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. 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. While these are perfectly good Lisp objects, and there are functions to deal with them, many other functions will have trouble with them. Functions that expect lists as their arguments often iterate down the chain of conses waiting to see a **nil**, and when handed a circular list this can cause them to compute forever. The printer (see page 388) is one of these functions; if you try to print a circular list the printer will never stop producing text. You have to be careful what you do with circular lists.

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 will 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 72.

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".

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*

This function creates a cons in a specific *area*. (Areas are an advanced feature of storage management, explained in chapter 15; 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*)

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 17, page 248.

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 72). Instead, the cons itself serves as a "locative" to its cdr (see page 197).

5.2 Lists

length *list-or-array*

length 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:

```
(length nil) => 0
(length '(a b c d)) => 4
(length '(a (b c) d)) => 3
(length "foobar") => 6
```

length could have been defined by:

```
(defun length (x)
  (if (arrayp x) (array-active-length x)
    (do ((n 0 (1+ n))
         (y x (cdr y)))
        ((null y) n))))
```

or by

```
(defun length (x)
   (cond ((arrayp x) (array-active-length x))
         ((null x) 0)
         ((1+ (length (cdr x)))) ))
```

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.

rest1 *list*
rest2 *list*
rest3 *list*
rest4 *list*

rest*n* returns the rest of the elements of a list, starting with element *n* (counting the first element as the zeroth). Thus 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.

nth *n list*

(nth *n list*) returns the *n*'th element of *list*, where the zeroth element is the car of the list.

Examples:

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

If *n* is greater than the length of the list, nil is returned.

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))
         (list list (cdr list)))
        ((= 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.
Example:

```
(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 &rest *args*

list 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 l))
           (a args (cdr a)))
          ((null a) list)
        (rplaca l (car a)))))
```

list* &rest *args*

list* is like **list** except that the last cons of the constructed list is "dotted". It must be given at least one argument.

Example:

```
(list* 'a 'b 'c 'd) => (a b c . d)
```

This is like

```
(cons 'a (cons 'b (cons 'c 'd)))
```

More examples:

```
(list* 'a 'b) => (a . b)
(list* 'a) => a
```

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* &rest *options*

This creates and returns a list containing *length* elements. *length* should be a fixnum. *options* are alternating keywords and values. The keywords may be either of the following:

:area — The value specifies in which area (see chapter 15, page 223) the list should be created. It should be either an area number (a fixnum), or **nil** to mean the default area.

:initial-value — The elements of the list will all be this value. It defaults to **nil**.

make-list always creates a cdr-coded list (see section 5.4, page 72).

Examples:

```
(make-list 3) => (nil nil nil)
(make-list 4 ':initial-value 7) => (7 7 7 7)
```

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 will continue to work, but the new keyword-argument form is preferred.

circular-list &rest *args*

circular-list 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))
  (rplacd (last elements) elements)
  elements)
```

copylist *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 72) to minimize storage. If the list is "dotted", that is, if (cdr (last *list*)) is a non-nil atom, this will be true of the returned list also. 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 72). This makes for increased efficiency if you nconc something onto the list later.

copyalist *list* &optional *area*

copyalist is for copying association lists (see section 5.5, page 74). 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.

copytree *tree* &optional *area*

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

reverse *list*

reverse creates a new list whose elements are the elements of *list* taken in reverse order. reverse does not modify its argument, unlike nreverse which is faster but does modify its argument. The list created by reverse is not cdr-coded.

Example:

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

reverse could have been defined by:

```
(defun reverse (x)
    (do ((l x (cdr l))          ; scan down argument,
         (r nil                 ; putting each element
            (cons (car l) r)))  ; into list, until
        ((null l) r)))          ; no more elements.
```

nreverse *list*

nreverse reverses its argument, which should be a list. The argument is destroyed by rplacd's all through the list (cf. reverse).
Example:

```
(nreverse '(a b c)) => (c b a)
```

nreverse could have been defined by:

```
(defun nreverse  (x)
    (cond ((null x) nil)
          ((nreverse1 x nil))))

(defun nreverse1 (x y)          ;auxiliary function
    (cond ((null (cdr x)) (rplacd x y))
          ((nreverse1 (cdr x) (rplacd x y)))))
          ;; this last call depends on order of argument evaluation.
```

Currently, nreverse does something inefficient with cdr-coded lists (see section 5.4, page 72), because it just uses rplacd in the straightforward way. This may be fixed someday. In the meantime reverse might be preferable in some cases.

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 will share the conses of the last argument to append, but all of the other conses will be 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 66); 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 67).

Example:

```
(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 is 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 d e f ...), repeating forever.

nconc could have been defined by:

```
(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.
```

nreconc *x y*

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

nreconc could have been defined by:

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

using the same nreverse1 as above.

butlast *list*

This creates and returns a list with the same elements as *list*, excepting the last element.

Examples:

```
(butlast '(a b c d)) => (a b c)
(butlast '((a b) (c d))) => ((a b))
(butlast '(a)) => nil
(butlast nil) => nil
```

The name is from the phrase "all elements but the last".

nbutlast *list*

This is the destructive version of butlast; it changes the cdr of the second-to-last cons of the list to nil. If there is no second-to-last cons (that is, if the list has fewer than two elements) it returns nil.

Examples:

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

firstn *n list*

firstn 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 will be **nil**.

Example:

```
(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** will return *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** will return **nil**.

ldiff *list tail*

list should be a list, and *tail* should be one of the conses that make up *list*. **ldiff** (meaning "list difference") will return 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)
(ldiff x y) => (a b c)
(ldiff x nil) => (a b c d e)
(ldiff x x) => nil
```

but

```
(ldiff '(a b c d) '(c d)) => (a b c d)
```

since the *tail* was not **eq** to any part of the *list*.

union *list* &rest *more-lists*

If lists are regarded as sets of their elements, **union** returns a list which is the union of the lists which are supplied as arguments. If none of the arguments contains any duplicate elements, neither does the value returned by **union**. Elements are compared using **eq**.

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**.

nunion *list* &rest *more-lists*

If lists are regarded as sets of their elements, **nunion** 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 **nunion**. Elements are compared using **eq**.

As with **delq**, **nunion**'s value should be stored in place of the first argument if you want to be sure that the argument is changed. Consider what happens if the argument's initial value is **nil**.

nintersection *list* &rest *more-lists*

If lists are regarded as sets of their elements, **intersection** modifies *list* to be the intersection of the lists which are supplied as arguments. This is done by deleting any elements which do not belong to the intersection. If *list* initially contains no duplicate elements, neither does the value returned by **nintersection**. Elements are compared using **eq**.

As with **delq**, **nunion**'s value should be stored in place of the first argument if you want to be sure that the argument is changed. Consider what happens if the argument's first element is removed.

5.3 Alteration of List Structure

The functions **rplaca** and **rplacd** are used to make alterations in already-existing list structure; that is, to change the cars and cdrs of existing conses.

The structure is not copied but is physically altered; hence caution should be exercised when using these functions, as strange side-effects can occur if portions of list structure become shared unbeknownst to the programmer. The **nconc**, **nreverse**, **nreconc**, and **nbutlast** functions already described, and the **delq** family described later, have the same property.

rplaca *x y*

(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*

(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)
```

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 66); the old practice of using **subst** to copy trees is unclear and obsolete.

Note: certain details of **subst** may be changed in the future. It may possibly be changed to use **eq** rather than **equal** for the comparison, and possibly may substitute only in cars, not in cdrs. This is still being discussed.

nsubst *new old tree*

nsubst is a destructive version of **subst**. The list structure of *tree* is altered by replacing each occurrence of *old* with *new*. **nsubst** could have been defined as

```
(defun nsubst (new old tree)
    (cond ((eq tree old) new)     ; If item eq to 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)))
```

sublis *alist tree*

sublis makes substitutions for symbols in a tree. The first argument to **sublis** is an association list (see section 5.5, page 74). The second argument is the tree in which substitutions are to be made. **sublis** looks at all symbols in the fringe of the tree; if a symbol appears in the association list occurrences of it are replaced by the object it is associated with. The argument is not modified; new conses are created where necessary and only where necessary, so the newly created tree shares as much of its substructure as possible with the old. For example, if no substitutions are made, the result is just the old tree.

Example:

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

sublis could have been defined by:

```
(defun sublis (alist sexp)
  (cond ((atom sexp)
         (let ((tem (assq sexp alist)))
           (if tem (cdr tem) sexp)))
        ((let ((car (sublis alist (car sexp)))
               (cdr (sublis alist (cdr sexp))))
           (if (and (eq (car sexp) car) (eq (cdr sexp) cdr))
               sexp
               (cons car cdr))))))
```

nsublis *alist tree*

nsublis is like sublis but changes the original tree instead of creating new.

nsublis could have been defined by:

```
(defun nsublis (alist tree)
  (cond ((atom tree)
         (let ((tem (assq tree alist)))
           (if tem (cdr tem) tree)))
        (t (rplaca tree (nsublis alist (car tree)))
           (rplacd 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. There's no problem with doing that.

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 will find 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 will be 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 be seen to 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 will be changed.

(**copylist** *x*) is a suitable way to copy a list, converting it into compact form (see page 66).

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, which models (approximately) the concept of a *set*. 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. Set union, intersection, and difference functions can be easily written using these.

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,

```
((tweety . bird) (sylvester . 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 19, page 298) or as flavors (chapter 20, page 321).

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 5.10, page 83), and a hashing function (**sxhash**) to aid users in constructing their own facilities.

5.6 Lists as Tables

memq *item list*

(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:

```
(memq 'a '(1 2 3 4)) => nil
(memq 'a '(g (x a y) c a d e a f)) => (a d e a f)
```

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:

```
(let ((sublist (memq x z)))      ;Search for x in the list z.
  (if (not (null sublist))       ;If it is found,
      (rplaca sublist y)))       ;Replace it with y.
```

memq could have been defined by:

```
(defun memq (item list)
    (cond ((null list) nil)
          ((eq item (car list)) list)
          (t (memq item (cdr list))) ))
```

memq is hand-coded in microcode and therefore especially fast.

member *item list*

member is like memq, except equal is used for the comparison, instead of eq.

member could have been defined by:

```
(defun member (item list)
    (cond ((null list) nil)
          ((equal item (car list)) list)
          (t (member item (cdr list))) ))
```

mem *predicate item list*

mem is the same as memq except that it takes an extra argument which should be a predicate of two arguments, which is used for the comparison instead of eq. (mem 'eq a b) is the same as (memq a b). (mem 'equal a b) is the same as (member a b).

mem is usually used with equality predicates other than eq and equal, such as =, char-equal or string-equal. It can also be used with non-commutative predicates. The predicate is called with *item* as its first argument and the element of *list* as its second argument, so

```
(mem #'< 4 list)
```

finds the first element in *list* for which (< 4 *x*) is true; that is, it finds the first element greater than 4.

find-position-in-list *item list*

find-position-in-list looks down *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 occurence of *item*, or **nil** if it did not find it at all. This function is sort of the complement of **nth** (see page 63); 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
```

find-position-in-list-equal *item list*

find-position-in-list-equal is exactly the same as **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 will be deleted.

Example:

```
(delq 'a '(b a c (a b) d a e)) => (b c (a b) d e)
```

delq could have been defined by:

```
(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 (rplacd 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.

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). (cf. mem, page 75)

remq *item list* &optional *n*

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

```
(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.

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). (cf. mem, page 75)

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.

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.

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

every 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.

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

some 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.

5.7 Association Lists

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) now
```

A typical 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))) ))
```

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). (cf. mem, page 75) As with mem, you may use non-commutative predicates; the first argument to the predicate is *item* and the second is the key of the element of *alist*.

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*). (cf. mem, page 75) As with mem, you may use non-commutative predicates; the first argument to the predicate is *item* and the second is the key of the element of *alist*.

rassq *item alist*

rassq means "reverse assq". It is like assq, but it tries to find an element of *alist* whose *cdr* (not car) is eq to *item*. rassq could have been defined by:

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

rassoc *item alist*

rassoc is to rassq as assoc is to assq. That is, it finds an element whose cdr is equal to *item*.

rass *predicate item alist*

rass is to rassq as ass is to assq. That is, it takes a predicate to be used instead of eq (cf. mem, page 75). As with mem, you may use non-commutative predicates; the first argument to the predicate is *item* and the second is the cdr of the element of *alist*.

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:

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

pairlis *cars cdrs*

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))
```

5.8 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 66). The system will not 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...* *Special Form*
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.

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) *Condition*

This is signaled if you rplacd a stack list (or a list overlayed with an array, or any other sort of structure).

5.9 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 99). A symbol can be used where a property list is expected; the property-list primitives will 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 359).

4. 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 13, page 197). Such locatives are typically created with the **locf** special form (see page 271).

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

Here is an example of the list of entries inside the property list of a symbol named **b1** which is being used by a program which deals with blocks:

```
(color blue on b6 associated-with (b2 b3 b4))
```

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

get *plist property-name*

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

```
(get 'foo 'baz) => 3
(get 'foo 'zoo) => nil
```

getl *plist property-name-list*

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

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

```
(bar (1 2 3) baz (3 2 1) color blue height six-two)
```

then

```
(getl 'foo '(baz height))
  => (baz (3 2 1) color blue height six-two)
```

When more than one of the names in *property-name-list* is present in *plist*, which one **getl** returns depends on the order of the properties. This is the only thing that depends on that order. The order maintained by **putprop** and **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*) will return *x*. If *plist* is a symbol, the symbol's associated property list is used.
Example:

```
(putprop 'Nixon t 'crook)
```

defprop *symbol x property-name* *Special Form*

defprop is a form of putprop with "unevaluated arguments", which is sometimes more convenient for typing. Normally it doesn't make sense to use a property list rather than a symbol as the first (or *plist*) 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.

5.10 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 5.10.3, page 87.

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.

Hash tables come in two kinds, the difference being whether the keys are compared using eq or using equal. In other words, there are hash tables which hash on Lisp *objects* (using eq) and there are hash tables which hash on trees (using equal). The following discussion refers to the eq kind of hash table; the other kind is described later, and works 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.

When a hash table is first created, it has a *size*, which how many entries it contains. 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 will grow automatically, and the entries will be *rehashed* (new hash values will be recomputed, and everything will be rearranged so that the fast hash lookup still works). This is transparent to the caller; it all happens automatically.

The **describe** function (see page 641) 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 with a special kind of array. **arrayp** of a hash table will return **t**. However, it is not recommended to use ordinary array operations on a hash table, because the way the array elements are used to represent the entries is internal and subject to change. Hash tables should be manipulated only with the functions described below.

5.10.1 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 100 (octal). 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).

:number-of-values
: 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 126). 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 will have to understand 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 flonum 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 0.7s0.

: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-function
: Specifies a function which, given a key, computes its hash code. For an eq hash table, the key is the code, and this option's value should be nil (which is the default for make-hash-table). make-equal-hash-table specifies an appropriate function which uses sxhash.

:compare-function
: Specifies a function which compares two keys to see if they count as the same for retrieval from this table. The default is eq, or equal for make-

equal-hash-table.

:funcallable-p Specifies whether the hash table should attempt to handle messages if applied as a function. If this option is non-nil, when the hash table is called as a function it uses the first argument as a hash key to find a function to call. The function is given the same arguments that the hash table received. Funcallable hash tables are somewhat analogous to select-method objects (see page 163).

Specifying a funcallable hash table automatically forces certain other options to have values that the microcode expects to deal with.

eq and equal hash tables are not the only possible kinds. You can create a 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.

gethash *key hash-table*

Finds the entry in *hash-table* whose key is *key*, and return the associated value. If there is no such entry, return nil. Returns 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. In particular, 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*

For each entry in *hash-table*, call *function* on two arguments: the key of the entry and the value of the entry.

If the hash table has more than one value per key, all the values, in order, are supplied as arguments, with the corresponding key.

maphash-return *function hash-table*
Like maphash, but accumulates and returns a list of all the values returned by *function* when it is applied to the items in the hash table.

clrhash *hash-table*
Removes all the entries from *hash-table*. Returns the hash table itself.

5.10.2 Hash Tables and the Garbage Collector

The eq type hash tables actually hash on the address of the representation of the object. When the copying garbage collector changes the addresses of objects, it lets the hash facility know so that gethash will rehash the table based on the new object addresses.

There will 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 silently removed from the table. When this option exists it will be documented in this section.

5.10.3 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, possibly a large one. 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, will 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))
```

To write an "intern" for trees, one could

```
(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 one is part of a tree that is sxhashed, it will ask 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. In particular, if two strings differ only in alphabetic case, sxhash will return the same thing for both of them because they are equal. The value returned by sxhash does not depend on the value of alphabetic-case-affects-string-comparison (see page 144).

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.

5.11 Sorting

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

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

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

The sorting package is smart about compact lists; it sorts compact sublists as if they were arrays. See section 5.4, page 72 for an explanation of compact lists, and A. I. Memo 587 by Guy L. Steele Jr. for an explanation of the sorting algorithm.

sort *table predicate*

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

The **sort** function proceeds to sort the contents of the array or list under the ordering imposed by the predicate, and returns the array or list modified into sorted order. Note that since sorting requires many comparisons, and thus many calls to the predicate, sorting will be much faster if the predicate is a compiled function rather than interpreted.

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

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

(sort 'fooarray
      (function (lambda (x y)
          (alphalessp (mostcar x) (mostcar y)))))
```

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

```
(Tokens (The lion sleeps tonight))
(Carpenters (Close to you))
((Rolling Stones) (Brown sugar))
((Beach Boys) (I get around))
(Beatles (I want to hold your hand))
```

then after the sort fooarray would contain:

```
((Beach Boys) (I get around))
(Beatles (I want to hold your hand))
(Carpenters (Close to you))
((Rolling Stones) (Brown sugar))
(Tokens (The lion sleeps tonight))
```

When sort is given a list, it may change the order of the conses of the list (using rplacd), and so it cannot be used merely for side-effect; only the *returned value* of sort will be the sorted list. This will mess up the original list; if you need both the original list and the sorted list, you must copy the original and sort the copy (see copylist, page 66).

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

If the argument to sort is an array with a fill pointer, note that, like most functions, sort considers the active length of the array to be the length, and so only the active part of the array will be sorted (see array-active-length, page 130).

sortcar *x predicate*

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

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

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

stable-sort *x predicate*

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

stable-sortcar *x predicate*

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

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

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

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

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

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, it is garbage collected. 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 small flonums 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 very 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

defresource *Special Form*

The **defresource** special form is used to define a new resource. The form looks like this:

```
(defresource name parameters
   keyword value
   keyword value
   ...)
```

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

parameters is a lambda-list giving names and default values (if **&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 *parameters* is **()**.

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

:constructor — The *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 *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.

:free-list-size — The *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.

:initial-copies — The *value* is a number (or **nil** which means 0). This many objects will be made as part of the evaluation of the **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 *parameters*, all the parameters must be **&optional** and the initial copies will have the default values of the parameters.

:initializer — 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 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 *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. The job of the checker is to determine whether the object is safe to allocate. 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:

```
(defresource 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

```
(using-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:

```
(defresource 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:

```
(defresource 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 *name* &rest *parameters*

Allocate an object from the resource specified by *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 *name resource*

Free the object *resource*, returning it to the free-object list of the resource specified by *name*.

clear-resource *name*

Forget all of the objects being remembered by the resource specified by *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 signalled when that object is deallocated.

using-resource (*variable resource parameters...*) *body...* *Special Form*

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 56.

Here is an example of the use of resources:

```
(defresource 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)
    ...                                  ;Within the body, the array can be used.
    (aset 5 temp-array i)
    ...))                                ;The array is returned at the end.
```

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.