

caddr
cdaar
cdadr
cddar
caaar
caaadr
caadar
cadadr
caddr
cadddr
cdaar
cdaadr
cdadar
cdaddr
cdddar
cdddar
cddddr

4.10.7 Additional List Functions and Synonyms
empty
cons?
empty?
first
rest
second
third
fourth
fifth
sixth
seventh
eighth
ninth
tenth
last
last-pair
make-list
list-update
list-set
index-of
index-where
indexes-of
indexes-where
take
drop
split-at
takef
dropf
splitf-at
take-right
dropf-right
splitf-at-right
list-prefix?
take-common-prefix
drop-common-prefix
split-common-prefix
add-between
append+
flatten
check-duplicates
remove-duplicates
filter-map
count
partition
range
inclusive-range
append-map
filter-not
shuffle
combinations
in-combinations
permutations
in-permutations
argmin
argmax

```
> '()
'()
> (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 vs 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)
```

4.10.2 List Operations

(length lst) → exact-nonnegative-integer? procedure
lst : list

Returns the number of elements in lst. This function takes time proportional to that length.

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.

This function takes time proportional to pos.

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)
```

```

group-by
cartesian-
product
remf
remf+
4.10.8 Immutable Cyclic
Data
make-reader-
graph
placeholder?
make-placeholder
placeholder-set!
placeholder-get
hash-
placeholder?
make-hash-
placeholder
make-hasheq-
placeholder
make-hasheqv-
placeholder
make-hashalw-
placeholder

```

```
(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.

This function takes time proportional to *pos*.

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.”

This function takes time proportional to the length of all arguments (added together) except the last argument.

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.

This function takes time proportional to the length of *lst*.

Example:

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

4.10.3 List Iteration

```
(map proc lst ...) → list?                                procedure
  proc : procedure?
  lst : list?
```

Applies *proc* to the elements of the *lists* from the first elements to the last. The *proc* argument must accept the same number of arguments as the number of supplied *lists*, and all *lists* 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 result is #*f* if any application of *proc* produces #*f*, in which case *proc* is not applied to later elements of the *lists*; and
- the result is that of *proc* applied to the last elements of the *lists*; more specifically, the application of *proc* to the last elements in the *lists* is in tail position with respect to the `andmap` call.

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

Examples:

```
> (andmap positive? '(1 2 3))
#t
> (andmap positive? '(1 2 a))
positive?: contract violation
  expected: real?
  provided: a
```

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))`.

```

~~~~~ ~~~
given: 'a
> (andmap positive? '(1 -2 a))
#f
> (andmap + '(1 2 3) '(4 5 6))
9

```

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

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

- 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.

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

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?
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
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
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)

```

4.10.4 List Filtering

(**filter** pred lst) → list?
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?
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), with *v* as the first argument and an element in *lst* as the second argument. If no element in *lst* is equal to *v* (according to *proc*), *lst* is returned unchanged.

Examples:

```
> (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)
> (remove "B" (list "a" "B" "b" "B"))
'("a" "A" "B")
> (remove 5 (list 1 2 3 2 4))
'(1 2 3 2 4)
```

Changed in version 8.2.0.2 of package `base`: Guaranteed that the output is `eq?` to *lst* if no removal occurs.

```
(remq v lst) → list?
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?
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)
```

```
(remw v lst) → list?
v : any/c
lst : list?
```

Returns (`remove v lst equal-always?`).

Examples:

```
> (remw 2 (list 1 2 3 4 5))
'(1 3 4 5)
> (remw '2) (list '(1) '(2) '(3)))
'((1) (3))
> (remw "2" (list "1" "2" "3"))
'("1" "3")
> (remw #\c (list #\a #\b #\c))
'(#\a #\b)
> (define b1 (box 5))
> (define b2 (box 5))
> (remw b2 (list @ b1 b2 2))
'(@ #\#5 1 2)
```

Added in version 8.5.0.3 of package `base`.

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

Changed in version 8.2.0.2 of package `base`: Guaranteed that the output is `eq?` to *lst* if no removal occurs.

```
(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)
```

```
(remw* v-lst lst) → list?                                procedure
  v-lst : list?
  lst : list?
```

Returns (`remove* v-lst lst equal-always?`).

Examples:

```
> (remw* (list 1 2) (list 1 2 3 2 4 5 2))
'(3 4 5)
> (define b1 (box 5))
> (define b2 (box 5))
> (remw* (list b2) (list 0 b1 1 b2 2 b2 3))
'(0 #b5 1 2 3)
```

Added in version 8.5.0.3 of package `base`.

```
(sort lst
      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<=?`).

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.

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.

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"))
```

4.10.5 List Searching

```
(member v lst [is-equal?]) → (or/c #f list? any/c)          procedure
  v : any/c
  lst : (or/c list? any/c)
  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`.

The `lst` argument need not actually be a list; `lst` must merely start with a chain of pairs until a matching element is found. If no matching element is found, then `lst` must be a list (and not a cyclic list). The result can be a non-list in the case that an element is found and the returned tail of `lst` is a non-list.

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:569:0 x> #<syntax:eval:569:0 y>)
```

```
> (member #'a (list #'x #'y) free-identifier=?)
#f
> (member 'b '(a b . etc))
'(b . etc)
```

```
(memw v lst) → (or/c #f list? any/c)                                procedure
v : any/c
lst : (or/c list? any/c)
```

Like `member`, but finds an element using `equal-always?`.

Examples:

```
> (memw 2 (list 1 2 3 4))
'(2 3 4)
> (memw 9 (list 1 2 3 4))
#f
> (define b1 (box 5))
> (define b2 (box 5))
> (memw b2 (list 0 b1 1 b2 2))
'(#b5 2)
```

Added in version 8.5.0.3 of package `base`.

```
(memv v lst) → (or/c #f list? any/c)                                procedure
v : any/c
lst : (or/c list? any/c)
```

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 #f list? any/c)                                procedure
v : any/c
lst : (or/c list? any/c)
```

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 #f list? any/c)                                procedure
proc : procedure?
lst : (or/c list? any/c)
```

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 : (or/c (listof pair?) any/c)
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`.

The `lst` argument need not actually be a list of pairs; `lst` must merely start with a chain of pairs contains pairs until a matching element is found. If no matching element is found, then `lst` must be a list of pairs (and not a cyclic list).

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)
```

```
(assw v lst) → (or/c pair? #f)                                procedure
v : any/c
lst : (or/c (listof pair?) any/c)
```

Like `assoc`, but finds an element using `equal-always?`.

Example:

```
> (assw 3 (list (list 1 2) (list 3 4) (list 5 6)))  
'(3 4)  
> (define b1 (box 0))  
> (define b2 (box 0))  
> (assw b2 (list (cons b1 1) (cons b2 2)))  
'(#b0 . 2)
```

Added in version 8.5.0.3 of package `base`.

```
(assv v lst) → (or/c pair? #f) procedure  
v : any/c  
lst : (or/c (listof pair?) any/c)
```

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 : (or/c (listof pair?) any/c)
```

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 pair? #f) procedure  
proc : procedure?  
lst : (or/c (listof pair?) any/c)
```

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)
```

4.10.6 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
```

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

Returns `(car (cdr v))`.

Example:

```
> (cadar '((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
```

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

procedure

Returns (`car (car (cdr v))`).

Example:

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

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

procedure

Returns (`car (cdr (car v))`).

Example:

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

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

procedure

Returns (`car (cdr (cdr v))`).

Example:

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

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

procedure

Returns (`cdr (car (car v))`).

Example:

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

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

procedure

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

procedure

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

procedure

Returns (`cdr (cdr (cdr v))`).

Example:

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

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

procedure

Returns (`car (car (car (car v)))`).

Example:

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

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

procedure

Returns (`car (car (car (cdr v)))`).

Example:

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

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

procedure

Returns (`car (car (cdr (car v)))`).

Example:

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

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

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

Example:

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

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

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

Example:

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

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

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

Example:

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

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

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

Example:

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

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

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

Example:

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

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

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

Example:

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

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

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

Example:

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

```
(cdadaar v) → any/c  
v : (cons/c (cons/c (cons/c pair? any/c) any/c) any/c)
```

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

Example:

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

```
(cdadadr v) → any/c  
v : (cons/c (cons/c (cons/c (cons/c pair? any/c) any/c) any/c))
```

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

procedure

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

procedure

Returns (`cdr (cdr (cdr (cdr v)))`).

Example:

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

4.10.7 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?
v : any/c
```

procedure

The same as (`pair? v`).

Example:

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

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

procedure

The same as (`null? v`).

Examples:

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

```
(first lst) → any/c
lst : list?
```

procedure

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?
lst : list?
```

procedure

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
lst : list?
```

procedure

Returns the second element of the list.

Example:

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

```
(third lst) → any
lst : list?
```

procedure

Returns the third element of the list.

Example:

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

```
(fourth lst) → any  
lst : list?
```

procedure

Returns the fourth element of the list.

Example:

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

```
(fifth lst) → any  
lst : list?
```

procedure

Returns the fifth element of the list.

Example:

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

```
(sixth lst) → any  
lst : list?
```

procedure

Returns the sixth element of the list.

Example:

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

```
(seventh lst) → any  
lst : list?
```

procedure

Returns the seventh element of the list.

Example:

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

```
(eighth lst) → any  
lst : list?
```

procedure

Returns the eighth element of the list.

Example:

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

```
(ninth lst) → any  
lst : list?
```

procedure

Returns the ninth element of the list.

Example:

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

```
(tenth lst) → any  
lst : list?
```

procedure

Returns the tenth element of the list.

Example:

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

```
(last lst) → any  
lst : list?
```

procedure

Returns the last element of the list.

This function takes time proportional to the length of *lst*.

Example:

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

```
(last-pair p) → pair?  
p : pair?
```

procedure

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

This function takes time proportional to the “length” of *p*.

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*)).

This function takes time proportional to *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*.

This function takes time proportional to *pos*.

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

Like `index-of`, but returns 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.

This function takes time proportional to *pos*.

Examples:

```
> (take '(1 2 3 4 5) 2)
'(1 2)
```

```
(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, but it will still take time proportional to *pos*.

```
(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.

This function takes time proportional to the length of *lst*.

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.

This function takes time proportional to the length of *lst*.

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
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, but it will still take time proportional to the length of *lst*.

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
lst : any/c
pred : procedure?
(dropf-right lst pred) → list?
lst : any/c
pred : procedure?
(splitf-at-right lst pred) → list? any/c
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?
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?
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?
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?
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
v
first-second-line second-third-line)
```

```

#: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 = (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
```

```
(remove-duplicates lst)
  [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 (negative? x) (abs x))) '(1 2 -3 -4 8))
'(3 4)
```

```
(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.

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

Similar to `in-inclusive-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` (included) is reached. If no `step` argument is provided, 1 is used.

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

Examples:

```
> (inclusive-range 10 20)
'(10 11 12 13 14 15 16 17 18 19 20)
> (inclusive-range 20 40 2)
'((20 22 24 26 28 30 32 34 36 38 40)
> (inclusive-range 20 10 -1)
'((20 19 18 17 16 15 14 13 12 11 10)
> (inclusive-range 10 15 1.5)
'(10 11.5 13.0 14.5)
```

Added in version 8.0.0.13 of package `base`.

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

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

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.

[Wikipedia combinations](#)

Examples:

```
> (combinations '(1 2 3))
'((() (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: 10 real time: 5 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?
  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
  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
  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?)
  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?)
  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)
'((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?
  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?
  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.

4.10.8 Immutable Cyclic Data

```
(make-reader-graph v) → any/c  
v : any/c
```

Returns a value like *v*, with *placeholders* created by `make-placeholder` replaced with the values that they contain, and with *hash 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?  
v : any/c
```

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

```
(make-placeholder v) → placeholder?  
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?  
ph : placeholder?  
datum : any/c
```

Changes the value of *ph* to *v*.

```
(placeholder-get ph) → any/c  
ph : placeholder?
```

Returns the value of *ph*.

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

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

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

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

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

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

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

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

```
(make-hashalw-placeholder assocs) → hash-placeholder?  
assocs : (listof pair?)
```

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

Added in version 8.5.0.3 of package `base`.