reduce 37
remainder 37
remainder-near 38
round-to-integral-exact 39
round-to-integral-value 39
square-root 39

Miscellaneous operations 41
and 41
canonical 42
class 42
compare-total 42
compare-total-magnitude 43
copy 43
copy-abs 44
copy-negate 44
copy-sign 44
invert 44
is-canonical 44
is-finite 45
is-infinite 45
is-NaN 45
is-normal 45
is-qNaN 46
is-signed 46
is-sNaN 46
is-subnormal 46
is-zero 46
logb 47
or 47
radix 47
rotate 47
same-quantum 48
scaleb 48
shift 49
xor 49

Exceptional conditions 51

Appendix A – The X3.274 subset 55

Appendix B – Design concepts 59

Appendix C – Changes 61

Index 69

Introduction

This document defines a general purpose decimal arithmetic for both limited precision floating-point (as defined by the IEEE 754 standard[1] approved in June 2008) and for arbitrary precision floating-point (following the same principles as IEEE 754 and the earlier IEEE 854-1987 standard).[2] In addition to floating-point arithmetic, integer and unrounded floating-point arithmetic are included as subsets.

The primary audience for this document is implementers, so examples and other explanatory material are included. Explanatory material is identified as Notes, Examples, or footnotes, and is not part of the formal specification.

Appendix A (see page 55) describes a simplified subset of the full arithmetic which implements the decimal floating-point arithmetic defined in the ANSI standard X3.274-1996[3] (this provides the model for the unrounded floating-point rules). Appendix B (see page 59) summarizes the design concepts behind the decimal arithmetic. Appendix C (see page 61) lists the changes to this specification.

This document in various softcopy formats, together with a reference implementation, testcases, concrete representations (encodings), and background information may be found at `http://speleotrove.com/decimal`

Comments on this draft are welcome. Please send any comments, suggestions, and corrections to the author, Mike Cowlishaw (`mfc@uk.ibm.com`).

Acknowledgements

Very many people have contributed to the arithmetic described in this document, especially the 1980 Rexx Language Committee, the IBM Rexx Architecture Review Board, the IBM Vienna Compiler group, the X3 (now NCITS) J18 technical committee, the authors of the IEEE 854 standard, and the members of the IEEE 754r (revision) committee. Special thanks for their contributions to the current design and this document are due to Aahz, Merav Aharoni, Nelson Beebe, Joshua Bloch, Dirk Bosmans, Paul-Georges Crismer, Joe Darcy, Gunnar Degnbol, Mark Dickinson, John Ehrman, Kit George, Peter Golde, Michel Hack, Brian Marks, Ilan Nehama, Dave Raggett, Fred Ris, Eric Schwarz, Ron Smith, and Phil Yeh.

---

1 IEEE 754-2008 – *IEEE Standard for Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 2008. (In press.)

2 IEEE 854-1987 – *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1987.

3 *American National Standard for Information Technology – Programming Language REXX, X3.274-1996*, American National Standards Institute, New York, 1996.

Scope

Objectives

This document defines a general purpose decimal arithmetic. A correct implementation of this specification using appropriate parameters will conform to the decimal arithmetic defined in IEEE standard 754-2008,[4] except for some minor restrictions (see page 8), and will also provide unrounded decimal arithmetic[5] and integer arithmetic as proper subsets.

Inclusions

This specification defines the following:

- Constraints on the values of decimal numbers
- Operations on decimal numbers, including
  - Required conversions between string and internal representations of numbers
  - Arithmetical operations on decimal numbers (addition, subtraction, *etc.*)
- Context information which alters the results of operation, and default contexts.
- Exceptional conditions, such as overflow, underflow, undefined results, and other exceptional situations which may occur during operations.

Exclusions

This specification does not define the following:

- Concrete representations (storage format) of decimal numbers[6]
- Concrete representations (storage format) of context information
- The means by which operations are effected

4 IEEE 754-2008 – *IEEE Standard for Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 2008. (In press.)
5 Sometimes called "fixed-point" decimal arithmetic.
6 The IEEE 754 decimal encodings for interchange formats are described in:
`http://speleotrove.com/decimal/decbits.pdf`

Restrictions

This specification deviates from the requirements of IEEE 754 in the following respects:

1. The remainder-near operator is restricted to those values where the intermediate integer can be represented in the current precision.[7]
2. The mathematical functions do not, in general, correspond to the recommended functions in IEEE 754 with the same or similar names; in particular, the *power* function has some different special cases, and most of the functions may be up to one unit wrong in the last place.
3. The *squareroot* function is only specified here for one rounding algorithm (IEEE 754 requires it to be supported for all rounding algorithms). However, it is defined to be correctly rounded.

The requirements of IEEE 854 over the use of the terms *single precision* and *double precision* are not followed in this specification because since that standard was published these terms have become synonymous with particular sizes of encodings (32-bit and 64-bit respectively).

7 This is because the conventional implementation of this operator would be unacceptably long-running for the range of numbers allowed by this specification (with up to nine digits of exponent). For restricted-range numbers, an implementation can easily be made to conform to IEEE 754 in this respect.

The Arithmetic Model

This specification is based on a model of decimal arithmetic which is a formalization of the decimal system of numeration (Algorism) as further defined and constrained by the relevant standards (IEEE 854, ANSI X3-274, and IEEE 754-2008).

There are three components to the model:

1. *numbers* – which represent the values which can be manipulated by, or be the results of, the core operations defined in this specification
2. *operations* – the core operations (such as addition, multiplication, *etc.*) which can be carried out on numbers
3. *context* – which represents the user-selectable parameters and rules which govern the results of arithmetic operations (for example, the precision to be used).

This specification defines these components in the abstract. It neither defines the way in which operations are expressed (which might vary depending on the computer language or other interface being used),[8] nor does it define the *concrete representation* (specific layout in storage, or in a processor's register, for example) of numbers or context.

The remainder of this section describes the abstract model for each component.

Abstract representation of numbers

Numbers represent the values which can be manipulated by, or be the results of, the core operations defined in this specification. Numbers may be *finite numbers* (numbers whose value can be represented exactly) or they may be *special values* (infinities and other values which are not finite numbers).

Finite numbers

Finite numbers are defined by three integer parameters:

1. *sign* – a value which must be either 0 or 1, where 1 indicates that the number is negative or is the negative zero and 0 indicates that the number is zero or positive.
2. *coefficient* – an integer which is zero or positive.

   In the abstract, there is no upper limit on the maximum size of the *coefficient*. In practice, an implementation may need to define a specific upper limit (for example, the length of the maximum coefficient supported by the concrete representation). This limit must be expressed as an integral number of decimal digits.[9]

8 Indeed, some variations of operations could be selected by using context settings outside the scope of this specification.
9 That is, the maximum value of the *coefficient* will be an integral power of ten, less one – for example,

3. *exponent* – a signed integer which indicates the power of ten by which the *coefficient* is multiplied.

   In the abstract, there is no upper limit on the absolute value of the *exponent*. In practice there may be some upper limit, E_{limit}, on the absolute value of the *exponent*.

   If the coefficient has a maximum length then it is required[10] that E_{limit} be greater than 5 × `mlength`, where `mlength` is the maximum length of the *coefficient* in decimal digits. It is recommended that E_{limit} be greater than 10 × `mlength`.

   The *adjusted exponent* is the value of the exponent of a number when that number is expressed as though in scientific notation with one digit (non-zero unless the coefficient is 0) before any decimal point. This is given by the value of the *exponent*+(`clength`-1), where `clength` is the length of the *coefficient* in decimal digits.

   When a limit to the *exponent* applies, it must result in a balanced range of positive or negative numbers,[11] taking into account the magnitude of the *coefficient*. To achieve this balanced range, the minimum and maximum values of the *adjusted exponent* (E_{min} and E_{max} respectively) must have magnitudes which differ by no more than one, so E_{min} will be $-E_{max}\pm1$. IEEE 754 further constrains this so that $E_{min} = 1-E_{max}$.

   Therefore, if the length of the *coefficient* is `clength` digits, the *exponent* may take any of the values $-E_{limit}$-(`clength`-1)+1 through E_{limit}-(`clength`-1).

   For example, if the *coefficient* had the value 123456789 (9 digits) and the *exponent* had an E_{limit} of 999 (3 digits), then the exponent could range from -1006 through +991. This would allow positive values of the number to range from 1.23456789E-998 through 1.23456789E+999.

   It is recommended that E_{max} be expressed as an integral number of decimal digits or be one of the numbers 1, 5, or 25, multiplied by an positive integral power of ten and optionally reduced by one (for example, 49 or 50).

The numerical *value* of a finite number is given by: $(-1)^{sign} \times coefficient \times 10^{exponent}$.

The *quantum* of a finite number is given by: 1×10^{exponent}. This is the value of a unit in the least significant position of the coefficient of a finite number.[12]

This abstract definition deliberately allows for multiple representations of values which are numerically equal but are visually distinct (such as 1 and 1.00). However, there is a one-to-one mapping between the abstract representation and the result of the primary conversion to string using **to-scientific-string** (see page 19) on that abstract representation. In other words, if one number has a different abstract representation to another, then the primary string conversion will also be different.

Notes:

1. Many concrete representations for finite numbers have been used successfully. Typically, the *coefficient* is represented in some form of binary coded or packed decimal, or is encoded using a base which is a higher power of ten. It may also be expressed as a binary integer. The *exponent* is typically represented by a two's complement or biased binary integer. The IEEE 754

9999999999999999999.

10 See IEEE 854 §3.1.

11 This rule, a requirement for both ANSI X3.274 and IEEE 854, constrains the number of values which would overflow or underflow when inverted (divided into 1).

12 This is slightly different from an *ulp* (unit in last position), which is defined such that ulp(x) is the difference between the two nearest bracketing representable values to x, and which if x is exactly representable and is an exact power of the base gives the "ulp below".

decimal-encoded concrete representations are described in detail at:
`http://speleotrove.com/decimal/decbits.html`

2. The one-to-one mapping between the abstract representation and the result of the primary conversion to string is required, as described above. However, no such constraint applies to a concrete representation (that is, there may be multiple concrete representations of a single abstract representation).
3. A number with a *coefficient* of 0 is permitted to have a non-zero *sign*. This *negative zero* is accepted as an operand for all operations (see IEEE 754 §3.2).

Special values

In addition to the *finite numbers*, numbers must also be able to represent one of three named *special values*:

1. *infinity* – a value representing a number whose magnitude is infinitely large (∞, see IEEE 754 §3.2 and §6.1)
2. *quiet NaN* – a value representing undefined results (“Not a Number”) which does not cause an Invalid operation condition. IEEE 754 recommends that additional diagnostic information be associated with NaNs (see IEEE 754 §3.2 and §6.2)
3. *signaling NaN* – a value representing undefined results (“Not a Number”) which will usually cause an Invalid operation condition if used in any operation defined in this specification (see IEEE 754 §3.2 and §6.2).

When a number has one of these special values, its *coefficient* and *exponent* are undefined.[13] A NaN, however, may have associated *diagnostic* information, also known as its *payload*. This is treated as though it can be encoded as a positive integer (greater than zero) which must be no larger than can be represented by the *coefficient* less one digit.[14]

All special values may have a *sign*, as for finite numbers. The *sign* of an infinity is significant (that is, it is possible to have both positive and negative infinity), and the *sign* of a NaN has no meaning, although it may be considered part of the diagnostic information.

Normal numbers, subnormal numbers and Underflow

In any context where exponents are bounded most finite numbers are *normal*. Non-zero finite numbers whose adjusted exponents are greater than or equal to E_{min} are called *normal* numbers; those non-zero numbers whose adjusted exponents are less than E_{min} are called *subnormal* numbers.[15] Like other numbers, subnormal numbers are accepted as operands for all operations, and may result from any operation. If a result is subnormal, before any rounding, then the Subnormal condition is raised.

For a subnormal result, the minimum value of the exponent becomes $E_{min} - (precision - 1)$, called E_{tiny}, where *precision* is the working precision, as described below (see page 13). The result will be rounded, if necessary, to ensure that the exponent is no smaller than E_{tiny}. If, during this rounding, the result

13 Typically, in a concrete representation, certain out-of-range values of the exponent are used to indicate the special values, and the coefficient is used to carry additional diagnostic information for quiet NaNs. In the case of the proposed IEEE 754 decimal formats, the exponent is 0, the coefficient (excluding the first digit) may hold a decimal value which is the diagnostic information, and the special value is indicated by the combination field and exponent continuation bits.

14 This restriction allows the abstract coefficient in IEEE 754 encodings to be used to hold the diagnostic information for a NaN.

15 That is, numbers whose absolute value is non-zero and is closer to zero than ten to the power of E_{min}.

becomes inexact, then the Underflow condition is raised. A subnormal result does not necessarily raise Underflow, therefore, but is always indicated by the Subnormal condition (even if, after rounding, its value is 0 or ten to the power of E_{min}).

When a number underflows to zero during a calculation, its *exponent* will be E_{tiny}. The maximum value of the *exponent* is unaffected.

Note that the minimum value of the exponent for subnormal numbers is the same as the minimum value of exponent which can arise during operations which do not result in subnormal numbers, which occurs in the case where `clength` = *precision*.

Notation

In later sections of this document, a specific finite number is described by its abstract representation, using the triad notation: [*sign*, *coefficient*, *exponent*], where each value is an integer. Only the *exponent* can be negative.

Similarly, pairs or triads are used to indicate the special values. These have the form [*sign*, *special-value*] or the form [*sign*, *special-value*, *diagnostic*], where the *sign* is indicated as before, and the *special-value* is one of `inf`, `qNaN`, or `sNaN`, representing *infinity*, *quiet NaN*, or *signaling NaN*, respectively, and *diagnostic* is a positive integer.

So, for example, the triad `[0,2708,-2]` represents the number `27.08`, the triad `[1,1953,0]` represents the integer `-1953`, the pair `[1,inf]` represents the number -∞, and the pair `[0,qNaN]` represents a quiet NaN.

Abstract representation of operations

The core operations which must be provided by an implementation are described in later sections which define Conversions (see page 17) and Arithmetic Operations (see page 23). Each operation is given an abstract name (for example, “add”), and its semantics are strictly defined. However, the implementation of each operation and the manner by which each is effected is not defined by this specification.

For example, in a object-oriented language, the addition operation might be effected by a method called `add`, whereas in a calculator application it might be effected by clicking on a button icon. In other uses, an infix “+” symbol might be used to indicate addition. And in all cases, the operation might be carried out in software, hardware, or some combination of these.

Similarly, operations which are distinct in the specification need not be mapped one-to-one to distinct operations in the implementation – it is only necessary that all the core operations are available. For example, conversions to a string could be handled by a single method, with variations determined from context or additional arguments.

Abstract representation of context

The *context* represents the user-selectable parameters and rules which govern the results of arithmetic operations (for example, the precision to be used). This context might be implied in some way, or be a global or local setting, or be passed to operations – depending on the implementation of the specification (for example, in a programming language).

The context is defined by the following parameters:

precision An integer which must be positive (greater than 0). This sets the maximum number of significant digits that can result from an arithmetic operation.

In the abstract, there is no upper bound on the precision (although a specific precision must always be provided). In practice there may need to be some upper limit to it (for example, the length of the maximum *coefficient* supported by a concrete representation). This limit must be expressed as an integral number of decimal digits.

Similarly, there may be a lower bound on the setting on precision, which may be the same as the upper bound (for example, if it is implied by the length of the maximum *coefficient* supported by a concrete representation). This limit must also be expressed as an integral number of decimal digits.

rounding A named value which indicates the algorithm to be used when rounding is necessary. Rounding is applied when a result *coefficient* has more significant digits than the value of *precision*; in this case the result coefficient is shortened to *precision* digits and may then be incremented by one (which may require a further shortening), depending on the rounding algorithm selected and the remaining digits of the original *coefficient*. The *exponent* is adjusted to compensate for any shortening.

The five following rounding algorithms are defined and must be supported: [16]

round-down (Round toward 0; truncate.) The discarded digits are ignored; the result is unchanged.

round-half-up If the discarded digits represent greater than or equal to half (0.5) of the value of a one in the next left position then the result coefficient should be incremented by 1 (rounded up). Otherwise the discarded digits are ignored.

round-half-even If the discarded digits represent greater than half (0.5) the value of a one in the next left position then the result coefficient should be incremented by 1 (rounded up). If they represent less than half, then the result coefficient is not adjusted (that is, the discarded digits are ignored).

Otherwise (they represent exactly half) the result coefficient is unaltered if its rightmost digit is even, or incremented by 1 (rounded up) if its rightmost digit is odd (to make an even digit).

round-ceiling (Round toward +∞.) If all of the discarded digits are zero or if the *sign* is 1 the result is unchanged. Otherwise, the result coefficient should be

16 The term "round to nearest" is not used because it is ambiguous. *round-half-up* is the usual round-to-nearest algorithm used in European countries, in international financial dealings, and in the USA for tax calculations. *round-half-even* is often used for other applications in the USA, where it is usually called "round to nearest" and is sometimes called "banker's rounding".

incremented by 1 (rounded up).

round-floor (Round toward $-\infty$.) If all of the discarded digits are zero or if the *sign* is 0 the result is unchanged. Otherwise, the sign is 1 and the result coefficient should be incremented by 1.

Three further rounding algorithms are defined; these are optional:

round-half-down If the discarded digits represent greater than half (0.5) of the value of a one in the next left position then the result coefficient should be incremented by 1 (rounded up). Otherwise (the discarded digits are 0.5 or less) the discarded digits are ignored.

round-up (Round away from 0.) If all of the discarded digits are zero the result is unchanged. Otherwise, the result coefficient should be incremented by 1 (rounded up).

round-05up (Round zero or five away from 0.) The same as *round-up*, except that rounding up only occurs if the digit to be rounded up is 0 or 5, and after overflow the result is the same as for *round-down*.[17]

When a result is rounded, the *coefficient* may become longer than the current *precision*. In this case the least significant digit of the coefficient (it will be a zero) is removed (reducing the precision by one), and the *exponent* is incremented by one. This in turn may give rise to an overflow condition (see page 53), which determines the result after overflow.

17 The rounding mode *round-05up* permits arithmetic at shorter lengths to be emulated in a fixed-precision environment without double rounding. For example, a multiplication at a precision of 9 can be effected by carrying out the multiplication at (say) 16 digits using *round-05up* and then rounding to the required length using the desired rounding algorithm.

flags and trap-enablers The exceptional conditions (see page 51) are grouped into *signals*, which can be controlled individually. The context contains a *flag* (which is either 0 or 1) and a *trap-enabler* (which also is either 0 or 1) for each signal.

For each of the signals, the corresponding flag is set to 1 when the signal occurs. It is only reset to 0 by explicit user action.

For each of the signals, the corresponding trap-enabler indicates which action is to be taken when the signal occurs (see IEEE 754 §7). If 0, a defined result is supplied, and execution continues (for example, an overflow is perhaps converted to a positive or negative infinity). If 1, then execution of the operation is ended or paused and control passes to a "trap handler", which will have access to the defined result.

The signals are:

clamped raised when the exponent of a result has been altered or constrained in order to fit the constraints of a specific concrete representation

division-by-zero raised when a non-zero dividend is divided by zero

inexact raised when a result is not exact (one or more non-zero coefficient digits were discarded during rounding)

invalid-operation raised when a result would be undefined or impossible

overflow raised when the exponent of a result is too large to be represented

rounded raised when a result has been rounded (that is, some zero or non-zero coefficient digits were discarded)

subnormal raised when a result is subnormal (its adjusted exponent is less than E_{min}), before any rounding

underflow raised when a result is both subnormal and inexact.

This specification does not define the means by which flags and traps are reset or altered, respectively, or the means by which traps are effected.[18]

The context might also specify further variables, such as E_{max} where a variable exponent bound is required.

Notes:

1. The setting of *precision* may be used to reduce a result to a narrower precision, using the **plus** operation.
2. IEEE 854 and IEEE 754 were designed under the assumption that some small number of known precisions would be available to users. This specification extends this concept to allow (but not require) variable precisions, as specified by ANSI X3.274. This generalization allows improved interoperation between software arbitrary-precision decimal packages and hardware implementations (which are expected to have relatively low maximum precision limits, typically just tens of digits).
3. *precision* can be set to positive values lower than nine. Small values, however, should be used with care – the loss of precision and rounding thus requested will affect all computations affected by the context, including comparisons. To conform to IEEE 854, this value should not

18 IEEE 754 suggests that there be a mechanism allowing traps to return a substitute result to the operation that raised the exception, but this may not be possible in some environments (including some object-oriented environments).

be set less than 6; the smallest IEEE 754 interchange format supports 7.

4. The concrete representation of *rounding* is often a series of integer constants, or an enumeration, held in an object or control register.
5. It has been proposed that each exceptional condition should have its own, distinct, signal and trap-enabler. This specification may change to this approach.

Default contexts

This specification defines optional *default contexts*, which define suitable settings for basic arithmetic and for the extended arithmetic defined by IEEE 854 and IEEE 754. It is recommended that the default contexts be easily selectable by the user.

Basic default context

In the *basic default context*, the parameters are set as follows:

- *flags* – all set to 0
- *trap-enablers* – *inexact*, *rounded*, and *subnormal* are set to 0; all others are set to 1 (that is, the other conditions are treated as errors)
- *precision* – is set to 9
- *rounding* – is set to *round-half-up*

Extended default contexts

In the *extended default contexts*, the parameters are set as follows:

- *flags* – all set to 0
- *trap-enablers* – all set to 0 (IEEE 854 §7)
- *precision* – is set to the appropriate precision for a given numerical format (for the IEEE 754 smallest and basic formats, the precisions are 7, 16, or 34 digits).
- *rounding* – is set to *round-half-even* (IEEE 754 §4.3.3)

Conversions

This section defines the required conversions between the abstract representation of numbers and string (character) form.[19] Two number-to-string conversions and one string-to-number conversion are defined.

It is strongly recommended that implementations also provide conversions to and from a Binary Coded Decimal (BCD) representation, as appropriate for the implementation platform. Most decimal data are encoded in some form of BCD, and also a BCD form (especially one digit per byte) is easy to manipulate for further processing, formatting, *etc.*

It is recommended that implementations also provide conversions to and from binary floating-point or integer numbers, if appropriate (that is, if such encodings are supported in the environment of the implementation). It is suggested that such conversions be exact, if possible (that is, when converting from binary to decimal), or alternatively give the same results as converting using an appropriate string representation as an intermediate form. It is also recommended that if a number is too large to be converted to a given binary integer format then an exceptional or error condition be raised, rather than losing high-order significant bits (decapitating).

It is recommended that implementations also provide additional number formatting routines (including some which are locale-dependent), and if available should accept non-European decimal digits in strings.

Notes:

1. The setting of *precision* may be used to convert a number from any precision to any other precision, using the **plus** operation.
2. Integers are a proper subset of numbers, hence no conversion operation from an integer to a number is necessary. Conversion from a number to an integer with an exponent of zero is effected by using the **quantize** operation (see page 36). Conversion from a number to an integral value (with a non-negative exponent) is effected by using the **round-to-integral-value** operation (see page 39) or the **round-to-integral-exact** operation (see page 39). These meet the requirements of IEEE 754 §5.3.1 and §5.3.2.

Numeric string syntax

Strings which are acceptable for conversion to the abstract representation of numbers, or which might result from conversion from the abstract representation to a string, are called *numeric strings*.

A *numeric string* is a character string that describes either a *finite number* or a *special value*.

- If it describes a *finite number*, it includes one or more decimal digits, with an optional decimal

19 See also IEEE 754 §5.12.

point. The decimal point may be embedded in the digits, or may be prefixed or suffixed to them. The group of digits (and optional point) thus constructed may have an optional sign (“`+`” or “`-`”) which must come before any digits or decimal point.

The string thus described may optionally be followed by an “`E`” (indicating an exponential part), an optional sign, and an integer following the sign that represents a power of ten that is to be applied. The “`E`” may be in uppercase or lowercase.

- If it describes a *special value*, it is one of the case-independent names “`Infinity`”, “`Inf`”, “`NaN`”, or “`sNaN`” (where the first two represent *infinity* and the second two represent *quiet NaN* and *signaling NaN* respectively). The name may be preceded by an optional sign, as for finite numbers. If a NaN, the name may also be followed by one or more digits, which encode any diagnostic information.

No blanks or other white space characters are permitted in a numeric string.

Formally:[20]

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' |
                    '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

where the characters in the strings accepted for `infinity` and `nan` may be in any case.

If an implementation supports the concept of diagnostic information on NaNs, the numeric strings for NaNs may include one or more digits, as shown above.[21] These digits encode the diagnostic information in an implementation-defined manner; however, conversions to and from string for diagnostic NaNs should be reversible if possible. If an implementation does not support diagnostic information on NaNs, these digits should be ignored where necessary. A plain “`NaN`” is usually the same as “`NaN0`”.

Examples:

Some numeric strings are:

```
  "0"         -- zero
  "12"         -- a whole number
 "-76"         -- a signed whole number
  "12.70"      -- some decimal places
  "+0.003"     -- a plus sign is allowed, too
 "017."        -- the same as 17
    ".5"       -- the same as 0.5
  "4E+9"       -- exponential notation
   "0.73e-7"   -- exponential notation, negative power
  "Inf"        -- the same as Infinity
  "-infinity"  -- the same as -Inf
  "NaN"        -- not-a-Number
  "NaN8275"    -- diagnostic NaN
```

20 Where quotes surround terminal characters, “`::=`” means “is defined as”, “`|`” means “or”, “`[]`” encloses an optional item, and “`[]...`” encloses an item which is repeated 0 or more times.

21 In IEEE 754 interchange formats, the diagnostic information (payload) is held in a similar manner to the coefficient of a finite number in the same format, but has one digit fewer.

Notes:

1. A single period alone or with a sign is not a valid numeric string.
2. A sign alone is not a valid numeric string.
3. Significant (after the decimal point) and insignificant leading zeros are permitted.

to-scientific-string – conversion to numeric string

This operation converts a number to a string, using scientific notation if an exponent is needed. The operation is not affected by the *context*.

If the number is a *finite number* then:

- The *coefficient* is first converted to a string in base ten using the characters 0 through 9 with no leading zeros (except if its value is zero, in which case a single 0 character is used).

  Next, the *adjusted exponent* is calculated; this is the *exponent*, plus the number of characters in the converted *coefficient*, less one. That is, *exponent*+(`clength-1`), where `clength` is the length of the *coefficient* in decimal digits.

  If the *exponent* is less than or equal to zero and the *adjusted exponent* is greater than or equal to `-`6, the number will be converted to a character form without using exponential notation. In this case, if the *exponent* is zero then no decimal point is added. Otherwise (the *exponent* will be negative), a decimal point will be inserted with the absolute value of the *exponent* specifying the number of characters to the right of the decimal point. "0" characters are added to the left of the converted *coefficient* as necessary. If no character precedes the decimal point after this insertion then a conventional "0" character is prefixed.

  Otherwise (that is, if the *exponent* is positive, or the *adjusted exponent* is less than `-6`), the number will be converted to a character form using exponential notation. In this case, if the converted *coefficient* has more than one digit a decimal point is inserted after the first digit. An exponent in character form is then suffixed to the converted *coefficient* (perhaps with inserted decimal point); this comprises the letter "`E`" followed immediately by the *adjusted exponent* converted to a character form. The latter is in base ten, using the characters 0 through 9 with no leading zeros, always prefixed by a sign character ("`-`" if the calculated exponent is negative, "`+`" otherwise).

Otherwise (the number is a *special value*):

- If the *special value* is *quiet NaN* then the resulting string is "`NaN`", optionally followed by one or more digits representing diagnostic information. The digits will have no leading zeros.
- If the *special value* is *signaling NaN* then the resulting string is "`sNaN`",[22] optionally followed by one or more digits representing diagnostic information, as for a quiet NaN.
- If the *special value* is *infinity* then the resulting string is "`Infinity`".

In all cases, the entire string is prefixed by a minus sign character[23] ("`-`") if *sign* is 1. No sign character is prefixed if *sign* is 0.

Examples:

For each abstract representation [*sign*, *coefficient*, *exponent*], [*sign*, *special-value*], or [*sign*, *special-value*,

22 This is a deviation from IEEE 854-1987 (see Notes) but is permitted by IEEE 754-2008.

23 This specification defines only the glyph representing a minus sign character. Depending on the implementation, this will often correspond to a hyphen rather than to a distinguishable "minus" character.

diagnostic] on the left, the resulting string is shown on the right.

```
[0,123,0]        "123"
[1,123,0]        "-123"
[0,123,1]        "1.23E+3"
[0,123,3]        "1.23E+5"
[0,123,-1]       "12.3"
[0,123,-5]       "0.00123"
[0,123,-10]      "1.23E-8"
[1,123,-12]      "-1.23E-10"
[0,0,0]          "0"
[0,0,-2]         "0.00"
[0,0,2]          "0E+2"
[1,0,0]          "-0"
[0,5,-6]         "0.000005"
[0,50,-7]        "0.0000050"
[0,5,-7]         "5E-7"
[0,inf]          "Infinity"
[1,inf]          "-Infinity"
[0,qNaN]         "NaN"
[0,qNaN,123]     "NaN123"
[1,sNaN]         "-sNaN"
```

Notes:

1. There is a one-to-one mapping between abstract representations and the result of this conversion. That is, every abstract representation has a unique **to-scientific-string** representation. Also, if that string representation is converted back to an abstract representation using **to-number** (see page 21) with sufficient precision, then the original abstract representation will be recovered.

   This one-to-one mapping guarantees that there is no hidden information in the internal representation of the numbers ("what you see is exactly what you've got").

2. The values *quiet NaN* and *signaling NaN* are distinguished in string form in order to preserve the one-to-one mapping just described. The strings chosen are those suggested by the IEEE 754 review committee and permitted by IEEE 754-2008.

3. The digits required for an exponent may be more than the number of digits required for E_{max} when a finite number is subnormal (see page 11).

to-engineering-string – conversion to numeric string

This operation converts a number to a string, using engineering notation if an exponent is needed.

The conversion exactly follows the rules for conversion to scientific numeric string except in the case of finite numbers where exponential notation is used. In this case,

- if the number is *non-zero*, the converted exponent is adjusted to be a multiple of three (engineering notation) by positioning the decimal point with one, two, or three characters preceding it (that is, the part before the decimal point will range from `1` through `999`); this may require the addition of either one or two trailing zeros (otherwise, if after the adjustment the decimal point would not be followed by a digit then it is not added)

- if the number is a *zero*, the zero will have a decimal point and one or two trailing zeros added, if necessary, so that the original exponent of the zero would be recovered by the **to-number** conversion.

If the final exponent is zero then no indicator letter and exponent is suffixed.

Examples:

For each abstract representation [*sign*, *coefficient*, *exponent*] on the left, the resulting string is shown on the right.

```
[0,123,1]        "1.23E+3"
[0,123,3]        "123E+3"
[0,123,-10]      "12.3E-9"
[1,123,-12]      "-123E-12"
[0,7,-7]         "700E-9"
[0,7,1]          "70"
[0,0,1]          "0.00E+3"
```

to-number – conversion from numeric string

This operation converts a string to a number, as defined by its abstract representation. The string is expected to conform to the numeric string syntax (see page 17).

Specifically, if the string represents a *finite number* then:

- If it has a leading sign, then the *sign* in the resulting abstract representation is set appropriately (1 for "-", 0 for "+"). Otherwise the *sign* is set to 0.

  The decimal-part and exponent-part (if any) are then extracted from the string and the exponent-part (following the indicator) is converted to form the integer *exponent* which will be negative if the exponent-part began with a "-" sign. If there is no exponent-part, the *exponent* is set to 0.

  If the decimal-part included a decimal point the *exponent* is then reduced by the count of digits following the decimal point (which may be zero) and the decimal point is removed. The remaining string of digits has any leading zeros removed (except for the rightmost digit) and is then converted to form the *coefficient* which will be zero or positive.

  A numeric string to finite number conversion is always exact unless there is an underflow or overflow (see below) or the number of digits in the decimal-part of the string is greater than the *precision* in the context. In this latter case the coefficient will be rounded (shortened) to exactly *precision* digits, using the *rounding* algorithm, and the *exponent* is increased by the number of digits removed. The *rounded* and other flags may be set, as if an arithmetic operation had taken place (see below).

  If the value of the *adjusted exponent* (see page 9) is less than E_{min}, then the number is subnormal (see page 11). In this case, the calculated coefficient and exponent form the result, unless the value of the *exponent* is less than E_{tiny}, in which case the *exponent* will be set to E_{tiny} (see page 11), and the coefficient will be rounded (possibly to zero) to match the adjustment of the exponent, with the *sign* remaining as set above. If this rounding gives an inexact result then the Underflow exceptional condition (see page 53) is raised.

  If (after any rounding of the coefficient) the value of the *adjusted exponent* is larger than E_{max} (see page 9), then an exceptional condition (overflow) results. In this case, the result is as defined under the Overflow exceptional condition (see page 53), and may be infinite. It will have the *sign* as set above.

If the string represents a *special value* then:

- For all special values, the *sign* of the number is set to 1 if the string has a leading "-". Otherwise (there is a leading "+", or no leading sign) the *sign* is set to 0.

- The strings "`Infinity`" and "`Inf`", independent of case, will be converted to *infinity*.
- The string "`NaN`", independent of case, is converted to *quiet NaN*. If any digits follow the string "`NaN`", any leading zeros are removed and the digits are then converted to form the diagnostic information for the NaN in a system-dependent way. If the implementation does not support diagnostic information on NaNs the digits should be ignored.
- The string "`sNaN`", independent of case, is converted to *signaling NaN*. If any digits follow the string "`sNaN`", they are treated in the same way as for quiet NaNs.

The result of attempting to convert a string which does not have the syntax of a *numeric string* is `[0,qNaN]`.

Examples:

For each string on the left, the resulting abstract representation [*sign*, *coefficient*, *exponent*], [*sign*, *special-value*], or [*sign*, *special-value*, *diagnostic*] is shown on the right. *precision* is at least 3.

```
"0"            [0,0,0]
"0.00"         [0,0,-2]
"123"          [0,123,0]
"-123"         [1,123,0]
"1.23E3"       [0,123,1]
"1.23E+3"      [0,123,1]
"12.3E+7"      [0,123,6]
"12.0"         [0,120,-1]
"12.3"         [0,123,-1]
"0.00123"      [0,123,-5]
"-1.23E-12"    [1,123,-14]
"1234.5E-4"    [0,12345,-5]
"-0"           [1,0,0]
"-0.00"        [1,0,-2]
"0E+7"         [0,0,7]
"-0E-7"        [1,0,-7]
"inf"          [0,inf]
"+inFiniTy"    [0,inf]
"-Infinity"    [1,inf]
"NaN"          [0,qNaN]
"-NAN"         [1,qNaN]
"SNaN"         [0,sNaN]
"Fred"         [0,qNaN]
```

Arithmetic operations

This section describes the arithmetic operations on, and some other functions of, numbers, including subnormal numbers, negative zeros, and special values (see also IEEE 754 §5 and §6).

Arithmetic operation notation

In this section, a simplified notation is used to illustrate arithmetic operations: a number is shown as the string that would result from using the **to-scientific-string** operation. Single quotes are used to indicate that a number converted from an abstract representation is implied.

Also, operations are indicated as functions (taking up to three operands), and the sequence ==> means “results in”. Hence:

```
add('12', '7.00') ==> '19.00'
```

means that the result of the **add** operation with the operands `[0,12,0]` and `[0,700,-2]` is `[0,1900,-2]`.

Finally, in this example and in the examples below, the context is assumed to have *precision* set to 9, *rounding* set to *round-half-up*, and all *trap-enablers* set to 0.

Arithmetic operation rules

The following general rules apply to all arithmetic operations except where stated below.

- Every operation on finite numbers is carried out (as described under the individual operations below) as though an exact mathematical result is computed, using integer arithmetic on the coefficient where possible.

  If the coefficient of the theoretical exact result has no more than *precision* digits, then (unless there is an underflow or overflow) it is used for the result without change. Otherwise (it has more than *precision* digits) it is rounded (shortened) to exactly *precision* digits, using the current *rounding* algorithm, and the *exponent* is increased by the number of digits removed.

  If the value of the *adjusted exponent* (see page 9) of the result is less than E_{min} (that is, the result is zero or subnormal), the calculated coefficient and exponent form the result, unless the value of the *exponent* is less than E_{tiny}, in which case the *exponent* will be set to E_{tiny}, the coefficient will be rounded (if necessary, and possibly to zero) to match the adjustment of the exponent, and the *sign* is unchanged.

  If the result (before rounding) was non-zero and subnormal then the Subnormal exceptional condition (see page 53) is raised. If rounding of a subnormal result leads to an inexact result then the Underflow exceptional condition (see page 53) is also raised.

If the value of the *adjusted exponent* of a non-zero result is larger than E_{max} (see page 9), then an exceptional condition (overflow) results. In this case, the result is as defined under the Overflow exceptional condition (see page 53), and may be infinite. It will have the same sign as the theoretical result.[24]

- Arithmetic using the special value *infinity* follows the usual rules, where `[1,inf]` is less than every finite number and `[0,inf]` is greater than every finite number. Under these rules, an infinite result is always exact. Certain uses of infinity raise an Invalid operation condition (see page 52).
- *signaling NaNs* always raise the Invalid operation condition when used as an operand to an arithmetic operation.
- The Invalid operation condition may also be raised when an operand to an operation is invalid (for example, if it exceeds the bounds that an implementation can handle, or the operation is a logarithm and the operand is negative).
- The result of any arithmetic operation which has an operand which is a NaN (a *quiet NaN* or a *signaling NaN*) is `[s,qNaN]` or `[s,qNaN,d]`. The sign and any diagnostic information is copied from the first operand which is a signaling NaN, or if neither is signaling then from the first operand which is a NaN. Whenever a result is a NaN, the sign of the result depends only on the copied operand (the following rules do not apply).
- The *sign* of the result of a multiplication or division will be 1 only if the operands have different signs.
- The *sign* of the result of an addition or subtraction will be 1 only if the result is less than zero, except for the special cases below where the result is a negative 0.
- A result which is a negative zero (`[1,0,`*n*`]`) can occur only when:
  - a result is rounded to zero, and the value before rounding had a *sign* of 1.
  - the operation is an addition of `[1,0,`*n*`]` to `[1,0,`*n*`]`, or a subtraction of `[0,0,`*n*`]` from `[1,0,`*n*`]`
  - the operation is an addition of operands with opposite signs (or is a subtraction of operands with the same sign), the result has a *coefficient* of 0, and the *rounding* is *round-floor*.
  - the operation is a multiplication or division and the result has a *coefficient* of 0 and the signs of the operands are different.
  - the operation is **power**, the left-hand operand is `[1,0,`*n*`]`, and the right-hand operand is positive, integral, and odd.
  - the operation is **power**, the left-hand operand is `[1,inf]`, and the right-hand operand is negative, integral, and odd.

---

24 In practice, it is only necessary to work with intermediate results of up to twice the current precision. Some rounding settings may require some inspection of possible remainders or additional digits (for example, to determine whether a result is exactly 0.5 in the next position), though their actual values would not be required.

For *round-half-up*, rounding can be effected by truncating the result to *precision* (and adding the count of truncated digits to the *exponent*). The first truncated digit is then inspected, and if it has the value 5 through 9 the result is incremented by 1. This could cause the result to again exceed *precision* digits, in which case it is divided by 10 and the *exponent* is incremented by 1.

- the operation is **quantize** or a **round-to-integral**, the left-hand operand is negative, and the magnitude of the result is zero. In either case the final exponent may be non-zero.
- the operation is **square-root** and the operand is `[1,0,`*n*`]`.
- the operation is one of the operations **max**, **max-magnitude**, **min**, **min-magnitude**, **next-plus**, **next-toward**, **reduce**, or is a copy operation.

Examples involving special values:

```
add('Infinity', '1')         ==>  'Infinity'
add('NaN', '1')              ==>  'NaN'
add('NaN', 'Infinity')       ==>  'NaN'
subtract('1', 'Infinity')    ==>  '-Infinity'
multiply('-1', 'Infinity')   ==>  '-Infinity'
subtract('-0', '0')          ==>  '-0'
multiply('-1', '0')          ==>  '-0'
divide('1', '0')             ==>  'Infinity'
divide('1', '-0')            ==>  '-Infinity'
divide('-1', '0')            ==>  '-Infinity'
```

Notes:

1. Operands may have more than *precision* digits and are not rounded before use.
2. The *context* (precision and rounding, *etc.*) for an operation might be wholly implied, or be a global or local setting, or be passed to operations individually – depending on the implementation of the specification (for example, in a programming language).
3. NaNs propagate any associated diagnostic information as described in IEEE 854 §6.2. The meaning of any such diagnostic information is outside the scope of this specification, but typically indicates the origin of the NaN. In IEEE 754-2008, this information is only held in the coefficient of decimal numbers and does not use the first digit of the coefficient.
4. The rules above imply that the **compare** operation can return a quiet NaN as a result, which indicates an “unordered” comparison (see IEEE 754 §5.11).
5. An implementation may use the **compare** operation “under the covers” to implement a closed set of comparison operations (greater than, equal, *etc.*) if desired. In this case, the additional constraints detailed in IEEE 754 §5.11 will apply; that is, a comparison (such a “greater than”) which does not explicitly allow for an “unordered” result yet would require an unordered result will give rise to an Invalid operation condition (see page 52).
6. If a result is rounded, remains finite, and is not subnormal, its coefficient will have exactly *precision* digits (except after the **quantize** or **round-to-integral** operations, as described below). That is, only unrounded or subnormal coefficients can have fewer than *precision* digits.
7. Trailing zeros are not removed after operations. The **reduce** operation may be used to remove trailing zeros if desired.

abs

abs takes one operand. If the operand is negative, the result is the same as using the **minus** operation (see page 33) on the operand. Otherwise, the result is the same as using the **plus** operation (see page 33) on the operand.

Examples:

```
abs('2.1')    ==>  '2.1'
abs('-100')   ==>  '100'
abs('101.5')  ==>  '101.5'
abs('-101.5') ==>  '101.5'
```

Note that the result of this operation is affected by context and may set *flags*. The **copy-abs** operation (see page 44) may be used if this is not desired.

add and subtract

add and **subtract** both take two operands. If either operand is a *special value* then the general rules apply.

Otherwise, the operands are added (after inverting the *sign* used for the second operand if the operation is a subtraction), as follows:

- The *coefficient* of the result is computed by adding or subtracting the aligned coefficients of the two operands. The aligned coefficients are computed by comparing the exponents of the operands:
  - If they have the same exponent, the aligned coefficients are the same as the original coefficients.
  - Otherwise the aligned coefficient of the number with the larger exponent is its original coefficient multiplied by 10^n, where n is the absolute difference between the exponents, and the aligned coefficient of the other operand is the same as its original coefficient.

  If the signs of the operands differ then the smaller aligned coefficient is subtracted from the larger; otherwise they are added.
- The *exponent* of the result is the minimum of the exponents of the two operands.
- The *sign* of the result is determined as follows:
  - If the result is non-zero then the sign of the result is the sign of the operand having the larger absolute value.
  - Otherwise, the *sign* of a zero result is 0 unless either both operands were negative or the signs of the operands were different and the *rounding* is *round-floor*.

The result is then rounded to *precision* digits if necessary, counting from the most significant digit of the result.

Examples:

```
add('12', '7.00')         ==>  '19.00'
add('1E+2', '1E+4')       ==>  '1.01E+4'
subtract('1.3', '1.07')   ==>  '0.23'
subtract('1.3', '1.30')   ==>  '0.00'
subtract('1.3', '2.07')   ==>  '-0.77'
```

compare

compare takes two operands and compares their values numerically. If either operand is a *special value* then the general rules apply. No flags are set unless an operand is a signaling NaN.

Otherwise, the operands are compared as follows.

If the signs of the operands differ, a value representing each operand (`'-1'` if the operand is less than zero, `'0'` if the operand is zero or negative zero, or `'1'` if the operand is greater than zero) is used in place of that operand for the comparison instead of the actual operand.[25]

The comparison is then effected by subtracting the second operand from the first and then returning a value according to the result of the subtraction: `'-1'` if the result is less than zero, `'0'` if the result is zero or negative zero, or `'1'` if the result is greater than zero.

An implementation may use this operation "under the covers" to implement a closed set of comparison operations (greater than, equal, *etc.*) if desired. It need not, in this case, expose the **compare** operation itself.

Examples:

```
compare('2.1', '3')      ==>  '-1'
compare('2.1', '2.1')    ==>  '0'
compare('2.1', '2.10')   ==>  '0'
compare('3', '2.1')      ==>  '1'
compare('2.1', '-3')     ==>  '1'
compare('-3', '2.1')     ==>  '-1'
```

Notes:

1. The result of **compare** is always exact and unrounded, and may be a NaN.
2. The **compare-total** operation (see page 42) can be used for a non-numerical comparison which provides a total ordering over the abstract representation of values.

compare-signal

compare-signal takes two operands and compares their values numerically. This operation is identical to **compare**, except that if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN. (That is, all NaNs signal, with signaling NaNs taking precedence over quiet NaNs.)

divide

divide takes two operands. If either operand is a *special value* then the general rules apply.

Otherwise, if the divisor is zero then either the Division undefined condition is raised (if the dividend is zero) and the result is NaN, or the Division by zero condition is raised and the result is an Infinity with a sign which is the *exclusive or* of the signs of the operands.

Otherwise, a "long division" is effected, as follows:

- An integer variable, `adjust`, is initialized to 0.
- If the dividend is non-zero, the *coefficient* of the result is computed as follows (using working copies of the operand coefficients, as necessary):

25 This rule removes the possibility of an arithmetic overflow during a numeric comparison.

1. The operand coefficients are adjusted so that the coefficient of the dividend is greater than or equal to the coefficient of the divisor and is also less than ten times the coefficient of the divisor, thus:
   - While the coefficient of the dividend is less than the coefficient of the divisor it is multiplied by 10 and `adjust` is incremented by 1.
   - While the coefficient of the dividend is greater than or equal to ten times the coefficient of the divisor the coefficient of the divisor is multiplied by 10 and `adjust` is decremented by 1.
2. The result coefficient is initialized to 0.
3. The following steps are then repeated until the division is complete:
   - While the coefficient of the divisor is smaller than or equal to the coefficient of the dividend the former is subtracted from the latter and the coefficient of the result is incremented by 1.
   - If the coefficient of the dividend is now 0 and `adjust` is greater than or equal to 0, or if the coefficient of the result has *precision* digits, the division is complete.

     Otherwise, the coefficients of the result and the dividend are multiplied by 10 and `adjust` is incremented by 1.
4. Any remainder (the final coefficient of the dividend) is recorded and taken into account for rounding.[26]

   Otherwise (the dividend is zero), the coefficient of the result is zero and `adjust` is unchanged (is 0).

- The *exponent* of the result is computed by subtracting the sum of the original exponent of the divisor and the value of `adjust` at the end of the coefficient calculation from the original exponent of the dividend.
- The *sign* of the result is the *exclusive or* of the signs of the operands.

The result is then rounded to *precision* digits, if necessary, according to the *rounding* algorithm and taking into account the remainder from the division.

Examples:

```
divide('1', '3'  )      ==>  '0.333333333'
divide('2', '3'  )      ==>  '0.666666667'
divide('5', '2'  )      ==>  '2.5'
divide('1', '10' )      ==>  '0.1'
divide('12', '12')      ==>  '1'
divide('8.00', '2')     ==>  '4.00'
divide('2.400', '2.0')  ==>  '1.20'
divide('1000', '100')   ==>  '10'
divide('1000', '1')     ==>  '1000'
divide('2.40E+6', '2')  ==>  '1.20E+6'
```

Note that the results as described above can alternatively be expressed as follows:

- The ideal (simplest) *exponent* for the result of a division is the exponent of the dividend less the exponent of the divisor.

26 In practice, only two bits need to be noted, indicating whether the remainder was 0, or was exactly half of the final coefficient of the divisor, or was in one of the two ranges above or below the half-way point.

- After the division, if the result is exact then the coefficient and exponent giving the correct value and with the exponent closest to the ideal exponent is returned. If the result is inexact, the coefficient will have exactly *precision* digits (unless the result is subnormal), and the exponent will be set appropriately.

divide-integer

divide-integer takes two operands; it divides two numbers and returns the integer part of the result. If either operand is a *special value* then the general rules apply.

Otherwise, the result returned is defined to be that which would result from repeatedly subtracting the divisor from the dividend while the dividend is larger than or equal to the divisor. During this subtraction, the absolute values of both the dividend and the divisor are used: the sign of the final result is the same as that which would result if normal division were used.

In other words, if the operands x and y were given to the **divide-integer** and **remainder** operations, resulting in i and r respectively, then the identity

$$x = i \times y + r$$

holds.

The *exponent* of the result must be 0. Hence, if the result cannot be expressed exactly within *precision* digits, the operation is in error and will fail – that is, the result cannot have more digits than the value of *precision* in effect for the operation, and will not be rounded. For example, `divide-integer('10000000000', '3')` requires ten digits to express the result exactly (`'3333333333'`) and would therefore fail if *precision* were in the range `1` through `9`.

Notes:

1. The divide-integer operation may not give the same result as truncating normal division (which could be affected by rounding and might be Inexact).
2. The divide-integer and remainder operations are defined so that they may be calculated as a by-product of the standard division operation (described above). The division process is ended as soon as the integer result is available; the residue of the dividend is the remainder.
3. The divide and divide-integer operation on the same operands give results of the same numerical value if no error occurs and there is no residue from the divide-integer operation.

Examples:

```
divide-integer('2', '3')    ==>  '0'
divide-integer('10', '3')   ==>  '3'
divide-integer('1', '0.3')  ==>  '3'
```

exp

exp takes one operand. If the operand is a NaN then the general rules for special values apply.

Otherwise, the result is *e* raised to the power of the operand, with the following cases:

- If the operand is -Infinity, the result is 0 and exact.
- If the operand is a zero, the result is 1 and exact.
- If the operand is +Infinity, the result is +Infinity and exact.

- Otherwise the result is inexact and will be rounded using the *round-half-even* algorithm. The coefficient will have exactly *precision* digits (unless the result is subnormal). These inexact results should be correctly rounded, but may be up to 1 ulp (unit in last place) in error.

Examples:

```
exp('-Infinity')    ==> '0'
exp('-1')           ==> '0.367879441'
exp('0')            ==> '1'
exp('1')            ==> '2.71828183'
exp('0.693147181')  ==> '2.00000000'
exp('+Infinity')    ==> 'Infinity'
```

Notes:

1. The *rounding* setting in the context is not used; this means that the algorithm described in *Variable Precision Exponential Function* by T. E. Hull and A. Abrham (ACM Transactions on Mathematical Software, Vol 12 #2, pp79–91, ACM, June 1986) may be used for this operation.
2. When the result is inexact, the cost of **exp** at precision d is likely to be at least $13\times\log_2(d)$ times the cost of an inexact multiplication at the same precision (see *Multiple-precision zero-finding methods and the complexity of elementary function evaluation* by R. P. Brent, in Analytic Computational Complexity pp151–176, Academic Press, York, 1976, and *Fast Multiple-Precision Evaluation of Elementary Functions* by the same author, in Journal of the ACM (JACM), Vol 23 # 2, pp242–251, ACM, April 1976).

fused-multiply-add

fused-multiply-add takes three operands; the first two are multiplied together, using **multiply**, with sufficient precision and exponent range that the result is exact and unrounded.[27] No *flags* are set by the multiplication unless one of the first two operands is a signaling NaN or one is a zero and the other is an infinity.

Unless the multiplication failed, the third operand is then added to the result of that multiplication, using **add**, under the current context.

In other words, `fused-multiply-add(x, y, z)` delivers a result which is `(x × y) + z` with only the one, final, rounding.

Examples:

```
fused-multiply-add('3', '5', '7')                ==>  '22'
fused-multiply-add('3', '-5', '7')               ==>  '-8'
fused-multiply-add('888565290', '1557.96930',
                                '-86087.7578')  ==>  '1.38435736E+12'
```

Note that the last example would have given the result `'1.38435735E+12'` if the operation had been carried out as a separate **multiply** followed by an **add**.

27 This requires up to twice the current exponent range and a precision which is the sum of the lengths of the two operands' coefficients.

ln

ln takes one operand. If the operand is a NaN then the general rules for special values apply.

Otherwise, the operand must be a zero or positive, and the result is the natural (base *e*) logarithm of the operand, with the following cases:

- If the operand is a zero, the result is -Infinity and exact.
- If the operand is +Infinity, the result is +Infinity and exact.
- If the operand equals one, the result is 0 and exact.
- Otherwise the result is inexact and will be rounded using the *round-half-even* algorithm. The coefficient will have exactly *precision* digits (unless the result is subnormal). These inexact results should be correctly rounded, but may be up to 1 ulp (unit in last place) in error.

Examples:

```
ln('0')            ==> '-Infinity'
ln('1.000')        ==> '0'
ln('2.71828183')   ==> '1.00000000'
ln('10')           ==> '2.30258509'
ln('+Infinity')    ==> 'Infinity'
```

Notes:

1. The *rounding* setting in the context is not used.
2. When the result is inexact, the cost of **ln** at a given precision is likely to be similar to, or more expensive than, the **exp** function (see notes under that function).

log10

log10 takes one operand. If the operand is a NaN then the general rules for special values apply.

Otherwise, the operand must be a zero or positive, and the result is the base 10 logarithm of the operand, with the following cases:

- If the operand is a zero, the result is -Infinity and exact.
- If the operand is +Infinity, the result is +Infinity and exact.
- If the operand equals an integral power of ten (including 10^0 and negative powers) and there is sufficient *precision* to hold the integral part of the result, the result is an integer (with an exponent of 0) and exact.
- Otherwise the result is inexact and will be rounded using the *round-half-even* algorithm. The coefficient will have exactly *precision* digits (unless the result is subnormal). These inexact results should be correctly rounded, but may be up to 1 ulp (unit in last place) in error.

Examples:

```
log10('0')           ==> '-Infinity'
log10('0.001')       ==> '-3'
log10('1.000')       ==> '0'
log10('2')           ==> '0.301029996'
log10('10')          ==> '1'
log10('70')          ==> '1.84509804'
log10('+Infinity')   ==> 'Infinity'
```

Notes:

1. The *rounding* setting in the context is not used.
2. When the result is inexact, the cost of **log10** at a given precision is likely to be similar to, or more expensive than, the **exp** function (see notes under that function).

max

max takes two operands, compares their values numerically, and returns the maximum.[28] If either operand is a NaN then the general rules apply, unless one is a quiet NaN and the other is numeric, in which case the numeric operand is returned.[29]

Otherwise, the operands are compared as as though by the **compare** operation (see page 27). If they are not numerically equal then the maximum (closer to positive infinity) of the two operands is chosen as the result. Otherwise (they are numerically equal):

- if the operand signs differ the operand with sign 0 is chosen
- if the signs and exponents are equal the operands are identical so either can be chosen
- if the signs are both positive, the operand with the maximum exponent is chosen
- if the signs are both negative, the operand with the minimum exponent is chosen.

For numerical results, the result is the same as using the **plus** operation (see page 33) on the chosen operand, except that the sign of a zero does not change.

Examples:

```
max('3', '2')    ==>  '3'
max('-10', '3')  ==>  '3'
max('1.0', '1')  ==>  '1'
max('7', 'NaN')  ==>  '7'
```

max-magnitude

max-magnitude takes two operands and compares their values numerically with their *sign* ignored and assumed to be 0.

If, without signs, the first operand is the larger then the original first operand is returned (that is, with the original sign). If, without signs, the second operand is the larger then the original second operand is returned. Otherwise the result is the same as from the **max** operation.

min

min takes two operands, compares their values numerically, and returns the minimum.[30] If either operand is a NaN then the general rules apply, unless one is a quiet NaN and the other is numeric, in which case the numeric operand is returned.

Otherwise, the operands are compared as as though by the **compare** operation (see page 27). If they are not numerically equal then the minimum (closer to negative infinity) of the two operands is chosen

28 This is the IEEE 754 *maxnum* operation, with an explicit result for equal operands.
29 This permits a useful ordering of data in which NaNs are used to indicate "unknown" values.
30 This is the IEEE 754 *minnum* operation, with an explicit result for equal operands.

as the result. Otherwise (they are numerically equal):

- if the operand signs differ the operand with sign 1 is chosen
- if the signs and exponents are equal the operands are identical so either can be chosen
- if the signs are both positive, the operand with the minimum exponent is chosen
- if the signs are both negative, the operand with the maximum exponent is chosen.

For numerical results, the result is the same as using the **plus** operation (see page 33) on the chosen operand, except that the sign of a zero does not change.

Examples:

```
min('3', '2')     ==>  '2'
min('-10', '3')   ==>  '-10'
min('1.0', '1')   ==>  '1.0'
min('7', 'NaN')   ==>  '7'
```

min-magnitude

min-magnitude takes two operands and compares their values numerically with their *sign* ignored and assumed to be 0.

If, without signs, the first operand is the smaller then the original first operand is returned (that is, with the original sign). If, without signs, the second operand is the smaller then the original second operand is returned. Otherwise the result is the same as from the **min** operation.

minus and plus

minus and **plus** both take one operand, and correspond to the prefix minus and plus operators in programming languages.

The operations are evaluated using the same rules as **add** and **subtract**; the operations `plus(a)` and `minus(a)` (where `a` and `b` refer to any numbers) are calculated as the operations `add('0', a)` and `subtract('0', b)` respectively, where the `'0'` has the same exponent as the operand.

Examples:

```
plus('1.3')     ==>  '1.3'
plus('-1.3')    ==>  '-1.3'
minus('1.3')    ==>  '-1.3'
minus('-1.3')   ==>  '1.3'
```

Note that the result of these operations is affected by context and may set *flags*. The **copy-negate** operation (see page 44) may be used instead of **minus** if this is not desired.

multiply

multiply takes two operands. If either operand is a *special value* then the general rules apply.

Otherwise, the operands are multiplied together ("long multiplication"), resulting in a number which may be as long as the sum of the lengths of the two operands, as follows:

- The *coefficient* of the result, before rounding, is computed by multiplying together the coefficients of the operands.

- The *exponent* of the result, before rounding, is the sum of the exponents of the two operands.
- The *sign* of the result is the *exclusive or* of the signs of the operands.

The result is then rounded to *precision* digits if necessary, counting from the most significant digit of the result.

Examples:

```
multiply('1.20', '3')          ==>  '3.60'
multiply('7', '3')             ==>  '21'
multiply('0.9', '0.8')         ==>  '0.72'
multiply('0.9', '-0')          ==>  '-0.0'
multiply('654321', '654321')  ==>  '4.28135971E+11'
```

next-minus

next-minus takes one operand; if the operand is a NaN then the general rules apply. Otherwise the result is the largest representable number that is smaller than the operand unless the operand is -Infinity, in which case the result is -Infinity. If the result is zero its *sign* will be 0 and its *exponent* will be the smallest possible. No *flags* will be set when the operand is numeric.

In the following examples, E_{max} and E_{min} are assumed to be +999 and -999 respectively.

Examples:

```
next-minus('1')             ==>  '0.999999999'
next-minus('1E-1007')       ==>  '0E-1007'
next-minus('-1.00000003')  ==>  '-1.00000004'
next-minus('Infinity')      ==>  '9.99999999E+999'
```

next-plus

next-plus takes one operand; if the operand is a NaN then the general rules apply. Otherwise the result is the smallest representable number that is larger than the operand unless the operand is +Infinity, in which case the result is +Infinity. If the result is zero its *sign* will be 1 and its *exponent* will be the smallest possible. No *flags* will be set when the operand is numeric.

In the following examples, E_{max} and E_{min} are assumed to be +999 and -999 respectively.

Examples:

```
next-plus('1')             ==>  '1.00000001'
next-plus('-1E-1007')      ==>  '-0E-1007'
next-plus('-1.00000003')  ==>  '-1.00000002'
next-plus('-Infinity')     ==>  '-9.99999999E+999'
```

next-toward

next-toward takes two operands; if either operand is a NaN then the general rules apply. Otherwise the result is the representable number closest to the first operand (but not the first operand) that is in the direction towards the second operand, unless the operands have the same value. Specifically:

- If the second operand is larger than the first operand then the result is the result of the operation **next-plus** on the first operand
- If the second operand is smaller than the first operand then the result is the result of the

operation **next-minus** on the first operand

- If the two operands are numerically equal, then the result is a copy of the first operand with the *sign* set to be the same as the *sign* of the second operand; in this case no *flags* are set.

In the first two cases, *flags* are set as though the operation had been computed by adding (in the first case) or subtracting (in the second) an infinitesimally small positive value to or from the first operand with the rounding mode set to be *round-ceiling* or *round-floor* respectively.[31]

In the following examples, E_{max} and E_{min} are assumed to be +999 and -999 respectively.

Examples:

```
next-toward('1', '2')                ==>  '1.00000001'
next-toward('-1E-1007', '1')         ==>  '-0E-1007'
next-toward('-1.00000003', '0')      ==>  '-1.00000002'
next-toward('1', '0')                ==>  '0.999999999'
next-toward('1E-1007', '-100')       ==>  '0E-1007'
next-toward('-1.00000003', '-10')    ==>  '-1.00000004'
next-toward('0.00', '-0.0000')       ==>  '-0.00'
```

This operation derives its anomalous rules for *flags* from the IEEE 754-1985 operation *nextAfter*; the operation was dropped from the IEEE 754-2008 standard.

power

power takes two operands, and raises a number (the left-hand operand) to a power (the right-hand operand). If either operand is a *special value* then the general rules apply, except as stated below.

The following rules apply:

- If both operands are zero, or if the left-hand operand is less than zero and the right-hand operand does not have an integral value[32] or is infinite, an Invalid operation condition (see page 52) is raised, the result is `[0,qNaN]`, and the following rules do not apply.
- If the left-hand operand is infinite, the result will be exact and will be infinite if the right-hand side is positive, 1 if the right-hand side is a zero, and 0 if the right-hand side is negative.
- If the left-hand operand is a zero, the result will be exact and will be infinite if the right-hand side is negative or 0 if the right-hand side is positive.
- If the right-hand operand is a zero, the result will be 1 and exact.
- In cases not covered above, the result will be inexact unless the right-hand side has an integral value and the result is finite and can be expressed exactly within *precision* digits. In this latter case, if the result is unrounded then its exponent will be that which would result if the operation were calculated by repeated multiplication (if the second operand is negative then the reciprocal of the first operand is used, with the absolute value of the second operand determining the multiplications).
- Inexact finite results should be correctly rounded, but may be up to 1 ulp (unit in last place) in error.
- The *sign* of the result will be 1 only if the right-hand side has an integral value and is odd (and is

31 The result can in fact be computed by an appropriate addition, with one infinite value having a special case result and the sign of a zero result being set appropriately.

32 That is, any fractional part (after the decimal point) is non-zero.

not infinite) and also the *sign* of the left-hand side is 1. In all other cases, the *sign* of the result will be 0.

Examples:

```
power('2', '3')              ==>  '8'
power('-2', '3')             ==>  '-8'
power('2', '-3')             ==>  '0.125'
power('1.7', '8')            ==>  '69.7575744'
power('10', '0.301029996')   ==>  '2.00000000'
power('Infinity', '-1')      ==>  '0'
power('Infinity', '0')       ==>  '1'
power('Infinity', '1')       ==>  'Infinity'
power('-Infinity', '-1')     ==>  '-0'
power('-Infinity', '0')      ==>  '1'
power('-Infinity', '1')      ==>  '-Infinity'
power('-Infinity', '2')      ==>  'Infinity'
power('0', '0')              ==>  'NaN'
```

Notes:

1. When the result is inexact, the cost of **power** at a given precision is likely to be at least twice as expensive as the **exp** function (see notes under that function).
2. An infinite result is always exact, as described in the general rules.
3. Versions of this specification prior to version 1.48 defined a simpler **power** operation which only required support for integer powers.
4. It can be argued that the special cases where one operand is zero and the other is infinite (such as `power('0', 'Infinity')` and `power('Infinity', '0')`) should return a NaN, whereas the specification above leads to results of 0 and 1 respectively for the two examples (for compatibility with the earlier version of this operation). If NaN results are desired instead, then these special cases should be tested for before calling the power operation.

quantize

quantize takes two operands. If either operand is a *special value* then the general rules apply, except that if either operand is infinite and the other is finite an Invalid operation condition (see page 52) is raised and the result is `[0,qNaN]`, or if both are infinite then the result is the first operand.

Otherwise (both operands are finite), **quantize** returns the number which is equal in value (except for any rounding) and sign to the first (left-hand) operand and which has an *exponent* set to be equal to the exponent of the second (right-hand) operand.

The *coefficient* of the result is derived from that of the left-hand operand. It may be rounded using the current *rounding* setting (if the *exponent* is being increased), multiplied by a positive power of ten (if the *exponent* is being decreased), or is unchanged (if the *exponent* is already equal to that of the right-hand operand).

Unlike other operations, if the length of the *coefficient* after the quantize operation would be greater than *precision* then an Invalid operation condition is raised. This guarantees that, unless there is an error condition, the *exponent* of the result of a quantize is always equal to that of the right-hand operand.

Also unlike other operations, quantize will never raise Underflow, even if the result is subnormal and inexact.

Examples:

```
quantize('2.17', '0.001')        ==>  '2.170'
quantize('2.17', '0.01')         ==>  '2.17'
quantize('2.17', '0.1')          ==>  '2.2'
quantize('2.17', '1e+0')         ==>  '2'
quantize('2.17', '1e+1')         ==>  '0E+1'
quantize('-Inf'  'Infinity')     ==>  '-Infinity'
quantize('2',    'Infinity')     ==>  'NaN'
quantize('-0.1', '1'  )          ==>  '-0'
quantize('-0',   '1e+5')         ==>  '-0E+5'
quantize('+35236450.6', '1e-2')  ==>  'NaN'
quantize('-35236450.6', '1e-2')  ==>  'NaN'
quantize('217',  '1e-1')         ==>  '217.0'
quantize('217',  '1e+0')         ==>  '217'
quantize('217',  '1e+1')         ==>  '2.2E+2'
quantize('217',  '1e+2')         ==>  '2E+2'
```

Notes:

1. In the penultimate example the result is `[0,22,1]`, leading to the string in scientific notation as shown.
2. This operation was previously called **rescale**, which had identical semantics except that the second operand specified the power of ten of the quantum. The **quantize** semantics specifies the desired quantum by example, which allows a faster implementation in most cases.
3. The sign and coefficient of the second operand are ignored; this allows a “match the quantum of a variable” operation to be effected directly.

reduce

reduce takes one operand. It has the same semantics as the **plus** operation, except that if the final result is finite it is reduced to its simplest form, with all trailing zeros removed and its sign preserved.

That is, while the *coefficient* is non-zero and a multiple of ten the *coefficient* is divided by ten and the *exponent* is incremented by 1. Otherwise (the *coefficient* is zero) the *exponent* is set to 0. In all cases the *sign* is unchanged.

Examples:

```
reduce('2.1')     ==>  '2.1'
reduce('-2.0')    ==>  '-2'
reduce('1.200')   ==>  '1.2'
reduce('-120')    ==>  '-1.2E+2'
reduce('120.00')  ==>  '1.2E+2'
reduce('0.00')    ==>  '0'
```

This operation was called *normalize* prior to version 1.68 of this specification.

remainder

remainder takes two operands; it returns the remainder from integer division. If either operand is a *special value* then the general rules apply.

Otherwise, the result is the residue of the dividend after the operation of calculating integer division as described for **divide-integer**, rounded to *precision* digits if necessary. The sign of the result, if non-zero, is the same as that of the original dividend.

This operation will fail under the same conditions as integer division (that is, if integer division on the same two operands would fail, the remainder cannot be calculated).

Examples:

```
remainder('2.1', '3')    ==>  '2.1'
remainder('10', '3')     ==>  '1'
remainder('-10', '3')    ==>  '-1'
remainder('10.2', '1')   ==>  '0.2'
remainder('10', '0.3')   ==>  '0.1'
remainder('3.6', '1.3')  ==>  '1.0'
```

Notes:

1. The divide-integer and remainder operations are defined so that they may be calculated as a by-product of the standard division operation (described above). The division process is ended as soon as the integer result is available; the residue of the dividend is the remainder.
2. The sign of the result will always be sign of the dividend.
3. The remainder operation differs from the remainder operation defined in IEEE 754 (the **remainder-near** operator), in that it gives the same results for numbers whose values are equal to integers as would the usual remainder operator on integers.

   For example, the result of the operation `remainder('10', '6')` as defined here is `'4'`, and `remainder('10.0', '6')` would give `'4.0'` (as would `remainder('10', '6.0')` or `remainder('10.0', '6.0')`). The IEEE 754 remainder operation would, however, give the result `'-2'` because its integer division step chooses the closest integer, not the one nearer zero.

remainder-near

remainder-near takes two operands. If either operand is a *special value* then the general rules apply.

Otherwise, if the operands are given by x and y, then the result is defined to be $x - y \times n$, where n is the integer nearest the exact value of $x \div y$ (if two integers are equally near then the even one is chosen). If the result is equal to 0 then its sign will be the sign of x. (See IEEE 754 §5.3.1.)

This operation will fail under the same conditions as integer division (that is, if integer division on the same two operands would fail, the remainder cannot be calculated), except when the quotient is very close to 10 raised to the power of the precision.[33]

Examples:

```
remainder-near('2.1', '3')    ==>  '-0.9'
remainder-near('10', '6')     ==>  '-2'
remainder-near('10', '3')     ==>  '1'
remainder-near('-10', '3')    ==>  '-1'
remainder-near('10.2', '1')   ==>  '0.2'
remainder-near('10', '0.3')   ==>  '0.1'
remainder-near('3.6', '1.3')  ==>  '-0.3'
```

Notes:

1. The **remainder-near** operation differs from the **remainder** operation in that it does not give the same results for numbers whose values are equal to integers as would the usual remainder operator on integers. For example, the operation `remainder('10', '6')` gives the result

33 This is a deviation from IEEE 754, necessary to assure realistic execution times when the operands have a wide range of exponents.

`'4'`, and `remainder('10.0', '6')` gives `'4.0'` (as would the operations `remainder('10', '6.0')` or `remainder('10.0', '6.0')`). However, `remainder-near('10', '6')` gives the result `'-2'` because its integer division step chooses the closest integer, not the one nearer zero.

2. The result of this operation is always exact.
3. This operation is sometimes known as "IEEE remainder".

round-to-integral-exact

round-to-integral-exact takes one operand. If the operand is a *special value*, or the exponent of the operand is non-negative, then the result is the same as the operand (unless the operand is a signaling NaN, as usual).

Otherwise (the operand has a negative exponent) the result is the same as using the **quantize** operation using the given operand as the left-hand-operand, 1E+0 as the right-hand-operand, and the precision of the operand as the *precision* setting. The rounding mode is taken from the context, as usual.

Examples:

```
round-to-integral-exact('2.1')      ==>  '2'
round-to-integral-exact('100')      ==>  '100'
round-to-integral-exact('100.0')    ==>  '100'
round-to-integral-exact('101.5')    ==>  '102'
round-to-integral-exact('-101.5')   ==>  '-102'
round-to-integral-exact('10E+5')    ==>  '1.0E+6'
round-to-integral-exact('7.89E+77') ==>  '7.89E+77'
round-to-integral-exact('-Inf')     ==>  '-Infinity'
```

round-to-integral-value

round-to-integral-value takes one operand. It is identical to the **round-to-integral-exact** operation except that the Inexact and Rounded flags are never set even if the operand is rounded (that is, the operation is quiet unless the operand is a signaling NaN).

square-root

square-root takes one operand. If the operand is a *special value* then the general rules apply.

Otherwise, the ideal exponent of the result is defined to be half the exponent of the operand (rounded to an integer, towards -Infinity,[34] if necessary) and then:

- If the operand is less than zero an Invalid operation condition is raised.
- If the operand is greater than zero, the result is the square root of the operand. If no rounding is necessary (the exact result requires *precision* digits or fewer) then the the coefficient and exponent giving the correct value and with the exponent closest to the ideal exponent is used. If the result must be inexact, it is rounded using the *round-half-even* algorithm and the coefficient will have exactly *precision* digits (unless the result is subnormal), and the exponent will be set to

34 This rule matches the typical implementations. For example, the square-root of either `[0,10,-1]` or `[0,11,-1]` is often calculated by first multiplying the coefficient by ten and reducing the exponent by 1 and then determining the square root. If the exponent is held as a two's complement binary number, the ideal exponent is trivially calculated by applying an arithmetic right shift of one bit.

maintain the correct value.

- Otherwise (the operand is equal to zero), the result will be the zero with the same sign as the operand and with the ideal exponent.

Examples:

```
square-root('0')     ==> '0'
square-root('-0')    ==> '-0'
square-root('0.39')  ==> '0.62449980'
square-root('100')   ==> '10'
square-root('1')     ==> '1'
square-root('1.0')   ==> '1.0'
square-root('1.00')  ==> '1.0'
square-root('7')     ==> '2.64575131'
square-root('10')    ==> '3.16227766'
```

Notes:

1. The *rounding* setting in the context is not used; this means that the algorithm described in *Properly Rounded Variable Precision Square Root* by T. E. Hull and A. Abrham (ACM Transactions on Mathematical Software, Vol 11 #3, pp229–237, ACM, September 1985) may be used for this operation.
2. A subnormal result is only possible if the working precision is greater than E_{max}+1.
3. The rules for setting the exponent of the result apply to many operations; they can be used for any operation for which an ideal exponent can be defined.
4. A negative zero is allowed as an operand as per IEEE 754 §5.4.1.
5. Square-root can also be calculated by using the **power** (see page 35) operation (with a second operand of 0.5). The result in that case will not be exact in most cases, and may not be correctly rounded.[35]

35 This is because a typical implementation of power(x,y) will calculate its result using exp(ln(x)*y), and few results of the exp function are exact.

Miscellaneous operations

This section describes miscellaneous operations on decimal numbers, including non-numeric comparisons, sign and other manipulations, and logical operations.

The logical operations (**and**, **invert**, **or**, and **xor**) take *logical operands*, which are finite numbers with a *sign* of 0, an *exponent* of 0, and a *coefficient* whose digits must all be either 0 or 1.[36] The length of the result will be at most *precision* digits (all of which will be either 0 or 1); operands are truncated on the left or padded with zeros on the left as necessary. The result of a logical operation is never rounded and the only *flag* that might be set is *invalid-operation* (set if an operand is not a valid logical operand).

Notes:

1. This section uses the simplified notation introduced in the previous section to illustrate operations.
2. It is possible to express the results of all these operations as a decimal number or string, but for some implementations other types may be available and might be more efficient.[37] If an implementation does return types which are not decimal numbers or strings then there must also be conversion operations provided to convert from those types to decimal values or strings, so a wholly decimal usage is possible.
3. As in the previous section, for any examples below, the context (where relevant) is assumed to have *precision* set to 9, *rounding* set to *round-half-up*, and all *trap-enablers* set to 0.

and

and is a logical operation which takes two logical operands (see above). The result is the digit-wise *and* of the two operands; each digit of the result is the logical and of the corresponding digits of the operands, aligned at the least-significant digit. A result digit is 1 if both of the corresponding operand digits are 1; otherwise it is 0.

Examples:

```
and('0', '0')         ==>  '0'
and('0', '1')         ==>  '0'
and('1', '0')         ==>  '0'
and('1', '1')         ==>  '1'
and('1100', '1010')  ==>  '1000'
and('1111', '10')    ==>  '10'
```

36 This *digit-wise* representation of bits in a decimal representation has been used since the 1950s; see, for example, *Binary and truth-function operations on a decimal computer with an extract command*, William H. Kautz, Communications of the ACM, Vol. 1 #5, pp12-13, ACM Press, May 1958.

37 For example, the operations which test whether a value is in a particular class might return a *boolean*.

canonical

canonical takes one operand. The result has the same value as the operand but always uses a *canonical* encoding. The definition of *canonical* is implementation-defined; if more than one internal encoding for a given NaN, Infinity, or finite number is possible then one "preferred" encoding is deemed canonical. This operation then returns the value using that preferred encoding.

If all possible operands have just one internal encoding each, then **canonical** always returns the operand unchanged (that is, it has the same effect as **copy**). This operation is unaffected by context and is quiet – no *flags* are changed in the context.

Example:

```
canonical('2.50')  ==>  '2.50'
```

class

class takes one operand. The result is an indication of the *class* of the operand, where the class is one of ten possibilities, corresponding to one of the strings `"sNaN"` (signaling NaN), `"NaN"` (quiet NaN), `"-Infinity"` (negative infinity), `"-Normal"` (negative normal finite number), `"-Subnormal"` (negative subnormal finite number), `"-Zero"` (negative zero), `"+Zero"` (non-negative zero), `"+Subnormal"` (positive subnormal finite number), `"+Normal"` (positive normal finite number), or `"+Infinity"` (positive infinity). This operation is quiet; no *flags* are changed in the context.

Implementations may indicate the class using a more easily tested representation than a string (for example, an integer or an enumeration), but in this case a mechanism for converting that representation to the corresponding class string listed above must be provided.

Finite numbers can only be classified as subnormal (see page 11) if the exponent range is limited (that is, there is a known value for E_{min}). In the following examples, E_{min} is assumed to be -999.

Examples:

```
class('Infinity')   ==>  "+Infinity"
class('1E-10')      ==>  "+Normal"
class('2.50')       ==>  "+Normal"
class('0.1E-999')   ==>  "+Subnormal"
class('0')          ==>  "+Zero"
class('-0')         ==>  "-Zero"
class('-0.1E-999')  ==>  "-Subnormal"
class('-1E-10')     ==>  "-Normal"
class('-2.50')      ==>  "-Normal"
class('-Infinity')  ==>  "-Infinity"
class('NaN')        ==>  "NaN"
class('-NaN')       ==>  "NaN"
class('sNaN')       ==>  "sNaN"
```

Note that unlike the special values in the model, the sign of any NaN is ignored in the classification, as required by IEEE 754.

compare-total

compare-total takes two operands and compares them using their abstract representation rather than their numerical value. A *total ordering* is defined for all possible abstract representations, as described below. If the first operand is lower in the total order than the second operand then the result is `'-1'`, if the operands have the same abstract representation then the result is `'0'`, and if the first operand is higher in the total order than the second operand then the result is `'1'`.

The total ordering is defined as follows.

1. The following items describe the ordering for representations whose *sign* is 0. If the *sign* is 1, the order is reversed. A representation with a *sign* of 1 is always lower in the ordering than one with a *sign* of 0.
2. Numbers (representations which are not NaNs) are ordered such that a larger numerical value is higher in the ordering. If two representations have the same numerical value then the exponent is taken into account; larger (more positive) exponents are higher in the ordering.
3. All quiet NaNs are higher in the total ordering than all signaling NaNs.
4. Quiet NaNs and signaling NaNs are ordered according to their *payload*; a larger payload is higher in the ordering.

For example, the following values are ordered from lowest to highest: `-NaN -sNaN -Infinity -127 -1.00 -1 -0.000 -0 0 1.2300 1.23 1E+9 Infinity sNaN NaN NaN456`.

Examples:

```
compare-total('12.73', '127.9')   ==>  '-1'
compare-total('-127',  '12')      ==>  '-1'
compare-total('12.30', '12.3')    ==>  '-1'
compare-total('12.30', '12.30')   ==>  '0'
compare-total('12.3',  '12.300')  ==>  '1'
compare-total('12.3',  'NaN')     ==>  '-1'
```

Notes:

1. The result of **compare-total** is always finite, exact, and unrounded.
2. The **compare** operation (see page 27) can be used when a numerical comparison of values is required.

compare-total-magnitude

compare-total-magnitude takes two operands and compares them using their abstract representation rather than their numerical value and with their *sign* ignored and assumed to be 0. The result is identical to that obtained by using **compare-total** on two operands which are the **copy-abs** copies of the operands to **compare-total-magnitude**; that is:

```
compare-total-magnitude(x, y)
```

is given by

```
compare-total(copy-abs(x), copy-abs(y))
```

copy

copy takes one operand. The result is a copy of the operand. This operation is unaffected by context and is quiet – no *flags* are changed in the context.

Examples:

```
copy('2.1')    ==>  '2.1'
copy('-1.00')  ==>  '-1.00'
```

copy-abs

copy-abs takes one operand. The result is a copy of the operand with the *sign* set to 0. Unlike the **abs** operation (see page 26), this operation is unaffected by context and is quiet – no *flags* are changed in the context.

Examples:

```
copy-abs('2.1')   ==>  '2.1'
copy-abs('-100')  ==>  '100'
```

copy-negate

copy-negate takes one operand. The result is a copy of the operand with the *sign* inverted (a *sign* of 0 becomes 1 and vice versa). Unlike the **minus** operation (see page 33), this operation is unaffected by context and is quiet – no *flags* are changed in the context.

Examples:

```
copy-negate('101.5')   ==>  '-101.5'
copy-negate('-101.5')  ==>  '101.5'
```

copy-sign

copy-sign takes two operands. The result is a copy of the first operand with the *sign* set to be the same as the *sign* of the second operand. This operation is unaffected by context and is quiet – no *flags* are changed in the context.

Examples:

```
copy-sign( '1.50',  '7.33')  ==>  '1.50'
copy-sign('-1.50',  '7.33')  ==>  '1.50'
copy-sign( '1.50', '-7.33')  ==>  '-1.50'
copy-sign('-1.50', '-7.33')  ==>  '-1.50'
```

invert

invert is a logical operation which takes one logical operand (see above). The result is the digit-wise *inversion* of the operand; each digit of the result is the inverse of the corresponding digit of the operand. A result digit is 1 if the corresponding operand digit is 0; otherwise it is 0.

Examples:

```
invert('0')          ==>  '111111111'
invert('1')          ==>  '111111110'
invert('111111111')  ==>  '0'
invert('101010101')  ==>  '10101010'
```

is-canonical

is-canonical takes one operand. The result is 1 if the operand is *canonical*; otherwise it is 0. The definition of *canonical* is implementation-defined; if more than one internal encoding for a given NaN, Infinity, or finite number is possible then one “preferred” encoding is deemed canonical. This operation then tests whether the internal encoding is that preferred encoding.

If all possible operands have just one internal encoding each, then **is-canonical** always returns 1. This operation is unaffected by context and is quiet – no *flags* are changed in the context.

Example:

```
is-canonical('2.50')  ==>  '1'
```

is-finite

is-finite takes one operand. The result is 1 if the operand is neither infinite nor a NaN (that is, it is a normal number, a subnormal number, or a zero); otherwise it is 0. This operation is unaffected by context and is quiet – no *flags* are changed in the context.

Examples:

```
is-finite('2.50')  ==>  '1'
is-finite('-0.3')  ==>  '1'
is-finite('0')     ==>  '1'
is-finite('Inf')   ==>  '0'
is-finite('NaN')   ==>  '0'
```

is-infinite

is-infinite takes one operand. The result is 1 if the operand is an Infinity; otherwise it is 0. This operation is unaffected by context and is quiet – no *flags* are changed in the context.

Examples:

```
is-infinite('2.50')  ==>  '0'
is-infinite('-Inf')  ==>  '1'
is-infinite('NaN')   ==>  '0'
```

is-NaN

is-NaN takes one operand. The result is 1 if the operand is a NaN (quiet or signaling); otherwise it is 0. This operation is unaffected by context and is quiet – no *flags* are changed in the context.

Examples:

```
is-NaN('2.50')   ==>  '0'
is-NaN('NaN')    ==>  '1'
is-NaN('-sNaN')  ==>  '1'
```

is-normal

is-normal takes one operand. The result is 1 if the operand is a positive or negative *normal number* (see page 11); otherwise it is 0. This operation is quiet; no *flags* are changed in the context.

Finite numbers can only be classified as *normal* or *subnormal* if the exponent range is limited (that is, there is a known value for E_{min}); if E_{min} is unknown, 1 is returned. In the following examples, E_{min} is assumed to be -999.

Examples:

```
is-normal('2.50')      ==>  '1'
is-normal('0.1E-999')  ==>  '0'
is-normal('0.00')      ==>  '0'
is-normal('-Inf')      ==>  '0'
is-normal('NaN')       ==>  '0'
```

is-qNaN

is-qNaN takes one operand. The result is 1 if the operand is a quiet NaN; otherwise it is 0. This operation is unaffected by context and is quiet – no *flags* are changed in the context.

Examples:

```
is-qNaN('2.50')  ==>  '0'
is-qNaN('NaN')   ==>  '1'
is-qNaN('sNaN')  ==>  '0'
```

is-signed

is-signed takes one operand. The result is 1 if the *sign* of the operand is 1; otherwise it is 0. This operation is unaffected by context and is quiet – no *flags* are changed in the context.

Examples:

```
is-signed('2.50')  ==>  '0'
is-signed('-12')   ==>  '1'
is-signed('-0')    ==>  '1'
```

is-sNaN

is-sNaN takes one operand. The result is 1 if the operand is a signaling NaN; otherwise it is 0. This operation is unaffected by context and is quiet – no *flags* are changed in the context.

Examples:

```
is-sNaN('2.50')  ==>  '0'
is-sNaN('NaN')   ==>  '0'
is-sNaN('sNaN')  ==>  '1'
```

is-subnormal

is-subnormal takes one operand. The result is 1 if the operand is a positive or negative *subnormal number* (see page 11); otherwise it is 0. This operation is quiet; no *flags* are changed in the context.

Finite numbers can only be classified as *normal* or *subnormal* if the exponent range is limited (that is, there is a known value for E_{min}); if E_{min} is unknown, 0 is returned. In the following examples, E_{min} is assumed to be -999.

Examples:

```
is-subnormal('2.50')      ==>  '0'
is-subnormal('0.1E-999')  ==>  '1'
is-subnormal('0.00')      ==>  '0'
is-subnormal('-Inf')      ==>  '0'
is-subnormal('NaN')       ==>  '0'
```

is-zero

is-zero takes one operand. The result is 1 if the operand is a zero; otherwise it is 0. This operation is unaffected by context and is quiet – no *flags* are changed in the context.

Examples:

```
is-zero('0')      ==>  '1'
is-zero('2.50')   ==>  '0'
is-zero('-0E+2')  ==>  '1'
```

logb

logb takes one operand. If the operand is a NaN then the general arithmetic rules apply. If the operand is infinite then +Infinity is returned. If the operand is a zero, then -Infinity is returned and the Division by zero exceptional condition (see page 51) is raised.

Otherwise, the result is the integer which is the exponent of the magnitude of the most significant digit of the operand (as though the operand were truncated to a single digit while maintaining the value of that digit and without limiting the resulting exponent). All results are exact unless an integer result does not fit in the available *precision*.

Examples:

```
logb('250')   ==>  '2'
logb('2.50')  ==>  '0'
logb('0.03')  ==>  '-2'
logb('0')     ==>  '-Infinity'
```

Note: The **scaleb** operation (see page 48) can be used to change the exponent of a number.

or

or is a logical operation which takes two logical operands (see above). The result is the digit-wise *inclusive or* of the two operands; each digit of the result is the logical or of the corresponding digits of the operands, aligned at the least-significant digit. A result digit is 1 if either or both of the corresponding operand digits is 1; otherwise it is 0.

Examples:

```
or('0', '0')         ==>  '0'
or('0', '1')         ==>  '1'
or('1', '0')         ==>  '1'
or('1', '1')         ==>  '1'
or('1100', '1010')   ==>  '1110'
or('1110', '10')     ==>  '1110'
```

radix

radix takes no operands. The result is the radix (base) in which arithmetic is effected; for this specification the result will have the value 10.[38]

Example:

```
radix()  ==>  '10'
```

rotate

rotate takes two operands. The second operand must be an integer (with an *exponent* of 0) in the range -*precision* through *precision*. If the first operand is a NaN then the general arithmetic rules apply, and if

38 This might be 1E+1 in the extraordinary case of *precision*=1.

it is infinite then the result is the Infinity unchanged.

Otherwise (the first operand is finite) the result has the same *sign* and *exponent* as the first operand, and a *coefficient* which is a rotated copy of the digits in the coefficient of the first operand. The number of places of rotation is taken from the absolute value of the second operand, with the rotation being to the left if the second operand is positive or to the right otherwise.

If the coefficient of the first operand has fewer than *precision* digits, it is treated as though it were padded on the left with zeros to length *precision* before the rotation. Similarly, if the coefficient of the first operand has more than *precision* digits, it is truncated on the left before use.

The only *flag* that might be set is *invalid-operation* (set if the first operand is an sNaN or the second is not valid).

Examples:

```
rotate('34', '8')             ==>  '400000003'
rotate('12', '9')             ==>  '12'
rotate('123456789', '-2')  ==>  '891234567'
rotate('123456789', '0')   ==>  '123456789'
rotate('123456789', '+2')  ==>  '345678912'
```

The shift operation (see page 49) can be used to shift rather than rotate a coefficient.

same-quantum

same-quantum takes two operands, and returns 1 if the two operands have the same *exponent* or 0 otherwise. The result is never affected by either the sign or the coefficient of either operand.

If either operand is a *special value*, 1 is returned only if both operands are NaNs or both are infinities.

same-quantum does not change any *flags* in the context. Implementations which support the concept of a *boolean* type may return *true* for 1 and *false* for 0.

Examples:

```
samequantum('2.17', '0.001')  ==>  '0'
samequantum('2.17', '0.01')   ==>  '1'
samequantum('2.17', '0.1')    ==>  '0'
samequantum('2.17', '1')      ==>  '0'
samequantum('Inf', '-Inf')    ==>  '1'
samequantum('NaN', 'NaN')     ==>  '1'
```

scaleb

scaleb takes two operands. If either operand is a NaN then the general arithmetic rules apply. Otherwise, the second operand must be a finite integer with an exponent of zero[39] and in the range $\pm 2 \times (E_{max} + precision)$ inclusive, where E_{max} is the largest value that can be returned by the **logb** operation (see page 47) at the same *precision* setting.[40] (If is is not, the Invalid Operation condition is raised and the result is NaN.)

If the first operand is infinite then that Infinity is returned, otherwise the result is the first operand modified by adding the value of the second operand to its *exponent*. The result may Overflow or Underflow.

39 Strictly speaking this is more restrictive that IEEE 754, which would allow 1E+1 for the second operand; however, it is in the spirit of IEEE 754 also permitting that the second operand be specifiable as an integer.

40 This E_{max} is the same as that described in IEEE 754.

Examples:

```
scaleb('7.50', '-2')  ==>  '0.0750'
scaleb('7.50', '0')   ==>  '7.50'
scaleb('7.50', '3')   ==>  '7.50E+3'
```

shift

shift takes two operands. The second operand must be an integer (with an *exponent* of 0) in the range -*precision* through *precision*. If the first operand is a NaN then the general arithmetic rules apply, and if it is infinite then the result is the Infinity unchanged.

Otherwise (the first operand is finite) the result has the same *sign* and *exponent* as the first operand, and a *coefficient* which is a shifted copy of the digits in the coefficient of the first operand. The number of places to shift is taken from the absolute value of the second operand, with the shift being to the left if the second operand is positive or to the right otherwise. Digits shifted into the coefficient are zeros.

The only *flag* that might be set is *invalid-operation* (set if the first operand is an sNaN or the second is not valid).

Examples:

```
shift('34', '8')           ==>  '400000000'
shift('12', '9')           ==>  '0'
shift('123456789', '-2')  ==>  '1234567'
shift('123456789', '0')   ==>  '123456789'
shift('123456789', '+2')  ==>  '345678900'
```

The rotate operation (see page 47) can be used to rotate rather than shift a coefficient.

xor

xor is a logical operation which takes two logical operands (see above). The result is the digit-wise *exclusive or* of the two operands; each digit of the result is the logical exclusive-or of the corresponding digits of the operands, aligned at the least-significant digit. A result digit is 1 if one of the corresponding operand digits is 1 and the other is 0; otherwise it is 0.

Examples:

```
xor('0', '0')          ==>  '0'
xor('0', '1')          ==>  '1'
xor('1', '0')          ==>  '1'
xor('1', '1')          ==>  '0'
xor('1100', '1010')  ==>  '110'
xor('1111', '10')     ==>  '1101'
```

Exceptional conditions

This section lists, in the abstract, the exceptional conditions that may arise during the operations defined in this specification.

For each condition, the corresponding *signal* in the *context* (see page 13) is given, along with the defined result. The value of the trap-enabler for each signal in the context determines whether an operation is completed after the condition is detected or whether the condition is trapped and hence not necessarily completed (see IEEE 754 §7 and §8).

This specification does not define the manner in which exceptions are reported or handled. For example, in a object-oriented language, an Arithmetic Exception object might be signalled or thrown, whereas in a calculator application an error message or other indication might be displayed.

The following exceptional conditions can occur:

Clamped This occurs and signals *clamped* if the exponent of a result has been altered in order to fit the constraints of a specific concrete representation. This may occur when the exponent of a zero result would be outside the bounds of a representation, or (in the IEEE 754 interchange formats) when a large normal number would have an encoded exponent that cannot be represented. In this latter case, the exponent is reduced to fit and the corresponding number of zero digits are appended to the coefficient ("fold-down"). The condition always occurs when a subnormal value rounds to zero.

Conversion syntax This occurs and signals *invalid-operation* if a string is being converted to a number and it does not conform to the numeric string syntax (see page 17). The result is `[0,qNaN]`.

Division by zero This occurs and signals *division-by-zero* if division of a finite number by zero was attempted (during a **divide-integer** or **divide** operation, or a **power** operation with negative right-hand operand), and the dividend was not zero.

The result of the operation is `[`*sign*`,inf]`, where *sign* is the exclusive or of the signs of the operands for divide, or is 1 for an odd power of `-0`, for power.

Division impossible This occurs and signals *invalid-operation* if the integer result of a **divide-integer** or **remainder** operation had too many digits (would be longer than *precision*). The result is `[0,qNaN]`.

Division undefined This occurs and signals *invalid-operation* if division by zero was attempted (during a **divide-integer**, **divide**, or **remainder** operation), and the dividend is also zero. The result is `[0,qNaN]`.

Inexact This occurs and signals *inexact* whenever the result of an operation is not exact (that is, it needed to be rounded and any discarded digits were non-zero), or if an overflow or underflow condition occurs. The result in all cases is unchanged.

The *inexact* signal may be tested (or trapped) to determine if a given operation (or sequence of operations) was inexact.[41]

Insufficient storage For many implementations, storage is needed for calculations and intermediate results, and on occasion an arithmetic operation may fail due to lack of storage. This is considered an operating environment error, which can be either be handled as appropriate for the environment, or treated as an **Invalid operation** condition. The result is `[0,qNaN]`.

Invalid context This occurs and signals *invalid-operation* if an invalid context was detected during an operation. This can occur if contexts are not checked on creation and either the *precision* exceeds the capability of the underlying concrete representation or an unknown or unsupported *rounding* was specified. These aspects of the context need only be checked when the values are required to be used. The result is `[0,qNaN]`.

Invalid operation This occurs and signals *invalid-operation* if:

- an operand to an operation is `[s,sNaN]` or `[s,sNaN,d]` (any *signaling NaN*)
- an attempt is made to add `[0,inf]` to `[1,inf]` during an addition or subtraction operation
- an attempt is made to multiply 0 by `[0,inf]` or `[1,inf]`
- an attempt is made to divide either `[0,inf]` or `[1,inf]` by either `[0,inf]` or `[1,inf]`
- the divisor for a remainder operation is zero
- the dividend for a remainder operation is either `[0,inf]` or `[1,inf]`
- either operand of the **quantize** operation is infinite, or the result of a **quantize** operation would require greater precision than is available
- the operand of the **ln** or the **log10** operation is less than zero
- the operand of the **square-root** operation has a *sign* of 1 and a non-zero *coefficient*
- both operands of the **power** operation are zero, or if the left-hand operand is less than zero and the right-hand operand does not have an integral value or is infinite
- an operand is invalid; for example, certain values of concrete representations may not correspond to numbers – an implementation is permitted (but is not required) to detect these invalid values and raise this condition.

The result of the operation after any of these invalid operations is `[0,qNaN]` except when the cause is a signaling NaN, in which case the result is `[s,qNaN]` or

41 Note that IEEE 854 is inconsistent in its treatment of Inexact in that it states in §7 that the Inexact exception can coincide with Underflow, but does not allow the possibility of Underflow signaling Inexact in §7.5. It is assumed that the latter is an accidental omission.

`[s,qNaN,d]` where the sign and diagnostic are copied from the signaling NaN.

Overflow This occurs and signals *overflow* if the *adjusted exponent* of a result (from a conversion or from an operation that is not an attempt to divide by zero), after rounding, would be greater than the largest value that can be handled by the implementation (the value E_{max}).

The result depends on the rounding mode:

- For *round-half-up* and *round-half-even* (and for *round-half-down* and *round-up*, if implemented), the result of the operation is `[`*sign*`,inf]`, where *sign* is the sign of the intermediate result.
- For *round-down*, (and *round-05up*, if implemented), the result is the largest finite number that can be represented in the current *precision*, with the sign of the intermediate result.
- For *round-ceiling*, the result is the same as for *round-down* if the sign of the intermediate result is 1, or is `[0,inf]` otherwise.
- For *round-floor*, the result is the same as for *round-down* if the sign of the intermediate result is 0, or is `[1,inf]` otherwise.

In all cases, **Inexact** and **Rounded** will also be raised.

Note: IEEE 854 §7.3 requires that the result delivered to a trap handler be different, depending on whether the overflow was the result of a conversion or of an arithmetic operation. This specification deviates from IEEE 854 in this respect; however, an implementation could comply with IEEE 854 by providing a separate mechanism for the special result to a trap handler. IEEE 754 has no such requirement.

Rounded This occurs and signals *rounded* whenever the result of an operation is rounded (that is, some zero or non-zero digits were discarded from the coefficient), or if an overflow or underflow condition occurs. The result in all cases is unchanged.

The *rounded* signal may be tested (or trapped) to determine if a given operation (or sequence of operations) caused a loss of precision.

Subnormal This occurs and signals *subnormal* whenever the result of a conversion or operation is subnormal (that is, its adjusted exponent is less than E_{min}, before any rounding). The result in all cases is unchanged.

The *subnormal* signal may be tested (or trapped) to determine if a given or operation (or sequence of operations) yielded a subnormal result.

Underflow This occurs and signals *underflow* if a result is inexact and the *adjusted exponent* of the result would be smaller (more negative) than the smallest value that can be handled by the implementation (the value E_{min}). That is, the result is both inexact and subnormal.[42]

The result after an underflow will be a subnormal number rounded, if necessary, so that its exponent is not less than E_{tiny}. This may result in 0 with the sign of the intermediate result and an exponent of E_{tiny}.

In all cases, **Inexact**, **Rounded**, and **Subnormal** will also be raised.

42 See IEEE 754 §7.5.

> **Note:** IEEE 854 §7.4 requires that the result delivered to a trap handler be different, depending on whether the underflow was the result of a conversion or of an arithmetic operation. This specification deviates from IEEE 854 in this respect; however, an implementation could comply with IEEE 854 by providing a separate mechanism for the result to a trap handler. IEEE 754 has no such requirement.

It is recommended that implementations distinguish the different conditions listed above, and also provide additional information about exceptional conditions where possible (for example, the operation being attempted and the values of the operand or operands involved – see also IEEE 754 §8).

Precedence of exceptions

The Clamped, Inexact, Rounded, and Subnormal conditions can coincide with each other or with other conditions. In these cases then any trap enabled for another condition takes precedence over (is handled before) all of these, any Subnormal trap takes precedence over Inexact, any Inexact trap takes precedence over Rounded, and any Rounded trap takes precedence over Clamped.

Appendix A – The X3.274 subset

The full specification in the body of this document defines a decimal floating-point arithmetic which gives exact results and preserves exponents where possible. If insufficient precision is available for this, then numbers are handled according to the rules of IEEE 854. The use of IEEE 854 rules implies that special values (infinities and NaNs) are allowed, as subnormal values and the value `-0`.

For some applications and programming languages (especially those intended for use by people who are not mathematically sophisticated), it may be appropriate to provide an arithmetic where infinite, NaN, or subnormal results are always treated as errors, `-0` results are hidden, and other (largely cosmetic) changes are provided to aid acceptance of results.

The arithmetic defined in ANSI X3.274 is such an arithmetic; this appendix describes the differences between this and the full specification. Implementations which support this subset explicitly might provide the subset behavior under the control of a parameter in the *context*[43] or might provide a different interface (additional or parameterized methods, for example).

Simplified number set

In the subset arithmetic, a reduced set of number values is supported and (where appropriate) numbers with positive exponents have their exponent reduced to zero. Specifically:

- In the **to-number** conversion, if the *coefficient* for a finite number has the value zero, then the *sign* and the *exponent* are both set to 0.
- If the *coefficient* in a result has the value zero, then the *sign* is set to 0 and (unless the operation is **quantize**) the *exponent* is set to 0.[44]
- In the **to-number** conversion, strings which represent special values are not permitted. (That is, only finite numbers are accepted.)
- Subnormal numbers are not permitted. If the result from a conversion or operation would be subnormal then an Underflow error results (see below).
- After any operation and the rounding of its result (unless the operation is **quantize**), a result with a positive exponent is converted to an integer provided that the resulting *coefficient* would have no more than *precision* digits. In other words, in this case a positive exponent is reduced to 0 by multiplying the *coefficient* by $10^{exponent}$ (which has the effect of suffixing *exponent* zeros).[45]

---

43 The decNumber package, for example, provides the subset behavior if the *extended* bit is set to 0.

44 This rule, together with the **to-number** definition, ensures that numbers with values such as `-0` or 0.0000 will not result from general operations in the subset arithmetic. This allows a concrete representation for the subset to comprise simply two integers in two's complement form.

45 The underlying intent here is that positive exponents in the operands are reduced to zero *before* the operation, so that all operations take place on numbers that could be expressed as "plain" decimal numbers with no exponent. The rule is

Operation differences

In the subset arithmetic, operands are rounded before use if necessary (as in Numerical Turing[46] and Rexx), the *Lost digits* condition is added to the context, the results of some operations are trimmed, the rounding rule after a subtraction is less conservative, and raising 0 to the power 0 is not treated as an error. Specifically:

- If the number of decimal digits in the *coefficient* of an operand to an operation is greater than the current *precision* in the context then the operand is rounded to *precision* significant digits using the *rounding* algorithm described by the context before being used in the computation. In other words, an automatic "convert to shorter" is applied before the operation.
- During an **add** or **subtract** operation, if either number is zero then the other number, rounded to *precision* if necessary, is used as the result (with sign adjustment as appropriate).[47]
- The **Lost digits** condition is added to the abstract context; it should be set to 0 in default contexts.

  This condition is raised when non-zero digits are discarded before an operation. This can occur when an operand which has more leading significant digits in its *coefficient* than the *precision* setting is rounded to *precision* digits before use

  Note that the lost digits test does not treat trailing decimal zeros in the *coefficient* as significant. For example, if *precision* had the value 5, then the operands

  ```
  [0,12345,-5]
  [0,12345,-2]
  [0,12345,0]
  [1,12345,0]
  [0,123450000,-4]
  [0,1234500000,0]
  ```

  would not cause an exception (whereas `[0,123451,-1]` or `[0,1234500001,0]` would).
- After a **divide** or **power** operation is complete and the result has been rounded, any insignificant trailing zeros are removed. That is, if the *exponent* is not zero and the *coefficient* is a multiple of a positive power of ten then the *coefficient* is divided by that power of ten and the *exponent* increased accordingly. If the *exponent* was negative it will not be increased above zero.
- After an addition operation, the result is rounded to *precision* digits if necessary, taking into account any extra (carry) digit on the left after an addition, but otherwise counting from the position corresponding to the most significant digit of the operands being added or subtracted (rather than the most significant digit of the result).
- For the **max** and **min** operations, the first (left-hand) operand is chosen if the operands are numerically equal.
- If both operands to a **power** operation are zero then the result is 1 (instead of being an error); however, if the left-hand operand is zero the right-hand operand must not be negative.
- The right-hand operand to a **power** operation may be an integer, and subset implementations are

---

expressed as a constraint on the result because it is often more convenient or efficient to implement it in this way. The rule also preserves integers as specified by ANSI X3.274, and in particular ensures that the results of the **divide** and **divide-integer** operations are identical when the result is an exact integer.

46 See: T. E. Hull, A. Abrham, M. S. Cohen, A. F. X. Curley, C. B. Hall, D. A. Penny, and J. T. M. Sawchuk, *Numerical Turing*, SIGNUM Newsletter, vol. 20 #3, pp26-34, ACM, May 1985.

47 In the subset arithmetic, zeros have no exponent.

only required to provide the power function for integer powers. In this case the algorithm described below may be used for calculating the result.

- The **fused-multiply-add** operation is not defined for subset implementations, because the rounding of operands rule conflicts with the requirement for fused-multiply-add to deliver a result with only one rounding.

Exceptional condition and rounding mode rules

In the subset arithmetic, exceptional conditions other than the informational conditions (Lost digits, Inexact, Rounded, and Subnormal) must be treated as errors, and results after these errors are undefined. Special values and subnormal numbers, therefore, are not part of the arithmetic.

In the subset, only the Lost digits trap enabler is required. Inexact, Rounded, and Subnormal trap enablers are optional, and the others are (in effect) always set. Similarly, the status bits in the *context* are optional.

Only the *round-half-up* rounding mode is required.

Calculating an integer power

Subset implementations are only required to provide the power function for integer powers. To calculate this, the number (left-hand operand) is in theory multiplied by itself for the number of times expressed by the power. If the right-hand operand is negative, the left-hand operand is used as-is, the absolute value of the right-hand operand is used, and the final result is inverted.[48]

In practice (see the note below for the reasons), the power is often calculated by the process of left-to-right binary reduction. For `power(x, n)`: "`n`" is converted to binary, and a temporary accumulator is set to 1. If "`n`" has the value 0 then the initial calculation is complete. Otherwise each bit (starting at the first non-zero bit) is inspected from left to right. If the current bit is 1 then the accumulator is multiplied by "`x`". If all bits have now been inspected then the initial calculation is complete, otherwise the accumulator is squared by multiplication and the next bit is inspected.

The multiplications and any final division are done under the normal arithmetic operation rules, using the precision supplied for the operation, except that the multiplications (and the division, if needed) are carried out using an increased precision of *precision*`+elength+1` digits. Here, `elength` is the length in decimal digits of the integer part (coefficient) of the whole number "`n`" (*i.e.*, excluding any sign, decimal part, decimal point, or insignificant leading zeros.[49]

If, when raising to a negative power, an underflow occurs during the division into 1, the operation is not halted at that point but continues.[50]

Notes:

1. A particular algorithm for calculating integer powers is described, since it is efficient (though not optimal) and considerably reduces the number of actual multiplications performed. It therefore gives better performance than the simpler definition of repeated multiplication. Since results can occasionally differ from those of repeated multiplication, the algorithm is defined

48 This rule is slightly more complicated than inverting before the calculation, in that it requires special treatment of overflow and underflow conditions (which were not an issue in X3.274).

49 The precision specified for the intermediate calculations ensuresthat the final result will differ by at most 1, in the least significant position, from the "true" result (given that the operands are expressed precisely under the current setting of **digits**). Half of this maximum error comes from the intermediate calculation, and half from the final rounding.

50 It can only be halted early if the result becomes zero.

here so that different implementations which use it will give identical results for the power operation on the same values, and may therefore use the same testcases. Other algorithms for the power operation may also be used, so long as the result is within 1 ulp (unit in last place) of the correct result.

2. Implementations are encouraged to provide a power operator which will accept a non-integral right-hand operand when the left-hand operand is non-negative, as described in the body of this specification.

Appendix B – Design concepts

This appendix summarizes the concepts underlying the arithmetic described in this document, as background information. It is not part of the specification.

The decimal arithmetic specified in this document is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – *computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school*.[51]

Many people are unaware that the algorithms taught for “manual” decimal arithmetic are quite different in different countries, but fortunately (and not surprisingly) the end results differ only in details of rounding and presentation. The particular model chosen was based on an extensive study of decimal arithmetic and was then evolved over several years (1979–1982) in response to feedback from thousands of users in more than forty countries. Numerous implementations have been written since 1982, and minor refinements to the definition were made during the process of ANSI standardization (1991–1996).[52]

This base floating-point model has proved suitable for extension to meet the additional requirements and facilities defined in ANSI/IEEE 854-1987,[53] and the full specification is, in effect, the union of the floating-point specifications of the two standards. This means that the same number system and arithmetic supports, without prejudice, both exact unrounded decimal arithmetic (sometimes called “fixed-point” arithmetic) and rounded floating-point arithmetic. The latter includes the facilities and number values which are now widespread in binary floating-point implementations.

Fundamental concepts

When people carry out arithmetic operations, such as adding or multiplying two numbers together, they commonly use decimal arithmetic where the decimal point “floats” as required, and the result that they eventually write down depends on three factors:

1. the specific operation carried out
2. the explicit information in the operand or operands to the operation
3. the information from the implied context in which the calculation is carried out (the precision required, *etc.*).

The information explicit in the written representation of an operand is more than that conventionally encoded for floating-point arithmetic. Specifically, there is:

51 For more discussion on why this is important, see the **Frequently Asked Questions** about decimal arithmetic at `http://speleotrove.com/decimal/decifaq.html`

52 See ANSI standard X3.274-1996: *American National Standard for Information Technology – Programming Language REXX, X3.274-1996*, American National Standards Institute, New York, 1996.

53 ANSI/IEEE 854-1987 – *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1987.

- an optional *sign* (only significant when negative)
- a numeric part, which may include a decimal point (which is only significant if followed by any digits)
- an optional *exponent*, which denotes a power of ten by which the numeric is multiplied (significant if both the numeric and exponent are non-zero).

The length of the numeric and original position of the decimal point are not encoded in traditional floating-point representations, such as ANSI/IEEE 754-1985,[54] yet they are essential information if the expected result is to be obtained.

For example, people expect trailing zeros to be indicated conventionally in a result: the sum `1.57 + 2.03` is expected to result in `3.60`, not `3.6`; however, if the positional information has been lost during the operation it is no longer possible to show the expected result. For some applications the loss of trailing zeros is materially significant.

Fortunately, the later standard ANSI/IEEE 854-1987, which is intended for decimal as well as binary floating-point arithmetic, does not proscribe representations which do preserve the desired information. A suitable internal representation for decimal numbers therefore comprises a sign, an integer (called the *coefficient* in this document), and an exponent (which is an integral power of ten).

Similarly, decimal arithmetic in a scientific or engineering context is based on a floating-point model, not a fixed-point or fixed-scale model (indeed, this is the original basis for the concepts behind binary floating-point). Fixed-point decimal arithmetic packages such as the BigDecimal class in Java 1.1 are therefore only useful for a subset of the problems for which arithmetic is used.

The information contained in the context of a calculation is also important. It usually applies to an entire sequence of operations, rather than to a single operation, and is not associated with individual operands. In practice, sensible defaults can be assumed, though provision for user control is necessary for many applications.

The most important contextual information is the desired precision for the calculation. This can range from rather small values (such as six digits) through very large values (hundreds or thousands of digits) for certain problems in Mathematics and Physics. Some decimal arithmetics (for example, the decimal arithmetic in the Atari Operating System) offer just one or two alternatives for precision – in some cases, for apparently arbitrary reasons. Again, this does not match the user model of decimal arithmetic; one designed for people to use must provide a wide range of available precisions.

This specification provides for user selection of precision; the representation (especially if it is to conform to the IEEE 854-1987 standard referred to above) may have a fixed maximum precision, but up to the maximum allowed by the representation the precision used for operations may be chosen by the programmer.

The provision of context for arithmetic operations is therefore a necessary precondition if the desired results are to be achieved, just as a “locale” is needed for operations involving text.

This specification provides for explicit control over several aspects of the context, including: the required *precision* (the point at which rounding is applied), the *rounding* algorithm to be used when digits have to be discarded, the range of normal numbers (which determines the bounds for overflow and underflow), and finally a set of *flags and trap-enablers* which report exceptional conditional and control how they are handled.

---

54 ANSI/IEEE 754-1985 – *IEEE Standard for Binary Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1985.

Appendix C – Changes

This appendix is not part of the specification. It documents changes to the combined arithmetic specification, including changes to the earlier two-layer specifications.

Changes with draft number 0.nn refer to changes in the original base specification since the first public draft of that specification (0.65, 26 Jul 2000).

Changes with draft number x.nn (for example, x.40) refer to changes in the original extended specification (inserted in their chronological position) since the first public draft of that specification (0.30, 9 Aug 2000).

Changes with version number 1.nn refer to changes in the combined arithmetic specification.

Changes in Draft 0.66 (28 Jul 2000)

- The rules constraining any limits applied to the *exponent* of a number (see page 9) have been added.
- Minor corrections and clarifications have been added.

Changes in Draft 0.69 (9 Aug 2000)

- A number produced by the **to-number** conversion operation has a *sign* of zero if the *coefficient* is 0; similarly, arithmetic operations cannot produce a result of -0. These rules allow concrete representations comprising two simple integers. Note that the Extended specification provides a mechanism for preserving and producing -0.
- The Exceptional conditions (see page 51) section has been extended to separate out more exceptions and to align them with IEEE 854.
- The names of some operations have been changed to achieve a consistent style.
- Minor corrections and clarifications have been added.

Changes in Draft 0.74 (27 Nov 2000)

- The rules constraining the limits applied to the *exponent* of a number (see page 9) have been corrected (E_{min} did not take into account the length of the *coefficient*).
- The rules for converting a number to a scientific string (see page 19) have been rephrased and corrected (the previous rules incorrectly converted some zero values).
- The Exceptional conditions (see page 51) section has been alphabetized, and the **Invalid**

context condition has been added.

- Minor corrections, clarifications, and additional examples have been added.

Changes in Draft 0.81 (5 Jan 2001)

- The *round-down* (truncation) rounding algorithm has been added.
- The rules constraining the right-hand operand of the **power** operation have been clarified, and the **Invalid operation** condition has been added for reporting errors.
- The rules for reporting underflow or overflow during a **power** operation to a negative power have been specified.
- The rules for preserving integers and removing insignificant zeros have been clarified.
- Minor clarifications and additional examples have been added.

Changes in Draft x.40 (14 May 2001)

- The *Exceptional conditions* section (see page 51) has been revised and sorted. Additional cases where the **Invalid operation** condition can be raised have been identified, and the **Invalid context** condition has been added.
- Subnormal numbers are explicitly permitted as operands and for results, provided that special values are also permitted.
- The string representations of NaN values have been changed to conform to recent discussions of the IEEE 754R committee (further changes may be necessary).
- Minor corrections and clarifications have been made.

Changes in Draft 0.83 (25 May 2001)

- The significand of a number has been renamed from *integer* to *coefficient*, to remove possible ambiguities.
- The **rescale** operation (see page 36) has been added, because it is available in most existing implementations in some form and is required for many formatting operations. It needs to be part of the base specification because it uses the parameters of the *context*.
- The treatment of zeros with exponents or fractional parts in the **to-number** conversion has been corrected.
- Minor clarifications and editorial changes have been made.

Changes in Draft x.41 (25 May 2001)

- The significand of a number has been renamed from *integer* to *coefficient*, to remove possible ambiguities.
- The treatment of zeros with exponents or fractional parts in **to-extended-number** has been clarified.

Changes in Draft x.43 (28 June 2001)

- The *rounded* condition (with associated signal and trap-enabler) has been added.

Changes in Draft x.52 (15 October 2001)

- The *special-values* flag in the context has been renamed *extended-values* to better reflect its effect and avoid confusion (the term *special values* refers only to infinities and NaNs).
- In order to permit more efficient implementations, the specification no longer requires that special and extended values raise an Invalid operation condition if a context with a *extended-values* of 0 is in use.
- For the same reason, extended zeros can no longer have non-zero exponents; in extended operations the precision of a zero should be ignored.
- Similarly, NaNs can no longer have a *sign* of 1. (An implementation can allow signed NaNs, but they would not be visible using the conversions specified.)
- The string conversion from `[0,sNaN]` has been changed to `"sNaN"`, as proposed in recent IEEE 754R discussions.
- Minor corrections and clarifications have been made, and additional examples have been added.

Changes in Draft 0.86 (30 October 2001)

- If the divisor of the **remainder** operation is 0, an Invalid operation condition is raised (instead of Division by zero), for compatibility with IEEE 854.
- Minor clarifications and editorial changes have been made.

Changes in Draft x.57 (28 November 2001)

- The operation of the **power** and **rescale** operators has been clarified.
- The behaviors of the Overflow and Underflow exceptional conditions have been clarified.
- The *trap-result* parameter of the *context* has been removed, as it is no longer needed for the exceptional conditions as specified.
- Additional cases where a result of `-0` is possible have been documented.
- Minor corrections and clarifications have been made, additional examples have been added, and differences from IEEE 854 have been identified.

Changes in Draft 0.87 (23 April 2002)

- The definition of the **rescale** operation has been changed so the the exponent is always set as specified, even if the coefficient is 0.
- Minor clarifications and editorial changes have been made.

Changes in Draft x.58 (23 April 2002)

- The operation of the **rescale** operator has been extended to match the base specification (the exponent is now always set as given, even if the coefficient is 0).

Changes in Draft 1.00 (5 July 2002)

This version combines the original base and extended specifications. There are necessarily extensive editorial changes. In addition, the following significant technical changes have been made:

- The **abs**, **max**, **min**, and **trim** operators have been added.
- A *precision* setting may now have a lower bound as well as an upper bound. This permits “fixed precision” implementations, for example, in hardware.
- The symbols E_{min} and E_{max} have been redefined to match the usage in IEEE 854 (that is, they now refer to the *adjusted exponent*).
- The **divide** operator no longer trims trailing zeros automatically. The **trim** operator has been added to provide this capability independently.
- The calculation of the *sign*, *coefficient*, and *exponent* has been separately detailed for addition, subtraction, multiplication, and division.
- Zero values accepted by **to-number** and produced by various operations may now have a non-zero *exponent*.
- The **rescale** operator now accepts a infinite left-hand operand. This has allowed the **round-to-integer** operator to be defined as a special case of **rescale**.
- The **power** operator is marked as “under review”; it may be redefined or removed in a later version. (It is currently included because it defines the results as presented in `power.decTest`.)

Changes in Draft 1.03 (1 September 2002)

- The specification allowed subnormal numbers to be more precise than permitted by IEEE 854. It has been changed to enforce a minimum *exponent* of E_{tiny}; this exponent will also be used when a conversion or calculation underflows to zero.
- The Underflow condition is now raised according to the IEEE 854 untrapped underflow criteria (instead of according to the IEEE 854 trapped criteria). That is, underflow is now only raised when a result is both subnormal and inexact.
- The Subnormal condition has been added, to allow detection of subnormal results even if Underflow is not raised.
- If an overflow or underflow occurs, the Overflow or Underflow conditions are raised, respectively, instead of special conversion conditions. This aligns the specification more closely with IEEE 854.
- The **power** operator has been changed to allow subnormals after raising a number to a negative power.
- The **to-number** conversion has been enhanced to round the converted *coefficient* (if necessary)

instead of raising overflow.

- Minor clarifications and editorial changes have been made.

Changes in Draft 1.06 (9 October 2002)

- The **normalize** operation has been added; it reduces a number to its shortest (coefficient) form. (This replaces the **trim** operator, which only removed trailing fractional zeros.)
- The definition of the **squareroot** operation has been simplified and now returns a result which is independent of the rounding mode in the context. This allows simpler implementations, and also allows the use of Hull and Abrham's variable-precision algorithm.[55]
- Input operands to the arithmetic operations are no longer rounded before use (this rounding, and the associated Lost digits condition, can therefore only occur in the X3.274 subset arithmetic). This change aligns the arithmetic with Java unlimited arithmetic, and also significantly simplifies hardware implementations which provide precision control.
- Minor clarifications and editorial changes have been made.

Changes in Draft 1.08 (14 November 2002)

- The description of the **compare** operation has been clarified; its result is always exact and unrounded.
- Two errors in the description of the **divide** operation have been corrected ("dividend" and "divisor" were swapped in the second While loop, and the calculation of the exponent when the dividend is zero was described incorrectly).

Changes in Draft 1.11 (21 February 2003)

- Changes have been made to accomodate the proposed decimal formats agreed by the IEEE 754r committee; in particular, E_{min} can now be $-E_{max}\pm 1$.
- The sign on a NaN is no longer required to be 0; it is now ignored (as in IEEE 754).
- A new context flag, *clamped*, has been added.
- Minor clarifications and editorial changes have been made.

Changes in Draft 1.14 (14 April 2003)

The description of the divide algorithm has been simplified (the algorithm is unchanged), and minor editorial corrections have been made.

Changes in Draft 1.20 (12 May 2003)

The following changes have been made to improve the consistency of some operations:

- The condition raised when the result of a **rescale** operation cannot fit has been changed from Overflow to Invalid operation. This better fits the description in IEEE 854, and avoids the

55 See *Properly Rounded Variable Precision Square Root*, T. E. Hull and A. Abrham, ACM Transactions on Mathematical Software, Vol 11 #3, pp229-237, ACM, September 1985.

problem of Overflow rebiasing if the condition is trapped and an implementation provides the alternative result.

- The description of the **square-root** operation has been expanded and changed to follow the same rules as the divide operator (briefly, the result with an exponent closest to the ideal exponent is used if the result is exact, otherwise the result will have *precision* digits).
- The result after dividing a zero by a non-zero divisor has been redefined to match the general rules for the **divide** operation. These rules are now included as a note.

Changes in Draft 1.30 (11 June 2003)

Following discussions at the May 2003 IEEE 754 revision committee meeting, the following changes have been made:

- The term *quantum* has been introduced. This is the value of a unit in the least significant digit of the coefficient of a finite number.
- The **rescale** operation has been renamed **quantize**. This has identical semantics except that the second operand specifies the desired quantum by example, which allows a faster implementation in most cases.
- The ideal exponent for the **square-root** operator is now floor(e/2), where e is the exponent of the operand. This is a better match to actual behavior of algorithms.

Changes in Draft 1.32 (23 July 2003)

Following discussions at the June 2003 IEEE 754 revision committee meeting, the following change has been made:

- The **round-to-integer** operator has been replaced by the **round-to-integral-value** operator (see page 39).

  The latter is more forgiving (no flags are set and no error is possible unless the operand is signaling NaN, and infinite values are allowed) but does not guarantee that the exponent of a finite result is 0 (it may be positive). To convert to an integer where the exponent must be 0, use the **quantize** operator.

Changes in Draft 1.33 (27 August 2003)

- The **same-quantum** operator has added; this checks that two numbers have the same quantum (exponent) and is the operator now in the draft IEEE 754 revision.
- The definition of **to-engineering-string** has been enhanced to match the JSR-13 proposed final specification: zeros preserve the original exponent.

Changes in Draft 1.36 (10 September 2003)

- The notion of optional diagnostic information associated with NaNs, as described in IEEE 854 §6.2., is now formalized. In particular, such information has limits, is propagated through numeric operations, and has a specific string representation which preserves one-to-one mapping.

- Similarly, signs are permitted on NaNs and are propagated in the same way as diagnostic information.
- Minor clarifications and editorial changes have been made.

Changes in Draft 1.37 (2 October 2003)

The **quantize** operator with two infinite arguments is no longer an Invalid operation, consistent with the **same-quantum** operator.

Changes in Draft 1.40 (15 March 2004)

- The **quantize** operator will never raise Underflow.
- Minor clarifications and editorial changes have been made.

Changes in Draft 1.45 (2 August 2004)

- The **max** and **min** operations follow the rules in the current IEEE 754 revision draft:
  - if one operand is a quiet NaN and the other is number, then the number is always returned
  - if both operands are finite and equal in numerical value then an ordering is applied: if the signs differ then **max** returns the operand with the positive sign and **min** returns the operand with the negative sign; if the signs are the same then the exponent is used to select the result.
- Minor clarifications and extra examples have been added.

Changes in Draft 1.50 (9 December 2005)

- The **exp** operation (see page 29) for raising *e* to a power has been added.
- The **ln** natural logarithm (see page 31) and **log10** base 10 logarithm (see page 31) operations have been added.
- The **power** operation (see page 35) has been redefined to allow raising a number to a non-integral power.
- Minor clarifications and editorial changes have been made.

Changes in Draft 1.51 (31 March 2006)

- Minor editorial changes have been made.

Changes in Draft 1.66 (13 March 2007)

The changes in this version add most of the new functionality required to comply with the draft IEEE 754 revision.

- The following seven operations have been added to the *Arithmetic operations* section: **compare-signal**, **fused-multiply-add**, **max-magnitude**, **min-magnitude**, **next-minus**, **next-plus**, **next-toward**, **round-to-integral-exact**.

- Twenty-seven miscellaneous operations have been added, in a new *Miscellaneous operations* section. These are non-arithmetic, but their results can all be expressed as decimal numbers or strings.
- The **same-quantum** operation has been moved to the *Miscellaenous operations* section.
- Minor clarifications and editorial changes have been made.

Note that some of these operations could still change their definition as the IEEE 754r draft is still in ballot.

Changes in Draft 1.68 (23 July 2008)

- The IEEE 754 standard was approved in June 2008, so some aspects of this document are no longer proposals and have been updated to reflect the new status of the standard.
- IEEE 754 has added the constraint that $E_{min} = 1 - E_{max}$.
- The requirements of IEEE 854 for the use of the terms *single precision* and *double precision* have been removed because in the last 21 years these terms have become synonymous with particular sizes of encodings (32-bit and 64-bit respectively).
- The rounding mode *round-05up* has been added. This permits arithmetic at shorter lengths to be carried out in a fixed-precision environment without double rounding.
- The **max-magnitude** and **min-magnitude** operations have been changed to match the operations in IEEE 754.
- The **normalize** operation has been renamed **reduce** to avoid confusion with *normal numbers*.
- All references to the General Decimal Arithmetic website have been updated to `http://speleotrove.com/decimal` (its new location).
- Various clarifications and editorial changes have been made.

Changes in Version 1.70 (25 Mar 2009)

The document is now formatted using OpenOffice (generated from GML), for improved PDF files with bookmarks, hot links, *etc.* There are no technical changes.

Index

A

abs
  definition 26
  see copy-abs 44
absolute value
  see abs 26, 44
  see copy-abs 44
abstract representation
  of context 13
  of numbers 9
  of operations 12
acknowledgements 5
add
  definition 26
  in fused-multiply-add 30
  in subset 56
adjusted exponent 10, 19
Algorism 9
algorithms, rounding 13
and
  definition 41
ANSI standard
  for REXX 5, 9, 55, 59
  IEEE 754-1985 9, 60
  IEEE 754-2008 9
  IEEE 854-1987 5, 9, 59
  X3.274-1996 5, 9, 55, 59
Arabic digits
  in numeric strings 17, 18
arbitrary precision arithmetic 23
arithmetic 23
  comparisons 27, 32, 33
  decimal 5
  errors 51
  exceptions 51
  lost digits 56
  operation rules 23
  overflow 53
  precision 13
  setting exponent 36
  testing exponent 48
  underflow 53

B

banker's rounding 13
basic default context 16
BCD conversions 17
binary coded decimal
  see BCD 17
binary floating-point conversions 17
binary integer conversions 17
blank
  in numeric strings 18

C

calculation
  context of 59
  operands of 59
  operation 59
canonical
  definition 42
canonical encoding 42, 44
checkQuantum
  see same-quantum 48
clamped 15
clamped exponents 51
class
  definition 42
coefficient 9
  concept 60
  in abstract numbers 9
  limits 9
  rotation 47
  shifting 49
comparative operations 27, 32, 33, 42, 43
compare
  definition 27
compare-signal
  definition 27
compare-total
  definition 42

compare-total-magnitude
  definition 43
comparison
  of all values 42, 43
  of numbers 27, 32, 33
concrete representation 9
conditions
  precedence of 54
conditions, exceptional 51
context 9
  abstract representation 13
  basic default 16
  defaults 16
  extended default 16
  invalid 52
  of calculation 59
conversion 17
  BCD 17
  binary floating-point 17
  binary integer 17
  errors 51
  from numeric string 21
  inexact 52
  operations 17
  rounded 53
  subnormal 53
  to engineering numeric string 20
  to scientific numeric string 19
  to scientific string 61
copy
  definition 43
  see copy-abs 43
  see copy-negate 43
  see copy-sign 43
copy operations 41
copy-abs
  definition 44
  see abs 44
copy-negate
  definition 44
  see minus 44
copy-sign
  definition 44

D

decapitation 17
decimal arithmetic 5, 23
  ANSI X3.274 subset 55
  concepts 59
  FAQ 59
decimal digits
  in numeric strings 17
decimal operations 41
decimal specification 5
default contexts 16
diagnostic NaN 11, 18, 19
diagnostic NaN
  string representation of 18
digit
  in numeric strings 17
digit-wise bits 41
divide
  definition 27
  in subset 56
divide-integer
  definition 29
division
  by zero 51
  impossible 51
  undefined 51
division-by-zero 15
double precision 8

E

e 29, 31
Emax 10, 48
Emin 10, 11
engineering notation 20
errors during arithmetic 51
Etiny 11
European digits
  in numeric strings 17
exceptional conditions 51
  in subset 57
exceptions 51
  clamped 51
  conversion syntax 51
  division by zero 51
  division impossible 51
  division undefined 51
  during arithmetic 51
  inexact 52
  insufficient storage 52
  invalid context 52
  invalid operation 52
  lost digits 56
  overflow 53
  precedence of 54
  rounded 53
  subnormal 53
  underflow 53
exclusions 7
exp
  definition 29
3.exponent 10
  adjusted 10, 19

changing 36
concept 60
extraction 47
in abstract numbers 10
in numeric strings 18
limits 10, 61
part of an operand 60
scaling 48
testing 48
exponential notation 18, 19
exponentiation 29
exp definition 29
power definition 35
extended default contexts 16

F

FAQ, decimal 59
finite numbers 9
string representation of 17
flags 15
fold-down 51
fused-multiply-add
definition 30
in subset 57

I

IEEE remainder 38
IEEE standard 754-1985 9, 60
IEEE standard 754-2008 5, 7
IEEE standard 854-1987 5, 9, 59
inclusions 7
inexact 15
infinity 11
sign of 11
string representation of 18
integer
preservation 55, 62
integer arithmetic 23
integer divide 29
invalid context 52
invalid operation 52
invalid-operation 15
invert
definition 44
is-canonical
definition 44
is-finite
definition 45
is-infinite
definition 45
is-NaN
definition 45
is-normal
definition 45
is-qNaN
definition 46
is-signed
definition 46
is-sNaN
definition 46
is-subnormal
definition 46
is-zero
definition 46

L

left shift 49
limits
coefficient 9
exponent 10
precision 13
ln
definition 31
log10
definition 31
logarithm
base 10 31
ln definition 31
log10 definition 31
natural 31
logb
definition 47
logical operands 41
logical operation 41
and 41
invert 44
or 47
xor 49
lost-digits
in abstract context 56
in subset 56

M

max
definition 32
in subset 56
max-magnitude
definition 32
min
definition 32
in subset 56
min-magnitude
definition 33
minus
definition 33
see copy-negate 44

minus zero
    see negative zero 9
miscellaneous operations 41
model 9
modulo
    see remainder operator 37
multiply
    definition 33
    in fused-multiply-add 30

N

NaN
    changes 62
    diagnostic 11, 18, 19
    payload 11
    quiet 11
    sign of 11
    signaling 11
    string representation of 18
negate
    see copy-negate 44
negation
    see minus 33
negative zero 9, 11, 23, 61
negative zero
    in to-number 21
next-minus
    definition 34
next-plus
    definition 34
next-toward
    definition 34
nextAfter
    see next-toward 34
nextDown
    see next-minus 34
nextUp
    see next-plus 34
non-European digits
    in numeric strings 17
normal
    definition 45
    numbers 11
normalize
    see reduce 37
notation
    for abstract representations 12
numbers 9, 23
    abstract representation 9
    arithmetic on 23
    changing quantum 36
    comparison of 27, 32, 33
    finite 9
    from strings 17
    notation for 12
    quantum 10
    string representation of 17
    testing quantum 48
    value of 10
numeric
    part of a numeric string 18
    part of an operand 60
numeric string 17
    syntax 17
    white space in 18

O

objectives 7
operand
    of calculation 59
    rounding of 56
operations 9, 59
operations
    abstract representation 12
    arithmetic 23
    conversion 17
    logical 41
    miscellaneous 41
or
    definition 47
ordering of values
    see total order 42
overflow
    arithmetic 15, 53, 62
    in operations 24
    in to-number 21

P

payload of a NaN 11
period
    in numeric strings 19
plain numbers
    see numbers 23
plus
    definition 33
power
    checking 62
    definition 35
    in subset 56, 57
precedence of exceptions 54
precision 13
    arbitrary 23
    double 8
    in abstract context 13
    limits 13
    of a calculation 60

of arithmetic 13
single 8

Q

quantize
definition 36
quantum
changing 36
of a number 10
testing 48
quiet NaN 11
string representation of 18

R

radix
definition 47
reduce
definition 37
remainder
definition 37
IEEE 38
remainder-near
definition 38
rescale
see quantize 36
residue
see remainder operator 37
restrictions 8
result
rounding of 24
right shift 49
rotate
definition 47
round-05up algorithm 14
round-ceiling algorithm 13
round-down algorithm 13, 62
round-floor algorithm 14
round-half-down algorithm 14
round-half-even algorithm 13
round-half-up algorithm 13
round-to-integral-exact
definition 39
round-to-integral-value
definition 39
round-up algorithm 14
rounded 15, 23
rounding 13
in abstract context 13
in operations 23
in subset 57
in to-number 21
of operands 56
of results 24
round-05up 14
round-ceiling 13
round-down 13
round-floor 14
round-half-down 14
round-half-even 13
round-half-up 13
round-up 14
to decimal places 36
to integral value 36, 39
to nearest 13

S

same-quantum
definition 48
scaleb
definition 48
scientific notation 19
scope 7
shift
definition 49
shortest form
see reduce 37
sign 9
concept 60
in abstract numbers 9
in numbers 21
in numeric strings 17-19
of an operand 60
of special values 11
signaling NaN 11
string representation of 18
signals 15
significant digits, in arithmetic 13
simple number
see numbers 23
single precision 8
sorting values
see total order 42
special values 9, 11, 23, 55
special values
in numeric strings 21
string representation of 18
square-root
definition 39
strings 17
subnormal 11, 12, 15
in operations 23
in to-number 21
numbers 11
subnormal numbers 23
subset, arithmetic 55
subtract

definition 26
in subset 56

T
to-engineering-string operation 20
to-number operation 21
to-scientific-string operation 19
total order 42
by magnitude 43
trailing zeros 25, 37, 55, 56
trap-enablers 15

U
ulp 10, 30, 31
underflow
arithmetic 15, 53, 62
in operations 11, 23
in to-number 21
unit in last position 10

V
value of a number 10

W
white space
in numeric strings 18

X
xor
definition 49

Z
zero
division by 51
in operations 23
in subset 56
negative 9, 11, 21, 55
trailing 37
underflow 12

-
- (minus)
in numbers 21
in numeric strings 17-19

.
. (period)
in numeric strings 19

+
+ (plus)
in numbers 21
in numeric strings 17-19