18

Destructuring

Destructuring is a generalization of assignment. The operators `setq` and `setf` do assignments to individual variables. Destructuring combines assignment with access: instead of giving a single variable as the first argument, we give a pattern of variables, which are each assigned the value occurring in the corresponding position in some structure.

18.1 Destructuring on Lists

As of CLTL2, Common Lisp includes a new macro called `destructuring-bind`. This macro was briefly introduced in Chapter 7. Here we consider it in more detail. Suppose that `lst` is a list of three elements, and we want to bind `x` to the first, `y` to the second, and `z` to the third. In raw CLTL1 Common Lisp, we would have had to say:

```
(let ((x (first lst))
      (y (second lst))
      (z (third lst)))
  ...)
```

With the new macro we can say instead

```
(destructuring-bind (x y z) lst
  ...)
```

which is not only shorter, but clearer as well. Readers grasp visual cues much faster than textual ones. In the latter form we are shown the relationship between `x`, `y`, and `z`; in the former, we have to infer it.

If such a simple case is made clearer by the use of destructuring, imagine the improvement in more complex ones. The first argument to `destructuring-bind` can be an arbitrarily complex tree. Imagine

```
(destructuring-bind ((first last) (month day year) . notes)
                    birthday
  ...)
```

written using `let` and the list access functions. Which raises another point: destructuring makes it easier to write programs as well as easier to read them.

Destructuring did exist in CLTL1 Common Lisp. If the patterns in the examples above look familiar, it's because they have the same form as macro parameter lists. In fact, `destructuring-bind` *is* the code used to take apart macro argument lists, now sold separately. You can put anything in the pattern that you would put in a macro parameter list, with one unimportant exception (the `&environment` keyword).

Establishing bindings *en masse* is an attractive idea. The following sections describe several variations upon this theme.

18.2 Other Structures

There is no reason to limit destructuring to lists. Any complex object is a candidate for it. This section shows how to write macros like `destructuring-bind` for other kinds of objects.

The natural next step is to handle sequences generally. Figure 18.1 contains a macro called `dbind`, which resembles `destructuring-bind`, but works for any kind of sequence. The second argument can be a list, a vector, or any combination thereof:

```
> (dbind (a b c) #(1 2 3)
    (list a b c))
(1 2 3)
> (dbind (a (b c) d) '( 1 #(2 3) 4)
    (list a b c d))
(1 2 3 4)
> (dbind (a (b . c) &rest d) '(1 "fribble" 2 3 4)
    (list a b c d))
(1 #\f "ribble" (2 3 4))
```

```
(defmacro dbind (pat seq &body body)
  (let ((gseq (gensym)))
    `(let ((,gseq ,seq))
       ,(dbind-ex (destruc pat gseq #'atom) body))))

(defun destruc (pat seq &optional (atom? #'atom) (n 0))
  (if (null pat)
      nil
      (let ((rest (cond ((funcall atom? pat) pat)
                        ((eq (car pat) '&rest) (cadr pat))
                        ((eq (car pat) '&body) (cadr pat))
                        (t nil))))
       (if rest
           `((,rest (subseq ,seq ,n)))
           (let ((p (car pat))
                 (rec (destruc (cdr pat) seq atom? (1+ n))))
             (if (funcall atom? p)
                 (cons `(,p (elt ,seq ,n))
                       rec)
                 (let ((var (gensym)))
                   (cons (cons `(,var (elt ,seq ,n))
                               (destruc p var atom?))
                         rec)))))))))

(defun dbind-ex (binds body)
  (if (null binds)
      `(progn ,@body)
      `(let ,(mapcar #'(lambda (b)
                         (if (consp (car b))
                             (car b)
                             b))
                     binds)
         ,(dbind-ex (mapcan #'(lambda (b)
                                (if (consp (car b))
                                    (cdr b)))
                            binds)
                    body))))
```

Figure 18.1: General sequence destructuring operator.

The `#(` read-macro is for representing vectors, and `#\` for representing characters. Since `"abc"` = `#(#\a #\b #\c)`, the first element of `"fribble"` is the character `#\f`. For the sake of simplicity, `dbind` supports only the `&rest` and `&body` keywords.

Compared to most of the macros seen so far, `dbind` is big. It's worth studying the implementation of this macro, not only to understand how it works, but also because it embodies a general lesson about Lisp programming. As section 3.4 mentioned, Lisp programs may intentionally be written in a way that will make them easy to test. In most code, we have to balance this desire against the need for speed. Fortunately, as Section 7.8 explained, speed is not so important in expander code. When writing code that generates macroexpansions, we can make life easier for ourselves. The expansion of `dbind` is generated by two functions, `destruc` and `dbind-ex`. Perhaps they both could be combined into one function which would do everything in a single pass. But why bother? As two separate functions, they will be easier to test. Why trade this advantage for speed we don't need?

The first function, `destruc`, traverses the pattern and associates each variable with the location of the corresponding object at runtime:

```
> (destruc '(a b c) 'seq #'atom)
((A (ELT SEQ 0)) (B (ELT SEQ 1)) (C (ELT SEQ 2)))
```

The optional third argument is the predicate used to distinguish pattern structure from pattern content.

To make access more efficient, a new variable (a gensym) will be bound to each subsequence:

```
> (destruc '(a (b . c) &rest d) 'seq)
((A (ELT SEQ 0))
 ((#:G2 (ELT SEQ 1)) (B (ELT #:G2 0)) (C (SUBSEQ #:G2 1)))
 (D (SUBSEQ SEQ 2)))
```

The output of `destruc` is sent to `dbind-ex`, which generates the bulk of the macroexpansion. It translates the tree produced by `destruc` into a nested series of `lets`:

```
> (dbind-ex (destruc '(a (b . c) &rest d) 'seq) '(body))
(LET ((A (ELT SEQ 0))
      (#:G4 (ELT SEQ 1))
      (D (SUBSEQ SEQ 2)))
  (LET ((B (ELT #:G4 0))
        (C (SUBSEQ #:G4 1)))
    (PROGN BODY)))
```

```
(defmacro with-matrix (pats ar &body body)
  (let ((gar (gensym)))
    `(let ((,gar ,ar))
       (let ,(let ((row -1))
               (mapcan
                 #'(lambda (pat)
                     (incf row)
                     (setq col -1)
                     (mapcar #'(lambda (p)
                                 `(,p (aref ,gar
                                            ,row
                                            ,(incf col))))
                             pat))
                 pats))
         ,@body))))

(defmacro with-array (pat ar &body body)
  (let ((gar (gensym)))
    `(let ((,gar ,ar))
       (let ,(mapcar #'(lambda (p)
                         `(,(car p) (aref ,gar ,@(cdr p))))
                     pat)
         ,@body))))
```

Figure 18.2: Destructuring on arrays.

Note that `dbind`, like `destructuring-bind`, assumes that it will find all the list structure it is looking for. Left-over variables are not simply bound to `nil`, as with `multiple-value-bind`. If the sequence given at runtime does not have all the expected elements, destructuring operators generate an error:

```
> (dbind (a b c) (list 1 2))
>>Error: 2 is not a valid index for the sequence (1 2)
```

What other objects have internal structure? There are arrays generally, which differ from vectors in having more than one dimension. If we define a destructuring macro for arrays, how do we represent the pattern? For two-dimensional arrays, it is still practical to use a list. Figure 18.2 contains a macro, `with-matrix`, for destructuring on two-dimensional arrays.

```
(defmacro with-struct ((name . fields) struct &body body)
  (let ((gs (gensym)))
    `(let ((,gs ,struct))
       (let ,(mapcar #'(lambda (f)
                         `(,f (,(symb name f) ,gs)))
                     fields)
         ,@body))))
```

Figure 18.3: Destructuring on structures.

```
> (setq ar (make-array '(3 3)))
#<Simple-Array T (3 3) C2D39E>
> (for (r 0 2)
    (for (c 0 2)
      (setf (aref ar r c) (+ (* r 10) c))))
NIL
> (with-matrix ((a b c)
                (d e f)
                (g h i)) ar
    (list a b c d e f g h i))
(0 1 2 10 11 12 20 21 22)
```

For large arrays or those with dimension 3 or higher, we want a different kind of approach. We are not likely to want to bind variables to each element of a large array. It will be more practical to make the pattern a sparse representation of the array—containing variables for only a few elements, plus coordinates to identify them. The second macro in Figure 18.2 is built on this principle. Here we use it to get the diagonal of our previous array:

```
> (with-array ((a 0 0) (d 1 1) (i 2 2)) ar
    (values a d i))
0
11
22
```

With this new macro we have begun to move away from patterns whose elements must occur in a fixed order. We can make a similar sort of macro to bind variables to fields in structures built by `defstruct`. Such a macro is defined in Figure 18.3. The first argument in the pattern is taken to be the prefix associated with the structure, and the rest are field names. To build access calls, this macro uses `symb` (page 58).

```
> (defstruct visitor  name title firm)
VISITOR
> (setq theo (make-visitor :name  "Theodebert"
                           :title 'king
                           :firm  'franks))
#S(VISITOR NAME "Theodebert" TITLE KING FIRM FRANKS)
> (with-struct (visitor- name firm title) theo
    (list name firm title))
("Theodebert" FRANKS KING)
```

18.3 Reference

CLOS brings with it a macro for destructuring on instances. Suppose `tree` is a class with three slots, `species`, `age`, and `height`, and that `my-tree` is an instance of `tree`. Within

```
(with-slots (species age height) my-tree
  ...)
```

we can refer to the slots of `my-tree` as if they were ordinary variables. Within the body of the `with-slots`, the symbol `height` refers to the `height` slot. It is not simply bound to the value stored there, but *refers* to the slot, so that if we write:

```
(setq height 72)
```

then the `height` slot of `my-tree` will be given the value 72. This macro works by defining `height` as a symbol-macro (Section 7.11) which expands into a slot reference. In fact, it was to support macros like `with-slots` that `symbol-macrolet` was added to Common Lisp.

Whether or not `with-slots` is really a destructuring macro, it has the same role pragmatically as `destructuring-bind`. As conventional destructuring is to call-by-value, this new kind is to call-by-name. Whatever we call it, it looks to be useful. What other macros can we define on the same principle?

We can create a call-by-name version of any destructuring macro by making it expand into a `symbol-macrolet` rather than a `let`. Figure 18.4 shows a version of `dbind` modified to behave like `with-slots`. We can use `with-places` as we do `dbind`:

```
> (with-places (a b c) #(1 2 3)
    (list a b c))
(1 2 3)
```

```
(defmacro with-places (pat seq &body body)
  (let ((gseq (gensym)))
    `(let ((,gseq ,seq))
       ,(wplac-ex (destruc pat gseq #'atom) body))))

(defun wplac-ex (binds body)
  (if (null binds)
      `(progn ,@body)
      `(symbol-macrolet ,(mapcar #'(lambda (b)
                                     (if (consp (car b))
                                         (car b)
                                         b))
                                 binds)
         ,(wplac-ex (mapcan #'(lambda (b)
                                (if (consp (car b))
                                    (cdr b)))
                            binds)
                    body))))
```

Figure 18.4: Reference destructuring on sequences.

But the new macro also gives us the option to `setf` positions in sequences, as we do slots in `with-slots`:

```
> (let ((lst '(1 (2 3) 4)))
    (with-places (a (b . c) d) lst
      (setf a 'uno)
      (setf c '(tre)))
    lst)
(UNO (2 TRE) 4)
```

As in a `with-slots`, the variables now refer to the corresponding locations in the structure. There is one important difference, however: you must use `setf` rather than `setq` to set these pseudo-variables. The `with-slots` macro must invoke a code-walker (page 273) to transform `setq`s into `setf`s within its body. Here, writing a code-walker would be a lot of code for a small refinement.

If `with-places` is more general than `dbind`, why not just use it all the time? While `dbind` associates a variable with a value, `with-places` associates it with a set of instructions for finding a value. Every reference requires a lookup. Where `dbind` would bind `c` to the value of `(elt x 2)`, `with-places` will make `c` a symbol-macro that expands into `(elt x 2)`. So if `c` is evaluated *n* times in the

body, that will entail *n* calls to `elt`. Unless you actually want to `setf` the variables created by destructuring, `dbind` will be faster.

The definition of `with-places` is only slightly changed from that of `dbind` (Figure 18.1). Within `wplac-ex` (formerly `dbind-ex`) the `let` has become a `symbol-macrolet`. By similar alterations, we could make a call-by-name version of any normal destructuring macro.

18.4 Matching

As destructuring is a generalization of assignment, *pattern-matching* is a generalization of destructuring. The term "pattern-matching" has many senses. In this context, it means comparing two structures, possibly containing variables, to see if there is some way of assigning values to the variables which makes the two equal. For example, if `?x` and `?y` are variables, then the two lists

```
(p ?x ?y c ?x)
(p  a  b c  a)
```

match when `?x` = `a` and `?y` = `b`. And the lists

```
(p ?x b ?y a)
(p ?y b  c a)
```

match when `?x` = `?y` = `c`.

Suppose a program works by exchanging messages with some outside source. To respond to a message, the program has to tell what kind of message it is, and also to extract its specific content. With a matching operator we can combine the two steps.

To be able to write such an operator we have to invent some way of distinguishing variables. We can't just say that all symbols are variables, because we will want symbols to occur as arguments within patterns. Here we will say that a pattern variable is a symbol beginning with a question mark. If it becomes inconvenient, this convention could be changed simply by redefining the predicate `var?`.

Figure 18.5 contains a pattern-matching function similar to ones that appear ○ in several introductions to Lisp. We give `match` two lists, and if they can be made to match, we will get back a list showing how:

```
> (match '(p a b c a) '(p ?x ?y c ?x))
((?Y . B) (?X . A))
T
```

```
(defun match (x y &optional binds)
  (acond2
    ((or (eql x y) (eql x '_) (eql y '_)) (values binds t))
    ((binding x binds) (match it y binds))
    ((binding y binds) (match x it binds))
    ((varsym? x) (values (cons (cons x y) binds) t))
    ((varsym? y) (values (cons (cons y x) binds) t))
    ((and (consp x) (consp y) (match (car x) (car y) binds))
     (match (cdr x) (cdr y) it))
    (t (values nil nil))))

(defun varsym? (x)
  (and (symbolp x) (eq (char (symbol-name x) 0) #\?)))

(defun binding (x binds)
  (labels ((recbind (x binds)
             (aif (assoc x binds)
                  (or (recbind (cdr it) binds)
                      it))))
    (let ((b (recbind x binds)))
      (values (cdr b) b))))
```

Figure 18.5: Matching function.

```
> (match '(p ?x b ?y a) '(p ?y b c a))
((?Y . C) (?X . ?Y))
T
> (match '(a b c) '(a a a))
NIL
NIL
```

As `match` compares its arguments element by element, it builds up assignments of values to variables, called *bindings*, in the parameter `binds`. If the match is successful, `match` returns the bindings generated, otherwise it returns `nil`. Since not all successful matches generate any bindings, `match`, like `gethash`, returns a second value to indicate whether the match succeeded or failed:

```
> (match '(p ?x) '(p ?x))
NIL
T
```

```
(defmacro if-match (pat seq then &optional else)
  `(aif2 (match ',pat ,seq)
         (let ,(mapcar #'(lambda (v)
                           `(,v (binding ',v it)))
                       (vars-in then #'atom))
           ,then)
         ,else))

(defun vars-in (expr &optional (atom? #'atom))
  (if (funcall atom? expr)
      (if (var? expr) (list expr))
      (union (vars-in (car expr) atom?)
             (vars-in (cdr expr) atom?))))

(defun var? (x)
  (and (symbolp x) (eq (char (symbol-name x) 0) #\?)))
```

Figure 18.6: Slow matching operator.

When `match` returns `nil` and `t` as above, it indicates a successful match which yielded no bindings.

Like Prolog, `match` treats _ (underscore) as a wild-card. It matches everything, and has no effect on the bindings:

```
> (match '(a ?x b) '(_ 1 _))
((?X . 1))
T
```

Given `match`, it is easy to write a pattern-matching version of `dbind`. Figure 18.6 contains a macro called `if-match`. Like `dbind`, its first two arguments are a pattern and a sequence, and it establishes bindings by comparing the pattern with the sequence. However, instead of a body it has two more arguments: a `then` clause to be evaluated, with new bindings, if the match succeeds; and an `else` clause to be evaluated if the match fails. Here is a simple function which uses `if-match`:

```
(defun abab (seq)
  (if-match (?x ?y ?x ?y) seq
      (values ?x ?y)
      nil))
```

If the match succeeds, it will establish values for `?x` and `?y`, which will be returned:

```
> (abab '(hi ho hi ho))
HI
HO
```

The function `vars-in` returns all the pattern variables in an expression. It calls `var?` to test if something is a variable. At the moment, `var?` is identical to `varsym?` (Figure 18.5), which is used to detect variables in binding lists. We have two distinct functions in case we want to use different representations for the two kinds of variables.

As defined in Figure 18.6, `if-match` is short, but not very efficient. It does too much work at runtime. We traverse both sequences at runtime, even though the first is known at compile-time. Worse still, during the process of matching, we cons up lists to hold the variable bindings. If we take advantage of information known at compile-time, we can write a version of `if-match` which performs no unnecessary comparisons, and doesn't cons at all.

If one of the sequences is known at compile-time, and only that one contains variables, then we can go about things differently. In a call to `match`, either argument could contain variables. By restricting variables to the first argument of `if-match`, we make it possible to tell at compile-time which variables will be involved in the match. Then instead of creating lists of variable bindings, we could keep the values of variables in the variables themselves.

The new version of `if-match` appears in Figure 18.7 and 18.8. When we can predict what code would be evaluated at runtime, we can simply generate it at compile-time. Here, instead of expanding into a call to `match`, we generate code which performs just the right comparisons.

If we are going to use the variable `?x` to contain the binding of `?x`, how do we represent a variable for which no binding has yet been established by the match? Here we will indicate that a pattern variable is unbound by binding it to a gensym. So `if-match` begins by generating code which will bind all the variables in the pattern to gensyms. In this case, instead of expanding into a `with-gensyms`, it's safe to make the gensyms once at compile-time and insert them directly into the expansion.

The rest of the expansion is generated by `pat-match`. This macro takes the same arguments as `if-match`; the only difference is that it establishes no new bindings for pattern variables. In some situations this is an advantage, and Chapter 19 will use `pat-match` as an operator in its own right.

In the new matching operator, the distinction between pattern content and pattern structure will be defined by the function `simple?`. If we want to be able to use quoted literals in patterns, the destructuring code (and `vars-in`) have to be told not to go inside lists whose first element is `quote`. With the new matching operator, we will be able to use lists as pattern elements, simply by quoting them.

```
(defmacro if-match (pat seq then &optional else)
  `(let ,(mapcar #'(lambda (v) `(,v ',(gensym)))
                 (vars-in pat #'simple?))
     (pat-match ,pat ,seq ,then ,else)))

(defmacro pat-match (pat seq then else)
  (if (simple? pat)
      (match1 `((,pat ,seq)) then else)
      (with-gensyms (gseq gelse)
        `(labels ((,gelse () ,else))
           ,(gen-match (cons (list gseq seq)
                             (destruc pat gseq #'simple?))
                       then
                       `(,gelse))))))

(defun simple? (x) (or (atom x) (eq (car x) 'quote)))

(defun gen-match (refs then else)
  (if (null refs)
      then
      (let ((then (gen-match (cdr refs) then else)))
        (if (simple? (caar refs))
            (match1 refs then else)
            (gen-match (car refs) then else)))))
```

Figure 18.7: Fast matching operator.

Like `dbind`, `pat-match` calls `destruc` to get a list of the calls that will take apart its argument at runtime. This list is passed on to `gen-match`, which recursively generates matching code for nested patterns, and thence to `match1`, which generates match code for each leaf of the pattern tree.

Most of the code which will appear in the expansion of an `if-match` comes from `match1`, which is shown in Figure 18.8. This function considers four cases. If the pattern argument is a gensym, then it is one of the invisible variables created by `destruc` to hold sublists, and all we need to do at runtime is test that it has the right length. If the pattern element is a wildcard (`_`), no code need be generated. If the pattern element is a variable, `match1` generates code to match it against, or set it to, the corresponding part of the sequence given at runtime. Otherwise, the pattern element is taken to be a literal value, and `match1` generates code to compare it with the corresponding part of the sequence.

```
(defun match1 (refs then else)
  (dbind ((pat expr) . rest) refs
    (cond ((gensym? pat)
           `(let ((,pat ,expr))
              (if (and (typep ,pat 'sequence)
                       ,(length-test pat rest))
                  ,then
                  ,else)))
          ((eq pat '_) then)
          ((var? pat)
           (let ((ge (gensym)))
             `(let ((,ge ,expr))
                (if (or (gensym? ,pat) (equal ,pat ,ge))
                    (let ((,pat ,ge)) ,then)
                    ,else))))
          (t `(if (equal ,pat ,expr) ,then ,else)))))

(defun gensym? (s)
  (and (symbolp s) (not (symbol-package s))))

(defun length-test (pat rest)
  (let ((fin (caadar (last rest))))
    (if (or (consp fin) (eq fin 'elt))
        `(= (length ,pat) ,(length rest))
        `(> (length ,pat) ,(- (length rest) 2)))))
```

Figure 18.8: Fast matching operator (continued).

Let's look at examples of how some parts of the expansion are generated. Suppose we begin with

```
(if-match (?x 'a) seq
    (print ?x)
    nil)
```

The pattern will be passed to `destruc`, with some gensym (call it g for legibility) to represent the sequence:

```
(destruc '(?x 'a) 'g #'simple?)
```

yielding:

```
((?x (elt g 0)) ((quote a) (elt g 1)))
```

On the front of this list we cons `(g seq)`:

```
((g seq) (?x (elt g 0)) ((quote a) (elt g 1)))
```

and send the whole thing to `gen-match`. Like the naive implementation of `length` (page 22), `gen-match` first recurses all the way to the end of the list, and then builds its return value on the way back up. When it has run out of elements, `gen-match` returns its `then` argument, which will be `?x`. On the way back up the recursion, this return value will be passed as the `then` argument to `match1`. Now we will have a call like:

```
(match1 '(((quote a) (elt g 1))) '(print ?x) '⟨else function⟩)
```

yielding:

```
(if (equal (quote a) (elt g 1))
    (print ?x)
    ⟨else function⟩)
```

This will in turn become the `then` argument to another call to `match1`, the value of which will become the `then` argument of the last call to `match1`. The full expansion of this `if-match` is shown in Figure 18.9.

In this expansion gensyms are used in two completely unrelated ways. The variables used to hold parts of the tree at runtime have gensymed names, in order to avoid capture. And the variable `?x` is initially bound to a gensym, to indicate
○ that it hasn't yet been assigned a value by matching.

In the new `if-match`, the pattern elements are now evaluated instead of being
○ implicitly quoted. This means that Lisp variables can be used in patterns, as well as quoted expressions:

```
> (let ((n 3))
    (if-match (?x n 'n '(a b)) '(1 3 n (a b))
      ?x))
1
```

Two further improvements appear because the new version calls `destruc` (Figure 18.1). The pattern can now contain `&rest` or `&body` keywords (`match` doesn't bother with those). And because `destruc` uses the generic sequence operators `elt` and `subseq`, the new `if-match` will work for any kind of sequence. If `abab` is defined with the new version, it can be used also on vectors and strings:

```
(if-match (?x 'a) seq
    (print ?x))
```

expands into:

```
(let ((?x '#:g1))
  (labels ((#:g3 nil nil))
    (let ((#:g2 seq))
      (if (and (typep #:g2 'sequence)
               (= (length #:g2) 2))
          (let ((#:g5 (elt #:g2 0)))
            (if (or (gensym? x) (equal ?x #:g5))
                (let ((?x #:g5))
                   (if (equal 'a (elt #:g2 1))
                       (print ?x)
                       (#:g3)))
                (#:g3)))
          (#:g3)))))
```

Figure 18.9: Expansion of an `if-match`.

```
> (abab "abab")
#\a
#\b
> (abab #(1 2 1 2))
1
2
```

In fact, patterns can be as complex as patterns to `dbind`:

```
> (if-match (?x (1 . ?y) . ?x) '((a b) #(1 2 3) a b)
      (values ?x ?y))
(A B)
#(2 3)
```

Notice that, in the second return value, the elements of the vector are displayed. To have vectors printed this way, set `*print-array*` to `t`.

In this chapter we are beginning to cross the line into a new kind of programming. We began with simple macros for destructuring. In the final version of `if-match` we have something that looks more like its own language. The remaining chapters describe a whole class of programs which operate on the same philosophy.