6

Functions as Representation

Generally, data structures are used to *represent.* An array could represent a geometric transformation; a tree could represent a hierarchy of command; a graph could represent a rail network. In Lisp we can sometimes use closures as a representation. Within a closure, variable bindings can store information, and can also play the role that pointers play in constructing complex data structures. By making a group of closures which share bindings, or can refer to one another, we can create hybrid objects which combine the advantages of data structures and programs.

Beneath the surface, shared bindings *are* pointers. Closures just bring us the convenience of dealing with them at a higher level of abstraction. By using closures to represent something we would otherwise represent with static data structures, we can often expect substantial improvements in elegance and efficiency.

6.1 Networks

Closures have three useful properties: they are active, they have local state, and we can make multiple instances of them. Where could we use multiple copies of active objects with local state? In applications involving networks, among others. In many cases we can represent nodes in a network as closures. As well as having its own local state, a closure can refer to another closure. Thus a closure representing a node in a network can know of several other nodes (closures) to which it must send its output. This means that we may be able to translate some networks straight into code.

```
> (run-node 'people)
Is the person a man?
>> yes
Is he living?
>> no
Was he American?
>> yes
Is he on a coin?
>> yes
Is the coin a penny?
>> yes
LINCOLN
```

Figure 6.1: Session of twenty questions.

In this section and the next we will look at two ways to traverse a network. First we will follow the traditional approach, with nodes defined as structures, and separate code to traverse the network. Then in the next section we'll show how to build the same program from a single abstraction.

As an example, we will use about the simplest application possible: one of those programs that play twenty questions. Our network will be a binary tree. Each non-leaf node will contain a yes/no question, and depending on the answer to the question, the traversal will continue down the left or right subtree. Leaf nodes will contain return values. When the traversal reaches a leaf node, its value will be returned as the value of the traversal. A session with this program might look as in Figure 6.1.

The traditional way to begin would be to define some sort of data structure to represent nodes. A node is going to have to know several things: whether it is a leaf; if so, which value to return, and if not, which question to ask; and where to go depending on the answer. A sufficient data structure is defined in Figure 6.2. It is designed for minimal size. The `contents` field will contain either a question or a return value. If the node is not a leaf, the `yes` and `no` fields will tell where to go depending on the answer to the question; if the node is a leaf, we will know it because these fields are empty. The global `*nodes*` will be a hash-table in which nodes are indexed by name. Finally, `defnode` makes a new node (of either type) and stores it in `*nodes*`. Using these materials we could define the first node of our tree:

```
(defnode 'people "Is the person a man?"
         'male 'female)
```

```
(defstruct node  contents yes no)

(defvar *nodes* (make-hash-table))

(defun defnode (name conts &optional yes no)
  (setf (gethash name *nodes*)
        (make-node :contents conts
                   :yes      yes
                   :no       no)))
```

Figure 6.2: Representation and definition of nodes.

```
(defnode 'people "Is the person a man?" 'male 'female)

(defnode 'male "Is he living?" 'liveman 'deadman)

(defnode 'deadman "Was he American?" 'us 'them)

(defnode 'us "Is he on a coin?" 'coin 'cidence)

(defnode 'coin "Is the coin a penny?" 'penny 'coins)

(defnode 'penny 'lincoln)
```

Figure 6.3: Sample network.

Figure 6.3 shows as much of the network as we need to produce the transcript in Figure 6.1.

Now all we need to do is write a function to traverse this network, printing out the questions and following the indicated path. This function, `run-node`, is shown in Figure 6.4. Given a name, we look up the corresponding node. If it is not a leaf, the `contents` are asked as a question, and depending on the answer, we continue traversing at one of two possible destinations. If the node is a leaf, `run-node` just returns its `contents`. With the network defined in Figure 6.3, this function produces the output shown in Figure 6.1.

```
(defun run-node (name)
  (let ((n (gethash name *nodes*)))
    (cond ((node-yes n)
           (format t "~A~%>> " (node-contents n))
           (case (read)
             (yes (run-node (node-yes n)))
             (t   (run-node (node-no n)))))
          (t (node-contents n)))))
```

Figure 6.4: Function for traversing networks.

```
(defvar *nodes* (make-hash-table))

(defun defnode (name conts &optional yes no)
  (setf (gethash name *nodes*)
        (if yes
            #'(lambda ()
                (format t "~A~%>> " conts)
                (case (read)
                  (yes (funcall (gethash yes *nodes*)))
                  (t   (funcall (gethash no  *nodes*)))))
            #'(lambda () conts))))
```

Figure 6.5: A network compiled into closures.

6.2 Compiling Networks

In the preceding section we wrote a network program as it might have been written in any language. Indeed, the program is so simple that it seems odd to think that we could write it any other way. But we can—in fact, we can write it much more simply.

The code in Figure 6.5 illustrates this point. It's all we really need to run our network. Instead of having nodes as data structures and a separate function to traverse them, we represent the nodes as closures. The data formerly contained in the structures gets stored in variable bindings within the closures. Now there is no need for `run-node`; it is implicit in the nodes themselves. To start the traversal,

```
(defvar *nodes* nil)

(defun defnode (&rest args)
  (push args *nodes*)
  args)

(defun compile-net (root)
  (let ((node (assoc root *nodes*)))
    (if (null node)
        nil
        (let ((conts (second node))
              (yes (third node))
              (no (fourth node)))
          (if yes
              (let ((yes-fn (compile-net yes))
                    (no-fn  (compile-net no)))
                #'(lambda ()
                    (format t "~A~%>> " conts)
                    (funcall (if (eq (read) 'yes)
                                 yes-fn
                                 no-fn))))
              #'(lambda () conts))))))
```

Figure 6.6: Compilation with static references.

we just funcall the node at which we want to begin:

```
(funcall (gethash 'people *nodes*))
Is the person a man?
>>
```

From then on, the transcript will be just as it was with the previous implementation.

By representing the nodes as closures, we are able to transform our twenty-questions network entirely into code. As it is, the code will have to look up the node functions by name at runtime. However, if we know that the network is not going to be redefined on the fly, we can add a further enhancement: we can have node functions call their destinations directly, without having to go through a hash-table.

Figure 6.6 contains a new version of the program. Now `*nodes*` is a disposable list instead of a hash-table. All the nodes are defined with `defnode` as before, but no closures are generated at this point. After all the nodes have been defined, we call `compile-net` to compile a whole network at once. This function recursively works its way right down to the leaves of the tree, and on the way back up, returns at each step the node/function for each of the two subtrees.[1] So now each node will have a direct handle on its two destinations, instead of having only their names. When the original call to `compile-net` returns, it will yield a function representing the portion of the network we asked to have compiled.

```
> (setq n (compile-net 'people))
#<Compiled-Function BF3C06>
> (funcall n)
Is the person a man?
>>
```

Notice that `compile-net` compiles in both senses. It compiles in the general sense, by translating the abstract representation of the network into code. Moreover, if `compile-net` itself is compiled, it will return compiled functions. (See page 25.)

After compiling the network, we will no longer need the list made by `defnode`. It can be cut loose (e.g. by setting `*nodes*` to `nil`) and reclaimed by the garbage collector.

6.3 Looking Forward

Many programs involving networks can be implemented by compiling the nodes into closures. Closures are data objects, and they can be used to represent things just as structures can. Doing so requires some unconventional thinking, but the rewards are faster and more elegant programs.

Macros help substantially when we use closures as a representation. "To represent with closures" is another way of saying "to compile," and since macros do their work at compile-time, they are a natural vehicle for this technique. After macros have been introduced, Chapters 23 and 24 will present much larger programs based on the strategy used here.

[1]This version assumes that the network is a tree, which it must be in this application.