The Maths Library

(v.1 23-03-12)

© W O Riha

Note that the section headings are hyperlinks.

1. Introduction

The Shen maths library contains all the functions found in the C - maths library (C-89 standard), see header file math.h, for example, and several other operations and functions which may be available in one form or another as built-in operations in other languages, but are not native to Shen (for example, modulus and integer division). The entire library has been coded in Shen, and hence, is portable.

Certain functions and operations which, mathematically speaking, only involve 'whole numbers' or 'integers' as arguments and function values were originally contained in a separate 'integer library', which also provided a subtype `integer`. As explained elsewhere, there now is no type `integer`, and so these functions have been included in the maths library. They are described in Section 4.

A user of the maths library (or any library, for that matter) will be familiar with at least some of the functions that are provided. For this reason, I have not explained in any detail exactly what certain functions compute, or how they are defined. For example, I assume that a potential user of the library will observe that the trigonometric functions `sin`, `cos` and `tan` are available (but not `cot`), and knows what they are used for, and so I do not define these functions. In the same vein, I do not explain the purpose of `atan2` (but provide a reference), because if you don't know what it does, you can't use it.

On the other hand, I do give explanations and/or definitions, when I am aware, or suspect, that there is an ambiguity, misconception or confusion as regards the meaning and purpose of a function. (See Section 3.3 on 'Rounding', and the 'Tutorial on Integer Division' in Section 4).

The examples given in the text are based on CL-Shen. Numeric results agree with those produced by JS-Shen, but not necessarily, where large numbers (integers) are involved. At the time of writing, no results for Clojure-Shen are available [1]. Unless otherwise stated, values are quoted as displayed to 16 or 17 places. This will exhibit rounding errors. Such inaccuracies are inherent in the platform.

Example:

```
(- 1.37 1.0)
0.3700000000000001 : number
```

2. Data Type `number`

`number` is one of the basic data types in Shen (see Shen Document **Numbers**). For the sake of simplicity, this type is equipped with only a few functions (see Shen Document **The Primitive Functions of Kλ**).

The arithmetic operations all have the signature

```
        number --> number --> number
```

The following functions are available:

a. `+` addition
b. `-` subtraction
c. `*` multiplication
d. `/` division (if `Y` is `0` , `(/ X Y)` raises the error "`division by zero`")

Notice that + and * are polyadic functions.

Examples:

```
(+ 7 8 9 10)
34 : number

(* 5.9 -8.2 9.4)
-454.772 : number
```

The comparison operations all have the signature

```
        number --> number --> boolean
```

The following functions are provided:

[1] Just before I finalised this documentation, a new Closure-Shen port was put up. It does load the maths library and seems to be working fine.

a. `=` equal (equality is defined for all native types).
b. `!=` not equal (`!=` is defined for all types[2] as shorthand for `(not (= X Y))`)
c. `<` less than
d. `>` greater than
e. `<=` less than or equal
f. `>=` greater than or equal

Note: The comparison operators are affected by (floating-point) rounding errors, in particular `=` and`!=`, however, this is perfectly normal.

Examples:
```
(= 12 (* 0.006 2000))
true : boolean

(= 12 (* 0.0006 20000))
false : boolean
```

The system predicate
```
	integer?:  A --> boolean
```
tests if an object is an 'integer'. ***Shen-integers*** are different from integers in other programming languages.

Examples:
```
(integer? -12.0)
true : boolean

(integer? 1.2)
false : boolean

(integer? (+ 1.2 2.8))
true : boolean

(integer? 15.67e3)
true : boolean

(integer? (* 0.42 100))
false : boolean

(integer? (* 0.41 100))
true : boolean
```

3. Maths Library Functions

3.1 Predicates

```
	natural? :  number --> boolean
```
Input: A number N.

Output: `true` if N is a non-negative integer, otherwise `false`.

Examples:
```
(natural? 12.5)
false : boolean

(natural? 100)
true : boolean
```
but also
```
(natural? 1.54e5)
true : boolean
```

[2] only if the maths library has been loaded!

```
        positive? :  number --> boolean
```
Input: A number N.

Output: `true` if N > 0, otherwise `false`.

Examples:
```
(positive? 12.5)
true : boolean
```

```
        negative? :  number --> boolean
```
Input: A number N.

Output: `true` if N < 0, otherwise `false`.

Examples:
```
(negative? 12.5)
false : boolean
```

```
        even? :  number --> boolean
```
Input: A number N.

Output: `true` if N is an even integer, otherwise `false`.

Examples:
```
(even? 2.5)
false : boolean

(even? 12)
true : boolean

(even? 13)
false : boolean
```

```
        odd? :  number --> boolean
```
Input: A number N.

Output: `true` if N is an odd integer, otherwise `false`.

Examples:
```
(odd? 12)
false : boolean

(odd? 13)
true : boolean
```
Note: `not even` is not equivalent to `odd:`
```
(odd? 13.3)
false : boolean

(not (even? 13.3))
true : boolean
```

3.2 Sign and Absolute Value

```
        sign :  number --> number
```
Input: A number N.

Output: The sign of N which is defined as follows

$$sign\ (N) = \begin{cases} 1 & \text{if } N > 0 \\ 0 & \text{if } N = 0 \\ -1 & \text{if } N < 0 \end{cases}$$

Example:
```
(sign -123.5)
-1 : number
```

```
        abs :  number --> number
```

Input: A number N.

Output: The absolute value, $|N|$, i.e. the value N × *sign* (N).

Example:

```
(abs -123.5)
123.5 : number
```

3.3 Rounding Functions

Rounding means replacing a numerical value by another value that is approximately equal, but is shorter or simpler. For example, rounding π = 3.1415926535… to three decimal places gives the value “3.142”; rounding 29.15 to the nearest integer gives the value “29”.

The most fundamental case is that of rounding a (real) number to an integer. There are five methods that are in common use (see for example, Rounding).

3.3.1 Rounding Up or Down

Definition: The largest integer not greater than *a*, called ***floor of a***, is denoted by $\lfloor a \rfloor$ or *floor (a).* Similarly, the smallest integer not less than *a*, called ***ceiling of a***, is denoted by $\lceil a \rceil$ or *ceiling (a).*

Here, *floor* (*a*) rounds down to the nearest integer (or towards minus infinity), whilst *ceiling* (*a*) rounds up (or towards plus infinity).

```
        floor :  number --> number
```

Input: A number N.

Output: The integer value $\lfloor N \rfloor$.

Examples: (see Table 1)

```
        ceiling :  number --> number
```

Input: A number N.

Output: The integer value $\lceil N \rceil$

Examples: (see Table 1)

Table 1

N	(floor N)	(ceiling N)
11.7	11	12
12	12	12
-11.7	-12	-11
-12	-12	-12

3.3.2 Rounding Towards Zero/Away from Zero

A real number such as 123.89 is composed of an ‘integer part’, 123, and a ‘fractional part’ 0.89. If the number is negative, e.g. -123.89 then, obviously, the integer part is -123. But what is the fractional part? It could be 0.89 or -0.89. In order to preserve the identity

$$\textit{integer part (a)} + \textit{fractional part (a)} = a$$

one usually opts for -0.89. This motivates the following

Definition: The number *sign* (*a*) × *floor* (*abs* (*a*)) is called the ***integer part of a***, denoted by *int-part* (*a*). The number *a* – *int-part* (*a*), is called the ***fractional part of a***, and is denoted by *frac-part* (*a*).

It is not difficult to see that

$$\textit{int-part}\ (a) = \begin{cases} \textit{floor}\ (a) & a \geq 0 \\ \textit{ceiling}\ (a) & a \leq 0 \end{cases}$$

which means that *int-part* (*a*) rounds towards zero. [*int-part* is often called *truncate* (or *trunc*), because the value is obtained by ignoring (throwing away) the fractional part].

```
        trunc :  number --> number
        int-part :  number --> number
```
Input: A number N.

Output: The integer part of N.

Examples: (see Table 2)

```
        frac-part :  number --> number
```
Input: A number N.

Output: The fractional part of N.

Examples:

Table 2

(fractional values have been rounded)

N	(int-part N)	(frac-part N)
11.7	11	0.7
12	12	0.0
-11.7	-11	-0.7

There is also a function (with a rather unfortunate name) which combines the preceding two functions

```
        modf :  number --> (number * number)
```
Input: A number N.

Output: The pair consisting of the integer- and fractional parts of N.

Example:
```
(modf -11.7)
(@p -11 -0.6999999999999993) : (number * number)
```

The counterpart to truncation ('rounding towards zero') is 'rounding away from zero'. This operation has no specific name and has not been implemented in the library. It is given by

$$a \rightarrow \begin{cases} \mathit{ceiling}\,(a) & a \geq 0 \\ \mathit{floor}\,(a) & a \leq 0 \end{cases}$$

For examples, see Table 3 below.

3.3.3 Rounding to the Nearest Integer

This way of rounding is used most often in practice. The idea is quite clear: either take the floor or the ceiling of the number, whichever is the closer. There is a problem, however, what to do if the number is precisely in the middle, i.e. when its fractional part is exactly equal to 0.5. There are various ways to resolve this dilemma by means of a 'tie-break'. The main concern is to avoid or to minimise a bias (for a discussion, see Rounding).

Some libraries may provide a choice of different tie-breaks. The method that is used in the Shen maths library is the default rounding mode of the IEEE 754 standard, which is also widely employed in bookkeeping. The rule, known as 'round half to even', is quite simple

if *frac-part* (a) = 0.5, then round to the *even* integer nearest to a.

For example, 13.5 is rounded to 14, 12.5 to 12, -12.5 to -12, and -13.5 to -14.

```
        maths-round0 : number --> number
```
(see below for a more convenient alternative)

Input: A number N.

Output: N rounded to the nearest integer.

Example: (also see Table 3)
```
(maths-round0 123.78)
124 : number

(maths-round -1234.5)
-1234 : number
```

Table 3 (from Rounding)

Comparison of the five integer rounding methods

N	round down (towards −∞)	round up (towards +∞)	round towards zero	round away from zero	round to nearest
	floor	***ceiling***	***trunc***		
+23.67	+23	+24	+23	+24	+24
+23.50	+23	+24	+23	+24	+24
+23.35	+23	+24	+23	+24	+23
+23.00	+23	+23	+23	+23	+23
0	0	0	0	0	0
−23.00	−23	−23	−23	−23	−23
−23.35	−24	−23	−23	−24	−23
−23.50	−24	−23	−23	−24	−24
−23.67	−24	−23	−23	−24	−24

3.3.4 Rounding to a Specified Number of Decimal Places

`maths-round' : number --> number --> number` (see below for a more convenient alternative)

Input: A number X and an integer N.

Output: X rounded to N decimal places, or an error message, if N is not an integer.

Example:

```
(maths-round' 123.7823457 5)
123.78235 : number
```

Note: The function also works for negative values of N.

```
(maths-round' 87451.167 -2)
87500.0
```

Functions `round0` and `round'` are available as a single function `round` with either one or two arguments

`round : number --> number` (corresponds to `maths-round0`)

`round : number --> number --> number` (corresponds to `maths-round'`)

Examples:

```
(round -12.789)
-13 : number
```

```
(round -12.789 2)
-12.79 : number
```

Notice that `(round X 0)` is equivalent to `(round X)`.

3.4 Mathematical Constants

e	`e`	= 2.71828182845904523536
log 10	`log10`	= 2.30258509299404568402
log 2	`log2`	= 0.69314718055994530942
π	`pi`	= 3.14159265358979323846
$\pi/2$	`pi/2`	= 1.57079632679489661923
$\pi/4$	`pi/4`	= 0.78539816339744830962
2π	`pi*2`	= 6.28318530717958647692
$1/\pi$	`one/pi`	= 0.31830988618379067154
$2/\pi$	`two/pi`	= 0.63661977236758134308
$\log_2 e$	`log2e`	= 1.44269504088896340736
$\log_{10} e$	`log10e`	= 0.43429448190325182765
$\sqrt{2}$	`sqrt2`	= 1.41421356237309504880
$1/\sqrt{2}$	`one/sqrt2`	= 0.70710678118654752440
$2/\sqrt{\pi}$	`two/sqrt-pi`	= 1.12837916709551257390

3.5 Degree Conversions

Angles can be measured or specified either in degrees or in radians. Measured in degrees, a full circle is 360°, a quarter is 90° (right angle), whereas in radians a circle corresponds to 2π, and a right angle to $\pi/2$. Furthermore, 1° is equal to 60' (60 minutes), and 1 minute is 60" (60 seconds). Specifying an angle in degrees, minutes and seconds, or in fractional degrees (minutes or seconds) is very common in astronomy and geodesy. In mathematics and in numerical computations, radians is assumed by default.

The syntax of a degree specification – what is allowed and what is not.

38° 15' 23.78"	ok
38° 15.45' 23"	not ok
38° 15.45'	ok
38° -15.45'	not ok
38.5° 15' 23"	not ok
38° 75' 23.78"	not ok
38° 23.78"	not ok
38° 0' 23.78"	ok
38.46° ok	

An angle is represented as a list of up three numbers based on the grammar:

<degrees-list> ≔ **[**<number>**]**

<minutes-list> ≔ **[**<integer> <non-negative-number-less-than-60>**]**

<seconds-list> ≔ **[**<integer> <natural-number-less-than-60> <non-negative-number-less-than-60>**]**

<angle> ≔ <degrees-list> | <minutes-list> | <seconds-list>

The following functions are available:

```
        dms->degs : (list number) --> number
```

Input: A list of three numbers.

Output: The corresponding angle in fractional degrees (or an error message).

Examples:

```
(dms->degs [38 0 23.8])
38.00661111111111 : number

(dms->degs [38 10.4 23.8])
dms->degs - type error in Mins
```

```
(dms->degs [-38 10 30])
-38.175 : number

(dms->degs [10 20 30 40.8])
too many arguments in dms->degs
```

The inverse of `dms->degs` is

`degs->dms : number --> (list number)`

Input: A number representing an angle in fractional degrees.

Output: The angle as a list of degrees, minutes and seconds.

Examples:

```
(degs->dms -38.175)
[-38 10 30] : (list number)

(degs->dms (dms->degs [30 39 59.9]))
[30 39 59.9] : (list number)
```

`degs->rad : number --> number`

Input: A number representing an angle in fractional degrees

Output: The angle expressed in radians.

Examples:

```
(degs->rad 30.0)
0.5235987755982988 : number  \* this is π/6 *\

(degs->rad -123.456)
-2.1547136813421197 : number
```

And its inverse

`rad->degs : number --> number`

Input: A number representing an angle in radians

Output: The angle expressed in fractional degrees.

Examples:

```
(rad->degs (value pi/4))
45.00000000000001 : number

(rad->degs (degs->rad -123.456))
-123.45600000000002 : number
```

3.6 Trigonometric Functions

All trigonometric functions assume that the argument is specified in radians (see degree conversion).

`sin : number --> number`

Input: A number X.

Output: sine of X – a value between -1 and +1, inclusive.

Examples:

```
(sin (degs->rad 45))
0.7071067811865475 : number
```

Note: The sine of 45° is $1/\sqrt{2}$

```
(/ 1 (value sqrt2))
0.7071067811865475 : number
```

```
(sin (value pi))
2.4821693935912637e-16 : number   \* should be 0 *\
```

```
        cos : number --> number
```
Input: A number X.

Output: cosine of X – a value between -1 and +1, inclusive.

Example:

Note: The cosine of 60° is ½.
```
(cos (degs->rad 60))
0.5000000000000001 : number
```

```
        tan : number --> number
```
Input: A number X.

Output: tangent of X.

Examples:
```
(tan (degs->rad 60))
1.7320508075688765 : number
```

Note: The tangent of 60° is √3.
```
(sqrt 3)
1.7320508075688774 : number
```

3.7 Inverse Trigonometric Functions

```
        asin : number --> number
```
Input: A number X, where $-1 \leq X \leq +1$.

Output: arcsine of X – a value between $-\pi/2$ and $+\pi/2$, inclusive, or an error message.

Examples:
```
(asin (sin 0.89))
0.8899999999999999 : number
```

but

```
(asin (sin 2.89))
0.2515926535897928 : number
```

which is equal to

```
(- (value pi) 2.89)
0.251592653589793 : number
```

```
(asin 1.5)
asin(x) for |x| > 1!
```

```
        acos : number --> number
```
Input: A number X, where $-1 \leq X \leq +1$.

Output: arccosine of X – a value between 0 and π, inclusive, or an error message.

Examples:
```
(acos -1)
3.141592653589793 : number \* should be π = 3.14159265358979323846 *\
```

```
(acos 1)
0.0 : number
```

```
(acos 1.3)
acos(x) for |x| > 1!
```

```
(cos (acos 0.789))
0.7890000000000001 : number
```

`atan : number --> number`

Input: A number X.

Output: arctangent of X – a value strictly between –π/2 and +π/2.

Examples:

```
(atan -1.5e15)
-1.570796326794896 : number \* close to -π/2 *\

(atan (tan 0.45))
0.4500000000000001 : number

(tan (atan 2.45))
2.449999999999999 : number
```

`atan2 : number --> number --> number`

Input: Two numbers X and Y.

Output: The two-argument arctangent of Y and X – a value between –π (not included) and +π (inclusive).

Examples:

```
(atan2 3.5 0)
1.5707963267948966 : number

(atan2 3.5 -2.3)
2.152176510599796 : number

(atan2 0 0)
atan2 - undefined

(atan2 0 -4.7)
3.141592653589793 : number
```

3.8 Exponential and Hyperbolic Functions

`exp : number --> number`

Input: A number X.

Output: The value of e^X.

Examples:

```
(exp 1.5)
4.481689070338065 : number

(exp -1.0)
0.3678794411714423 : number

(exp 1.0)
2.7182818284590455 : number

(exp 100.0)
2.688117141816169e43 : number
```

`sinh : number --> number`

Input: A number X.

Output: The value of the hyperbolic sine of X.

Examples:

```
(sinh 10.12)
12417.385397399632 : number
```

```
(sinh -8.23)
-1875.9167427768757 : number

(sinh 0)
0 : number
```

`cosh : number --> number`

Input: A number X.

Output: The value of the hyperbolic cosine of X.

Examples:

```
(cosh 10.12)
12417.385437665756 : number

(cosh -8.23)
1875.917009313206 : number

(cosh 0)
1 : number
```

`tanh : number --> number`

Input: A number X.

Output: The value of the hyperbolic tangent of X.

Examples:

```
(tanh -8.23)
-0.9999998579167795 : number

(tanh 1)
0.761594155955765 : number
```

3.9 Logarithms

Mathematically most important is the inverse of `exp`, called the ***natural logarithm***, or ***logarithm to base e***. It is commonly denoted by $\log_e$ (or simply `log or ln`). By definition

$$\log_e e^x = e^{\log_e x} = x$$

Sometimes, logarithms to other bases, especially 2 and 10 are needed. If the base is `b (> 0)` then

$$\log_b b^x = b^{\log_b x} = x$$

The library provides the following `log` functions

`log : number --> number`

Input: A positive number X.

Output: The value of the natural logarithm of X, or an error message if X ≤ 0.

Examples:

```
(log (value e))
1.0000000000000004 : number

(log (exp 12.45))
12.45 : number

(log -5.1)
log(x) for x < 0!
```

`log2 : number --> number`

Input: A positive number X.

Output: The value of the base-2 logarithm of X, or an error message if X ≤ 0.

Examples:
```
(log2 1024)
9.999999999999998 : number

(log2 10)
3.321928094887361 : number
```

```
        log10 : number --> number
```
Input: A positive number X.

Output: The value of the base-10 logarithm of X, or an error message if X ≤ 0.

Examples:
```
(log10 (power 10 17))
16.999999999999996

(log10 2)
0.3010299956639811
```

```
        log' : number --> number --> number
```
Input: Two positive numbers X and B.

Output: The value of the base-B logarithm of X, or an error message if X ≤ 0 or B ≤ 0.

Examples:
```
(log' 234.89 9)           \* base 9 *\
2.4845513634614647 : number

(log' (expt 17 3.456) 17)  \* base 17 *\
3.456 : number

(log' (expt 1.5 3.456) 1.5) \* base 1.5 *\
3.456 : number
```

13.10 Power Functions

```
        square : number --> number
```
Input: A number X.

Output: The square of X.

Examples:
```
(square 12.89)
166.15210000000002

(square 500)
250000
```

A fast integer power

```
        power : number --> number --> number
```
Input: A number X and an integer N.

Output: The value of X^N or an error message, if N is not an integer.

Note: The computation runs in log N - time.

Examples:
```
(power 1.5 10)
57.6650390625 : number

(power 1.5 10.9)
power - exponent must be an integer
```

```
(power -1.5 -11)
-0.011561019943888409 : number
```

The general power function

```
        expt : number --> number --> number
```

Input: Two numbers X and Y.

Output: The value of X^Y or an error message, if X^Y is not defined or not a real number.

Examples:

```
(expt (value pi) (/ 1 (value pi))) \* π^(1/π)  *\
1.4396194958475907 : number       \* the figures quoted are all correct! *\

(expt -1.5 -1.2)
expt undefined!

(expt -1.5 3)
-3.375 : number

(expt -1.5 0)
1 : number

(expt 0 -3)
expt undefined!
```

Function `expt` can of course be used to compute square roots (and other roots). A special function is provided which computes square roots using an iterative algorithm. The accuracies of the two methods are very similar.

```
        sqrt : number --> number
```

Input: A non-negative number X.

Output: The value of $\sqrt{X}$, or an error message, if X is negative.

Examples:

```
(sqrt 1000000)
1000.0 : number

(expt 1000000 0.5)
999.9999999999994 : number
```

The square root of the golden ratio: $((\sqrt{5} +1)/2)^{½}$ = 1.272 019 649 514 068 962 … (Mathematical Constants)

```
(sqrt(/ (+ (sqrt 5) 1) 2))
1.272019649514069 : number

(expt (/ (+ (expt 5 0.5) 1) 2) 0.5)
1.2720196495140688 : number
```

13.11 Various Functions

The following function, called ‘free the exponent’ splits a number into its mantissa (significand) and exponent.

```
        frexp : number --> (number * number)
```

Input: A number X.

Output: A pair `(@p M E)` such that the relationship $M \times 2^E$ holds, where E is an integer and $0.5 \leq |M| < 1$, or $M = 0$

Examples:

```
(frexp 0.1)
(@p 0.8 -3) : (number * number)

(frexp -256.0)
(@p -0.5 9) : (number * number)

(frexp 0.0)
(@p 0 0) : (number * number)
```

An ‘inverse’ of `frexp`, called ‘load exponent’ is

```
        ldexp : number --> number --> number
```

Input: A number M and an integer E.

Output: The value $M \times 2^E$, or an error message if E is not an integer.

Examples:

```
(ldexp 0.8 -3)
0.1

(ldexp  1.5 9.1)
power - exponent must be an integer

(ldexp 0.95 4)
15.200000000000001
```

The following function called ‘floating-point modulus’ is a generalisation of the remainder function `rem` for integers (see Section 4.1.3)

```
        fmod : number --> number --> number
```

Input: Two numbers A and B.

Output: The value $A - K \times B$, where K is an integer such that the result is 0 or has the same sign as A, and magnitude less than the magnitude of B.

Examples:

```
(fmod 123.45 17.4)
1.6500000000000057 : number

(fmod -417.2 29.8)
0.0 : number
```

Two polyadic functions are available

```
        max : number --> . . . --> number
```

Input: Any number of numerical values A, B, C . . .

Output: The maximum value of the given arguments.

Examples:

```
(max 12.3 -1 33.4567 2.45)
33.4567 : number
```

```
(max 3 2)
3 : number

(max -12)
-12 : number
```

```
       min : number --> . . . --> number
```

Input: Any number of numerical values A, B, C . . .

Output: The minimum value of the given arguments.

Examples:

```
(min 12.3 -1 33.4567 2.45)
-1 : number

(min 3 2)
2 : number

(min -12)
-12 : number
```

4. Integer Functions

4.1 Integer Division and Remainder – A Short Tutorial

4.1.1 Mathematical Background

In general, dividing a whole number by another, does not result in a whole number. For example, 19/5 = 3.8, which is not an integer. In many practical situations it may be necessary to pretend that the result is in fact an integer 'close' to the actual result, as in the situation when 19 eggs have to be distributed equally amongst 5 people. The mathematical answer is to give each person 3.8 eggs, which is clearly impossible and preposterous. In practice, one will probably give each recipient 3 eggs with 4 eggs left over. Thus 19 = 5 × 3 + 4, where 3 is the 'integer quotient 19/5' and 4 is the 'remainder'.

Programming languages often provide operations for finding 'remainders', variously called `mod`, `rem`, `%` (see below), but not always operators for 'integer division'. Examples are LISP and JavaScript. Other programming languages provide both: C/C++, Pascal, Python, and often two different remainder operations (see table in Modulo operation). The semantics of these operations varies from language to language, and `%` in one can have a different meaning in another. This, of course, is confusing and can cause enormous problems when translating programs between languages. The different semantics will be explained below.

The mathematical basis of integer division rests on a fundamental theorem in elementary number theory:

Theorem: Let $a, b \in \mathbb{Z}$ be two integers, with $b > 0$. There exist $q, r \in \mathbb{Z}$, such that

$$a = qb + r, \qquad 0 \le r < b$$

q and r are unique. q is said to be the (***integer***) ***quotient***, and r the ***remainder,*** on dividing a by b.

Notice the condition $b > 0$. The result can be generalised as follows (see Division algorithm)

Theorem: Let $a, b \in \mathbb{Z}$ be two integers, with $b \neq 0$. There exist $q, r \in \mathbb{Z}$, such that

$$a = qb + r, \qquad 0 \le r < |b| \qquad (*)$$

Note that (here) the remainder, r, is always non-negative and satisfies $-b < r < b$. This is known as the ***Euclidean remainder*** (according to Raymond T. Boute, see Modulo operation).

4.1.2 Integer Division and Remainder in Programming Languages

How do programming languages deal with integer division and remainder? The simple answer is: in various ways! All programming languages tend to agree that the remainder should not be greater than the (absolute value of the) divisor, i.e. that $|r| < |b|$. There are a number of possibilities, besides the above, ***Euclidean*** definition of ***quotient and remainder*** (which is only used in Algol68, some versions of Pascal and in Stata). Note that the positivity of the remainder (*) is useful in certain applications, e.g. calendar computations.

To satisfy the condition $|r| < |b|$, it is possible to choose the sign of r. This will then define the value of the quotient q, and hence, the meaning of "integer division".

			Name
1.	$\mathrm{sign}(r) = \mathrm{sign}(a)$	remainder has the same sign as the dividend	***rem***
2.	$\mathrm{sign}(r) = \mathrm{sign}(b)$	remainder has the same sign as the divisor	***mod***
3.	$r \geq 0$	remainder is non-negative (Euclidean)	***%*** (not in common use!)

Some programming languages provide both (1) and (2) [see Modulo operation]

Here is an example that illustrates the three choices

Example: Let $a = \pm 100$, $b = \pm 14$

1. ***Truncated integer division*** – remainder has same sign as the dividend

   Used in versions of C/C++, C#, Java, JavaScript, Lisp

   Here *q = trunc* (*a*/*b*)

   r = rem (*a*, *b*)

a	*b*	*a*/*b*	*q*	*r*	*a* = *q* × *b* + *r*
100	14	7.142…	7	2	100 = 7 × 14 + 2
100	-14	-7.142…	-7	2	100 = (-7) × (-14) + 2
-100	14	-7.142…	-7	-2	-100 = (-7) × 14 –2
-100	-14	7.142…	7	-2	-100 = 7 × (-14) –2

2. ***Floored integer division*** – remainder has same sign as the divisor

   Used in Clojure, Mathematica, Python, Lisp.

   Here *q = floor (a/b)*

   r = mod (*a*, *b*) This is perhaps the most useful definition.

a	*b*	*a*/*b*	*q*	*r*	*a* = *q* × *b* + *r*
100	14	7.142…	7	2	100 = 7 × 14 + 2
100	-14	-7.142…	-8	-12	100 = (-8) × (-14) - 12
-100	14	-7.142…	-8	12	-100 = (-8) × 14 + 12
-100	-14	7.142…	7	-2	-100 = 7 × (-14) – 2

3. ***Euclidean integer division*** – remainder is always non-negative

   Used rarely, because it is awkward to implement (Algol68, some versions of Pascal, Stata)

   Here

$$q = \begin{cases} \mathit{floor}\ (a/b) & b \geq 0 \\ \mathit{ceiling}\ (a/b) & b \leq 0 \end{cases}$$

r = % (*a*, *b*)

a	*b*	*a*/*b*	*q*	*r*	*a* = *q* × *b* + *r*
100	14	7.142…	7	2	100 = 7 × 14 + 2
100	-14	-7.142…	-7	2	100 = (-7) × (-14) + 2
-100	14	-7.142…	-8	12	-100 = (-8) × 14 + 12
-100	-14	7.142…	8	12	-100 = 8 × (-14) + 12

We see that

for $a > 0$, $b > 0$ (1), (2) and (3) agree

for $a > 0$, $b < 0$, (1) and (3) agree

for $a < 0$, $b > 0$, (2) and (3) agree

for $a < 0$, $b < 0$, (1) and (2) agree

4.1.3 Integer Division and Remainder in Shen

The maths library provides all three types of integer division with corresponding remainder. As one frequently requires both the integer quotient and the remainder, functions are available that return both. This avoids unnecessary re-computation of the quotient.

Note that all references to `number` in the signatures below are references to integers. Also note that all functions in this section are evaluated using integer arithmetic only (provided, of course, that the platform supports it)! On a platform, like Common Lisp, 'big integers' are available, and when used as function arguments on this platform, integer function invocations will produce correct results. This may not be the case on other platforms. See examples below.

```
/rem :  number --> number --> (number * number)
```

Input: Two integers A and B.

Output: The pair consisting of the truncated integer quotient and remainder,
or an error message if B is 0 or not an integer

Examples:

```
(/rem -1000 -33)
(@p 30 -10) : (number * number)

(/rem 1000 33.3)
divisor must be an integer!
```

```
trunc-div :  number --> number --> number
```

Input: Two integers A and B.

Output: The truncated integer quotient of A and B, or an error message if B is 0 or not an integer.

Note: `(trunc-div A B)` is mathematically equivalent to `(trunc (/ A B))`, but using this would involve floating-point arithmetic, and hence introduce inaccuracies.

Examples:

```
(trunc-div 100 -6)
-16 : number

(trunc-div 10000000000000000000001 33) \* correct *\
303030303030303030303 : number

(trunc (/ 10000000000000000000001 33))
303030303030302998528 : number         \* not correct *\
```

Note: Due to the large integer, these two examples will not work correctly in JS-Shen.

```
rem :  number --> number --> number
```

Input: Two integers A and B.

Output: The value of A – B × (trunc-div A B), or an error message if B is 0 or not an integer.

Note: The given expression is not the basis for computing `rem`.

Example:

```
(rem 100 -6)
4 : number
```

`/mod :  number --> number --> (number * number)`

Input: Two integers A and B.

Output: The pair consisting of the floored integer quotient and remainder,
or an error message if B is 0 or not an integer

Examples:

```
(/mod 1000 -33)
(@p -31 -23) : (number * number)

(/mod -1000 -33)
(@p 30 -10) : (number * number)

(/mod 1000 33.3)
divisor must be an integer!
```

`div :  number --> number --> number`

Input: Two integers A and B.

Output: The floored integer quotient of A and B, or an error message if B is 0 or not an integer.

Note: `(div A B)` could be computed as `(floor (/ A B))`, but this would involve floating-point arithmetic. (See note under `trunc-div`)

Examples:

```
(div 100 -6)
-17 : number

(div 100 -6.2)
divisor must be an integer!
```

`mod :  number --> number --> number`

Input: Two integers A and B.

Output: The value of A – B × (div A B), or an error message if B is 0 or not an integer.

Note: The given expression is not the basis for computing `mod`.

Example:

```
(mod 100 -6)
-2 : number
```

`/% :  number --> number --> (number * number)`

Input: Two integers A and B.

Output: The pair consisting of the Euclidean integer quotient and remainder,
or an error message if B is 0 or not an integer

Examples:

```
(/% 1000 -33)
(@p -30 10) : (number * number)

(/% -1000 -33)
(@p 31 23) : (number * number)

(/% 1000 33.3)
divisor must be an integer!
```

`div-eucl :  number --> number --> number`

Input: Two integers A and B.

Output: The Euclidean integer quotient of A and B, or an error message if B is 0 or not an integer.

Examples:

```
(div-eucl 100 -6)
-16 : number
```

```
        % :  number --> number --> number
```
Input: Two integers A and B.

Output: The value of A – B × (div-eucl A B), or an error message if B is 0 or not an integer.

Note: The given expression is not the basis for computing `%`.

Example:
```
(% 100 -6)
4 : number
```

Note: In the above functions, it is stated that both inputs, A and B, should be integers, and if B is not, then an error will result. If A is not an integer then the functions will produce results, which are actually meaningful. Here is the explanation for `/rem`. The behaviour of `/mod` and `/%` can be explained similarly.

Example:
```
(/rem -101.45 3)
(@p -33 -2.450000000000003) : (number * number)
```
We observe that `-101.45 = 3 × (-33) - 2.45`. In general, the quotient q = trunc (A/B) and the remainder is A – B × q.

4.2 Basic Number Theory Functions

Closely related to the computation of remainders is the notion of divisibility: An integer N is divisible by an integer M, precisely when the remainder on dividing N by M is 0. It is clearly irrelevant which of the three remainders discussed above one takes. For reasons of efficiency a special function is provided

```
        divisible-by? :  number --> number --> boolean
```
Input: Two integers A > 0 and B > 0.

Output: `true` if B divides A, otherwise `false`.

Note: For the sake of efficiency, the inputs are not tested for validity. Illegal inputs will cause the program to crash!

Examples:
```
(divisible-by? 10175 25)
true : boolean

(divisible-by? 101 13)
false : boolean
```

Also included is a simple primality test (based on trial division), which makes use of the preceding function.

```
        prime? :  number --> boolean
```
Input: A positive integer N.

Output: `true` if N is a prime number, otherwise `false`.

Examples:
```
(prime? 16127)
true : boolean

(prime? 11111)
false : boolean
```

This simple primality-test function is reasonably fast. On platforms supporting only 32-bit integers, the largest prime number that can be represented is the Mersenne prime $2^{31} – 1$ = 2147483647. On my PC (using CL-Shen) the run time is just over ½ second. Clojure-Shen is a lot faster (see timings below)

```
(time (prime? 2147483647))
run time: 0.62 secs  \* truncated *\    \* 0.36 secs in Clojure-Shen *\
true : boolean
```

For larger primes the run time may become prohibitive

```
(time (prime? 274876858367))
run time: 9.12 secs             \* 2.76 secs in Clojure-Shen *\
true : boolean

(time (prime? 4398050705407))
run time: 41.77 secs            \* 12.00 secs in Clojure-Shen *\
true : boolean
```

Two polyadic functions for the 'greatest common divisor' and the 'least common multiple' are available

```
      gcd : number --> . . . --> number
```

Input: Any number of integer values A, B, C . . .

Output: The greatest common divisor of the given arguments. (Always ≥ 0).

Examples:

```
(gcd 123 78 888888)
3 : number

(gcd 2262 -2726 1682 -1414562)
58 : number
```

```
      lcm : number --> . . . --> number
```

Input: Any number of integer values A, B, C . . .

Output: The least common multiple of the given arguments. (Always ≥ 0).

Examples:

```
(lcm 2262 -2726 1682)
-3083106 : number

(lcm 12 34)
204 : number

(lcm 12)
12 : number
```