5. Manipulating List Structure

This chapter discusses functions that manipulate conses, and higher-level structures made up of conses such as lists and trees. It also discusses hash tables and resources, which are related facilities.

A cons is a primitive Lisp data object that is extremely simple: it knows about two other objects, called its car and its cdr.

A list is recursively defined to be the symbol nil, or a cons whose cdr is a list. A typical list is a chain of conses: the cdr of each is the next cons in the chain, and the cdr of the last one is the symbol nil. The cars of each of these conses are called the elements of the list. A list has one element for each cons; the empty list, nil, has no elements at all. Here are the printed representations of some typical lists:

- (foo bar) ;This list has two elements.
- (a (b c d) e) ;This list has three elements.

Note that the second list has three elements: a, (b c d), and e. The symbols b, c, and d are not elements of the list itself. (They are elements of the list which is the second element of the original list.)

A dotted list is like a list except that the cdr of the last cons does not have to be nil. This name comes from the printed representation, which includes a "dot" character (period). Here is an example:

- (a b . c)

This dotted list is made of two conses. The car of the first cons is the symbol a, and the cdr of the first cons is the second cons. The car of the second cons is the symbol b, and the cdr of the second cons is the symbol c.

A tree is any data structure made up of conses whose cars and cdrs are other conses. The following are all printed representations of trees:

- (foo . bar)
- ((a . b) (c . d))
- ((a . b) (c d e f (g . 5) s) (7 . 4))

These definitions are not mutually exclusive. Consider a cons whose car is a and whose cdr is (b c d) e). Its printed representation is

- (a b (c d) e)

It can be thought of and treated as a cons, or as a list of four elements, or as a tree containing six conses. You can even think of it as a dotted list whose last cons just happens to have nil as a cdr. Thus, lists and dotted lists and trees are not fundamental data types; they are just ways of thinking about structures of conses.

A circular list is like a list except that the cdr of the last cons, instead of being nil, is the first cons of the list. This means that the conses are all hooked together in a ring, with the cdr of each cons being the next cons in the ring. These are legitimate Lisp objects, but dealing with them requires special techniques; straightforward tree-walking recursive functions often loop infinitely when given a circular list. The printer is an example of both aspects of the handling of circular lists: if *print-circle* is non-nil the printer uses special techniques to detect circular
structure and print it with a special encoding, but if *print-circle* is nil the printer does not check for circularity and loops infinitely unless *print-level* or *print-length* imposes a “time limit”. See page 514 for more information on *print-circle* and related matters.

The Lisp Machine internally uses a storage scheme called cdr-coding to represent conses. This scheme is intended to reduce the amount of storage used in lists. The use of cdr-coding is invisible to programs except in terms of storage efficiency; programs work the same way whether or not lists are cdr-coded or not. Several of the functions below mention how they deal with cdr-coding. You can completely ignore all this if you want. However, if you are writing a program that allocates a lot of conses and you are concerned with storage efficiency, you may want to learn about the cdr-coded representation and how to control it. The cdr-coding scheme is discussed in section 5.4, page 100.

5.1 Conses

car x

Returns the car of x.

Example:

(car '(a b c)) => a

cdr x

Returns the cdr of x.

Example:

(cdr '(a b c)) => (b c)

Officially car and cdr are only applicable to conses and locatives. However, as a matter of convenience, car and cdr of nil return nil. car or cdr of anything else is an error.

c...r x

All of the compositions of up to four car’s and cdr’s are defined as functions in their own right. The names of these functions begin with c and end with r, and in between is a sequence of a’s and d’s corresponding to the composition performed by the function.

Example:

(cddadr x) is the same as (cdr (cdr (car (cdr x))))

The error checking for these functions is exactly the same as for car and cdr above.

cons x y

cons is the primitive function to create a new cons, whose car is x and whose cdr is y.

Examples:

(cons 'a 'b) => (a . b)
(cons 'a (cons 'b (cons 'c nil))) => (a b c)
(cons 'a '(b c d)) => (a b c d)

ncons x

(ncons x) is the same as (cons x nil). The name of the function is from “nil-cons”.

PS:<L_MAN>FD-CON.TEXT.28 8-JUN-84
xcons x y
xcons ("exchanged cons") is like cons except that the order of the arguments is reversed.
Example:

(xcons 'a 'b) => (b . a)

cons-in-area x y area-number
Creates a cons in a specific area. (Areas are an advanced feature of storage management, explained in chapter 16; if you aren't interested in them, you can safely skip all this stuff). The first two arguments are the same as the two arguments to cons, and the third is the number of the area in which to create the cons.
Example:

(cons-in-area 'a 'b my-area) => (a . b)

ncons-in-area x area-number
(ncons-in-area x area-number) = (cons-in-area x nil area-number)

xcons-in-area x y area-number
(xcons-in-area x y area-number) = (cons-in-area y x area-number)

push item place
Adds an element item to the front of a list that is stored in place. A new cons is allocated whose car is item and whose cdr is the old contents of place. This cons is stored into place.

The form

(push (hairy-function x y z) variable)

replaces the commonly-used construct

(setq variable (cons (hairy-function x y z) variable))

and is intended to be more explicit and esthetic.

place can be any form that setf can store into. For example,

(push x (get y z))

=> (putprop y (cons x (get y z)) z)

The returned value of push is not defined.

pop place
Removes an element from the front of the list that is stored in place. It finds the cons in place, stores the cdr of the cons back into place, and returns the car of that cons. place can be any form that setf can store into.
Example:

(setq x '(a b c))
(pop x) => a
x => (b c)

The backquote reader macro facility is also generally useful for creating list structure, especially mostly-constant list structure, or forms constructed by plugging variables into a template. It is documented in the chapter on macros; see chapter 18, page 320.
**car-location** cons

`car-location` returns a locative pointer to the cell containing the car of *cons*.

Note: there is no `cdr-location` function; it is difficult because of the `cdr-coding` scheme (see section 5.4, page 100). Instead, the `cons` itself serves as a kind of locative to its `cdr` (see page 267).

The functions `rplaca` and `rplacd` are used to make alterations in already-existing list structure; that is, to change the cars andcdrs of existing conses. The structure is altered rather than copied. Exercise caution when using these functions, as strange side-effects can occur if they are used to modify portions of list structure which have become shared unbeknownst to the programmer. The `nconc`, `nreverse`, `nreconc`, `nbutlast` and `delq` functions and others, described below, have the same property, because they call `rplaca` or `rplacd`.

**rplaca** *x y*

Changes the car of *x* to *y* and returns (the modified) *x*. *x* must be a `cons` or a locative. *y* may be any Lisp object.

Example:

```
(setq g '(a b c))
(rplaca (cdr g) 'd) => (d c)
Now g => (a d c)
```

**rplacd** *x y*

Changes the cdr of *x* to *y* and returns (the modified) *x*. *x* must be a `cons` or a locative. *y* may be any Lisp object.

Example:

```
(setq x '(a b c))
(rplacd x 'd) => (a . d)
Now x => (a . d)
```

`(setf (car x) y)` and `(setf (car x) y)` are much like `rplaca` and `rplacd`, but they return *y* rather than *x*.

**5.2 Lists**

**list** &rest *args*

Constructs and returns a list of its arguments.

Example:

```
(list 3 4 '(a (car '((b . c)) (+ 6 -2))) => (3 4 a b 4)
```

`list` could have been defined by:

```
(defun list (&rest args)
  (let ((list (make-list (length args))))
    (do ((l list (cdr 1)))
        ((null a) list)
      ((rplaca 1 (car a)))))))
```
lis\textsuperscript{*} &rest args
lis\textsuperscript{*} is like list except that the last cons of the constructed list is dotted. It must be given
at least one argument.
Example:
\[(\text{list}\textsuperscript{*} 'a 'b 'c 'd) => (a b c . d)\]
This is like
\[(\text{cons} 'a (\text{cons} 'b (\text{cons} 'c 'd)))\]
More examples:
\[(\text{list}\textsuperscript{*} 'a 'b) => (a . b)\]
\[(\text{list}\textsuperscript{*} 'a) => a\]

length list-or-array
Returns the length of list-or-array. The length of a list is the number of elements in it;
the number of times you can cdr it before you get a non-cons.
Examples:
\[(\text{length} \texttt{nil}) => 0\]
\[(\text{length} '((a b c d)) => 4\]
\[(\text{length} '((a (b c) d)) => 3\]
\[(\text{length} "foobar") => 6\]
length could have been defined by:
\[(\text{defun} \text{length} \texttt{(x)}\]
\[(\text{if} (\text{arrayp} \texttt{x}) (\text{array-active-length} \texttt{x})\]
\[(\text{do} ((\text{n} 0 (1+ \text{n}))\]
\[(\text{y} \texttt{x} (\text{cdr} \texttt{y}))\]
\[(\text{null} \texttt{y}) \text{n}))\]

list-length list
Returns the length of list, or nil if list is circular. (The function length would loop
forever if given a circular list.)

first list
second list
third list
fourth list
fifth list
sixth list
seventh list
These functions take a list as an argument, and return the first, second, etc. element of
the list. first is identical to car, second is identical to cadr, and so on. The reason
these names are provided is that they make more sense when you are thinking of the
argument as a list rather than just as a cons.

rest list
rest1 list
rest2 list
rest3 list
rest4 list
rest\textsubscript{n} returns the rest of the elements of a list, starting with element \text{n} (counting the first
element as the zeroth). Thus rest or rest1 is identical to cdr, rest2 is identical to cddr, and so on. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

endp list
Returns t if list is nil, nil if list is a cons cell. Signals an error if list is not a list. This is the way Common Lisp recommends for terminating a loop which cdr's down a list. However, Lisp Machine system functions generally prefer to test for the end of the list with atom; it is regarded as a feature that these functions do something useful for dotted lists.

nth n list
(nth n list) returns the n'th element of list, where the zeroth element is the car of the list. If n is greater than the length of the list, nil is returned.
Examples:

(nth 1 '(foo bar gack)) => bar
(nth 3 '(foo bar gack)) => nil

Note: this is not the same as the InterLisp function called nth, which is similar to but not exactly the same as the Lisp Machine function nthcdr. Also, some people have used macros and functions called nth of their own in their Maclisp programs, which may not work the same way; be careful.

nth could have been defined by:

(defun nth (n list)
  (do ((i n (1- i))
       (l list (cdr l)))
       ((zerop i) (car l))))

nthcdr n list
(nthcdr n list) cdrs list n times, and returns the result.
Examples:

(nthcdr 0 '(a b c)) => (a b c)
(nthcdr 2 '(a b c)) => (c)

In other words, it returns the n'th cdr of the list. If n is greater than the length of the list, nil is returned.

This is similar to InterLisp's function nth, except that the InterLisp function is one-based instead of zero-based; see the InterLisp manual for details. nthcdr could have been defined by:

(defun nthcdr (n list)
  (do ((i 0 (1+ i))
       (l list (cdr l)))
       ((= i n) list)))
last list

last returns the last cons of list. If list is nil, it returns nil. Note that last is
unfortunately not analogous to first (first returns the first element of a list, but last
doesn’t return the last element of a list); this is a historical artifact.
Examples:

(setq x '(a b c d))
(last x) => (d)
(rplacd (last x) '(e f))
x => '(a b c d e f)

last could have been defined by:

(defun last (x)
  (cond ((atom x) x)
        ((atom (cdr x)) x)
        ((last (cdr x)))))

list-match-p object pattern

object is evaluated and matched against pattern; the value is t if it matches, nil otherwise.
pattern is made with backquotes (section 18.2.2, page 325); whereas normally a backquote
expression says how to construct list structure out of constant and variable parts, in this
context it says how to match list structure against constants and variables. Constant parts
of the backquote expression must match exactly: variables preceded by commas can match
anything but set the variable to what was matched. (Some of the variables may be set
even if there is no match.) If a variable appears more than once, it must match the same
thing (equal list structures) each time. ,ignore can be used to match anything and ignore it.

For example, '(x (y) ,z) is a pattern that matches a list of length at least two whose
first element is x and whose second element is a list of length one; if a list matches, the
caadr of the list is stored into the value of y and the cddr of the list is stored into z.

Variables set during the matching remain set after the list-match-p returns; in effect,
list-match-p expands into code which can setq the variables. If the match fails, some or
all of the variables may already have been set.

Example:

(list-match-p foo
  '((a ,x) ,ignore ,c))

is t if foo’s value is a list of two or more elements, the first of which is a list of two
elements; and in that case it sets x to (caadr foo) and c to (cddr foo). An equivalent
expression would be
(let ((tem foo))
  (and (consp tem)
       (consp (car tem))
       (eq (caar tem) 'a)
       (consp (cdar tem))
       (progn (setq x (cadar tem)) t)
       (null (cddar tem))
       (consp (cdr tem))
       (setq c (cddr tem))))

but list-match-p is faster.

list-match-p generates highly optimized code using special instructions.

list-in-area area-number &rest args

list-in-area is exactly the same as list except that it takes an extra argument, an area
number, and creates the list in that area.

list*-in-area area-number &rest args

list*-in-area is exactly the same as list* except that it takes an extra argument, an area
number, and creates the list in that area.

make-list length &key area initial-element

Creates and returns a list containing length elements. length should be a fixnum. area, if
specified, is the area in which to create the list (see chapter 16, page 296). If it is nil,
the area used is the value of working-storage-area.

initial-element is stored in each element of the new list.

make-list always creates a cdr-coded list (see section 5.4, page 100).
Examples:
  (make-list 3) => (nil nil nil)
  (make-list 4 :initial-element 7) => (7 7 7 7)

The keyword :initial-value may be used in place of :initial-element.

When make-list was originally implemented, it took exactly two arguments: the area and
the length. This obsolete form is still supported so that old programs can continue to
work, but the new keyword-argument form is preferred.

circular-list &rest args

Constructs a circular list whose elements are args, repeated infinitely. circular-list is the
same as list except that the list itself is used as the last cdr, instead of nil. circular-list
is especially useful with mapcar, as in the expression
  (mapcar (function +) foo (circular-list 5))
which adds each element of foo to 5.
circular-list could have been defined by:
(defun circular-list (&rest elements)
 (setq elements (copylist* elements))
 (replace (last elements) elements)
 elements)

copylist list &optional area

copy-list list &optional area
Returns a list which is equal to list, but not eq. copylist does not copy any elements of the list, only the conses of the list itself. The returned list is fully cdr-coded (see section 5.4, page 100) to minimize storage. If list is dotted, that is, if (cdr (last list)) is a non-nil atom, then the copy also has this property. You may optionally specify the area in which to create the new copy.

copylist* list &optional area
This is the same as copylist except that the last cons of the resulting list is never cdr-coded (see section 5.4, page 100). This makes for increased efficiency if you nconc something onto the list later.

copyalist list &optional area

copy-alist list &optional area

copyalist is for copying association lists (see section 5.5, page 102). The list is copied, as in copylist. In addition, each element of list which is a cons is replaced in the copy by a new cons with the same car and cdr. You may optionally specify the area in which to create the new copy.

append &rest lists

The arguments to append are lists. The result is a list which is the concatenation of the arguments. The arguments are not changed (cf. nconc).

Example:
(append '(a b c) '(d e f) nil '(g)) => (a b c d e f g)
append makes copies of the conses of all the lists it is given, except for the last one. So the new list shares the conses of the last argument to append, but all of the other conses are newly created. Only the lists are copied, not the elements of the lists.

A version of append which only accepts two arguments could have been defined by:
(defun append2 (x y)
 (cond ((null x) y)
 ((cons (car x) (append2 (cdr x) y)))))

The generalization to any number of arguments could then be made (relying on car of nil being nil):
(defun append (&rest args)
 (if (< (length args) 2) (car args)
  (append2 (car args)
   (apply (function append) (cdr args))))
These definitions do not express the full functionality of append; the real definition minimizes storage utilization by turning all the arguments that are copied into one cdr-coded list.

To copy a list, use copylist (see page 94); the old practice of using append to copy lists is unclear and obsolete.

`nconc` &rest `lists`

`nconc` takes lists as arguments. It returns a list which is the arguments concatenated together. The arguments are changed, rather than copied (cf. append, page 94). Example:

```lisp
(setq x '(a b c))
(setq y '(d e f))
(nconc x y) => (a b c d e f)
x => (a b c d e f)
```

Note that the value of `x` is now different, since its last cons has been rplacd'd to the value of `y`. If the `nconc` form were evaluated again, it would yield a piece of circular list structure, whose printed representation would be `(a b c d e f d e f ...)`, repeating forever.

`nconc` could have been defined by:

```lisp
(defun nconc (x y) ; for simplicity, this definition
  (cond ((null x) y) ; only works for 2 arguments.
    (t (rplacd (last x) y) ; hook y onto x
      x))) ; and return the modified x.
```

`revappend` `x` `y`

`revappend` `x` `y` is exactly the same as `(nconc (reverse x) y)` except that it is more efficient. Both `x` and `y` should be lists.

`revappend` could have been defined by:

```lisp
(defun revappend (x y)
  (cond ((null x) y) ; only works for 2 arguments.
    (t (revappend (cdr x) (cons (car x) y)))))
```

`nreverse1` `x` `y`

`nreverse1` `x` `y` is exactly the same as `(nconc (nreverse x) y)` except that it is more efficient. Both `x` and `y` should be lists.

`nreverse1` could have been defined by:

```lisp
(defun nreverse1 (x y)
  (cond ((null x) y)
    ((nreverse1 x y))
  ))
```

using the same `nreverse1` as above.
**butlast** list &optional (n 1)
This creates and returns a list with the same elements as list, excepting the last n elements.
Examples:

```
(butlast '(a b c d)) => (a b c)
(butlast '(a b c d) 3) => (a)
(butlast '(a b c d) 4) => nil
(butlast nil) => nil
```
The name is from the phrase “all elements but the last”.

**nbutlast** list &optional (n 1)
This is the destructive version of butlast; it changes the cdr of the last cons but n of the list to nil. The value is list, as modified. If list does not have more than n elements then it is not really changed and the value is nil.
Examples:

```
(setq foo '(a b c d))
(nbutlast foo) => (a b c)
foo => (a b c)
(nbutlast foo 2) => (a)
foo => (a)
(nbutlast foo) => nil
foo => (a)
```

**firstn** n list
Returns a list of length n, whose elements are the first n elements of list. If list is fewer than n elements long, the remaining elements of the returned list are nil.
Examples:

```
(firstn 2 '(a b c d)) => (a b)
(firstn 0 '(a b c d)) => nil
(firstn 6 '(a b c d)) => (a b c d nil nil)
```

**nleft** n list &optional tail
Returns a “tail” of list, i.e. one of the conses that makes up list, or nil. (nleft n list) returns the last n elements of list. If n is too large, nleft returns list.

(nleft n list tail) takes cdr of list enough times that taking n more cdrs would yield tail, and returns that. You can see that when tail is nil this is the same as the two-argument case. If tail is not eq to any tail of list, nleft returns nil.
Examples:

```
(setq x '(a b c d e f))
(nleft 2 x) => (e f)
(nleft 2 x (cddddr x)) => (c d e f)
```

**ldiff** list tail
list should be a list, and tail should be one of the conses that make up list. ldiff (meaning ‘list difference’) returns a new list, whose elements are those elements of list that appear before tail.
Examples:
\[
(setq x '(a b c d e))
(setq y (cdddr x)) => (d e)
(diff x y) => (a b c)
(diff x nil) => (a b c d e)
(diff x x) => nil
\]
but
\[
(diff '(a b c d) '(c d)) => (a b c d)
\]
since the tail was not eq to any part of the list.

\textbf{car-safe} object
\textbf{cdr-safe} object
\textbf{caar-safe} object
\textbf{cadr-safe} object
\textbf{cdar-safe} object
\textbf{cddr-safe} object
\textbf{caddr-safe} object
\textbf{cdaddr-safe} object
\textbf{nth-safe} \textit{n} object
\textbf{nthcdr-safe} \textit{n} object

Return the same things as the corresponding non-safe functions, except nil if the non-safe function would get an error. These functions are about as fast as the non-safe functions. The same effect could be had by handling the \texttt{sys:wrong-type-argument} error, but that would be slower. Examples:
\[
(car-safe '(a . b)) => a
(car-safe nil) => nil
(car-safe 'a) => nil
(car-safe "foo") => nil
(cadr-safe '(a . b)) => nil
(cadr-safe 3) => nil
\]

5.3 Cons Cells as Trees

\textbf{copytree} \textit{tree} &optional \textit{area}
\textbf{copy-tree} \textit{tree} &optional \textit{area}

\texttt{copytree} copies all the conses of a tree and makes a new maximally cdr-coded tree with the same fringe. If \textit{area} is specified, the new tree is constructed in that area.

\textbf{tree-equal} \textit{x} \textit{y} &key \textit{test} \textit{test-not}

Compares two trees recursively to all levels. Atoms must match under the function \textit{test} (which defaults to eq). Conses must match recursively in both the car and the cdr.

If \textit{test-not} is specified instead of \textit{test}, two atoms match if \textit{test-not} returns \texttt{nil}. 
**subst** new old tree

(subst new old tree) substitutes new for all occurrences of old in tree, and returns the modified copy of tree. The original tree is unchanged, as subst recursively copies all of tree replacing elements equal to old as it goes.

Example:

(subst 'Tempest 'Hurricane
 ' (Shakespeare wrote (The Hurricane)))
 => (Shakespeare wrote (The Tempest))

subst could have been defined by:

(defun subst (new old tree)
 (cond ((equal tree old) new) ; if item equal to old, replace.
   ((atom tree) tree) ; if no substructure, return arg.
   ((cons (subst new old (car tree)) ; otherwise recurse.
       (subst new old (cdr tree)))))

Note that this function is not destructive; that is, it does not change the car or cdr of any already-existing list structure.

To copy a tree, use copytree (see page 97); the old practice of using subst to copy trees is unclear and obsolete.

**c1i:subst** new old tree &key test test-not key

The Common Lisp version of subst replaces with new every atom or subtree in tree which matches old, returning a new tree. List structure is copied as necessary to avoid clobbering parts of tree. This differs from the traditional subst function, which always copies the entire tree.

test or test-not is used to do the matching. If test is specified, a match happens when test returns non-nil; otherwise, if test-not is specified, a match happens when it returns nil. If neither is specified, then eql is used for test.

The first argument to the test or test-not function is always old. The second argument is normally a leaf or subtree of tree. However, if key is non-nil, then it is called with the subtree as argument, and the result of this becomes the second argument to the test or test-not function.

Because (subst nil nil tree) is a widely used idiom for copying a tree, even though it is obsolete, there is no practical possibility of installing this function as the standard subst for a long time.

**nsubst** new old tree &key test test-not key

nsubst is a destructive version of subst. The list structure of tree is altered by replacing each occurrence of old with new. No new list structure is created. The keyword arguments are as in c1i:subst.

A simplified version of nsubst, handling only the three required arguments, could be defined as
(defun nsubst (new old tree)
  (cond ((eq1 tree old) new) ; If item matches old, replace.
    ((atom tree) tree) ; If no substructure, return arg.
    (t ; Otherwise, recurse.
      (rplaca tree (nsubst new old (car tree)))
      (rplacd tree (nsubst new old (cdr tree)))
      tree)))

subst-if new predicate tree &key key
Replaces with new every atom or subtree in tree which satisfies predicate. List structure is
copied as necessary so that the original tree is not modified. key, if non-nil, is a function
applied to each tree node to get the object to match against. If key is nil or omitted, the
tree node itself is used.

subst-if-not new predicate tree &key key
Similar, but replaces tree nodes which do not satisfy predicate.

nsubst-if new predicate tree &key key
nsubst-if-not new predicate tree &key key
Like subst-if and subst-if-not except that they destructively modify tree itself and return
it, creating no new list structure.

sublis alist tree &key test test-not key
Performs multiple parallel replacements on tree, returning a new tree. tree itself is not
modified because list structure is copied as necessary. If no substitutions are made, the
result is tree. alists is an association list (see section 5.5, page 102). Each element of alists
specifies one replacement; the car is what to look for, and the cdr is what to replace it
with.

test, test-not and key control how matching is done between nodes of the tree (cons cells
or atoms) and objects to be replaced. See cli:subst, above, for the details of how they
work. The first argument to test or test-not is the car of an element of alists.
Example:

(sublis '((x . 100) (z . zprime))
  '(plus x (minus g z x p) 4))
=> (plus 100 (minus g zprime 100 p) 4)

A simplified sublis could be defined by:
(defun sublis (alist tree)
  (let ((tem (assq tree alist)))
    (cond (tem (cdr tem))
      ((atom tree) tree)
      (t
        (let ((car (sublis alist (car tree))))
          (cdr (sublis alist (cdr tree))))
          (if (and (eq (car tree) car) (eq (cdr tree) cdr))
           tree
           (cons car cdr))))))

nsublis alist tree &key test test-not key
nsublis is like sublis but changes the original tree instead of allocating new structure.

A simplified nsublis could be defined by:
(defn nsublis (alist tree)
  (let (((tem (assq tree alist)))
      (cond (tem (cdr tem))
            (atom tree) tree)
      (t (replaca tree (nsublis alist (car tree)))
        (replacd tree (nsublis alist (cdr tree)))
        tree))))

5.4 Cdr-Coding

This section explains the internal data format used to store conses inside the Lisp Machine. Casual users don’t have to worry about this; you can skip this section if you want. It is only important to read this section if you require extra storage efficiency in your program.

The usual and obvious internal representation of conses in any implementation of Lisp is as a pair of pointers, contiguous in memory. If we call the amount of storage that it takes to store a Lisp pointer a ‘word’, then conses normally occupy two words. One word (say it’s the first) holds the car, and the other word (say it’s the second) holds the cdr. To get the car or cdr of a list, you just reference this memory location, and to change the car or cdr, you just store into this memory location.

Very often, conses are used to store lists. If the above representation is used, a list of $n$ elements requires two times $n$ words of memory: $n$ to hold the pointers to the elements of the list, and $n$ to point to the next cons or to nil. To optimize this particular case of using conses, the Lisp Machine uses a storage representation called cdr-coding to store lists. The basic goal is to allow a list of $n$ elements to be stored in only $n$ locations, while allowing conses that are not parts of lists to be stored in the usual way.

The way it works is that there is an extra two-bit field in every word of memory, called the cdr-code field. There are three meaningful values that this field can have, which are called cdr-normal, cdr-next, and cdr-nil. The regular, non-compact way to store a cons is by two contiguous words, the first of which holds the car and the second of which holds the cdr. In this case, the cdr-code of the first word is cdr-normal. (The cdr-code of the second word doesn’t matter; as we will see, it is never looked at.) The cons is represented by a pointer to the first of the two words. When a list of $n$ elements is stored in the most compact way, pointers to the $n$ elements occupy $n$ contiguous memory locations. The cdr-codes of all these locations are cdr-next, except the last location whose cdr-code is cdr-nil. The list is represented as a pointer to the first of the $n$ words.

Now, how are the basic operations on conses defined to work based on this data structure? Finding the car is easy: you just read the contents of the location addressed by the pointer. Finding the cdr is more complex. First you must read the contents of the location addressed by the pointer, and inspect the cdr-code you find there. If the code is cdr-normal, then you add one to the pointer, read the location it addresses, and return the contents of that location; that is,
you read the second of the two words. If the code is cdr-next, you add one to the pointer, and simply return that pointer without doing any more reading; that is, you return a pointer to the next word in the n-word block. If the code is cdr-nil, you simply return nil.

If you examine these rules, you will find that they work fine even if you mix the two kinds of storage representation within the same list.

How about changing the structure? Like car, rplaca is very easy: you just store into the location addressed by the pointer. To do rplacd you must read the location addressed by the pointer and examine the cdr-code. If the code is cdr-normal, you just store into the location one greater than that addressed by the pointer; that is, you store into the second word of the two words. But if the cdr-code is cdr-next or cdr-nil, there is a problem: there is no memory cell that is storing the cdr of the cons. That is the cell that has been optimized out; it just doesn't exist.

This problem is dealt with by the use of invisible pointers. An invisible pointer is a special kind of pointer, recognized by its data type (Lisp Machine pointers include a data type field as well as an address field). The way they work is that when the Lisp Machine reads a word from memory, if that word is an invisible pointer then it proceeds to read the word pointed to by the invisible pointer and use that word instead of the invisible pointer itself. Similarly, when it writes to a location, it first reads the location, and if it contains an invisible pointer then it writes to the location addressed by the invisible pointer instead. (This is a somewhat simplified explanation; actually there are several kinds of invisible pointer that are interpreted in different ways at different times, used for things other than the cdr-coding scheme.)

Here's how to do rplacd when the cdr-code is cdr-next or cdr-nil. Call the location addressed by the first argument to rplacd l. First, you allocate two contiguous words in the same area that l points to. Then you store the old contents of l (the car of the cons) and the second argument to rplacd (the new cdr of the cons) into these two words. You set the cdr-code of the first of the two words to cdr-normal. Then you write an invisible pointer, pointing at the first of the two words, into location l. (It doesn't matter what the cdr-code of this word is, since the invisible pointer data type is checked first, as we will see.)

Now, whenever any operation is done to the cons (car, cdr, rplaca, or rplacd), the initial reading of the word pointed to by the Lisp pointer that represents the cons finds an invisible pointer in the addressed cell. When the invisible pointer is seen, the address it contains is used in place of the original address. So the newly-allocated two-word cons is used for any operation done on the original object.

Why is any of this important to users? In fact, it is all invisible to you; everything works the same way whether or not compact representation is used, from the point of view of the semantics of the language. That is, the only difference that any of this makes is a difference in efficiency. The compact representation is more efficient in most cases. However, if the conses are going to get rplacd'ed, then invisible pointers will be created, extra memory will be allocated, and the compact representation will degrade storage efficiency rather than improve it. Also, accesses that go through invisible pointers are somewhat slower, since more memory references are needed. So if you care a lot about storage efficiency, you should be careful about which lists get stored in which representations.
You should try to use the normal representation for those data structures that will be subject to rplacd operations, including nconc and nreverse, and the compact representation for other structures. The functions cons, xcons, ncons, and their area variants make conses in the normal representation. The functions list, list*, list-in-area, make-list, and append use the compact representation. The other list-creating functions, including read, currently make normal lists, although this might get changed. Some functions, such as sort, take special care to operate efficiently on compact lists (sort effectively treats them as arrays). nreverse is rather slow on compact lists; currently, since it simple-mindedly uses rplacd, but this may be changed.

(copylist x) is a suitable way to copy a list, converting it into compact form (see page 94).

5.5 Tables

Zetalisp includes functions which simplify the maintenance of tabular data structures of several varieties. The simplest is a plain list of items. There are functions to add (cons), remove (delete, delq, del), del-if, del-if-not, remove, remq, rem, rem-if, rem-if-not), and search for (member, memq, mem) items in a list.

Association lists are very commonly used. An association list is a list of conses. The car of each cons is a “key” and the cdr is a “datum”, or a list of associated data. The functions assoc, assq, ass, memass, and rassoc may be used to retrieve the data, given the key. For example,

\[
((\text{tweety . bird}) (\text{svlvester . cat}))
\]

is an association list with two elements. Given a symbol representing the name of an animal, it can retrieve what kind of animal this is.

Structured records can be stored as association lists or as stereotyped cons-structures where each element of the structure has a certain car-cdr path associated with it. However, these are better implemented using structure macros (see chapter 20, page 372) or as flavors (chapter 21, page 401).

Simple list-structure is very convenient, but may not be efficient enough for large data bases because it takes a long time to search a long list. Zetalisp includes hash table facilities for more efficient but more complex tables (see section 5.11, page 116), and a hashing function (sxhash) to aid users in constructing their own facilities.

5.6 Lists as Tables

memq item list

Returns nil if item is not one of the elements of list. Otherwise, it returns the sublist of list beginning with the first occurrence of item; that is, it returns the first cons of the list whose car is item. The comparison is made by eq. Because memq returns nil if it doesn't find anything, and something non-nil if it finds something, it is often used as a predicate.

Examples:

\[
\begin{align*}
  \text{(memq 'a '(1 2 3 4))} & \Rightarrow \text{nil} \\
  \text{(memq 'a '(g (x a y) c a d e a f))} & \Rightarrow (a d e a f)
\end{align*}
\]

Note that the value returned by memq is eq to the portion of the list beginning with a.
Thus rplaca on the result of memq may be used, if you first check to make sure memq did not return nil.

Example:

\[
\begin{align*}
\text{(let ((sublist (memq x z))) \\
\quad (if (not (null sublist)) \\
\qquad (rplaca sublist y)))}
\end{align*}
\]

; Search for x in the list z.
; If it is found,
; Replace it with y.

memq could have been defined by:

\[
\begin{align*}
\text{(defun memq (item list) \\
\quad (cond ((null list) nil) \\
\qquad ((eq item (car list)) list) \\
\qquad (t (memq item (cdr list)))))}
\end{align*}
\]

memq is hand-coded in microcode and therefore especially fast. It is equivalent to cli:member with eq specified as the test argument.

\[\text{member item list}\]

can be like memq, except equal is used for the comparison, instead of eq. Note that the member function of Common Lisp, which is cli:member, is similar but thoroughly incompatible (see below).

member could have been defined by:

\[
\begin{align*}
\text{(defun member (item list) \\
\quad (cond ((null list) nil) \\
\qquad ((equal item (car list)) list) \\
\qquad (t (member item (cdr list))))}
\end{align*}
\]

\[\text{cli:member item list &key test test-not key}\]

The Common Lisp member function. It is like memq or member except that there is more generality in how elements of list are matched against item—and the default is incompatible.

\[\text{test, test-not and key are used in matching the elements, just as described under cli:subst (see page 98). If neither test nor test-not is specified, the default is to compare with eql, whereas member compares with equal.}\]

Usually test is a commutative predicate such as eq, equal, =, char-equal or string-equal. It can also be a non-commutative predicate. The predicate is called with item as its first argument and the element of list as its second argument. Example:

\[
\text{(cli:member 4 '(1.5 2.5 2 3.5 4.5 8) :test '<) => (4.5 8)}
\]

\[\text{member-if predicate list &key key}\]

Searches the elements of list for one which satisfies predicate. If one is found, the value is the tail of list whose car is that element. Otherwise the value is nil.

If key is non-nil, then predicate is applied to (funcall key element) rather than to the element itself.
member-if-not predicate list &key key
    Searches for an element which does not satisfy predicate. Otherwise like member-if.

mem predicate item list
    Is equivalent to
    (cli:member item list :test predicate)

The function mem antedates cli:member.

find-position-in-list item list
    Searches list for an element which is eq to item, like memq. However, it returns the
    numeric index in the list at which it found the first occurrence of item, or nil if it did not
    find it at all. This function is sort of the complement of nth (see page 91); like nth, it
    is zero-based.
    Examples:
    (find-position-in-list 'a '(a b c)) => 0
    (find-position-in-list 'C '(a b c)) => 2
    (find-position-in-list 'e '(a b c)) => nil

See also the generic sequence function position (page 198).

find-position-in-list-equal item list
    Is like find-position-in-list, except that the comparison is done with equal instead of
    eq.

tailp sublist list
    Returns t if sublist is a sublist of list (i.e. one of the conses that makes up list).
    Otherwise returns nil. Another way to look at this is that tailp returns t if (nthcdr n list)
    is sublist, for some value of n. tailp could have been defined by:
    (defun tailp (sublist list)
      (do list list (cdr list) (null list)
          (if (eq sublist list)
              (return t)))))

delq item list &optional n
    (delq item list) returns the list with all occurrences of item removed. eq is used for the
    comparison. The argument list is actually modified (rplacd'ed) when instances of item are
    spliced out. delq should be used for value, not for effect. That is, use
    (setq a (delq 'b a))
    rather than
    (delq 'b a)
    These two are not equivalent when the first element of the value of a is b.

(delq item list n) is like (delq item list) except only the first n instances of item are
    deleted. n is allowed to be zero. If n is greater than or equal to the number of
    occurrences of item in the list, all occurrences of item in the list are deleted.
Example:
    (delq 'a '(b a c (a b) d a e)) => (b c (a b) d e)
delq could have been defined by:

```lisp
(defun delq (item list &optional (n -1))
  (cond ((or (atom list) (zerop n)) list)
        ((eq item (car list))
         (delq item (cdr list) (1- n)))
        (t (replacd list (delq item (cdr list) n))))
```

If the third argument (n) is not supplied, it defaults to -1 which is effectively infinity since it can be decremented any number of times without reaching zero.

**delete** item list &optional n

Delete is the same as delq except that equal is used for the comparison instead of eq.

Common Lisp programs have a different, incompatible function called delete; see page 195. This function may be useful in non-Common-Lisp programs as well, where it can be referred to as cli:delete.

**del** predicate item list &optional n

Del is the same as delq except that it takes an extra argument which should be a predicate of two arguments, which is used for the comparison instead of eq. (del 'eq a b) is the same as (delq a b). See also mem, page 104.

Use of del is equivalent to

```lisp
(cli:delete item list :test predicate)
```

**remq** item list &optional n

Remq is similar to delq, except that the list is not altered; rather, a new list is returned. Examples:

```lisp
(setq x '(a b c d e f))
(remq 'b x) => (a c d e f)
x => (a b c d e f)
(remq 'b '(a b c b a b) 2) => (a c a b)
```

**remove** item list &optional n

Remove is the same as remq except that equal is used for the comparison instead of eq. Common Lisp programs have a different, incompatible function called remove; see page 195. This function may be useful in non-Common-Lisp programs as well, where it can be referred to as cli:remove.

**rem** predicate item list &optional n

Rem is the same as remq except that it takes an extra argument which should be a predicate of two arguments, which is used for the comparison instead of eq. (rem 'eq a b) is the same as (remq a b). See also mem, page 104.

The function rem in Common Lisp programs is actually cli:rem, a remainder function. See page 144.
**subset predicate list**

**rem-if-not predicate list**

*predicate* should be a function of one argument. A new list is made by applying *predicate* to all of the elements of *list* and removing the ones for which the *predicate* returns nil. One of this function’s names (*rem-if-not*) means “remove if this condition is not true”; i.e. it keeps the elements for which *predicate* is true. The other name (subset) refers to the function’s action if *list* is considered to represent a mathematical set.

Example:

```
(subset #'-minusp '(1 2 -4 2 -3)) => (-4 -3)
```

**subset-not predicate list**

**rem-if predicate list**

*predicate* should be a function of one argument. A new list is made by applying *predicate* to all of the elements of *list* and removing the ones for which the *predicate* returns non-nil. One of this function’s names (*rem-if*) means “remove if this condition is true”. The other name (subset-not) refers to the function’s action if *list* is considered to represent a mathematical set.

**del-if predicate list**

*del-if* is just like *rem-if* except that it modifies *list* rather than creating a new list.

**del-if-not predicate list**

*del-if-not* is just like *rem-if-not* except that it modifies *list* rather than creating a new list.

See also the generic sequence functions delete-if, delete-if-not, remove-if and remove-if-not (page 194).

**every list predicate &optional step-function**

Returns t if *predicate* returns non-nil when applied to every element of *list*, or nil if *predicate* returns nil for some element. If *step-function* is present, it replaces cdr as the function used to get to the next element of the list; cddr is a typical function to use here.

In Common Lisp programs, the name *every* refers to a different, incompatible function which serves a similar purpose. It is documented in the manual under the name cli:every. See page 192.

**some list predicate &optional step-function**

Returns a tail of *list* such that the car of the tail is the first element that the *predicate* returns non-nil when applied to, or nil if *predicate* returns nil for every element. If *step-function* is present, it replaces cdr as the function used to get to the next element of the list; cddr is a typical function to use here.

In Common Lisp programs, the name *some* refers to a different, incompatible function which serves a similar purpose. It is documented in the manual under the name cli:some. See page 191.
5.7 Lists as Sets

A list can be used to represent an unordered set of objects, simply by using it in a way that ignores the order of the elements. Membership in the set can be tested with `memq` or `member`, and some other functions in the previous section also make sense on lists representing sets. This section describes several functions specifically intended for lists that represent sets.

It is often desirable to avoid adding duplicate elements in the list. The set functions attempt to introduce no duplications, but do not attempt to eliminate duplications present in their arguments. If you need to make absolutely certain that a list contains no duplicates, use `remove-duplicates` or `delete-duplicates` (see page 196).

```
subsetp list1 list2 &key test test-not key
  t if every element of list1 matches some element of list2.

The keyword arguments control how matching is done. Either test or test-not should be a function of two arguments. Normally it is called with an element of list1 as the first argument and an element of list2 as the second argument. If test is specified, a match happens when test returns non-nil; otherwise, if test-not is specified, a match happens when it returns nil. If neither is specified, then eql is used for test.

If key is non-nil, it should be a function of one argument. key is applied to each list element to get a key to be passed to test or test-not instead of the element.
```

```
adjoin item list &key test test-not key
Returns a list like list but with item as an additional element if no existing element matches item. It is done like this:
  (if (cli:member (if key (funcall key item) item)
               list other-args...)
      list
      (cons item list))

The keyword arguments work as in subsetp.
```

```
pushnew item list-place &key test test-not key
Macro
Pushes item onto list-place unless item matches an existing element of the value stored in that place. Equivalent to
  (setf list-place
       (adjoin item list-place keyword-args...))
```

```
union list &rest more-lists
Returns a list representing the set which is the union of the sets represented by the arguments. Anything which is an element of at least one of the arguments is also an element of the result.

Each element of each list is compared, using eq, with all elements of the other lists, to make sure that no duplications are introduced into the result. As long as no individual argument list contains duplications, the result does not either.
```
It is best to use union with only two arguments so that your code will not be sensitive to the difference between the traditional version of union and the Common Lisp version cli:union, below.

**intersection** list &rest more-lists
If lists are regarded as sets of their elements, intersection returns a list which is the intersection of the lists which are supplied as arguments. If list contains no duplicate elements, neither does the value returned by intersection. Elements are compared using eq.

It is best to use intersection with only two arguments so that your code will not be sensitive to the difference between the traditional version of intersection and the Common Lisp version cli:intersection, below.

**union** list &rest more-lists
If lists are regarded as sets of their elements, union modifies list to become the union of the lists which are supplied as arguments. This is done by adding on, at the end, any elements of the other lists that were not already in list. If none of the arguments contains any duplicate elements, neither does the value returned by union. Elements are compared using eq.

It is not safe to assume that list has been modified properly in place, as this will not be so if list is nil. Rather, you must store the value returned by union in place of list.

It is best to use union with only two arguments so that your code will not be sensitive to the difference between the traditional version of union and the Common Lisp version cli:union, below.

**nintersection** list &rest more-lists
Like intersection but produces the value by deleting elements from list until the desired result is reached, and then returning list as modified.

It is not safe to assume that list has been modified properly in place, as this will not be so if the first element was deleted. Rather, you must store the value returned by nintersection in place of list.

It is best to use nintersection with only two arguments so that your code will not be sensitive to the difference between the traditional version of nintersection and the Common Lisp version cli:nintersection, below.

**cli:union** list1 list2 &key test test-not key
**cli:intersection** list1 list2 &key test test-not key
**cli:union** list1 list2 &key test test-not key
**cli:nintersection** list1 list2 &key test test-not key
The Common Lisp versions of the above functions, which accept only two sets to operate on, but permit additional arguments to control how elements are matched. These keyword arguments work the same as in subseq.
set-difference list1 list2 &key test test-not key
Returns a list which has all the elements of list1 which do not match any element of list2. The keyword arguments control comparison of elements just as in subsetp.

The result contains no duplicate elements as long as list1 contains none.

set-exclusive-or list1 list2 &key test test-not key
Returns a list which has all the elements of list1 which do not match any element of list2, and also all the elements of list2 which do not match any element of list1. The keyword arguments control comparison of elements just as in subsetp.

The result contains no duplicate elements as long as neither list1 nor list2 contains any.

nset-difference list1 list2 &key test test-not key
Like set-difference but destructively modifies list1 to produce the value. See the caveat in nintersection, above.

nset-exclusive-or list1 list2 &key test test-not key
Like set-exclusive-or but may destructively modify both list1 and list2 to produce the value. See the caveat in nintersection, above.

5.8 Association Lists

In all the alist-searching functions, alist elements which are nil are ignored; they do not count as equivalent to (nil . nil). Elements which are not lists cause errors.

pairlis cars cdrs &optional tail
pairlis takes two lists and makes an association list which associates elements of the first list with corresponding elements of the second list.

Example:

(pairlis '(beef clams kitty) '(roast fried yu-shiang))
=> ((beef . roast) (clams . fried) (kitty . yu-shiang))

If tail is non-nil, it should be another alist. The new alist continues with tail following the newly constructed mappings.

pairlis is defined as:

(defun pairlis (cars cdrs &optional tail)
  (nconc (mapcar 'cons cars cdrs) tail))

acons acar acadr tail
Returns (cons (cons acar acadr) tail). This adds one additional mapping from acar to acadr onto the alist tail.
assq item alist

(assq item alist) looks up item in the association list (list of conses) alist. The value is the first cons whose car is eq to x, or nil if there is none such.

Examples:

(assq 'r '((a . b) (c . d) (r . x) (s . y) (r . z)))
=> (r . x)

(assq 'fooo '((foo . bar) (zoo . goo))) => nil

(assq 'b '((a b c) (b c d) (x y z))) => (b c d)

It is okay to rplacd the result of assq as long as it is not nil, if your intention is to "update" the "table" that was assq's second argument.

Example:

(setq values '((x . 100) (y . 200) (z . 50)))
(assq 'y values) => (y . 200)
(rplacd (assq 'y values) 201)
(assq 'y values) => (y . 201)

A common trick is to say (cdr (assq x y)). Since the cdr of nil is guaranteed to be nil, this yields nil if no pair is found (or if a pair is found whose cdr is nil.)

assq could have been defined by:

(defun assq (item list)
  (cond ((null list) nil)
        ((eq item (caar list)) (car list))
        ((assq item (cdr list)))))

assoc item alist

assoc is like assq except that the comparison uses equal instead of eq.

Example:

(assoc '(a b) '((x . y) ((a b) . 7) ((c . d) . e)))
=> ((a b) . 7)

assoc could have been defined by:

(defun assoc (item list)
  (cond ((null list) nil)
        ((equal item (caar list)) (car list))
        ((assoc item (cdr list)))))

cli:assoc item alist &key test test-not

The Common Lisp version of assoc, this function returns the first element of alist which is a cons whose car matches item, or nil if there is no such element.

Test and test-not are used in comparing elements, as in cli:subst (page 98), but note that there is no key argument in cli:assoc.

cli:assoc is incompatible with the traditional function assoc in that, like most Common Lisp functions, it uses eql by default rather than equal for the comparison.
ass predicate item alist
ass is the same as assq except that it takes an extra argument which should be a
predicate of two arguments, which is used for the comparison instead of eq. (ass 'eq a
b) is the same as (assq a b). See also mem, page 104.

This function is part of The mem, rem, del series, whose names were chose partly
because they created a situation in which this function simply had to be called ass. It's
too bad that cli:assoc is so general and subsumes ass, which is equivalent to

(cui:assoc item :test predicate)

assoc-if predicate alist
Returns the first element of alist which is a cons whose car satisfies predicate, or nil if
there is no such element.

assoc-if-not predicate alist
Returns the first element of alist which is a cons whose car does not satisfy predicate, or
nil if there is no such element.

memass predicate item alist
memass searches alist just like ass, but returns the portion of the list beginning with the
pair containing item, rather than the pair itself. (car (memass x y z)) = (ass x y z).
See also mem, page 104.

rassq item alist
rassoc item alist
rass predicate item alist
cli:rassoc item alist &key test test-not
rassoc-if predicate alist
rassoc-if-not predicate alist
The reverse-association functions are like assq, assoc, etc. but match or test the cdrs of
the alist elements instead of the car's. For example, rassq could have been defined by:

(defun rassq (item in-list)
  (do | in-list (cdr |) (null |)
      (and (eq item (cdar |))
        (return (car |)))))

sassq item alist fcn
(sassq item alist fcn) is like (assq item alist) except that if item is not found in alist,
instead of returning nil, sassq calls the function fcn with no arguments. sassq could
have been defined by:

(defun sassq (item alist fcn)
  (or (assq item alist)
      (apply fcn nil)))

sassq and sassoc (see below) are of limited use. These are primarily leftovers from Lisp
1.5.
**sassoc item alist fcn**

(sassoc item alist fcn) is like (assoc item alist) except that if item is not found in alist, instead of returning nil, sassoc calls the function fcn with no arguments. sassoc could have been defined by:

```lisp
(defun sassoc (item alist fcn)
  (or (assoc item alist)
      (apply fcn nil)))
```

### 5.9 Stack Lists

When you are creating a list that will not be needed any more once the function that creates it is finished, it is possible to create the list on the stack instead of by consing it. This avoids any permanent storage allocation, as the space is reclaimed as part of exiting the function. By the same token, it is a little risky; if any pointers to the list remain after the function exits, they will become meaningless.

These lists are called *temporary lists* or *stack lists*. You can create them explicitly using the special forms with-stack-list and with-stack-list*. &rest arguments also sometimes create stack lists.

If a stack list, or a list which might be a stack list, is to be returned or made part of permanent list-structure, it must first be copied (see copylist, page 94). The system cannot detect the error of omitting to copy a stack list; you will simply find that you have a value that seems to change behind your back.

**with-stack-list** (variable element...) body...  
**with-stack-list** (variable element... tail) body...  

*Special form*

These special forms create stack lists that live inside the stack frame of the function that they are used in. You should assume that the stack lists are only valid until the special form is exited.

```
(with-stack-list (foo x y)
  (mumblify foo))
```

is equivalent to

```
(let ((foo (list x y)))
  (mumblify foo))
```

except for the fact that foo's value in the first example is a stack list.

The list created by with-stack-list* looks like the one created by list*; tail's value becomes the ultimate cdr rather than an element of the list.

Here is a practical example. condition-resume (see page 723) might have been defined as follows:

```
(defun condition-resume (handler &body body)
  `(with-stack-list* (eh:condition-resume-handlers
                      ,handler eh:condition-resume-handlers)
    ,body))
```

---

PS:<1..MAN>FD-CON.TEXT.28  8-JUN-84
It is an error to do `rplacd` on a stack list (except for the tail of one made using `with-stack-list`). `rplaca` works normally.

**sys:rplacd-wrong-representation-type (error)**

This is signaled if you `rplacd` a stack list (or a list overlayed with an array or other structure).

### 5.10 Property Lists

From time immemorial, Lisp has had a kind of tabular data structure called a *property list* (plist for short). A property list contains zero or more entries; each entry associates from a keyword symbol (called the *property name*, or sometimes the *indicator*) to a Lisp object (called the *value* or, sometimes, the *property*). There are no duplications among the property names; a property-list can have only one property at a time with a given name.

This is very similar to an association list. The important difference is that a property list is an object with a unique identity; the operations for adding and removing property-list entries are side-effecting operations which alter the property-list rather than making a new one. An association list with no entries would be the empty list () i.e. the symbol nil. There is only one empty list, so all empty association lists are the same object. Each empty property-list is a separate and distinct object.

The implementation of a property list is a memory cell containing a list with an even number (possibly zero) of elements. Each pair of elements constitutes a *property*; the first of the pair is the name and the second is the value. (It would have been possible to use an alist to hold the pairs; this format was chosen when Lisp was young.) The memory cell is there to give the property list a unique identity and to provide for side-effecting operations.

The term 'property list' is sometimes incorrectly used to refer to the list of entries inside the property list, rather than the property list itself. This is regrettable and confusing.

How do we deal with "memory cells" in Lisp? That is, what kind of Lisp object is a property list? Rather than being a distinct primitive data type, a property list can exist in one of three forms:

1. Any cons can be used as a property list. The cdr of the cons holds the list of entries (property names and values). Using the cons as a property list does not use the car of the cons; you can use that for anything else.

2. The system associates a property list with every symbol (see section 6.3, page 131). A symbol can be used where a property list is expected; the property-list primitives automatically find the symbol's property list and use it.

3. A flavor instance may have a property list. The property list functions operate on instances by sending messages to them, so the flavor can store the property list any way it likes. See page 445.
4. A named structure may have a property list. The property list functions automatically call
\texttt{named-structure-invoke} when a named structure is supplied as the property list. See page 390.

5. A property list can be a memory cell in the middle of some data structure, such as a list,
an array, an instance, or a defstruct. An arbitrary memory cell of this kind is named by a
locative (see chapter 14, page 267). Such locatives are typically created with the \texttt{locf} special form
(see page 38).

Property lists of the first kind are called \textit{disembodied} property lists because they are not
associated with a symbol or other data structure. The way to create a disembodied property list is
\texttt{(ncons nil)}, or \texttt{(ncons data)} to store \texttt{data} in the car of the property list.

Suppose that, inside a program which deals with blocks, the property list of the symbol \texttt{b1}
contains this list (which would be the value of \texttt{(symbol-plist 'b1)}):

\begin{verbatim}
   (color blue on b6 associated-with (b2 b3 b4))
\end{verbatim}

The list has six elements, so there are three properties. The first property's name is the symbol
\texttt{color}, and its value is the symbol \texttt{blue}. One says that "the value of \texttt{b1}'s color property
is \texttt{blue}," or, informally, that "\texttt{b1}'s color property is \texttt{blue}." The program is probably representing
the information that the block represented by \texttt{b1} is painted blue. Similarly, it is probably
representing in the rest of the property list that block \texttt{b1} is on top of block \texttt{b6}, and that \texttt{b1}
is associated with blocks \texttt{b2}, \texttt{b3}, and \texttt{b4}.

\begin{verbatim}
get plist property-name &optional default-value
\end{verbatim}

get looks up \texttt{plist}'s \texttt{property-name} property. If it finds such a property, it returns
the value; otherwise, it returns \texttt{default-value}. If \texttt{plist} is a symbol, the symbol's associated
property list is used. For example, if the property list of \texttt{foo} is \texttt{(baz 3)}, then

\begin{verbatim}
(get 'foo 'baz) => 3
(get 'foo 'zoo) => nil
(get 'foo 'baz t) => 3
(get 'foo 'zoo t) => t
\end{verbatim}

\begin{verbatim}
getl plist property-name-list
\end{verbatim}

\begin{verbatim}
getl is like get, except that the second argument is a list of property names. getl searches
down \texttt{plist} for any of the names in \texttt{property-name-list}, until it finds a property
whose name is one of them. If \texttt{plist} is a symbol, the symbol's associated property list is
used.

getl returns the portion of the list inside \texttt{plist} beginning with the first such property that
it found. So the car of the returned list is a property name, and the \texttt{cadr} is the property
value. If none of the property names on \texttt{property-name-list} are on the property list, \texttt{getl}
returns nil. For example, if the property list of \texttt{foo} were

\begin{verbatim}
   (bar (1 2 3) baz (3 2 1) color blue height six-two)
\end{verbatim}

then

\begin{verbatim}
(getl 'foo '((baz height))
   => (baz (3 2 1) color blue height six-two)
\end{verbatim}

When more than one of the names in \texttt{property-name-list} is present in \texttt{plist}, which one
\texttt{getl} returns depends on the order of the properties. This is the only thing that depends
on that order. The order maintained by \texttt{putprop} and \texttt{defprop} is not defined (their
behavior with respect to order is not guaranteed and may be changed without notice).

**putprop** plist x property-name

This gives plist an property-name-property of x. After this is done, (get plist property-name) returns x. If plist is a symbol, the symbol's associated property list is used.

Example:

```
(putprop 'nixon t 'crook)
```

It is more modern to write

```
(setf (get plist property-name) x)
```

which avoids the counterintuitive order in which putprop takes its arguments.

**defprop** symbol x property-name

Defprop is a form of putprop with unevaluated arguments, which is sometimes more convenient for typing. Normally only a symbol makes sense as the first argument.

Example:

```
(defprop foo bar next-to)
```

is the same as

```
(putprop 'foo 'bar 'next-to)
```

**remprop** plist property-name

This removes plist's property-name property, by splicing it out of the property list. It returns that portion of the list inside plist of which the former property-name-property was the car. car of what remprop returns is what get would have returned with the same arguments. If plist is a symbol, the symbol's associated property list is used. For example, if the property list of foo was

```
(color blue height six-three near-to bar)
```

then

```
(remprop 'foo 'height) => (six-three near-to bar)
```

and foo's property list would be

```
(color blue near-to bar)
```

If plist has no property-name-property, then remprop has no side-effect and returns nil.

**getf** place property &optional default

Macro

Equivalent to (get (locf place) property default), except that getf is defined in Common Lisp, which does not have locf or locatives of any kind.

```
(setf (getf place property) value)
```

can be used to store properties into the property list at place.

**remf** place property

Macro

Equivalent to (remprop (locf place) property), except that remf is defined in Common Lisp.

**get-properties** place list-of-properties

Macro

Like (getf (locf place) list-of-properties) but returns slightly different values. Specifically, it searches the property list for a property name which is memq in list-of-properties, then returns three values:
proprname the property name found
value the value of that property
cell the property list cell found, whose car is proprname and whose cadr is value.

If nothing is found, all three values are nil.

It is possible to continue searching down the property list by using cddr of the third value as the argument to another call to get-properties.

5.11 Hash Tables

A hash table is a Lisp object that works something like a property list. Each hash table has a set of entries, each of which associates a particular key with a particular value (or sequence of values). The basic functions that deal with hash tables can create entries, delete entries, and find the value that is associated with a given key. Finding the value is very fast even if there are many entries, because hashing is used; this is an important advantage of hash tables over property lists. Hashing is explained in section 5.11.4, page 121.

A given hash table stores a fixed number of values for each key; by default, there is only one value. Each time you specify a new value or sequence of values, the old one(s) are lost.

There are three standard kinds of hash tables, which differ in how they compare keys: with eq, with eql or with equal. In other words, there are hash tables which hash on Lisp objects (using eq or eql) and there are hash tables which hash on trees (using equal).

You can also create a nonstandard hash table with any comparison function you like, as long as you also provide a suitable hash function. Any two objects which would be regarded as the same by the comparison function should produce the same hash code under the hash function. See the :compare-function and :hash-function keywords under make-hash-table, below.

The following discussion refers to the eq kind of hash table; the other kinds are described later, and work analogously.

eq hash tables are created with the function make-hash-table, which takes various options. New entries are added to hash tables with the puthash function. To look up a key and find the associated value(s), the gethash function is used. To remove an entry, use remhash. Here is a simple example.

(setq a (make-hash-table))

(puthash 'color 'brown a)
(puthash 'name 'fred a)

(gethash 'color a) => brown
(gethash 'name a) => fred
In this example, the symbols color and name are being used as keys, and the symbols brown and fred are being used as the associated values. The hash table remembers one value for each key, since we did not specify otherwise, and has two items in it, one of which associates from color to brown, and the other of which associates from name to fred.

Keys do not have to be symbols; they can be any Lisp object. Likewise values can be any Lisp object. Since eq does not work reliably on numbers (except for fixnums), they should not be used as keys in an eq hash table. Use an eql hash table if you want to hash on numeric values.

When a hash table is first created, it has a size, which is the number of entries it has room for. But hash tables which are nearly full become slow to search, so if more than a certain fraction of the entries become in use, the hash table is automatically made larger, and the entries are rehashed (new hash values are recomputed, and everything is rearranged so that the fast hash lookup still works). This is transparent to the caller; it all happens automatically.

The describe function (see page 791) prints a variety of useful information when applied to a hash table.

This hash table facility is similar to the hasharray facility of Interlisp, and some of the function names are the same. However, it is not compatible. The exact details and the order of arguments are designed to be consistent with the rest of Zetalisp rather than with Interlisp. For instance, the order of arguments to maphash is different, we do not have the Interlisp "system hash table", and we do not have the Interlisp restriction that keys and values may not be nil. Note, however, that the order of arguments to gethash, puthash, and remhash is not consistent with the Zetalisp's get, putprop, and remprop, either. This is an unfortunate result of the haphazard historical development of Lisp.

Hash tables are implemented as instances of the flavor hash-table. The internals of a hash table are subject to change without notice. Hash tables should be manipulated only with the functions and operations described below.

5.11.1 Hash Table Functions

make-hash-table &rest options
make-equal-hash-table &rest options

These functions create new hash tables. make-equal-hash-table creates an equal hash table. make-hash-table normally creates an eq hash table, but this can be overridden by keywords as described below. Valid option keywords are:

:size
Sets the initial size of the hash table, in entries, as a fixnum. The default is 64. The actual size is rounded up from the size you specify to the next size that is good for the hashing algorithm. The number of entries you can actually store in the hash table before it is rehashed is at least the actual size times the rehash threshold (see below).

:test
This keyword is the Common Lisp way to specify the kind of hashing desired. The value must be eq, eql or equal. The one specified is used as the compare function and an appropriate hash function is chosen.
automatically to go with it.

:compare-function
Specifies a function of two arguments which compares two keys to see if they count as the same for retrieval from this table. For reasonable results, this function should be an equivalence relation. The default is eq. For make-equal-hash-table the default is equal; that is the only difference between that function and make-hash-table.

:hash-function
Specifies a function of one argument which, given a key, computes its hash code. The hash code may be any Lisp object. The purpose of the hash function is to map equivalent keys into identical objects: if two keys would cause the compare function to return non-nil, the hash function must produce identical (eq) hash codes for them.

For an eq hash table, the key itself is a suitable hash code, so no hash function is needed. Then this option's value should be nil (identity would also work, but slower). nil is the default in make-hash-table. make-equal-hash-table specifies an appropriate function which uses sxhash.

:number-of-values
A positive integer which specifies how many values to associate with each key. The default is one.

:area
Specifies the area in which the hash table should be created. This is just like the :area option to make-array (see page 167). Defaults to nil (i.e. default-cons-area).

:rehash-function
Specifies the function to be used for rehashing when the table becomes full. Defaults to the internal rehashing function that does the usual thing. If you want to write your own rehashing function, you must know all the internals of how hash tables work. These internals are not documented here, as the best way to learn them is to read the source code.

:rehash-size
Specifies how much to increase the size of the hash table when it becomes full. This can be a fixnum which is the number of entries to add, or it can be a float which is the ratio of the new size to the old size. The default is 1.3, which causes the table to be made 30% bigger each time it has to grow.

:rehash-threshold
Sets a maximum fraction of the entries which can be in use before the hash table is made larger and rehashed. The default is 0.75. Alternately, an integer may be specified. It is the exact number of filled entries at which a rehash should be done. When the rehash happens, if the threshold is an integer it is increased in the same proportion as the table has grown.

:rehash-before-cold
If non-nil, this hash table should be rehashed (if that is necessary due to
garbage collection) by `disk-save'. This avoids a delay for rehashing the hash table the first time it is referenced after booting the saved band.

:actual-size

Specifies exactly the size for the hash table. Hash tables used by the microcode for flavor method lookup must be a power of two in size. This differs from :size in that :size is rounded up to a nearly prime number, but :actual-size is used exactly as specified. :actual-size overrides :size.

**hash-table-p** object

t if object is a hash table.

(\texttt{hash-table-p object})

is equivalent to

(\texttt{typep object 'hash-table})

The following functions are equivalent to sending appropriate messages to the hash table.

**gethash** key hash-table &optional default-value

Finds the entry in hash-table whose key is key, and return the associated value. If there is no such entry, returns default-value. Returns also a second value, which is t if an entry was found or nil if there is no entry for key in this table.

Returns also a third value, a list which overlays the hash table entry. Its car is the key; the remaining elements are the values in the entry. This is how you can access values other than the first, if the hash table contains more than one value per entry.

**puthash** key value hash-table &rest extra-values

Creates an entry associating key to value; if there is already an entry for key, then replace the value of that entry with value. Returns value. The hash table automatically grows if necessary.

If the hash table associates more than one value with each key, the remaining values in the entry are taken from extra-values.

**remhash** key hash-table

Removes any entry for key in hash-table. Returns t if there was an entry or nil if there was not.

**swaphash** key value hash-table &rest extra-values

This specifies new value(s) for key like puthash, but returns values describing the previous state of the entry, just like gethash. It returns the previous (replaced) associated value as the first value, and returns t as the second value if the entry existed previously.

**maphash** function hash-table &rest extra-args

For each occupied entry in hash-table, call function. The arguments passed to function are the key of the entry, the value(s) of the entry (however many there are), and the extra-args (however many there are).

If the hash table has more than one value per key, all the values, in order, are supplied as successive arguments.
\textbf{maphash-return} function hash-table

Like maphash, but accumulates and returns a list of all the values returned by \textit{function} when it is applied to the items in the hash table.

\textbf{clrhash} hash-table

Removes all the entries from \textit{hash-table}. Returns the hash table itself.

\textbf{hash-table-count} hash-table

Returns the number of filled entries in \textit{hash-table}.

\subsection*{5.11.2 Hash Table Operations}

Hash tables are instances, and support the following operations:

\begin{itemize}
  \item \textbf{size} \hspace{1cm} \textit{Operation on hash-table}
    
    Returns the number of entries in the hash table. Note that the hash table is rehashed when only a fraction of this many (the rehash threshold) are full.

  \item \textbf{filled-entries} \hspace{1cm} \textit{Operation on hash-table}
    
    Returns the number of entries that are currently occupied.

  \item \textbf{get-hash} key \hspace{1cm} \textit{Operation on hash-table}
  
  \item \textbf{put-hash} key &rest values \hspace{1cm} \textit{Operation on hash-table}
  
  \item \textbf{swap-hash} key &rest values \hspace{1cm} \textit{Operation on hash-table}
  
  \item \textbf{rem-hash} key \hspace{1cm} \textit{Operation on hash-table}
  
  \item \textbf{map-hash} function &rest extra-args \hspace{1cm} \textit{Operation on hash-table}
  
  \item \textbf{map-hash-return} function \hspace{1cm} \textit{Operation on hash-table}
  
  \item \textbf{clear-hash} \hspace{1cm} \textit{Operation on hash-table}
  
  \item \textbf{filled-entries} \hspace{1cm} \textit{Operation on hash-table}

  Are equivalent to the functions gethash, puthash, swaphash, remhash, maphash, maphash-return, clrhash and hash-table-count except that the hash table need not be specified as an argument because it is the object that receives the message. Those functions (documented in the previous section) actually work by invoking these operations.

  \item \textbf{modify-hash} key function &rest additional-args \hspace{1cm} \textit{Operation on hash-table}

  Passes the value associated with \textit{key} in the table to \textit{function}; whatever \textit{function} returns is stored in the table as the new value for \textit{key}. Thus, the hash association for \textit{key} is both examined and updated according to \textit{function}.

  The arguments passed to \textit{function} are \textit{key}, the value associated with \textit{key}, a flag (t if \textit{key} is actually found in the hash table), and the \textit{additional-args} that you specify.

  If the hash table stores more than one value per key, only the first value is examined and updated.
5.11.3 Hash Tables and the Garbage Collector

The eq type hash tables actually hash on the address of the representation of the object. equal hash tables do so too, if given keys containing unusual objects (other than symbols, numbers, strings and lists of the above). When the copying garbage collector changes the addresses of objects, it lets the hash facility know so that the next gethash will rehash the table based on the new object addresses.

There may eventually be an option to make-hash-table which tells it to make a “non-GC-protecting” hash table. This is a special kind of hash table with the property that if one of its keys becomes garbage, i.e. is an object not known about by anything other than the hash table, then the entry for that key will be removed silently from the table. When this option exists it will be documented in this section.

5.11.4 Hash Primitive

Hashing is a technique used in algorithms to provide fast retrieval of data in large tables. A function, known as the hash function, takes an object that might be used as a key, and produces a number associated with that key. This number, or some function of it, can be used to specify where in a table to look for the datum associated with the key. It is always possible for two different objects to hash to the same value; that is, for the hash function to return the same number for two distinct objects. Good hash functions are designed to minimize this by evenly distributing their results over the range of possible numbers. However, hash table algorithms must still deal with this problem by providing a secondary search, sometimes known as a rehash. For more information, consult a textbook on computer algorithms.

sxhash tree &optional ok-to-use-address

sxhash computes a hash code of a tree, and returns it as a fixnum. A property of sxhash is that (equal x y) always implies (= (sxhash x) (sxhash y)). The number returned by sxhash is always a non-negative fixnum. sxhash tries to compute its hash code in such a way that common permutations of an object, such as interchanging two elements of a list or changing one character in a string, always change the hash code.

Here is an example of how to use sxhash in maintaining hash tables of trees:

(defun knownp (x &aux i bkt) ; look up x in the table
  (setq i (abs (remainder (sxhash x) 176)))
  ; The remainder should be reasonably randomized.
  (setq bkt (aref table i))
  ; bkt is thus a list of all those expressions that
  ; hash into the same number as does x.
  (memq x bkt))

For an “intern” for trees, one could write:
(defun sintern (x &aux bkt i tem)
  (setq i (abs (remainder (sxhash x) 2n-1)))
  ;2n-1 stands for a power of 2 minus one.
  ;This is a good choice to randomize the
  ;result of the remainder operation.
  (setq bkt (aref table i))
  (cond ((setq tem (memq x bkt))
     (car tem))
    (t (aset (cons x bkt) table i)
     x)))

If sxhash is given a named structure or a flavor instance, or if such an object is part of a
tree that is sxhash'ed, it asks the object to supply its own hash code by performing the sxhash
operation if the object supports it. This should return a suitable nonnegative hash code. The
easiest way to compute one is usually by applying sxhash to one or more of the components of
the structure or the instance variables of the instance.

For named structures and flavor instances that do not handle the sxhash operation, and
other unusual kinds of objects, sxhash can optionally use the object's address as its hash code, if
you specify a non-nil second argument. If you use this option, you must be prepared to deal
with hash codes changing due to garbage collection.

sxhash provides what is called "hashing on equal"; that is, two objects that are equal are
considered to be "the same" by sxhash. If two strings differ only in alphabetic case, sxhash
returns the same thing for both of them, making it suitable for equalp hashing as well in some
cases.

Therefore, sxhash is useful for retrieving data when two keys that are not the same object
but are equal are considered the same. If you consider two such keys to be different, then you
need "hashing on eq", where two different objects are always considered different. In some Lisp
implementations, there is an easy way to create a hash function that hashes on eq, namely, by
returning the virtual address of the storage associated with the object. But in other
implementations, of which Zetalisp is one, this doesn't work, because the address associated with
an object can be changed by the relocating garbage collector. The hash tables created by make-
hash-table deal with this problem by using the appropriate subprimitives so that they interface
correctly with the garbage collector. If you need a hash table that hashes on eq, it is already
provided; if you need an eq hash function for some other reason, you must build it yourself,
either using the provided eq hash table facility or carefully using subprimitives.
5.12 Resources

Storage allocation is handled differently by different computer systems. In many languages, the programmer must spend a lot of time thinking about when variables and storage units are allocated and deallocated. In Lisp, freeing of allocated storage is normally done automatically by the Lisp system; when an object is no longer accessible to the Lisp environment, the garbage collector reuses its storage for some other object. This relieves the programmer of a great burden, and makes writing programs much easier.

However, automatic freeing of storage incurs an expense: more computer resources must be devoted to the garbage collector. If a program is designed to allocate temporary storage, which is then left as garbage, more of the computer must be devoted to the collection of garbage; this expense can be high. In some cases, the programmer may decide that it is worth putting up with the inconvenience of having to free storage under program control, rather than letting the system do it automatically, in order to prevent a great deal of overhead from the garbage collector.

It usually is not worth worrying about freeing of storage when the units of storage are very small things such as conses or small arrays. Numbers are not a problem, either: fixnums and short floats do not occupy storage, and the system has a special way of garbage-collecting the other kinds of numbers with low overhead. But when a program allocates and then gives up very large objects at a high rate (or large objects at a very high rate), it can be worthwhile to keep track of that one kind of object manually. Within the Lisp Machine system, there are several programs that are in this position. The Chaosnet software allocates and frees “packets”, which are moderately large, at a very high rate. The window system allocates and frees certain kinds of windows, which are very large, moderately often. Both of these programs manage their objects manually, keeping track of when they are no longer used.

When we say that a program “manually frees” storage, it does not really mean that the storage is freed in the same sense that the garbage collector frees storage. Instead, a list of unused objects is kept. When a new object is desired, the program first looks on the list to see if there is one around already, and if there is it uses it. Only if the list is empty does it actually allocate a new one. When the program is finished with the object, it returns it to this list.

The functions and special forms in this section perform the above function. The set of objects forming each such list is called a resource; for example, there might be a Chaosnet packet resource. defresource defines a new resource: allocate-resource allocates one of the objects; deallocate-resource frees one of the objects (putting it back on the list); and using-resource temporarily allocates an object and then frees it.
5.12.1 Defining Resources

\textbf{defresource} \hspace{1cm} \textit{Macro}

The \texttt{defresource} special form is used to define a new resource. The form looks like this:
\begin{verbatim}
(defresource name parameters
doc-string
  keyword value
  keyword value
  ...
)
\end{verbatim}

\texttt{name} should be a symbol; it is the name of the resource and gets a \texttt{defresource} property of the internal data structure representing the resource.

\texttt{parameters} is a lambda-list giving names and default values (if \texttt{optional} is used) of parameters to an object of this type. For example, if one had a resource of two-dimensional arrays to be used as temporary storage in a calculation, the resource would typically have two parameters, the number of rows and the number of columns. In the simplest case \texttt{parameters} is (\).

The documentation string is recorded for \texttt{(documentation name 'resource)} to access. It may be omitted.

The keyword options control how the objects of the resource are made and kept track of. The following keywords are allowed:

\texttt{:constructor} The \texttt{value} is either a form or the name of a function. It is responsible for making an object, and will be used when someone tries to allocate an object from the resource and no suitable free objects exist. If the \texttt{value} is a form, it may access the parameters as variables. If it is a function, it is given the internal data structure for the resource and any supplied parameters as its arguments; it will need to default any unsupplied optional parameters. This keyword is required.

\texttt{:free-list-size} The \texttt{value} is the number of objects which the resource data structure should have room, initially, to remember. This is not a hard limit, since the data structure will be made bigger if necessary.

\texttt{:initial-copies} The \texttt{value} is a number (or \texttt{nil} which means 0). This many objects will be made as part of the evaluation of the \texttt{defresource}; thus is useful to set up a pool of free objects during loading of a program. The default is to make no initial copies.

If initial copies are made and there are \texttt{parameters}, all the parameters must be \texttt{optional} and the initial copies will have the default values of the parameters.

\texttt{:initializer} The \texttt{value} is a form or a function as with \texttt{:constructor}. In addition to the parameters, a form here may access the variable \texttt{object} (in the current package). A function gets the object as its second argument, after the data structure and before the parameters. The purpose of the initializer function or form is to clean up the contents of the object before each use.
It is called or evaluated each time an object is allocated, whether just constructed or being reused.

`:finder`

The `value` is a form or a function as with `:constructor` and sees the same arguments. If this option is specified, the resource system does not keep track of the objects. Instead, the finder must do so. It will be called inside a `without-interrupts` and must find a usable object somehow and return it.

`:matcher`

The `value` is a form or a function as with `:constructor`. In addition to the parameters, a form here may access the variable `object` (in the current package). A function gets the object as its second argument, after the data structure and before the parameters. The job of the matcher is to make sure that the object matches the specified parameters. If no matcher is supplied, the system will remember the values of the parameters (including optional ones that defaulted) that were used to construct the object, and will assume that it matches those particular values for all time. The comparison is done with `equal` (not `eq`). The matcher is called inside a `without-interrupts`.

`:checker`

The job of the checker is to determine whether the object is safe to allocate. The `value` is a form or a function, as above. In addition to the parameters, a form here may access the variables `object` and `in-use-p` (in the current package). A function receives these as its second and third arguments, after the data structure and before the parameters. If no checker is supplied, the default checker looks only at `in-use-p`; if the object has been allocated and not freed it is not safe to allocate, otherwise it is. The checker is called inside a `without-interrupts`.

If these options are used with forms (rather than functions), the forms get compiled into functions as part of the expansion of `defresource`. The functions, whether user-provided or generated from forms, are given names like `(:property resource-name si:resource-constructor)`; these names are not guaranteed not to change in the future.

Most of the options are not used in typical cases. Here is an example:

```
(defun resource (two-dimensional-array (rows columns)
    :constructor (make-array (list rows columns)))
```

Suppose the array was usually going to be 100 by 100, and you wanted to preallocate one during loading of the program so that the first time you needed an array you wouldn't have to spend the time to create one. You might simply put

```
(coming-resource (foo two-dimensional-array 100 100)

```

after your `defresource`, which would allocate a 100 by 100 array and then immediately free it. Alternatively you could write:

```
(defun resource (two-dimensional-array
    (optional (rows 100) (columns 100))
    :constructor (make-array (list rows columns))
    :initial-copies 1)
```
Here is an example of how you might use the :matcher option. Suppose you wanted to have a resource of two-dimensional arrays, as above, except that when you allocate one you don’t care about the exact size, as long as it is big enough. Furthermore you realize that you are going to have a lot of different sizes and if you always allocated one of exactly the right size, you would allocate a lot of different arrays and would not reuse a pre-existing array very often. So you might write:

```
(defun resource sloppy-two-dimensional-array (rows columns)
  :constructor (make-array (list rows columns))
  :matcher (and (≥ (array-dimension-n 1 object) rows)
                 (≥ (array-dimension-n 2 object) columns)))
```

5.12.2 Allocating Resource Objects

`allocate-resource resource-name &rest parameters`
Allocates an object from the resource specified by resource-name. The various forms and/or functions given as options to defresource, together with any parameters given to allocate-resource, control how a suitable object is found and whether a new one has to be constructed or an old one can be reused.

Note that the using-resource special form is usually what you want to use, rather than allocate-resource itself; see below.

`deallocate-resource resource-name resource-object`
Frees the object resource-object, returning it to the free-object list of the resource specified by resource-name.

`using-resource (variable resource parameters...) body...` Macro
The body forms are evaluated sequentially with variable bound to an object allocated from the resource named resource, using the given parameters. The parameters (if any) are evaluated, but resource is not.

`using-resource` is often more convenient than calling allocate-resource and deallocate-resource. Furthermore it is careful to free the object when the body is exited, whether it returns normally or via throw. This is done by using unwind-protect; see page 82.

Here is an example of the use of resources:
```
(defun resource huge-16b-array (&optional (size 1000))
  :constructor (make-array size :type 'art-16b))
```

```
(defun do-complex-computation (x y)
  (using-resource (temp-array huge-16b-array)
      (aset 5 temp-array i) ;Within the body, the array can be used.
    ...))
```

;The array is deallocated at the end.
deallocate-whole-resource resource-name
Frees all objects in resource-name. This is like doing deallocate-resource on each one individually. This function is often useful in warm-boot initializations.

map-resource function resource-name &rest extra-args
Calls function on each object created in resource-name. Each time function is called, it receives three fixed arguments, plus whatever extra-args were specified. The three fixed arguments are an object of the resource; t if the object is currently allocated ("in use"); and the resource data structure itself.

clear-resource resource-name
Forgets all of the objects being remembered by the resource specified by resource-name. Future calls to allocate-resource will create new objects. This function is useful if something about the resource has been changed incompatibly, such that the old objects are no longer usable. If an object of the resource is in use when clear-resource is called, an error will be signaled when that object is deallocated.

5.12.3 Accessing the Resource Data Structure

The constructor, initializer, matcher and checker functions receive the internal resource data structure as an argument. This is a named structure array whose elements record the objects both free and allocated, and whose array leader contains sundry other information. This structure should be accessed using the following primitives:

si:resource-object resource-structure index
Returns the index’th object remembered by the resource. Both free and allocated objects are remembered.

si:resource-in-use-p resource-structure index
Returns t if the index’th object remembered by the resource has been allocated and not deallocated. Simply defined resources will not reallocate an object in this state.

si:resource-parameters resource-structure index
Returns the list of parameters from which the index’th object was originally created.

si:resource-n-objects resource-structure
Returns the number of objects currently remembered by the resource. This will include all objects ever constructed, unless clear-resource has been used.

si:resource-parametizer resource-structure
Returns a function, created by defresource, which accepts the supplied parameters as arguments, and returns a complete list of parameter values, including defaults for the optional ones.
5.12.4 Fast Pseudo-Resources

When small temporary data structures are allocated so often that they amount to a considerable drain of storage space, an ordinary resource may be unacceptably slow. Here is a simple technique that provides in such cases nearly all the benefit of a resource while costing nearly nothing. The function read uses it to allocate a buffer for reading tokens of input.

\begin{verbatim}
(defvar buffer-for-reuse nil)
(defsubst get-buffer ()
  (or (do (old)
            ((%store-conditional (locf buffer-for-reuse)
              (setq old buffer-for-reuse)
              nil)
             old))
    (construct-new-buffer)))

(defsubst free-buffer (buffer)
  (setq buffer-for-reuse buffer))
\end{verbatim}

To allocate a buffer for use, do (get-buffer). To free it when you are done with it, call free-buffer. It is assumed that construct-new-buffer is the function which can create a new buffer when there is none available for reuse.

This technique keeps track of at most one buffer which has been freed and may be reused. It is not effective in this simple form when more than one buffer is needed at any given time by one application. In the case of read, only one token is being read in at any time.

It is safe for more than one process to call read because get-buffer is designed to guarantee that a request cannot get a buffer already handed out and not freed. Likewise, nothing terrible happens if there is an error inside read and read is called recursively within the debugger. The only problem is that multiple buffers will be allocated, which means that some of them will be lost. But the cost of this is minor in the cases where this technique is applicable. For example, if two processes are reading files, process switching will probably happen a few times a second, each time costing one buffer not reused. This is insignificant compared to the storage used up for other purposes by reading large amounts of data.