4

Conditionals

4.1 INTRODUCTION

Decision making is a fundamental part of computing; all nontrivial programs make decisions. In this chapter we will study some special decision-making functions, called **conditionals**, that choose their result from among a set of alternatives based on the value of one or more **predicate expressions**. (A predicate expression is an expression whose value is interpreted as either ‘‘true’’ or ‘‘false.’’)

Conditionals allow functions to vary their behavior for different sorts of inputs. Since we can construct our own predicate expressions to control these conditionals, we can write functions that make arbitrarily complex decisions.

4.2 THE IF SPECIAL FUNCTION

IF is the simplest Lisp conditional. Conditionals are always macros or special functions,* so their arguments do not get evaluated automatically. DEFUN and QUOTE are two other function we’ve studied with this property. Ordinary functions, like + and CONS, always evaluate their arguments.

*This terminology was suggested by Robert Wilensky. The distinction between ‘‘macro’’ functions and ‘‘special’’ functions is explained in Chapter 14; for now you can think of them as the same.

The IF special function takes three arguments: a **test**, a **true-part**, and a **false-part**. If the test is true, IF returns the value of the true-part. If the test is false, it skips the true-part and instead returns the value of the false-part. Here are some examples.

```
(if (oddp 1) 'odd 'even)  ⇒  odd

(if (oddp 2) 'odd 'even)  ⇒  even

(if t 'test-was-true 'test-was-false)  ⇒
   test-was-true

(if nil 'test-was-true 'test-was-false)  ⇒
   test-was-false

(if (symbolp 'foo) (* 5 5) (+ 5 5))  ⇒  25

(if (symbolp 1) (* 5 5) (+ 5 5))  ⇒  10
```

Let's use IF to construct a function that takes the absolute value of a number. Absolute values are always nonnegative. For negative numbers the absolute value is the negation of the number; for positive numbers and zero the absolute value is the number itself. This leads to a simple definition for MY-ABS, our absolute value function. (We call the function MY-ABS rather than ABS because there is already an ABS function built in to Common Lisp; we don't want to interfere with any of Lisp's built-in functions.)

```
(defun my-abs (x)
  (if (< x 0) (- x) x))
```

The test part of the IF is the expression (< X 0). If the test evaluates to true, the true-part, (- X), will be evaluated and will return the negation of X. If the test evaluates to false, meaning X is zero or positive, the false-part of the IF will be evaluated. The false-part is just X, so the input to MY-ABS will be returned unchanged in this case. Here is how you should be reading the definition of MY-ABS: ''DEFUN MY-ABS of X: IF (< X 0) then minus X else X.'' The words ''then'' and ''else'' don't actually appear in the function, but mentally inserting them can help to clarify the function in your mind.

```
> (my-abs -5)      True-part takes the negation.
5

> (my-abs 5)       False-part returns the number unchanged.
5
```

Here's another simple decision-making function. SYMBOL-TEST returns a message telling whether or not its input is a symbol.

```
(defun symbol-test (x)
  (if (symbolp x) (list 'yes x 'is 'a 'symbol)
      (list 'no x 'is 'not 'a 'symbol)))
```

When you read this function definition to yourself, you should read the IF part as ''If SYMBOLP of X then...else....''

```
> (symbol-test 'rutabaga)          Evaluate true-part.
(YES RUTABAGA IS A SYMBOL)

> (symbol-test 12345)              Evaluate false-part.
(NO 12345 IS NOT A SYMBOL)
```

IF can be given two inputs instead of three, in which case it behaves as if its third input (the false-part) were the symbol NIL.

```
(if t 'happy)  ⇒  happy

(if nil 'happy)  ⇒  nil
```

EXERCISES

4.1. Write a function MAKE-EVEN that makes an odd number even by adding one to it. If the input to MAKE-EVEN is already even, it should be returned unchanged.

4.2. Write a function FURTHER that makes a positive number larger by adding one to it, and a negative number smaller by subtracting one from it. What does your function do if given the number 0?

4.3. Recall the primitive function NOT: It returns NIL for a true input and T for a false one. Suppose Lisp didn't have a NOT primitive. Show how to write NOT using just IF and constants (no other functions). Call your function MY-NOT.

4.4. Write a function ORDERED that takes two numbers as input and makes a list of them in ascending order. (ORDERED 3 4) should return the list (3 4). (ORDERED 4 3) should also return (3 4), in other words, the first and second inputs should appear in reverse order when the first is greater than the second.

4.3 THE COND MACRO

COND is the classic Lisp conditional. Its input consists of any number of test-and-consequent **clauses**. The general form of a COND expression will be described in Chapter 5, but a slightly simplified form is:

```
(COND (test-1 consequent-1)
      (test-2 consequent-2)
      (test-3 consequent-3)
            ....
      (test-n consequent-n))
```

Here is how COND works: It goes through the clauses sequentially. If the test part of a clause evaluates to *true*, COND evaluates the consequent part and returns its value; it does not consider any more clauses. If the test evaluates to *false*, COND skips the consequent part and examines the next clause. If it goes through all the clauses without finding any whose test is true, CONS returns NIL.

Let's use COND to write a function COMPARE that compares two numbers. If the numbers are equal, COMPARE will say ''numbers are the same''; if the first number is less than the second, it will say ''first is smaller''; if the first number is greater than the second, it will say ''first is bigger.'' Each case is handled by a separate COND clause.

```
(defun compare (x y)
  (cond ((equal x y) 'numbers-are-the-same)
        ((< x y) 'first-is-smaller)
        ((> x y) 'first-is-bigger)))
```

Take a closer look at the COND. It is a four-element list, where the first element is the symbol COND and the remaining three elements are test-and-consequent clauses. The first clause is a two-element list whose first element is the expression (EQUAL X Y). This is the test part of the clause. The second element, the consequent part, is the quoted symbol 'NUMBERS-ARE-THE-SAME.

Here are some examples of the COMPARE function:

```
(compare 3 5)  ⇒  first-is-smaller

(compare 7 2)  ⇒  first-is-bigger

(compare 4 4)  ⇒  numbers-are-the-same
```

EXERCISE

4.5. For each of the following calls to COMPARE, write ‘‘1,’’ ‘‘2,’’ or ‘‘3’’ to indicate which clause of the COND will have a predicate that evaluates to true.

```
_____   (compare 9 1)

_____   (compare (+ 2 2) 5)

_____   (compare 6 (* 2 3))
```

COND and IF are similar functions. COND may appear more versatile since it accepts any number of clauses, but there is a way to do the same thing with nested IFs. This is explained later in the chapter.

4.4 USING T AS A TEST

One of the standard tricks for using COND is to include a clause of form

```
(T consequent)
```

The test T is always true, so if COND ever reaches this clause, it is guaranteed to evaluate the consequent. We put this clause at the very end so that it will be reached only if all the preceding clauses' tests fail. *Example:* The following function returns the country in which a given city is. If the function doesn't know a particular city, it returns the symbol UNKNOWN.

```
(defun where-is (x)
  (cond ((equal x 'paris) 'france)
        ((equal x 'london) 'england)
        ((equal x 'beijing) 'china)
        (t 'unknown)))
```

Note that the last clause of the COND begins with T. If none of the preceding clauses have tests that return true, the last clause will be reached and the function will return UNKNOWN.

```
(where-is 'london)   ⇒  england

(where-is 'beijing)  ⇒  china

(where-is 'hackensack)  ⇒  unknown
```

Recall that the general form of an IF expression is

```
(IF test true-part false-part)
```

We can translate any IF expression into a COND expression using two clauses:

```
(COND (test true-part)
      (T false-part))
```

EXERCISE

4.6. Write a version of the absolute value function MY-ABS using COND instead of IF.

4.5 TWO MORE EXAMPLES OF COND

Here is another function, called EMPHASIZE, that changes the first word of a phrase from ‘‘good’’ to ‘‘great,’’ or from ‘‘bad’’ to ‘‘awful,’’ and returns the modified phrase:

```
(defun emphasize (x)
  (cond ((equal (first x) 'good) (cons 'great (rest x)))
        ((equal (first x) 'bad) (cons 'awful (rest x)))))
```

Let’s take as an example the phrase (GOOD MYSTERY STORY). What happens inside EMPHASIZE? The variable X is assigned the value (GOOD MYSTERY STORY), and COND starts going through the test-and-consequent clauses. The first one is:

```
((equal (first x) 'good) (cons 'great (rest x)))
```

Since (FIRST X) evaluates to GOOD, the test part of this clause is true. The consequent part then constructs a new list from the symbol GREAT and the REST of the input, and that is what the function returns:

```
(emphasize '(good mystery story))
  ⇒  (great mystery story)
```

Now suppose we try to emphasize (MEDIOCRE MYSTERY STORY). The first clause compares MEDIOCRE to GOOD and returns NIL. The next compares MEDIOCRE to BAD and also returns NIL. Now COND has run out of clauses, so it returns NIL. Therefore, NIL is the result of the EMPHASIZE function:

```
(emphasize '(mediocre mystery story))  ⇒  nil
```

What if we want EMPHASIZE to return the original input instead of NIL

when it can't figure out how to emphasize it? We simply use the T-as-test trick, demonstrated in the function EMPHASIZE2:

```
(defun emphasize2 (x)
  (cond ((equal (first x) 'good) (cons 'great (rest x)))
        ((equal (first x) 'bad) (cons 'awful (rest x)))
        (t x)))
```

If the COND reaches the last clause, the test T is guaranteed to evaluate to true and the input, X, is returned.

```
(emphasize2 '(good day))  ⇒  (great day)

(emphasize2 '(bad day))  ⇒  (awful day)

(emphasize2 '(long day))  ⇒  (long day)
```

Here is a function COMPUTE that takes three inputs. If the first input is the symbol SUM-OF, the function returns the sum of the second and third inputs. If it is the symbol PRODUCT-OF, the function returns the product of the second and third inputs. Otherwise it returns the list (THAT DOES NOT COMPUTE).

```
(defun compute (op x y)
  (cond ((equal op 'sum-of) (+ x y))
        ((equal op 'product-of) (* x y))
        (t '(that does not compute))))
```

Here are some examples of the COMPUTE function:

```
(compute 'sum-of 3 7)  ⇒  10

(compute 'product-of 2 4)  ⇒  8

(compute 'zorch-of 3 1)
  ⇒  (that does not compute)
```

4.6 COND AND PARENTHESIS ERRORS

Parenthesis errors can play havoc with COND expressions. Most COND clauses begin with exactly two parentheses. The first marks the beginning of the clause, and the second marks the beginning of the clause's test. For example, in the WHERE-IS function, the test part of the first clause is the expression

```
(EQUAL X 'PARIS)
```

so the clause itself looks like

```
((EQUAL X 'PARIS) . . .)
```

If the test part of a clause is just a symbol, not a call to a function, then the clause should begin with a single parenthesis. Notice that in WHERE-IS the clause with T as the test begins with only one parenthesis.

Here are two of the more common parenthesis errors made with COND. First, suppose we leave a parenthesis out of a COND clause. What would happen?

```
(cond (equal x 'paris 'france)
      (. . .)
      (. . .)
      (t 'unknown))
```

The first clause of the COND starts with only one left parenthesis instead of two. As a result, the test part of this clause is just the symbol EQUAL. When the test is evaluated, EQUAL will cause an unassigned variable error.

On the other hand, consider what happens when too many parentheses are used:

```
(cond ((. . .) 'france)
      ((. . .) 'england)
      ((. . .) 'china)
      ((t 'unknown)))
```

If X has the value HACKENSACK, we will reach the fourth COND clause. Due to the presence of an extra pair of parentheses in this clause, the test is (T 'UNKNOWN) instead of simply T. T is not a function, so this test will generate an undefined function error.

EXERCISES

4.7. For each of the following COND expressions, tell whether the parenthesization is correct or incorrect. If incorrect, explain where the error lies.

```
(cond (symbolp x) 'symbol
      (t 'not-a-symbol))

(cond ((symbolp x) 'symbol)
      (t 'not-a-symbol))
```

```
(cond ((symbolp x) ('symbol))
      (t 'not-a-symbol))

(cond ((symbolp x) 'symbol)
      ((t 'not-a-symbol)))
```

4.8. Write EMPHASIZE3, which is like EMPHASIZE2 but adds the symbol VERY onto the list if it doesn't know how to emphasize it. For example, EMPHASIZE3 of (LONG DAY) should produce (VERY LONG DAY). What does EMPHASIZE3 of (VERY LONG DAY) produce?

4.9. Type in the following suspicious function definition:

```
(defun make-odd (x)
  (cond (t x)
        ((not (oddp x)) (+ x 1))))
```

What is wrong with this function? Try out the function on the numbers 3, 4, and -2. Rewrite it so it works correctly.

4.10. Write a function CONSTRAIN that takes three inputs called X, MAX, and MIN. If X is less than MIN, it should return MIN; if X is greater than MAX, it should return MAX. Otherwise, since X is between MIN and MAX, it should return X. (CONSTRAIN 3 -50 50) should return 3. (CONSTRAIN 92 -50 50) should return 50. Write one version using COND and another using nested IFs.

4.11. Write a function FIRSTZERO that takes a list of three numbers as input and returns a word (one of ''first,'' ''second,'' ''third,'' or ''none'') indicating where the first zero appears in the list. Example: (FIRSTZERO '(3 0 4)) should return SECOND. What happens if you try to call FIRSTZERO with three separate numbers instead of a list of three numbers, as in (FIRSTZERO 3 0 4)?

4.12. Write a function CYCLE that cyclically counts from 1 to 99. CYCLE called with an input of 1 should return 2, with an input of 2 should return 3, with an input of 3 should return 4, and so on. With an input of 99, CYCLE should return 1. That's the cyclical part. Do not try to solve this with 99 COND clauses!

4.13. Write a function HOWCOMPUTE that is the inverse of the COMPUTE function described previously. HOWCOMPUTE takes three numbers as input and figures out what operation would produce the third from the first two. (HOWCOMPUTE 3 4 7) should return SUM-OF.

(HOWCOMPUTE 3 4 12) should return PRODUCT-OF. HOWCOMPUTE should return the list (BEATS ME) if it can't find a relationship between the first two inputs and the third. Suggest some ways to extend HOWCOMPUTE.

4.7 THE AND AND OR MACROS

We will often need to construct complex predicates from simple ones. The AND and OR macros make this possible. Before giving the precise rules for evaluating AND and OR, let's just look at an example. Suppose we want a predicate for small (no more than two digit) positive odd numbers. We can use AND to express this conjunction of simple conditions:

```
(defun small-positive-oddp (x)
  (and (< x 100)
       (> x 0)
       (oddp x)))
```

Or suppose we want a function GTEST that takes two numbers as input and returns T if either the first is greater than the second or one of them is zero. These conditions form a disjunctive set; only one need be true for GTEST to return T. OR is used for disjunctions.

```
(defun gtest (x y)
  (or (> x y)
      (zerop x)
      (zerop y)))
```

Like COND, AND and OR are macros: they can accept any number of clauses, and they do not evaluate their arguments first. For AND and OR, however, the clauses are simply tests, not test-and-consequent pairs.

4.8 EVALUATING AND AND OR

AND and OR have slightly different meanings in Lisp than they do in logic or in English. The precise rule for evaluating AND is: Evaluate the clauses one at a time. If a clause returns NIL, stop and return NIL; otherwise go on to the next one. If all the clauses yield non-NIL results, return the value of the last clause. Examples:

```
(and nil t t)  ⇒  nil
```

```
(and 'george nil 'harry)  ⇒  nil

(and 'george 'fred 'harry)  ⇒  harry

(and 1 2 3 4 5)  ⇒  5
```

The rule for evaluating OR is: Evaluate the clauses one at a time. If a clause returns something other than NIL, stop and return that value; otherwise go on to the next clause, or return NIL if none are left.

```
(or nil t t)  ⇒  t

(or 'george nil 'harry)  ⇒  george

(or 'george 'fred 'harry)  ⇒  george

(or nil 'fred 'harry)  ⇒  fred
```

EXERCISE

4.14. What results do the following expressions produce? Read the evaluation rules for AND and OR carefully before answering.

```
(and 'fee 'fie 'foe)

(or 'fee 'fie 'foe)

(or nil 'foe nil)

(and 'fee 'fie nil)

(and (equal 'abc 'abc) 'yes)

(or (equal 'abc 'abc) 'yes)
```

4.9 BUILDING COMPLEX PREDICATES

The HOW-ALIKE function compares two numbers several different ways to see in what way they are similar. It uses AND to construct complex predicates as part of a COND clause:

```
(defun how-alike (a b)
  (cond ((equal a b) 'the-same)
        ((and (oddp a) (oddp b)) 'both-odd)
        ((and (not (oddp a)) (not (oddp b)))
         'both-even)
        ((and (< a 0) (< b 0)) 'both-negative)
        (t 'not-alike)))

(how-alike 7 7)  ⇒  the-same

(how-alike 3 5)  ⇒  both-odd

(how-alike -2 -3)  ⇒  both-negative

(how-alike 5 8)  ⇒  not-alike
```

The SAME-SIGN predicate uses a combination of AND and OR to test if its two inputs have the same sign:

```
(defun same-sign (x y)
  (or (and (zerop x) (zerop y))
      (and (< x 0) (< y 0))
      (and (> x 0) (> y 0))))
```

SAME-SIGN returns T if any of the inputs to OR returns T. Each of these inputs is an AND expression. The first one tests whether X is zero and Y is zero, the second tests whether X is negative and Y is negative, and the third tests whether X is positive and Y is positive. Examples:

```
(same-sign 0 0)  ⇒  t

(same-sign -3 -4)  ⇒  t

(same-sign 3 4)  ⇒  t

(same-sign -3 4)  ⇒  nil
```

EXERCISES

4.15. Write a predicate called GEQ that returns T if its first input is greater than or equal to its second input.

4.16. Write a function that squares a number if it is odd and positive, doubles it if it is odd and negative, and otherwise divides the number by 2.

4.17. Write a predicate that returns T if the first input is either BOY or GIRL

and the second input is CHILD, or the first input is either MAN or WOMAN and the second input is ADULT.

4.18. Write a function to act as referee in the Rock-Scissors-Paper game. In this game, each player picks one of Rock, Scissors, or Paper, and then both players tell what they picked. Rock ‘‘breaks’’ Scissors, so if the first player picks Rock and the second picks Scissors, the first player wins. Scissors ‘‘cuts’’ Paper, and Paper ‘‘covers’’ Rock. If both players pick the same thing, it’s a tie. The function PLAY should take two inputs, each of which is either ROCK, SCISSORS, or PAPER, and return one of the symbols FIRST-WINS, SECOND-WINS, or TIE. Examples: (PLAY ’ROCK ’SCISSORS) should return FIRST-WINS. (PLAY ’PAPER ’SCISSORS) should return SECOND-WINS.

4.10 WHY AND AND OR ARE CONDITIONALS

Why are AND and OR classed as conditionals instead of regular functions? The reason is that they are not required to evaluate every clause. If any clause of an AND returns NIL, or any clause of an OR returns non-NIL, none of the succeeding clauses get evaluated. This property can be valuable, because we may need to halt evaluation to avoid errors that would otherwise occur. For example, consider the POSNUMP predicate:

```
(defun posnump (x)
  (and (numberp x) (plusp x)))
```

POSNUMP returns T if its input is a number and is positive. The built-in PLUSP predicate can be used to tell if a number is positive, but if PLUSP is used on something other than a number, it signals a ‘‘wrong type input’’ error, so it is important to make sure that the input to POSNUMP is a number *before* invoking PLUSP. If the input isn’t a number, we must not call PLUSP.

Here is an incorrect version of POSNUMP:

```
(defun faulty-posnump (x)
  (and (plusp x) (numberp x)))
```

If FAULTY-POSNUMP is called on the symbol FRED instead of a number, the first thing it does is check if FRED is greater than 0, which causes a wrong type input error. However, if the regular POSNUMP function is called with input FRED, the NUMBERP predicate returns NIL, so AND returns NIL *without ever calling* PLUSP.

4.11 CONDITIONALS ARE INTERCHANGEABLE

Functions that use AND and OR can also be implemented using COND or IF, and vice versa. Recall the definition of POSNUMP:

```
(defun posnump (x)
  (and (numberp x) (> x 0)))
```

Here is a version of POSNUMP written with IF instead of AND:

```
(defun posnump-2 (x)
  (if (numberp x) (> x 0) nil))
```

This version of POSNUMP tests for a number first, and if the condition succeeds, the true-part of the IF evaluates (> X 0). If the number test fails, the false-part of the IF is NIL. Trace the evaluation of the function on paper with inputs like FRED, 7, and -2 to better understand how it works. Here is another version of POSNUMP, this time using COND:

```
(defun posnump-3 (x)
  (cond ((numberp x) (> x 0))
        (t nil)))
```

Let's look at another use of conditionals. This is the original version of WHERE-IS, using COND:

```
(defun where-is (x)
  (cond ((equal x 'paris) 'france)
        ((equal x 'london) 'england)
        ((equal x 'beijing) 'china)
        (t 'unknown)))
```

This COND has four clauses. We can write WHERE-IS using IF instead of COND by putting three IFs together. Such a construct is called a **nested if**.

```
(defun where-is-2 (x)
  (if (equal x 'paris) 'france
      (if (equal x 'london) 'england
          (if (equal x 'beijing) 'china
              'unknown))))
```

Suppose we call WHERE-IS-2 with the input BEIJING. As the evaltrace shows, the local variable X is assigned the value BEIJING, and the body is evaluated. The body of WHERE-IS-2 is a single IF whose test checks if X is equal to PARIS. It is not, so the IF evaluates its false-part. The false-part is also an IF, and this IF's test checks whether X is equal to LONDON. It is not, so the IF evaluates its own false-part—yet another IF. This third IF tests if X

is equal to BEIJING, which it is, so its true part evaluates to CHINA. The third IF returns CHINA, which is now the value of the false-part of the second IF so it returns CHINA, which is now the value of the false-part of the first IF so it returns CHINA as well. The result of (WHERE-IS-2 'BEIJING) is CHINA.

We can write another version of WHERE-IS using AND and OR. This version employs a simple two-level scheme rather than the more complex nesting required for IF.

```
(defun where-is-3 (x)
  (or (and (equal x 'paris) 'france)
      (and (equal x 'london) 'england)
      (and (equal x 'beijing) 'china)
      'unknown))
```

Let's evaluate (WHERE-IS-3 'LONDON). X is bound to LONDON, and OR starts going through its clauses looking for one that isn't NIL. The first clause is an AND expression; AND evaluates (EQUAL X 'PARIS) and gets a NIL result, so AND gives up and returns NIL. OR moves on to its second clause. This is also an AND expression; (EQUAL X 'LONDON) returns T, so

AND moves on to its next clause. 'ENGLAND evaluates to ENGLAND; AND has run out of clauses, so it returns the value of the last one. Since OR has found a non-NIL clause, OR now returns ENGLAND.

Since IF, COND, and AND/OR are interchangeable conditionals, you may wonder why Lisp has more than one. It's a matter of convenience. IF is the easiest to use for simple functions like absolute value. AND and OR are good for writing complex predicates like SMALL-POSITIVE-ODDP. COND is easiest to use when there are many tests, as in WHERE-IS and HOW-ALIKE. Choosing the right conditional for the job is part of the art of programming.

EXERCISES

4.19. Show how to write the expression (AND X Y Z W) using COND instead of AND. Then show how to write it using nested IFs instead of AND.

4.20. Write a version of the COMPARE function using IF instead of COND. Also write a version using AND and OR.

4.21. Write versions of the GTEST function using IF and COND.

4.22. Use COND to write a predicate BOILINGP that takes two inputs, TEMP and SCALE, and returns T if the temperature is above the boiling point of water on the specified scale. If the scale is FAHRENHEIT, the boiling point is 212 degrees; if CELSIUS, the boiling point is 100 degrees. Also write versions using IF and AND/OR instead of COND.

4.23. The WHERE-IS function has four COND clauses, so WHERE-IS-2 needs three nested IFs. Suppose WHERE-IS had eight COND clauses. How many IFs would WHERE-IS-2 need? How many ORs would WHERE-IS-3 need? How many ANDs would it need?

SUMMARY

Conditionals allow the computer to make decisions that control its behavior. IF is a simple conditional; its syntax is (IF condition true-part false-part). COND, the most general conditional, takes a set of test-and-consequent clauses as input and evaluates the tests one at a time until it finds a true one. It then returns the value of the consequent of that clause. If none of the tests are true, COND returns NIL.

AND and OR are also conditionals. AND evaluates clauses one at a time until one of them returns NIL, which AND then returns. If all the clauses evaluate to true, AND returns the value of the last one. OR evaluates clauses

until a non-NIL value is found, and returns that value. If all the clauses evaluate to NIL, OR returns NIL. AND and OR aren't considered predicates because they're not ordinary functions.

A useful programming trick when writing COND expressions is to place a list of form (T consequent) as the final clause of the COND. Since the test T is always true, the clause serves as a kind of catchall case that will be evaluated when the tests of all the preceding clauses are false.

An important feature of conditionals is their ability to not evaluate all of their inputs. This lets us prevent errors by protecting a sensitive expression with predicate expressions that can cause evaluation to stop. Conditionals can do this because they are either macros or special functions, not ordinary functions.

REVIEW EXERCISES

4.24. Why are conditionals important?

4.25. What does IF do if given two inputs instead of three?

4.26. COND can accept any number of clauses, but IF takes at most three inputs. How is it then that any function involving COND can be rewritten to use IF instead?

4.27. What does COND return if given *no* clauses, in other words, what does (COND) evaluate to?

4.28. We can usually rewrite an IF as a combination of AND plus OR by following this simple scheme: Replace (IF *test true-part false-part*) with the equivalent expression (OR (AND *test true-part*) *false-part*). But this scheme fails for the expression (IF (ODDP 5) (EVENP 7) 'FOO). Why does it fail? Suggest a more sophisticated way to rewrite IF as a combination of ANDs and ORs that does not fail.

FUNCTIONS COVERED IN THIS CHAPTER

Conditionals: IF, COND, AND, OR.

Predicate: PLUSP.