9. Generic Sequence Functions

The type specifier **sequence** is defined to include lists and vectors (arrays of rank one). Lists and vectors are similar in that both can be regarded as sequences of elements: there is a first element, a second element, and so on. Element *n* of a list is (nth *n list*), and element *n* of a vector is (aref *vector n*). Many useful operations which apply in principle to a sequence of objects can work equally well on lists and vectors. These are the generic sequence functions.

All the generic sequence functions accept nil as a sequence of length zero.

9.1 Primitive Sequence Operations

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

Creates a sequence of type *type*, *size* elements long. *size* must be an integer and *type* must be either list or some kind of array type. *type* could be just array or vector to make a general vector, it could be (vector (byte 8)) to make an art-8b vector, and so on.

If *initial-element* is specified, each element of the new sequence contains *initial-element*. Otherwise, the new sequence is initialized to contain nil if that is possible, zero otherwise (for numeric array types).

```
(make-sequence 'list 3)
  =>  (nil nil nil)

(make-sequence 'array 5 :initial-element t)
  =>  #(t t t t t)

(make-sequence '(vector bit) 5)
  =>  #*00000
```

elt *sequence index*

Returns the element at index *index* in *sequence*. If *sequence* is a list, this is (nth *index sequence*). If *sequence* is a vector, this is (aref *index sequence*). Being microcoded, elt is as fast as either nth or aref.

(setf (elt *sequence index*) *value*) is the way to set an element of a sequence.

length *sequence*

Returns the length of *sequence*, as an integer. For a vector with a fill pointer, this is the fill pointer value. For a list, it is the traditional Lisp function; note that lists ending with atoms other than nil are accepted, so that the length of (a b . c) is 2.

9.2 Simple Sequence Operations

copy-seq *sequence*

Returns a new sequence of the same type, length and contents as *sequence*.

concatenate *result-type* &rest *sequences*

Returns a new sequence, of type *result-type*, whose contents are made from the contents of all the *sequences*. *result-type* can be list or any array type, just as in make-sequence above. Examples:

```
(concatenate 'list '(1 2) '#(A 3))  =>  (1 2 A 3)
(concatenate 'vector '(1 2) '#(A 3)  =>  #(1 2 A 3)
```

subseq *sequence start* &optional *end*

Returns a new sequence whose elements are a subsequence of *sequence*. The new sequence is of the same type as *sequence*.

start is the index of the first element of *sequence* to take. *end* is the index of where to stop—the first element *not* to take. *end* can also be nil, meaning take everything from *start* up to the end of *sequence*.

Examples:

```
(subseq "Foobar" 3 5)  =>  "ba"
(subseq '(a b c) 1)  =>  (b c)
```

It is also possible to setf a call to subseq. This means to store into part of the sequence passed to subseq. Thus,

```
(setf (subseq "Foobar" 3 5) "le")
```

modifies the string "Foobar" so that it contains "Fooler" instead.

replace *into-sequence-1 from-sequence-2* &key (*start1* 0) *end1* (*start2* 0) *end2*

Copies part of *from-sequence-2* into part of *to-sequence-1*. *start2* and *end2* are the indices of the part of *from-sequence-2* to copy from, and *start1* and *end1* are the indices of the part of *to-sequence-1* to copy into.

If the number of elements to copy out of *from-sequence-2* is less than the number of elements of *to-sequence-1* to be copied into, the extra elements of *to-sequence-1* are not changed. If the number of elements to copy out is more than there is room for, the last extra elements are ignored.

If the two sequence arguments are the same sequence, then the elements to be copied are copied first into a temporary sequence (if necessary) to make sure that no element is overwritten before it is copied. Example:

```
(setq str "Elbow")
(replace str str :start1 2 :end1 5 :start2 1 :end2 4)
```

modifies str to contain "Ellbo".

into-sequence-1 is returned as the value of replace.

fill *sequence item* &key (*start* 0) *end*

Modifies the contents of *sequence* by setting all the elements to *item*. *start* and *end* may be specified to limit the operation to some contiguous portion of *sequence*; then the elements before *start* or after *end* are unchanged. If *end* is nil, the filling goes to the end of *sequence*.

The value returned by fill is *sequence*. Example:

```
(setq l '(a b c d e))
(fill l 'lose :start 2)

l => (a b lose lose lose)
```

reverse *sequence*

Returns a new sequence containing the same elements as *sequence* but in reverse order. The new sequence is of the same type and length as *sequence*. reverse does not modify its argument, unlike nreverse which is faster but does modify its argument. The list created by reverse is not cdr-coded.

```
(reverse "foo")  =>  "oof"
(reverse '(a b (c d) e)) => (e (c d) b a)
```

nreverse *sequence*

Modifies *sequence* destructively to have its elements in reverse order, and returns *sequence* as modified. For a vector, this is done by copying the elements to different positions. For a list, this is done by modifying cdr pointers. This has two important consequences: it is most efficient when the list is not cdr-coded, and the rearranged list starts with the cell that used to be at the end. Although the altered list as a whole contains the same cells as the original, the actual value of the altered list is not eq to the original list. For this reason, one must always store the value of nreverse into the place where the list will be used. Do not just use nreverse for effect on a list.

```
(setq a '#(1 2 3 4 5))
(nreverse a)
(concatenate 'list a) => (5 4 3 2 1)

(setq b '(1 2 3 4 5)
      c b
      d (last b))
(setq b (nreverse b))

b => (5 4 3 2 1)
c => (1)
(eq b d)  =>  t
```

nreverse is most frequently used after a loop which computes elements for a new list one by one. These elements can be put on the new list with push, but this produces a list which has the elements in reverse order (first one generated at the end of the list).

```
(let (accumulate)
  (dolist (x input)
    (push (car x) accumulate)
    (push (cdr x) accumulate))
  (nreverse accumulate))
```

Currently, **nreverse** is inefficient with cdr-coded lists (see section 5.4, page 100), because it just uses **rplacd** in the straightforward way. This may be fixed someday. In the meantime **reverse** might be preferable in some cases.

9.3 Mapping On Sequences

cli:map *result-type function* &rest *sequences*

The Common Lisp **map** function maps *function* over successive elements of each *sequence*, constructing and returning a sequence of the results that *function* returns. The constructed sequence is of type *result-type* (see **make-sequence**, page 188).

function is called first on the first elements of all the sequences, then on the second elements of all, and so on until some argument sequence is exhausted.

```
(map 'list 'list '(1 2 3) '#(A B C D))
  =>  ((1 A) (2 B) (3 C))

(setq vect (map '(vector (mod 16.)) '+
                '(3 4 5 6 7) (circular-list 1)))
(concatenate 'list vect)  =>  (2 3 4 5 6)
(array-element-type vect)  =>  (mod 16.)
```

result-type can also be **nil**. Then the values returned by *function* are thrown away, no sequence is constructed, and **map** returns **nil**.

This function is available under the name **map** in Common Lisp programs. In traditional Zetalisp programs, **map** is another function which does something related but different; see page 84. Traditional programs can call this function as **cli:map**.

cli:some *predicate* &rest *sequences*

Applies *predicate* to successive elements of each sequence. If *predicate* ever returns a non-nil value, **cli:some** immediately returns the same value. If one of the argument sequences is exhausted, **cli:some** returns **nil**.

Each time *predicate* is called, it receives one argument from each sequence. The first time, it gets the first element of each sequence, then the second element of each, and so on until a sequence is exhausted. Examples:

```
(cli:some 'plusp '(-4 0 5 6))  =>  5
(cli:some '> '(-4 0 5 6) '(0 12 12 12))  =>  nil
(cli:some '> '(-4 0 5 6) '(3 3 3 3))  =>  5
(cli:some '> '(-4 0 5 6) '(3 3))  =>  nil
```

This function is available under the name **some** in Common Lisp programs. In traditional Zetalisp programs, **some** is another function which does something related but different; see page 106. Traditional programs can call this function as **cli:some**.

cli:every *predicate* &rest *sequences*

Applies *predicate* to successive elements of each sequence. If *predicate* ever returns **nil**, **cli:every** immediately returns **nil**. If one of the argument sequences is exhausted, **cli:every** returns **t**.

Each time *predicate* is called, it receives one argument from each sequence. The first time, it gets the first element of each sequence, then the second element of each, and so on until a sequence is exhausted. Examples:

```
(cli:every 'plusp '(-4 0 5 6))  =>  nil
(cli:every 'plusp '(5 6))  =>  t
```

This function is available under the name **every** in Common Lisp programs. In traditional Zetalisp programs, **every** is another function which does something related but different; see page 106. Traditional programs can call this function as **cli:every**.

notany *predicate* &rest *sequences*
notevery *predicate* &rest *sequences*

These are the opposites of **cli:some** and **cli:every**.

(notany ...) is equivalent to **(not (cli:some ...))**.
(notevery ...) is equivalent to **(not (cli:every ...))**.

reduce *function sequence* &key *from-end* (*start* 0) *end initial-value*

Combines the elements of *sequence* using *function*, a function of two args. *function* is applied to the first two elements; then to that result and the third element; then to that result and the fourth element; and so on.

start and *end* are indices that restrict the action to a part of *sequence*, as if the rest of *sequence* were not there. They default to 0 and **nil** (**nil** for *end* means go all the way to the end of *sequence*).

If *from-end* is non-**nil**, processing starts with the last of the elements. *function* is first applied to the last two elements; then to the previous element and that result; then to the previous element and that result; and so on until element number *start* has been used.

If *initial-value* is specified, it acts like an extra element of *sequence*, used in addition to the actual elements of the specified part of *sequence*. It comes, in effect, at the beginning if *from-end* is **nil**, but at the end if *from-end* is non-**nil**, so that in any case it is the first element to be processed.

If there is only one element to be processed, that element is returned and *function* is not called.

If there are no elements (*sequence* is of length zero and no *initial-value*), *function* is called with no arguments and its value is returned.

Examples:

```
(reduce '+ '(1 2 3))  =>  6
(reduce '- '(1 2 3))  =>  -4
(reduce '- '(1 2 3) :from-end t)  =>  2  ;; 1 - (2 - 3)
(reduce 'cons '(1 2 3) :from-end t)  =>  (1 2 . 3)
(reduce 'cons '(1 2 3))  =>  ((1 . 2) . 3)
```

9.4 Operating on Selected Elements

The generic sequence functions for searching, substituting and removing elements from sequences take similar arguments whose meanings are standard. This is because they all look at each element of the sequence to decide whether it should be processed.

Functions which conceptually modify the sequence come in pairs. One function in the pair copies the sequence if necessary and never modifies the argument. The copy is a list if the original sequence is a list; otherwise, the copy is an **art-q** array. If the sequence is a list, it may be copied only partially, sharing any unchanged tail with the original argument. If no elements match, the result sequence may be **eq** to the argument sequence.

The other function in the pair may alter the original sequence and return it, or may make a copy and return that.

There are two ways the function can decide which elements to operate on. The functions whose names end in **-if** or **-if-not** have an argument named *predicate* which should be a function of one argument. This function is applied to each element and the value determines whether the element is processed.

The other functions have an argument named *item* or something similar which is an object to compare each element with. The elements that match *item* are processed. By default, the comparison is done with **eql**. You can specify any function of two arguments to be used instead, as the *test* keyword argument. *item* is always the first argument, and an element of the sequence is the second argument. The element matches *item* if *test* returns non-**nil**. Alternatively, you can specify the *test-not* keyword argument; then the element matches if *test-not* returns **nil**.

The elements may be tested in any order, and may be tested more than once. For predictable results, your *predicate*, *test* and *test-not* functions should be side-effect free.

The five keyword arguments *start*, *end*, *key*, *count* and *from-end* have the same meanings for all of the functions, except that *count* is not relevant for some kinds of operations. Here is what they do:

start, end — *start* and *end* are indices in the sequence; they restrict the processing to the portion between those indices. Only elements in this portion are tested, replaced or removed. For the search functions, only this portion is searched. For element removal functions, elements outside the portion are unchanged.

start is the index of the first element to be processed, and *end* is the index of the element after the last element to be processed. *end* can also be nil, meaning that processing should continue to the end of the sequence.

start always defaults to 0, and *end* always defaults to nil.

key — *key*, if not nil, is a function of one argument which is applied to each element of the sequence to get a value which is passed to the *test*, *test-not* or *predicate* function in place of the element. For example, if *key* is car, the car of each element is compared or tested. The default for *key* is nil, which means to compare or test the element itself.

from-end — If *from-end* is non-nil, elements are (conceptually) processed in the reverse of the sequence order, from the later elements to the earlier ones. In some functions this argument makes no difference, or matters only when *count* is non-nil.

Note: the actual testing of elements may happen in any order.

count — *count*, if not nil, should be an integer specifying the number of matching elements to be processed. For example, if *count* is 2, only the first two elements that match are removed, replaced, etc. If *from-end* is non-nil, the last two matching elements are the ones removed or replaced.

The default for *count* is nil, which means all elements are tested and all matching ones are processed.

9.4.1 Removing Elements from Sequences

These functions remove certain elements of a sequence. The **remove** series functions copy the argument; the **delete** series functions can modify it destructively (currently they always copy anyway if the argument is a vector).

remove-if *predicate sequence* &key (*start* 0) *end count key from-end*
delete-if *predicate sequence* &key (*start* 0) *end count key from-end*

Returns a sequence like *sequence* but missing any elements that satisfy *predicate*. *predicate* is a function of one argument which is applied to one element at a time; if *predicate* returns non-nil, that element is removed. remove-if copies structure as necessary to avoid modifying *sequence*, while delete-if can either modify the original sequence and return it or make a copy and return that. (Currently, a list is always modified, and a vector is always copied, but don't depend on this.)

The *start*, *end*, *key count* and *from-end* arguments are handled in the standard way.

```
(remove-if 'plusp '(1 -1 2 -2 3 -3))  =>  (-1 -2 -3)
(remove-if 'plusp '(1 -1 2 -2 3 -3) :count 2)
  =>  (-1 -2 3 -3)
(remove-if 'plusp '(1 -1 2 -2 3 -3) :count 2 :from-end t)
  =>  (1 -1 -2 -3)
(remove-if 'plusp '(1 -1 2 -2 3 -3) :start 4)
  =>  (1 -1 2 -2 -3)
(remove-if 'zerop '(1 -1 2 -2 3 -3) :key '1-)
  =>  (-1 2 -2 3 -3)
```

remove-if-not *predicate sequence* &key (*start* 0) *end count key from-end*
delete-if-not *predicate sequence* &key (*start* 0) *end count key from-end*

Like remove-if and delete-if except that the elements removed are those for which *predicate* returns nil.

cli:remove *item sequence* &key (*test* 'eql) *test-not* (*start* 0) *end count key from-end*
cli:delete *item sequence* &key (*test* 'eql) *test-not* (*start* 0) *end count key from-end*

The Common Lisp functions for eliminating elements from a sequence test the elements of *sequence* one by one by comparison with *item*, using the *test* or *test-not* function, and eliminate the elements that match. cli:remove copies structure as necessary to avoid modifying *sequence*, while cli:delete can either modify the original sequence and return it or make a copy and return that. (Currently, a list is always modified, and a vector is always copied.)

The *start*, *end*, *key count* and *from-end* arguments are handled in the standard way.

```
(cli:remove 'x '(x (a) (x) (a x)))
  =>  ((a) (x) (a x))

(cli:remove 'x '((a) (x) (a x)) :test 'memq)
  =>  ((a))

(cli:remove 'x '((a) (x) (a x)) :test-not 'memq)
  =>  ((x) (a x))

(cli:remove 'x '((a) (x) (a x))
            :test 'memq :count 1)
  =>  ((a) (a x))

(cli:remove 'x '((a) (x) (a x)) :key 'car)
  =>  ((a) (a x))
```

These functions are available under the names remove and delete in Common Lisp programs. Traditional Zetalisp provides functions remove and delete which serve similar functions, on lists only, and with different calling sequences; see page 105 and page 105. Traditional programs can call these functions as cli:remove and cli:delete.

remove-duplicates *sequence* &key (*test* 'eql) *test-not* (*start* 0) *end* *key* *from-end*
delete-duplicates *sequence* &key (*test* 'eql) *test-not* (*start* 0) *end* *key* *from-end*

remove-duplicates returns a new sequence like *sequence* except that all but one of any set of matching elements have been removed. delete-duplicates is the same except that it may destructively modify and then return *sequence* itself.

Elements are compared using *test*, a function of two arguments. Two elements match if *test* returns non-nil. Each element is compared with all the following elements and slated for removal if it matches any of them.

If *test-not* is specified, it is used instead of *test*, but then elements match if *test-not* returns nil. If neither *test* nor *test-not* is specified, eql is used for *test*.

If *key* is non-nil, it should be a function of one argument. *key* is applied to each element, and the value *key* returns is passed to *test* or *test-not*.

If *from-end* is non-nil, then elements are processed (conceptually) from the end of *sequence* forward. Each element is compared with all the preceding ones and slated for removal if it matches any of them. For a well-behaved comparison function, the only difference *from-end* makes is which elements of a matching set are removed. Normally the last one is kept; with *from-end*, it is the first one that is kept.

If *start* or *end* is used to restrict processing to a portion of *sequence*, both removal and comparison are restricted. An element is removed only if it is itself within the specified portion, and matches another element within the specified portion.

9.4.2 Substitution Functions

The functions in this section substitute a new value for certain of the elements in a sequence—those that match a specified object or satisfy a predicate. For example, you could replace every t in the sequence with nil, leaving all elements other than t unchanged. The substitute series functions make a copy and return it, leaving the original sequence unmodified. The nsubstitute series functions always alter the original sequence destructively and return it. They do not use up any storage.

Note the difference between these functions and the function cli:subst. subst operates only on lists, and it searches all levels of list structure in both car and cdr positions. substitute, when given a list, considers for replacement only the elements of the list.

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

substitute-if returns a new sequence like *sequence* but with *newitem* substituted for each element of *sequence* that satisfies *predicate*. *sequence* itself is unchanged. If it is a list, only enough of it is copied to avoid changing *sequence*.

nsubstitute-if replaces elements in *sequence* itself, modifying it destructively, and returns *sequence*.

start, *end*, *key*, *count* and *from-end* are handled in the standard fashion as described above.

```
(substitute-if 0 'plusp '(1 -1 2 -2 3) :from-end t :count 2)
  =>  (1 -1 0 -2 0)
```

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

Like substitute-if and nsubstitute-if except that the elements replaced are those for which *predicate* returns nil.

substitute *newitem olditem sequence* &key (*test* 'eql) *test-not start end count key from-end*
nsubstitute *newitem olditem sequence* &key (*test* 'eql) *test-not start end count key from-end*

Like substitute-if and nsubstitute-if except that elements are tested by comparison with *olditem*, using *test* or *test-not* as a comparison function.

start, *end*, *key*, *count* and *from-end* are handled in the standard fashion as described above.

```
(substitute 'a 'b '(a b (a b)))
  =>  (a a (a b))
```

9.4.3 Searching for Elements

The functions in this section find an element or elements of a sequence which satisfy a predicate or match a specified object. The position series functions find one element and return the index of the element found in the specified sequence. The find series functions return the element itself. The count series functions find all the elements that match and returns the number of them that were found.

All of the functions accept the keyword arguments *start*, *end*, *count* and *from-end*, and handle them in the standard way described in section 9.4, page 193.

position-if *predicate sequence* &key (*start* 0) *end key from-end*
find-if *predicate sequence* &key (*start* 0) *end key from-end*

Find the first element of *sequence* (last element, if *from-end* is non-nil) which satisfies *predicate*. position-if returns the index in sequence of the element found; find-if returns the element itself. If no element is found, the value is nil for either function.

See section 9.4, page 193 for a description of the standard arguments *start*, *end* and *key*. If *start* or *end* is used to restrict operation to a portion of *sequence*, elements outside the portion are not tested, but the index returned is still the index in the entire sequence.

```
(position-if 'plusp '(-3 -2 -1 0 1 2 3))
  =>  4
(find-if 'plusp '(-3 -2 -1 0 1 2 3))
  =>  1
(position-if 'plusp '(-3 -2 -1 0 1 2 3) :start 5)
  =>  5
(position-if 'plusp '(-3 -2 -1 0 1 2 3) :from-end t)
  =>  6
(find-if 'plusp '(-3 -2 -1 0 1 2 3) :from-end t)
  =>  3
```

position-if-not *predicate sequence* &key (*start* 0) *end key from-end*
find-if-not *predicate sequence* &key (*start* 0) *end key from-end*

Like position-if and find-if but search for an element for which *predicate* returns nil.

position *item sequence sequence* &key *test test-not* (*start* 0) *end key from-end*
find *item sequence sequence* &key *test test-not* (*start* 0) *end key from-end*

Like position-if and find-if but search for an element which matches *item*, using *test* or *test-not* for comparison.

```
(position #\A "BabA" :test 'char-equal)  =>  1
(position #*/A "BabA" :test 'equalp)  =>  1
(position #\A "BabA" :test 'char=)  =>  3
(position #*/A "BabA" :test 'eq)  =>  3
```

find-position-in-list is equivalent to position with eq as the value of *test*.

count-if *predicate sequence* &key *start end key*

Tests each element of *sequence* with *predicate* and counts how many times *predicate* returns non-nil. This number is returned.

start, *end* and *key* are used in the standard way, as described in section 9.4, page 193. The *from-end* keyword argument is accepted without error, but it has no effect.

```
(count-if 'symbolp #(a b "foo" 3))  =>  2
```

count-if-not *predicate sequence* &key *start end key*

Like count-if but returns the number of elements for which *predicate* returns **nil**.

count *item sequence* &key *test test-not start end key*

Like count but returns the number of elements which match *item*. *test* or *test-not* is the function used for the comparison.

```
(count 4 '(1 2 3 4 5) :test '>)  =>  3
```

9.5 Comparison Functions

mismatch *sequence1 sequence2* &key (*test* 'eql) *test-not* (*start1* 0) *end1* (*start2* 0) *end2* *key* *from-end*

Compares successive elements of *sequence1* with successive elements of *sequence2*, returning nil if they all match, or else the index in *sequence1* of the first mismatch. If the sequences differ in length but match as far as they go, the value is the index in *sequence1* of the place where one sequence ran out. If *sequence1* is the one which ran out, this value equals the length of *sequence1*, so it isn't the index of an actual element, but it still describes the place where comparison stopped.

Elements are compared using the function *test*, which should accept two arguments. If it returns non-nil, the elements are considered to match. If you specify *test-not* instead of *test*, it is used similarly as a function, but the elements match if *test-not* returns nil.

If *key* is non-nil, it should be a function of one argument. It is applied to each element to get an object to pass to *test* or *test-not* in place of the element. Thus, if car is supplied as *key*, the cars of the elements are compared using *test* or *test-not*.

start1 and *end1* can be used to specify a portion of *sequence1* to use in the comparison, and *start2* and *end2* can be used to specify a portion of *sequence2*. The comparison uses the first element of each sequence portion, then the second element of each sequence portion, and so on. If the two specified portions differ in length, comparison stops where the first one runs out. In any case, the index returned by mismatch is still relative to the whole of *sequence1*.

If *from-end* is non-nil, the comparison proceeds conceptually from the end of each sequence or portion. The first comparison uses the last element of each sequence portion, the second comparison uses the next-to-the-last element of each sequence portion, and so on. When a mismatch is encountered, the value returned is *one greater than* the index of the first mismatch encountered in order of processing (closest to the ends of the sequences).

```
(mismatch "Foo" "Fox")  =>  2
(mismatch "Foo" "FOO" :test 'char-equal)  =>  nil
(mismatch "Foo" "FOO" :key 'char-upcase)  =>  nil
(mismatch '(a b) #(a b c))  =>  2
(mismatch "Win" "The Winner" :start2 4 :end2 7)  =>  nil
(mismatch "Foo" "Boo" :from-end t)  =>  1
```

search *for-sequence-1 in-sequence-2* &key *from-end test test-not key* (*start1* 0) *end1* (*start2* 0) *end2*

Searches *in-sequence-2* (or portion of it) for a subsequence that matches *for-sequence-1* (or portion of it) element by element, and returns the index in *in-sequence-2* of the beginning of the matching subsequence. If no matching subsequence is found, the value is nil. The comparison of each subsequence of *in-sequence-2* is done with mismatch, and the *test*, *test-not* and *key* arguments are used only to pass along to mismatch.

Normally, subsequences are considered starting with the beginning of the specified portion of *in-sequence-2* and proceeding toward the end. The value is therefore the index of the earliest subsequence that matches. If *from-end* is non-nil, the subsequences are tried in the reverse order, and the value identifies the latest subsequence that matches. In either case, the value identifies the beginning of the subsequence found.

```
(search '(#\A #\B) "cabbage" :test 'char-equal)  =>  1
```

9.6 Sorting and Merging

Several functions are provided for sorting vectors and lists. These functions use algorithms which always terminate no matter what sorting predicate is used, provided only that the predicate always terminates. The main sorting functions are not *stable*; that is, equal items may not stay in their original order. If you want a stable sort, use the stable versions. But if you don't care about stability, don't use them since stable algorithms are significantly slower.

After sorting, the argument (be it list or vector) has been rearranged internally so as to be completely ordered. In the case of a vector argument, this is accomplished by permuting the elements of the vector, while in the list case, the list is reordered by rplacd's in the same manner as nreverse. Thus if the argument should not be clobbered, the user must sort a copy of the argument, obtainable by fillarray or copylist, as appropriate. Furthermore, sort of a list is like delq in that it should not be used for effect; the result is conceptually the same as the argument but in fact is a different Lisp object.

Should the comparison predicate cause an error, such as a wrong type argument error, the state of the list or vector being sorted is undefined. However, if the error is corrected the sort proceeds correctly.

The sorting package is smart about compact lists; it sorts compact sublists as if they were vectors. See section 5.4, page 100 for an explanation of compact lists, and MIT A. I. Lab Memo 587 by Guy L. Steele Jr. for an explanation of the sorting algorithm.

sort *sequence predicate*

The first argument to sort is a vector or a list whose elements are to be sorted. The second is a predicate, which must be applicable to all the objects in the sequence. The predicate should take two arguments, and return non-nil if and only if the first argument is strictly less than the second (in some appropriate sense).

The sort function proceeds to reorder the elements of the sequence according to the predicate, and returns a modified sequence. Note that since sorting requires many comparisons, and thus many calls to the predicate, sorting is much faster if the predicate is a compiled function rather than interpreted.

Example: Sort a list alphabetically by the first symbol found at any level in each element.

```
(defun mostcar (x)
    (cond ((symbolp x) x)
          ((mostcar (car x)))))

(sort 'fooarray
      #'(lambda (x y)
          (string-lessp (mostcar x) (mostcar y))))
```

If **fooarray** contained these items before the sort:

```
(Tokens (The alien lurks tonight))
(Carpenters (Close to you))
((Rolling Stones) (Brown sugar))
((Beach Boys) (I get around))
(Beatles (I want to hold you up))
```

then after the sort **fooarray** would contain:

```
((Beach Boys) (I get around))
(Beatles (I want to hold you up))
(Carpenters (Close to you))
((Rolling Stones) (Brown sugar))
(Tokens (The alien lurks tonight))
```

When **sort** is given a list, it may change the order of the conses of the list (using **rplacd**), and so it cannot be used merely for side-effect; only the *returned value* of **sort** is the sorted list. The original list may have some of its elements missing when **sort** returns. If you need both the original list and the sorted list, you must copy the original and sort the copy (see **copylist**, page 94).

Sorting a vector just moves the elements of the vector into different places, and so sorting a vector for side-effect only is all right.

If the argument to **sort** is a vector with a fill pointer, note that, like most functions, **sort** considers the active length of the vector to be the length, and so only the active part of the vector is sorted (see **array-active-length**, page 174).

sortcar *sequence predicate*

sortcar is the same as **sort** except that the predicate is applied to the cars of the elements of *sequence*, instead of directly to the elements of *sequence*. Example:

```
(sortcar '((3 . dog) (1 . cat) (2 . bird)) #'<)
                      =>   ((1 . cat) (2 . bird) (3 . dog))
```

Remember that **sortcar**, when given a list, may change the order of the conses of the list (using **rplacd**), and so it cannot be used merely for side-effect; only the *returned value* of **sortcar** is the sorted list. The original list is destroyed by sorting.

stable-sort *sequence predicate*

stable-sort is like sort, but if two elements of *sequence* are equal, i.e. *predicate* returns nil when applied to them in either order, then they remain in their original order.

stable-sortcar *sequence predicate*

stable-sortcar is like sortcar, but if two elements of *sequence* are equal, i.e. *predicate* returns nil when applied to their cars in either order, then they remain in their original order.

sort-grouped-array *array group-size predicate*

sort-grouped-array considers its array argument to be composed of records of *group-size* elements each. These records are considered as units, and are sorted with respect to one another. The *predicate* is applied to the first element of each record; so the first elements act as the keys on which the records are sorted.

sort-grouped-array-group-key *array group-size predicate*

This is like sort-grouped-array except that the *predicate* is applied to four arguments: an array, an index into that array, a second array, and an index into the second array. *predicate* should consider each index as the subscript of the first element of a record in the corresponding array, and compare the two records. This is more general than sort-grouped-array since the function can get at all of the elements of the relevant records, instead of only the first element.

merge *result-type sequence1 sequence2 predicate* &key *key*

Returns a single sequence containing the elements of *sequence1* and *sequence2* interleaved in order according to *predicate*. The length of the result sequence is the sum of the lengths of *sequence1* and *sequence2*. *result-type* specifies the type of sequence to create, as in make-sequence.

The interleaving is done by taking the next element of *sequence1* unless the next element of *sequence2* is "less" than it according to *predicate*. Therefore, if each of the argument sequences is sorted, the result of merge is also sorted.

key, if non-nil, is applied to each element to get the object to pass to *predicate*, rather than the element itself. Thus, if *key* is car, the cars of the elements are compared rather than the entire elements.

```
(merge 'list '(1 2 5 6) '(3 5.0 5.1) '<)
  =>  (1 2 3 5 5.0 5.1 6)
```