4.9 Pairs and Lists

☞ Pairs and Lists in The Racket Guide introduces pairs and lists.

A *pair* combines exactly two values. The first value is accessed with the `car` procedure, and the second value is accessed with the `cdr` procedure. Pairs are not mutable (but see Mutable Pairs and Lists).

A *list* is recursively defined: it is either the constant `null`, or it is a pair whose second value is a list.

A list can be used as a single-valued sequence (see Sequences). The elements of the list serve as elements of the sequence. See also `in-list`.

Cyclic data structures can be created using only immutable pairs via `read` or `make-reader-graph`. If starting with a pair and using some number of `cdr`s returns to the starting pair, then the pair is not a list.

See Reading Pairs and Lists for information on `read`ing pairs and lists and Printing Pairs and Lists for information on `print`ing pairs and lists.

4.9.1 Pair Constructors and Selectors

(pair? *v*) → boolean? procedure
v : any/c

Returns `#t` if *v* is a pair, `#f` otherwise.

Examples:

```
> (pair? 1)
#f
> (pair? (cons 1 2))
#t
> (pair? (list 1 2))
#t
> (pair? '(1 2))
#t
> (pair? '())
#f
```

(null? *v*) → boolean? procedure
v : any/c

Returns `#t` if *v* is the empty list, `#f` otherwise.

Examples:

```
> (null? 1)
#f
> (null? '(1 2))
#f
> (null? '())
#t
> (null? (cdr (list 1)))
#t
```

(cons *a d***)** → pair? procedure

a : any/c

d : any/c

Returns a newly allocated pair whose first element is *a* and second element is *d*.

Examples:

```
> (cons 1 2)
'(1 . 2)
> (cons 1 '())
'(1)
```

(car *p***)** → any/c procedure

p : pair?

Returns the first element of the pair *p*.

Examples:

```
> (car '(1 2))
1
> (car (cons 2 3))
2
```

(cdr *p***)** → any/c procedure

p : pair?

Returns the second element of the pair *p*.

Examples:

```
> (cdr '(1 2))
'(2)
> (cdr '(1))
'()
```

---

null : null? value

The empty list.

Examples:

```
> null
'()
> '()
'()
> (eq? '() null)
#t
```

(**list?** *v*) → boolean? procedure
v : any/c

Returns #t if *v* is a list: either the empty list, or a pair whose second element is a list. This procedure effectively takes constant time due to internal caching (so that any necessary traversals of pairs can in principle count as an extra cost of allocating the pairs).

Examples:

```
> (list? '(1 2))
#t
> (list? (cons 1 (cons 2 '())))
#t
> (list? (cons 1 2))
#f
```

(**list** *v* ...) → list? procedure
v : any/c

Returns a newly allocated list containing the *v*s as its elements.

Examples:

```
> (list 1 2 3 4)
'(1 2 3 4)
> (list (list 1 2) (list 3 4))
'((1 2) (3 4))
```

(**list*** *v* ... *tail*) → any/c procedure
v : any/c
tail : any/c

Like list, but the last argument is used as the tail of the result, instead of the final element. The result is a list only if the last argument is a list.

Examples:

```
> (list* 1 2)
'(1 . 2)
> (list* 1 2 (list 3 4))
'(1 2 3 4)
```

(**build-list** *n proc*) → list? procedure

n : exact-nonnegative-integer?

proc : (exact-nonnegative-integer? . -> . any)

Creates a list of *n* elements by applying *proc* to the integers from 0 to (sub1 *n*) in order. If *lst* is the resulting list, then (list-ref *lst i*) is the value produced by (*proc i*).

Examples:

```
> (build-list 10 values)
'(0 1 2 3 4 5 6 7 8 9)
> (build-list 5 (lambda (x) (* x x)))
'(0 1 4 9 16)
```

4.9.2 List Operations

(**length** *lst*) → exact-nonnegative-integer? procedure

lst : list?

Returns the number of elements in *lst*.

Examples:

```
> (length (list 1 2 3 4))
4
> (length '())
0
```

(**list-ref** *lst pos*) → any/c procedure

lst : pair?

pos : exact-nonnegative-integer?

Returns the element of *lst* at position *pos*, where the list's first element is position 0. If the list has *pos* or fewer elements, then the exn:fail:contract exception is raised.

The *lst* argument need not actually be a list; *lst* must merely start with a chain of at least (add1 *pos*) pairs.

Examples:

```
> (list-ref (list 'a 'b 'c) 0)
'a
> (list-ref (list 'a 'b 'c) 1)
```

```
'b
> (list-ref (list 'a 'b 'c) 2)
'c
> (list-ref (cons 1 2) 0)
1
> (list-ref (cons 1 2) 1)
list-ref: index reaches a non-pair
  index: 1
  in: '(1 . 2)
```

(**list-tail** *lst pos*) → any/c — procedure

lst : any/c

pos : exact-nonnegative-integer?

Returns the list after the first *pos* elements of *lst*. If the list has fewer than *pos* elements, then the exn:fail:contract exception is raised.

The *lst* argument need not actually be a list; *lst* must merely start with a chain of at least *pos* pairs.

Examples:

```
> (list-tail (list 1 2 3 4 5) 2)
'(3 4 5)
> (list-tail (cons 1 2) 1)
2
> (list-tail (cons 1 2) 2)
list-tail: index reaches a non-pair
  index: 2
  in: '(1 . 2)
> (list-tail 'not-a-pair 0)
'not-a-pair
```

(**append** *lst* ...) → list? — procedure

lst : list?

(**append** *lst* ... *v*) → any/c

lst : list?

v : any/c

When given all list arguments, the result is a list that contains all of the elements of the given lists in order. The last argument is used directly in the tail of the result.

The last argument need not be a list, in which case the result is an “improper list.”

Examples:

```
> (append (list 1 2) (list 3 4))
'(1 2 3 4)
> (append (list 1 2) (list 3 4) (list 5 6) (list 7 8))
'(1 2 3 4 5 6 7 8)
```

(**reverse** *lst*) → list? procedure
  lst : list?

Returns a list that has the same elements as *lst*, but in reverse order.

Example:

```
> (reverse (list 1 2 3 4))
'(4 3 2 1)
```

4.9.3 List Iteration

(**map** *proc lst* ...+) → list? procedure
  proc : procedure?
  lst : list?

Applies *proc* to the elements of the *lst*s from the first elements to the last. The *proc* argument must accept the same number of arguments as the number of supplied *lst*s, and all *lst*s must have the same number of elements. The result is a list containing each result of *proc* in order.

Examples:

```
> (map (lambda (number)
         (+ 1 number))
       '(1 2 3 4))
'(2 3 4 5)
> (map (lambda (number1 number2)
         (+ number1 number2))
       '(1 2 3 4)
       '(10 100 1000 10000))
'(11 102 1003 10004)
```

(**andmap** *proc lst* ...+) → any procedure
  proc : procedure?
  lst : list?

Similar to `map` in the sense that *proc* is applied to each element of *lst*, but

> The `andmap` function is actually closer to `foldl` than `map`, since `andmap` doesn't produce a list. Still, `(andmap f (list x y z))` is equivalent to `(and (f x) (f y) (f z))` in the same way that `(map f (list x y z))` is equivalent to `(list (f x) (f y) (f z))`.

- the result is `#f` if any application of *proc* produces `#f`, in which case *proc* is not applied to later elements of the *lst*s; and

- the result is that of *proc* applied to the last elements of the *lst*s; more specifically, the application of *proc* to the last elements in the *lst*s is in tail position with respect to the andmap call.

If the *lst*s are empty, then #t is returned.

Examples:

```
> (andmap positive? '(1 2 3))
#t
> (andmap positive? '(1 2 a))
positive?: contract violation
  expected: real?
  given: 'a
> (andmap positive? '(1 -2 a))
#f
> (andmap + '(1 2 3) '(4 5 6))
9
```

(**ormap** *proc lst ...+*) → any — procedure

proc : procedure?

lst : list?

Similar to map in the sense that *proc* is applied to each element of *lst*, but

> To continue the andmap note above, (ormap f (list x y z)) is equivalent to (or (f x) (f y) (f z)).

- the result is #f if every application of *proc* produces #f; and
- the result is that of the first application of *proc* producing a value other than #f, in which case *proc* is not applied to later elements of the *lst*s; the application of *proc* to the last elements of the *lst*s is in tail position with respect to the ormap call.

If the *lst*s are empty, then #f is returned.

Examples:

```
> (ormap eq? '(a b c) '(a b c))
#t
> (ormap positive? '(1 2 a))
#t
> (ormap + '(1 2 3) '(4 5 6))
5
```

(**for-each** *proc lst ...+*) → void? — procedure

proc : procedure?

lst : list?

Similar to `map`, but *proc* is called only for its effect, and its result (which can be any number of values) is ignored.

Example:

```
> (for-each (lambda (arg)
              (printf "Got ~a\n" arg)
              23)
            '(1 2 3 4))
Got 1
Got 2
Got 3
Got 4
```

(**foldl** *proc init lst ...+*) → any/c — procedure

proc : procedure?
init : any/c
lst : list?

Like `map`, `foldl` applies a procedure to the elements of one or more lists. Whereas `map` combines the return values into a list, `foldl` combines the return values in an arbitrary way that is determined by *proc*.

If `foldl` is called with n lists, then *proc* must take $n+1$ arguments. The extra argument is the combined return values so far. The *proc* is initially invoked with the first item of each list, and the final argument is *init*. In subsequent invocations of *proc*, the last argument is the return value from the previous invocation of *proc*. The input *lst*s are traversed from left to right, and the result of the whole `foldl` application is the result of the last application of *proc*. If the *lst*s are empty, the result is *init*.

Unlike `foldr`, `foldl` processes the *lst*s in constant space (plus the space for each call to *proc*).

Examples:

```
> (foldl cons '() '(1 2 3 4))
'(4 3 2 1)
> (foldl + 0 '(1 2 3 4))
10
> (foldl (lambda (a b result)
           (* result (- a b)))
         1
         '(1 2 3)
         '(4 5 6))
-27
```

(**foldr** *proc init lst ...+*) → any/c — procedure

proc : procedure?
init : any/c
lst : list?

Like `foldl`, but the lists are traversed from right to left. Unlike `foldl`, `foldr` processes the *lst*s in space proportional to the length of *lst*s (plus the space for each call to *proc*).

Examples:

```
> (foldr cons '() '(1 2 3 4))
'(1 2 3 4)
> (foldr (lambda (v l) (cons (add1 v) l)) '() '(1 2 3 4))
'(2 3 4 5)
```

4.9.4 List Filtering

(**filter** *pred lst*) → list? procedure
pred : procedure?
lst : list?

Returns a list with the elements of *lst* for which *pred* produces a true value. The *pred* procedure is applied to each element from first to last.

Example:

```
> (filter positive? '(1 -2 3 4 -5))
'(1 3 4)
```

(**remove** *v lst* [*proc*]) → list? procedure
v : any/c
lst : list?
proc : procedure? = equal?

Returns a list that is like *lst*, omitting the first element of *lst* that is equal to *v* using the comparison procedure *proc* (which must accept two arguments).

Examples:

```
> (remove 2 (list 1 2 3 2 4))
'(1 3 2 4)
> (remove 2 (list 1 2 3 2 4) =)
'(1 3 2 4)
> (remove '(2) (list '(1) '(2) '(3)))
'((1) (3))
> (remove "2" (list "1" "2" "3"))
'("1" "3")
> (remove #\c (list #\a #\b #\c))
'(#\a #\b)
```

(**remq** *v lst*) → list? procedure
v : any/c
lst : list?

Returns `(remove v lst eq?)`.

Examples:

```
> (remq 2 (list 1 2 3 4 5))
'(1 3 4 5)
> (remq '(2) (list '(1) '(2) '(3)))
'((1) (2) (3))
> (remq "2" (list "1" "2" "3"))
'("1" "3")
> (remq #\c (list #\a #\b #\c))
'(#\a #\b)
```

(remv *v lst***)** → list? procedure

v : any/c

lst : list?

Returns `(remove v lst eqv?)`.

Examples:

```
> (remv 2 (list 1 2 3 4 5))
'(1 3 4 5)
> (remv '(2) (list '(1) '(2) '(3)))
'((1) (2) (3))
> (remv "2" (list "1" "2" "3"))
'("1" "3")
> (remv #\c (list #\a #\b #\c))
'(#\a #\b)
```

(remove* *v-lst lst* [*proc*]**)** → list? procedure

v-lst : list?

lst : list?

proc : procedure? = equal?

Like `remove`, but removes from *lst* every instance of every element of *v-lst*.

Example:

```
> (remove* (list 1 2) (list 1 2 3 2 4 5 2))
'(3 4 5)
```

(remq* *v-lst lst***)** → list? procedure

v-lst : list?

lst : list?

Returns `(remove* v-lst lst eq?)`.

Example:

```
> (remq* (list 1 2) (list 1 2 3 2 4 5 2))
'(3 4 5)
```

(**remv*** *v-lst lst*) → list? procedure

v-lst : list?

lst : list?

Returns (remove* *v-lst lst* eqv?).

Example:

```
> (remv* (list 1 2) (list 1 2 3 2 4 5 2))
'(3 4 5)
```

(**sort** *lst* procedure
less-than?
[#:key *extract-key*
#:cache-keys? *cache-keys?*]) → list?

lst : list?

less-than? : (any/c any/c . -> . any/c)

extract-key : (any/c . -> . any/c) = (lambda (x) x)

cache-keys? : boolean? = #f

Returns a list sorted according to the *less-than?* procedure, which takes two elements of *lst* and returns a true value if the first is less (i.e., should be sorted earlier) than the second.

The sort is stable; if two elements of *lst* are "equal" (i.e., *less-than?* does not return a true value when given the pair in either order), then the elements preserve their relative order from *lst* in the output list. To preserve this guarantee, use sort with a strict comparison functions (e.g., < or string<?; not <= or string<=?).

> Because of the peculiar fact that the IEEE-754 number system specifies that +nan.0 is neither greater nor less than nor equal to any other number, sorting lists containing this value may produce a surprising result.

The #:key argument *extract-key* is used to extract a key value for comparison from each list element. That is, the full comparison procedure is essentially

```
(lambda (x y)
  (less-than? (extract-key x) (extract-key y)))
```

By default, *extract-key* is applied to two list elements for every comparison, but if *cache-keys?* is true, then the *extract-key* function is used exactly once for each list item. Supply a true value for *cache-keys?* when *extract-key* is an expensive operation; for example, if file-or-directory-modify-seconds is used to extract a timestamp for every file in a list, then *cache-keys?* should be #t to minimize file-system calls, but if *extract-key* is car, then *cache-keys?* should be #f. As another example, providing *extract-key* as (lambda (x) (random)) and #t for *cache-keys?* effectively shuffles the list.

Examples:

```
> (sort '(1 3 4 2) <)
'(1 2 3 4)
> (sort '("aardvark" "dingo" "cow" "bear") string<?)
'("aardvark" "bear" "cow" "dingo")
> (sort '(("aardvark") ("dingo") ("cow") ("bear"))
        #:key car string<?)
'(("aardvark") ("bear") ("cow") ("dingo"))
```

4.9.5 List Searching

(**member** *v lst* [*is-equal?*]) → (or/c list? #f) procedure

v : any/c

lst : list?

is-equal? : (any/c any/c -> any/c) = equal?

Locates the first element of *lst* that is `equal?` to *v*. If such an element exists, the tail of *lst* starting with that element is returned. Otherwise, the result is `#f`.

Examples:

```
> (member 2 (list 1 2 3 4))
'(2 3 4)
> (member 9 (list 1 2 3 4))
#f
> (member #'x (list #'x #'y) free-identifier=?)
'(#<syntax:eval:509:0 x> #<syntax:eval:509:0 y>)
> (member #'a (list #'x #'y) free-identifier=?)
#f
```

(**memv** *v lst*) → (or/c list? #f) procedure

v : any/c

lst : list?

Like `member`, but finds an element using `eqv?`.

Examples:

```
> (memv 2 (list 1 2 3 4))
'(2 3 4)
> (memv 9 (list 1 2 3 4))
#f
```

(**memq** *v lst*) → (or/c list? #f) procedure

v : any/c

lst : list?

Like member, but finds an element using eq?.

Examples:

```
> (memq 2 (list 1 2 3 4))
'(2 3 4)
> (memq 9 (list 1 2 3 4))
#f
```

(**memf** *proc lst*) → (or/c list? #f) procedure

proc : procedure?

lst : list?

Like member, but finds an element using the predicate *proc*; an element is found when *proc* applied to the element returns a true value.

Example:

```
> (memf (lambda (arg)
          (> arg 9))
        '(7 8 9 10 11))
'(10 11)
```

(**findf** *proc lst*) → any/c procedure

proc : procedure?

lst : list?

Like memf, but returns the element or #f instead of a tail of *lst* or #f.

Example:

```
> (findf (lambda (arg)
           (> arg 9))
         '(7 8 9 10 11))
10
```

(**assoc** *v lst* [*is-equal?*]) → (or/c pair? #f) procedure

v : any/c

lst : (listof pair?)

is-equal? : (any/c any/c -> any/c) = equal?

Locates the first element of *lst* whose car is equal to *v* according to *is-equal?*. If such an element exists, the pair (i.e., an element of *lst*) is returned. Otherwise, the result is #f.

Examples:

```
> (assoc 3 (list (list 1 2) (list 3 4) (list 5 6)))
'(3 4)
> (assoc 9 (list (list 1 2) (list 3 4) (list 5 6)))
#f
```

```
> (assoc 3.5
         (list (list 1 2) (list 3 4) (list 5 6))
         (lambda (a b) (< (abs (- a b)) 1)))
'(3 4)
```

(**assv** *v lst*) → (or/c pair? #f) procedure

v : any/c

lst : (listof pair?)

Like assoc, but finds an element using eqv?.

Example:

```
> (assv 3 (list (list 1 2) (list 3 4) (list 5 6)))
'(3 4)
```

(**assq** *v lst*) → (or/c pair? #f) procedure

v : any/c

lst : (listof pair?)

Like assoc, but finds an element using eq?.

Example:

```
> (assq 'c (list (list 'a 'b) (list 'c 'd) (list 'e 'f)))
'(c d)
```

(**assf** *proc lst*) → (or/c list? #f) procedure

proc : procedure?

lst : list?

Like assoc, but finds an element using the predicate *proc*; an element is found when *proc* applied to the car of an *lst* element returns a true value.

Example:

```
> (assf (lambda (arg)
          (> arg 2))
        (list (list 1 2) (list 3 4) (list 5 6)))
'(3 4)
```

4.9.6 Pair Accessor Shorthands

(**caar** *v*) → any/c procedure

v : (cons/c pair? any/c)

Returns (car (car *v*)).

Example:

```
> (caar '((1 2) 3 4))
1
```

(**cadr** *v*) → any/c procedure
  v : (cons/c any/c pair?)

Returns (car (cdr *v*)).

Example:

```
> (cadr '((1 2) 3 4))
3
```

(**cdar** *v*) → any/c procedure
  v : (cons/c pair? any/c)

Returns (cdr (car *v*)).

Example:

```
> (cdar '((7 6 5 4 3 2 1) 8 9))
'(6 5 4 3 2 1)
```

(**cddr** *v*) → any/c procedure
  v : (cons/c any/c pair?)

Returns (cdr (cdr *v*)).

Example:

```
> (cddr '(2 1))
'()
```

(**caaar** *v*) → any/c procedure
  v : (cons/c (cons/c pair? any/c) any/c)

Returns (car (car (car *v*))).

Example:

```
> (caaar '(((6 5 4 3 2 1) 7) 8 9))
6
```

(**caadr** *v*) → any/c procedure
  v : (cons/c any/c (cons/c pair? any/c))

Returns (car (car (cdr *v*))).

Example:

```
> (caadr '(9 (7 6 5 4 3 2 1) 8))
7
```

(**cadar** *v*) → any/c procedure
v : (cons/c (cons/c any/c pair?) any/c)

Returns (car (cdr (car *v*))).

Example:

```
> (cadar '((7 6 5 4 3 2 1) 8 9))
6
```

(**caddr** *v*) → any/c procedure
v : (cons/c any/c (cons/c any/c pair?))

Returns (car (cdr (cdr *v*))).

Example:

```
> (caddr '(3 2 1))
1
```

(**cdaar** *v*) → any/c procedure
v : (cons/c (cons/c pair? any/c) any/c)

Returns (cdr (car (car *v*))).

Example:

```
> (cdaar '(((6 5 4 3 2 1) 7) 8 9))
'(5 4 3 2 1)
```

(**cdadr** *v*) → any/c procedure
v : (cons/c any/c (cons/c pair? any/c))

Returns (cdr (car (cdr *v*))).

Example:

```
> (cdadr '(9 (7 6 5 4 3 2 1) 8))
'(6 5 4 3 2 1)
```

(**cddar** *v*) → any/c procedure
v : (cons/c (cons/c any/c pair?) any/c)

Returns (cdr (cdr (car *v*))).

Example:

```
> (cddar '((7 6 5 4 3 2 1) 8 9))
'(5 4 3 2 1)
```

(**cdddr** *v*) → any/c procedure
v : (cons/c any/c (cons/c any/c pair?))

Returns (cdr (cdr (cdr *v*))).

Example:

```
> (cdddr '(3 2 1))
'()
```

(**caaaar** *v*) → any/c procedure
v : (cons/c (cons/c (cons/c pair? any/c) any/c) any/c)

Returns (car (car (car (car *v*)))).

Example:

```
> (caaaar '((((5 4 3 2 1) 6) 7) 8 9))
5
```

(**caaadr** *v*) → any/c procedure
v : (cons/c any/c (cons/c (cons/c pair? any/c) any/c))

Returns (car (car (car (cdr *v*)))).

Example:

```
> (caaadr '(9 ((6 5 4 3 2 1) 7) 8))
6
```

(**caadar** *v*) → any/c procedure
v : (cons/c (cons/c any/c (cons/c pair? any/c)) any/c)

Returns (car (car (cdr (car *v*)))).

Example:

```
> (caadar '((7 (5 4 3 2 1) 6) 8 9))
5
```

(**caaddr** *v*) → any/c procedure
v : (cons/c any/c (cons/c any/c (cons/c pair? any/c)))

Returns (car (car (cdr (cdr *v*)))).

Example:

```
> (caaddr '(9 8 (6 5 4 3 2 1) 7))
6
```

(**cadaar** *v*) → any/c procedure
v : (cons/c (cons/c (cons/c any/c pair?) any/c) any/c)

Returns (car (cdr (car (car *v*)))).

Example:

```
> (cadaar '(((6 5 4 3 2 1) 7) 8 9))
5
```

(**cadadr** *v*) → any/c procedure
v : (cons/c any/c (cons/c (cons/c any/c pair?) any/c))

Returns (car (cdr (car (cdr *v*)))).

Example:

```
> (cadadr '(9 (7 6 5 4 3 2 1) 8))
6
```

(**caddar** *v*) → any/c procedure
v : (cons/c (cons/c any/c (cons/c any/c pair?)) any/c)

Returns (car (cdr (cdr (car *v*)))).

Example:

```
> (caddar '((7 6 5 4 3 2 1) 8 9))
5
```

(**cadddr** *v*) → any/c procedure
v : (cons/c any/c (cons/c any/c (cons/c any/c pair?)))

Returns (car (cdr (cdr (cdr *v*)))).

Example:

```
> (cadddr '(4 3 2 1))
1
```

(**cdaaar** *v*) → any/c procedure
v : (cons/c (cons/c (cons/c pair? any/c) any/c) any/c)

Returns (cdr (car (car (car *v*)))).

Example:

```
> (cdaaar '((((5 4 3 2 1) 6) 7) 8 9))
'(4 3 2 1)
```

(**cdaadr** *v*) → any/c procedure
v : (cons/c any/c (cons/c (cons/c pair? any/c) any/c))

Returns (cdr (car (car (cdr *v*)))).

Example:

```
> (cdaadr '(9 ((6 5 4 3 2 1) 7) 8))
'(5 4 3 2 1)
```

(**cdadar** *v*) → any/c procedure
v : (cons/c (cons/c any/c (cons/c pair? any/c)) any/c)

Returns (cdr (car (cdr (car *v*)))).

Example:

```
> (cdadar '((7 (5 4 3 2 1) 6) 8 9))
'(4 3 2 1)
```

(**cdaddr** *v*) → any/c procedure
v : (cons/c any/c (cons/c any/c (cons/c pair? any/c)))

Returns (cdr (car (cdr (cdr *v*)))).

Example:

```
> (cdaddr '(9 8 (6 5 4 3 2 1) 7))
'(5 4 3 2 1)
```

(**cddaar** *v*) → any/c procedure
v : (cons/c (cons/c (cons/c any/c pair?) any/c) any/c)

Returns (cdr (cdr (car (car *v*)))).

Example:

```
> (cddaar '(((6 5 4 3 2 1) 7) 8 9))
'(4 3 2 1)
```

(**cddadr** *v*) → any/c procedure
v : (cons/c any/c (cons/c (cons/c any/c pair?) any/c))

Returns (cdr (cdr (car (cdr *v*)))).

Example:

```
> (cddadr '(9 (7 6 5 4 3 2 1) 8))
'(5 4 3 2 1)
```

(cdddar *v***)** → any/c procedure
v : (cons/c (cons/c any/c (cons/c any/c pair?)) any/c)

Returns (cdr (cdr (cdr (car *v*)))).

Example:

```
> (cdddar '((7 6 5 4 3 2 1) 8 9))
'(4 3 2 1)
```

(cddddr *v***)** → any/c procedure
v : (cons/c any/c (cons/c any/c (cons/c any/c pair?)))

Returns (cdr (cdr (cdr (cdr *v*)))).

Example:

```
> (cddddr '(4 3 2 1))
'()
```

4.9.7 Additional List Functions and Synonyms

(require racket/list) package: base

The bindings documented in this section are provided by the racket/list and racket libraries, but not racket/base.

---

empty : null? value

The empty list.

Examples:

```
> empty
'()
> (eq? empty null)
#t
```

(cons? *v***)** → boolean? procedure
v : any/c

The same as (pair? *v*).

Example:

```
> (cons? '(1 2))
#t
```

(**empty?** *v*) → boolean? procedure
v : any/c

The same as (null? *v*).

Examples:

```
> (empty? '(1 2))
#f
> (empty? '())
#t
```

(**first** *lst*) → any/c procedure
lst : list?

The same as (car *lst*), but only for lists (that are not empty).

Example:

```
> (first '(1 2 3 4 5 6 7 8 9 10))
1
```

(**rest** *lst*) → list? procedure
lst : list?

The same as (cdr *lst*), but only for lists (that are not empty).

Example:

```
> (rest '(1 2 3 4 5 6 7 8 9 10))
'(2 3 4 5 6 7 8 9 10)
```

(**second** *lst*) → any procedure
lst : list?

Returns the second element of the list.

Example:

```
> (second '(1 2 3 4 5 6 7 8 9 10))
2
```

(**third** *lst*) → any procedure
lst : list?

Returns the third element of the list.

Example:

```
> (third '(1 2 3 4 5 6 7 8 9 10))
3
```

(**fourth** *lst*) → any procedure
lst : list?

Returns the fourth element of the list.

Example:

```
> (fourth '(1 2 3 4 5 6 7 8 9 10))
4
```

(**fifth** *lst*) → any procedure
lst : list?

Returns the fifth element of the list.

Example:

```
> (fifth '(1 2 3 4 5 6 7 8 9 10))
5
```

(**sixth** *lst*) → any procedure
lst : list?

Returns the sixth element of the list.

Example:

```
> (sixth '(1 2 3 4 5 6 7 8 9 10))
6
```

(**seventh** *lst*) → any procedure
lst : list?

Returns the seventh element of the list.

Example:

```
> (seventh '(1 2 3 4 5 6 7 8 9 10))
7
```

(**eighth** *lst*) → any procedure
lst : list?

Returns the eighth element of the list.

Example:

```
> (eighth '(1 2 3 4 5 6 7 8 9 10))
8
```

(**ninth** *lst*) → any procedure
  lst : list?

Returns the ninth element of the list.

Example:

```
> (ninth '(1 2 3 4 5 6 7 8 9 10))
9
```

(**tenth** *lst*) → any procedure
  lst : list?

Returns the tenth element of the list.

Example:

```
> (tenth '(1 2 3 4 5 6 7 8 9 10))
10
```

(**last** *lst*) → any procedure
  lst : list?

Returns the last element of the list.

Example:

```
> (last '(1 2 3 4 5 6 7 8 9 10))
10
```

(**last-pair** *p*) → pair? procedure
  p : pair?

Returns the last pair of a (possibly improper) list.

Example:

```
> (last-pair '(1 2 3 4))
'(4)
```

(**make-list** *k v*) → list? procedure
  k : exact-nonnegative-integer?

v : any/c

Returns a newly constructed list of length *k*, holding *v* in all positions.

Example:

```
> (make-list 7 'foo)
'(foo foo foo foo foo foo foo)
```

(**list-update** *lst pos updater*) → list? procedure

lst : list?

pos : (and/c (>=/c 0) (</c (length *lst*)))

updater : (-> any/c any/c)

Returns a list that is the same as *lst* except at the specified index. The element at the specified index is (*updater* (list-ref *lst pos*)).

Example:

```
> (list-update '(zero one two) 1 symbol->string)
'(zero "one" two)
```

Added in version 6.3 of package base.

(**list-set** *lst pos value*) → list? procedure

lst : list?

pos : (and/c (>=/c 0) (</c (length *lst*)))

value : any/c

Returns a list that is the same as *lst* except at the specified index. The element at the specified index is *value*.

Example:

```
> (list-set '(zero one two) 2 "two")
'(zero one "two")
```

Added in version 6.3 of package base.

(**index-of** *lst v* [*is-equal?*]) → (or/c exact-nonnegative-integer? #f) procedure

lst : list?

v : any/c

is-equal? : (any/c any/c . -> . any/c) = equal?

Like member, but returns the index of the first element found instead of the tail of the list.

Example:

```
> (index-of '(1 2 3 4) 3)
2
```

Added in version 6.7.0.3 of package base.

(**index-where** *lst proc*) → (or/c exact-nonnegative-integer? #f) procedure
lst : list?
proc : (any/c . -> . any/c)

Like index-of but with the predicate-searching behavior of memf.

Example:

```
> (index-where '(1 2 3 4) even?)
1
```

Added in version 6.7.0.3 of package base.

(**indexes-of** *lst v* [*is-equal?*]) procedure
→ (listof exact-nonnegative-integer?)
lst : list?
v : any/c
is-equal? : (any/c any/c . -> . any/c) = equal?

Like index-of, but returns the a list of all the indexes where the element occurs in the list instead of just the first one.

Example:

```
> (indexes-of '(1 2 1 2 1) 2)
'(1 3)
```

Added in version 6.7.0.3 of package base.

(**indexes-where** *lst proc*) → (listof exact-nonnegative-integer?) procedure
lst : list?
proc : (any/c . -> . any/c)

Like indexes-of but with the predicate-searching behavior of index-where.

Example:

```
> (indexes-where '(1 2 3 4) even?)
'(1 3)
```

Added in version 6.7.0.3 of package base.

(**take** *lst pos*) → list? procedure
lst : any/c
pos : exact-nonnegative-integer?

Returns a fresh list whose elements are the first *pos* elements of *lst*. If *lst* has fewer than *pos* elements, the exn:fail:contract exception is raised.

The *lst* argument need not actually be a list; *lst* must merely start with a chain of at least *pos* pairs.

Examples:

```
> (take '(1 2 3 4 5) 2)
'(1 2)
> (take 'non-list 0)
'()
```

(**drop** *lst pos*) → any/c — procedure

lst : any/c

pos : exact-nonnegative-integer?

Just like `list-tail`.

(**split-at** *lst pos*) → list? any/c — procedure

lst : any/c

pos : exact-nonnegative-integer?

Returns the same result as

```
(values (take lst pos) (drop lst pos))
```

except that it can be faster.

(**takef** *lst pred*) → list? — procedure

lst : any/c

pred : procedure?

Returns a fresh list whose elements are taken successively from *lst* as long as they satisfy *pred*. The returned list includes up to, but not including, the first element in *lst* for which *pred* returns `#f`.

The *lst* argument need not actually be a list; the chain of pairs in *lst* will be traversed until a non-pair is encountered.

Examples:

```
> (takef '(2 4 5 8) even?)
'(2 4)
> (takef '(2 4 6 8) odd?)
'()
> (takef '(2 4 . 6) even?)
'(2 4)
```

(**dropf** *lst pred*) → any/c — procedure

lst : any/c

pred : procedure?

Drops elements from the front of *lst* as long as they satisfy *pred*.

Examples:

```
> (dropf '(2 4 5 8) even?)
'(5 8)
> (dropf '(2 4 6 8) odd?)
'(2 4 6 8)
```

(splitf-at *lst pred***)** → list? any/c — procedure

lst : any/c

pred : procedure?

Returns the same result as

```
(values (takef lst pred) (dropf lst pred))
```

except that it can be faster.

(take-right *lst pos***)** → any/c — procedure

lst : any/c

pos : exact-nonnegative-integer?

Returns the list's *pos*-length tail. If *lst* has fewer than *pos* elements, then the exn:fail:contract exception is raised.

The *lst* argument need not actually be a list; *lst* must merely end with a chain of at least *pos* pairs.

Examples:

```
> (take-right '(1 2 3 4 5) 2)
'(4 5)
> (take-right 'non-list 0)
'non-list
```

(drop-right *lst pos***)** → list? — procedure

lst : any/c

pos : exact-nonnegative-integer?

Returns a fresh list whose elements are the prefix of *lst*, dropping its *pos*-length tail. If *lst* has fewer than *pos* elements, then the exn:fail:contract exception is raised.

The *lst* argument need not actually be a list; *lst* must merely end with a chain of at least *pos* pairs.

Examples:

```
> (drop-right '(1 2 3 4 5) 2)
'(1 2 3)
> (drop-right 'non-list 0)
```

```
'()
```

(**split-at-right** *lst pos*) → list? any/c — procedure

lst : any/c

pos : exact-nonnegative-integer?

Returns the same result as

```
(values (drop-right lst pos) (take-right lst pos))
```

except that it can be faster.

Examples:

```
> (split-at-right '(1 2 3 4 5 6) 3)
'(1 2 3)
'(4 5 6)
> (split-at-right '(1 2 3 4 5 6) 4)
'(1 2)
'(3 4 5 6)
```

(**takef-right** *lst pred*) → any/c — procedure

lst : any/c

pred : procedure?

(**dropf-right** *lst pred*) → list? — procedure

lst : any/c

pred : procedure?

(**splitf-at-right** *lst pred*) → list? any/c — procedure

lst : any/c

pred : procedure?

Like takef, dropf, and splitf-at, but combined with the from-right functionality of take-right, drop-right, and split-at-right.

(**list-prefix?** *l r* [*same?*]) → boolean? — procedure

l : list?

r : list?

same? : (any/c any/c . -> . any/c) = equal?

True if *l* is a prefix of *r*.

Example:

```
> (list-prefix? '(1 2) '(1 2 3 4 5))
#t
```

Added in version 6.3 of package base.

(**take-common-prefix** *l r* [*same?*]) → list? — procedure

l : list?
r : list?
same? : (any/c any/c . -> . any/c) = equal?

Returns the longest common prefix of *l* and *r*.

Example:

```
> (take-common-prefix '(a b c d) '(a b x y z))
'(a b)
```

Added in version 6.3 of package base.

(drop-common-prefix *l r* [*same?*]**)** → list? list? procedure
l : list?
r : list?
same? : (any/c any/c . -> . any/c) = equal?

Returns the tails of *l* and *r* with the common prefix removed.

Example:

```
> (drop-common-prefix '(a b c d) '(a b x y z))
'(c d)
'(x y z)
```

Added in version 6.3 of package base.

(split-common-prefix *l r* [*same?*]**)** → list? list? list? procedure
l : list?
r : list?
same? : (any/c any/c . -> . any/c) = equal?

Returns the longest common prefix together with the tails of *l* and *r* with the common prefix removed.

Example:

```
> (split-common-prefix '(a b c d) '(a b x y z))
'(a b)
'(c d)
'(x y z)
```

Added in version 6.3 of package base.

(add-between *lst* procedure
v
[#:before-first *before-first*
#:before-last *before-last*
#:after-last *after-last*
#:splice? *splice?*]) → list?

```
lst : list?
v : any/c
before-first : list? = '()
before-last : any/c = v
after-last : list? = '()
splice? : any/c = #f
```

Returns a list with the same elements as *lst*, but with *v* between each pair of elements in *lst*; the last pair of elements will have *before-last* between them, instead of *v* (but *before-last* defaults to *v*).

If *splice?* is true, then *v* and *before-last* should be lists, and the list elements are spliced into the result. In addition, when *splice?* is true, *before-first* and *after-last* are inserted before the first element and after the last element respectively.

Examples:

```
> (add-between '(x y z) 'and)
'(x and y and z)
> (add-between '(x) 'and)
'(x)
> (add-between '("a" "b" "c" "d") "," #:before-last "and")
'("a" "," "b" "," "c" "and" "d")
> (add-between '(x y z) '(-) #:before-last '(- -)
               #:before-first '(begin) #:after-last '(end LF)
               #:splice? #t)
'(begin x - y - - z end LF)
```

(append* *lst* ... *lsts*) → list? procedure

```
lst : list?
lsts : (listof list?)
```

(append* *lst* ... *lsts*) → any/c

```
lst : list?
lsts : list?
```

Like append, but the last argument is used as a list of arguments for append, so (append* *lst* ... *lsts*) is the same as (apply append *lst* ... *lsts*). In other words, the relationship between append and append* is similar to the one between list and list*.

Examples:

```
> (append* '(a) '(b) '((c) (d)))
'(a b c d)
> (cdr (append* (map (lambda (x) (list ", " x))
                     '("Alpha" "Beta" "Gamma"))))
'("Alpha" ", " "Beta" ", " "Gamma")
```

(flatten *v*) → list? procedure

```
v : any/c
```

Flattens an arbitrary S-expression structure of pairs into a list. More precisely, *v* is treated as a binary tree where pairs are interior nodes, and the resulting list contains all of the non-`null` leaves of the tree in the same order as an inorder traversal.

Examples:

```
> (flatten '((a) b (c (d) . e) ()))
'(a b c d e)
> (flatten 'a)
'(a)
```

(**check-duplicates** *lst* procedure
[*same?*
#:key *extract-key*
#:default *failure-result*]) → any
lst : list?
same? : (any/c any/c . -> . any/c) = equal?
extract-key : (-> any/c any/c) = (lambda (x) x)
failure-result : (failure-result/c any/c) = (lambda () #f)

Returns the first duplicate item in *lst*. More precisely, it returns the first *x* such that there was a previous *y* where (*same?* (*extract-key x*) (*extract-key y*)).

If no duplicate is found, then *failure-result* determines the result:

- If *failure-result* is a procedure, it is called (through a tail call) with no arguments to produce the result.
- Otherwise, *failure-result* is returned as the result.

The *same?* argument should be an equivalence predicate such as `equal?` or `eqv?` or a dictionary. The procedures `equal?`, `eqv?`, and `eq?` automatically use a dictionary for speed.

Examples:

```
> (check-duplicates '(1 2 3 4))
#f
> (check-duplicates '(1 2 3 2 1))
2
> (check-duplicates '((a 1) (b 2) (a 3)) #:key car)
'(a 3)
> (check-duplicates '(1 2 3 4 5 6)
                    (lambda (x y) (equal? (modulo x 3) (modulo y 3))))
4
> (check-duplicates '(1 2 3 4) #:default "no duplicates")
"no duplicates"
```

Added in version 6.3 of package `base`.
Changed in version 6.11.0.2: Added the `#:default` optional argument.

(**remove-duplicates** *lst* procedure

```
                    [same?
                     #:key extract-key]) → list?
  lst : list?
  same? : (any/c any/c . -> . any/c) = equal?
  extract-key : (any/c . -> . any/c) = (lambda (x) x)
```

Returns a list that has all items in *lst*, but without duplicate items, where *same?* determines whether two elements of the list are equivalent. The resulting list is in the same order as *lst*, and for any item that occurs multiple times, the first one is kept.

The `#:key` argument *extract-key* is used to extract a key value from each list element, so two items are considered equal if `(same? (extract-key x) (extract-key y))` is true.

Examples:

```
> (remove-duplicates '(a b b a))
'(a b)
> (remove-duplicates '(1 2 1.0 0))
'(1 2 1.0 0)
> (remove-duplicates '(1 2 1.0 0) =)
'(1 2 0)
```

`(filter-map proc lst ...+) → list?` procedure

```
  proc : procedure?
  lst : list?
```

Like `(map proc lst ...)`, except that, if *proc* returns `#false`, that element is omitted from the resulting list. In other words, `filter-map` is equivalent to `(filter (lambda (x) x) (map proc lst ...))`, but more efficient, because `filter-map` avoids building the intermediate list.

Example:

```
> (filter-map (lambda (x) (and (positive? x) x)) '(1 2 3 -2 8))
'(1 2 3 8)
```

`(count proc lst ...+) → exact-nonnegative-integer?` procedure

```
  proc : procedure?
  lst : list?
```

Returns `(length (filter-map proc lst ...))`, but without building the intermediate list.

Example:

```
> (count positive? '(1 -1 2 3 -2 5))
4
```

`(partition pred lst) → list? list?` procedure

```
  pred : procedure?
  lst : list?
```

Similar to `filter`, except that two values are returned: the items for which *pred* returns a true value, and the items for which *pred* returns `#f`.

The result is the same as

```
(values (filter pred lst) (filter (negate pred) lst))
```

but *pred* is applied to each item in *lst* only once.

Example:

```
> (partition even? '(1 2 3 4 5 6))
'(2 4 6)
'(1 3 5)
```

(range *end***)** → list? procedure
  end : real?
(range *start end* [*step*]**)** → list?
  start : real?
  end : real?
  step : real? = 1

Similar to `in-range`, but returns lists.

The resulting list holds numbers starting at *start* and whose successive elements are computed by adding *step* to their predecessor until *end* (excluded) is reached. If no starting point is provided, `0` is used. If no *step* argument is provided, `1` is used.

Like `in-range`, a `range` application can provide better performance when it appears directly in a `for` clause.

Examples:

```
> (range 10)
'(0 1 2 3 4 5 6 7 8 9)
> (range 10 20)
'(10 11 12 13 14 15 16 17 18 19)
> (range 20 40 2)
'(20 22 24 26 28 30 32 34 36 38)
> (range 20 10 -1)
'(20 19 18 17 16 15 14 13 12 11)
> (range 10 15 1.5)
'(10 11.5 13.0 14.5)
```

Changed in version 6.7.0.4 of package `base`: Adjusted to cooperate with `for` in the same way that `in-range` does.

(append-map *proc lst* ...+**)** → list? procedure
  proc : procedure?
  lst : list?

Returns `(append* (map proc lst ...))`.

Example:

```
> (append-map vector->list '(#(1) #(2 3) #(4)))
'(1 2 3 4)
```

`(filter-not pred lst)` → `list?` procedure
`pred` : `(any/c . -> . any/c)`
`lst` : `list?`

Like `filter`, but the meaning of the `pred` predicate is reversed: the result is a list of all items for which `pred` returns `#f`.

Example:

```
> (filter-not even? '(1 2 3 4 5 6))
'(1 3 5)
```

`(shuffle lst)` → `list?` procedure
`lst` : `list?`

Returns a list with all elements from `lst`, randomly shuffled.

Examples:

```
> (shuffle '(1 2 3 4 5 6))
'(3 2 6 4 5 1)
> (shuffle '(1 2 3 4 5 6))
'(2 1 5 4 3 6)
> (shuffle '(1 2 3 4 5 6))
'(2 6 5 1 3 4)
```

`(combinations lst)` → `list?` procedure
`lst` : `list?`
`(combinations lst size)` → `list?`
`lst` : `list?`
`size` : `exact-nonnegative-integer?`

> Wikipedia combinations

Return a list of all combinations of elements in the input list (aka the powerset of `lst`). If `size` is given, limit results to combinations of `size` elements.

Examples:

```
> (combinations '(1 2 3))
'(() (1) (2) (1 2) (3) (1 3) (2 3) (1 2 3))
> (combinations '(1 2 3) 2)
```

```
'((1 2) (1 3) (2 3))
```

(**in-combinations** *lst*) → sequence? procedure
  lst : list?
(**in-combinations** *lst size*) → sequence?
  lst : list?
  size : exact-nonnegative-integer?

Returns a sequence of all combinations of elements in the input list, or all combinations of length *size* if *size* is given. Builds combinations one-by-one instead of all at once.

Examples:

```
> (time (begin (combinations (range 15)) (void)))
cpu time: 44 real time: 15 gc time: 0
> (time (begin (in-combinations (range 15)) (void)))
cpu time: 0 real time: 0 gc time: 0
```

(**permutations** *lst*) → list? procedure
  lst : list?

Returns a list of all permutations of the input list. Note that this function works without inspecting the elements, and therefore it ignores repeated elements (which will result in repeated permutations). Raises an error if the input list contains more than 256 elements.

Examples:

```
> (permutations '(1 2 3))
'((1 2 3) (2 1 3) (1 3 2) (3 1 2) (2 3 1) (3 2 1))
> (permutations '(x x))
'((x x) (x x))
```

(**in-permutations** *lst*) → sequence? procedure
  lst : list?

Returns a sequence of all permutations of the input list. It is equivalent to `(in-list (permutations l))` but much faster since it builds the permutations one-by-one on each iteration Raises an error if the input list contains more than 256 elements.

(**argmin** *proc lst*) → any/c procedure
  proc : (-> any/c real?)
  lst : (and/c pair? list?)

Returns the first element in the list *lst* that minimizes the result of *proc*. Signals an error on an empty list. See also `min`.

Examples:

```
> (argmin car '((3 pears) (1 banana) (2 apples)))
```

```
'(1 banana)
> (argmin car '((1 banana) (1 orange)))
'(1 banana)
```

(**argmax** *proc lst*) → any/c — procedure

proc : (-> any/c real?)
lst : (and/c pair? list?)

Returns the first element in the list *lst* that maximizes the result of *proc*. Signals an error on an empty list. See also max.

Examples:

```
> (argmax car '((3 pears) (1 banana) (2 apples)))
'(3 pears)
> (argmax car '((3 pears) (3 oranges)))
'(3 pears)
```

(**group-by** *key lst* [*same?*]) → (listof list?) — procedure

key : (-> any/c any/c)
lst : list?
same? : (any/c any/c . -> . any/c) = equal?

Groups the given list into equivalence classes, with equivalence being determined by *same?*. Within each equivalence class, group-by preserves the ordering of the original list. Equivalence classes themselves are in order of first appearance in the input.

Example:

```
> (group-by (lambda (x) (modulo x 3)) '(1 2 1 2 54 2 5 43 7 2 643 1 2 0))
'((1 1 43 7 643 1) (2 2 2 5 2 2) (54 0))
```

Added in version 6.3 of package base.

(**cartesian-product** *lst* ...) → (listof list?) — procedure

lst : list?

Computes the n-ary cartesian product of the given lists.

Examples:

```
> (cartesian-product '(1 2 3) '(a b c))
'((1 a) (1 b) (1 c) (2 a) (2 b) (2 c) (3 a) (3 b) (3 c))
> (cartesian-product '(4 5 6) '(d e f) '(#t #f))
'((4 d #t)
  (4 d #f)
  (4 e #t)
  (4 e #f)
  (4 f #t)
  (4 f #f)
```

```
  (5 d #t)
  (5 d #f)
  (5 e #t)
  (5 e #f)
  (5 f #t)
  (5 f #f)
  (6 d #t)
  (6 d #f)
  (6 e #t)
  (6 e #f)
  (6 f #t)
  (6 f #f))
```

Added in version 6.3 of package base.

(**remf** *pred lst*) → list? procedure

pred : procedure?

lst : list?

Returns a list that is like *lst*, omitting the first element of *lst* for which *pred* produces a true value.

Example:

```
> (remf negative? '(1 -2 3 4 -5))
'(1 3 4 -5)
```

Added in version 6.3 of package base.

(**remf*** *pred lst*) → list? procedure

pred : procedure?

lst : list?

Like remf, but removes all the elements for which *pred* produces a true value.

Example:

```
> (remf* negative? '(1 -2 3 4 -5))
'(1 3 4)
```

Added in version 6.3 of package base.

4.9.8 Immutable Cyclic Data

(**make-reader-graph** *v*) → any/c procedure

v : any/c

Returns a value like *v*, with placeholders created by make-placeholder replaced with the values that they contain, and with placeholders created by make-hash-placeholder with an

immutable hash table. No part of *v* is mutated; instead, parts of *v* are copied as necessary to construct the resulting graph, where at most one copy is created for any given value.

Since the copied values can be immutable, and since the copy is also immutable, `make-reader-graph` can create cycles involving only immutable pairs, vectors, boxes, and hash tables.

Only the following kinds of values are copied and traversed to detect placeholders:

- pairs
- vectors, both mutable and immutable
- boxes, both mutable and immutable
- hash tables, both mutable and immutable
- instances of a prefab structure type
- placeholders created by `make-placeholder` and `make-hash-placeholder`

Due to these restrictions, `make-reader-graph` creates exactly the same sort of cyclic values as `read`.

Example:

```
> (let* ([ph (make-placeholder #f)]
         [x (cons 1 ph)])
    (placeholder-set! ph x)
    (make-reader-graph x))
#0='(1 . #0#)
```

(placeholder? *v*) → boolean? procedure

v : any/c

Returns `#t` if *v* is a placeholder created by `make-placeholder`, `#f` otherwise.

(make-placeholder *v*) → placeholder? procedure

v : any/c

Returns a placeholder for use with `placeholder-set!` and `make-reader-graph`. The *v* argument supplies the initial value for the placeholder.

(placeholder-set! *ph datum*) → void? procedure

ph : placeholder?

datum : any/c

Changes the value of *ph* to v.

(placeholder-get *ph*) → any/c procedure

ph : placeholder?

Returns the value of *ph*.

(**hash-placeholder?** *v*) → boolean? procedure
v : any/c

Returns #t if *v* is a placeholder created by make-hash-placeholder, #f otherwise.

(**make-hash-placeholder** *assocs*) → hash-placeholder? procedure
assocs : (listof pair?)

Like make-immutable-hash, but produces a table placeholder for use with make-reader-graph.

(**make-hasheq-placeholder** *assocs*) → hash-placeholder? procedure
assocs : (listof pair?)

Like make-immutable-hasheq, but produces a table placeholder for use with make-reader-graph.

(**make-hasheqv-placeholder** *assocs*) → hash-placeholder? procedure
assocs : (listof pair?)

Like make-immutable-hasheqv, but produces a table placeholder for use with make-reader-graph.