7

Applicative Programming

7.1 INTRODUCTION

The three programming styles we will cover in this book are **applicative** programming, **recursion**, and **iteration**. Many instructors prefer to teach recursion first, but I believe applicative programming is the easiest for beginners to learn. To accomodate everyone's taste, Chapters 7 and 8 have been made independent; they can be covered in either order.

Applicative programming is based on the idea that functions are data, just like symbols and lists are data, so one should be able to pass functions as inputs to other functions, and also return functions as values. The **applicative operators** we will study in this chapter are functions that take another function as input and apply it to the elements of a list in various ways. These operators are all built from a primitive function known as FUNCALL. In the Advanced Topics section we will write our own applicative operator, and also write a function that constructs and returns new functions.

7.2 FUNCALL

FUNCALL calls a function on some inputs. We can use FUNCALL to call the CONS function on the inputs A and B like this:

```
(funcall #'cons 'a 'b)  ⇒  (a . b)
```

The #' (or ''sharp quote'') notation is the correct way to quote a function in Common Lisp. If you want to see what the function CONS looks like in your implementation, try the following example in your Lisp:

```
> (setf fn #'cons)
#<Compiled-function CONS {6041410}>

> fn
#<Compiled-function CONS {6041410}>

> (type-of fn)
COMPILED-FUNCTION

> (funcall fn 'c 'd)
(C . D)
```

The value of the variable FN is a function object. TYPE-OF shows that the object is of type COMPILED-FUNCTION. So you see that functions and symbols are not the same. The symbol CONS serves as the name of the CONS function, but it is not the actual function. The relationship between functions and the symbols that name them is explained in Advanced Topics section 3.18.

Note that only ordinary functions can be quoted with #'. It is an error to quote a macro function or special function this way, or to quote a symbol with #' if that symbol does not name a function.

```
> #'if
Error: IF is not an ordinary function.

> #'turnips
Error: TURNIPS is an undefined function.
```

7.3 THE MAPCAR OPERATOR

MAPCAR is the most frequently used applicative operator. It applies a function to each element of a list, one at a time, and returns a list of the results. Suppose we have written a function to square a single number. By itself, this function cannot square a list of numbers, because * doesn't work on lists.

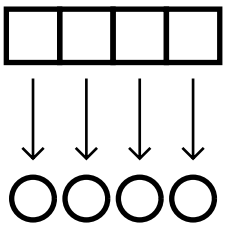
```
(defun square (n) (* n n))

(square 3)  ⇒  9
```

```
(square '(1 2 3 4 5))  ⇒  Error! Wrong type input to *.
```

With MAPCAR we can apply SQUARE to each element of the list individually. To pass the SQUARE function as an input to MAPCAR, we quote it by writing #'SQUARE.

```
>  (mapcar #'square '(1 2 3 4 5))
(1 4 9 16 25)

> (mapcar #'square '(3 8 -3 5 2 10))
(9 64 9 25 4 100)
```

Here is a graphical description of the MAPCAR operator. As you can see, each element of the input list is mapped independently to a corresponding element in the output.



When MAPCAR is used on a list of length *n*, the resulting list also has exactly *n* elements. So if MAPCAR is used on the empty list, the result is the empty list.

```
(mapcar #'square '())  ⇒  nil
```

7.4 MANIPULATING TABLES WITH MAPCAR

Suppose we set the global variable WORDS to a table of English and French words:

```
(setf words
  '((one un)
    (two deux)
    (three trois)
    (four quatre)
    (five cinq)))
```

We can perform several useful manipulations on this table with MAPCAR. We can extract the English words by taking the first component of each table entry:

```
> (mapcar #'first words)
(ONE TWO THREE FOUR FIVE)
```

We can extract the French words by taking the second component of each entry:

```
> (mapcar #'second words)
(UN DEUX TROIS QUATRE CINQ)
```

We can create a French–English dictionary from the English–French one by reversing each table element:

```
> (mapcar #'reverse words)
((UN ONE)
 (DEUX TWO)
 (TROIS THREE)
 (QUATRE FOUR)
 (CINQ FIVE))
```

Given a function TRANSLATE, defined using ASSOC, we can translate a string of English digits into a string of French ones:

```
(defun translate (x)
  (second (assoc x words)))

> (mapcar #'translate '(three one four one five))
(TROIS UN QUATRE UN CINQ)
```

Besides MAPCAR, there are several other applicative operators built in to Common Lisp. Many more are defined by programmers as they are needed, using FUNCALL.

EXERCISES

7.1. Write an ADD1 function that adds one to its input. Then write an expression to add one to each element of the list (1 3 5 7 9).

7.2. Let the global variable DAILY-PLANET contain the following table:

```
((olsen jimmy 123-76-4535 cub-reporter)
 (kent  clark 089-52-6787 reporter)
 (lane  lois  951-26-1438 reporter)
 (white perry 355-16-7439 editor))
```

Each table entry consists of a last name, a first name, a social security number, and a job title. Use MAPCAR on this table to extract a list of social security numbers.

7.3. Write an expression to apply the ZEROP predicate to each element of the list (2 0 3 4 0 -5 -6). The answer you get should be a list of Ts and NILs.

7.4. Suppose we want to solve a problem similar to the preceding one, but instead of testing whether an element is zero, we want to test whether it is greater than five. We can't use > directly for this because > is a function of two inputs; MAPCAR will only give it one input. Show how first writing a one-input function called GREATER-THAN-FIVE-P would help.

7.5 LAMBDA EXPRESSIONS

There are two ways to specify the function to be used by an applicative operator. The first way is to define the function with DEFUN and then specify it by #'*name*, as we have been doing. The second way is to pass the function definition directly. This is done by writing a list called a **lambda expression**. For example, the following lambda expression computes the square of its input:

```
(lambda (n) (* n n))
```

Since lambda expressions are functions, they can be passed directly to MAPCAR by quoting them with `#'`. This saves you the trouble of writing a separate DEFUN before calling MAPCAR.

```
> (mapcar #'(lambda (n) (* n n))  '(1 2 3 4 5))
(1 4 9 16 25)
```

Lambda expressions look similar to DEFUNs, except that the function name is missing and the word LAMBDA appears in place of DEFUN. But lambda expressions are actually unnamed functions. LAMBDA is not a macro or special function that has to be evaluated, like DEFUN. Rather it is a marker that says ''this list represents a function.''

Lambda expressions are especially useful for synthesizing one-input functions from related functions of two inputs. For example, suppose we wanted to multiply every element of a list by 10. We might be tempted to write something like:

```
(mapcar #'* '(1 2 3 4 5))
```

but where is the 10 supposed to go? The * function needs two inputs, but MAPCAR is only going to give it one. The correct way to solve this problem is to write a lambda expression of *one* input that multiplies its input by 10. Then we can feed the lambda expression to MAPCAR.

```
> (mapcar #'(lambda (n) (* n 10)) '(1 2 3 4 5))
(10 20 30 40 50)
```

Here is another example of the use of MAPCAR along with a lambda expression. We will turn each element of a list of names into a list (HI THERE *name*).

```
> (mapcar #'(lambda (x) (list 'hi 'there x))
          '(joe fred wanda))
((HI THERE JOE) (HI THERE FRED) (HI THERE WANDA))
```

If you type in a quoted lambda expression at top level, the result you get back depends on the particular Lisp implementation you're using. It might look like any of the following:

```
> (lambda (n) (* n 10))          Don't forget to quote it!
Error: Undefined function LAMBDA.

> #'(lambda (n) (* n 10))
(LAMBDA (N) (* N 10))

> #'(lambda (n) (* n 10))
#<Interpreted-function 3515162>

> #'(lambda (n) (* n 10))
#<Lexical-closure {7142156}>
```

Throughout this book we will refer to the objects you get back from a #'(LAMBDA...) expression as **lexical closures**. They will be discussed in more detail in the Advanced Topics section.

EXERCISES

7.5. Write a lambda expression to subtract seven from a number.

7.6. Write a lambda expression that returns T if its input is T or NIL, but NIL for any other input.

7.7. Write a function that takes a list such as (UP DOWN UP UP) and "flips" each element, returning (DOWN UP DOWN DOWN). Your

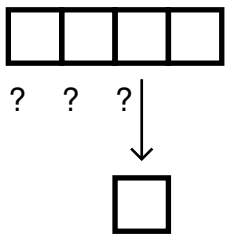
function should include a lambda expression that knows how to flip an individual element, plus an applicative operator to do this to every element of the list.

7.6 THE FIND-IF OPERATOR

FIND-IF is another applicative operator. If you give FIND-IF a predicate and a list as input, it will find the first element of the list for which the predicate returns *true* (any non-NIL value). FIND-IF returns that element.

```
> (find-if #'oddp '(2 4 6 7 8 9))
7

> (find-if #'(lambda (x) (> x 3))
           '(2 4 6 7 8 9))
4
```

Here is a graphical description of what FIND-IF does:



If no elements satisfy the predicate, FIND-IF returns NIL.

```
(find-if #'oddp '(2 4 6 8))  ⇒  nil
```

7.7 WRITING ASSOC WITH FIND-IF

ASSOC searches for a table entry with a specified key. We can write a simple version of ASSOC that uses FIND-IF to search the table.

```
(defun my-assoc (key table)
   (find-if #'(lambda (entry)
                (equal key (first entry)))
            table))
```

```
(my-assoc 'two words)  ⇒  (TWO DEUX)
```

The lambda expression (actually a lexical closure) that MY-ASSOC passes to FIND-IF takes a table entry such as (ONE UN) as input. It returns T if the first element of the entry matches the key that is the first input to MY-ASSOC. FIND-IF calls the closure on each entry in the table, until it finds one that makes the closure return T.

Notice that the expression (EQUAL KEY (FIRST ENTRY)) that appears in the body of the lambda expression refers to two variables. ENTRY is local to the lambda expression, but KEY is not. KEY is local to MY-ASSOC. This illustrates an important point about lambda expressions: Inside the body of a lambda expression we can not only reference its local variables, we can also reference any local variables of the function containing the lambda expression.

EXERCISES

7.8. Write a function that takes two inputs, X and K, and returns the first number in the list X that is roughly equal to K. Let's say that ''roughly equal'' means no less than K − 10 and no more than K + 10.

7.9. Write a function FIND-NESTED that returns the first element of a list that is itself a non-NIL list.

MINI KEYBOARD EXERCISE

7.10. In this exercise we will write a program to transpose a song from one key to another. In order to manipulate notes more efficiently, we will translate them into numbers. Here is the correspondence between notes and numbers for a one-octave scale:

```
C       =  1          F-SHARP =  7
C-SHARP =  2          G       =  8
D       =  3          G-SHARP =  9
D-SHARP =  4          A       = 10
E       =  5          A-SHARP = 11
F       =  6          B       = 12
```

a. Write a table to represent this information. Store it in a global variable called NOTE-TABLE.

b. Write a function called NUMBERS that takes a list of notes as input and returns the corresponding list of numbers. (NUMBERS '(E D C D E E E)) should return (5 3 1 3 5 5 5). This list represents the first seven notes of ''Mary Had a Little Lamb.''

c. Write a function called NOTES that takes a list of numbers as input and returns the corresponding list of notes. (NOTES '(5 3 1 3 5 5 5)) should return (E D C D E E E). *Hint:* Since NOTE-TABLE is keyed by note, ASSOC can't look up numbers in it; neither can RASSOC, since the elements are lists, not dotted pairs. Write your own table-searching function to search NOTE-TABLE by number instead of by note.

d. Notice that NOTES and NUMBERS are mutual inverses:

For X a list of notes:

```
X  =  (NOTES (NUMBERS X))
```

For X a list of numbers:

```
X  =  (NUMBERS (NOTES X))
```

What can be said about (NOTES (NOTES X)) and (NUMBERS (NUMBERS X))?

e. To transpose a piece of music up by *n* half steps, we begin by adding the value *n* to each note in the piece. Write a function called RAISE that takes a number *n* and a list of numbers as input and raises each number in the list by the value *n*. (RAISE 5 '(5 3 1 3 5 5 5)) should return (10 8 6 8 10 10 10), which is ''Mary Had a Little Lamb'' transposed five half steps from the key of C to the key of F.

f. Sometimes when we raise the value of a note, we may raise it right into the next octave. For instance, if we raise the triad C-E-G represented by the list (1 5 8) into the key of F by adding five to each note, we get (6 10 13), or F-A-C. Here the C note, represented by the number 13, is an octave above the regular C, represented by 1. Write a function called NORMALIZE that takes a list of numbers as input and ''normalizes'' them to make them be between 1 and 12. A number greater than 12 should have 12 subtracted from it; a number less than 1 should have 12 added to it. (NORMALIZE '(6 10 13)) should return (6 10 1).

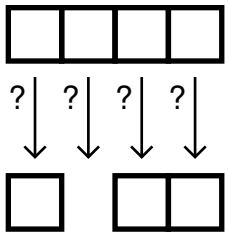
g. Write a function TRANSPOSE that takes a number *n* and a song as input, and returns the song transposed by *n* half steps. (TRANSPOSE 5 '(E D C D E E E)) should return (A G F G A A A). Your solution should assume the availability of the NUMBERS, NOTES, RAISE, and NORMALIZE functions. Try transposing ''Mary Had a Little Lamb'' up by 11 half steps. What happens if you transpose it by 12 half steps? How about −1 half steps?

7.8 REMOVE-IF AND REMOVE-IF-NOT

REMOVE-IF is another applicative operator that takes a predicate as input. REMOVE-IF removes all the items from a list that satisfy the predicate, and returns a list of what's left.

```
> (remove-if #'numberp '(2 for 1 sale))
(FOR SALE)

> (remove-if #'oddp '(1 2 3 4 5 6 7))
(2 4 6)
```

Here is a graphical description of REMOVE-IF:



Suppose we want to find all the positive elements in a list of numbers. The PLUSP predicate tests if a number is greater than zero. To invert the sense of this predicate we wrap a NOT around it using a lambda expression, as shown in the following. After removing all the elements that satisfy (NOT (PLUSP *x*)), what we have left are the positive elements.

```
> (remove-if #'(lambda (x) (not (plusp x)))
             '(2 0 -4 6 -8 10))
(2 6 10)
```

The REMOVE-IF-NOT operator is used more frequently than REMOVE-IF. It works just like REMOVE-IF except it automatically inverts the sense of the predicate. This means the only items that will be removed are those for which the predicate returns NIL. So REMOVE-IF-NOT returns a list of all the items that *satisfy* the predicate. Thus, if we choose PLUSP as the predicate, REMOVE-IF-NOT will find all the positive numbers in a list.

```
> (remove-if-not #'plusp '(2 0 -4 6 -8 10))
(2 6 10)
  > (remove-if-not #'oddp '(2 0 -4 6 -8 10))
NIL
```

Here are some additional examples of REMOVE-IF-NOT:

```
> (remove-if-not #'(lambda (x) (> x 3))
                 '(2 4 6 8 4 2 1))
(4 6 8 4)

> (remove-if-not #'numberp
      '(3 apples 4 pears and 2 little plums))
(3 4 2)

> (remove-if-not #'symbolp
       '(3 apples 4 pears and 2 little plums))
(APPLES PEARS AND LITTLE PLUMS)
```

Here is a function, COUNT-ZEROS, that counts how many zeros appear in a list of numbers. It does this by taking the subset of the list elements that are zero, and then taking the length of the result.

```
(remove-if-not #'zerop '(34 0 0 95 0))  ⇒  (0 0 0)

(defun count-zeros (x)
    (length (remove-if-not #'zerop x)))

(count-zeros '(34 0 0 95 0))  ⇒  3

(count-zeros '(1 0 63 0 38))  ⇒  2

(count-zeros '(0 0 0 0 0))  ⇒  5

(count-zeros '(1 2 3 4 5))  ⇒  0
```

EXERCISES

7.11. Write a function to pick out those numbers in a list that are greater than one and less than five.

7.12. Write a function that counts how many times the word ‘‘the’’ appears in a sentence.

7.13. Write a function that picks from a list of lists those of exactly length two.

7.14. Here is a version of SET-DIFFERENCE written with REMOVE-IF:

```
(defun my-setdiff (x y)
  (remove-if #'(lambda (e) (member e y))
             x))
```

Show how the INTERSECTION and UNION functions can be written using REMOVE-IF or REMOVE-IF-NOT.

MINI KEYBOARD EXERCISE

7.15. In this keyboard exercise we will manipulate playing cards with applicative operators. A card will be represented by a list of form (*rank suit*), for example, (ACE SPADES) or (2 CLUBS). A hand will be represented by a list of cards.

a. Write the functions RANK and SUIT that return the rank and suit of a card, respectively. (RANK '(2 CLUBS)) should return 2, and (SUIT '(2 CLUBS)) should return CLUBS.

b. Set the global variable MY-HAND to the following hand of cards:

```
((3 hearts)
 (5 clubs)
 (2 diamonds)
 (4 diamonds)
 (ace spades))
```

Now write a function COUNT-SUIT that takes two inputs, a suit and a hand of cards, and returns the number of cards belonging to that suit. (COUNT-SUIT 'DIAMONDS MY-HAND) should return 2.

c. Set the global variable COLORS to the following table:

```
((clubs black)
 (diamonds red)
 (hearts red)
 (spades black))
```

Now write a function COLOR-OF that uses the table COLORS to retrieve the color of a card. (COLOR-OF '(2 CLUBS)) should return BLACK. (COLOR-OF '(6 HEARTS)) should return RED.

d. Write a function FIRST-RED that returns the first card of a hand that is of a red suit, or NIL if none are.

e. Write a function BLACK-CARDS that returns a list of all the black cards in a hand.

f. Write a function WHAT-RANKS that takes two inputs, a suit and a hand, and returns the ranks of all cards belonging to that suit. (WHAT-RANKS 'DIAMONDS MY-HAND) should return the list (2 4). (WHAT-RANKS 'SPADES MY-HAND) should return the

list (ACE). *Hint:* First extract all the cards of the specified suit, then use another operator to get the ranks of those cards.

g. Set the global variable ALL-RANKS to the list

```
(2 3 4 5 6 7 8 9 10 jack queen king ace)
```

Then write a predicate HIGHER-RANK-P that takes two cards as input and returns true if the first card has a higher rank than the second. *Hint:* look at the BEFOREP predicate on page 171 of Chapter 6.

h. Write a function HIGH-CARD that returns the highest ranked card in a hand. *Hint:* One way to solve this is to use FIND-IF to search a list of ranks (ordered from high to low) to find the highest rank that appears in the hand. Then use ASSOC on the hand to pick the card with that rank. Another solution would be to use REDUCE (defined in the next section) to repeatedly pick the highest card of each pair.

7.9 THE REDUCE OPERATOR

REDUCE is an applicative operator that reduces the elements of a list into a single result. REDUCE takes a function and a list as input, but unlike the other operators we've seen, REDUCE must be given a function that accepts *two* inputs. Example: To add up a list of numbers with REDUCE, we use + as the reducing function.

```
(reduce #'+ '(1 2 3))  ⇒  6

(reduce #'+ '(10 9 8 7 6))  ⇒  40

(reduce #'+ '(5))  ⇒  5

(reduce #'+ nil)  ⇒  0
```

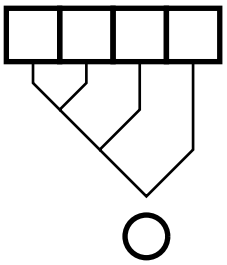
Similarly, to multiply a bunch of numbers together, we use * as the reducing function:

```
(reduce #'* '(2 4 5))  ⇒  40

(reduce #'* '(3 4 0 7))  ⇒  0

(reduce #'* '(8))  ⇒  8
```

We can also apply reduction to lists of lists. To turn a table into a one-level list, we use APPEND as the reducing function:

```
> (reduce #'append
          '((one un) (two deux) (three trois)))
(ONE UN TWO DEUX THREE TROIS)
```

Here is A graphical description of REDUCE:



EXERCISES

7.16. Suppose we had a list of sets ((A B C) (C D A) (F B D) (G)) that we wanted to collapse into one big set. If we use APPEND for our reducing function, the result won't be a true set, because some elements will appear more than once. What reducing function should be used instead?

7.17. Write a function that, given a list of lists, returns the total length of all the lists. This problem can be solved two different ways.

7.18. (REDUCE #'+ NIL) returns 0, but (REDUCE #'* NIL) returns 1. Why do you think this is?

7.10 EVERY

EVERY takes a predicate and a list as input. It returns T if there is no element that causes the predicate to return false. Examples:

```
> (every #'numberp '(1 2 3 4 5))
T

> (every #'numberp '(1 2 A B C 5))
NIL
```

```
> (every #'(lambda (x) (> x 0)) '(1 2 3 4 5))
T

> (every #'(lambda (x) (> x 0)) '(1 2 3 -4 5))
NIL
```

If EVERY is called with NIL as its second argument, it simply returns T, since the empty list has no elements that could fail to satisfy the predicate.

```
> (every #'oddp nil)
T

> (every #'evenp nil)
T
```

EVERY can also operate on multiple lists, given a predicate that accepts multiple inputs.

```
> (every #'> '(10 20 30 40) '(1 5 11 23))
T
```

Since 10 is greater than 1, 20 greater than 5, 30 greater than 11, and 40 greater than 23, EVERY returns T.

EXERCISES

7.19. Write a function ALL-ODD that returns T if every element of a list of numbers is odd.

7.20. Write a function NONE-ODD that returns T if every element of a list of numbers is not odd.

7.21. Write a function NOT-ALL-ODD that returns T if not every element of a list of numbers is odd.

7.22. Write a function NOT-NONE-ODD that returns T if it is not the case that a list of numbers contains no odd elements.

7.23. Are all four of the above functions distinct from one another, or are some of them the same? Can you think of better names for the last two?

SUMMARY

Applicative operators are functions that apply other functions to data structures. There are many possible applicative operators, only a few of which are built in to Lisp. Advanced Lisp programmers make up their own operators whenever they need new ones.

MAPCAR applies a function to every element of a list and returns a list of the results. FIND-IF searches a list and returns the first element that satisfies a predicate. REMOVE-IF removes all the elements of a list that satisfy a predicate, so the list it returns contains only those elements that fail to satisfy it. REMOVE-IF-NOT is used more frequently than REMOVE-IF. It returns all the elements that *do* satisfy the predicate, having removed those that don't satisfy it. EVERY returns T only if every element of a list satisfies a predicate. REDUCE uses a reducing function to reduce a list to a single value.

REVIEW EXERCISES

7.24. What is an applicative operator?

7.25. Why are lambda expressions useful? Is it possible to do without them?

7.26. Show how to write FIND-IF given REMOVE-IF-NOT.

7.27. Show how to write EVERY given REMOVE-IF.

7.28. Devise a graphical description for the EVERY operator.

FUNCTIONS COVERED IN THIS CHAPTER

Applicative operators: MAPCAR, FIND-IF, REMOVE-IF, REMOVE-IF-NOT, REDUCE, EVERY.

Lisp Toolkit: TRACE and DTRACE

The TRACE macro is used to watch particular functions as they are called and as they return. With each call you will see the arguments to the function; when the function returns you will see the return values. Each Lisp implementation has its own style for displaying trace information. The example below is typical:

```
(defun half (n) (* n 0.5))

(defun average (x y)
  (+ (half x) (half y)))
```

```
> (trace half average)
(HALF AVERAGE)

> (average 3 7)
 0: (AVERAGE 3 7)
   1: (HALF 3)
   1: returned 1.5
   1: (HALF 7)
   1: returned 3.5
 0: returned 5.0
5.0
```

If you call TRACE with no arguments, it returns the list of currently traced functions.

```
> (trace)
(HALF AVERAGE)
```

The UNTRACE macro turns off tracing for one or more functions. Since UNTRACE is a macro function like TRACE, its arguments should not be quoted.

```
> (untrace HALF)
(HALF)
```

Calling UNTRACE with no arguments untraces all currently traced functions.

```
> (untrace)
(AVERAGE)
```

In the remainder of this book we will use a more detailed tracing format that shows each variable in the argument list along with the value to which it is bound. For example:

```
> (average 3 7)
----Enter AVERAGE
|     X = 3
|     Y = 7
|   ----Enter HALF
|   |     N = 3
|    \--HALF returned 1.5
|   ----Enter HALF
|   |     N = 7
|    \--HALF returned 3.5
 \--AVERAGE returned 5.0
5.0
```

If your Lisp's TRACE isn't this detailed, don't panic, you can use mine. It's called DTRACE, and the full program listing is given in an appendix at the end of the book. This style of trace is especially helpful when tracing functions with several inputs, and even more so when the inputs are long, possibly nested, lists.

```
(defun add-to-end (x y)
  (append x (list y)))

(defun repeat-first (phrase)
  (add-to-end phrase (first phrase)))

> (dtrace add-to-end repeat-first)
(ADD-TO-END REPEAT-FIRST)

> (repeat-first '(for whom the bell tolls))
----Enter REPEAT-FIRST
|     PHRASE = (FOR WHOM THE BELL TOLLS)
|   ----Enter ADD-TO-END
|   |     X = (FOR WHOM THE BELL TOLLS)
|   |     Y = FOR
|    \--ADD-TO-END returned
|           (FOR WHOM THE BELL TOLLS FOR)
 \--REPEAT-FIRST returned
        (FOR WHOM THE BELL TOLLS FOR)
(FOR WHOM THE BELL TOLLS FOR)
```

DUNTRACE undoes the effect of DTRACE. Don't try to trace a function with both TRACE and DTRACE at the same time: You may get very strange results.

We can use DTRACE to observe the behavior of applicative operators like FIND-IF. We will trace the ODDP function and then use ODDP as an input to FIND-IF.

```
(defun find-first-odd (x)
  (find-if #'oddp x))

> (dtrace find-first-odd oddp)
(FIND-FIRST-ODD ODDP)
```

```
> (find-first-odd '(2 4 6 7 8))
----Enter FIND-FIRST-ODD
|     X = (2 4 6 7 8)
|   ----Enter ODDP
|   |     NUMBER = 2
|    \--ODDP returned NIL
|   ----Enter ODDP
|   |     NUMBER = 4
|    \--ODDP returned NIL
|   ----Enter ODDP
|   |     NUMBER = 6
|    \--ODDP returned NIL
|   ----Enter ODDP
|   |     NUMBER = 7
|    \--ODDP returned T
 \--FIND-FIRST-ODD returned 7
7
```

This brings up one last point about the use of TRACE and DTRACE. Although they may be used to trace built-in functions such as ODDP, this sometimes turns out to be dangerous. Avoid tracing the most fundamental built-in functions such as EVAL, CONS, and +. Otherwise your Lisp might end up in an infinite loop, and you will have to abandon it and start over.

Keyboard Exercise

In this keyboard exercise we will develop a system for representing knowledge about ''blocks world'' scenes such as Figure 7-1. Assertions about the objects in a scene are represented as triples of form (*block attribute value*). Here are some assertions about block B2's attributes:

```
(b2 shape brick)
(b2 color red)
(b2 size small)
(b2 supports b1)
(b2 left-of b3)
```

A collection (in other words, a list) of assertions is called a **database**.

Figure 7-1 A typical blocks world scene.

Given a database describing the blocks in the figure, we can write functions to answer questions such as, ‘‘What color is block B2?’’ or ‘‘What blocks support block B1?’’ To answer these questions, we will use a function called a **pattern matcher** to search the database for us. For example, to find out the color of block B2, we use the pattern (B2 COLOR ?).

```
> (fetch '(b2 color ?))
((B2 COLOR RED))
```

To find which blocks support B1, we use the pattern (? SUPPORTS B1):

```
> (fetch '(? supports b1))
((B2 SUPPORTS B1) (B3 SUPPORTS B1))
```

FETCH returns those assertions from the database that match a given pattern. It should be apparent from the preceding examples that a pattern is a triple, like an assertion, with some of its elements replaced by question marks. Figure 7-2 shows some patterns and their English interpretations.

A question mark in a pattern means any value can match in that position. Thus, the pattern (B2 COLOR ?) can match assertions like (B2 COLOR RED), (B2 COLOR GREEN), (B2 COLOR BLUE), and so on. It cannot match the assertion (B1 COLOR RED), because the first element of the pattern is the symbol B2, whereas the first element of the assertion is B1.

Pattern	English Interpretation
`(b1 color ?)`	*What color is b1?*
`(? color red)`	*Which blocks are red?*
`(b1 color red)`	*Is b1 known to be red?*
`(b1 ? b2)`	*What relation is b1 to b2?*
`(b1 ? ?)`	*What is known about b1?*
`(? supports ?)`	*What support relationships exist?*
`(? ? b1)`	*What blocks are related to b1?*
`(? ? ?)`	*What's in the database?*

Figure 7-2 Some patterns and their interpretations.

EXERCISE

7.29. If the blocks database is already stored on the computer for you, load the file containing it. If not, you will have to type it in as it appears in Figure 7-3. Save the database in the global variable DATABASE.

a. Write a function MATCH-ELEMENT that takes two symbols as inputs. If the two are equal, or if the second is a question mark, MATCH-ELEMENT should return T. Otherwise it should return NIL. Thus (MATCH-ELEMENT 'RED 'RED) and (MATCH-ELEMENT 'RED '?) should return T, but (MATCH-ELEMENT 'RED 'BLUE) should return NIL. Make sure your function works correctly before proceeding further.

b. Write a function MATCH-TRIPLE that takes an assertion and a pattern as input, and returns T if the assertion matches the pattern. Both inputs will be three-element lists. (MATCH-TRIPLE '(B2 COLOR RED) '(B2 COLOR ?)) should return T. (MATCH-TRIPLE '(B2 COLOR RED) '(B1 COLOR GREEN)) should return NIL. *Hint:* Use MATCH-ELEMENT as a building block.

c. Write the function FETCH that takes a pattern as input and returns all assertions in the database that match the pattern. Remember that