1 Rounding

Converting a rational number $x \in \mathcal{Q}$ to a nearby integer $n \in \mathcal{Z}$ can be done in surprisingly many ways. We shall take the floor operation

$$\lfloor x \rfloor = \max\{n \in \mathcal{Z} | n \leq x\}$$

as the basic operation. We also need $\perp$ (undefined), even(n) testing whether n is a multiple of 2, $p\,?\,t : f$ (if p then t else f), and $\sigma(x) = x \geq 0\,?\,1 : -1$.

exact	$x = \lfloor x \rfloor\,?\,\lfloor x \rfloor : \perp$	exception if not integral
down	$\lfloor x \rfloor$	floor
up	$\lceil x \rceil = -\lfloor -x \rfloor$	ceiling
in	$\sigma(x) \times \lfloor \lvert x \rvert \rfloor$	truncate
out	$\sigma(x) \times \lfloor \lvert x \rvert + \frac{1}{2} \rfloor$	away from zero
evenr	$\text{even}(\lfloor x \rfloor)\,?\,\lfloor x \rfloor : \lceil x \rceil$	round to even
oddr	$\text{even}(\lfloor x \rfloor)\,?\,\lceil x \rceil : \lfloor x \rfloor$	round to odd (not used in practice)

None of these is rounding. For that we need

$$\text{nearest} f x = d > \frac{1}{2}\,?\,\lceil x \rceil : d < \frac{1}{2}\,?\,\lfloor x \rfloor : f(x)$$

where $d = x - \lfloor x \rfloor$

This higher-order function can be combined with the seven conversions above to make seven "round to nearest" modes such as nearest out (banker's rounding) or nearest evenr (IEEE rounding).

2 Existing standards

2.1 ISO 10967 Language-independent arithmetic

Roughly speaking, the three parts of LIA cover everything in Fortran 95 and C99 and a little bit more.

We have to discuss ISO 10967 for two reasons: it is the standard that is meant to serve as a reference for arithmetic in programming languages, and it does define fixed point numbers (but no operations on them). Any definition of fixed point arithmetic should be practically compatible with it.

Further, the 1998 edition provided axioms (though no evidence that the system described was consistent with them) which could serve as models for checking a fixed-point specification.

The Language-independent arithmetic standard has three parts.

2.1.1 Part: 1 Integer and floating point arithmetic

An integer data type I has parameters $bounded_I$ true if the set is finite, $minint_I$ being the smallest value or $-\infty$, and $maxint_I$ being the largest value or ∞. If $bounded_I$ is true, then $minint_I$ can be 0, -$maxint_I$, or -($maxint_I$+1). Curiously,

while both signed and unsigned bounded integers are allowed, unbounded integers must be signed; "unbounded natural numbers are not covered by this document." The usual 6 comparison operations are defined.

The arithmetic operations are neg_I (unary negation), add_I, sub_I, mul_I, flooring division $quot_I$ and the corresponding remainder mod_I, abs_I, and $signum_I$ taking values 1 and -1. The 1998 edition included $sign_I$ with values 1, 0, and -1, but that was dropped. The 1998 edition included truncating division div_I^t and the corresponding remainder rem_I^t and two additional remainder functions mod_I^a and mod_I^p, all of which were deleted in the 2012 edition.

LIA-1 integer arithmetic must handle overflow by causing an overflow indication. This includes conversion to a smaller integer type. Wrapping arithmetic is covered in part 2.

A floating point data type F has parameters r_F (radix), p_F (precision), $emax_F$ (largest exponent), $emin_F$ (smallest exponent), $denorm_F$ (whether IEEE-style subnormals are supported), and ice_60559_F (whether F conforms to the IEEE standard). The usual 6 comparison operations are defined, taking care to dodge around infinities and NaNs.

Additional tests $isnegzero_F$, $istiny_F$, $isnan_F$, and $issignan_F$ express tests that cannot be done using ordinary comparison against zero.

Three sets of basic operations (rounding to nearest even, down, or up) are defined if $denorm_F$ is true, just the rounded to nearest set if $denorm_F$ is false. The basic operations are *add*, *sub*, *mul*, *div*, and *sqrt*.

Some arithmetic operations are not sensitive to rounding mode: abs_F, two-valued $signum_F$, the IEEE remainder $residue_F$, $intpart_F$, $fractpart_F$, $succ_F$ (next higher), $pred_F$ (next lower), and ulp_F.

There are three conversions from floating point to integer: $ceiling_{F\to I}$, $floor_{F\to I}$, and $rounding_{F\to I}$.

Section 5.3.5 defines floating point to fixed point conversion (for the sake of format conversions like C's `%.nf`) and in passing defines unbounded fixed point types, with parameters r_D (radix), d_D ("density" = scale), and $dmax_D$ (a positive element of $\mathcal{R}$ or undefined). Two sets are defined:

$$\begin{aligned} D^* &= \{n/(r_D^{d_D}) | n \in \mathcal{Z}\} \\ D &= D^* \text{if unbounded} \\ D &= D^* \cap [-dmax_D, dmax_D] \text{if bounded} \end{aligned}$$

Three rounding functions $nearest_D$ (our nearest evenr), up_D (ceiling), and $down_D$ (floor), all $\mathcal{R} \to D^*$, are defined. Conversion from floating point to fixed point and from fixed point to floating point uses one of these rounding modes just like the basic arithmetic operations.

However, fixed point exists in LIA *only* for I/O conversion; there are no comparison or arithmetic operations on D.

Floating point arithmetic with truncation is conspiciously absent from the 2012 edition of LIA-1. What it describes is accurately rounded arithmetic and the building blocks for interval arithmetic.

2.1.2 Part: 2 Elementary numerical functions

This part was issued in 2001. There are some slight "continuity errors" due to the revision of LIA-1: iec_559_F is now iec_60559_F, for example. This is another example of why standards need to be machine-checked.

The additional integer operations include maximum, minimum, greatest common divisor, and least common multiple of a pair or sequence, if $x \geq y$ then $x - y$ else 0 (diminish), integer power (but broken for 0^0 so of very little use), floored shifting by a power of 2 or 10, floored square root, divisibility, evenness, oddness, floored ($quot_I$, mod_I), rounded ($ratio_I$, $residue_I$), and ceilinged ($group_I$, pad) quotient and remainder. (The pad operation is actually $\lceil x/y \rceil \times y - x$.)

There are also wrapping versions of *add*, *sub*, and *mul*, plus "overflow" versions giving you the high bits lost by wrapping so that multiprecision arithmetic can be written.

The additional floating point operations include maximum and minimum of a pair or sequence, diminish, conversion to integral in floating form ($floor_F$, $rounding_F$, and $ceiling_F$) together with $*\_rest_F$ operations providing the lost fraction parts exactly, square root (already in part 1) and reciprocal square root. There is an operation for multiplying two floats giving a result in a wider format with a single rounding, just right for dot product.

The elementary transcendental operations include raising a floating point number to an integer or floating point power, plus the usual exponential, logarithmic, trigonometric, and hyperbolic functions in several forms.

2.1.3 Part 3: Complex integer and floating point arithmetic and complex elementary numerical functions

This part was issued in 2006. It defines imaginary and complex integer and imaginary and complex floating point data types, basic operations on them, and elementary transcendental operations. Maximum and minimum are extended to imaginary but not to complex types.

Part 3 ends with recommended changes to some of the operations defined in Part 2 "so that the returned angle is for the proper quadrant". Again this suggests that a standard on which great care was taken might have benefited from testing.

3 ISO/IEC/IEEE 60559:2011

This is the standard for floating point arithmetic. It is the ISO/IEC edition of IEEE 754:2008, which in turn was an update of classic IEEE 754 arithmetic to include decimal floating point. It specifies 32-, 64-, and 128-bit binary formats that can hold about 7, 16, and 34 decimal digits respectively and 64- and 128-bit decimal formats that can hold 16 and 34 decimal digits respectively (table 3.2).

It is common belief that IEEE 754 arithmetic is all but universal. However, few systems support 128-bit binary floats (the popular ARM, x86, and x86-64

architectures do not; SPARC does), and even fewer support decimal floating-point natively (recent IBM z/Series and POWER machines do).

IEEE 754 originally defined four rounding modes, up, down, in, and nearest evenr in our terminology. The current standard adds nearest out (banker's rounding).

3.1 COBOL

COBOL is one of the oldest programming languages still in use. It was designed to support accounting applications on small machines. For example, the Siemens 3003, regarded as "a large-scale ... computer", had 8000 words of storage; the COBOL 61 compiler for that machine needed 10 passes. Brinch Hansen and House [ref] report that "The major problem of implementation turned out to be the numerous definition problems created by the vagueness of the official Cobol report."

One of the major features of COBOL from the beginning was its use of decimal fixed point arithmetic rather than floating point. Accounting applications required answers that were exact or computed according to known rounding rules. Binary integers and floats were eventually adopted in 2002, but had been common extensions for many years.

A number is characterised by (a) whether it is signed or not, (b) its precision, and (c) its scale. A declaration like
77 SOME-NUMBER PICTURE S9(b)V9(a)
declares a signed variable (S) with a precision of $b+a$ decimal digits and a scale of a. The letter V designates a virtual (not actually stored) decimal point. In terms of LIA-1, this corresponds to $r_D = 10$, $d_D = a$, $dmax_D = (10^{a+b} - 1)/10^b$.

In LIA-1, arithmetic is only defined between values of the same type. In COBOL, it is extremely common for the operands and result of an operation to be of different types. A typical instruction is

```
MULTIPLY X BY Y GIVING Z ROUNDED
    ON SIZE ERROR statements
    NOT SIZE ERROR statements
END MULTIPLY
```

where the product $X \times Y$ is computed according to exact rational arithmetic, rounded to the scale of Z, and if the result is too large to fit, the ON SIZE ERROR statements are executed, otherwise the result is stored in Z and the NOT SIZE ERROR statements are executed.

3.1.1 Four semantics for arithmetic

The previous paragraph is a lie. The current COBOL standard does not offer a semantics for arithmetic. It offers *four* semantics for arithmetic. A COBOL program may contain an OPTIONS paragraph near the beginning:

OPTIONS.

[ARITHMETIC IS {NATIVE | STANDARD |
STANDARD-BINARY | STANDARD-DECIMAL }]
[DEFAULT ROUNDED MODE IS one of eight]
...
[INTERMEDIATE ROUNDING IS one of four]

In our terminology, the rounding modes are out, nearest out, nearest even, nearest in, exact, up, down, and in.

There is a subtle difference between these statements:

```
MULTIPLY X BY Y GIVING Z.
COMPUTE Z = X * Y.
```

The latter involves an arithmetic expression which has an intermediate result that is then assigned to the destination. The former does not. It appears that the MULTIPLY statement involves a single rounding whereas the COMPUTE statement may involve a double rounding.

Native arithmetic is whatever the implementor says it is.

Standard arithmetic is what used to be standard arithmetic in COBOL-85. It does not cover binary or floating-point operations, which are treated as native. Standard arithmetic is declared to be obsolete. "A standard intermediate data item is ... the unique value zero or an abstract, signed, normalized decimal floating-point temporary data item ... with a precision of 32 decimal digits" and a scale of -999 to 999 (roughly speaking). Intermediate values are rounded in certain circumstances.

In expressions, addition, subtraction, multiplication, and division are are "the exact [result] truncated to 32 significant digits, normalized, and stored in a standard intermediate data item".

Standard-binary arithmetic requires arithmetic operations to be done according to IEEE binary floating-point arithmetic (128-bit format).

Standard-decimal arithmetic requires arithmetic operations to be done according to IEEE decimal floating-point arithmetic (128-bit format).

The result is that the meaning of an arithmetic statement or expressions can be changed by a line thousands of lines away.

The standard tries very hard to avoid using mathematical notation. Let us take Format 5 of the DIVIDE statement as an example:

DIVIDE {identifier-2|literal-2} BY {identifier-1|literal-1}
GIVINE identifier-3 [rounded-prhase]
REMAINDER identifier-4
[ON SIZE ERROR imperative-statement-1]
[NOT SIZE ERROR imperative-statement-2]
[END-DIVIDE]

1) When native arithmetic is in effect, the quotient is the result of dividing the dividend by the divisor. When standard arithmetic,

> standard-decimal arithmetic, or standard-binary arithmetic is in effect, the quotient is the result of the arithmetic expression
>
> (dividend/divisor)
>
> where the values of dividend and divisor are ...
>
> 6b) The quotient is then formed as specified in general rule 1 and stored in identifier-3 ...
>
> 6c) If the size error condition is not raised, a subsidiary quotient is developed that is signed and derived from the quotient by truncation of digits at the least significant end and that has the same number of digits and the same decimal point location as the data item references by identifier-3. The remainder is calculated as indicated in general rules 7 and is stored in the data item referenced by identifier-4 unless storing the value would cause a size error condition ...
>
> 7) When native arithmetic is in effect, the remainder is the result of multiplying the subsidiary quotient and the divisor and subtracting the product from the dividend. When standard standard arithmetic, standard-decimal arithmetic, or standard-binary arithmetic is in effect, the remainder is the result of the arithmetic expression
>
> (divided − (subsidiary-quotient ∗ divisor))
>
> where ...

The reason the standard *does* use some expressions here is so that the rules for intermediate rounding apply to them. With 4 arithmetics, 8 intermediate roundings, and 4 default roundings, a simple DIVIDE statement could mean 128 different things.

3.2 PL/I

PL/I was introduced in the mid-1960s as a blend of Algol, Fortran, and COBOL. The ECMA-50 (1976), ANSI X3.53-1976, BS 7148:1990, NF Z 65-500, and ISO 6160:1979 standards for PL/I are all basically the same. They have expired. BS ISO/IEC 6522:1992 (Information technology, Programming languages, PL/I general purpose subset) is still current.

PL/I fixed point types are characterised by a *base* (2 or 10), a *precision* (total number of digits), and a *scale* (number of digits after the radix point). There are actually two sets of rules for determining the precision and scale of a result (ANSI and IBM). For FIXED DECIMAL they coincide:

- $(p_1, s_1) \pm (p_2, s_2) \Rightarrow (p, s)$ where
  $s = s_1 \vee s_2$
  $p = 1 + s + (p_1 - s_1 \vee p_2 - s_2)$

- $(p_1, s_1) \times (p_2, s_2) \Rightarrow (p, s)$ where
  $s = s_1 + s_2$
  $p = 1 + p_1 + p_2$
- $(p_1, s_1)/(p_2, s_2) \Rightarrow (p, s)$ where
  $s = N - p_1 + s_1 - s + 2$
  $p = N$ and N is the implementation-defined maximum precision.
- $\mathrm{MOD}((p_1, s_1), (p_2, s_2)) \Rightarrow (p, s)$ where
  $s - s_1 \vee s_2$
  $p = N \wedge (p_2 - s_2 + s)$.
- $(p_1, s_1) * *k \Rightarrow (p, s)$ where
  $s = s_1 \times k$
  $p = (p_1 + 1) \times k - 1$, if k is a literal integer.
- $\mathrm{ROUND}((p_1, s_1), s) \Rightarrow (p, s)$ where
  s is given as the second argument
  $p = 1 \vee (p_1 - s_1 + 1 - s \wedge N)$

The rule for division can lead to some nasty surprises. For example, 25+01/3 yields 25.3333333333333 but 25+1/3 results in an overflow. So there are special functions ADD(x, y, p[, s]), SUBTRACT(x, y, p[, s]), MULTIPLY(x, y, p[, s]), DIVIDE(x, y, p[, s]), computing a result to a specified precision and scale (default s = 0) in a single rounding. There is no version of MOD with precision and scale parameters. In assignment statements, "padding or truncation can occur on the left or the right. If nonzero ... digits on the left are lost, the SIZE condition is raised." This means that a statement like
X = X * Y;
uses the "in" rounding mode. Unlike COBOL, PL/I does not offer control over rounding, not even using the special functions, so there is no way to change this.

3.3 SQL

The ISO/IEC 9075-1:2003 standard, SQL 2003, part 1, says in section 4.4.3.1 that "There are two classes of numeric type: *exact numeric*, which includes integer types and types with specified precision and scale; and *approximate numeric*, which is essentially floating point, and for which a precision may optionally be specified." The result of an arithmetic operation "is of a numeric type that depends only on the numeric type of the operands. If the result cannot be represented exactly in the result type, then whether it is rounded or truncated is implementation-defined. An exception condition is raised if the result is outside the range of numeric values of the result type, or if the arithmetic operation is not defined for the operands."

Numbers and arithmetic are defined in section 4.4 of part 2.

> 'An exact numeric type has a precision P and a scale S. P is a positive integer that determines the number of significant digits in

> a particular radix R, where R is either 2 or 10. S is a non-negative integer. Every value of an exact numeric type of scale S is of the form $n \times 10^{-S}$,[1] where n is an integer such that $-R^P \leq n < R^P$. [Note 13 — Not every value in that range is necessarily a value of the type in question.]"

The intent of note 13 may be to allow both sign-and-magnitude representation for NUMERIC types and R's-complement representation for INTEGER types; it's not obvious why any other value would be excluded.

Conversion may be done by rounding or truncation; it is not the programmer who chooses.

> "If an assignment of some number would result in a loss of its most significant digit, an exception condition is raised. If least significant digits are lost, implementation-defined rounding or truncating occurs, with no exception condition being raised."
>
> "An approximation obtained by truncation of a numeric value N for an ⟨exact numeric type⟩ T is a value V in T such that N is not closer to zero than is V and there is no value in T between V and N.
>
> An approximation obtained by rounding of a numeric value N for an ⟨exact numeric type⟩ T is a value V in T such that the absolute value of the difference between N and the numeric value of V is not greater than half the absolute value of the difference between two successive numeric values in T. If there is more than one such value V, then it is implementation-defined which one is taken."

Aside from the non-portable nature of rounding, arithmetic is supposed to be unsurprising:

> "Operations on numbers are performed according to the normal rules of arithmetic, within implementation-defined limits . . . "
>
> "Numbers are compared with respect to their algebraic value." (8.2)

Paraphrasing sections 4.4.3 and 6.26,

- The POWER function never returns an exact answer (unlike PL/I).
- The floor and ceiling functions return exact integers $S = 0$.
- Unary plus, unary minus, and the absolute value function return a result of the same type as their argument.
- The MOD function only accepts integer arguments.
- WIDTH_BUCKET(x, l, u, n) appears to mean

[1] It is possible that 10 should have been R, but only INTEGER types with $S = 0$ may have $R = 2$.

if $l < u$ then
    if $x < l$ then 0
    else if $x \geq u$ then $n + 1$
    else $\lfloor n \times ((x - l)/(u - l)) \rfloor$
else if $l > u$ then
    if $x > l$ then 0
    else if $x < u$ then $n + 1$
    else $\lfloor n \times ((x - u)/(l - u)) \rfloor$

and the result has the same type as n, which must be integral. A system that can implement this function ought to be able to implement MOD on non-integral exact numbers.

- If x has radix R, precision P_x, and scale S_x, and y has radix R, precision P_y, and scale S_y, then $x + y$ and $x - y$ have scale $\max(S_x, S_y)$ and implementation-defined precision;
- $x * y$ has scale $S_x + S_y$ and implementation-defined precision;
- and x/y has implementation-defined precision and implementation-defined scale.
- "If the operator is not division and the mathematical result of the operation is not exactly representable with the precision and scale of the result data type, then an exception condition is raised."
- "If the operator is division and the approximate mathematical result of the operation represented with the precision and scale of the result data type loses one or more leading significant digits after rounding or truncating if necessary, then an exception condition is raised ... The choice of whether to round or truncate is implementation-defined."

COBOL and PL/I have fixed-point decimal types in order to support accounting. SQL was given fixed-point NUMERIC types in order to inter-operate with COBOL and PL/I, and is very much used for accounting data. The problem is that SQL arithmetic is *not* guaranteed to be consistent with COBOL or PL/I or accounting rules:

- there is no programmer control over rounding;
- there are no precisions that have to be supported, not even older COBOL's 18 digits;
- there is no fixed-point remainder; and
- if you want to calculate the price of one apple at $2.72 per dozen, 2.72/12 could give you the answer 0 if an implementation defined the scale for division to always be $S = 0$.

3.4 Smalltalk

Smalltalk-80 is the programming language that brought Object-Oriented programming and Integrated Development ENvironments to the world. It got an ANSI standard in 1989. The standard includes unbounded integers, three sizes of floating point, rational numbers, and a ScaledDecimal class.

For the semantics of operations on ScaledDecimal numbers, the Smalltalk standard defers to ISO 10967 (LIA). Unfortunately, LIA has, by intent and quite explicitly, nothing to say about fixed point arithmetic.

The result is that implementations use one of four interpretations of ScaledDecimal:

- An arbitrary precision integer scaled by a power of 10. This agrees well with the standard, which says that "Scaled decimal objects provide a precise representation of decimal fractions with an explicitly specified number of fractional digits.".
- An integer scaled by a power of 10, represented as a 16-byte packed decimal value. This also agrees well with the standard.
- An arbitrary precision rational number that is not in general a whole multiple of a power of 10, rounded to a specified number of digits when and only when printing.
- The same, but with truncation instead of rounding.

For example,

(2/3) asScaledDecimal: 2 $\Longrightarrow$ 0.66s2
"and" 0.66s2 * 3 $\Longrightarrow$ 1.98s2
"but" ((2/3) asScaledDecimal: 2) * 3 $\Longrightarrow$ 2.00s2

in several Smalltalks (several others yield 0.67s2), while

(2/3) asScaledDecimal: 2 $\Longrightarrow$ 0.67s2
0.67s2 * 3 $\Longrightarrow$ 2.01s2
((2/3) asScaledDecimal: 2) * 3 $\Longrightarrow$ 2.01s2

in Smalltalks using a scaled-integer interpretation.

3.5 C♯

Summary: decimal arithmetic is standard in C♯. Such numbers are value types, as in COBOL and PL/I. C♯ does not support sufficient precision or sufficient scale to allow unproblematic conversion of COBOL 2002 code or data, though it is adequate for COBOL 85. There is no programmer control over rounding. Scale is part of the value, not part of the type, and the rule for division makes it hard to track scale at compile time. Above all, decimal arithmetic in C♯ is exact *except when it isn't*. The compiler will not tell you when unexpected rounding *might* happen, nor does the runtime system tell you when it *has* happened.

ISO/IEC 23270:2006 defines C♯ as it was in 2005. ISO/IEC 23231:2012 defines the Common Language Infrastructure The CLI standard omits the "Extended numerics library" (section IV.5.7), including System.Decimal, but the language standard has much to say.

Section 8.2.1 says

> The `decimal` type is appropriate for calculations in which rounding errors caused by floating-point representations are unacceptable.

Except for rounding errors introduced silently by C♯ itself.

Section 11.1.7 defines the `decimal` type.

> The `decimal` type is a 128-bit data type suitable for financial and monetary calcuations. The `decimal` type can represent values including those in the range 1×10^{-28} through 1×10^{28} with at least 28 significant digits. The finite set of values of of type `decimal` are of the form $(-1)^s \times c \times 10^{-e}$, where the sign s is 0 or 1, the coefficient c is given by $0 \leq c < Cmax$ and the scale e is such that $Emin \leq e \leq Emax$, where $Cmax$ is at least 1×10^{28}, $Emin \leq 0$, and $Emax \geq 28$. The decimal type does not necessarily support signed zeros, infinities, or NaN's.
>
> A `decimal` is represented as an integer scaled by a power of ten. For *decimal*s with an absolute value less than `1.0m`, the value is exact to at least the 28^{th} decimal place. For `decimal`s with an absolute value greater than or equal to `1.0m`, the value is exact to at least 28 digits. Contrary to the `float` and `double` data types, decimal fractional values such as `0.1` can be represented exactly in the `decimal` representation. In the `float` and `double` representations, such numbers often have non-terminating binary expansions, making those representations more prone to round-off errors.
>
> The result of an operation on values of type *decimal* is that which would result from calculating an exact result (preserving scale, as defined for each operator) and then rounding to fit the representation. Results are rounded to the nearest representable value, and, when a result is equally close to two representable values, to the value that has an even number in the least significant digit position (this is known as "banker's rounding"). That is, results are exact to at least the 28^{th} decimal place. Note that rounding may produce a zero value from a non-zero value.
>
> If a `decimal` arithmetic operation produces a result whose magnitude is too large for the `decimal format`, a `System.OverflowException` is thrown.
>
> The `decimal` type has greater precision but may have a smaller range than the floating-point types. Thus, conversions from the floating-point types to *decimal* might produce overflow exceptions, and conversions from `decimal` to the floating-point types might cause loss

> of precision or overflow exceptions. For these reasons, no implicit conversions exist between the floating-point types and `decimal`, and without explicit casts, a compile-time error occurs when floating-point and `decimal` operands are directly mixed in the same expression.

Section 14.7, "arithmetic operators", says

> The scale of the result [of a decimal multiplication], before any rounding, is the sum of the scales of the two operands.
>
> The scale of the result [of a decimal division], before any rounding, is the closest scale to the preferred scale which will preserve a result equal to the exact result. The preferred scale is the scale of x less the scale of y.
>
> The scale of the result [of a decimal remainder], before any rounding, is the larger of the scales of the two operands, and the sign of the result, if non-zero, is the same as that of [the first operand].
>
> The scale of the result [of a decimal addition or subtraction], before any rounding, is the larger of the scales of the two operands.

With the exception of division, the scale rules are familiar and sensible. There is no division method that takes a desired scale argument, unlike PL/I.

The scale rule for division appears to be intended to ensure that $\text{scale}((x * y)/y) = \text{scale}(x)$, but if we take `1.0m/3.0m` there is **no** scale which "will preserve a result equal to the exact result", and this is the case for almost every possible division. What you get is the highest supported scale that doesn't result in an overflow.

There is no direct way to ask a `decimal` for its scale, but two numerically equivalent values can be operationally distinguished:

1.0m.ToString() $\Longrightarrow$ "1.0"
1.00m.ToString() $\Longrightarrow$ "1.00"
4.0m/2.0m $\Longrightarrow$ 2
4.00m/2.0m $\Longrightarrow$ 2.0

The remainder operator is the counterpart of the operation that divides two decimal numbers giving an exact integer result. There is no such operation.

The `System.Decimal` class has methods `Ceiling`, `Round`, `Floor`, and `Truncate` which return exact integer answers in `decimal` form, no integer type is certain to be large enough. The Round method has optional desired-scale and how-to-round-0.5 arguments, supporting nearest out and nearest evenr.

3.6 Java

Summary: decimal numbers in Java are a reference type, `java.math.BigDecimal`, not a primitive type. There are no literals for this type nor are the usual arithmetic operators available. It is not described in the Java Language Specification.

However, full control over rounding is possible, and Java gives exact results unless you ask for rounding. It *is* possible to convert COBOL 2002 and data to Java. Any finite `float` or `double` value can be represented exactly as a `BigDecimal`.

There is no international standard for Java, but there is the much-revised Java Language Specification [https://docs.oracle.com/javase/specs/jls/se8/html/index.html] and the Java Community Process for maintaining Java, which is not entirely unlike a standards process.

The version of Java current at the time of writing was 1.8, and that edition of the Java Language Specification has nothing to say about `BigDecimal`.

The Java API Online Reference [https://docs.oracle.com/javase/8/docs/api/overview-summary.html] says that

> [`BigDecimal`s are i]mmutable, arbitrary-precision signed decimal numbers. A `BigDecimal` consists of an arbitrary precision integer *unscaled value* and a 32-bit integer *scale* [which may be positive, negative, or zero]. The value represented by [a] `bigDecimal` is therefore $(\text{unscaledValue} \times 10^{-\text{scale}})$.
>
> The `BigDecimal` class gives its user complete control over rounding behavior. If no rounding mode is specified and the exact result cannot be represented, an exception is thrown; otherwise, calculations can be carried out to a chosen precision and rounding mode by supplying an appropriate `MathContext` object to the operation. In either case, eight rounding modes are provided for the control of rounding.
>
> When a `MathContext` object is supplied with a precision setting 0f 0 ..., arithmetic operations are exact. ...In the case of divide, the exact quotient could have an infinitely long decimal expansion; for example, 1 divided by 3. If the quotient has a nonterminating decimal expansion and the operation is specified to return an exact result, an ArithmeticException is thrown. Otherwise, the exact result of the division is returned, as done for other operations.
>
> Since the same numerical value can have different representations (with different scales), the rules of arithmetic and rounding must specify both the numerical result and the scale used in the result's representation.
>
> For all arithmetic operators, the operation is carried out as though an exact intermediate result were first calculated and then rounded to the number of digits specified by the precision setting (if necessary), using the selected rounding mode.
>
> Besides a logical exact result, each arithmetic operation has a preferred scale for representing a result.

Operation — Preferred Scale of Result
Add — max(addend.scale(), augend.scale())
Subtract — max(minuend.scale(), subtrahend.scale())
Multiply — multiplier.scale() + multiplicand.scale()
Divide — dividend.scale() - divisor.scale()

> . . . an exact divide may have to use a larger scale [when] the exact result has more digits.
>
> `BigDecimal`'s *natural ordering* is *inconsistent with equals.*

The issue here is that `compareTo` compares the algebraic values of two `BigDecimal`s, but `equals` compares their representations.

Quotient returning an integer in `BigDecimal` form and corresponding remainder are supported.

3.7 Swift

Summary: Swift supports decimal numbers with up to 38 digits and a scale of -128 to 127.

Swift is a fast-changing language pushed by Apple. At the time of writing the current version was Swift 3.

Swift 2 supported "toll-free bridging" to the Objective-C type `NSDecimalNumber`, "[a]n instance [of which] can represent any number that can be expressed as mantissa $\times$ $10^{exponent}$ where mantissa is a decimal integer up to 38 digits long, and exponent is an integer from -128 through 127."

Swift 3 has a native `Decimal` "value type which offers the same functionality as the `NSDecimalNumber` reference type, and the two can be used interchangeably in Swift code that interacts with Objective-C APIs." Literals in integer or floating-point form can be used where Decimal literals are wanted, but they are first stored as Int or Float and then converted to Decimal.

Swift *Decimal*s include a NaN value but not infinities or negative zero.

The `NSDecimal` methods `adding`, `subtracting`, `multiplying`, `dividing`, and `raising` have an optional `NSDecimalNumberBehavior` argument, like the corresponding Java methods, which specifies a desired scale and rounding mode, which can be up, down, in, or nearest out. There is also `NSDecimalNumberHandler` which can specify what to do if an answer is inexact or overflows. In Swift, there are methods `add, +`, `subtract, -`, `multiply, *`, and `divide, /` without control arguments and NSDecimalAdd, NSDecimalSubtract, NSDecimalMultiply, and NSDecimalDivide methods with RoundingMode parameter.

[The current state of the documentation for Swift 3 is **horrible**. I'm having serious trouble finding *anything* about how this all works.]