General Decimal Arithmetic Specification

14th November 2002

Mike Cowlishaw

IBM Fellow
IBM UK Laboratories
mfc@uk.ibm.com

Version 1.08

Table of Contents

Introduction

This document defines a general purpose decimal arithmetic. A correct implementation of this specification is a decimal arithmetic which conforms to the requirements of the ANSI/IEEE standard 854-1987,[1] while supporting integer and unrounded floating-point arithmetic as a subset.

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. Additional explanatory material can be found in the article *A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic*.[2]

Appendix A (see page 40) 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 43) summarizes the design concepts behind the decimal arithmetic.

Appendix C (see page 46) summarizes changes to this specification.

This document in various softcopy formats, together with a reference implementation, testcases, proposed concrete representations, and background information may be found at `http://www2.hursley.ibm.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 current IEEE 754R (revision) committee. Special thanks for their contributions to the current design and this document are due to Aahz, Joshua Bloch, Dirk Bosmans, Paul-Georges Crismer, Joe Darcy, John Ehrman, Kit George, Peter Golde, Brian Marks, Dave Raggett, and Fred Ris.

---

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

2 by W. J. Cody *et al*, published in the IEEE Micro magazine, August 1984, pp86–100.

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 will conform to the decimal arithmetic defined in ANSI/IEEE standard 854-1987,[4] except for some minor restrictions (see page 3), 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 854-1987 – *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1987.

5 Sometimes called “fixed-point” decimal arithmetic.

6 Some proposed representations can be found in:
`http://www2.hursley.ibm.com/decimal/deccode.pdf`

Restrictions

This specification deviates from the requirements of IEEE 854 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. This specification does not require that values returned after overflow and underflow change if the exception trap-enabler is set, and the criteria for underflow similarly do not change if its trap-enabler is set (the Subnormal condition has been added to allow the alternative underflow condition to be detected).[8]
3. The string representations of NaN values are `"NaN"` and `"sNaN"`, instead of just `"NaN"` with an optional sign.[9]

Note that all other requirements of IEEE 854 (such as subnormal numbers and the negative zero) are included in this specification.

---

[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 854 in this respect.

[8] This follows the current proposals of the IEEE 754R committee which is revising IEEE 754 and IEEE 854. Dependence on the trap-enabler setting is difficult to make thread-safe, and also the IEEE 854 definition does not generalize to the power operator (the required replacement value may itself overflow or underflow). An implementation may provide the IEEE values, if feasible, to a trap handler via some separate mechanism, but this is not required by this specification.

[9] This follows the current proposals of the IEEE 754R committee which is revising IEEE 754 and IEEE 854.

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 and ANSI X3-274).

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),[10] 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.

[10] Indeed, some variations of operations could be selected by using context settings outside the scope of this specification.

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 may be 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.[11]

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*. It is recommended that this limit 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).

   If the coefficient has a maximum length then it is required[12] that E_{limit} be greater than $5 \times$ `mlength`, where `mlength` is the maximum length of the *coefficient* in decimal digits. It is recommended that E_{limit} be greater than $10 \times$ `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,[13] 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 the same magnitude. E_{max} will always equal E_{limit} (the largest value of the exponent) and E_{min} will always equal $-E_{max}$.

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

---

[11] That is, the maximum value of the *coefficient* will be an integral power of ten, less one – for example, 9999999999999999999.

[12] See IEEE 854 §3.1.

[13] 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).

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 –1007 through +991. This would allow positive values of the number to range from 1.23456789E–999 through 1.23456789E+999.

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

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 twos complement or biased binary integer. Some possible concrete representations are described in detail at: `http://www2.hursley.ibm.com/decimal/deccode.html`

2. 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 16) 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.

   No such constraint applies to the 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 854 §3.1).

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 854 §6.1)

2. *quiet NaN* – a value representing undefined results ("Not a Number") which does not cause an Invalid operation condition. IEEE 854 recommends that additional diagnostic information be associated with quiet NaNs (see IEEE 854 §6.2)

3. *signaling NaN* – a value representing undefined results ("Not a Number") which will cause an Invalid operation condition if used in any operation defined in this specification (see IEEE 854 §6.2).

When a number has one of these special values, its *coefficient* and *exponent* are undefined.[14] The *sign*, however, is significant (that is, it is possible to have both positive and negative infinity, and the *sign* of a NaN is always 0).

[14] 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.

Subnormal numbers and Underflow

Numbers whose adjusted exponents are less than E_{min} are called *subnormal* numbers.[15] These 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 $-E_{limit}-(precision-1)$, called E_{tiny}, where *precision* is the working precision, as described below (see page 9). The result will be rounded, if necessary, to ensure that the exponent is no smaller than E_{tiny}. If, during this rounding, the result 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).

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, duples are used to indicate the special values. These have the form [*sign*, *special–value*], 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.

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

[15] IEEE 854 defines *subnormal* numbers as numbers whose absolute value is non-zero and is closer to zero than ten to the power of E_{min}. This definition includes zeros with tiny exponents.

Abstract representation of operations

The core operations which must be provided by an implementation are described in later sections which define Conversions (see page 14) and Arithmetic Operations (see page 21). 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). It 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.

An implementation must designate a precision to be known as *single precision* (see IEEE 854 §3.2.1). This must be greater than 5 (see IEEE 854 §3.1) and within the range of implemented precisions. It is recommended that it be at least 9.[16]

An implementation may also designate a precision to be known as *double precision*, which must be within the range of implemented precisions (see IEEE 854 §3.2.2). If a double precision is designated, then the following constraints apply:

- If the value of *single precision* is given by P_s, and the value of *double precision* is given by P_d, then P_d must be greater than or equal to $2 \times P_s + 1$ (see IEEE 854 §3.2.2).
- The maximum *exponent* (E_{limit}) at the designated single precision must be at least 1 less than the E_{limit} at double precision, divided by 8 (see IEEE 854 §3.2.2).[17]

If these constraints cannot be implemented (for example, an implementation may support very large exponents and not be able to have different exponent limits for differing precisions), then a double precision must not be designated.

---

[16] This is the "narrowest basic precision" described in IEEE 854 §3.2.1. Strictly speaking, *single precision* should be the narrowest precision supported; however it is assumed that when precision is fully variable the intent of IEEE 854 is that the designation applies to the narrowest *default* precision – the programmer is permitted to specify a narrower precision explicitly.

[17] This constraint is very slightly tighter than that defined by IEEE 854, which specifies that E_{limit} for double be greater than or equal to $8 \times E_{limit}$ for double, plus 7. There is a preference for human-oriented limits, so it is suggested that the E_{limit} for single be one tenth of, or one digit shorter than, the E_{limit} for double.

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 following rounding algorithms are defined and must be supported:[18]

round-down

(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 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 should be incremented by 1 (rounded up). If they represent less than half, then the result is not adjusted (that is, the discarded digits are ignored).

Otherwise (they represent exactly half) the result 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 should be incremented by 1 (rounded up). If this would cause overflow then the result will be `[0,inf]`.

round-floor

(Round toward −∞.) 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 coefficient should be incremented by 1. If this would cause overflow then the result will be `[1,inf]`.

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 36).

---

[18] 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”.

flags and trap-enablers

The exceptional conditions (see page 36) 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 854 §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:

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

This signal cannot occur, and is therefore optional, in an implementation where the lower bound for *precision* is equal to the maximum length of the *coefficient*.

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.[19]

[19] IEEE 854 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).

Notes:

1. The setting of *precision* may be used to reduce a result from double to single precision, using the **plus** operation. This meets the requirements of IEEE 854 § 4.3.
2. IEEE 854 was 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 be set less than 6.
4. For completeness, it is recommended that implementations also offer two further rounding modes: *round–half–down* (round to nearest, where a 0.5 case is rounded down) and *round–up* (round away from zero).
5. The concrete representation of *rounding* is often a series of integer constants, or an enumeration, held in an object or control register.
6. 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 two *default contexts*, which define suitable settings for basic arithmetic and for the extended arithmetic defined by IEEE 854. 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 context

In the *extended default context*, 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 designated *single precision*
- *rounding* – is set to *round–half–even* (IEEE 854 §4.1)

It is recommended that if a *double precision* is designated then a third *extended double default context* be provided, with the same settings as the extended default context except that the *precision* is set to the double precision.

Conversions

This section defines the required conversions between the abstract representation of numbers and string (character) form.[20] Two number-to-string conversions and one string-to-number conversion are defined.

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-Arabic 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. This meets the requirements of IEEE 854 §5.3.

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 is effected by using the **round–to–integer** operation (see page 32). This meets the requirements of IEEE 854 §5.4 and §5.5.

[20] See also IEEE 854 §5.6.

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 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 may be preceded by an optional sign, as for finite numbers, and the second two represent *quiet NaN* and *signaling NaN* respectively).

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

Formally:[21]

```
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' | 'sNaN'
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | nan
```

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

[21] 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.

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
```

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. 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).

Finally, the entire string is prefixed by a minus sign character[22] ("–") if *sign* is 1. No sign character is prefixed if *sign* is 0.

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

- If the *special value* is *quiet NaN* then the resulting string is "`NaN`".
- If the *special value* is *signaling NaN* then the resulting string is "`sNaN`".[23]
- If the *special value* is *infinity* then the resulting string is "`Infinity`". In this case, if the *sign* of the number is 1 then the string is preceded by a "–" character. Otherwise (the *sign* is is 0) no sign character is prefixed.

Examples:

For each abstract representation [*sign*, *coefficient*, *exponent*] or [*sign*, *special–value*] 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,inf]          "Infinity"
[1,inf]          "-Infinity"
[0,qNaN]         "NaN"
[0,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 19) 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 currently under consideration by the IEEE 754 review committee.

---

[22] 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.

[23] This is a deviation from IEEE 854-1987 (see Notes).

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 7).

4. IEEE 854 allows additional information to be suffixed to the string representation of special values. Any such suffixes are not permitted by this specification (again, to preserve the one-to-one mapping). It is suggested that if additional information is held in a concrete representation then a separate mechanism or operation is provided for accessing that information.

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, 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.

If after the adjustment the decimal point would not be followed by a digit then it is not added. 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"
```


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 15).

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.

  If the value of the *adjusted exponent* (see page 5) is less than E_{min}, then the number is subnormal (see page 7). In this case, the calculated coefficient and exponent form the result, unless the value of the *exponent* (see page 7) is less than E_{tiny}, in which case the *exponent* will be set to E_{tiny} (see page 7), 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 38) is raised.

  If (after any rounding of the coefficient) the value of the *adjusted exponent* is larger than E_{max} (see page 5), then an exceptional condition (overflow) results. In this case, the result is as defined under the Overflow Exceptional condition (see page 38), and may be infinite. It will have the *sign* as set above.

If the string represents a *special value* then:

- The string "`NaN`", independent of case, is converted to *quiet NaN*, with *sign* 0.
- The string "`sNaN`", independent of case, is converted to *signaling NaN*, with *sign* 0.
- The strings "`Infinity`" and "`Inf`", optionally preceded by a sign character and independent of case, will be converted to *infinity*. In this case, the *sign* of the number is set to 1 if the string is preceded by a "–". Otherwise the *sign* is set to 0.

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*] or [*sign*, *special–value*] 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]
"SNaN"          [0,sNaN]
"Fred"          [0,qNaN]
```


Arithmetic operations

This section describes the arithmetic operations on numbers, including subnormal numbers, negative zeros, and special values (see also IEEE 854 §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 either one or two 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.

- 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 5) of the result is less than E_{min}, then an exceptional condition (subnormal) results. In this case, the calculated coefficient and exponent form the result, unless the value of the *exponent* (see page 5) is less than E_{tiny}, in which case the *exponent* will be set to E_{tiny}, the coefficient will be rounded (possibly to zero) to match the adjustment of the exponent, and the *sign* is unchanged. If this rounding gives an inexact result then the Underflow Exceptional condition (see page 38) is raised.

If the value of the *adjusted exponent* of the result is larger than E_{max} (see page 5), then an exceptional condition (overflow) results. In this case, the result is as defined under the Overflow Exceptional condition (see page 38), 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 37).
- *signaling NaNs* always raise the Invalid operation condition when used as an operand to an arithmetic operation. The result in this case is `[0,qNaN]`.
- The result of any arithmetic operation which has an operand which is a NaN (a *quiet NaN*, or *signaling NaNs* when the *invalid–operation* trap enabler is 0) is `[0,qNaN]`. In this case, the signs of the operands are ignored (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 and neither is a NaN.
- The *sign* of the result of an addition or subtraction will be 1 only if the result is less than zero and neither operand is a NaN, 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 under the following conditions only:
  - a result is rounded to zero, and the value before rounding had a *sign* of 1.
  - the operation is an addition of `[1,0,0]` to `[1,0,0]`, or a subtraction of `[0,0,0]` from `[1,0,0]`
  - 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 `[1,0,0]`, and the right-hand operand is odd (and positive).
  - the operation is **rescale** or **round–to–integer**, the left-hand operand is negative, and the magnitude of the result is zero. In the case of **rescale** the final exponent may also be non-zero.
  - the operation is **square–root** and the operand is `[1,0,0]`.

---

[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.

Examples involving special values:

```
add('Infinity', '1')        ==>  'Infinity'
add('NaN', '1')             ==>  'NaN'
subtract('1', 'Infinity')   ==>  '-Infinity'
multiply('-1', 'Infinity')  ==>  '-Infinity'
subtract('-0', '0')         ==>  '-0'
multiply('-1', '0')         ==>  '-0'
divide('-1', 'Infinity')    ==>  '-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. Quiet NaNs are permitted to propagate diagnostic information pertaining to the origin of the NaN (see IEEE 854 §6.2). Any such diagnostic information, and the means by which it is propagated, is outside the scope of this specification.
3. The rules above imply that the **compare** operation can return a quiet NaN as a result, which indicates an "unordered" comparison (see IEEE 854 §5.7).
4. 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 854 §5.7 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 37).
5. If a result is rounded, remains finite, and is not subnormal, its coefficient will have exactly *precision* digits (except after the **rescale**, **round–to–integer**, or **square–root** operations, as described below). That is, only unrounded or subnormal coefficients can have fewer than *precision* digits.
6. Trailing zeros are not removed after operations. That is, results are unnormalized.

abs

abs takes one operand. If the operand is negative, the result is the same as using the **minus** operation (see page 28) on the operand. Otherwise, the result is the same as using the **plus** operation (see page 28) on the operand.

Examples:

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

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 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.

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'
```

Note that the result of a compare is always exact and unrounded.

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, with the division being complete when either *precision* digits have been accumulated or the remainder from a subtraction in the division is zero, 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):
    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:

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

- 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 the *coefficient* of the result is zero and `adjust` is set as described in step 1 above, calculated as if the dividend coefficient were 1. (It will then have a value which is one less than the length of the coefficient of the divisor.)

- 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'
```

[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.

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 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).
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'
```


max

max takes two operands, compares their values numerically, and returns the maximum. If either operand is a NaN then the general rules apply.

Otherwise, the operands are compared as as though by the **compare** operation (see page 25). If they are numerically equal then the left-hand operand is chosen as the result. Otherwise the maximum (closer to positive infinity) of the two operands is chosen as the result. In either case, the result is the same as using the **plus** operation (this page) on the chosen operand.

Examples:

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

min

min takes two operands, compares their values numerically, and returns the minimum. If either operand is a NaN then the general rules apply.

Otherwise, the operands are compared as as though by the **compare** operation (see page 25). If they are numerically equal then the left-hand operand is chosen as the result. Otherwise the minimum (closer to negative infinity) of the two operands is chosen as the result. In either case, the result is the same as using the **plus** operation (this page) on the chosen operand.

Examples:

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

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'
```


multiply

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

Otherwise, the 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 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'
```

normalize

normalize takes one operand. It has the same semantics as the **plus** operation, except that the final result is reduced to its simplest form, with all trailing zeros removed.

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. Alternatively, if the *coefficient* is zero the *exponent* is set to 0. In all cases the *sign* is unchanged.

Examples:

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


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 remainder operation differs from the remainder operation defined in IEEE 854 (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 854 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 §5.1.)

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).[27]

[27] This is a deviation from IEEE 854, necessary to assure realistic execution times when the operands have a wide range of exponents.

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 `'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”.

rescale

rescale takes two operands. If either operand is a *special value* then the general rules apply (and infinities are unchanged), except that if the right-hand operand is infinite, an Invalid operation condition (see page 37) is raised, and the result is `[0,qNaN]`.

Otherwise, it 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 the value of the second (right-hand) operand.

The right-hand operand must be a whole number whose integer part (after any exponent has been applied) is no more than E_{max} and no less then E_{tiny}, and whose fractional part (if any) is all zeros.

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 the right-hand operand).

Unlike other operations, if the length of the *coefficient* after the rescaling would be greater than *precision* then an Overflow condition results. This guarantees that, unless there is an error condition, the *exponent* of the result of a rescale is always the value specified by the right-hand operand.

Examples:

```
rescale('2.17', '-3')          ==>  '2.170'
rescale('2.17', '-2')          ==>  '2.17'
rescale('2.17', '-1')          ==>  '2.2'
rescale('2.17', '0')           ==>  '2'
rescale('2.17', '1')           ==>  '0E+1'
rescale('2', 'Infinity')       ==>  'NaN'
rescale('-0.1', '0')           ==>  '-0'
rescale('-0', '5')             ==>  '-0E+5'
rescale('+35236450.6', '-2')   ==>  'Infinity'
rescale('-35236450.6', '-2')   ==>  '-Infinity'
rescale('217',  '-1')          ==>  '217.0'
rescale('217',  '0')           ==>  '217'
rescale('217',  '1')           ==>  '2.2E+2'
rescale('217',  '2')           ==>  '2E+2'
```

Note that in the penultimate example the number is `[0,22,1]`, leading to the string in scientific notation as shown.

round-to-integer

round–to–integer takes one operand. Its result is the same as using the **rescale** operation using the given operand as the left-hand-operand and 0 as the right-hand-operand.[28]

Examples:

```
round-to-integer('2.1')     ==>  '2'
round-to-integer('100')     ==>  '100'
round-to-integer('100.0')   ==>  '100'
round-to-integer('101.5')   ==>  '102'
round-to-integer('-101.5')  ==>  '-102'
round-to-integer('10E+5')   ==>  '1000000'
```

Note: IEEE 854 refers to §4 for this operation, but then implies that *round–half–even* rounding should always be used (whereas §4 specifically allows directed rounding). It is assumed that it was not intended to exclude directed rounding.

[28] This operation is defined in order to provide the Round Floating-Point Number to Integral Value operator described in IEEE 854 §5.5.

square-root

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

Otherwise, the operand must be greater than or equal to 0. If the value of the operand is –0 then the result is `[1,0,0]`.

Otherwise, the result is the exact square root of the operand, rounded according to the setting of *precision* using the *round–half–even* algorithm, and then normalized (as though by the **normalize** operation).

Examples:

```
square-root('0')      ==> '0'
square-root('-0')     ==> '-0'
square-root('0.39')   ==> '0.6244998'
square-root('1.00')   ==> '1'
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 result of this operation is normalized because an unnormalized result with an integer coefficient cannot always be defined (*e.g.*, `square-root('4.0')`); this normalization may cause the Rounded condition.

power

> *The following operation is under review. It will probably either be removed or be changed to simply state that the the result must be within one ulp. The definition in this section is included as it defines the results for the* `power` *testcase group and for the reference implementation.*

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

Otherwise, the right-hand operand must be a whole number whose integer part (after any exponent has been applied) has no more than 9 digits and whose fractional part (if any) is all zeros before any rounding. The operand may be positive, negative, or zero; if negative, the absolute value of the power is used, and the left-hand operand is inverted (divided into 1) before use.

For calculating the power, the number (left-hand operand) is in theory multiplied by itself for the number of times expressed by the power.

In practice (see the note below for the reasons), the power is 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 initial division are done under the normal arithmetic operation and rounding rules, using the context 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.[29]

If the increased precision needed for the intermediate calculations exceeds the capabilities of the implementation then an **Invalid operation** condition is raised.

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.[30]

In addition:

- If both operands are zero, an Invalid operation condition (see page 37) results.
- If the right-hand operand is infinite, an Invalid operation condition (see page 37) 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 infinite if the right-hand side is positive, 1 if the right-hand side is zero, and zero if the right-hand side is negative. The *sign* of the result will be 0 if the right-hand-side is even (or zero), or will be the same as the *sign* of the left-hand-side if the right-hand-side is odd.
- If the operation overflows or underflows, the *sign* of the result will be 0 if the right-hand-side is even, or will be the same as the *sign* of the left-hand-side if the right-hand-side is odd.

---

[29] The precision specified for the intermediate calculations ensures that 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.

[30] It can only be halted early if the result becomes zero.

Examples:

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

Notes:

1. The result of the power operator is negative if (and only if) the left-hand operand is negative and the right-hand operand is odd.

2. A particular algorithm for calculating 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 must be defined here so that different implementations will give identical results for the same operation on the same values. Other algorithms for this (and other) operations may always be used, so long as they give identical results to those described here.

3. Mathematical and transcendental functions are outside the scope of this specification. However, 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. In this case it is not required that the more general function return identical results to the operation described above.

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 12) 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 854 §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:

Conversion syntax

This occurs and signals *invalid–operation* if an string is being converted to a number and it does not conform to the numeric string syntax (see page 15). The result is `[0,qNaN]`.

Division by zero

This occurs and signals *division–by–zero* if division 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 sign of the dividend.

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.[31]

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 `[0,sNaN]` (*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]`
- the right-hand operand of the **rescale** operation has a non-zero fractional part, or is outside the permitted range
- 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 the right-hand operand has a non-zero fractional part, or has more than 9 digits, 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]`.

---

[31] 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.

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) would be greater than the largest value that can be handled by the implementation (the value E_{max}). It also occurs if a **rescale** operation would require greater precision than is available.

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*, 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.

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.[32]

[32] See IEEE 854 §7.4.

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.

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.

The 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, and any Inexact trap takes precedence over Rounded.

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 854 §8.1).

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*[33] 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 **rescale**) the *exponent* is set to 0.[34]
- 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).

---

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

[34] 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 twos complement form.

- After any operation and the rounding of its result (unless the operation is **rescale**), 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).[35]

Operation differences

In the subset arithmetic, operands are rounded before use if necessary (as in Numerical Turing[36] 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.
- 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).

---

[35] The rule 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.

[36] 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.

- If both operands to a **power** operation are zero then the result is 1 (instead of being an error).
- If the right-hand operand to a **power** operation is negative, the left-hand operand is used as-is and the final result is inverted. The integer part of the right-hand operand must fit in *precision* digits.
- The integer part of the right-hand operand to a **rescale** operation must fit in *precision* digits.

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.

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*.[37]

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).[38]

This base floating-point model has proved suitable for extension to meet the additional requirements and facilities defined in ANSI/IEEE 854-1987,[39] 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

---

[37] For more discussion on why this is important, see the **Frequently Asked Questions** about decimal arithmetic at `http://www2.hursley.ibm.com/decimal/decifaq.html`

[38] 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.

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

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:

- 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,[40] 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 ADAR[41] or 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[42] 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.

---

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

[41] "Ada Decimal Arithmetic and Representations"
See *An Ada Decimal Arithmetic Capability*, Brosgol et al. 1993.
`http://www.cdrom.com/pub/ada/swcomps/adar/`

[42] See, for example, *The* [*Atari*] *Floating Point Arithmetic Package*, C. Lisowski.
`http://intrepid.mcs.kent.edu/%7Eclisowsk/8bit/atr11.html`

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 three aspects of the context: the required *precision* – the point at which rounding is applied, the *rounding* algorithm to be used when digits have to be discarded, and finally a set of *flags and trap–enablers* which report exceptional conditional and control how they are handled.

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 5) 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 36) 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 5) 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 16) have been rephrased and corrected (the previous rules incorrectly converted some zero values).

- The Exceptional conditions (see page 36) 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 36) 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 31) 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 a canonical 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.[43]
- 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).

[43] 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.

Index

D

E

F

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z

