Data: Data Structures

Version 9.4.0.5

November 16, 2026

This manual documents data structure libraries available in the `data` collection.

1 Imperative Queues

```
(require data/queue)
```
package: base

This module provides a simple mutable queue representation, providing first-in/first-out semantics.

Operations on queues mutate it in a thread-unsafe way.

```
(make-queue) → queue?
```

Produces an empty queue.

```
(enqueue! q v) → void?
  q : queue?
  v : any/c
```

Adds an element to the back of a queue.

This takes constant time, independent of the number of elements in `q`.

```
(enqueue-front! q v) → void?
  q : queue?
  v : any/c
```

Adds an element to the front of a queue.

This takes constant time, independent of the number of elements in `q`.

```
(dequeue! q) → any/c
  q : non-empty-queue?
```

Removes an element from the front of a non-empty queue, and returns that element.

This takes constant time, independent of the number of elements in `q`.

Examples:

```
(define q (make-queue))
> (enqueue! q 1)
> (dequeue! q)
1
> (enqueue! q 2)
> (enqueue! q 3)
> (dequeue! q)
```

```
2
> (dequeue! q)
3
> (enqueue! q 2)
> (enqueue! q 1)
> (enqueue-front! q 3)
> (enqueue-front! q 4)
> (queue->list q)
'(4 3 2 1)
```

```
(queue-filter! q pred?) → void?
  q : queue?
  pred? : (-> any/c any/c)
```

Applies *pred?* to each element of the queue, removing any where *pred?* returns `#f`.

This takes time proportional to the number of elements in *q* (assuming that *pred?* takes constant time, independent of the number of elements in *q*). It does not allocate and it calls *pred?* exactly once for each element of *q*.

Examples:

```
(define q (make-queue))
> (enqueue! q 1)
> (enqueue! q 2)
> (enqueue! q 3)
> (enqueue! q 4)
> (queue-filter! q even?)
> (queue->list q)
'(2 4)
```

```
(queue->list q) → (listof any/c)
  q : queue?
```

Returns an immutable list containing the elements of the queue in the order the elements were added.

This takes time proportional to the number of elements in *q*.

Examples:

```
(define q (make-queue))
> (enqueue! q 8)
> (enqueue! q 9)
> (enqueue! q 0)
> (queue->list q)
'(8 9 0)
```

```
(queue-length q) → exact-nonnegative-integer?
  q : queue?
```

Returns the number of elements in the queue.

This takes constant time, independent of the number of elements in `q`.

Examples:

```
(define q (make-queue))
> (queue-length q)
0
> (enqueue! q 5)
> (enqueue! q 12)
> (queue-length q)
2
> (dequeue! q)
5
> (queue-length q)
1
```

```
(queue-empty? q) → boolean?
  q : queue?
```

Recognizes whether a queue is empty or not.

This takes constant time, independent of the number of elements in `q`.

Examples:

```
(define q (make-queue))
> (queue-empty? q)
#t
> (enqueue! q 1)
> (queue-empty? q)
#f
> (dequeue! q)
1
> (queue-empty? q)
#t
```

```
(queue? v) → boolean?
  v : any/c
```

This predicate recognizes queues.

This takes constant time, independent of the size of the argument *v*.

Examples:

```
> (queue? (make-queue))
#t
> (queue? 'not-a-queue)
#f
```

```
(non-empty-queue? v) → boolean?
  v : any/c
```

This predicate recognizes non-empty queues.

This takes constant time, independent of the size of the argument *v*.

Examples:

```
> (non-empty-queue? (let ([q (make-queue)])
                      (enqueue! q 1)
                      q))
#t
> (non-empty-queue? (make-queue))
#f
> (non-empty-queue? 'not-a-queue)
#f
```

```
(in-queue q) → sequence?
  q : queue?
```

Returns a sequence whose elements are the elements of *q*.

```
queue/c : flat-contract?
nonempty-queue/c : flat-contract?
```

These are provided for backwards compatibility. They are identical to queue? and non-empty-queue?, respectively.

2 Growable Vectors

```
(require data/gvector)
```
package: data-lib

A growable vector (gvector) is a mutable sequence whose length can change over time. A gvector also acts as a dictionary (`dict?` from `racket/dict`), where the keys are zero-based indexes and the values are the elements of the gvector. A gvector can be extended by adding an element to the end, and it can be shrunk by removing any element, although removal can take time linear in the number of elements in the gvector.

Two gvectors are `equal?` if they contain the same number of elements and if they contain equal elements at each index.

Operations on gvectors are not thread-safe.

Additionally, gvectors are serializable with the `racket/serialize` collection.

```
(make-gvector [#:capacity capacity]) → gvector?
  capacity : exact-nonnegative-integer? = 10
```

Creates a new empty gvector with an initial capacity of *capacity*.

Changed in version 1.3 of package data-lib: Allowed a *capacity* of 0.

```
(gvector elem ...) → gvector?
  elem : any/c
```

Creates a new gvector containing each *elem* in order.

```
(gvector? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a gvector, `#f` otherwise.

```
(gvector-ref gv index [default]) → any/c
  gv : gvector?
  index : exact-nonnegative-integer?
  default : any/c = (error ....)
```

Returns the element at index *index*, if *index* is less than `(gvector-count gv)`. Otherwise, *default* is invoked if it is a procedure, returned otherwise.

```
(gvector-add! gv value ...) → void?
  gv : gvector?
  value : any/c
```

Adds each *value* to the end of the gvector *gv*. Takes (amortized) time proportional to the number of added *value*s.

```
(gvector-append! gv other) → void?
  gv : gvector?
  other : gvector?
```

Adds the elements of *other* to the end of *gv*, in order, and leaves *other* unchanged; the two may be the same gvector. Takes (amortized) time proportional to `(gvector-count other)`.

Added in version 1.3 of package `data-lib`.

```
(gvector-append gv other) → gvector?
  gv : gvector?
  other : gvector?
```

Returns a new gvector with the elements of *gv* followed by those of *other*, and leaves both unchanged.

Added in version 1.3 of package `data-lib`.

```
(gvector-insert! gv index value) → void?
  gv : gvector?
  index : (and/c exact-nonnegative-integer?
                 (</c (+ 1 (gvector-count gv))))
  value : any/c
```

Adds the *value* to the gvector *gv* at index *index*, shifting all remaining elements by one element. Takes time proportional to `(- (gvector-count gv) index)`.

```
(gvector-set! gv index value) → void?
  gv : gvector?
  index : (and/c exact-nonnegative-integer?
                 (</c (+ 1 (gvector-count gv))))
  value : any/c
```

Sets the value at index *index* to be *value*. If *index* is `(gvector-count gv)`—that is, one more than the greatest used index—the effect is the same as `(gvector-add! gv value)`.

```
(gvector-remove! gv index) → void?
  gv : gvector?
  index : (and/c exact-nonnegative-integer?
                 (</c (gvector-count gv)))
```

Removes the item at *index*, shifting items at higher indexes down. Takes time proportional to `(- (gvector-count gv) index)`.

```
(gvector-remove-last! gv) → any/c
  gv : gvector?
```

Removes the element at the end and returns it, or raises an exception if *gv* is empty. Takes (amortized) constant time.

```
(gvector-count gv) → exact-nonnegative-integer?
  gv : gvector?
```

Returns the number of items in *gv*.

```
(gvector->vector gv) → vector?
  gv : gvector?
```

Returns a vector of length `(gvector-count gv)` containing the elements of *gv* in order.

```
(vector->gvector v) → gvector?
  v : vector?
```

Returns a gvector of length `(vector-length v)` containing the elements of *v* in order.

```
(gvector->list gv) → list?
  gv : gvector?
```

Returns a list of length `(gvector-count gv)` containing the elements of *gv* in order.

```
(list->gvector l) → gvector?
  l : list?
```

Returns a gvector of length `(length l)` containing the elements of *l* in order.

```
(in-gvector gv) → sequence?
  gv : gvector?
```

Returns a sequence whose elements are the elements of *gv*. Mutation of *gv* while the sequence is running changes the elements produced by the sequence. To obtain a sequence from a snapshot of *gv*, use `(in-vector (gvector->vector gv))` instead.

```
(for/gvector maybe-capacity (for-clause ...) body ...+)

maybe-capacity =
               | #:capacity capacity-expr
(for*/gvector maybe-capacity (for-clause ...) body ...+)
```

Analogous to `for/list` and `for*/list`, but constructs a gvector instead of a list. The gvector starts with the capacity that *`capacity-expr`* produces, or with the default capacity of `make-gvector` without one.

Unlike `for/list`, the *`body`* may return zero or multiple values; all returned values are added to the gvector, in order, on each iteration.

Changed in version 1.3 of package `data-lib`: Added `#:capacity`.

3 Orders and Ordered Dictionaries

```
(require data/order)
```
package: `data-lib`

This library defines *orders* and the *ordered dictionary* generic interface.

```
ordering/c : flat-contract?
```

Contract for orderings, represented by the symbols `'=`, `'<`, and `'>`.

```
gen:ordered-dict : any/c
```

A generic interface for defining new ordered dictionary types. Methods can be attached to the `gen:ordered-dict` interface using the `#:methods` keyword in a structure type definition. Two "extrema" methods and four "search" methods should be implemented. The extrema methods must satisfy *e/c* and the search methods must satisfy *s/c*:

```
e/c = (->i ([d ordered-dict?])
           [_ (d) (or/c #f (dict-iter-contract d))])
s/c = (->i ([d ordered-dict?]
            [k (d) (dict-key-contract d)])
            [_ (d) (or/c #f (dict-iter-contract d))])
```

The methods are implementations of the following generic functions:

- `dict-iterate-least`
- `dict-iterate-greatest`
- `dict-iterate-least/>?`
- `dict-iterate-least/>=?`
- `dict-iterate-greatest/<?`
- `dict-iterate-greatest/<=?`

A struct type that implements `gen:ordered-dict` must also implement `gen:dict`.

```
prop:ordered-dict : (struct-type-property/c
                     (vectorof e/c e/c s/c s/c s/c s/c))
```

A deprecated structure type property used to defined custom ordered dictionaries. Use `gen:ordered-dict` instead. Accepts a vector of 6 procedures with the same arguments as the methods of `gen:ordered-dict`.

```
(ordered-dict? x) → boolean?
  x : any/c
```

Returns #t if *x* is an instance of a struct implementing the ordered dictionary interface (via gen:ordered-dict).

```
(dict-iterate-least dict) → (or/c (dict-iter-contract dict) #f)
  dict : ordered-dict?
(dict-iterate-greatest dict)
 → (or/c (dict-iter-contract dict) #f)
  dict : ordered-dict?
```

Returns the position of the least (greatest) key in the ordered dictionary *dict*. If *dict* is empty, #f is returned.

```
(dict-iterate-least/>? dict key)
 → (or/c (dict-iter-contract dict) #f)
  dict : ordered-dict?
  key : any/c
(dict-iterate-least/>=? dict key)
 → (or/c (dict-iter-contract dict) #f)
  dict : ordered-dict?
  key : any/c
(dict-iterate-greatest/<? dict key)
 → (or/c (dict-iter-contract dict) #f)
  dict : ordered-dict?
  key : any/c
(dict-iterate-greatest/<=? dict key)
 → (or/c (dict-iter-contract dict) #f)
  dict : ordered-dict?
  key : any/c
```

Returns the position of the least key greater than *key*, the least key greater than or equal to *key*, the greatest key less than *key*, and the greatest key less than or equal to *key*, respectively. If no key satisfies the criterion, #f is returned.

```
(order name domain-contract comparator)
 → (and/c order? procedure?)
  name : symbol?
  domain-contract : contract?
  comparator : (-> any/c any/c ordering/c)
(order name domain-contract =? <? [>?]) → (and/c order? procedure?)
  name : symbol?
  domain-contract : contract?
```

```
  =? : (-> any/c any/c boolean?)
  <? : (-> any/c any/c boolean?)
  >? : (-> any/c any/c boolean?) = (lambda (x y) (<? y x))
```

Produces an order object encapsulating a domain contract and a comparator function. If a single procedure is given, it is used directly as the comparator. If two or three procedures are given, they are used to construct the comparator. The given *name* names the order object as returned by `object-name`.

The *domain-contract* is not applied to the comparison function; rather, clients of the order are advised to incorporate the domain contracts into their own contracts. For example, when a splay-tree (see `data/splay-tree`) is constructed with an order, it applies the domain-contract to its keys. Thus the contract is checked once per dictionary procedure call, rather than on every comparison.

An order object is applicable as a procedure; it behaves as its comparator.

Examples:

```
> (define string-order (order 'string-order string? string=? string<?))
> (string-order "abc" "acdc")
'<
> (string-order "x" 12)
string=?: contract violation
  expected: string?
  given: 12
```

Changed in version 1.2.1 of package `data-lib`: Changed the name of an order object as returned by `object-name` to *name* instead of `(object-name comparator)`.

```
(order? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is an order object, `#f` otherwise.

```
(order-comparator ord) → (-> any/c any/c ordering/c)
  ord : order?
```

Extracts the comparator function from an order object.

```
(order-domain-contract ord) → contract?
  ord : order?
```

Extracts the domain contract from an order object.

```
(order-=? ord) → (-> any/c any/c boolean?)
  ord : order?
(order-<? ord) → (-> any/c any/c boolean?)
  ord : order?
```

Returns a procedure representing the order's equality relation or less-than relation, respectively.

```
real-order : order?
```

The order of the real numbers. The domain of `real-order` excludes `+nan.0` but includes `+inf.0` and `-inf.0`. The standard numeric comparisons (`=`, `<`) are used; exact `1` is equal to inexact `1.0`.

Examples:

```
> (real-order 1.0 1)
'=
> (real-order 5 7)
'<
> (real-order 9.0 3.4)
'>
> (real-order 1 +inf.0)
'<
> (real-order 5 -inf.0)
'>
```

```
datum-order : order?
```

An ad hoc order that encompasses many built-in Racket data types as well as prefab structs and fully-transparent structs. The `datum-order` comparator orders values of the same data type according to the data type's natural order: `string=?`, `string<?` for strings, for example (but see the warning about numbers below). Different data types are ordered arbitrarily but contiguously; for example, all strings sort before all vectors, or vice versa. Prefab and fully-transparent structs are ordered according to their most specific struct type, and prefab structs are ordered first by their prefab struct keys. The ordering of struct types is independent of the struct type hierarchy; a struct type may sort before one of its subtypes but after another.

Programs should not rely on the ordering of different data types, since it may change in future versions of Racket to improve comparison performance. The ordering of non-prefab struct types may change between one execution of a program and the next.

The order is guaranteed, however, to lexicographically sort proper lists, vectors, prefab structs, and fully-transparent structs. Improper lists sort lexicographically considered as

pairs, but the ordering of an improper list and its proper prefix, such as `'(a b . c)` and `'(a b)`, is not specified.

The `datum-order` comparator does not perform cycle-detection; comparisons involving cyclic data may diverge.

Warning: `datum-order` is not compatible with the standard numeric order; all exact numbers are ordered separately from all inexact numbers. Thus `1` is considered distinct from `1.0`, for example.

The following data types are currently supported: numbers, strings, bytes, keywords, symbols, booleans, characters, paths, null, pairs, vectors, boxes, prefab structs, and fully-transparent structs.

The following example comparisons are specified to return the results shown:

```
> (datum-order 1 2)
'<
> (datum-order 8.0 5.0)
'>
> (datum-order 'apple 'candy)
'<
> (datum-order '(a #:b c) '(a #:c d c))
'<
> (datum-order '(5 . 4) '(3 2 1))
'>
> (datum-order '(a b . c) '(a b . z))
'<
> (datum-order "apricot" "apple")
'>
> (datum-order '#(1 2 3) '#(1 2))
'>
> (datum-order '#(1 2 3) '#(1 3))
'<
> (datum-order (box 'car) (box 'candy))
'>
> (datum-order '#s(point a 1) '#s(point b 0))
'<
> (datum-order '#s(A 1 2) '#s(Z 3 4 5))
'<
> (struct fish (name) #:transparent)
> (datum-order (fish 'alewife) (fish 'sockeye))
'<
```

The following example comparisons are unspecified but consistent within all executions of a single version of Racket:

```
(datum-order 1 2.0)
(datum-order 3+5i 3+2i)
(datum-order 'apple "zucchini")
(datum-order '(a b) '(a b . c))
(datum-order 0 'zero)
```

The following example comparison is unspecified but consistent within a single execution of a program:

```
(struct fowl (name) #:transparent)
(datum-order (fish 'alewife) (fowl 'dodo))
```

4 Splay Trees

```
(require data/splay-tree)
```
package: data-lib

Splay trees are an efficient data structure for mutable dictionaries with totally ordered keys. They were described in the paper "Self-Adjusting Binary Search Trees" by Daniel Sleator and Robert Tarjan in Journal of the ACM 32(3) pp652-686.

A splay-tree is a ordered dictionary (`dict?` and `ordered-dict?`).

Operations on splay-trees are not thread-safe. If a key in a splay-tree is mutated, the splay-tree's internal invariants may be violated, causing its behavior to become unpredictable.

```
(make-splay-tree [ord
                  #:key-contract key-contract
                  #:value-contract value-contract]) → splay-tree?
  ord : order? = datum-order
  key-contract : contract? = any/c
  value-contract : contract? = any/c
```

Makes a new empty splay-tree. The splay tree uses *ord* to order keys; in addition, the domain contract of *ord* is combined with *key-contract* to check keys.

Examples:

```
> (define splay-tree
    (make-splay-tree (order 'string-order string? string=? string<?)))
> (splay-tree-set! splay-tree "dot" 10)
> (splay-tree-set! splay-tree "cherry" 500)
> (dict-map splay-tree list)
'(("cherry" 500) ("dot" 10))
> (splay-tree-ref splay-tree "dot")
10
> (splay-tree-remove! splay-tree "cherry")
> (splay-tree-count splay-tree)
1
> (splay-tree-set! splay-tree 'pear 3)
splay-tree-set!: contract violation
  expected: string?
  given: 'pear
  in: the key argument of
      (->i
       ((s splay-tree?)
        (key (s) (key-c s))
        (v (s) (val-c s)))
       (_r void?))
```

contract from:
<pkgs>/data-lib/data/splay-tree.rkt
blaming: top-level
(assuming the contract is correct)
at: <pkgs>/data-lib/data/splay-tree.rkt:609:2

```
(make-adjustable-splay-tree [#:key-contract key-contract
                             #:value-contract value-contract])
 → splay-tree?
  key-contract : contract? = any/c
  value-contract : contract? = any/c
```

Makes a new empty splay-tree that permits only exact integers as keys (in addition to any constraints imposed by *key-contract*). The resulting splay tree answers true to `adjustable-splay-tree?` and supports efficient key adjustment.

Examples:

```
> (define splay-tree (make-adjustable-splay-tree))
> (splay-tree-set! splay-tree 3 'apple)
> (splay-tree-set! splay-tree 6 'cherry)
> (dict-map splay-tree list)
'((3 apple) (6 cherry))
> (splay-tree-ref splay-tree 3)
'apple
> (splay-tree-remove! splay-tree 6)
> (splay-tree-count splay-tree)
1
```

```
(splay-tree? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a splay-tree, `#f` otherwise.

```
(adjustable-splay-tree? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a splay-tree that supports key adjustment; see `splay-tree-contract!` and `splay-tree-expand!`.

```
(splay-tree-ref s key [default]) → any
  s : splay-tree?
  key : any/c
  default : any/c = (lambda () (error ....))
```

```
(splay-tree-set! s key value) → void?
  s : splay-tree?
  key : any/c
  value : any/c
(splay-tree-remove! s key) → void?
  s : splay-tree?
  key : any/c
(splay-tree-count s) → exact-nonnegative-integer?
  s : splay-tree?
(splay-tree-iterate-first s) → (or/c #f splay-tree-iter?)
  s : splay-tree?
(splay-tree-iterate-next s iter) → (or/c #f splay-tree-iter?)
  s : splay-tree?
  iter : splay-tree-iter?
(splay-tree-iterate-key s iter) → any/c
  s : splay-tree?
  iter : splay-tree-iter?
(splay-tree-iterate-value s iter) → any/c
  s : splay-tree?
  iter : splay-tree-iter?
```

Implementations of `dict-ref`, `dict-set!`, `dict-remove!`, `dict-count`, `dict-iterate-first`, `dict-iterate-next`, `dict-iterate-key`, and `dict-iterate-value`, respectively.

```
(splay-tree-remove-range! s from to) → void?
  s : splay-tree?
  from : any/c
  to : any/c
```

Removes all keys in [*from*, *to*); that is, all keys greater than or equal to *from* and less than *to*.

This operation takes *O(N)* time, or *O(log N)* time if `(adjustable-splay-tree? s)`.

```
(splay-tree-contract! s from to) → void?
  s : adjustable-splay-tree?
  from : exact-integer?
  to : exact-integer?
```

Like `splay-tree-remove-range!`, but also decreases the value of all keys greater than or equal to *to* by `(- to from)`.

This operation is only allowed on adjustable splay trees, and it takes *O(log N)* time.

```
(splay-tree-expand! s from to) → void?
  s : adjustable-splay-tree?
  from : exact-integer?
  to : exact-integer?
```

Increases the value of all keys greater than or equal to *from* by (- *to* *from*).

This operation is only allowed on adjustable splay trees, and it takes *O(log N)* time.

```
(splay-tree-iterate-least s) → (or/c #f splay-tree-iter?)
  s : splay-tree?
(splay-tree-iterate-greatest s) → (or/c #f splay-tree-iter?)
  s : splay-tree?
(splay-tree-iterate-least/>? s key) → (or/c #f splay-tree-iter?)
  s : splay-tree?
  key : any/c
(splay-tree-iterate-least/>=? s key)
 → (or/c #f splay-tree-iter?)
  s : splay-tree?
  key : any/c
(splay-tree-iterate-greatest/<? s key)
 → (or/c #f splay-tree-iter?)
  s : splay-tree?
  key : any/c
(splay-tree-iterate-greatest/<=? s key)
 → (or/c #f splay-tree-iter?)
  s : splay-tree?
  key : any/c
```

Implementations of dict-iterate-least, dict-iterate-greatest, dict-iterate-least/>?, dict-iterate-least/>=?, dict-iterate-greatest/<?, and dict-iterate-greatest/<=?, respectively.

```
(splay-tree-iter? x) → boolean?
  x : any/c
```

Returns #t if *x* represents a position in a splay-tree, #f otherwise.

```
(splay-tree->list s) → (listof pair?)
  s : splay-tree?
```

Returns an association list with the keys and values of *s*, in order.

5 Skip Lists

```
(require data/skip-list)
```
package: data-lib

Skip-lists are a simple, efficient data structure for mutable dictionaries with totally ordered keys. They were described in the paper "Skip Lists: A Probabilistic Alternative to Balanced Trees" by William Pugh in Communications of the ACM, June 1990, 33(6) pp668-676.

A skip-list is an ordered dictionary (`dict?` and `ordered-dict?`). It also supports extensions of the dictionary interface for iterator-based search and mutation.

Operations on skip-lists are not thread-safe. If a key in a skip-list is mutated, the skip-list's internal invariants may be violated, causings its behavior to become unpredictable.

Time complexities are given for many of the operations below. With a few exceptions, the time complexities below are probabilistic and assume that key comparison is constant-time. *N* refers to the number of elements in the skip-list.

```
(make-skip-list [ord
                 #:key-contract key-contract
                 #:value-contract value-contract]) → skip-list?
  ord : order? = datum-order
  key-contract : contract? = any/c
  value-contract : contract? = any/c
```

Makes a new empty skip-list. The skip-list uses *ord* to order keys; in addition, the domain contract of *ord* is combined with *key-contract* to check keys.

Examples:

```
> (define skip-list (make-skip-list real-order))
> (skip-list-set! skip-list 3 'apple)
> (skip-list-set! skip-list 6 'cherry)
> (dict-map skip-list list)
'((3 apple) (6 cherry))
> (skip-list-ref skip-list 3)
'apple
> (skip-list-remove! skip-list 6)
> (skip-list-count skip-list)
1
```

```
(make-adjustable-skip-list [#:key-contract key-contract
                            #:value-contract value-contract])
 → adjustable-skip-list?
  key-contract : contract? = any/c
  value-contract : contract? = any/c
```

Makes a new empty skip-list that permits only exact integers as keys (in addition to any constraints imposed by *key-contract*). The resulting skip-list answers true to `adjustable-skip-list?` and supports efficient key adjustment; see `skip-list-contract!` and `skip-list-expand!`.

```
(skip-list? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a skip-list, `#f` otherwise.

```
(adjustable-skip-list? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a skip-list that supports key adjustment; see `skip-list-contract!` and `skip-list-expand!`.

```
(skip-list-ref skip-list key [default]) → any/c
  skip-list : skip-list?
  key : any/c
  default : any/c = (lambda () (error ....))
(skip-list-set! skip-list key value) → void?
  skip-list : skip-list?
  key : any/c
  value : any/c
(skip-list-remove! skip-list key) → void?
  skip-list : skip-list?
  key : any/c
(skip-list-count skip-list) → exact-nonnegative-integer?
  skip-list : skip-list?
```

Implementations of `dict-ref`, `dict-set!`, `dict-remove!`, and `dict-count`, respectively.

The `skip-list-ref`, `skip-list-set!`, and `skip-list-remove!` operations take *O(log N)* time. The `skip-list-count` operation takes constant time.

```
(skip-list-remove-range! skip-list from to) → void?
  skip-list : skip-list?
  from : any/c
  to : any/c
```

Removes all keys in [*from*, *to*); that is, all keys greater than or equal to *from* and less than *to*.

This operation takes probabilistic *O(log N)* time.

```
(skip-list-contract! skip-list from to) → void?
  skip-list : adjustable-skip-list?
  from : exact-integer?
  to : exact-integer?
```

Like `skip-list-remove-range!`, but also decreases the value of all keys greater than or equal to *to* by `(- to from)`.

This operation takes probabilistic *O(log N)* time.

```
(skip-list-expand! skip-list from to) → void?
  skip-list : adjustable-skip-list?
  from : exact-integer?
  to : exact-integer?
```

Increases each key greater than or equal to *from* by `(- to from)`.

This operation takes probabilistic *O(log N)* time.

```
(skip-list-iterate-first skip-list) → (or/c skip-list-iter? #f)
  skip-list : skip-list?
(skip-list-iterate-next skip-list iter)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  iter : skip-list-iter?
(skip-list-iterate-key skip-list iter) → any/c
  skip-list : skip-list?
  iter : skip-list-iter?
(skip-list-iterate-value skip-list iter) → any/c
  skip-list : skip-list?
  iter : skip-list-iter?
```

Implementations of `dict-iterate-first`, `dict-iterate-next`, `dict-iterate-key`, and `dict-iterate-value`, respectively.

A skip-list iterator is invalidated if the entry it points to is deleted from the skip-list (even if another entry is later inserted with the same key). The `skip-list-iterate-key` and `skip-list-iterate-value` operations raise an exception when called on an invalidated iterator, but `skip-list-iterate-next` advances to the next undeleted entry that was visible to *iter* when it was valid.

These operations take constant time.

```
(skip-list-iterate-least/>? skip-list key)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  key : any/c
```

```
(skip-list-iterate-least/>=? skip-list key)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  key : any/c
(skip-list-iterate-greatest/<? skip-list
                               key)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  key : any/c
(skip-list-iterate-greatest/<=? skip-list
                                key)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  key : any/c
(skip-list-iterate-least skip-list) → (or/c skip-list-iter? #f)
  skip-list : skip-list?
(skip-list-iterate-greatest skip-list)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
```

Implementations of `dict-iterate-least/>?`, `dict-iterate-least/>=?`, `dict-iterate-greatest/<?`, `dict-iterate-greatest/<=?`, `dict-iterate-least`, and `dict-iterate-greatest`, respectively.

See notes on iterators at `skip-list-iterate-first`.

The `skip-list-iterate-least` operation takes constant time; the rest take *O(log N)* time.

```
(skip-list-iter? v) → boolean?
  v : any/c
```

Returns `#t` if `v` represents a position in a skip-list, `#f` otherwise.

```
(skip-list-iter-valid? iter) → boolean?
  iter : skip-list-iter?
```

Returns `#t` if the iterator is valid, or `#f` if invalidated by deletion; see `skip-list-iterate-first` for details about invalidation.

```
(skip-list->list skip-list) → (listof pair?)
  skip-list : skip-list?
```

Returns an association list with the keys and values of *skip-list*, in order.

This operation takes *O(N)* time, where *N* is the number of entries in the skip-list.

6 Interval Maps

```
(require data/interval-map)
```
package: data-lib

An interval-map is a mutable data structure that maps *half-open* intervals of exact integers to values. An interval-map is queried at a discrete point, and the result of the query is the value mapped to the interval containing the point.

Internally, interval-maps use a skip-list (`data/skip-list`) of intervals for efficient query and update, including efficient contraction and expansion of intervals.

Interval-maps implement the dictionary (`racket/dict`) interface to a limited extent. Only `dict-ref` and the iteration-based methods (`dict-iterate-first`, `dict-map`, etc) are supported. For the iteration-based methods, the mapping's keys are considered the pairs of the start and end positions of the mapping's (half-open) intervals.

Examples:

```
> (define r (make-interval-map))
> (interval-map-set! r 1 5 'apple)
> (interval-map-set! r 6 10 'pear)
> (interval-map-set! r 3 7 'banana)
> r
(make-interval-map '(((1 . 3) . apple) ((3 . 7) . banana) ((7 .
10) . pear)))
> (interval-map-ref r 1 #f)
'apple
> (interval-map-ref r 3 #f)
'banana
> (interval-map-ref r 10 #f)
#f
```

Operations on interval-maps are not thread-safe.

```
(make-interval-map [contents
                    #:key-contract key-contract
                    #:value-contract value-contract])
 → interval-map?
  contents : (listof (cons/c (cons/c exact-integer? exact-integer?) any/c))
           = null
  key-contract : contract? = any/c
  value-contract : contract? = any/c
```

Makes a new interval-map initialized with *contents*, which has the form

```
(list (cons (cons start end) value) ...)
```

Examples:

```
> (define r (make-interval-map '(((0 . 5) . apple) ((5 . 10) .
banana))))
> (interval-map-ref r 2)
'apple
> (interval-map-ref r 5)
'banana
```

```
(interval-map? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an interval-map, `#f` otherwise.

```
(interval-map-ref  interval-map
                   position
                  [default])     → any/c
  interval-map : interval-map?
  position : exact-integer?
  default : any/c = (lambda () (error ....))
```

Return the value associated with *position* in *interval-map*. If no mapping is found, *default* is applied if it is a procedure, or returned otherwise.

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

```
(interval-map-ref/bounds  interval-map
                          position
                         [default])
 → (or/c #f exact-integer?)
   (or/c #f exact-integer?)
   any/c
  interval-map : interval-map?
  position : exact-integer?
  default : any/c = (lambda () (error ....))
```

Like `interval-map-ref`, but also returns the bounds of the interval associated with *position*. If no mapping is found and *default* is a procedure, it is applied. If no mapping is found and *default* is not a procedure, `#f` is returned for the start and end positions and *default* is returned as the value.

```
(interval-map-set! interval-map
                   start
                   end
                   value)        → void?
```

```
  interval-map : interval-map?
  start : exact-integer?
  end : exact-integer?
  value : any/c
```

Updates *interval-map*, associating every position in [*start*, *end*) with *value*.

Existing interval mappings contained in [*start*, *end*) are destroyed, and partly overlapping intervals are truncated. See **interval-map-update*!** for an updating procedure that preserves distinctions within [*start*, *end*).

```
(interval-map-update*!  interval-map
                        start
                        end
                        updater
                       [default])     → void?
  interval-map : interval-map?
  start : exact-integer?
  end : exact-integer?
  updater : (-> any/c any/c)
  default : any/c = (lambda () (error ....))
```

Updates *interval-map*, associating every position in [*start*, *end*) with the result of applying *updater* to the position's previously associated value, or to the default value produced by *default* if no mapping exists.

Unlike **interval-map-set!**, **interval-map-update*!** preserves existing distinctions within [*start*, *end*).

```
(interval-map-remove! interval-map
                      start
                      end)          → void?
  interval-map : interval-map?
  start : (or/c exact-integer? -inf.0)
  end : (or/c exact-integer? +inf.0)
```

Removes the value associated with every position in [*start*, *end*).

```
(interval-map-contract! interval-map
                        start
                        end)          → void?
  interval-map : interval-map?
  start : exact-integer?
  end : exact-integer?
```

Contracts *interval-map*'s domain by removing all mappings on the interval [*start*, *end*) and decreasing intervals initally after *end* by `(- end start)`.

If *start* is not less than *end*, an exception is raised.

```
(interval-map-expand! interval-map
                      start
                      end)          → void?
  interval-map : interval-map?
  start : exact-integer?
  end : exact-integer?
```

Expands *interval-map*'s domain by introducing a gap [*start*, *end*) and increasing intervals starting at or after *start* by `(- end start)`.

If *start* is not less than *end*, an exception is raised.

```
(interval-map-cons*! interval-map
                     start
                     end
                     v
                    [default])     → void?
  interval-map : interval-map?
  start : any/c
  end : any/c
  v : any/c
  default : any/c = null
```

Same as the following:

```
(interval-map-update*! interval-map start end
                       (lambda (old) (cons v old))
                       default)
```

```
(interval-map-iterate-first interval-map)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
(interval-map-iterate-next interval-map
                           iter)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
  iter : interval-map-iter?
(interval-map-iterate-key interval-map
                          iter)        → pair?
  interval-map : interval-map?
  iter : interval-map-iter?
(interval-map-iterate-value interval-map
                            iter)        → any
  interval-map : interval-map?
  iter : interval-map-iter?
```

Implementations of dict-iterate-first, dict-iterate-next, dict-iterate-key, and dict-iterate-value, respectively.

```
(interval-map-iterate-least interval-map)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
(interval-map-iterate-greatest interval-map)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
```

Like dict-iterate-least and dict-iterate-greatest, respectively, though interval maps do not implement the gen:ordered-dict interface.

Added in version 1.2 of package data-lib.

```
(interval-map-iterate-least/start>? interval-map
                                    start)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
  start : exact-integer?
(interval-map-iterate-least/start>=? interval-map
                                     start)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
  start : exact-integer?
(interval-map-iterate-greatest/start<? interval-map
                                       start)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
  start : exact-integer?
(interval-map-iterate-greatest/start<=? interval-map
                                        start)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
  start : exact-integer?
(interval-map-iterate-least/end>? interval-map
                                  end)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
  end : exact-integer?
(interval-map-iterate-least/end>=? interval-map
                                   end)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
  end : exact-integer?
```

```
(interval-map-iterate-greatest/end<? interval-map
                                     end)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
  end : exact-integer?
(interval-map-iterate-greatest/end<=? interval-map
                                      end)
 → (or/c interval-map-iter? #f)
  interval-map : interval-map?
  end : exact-integer?
```

Like `dict-iterate-least/>?`, `dict-iterate-least/>=?`, `dict-iterate-greatest/<?`, and `dict-iterate-greatest/<=?`, but each function comes in a `start` and an `end` variant corresponding to the start or end of each interval, respectively.

Note that interval maps do not implement the `gen:ordered-dict` interface, as these operations accept individual bounds rather than the pairs returned by `interval-map-iterate-key`. Therefore, these operations must be used directly.

Added in version 1.2 of package `data-lib`.

```
(interval-map-iter? v) → boolean?
  v : any/c
```

Returns `#t` if *v* represents a position in an interval-map, `#f` otherwise.

7 Binary Heaps

```
(require data/heap)
```
package: data-lib

Binary heaps are a simple implementation of priority queues. For a heap of n elements, `heap-add!` and `heap-remove-min!` take O(log n) time per added or removed element, while `heap-min` and `heap-count` take constant time; `heap-remove!` takes O(n) time, and `heap-remove-eq!` takes O(log n) time on average; `heap-sort!` takes O(n log n) time.

Operations on binary heaps are not thread-safe.

All functions are also provided by `data/heap/unsafe` without contracts.

```
(make-heap <=?) → heap?
  <=? : (-> any/c any/c any/c)
```

Makes a new empty heap using `<=?` to order elements.

Examples:

```
> (define a-heap-of-strings (make-heap string<=?))
> a-heap-of-strings
#<heap>
; With structs:
> (struct node (name val))
> (define (node<=? x y)
    (<= (node-val x) (node-val y)))
> (define a-heap-of-nodes (make-heap node<=?))
> a-heap-of-nodes
#<heap>
```

```
(heap? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a heap, `#f` otherwise.

Examples:

```
> (heap? (make-heap <=))
#t
> (heap? "I am not a heap")
#f
```

```
(heap-count h) → exact-nonnegative-integer?
  h : heap?
```

Returns the number of elements in the heap.

Examples:

```
> (define a-heap (make-heap <=))
> (heap-add-all! a-heap '(7 3 9 1 13 21 15 31))
> (heap-count a-heap)
8
```

```
(heap-add! h v ...) → void?
  h : heap?
  v : any/c
```

Adds each `v` to the heap.

Examples:

```
> (define a-heap (make-heap <=))
> (heap-add! a-heap 2009 1009)
```

```
(heap-add-all! h v) → void?
  h : heap?
  v : (or/c list? vector? heap?)
```

Adds each element contained in `v` to the heap, leaving `v` unchanged.

Examples:

```
> (define heap-1 (make-heap <=))
> (define heap-2 (make-heap <=))
> (define heap-12 (make-heap <=))
> (heap-add-all! heap-1 '(3 1 4 1 5 9 2 6))
> (heap-add-all! heap-2 #(2 7 1 8 2 8 1 8))
> (heap-add-all! heap-12 heap-1)
> (heap-add-all! heap-12 heap-2)
> (heap-count heap-12)
16
```

```
(heap-min h) → any/c
  h : heap?
```

Returns the least element in the heap `h`, according to the heap's ordering. If the heap is empty, an exception is raised.

Examples:

```
> (define a-heap (make-heap string<=?))
> (heap-add! a-heap "sneezy" "sleepy" "dopey" "doc"
             "happy" "bashful" "grumpy")
> (heap-min a-heap)
"bashful"
; Taking the min of the empty heap is an error:
> (heap-min (make-heap <=))
heap-min: empty heap
```

```
(heap-remove-min! h) → void?
  h : heap?
```

Removes the least element in the heap *h*. If the heap is empty, an exception is raised.

Examples:

```
> (define a-heap (make-heap string<=?))
> (heap-add! a-heap "fili" "fili" "oin" "gloin" "thorin"
                    "dwalin" "balin" "bifur" "bofur"
                    "bombur" "dori" "nori" "ori")
> (heap-min a-heap)
"balin"
> (heap-remove-min! a-heap)
> (heap-min a-heap)
"bifur"
```

```
(heap-remove! h v [#:same? same?]) → boolean?
  h : heap?
  v : any/c
  same? : (-> any/c any/c any/c) = equal?
```

Removes *v* from the heap *h* if it exists, and returns `#t` if the removal was successful, `#f` otherwise. This operation takes O(n) time—see also `heap-remove-eq!`.

Examples:

```
> (define a-heap (make-heap string<=?))
> (heap-add! a-heap "a" "b" "c")
> (heap-remove! a-heap "b")
#t
> (for/list ([a (in-heap a-heap)]) a)
'("a" "c")
```

Changed in version 7.6.0.18 of package `data-lib`: Returns a `boolean?` instead of `void?`

```
(heap-remove-eq! h v) → boolean?
  h : heap?
  v : any/c
```

Removes `v` from the heap `h` if it exists according to `eq?`, and returns `#t` if the removal was successful, `#f` otherwise. This operation takes O(log n) time, plus the indexing cost (which is O(1) on average, but O(n) in the worst case). The heap must not contain duplicate elements according to `eq?`, otherwise it may not be possible to remove all duplicates (see the example below).

Examples:

```
> (define h (make-heap string<=?))
> (define elt1 "123")
> (define elt2 "abcxyz")
> (heap-add! h elt1 elt2)
; The element is not found because no element of the heap is `eq?`
; to the provided value:
> (heap-remove-eq! h (string-append "abc" "xyz"))
#f
> (heap->vector h)
'#("123" "abcxyz")
; But this succeeds:
> (heap-remove-eq! h elt2)
#t
> (heap->vector h)
'#("123")
; Removing duplicate elements (according to `eq?`) may fail:
> (heap-add! h elt2 elt2)
> (heap->vector h)
'#("123" "abcxyz" "abcxyz")
> (heap-remove-eq! h elt2)
#t
> (heap-remove-eq! h elt2)
#f
> (heap->vector h)
'#("123" "abcxyz")
; But we can resort to the more general `heap-remove!`:
> (heap-remove! h elt2 #:same? string=?)
#t
> (heap->vector h)
'#("123")
```

Added in version 7.8.0.5 of package `data-lib`.

```
(vector->heap <=? items) → heap?
  <=? : (-> any/c any/c any/c)
  items : vector?
```

Builds a heap with the elements from *items*. The vector is not modified.

Examples:

```
> (struct item (val frequency))
> (define (item<=? x y)
    (<= (item-frequency x) (item-frequency y)))
> (define some-sample-items
    (vector (item #\a 17) (item #\b 12) (item #\c 19)))
> (define a-heap (vector->heap item<=? some-sample-items))
```

```
(heap->vector h) → vector?
  h : heap?
```

Returns a vector containing the elements of heap *h* in the heap's order. The heap is not modified.

Examples:

```
> (define word-heap (make-heap string<=?))
> (heap-add! word-heap "pile" "mound" "agglomerate" "cumulation")
> (heap->vector word-heap)
'#("agglomerate" "cumulation" "mound" "pile")
```

```
(heap-copy h) → heap?
  h : heap?
```

Makes a copy of heap *h*.

Examples:

```
> (define word-heap (make-heap string<=?))
> (heap-add! word-heap "pile" "mound" "agglomerate" "cumulation")
> (define a-copy (heap-copy word-heap))
> (heap-remove-min! a-copy)
> (heap-count word-heap)
4
> (heap-count a-copy)
3
```

```
(in-heap/consume! heap) → sequence?
  heap : heap?
```

Returns a sequence equivalent to *heap*, maintaining the heap's ordering. The heap is consumed in the process. Equivalent to repeated calling `heap-min`, then `heap-remove-min!`.

Examples:

```
> (define h (make-heap <=))
> (heap-add-all! h '(50 40 10 20 30))
> (for ([x (in-heap/consume! h)])
    (displayln x))
10
20
30
40
50
> (heap-count h)
0
```

```
(in-heap heap) → sequence?
  heap : heap?
```

Returns a sequence equivalent to *heap*, maintaining the heap's ordering. Equivalent to `in-heap/consume!` except the heap is copied first.

Examples:

```
> (define h (make-heap <=))
> (heap-add-all! h '(50 40 10 20 30))
> (for ([x (in-heap h)])
    (displayln x))
10
20
30
40
50
> (heap-count h)
5
```

```
(heap-sort! v <=?) → void?
  v : (and/c vector? (not/c immutable?))
  <=? : (-> any/c any/c any/c)
```

Sorts vector `v` using the comparison function `<=?`.

Examples:

```
> (define terms (vector "flock" "hatful" "deal" "batch" "lot" "good
deal"))
> (heap-sort! terms string<=?)
> terms
'#("batch" "deal" "flock" "good deal" "hatful" "lot")
```

8 Integer Sets

```
(require data/integer-set)
```
package: base

This library provides functions for working with finite sets of integers. This module is designed for sets that are compactly represented as groups of intervals, even when their cardinality is large. For example, the set of integers from -1000000 to 1000000 except for 0, can be represented as {[-1000000, -1], [1, 1000000]}. This data structure would not be a good choice for the set of all odd integers between 0 and 1000000, which would be {[1, 1], [3, 3], ... [999999, 999999]}.

In addition to the *integer set* abstract type, a *well-formed set* is a list of pairs of exact integers, where each pair represents a closed range of integers, and the entire set is the union of the ranges. The ranges must be disjoint and increasing. Further, adjacent ranges must have at least one integer between them. For example: `'((-1 . 2) (4 . 10))` is a well-formed-set as is `'((1 . 1) (3 . 3))`, but `'((1 . 5) (6 . 7))`, `'((1 . 5) (-3 . -1))`, `'((5 . 1))`, and `'((1 . 5) (3 . 6))` are not.

An integer set implements the stream and sequence generic interfaces.

Example:

```
> (for/list ([i (make-integer-set '((2 . 3)
                                    (5 . 6)
                                    (10 . 15)))])
    i)
'(2 3 5 6 10 11 12 13 14 15)
```

```
(make-integer-set wfs) → integer-set?
  wfs : well-formed-set?
```

Creates an integer set from a well-formed set.

```
(integer-set-contents s) → well-formed-set?
  s : integer-set?
```

Produces a well-formed set from an integer set.

```
(set-integer-set-contents! s wfs) → void?
  s : integer-set?
  wfs : well-formed-set?
```

Mutates an integer set.

```
(integer-set? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an integer set, `#f` otherwise.

```
(well-formed-set? v) → boolean?
  v : any/c
```

Recognizes `(listof (cons/c exact-integer? exact-integer?))`, where the result of `(flatten v)` is sorted by `<=`, the elements of the pairs in the list are distinct (and thus strictly increasing), and the second element in a pair is at least one less than the first element of the subsequent pair.

Examples:

```
> (well-formed-set? '((-1 . 2) (4 . 10)))
#t
> (well-formed-set? '((1 . 1) (3 . 3)))
#t
> (well-formed-set? '((1 . 5) (6 . 7)))
#f
> (well-formed-set? '((1 . 5) (-3 . -1)))
#f
> (well-formed-set? '((5 . 1)))
#f
> (well-formed-set? '((1 . 5) (3 . 6)))
#f
```

```
(make-range) → integer-set?
(make-range elem) → integer-set?
  elem : exact-integer?
(make-range start end) → integer-set?
  start : exact-integer?
  end : exact-integer?
```

Produces, respectively, an empty integer set, an integer set containing only *elem*, or an integer set containing the integers from *start* to *end* inclusive, where `(<= start end)`.

```
(intersect x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Returns the intersection of the given sets.

```
(subtract x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Returns the difference of the given sets (i.e., elements in *x* that are not in *y*).

```
(union x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Returns the union of the given sets.

```
(split x y) → integer-set? integer-set? integer-set?
  x : integer-set?
  y : integer-set?
```

Produces three values: the first is the intersection of *x* and *y*, the second is the difference *x* remove *y*, and the third is the difference *y* remove *x*.

```
(complement s start end) → integer-set?
  s : integer-set?
  start : exact-integer?
  end : exact-integer?
```

Returns a set containing the elements between *start* to *end* inclusive that are not in *s*, where `(<= start-k end-k)`.

```
(symmetric-difference x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Returns an integer set containing every member of *x* and *y* that is not in both sets.

```
(member? k s) → boolean?
  k : exact-integer?
  s : integer-set?
```

Returns `#t` if *k* is in *s*, `#f` otherwise.

```
(get-integer set) → (or/c exact-integer? #f)
  set : integer-set?
```

Returns a member of *set*, or `#f` if *set* is empty.

```
(foldr proc base-v s) → any/c
  proc : (exact-integer? any/c . -> . any/c)
  base-v : any/c
  s : integer-set?
```

Applies *proc* to each member of *s* in ascending order, where the first argument to *proc* is the set member, and the second argument is the fold result starting with *base-v*. For example, `(foldr cons null s)` returns a list of all the integers in *s*, sorted in increasing order.

```
(partition s) → (listof integer-set?)
  s : (listof integer-set?)
```

Returns the coarsest refinement of the sets in *s* such that the sets in the result list are pairwise disjoint. For example, partitioning the sets that represent `'((1 . 2) (5 . 10))` and `'((2 . 2) (6 . 6) (12 . 12))` produces the a list containing the sets for `'((1 . 1) (5 . 5) (7 . 10))` `'((2 . 2) (6 . 6))`, and `'((12 . 12))`.

```
(count s) → exact-nonnegative-integer?
  s : integer-set?
```

Returns the number of integers in the given integer set.

```
(subset? x y) → boolean?
  x : integer-set?
  y : integer-set?
```

Returns true if every integer in *x* is also in *y*, otherwise `#f`.

9 Bit Vectors

```
(require data/bit-vector)
```
package: base

A *bit vector* is a mutable sequence whose elements are booleans. A bit vector also acts as a dictionary (`dict?` from `racket/dict`), where the keys are zero-based indexes and the values are the elements of the bit-vector. A bit-vector has a fixed size.

Two bit-vectors are `equal?` if they contain the same number of elements and if they contain equal elements at each index.

```
(make-bit-vector size [fill]) → bit-vector?
  size : exact-integer?
  fill : boolean? = #f
```

Creates a new bit-vector of size *size*. All elements are initialized to *fill*.

Examples:

```
> (bit-vector-ref (make-bit-vector 3) 2)
#f
> (bit-vector-ref (make-bit-vector 3 #t) 2)
#t
```

```
(bit-vector elem ...) → bit-vector?
  elem : boolean?
```

Creates a new bit-vector containing each *elem* in order.

Example:

```
> (bit-vector-ref (bit-vector #f #t #f) 1)
#t
```

```
(bit-vector? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a bit-vector, `#f` otherwise.

```
(bit-vector-ref bv index [default]) → any/c
  bv : bit-vector?
  index : exact-nonnegative-integer?
  default : any/c = (error ....)
```

Returns the element at index *index*, if *index* is less than (bit-vector-length *bv*). Otherwise, *default* is invoked if it is a procedure, returned otherwise.

Examples:

```
> (bit-vector-ref (bit-vector #f #t) 1)
#t
> (bit-vector-ref (bit-vector #f #t) 5 'not-there)
'not-there
```

```
(bit-vector-set! bv index value) → void?
  bv : bit-vector?
  index : (and/c exact-nonnegative-integer?
                 (</c (+ 1 (bit-vector-length bv))))
  value : boolean?
```

Sets the value at index *index* to be *value*.

Examples:

```
> (define bv (bit-vector #f #t))
> (bit-vector-ref bv 0)
#f
> (bit-vector-set! bv 0 #t)
> (bit-vector-ref bv 0)
#t
```

```
(bit-vector-length bv) → exact-nonnegative-integer?
  bv : bit-vector?
```

Returns the number of items in the bit-vector *bv*.

```
(bit-vector-popcount bv) → exact-nonnegative-integer?
  bv : bit-vector?
```

Returns the number of set bits in the bit-vector *bv*.

Example:

```
> (bit-vector-popcount (bit-vector #f #t #t))
2
```

```
(bit-vector-copy bv [start end]) → bit-vector?
  bv : bit-vector?
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (bit-vector-length bv)
```

Creates a fresh bit-vector with the same elements as *bv* from *start* (inclusive) to *end* (exclusive).

```
(in-bit-vector bv) → sequence?
  bv : bit-vector?
```

Returns a sequence whose elements are the elements of the bit-vector *bv*. Mutation of *bv* while the sequence is running changes the elements produced by the sequence. To obtain a sequence from a snapshot of *bv*, use `(in-bit-vector (bit-vector-copy bv))` instead.

Examples:

```
> (define bv (bit-vector #f #t #f))
> (for/list ([x (in-bit-vector bv)]) x)
'(#f #t #f)
```

```
(for/bit-vector maybe-length (for-clause ...)
  body-or-break ... body)

maybe-length =
             | #:length length-expr
             | #:length length-expr #:fill fill-expr

  length-expr : exact-nonnegative-integer?
```

Iterates like `for/vector`, but results are accumulated into a bit-vector instead of a vector.

If the optional `#:length` clause is specified, the result of *length-expr* determines the length of the result bit-vector. In that case, the iteration can be performed more efficiently, and it terminates when the bit-vector is full or the requested number of iterations have been performed, whichever comes first. If *length-expr* specifies a length longer than the number of iterations, then the remaining slots of the vector are initialized to the value of *fill-expr*, which defaults to `#f` (i.e., the default argument of `make-bit-vector`).

Examples:

```
> (bit-vector->list
   (for/bit-vector ([i '(1 2 3)]) (odd? i)))
'(#t #f #t)
> (bit-vector->list
   (for/bit-vector #:length 2 ([i '(1 2 3)]) (odd? i)))
'(#t #f)
> (bit-vector->list
   (for/bit-vector #:length 4 ([i '(1 2 3)]) (odd? i)))
'(#t #f #t #f)
```

```
> (bit-vector->list
   (for/bit-vector #:length 4 #:fill #t ([i '(1 2 3)]) (odd? i)))
'(#t #f #t #t)
```

The `for/bit-vector` form may allocate a bit-vector and mutate it after each iteration of *body*, which means that capturing a continuation during *body* and applying it multiple times may mutate a shared bit-vector.

```
(for*/bit-vector maybe-length (for-clause ...)
  body-or-break ... body)
```

Like `for/bit-vector` but with the implicit nesting of `for*`.

```
(bit-vector->list bv) → (listof boolean?)
  bv : bit-vector?
(list->bit-vector bits) → bit-vector?
  bits : (listof boolean?)
(bit-vector->string bv) → (and/c string? #rx"^[01]*$")
  bv : bit-vector?
(string->bit-vector s) → bit-vector?
  s : (and/c string? #rx"^[01]*$")
```

Converts between bit-vectors and their representations as lists and strings.

Examples:

```
> (bit-vector->list (string->bit-vector "100111"))
'(#t #f #f #t #t #t)
> (bit-vector->string (list->bit-vector '(#t #f #t #t)))
"1011"
```

10 Union-Find: Sets with only Canonical Elements

```
(require data/union-find)
```
package: data-lib

The union-find algorithm and data structure provides an API for representing sets that contain a single, canonical element. The sets support an (imperative) union operation (the library picks one of the canonical elements of the original sets to be the canonical element of the union), as well as getting and setting the canonical element.

These operations are not thread-safe.

```
(uf-new c) → uf-set?
  c : any/c
```

Makes a new set with the canonical element `c`.

This is a constant time operation.

Examples:

```
> (uf-new 'whale)
#<uf-set: 'whale>
> (uf-new 'dwarf-lantern)
#<uf-set: 'dwarf-lantern>
```

```
(uf-set? x) → boolean?
  x : any/c
```

Returns `#t` if `x` was created with `uf-new`, and `#f` otherwise.

This is a constant time operation.

Examples:

```
> (uf-set? (uf-new 'spiny-dogfish))
#t
> (uf-set? "I am not a uf-set")
#f
```

```
(uf-find a) → any/c
  a : uf-set?
```

Returns the canonical element of `a`.

This is an amortized (essentially) constant time operation.

Example:

```
> (uf-find (uf-new 'tasselled-wobbegong))
'tasselled-wobbegong
```

```
(uf-union! a b) → void?
  a : uf-set?
  b : uf-set?
```

Imperatively unifies `a` and `b`, making them both have the same canonical element. Either of `a` or `b`'s canonical elements may become the canonical element for the union.

This is an amortized (essentially) constant time operation.

Examples:

```
> (define a (uf-new 'sand-devil))
> (define b (uf-new 'pigeye))
> (uf-union! a b)
> (uf-find a)
'sand-devil
> (uf-find b)
'sand-devil
```

```
(uf-same-set? a b) → boolean?
  a : uf-set?
  b : uf-set?
```

Returns `#t` if the sets `a` and `b` have been unioned.

This is an amortized (essentially) constant time operation.

Examples:

```
> (define a (uf-new 'finetooth))
> (define b (uf-new 'speartooth))
> (uf-same-set? a b)
#f
> (uf-union! a b)
> (uf-same-set? a b)
#t
```

```
(uf-set-canonical! a c) → void?
  a : uf-set?
  c : any/c
```

Changes `a` to have a new canonical element.

This is an amortized (essentially) constant time operation.

Examples:

```
> (define a (uf-new 'sand-devil))
> (uf-set-canonical! a 'lemon)
> (uf-find a)
'lemon
> (define b (uf-new 'pigeye))
> (uf-union! a b)
> (uf-set-canonical! b 'sicklefin-lemon)
> (uf-find a)
'sicklefin-lemon
```

11 Enumerations

```
(require data/enumerate/lib)
```
package: data-enumerate-lib

This library defines *enumerations*. Enumerations are represented as bijections between the natural numbers (or a prefix of them) and the values of some contract. Most of the enumerations defined in this library guarantee that the constructed bijection is efficient, in the sense that decoding a number is roughly linear in the number of bits taken to represent the number.

The two main options on an enumeration convert natural numbers back (`from-nat`) and forth (`to-nat`) between the elements of the contract. The simplest enumeration, `natural/e` is just a pair of identity functions:

```
> (from-nat natural/e 0)
0
> (to-nat natural/e 1)
1
```

but the library builds up more complex enumerations from simple ones. For example, you can enumerate lists of the elements of other enumerations using `list/e`:

```
> (from-nat (list/e natural/e natural/e natural/e) 0)
'(0 0 0)
> (from-nat (list/e natural/e natural/e natural/e) 1)
'(0 0 1)
> (from-nat (list/e natural/e natural/e natural/e) (expt 2 100))
'(10822639409 2238661967 110761420)
> (to-nat (list/e natural/e natural/e natural/e)
          (list 123456789 123456789 123456789))
188167641751389148183899 9
```

To interleave two enumerations, use `or/e`:

```
> (from-nat (or/e natural/e (list/e natural/e natural/e)) 0)
0
> (from-nat (or/e natural/e (list/e natural/e natural/e)) 1)
'(0 0)
> (from-nat (or/e natural/e (list/e natural/e natural/e)) 2)
1
> (from-nat (or/e natural/e (list/e natural/e natural/e)) 3)
'(0 1)
> (from-nat (or/e natural/e (list/e natural/e natural/e)) 4)
2
```

and to construct recursive data structures, use `delay/e` (with a little help from `single/e` to build a singleton enumeration for the base-case):

```
(define bt/e
  (delay/e
   (or/e (single/e #f)
         (list/e bt/e bt/e))))

> (from-nat bt/e 0)
#f
> (from-nat bt/e 1)
'(#f #f)
> (from-nat bt/e 2)
'(#f (#f #f))
> (from-nat bt/e 3)
'((#f #f) #f)
> (from-nat bt/e (expt 2 100))
'(((((((#f #f) #f) (#f (#f #f))) (((#f (#f #f)) (#f #f)) ((#f #f)
#f)))
    (((#f #f) (#f #f)) (#f ((#f #f) (#f (#f #f))))))
   (((((#f #f) #f) (#f (#f #f))) (((#f (#f #f)) (#f #f)) ((#f #f)
#f)))
    (((#f #f) #f) (#f ((#f #f) (#f (#f #f)))))))
  ((((((#f #f) #f) (#f (#f #f))) (((#f (#f #f)) (#f #f)) ((#f #f)
#f)))
    (((#f #f) (#f #f)) (#f ((#f #f) (#f (#f #f))))))
   (((((#f #f) #f) (#f (#f #f))) (((#f (#f #f)) (#f #f)) ((#f #f)
#f)))
    (((#f #f) #f) (#f ((#f #f) (#f (#f #f)))))))
```

The library also supports dependent enumerations. For example, to build ordered pairs, we can allow any natural number in the first component, but we want to have only natural numbers that are larger than that in the second component. The `cons/de` lets us express the dependency (using a notation similar to the contract `cons/dc`) and then we can use `nat+/e`, which builds a enumerators of natural numbers that are larger than or equal to its argument:

```
(define ordered-pair/e
  (cons/de [hd natural/e]
           [tl (hd) (nat+/e (+ hd 1))]))

> (for/list ([i (in-range 10)])
    (from-nat ordered-pair/e i))
'((0 . 1)
  (0 . 2)
  (1 . 2)
  (1 . 3)
  (0 . 3)
  (1 . 4)
  (2 . 3)
```

```
  (2 . 4)
  (2 . 5)
  (0 . 4))
```

Sometimes the best way to get a new enumeration is to adjust the output of a previous one. For example, if we wanted ordered pairs that were ordered in the other direction, we just need to swap the components of the pair from the `ordered-pair/e` enumeration. The function `map/e` adjusts existing enumerations. It accepts a contract describing the new kinds of values and functions that convert back and forth between the new and old kinds of values.

```
(define (swap-components x) (cons (cdr x) (car x)))
(define other-ordered-pair/e
  (map/e swap-components
         swap-components
         ordered-pair/e
         #:contract (cons/c natural?
                            natural?)))
```

```
> (for/list ([i (in-range 10)])
    (from-nat other-ordered-pair/e i))
'((1 . 0)
  (2 . 0)
  (2 . 1)
  (3 . 1)
  (3 . 0)
  (4 . 1)
  (3 . 2)
  (4 . 2)
  (5 . 2)
  (4 . 0))
```

Some of the combinators in the library are guaranteed to build enumeration functions that are bijections. But since `map/e` accepts arbitrary functions and `or/e` accepts enumerations with arbitrary contracts, they may project enumerations that are not be bijections. To help avoid errors, the contracts on `map/e` and `or/e` does some random checking to see if the result would be a bijection. Here's an example that, with high probability, signals a contract violation.

```
> (map/e (λ (x) (floor (/ x 100)))
         (λ (x) (* x 100))
         natural/e
         #:contract natural?)
map/e: contract violation;
 new enumeration would not be two-way
 passing 878 to `from-nat` produces:
```

```
  878
which, when passed through `in' and `out', produces:
  800
which, when passed to `to-nat' produces 800,
but it should have been 878
  in: (->i
          ((in
            (e es c)
            (cond
              ((null? es) (-> (enum-contract e) c))
              (else
                (dynamic->*
                  #:mandatory-domain-contracts
                  (map enum-contract ...)
                  #:range-contracts
                  (list c)))))
           (out
            (e es c)
            (cond
              ((null? es) (-> c (enum-contract e)))
              (else
                (dynamic->*
                  #:mandatory-domain-contracts
                  (list c)
                  #:range-contracts
                  (map enum-contract ...)))))
           (e enum?)
           #:contract
           (c contract?))
          #:rest
          (es (listof enum?))
          #:pre/desc
          (in out e es)
          (appears-to-be-a-bijection?
            in
            out
            (cons e es))
          (result enum?))
  contract from:
        <pkgs>/data-enumerate-lib/data/enumerate.rkt
  blaming: top-level
    (assuming the contract is correct)
  at: <pkgs>/data-enumerate-lib/data/enumerate.rkt:45:3
```

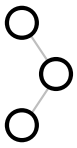
The contract on `or/e` has a similar kind of checking that attempts to find overlaps between

the elements of the enumerations in its arguments.

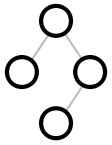
Sometimes, there is no easy way to make two functions that form a bijection. In that case you can use pam/e and supply only one function to make a one way enumeration. For example, we can make an enumeration of picts of binary trees like this:

```
(define pict-bt/e
  (pam/e
   (λ (bt)
     (binary-tidier
      (let loop ([bt bt])
        (cond
          [(list? bt) (apply tree-layout (map loop bt))]
          [else #f]))))
   bt/e
   #:contract pict?))
```

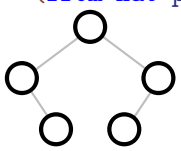
```
> (from-nat pict-bt/e 10)
```



```
> (from-nat pict-bt/e 11)
```



```
> (from-nat pict-bt/e 12)
```



Putting all these pieces together, here is a definition of an enumeration of closed expressions of the untyped lambda calculus.

```
(define/contract (lc-var/e bvs memo)
  (-> (set/c symbol?) (hash/c (set/c symbol?) enum?) enum?)
  ; memoization is a significant performance improvement
  (hash-ref!
   memo
   bvs
   (delay/e
```

```
    (or/e
     ; the variables currently in scope
     (apply fin/e (set->list bvs))

     ; the λ case; first we build a dependent
     ; pair of a bound variable and a body expression
     ; and then use map/e to build the usual syntax
     (map/e
      (λ (pr) `(λ (,(car pr)) ,(cdr pr)))
      (λ (λ-exp) (cons (caadr λ-exp) (caddr λ-exp)))
      (cons/de
       [hd symbol/e]
       [tl (hd) (lc-var/e (set-add bvs hd) memo)])
      #:contract (list/c 'λ (list/c symbol?) lc-exp?))

     ; application expressions
     (list/e (lc-var/e bvs memo) (lc-var/e bvs memo))))))

(define (lc-exp? x)
  (match x
    [(? symbol?) #t]
    [`(λ (,x) ,e) (and (symbol? x) (lc-exp? e))]
    [`(,a ,b) (and (lc-exp? a) (lc-exp? b))]))

(define lc/e (lc-var/e (set) (make-hash)))

> (from-nat lc/e 0)
'(λ (a) a)
> (from-nat lc/e 1)
'((λ (a) a) (λ (a) a))
> (from-nat lc/e 2)
'(λ (a) (λ (a) a))
> (to-nat lc/e
          '(λ (f)
             ((λ (x) (f (x x)))
              (λ (x) (f (x x))))))
120491078480010
```

11.1 Core Enumeration

```
(require data/enumerate)
```
package: `data-enumerate-lib`

The `data/enumerate` library contains the core subset of the enumeration library; its exports are described in the sections §11.2 “Enumeration Properties”, §11.3 “Querying Enu-

merations", and §11.4 "Constructing Enumerations".

There are more enumeration functions than just the core, provided by `data/enumerate/lib`.

11.2 Enumeration Properties

In addition to the functions that form the bijection, an enumeration also has a contract, a count, and three boolean properties associated with it: if it is finite or not, if it is a bijection to the natural numbers or merely maps from the natural numbers without going back, and if the contract it has is a `flat-contract?`.

The functions in this section are predicates for the boolean properties and selection functions for other properties.

When an enumeration prints out, it shows the first few elements of the enumeration and, if it is either a finite enumeration or a one way enumeration, it prints `finite` and `one-way`, as appropriate. If those prefixes are not printed, then the enumeration is not finite and is not one-way.

```
(enum? x) → boolean?
  x : any/c
```

Identifies a value as an enumeration.

```
(finite-enum? v) → boolean?
  v : any/c
```

Identifies *finite enumerations*.

```
(infinite-enum? v) → boolean?
  v : any/c
```

Identifies *infinite enumerations*, i. e., enumerations that map all natural numbers.

```
(two-way-enum? v) → boolean?
  v : any/c
```

Identifies *two way enumerations*, i. e., enumerations that can map back and forth from values that satisfy the enumeration's contract to the natural numbers.

```
(one-way-enum? v) → boolean?
  v : any/c
```

Identifies *one way enumerations*, i. e., enumerations that can map only from the natural numbers to values that satisfy the enumeration's contract, but not back.

```
(flat-enum? v) → boolean?
  v : any/c
```

Identifies *flat enumerations*, i. e., enumerations whose contracts are `flat-contract?`s.

```
(enum-count e) → natural?
  e : finite-enum?
```

Returns the number of elements of an enumeration.

```
(enum-contract e) → contract?
  e : enum?
```

Returns the `contract?` that `e` enumerates.

11.3 Querying Enumerations

The functions in this section exercise the enumeration, turning natural numbers back and forth to the values that an enumeration enumerates.

```
(from-nat e n) → (enum-contract e)
  e : enum?
  n : (if (finite-enum? e)
          (integer-in 0 (enum-count e))
          natural?)
```

Decodes `n` from `e`.

```
(to-nat e x) → (if (finite-enum? e)
                   (integer-in 0 (enum-count e))
                   natural?)
  e : two-way-enum?
  x : (enum-contract e)
```

Encodes `x` from `e`.

```
(enum->list e [n]) → (listof (enum-contract e))
  e : enum?
  n : (if (finite-enum? e)              = (enum-count e)
          (integer-in 0 (enum-count e))
          natural?)
```

Returns a list of the first *n* values in *e*.

If *n* is not supplied, then *e* must be a finite-enum.

Examples:

```
> (enum->list (list/e natural/e natural/e) 8)
'((0 0) (0 1) (1 0) (1 1) (0 2) (1 2) (2 0) (2 1))
> (enum->list (below/e 8))
'(0 1 2 3 4 5 6 7)
```

```
(in-enum e) → sequence?
  e : enum?
```

Constructs a sequence suitable for use with `for` loops.

Note that enumerations are also sequences directly, too.

Example:

```
> (for/list ([i (in-enum (below/e 5))])
    i)
'(0 1 2 3 4)
```

11.4 Constructing Enumerations

This section contains the fundamental operations for building enumerations.

```
natural/e : (and/c infinite-enum? two-way-enum? flat-enum?)
```

An enumeration of the natural numbers.

Examples:

```
> (from-nat natural/e 5)
5
> (to-nat natural/e 5)
5
```

```
(below/e max) → (and/c (if (= max +inf.0)
                           finite-enum?
                           infinite-enum?)
                       two-way-enum?
                       flat-enum?)
  max : (or/c natural? +inf.0)
```

An enumeration of the first *max* naturals or, if *max* is +inf.0, all of the naturals.

Example:

```
> (enum->list (below/e 10))
'(0 1 2 3 4 5 6 7 8 9)
```

```
empty/e : (and/c finite-enum? two-way-enum? flat-enum?)
```

The empty enumeration.

Example:

```
> (enum->list empty/e)
'()
```

```
(map/e f f-inv #:contract c e) → enum?
  f : (-> (enum-contract e) c)
  f-inv : (-> c (enum-contract e))
  c : contract?
  e : enum?
(map/e f f-inv #:contract c e ...+) → enum?
  f : (dynamic->* #:mandatory-domain-contracts (map enum-contract e)
                  #:range-contracts (list c))
  f-inv : (dynamic->* #:mandatory-domain-contracts (list c)
                      #:range-contracts (map enum-contract e))
  c : contract?
  e : enum?
```

Builds an enumeration of *c* from *e* by calling *f* on each element of the enumeration and *f-inv* of each value of *c*.

If multiple enumerations are supplied, *f* is expected to accept any combination of elements of the given enumerations, i. e., the enumerations are not processed in parallel like the lists in map, but instead any element from the first enumeration may appear as the first argument to *f* and any element from the second may appear as the second argument to *f*, etc.

If *e* is a one way enumeration, then the result is a one way enumeration and *f-inv* is ignored. Otherwise, the result is a two way enumeration.

Examples:

```
> (define evens/e
    (map/e (λ (x) (* x 2))
           (λ (x) (/ x 2))
```

```
           natural/e
           #:contract (and/c natural?
                             even?)))
> (enum->list evens/e 10)
'(0 2 4 6 8 10 12 14 16 18)
> (define odds/e
    (map/e add1
           sub1
           evens/e
           #:contract (and/c natural? odd?)))
> (enum->list odds/e 10)
'(1 3 5 7 9 11 13 15 17 19)
> (define ordered-pair/e
    (map/e (λ (x y) (cons x (+ x y)))
           (λ (p)
             (define x (car p))
             (define y (cdr p))
             (values x (- y x)))
           natural/e
           natural/e
           #:contract (and/c (cons/c natural? natural?)
                             (λ (xy) (<= (car xy) (cdr xy))))))
> (enum->list ordered-pair/e 10)
'((0 . 0)
  (0 . 1)
  (1 . 1)
  (1 . 2)
  (0 . 2)
  (1 . 3)
  (2 . 2)
  (2 . 3)
  (2 . 4)
  (0 . 3))
```

```
(pam/e f #:contract c e ...+) → one-way-enum?
  f : (dynamic->* #:mandatory-domain-contracts (map enum-contract e)
                  #:range-contracts (list c))
  c : contract?
  e : enum?
```

Builds a one way enumeration from the given enumerations, combining their elements with *f*, in a manner similar to map/e.

Examples:

```
> (define rationals/e
```

```
    (pam/e /
           (nat+/e 1)
           (nat+/e 2)
           #:contract (and/c exact? rational? positive?)))
> (enum->list rationals/e 10)
'(1/2 1/3 1 2/3 1/4 1/2 3/2 1 3/4 1/5)
```

```
(except/e e [#:contract c] x ...) → two-way-enum?
  e : two-way-enum?
  c : (or/c #f contract?) = #f
  x : (enum-contract e)
```

Returns a two way enumeration identical to *e* except that all *x* are removed from the enumeration. See also but-not/e.

If *c* is #f, then it is not treated as a contract, instead the resulting contract is synthesized from contract on *e* and the *x*s.

Examples:

```
> (define except-1/e
    (except/e natural/e 3))
> (from-nat except-1/e 2)
2
> (from-nat except-1/e 4)
5
> (to-nat except-1/e 2)
2
> (to-nat except-1/e 4)
3
```

```
(or/e [#:one-way-enum? one-way-enum?] e-p ...) → enum?
  one-way-enum? : boolean? = #f
  e-p : (or/c enum? (cons/c enum? (-> any/c boolean?)))
```

An enumeration of all of the elements of the enumerations in the *e-p* arguments.

If the enumerations have overlapping elements, then pass #t as *one-way-enum?* so the result is a one way enumeration.

In more detail, if all of the arguments have or are two way enumerations and *one-way-enum?* is #f, then the result is also a two way enumeration and each argument must come with a predicate to distinguish its elements from the elements of the other enumerations. If the argument is a pair, then the predicate in the second position of the pair is used. If the argument is an enumeration, then it must be a flat enumeration and the contract is used as its predicate.

If any of the arguments are one way enumerations (or *one-way-enum?* is not #f), then the result is a one way enumeration and any predicates in the arguments are ignored.

Example:

```
> (enum->list (or/e natural/e (list/e natural/e natural/e))
              10)
'(0 (0 0) 1 (0 1) 2 (1 0) 3 (1 1) 4 (0 2))
```

```
(append/e [#:one-way-enum? one-way-enum?]
          e-p ...+)                          → enum?
  one-way-enum? : boolean? = #f
  e-p : (or/c enum? (cons/c enum? (-> any/c boolean?)))
```

An enumeration of the elements of the enumerations given in *e-p* that enumerates the elements in order that the enumerations are supplied. All but the last enumeration must be finite.

Like or/e the resulting enumeration is either a one way enumeration or a two way enumeration depending on the status of the arguments, and append/e has the same constraints on overlapping elements in the arguments.

Example:

```
> (enum->list
   (append/e (take/e natural/e 4)
             (list/e natural/e natural/e))
   10)
'(0 1 2 3 (0 0) (0 1) (1 0) (1 1) (0 2) (1 2))
```

```
(thunk/e  eth
         [#:count count
          #:two-way-enum? is-two-way-enum?
          #:flat-enum? is-flat-enum?])      → enum?
  eth : (-> (and/c (if (= count +inf.0)
                       infinite-enum?
                       (and/c finite-enum?
                              (let ([matching-count? (λ (e) (= (enum-count e) count))])
                                matching-count?)))
                   (if is-two-way-enum?
                       two-way-enum?
                       one-way-enum?)
                   (if is-flat-enum?
                       flat-enum?
                       (not/c flat-enum?))))
  count : (or/c +inf.0 natural?) = +inf.0
```

```
  is-two-way-enum? : any/c = #t
  is-flat-enum? : any/c = #t
```

A delayed enumeration identical to the result of *eth*.

The *count*, *is-two-way-enum?*, and *is-flat-enum?* arguments must be accurate predications of the properties of the result of *eth*.

The argument *eth* is invoked when the result enumeration's contract or bijection is used, either directly or indirectly via a call to `enum-contract`, `from-nat`, or `to-nat`.

Example:

```
> (letrec ([bt/e (thunk/e
                  (λ ()
                    (or/e (single/e #f)
                          (list/e bt/e bt/e))))])
    (enum->list bt/e 5))
'(#f (#f #f) (#f (#f #f)) ((#f #f) #f) ((#f #f) (#f #f)))
```

```
(list/e [#:ordering ordering] e ...) → enum?
  ordering : (or/c 'diagonal 'square) = 'square
  e : enum?
```

An enumeration of lists of values enumerated by the `e`.

If *ordering* is `'square`, it uses a generalized form of Szudzik's "elegant" ordering and if *ordering* is `'diagonal`, it uses a generalized form of Cantor's mapping from pairs of naturals to naturals.

Examples:

```
> (enum->list (list/e
               (fin/e "Brian" "Jenny" "Ki" "Ted")
               natural/e
               (fin/e "Terra" "Locke" "Edgar" "Mash"))
              5)
'(("Brian" 0 "Terra")
  ("Jenny" 0 "Terra")
  ("Ki" 0 "Terra")
  ("Ted" 0 "Terra")
  ("Brian" 0 "Locke"))
> (enum->list (list/e natural/e natural/e)
              10)
'((0 0) (0 1) (1 0) (1 1) (0 2) (1 2) (2 0) (2 1) (2 2) (0 3))
```

```
> (enum->list (list/e #:ordering  'diagonal natural/e natural/e)
              10)
'((0 0) (0 1) (1 0) (0 2) (1 1) (2 0) (0 3) (1 2) (2 1) (3 0))
```

```
(dep/e  e
        f
       [#:f-range-finite? f-range-finite?
        #:flat? flat?
        #:one-way? one-way?])              → enum?
  e : enum?
  f : (-> (enum-contract e)
          (and/c (if f-range-finite?
                     finite-enum?
                     infinite-enum?)
                 (if one-way?
                     one-way-enum?
                     two-way-enum?)
                 (if flat?
                     flat-enum?
                     (not/c flat-enum?))))
  f-range-finite? : boolean? = #f
  flat? : boolean? = #t
  one-way? : boolean? = (one-way-enum? e)
```

Constructs an enumeration of pairs like the first case of `cons/de`.

Examples:

```
> (define dep/e-ordered-pair/e
    (dep/e natural/e
           (λ (hd) (nat+/e (+ hd 1)))))
> (enum->list dep/e-ordered-pair/e 10)
'((0 . 1)
  (0 . 2)
  (1 . 2)
  (1 . 3)
  (0 . 3)
  (1 . 4)
  (2 . 3)
  (2 . 4)
  (2 . 5)
  (0 . 4))
```

```
(bounded-list/e k n)
 → (and/c finite-enum? two-way-enum? flat-enum?)
```

```
k : natural?
n : natural?
```

An enumeration of tuples of naturals with `max` *n* of length *k*.

Example:

```
> (enum->list (bounded-list/e 3 2)
              5)
'((0 0 2) (1 0 2) (0 1 2) (1 1 2) (0 2 0))
```

11.5 More Enumeration Operations

`(require data/enumerate/lib)` package: `data-enumerate-lib`

The `data/enumerate/lib` library extends the `data/enumerate` library with some higher-level enumerations and functions on enumerations. Its contents are described in the sections §11.6 “Derived Enumeration Constructors”, §11.7 “Enumeration Utility”, and §11.8 “Pre-built Enumerations”.

11.6 Derived Enumeration Constructors

```
(cons/de [car-id car-enumeration-expr]
         [cdr-id (car-id) cdr-enumeration-expr]
         cons/de-option)
(cons/de [car-id (cdr-id) car-enumeration-expr]
         [cdr-id cdr-enumeration-expr]
         cons/de-option)

cons/de-option =
               | #:dep-expression-finite? expr cons/de-option
               | #:flat? expr cons/de-option
               | #:one-way? expr cons/de-option
```

Constructs an enumeration of pairs where the first component of the pair is drawn from the *car-enumeration-expr*’s value and the second is drawn from the *cdr-enumeration-expr*’s value.

In the first form, the *cdr-enumeration-expr* can use *car-id*, which is bound to the value of the car position of the pair, mutatis mutandis in the second case.

If `#:dep-expression-finite?` keyword and expression are present, then the value of the dependent expression is expected to be an infinite enumeration if the expression evaluates

to `#f` and a finite enumeration otherwise. If the keyword is not present, then the dependent expressions are expected to always produce infinite enumerations.

If `#:flat?` is present and evaluates to a true value, then the value of both sub-expressions are expected to be flat enumerations and if it evaluates to `#f`, then the enumerations must not be flat enumerations. If the keyword is not present, then the dependent expressions are expected to always produce flat enumerations.

If `#:one-way?` is present and evaluates to a true value, then the result enumeration is a one way enumeration

The dependent expressions are expected to always produce two way enumerations if the non-dependent expression is a two way enumeration and the dependent the dependent expressions are expected to always produce one way enumerations if the non-dependent expression is a one way enumeration.

Examples:

```
> (define ordered-pair/e
    (cons/de [hd natural/e]
             [tl (hd) (nat+/e (+ hd 1))]))
> (enum->list ordered-pair/e 10)
'((0 . 1)
  (0 . 2)
  (1 . 2)
  (1 . 3)
  (0 . 3)
  (1 . 4)
  (2 . 3)
  (2 . 4)
  (2 . 5)
  (0 . 4))
```

```
(flip-dep/e  e
             f
            [#:f-range-finite? f-range-finite?]
             #:flat? flat?
            [#:one-way? one-way?])              → enum?
  e : enum?
```

```
f : (-> (enum-contract e)
        (and/c (if f-range-finite?
                   finite-enum?
                   infinite-enum?)
               (if one-way?
                   one-way-enum?
                   two-way-enum?)
               (if flat?
                   flat-enum?
                   (not/c flat-enum?))))
f-range-finite? : boolean? = #f
flat? : #t
one-way? : boolean? = (one-way-enum? e)
```

Constructs an enumeration of pairs like the second case of `cons/de`.

Examples:

```
> (define flip-dep/e-ordered-pair/e
    (flip-dep/e natural/e
                (λ (tl) (below/e tl))
                #:f-range-finite? #t))
> (enum->list flip-dep/e-ordered-pair/e 10)
'((0 . 1)
  (0 . 2)
  (1 . 2)
  (0 . 3)
  (1 . 3)
  (2 . 3)
  (0 . 4)
  (1 . 4)
  (2 . 4)
  (3 . 4))
```

```
(cons/e e1 e2 [#:ordering ordering]) → enum?
  e1 : enum?
  e2 : enum?
  ordering : (or/c 'diagonal 'square) = 'square
```

An enumeration of pairs of the values from *e1* and *e2*. Like `list/e`, the *ordering* argument controls how the resting elements appear.

Examples:

```
> (enum->list (cons/e (take/e natural/e 4) (take/e natural/e 5)) 5)
```

```
'((0 . 0) (1 . 0) (2 . 0) (3 . 0) (0 . 1))
> (enum->list (cons/e natural/e (take/e natural/e 5)) 5)
'((0 . 0) (0 . 1) (0 . 2) (0 . 3) (0 . 4))
> (enum->list (cons/e (take/e natural/e 4) natural/e) 5)
'((0 . 0) (1 . 0) (2 . 0) (3 . 0) (0 . 1))
> (enum->list (cons/e natural/e natural/e) 5)
'((0 . 0) (0 . 1) (1 . 0) (1 . 1) (0 . 2))
```

```
(listof/e  e
           [#:simple-recursive? simple-recursive?]) → enum?
  e : (if simple-recursive?
          enum?
          infinite-enum?)
  simple-recursive? : any/c = #t
```

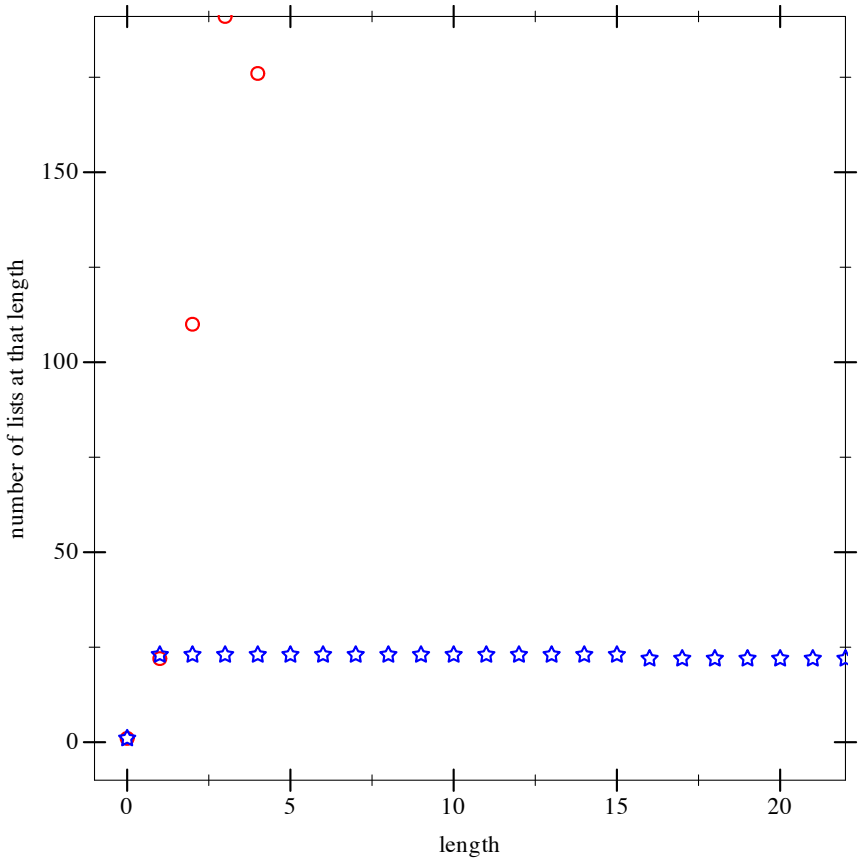
An enumeration of lists of values enumerated by `e`.

If *simple-recursive?* is `#f`, then the enumeration is constructed by first choosing a length and then using `list/e` to build lists of that length. If not, it builds a recursive enumeration using `delay/e`. The second option (which is the default) method is significantly more efficient when calling `from-nat` with large numbers, but it also has much shorter lists near the beginning of the enumeration.

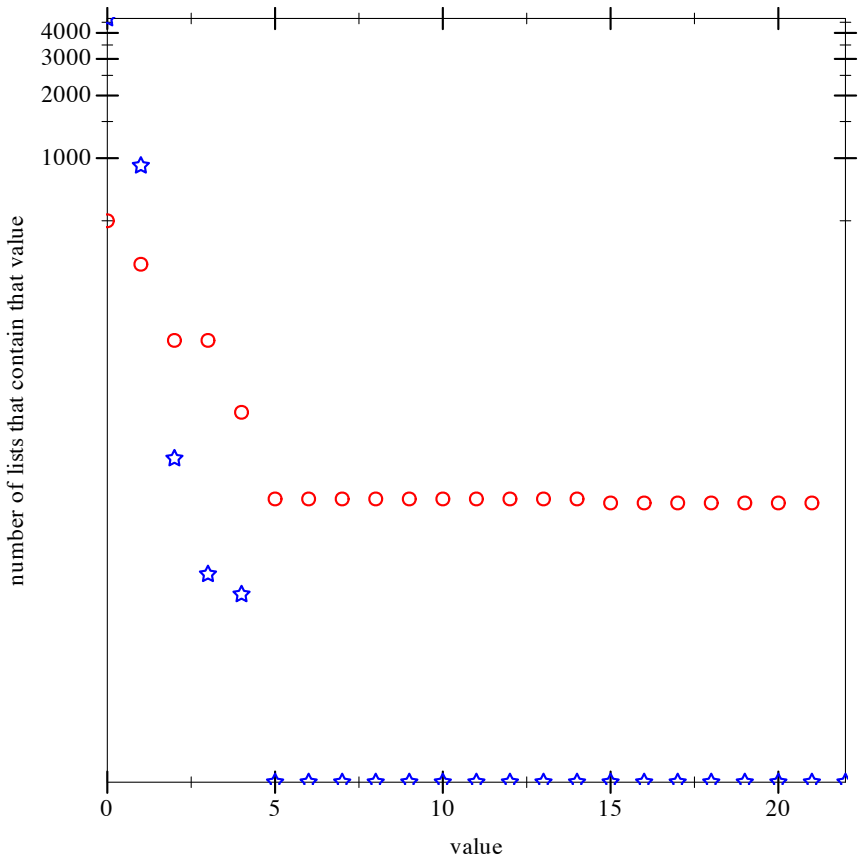
Examples:

```
> (enum->list (listof/e natural/e #:simple-recursive? #f) 10)
'(() (0) (1) (0 0) (0 1) (2) (1 0) (0 0 0) (0 0 1) (0 1 0))
> (enum->list (listof/e natural/e) 10)
'(() (0) (0 0) (1) (1 0) (0 0 0) (1 0 0) (2) (2 0) (2 0 0))
> (to-nat (listof/e natural/e #:simple-recursive? #f) '(1 2 3 4 5 6))
2929082647
> (to-nat (listof/e natural/e) '(1 2 3 4 5 6))
19656567028457999961819135393421096124461042490733963
```

This plot shows some statistics for the first 500 items in each enumeration. The first plot shows how many different lengths each encounters. The red circles are when the `#:simple-recursive?` argument is `#t` and the blue stars are when that argument is `#f`.



This plot shows the different values, but this time on a log scale. As you can see, zero appears much more frequently when the `#:simple-recursive?` argument is `#f`.




```
(non-empty-listof/e  e
                     [#:simple-recursive? simple-recursive?])
 → enum?
  e : (if simple-recursive?
          enum?
          infinite-enum?)
  simple-recursive? : any/c = #t
```

Like `listof/e`, but without the empty list.

Example:

```
> (enum->list (non-empty-listof/e natural/e) 5)
'((0) (0 0) (1) (1 0) (0 0 0))
```

```
(listof-n/e e n) → enum?
  e : (if simple-recursive?
          enum?
          infinite-enum?)
```

```
n : natural?
```

Example:

```
> (enum->list (listof-n/e natural/e 3) 10)
'((0 0 0)
  (0 0 1)
  (0 1 0)
  (0 1 1)
  (1 0 0)
  (1 0 1)
  (1 1 0)
  (1 1 1)
  (0 0 2)
  (1 0 2))
```

```
(delay/e enum-expression ... keyword-options)

keyword-options =
                | #:count count-expression keyword-options
                | #:two-way-enum? two-way-boolean-expression keyword-options
                | #:flat-enum? flat-boolean-expression keyword-options
```

Returns an enumeration immediately, without evaluating the *enum-expression*s. When the result enumeration is inspected (directly or indirectly) via `from-nat`, `to-nat`, or `enum-contract`, the *enum-expression*s are evaluated and the value of the last one is cached. The value is then used as the enumeration.

If the *count-expression* is not supplied or if it evaluates to `+inf.0`, the resulting enumeration is a infinite enumeration. Otherwise the expression must evaluate to an `natural?` and the resulting enumeration is a finite enumeration of the given count.

If *two-way-boolean-expression* is supplied and it evaluates to anything other than `#f`, the resulting enumeration must be a two way enumeration; otherwise it must be a one way enumeration.

If *flat-boolean-expression* is supplied and it evaluates to anything other than `#f`, the resulting enumeration must be a flat enumeration; otherwise it must not be.

This expression form is useful for building recursive enumerations.

Example:

```
> (letrec ([bt/e (delay/e
                  (or/e (single/e #f)
                        (list/e bt/e bt/e)))])
```

```
    (enum->list bt/e 5))
'(#f (#f #f) (#f (#f #f)) ((#f #f) #f) ((#f #f) (#f #f)))
```

```
(take/e e n #:contract contract) → finite-enum?
  e : enum?
  n : (if (finite-enum? e)
          (integer-in 0 (enum-count e))
          natural?)
  contract : (λ (x)
               (and ((enum-contract e) x)
                    (< (to-nat e x) n)))
```

Identical to *e* but only includes the first *n* values.

If the *contract* argument is not supplied, then *e* must be both a two way enumeration and a flat enumeration.

Example:

```
> (enum->list (take/e natural/e 5))
'(0 1 2 3 4)
```

```
(slice/e e lo hi #:contract contract) → finite-enum?
  e : enum?
  lo : (and/c (if (finite-enum? e)
                  (integer-in 0 (enum-count e))
                  natural?)
              (<=/c hi))
  hi : (if (finite-enum? e)
           (integer-in 0 (enum-count e))
           natural?)
  contract : (and/c (enum-contract e)
                    (λ (x)
                      (<= lo (to-nat e x))
                      (< (to-nat e x) hi)))
```

Identical to *e* but only includes the values between *lo* (inclusive) and *hi* (exclusive).

Examples:

```
> (enum->list (slice/e natural/e 5 10))
'(5 6 7 8 9)
> (slice/e natural/e 20 20)
#<empty-enum>
```

```
(fin/e x ...) → (and/c finite-enum? flat-enum?)
  x : any/c
```

Builds an enumeration containing each `x`, in the order given.

If there are multiple arguments, then they must all be distinct; numbers except for `+nan.0` and `+nan.0` are compared using `=` and all other values (including `+nan.0` and `+nan.0`) are compared using `equal?`.

If some other equality function is appropriate, use `map/e` with `(below/e n)` as the first argument to explicitly specify how to differentiate the elements of the enumeration.

If all of the arguments match the contract

```
(or/c symbol? boolean? char? keyword? null?
      string? bytes? number?)
```

then the result is a two way enumeration, otherwise it is a one way enumeration.

Examples:

```
> (enum->list (fin/e "Brian" "Jenny" "Ki" "Ted"))
'("Brian" "Jenny" "Ki" "Ted")
> (enum->list (fin/e 1 3 5 7 9 11 13 15))
'(1 3 5 7 9 11 13 15)
```

```
(single/e v #:equal? same?) → (and/c finite-enum? two-way-enum?)
  v : any/c
  same? : equal?
```

Returns an enumeration of count one containing only `v`.

It uses *same?* to build the contract in the enumeration, always passing `v` as the first argument to *same?*.

Examples:

```
> (enum->list (single/e 12345))
'(12345)
> (enum->list (single/e (λ (x) x)))
'(#<procedure>)
```

```
(range/e lo hi) → (and/c two-way-enum? flat-enum?)
  lo : (and/c (or/c -inf.0 exact-integer?)
              (<=/c hi))
  hi : (or/c exact-integer? +inf.0)
```

An enumeration of the exact integers between *lo* and *hi*.

Examples:

```
> (enum->list (range/e 10 20))
'(10 11 12 13 14 15 16 17 18 19 20)
> (enum->list (range/e 10 10))
'(10)
> (enum->list (range/e -inf.0 0) 10)
'(0 -1 -2 -3 -4 -5 -6 -7 -8 -9)
> (enum->list (range/e -inf.0 +inf.0) 10)
'(0 1 -1 2 -2 3 -3 4 -4 5)
```

```
(nat+/e lo) → (and/c infinite-enum? two-way-enum? flat-enum?)
  lo : natural?
```

An enumeration of natural numbers larger than *lo*.

Example:

```
> (enum->list (nat+/e 42) 5)
'(42 43 44 45 46)
```

```
(but-not/e big small) → two-way-enum?
  big : two-way-enum?
  small : (and/c two-way-enum? flat-enum? finite-enum?)
```

Returns a two way enumeration like *big* except that the elements of *small* are removed. Every element in *small* must also be in *big*. See also `except/e`.

This operation is the one from Yorgey and Foner (2018)'s paper on subtracting bijections.

Example:

```
> (enum->list (but-not/e (below/e 10) (below/e 5)))
'(5 6 7 8 9)
```

Generally, `but-not/e` produces an enumeration that performs better than the result of `(apply except/e big (enum->list small))` when the range of *small* is a large set. When it is small, using `except/e` performs better.

The two enumerations may also be in different orders.

Examples:

```
> (define (evens-below/e n)
    (map/e (λ (x) (* x 2))
           (λ (x) (/ x 2))
           (below/e (/ n 2))
           #:contract (and/c natural? even? (<=/c n))))
> (enum->list
   (but-not/e (below/e 20)
              (evens-below/e 20)))
'(5 11 3 13 7 15 1 17 9 19)
> (enum->list
   (apply except/e (below/e 20)
          (enum->list (evens-below/e 20))))
'(1 3 5 7 9 11 13 15 17 19)
```

```
(vector/e [#:ordering ordering] e ...) → enum?
  ordering : (or/c 'diagonal 'square) = 'square
  e : enum?
```

An enumeration of vectors of values enumerated by the *e*.

The *ordering* argument is the same as the one to `list/e`.

Example:

```
> (enum->list (vector/e (fin/e "Brian" "Jenny" "Ki" "Ted")
                        natural/e
                        (fin/e "Terra" "Locke" "Edgar" "Mash"))
              5)
'(#("Brian" 0 "Terra")
  #("Jenny" 0 "Terra")
  #("Ki" 0 "Terra")
  #("Ted" 0 "Terra")
  #("Brian" 0 "Locke"))
```

```
(permutations-of-n/e n)
 → (and/c finite-enum? two-way-enum? flat-enum?)
  n : natural?
```

Returns an enumeration of the permutations of the natural numbers smaller than *n*.

Example:

```
> (enum->list (permutations-of-n/e 3))
'((0 1 2) (0 2 1) (1 0 2) (1 2 0) (2 0 1) (2 1 0))
```

```
(permutations/e l) → enum?
  l : list?
```

Returns an enumeration of the permutations of `l`.

Example:

```
> (enum->list (permutations/e '(Brian Jenny Ted Ki)))
'((Brian Jenny Ted Ki)
  (Brian Jenny Ki Ted)
  (Brian Ted Jenny Ki)
  (Brian Ted Ki Jenny)
  (Brian Ki Jenny Ted)
  (Brian Ki Ted Jenny)
  (Jenny Brian Ted Ki)
  (Jenny Brian Ki Ted)
  (Jenny Ted Brian Ki)
  (Jenny Ted Ki Brian)
  (Jenny Ki Brian Ted)
  (Jenny Ki Ted Brian)
  (Ted Brian Jenny Ki)
  (Ted Brian Ki Jenny)
  (Ted Jenny Brian Ki)
  (Ted Jenny Ki Brian)
  (Ted Ki Brian Jenny)
  (Ted Ki Jenny Brian)
  (Ki Brian Jenny Ted)
  (Ki Brian Ted Jenny)
  (Ki Jenny Brian Ted)
  (Ki Jenny Ted Brian)
  (Ki Ted Brian Jenny)
  (Ki Ted Jenny Brian))
```

```
(set/e e) → enum?
  e : enum?
```

Returns an enumeration of finite sets of values from `e`.

Examples:

```
> (enum->list (set/e (fin/e "Brian" "Jenny" "Ki")))
(list
 (set)
 (set "Brian")
```

```
 (set "Jenny")
 (set "Brian" "Jenny")
 (set "Ki")
 (set "Brian" "Ki")
 (set "Jenny" "Ki")
 (set "Brian" "Jenny" "Ki"))
> (enum->list (set/e natural/e) 10)
(list
 (set)
 (set 0)
 (set 1)
 (set 0 1)
 (set 2)
 (set 0 2)
 (set 1 2)
 (set 0 1 2)
 (set 3)
 (set 0 3))
```

(infinite-sequence/e *e*) → one-way-enum?
e : finite-enum?

Returns an enumeration of infinite sequences of elements of *e*. If *e* is an empty enumeration, returns an empty enumeration.

The infinite sequence corresponding to the natural number *n* is based on dividing the bits of (* (+ 1 *n*) pi) into chunks of bits where the largest value is (enum-count *e*). Since (* (+ 1 *n*) pi) has infinite digits, there are infinitely many such chunks. Since * is defined on all naturals, there are infinitely many such numbers. The generation of the sequence is efficient in the sense that the digits are generated incrementally without needing to go deeper than to find the requested value. The generation of the sequence is inefficient in the sense that the approximation of (* (+ 1 *n*) pi) gets larger and larger as you go deeper into the sequence.

Examples:

```
> (define bjtks/e (infinite-sequence/e
                   (fin/e 'Brian 'Jenny 'Ted 'Ki)))
> (for ([e (from-nat bjtks/e 42)]
        [i (in-range 10)])
    (printf "~a = ~a\n" i e))
0 = Ted
1 = Brian
2 = Ted
3 = Jenny
4 = Jenny
```

```
5 = Jenny
6 = Ki
7 = Jenny
8 = Ki
9 = Jenny
```

```
(hash-traverse/e f
                 xs
                 #:get-contract get-contract
                 #:contract contract)        → enum?
  f : (-> any/c enum?)
  xs : (hash/c any/c any/c)
  get-contract : (-> any/c contract?)
  contract : contract?
```

Constructs an enumeration that simultaneously enumerates each of the enumerations returned by *f* applied to each value of *xs*.

If supplied, the *get-contract* argument is applied to the keys in the hash and is expected to return the contract for the corresponding enumeration. If the *contract* argument is supplied, it is used directly as the contract for all of enumerations. One of the two arguments must be supplied.

Examples:

```
> (define hash-traverse-1/e
    (let ([h (hash "Brian" 5 "Jenny" 15 "Ted" 25 "Ki" 30)])
      (hash-traverse/e (λ (n) (below/e n))
                       h
                       #:get-contract
                       (λ (v) (and/c exact-integer? (<=/c (hash-
ref h v)))))))
> (enum->list hash-traverse-1/e 5)
'(#hash(("Brian" . 0) ("Jenny" . 0) ("Ki" . 0) ("Ted" . 0))
  #hash(("Brian" . 1) ("Jenny" . 0) ("Ki" . 0) ("Ted" . 0))
  #hash(("Brian" . 2) ("Jenny" . 0) ("Ki" . 0) ("Ted" . 0))
  #hash(("Brian" . 3) ("Jenny" . 0) ("Ki" . 0) ("Ted" . 0))
  #hash(("Brian" . 4) ("Jenny" . 0) ("Ki" . 0) ("Ted" . 0)))
> (to-nat hash-traverse-1/e
          '#hash(("Brian" . 4) ("Jenny" . 1) ("Ted" . 16) ("Ki" .
7)))
14334
```

```
(fold-enum f
           bs
           #:f-range-finite? f-range-finite?) → enum?
```

```
f : (if f-range-finite?
        (-> list? any/c finite-enum?)
        (-> list? any/c infinite-enum?))
bs : list?
f-range-finite? : #f
```

This is like `foldr`, but `f` returns enumerations of `a`s and assumes that the accumulator is initialized to `'()`.

Examples:

```
> (define fold-enum-1/e
    (fold-enum (λ (as b)
                 (below/e (+ (foldr + 0 as) b)))
               (list 1 2 3)
               #:f-range-finite? #t))
> (enum->list fold-enum-1/e 5)
'((0 0 0) (0 0 1) (0 0 2) (0 1 0) (0 1 1))
> (to-nat fold-enum-1/e (list 0 1 1))
4
```

11.7 Enumeration Utility

```
(random-index e) → natural?
  e : enum?
```

Returns a random index into `e`. This works for finite and infinite enumerations, regardless of the count of the enumeration. For finite enumerations, it picks an index uniformly at random using `random-natural` and for infinite enumerations it picks a natural number `n` from the geometric distribution and uses that as an exponent, picking uniformly at random in the interval between `(expt 2 n)` and `(expt 2 (+ n 1))`.

Examples:

```
> (random-index natural/e)
209981690501148218696281322364071115691769047798500714 9
> (random-index (below/e 5000000000))
3049866119
```

11.8 Pre-built Enumerations

This section describes enumerations of some common Racket datatypes.

```
char/e : (and/c finite-enum? two-way-enum? flat-enum?)
```

An enumeration of characters.

Examples:

```
> (enum->list char/e 5)
'(#\a #\b #\c #\d #\e)
> (to-nat char/e #\λ)
955
```

```
string/e : (and/c infinite-enum? two-way-enum? flat-enum?)
```

An enumeration of strings.

Examples:

```
> (enum->list string/e 5)
'("a" "b" "c" "d" "e")
> (to-nat string/e "racket")
340156678982215611231612783145 14
```

```
bool/e : (and/c finite-enum? two-way-enum? flat-enum?)
```

An enumeration of booleans.

Example:

```
> (enum->list bool/e)
'(#t #f)
```

```
symbol/e : (and/c infinite-enum? two-way-enum? flat-enum?)
```

An enumeration of symbols.

Examples:

```
> (enum->list symbol/e 5)
'(a b c d e)
> (to-nat symbol/e 'racket/base)
144633637012508760595483770150029186853157166750279774482575 54
```

```
integer/e : (and/c infinite-enum? two-way-enum? flat-enum?)
```

An enumeration of the integers.

Example:

```
> (enum->list integer/e 10)
'(0 1 -1 2 -2 3 -3 4 -4 5)
```

```
flonum/e : (and/c infinite-enum? two-way-enum? flat-enum?)
```

An enumeration of `flonum?`s.

Examples:

```
> (enum->list flonum/e 10)
'(+inf.0 -inf.0 +nan.0 0.0 5e-324 -5e-324 1e-323 -1e-323 1.5e-323
-1.5e-323)
> (to-nat flonum/e 1.0)
9214364837600034818
> (to-nat flonum/e -1.0)
9214364837600034819
```

```
exact-rational/e
 : (and/c infinite-enum? one-way-enum? flat-enum?)
```

An enumeration of rational numbers that duplicates entries (roughly, it enumerates all pairs of integers and natural numbers and then divides them which leads to duplicates).

Example:

```
> (enum->list exact-rational/e 13)
'(0 1/2 -1/2 1/3 -1/3 1 -1 2/3 -2/3 1/4 -1/4 1/2 -1/2)
```

```
two-way-real/e : (and/c infinite-enum? two-way-enum? flat-enum?)
```

An enumeration of reals; it includes only `integer/e` and `flonum/e`.

Example:

```
> (enum->list two-way-real/e 5)
'(0 +inf.0 1 -inf.0 -1)
```

```
real/e : (and/c infinite-enum? one-way-enum? flat-enum?)
```

An enumeration of reals; it includes `exact-rational/e` and `flonum/e`.

Example:

```
> (enum->list real/e 10)
'(+inf.0 0 -inf.0 1/2 +nan.0 -1/2 0.0 1/3 5e-324 -1/3)
```

```
two-way-number/e
 : (and/c infinite-enum? two-way-enum? flat-enum?)
```

An enumeration of numbers; it includes `two-way-real/e` and complex numbers made from pairs of those real numbers.

Example:

```
> (enum->list two-way-number/e 10)
'(+inf.0
  0
  +inf.0+inf.0i
  1
  0.0+inf.0i
  -inf.0+inf.0i
  -inf.0
  0+1i
  +nan.0+inf.0i
  -1)
```

```
number/e : (and/c infinite-enum? one-way-enum? flat-enum?)
```

An enumeration of numbers; it includes `real/e` and complex numbers made from pairs of those real numbers.

Example:

```
> (enum->list number/e 10)
'(+inf.0 +inf.0+inf.0i 0 0 -inf.0+inf.0i 0+1/2i -inf.0
+nan.0+inf.0i 1/2 1/2)
```

Bibliography

Brent Yorgey and Kenneth Foner. What's the difference? A Functional Pearl on Subtracting Bijections. In Proc. *International Conference on Functional Programming*, 2018.