11

Classic Macros

This chapter shows how to define the most commonly used types of macros. They fall into three categories—with a fair amount of overlap. The first group are macros which create context. Any operator which causes its arguments to be evaluated in a new context will probably have to be defined as a macro. The first two sections describe the two basic types of context, and show how to define macros for each.

The next three sections describe macros for conditional and repeated evaluation. An operator whose arguments are to be evaluated less than once, or more than once, must also be defined as a macro. There is no sharp distinction between operators for conditional and repeated evaluation: some of the examples in this chapter do both (as well as binding). The final section explains another similarity between conditional and repeated evaluation: in some cases, both can be done with functions.

11.1 Creating Context

Context here has two senses. One sort of context is a lexical environment. The `let` special form creates a new lexical environment; the expressions in the body of a `let` will be evaluated in an environment which may contain new variables. If `x` is set to `a` at the toplevel, then

```
(let ((x 'b)) (list x))
```

will nonetheless return `(b)`, because the call to `list` will be made in an environment containing a new `x`, whose value is `b`.

```
(defmacro our-let (binds &body body)
  `((lambda ,(mapcar #'(lambda (x)
                         (if (consp x) (car x) x))
                     binds)
      ,@body)
    ,@(mapcar #'(lambda (x)
                  (if (consp x) (cadr x) nil))
              binds)))
```

Figure 11.1: Macro implementation of `let`.

An operator which is to have a body of expressions must usually be defined as a macro. Except for cases like `prog1` and `progn`, the purpose of such an operator will usually be to cause the body to be evaluated in some new context. A macro will be needed to wrap context-creating code around the body, even if the context does not include new lexical variables.

Figure 11.1 shows how `let` could be defined as a macro on `lambda`. An `our-let` expands into a function application—

```
(our-let ((x 1) (y 2))
  (+ x y))
```

expands into

```
((lambda (x y) (+ x y)) 1 2)
```

Figure 11.2 contains three new macros which establish lexical environments. Section 7.5 used `when-bind` as an example of parameter list destructuring, so this macro has already been described on page 94. The more general `when-bind*` takes a list of pairs of the form (*symbol expression*)—the same form as the first argument to `let`. If any *expression* returns `nil`, the whole `when-bind*` expression returns `nil`. Otherwise its body will be evaluated with each *symbol* bound as if by `let*`:

```
> (when-bind* ((x (find-if #'consp '(a (1 2) b)))
               (y (find-if #'oddp x)))
    (+ y 10))
11
```

Finally, the macro `with-gensyms` is itself for use in writing macros. Many macro definitions begin with the creation of gensyms, sometimes quite a number of them. The macro `with-redraw` (page 115) had to create five:

```
(defmacro when-bind ((var expr) &body body)
  `(let ((,var ,expr))
     (when ,var
       ,@body)))

(defmacro when-bind* (binds &body body)
  (if (null binds)
      `(progn ,@body)
      `(let (,(car binds))
         (if ,(caar binds)
             (when-bind* ,(cdr binds) ,@body)))))

(defmacro with-gensyms (syms &body body)
  `(let ,(mapcar #'(lambda (s)
                     `(,s (gensym)))
                 syms)
     ,@body))
```

Figure 11.2: Macros which bind variables.

```
(defmacro with-redraw ((var objs) &body body)
  (let ((gob (gensym))
        (x0 (gensym)) (y0 (gensym))
        (x1 (gensym)) (y1 (gensym)))
    ...))
```

Such definitions are simplified by `with-gensyms`, which binds a whole list of variables to gensyms. With the new macro we would write just:

```
(defmacro with-redraw ((var objs) &body body)
  (with-gensyms (gob x0 y0 x1 y1)
    ...))
```

This new macro will be used throughout the remaining chapters.

If we want to bind some variables and then, depending on some condition, evaluate one of a set of expressions, we just use a conditional within a `let`:

```
(let ((sun-place 'park) (rain-place 'library))
  (if (sunny)
      (visit sun-place)
      (visit rain-place)))
```

```
(defmacro condlet (clauses &body body)
  (let ((bodfn (gensym))
        (vars (mapcar #'(lambda (v) (cons v (gensym)))
                      (remove-duplicates
                        (mapcar #'car
                                (mappend #'cdr clauses))))))
    `(labels ((,bodfn ,(mapcar #'car vars)
                ,@body))
       (cond ,@(mapcar #'(lambda (cl)
                           (condlet-clause vars cl bodfn))
                       clauses)))))

(defun condlet-clause (vars cl bodfn)
  `(,(car cl) (let ,(mapcar #'cdr vars)
                (let ,(condlet-binds vars cl)
                  (,bodfn ,@(mapcar #'cdr vars))))))


(defun condlet-binds (vars cl)
  (mapcar #'(lambda (bindform)
              (if (consp bindform)
                  (cons (cdr (assoc (car bindform) vars))
                        (cdr bindform))))
          (cdr cl)))
```

Figure 11.3: Combination of `cond` and `let`.

Unfortunately, there is no convenient idiom for the opposite situation, where we always want to evaluate the same code, but where the *bindings* must vary depending on some condition.

Figure 11.3 contains a macro intended for such situations. As its name suggests, `condlet` behaves like the offspring of `cond` and `let`. It takes as arguments a list of binding clauses, followed by a body of code. Each of the binding clauses is guarded by a test expression; the body of code will be evaluated with the bindings specified by the first binding clause whose test expression returns true. Variables which occur in some clauses and not others will be bound to `nil` if the successful clause does not specify bindings for them:

```
> (condlet (((= 1 2) (x (princ 'a)) (y (princ 'b)))
            ((= 1 1) (y (princ 'c)) (x (princ 'd)))
            (t       (x (princ 'e)) (z (princ 'f))))
    (list x y z))
CD
(D C NIL)
```

The definition of `condlet` can be understood as a generalization of the definition of `our-let`. The latter makes its body into a function, which is applied to the results of evaluating the initial-value forms. A `condlet` expands into code which defines a local function with `labels`; within it a `cond` clause determines which set of initial-value forms will be evaluated and passed to the function.

Notice that the expander uses `mappend` instead of `mapcan` to extract the variable names from the binding clauses. This is because `mapcan` is destructive, and as Section 10.3 warned, it is dangerous to modify parameter list structure.

11.2 The `with-` Macro

There is another kind of context besides a lexical environment. In the broader sense, the context is the state of the world, including the values of special variables, the contents of data structures, and the state of things outside Lisp. Operators which build this kind of context must be defined as macros too, unless their code bodies are to be packaged up in closures.

The names of context-building macros often begin with `with-`. The most commonly used macro of this type is probably `with-open-file`. Its body is evaluated with a newly opened file bound to a user-supplied variable:

```
(with-open-file (s "dump" :direction :output)
  (princ 99 s))
```

After evaluation of this expression the file `"dump"` will automatically be closed, and its contents will be the two characters `"99"`.

This operator clearly has to be defined as a macro, because it binds `s`. However, operators which cause forms to be evaluated in a new context must be defined as macros anyway. The `ignore-errors` macro, new in CLTL2, causes its arguments to be evaluated as if in a `progn`. If an error occurs at any point, the whole `ignore-errors` form simply returns `nil`. (This would be useful, for example, when reading input typed by the user.) Though `ignore-errors` creates no variables, it still must be defined as a macro, because its arguments are evaluated in a new context.

Generally, macros which create context will expand into a block of code; additional expressions may be placed before the body, after it, or both. If code

occurs after the body, its purpose may be to leave the system in a consistent state—to clean up something. For example, `with-open-file` has to close the file it opened. In such situations, it is typical to make the context-creating macro expand into an `unwind-protect`.

The purpose of `unwind-protect` is to ensure that certain expressions are evaluated even if execution is interrupted. It takes one or more arguments, which are evaluated in order. If all goes smoothly it will return the value of the first argument, like a `prog1`. The difference is, the remaining arguments will be evaluated even if an error or throw interrupts evaluation of the first.

```
> (setq x 'a)
A
> (unwind-protect
    (progn (princ "What error?")
           (error "This error."))
    (setq x 'b))
What error?
>>Error: This error.
```

The `unwind-protect` form as a whole yields an error. However, after returning to the toplevel, we notice that the second argument still got evaluated:

```
> x
B
```

Because `with-open-file` expands into an `unwind-protect`, the file it opens will usually be closed even if an error occurs during the evaluation of its body.

Context-creating macros are mostly written for specific applications. As an example, suppose we are writing a program which deals with multiple, remote databases. The program talks to one database at a time, indicated by the global variable `*db*`. Before using a database, we have to lock it, so that no one else can use it at the same time. When we are finished we have to release the lock. If we want the value of the query `q` on the database `db`, we might say something like:

```
(let ((temp *db*))
  (setq *db* db)
  (lock *db*)
  (prog1 (eval-query q)
         (release *db*)
         (setq *db* temp)))
```

With a macro we can hide all this bookkeeping. Figure 11.4 defines a macro which will allow us to deal with databases at a higher level of abstraction. Using `with-db`, we would say just:

Pure macro:

```
(defmacro with-db (db &body body)
  (let ((temp (gensym)))
    `(let ((,temp *db*))
       (unwind-protect
         (progn
           (setq *db* ,db)
           (lock *db*)
           ,@body)
         (progn
           (release *db*)
           (setq *db* ,temp))))))
```

Combination of macro and function:

```
(defmacro with-db (db &body body)
  (let ((gbod (gensym)))
    `(let ((,gbod #'(lambda () ,@body)))
       (declare (dynamic-extent ,gbod))
       (with-db-fn *db* ,db ,gbod))))

(defun with-db-fn (old-db new-db body)
  (unwind-protect
    (progn
      (setq *db* new-db)
      (lock *db*)
      (funcall body))
    (progn
      (release *db*)
      (setq *db* old-db))))
```

Figure 11.4: A typical with- macro.

```
(with-db db
  (eval-query q))
```

Calling with-db is also safer, because it expands into an unwind-protect instead of a simple prog1.

The two definitions of with-db in Figure 11.4 illustrate two possible ways to write this kind of macro. The first is a pure macro, the second a combination of a function and a macro. The second approach becomes more practical as the

```
(defmacro if3 (test t-case nil-case ?-case)
  `(case ,test
     ((nil) ,nil-case)
     (?     ,?-case)
     (t     ,t-case)))

(defmacro nif (expr pos zero neg)
  (let ((g (gensym)))
    `(let ((,g ,expr))
       (cond ((plusp ,g) ,pos)
             ((zerop ,g) ,zero)
             (t ,neg)))))
```

Figure 11.5: Macros for conditional evaluation.

desired `with-` macro grows in complexity.

In CLTL2 Common Lisp, the `dynamic-extent` declaration allows the closure containing the body to be allocated more efficiently (in CLTL1 implementations, it will be ignored). We only need this closure for the duration of the call to `with-db-fn`, and the declaration says as much, allowing the compiler to allocate space for it on the stack. This space will be reclaimed automatically on exit from the `let` expression, instead of being reclaimed later by the garbage-collector.

11.3 Conditional Evaluation

Sometimes we want an argument in a macro call to be evaluated only under certain conditions. This is beyond the ability of functions, which always evaluate all their arguments. Built-in operators like `if`, `and`, and `cond` protect some of their arguments from evaluation unless other arguments return certain values. For example, in this expression

```
(if t
    'phew
    (/ x 0))
```

the third argument would cause a division-by-zero error if it were evaluated. But since only the first two arguments ever will be evaluated, the `if` as a whole will always safely return `phew`.

We can create new operators of this sort by writing macros which expand into calls to the existing ones. The two macros in Figure 11.5 are two of many possible

variations on `if`. The definition of `if3` shows how we could define a conditional for a three-valued logic. Instead of treating `nil` as false and everything else as true, this macro considers three categories of truth: true, false, and *uncertain*, represented as `?`. It might be used as in the following description of a five year-old:

```
(while (not sick)
  (if3 (cake-permitted)
       (eat-cake)
       (throw 'tantrum nil)
       (plead-insistently)))
```

The new conditional expands into a `case`. (The `nil` key has to be enclosed within a list because a `nil` key alone would be ambiguous.) Only one of the last three arguments will be evaluated, depending on the value of the first.

The name `nif` stands for "numeric if." Another implementation of this macro appeared on page 86. It takes a numeric expression as its first argument, and depending on its sign evaluates one of the remaining three arguments.

```
> (mapcar #'(lambda (x)
              (nif x 'p 'z 'n))
          '(0 1 -1))
(Z P N)
```

Figure 11.6 contains several more macros which take advantage of conditional evaluation. The macro `in` is to test efficiently for set membership. When you want to test whether an object is one of a set of alternatives, you could express the query as a disjunction:

```
(let ((x (foo)))
  (or (eql x (bar)) (eql x (baz))))
```

or you could express it in terms of set membership:

```
(member (foo) (list (bar) (baz)))
```

The latter is more abstract, but less efficient. The `member` expression incurs unnecessary costs from two sources. It conses, because it must assemble the alternatives into a list for `member` to search. And to form the alternatives into a list they all have to be evaluated, even though some of the values may never be needed. If the value of `(foo)` is equal to the value of `(bar)`, then there is no need to evaluate `(baz)`. Whatever its conceptual advantages, this is not a good way to use `member`. We can get the same abstraction more efficiently with a macro: `in` combines the abstraction of `member` with the efficiency of `or`. The equivalent `in` expression

```
(defmacro in (obj &rest choices)
  (let ((insym (gensym)))
    `(let ((,insym ,obj))
       (or ,@(mapcar #'(lambda (c) `(eql ,insym ,c))
                     choices)))))

(defmacro inq (obj &rest args)
  `(in ,obj ,@(mapcar #'(lambda (a)
                          `',a)
                      args)))

(defmacro in-if (fn &rest choices)
  (let ((fnsym (gensym)))
    `(let ((,fnsym ,fn))
       (or ,@(mapcar #'(lambda (c)
                         `(funcall ,fnsym ,c))
                     choices)))))

(defmacro >case (expr &rest clauses)
  (let ((g (gensym)))
    `(let ((,g ,expr))
       (cond ,@(mapcar #'(lambda (cl) (>casex g cl))
                       clauses)))))

(defun >casex (g cl)
  (let ((key (car cl)) (rest (cdr cl)))
    (cond ((consp key) `((in ,g ,@key) ,@rest))
          ((inq key t otherwise) `(t ,@rest))
          (t (error "bad >case clause")))))
```

Figure 11.6: Macros for conditional evaluation.

```
(in (foo) (bar) (baz))
```

has the same shape as the `member` expression, but expands into

```
(let ((#:g25 (foo)))
  (or (eql #:g25 (bar))
      (eql #:g25 (baz))))
```

As is often the case, when faced with a choice between a clean idiom and an efficient one, we go between the horns of the dilemma by writing a macro which

transforms the former into the latter.

Pronounced "in queue," `inq` is a quoting variant of `in`, as `setq` used to be of `set`. The expression

```
(inq operator + - *)
```

expands into

```
(in operator '+ '- '*)
```

As `member` does by default, `in` and `inq` use `eql` to test for equality. When you want to use some other test—or any other function of one argument—you can use the more general `in-if`. What `in` is to `member`, `in-if` is to `some`. The expression

```
(member x (list a b) :test #'equal)
```

can be duplicated by

```
(in-if #'(lambda (y) (equal x y)) a b)
```

and

```
(some #'oddp (list a b))
```

becomes

```
(in-if #'oddp a b)
```

Using a combination of `cond` and `in`, we can define a useful variant of `case`. The Common Lisp `case` macro assumes that its keys are constants. Sometimes we may want the behavior of a `case` expression, but with keys which are evaluated. For such situations we define `>case`, like `case` except that the keys guarding each clause are evaluated before comparison. (The `>` in the name is intended to suggest the arrow notation used to represent evaluation.) Because `>case` uses `in`, it evaluates no more of the keys than it needs to.

Since keys can be Lisp expressions, there is no way to tell if `(x y)` is a call or a list of two keys. To avoid ambiguity, keys (other than `t` and `otherwise`) must always be given in a list, even if there is only one of them. In `case` expressions, `nil` may not appear as the car of a clause on grounds of ambiguity. In a `>case` expression, `nil` is no longer ambiguous as the car of a clause, but it does mean that the rest of the clause will never be evaluated.

For clarity, the code that generates the expansion of each `>case` clause is defined as a separate function, `>casex`. Notice that `>casex` itself uses `inq`.

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))

(defmacro till (test &body body)
  `(do ()
       (,test)
     ,@body))

(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))
```

Figure 11.7: Simple iteration macros.

11.4 Iteration

Sometimes the trouble with functions is not that their arguments are always evaluated, but that they are evaluated only once. Because each argument to a function will be evaluated exactly once, if we want to define an operator which takes some body of expressions and iterates through them, we will have to define it as a macro.

The simplest example would be a macro which evaluated its arguments in sequence forever:

```
(defmacro forever (&body body)
  `(do ()
       (nil)
     ,@body))
```

This is just what the built-in `loop` macro does if you give it no loop keywords. It might seem that there is not much future (or too much future) in looping forever. But combined with `block` and `return-from`, this kind of macro becomes the most natural way to express loops where termination is always in the nature of an emergency.

Some of the simplest macros for iteration are shown in Figure 11.7. We have already seen `while` (page 91), whose body will be evaluated while a test

expression returns true. Its converse is `till`, which does the same while a test expression returns false. Finally `for`, also seen before (page 129), iterates for a range of numbers.

By defining these macros to expand into `do`s, we enable the use of `go` and `return` within their bodies. As `do` inherits these rights from `block` and `tagbody`, `while`, `till`, and `for` inherit them from `do`. As explained on page 131, the `nil` tag of the implicit `block` around `do` will be captured by the macros defined in Figure 11.7. This is more of a feature than a bug, but it should at least be mentioned explicitly.

Macros are indispensable when we need to define more powerful iteration constructs. Figure 11.8 contains two generalizations of `dolist`; both evaluate their body with a tuple of variables bound to successive subsequences of a list. For example, given two parameters, `do-tuples/o` will iterate by pairs:

```
> (do-tuples/o (x y) '(a b c d)
    (princ (list x y)))
(A B)(B C)(C D)
NIL
```

Given the same arguments, `do-tuples/c` will do the same thing, then wrap around to the front of the list:

```
> (do-tuples/c (x y) '(a b c d)
    (princ (list x y)))
(A B)(B C)(C D)(D A)
NIL
```

Both macros return `nil`, unless an explicit `return` occurs within the body.

This kind of iteration is often needed in programs which deal with some notion of a path. The suffixes `/o` and `/c` are intended to suggest that the two versions traverse open and closed paths, respectively. For example, if `points` is a list of points and (`drawline` *x y*) draws the line between *x* and *y*, then to draw the path from the first point to the last we write.

```
(do-tuples/o (x y) points (drawline x y))
```

whereas, if `points` is a list of the vertices of a polygon, to draw its perimeter we write

```
(do-tuples/c (x y) points (drawline x y))
```

The list of parameters given as the first argument can be any length, and iteration will proceed by tuples of that length. If just one parameter is given, both

```
(defmacro do-tuples/o (parms source &body body)
  (if parms
      (let ((src (gensym)))
        `(prog ((,src ,source))
           (mapc #'(lambda ,parms ,@body)
                 ,@(map0-n #'(lambda (n)
                               `(nthcdr ,n ,src))
                           (1- (length parms))))))))

(defmacro do-tuples/c (parms source &body body)
  (if parms
      (with-gensyms (src rest bodfn)
        (let ((len (length parms)))
          `(let ((,src ,source))
             (when (nthcdr ,(1- len) ,src)
               (labels ((,bodfn ,parms ,@body))
                 (do ((,rest ,src (cdr ,rest)))
                     ((not (nthcdr ,(1- len) ,rest))
                      ,@(mapcar #'(lambda (args)
                                    `(,bodfn ,@args))
                                (dt-args len rest src))
                      nil)
                   (,bodfn ,@(map1-n #'(lambda (n)
                                         `(nth ,(1- n)
                                               ,rest))
                                     len)))))))))

(defun dt-args (len rest src)
  (map0-n #'(lambda (m)
              (map1-n #'(lambda (n)
                          (let ((x (+ m n)))
                            (if (>= x len)
                                `(nth ,(- x len) ,src)
                                `(nth ,(1- x) ,rest))))
                      len))
          (- len 2)))
```

Figure 11.8: Macros for iteration by subsequences.

```
(do-tuples/c (x y z) '(a b c d)
  (princ (list x y z)))
```

expands into:

```
(let ((#:g2 '(a b c d)))
  (when (nthcdr 2 #:g2)
    (labels ((#:g4 (x y z)
               (princ (list x y z))))
      (do ((#:g3 #:g2 (cdr #:g3)))
          ((not (nthcdr 2 #:g3))
           (#:g4 (nth 0 #:g3)
                 (nth 1 #:g3)
                 (nth 0 #:g2))
           (#:g4 (nth 1 #:g3)
                 (nth 0 #:g2)
                 (nth 1 #:g2))
           nil)
        (#:g4 (nth 0 #:g3)
              (nth 1 #:g3)
              (nth 2 #:g3))))))
```

Figure 11.9: Expansion of a call to `do-tuples/c`.

degenerate to `dolist`:

```
> (do-tuples/o (x) '(a b c) (princ x))
ABC
NIL
> (do-tuples/c (x) '(a b c) (princ x))
ABC
NIL
```

The definition of `do-tuples/c` is more complex than that of `do-tuples/o`, because it has to wrap around on reaching the end of the list. If there are n parameters, `do-tuples/c` must do $n-1$ more iterations before returning:

```
> (do-tuples/c (x y z) '(a b c d)
    (princ (list x y z)))
(A B C)(B C D)(C D A)(D A B)
NIL
```

```
> (do-tuples/c (w x y z) '(a b c d)
    (princ (list w x y z)))
(A B C D)(B C D A)(C D A B)(D A B C)
NIL
```

The expansion of the former call to `do-tuples/c` is shown in Figure 11.9. The hard part to generate is the sequence of calls representing the wrap around to the front of the list. These calls (in this case, two of them) are generated by `dt-args`.

11.5 Iteration with Multiple Values

The built-in `do` macros have been around longer than multiple return values. Fortunately `do` can evolve to suit the new situation, because the evolution of Lisp is in the hands of the programmer. Figure 11.10 contains a version of `do*` adapted for multiple values. With `mvdo*`, each of the initial clauses can bind more than one variable:

```
> (mvdo* ((x 1 (1+ x))
          ((y z) (values 0 0) (values z x)))
         ((> x 5) (list x y z))
    (princ (list x y z)))
(1 0 0)(2 0 2)(3 2 3)(4 3 4)(5 4 5)
(6 5 6)
```

This kind of iteration is useful, for example, in interactive graphics programs, which often have to deal with multiple quantities like coordinates and regions.

Suppose that we want to write a simple interactive game, in which the object is to avoid being squashed between two pursuing objects. If the two pursuers both hit you at the same time, you lose; if they crash into one another first, you win. Figure 11.11 shows how the main loop of this game could be written using `mvdo*`.

It is also possible to write an `mvdo`, which binds its local variables in parallel:

```
> (mvdo ((x 1 (1+ x))
         ((y z) (values 0 0) (values z x)))
        ((> x 5) (list x y z))
    (princ (list x y z)))
(1 0 0)(2 0 1)(3 1 2)(4 2 3)(5 3 4)
(6 4 5)
```

The need for `psetq` in defining `do` was described on page 96. To define `mvdo`, ○ we need a multiple-value version of `psetq`. Since Common Lisp doesn't have one, we have to write it ourselves, as in Figure 11.12. The new macro works as follows:

```
(defmacro mvdo* (parm-cl test-cl &body body)
  (mvdo-gen parm-cl parm-cl test-cl body))

(defun mvdo-gen (binds rebinds test body)
  (if (null binds)
      (let ((label (gensym)))
        `(prog nil
           ,label
           (if ,(car test)
               (return (progn ,@(cdr test))))
           ,@body
           ,@(mvdo-rebind-gen rebinds)
           (go ,label)))
      (let ((rec (mvdo-gen (cdr binds) rebinds test body)))
        (let ((var/s (caar binds)) (expr (cadar binds)))
          (if (atom var/s)
              `(let ((,var/s ,expr)) ,rec)
              `(multiple-value-bind ,var/s ,expr ,rec))))))

(defun mvdo-rebind-gen (rebinds)
  (cond ((null rebinds) nil)
        ((< (length (car rebinds)) 3)
         (mvdo-rebind-gen (cdr rebinds)))
        (t
         (cons (list (if (atom (caar rebinds))
                         'setq
                         'multiple-value-setq)
                     (caar rebinds)
                     (third (car rebinds)))
               (mvdo-rebind-gen (cdr rebinds))))))
```

Figure 11.10: Multiple value binding version of `do*`.

```
> (let ((w 0) (x 1) (y 2) (z 3))
    (mvpsetq (w x) (values 'a 'b) (y z) (values w x))
    (list w x y z))
(A B 0 1)
```

The definition of `mvpsetq` relies on three utility functions: `mklist` (page 45), `group` (page 47), and `shuffle`, defined here, which interleaves two lists:

```
(mvdo* (((px py) (pos player)   (move player mx my))
        ((x1 y1) (pos obj1)     (move obj1 (- px x1)
                                           (- py y1)))
        ((x2 y2) (pos obj2)     (move obj2 (- px x2)
                                           (- py y2)))
        ((mx my) (mouse-vector) (mouse-vector))
        (win     nil            (touch obj1 obj2))
        (lose    nil            (and (touch obj1 player)
                                     (touch obj2 player))))
       ((or win lose) (if win 'win 'lose))
  (clear)
  (draw obj1)
  (draw obj2)
  (draw player))
```

`(pos` *obj*`)` returns two values `x,y` representing the position of *obj*. Initially, the three objects have random positions.

`(move` *obj dx dy*`)` moves the object *obj* depending on its type and the vector ⟨*dx*,*dy*⟩. Returns two values `x,y` indicating the new position.

`(mouse-vector)` returns two values `dx,dy` indicating the current movement of the mouse.

`(touch` *obj1 obj2*`)` returns true if *obj1* and *obj2* are touching.

`(clear)` clears the game region.

`(draw` *obj*`)` draws *obj* at its current position.

Figure 11.11: A game of squash.

```
> (shuffle '(a b c) '(1 2 3 4))
(A 1 B 2 C 3 4)
```

With `mvpsetq`, we can define `mvdo` as in Figure 11.13. Like `condlet`, this macro uses `mappend` instead of `mapcar` to avoid modifying the original macro call. The `mappend-mklist` idiom flattens a tree by one level:

```
> (mappend #'mklist '((a b c) d (e (f g) h) ((i)) j))
(A B C D E (F G) H (I) J)
```

```
(defmacro mvpsetq (&rest args)
  (let* ((pairs (group args 2))
         (syms  (mapcar #'(lambda (p)
                            (mapcar #'(lambda (x) (gensym))
                                    (mklist (car p))))
                        pairs)))
    (labels ((rec (ps ss)
               (if (null ps)
                   `(setq
                     ,@(mapcan #'(lambda (p s)
                                   (shuffle (mklist (car p))
                                            s))
                               pairs syms))
                   (let ((body (rec (cdr ps) (cdr ss))))
                     (let ((var/s (caar ps))
                           (expr (cadar ps)))
                       (if (consp var/s)
                           `(multiple-value-bind ,(car ss)
                                                 ,expr
                              ,body)
                           `(let ((,@(car ss) ,expr))
                              ,body)))))))
      (rec pairs syms))))

(defun shuffle (x y)
  (cond ((null x) y)
        ((null y) x)
        (t (list* (car x) (car y)
                  (shuffle (cdr x) (cdr y))))))
```

Figure 11.12: Multiple value version of `psetq`.

To help in understanding this rather large macro, Figure 11.14 contains a sample expansion.

11.6 Need for Macros

Macros aren't the only way to protect arguments against evaluation. Another is to wrap them in closures. Conditional and repeated evaluation are similar because neither problem inherently requires macros. For example, we could write a version

```
(defmacro mvdo (binds (test &rest result) &body body)
  (let ((label (gensym))
        (temps (mapcar #'(lambda (b)
                           (if (listp (car b))
                               (mapcar #'(lambda (x)
                                           (gensym))
                                       (car b))
                               (gensym)))
                       binds)))
    `(let ,(mappend #'mklist temps)
       (mvpsetq ,@(mapcan #'(lambda (b var)
                              (list var (cadr b)))
                          binds
                          temps))
       (prog ,(mapcar #'(lambda (b var) (list b var))
                      (mappend #'mklist (mapcar #'car binds))
                      (mappend #'mklist temps))
         ,label
         (if ,test
             (return (progn ,@result)))
         ,@body
         (mvpsetq ,@(mapcan #'(lambda (b)
                                (if (third b)
                                    (list (car b)
                                          (third b))))
                            binds))
         (go ,label)))))
```

Figure 11.13: Multiple value binding version of `do`.

of `if` as a function:

```
(defun fnif (test then &optional else)
  (if test
      (funcall then)
      (if else (funcall else))))
```

We would protect the `then` and `else` arguments by expressing them as closures, so instead of

```
(if (rich) (go-sailing) (rob-bank))
```

```
(mvdo ((x 1 (1+ x))
       ((y z) (values 0 0) (values z x)))
      ((> x 5) (list x y z))
  (princ (list x y z)))
```

expands into:

```
(let (#:g2 #:g3 #:g4)
  (mvpsetq #:g2 1
           (#:g3 #:g4) (values 0 0))
  (prog ((x #:g2) (y #:g3) (z #:g4))
     #:g1
     (if (> x 5)
         (return (progn (list x y z))))
     (princ (list x y z))
     (mvpsetq x (1+ x)
              (y z) (values z x))
     (go #:g1)))
```

Figure 11.14: Expansion of a call to `mvdo`.

we would say

```
(fnif (rich)
      #'(lambda () (go-sailing))
      #'(lambda () (rob-bank)))
```

If all we want is conditional evaluation, macros aren't absolutely necessary. They just make programs cleaner. However, macros are necessary when we want to take apart argument forms, or bind variables passed as arguments.

The same applies to macros for iteration. Although macros offer the only way to define an iteration construct which can be followed by a body of expressions, it is possible to do iteration with functions, so long as the body of the loop is packaged up in a function itself.[1] The built-in function `mapc`, for example, is the functional counterpart of `dolist`. The expression

```
(dolist (b bananas)
  (peel b)
  (eat b))
```

[1] It's not *impossible* to write an iteration function which doesn't need its argument wrapped up in a function. We could write a function that called `eval` on expressions passed to it as arguments. For an explanation of why it's usually bad to call `eval`, see page 278.

has the same side-effects as

```
(mapc #'(lambda (b)
          (peel b)
          (eat b))
      bananas)
```

(though the former returns `nil` and the latter returns the list `bananas`). We could likewise implement `forever` as a function,

```
(defun forever (fn)
  (do ()
      (nil)
    (funcall fn)))
```

if we were willing to pass it a closure instead of a body of expressions.

However, iteration constructs usually want to do more than just iterate, as `forever` does: they usually want to do a combination of binding and iteration. With a function, the prospects for binding are limited. If you want to bind variables to successive elements of lists, you can use one of the mapping functions. But if the requirements get much more complicated than that, you'll have to write a macro.