Math Library

Version 9.4.0.2

Neil Toronto <ntoronto@racket-lang.org> Jens Axel Søgaard <jensaxel@soegaard.net>

October 20, 2026

```
(require math)
```
package: `math-lib`

The `math` library provides functions and data structures useful for working with numbers and collections of numbers. These include

- `math/base`: Constants and elementary functions
- `math/flonum`: Flonum functions, including high-accuracy support
- `math/special-functions`: Special (i.e. non-elementary) functions
- `math/bigfloat`: Arbitrary-precision floating-point functions
- `math/number-theory`: Number-theoretic functions
- `math/array`: Functional arrays for operating on large rectangular data sets
- `math/matrix`: Linear algebra functions for arrays
- `math/distributions`: Probability distributions
- `math/statistics`: Statistical functions

With this library, we hope to support a wide variety of applied mathematics in Racket, including simulation, statistical inference, signal processing, and combinatorics. If you find it lacking for your variety of mathematics, please

- Visit the Math Library Features wiki page to see what is planned.
- Contact us or post to one of the mailing lists to make suggestions or submit patches.

This is a Typed Racket library. It is most efficient to use it in Typed Racket, so that contracts are checked statically. However, almost all of it can be used in untyped Racket. Exceptions and performance warnings are in **bold text**.

1 Constants and Elementary Functions

```
(require math/base)
```
package: math-lib

For convenience, `math/base` re-exports `racket/math` as well as providing the values document below.

In general, the functions provided by `math/base` are *elementary* functions, or those functions that can be defined in terms of a finite number of arithmetic operations, logarithms, exponentials, trigonometric functions, and constants. For others, see `math/special-functions` and `math/distributions`.

1.1 Constants

If you need more accurate approximations than the following flonums, see, for example, `phi.bf` and `bigfloat->rational`.

`phi.0 : Positive-Flonum`

An approximation of φ, the golden ratio.

```
> phi.0
1.618033988749895
```

`euler.0 : Positive-Flonum`

An approximation of e, or Euler's number.

```
> euler.0
2.718281828459045
> (exp 1)
2.718281828459045
```

`gamma.0 : Positive-Flonum`

An approximation of γ, the Euler-Mascheroni constant.

```
> gamma.0
0.5772156649015329
```

`catalan.0 : Positive-Flonum`

An approximation of G, or Catalan's constant.

```
> catalan.0
0.915965594177219
```

1.2 Functions

```
(float-complex? v) → Boolean
  v : Any
```

Returns #t when v is of type Float-Complex. Analogous to flonum?.

```
(number->float-complex x) → Float-Complex
  x : Number
```

Returns a new complex number with a flonum real part and a flonum imaginary part. Analogous to real->double-flonum.

```
(power-of-two? x) → Boolean
  x : Real
```

Returns #t when x is an integer power of 2.

Examples:

```
> (power-of-two? 1.0)
#t
> (power-of-two? 1/2)
#t
> (power-of-two? (flnext 2.0))
#f
```

```
(asinh z) → Number
  z : Number
(acosh z) → Number
  z : Number
(atanh z) → Number
  z : Number
```

The inverses of sinh, cosh, and tanh, which are defined in racket/math (and re-exported by math/base).

```
(sum xs) → Real
  xs : (Listof Real)
```

Like (apply + xs), but incurs rounding error only once when adding inexact numbers. (In fact, the inexact numbers in xs are summed separately using flsum.)

1.3 Random Number Generation

```
(random-natural k [p]) → Natural
  k : Integer
  p : Pseudo-Random-Generator
    = (current-pseudo-random-generator)
```

Returns a random natural number less than `k`, which must be positive. Use `(random-natural k)` instead of `(random k)` when `k` could be larger than `4294967087`.

Uses `p` for the underlying generation of randomness.

```
(random-integer a b [p]) → Integer
  a : Integer
  b : Integer
  p : Pseudo-Random-Generator
    = (current-pseudo-random-generator)
```

Returns a random integer `n` such that `(<= a n)` and `(< n b)`.

Uses `p` for the underlying generation of randomness.

```
(random-bits num [p]) → Natural
  num : Integer
  p : Pseudo-Random-Generator
    = (current-pseudo-random-generator)
```

Returns a random natural smaller than `(expt 2 num)`; `num` must be non-negative. For powers of two, this is faster than using `random-natural`, which is implemented in terms of `random-bits`, using biased rejection sampling.

As an example of use, the significands of the numbers returned by `bfrandom` are chosen by `(random-bits (bf-precision))`.

Uses `p` for the underlying generation of randomness.

1.4 Measuring Error

```
(absolute-error x r) → Real
  x : Real
  r : Real
```

Usually computes `(abs (- x r))` using exact rationals, but handles non-rational reals such as `+inf.0` specially.

Examples:

```
> (absolute-error 1/2 1/2)
0
> (absolute-error 0.14285714285714285 1/7)
7.93016446160826e-18
> (absolute-error +inf.0 +inf.0)
0.0
> (absolute-error +inf.0 +nan.0)
+inf.0
> (absolute-error 1e-20 0.0)
1e-20
> (absolute-error (- 1.0 (fl 4999999/5000000)) 1/5000000)
5.751132903242251e-18
```

```
(relative-error x r) → Real
  x : Real
  r : Real
```

Measures how close an approximation `x` is to the correct value `r`, relative to the magnitude of `r`.

This function usually computes `(abs (/ (- x r) r))` using exact rationals, but handles non-rational reals such as `+inf.0` specially, as well as `r = 0`.

Examples:

```
> (relative-error 1/2 1/2)
0
> (relative-error 0.14285714285714285 1/7)
5.551115123125783e-17
> (relative-error +inf.0 +inf.0)
0.0
> (relative-error +inf.0 +nan.0)
+inf.0
> (relative-error 1e-20 0.0)
+inf.0
> (relative-error (- 1.0 (fl 4999999/5000000)) 1/5000000)
2.8755664516211255e-11
```

In the last two examples, relative error is high because the result is near zero. (Compare the same examples with `absolute-error`.) Because flonums are particularly dense near zero, this makes relative error better than absolute error for measuring the error in a flonum approximation. An even better one is error in ulps; see `flulp-error`.

2 Flonums

```
(require math/flonum)
```
package: math-lib

For convenience, math/flonum re-exports racket/flonum as well as providing the functions document below.

2.1 Additional Flonum Functions

```
(fl x) → Flonum
  x : Real
```

Equivalent to (real->double-flonum x), but much easier to read and write.

Examples:

```
> (fl 1/2)
0.5
> (fl 0.5)
0.5
> (fl #i0.5)
0.5
```

Note that exact->inexact does not always convert a Real to a Flonum:

```
> (exact->inexact #i0.5)
0.5
> (flabs (exact->inexact #i0.5))
0.5
```

You should prefer fl over exact->inexact, especially in Typed Racket code.

```
(flsgn x) → Flonum
  x : Flonum
(fleven? x) → Boolean
  x : Flonum
(flodd? x) → Boolean
  x : Flonum
```

Like sgn, even? and odd?, but restricted to flonum input.

Examples:

```
> (map flsgn '(-2.0 -0.0 0.0 2.0))
'(-1.0 0.0 0.0 1.0)
> (map fleven? '(2.0 1.0 0.5))
'(#t #f #f)
> (map flodd? '(2.0 1.0 0.5))
'(#f #t #f)
```

```
(flrational? x) → Boolean
  x : Flonum
(flinfinite? x) → Boolean
  x : Flonum
(flnan? x) → Boolean
  x : Flonum
(flinteger? x) → Boolean
  x : Flonum
```

Like `rational?`, `infinite?`, `nan?` and `integer?`, but restricted to flonum input. In Typed Racket, these are 2-3 times faster as well.

```
(flhypot x y) → Flonum
  x : Flonum
  y : Flonum
```

Computes `(flsqrt (+ (* x x) (* y y)))` in way that overflows only when the answer is too large.

Examples:

```
> (flsqrt (+ (* 1e+200 1e+200) (* 1e+199 1e+199)))
+inf.0
> (flhypot 1e+200 1e+199)
1.004987562112089e+200
```

```
(flsum xs) → Flonum
  xs : (Listof Flonum)
```

Like `(apply + xs)`, but incurs rounding error only once.

Examples:

```
> (+ 1.0 1e-16)
1.0
> (+ (+ 1.0 1e-16) 1e-16)
1.0
> (flsum '(1.0 1e-16 1e-16))
1.0000000000000002
```

The sum function does the same for heterogenous lists of reals.

Worst-case time complexity is $O(n^2)$, though the pathological inputs needed to observe quadratic time are exponentially improbable and are hard to generate purposely. Expected time complexity is $O(n \log(n))$.

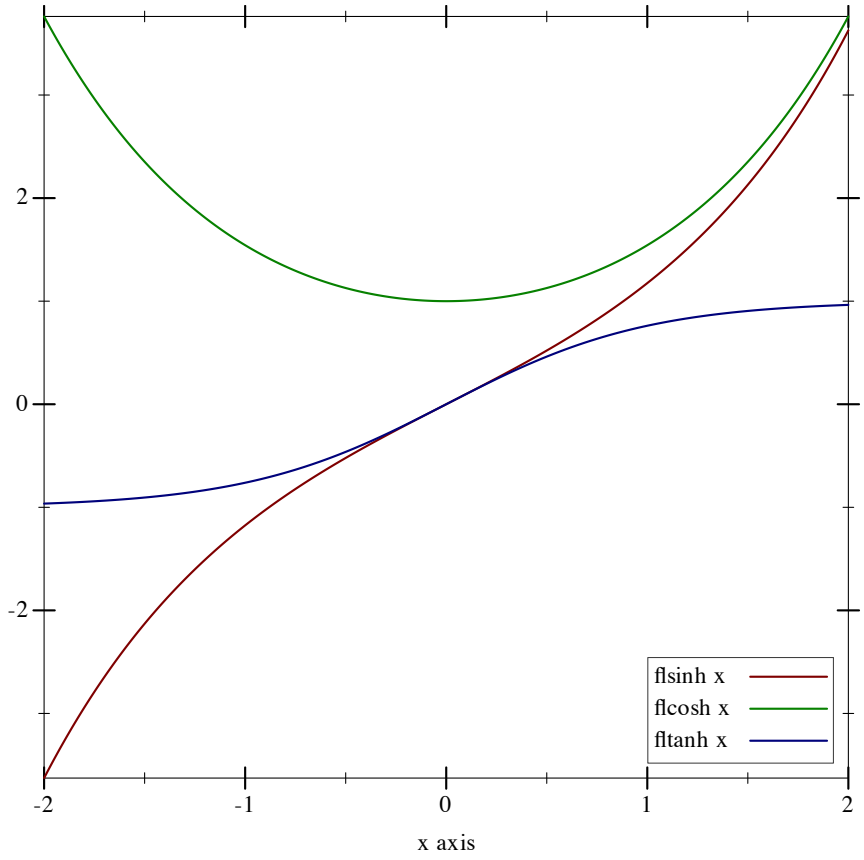
See flvector-sums for a variant that computes all the partial sums in *xs*.

```
(flsinh x) → Flonum
  x : Flonum
(flcosh x) → Flonum
  x : Flonum
(fltanh x) → Flonum
  x : Flonum
```

Return the hyperbolic sine, cosine and tangent of *x*, respectively.

Example:

```
> (plot (list
         (function (compose flsinh fl) #:label "flsinh x")
         (function (compose flcosh fl) #:label "flcosh
x" #:color 2)
         (function (compose fltanh fl) #:label "fltanh
x" #:color 3))
        #:x-min -2 #:x-max 2 #:y-label #f #:legend-anchor 'bottom-
right)
```



Maximum observed error is 2 ulps, making these functions (currently) much more accurate than their `racket/math` counterparts. They also return sensible values on the largest possible domain.

```
(flasinh y) → Flonum
  y : Flonum
(flacosh y) → Flonum
  y : Flonum
(flatanh y) → Flonum
  y : Flonum
```

Return the inverse hyperbolic sine, cosine and tangent of *y*, respectively.

These functions are as robust and accurate as their corresponding inverses.

```
(flfactorial n) → Flonum
  n : Flonum
```

```
(flbinomial n k) → Flonum
  n : Flonum
  k : Flonum
(flpermutations n k) → Flonum
  n : Flonum
  k : Flonum
(flmultinomial n ks) → Flonum
  n : Flonum
  ks : (Listof Flonum)
```

Like `(fl (factorial (fl->exact-integer n)))` and so on, but computed in constant time. Also, these return `+nan.0` instead of raising exceptions.

For factorial-like functions that return sensible values for non-integers, see `gamma` and `beta`.

```
(fllog+ a b) → Flonum
  a : Flonum
  b : Flonum
(fllog-quotient a b) → Flonum
  a : Flonum
  b : Flonum
(fllog-hypot a b) → Flonum
  a : Flonum
  b : Flonum
```

Like `(fllog (+ a b))`, `(fllog (/ a b))` and `(fllog (flhypot a b))` but avoiding over/underflow and accurate when the argument to `fllog` is near 1 (within 1 ulp).

```
(fllog-factorial n) → Flonum
  n : Flonum
(fllog-binomial n k) → Flonum
  n : Flonum
  k : Flonum
(fllog-permutations n k) → Flonum
  n : Flonum
  k : Flonum
(fllog-multinomial n ks) → Flonum
  n : Flonum
  ks : (Listof Flonum)
```

Like `(fllog (flfactorial n))` and so on, but more accurate and without unnecessary overflow.

For log-factorial-like functions that return sensible values for non-integers, see `log-gamma` and `log-beta`.

```
(fllog1p x) → Flonum
  x : Flonum
(flexpm1 x) → Flonum
  x : Flonum
```

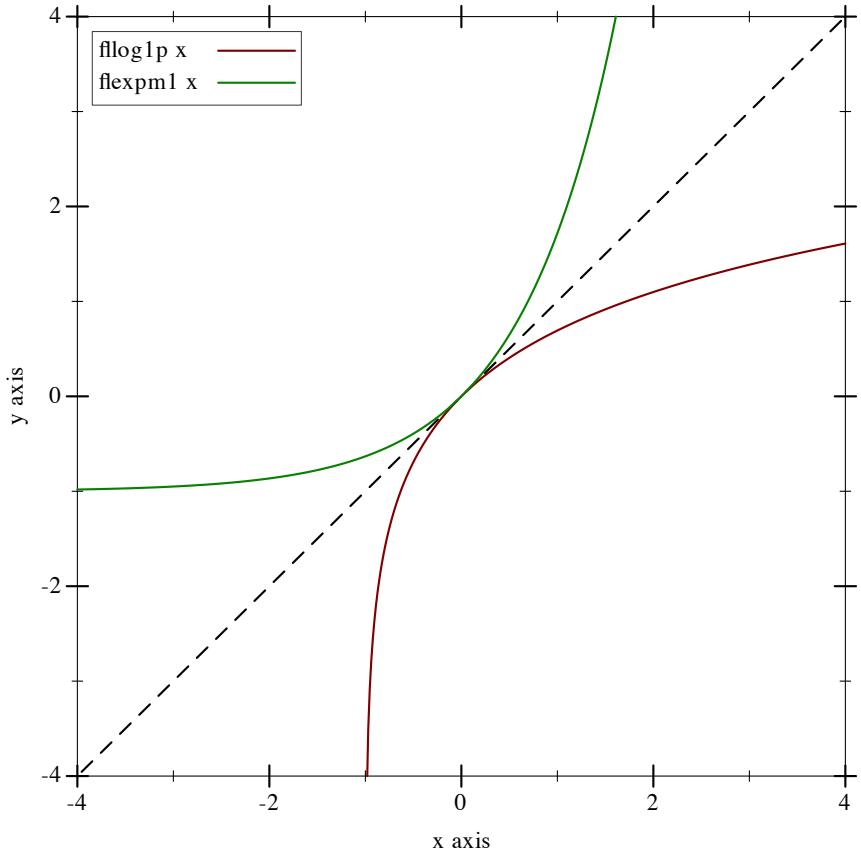
Like `(fllog (+ 1.0 x))` and `(- (flexp x) 1.0)`, but accurate when `x` is small (within 1 ulp).

For example, one difficult input for `(fllog (+ 1.0 x))` and `(- (flexp x) 1.0)` is `x` = `1e-14`, which `fllog1p` and `flexpm1` compute correctly:

```
> (fllog (+ 1.0 1e-14))
9.992007221626358e-15
> (fllog1p 1e-14)
9.99999999999995e-15
> (- (flexp 1e-14) 1.0)
9.992007221626409e-15
> (flexpm1 1e-14)
1.0000000000000049e-14
```

These functions are mutual inverses:

```
> (plot (list
         (function (λ (x) x) #:color 0 #:style 'long-dash)
         (function (compose fllog1p fl) #:label "fllog1p x")
         (function (compose flexpm1 fl) #:label "flexpm1
x" #:color 2))
        #:x-min -4 #:x-max 4 #:y-min -4 #:y-max 4)
```



Notice that both graphs pass through the origin. Thus, inputs close to `0.0`, around which flonums are particularly dense, result in outputs that are also close to `0.0`. Further, both functions are approximately the identity function near `0.0`, so the output density is approximately the same.

Many flonum functions defined in terms of `fllog` and `flexp` become much more accurate when their defining expressions are put in terms of `fllog1p` and `flexpm1`. The functions exported by this module and by `math/special-functions` use them extensively.

One notorious culprit is `(flexpt (- 1.0 x) y)`, when `x` is near `0.0`. Computing it directly too often results in the wrong answer:

```
> (flexpt (- 1.0 1e-20) 1e+20)
1.0
```

We should expect that multiplying a number just less than `1.0` by itself that many times would result in something less than `1.0`. The problem comes from subtracting such a small number from `1.0` in the first place:

```
> (- 1.0 1e-20)
1.0
```

Fortunately, we can compute this correctly by putting the expression in terms of `fllog1p`, which avoids the error-prone subtraction:

```
> (flexp (* 1e+20 (fllog1p (- 1e-20))))
0.36787944117144233
```

But see `flexpt1p`, which is more accurate still.

```
(flexpt1p x y) → Flonum
  x : Flonum
  y : Flonum
```

Like `(flexpt (+ 1.0 x) y)`, but accurate for any x and y.

```
(flexpt+ x1 x2 y) → Flonum
  x1 : Flonum
  x2 : Flonum
  y : Flonum
```

Like `(flexpt (+ x1 x2) y)`, but more accurate.

```
(flexp2 x) → Nonnegative-Flonum
  x : Flonum
```

Equivalent to `(flexpt 2.0 x)`, but faster when x is an integer.

```
(fllog2 x) → Flonum
  x : Flonum
```

Computes the base-2 log of x more accurately than `(/ (fllog x) (fllog 2.0))`. In particular, `(fllog2 x)` is correct for any power of two x.

Examples:

```
> (fllog2 4.5)
2.169925001442312
> (/ (fllog (flexp2 -1066.0)) (fllog 2.0))
-1066.0000000000002
> (fllog2 (flexp2 -1066.0))
-1066.0
```

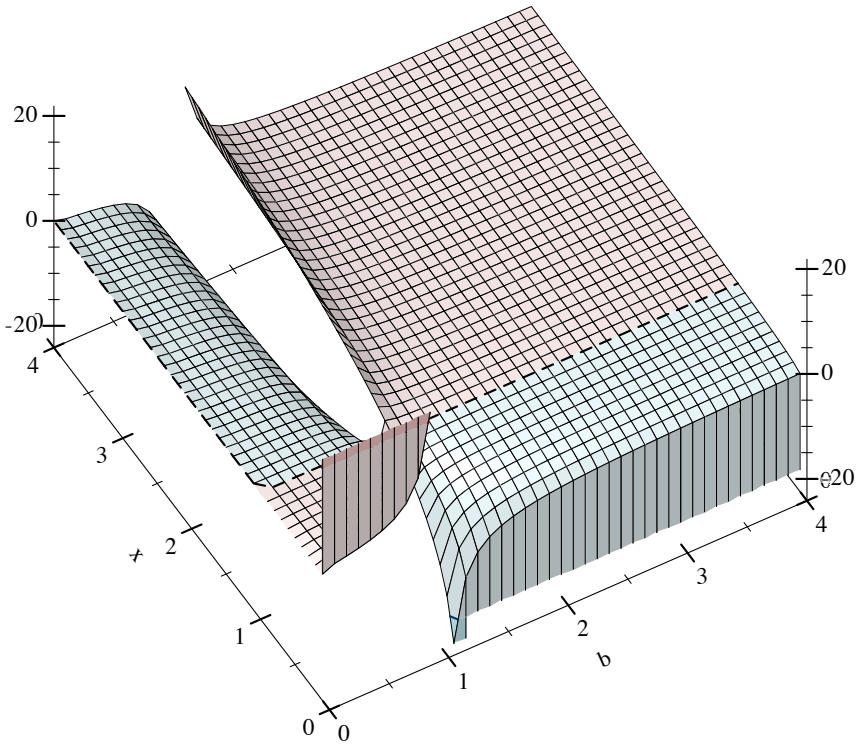
Maximum observed error is 0.5006 ulps, but is almost always no more than 0.5 (i.e. it is almost always *correct*).

```
(fllogb b x) → Flonum
  b : Flonum
  x : Flonum
```

Computes the base-`b` log of `x` more accurately than `(/ (fllog x) (fllog b))`, and handles limit values correctly.

Example:

```
> (plot3d (contour-intervals3d (λ (b x) (fllogb (fl b) (fl x))) 0 4 0 4)
          #:x-label "b" #:y-label "x")
```



Maximum observed error is 2.1 ulps, but is usually less than 0.7 (i.e. near rounding error).

Except possibly at limit values (such as `0.0` and `+inf.0`, and `b = 1.0`) and except when the inner expression underflows or overflows, `fllogb` approximately meets these identities for `b > 0.0`:

- Left inverse: `(fllogb b (flexpt b y)) = y`
- Right inverse: `(flexpt b (fllogb b x)) = x` when `x > 0.0`

Unlike with `flexpt`, there is no standard for `fllogb`'s behavior at limit values. Fortunately, deriving the following rules (applied in order) is not prohibitively difficult.

Case	Condition	Value
`(fllogb b 1.0)`		`0.0`
`(fllogb 1.0 x)`		`+nan.0`
`(fllogb b x)`	`b < 0.0` or `x < 0.0`	`+nan.0`
Double limits		
`(fllogb 0.0 0.0)`		`+inf.0`
`(fllogb 0.0 +inf.0)`		`-inf.0`
`(fllogb +inf.0 0.0)`		`-inf.0`
`(fllogb +inf.0 +inf.0)`		`+inf.0`
Limits with respect to `b`		
`(fllogb 0.0 x)`	`x < 1.0`	`0.0`
`(fllogb 0.0 x)`	`x > 1.0`	`-0.0`
`(fllogb +inf.0 x)`	`x > 1.0`	`0.0`
`(fllogb +inf.0 x)`	`x < 1.0`	`-0.0`
Limits with respect to `x`		
`(fllogb b 0.0)`	`b < 1.0`	`+inf.0`
`(fllogb b 0.0)`	`b > 1.0`	`-inf.0`
`(fllogb b +inf.0)`	`b > 1.0`	`+inf.0`
`(fllogb b +inf.0)`	`b < 1.0`	`-inf.0`

Most of these rules are derived by taking limits of the mathematical base-`b` log function. Except for `(fllogb 1.0 x)`, when doing so gives rise to ambiguities, they are resolved using `flexpt`'s behavior, which follows the IEEE 754 and C99 standards for `pow`.

For example, consider `(fllogb 0.0 0.0)`. Taking an interated limit, we get ∞ if the outer limit is with respect to `x`, or 0 if the outer limit is with respect to `b`. This would normally mean `(fllogb 0.0 0.0) = +nan.0`.

However, choosing `+inf.0` ensures that these additional left-inverse and right-inverse identities hold:

```
(fllogb 0.0 (flexpt 0.0 +inf.0)) = +inf.0
(flexpt 0.0 (fllogb 0.0 0.0)) = 0.0
```

Further, choosing `0.0` does not ensure that any additional identities hold.

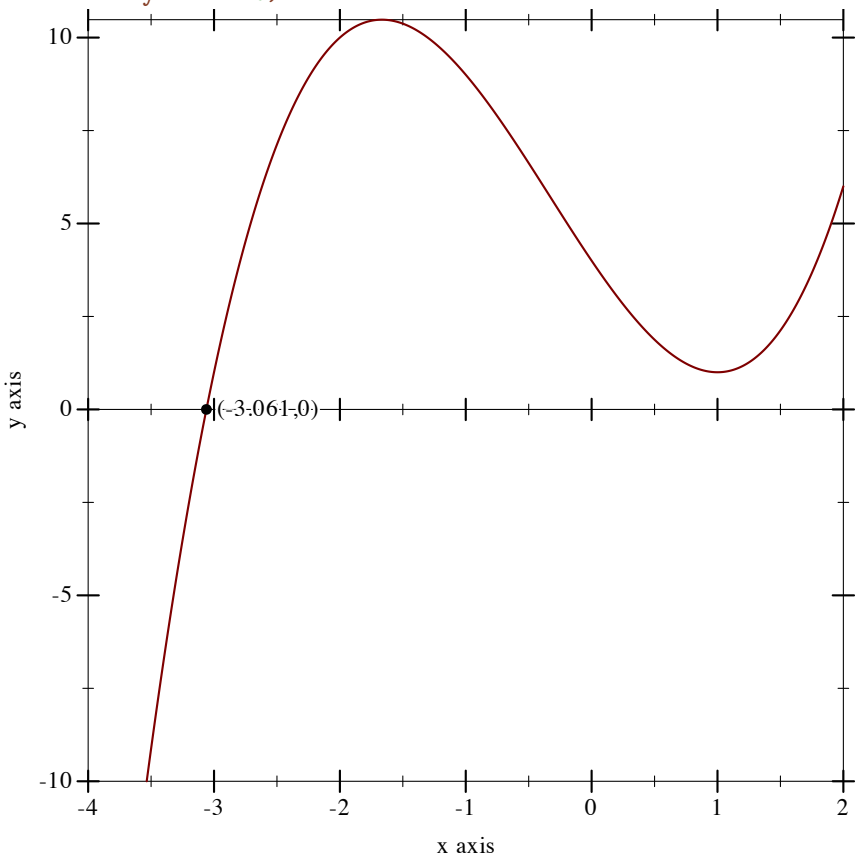
`(flbracketed-root f a b)` → `Flonum`

```
f : (Flonum -> Flonum)
a : Flonum
b : Flonum
```

Uses the Brent-Dekker method to find a floating-point root of `f` (an `x : Flonum` for which `(f x)` is very near a zero crossing) between `a` and `b`. The values `(f a)` and `(f b)` must have opposite signs, but `a` and `b` may be in any order.

Examples:

```
> (define (f x) (+ 1.0 (* (+ x 3.0) (sqr (- x 1.0)))))
> (define x0 (flbracketed-root f -4.0 2.0))
> (plot (list (x-axis)
              (function f -4 2)
              (function-label f x0))
        #:y-min -10)
```




```
> (f (flprev x0))
-7.105427357601002e-15
> (f x0)
```

```
6.661338147750939e-16
> (flbracketed-root f -1.0 2.0)
+nan.0
```

Caveats:

- There is no guarantee that `flbracketed-root` will find any *particular* root. Moreover, future updates to its implementation could make it find different ones.
- There is currently no guarantee that it will find the *closest* `x` to an exact root.
- It currently runs for at most 5000 iterations.

It usually requires far fewer iterations, especially if the initial bounds `a` and `b` are tight.

```
(make-flexpt x) → (Flonum -> Flonum)
  x : Real
```

Equivalent to `(λ (y) (flexpt x y))` when `x` is a flonum, but much more accurate for large `y` when `x` cannot be exactly represented by a flonum.

Suppose we want to compute π^y, where `y` is a flonum. If we use `flexpt` with an *approximation* of the irrational base π, the error is low near zero, but grows with distance from the origin:

```
> (bf-precision 128)
> (define y 150.0)
> (define pi^y (bigfloat->rational (bfexpt pi.bf (bf y))))
> (flulp-error (flexpt pi y) pi^y)
43.12619934359266
```

Using `make-flexpt`, the error is near rounding error everywhere:

```
> (define flexppi (make-flexpt (bigfloat->rational pi.bf)))
> (flulp-error (flexppi y) pi^y)
0.8738006564073412
```

This example is used in the implementations of `zeta` and `psi`.

```
(flsqrt1pm1 x) → Flonum
  x : Flonum
```

Like `(- (flsqrt (+ 1.0 x)) 1.0)`, but accurate when `x` is small.

```
(fllog1pmx x) → Flonum
  x : Flonum
```

Like `(- (fllog1p x) x)`, but accurate when `x` is small.

```
(flexpsqr x) → Flonum
  x : Flonum
```

Like `(flexp (* x x))`, but accurate when `x` is large.

```
(flgauss x) → Flonum
  x : Flonum
```

Like `(flexp (- (* x x)))`, but accurate when `x` is large.

```
(flexp1p x) → Flonum
  x : Flonum
```

Like `(flexp (+ 1.0 x))`, but accurate when `x` is near a power of 2.

```
(flsinpix x) → Flonum
  x : Flonum
(flcospix x) → Flonum
  x : Flonum
(fltanpix x) → Flonum
  x : Flonum
```

Like `(flsin (* pi x))`, `(flcos (* pi x))` and `(fltan (* pi x))`, respectively, but accurate near roots and singularities. When `x = (+ n 0.5)` for some integer `n`, `(fltanpix x) = +nan.0`.

```
(flcscpix x) → Flonum
  x : Flonum
(flsecpix x) → Flonum
  x : Flonum
(flcotpix x) → Flonum
  x : Flonum
```

Like `(/ 1.0 (flsinpix x))`, `(/ 1.0 (flcospix x))` and `(/ 1.0 (fltanpix x))`, respectively, but the first two return `+nan.0` at singularities and `flcotpix` avoids a double reciprocal.

2.2 Log-Space Arithmetic

It is often useful, especially when working with probabilities and probability densities, to represent nonnegative numbers in *log space*, or as the natural logs of their true values. Generally, the reason is that the *smallest* positive flonum is *too large*.

For example, say we want the probability density of the standard normal distribution (the bell curve) at 50 standard deviations from zero:

```
> (require math/distributions)
> (pdf (normal-dist) 50.0)
0.0
```

Mathematically, the density is nonzero everywhere, but the density at 50 is less than `+min.0`. However, its density in log space, or its log-density, is representable:

```
> (pdf (normal-dist) 50.0 #t)
-1250.9189385332047
```

While this example may seem contrived, it is very common, when computing the density of a *vector* of data, for the product of the densities to be too small to represent directly.

In log space, exponentiation becomes multiplication, multiplication becomes addition, and addition becomes tricky. See `lg+` and `lgsum` for solutions.

```
(lg* logx logy) → Flonum
  logx : Flonum
  logy : Flonum
(lg/ logx logy) → Flonum
  logx : Flonum
  logy : Flonum
(lgprod logxs) → Flonum
  logxs : (Listof Flonum)
```

Equivalent to `(fl+ logx logy)`, `(fl- logx logy)` and `(flsum logxs)`, respectively.

```
(lg+ logx logy) → Flonum
  logx : Flonum
  logy : Flonum
(lg- logx logy) → Flonum
  logx : Flonum
  logy : Flonum
```

Like `(fllog (+ (flexp logx) (flexp logy)))` and `(fllog (- (flexp logx) (flexp logy)))`, respectively, but more accurate and less prone to overflow and underflow.

When *logy* > *logx*, lg- returns +nan.0. Both functions correctly treat -inf.0 as log-space 0.0.

To add more than two log-space numbers with the same guarantees, use lgsum.

Examples:

```
> (lg+ (fllog 0.5) (fllog 0.2))
-0.35667494393873234
> (flexp (lg+ (fllog 0.5) (fllog 0.2)))
0.7000000000000001
> (lg- (fllog 0.5) (fllog 0.2))
-1.203972804325936
> (flexp (lg- (fllog 0.5) (fllog 0.2)))
0.30000000000000004
> (lg- (fllog 0.2) (fllog 0.5))
+nan.0
```

Though more accurate than a naive implementation, both functions are prone to catastrophic cancellation in regions where they output a value close to 0.0 (or log-space 1.0). While these outputs have high relative error, their absolute error is very low, and when exponentiated, nearly have just rounding error. Further, catastrophic cancellation is unavoidable when *logx* and *logy* themselves have error, which is by far the common case.

These are, of course, excuses—but for floating-point research generally. There are currently no reasonably fast algorithms for computing lg+ and lg- with low relative error. For now, if you need that kind of accuracy, use math/bigfloat.

```
(lgsum logxs) → Flonum
  logxs : (Listof Flonum)
```

Like folding lg+ over *logxs*, but more accurate. Analogous to flsum.

```
(lg1+ logx) → Flonum
  logx : Flonum
(lg1- logx) → Flonum
  logx : Flonum
```

Equivalent to (lg+ (fllog 1.0) *logx*) and (lg- (fllog 1.0) *logx*), respectively, but faster.

```
(flprobability? x [log?]) → Boolean
  x : Flonum
  log? : Any = #f
```

When *log?* is #f, returns #t when (<= 0.0 *x* 1.0). When *log?* is #t, returns #t when (<= -inf.0 *x* 0.0).

Examples:

```
> (flprobability? -0.1)
#f
> (flprobability? 0.5)
#t
> (flprobability? +nan.0 #t)
#f
```

2.3 Debugging Flonum Functions

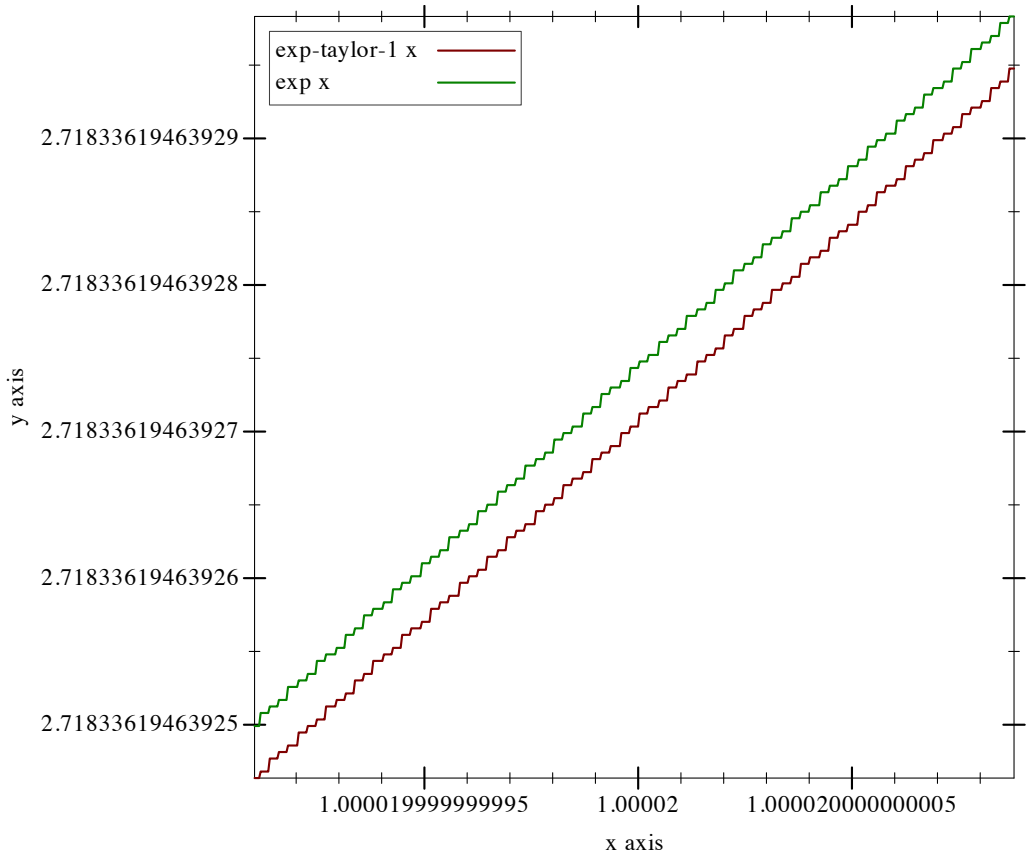
The following functions and constants are useful in authoring and debugging flonum functions that must be accurate on the largest possible domain.

Suppose we approximate `flexp` using its Taylor series centered at `1.0`, truncated after three terms (a second-order polynomial):

```
(define (exp-taylor-1 x)
  (let ([x  (- x 1.0)])
    (* (flexp 1.0) (+ 1.0 x (* 0.5 x x)))))
```

We can use `plot` and `flstep` (documented below) to compare its output to that of `flexp` on very small intervals:

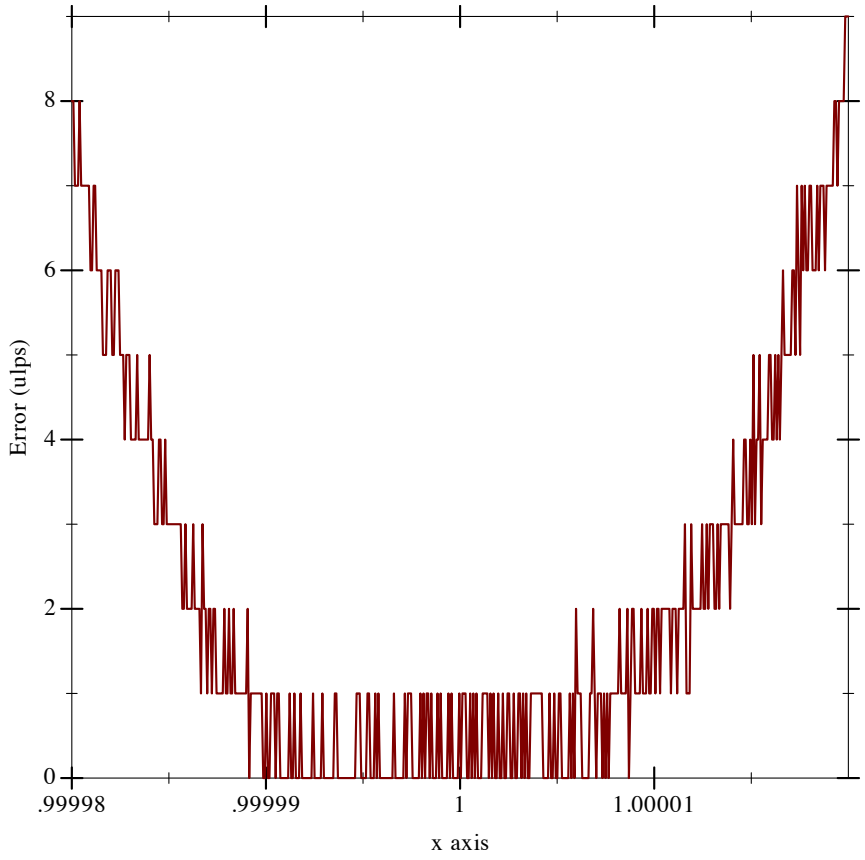
```
> (plot (list (function exp-taylor-1 #:label "exp-taylor-1 x")
              (function exp #:color 2 #:label "exp x"))
        #:x-min (flstep 1.00002 -40)
        #:x-max (flstep 1.00002 40)
        #:width 480)
```



Such plots are especially useful when centered at a boundary between two different approximation methods.

For larger intervals, assuming the approximated function is fairly smooth, we can get a better idea how close the approximation is using `flulp-error`:

```
> (plot (function (λ (x) (flulp-error (exp-taylor-1 x) (exp x))))
        #:x-min 0.99998 #:x-max 1.00002 #:y-label "Error (ulps)")
```



We can infer from this plot that our Taylor series approximation has close to rounding error (no more than an ulp) near `1.0`, but quickly becomes worse farther away.

To get a ground-truth function such as `exp` to test against, compute the outputs as accurately as possible using exact rationals or high-precision bigfloats.

2.3.1 Measuring Floating-Point Error

```
(flulp x) → Flonum
  x : Flonum
```

Returns `x`'s *ulp*, or **u**nit in **l**ast **p**lace: the magnitude of the least significant bit in `x`.

Examples:

```
> (flulp 1.0)
```

```
2.220446049250313e-16
> (flulp 1e-100)
1.2689709186578246e-116
> (flulp 1e+200)
1.6996415770136547e+184
```

```
(flulp-error x r) → Flonum
  x : Flonum
  r : Real
```

Returns the absolute number of ulps difference between `x` and `r`.

For non-rational arguments such as `+nan.0`, `flulp-error` returns `0.0` if `(eqv? x r)`; otherwise it returns `+inf.0`.

A flonum function with maximum error `0.5` ulps exhibits only rounding error; it is *correct*. A flonum function with maximum error no greater than a few ulps is *accurate*. Most moderately complicated flonum functions, when implemented directly, seem to have over a hundred thousand ulps maximum error.

Examples:

```
> (flulp-error 0.5 1/2)
0.0
> (flulp-error 0.14285714285714285 1/7)
0.2857142857142857
> (flulp-error +inf.0 +inf.0)
0.0
> (flulp-error +inf.0 +nan.0)
+inf.0
> (flulp-error 1e-20 0.0)
+inf.0
> (flulp-error (- 1.0 (fl 4999999/5000000)) 1/5000000)
217271.6580864
```

The last example subtracts two nearby flonums, the second of which had already been rounded, resulting in horrendous error. This is an example of *catastrophic cancellation*. Avoid subtracting nearby flonums whenever possible.*

See `relative-error` for a similar way to measure approximation error when the approximation is not necessarily represented by a flonum.

* You can make an exception when the result is to be exponentiated. If `x` has small `absolute-error`, then `(exp x)` has small `relative-error` and small `flulp-error`.

2.3.2 Flonum Constants

```
-max.0 : Flonum
-min.0 : Flonum
+min.0 : Flonum
+max.0 : Flonum
```

The nonzero, rational flonums with maximum and minimum magnitude.

Example:

```
> (list -max.0 -min.0 +min.0 +max.0)
'(-1.7976931348623157e+308 -5e-324 5e-324 1.7976931348623157e+308)
```

```
epsilon.0 : Flonum
```

The smallest flonum that can be added to `1.0` to yield a larger number, or the magnitude of the least significant bit in `1.0`.

Examples:

```
> epsilon.0
2.220446049250313e-16
> (flulp 1.0)
2.220446049250313e-16
```

Epsilon is often used in stopping conditions for iterative or additive approximation methods. For example, the following function uses it to stop Newton's method to compute square roots. (Please do not assume this example is robust.)

```
(define (newton-sqrt x)
  (let loop ([y  (* 0.5 x)])
    (define dy (/ (- x (sqr y)) (* 2.0 y)))
    (if ((abs dy) . <= . (abs (* 0.5 epsilon.0 y)))
        (+ y dy)
        (loop (+ y dy)))))
```

When `(<= (abs dy) (abs (* 0.5 epsilon.0 y)))`, adding `dy` to `y` rarely results in a different flonum. The value `0.5` can be changed to allow looser approximations. This is a good idea when the approximation does not have to be as close as possible (e.g. it is only a starting point for another approximation method), or when the computation of `dy` is known to be inaccurate.

Approximation error is often understood in terms of relative error in epsilons. Number of epsilons relative error roughly corresponds with error in ulps, except when the approximation is subnormal.

2.3.3 Low-Level Flonum Operations

```
(flonum->bit-field x) → Natural
  x : Flonum
```

Returns the bits comprising *x* as an integer. A convenient shortcut for composing `integer-bytes->integer` with `real->floating-point-bytes`.

Examples:

```
> (number->string (flonum->bit-field -inf.0) 16)
"fff0000000000000"
> (number->string (flonum->bit-field +inf.0) 16)
"7ff0000000000000"
> (number->string (flonum->bit-field -0.0) 16)
"8000000000000000"
> (number->string (flonum->bit-field 0.0) 16)
"0"
> (number->string (flonum->bit-field -1.0) 16)
"bff0000000000000"
> (number->string (flonum->bit-field 1.0) 16)
"3ff0000000000000"
> (number->string (flonum->bit-field +nan.0) 16)
"7ff8000000000000"
```

```
(bit-field->flonum i) → Flonum
  i : Integer
```

The inverse of `flonum->bit-field`.

```
(flonum->ordinal x) → Integer
  x : Flonum
```

Returns the signed ordinal index of *x* in a total order over flonums.

When inputs are not `+nan.0`, this function is monotone and symmetric; i.e. if `(fl<= x y)` then `(<= (flonum->ordinal x) (flonum->ordinal y))`, and `(= (flonum->ordinal (- x)) (- (flonum->ordinal x)))`.

Examples:

```
> (flonum->ordinal -inf.0)
-9218868437227405312
> (flonum->ordinal +inf.0)
9218868437227405312
```

```
> (flonum->ordinal -0.0)
0
> (flonum->ordinal 0.0)
0
> (flonum->ordinal -1.0)
-4607182418800017408
> (flonum->ordinal 1.0)
4607182418800017408
> (flonum->ordinal +nan.0)
9221120237041090560
```

These properties mean that `flonum->ordinal` does not distinguish `-0.0` and `0.0`.

```
(ordinal->flonum i) → Flonum
  i : Integer
```

The inverse of `flonum->ordinal`.

```
(flonums-between x y) → Integer
  x : Flonum
  y : Flonum
```

Returns the number of flonums between *x* and *y*, excluding one endpoint. Equivalent to `(- (flonum->ordinal y) (flonum->ordinal x))`.

Examples:

```
> (flonums-between 0.0 1.0)
4607182418800017408
> (flonums-between 1.0 2.0)
4503599627370496
> (flonums-between 2.0 3.0)
2251799813685248
> (flonums-between 1.0 +inf.0)
4611686018427387904
```

```
(flstep x n) → Flonum
  x : Flonum
  n : Integer
```

Returns the flonum *n* flonums away from *x*, according to `flonum->ordinal`. If *x* is `+nan.0`, returns `+nan.0`.

Examples:

```
> (flstep 0.0 1)
5e-324
> (flstep (flstep 0.0 1) -1)
0.0
> (flstep 0.0 -1)
-5e-324
> (flstep +inf.0 1)
+inf.0
> (flstep +inf.0 -1)
1.7976931348623157e+308
> (flstep -inf.0 -1)
-inf.0
> (flstep -inf.0 1)
-1.7976931348623157e+308
> (flstep +nan.0 1000)
+nan.0
```

```
(flnext x) → Flonum
  x : Flonum
(flprev x) → Flonum
  x : Flonum
```

Equivalent to `(flstep x 1)` and `(flstep x -1)`, respectively.

```
(flsubnormal? x) → Boolean
  x : Flonum
```

Returns `#t` when `x` is a subnormal number.

Though flonum operations on subnormal numbers are still often implemented by software exception handling, the situation is improving. Robust flonum functions should handle subnormal inputs correctly, and reduce error in outputs as close to zero ulps as possible.

```
-max-subnormal.0 : Flonum
+max-subnormal.0 : Flonum
```

The maximum positive and negative subnormal flonums. A flonum `x` is subnormal when it is not zero and `(<= (abs x) +max-subnormal.0)`.

Example:

```
> +max-subnormal.0
2.225073858507201e-308
```

2.4 Double-Double Operations

For extra precision, floating-point computations may use two nonoverlapping flonums to represent a single number. Such pairs are often called *double-double* numbers. The exact sum of the pair is the number it represents. (Because they are nonoverlapping, the floating-point sum is equal to the largest.)

For speed, especially with arithmetic operations, there is no data type for double-double numbers. They are always unboxed: given as two arguments, and received as two values. In both cases, the number with higher magnitude is first.

Inputs are never checked to ensure they are sorted and nonoverlapping, but outputs are guaranteed to be sorted and nonoverlapping if inputs are.

```
(fl2 x) → (Values Flonum Flonum)
  x : Real
(fl2 x y) → (Values Flonum Flonum)
  x : Flonum
  y : Flonum
```

Converts a real number or the sum of two flonums into a double-double.

Examples:

```
> (fl 1/7)
0.14285714285714285
> (relative-error (fl 1/7) 1/7)
5.551115123125783e-17
> (define-values (x2 x1) (fl2 1/7))
> (list x2 x1)
'(0.14285714285714285 7.93016446160826e-18)
> (fl (relative-error (+ (inexact->exact x2)
                         (inexact->exact x1))
                      1/7))
3.0814879110195774e-33
```

Notice that the exact sum of `x2` and `x1` in the preceeding example has very low relative error.

If `x` is not rational, `fl2` returns `(values x 0.0)`.

```
(fl2->real x2 x1) → Real
  x2 : Flonum
  x1 : Flonum
```

Returns the exact sum of *x2* and *x1* if *x2* is rational, *x2* otherwise.

Examples:

```
> (define-values (x2 x1) (fl2 1/7))
> (fl2->real x2 x1)
463597933797752466833080029394 65/3245185536584267267831560205762 56
```

```
(fl2? x2 x1) → Boolean
  x2 : Flonum
  x1 : Flonum
```

When *x2* is rational, returns `#t` when `(flabs x2) > (flabs x1)` and *x2* and *x1* are nonoverlapping. When *x2* is not rational, returns `(fl= x1 0.0)`.

Examples:

```
> (define-values (x2 x1) (fl2 1/7))
> (fl2? x2 x1)
#t
> (fl2? 0.14285714285714285 0.07692307692307693)
#f
> (fl2? +inf.0 0.0001)
#f
```

This function is quite slow, so it is used only for testing.

```
(fl+/error x y) → (Values Flonum Flonum)
  x : Flonum
  y : Flonum
(fl-/error x y) → (Values Flonum Flonum)
  x : Flonum
  y : Flonum
(fl*/error x y) → (Values Flonum Flonum)
  x : Flonum
  y : Flonum
(fl//error x y) → (Values Flonum Flonum)
  x : Flonum
  y : Flonum
(flsqr/error x) → (Values Flonum Flonum)
  x : Flonum
(flsqrt/error x) → (Values Flonum Flonum)
  x : Flonum
(flexp/error x) → (Values Flonum Flonum)
  x : Flonum
(flexpm1/error x) → (Values Flonum Flonum)
  x : Flonum
```

Compute the same values as `(fl+ x y)`, `(fl- x y)`, `(fl* x y)`, `(fl/ x y)`, `(fl* x x)`, `(flsqrt x)`, `(flexp x)` and `(flexpm1 x)`, but return the normally rounded-off low-order bits as the second value. The result is an unboxed double-double.

Use these functions to generate double-double numbers directly from the results of floating-point operations.

Examples:

```
> (define x1 (fl 1/7))
> (define x2 (fl 1/13))
> (define z* (bigfloat->real (bfexp (bf* (bf x1) (bf x2)))))
> (relative-error (flexp (fl* x1 x2)) z*)
9.755408946378402e-17
> (let*-values ([(y2 y1)  (fl*/error x1 x2)]
                [(z2 z1)  (fl2exp y2 y1)])
    (fl (relative-error (fl2->real z2 z1) z*)))
4.890426935548821e-33
```

For `flexp/error` and `flexpm1/error`, the largest observed error is 3 ulps. (See `fl2ulp`.) For the rest, the largest observed error is 0.5 ulps.

```
(fl2zero? x2 x1) → Boolean
  x2 : Flonum
  x1 : Flonum
(fl2rational? x2 x1) → Boolean
  x2 : Flonum
  x1 : Flonum
(fl2positive? x2 x1) → Boolean
  x2 : Flonum
  x1 : Flonum
(fl2negative? x2 x1) → Boolean
  x2 : Flonum
  x1 : Flonum
(fl2infinite? x2 x1) → Boolean
  x2 : Flonum
  x1 : Flonum
(fl2nan? x2 x1) → Boolean
  x2 : Flonum
  x1 : Flonum
```

Like `zero?`, `rational?`, `positive?`, `negative?`, `infinite?` and `nan?`, but for double-double flonums.

```
(fl2+ x2 x1 y2 [y1]) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
  y2 : Flonum
  y1 : Flonum = 0.0
```

```
(fl2- x2 x1 y2 [y1]) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
  y2 : Flonum
  y1 : Flonum = 0.0
(fl2* x2 x1 y2 [y1]) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
  y2 : Flonum
  y1 : Flonum = 0.0
(fl2/ x2 x1 y2 [y1]) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
  y2 : Flonum
  y1 : Flonum = 0.0
(fl2abs x2 [x1]) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum = 0.0
(fl2sqr x2 [x1]) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum = 0.0
(fl2sqrt x2 [x1]) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum = 0.0
```

Arithmetic and square root for double-double flonums.

For arithmetic, error is less than 8 ulps. (See `fl2ulp`.) For `fl2sqr` and `fl2sqrt`, error is less than 1 ulp, and `fl2abs` is exact.

```
(fl2= x2 x1 y2 y1) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
  y2 : Flonum
  y1 : Flonum
(fl2> x2 x1 y2 y1) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
  y2 : Flonum
  y1 : Flonum
(fl2< x2 x1 y2 y1) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
  y2 : Flonum
  y1 : Flonum
```

```
(fl2>= x2 x1 y2 y1) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
  y2 : Flonum
  y1 : Flonum
(fl2<= x2 x1 y2 y1) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
  y2 : Flonum
  y1 : Flonum
```

Comparison functions for double-double flonums.

```
(fl2exp x2 x1) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
(fl2log x2 x1) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
(fl2expm1 x2 x1) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
(fl2log1p x2 x1) → (Values Flonum Flonum)
  x2 : Flonum
  x1 : Flonum
```

Like `flexp`, `fllog`, `flexpm1` and `fllog1p`, but for double-double flonums.

For `fl2exp` and `fl2expm1`, error is less than 3 ulps. (See `fl2ulp`.) For `fl2log` and `fl2log1p`, error is less than 2 ulps.

2.4.1 Debugging Double-Double Functions

```
(fl2ulp x2 x1) → Flonum
  x2 : Flonum
  x1 : Flonum
(fl2ulp-error x2 x1 r) → Flonum
  x2 : Flonum
  x1 : Flonum
  r : Real
```

Like `flulp` and `flulp-error`, but for double-double flonums.

The unit in last place of a double-double is that of the higher-order of the pair, shifted 52 bits right.

Examples:

```
> (fl2ulp 1.0 0.0)
4.930380657631324e-32
> (let-values ([(x2 x1)  (fl2 1/7)])
    (fl2ulp-error x2 x1 1/7))
0.07142857142857142
```

```
+max.hi : Flonum
+max.lo : Flonum
-max.hi : Flonum
-max.lo : Flonum
```

The maximum-magnitude, unboxed double-double flonums.

```
+max-subnormal.hi : Flonum
-max-subnormal.hi : Flonum
```

The high-order flonum of the maximum-magnitude, subnormal double-double flonums.

```
> +max-subnormal.0
2.225073858507201e-308
> +max-subnormal.hi
1.0020841800044864e-292
```

Try to avoid computing with double-doubles in the subnormal range in intermediate computations.

2.4.2 Low-Level Double-Double Operations

The following syntactic forms are fast versions of functions like `fl+/error`. They are fast because they make assumptions about the magnitudes of and relationships between their arguments, and do not handle non-rational double-double flonums properly.

```
(fast-mono-fl+/error x y)
(fast-mono-fl-/error x y)
```

Return two values: `(fl+ x y)` or `(fl- x y)`, and its rounding error. Both assume `(flabs x) > (flabs y)`. The values are unspecified when `x` or `y` is not rational.

```
(fast-fl+/error x y)
(fast-fl-/error x y)
```

Like `fast-mono-fl+/error` and `fast-mono-fl-/error`, but do not assume `(flabs x) > (flabs y)`.

```
(fast-fl*/error x y)
(fast-fl//error x y)
(fast-flsqr/error x)
```

Like `fl*/error`, `fl//error` and `flsqr/error`, but faster, and may return garbage when an argument is subnormal or nearly infinite.

```
(flsplit x)
```

Returns nonoverlapping `(values y2 y1)`, each with 26 bits precision, with `(flabs y2) > (flabs y1)`, such that `(fl+ y2 y1) = x`. For `(flabs x) > 1.3393857490036326e+300`, returns `(values +nan.0 +nan.0)`.

Used to implement double-double multiplication.

2.5 Additional Flonum Vector Functions

```
(build-flvector n proc) → FlVector
  n : Integer
  proc : (Index -> Flonum)
```

Creates a length-*n* flonum vector by applying *proc* to the indexes from `0` to `(- n 1)`. Analogous to `build-vector`.

Example:

```
> (build-flvector 10 fl)
(flvector 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0)
```

```
(inline-build-flvector n proc)

  n : Integer
  proc : (Index -> Flonum)
```

Like `build-flvector`, but always inlined. This increases speed at the expense of code size.

```
(flvector-map proc xs xss ...) → FlVector
  proc : (Flonum Flonum ... -> Flonum)
  xs : FlVector
  xss : FlVector
```

Applies *proc* to the corresponding elements of *xs* and *xss*. Analogous to vector-map.

The *proc* is meant to accept the same number of arguments as the number of its following flonum vector arguments. However, a current limitation in Typed Racket requires *proc* to accept *any* number of arguments. To map a single-arity function such as fl+ over the corresponding number of flonum vectors, for now, use inline-flvector-map.

```
(inline-flvector-map proc xs xss ...)

  proc : (Flonum Flonum ... -> Flonum)
  xs : FlVector
  xss : FlVector
```

Like flvector-map, but always inlined.

```
(flvector-copy! dest
                dest-start
                src
               [src-start
                src-end])  → Void
  dest : FlVector
  dest-start : Integer
  src : FlVector
  src-start : Integer = 0
  src-end : Integer = (flvector-length src)
```

Like vector-copy!, but for flonum vectors.

```
(list->flvector vs) → FlVector
  vs : (Listof Real)
(flvector->list xs) → (Listof Flonum)
  xs : FlVector
(vector->flvector vs) → FlVector
  vs : (Vectorof Real)
(flvector->vector xs) → (Vectorof Flonum)
  xs : FlVector
```

Convert between lists and flonum vectors, and between vectors and flonum vectors.

```
(flvector+ xs ys) → FlVector
  xs : FlVector
  ys : FlVector
(flvector* xs ys) → FlVector
  xs : FlVector
  ys : FlVector
```

```
(flvector- xs) → FlVector
  xs : FlVector
(flvector- xs ys) → FlVector
  xs : FlVector
  ys : FlVector
(flvector/ xs) → FlVector
  xs : FlVector
(flvector/ xs ys) → FlVector
  xs : FlVector
  ys : FlVector
(flvector-scale xs y) → FlVector
  xs : FlVector
  y : Flonum
(flvector-abs xs) → FlVector
  xs : FlVector
(flvector-sqr xs) → FlVector
  xs : FlVector
(flvector-sqrt xs) → FlVector
  xs : FlVector
(flvector-min xs ys) → FlVector
  xs : FlVector
  ys : FlVector
(flvector-max xs ys) → FlVector
  xs : FlVector
  ys : FlVector
```

Arithmetic lifted to operate on flonum vectors.

```
(flvector-sum xs) → Flonum
  xs : FlVector
```

Like `flsum`, but operates on flonum vectors. In fact, `flsum` is defined in terms of `flvector-sum`.

```
(flvector-sums xs) → FlVector
  xs : FlVector
```

Computes the partial sums of the elements in *xs* in a way that incurs rounding error only once for each partial sum.

Example:

```
> (flvector-sums
   (flvector 1.0 1e-16 1e-16 1e-16 1e-16 1e+100 -1e+100))
(flvector
```

```
 1.0
 1.0
 1.0000000000000002
 1.0000000000000002
 1.0000000000000004
 1e+100
 1.0000000000000004)
```

Compare the same example computed by direct summation:

```
> (rest
   (reverse
    (foldl (λ (x xs) (cons (+ x (first xs)) xs))
           (list 0.0)
           '(1.0 1e-16 1e-16 1e-16 1e-16 1e+100 -1e+100))))
'(1.0 1.0 1.0 1.0 1.0 1e+100 0.0)
```

3 Special Functions

```
(require math/special-functions)
```
package: math-lib

The term "special function" has no formal definition. However, for the purposes of the math library, a *special function* is one that is not elementary.

The special functions are split into two groups: §3.1 "Real Functions" and §3.2 "Flonum Functions". Functions that accept real arguments are usually defined in terms of their flonum counterparts, but are different in two crucial ways:

- Many return exact values for certain exact arguments.
- When applied to exact arguments outside their domains, they raise an exn:fail:contract instead of returning +nan.0.

Currently, math/special-functions does not export any functions that accept or return complex numbers. Mathematically, some of them could return complex numbers given real numbers, such hurwitz-zeta when given a negative second argument. In these cases, they raise an exn:fail:contract (for an exact argument) or return +nan.0 (for an inexact argument).

Most real functions have more than one type, but they are documented as having only one. The documented type is the most general type, which is used to generate a contract for uses in untyped code. Use :print-type to see all of a function's types.

A function's types state theorems about its behavior in a way that Typed Racket can understand and check. For example, lambert has these types:

```
(case-> (Zero -> Zero)
        (Flonum -> Flonum)
        (Real -> (U Zero Flonum)))
```

Because lambert : Zero -> Zero, Typed Racket proves during typechecking that one of its exact cases is (lambert 0) = 0.

Because the theorem lambert : Flonum -> Flonum is stated as a type and proved by typechecking, Typed Racket's optimizer can transform the expressions around its use into bare-metal floating-point operations. For example, (+ 2.0 (lambert 3.0)) is transformed into (unsafe-fl+ 2.0 (lambert 3.0)).

The most general type Real -> (U Zero Flonum) is used to generate lambert's contract when it is used in untyped code. Except for this discussion, this the only type documented for lambert.

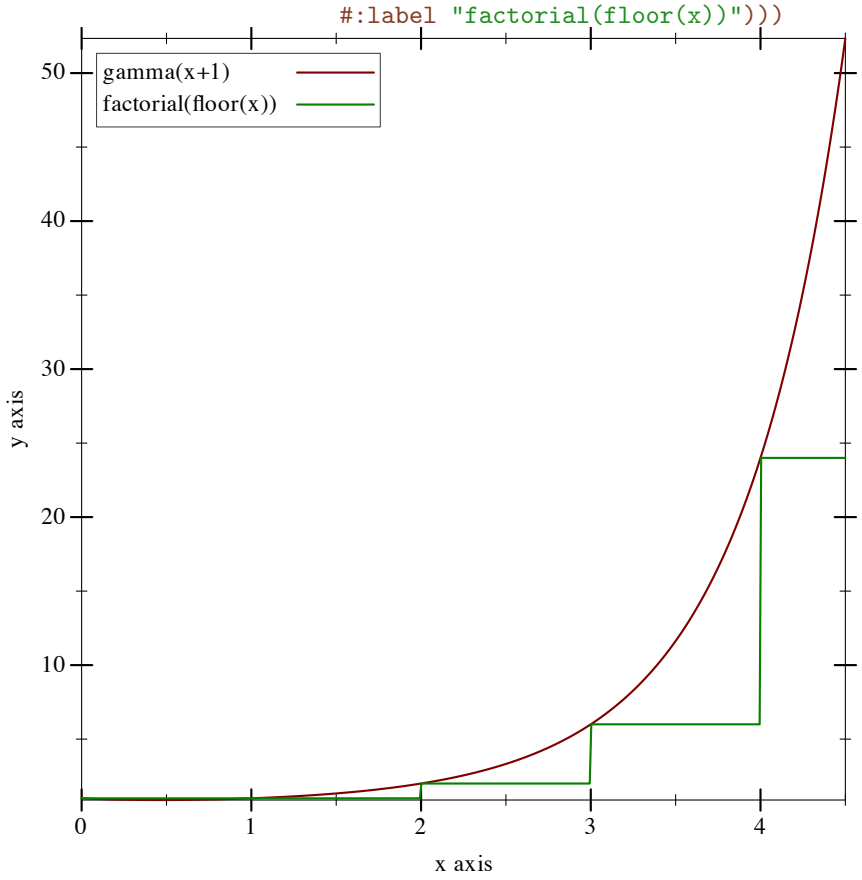
3.1 Real Functions

```
(gamma x) → (U Positive-Integer Flonum)
  x : Real
```

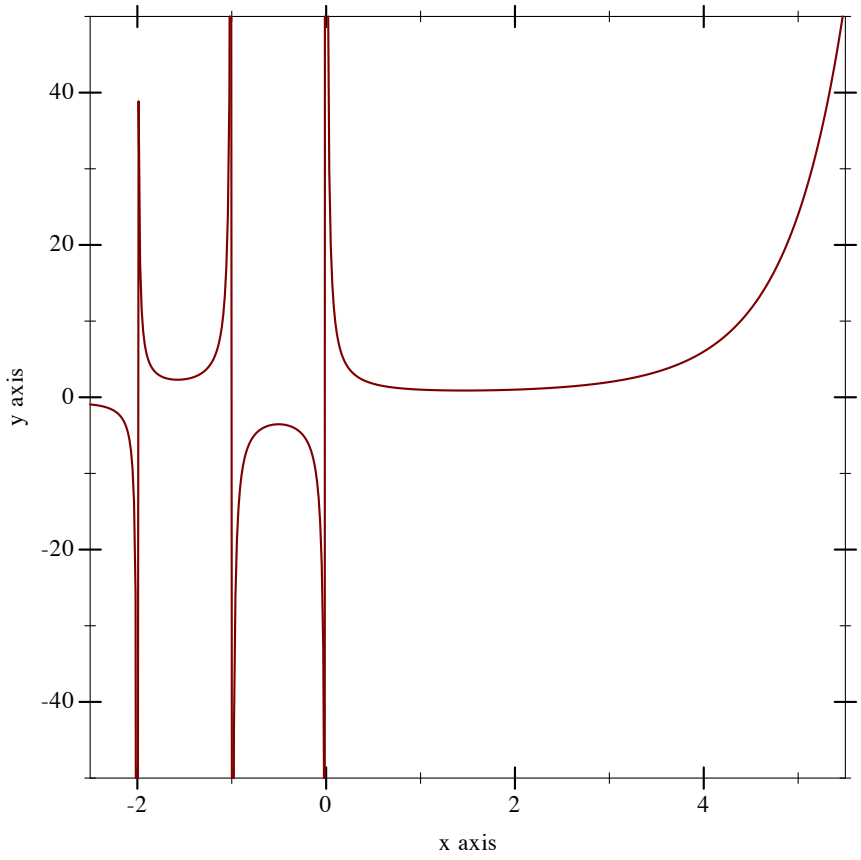
Computes the gamma function, a generalization of the factorial function to the entire real line, except nonpositive integers. When `x` is an exact integer, `(gamma x)` is exact.

Examples:

```
> (plot (list (function (λ (x) (gamma (+ 1 x))) 0 4.5
                        #:label "gamma(x+1)")
              (function (λ (x) (factorial (truncate x))) #:color 2
                        #:label "factorial(floor(x))")))
```




```
> (plot (function gamma -2.5 5.5) #:y-min -50 #:y-max 50)
```




```
> (gamma 5)
24
> (gamma 5.0)
24.0
> (factorial 4)
24
> (gamma -1)
gamma: contract violation
  expected: Real, not Zero or Negative-Integer
  given: -1
> (gamma -1.0)
+nan.0
> (gamma 0.0)
+inf.0
> (gamma -0.0)
-inf.0
> (gamma 172.0)
+inf.0
> (bf (gamma 172))
```

```
(bf "1.241018070217667823424840524103103992618e309")
```

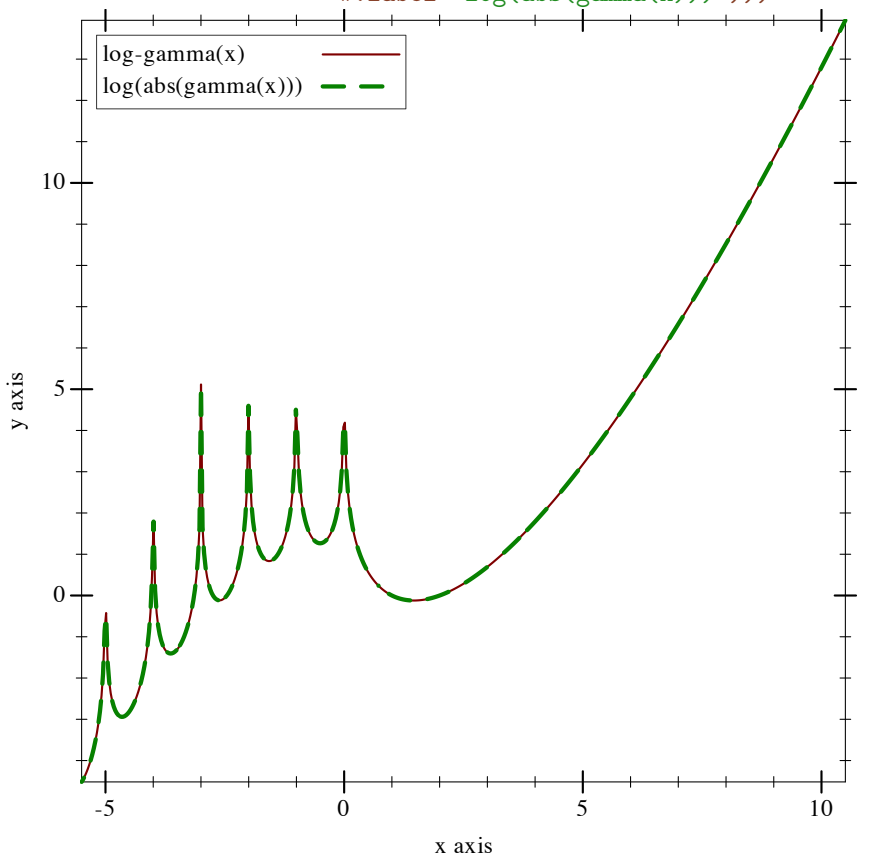
Error is no more than 10 ulps everywhere that has been tested, and is usually no more than 4 ulps.

```
(log-gamma x) → (U Zero Flonum)
  x : Real
```

Like `(log (abs (gamma x)))`, but more accurate and without unnecessary overflow. The only exact cases are `(log-gamma 1) = 0` and `(log-gamma 2) = 0`.

Examples:

```
> (plot (list (function log-gamma -5.5 10.5 #:label "log-
gamma(x)")
              (function (λ (x) (log (abs (gamma x))))
                        #:color 2 #:style 'long-dash #:width 2
                        #:label "log(abs(gamma(x)))")))
```



```
> (log-gamma 5)
3.1780538303479458
> (log (abs (gamma 5)))
3.1780538303479458
> (log-gamma -1)
log-gamma: contract violation
  expected: Real, not Zero or Negative-Integer
  given: -1
> (log-gamma -1.0)
+inf.0
> (log-gamma 0.0)
+inf.0
> (log (abs (gamma 172.0)))
+inf.0
> (log-gamma 172.0)
711.71472580229
```

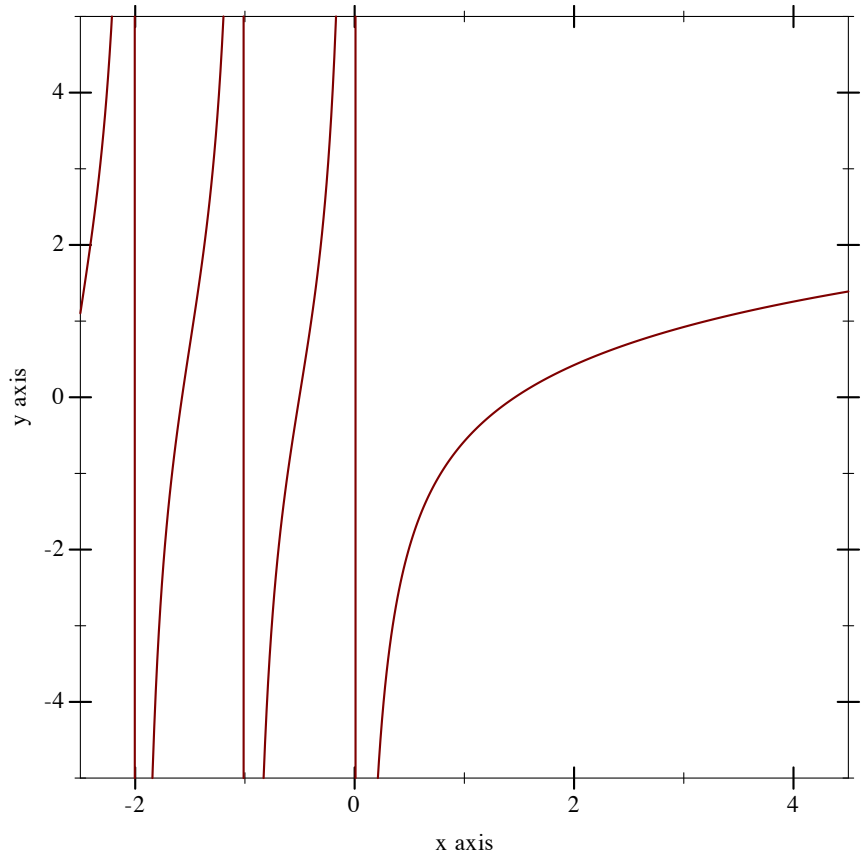
Error is no more than 11 ulps everywhere that has been tested, and is usually no more than 2 ulps. Error reaches its maximum near negative roots.

```
(psi0 x) → Flonum
  x : Real
```

Computes the digamma function, the logarithmic derivative of the gamma function.

Examples:

```
> (plot (function psi0 -2.5 4.5) #:y-min -5 #:y-max 5)
```




```
> (psi0 0)
psi0: contract violation
  expected: Real, not Zero or Negative-Integer
  given: 0
> (psi0 1)
-0.5772156649015329
> (- gamma.0)
-0.5772156649015329
```

Except near negative roots, maximum observed error is 2 ulps, but is usually no more than 1.

Near negative roots, which occur singly between each pair of negative integers, `psi0` exhibits catastrophic cancellation from using the reflection formula, meaning that relative error is effectively unbounded. However, maximum observed `absolute-error` is `(* 5 epsilon.0)`. This is the best we can do for now, because there are currently no reasonably fast algorithms for computing `psi0` near negative roots with low relative error.

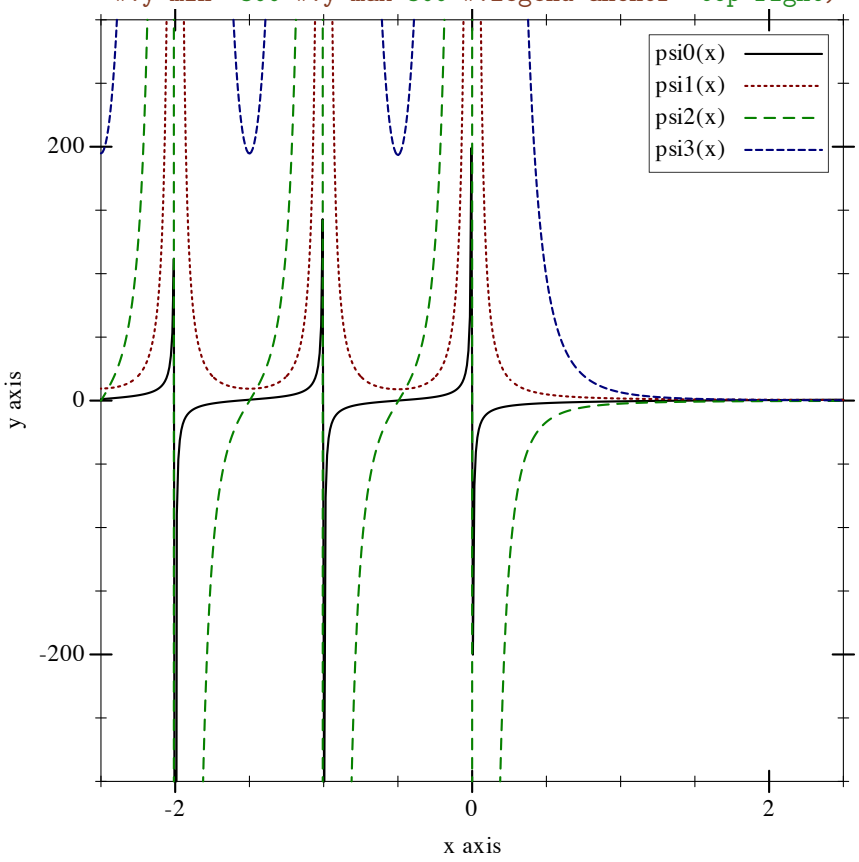
If you need low relative error near negative roots, use `bfpsi0`.

```
(psi m x) → Flonum
  m : Integer
  x : Real
```

Computes a polygamma function, or the mth logarithmic derivative of the gamma function. The order m must be a natural number, and x may not be zero or a negative integer. Note that `(psi 0 x)` = `(psi0 x)`.

Examples:

```
> (plot (for/list ([m  (in-range 4)])
          (function (λ (x) (psi m x)) -2.5 2.5
                    #:color m #:style m #:label (format "psi~a(x)" m)))
        #:y-min -300 #:y-max 300 #:legend-anchor 'top-right)
```




```
> (psi -1 2.3)
psi: contract violation
  expected: Natural
  given: -1
```

```
  argument position: 1st
  other arguments...:
   2.3
> (psi 0 -1.1)
10.154163959143848
> (psi0 -1.1)
10.154163959143848
```

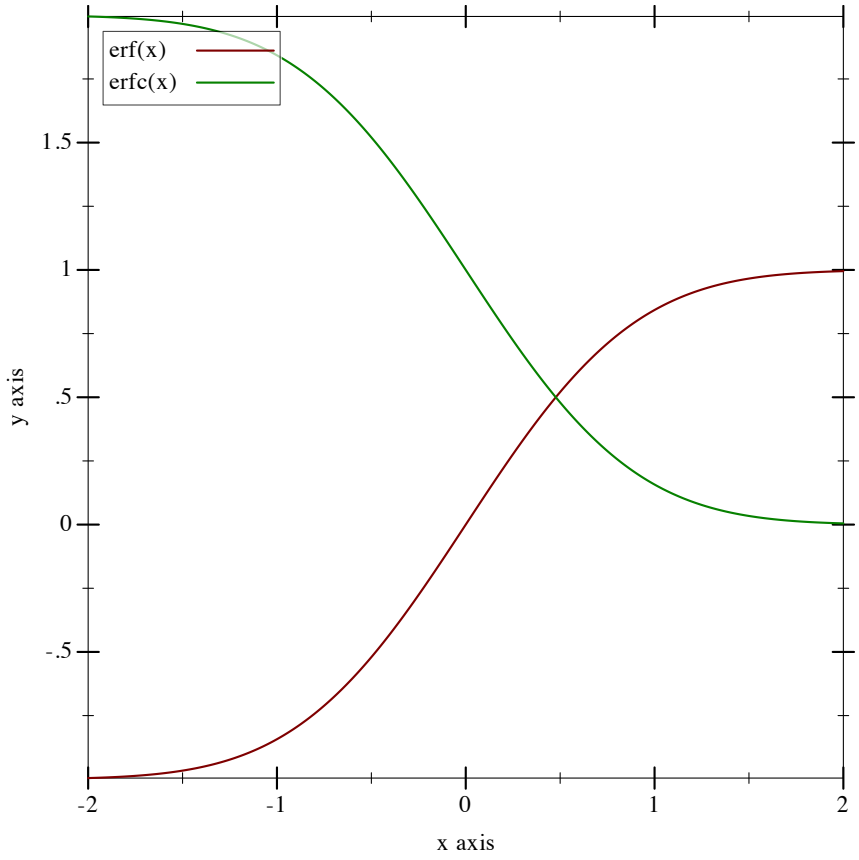
From spot checks with `m > 0`, error appears to be as with `psi0`: very low except near negative roots. Near negative roots, relative error is apparently unbounded, but absolute error is low.

```
(erf x) → Real
  x : Real
(erfc x) → Real
  x : Real
```

Compute the error function and complementary error function, respectively. The only exact cases are `(erf 0) = 0` and `(erfc 0) = 1`.

Examples:

```
> (plot (list (function erf -2 2 #:label "erf(x)")
              (function erfc #:color 2 #:label "erfc(x)")))
```




```
> (erf 0)
0
> (erf 1)
0.8427007929497148
> (- 1 (erfc 1))
0.8427007929497148
> (erf -1)
-0.8427007929497149
> (- (erfc 1) 1)
-0.8427007929497148
```

Mathematically, erfc(x) = 1 - erf(x), but having separate implementations can help maintain accuracy. To compute an expression containing erf, use `erf` for `x` near `0.0`. For positive `x` away from `0.0`, manipulate `(- 1.0 (erfc x))` and its surrounding expressions to avoid the subtraction:

```
> (define x 5.2)
> (bf-precision 128)
> (define log-erf-x (bigfloat->rational (bflog (bferf (bf x)))))
```

```
> (flulp-error (log (erf x)) log-erf-x)
873944876280.6095
> (flulp-error (log (- 1.0 (erfc x))) log-erf-x)
873944876280.6095
> (flulp-error (fllog1p (- (erfc x))) log-erf-x)
1.609486456125461
```

For negative `x` away from `0.0`, do the same with `(- (erfc (- x)) 1.0)`.

For `erf`, error is no greater than 2 ulps everywhere that has been tested, and is almost always no greater than 1. For `erfc`, observed error is no greater than 4 ulps, and is usually no greater than 2.

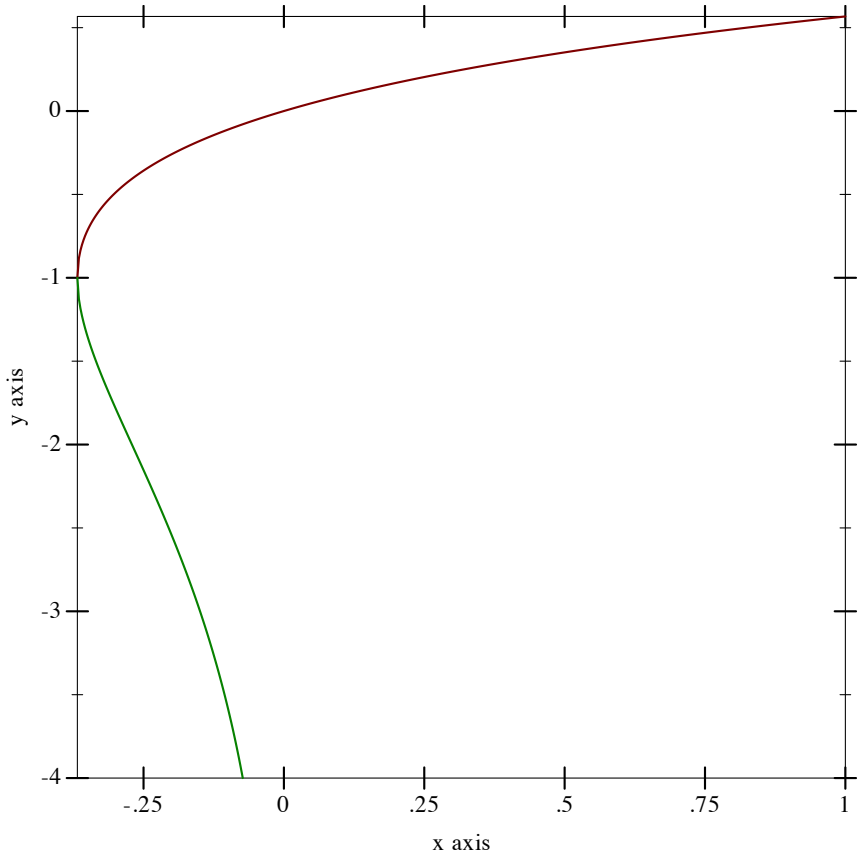
```
(lambert x) → (U Zero Flonum)
  x : Real
(lambert- x) → Flonum
  x : Real
```

Compute the Lambert W function, or the inverse of `x = (* y (exp y))`.

This function has two real branches. The `lambert` variant computes the upper branch, and is defined for `x >= (- (exp -1))`. The `lambert-` variant computes the lower branch, and is defined for *negative* `x >= (- (exp -1))`. The only exact case is `(lambert 0) = 0`.

Examples:

```
> (plot (list (function lambert (- (exp -1)) 1)
              (function lambert- (- (exp -1)) -min.0 #:color 2))
        #:y-min -4)
```




```
> (lambert 0)
0
> (lambert (- (exp -1)))
-1.0
> (lambert -1/2)
lambert: contract violation
  expected: Real >= (- (exp -1))
  given: -1/2
> (lambert- 0)
lambert-: contract violation
  expected: Negative-Real >= (- (exp -1))
  given: 0
> (define y0 (lambert -0.1))
> (define y1 (lambert- -0.1))
> y0
-0.11183255915896297
> y1
-3.577152063957297
> (* y0 (exp y0))
```

```
-0.1
> (* y1 (exp y1))
-0.10000000000000002
```

The Lambert W function often appears in solutions to equations that contain $n \log(n)$, such as those that describe the running time of divide-and-conquer algorithms.

For example, suppose we have a sort that takes `t = (* c n (log n))` time, and we measure the time it takes to sort an `n = 10000`-element list at `t = 0.245` ms. Solving for `c`, we get

```
> (define n 10000)
> (define t 0.245)
> (define c (/ t (* n (log n))))
> c
2.6600537016574172e-6
```

Now we would like to know how many elements we can sort in 100ms. We solve for `n` and use the solution to define a function `time->sort-size`:

```
> (define (time->sort-size t)
    (exact-floor (exp (lambert (/ t c)))))
> (time->sort-size 100)
2548516
```

Testing the solution, we get

```
> (define lst2 (build-list 2548516 values))
> (time (sort lst2 <))
cpu time: 80 real time: 93 gc time: 0
```

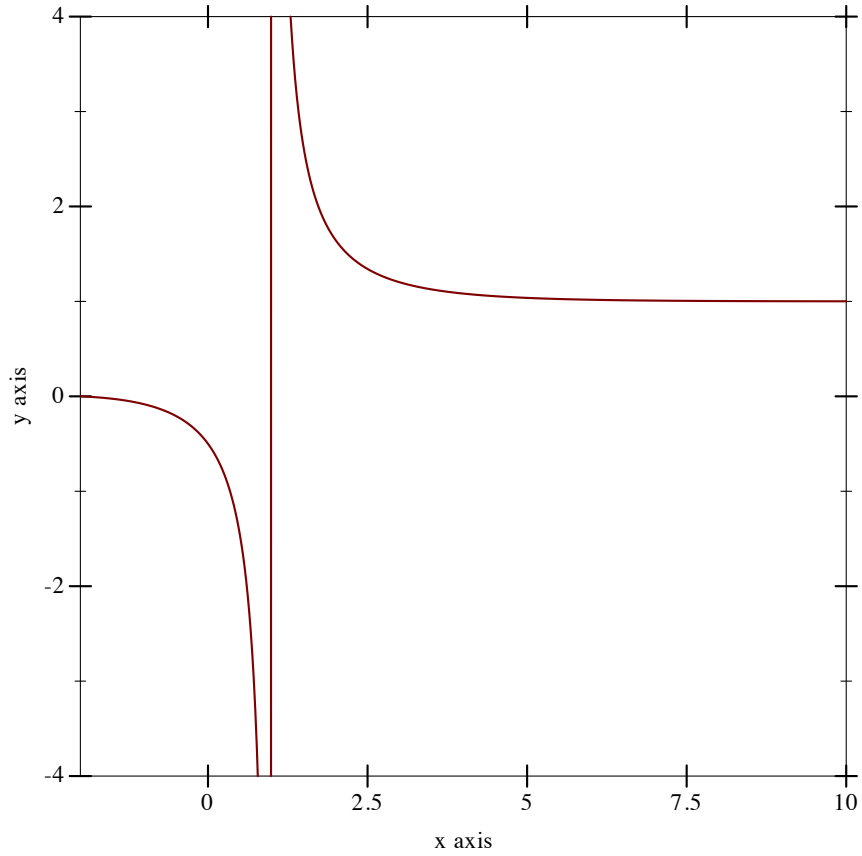
For both branches, error is no more than 2 ulps everywhere tested.

```
(zeta x) → Real
  x : Real
```

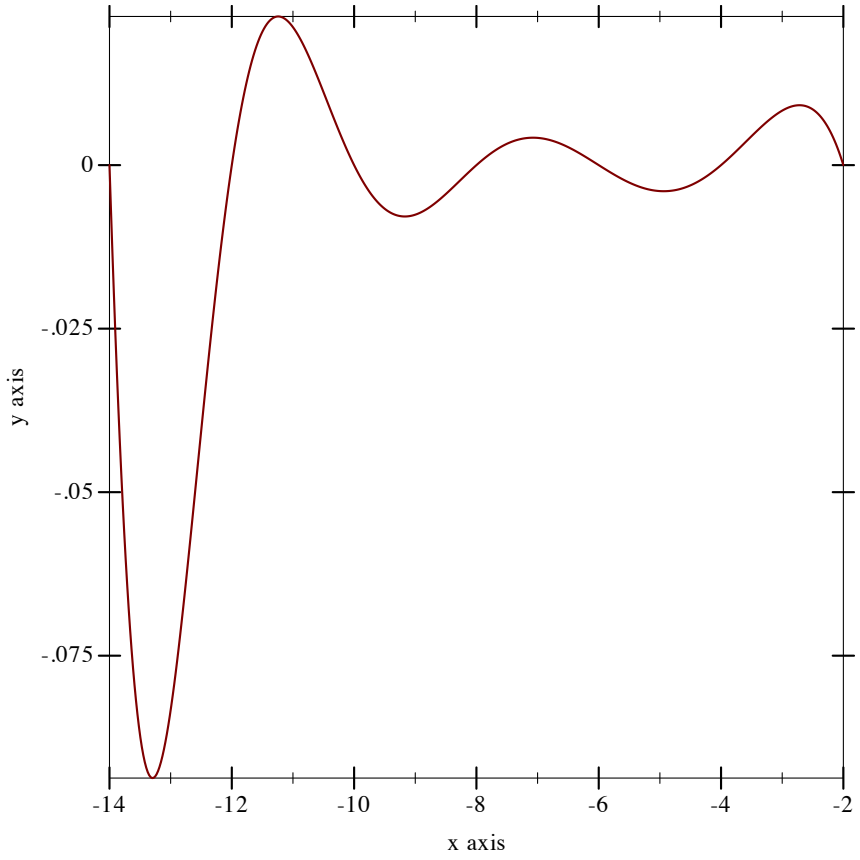
Computes the Riemann zeta function. If `x` is a nonpositive exact integer, `(zeta x)` is exact.

Examples:

```
> (plot (function zeta -2 10) #:y-min -4 #:y-max 4)
```




```
> (plot (function zeta -14 -2))
```




```
> (zeta 0)
-1/2
> (zeta 1)
zeta: contract violation
  expected: Real, not One
  given: 1
> (zeta 1.0)
-inf.0
> (zeta -1)
-1/12
> (define num 1000000)
> (define num-coprime
    (for/sum ([_  (in-range num)])
      (if (coprime? (random-bits 16) (random-bits 16)) 1 0)))
> (fl (/ num-coprime num))
0.607901
> (/ 1 (zeta 2))
0.6079271018540264
```

When `s` is an odd, negative exact integer, `(zeta s)` computes `(bernoulli (- 1 s))`, which can be rather slow.

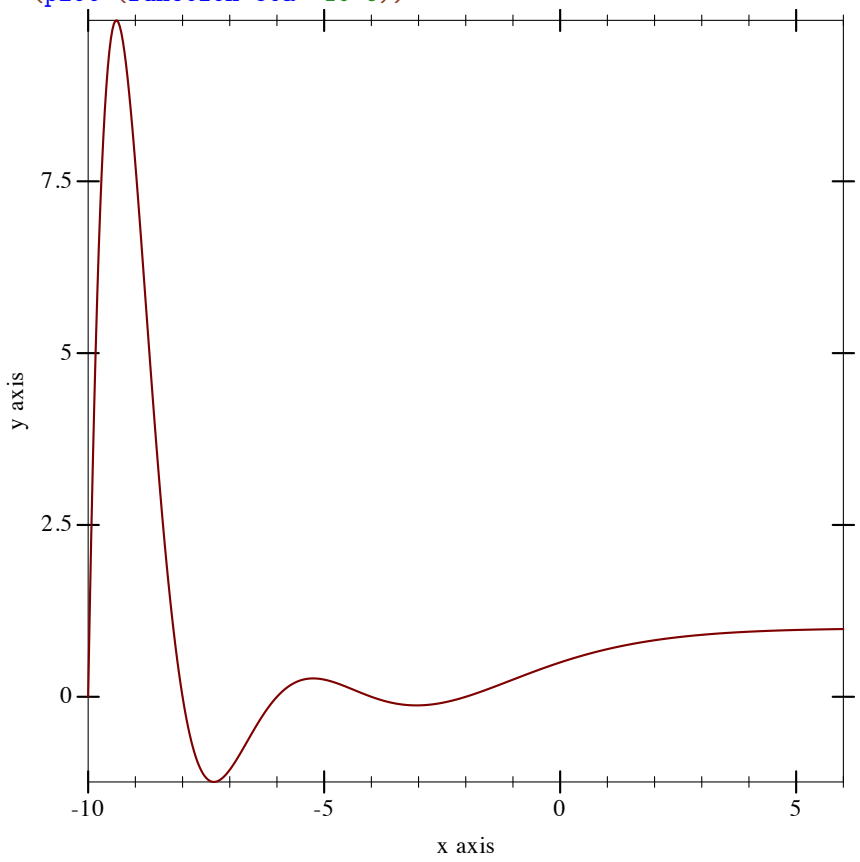
Maximum observed error is 6 ulps, but is usually 3 or less.

```
(eta x) → Real
  x : Real
```

Computes the Dirichlet eta function. If `x` is a nonpositive exact integer, `(eta x)` is exact.

Examples:

```
> (plot (function eta -10 6))
```




```
> (eta 0)
1/2
> (eta -1)
1/4
> (eta 1)
0.6931471805599453
```

```
> (log 2)
0.6931471805599453
```

When `s` is an odd, negative exact integer, `(eta s)` computes `(bernoulli (- 1 s))`, which can be rather slow.

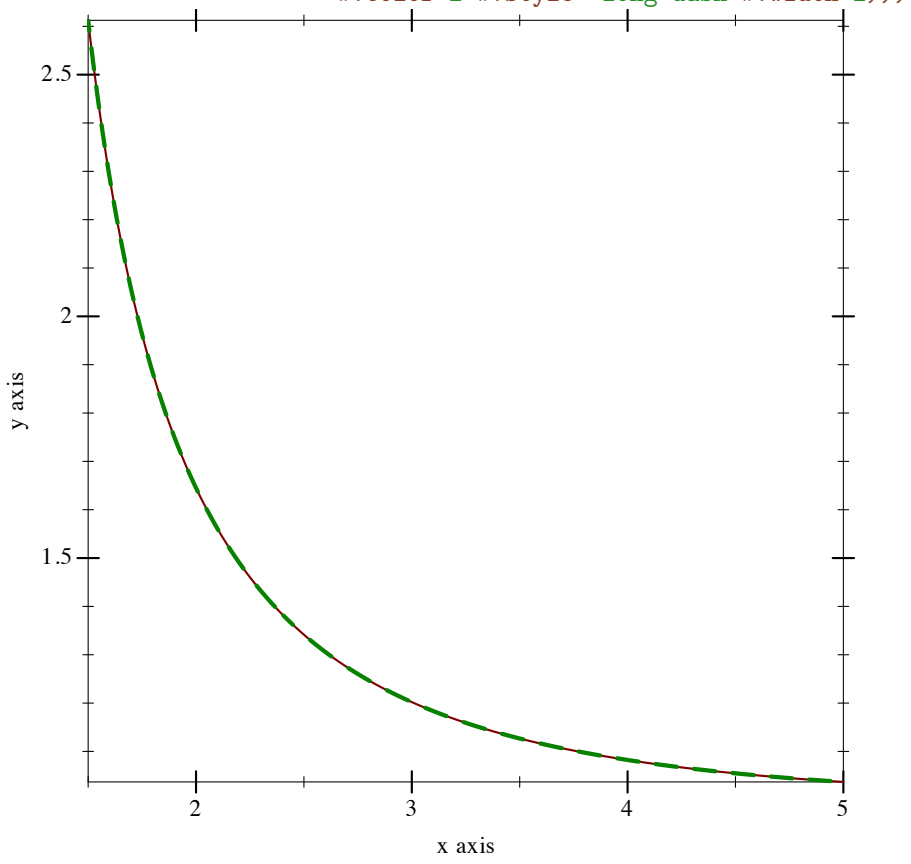
Maximum observed error is 11 ulps, but is usually 4 or less.

```
(hurwitz-zeta s q) → Real
  s : Real
  q : Real
```

Computes the Hurwitz zeta function for `s > 1` and `q > 0`. When `s = 1.0` or `q = 0.0`, `(hurwitz-zeta s q) = +inf.0`.

Examples:

```
> (plot (list (function zeta 1.5 5)
              (function (λ (s) (hurwitz-zeta s 1))
                        #:color 2 #:style 'long-dash #:width 2)))
```



```
> (hurwitz-zeta 1 1)
hurwitz-zeta: contract violation
  expected: Real > 1
  given: 1
  argument position: 1st
  other arguments...:
   1
> (hurwitz-zeta 1.0 1.0)
+inf.0
> (hurwitz-zeta 2 1/4)
17.197329154507113
> (+ (sqr pi) (* 8 catalan.0))
17.19732915450711
```

While `hurwitz-zeta` currently raises an exception for $s < 1$, it may in the future return real values.

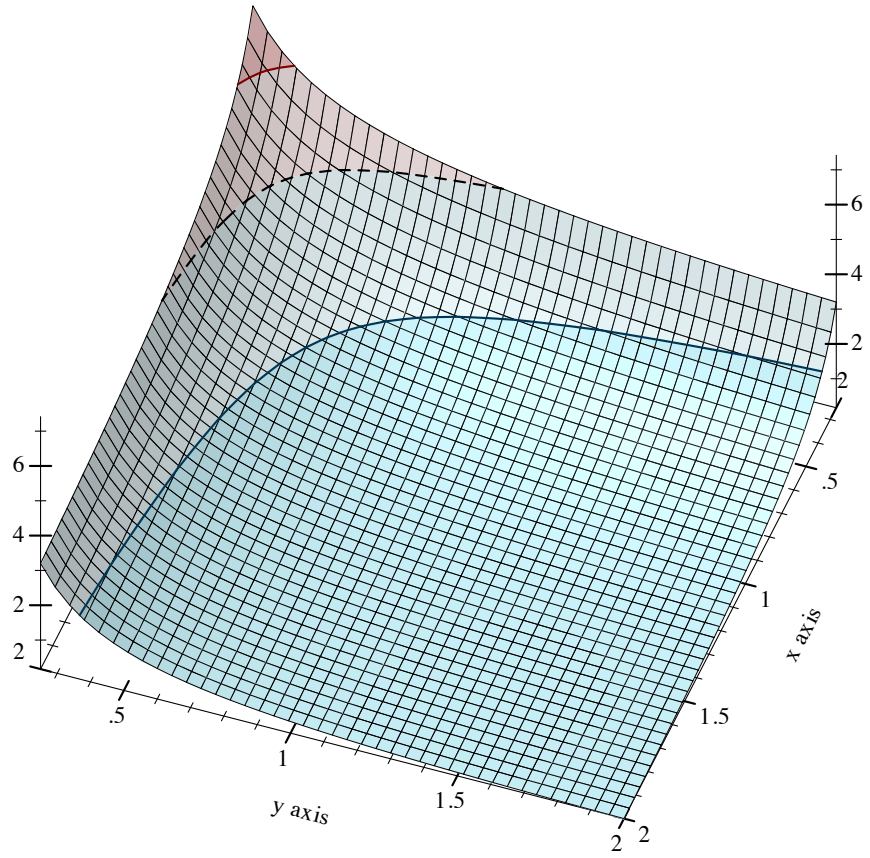
Maximum observed error is 6 ulps, but is usually 2 or less.

```
(beta x y) → (U Exact-Rational Flonum)
  x : Real
  y : Real
```

Computes the beta function for positive real x and y. Like `(/ (* (gamma x) (gamma y)) (gamma (+ x y)))`, but more accurate.

Examples:

```
> (plot3d (contour-intervals3d beta 0.25 2 0.25 2) #:angle 250)
```




```
> (beta 0 0)
beta: contract violation
  expected: positive Real
  given: 0
  argument position: 1st
  other arguments...:
   0
> (beta 1 5)
1/5
> (beta 1.0 5.0)
0.2
```

```
(log-beta x y) → (U Zero Flonum)
  x : Real
  y : Real
```

Like `(log (beta x y))`, but more accurate and without unnecessary overflow. The only exact case is `(log-beta 1 1) = 0`.

```
(gamma-inc k x [upper? regularized?]) → Flonum
  k : Real
  x : Real
  upper? : Any = #f
  regularized? : Any = #f
```

Computes the incomplete gamma integral for `k > 0` and `x >= 0`. When `upper? = #f`, it integrates from zero to `x`; otherwise it integrates from `x` to infinity.

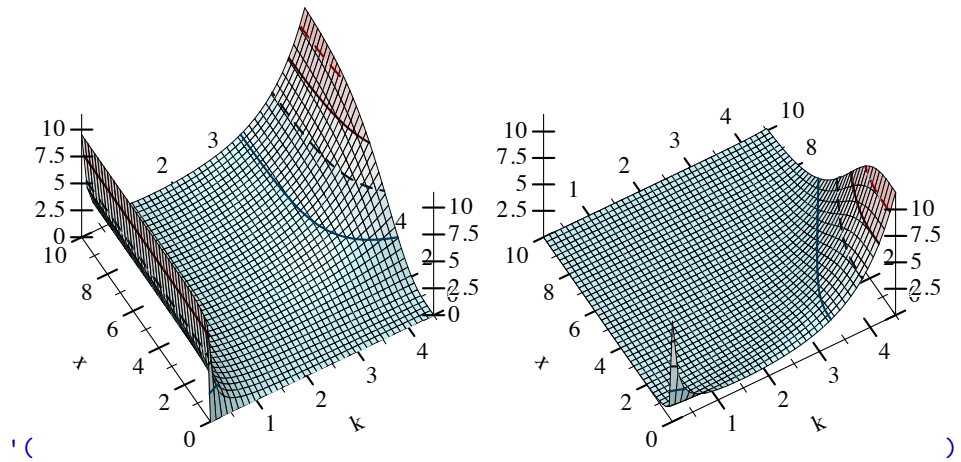
If you are doing statistical work, you should probably use `gamma-dist` instead, which is defined in terms of `gamma-inc` and is more flexible (e.g. it allows negative `x`).

The following identities should hold:

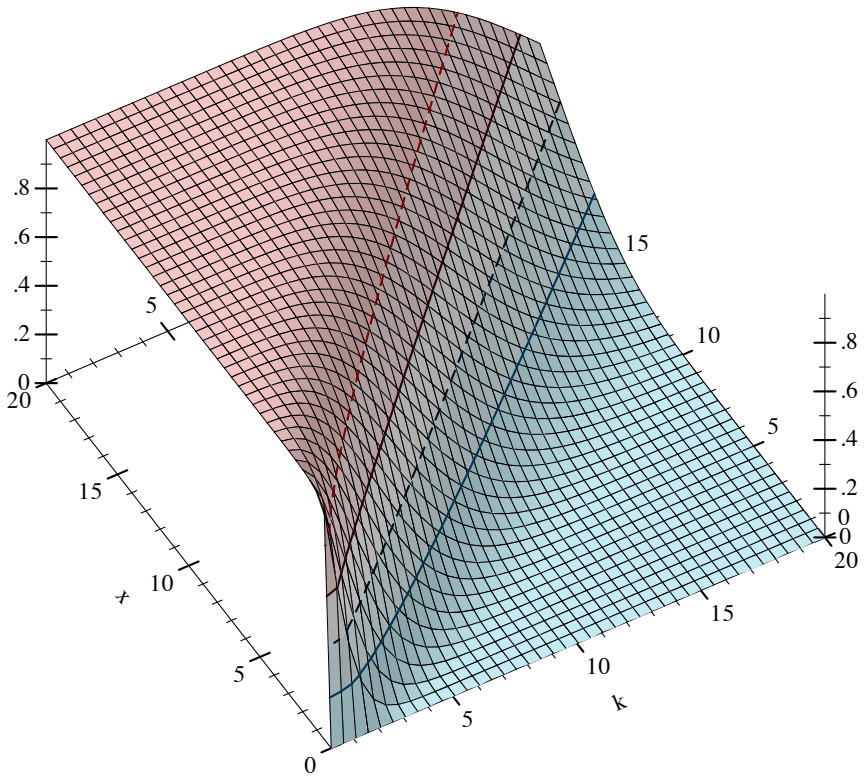
- `(gamma-inc k 0) = 0`
- `(gamma-inc k +inf.0) = (gamma k)`
- `(+ (gamma-inc k x #f) (gamma-inc k x #t)) = (gamma k)` (approximately)
- `(gamma-inc k x upper? #t) = (/ (gamma-inc k x upper? #f) (gamma k))` (approximately)
- `(gamma-inc k +inf.0 #t #t) = 1.0`
- `(+ (gamma-inc k x #f #t) (gamma-inc k x #t #t)) = 1.0` (approximately)

Examples:

```
> (list
   (plot3d (contour-intervals3d gamma-inc 0.1 4.5 0 10)
           #:x-label "k" #:y-label "x" #:width 210 #:height 210)
   (plot3d (contour-intervals3d
            (λ (k x) (gamma-inc k x #t)) 0.1 4.5 0 10)
           #:x-label "k" #:y-label "x" #:width 210 #:height 210))
```



```
'(                                                                    )
> (plot3d (contour-intervals3d
           (λ (k x) (gamma-inc k x #f #t)) 0.1 20 0 20)
          #:x-label "k" #:y-label "x")
```



```
> (gamma 4.0)
```

```
6.0
> (+ (gamma-inc 4.0 0.5 #f) (gamma-inc 4.0 0.5 #t))
6.0
> (gamma-inc 4.0 +inf.0)
6.0
> (/ (gamma-inc 200.0 50.0 #f) (gamma 200.0))
+nan.0
> (gamma-inc 200.0 50.0 #f #t)
2.0247590148473565e-57
> (gamma-inc 0 5.0)
gamma-inc: contract violation
  expected: Positive-Real
  given: 0
  argument position: 1st
  other arguments...:
   5.0
> (gamma-inc 0.0 5.0)
+inf.0
```

```
(log-gamma-inc k x [upper? regularized?]) → Flonum
  k : Real
  x : Real
  upper? : Any = #f
  regularized? : Any = #f
```

Like `(log (gamma-inc k x upper? regularized?))`, but more accurate and without unnecessary overflow.

```
(beta-inc a b x [upper? regularized?]) → Flonum
  a : Real
  b : Real
  x : Real
  upper? : Any = #f
  regularized? : Any = #f
```

Computes the incomplete beta integral for `a > 0`, `b > 0` and `0 <= x <= 1`. When *upper?* `= #f`, it integrates from zero to *x*; otherwise, it integrates from *x* to one.

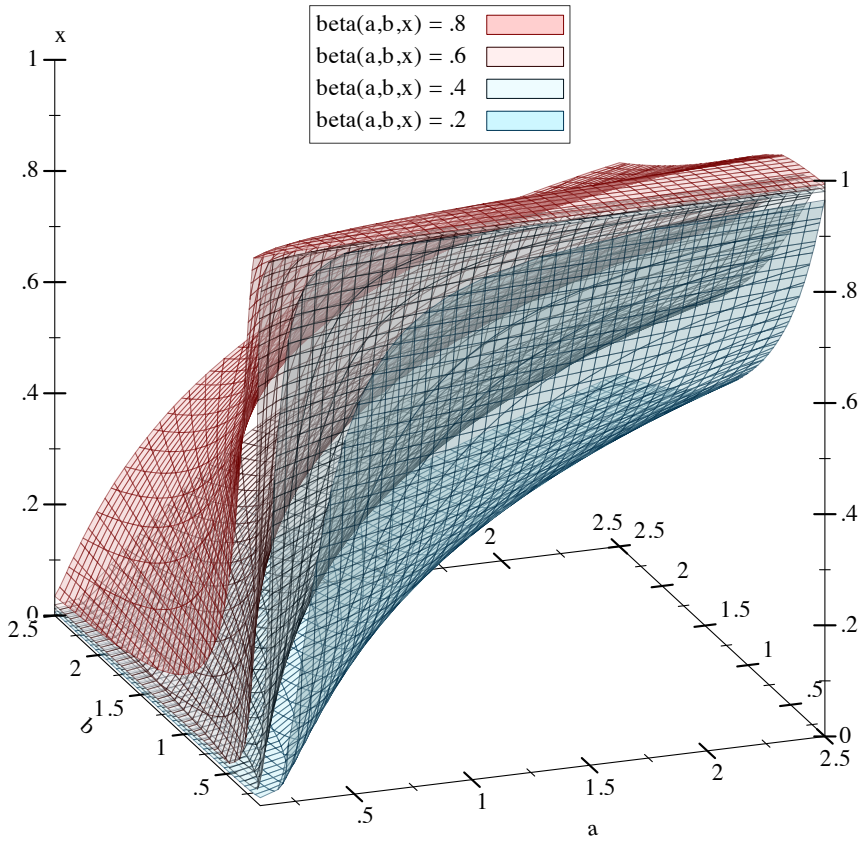
If you are doing statistical work, you should probably use `beta-dist` instead, which is defined in terms of `beta-inc` and is more flexible (e.g. it allows negative *x*).

Similar identities should hold as with `gamma-inc`.

Example:

```
> (plot3d (isosurfaces3d (λ (a b x) (beta-inc a b x #f #t))
```

```
                    0.1 2.5 0.1 2.5 0 1 #:label "beta(a,b,x)")
#:x-label "a" #:y-label "b" #:z-label "x"
#:angle 20 #:altitude 20 #:legend-anchor 'top)
```




```
(log-beta-inc a b x [upper? regularized?]) → Flonum
  a : Real
  b : Real
  x : Real
  upper? : Any = #f
  regularized? : Any = #f
```

Like `(log (beta-inc a b x upper? regularized?))`, but more accurate and without unnecessary overflow.

While most areas of this function have error less than `5e-15`, when `a` and `b` have very dissimilar magnitudes (e.g. `1e-16` and `1e+16`), it exhibits catastrophic cancellation. We are working on it.

```
(Fresnel-S x) → Real
  x : Real
(Fresnel-C x) → Real
  x : Real
(Fresnel-RS x) → Real
  x : Real
(Fresnel-RC x) → Real
  x : Real
```

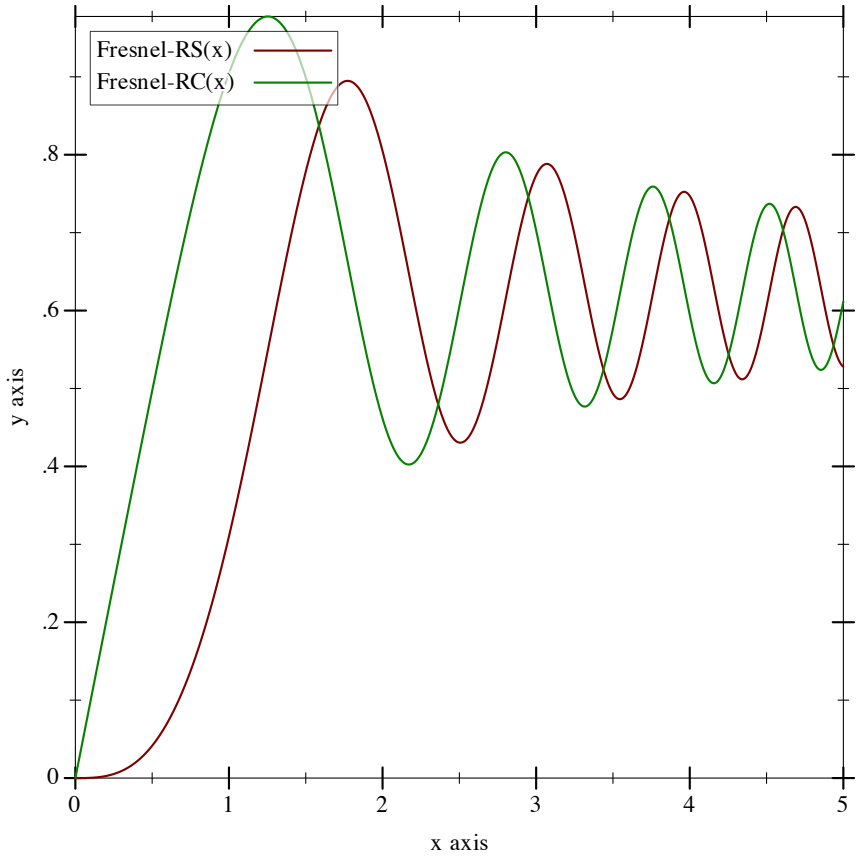
Compute the Fresnel integrals. Where

- `(Fresnel-S x)` calculates $\int \sin(\pi t^2/2)$ |0->x
- `(Fresnel-C x)` calculates $\int \cos(\pi t^2/2)$ |0->x
- `(Fresnel-RS x)` calculates $\int \sin(t^2)$ |0->x
- `(Fresnel-RC x)` calculates $\int \cos(t^2)$ |0->x

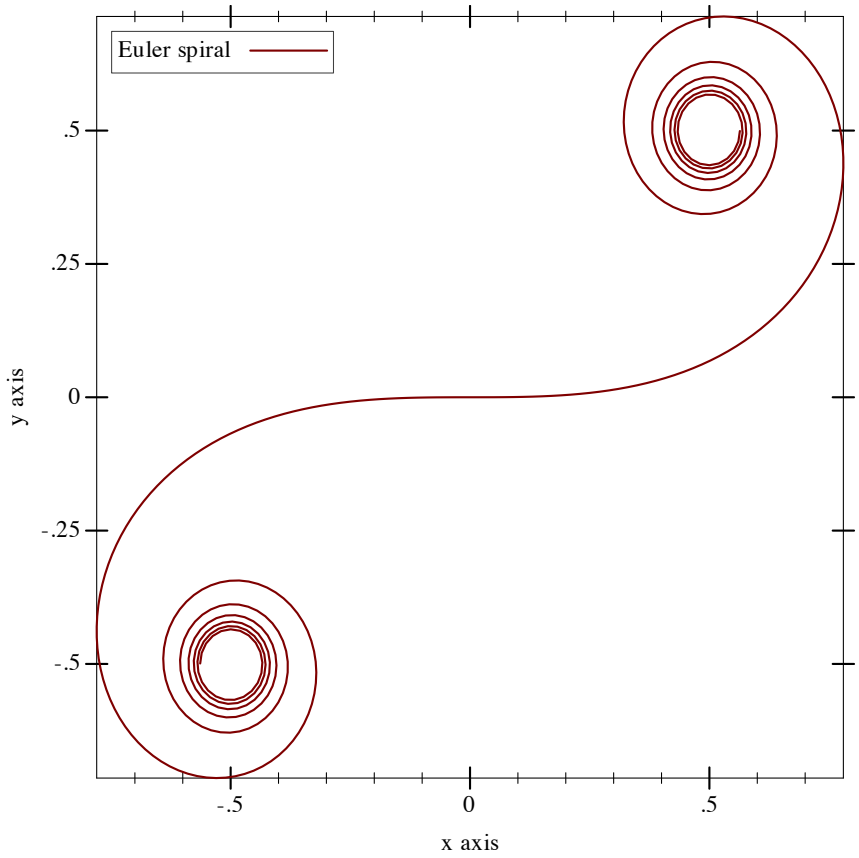
The first two are sometimes also referred to as the natural Fresnel integrals.

Examples:

```
> (plot (list (function Fresnel-RS 0 5 #:label "Fresnel-RS(x)")
              (function Fresnel-RC 0 5 #:color 2 #:label "Fresnel-
RC(x)")))
```




```
> (plot (parametric (λ (t) (list (Fresnel-C t) (Fresnel-
S t))) -5 5 #:label "Euler spiral"))
```




```
> (Fresnel-RS 1)
0.31026830172338116
> (* (sqrt (/ pi 2)) (Fresnel-S (* (sqrt (/ 2 pi)) 1)))
0.31026830172338116
```

Spot-checks within the region 0<=x<=150 sugest that the error is no greater than 1e-14 everywhere that has been tested, and usually is lower than 2e-15.

3.2 Flonum Functions

```
(flgamma x) → Flonum
  x : Flonum
```

```
(fllog-gamma x) → Flonum
  x : Flonum
```

```
(flpsi0 x) → Flonum
  x : Flonum
```

```
(flpsi m x) → Flonum
  m : Integer
  x : Flonum
```

```
(flerf x) → Flonum
  x : Flonum
```

```
(flerfc x) → Flonum
  x : Flonum
```

```
(fllambert x) → Flonum
  x : Flonum
```

```
(fllambert- x) → Flonum
  x : Flonum
```

```
(flzeta x) → Flonum
  x : Flonum
```

```
(fleta x) → Flonum
  x : Flonum
```

```
(flhurwitz-zeta s q) → Flonum
  s : Flonum
  q : Flonum
```

```
(flbeta x y) → Flonum
  x : Flonum
  y : Flonum
```

```
(fllog-beta x y) → Flonum
  x : Flonum
  y : Flonum
```

```
(flgamma-inc k x upper? regularized?) → Flonum
  k : Flonum
  x : Flonum
  upper? : Any
  regularized? : Any
```

```
(fllog-gamma-inc k x upper? regularized?) → Flonum
  k : Flonum
  x : Flonum
  upper? : Any
  regularized? : Any
```

```
(flbeta-inc a b x upper? regularized?) → Flonum
  a : Flonum
  b : Flonum
  x : Flonum
  upper? : Any
  regularized? : Any
```

```
(fllog-beta-inc a b x upper? regularized?) → Flonum
  a : Flonum
  b : Flonum
  x : Flonum
  upper? : Any
  regularized? : Any
```

```
(flFresnel-S x) → Flonum
  x : Flonum
```

```
(flFresnel-C x) → Flonum
  x : Flonum
```

Flonum versions of the above functions. These return `+nan.0` instead of raising errors and do not have optional arguments. They can be a little faster to apply because they check fewer special cases.

4 Number Theory

```
(require math/number-theory)
```
package: math-lib

4.1 Congruences and Modular Arithmetic

Wikipedia: Divisor

```
(divides? m n) → Boolean
  m : Integer
  n : Integer
```

Returns #t if m divides n, #f otherwise.

Formally, an integer m divides an integer n when there exists a unique integer k such that (* m k) = n.

Examples:

```
> (divides? 2 9)
#f
> (divides? 2 8)
#t
```

Note that 0 cannot divide anything:

```
> (divides? 0 5)
#f
> (divides? 0 0)
#f
```

Practically, if (divides? m n) is #t, then (/ n m) will return an integer and will not raise exn:fail:contract:divide-by-zero.

Wikipedia: Bezout's Identity

```
(bezout a b c ...) → (Listof Integer)
  a : Integer
  b : Integer
  c : Integer
```

Given integers a b c ... returns a list of integers (list u v w ...) such that (gcd a b c ...) = (+ (* a u) (* b v) (* c w) ...).

Examples:

```
> (bezout 6 15)
'(-2 1)
> (+ (* -2 6) (* 1 15))
3
> (gcd 6 15)
3
```

Wikipedia: Coprime

```
(coprime? a b ...) → Boolean
  a : Integer
  b : Integer
```

Returns `#t` if the integers `a b ...` are coprime. Formally, a set of integers is considered coprime (also called relatively prime) if their greatest common divisor is 1.

Example:

```
> (coprime? 2 6 15)
#t
```

Wikipedia: Pairwise Coprime

```
(pairwise-coprime? a b ...) → Boolean
  a : Integer
  b : Integer
```

Returns `#t` if the integers `a b ...` are *pairwise* coprime, meaning that each pair of integers is coprime.

The numbers 2, 6 and 15 are coprime, but not *pairwise* coprime, because 6 and 15 share the factor 3:

```
> (pairwise-coprime? 2 6 15)
#f
```

Wikipedia: Chinese Remainder Theorem

```
(solve-chinese as ns) → Natural
  as : (Listof Integer)
  ns : (Listof Integer)
```

Given a length-`k` list of integers `as` and a length-`k` list of coprime moduli `ns`, `(solve-chinese as ns)` returns the least natural number `x` that is a solution to the equations

$$x = a_1 \pmod{n_1}$$
$$\ldots$$
$$x = a_k \pmod{n_k}$$

The solution `x` is less than `(* n1 ... nk)`.

The moduli *ns* must all be positive.

What is the least number `x` that when divided by 3 leaves a remainder of 2, when divided by 5 leaves a remainder of 3, and when divided by 7 leaves a remainder of 2?

```
> (solve-chinese '(2 3 2) '(3 5 7))
23
```

Wikipedia: Quadratic Residue

```
(quadratic-residue? a n) → Boolean
  a : Integer
  n : Integer
```

Returns `#t` if `a` is a quadratic residue modulo *n*, otherwise `#f`. The modulus *n* must be positive, and `a` must be nonnegative.

Formally, `a` is a quadratic residue modulo *n* if there exists a number `x` such that `(* x x) = a` (mod *n*). In other words, `(quadratic-residue? a n)` is `#t` when `a` is a perfect square modulo *n*.

Examples:

```
> (quadratic-residue? 0 4)
#f
> (quadratic-residue? 1 4)
#t
> (quadratic-residue? 2 4)
#f
> (quadratic-residue? 3 4)
#f
```

Wikipedia: Legendre Symbol

```
(quadratic-character a p) → (U -1 0 1)
  a : Integer
  p : Integer
```

Returns the value of the quadratic character modulo the prime *p*. That is, for a non-zero `a` the number `1` is returned when `a` is a quadratic residue, and `-1` is returned when `a` is a non-residue. If `a` is zero, then `0` is returned.

If `a` is negative or *p* is not positive, `quadratic-character` raises an error. If *p* is not prime, `(quadratic-character a p)` is indeterminate.

This function is also known as the *Legendre symbol*.

```
> (quadratic-character 0 5)
0
> (quadratic-character 1 5)
1
> (quadratic-character 2 5)
-1
> (quadratic-character 3 5)
-1
```

Wikipedia: Jacobi Symbol

```
(jacobi-symbol a n) → (U -1 0 1)
  a : Nonnegative-Integer
  n : Positive-Integer
```

Computes the Jacobi symbol for any nonnegative integer `a` and any positive odd integer `n`.

If `n` is not an odd positive integer, `(jacobi-symbol a n)` throws an exception.

```
> (jacobi-symbol 1 1)
1
> (jacobi-symbol 8 11)
-1
> (jacobi-symbol 39 27)
0
> (jacobi-symbol 22 59)
1
> (jacobi-symbol 32 8)
jacobi: contract violation
  expected: odd?
  given: 8
  argument position: 2nd
  other arguments...:
   32
```

Wikipedia: Multiplicative Inverse

```
(modular-inverse a n) → Natural
  a : Integer
  n : Integer
```

Returns the inverse of `a` modulo `n` if `a` and `n` are coprime, otherwise raises an error. The modulus `n` must be positive, and `a` must be nonzero.

Formally, if `a` and `n` are coprime, `b = (modular-inverse a n)` is the unique natural number less than `n` such that `(* a b) = 1` (mod `n`).

```
> (modular-inverse 2 5)
3
> (modulo (* 2 3) 5)
1
```

```
(modular-expt a b n) → Natural
  a : Integer
  b : Integer
  n : Integer
```

Computes `(modulo (expt a b) n)`, but much more efficiently. The modulus `n` must be positive.

Examples:

```
> (modulo (expt -6 523) 19)
13
> (modular-expt -6 523 19)
13
> (modular-expt 9 158235208 19)
4
> (modular-expt 2 -1 11)
6
> ; don't try this at home!
  (modulo (expt 9 158235208) 19)
4
```

4.1.1 Parameterized Modular Arithmetic

Wikipedia: Modular Arithmetic

The `math/number-theory` library supports modular arithmetic parameterized on a current modulus. For example, the code

```
(with-modulus n
  ((modexpt a b) . mod= . c))
```

corresponds with the mathematical statement $a^b = c \pmod{n}$.

The current modulus is stored in a parameter that, for performance reasons, can only be set using `with-modulus`. (The basic modular operators cache parameter reads, and this restriction guarantees that the cached values are current.)

```
(with-modulus n body ...)

  n : Integer
```

Alters the current modulus within the dynamic extent of *body*. The expression *n* must evaluate to a positive integer.

By default, the current modulus is 1, meaning that every modular arithmetic expression that does not raise an error returns 0.

```
(current-modulus) → Positive-Integer
```

Returns the current modulus.

Examples:

```
> (current-modulus)
1
> (with-modulus 5 (current-modulus))
5
```

```
(mod x) → Natural
  x : Exact-Rational
```

Converts a rational number *x* to a natural number less than the current modulus.

If *x* is an integer, this is equivalent to `(modulo x n)`. If *x* is a fraction, an integer input is generated by multiplying its numerator by its denominator's modular inverse.

Examples:

```
> (with-modulus 7 (mod (* 218 7)))
0
> (with-modulus 7 (mod 3/2))
5
> (with-modulus 7 (mod/ 3 2))
5
> (with-modulus 7 (mod 3/7))
modular-inverse: expected argument that is coprime to
modulus 7; given 7
```

```
(mod+ a ...) → Natural
  a : Integer
(mod* a ...) → Natural
  a : Integer
```

Equivalent to `(modulo (+ a ...) (current-modulus))` and `(modulo (* a ...) (current-modulus))`, respectively, but generate smaller intermediate values.

```
(modsqr a) → Natural
  a : Integer
(modexpt a b) → Natural
  a : Integer
  b : Integer
```

Equivalent to `(mod* a a)` and `(modular-expt a b (current-modulus))`, respectively.

```
(mod- a b ...) → Natural
  a : Integer
  b : Integer
```

Equivalent to `(modulo (- a b ...) (current-modulus))`, but generates smaller intermediate values. Note that `(mod- a) = (mod (- a))`.

```
(mod/ a b ...) → Natural
  a : Integer
  b : Integer
```

Divides `a` by `(* b ...)`, by multiplying `a` by the multiplicative inverse of `(* b ...)`. The one-argument variant returns the modular inverse of `a`.

Note that `(mod/ a b ...)` is **not** equivalent to `(modulo (/ a b ...) (current-modulus))`; see `mod=` for a demonstration.

```
(mod= a b ...) → Boolean
  a : Integer
  b : Integer
(mod< a b ...) → Boolean
  a : Integer
  b : Integer
(mod<= a b ...) → Boolean
  a : Integer
  b : Integer
(mod> a b ...) → Boolean
  a : Integer
  b : Integer
(mod>= a b ...) → Boolean
  a : Integer
  b : Integer
```

Each of these is equivalent to `(op (mod a) (mod b) ...)`, where `op` is the corresponding numeric comparison function. Additionally, when given one argument, the inequality tests always return `#t`.

Suppose we wanted to know why 17/4 = 8 (mod 15), but 51/12 (mod 15) is undefined, even though normally 51/12 = 17/4. In code,

```
> (with-modulus 15 (mod/ 17 4))
8
> (/ 51 12)
17/4
> (with-modulus 15 (mod/ 51 12))
modular-inverse: expected argument that is coprime to
modulus 15; given 12
```

We could try to divide by brute force: find, modulo 15, all the numbers `a` for which `(mod* a 4)` is `17`, then find all the numbers `b` for which `(mod* a 12)` is `51`.

```
> (with-modulus 15
    (for/list ([a  (in-range 15)]
               #:when (mod= (mod* a 4) 17))
      a))
'(8)
> (with-modulus 15
    (for/list ([b  (in-range 15)]
               #:when (mod= (mod* b 12) 51))
      b))
'(3 8 13)
```

So the problem isn't that `b` doesn't exist, it's that `b` isn't *unique*.

4.2 Primes

Wikipedia: Prime Number

```
(prime? z) → Boolean
  z : Integer
```

Returns `#t` if `z` is a prime, `#f` otherwise.

Formally, an integer `z` is prime when the only positive divisors of `z` are `1` and `(abs z)`.

The positive primes below 20 are:

```
> (filter prime? (range 1 21))
'(2 3 5 7 11 13 17 19)
```

The corresponding negative primes are:

```
> (filter prime? (range 1 -21 -1))
'(-2 -3 -5 -7 -11 -13 -17 -19)
```

(odd-prime? z) → Boolean
  z : Integer

Returns #t if z is a odd prime, #f otherwise.

```
> (odd-prime? 2)
#f
> (odd-prime? 3)
#t
```

(nth-prime n) → Natural
  n : Integer

Returns the nth positive prime; n must be nonnegative.

```
> (nth-prime 0)
2
> (nth-prime 1)
3
> (nth-prime 2)
5
```

(random-prime n) → Natural
  n : Integer

Returns a random prime smaller than n, which must be greater than 2.

The function random-prime picks random numbers below n until a prime is found.

```
> (random-prime 10)
2
> (random-prime 10)
5
> (random-prime 10)
3
```

(next-prime z) → Integer
  z : Integer

Returns the first prime larger than z.

```
> (next-prime 4)
5
> (next-prime 5)
7
```

```
(prev-prime z) → Integer
  z : Integer
```

Returns the first prime smaller than z.

```
> (prev-prime 4)
3
> (prev-prime 5)
3
```

```
(next-primes z n) → (Listof Integer)
  z : Integer
  n : Integer
```

Returns list of the next n primes larger than z; n must be nonnegative.

```
> (next-primes 2 4)
'(3 5 7 11)
```

```
(prev-primes z n) → (Listof Integer)
  z : Integer
  n : Integer
```

Returns list of the next n primes smaller than z; n must be nonnegative.

```
> (prev-primes 13 4)
'(11 7 5 3)
```

Wikipedia: Integer Factorization

```
(factorize n) → (Listof (List Natural Natural))
  n : Natural
```

Returns the factorization of a natural number n. The factorization consists of a list of corresponding primes and exponents. The primes will be in ascending order.

The prime factorization of 600 = 2^3 * 3^1 * 5^2:

```
> (factorize 600)
'((2 3) (3 1) (5 2))
```

```
(defactorize f) → Natural
  f : (Listof (List Natural Natural))
```

Returns the natural number, whose factorization is given by `f`. The factorization `f` is represented as described in `factorize`.

```
> (defactorize '((2 3) (3 1) (5 2)))
600
```

```
(divisors z) → (Listof Natural)
  z : Integer
```

Returns a list of all positive divisors of the integer `z`. The divisors appear in ascending order.

```
> (divisors 120)
'(1 2 3 4 5 6 8 10 12 15 20 24 30 40 60 120)
> (divisors -120)
'(1 2 3 4 5 6 8 10 12 15 20 24 30 40 60 120)
```

```
(prime-divisors z) → (Listof Natural)
  z : Natural
```

Returns a list of all positive prime divisors of the integer `z`. The divisors appear in ascending order.

```
> (prime-divisors 120)
'(2 3 5)
```

```
(prime-exponents z) → (Listof Natural)
  z : Natural
```

Returns a list of the exponents of in a factorization of the integer `z`.

```
> (define z (* 2 2 2 3 5 5))
> (prime-divisors z)
'(2 3 5)
> (prime-exponents z)
'(3 1 2)
```

4.3 Roots

```
(integer-root n m) → Natural
  n : Natural
  m : Natural
```

Returns the mth integer root of n. This is the largest integer `r` such that `(expt r m) <= n`.

```
> (integer-root (expt 3 4) 4)
3
> (integer-root (+ (expt 3 4) 1) 4)
3
```

```
(integer-root/remainder n m) → Natural Natural
  n : Natural
  m : Natural
```

Returns two values. The first, `r`, is the mth integer root of n. The second is `n-r^m`.

```
> (integer-root/remainder (expt 3 4) 4)
3
0
> (integer-root/remainder (+ (expt 3 4) 1) 4)
3
1
```

4.4 Powers

```
(max-dividing-power a b) → Natural
  a : Integer
  b : Integer
```

Returns the largest exponent, `n`, of a power with base `a` that divides b.

That is, `(expt a n)` divides b but `(expt a (+ n 1))` does not divide b.

```
> (max-dividing-power 3 (expt 3 4))
4
> (max-dividing-power 3 5)
0
```

```
(perfect-power m) → (U (List Natural Natural) #f)
  m : Integer
```

If m is a perfect power, a list with two elements `b` and `n` such that `(expt b n) = m` is returned, otherwise `#f` is returned.

```
> (perfect-power (expt 3 4))
'(3 4)
> (perfect-power (+ (expt 3 4) 1))
#f
```

```
(perfect-power? m) → Boolean
  m : Integer
```

Returns `#t` if `m` is a perfect power, otherwise `#f`.

```
> (perfect-power? (expt 3 4))
#t
> (perfect-power? (+ (expt 3 4) 1))
#f
```

```
(prime-power m) → (U (List Natural Natural) #f)
  m : Natural
```

If `m` is a power of the form `(expt p n)` where `p` is prime, then a list with the prime and the exponent is returned, otherwise `#f` is returned.

```
> (prime-power (expt 3 4))
'(3 4)
> (prime-power (expt 6 4))
#f
```

```
(prime-power? m) → Boolean
  m : Natural
```

Returns `#t` if `m` is a prime power, otherwise `#f`.

```
> (prime-power? (expt 3 4))
#t
> (prime-power? (expt 6 4))
#f
> (prime-power? 1)
#f
> (prime-power? 0)
#f
```

```
(odd-prime-power? m) → Boolean
  m : Natural
```

Returns `#t` if `m` is a power of an odd prime, otherwise `#f`.

```
> (odd-prime-power? (expt 2 4))
#f
> (odd-prime-power? (expt 3 4))
#t
> (odd-prime-power? (expt 15 4))
#f
```

```
(as-power m) → Natural Natural
  m : Positive-Integer
```

Returns two values `b` and `n` such that `m = (expt b n)` and `n` is maximal.

```
> (as-power (* (expt 2 4) (expt 3 4)))
6
4
> (expt 6 4)
1296
> (* (expt 2 4) (expt 3 4))
1296
> (as-power (* (expt 2 4) (expt 3 5)))
3888
1
```

```
(perfect-square m) → (U Natural #f)
  m : Natural
```

Returns `(sqrt m)` if `m` is perfect square, otherwise `#f`.

```
> (perfect-square 9)
3
> (perfect-square 10)
#f
```

4.5 Multiplicative and Arithmetic Functions

The functions in this section are *multiplicative* (with exception of the Von Mangoldt function). In number theory, a multiplicative function is a function `f` such that `(f (* a b)) = (* (f a) (f b))` for all coprime natural numbers `a` and `b`.

Wikipedia: Euler's Totient

```
(totient n) → Natural
  n : Natural
```

Returns the number of integers from 1 to `n` that are coprime with `n`.

This function is known as Eulers totient or phi function.

```
> (totient 9)
6
> (length (filter (curry coprime? 9) (range 10)))
6
```

Wikipedia: Moebius Function

```
(moebius-mu n) → (U -1 0 1)
  n : Natural
```

Returns:

- `1` if `n` is a square-free product of an even number of primes
- `-1` if `n` is a square-free product of an odd number of primes
- `0` if `n` has a multiple prime factor

```
> (moebius-mu (* 2 3 5))
-1
> (moebius-mu (* 2 3 5 7))
1
> (moebius-mu (* 2 2 3 5 7))
0
```

Wikipedia: Divisor Function

```
(divisor-sum n k) → Natural
  n : Natural
  k : Natural
```

Returns sum of the `k`th powers of all divisors of `n`.

```
> (divisor-sum 12 2)
210
> (apply + (map sqr (divisors 12)))
210
```

OEIS: Big Omega

```
(prime-omega n) → natural?
  n : Natural
```

Counting multiplicities the number of prime factors of `n` is returned.

```
> (prime-omega (* 2 2 2 3 3 5))
6
```

Wikipedia: Von Mangoldt Function

```
(mangoldt-lambda n) → Real
  n : Natural
```

The von Mangoldt function. If `n=p^k` for a prime `p` and an integer `k>=1` then `(log n)` is returned. Otherwise 0 is returned.

Note: The Von Mangoldt function is not multiplicative.

```
> (mangoldt-lambda (* 3 3))
1.0986122886681098
> (log 3)
1.0986122886681098
```

4.6 Number Sequences

Wikipedia: Bernoulli Number

```
(bernoulli-number n) → Exact-Rational
  n : Integer
```

Returns the *n*th Bernoulli number; *n* must be nonnegative.

```
> (map bernoulli-number (range 9))
'(1 -1/2 1/6 0 -1/30 0 1/42 0 -1/30)
```

Note that these are the *first* Bernoulli numbers, since `(bernoulli-number 1) = -1/2`.

MathWorld: Eulerian Number

```
(eulerian-number n k) → Natural
  n : Integer
  k : Integer
```

Returns the Eulerian number <n,k>; both arguments must be nonnegative.

```
> (eulerian-number 5 2)
66
```

Wikipedia: Fibonacci Number

```
(fibonacci n) → Natural
  n : Integer
```

Returns the nth Fibonacci number; n must be nonnegative.

The ten first Fibonacci numbers.

```
> (map fibonacci (range 10))
'(0 1 1 2 3 5 8 13 21 34)
```

```
(make-fibonacci a b) → (Integer -> Integer)
  a : Integer
  b : Integer
```

Returns a function representing a Fibonacci sequence with the first two numbers `a` and `b`. The `fibonacci` function is defined as `(make-fibonacci 0 1)`.

Wikipedia: Lucas Number

The Lucas numbers are defined as a Fibonacci sequence starting with 2 and 1:

```
> (map (make-fibonacci 2 1) (range 10))
'(2 1 3 4 7 11 18 29 47 76)
```

```
(modular-fibonacci n m) → Natural
  n : Integer
  m : Integer
```

Returns the nth Fibonacci number modulo m; n must be nonnegative and m must be positive.

The ten first Fibonacci numbers modulo 5.

```
> (map (λ (n) (modular-fibonacci n 5)) (range 10))
'(0 1 1 2 3 0 3 3 1 4)
```

```
(make-modular-fibonacci a b) → (Integer Integer -> Integer)
  a : Integer
  b : Integer
```

Like `make-fibonacci`, but makes a modular Fibonacci sequence.

Wikipedia: Farey Sequence

```
(farey-sequence n) → (Listof Exact-Rational)
  n : Integer
```

Returns a list of the numbers in the nth Farey sequence; n must be positive.

The nth Farey sequence is the sequence of all completely reduced rational numbers from 0 to 1 which denominators are less than or equal to n.

```
> (farey-sequence 1)
'(0 1)
> (farey-sequence 2)
'(0 1/2 1)
> (farey-sequence 3)
'(0 1/3 1/2 2/3 1)
```

MathWorld: Tangent Number

```
(tangent-number n) → Integer
  n : Integer
```

Returns the nth tangent number; n must be nonnegative.

```
> (tangent-number 1)
1
> (tangent-number 2)
0
> (tangent-number 3)
2
```

4.7 Combinatorics

Wikipedia: Factorial

```
(factorial n) → Natural
  n : Integer
```

Returns the factorial of n, which must be nonnegative. The factorial of n is the number `(* n (- n 1) (- n 2) ... 1)`.

```
> (factorial 3)
6
> (factorial 0)
1
```

Wikipedia: Binomial Coefficient

```
(binomial n k) → Natural
  n : Integer
  k : Integer
```

Returns the number of ways to choose a *set* of k items from a set of n items; i.e. the order of the k items is not significant. Both arguments must be nonnegative.

When `k > n`, `(binomial n k) = 0`. Otherwise, `(binomial n k)` is equivalent to `(/ (factorial n) (factorial k) (factorial (- n k)))`, but computed more quickly.

```
> (binomial 5 3)
10
```

Wikipedia: Permutations

```
(permutations n k) → Natural
  n : Integer
  k : Integer
```

Returns the number of ways to choose a *sequence* of `k` items from a set of `n` items; i.e. the order of the `k` items is significant. Both arguments must be nonnegative.

When `k > n`, `(permutations n k) = 0`. Otherwise, `(permutations n k)` is equivalent to `(/ (factorial n) (factorial (- n k)))`.

```
> (permutations 5 3)
60
```

Wikipedia: Multinomial Coeffecient

```
(multinomial n ks) → Natural
  n : Integer
  ks : (Listof Integer)
```

A generalization of `binomial` to multiple sets of choices; e.g. `(multinomial n (list k0 k1 k2))` is the number of ways to choose a set of `k0` items, a set of `k1` items, and a set of `k2` items from a set of `n` items. All arguments must be nonnegative.

When `(apply + ks) = n`, this is equivalent to `(apply / (factorial n) (map factorial ks))`. Otherwise, `multinomial` returns `0`.

```
> (multinomial 5 '(3 2))
10
> (= (multinomial 8 '(5 3))
     (binomial 8 5)
     (binomial 8 3))
#t
> (multinomial 10 '(5 3 2))
2520
> (multinomial 0 '())
1
> (multinomial 4 '(1 1))
0
```

Wikipedia: Partition

```
(partitions n) → Natural
  n : Integer
```

Returns the number of partitions of `n`, which must be nonnegative. A partition of a positive integer `n` is a way of writing `n` as a sum of positive integers. The number 3 has the partitions `(+ 1 1 1)`, `(+ 1 2)` and `(+ 3)`.

```
> (partitions 3)
3
> (partitions 4)
5
```

4.8 Special Numbers

4.8.1 Polygonal Numbers

Wikipedia: Polygonal Number

```
(triangle-number? n) → Boolean
  n : Natural
(square-number? n) → Boolean
  n : Natural
(pentagonal-number? n) → Boolean
  n : Natural
(hexagonal-number? n) → Boolean
  n : Natural
(heptagonal-number? n) → Boolean
  n : Natural
(octagonal-number? n) → Boolean
  n : Natural
```

These functions check whether the input is a polygonal number of the types triangle, square, pentagonal, hexagonal, heptagonal and octogonal respectively.

```
(triangle-number n) → Natural
  n : Natural
(sqr n) → Natural
  n : Natural
(pentagonal-number n) → Natural
  n : Natural
(hexagonal-number n) → Natural
  n : Natural
(heptagonal-number n) → Natural
  n : Natural
(octagonal-number n) → Natural
  n : Natural
```

These functions return the `n`th polygonal number of the corresponding type of polygonal number.

Wikipedia: Mediant

4.9 Fractions

```
(mediant x y) → Exact-Rational
  x : Exact-Rational
  y : Exact-Rational
```

Computes the mediant of the numbers x and y. The mediant of two fractions p/q and r/s in their lowest term is the number (p+r)/(q+s).

```
> (mediant 1/2 5/6)
3/4
```

4.10 The Quadratic Equation

```
(quadratic-solutions a b c) → (Listof Real)
  a : Real
  b : Real
  c : Real
```

Returns a list of all real solutions to the equation a x^2 + b x +c = 0.

```
> (quadratic-solutions 1 0 -1)
'(-1 1)
> (quadratic-solutions 1 2 1)
'(-1)
> (quadratic-solutions 1 0 1)
'()
```

The implementation of quadratic-solutions handles cancellation and overflow intelligently:

```
> (quadratic-solutions 1e+200 2e+200 1e+200)
'(-1.0)
> (quadratic-solutions 1e-200 -2e-200 1e-200)
'(1.0)
```

For exact inputs, if the output can be exactly represented, it will be:

```
> (quadratic-solutions -1 2/3 1/3)
'(1 -1/3)
```

```
(quadratic-integer-solutions a b c) → (Listof Integer)
  a : Real
  b : Real
  c : Real
```

Returns a list of all integer solutions to the equation `a x^2 + b x +c = 0`.

```
> (quadratic-integer-solutions 1 0 -1)
'(-1 1)
> (quadratic-integer-solutions 1 0 -2)
'()
```

```
(quadratic-natural-solutions a b c) → (Listof Natural)
  a : Real
  b : Real
  c : Real
```

Returns a list of all natural solutions to the equation `a x^2 + b x +c = 0`.

```
> (quadratic-natural-solutions 1 0 -1)
'(1)
> (quadratic-natural-solutions 1 0 -2)
'()
```

```
(complex-quadratic-solutions a b c) → (Listof Complex)
  a : Complex
  b : Complex
  c : Complex
```

Returns a list of all complex solutions to the equation `a x^2 + b x +c = 0`. This function allows complex coeffecients.

```
> (complex-quadratic-solutions 1 0 1)
'(0-1i 0+1i)
> (complex-quadratic-solutions 1 0 (sqrt -1))
'(-0.7071067811865476+0.7071067811865475i
  0.7071067811865476-0.7071067811865475i)
> (complex-quadratic-solutions 1 0 1)
'(0-1i 0+1i)
```

Added in version 1.1 of package `math-lib`.

4.11 The group Zn and Primitive Roots

Wikipedia: The Group Zn

The numbers `0, 1, ..., n-1` with addition and multiplication modulo `n` is a ring called `Zn`.

The group of units in `Zn` with respect to multiplication modulo `n` is called `Un`.

The order of an element x in Un is the least k>0 such that x^k=1 mod n.

A generator the group Un is called a *primitive root* modulo n. Note that g is a primitive root if and only if order(g)=totient(n). A group with a generator is called *cyclic*.

```
(unit-group n) → (Listof Positive-Integer)
  n : Integer
```

Returns a list of all elements of Un, the unit group modulo *n*. The modulus *n* must be positive.

```
> (unit-group 5)
'(1 2 3 4)
> (unit-group 6)
'(1 5)
```

```
(unit-group-order x n) → Positive-Integer
  x : Integer
  n : Integer
```

Returns the order of *x* in the group Un; both arguments must be positive. If *x* and *n* are not coprime, (unit-group-order *x* *n*) raises an error.

```
> (unit-group-order 2 5)
4
> (unit-group-order 2 6)
unit-group-order: expected coprime arguments; given 2 and 6
```

```
(unit-group-orders n) → (Listof Positive-Integer)
  n : Integer
```

Returns a list (list (unit-group-order x0 *n*) (unit-group-order x1 *n*) ...) where x0, x1, ... are the elements of Un. The modulus *n* must be positive.

```
> (unit-group-orders 5)
'(1 4 4 2)
> (map (curryr unit-group-order 5) (unit-group 5))
'(1 4 4 2)
```

```
(primitive-root? x n) → Boolean
  x : Integer
  n : Integer
```

Returns #t if the element *x* in Un is a primitive root modulo *n*, otherwise #f is returned. An error is signaled if *x* is not a member of Un. Both arguments must be positive.

```
> (primitive-root? 1 5)
#f
> (primitive-root? 2 5)
#t
> (primitive-root? 5 5)
primitive-root?: expected coprime arguments; given 5 and 5
```

```
(exists-primitive-root? n) → Boolean
  n : Integer
```

Returns `#t` if the group `Un` has a primitive root (i.e. it is cyclic), otherwise `#f` is returned. In other words, `#t` is returned if *n* is one of `1, 2, 4, p^e, 2*p^e` where `p` is an odd prime, and `#f` otherwise. The modulus *n* must be positive.

```
> (exists-primitive-root? 5)
#t
> (exists-primitive-root? 6)
#t
> (exists-primitive-root? 12)
#f
```

```
(primitive-root n) → (Union Natural #f)
  n : Integer
```

Returns a primitive root of `Un` if one exists, otherwise `#f` is returned. The modulus *n* must be positive.

```
> (primitive-root 5)
2
> (primitive-root 6)
5
```

```
(primitive-roots n) → (Listof Natural)
  n : Integer
```

Returns a list of all primitive roots of `Un`. The modulus *n* must be positive.

```
> (primitive-roots 3)
'(2)
> (primitive-roots 5)
'(2 3)
> (primitive-roots 6)
'(5)
```

5 Arbitrary-Precision Floating-Point Numbers (*Bigfloats*)

`(require math/bigfloat)` package: `math-lib`

This library provides a Typed Racket interface to MPFR, a C library that provides

- A C type of arbitrary-precision floating-point numbers.
- Elementary and special functions that are efficient and proved correct.
- Well-defined semantics that correspond with the latest IEEE 754 standard.

The arbitrary-precision floating-point numbers MPFR provides and operates on are represented by the Typed Racket type `Bigfloat` and identified by the predicate `bigfloat?`.

With a few noted exceptions, bigfloat functions regard their arguments as if they were exact, regardless of their precision. Conceptually, they compute exact results using infinitely many bits, and return results with `(bf-precision)` bits by rounding them using `(bf-rounding-mode)`. In practice, they use finite algorithms that have been painstakingly proved to be equivalent to that conceptual, infinite process.

MPFR is free and license-compatible with commercial software. It is distributed with Racket for Windows and Mac OS X, is installed on most Linux systems, and is easy to install on major Unix-like platforms.

5.1 Quick Start

1. Set the bigfloat function result precision using `(bf-precision <some-number-of-bits>)`.
2. Use `bf` to convert real values and well-formed strings to bigfloats.
3. Operate on bigfloats using `bf`-prefixed functions like `bf+` and `bfsin`.
4. Convert bigfloats to real values using `bigfloat->real`, `bigfloat->flonum`, and `bigfloat->integer`. Format them for display using `bigfloat->string`.

For examples, continue through the FAQ.

5.2 Fictionally Asked Questions

Why use `math/bigfloat`?

There are a few reasons.

Reason: Flonums have either too much or too little precision for your application.

Examples:

```
> (flsqrt 3.0)
1.7320508075688772
> pi
3.141592653589793
> (bf-precision 16)
> (bfsqrt (bf 3))
(bf #e1.73206)
> (bf-precision 179)
> pi.bf
(bf #e3.14159265358979323846264338327950288419716939937510582)
```

A flonum has a 53-bit significand (we'll say it has 53 bits of *precision*) and an 11-bit *exponent*. A bigfloat has an arbitrary precision of at least 2 bits and a 31-bit exponent.

Reason: To compute ridiculously large or small numbers with confidence.

Examples:

```
> (bf-precision 128)
> (bfexp (bfexp (bfexp (bf 3))))
(bf "2.0509864360516488951058609420720546745790e229520860")
> (bflog (bflog (bflog (bfexp (bfexp (bfexp (bf 3)))))))
(bf 3)
```

Reason: To verify your floating-point hardware.

IEEE 754-2008 stipulates that conforming implementations must correctly round the results of all operations. Roughly speaking, results can't be more than half a bit off, where the bit in question is the least significant in the significand.

Of course, implementations don't always adhere to standards. For example, on my old laptop, evaluating `(exp 400)` results in `5.221469689764346e+173`. Note the last four decimal digits in the significand: `4346`. But they should be `4144`:

```
> (bf-precision 53)
> (bigfloat->flonum (bfexp (bf 400)))
5.221469689764144e+173
```

My new laptop computes `5.221469689764144e+173` as it should.

Reason: To control rounding of the least significant bit.

IEEE 754 provides for different rounding modes for the smallest bit of a flonum result, such as round to even and round toward zero. We might use this to implement interval arithmetic correctly, by rounding lower bounds downward and upper bounds upward. But there isn't a portable way to set the rounding mode!

MPFR allows the rounding mode to be different for any operation, and `math/bigfloat` exposes this capability using the parameter `bf-rounding-mode`.

When shouldn't I use `math/bigfloat`?

When you need raw speed. Bigfloat functions can be hundreds to thousands of times slower than flonum functions.

That's not to say that they're *inefficient*. For example, `bflog` implements the algorithm with the best known asymptotic complexity. It just doesn't run directly on hardware, and it can't take fixed-precision-only shortcuts.

Why are there junk digits on the end of `(bf 1.1)`?

That's approximately the value of the flonum `1.1`. Use `(bf #e1.1)` or `(bf "1.1")` to make the junk go away. In general, you should prefer to convert exact rationals and strings to bigfloats.

Why is the last digit of `pi.bf` not rounded correctly?

All the *bits* but the last is exact, and the last bit is correctly rounded. This doesn't guarantee that the last digit will be.

A decimal digit represents at most log(10)/log(2) ≈ 3.3 bits. This is an irrational number, so the decimal/bit boundary never lines up except at the decimal point. Thus, the last decimal digit of any bigfloat must represent fewer than 3.3 bits, so it's wrong more often than not. But it's the last *bit* that counts.

5.3 Type and Constructors

```
Bigfloat
(bigfloat? v) → Boolean
  v : Any
```

An opaque type that represents an arbitrary-precision floating-point number, or a bigfloat, and the opaque type's predicate.

```
(bf x) → Bigfloat
  x : (U String Real)
```

```
(bf sig exp) → Bigfloat
  sig : Integer
  exp : Integer
```

The one-argument variant converts a string or real *x* to a bigfloat.

```
> (bf-precision 128)
> (bf 4)
(bf 4)
> (bf 1/7)
(bf #e0.1428571428571428571428571428571428571426)
> (bf 41/10)
(bf #e4.099999999999999999999999999999999999995)
> (bf "not a number")
bf: expected a well-formed decimal number; given "not a number"
> (bf "15e200000000")
(bf "1.499999999999999999999999999999999999998e200000001")
```

In the last example, the result of `(bf "15e200000000")` is displayed as a string conversion because the exact rational number would be very large.

Converting from flonum literals is usually a bad idea* because flonums have only 53 bits precision. Prefer to pass exact rationals and strings to `bf`.

* It can be a good idea if you're testing a flonum implementation of a function against a bigfloat implementation.

The two-argument variant converts a signed significand *sig* and a power of 2 *exp* to a bigfloat. Generally, `(bf sig exp) = (bf (* sig (expt 2 exp)))`, but the two-argument variant is much faster, especially for large *exp*.

Examples:

```
> (bf 200 56)
(bf 14411518807585587200)
> (bf (* 200 (expt 2 56)))
(bf 14411518807585587200)
```

The `bfrandom` function generates random bigfloats between 0 and 1 using the two-argument variant in this way:

```
> (require (only-in math/base random-bits))
> (bf-precision 64)
> (bf (random-bits 64) -64)
(bf #e0.416872969910248753552)
```

```
(bfrandom) → Bigfloat
```

Returns a uniformly distributed random bigfloat in the interval [0,1].

```
(bfcopy x) → Bigfloat
  x : Bigfloat
```

Returns the (bf-precision)-bit bigfloat closest to x, under the current bf-rounding-mode.

A common pattern to compute bigfloats in higher precision is

```
> (bf-precision 64)
> (bfcopy
   (parameterize ([bf-precision  (+ (bf-precision) 10)])
     (bf/ (bf+ 1.bf (bfsqrt 5.bf)) 2.bf)))
(bf #e1.61803398874989484821)
```

This example computes the golden ratio (phi.bf) with 10 bits more than requested, to make up for triple rounding error.

5.4 Accessors and Conversion Functions

```
(bigfloat-precision x) → Exact-Positive-Integer
  x : Bigfloat
```

Returns the number of bits in the significand of x. This is almost always the value of (bf-precision) when x was created.

```
(bigfloat-signbit x) → (U 0 1)
  x : Bigfloat
```

Returns the sign bit of the significand of x.

Examples:

```
> (bigfloat-signbit -1.bf)
1
> (bigfloat-signbit 0.bf)
0
> (bigfloat-signbit -0.bf)
1
> (bigfloat-signbit -inf.bf)
1
```

```
(bigfloat-significand x) → Integer
  x : Bigfloat
(bigfloat-exponent x) → Integer
  x : Bigfloat
```

Return the *signed* significand or exponent of `x`.

To access the significand and exponent at the same time, use `bigfloat->sig+exp`.

```
(bigfloat->sig+exp x) → (Values Integer Integer)
  x : Bigfloat
```

Returns the *signed* significand and exponent of `x`.

If `(values sig exp) = (bigfloat->sig+exp x)`, its value as an exact rational is `(* sig (expt 2 exp))`. In fact, `bigfloat->rational` converts bigfloats to rationals in exactly this way, after ensuring that `(bfrational? x)` is `#t`.

This function and the two-argument variant of `bf` are mutual inverses.

```
(bigfloat->integer x) → Integer
  x : Bigfloat
(bigfloat->rational x) → Exact-Rational
  x : Bigfloat
(bigfloat->real x) → (U Exact-Rational Flonum)
  x : Bigfloat
(bigfloat->flonum x) → Flonum
  x : Bigfloat
```

Convert bigfloats to integer, exact rational, real and flonum values respectively.

`bigfloat->integer`, `bigfloat->rational` and `bigfloat->real` return values that can be converted exactly back to `x` using `bf`. For the first two, this is done by raising an error if `x` is not respectively integer or rational. On the other hand, `bigfloat->real` returns `+inf.0`, `-inf.0` or `+nan.0` when `x` is not a rational bigfloat.

`bigfloat->flonum` rounds `x` to 53 bits precision to fit the value into a flonum, using the current value of `bf-rounding-mode`.

```
> (bf-precision 64)
> (bigfloat->integer (bf 21/10))
bigfloat->integer: contract violation
  expected: bfinteger?
  given: (bf #e2.09999999999999999991)
> (bigfloat->integer (bfround (bf 21/10)))
```

```
2
> (define x (bf 1/7))
> (bigfloat->flonum x)
0.14285714285714285
> (bigfloat->rational x)
1054099661354831520 9/7378697629483820646 4
> (rationalize (bigfloat->rational x) (expt 2 (- (bf-precision))))
1/7
> (bf= x (bf (bigfloat->rational x)))
#t
```

Be careful with exact conversions. Bigfloats with large exponents may not fit in memory as integers or exact rationals. Worse, they might fit, but have all your RAM and swap space for lunch.

```
(bigfloat->string x) → String
  x : Bigfloat
(string->bigfloat s) → (U Bigfloat False)
  s : String
```

Convert a bigfloat *x* to a string *s* and back.

The string returned by `bigfloat->string` includes enough digits that `string->bigfloat` can reconstruct the bigfloat precisely. In other words, `string->bigfloat` is a left inverse of `bigfloat->string`.

If *s* isn't a well-formed decimal number with an optional exponent part, `string->bigfloat` returns `#f`. (In contrast, `(bf s)` raises an error.)

Examples:

```
> (bf-precision 64)
> (bigfloat->string (bf 4))
"4"
> (bigfloat->string (bf 1/10000))
"1.0000000000000000001e-4"
> (string->bigfloat "0.14285714285714285714")
(bf #e0.1428571428571428571 4)
> (string->bigfloat "square root of two")
#f
> (string->bigfloat (bigfloat->string pi.bf))
(bf #e3.141592653589793238 51)
> pi.bf
(bf #e3.141592653589793238 51)
```

5.5 Parameters

```
(bf-precision) → Integer
(bf-precision bits) → void?
  bits : Integer
```

A parameter that determines the precision of bigfloats returned from most bigfloat functions. Exceptions are noted in the documentation for functions that do not use `bf-precision`.

For nonzero, rational bigfloats, the number of bits *bits* includes the leading one bit. For example, to simulate 64-bit floating point, use `(bf-precision 53)` even though flonums have a 52-bit significand, because the one bit is implicit in a flonum.

This parameter has a guard that ensures `(bf-precision)` is between `bf-min-precision` and `bf-max-precision`.

```
(bf-rounding-mode) → (U 'nearest 'zero 'up 'down)
(bf-rounding-mode mode) → void?
  mode : (U 'nearest 'zero 'up 'down)
```

A parameter that determines the mode used to round the results of most bigfloat functions. Conceptually, rounding is applied to infinite-precision results to fit them into `(bf-precision)` bits.

```
bf-min-precision : Exact-Positive-Integer
```

Equal to 2, because single-bit bigfloats can't be correctly rounded.

```
bf-max-precision : Exact-Positive-Integer
```

The largest value of `(bf-precision)`. This is platform-dependent, and probably much larger than you'll ever need.

5.6 Constants

Most bigfloat "constants" are actually identifier macros that expand to the application of a zero-argument function. This allows, for example, `pi.bf` to depend on the current value of `bf-precision`, and allows all of them to be constructed lazily. Most constants are memoized, possibly at multiple precisions.

```
pi.bf : Bigfloat
phi.bf : Bigfloat
gamma.bf : Bigfloat
catalan.bf : Bigfloat
log2.bf : Bigfloat
```

Approximations of π, φ, γ, G and log(2).

Examples:

```
> (bf-precision 10)
> pi.bf
(bf #e3.1406)
> (bf-precision 179)
> pi.bf
(bf #e3.14159265358979323846264338327950288419716939937510582)
> phi.bf
bf #e1.61803398874989484820458683436563811772030917980576286)
> gamma.bf
(bf #e0.57721566490153286060651209008240243104215933593992359)
> catalan.bf
(bf #e0.91596559417721901505460351493238411077414937428167213)
> log2.bf
(bf #e0.69314718055994530941723212145817656807550013436025525)
```

```
-inf.bf : Bigfloat
-max.bf : Bigfloat
-min.bf : Bigfloat
-0.bf : Bigfloat
0.bf : Bigfloat
+min.bf : Bigfloat
+max.bf : Bigfloat
+inf.bf : Bigfloat
+nan.bf : Bigfloat
epsilon.bf : Bigfloat
```

Bigfloat constants corresponding to `-inf.0`, `-max.0` `-min.0`, `-0.0`, `0.0`, `+min.0`, `+max.0`, `+inf.0`, `+nan.0` and `epsilon.0`.

The constants `-inf.bf`, `-0.bf`, `0.bf`, `+inf.bf`, and `+nan.bf` have fixed precision.

```
-10.bf : Bigfloat
-9.bf : Bigfloat
-8.bf : Bigfloat
-7.bf : Bigfloat
-6.bf : Bigfloat
-5.bf : Bigfloat
-4.bf : Bigfloat
-3.bf : Bigfloat
-2.bf : Bigfloat
-1.bf : Bigfloat
1.bf : Bigfloat
```

```
2.bf : Bigfloat
3.bf : Bigfloat
4.bf : Bigfloat
5.bf : Bigfloat
6.bf : Bigfloat
7.bf : Bigfloat
8.bf : Bigfloat
9.bf : Bigfloat
10.bf : Bigfloat
```

More fixed-precision bigfloat constants.

5.7 Predicates

```
(bfzero? x) → Boolean
  x : Bigfloat
(bfpositive? x) → Boolean
  x : Bigfloat
(bfnegative? x) → Boolean
  x : Bigfloat
(bfinteger? x) → Boolean
  x : Bigfloat
(bfeven? x) → Boolean
  x : Bigfloat
(bfodd? x) → Boolean
  x : Bigfloat
(bfrational? x) → Boolean
  x : Bigfloat
(bfinfinite? x) → Boolean
  x : Bigfloat
(bfnan? x) → Boolean
  x : Bigfloat
```

Unary predicates corresponding to `zero?`, `positive?`, `negative?`, `integer?`, `even?`, `odd?`, `rational?`, `infinite?` and `nan?`.

```
(bf= x y) → Boolean
  x : Bigfloat
  y : Bigfloat
(bf> x y) → Boolean
  x : Bigfloat
  y : Bigfloat
(bf< x y) → Boolean
  x : Bigfloat
  y : Bigfloat
```

```
(bf>= x y) → Boolean
  x : Bigfloat
  y : Bigfloat
(bf<= x y) → Boolean
  x : Bigfloat
  y : Bigfloat
```

Standard comparison functions. As is usual, infinities are either greater or less than any other bigfloat, and every comparison returns `#f` when either argument is `+nan.bf`.

5.8 Rounding

```
(bftruncate x) → Bigfloat
  x : Bigfloat
(bffloor x) → Bigfloat
  x : Bigfloat
(bfceiling x) → Bigfloat
  x : Bigfloat
(bfround x) → Bigfloat
  x : Bigfloat
```

Like `truncate`, `floor`, `ceiling` and `round`, but for bigfloats.

Rounding is to the nearest integer, with ties broken by rounding to even.

Examples:

```
> (bfround (bf 1.5))
(bf 2)
> (bfround (bf 2.5))
(bf 2)
> (bfround (bf -1.5))
(bf -2)
> (bfround (bf -2.5))
(bf -2)
```

```
(bffrac x) → Bigfloat
  x : Bigfloat
```

Returns the fractional part of *x*, with the same sign as *x*.

```
(bfrint x) → Bigfloat
  x : Bigfloat
```

Rounds *x* to the nearest integer bigfloat, in the direction specified by `(bf-rounding-mode)`.

5.9 Mathematical Operations

```
(bfmax x ...) → Bigfloat
  x : Bigfloat
(bfmin x ...) → Bigfloat
  x : Bigfloat
```

Return the maximum and minimum of their arguments, respectively.

When given no arguments, `bfmin` returns `+inf.bf`, and `bfmax` returns `-inf.bf`.

```
(bf+ x ...) → Bigfloat
  x : Bigfloat
(bf* x ...) → Bigfloat
  x : Bigfloat
(bf- x y ...) → Bigfloat
  x : Bigfloat
  y : Bigfloat
(bf/ x y ...) → Bigfloat
  x : Bigfloat
  y : Bigfloat
(bfsqr x) → Bigfloat
  x : Bigfloat
(bfabs x) → Bigfloat
  x : Bigfloat
(bfsgn x) → Bigfloat
  x : Bigfloat
```

Standard arithmetic functions, corresponding to `+`, `*`, `-`, `/`, `sqr`, `abs` and `sgn`.

When `bf+` and `bf-` are given more than two arguments, they compute the answers in a way that incurs rounding error only once.

```
(bfsqrt x) → Bigfloat
  x : Bigfloat
(bf1/sqrt x) → Bigfloat
  x : Bigfloat
(bfcbrt x) → Bigfloat
  x : Bigfloat
```

Return the square root, *reciprocal* square root, and cube root of `x`.

```
(bfroot x n) → Bigfloat
  x : Bigfloat
  n : Integer
```

Returns the nth root of x. n must be a nonnegative fixnum.

```
(bfhypot x y) → Bigfloat
  x : Bigfloat
  y : Bigfloat
```

Computes `(bfsqrt (bf+ (bfsqr x) (bfsqr y)))` without uncessary overflow, incurring rounding error only once. See `flhypot` for an example using flonums.

```
(bflog x) → Bigfloat
  x : Bigfloat
(bflog2 x) → Bigfloat
  x : Bigfloat
(bflog10 x) → Bigfloat
  x : Bigfloat
```

Return the log of x in base e, 2 and 10.

```
(bfexp x) → Bigfloat
  x : Bigfloat
(bfexp2 x) → Bigfloat
  x : Bigfloat
(bfexp10 x) → Bigfloat
  x : Bigfloat
```

Return the exponential of x in base e, 2 and 10.

```
(bflog1p x) → Bigfloat
  x : Bigfloat
(bfexpm1 x) → Bigfloat
  x : Bigfloat
```

Like `(bflog (bf+ 1.bf x))` and `(bf- (bfexp x) 1.bf)`, but correct when x is near zero. See `fllog1p` for motivation and examples.

```
(bfexpt x y) → Bigfloat
  x : Bigfloat
  y : Bigfloat
```

Computes x^y. See `flexpt` and `expt`.

```
(bfsin x) → Bigfloat
  x : Bigfloat
(bfcos x) → Bigfloat
  x : Bigfloat
```

```
(bftan x) → Bigfloat
  x : Bigfloat
(bfasin x) → Bigfloat
  x : Bigfloat
(bfacos x) → Bigfloat
  x : Bigfloat
(bfatan x) → Bigfloat
  x : Bigfloat
(bfatan2 y x) → Bigfloat
  y : Bigfloat
  x : Bigfloat
```

Standard trigonometric functions and their inverses.

```
(bfsinh x) → Bigfloat
  x : Bigfloat
(bfcosh x) → Bigfloat
  x : Bigfloat
(bftanh x) → Bigfloat
  x : Bigfloat
(bfasinh x) → Bigfloat
  x : Bigfloat
(bfacosh x) → Bigfloat
  x : Bigfloat
(bfatanh x) → Bigfloat
  x : Bigfloat
```

Standard hyperbolic functions and their inverses.

```
(bfsec x) → Bigfloat
  x : Bigfloat
(bfcsc x) → Bigfloat
  x : Bigfloat
(bfcot x) → Bigfloat
  x : Bigfloat
```

Standard *reciprocal* trigonometric functions. MPFR does not implement their inverses.

```
(bfsech x) → Bigfloat
  x : Bigfloat
(bfcsch x) → Bigfloat
  x : Bigfloat
(bfcoth x) → Bigfloat
  x : Bigfloat
```

Standard *reciprocal* hyperbolic functions. MPFR does not implement their inverses.

```
(bfsin+cos x) → (Values Bigfloat Bigfloat)
  x : Bigfloat
```

Simultaneously computes the sine and cosine of `x`.

```
(bfsinh+cosh x) → (Values Bigfloat Bigfloat)
  x : Bigfloat
```

Simultaneously computes the hyperbolic sine and cosine of `x`.

```
(bffactorial x) → Bigfloat
  x : Integer
```

Returns the factorial of `x`.

```
(bfremainder n m) → Bigfloat
  n : Bigfloat
  m : Bigfloat
```

Returns the remainder analogous to the Racket `remainder` function. When `m` is equal to zero, `+nan.bf` is returned.

```
(bfgamma x) → Bigfloat
  x : Bigfloat
```

Computes the gamma function, a generalization of the factorial function.

```
(bflog-gamma x) → Bigfloat
  x : Bigfloat
(bflog-gamma/sign x) → (Values Bigfloat (U -1 1))
  x : Bigfloat
```

Computes the log-gamma function, or the log of the absolute value of the gamma function. `bflog-gamma/sign` additionally returns the sign of `(bfgamma x)`.

```
(bfpsi0 x) → Bigfloat
  x : Bigfloat
```

Computes the digamma function, the logarithmic derivative of the gamma function.

```
(bfeint x) → Bigfloat
  x : Bigfloat
```

Returns the exponential integral of `x`.

```
(bfli2 x) → Bigfloat
  x : Bigfloat
```

Returns the dilogarithm of `x`, or the polylogarithm of order 2.

```
(bfzeta x) → Bigfloat
  x : Bigfloat
```

Computes the Riemann zeta function.

```
(bferf x) → Bigfloat
  x : Bigfloat
(bferfc x) → Bigfloat
  x : Bigfloat
```

Compute the error function and complementary error function, respectively.

```
(bfbesj0 x) → Bigfloat
  x : Bigfloat
(bfbesj1 x) → Bigfloat
  x : Bigfloat
(bfbesj n x) → Bigfloat
  n : Integer
  x : Bigfloat
(bfbesy0 x) → Bigfloat
  x : Bigfloat
(bfbesy1 x) → Bigfloat
  x : Bigfloat
(bfbesy n x) → Bigfloat
  n : Integer
  x : Bigfloat
```

These compute Bessel functions.

A "`j`" in the name indicates that a function computes a Bessel function of the first kind. A "`y`" indicates the second kind.

The "`j`" or "`y`" is followed by the *order*: zero, one, or `n` (user-specified).

```
(bfagm x y) → bigfloat
  x : Bigfloat
  y : Bigfloat
```

Returns the arithmetic-geometric mean of `x` and `y`. Typically, this isn't directly useful, but it's used in some asymptotically fast algorithms such as the one that computes `bflog`.

5.10 Low-level Functions

```
(bigfloat->ordinal x) → Integer
  x : Bigfloat
(ordinal->bigfloat n) → Bigfloat
  n : Integer
(bigfloats-between x y) → Integer
  x : Bigfloat
  y : Bigfloat
(bfstep x n) → Bigfloat
  x : Bigfloat
  n : Integer
(bfnext x) → Bigfloat
  x : Bigfloat
(bfprev x) → Bigfloat
  x : Bigfloat
```

Like `flonum->ordinal`, `ordinal->flonum`, `flonums-between`, `flstep`, `flnext` and `flprev`, but for bigfloats.

The major difference is that these operate using `(bf-precision)` bits. Additionally, unlike other bigfloat functions, all of these convert their bigfloat arguments to `(bf-precision)` bits.

```
(bfshift x n) → Bigfloat
  x : Bigfloat
  n : Integer
```

Like `arithmetic-shift`, but for bigfloats. More precisely, this returns `(bf* x (bfexpt (bf 2) (bf n)))`, but is much faster.

```
(bfcanonicalize x) → Bigfloat
  x : Bigfloat
```

Bigfloats are canonicalized before hashing, to ensure that equality implies an equal hash.

If `x` is nonzero and rational, returns a new bigfloat with no more bits of precision than are necessary to encode `x` exactly, by removing all low-order zeros from the significand and adjusting the exponent.

For zero or non-rational `x`, returns `-inf.bf`, `-0.bf`, `0.bf`, `+inf.bf`, or `+nan.bf`, depending on the value of `x`.

Two nonzero, rational bigfloats are `equal?` if and only if their canonicalized significands and exponents are equal. Two zero or non-rational bigfloats are `equal?` if and only if their canonicalizations are `eq?`.

Canonicalizing bigfloats won't change answers computed from them.

Examples:

```
> (bf-precision 64)
> (define x (bf 1 -2))
> x
(bf #e0.25)
> (bfcanonicalize x)
(bf #e0.25)
> (bigfloat-precision x)
64
> (bigfloat-precision (bfcanonicalize x))
2
```

6 Arrays

Performance Warning: Indexing the elements of arrays created in untyped Racket is currently 25-50 times slower than doing the same in Typed Racket, due to the overhead of checking higher-order contracts. We are working on it.

For now, if you need speed, use the `typed/racket` language.

```
(require math/array)
```
package: `math-lib`

One of the most common ways to structure data is with an array: a rectangular grid of homogeneous, independent elements. But an array data type is usually absent from functional languages' libraries. This is probably because arrays are perceived as requiring users to operate on them using destructive updates, write loops that micromanage array elements, and in general, stray far from the declarative ideal.

Normally, they do. However, experience in Python, and more recently Data-Parallel Haskell, has shown that providing the right data types and a rich collection of whole-array operations allows working effectively with arrays in a functional, declarative style. As a bonus, doing so opens the possibility of parallelizing nearly every operation.

6.1 Quick Start

Arrays can be created from expressions denoting each element's value using the `array` macro:

```
> (array #[0 1 2 3 4])
- : (Array Byte)
(array #[0 1 2 3 4])
> (array #[#['first 'row 'data] #['second 'row 'data]])
- : (Array (U 'data 'first 'row 'second))
(array #[#['first 'row 'data] #['second 'row 'data]])
> (array "This array has zero axes and one element")
- : (Array String)
(array "This array has zero axes and one element")
```

They can also be created using `build-array` to specify a shape and procedure:

```
> (define arr
    (build-array #(4 5) (λ: ([js : Indexes])
                          (match-define (vector j0 j1) js)
                          (+ j0 j1))))
> arr
- : (Array Nonnegative-Fixnum)
(array #[#[0 1 2 3 4] #[1 2 3 4 5] #[2 3 4 5 6] #[3 4 5 6 7]])
```

Other ways to create arrays are to convert them from lists and vectors using `list->array`, `list*->array`, `vector->array` and `vector*->array`, and to generate them in a loop using `for/array:` and `for*/array:`.

Arrays can be indexed using `array-ref`, and settable arrays can be mutated using `array-set!`:

```
> (array-ref arr #(2 3))
- : Integer [more precisely: Nonnegative-Fixnum]
5
> (define brr (array->mutable-array arr))
> (array-set! brr #(2 3) -1000)
> brr
- : (Mutable-Array Integer)
(mutable-array #[#[0 1 2 3 4] #[1 2 3 4 5] #[2 3 4 -1000 6] #[3 4
5 6 7]])
```

However, both of these activities are discouraged in favor of functional, whole-array operations.

Arrays can be mapped over and otherwise operated on pointwise:

```
> (array-map (λ: ([n : Natural]) (* 2 n)) arr)
- : (Array Nonnegative-Integer)
(array #[#[0 2 4 6 8] #[2 4 6 8 10] #[4 6 8 10 12] #[6 8 10 12
14]])
> (array+ arr arr)
- : (Array Nonnegative-Integer)
(array #[#[0 2 4 6 8] #[2 4 6 8 10] #[4 6 8 10 12] #[6 8 10 12
14]])
```

When arrays have different shapes, they can often be broadcast, or stretched, to be the same shape before applying the pointwise operation:

```
> (array* arr (array 2))
- : (Array Nonnegative-Integer)
(array #[#[0 2 4 6 8] #[2 4 6 8 10] #[4 6 8 10 12] #[6 8 10 12
14]])
> (array* arr (array #[0 2 0 2 0]))
- : (Array Nonnegative-Integer)
(array #[#[0 2 0 6 0] #[0 4 0 8 0] #[0 6 0 10 0] #[0 8 0 12 0]])
```

By default, zero-dimensional arrays like `(array 2)` can be broadcast to any shape. See §6.3 “Broadcasting” for details.

Arrays can be sliced to yield sub-arrays, using a list of slice specifications that correspond to array axes. For example, keeping every row of `arr` and every even-numbered column:

```
> (array-slice-ref arr (list (::) (:: 0 5 2)))
- : (Array Nonnegative-Fixnum)
(array #[#[0 2 4] #[1 3 5] #[2 4 6] #[3 5 7]])
```

Here, `::` has semantics almost, but not quite, entirely unlike `in-range`. See §6.4 “Slicing” for details.

Functional code that uses whole-array operations often creates many short-lived, intermediate arrays whose elements are referred to only once. The overhead of allocating and filling storage for these arrays can be removed entirely by using nonstrict arrays, sometimes at the cost of making the code’s performance more difficult to reason about. Another bonus is that computations with nonstrict arrays have fewer synchronization points, meaning that they will be easier to parallelize as Racket’s support for parallel computation improves. See §6.5 “Nonstrict Arrays” for details.

6.2 Definitions

An array’s domain is determined by its *shape*, a vector of nonnegative integers such as `#(4 5)`, `#(10 1 5 8)` or `#()`. The shape’s length is the number of array dimensions, or *axes*. The shape’s contents are the length of each axis.

The product of the axis lengths is the array’s size. In particular, an array with shape `#()` has one element.

Indexes are a vector of nonnegative integers that identify a particular element. Indexes are in-bounds when there are the same number of them as axes, and each is *less than* its corresponding axis length.

An array’s contents are determined by its *procedure*, which returns an element when applied to in-bounds indexes. By default, most arrays’ procedures look up elements in memory. Others, such as those returned by `make-array`, return computed values.

A *pointwise* operation is one that operates on each array element independently, on each corresponding pair of elements from two arrays independently, or on a corresponding collection of elements from many arrays independently. This is usually done using `array-map`.

When a pointwise operation is performed on arrays with different shapes, the arrays are *broadcast* so that their shapes match. See §6.3 “Broadcasting” for details.

6.3 Broadcasting

It is often useful to apply a pointwise operation to two or more arrays in a many-to-one manner. Library support for this, which `math/array` provides, is called broadcasting.

Creating a 6×6 identity matrix:

```
> (define diag (diagonal-array 2 6 1 0))
> (array-shape diag)
- : Indexes
'#(6 6)
> diag
- : (Array (U One Zero))
(array
 #[#[1 0 0 0 0 0]
   #[0 1 0 0 0 0]
   #[0 0 1 0 0 0]
   #[0 0 0 1 0 0]
   #[0 0 0 0 1 0]
   #[0 0 0 0 0 1]])
```

Multiplying each element by `10`:

```
> (array-shape (array 10))
- : Indexes
'#()
> (array* diag (array 10))
- : (Array Index)
(array
 #[#[10 0 0 0 0 0]
   #[0 10 0 0 0 0]
   #[0 0 10 0 0 0]
   #[0 0 0 10 0 0]
   #[0 0 0 0 10 0]
   #[0 0 0 0 0 10]])
```

Adding `(array #[0 1 2 3 4 5])` pointwise to every row:

```
> (array+ (array* diag (array 10))
          (array #[0 1 2 3 4 5]))
- : (Array Nonnegative-Fixnum)
(array
 #[#[10 1 2 3 4 5]
   #[0 11 2 3 4 5]
   #[0 1 12 3 4 5]
   #[0 1 2 13 4 5]
   #[0 1 2 3 14 5]
   #[0 1 2 3 4 15]])
```

6.3.1 Broadcasting Rules

Suppose we have two array shapes `ds = (vector d0 d1 ...)` and `es = (vector e0 e1 ...)`. Broadcasting proceeds as follows:

1. The shorter shape is padded on the left with `1` until it is the same length as the longer shape.
2. For each axis `k`, `dk` and `ek` are compared. If `dk = ek`, the result axis is `dk`; if one axis is length `1`, the result axis is the length of the other; otherwise fail.
3. Both arrays' axes are stretched by (conceptually) copying the rows of axes with length `1`.

Example: Suppose we have an array `drr` with shape `ds = #(4 1 3)` and another array `err` with shape `es = #(3 3)`. Following the rules:

1. `es` is padded to get `#(1 3 3)`.
2. The result axis is derived from `#(4 1 3)` and `#(1 3 3)` to get `#(4 3 3)`.
3. `drr`'s second axis is stretched to length `3`, and `err`'s new first axis (which is length `1` by rule 1) is stretched to length `4`.

The same example, but more concrete:

```
> (define drr
    (array #[#[#["00" "01" "02"]]
             #[#["10" "11" "12"]]
             #[#["20" "21" "22"]]
             #[#["30" "31" "32"]]]))
> (array-shape drr)
- : Indexes
'#(4 1 3)
> (define err
    (array #[#["aa" "ab" "ac"]
             #["ba" "bb" "bc"]
             #["ca" "cb" "cc"]]))
> (array-shape err)
- : Indexes
'#(3 3)
> (define drr+err (array-map string-append drr err))
> (array-shape drr+err)
- : Indexes
```

```
'#(4 3 3)
> drr+err
- : (Array String)
(array
 #[#[#["00aa" "01ab" "02ac"]
     #["00ba" "01bb" "02bc"]
     #["00ca" "01cb" "02cc"]]
   #[#["10aa" "11ab" "12ac"]
     #["10ba" "11bb" "12bc"]
     #["10ca" "11cb" "12cc"]]
   #[#["20aa" "21ab" "22ac"]
     #["20ba" "21bb" "22bc"]
     #["20ca" "21cb" "22cc"]]
   #[#["30aa" "31ab" "32ac"]
     #["30ba" "31bb" "32bc"]
     #["30ca" "31cb" "32cc"]]])
```

Notice how the row `#["00" "01" "02"]` in `drr` is repeated in the result because `drr`'s second axis was stretched during broadcasting. Also, the column `#[#["aa"] #["ba"] #["ca"]]` in `err` is repeated because `err`'s first axis was stretched.

For the above example, `array-map` does this before operating on `drr` and `err`:

```
> (define ds (array-shape-broadcast (list (array-shape drr)
                                          (array-shape err))))
> ds
- : Indexes
'#(4 3 3)
> (array-broadcast drr ds)
- : (Array String)
(array
 #[#[#["00" "01" "02"]
     #["00" "01" "02"]
     #["00" "01" "02"]]
   #[#["10" "11" "12"]
     #["10" "11" "12"]
     #["10" "11" "12"]]
   #[#["20" "21" "22"]
     #["20" "21" "22"]
     #["20" "21" "22"]]
   #[#["30" "31" "32"]
     #["30" "31" "32"]
     #["30" "31" "32"]]])
> (array-broadcast err ds)
- : (Array String)
(array
```

```
#[#[#["aa" "ab" "ac"]
    #["ba" "bb" "bc"]
    #["ca" "cb" "cc"]]
  #[#["aa" "ab" "ac"]
    #["ba" "bb" "bc"]
    #["ca" "cb" "cc"]]
  #[#["aa" "ab" "ac"]
    #["ba" "bb" "bc"]
    #["ca" "cb" "cc"]]
  #[#["aa" "ab" "ac"]
    #["ba" "bb" "bc"]
    #["ca" "cb" "cc"]]])
```

6.3.2 Broadcasting Control

The parameter `array-broadcasting` controls how pointwise operations broadcast arrays. Its default value is `#t`, which means that broadcasting proceeds as described in §6.3.1 "Broadcasting Rules". Another possible value is `#f`, which allows pointwise operations to succeed only if array shapes match exactly:

```
> (parameterize ([array-broadcasting #f])
    (array* (index-array #(3 3)) (array 10)))
array-shape-broadcast: incompatible array shapes
(array-broadcasting #f): '#(3 3), '#()
```

Another option is R-style permissive broadcasting, which allows pointwise operations to *always* succeed, by repeating shorter axes' rows instead of repeating just singleton axes' rows:

```
> (define arr10 (array-map number->string (index-array #(10))))
> (define arr3 (array-map number->string (index-array #(3))))
> arr10
- : (Array String)
(array #["0" "1" "2" "3" "4" "5" "6" "7" "8" "9"])
> arr3
- : (Array String)
(array #["0" "1" "2"])
> (array-map string-append arr10 (array #["+" "-"]) arr3)
array-shape-broadcast: incompatible array shapes
(array-broadcasting #t): '#(10), '#(2), '#(3)
> (parameterize ([array-broadcasting 'permissive])
    (array-map string-append arr10 (array #["+" "-"]) arr3))
- : (Array String)
(array #["0+0" "1-1" "2+2" "3-0" "4+1" "5-2" "6+0" "7-1" "8+2"
"9-0"])
```

Notice that `(array #["+" "-"])` was repeated five times, and that `arr3` was repeated three full times and once partially.

6.4 Slicing

One common array transformation is *slicing*: extracting sub-arrays by picking rows from each axis independently.

Slicing is done by applying `array-slice-ref` or `array-slice-set!` to an array and a list of *slice specifications* corresponding to array axes. There are five types of slice specification:

- `(Sequenceof Integer)`: pick rows from an axis by index.
- `Slice`: pick rows from an axis as with an `in-range` sequence.
- `Slice-Dots`: preserve remaining adjacent axes
- `Integer`: remove an axis by replacing it with one of its rows.
- `Slice-New-Axis`: insert an axis of a given length.

Create `Slice` objects using `::` and `Slice-New-Axis` objects using `::new`. There is only one `Slice-Dots` object, namely `::...`.

When slicing an array with `n` axes, unless a list of slice specifications contains `::...`, it must contain exactly `n` slice specifications.

The remainder of this section uses the following example array:

```
> (define arr
    (build-array
     #(2 3 4)
     (λ: ([js : Indexes])
       (string-append* (map number->string (vector->list js))))))
> arr
- : (Array String)
(array
 #[#[#["000" "001" "002" "003"]
     #["010" "011" "012" "013"]
     #["020" "021" "022" "023"]]
   #[#["100" "101" "102" "103"]
     #["110" "111" "112" "113"]
     #["120" "121" "122" "123"]]])
```

6.4.1 `(Sequenceof Integer)`: pick rows

Using a sequence of integers as a slice specification picks rows from the corresponding axis. For example, we might use lists of integers to pick *every* row from every axis:

```
> (array-slice-ref arr (list '(0 1) '(0 1 2) '(0 1 2 3)))
- : (Array String)
(array
 #[#[#["000" "001" "002" "003"]
     #["010" "011" "012" "013"]
     #["020" "021" "022" "023"]]
   #[#["100" "101" "102" "103"]
     #["110" "111" "112" "113"]
     #["120" "121" "122" "123"]]])
```

This simply copies the array.

More usefully, we can use sequences to swap rows on the same axis:

```
> (array-slice-ref arr (list '(1 0) '(0 1 2) '(0 1 2 3)))
- : (Array String)
(array
 #[#[#["100" "101" "102" "103"]
     #["110" "111" "112" "113"]
     #["120" "121" "122" "123"]]
   #[#["000" "001" "002" "003"]
     #["010" "011" "012" "013"]
     #["020" "021" "022" "023"]]])
```

We can also remove rows:

```
> (array-slice-ref arr (list '(0 1) '(0 2) '(0 2)))
- : (Array String)
(array #[#[#["000" "002"] #["020" "022"]] #[#["100" "102"] #["120"
"122"]]])
> (array-slice-ref arr (list '(0 1) '(0 1 2) '()))
- : (Array String)
(array #[#[#[] #[] #[]] #[#[] #[] #[]]])
```

Or duplicate rows:

```
> (array-slice-ref arr (list '(0 1) '(0 1 2) '(0 0 1 2 2 3)))
- : (Array String)
(array
 #[#[#["000" "000" "001" "002" "002" "003"]
```

```
    #["010" "010" "011" "012" "012" "013"]
    #["020" "020" "021" "022" "022" "023"]]
  #[#["100" "100" "101" "102" "102" "103"]
    #["110" "110" "111" "112" "112" "113"]
    #["120" "120" "121" "122" "122" "123"]]])
```

However, a sequence slice specification cannot alter the number of axes.

Using sequence constructors like `in-range`, we can pick every even-indexed row in an axis:

```
> (array-slice-ref arr (list '(1 0) '(0 1 2) (in-range 0 4 2)))
- : (Array String)
(array
 #[#[#["100" "102"]
     #["110" "112"]
     #["120" "122"]]
   #[#["000" "002"]
     #["010" "012"]
     #["020" "022"]]])
```

We could also use `in-range` to pick every row instead of enumerating their indexes in a list, but that would require another kind of tedium:

```
> (define ds (array-shape arr))
> (array-slice-ref arr (list (in-range (vector-ref ds 0))
                             (in-range (vector-ref ds 1))
                             (in-range (vector-ref ds 2))))
- : (Array String)
(array
 #[#[#["000" "001" "002" "003"]
     #["010" "011" "012" "013"]
     #["020" "021" "022" "023"]]
   #[#["100" "101" "102" "103"]
     #["110" "111" "112" "113"]
     #["120" "121" "122" "123"]]])
```

The situation calls for an `in-range`-like slice specification that is aware of the lengths of the axes it is applied to.

6.4.2 `Slice`: pick rows in a length-aware way

As a slice specification, a `Slice` object acts like the sequence object returned by `in-range`, but either `start` or `end` may be `#f`.

If `start` is `#f`, it is interpreted as the first valid axis index in the direction of `step`. If `end` is `#f`, it is interpreted as the last valid axis index in the direction of `step`.

Possibly the most common slice is `(::)`, equivalent to `(:: #f #f 1)`. With a positive `step = 1`, `start` is interpreted as `0` and `end` as the length of the axis. Thus, `(::)` picks all rows from any axis:

```
> (array-slice-ref arr (list (::) (::) (::)))
- : (Array String)
(array
 #[#[#["000" "001" "002" "003"]
     #["010" "011" "012" "013"]
     #["020" "021" "022" "023"]]
   #[#["100" "101" "102" "103"]
     #["110" "111" "112" "113"]
     #["120" "121" "122" "123"]]])
```

The slice `(:: #f #f -1)` reverses an axis:

```
> (array-slice-ref arr (list (::) (::) (:: #f #f -1)))
- : (Array String)
(array
 #[#[#["003" "002" "001" "000"]
     #["013" "012" "011" "010"]
     #["023" "022" "021" "020"]]
   #[#["103" "102" "101" "100"]
     #["113" "112" "111" "110"]
     #["123" "122" "121" "120"]]])
```

The slice `(:: 2 #f 1)` picks every row starting from index `2`:

```
> (array-slice-ref arr (list (::) (::) (:: 2 #f 1)))
- : (Array String)
(array
 #[#[#["002" "003"]
     #["012" "013"]
     #["022" "023"]]
   #[#["102" "103"]
     #["112" "113"]
     #["122" "123"]]])
```

The slice `(:: 1 #f 2)` picks every odd-indexed row:

```
> (array-slice-ref arr (list (::) (::) (:: 1 #f 2)))
- : (Array String)
```

```
(array
 #[#[#["001" "003"]
     #["011" "013"]
     #["021" "023"]]
   #[#["101" "103"]
     #["111" "113"]
     #["121" "123"]]])
```

Notice that every example starts with two `(::)`. In fact, slicing only one axis is so common that there is a slice specification object that represents any number of `(::)`.

6.4.3 `Slice-Dots`: preserve remaining axes

As a slice specification, a `Slice-Dots` object represents any number of leftover, adjacent axes, and preserves them all.

For example, picking every odd-indexed row of the last axis can be done by

```
> (array-slice-ref arr (list ::... (:: 1 #f 2)))
- : (Array String)
(array
 #[#[#["001" "003"]
     #["011" "013"]
     #["021" "023"]]
   #[#["101" "103"]
     #["111" "113"]
     #["121" "123"]]])
```

For `arr` specifically, `::...` represents two `(::)`.

Slicing only the first axis while preserving the rest can be done by

```
> (array-slice-ref arr (list '(0) ::...))
- : (Array String)
(array
 #[#[#["000" "001" "002" "003"]
     #["010" "011" "012" "013"]
     #["020" "021" "022" "023"]]])
```

If more than one `::...` appears in the list, only the first is expanded:

```
> (array-slice-ref arr (list ::... '(1) ::...))
- : (Array String)
```

```
(array #[#[#["001"] #["011"] #["021"]] #[#["101"] #["111"]
#["121"]]])
> (array-slice-ref arr (list ::... '(1)))
- : (Array String)
(array #[#[#["001"] #["011"] #["021"]] #[#["101"] #["111"]
#["121"]]])
```

If there are no leftover axes, `::...` does nothing when placed in any position:

```
> (array-slice-ref arr (list ::... '(1) '(1) '(1)))
- : (Array String)
(array #[#[#["111"]]])
> (array-slice-ref arr (list '(1) ::... '(1) '(1)))
- : (Array String)
(array #[#[#["111"]]])
> (array-slice-ref arr (list '(1) '(1) ::... '(1)))
- : (Array String)
(array #[#[#["111"]]])
> (array-slice-ref arr (list '(1) '(1) '(1) ::...))
- : (Array String)
(array #[#[#["111"]]])
```

6.4.4 `Integer`: remove an axis

All of the slice specifications so far preserve the dimensions of the array. Removing an axis can be done by using an integer as a slice specification.

This example removes the first axis by collapsing it to its first row:

```
> (array-slice-ref arr (list 0 ::...))
- : (Array String)
(array
 #[#["000" "001" "002" "003"]
   #["010" "011" "012" "013"]
   #["020" "021" "022" "023"]])
```

Removing the second axis by collapsing it to the row with index `1`:

```
> (array-slice-ref arr (list (::) 1 ::...))
- : (Array String)
(array #[#["010" "011" "012" "013"] #["110" "111" "112" "113"]])
```

Removing the second-to-last axis (which for `arr` is the same as the second):

```
> (array-slice-ref arr (list ::... 1 (::)))
- : (Array String)
(array #[#["010" "011" "012" "013"] #["110" "111" "112" "113"]])
```

All of these examples can be done using `array-axis-ref`. However, removing an axis relative to the dimension of the array (e.g. the second-to-last axis) is easier to do using `array-slice-ref`, and it is sometimes convenient to combine axis removal with other slice operations.

6.4.5 `Slice-New-Axis`: add an axis

As a slice specification, `(::new dk)` inserts `dk` into the resulting array's shape, in the corresponding axis position. The new axis has length `dk`, which must be nonnegative.

For example, we might conceptually wrap another `#[]` around an array's data:

```
> (array-slice-ref arr (list (::new) ::...))
- : (Array String)
(array
 #[#[#[#["000" "001" "002" "003"]
       #["010" "011" "012" "013"]
       #["020" "021" "022" "023"]]
     #[#["100" "101" "102" "103"]
       #["110" "111" "112" "113"]
       #["120" "121" "122" "123"]]]])
```

Or duplicate the array twice, within two new outer rows:

```
> (array-slice-ref arr (list (::new 2) ::...))
- : (Array String)
(array
 #[#[#[#["000" "001" "002" "003"]
       #["010" "011" "012" "013"]
       #["020" "021" "022" "023"]]
     #[#["100" "101" "102" "103"]
       #["110" "111" "112" "113"]
       #["120" "121" "122" "123"]]]
   #[#[#["000" "001" "002" "003"]
       #["010" "011" "012" "013"]
       #["020" "021" "022" "023"]]
     #[#["100" "101" "102" "103"]
       #["110" "111" "112" "113"]
       #["120" "121" "122" "123"]]]])
```

Of course, `dk = 0` is a valid new axis length, but is usually not very useful:

```
> (array-slice-ref arr (list (::) (::new 0) ::...))
- : (Array String)
(array #[#[] #[]])
```

Inserting axes can also be done using `array-axis-insert`.

6.5 Nonstrict Arrays

With few exceptions, by default, the functions exported by `math/array` return *strict* arrays, which are arrays whose procedures compute elements by looking them up in a vector.

This conservative default often wastes time and space. In functional code that operates on arrays, the elements in most intermediate arrays are referred to exactly once, so allocating and filling storage for them should be unnecessary. For example, consider the following array:

```
> (define (make-hellos)
    (array-map string-append
               (array-map string-append
                          (array #["Hello " "Hallo " "Jó napot "])
                          (array #["Ada" "Edsger" "John"]))
               (make-array #(3) "!")))
> (define arr (make-hellos))
> (array-strict? arr)
- : Boolean
#t
> arr
- : (Array String)
(array #["Hello Ada!" "Hallo Edsger!" "Jó napot John!"])
```

By default, the result of the inner `array-map` has storage allocated for it and filled with strings such as `"Hello Ada"`, even though its storage will be thrown away at the next garbage collection cycle.

An additional concern becomes even more important as Racket's support for parallel computation improves. Allocating storage for intermediate arrays is a synchronization point in long computations, which divides them into many short computations, making them difficult to parallelize.

A solution is to construct *nonstrict* arrays*, which are arrays whose procedures can do more than simply look up elements. Setting the parameter `array-strictness` to `#f` causes almost all `math/array` functions to return nonstrict arrays:

* **Regular, shape-polymorphic, parallel arrays in Haskell**, Gabriele Keller, Manuel Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. ICFP 2010. (PDF)

```
> (define arr (parameterize ([array-strictness #f])
                (make-hellos)))
> (array-strict? arr)
- : Boolean
#f
> arr
- : (Array String)
(array #["Hello Ada!" "Hallo Edsger!" "Jó napot John!"])
```

In `arr`, the first element is the *computation* `(string-append (string-append "Hello " "Ada") "!")`, not the value `"Hello Ada!"`. The value `"Hello Ada!"` is recomputed every time the first element is referred to.

To use nonstrict arrays effectively, think of every array as if it were the array's procedure itself. In other words,

> **An *array* is just a function with a finite, rectangular domain.**

Some arrays are mutable, some are lazy, some are strict, some are sparse, and most do not even allocate contiguous space to store their elements. All are functions that can be applied to indexes to retrieve elements.

The two most common kinds of operations, mapping over and transforming arrays, are compositions. Mapping `f` over array `arr` is nothing more than composing `f` with `arr`'s procedure. Transforming `arr` using `g`, a function from new indexes to old indexes, is nothing more than composing `arr`'s procedure with `g`.

6.5.1 Caching Nonstrict Elements

Nonstrict arrays are not lazy. Very few nonstrict arrays cache computed elements, but like functions, recompute them every time they are referred to. Unlike functions, they can have every element computed and cached at once, by making them strict.

To compute and store an array's elements, use `array-strict!` or `array-strict`:

```
> (array-strict? arr)
- : Boolean
#f
> (array-strict! arr)
> (array-strict? arr)
- : Boolean
#t
> (array-strict arr)
- : (Array String)
```

```
(array #["Hello Ada!" "Hallo Edsger!" "Jó napot John!"])
```

If the array is already strict, as in the last example above, `array-strict!` and `array-strict` do nothing.

To make a strict *copy* of an array without making the original array strict, use `array->mutable-array`.

6.5.2 Performance Considerations

One downside to nonstrict arrays is that it is more difficult to reason about the performance of operations on them. Another is that the user must decide which arrays to make strict. Fortunately, there is a simple rule of thumb:

> **Make arrays strict when you must refer to most of their elements more than once or twice.**

Having to name an array is a good indicator that it should be strict. In the following example, which computes `(+ (expt x x) (expt x x))` for `x` from `0` to `2499`, each element in `xrr` is computed twice whenever its corresponding element in `res` is referred to:

```
(define xrr (array-map expt
                       (index-array #(50 50))
                       (index-array #(50 50))))
(define res (array+ xrr xrr))
```

Having to name `xrr` means we should make it strict:

```
(define xrr (array-strict
             (array-map expt
                        (index-array #(50 50))
                        (index-array #(50 50)))))
(define res (array+ xrr xrr))
```

Doing so halves the time it takes to compute `res`'s elements.

When returning an array from a function, return nonstrict arrays as they are, to allow the caller to decide whether the result should be strict.

When writing library functions that may be called with either `(array-strictness #t)` or `(array-strictness #f)`, operate on nonstrict arrays and wrap the result with `array-default-strict` to return what the user is expecting. For example, if `make-hellos` is a library function, it should be written as

```
(define (make-hellos)
  (array-default-strict
   (parameterize ([array-strictness #f])
     (array-map string-append
                (array-map string-append
                           (array #["Hello " "Hallo " "Jó napot
"])
                           (array #["Ada" "Edsger" "John"]))
                (make-array #(3) "!")))))
```

If you cannot determine whether to make arrays strict, or are using arrays for so-called "dynamic programming," you can make them lazy using `array-lazy`.

6.6 Types, Predicates and Accessors

`(Array A)`

The parent array type. Its type parameter is the type of the array's elements.

The polymorphic `Array` type is *covariant*, meaning that `(Array A)` is a subtype of `(Array B)` if `A` is a subtype of `B`:

```
> (define arr (array #[1 2 3 4 5]))
> arr
- : (Array Positive-Byte)
(array #[1 2 3 4 5])
> (ann arr (Array Real))
- : (Array Real)
(array #[1 2 3 4 5])
> (ann arr (Array Any))
- : (Array Any)
(array #[1 2 3 4 5])
```

Because subtyping is transitive, the `(Array A)` in the preceeding subtyping rule can be replaced with any of `(Array A)`'s subtypes, including descendant types of `Array`. For example, `(Mutable-Array A)` is a subtype of `(Array B)` if `A` is a subtype of `B`:

```
> (define arr (mutable-array #[1 2 3 4 5]))
> arr
- : (Mutable-Array Integer)
(mutable-array #[1 2 3 4 5])
> (ann arr (Array Real))
- : (Array Real)
```

```
(mutable-array #[1 2 3 4 5])
> (ann arr (Array Any))
- : (Array Any)
(mutable-array #[1 2 3 4 5])
```

(Settable-Array *A*)

The parent type of arrays whose elements can be mutated. Functions like `array-set!` and `array-slice-set!` accept arguments of this type. Examples of subtypes are `Mutable-Array`, `FlArray` and `FCArray`.

This type is *invariant*, meaning that `(Settable-Array A)` is **not** a subtype of `(Settable-Array B)` if `A` and `B` are different types, even if `A` is a subtype of `B`:

```
> (define arr (mutable-array #[1 2 3 4 5]))
> arr
- : (Mutable-Array Integer)
(mutable-array #[1 2 3 4 5])
> (ann arr (Settable-Array Integer))
- : (Settable-Array Integer)
(mutable-array #[1 2 3 4 5])
> (ann arr (Settable-Array Real))
eval:93:0: Type Checker: type mismatch
  expected: (Settable-Array Real)
  given: (Mutable-Array Integer)
  in: Real
```

(Mutable-Array *A*)

The type of mutable arrays. Its type parameter is the type of the array's elements.

Arrays of this type store their elements in a `(Vectorof A)`:

```
> (define arr (mutable-array #[#[1 2] #[3 4]]))
> (vector-set! (mutable-array-data arr) 0 -10)
> arr
- : (Mutable-Array Integer)
(mutable-array #[#[-10 2] #[3 4]])
```

Mutable arrays are always strict.

Indexes

The type of array shapes and array indexes *produced* by `math/array` functions. Defined as `(Vectorof Index)`.

Example:

```
> (array-shape (array #[#[#[0]]]))
- : Indexes
'#(1 1 1)
```

In-Indexes

The type of array shapes and array indexes *accepted* by `math/array` functions. Defined as `(U Indexes (Vectorof Integer))`.

Examples:

```
> (define ds #(3 2))
> ds
- : (Immutable-Vector Positive-Byte Positive-Byte)
'#(3 2)
> (make-array ds (void))
- : (Array Void)
(array #[#[#<void> #<void>] #[#<void> #<void>] #[#<void>
#<void>]])
```

This makes indexes-accepting functions easier to use, because it is easier to convince Typed Racket that a vector contains `Integer` elements than that a vector contains `Index` elements.

`In-Indexes` is not defined as `(Vectorof Integer)` because mutable container types like `Vector` and `Vectorof` are invariant. In particular, `(Vectorof Index)` is not a subtype of `(Vectorof Integer)`:

```
> (define js ((inst vector Index) 3 4 5))
> js
- : (Mutable-Vectorof Index)
'#(3 4 5)
> (ann js (Vectorof Integer))
eval:103:0: Type Checker: type mismatch
  expected: (Vectorof Integer)
  given: (Mutable-Vectorof Index)
  in: Integer
> (ann js In-Indexes)
- : In-Indexes
'#(3 4 5)
```

```
(array? v) → Boolean
  v : Any
(settable-array? v) → Boolean
  v : Any
(mutable-array? v) → Boolean
  v : Any
```

Predicates for the types `Array`, `Settable-Array`, and `Mutable-Array`.

Because `Settable-Array` and its descendants are invariant, `settable-array?` and its descendants' predicates are generally not useful in occurrence typing. For example, if we know we have an `Array` but would like to treat it differently if it happens to be a `Mutable-Array`, we are basically out of luck:

```
> (: maybe-array-data (All (A) ((Array A) -> (U #f (Vectorof A)))))
> (define (maybe-array-data arr)
    (cond [(mutable-array? arr)  (mutable-array-data arr)]
          [else  #f]))
eval:106:0: Type Checker: Polymorphic function
 `mutable-array-data' could not be applied to arguments:
Argument 1:
  Expected: (Mutable-Array A)
  Given:      (Struct Mutable-Array)

  in: #f
```

In general, predicates with a `Struct` filter do not give conditional branches access to a struct's accessors. Because `Settable-Array` and its descendants are invariant, their predicates have `Struct` filters:

```
> array?
- : (-> Any Boolean : (Array Any))
#<procedure:Array?>
> settable-array?
- : (-> Any Boolean : (Struct Settable-Array))
#<procedure:Settable-Array?>
> mutable-array?
- : (-> Any Boolean : (Struct Mutable-Array))
#<procedure:Mutable-Array?>
```

```
(array-shape arr) → Indexes
  arr : (Array A)
```

Returns *arr*'s shape, a vector of indexes that contains the lengths of *arr*'s axes.

Examples:

```
> (array-shape (array 0))
- : Indexes
'#()
> (array-shape (array #[0 1]))
- : Indexes
```

```
'#(2)
> (array-shape (array #[#[0 1]]))
- : Indexes
'#(1 2)
> (array-shape (array #[]))
- : Indexes
'#(0)
```

```
(array-size arr) → Index
  arr : (Array A)
```

Returns the number of elements in *arr*, which is the product of its axis lengths.

Examples:

```
> (array-size (array 0))
- : Integer [more precisely: Index]
1
> (array-size (array #[0 1]))
- : Integer [more precisely: Index]
2
> (array-size (array #[#[0 1]]))
- : Integer [more precisely: Index]
2
> (array-size (array #[]))
- : Integer [more precisely: Index]
0
```

```
(array-dims arr) → Index
  arr : (Array A)
```

Returns the number of *arr*'s dimensions. Equivalent to `(vector-length (array-shape arr))`.

```
(mutable-array-data arr) → (Vectorof A)
  arr : (Mutable-Array A)
```

Returns the vector of data that *arr* contains.

6.7 Construction

```
(array #[#[...] ...] maybe-type-ann)

maybe-type-ann =
               | : type
```

Creates an `Array` from nested rows of expressions.

The vector syntax `#[...]` delimits rows. These may be nested to any depth, and must have a rectangular shape. Using square parentheses is not required, but is encouraged to help visually distinguish array contents from array indexes and other vectors. (See the examples for `indexes-array` for an illustration.)

Examples:

```
> (array 0)
- : (Array Zero)
(array 0)
> (array #[0 1 2 3])
- : (Array Byte)
(array #[0 1 2 3])
> (array #[#[1 2 3] #[4 5 6]])
- : (Array Positive-Byte)
(array #[#[1 2 3] #[4 5 6]])
> (array #[#[1 2 3] #[4 5]])
eval:121:0: array: expected rectangular data
  at: #(#(1 2 3) #(4 5))
  in: (array/syntax array list unsafe-list->array #(#(1 2 3)
#(4 5)))
```

As with the `list` constructor, the type chosen for the array is the narrowest type all the elements can have. Unlike `list`, because `array` is syntax, instantiating `array` with the desired element type is a syntax error:

```
> (list 1 2 3)
- : (List One Positive-Byte Positive-Byte)
'(1 2 3)
> (array #[1 2 3])
- : (Array Positive-Byte)
(array #[1 2 3])
> ((inst list Real) 1 2 3)
- : (Listof Real)
'(1 2 3)
> ((inst array Real) #[1 2 3])
eval:125:0: array: not allowed as an expression
  in: array
```

There are two easy ways to annotate the element type:

```
> (array #[1 2 3] : Real)
- : (Array Real)
(array #[1 2 3])
```

```
> (ann (array #[1 2 3]) (Array Real))
- : (Array Real)
(array #[1 2 3])
```

Annotating should rarely be necessary because the `Array` type is covariant.

Normally, the datums within literal vectors are implicitly quoted. However, when used within the `array` form, the datums must be explicitly quoted.

```
> #(this is okay)
- : (Immutable-Vector 'this 'is 'okay)
'#(this is okay)
> (array #[not okay])
eval:129:0: Type Checker: missing type for top-level
identifier;
 either undefined or missing a type annotation
  identifier: okay
  in: #(not okay)
> (array #['this 'is 'okay])
- : (Array (U 'is 'okay 'this))
(array #['this 'is 'okay])
> (array #['#(an) '#(array) '#(of) '#(vectors)])
- : (Array
     (U (Immutable-Vector 'an)
        (Immutable-Vector 'array)
        (Immutable-Vector 'of)
        (Immutable-Vector 'vectors)))
(array #['#(an) '#(array) '#(of) '#(vectors)])
```

Arrays returned by `array` are strict. Another way to create immutable, strict arrays from literal data is to use `list->array`.

```
(mutable-array #[#[...] ...] maybe-type-ann)

maybe-type-ann =
               | : type
```

Creates a `Mutable-Array` from nested rows of expressions.

The semantics are almost identical to `array`'s, except the result is mutable:

```
> (define arr (mutable-array #[0 1 2 3]))
> arr
- : (Mutable-Array Integer)
(mutable-array #[0 1 2 3])
```

```
> (array-set! arr #(0) 10)
> arr
- : (Mutable-Array Integer)
(mutable-array #[10 1 2 3])
```

Because mutable arrays are invariant, this form additionally accepts a type annotation for the array's elements:

```
> (define arr (mutable-array #[0 1 2 3] : Real))
> arr
- : (Mutable-Array Real)
(mutable-array #[0 1 2 3])
> (array-set! arr #(0) 10.0)
> arr
- : (Mutable-Array Real)
(mutable-array #[10.0 1 2 3])
```

Another way to create mutable arrays from literal data is to use `vector->array`.

```
(make-array ds value) → (Array A)
  ds : In-Indexes
  value : A
```

Returns an array with shape *ds*, with every element's value as *value*. Analogous to `make-vector`.

Examples:

```
> (make-array #() 5)
- : (Array Positive-Byte)
(array 5)
> (make-array #(1 2) 'sym)
- : (Array 'sym)
(array #[#['sym 'sym]])
> (make-array #(4 0 2) "Invisible")
- : (Array String)
(array #[#[] #[] #[] #[]])
```

The arrays returned by `make-array` do not allocate storage for their elements and are strict.

```
(build-array ds proc) → (Array A)
  ds : In-Indexes
  proc : (Indexes -> A)
```

Returns an array with shape *ds* and procedure *proc*. Analogous to `build-vector`.

```
(array->mutable-array arr) → (Mutable-Array A)
  arr : (Array A)
```

Returns a mutable array with the same elements as *arr*. The result is a copy of *arr*, even when *arr* is mutable.

```
(mutable-array-copy arr) → (Mutable-Array A)
  arr : (Mutable-Array A)
```

Like `(array->mutable-array arr)`, but restricted to mutable arrays. It is also faster.

```
(indexes-array ds) → (Array Indexes)
  ds : In-Indexes
```

Returns an array with shape *ds*, with each element set to its position in the array.

Examples:

```
> (indexes-array #())
- : (Array Indexes)
(array '#())
> (indexes-array #(4))
- : (Array Indexes)
(array #['#(0) '#(1) '#(2) '#(3)])
> (indexes-array #(2 3))
- : (Array Indexes)
(array #[#['#(0 0) '#(0 1) '#(0 2)] #['#(1 0) '#(1 1) '#(1 2)]])
> (indexes-array #(4 0 2))
- : (Array Indexes)
(array #[#[] #[] #[] #[]])
```

The resulting array does not allocate storage for its return value's elements, and is strict. (It is essentially the identity function for the domain *ds*.)

```
(index-array ds) → (Array Index)
  ds : In-Indexes
```

Returns an array with shape *ds*, with each element set to its row-major index in the array.

Examples:

```
> (index-array #(2 3))
- : (Array Index)
(array #[#[0 1 2] #[3 4 5]])
> (array-flatten (index-array #(2 3)))
- : (Array Index)
(array #[0 1 2 3 4 5])
```

As with `indexes-array`, the result does not allocate storage for its elements, and is strict.

```
(axis-index-array ds axis) → (Array Index)
  ds : In-Indexes
  axis : Integer
```

Returns an array with shape *ds*, with each element set to its position in axis *axis*. The axis number *axis* must be nonnegative and less than the number of axes (the length of *ds*).

Examples:

```
> (axis-index-array #(3 3) 0)
- : (Array Index)
(array #[#[0 0 0] #[1 1 1] #[2 2 2]])
> (axis-index-array #(3 3) 1)
- : (Array Index)
(array #[#[0 1 2] #[0 1 2] #[0 1 2]])
> (axis-index-array #() 0)
axis-index-array: contract violation
  expected: Index < 0
  given: 0
  argument position: 2nd
  other arguments...:
   '#()
```

As with `indexes-array`, the result does not allocate storage for its elements, and is strict.

```
(diagonal-array dims
                axes-length
                on-value
                off-value)  → (Array A)
  dims : Integer
  axes-length : Integer
  on-value : A
  off-value : A
```

Returns an array with *dims* axes, each with length *axes-length*. (For example, the returned array for *dims* = 2 is square.) The elements on the diagonal (i.e. at indexes of the form `(vector j j ...)` for `j < axes-length`) have the value *on-value*; the rest have *off-value*.

Example:

```
> (diagonal-array 2 7 1 0)
- : (Array (U One Zero))
```

```
(array
 #[#[1 0 0 0 0 0 0]
   #[0 1 0 0 0 0 0]
   #[0 0 1 0 0 0 0]
   #[0 0 0 1 0 0 0]
   #[0 0 0 0 1 0 0]
   #[0 0 0 0 0 1 0]
   #[0 0 0 0 0 0 1]])
```

As with `indexes-array`, the result does not allocate storage for its elements, and is strict.

6.8 Conversion

```
(Listof* A)
```

Equivalent to `(U A (Listof A) (Listof (Listof A)) ...)` if infinite unions were allowed. This is used as an argument type to `list*->array` and as the return type of `array->list*`.

```
(Vectorof* A)
```

Like `(Listof* A)`, but for vectors. See `vector*->array` and `array->vector*`.

```
(list->array lst) → (Array A)
  lst : (Listof A)
(list->array ds lst) → (Array A)
  ds : In-Indexes
  lst : (Listof A)
(array->list arr) → (Listof A)
  arr : (Array A)
```

Convert lists to immutable arrays and back.

The two-argument variant of `list->array` assumes the elements in *lst* are in row-major order.

For `array->list`, if *arr* has no axes or more than one axis, it is (conceptually) flattened before being converted to a list.

Examples:

```
> (list->array '(1 2 3))
- : (Array Positive-Byte)
(array #[1 2 3])
> (list->array '((1 2 3) (4 5)))
```

```
- : (Array
     (U (List One Positive-Byte Positive-Byte)
        (List Positive-Byte Positive-Byte)))
(array #['(1 2 3) '(4 5)])
> (list->array #(2 2) '(1 2 3 4))
- : (Array Positive-Byte)
(array #[#[1 2] #[3 4]])
> (array->list (array #[1 2 3]))
- : (Listof Positive-Byte)
'(1 2 3)
> (array->list (array 10))
- : (Listof Positive-Byte)
'(10)
> (array->list (array #[#[1 2 3] #[4 5 6]]))
- : (Listof Positive-Byte)
'(1 2 3 4 5 6)
```

For conversion between nested lists and multidimensional arrays, see `list*->array` and `array->list*`. For conversion from flat values to mutable arrays, see `vector->array`.

The arrays returned by `list->array` are always strict.

```
(vector->array vec) → (Mutable-Array A)
  vec : (Vectorof A)
(vector->array ds vec) → (Mutable-Array A)
  ds : In-Indexes
  vec : (Vectorof A)
(array->vector arr) → (Vectorof A)
  arr : (Array A)
```

Like `list->array` and `array->list`, but for vectors.

Examples:

```
> (vector->array #(1 2 3))
- : (Mutable-Array Integer)
(mutable-array #[1 2 3])
> (vector->array #((1 2 3) (4 5)))
- : (Mutable-Array
     (U (List One Positive-Byte Positive-Byte)
        (List Positive-Byte Positive-Byte)))
(mutable-array #['(1 2 3) '(4 5)])
> (vector->array #(2 2) #(1 2 3 4))
- : (Mutable-Array Integer)
(mutable-array #[#[1 2] #[3 4]])
> (array->vector (array #[1 2 3]))
```

```
- : (Vectorof Integer)
'#(1 2 3)
> (array->vector (array 10))
- : (Vectorof Integer)
'#(10)
> (array->vector (array #[#[1 2 3] #[4 5 6]]))
- : (Vectorof Integer)
'#(1 2 3 4 5 6)
```

For conversion between nested vectors and multidimensional arrays, see `vector*->array` and `array->vector*`.

```
(list*->array lsts pred?) → (Array A)
  lsts : (Listof* A)
  pred? : ((Listof* A) -> Any : A)
```

Converts a nested list of elements of type `A` to an array. The predicate *pred?* identifies elements of type `A`. The shape of *lsts* must be rectangular.

Examples:

```
> (list*->array 'singleton symbol?)
- : (Array Symbol)
(array 'singleton)
> (list*->array '(0 1 2 3) byte?)
- : (Array Byte)
(mutable-array #[0 1 2 3])
> (list*->array (list (list (list 5) (list 2 3))
                      (list (list 4.0) (list 1.4 0.2 9.3)))
                (make-predicate (Listof Nonnegative-Real)))
- : (Array (Listof Nonnegative-Real))
(mutable-array #[#['(5) '(2 3)] #['(4.0) '(1.4 0.2 9.3)]])
```

There is no well-typed Typed Racket function that behaves like `list*->array` but does not require *pred?*. Without an element predicate, there is no way to prove to the type checker that `list*->array`'s implementation correctly distinguishes elements from rows.

The arrays returned by `list*->array` are always strict.

```
(array->list* arr) → (Listof* A)
  arr : (Array A)
```

The inverse of `list*->array`.

```
(vector*->array vecs pred?) → (Mutable-Array A)
  vecs : (Vectorof* A)
  pred? : ((Vectorof* A) -> Any : A)
```

Like `list*->array`, but accepts nested vectors of elements.

Examples:

```
> (vector*->array 'singleton symbol?)
- : (Mutable-Array Symbol)
(mutable-array 'singleton)
> ((inst vector*->array Byte) #(0 1 2 3) byte?)
- : (Mutable-Array Byte)
(mutable-array #[0 1 2 3])
```

As in the last example, Typed Racket often needs help inferring `vector*->array`'s type parameters.

```
(array->vector* arr) → (Vectorof* A)
  arr : (Array A)
```

Like `array->list*`, but produces nested vectors of elements.

```
(array-list->array arrs [axis]) → (Array A)
  arrs : (Listof (Array A))
  axis : Integer = 0
```

Concatenates *arrs* along axis *axis* to form a new array. If the arrays have different shapes, they are broadcast first. The axis number *axis* must be nonnegative and *no greater than* the number of axes in the highest dimensional array in *arrs*.

Examples:

```
> (array-list->array (list (array 0) (array 1) (array 2) (array 3)))
- : (Array Byte)
(array #[0 1 2 3])
> (array-list->array (list (array 0) (array 1) (array 2) (array 3)) 1)
array-list->array: expected axis Index <= 0; given 1
> (array-list->array (list (array #[0 1 2 3]) (array #['a 'b 'c 'd])))
- : (Array (U 'a 'b 'c 'd Byte))
(array #[#[0 1 2 3] #['a 'b 'c 'd]])
> (array-list->array (list (array #[0 1 2 3]) (array '!)))
- : (Array (U '! Byte))
(array #[#[0 1 2 3] #['! '! '! '!]])
> (array-list->array (list (array #[0 1 2 3]) (array '!)) 1)
- : (Array (U '! Byte))
(array #[#[0 '!] #[1 '!] #[2 '!] #[3 '!]])
```

This function is a left inverse of `array->array-list`. (It cannot be a right inverse because broadcasting cannot be undone.)

For a similar function that does not increase the dimension of the broadcast arrays, see `array-append*`.

```
(array->array-list arr [axis]) → (Listof (Array A))
  arr : (Array A)
  axis : Integer = 0
```

Turns one axis of *arr* into a list of arrays. Each array in the result has the same shape. The axis number *axis* must be nonnegative and less than the number of *arr*'s axes.

Examples:

```
> (array->array-list (array #[0 1 2 3]))
- : (Listof (Array Byte))
(list (array 0) (array 1) (array 2) (array 3))
> (array->array-list (array #[#[1 2] #[10 20]]))
- : (Listof (Array Positive-Byte))
(list (array #[1 2]) (array #[10 20]))
> (array->array-list (array #[#[1 2] #[10 20]]) 1)
- : (Listof (Array Positive-Byte))
(list (array #[1 10]) (array #[2 20]))
> (array->array-list (array 10))
array->array-list: expected axis Index < 0; given 0
```

6.8.1 Printing

```
(array-custom-printer) → (All (A) ((Array A)
                                   Symbol
                                   Output-Port
                                   (U Boolean 0 1) -> Any))
(array-custom-printer print-array) → void?
  print-array : (All (A) ((Array A)
                          Symbol
                          Output-Port
                          (U Boolean 0 1) -> Any))
```

A parameter whose value is used to print subtypes of `Array`.

```
(print-array arr name port mode) → Any
  arr : (Array A)
  name : Symbol
  port : Output-Port
  mode : (U Boolean 0 1)
```

Prints an array using `array` syntax, using *name* instead of `'array` as the head form. This function is set as the value of `array-custom-printer` when `math/array` is first required.

Well-behaved `Array` subtypes do not call this function directly to print themselves. They call the current `array-custom-printer`:

```
> ((array-custom-printer)
   (array #[0 1 2 3])
   'my-cool-array
   (current-output-port)
   #t)
(my-cool-array #[0 1 2 3])
```

See `prop:custom-write` for the meaning of the *port* and *mode* arguments.

6.9 Comprehensions and Sequences

Sometimes sequential processing is unavoidable, so `math/array` provides loops and sequences.

```
(for/array: maybe-shape maybe-fill (for:-clause ...) maybe-type-ann
  body ...+)
(for*/array: maybe-shape maybe-fill (for:-clause ...) maybe-type-ann
  body ...+)

   maybe-shape =
               | #:shape ds

    maybe-fill =
               | #:fill fill

maybe-type-ann =
               | : body-type

  ds : In-Indexes
  fill : body-type
```

Creates arrays by generating elements in a `for`-loop or `for*`-loop. Unlike other Typed Racket loop macros, these accept a *body annotation*, which declares the type of elements. They do not accept an annotation for the entire type of the result.

Examples:

```
> (for/array: ([x  (in-range 3)] [y  (in-range 3)]) : Integer
```

```
    (+ x y))
- : (Mutable-Array Integer)
(mutable-array #[0 2 4])
> (for*/array: ([x  (in-range 3)] [y  (in-range 3)]) : Integer
    (+ x y))
- : (Mutable-Array Integer)
(mutable-array #[0 1 2 1 2 3 2 3 4])
```

The shape of the result is independent of the loop clauses: note that the last example does not have shape `#(3 3)`, but shape `#(9)`. To control the shape, use the `#:shape` keyword:

```
> (for*/array: #:shape #(3 3) ([x  (in-range 3)]
                               [y  (in-range 3)]) : Integer
    (+ x y))
- : (Mutable-Array Integer)
(mutable-array #[#[0 1 2] #[1 2 3] #[2 3 4]])
```

If the loop does not generate enough elements, the rest are filled with the *first* generated value:

```
> (for*/array: #:shape #(4) ([x  (in-range 1 3)]) x)
- : (Mutable-Array Any)
(mutable-array #[1 2 1 1])
```

To change this behavior, use the `#:fill` keyword:

```
> (for*/array: #:shape #(4) #:fill -1 ([x  (in-range 1 3)]) x)
- : (Mutable-Array Any)
(mutable-array #[1 2 -1 -1])
```

In the last two examples, the array's type is `(Mutable-Array Any)` because a body annotation was not given.

```
(for/array maybe-shape maybe-fill (for-clause ...)
  body ...+)
(for*/array maybe-shape maybe-fill (for-clause ...)
  body ...+)
```

Untyped versions of the loop macros.

```
(in-array arr) → (Sequenceof A)
  arr : (Array A)
```

Returns a sequence of *arr*'s elements in row-major order.

Examples:

```
> (define arr (array #[#[1 2] #[10 20]]))
> (for/list: : (Listof Integer) ([x  (in-array arr)]) x)
- : (Listof Integer)
'(1 2 10 20)
```

```
(in-array-axis arr [axis]) → (Sequenceof (Array A))
  arr : (Array A)
  axis : Integer = 0
```

Like `array->array-list`, but returns a sequence.

Examples:

```
> (define arr (array #[#[1 2] #[10 20]]))
> (sequence->list (in-array-axis arr))
- : (Listof (Array Positive-Byte))
(list (array #[1 2]) (array #[10 20]))
> (sequence->list (in-array-axis arr 1))
- : (Listof (Array Positive-Byte))
(list (array #[1 10]) (array #[2 20]))
```

```
(in-array-indexes ds) → (Sequenceof Indexes)
  ds : In-Indexes
```

Returns a sequence of indexes for shape *ds*, in row-major order.

Examples:

```
> (for/array: #:shape #(3 3) ([js  (in-array-
indexes #(3 3))]) : Indexes
    js)
- : (Mutable-Array Indexes)
(mutable-array
 #[#['#(0 0) '#(0 1) '#(0 2)]
   #['#(1 0) '#(1 1) '#(1 2)]
   #['#(2 0) '#(2 1) '#(2 2)]])
> (for*/array: #:shape #(3 3) ([j0  (in-range 3)]
                               [j1  (in-range 3)]) : In-Indexes
    (vector j0 j1))
- : (Mutable-Array In-Indexes)
(mutable-array
 #[#['#(0 0) '#(0 1) '#(0 2)]
   #['#(1 0) '#(1 1) '#(1 2)]
   #['#(2 0) '#(2 1) '#(2 2)]])
```

```
> (indexes-array #(3 3))
- : (Array Indexes)
(array
 #[#['#(0 0) '#(0 1) '#(0 2)]
   #['#(1 0) '#(1 1) '#(1 2)]
   #['#(2 0) '#(2 1) '#(2 2)]])
```

6.10 Pointwise Operations

Most of the operations documented in this section are simple macros that apply `array-map` to a function and their array arguments.

```
(array-map f) → (Array R)
  f : (-> R)
(array-map f arr0) → (Array R)
  f : (A -> R)
  arr0 : (Array A)
(array-map f arr0 arr1 arrs ...) → (Array R)
  f : (A B Ts ... -> R)
  arr0 : (Array A)
  arr1 : (Array B)
  arrs : (Array Ts)
```

Composes *f* with the given arrays' procedures. When the arrays' shapes do not match, they are broadcast to the same shape first. If broadcasting fails, `array-map` raises an error.

Examples:

```
> (array-map (λ: ([x : String]) (string-append x "!"))
             (array #[#["Hello" "I"] #["Am" "Shouting"]]))
- : (Array String)
(array #[#["Hello!" "I!"] #["Am!" "Shouting!"]])
> (array-map string-append
             (array #[#["Hello" "I"] #["Am" "Shouting"]])
             (array "!"))
- : (Array String)
(array #[#["Hello!" "I!"] #["Am!" "Shouting!"]])
> (array-map + (index-array #(3 3 3)) (array 2))
- : (Array Positive-Fixnum)
(array
 #[#[#[2 3 4]
     #[5 6 7]
     #[8 9 10]]
   #[#[11 12 13]
```

```
     #[14 15 16]
     #[17 18 19]]
   #[#[20 21 22]
     #[23 24 25]
     #[26 27 28]]])
> (array-map + (index-array #(2 2)) (index-array #(3 3)))
array-shape-broadcast: incompatible array shapes
(array-broadcasting #t): '#(2 2), '#(3 3)
```

Typed Racket can often derive fairly precise element types for the resulting array:

```
> (array-map * (array #[-4.3 -1.2 -0.2]) (array -2.81))
- : (Array Nonnegative-Flonum)
(array #[12.083 3.372 0.562])
```

How precise the result type is depends on the type of `f`. Preserving precise result types for lifted arithmetic operators is the main reason most pointwise operations are macro wrappers for `array-map`.

Unlike `map`, `array-map` can map a zero-argument function:

```
> (array-map (λ () "Whoa, Nelly!"))
- : (Array String)
(array "Whoa, Nelly!")
```

If the resulting zero-dimensional array is used in a pointwise operation with other arrays, it will be broadcast to their shape:

```
> (array-map + (array #[1 2 3]) (array-map (λ () -10)))
- : (Array Fixnum)
(array #[-9 -8 -7])
```

When explicitly instantiating `array-map`'s types using `inst`, instantiate `R` (the return type's element type) first, then the arguments' element types in order.

```
(inline-array-map f arrs ...)
```

Like `array-map`, but possibly faster. Inlining a map operation can allow Typed Racket's optimizer to replace `f` with something unchecked and type-specific (for example, replace `*` with `unsafe-fl*`), at the expense of code size.

```
(array+ arrs ...)
(array* arrs ...)
(array- arr0 arrs ...)
(array/ arr0 arrs ...)
(array-min arr0 arrs ...)
(array-max arr0 arrs ...)
```

Equivalent to mapping arithmetic operators over arrays. Note that because `(array-map f)` returns sensible answers, so do `(array+)` and `(array*)`.

Examples:

```
> (array+ (array #[#[0.0 1.0] #[2.0 3.0]]) (array 200))
- : (Array Positive-Flonum)
(array #[#[200.0 201.0] #[202.0 203.0]])
> (array+)
- : (Array Zero)
(array 0)
> (array*)
- : (Array One)
(array 1)
> (array/ (array #[2 1/2]))
- : (Array Positive-Exact-Rational)
(array #[1/2 2])
```

```
(array-scale arr x)
```

Equivalent to `(array* arr (array x))`, but faster.

```
(array-abs arr)
(array-sqr arr)
(array-sqrt arr)
(array-conjugate arr)
```

Equivalent to `(array-map f arr)`, where `f` is respectively `abs`, `sqr`, `sqrt`, or `conjugate`.

```
(array-real-part arr)
(array-imag-part arr)
(array-make-rectangular arr0 arr1)
(array-magnitude arr)
(array-angle arr)
(array-make-polar arr0 arr1)
```

Conversions to and from complex numbers, lifted to operate on arrays.

```
(array< arr0 arr1 arrs ...)
(array<= arr0 arr1 arrs ...)
(array> arr0 arr1 arrs ...)
(array>= arr0 arr1 arrs ...)
(array= arr0 arr1 arrs ...)
```

Equivalent to `(array-map f arr0 arr1 arrs ...)`, where `f` is respectively `<`, `<=`, `>`, `>=`, or `=`.

```
(array-not arr)
(array-and arr ...)
(array-or arr ...)
(array-if cond-arr true-arr false-err)
```

Boolean operators lifted to operate on arrays.

When given nonstrict arrays, the short-cutting behavior of `array-and`, `array-or` and `array-if` can keep their elements from being referred to (and thus computed). However, these macros cannot be used to distinguish base and inductive cases in a recursive function, because the array arguments are eagerly evaluated. For example, this function never returns, even when `array-strictness` is `#f`:

```
(: array-factorial ((Array Integer) -> (Array Integer)))
(define (array-factorial arr)
  (array-if (array<= arr (array 0))
            (array 1)
            (array* arr (array-factorial (array-
 arr (array 1))))))
```

6.10.1 Broadcasting

```
(array-broadcasting) → (U Boolean 'permissive)
(array-broadcasting broadcasting) → void?
  broadcasting : (U Boolean 'permissive)
```

Determines the rules used when broadcasting arrays for pointwise operations. See §6.3.2 “Broadcasting Control”.

```
(array-shape-broadcast dss [broadcasting]) → Indexes
  dss : (Listof Indexes)
  broadcasting : (U Boolean 'permissive) = (array-broadcasting)
```

Determines the shape of the resulting array if some number of arrays with shapes `dss` were broadcast for a pointwise operation using the given broadcasting rules. If broadcasting fails, `array-shape-broadcast` raises an error.

Examples:

```
> (array-shape-broadcast '())
- : Indexes
```

```
'#()
> (array-shape-broadcast (list (vector) ((inst vector Index) 10)))
- : Indexes
'#(10)
> (array-shape-broadcast (list ((inst vector Index) 2)
                               ((inst vector Index) 10)))
array-shape-broadcast: incompatible array shapes
(array-broadcasting #t): '#(2), '#(10)
> (array-shape-broadcast (list ((inst vector Index) 2)
                               ((inst vector Index) 10))
                         'permissive)
- : Indexes
'#(10)
```

```
(array-broadcast arr ds) → (Array A)
  arr : (Array A)
  ds : Indexes
```

Returns an array with shape *ds* made by inserting new axes and repeating rows. This is used for both `(array-broadcasting #t)` and `(array-broadcasting 'permissive)`.

Examples:

```
> (array-broadcast (array 10) ((inst vector Index) 10))
- : (Array Positive-Byte)
(array #[10 10 10 10 10 10 10 10 10 10])
> (array-broadcast (array #[0 1]) #())
array-broadcast: cannot broadcast to a lower-dimensional
shape; given (array #[0 1]) and '#()
> (array-broadcast (array #[0 1]) ((inst vector Index) 5))
- : (Array (U One Zero))
(array #[0 1 0 1 0])
```

When `array-strictness` is `#f`, `array-broadcast` always returns a nonstrict array.

When `array-strictness` is `#t`, `array-broadcast` returns a strict array when *arr* is nonstrict and the result has more elements than *arr*.

6.11 Indexing and Slicing

```
(array-ref arr js) → A
  arr : (Array A)
  js : In-Indexes
```

Returns the element of *arr* at position *js*. If any index in *js* is negative or not less than its corresponding axis length, array-ref raises an error.

```
(array-set! arr js value) → Void
  arr : (Settable-Array A)
  js : In-Indexes
  value : A
```

Sets the element of *arr* at position *js* to *value*. If any index in *js* is negative or not less than its corresponding axis length, array-set! raises an error.

```
(array-indexes-ref arr idxs) → (Array A)
  arr : (Array A)
  idxs : (Array In-Indexes)
```

High-level explanation: Returns multiple elements from *arr* in a new array.

Lower-level explanation: Returns an array with same shape as *idxs*, whose elements are array-ref'd from *arr* using the indexes in *idxs*.

Examples:

```
> (define arr (array #[#[1 2] #[10 20]]))
> (define idxs (array #['#(0 0) '#(1 1)]))
> (array-indexes-ref arr idxs)
- : (Array Positive-Byte)
(array #[1 20])
```

Implementation-level explanation: (array-indexes-ref *arr idxs*) is equivalent to

```
> (build-array (array-shape idxs)
               (λ: ([js : Indexes])
                 (array-ref arr (array-ref idxs js))))
- : (Array Positive-Byte)
(array #[1 20])
```

but faster.

```
(array-indexes-set! arr idxs vals) → Void
  arr : (Settable-Array A)
  idxs : (Array In-Indexes)
  vals : (Array A)
```

Indexes *arr* in the same way that array-indexes-ref does, but mutates elements. If *idxs* and *vals* do not have the same shape, they are broadcast first.

Examples:

```
> (define arr (mutable-array #[#[1 2] #[10 20]]))
> (define idxs (array #['#(0 0) '#(1 1)]))
> (array-indexes-set! arr idxs (array -1))
> arr
- : (Mutable-Array Integer)
(mutable-array #[#[-1 2] #[10 -1]])
```

When two indexes in *idxs* are the same, the element at that index is mutated more than once in some unspecified order.

```
(array-slice-ref arr specs) → (Array A)
  arr : (Array A)
  specs : (Listof Slice-Spec)
```

Returns a transformation of *arr* according to the list of slice specifications *specs*. See §6.4 "Slicing" for a discussion and examples.

```
(array-slice-set! arr specs vals) → Void
  arr : (Settable-Array A)
  specs : (Listof Slice-Spec)
  vals : (Array A)
```

Like array-indexes-set!, but for slice specifications. Equivalent to

```
(let ([idxs  (array-slice-ref (indexes-array (array-
shape arr)) specs)])
  (array-indexes-set! arr idxs vals))
```

Examples:

```
> (define arr (array->mutable-array (axis-index-array #(5 5) 1)))
> (array-slice-set! arr (list (:: 1 #f 2) (::)) (array 1))
> arr
- : (Mutable-Array Integer)
(mutable-array
 #[#[0 1 2 3 4]
   #[1 1 1 1 1]
   #[0 1 2 3 4]
   #[1 1 1 1 1]
   #[0 1 2 3 4]])
> (array-slice-set!
   arr (list (::) (:: 1 #f 2))
   (array-scale (array-slice-ref arr (list (::) (:: 1 #f 2))) -1))
> arr
- : (Mutable-Array Integer)
```

```
(mutable-array
 #[#[0 -1 2 -3 4]
   #[1 -1 1 -1 1]
   #[0 -1 2 -3 4]
   #[1 -1 1 -1 1]
   #[0 -1 2 -3 4]])
```

When a slice specification refers to an element in *arr* more than once, the element is mutated more than once in some unspecified order.

```
Slice-Spec
```

The type of a slice specification. Currently defined as

```
(U (Sequenceof Integer) Slice Slice-Dots Integer Slice-New-Axis)
```

A `(Sequenceof Integer)` slice specification causes `array-slice-ref` to pick rows from an axis. An `Integer` slice specification causes `array-slice-ref` to remove an axis by replacing it with one of its rows.

See §6.4 “Slicing” for an extended example.

```
Slice
(:: [end]) → Slice
  end : (U #f Integer) = #f
(:: start end [step]) → Slice
  start : (U #f Integer)
  end : (U #f Integer)
  step : Integer = 1
(slice? v) → Boolean
  v : Any
(slice-start s) → (U #f Fixnum)
  s : Slice
(slice-end s) → (U #f Fixnum)
  s : Slice
(slice-step s) → Fixnum
  s : Slice
```

The type of `in-range`-like slice specifications, its constructor, predicate, and accessors.

`array-slice-ref` interprets a `Slice` like an `in-range` sequence object. When *start* or *end* is `#f`, it is interpreted as an axis-length-dependent endpoint.

```
(slice->range-values s dk) → (Values Fixnum Fixnum Fixnum)
  s : Slice
  dk : Index
```

Given a slice *s* and an axis length *dk*, returns the arguments to `in-range` that would produce an equivalent slice specification.

This is used internally by `array-slice-ref` to interpret a `Slice` object as a sequence of indexes.

```
Slice-Dots
::... : Slice-Dots
(slice-dots? v) → Boolean
  v : Any
```

The type of greedy, multiple-axis-preserving slice specifications, its singleton value, and predicate.

```
Slice-New-Axis
(::new [dk]) → Slice-New-Axis
  dk : Integer = 1
(slice-new-axis? v) → Boolean
  v : Any
(slice-new-axis-length s) → Index
  s : Slice-New-Axis
```

The type of slice specifications that indicate inserting a new axis, its constructor, predicate, and accessor. The axis length *dk* must be nonnegative.

6.12 Transformations

```
(array-transform arr ds proc) → (Array A)
  arr : (Array A)
  ds : In-Indexes
  proc : (Indexes -> In-Indexes)
```

Returns an array with shape *ds* and elements taken from *arr*, by composing *arr*'s procedure with *proc*.

Possibly the most useless, but simplest example of *proc* disregards its input indexes and returns constant indexes:

```
> (define arr (array #[#[0 1] #[2 'three]]))
> (array-transform arr #(3 3) (λ: ([js : Indexes]) #(1 1)))
- : (Array (U 'three Byte))
(array
 #[#['three 'three 'three]
   #['three 'three 'three]
   #['three 'three 'three]])
```

Doubling an array in every dimension by duplicating elements:

```
> (define arr (index-array #(3 3)))
> arr
- : (Array Index)
(array #[#[0 1 2] #[3 4 5] #[6 7 8]])
> (array-transform
   arr
   (vector-map (λ: ([d : Index]) (* d 2)) (array-shape arr))
   (λ: ([js : Indexes])
     (vector-map (λ: ([j : Index]) (quotient j 2)) js)))
- : (Array Index)
(array
 #[#[0 0 1 1 2 2]
   #[0 0 1 1 2 2]
   #[3 3 4 4 5 5]
   #[3 3 4 4 5 5]
   #[6 6 7 7 8 8]
   #[6 6 7 7 8 8]])
```

When `array-strictness` is `#f`, the above result takes little more space than the original array.

Almost all array transformations, including §6.4 “Slicing”, are implemented using `array-transform` or its unsafe counterpart.

```
(array-append* arrs [k]) → (Array A)
  arrs : (Listof (Array A))
  k : Integer = 0
```

Appends the arrays in *arrs* along axis *k*. If the arrays’ shapes are not the same, they are first broadcast along all axes (except the *k*th).

Examples:

```
> (define arr (array #[#[0 1] #[2 3]]))
> (define brr (array #[#['a 'b] #['c 'd]]))
> (array-append* (list arr brr))
- : (Array (U 'a 'b 'c 'd Byte))
(array #[#[0 1] #[2 3] #['a 'b] #['c 'd]])
> (array-append* (list arr brr) 1)
- : (Array (U 'a 'b 'c 'd Byte))
(array #[#[0 1 'a 'b] #[2 3 'c 'd]])
> (array-append* (list arr (array 'x)))
- : (Array (U 'x Byte))
(array #[#[0 1] #[2 3] #['x 'x]])
```

For an append-like operation that increases the dimension of the broadcast arrays, see array-list->array.

```
(array-axis-insert arr k [dk]) → (Array A)
  arr : (Array A)
  k : Integer
  dk : Integer = 1
```

Inserts an axis of length *dk* before axis number *k*, which must be no greater than the dimension of *arr*.

Examples:

```
> (define arr (array #[#[0 1] #[2 3]]))
> (array-axis-insert arr 0)
- : (Array Byte)
(array #[#[#[0 1] #[2 3]]])
> (array-axis-insert arr 1)
- : (Array Byte)
(array #[#[#[0 1]] #[#[2 3]]])
> (array-axis-insert arr 2)
- : (Array Byte)
(array #[#[#[0] #[1]] #[#[2] #[3]]])
> (array-axis-insert arr 1 2)
- : (Array Byte)
(array #[#[#[0 1] #[0 1]] #[#[2 3] #[2 3]]])
```

```
(array-axis-ref arr k jk) → (Array A)
  arr : (Array A)
  k : Integer
  jk : Integer
```

Removes an axis from *arr* by keeping only row *jk* in axis *k*, which must be less than the dimension of *arr*.

Examples:

```
> (define arr (array #[#[0 1] #[2 3]]))
> (array-axis-ref arr 0 0)
- : (Array Byte)
(array #[0 1])
> (array-axis-ref arr 0 1)
- : (Array Byte)
(array #[2 3])
> (array-axis-ref arr 1 0)
- : (Array Byte)
(array #[0 2])
```

```
(array-axis-swap arr k0 k1) → (Array A)
  arr : (Array A)
  k0 : Integer
  k1 : Integer
```

Returns an array like *arr*, but with axes *k0* and *k1* swapped. In two dimensions, this is called a transpose.

Examples:

```
> (array-axis-swap (array #[#[0 1] #[2 3]]) 0 1)
- : (Array Byte)
(array #[#[0 2] #[1 3]])
> (define arr (indexes-array #(2 2 2)))
> arr
- : (Array Indexes)
(array
 #[#[#['#(0 0 0) '#(0 0 1)]
     #['#(0 1 0) '#(0 1 1)]]
   #[#['#(1 0 0) '#(1 0 1)]
     #['#(1 1 0) '#(1 1 1)]]])
> (array-axis-swap arr 0 1)
- : (Array Indexes)
(array
 #[#[#['#(0 0 0) '#(0 0 1)]
     #['#(1 0 0) '#(1 0 1)]]
   #[#['#(0 1 0) '#(0 1 1)]
     #['#(1 1 0) '#(1 1 1)]]])
> (array-axis-swap arr 1 2)
- : (Array Indexes)
(array
 #[#[#['#(0 0 0) '#(0 1 0)]
     #['#(0 0 1) '#(0 1 1)]]
   #[#['#(1 0 0) '#(1 1 0)]
     #['#(1 0 1) '#(1 1 1)]]])
```

```
(array-axis-permute arr perm) → (Array A)
  arr : (Array A)
  perm : (Listof Integer)
```

Returns an array like *arr*, but with axes permuted according to *perm*.

The list *perm* represents a mapping from source axis numbers to destination axis numbers: the source is the list position, the destination is the list element. For example, the permutation

`'(0 1 2)` is the identity permutation for three-dimensional arrays, `'(1 0)` swaps axes `0` and `1`, and `'(3 1 2 0)` swaps axes `0` and `3`.

The permutation must contain each integer from `0` to `(- (array-dims arr) 1)` exactly once.

Examples:

```
> (array-axis-swap (array #[#[0 1] #[2 3]]) 0 1)
- : (Array Byte)
(array #[#[0 2] #[1 3]])
> (array-axis-permute (array #[#[0 1] #[2 3]]) '(1 0))
- : (Array Byte)
(array #[#[0 2] #[1 3]])
```

```
(array-reshape arr ds) → (Array A)
  arr : (Array A)
  ds : In-Indexes
```

Returns an array with elements in the same row-major order as *arr*, but with shape *ds*. The product of the indexes in *ds* must be `(array-size arr)`.

Examples:

```
> (define arr (indexes-array #(2 3)))
> arr
- : (Array Indexes)
(array #[#['#(0 0) '#(0 1) '#(0 2)] #['#(1 0) '#(1 1) '#(1 2)]])
> (array-reshape arr #(3 2))
- : (Array Indexes)
(array #[#['#(0 0) '#(0 1)] #['#(0 2) '#(1 0)] #['#(1 1) '#(1
2)]])
> (array-reshape (index-array #(3 3)) #(9))
- : (Array Index)
(array #[0 1 2 3 4 5 6 7 8])
```

```
(array-flatten arr) → (Array A)
  arr : (Array A)
```

Returns an array with shape `(vector (array-size arr))`, with the elements of *arr* in row-major order.

Examples:

```
> (array-flatten (array 10))
```

```
- : (Array Positive-Byte)
(array #[10])
> (array-flatten (array #[#[0 1] #[2 3]]))
- : (Array Byte)
(array #[0 1 2 3])
```

6.13 Folds, Reductions and Expansions

6.13.1 Axis Folds

```
(array-axis-fold arr k f) → (Array A)
  arr : (Array A)
  k : Integer
  f : (A A -> A)
(array-axis-fold arr k f init) → (Array B)
  arr : (Array A)
  k : Integer
  f : (A B -> B)
  init : B
```

Folds a binary function *f* over axis *k* of *arr*. The result has the shape of *arr* but with axis *k* removed.

The three-argument variant uses the first value of each row in axis *k* as *init*. It therefore requires axis *k* to have positive length.

Examples:

```
> (define arr (index-array #(3 4)))
> arr
- : (Array Index)
(array #[#[0 1 2 3] #[4 5 6 7] #[8 9 10 11]])
> (array-axis-fold arr 0 +)
- : (Array Nonnegative-Integer)
(array #[12 15 18 21])
> (array-axis-fold arr 1 (inst cons Index (Listof Index)) empty)
- : (Array (Listof Index))
(array #['(3 2 1 0) '(7 6 5 4) '(11 10 9 8)])
```

Notice that the second example returns an array of reversed lists. This is therefore a left fold; see `foldl`.

If you need control over the order of evaluation in axis *k*'s rows, see `array-axis-reduce`.

```
(array-axis-sum arr k)
(array-axis-sum arr k init)
(array-axis-prod arr k)
(array-axis-prod arr k init)

  arr : (Array Number)
  k : Integer
  init : Number
```

```
(array-axis-min arr k)
(array-axis-min arr k init)
(array-axis-max arr k)
(array-axis-max arr k init)

  arr : (Array Real)
  k : Integer
  init : Real
```

Some standard per-axis folds, defined in terms of `array-axis-fold`. The two-argument variants require axis `k` to have positive length.

Examples:

```
> (define arr (index-array #(3 4)))
> arr
- : (Array Index)
(array #[#[0 1 2 3] #[4 5 6 7] #[8 9 10 11]])
> (array-axis-fold arr 1 +)
- : (Array Nonnegative-Integer)
(array #[6 22 38])
> (array-axis-sum arr 1)
- : (Array Nonnegative-Integer)
(array #[6 22 38])
> (array-axis-sum arr 0 0.0)
- : (Array Nonnegative-Flonum)
(array #[12.0 15.0 18.0 21.0])
```

```
(array-axis-count arr k pred?) → (Array Index)
  arr : (Array A)
  k : Integer
  pred? : (A -> Any)
```

Counts the elements `x` in rows of axis `k` for which `(pred? x)` is true.

Examples:

```
> (define arr (index-array #(3 3)))
> arr
- : (Array Index)
(array #[#[0 1 2] #[3 4 5] #[6 7 8]])
> (array-axis-count arr 1 odd?)
- : (Array Index)
(array #[1 2 1])
```

```
(array-axis-and arr k) → (Array (U A Boolean))
  arr : (Array A)
  k : Integer
(array-axis-or arr k) → (Array (U A #f))
  arr : (Array A)
  k : Integer
```

Apply `and` or `or` to each row in axis *k* of array *arr*. Evaluation is short-cut as with the `and` and `or` macros, which is only observable if *arr* is nonstrict.

In the following example, computing the second array element sets `second?` to `#t`:

```
> (define second? (ann #f Boolean))
> (define arr
    (parameterize ([array-strictness #f])
      (build-array #(2) (λ: ([js : Indexes])
                          (cond [(zero? (vector-ref js 0))  #f]
                                [else  (set! second? #t)
                                       #t])))))
```

Printing *arr* causes `(set! second? #t)` to be evaluated, but applying `array-axis-and` does not:

```
> (array-axis-and arr 0)
- : (Array Boolean)
(array #f)
> second?
- : Boolean
#f
```

However, if *arr* were strict, `(set! second? #t)` would be evaluated when *arr* was created.

6.13.2 Whole-Array Folds

```
(array-fold arr g) → (Array A)
  arr : (Array A)
  g : ((Array A) Index -> (Array A))
```

Folds *g* over *each axis* of *arr*, in reverse order. The arguments to *g* are an array (initially *arr*) and the current axis. It should return an array with one fewer dimension than the array given, but does not have to.

Examples:

```
> (define arr (index-array #(3 4)))
> arr
- : (Array Index)
(array #[#[0 1 2 3] #[4 5 6 7] #[8 9 10 11]])
> (array-fold arr (λ: ([arr : (Array Integer)] [k : Index])
                    (array-axis-sum arr k)))
- : (Array Integer)
(array 66)
> (apply + (array->list arr))
- : Integer [more precisely: Nonnegative-Integer]
66
> (array-ref (array-fold arr (inst array->list-
array (Listof* Integer)))
             #())
- : (U (Listof (Rec g1363092 (U (Listof g1363092) Integer))) In-
dex)
'((0 1 2 3) (4 5 6 7) (8 9 10 11))
```

```
(array-all-fold arr f) → A
  arr : (Array A)
  f : (A A -> A)
(array-all-fold arr f init) → A
  arr : (Array A)
  f : (A A -> A)
  init : A
```

Folds *f* over every element of *arr* by folding *f* over each axis in reverse order. The two-argument variant is equivalent to

```
(array-ref (array-fold arr (λ: ([arr : (Array A)] [k : Index])
                             (array-axis-fold arr k f)))
           #())
```

and the three-argument variant is similar. The two-argument variant requires every axis to have positive length.

Examples:

```
> (define arr (index-array #(3 4)))
> arr
- : (Array Index)
(array #[#[0 1 2 3] #[4 5 6 7] #[8 9 10 11]])
> (array-all-fold arr +)
- : Integer [more precisely: Nonnegative-Integer]
66
> (array-all-fold (array #[]) + 0.0)
- : Flonum [more precisely: Nonnegative-Flonum]
0.0
```

Because *f* is folded over the last axis first, it receives *arr*'s elements (as its first argument) in row-major order.

```
(array-all-sum arr)
(array-all-sum arr init)
(array-all-prod arr)
(array-all-prod arr init)

  arr : (Array Number)
  init : Number
```

```
(array-all-min arr)
(array-all-min arr init)
(array-all-max arr)
(array-all-max arr init)

  arr : (Array Real)
  init : Real
```

Some standard whole-array folds, defined in terms of `array-all-fold`. The one-argument variants require each axis in *arr* to have positive length.

Examples:

```
> (define arr (index-array #(3 4)))
> arr
- : (Array Index)
(array #[#[0 1 2 3] #[4 5 6 7] #[8 9 10 11]])
> (array-all-fold arr +)
- : Integer [more precisely: Nonnegative-Integer]
66
```

```
> (array-all-sum arr)
- : Integer [more precisely: Nonnegative-Integer]
66
> (array-all-sum arr 0.0)
- : Real [more precisely: Nonnegative-Real]
66.0
```

```
(array-all-and arr) → (U A Boolean)
  arr : (Array A)
(array-all-or arr) → (U A #f)
  arr : (Array A)
```

Apply `and` or `or` to *arr*'s elements using short-cut evaluation in row-major order.

Examples:

```
> (define arr (index-array #(3 3)))
> (array-all-and (array= arr arr))
- : Boolean
#t
> (define brr (array+ arr (array 1)))
> (array-all-and (array= arr brr))
- : Boolean
#f
> (array-all-or (array= arr (array 0)))
- : Boolean
#t
```

`(array-all-and arr)` is defined as

```
(parameterize ([array-strictness #f])
  (array-ref (array-fold arr array-axis-and) #()))
```

and `array-all-or` is defined similarly, using `array-axis-or`.

```
(array-count pred? arrs ...)

  arrs : (Array Ts)
  pred? : (Ts ... -> Any)
```

When given one array `arr`, returns the number of elements `x` in `arr` for which `(pred? x)` is true. When given multiple arrays, `array-count` does the same with the corresponding elements from any number of arrays. If the arrays' shapes are not the same, they are broadcast first.

Examples:

```
> (array-count zero? (array #[#[0 1 0 2] #[0 3 -1 4]]))
- : Integer [more precisely: Index]
3
> (array-count equal?
               (array #[#[0 1] #[2 3] #[0 1] #[2 3]])
               (array #[0 1]))
- : Integer [more precisely: Index]
4
```

`(array-count` *`pred?`* *`arrs`* `...)` is like

```
(array-all-sum (array-map (λ (x ...) (if (pred? x ...) 1 0)) arrs ...)
               0)
```

but does not create intermediate (strict) arrays, and always returns an `Index`.

```
(array-andmap pred? arrs ...)
(array-ormap pred? arrs ...)

  arrs : (Array Ts)
  pred? : (Ts ... -> Any)
```

Like `andmap` and `ormap`, but for arrays. Evaluation is short-cut, in row-major order. If the arrays' shapes are not the same, they are broadcast first.

Determining whether each row is equal to `(array #[0 1])`:

```
> (array-andmap equal?
                (array #[#[0 1] #[0 1] #[0 1] #[0 1]])
                (array #[0 1]))
- : Boolean
#t
```

Determining whether any row has `0` as its first element or `1` as its second:

```
> (array-ormap equal?
               (array #[#[0 2] #[2 3] #[1 1] #[2 3]])
               (array #[0 1]))
- : Boolean
#t
```

Determining whether any row is equal to `(array #[0 1])`:

```
> (array-ormap equal?
```

```
                (array->list-array (array #[#[0 2] #[2 3] #[1 1] #[2 3]]))
                (array->list-array (array #[0 1])))
- : Boolean
#f
```

`(array-andmap pred? arrs ...)` is defined as

```
(parameterize ([array-strictness #f])
  (array-all-and (array-map pred? arrs ...)))
```

and `array-ormap` is defined similarly, using `array-all-or`.

6.13.3 General Reductions and Expansions

```
(array-axis-reduce arr k h) → (Array B)
  arr : (Array A)
  k : Integer
  h : (Index (Integer -> A) -> B)
```

Like `array-axis-fold`, but allows evaluation control (such as short-cutting `and` and `or`) by moving the loop into *h*. The result has the shape of *arr*, but with axis *k* removed.

The arguments to *h* are the length of axis *k* and a procedure that retrieves elements from that axis's rows by their indexes in axis *k*. It should return the elements of the resulting array.

For example, summing the squares of the rows in axis `1`:

```
> (define arr (index-array #(3 3)))
> arr
- : (Array Index)
(array #[#[0 1 2] #[3 4 5] #[6 7 8]])
> (array-axis-reduce
   arr 1
   (λ: ([dk : Index] [proc : (Integer -> Real)])
     (for/fold: ([s : Real 0]) ([jk  (in-range dk)])
       (+ s (sqr (proc jk))))))
- : (Array Real)
(array #[5 50 149])
> (array-axis-sum (array-map sqr arr) 1)
- : (Array Nonnegative-Integer)
(array #[5 50 149])
```

Transforming an axis into a list using `array-axis-fold` and `array-axis-reduce`:

```
> (array-map (inst reverse Index)
             (array-axis-fold arr 1
                              (inst cons Index (Listof Index))
                              empty))
- : (Array (Listof Index))
(array #['(0 1 2) '(3 4 5) '(6 7 8)])
> (array-axis-reduce arr 1 (inst build-list Index))
- : (Array (Listof Index))
(array #['(0 1 2) '(3 4 5) '(6 7 8)])
```

The latter is essentially the definition of `array->list-array`.

Every fold, including `array-axis-fold`, is ultimately defined using `array-axis-reduce` or its unsafe counterpart.

```
(array-axis-expand arr k dk g) → (Array B)
  arr : (Array A)
  k : Integer
  dk : Integer
  g : (A Index -> B)
```

Inserts a new axis number *k* of length *dk*, using *g* to generate values; *k* must be *no greater than* the dimension of *arr*, and *dk* must be nonnegative.

Conceptually, *g* is applied *dk* times to each element in each row of axis *k*, once for each nonnegative index `jk < dk`.

Turning vector elements into rows of a new last axis using `array-axis-expand` and `vector-ref`:

```
> (define arr (array #['#(a b c) '#(d e f) '#(g h i)]))
> (array-axis-expand arr 1 3 vector-ref)
- : (Array Any)
(array #[#['a 'b 'c] #['d 'e 'f] #['g 'h 'i]])
```

Creating a `vandermonde-matrix`:

```
> (array-axis-expand (list->array '(1 2 3 4)) 1 5 expt)
- : (Array Positive-Integer)
(array #[#[1 1 1 1 1] #[1 2 4 8 16] #[1 3 9 27 81] #[1 4 16 64
256]])
```

This function is a dual of `array-axis-reduce` in that it can be used to invert applications of `array-axis-reduce`. To do so, *g* should be a destructuring function that is dual to

the constructor passed to array-axis-reduce. Example dual pairs are vector-ref and build-vector, and list-ref and build-list.

(Do not pass list-ref to array-axis-expand if you care about performance, though. See list-array->array for a more efficient solution.)

```
(array->list-array arr [k]) → (Array (Listof A))
  arr : (Array A)
  k : Integer = 0
```

Returns an array of lists, computed as if by applying list to the elements in each row of axis k.

Examples:

```
> (define arr (index-array #(3 3)))
> arr
- : (Array Index)
(array #[#[0 1 2] #[3 4 5] #[6 7 8]])
> (array->list-array arr 1)
- : (Array (Listof Index))
(array #['(0 1 2) '(3 4 5) '(6 7 8)])
> (array-ref (array->list-array (array->list-array arr 1) 0) #())
- : (Listof (Listof Index))
'((0 1 2) (3 4 5) (6 7 8))
```

See mean for more useful examples, and array-axis-reduce for an example that shows how array->list-array is implemented.

```
(list-array->array arr [k]) → (Array A)
  arr : (Array (Listof A))
  k : Integer = 0
```

Returns an array in which the list elements of arr comprise a new axis k. Equivalent to (array-axis-expand arr k n list-ref) where n is the length of the lists in arr, but with O(1) indexing.

Examples:

```
> (define arr (array->list-array (index-array #(3 3)) 1))
> arr
- : (Array (Listof Index))
(array #['(0 1 2) '(3 4 5) '(6 7 8)])
> (list-array->array arr 1)
- : (Array Index)
(array #[#[0 1 2] #[3 4 5] #[6 7 8]])
```

For fixed *k*, this function and array->list-array are mutual inverses with respect to their array arguments.

6.14 Other Array Operations

6.14.1 Fast Fourier Transform

Wikipedia: Discrete Fourier Transform

```
(array-axis-fft arr k) → (Array Float-Complex)
  arr : (Array Number)
  k : Integer
```

Performs a discrete Fourier transform on axis *k* of *arr*. The length of *k* must be an integer power of two. (See power-of-two?.) The scaling convention is determined by the parameter dft-convention, which defaults to the convention used in signal processing.

The transform is done in parallel using max-math-threads threads.

```
(array-axis-inverse-fft arr k) → (Array Float-Complex)
  arr : (Array Number)
  k : Integer
```

The inverse of array-axis-fft, performed by parameterizing the forward transform on (dft-inverse-convention).

```
(array-fft arr) → FCArray
  arr : (Array Number)
```

Performs a discrete Fourier transform on each axis of *arr* using array-axis-fft.

```
(array-inverse-fft arr) → FCArray
  arr : (Array Number)
```

The inverse of array-fft, performed by parameterizing the forward transform on (dft-inverse-convention).

6.15 Subtypes

6.15.1 Flonum Arrays

```
FlArray
```

The type of *flonum arrays*, a subtype of `(Settable-Array Flonum)` that stores its elements in an `FlVector`. A flonum array is always strict.

```
(flarray #[#[...] ...])
```

Like `array`, but creates flonum arrays. The listed elements must be real numbers, and may be exact.

Examples:

```
> (flarray 0.0)
- : FlArray
(flarray 0.0)
> (flarray #['x])
eval:312:0: Type Checker: type mismatch
  expected: Real
  given: 'x
  in: #((quote x))
> (flarray #[#[1 2] #[3 4]])
- : FlArray
(flarray #[#[1.0 2.0] #[3.0 4.0]])
```

```
(array->flarray arr) → FlArray
  arr : (Array Real)
```

Returns a flonum array that has approximately the same elements as *arr*. Exact elements will likely lose precision during conversion.

```
(flarray-data arr) → FlVector
  arr : FlArray
```

Returns the elements of *arr* in a flonum vector, in row-major order.

Example:

```
> (flarray-data (flarray #[#[1 2] #[3 4]]))
- : FlVector
(flvector 1.0 2.0 3.0 4.0)
```

```
(flarray-map f arrs ...) → FlArray
  f : (Flonum ... -> Flonum)
  arrs : FlArray
```

Maps the function *f* over the arrays *arrs*. If the arrays do not have the same shape, they are broadcast first. If the arrays do have the same shape, this operation can be quite fast.

The function *f* is meant to accept the same number of arguments as the number of its following flonum array arguments. However, a current limitation in Typed Racket requires *f* to accept *any* number of arguments. To map a single-arity function such as `fl+`, for now, use `inline-flarray-map` or `array-map`.

```
(inline-flarray-map f arrs ...)

  f : (Flonum ... -> Flonum)
  arrs : FlArray
```

Like `inline-array-map`, but for flonum arrays.

This is currently unavailable in untyped Racket.

```
(flarray+ arr0 arr1) → FlArray
  arr0 : FlArray
  arr1 : FlArray
(flarray* arr0 arr1) → FlArray
  arr0 : FlArray
  arr1 : FlArray
(flarray- arr) → FlArray
  arr : FlArray
(flarray- arr0 arr1) → FlArray
  arr0 : FlArray
  arr1 : FlArray
(flarray/ arr) → FlArray
  arr : FlArray
(flarray/ arr0 arr1) → FlArray
  arr0 : FlArray
  arr1 : FlArray
(flarray-min arr0 arr1) → FlArray
  arr0 : FlArray
  arr1 : FlArray
(flarray-max arr0 arr1) → FlArray
  arr0 : FlArray
  arr1 : FlArray
(flarray-scale arr x) → FlArray
  arr : FlArray
  x : Flonum
(flarray-abs arr) → FlArray
  arr : FlArray
(flarray-sqr arr) → FlArray
  arr : FlArray
(flarray-sqrt arr) → FlArray
  arr : FlArray
```

Arithmetic lifted to flonum arrays.

6.15.2 Float-Complex Arrays

```
FCArray
```

The type of *float-complex arrays*, a subtype of `(Settable-Array Float-Complex)` that stores its elements in a pair of `FlVectors`. A float-complex array is always strict.

```
(fcarray #[#[...] ...])
```

Like `array`, but creates float-complex arrays. The listed elements must be numbers, and may be exact.

Examples:

```
> (fcarray 0.0)
- : FCArray
(fcarray 0.0+0.0i)
> (fcarray #['x])
eval:316:0: Type Checker: type mismatch
  expected: Number
  given: 'x
  in: #((quote x))
> (fcarray #[#[1 2+1i] #[3 4+3i]])
- : FCArray
(fcarray #[#[1.0+0.0i 2.0+1.0i] #[3.0+0.0i 4.0+3.0i]])
```

```
(array->fcarray arr) → FCArray
  arr : (Array Number)
```

Returns a float-complex array that has approximately the same elements as *arr*. Exact elements will likely lose precision during conversion.

```
(fcarray-real-data arr) → FlVector
  arr : FCArray
(fcarray-imag-data arr) → FlVector
  arr : FCArray
```

Return the real and imaginary parts of *arr*'s elements in flonum vectors, in row-major order.

Examples:

```
> (define arr (fcarray #[#[1 2+1i] #[3 4+3i]]))
> (fcarray-real-data arr)
- : FlVector
(flvector 1.0 2.0 3.0 4.0)
> (fcarray-imag-data arr)
- : FlVector
(flvector 0.0 1.0 0.0 3.0)
```

```
(fcarray-map f arrs ...) → FCArray
  f : (Float-Complex ... -> Float-Complex)
  arrs : FCArray
```

Maps the function *f* over the arrays *arrs*. If the arrays do not have the same shape, they are broadcast first. If the arrays do have the same shape, this operation can be quite fast.

The function *f* is meant to accept the same number of arguments as the number of its following float-complex array arguments. However, a current limitation in Typed Racket requires *f* to accept *any* number of arguments. To map a single-arity function, for now, use `inline-fcarray-map` or `array-map`.

```
(inline-fcarray-map f arrs ...)

  f : (Float-Complex ... -> Float-Complex)
  arrs : FCArray
```

Like `inline-array-map`, but for float-complex arrays.

This is currently unavailable in untyped Racket.

```
(fcarray+ arr0 arr1) → FCArray
  arr0 : FCArray
  arr1 : FCArray
(fcarray* arr0 arr1) → FCArray
  arr0 : FCArray
  arr1 : FCArray
(fcarray- arr) → FCArray
  arr : FCArray
(fcarray- arr0 arr1) → FCArray
  arr0 : FCArray
  arr1 : FCArray
(fcarray/ arr) → FCArray
  arr : FCArray
(fcarray/ arr0 arr1) → FCArray
  arr0 : FCArray
  arr1 : FCArray
```

```
(fcarray-scale arr z) → FCArray
  arr : FCArray
  z : Float-Complex
(fcarray-sqr arr) → FCArray
  arr : FCArray
(fcarray-sqrt arr) → FCArray
  arr : FCArray
(fcarray-conjugate arr) → FCArray
  arr : FCArray
```

Arithmetic lifted to float-complex arrays.

```
(fcarray-real-part arr) → FlArray
  arr : FCArray
(fcarray-imag-part arr) → FlArray
  arr : FCArray
(fcarray-make-rectangular arr0 arr1) → FCArray
  arr0 : FlArray
  arr1 : FlArray
(fcarray-magnitude arr) → FlArray
  arr : FCArray
(fcarray-angle arr) → FlArray
  arr : FCArray
(fcarray-make-polar arr0 arr1) → FCArray
  arr0 : FlArray
  arr1 : FlArray
```

Conversions to and from complex numbers, lifted to flonum and float-complex arrays.

6.16 Strictness

```
(array-strictness) → Boolean
(array-strictness strictness) → void?
  strictness : Boolean
```

Determines whether `math/array` functions return strict arrays. The default value is `#t`.

See §6.5 “Nonstrict Arrays” for a discussion on nonstrict arrays.

```
(array-strict? arr) → Boolean
  arr : (Array A)
```

Returns `#t` when *arr* is strict.

Examples:

```
> (define arr
    (parameterize ([array-strictness #f])
      (array+ (array 10) (array #[0 1 2 3]))))
> (array-strict? arr)
- : Boolean
#f
> (array-strict! arr)
> (array-strict? arr)
- : Boolean
#t
```

```
(array-strict! arr) → Void
  arr : (Array A)
```

Causes *arr* to compute and store all of its elements. Thereafter, *arr* computes its elements by retrieving them from the store.

If *arr* is already strict, (array-strict! *arr*) does nothing.

```
(array-strict arr)

  arr : (Array A)
```

An expression form of array-strict!, which is often more convenient. First evaluates (array-strict! *arr*), then returns *arr*.

This is a macro so that Typed Racket will preserve *arr*'s type exactly. If it were a function, (array-strict *arr*) would always have the type (Array A), even if *arr* were a subtype of (Array A), such as (Mutable-Array A).

```
(array-default-strict! arr) → Void
  arr : (Array A)
(array-default-strict arr)

  arr : (Array A)
```

Like array-strict! and array-strict, but do nothing when array-strictness is #f.

Apply one of these to return values from library functions to ensure that users get strict arrays by default. See §6.5 "Nonstrict Arrays" for details.

```
(build-simple-array ds proc) → (Array A)
  ds : In-Indexes
  proc : (Indexes -> A)
```

Like build-array, but returns an array without storage that is nevertheless considered to be strict, regardless of the value of array-strictness. Such arrays will *not* cache their elements when array-strict! or array-strict is applied to them.

Use build-simple-array to create arrays that represent simple functions of their indexes. For example, basic array constructors such as make-array are defined in terms of this or its unsafe counterpart.

Be careful with this function. While it creates arrays that are always memory-efficient, it is easy to ruin your program's performance by using it to define arrays for which element lookup is permanently expensive. In the wrong circumstances, using it instead of build-array can turn a linear algorithm into an exponential one!

In general, use build-simple-array when

- Computing an element is never more expensive than computing a row-major index followed by applying vector-ref. An example is index-array, which only computes row-major indexes.
- Computing an element is independent of any other array's elements. In this circumstance, it is impossible to compose some unbounded number of possibly expensive array procedures.
- You can prove that each element will be computed at most once, throughout the entire life of your program. This is true, for example, when the result is sent only to a function that makes a copy of it, such as array-lazy or array->mutable-array.

See array-lazy for an example of the last circumstance.

```
(array-lazy arr) → (Array A)
  arr : (Array A)
```

Returns an immutable, nonstrict array with the same elements as *arr*, but element computations are cached.

Perhaps the most natural way to use array-lazy is for so-called "dynamic programming," or memoizing a function that happens to have a rectangular domain. For example, this computes the first 10 Fibonacci numbers in linear time:

```
> (define: fibs : (Array Natural)
    (array-lazy
     (build-simple-array
      #(10) (λ: ([js : Indexes])
              (define j (vector-ref js 0))
              (cond [(j . < . 2)  j]
```

```
                    [else  (+ (array-ref fibs (vector (- j 1)))
                              (array-ref fibs (vector (- j 2))))])))))
> fibs
- : (Array Nonnegative-Integer)
(array #[0 1 1 2 3 5 8 13 21 34])
```

Because `build-simple-array` never stores its elements, its procedure argument may refer to the array it returns. Wrapping its result with `array-lazy` makes each `array-ref` take no more than linear time; further, each takes constant time when the elements of `fibs` are computed in order. Without `array-lazy`, computing the elements of `fibs` would take exponential time.

Printing a lazy array computes and caches all of its elements, as does applying `array-strict!` or `array-strict` to it.

Except for arrays returned by `build-simple-array`, it is useless to apply `array-lazy` to a strict array. Using the lazy copy instead of the original only degrades performance.

While it may seem that `array-lazy` should just return *arr* when *arr* is strict, this would violate the invariant that `array-lazy` returns immutable arrays. For example:

```
> (: array-maybe-lazy (All (A) ((Array A) -> (Array A))))
> (define (array-maybe-lazy arr)
    (if (array-strict? arr) arr (array-lazy arr)))
> (define arr (mutable-array #[0 1 2 3]))
> (define brr (array-maybe-lazy arr))
> (array-set! arr #(0) -1000)
> brr
- : (Array Integer)
(array #[-1000 1 2 3])
```

7 Matrices and Linear Algebra

Performance Warning: Matrix values are arrays, as exported by `math/array`. The same performance warning applies: operations are currently 25-50 times slower in untyped Racket than in Typed Racket, due to the overhead of checking higher-order contracts. We are working on it.

For now, if you need speed, use the `typed/racket` language.

```
(require math/matrix)
```
package: `math-lib`

Like all of `math`, `math/matrix` is a work in progress. Most of the basic algorithms are implemented, but some are still in planning. Possibly the most useful unimplemented algorithms are

- LUP decomposition (currently, LU decomposition is implemented, in `matrix-lu`)
- `matrix-solve` for triangular matrices
- Singular value decomposition (SVD)
- Eigendecomposition
- Decomposition-based solvers
- Pseudoinverse and least-squares solving

7.1 Introduction

From the point of view of the functions in `math/matrix`, a *matrix* is an `Array` with two axes and at least one entry, or an array for which `matrix?` returns `#t`.

Technically, a matrix's entries may be any type, and some fully polymorphic matrix functions such as `matrix-row` and `matrix-map` operate on any kind of matrix. Other functions, such as `matrix+`, require their matrix arguments to contain numeric values.

7.1.1 Function Types

The documentation for `math/matrix` functions use the type `Matrix`, a synonym of `Array`, when the function either requires that an argument is a matrix or ensures that a return value is a matrix.

Most functions that implement matrix algorithms are documented as accepting `(Matrix Number)` values. This includes `(Matrix Real)`, which is a subtype. Most of these functions have a more precise type than is documented. For example, `matrix-conjugate` has the type

```
(case-> ((Matrix Flonum)         -> (Matrix Flonum))
        ((Matrix Real)           -> (Matrix Real))
        ((Matrix Float-Complex) -> (Matrix Float-Complex))
        ((Matrix Number)         -> (Matrix Number)))
```

but is documented as having the type `((Matrix Number) -> (Matrix Number))`.

Precise function types allow Typed Racket to prove more facts about `math/matrix` client programs. In particular, it is usually easy for it to prove that operations on real matrices return real matrices:

```
> (matrix-conjugate (matrix [[1 2 3] [4 5 6]]))
- : (Array Real)
(array #[#[1 2 3] #[4 5 6]])
```

and that operations on inexact matrices return inexact matrices:

```
> (matrix-conjugate (matrix [[1.0+2.0i 2.0+3.0i 3.0+4.0i]
                             [4.0+5.0i 5.0+6.0i 6.0+7.0i]]))
- : (Array Float-Complex)
(array #[#[1.0-2.0i 2.0-3.0i 3.0-4.0i] #[4.0-5.0i 5.0-6.0i 6.0-
7.0i]])
```

7.1.2 Failure Arguments

In many matrix operations, such as inversion, failure is easy to detect during computation, but is just as expensive to detect ahead of time as the operation itself. In these cases, the functions implementing the operations accept an optional *failure thunk*, or a zero-argument function that returns the result of the operation in case of failure.

For example, the (simplified) type of `matrix-inverse` is

```
(All (F) (case-> ((Matrix Number)         -> (Matrix Number))
                 ((Matrix Number) (-> F) -> (U F (Matrix Number)))))
```

Thus, if a failure thunk is given, the call site is required to check for return values of type `F` explicitly.

Default failure thunks usually raise an error, and have the type `(-> Nothing)`. For such failure thunks, `(U F (Matrix Number))` is equivalent to `(Matrix Number)`, because `Nothing` is part of every type. (In Racket, any expression may raise an error.) Thus, in this case,

no explicit test for values of type `F` is necessary (though of course they may be caught using `with-handlers` or similar).

7.1.3 Broadcasting

Unlike array operations, pointwise matrix operations **do not** broadcast their arguments when given matrices with different axis lengths:

```
> (matrix+ (identity-matrix 2) (matrix [[10]]))
matrix-map: matrices must have the same shape; given (array
#[#[1 0] #[0 1]]) (array #[#[10]])
```

If you need broadcasting, use array operations:

```
> (array+ (identity-matrix 2) (matrix [[10]]))
- : (Array Index)
(array #[#[11 10] #[10 11]])
```

7.1.4 Strictness

Functions exported by `math/matrix` return strict or nonstrict arrays based on the value of the `array-strictness` parameter. See §6.5 “Nonstrict Arrays” for details.

7.2 Types, Predicates and Accessors

```
(Matrix A)
```

Equivalent to `(Array A)`, but used for values `M` for which `(matrix? M)` is `#t`.

```
(matrix? arr) → Boolean
  arr : (Array A)
```

Returns `#t` when *arr* is a matrix: a nonempty array with exactly two axes.

Examples:

```
> (matrix? (array 10))
- : Boolean
#f
> (matrix? (array #[1 2 3]))
- : Boolean
```

```
#f
> (matrix? (make-array #(5 0) 0))
- : Boolean
#f
> (matrix? (array #[#[1 0] #[0 1]]))
- : Boolean
#t
```

```
(row-matrix? arr) → Boolean
  arr : (Array A)
```

Returns `#t` when `arr` is a *row matrix*: a matrix with exactly one row.

```
(col-matrix? arr) → Boolean
  arr : (Array A)
```

Returns `#t` when `arr` is a *column matrix*: a matrix with exactly one column.

```
(square-matrix? arr) → Boolean
  arr : (Array A)
```

Returns `#t` when `arr` is a matrix with the same number of rows and columns.

```
(matrix-shape M) → (Values Index Index)
  M : (Matrix A)
```

Returns `M`'s row and column count, respectively. Raises an error if `(matrix? M)` is `#f`.

Examples:

```
> (matrix-shape (row-matrix [1 2 3]))
- : (values Integer Integer) [more precisely: (Values Index In-
dex)]
1
3
> (matrix-shape (col-matrix [1 2 3]))
- : (values Integer Integer) [more precisely: (Values Index In-
dex)]
3
1
> (matrix-shape (identity-matrix 3))
- : (values Integer Integer) [more precisely: (Values Index In-
dex)]
3
3
```

```
(matrix-num-rows M) → Index
  M : (Matrix A)
```

Returns the number of rows in `M`, or the first value of `(matrix-shape M)`.

```
(matrix-num-cols M) → Index
  M : (Matrix A)
```

Returns the number of columns in `M`, or the second value of `(matrix-shape M)`.

```
(square-matrix-size M) → Index
  M : (Matrix A)
```

Returns the number of rows/columns in `M`. Raises an error if `(square-matrix? M)` is `#f`.

Examples:

```
> (square-matrix-size (identity-matrix 3))
- : Integer [more precisely: Index]
3
> (square-matrix-size (row-matrix [1 2 3]))
square-matrix-size: contract violation
  expected: square-matrix?
  given: (array #[#[1 2 3]])
```

7.3 Construction

```
(matrix [[expr ...+] ...+] maybe-type-ann)

maybe-type-ann =
               | : type
```

Like the `array` form for creating arrays, but does not require `#[...]` to delimit nested rows, and the result is constrained to be a `matrix?`.

Examples:

```
> (matrix [[1 2 3] [4 5 6]])
- : (Array Positive-Byte)
(array #[#[1 2 3] #[4 5 6]])
> (matrix [[1 2 3] [4 5 6]] : Number)
- : (Array Number)
(array #[#[1 2 3] #[4 5 6]])
```

```
> (matrix [[]])
eval:20:0: matrix: given empty row
  at: ()
  in: (matrix (()))
```

```
(row-matrix [expr ...+] maybe-type-ann)

maybe-type-ann =
               | : type
```

Like `matrix`, but returns a row matrix.

Examples:

```
> (row-matrix [1 2 3])
- : (Array Positive-Byte)
(array #[#[1 2 3]])
> (row-matrix [1 2 3] : Number)
- : (Array Number)
(array #[#[1 2 3]])
> (row-matrix [])
eval:23:0: row-matrix: given empty row
  at: ()
  in: (row-matrix ())
```

```
(col-matrix [expr ...+] maybe-type-ann)

maybe-type-ann =
               | : type
```

Like `matrix`, but returns a column matrix.

Examples:

```
> (col-matrix [1 2 3])
- : (Array Positive-Byte)
(array #[#[1] #[2] #[3]])
> (col-matrix [1 2 3] : Number)
- : (Array Number)
(array #[#[1] #[2] #[3]])
> (col-matrix [])
eval:26:0: col-matrix: given empty column
  at: ()
  in: (col-matrix ())
```

```
(identity-matrix n [one zero]) → (Matrix A)
  n : Integer
  one : A = 1
  zero : A = 0
```

Returns an $n \times n$ identity matrix, which has the value *one* on the diagonal and *zero* everywhere else. The height/width *n* must be positive.

Examples:

```
> (identity-matrix 3)
- : (Array (U One Zero))
(array #[#[1 0 0] #[0 1 0] #[0 0 1]])
> (identity-matrix 4 1.0+0.0i 0.0+0.0i)
- : (Array Float-Complex)
(array
 #[#[1.0+0.0i 0.0+0.0i 0.0+0.0i 0.0+0.0i]
   #[0.0+0.0i 1.0+0.0i 0.0+0.0i 0.0+0.0i]
   #[0.0+0.0i 0.0+0.0i 1.0+0.0i 0.0+0.0i]
   #[0.0+0.0i 0.0+0.0i 0.0+0.0i 1.0+0.0i]])
```

```
(make-matrix m n x) → (Matrix A)
  m : Integer
  n : Integer
  x : A
```

Returns an $m \times n$ matrix filled with the value *x*; both *m* and *n* must be positive. Analogous to `make-array` (and defined in terms of it).

```
(build-matrix m n proc) → (Matrix A)
  m : Integer
  n : Integer
  proc : (Index Index -> A)
```

Returns an $m \times n$ matrix with entries returned by *proc*; both *m* and *n* must be positive. Analogous to `build-array` (and defined in terms of it).

```
(diagonal-matrix xs [zero]) → (Matrix A)
  xs : (Listof A)
  zero : A = 0
```

Returns a matrix with *xs* along the diagonal and *zero* everywhere else. The length of *xs* must be positive.

Examples:

```
> (diagonal-matrix '(1 2 3 4 5 6))
- : (Array Byte)
(array
 #[#[1 0 0 0 0 0]
   #[0 2 0 0 0 0]
   #[0 0 3 0 0 0]
   #[0 0 0 4 0 0]
   #[0 0 0 0 5 0]
   #[0 0 0 0 0 6]])
> (diagonal-matrix '(1.0 2.0 3.0 4.0 5.0) 0.0)
- : (Array (U Flonum-Positive-Zero Positive-Float-No-NaN))
(array
 #[#[1.0 0.0 0.0 0.0 0.0]
   #[0.0 2.0 0.0 0.0 0.0]
   #[0.0 0.0 3.0 0.0 0.0]
   #[0.0 0.0 0.0 4.0 0.0]
   #[0.0 0.0 0.0 0.0 5.0]])
```

```
(block-diagonal-matrix Xs [zero]) → (Array A)
  Xs : (Listof (Array A))
  zero : A = 0
```

Returns an array with two-dimensional arrays *Xs* along the diagonal and *zero* everywhere else.

Wikipedia: Block-diagonal matrices

Examples:

```
> (block-diagonal-matrix (list (matrix [[6 7] [8 9]])
                               (diagonal-matrix '(7 5 7))
                               (col-matrix [1 2 3])
                               (row-matrix [4 5 6])))
- : (Array Byte)
(array
 #[#[6 7 0 0 0 0 0 0 0]
   #[8 9 0 0 0 0 0 0 0]
   #[0 0 7 0 0 0 0 0 0]
   #[0 0 0 5 0 0 0 0 0]
   #[0 0 0 0 7 0 0 0 0]
   #[0 0 0 0 0 1 0 0 0]
   #[0 0 0 0 0 2 0 0 0]
   #[0 0 0 0 0 3 0 0 0]
   #[0 0 0 0 0 0 4 5 6]])
> (block-diagonal-matrix (list (make-matrix 2 2 2.0+3.0i)
                               (make-matrix 2 2 5.0+7.0i))
                         0.0+0.0i)
```

```
- : (Array Float-Complex)
(array
 #[#[2.0+3.0i 2.0+3.0i 0.0+0.0i 0.0+0.0i]
   #[2.0+3.0i 2.0+3.0i 0.0+0.0i 0.0+0.0i]
   #[0.0+0.0i 0.0+0.0i 5.0+7.0i 5.0+7.0i]
   #[0.0+0.0i 0.0+0.0i 5.0+7.0i 5.0+7.0i]])
```

Empty two-dimensional arrays are valid inputs. They contribute to the resulting array's shape.

Examples:

```
> (block-diagonal-matrix (list (make-array #(2 0) 1)
                               (matrix [[6 7] [8 9]])))
- : (Array Byte)
(array #[#[0 0] #[0 0] #[6 7] #[8 9]])
> (block-diagonal-matrix (list (matrix [[6 7] [8 9]])
                               (make-array #(2 0) 1)))
- : (Array Byte)
(array #[#[6 7] #[8 9] #[0 0] #[0 0]])
> (block-diagonal-matrix (list (make-array #(0 2) 1)
                               (matrix [[6 7] [8 9]])))
- : (Array Byte)
(array #[#[0 0 6 7] #[0 0 8 9]])
> (block-diagonal-matrix (list (matrix [[6 7] [8 9]])
                               (make-array #(0 2) 1)))
- : (Array Byte)
(array #[#[6 7 0 0] #[8 9 0 0]])
> (block-diagonal-matrix (list (matrix [[6 7] [8 9]])
                               (make-array #(2 0) 1)
                               (diagonal-matrix '(7 5 7))
                               (make-array #(0 2) 1)
                               (col-matrix [1 2 3])
                               (row-matrix [4 5 6])))
- : (Array Byte)
(array
 #[#[6 7 0 0 0 0 0 0 0 0 0]
   #[8 9 0 0 0 0 0 0 0 0 0]
   #[0 0 0 0 0 0 0 0 0 0 0]
   #[0 0 0 0 0 0 0 0 0 0 0]
   #[0 0 7 0 0 0 0 0 0 0 0]
   #[0 0 0 5 0 0 0 0 0 0 0]
   #[0 0 0 0 7 0 0 0 0 0 0]
   #[0 0 0 0 0 0 0 1 0 0 0]
   #[0 0 0 0 0 0 0 2 0 0 0]
   #[0 0 0 0 0 0 0 3 0 0 0]
```

```
   #[0 0 0 0 0 0 0 0 4 5 6]])
> (block-diagonal-matrix (list (make-array #(2 0) 1)
                               (make-array #(0 3) 1)))
- : (Array (U One Zero))
(array #[#[0 0 0] #[0 0 0]])
> (block-diagonal-matrix (list (make-array #(0 3) 1)
                               (make-array #(2 0) 1)))
- : (Array (U One Zero))
(array #[#[0 0 0] #[0 0 0]])
```

If *Xs* is `null`, the result is an empty array with shape `#(0 0)`.

Example:

```
> (block-diagonal-matrix '())
- : (Array Nothing)
(array #[])
```

```
(vandermonde-matrix xs n) → (Matrix Number)
  xs : (Listof Number)
  n : Integer
```

Returns an `m`×*n* Vandermonde matrix, where `m = (length xs)`.

Wikipedia: Vandermonde matrix

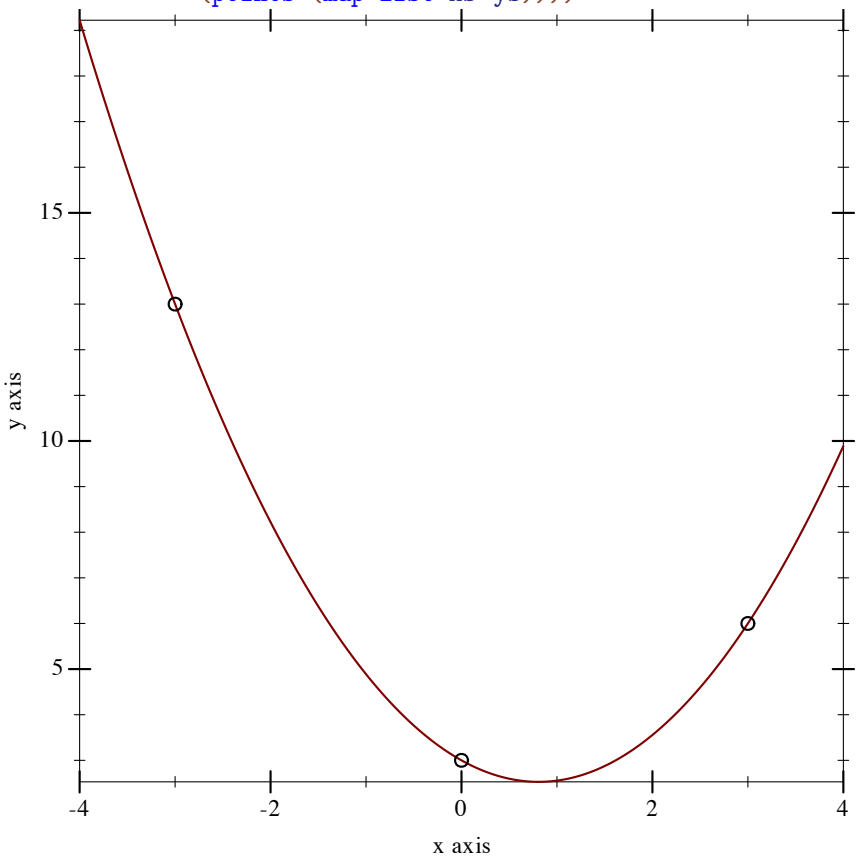
Examples:

```
> (vandermonde-matrix '(1 2 3 4) 5)
- : (Array Real)
(array #[#[1 1 1 1 1] #[1 2 4 8 16] #[1 3 9 27 81] #[1 4 16 64
256]])
> (vandermonde-matrix '(5.2 3.4 2.0) 3)
- : (Array Flonum)
(array
 #[#[1.0 5.2 27.040000000000003]
   #[1.0 3.4 11.559999999999999]
   #[1.0 2.0 4.0]])
```

Using a Vandermonde matrix to find a Lagrange polynomial (the polynomial of least degree that passes through a given set of points):

```
> (define (lagrange-polynomial xs ys)
    (array->list (matrix-solve (vandermonde-
matrix xs (length xs))
                               (->col-matrix ys))))
> (define xs '(-3 0 3))
```

```
> (define ys '(13 3 6))
> (match-define (list c b a) (lagrange-polynomial xs ys))
> (plot (list (function (λ (x) (+ c (* b x) (* a x x))) -4 4)
              (points (map list xs ys))))
```



Note that the above example is in untyped Racket.

This function is defined in terms of `array-axis-expand`.

```
(for/matrix: m n maybe-fill (for:-clause ...) maybe-type-ann
  body ...+)
```

```
(for*/matrix: m n maybe-fill (for:-clause ...) maybe-type-ann
  body ...+)

    maybe-fill =
               | #:fill fill

maybe-type-ann =
               | : body-type

  m : Integer
  n : Integer
  fill : body-type
```

Like `for/array:` and `for*/array:`, but for matrices. The only material difference is that the shape *m n* is required and must be positive.

```
(for/matrix m n maybe-fill (for-clause ...)
  body ...+)
(for*/matrix m n maybe-fill (for-clause ...)
  body ...+)
```

Untyped versions of the loop macros.

7.4 Conversion

```
(list->matrix m n xs) → (Matrix A)
  m : Integer
  n : Integer
  xs : (Listof A)
(matrix->list M) → (Listof A)
  M : (Matrix A)
```

Convert a flat list to an $m \times n$ matrix and back; both *m* and *n* must be positive, and `(* m n) = (length xs)`. The entries in *xs* are in row-major order.

Examples:

```
> (list->matrix 2 3 '(1 2 3 4 5 6))
- : (Array Positive-Byte)
(array #[#[1 2 3] #[4 5 6]])
> (matrix->list (matrix [[1 2] [3 4] [5 6]]))
- : (Listof Positive-Byte)
'(1 2 3 4 5 6)
```

```
(vector->matrix m n xs) → (Matrix A)
  m : Integer
  n : Integer
  xs : (Vectorof A)
(matrix->vector M) → (Vectorof A)
  M : (Matrix A)
```

Like `list->matrix` and `matrix->list`, but for vectors.

Examples:

```
> (vector->matrix 2 3 #(1 2 3 4 5 6))
- : (Mutable-Array Integer)
(mutable-array #[#[1 2 3] #[4 5 6]])
> (matrix->vector (matrix [[1 2] [3 4] [5 6]]))
- : (Vectorof Integer)
'#(1 2 3 4 5 6)
```

```
(->row-matrix xs) → (Matrix A)
  xs : (U (Listof A) (Vectorof A) (Array A))
(->col-matrix xs) → (Matrix A)
  xs : (U (Listof A) (Vectorof A) (Array A))
```

Convert a list, vector, or array into a row or column matrix. If *xs* is an array, it must be nonempty and **not** have more than one axis with length greater than `1`.

Examples:

```
> (->row-matrix '(1 2 3))
- : (Array Positive-Byte)
(array #[#[1 2 3]])
> (->row-matrix #(1 2 3))
- : (Array Positive-Byte)
(array #[#[1 2 3]])
> (->row-matrix (col-matrix [1 2 3]))
- : (Array Positive-Byte)
(array #[#[1 2 3]])
> (->col-matrix (array #[#[#[1]] #[#[2]] #[#[3]]]))
- : (Array Positive-Byte)
(array #[#[1] #[2] #[3]])
> (->col-matrix (matrix [[1 0] [0 1]]))
array->col-matrix: contract violation
  expected: nonempty Array with exactly one axis of length
>= 1
  given: (array #[#[1 0] #[0 1]])
```

```
(list*->matrix xss) → (Matrix A)
  xss : (Listof (Listof A))
(matrix->list* M) → (Listof (Listof A))
  M : (Matrix A)
```

Convert a list of lists of entries into a matrix and back.

Examples:

```
> (list*->matrix '((1 2 3) (4 5 6)))
- : (Array Positive-Byte)
(array #[#[1 2 3] #[4 5 6]])
> (matrix->list* (matrix [[1 2 3] [4 5 6]]))
- : (Listof (Listof Positive-Byte))
'((1 2 3) (4 5 6))
```

These functions are like `list*->array` and `array->list*`, but use a fixed-depth (i.e. non-recursive) list type, and do not require a predicate to distinguish entries from rows.

```
(vector*->matrix xss) → (Matrix A)
  xss : (Vectorof (Vectorof A))
(matrix->vector* M) → (Vectorof (Vectorof A))
  M : (Matrix A)
```

Like `list*->matrix` and `matrix*->list`, but for vectors.

Examples:

```
> ((inst vector*->matrix Integer) #(#(1 2 3) #(4 5 6)))
- : (Mutable-Array Integer)
(mutable-array #[#[1 2 3] #[4 5 6]])
> (matrix->vector* (matrix [[1 2 3] [4 5 6]]))
- : (Vectorof (Vectorof Integer))
'#(#(1 2 3) #(4 5 6))
```

As in the first example, Typed Racket often needs help inferring the type `A`.

7.5 Entrywise Operations and Arithmetic

```
(matrix+ M N ...) → (Matrix Number)
  M : (Matrix Number)
  N : (Matrix Number)
```

```
(matrix- M N ...) → (Matrix Number)
  M : (Matrix Number)
  N : (Matrix Number)
(matrix* M N ...) → (Matrix Number)
  M : (Matrix Number)
  N : (Matrix Number)
```

Matrix addition, subtraction and products respectively.

For matrix addition and subtraction all matrices must have the same shape.

For matrix product the number of columns of one matrix must equal the number of rows in the following matrix.

Examples:

```
> (define A (matrix ([1 2]
                     [3 4])))
> (define B (matrix ([5 6]
                     [7 8])))
> (define C (matrix ([ 9 10 11]
                     [12 13 14])))
> (matrix+ A B)
(array #[#[6 8] #[10 12]])
> (matrix- A B)
(array #[#[-4 -4] #[-4 -4]])
> (matrix* A C)
(array #[#[33 36 39] #[75 82 89]])
```

```
(matrix-expt M n) → (Matrix Number)
  M : (Matrix Number)
  n : Integer
```

Computes (matrix* M ...) with n arguments, but more efficiently. M must be a square-matrix? and n must be nonnegative.

Examples:

```
> (matrix* (matrix-expt (matrix [[1 1] [1 0]]) 100)
           (col-matrix [0 1]))
(array #[#[354224848179261915075] #[218922995834555169026]])
> (->col-matrix (list (fibonacci 100) (fibonacci 99)))
(array #[#[354224848179261915075] #[218922995834555169026]])
```

```
(matrix-scale M z) → (Matrix Number)
  M : (Matrix Number)
  z : Number
```

Computes the matrix `zM`, a matrix of the same shape as *M* where each entry in *M* is multiplied with *z*.

Example:

```
> (matrix-scale (matrix [[1 2] [3 4]]) 2)
(array #[#[2 4] #[6 8]])
```

```
(matrix-kronecker M N ...) → (Matrix Number)
  M : (Matrix Number)
  N : (Matrix Number)
```

Computes the Kronecker product.

Wikipedia: Kronecker product

Example:

```
> (matrix-kronecker (matrix [[1 2] [3 4] [5 6]])
                    (matrix [[7 8] [9 10]]))
(mutable-array
 #[#[7 8 14 16]
   #[9 10 18 20]
   #[21 24 28 32]
   #[27 30 36 40]
   #[35 40 42 48]
   #[45 50 54 60]])
```

Added in version 1.2 of package `math-lib`.

```
(matrix-map f M) → (Matrix R)
  f : (A -> R)
  M : (Matrix A)
(matrix-map f M0 M1 N ...) → (Matrix R)
  f : (A B Ts ... -> R)
  M0 : (Matrix A)
  M1 : (Matrix B)
  N : (Matrix Ts)
```

Like `array-map`, but requires at least one matrix argument and never broadcasts.

Examples:

```
> (matrix-map sqr (matrix [[1 2] [3 4]]))
(array #[#[1 4] #[9 16]])
> (matrix-map  +  (matrix [[1 2] [3 4]])
                  (matrix [[5 6] [7 8]]))
```

```
(array #[#[6 8] #[10 12]])
```

```
(matrix-sum Ms) → (Matrix Number)
  Ms : (Listof (Matrix Number))
```

Like `(apply matrix+ Ms)`, but raises a runtime error when `Ms` is empty.

```
(matrix= M0 M1 N ...) → Boolean
  M0 : (Matrix Number)
  M1 : (Matrix Number)
  N : (Matrix Number)
```

Returns `#t` when its arguments are the same size and are equal entrywise.

See `matrix-relative-error` and `matrix-absolute-error` for equality testing that is tolerant to floating-point error.

7.6 Polymorphic Operations

```
(matrix-ref M i j) → A
  M : (Matrix A)
  i : Integer
  j : Integer
```

Returns the entry on row `i` and column `j`.

Examples:

```
> (define M (matrix ([1 2 3] [4 5 6])))
> (matrix-ref M 0 2)
3
> (matrix-ref M 1 2)
6
```

```
(matrix-row M i) → (Matrix A)
  M : (Matrix A)
  i : Integer
(matrix-col M j) → (Matrix A)
  M : (Matrix A)
  j : Integer
```

Returns the `i`th row or `j`th column as a matrix.

Examples:

```
> (define M (matrix ([1 2 3] [4 5 6])))
> (matrix-row M 1)
(array #[#[4 5 6]])
> (matrix-col M 0)
(array #[#[1] #[4]])
```

```
(submatrix M is js) → (Array A)
  M : (Matrix A)
  is : (U Slice (Sequenceof Integer))
  js : (U Slice (Sequenceof Integer))
```

Returns a submatrix or subarray of *M*, where *is* and *js* specify respectively the rows and columns to keep. Like array-slice-ref, but constrained so the result has exactly two axes.

Examples:

```
> (submatrix (identity-matrix 5) (:: 1 #f 2) (::))
- : (Array (U One Zero))
(array #[#[0 1 0 0 0] #[0 0 0 1 0]])
> (submatrix (identity-matrix 5) '() '(1 2 4))
- : (Array (U One Zero))
(array #[])
```

Note that submatrix may return an empty array, which is not a matrix.

```
(matrix-diagonal M) → (Array A)
  M : (Matrix A)
```

Returns array of the entries on the diagonal of *M*.

Example:

```
> (matrix-diagonal
   (matrix ([1 2 3] [4 5 6] [7 8 9])))
(array #[1 5 9])
```

```
(matrix-upper-triangle M [zero]) → (Matrix A)
  M : (Matrix A)
  zero : A = 0
(matrix-lower-triangle M [zero]) → (Matrix A)
  M : (Matrix A)
  zero : A = 0
```

The function matrix-upper-triangle returns an upper triangular matrix (entries below the diagonal have the value *zero*) with entries from the given matrix. Likewise the function matrix-lower-triangle returns a lower triangular matrix.

Examples:

```
> (define M (array+ (array 1) (axis-index-array #(5 7) 1)))
> M
- : (Array Positive-Fixnum)
(array
 #[#[1 2 3 4 5 6 7]
   #[1 2 3 4 5 6 7]
   #[1 2 3 4 5 6 7]
   #[1 2 3 4 5 6 7]
   #[1 2 3 4 5 6 7]])
> (matrix-upper-triangle M)
- : (Array Nonnegative-Fixnum)
(array
 #[#[1 2 3 4 5 6 7]
   #[0 2 3 4 5 6 7]
   #[0 0 3 4 5 6 7]
   #[0 0 0 4 5 6 7]
   #[0 0 0 0 5 6 7]])
> (matrix-lower-triangle M)
- : (Array Nonnegative-Fixnum)
(array
 #[#[1 0 0 0 0 0 0]
   #[1 2 0 0 0 0 0]
   #[1 2 3 0 0 0 0]
   #[1 2 3 4 0 0 0]
   #[1 2 3 4 5 0 0]])
> (matrix-lower-triangle (array->flarray M) 0.0)
- : (Array Flonum)
(array
 #[#[1.0 0.0 0.0 0.0 0.0 0.0 0.0]
   #[1.0 2.0 0.0 0.0 0.0 0.0 0.0]
   #[1.0 2.0 3.0 0.0 0.0 0.0 0.0]
   #[1.0 2.0 3.0 4.0 0.0 0.0 0.0]
   #[1.0 2.0 3.0 4.0 5.0 0.0 0.0]])
```

```
(matrix-rows M) → (Listof (Matrix A))
  M : (Matrix A)
(matrix-cols M) → (Listof (Matrix A))
  M : (Matrix A)
```

The functions respectively returns a list of the rows or columns of the matrix.

Examples:

```
> (define M (matrix ([1 2 3] [4 5 6])))
> (matrix-rows M)
(list (array #[#[1 2 3]]) (array #[#[4 5 6]]))
> (matrix-cols M)
(list (array #[#[1] #[4]]) (array #[#[2] #[5]]) (array #[#[3]
#[6]]))
```

```
(matrix-augment Ms) → (Matrix A)
  Ms : (Listof (Matrix A))
(matrix-stack Ms) → (Matrix A)
  Ms : (Listof (Matrix A))
```

The function matrix-augment returns a matrix whose columns are the columns of the matrices in *Ms*. The matrices in list must have the same number of rows.

The function matrix-stack returns a matrix whose rows are the rows of the matrices in *Ms*. The matrices in list must have the same number of columns.

Examples:

```
> (define M0 (matrix ([1 1] [1 1])))
> (define M1 (matrix ([2 2] [2 2])))
> (define M2 (matrix ([3 3] [3 3])))
> (matrix-augment (list M0 M1 M2))
(array #[#[1 1 2 2 3 3] #[1 1 2 2 3 3]])
> (matrix-stack (list M0 M1 M2))
(array #[#[1 1] #[1 1] #[2 2] #[2 2] #[3 3] #[3 3]])
```

```
(matrix-set-col M idx new-col) → (Matrix A)
  M : (Matrix A)
  idx : Integer
  new-col : (Matrix A)
(matrix-set-row M idx new-row) → (Matrix A)
  M : (Matrix A)
  idx : Integer
  new-row : (Matrix A)
```

The function matrix-set-col returns a matrix whose *idx*th column is *new-col*. The function matrix-set-row returns a matrix whose *idx*th row is *new-row*.

Examples:

```
> (define mat (matrix [[1 2 3]
```

```
                      [4 5 6]
                      [7 8 9]]))
> (define new-col (col-matrix [-1 -2 -3]))
> (define new-row (row-matrix [-1 -2 -3]))
> (matrix-set-col mat 0 new-col)
(array #[#[-1 2 3] #[-2 5 6] #[-3 8 9]])
> (matrix-set-row mat 0 new-row)
(array #[#[-1 -2 -3] #[4 5 6] #[7 8 9]])
```

```
(matrix-map-rows f M) → (Matrix B)
  f : ((Matrix A) -> (Matrix B))
  M : (Matrix A)
(matrix-map-rows f M fail) → (U F (Matrix B))
  f : ((Matrix A) -> (U #f (Matrix B)))
  M : (Matrix A)
  fail : (-> F)
```

The two-argument case applies the function `f` to each row of `M`. If the rows are called `r0`, `r1`, ..., the result matrix has the rows `(f r0)`, `(f r1)`, ...

Examples:

```
> (define M (matrix ([1 2 3] [4 5 6] [7 8 9] [10 11 12])))
> (define (double-row r) (matrix-scale r 2))
> (matrix-map-rows double-row M)
(array #[#[2 4 6] #[8 10 12] #[14 16 18] #[20 22 24]])
```

In the three argument case, if `f` returns `#f`, the result of `(fail)` is returned:

```
> (define Z (make-matrix 4 4 0))
> Z
- : (Array Zero)
(array #[#[0 0 0 0] #[0 0 0 0] #[0 0 0 0] #[0 0 0 0]])
> (matrix-map-rows (λ: ([r : (Matrix Real)])
                     (matrix-normalize r 2 (λ () #f)))
                   Z
                   (λ () 'FAILURE))
- : (U 'FAILURE (Array Real))
'FAILURE
```

```
(matrix-map-cols f M) → (Matrix B)
  f : ((Matrix A) -> (Matrix B))
  M : (Matrix A)
(matrix-map-cols f M fail) → (U F (Matrix B))
  f : ((Matrix A) -> (U #f (Matrix B)))
  M : (Matrix A)
  fail : (-> F)
```

Like matrix-map-rows, but maps f over columns.

7.7 Basic Operations

```
(matrix-conjugate M) → (Matrix Number)
  M : (Matrix Number)
```

Returns a matrix where each entry of the given matrix is conjugated.

Wikipedia: Conjugate

Example:

```
> (matrix-conjugate (matrix ([1 0+1i] [-1 2+1i])))
(array #[#[1 0-1i] #[-1 2-1i]])
```

```
(matrix-transpose M) → (Matrix A)
  M : (Matrix A)
(matrix-hermitian M) → (Matrix Number)
  M : (Matrix Number)
```

Returns the transpose or the hermitian of the matrix. The hermitian of a matrix is the conjugate of the transposed matrix. For a real matrix these operations return the the same result.

Wikipedia: Transpose, Hermitian

Examples:

```
> (matrix-transpose (matrix ([1 1] [2 2] [3 3])))
(array #[#[1 2 3] #[1 2 3]])
> (matrix-hermitian (matrix ([1 0+1i] [2 0+2i] [3 0+3i])))
(array #[#[1 2 3] #[0-1i 0-2i 0-3i]])
```

```
(matrix-trace M) → Number
  M : (Matrix Number)
```

Returns the trace of the square matrix. The trace of matrix is the the sum of the diagonal entries.

Wikipedia: Trace

Example:

```
> (matrix-trace (matrix ([1 2] [3 4])))
5
```

7.8 Inner Product Space Operations

The following functions treat matrices as vectors in an inner product space. It often makes most sense to use these vector-space functions only for row matrices and column matrices, which are essentially vectors as we normally think of them. There are exceptions, however, such as the fact that the Frobenius or Euclidean norm (implemented by `matrix-2norm`) can be used to measure error between matrices in a way that meets certain reasonable criteria (specifically, it is submultiplicative).

See §7.12 "Operator Norms and Comparing Matrices" for similar functions (e.g. norms and angles) defined by considering matrices as operators between inner product spaces consisting of column matrices.

```
(matrix-1norm M) → Nonnegative-Real
  M : (Matrix Number)
(matrix-2norm M) → Nonnegative-Real
  M : (Matrix Number)
(matrix-inf-norm M) → Nonnegative-Real
  M : (Matrix Number)
(matrix-norm M [p]) → Nonnegative-Real
  M : (Matrix Number)
  p : Real = 2
```

Respectively compute the L_1 norm, L_2 norm, L_∞, and L_p norm. Wikipedia: Norm

The L_1 norm is also known under the names Manhattan or taxicab norm. The L_1 norm of a matrix is the sum of magnitudes of the entries in the matrix.

The L_2 norm is also known under the names Euclidean or Frobenius norm. The L_2 norm of a matrix is the square root of the sum of squares of magnitudes of the entries in the matrix.

The L_∞ norm is also known as the maximum or infinity norm. The L_∞ norm computes the maximum magnitude of the entries in the matrix.

For `p >= 1`, `matrix-norm` computes the L_p norm: the pth root of the sum of all entry magnitudes to the pth power.

Examples:

```
> (matrix-1norm     (col-matrix [1 2]))
3
> (matrix-2norm     (col-matrix [1 2]))
2.23606797749979
> (matrix-inf-norm (col-matrix [1 2]))
2
> (matrix-norm      (col-matrix [1 2]) 3)
2.080083823051904
```

```
> (matrix-norm     (col-matrix [1 2]) +inf.0)
2
```

```
(matrix-dot M) → Nonnegative-Real
  M : (Matrix Number)
(matrix-dot M N) → Number
  M : (Matrix Number)
  N : (Matrix Number)
```

The call `(matrix-dot M N)` computes the Frobenius inner product of the two matrices with the same shape. In other words the sum of `(* a (conjugate b))` is computed where `a` runs over the entries in `M` and `b` runs over the corresponding entries in `N`.

The call `(matrix-dot M)` computes `(matrix-dot M M)` efficiently.

Examples:

```
> (matrix-dot (col-matrix [1 2]) (col-matrix [3 4]))
11
> (+ (* 1 3) (* 2 4))
11
```

```
(matrix-cos-angle M N) → Number
  M : (Matrix Number)
  N : (Matrix Number)
```

Returns the cosine of the angle between two matrices w.r.t. the inner produce space induced by the Frobenius inner product. That is it returns

```
(/ (matrix-dot M N) (* (matrix-2norm M) (matrix-2norm N)))
```

Examples:

```
> (define M (col-matrix [1 0]))
> (define N (col-matrix [0 1]))
> (matrix-cos-angle M N)
0
> (matrix-cos-angle M (matrix+ M N))
0.7071067811865475
```

```
(matrix-angle M N) → Number
  M : (Matrix Number)
  N : (Matrix Number)
```

Equivalent to `(acos (matrix-cos-angle M N))`.

Examples:

```
> (require (only-in math/base radians->degrees))
> (define M (col-matrix [1 0]))
> (define N (col-matrix [0 1]))
> (radians->degrees (matrix-angle M N))
90.0
> (radians->degrees (matrix-angle M (matrix+ M N)))
45.00000000000001
```

```
(matrix-normalize M [p fail]) → (U F (Matrix Number))
  M : (Matrix Number)
  p : Real = 2
  fail : (-> F) = (λ () (error ...))
```

Normalizes `M` with respect to the L_p norm.

Examples:

```
> (matrix-normalize (col-matrix [1 1]))
- : (Array Real)
(array #[#[0.7071067811865475] #[0.7071067811865475]])
> (matrix-normalize (col-matrix [1 1]) 1)
- : (Array Real)
(array #[#[1/2] #[1/2]])
> (matrix-normalize (col-matrix [1 1]) +inf.0)
- : (Array Real)
(array #[#[1] #[1]])
```

The result of applying the failure thunk `fail` is returned if `M`'s norm is zero.

```
(matrix-normalize-rows M [p fail]) → (Matrix Number)
  M : (Matrix Number)
  p : Real = 2
  fail : (-> F) = (λ () (error ...))
(matrix-normalize-cols M [p fail]) → (Matrix Number)
  M : (Matrix Number)
  p : Real = 2
  fail : (-> F) = (λ () (error ...))
```

As `matrix-normalize` but each row or column is normalized separately. The result is a matrix with unit vectors as rows or columns.

Examples:

```
> (matrix-normalize-rows (matrix [[1 2] [2 4]]))
- : (Array Real)
```

```
(array
 #[#[0.4472135954999579 0.8944271909999159]
   #[0.4472135954999579 0.8944271909999159]])
> (matrix-normalize-cols (matrix [[1 2] [2 4]]))
- : (Array Real)
(array
 #[#[0.4472135954999579 0.4472135954999579]
   #[0.8944271909999159 0.8944271909999159]])
```

The result of applying the failure thunk *fail* is returned if the norm of any row or column in *M* is zero.

```
(matrix-rows-orthogonal? M [eps]) → Boolean
  M : (Matrix Number)
  eps : Real = (* 10 epsilon.0)
(matrix-cols-orthogonal? M [eps]) → Boolean
  M : (Matrix Number)
  eps : Real = (* 10 epsilon.0)
```

Returns `#t` if the rows or columns of *M* are very close of being orthogonal (by default a few epsilons).

Examples:

```
> (matrix-rows-orthogonal? (matrix [[1 1] [-1 1]]))
#t
> (matrix-cols-orthogonal? (matrix [[1 1] [-1 1]]))
#t
```

7.9 Solving Systems of Equations

```
(matrix-solve M B [fail]) → (U F (Matrix Number))
  M : (Matrix Number)
  B : (Matrix Number)
  fail : (-> F) = (λ () (error ...))
```

Returns the matrix `X` for which `(matrix* M X)` is *B*. *M* and *B* must have the same number of rows.

It is typical for *B* (and thus `X`) to be a column matrix, but not required. If *B* is not a column matrix, `matrix-solve` solves for all the columns in *B* simultaneously.

Examples:

```
> (define M (matrix [[7 5] [3 -2]]))
> (define B0 (col-matrix [3 22]))
> (define B1 (col-matrix [19 4]))
> (matrix-solve M B0)
- : (Array Real)
(array #[#[4] #[-5]])
> (matrix* M (col-matrix [4 -5]))
- : (Array Integer)
(array #[#[3] #[22]])
> (matrix-solve M B1)
- : (Array Real)
(array #[#[2] #[1]])
> (matrix-cols (matrix-solve M (matrix-augment (list B0 B1))))
- : (Listof (Array Real))
(list (array #[#[4] #[-5]]) (array #[#[2] #[1]]))
```

`matrix-solve` does not solve overconstrained or underconstrained systems, meaning that *M* must be invertible. If *M* is not invertible, the result of applying the failure thunk *fail* is returned.

`matrix-solve` is implemented using `matrix-gauss-elim` to preserve exactness in its output, with partial pivoting for greater numerical stability when *M* is not exact.

See `vandermonde-matrix` for an example that uses `matrix-solve` to compute Legendre polynomials.

```
(matrix-inverse M [fail]) → (U F (Matrix Number))
  M : (Matrix Number)
  fail : (-> F) = (λ () (error ...))
```

Returns the inverse of *M* if it exists; otherwise returns the result of applying the failure thunk *fail*.

Wikipedia: Invertible Matrix

Examples:

```
> (matrix-inverse (identity-matrix 3))
- : (Array Real)
(array #[#[1 0 0] #[0 1 0] #[0 0 1]])
> (matrix-inverse (matrix [[7 5] [3 -2]]))
- : (Array Real)
(array #[#[2/29 5/29] #[3/29 -7/29]])
> (matrix-inverse (matrix [[1 2] [10 20]]))
matrix-inverse: contract violation
  expected: matrix-invertible?
  given: (array #[#[1 2] #[10 20]])
> (matrix-inverse (matrix [[1 2] [10 20]]) (λ () #f))
```

```
- : (U (Array Real) False)
#f
```

```
(matrix-invertible? M) → Boolean
  M : (Matrix Number)
```

Returns `#t` when `M` is a `square-matrix?` and `(matrix-determinant M)` is nonzero.

```
(matrix-determinant M) → Number
  M : (Matrix Number)
```

Returns the determinant of `M`, which must be a `square-matrix?`.

Wikipedia: Determinant

Examples:

```
> (matrix-determinant (diagonal-matrix '(1 2 3 4)))
- : Real
24
> (* 1 2 3 4)
- : Integer [more precisely: Positive-Integer]
24
> (matrix-determinant (matrix [[1 2] [10 20]]))
- : Real
0
> (matrix-determinant (col-matrix [1 2]))
square-matrix-size: contract violation
  expected: square-matrix?
  given: (array #[#[1] #[2]])
```

7.10 Row-Based Algorithms

```
(matrix-gauss-elim M
                   [jordan?
                    unitize-pivot?
                    pivoting])
 → (Values (Matrix Number) (Listof Index))
  M : (Matrix Number)
  jordan? : Any = #f
  unitize-pivot? : Any = #f
  pivoting : (U 'first 'partial) = 'partial
```

Implements Gaussian elimination or Gauss-Jordan elimination.

Wikipedia: Gaussian elimination, Gauss-Jordan elimination

If *jordan?* is true, row operations are done both above and below the pivot. If *unitize-pivot?* is true, the pivot's row is scaled so that the pivot value is 1. When both are true, the algorithm is called Gauss-Jordan elimination, and the result matrix is in *reduced* row echelon form.

If *pivoting* is 'first, the first nonzero entry in the current column is used as the pivot. If *pivoting* is 'partial, the largest-magnitude nonzero entry is used, which improves numerical stability on average when *M* contains inexact entries.

The first return value is the result of Gaussian elimination.

The second return value is a list of indexes of columns that did not have a nonzero pivot.

See matrix-row-echelon for examples.

```
(matrix-row-echelon  M
                     [jordan?
                      unitize-pivot?
                      pivoting])      → (Matrix Number)
  M : (Matrix Number)
  jordan? : Any = #f
  unitize-pivot? : Any = #f
  pivoting : (U 'first 'partial) = 'partial
```

Like matrix-gauss-elim, but returns only the result of Gaussian elimination.

Wikipedia: Row echelon form

Examples:

```
> (define M (matrix [[2 1 -1] [-3 -1 2] [-2 1 2]]))
> (matrix-row-echelon M)
- : (Array Real)
(mutable-array #[#[-3 -1 2] #[0 5/3 2/3] #[0 0 1/5]])
> (matrix-row-echelon M #t)
- : (Array Real)
(mutable-array #[#[-3 0 0] #[0 5/3 0] #[0 0 1/5]])
> (matrix-identity? (matrix-row-echelon M #t #t))
- : Boolean
#t
```

The last example shows that *M* is invertible.

Using matrix-row-echelon to solve a system of linear equations (without checking for failure):

```
> (define B (col-matrix [8 -11 -3]))
> (define MB (matrix-augment (list M B)))
> (matrix-col (matrix-row-echelon MB #t #t) 3)
```

```
- : (Array Real)
(array #[#[2] #[3] #[-1]])
> (matrix-solve M B)
- : (Array Real)
(array #[#[2] #[3] #[-1]])
```

Using `matrix-row-echelon` to invert a matrix (also without checking for failure):

```
> (define MI (matrix-augment (list M (identity-matrix 3))))
> (submatrix (matrix-row-echelon MI  #t #t) (::) (:: 3 #f))
- : (Array Real)
(array #[#[4 3 -1] #[-2 -2 1] #[5 4 -1]])
> (matrix-inverse M)
- : (Array Real)
(array #[#[4 3 -1] #[-2 -2 1] #[5 4 -1]])
```

```
(matrix-lu M [fail])
 → (Values (U F (Matrix Number)) (Matrix Number))
  M : (Matrix Number)
  fail : (-> F) = (λ () (error ...))
```

Returns the LU decomposition of *M* (which must be a `square-matrix?`) if one exists. An LU decomposition exists if *M* can be put in row-echelon form without swapping rows.

Wikipedia: LU decomposition

Because `matrix-lu` returns a *unit* lower-triangular matrix (i.e. a lower-triangular matrix with only ones on the diagonal), the decomposition is unique if it exists.

Examples:

```
> (define-values (L U)
    (matrix-lu (matrix [[4 3] [6 3]])))
> (values L U)
- : (values (Array Real) (Array Real))
(mutable-array #[#[1 0] #[3/2 1]])
(mutable-array #[#[4 3] #[0 -3/2]])
> (matrix* L U)
- : (Array Real)
(array #[#[4 3] #[6 3]])
```

If *M* does not have an LU decomposition, the first result is the result of applying the failure thunk *fail*, and the second result is the original argument *M*:

```
> (matrix-lu (matrix [[0 1] [1 1]]))
matrix-lu: contract violation
  expected: LU-decomposable matrix
```

```
  given: (array #[#[0 1] #[1 1]])
> (matrix-lu (matrix [[0 1] [1 1]]) (λ () #f))
- : (values (U (Array Real) False) (Array Real))
#f
(array #[#[0 1] #[1 1]])
```

7.11 Orthogonal Algorithms

```
(matrix-gram-schmidt M [normalize? start-col]) → (Array Number)
  M : (Matrix Number)
  normalize? : Any = #f
  start-col : Integer = 0
```

Returns an array whose columns are orthogonal and span the same subspace as *M*'s columns. The number of columns in the result is the rank of *M*. If *normalize?* is true, the columns are also normalized.

Wikipedia: Gram-Schmidt process

Examples:

```
> (define M
    (matrix [[12 -51   4]
             [ 6 167 -68]
             [-4  24 -41]]))
> (matrix-gram-schmidt M)
- : (Array Real)
(array #[#[12 -69 -58/5] #[6 158 6/5] #[-4 30 -33]])
> (matrix-gram-schmidt M #t)
- : (Array Real)
(array #[#[6/7 -69/175 -58/175] #[3/7 158/175 6/175] #[-2/7 6/35
-33/35]])
> (matrix-cols-orthogonal? (matrix-gram-schmidt M))
- : Boolean
#t
> (matrix-orthonormal? (matrix-gram-schmidt M #t))
- : Boolean
#t
```

Examples with rank-deficient matrices:

```
> (matrix-gram-schmidt (matrix [[ 1 -2 1]
                                [-2  4 9]
                                [ 3 -6 7]]))
- : (Array Real)
(array #[#[1 5/7] #[-2 67/7] #[3 43/7]])
```

```
> (matrix-gram-schmidt (make-matrix 3 3 0))
- : (Array Real)
(array #[#[] #[] #[]])
```

When *start-col* is positive, the Gram-Schmidt process is begun on column *start-col* (but still using the previous columns to orthogonalize the remaining columns). This feature is generally not directly useful, but is used in the implementation of matrix-basis-extension.

While Gram-Schmidt with inexact matrices is known to be unstable, using it twice tends to remove instabilities:*

* **On the round-off error analysis of the Gram-Schmidt algorithm with reorthogonalization.**, Luc Giraud, Julien Langou and Miroslav Rozloznik. 2002. (PDF)

```
> (define M (matrix [[0.7 0.70711]
                     [0.70001 0.70711]]))
> (matrix-orthonormal?
   (matrix-gram-schmidt M #t))
- : Boolean
#f
> (matrix-orthonormal?
   (matrix-gram-schmidt (matrix-gram-schmidt M) #t))
- : Boolean
#t
```

```
(matrix-basis-extension M) → (Array Number)
  M : (Matrix Number)
```

Returns additional orthogonal columns which, if augmented with *M*, would result in an orthogonal matrix of full rank. If *M*'s columns are normalized, the result's columns are normalized.

```
(matrix-qr M) → (Values (Matrix Number) (Matrix Number))
  M : (Matrix Number)
(matrix-qr M full?) → (Values (Matrix Number) (Matrix Number))
  M : (Matrix Number)
  full? : Any
```

Computes a QR-decomposition of the matrix *M*. The values returned are the matrices Q and R. If *full?* is #f, then a reduced decomposition is returned, otherwise a full decomposition is returned.

Wikipedia: QR decomposition

The (full) decomposition of a square matrix consists of two matrices: a orthogonal matrix Q and an upper triangular matrix R, such that QR = *M*.

An *orthonormal* matrix has columns which are orthoginal, unit vectors.

For tall non-square matrices R, the triangular part of the full decomposition, contains zeros below the diagonal. The reduced decomposition leaves the zeros out. See the Wikipedia entry on QR decomposition for more details.

Examples:

```
> (define M
    (matrix [[12 -51   4]
             [ 6 167 -68]
             [-4  24 -41]]))
> (define-values (Q R) (matrix-qr M))
> (values Q R)
- : (values (Array Real) (Array Real))
(array #[#[6/7 -69/175 -58/175] #[3/7 158/175 6/175] #[-2/7 6/35
-33/35]])
(array #[#[14 21 -14] #[0 175 -70] #[0 0 35]])
> (matrix= M (matrix* Q R))
- : Boolean
#t
```

The difference between full and reduced decompositions:

```
> (define M
    (matrix [[12 -51]
             [ 6 167]
             [-4  24]]))
> (define-values (Q1 R1) (matrix-qr M #f))
> (define-values (Q2 R2) (matrix-qr M #t))
> (values Q1 R1)
- : (values (Array Real) (Array Real))
(array #[#[6/7 -69/175] #[3/7 158/175] #[-2/7 6/35]])
(array #[#[14 21] #[0 175]])
> (values Q2 R2)
- : (values (Array Real) (Array Real))
(array #[#[6/7 -69/175 58/175] #[3/7 158/175 -6/175] #[-2/7 6/35
33/35]])
(array #[#[14 21] #[0 175] #[0 0]])
> (matrix= M (matrix* Q1 R1))
- : Boolean
#t
> (matrix= M (matrix* Q2 R2))
- : Boolean
#t
```

The decomposition `M = QR` is useful for solving the equation `Mx=v`. Since the inverse of Q is simply the transpose of Q, `Mx=v <=> QRx=v <=> Rx = Q^T v`. And since `R` is upper triangular, the system can be solved by back substitution.

The algorithm used is Gram-Schmidt with reorthogonalization.

7.12 Operator Norms and Comparing Matrices

§7.8 "Inner Product Space Operations" describes functions that deal with matrices as vectors in an inner product space. This section describes functions that deal with matrices as *linear operators*, or as functions from column matrices to column matrices.

Wikipedia: Induced norm

In this setting, a norm is the largest relative change in magnitude an operator (i.e. matrix) can effect on a column matrix, where "magnitude" is defined by a vector norm. (See the Wikipedia article linked to in the margin for a formal definition.) Matrix norms that are defined in terms of a vector norm are called *induced norms*, or *operator norms*.

```
(matrix-op-1norm M) → Nonnegative-Real
  M : (Matrix Number)
```

The operator norm induced by the vector norm `matrix-1norm`.

When M is a column matrix, `(matrix-op-1norm M)` is equivalent to `(matrix-1norm M)`.

```
(matrix-op-2norm M) → Nonnegative-Real
  M : (Matrix Number)
```

The operator norm induced by the vector norm `matrix-2norm`.

This function is currently undefined because a required algorithm (singular value decomposition or eigendecomposition) is not yet implemented in `math/matrix`.

When M is a column matrix, `(matrix-op-2norm M)` is equivalent to `(matrix-2norm M)`.

```
(matrix-op-inf-norm M) → Nonnegative-Real
  M : (Matrix Number)
```

The operator norm induced by the vector norm `matrix-inf-norm`.

When M is a column matrix, `(matrix-op-inf-norm M)` is equivalent to `(matrix-inf-norm M)`.

```
(matrix-basis-cos-angle M0 M1) → Number
  M0 : (Matrix Number)
  M1 : (Matrix Number)
```

Returns the cosine of the angle between the two subspaces spanned by `M0` and `M1`.

This function is currently undefined because a required algorithm (singular value decomposition or eigendecomposition) is not yet implemented in `math/matrix`.

When `M0` and `M1` are column matrices, `(matrix-basis-cos-angle M0 M1)` is equivalent to `(matrix-cos-angle M0 M1)`.

```
(matrix-basis-angle M0 M1) → Number
  M0 : (Matrix Number)
  M1 : (Matrix Number)
```

Equivalent to `(acos (matrix-basis-cos-angle M0 M1))`.

The function is currently undefined because `matrix-basis-cos-angle` is currently undefined.

```
(matrix-error-norm) → ((Matrix Number) -> Nonnegative-Real)
(matrix-error-norm norm) → void?
  norm : ((Matrix Number) -> Nonnegative-Real)
```

The norm used by `matrix-relative-error` and `matrix-absolute-error`. The default value is `matrix-op-inf-norm`.

Besides being a true norm, `norm` should also be *submultiplicative*:

```
(norm (matrix* M0 M1)) <= (* (norm M0) (norm M1))
```

This additional triangle-like inequality makes it possible to prove error bounds for formulas that involve matrix multiplication.

All operator norms (`matrix-op-1norm`, `matrix-op-2norm`, `matrix-op-inf-norm`) are submultiplicative by definition, as is the Frobenius norm (`matrix-2norm`).

```
(matrix-absolute-error M R [norm]) → Nonnegative-Real
  M : (Matrix Number)
  R : (Matrix Number)
  norm : ((Matrix Number) -> Nonnegative-Real)
       = (matrix-error-norm)
```

Basically equivalent to `(norm (matrix- M R))`, but handles non-rational flonums like `+inf.0` and `+nan.0` specially.

See `absolute-error` for the scalar version of this function.

```
(matrix-relative-error M R [norm]) → Nonnegative-Real
  M : (Matrix Number)
  R : (Matrix Number)
  norm : ((Matrix Number) -> Nonnegative-Real)
       = (matrix-error-norm)
```

Measures the error in `M` relative to the true matrix `R`, under the norm `norm`. Basically equivalent to `(/ (norm (matrix- M R)) (norm R))`, but handles non-rational flonums like `+inf.0` and `+nan.0` specially, as well as the case `(norm R) = 0`.

See `relative-error` for the scalar version of this function.

```
(matrix-zero? M [eps]) → Boolean
  M : (Matrix Number)
  eps : Real = (* 10 epsilon.0)
```

Returns `#t` when *M* is very close to a zero matrix (by default, within a few epsilons). Equivalent to

```
(<= (matrix-absolute-error M (make-matrix m n 0)) eps)
```

where `m n` is the shape of *M*.

```
(matrix-identity? M [eps]) → Boolean
  M : (Matrix Number)
  eps : Real = (* 10 epsilon.0)
```

Returns `#t` when *M* is very close to the identity matrix (by default, within a few epsilons). Equivalent to

```
(and (square-matrix? M)
     (<= (matrix-relative-error M (identity-matrix (square-matrix-
size M)))
         eps))
```

```
(matrix-orthonormal? M [eps]) → Boolean
  M : (Matrix Number)
  eps : Real = (* 10 epsilon.0)
```

Returns `#t` when *M* is very close to being orthonormal; that is, when `(matrix* M (matrix-hermitian M))` is very close to an identity matrix. Equivalent to

```
(matrix-identity? (matrix* M (matrix-hermitian M)) eps)
```

8 Statistics Functions

```
(require math/statistics)
```
package: `math-lib`

This module exports functions that compute *statistics*, meaning summary values for collections of samples, and functions for managing sequences of weighted or unweighted samples.

Most of the functions that compute statistics accept a sequence of nonnegative reals that correspond one-to-one with sample values. These are used as weights; equivalently counts, pseudocounts or unnormalized probabilities. While this makes it easy to work with weighted samples, it introduces some subtleties in bias correction. In particular, central moments must be computed without bias correction by default. See §8.1 “Expected Values” for a discussion.

8.1 Expected Values

Functions documented in this section that compute higher central moments, such as `variance`, `stddev` and `skewness`, can optionally apply bias correction to their estimates. For example, when `variance` is given the argument `#:bias #t`, it multiplies the result by `(/ n (- n 1))`, where `n` is the number of samples.

The meaning of “bias correction” becomes less clear with weighted samples, however. Often, the weights represent counts, so when moment-estimating functions receive `#:bias #t`, they interpret it as “use the sum of `ws` for `n`.” In the following example, the sample `4` is first counted twice and then given weight `2`; therefore `n = 5` in both cases:

```
> (variance '(1 2 3 4 4) #:bias #t)
- : Real [more precisely: Nonnegative-Real]
17/10
> (variance '(1 2 3 4) '(1 1 1 2) #:bias #t)
- : Real [more precisely: Nonnegative-Real]
17/10
```

However, sample weights often do not represent counts. For these cases, the `#:bias` keyword can be followed by a real-valued pseudocount, which is used for `n`:

```
> (variance '(1 2 3 4) '(1/2 1/2 1/2 1) #:bias 5)
- : Real [more precisely: Nonnegative-Real]
17/10
```

Because the magnitude of the bias correction for weighted samples cannot be known without user guidance, in all cases, the bias argument defaults to `#f`.

```
(mean xs [ws]) → Real
```

```
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
```

When `ws` is `#f` (the default), returns the sample mean of the values in `xs`. Otherwise, returns the weighted sample mean of the values in `xs` with corresponding weights `ws`.

Examples:

```
> (mean '(1 2 3 4 5))
- : Real
3
> (mean '(1 2 3 4 5) '(1 1 1 1 10.0))
- : Real
4.285714285714286
> (define d (normal-dist))
> (mean (sample d 10000))
- : Real
0.004833164529803956
> (define arr (array-strict (build-array #(5 1000) (λ (_) (sample d)))))
> (array-map mean (array->list-array arr 1))
- : (Array Real)
(array
 #[-0.001922809501835001
   0.04097374534242923
   0.018194229442495848
   0.011536221344938015
   -0.018249748684997555])
```

```
(variance xs [ws #:bias bias]) → Nonnegative-Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
(stddev xs [ws #:bias bias]) → Nonnegative-Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
(skewness xs [ws #:bias bias]) → Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
(kurtosis xs [ws #:bias bias]) → Nonnegative-Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
```

If `ws` is `#f`, these compute the sample variance, standard deviation, skewness and excess

kurtosis the samples in *xs*. If *ws* is not #f, they compute weighted variations of the same.

Examples:

```
> (stddev '(1 2 3 4 5))
- : Real [more precisely: Nonnegative-Real]
1.4142135623730951
> (stddev '(1 2 3 4 5) '(1 1 1 1 10))
- : Real [more precisely: Nonnegative-Real]
1.2777531299998799
```

See §8.1 “Expected Values” for the meaning of the *bias* keyword argument.

```
(variance/mean m xs [ws #:bias bias]) → Nonnegative-Real
  m : Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
(stddev/mean m xs [ws #:bias bias]) → Nonnegative-Real
  m : Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
(skewness/mean m xs [ws #:bias bias]) → Real
  m : Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
(kurtosis/mean m xs [ws #:bias bias]) → Nonnegative-Real
  m : Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
```

Like variance, stddev, skewness and kurtosis, but computed using known mean *m*.

8.2 Running Expected Values

The statistics object allows computing the sample minimum, maximum, count, mean, variance, skewness, and excess kurtosis of a sequence of samples in O(1) space.

To use it, start with empty-statistics, then use update-statistics to obtain a new statistics object with updated values. Use statistics-min, statistics-mean, and similar functions to get the current estimates.

Example:

```
> (let* ([s  empty-statistics]
         [s  (update-statistics s 1)]
         [s  (update-statistics s 2)]
         [s  (update-statistics s 3)]
         [s  (update-statistics s 4 2)])
    (values (statistics-mean s)
            (statistics-stddev s #:bias #t)))
- : (values Flonum Flonum) [more precisely: (Values Flonum
Nonnegative-Flonum)]
2.8
1.3038404810405297
```

```
(struct statistics (min max count))
  min : Flonum
  max : Flonum
  count : Nonnegative-Flonum
```

Represents running statistics.

The `min` and `max` fields are the minimum and maximum value observed so far, and the `count` field is the total weight of the samples (which is the number of samples if all samples are unweighted). The remaining, hidden fields are used to compute moments, and their number and meaning may change in future releases.

```
empty-statistics : statistics
```

The empty statistics object.

Examples:

```
> (statistics-min empty-statistics)
- : Flonum
+inf.0
> (statistics-max empty-statistics)
- : Flonum
-inf.0
> (statistics-range empty-statistics)
- : Flonum [more precisely: Nonnegative-Flonum]
+nan.0
> (statistics-count empty-statistics)
- : Flonum [more precisely: Nonnegative-Flonum]
0.0
> (statistics-mean empty-statistics)
```

```
- : Flonum
+nan.0
> (statistics-variance empty-statistics)
- : Flonum [more precisely: Nonnegative-Flonum]
+nan.0
> (statistics-skewness empty-statistics)
- : Flonum
+nan.0
> (statistics-kurtosis empty-statistics)
- : Flonum [more precisely: Nonnegative-Flonum]
+nan.0
```

```
(update-statistics s x [w]) → statistics
  s : statistics
  x : Real
  w : Real = 1.0
```

Returns a new statistics object that includes `x` in the computed statistics. If `w` is given, `x` is weighted by `w` in the moment computations.

```
(update-statistics* s xs [ws]) → statistics
  s : statistics
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
```

Like `update-statistics`, but includes all of `xs`, possibly weighted by corresponding elements in `ws`, in the returned statistics object.

Examples:

```
> (define s (update-statistics* empty-statistics '(1 2 3 4) '(1 1 1 2)))
> (statistics-mean s)
- : Flonum
2.8
> (statistics-stddev s #:bias #t)
- : Flonum [more precisely: Nonnegative-Flonum]
1.3038404810405297
```

This function uses O(1) space regardless of the length of `xs`.

```
(statistics-range s) → Nonnegative-Flonum
  s : statistics
(statistics-mean s) → Flonum
  s : statistics
```

```
(statistics-variance s [#:bias bias]) → Nonnegative-Flonum
  s : statistics
  bias : (U #t #f Real) = #f
(statistics-stddev s [#:bias bias]) → Nonnegative-Flonum
  s : statistics
  bias : (U #t #f Real) = #f
(statistics-skewness s [#:bias bias]) → Flonum
  s : statistics
  bias : (U #t #f Real) = #f
(statistics-kurtosis s [#:bias bias]) → Nonnegative-Flonum
  s : statistics
  bias : (U #t #f Real) = #f
```

Compute the range, mean, variance, standard deviation, skewness, and excess kurtosis of the observations summarized in *s*.

See §8.1 “Expected Values” for the meaning of the *bias* keyword argument.

8.3 Correlation

```
(covariance xs ys [ws #:bias bias]) → Real
  xs : (Sequenceof Real)
  ys : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
(correlation xs ys [ws #:bias bias]) → Real
  xs : (Sequenceof Real)
  ys : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
```

Compute the sample covariance and correlation of *xs* and *ys*, optionally weighted by *ws*.

Examples:

```
> (define xs (sample (normal-dist) 10000))
> (define ys (map (λ: ([x : Real]) (sample (normal-dist x))) xs))
> (correlation xs ys)
- : Real
0.7023415652478404
```

Removing the correlation using importance weights:

```
> (define ws (map (λ: ([x : Real] [y : Real])
```

```
                    (/ (pdf (normal-dist) y)
                       (pdf (normal-dist x) y)))
                  xs ys))
> (correlation xs ys (ann ws (Sequenceof Real)))
- : Real
0.02023850842411348
```

See §8.1 “Expected Values” for the meaning of the *bias* keyword argument.

```
(covariance/means mx my xs ys [ws #:bias bias]) → Real
  mx : Real
  my : Real
  xs : (Sequenceof Real)
  ys : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
(correlation/means mx my xs ys [ws #:bias bias]) → Real
  mx : Real
  my : Real
  xs : (Sequenceof Real)
  ys : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
```

Like `covariance` and `correlation`, but computed using known means *mx* and *my*.

8.4 Counting and Binning

```
(samples->hash xs) → (HashTable A Positive-Integer)
  xs : (Sequenceof A)
(samples->hash xs ws) → (HashTable A Nonnegative-Real)
  xs : (Sequenceof A)
  ws : (U #f (Sequenceof Real))
```

Returns a hash table mapping each unique element in *xs* (under `equal?`) to its count, or, if *ws* is not `#f`, to its total weight.

Examples:

```
> (samples->hash '(1 2 3 4 4))
- : (HashTable Integer Positive-Integer)
'#hash((1 . 1) (2 . 1) (3 . 1) (4 . 2))
> (samples->hash '(1 1 2 3 4) '(1/2 1/2 1 1 2))
- : (HashTable Integer Nonnegative-Real)
'#hash((1 . 1) (2 . 1) (3 . 1) (4 . 2))
```

```
(count-samples xs)
 → (Values (Listof A) (Listof Positive-Integer))
  xs : (Sequenceof A)
(count-samples xs ws)
 → (Values (Listof A) (Listof Nonnegative-Real))
  xs : (Sequenceof A)
  ws : (U #f (Sequenceof Real))
```

Like `samples->hash`, but returns two lists. The elements in the returned `(Listof A)` are in order of first appearance in *xs*.

Examples:

```
> (count-samples '(1 2 3 4 4))
- : (values (Listof Positive-Byte) (Listof Positive-Integer))
'(1 2 3 4)
'(1 1 1 2)
> (count-samples '(1 1 2 3 4) '(1/2 1/2 1 1 2))
- : (values (Listof Positive-Byte) (Listof Nonnegative-Real))
'(1 2 3 4)
'(1 1 1 2)
```

```
(struct sample-bin (min max values weights))
  min : B
  max : B
  values : (Listof A)
  weights : (U #f (Listof Nonnegative-Real))
```

Represents a *bin*, or a group of samples within an interval in a total order. The values and bounds have a different type to allow `bin-samples/key` to group elements based on a function of their values.

```
(bin-samples bounds lte? xs ws) → (Listof (sample-bin A A))
  bounds : (Sequenceof A)
  lte? : (A A -> Any)
  xs : (Sequenceof A)
  ws : (U #f (Sequenceof Real))
```

Similar to `(sort xs lte?)`, but additionally groups samples into bins. The bins' *bounds* are sorted before binning *xs*.

If `n = (length bounds)`, then `bin-samples` returns *at least* `(- n 1)` bins, one for each pair of adjacent (sorted) bounds. If some values in *xs* are less than the smallest bound, they are grouped into a single bin in front. If some are greater than the largest bound, they are grouped into a single bin at the end.

Examples:

```
> (bin-samples '() <= '(0 1 2 3 4 5 6))
- : (Listof (sample-bin Byte Byte))
(list (sample-bin 0 6 '(0 1 2 3 4 5 6) #f))
> (bin-samples '(3) <= '(0 1 2 3 4 5 6))
- : (Listof (sample-bin Byte Byte))
(list (sample-bin 0 3 '(0 1 2 3) #f) (sample-bin 3 6 '(4 5 6) #f))
> (bin-samples '(2 4) <= '(0 1 2 3 4 5 6))
- : (Listof (sample-bin Byte Byte))
(list
 (sample-bin 0 2 '(0 1 2) #f)
 (sample-bin 2 4 '(3 4) #f)
 (sample-bin 4 6 '(5 6) #f))
> (bin-samples '(2 4) <=
               '(0 1 2 3 4 5 6)
               '(10 20 30 40 50 60 70))
- : (Listof (sample-bin Byte Byte))
(list
 (sample-bin 0 2 '(0 1 2) '(10 20 30))
 (sample-bin 2 4 '(3 4) '(40 50))
 (sample-bin 4 6 '(5 6) '(60 70)))
```

If *lte?* is a less-than-or-equal relation, the bins represent half-open intervals (min, max] (except possibly the first, which may be closed). If *lte?* is a less-than relation, the bins represent half-open intervals [min, max) (except possibly the last, which may be closed). In either case, the sorts applied to *bounds* and *xs* are stable.

Because intervals used in probability measurements are normally open on the left, prefer to use less-than-or-equal relations for *lte?*.

If *ws* is #f, bin-samples returns bins with #f weights.

```
(bin-samples/key bounds lte? key xs ws) → (Listof (sample-bin A B))
  bounds : (Sequenceof B)
  lte? : (B B -> Any)
  key : (A -> B)
  xs : (Sequenceof A)
  ws : (U #f (Sequenceof Real))
```

Similar to (sort *xs lte?* #:key *key* #:cache-keys? #t), but additionally groups samples into bins.

Example:

```
> (bin-samples/key '(2 4) <= (inst car Real String)
```

```
                    '((1 . "1") (2 . "2") (3 . "3") (4 . "4") (5 .
"5")))
- : (Listof (sample-bin (Pairof Positive-Byte String) Real))
(list
 (sample-bin 1 2 '((1 . "1") (2 . "2")) #f)
 (sample-bin 2 4 '((3 . "3") (4 . "4")) #f)
 (sample-bin 4 5 '((5 . "5")) #f))
```

See bin-samples for the simpler, one-type variant.

(sample-bin-compact *bin*) → (sample-bin A B)
  bin : (sample-bin A B)

Compacts *bin* by applying count-samples to its values and weights.

Example:

```
> (sample-bin-compact (sample-bin 1 4 '(1 2 3 4 4) #f))
- : (sample-bin Positive-Byte Positive-Byte)
(sample-bin 1 4 '(1 2 3 4) '(1 1 1 2))
```

(sample-bin-total *bin*) → Nonnegative-Real
  bin : (sample-bin A B)

If (sample-bin-weights *bin*) is #f, returns the number of samples in *bin*; otherwise, returns the sum of their weights.

Examples:

```
> (sample-bin-total (sample-bin 1 4 '(1 2 3 4 4) #f))
- : Real [more precisely: Nonnegative-Real]
5
> (sample-bin-total (sample-bin-compact (sample-
bin 1 4 '(1 2 3 4 4) #f)))
- : Real [more precisely: Nonnegative-Real]
5
```

8.5 Order Statistics

(sort-samples *lt?* *xs*) → (Listof A)
  lt? : (A A -> Any)
  xs : (Sequenceof A)
(sort-samples *lt?* *xs* *ws*)

```
→ (Values (Listof A) (Listof Nonnegative-Real))
 lt? : (A A -> Any)
 xs : (Sequenceof A)
 ws : (U #f (Sequenceof Real))
```

Sorts possibly weighted samples according to *lt?*, which is assumed to define a total order over the elements in *xs*.

Examples:

```
> (sort-samples < '(5 2 3 1))
- : (Listof Positive-Byte)
'(1 2 3 5)
> (sort-samples < '(5 2 3 1) '(50 20 30 10))
- : (values (Listof Positive-Byte) (Listof Nonnegative-Real))
'(1 2 3 5)
'(10 20 30 50)
> (sort-samples < '(5 2 3 1) #f)
- : (values (Listof Positive-Byte) (Listof Nonnegative-Real))
'(1 2 3 5)
'(1 1 1 1)
```

Because `sort-samples` is defined in terms of `sort`, the sort is only guaranteed to be stable if *lt?* is strictly a less-than relation.

```
(median lt? xs [ws]) → A
  lt? : (A A -> Any)
  xs : (Sequenceof A)
  ws : (U #f (Sequenceof Real)) = #f
```

Equivalent to `(quantile 1/2 lt? xs ws)`.

```
(quantile p lt? xs [ws]) → A
  p : Real
  lt? : (A A -> Any)
  xs : (Sequenceof A)
  ws : (U #f (Sequenceof Real)) = #f
```

Computes the inverse of the empirical cdf represented by the samples *xs*, which are optionally weighted by *ws*.

Examples:

```
> (quantile 0 < '(1 3 5))
- : Integer [more precisely: Positive-Byte]
```

```
1
> (quantile 0.5 < '(1 2 3 4))
- : Integer [more precisely: Positive-Byte]
2
> (quantile 0.5 < '(1 2 3 4) '(0.25 0.2 0.2 0.35))
- : Integer [more precisely: Positive-Byte]
3
```

If `p = 0`, `quantile` returns the smallest element of *xs* under the ordering relation *lt?*. If `p = 1`, it returns the largest element.

For weighted samples, `quantile` sorts *xs* and *ws* together (using `sort-samples`), then finds the least `x` for which the proportion of its cumulative weight is greater than or equal to *p*.

For unweighted samples, `quantile` uses the quickselect algorithm to find the element that would be at index `(ceiling (- (* p n) 1))` if *xs* were sorted, where `n` is the length of *xs*.

```
(absdev xs [ws]) → Nonnegative-Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
```

Computes the average absolute difference between the elements in *xs* and `(median < xs ws)`. If *ws* is not `#f`, it is a weighted average.

```
(absdev/median median xs [ws]) → Nonnegative-Real
  median : Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
```

Like `(absdev xs ws)`, but computed using known median *median*.

```
(hpd-interval lt? δ p xs [ws]) → (Values A A)
  lt? : (A A -> Any)
  δ : (A A -> Real)
  p : Real
  xs : (Sequenceof A)
  ws : (U #f (Sequenceof Real)) = #f
(hpd-interval/sorted δ p xs [ws]) → (Values A A)
  δ : (A A -> Real)
  p : Real
  xs : (Sequenceof A)
  ws : (U #f (Sequenceof Real)) = #f
```

Estimates the smallest interval for which the distribution that produced *xs* (optionally

weighted by *ws*) assigns probability *p*, which must be positive. The type A represents an ordered metric space with order *lt?* and metric δ.

To compute an HPD interval from sorted samples, use hpd-interval/sorted.

You almost certainly want to use real-hpd-interval or real-hpd-interval/sorted instead, which are defined in terms of these.

```
(real-hpd-interval p xs [ws]) → (Values Real Real)
  p : Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
(real-hpd-interval/sorted p xs [ws]) → (Values Real Real)
  p : Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
```

Equivalent to (hpd-interval < - *p xs ws*) and (hpd-interval/sorted - *p xs ws*).

Examples:

```
> (define beta32 (beta-dist 3 2))
> (real-dist-hpd-interval beta32 0.8)
- : (values Flonum Flonum)
0.36542991742846176
0.8939657937826784
> (real-hpd-interval 0.8 (sample beta32 10000))
- : (values Real Real)
0.3672288712838851
0.8926394607377907
```

8.6 Simulations

The functions in this section support Monte Carlo simulation; for example, quantifying uncertainty about statistics estimated from samples.

```
(mc-variance xs [ws #:bias bias]) → Nonnegative-Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
(mc-stddev xs [ws #:bias bias]) → Nonnegative-Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
```

Estimate the variance and standard deviation of expected values computed from random samples.

If *xs* are random variable samples, then

```
(define θ (mean xs ws))
```

is also a random variable sample. These two values:

```
(mc-variance xs ws #:bias bias)
(mc-stddev xs ws #:bias bias)
```

estimate the variance and standard deviation of θ. The latter is simply the square root of the variance, and bias correction is applied as described in §8.1 “Expected Values”.

Two different ways to estimate the standard deviation of a mean computed from 1000 samples are

```
> (mc-stddev (sample (normal-dist 0 1) 1000))
- : Real [more precisely: Nonnegative-Real]
0.03221600565674014
> (stddev (for/list : (Listof Real) ([_ (in-range 100)])
            (mean (sample (normal-dist 0 1) 1000))))
- : Real [more precisely: Nonnegative-Real]
0.03283213509379487
```

Using `mc-stddev` is 100 times faster in this case, and in most statistical applications, replicating a sampling process 100 times is infeasible.

```
(mc-stddev/mean m xs [ws #:bias bias]) → Nonnegative-Real
  m : Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
(mc-variance/mean m xs [ws #:bias bias]) → Nonnegative-Real
  m : Real
  xs : (Sequenceof Real)
  ws : (U #f (Sequenceof Real)) = #f
  bias : (U #t #f Real) = #f
```

Use these in the exceedingly rare cases in which you know the mean *m* but are estimating uncertainty in an estimate of the mean anyway.

```
(indicator pred?) → (A -> (U 0 1))
  pred? : (A -> Any)
```

Converts a predicate into an indicator function.

Examples:

```
> (fl (mean (map (indicator (λ ([x : Real]) (< -inf.0 x -1)))
                 (sample (normal-dist 0 1) 5000))))
- : Flonum
0.1548
> (real-dist-prob (normal-dist 0 1) -inf.0 -1)
- : Flonum
0.15865525393145705
```

```
(mc-probability pred? xs [ws]) → Nonnegative-Real
  pred? : (A -> Any)
  xs : (Sequenceof A)
  ws : (U #f (Sequenceof Real)) = #f
```

Estimates the probability of *pred?* from possibly weighted samples. Equivalent to `(mean (sequence-map (indicator pred?) xs) ws)`.

Example:

```
> (fl (mc-probability (λ ([x : Real]) (< -inf.0 x -1))
                      (sample (normal-dist 0 1) 5000)))
- : Flonum [more precisely: Nonnegative-Flonum]
0.1552
```

```
(mc-prob-dist pred? xs [ws]) → Beta-Dist
  pred? : (A -> Any)
  xs : (Sequenceof A)
  ws : (U #f (Sequenceof Real)) = #f
```

Returns a beta distribution estimated from possibly weighted samples whose mean is `(mc-probability pred? xs ws)`.

Computing a confidence interval for a probability whose endpoints are guaranteed to be between 0 and 1:

```
> (real-dist-hpd-interval
   (mc-prob-dist (λ ([x : Real]) (< -inf.0 x -1))
                 (sample (normal-dist 0 1) 5000))
   0.95)
- : (values Flonum Flonum)
0.14442705687887947
0.16445050897892943
```

9 Probability Distributions

```
(require math/distributions)
```
package: `math-lib`

The `math/distributions` module exports the following:

1. Distribution objects, which represent probability distributions
2. Functions that operate on distribution objects
3. The low-level flonum functions used to define distribution objects

Performance Warning: Using distribution objects in untyped Racket is currently 25-50 times slower than using them in Typed Racket, due to the overhead of checking higher-order contracts. We are working on it.

For now, if you need speed, either use the `typed/racket` language, or use just the low-level flonum functions, which are documented in §9.6 "Low-Level Distribution Functions".

9.1 Distribution Objects

A *distribution object* represents a probability distribution over a common domain, such as the real numbers, integers, or a set of symbols. Their constructors correspond with distribution families, such as the family of normal distributions.

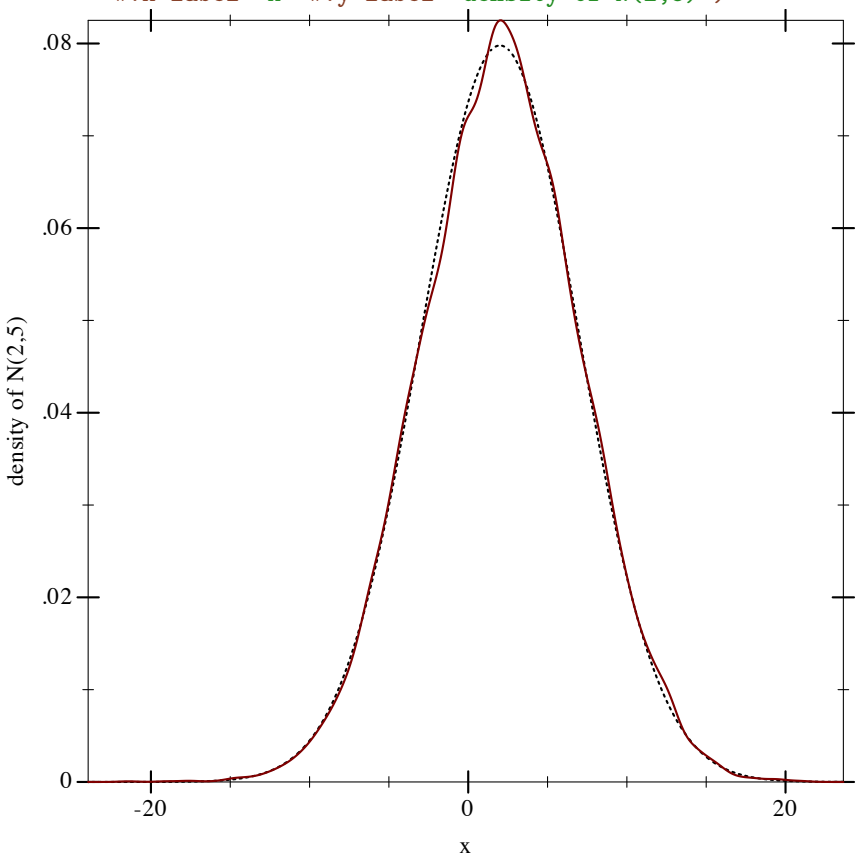
A distribution object, or a value of type `dist`, has a density function (a *pdf*) and a procedure to generate random samples. An *ordered* distribution object, or a value of type `ordered-dist`, additionally has a cumulative distribution function (a *cdf*), and its generalized inverse (an *inverse cdf*).

The following example creates an ordered distribution object representing a normal distribution with mean 2 and standard deviation 5, computes an approximation of the probability of the half-open interval (1/2,1], and computes another approximation from random samples:

```
> (define d (normal-dist 2 5))
> (real-dist-prob d 0.5 1.0)
0.038651712749849576
> (define xs (sample d 10000))
> (fl (/ (count (λ (x) (and (1/2 . < . x) (x . <= . 1))) xs)
         (length xs)))
0.0391
```

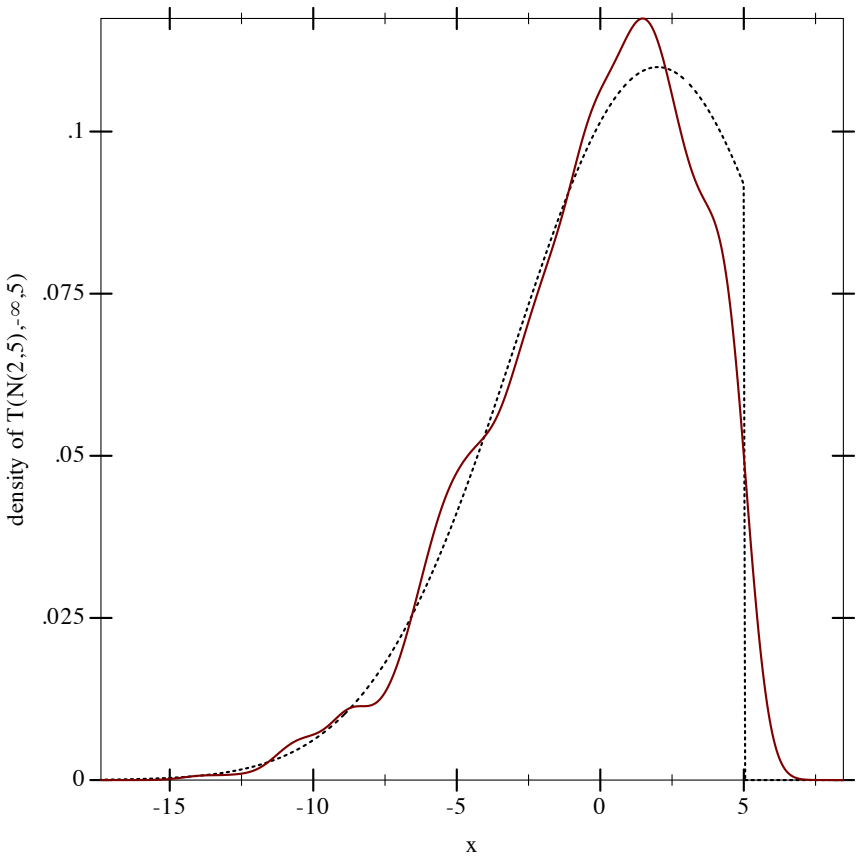
This plots the pdf and a kernel density estimate of the pdf from random samples:

```
> (plot (list (function (distribution-pdf d) #:color 0 #:style 'dot)
              (density xs))
        #:x-label "x" #:y-label "density of N(2,5)")
```



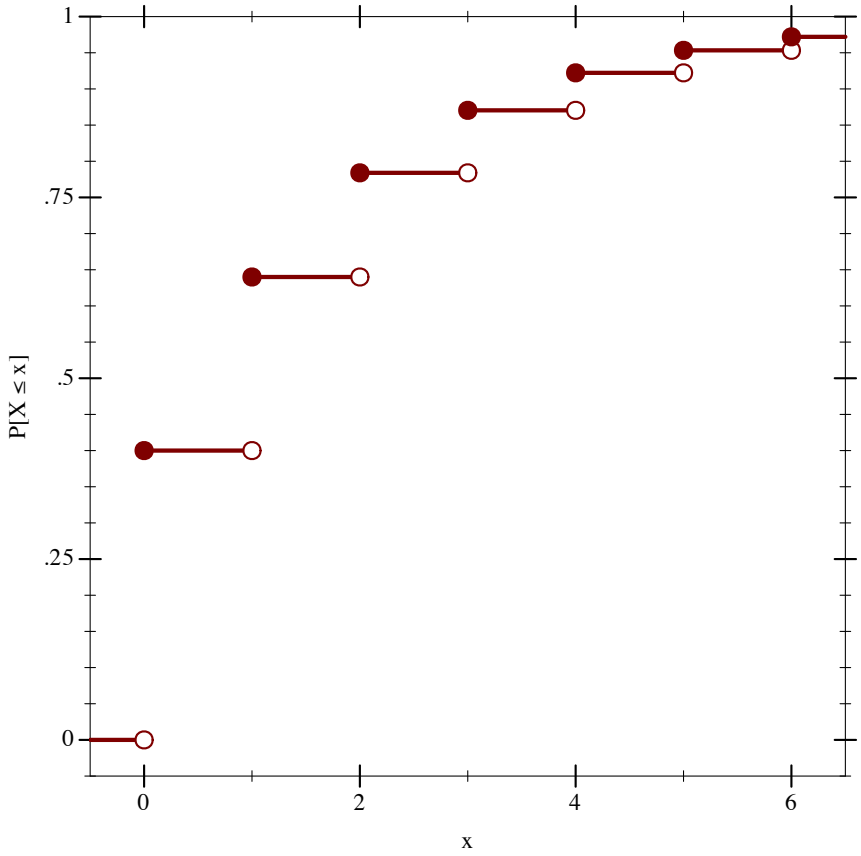
There are also higher-order distributions, which take other distributions as constructor arguments. For example, the truncated distribution family returns a distribution like its distribution argument, but sets probability outside an interval to 0 and renormalizes the probabilities within the interval:

```
> (define d-trunc (truncated-dist d -inf.0 5))
> (real-dist-prob d-trunc 5 6)
0.0
> (real-dist-prob d-trunc 0.5 1.0)
0.0532578419490049
> (plot (list (function (distribution-pdf d-
trunc) #:color 0 #:style 'dot)
              (density (sample d-trunc 1000)))
        #:x-label "x" #:y-label "density of T(N(2,5),-∞,5)")
```



Because real distributions' cdfs represent the probability P[$X \leqslant x$], they are right-continuous (i.e. continuous *from the right*):

```
> (define d (geometric-dist 0.4))
> (plot (for/list ([i  (in-range -1 7)])
          (define i+1-ε (flprev (+ i 1.0)))
          (list (lines (list (vector i (cdf d i))
                             (vector i+1-ε (cdf d i+1-ε)))
                       #:width 2)
                (points (list (vector i (cdf d i)))
                        #:sym 'fullcircle5 #:color 1)
                (points (list (vector i+1-ε (cdf d i+1-ε)))
                        #:sym 'fullcircle5 #:color 1 #:fill-
color 0)))
        #:x-min -0.5 #:x-max 6.5 #:y-min -0.05 #:y-max 1
        #:x-label "x" #:y-label "P[X ⩽ x]")
```



For convenience, cdfs are defined over the extended reals regardless of their distribution's support, but their inverses return values only within the support:

```
> (cdf d +inf.0)
1.0
> (cdf d 1.5)
0.64
> (cdf d -inf.0)
0.0
> (inv-cdf d (cdf d +inf.0))
+inf.0
> (inv-cdf d (cdf d 1.5))
1.0
> (inv-cdf d (cdf d -inf.0))
0.0
```

A distribution's inverse cdf is defined on the interval [0,1] and is always left-continuous, except possibly at 0 when its support is bounded on the left (as with `geometric-dist`).

Every pdf and cdf can return log densities and log probabilities, in case densities or probabilities are too small to represent as flonums (i.e. are less than `+min.0`):

```
> (define d (normal-dist))
> (pdf d 40.0)
0.0
> (cdf d -40.0)
0.0
> (pdf d 40.0 #t)
-800.9189385332047
> (cdf d -40.0 #t)
-804.6084420137538
```

Additionally, every cdf can return upper-tail probabilities, which are always more accurate when lower-tail probabilities are greater than `0.5`:

```
> (cdf d 20.0)
1.0
> (cdf d 20.0 #f #t)
2.7536241186062337e-89
```

Upper-tail probabilities can also be returned as log probabilities in case probabilities are too small:

```
> (cdf d 40.0)
1.0
> (cdf d 40.0 #f #t)
0.0
> (cdf d 40.0 #t #t)
-804.6084420137538
```

Inverse cdfs accept log probabilities and upper-tail probabilities.

The functions `lg+` and `lgsum`, as well as others in `math/flonum`, perform arithmetic on log probabilities.

When distribution object constructors receive parameters outside their domains, they return *undefined distributions*, or distributions whose functions all return `+nan.0`:

```
> (pdf (gamma-dist -1 2) 2)
+nan.0
> (sample (poisson-dist -2))
+nan.0
> (cdf (beta-dist 0 0) 1/2)
+nan.0
> (inv-cdf (geometric-dist 1.1) 0.2)
+nan.0
```

9.2 Distribution Types and Operations

```
(PDF In)
```

The type of probability density functions, or pdfs, defined as

```
(case-> (In -> Flonum)
        (In Any -> Flonum))
```

For any function of this type, the second argument should default to `#f`. When not `#f`, the function should return a log density.

```
(Sample Out)
```

The type of a distribution's sampling procedure, defined as

```
(case-> (-> Out)
        (Integer -> (Listof Out)))
```

When given a nonnegative integer `n` as an argument, a sampling procedure should return a length-`n` list of independent, random samples.

```
(CDF In)
```

The type of cumulative distribution functions, or cdfs, defined as

```
(case-> (In -> Flonum)
        (In Any -> Flonum)
        (In Any Any -> Flonum))
```

For any function of this type, both optional arguments should default to `#f`, and be interpreted as specified in the description of `cdf`.

```
(Inverse-CDF Out)
```

The type of inverse cumulative distribution functions, or inverse cdfs, defined as

```
(case-> (Real -> Out)
        (Real Any -> Out)
        (Real Any Any -> Out))
```

For any function of this type, both optional arguments should default to `#f`, and be interpreted as specified in the description of `inv-cdf`.

```
(distribution In Out)
```

```
(struct distribution (pdf sample))
  pdf : (PDF In)
  sample : (Sample Out)
```

The parent type of distribution objects. The `In` type parameter is the data type a distribution accepts as arguments to its pdf. The `Out` type parameter is the data type a distribution returns as random samples.

Examples:

```
> (distribution? (discrete-dist '(a b c)))
#t
> (distribution? (normal-dist))
#t
> ((distribution-pdf (normal-dist)) 0)
0.39894228040143265
> ((distribution-sample (normal-dist)))
-0.04563219155128638
```

See `pdf` and `sample` for uncurried forms of `distribution-pdf` and `distribution-sample`.

Type Examples:

```
> (ann (poisson-dist) (distribution Real Real))
- : (distribution Real Real)
(poisson-dist 0.5)
```

```
(ordered-dist In Out)
```

```
(struct ordered-dist distribution (cdf inv-cdf min max median))
  cdf : (CDF In)
  inv-cdf : (Inverse-CDF Out)
  min : Out
  max : Out
  median : (Promise Out)
```

The parent type of *ordered* distribution objects.

Similarly to `distribution`, the `In` type parameter is the data type an ordered distribution accepts as arguments to its pdf, and the `Out` type parameter is the data type an ordered distribution returns as random samples. Additionally, its cdf accepts values of type `In`, and its inverse cdf returns values of type `Out`.

Examples:

```
> (ordered-dist? (discrete-dist '(a b c)))
#f
> (ordered-dist? (normal-dist))
#t
```

The median is stored in an `ordered-dist` to allow interval probabilities to be computed accurately. For example, for `d = (normal-dist)`, whose median is `0.0`, `(real-dist-prob d -2.0 -1.0)` is computed using lower-tail probabilities, and `(real-dist-prob d 1.0 2.0)` is computed using upper-tail probabilities.

Type Examples:

```
> (ann (poisson-dist) (ordered-dist Real Real))
- : (ordered-dist Real Real)
(poisson-dist 0.5)
```

Real-Dist

The parent type of real-valued distributions, such as any distribution returned by `normal-dist`. Equivalent to the type `(ordered-dist Real Flonum)`.

```
(pdf d v [log?]) → Flonum
  d : (dist In Out)
  v : In
  log? : Any = #f
```

An uncurried form of `distribution-pdf`. When *log?* is not `#f`, returns a log density.

Examples:

```
> (pdf (discrete-dist '(a b c) '(1 2 3)) 'a)
0.16666666666666666
> (pdf (discrete-dist '(a b c) '(1 2 3)) 'a #t)
-1.791759469228055
```

```
(sample d) → Out
  d : (dist In Out)
(sample d n) → (Listof Out)
  d : (dist In Out)
  n : Integer
```

An uncurried form of distribution-sample.

Examples:

```
> (sample (exponential-dist))
1.5478486780909397
> (sample (exponential-dist) 3)
'(1.4250371853575619 0.11196573377270272 0.3266063849268669)
```

```
(cdf d v [log? 1-p?]) → Flonum
  d : (ordered-dist In Out)
  v : In
  log? : Any = #f
  1-p? : Any = #f
```

An uncurried form of ordered-dist-cdf.

When log? is #f, cdf returns a probability; otherwise, it returns a log probability.

When 1-p? is #f, cdf returns a lower-tail probability or log probability (depending on log?); otherwise, it returns an upper-tail probability or log-probability.

```
(inv-cdf d p [log? 1-p?]) → Out
  d : (ordered-dist In Out)
  p : Real
  log? : Any = #f
  1-p? : Any = #f
```

An uncurried form of ordered-dist-inv-cdf.

When log? is #f, inv-cdf interprets p as a probability; otherwise, it interprets p as a log probability.

When 1-p? is #f, inv-cdf interprets p as a lower-tail probability or log probability (depending on log?); otherwise, it interprets p as an upper-tail probability or log probability.

```
(real-dist-prob d a b [log? 1-p?]) → Flonum
  d : Real-Dist
  a : Real
  b : Real
  log? : Any = #f
  1-p? : Any = #f
```

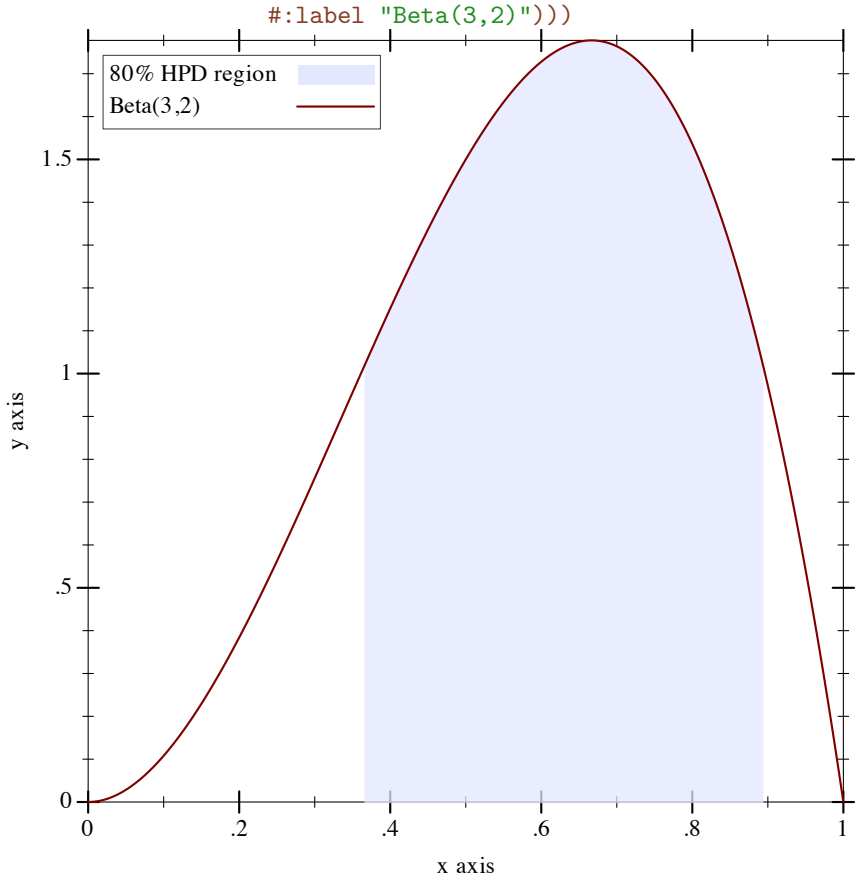
Computes the probability of the half-open interval (a, b]. (If b < a, the two endpoints are swapped first.) The log? and 1-p? arguments determine the meaning of the return value in the same way as the corresponding arguments to cdf.

```
(real-dist-hpd-interval d p) → (Values Flonum Flonum)
  d : Real-Dist
  p : Real
```

Finds the smallest interval for which `d` assigns probability `p`, if one exists.

Examples:

```
> (define d (beta-dist 3 2))
> (define-values (x0 x1) (real-dist-hpd-interval d 0.8))
> (plot (list
         (function-interval (λ (x) 0) (distribution-pdf d) x0 x1
                            #:line1-style 'transparent
                            #:line2-style 'transparent
                            #:label "80% HPD region")
         (function (distribution-pdf d) 0 1
                   #:label "Beta(3,2)")))
```



9.3 Finite Distribution Families

9.3.1 Unordered Discrete Distributions

```
(Discrete-Dist A)
(discrete-dist xs) → (Discrete-Dist A)
  xs : (Sequenceof A)
(discrete-dist xs ws) → (Discrete-Dist A)
  xs : (Sequenceof A)
  ws : (Sequenceof Real)
(discrete-dist-values d) → (Listof A)
  d : (Discrete-Dist A)
(discrete-dist-probs d) → (Listof Positive-Flonum)
  d : (Discrete-Dist A)
```

Represents families of unordered, discrete distributions over values of type `A`, with equality decided by `equal?`.

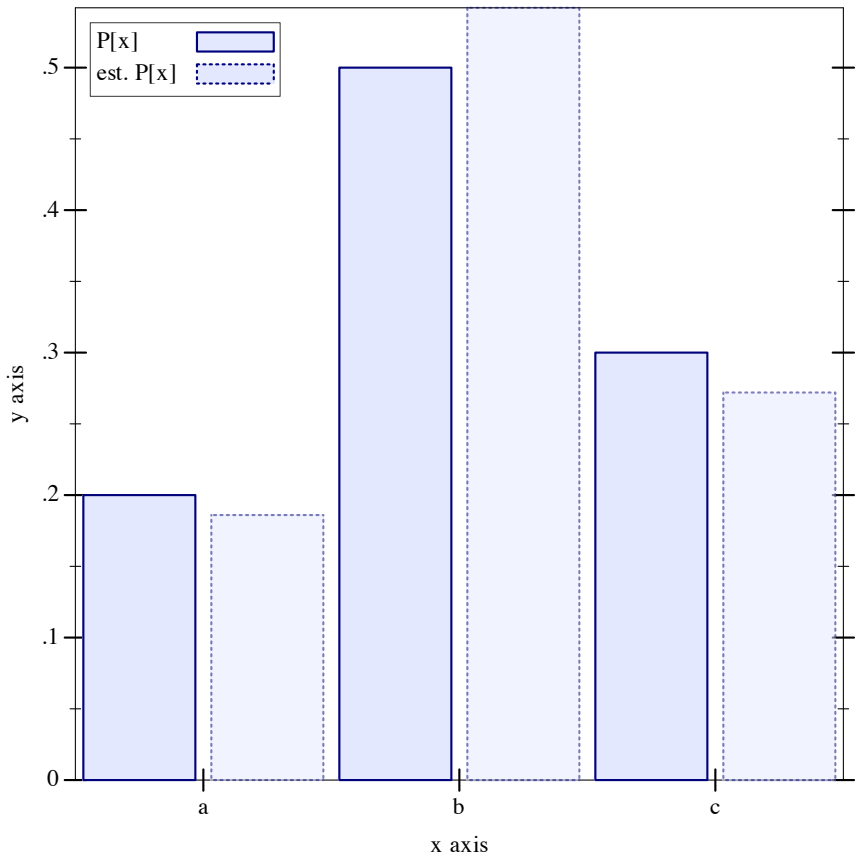
The weights in `ws` must be nonnegative, and are treated as unnormalized probabilities. When `ws` is not given, the values in `xs` are assigned uniform probabilities.

The type `(Discrete-Dist A)` is a subtype of `(dist A A)`. This means that discrete distribution objects are unordered, and thus have only a pdf and a procedure to generate random samples.

Note, however, that the `discrete-dist-values` and `discrete-dist-probs` functions produce lists that may be paired; that is, if the result of calling `discrete-dist-values` on a given distribution produces a list whose third element is `'a`, and the result of calling `discrete-dist-probs` on the same distribution produces a list whose third element is `0.25`, then the given distribution associates the probability `0.25` with the value `'a`.

Examples:

```
> (define xs '(a b c))
> (define d (discrete-dist xs '(2 5 3)))
> (define n 500)
> (define h (samples->hash (sample d n)))
> (plot (list (discrete-histogram
               (map vector xs (map (distribution-pdf d) xs))
               #:x-min 0 #:skip 2 #:label "P[x]")
              (discrete-histogram
               (map vector xs (map (λ (x) (/ (hash-
ref h x) n)) xs))
               #:x-min 1 #:skip 2 #:line-style 'dot #:alpha 0.5
               #:label "est. P[x]")))
```



9.4 Integer Distribution Families

Mathematically, integer distributions are commonly defined in one of two ways: over extended reals, or over extended integers. The most common definitions use the extended reals, so the following distribution object constructors return objects of type `Real-Dist`.

(Another reason is that the extended integers correspond with the type `(U Integer +inf.0 -inf.0)`. Values of this type have little support in Racket's library.)

This leaves us with a quandary and two design decisions users should be aware of. The quandary is that, when an integer distribution is defined over the reals, it has a cdf, but *no well-defined pdf*: the pdf would be zero except at integer points, where it would be undefined.

Unfortunately, an integer distribution without a pdf is nearly useless. So the pdfs of these integer distributions are pdfs defined over integers, while their cdfs are defined over reals.

In measure-theory parlance, the pdfs are defined with respect to counting measure, while the cdfs are defined with respect to Lebesgue measure.

Most implementations, such as R's, make the same design choice. Unlike R's, this imple-

mentation's pdfs return +nan.0 when given non-integers, for three reasons:

- Their domain of definition is the integers.
- Applying an integer pdf to a non-integer almost certainly indicates a logic error, which is harder to detect when a program returns an apparently sensible value.
- If this design choice turns out to be wrong and we change pdfs to return 0.0, this should affect very few programs. A change from 0.0 to +nan.0 could break many programs.

Integer distributions defined over the extended integers are not out of the question, and may show up in future versions of math/distributions if there is a clear need.

9.4.1 Bernoulli Distributions

Wikipedia: Bernoulli Distribution.

```
Bernoulli-Dist
(bernoulli-dist prob) → Bernoulli-Dist
  prob : Real
(bernoulli-dist-prob d) → Flonum
  d : Bernoulli-Dist
```

Represents the Bernoulli distribution family parameterized by probability of success.

(bernoulli-dist prob) is equivalent to (binomial-dist 1 prob), but operations on it are faster.

Examples:

```
> (define d (bernoulli-dist 0.75))
> (map (distribution-pdf d) '(0 1))
'(0.25 0.75)
> (map (ordered-dist-cdf d) '(0 1))
'(0.25 1.0)
> (define d (binomial-dist 1 0.75))
> (map (distribution-pdf d) '(0 1))
'(0.25 0.75)
> (map (ordered-dist-cdf d) '(0 1))
'(0.25 1.0)
```

9.4.2 Binomial Distributions

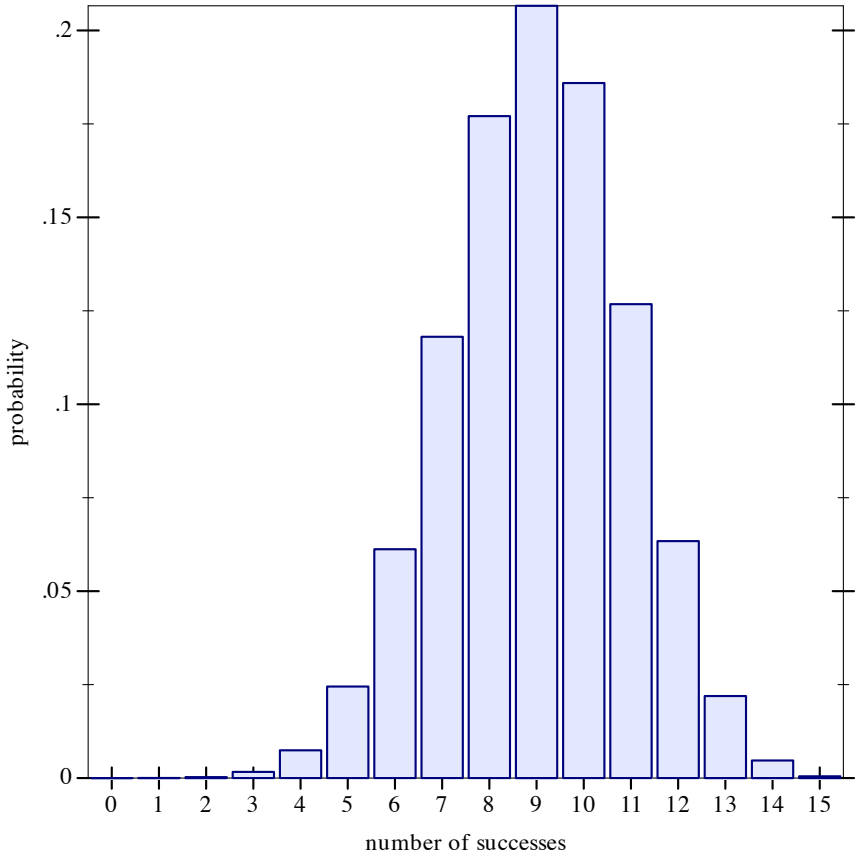
Wikipedia: Binomial Distribution.

```
Binomial-Dist
(binomial-dist count prob) → Binomial-Dist
  count : Real
  prob : Real
(binomial-dist-count d) → Flonum
  d : Binomial-Dist
(binomial-dist-prob d) → Flonum
  d : Binomial-Dist
```

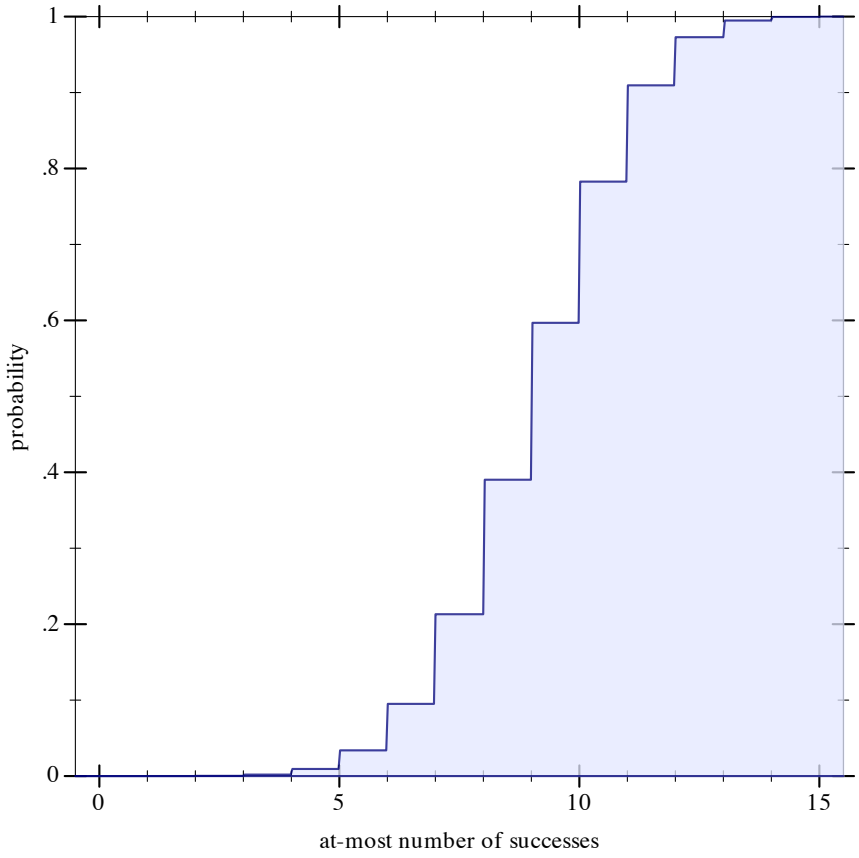
Represents the binomial distribution family parameterized by count (number of trials) and probability of success.

Examples:

```
> (define d (binomial-dist 15 0.6))
> (plot (discrete-histogram
         (map vector (build-list 16 values) (build-
list 16 (distribution-pdf d))))
        #:x-label "number of successes" #:y-label "probability")
```




```
> (plot (function-interval (λ (x) 0) (ordered-dist-
cdf d) -0.5 15.5)
        #:x-label "at-most number of successes" #:y-
label "probability")
```



9.4.3 Geometric Distributions

Wikipedia: Geometric Distribution.

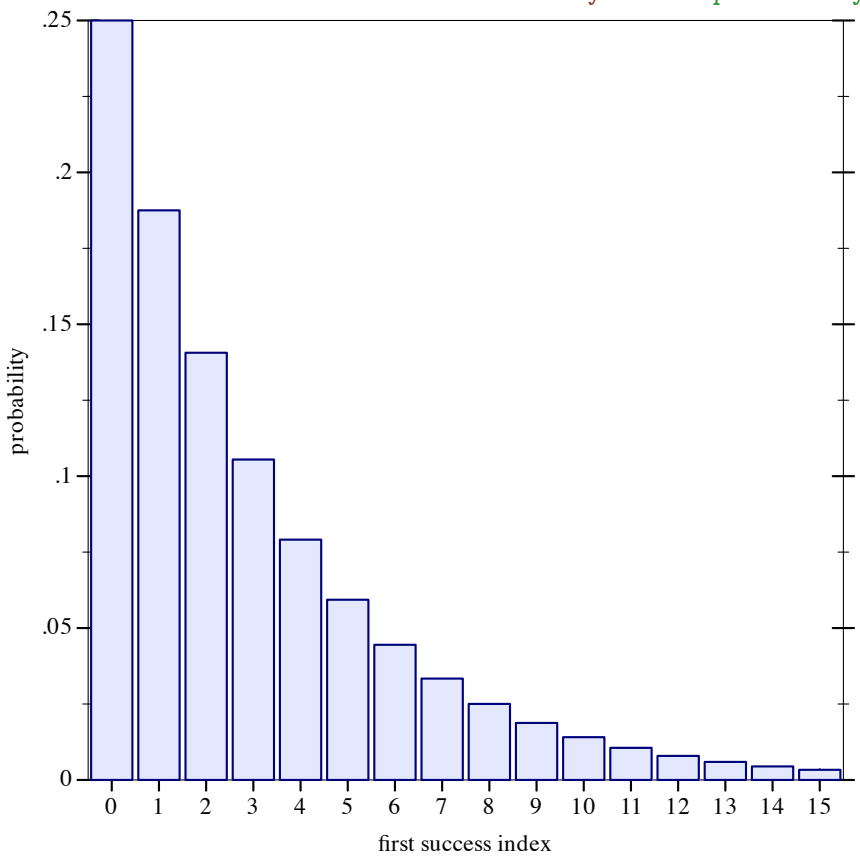
```
Geometric-Dist
(geometric-dist prob) → Geometric-Dist
  prob : Real
(geometric-dist-prob d) → Flonum
  d : Geometric-Dist
```

Represents the geometric distribution family parameterized by success probability. The random variable is the number of failures before the first success, or equivalently, the index of the first success starting from zero.

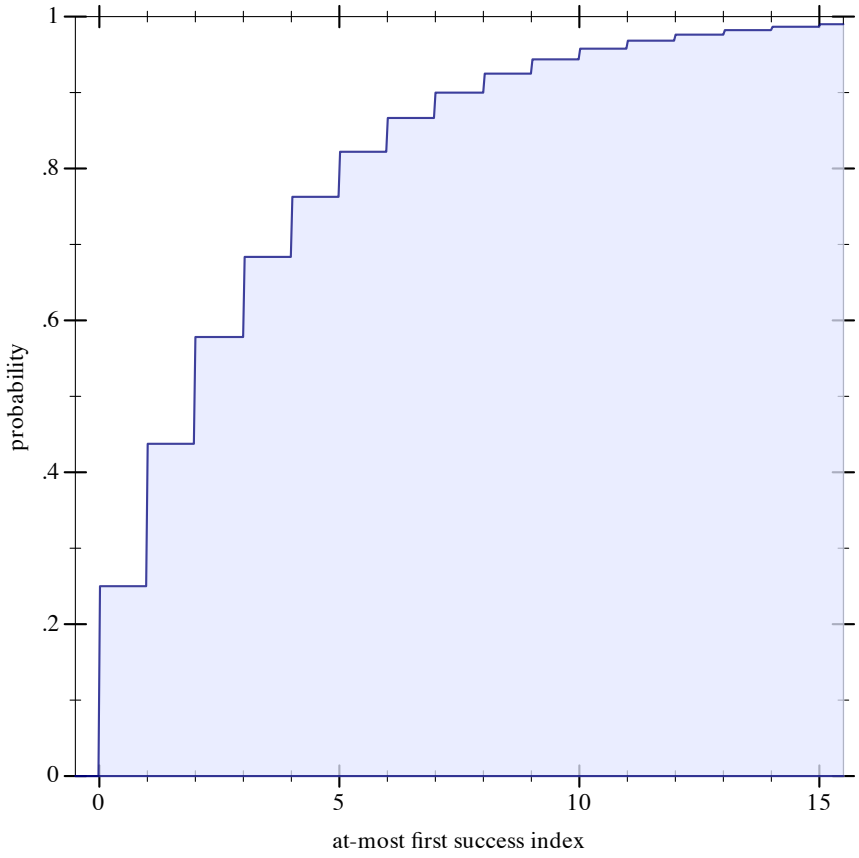
Examples:

```
> (define d (geometric-dist 0.25))
> (plot (discrete-histogram
```

```
        (map vector (build-list 16 values) (build-
list 16 (distribution-pdf d))))
       #:x-label "first success index" #:y-label "probability")
```




```
> (plot (function-interval (λ (x) 0) (ordered-dist-
cdf d) -0.5 15.5)
       #:x-label "at-most first success index" #:y-
label "probability"
       #:y-max 1)
```



9.4.4 Poisson Distributions

Wikipedia: Poisson Distribution.

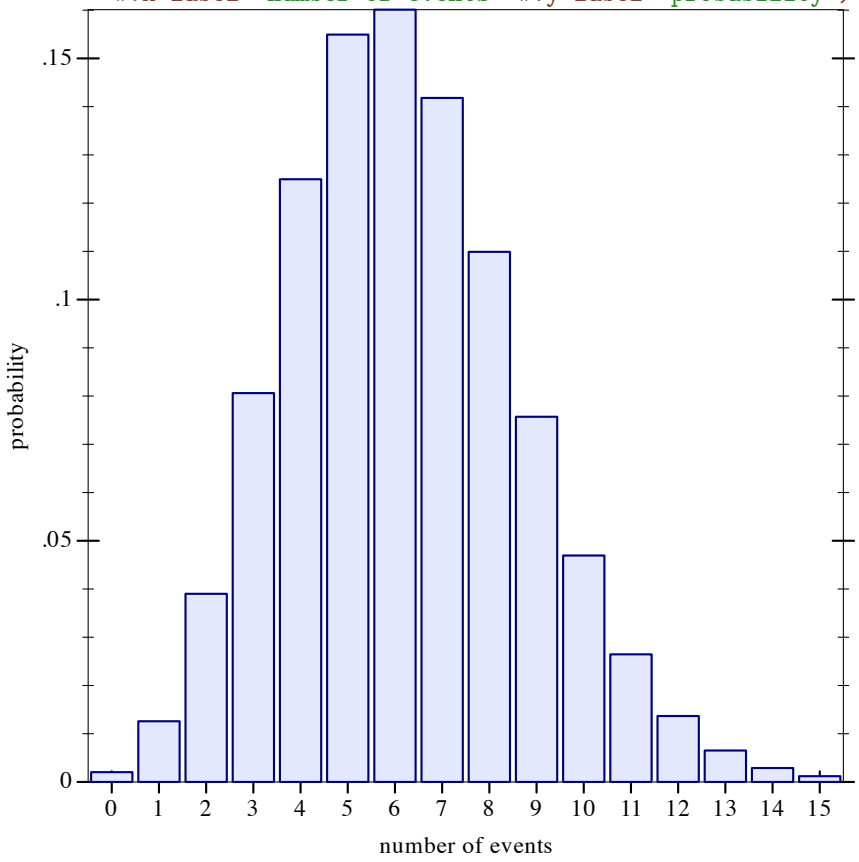
```
Poisson-Dist
(poisson-dist mean) → Poisson-Dist
  mean : Real
(poisson-dist-mean d) → Flonum
  d : Poisson-Dist
```

Represents the Poisson distribution family parameterized by the mean number of occurrences of independent events.

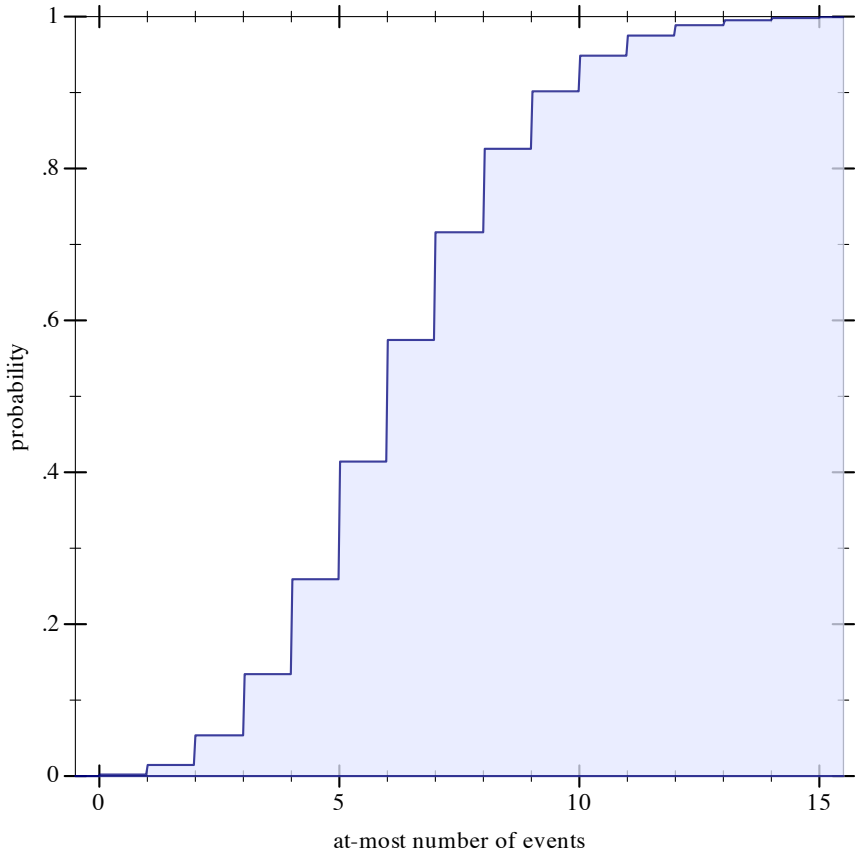
Examples:

```
> (define d (poisson-dist 6.2))
> (plot (discrete-histogram
```

```
        (map vector (build-list 16 values) (build-
list 16 (distribution-pdf d))))
       #:x-label "number of events" #:y-label "probability")
```




```
> (plot (function-interval (λ (x) 0) (ordered-dist-
cdf d) -0.5 15.5)
       #:x-label "at-most number of events" #:y-
label "probability"
       #:y-max 1)
```



9.5 Real Distribution Families

The distribution object constructors documented in this section return uniquely defined distributions for the largest possible parameter domain. This usually means that they return distributions for a larger domain than their mathematical counterparts are defined on.

For example, those that have a scale parameter, such as `cauchy-dist`, `logistic-dist`, `exponential-dist` and `normal-dist`, are typically undefined for a zero scale. However, in floating-point math, it is often useful to simulate limits in finite time using special values like `+inf.0`. Therefore, when a scale-parameterized family's constructor receives `0`, it returns a distribution object that behaves like a `Delta-Dist`:

```
> (pdf (normal-dist 1 0) 1)
+inf.0
> (pdf (normal-dist 1 0) 1.0000001)
0.0
```

Further, negative scales are accepted, even for `exponential-dist`, which results in a distribution with positive scale reflected about zero.

Some parameters' boundary values give rise to non-unique limits. Sometimes the ambiguity can be resolved using necessary properties; see `Gamma-Dist` for an example. When no resolution exists, as with `(beta-dist 0 0)`, which puts an indeterminate probability on the value `0` and the rest on `1`, the constructor returns an undefined distribution.

Some distribution object constructors attempt to return sensible distributions when given special values such as `+inf.0` as parameters. Do not count on these yet.

Many distribution families, such as `Gamma-Dist`, can be parameterized on either scale or rate (which is the reciprocal of scale). In all such cases, the implementations provided by `math/distributions` are parameterized on scale.

9.5.1 Beta Distributions

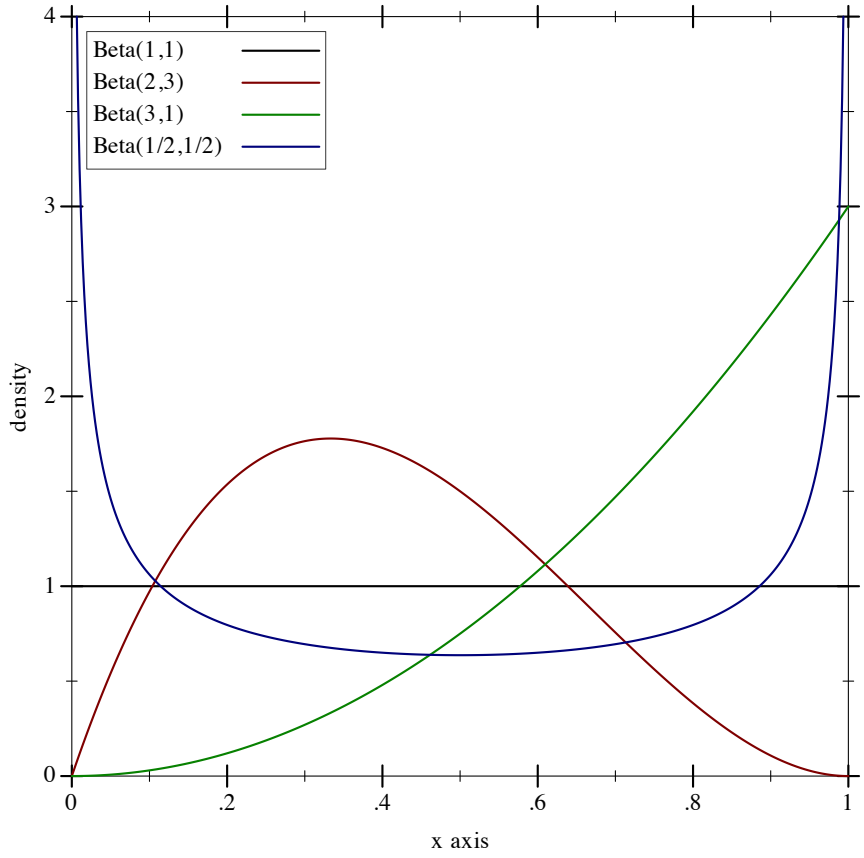
Wikipedia: Beta Distribution.

```
Beta-Dist
(beta-dist alpha beta) → Beta-Dist
  alpha : Real
  beta : Real
(beta-dist-alpha d) → Flonum
  d : Beta-Dist
(beta-dist-beta d) → Flonum
  d : Beta-Dist
```

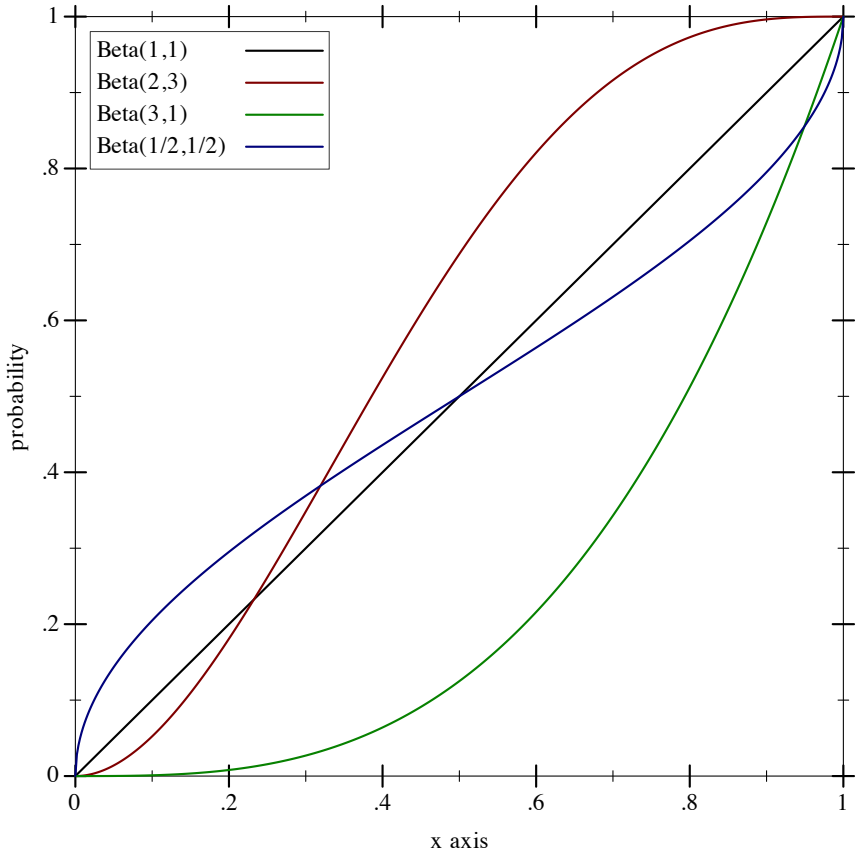
Represents the beta distribution family parameterized by two shape parameters, or pseudo-counts, which must both be nonnegative.

Examples:

```
> (plot (for/list ([α  (in-list '(1 2 3 1/2))]
                   [β  (in-list '(1 3 1 1/2))]
                   [i  (in-naturals)])
           (function (distribution-pdf (beta-dist α β))
                     #:color i #:label (format "Beta(~a,~a)" α β)))
        #:x-min 0 #:x-max 1 #:y-max 4 #:y-label "density")
```




```
> (plot (for/list ([α  (in-list '(1 2 3 1/2))]
                   [β  (in-list '(1 3 1 1/2))]
                   [i  (in-naturals)])
          (function (ordered-dist-cdf (beta-dist α β))
                    #:color i #:label (format "Beta(~a,~a)" α β)))
        #:x-min 0 #:x-max 1 #:y-label "probability")
```



`(beta-dist 0 0)` and `(beta-dist +inf.0 +inf.0)` are undefined distributions.

When `a = 0` or `b = +inf.0`, the returned distribution acts like `(delta-dist 0)`.

When `a = +inf.0` or `b = 0`, the returned distribution acts like `(delta-dist 1)`.

9.5.2 Cauchy Distributions

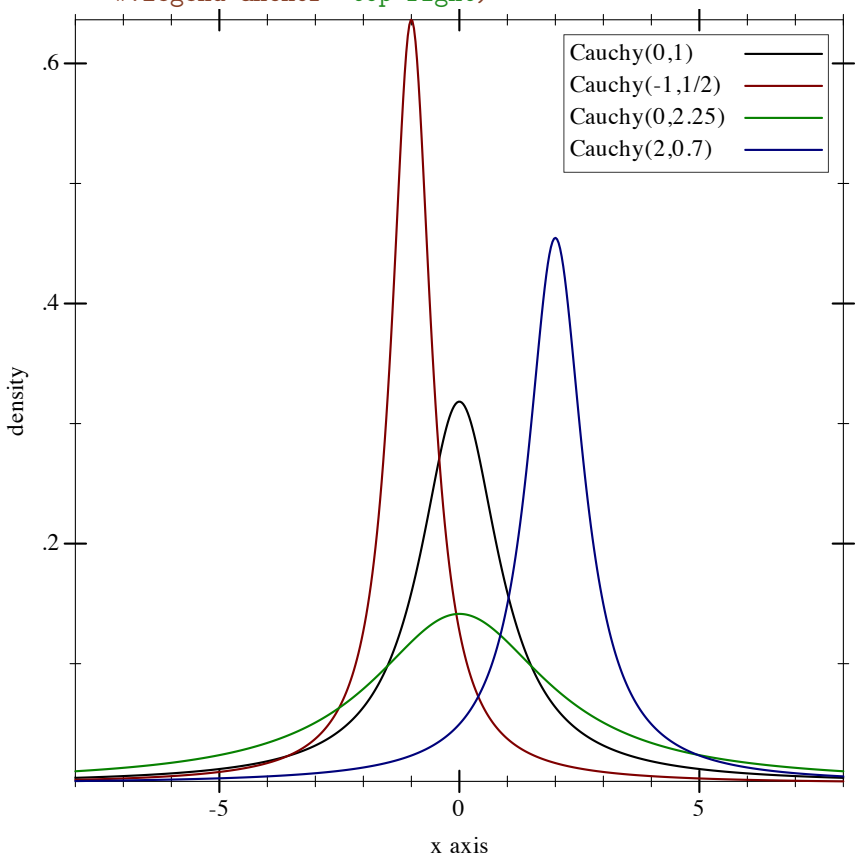
Wikipedia: Cauchy Distribution.

```
Cauchy-Dist
(cauchy-dist [mode scale]) → Cauchy-Dist
  mode : Real = 0
  scale : Real = 1
(cauchy-dist-mode d) → Flonum
  d : Cauchy-Dist
(cauchy-dist-scale d) → Flonum
  d : Cauchy-Dist
```

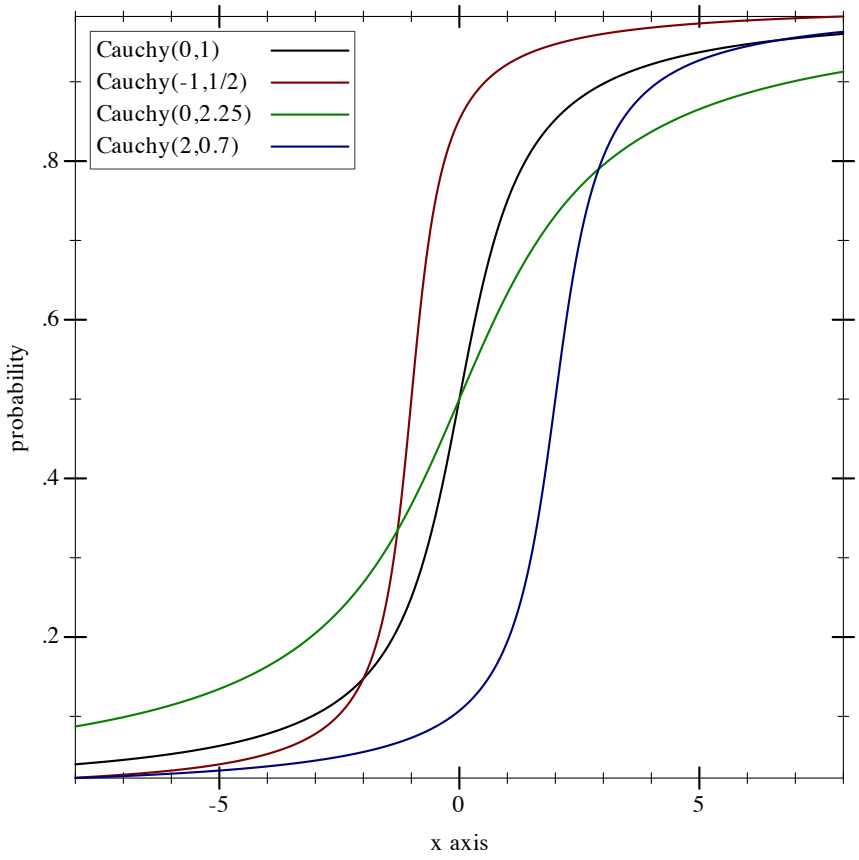
Represents the Cauchy distribution family parameterized by mode and scale.

Examples:

```
> (plot (for/list ([m  (in-list '(0 -1 0 2))]
                   [s  (in-list '(1 1/2 2.25 0.7))]
                   [i  (in-naturals)])
          (function (distribution-pdf (cauchy-dist m s))
                    #:color i #:label (format "Cauchy(~a,~a)" m s)))
        #:x-min -8 #:x-max 8 #:y-label "density"
        #:legend-anchor 'top-right)
```




```
> (plot (for/list ([m  (in-list '(0 -1 0 2))]
                   [s  (in-list '(1 1/2 2.25 0.7))]
                   [i  (in-naturals)])
          (function (ordered-dist-cdf (cauchy-dist m s))
                    #:color i #:label (format "Cauchy(~a,~a)" m s)))
        #:x-min -8 #:x-max 8 #:y-label "probability")
```



9.5.3 Delta Distributions

Wikipedia: Dirac Delta Function.

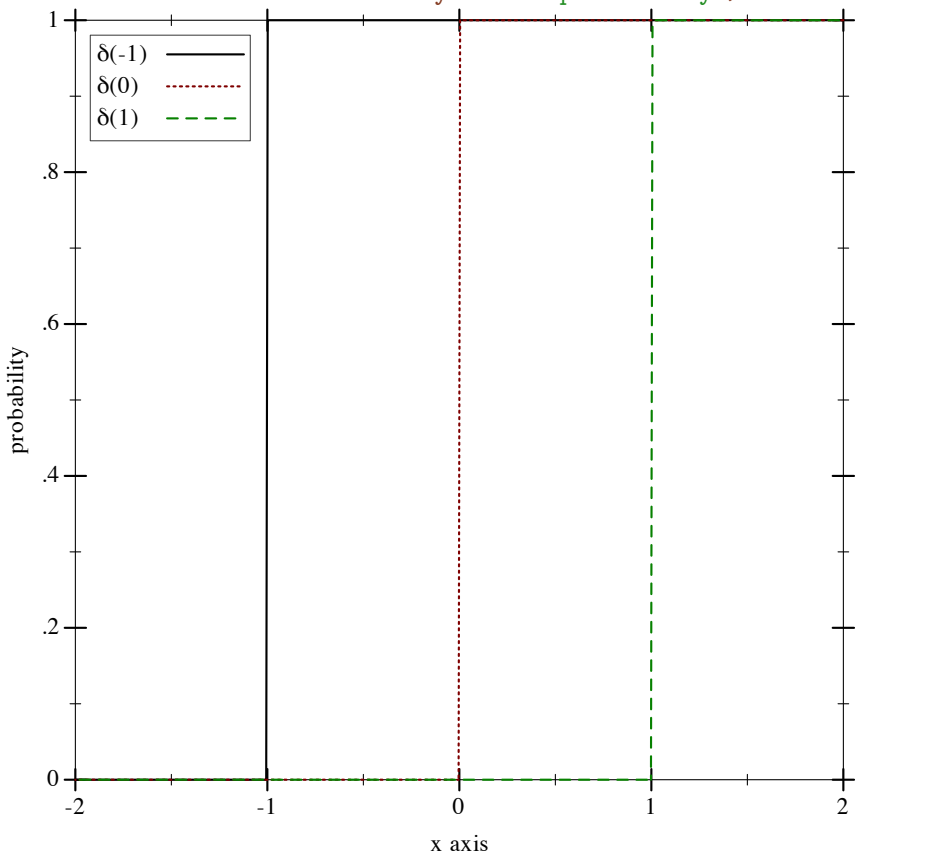
```
Delta-Dist
(delta-dist [mean]) → Delta-Dist
  mean : Real = 0
(delta-dist-mean d) → Flonum
  d : Delta-Dist
```

Represents the family of distributions whose densities are Dirac delta functions.

Examples:

```
> (pdf (delta-dist) 0)
+inf.0
> (pdf (delta-dist) 1)
0.0
```

```
> (plot (for/list ([μ  (in-list '(-1 0 1))]
                   [i  (in-naturals)])
          (function (ordered-dist-cdf (delta-dist μ))
                    #:color i #:style i #:label (format "δ(~a)" μ)))
        #:x-min -2 #:x-max 2 #:y-label "probability")
```



9.5.4 Exponential Distributions

Wikipedia: Exponential Distribution.

```
Exponential-Dist
(exponential-dist [mean]) → Exponential-Dist
  mean : Real = 1
(exponential-dist-mean d) → Flonum
  d : Exponential-Dist
```

Represents the exponential distribution family parameterized by mean, or scale.

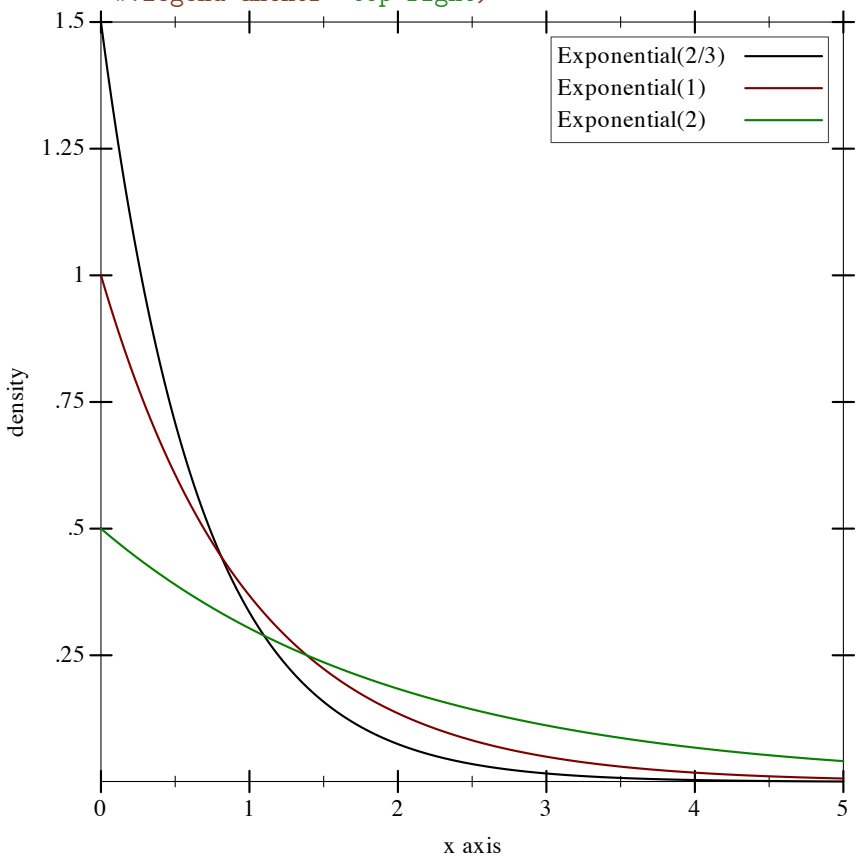
Warning: The exponential distribution family is often parameterized by *rate*, which is the

reciprocal of mean or scale. Construct exponential distributions from rates using

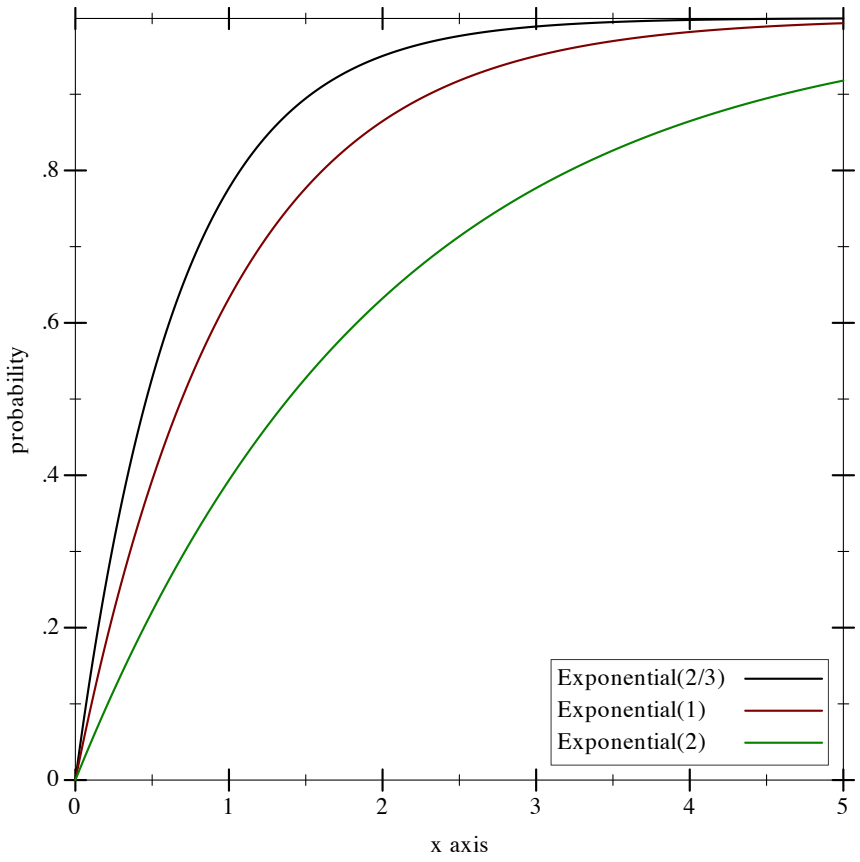
```
(exponential-dist (/ 1.0 rate))
```

Examples:

```
> (plot (for/list ([μ  (in-list '(2/3 1 2))]
                   [i  (in-naturals)])
          (function (distribution-pdf (exponential-dist μ))
                    #:color i #:label (format "Exponential(~a)" μ)))
        #:x-min 0 #:x-max 5 #:y-label "density"
        #:legend-anchor 'top-right)
```




```
> (plot (for/list ([μ  (in-list '(2/3 1 2))]
                   [i  (in-naturals)])
          (function (ordered-dist-cdf (exponential-dist μ))
                    #:color i #:label (format "Exponential(~a)" μ)))
        #:x-min 0 #:x-max 5 #:y-label "probability"
        #:legend-anchor 'bottom-right)
```



9.5.5 Gamma Distributions

Wikipedia: Gamma Distribution.

```
Gamma-Dist
(gamma-dist [shape scale]) → Gamma-Dist
  shape : Real = 1
  scale : Real = 1
(gamma-dist-shape d) → Flonum
  d : Gamma-Dist
(gamma-dist-scale d) → Flonum
  d : Gamma-Dist
```

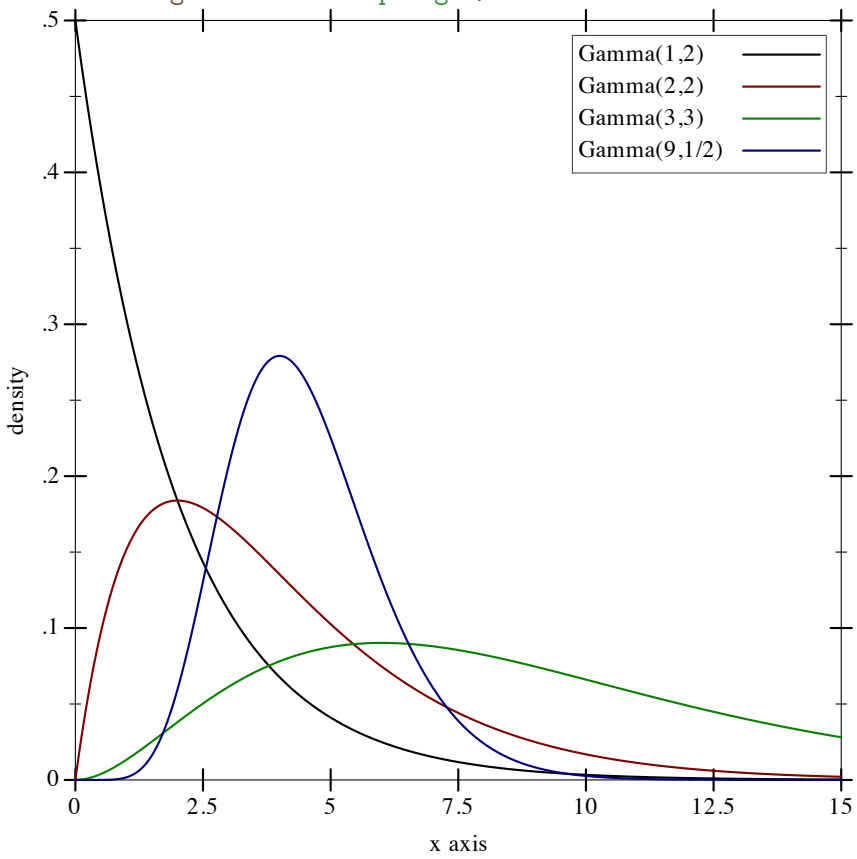
Represents the gamma distribution family parameterized by shape and scale. The *shape* parameter must be nonnegative.

Warning: The gamma distribution family is often parameterized by shape and *rate*, which is the reciprocal of scale. Construct gamma distributions from rates using

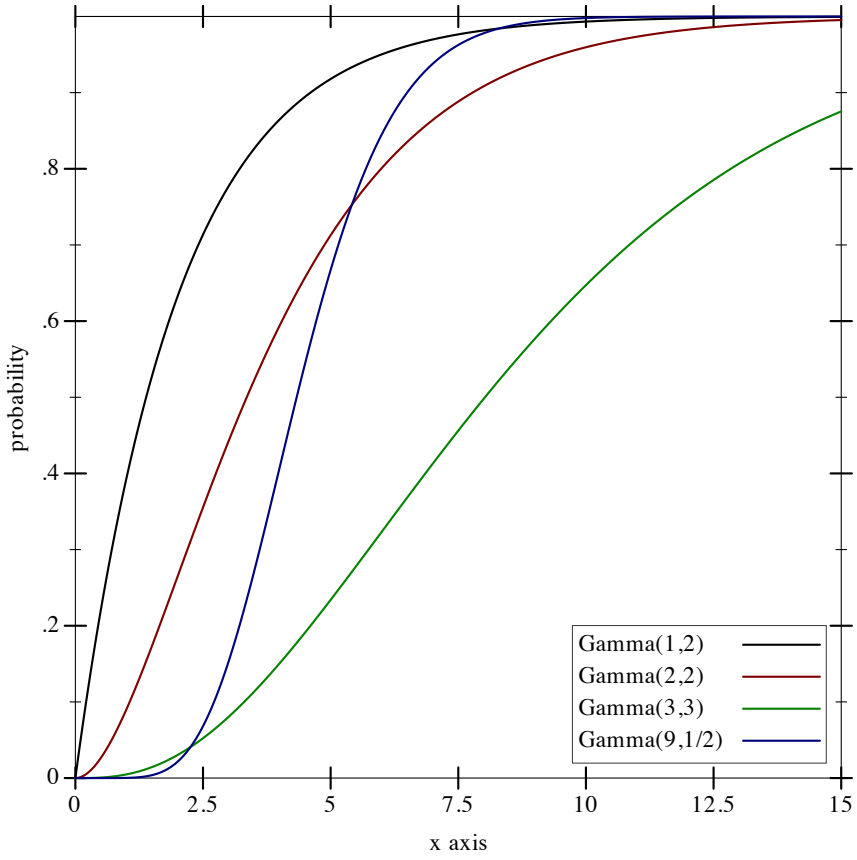
```
(gamma-dist shape (/ 1.0 rate))
```

Examples:

```
> (plot (for/list ([k  (in-list '(1 2 3 9))]
                   [s  (in-list '(2 2 3 1/2))]
                   [i  (in-naturals)])
          (function (distribution-pdf (gamma-dist k s))
                    #:color i #:label (format "Gamma(~a,~a)" k s)))
        #:x-min 0 #:x-max 15 #:y-label "density"
        #:legend-anchor 'top-right)
```




```
> (plot (for/list ([k  (in-list '(1 2 3 9))]
                   [s  (in-list '(2 2 3 1/2))]
                   [i  (in-naturals)])
          (function (ordered-dist-cdf (gamma-dist k s))
                    #:color i #:label (format "Gamma(~a,~a)" k s)))
        #:x-min 0 #:x-max 15 #:y-label "probability"
        #:legend-anchor 'bottom-right)
```



The cdf of the gamma distribution with `shape = 0` could return either `0.0` or `1.0` at `x = 0`, depending on whether a double limit is taken with respect to `scale` or with respect to `x` first. However the limits are taken, the cdf must return `1.0` for `x > 0`. Because cdfs are right-continuous, the only correct choice is

```
> (cdf (gamma-dist 0 1) 0)
1.0
```

Therefore, a gamma distribution with `shape = 0` behaves like `(delta-dist 0)`.

9.5.6 Weibull Distributions

Wikipedia: Weibull Distribution.

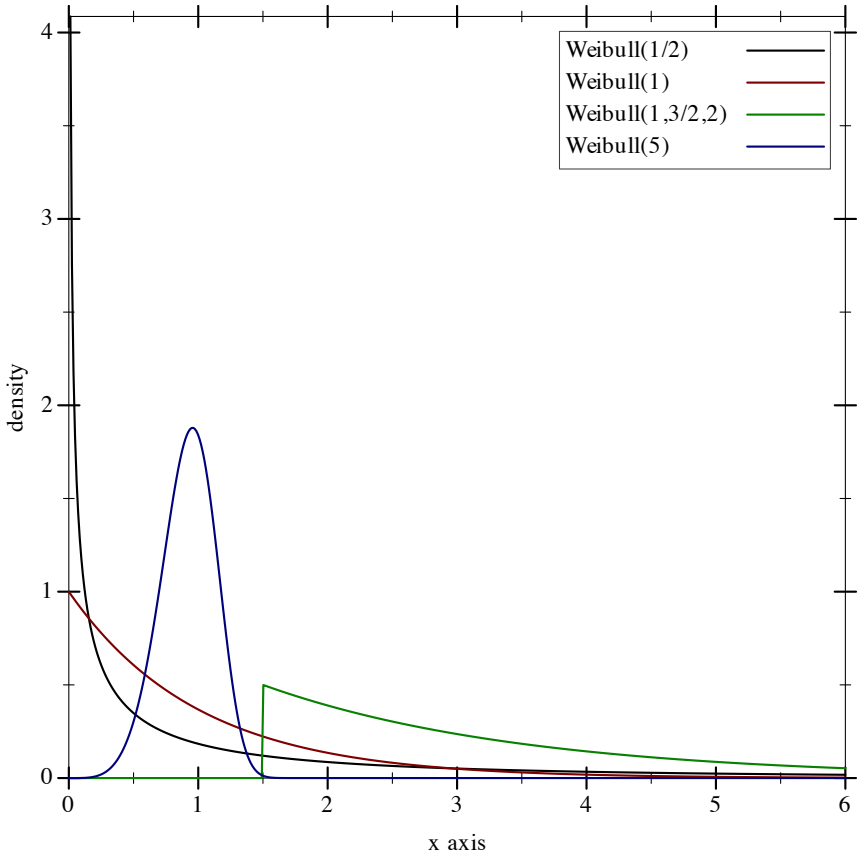
```
Weibull-Dist
```

```
(weibull-dist shape [location scale]) → Weibull-Dist
  shape : Positive-Real
  location : Real = 0
  scale : Real = 1
(weibull-dist-shape k) → Positive-Flonum
  k : Weibull-Dist
(weibull-dist-location d) → Real
  d : Weibull-Dist
(weibull-dist-scale d) → Real
  d : Weibull-Dist
```

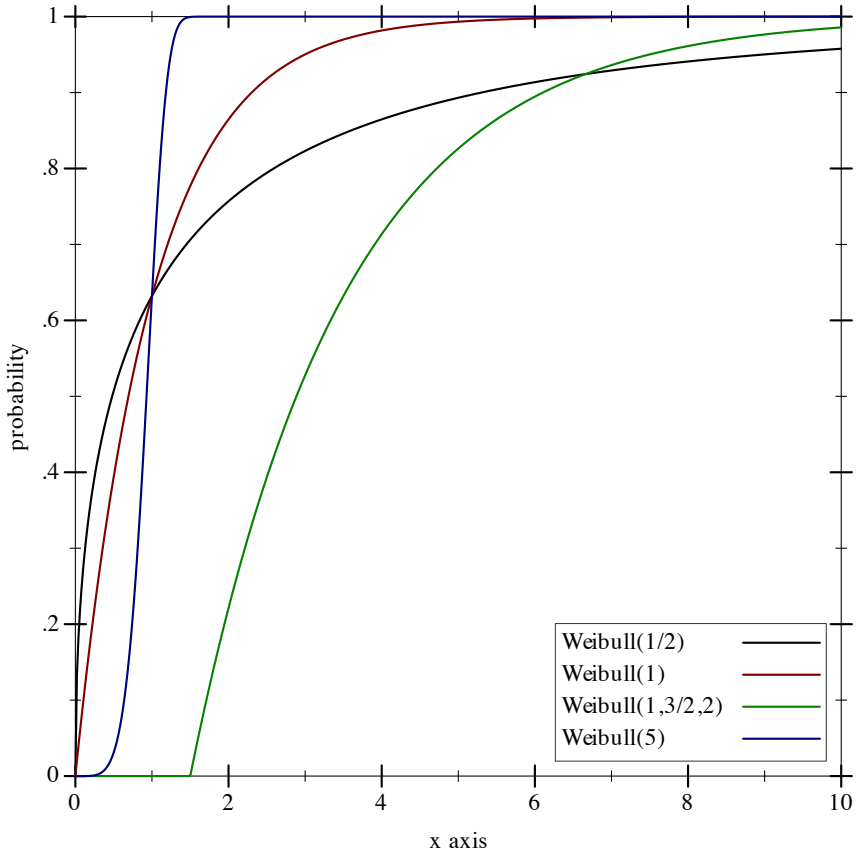
Represents the Weibull distribution family parameterized by shape, location and scale.

Examples:

```
> (plot (for/list ([k  (in-list '(1/2 1 1 5))]
                   [d  (in-list '(0 0 3/2 0))]
                   [s  (in-list '(1 1 2 1))]
                   [i  (in-naturals)])
           (function (distribution-pdf (weibull-dist k d s))
                     #:color i #:label (if (and (= d 0) (= s 1))
                                           (format "Weibull(~a)" k)
                                           (format "Weibull(~a,~a,~a)" k d s))))
         #:x-min 0 #:x-max 6 #:y-label "density"
         #:legend-anchor 'top-right)
```




```
> (plot (for/list ([k  (in-list '(1/2 1 1 5))]
                   [d  (in-list '(0 0 3/2 0))]
                   [s  (in-list '(1 1 2 1))]
                   [i  (in-naturals)])
          (function (ordered-dist-cdf (weibull-dist k d s))
                    #:color i #:label (if (and (= d 0) (= s 1))
                                          (format "Weibull(~a)" k)
                                          (format "Weibull(~a,~a,~a)" k d s))))
        #:x-min 0 #:x-max 10 #:y-label "probability"
        #:legend-anchor 'bottom-right)
```



9.5.7 Logistic Distributions

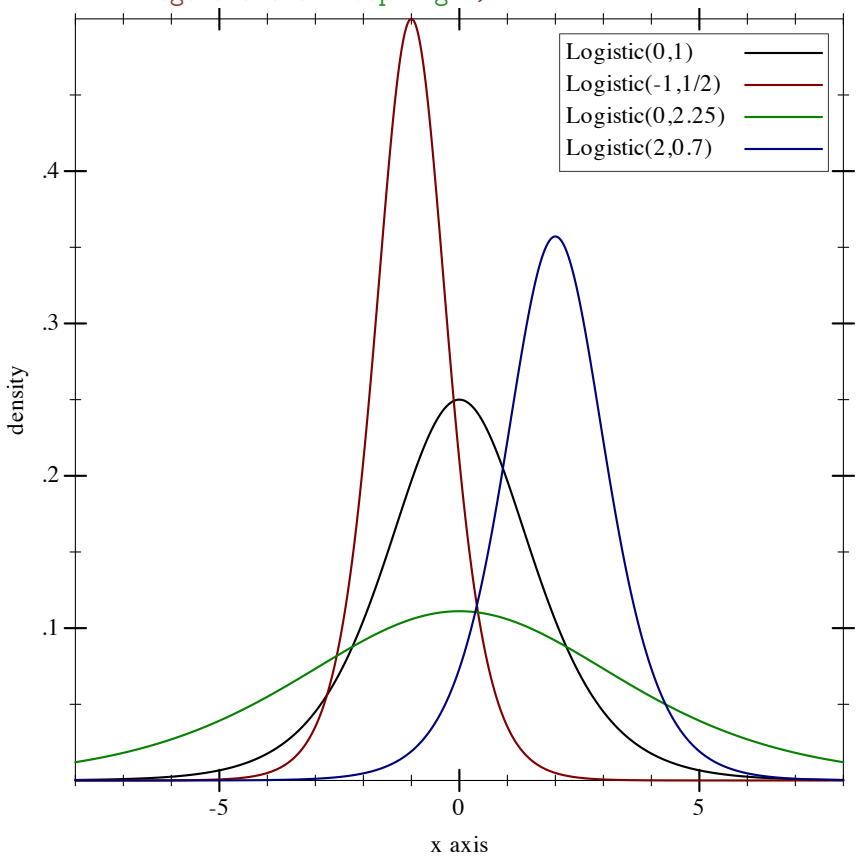
Wikipedia: Logistic Distribution.

```
Logistic-Dist
(logistic-dist [mean scale]) → Logistic-Dist
  mean : Real = 0
  scale : Real = 1
(logistic-dist-mean d) → Flonum
  d : Logistic-Dist
(logistic-dist-scale d) → Flonum
  d : Logistic-Dist
```

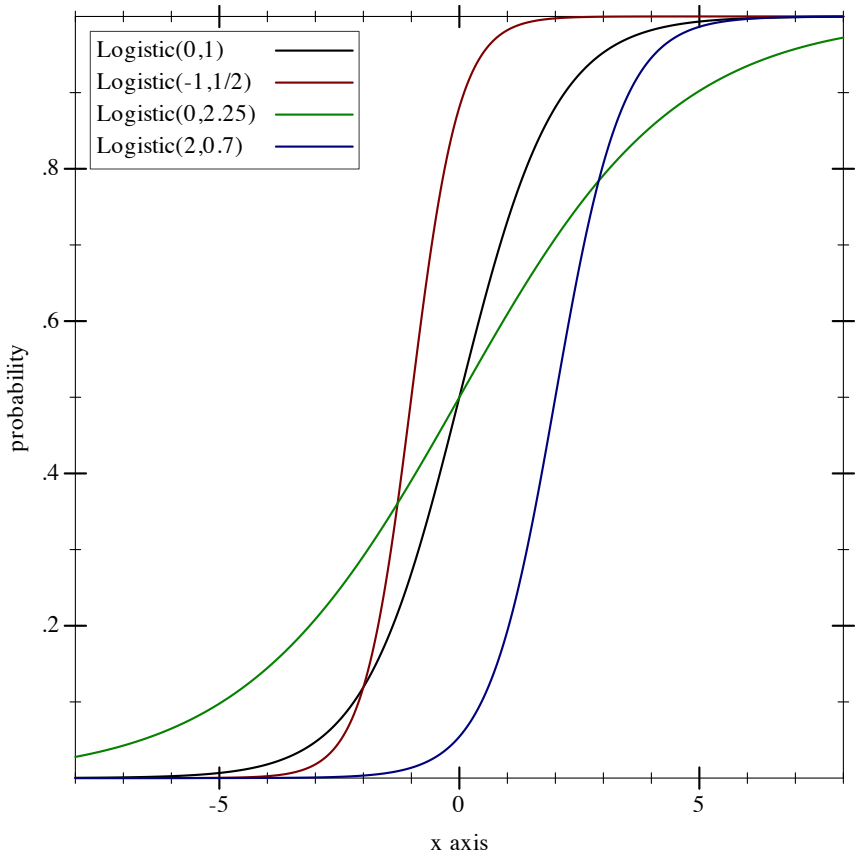
Represents the logistic distribution family parameterized by mean (also called "location") and scale. In this parameterization, the variance is `(* 1/3 (sqr (* pi scale)))`.

Examples:

```
> (plot (for/list ([μ  (in-list '(0 -1 0 2))]
                   [s  (in-list '(1 1/2 2.25 0.7))]
                   [i  (in-naturals)])
          (function (distribution-pdf (logistic-dist μ s))
                    #:color i #:label (format "Logistic(~a,~a)" μ s)))
        #:x-min -8 #:x-max 8 #:y-label "density"
        #:legend-anchor 'top-right)
```




```
> (plot (for/list ([μ  (in-list '(0 -1 0 2))]
                   [s  (in-list '(1 1/2 2.25 0.7))]
                   [i  (in-naturals)])
          (function (ordered-dist-cdf (logistic-dist μ s))
                    #:color i #:label (format "Logistic(~a,~a)" μ s)))
        #:x-min -8 #:x-max 8 #:y-label "probability")
```



9.5.8 Normal Distributions

Wikipedia: Normal Distribution.

```
Normal-Dist
(normal-dist [mean stddev]) → Normal-Dist
  mean : Real = 0
  stddev : Real = 1
(normal-dist-mean d) → Flonum
  d : Normal-Dist
(normal-dist-stddev d) → Flonum
  d : Normal-Dist
```

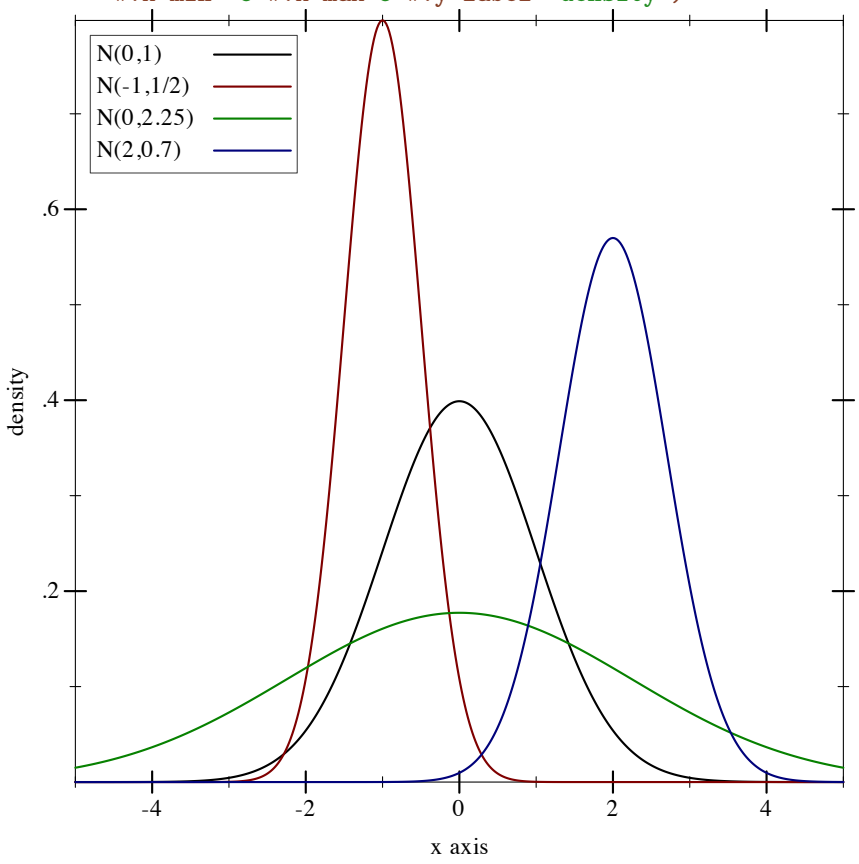
Represents the normal distribution family parameterized by mean and standard deviation.

Warning: The normal distribution family is often parameterized by mean and *variance*, which is the square of standard deviation. Construct normal distributions from variances using

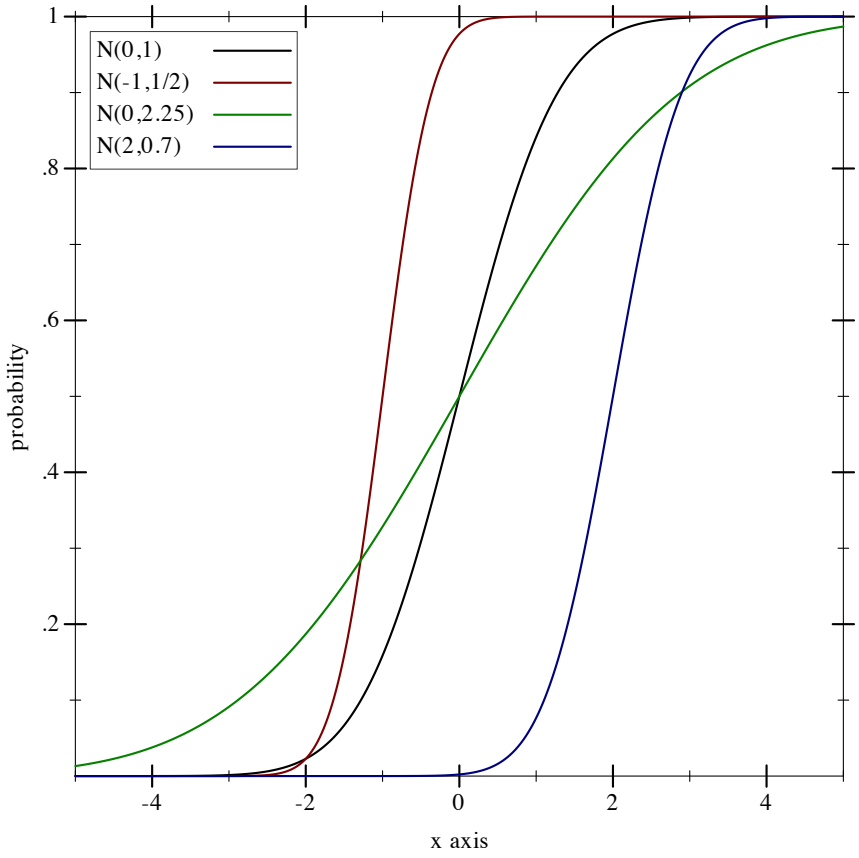
```
(normal-dist mean (sqrt var))
```

Examples:

```
> (plot (for/list ([μ  (in-list '(0 -1 0 2))]
                   [σ  (in-list '(1 1/2 2.25 0.7))]
                   [i  (in-naturals)])
          (function (distribution-pdf (normal-dist μ σ))
                    #:color i #:label (format "N(~a,~a)" μ σ)))
        #:x-min -5 #:x-max 5 #:y-label "density")
```




```
> (plot (for/list ([μ  (in-list '(0 -1 0 2))]
                   [σ  (in-list '(1 1/2 2.25 0.7))]
                   [i  (in-naturals)])
          (function (ordered-dist-cdf (normal-dist μ σ))
                    #:color i #:label (format "N(~a,~a)" μ σ)))
        #:x-min -5 #:x-max 5 #:y-label "probability")
```



9.5.9 Student-t Distributions

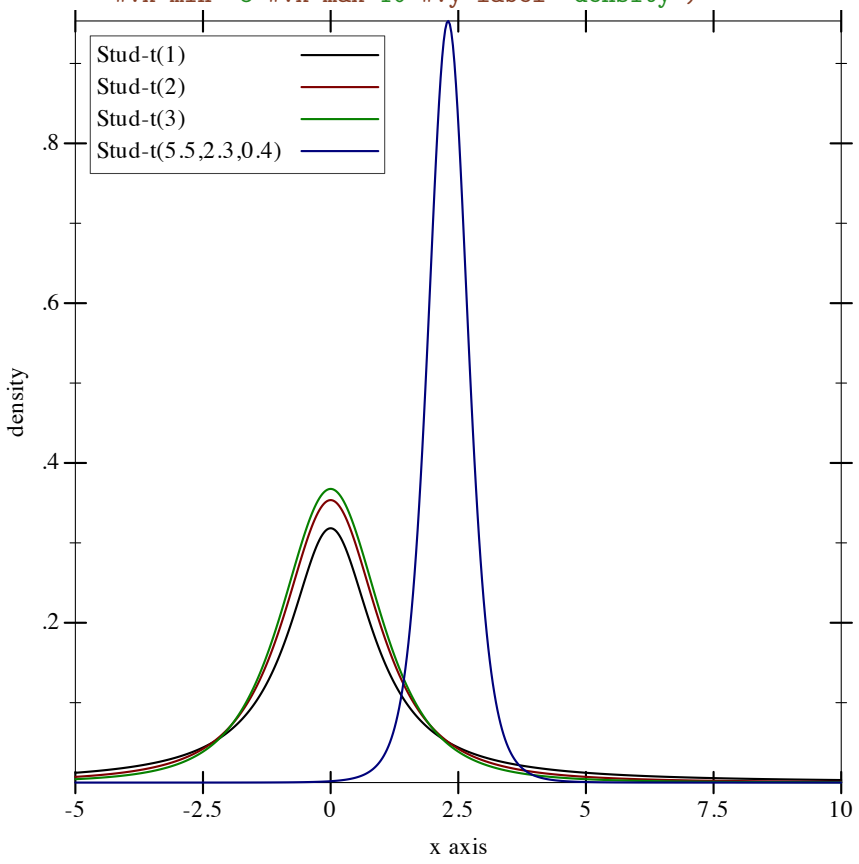
Wikipedia: Student-t Distribution.

```
Student-t-Dist
(student-t-dist freedom [mean scale]) → Student-t-Dist
  freedom : Positive-Real
  mean : Real = 0
  scale : Real = 1
(student-t-dist-freedom d) → Positive-Flonum
  d : Student-t-Dist
(student-t-dist-mean d) → Real
  d : Student-t-Dist
(student-t-dist-scale d) → Real
  d : Student-t-Dist
```

Represents the student-t distribution family parameterized by degree of freedom, mean (also called “location”) and scale.

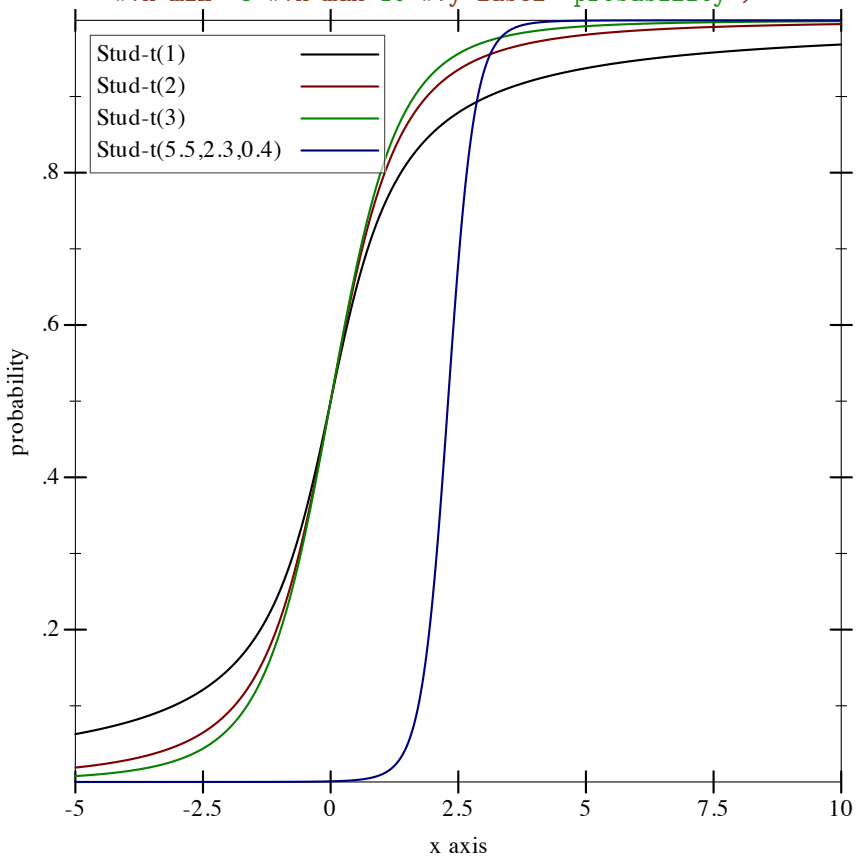
Examples:

```
> (plot (for/list ([ν  (in-list '(1 2 3 5.5))]
                   [μ  (in-list '(0 0 0 2.3))]
                   [σ  (in-list '(1 1 1 0.4))]
                   [i  (in-naturals)])
          (function (distribution-pdf (student-t-dist ν μ σ))
                    #:color i #:label (if (and (= μ 0) (= σ 1))
                                          (format "Stud-t(~a)" ν)
                                          (format "Stud-
t(~a,~a,~a)" ν μ σ))))
        #:x-min -5 #:x-max 10 #:y-label "density")
```




```
> (plot (for/list ([ν  (in-list '(1 2 3 5.5))]
                   [μ  (in-list '(0 0 0 2.3))]
                   [σ  (in-list '(1 1 1 0.4))]
                   [i  (in-naturals)])
          (function (ordered-dist-cdf (student-t-dist ν μ σ))
                    #:color i #:label (if (and (= μ 0) (= σ 1))
                                          (format "Stud-t(~a)" ν)
```

```
                                                    (format "Stud-
t(~a,~a,~a)" ν μ σ))))
        #:x-min -5 #:x-max 10 #:y-label "probability")
```



9.5.10 Triangular Distributions

Wikipedia: Triangular Distribution.

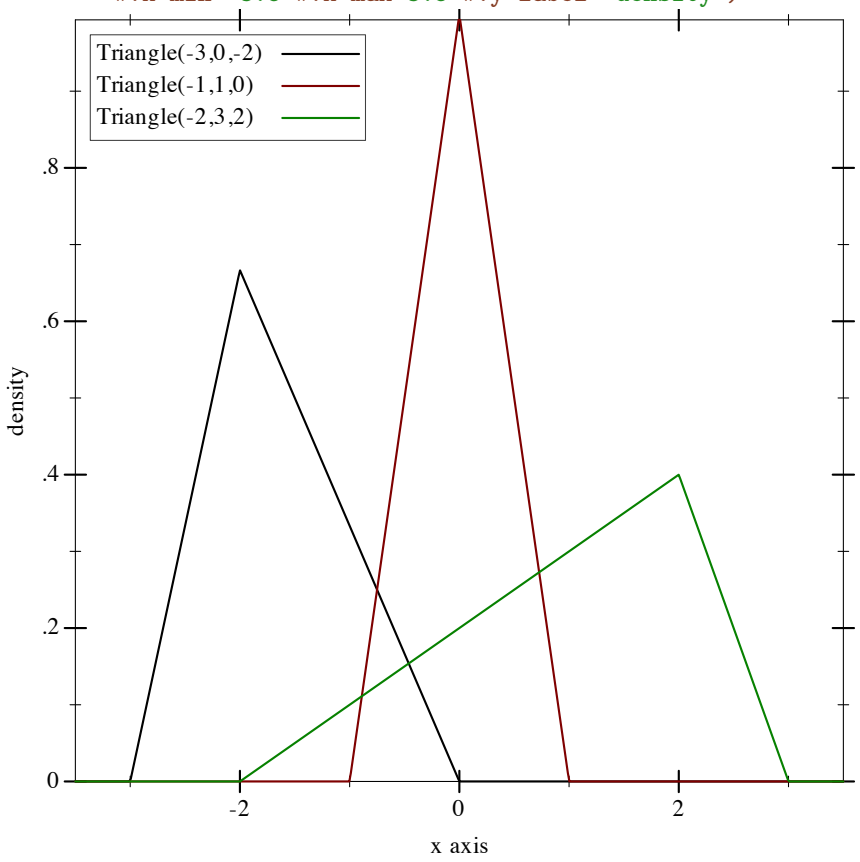
```
Triangle-Dist
(triangle-dist [min max mode]) → Triangle-Dist
  min : Real = 0
  max : Real = 1
  mode : Real = (* 0.5 (+ min max))
(triangle-dist-min d) → Flonum
  d : Triangle-Dist
(triangle-dist-max d) → Flonum
  d : Triangle-Dist
(triangle-dist-mode d) → Flonum
  d : Triangle-Dist
```

Represents the triangular distribution family parameterized by minimum, maximum and mode.

If *min*, *mode* and *max* are not in ascending order, they are sorted before constructing the distribution object.

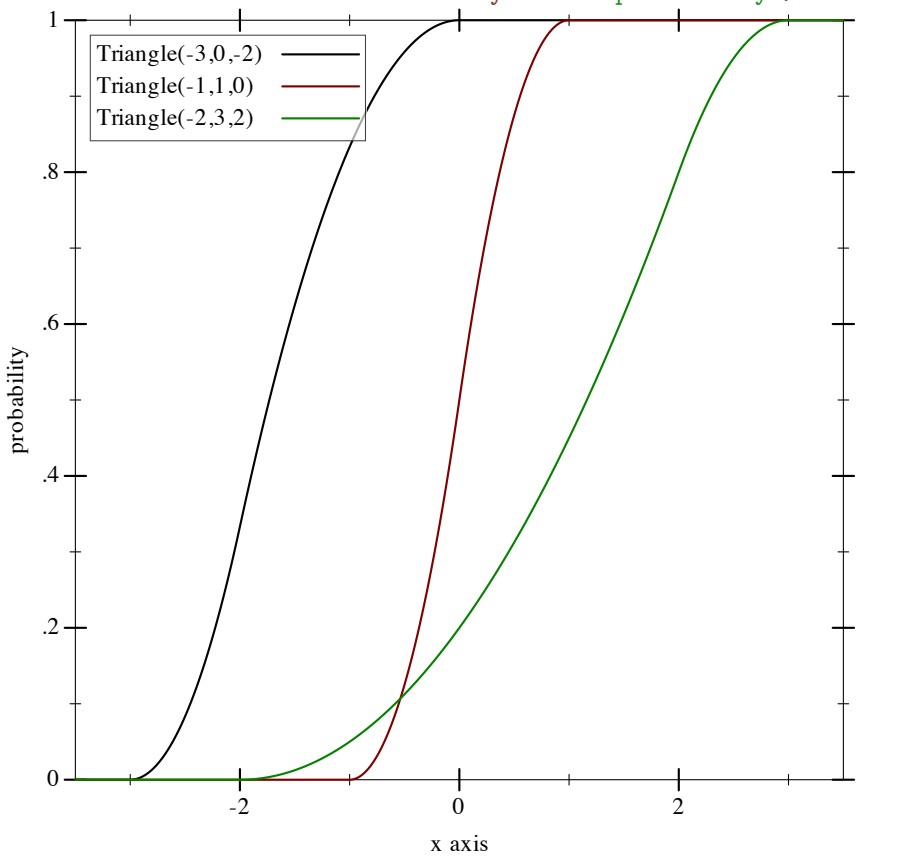
Examples:

```
> (plot (for/list ([a  (in-list '(-3 -1 -2))]
                   [b  (in-list '(0 1 3))]
                   [m  (in-list '(-2 0 2))]
                   [i  (in-naturals)])
          (function (distribution-pdf (triangle-
dist a b m)) #:color i
                    #:label (format "Triangle(~a,~a,~a)" a b m)))
        #:x-min -3.5 #:x-max 3.5 #:y-label "density")
```




```
> (plot (for/list ([a  (in-list '(-3 -1 -2))]
                   [b  (in-list '(0 1 3))]
                   [m  (in-list '(-2 0 2))]
```

```
                  [i  (in-naturals)])
          (function (ordered-dist-cdf (triangle-
dist a b m)) #:color i
                    #:label (format "Triangle(~a,~a,~a)" a b m)))
        #:x-min -3.5 #:x-max 3.5 #:y-label "probability")
```



`(triangle-dist c c c)` for any real `c` behaves like a support-limited delta distribution centered at `c`.

9.5.11 Truncated Distributions

`Truncated-Dist`

```
(truncated-dist d) → Truncated-Dist
  d : Real-Dist
(truncated-dist d max) → Truncated-Dist
  d : Real-Dist
  max : Real
(truncated-dist d min max) → Truncated-Dist
  d : Real-Dist
  min : Real
  max : Real
(truncated-dist-original t) → Real-Dist
  t : Truncated-Dist
(truncated-dist-min t) → Flonum
  t : Truncated-Dist
(truncated-dist-max t) → Flonum
  t : Truncated-Dist
```

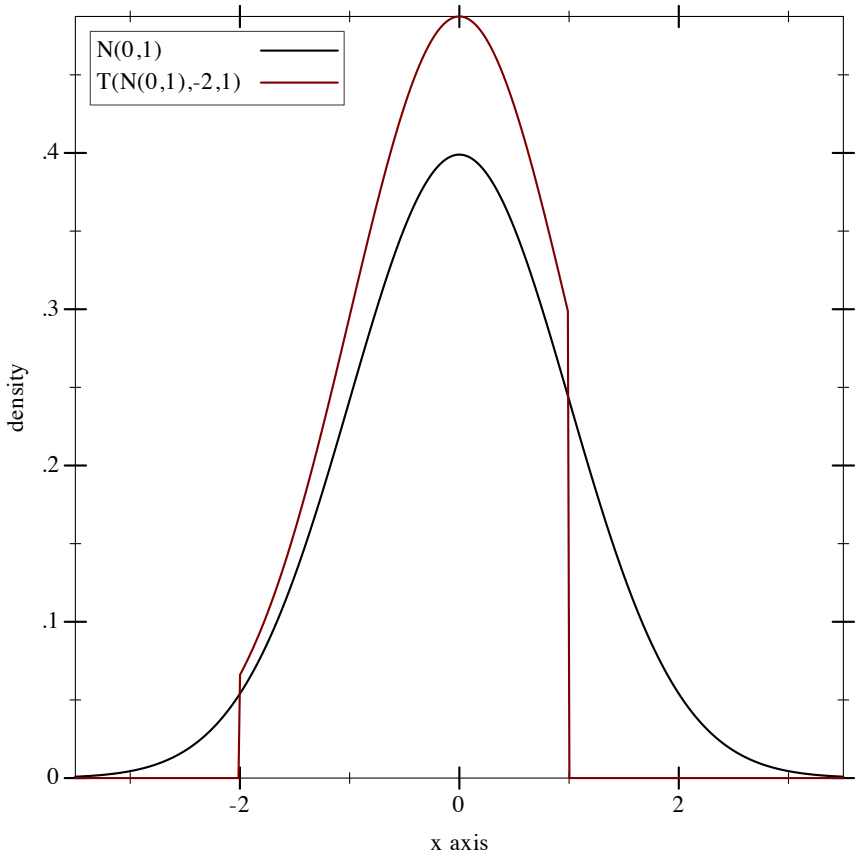
Represents distributions like *d*, but with zero density for `x < min` and for `x > max`. The probability of the interval [*min*, *max*] is renormalized to one.

`(truncated-dist d)` is equivalent to `(truncated-dist d -inf.0 +inf.0)`. `(truncated-dist d max)` is equivalent to `(truncated-dist d -inf.0 max)`. If `min > max`, they are swapped before constructing the distribution object.

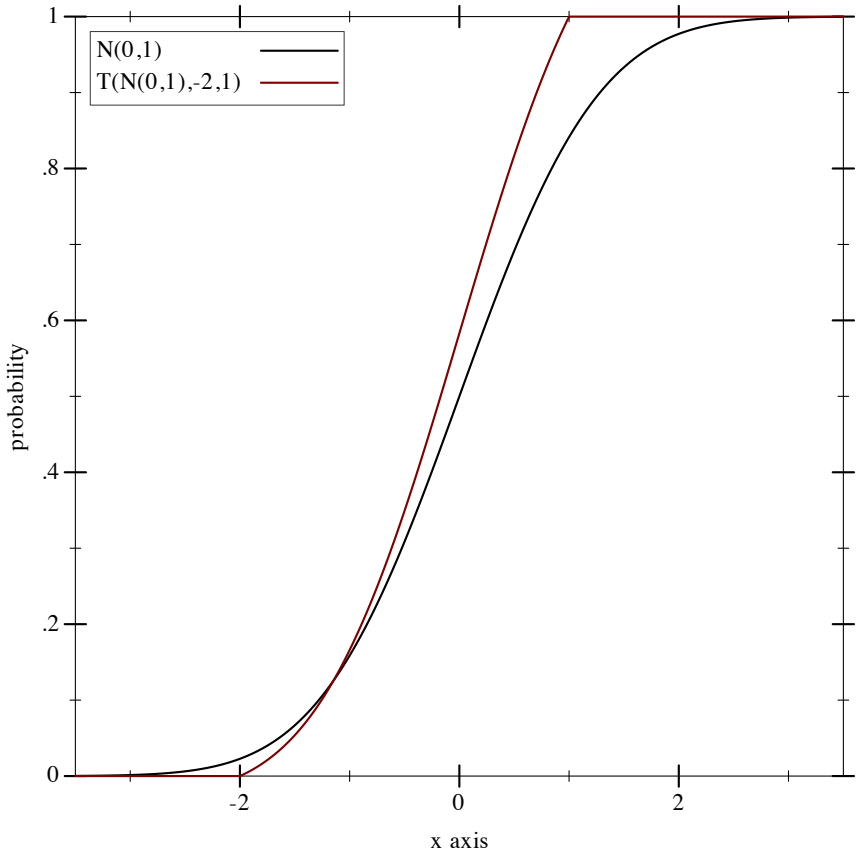
Samples are taken by applying the truncated distribution's inverse cdf to uniform samples.

Examples:

```
> (define d (normal-dist))
> (define t (truncated-dist d -2 1))
> t
(truncated-dist (normal-dist 0.0 1.0) -2.0 1.0)
> (plot (list (function (distribution-pdf d) #:label "N(0,1)" #:color 0)
              (function (distribution-pdf t) #:label "T(N(0,1),-
2,1)"))
        #:x-min -3.5 #:x-max 3.5 #:y-label "density")
```




```
> (plot (list (function (ordered-dist-cdf d) #:label "N(0,1)" #:color 0)
              (function (ordered-dist-cdf t) #:label "T(N(0,1),-
2,1)"))
        #:x-min -3.5 #:x-max 3.5 #:y-label "probability")
```



9.5.12 Uniform Distributions

Wikipedia: Uniform Distribution.

```
Uniform-Dist
(uniform-dist) → Uniform-Dist
(uniform-dist max) → Uniform-Dist
  max : Real
(uniform-dist min max) → Uniform-Dist
  min : Real
  max : Real
(uniform-dist-min d) → Flonum
  d : Uniform-Dist
(uniform-dist-max d) → Flonum
  d : Uniform-Dist
```

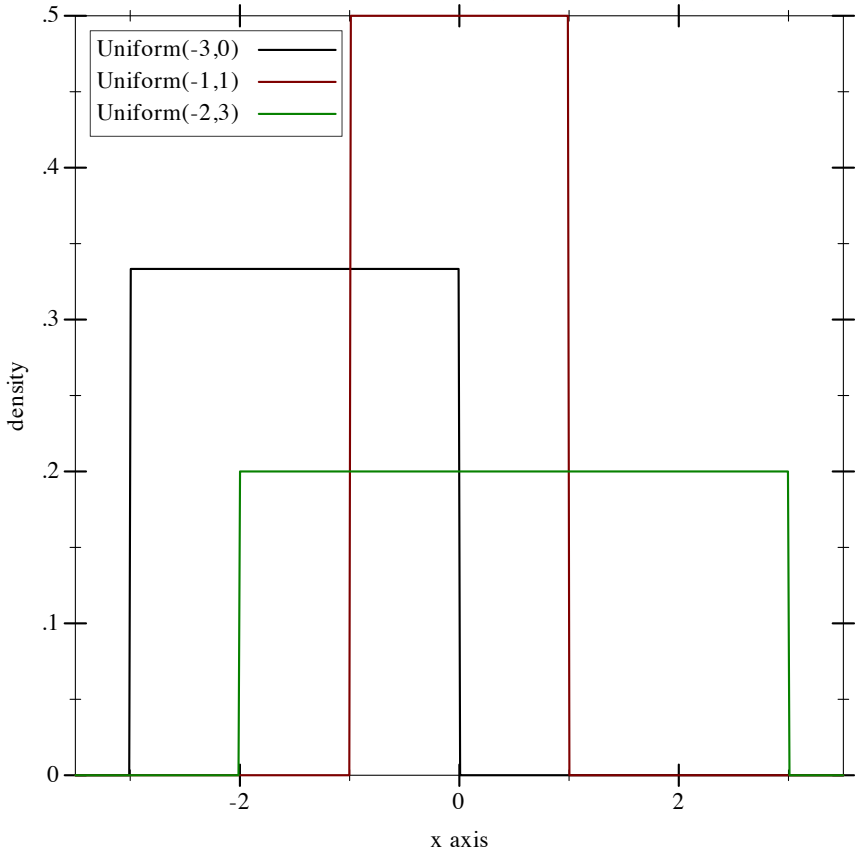
Represents the uniform distribution family parameterized by minimum and maximum.

`(uniform-dist)` is equivalent to `(uniform-dist 0 1)`. `(uniform-dist max)` is

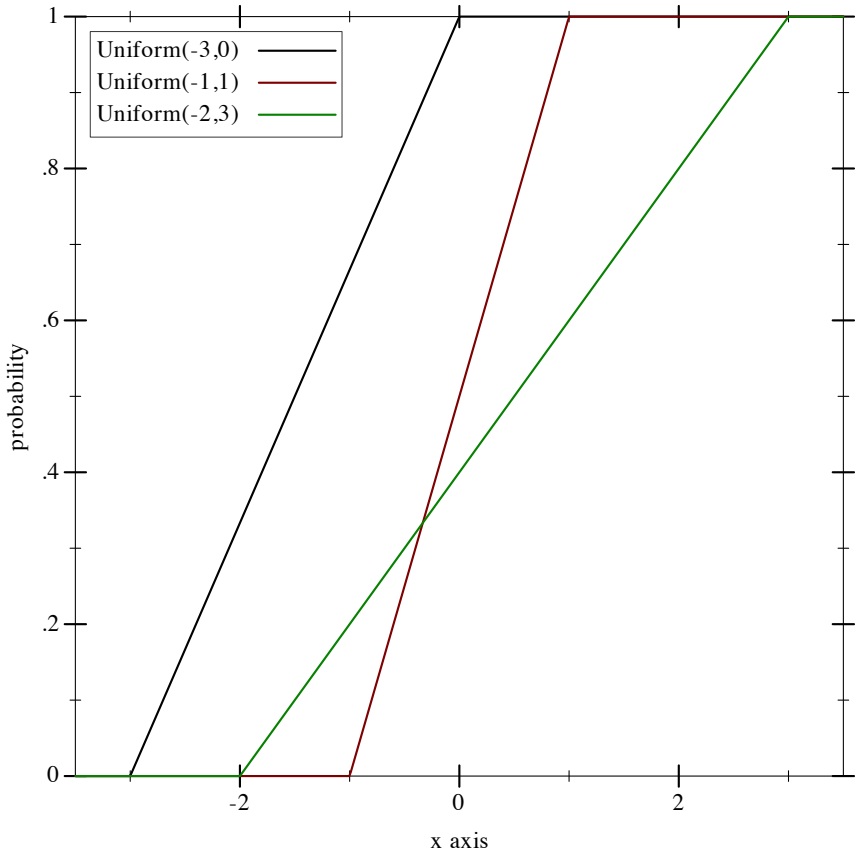
equivalent to `(uniform-dist 0 max)`. If `max < min`, they are swapped before constructing the distribution object.

Examples:

```
> (plot (for/list ([a  (in-list '(-3 -1 -2))]
                   [b  (in-list '(0 1 3))]
                   [i  (in-naturals)])
          (function (distribution-pdf (uniform-
dist a b)) #:color i
                    #:label (format "Uniform(~a,~a)" a b)))
        #:x-min -3.5 #:x-max 3.5 #:y-label "density")
```




```
> (plot (for/list ([a  (in-list '(-3 -1 -2))]
                   [b  (in-list '(0 1 3))]
                   [i  (in-naturals)])
          (function (ordered-dist-cdf (uniform-
dist a b)) #:color i
                    #:label (format "Uniform(~a,~a)" a b)))
        #:x-min -3.5 #:x-max 3.5 #:y-label "probability")
```



`(uniform-dist x x)` for any real `x` behaves like a support-limited delta distribution centered at `x`.

9.6 Low-Level Distribution Functions

The following functions are provided for users who need lower overhead than that of distribution objects, such as untyped Racket users (currently), and library writers who are implementing their own distribution abstractions.

Because applying these functions is meant to be fast, none of them have optional arguments. In particular, the boolean flags `log?` and `1-p?` are always required.

Every low-level function's argument list begins with the distribution family parameters. In the case of pdfs and cdfs, these arguments are followed by a domain value and boolean flags. In the case of inverse cdfs, they are followed by a probability argument and boolean flags. For sampling procedures, the distribution family parameters are followed by the requested

number of random samples.

Generally, prob is a probability parameter, k is an integer domain value, x is a real domain value, p is the probability argument to an inverse cdf, and n is the number of random samples.

9.6.1 Integer Distribution Functions

```
(flbernoulli-pdf prob k log?) → Flonum
  prob : Flonum
  k : Flonum
  log? : Any
(flbernoulli-cdf prob k log? 1-p?) → Flonum
  prob : Flonum
  k : Flonum
  log? : Any
  1-p? : Any
(flbernoulli-inv-cdf prob p log? 1-p?) → Flonum
  prob : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
(flbernoulli-sample prob n) → FlVector
  prob : Flonum
  n : Integer
```

Low-level flonum functions used to implement bernoulli-dist.

```
(flbinomial-pdf count prob k log?) → Flonum
  count : Flonum
  prob : Flonum
  k : Flonum
  log? : Any
(flbinomial-cdf count prob k log? 1-p?) → Flonum
  count : Flonum
  prob : Flonum
  k : Flonum
  log? : Any
  1-p? : Any
(flbinomial-inv-cdf count prob p log? 1-p?) → Flonum
  count : Flonum
  prob : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
```

```
(flbinomial-sample count prob n) → FlVector
  count : Flonum
  prob : Flonum
  n : Integer
```

Low-level flonum functions used to implement binomial-dist.

```
(flgeometric-pdf prob k log?) → Flonum
  prob : Flonum
  k : Flonum
  log? : Any
(flgeometric-cdf prob k log? 1-p?) → Flonum
  prob : Flonum
  k : Flonum
  log? : Any
  1-p? : Any
(flgeometric-inv-cdf prob p log? 1-p?) → Flonum
  prob : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
(flgeometric-sample prob n) → FlVector
  prob : Flonum
  n : Integer
```

Low-level flonum functions used to implement geometric-dist.

```
(flpoisson-pdf mean k log?) → Flonum
  mean : Flonum
  k : Flonum
  log? : Any
(flpoisson-cdf mean k log? 1-p?) → Flonum
  mean : Flonum
  k : Flonum
  log? : Any
  1-p? : Any
(flpoisson-inv-cdf mean p log? 1-p?) → Flonum
  mean : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
(flpoisson-sample mean n) → FlVector
  mean : Flonum
  n : Integer
(flpoisson-median mean) → Flonum
  mean : Flonum
```

Low-level flonum functions used to implement `poisson-dist`.

`(flpoisson-median mean)` runs faster than `(flpoisson-inv-cdf mean 0.5 #f #f)`, significantly so when *mean* is large.

9.6.2 Real Distribution Functions

```
(flbeta-pdf alpha beta x log?) → Flonum
  alpha : Flonum
  beta : Flonum
  x : Flonum
  log? : Any
(flbeta-cdf alpha beta x log? 1-p?) → Flonum
  alpha : Flonum
  beta : Flonum
  x : Flonum
  log? : Any
  1-p? : Any
(flbeta-inv-cdf alpha beta p log? 1-p?) → Flonum
  alpha : Flonum
  beta : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
(flbeta-sample alpha beta n) → FlVector
  alpha : Flonum
  beta : Flonum
  n : Integer
```

Low-level flonum functions used to implement `beta-dist`.

```
(flcauchy-pdf mode scale x log?) → Flonum
  mode : Flonum
  scale : Flonum
  x : Flonum
  log? : Any
(flcauchy-cdf mode scale x log? 1-p?) → Flonum
  mode : Flonum
  scale : Flonum
  x : Flonum
  log? : Any
  1-p? : Any
```

```
(flcauchy-inv-cdf mode scale p log? 1-p?) → Flonum
  mode : Flonum
  scale : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
(flcauchy-sample mode scale n) → FlVector
  mode : Flonum
  scale : Flonum
  n : Integer
```

Low-level flonum functions used to implement `cauchy-dist`.

```
(fldelta-pdf mean x log?) → Flonum
  mean : Flonum
  x : Flonum
  log? : Any
(fldelta-cdf mean x log? 1-p?) → Flonum
  mean : Flonum
  x : Flonum
  log? : Any
  1-p? : Any
(fldelta-inv-cdf mean p log? 1-p?) → Flonum
  mean : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
```

Low-level flonum functions used to implement `delta-dist`.

To get delta-distributed random samples, use `(make-flvector n mean)`.

```
(flexponential-pdf mean x log?) → Flonum
  mean : Flonum
  x : Flonum
  log? : Any
(flexponential-cdf mean x log? 1-p?) → Flonum
  mean : Flonum
  x : Flonum
  log? : Any
  1-p? : Any
(flexponential-inv-cdf mean p log? 1-p?) → Flonum
  mean : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
```

```
(flexponential-sample mean n) → FlVector
  mean : Flonum
  n : Integer
```

Low-level flonum functions used to implement exponential-dist.

```
(flgamma-pdf shape scale x log?) → Flonum
  shape : Flonum
  scale : Flonum
  x : Flonum
  log? : Any
(flgamma-cdf shape scale x log? 1-p?) → Flonum
  shape : Flonum
  scale : Flonum
  x : Flonum
  log? : Any
  1-p? : Any
(flgamma-inv-cdf shape scale p log? 1-p?) → Flonum
  shape : Flonum
  scale : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
(flgamma-sample shape scale n) → FlVector
  shape : Flonum
  scale : Flonum
  n : Integer
```

Low-level flonum functions used to implement gamma-dist.

```
(fllogistic-pdf mean scale x log?) → Flonum
  mean : Flonum
  scale : Flonum
  x : Flonum
  log? : Any
(fllogistic-cdf mean scale x log? 1-p?) → Flonum
  mean : Flonum
  scale : Flonum
  x : Flonum
  log? : Any
  1-p? : Any
(fllogistic-inv-cdf mean scale p log? 1-p?) → Flonum
  mean : Flonum
  scale : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
```

```
(fllogistic-sample mean scale n) → FlVector
  mean : Flonum
  scale : Flonum
  n : Integer
```

Low-level flonum functions used to implement `logistic-dist`.

```
(flnormal-pdf mean stddev x log?) → Flonum
  mean : Flonum
  stddev : Flonum
  x : Flonum
  log? : Any
(flnormal-cdf mean stddev x log? 1-p?) → Flonum
  mean : Flonum
  stddev : Flonum
  x : Flonum
  log? : Any
  1-p? : Any
(flnormal-inv-cdf mean stddev p log? 1-p?) → Flonum
  mean : Flonum
  stddev : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
(flnormal-sample mean stddev n) → FlVector
  mean : Flonum
  stddev : Flonum
  n : Integer
```

Low-level flonum functions used to implement `normal-dist`.

```
(fltriangle-pdf min max mode x log?) → Flonum
  min : Flonum
  max : Flonum
  mode : Flonum
  x : Flonum
  log? : Any
(fltriangle-cdf min max mode x log? 1-p?) → Flonum
  min : Flonum
  max : Flonum
  mode : Flonum
  x : Flonum
  log? : Any
  1-p? : Any
```

```
(fltriangle-inv-cdf min max mode p log? 1-p?) → Flonum
  min : Flonum
  max : Flonum
  mode : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
(fltriangle-sample min max mode n) → FlVector
  min : Flonum
  max : Flonum
  mode : Flonum
  n : Integer
```

Low-level flonum functions used to implement `triangle-dist`.

```
(fluniform-pdf min max x log?) → Flonum
  min : Flonum
  max : Flonum
  x : Flonum
  log? : Any
(fluniform-cdf min max x log? 1-p?) → Flonum
  min : Flonum
  max : Flonum
  x : Flonum
  log? : Any
  1-p? : Any
(fluniform-inv-cdf min max p log? 1-p?) → Flonum
  min : Flonum
  max : Flonum
  p : Flonum
  log? : Any
  1-p? : Any
(fluniform-sample min max n) → FlVector
  min : Flonum
  max : Flonum
  n : Integer
```

Low-level flonum functions used to implement `uniform-dist`.

10 Stuff That Doesn't Belong Anywhere Else

```
(require math/utils)
```
package: math-lib

10.1 Parallelization

```
(max-math-threads) → Positive-Integer
(max-math-threads num) → void?
  num : Positive-Integer
```

The maximum number of threads a parallelized `math` function will use. The default value is `(max 1 (processor-count))`.

10.2 Discrete Fourier Transform Conventions

```
(dft-convention) → (List Real Real)
(dft-convention lst) → void?
  lst : (List Real Real)
```

A parameter controlling the convention used for scaling discrete Fourier transforms, such as those performed by `array-fft`. The default value is `'(1 -1)`, which represents the convention used in signal processing.

In general, if *lst* is `(list a b)` and `n` is the length of a transformed array axis or vector, then

- Each sum is scaled by `(expt n (/ (- a 1) 2))`.
- Each exponential in the sum has its argument scaled by `b`.

Conveniently, a Fourier transform with convention `(list (- a) (- b))` is the inverse of a Fourier transform with convention `(list a b)`.

See Mathematica's documentation on `Fourier`, from which this excellent idea was stolen.

```
(dft-inverse-convention) → (List Real Real)
```

Returns the convention used for inverse Fourier transforms, given the current convention.

10.3 Floating-Point Compliance Testing

```
(test-floating-point n) → (Listof (List Any Any))
  n : Natural
```

Runs a comprehensive test of the system's IEEE 754 (floating-point) compliance, and reports unexpected inaccuracies and errors.

In each test, a function is applied to some carefully chosen values, as well as *n* additional random values. Its corresponding bigfloat function is applied to the same values, and the answers are compared. Each test returns a list of failures, which are appended and returned.

Each failure in a failure list is formatted

```
(list (list name args ...) reason)
```

where `name` is the name of a function, such as `'fl+`, `args ...` are the arguments it was applied to, and `reason` is the reason for the failure.

If `reason` is a flonum, the failure was due to inaccuracy. For example,

```
(list (list 'fl+ 4.5 2.3) 0.76)
```

means the result of `(fl+ 4.5 2.3)` was off by `0.76` ulps.

The threshold for reporting unexpected inaccuracy depends on the function tested. All the arithmetic and irrational functions exported by `racket/flonum`, for example, must have no more than `0.5` ulps error in order to be compliant.

Two other possible failure reasons are

```
(list 'different-zero 0.0 -0.0)
(list 'different-zero -0.0 0.0)
```

The first zero is the answer returned by the function, and the second zero is the expected answer.

Other possible failure reasons have the form

```
(list 'not-fl2? x y)
```

meaning that the result `(values x y)` is not a valid flonum expansion. Such reasons are only given for failures of functions whose names begin with `fl2` or contain `/error`. These functions are currently undocumented, but are used to implement many `math/flonum`, `math/special-functions`, and `math/distributions` functions.

Tests of functions that operate on and return flonum expansions are the strictest tests, requiring hardware arithmetic to be perfectly IEEE 754 compliant. They reliably fail on seemingly innocuous noncompliant behavior, such as computing intermediate results with 80-bit precision.

```
(print-fp-test-progress?) → Boolean
(print-fp-test-progress? print?) → void?
  print? : Boolean
```

When `(print-fp-test-progress?)` is `#t`, floating-point tests print and flush a representation of their progress as they run. The default value is `#t`.