arew-scheme

sourcehut success

Various libraries for Chez Scheme (R7RS, SRFI and more...)

Status

- [ ] SRFI-1: (scheme list) ???
- [ ] SRFI-2: ???
- [ ] SRFI-4: ???
- [ ] SRFI-5: ???
- [ ] SRFI-6: ???
- [ ] SRFI-8: ???
- [ ] SRFI-9: ???
- [ ] SRFI-11: ???
- [ ] SRFI-13: ???
- [ ] SRFI-14: (scheme charset) ???
- [ ] SRFI-16: ???
- [ ] SRFI-17: ???
- [ ] SRFI-19: ???
- [ ] SRFI-23: ???
- [ ] SRFI-25: missing
- [ ] SRFI-26: missing doc
- [ ] SRFI-27: missing
- [ ] SRFI-28: missing tests

- [ ] SRFI-29: missing tests, missing doc
- [ ] SRFI-31: missing tests, missing doc
- [ ] SRFI-34: missing tests, missing doc
- [ ] SRFI-35: missing tests, missing doc
- [ ] SRFI-37: missing tests, missing doc
- [ ] SRFI-38: missing tests, missing doc
- [ ] SRFI-39: missing doc
- [ ] SRFI-41: (scheme stream) missing doc, missing tests
- [ ] SRFI-42: missing doc, missing tests
- [ ] SRFI-43: missing doc, missing tests
- [ ] SRFI-45: missing doc, missing tests
- [ ] SRFI-48: missing doc, missing tests
- [ ] SRFI-51: missing doc, missing tests
- [ ] SRFI-54: missing doc, missing tests
- [ ] SRFI-60: missing doc, missing tests
- [ ] SRFI-61: missing doc, missing tests
- [ ] SRFI-64: no
- [ ] SRFI-67: missing doc, missing tests
- [ ] SRFI-69: missing doc, missing tests
- [ ] SRFI-78: no
- [ ] SRFI-98: missing doc, missing tests
- [ ] SRFI-99: not yet
- [ ] SRFI-101: (scheme rlist): missing doc, missing tests
- [ ] SRFI-111: (scheme box): missing tests
- [ ] SRFI-113: (scheme set): missing tests
- [ ] SRFI-115: (scheme regex): not yet
- [ ] SRFI-116: (scheme ilist): not yet
- [ ] SRFI-117: (scheme list-queue): missing doc, missing tests
- [ ] SRFI-124: (scheme ephemeron): missing doc, missing tests
- [ ] SRFI-125: (scheme hash-table): missing tests
- [ ] SRFI-126: not yet
- [ ] SRFI-127: (scheme lseq): missing doc, missing tests
- [ ] SRFI-128: (scheme comparator): missing tests
- [ ] SRFI-129: not yet
- [ ] SRFI-132: (scheme sort): not yet
- [ ] SRFI-133: (scheme vector): not yet
- [ ] SRFI-134: (scheme idque)
- [ ] SRFI-135: (scheme text)
- [ ] SRFI-141: (scheme division)
- [ ] SRFI-143: (scheme fixnum)
- [ ] SRFI-144: (scheme flonum)
- [ ] SRFI-145
- [ ] SRFI-146: (scheme mapping) and (scheme mapping hash)
- [ ] SRFI-151: (scheme bitwise)
- [ ] SRFI-152
- [ ] SRFI-156

- [ ] SRFI-158: (scheme generator)
- [ ] SRFI-159: (scheme show)
- [ ] SRFI-160: (scheme vector @)
- [ ] SRFI-167
- [ ] SRFI-168
- [ ] SRFI-170
- [ ] SRFI-173 ## `(srfi srfi-1)`

This is based on SRFI-1.

Abstract

TODO

Reference

Constructors

`(cons a d)`

The primitive constructor. Returns a newly allocated pair whose `car` is `a` and whose `cdr` is `d`. The pair is guaranteed to be different (in the sense of `eqv?`) from every existing object.

`(list object ...)`

Returns a newly allocated list of its arguments.

`(xcons d a)`

```
(lambda (d a) (cons a d))
```

Of utility only as a value to be conveniently passed to higher-order procedures.

```
(xcons '(b c) 'a) ;; => (a b c)
```

The name stands for “eXchanged CONS.”

`(cons* obj ... tail)`

Like list, but the last argument provides the tail of the constructed list.

`(make-list n [fill])`

Returns an n-element list, whose elements are all the value fill. If the fill argument is not given, the elements of the list may be arbitrary values.

```
(make-list 4 'c) => (c c c c)
```

`(list-tabulate n init-proc)`

Returns an n-element list. Element i of the list, where 0 <= i < n, is produced by (init-proc i). No guarantee is made about the dynamic order in which init-proc is applied to these indices.

```
(list-tabulate 4 values) => (0 1 2 3)
```

`(list-copy flist)`

Copies the spine of the argument.

`(circular-list elt1 elt2 ...)`

Constructs a circular list of the elements.

```
(circular-list 'z 'q) => (z q z q z q ...)
```

`(iota count [start step])`

Returns a list containing the elements:

```
(start start+step ... start+(count-1)*step)
```

The start and step parameters default to 0 and 1, respectively.

```
(iota 5) => (0 1 2 3 4)
(iota 5 0 -0.1) => (0 -0.1 -0.2 -0.3 -0.4)
```

Predicates

`(proper-list? x)`

Returns true iff x is a proper list – a finite, nil-terminated list.

More carefully: The empty list is a proper list. A pair whose cdr is a proper list is also a proper list. The opposite of proper is improper.

`(circular-list? x)`

True if x is a circular list. A circular list is a value such that for every n >= 0, cdrn(x) is a pair.

Terminology: The opposite of circular is finite.

‘(dotted-list? x)

True if x is a finite, non-nil-terminated list. That is, there exists an n >= 0 such that cdrn(x) is neither a pair nor (). This includes non-pair, non-() values (e.g. symbols, numbers), which are considered to be dotted lists of length 0.

`(pair? obj)`

Returns #t if object is a pair; otherwise, #f.

`(null? obj)`

Returns #t if object is the empty list; otherwise, #f.

`(null-list? list)`

List is a proper or circular list. This procedure returns true if the argument is the empty list (), and false otherwise. It is an error to pass this procedure a value which is not a proper or circular list. This procedure is recommended as the termination condition for list-processing procedures that are not defined on dotted lists.

‘(not-pair? x)

Provided as a procedure as it can be useful as the termination condition for list-processing procedures that wish to handle all finite lists, both proper and dotted.

`(list= elt= list1 ...)`

Determines list equality, given an element-equality procedure.

Selectors

(car pair)

(cdr pair)

These functions return the contents of the car and cdr field of their argument, respectively. Note that it is an error to apply them to the empty list.

Also the following selectors are defined:

- `caar`
- `cadr`
- `cdar`
- `cddr`
- `caaar`
- `caadr`
- `cadar`
- `caddr`
- `cdaar`
- `cdadr`
- `cddar`

- cdddr
- caaaar
- caaadr
- caadar
- caaddr
- cadaar
- cadadr
- caddar
- cadddr
- cdaaar
- cdaadr
- cdadar
- cdaddr
- cddaar
- cddadr
- cdddar
- cddddr

`(list-ref clist i)`

Returns the ith element of clist. (This is the same as the car of (drop clist i).) It is an error if i >= n, where n is the length of clist.

```
(list-ref '(a b c d) 2) => c
```

`(first pair)`

`(second pair)`

`(third pair)`

`(fourth pair)`

`(fifth pair)`

`(sixth pair)`

`(seventh pair)`

`(eighth pair)`

`(ninth pair)`

`(tenth pair)`

Synonyms for `car`, `cadr`, `caddr`, . . .

`(car+cdr pair)`

The fundamental pair deconstructor:

`(lambda (p) (values (car p) (cdr p)))`

This can, of course, be implemented more efficiently by a compiler.

`(take lst i)`

`(drop lst i)`

`take` returns the first `I` elements of list `LST`. `drop` returns all but the first `i` elements of list `LST`.

```
(take '(a b c d e) 2) ;; => (a b)
(drop '(a b c d e) 2) ;; => (c d e)
```

`LST` may be any value – a proper, circular, or dotted list:

```
(take '(1 2 3 . d) 2) ;; => (1 2)
(drop '(1 2 3 . d) 2) ;; => (3 . d)
(take '(1 2 3 . d) 3) ;; => (1 2 3)
(drop '(1 2 3 . d) 3) ;; => d
```

For a legal `I`, `take` and `drop` partition the list in a manner which can be inverted with append:

`(equal? (append (take lst i) (drop x i)) lst)`

`drop` is exactly equivalent to performing `i cdr` operations on `LST`; the returned value shares a common tail with `LST`. If the argument is a list of non-zero length, `take` is guaranteed to return a freshly-allocated list, even in the case where the entire list is taken, e.g. `(take lst (length lst))`.

`(take-right flist i)`

`(drop-right flist i)`

`take-right` returns the last `I` elements of `FLIST`. `drop-right` returns all but the last `I` elements of `FLIST`.

```
(take-right '(a b c d e) 2) => (d e)
(drop-right '(a b c d e) 2) => (a b c)
```

The returned list may share a common tail with the argument list.

`FLIST` may be any finite list, either proper or dotted:

```
(take-right '(1 2 3 . d) 2) => (2 3 . d)
(drop-right '(1 2 3 . d) 2) => (1)
(take-right '(1 2 3 . d) 0) => d
(drop-right '(1 2 3 . d) 0) => (1 2 3)
```

For a legal `I`, `take-right` and `drop-right` partition the list in a manner which can be inverted with `append`:

```
(equal? (append (take flist i) (drop flist i)) flist)
```

`take-right`'s return value is guaranteed to share a common tail with `FLIST`. If the argument is a list of non-zero length, `drop-right` is guaranteed to return a freshly-allocated list, even in the case where nothing is dropped, e.g. `(drop-right flist 0)`.

`(take! x i)`

`(drop-right! flist i)`

`take!` and `drop-right!` are "linear-update" variants of `take` and `drop-right`: the procedure is allowed, but not required, to alter the argument list to produce the result.

If `x` is circular, `take!` may return a shorter-than-expected list:

```
(take! (circular-list 1 3 5) 8) => (1 3)
(take! (circular-list 1 3 5) 8) => (1 3 5 1 3 5 1 3)
```

`(split-at  x i)`

`(split-at! x i)`

`split-at` splits the list `x` at index `i`, returning a list of the first `i` elements, and the remaining tail. It is equivalent to:

```
(values (take x i) (drop x i))
```

`split-at!` is the linear-update variant. It is allowed, but not required, to alter the argument list to produce the result.

```
(split-at '(a b c d e f g h) 3) ;; => (a b c) and (d e f g h)
```

`(last pair)`

`(last-pair pair)`

`last` returns the last element of the non-empty, finite list `pair`. `last-pair` returns the last pair in the non-empty, finite list `pair`:

```
(last '(a b c)) ;; => c
(last-pair '(a b c)) ;; => (c)
```

Miscellaneous

`(length list)`

`(length+ clist)`

Both `length` and `length+` return the length of the argument. It is an error to pass a value to length which is not a proper list (finite and nil-terminated). In particular, this means an implementation may diverge or signal an error when length is applied to a circular list.

`length+`, on the other hand, returns `#f` when applied to a circular list.

The length of a proper list is a non-negative integer `n` such that `cdr` applied `n` times to the list produces the empty list.

`(append  list1 ...)`

`(append! list1 ...)`

`append` returns a list consisting of the elements of `list1` followed by the elements of the other list parameters.

```
(append '(x) '(y))         =>  (x y)
(append '(a) '(b c d))     =>  (a b c d)
(append '(a (b)) '((c)))   =>  (a (b) (c))
```

The resulting list is always newly allocated, except that it shares structure with the final list argument. This last argument may be any value at all; an improper list results if it is not a proper list. All other arguments must be proper lists.

```
(append '(a b) '(c . d))   =>  (a b c . d)
(append '() 'a)            =>  a
(append '(x y))            =>  (x y)
(append)                   =>  ()
```

`append!` is the “linear-update” variant of append – it is allowed, but not required, to alter `cons` cells in the argument lists to construct the result list. The last argument is never altered; the result list shares structure with this parameter.

`(concatenate  list-of-lists)`

`(concatenate! list-of-lists)`

These functions append the elements of their argument together. That is, `concatenate` returns:

```
(apply append list-of-lists)
```

Or, equivalently,

```
(reduce-right append '() list-of-lists)
```

`concatenate!` is the linear-update variant, defined in terms of `append!` instead of `append`.

As with `append` and `append!`, the last element of the input list may be any value at all.

`(reverse list)`

`(reverse! list)`

`reverse` returns a newly allocated list consisting of the elements of list in reverse order.

```
(reverse '(a b c)) ;; =>  (c b a)
(reverse '(a (b c) d (e (f)))) ;; =>  ((e (f)) d (b c) a)
```

`reverse!` is the linear-update variant of reverse. It is permitted, but not required, to alter the argument's `cons` cells to produce the reversed list.

`(append-reverse rev-head tail)`

`(append-reverse! rev-head tail)`

`append-reverse` returns `(append (reverse rev-head) tail)`. It is provided because it is a common operation – a common list-processing style calls for this exact operation to transfer values accumulated in reverse order onto the front of another list, and because the implementation is significantly more efficient than the simple composition it replaces. (But note that this pattern of iterative computation followed by a `reverse` can frequently be rewritten as a recursion, dispensing with the `reverse` and `append-reverse` steps, and shifting temporary, intermediate storage from the heap to the stack, which is typically a win for reasons of cache locality and eager storage reclamation.)

`append-reverse!` is just the linear-update variant – it is allowed, but not required, to alter rev-head's `cons` cells to construct the result.

`(zip clist1 clist2 ...)`

```
(lambda lists (apply map list lists))
```

If `zip` is passed n lists, it returns a list as long as the shortest of these lists, each element of which is an n-element list comprised of the corresponding elements from the parameter lists.

```
(zip '(one two three)
     '(1 2 3)
     '(odd even odd even odd even odd even))
     ;; => ((one 1 odd) (two 2 even) (three 3 odd))

(zip '(1 2 3)) => ((1) (2) (3))
```

`(unzip1 list)`

(unzip2 list)

(unzip3 list)

(unzip4 list)

(unzip5 list)

`unzip1` takes a list of lists, where every list must contain at least one element, and returns a list containing the initial element of each such list. That is, it returns `(map car lists)`. `unzip2` takes a list of lists, where every list must contain at least two elements, and returns two values: a list of the first elements, and a list of the second elements. `unzip3` does the same for the first three elements of the lists, and so forth.

```
(unzip2 '((1 one) (2 two) (3 three))) ;; => '((1 2 3) (one two three))
```

(count pred clist1 ...)

`pred` is a procedure taking as many arguments as there are lists and returning a single value. It is applied element-wise to the elements of the lists, and a count is tallied of the number of elements that produce a true value. This count is returned. `count` is "iterative" in that it is guaranteed to apply `pred` to the list elements in a left-to-right order. The counting stops when the shortest list expires.

```
(count even? '(3 1 4 1 5 9 2 5 6)) => 3
(count < '(1 2 4 8) '(2 4 6 8 10 12 14 16)) => 3
```

At least one of the argument lists must be finite:

```
(count < '(3 1 4 1) (circular-list 1 10)) => 2
```

Fold, unfold & map

(fold kons knil list1 ...)

TODO

(fold-right kons knil list1 ...)

TODO

(pair-fold kons knil list1 ...)

TODO

`(pair-fold-right kons knil list1 ...)`

TODO

`(reduce f ridentity list)`

`reduce` is a variant of `fold`.

`ridentity` should be a "right identity" of the procedure `f` – that is, for any value `x` acceptable to `f`:

```
(f x ridentity) ;; => x
```

Note: that `ridentity` is used only in the empty-list case. You typically use reduce when applying `f` is expensive and you'd like to avoid the extra application incurred when fold applies `f` to the head of list and the identity value, redundantly producing the same value passed in to `f`. For example, if `f` involves searching a file directory or performing a database query, this can be significant. In general, however, `fold` is useful in many contexts where `reduce` is not (consider the examples given in the `fold` definition – only one of the five folds uses a function with a right identity. The other four may not be performed with reduce).

`(reducse-right f ridentity list)`

`reduce-right` is the `fold-right` variant of reduce. It obeys the following definition:

```
(reduce-right f ridentity '()) = ridentity
(reduce-right f ridentity '(e1)) = (f e1 ridentity) = e1
(reduce-right f ridentity '(e1 e2 ...)) =
    (f e1 (reduce f ridentity (e2 ...)))
```

... in other words, we compute `(fold-right f ridentity list)`.

```
;; Append a bunch of lists together.
;; I.e., (apply append list-of-lists)
(reduce-right append '() list-of-lists)
```

`(unfold p f g seed [tail-gen])`

TODO

`(unfold-right p f g seed [tail-gen])`

TODO

`(map proc list1 ...)`

`proc` is a procedure taking as many arguments as there are list arguments and returning a single value. map applies `proc` element-wise to the elements of the lists and returns a list of the results, in order. The dynamic order in which proc is applied to the elements of the lists is unspecified.

```
(map cadr '((a b) (d e) (g h))) =>  (b e h)

(map (lambda (n) (expt n n))
     '(1 2 3 4 5))
    =>  (1 4 27 256 3125)

(map + '(1 2 3) '(4 5 6)) =>  (5 7 9)

(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b))) =>  (1 2) or (2 1)
```

At least one of the argument lists must be finite:

```
(map + '(3 1 4 1) (circular-list 1 0)) ;; => (4 1 5 1)
```

`(for-each proc clist1 ...)`

The arguments to `for-each` are like the arguments to `map`, but `for-each` calls `proc` for its side effects rather than for its values. Unlike `map`, `for-each` is guaranteed to call `proc` on the elements of the lists in order from the first element(s) to the last, and the value returned by for-each is unspecified.

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
              (vector-set! v i (* i i)))
            '(0 1 2 3 4))
  v)  =>  #(0 1 4 9 16)
```

At least one of the argument lists must be finite.

`(append-map f list ...)`

`(append-map! f list ...)`

Equivalent to:

```
(apply append (map f clist1 clist2 ...))
```

And:

```
(apply append! (map f clist1 clist2 ...))
```

Map `f` over the elements of the lists, just as in the `map` function. However, the results of the applications are appended together to make the final result.

`append-map` uses `append` to append the results together; `append-map!` uses `append!`.

The dynamic order in which the various applications of `f` are made is not specified.

Example:

```
(append-map! (lambda (x) (list x (- x))) '(1 3 8)) ;; => (1 -1 3 -3 8 -8)
```

At least one of the list arguments must be finite.

`(map! f list1 ...)`

Linear-update variant of `map` – `map!` is allowed, but not required, to alter the cons cells of `list1` to construct the result list.

The dynamic order in which the various applications of `f` are made is not specified. In the n-ary case, `clist2`, `clist3`, ... must have at least as many elements as `list1`.

`(map-in-order f clist1 ...)`

A variant of the map procedure that guarantees to apply `f` across the elements of the `clisti` arguments in a left-to-right order. This is useful for mapping procedures that both have side effects and return useful values.

At least one of the list arguments must be finite.

`(pair-for-each f clist1 ...)`

Like for-each, but `f` is applied to successive sublists of the argument lists. That is, `f` is applied to the `cons` cells of the lists, rather than the lists' elements. These applications occur in left-to-right order.

The f procedure may reliably apply `set-cdr!` to the pairs it is given without altering the sequence of execution.

```
(pair-for-each (lambda (pair) (display pair) (newline)) '(a b c))
 ;; => (a b c)
 ;; => (b c)
 ;; => (c)
```

At least one of the list arguments must be finite.

`(filter-map f clist1 ...)`

Like `map`, but only true values are saved.

```
(filter-map (lambda (x) (and (number? x) (* x x))) '(a 1 b 3 c 7))
    ;; => (1 9 49)
```

The dynamic order in which the various applications of `f` are made is not specified.

At least one of the list arguments must be finite.

Filtering & partitioning

`(filter pred list)`

Return all the elements of list that satisfy predicate `pred`. The list is not disordered – elements that appear in the result list occur in the same order as they occur in the argument list. The returned list may share a common tail with the argument list. The dynamic order in which the various applications of `pred` are made is not specified.

```
(filter even? '(0 7 8 8 43 -4)) => (0 8 8 -4)
```

`(partition pred list)`

Partitions the elements of list with predicate `pred`, and returns two values: the list of in-elements and the list of out-elements. The list is not disordered – elements occur in the result lists in the same order as they occur in the argument list. The dynamic order in which the various applications of pred are made is not specified. One of the returned lists may share a common tail with the argument list.

```
(partition symbol? '(one 2 3 four five 6)) =>
    (one four five)
    (2 3 6)
```

`(remove pred list)`

Returns `list` without the elements that satisfy predicate `pred`:

```
(lambda (pred list) (filter (lambda (x) (not (pred x))) list))
```

The `list` is not disordered – elements that appear in the result list occur in the same order as they occur in the argument list. The returned list may share a common tail with the argument list. The dynamic order in which the various applications of `pred` are made is not specified.

```
(remove even? '(0 7 8 8 43 -4)) => (7 43)
```

`(filter! pred list)`

`(partition! pred list)`

`(remove! pred list)`

Linear-update variants of `filter`, `partition` and `remove`. These procedures are allowed, but not required, to alter the cons cells in the argument list to construct the result lists.

Searching

`(find pred clist)`

Return the first element of clist that satisfies predicate `pred`; false if no element does.

```
(find even? '(3 1 4 1 5 9)) => 4
```

Note that `find` has an ambiguity in its lookup semantics – if `find` returns `#f`, you cannot tell (in general) if it found a #f element that satisfied `pred`, or if it did not find any element at all. In many situations, this ambiguity cannot arise – either the list being searched is known not to contain any `#f` elements, or the list is guaranteed to have an element satisfying `pred`. However, in cases where this ambiguity can arise, you should use `find-tail` instead of find – `find-tail` has no such ambiguity.

`(find-tail pred clist)`

Return the first pair of `clist` whose `car` satisfies `pred`. If no pair does, return false.

`find-tail` can be viewed as a general-predicate variant of the member function.

Examples:

```
(find-tail even? '(3 1 37 -8 -5 0 0)) => (-8 -5 0 0)
(find-tail even? '(3 1 37 -5)) => #f

;; MEMBER X LIS:
(find-tail (lambda (elt) (equal? x elt)) lis)
```

In the circular-list case, this procedure “rotates” the list.

`find-tail` is essentially drop-while, where the sense of the predicate is inverted: `find-tail` searches until it finds an element satisfying the predicate; drop-while searches until it finds an element that doesn’t satisfy the predicate.

`(take-while  pred clist)`

`(take-while! pred clist)`

Returns the longest initial prefix of `clist` whose elements all satisfy the predicate `pred`.

`take-while!` is the linear-update variant. It is allowed, but not required, to alter the argument list to produce the result.

```
(take-while even? '(2 18 3 10 22 9)) => (2 18)
```

`(drop-while pred clist)`

Drops the longest initial prefix of `clist` whose elements all satisfy the predicate `pred`, and returns the rest of the list.

```
(drop-while even? '(2 18 3 10 22 9)) => (3 10 22 9)
```

The circular-list case may be viewed as "rotating" the list.

`(span pred clist)`

`(span!  pred list)`

`(break  pred clist)`

'(break! pred list)

Span splits the list into the longest initial prefix whose elements all satisfy `pred`, and the remaining tail. Break inverts the sense of the predicate: the tail commences with the first element of the input list that satisfies the predicate.

In other words: `span` finds the intial span of elements satisfying `pred`, and break breaks the list at the first element satisfying `pred`.

Span is equivalent to

```
(values (take-while pred clist)
        (drop-while pred clist))
```

`span!` and `break!` are the linear-update variants. They are allowed, but not required, to alter the argument list to produce the result.

```
(span even? '(2 18 3 10 22 9)) =>
      (2 18)
      (3 10 22 9)

(break even? '(3 1 4 1 5 9)) =>
      (3 1)
      (4 1 5 9)
```

`(any pred clist1 ...)`

Applies the predicate across the lists, returning true if the predicate returns true on any application.

If there are n list arguments `clist1 ... clistn`, then `pred` must be a procedure taking n arguments and returning a single value, interpreted as a boolean (that is, `#f` means false, and any other value means true).

`any` applies `pred` to the first elements of the `clisti` parameters. If this application returns a true value, any immediately returns that value. Otherwise, it iterates, applying `pred` to the second elements of the `clisti` parameters, then the third, and so forth. The iteration stops when a true value is produced or one of the lists runs out of values; in the latter case, any returns `#f`. The application of pred to the last element of the lists is a tail call.

Note the difference between `find` and `any` – `find` returns the element that satisfied the predicate; `any` returns the true value that the predicate produced.

Like `every`, `any`'s name does not end with a question mark – this is to indicate that it does not return a simple boolean (#t or #f), but a general value.

```
(any integer? '(a 3 b 2.7))   => #t
(any integer? '(a 3.1 b 2.7)) => #f
(any < '(3 1 4 1 5)
          '(2 7 1 8 2)) => #t
```

`(every pred clist1 ...)`

Applies the predicate across the lists, returning true if the predicate returns true on every application.

If there are n list arguments `clist1 ... clistn`, then `pred` must be a procedure taking n arguments and returning a single value, interpreted as a boolean (that is, #f means false, and any other value means true).

`every` applies `pred` to the first elements of the `clisti` parameters. If this application returns false, every immediately returns false. Otherwise, it iterates, applying `pred` to the second elements of the `clisti` parameters, then the third, and so forth. The iteration stops when a false value is produced or one of the lists runs out of values. In the latter case, `every` returns the true value produced by its final application of pred. The application of `pred` to the last element of the lists is a tail call.

If one of the `clisti` has no elements, `every` simply returns #t.

Like `any`, `every`'s name does not end with a question mark – this is to indicate that it does not return a simple boolean (`#t` or `#f`), but a general value.

`(list-index pred clist1 ...)`

Return the index of the leftmost element that satisfies `pred`.

If there are n list arguments `clist1 ... clistn`, then `pred` must be a function taking n arguments and returning a single value, interpreted as a boolean (that is, `#f` means false, and any other value means true).

`list-index` applies `pred` to the first elements of the `clisti` parameters. If this application returns true, `list-index` immediately returns zero. Otherwise, it iterates, applying `pred` to the second elements of the `clisti` parameters, then the third, and so forth. When it finds a tuple of list elements that cause `pred` to return true, it stops and returns the zero-based index of that position in the lists.

The iteration stops when one of the lists runs out of values; in this case, `list-index` returns `#f`.

```
(list-index even? '(3 1 4 1 5 9)) => 2
(list-index < '(3 1 4 1 5 9 2 5 6) '(2 7 1 8 2)) => 1
(list-index = '(3 1 4 1 5 9 2 5 6) '(2 7 1 8 2)) => #f
```

`(member x list [=])`

`(memq x list)`

`(memv x list)`

These procedures return the first sublist of `list` whose `car` is `x`, where the sublists of `list` are the non-empty lists returned by `(drop list i)` for `i` less than the length of list. If `x` does not occur in list, then `#f` is returned. `memq` uses `eq?` to compare `x` with the elements of list, while `memv` uses `eqv?`, and `member` uses `equal?`.

```
(memq 'a '(a b c))           =>  (a b c)
(memq 'b '(a b c))           =>  (b c)
(memq 'a '(b c d))           =>  #f
(memq (list 'a) '(b (a) c)) =>  #f
(member (list 'a) '(b (a) c))          =>  ((a) c)
(memq 101 '(100 101 102))    =>  *unspecified*
(memv 101 '(100 101 102))    =>  (101 102)
```

The comparison procedure is used to compare the elements `ei` of list to the key `x` in this way:

```
(= x ei) ; list is (e1 ... en)
```

That is, the first argument is always `x`, and the second argument is one of the list elements. Thus one can reliably find the first element of list that is greater than five with `(member 5 list <)`

Note that fully general list searching may be performed with the `find-tail` and `find` procedures, e.g.

```
(find-tail even? list) ; Find the first elt with an even key.
```

Deleting

`(delete x list)`

`(delete! x list)`

`delete` uses the comparison procedure `=`, which defaults to `equal?`, to find all elements of list that are equal to `x`, and deletes them from list. The dynamic order in which the various applications of `=` are made is not specified.

The list is not disordered – elements that appear in the result list occur in the same order as they occur in the argument list. The result may share a common tail with the argument list.

Note that fully general element deletion can be performed with the `remove` and `remove!` procedures, e.g.:

```
;; Delete all the even elements from LIS:
(remove even? lis)
```

The comparison procedure is used in this way: `(= x ei)`. That is, `x` is always the first argument, and a list element is always the second argument. The comparison procedure will be used to compare each element of list exactly once; the order in which it is applied to the various `ei` is not specified. Thus, one can reliably remove all the numbers greater than five from a list with `(delete 5 list <)`.

`delete!` is the linear-update variant of `delete`. It is allowed, but not required, to alter the cons cells in its argument list to construct the result.

`(delete-duplicates  list [=])`

`(delete-duplicates! list [=])`

`delete-duplicates` removes duplicate elements from the list argument. If there are multiple equal elements in the argument list, the result list only contains the first or leftmost of these elements in the result. The order of these surviving elements is the same as in the original list – delete-duplicates does not disorder the list (hence it is useful for "cleaning up" association lists).

The `=` parameter is used to compare the elements of the list; it defaults to equal?. If x comes before y in list, then the comparison is performed (= x y). The comparison procedure will be used to compare each pair of elements in list no more than once; the order in which it is applied to the various pairs is not specified.

Implementations of `delete-duplicates` are allowed to share common tails between argument and result lists – for example, if the list argument contains only unique elements, it may simply return exactly this list.

Be aware that, in general, delete-duplicates runs in time O(n2) for n-element lists. Uniquifying long lists can be accomplished in O(n lg n) time by sorting the list to bring equal elements together, then using a linear-time algorithm to remove equal elements. Alternatively, one can use algorithms based on element-marking, with linear-time results.

`delete-duplicates!` is the linear-update variant of `delete-duplicates`; it is allowed, but not required, to alter the cons cells in its argument list to construct the result.

```
(delete-duplicates '(a b a c a b c z)) => (a b c z)

;; Clean up an alist:
(delete-duplicates '((a . 3) (b . 7) (a . 9) (c . 1))
                   (lambda (x y) (eq? (car x) (car y))))
;;  => ((a . 3) (b . 7) (c . 1))
```

Association lists

An "association list" (or "alist") is a list of pairs. The car of each pair contains a key value, and the cdr contains the associated data value. They can be used to construct simple look-up tables in Scheme. Note that association lists are probably inappropriate for performance-critical use on large data; in these cases, hash tables or some other alternative should be employed.

`(assoc key alist [=])`

`(assq key alist)`

`(assv key alist)`

`alist` must be an association list – a list of pairs. These procedures find the first pair in `alist` whose `car` field is `key`, and returns that pair. If no pair in `alist` has `key` as its `car`, then `#f` is returned. `assq` uses `eq?` to compare `key` with the `car` fields of the pairs in `alist`, while `assv` uses `eqv?` and `assoc` uses `equal?`.

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)                            =>  (a 1)
(assq 'b e)                            =>  (b 2)
(assq 'd e)                            =>  #f
(assq (list 'a) '(((a)) ((b)) ((c))))  =>  #f
(assoc (list 'a) '(((a)) ((b)) ((c)))) =>  ((a))
(assq 5 '((2 3) (5 7) (11 13)))    =>  *unspecified*
(assv 5 '((2 3) (5 7) (11 13)))    =>  (5 7)
```

The comparison procedure is used to compare the elements `ei` of list to the `key` parameter in this way:

```
(= key (car ei)) ; list is (E1 ... En)
```

That is, the first argument is always `key`, and the second argument is one of the list elements. Thus one can reliably find the first entry of alist whose key is greater than five with `(assoc 5 alist <)`

Note that fully general `alist` searching may be performed with the `find-tail` and `find` procedures, e.g.

```
;; Look up the first association in alist with an even key:
(find (lambda (a) (even? (car a))) alist)
```

`(alist-cons key datum alist)`

```
(lambda (key datum alist) (cons (cons key datum) alist))
```

`cons` a new entry mapping `key` to `datum` onto `alist`.

`(alist-copy alist)`

Make a fresh copy of `alist`. This means copying each pair that forms an association as well as the spine of the list, i.e.

```
(lambda (a) (map (lambda (elt) (cons (car elt) (cdr elt))) a))
```

`(alist-delete  key alist [=])`

`(alist-delete! key alist [=])`

`alist-delete` deletes all associations from alist with the given key, using key-comparison procedure `=`, which defaults to equal?. The dynamic order in which the various applications of = are made is not specified.

Return values may share common tails with the alist argument. The alist is not disordered – elements that appear in the result alist occur in the same order as they occur in the argument alist.

The comparison procedure is used to compare the element keys ki of alist's entries to the key parameter in this way: (= key ki). Thus, one can reliably remove all entries of alist whose key is greater than five with (alist-delete 5 alist <)

alist-delete! is the linear-update variant of alist-delete. It is allowed, but not required, to alter cons cells from the alist parameter to construct the result.

Set operations on lists

These procedures implement operations on sets represented as lists of elements. They all take an = argument used to compare elements of lists. This equality procedure is required to be consistent with eq?. That is, it must be the case that `(eq? x y) => (= x y)`.

Note that this implies, in turn, that two lists that are eq? are also set-equal by any legal comparison procedure. This allows for constant-time determination of set operations on eq? lists.

Be aware that these procedures typically run in time O(n * m) for n- and m-element list arguments. Performance-critical applications operating upon large sets will probably wish to use other data structures and algorithms.

`(lset<= = list1 ...)`

Returns true iff every listi is a subset of listi+1, using = for the element-equality procedure. List A is a subset of list B if every element in A is equal to some element of B. When performing an element comparison, the = procedure's first argument is an element of A; its second, an element of B.

```
(lset<= eq? '(a) '(a b a) '(a b c c)) => #t

(lset<= eq?) => #t             ; Trivial cases
(lset<= eq? '(a)) => #t
```

`(lset= = list1 ...)`

Returns true iff every listi is set-equal to listi+1, using = for the element-equality procedure. "Set-equal" simply means that listi is a subset of listi+1, and listi+1 is a subset of listi. The = procedure's first argument is an element of listi; its second is an element of listi+1.

```
(lset= eq? '(b e a) '(a e b) '(e e b a)) => #t

(lset= eq?) => #t              ; Trivial cases
(lset= eq? '(a)) => #t
```

`(lset-adjoin = list elt1 ...)`

Adds the elti elements not already in the list parameter to the result list. The result shares a common tail with the list parameter. The new elements are added to the front of the list, but no guarantees are made about their order. The = parameter is an equality procedure used to determine if an elti is already a member of list. Its first argument is an element of list; its second is one of the elti.

The list parameter is always a suffix of the result – even if the list parameter contains repeated elements, these are not reduced.

```
(lset-adjoin eq? '(a b c d c e) 'a 'e 'i 'o 'u) => (u o i a b c d c e)
```

`(lset-union = list1 ...)`

Returns the union of the lists, using = for the element-equality procedure.

The union of lists A and B is constructed as follows:

- If A is the empty list, the answer is B (or a copy of B).
- Otherwise, the result is initialised to be list A (or a copy of A).
- Proceed through the elements of list B in a left-to-right order. If b is such an element of B, compare every element r of the current result list to b: (= r b). If all comparisons fail, b is consed onto the front of the result.

However, there is no guarantee that = will be applied to every pair of arguments from A and B. In particular, if A is eq? to B, the operation may immediately terminate.

In the n-ary case, the two-argument list-union operation is simply folded across the argument lists.

```
(lset-union eq? '(a b c d e) '(a e i o u)) =>
        (u o i a b c d e)

;; Repeated elements in LIST1 are preserved.
(lset-union eq? '(a a c) '(x a x)) => (x a a c)

;; Trivial cases
(lset-union eq?) => ()
(lset-union eq? '(a b c)) => (a b c)
```

`(lset-intersection = list1 list2 ...)`

Returns the intersection of the lists, using = for the element-equality procedure.

The intersection of lists A and B is comprised of every element of A that is = to some element of B: (= a b), for a in A, and b in B. Note this implies that an element which appears in B and multiple times in list A will also appear multiple times in the result.

The order in which elements appear in the result is the same as they appear in list1 – that is, lset-intersection essentially filters list1, without disarranging element order. The result may share a common tail with list1.

In the n-ary case, the two-argument list-intersection operation is simply folded across the argument lists. However, the dynamic order in which the applications of = are made is not specified. The procedure may check an element of list1 for membership in every other list before proceeding to consider the next element of list1, or it may completely intersect list1 and list2 before proceeding to list3, or it may go about its work in some third order.

```
(lset-intersection eq? '(a b c d e) '(a e i o u)) => (a e)

;; Repeated elements in LIST1 are preserved.
(lset-intersection eq? '(a x y a) '(x a x z)) => '(a x a)
```

```
(lset-intersection eq? '(a b c)) => (a b c)      ; Trivial case
```

(lset-difference = list1 list2 ...)

Returns the difference of the lists, using = for the element-equality procedure – all the elements of list1 that are not = to any element from one of the other listi parameters.

The = procedure's first argument is always an element of list1; its second is an element of one of the other listi. Elements that are repeated multiple times in the list1 parameter will occur multiple times in the result. The order in which elements appear in the result is the same as they appear in list1 – that is, lset-difference essentially filters list1, without disarranging element order. The result may share a common tail with list1. The dynamic order in which the applications of = are made is not specified. The procedure may check an element of list1 for membership in every other list before proceeding to consider the next element of list1, or it may completely compute the difference of list1 and list2 before proceeding to list3, or it may go about its work in some third order.

```
(lset-difference eq? '(a b c d e) '(a e i o u)) => (b c d)

(lset-difference eq? '(a b c)) => (a b c) ; Trivial case
```

'(lset-xor = list1 . . .)

Returns the exclusive-or of the sets, using = for the element-equality procedure. If there are exactly two lists, this is all the elements that appear in exactly one of the two lists. The operation is associative, and thus extends to the n-ary case – the elements that appear in an odd number of the lists. The result may share a common tail with any of the listi parameters.

More precisely, for two lists A and B, A xor B is a list of

- every element a of A such that there is no element b of B such that (= a b), and
- every element b of B such that there is no element a of A such that (= b a).

However, an implementation is allowed to assume that = is symmetric – that is, that

```
(= a b) => (= b a).
```

This means, for example, that if a comparison (= a b) produces true for some a in A and b in B, both a and b may be removed from inclusion in the result.

In the n-ary case, the binary-xor operation is simply folded across the lists.

```
(lset-xor eq? '(a b c d e) '(a e i o u)) => (d c b i o u)
```

```
;; Trivial cases.
(lset-xor eq?) => ()
(lset-xor eq? '(a b c d e)) => (a b c d e)
```

`(lset-diff+intersection = list1 list2 ...)`

Returns two values – the difference and the intersection of the lists. Is equivalent to:

```
(values (lset-difference = list1 list2 ...)
        (lset-intersection = list1
                             (lset-union = list2 ...)))
```

But can be implemented more efficiently.

The = procedure's first argument is an element of list1; its second is an element of one of the other listi.

Either of the answer lists may share a common tail with list1. This operation essentially partitions list1.

`(lset-union! list1 ...)`

`(lset-intersection! list1 ...)`

`(lset-difference! list1 ...)`

`(lset-xor! list1 ...)`

`(lset-diff+intersection! list1 ...)`

These are linear-update variants. They are allowed, but not required, to use the cons cells in their first list parameter to construct their answer. lset-union! is permitted to recycle cons cells from any of its list arguments.

Primitive side-effects

`(set-car! pair object)`

`(set-cdr! pair object)`

These procedures store object in the car and cdr field of pair, respectively. The value returned is unspecified.

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3) =>  *unspecified*
(set-car! (g) 3) =>  *error*
```

`(srfi srfi-2)`

This is based on SRFI-2.

Abstract

Like an ordinary `and`, an `and-let*` special form evaluates its arguments – expressions – one after another in order, till the first one that yields `#f`. Unlike `and`, however, a non-`#f` result of one expression can be bound to a fresh variable and used in the subsequent expressions. `and-let*` is a cross-breed between `let*` and `and`.

Reference

`and-let*`

`and-let*` is a generalized `and`: it evaluates a sequence of forms one after another till the first one that yields `#f`; the non-`#f` result of a form can be bound to a fresh variable and used in the subsequent forms. ## `(srfi srfi-4)`

This is based on SRFI-4.

Abstract

This SRFI describes a set of datatypes for vectors whose elements are of the same numeric type (signed or unsigned exact integer or inexact real of a given precision). These datatypes support operations analogous to the Scheme vector type, but they are distinct datatypes.

Reference

Signed 8 bits integer

s8vector?

make-s8vector

s8vector

s8vector-length

s8vector-ref

s8vector-set!

s8vector->list

list->s8vector

Signed 16 bits integer

s16vector?

make-s16vector

s16vector

s16vector-length

s16vector-ref

s16vector-set!

s16vector->list

list->s16vector

Signed 32 bits integer

s32vector?

make-s32vector

s32vector

s32vector-length

s32vector-ref

s32vector-set!

s32vector->list

list->s32vector

Signed 64 bits integer

s64vector?

make-s64vector

s64vector

s64vector-length

s64vector-ref

s64vector-set!

s64vector->list

list->s64vector

Unsigned 8 bits integer

u8vector?

make-u8vector

u8vector

u8vector-length

u8vector-ref

u8vector-set!

u8vector->list

list->u8vector

Unsigned 16 bits integer

u16vector?

make-u16vector

u16vector

u16vector-length

u16vector-ref

u16vector-set!

u16vector->list

list->u16vector

Unsigned 32 bits integer

u32vector?

make-u32vector

u32vector

u32vector-length

u32vector-ref

u32vector-set!

u32vector->list

list->u32vector

Unsigned 64 bits integer

u64vector?

make-u64vector

u64vector

u64vector-length

u64vector-ref

u64vector-set!

u64vector->list

list->u64vector

32 bits float

f32vector?

make-f32vector

f32vector

f32vector-length

f32vector-ref

f32vector-set!

f32vector->list

list->f32vector

64 bits float

f64vector?

make-f64vector

f64vector

f64vector-length

f64vector-ref

f64vector-set!

f64vector->list

list->f64vector

`(srfi srfi-5)`

This is based on SRFI-5.

Abstract

The named-`let` incarnation of the `let` form has two slight inconsistencies with the `define` form. As defined, the `let` form makes no accommodation for rest arguments, an issue of functionality and consistency. As defined, the let form does not accommodate signature-style syntax, an issue of aesthetics and consistency. Both issues are addressed here in a manner which is compatible with the traditional `let` form but for minor extensions.

Reference

TODO ## `(srfi srfi-6)`

This is based on SRFI-6.

Abstract

Scheme's i/o primitives are extended by adding three new procedures that

- create an input port from a string,
- create an output port whose contents are accumulated in Scheme's working memory instead of an external file, and
- extract the accumulated contents of an in-memory output port and return them in the form of a string.

Reference

`(open-input-string string)`

Takes a string and returns an input port that delivers characters from the string. The port can be closed by `close-input-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

```
(define p
  (open-input-string "(a . (b . (c . ()))) 34"))

(input-port? p) ;; =>  #t
(read p) ;; => (a b c)
(read p) ;; => 34
(eof-object? (peek-char p)) ;; => #t
```

`(open-output-string)`

Returns an output port that will accumulate characters for retrieval by `get-output-string`. The port can be closed by the procedure `close-output-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

```
(let ((q (open-output-string))
      (x '(a b c)))
  (write (car x) q)
  (write (cdr x) q)
  (get-output-string q)) ;; => "a(b c)"
```

`(get-output-string output-port)`

Given an output port created by `open-output-string`, returns a string consisting of the characters that have been output to the port so far. ## `(srfi srfi-8)`

This is based on SRFI-8.

Abstract

The only mechanism that R5RS provides for binding identifiers to the values of a multiple-valued expression is the primitive `call-with-values`. This SRFI proposes a more concise, more readable syntax for creating such bindings.

Reference

`(receive <formals> <expression> <body>)` syntax

(srfi srfi-9)

This is based on SRFI-9.

Abstract

Syntax for creating new data types, called record types. A predicate, constructor, and field accessors and modifiers are defined for each record type. Each new record type is distinct from all existing types, including other record types and Scheme's predefined types.

Reference

`(define-record-type ...)` syntax

The following:

```
  (define-record-type <pare>
    (kons x y)
    pare?
    (x kar set-kar!)
    (y kdr))
```

Defines `KONS` to be a constructor, `KAR` and `KDR` to be accessors, `SET-KAR!` to be a modifier, and `PARE?` to be a predicate for `<PAREs>`.

```
  (pare? (kons 1 2))         --> #t
  (pare? (cons 1 2))         --> #f
  (kar (kons 1 2))           --> 1
  (kdr (kons 1 2))           --> 2
  (let ((k (kons 1 2)))
    (set-kar! k 3)
    (kar k))                 --> 3
```

(srfi srfi-13)

This is based on SRFI-13.

Abstract

TODO

Reference

Predicates

`string?`

`string-null?`

`string-every`

`string-any`

Constructors

`make-string`

`string`

`string-tabulate`

List & string conversion

`string->list`

`list->string`

`reverse-list->string`

`string-join`

Selection

`string-length`

`string-ref`

`string-copy`

`substring/shared`

`string-copy!`

`string-take`

`string-take-right`

`string-drop`

`string-drop-right`

`string-pad`

`string-pad-right`

`string-trim`

`string-trim-right`

`string-trim-both`

Modification

`string-set!`

`string-fill!`

Comparison

`string-compare`

`string-compare-ci`

`string<>`

`string=`

`string<`

`string>`

`string<=`

`string>=`

`string-ci<>`

`string-ci=`

`string-ci<`

`string-ci>`

`string-ci<=`

`string-ci>=`

`string-hash`

`string-hash-ci`

Prefixes & suffixes

`string-prefix-length`

`string-suffix-length`

`string-prefix-length-ci`

`string-suffix-length-ci`

`string-prefix?`

`string-suffix?`

`string-prefix-ci?`

`string-suffix-ci?`

Searching

`string-index`

`string-index-right`

`string-skip`

`string-skip-right`

`string-count`

`string-contains`

`string-contains-ci`

Alphabetic case mapping

`string-titlecase`

`string-upcase`

`string-downcase`

`string-titlecase!`

`string-upcase!`

`string-downcase!`

Reverse & append

`string-reverse`

`string-reverse!`

`string-append`

`string-concatenate`

`string-concatenate/shared`

`string-append/shared`

`string-concatenate-reverse`

`string-concatenate-reverse/shared`

Fold, unfold & map

`string-map`

`string-map!`

`string-fold`

`string-fold-right`

`string-unfold`

`string-unfold-right`

`string-for-each`

`string-for-each-index`

Replicate & rotate

`xsubstring`

`string-xcopy!`

Miscellaneous: insertion, parsing

`string-replace`

`string-tokenize`

Filtering & deleting

`string-filter`

`string-delete`

Low-level procedures

`string-parse-start+end`

`string-parse-final-start+end`

`let-string-start+end`

`check-substring-spec`

`substring-spec-ok?`

`make-kmp-restart-vector`

`kmp-step`

`string-kmp-partial-search`

`(srfi srfi-14)`

This library is based on SRFI-14.

Abstract

The ability to efficiently represent and manipulate sets of characters is an unglamorous but very useful capability for text-processing code – one that tends to pop up in the definitions of other libraries.

Reference

->char-set

char-set

char-set->list

char-set->string

char-set-adjoin

char-set-adjoin!

char-set-any

char-set-complement

char-set-complement!

char-set-contains?

char-set-copy

char-set-count

char-set-cursor

char-set-cursor-next

char-set-delete

char-set-delete!

char-set-diff+intersection

char-set-diff+intersection!

char-set-difference

char-set-difference!

char-set-every

char-set-filter

char-set-filter!

char-set-fold

char-set-for-each

char-set-hash

char-set-intersection

char-set-intersection!

char-set-map

char-set-ref

char-set-size

char-set-unfold

char-set-unfold!

char-set-union

char-set-union!

char-set-xor

char-set-xor!

char-set:ascii

char-set:blank

char-set:digit

char-set:empty

char-set:full

char-set:graphic

char-set:hex-digit

char-set:iso-control

char-set:letter

char-set:letter+digit

char-set:lower-case

char-set:printing

char-set:punctuation

char-set:symbol

char-set:title-case

char-set:upper-case

char-set:whitespace

char-set<=

char-set=

char-set?

end-of-char-set?

list->char-set

list->char-set!

string->char-set

string->char-set!

ucs-range->char-set

ucs-range->char-set!)

(srfi srfi-16)

(case-lambda clause1 clause2 ...) syntax

Each clause is of the form `(formals body)`, where `formals` and `body` have the same syntax as in a lambda expression.

A case-lambda expression evaluates to a procedure that accepts a variable number of arguments and is lexically scoped in the same manner as a procedure resulting from a lambda expression. When the procedure is called, the first clause for which the arguments agree with `formals` is selected, where agreement is specified as for the `formals` of a lambda expression. The variables of `formals` are bound to fresh locations, the values of the arguments are stored in those locations, the `body` is evaluated in the extended environment, and the results of `body` are returned as the results of the procedure call.

It is an error for the arguments not to agree with the `formals` of any clause‘.

Example:

```
(define add1
  (case-lambda
    ((a) (add1 a 0))
    ((a b) (+ 1 a b))))

(add1 1) ;; => 2
(add1 1 2) ;; => 4
```

(srfi srfi-17)

This is based on SRFI-17.

Abstract

Allow procedure calls that evaluate to the "value of a location" to be used to set the value of the location, when used as the first operand of `set!`. For example:

```
(set! (car x) (car y))
```

becomes equivalent to

```
(set-car! x (car y))
```

Reference

```
(set! (car x) v) == (set-car! x v)
(set! (cdr x) v) == (set-cdr! x v)
(set! (caar x) v) == (set-car! (car x) v)
(set! (cadr x) v) == (set-car! (cdr x) v)
....
(set! (caXXr x) v) == (set-car! (cXXr x) v)
```

```
(set! (cdXXr x) v) == (set-cdr! (cXXr x) v)
(set! (string-ref x i) v) == (string-set! x i v)
(set! (vector-ref x i) v) == (vector-set! x i v)
```

(srfi srfi-19)

This is based on SRFI-19.

Abstract

TODO

Reference

`time-duration`

`time-monotonic`

`time-process`

`time-tai`

`time-thread`

`time-utc`

`current-date`

`current-julian-day`

`current-modified-julian-day`

`current-time`

`time-resolution`

`make-time`

`time?`

`time-type`

`time-nanosecond`

`time-second`

`set-time-type!`

`set-time-nanosecond!`

`set-time-second!`

`copy-time`

`time<=?`

`time<?`

`time=?`

`time>=?`

`time>?`

`time-difference`

`time-difference!`

`add-duration`

`add-duration!`

`subtract-duration`

`subtract-duration!`

`make-date`

`date?`

`date-nanosecond`

`date-second`

`date-minute`

date-hour

date-day

date-month

date-year

date-zone-offset

date-year-day

date-week-day

date-week-number

date->julian-day

date->modified-julian-day

date->time-monotonic

date->time-tai

date->time-utc

julian-day->date

julian-day->time-monotonic

julian-day->time-tai

julian-day->time-utc

modified-julian-day->date

modified-julian-day->time-monotonic

modified-julian-day->time-tai

modified-julian-day->time-utc

`time-monotonic->date`

`time-monotonic->julian-day`

`time-monotonic->modified-julian-day`

`time-monotonic->time-tai`

`time-monotonic->time-tai!`

`time-monotonic->time-utc`

`time-monotonic->time-utc!`

`time-tai->date`

`time-tai->julian-day`

`time-tai->modified-julian-day`

`time-tai->time-monotonic`

`time-tai->time-monotonic!`

`time-tai->time-utc`

`time-tai->time-utc!`

`time-utc->date`

`time-utc->julian-day`

`time-utc->modified-julian-day`

`time-utc->time-monotonic`

`time-utc->time-monotonic!`

`time-utc->time-tai`

`time-utc->time-tai!`

`date->string`

`string->date`

`(srfi srfi-23)`

This is based on SRFI-23.

Abstract

A mechanism is proposed to allow Scheme code to report errors and abort execution. The proposed mechanism is already implemented in several Scheme systems and can be implemented, albeit imperfectly, in any R5RS conforming Scheme.

Reference

`(error <reason> [arg ...])`

`(srfi srfi-25)`

This is based on SRFI-25.

Abstract

A core set of procedures for creating and manipulating heterogeneous multidimensional arrays is proposed. The design is consistent with the rest of Scheme and independent of other container data types. It provides easy sharing of parts of an array as other arrays without copying, encouraging a declarative style of programming.

Reference

TODO ## `(srfi srfi-26)`

This is based on SRFI-26.

Abstract

When programming in functional style, it is frequently necessary to specialize some of the parameters of a multi-parameter procedure. For example, from the binary operation cons one might want to obtain the unary operation (lambda (x) (cons 1 x)). This specialization of parameters is also known as “partial application”, “operator section” or “projection”.

Reference

`(cut ...)` syntax

`(cute ...)` syntax

`(srfi srfi-28)`

This is based on SRFI-28.

Abstract

A method of interpreting a Scheme string which contains a number of escape sequences that are replaced with other string data according to the semantics of each sequence. Also called string interpolation.

Reference

`(format format-string [obj ...])`

Accepts a message template (a Scheme string), and processes it, replacing any escape sequences in order with one or more characters, the characters themselves dependent on the semantics of the escape sequence encountered.

An escape sequence is a two character sequence in the string where the first character is a tilde '~'. Each escape code's meaning is as follows:

- `~a` The corresponding value is inserted into the string as if printed with display.
- `~s` The corresponding value is inserted into the string as if printed with write.
- `~%` A newline is inserted.
- `~~` A tilde `~` is inserted.

`~a` and `~s`, when encountered, require a corresponding Scheme value to be present after the format string. The values provided as operands are used by the escape sequences in order. It is an error if fewer values are provided than escape sequences that require them.

`~%` and `~~` require no corresponding value.

Examples:

```
(format "Hello, ~a" "World!")
;; => "Hello, World!"

(format "Error, list is too short: ~s~%" '(one "two" 3))
;; => "Error, list is too short: (one \"two\" 3))"
```

`(srfi srfi-29)`

This is based on SRFI-29.

Abstract

An interface to retrieving and displaying locale sensitive messages. A Scheme program can register one or more translations of templated messages, and then write Scheme code that can transparently retrieve the appropriate message for the locale under which the Scheme system is running.

Reference

`current-language`

`current-country`

`current-locale-details`

`declare-bundle!`

`store-bundle`

`store-bundle!`

`load-bundle!`

`localized-template`

`(srfi srfi-31)`

This is based on SRFI-31.

Abstract

TODO

Reference

TODO ## `(srfi srfi-34)`

This is based on SRFI-34.

Abstract

TODO

Reference

`with-exception-handler`

`guard`

`raise`

(srfi srfi-35)

This is based on SRFI-35.

Abstract

Defines constructs for creating and inspecting condition types and values. A condition value encapsulates information about an exceptional situation, or exception. This SRFI also defines a few basic condition types.

Reference

`make-condition-type`

`condition-type?`

`condition-has-type?`

`condition-ref`

`make-compound-condition`

`extract-condition`

`define-condition-type`

`&condition`

`make-condition`

`condition?`

`condition`

`&serious`

`serious-condition?`

`&error`

`error?`

`&message`

`message-condition?`

`condition-message`

`(srfi srfi-37)`

This is based on SRFI-37.

Abstract

Many operating systems make the set of argument strings used to invoke a program available (often following the program name string in an array called argv). Most programs need to parse and process these argument strings in one way or another. This SRFI describes a set of procedures that support processing program arguments according to POSIX and GNU C Library Reference Manual guidelines.

Reference

`args-fold`

`option`

`option?`

`option-names`

`option-required-arg?`

`option-optional-arg?`

`option-processor`

`(srfi srfi-38)`

This is based on SRFI-38.

Abstract

TODO

Reference

`(write-with-shared-structure obj [port [optarg]])`

`(read-with-shared-structure [port])`

(srfi srfi-39)

This is based on SRFI-39.

Abstract

This SRFI defines parameter objects, the procedure make-parameter to create parameter objects and the parameterize special form to dynamically bind parameter objects. In the dynamic environment, each parameter object is bound to a cell containing the value of the parameter. When a procedure is called the called procedure inherits the dynamic environment from the caller. The parameterize special form allows the binding of a parameter object to be changed for the dynamic extent of its body.

Reference

`make-parameter`

`parameterize`

(srfi srfi-41)

This is based on SRFI-41.

Abstract

Streams, sometimes called lazy lists, are a sequential data structure containing elements computed only on demand. A stream is either null or is a pair with a stream in its cdr. Since elements of a stream are computed only when accessed, streams can be infinite. Once computed, the value of a stream element is cached in case it is needed again.

Reference

`stream-null`

`stream-cons`

`stream?`

`stream-null?`

`stream-pair?`

`stream-car`

`stream-cdr`

`stream-lambda`

`define-stream`

`list->stream`

`port->stream`

`stream`

`stream->list`

`stream-append`

`stream-concat`

`stream-constant`

`stream-drop`

`stream-drop-while`

`stream-filter`

`stream-fold`

`stream-for-each`

`stream-from`

`stream-iterate`

`stream-length`

`stream-let`

`stream-map`

`stream-match`

`stream-of`

`stream-range`

`stream-ref`

`stream-reverse`

`stream-scan`

`stream-take`

`stream-take-while`

`stream-unfold`

`stream-unfolds`

`stream-zip`

`(srfi srfi-42)`

This is based on SRFI-42.

Abstract

TODO

Reference

`do-ec`

`list-ec`

`append-ec`

`string-ec`

`string-append-ec`

`vector-ec`

`vector-of-length-ec`

`sum-ec`

`product-ec`

`min-ec`

`max-ec`

`any?-ec`

`every?-ec`

`first-ec`

`last-ec`

`fold-ec`

`fold3-ec`

`:`

`:list`

`:string`

`:vector`

`:integers`

`:range`

`:real-range`

`:char-range`

`:port`

`:dispatched`

`:do`

`:let`

`:parallel`

`:while`

`:until`

`:-dispatch-ref`

`:-dispatch-set!`

`make-initial-:-dispatch`

`dispatch-union`

`:generator-proc`

`(srfi srfi-43)`

This is based on SRFI-43.

Abstract

TODO

Reference

TODO ## `(srfi srfi-45)`

This is based on SRFI-45.

Abstract

TODO

Reference

`delay`

`lazy`

`force`

`eager`

`(srfi srfi-48)`

This is based on SRFI-48.

Abstract

TODO

Reference

TODO ## `(srfi srfi-51)`

This is based on SRFI-51.

Abstract

TODO

Reference

TODO ## `(srfi srfi-54)`

This is based on SRFI-54.

Abstract

TODO

Reference

TODO ## `(srfi srfi-60)`

This is based on SRFI-60.

Abstract

TODO

Reference

logand

bitwise-and

logior

bitwise-ior

logxor

bitwise-xor

lognot

bitwise-not

bitwise-if

bitwise-merge

logtest

any-bits-set?

logcount

bit-count

integer-length

log2-binary-factors

first-set-bit

logbit?

bit-set?

copy-bit

bit-field

copy-bit-field

ash

arithmetic-shift

rotate-bit-field

reverse-bit-field

integer->list

integer->list

list->integer

booleans->integer

`(srfi srfi-61)`

This is based on SRFI-61.

Abstract

TODO

Reference

TODO ## `(srfi srfi-67)`

This is based on SRFI-67.

Abstract

TODO

Reference

TODO ## `(srfi srfi-69)`

This is based on SRFI-69.

Abstract

TODO

Reference

TODO ## `(srfi srfi-98)`

This is based on SRFI-98.

Abstract

TODO

Reference

TODO ## `(srfi srfi-101)`

This library is based on SRFI-101.

Abstract

TODO

Reference

TODO ## `(srfi srfi-111)`

This library is based on SRFI-111.

Abstract

Boxes are objects with a single mutable state. Several Schemes have them, sometimes called cells. A constructor, predicate, accessor, and mutator are provided.

Reference

`(box value)`

Constructor. Returns a newly allocated box initialized to value.

`(box? object)`

Predicate. Returns `#t` if object is a box, and `#f` otherwise.

`(unbox box)`

Accessor. Returns the current value of box.

`(set-box! box value)`

Mutator. Changes box to hold value. ## `(srfi srfi-113)`

This library is based on SRFI-113.

Abstract

Sets and bags (also known as multisets) are unordered collections that can contain any Scheme object. Sets enforce the constraint that no two elements can be the same in the sense of the set's associated equality predicate; bags do not.

Reference

TODO

Set

set

set-unfold

set?

set-contains?

set-empty?

set-disjoint?

set-member

set-element-comparator

set-adjoin

set-adjoin!

set-replace

set-replace!

set-delete

set-delete!

set-delete-all

set-delete-all!

set-search!

set-size

set-find

set-count

set-any?

set-every?

set-map

set-for-each

set-fold

set-filter

set-remove

set-remove

set-partition

set-filter!

set-remove!

set-partition!

set-copy

set->list

list->set

list->set!

set=?

set<?

set>?

set<=?

set>=?

set-union

set-intersection

set-difference

set-xor

set-union!

set-intersection!

set-difference!

set-xor!

set-comparator

Bag

bag

bag-unfold

bag?

bag-contains?

bag-empty?

bag-disjoint?

bag-member

bag-element-comparator

bag-adjoin

bag-adjoin!

bag-replace

bag-replace!

bag-delete

bag-delete!

bag-delete-all

bag-delete-all!

bag-search!

bag-size

bag-find

bag-count

bag-any?

bag-every?

bag-map

bag-for-each

bag-fold

bag-filter

bag-remove

bag-partition

bag-filter!

bag-remove!

bag-partition!

bag-copy

bag->list

list->bag

list->bag!

bag=?

bag<?

bag>?

bag<=?

bag>=?

bag-union

bag-intersection

bag-difference

bag-xor

bag-union!

bag-intersection!

bag-difference!

bag-xor!

bag-comparator

bag-sum

bag-sum!

bag-product

bag-product!

bag-unique-size

bag-element-count

bag-for-each-unique

bag-fold-unique

bag-increment!

bag-decrement!

bag->set

set->bag

set->bag!

bag->alist

alist->bag

`(srfi srfi-115)`

This library is based on SRFI-115.

Abstract

TODO

Reference

TODO ## `(srfi srfi-116)`

This library is based on SRFI-116.

Abstract

TODO

Reference

TODO ## `(srfi srfi-117)`

This library is based on SRFI-117.

Abstract

TODO

Reference

TODO ## `(srfi srfi-124)`

This library is based on SRFI-124.

Abstract

TODO

Reference

`(ephemeron? obj)`

`make-ephemeron`

`ephemeron-broken?`

`ephemeron-key`

`ephemeron-value`

`(srfi srfi-125)`

This library is based on srfi-125.

The library doesn't implement deprecated features. Application must rely on `(scheme comparator)` to specify equal predicate and hash function.

Abstract

This SRFI defines an interface to hash tables, which are widely recognized as a fundamental data structure for a wide variety of applications. A hash table is a data structure that:

- Is disjoint from all other types.
- Provides a mapping from objects known as keys to corresponding objects known as values.
  - Keys may be any Scheme objects in some kinds of hash tables, but are restricted in other kinds.
  - Values may be any Scheme objects.
- Has no intrinsic order for the key-value associations it contains.

- Provides an equality predicate which defines when a proposed key is the same as an existing key. No table may contain more than one value for a given key.
- Provides a hash function which maps a candidate key into a non-negative exact integer.
- Supports mutation as the primary means of setting the contents of a table.
- Provides key lookup and destructive update in (expected) amortized constant time, provided a satisfactory hash function is available.
- Does not guarantee that whole-table operations work in the presence of concurrent mutation of the whole hash table (values may be safely mutated).

Reference

Constructors

`(make-hash-table comparator . args)`

Returns a newly allocated hash table using `(scheme comparator)` object `COMPARATOR`. For the time being, `ARGS` is ignored.

`(hash-table comparator [key value] ...)`

Returns a newly allocated hash table using `(scheme comparator)` object `COMPARATOR`. For each pair of arguments, an association is added to the new hash table with key as its key and value as its value. If the same key (in the sense of the equality predicate) is specified more than once, it is an error.

`(hash-table-unfold stop? mapper successor seed comparator args ...)`

Create a new hash table as if by `make-hash-table` using `comparator` and the `args`. If the result of applying the predicate `stop?` to `seed` is true, return the hash table. Otherwise, apply the procedure `mapper` to `seed`. `mapper` returns two values, which are inserted into the hash table as the key and the value respectively. Then get a new `seed` by applying the procedure `successor` to `seed`, and repeat this algorithm.

`(alist->hash-table alist comparator arg ...)`

Returns a newly allocated hash-table as if by `make-hash-table` using `comparator` and the `args`. It is then initialized from the associations of `alist`. Associations earlier in the list take precedence over those that come later.

Predicates

`(hash-table? obj)`

Returns #t if obj is a hash table, and #f otherwise

`(hash-table-contains? hash-table key)`

Returns #t if there is any association to key in hash-table, and #f otherwise.

`(hash-table-empty? hash-table)`

Returns #t if hash-table contains no associations, and #f otherwise.

`(hash-table=? value-comparator hash-table1 hash-table2)`

Returns #t if hash-table1 and hash-table2 have the same keys (in the sense of their common equality predicate) and each key has the same value (in the sense of value-comparator), and #f otherwise.

`(hash-table-mutable? hash-table)`

Returns #t if the hash table is mutable.

Accessors

The following procedures, given a key, return the corresponding value.

`(hash-table-ref hash-table key [failure [success]])`

Extracts the value associated to key in hash-table, invokes the procedure success on it, and returns its result; if success is not provided, then the value itself is returned. If key is not contained in hash-table and failure is supplied, then failure is invoked on no arguments and its result is returned.

`(hash-table-ref/default hash-table key default)`

TODO

Mutators

The following procedures alter the associations in a hash table either unconditionally, or conditionally on the presence or absence of a specified key. It is an error to add an association to a hash table whose key does not satisfy the type test predicate of the comparator used to create the hash table.

`(hash-table-set! hash-table key value ...)`

Repeatedly mutates hash-table, creating new associations in it by processing the arguments from left to right. The args alternate between keys and values. Whenever there is a previous association for a key, it is deleted. It is an error if the type check procedure of the comparator of hash-table, when invoked on a key, does not return #t. Likewise, it is an error if a key is not a valid argument to the equality predicate of hash-table. Returns an unspecified value.

`(hash-table-delete! hash-table key ...)`

Deletes any association to each key in hash-table and returns the number of keys that had associations.

`(hash-table-intern! hash-table key failure)`

Effectively invokes hash-table-ref with the given arguments and returns what it returns. If key was not found in hash-table, its value is set to the result of calling failure.

`(hash-table-update! hash-table key updater [failure [success]])`

TODO:

`(hash-table-pop! hash-table)`

Chooses an arbitrary association from hash-table and removes it, returning the key and value as two values.

It is an error if hash-table is empty.

`(hash-table-clear! hash-table)`

Delete all the associations from hash-table.

The whole hash table

These procedures process the associations of the hash table in an unspecified order.

`(hash-table-size hash-table)`

Returns the number of associations in hash-table as an exact integer.

`(hash-table-keys hash-table)`

Returns a newly allocated list of all the keys in hash-table.

`(hash-table-values hash-table)`

Returns a newly allocated list of all the keys in hash-table.

`(hash-table-entries hash-table)`

Returns two values, a newly allocated list of all the keys in hash-table and a newly allocated list of all the values in hash-table in the corresponding order.

`(hash-table-find proc hash-table failure)`

For each association of hash-table, invoke proc on its key and value. If proc returns true, then hash-table-find returns what proc returns. If all the calls to proc return #f, return the result of invoking the thunk failure.

`(hash-table-count pred hash-table)`

For each association of hash-table, invoke pred on its key and value. Return the number of calls to pred which returned true.

Mapping and folding

These procedures process the associations of the hash table in an unspecified order.

`(hash-table-map proc comparator hash-table)`

Returns a newly allocated hash table as if by `(make-hash-table comparator)`. Calls `PROC` for every association in `hash-table` with the value of the association. The key of the association and the result of invoking `proc` are entered into the new hash table. Note that this is not the result of lifting mapping over the domain of hash tables, but it is considered more useful.

If comparator recognizes multiple keys in the hash-table as equivalent, any one of such associations is taken.

`(hash-table-for-each proc hash-table)`

Calls proc for every association in hash-table with two arguments: the key of the association and the value of the association. The value returned by proc is discarded. Returns an unspecified value.

`(hash-table-map! proc hash-table)`

Calls proc for every association in hash-table with two arguments: the key of the association and the value of the association. The value returned by proc is used to update the value of the association. Returns an unspecified value.

`(hash-table-map->list proc hash-table)`

Calls proc for every association in hash-table with two arguments: the key of the association and the value of the association. The values returned by the invocations of proc are accumulated into a list, which is returned.

`(hash-table-fold proc seed hash-table)`

Calls proc for every association in hash-table with three arguments: the key of the association, the value of the association, and an accumulated value val. Val is seed for the first invocation of procedure, and for subsequent invocations of proc, the returned value of the previous invocation. The value returned by hash-table-fold is the return value of the last invocation of proc.

`(hash-table-prune! proc hash-table)`

Calls proc for every association in hash-table with two arguments, the key and the value of the association, and removes all associations from hash-table for which proc returns true. Returns an unspecified value.

Copying and conversion

`(hash-table-copy hash-table [mutable?])`

Returns a newly allocated hash table with the same properties and associations as hash-table. If the second argument is present and is true, the new hash table is mutable. Otherwise it is immutable provided that the implementation supports immutable hash tables.

`(hash-table-empty-copy hash-table)`

Returns a newly allocated mutable hash table with the same properties as hash-table, but with no associations.

`(hash-table->alist hash-table)`

Returns an alist with the same associations as hash-table in an unspecified order.

Hash tables as sets

`(hash-table-union! hash-table1 hash-table2)`

Adds the associations of hash-table2 to hash-table1 and returns hash-table1. If a key appears in both hash tables, its value is set to the value appearing in hash-table1. Returns hash-table1.

`(hash-table-intersection! hash-table1 hash-table2)`

Deletes the associations from hash-table1 whose keys don't also appear in hash-table2 and returns hash-table1.

`(hash-table-difference! hash-table1 hash-table2)`

Deletes the associations of hash-table1 whose keys are also present in hash-table2 and returns hash-table1.

`(hash-table-xor! hash-table1 hash-table2)`

Deletes the associations of hash-table1 whose keys are also present in hash-table2, and then adds the associations of hash-table2 whose keys are not present in hash-table1 to hash-table1. Returns hash-table1. ## `(srfi srfi-127)`

This library is based on SRFI-127.

Abstract

TODO

Reference

TODO ## `(srfi srfi-128)`

This library is based on SRFI-128.

Abstract

A comparator is an object of a disjoint type. It is a bundle of procedures that are useful for comparing two objects either for equality or for ordering. There are four procedures in the bundle:

- The type test predicate returns #t if its argument has the correct type to be passed as an argument to the other three procedures, and #f otherwise.
- The equality predicate returns #t if the two objects are the same in the sense of the comparator, and #f otherwise. It is the programmer's responsibility to ensure that it is reflexive, symmetric, transitive, and can handle any arguments that satisfy the type test predicate.
- The comparison procedure returns -1, 0, or 1 if the first object precedes the second, is equal to the second, or follows the second, respectively, in a total order defined by the comparator. It is the programmer's responsibility to ensure that it is reflexive, weakly antisymmetric, transitive, can handle any arguments that satisfy the type test predicate, and returns 0 iff the equality predicate returns #t.

- The hash function takes one argument, and returns an exact non-negative integer. It is the programmer's responsibility to ensure that it can handle any argument that satisfies the type test predicate, and that it returns the same value on two objects if the equality predicate says they are the same (but not necessarily the converse).

It is also the programmer's responsibility to ensure that all four procedures provide the same result whenever they are applied to the same object(s) (in the sense of eqv?), unless the object(s) have been mutated since the last invocation. In particular, they must not depend in any way on memory addresses in implementations where the garbage collector can move objects in memory.

Limitations

The comparator objects defined in this library are not applicable to circular structure or to NaNs or objects containing them. Attempts to pass any such objects to any procedure defined here, or to any procedure that is part of a comparator defined here, is an error except as otherwise noted.

Reference

Predicates

`(comparator? obj)`

Returns #t if obj is a comparator, and #f otherwise.

`(comparator-comparison-procedure? comparator)`

Returns #t if comparator has a supplied comparison procedure, and #f otherwise.

`(comparator-hash-function? comparator)`

Returns #t if comparator has a supplied hash function, and #f otherwise.

Standard comparators

`boolean-comparator`

Compares booleans using the total order #f < #t.

`char-comparator`

Compares characters using the total order implied by char<?. On R6RS and R7RS systems, this is Unicode codepoint order.

`char-ci-comparator`

Compares characters using the total order implied by char-ci<? On R6RS and R7RS systems, this is Unicode codepoint order after the characters have been folded to lower case.

`string-comparator`

Compares strings using the total order implied by string<?. Note that this order is implementation-dependent.

`string-ci-comparator`

Compares strings using the total order implied by string-ci<?. Note that this order is implementation-dependent.

`symbol-comparator`

Compares symbols using the total order implied by applying symbol->string to the symbols and comparing them using the total order implied by string<?. It is not a requirement that the hash function of symbol-comparator be consistent with the hash function of string-comparator, however.

`exact-integer-comparator`

`integer-comparator`

`rational-comparator`

`real-comparator`

`complex-comparator`

`number-comparator`

These comparators compare exact integers, integers, rational numbers, real numbers, complex numbers, and any numbers using the total order implied by <. They must be compatible with the R5RS numerical tower in the following sense: If S is a subtype of the numerical type T and the two objects are members of S , then the equality predicate and comparison procedures (but not necessarily the hash function) of S-comparator and T-comparator compute the same results on those objects.

Since non-real numbers cannot be compared with <, the following least-surprising ordering is defined: If the real parts are < or >, so are the numbers; otherwise, the numbers are ordered by their imaginary parts. This can still produce surprising results if one real part is exact and the other is inexact.

`pair-comparator`

This comparator compares pairs using default-comparator (see below) on their cars. If the cars are not equal, that value is returned. If they are equal, default-comparator is used on their cdrs and that value is returned.

`list-comparator`

This comparator compares lists lexicographically, as follows:

- The empty list compares equal to itself.
- The empty list compares less than any non-empty list.
- Two non-empty lists are compared by comparing their cars. If the cars are not equal when compared using default-comparator (see below), then the result is the result of that comparison. Otherwise, the cdrs are compared using list-comparator.

`vector-comparator`

`bytevector-comparator`

These comparators compare vectors and bytevectors by comparing their lengths. A shorter argument is always less than a longer one. If the lengths are equal, then each element is compared in turn using default-comparator (see below) until a pair of unequal elements is found, in which case the result is the result of that comparison. If all elements are equal, the arguments are equal.

If the implementation does not support bytevectors, bytevector-comparator has a type testing procedure that always returns #f.

The default comparator

`default-comparator`

This is a comparator that accepts any two Scheme values (with the exceptions listed in the Limitations section) and orders them in some implementation-defined way, subject to the following conditions:

- The following ordering between types must hold: the empty list precedes pairs, which precede booleans, which precede characters, which precede strings, which precede symbols, which precede numbers, which precede vectors, which precede bytevectors, which precede all other objects.
- When applied to pairs, booleans, characters, strings, symbols, numbers, vectors, or bytevectors, its behavior must be the same as pair-comparator, boolean-comparator, character-comparator, string-comparator, symbol-comparator, number-comparator, vector-comparator, and bytevector-comparator respectively. The same should be true when

applied to an object or objects of a type for which a standard comparator is defined elsewhere.

- Given disjoint types a and b, one of three conditions must hold:
  - All objects of type a compare less than all objects of type b.
  - All objects of type a compare greater than all objects of type b.
  - All objects of either type a or type b compare equal to each other. This is not permitted for any of the standard types mentioned above.

Comparator constructors

`(make-comparator type-test equality compare hash)`

Returns a comparator which bundles the type-test, equality, compare, and hash procedures provided. As a convenience, the following additional values are accepted:

- If type-test is #t, a type-test procedure that accepts any arguments is provided.
- If equality is #t, an equality predicate is provided that returns #t iff compare returns 0.
- If compare or hash is #f, a procedure is provided that signals an error on application. The predicates comparator-comparison-procedure? and/or comparator-hash-function?, respectively, will return #f in these cases.

`(make-inexact-real-comparator epsilon rounding nan-handling)`

Returns a comparator that compares inexact real numbers including NaNs as follows: if after rounding to the nearest epsilon they are the same, they compare equal; otherwise they compare as specified by <. The direction of rounding is specified by the rounding argument, which is either a procedure accepting two arguments (the number and epsilon, or else one of the symbols floor, ceiling, truncate, or round.

The argument nan-handling specifies how to compare NaN arguments to non-NaN arguments. If it is a procedure, the procedure is invoked on the other argument if either argument is a NaN. If it is the symbol min, NaN values precede all other values; if it is the symbol max, they follow all other values, and if it is the symbol error, an error is signaled if a NaN value is compared. If both arguments are NaNs, however, they always compare as equal.

`(make-list-comparator element-comparator)`

`(make-vector-comparator element-comparator)`

(make-bytevector-comparator element-comparator)

These procedures return comparators which compare two lists, vectors, or bytevectors in the same way as list-comparator, vector-comparator, and bytevector-comparator respectively, but using element-comparator rather than default-comparator.

If the implementation does not support bytevectors, the result of invoking make-bytevector-comparator is a comparator whose type testing procedure always returns #f.

(make-listwise-comparator type-test element-comparator empty? head tail)

Returns a comparator which compares two objects that satisfy type-test as if they were lists, using the empty? procedure to determine if an object is empty, and the head and tail procedures to access particular elements.

(make-vectorwise-comparator type-test element-comparator length ref)

Returns a comparator which compares two objects that satisfy type-test as if they were vectors, using the length procedure to determine the length of the object, and the ref procedure to access a particular element.

(make-car-comparator comparator)

Returns a comparator that compares pairs on their cars alone using comparator.

(make-cdr-comparator comparator)

Returns a comparator that compares pairs on their cdrs alone using comparator.

(make-pair-comparator car-comparator cdr-comparator)

Returns a comparator that compares pairs first on their cars using car-comparator. If the cars are equal, it compares the cdrs using cdr-comparator.

(make-improper-list-comparator element-comparator)

Returns a comparator that compares arbitrary objects as follows: the empty list precedes all pairs, which precede all other objects. Pairs are compared as if with (make-pair-comparator element-comparator element-comparator). All other objects are compared using element-comparator.

(make-selecting-comparator comparator1 comparator2 ...)

Returns a comparator whose procedures make use of the comparators as follows:

The type test predicate passes its argument to the type test predicates of comparators in the sequence given. If any of them returns #t, so does the type test predicate; otherwise, it returns #f.

The arguments of the equality, compare, and hash functions are passed to the type test predicate of each comparator in sequence. The first comparator whose type test predicate is satisfied on all the arguments is used when comparing those arguments. All other comparators are ignored. If no type test predicate is satisfied, an error is signaled.

(make-refining-comparator comparator1 comparator2 ...)

Returns a comparator that makes use of the comparators in the same way as make-selecting-comparator, except that its procedures can look past the first comparator whose type test predicate is satisfied. If the comparison procedure of that comparator returns zero, then the next comparator whose type test predicate is satisfied is tried in place of it until one returns a non-zero value. If there are no more such comparators, then the comparison procedure returns zero. The equality predicate is defined in the same way. If no type test predicate is satisfied, an error is signaled.

The hash function of the result returns a value which depends, in an implementation-defined way, on the results of invoking the hash functions of the comparators whose type test predicates are satisfied on its argument. In particular, it may depend solely on the first or last such hash function. If no type test predicate is satisfied, an error is signaled.

This procedure is analogous to the expression type refine-compare from SRFI 67.

(make-reverse-comparator comparator)

Returns a comparator that behaves like comparator, except that the compare procedure returns 1, 0, and -1 instead of -1, 0, and 1 respectively. This allows ordering in reverse.

(make-debug-comparator comparator)

Returns a comparator that behaves exactly like comparator, except that whenever any of its procedures are invoked, it verifies all the programmer responsibilities (except stability), and an error is signaled if any of them are violated. Because it requires three arguments, transitivity is not tested on the first call to a debug comparator; it is tested on all future calls using an arbitrarily chosen argument from the previous invocation. Note that this may cause unexpected storage leaks.

Wrapped equality predicates

`eq-comparator`

`eqv-comparator`

`equal-comparator`

The equality predicates of these comparators are eq?, eqv?, and equal? respectively. When their comparison procedures are applied to non-equal objects, their behavior is implementation-defined. The type test predicates always return #t.

These comparators accept circular structure (in the case of equal-comparator, provided the implementation's equal does so) and NaNs.

Accessors

`(comparator-type-test-procedure comparator)`

Returns the type test predicate of comparator.

`(comparator-equality-predicate comparator)`

Returns the equality predicate of comparator.

`(comparator-comparison-procedure comparator)`

Returns the comparison procedure of comparator.

`(comparator-hash-function comparator)`

Returns the hash function of comparator.

Primitive applicators

`(comparator-test-type comparator obj)`

Invokes the type test predicate of comparator on obj and returns what it returns.

`(comparator-check-type comparator obj)`

Invokes the type test predicate of comparator on obj and returns true if it returns true and signals an error otherwise.

`(comparator-equal? comparator obj1 obj2)`

Invokes the equality predicate of comparator on obj1 and obj2 and returns what it returns.

`(comparator-compare comparator obj1 obj2)`

Invokes the comparison procedure of comparator on obj1 and obj2 and returns what it returns.

`(comparator-hash comparator obj)`

Invokes the hash function of comparator on obj and returns what it returns.

Comparison procedure constructors

`(make-comparison< lt-pred)`

`(make-comparison> gt-pred)`

`(make-comparison<= le-pred)`

`(make-comparison>= ge-pred)`

`(make-comparison=/< eq-pred lt-pred)`

`(make-comparison=/> eq-pred gt-pred)`

These procedures return a comparison procedure, given a less-than predicate, a greater-than predicate, a less-than-or-equal-to predicate, a greater-than-or-equal-to predicate, or the combination of an equality predicate and either a less-than or a greater-than predicate.

Comparison syntax

The following expression types allow the convenient use of comparison procedures.

`(if3 <expr> <less> <equal> <greater>)`

The expression `<expr>` is evaluated; it will typically, but not necessarily, be a call on a comparison procedure. If the result is -1, `<less>` is evaluated and its value(s) are returned; if the result is 0, `<equal>` is evaluated and its value(s) are returned; if the result is 1, `<greater>` is evaluated and its value(s) are returned. Otherwise an error is signaled.

`(if=? <expr> <consequent> [ <alternate> ])`

`(if<? <expr> <consequent> [ <alternate> ])`

`(if>? <expr> <consequent> [ <alternate> ])`

`(if<=? <expr> <consequent> [ <alternate> ])`

`(if>=? <expr> <consequent> [ <alternate> ])`

`(if-not=? <expr> <consequent> [ <alternate> ])`

The expression `<expr>` is evaluated; it will typically, but not necessarily, be a call on a comparison procedure. It is an error if its value is not -1, 0, or 1. If the value is consistent with the specified relation, `<consequent>` is evaluated and its value(s) are returned. Otherwise, if `<alternate>` is present, it is evaluated and its value(s) are returned; if it is absent, an unspecified value is returned.

Comparison predicates

`(=? comparator object1 object2 object3 ...)`

`(<? comparator object1 object2 object3 ...)`

`(>? comparator object1 object2 object3 ...)`

`(<=? comparator object1 object2 object3 ...)`

`(>=? comparator object1 object2 object3 ...)`

These procedures are analogous to the number, character, and string comparison predicates of Scheme. They allow the convenient use of comparators in situations where the expression types are not usable. They are also analogous to the similarly named procedures SRFI 67, but handle arbitrary numbers of arguments, which in SRFI 67 requires the use of the variants whose names begin with chain.

These procedures apply the comparison procedure of comparator to the objects as follows. If the specified relation returns #t for all objecti and objectj where n is the number of objects and 1 <= i < j <= n, then the procedures return #t, but otherwise #f.

The order in which the values are compared is unspecified. Because the relations are transitive, it suffices to compare each object with its successor.

Comparison predicate constructors

`(make=? comparator)`

`(make<? comparator)`

`(make>? comparator)`

`(make<=? comparator)`

`(make>=? comparator)`

These procedures return predicates which, when applied to two or more arguments, return #t if comparing obj1 and obj2 using the equality or comparison procedures of comparator shows that the objects bear the specified relation to one another. Such predicates can be used in contexts that do not understand or expect comparators.

Interval (ternary) comparison predicates

These procedures return true or false depending on whether an object is contained in an open, closed, or half-open interval. All comparisons are done in the sense of comparator, which is default-comparator if omitted.

`(in-open-interval? [comparator] obj1 obj2 obj3)`

Return #t if obj1 is less than obj2, which is less thanobj3, and #f otherwise.

`(in-closed-interval? [comparator] obj1 obj2 obj3)`

Returns #t if obj1 is less than or equal to obj2, which is less than or equal to obj3, and #f otherwise.

`(in-open-closed-interval? [comparator] obj1 obj2 obj3)`

Returns #t if obj1 is less than obj2, which is less than or equal to obj3, and #f otherwise.

`(in-closed-open-interval? [comparator] obj1 obj2 obj3)`

Returns #t if obj1 is less than or equal to obj2, which is less than obj3, and #f otherwise.

Min/max comparison procedures

`(comparator-min comparator object1 object2 ...)`

`(comparator-max comparator object1 object2 ...)`

These procedures are analogous to min and max respectively. They apply the comparison procedure of comparator to the objects to find and return a minimal (or maximal) object. The order in which the values are compared is unspecified. ## `(srfi srfi-132)`

This library is based on SRFI-132.

Abstract

TODO

Reference

TODO ## `(srfi srfi-133)`

This library is based on SRFI-133.

Abstract

TODO

Reference

TODO