Draft 12.24, X3J13/92-102.
Thu 9-Apr-1992 2:33am EDT

Programming Language—Common Lisp

17. Sequences

17.1 Sequence Concepts

Sequences can be created by the *function* **make-sequence**, as well as other *functions* that create *objects* of *types* that are *subtypes* of **sequence** (*e.g.*, **list**, **make-list**, **mapcar**, and **vector**).

A ***sequence function*** is a *function* defined by this specification or added as an extension by the *implementation* that operates on one or more *sequences*. Whenever a *sequence function* must construct and return a new *vector*, it always returns a *simple vector*. Similarly, any *strings* constructed will be *simple strings*.

concatenate	**length**	**remove**
copy-seq	**map**	**remove-duplicates**
count	**map-into**	**remove-if**
count-if	**merge**	**remove-if-not**
count-if-not	**mismatch**	**replace**
delete	**notany**	**reverse**
delete-duplicates	**notevery**	**search**
delete-if	**nreverse**	**some**
delete-if-not	**nsubstitute**	**sort**
elt	**nsubstitute-if**	**stable-sort**
every	**nsubstitute-if-not**	**subseq**
fill	**position**	**substitute**
find	**position-if**	**substitute-if**
find-if	**position-if-not**	**substitute-if-not**
find-if-not	**reduce**	

Figure 17–1. Standardized Sequence Functions

17.2 Rules about Test Functions

17.2.1 Satisfying a Two-Argument Test

When an *object* O is being considered iteratively against each *element* E_i of a *sequence* S by an *operator* F listed in Figure 17–2, it is sometimes useful to control the way in which the presence of O is tested in S is tested by F. This control is offered on the basis of a *function* designated with either a `:test` or `:test-not` *argument*.

adjoin	**nset-exclusive-or**	**search**
assoc	**nsublis**	**set-difference**
count	**nsubst**	**set-exclusive-or**
delete	**nsubstitute**	**sublis**
find	**nunion**	**subsetp**
intersection	**position**	**subst**
member	**pushnew**	**substitute**
mismatch	**rassoc**	**tree-equal**
nintersection	**remove**	**union**
nset-difference	**remove-duplicates**	

Figure 17–2. Operators that have Two-Argument Tests to be Satisfied

The object O might not be compared directly to E_i. If a `:key` *argument* is provided, it is a *designator* for a *function* of one *argument* to be called with each E_i as an *argument*, and *yielding* an *object* Z_i to be used for comparison. (If there is no `:key` *argument*, Z_i is E_i.)

The *function* designated by the `:key` *argument* is never called on O itself. However, if the function operates on multiple sequences (*e.g.*, as happens in **set-difference**), O will be the result of calling the `:key` function on an *element* of the other sequence.

A `:test` *argument*, if supplied to F, is a *designator* for a *function* of two *arguments*, O and Z_i. An E_i is said (or, sometimes, an O and an E_i are said) to ***satisfy the test*** if this `:test` *function* returns a *boolean* representing *true*.

A `:test-not` *argument*, if supplied to F, is *designator* for a *function* of two *arguments*, O and Z_i. An E_i is said (or, sometimes, an O and an E_i are said) to ***satisfy the test*** if this `:test-not` *function* returns a *boolean* representing *false*.

If neither a `:test` nor a `:test-not` *argument* is supplied, it is as if a `:test` argument of `#'eql` was supplied.

The consequences are unspecified if both a `:test` and a `:test-not` *argument* are supplied in the same *call* to F.

17.2.1.1 Examples of Satisfying a Two-Argument Test

```
 (remove "FOO" '(foo bar "FOO" "BAR" "foo" "bar") :test #'equal)
→ (foo bar "BAR" "foo" "bar")
 (remove "FOO" '(foo bar "FOO" "BAR" "foo" "bar") :test #'equalp)
→ (foo bar "BAR" "bar")
 (remove "FOO" '(foo bar "FOO" "BAR" "foo" "bar") :test #'string-equal)
→ (bar "BAR" "bar")
 (remove "FOO" '(foo bar "FOO" "BAR" "foo" "bar") :test #'string=)
→ (BAR "BAR" "foo" "bar")

 (remove 1 '(1 1.0 #C(1.0 0.0) 2 2.0 #C(2.0 0.0)) :test-not #'eql)
→ (1)
 (remove 1 '(1 1.0 #C(1.0 0.0) 2 2.0 #C(2.0 0.0)) :test-not #'=)
→ (1 1.0 #C(1.0 0.0))
 (remove 1 '(1 1.0 #C(1.0 0.0) 2 2.0 #C(2.0 0.0)) :test (complement #'=))
→ (1 1.0 #C(1.0 0.0))

 (count 1 '((one 1) (uno 1) (two 2) (dos 2)) :key #'cadr) → 2

 (count 2.0 '(1 2 3) :test #'eql :key #'float) → 1

 (count "FOO" (list (make-pathname :name "FOO" :type "X")
                    (make-pathname :name "FOO" :type "Y"))
       :key #'pathname-name
       :test #'equal)
→ 2
```

17.2.2 Satisfying a One-Argument Test

When using one of the *functions* in Figure 17–3, the elements E of a *sequence* S are filtered not on the basis of the presence or absence of an object O under a two *argument predicate*, as with the *functions* described in Section 17.2.1 (Satisfying a Two-Argument Test), but rather on the basis of a one *argument predicate*.

assoc-if	**member-if**	**rassoc-if**
assoc-if-not	**member-if-not**	**rassoc-if-not**
count-if	**nsubst-if**	**remove-if**
count-if-not	**nsubst-if-not**	**remove-if-not**
delete-if	**nsubstitute-if**	**subst-if**
delete-if-not	**nsubstitute-if-not**	**subst-if-not**
find-if	**position-if**	**substitute-if**
find-if-not	**position-if-not**	**substitute-if-not**

Figure 17–3. Operators that have One-Argument Tests to be Satisfied

The element E_i might not be considered directly. If a `:key` *argument* is provided, it is a *designator* for a *function* of one *argument* to be called with each E_i as an *argument*, and *yielding* an *object* Z_i to be used for comparison. (If there is no `:key` *argument*, Z_i is E_i.)

Functions defined in this specification and having a name that ends in "`-if`" accept a first *argument* that is a *designator* for a *function* of one *argument*, Z_i. An E_i is said to ***satisfy the test*** if this `:test` *function* returns a *boolean* representing *true*.

Functions defined in this specification and having a name that ends in "`-if-not`" accept a first *argument* that is a *designator* for a *function* of one *argument*, Z_i. An E_i is said to ***satisfy the test*** if this `:test` *function* returns a *boolean* representing *false*.

17.2.2.1 Examples of Satisfying a One-Argument Test

```
 (count-if #'zerop '(1 #C(0.0 0.0) 0 0.0d0 0.0s0 3)) → 4

 (remove-if-not #'symbolp '(0 1 2 3 4 5 6 7 8 9 A B C D E F))
→ (A B C D E F)
 (remove-if (complement #'symbolp) '(0 1 2 3 4 5 6 7 8 9 A B C D E F))
→ (A B C D E F)

 (count-if #'zerop '("foo" "" "bar" "" "" "baz" "quux") :key #'length)
→ 3
```

sequence *System Class*

Class Precedence List:

sequence, **t**

Description:

Sequences are ordered collections of *objects*, called the *elements* of the *sequence*.

The *types* **vector** and the *type* **list** are *disjoint subtypes* of *type* **sequence**, but are not necessarily an *exhaustive partition* of *sequence*.

When viewing a *vector* as a *sequence*, only the *active elements* of that *vector* are considered *elements* of the *sequence*; that is, *sequence* operations respect the *fill pointer* when given *sequences* represented as *vectors*.

copy-seq *Function*

Syntax:

copy-seq *sequence* → *copied-sequence*

Arguments and Values:

sequence—a *proper sequence*.

copied-sequence—a *proper sequence*.

Description:

Creates a copy of *sequence*. The *elements* of the new *sequence* are the *same* as the corresponding *elements* of the given *sequence*.

If *sequence* is a *vector*, the result is a *fresh simple array* of *rank* one that has the same *actual array element type* as *sequence*. If *sequence* is a *list*, the result is a *fresh list*.

Examples:

```
(setq str "a string") → "a string"
(equalp str (copy-seq str)) → true
(eql str (copy-seq str)) → false
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*.

See Also:

copy-list

Notes:

From a functional standpoint, `(copy-seq x)` ≡ `(subseq x 0)`

However, the programmer intent is typically very different in these two cases.

elt *Accessor*

Syntax:

elt *sequence index* → *object*

(setf (**elt** *sequence index*) *new-object*)

Arguments and Values:

sequence—a *proper sequence*.

index—a *valid sequence index* for *sequence*.

object—an *object*.

new-object—an *object*.

Description:

Accesses the *element* of *sequence* specified by *index*.

Examples:

```
(setq str (copy-seq "0123456789")) → "0123456789"
(elt str 6) → #\6
(setf (elt str 0) #\#) → #\#
str → "#123456789"
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*. Should signal an error of *type* **type-error** if *index* is not a *valid sequence index* for *sequence*.

See Also:

aref, **nth**, Section 3.2.1 (Terminology)

Notes:

aref may be used to *access vector* elements that are beyond the *vector*'s *fill pointer*.

fill *Function*

Syntax:

fill *sequence item* &key *start end* → *sequence*

Arguments and Values:

sequence—a *proper sequence*.

item—a *sequence*.

start, *end*—*bounding index designators* of *sequence*. The defaults for *start* and *end* are 0 and **nil**, respectively.

Description:

Replaces the *elements* of *sequence* *bounded* by *start* and *end* with *item*.

Examples:

```
(fill (list 0 1 2 3 4 5) '(444)) → ((444) (444) (444) (444) (444) (444))
(fill (copy-seq "01234") #\e :start 3) → "012ee"
(setq x (vector 'a 'b 'c 'd 'e)) → #(A B C D E)
(fill x 'z :start 1 :end 3) → #(A Z Z D E)
x → #(A Z Z D E)
(fill x 'p) → #(P P P P P)
x → #(P P P P P)
```

Side Effects:

Sequence is destructively modified.

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*. Should signal an error of *type* **type-error** if *start* is not a non-negative *integer*. Should signal an error of *type* **type-error** if *end* is not a non-negative *integer* or **nil**.

See Also:

replace, **nsubstitute**

Notes:

(fill *sequence item*) ≡(nsubstitute-if *item* (constantly t) *sequence*)

make-sequence

make-sequence *Function*

Syntax:

make-sequence *result-type size* &key *initial-element* → *sequence*

Arguments and Values:

result-type—a **sequence** *type specifier*.

size—a non-negative *integer*.

initial-element—an *object*. The default is *implementation-dependent*.

sequence—a *proper sequence*.

Description:

Returns a *sequence* of the type *result-type* and of length *size*, each of the *elements* of which has been initialized to *initial-element*.

If the *result-type* is a *subtype* of **list**, the result will be a *list*.

If the *result-type* is a *subtype* of **vector**, then if the implementation can determine the element type specified for the *result-type*, the element type of the resulting array is the result of *upgrading* that element type; or, if the implementation can determine that the element type is unspecified (or *), the element type of the resulting array is **t**; otherwise, an error is signaled.

Examples:

```
 (make-sequence 'list 0) → ()
 (make-sequence 'string 26 :initial-element #\.)
→ ".........................."
 (make-sequence '(vector double-float) 2
                :initial-element 1d0)
→ #(1.0d0 1.0d0)


 (make-sequence '(vector * 2) 3) should signal an error
 (make-sequence '(vector * 4) 3) should signal an error
```

Affected By:

The *implementation*.

Exceptional Situations:

The consequences are unspecified if *initial-element* is not an *object* which can be stored in the resulting *sequence*.

An error of *type* **type-error** must be signaled if the *result-type* is neither a *recognizable subtype* of **list**, nor a *recognizable subtype* of **vector**.

An error of *type* **type-error** should be signaled if *result-type* specifies the number of elements and *size* is different from that number.

See Also:

make-array, **make-list**

Notes:

```
(make-sequence 'string 5) ≡ (make-string 5)
```

subseq *Accessor*

Syntax:

subseq *sequence start* &optional *end* → *subsequence*

(**setf** (**subseq** *sequence start* &optional *end*) *new-subsequence*)

Arguments and Values:

sequence—a *proper sequence*.

start, *end*—*bounding index designators* of *sequence*. The default for *end* is **nil**.

subsequence—a *proper sequence*.

new-subsequence—a *proper sequence*.

Description:

subseq creates a *sequence* that is a copy of the subsequence of *sequence bounded* by *start* and *end*.

Start specifies an offset into the original *sequence* and marks the beginning position of the subsequence. *end* marks the position following the last element of the subsequence.

subseq always allocates a new *sequence* for a result; it never shares storage with an old *sequence*. The result subsequence is always of the same *type* as *sequence*.

If *sequence* is a *vector*, the result is a *fresh simple array* of *rank* one that has the same *actual array element type* as *sequence*. If *sequence* is a *list*, the result is a *fresh list*.

setf may be used with **subseq** to destructively replace *elements* of a subsequence with *elements* taken from a *sequence* of new values. If the subsequence and the new sequence are not of equal length, the shorter length determines the number of elements that are replaced. The remaining *elements* at the end of the longer sequence are not modified in the operation.

Examples:

```
(setq str "012345") → "012345"
(subseq str 2) → "2345"
(subseq str 3 5) → "34"
(setf (subseq str 4) "abc") → "abc"
str → "0123ab"
(setf (subseq str 0 2) "A") → "A"
str → "A123ab"
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*. Should be prepared to signal an error of *type* **type-error** if *new-subsequence* is not a *proper sequence*.

See Also:

replace

map *Function*

Syntax:

map *result-type function* &rest *sequences*$^+$ → *result*

Arguments and Values:

result-type – a **sequence** *type specifier*, or **nil**.

function—a *function designator*. *function* must take as many arguments as there are *sequences*.

sequence—a *proper sequence*.

result—if *result-type* is a *type specifier* other than **nil**, then a *sequence* of the *type* it denotes; otherwise (if the *result-type* is **nil**), **nil**.

Description:

Applies *function* to successive sets of arguments in which one argument is obtained from each *sequence*. The *function* is called first on all the elements with index 0, then on all those with index 1, and so on. The *result-type* specifies the *type* of the resulting *sequence*.

map returns **nil** if *result-type* is **nil**. Otherwise, **map** returns a *sequence* such that element j is the result of applying *function* to element j of each of the *sequences*. The result *sequence* is as long as the shortest of the *sequences*. The consequences are undefined if the result of applying *function* to the successive elements of the *sequences* cannot be contained in a *sequence* of the *type* given by *result-type*.

If the *result-type* is a *subtype* of **list**, the result will be a *list*.

If the *result-type* is a *subtype* of **vector**, then if the implementation can determine the element type specified for the *result-type*, the element type of the resulting array is the result of *upgrading* that element type; or, if the implementation can determine that the element type is unspecified (or *), the element type of the resulting array is **t**; otherwise, an error is signaled.

Examples:

```
(map 'string #'(lambda (x y)
                 (char "01234567890ABCDEF" (mod (+ x y) 16)))
      '(1 2 3 4)
      '(10 9 8 7)) → "AAAA"
(setq seq '("lower" "UPPER" "" "123")) → ("lower" "UPPER" "" "123")
(map nil #'nstring-upcase seq) → NIL
seq → ("LOWER" "UPPER" "" "123")
(map 'list #'- '(1 2 3 4)) → (-1 -2 -3 -4)
(map 'string
     #'(lambda (x) (if (oddp x) #\1 #\0))
     '(1 2 3 4)) → "1010"


(map '(vector * 4) #'cons "abc" "de") should signal an error
```

Exceptional Situations:

An error of *type* **type-error** must be signaled if the *result-type* is not a *recognizable subtype* of **list**, not a *recognizable subtype* of **vector**, and not **nil**.

Should be prepared to signal an error of *type* **type-error** if any *sequence* is not a *proper sequence*.

An error of *type* **type-error** should be signaled if *result-type* specifies the number of elements and the minimum length of the *sequences* is different from that number.

See Also:

Section 3.6 (Traversal Rules and Side Effects)

map-into

map-into *Function*

Syntax:

map-into *result-sequence function* &rest *sequences* → *result-sequence*

Arguments and Values:

result-sequence—a *proper sequence*.

function—a *designator* for a *function* of as many *arguments* as there are *sequences*.

sequence—a *proper sequence*.

Description:

Destructively modifies *result-sequence* to contain the results of applying *function* to each element in the argument *sequences* in turn.

result-sequence and each element of *sequences* can each be either a *list* or a *vector*. If *result-sequence* and each element of *sequences* are not all the same length, the iteration terminates when the shortest *sequence* (of any of the *sequences* or the *result-sequence*) is exhausted. If *result-sequence* is a *vector* with a *fill pointer*, the *fill pointer* is ignored when deciding how many iterations to perform, and afterwards the *fill pointer* is set to the number of times *function* was applied. If *result-sequence* is longer than the shortest element of *sequences*, extra elements at the end of *result-sequence* are left unchanged. If *result-sequence* is **nil**, **map-into** immediately returns **nil**, since **nil** is a *sequence* of length zero.

If *function* has side effects, it can count on being called first on all of the elements with index 0, then on all of those numbered 1, and so on.

Examples:

```
(setq a (list 1 2 3 4) b (list 10 10 10 10)) → (10 10 10 10)
(map-into a #'+ a b) → (11 12 13 14)
a → (11 12 13 14)
b → (10 10 10 10)
(setq k '(one two three)) → (ONE TWO THREE)
(map-into a #'cons k a) → ((ONE . 11) (TWO . 12) (THREE . 13) 14)
(map-into a #'gensym) → (#:G9090 #:G9091 #:G9092 #:G9093)
a → (#:G9090 #:G9091 #:G9092 #:G9093)
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *result-sequence* is not a *proper sequence*. Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*.

Notes:

map-into differs from **map** in that it modifies an existing *sequence* rather than creating a new one. In addition, **map-into** can be called with only two arguments, while **map** requires at least three arguments.

map-into could be defined by:

```
(defun map-into (result-sequence function &rest sequences)
  (loop for index below (apply #'min
                               (length result-sequence)
                               (mapcar #'length sequences))
        do (setf (elt result-sequence index)
                 (apply function
                        (mapcar #'(lambda (seq) (elt seq index))
                                sequences))))
  result-sequence)
```

reduce *Function*

Syntax:

reduce *function sequence* &key *key from-end start end initial-value* → *result*

Arguments and Values:

function—a *designator* for a *function* that might be called with either zero or two *arguments*.

sequence—a *proper sequence*.

key—a *designator* for a *function* of one argument, or **nil**.

from-end—a *boolean*. The default is *false*.

start, *end*—*bounding index designators* of *sequence*. The defaults for *start* and *end* are 0 and **nil**, respectively.

initial-value—an *object*.

result—an *object*.

Description:

reduce uses a binary operation, *function*, to combine the *elements* of *sequence bounded* by *start* and *end*.

The *function* must accept as *arguments* two *elements* of *sequence* or the results from combining those *elements*. The *function* must also be able to accept no arguments.

If *key* is supplied, it is used is used to extract the values to reduce. The *key* function is applied exactly once to each element of *sequence* in the order implied by the reduction order but not to the value of *initial-value*, if supplied. The *key* function typically returns part of the *element* of *sequence*. If *key* is not supplied or is **nil**, the *sequence element* itself is used.

The reduction is left-associative, unless *from-end* is *true* in which case it is right-associative.

If *initial-value* is supplied, it is logically placed before the subsequence (or after it if *from-end* is *true*) and included in the reduction operation.

In the normal case, the result of **reduce** is the combined result of *function*'s being applied to successive pairs of *elements* of *sequence*. If the subsequence contains exactly one *element* and no *initial-value* is given, then that *element* is returned and *function* is not called. If the subsequence is empty and an *initial-value* is given, then the *initial-value* is returned and *function* is not called. If the subsequence is empty and no *initial-value* is given, then the *function* is called with zero arguments, and **reduce** returns whatever *function* does. This is the only case where the *function* is called with other than two arguments.

Examples:

```
 (reduce #'* '(1 2 3 4 5)) → 120
 (reduce #'append '((1) (2)) :initial-value '(i n i t)) → (I N I T 1 2)
 (reduce #'append '((1) (2)) :from-end t
                             :initial-value '(i n i t)) → (1 2 I N I T)
 (reduce #'- '(1 2 3 4)) ≡ (- (- (- 1 2) 3) 4) → -8
 (reduce #'- '(1 2 3 4) :from-end t)    ;Alternating sum.
≡ (- 1 (- 2 (- 3 4))) → -2
 (reduce #'+ '()) → 0
 (reduce #'+ '(3)) → 3
 (reduce #'+ '(foo)) → FOO
 (reduce #'list '(1 2 3 4)) → (((1 2) 3) 4)
 (reduce #'list '(1 2 3 4) :from-end t) → (1 (2 (3 4)))
 (reduce #'list '(1 2 3 4) :initial-value 'foo) → ((((foo 1) 2) 3) 4)
 (reduce #'list '(1 2 3 4)
        :from-end t :initial-value 'foo) → (1 (2 (3 (4 foo))))
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*.

See Also:

Section 3.6 (Traversal Rules and Side Effects)

count, count-if, count-if-not *Function*

Syntax:

count *item sequence* `&key` *from-end start end key test test-not* → *n*

count-if *predicate sequence* `&key` *from-end start end key* → *n*

count-if-not *predicate sequence* `&key` *from-end start end key* → *n*

Arguments and Values:

item—an *object*.

sequence—a *proper sequence*.

predicate—a *designator* for a *function* of one *argument* that returns a *boolean*.

from-end—a *boolean*. The default is *false*.

test—a *designator* for a *function* of two *arguments* that returns a *boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *boolean*.

start, *end*—*bounding index designators* of *sequence*. The defaults for *start* and *end* are `0` and **nil**, respectively.

key—a *designator* for a *function* of one argument, or **nil**.

n—a non-negative *integer* less than or equal to the *length* of *sequence*.

Description:

count, **count-if**, and **count-if-not** count and return the number of *elements* in the *sequence* *bounded* by *start* and *end* that *satisfy the test*.

The *from-end* has no direct effect on the result. However, if *from-end* is *true*, the *elements* of *sequence* will be supplied as *arguments* to the *test*, *test-not*, and *key* in reverse order, which may change the side-effects, if any, of those functions.

Examples:

```
(count #\a "how many A's are there in here?") → 2
(count-if-not #'oddp '((1) (2) (3) (4)) :key #'car) → 2
(count-if #'upper-case-p "The Crying of Lot 49" :start 4) → 2
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*.

See Also:

Section 17.2 (Rules about Test Functions), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` *argument* is deprecated.

The *function* **count-if-not** is deprecated.

length *Function*

Syntax:

length *sequence* → *n*

Arguments and Values:

sequence—a *proper sequence*.

n—a non-negative *integer*.

Description:

Returns the number of *elements* in *sequence*.

If *sequence* is a *vector* with a *fill pointer*, the active length as specified by the *fill pointer* is returned.

Examples:

```
(length "abc") → 3
(setq str (make-array '(3) :element-type 'character
                            :initial-contents "abc"
                            :fill-pointer t)) → "abc"
(length str) → 3
(setf (fill-pointer str) 2) → 2
(length str) → 2
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*.

See Also:

list-length, **sequence**

Notes:

reverse, nreverse *Function*

Syntax:

reverse *sequence* → *reversed-sequence*

nreverse *sequence* → *reversed-sequence*

Arguments and Values:

sequence—a *proper sequence*.

reversed-sequence—a *sequence*.

Description:

reverse and **nreverse** return a new *sequence* of the same kind as *sequence*, containing the same *elements*, but in reverse order.

reverse and **nreverse** differ in that **reverse** always creates and returns a new *sequence*, whereas **nreverse** might modify and return the given *sequence*. **reverse** never modifies the given *sequence*.

For **reverse**, if *sequence* is a *vector*, the result is a *fresh simple array* of *rank* one that has the same *actual array element type* as *sequence*. If *sequence* is a *list*, the result is a *fresh list*.

For **nreverse**, if *sequence* is a *vector*, the result is a *vector* that has the same *actual array element type* as *sequence*. If *sequence* is a *list*, the result is a *list*.

For **nreverse**, *sequence* might be destroyed and re-used to produce the result. The result might or might not be *identical* to *sequence*. Specifically, when *sequence* is a *list*, **nreverse** is permitted to **setf** any part, **car** or **cdr**, of any *cons* that is part of the *list structure* of *sequence*. When *sequence* is a *vector*, **nreverse** is permitted to re-order the elements of *sequence* in order to produce the resulting *vector*.

Examples:

```
(setq str "abc") → "abc"
(reverse str) → "cba"
str → "abc"
(setq str (copy-seq str)) → "abc"
(nreverse str) → "cba"
str → implementation-dependent
(setq l (list 1 2 3)) → (1 2 3)
(nreverse l) → (3 2 1)
l → implementation-dependent
```

Side Effects:

nreverse might either create a new *sequence*, modify the argument *sequence*, or both. (**reverse** does not modify *sequence*.)

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*.

sort, stable-sort *Function*

Syntax:

sort *sequence predicate* &key *key* → *sorted-sequence*

stable-sort *sequence predicate* &key *key* → *sorted-sequence*

Arguments and Values:

sequence—a *proper sequence*.

predicate—a *designator* for a *function* of two arguments that returns a *boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

sorted-sequence—a *sequence*.

Description:

sort and **stable-sort** destructively sort *sequences* according to the order determined by the *predicate* function.

If *sequence* is a *vector*, the result is a *vector* that has the same *actual array element type* as *sequence*. The result might or might not be simple, and might or might not be *identical* to *sequence*. If *sequence* is a *list*, the result is a *list*.

sort determines the relationship between two elements by giving keys extracted from the elements to the *predicate*. The first argument to the *predicate* function is the part of one element of *sequence* extracted by the *key* function (if supplied); the second argument is the part of another element of *sequence* extracted by the *key* function (if supplied). *Predicate* should return *true* if and only if the first argument is strictly less than the second (in some appropriate sense). If the first argument is greater than or equal to the second (in the appropriate sense), then the *predicate* should return *false*.

The argument to the *key* function is the *sequence* element. The return value of the *key* function becomes an argument to *predicate*. If *key* is not supplied or **nil**, the *sequence* element itself is used. There is no guarantee on the number of times the *key* will be called.

If the *key* and *predicate* always return, then the sorting operation will always terminate, producing a *sequence* containing the same *elements* as *sequence* (that is, the result is a permutation of *sequence*). This is guaranteed even if the *predicate* does not really consistently represent a total order (in which case the *elements* will be scrambled in some unpredictable way, but no *element* will be lost). If the *key* consistently returns meaningful keys, and the *predicate* does reflect some total ordering criterion on those keys, then the *elements* of the *sorted-sequence* will be properly sorted according to that ordering.

The sorting operation performed by **sort** is not guaranteed stable. Elements considered equal by the *predicate* might or might not stay in their original order. The *predicate* is assumed to consider two elements x and y to be equal if (funcall *predicate x y*) and (funcall *predicate y x*) are both *false*. **stable-sort** guarantees stability.

The sorting operation can be destructive in all cases. In the case of a *vector* argument, this is accomplished by permuting the elements in place. In the case of a *list*, the *list* is destructively reordered in the same manner as for **nreverse**.

Examples:

```
 (setq tester (copy-seq "lkjashd")) → "lkjashd"
 (sort tester #'char-lessp) → "adhjkls"
 (setq tester (list '(1 2 3) '(4 5 6) '(7 8 9))) → ((1 2 3) (4 5 6) (7 8 9))
 (sort tester #'> :key #'car)  → ((7 8 9) (4 5 6) (1 2 3))
 (setq tester (list 1 2 3 4 5 6 7 8 9 0)) → (1 2 3 4 5 6 7 8 9 0)
 (stable-sort tester #'(lambda (x y) (and (oddp x) (evenp y))))
→ (1 3 5 7 9 2 4 6 8 0)
 (sort (setq committee-data
             (vector (list (list "JonL" "White") "Iteration")
                     (list (list "Dick" "Waters") "Iteration")
                     (list (list "Dick" "Gabriel") "Objects")
                     (list (list "Kent" "Pitman") "Conditions")
                     (list (list "Gregor" "Kiczales") "Objects")
                     (list (list "David" "Moon") "Objects")
                     (list (list "Kathy" "Chapman") "Editorial")
                     (list (list "Larry" "Masinter") "Cleanup")
                     (list (list "Sandra" "Loosemore") "Compiler")))
       #'string-lessp :key #'cadar)
→ #((("Kathy" "Chapman") "Editorial")
    (("Dick" "Gabriel") "Objects")
    (("Gregor" "Kiczales") "Objects")
    (("Sandra" "Loosemore") "Compiler")
    (("Larry" "Masinter") "Cleanup")
    (("David" "Moon") "Objects")
    (("Kent" "Pitman") "Conditions")
    (("Dick" "Waters") "Iteration")
    (("JonL" "White") "Iteration"))
 ;; Note that individual alphabetical order within ‘committees’
 ;; is preserved.
 (setq committee-data
       (stable-sort committee-data #'string-lessp :key #'cadr))
→ #((("Larry" "Masinter") "Cleanup")
```

```
(("Sandra" "Loosemore") "Compiler")
(("Kent" "Pitman") "Conditions")
(("Kathy" "Chapman") "Editorial")
(("Dick" "Waters") "Iteration")
(("JonL" "White") "Iteration")
(("Dick" "Gabriel") "Objects")
(("Gregor" "Kiczales") "Objects")
(("David" "Moon") "Objects"))
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*.

See Also:

merge, Section 3.2.1 (Terminology), Section 3.6 (Traversal Rules and Side Effects), Section 3.7 (Destructive Operations)

find, find-if, find-if-not *Function*

Syntax:

find *item sequence* &key *from-end test test-not start end key* → *element*

find-if *predicate sequence* &key *from-end start end key* → *element*

find-if-not *predicate sequence* &key *from-end start end key* → *element*

Arguments and Values:

item—an *object*.

sequence—a *proper sequence*.

predicate—a *designator* for a *function* of one *argument* that returns a *boolean*.

from-end—a *boolean*. The default is *false*.

test—a *designator* for a *function* of two *arguments* that returns a *boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *boolean*.

start, *end*—*bounding index designators* of *sequence*. The defaults for *start* and *end* are 0 and **nil**, respectively.

key—a *designator* for a *function* of one argument, or **nil**.

element—an *element* of the *sequence*, or **nil**.

Description:

find, **find-if**, and **find-if-not** each search for an *element* of the *sequence* *bounded* by *start* and *end* that *satisfies the predicate* *predicate* or that *satisfies the test* *test* or *test-not*, as appropriate.

If *from-end* is *true*, then the result is the rightmost *element* that *satisfies the test*.

If the *sequence* contains an *element* that *satisfies the test*, then the leftmost or rightmost *sequence* element, depending on *from-end*, is returned; otherwise **nil** is returned.

Examples:

```
 (find #\d "here are some letters that can be looked at" :test #'char>)
→ #\Space
 (find-if #'oddp '(1 2 3 4 5) :end 3 :from-end t) → 3
 (find-if-not #'complexp
            '#(3.5 2 #C(1.0 0.0) #C(0.0 1.0))
            :start 2) → NIL
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*.

See Also:

position, Section 17.2 (Rules about Test Functions), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` *argument* is deprecated.

The *function* **find-if-not** is deprecated.

position, position-if, position-if-not *Function*

Syntax:

position *item sequence* `&key` *from-end test test-not start end key* → *position*

position-if *predicate sequence* `&key` *from-end start end key* → *position*

position-if-not *predicate sequence* `&key` *from-end start end key* → *position*

Arguments and Values:

item—an *object*.

sequence—a *proper sequence*.

predicate—a *designator* for a *function* of one argument that returns a *boolean*.

from-end—a *boolean*. The default is *false*.

test—a *designator* for a *function* of two *arguments* that returns a *boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *boolean*.

start, *end*—*bounding index designators* of *sequence*. The defaults for *start* and *end* are 0 and **nil**, respectively.

key—a *designator* for a *function* of one argument, or **nil**.

position—a *bounding index* of *sequence*, or **nil**.

Description:

position, **position-if**, and **position-if-not** each search *sequence* for an *element* that *satisfies the test*.

The *position* returned is the index within *sequence* of the leftmost (if *from-end* is *true*) or of the rightmost (if *from-end* is *false*) *element* that *satisfies the test*; otherwise **nil** is returned. The index returned is relative to the left-hand end of the entire *sequence*, regardless of the value of *start*, *end*, or *from-end*.

Examples:

```
(position #\a "baobab" :from-end t) → 4
(position-if #'oddp '((1) (2) (3) (4)) :start 1 :key #'car) → 2
(position 595 '()) → NIL
(position-if-not #'integerp '(1 2 3 4 5.0)) → 4
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*.

See Also:

find, Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` *argument* is deprecated.

The *function* **position-if-not** is deprecated.

search *Function*

Syntax:

search *sequence-1 sequence-2* &key *from-end test test-not*
key start1 start2
end1 end2

→ *position*

Arguments and Values:

Sequence-1—a *sequence*.

Sequence-2—a *sequence*.

from-end—a *boolean*. The default is *false*.

test—a *designator* for a *function* of two *arguments* that returns a *boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

start1, *end1*—*bounding index designators* of *sequence-1*. The defaults for *start1* and *end1* are 0 and **nil**, respectively.

start2, *end2*—*bounding index designators* of *sequence-2*. The defaults for *start2* and *end2* are 0 and **nil**, respectively.

position—a *bounding index* of *sequence-2*, or **nil**.

Description:

Searches *sequence-2* for a subsequence that matches *sequence-1*.

The implementation may choose to search *sequence-2* in any order; there is no guarantee on the number of times the test is made. For example, when *start-end* is *true*, the *sequence* might actually be searched from left to right instead of from right to left (but in either case would return the rightmost matching subsequence). If the search succeeds, **search** returns the offset into *sequence-2* of the first element of the leftmost or rightmost matching subsequence, depending on *from-end*; otherwise **search** returns **nil**.

If *from-end* is *true*, the index of the leftmost element of the rightmost matching subsequence is returned.

Examples:

```
(search "dog" "it's a dog's life") → 7
(search '(0 1) '(2 4 6 1 3 5) :key #'oddp) → 2
```

See Also:

Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` *argument* is deprecated.

mismatch *Function*

Syntax:

mismatch *sequence-1 sequence-2* &key *from-end test test-not key start1 start2 end1 end2*
→ *position*

Arguments and Values:

Sequence-1—a *sequence*.

Sequence-2—a *sequence*.

from-end—a *boolean*. The default is *false*.

test—a *designator* for a *function* of two *arguments* that returns a *boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *boolean*.

start1, *end1*—*bounding index designators* of *sequence-1*. The defaults for *start1* and *end1* are `0` and **nil**, respectively.

start2, *end2*—*bounding index designators* of *sequence-2*. The defaults for *start2* and *end2* are `0` and **nil**, respectively.

key—a *designator* for a *function* of one argument, or **nil**.

position—a *bounding index* of *sequence-1*, or **nil**.

Description:

The specified subsequences of *sequence-1* and *sequence-2* are compared element-wise.

The *key* argument is used for both the *sequence-1* and the *sequence-2*.

If *sequence-1* and *sequence-2* are of equal length and match in every element, the result is *false*. Otherwise, the result is a non-negative *integer*, the index within *sequence-1* of the leftmost or rightmost position, depending on *from-end*, at which the two subsequences fail to match. If one subsequence is shorter than and a matching prefix of the other, the result is the index relative to *sequence-1* beyond the last position tested.

If *from-end* is *true*, then one plus the index of the rightmost position in which the *sequences* differ is returned. In effect, the subsequences are aligned at their right-hand ends; then, the last elements are compared, the penultimate elements, and so on. The index returned is an index relative to *sequence-1*.

Examples:

```
(mismatch "abcd" "ABCDE" :test #'char-equal) → 4
(mismatch '(3 2 1 1 2 3) '(1 2 3) :from-end t) → 3
(mismatch '(1 2 3) '(2 3 4) :test-not #'eq :key #'oddp) → NIL
(mismatch '(1 2 3 4 5 6) '(3 4 5 6 7) :start1 2 :end2 4) → NIL
```

See Also:

Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` *argument* is deprecated.

replace *Function*

Syntax:

replace *sequence-1 sequence-2* `&key` *start1 end1 start2 end2* → *sequence-1*

Arguments and Values:

sequence-1—a *sequence*.

sequence-2—a *sequence*.

start1, *end1*—*bounding index designators* of *sequence-1*. The defaults for *start1* and *end1* are `0` and **nil**, respectively.

start2, *end2*—*bounding index designators* of *sequence-2*. The defaults for *start2* and *end2* are `0` and **nil**, respectively.

Description:

Destructively modifies *sequence-1* by replacing the *elements* of *subsequence-1 bounded* by *start1* and *end1* with the *elements* of *subsequence-2 bounded* by *start2* and *end2*.

Sequence-1 is destructively modified by copying successive *elements* into it from *sequence-2*. *Elements* of the subsequence of *sequence-2 bounded* by *start2* and *end2* are copied into the subsequence of *sequence-1 bounded* by *start1* and *end1*. If these subsequences are not of the same length, then the shorter length determines how many *elements* are copied; the extra *elements* near the end of the longer subsequence are not involved in the operation. The number of elements copied can be expressed as:

```
(min (- end1 start1) (- end2 start2))
```

If *sequence-1* and *sequence-2* are the *same object* and the region being modified overlaps the region being copied from, then it is as if the entire source region were copied to another place and only then copied back into the target region. However, if *sequence-1* and *sequence-2* are not the same, but the region being modified overlaps the region being copied from (perhaps because of shared list structure or displaced *arrays*), then after the **replace** operation the subsequence of *sequence-1* being modified will have unpredictable contents. It is an error if the elements of *sequence-2* are not of a *type* that can be stored into *sequence-1*.

Examples:

```
 (replace "abcdefghij" "0123456789" :start1 4 :end1 7 :start2 4)
→ "abcd456hij"
 (setq lst "012345678") → "012345678"
 (replace lst lst :start1 2 :start2 0) → "010123456"
 lst → "010123456"
```

Side Effects:

The *sequence-1* is modified.

See Also:

fill

Notes:

substitute, substitute-if, substitute-if-not, nsubstitute, nsubstitute-if, nsubstitute-if-not *Function*

Syntax:

substitute *newitem olditem sequence* &key *from-end test test-not start end count key*

→ *result-sequence*

substitute-if *newitem predicate sequence* &key *from-end start end count key*
→ *result-sequence*

substitute-if-not *newitem predicate sequence* &key *from-end start end count key*
→ *result-sequence*

nsubstitute *newitem olditem sequence* &key *from-end test test-not start end count key*
→ *sequence*

nsubstitute-if *newitem predicate sequence* &key *from-end start end count key*
→ *sequence*

nsubstitute-if-not *newitem predicate sequence* &key *from-end start end count key*
→ *sequence*

Arguments and Values:

newitem—an *object*.

olditem—an *object*.

sequence—a *proper sequence*.

predicate—a *designator* for a *function* of one *argument* that returns a *boolean*.

from-end—a *boolean*. The default is *false*.

test—a *designator* for a *function* of two *arguments* that returns a *boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *boolean*.

start, *end*—*bounding index designators* of *sequence*. The defaults for *start* and *end* are 0 and **nil**, respectively.

count—an *integer* or **nil**. The default is **nil**.

key—a *designator* for a *function* of one argument, or **nil**.

result-sequence—a *sequence*.

Description:

substitute, **substitute-if**, and **substitute-if-not** return a copy of *sequence* in which each *element* that *satisfies the test* has been replaced with *newitem*.

nsubstitute, **nsubstitute-if**, and **nsubstitute-if-not** are like **substitute**, **substitute-if**, and **substitute-if-not** respectively, but they may modify *sequence*.

If *sequence* is a *vector*, the result is a *vector* that has the same *actual array element type* as *sequence*. The result might or might not be simple, and might or might not be *identical* to *sequence*. If *sequence* is a *list*, the result is a *list*.

Count, if supplied, limits the number of elements altered; if more than *count elements satisfy the test*, then of these *elements* only the leftmost or rightmost, depending on *from-end*, are replaced, as many as specified by *count*. If *count* is supplied and negative, the behavior is as if zero had been supplied instead. If *count* is **nil**, all matching items are affected.

Supplying a *from-end* of *true* matters only when the *count* is provided (and *non-nil*); in that case, only the rightmost *count elements satisfying the test* are removed (instead of the leftmost).

predicate, *test*, and *test-not* might be called more than once for each *sequence element*, and their

substitute, substitute-if, substitute-if-not, ...

side effects can happen in any order.

The result of all these functions is a *sequence* of the same *type* as *sequence* that has the same elements except that those in the subsequence *bounded* by *start* and *end* and *satisfying the test* have been replaced by *newitem*.

substitute, **substitute-if**, and **substitute-if-not** return a *sequence* which can share with *sequence* or may be *identical* to the input *sequence* if no elements need to be changed.

nsubstitute and **nsubstitute-if** are required to **setf** any **car** (if *sequence* is a *list*) or **aref** (if *sequence* is a *vector*) of *sequence* that is required to be replaced with *newitem*. If *sequence* is a *list*, none of the *cdrs* of the top-level *list* can be modified.

Examples:

```
 (substitute #\. #\SPACE "0 2 4 6") → "0.2.4.6"
 (substitute 9 4 '(1 2 4 1 3 4 5)) → (1 2 9 1 3 9 5)
 (substitute 9 4 '(1 2 4 1 3 4 5) :count 1) → (1 2 9 1 3 4 5)
 (substitute 9 4 '(1 2 4 1 3 4 5) :count 1 :from-end t)
→ (1 2 4 1 3 9 5)
 (substitute 9 3 '(1 2 4 1 3 4 5) :test #'>) → (9 9 4 9 3 4 5)

 (substitute-if 0 #'evenp '((1) (2) (3) (4)) :start 2 :key #'car)
→ ((1) (2) (3) 0)
 (substitute-if 9 #'oddp '(1 2 4 1 3 4 5)) → (9 2 4 9 9 4 9)
 (substitute-if 9 #'evenp '(1 2 4 1 3 4 5) :count 1 :from-end t)
→ (1 2 4 1 3 9 5)

 (setq some-things (list 'a 'car 'b 'cdr 'c)) → (A CAR B CDR C)
 (nsubstitute-if "function was here" #'fboundp some-things
                 :count 1 :from-end t) → (A CAR B "function was here" C)
 some-things → (A CAR B "function was here" C)
 (setq alpha-tester (copy-seq "ab ")) → "ab "
 (nsubstitute-if-not #\z #'alpha-char-p alpha-tester) → "abz"
 alpha-tester → "abz"
```

Side Effects:

nsubstitute, **nsubstitute-if**, and **nsubstitute-if-not** modify *sequence*.

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*.

See Also:

subst, **nsubst**, Section 3.2.1 (Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` *argument* is deprecated.

The functions **substitute-if-not** and **nsubstitute-if-not** are deprecated.

nsubstitute and **nsubstitute-if** can be used in for-effect-only positions in code.

concatenate *Function*

Syntax:

concatenate *result-type* &rest *sequences* → *result-sequence*

Arguments and Values:

result-type—a **sequence** *type specifier*.

sequences—a *sequence*.

result-sequence—a *proper sequence* of *type result-type*.

Description:

concatenate returns a *sequence* that contains all the individual elements of all the *sequences* in the order that they are supplied. The *sequence* is of type *result-type*, which must be a *subtype* of *type* **sequence**.

All of the *sequences* are copied from; the result does not share any structure with any of the *sequences*. Therefore, if only one *sequence* is provided and it is of type *result-type*, **concatenate** is required to copy *sequence* rather than simply returning it.

It is an error if any element of the *sequences* cannot be an element of the *sequence* result.

If the *result-type* is a *subtype* of **list**, the result will be a *list*.

If the *result-type* is a *subtype* of **vector**, then if the implementation can determine the element type specified for the *result-type*, the element type of the resulting array is the result of *upgrading* that element type; or, if the implementation can determine that the element type is unspecified (or `*`), the element type of the resulting array is **t**; otherwise, an error is signaled.

Examples:

```
(concatenate 'string "all" " " "together" " " "now") → "all together now"
(concatenate 'list "ABC" '(d e f) #(1 2 3) #*1011)
→ (#\A #\B #\C D E F 1 2 3 1 0 1 1)
(concatenate 'list) → NIL

  (concatenate '(vector * 2) "a" "bc") should signal an error
```

Exceptional Situations:

An error is signaled if the *result-type* is neither a *recognizable subtype* of **list**, nor a *recognizable subtype* of **vector**.

An error of *type* **type-error** should be signaled if *result-type* specifies the number of elements and the sum of *sequences* is different from that number.

See Also:

append

merge *Function*

Syntax:

merge *result-type sequence-1 sequence-2 predicate* &key *key* → *result-sequence*

Arguments and Values:

result-type—a **sequence** *type specifier*.

sequence-1—a *sequence*.

sequence-2—a *sequence*.

predicate—a *designator* for a *function* of two arguments that returns a *boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

result-sequence—a *proper sequence* of *type* *result-type*.

Description:

Destructively merges *sequence-1* with *sequence-2* according to an order determined by the *predicate*. **merge** determines the relationship between two elements by giving keys extracted from the sequence elements to the *predicate*.

The first argument to the *predicate* function is an element of *sequence-1* as returned by the *key* (if supplied); the second argument is an element of *sequence-2* as returned by the *key* (if supplied). *Predicate* should return *true* if and only if its first argument is strictly less than the second (in some appropriate sense). If the first argument is greater than or equal to the second (in the appropriate sense), then *predicate* should return *false*. **merge** considers two elements `x` and `y` to be equal if `(funcall predicate x y)` and `(funcall predicate y x)` both *yield false*.

The argument to the *key* is the *sequence* element. Typically, the return value of the *key* becomes the argument to *predicate*. If *key* is not supplied or **nil**, the sequence element itself is used. The *key* may be executed more than once for each *sequence element*, and its side effects may occur in any order.

merge

If *key* and *predicate* return, then the merging operation will terminate. The result of merging two *sequences* `x` and `y` is a new *sequence* of type *result-type* `z`, such that the length of `z` is the sum of the lengths of `x` and `y`, and `z` contains all the elements of `x` and `y`. If `x1` and `x2` are two elements of `x`, and `x1` precedes `x2` in `x`, then `x1` precedes `x2` in `z`, and similarly for elements of `y`. In short, `z` is an interleaving of `x` and `y`.

If `x` and `y` were correctly sorted according to the *predicate*, then `z` will also be correctly sorted. If `x` or `y` is not so sorted, then `z` will not be sorted, but will nevertheless be an interleaving of `x` and `y`.

The merging operation is guaranteed stable; if two or more elements are considered equal by the *predicate*, then the elements from *sequence-1* will precede those from *sequence-2* in the result.

sequence-1 and/or *sequence-2* may be destroyed.

If the *result-type* is a *subtype* of **list**, the result will be a *list*.

If the *result-type* is a *subtype* of **vector**, then if the implementation can determine the element type specified for the *result-type*, the element type of the resulting array is the result of *upgrading* that element type; or, if the implementation can determine that the element type is unspecified (or `*`), the element type of the resulting array is **t**; otherwise, an error is signaled.

Examples:

```
 (setq test1 (list 1 3 4 6 7))
 (setq test2 (list 2 5 8))
 (merge 'list test1 test2 #'<) → (1 2 3 4 5 6 7 8)
 (setq test1 (copy-seq "BOY"))
 (setq test2 (copy-seq :nosy"))
 (merge 'string test1 test2 #'char-lessp) → "BnOosYy"
 (setq test1 (vector ((red . 1) (blue . 4))))
 (setq test2 (vector ((yellow . 2) (green . 7))))
 (merge 'vector test1 test2 #'< :key #'cdr)
→ #((RED . 1) (YELLOW . 2) (BLUE . 4) (GREEN . 7))

 (merge '(vector * 4) '(1 5) '(2 4 6) #'<) should signal an error
```

Exceptional Situations:

An error must be signaled if the *result-type* is neither a *recognizable subtype* of **list**, nor a *recognizable subtype* of **vector**.

An error of *type* **type-error** should be signaled if *result-type* specifies the number of elements and the sum of the lengths of *sequence-1* and *sequence-2* is different from that number.

See Also:

sort, **stable-sort**, Section 3.2.1 (Terminology), Section 3.6 (Traversal Rules and Side Effects)

remove, remove-if, remove-if-not, delete, delete-if, delete-if-not *Function*

Syntax:

remove *item sequence* &key *from-end test test-not start end count key* → *result-sequence*

remove-if *test sequence* &key *from-end start end count key* → *result-sequence*

remove-if-not *test sequence* &key *from-end start end count key* → *result-sequence*

delete *item sequence* &key *from-end test test-not start end count key* → *result-sequence*

delete-if *test sequence* &key *from-end start end count key* → *result-sequence*

delete-if-not *test sequence* &key *from-end start end count key* → *result-sequence*

Arguments and Values:

item—an *object*.

sequence—a *proper sequence*.

test—a *designator* for a *function* of one *argument* that returns a *boolean*.

from-end—a *boolean*. The default is *false*.

test—a *designator* for a *function* of two *arguments* that returns a *boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *boolean*.

start, *end*—*bounding index designators* of *sequence*. The defaults for *start* and *end* are 0 and **nil**, respectively.

count—an *integer* or **nil**. The default is **nil**.

key—a *designator* for a *function* of one argument, or **nil**.

result-sequence—a *sequence*.

Description:

remove, **remove-if**, and **remove-if-not** return a *sequence* from which the elements that *satisfy the test* have been removed.

remove, remove-if, remove-if-not, delete, delete-if, ...

delete, **delete-if**, and **delete-if-not** are like **remove**, **remove-if**, and **remove-if-not** respectively, but they may modify *sequence*.

If *sequence* is a *vector*, the result is a *vector* that has the same *actual array element type* as *sequence*. The result might or might not be simple, and might or might not be *identical* to *sequence*. If *sequence* is a *list*, the result is a *list*.

Supplying a *from-end* of *true* matters only when the *count* is provided; in that case only the rightmost *count* elements *satisfying the test* are deleted.

Count, if supplied, limits the number of elements removed or deleted; if more than *count* elements *satisfy the test*, then of these elements only the leftmost or rightmost, depending on *from-end*, are deleted or removed, as many as specified by *count*. If *count* is supplied and negative, the behavior is as if zero had been supplied instead. If *count* is **nil**, all matching items are affected.

For all these functions, elements not removed or deleted occur in the same order in the result as they did in *sequence*.

remove, **remove-if**, **remove-if-not** return a *sequence* of the same *type* as *sequence* that has the same elements except that those in the subsequence *bounded* by *start* and *end* and *satisfying the test* have been removed. This is a non-destructive operation. If any elements need to be removed, the result will be a copy. The result of **remove** may share with *sequence*; the result may be *identical* to the input *sequence* if no elements need to be removed.

delete, **delete-if**, and **delete-if-not** return a *sequence* of the same *type* as *sequence* that has the same elements except that those in the subsequence *bounded* by *start* and *end* and *satisfying the test* have been deleted. *Sequence* may be destroyed and used to construct the result; however, the result might or might not be *identical* to *sequence*.

delete, when *sequence* is a *list*, is permitted to **setf** any part, **car** or **cdr**, of the top-level list structure in that *sequence*. When *sequence* is a *vector*, **delete** is permitted to change the dimensions of the *vector* and to slide its elements into new positions without permuting them to produce the resulting *vector*.

delete-if is constrained to behave exactly as follows:

```
(delete nil sequence
            :test #'(lambda (ignore item) (funcall test item))
            ...)
```

Examples:

```
(remove 4 '(1 3 4 5 9)) → (1 3 5 9)
(remove 4 '(1 2 4 1 3 4 5)) → (1 2 1 3 5)
(remove 4 '(1 2 4 1 3 4 5) :count 1) → (1 2 1 3 4 5)
(remove 4 '(1 2 4 1 3 4 5) :count 1 :from-end t) → (1 2 4 1 3 5)
(remove 3 '(1 2 4 1 3 4 5) :test #'>) → (4 3 4 5)
```

remove, remove-if, remove-if-not, delete, delete-if, ...

```
(setq lst '(list of four elements)) → (LIST OF FOUR ELEMENTS)
(setq lst2 (copy-seq lst)) → (LIST OF FOUR ELEMENTS)
(setq lst3 (delete 'four lst)) → (LIST OF ELEMENTS)
(equal lst lst2) → false
(remove-if #'oddp '(1 2 4 1 3 4 5)) → (2 4 4)
(remove-if #'evenp '(1 2 4 1 3 4 5) :count 1 :from-end t)
→ (1 2 4 1 3 5)
(remove-if-not #'evenp '(1 2 3 4 5 6 7 8 9) :count 2 :from-end t)
→ (1 2 3 4 5 6 8)
(setq tester (list 1 2 4 1 3 4 5)) → (1 2 4 1 3 4 5)
(delete 4 tester) → (1 2 1 3 5)
(setq tester (list 1 2 4 1 3 4 5)) → (1 2 4 1 3 4 5)
(delete 4 tester :count 1) → (1 2 1 3 4 5)
(setq tester (list 1 2 4 1 3 4 5)) → (1 2 4 1 3 4 5)
(delete 4 tester :count 1 :from-end t) → (1 2 4 1 3 5)
(setq tester (list 1 2 4 1 3 4 5)) → (1 2 4 1 3 4 5)
(delete 3 tester :test #'>) → (4 3 4 5)
(setq tester (list 1 2 4 1 3 4 5)) → (1 2 4 1 3 4 5)
(delete-if #'oddp tester) → (2 4 4)
(setq tester (list 1 2 4 1 3 4 5)) → (1 2 4 1 3 4 5)
(delete-if #'evenp tester :count 1 :from-end t) → (1 2 4 1 3 5)
(setq tester (list 1 2 3 4 5 6)) → (1 2 3 4 5 6)
(delete-if #'evenp tester) → (1 3 5)
tester → implementation-dependent

(setq foo (list 'a 'b 'c)) → (A B C)
(setq bar (cdr foo)) → (B C)
(setq foo (delete 'b foo)) → (A C)
bar → ((C)) or ...
(eq (cdr foo) (car bar)) → T or ...
```

Side Effects:

For **delete**, **delete-if**, and **delete-if-not**, *sequence* may be destroyed and used to construct the result.

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *sequence* is not a *proper sequence*.

See Also:

Section 3.2.1 (Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` *argument* is deprecated.

The functions **delete-if-not** and **remove-if-not** are deprecated.

remove-duplicates, delete-duplicates *Function*

Syntax:

remove-duplicates *sequence* &key *from-end test test-not start end key*

→ *result-sequence*

delete-duplicates *sequence* &key *from-end test test-not start end key*

→ *result-sequence*

Arguments and Values:

sequence—a *proper sequence*.

from-end—a *boolean*. The default is *false*.

test—a *designator* for a *function* of two *arguments* that returns a *boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *boolean*.

start, *end*—*bounding index designators* of *sequence*. The defaults for *start* and *end* are 0 and **nil**, respectively.

key—a *designator* for a *function* of one argument, or **nil**.

result-sequence—a *sequence*.

Description:

remove-duplicates returns a modified copy of *sequence* from which any element that matches another element occurring in *sequence* has been removed.

If *sequence* is a *vector*, the result is a *vector* that has the same *actual array element type* as *sequence*. The result might or might not be simple, and might or might not be *identical* to *sequence*. If *sequence* is a *list*, the result is a *list*.

delete-duplicates is like **remove-duplicates**, but **delete-duplicates** may modify *sequence*.

The elements of *sequence* are compared *pairwise*, and if any two match, then the one occurring earlier in *sequence* is discarded, unless *from-end* is *true*, in which case the one later in *sequence* is discarded.

remove-duplicates, delete-duplicates

remove-duplicates and **delete-duplicates** return a *sequence* of the same *type* as *sequence* with enough elements removed so that no two of the remaining elements match. The order of the elements remaining in the result is the same as the order in which they appear in *sequence*.

remove-duplicates returns a *sequence* that may share with *sequence* or may be *identical* to *sequence* if no elements need to be removed.

delete-duplicates, when *sequence* is a *list*, is permitted to **setf** any part, **car** or **cdr**, of the top-level list structure in that *sequence*. When *sequence* is a *vector*, **delete-duplicates** is permitted to change the dimensions of the *vector* and to slide its elements into new positions without permuting them to produce the resulting *vector*.

Examples:

```
(remove-duplicates "aBcDAbCd" :test #'char-equal :from-end t) → "aBcD"
(remove-duplicates '(a b c b d d e)) → (A C B D E)
(remove-duplicates '(a b c b d d e) :from-end t) → (A B C D E)
(remove-duplicates '((foo #\a) (bar #\%) (baz #\A))
    :test #'char-equal :key #'cadr) → ((BAR #\%) (BAZ #\A))
(remove-duplicates '((foo #\a) (bar #\%) (baz #\A))
    :test #'char-equal :key #'cadr :from-end t) → ((FOO #\a) (BAR #\%))
(setq tester (list 0 1 2 3 4 5 6))
(delete-duplicates tester :key #'oddp :start 1 :end 6) → (0 4 5 6)
```

Side Effects:

delete-duplicates might destructively modify *sequence*.

Exceptional Situations:

Should signal an error of *type* **type-error** if *sequence* is not a *proper sequence*.

See Also:

Section 3.2.1 (Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` *argument* is deprecated.

These functions are useful for converting *sequence* into a canonical form suitable for representing a set.

Table of Contents