18. Macros

18.1 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 two arguments. 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 ignore)
       (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 arguments the original form that eval was handed, and an environment data structure that this macro does not use. So it calls (lambda (x ignore) (list 'car (cadr x))) and the first argument is (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 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 (form-arg env-arg)
    body)
```

name can be any function spec. *form-arg* and *env-arg* must be variables. *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 ignore)
    (list 'car (cadr x)))
```

Only sophisticated macros need to use value passed for the *env-arg*; this one does not need it, so the argument variable ignore is used for it. See page 324 for information on it.

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 ignore)
   (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 ignore)
    (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 36); 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 present in a call to it; 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 ignore)
  (list* 'do
         (list (list (second x) (third x)
                     (list '1+ (second x))))
         (list (list '> (second x) (fourth x)))
         (cddddr x)))
```

for can now be used as if it were a built-in Lisp control construct.

18.2 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 (second 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.

18.2.1 Defmacro

Instead of referring to the parts of our form by (second x) and such, we would like to give names to the various pieces of the form, and somehow have the (second 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)
  (list* 'do
         (list (list var lower (list '1+ var)))
         (list (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 is 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))). var, lower, upper, and body are then used by the body of the macro definition.

defmacro *Macro*

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

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

name is the name of the macro to be defined; it can be any function spec (see section 11.2, page 223). Normally it is not useful to define anything but a symbol, since that is the only place that the evaluator looks for macro definitions. However, sometimes it is useful to define a :property function spec as a macro, when some part of the system (such as locf) will look for an expander function on a property.

The *pattern* may be anything made up out of symbols and conses. When the macro is called, *pattern* 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*. *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.

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. Subpatterns of the lambda list pattern can also use &-keywords, a usage not allowed in functions.

&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 at least be enclosed in a singleton list. If *variable* is a pattern, *default* can be evaluated more than once. Example:

```
(defmacro foo (&optional ((x &optional y) '(a)))
  ...)
```

Here the first argument of foo is optional, and should be a list of one or two elements which become x and y. If foo is given no arguments, the list (a) is decomposed to get x and y, so that x's value is a and y's value is nil.

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.

When &key is used in a defmacro pattern, the keywords are decoded at macro expansion time. Therefore, they must be constants. Example:

```
(defmacro l1 (&key a b c)
  (list 'list a b c))

(l1 :b 5 :c (car d))
  ==> (list nil 5 (car d))
```

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

defmacro implements a few 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 ordinary arguments and should be indented accordingly. Example:

```
(defmacro with-open-file
          ((streamvar filename &rest options)
           &body body)
  ...)
```

&whole causes the variable that follows it to be bound to the entire macro call, just as the *form-arg* variable in macro would be. &whole exists to make defmacro able to do anything that macro can be used for, for the sake of Common Lisp, in which defmacro is the primitive and macro does not exist. &whole is also useful in macrolet.

&environment causes the variable that follows it to be bound to the *local macros environment* of the macro call being expanded. This is useful if the code for expanding this macro needs to invoke macroexpand on subforms of the macro call. Then, to achieve correct interaction with macrolet, this local macros environment should be passed to macroexpand as its second argument.

&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)
                        '((*print-base* 10.)
                          (*print-radix* nil))))
  '((lambda (,@vars) (print ,x))
    ,@vals))

(print-let foo)  ==>
((lambda (*print-base* *print-radix*)
   (print foo))
 10 nil)

(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).

18.2.2 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 backquote and comma make it easier to type forms that create new list structure. They allow 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 list or vector. If the comma character does not appear within the text for the list or vector, the backquote acts just like a single quote: it creates a form which, when evaluated, produces the list or vector specified. For example,

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

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))
'#(a ,b) => #(a 1)
```

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

The list or vector following a backquote can be thought of as a template for some new data structure. The parts of it that are preceded by commas are forms that fill in slots in the template; everything else is just constant structure that appears as written 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, which 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 reader 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 demonstrate finally 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 at-sign 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. One way to define do-forever would be for it to expand

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

into

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

You could define the macro by

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

(This definition has the disadvantage of interfering with use of the go tag a to go from the body of the do-forever to a tag defined outside of it. A more robust implementation would construct a new tag each time, using gensym.)

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 any possible expansion of this would make a new list. One possible expansion is

```
(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. For example, here is a very simple version of defstruct (see page 374) which does not allow any options and only the simplest slot descriptors. Its invocation looks like:

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

We would like this form to expand into

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

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 . ,(nreverse ans)))
       (push '(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 comma 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.

18.3 Local Macro Definitions

defmacro or macro defines a macro whose name has global scope; it can be used in any function anywhere (subject to separation of name spaces by packages). You can also make local macro definitions which are in effect only in one piece of code. This is done with macrolet. Like lexical variable bindings made by let or the local function definitions made by flet, macrolet macro definitions are in effect only for code contained lexically within the body of the macrolet construct.

macrolet (*local-macros...*) *body...* *Special form*

Executes *body* and returns the values of the last form in it, with local macro definitions in effect according to *local-macros*.

Each element of *local-macros* looks like the cdr of a defmacro form:

(*name lambda-list macro-body...*)

and it is interpreted just the same way. However, *name* is only thus defined for expressions appearing within *body*.

```
(macrolet ((ifnot (x y . z) '(if (not ,x) ,y . ,z)))
  (ifnot foo (print bar) (print t)))

==> (if (not foo) (print bar) (print t))
```

It is permissible for *name* to have a global definition also, as a macro or as a function. The global definition is shadowed within *body*.

```
(macrolet ((car (x) '(cdr (assq ,x '((a . ferrari)
                                     (b . ford))))))
  ...(print (car symbol))...)
```

makes car have an unusual meaning for its explicit use, but due to lexical scoping it has no effect on what happens if print calls car.

macrolet can also hide other local definitions made by macrolet, flet or labels (page 45).

18.4 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 works just as if it had been defined by defun. If foo is compiled, however, the squaring is substituted into it and it produces the same code as

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

square's definition would be

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

(The internal formats of substs are explained in section 11.5.1, page 230.)

A similar square could be defined as a macro, but the simple way

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

has a bug: it causes the argument to be computed twice. The simplest correct definition as a macro is

```
(defmacro square (x)
  (once-only (x)
    '(* ,x ,x)))
```

See page 338 for information on once-only.

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, using once-only or creating temporary variables to make sure the arguments are computed once and in the proper order. The disadvantage of macros is that they are not functions, and so cannot be applied to arguments. Also, the effort required to guarantee the order of evaluation is a disadvantage. 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 apply. Because of an optimization, if

```
(defsubst xhack (&rest indices)
  (apply 'xfun xarg1 indices))
```

has been done then

```
(xhack a (car b))
```

is equivalent to

```
(xfun xarg1 a (car b))
```
If xfun is itself a subst, it is 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 242). 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 306.

Straightforward substitution of the arguments could cause arguments to be computed more than once, or in the wrong order. For instance, 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. In fact, a more complicated form of substitution (implemented by si:sublis-eval-once, page 348) is used so that local variables are introduced as necessary to prevent such problems.

Note that 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.

18.5 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 329) 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.

18.5.1 Name Conflicts

One of the most common errors in writing macros is best illustrated by example. Suppose we wanted to write dolist (see page 74) as a macro that expanded into a do (see page 70). 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 example's 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, in a package which only you will write code in. 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 avoid the name conflict reliably is to use an uninterned symbol as the variable in the generated code. The function gensym (see page 133) 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. (This is how the expression would print; this text would not read in properly because a new uninterned symbol would be created by each use of #:.) 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. These symbols are in the si package; as a result, a name conflict is possible only in code which uses variables in the si package. This would not normally happen in user code, which resides in other packages.

18.5.2 Block-Name Conflicts

A related problem occurs when you write a macro that expands into a prog or do (or anything equivalent) behind the user's back (unlike dolist, which is documented to be like do). Consider the error-restart special form (see page 724). Suppose we wanted to implement it as a macro that expands into a do-forever, which becomes a prog. Then the following (contrived) Lisp program would not behave correctly:

```
(dolist (a list)
  (error-restart ((sys:abort error) "Return from FOO.")
    (cond ((> a 10)
           (return 5))
          ((> a 4)
           (ferror 'lose "You lose.")))))
```

The problem is that the return would return from the error-restart instead of the prog.

There are two possible ways to avoid this. The best is to make the expanded code use only explicit block's with obscure or gensymmed block names, and never a prog or do.

The other is to give any prog or do the name t. t as a prog name is special; it causes the prog to generate only a block named t, omitting the usual block named nil which is normally generated as well. Because only blocks named nil affect return, the problem is avoided.

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

Macros like dolist specifically should expand into an ordinary do, because the user expects to be able to exit them with return.

18.5.3 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 34) as a macro. defconst must do two things, declare the variable to be special and set the variable to its initial value. (Here we 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:

```
(proclaim '(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 (proclaim '(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. The proclaim form and the setq form are evaluated. The compiler recognizes progn specially and treats each argument of the progn form as if it had been encountered at top level. Here is the macro definition:

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

Here is another example of a form that wants to expand into several things. We 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 character that should invoke the function in this subsystem: Suppose that in this subsystem, commands are always functions of no arguments, and characters are used to index a vector called dispatch-table to find the function to use. A typical call to define-command would look like:

```
(define-command move-to-top #\meta-<
  (do () ((at-the-top-p))
    (move-up-one)))
```

Expanding into:

```
(progn (setf (aref dispatch-table #\meta-<)
             'move-to-top)
       (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 specified character to invoke this command. 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 setf 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 305) 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 attach the code for a command character to the character itself. Macro expansion allows the function definitions and the command dispatch table to be made from the same source code.

18.5.4 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 a modified 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 ((*print-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 ((*print-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 standard Zetalisp constructs that are similar to this macro; see with-open-file (page 580) and with-input-from-string (page 473), for example. The really interesting thing, of course, is that you can define your own such constructs for your 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 construct 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 constructs 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 construct is "do this stuff *with* the following things true". Another convention is that any "parameters" to the construct should appear in a list that is the first subform of the construct, and that the rest of the elements 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 construct. 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 construct (page 473), 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 indent the elements of the body (see page 324).

The other thing to keep in mind is that control can leave the construct either by the last form's returning, or by a non-local exit (go, return or throw). You should write the definition in such a way that everything is cleaned up appropriately no matter how 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 82) 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 126) expands

```
(using-resource (window menu-resource) body...)
```

into

```
(let ((window nil))
  (unwind-protect
      (progn (setq window
                   (allocate-resource 'menu-resource))
             body...)
    (and window
         (deallocate-resource 'menu-resource window))))
```

This way the allocated resource item is deallocated whenever control leaves the using-resource special form.

18.5.5 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 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)))
```

Once again, the expansion would print this way, but this text would not read in as a valid expression due to the inevitable problems of #:.

In general, when you define a new construct which contains one or more argument forms, you must be careful that the expansion evaluates the argument forms the proper number of times and in the proper order. 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. But if this happens unexpectedly, it can make invocations fail to work as they appear they should.

once-only is a macro that can be used to avoid multiple evaluation. 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 *var-list body...* *Macro*

var-list is a list of variables. The *body* is 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 *body* in this new binding environment and, when they have been evaluated, it undoes the bindings. The result of the evaluation of the last form in *body* 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* is 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.

18.5.6 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 the binding of var was unmade when the expander function was done. By the time the collect form gets expanded, the binding is long gone. 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.

One way to write these macros is using macrolet:

```
(defmacro with-collection (&body body)
   (let ((var (gensym)))
     '(macrolet ((collect (argument)
                   '(push ,argument ,',var)))
        (let ((,var nil))
          ,@body
          (nreverse ,var)))))
```

Here with-collection expands into code which defines collect specially to know about which variable to collect into. ,', causes var's value to be substituted when the outer backquote, the one around the macrolet, is executed. argument, however, is substituted in when the inner backquote is executed, which happens when collect is expanded.

This technique has the interesting consequence that collect is defined only within the body of a with-collection. It would simply not be recognized elsewhere; or it could have another definition, for some other purpose, globally. This has both advantages and disadvantages.

Another technique is to communicate through local declarations. The code generated by with-collection can contain a local-declare. The expansion of collect can examine the declararion with getdecl to decide what to do. Here is the code:

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

(defmacro collect (argument)
  (let ((var ,(getdecl nil 'collection-var)))
    (unless var
      (ferror nil "COLLECT not within a WITH-COLLECTION"))
    '(push ,argument var)))
```

Another way, used before getdecl existed, was with compiler-let (see page 316). compiler-let is identical to let as far as the interpreter is concerned, so the macro continues to work in the interpreter with this change. 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 in this fashion:

```
(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*))
```

18.5.7 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 305 for more details. Doing this means that at compile time the definition of the function is interpreted, not compiled, and hence runs 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, and also all functions that the macros call. 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 28, page 660) helps keep these various files straight, compiling and loading things in the right order.

18.6 Aids for Debugging Macros

mexp &optional *form*

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 528) to improve readability). When the form itself has been expanded until it is no longer a macro call, macroexpand-all is used to expand all its subforms, and the result is printed if it is different from what preceded. This allows you to see what your macros are expanding into, without actually evaluating the result of the expansion.

If the form you type is an atom, mexp returns. Usually one simply uses Abort to exit it.

If the form you type is a list that not a macro call, nothing is printed. You are prompted immediately for another form.

If the argument *form* is given, it is expanded and printed as usual, and then mexp returns immediately.

If you type

```
(mexp)
```

followed by

```
(rest (first x))
```

then mexp will print

```
(cdr (first x))
```

and then

```
(cdr (car x))
```

You would then type Abort to exit mexp.

18.7 Displacing Macro Calls

Every time the the evaluator sees a macro form, it must call the macro to expand the form. This is time consuming. To speed things up, the expansion of the macro is recorded automatically by modifying the form using rplaca and rplacd so that it no longer appears to need expansion. If the same form is evaluated again, it can be processed straight away. This is done using the function displace.

A consequence of the evaluator's policy of displacing macro calls is that if you change the definition of a macro, the new definition does not take effect in any form that has already been displaced. An existing form which calls 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 form expansion)
```

When a form whose car is si:displaced is evaluated, the evaluator simply extracts the expansion and evaluates it. *old-form-copy* is a newly consed pair whose car and cdr are the same as the original car and cdr of the form; thus, it records the macro call which was expanded. grindef uses this information to print the code as it was, rather than as it

has been expanded.

displace returns *expansion*.

The precise format of a displaced macro call may be changed in the future to facilitate the implementation of automatic reexpansion if the called macro changes.

18.8 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* &optional *local-macros-environment*

If *form* is a macro form, this expands it (once) and returns the expanded form. Otherwise it just returns *form*. The second value is t if *form* has been expanded.

local-macros-environment is a data structure which specifies the local macro definitions (made by macrolet) to be used for this expansion in addition to the global macro definitions (made by defmacro and recorded in function cells of symbols). When macroexpand-1 is called by the evaluator, this argument comes from the evaluator's own data structures set up by any macrolet forms which *form* was found within. When macroexpand-1 is called by the compiler, this argument comes from data structures kept by the compiler in its handling of macrolet.

Sometimes macro definitions call macroexpand-1; in that case, if *form* was a subform of the macro call, a &environment argument in the macro definition can be used to obtain a value to pass as *local-macros-environment*. See page 324. setf is one example of a macro that needs to use &environment since it expands some of its subforms in deciding what code to expand into. See setf, page 36.

If *local-macros-environment* is omitted or nil, only global macro definitions are used.

macroexpand-1 expands defsubst function forms as well as macro forms.

macroexpand *form* &optional *local-macros-environment*

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*. The second value is t if one or more expansions have take place. Everything said about *local-macros-environment* under macroexpand-1 applies here too.

macroexpand expands defsubst function forms as well as macro forms.

macroexpand-all *form* &optional *local-macros-environment*

Expands all macro calls in *form*, including those which are its subforms, and returns the result. By contrast, macroexpand would not expand the subforms. This function knows the syntax of all Lisp special forms, so the result is completely accurate. Note, however, that quoted list structure within *form* is not altered; there is no way to know whether you intend such list structure to be code or to be used in constructing code.

macroexpand-hook *Variable*

The value is a function which is used by macroexpand-1 to invoke the expander function of a macro. It receives arguments just like funcall: the expander function, and the arguments for it.

In fact, the default value of this variable *is* funcall. The variable exists so that the user can set it to some other function, which performs the funcall and possibly other associated record-keeping.

macroexpand-hook is not used when a macro is expanded by the interpreter.

18.9 Definitions of Macros

The definition of a macro is a list whose car is the symbol macro. The cdr of the list is the macro's *expander function*. This expander function contains the code written in the defmacro or other construct which was used to define the macro. It may be a lambda expression, or it may be a compiled function object (FEF). Expanding the macro is done by invoking the expander function.

When an expander function is called, it receives two arguments: the macro call to be expanded, and the local macros environment. If the expansion is being done by macroexpand-1 then the local macros environment passed is the one that was given to macroexpand-1. In a macro defined with defmacro, the local macros environment can be accessed by writing an &environment parameter (see page 324).

Expander functions used to be given only one argument. For compatibility, it is useful to define expander functions so that the second argument is optional; defmacro does so. In addition, old macro definitions still work, because macroexpand-1 actually checks the number of arguments which the expander function is ready to receive, and passes only one argument if the expander function expects only one. This is done using call (see page 48).

macro-function *function-spec*

If *function-spec* is defined as a macro, then this returns its expander-function: the function which should be called, with a macro call as its sole argument, to produce the macro expansion. For certain special forms, macro-function returns the "alternate macro definition" (see below). Otherwise, macro-function returns nil.

Since a definition as a macro is really a list of the form (macro . *expander-function*), you can get the expander function using (cdr (fdefinition *function-spec*)). But it is cleaner to use macro-function.

(setf (macro-function *function-spec*) *expander*)
is permitted, and is equivalent to
(fdefine function-spec (cons 'macro expander))

Certain constructs which Common Lisp specifies as macros are actually implemented as special forms (cond, for example). These special forms have "alternate macro definitions" which are the definitions they might have if they were implemented as macros. This is so that the caller of macro-function, if it is a portable Common Lisp program, need not

know about any special forms except the standard Common Lisp ones in order to make deductions about all valid Common Lisp programs. It can instead regard as a macro any symbol on which macro-function returns a non-nil value, and treat that value as the macro expander function.

The alternate macro definition of a symbol such as cond is not actually its function definition. It exists only for macro-function to return. The existence of alternate macro definitions means that macro-function is not useful for testing whether a symbol really is defined as a macro.

18.10 Extending setf and locf

This section would logically belong within section 3.2, page 35, but it is too advanced to go there. It is placed in this chapter because it deals with concepts related to macro-expansion.

There are three ways to tell the system how to setf a function: simple defsetf when it is trivial, general defsetf which handles most other cases, and define-setf-method which provides the utmost generality.

defsetf *Macro*

The simple way to use defsetf is useful when there is a setting function which does all the work of storing a value into the appropriate place and has the proper calling conventions.

`(defsetf` *function setting-function*`)`

says that the way to store into (*function args...*) is to do (*setting-function args... new-value*). For example,

`(defsetf car sys:setcar)`

is the way setf of car is defined. Its meaning is that (setf (car *x*) *y*) should expand into (sys:setcar *x y*). (setcar is like rplaca except that setcar returns its second argument).

The more general form of defsetf is used when there is no setting function with exactly the right calling sequence. Thus,

`(defsetf` *function* (*function-args...*) (*value-arg*) *body...*`)`

tells setf how to store into (*function args...*) by providing something like a macro defininition to expand into code to do the storing. *body* computes the code; the last form in *body* returns a suitable expression. *function-args* should be a lambda list, which can have optional and rest args. *body* can substitute the values of the variables in this lambda list, to refer to the arguments in the form being setf'ed. Likewise, it can substitute in *value-arg* to refer to the value to be stored.

In fact, the *function-args* and *value-arg* are not actually the subforms of the form being setfd and the value to be stored; they are gensyms. After the *body* returns, the corresponding expressions may be substituted for the gensyms, or the gensyms may remain as local variables with a suitable let provided to bind them. This is how setf ensures a correct order of evaluation.

Example:

```
(defsetf car (list) (value) '(sys:setcar ,list ,value))
```

is how one could define the setf'ing of car using the general form of defsetf. The

simple form of defsetf can be regarded as an abbreviation for something like this.

Since setf automatically expands macros, if you define a macro whose expansion is usable in setf then the macro is usable there also. Sometimes this is not desirable. For example, the accessor subst for a slot in a defstruct structure probably expands into aref, but if the slot is declared :read-only this should not be allowed. It is prevented by means of a defsetf like this:

```
(defsetf accessor-function)
```

This means that setf is explicitly prohibited on that function.

define-setf-method *function* (*function-args...*) (*value-arg*) *body...* *Macro*

Defines how to do setf on *place*'s starting with *function*, with more power and generality than defsetf provides, but more complexity of use.

The define-setf-method form receives its arguments almost like an analogous defsetf. However, the values it receives are the actual subforms, and the actual form for the value, rather than gensyms which stand for them. The *function-args* are the actual subforms of the place to be setf'ed, and the full power of defmacro arglists can be used to match against it. *value-arg* is the actual form used as the second argument to setf.

body is once again evaluated, but it does not return an expression to do the storing. Instead, it returns five values which contain sufficient information to enable anyone to examine and modify the contents of the place. This information tells the caller which subforms of the place need to be evaluated, and how to use them to examine or set the value of the place. (Generally the function-args arglist is arranged to make each arg get one subform.) A temporary variable must be found or made (usually with gensym) for each of them. Another temporary variable should be made to correspond to the value to be stored.

Then the five values to be returned are:

0 A list of the temporary variables for the subforms of the place.

1 A list of the subforms that they correspond to.

2 A list of the temporary variables for the values to be stored. Currently there can only be one value to be stored, so there is only one variable in this list, always.

3 A form to do the storing. This form refers to some or all of the temporary variables listed in value 1.

4 A form to get the value of the place. setf does not need to do this, but push and incf do. This too should refer only to the temporary variables. No expression of contained it it should be a subexpression of the place being stored in.

This information is everything that the macro (setf or something more complicated) needs to know to decide what to do.

Example:

```
(define-setf-method car (function-spec)
  (let ((tempvars (list (gensym)))
        (tempargs (list (list-form)))
        (storevar (gensym)))
    (values tempvars tempargs (list storevar)
            '(sys:setcar ,(first tempvars) ,storevar)
            '(car ,(first tempvars)))))
```

is how one could define the setf'ing of car using define-setf-method. This definition is equivalent to the other two definitions using the simpler techniques.

get-setf-method *form*

Invokes the setf method for form (which must be a list) and returns the five values produced by the body of the define-setf-method for the symbol which is the car of form. The meanings of these five values are given immediately above. If the way to setf that symbol was defined with defsetf you still get five values, which you can interpret in the same ways; thus, defsetf is effectively an abbreviation for a suitable define-setf-method.

There are two ways to use get-setf-method. One is in a macro which, like setf or incf or push, wants to store into a place. The other is in a define-setf-method for something like ldb, which is setf by setting one of its arguments. You would append your new tempvars and tempargs to the ones you got from get-setf-method to get the combined lists which you return. The forms returned by the get-setf-method you would stick into the forms you return.

An example of a macro which uses get-setf-method is pushnew. (The real pushnew is a little hairier than this, to handle the *test*, *test-not* and *key* arguments).

```
(defmacro pushnew (value place)
  (multiple-value-bind
      (tempvars tempargs storevars storeform refform)
      (get-setf-method place)
    (si:sublis-eval-once
      (cons '(-val- . ,value) (pairlis tempvars tempargs))
      '(if (memq -val- ,refform)
           ,refform
         ,(sublis (list (cons (car storevars)
                              '(cons -val- ,refform)))
                  storeform))
      t t)))
```

An example of a define-setf-method that uses get-setf-method is that for ldb:

```
(define-setf-method ldb (bytespec int)
  (multiple-value-bind
        (temps vals stores store-form access-form)
      (get-setf-method int)
    (let ((btemp (gensym))
          (store (gensym))
          (itemp (first stores)))
      (values (cons btemp temps)
              (cons bytespec vals)
              (list store)
              '(progn
                 ,(sublis
                    (list (cons itemp
                                '(dpb ,store ,btemp
                                      ,access-form)))
                    store-form)
                 ,store)
              '(ldb ,btemp ,access-form)))))
```

What this says is that the way to setf (ldb *byte* (foo)) is computed based on the way to setf (foo).

si:sublis-eval-once *alist form* &optional *reuse-tempvars sequential-flag*

Replaces temporary variables in *form* with corresponding values according to *alist*, but generates local variables when necessary to make sure that the corresponding values are evaluated exactly once and in same order that they appear in *alist*. (This complication is skipped when the values are constant). *alist* should be a list of elements (*tempvar* . *value*). The result is a form equivalent to

```
'(let ,(mapcar #'(lambda (elt) (list (car elt) (cdr elt)))
               alist)
   ,form)
```

but it usually contains fewer temporary variables and executes faster.

If *reuse-tempvars* is non-nil, the temporary variables which appear as the cars of the elements of *alist* are allowed to appear in the resulting form. Otherwise, none of them appears in the resulting form, and if any local variables turn out to be needed, they are made afresh with gensym. *reuse-tempvars* should be used only when it is guaranteed that none of the temporary variables in *alist* is referred to by any of the values to be substituted; as, when the temporary variables have been freshly made with gensym.

If *sequential-flag* is non-nil, then the value substituted for a temporary variable is allowed to refer to the temporary variables preceding it in alist. setf and similar macros should all use this option.

define-modify-macro *macro-name (lambda-list...) combiner-function [doc-string]*

Is a quick way to define setf'ing macros which resemble incf. For example, here is how incf is defined:

```
(define-modify-macro incf (&optional (delta 1)) +
   "Increment PLACE's value by DELTA.")
```

lambda-list describes any arguments the macro accepts, but not first argument, which is always the place to be examined and modified. The old value of this place, and any additional arguments such as delta in the case of incf, are combined using the *combiner-function* (in this case, +) to get the new value which is stored back in the place.

deflocf *Macro*

Defines how to perform locf on a generalized variable. There are two forms of usage, analogous to those of defsetf.

(deflocf *function locating-function*)

says that the way to get the location of (*function args...*) is to do (*locating-function args...*). For example,

```
(deflocf car sys:car-location)
```

could be used to define locf on car forms. is the way setf of car is defined. Its meaning is that (locf (car *x*)) should expand into (sys:car-location *x*).

The more general form of deflocf is used when there is no locating function with exactly the right calling sequence. Thus,

(deflocf *function* (*function-args...*) *body...*)

tells locf how to locate (*function args...*) by providing something like a macro defininition to expand into code to do the locating. *body* computes the code; the last form in *body* returns a suitable expression. *function-args* should be a lambda list, which can have optional and rest args. *body* can substitute the values of the variables in this lambda list, to refer to the arguments in the form being locf'ed.

Example:

```
(deflocf car (list) '(sys:car-location ,list))
```

is how one could define the locf'ing of car using the general form of deflocf. The simple form of deflocf can be regarded as an abbreviation for something like this.

(deflocf *function*)

says that locf should not be allowed on forms starting with *function*. This is useful only when *function* is defined as a macro or subst, for then locf's normal action is to expand the macro call and try again. In other cases there is no way to locf a function unless you define one, so you can simply refrain from defining any way.