Mathematical Operations and Elementary Functions

Julia provides a complete collection of basic arithmetic and bitwise operators across all of its numeric primitive types, as well as providing portable, efficient implementations of a comprehensive collection of standard mathematical functions.

Arithmetic Operators

The following arithmetic operators are supported on all primitive numeric types:

Expression	Name	Description
`+x`	unary plus	the identity operation
`-x`	unary minus	maps values to their additive inverses
`x + y`	binary plus	performs addition
`x - y`	binary minus	performs subtraction
`x * y`	times	performs multiplication
`x / y`	divide	performs division
`x ÷ y`	integer divide	x / y, truncated to an integer
`x \ y`	inverse divide	equivalent to `y / x`
`x ^ y`	power	raises `x` to the `y`th power
`x % y`	remainder	equivalent to `rem(x,y)`

as well as the negation on `Bool` types:

Expression	Name	Description
`!x`	negation	changes `true` to `false` and vice versa

A numeric literal placed directly before an identifier or parentheses, e.g. `2x` or `2(x+y)`, is treated as a multiplication, except with higher precedence than other binary operations. See Numeric Literal Coefficients for details.

Julia's promotion system makes arithmetic operations on mixtures of argument types "just work" naturally and automatically. See Conversion and Promotion for details of the promotion system.

Here are some simple examples using arithmetic operators:

```
julia> 1 + 2 + 3
6

julia> 1 - 2
-1

julia> 3*2/12
0.5
```

(By convention, we tend to space operators more tightly if they get applied before other nearby operators. For instance, we would generally write `-x + 2` to reflect that first `x` gets negated, and then `2` is added to that result.)

When used in multiplication, `false` acts as a *strong zero*:

```
julia> NaN * false
0.0

julia> false * Inf
0.0
```

This is useful for preventing the propagation of `NaN` values in quantities that are known to be zero. See Knuth (1992) for motivation.

Bitwise Operators

The following bitwise operators are supported on all primitive integer types:

Expression	Name
`~x`	bitwise not
`x & y`	bitwise and

`x | y`	bitwise or
`x ⊻ y`	bitwise xor (exclusive or)
`x >>> y`	logical shift right
`x >> y`	arithmetic shift right
`x << y`	logical/arithmetic shift left

Here are some examples with bitwise operators:

```
julia> ~123
-124

julia> 123 & 234
106

julia> 123 | 234
251

julia> 123 ⊻ 234
145

julia> xor(123, 234)
145

julia> ~UInt32(123)
0xffffff84

julia> ~UInt8(123)
0x84
```

Updating operators

Every binary arithmetic and bitwise operator also has an updating version that assigns the result of the operation back into its left operand. The updating version of the binary operator is formed by placing a `=` immediately after the operator. For example, writing `x += 3` is equivalent to writing `x = x + 3`:

```
julia> x = 1
1
```

```
julia> x += 3
4

julia> x
4
```

The updating versions of all the binary arithmetic and bitwise operators are:

```
+=  -=  *=  /=  \=  ÷=  %=  ^=  &=  |=  ⊻=  >>>=  >>=  <<=
```

Note

An updating operator rebinds the variable on the left-hand side. As a result, the type of the variable may change.

```
julia> x = 0x01; typeof(x)
UInt8

julia> x *= 2 # Same as x = x * 2
2

julia> typeof(x)
Int64
```

Vectorized "dot" operators

For *every* binary operation like `^`, there is a corresponding "dot" operation `.^` that is *automatically* defined to perform `^` element-by-element on arrays. For example, `[1,2,3] ^ 3` is not defined, since there is no standard mathematical meaning to "cubing" a (non-square) array, but `[1,2,3] .^ 3` is defined as computing the elementwise (or "vectorized") result `[1^3, 2^3, 3^3]`. Similarly for unary operators like `!` or `√`, there is a corresponding `.√` that applies the operator elementwise.

```
julia> [1,2,3] .^ 3
3-element Array{Int64,1}:
  1
  8
 27
```

More specifically, `a .^ b` is parsed as the "dot" call `(^).(a,b)`, which performs a broadcast operation:

it can combine arrays and scalars, arrays of the same size (performing the operation elementwise), and even arrays of different shapes (e.g. combining row and column vectors to produce a matrix). Moreover, like all vectorized "dot calls," these "dot operators" are *fusing*. For example, if you compute `2 .* A.^2 .+ sin.(A)` (or equivalently `@. 2A^2 + sin(A)`, using the `@.` macro) for an array `A`, it performs a *single* loop over `A`, computing `2a^2 + sin(a)` for each element of `A`. In particular, nested dot calls like `f.(g.(x))` are fused, and "adjacent" binary operators like `x .+ 3 .* x.^2` are equivalent to nested dot calls `(+).(x, (*).(3, (^).(x, 2)))`.

Furthermore, "dotted" updating operators like `a .+= b` (or `@. a += b`) are parsed as `a .= a .+ b`, where `.=` is a fused *in-place* assignment operation (see the dot syntax documentation).

Note the dot syntax is also applicable to user-defined operators. For example, if you define `⊗(A,B) = kron(A,B)` to give a convenient infix syntax `A ⊗ B` for Kronecker products (`kron`), then `[A,B] .⊗ [C,D]` will compute `[A⊗C, B⊗D]` with no additional coding.

Combining dot operators with numeric literals can be ambiguous. For example, it is not clear whether `1.+x` means `1. + x` or `1 .+ x`. Therefore this syntax is disallowed, and spaces must be used around the operator in such cases.

Numeric Comparisons

Standard comparison operations are defined for all the primitive numeric types:

Operator	Name
`==`	equality
`!=`, `≠`	inequality
`<`	less than
`<=`, `≤`	less than or equal to
`>`	greater than
`>=`, `≥`	greater than or equal to

Here are some simple examples:

```
julia> 1 == 1
true
```

```
julia> 1 == 2
false

julia> 1 != 2
true

julia> 1 == 1.0
true

julia> 1 < 2
true

julia> 1.0 > 3
false

julia> 1 >= 1.0
true

julia> -1 <= 1
true

julia> -1 <= -1
true

julia> -1 <= -2
false

julia> 3 < -0.5
false
```

Integers are compared in the standard manner – by comparison of bits. Floating-point numbers are compared according to the IEEE 754 standard:

- Finite numbers are ordered in the usual manner.
- Positive zero is equal but not greater than negative zero.
- `Inf` is equal to itself and greater than everything else except `NaN`.
- `-Inf` is equal to itself and less than everything else except `NaN`.
- `NaN` is not equal to, not less than, and not greater than anything, including itself.

The last point is potentially surprising and thus worth noting:

```
julia> NaN == NaN
false
```

```
julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false
```

and can cause headaches when working with arrays:

```
julia> [1 NaN] == [1 NaN]
false
```

Julia provides additional functions to test numbers for special values, which can be useful in situations like hash key comparisons:

Function	Tests if
`isequal(x, y)`	`x` and `y` are identical
`isfinite(x)`	`x` is a finite number
`isinf(x)`	`x` is infinite
`isnan(x)`	`x` is not a number

`isequal` considers `NaN`s equal to each other:

```
julia> isequal(NaN, NaN)
true

julia> isequal([1 NaN], [1 NaN])
true

julia> isequal(NaN, NaN32)
true
```

`isequal` can also be used to distinguish signed zeros:

```
julia> -0.0 == 0.0
true
```

```
julia> isequal(-0.0, 0.0)
false
```

Mixed-type comparisons between signed integers, unsigned integers, and floats can be tricky. A great deal of care has been taken to ensure that Julia does them correctly.

For other types, `isequal` defaults to calling `==`, so if you want to define equality for your own types then you only need to add a `==` method. If you define your own equality function, you should probably define a corresponding `hash` method to ensure that `isequal(x,y)` implies `hash(x) == hash(y)`.

Chaining comparisons

Unlike most languages, with the notable exception of Python, comparisons can be arbitrarily chained:

```
julia> 1 < 2 <= 2 < 3 == 3 > 2 >= 1 == 1 < 3 != 5
true
```

Chaining comparisons is often quite convenient in numerical code. Chained comparisons use the `&&` operator for scalar comparisons, and the `&` operator for elementwise comparisons, which allows them to work on arrays. For example, `0 .< A .< 1` gives a boolean array whose entries are true where the corresponding elements of `A` are between 0 and 1.

Note the evaluation behavior of chained comparisons:

```
julia> v(x) = (println(x); x)
v (generic function with 1 method)

julia> v(1) < v(2) <= v(3)
2
1
3
true

julia> v(1) > v(2) <= v(3)
2
1
false
```

The middle expression is only evaluated once, rather than twice as it would be if the expression were written as `v(1) < v(2) && v(2) <= v(3)`. However, the order of evaluations in a chained comparison is undefined. It is strongly recommended not to use expressions with side effects (such as printing) in chained comparisons. If side effects are required, the short-circuit `&&` operator should be used explicitly (see Short-Circuit Evaluation).

Elementary Functions

Julia provides a comprehensive collection of mathematical functions and operators. These mathematical operations are defined over as broad a class of numerical values as permit sensible definitions, including integers, floating-point numbers, rationals, and complex numbers, wherever such definitions make sense.

Moreover, these functions (like any Julia function) can be applied in "vectorized" fashion to arrays and other collections with the dot syntax `f.(A)`, e.g. `sin.(A)` will compute the sine of each element of an array `A`.

Operator Precedence and Associativity

Julia applies the following order and associativity of operations, from highest precedence to lowest:

Category	Operators	Associativity
Syntax	`.` followed by `::`	Left
Exponentiation	`^`	Right
Unary	`+ - √`	Right[1]
Bitshifts	`<< >> >>>`	Left
Fractions	`//`	Left
Multiplication	`* / % & \ ÷`	Left[2]
Addition	`+ - | ⊻`	Left[2]
Syntax	`: ..`	Left
Syntax	`|>`	Left
Syntax	`<|`	Right
Comparisons	`> < >= <= == === != !== <:`	Non-associative

Control flow	`&&` followed by `||` followed by `?`	Right
Pair	`=>`	Right
Assignments	`= += -= *= /= //= \= ^= ÷= %= |= &= ⊻= <<= >>= >>>=`	Right

For a complete list of *every* Julia operator's precedence, see the top of this file: `src/julia-parser.scm`

Numeric literal coefficients, e.g. `2x`, are treated as multiplications with higher precedence than any other binary operation, and also have higher precedence than `^`.

You can also find the numerical precedence for any given operator via the built-in function `Base.operator_precedence`, where higher numbers take precedence:

```
julia> Base.operator_precedence(:+), Base.operator_precedence(:*), Base.operator_pre
(11, 12, 17)

julia> Base.operator_precedence(:sin), Base.operator_precedence(:+=), Base.operator_
(0, 1, 1)
```

A symbol representing the operator associativity can also be found by calling the built-in function `Base.operator_associativity`:

```
julia> Base.operator_associativity(:-), Base.operator_associativity(:+), Base.operat
(:left, :none, :right)

julia> Base.operator_associativity(:⊗), Base.operator_associativity(:sin), Base.oper
(:left, :none, :right)
```

Note that symbols such as `:sin` return precedence `0`. This value represents invalid operators and not operators of lowest precedence. Similarly, such operators are assigned associativity `:none`.

Numerical Conversions

Julia supports three forms of numerical conversion, which differ in their handling of inexact conversions.

- The notation `T(x)` or `convert(T,x)` converts `x` to a value of type `T`.
    - If `T` is a floating-point type, the result is the nearest representable value, which could be positive or negative infinity.

    - If `T` is an integer type, an `InexactError` is raised if `x` is not representable by `T`.
- `x % T` converts an integer `x` to a value of integer type `T` congruent to `x` modulo `2^n`, where `n` is the number of bits in `T`. In other words, the binary representation is truncated to fit.
- The Rounding functions take a type `T` as an optional argument. For example, `round(Int,x)` is a shorthand for `Int(round(x))`.

The following examples show the different forms.

```
julia> Int8(127)
127

julia> Int8(128)
ERROR: InexactError: trunc(Int8, 128)
Stacktrace:
[...]

julia> Int8(127.0)
127

julia> Int8(3.14)
ERROR: InexactError: Int8(3.14)
Stacktrace:
[...]

julia> Int8(128.0)
ERROR: InexactError: Int8(128.0)
Stacktrace:
[...]

julia> 127 % Int8
127

julia> 128 % Int8
-128

julia> round(Int8,127.4)
127

julia> round(Int8,127.6)
ERROR: InexactError: trunc(Int8, 128.0)
Stacktrace:
[...]
```

See Conversion and Promotion for how to define your own conversions and promotions.

Rounding functions

Function	Description	Return type
`round(x)`	round `x` to the nearest integer	`typeof(x)`
`round(T, x)`	round `x` to the nearest integer	`T`
`floor(x)`	round `x` towards `-Inf`	`typeof(x)`
`floor(T, x)`	round `x` towards `-Inf`	`T`
`ceil(x)`	round `x` towards `+Inf`	`typeof(x)`
`ceil(T, x)`	round `x` towards `+Inf`	`T`
`trunc(x)`	round `x` towards zero	`typeof(x)`
`trunc(T, x)`	round `x` towards zero	`T`

Division functions

Function	Description
`div(x,y)`, `x÷y`	truncated division; quotient rounded towards zero
`fld(x,y)`	floored division; quotient rounded towards `-Inf`
`cld(x,y)`	ceiling division; quotient rounded towards `+Inf`
`rem(x,y)`	remainder; satisfies `x == div(x,y)*y + rem(x,y)`; sign matches `x`
`mod(x,y)`	modulus; satisfies `x == fld(x,y)*y + mod(x,y)`; sign matches `y`
`mod1(x,y)`	`mod` with offset 1; returns `r∈(0,y]` for `y>0` or `r∈[y,0)` for `y<0`, where `mod(r, y) == mod(x, y)`
`mod2pi(x)`	modulus with respect to 2pi; `0 <= mod2pi(x) < 2pi`
`divrem(x,y)`	returns `(div(x,y),rem(x,y))`
`fldmod(x,y)`	returns `(fld(x,y),mod(x,y))`

`gcd(x,y...)`	greatest positive common divisor of `x`, `y`,...
`lcm(x,y...)`	least positive common multiple of `x`, `y`,...

Sign and absolute value functions

Function	Description
`abs(x)`	a positive value with the magnitude of `x`
`abs2(x)`	the squared magnitude of `x`
`sign(x)`	indicates the sign of `x`, returning -1, 0, or +1
`signbit(x)`	indicates whether the sign bit is on (true) or off (false)
`copysign(x,y)`	a value with the magnitude of `x` and the sign of `y`
`flipsign(x,y)`	a value with the magnitude of `x` and the sign of `x*y`

Powers, logs and roots

Function	Description
`sqrt(x)`, `√x`	square root of `x`
`cbrt(x)`, `∛x`	cube root of `x`
`hypot(x,y)`	hypotenuse of right-angled triangle with other sides of length `x` and `y`
`exp(x)`	natural exponential function at `x`
`expm1(x)`	accurate `exp(x)-1` for `x` near zero
`ldexp(x,n)`	`x*2^n` computed efficiently for integer values of `n`
`log(x)`	natural logarithm of `x`
`log(b,x)`	base `b` logarithm of `x`
`log2(x)`	base 2 logarithm of `x`

`log10(x)`	base 10 logarithm of `x`
`log1p(x)`	accurate `log(1+x)` for `x` near zero
`exponent(x)`	binary exponent of `x`
`significand(x)`	binary significand (a.k.a. mantissa) of a floating-point number `x`

For an overview of why functions like `hypot`, `expm1`, and `log1p` are necessary and useful, see John D. Cook's excellent pair of blog posts on the subject: expm1, log1p, erfc, and hypot.

Trigonometric and hyperbolic functions

All the standard trigonometric and hyperbolic functions are also defined:

```
sin    cos    tan    cot    sec    csc
sinh   cosh   tanh   coth   sech   csch
asin   acos   atan   acot   asec   acsc
asinh  acosh  atanh  acoth  asech  acsch
sinc   cosc
```

These are all single-argument functions, with `atan` also accepting two arguments corresponding to a traditional `atan2` function.

Additionally, `sinpi(x)` and `cospi(x)` are provided for more accurate computations of `sin(pi*x)` and `cos(pi*x)` respectively.

In order to compute trigonometric functions with degrees instead of radians, suffix the function with `d`. For example, `sind(x)` computes the sine of `x` where `x` is specified in degrees. The complete list of trigonometric functions with degree variants is:

```
sind   cosd   tand   cotd   secd   cscd
asind  acosd  atand  acotd  asecd  acscd
```

Special functions

Many other special mathematical functions are provided by the package SpecialFunctions.jl.

- 1 The unary operators `+` and `-` require explicit parentheses around their argument to disambiguate them from the operator `++`, etc. Other compositions of unary operators are parsed with right-associativity, e. g., `√√-a` as `√(√(-a))`.

- 2 The operators `+`, `++` and `*` are non-associative. `a + b + c` is parsed as `+(a, b, c)` not `+(+(a, b), c)`. However, the fallback methods for `+(a, b, c, d...)` and `*(a, b, c, d...)` both default to left-associative evaluation.

« Integers and Floating-Point Numbers Complex and Rational Numbers »