Standard Decimal Arithmetic Specification

5th January 2001

Mike Cowlishaw

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

Draft – Version 0.81

Table of Contents

Introduction

This document defines a general purpose decimal arithmetic. A correct implementation of this specification will conform to the decimal arithmetic defined in the ANSI standard X3.274-1996.[1] This document describes that arithmetic in a language-independent manner, and it is also the base document for an extended arithmetic specification which also conforms to the ANSI/IEEE standard 854-1987.[2]

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

Appendix A (see page 21) summarizes the design concepts behind the decimal arithmetic. For further background details, including a suggested concrete representation which conforms to IEEE 854, please see the material at the associated web site:
`http://www2.hursley.ibm.com/decimal`

Appendix B (see page 24) summarizes the changes to this specification since the first public draft.

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, and the X3 (now NCITS) J18 technical committee. Special thanks for their contributions to the current design and this document are due to Joshua Bloch, Dirk Bosmans, Brian Marks, and Dave Raggett.

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

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

Scope

Objectives

This document defines a general purpose decimal arithmetic. A correct implementation of this specification will conform to the decimal arithmetic defined in the ANSI standard X3.274-1996.[3] Recommendations are also included where the application of additional constraints will aid conformance with the ANSI/IEEE standard 854-1987.[4]

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 operations
- 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
- The means by which operations are effected
- Concrete representations (storage format) of context information
- Extensions which permit full conformance with IEEE 854.

---

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

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

The Arithmetic Model

This specification is based on a model of decimal arithmetic which is a formalization of the arithmetic taught in schools as defined and constrained by the relevant standards (ANSI X3-274 and IEEE 854).

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 changeable parameters or rules which govern the results of a 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), nor does it define the *concrete representation* (specific layout in storage, or in a processor's register, for example) of numbers or context.[5]

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

Abstract representation of numbers

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 and 0 that the number is positive.
2. *integer* – an integer which may be zero or positive.

   In the abstract, there is no upper limit on the maximum size of the integer. In practice there may be some upper limit to the integer, in which case this limit must be expressed as an integral number of decimal digits.[6]
3. *exponent* – a signed integer which indicates the power of ten by which the integer component is multiplied.

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

[6] That is, the maximum value of the integer will be an integral power of ten, less one – for example, 99999999999999999999.

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, in which case this limit must be expressed as an integral number of decimal digits or be one of the numbers 1, 5, or 25, multiplied by an integral power of ten.

It is recommended that an E_{limit} of nine or more digits be supported. It is also recommended that E_{limit} be greater than 10 × `ilength`, where `ilength` is the length of the *integer* in decimal digits.[7]

When a limit to the exponent applies, it must result in a balanced range of positive or negative numbers,[8] taking into account the magnitude of the integer. To achieve a balanced range, the minimum and maximum values of the exponent (E_{min} and E_{max} respectively) will have different magnitudes, depending on the length of the integer. The value of E_{min} will be $-E_{limit}$–(`ilength`–1), and the value of E_{max} will be E_{limit}–(`ilength`–1), where `ilength` is again the length of the *integer* in decimal digits.

For example, if the *integer* had the value 123456789 (9 digits) and the *exponent* had an E_{limit} of 999 (3 digits), then E_{min} would be –1007 and E_{max} would be +991. This would allow positive values of the number to range from 1.23456789E–999 through 1.23456789E+999.

E_{limit}, therefore, is the maximum absolute value of the exponent of a number when that number is presented in scientific notation with one digit before any decimal point.

The numerical *value* of the number is then given by: $(-1)^{sign} \times integer \times 10^{exponent}$

Notes:

1. Many concrete representations for numbers have been used successfully. The integer is typically represented in some form of binary coded decimal (BCD) or using a base which is a higher power of ten, but it may also be expressed as a binary integer. The exponent is typically represented by a twos complement or biased binary integer. One possible concrete representation is described in detail at:
`http://www2.hursley.ibm.com/decimal/decconc.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 9) 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. For the purposes of this specification the number could have been described by two signed integers. However, one of these has been separated into a sign and non-negative integer for convenience of description and also to allow for an extended arithmetic in which a value of negative zero is a possibility.

[7] This is an IEEE 854 recommendation; IEEE 854 also requires that E_{limit} be greater than 5 × `ilength`.

[8] 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).

4. When implementing this arithmetic for use in the Rexx language, ANSI X3.274 requires that E_{limit} be at least 999999999, and that integer lengths of at least 999 digits are supported.

Notation

In later sections of this document, a specific number is described by its abstract representation, using the triad notation: [*sign*, *integer*, *exponent*].

So, for example, the triad `[0,2708,–2]` represents the number `27.08`, and the triad `[1,1953,0]` represents the integer `–1953`.

Abstract representation of operations

The core operations which must be provided by an implementation are described in later sections which define Conversions (see page 8) and Arithmetic Operations (see page 12). Each operation is given an abstract name (for example, "add"), and its semantics are strictly defined. However, the manner in which each operation 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.

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

Context is defined by three parameters:

1. *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 maximum size of the precision. In practice there may be some upper limit to it (for example, the length of the maximum *integer* supported by a concrete representation), in which case this limit must be expressed as an integral number of decimal digits.

   If a default precision is supplied by some environment, it is recommended that the default be 9.

2. *rounding* – a named value which indicates the algorithm to be used when rounding is necessary. Rounding is applied when a result *integer* needs more digits than the value of *precision*; in this case the digit to the left of the first discarded digit may be incremented by one, depending on the rounding algorithm selected and the remaining digits of the integer.

The following rounding algorithms are defined:[9]

round-down

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).

When a result is rounded, the *integer* may become longer than the current *precision*. In this case (it will be a multiple of ten) it is divided by ten, and the *exponent* incremented by one. This in turn may give rise to an overflow condition (see page 19).

This specification requires only that *round–half–up* be provided.[10] It is recommended that *round–down* and *round–half–even* also be provided.[11]

3. *lost–digits* – a value which must be either 0 or 1. If 0, an operand which has more leading significant digits in its *integer* than the *precision* setting will be rounded to *precision* digits before use, using the *rounding* algorithm. If 1, this "lost digits" condition will be treated as an Exceptional condition (see page 19).

   The lost digits test does not treat trailing decimal zeros in the *integer* as significant. 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).

Notes:

1. *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

[9] The term "round to nearest" is not used as it is ambiguous; *round–half–up* is the usual round-to-nearest algorithm used in European countries and in international financial dealings; *round–half–even* is often used in the USA.

[10] ANSI X3.274 specifies *round–half–up* as its default (only) rounding algorithm.

[11] IEEE 854 specifies *round–half–even* as its default rounding algorithm. IEEE 854 further requires that three additional rounding modes be implemented (*round–ceiling*, *round–down*, and *round–floor*).

computations affected by the context, including comparisons. To conform to IEEE 854, this value should not be set less than 6.

2. The concrete representation of *rounding* is often a series of integer constants, or enumerations, held in an object or control register.

Conversions to and from strings

This section defines the required conversions between the abstract representation of numbers and string (character) form. Two number-to-string conversions and one string-to-number conversion are defined; these are not affected by the *context*.

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.

Numeric string syntax

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

A *numeric string* is a character string that 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. No blanks or other white space characters are permitted in a numeric string.

Formally:[12]

```
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
numeric–string ::=  [sign] decimal–part [exponent–part]
```

[12] 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. */
        "17."        /* The same as 17               */
          ".5"       /* The same as 0.5              */
        "4E+9"       /* Exponential notation         */
         "0.73e-7"   /* Exponential notation         */
```

The `"4E+9"` can be considered a short way of writing `4000000000`, and the `"0.73e-7"` is short for `0.000000073`.

Notes:

1. A single period alone is not a valid numeric string.

2. Leading zeros are permitted by the definition above.

to-scientific-string – conversion to numeric string

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

The *integer* 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, an *adjusted exponent* is calculated; this is the exponent in the abstract representation, plus the number of characters in the converted *integer*, less one. That is, *exponent*+(`ilength`–1), where `ilength` is the length of the *integer* 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 integer 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 integer 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 integer (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[13] ("–") if *sign* is 1. No sign character is prefixed if *sign* is 0.

Examples:

For each abstract representation [*sign*, *integer*, *exponent*] 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"
```

Note: 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 11), 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").

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 follows the rules for conversion to scientific numeric string except in the case 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 adjusted exponent is zero then no indicator letter and exponent is suffixed.

Examples:

For each abstract representation [*sign*, *integer*, *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"
```

[13] This specification defines only the glyph representing a minus sign character. Depending on the implementation, this may correspond to a hyphen rather than to a distinguishable "minus" character.

to-number – conversion from numeric string

This operation converts a string to a number, as defined by its abstract representation.

Specifically, the string must conform to the above numeric string syntax (see page 8). 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 then the *exponent* is 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 *integer* which will be zero or positive. If the integer is zero, then the *sign* is set to 0.[14]

A numeric string to number conversion is always exact. If the value of the *integer* or *exponent* is too large for an implementation to handle (perhaps because the concrete representation has size limits) then an exceptional condition (error) must result.

Examples:

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

```
"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]
"0E+7"         [0,0,7]
"–0E+7"        [0,0,7]
"–0"           [0,0,0]
```

[14] This rule, together with the arithmetic rules, ensures that numbers with value −0 will not result from this specification (though they are permitted in the extended specification). This allows a concrete representation for this specification to comprise simply two integers in twos complement form or equivalent.

Arithmetic operations

This section describes the arithmetic operations. These are identical to those defined in ANSI X3.274, where an algorithmic definition of each operation may be found.

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, rather than as a triad. 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 be a *precision* of 9, a *rounding* setting of *round–half–up*, and a *lost–digits* setting of 0.

Arithmetic operation rules

The following general rules apply to all arithmetic operations:

If the number of decimal digits in the *integer* of an operand to an operation is greater than the current *precision* in the context then (unless the *lost–digits* condition (see page 5) is triggered) the operand is rounded to *precision* significant digits using the *rounding* algorithm described by the context before being used in the computation.

The operation is then carried out as described under the individual operations below to give an exact result. This is then rounded to *precision* digits, again using the current *rounding* algorithm, if necessary.[15]

[15] 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.

After rounding, a positive exponent is reduced to 0 (by multiplying the *integer* by $10^{exponent}$) if the resulting *integer* would have no more than *precision* digits.[16]

Finally, if the *integer* in the result has the value zero, then the *sign*[17] and *exponent* are set to 0. Other than this case, trailing zeros are not removed after operations, except as described below (for example, after division).

add and subtract

add and **subtract** both take two operands. If either number is zero (that is, its *integer* is zero) then the other number, rounded to *precision* digits if necessary, is used as the result (with sign and exponent adjustment as appropriate).

Otherwise, the two numbers are aligned at their units digit, taking account of any exponent, and extended with zeros on the right and left as necessary to overlap all digits of both numbers.[18] The numbers are then added or subtracted as requested.

For example, the addition

```
add('xxxx.xxx', 'yy.yyyyy')
```

(where “x” and “y” are any decimal digits) becomes:

```
  xxxx.xxx00
+ 00yy.yyyyy
------------
  zzzz.zzzzz
```

The result is then 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.

Examples:

```
add('12', '7.00')          ==>  '19.00'
subtract('1.3', '1.07')  ==>  '0.23'
subtract('1.3', '2.07')  ==>  '-0.77'
```

[16] This 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.

[17] This rule, together with the **to–number** definition, ensures that numbers with value −0 will not result from this specification (though they are permitted in the extended specification). This allows a concrete representation for this specification to comprise simply two integers in twos complement form.

[18] If adding, and *rounding* is *round–half–up*, it is only necessary to extend the numbers up to a total maximum of *precision*+1 digits. The number with the smaller absolute value may then lose some or all of its digits on the right. In the example, `'yy.yyyyy'` would have three digits truncated if *precision* were 5.

plus and minus

plus and **minus** both take one operand, and correspond to the prefix plus and minus 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 `add('0', a)` and `subtract('0', b)` respectively.

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. The numbers are multiplied together ("long multiplication") resulting in a number which may be as long as the sum of the lengths of the two operands. For example:

```
multiply('xxx.xxx', 'yy.yyyyy')
```

becomes:

```
'zzzzz.zzzzzzzz'
```

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

Examples:

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

divide

divide takes two operands. For the division:

```
divide('yyy', 'xxxxx')
```

the following steps are taken: first, the number `'yyy'` is extended with zeros on the right until it is larger than the number `'xxxxx'` (with note being taken of the change in the power of ten that this implies). Thus in this example, `'yyy'` might become `'yyy00'`. Traditional long division then takes place, which can be written:

```
         zzzz
  xxxxx ) yyy00
```

The length of the result (`'zzzz'`) is such that the rightmost "z" will be at least as far right as the rightmost digit of the (extended) "y" number in the example. During the division, the “y” number will be extended further as necessary, and the “z” number (which will not include any leading zeros) is also extended as necessary until the division is complete.

The division is complete when *precision* digits have been accumulated; at this point the result is rounded according to the *rounding* algorithm and the remainder from the division.[19]

Finally, any insignificant trailing zeros are removed. That is, if the *exponent* is not zero and the *integer* is a multiple of a power of ten then the *integer* is divided by that power of ten and the *exponent* increased accordingly. If the *exponent* was negative it will not be increased above zero.

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'
divide('1000', '100') ==>  '10'
divide('1000', '1')   ==>  '1000'
```

power

power takes two operands, and raises a number (the left-hand operand) to a whole number power (the right-hand operand).

The right-hand operand must be a whole number whose integer part (after any exponent has been applied) has no more digits than *precision* and whose decimal 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 then the result is inverted (divided into 1).

For calculating the power, the number is in theory multiplied by itself for the number of times expressed by the power, and finally trailing zeros are removed (as though the result were divided by one).

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. When the initial calculation is complete, the temporary result is divided into 1 if the power was negative.

The multiplications and 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 a precision of `digits+elength+1` digits. Here, `elength` is the length in decimal digits of the integer

[19] For *round–half–up*, rounding can be effected by continuing the division to *precision*+1 result digits, at which point the final digit can be inspected to determine rounding and the remainder need not be used.

part of the whole number "n" (*i.e.*, excluding any sign, decimal part, decimal point, or insignificant leading zeros.[20]

Finally, any insignificant trailing zeros are removed, as for **divide**.

If, when raising to a negative power, an overflow or underflow occurs before the division into 1, the condition raised will be adjusted to reflect the pending division. That is, an overflow will cause an **Underflow** condition, and an underflow will cause an **Overflow** condition.

Examples:

```
power('2', '3')    ==>  '8'
power('2', '-3')   ==>  '0.125'
power('1.7', '8')  ==>  '69.7575744'
```

Note: 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.

divide-integer

divide–integer takes two operands; it divides two numbers and returns the integer part of the result. 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`.

[20] 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.

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 identical results 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'
```

remainder

remainder takes two operands; it returns the remainder from integer division, and is defined as being the residue of the dividend after the operation of calculating integer division as just 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, 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.

compare

compare takes two operands and compares their values numerically.

The comparison is effected by subtracting the two numbers (calculating the difference, as though by using the **subtract** operation with the same operands) and then returning an indication of the sign of the result (`'-1'` if the result is negative, `'0'` if the result is zero, or `'1'` if the result is positive).

It is therefore the *difference* between two numbers, when subtracted under the rules for the **subtract** operation, which determines their equality.

When the signs of the operands are different a value representing the sign of each operand (`'-1'` if negative, `'0'` if zero, or `'1'` if positive) is used in place of that operand for the comparison instead of the actual operand.[21]

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'
```

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

Exceptional conditions

This section lists, in the abstract, the exceptional conditions that can arise during the operations defined in this specification. These conditions are all abnormal; once one occurs the current operation is not completed and the implementation must not quietly continue with some substituted result.

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 might be displayed.

The following exceptional conditions can occur:

Conversion overflow

This occurs if an string is being converted to a number and the value of the *integer* or *exponent* resulting from the conversion is too large for an implementation to handle (perhaps because the concrete representation has size limits).

Conversion syntax

This occurs if an string is being converted to a number and it does not conform to the numeric string syntax (see page 8).

Conversion underflow

This occurs if an string is being converted to a number and the value of the *exponent* resulting from the conversion is too small for an implementation to handle (perhaps because the concrete representation has size limits).

Division by zero

This occurs if division by zero was attempted (during a **divide–integer**, **divide**, or **remainder** operation), and the dividend was not zero.

Division impossible

This occurs if the integer result of a **divide–integer** or **remainder** operation had too many digits (would be longer than *precision*).

Division undefined

This occurs if division by zero was attempted (during a **divide–integer**, **divide**, or **remainder** operation), and the dividend is also zero.

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 in the same way as other exceptional conditions during arithmetic.

Invalid operation

This occurs if the right-hand operand to a **power** operation has a non-zero decimal part or has more than *precision* digits.

Invalid context

This occurs 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.

Lost digits

This occurs if an operand to an arithmetic operation has more leading significant digits than the *precision* in the context, and *lost–digits* in the context is 1.

Overflow

This occurs if the *exponent* of a result (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.

Underflow

This occurs if the *exponent* of a result (from an operation that is not an attempt to divide by zero) would be smaller (more negative) than the smallest value that can be handled by the implementation.

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).

It is also recommended that exceptional conditions be recorded as a series of accumulated flags. This allows a series of operations to be carried out, with a check being made at the end of the sequence to detect whether any error occurred.

Appendix A – 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 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*.[22]

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 presentation.

The arithmetic described here 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).

In the past eighteen years the arithmetic has been used successfully for hundreds of thousands of applications covering the entire spectrum of computing; among other fields, that spectrum includes operating system and application scripting, text processing, commercial data processing, engineering, scientific analysis, and pure mathematics research. From this experience there is some confidence that the various defaults and other design choices are sound.

Fundamental concepts

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

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

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

[22] 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`

- an optional *sign* (only significant when negative)
- a numeric part, or *numeric*, 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,[23] 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.

Fortunately, the later standard ANSI/IEEE 854-1987,[24] 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, and an exponent (which is a 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[25] 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[26] in the Atari Operating System) offer just one or two alternatives for precision – in some cases, for apparently arbitrary reasons. Again, this does not match the user model of decimal arithmetic; one designed for people to use must provide a wide range of available precisions.

This specification provides for user selection of precision; the representation (especially if it is to conform to the IEEE 854-1987 standard referred to above) may have only a few options for precisions, but within the limits of the representation the precision used for operations may be chosen by the programmer.

---

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

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

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

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

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 whether *lost–digits* checking is to be applied. Other items could be included as future extensions, as in the Extended Specification.

Appendix B – Changes

This appendix documents changes since the first public draft of this specification (0.65, 26 Jul 2000). It is not part of the specification.

Changes in Draft 0.66 (28 Jul 2000)

- The rules constraining any limits applied to the *exponent* of a number (see page 3) 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 *integer* 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 values of –0.
- The Exceptional conditions (see page 19) 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 3) have been corrected (E_{min} did not take into account the length of the *integer*).
- The rules for converting a number to a scientific string (see page 9) have been rephrased and corrected (the previous rules incorrectly converted some zero values).
- The Exceptional conditions (see page 19) 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 as a recommendation.
- The rules constraining the right-hand operand of the **power** operation have been clarified, and the **Invalid operation** condition has been added to report a error in the operand.
- 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 trailing zeros have been clarified.
- Minor clarifications and additional examples have been added.

Index