17. Macros

17.1 Introduction to Macros

If eval is handed a list whose car is a symbol, then eval inspects the definition of the symbol to find out what to do. If the definition is a cons, and the car of the cons is the symbol macro, then the definition (i.e. that cons) is called a *macro*. The cdr of the cons should be a function of one argument. eval applies the function to the form it was originally given, takes whatever is returned, and evaluates that in lieu of the original form.

Here is a simple example. Suppose the definition of the symbol first is

```
(macro lambda (x)
       (list 'car (cadr x)))
```

This thing is a macro: it is a cons whose car is the symbol macro. What happens if we try to evaluate a form (first '(a b c))? Well, eval sees that it has a list whose car is a symbol (namely, first), so it looks at the definition of the symbol and sees that it is a cons whose car is macro; the definition is a macro.

eval takes the cdr of the cons, which is supposed to be the macro's *expander function*, and calls it providing as an argument the original form that eval was handed. So it calls (lambda (x) (list 'car (cadr x))) with argument (first '(a b c)). Whatever this returns is the *expansion* of the macro call. It will be evaluated in place of the original form.

In this case, x is bound to (first '(a b c)), (cadr x) evaluates to '(a b c), and (list 'car (cadr x)) evaluates to (car '(a b c)), which is the expansion. eval now evaluates the expansion. (car '(a b c)) returns a, and so the result is that (first '(a b c)) returns a.

What have we done? We have defined a macro called first. What the macro does is to *translate* the form to some other form. Our translation is very simple—it just translates forms that look like (first *x*) into (car *x*), for any form *x*. We can do much more interesting things with macros, but first we will show how to define a macro.

macro *Special Form*

The primitive special form for defining macros is macro. A macro definition looks like this:

```
(macro name (arg)
    body)
```

name can be any function spec. *arg* must be a variable. *body* is a sequence of Lisp forms that expand the macro; the last form should return the expansion.

To define our first macro, we would say

```
(macro first (x)
    (list 'car (cadr x)))
```

Here are some more simple examples of macros. Suppose we want any form that looks like (addone *x*) to be translated into (plus 1 *x*). To define a macro to do this we would say

```
(macro addone (x)
   (list 'plus '1 (cadr x)))
```

Now say we wanted a macro which would translate (increment *x*) into (setq *x* (1+ *x*)). This would be:

```
(macro increment (x)
    (list 'setq (cadr x) (list '1+ (cadr x))))
```

Of course, this macro is of limited usefulness. The reason is that the form in the *cadr* of the increment form had better be a symbol. If you tried (increment (car x)), it would be translated into (setq (car x) (1+ (car x))), and setq would complain. (If you're interested in how to fix this problem, see setf (page 270); but this is irrelevant to how macros work.)

You can see from this discussion that macros are very different from functions. A function would not be able to tell what kind of subforms are around in a call to itself; they get evaluated before the function ever sees them. However, a macro gets to look at the whole form and see just what is going on there. Macros are *not* functions; if first is defined as a macro, it is not meaningful to apply first to arguments. A macro does not take arguments at all; its expander function takes a *Lisp form* and turns it into another *Lisp form*.

The purpose of functions is to *compute*; the purpose of macros is to *translate*. Macros are used for a variety of purposes, the most common being extensions to the Lisp language. For example, Lisp is powerful enough to express many different control structures, but it does not provide every control structure anyone might ever possibly want. Instead, if a user wants some kind of control structure with a syntax that is not provided, he can translate it into some form that Lisp *does* know about.

For example, someone might want a limited iteration construct which increments a variable by one until it exceeds a limit (like the FOR statement of the BASIC language). He might want it to look like

```
(for a 1 100 (print a) (print (* a a)))
```

To get this, he could write a macro to translate it into

```
(do a 1 (1+ a) (> a 100) (print a) (print (* a a)))
```

A macro to do this could be defined with

```
(macro for (x)
  (cons 'do
        (cons (cadr x)
              (cons (caddr x)
                    (cons (list '1+ (cadr x))
                          (cons (list '> (cadr x) (cadddr x))
                                (cddddr x)))))))
```

Now he has defined his own new control structure primitive, and it will act just as if it were a special form provided by Lisp itself.

17.2 Aids for Defining Macros

The main problem with the definition for the **for** macro is that it is verbose and clumsy. If it is that hard to write a macro to do a simple specialized iteration construct, one would wonder how anyone could write macros of any real sophistication.

There are two things that make the definition so inelegant. One is that the programmer must write things like "(cadr x)" and "(cddddr x)" to refer to the parts of the form he wants to do things with. The other problem is that the long chains of calls to the **list** and **cons** functions are very hard to read.

Two features are provided to solve these two problems. The **defmacro** macro solves the former, and the "backquote" (`) reader macro solves the latter.

17.2.1 Defmacro

Instead of referring to the parts of our form by "(cadr x)" and such, we would like to give names to the various pieces of the form, and somehow have the (cadr x) automatically generated. This is done by a macro called **defmacro**. It is easiest to explain what **defmacro** does by showing an example. Here is how you would write the **for** macro using **defmacro**:

```
(defmacro for (var lower upper . body)
  (cons 'do
        (cons var
              (cons lower
                    (cons (list '1+ var)
                          (cons (list '> var upper)
                                body))))))
```

The (var lower upper . body) is a *pattern* to match against the body of the form (to be more precise, to match against the cdr of the argument to the macro's expander function). If defmacro tries to match the two lists

```
(var lower upper . body)
```

and

```
(a 1 100 (print a) (print (* a a)))
```

var will get bound to the symbol a, lower to the fixnum 1, upper to the fixnum 100, and body to the list ((print a) (print (* a a))). Then inside the body of the defmacro, var, lower, upper, and body are variables, bound to the matching parts of the macro form.

defmacro *Macro*

defmacro is a general purpose macro-defining macro. A defmacro form looks like

```
(defmacro name pattern . body)
```

The *pattern* may be anything made up out of symbols and conses. It is matched against the body of the macro form; both *pattern* and the form are car'ed and cdr'ed identically, and whenever a non-nil symbol is hit in *pattern*, the symbol is bound to the corresponding part of the form. All of the symbols in *pattern* can be used as variables within *body*. *name* is the name of the macro to be defined; it can be any function spec (see section 10.2, page 154). *body* is evaluated with these bindings in effect, and its result is returned to the evaluator as the expansion of the macro.

Note that the pattern need not be a list the way a lambda-list must. In the above example, the pattern was a "dotted list", since the symbol **body** was supposed to match the cddddr of the macro form. If we wanted a new iteration form, like **for** except that our example would look like

```
(for a (1 100) (print a) (print (* a a)))
```

(just because we thought that was a nicer syntax), then we could do it merely by modifying the pattern of the **defmacro** above; the new pattern would be **(var (lower upper) . body)**.

Here is how we would write our other examples using **defmacro**:

```
(defmacro first (the-list)
    (list 'car the-list))

(defmacro addone (form)
   (list 'plus '1 form))

(defmacro increment (symbol)
   (list 'setq symbol (list '1+ symbol)))
```

All of these were very simple macros and have very simple patterns, but these examples show that we can replace the **(cadr x)** with a readable mnemonic name such as **the-list** or **symbol**, which makes the program clearer, and enables documentation facilities such as the **arglist** function to describe the syntax of the special form defined by the macro.

There is another version of **defmacro** which defines displacing macros (see section 17.6, page 267). **defmacro** has other, more complex features; see section 17.7, page 268.

17.2.2 Backquote

Now we deal with the other problem: the long strings of calls to **cons** and **list**. This problem is relieved by introducing some new characters that are special to the Lisp reader. Just as the single-quote character makes it easier to type things of the form **(quote** *x*), so will some more new special characters make it easier to type forms that create new list structure. The functionality provided by these characters is called the *backquote* facility, which allows you to create a list from a template including constant and variable parts.

The backquote facility is used by giving a backquote character (`), followed by a form. If the form does not contain any use of the comma character, the backquote acts just like a single quote: it creates a form which, when evaluated, produces the form following the backquote. For example,

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

So in the simple cases, backquote is just like the regular single-quote macro. The way to get it to do interesting things is to include a comma somewhere inside of the form following the backquote. The comma is followed by a form, and that form gets evaluated even though it is inside the backquote. For example,

```
(setq b 1)
`(a b c)  => (a b c)
`(a ,b c) => (a 1 c)
`(abc ,(+ b 4) ,(- b 1) (def ,b)) => (abc 5 0 (def 1))
```

In other words, backquote quotes everything *except* things preceded by a comma; those things get

evaluated.

A list following a backquote can be thought of as a template for some new list structure. The parts of the list that are preceded by commas are forms that fill in slots in the template; everything else is just constant structure that will appear in the result. This is usually what you want in the body of a macro; some of the form generated by the macro is constant, the same thing on every invocation of the macro. Other parts are different every time the macro is called, often being functions of the form that the macro appeared in (the "arguments" of the macro). The latter parts are the ones for which you would use the comma. Several examples of this sort of use follow.

When the reader sees the '(a ,b c) it is actually generating a form such as (list 'a b 'c). The actual form generated may use list, cons, append, or whatever might be a good idea; you should never have to concern yourself with what it actually turns into. All you need to care about is what it evaluates to. Actually, it doesn't use the regular functions cons, list, and so forth, but uses special ones instead so that the grinder can recognize a form which was created with the backquote syntax, and print it using backquote so that it looks like what you typed in. You should never write any program that depends on this, anyway, because backquote makes no guarantees about how it does what it does. In particular, in some circumstances it may decide to create constant forms, that will cause sharing of list structure at run time, or it may decide to create forms that will create new list structure at run time. For example, if the readers sees '(r . ,nil), it may produce the same thing as (cons 'r nil), or '(r . nil). Be careful that your program does not depend on which of these it does.

This is generally found to be pretty confusing by most people; the best way to explain further seems to be with examples. Here is how we would write our three simple macros using both the defmacro and backquote facilities.

```
(defmacro first (the-list)
    '(car ,the-list))

(defmacro addone (form)
   '(plus 1 ,form))

(defmacro increment (symbol)
   '(setq ,symbol (1+ ,symbol)))
```

To finally demonstrate how easy it is to define macros with these two facilities, here is the final form of the for macro.

```
(defmacro for (var lower upper . body)
  '(do ,var ,lower (1+ ,var) (> ,var ,upper) . ,body))
```

Look at how much simpler that is than the original definition. Also, look how closely it resembles the code it is producing. The functionality of the for really stands right out when written this way.

If a comma inside a backquote form is followed by an "atsign" character (@), it has a special meaning. The ",@" should be followed by a form whose value is a *list*; then each of the elements of the list is put into the list being created by the backquote. In other words, instead of generating a call to the cons function, backquote generates a call to append. For example, if a is bound to (x y z), then '(1 ,a 2) would evaluate to (1 (x y z) 2), but '(1 ,@a 2) would evaluate to (1 x y z 2).

Here is an example of a macro definition that uses the ",@" construction. On way to define do-forever would be for it to expand

```
(do-forever form1 form2 form3)
```

into

```
(prog ()
    a form1
      form2
      form3
      (go a))
```

You could define the macro by

```
(defmacro do-forever (&body body)
  `(prog ()
       a ,@body
         (go a)))
```

A similar construct is ",." (comma, dot). This means the same thing as ",@" except that the list which is the value of the following form may be modified destructively; backquote uses nconc rather than append. This should of course be used with caution.

Backquote does not make any guarantees about what parts of the structure it shares and what parts it copies. You should not do destructive operations such as nconc on the results of backquote forms such as

```
`(,a b c d)
```

since backquote might choose to implement this as

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

and nconc would smash the constant. On the other hand, it would be safe to nconc the result of

```
`(a b ,c ,d)
```

since there is nothing this could expand into that does not involve making a new list, such as

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

Backquote of course guarantees not to do any destructive operations (rplaca, rplacd, nconc) on the components of the structure it builds, unless the ",." syntax is used.

Advanced macro writers sometimes write "macro-defining macros": forms which expand into forms which, when evaluated, define macros. In such macros it is often useful to use nested backquote constructs. The following example illustrates the use of nested backquotes in the writing of macro-defining macros.

This example is a very simple version of defstruct (see page 300). You should first understand the basic description of defstruct before proceeding with this example. The defstruct below does not accept any options and allows only the simplest kind of items; that is, it allows only forms like

```
(defstruct (name)
     item1
     item2
     item3
     item4
     ...)
```

We would like this form to expand into

```
(progn 'compile
 (defmacro item1 (x)
      '(aref ,x 0))
 (defmacro item2 (x)
      '(aref ,x 1))
 (defmacro item3 (x)
      '(aref ,x 2))
 (defmacro item4 (x)
      '(aref ,x 3))
 ...)
```

(The meaning of the (progn 'compile ...) is discussed on page 260.) Here is the macro to perform the expansion:

```
(defmacro defstruct ((name) . items)
     (do ((item-list items (cdr item-list))
          (ans nil)
          (i 0 (1+ i)))
         ((null item-list)
          '(progn 'compile . ,(nreverse ans)))
       (setq ans
             (cons '(defmacro ,(car item-list) (x)
                              '(aref ,x ,',i))
                   ans))))
```

The interesting part of this definition is the body of the (inner) defmacro form:

```
'(aref ,x ,',i)
```

Instead of using this backquote construction, we could have written

```
(list 'aref x ,i)
```

That is, the ",'," acts like a comma that matches the outer backquote, while the "," preceding the "x" matches with the inner backquote. Thus, the symbol i is evaluated when the defstruct form is expanded, whereas the symbol x is evaluated when the accessor macros are expanded.

Backquote can be useful in situations other than the writing of macros. Whenever there is a piece of list structure to be consed up, most of which is constant, the use of backquote can make the program considerably clearer.

17.3 Substitutable Functions

A substitutable function is a function that is open coded by the compiler. It is like any other function when applied, but it can be expanded instead, and in that regard resembles a macro.

defsubst *Special Form*

defsubst is used for defining substitutable functions. It is used just like defun.

```
(defsubst name lambda-list . body)
```

and does almost the same thing. It defines a function that executes identically to the one that a similar call to defun would define. The difference comes when a function that *calls* this one is compiled. Then, the call is open-coded by substituting the substitutable function's definition into the code being compiled. The function itself looks like (named-subst *name lambda-list . body*). Such a function is called a subst. For example, if we define

```
(defsubst square (x) (* x x))
(defun foo (a b) (square (+ a b)))
```

then if foo is used interpreted, square will work just as if it had been defined by defun. If foo is compiled, however, the squaring will be substituted into it and it will compile just like

```
(defun foo (a b) (* (+ a b) (+ a b)))
```

square's definition would be

```
(named-subst square (x) (* x x))
```

(The internal formats of substs and named-substs are explained in section 10.5.1, page 161.)

A similar square could be defined as a macro, with

```
(defmacro square (x) '(* ,x ,x))
```

In general, anything that is implemented as a subst can be re-implemented as a macro, just by changing the defsubst to a defmacro and putting in the appropriate backquote and commas. The disadvantage of macros is that they are not functions, and so cannot be applied to arguments. Their advantage is that they can do much more powerful things than substs can. This is also a disadvantage since macros provide more ways to get into trouble. If something can be implemented either as a macro or as a subst, it is generally better to make it a subst.

The *lambda-list* of a subst may contain &optional and &rest, but no other lambda-list keywords. If there is a rest-argument, it is replaced in the body with an explicit call to list:

```
(defsubst append-to-foo (&rest args)
  (setq foo (append args foo)))

(append-to-foo x y z)
```

expands to

```
(setq foo (append (list x y z) foo))
```

Rest arguments in substs are most useful with lexpr-funcall, because of an optimization that is done:

```
(defsubst xhack (&rest indices)
  (lexpr-funcall 'xfun xarg1 indices))

(xhack a (car b))
```

is equivalent to

```
(xfun xarg1 a (car b))
```

If xfun is itself a **subst**, it will be expanded in turn.

When a **defsubst** is compiled, its list structure definition is kept around so that calls can still be open-coded by the compiler. But non-open-coded calls to the function run at the speed of compiled code. The interpreted definition is kept in the compiled definition's debugging info alist (see page 172). Undeclared free variables used in a **defsubst** being compiled do not get any warning, because this is a common practice that works properly with nonspecial variables when calls are open coded.

If you are using a **defsubst** from outside the program to which it belongs, you might sometimes be better off if it is not open-coded. The decrease in speed might not be significant, and you would have the advantage that you would not need to recompile your program if the definition is changed. You can prevent open-coding by putting **dont-optimize** around the call to the **defsubst**.

```
(dont-optimize (xhack a (car b)))
```

See page 233.

You will notice that the substitution performed is very simple and takes no care about the possibility of computing an argument twice when it really ought to be computed once. For instance, in the current implementation, the functions

```
(defsubst reverse-cons (x y) (cons y x))
(defsubst in-order (a b c) (and (< a b) (< b c)))
```

would present problems. When compiled, because of the substitution a call to **reverse-cons** would evaluate its arguments in the wrong order, and a call to **in-order** could evaluate its second argument twice. This will be fixed at some point in the future, but for now the writer of **defsubst**'s must be cautious. Also all occurrences of the argument names in the body are replaced with the argument forms, wherever they appear. Thus an argument name should not be used in the body for anything else, such as a function name or a symbol in a constant.

As with **defun**, *name* can be any function spec.

17.4 Hints to Macro Writers

There are many useful techniques for writing macros. Over the years, Lisp programmers have discovered techniques that most programmers find useful, and have identified pitfalls that must be avoided. This section discusses some of these techniques and illustrates them with examples.

The most important thing to keep in mind as you learn to write macros is that the first thing you should do is figure out what the macro form is supposed to expand into, and only then should you start to actually write the code of the macro. If you have a firm grasp of what the generated Lisp program is supposed to look like, you will find the macro much easier to write.

In general any macro that can be written as a substitutable function (see page 255) should be written as one, not as a macro, for several reasons: substitutable functions are easier to write and to read; they can be passed as functional arguments (for example, you can pass them to mapcar); and there are some subtleties that can occur in macro definitions that need not be worried about in substitutable functions. A macro can be a substitutable function only if it has exactly the semantics of a function, rather than of a special form. The macros we will see in this section are not semantically like functions; they must be written as macros.

17.4.1 Name Conflicts

One of the most common errors in writing macros is best illustrated by example. Suppose we wanted to write dolist (see page 49) as a macro that expanded into a do (see page 45). The first step, as always, is to figure out what the expansion should look like. Let's pick a representative example form, and figure out what its expansion should be. Here is a typical dolist form.

```
(dolist (element (append a b))
  (push element *big-list*)
  (foo element 3))
```

We want to create a do form that does the thing that the above dolist form says to do. That is the basic goal of the macro: it must expand into code that does the same thing that the original code says to do, but it should be in terms of existing Lisp constructs. The do form might look like this:

```
(do ((list (append a b) (cdr list))
     (element))
    ((null list))
  (setq element (car list))
  (push element *big-list*)
  (foo element 3))
```

Now we could start writing the macro that would generate this code, and in general convert any dolist into a do, in an analogous way. However, there is a problem with the above scheme for expanding the dolist. The above expansion works fine. But what if the input form had been the following:

```
(dolist (list (append a b))
  (push list *big-list*)
  (foo list 3))
```

This is just like the form we saw above, except that the programmer happened to decide to name the looping variable list rather than element. The corresponding expansion would be:

```
(do ((list (append a b) (cdr list))
     (list))
    ((null list))
  (setq list (car list))
  (push list *big-list*)
  (foo list 3))
```

This doesn't work at all! In fact, this is not even a valid program, since it contains a do that uses the same variable in two different iteration clauses.

Here's another example that causes trouble:

```
(let ((list nil))
  (dolist (element (append a b))
    (push element list)
    (foo list 3)))
```

If you work out the expansion of this form, you will see that there are two variables named list, and that the programmer meant to refer to the outer one but the generated code for the push actually uses the inner one.

The problem here is an accidental name conflict. This can happen in any macro that has to create a new variable. If that variable ever appears in a context in which user code might access it, then you have to worry that it might conflict with some other name that the user is using for his own program.

One way to avoid this problem is to choose a name that is very unlikely to be picked by the user, simply by choosing an unusual name. This will probably work, but it is inelegant since there is no guarantee that the user won't just happen to choose the same name. The way to really avoid the name conflict is to use an uninterned symbol as the variable in the generated code. The function gensym (see page 101) is useful for creating such symbols.

Here is the expansion of the original form, using an uninterned symbol created by gensym.

```
(do ((g0005 (append a b) (cdr g0005))
     (element))
    ((null g0005))
  (setq element (car g0005))
  (push element *big-list*)
  (foo element 3))
```

This is the right kind of thing to expand into. Now that we understand how the expansion works, we are ready to actually write the macro. Here it is:

```
(defmacro dolist ((var form) . body)
  (let ((dummy (gensym)))
    '(do ((,dummy ,form (cdr ,dummy))
          (,var))
         ((null ,dummy))
       (setq ,var (car ,dummy))
       . ,body)))
```

Many system macros do not use **gensym** for the internal variables in their expansions. Instead they use symbols whose print names begin and end with a dot. This provides meaningful names for these variables when looking at the generated code and when looking at the state of a computation in the error-handler. However, this convention means that users should avoid naming variables this way.

17.4.2 prog-context Conflicts

A related problem occurs when you write a macro that expands into a **prog** (or a **do**, or something that expands into **prog** or **do**) behind the user's back (unlike **dolist**, which is documented to be like **do**). Consider the **error-restart** special form (see page 577). Suppose we wanted to implement it as a macro that expands into a **prog**. If it expanded into a plain old **prog**, then the following (contrived) Lisp program would not behave correctly:

```
(prog ()
   (setq a 3)
   (error-restart ((sys:abort error) "Return from FOO.")
     (cond ((> a 10)
            (return 5))
           ((> a 4)
            (ferror 'lose "You lose."))))
   (setq b 7))
```

The problem is that the **return** would return from the **error-restart** instead of the **prog**. The way to avoid this problem is to use a named **prog** whose name is **t**. The name **t** is special in that it is invisible to the **return** function. If we write **error-restart** as a macro that expands into a **prog** named **t**, then the **return** will pass right through the **error-restart** form and return from the **prog**, as it ought to.

When **error-restart**'s expansion is supposed to return from the **prog** named **t**, it uses **return-from t**.

In general, when a macro expands into a **prog** or a **do** around the user's code, the **prog** or **do** should be named **t** so that **return** forms in the user code will return to the right place, unless the macro is documented as generating a **prog**/**do**-like form which may be exited with **return**.

17.4.3 Macros Expanding into Many Forms

Sometimes a macro wants to do several different things when its expansion is evaluated. Another way to say this is that sometimes a macro wants to expand into several things, all of which should happen sequentially at run time (not macro-expand time). For example, suppose you wanted to implement defconst (see page 20) as a macro. defconst must do two things, declare the variable to be special and set the variable to its initial value. (We will implement a simplified defconst that does only these two things, and doesn't have any options.) What should a defconst form expand into? Well, what we would like is for an appearance of

```
(defconst a (+ 4 b))
```

in a file to be the same thing as the appearance of the following two forms:

```
(declare (special a))
(setq a (+ 4 b))
```

However, because of the way that macros work, they only expand into one form, not two. So we need to have a defconst form expand into one form that is just like having two forms in the file.

There is such a form. It looks like this:

```
(progn 'compile
       (declare (special a))
       (setq a (+ 4 b)))
```

In interpreted Lisp, it is easy to see what happens here. This is a progn special form, and so all its subforms are evaluated, in turn. First the form 'compile is evaluated. The result is the symbol compile; this value is not used, and evaluation of 'compile has no side-effects, so the 'compile subform is effectively ignored. Then the declare form and the setq form are evaluated. So far, so good.

The interesting thing is the way this form is treated by the compiler. The compiler specially recognizes any progn form at top level in a file whose first subform is 'compile. When it sees such a form, it processes each of the remaining subforms of the progn just as if that form had appeared at top level in the file. So the compiler behaves exactly as if it had encountered the declare form at top level, and then encountered the setq form at top level, even though neither of those forms was actually at top-level (they were both inside the progn). This feature of the compiler is provided specifically for the benefit of macros that want to expand into several things.

Here is the macro definition:

```
(defmacro defconst (variable init-form)
  '(progn 'compile
          (declare (special ,variable))
          (setq ,variable ,init-form)))
```

Here is another example of a form that wants to expand into several things. We will implement a special form called define-command, which is intended to be used in order to define commands in some interactive user subsystem. For each command, there are two things provided by the define-command form: a function that executes the command, and a text string that contains the documentation for the command (in order to provide an on-line interactive documentation feature). This macro is a simplified version of a macro that is actually used in the Zwei editor. Suppose that in this subsystem, commands are always functions of no arguments, documentation strings are placed on the help property of the name of the command, and the

names of all commands are put onto a list. A typical call to define-command would look like:

```
(define-command move-to-top
   "This command moves you to the top."
   (do ()
       ((at-the-top-p))
     (move-up-one)))
```

This could expand into:

```
(progn 'compile
       (defprop
         move-to-top
         "This command moves you to the top."
         help)
       (push 'move-to-top *command-name-list*)
       (defun move-to-top ()
         (do ()
             ((at-the-top-p))
           (move-up-one)))
       )
```

The define-command expands into three forms. The first one sets up the documentation string and the second one puts the command name onto the list of all command names. The third one is the defun that actually defines the function itself. Note that the defprop and push happen at load-time (when the file is loaded); the function, of course, also gets defined at load time. (See the description of eval-when (page 231) for more discussion of the differences between compile time, load time, and eval time.)

This technique makes Lisp a powerful language in which to implement your own language. When you write a large system in Lisp, frequently you can make things much more convenient and clear by using macros to extend Lisp into a customized language for your application. In the above example, we have created a little language extension: a new special form that defines commands for our system. It lets the writer of the system put his documentation strings right next to the code that they document, so that the two can be updated and maintained together. The way that the Lisp environment works, with load-time evaluation able to build data structures, lets the documentation data base and the list of commands be constructed automatically.

17.4.4 Macros that Surround Code

There is a particular kind of macro that is very useful for many applications. This is a macro that you place "around" some Lisp code, in order to make the evaluation of that code happen in some context. For a very simple example, we could define a macro called with-output-in-base, that executes the forms within its body with any output of numbers that is done defaulting to a specified base.

```
(defmacro with-output-in-base ((base-form) &body body)
   `(let ((base ,base-form))
      . ,body))
```

A typical use of this macro might look like:

```
(with-output-in-base (*default-base*)
   (print x)
   (print y))
```

which would expand into

```
(let ((base *default-base*))
  (print x)
  (print y))
```

This example is too trivial to be very useful; it is intended to demonstrate some stylistic issues. There are some special forms in Zetalisp that are similar to this macro; see with-open-file (page 431) and with-input-from-string (page 150), for example. The really interesting thing, of course, is that you can define your own such special forms for your own specialized applications. One very powerful application of this technique was used in a system that manipulates and solves the Rubik's cube puzzle. The system heavily uses a special form called with-front-and-top, whose meaning is "evaluate this code in a context in which this specified face of the cube is considered the front face, and this other specified face is considered the top face".

The first thing to keep in mind when you write this sort of macro is that you can make your macro much clearer to people who might read your program if you conform to a set of loose standards of syntactic style. By convention, the names of such special forms start with "with-". This seems to be a clear way of expressing the concept that we are setting up a context; the meaning of the special form is "do this stuff *with* the following things true". Another convention is that any "parameters" to the special form should appear in a list that is the first subform of the special form, and that the rest of the subforms should make up a body of forms that are evaluated sequentially with the last one returned. All of the examples cited above work this way. In our with-output-in-base example, there was one parameter (the base), which appears as the first (and only) element of a list that is the first subform of the special form. The extra level of parentheses in the printed representation serves to separate the "parameter" forms from the "body" forms so that it is textually apparent which is which; it also provides a convenient way to provide default parameters (a good example is the with-input-from-string special form (page 150), which takes two required and two optional "parameters"). Another convention/technique is to use the &body keyword in the defmacro to tell the editor how to correctly indent the special form (see page 268).

The other thing to keep in mind is that control can leave the special form either by the last form's returning, or by a non-local exit (that is, something doing a *throw). You should write the special form in such a way that everything will be cleaned up appropriately no matter which way control exits. In our with-output-in-base example, there is no problem, because non-local exits undo lambda-bindings. However, in even slightly more complicated cases, an unwind-protect form (see page 56) is needed: the macro must expand into an unwind-protect that surrounds the body, with "cleanup" forms that undo the context-setting-up that the macro did. For example, using-resource (see page 94) is implemented as a macro that does an allocate-resource and then performs the body inside of an unwind-protect that has a deallocate-resource in its "cleanup" forms. This way the allocated resource item will be deallocated whenever control leaves the using-resource special form.

17.4.5 Multiple and Out-of-order Evaluation

In any macro, you should always pay attention to the problem of multiple or out-of-order evaluation of user subforms. Here is an example of a macro with such a problem. This macro defines a special form with two subforms. The first is a reference, and the second is a form. The special form is defined to create a cons whose car and cdr are both the value of the second subform, and then to set the reference to be that cons. Here is a possible definition:

```
(defmacro test (reference form)
   `(setf ,reference (cons ,form ,form)))
```

Simple cases will work all right:

```
(test foo 3) ==>
  (setf foo (cons 3 3))
```

But a more complex example, in which the subform has side effects, can produce surprising results:

```
(test foo (setq x (1+ x))) ==>
  (setf foo (cons (setq x (1+ x))
                  (setq x (1+ x))))
```

The resulting code evaluates the **setq** form twice, and so **x** is increased by two instead of by one. A better definition of **test** that avoids this problem is:

```
(defmacro test (reference form)
   (let ((value (gensym)))
     `(let ((,value ,form))
        (setf ,reference (cons ,value ,value)))))
```

With this definition, the expansion works as follows:

```
(test foo (setq x (1+ x))) ==>
  (let ((g0005 (setq x (1+ x))))
    (setf foo (cons g0005 g0005)))
```

In general, when you define a new special form that has some forms as its subforms, you have to be careful about just when those forms get evaluated. If you aren't careful, they can get evaluated more than once, or in an unexpected order, and this can be semantically significant if the forms have side-effects. There's nothing fundamentally wrong with multiple or out-of-order evalation if that is really what you want and if it is what you document your special form to do. However, it is very common for special forms to simply behave like functions, and when they are doing things like what functions do, it's natural to expect them to be function-like in the evaluation of their subforms. Function forms have their subforms evaluated, each only once, in left-to-right order, and special forms that are similar to function forms should try to work that way too for clarity and consistency.

There is a tool that makes it easier for you to follow the principle explained above. It is a macro called **once-only**. It is most easily explained by example. You would write **test** using **once-only** as follows:

```
(defmacro test (reference form)
  (once-only (form)
    `(setf ,reference (cons ,form ,form))))
```

This defines **test** in such a way that the **form** is only evaluated once, and references to **form** inside the macro body refer to that value. **once-only** automatically introduces a lambda-binding of a generated symbol to hold the value of the form. Actually, it is more clever than that; it avoids introducing the lambda-binding for forms whose evaluation is trivial and may be repeated

without harm or cost, such as numbers, symbols, and quoted structure. This is just an optimization that helps produce more efficient code.

The once-only macro makes it easier to follow the principle, but it does not completely or automatically solve the problems of multiple and out-of-order evaluation. It is just a tool that can solve some of the problems some of the time; it is not a panacea.

The following description attempts to explain what once-only does, but it is a lot easier to use once-only by imitating the example above than by trying to understand once-only's rather tricky definition.

once-only *Macro*

A once-only form looks like

```
(once-only var-list
  form1
  form2
  ...)
```

var-list is a list of variables. The *forms* are a Lisp program that presumably uses the values of those variables. When the form resulting from the expansion of the once-only is evaluated, the first thing it does is to inspect the values of each of the variables in *var-list*; these values are assumed to be Lisp forms. For each of the variables, it binds that variable either to its current value, if the current value is a trivial form, or to a generated symbol. Next, once-only evaluates the *forms* in this new binding environment and, when they have been evaluated, it undoes the bindings. The result of the evaluation of the last *form* is presumed to be a Lisp form, typically the expansion of a macro. If all of the variables have been bound to trivial forms, then *once-only* just returns that result. Otherwise, once-only returns the result wrapped in a lambda-combination that binds the generated symbols to the result of evaluating the respective non-trivial forms.

The effect is that the program produced by evaluating the once-only form is coded in such a way that, each of the forms which was the value of one of the variables in *var-list* will be evaluated only once, unless the form is such as to have no side effects. At the same time, no unnecessary temporary variables appear in the generated code, but the body of the once-only is not cluttered up with extraneous code to decide whether temporary variables are needed.

Caution! A number of system macros, setf for example, fail to follow this convention. Unexpected multiple evaluation and out-of-order evaluation can occur with them. This was done for the sake of efficiency, is prominently mentioned in the documentation of these macros, and will be fixed in the future. It would be best not to compromise the semantic simplicity of your own macros in this way.

17.4.6 Nesting Macros

A useful technique for building language extensions is to define programming constructs that employ two special forms, one of which is used inside the body of the other. Here is a simple example. There are two special forms. The outer one is called with-collection, and the inner one is called collect. collect takes one subform, which it evaluates; with-collection just has a body, whose forms it evaluates sequentially. with-collection returns a list of all of the values that were given to collect during the evaluation of the with-collection's body. For example,

```
(with-collection
  (dotimes (i 5)
    (collect i)))

=> (1 2 3 4 5)
```

Remembering the first piece of advice we gave about macros, the next thing to do is to figure out what the expansion looks like. Here is how the above example could expand:

```
(let ((g0005 nil))
  (dotimes (i 5)
     (push i g0005))
  (nreverse g0005))
```

Now, how do we write the definition of the macros? Well, with-collection is pretty easy:

```
(defmacro with-collection (&body body)
  (let ((var (gensym)))
     `(let ((,var nil))
        ,@body
        (nreverse ,var)))))
```

The hard part is writing collect. Let's try it:

```
(defmacro collect (argument)
  `(push ,argument ,var))
```

Note that something unusual is going on here: collect is using the variable var freely. It is depending on the binding that takes place in the body of with-collection in order to get access to the value of var. Unfortunately, that binding took place when with-collection got expanded; with-collection's expander function bound var, and it got unbound when the expander function was done. By the time the collect form gets expanded, var has long since been unbound. The macro definitions above do not work. Somehow the expander function of with-collection has to communicate with the expander function of collect to pass over the generated symbol.

The only way for with-collection to convey information to the expander function of collect is for it to expand into something that passes that information. What we can do is to define a special variable (which we will call *collect-variable*), and have with-collection expand into a form that binds this variable to the name of the variable that the collect should use. Now, consider how this works in the interpreter. The evaluator first sees the with-collection form and calls the expander function to expand it. The expander function creates the expansion and returns to the evaluator, which then evaluates the expansion. The expansion includes in it a let form to bind *collect-variable* to the generated symbol. When the evaluator sees this let form during the evaluation of the expansion of the with-collection form, it sets up the binding and recursively evaluates the body of the let. Now, during the evaluation of the body of the let, our special variable is bound, and if the expander function of collect gets run, it is able to see the value of collection-variable and incorporate the generated symbol into its own expansion.

Writing the macros this way is not quite right. It works fine interpreted, but the problem is that it does not work when we try to compile Lisp code that uses these special forms. When code is being compiled, there isn't any interpreter to do the binding in our new **let** form; macro expansion is done at compile time, but generated code does not get run until the results of the compilation are loaded and run. The way to fix our definitions is to use **compiler-let** instead of **let**. **compiler-let** (see page 238) is a special form that exists specifically to do the sort of thing we are trying to do here. **compiler-let** is identical to **let** as far as the interpreter is concerned, so changing our **let** to a **compiler-let** won't affect the behavior in the interpreter; it will continue to work. When the compiler encounters a **compiler-let**, however, it actually performs the bindings that the **compiler-let** specifies and proceeds to compile the body of the **compiler-let** with all of those bindings in effect. In other words, it acts as the interpreter would.

Here's the right way to write these macros:

```
(defvar *collect-variable*)

(defmacro with-collection (&body body)
  (let ((var (gensym)))
     '(let ((,var nil))
        (compiler-let ((*collect-variable* ',var))
          . ,body)
        (nreverse ,var))))

(defmacro collect (argument)
  '(push ,argument ,*collect-variable*))
```

17.4.7 Functions Used During Expansion

The technique of defining functions to be used during macro expansion deserves explicit mention here. It may not occur to you, but a macro expander function is a Lisp program like any other Lisp program, and it can benefit in all the usual ways by being broken down into a collection of functions that do various parts of its work. Usually macro expander functions are pretty simple Lisp programs that take things apart and put them together slightly differently, but some macros are quite complex and do a lot of work. Several features of Zetalisp, including flavors, **loop**, and **defstruct**, are implemented using very complex macros, which, like any complex well-written Lisp program, are broken down into modular functions. You should keep this in mind if you ever invent an advanced language extension or ever find yourself writing a five-page expander function.

A particular thing to note is that any functions used by macro-expander functions must be available at compile-time. You can make a function available at compile time by surrounding its defining form with an **(eval-when (compile load eval) ...)**; see page 231 for more details. Doing this means that at compile time the definition of the function will be interpreted, not compiled, and hence will run more slowly. Another approach is to separate macro definitions and the functions they call during expansion into a separate file, often called a "defs" (definitions) file. This file defines all the macros but does not use any of them. It can be separately compiled and loaded up before compiling the main part of the program, which uses the macros. The *system* facility (see chapter 25, page 520) helps keep these various files straight, compiling and loading things in the right order.

17.5 Aids for Debugging Macros

mexp

mexp goes into a loop in which it reads forms and sequentially expands them, printing out the result of each expansion (using the grinder (see page 426) to improve readability). It terminates when it reads an atom (anything that is not a cons). If you type in a form that is not a macro form, there will be no expansions and so it will not type anything out, but just prompt you for another form. This allows you to see what your macros are expanding into, without actually evaluating the result of the expansion.

17.6 Displacing Macros

Every time the the evaluator sees a macro form, it must call the macro to expand the form. If this expansion always happens the same way, then it is wasteful to expand the whole form every time it is reached; why not just expand it once? A macro is passed the macro form itself, and so it can change the car and cdr of the form to something else by using rplaca and rplacd! This way the first time the macro is expanded, the expansion will be put where the macro form used to be, and the next time that form is seen, it will already be expanded. A macro that does this is called a *displacing macro*, since it displaces the macro form with its expansion.

The major problem with this is that the Lisp form gets changed by its evaluation. If you were to write a program which used such a macro, call grindef to look at it, then run the program and call grindef again, you would see the expanded macro the second time. Presumably the reason the macro is there at all is that it makes the program look nicer; we would like to prevent the unnecessary expansions, but still let grindef display the program in its more attractive form. This is done with the function displace.

Anothing thing to worry about with displacing macros is that if you change the definition of a displacing macro, then your new definition will not take effect in any form that has already been displaced. If you redefine a displacing macro, an existing form using the macro will use the new definition only if the form has never been evaluated.

displace *form expansion*

form must be a list. displace replaces the car and cdr of *form* so that it looks like:

```
(si:displaced original-form expansion)
```

original-form is equal to *form* but has a different top-level cons so that the replacing mentioned above doesn't affect it. si:displaced is a macro, which returns the caddr of its own macro form. So when the si:displaced form is given to the evaluator, it "expands" to *expansion*. displace returns *expansion*.

The grinder knows specially about si:displaced forms, and will grind such a form as if it had seen the original form instead of the si:displaced form.

So if we wanted to rewrite our addone macro as a displacing macro, instead of writing

```
(macro addone (x)
   (list 'plus '1 (cadr x)))
```

we would write

```
(macro addone (x)
  (displace x (list 'plus '1 (cadr x))))
```

Of course, we really want to use defmacro to define most macros. Since there is no way to get at the original macro form itself from inside the body of a defmacro, another version of it is provided:

defmacro-displace *Macro*

defmacro-displace is just like defmacro except that it defines a displacing macro, using the displace function.

Now we can write the displacing version of addone as

```
(defmacro-displace addone (val)
  (list 'plus '1 val))
```

All we have done in this example is change the defmacro into defmacro-displace. addone is now a displacing macro.

17.7 Advanced Features of Defmacro

The pattern in a defmacro is more like the lambda-list of a normal function than revealed above. It is allowed to contain certain &-keywords.

&optional is followed by *variable*, (*variable*), (*variable default*), or (*variable default present-p*), exactly the same as in a function. Note that *default* is still a form to be evaluated, even though *variable* is not being bound to the value of a form. *variable* does not have to be a symbol; it can be a pattern. In this case the first form is disallowed because it is syntactically ambigous. The pattern must be enclosed in a singleton list. If *variable* is a pattern, *default* can be evaluated more than once.

Using &rest is the same as using a dotted list as the pattern, except that it may be easier to read and leaves a place to put &aux.

&aux is the same in a macro as in a function, and has nothing to do with pattern matching.

defmacro has a couple of additional keywords not allowed in functions.

&body is identical to &rest except that it informs the editor and the grinder that the remaining subforms constitute a "body" rather than "arguments" and should be indented accordingly.

&list-of *pattern* requires that the corresponding position of the form being translated must contain a list (or nil). It matches *pattern* against each element of that list. Each variable in *pattern* is bound to a list of the corresponding values in each element of the list matched by the &list-of. This may be clarified by an example. Suppose we want to be able to say things like

```
(send-commands (aref turtle-table i)
  (forward 100)
  (beep)
  (left 90)
  (pen 'down 'red)
  (forward 50)
  (pen 'up))
```

We could define a send-commands macro as follows:

```
(defmacro send-commands (object
                         &body &list-of (command . arguments))
  '(let ((o ,object))
     . ,(mapcar #'(lambda (com args) '(send o ',com . ,args))
                command arguments)))
```

Note that this example uses &body together with &list-of, so you don't see the list itself; the list is just the rest of the macro-form.

You can combine &optional and &list-of. Consider the following example:

```
(defmacro print-let (x &optional &list-of ((vars vals)
                                           '((base 10.)
                                             (*nopoint t))))
  '((lambda (,@vars) (print ,x))
    ,@vals))

(print-let foo)  ==>
((lambda (base *nopoint)
   (print foo))
 12
 t)

(print-let foo ((bar 3)))  ==>
((lambda (bar)
   (print foo))
 3)
```

In this example we aren't using &body or anything like it, so you do see the list itself; that is why you see parentheses around the (bar 3).

17.8 Functions to Expand Macros

The following two functions are provided to allow the user to control expansion of macros; they are often useful for the writer of advanced macro systems, and in tools that want to examine and understand code that may contain macros.

macroexpand-1 *form*

If *form* is a macro form, this expands it (once) and returns the expanded form. Otherwise it just returns *form*. macroexpand-1 expands defsubst function forms as well as macro forms.

macroexpand *form*

If *form* is a macro form, this expands it repeatedly until it is not a macro form and returns the final expansion. Otherwise, it just returns *form*. macroexpand expands defsubst function forms as well as macro forms.

17.9 Generalized Variables

In Lisp, a variable is something that can remember one piece of data. The main operations on a variable are to recover that piece of data and to change it. These might be called *access* and *update*. The concept of variables named by symbols, explained in section 3.1, page 15, can be generalized to any storage location that can remember one piece of data, no matter how that location is named.

For each kind of generalized variable, there are typically two functions which implement the conceptual *access* and *update* operations. For example, symeval accesses a symbol's value cell, and set updates it. array-leader accesses the contents of an array leader element, and store-array-leader updates it. car accesses the car of a cons, and rplaca updates it.

Rather than thinking of this as two functions, which operate on a storage location somehow deduced from their arguments, we can shift our point of view and think of the access function as a *name* for the storage location. Thus (symeval 'foo) is a name for the value of foo, and (aref a 105) is a name for the 105th element of the array a. Rather than having to remember the update function associated with each access function, we adopt a uniform way of updating storage locations named in this way, using the setf special form. This is analogous to the way we use the setq special form to convert the name of a variable (which is also a form which accesses it) into a form that updates it.

setf is particularly useful in combination with structure-accessing macros, such as those created with defstruct, because the knowledge of the representation of the structure is embedded inside the macro, and the programmer shouldn't have to know what it is in order to alter an element of the structure.

setf is actually a macro which expands into the appropriate update function. It has a database, explained below, that associates from access functions to update functions.

setf *access-form value* *Macro*

setf takes a form that *accesses* something and "inverts" the form to produce a corresponding form to *update* the thing. A setf expands into an update form, which stores the result of evaluating the form *value* into the place referenced by the *access-form*. Examples:

```
(setf (array-leader foo 3) 'bar)
                ==> (store-array-leader 'bar foo 3)
(setf a 3) ==> (setq a 3)
(setf (plist 'a) '(foo bar)) ==> (setplist 'a '(foo bar))
(setf (aref q 2) 56) ==> (aset 56 q 2)
(setf (cadr w) x) ==> (rplaca (cdr w) x)
```

If *access-form* invokes a macro or a substitutable function, then setf expands the *access-form* and starts over again. This lets you use setf together with defstruct accessor macros.

For the sake of efficiency, the code produced by setf does not preserve order of evaluation of the argument forms. This is only a problem if the argument forms have interacting side-effects. For example, if you evaluate

```
(setq x 3)
(setf (aref a x) (setq x 4))
```

then the form might set element 3 or element 4 of the array. We do not guarantee which one it will do; don't just try it and see and then depend on it, because it is subject to change without notice.

Furthermore, the value produced by setf depends on the structure type and is not guaranteed; setf should be used for side effect only.

Besides the *access* and *update* conceptual operations on variables, there is a third basic operation, which we might call *locate*. Given the name of a storage cell, the *locate* operation will return the address of that cell as a locative pointer (see chapter 13, page 197). This locative pointer is a first-class Lisp data object which is a kind of reference to the cell. It can be passed as an argument to a function which operates on any kind of variable, regardless of how it is named. It can be used to *bind* the variable, using the bind subprimitive (see page 212).

Of course, this can work only on variables whose implementation is really to store their value in a memory cell. A variable with an *update* operation that encrypts the value and an *access* operation that decrypts it could not have the *locate* operation, since the value per se is not actually stored anywhere.

locf *access-form* *Macro*

locf takes a form that *accesses* some cell, and produces a corresponding form to create a locative pointer to that cell.

Examples:

```
(locf (array-leader foo 3)) ==> (ap-leader foo 3)
(locf a) ==> (value-cell-location 'a)
(locf (plist 'a)) ==> (property-cell-location 'a)
(locf (aref q 2)) ==> (aloc q 2)
```

If *access-form* invokes a macro or a substitutable function, then locf expands the *access-form* and starts over again. This lets you use locf together with defstruct accessor macros.

Both setf and locf work by means of property lists. When the form (setf (aref q 2) 56) is expanded, setf looks for the setf property of the symbol aref. The value of the setf property of a symbol should be a cons whose car is a pattern to be matched with the *access-form* and whose cdr is the corresponding *update-form*, with the symbol si:val in place of the value to be stored. The setf property of aref is a cons whose car is (aref array . subscripts) and whose cdr is (aset si:val array . subscripts). If the transformation that setf is to do cannot be expressed as a simple pattern, an arbitrary function may be used: when the form (setf (foo bar) baz) is being expanded, if the setf property of foo is a symbol, the function definition of that symbol will be applied to two arguments, (foo bar) and baz, and the result will be taken to be the expansion of the setf.

Similarly, the locf function uses the locf property, whose value is analogous. For example, the locf property of aref is a cons whose car is (aref array . subscripts) and whose cdr is (aloc array . subscripts). There is no si:val in the case of locf.

Both setf and locf use getdecl to look for the property, so you can define the property with defdecl and it will be available for use at compile time in the rest of the same file.

unknown-setf-reference (error) *Condition*
unknown-locf-reference (error) *Condition*

These are signaled when setf or locf does not know how to expand the *access-form*. The :form operation on the condition instance returns the *access-form*.

incf *access-form* [*amount*] *Macro*

Increments the value of a generalized variable. (incf *ref*) increments the value of *ref* by 1. (incf *ref amount*) adds *amount* to *ref* and stores the sum back into *ref*.

incf expands into a setf form, so *ref* can be anything that setf understands as its *access-form*. This also means that you should not depend on the returned value of an incf form.

You must take great care with incf because it may evaluate parts of *ref* more than once. For example,

```
(incf (car (mumble))) ==>
(setf (car (mumble)) (1+ (car (mumble)))) ==>
(rplaca (mumble) (1+ (car (mumble))))
```

The mumble function is called more than once, whiuch may be significantly inefficient if mumble is expensive and may be downright wrong if mumble has side-effects. The same problem can come up with the decf, push, and pop macros (see below).

decf *access-form* [*amount*] *Macro*

Decrements the value of a generalized variable. (decf *ref*) decrements the value of *ref* by 1. (decf *ref amount*) subtracts *amount* from *ref* and stores the difference back into *ref*.

decf expands into a setf form, so *ref* can be anything that setf understands as its *access-form*. This also means that you should not depend on the returned value of a decf form.

push *item access-form* *Macro*

Adds an item to the front of a list that is stored in a generalized variable. (push *item ref*) creates a new cons whose car is the result of evaluating *item* and whose cdr is the contents of *ref*, and stores the new cons into *ref*.

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.

All the caveats that apply to incf apply to push as well: forms within *ref* may be evaluated more than once. The returned value of push is not defined.

pop *access-form* *Macro*

Removes an element from the front of a list that is stored in a generalized variable. (pop *ref*) finds the cons in *ref*, stores the cdr of the cons back into *ref*, and returns the car of the cons.

Example:

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

All the caveats that apply to incf apply to pop as well: forms within *ref* may be evaluated more than once.