3

CONTROL STRUCTURES

We have seen recursion using recursive function definitions, explicit sequencing using PROGN, and conditional evaluation using IF. Lisp has a much richer set of control primitives, however. Some, such as COND, AND, and OR, are not inherently more powerful than those we have already seen, but they often let us write more understandable code. We can do this by choosing the one that best expresses the structure or "sense" of the desired computation. Other special forms provide new levels of control. For instance, LET and LET* provide a mechanism for introducing new local variables besides those that appear in a functions argument list. In addition, *iteration* is a useful and valid Lisp control structure. Later we will describe the DO special form, which provides a structured approach to iteration. Additionally, DOLIST and DOTIMES provide syntactically simpler specialized versions of DO that cover many of its common uses. These are described in chapter 6. In chapter 12, we will see some other Lisp control structures, maintained in Common Lisp for compatibility with old Lisps, which provide an unstructured approach to iteration.

3.1 SPECIAL FORMS FOR MORE READABLE CODE

Examine again our definition of EQUAL in section 2.7. Notice how the code tends to slide over toward the right of the page with an IF forming the *else* part of each previous IF. There is another special form in Lisp that provides a convenient way to write such strings of *if* tests. Called COND[1] for *conditional*, it takes the form

(COND (*test-1 result-1*) (*test-2 result-2*) ...).

When the Lisp evaluation procedure sees a COND, it evaluates each of the test forms (e.g., *test-1*, *test-2*, etc.) in turn. When one evaluates to non-NIL, the corresponding result form (e.g., *result-1*, *result-2*, etc.) is evaluated and its value is returned as the result of the COND. In such a case, no further test or result forms are evaluated. If all test forms evaluate to NIL, NIL is returned as the result of the COND.[2]

We can rewrite our function as

```
(DEFUN EQUAL (X Y)
  (COND ((ATOM X) (EQL X Y))
        ((ATOM Y) 'NIL)
        ((EQUAL (CAR X) (CAR Y))
         (EQUAL (CDR X) (CDR Y)))
        (T 'NIL)))
```

[1] Actually COND historically predates IF in Lisp. Indeed, in some Lisps IF is implemented in terms of COND.

[2] In some Lisps, it is required that at least one of the test forms evaluate to non-NIL; it is illegal to "fall off the end" of a COND.

The final clause (i.e., (T 'NIL)) illustrates a common practice with COND. The test form here is T, which always evaluates to T. Hence, if no other test form has turned up non-NIL, this test form forces its result to be returned. Equally valid clauses with identical behavior could have been

```
('T 'NIL)
(3.14 'NIL)
('(ANY ARBITRARY CONSTANT (FOR INSTANCE THIS LIST)) 'NIL)
```

By convention, the symbol T, is usually used when an "always succeeds" clause is wanted, and this has become an idiom in Lisp programming.[3] Notice that there is no point in writing further clauses after such a "T clause," because they can never be reached.

Notice also that in this particular case we could have left the final clause out and the function would have exactly the same behavior. Putting the clause in makes for a clearer program, stating explicitly what the result will be if none of the other test clauses succeed.

Two other useful special forms are AND and OR. These correspond roughly to logical *and* and *or*. They look like

(AND *form-1 form-2... form-n*)

and

(OR *form-1 form-2... form-n*).

In each case, their argument forms are evaluated left to right until it is first possible to say whether all the forms evaluate to non-NIL (in the case of AND) or whether at least one of the forms evaluates to non-NIL (in the case of OR). Thus, AND evaluates each argument form in turn until one returns NIL—in which case AND returns NIL without evaluating the rest of the forms. If all of the forms return non-NIL, the value of the last is returned. Thus, AND returns "truth" only

[3] So much so that it has been transported to various other constructs in some Lisps where there is no such logical explanation of the idiom, and in fact the only justification that can be given is "so that it looks like COND."

if all its arguments return "truth." On the other hand OR returns "truth" if any of its arguments return "truth." It does this by evaluating each argument form in turn until one returns non-NIL, and then it returns the result as the result of the OR. If all argument forms return NIL, the result of the OR is NIL.

We can now rewrite our simple version of EQL in a somewhat clearer way:

```
(DEFUN EQL (A B)
    (OR (EQ A B)
        (AND (OR (AND (TYPEP A 'FIXNUM)
                      (TYPEP B 'FIXNUM))
                 (AND (TYPEP A 'SINGLE-FLOAT)
                      (TYPEP B 'SINGLE-FLOAT)))
             (ZEROP (- A B)))))
```

The reorganization of EQL is based on the following observation: If the two arguments are the same object as determined by EQ, they are certainly EQL. If that is not true, the second subform of the OR special form is evaluated. If that is the case, EQL should only return T if the two arguments have the same type and if their difference is zero. The outer AND has two subclauses. The first ensures that the two arguments have the same type, while the second compares them. Here we see the importance of AND giving up evaluating clauses as soon as one of them returns NIL, as the subtraction should not be done unless we are sure that B is a number. This sort of type checking is the most typical use of AND---often more than two clauses are used, however. Within the first clause of the outer AND, there are two possibilities that need to be checked. If either both numbers have type FIXNUM or both have type SINGLE-FLOAT, the comparison in the second subclause should determine the value of the EQL test.

In the foregoing definition of EQL, notice that neither NIL nor T appear explicitly. The former can be returned as the value of a failing AND clause, while the latter can be returned as the result of a call to EQ or ZEROP.

An example of EQUAL using the AND special form where possible follows.

```
(DEFUN EQUAL (X Y)
    (COND ((ATOM X) (EQL X Y))
          ((ATOM Y) 'NIL)
          (T (AND (EQUAL (CAR X) (CAR Y))
                  (EQUAL (CDR X) (CDR Y))))))
```

Here the AND explicitly demonstrates that both the CARs *and* the CDRs must be equal for the two cons cells to be equal. At the same time, the semantics of AND ensure that the CDRs are never needlessly compared if the CARs do not match. Notice also that we have removed the final (T 'NIL) clause in this version of EQUAL without detracting from the understandability of the function definition.

EXERCISES

E3.1.1 For each s-expression, list the predicates that are evaluated in order of evaluation:

a. (AND (NUMBERP 3) (ATOM 'A) (NULL 'T))
b. (AND (NUMBERP 3) (ATOM '(A B)) (NULL 'T))
c. (OR (NUMBERP 3) (ATOM '(A B)) (NULL 'T))
d. (OR (ATOM '(X Y Z)) (NUMBERP 'B) (CONSP 'X))

E3.1.2 Write the following code without using COND:

```
(COND ((EQ DAY 'SATURDAY)
       'FOOTBALL)
      ((EQ DAY 'SUNDAY)
       'DIEM-SUM)
      ((EQ DAY 'WEDNESDAY)
       'LAB)
      (T 'CLASS))
```

3.2 ORGANIZING FOR EFFICIENCY

Look again at the trace of our first definition for EQUAL. Notice in particular how long it took the recursive calls to decide that F is equal to F and NIL is equal to NIL. Also notice that if two EQ lists were given to this EQUAL it would recursively decompose them and compare all of their components. However, any two things that are EQ are certainly EQUAL, since they are the same object.

Thus, it makes sense to first check for EQness of the two arguments to EQUAL because that case is quite common; it is the case of all nonnumeric leafs in comparing any two lists. Since the case of comparing two numbers would still have to be handled by EQL, the test might well be made into an EQL test. Doing this has ramifications for the rest of the function too. In the clause that succeeds when X is an atom, it is no longer necessary to include a test of whether it is EQL to Y. However, it is necessary to check for the atomicity of both X and Y before treating them as cons cells in the last clause of the COND. The result is

```
(DEFUN EQUAL (X Y)
    (COND ((EQL X Y) 'T)
          ((OR (ATOM X)
               (ATOM Y))
           'NIL)
          (T (AND (EQUAL (CAR X) (CAR Y))
                  (EQUAL (CDR X) (CDR Y))))))
```

Notice that there are exactly the same calls to predefined Lisp functions as before—they simply have been reordered. Notice also that now there is an explicit T in the function definition.

3.3 LET AND LET*: BINDING LOCAL VARIABLES

So far, whenever we have computed a value we have handed it directly to a function needing it. Sometimes we may need to hand a computed value to two or more different places, or we may need to save a result while we do some order-critical computations (where side effects are involved). We saw an example of the latter in our definition of NREVERSE and NREVERSE-AUX—we were saved there by making use of the left to right evaluation of arguments by the Lisp evaluation procedure.

The special form LET provides a way of introducing temporary variables into a function. Temporary variables can save the result of a computation. The general form is

```
(LET ((var-1  value-1)
      (var-2  value-2)
      .
      .
      .
      (var-n  value-n))
     body)
```

Here *var-1*, *var-2*, ... *var-n* are symbols (LET does not evaluate them) that will be used as names of introduced variables. They can be referred to by any code that appears in the *body* form. When the LET is entered, each of the s-expressions *value-1*, *value-2*, ... *value-n* is evaluated in turn. When they are all done, the new variables are given their values. Thus, *value-3*, say, cannot legitimately refer to *var-1*.

Suppose we represent points in the plane as a list of two numbers, the first being the point's x coordinate and the second its y coordinate. Then we could compute the distance between two points with

```
(DEFUN DISTANCE (P1 P2)
    (LET ((XDIFF (- (CAR P1) (CAR P2)))
          (YDIFF (- (CAR (CDR P1)) (CAR (CDR P2)))))
         (SQRT (+ (* XDIFF XDIFF) (* YDIFF YDIFF)))))
```

(Notice that SQRT is a function of one numeric argument that computes its square

root.) Without the benefit of temporary variables, we either would have had to extract the x and y components from each point list twice and computed their difference twice, or we would have had to define a function SQARE to compute the square of the difference. LET is more appropriate when we won't be using the auxilary function anywhere else.

With LET, we can now redefine NREVERSE in a clearer manner as

```
(DEFUN NREVERSE (X)
    (NREVERSE-AUX X 'NIL))

(DEFUN NREVERSE-AUX (REM SOFAR)
    (IF (NULL REM)
        SOFAR
        (LET ((NEWREM (CDR REM)))
             (PROGN (RPLACD REM SOFAR)
                    (NREVERSE-AUX NEWREM REM)))))
```

The essential sequential ordering of the computations is now quite explicit.

Now consider a third example. Suppose that we have a list of names of people that could be surnames or first-name/surname pairs, so that it might look as follows:

```
((BILL BROWN) EINSTEIN NEWTON (ADA LOVELACE)
 (JAN BROWN) BULLWINKLE)
```

Suppose further that we want to pass each name to a function PROCESS-NAME that takes two arguments—a surname and a first name—but the latter can be NIL. Then we could write something similar to

```
(DEFUN PROCESS-NAME (SURNAME FIRSTNAME)
  ...)

(DEFUN PROCESS-LIST-OF-NAMES (LIST)
  (IF (NULL LIST)
      'T
      (LET ((NAME (CAR LIST)))
           (LET ((NAME (IF (ATOM NAME)
                           NAME
                           (CAR (CDR NAME))))
                 (FNAME (IF (ATOM NAME)
                            'NIL
                            (CAR NAME))))
```

```
(PROGN (PROCESS-NAME NAME FNAME)
       (DO-SOMETHING-ELSE NAME FNAME)
       (PROCESS-LIST-OF-NAMES (CDR LIST)))))))
```

to feed PROCESS-NAME. Notice that there are two variables called NAME in this example, one in each LET. This is poor practice because it makes the program harder to read and debug, but we will use this example to illustrate a point. The value for FNAME is computed after the value for the inner NAME has been computed, but references to the variable NAME within FNAME's value are to the first, or outer, variable NAME. This is because the new variable NAME is not available until all of the values of all of the variables in the inner LET have been computed.

A variation on LET is LET*. It sequentially binds each new variable as its value is computed, and so later values can use variables introduced by the current LET*. Thus, for example, the following two functions are equivalent, but the latter suffers less from "right-crawl."

```
(DEFUN PAINT-COST (COLOR)
    (LET ((PAIR (ASSOC COLOR
                       '((BLUE . 8.00)
                         (RED . 5.50)
                         (CHARTREUSE . 13.25)))))
      (LET ((PRICE (IF (NULL PAIR)
                       *DEFAULT-PAINT-PRICE*
                       (CDR PAIR))))
        (+ PRICE
           (* *TAX-RATE* PRICE)))))
```

```
(DEFUN PAINT-COST (COLOR)
    (LET* ((PAIR (ASSOC COLOR
                        '((BLUE . 8.00)
                          (RED . 5.50)
                          (CHARTREUSE . 13.25))))
           (PRICE (IF (NULL PAIR)
                      *DEFAULT-PAINT-PRICE*
                      (CDR PAIR))))
      (+ PRICE
         (* *TAX-RATE* PRICE))))
```

Notice how the convention of starting and ending global variables with an "*" makes this function much easier to understand; there is no need to search for a variable binding for such variables.

EXERCISES

E3.3.1 What do each of these evaluate to?

a.
```
(LET ((A 3))
     (CONS A
           (LET ((A 4))
                A)))
```

b.
```
(LET ((A 3))
     (LET ((A 4)
           (B A))
          (CONS A B)))
```

c.
```
(LET ((A 3))
     (LET* ((A 4)
            (B A))
           (CONS A B)))
```

3.4 DEFUN, COND AND LET HAVE IMPLICIT PROGNS

In chapter 2, we saw the special form PROGN, which looked like:

(PROGN *form-1 form-2 ... form-n*)

It evaluates each of its argument forms in turn and returns the value of the last as the value of the PROGN. This special form is only useful if at least all but the last arguments have side effects. It provides a method for sequencing forms with side effects.

A number of Lisp constructs save the user from having to type an explicit PROGN by allowing sequences of forms in their bodies; this is known as "having an *implicit* PROGN." In particular, the bodies of DEFUN and LET are implicit PROGNs—they each can be a series of statements that are evaluated in order with the result of the last being returned as the result of the enclosing form. Thus, the following three functions have equivalent behavior (although the third is a somewhat strange way to go about it):

```
(DEFVAR *COUNT* 0)

(DEFUN COUNT-CONS-1 (X Y)
    (PROGN (SETQ *COUNT* (+ 1 *COUNT*))
           (CONS X Y)))
```

```
(DEFUN COUNT-CONS-2 (X Y)
    (SETQ *COUNT* (+ 1 *COUNT*))
    (CONS X Y))


(DEFUN COUNT-CONS-3 (X Y)
    (LET ((CONS-CELL (CONS 'NIL 'NIL)))
         (RPLACA CONS-CELL X)
         (RPLACD CONS-CELL Y)
         (SETQ *COUNT* (+ 1 *COUNT*))
         CONS-CELL))
```

The special form COND is also more general than previously indicated. Each of its clauses may contain an implicit PROGN. Thus, the general form looks like

```
(COND (test-1 form-1-a form-1-b ...result-1)
      (test-2 form-2-a form-2-b ...result-2)
      ...)
```

If a test form evaluates to non-NIL, each of the following forms is evaluated and the result of the last is returned as the result of the COND.

3.5 COMMENTING CODE

Comments can make programs understandable. They can describe the expected class of inputs to a function and the resulting outputs. They can explain how clever algorithms really do produce the claimed result. They can describe the assumptions about, and effects on, global data structures that the function alters.

Comments can be easily interspersed with Lisp code. Whenever Lisp sees a semicolon, it ignores it and everything else until the next new line. The following conventions are adhered to by many Lisp programmers and are used throughout this book. In addition, many automated programming aids (such as editors) understand about these conventions and make some inferences from them in choosing the best way to display a program on a screen or in a listing. There are other sets of conventions used by various groups of programmers—it is important to use a consistent set of conventions within a project.

Our conventions are as follows:

1. Any s-expression that is top level within a file is preceded and followed by a blank line.

2. If a comment is outside the scope of a function three semicolons are used, flush against the left margin.
3. If a comment is within a function but on a line of its own, two successive semicolons are used, and they are indented to the same place a piece of Lisp code would logically appear.
4. When a comment is placed on a line that also contains Lisp code, at least one space is left after the Lisp code, a single semicolon is used, and the comment follows. Often editors will ensure that all such comments have their semicolons line up on a particular column on the page. Another possibility is to use a tab because these are ignored by Lisp.
5. A much more optional convention is that Lisp code be typed in uppercase lettering[4] and that comments appear in lowercase lettering with the usual capitalization for punctation.[5]

Here is a function from section 3.2 with comments written according to these conventions:

```
;;; Given a paint color compute the selling price of a
;;; standard can of paint, including the tax.


(DEFUN PAINT-COST (COLOR)
  ;; a few colors have special prices
  (LET* ((PAIR (ASSOC
                 COLOR
                 '((BLUE . 8.00)
                   (RED . 5.50)
                   (CHARTREUSE . 13.25)))) ;at this price!!
         (PRICE (IF (NULL PAIR)
                    ;; this should be SETQed before use
                    *DEFAULT-PAINT-PRICE*
                    (CDR PAIR)))) ;get the price from the cons
    (+ PRICE
       ;; tax rate is a fraction of 1.00
       (* *TAX-RATE* PRICE))))
```

[4]This does not work in Franz Lisp, which is case-sensitive; so while CAR is undefined, car works as expected.

[5]The ZWEI editor for Lisp machines has an ELECTRIC-SHIFT-LOCK mode that detects whether you are typing Lisp code or comments and automatically sets the case of type-in to follow this convention. It is even smart enough to handle strings correctly within Lisp code.

3.6 DO: A SPECIAL FORM FOR ITERATION

We have seen many examples of functions that can be defined recursively. Often, however, the structure of a computation has a much more natural expression as an iterative procedure. The primitive for iteration within Common Lisp is DO.[6] The key property of DO is that it involves multiple iteration variables that are all stepped according to their own stepping rules in parallel. It takes the general form

```
(DO ((var-1 init-1 stepper-1)
     (var-2 init-2 stepper-2)
     .
     .
     .
     (var-n init-n stepper-n))
    (end-test
     end-form-1
     .
     .
     .
     end-form-k
     return-value)
  body-1
  body-2
  .
  .
  .
  body-m)
```

It looks something like a LET with a set of variables—and values to which they get bound—all in parallel. In fact, the operation up through the binding of each *var-i* to the value of each *init-i* is identical to that of LET. Then the *end-test* is evaluated. If it returns non-NIL, each of the optional *end-form-j*'s are evaluated and their values discarded. Finally, the *return-value* is evaluated and its value is returned as the value of the DO. Thus, there is a COND-clause like structure following the variable bindings. If the *end-test* returns NIL, the DO continues. Each of its *body-1* expressions is evaluated in turn. If any expression of the form

[6] DO was inherited from Maclisp and all Maclisp descendants have left it intact. Thus, the Common Lisp definition is quite commonly accepted, which is quite surprising for the complexity of its description; usually such large organisms mutate quickly.

(RETURN *value*) is evaluated, the DO is exited and *value* is returned as its value. If not, the iteration is complete. The next iteration starts by evaluating each of the *stepper* forms in terms of the current bindings of the DO variables, and once they are all done the variables are all bound to the new values in parallel. Now the iteration continues as before by evaluating the end test and then the body.

As an example, let's redefine iteratively a Lisp function that we have studied recursively.

```
(DEFUN LENGTH (X)
    (DO ((LST X (CDR LST))
         (COUNT 0 (+ 1 COUNT)))
        ((NULL LST)
         COUNT)))
```

There is a no body of the DO in this particular example—it is optional.

Two variables are introduced by the DO: LST and COUNT. They get bound to the argument list and 0, respectively, and they are each stepped in parallel. At each iteration, LST is stepped another CDR down the list, while COUNT is increased by one. When the end of the argument list is reached, the DO is exited with the current value of COUNT. Thus, the following behavior happens internally:

```
eval> (LENGTH '(A B))

        LST = (A B)
        COUNT = 0
        (NULL LST) = (NULL '(A B)) = NIL

        LST = (CDR LST) = (B)
        COUNT = (+ 1 COUNT) = 1
        (NULL LST) = (NULL '(B) = NIL

        LST = (CDR LST) = NIL
        COUNT = (+ 1 COUNT) = 2
        (NULL LST) = (NULL 'NIL) = T

2
```

Notice that the end test is performed *after* the variables have received their new bindings.

Now compare an iterative formulation of the function REVERSE with our previous recursive formulation.

```
;;; iterative formulation

(DEFUN I-REVERSE (LST)
    (DO ((REM LST (CDR REM))
         (SOFAR 'NIL (CONS (CAR REM) SOFAR)))
        ((NULL REM)
         SOFAR)))


;;; recursive formulation


(DEFUN R-REVERSE (LST)
    (R-REVERSE-AUX LST 'NIL))

(DEFUN R-REVERSE-AUX (REM SOFAR)
    (IF (NULL REM)
        SOFAR
        (R-REVERSE-AUX (CDR REM) (CONS (CAR REM) SOFAR))))
```

The variables REM and SOFAR play identical roles in the two versions. The function R-REVERSE does the same as the initial forms in the DO of I-REVERSE. The recursive calls to R-REVERSE are equivalent to supplying the stepping values in the DO.[7]

The next example shows a DO-loop whose controlling variable is an integer. The variable I is stepped from N down to 1, and the iteration stops as soon as it stepped past there to 0.

```
eval> (DEFUN MAKE-ASCENDING-LIST (N)
          (DO ((I N (- I 1))
               (LST '()
                    (CONS I LST)))
              ((ZEROP I)
               LST)))
MAKE-ASCENDING-LIST
```

[7] In fact, a good Lisp compiler should compile almost the same code for the recursive R-REVERSE-AUX and the new iterative I-REVERSE. Using a technique called *tail recursion* analysis, the compiler should recognize that R-REVERSE-AUX is a recursive implementation of an inherently iterative process. Even a good Lisp compiler will not recognize the iterativeness of our previous recursive definition of LENGTH. The foregoing iterative definition (of LENGTH) does, however, suggest another recursive implementation of LENGTH using an auxiliary function in the spirit of REVERSE-AUX. Can you see it?


```
eval> (MAKE-ASCENDING-LIST 4)
(1 2 3 4)
```

So far, all the examples have essentially used one variable as a driver for the iteration, stopping on some condition met by the single variable. That is by no means always the case, as the following example shows. In this case, the DO has a body. (Notice that a DO body makes sense only if it has side effects or uses RETURN.)

```
(DEFUN GET-LONGEST (LIST1 LIST2)
    (DO ((REM1 LIST1 (CDR REM1))
         (REM2 LIST2 (CDR REM2)))
        (NIL)
      (COND ((NULL REM1) (RETURN LIST2))
            ((NULL REM2) (RETURN LIST1)))))
```

In GET-LONGEST, the end test of the DO is NIL, so it can only be exited by a RETURN from the body. Care must be taken in using NIL as an end test to ensure that there is a guaranteed way that the DO will eventually be exited from its body. Notice that an end test is required, even if it is NIL.

A common habit of many DO users is to use stepping variable names that are already the names of variables containing a value to be iterated upon. Thus, they would write our iterative REVERSE as

```
(DEFUN REVERSE (LST)
    (DO ((LST LST (CDR LST))
         (NEW 'NIL (CONS (CAR LST) NEW)))
        ((NULL LST)
         NEW)))
```

The stepping variable LST is a new variable introduced by the DO. It is initialized to the value of the outer LST; the one used as a function argument. All variable references in initialization forms are to variables introduced outside the scope of the DO. Many other programmers (including the author) do not like this particular style of variable naming. They find it confusing to have two variables with the same name, especially since they do start out referring to the same thing but then quickly diverge.

It is permissible to leave out a stepper form for any variable. In that case, the variable is initialized but its value is not changed at the time all other variables are stepped. Stylistically, it might be clearer to delete such variables from the DO and have them initialized in a LET wrapped around it.

EXERCISES

E3.6.1 Evaluate each of the following s-expressions:

a.
```
(LET ((COUNT 0)) ;gets SETQed below
     (DO ((LST '(A B C D E) (CDR LST)))
         ((NULL LST)
          COUNT)
         (SETQ COUNT (+ 1 COUNT))))
```

b.
```
(LET ((COUNT 0)) ;gets SETQed below
     (DO ((I 3 (+ 1 I)))
         ((EQL I 9)
          COUNT)
         (SETQ COUNT (+ 1 COUNT))))
```

c.
```
(DO ((LST1 '(A B C D E) (CDR LST1))
     (LST2 '(E D C B A) (CDR LST2))
     (COUNT 0 (+ 1 COUNT)))
    ((EQ (CAR LST1) (CAR LST2))
     COUNT))
```

E3.6.2 What does this function do?

```
(DEFUN ANON (LIST)
    (DO ((LST (CDR LIST) (CDR LST))
         (HEAD (CAR LIST) (CAR LST))
         (LASTHEAD LIST HEAD))
        ((NULL LST)
         'NIL)
        (IF (EQ HEAD LASTHEAD)
            (RETURN LST))))
```

SUMMARY

The special forms examined in this chapter are shown in Table 3-1.

Special form COND provides a more general test and branch control structure than does IF. Forms AND and OR evaluate just enough of their argument subforms to determine whether the combination is logically true under conjunction or disjunction. They sometimes make the intent of conditional evaluations more clear.

The special forms LET and LET* are used to introduce new variables and to give them initial values. The first binds all of its variables in parallel, while the second does them sequentially. The special form DO also introduces variables and gives them initial values in parallel. Then it iteratively evaluates body statements and gives all the variables new values, again in parallel. End tests

Table 3-1

Special Form	*Subforms*	*Description*
COND	$1 \Rightarrow \infty$	conditionally evaluate expressions
AND	$0 \Rightarrow \infty$	conditionally evaluate logical conjunction
OR	$0 \Rightarrow \infty$	conditionally evaluate logical disjunction
LET	$2 \Rightarrow \infty$	parallel binding of variables
LET*	$2 \Rightarrow \infty$	sequential binding of variables
DO	$2 \Rightarrow \infty$	iteration primitive

and explicit returns can terminate the iteration. For all three special forms, the introduced variables remain "in force" only within, or during, the evaluation of the special form. The variables disappear on exit, and, furthermore, are not accessible from functions called within the form. The *scope* of the introduced variable is the special form.

When a variable is referenced in a Lisp function, the Lisp system must determine its value. It looks outwards through s-expressions of increasing scope, that contain the variable reference. The first such s-expression that is a special form binding a variable of the desired name determines the variable that is being referenced. The initialization forms for variables in LET, LET*, and DO, while structurally within the scope of their special forms, are treated as lying lexically outside the forms. The only other variable introducing form we have seen is function definition; the argument list lists arguments whose scope is the whole function. The foregoing class of variables are referred to as *lexical* or *local* variables. If no variable is found within the scope of the function to match a variable reference, the reference is assumed to be to a global variable. All global variables are accessible to all functions, except where their name has been shadowed by a newly introduced lexical variable.

PROBLEMS

P3.1 Rewrite function EQUAL-SYMBOLS from problem 2.4 using some of the special forms introduced in this chapter.

P3.2 Some Lisps provide TCONC structures, which make it easy to add elements to the end of a list. A TCONC is the cons of a list and its last cons cell. This cons cell is then modified as elements are added to the end of the underlying list and removed from the front—thus TCONCs provide a first-in first-out structure. Write the following functions:

- (MAKE-TCONC) Makes a fresh and empty TCONC; that is, (NIL).
- (TCONC *tconc element*) Adds an element to an existing TCONC.
- (LCONC *tconc list*) Adds a list of elements to an existing TCONC.
- (REMOVE-HEAD *tconc*) Removes the first element from a TCONC and returns it.

All but the last of these functions should return the TCONC itself. The most difficult part about writing these functions is dealing with the cases of emptiness—in particular, a null list to be LCONCed, an empty TCONC in any of the last three cases, or a TCONC with only one element in the last case. For example,

```
eval> (SETQ TC (MAKE-TCONC))
(NIL)

eval> (TCONC TC 'A)
((A) A)

eval> (REMOVE-HEAD TC)
A

eval> TC
(NIL)

eval> (TCONC TC 'B)
((B) B)

eval> (TCONC TC 'C)
((B C) C)

eval> (LCONC TC 'NIL)
((B C) C)

eval> (LCONC TC '(D E F))
((B C D E F) F)

eval> (REMOVE-HEAD TC)
B

eval> TC
((C D E F) F)
```

P3.3 Why are TCONCs more efficient than ordinary lists for these operations?

P3.4 Define an iterative function RANGE that takes a list of numbers (at least one long) and returns a list of length 2 of the smallest and largest numbers. Recall predicates < and > from section 1.9. Function RANGE should have the following behavior:

```
eval> (RANGE '(0 7 8 2 3 -1))
(-1 8)
```



```
eval> (RANGE '(7 6 5 4 3))
(3 7)
```

Be sure that your solution only iterates down the list once.

P3.5 Write another function, VALID-RANGE, that returns the same result as RANGE if all elements of the argument list are numbers, but that returns the atom INVALID if the list contains a nonnumber.

P3.6 Write a function that finds the maximum number that occurs at any depth in a list. For example,

```
eval> (MAX-NO '((A 3) (B (C 4.5)) 3.2))
4.5
```

P3.7 The sequence of numbers 0, 1, 1, 2, 3, 5, 8, 13 ... are known as the Fibonacci numbers, where the fourth Fibonacci number is 5 and the fifth is 8, and so on. Each Fibonacci number is the sum of the previous two. If we let $F(n)$ represent the nth Fibonacci number, $F(0) = 0$, $F(1) = 1$, and $F(n) = F(n-1) + F(n-2)$. Write a recursive definition RECF to compute Fibonacci numbers. Then write an iterative version. Compare your recursive and iterative implementations by trying them for $n = 16, 17, 18$, or so. What's the difference and why?

Not Now