0.5

hoard

A Collections Library for Bigloo scheme
User manual for version 0.5
May

Joseph Donaldson

Copyright © 2016-17 Joseph Donaldson

hoard is free software: you can redistribute it and/or modify
it under the terms of the GNU Lesser General Public License as
published by the Free Software Foundation, either version 3 of the
License, or (at your option) any later version.

hoard is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with collections. If not, see
<http://www.gnu.org/licenses/>.

1 Overview of hoard

hoard is a collections library for Bigloo. It consists of a number of generic protocols and corresponding concrete implementations. Currently, it supports generic protocols for abstract collections, dictionaries, sets, queues, stacks, and priority queues as well as generic protocols for enumeration (i.e., iteration) and comparison. Multiple concrete implementations of each protocol are provided. For example, there are both sorted/tree and hash based versions of sets and dictionaries.

Note: hoard heavily leverages bigloo-specific functionality, such as the object system, modules, and keyword arguments. Porting to other scheme systems would require significant effort.

2 Protocols

This chapter documents the protocols provided by hoard. Each protocol is described, the api documented, and examples given.

2.1 collection

2.1.1 Overview

The collection protocol provides the operations expected of an entity that represents an aggregate or container of other entities.

2.1.2 API Reference

`collection?` *object* [generic]

returns a boolean indicating whether or not `object` supports the collection protocol.

`collection-length` *object* [generic]

requires `object` support the `collection` protocol

returns the length of the `collection` `object`

`collection-empty?` *object* [generic]

requires `object` support the `collection` protocol

returns a boolean indicating whether the `collection` `object` is empty or not

`collection-contains?` *object itm* [generic]

requires `object` support the `collection` protocol and `itm` be an arbitrary value

returns a boolean indicating whether or not `itm` is in `object`

Note `collection-contains?` uses `equal?` for comparison, and for dictionary-like entities such as a hashtable or sorted-dictionary, it determines whether `itm` is a value in the collection, not whether it is a key in the collection. If you want to check to see if a key exists in such a collection, use `dictionary-contains?`

`collection-enumerator` *object* [generic]

requires `object` support the `collection` protocol

returns an `enumerator`

note an `enumerator` provides the ability to enumerate or iterate through the items in a collection.

`collection-copy` *object* [generic]

requires `object` support the `collection` protocol

returns a shallow copy of `object`

2.1.3 Examples

The standard Bigloo lists, vectors, strings, and hashtables are collections.

```
(collection? (list 1 2 3))
   ⇒ #t

(collection? (vector 1 2 3))
   ⇒ #t

(collection? (create-hashtable))
   ⇒ #t

(collection? "example")
   ⇒ #t
```

We can check to see if a collection is empty, obtain its length, and check to see if an item is contained with in it.

```
(collection-length (list 1 2 3))
   ⇒ 3

(collection-empty? '())
   ⇒ #t

(collection-empty? (list 1 2 3))
   ⇒ #f

(collection-contains? (list 1 2 3) 3)
   ⇒ #t
```

When needed, we can also obtain a shallow copy of a collection.

```
(let* ((vec1 (vector 1 2 3))
       (vec2 (collection-copy vec1)))
   (and (equal? vec1 vec2)
        (not (eq? vec1 vec2))))
  ⇒ #t
```

The `collection-enumerator` function is used to implement the `enumerable` functionality. See Section 2.6 [enumerable], page 9,

2.2 indexable

2.2.1 Overview

The indexable protocol specifies the operations expected of an indexable collection. An indexable collection is a collection which individual items can be referenced and potentially set via a key of some sort. This key can be an integer as is the case of lists, vectors, and strings, or arbitrary objects in the case of dictionary types.

2.2.2 API Reference

`collection-indexable?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `indexable` protocol

`collection-ref` *object key* [*default-if-not-found*] [generic]

requires `object` support the `indexable` protocol, `key` be an arbitrary value, and optionally, `default-if-not-found` be an arbitrary value

returns the value at the given `key` or throws `&invalid-index-exception` unless `default-if-not-found` is provided in which case `default-if-not-found` is returned

note For vectors, lists, and their ilk, `key` is an integer value representing the position in the collection. For dictionaries, `key` is the traditional key associated with such types.

`collection-set!` *object key value* [generic]

requires `object` support the `indexable` protocol and both `key` and `value` be arbitrary values **returns** `#unspecified` or throws `&unsupported-operation-exception` if `collection-set!` is not supported.

modifies `object` so that value at `key` is now `value` A generic function that sets the value found at key to the given value. If the key is not valid for the collection, `&invalid-index-exception` is thrown.

`collection-slice` *object keys* [generic]

requires `object` support the `indexable` protocol and `keys` be an object implementing the `enumerable` protocol

returns an `enumerator` providing access to the elements in the collection `object` represented by the provided enumerable of `keys`.

2.2.3 Examples

The standard Bigloo lists, vectors, strings, and hashtables are indexable.

```
(collection-indexable? (list 1 2 3))
   ⇒ #t

(collection-indexable? (vector 1 2 3))
   ⇒ #t

(collection-indexable? "test string")
   ⇒ #t

(collection-indexable? (create-hashtable))
   ⇒ #t
```

For collections that are position addressable such as lists, vectors, and strings as well as other, you can reference each individual element with `collection-ref` given its position as a key.

```
(collection-ref (list 1 2 3) 1)
   ⇒ 2

(collection-ref (vector 1 2 3) 2)
   ⇒ 3

(collection-ref "test string" 0)
   ⇒ t

(collection-ref (vector 1 2 3) 3 -1)
   ⇒ -1
```

```
(collection-ref (vector 1 2 3) 3)
   error  &invalid-index-exception
```

For dictionary-like collections, such as a hashtables, `collection-ref` obtains the value in the collection associated with the provided `key`.

```
(let ((table (create-hashtable)))
   (hashtable-put! 'a 1)
   (hashtable-put! 'b 2)
   (hashtable-put! 'c 3)
   (collection-ref table 'b))
   ⇒ 2

(let ((table (create-hashtable)))
   (hashtable-put! 'a 1)
   (hashtable-put! 'b 2)
   (hashtable-put! 'c 3)
   (collection-ref table 'd -1))
   ⇒ -1

(let ((table (create-hashtable)))
   (hashtable-put! 'a 1)
   (hashtable-put! 'b 2)
   (hashtable-put! 'c 3)
   (collection-ref table 'd))
   error  &invalid-index-exception
```

It is sometimes useful to obtain a subset of elements provided by an indexable collection. `collection-slice` provides this functionality.

```
(let ((slice-enumer (collection-slice (iota 50) (range :start 15 :end 30))))
      (enumerable-collect slice-enumer +list-collector+))
   ⇒ (15 16 17 18 19 20 21 22 23 24 25 26 27 28 29)

(enumerable-collect (collection-slice (vector 3 4 5 6) '(2 3))
                    +list-collector+)
   ⇒ (5 6)
```

2.3 extendable

2.3.1 Overview

The extendable protocol specifies the operations expected of a collection that can be dynamically extended beyond its initially allocated size.

2.3.2 API Reference

`collection-extendable?` *object* [generic]

returns a boolean indicating whether or not `object` is extendable.

`collection-extend!` *object value* [generic]

requires `object` support the `extendable` protocol and `value` be an arbitrary value. **returns** `#unspecified` or throws `&unsupported-operation-exception` if `collection-extend!` is not supported.

modifies `object` by adding `value`.

note for dictionary-like collections `value` should be an `association` representing the key and value.

2.3.3 Examples

Bigloo lists and hashtables (as well as a number of other collection types) implement the `extendable` protocol:

```
(collection-extendable? (list 1 2 3))
  ⇒ #t

(collection-extendable? (create-hashtable))
  ⇒ #t
```

Both can be extended:

```
(collection-extend! (list 1 2 3 4) 5)
  ⇒ (5 1 2 3 4)

(let ((table (hashtable :comparator +string-comparator+ (=> "a" 1) (=> "b" 2))))
   (collection-extend! table (=> "c" 3))
   (enumerable-collect table +list-collector+))
  ⇒ (1 2 3)
```

2.4 mutable

2.4.1 Overview

The mutable protocol specifies the operations expected of an mutable collection. A mutable collection is one in which the collection can be modified in place.

2.4.2 API Reference

`collection-mutable?` *object* [generic]

returns a boolean indicating whether or not `object` is mutable

2.4.3 Examples

Not surprisingly, the standard Bigloo lists, vectors, and hashtables are mutable.

```
(collection-mutable? (list))
   ⇒ #t

(collection-mutable? (vector 1 2 3))
   ⇒ #t

(collection-mutable? "test string")
   ⇒ #t
```

2.5 enumerator

2.5.1 Overview

The `enumerator` protocol is an abstraction for enumerating or iterating through the items contained by an object. These items may be pre-existing or generated on demand. Collections are the most common object to provide `enumerators` but other objects, such as interval ranges, can provide enumerators as well.

2.5.2 API Reference

`enumerator?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `enumerator` protocol

`enumerator-move-next!` *object* [generic]

requires `object` support the `enumerator` protocol

returns a boolean indicating whether or not `object` additional items available

modifies `object` such that upon returning `#t` the next item under enumeration is current. Otherwise, `object` remains unmodified.

note `enumerator-move-next!` must be called before `enumerator-current`. If it is not, `&invalid-state-exception` is thrown.

`enumerator-current` *object* [generic]

requires `object` support the `enumerator` protocol

returns the item found at the `enumerators` current position or if `enumerator-move-next!` has not been called before, `&invalid-state-exception` is thrown.

`enumerator-copy` *object* [generic]

requires `object` support the `enumerator` protocol

returns a copy of the `enumerator`.

2.5.3 Examples

You seldom use enumerator directly but, instead, use the procedures and macros provided by `enumerable`. However, if needed you can use the `enumerator` protocol directly.

```
(let ((enumer (enumerable-enumerator (list 1 2 3 4 5))))
   (let loop ((cont (enumerator-move-next! enumer    )))
      (when cont
         (print (enumerator-current enumer))
         (loop (enumerator-move-next! enumer)))))
   ⊣ 1
   ⊣ 2
   ⊣ 3
   ⊣ 4
   ⊣ 5
   ⇒ #unspecified
```

2.6 enumerable

2.6.1 Overview

`enumerable` is a protocol implemented by those objects that support a notion of enumeration or iteration. By providing an `enumerator`, they gain support for mapping, folding, filtering, and more.

2.6.2 API Reference

`enumerable?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `enumerable` protocol

`enumerable-enumerator` *obj* [generic]

requires `object` support the `enumerable` protocol

returns an `enumerator`

`enumerable-for-each` *proc enum1 enum2 . . .* [syntax]

requires `enum`s to either support the `enumerable` or `enumerator` protocol and `proc` be a procedure with an arity compatabile with applying it element-wise to the elements provided by the `enums`.

returns `#unspecified`

note `enumerable-for-each` is applied for the side-effects resulting from applying `proc`.

`enumerable-map` *proc enum1 enum2 . . .* [syntax]

requires `enum`s to either support the `enumerable` or `enumerator` protocol and `proc` be a procedure with an arity compatabile with applying it element-wise to the elements provided by the `enums`.

returns an `enumerator` providing the results of applying `proc`

`enumerable-filter` *predicate enum* [syntax]

requires `enum` to either support the `enumerable` or `enumerator` protocol and `predicate` be a procedure returning a boolean indicating whether or not an element should be retained.

returns an `enumerator` providing the retained elements

`enumerable-take-while` *predicate enum* [syntax]

requires `enum` to either support the `enumerable` or `enumerator` protocol and `predicate` be a procedure returning a boolean indicating whether or not an elements should continue to be consumed.

returns an `enumerator` providing the consumed elements `enumerable-take` *n enum* [syntax]

requires `enum` to either support the `enumerable` or `enumerator` protocol and `n` be an integer representing the max number of items to take

returns an `enumerator` providing upto `n` elements

`enumerable-fold` *proc seed enum1 enum2 . . .* [syntax]

requires `enums` to either support the `enumerable` or `enumerator` protocol, `proc` to be a procedure compatible with applying it to a seed and the element-wise elements provided by the `enums` (it should produce a new seed value), and `seed` should be an appropriate initial seed value for `proc`.

returns the result of folding `proc` over the provided `enums`

note `proc` is applied left to right with the seed resulting from the previous application being used in the next. When no more elements are available, the final seed is returned.

`enumerable-any?` *predicate enum1 enum2 . . .* [syntax]

requires `enums` to either support the `enumerable` or `enumerator` protocol and `predicate` to be a procedure compatible with applying it element-wise to the elements provided by the `enums`, returning a boolean.

returns a boolean indicating whether or not any of the elements are `#t` for `predicate`.

`enumerable-every?` *predicate enum1 enum2 . . .* [syntax]

requires `enums` to either support the `enumerable` or `enumerator` protocol and `predicate` to be a procedure compatible with applying it element-wise to the elements provided by the `enums`, returning a boolean.

returns a boolean indicating whether or not all of the elements are `#t` for `predicate`.

`enumerable-skip-while` *predicate enum1 enum2 . . .* [syntax]

requires `enums` to either support the `enumerable` or `enumerator` protocol and `predicate` to be a procedure compatible with applying it element-wise to the elements provided by the `enums`, returning a boolean indicating whether those elements should skipped.

returns an `enumerator` providing acces to the elements in the original `enums` after those that were skipped according to `predicate`.

`enumerable-skip` *n enum1 enum2 . . .* [syntax]

requires `enums` to either support the `enumerable` or `enumerator` protocol and `n` to be an integer representing the number of elements to skip from `enums`.

returns an `enumerator` providing access to the elements in the original `enums` after those that were skipped.

`enumerable-append` *enum1 enum2 . . .* [syntax]

requires `enums` to either support the `enumerable` or `enumerator` protocol.

returns an `enumerator` providing access to the elements in the original `enums` appended left to right.

`enumerable-collect` *enum collector* [syntax]

requires `enum` to either support the `enumerable` or `enumerator` protocol and `collector` to support the `collector` protocol.

returns a value collected from the given `enum` according to provided `collector`.

note `enumerable-collect` is commonly used to transform the values provided by `enumerators` into concrete collections, although their uses are more flexible than that. For example, to collect all of the values from an `enumerator` into a list you could do the following:

```
(enumerable-collect enumer +list-collector+)
```

See Section 2.7 [collector], page 12, for more information.

2.6.3 Examples

All of the built in Bigloo collection types are `enumerables`.

```
(enumerable? (list 1 2 3))
  ⇒ #t

(enumerable? (vector 1 2 3))
  ⇒ #t

(enumerable? "test string")
  ⇒ #t

(enumerable? (create-hashtable))
  ⇒ #t
```

With `numerable-for-each`, it is possible to iterate over any enumerable. A few examples follow:

```
(let ((count 0))
  (enumerable-for-each (lambda (x) (set! count (+ x count)))
                       (vector 1 2 3))
  count)

  ⇒ 6

(enumerable-for-each (lambda (x) (print x)) (range :start 1 :end 5))

  ⊣ 1
  ⊣ 2
  ⊣ 3
  ⊣ 4
  ⇒ #unspecified
```

It is also possible to map a function over `enumerables`.

```
(enumerable-collect (enumerable-map (lambda (x) (+ x 1))
                       (list 1 2 3 4))
```

```
 +list-collector+)

⇒ (2 3 4 5)
```

For dictionary type collections, the values are mapped over.

```
(let ((table (hashtable (=> 'a 1) (=> 'b 2) (=> 'c 3))))
  (print (enumerable-collect (enumerable-map (lambda (x) (+ x 1)) table)
           +vector-collector+)))
```

Given an appropriate seed an procedure, `enumerable-fold` can be used to reduce an enumerable to a single value.

```
(enumerable-fold (lambda (s v) (+ s v)) 0 (range :start 1 :end 6))

 ⇒ 15
```

Filtering of values is also supported.

```
(enumerable-collect (enumerable-filter odd? (range :start 1 :end 10))
  +list-collector+)

 ⇒ (1 3 5 7 9)
```

It is also possible to consume values while a predicate remains true.

```
(enumerable-collect
  (enumerable-take-while (lambda (x) (< x 5)) (range :start 0 :end 10))
  +list-collector+)

 ⇒ (0 1 2 3 4)
```

Or to test whether any or all values match a give predicate.

```
(enumerable-any? odd? (list 4 6 8 7))

 ⇒ #t

(enumerable-any? odd? (list 4 6 8 10))

 ⇒ #f

(enumerable-every? even? (list 2 4 6 8))

 ⇒ #t

(enumerable-every? even? (list 2 4 6 7))

 ⇒ #f
```

As shown in the above examples, `enumerable-collect` can be used to gather the values of an enumerable into a new collection, but it is more general than that. In fact, it is a general reduction facility. For example, the sum of an `enumerable` can be obtained as follows:

```
(enumerable-collect (range :start 1 :end 6) +sum-collector+)

 ⇒ 15
```

For full details, See Section 2.7 [collector], page 12.

2.7 collector

2.7.1 Overview

The `collector` protocol defines the methods required for a general reduction facility for `enumerables`. Examples include `collection` conversion and various accumulations. It is modeled after the `Collector` interface of Java.

2.7.2 API Reference

`collector?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `collector` protocol

`collector-supplier` *object* [generic]

requires `object` support the `collector` protocol

returns a thunk that when called returns an object for collecting results

`collector-accumulate` *coll supp val* [generic]

requires `coll` support the `collector` protocol, `supp` be an object originally created via the thunk returned by `collector-supplier`, and `val` be an arbitrary item to be accumulated.

returns `supp` updated with `val`

`collector-combine` *coll suppa suppb* [generic]

requires `coll` support the `collector` protocol and `suppa` and `suppb` be objects originally created via the thunk returned by `collector-supplier`,

returns a single entity combining both `suppa` and `suppb`

`collector-finish` *coll supp* [generic]

requires `coll` support the `collector` protocol and `supp` be an object originally created via the thunk returned by `collector-supplier`,

returns a single entity from the accumulated `supp` possibly with a final transformation.

`make-collector` *:supplier :accumulate :combine :finish* [procedure]

requires `:supplier` be a thunk returning an object used for accumulation by the collector, `:accumulate` be a procedure taking the current accumulation object and a val and returning an updated accumulation object, `:combine` be a procedure taking 2 accumulation objects and returning an accumulation object combining the 2, and `:finish` be a procedure taking the final accumulation object and returning an possibly transformed result.

returns a final accumulation object.

`variable` *+list-collector+* [Variable]

`+list-collector+` is a collector that accumulates the items provided by an `enumerator` into a list.

`variable` +*stretchy-vector-collector*+ [Variable]

`+stretchy-vector-collector+` is a collector that accumulates the items provided by an `enumerator` into a stretchy-vector.

`variable` +*vector-collector*+ [Variable]

`+vector-collector+` is a collector that accumulates the items provided by an `enumerator` into a vector.

`variable` +*sum-collector*+ [Variable]

`+sum-collector+` is a collector that sums the items provided by an `enumerator`.

2.7.3 Examples

To create your own version of a list `collector` is as simple as the following:

```
(define +my-list-collector+
   (make-collector :supplier (lambda ()
                               '())
      :accumulate (lambda (supp val)
                    (cons val supp))
      :combine (lambda (a b)
                 (append a b))
      :finish (lambda (x) (reverse x))))
```

To obtain the product of all numbers in an `enumerable`

```
(define +product-collector+
   (make-collector :supplier (lambda () 1)
      :accumulate (lambda (supp val)
                    (* supp val))
      :combine (lambda (a b) (* a b))
      :finish (lambda (x)
                x)))
```

2.8 comparator

2.8.1 Overview

The `comparator` protocol defines those methods required to support comparison and, optionally, total ordering of a specific type. Although not identical, it is very similar to the functionality found in SRFI-128.

2.8.2 API Reference

`comparator?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `comparator` protocol

`comparator-ordered?` *comp* [generic]

requires `comp` implement the `comparator` protocol.

returns a boolean indicating whether or not `comp` supports ordering.

`comparator-hashable?` *comp* [generic]

requires `comp` implement the `comparator` protocol.

returns a boolean indicating whether or not `comp` supports hashing.

`comparator-type?` *comp val* [generic]

requires `comp` implement the `comparator` protocol and `val` be an arbitrary object.

returns a boolean indicating whether or not `val` is of an appropriate type for `comp`

`comparator=?` *comp a b* [generic]

requires `comp` implement the `comparator` protocol and `a` and `b` be of the type supported by `comp`.

returns a boolean indicating whether or not `a` is equal to `b`

note `comparator=?` is supported by all `comparator` instances.

`comparator<?` *comp a b* [generic]

requires `comp` implement the `comparator` protocol and `a` and `b` be of the type supported by `comp`.

returns a boolean indicating whether or not `a` is less than `b`

note `comparator<?` is only supported by `comparator` instances that are ordered.

`comparator>?` *comp a b* [generic]

requires `comp` implement the `comparator` protocol and `a` and `b` be of the type supported by `comp`.

returns a boolean indicating whether or not `a` is greater than `b`

note `comparator>?` is only supported by `comparator` instances that are ordered.

`comparator<=?` *comp a b* [generic]

requires `comp` implement the `comparator` protocol and `a` and `b` be of the type supported by `comp`.

returns a boolean indicating whether or not `a` is less than or equal to `b`

note `comparator<=?` is only supported by `comparator` instances that are ordered.

`comparator>=?` *comp a b* [generic]

requires `comp` implement the `comparator` protocol and `a` and `b` be of the type supported by `comp`.

returns a boolean indicating whether or not `a` is greater than or equal to `b`

note `comparator>=?` is only supported by `comparator` instances that are ordered.

`comparator-hash` *comp val* [generic]

requires `comp` implement the `comparator` protocol and `val` of the type supported by `comp`.

returns an integer hash value

note `comparator-hash` is only supported by `comparator` instances that are hashable.

`make-comparator` *:type? :equal?* [*:less-than?*] [*:hash*] [procedure]

requires `:type?` be a single argument procedure returning a boolean indicating whether or not the argument is of the type supported by the `comparator`, *:equal?* be a 2 argument procedure returning whether or not the arguments are equal, `:less-than?` be a 2 argument procedure returning whether or not the first argument is less than or equal to the second, and `:hash` be a single argument procedure returning an appropriate integer value.

returns a `comparator`

note `:less-than?` and `:hash` are optional but at least one needs to be provided. Obviously, both can also be provided. If `:less-than?` is not provided, the comparator will not be ordered. Similarly, if the `:hash` is not provided, the comparator will not be hashable.

`variable` +*number-comparator*++ [Variable]

`+number-comparator+` is a comparator instance for numbers. It is ordered and hashable.

`variable` +*char-comparator*+ [Variable]

`+char-comparator+` is a comparator instance for characters. It is ordered and hashable. The ordering is case-sensitive.

`variable` +*char-ci-comparator*+ [Variable]

`+char-ci-comparator+` is a comparator instance for characters. It is ordered and hashable. The ordering is case-insensitive.

`variable` +*ucs2-comparator*+ [Variable]

`+ucs2-comparator+` is a comparator instance for unicode characters. It is ordered and hashable. The ordering is case-sensitive.

`variable` +*ucs2-ci-comparator*+ [Variable]

`+ucs2-ci-comparator+` is a comparator instance for unicode characters. It is ordered and hashable. The ordering is case-insensitive.

`variable` +*string-comparator*+ [Variable]

`+string-comparator+` is a comparator instance for strings. It is ordered and hashable. The ordering is case-sensitive.

`variable` +*string-ci-comparator*+ [Variable]

`+string-ci-comparator+` is a comparator instance for strings. It is ordered and hashable. The ordering is case-insensitive.

`variable` +*symbol-comparator*+ [Variable]

`+symbol-comparator+` is a comparator instance for symbols. It is ordered and hashable. The ordering is case-sensitive.

`variable` +*symbol-ci-comparator*+ [Variable]

`+symbol-ci-comparator+` is a comparator instance for symbols. It is ordered and hashable. The ordering is case-insensitive.

`variable` +*ucs2-string-comparator*+ [Variable]

`+ucs2-string-comparator+` is a comparator instance for unicode strings. It is ordered and hashable. The ordering is case-sensitive.

`variable` +*ucs2-string-ci-comparator*+ [Variable]

`+ucs2-string-ci-comparator+` is a comparator instance for unicode strings. It is ordered and hashable. The ordering is case-insensitive.

`variable` +*keyword-comparator*+ [Variable]

`+keyword-comparator+` is a comparator instance for keywords. It is ordered and hashable. The ordering is case-sensitive.

`variable` +*keyword-ci-comparator*+ [Variable]

`+keyword-ci-comparator+` is a comparator instance for keywords. It is ordered and hashable. The ordering is case-insensitive.

2.8.3 Examples

To demonstrate the use of the comparator protocol, we will use the `+number-comparator+` instance.

`+number-comparator+` is both ordered and hashable:

```
(comparator-ordered? +number-comparator+)
  ⇒ #t

(comparator-hashable? +number-comparator+)
  ⇒ #t
```

All comparators must support type checking and equality:

```
(comparator-type? +number-comparator+ 4)
  ⇒ #t

(comparator-type? +number-comparator+ "dog")
  ⇒ #f

(comparator=? +number-comparator+ 4 4)
  ⇒ #t

(comparator=? +number-comparator+ 3 4)
  ⇒ #f
```

Being ordered, we can use the ordered comparision methods with +number-comparator+:

```
(comparator<? +number-comparator+ 4 5)
  ⇒ #t

(comparator>? +number-comparator+ 4.0 5)
  ⇒ #f

(comparator<=? +number-comparator+ 5 5)
  ⇒ #t

(comparator>=? +number-comparator+ 5 6.0)
```

```
    ⇒ #f
```

And being hashable, we can use the hash method:

```
(comparator-hash comp 4)
   ⇒ 4

(comparator-hash comp 5.0)
   ⇒ 5
```

2.9 bag

2.9.1 Overview

The `bag` protocol defines those methods required by a bag implementation. A bag is set-like data structure that can contain multiple copies of an item.

2.9.2 API Reference

`bag?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `bag` protocol

`bag-copy` *bag* [generic]

requires `bag` implement the bag protocol.

returns a shallow copy of `bag`.

`bag-empty?` *bag* [generic]

requires `bag` implement the bag protocol.

returns a boolean indicaing whether or not `bag` contains any items.

`bag-insert!` *bag item* [generic]

requires `bag` implement the bag protocol, and `item` be an arbitrary object.

modifies `bag` so that it contains a copy or an addtional copy, if one already exists, of `item`.

returns `unspecified`

`bag-delete!` *bag item* [generic]

requires `bag` implement the bag protocol, and `item` be an arbitrary object.

modifies `bag` so that it contains one less copy of `item`. If 0 copies of `item` exist it is removed from `bag`.

returns `unspecified`

`bag-contains?` *bag item* [generic]

requires `bag` implement the bag protocol, and `item` be an arbitrary object.

returns a boolean indicating whether or not the `bag` contains `item`.

`bag-count` *bag item* [generic]

requires `bag` implement the bag protocol, and `item` be an arbitrary object.

returns the number of `item`s found in `bag`.

`bag-count-set!` *bag item count* [generic]

requires `bag` implement the bag protocol, `item` be an arbitrary object, and `count` be an integer representing the number of `item`s to include in `bag`

modifies `bag` to contain `count` number of `items` unless `count` is less than or equal to 0 which results in the all `item`s begin removed from `bag`.

returns `#unspecified`

`bag-length` *bag* [generic]

requires `bag` implement the bag protocol

returns the number of items contained in `bag`

2.9.3 Examples

2 different implementations of the bag protocol are currently provided by hoard. One is tree-based, and the other is hash-based. To create a tree-based bag use:

```
(let ((bag (make-sorted-bag :comparator +number-comparator+)))
   (bag? bag))
   ⇒ #t
```

The `:comparator` argument must be an object implementing the comparator protocol for the type of item to be stored in the bag. The comparator must be ordered.

To create a hash-based bag use:

```
(let ((bag (make-hash-bag :comparator +number-comparator+)))
   (bag? bag))
   ⇒ #t
```

The `:comparator` argument, like in the tree-based example, must be an object implementing the comparator protocol for the type of item to be stored in the bag. However, the comparator must be hashable not ordered.

Assuming a bag has been created as above, to insert an item into a bag use:

```
(bag-insert! bag 1)
   ⇒ #unspecified
```

The count of an item can be obtained with:

```
(bag-count bag 1)
   ⇒ 1

(bag-count bag 2)
   ⇒ 0
```

And the count can be explicitly set using:

```
(bag-count-set! bag 1 4)
   ⇒ #unspecified

(bag-count bag 1)
   ⇒ 4
```

Or a single copy deleted with:

```
(bag-delete! bag 1)
   ⇒ #unspecified

(bag-count bag 1)
   ⇒ 3
```

To check to see if a bag is empty:

```
(bag-empty? bag)
   ⇒ #f
```

To check whether an item is a member of a bag:

```
(bag-contains? bag 1)
   ⇒ #t

(bag-contains? bag 3)
   ⇒ #f
```

And finally, to make a copy:

```
(let ((bag2 (bag-copy? bag)))
   (eq? bag bag2))
   ⇒ #f
```

In addition, `sorted-bag` implements the `enumerable` protocol. See Section 2.6.3 [enumerable Examples], page 11.

2.10 set

2.10.1 Overview

The `set` protocol defines those methods required by a set implementation. A set is a collection of objects where each object is unique.

2.10.2 API Reference

`set?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `set` protocol

`set-copy` *set* [generic]

requires `set` implement the set protocol.

returns a shallow copy of `set`.

`set-empty?` *set* [generic]

requires `set` implement the set protocol.

returns a boolean indicaing whether or not `set` contains any items.

`set-insert!` *set item* [generic]

requires `set` implement the set protocol, and `item` be an arbitrary object.

modifies `set` so that it contains a copy or an addtional copy, if one already exists, of `item`.

returns `unspecified`

`set-delete!` *set item* [generic]

requires `set` implement the set protocol, and `item` be an arbitrary object.

modifies `set` so that it contains one less copy of `item`. If 0 copies of `item` exist it is removed from `set`.

returns `unspecified`

`set-contains?` *set item* [generic]

requires `set` implement the set protocol, and `item` be an arbitrary object.

returns a boolean indicating whether or not the `set` contains `item`.

`set-length` *set* [generic]

requires `set` implement the set protocol

returns the number of items contained in `set`

`set-union!` *set . sets* [generic]

requires `set` implement the set protocol and `sets` be an arbitrary number of additional objects implementing the set protocol.

modifies `set` to be the union of all sets provided. **returns** #unspecified

`set-union` *set . sets* [generic]

requires `set` implement the set protocol and `sets` be an arbitrary number of additional objects implementing the set protocol.

returns a new set of the same type as `set` containing the union of all sets provided.

`set-intersect!` *set . sets* [generic]

requires `set` implement the set protocol and `sets` be an arbitrary number of additional objects implementing the set protocol.

modifies `set` to be the intersection of all sets provided. **returns** #unspecified

`set-intersect` *set . sets* [generic]

requires `set` implement the set protocol and `sets` be an arbitrary number of additional objects implementing the set protocol.

returns a new set of the same type as `set` containing the intersection of all sets provided.

`set-difference!` *set . sets* [generic]

requires `set` implement the set protocol and `sets` be an arbitrary number of additional objects implementing the set protocol.

modifies `set` to be the difference of all sets provided. **returns** #unspecified `set-difference` *set . sets* [generic]

requires `set` implement the set protocol and `sets` be an arbitrary number of additional objects implementing the set protocol.

returns a new set of the same type as `set` containing the difference of all sets provided.

2.10.3 Examples

2 different implementations of the set protocol are currently provided by hoard. One is tree-based, and the other is hash-based. To create a tree-based set use:

```
(let ((set (make-sorted-set :comparator +number-comparator+)))
   (set? set))
   ⇒ #t
```

The `:comparator` argument must be an object implementing the comparator protocol for the type of item to be stored in the set. The comparator must be ordered.

To create a hash-based set use:

```
(let ((set (make-hash-set :comparator +number-comparator+)))
   (set? set))
   ⇒ #t
```

The `:comparator` argument, like in the tree-based example, must be an object implementing the comparator protocol for the type of item to be stored in the set. However, the comparator must be hashable not ordered.

Assuming a set has been created, to insert an item into a set use:

```
(set-insert! set 1)
   ⇒ #unspecified
```

Or deleted with:

```
(set-delete! set 1)
   ⇒ #unspecified

(set-contains? set 1)
   ⇒ #f
```

To check to see if a set is empty:

```
(set-empty? set)
   ⇒ #f
```

To check whether an item is a member of a set:

```
(set-contains? set 1)
   ⇒ #t

(set-contains? set 3)
   ⇒ #f
```

To make a copy:

```
(let ((set2 (set-copy? set)))
   (eq? set set2))
   ⇒ #f
```

To non-destructively perform the union, intersection, and difference:

```
(let* ((set1 (sorted-set :comparator +number-comparator+ 1 2))
       (set2 (hash-set :comparator +number-comparator+ 2 3))
       (uset (set-union set1 set2))
       (iset (set-intersect set1 set2))
```

```
    (dset (set-difference set1 set2)))

  (list (enumerable-collect uset +list-collector+))
  (enumerable-collect iset +list-collector+)
  (enumerable-collect dset +list-collector+))
```

2.11 queue

2.11.1 Overview

The `queue` protocol defines the methods required by a queue implementation. A queue is a first-in-first-out sequential data structure.

2.11.2 API Reference

`queue?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `queue` protocol

`queue-copy` *queue* [generic]

requires `queue` implement the queue protocol.

returns a shallow copy of `queue`.

`queue-empty?` *queue* [generic]

requires `queue` implement the queue protocol.

returns a boolean indicating whether or not `queue` contains any items.

`queue-first` *queue* [generic]

requires `queue` implement the queue protocol.

returns returns the first item in `queue`. If `queue` is empty, a `&invalid-state-exception` is thrown.

`queue-enqueue!` *queue item* [generic]

requires `queue` implement the queue protocol and `item` be an arbitrary item

returns returns `unspecified`. If `queue` has a fixed capacity and is full, a `&invalid-state-exception` is thrown.

modifies `queue` by placing `item` on the end.

`queue-dequeue!` *queue* [generic]

requires `queue` implement the queue protocol.

returns returns the first item in `queue`. If `queue` is empty, a `&invalid-state-exception` is thrown.

modifies `queue` by removing the first item.

`queue-length` *queue* [generic]

requires `queue` implement the queue protocol.

returns returns the the number of items in `queue`

`queue-fixed-capacity?` *queue* [generic]

requires `queue` implement the queue protocol.

returns returns a boolean indicating whether or not `queue` has a fixed capacity.

`queue-capacity` *queue* [generic]

requires `queue` implement the queue protocol.

returns returns the capacity (i.e., the number of items that it can hold) of `queue` or `unspecified` if it has no fixed capacity.

2.11.3 Examples

hoard provides linked and contiguous implementations of the `queue` protocol.

```
(queue? (contiguous-queue :capacity 5))
   ⇒ #t
(queue? (linked-queue))
   ⇒ #t
```

You can add items to a queue:

```
(let ((q (linked-queue)))
   (queue-enqueue! q 1)
   (queue-enqueue! q 2)
   (enumerable-collect q +list-collector+))
   ⇒ (1 2)
```

look at the first item:

```
(let ((q (linked-queue)))
   (queue-enqueue! q 1)
   (queue-enqueue! q 2)
   (queue-first q))
   ⇒ 1
```

or remove the first item:

```
(let ((q (linked-queue)))
   (queue-enqueue! q 1)
   (queue-enqueue! q 2)
   (queue-dequeue! q)
   (queue-front q))
   ⇒ 2
```

It is also possible to test to see if a queue is empty:

```
(queue-empty? (linked-queue))
   ⇒ #t

(queue-empty? (linked-queue 1 2 3))
   ⇒ #f
```

A copy of a queue can be useful:

```
(let* ((q1 (linked-queue 1 2 3))
       (q2 (queue-copy q1)))
   (eq? q1 q2))
```

```
   ⇒ #f
```

Some queue implementations have a fixed-capacity:

```
(queue-fixed-capacity? (contiguous-queue :capacity 5))
   ⇒ #t

(queue-fixed-capacity? (linked-queue))
   ⇒ #f

(queue-capacity (contiguous-queue :capacity 5))
   ⇒ 5

(queue-capacity (linked-queue))
   ⇒ #unspecified
```

2.12 stack

2.12.1 Overview

The `stack` protocol defines the methods required by a stack implementation. A stack is a last-in-first-out sequential data structure.

2.12.2 API Reference

`stack?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `stack` protocol

`stack-copy` *stack* [generic]

requires `stack` implement the stack protocol.

returns a shallow copy of `stack`.

`stack-empty?` *stack* [generic]

requires `stack` implement the stack protocol.

returns a boolean indicating whether or not `stack` contains any items.

`stack-top` *stack* [generic]

requires `stack` implement the stack protocol.

returns returns the top item on `stack`. If `stack` is empty, a `&invalid-state-exception` is thrown.

`stack-push!` *stack item* [generic]

requires `stack` implement the stack protocol and `item` be an arbitrary item

returns returns `unspecified`. If `stack` has a fixed capacity and is full, a `&invalid-state-exception` is thrown.

modifies `stack` by placing `item` on the top.

`stack-pop!` *stack* [generic]

requires `stack` implement the stack protocol.

returns returns the top item on `stack`. If `stack` is empty, a `&invalid-state-exception` is thrown.

modifies `stack` by removing the top item.

`stack-length` *stack* [generic]

requires `stack` implement the stack protocol.

returns returns the the number of items on `stack`

`stack-fixed-capacity?` *stack* [generic]

requires `stack` implement the stack protocol.

returns returns a boolean indicating whether or not `stack` has a fixed capacity.

`stack-capacity` *stack* [generic]

requires `stack` implement the stack protocol.

returns returns the capacity (i.e., the number of items that it can hold) of `stack` or `unspecified` if it has no fixed capacity.

2.12.3 Examples

hoard provides linked and contiguous implementations of the `stack` protocol.

```
(stack? (contiguous-stack :capacity 5))
   ⇒ #t
(stack? (linked-stack))
   ⇒ #t
```

You can push items on a stack:

```
(let ((s (linked-stack)))
   (stack-push! s 1)
   (stack-push! s 2)
   (enumerable-collect s +list-collector+))
   ⇒ (2 1)
```

look at the top item:

```
(let ((s (linked-stack)))
   (stack-push! s 1)
   (stack-push! s 2)
   (stack-top s))
   ⇒ 2
```

or remove the top item:

```
(let ((s (linked-stack)))
   (stack-push! q 1)
   (stack-push! q 2)
   (stack-pop! q)
   (stack-top q))
   ⇒ 1
```

It is also possible to test to see if a stack is empty:

```
(stack-empty? (linked-stack))
   ⇒ #t

(stack-empty? (linked-stack 1 2 3))
   ⇒ #f
```

A copy of a stack can be useful:

```
(let* ((q1 (linked-stack 1 2 3))
       (q2 (stack-copy q1)))
   (eq? q1 q2))
   ⇒ #f
```

Some stack implementations have a fixed-capacity:

```
(stack-fixed-capacity? (contiguous-stack :capacity 5))
   ⇒ #t

(stack-fixed-capacity? (linked-stack))
   ⇒ #f

(stack-capacity (contiguous-stack :capacity 5))
   ⇒ 5

(stack-capacity (linked-stack))
   ⇒ #unspecified
```

2.13 priority-queue

2.13.1 Overview

The `priority-queue` protocol defines the methods required by a priority-queue implementation. A priority-queue is a sequential data structure where items are ordered by priority.

2.13.2 API Reference

`priority-queue?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `priority-queue` protocol

`priority-queue-copy` *priority-queue* [generic]

requires `priority-queue` implement the priority-queue protocol.

returns a shallow copy of `priority-queue`.

`priority-queue-empty?` *priority-queue* [generic]

requires `priority-queue` implement the priority-queue protocol.

returns a boolean indicating whether or not `priority-queue` contains any items.

`priority-queue-first` *priority-queue* [generic]

requires `priority-queue` implement the priority-queue protocol.

returns returns the first item in `priority-queue`. If `priority-queue` is empty, a `&invalid-state-exception` is thrown.

`priority-queue-enqueue!` *priority-queue item* [generic]

requires `priority-queue` implement the priority-queue protocol and `item` be an arbitrary item

returns returns `unspecified`. If `priority-queue` has a fixed capacity and is full, a `&invalid-state-exception` is thrown.

modifies `priority-queue` by placing `item` in the queue according to its priority.

`priority-queue-dequeue!` *priority-queue* [generic]

requires `priority-queue` implement the priority-queue protocol.

returns returns the first item in `priority-queue`. If `priority-queue` is empty, a `&invalid-state-exception` is thrown.

modifies `priority-queue` by removing the first item.

`priority-queue-length` *priority-queue* [generic]

requires `priority-queue` implement the priority-queue protocol.

returns returns the the number of items in `priority-queue`

`priority-queue-fixed-capacity?` *priority-queue* [generic]

requires `priority-queue` implement the priority-queue protocol.

returns returns a boolean indicating whether or not `priority-queue` has a fixed capacity.

`priority-queue-capacity` *priority-queue* [generic]

requires `priority-queue` implement the priority-queue protocol.

returns returns the capacity (i.e., the number of items that it can hold) of `priority-queue` or `unspecified` if it has no fixed capacity.

2.13.3 Examples

hoard provides linked and contiguous implementations of the `priority-queue` protocol.

```
(priority-queue? (binary-heap :capacity 5 :comparator +number-comparator+))
   ⇒ #t
(priority-queue? (pairing-heap :comparator +number-comparator+))
   ⇒ #t
```

You can add items to a priority-queue:

```
(let ((q (pairing-heap :comparator +number-comparator+)))
   (priority-queue-enqueue! q 1)
   (priority-queue-enqueue! q 2)
   (enumerable-collect q +list-collector+))
   ⇒ (1 2)
```

look at the first item:

```
(let ((q (pairing-heap :comparator +number-comparator+)))
   (priority-queue-enqueue! q 2)
```

```
    (priority-queue-enqueue! q 1)
    (priority-queue-first q))
    ⇒ 1
```

or remove the first item:

```
(let ((q (pairing-heap :comparator +number-comparator+)))
    (priority-queue-enqueue! q 1)
    (priority-queue-enqueue! q 2)
    (priority-queue-dequeue! q)
    (priority-queue-front q))
    ⇒ 2
```

It is also possible to test to see if a priority-queue is empty:

```
(priority-queue-empty? (pairing-heap :comparator +number-comparator+))
    ⇒ #t

(priority-queue-empty? (pairing-heap :comparator +number-comparator+))
    ⇒ #f
```

A copy of a priority-queue can be useful:

```
(let* ((q1 (pairing-heap :comparator +number-comparator+ 1 2 3))
       (q2 (priority-queue-copy q1)))
    (eq? q1 q2))
    ⇒ #f
```

Some priority-queue implementations have a fixed-capacity:

```
(priority-queue-fixed-capacity? (binary-heap :capacity 5 :comparator +number-comparator+))
    ⇒ #t

(priority-queue-fixed-capacity? (pairing-heap :comparator +number-comparator+))
    ⇒ #f

(priority-queue-capacity (binary-heap :capacity 5 :comparator +number-comparator+))
    ⇒ 5

(priority-queue-capacity  (pairing-heap :comparator +number-comparator+))
    ⇒ #unspecified
```

2.14 dictionary

2.14.1 Overview

The `dictionary` protocol defines the methods required by a dictionary implementation. A dictionary is a data structure maintaining associations between keys and values.

2.14.2 API Reference

`dictionary?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `dictionary` protocol

`dictionary-copy` *dict* [generic]

requires `dict` implement the dictionary protocol.

returns a shallow copy of `dict`

`dictionary-empty?` *dict* [generic]

requires `dict` implement the dictionary protocol.

returns a boolean indicating whether or not the dictionary contains any associations.

`dictionary-get` *dict key* [generic]

requires `dict` implement the dictionary protocol and `key` be an arbitrary object

returns the value associated with `key` or `#f` if no such association exists.

`dictionary-put!` *dict key value* [generic]

requires `dict` implement the dictionary protocol and `key` and `value` be arbitrary objects

modifies `dict` so that it contains the association of `key` to `value`. If an association with `key` already exists it is replaced, and if not, a new association is created.

returns `#unspecified`

`dictionary-length` *dict* [generic]

requires `dict` implement the dictionary protocol.

returns the number of items in `dict`.

`dictionary-remove!` *dict key* [generic]

requires `dict` implement the dictionary protocol and `key` be an arbitrary object.

modifies `dict` by removing the association with the key `key`.

returns `#unspecified`

`dictionary-contains?` *dict key* [generic]

requires `dict` implement the dictionary protocol and `key` be an arbitrary objects.

returns a boolean indicating whether or not `dict` contains an association with the key `key`.

`dictionary-update!` *dict key value exist-fun* [generic]

requires `dict` implement the dictionary protocol, `key` and `value` be arbitrary objects, and `exist-fun` which is a procedure excepting a single value and returning an updated value when an association with a key `key` already exists.

modifies `dict` such that if the dictionary doesn't currently contain an association with a key `key`, it contains an association with key `key` and value `value`, or if an existing association exists updates it so that has the value obtained by applying `exist-fun` to its value.

returns `unspecified`

`dictionary-enumerator` *dict* [generic]

requires `dict` implement the dictionary protocol.

returns an object implementing the `dictionary-enumerator` protocol allowing the enumeration of the associations contained in `dict`.

2.14.3 Examples

Bigloo's native hashtable and the hoard provided sorted-dictionary implement the `dictionary` protocol.

```
(dictionary? (create-hashtable))
   ⇒#t

(dictionary? (sorted-dictionary :comparator +number-comparator+))
```

Checking whether a dictionary is empty:

```
(dictionary-empty? (create-hashtable))
   ⇒ #t

(dictionary-empty? (hashtable (=> "A" 1)))
   ⇒ #f
```

Associations can be added to a dictionary:

```
(let ((dict (create-hashtable)))
   (dictionary-put! dict "a" 1)
   (dictionary-put! dict "b" 2)
   (map (lambda (kv) (cons (=>key kv) (=>value kv)))
      (dictionary-enumerable-collect dict +list-collector+)))
   ⇒ (("a" . 1) ("b" . 2))
```

And removed (assuming the insertions above):

```
(dictionary-remove! dict "a")
(map (lambda (kv) (cons (=>key kv) (=>value kv)))
   (dictionary-enumerable-collect dict +list-collector+))
   ⇒ (("b" . 2))
```

The value associated with a given key is easily obtained:

```
(let ((dict (hashtable (=> "a" 1) (=> "b" 2) (=> "c" 3))))
   (dictionary-get dict "b")
   ⇒ 2
```

The `dictionary-update!` method can be used to update an existing value or insert a new value if an existing association does not exist:

```
(let ((dict (hashtable (=> "a" 1) (=> "b" 2) (=> "c" 3))))
   (dictionary-update! dict "b" 0 (lambda (x) (+ x 1)))
   (dictionary-update! dict "d" 0 (lambda (x) (+ x 1)))
   (dictionary-get dict "b")
   ⇒ 3
   (dictionary-get dict "d")
   ⇒ 0
```

To obtain the number of associations in a dictionary, call `dictionary-length`:

```
(let ((dict (hashtable (=> "a" 1) (=> "b" 2) (=> "c" 3))))
   (dictionary-length dict)
   ⇒ 3
```

Querying whether an association with a given key is accomplished with `dictionary-contains?`:

```
(let ((dict (hashtable (=> "a" 1) (=> "b" 2) (=> "c" 3))))
   (dictionary-contains? dict "a")
   ⇒ #t

   (dictionary-contains? dict "d")
   ⇒ #f
```

Copying a dictionary is accomplished with `dictionary-copy`:

```
(let* ((dict1 (hashtable (=> "a" 1) (=> "b" 2) (=> "c" 3)))
       (dict2 (dictionary-copy dict1)))
    (eq? dict1 dict2)
    ⇒ #f
    (and (dictionary-contains? dict1 "a")
       (dictionary-contains? dict2 "a"))
    ⇒ #t
```

And last, but not least, you can obtain a `dictionary-enumerator` to enumerate the elements of the dictionary:

```
(let ((dict (hashtable (=> "a" 1) (=> "b" 2) (=> "c" 3))))
   (dictionary-enumerator? (dictionary-enumerator dict)))
   ⇒ #t
```

2.15 dictionary-enumerator

2.15.1 Overview

The `dictionary-enumerator` protocol is an abstraction for enumerating or iterating through the items contained by an object supporting the `dictionary` protocol. These items may be pre-existing or generated on demand. It is very similar to the `enumerator` protocol but provides access to both the keys and values of the dictionary.

2.15.2 API Reference

`dictionary-enumerator?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `dictionary-enumerator` protocol

`dictionary-enumerator-move-next!` *object* [generic]

requires `object` support the `dictionary-enumerator` protocol

returns a boolean indicating whether or not `object` has additional items available

modifies `object` such that upon returning `#t` the next item under enumeration is current. Otherwise, `object` remains unmodified.

note `dictionary-enumerator-move-next!` must be called before `dictionary-enumerator-current`. If it is not, `&invalid-state-exception` is thrown.

`dictionary-enumerator-current` *object* [generic]

requires `object` support the `dictionary-enumerator` protocol

returns the association found at the `dictionary-enumerators` current position or if `dictionary-enumerator-move-next!` has not been called before, `&invalid-state-exception` is thrown.

`dictionary-enumerator-key` *object* [generic]

requires `object` support the `dictionary-enumerator` protocol

returns the key found at the `dictionary-enumerators` current position or if `dictionary-enumerator-move-next!` has not been called before, `&invalid-state-exception` is thrown.

`dictionary-enumerator-value` *object* [generic]

requires `object` support the `dictionary-enumerator` protocol

returns the value found at the `dictionary-enumerators` current position or if `dictionary-enumerator-move-next!` has not been called before, `&invalid-state-exception` is thrown.

`dictionary-enumerator-copy` *object* [generic]

requires `object` support the `dictionary-enumerator` protocol

returns a copy of the `dictionary-enumerator`.

2.15.3 Examples

You seldom use dictionary-enumerator directly but, instead, use the procedures and macros provided by `dictionary-enumerable`. However, if needed you can use the `dictionary-enumerator` protocol directly.

```
(let ((enumer (enumerable-dictionary-enumerator (hashtable :comparator +number-comparator+
                                                  (=> 'a 1) (=> 'b 2) (=> 'c 3))))
   (let loop ((cont (dictionary-enumerator-move-next! enumer    )))
      (when cont
         (print (dictionary-enumerator-value enumer))
         (loop (dictionary-enumerator-move-next! enumer)))))
   ⊣ 1
   ⊣ 2
   ⊣ 3
   ⇒ #unspecified
```

2.16 dictionary-enumerable

2.16.1 Overview

`dictionary-enumerable` is a protocol implemented by those dictionary objects that support a notion of enumeration or iteration. By providing an `enumerator`, they gain support for mapping, folding, filtering, and more.

2.16.2 API Reference

`dictionary-enumerable?` *object* [generic]

returns a boolean indicating whether or not `object` supports the `dictionary-enumerable` protocol

`dictionary-enumerable-enumerator` *obj* [generic]

requires `object` support the `dictionary-enumerable` protocol

returns an `dictionary-enumerator`

`dictionary-enumerable-for-each` *proc enum* [syntax]

requires `enum` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocol and `proc` be a procedure that accepts two arguments, a key and value.

returns `#unspecified`

note `dictionary-enumerable-for-each` is applied for the side-effects resulting from applying `proc`.

`dictionary-enumerable-map` *proc enum* [syntax]

requires `enum` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocol and `proc` be a procedure taking the key and value of an association and returning a new association.

returns an `enumerator` providing the results of applying `proc`

`dictionary-enumerable-filter` *predicate enum* [syntax]

requires `enum` to either support the `dictionary-enumerable` or `enumerator` protocol and `predicate` be a procedure accepting a key and a value and returning a boolean indicating whether or not the association should be retained.

returns an `dictionary-enumerator` providing the retained elements

`dictionary-enumerable-fold` *proc seed enum* [syntax]

requires `enum` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocols, `proc` to be a procedure compatible with applying it to a seed and a key and value provided by `enum` (it should produce a new seed value), and `seed` should be an appropriate initial seed value for `proc`.

returns the result of folding `proc` over the provided `enum`

note `proc` is applied left to right with the seed resulting from the previous application being used in the next. When no more elements are available, the final seed is returned.

`dictionary-enumerable-any?` *predicate enum* [syntax]

requires `enum` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocols and `predicate` to be a procedure accepting a key and a value and returning a boolean.

returns a boolean indicating whether or not any of the associations are `#t` for `predicate`.

`dictionary-enumerable-every?` *predicate enum* [syntax]

requires `enum` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocols and `predicate` to be a procedure accepting a key and a value and returning a boolean.

returns a boolean indicating whether or not all of the associations are `#t` for `predicate`.

`dictionary-enumerable-append` *enum1 enum2 . . .* [syntax]

requires `enums` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocols.

returns an `dictionary-enumerator` providing access to the elements in the original `enums` appended left to right.

`dictionary-enumerable-collect` *enum collector* [syntax]

requires `enum` to either support the `dictionary-enumerable` or `dictionary-enumerator` protocol and `collector` to support the `collector` protocol.

returns a value collected from the given `enum` according to the provided `collector`.

note `dictionary-enumerable-collect` is commonly used to transform the keys and values provided by `dictionary-enumerator`s into concrete collections, although their uses are more flexible than that. For example, to collect all of the values from an `dictionary-enumerator` into a list you could do the following:

```
(dictionary-enumerable-collect enumer +list-collector+)
```

See Section 2.7 [collector], page 12, for more information.

2.16.3 Examples

Bigloo's hashtables and hoard's sorted-dictionary are `dictionary-enumerables`.

```
(dictionary-enumerable? (create-hashtable))
  ⇒ #t

(dictionary-enumerable? (sorted-dictionary :comparator +number-comparator+))
```

With `numerable-for-each`, it is possible to iterate over any dictionary-enumerable. A few examples follow:

```
(let ((count 0))
  (dictionary-enumerable-for-each (lambda (k v) (set! count (+ v count)))
                         (hashtable (=> 'a 1) (=> 'b 2) (=> 'c 3)))
  count)

  ⇒ 6
```

It is also possible to map a function over `dictionary-enumerable`s.

```
(map =>value (dictionary-enumerable-collect (dictionary-enumerable-map (lambda (k v) (=> k (+ v 1))) (has
collector+))
  ⇒ (2 3 4)
```

Given an appropriate seed an procedure, `dictionary-enumerable-fold` can be used to reduce an dictionary-enumerable to a single value.

```
(dictionary-enumerable-fold (lambda (s k v) (+ s v)) 0
                              (hashtable (=> 'a 1) (=> 'b 2)
                                (=> 'c 3) (=> 'd 4) (=> 'e 5)))

  ⇒ 15
```

Filtering of values is also supported.

```
(map =>value (dictionary-enumerable-collect (dictionary-enumerable-filter (lambda (k v) (odd? v))
                                                (hashtable (=> 'a 1) (=> 'b 2)
                                                  (=> 'c 3) (=> 'd 4) (=> 'e 5)))
                +list-collector+))
  ⇒ (2 3 4 5)
```

Or to test whether any or all associations match a give predicate.

```
(dictionary-enumerable-any? (lambda (k v) (odd? v)) (hashtable (=> 'a 1) (=> 'b 2)
                                   (=> 'c 3) (=> 'd 4) (=> 'e 5)))
  ⇒ #t

(dictionary-enumerable-any? (lambda (k v) (odd? v)) (hashtable (=> 'a 2) (=> 'b 4)
                                   (=> 'c 6) (=> 'd 8) (=> 'e 10)))
  ⇒ #f

(dictionary-enumerable-every? (lambda (k v) (even? v)) (hashtable (=> 'a 2) (=> 'b 4)
                                   (=> 'c 6) (=> 'd 8) (=> 'e 10)))
  ⇒ #t

(dictionary-enumerable-every? (lambda (k v) (even? v)) (hashtable (=> 'a 2) (=> 'b 3)
                                   (=> 'c 6) (=> 'd 8) (=> 'e 10)))
  ⇒ #f
```

As shown in the above examples, `dictionary-enumerable-collect` can be used to gather the values of an dictionary-enumerable into a new collection, but it is more general than that. In fact, it is a general reduction facility. For full details, See Section 2.7 [collector], page 12.

3 Implementations and Supporting Data Types

Each of the collection implementations and supporting data types are documented here.

3.1 association

3.1.1 Overview

`association` is a simple data structure containing a key and a value. It is used by the `dictionary` protocol.

3.1.2 API Reference

`association?` *object* [procedure]

returns a boolean indicating whether or not `object` is an association.

`=>` *key value* [procedure]

requires `key` and `value` be arbitrary objects.

returns a new association with key `key` and value `value`.

`=>key` *assoc* [procedure]

requires `assoc` be an association

returns the key of the association.

`=>value` *assoc* [procedure]

requires `assoc` be an association

returns the value of the association.

`pair->association` *kv-pair* [procedure]

requires `kv-pair` be a pair with the car the key and the cdr the value

returns an association with key the car of `kv-pair` and the value the cdr of `kv-pair`.

3.1.3 Examples

`association?` tests whether an object is an `association` or not:

```
(association? '(key . value))
  ⇒ #f

(association? (=> 'key 'value))
  ⇒ #t
```

A new association is constructucted with `=>`:

```
(let ((assoc (=> 'key 'value)))
   (=>key assoc))
   ⇒ 'key
```

A pair consisting of a key and value can also be converted to an `association`:

```
(let ((assoc (pair->association '(key . value))))
   (=>key assoc))
```

```
⇒ 'key
```

The key of an association is obtained with `=>key`:

```
(=>key (=> "key" 1))
  ⇒ "key"
```

The value of an association is obtained with `=>value`:

```
(=>value (=> "key" 1))
  ⇒ 1
```

3.2 range

3.2.1 Overview

`range` is a data structure representing an integer interval having a start and an end (exclusive) possesing values between the two at a given step. It supports methods for iterating and mapping over the specified interval.

3.2.2 API Reference

`range` implements the `enumerable` protocol. See Section 2.6 [enumerable], page 9.

`range?` *object* [procedure]

returns a boolean indicating whether or not `object` is a range.

`range` [*:start 0*] *:end* [*:step 1*] [procedure]

requires `:start`, `:end`, and `step` be integers with `:start <= :end` when `step` is positive or `:start >= :end` when `step` is negative. `:step` can not be 0. `:start` and `:step` are optional and default to 0 and 1, respectively.

returns a range with the specified `start`, `end`, and `step`.

`range-for-each` *proc range* [procedure]

requires `proc` be a procedure compatible with applying it element-wise to the values represented by `range` and `range` be a valid instance of the range data type.

returns #unspecified.

note `range-for-each` is applied for the side-effects resulting from applying `proc`.

`range-map` *proc range* [procedure]

requires `proc` be a procedure compatible with applying it element-wise to the values represented by `range` and `range` be a valid instance of the range data type.

returns a list providing the results of applying `proc` to each integer in `range`

3.2.3 Examples

Creating a `range` is straight forward:

```
(range :end 5) ; [0...4]
(range :start 2 :end 5) ; [2,3,4]
(range :start 2 :end 10 :step 2) ; [2,4,6,8]
```

Once you have a `range`, you can iterate and map over it:

```
(range-for-each print (range :end 5))
  ⊣ 0
  ⊣ 1
  ⊣ 2
  ⊣ 3
  ⊣ 4
  ⇒ #unspecified

(range-map (lambda (x) (+ x 1)) (range :end 5))
  ⇒ (1 2 3 4 5)
```

In addition, `range` implements the `enumerable` protocol. See Section 2.6.3 [enumerable Examples], page 11.

3.3 sorted-bag

3.3.1 Overview

`sorted-bag` is an implementation of bag. It is based on a balanced-binary tree and keeps the elements of the bag sorted.

3.3.2 API Reference

`sorted-bag` implements the `bag`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section 2.9 [bag], page 18. See Section 2.1 [collection], page 3. See Section 2.3 [extendable], page 6. See Section 2.4 [mutable], page 7. See Section 2.6 [enumerable], page 9.

`sorted-bag?` *object* [procedure]

returns a boolean indicating whether or not `object` is a sorted-bag.

`make-sorted-bag` *:comparator* [procedure]

requires `:comparator` be an object implementing the `comparator` protocol. `comparator` must be ordered.

returns an instance of `sorted-bag`

`sorted-bag` *:comparator . items* [procedure]

requires `:comparator` be an object implementing the `comparator` protocol and `items` be a list of objects for which `comparator` is applicable. `comparator` must be ordered.

returns an instance of `sorted-bag`

modifies the returned `sorted-bag` so that it contains `items`

`sorted-bag-copy` *bag* [procedure]

requires `bag` be a sorted-bag

returns a shallow copy of `bag`.

`sorted-bag-empty?` *bag* [procedure]

requires `bag` be a sorted-bag

returns a boolean indicaing whether or not `bag` contains any items.

`sorted-bag-insert!` *bag item* [procedure]

requires `bag` be a sorted-bag, and `item` be an arbitrary object supported by the comparator used to create `bag`.

modifies `bag` so that it contains a copy or an addtional copy, if one already exists, of `item`.

returns `unspecified`

`sorted-bag-delete!` *bag item* [procedure]

requires `bag` be a sorted-bag, and `item` be an arbitrary object supported by the comparator used to create `bag`.

modifies `bag` so that it contains one less copy of `item`. If 0 copies of `item` exist it is removed from `bag`.

returns `unspecified`

`sorted-bag-contains?` *bag item* [procedure]

requires `bag` be a sorted-bag, and `item` be an arbitrary object supported by the comparator used to create `bag`

returns a boolean indicating whether or not the `bag` contains `item`.

`sorted-bag-count` *bag item* [procedure]

requires `bag` be a sorted-bag, and `item` be an arbitrary object supported by the comparator used to create `bag`

returns the number of `item`s found in `bag`.

`sorted-bag-count-set!` *bag item count* [procedure]

requires `bag` be a sorted-bag, `item` be `item` be an arbitrary object supported by the comparator used to create `bag`, and `count` be an integer representing the number of `item`s to include in `bag`

modifies `bag` to contain `count` number of `items` unless `count` is less than or equal to 0 which results in the all `item`s begin removed from `bag`.

returns `#unspecified`

`bag-length` *bag* [procedure]

requires `bag` be a sorted-bag

returns the number of items contained in `bag`

3.3.3 Examples

2 procedures are provided for creating a `sorted-bag`. The first creates an empty bag and the other populates the bag with the items passed to it:

```
(enumerable-collect (make-sorted-bag :comparator +number-comparator+)
```

```
   +list-collector+)
  ⇒ ()

(enumerable-collect (sorted-bag :comparator +number-comparator+ 1 1 1 3)
   +list-collector+)

  ⇒ (1 1 1 3)
```

Assuming a sorted-bag has been created as above, to insert an item into a sorted-bag use:

```
(sorted-bag-insert! bag 1)
   ⇒ #unspecified
```

The count of an item can be obtained with:

```
(sorted-bag-count bag 1)
   ⇒ 1

(sorted-bag-count bag 2)
   ⇒ 0
```

And the count can be explicitly set using:

```
(sorted-bag-count-set! bag 1 4)
   ⇒ #unspecified

(sorted-bag-count bag 1)
   ⇒ 4
```

Or a single copy deleted with:

```
(sorted-bag-delete! bag 1)
   ⇒ #unspecified

(sorted-bag-count bag 1)
   ⇒ 3
```

To check to see if a sorted-bag is empty:

```
(sorted-bag-empty? bag)
   ⇒ #f
```

To check whether an item is a member of a sorted-bag:

```
(sorted-bag-contains? bag 1)
   ⇒ #t

(sorted-bag-contains? bag 3)
   ⇒ #f
```

And finally, to make a copy:

```
(let ((bag2 (sorted-bag-copy? bag)))
   (eq? bag bag2))
   ⇒ #f
```

`sorted-bag` also implements the `bag`, `collection`, `mutable`, and `enumerable` protocols. See Section 2.9.3 [bag Examples], page 19. See Section 2.1.3 [collection Examples], page 4. See Section 2.3.3 [extendable Examples], page 7, See Section 2.4.3 [mutable Examples], page 7. See Section 2.6.3 [enumerable Examples], page 11.

3.4 hash-bag

3.4.1 Overview

`hash-bag` is an implementation of bag. As its name would imply, it is a hashtable-based implementation.

3.4.2 API Reference

`hash-bag` implements the `bag`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section 2.9 [bag], page 18. See Section 2.1 [collection], page 3. See Section 2.3 [extendable], page 6. See Section 2.4 [mutable], page 7. See Section 2.6 [enumerable], page 9.

`hash-bag?` *object* [procedure]
: **returns** a boolean indicating whether or not `object` is a hash-bag.

`make-hash-bag` [*:comparator*] [procedure]
: **requires** `:comparator` be an object implementing the `comparator` protocol. `comparator` must be hashable. If it is not provided the default hashtable equality and hash functions are used.

  returns an instance of `hash-bag`

`hash-bag` [*:comparator*] *. items* [procedure]
: **requires** `:comparator` be an object implementing the `comparator` protocol and `items` be a list of objects for which `comparator` is applicable. `comparator` must be hashable. If it is not provided the default hashtable equality and hash functions are used.

  returns an instance of `hash-bag`

  modifies the returned `hash-bag` so that it contains `items`

`hash-bag-copy` *bag* [procedure]
: **requires** `bag` be a hash-bag

  returns a shallow copy of `bag`.

`hash-bag-empty?` *bag* [procedure]
: **requires** `bag` be a hash-bag

  returns a boolean indicaing whether or not `bag` contains any items.

`hash-bag-insert!` *bag item* [procedure]
: **requires** `bag` be a hash-bag, and `item` be an arbitrary object supported by the comparator used to create `bag`.

  modifies `bag` so that it contains a copy or an addtional copy, if one already exists, of `item`.

  returns `unspecified`

`hash-bag-delete!` *bag item* [procedure]

requires `bag` be a hash-bag, and `item` be an arbitrary object supported by the comparator used to create `bag`.

modifies `bag` so that it contains one less copy of `item`. If 0 copies of `item` exist it is removed from `bag`.

returns `unspecified`

`hash-bag-contains?` *bag item* [procedure]

requires `bag` be a hash-bag, and `item` be an arbitrary object supported by the comparator used to create `bag`

returns a boolean indicating whether or not the `bag` contains `item`.

`hash-bag-count` *bag item* [procedure]

requires `bag` be a hash-bag, and `item` be an arbitrary object supported by the comparator used to create `bag`

returns the number of `item`s found in `bag`.

`hash-bag-count-set!` *bag item count* [procedure]

requires `bag` be a hash-bag, `item` be `item` be an arbitrary object supported by the comparator used to create `bag`, and `count` be an integer representing the number of `item`s to include in `bag`

modifies `bag` to contain `count` number of `items` unless `count` is less than or equal to 0 which results in the all `item`s begin removed from `bag`.

returns `#unspecified`

`bag-length` *bag* [procedure]

requires `bag` be a hash-bag

returns the number of items contained in `bag`

3.4.3 Examples

2 procedures are provided for creating a `hash-bag`. The first creates an empty bag and the other populates the bag with the items passed to it:

```
(enumerable-collect (make-hash-bag :comparator +number-comparator+)
   +list-collector+)
  ⇒ ()

(enumerable-collect (hash-bag :comparator +number-comparator+ 1 1 1 3)
   +list-collector+)

  ⇒ (1 1 1 3)
```

Assuming a hash-bag has been created as above, to insert an item into a hash-bag use:

```
(hash-bag-insert! bag 1)
```

```
    ⇒ #unspecified
```

The count of an item can be obtained with:

```
(hash-bag-count bag 1)
   ⇒ 1

(hash-bag-count bag 2)
   ⇒ 0
```

And the count can be explicitly set using:

```
(hash-bag-count-set! bag 1 4)
   ⇒ #unspecified

(hash-bag-count bag 1)
   ⇒ 4
```

Or a single copy deleted with:

```
(hash-bag-delete! bag 1)
   ⇒ #unspecified

(hash-bag-count bag 1)
   ⇒ 3
```

To check to see if a hash-bag is empty:

```
(hash-bag-empty? bag)
   ⇒ #f
```

To check whether an item is a member of a hash-bag:

```
(hash-bag-contains? bag 1)
   ⇒ #t

(hash-bag-contains? bag 3)
   ⇒ #f
```

And finally, to make a copy:

```
(let ((bag2 (hash-bag-copy? bag)))
   (eq? bag bag2))
   ⇒ #f
```

`hash-bag` also implements the `bag`, `collection`, `mutable`, and `enumerable` protocols. See Section 2.9.3 [bag Examples], page 19. See Section 2.1.3 [collection Examples], page 4. See Section 2.3.3 [extendable Examples], page 7, See Section 2.4.3 [mutable Examples], page 7. See Section 2.6.3 [enumerable Examples], page 11.

3.5 sorted-set

3.5.1 Overview

`sorted-set` is an implementation of set. It is based on a balanced-binary tree and keeps the elements of the set sorted.

3.5.2 API Reference

`sorted-set` implements the `set`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section 2.10 [set], page 20. See Section 2.1 [collection], page 3. See Section 2.3 [extendable], page 6. See Section 2.4 [mutable], page 7. See Section 2.6 [enumerable], page 9.

`sorted-set?` *object* [procedure]

returns a boolean indicating whether or not `object` is a sorted-set.

`make-sorted-set` *:comparator* [procedure]

requires `:comparator` be an object implementing the `comparator` protocol. `comparator` must be ordered.

returns an instance of `sorted-set`

`sorted-set` *:comparator . items* [procedure]

requires `:comparator` be an object implementing the `comparator` protocol and `items` be a list of objects for which `comparator` is applicable. `comparator` must be ordered.

returns an instance of `sorted-set`

modifies the returned `sorted-set` so that it contains `items`

`sorted-set-copy` *set* [procedure]

requires `set` be a sorted-set

returns a shallow copy of `set`.

`sorted-set-empty?` *set* [procedure]

requires `set` be a sorted-set

returns a boolean indicaing whether or not `set` contains any items.

`sorted-set-insert!` *set item* [procedure]

requires `set` be a sorted-set, and `item` be an arbitrary object supported by the comparator used to create `set`.

modifies `set` so that it contains a copy or an addtional copy, if one already exists, of `item`.

returns `unspecified`

`sorted-set-delete!` *set item* [procedure]

requires `set` be a sorted-set, and `item` be an arbitrary object supported by the comparator used to create `set`.

modifies `set` so that it contains one less copy of `item`. If 0 copies of `item` exist it is removed from `set`.

returns `unspecified`

`sorted-set-contains?` *set item* [procedure]

requires `set` be a sorted-set, and `item` be an arbitrary object supported by the comparator used to create `set`

returns a boolean indicating whether or not the `set` contains `item`.

`set-length` *set* [procedure]

requires `set` be a sorted-set

returns the number of items contained in `set`

3.5.3 Examples

2 procedures are provided for creating a `sorted-set`. The first creates an empty set and the other populates the set with the items passed to it:

```
(enumerable-collect (make-sorted-set :comparator +number-comparator+)
  +list-collector+)
 ⇒ ()

(enumerable-collect (sorted-set :comparator +number-comparator+ 1 1 1 3)
  +list-collector+)

 ⇒ (1 3)
```

Assuming a sorted-set has been created as above, to insert an item into a sorted-set use:

```
(sorted-set-insert! set 1)
  ⇒ #unspecified
```

Or deleted with:

```
(sorted-set-delete! set 1)
  ⇒ #unspecified
```

To check to see if a sorted-set is empty:

```
(sorted-set-empty? set)
  ⇒ #f
```

To check whether an item is a member of a sorted-set:

```
(sorted-set-contains? set 1)
  ⇒ #t

(sorted-set-contains? set 3)
  ⇒ #f
```

And finally, to make a copy:

```
(let ((set2 (sorted-set-copy? set)))
  (eq? set set2))
  ⇒ #f
```

`sorted-set` also implements the `set`, `collection`, `mutable`, and `enumerable` protocols. See Section 2.10.3 [set Examples], page 22. See Section 2.1.3 [collection Examples], page 4. See Section 2.3.3 [extendable Examples], page 7, See Section 2.4.3 [mutable Examples], page 7. See Section 2.6.3 [enumerable Examples], page 11.

3.6 hash-set

3.6.1 Overview

`hash-set` is an implementation of set. As its name would imply, it is a hashtable-based implementation.

3.6.2 API Reference

`hash-set` implements the `set`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section 2.10 [set], page 20. See Section 2.1 [collection], page 3. See Section 2.3 [extendable], page 6. See Section 2.4 [mutable], page 7. See Section 2.6 [enumerable], page 9.

`hash-set?` *object* [procedure]

returns a boolean indicating whether or not `object` is a hash-set.

`make-hash-set` [*:comparator*] [procedure]

requires `:comparator` be an object implementing the `comparator` protocol. `comparator` must be hashable. If it is not provided the default hashtable equality and hash functions are used.

returns an instance of `hash-set`

`hash-set` [*:comparator*] *. items* [procedure]

requires `:comparator` be an object implementing the `comparator` protocol and `items` be a list of objects for which `comparator` is applicable. `comparator` must be hashable. If it is not provided the default hashtable equality and hash functions are used.

returns an instance of `hash-set`

modifies the returned `hash-set` so that it contains `items`

`hash-set-copy` *set* [procedure]

requires `set` be a hash-set

returns a shallow copy of `set`.

`hash-set-empty?` *set* [procedure]

requires `set` be a hash-set

returns a boolean indicaing whether or not `set` contains any items.

`hash-set-insert!` *set item* [procedure]

requires `set` be a hash-set, and `item` be an arbitrary object supported by the comparator used to create `set`.

modifies `set` so that it contains a copy or an addtional copy, if one already exists, of `item`.

returns `unspecified`

`hash-set-delete!` *set item* [procedure]

requires `set` be a hash-set, and `item` be an arbitrary object supported by the comparator used to create `set`.

modifies `set` so that it contains one less copy of `item`. If 0 copies of `item` exist it is removed from `set`.

returns `unspecified`

`hash-set-contains?` *set item* [procedure]

requires `set` be a hash-set, and `item` be an arbitrary object supported by the comparator used to create `set`

returns a boolean indicating whether or not the `set` contains `item`.

`set-length` *set* [procedure]

requires `set` be a hash-set

returns the number of items contained in `set`

3.6.3 Examples

2 procedures are provided for creating a `hash-set`. The first creates an empty set and the other populates the set with the items passed to it:

```
(enumerable-collect (make-hash-set :comparator +number-comparator+)
  +list-collector+)
  ⇒ ()

(enumerable-collect (hash-set :comparator +number-comparator+ 1 1 1 3)
  +list-collector+)

  ⇒ (1 3)
```

Assuming a hash-set has been created as above, to insert an item into a hash-set use:

```
(hash-set-insert! set 1)
  ⇒ #unspecified
```

Or deleted with:

```
(hash-set-delete! set 1)
  ⇒ #unspecified
```

To check to see if a hash-set is empty:

```
(hash-set-empty? set)
  ⇒ #f
```

To check whether an item is a member of a hash-set:

```
(hash-set-contains? set 1)
  ⇒ #t

(hash-set-contains? set 3)
  ⇒ #f
```

And finally, to make a copy:

```
(let ((set2 (hash-set-copy? set)))
  (eq? set set2))
  ⇒ #f
```

`hash-set` also implements the `set`, `collection`, `mutable`, and `enumerable` protocols. See Section 2.10.3 [set Examples], page 22. See Section 2.1.3 [collection Examples], page 4. See Section 2.3.3 [extendable Examples], page 7, See Section 2.4.3 [mutable Examples], page 7. See Section 2.6.3 [enumerable Examples], page 11.

3.7 stretchy-vector

3.7.1 Overview

`stretchy-vector` is an auto-resizing vector; it grows and shrinks as items are added and deleted. It provides amortized constant time access.

3.7.2 API Reference

`stretchy-vector` implements the `collection`, `extendable`, `indexable`, `mutable`, and `enumerable` protocols. See Section 2.10 [set], page 20. See Section 2.1 [collection], page 3. See Section 2.3 [extendable], page 6. See Section 2.2 [indexable], page 4. See Section 2.4 [mutable], page 7. See Section 2.6 [enumerable], page 9.

`stretchy-vector?` *object* [procedure]

returns a boolean indicating whether or not `object` is a stretchy-vector.

`make-stretchy-vector` *len* [*fill*] [procedure]

requires `len` be an integer specifying the requested initial vector capacity and `fill` be an arbitrary value to initialize each element. If not provided, `fill` defaults to `#unspecified`.

returns a stretchy-vector with an initial capacity of at least `len`.

`stretchy-vector` . *elems* [procedure]

requires `elems` be a list of arbitrary objects.

returns a stretchy-vector with initial elements `elems`.

`stretchy-vector-length` *vec* [procedure]

requires `vec` be a stretchy-vector.

returns the length of the stretchy-vector `vec`.

`stretchy-vector-expand!` *vec len* [procedure]

requires `vec` be a stretchy-vector and `len` be a positive integer.

returns a `vec` expanded to at least `len`.

`stretchy-vector-resize!` *vec new-len* [procedure]

requires `vec` be a stretchy-vector and `new-len` be a positive integer.

modifies `vec` such that if `new-len` is larger than the the current length then all of the new elements are set to `#unspecified`, and if `new-len` is smaller than the current length, the elements greater than or equal to `new-len` are dropped.

returns a `vec` resized to `new-len`.

`stretchy-vector-capacity` *vec* [procedure]

requires `vec` be a stretchy-vector.

returns the capacity of the stretchy-vector `vec`.

`stretchy-vector-set!` *vec index val* [procedure]

requires `vec` be a stretchy-vector, `index` be a positive integer, and `val` be an arbitrary object.

modifies `vec` so that the value at `index` is `value`. This may require expanding the `vect`, if it currently does not contain `index`.

returns `#unspecified`.

`stretchy-vector-ref` *vec index* [procedure]

requires `vec` be a stretchy-vector and `index` be a positive integer.

returns the value found at `index` or throws `&invalid-index-exception` if `index` is not in `vec`.

`list->stretchy-vector` *lst* [procedure]

requires `lst` be a list of arbitrary objects.

returns a stretchy-vector containing the elements of `lst`.

`stretchy-vector->list` *vec* [procedure]

requires `vec` be a stretchy-vector of arbitrary objects.

returns a list containing the elements of `vec`.

`vector->stretchy-vector` *vec* [procedure]

requires `vec` be a vector of arbitrary objects.

returns a stretchy-vector containing the elements of `vec`.

`stretchy-vector->vector` *vec* [procedure]

requires `vec` be a stretchy-vector of arbitrary objects.

returns a vector containing the elements of `vec`.

`stretchy-vector-map` *proc vec* [procedure]

requires `proc` be a single argument procedure compatible with being called element-wise to `vec` and `vec` be a stretchy-vector of arbitrary objects.

returns a new stretchy-vector containing the results of applying `proc` to th elements of `vec`.

`stretchy-vector-map` *proc vec* [procedure]

requires `proc` be a single argument procedure compatible with being called element-wise to `vec` and `vec` be a stretchy-vector of arbitrary objects.

modifies the elements of `vec` so that they are the value obtained by applying `proc` to each element.

returns `vec`.

`stretchy-vector-copy` *vec* [procedure]

requires `vec` be a stretchy-vector of arbitrary objects.

returns a shallow copy of `vec`.

`stretchy-vector-extend!` *vec val* [procedure]

requires `vec` be a stretchy-vector of arbitrary objects and `val` be an arbitrary object.

modifies `vect` by adding `val` to the end.

returns `#unspecified`.

`stretchy-vector-remove!` *vec* [procedure]

requires `vec` be a stretchy-vector of arbitrary objects

modifies `vect` by removing the last element.

returns the element removed from `vec`.

`stretchy-vector-append` *vec1 vec2* [procedure]

requires `vec1` and `vec2` be stretchy-vectors of arbitrary objects

returns a new stretchy-vector containing the elements of `vec1` followed by those in `vec2`.

`stretchy-vector-append!` *vec1 vec2* [procedure]

requires `vec1` and `vec2` be stretchy-vectors of arbitrary objects

modifies `vec1` so that its current elements are followed by the elements of `vec2`. **returns** `#unspecified`.

3.7.3 Examples

To test whether an object is a `stretchy-vector` use the predicate `stretchy-vector?`:

```
(stretchy-vector? (stretchy-vector))
  ⇒ #t

(stretchy-vector? (vector))
  ⇒ #f
```

Two procedures are used to create `stretchy-vectors`. The first creates an empty vector of a specified size with an optional fill value, and the second allows for the creation of a stretchy vector containing the passed values.

```
(let* ((fill 9)
       (vec (make-stretchy-vector 3 9)))
   (enumerable-collect vec +list-collector+))
  ⇒ (9 9 9)

(let ((vec (stretchy-vector 1 2 3)))
   (enumerable-collect vec +list-collector+))
  ⇒ (1 2 3)
```

As with regular vectors, you can reference and set the values of individual `stretchy-vector` elements:

```
(let ((vec (stretchy-vector 1 2 3)))
   (stretchy-vector-ref vec 1))
  ⇒ 2

(let ((vec (stretchy-vector 4 5 6)))
   (stretchy-vector-set! vec 2 7)
   (stretchy-vector-ref vec 2))
  ⇒ 7
```

The length of a `stretchy-vector` is determined with `stretchy-vector-length`:

```
(stretchy-vector-length (stretchy-vector 1 2 3 4 5))
  ⇒ 5
```

A shallow copy of a `stretchy-vector` is obtained with `stretchy-vector-copy`:

```
(let* ((vec1 (stretchy-vector 1 2 3))
       (vec2 (stretchy-vector-copy vec1)))
    (eq? vec1 vec2)
    ⇒ #f
    (equal? vec2 vec2)
    ⇒ #t
```

`stretchy-vector` also implements the `collection`, `mutable`, `indexable`, `extendable`, `enumerable`, and `dictionary-enumerable` protocols. See Section 2.1.3 [collection Examples], page 4. See Section 2.4.3 [mutable Examples], page 7. See Section 2.2.3 [indexable Examples], page 5. See Section 2.3.3 [extendable Examples], page 7. See Section 2.6.3 [enumerable Examples], page 11. See Section 2.16.3 [dictionary-enumerable Examples], page 35.

3.8 contiguous-stack

3.8.1 Overview

`contiguous-stack` is a contiguous implementation of a stack, a last-in-first-out data structure, with a finite capacity.

3.8.2 API Reference

`contiguous-stack` implements the `stack`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section 2.12 [stack], page 25. See Section 2.1 [collection], page 3. See Section 2.3 [extendable], page 6. See Section 2.4 [mutable], page 7. See Section 2.6 [enumerable], page 9.

`contiguous-stack?` *object* [procedure]

  returns a boolean indicating whether or not `object` is a contiguous-stack.

`make-contiguous-stack` *:capacity* [procedure]

  requires `:capacity` be a positive integer specifying the maximum capacity of the stack.

  returns a new `contiguous-stack` with a capacity of `:capacity`

`contiguous-stack` *:capacity . items* [procedure]

  requires `:capacity` be a positive integer specifying the maximum capacity of the stack and items be a list of items to initialize the stack with.

returns a new `contiguous-stack` with a capacity of `:capacity` containing `items` pushed from right-to-left onto the stack. If the number of `items` is greater than `:capacity`, `&invalid-argument-exeption` is thrown.

`contiguous-stack-copy` *stack* [procedure]

requires `stack` be a `contiguous-stack`.

returns returns a shallow copy of `stack`.

`contiguous-stack-empty?` *stack* [procedure]

requires `stack` be a `contiguous-stack`.

returns returns a boolean indicating whether or not the stack is empty.

`contiguous-stack-length` *stack* [procedure]

requires `stack` be a `contiguous-stack`.

returns returns the number of items on the stack (i.e., length) .

`contiguous-stack-capacity` *stack* [procedure]

requires `stack` be a `contiguous-stack`.

returns returns the maximum size (i.e., length) of `stack`.

`contiguous-stack-push!` *stack item* [procedure]

requires `stack` be a `contiguous-stack` and item be an arbitrary object.

modifies `stack` by pushing item on the top of the stack.

returns returns `#unspecified` or if there is no free capacity, throws `&invalid-state-exception`.

`contiguous-stack-pop!` *stack* [procedure]

requires `stack` be a `contiguous-stack`.

modifies `stack` by removing the item on the top of the stack.

returns returns the item removed from `stack` or if `stack` is empty, throws `&invalid-state-exception`.

`contiguous-stack-top` *stack* [procedure]

requires `stack` be a `contiguous-stack`.

returns returns the top item from `stack` or if `stack` is empty, throws `&invalid-state-exception`.

3.8.3 Examples

2 procedures are provided for creating a `contiguous-stack`: The first creates an empty stack and the other populates the stack with the items passed to it:

```
(enumerable-collect (make-contiguous-stack :capacity 4)
  +list-collector+)
  ⇒ ()

(enumerable-collect (contiguous-stack :capacity 4 1 2 3)
  +list-collector+)
  ⇒ (1 2 3)
```

An item can be pushed onto the stack with `contiguous-stack-push!`:

```
(let ((stack (contiguous-stack 4)))
  (contiguous-stack-push! stack 1)
  (contiguous-stack-top stack))
  ⇒ 1
```

The top item of the stack can be non-destructively obtained with `contiguous-stack-top`:

```
(let ((stack (contiguous-stack 4 1 2 3)))
  (contiguous-stack-top stack))
  ⇒ 1
```

The top item can be removed from the stack with `contiguous-stack-pop!`:

```
(let ((stack (contiguous-stack 4 1 2 3)))
  (contiguous-stack-pop! stack)
  (contiguous-stack-top stack))
  ⇒ 2
```

To test if a stack is empty, use `contiguous-stack-empty?`:

```
(contiguous-stack-empty? (contiguous-stack 4))
  ⇒ #t

(contiguous-stack-empty? (contiguous-stack 4 1 2))
  ⇒ #f
```

The size or length of a stack is obtained with `contiguous-stack-length`:

```
(contiguous-stack-length (contiguous-stack 4 1 2))
  ⇒ 2
```

The capacity or maximum length of a stack is obtained with `contiguous-stack-capacity`:

```
(contiguous-stack-capacity (contiguous-stack-capacity 4 1 2))
  ⇒ 4
```

To make a shallow copy a stack, use `contiguous-stack-copy`:

```
(let* ((stack1 (contiguous-stack 4 1 2))
       (stack2 (contiguous-stack-copy stack1)))
   (eq? stack1 stack2))
  ⇒ #f
```

`contiguous-stack` also implements the `stack`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section 2.12.3 [stack Examples], page 26. See Section 2.1.3 [collection Examples], page 4. See Section 2.3.3 [extendable Examples], page 7. See Section 2.4.3 [mutable Examples], page 7. See Section 2.6.3 [enumerable Examples], page 11.

3.9 linked-stack

3.9.1 Overview

`linked-stack` is a linked-list based implementation of a stack, a last-in-first-out data structure.

3.9.2 API Reference

`linked-stack` implements the `stack`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section 2.12 [stack], page 25. See Section 2.1 [collection], page 3. See Section 2.3 [extendable], page 6. See Section 2.4 [mutable], page 7. See Section 2.6 [enumerable], page 9.

`linked-stack?` *object* [procedure]

returns a boolean indicating whether or not `object` is a linked-stack.

`make-linked-stack` [procedure]

returns a new empty `linked-stack`.

`linked-stack .` *items* [procedure]

requires items be a list of items to initialize the stack with.

returns a new `linked-stack` containing `items` pushed from right-to-left onto the stack.

`linked-stack-copy` *stack* [procedure]

requires `stack` be a `linked-stack`.

returns returns a shallow copy of `stack`.

`linked-stack-empty?` *stack* [procedure]

requires `stack` be a `linked-stack`.

returns returns a boolean indicating whether or not the stack is empty.

`linked-stack-length` *stack* [procedure]

requires `stack` be a `linked-stack`.

returns returns the number of items on the stack (i.e., length) .

`linked-stack-push!` *stack item* [procedure]

requires `stack` be a `linked-stack` and item be an arbitrary object.

modifies `stack` by pushing item on the top of the stack.

returns returns `#unspecified`.

`linked-stack-pop!` *stack* [procedure]

requires `stack` be a `linked-stack`.

modifies `stack` by removing the item on the top of the stack.

returns returns the item removed from `stack` or if `stack` is empty, throws `&invalid-state-exception`.

`linked-stack-top` *stack* [procedure]

requires `stack` be a `linked-stack`.

returns returns the top item from `stack` or if `stack` is empty, throws `&invalid-state-exception`.

3.9.3 Examples

2 procedures are provided for creating a `linked-stack`: The first creates an empty stack and the other populates the stack with the items passed to it:

```
(enumerable-collect (make-linked-stack)
  +list-collector+)
  ⇒ ()

(enumerable-collect (linked-stack 1 2 3)
  +list-collector+)
  ⇒ (1 2 3)
```

An item can be pushed onto the stack with `linked-stack-push!`:

```
(let ((stack (linked-stack)))
  (linked-stack-push! stack 1)
  (linked-stack-top stack))
  ⇒ 1
```

The top item of the stack can be non-destructively obtained with `linked-stack-top`:

```
(let ((stack (linked-stack 1 2 3)))
  (linked-stack-top stack))
  ⇒ 1
```

The top item can be removed from the stack with `linked-stack-pop!`:

```
(let ((stack (linked-stack 1 2 3)))
  (linked-stack-pop! stack)
  (linked-stack-top stack))
  ⇒ 2
```

To test if a stack is empty, use `linked-stack-empty?`:

```
(linked-stack-empty? (linked-stack))
  ⇒ #t

(linked-stack-empty? (linked-stack 1 2))
  ⇒ #f
```

The size or length of a stack is obtained with `linked-stack-length`:

```
(linked-stack-length (linked-stack 1 2))
  ⇒ 2
```

To make a shallow copy a stack, use `linked-stack-copy`:

```
(let* ((stack1 (linked-stack 1 2))
       (stack2 (linked-stack-copy stack1)))
   (eq? stack1 stack2))
  ⇒ #f
```

`linked-stack` also implements the `stack`, `collection`, `extendable`, `mutable`, and `enumerable` protocols. See Section 2.12.3 [stack Examples], page 26. See Section 2.1.3 [collection Examples], page 4. See Section 2.3.3 [extendable Examples], page 7. See Section 2.4.3 [mutable Examples], page 7. See Section 2.6.3 [enumerable Examples], page 11.

3.10 contiguous-queue

3.10.1 Overview

3.10.2 API Reference

3.10.3 Examples

3.11 linked-queue

3.11.1 Overview

3.11.2 API Reference

3.11.3 Examples

3.12 binary-heap

3.12.1 Overview

A `binary-heap` is an implementation of a priority queue featuring contiguous storage and a specified maximum size.

3.12.2 API Reference

3.12.3 Examples

3.13 pairing-heap

3.13.1 Overview

3.13.2 API Reference

3.13.3 Examples

3.14 sorted-dictionary

3.14.1 Overview

3.14.2 API Reference

3.14.3 Examples

4 Global Index

V

5 Table of contents

Table of Contents

Short Contents