Keyboard Exercise

In this keyboard exercise we will write some routines for moving Robbie the robot around in a house. The map of the house appears in Figure 6-2. Robbie can move in any of four directions: north, south, east, or west.

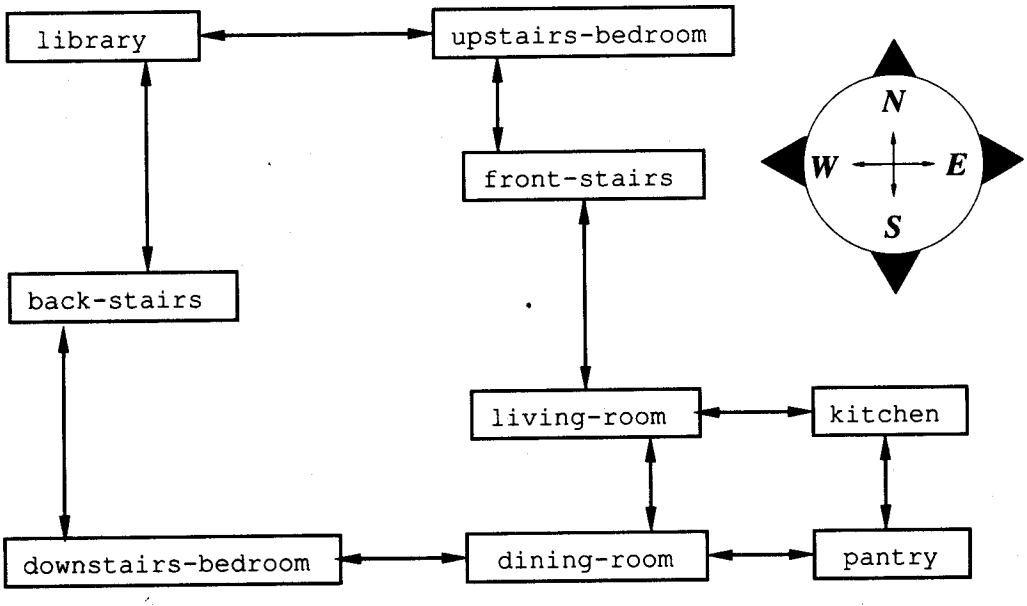

Figure 6-2 Map of the House.

The layout of the house is described in a table called ROOMS, with one element for each room:

```
((living-room ...)
 (upstairs-bedroom ...)
 (dining-room ...)
 (kitchen ...)
```

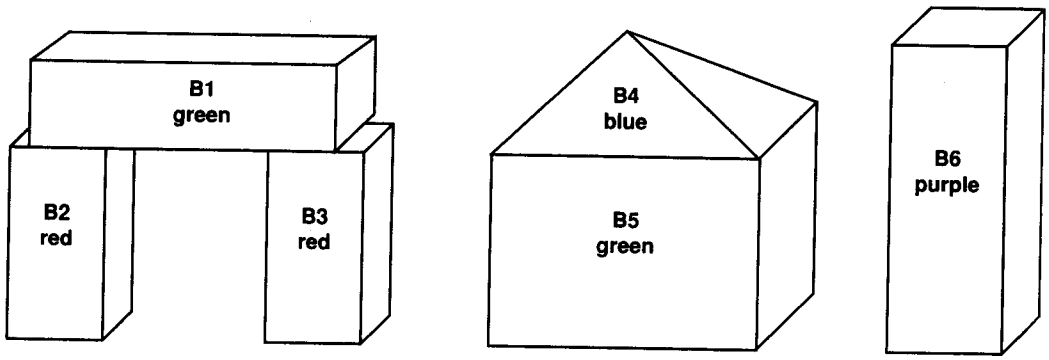

Figure 7-1 A typical blocks world scene.

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

Figure 8-9 Recursively nested cats, from *The Cat in the Hat Comes Back*, by Dr. Suess. Copyright (c) 1958 by Dr. Suess. Reprinted by permission of Random House, Inc.

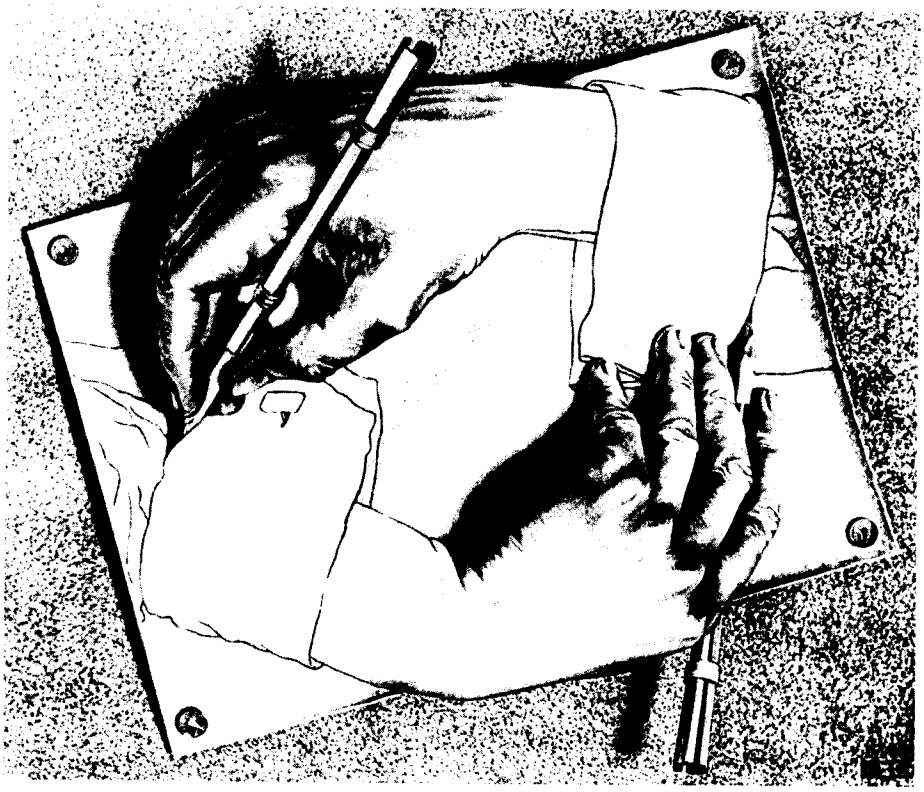

Figure 8-10 "Drawing Hands" by M. C. Escher. Copyright (c) 1989 M. C. Escher heirs/Cordon Art–Baarn–Holland.

SUMMARY

Recursion is a very powerful control structure, and one of the most important ideas in computer science. A function is said to be "recursive" if it calls itself. To write a recursive function, we must solve three problems posed by the Dragon's three rules of recursion:

1. Know when to stop.
2. Decide how to take one step.
3. Break the journey down into that step plus a smaller journey.

Keyboard Exercise

In this exercise we will extract different sorts of information from a genealogical database. The database gives information for five generations of a family, as shown in Figure 8-11. Such diagrams are usually called family trees, but this family's genealogical history is not a simple tree structure. Marie has married her first cousin Nigel. Wanda has had one child with Vincent and another with Ivan. Zelda and Robert, the parents of Yvette, have two great grandparents in common. (This might explain why Yvette turned out so weird.) And only Tamara knows who Frederick's father is; she's not telling.

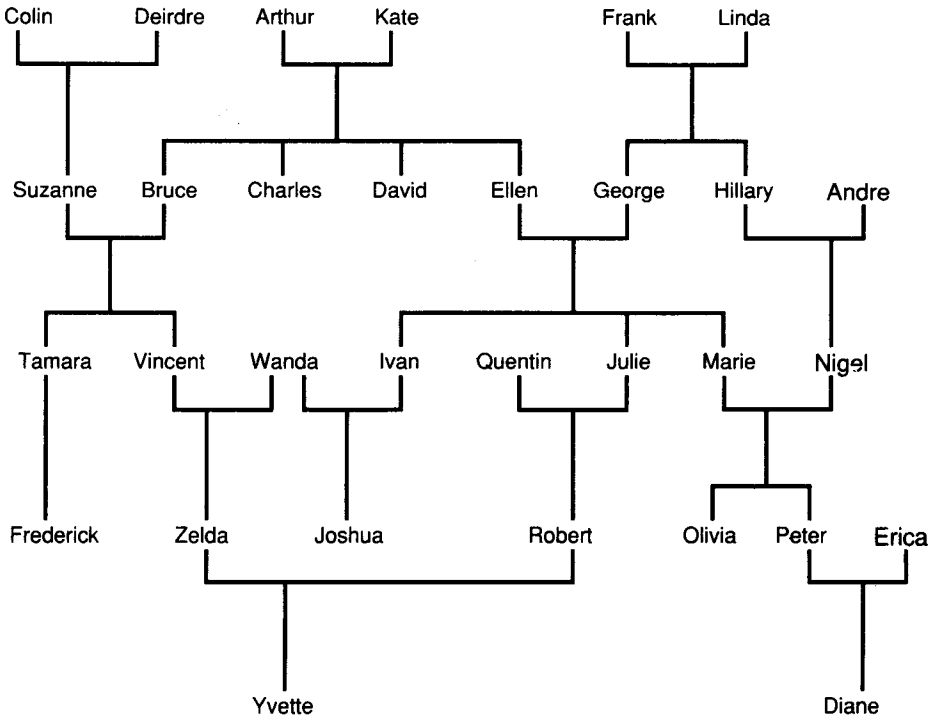

Figure 8-11 Genealogy information for five generations of a family.

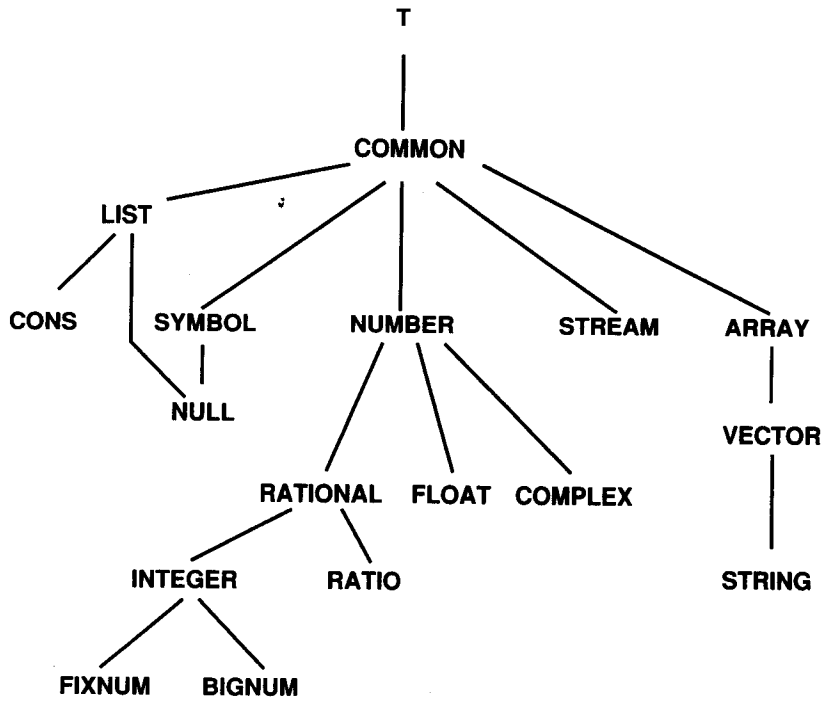

Figure 12-1 A portion of the Common Lisp type hierarchy.

12.3 DEFINING STRUCTURES

Structures are programmer-defined Lisp objects with an arbitrary number of named components. Structure types automatically become part of the Lisp type hierarchy. The DEFSTRUCT macro defines new structures and specifies the names and default values of their components. For example, we can define a structure called STARSHIP like this:

```
(defstruct starship
  (name nil)
  (speed 0)
  (condition 'green)
  (shields 'down))
```

This DEFSTRUCT form defines a new type of object called a STARSHIP whose components are called NAME, SPEED, CONDITION, and SHIELDS. STARSHIP becomes part of the system type hierarchy and can be referenced by such functions as TYPEP and TYPE-OF.

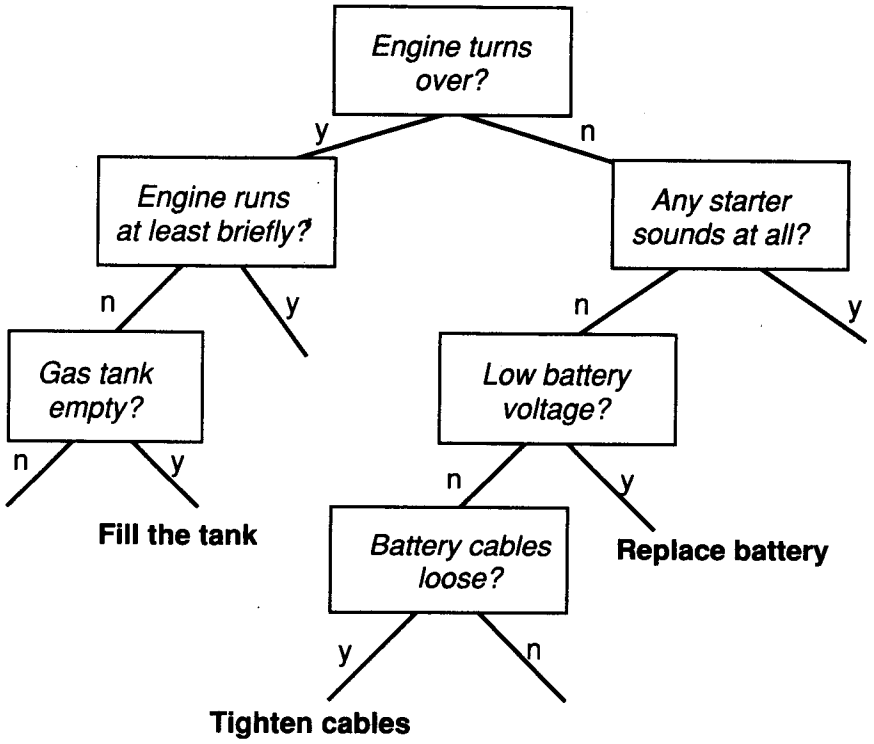

Figure 12-2 A portion of a discrimination net for solving automotive diagnosis problems.

EXERCISE

12.4. In this exercise you will create a discrimination net for automotive diagnosis that mimics the behavior of the system shown in the preceding pages.

a. Write a DEFSTRUCT for a structure called NODE, with four components called NAME, QUESTION, YES-CASE, and NO-CASE.

b. Define a global variable *NODE-LIST* that will hold all the nodes in the discrimination net. Write a function INIT that initializes the network by setting *NODE-LIST* to NIL.

c. Write ADD-NODE. It should return the name of the node it added.

d. Write FIND-NODE, which takes a node name as input and returns the node if it appears in *NODE-LIST*, or NIL if it doesn't.

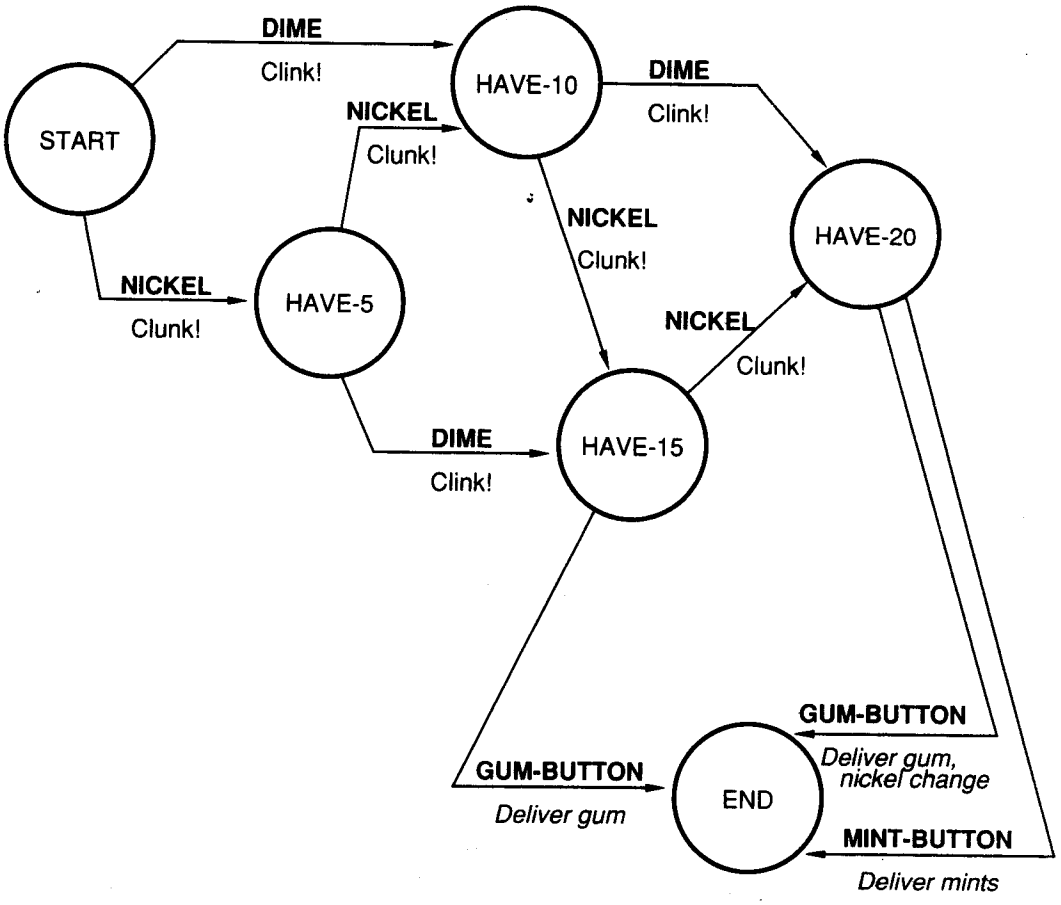

Figure 14-1 Finite state diagram for a vending machine.

moves to a state named HAVE-5. If it's in state HAVE-5 and it gets the symbol DIME as input, it goes "Clink!" and moves to state HAVE-15. In state HAVE-15, if it gets the input GUM-BUTTON, it delivers a packet of gum and goes to state END.

The machine has a total of six states: START, HAVE-5, HAVE-10, HAVE-15, HAVE-20, and END. (It's called a finite state machine precisely because the number of states is finite.) Each state is represented by a node in Figure 14-1, and each possible transition from one state to the next is represented by an arc (an arrow). The arc is labeled with the input needed to make the transition and the action the machine should take when it follows that transition. For example, the arc from HAVE-10 to HAVE-15 is labeled NICKEL / "Clunk!".