5
Computing with Register Machines

> My aim is to show that the heavenly machine is not a kind of divine, live being, but a kind of clockwork (and he who believes that a clock has soul attributes the maker's glory to the work), insofar as nearly all the manifold motions are caused by a most simple and material force, just as all motions of the clock are caused by a single weight.
>
> Johannes Kepler (letter to Herwart von Hohenburg, 1605)

We began this book by studying processes and by describing processes in terms of procedures written in Lisp. To explain the meanings of these procedures, we used a succession of models of evaluation: the substitution model of chapter 1, the environment model of chapter 3, and the metacircular evaluator of chapter 4. Our examination of the metacircular evaluator, in particular, dispelled much of the mystery of how Lisp-like languages are interpreted. But even the metacircular evaluator leaves important questions unanswered, because it fails to elucidate the mechanisms of control in a Lisp system. For instance, the evaluator does not explain how the evaluation of a subexpression manages to return a value to the expression that uses this value, nor does the evaluator explain how some recursive procedures generate iterative processes (that is, are evaluated using constant space) whereas other recursive procedures generate recursive processes. These questions remain unanswered because the metacircular evaluator is itself a Lisp program and hence inherits the control structure of the underlying Lisp system. In order to provide a more complete description of the control structure of the Lisp evaluator, we must work at a more primitive level than Lisp itself.

In this chapter we will describe processes in terms of the step-by-step operation of a traditional computer. Such a computer, or *register machine*, sequentially executes *instructions* that manipulate the contents of a fixed set of storage elements called *registers*. A typical register-machine instruction applies a primitive operation to the contents of some registers and assigns the result to another register. Our descriptions of processes executed by register machines will look very much like "machine-language" programs for traditional computers. However, in-

stead of focusing on the machine language of any particular computer, we will examine several Lisp procedures and design a specific register machine to execute each procedure. Thus, we will approach our task from the perspective of a hardware architect rather than that of a machine-language computer programmer. In designing register machines, we will develop mechanisms for implementing important programming constructs such as recursion. We will also present a language for describing designs for register machines. In section 5.2 we will implement a Lisp program that uses these descriptions to simulate the machines we design.

Most of the primitive operations of our register machines are very simple. For example, an operation might add the numbers fetched from two registers, producing a result to be stored into a third register. Such an operation can be performed by easily described hardware. In order to deal with list structure, however, we will also use the memory operations `car`, `cdr`, and `cons`, which require an elaborate storage-allocation mechanism. In section 5.3 we study their implementation in terms of more elementary operations.

In section 5.4, after we have accumulated experience formulating simple procedures as register machines, we will design a machine that carries out the algorithm described by the metacircular evaluator of section 4.1. This will fill in the gap in our understanding of how Scheme expressions are interpreted, by providing an explicit model for the mechanisms of control in the evaluator. In section 5.5 we will study a simple compiler that translates Scheme programs into sequences of instructions that can be executed directly with the registers and operations of the evaluator register machine.

5.1 Designing Register Machines

To design a register machine, we must design its *data paths* (registers and operations) and the *controller* that sequences these operations. To illustrate the design of a simple register machine, let us examine Euclid's Algorithm, which is used to compute the greatest common divisor (GCD) of two integers. As we saw in section 1.2.5, Euclid's Algorithm can be carried out by an iterative process, as specified by the following procedure:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

A machine to carry out this algorithm must keep track of two numbers, a and b, so let us assume that these numbers are stored in two registers with those names. The basic operations required are testing whether the contents of register `b` is zero and computing the remainder of the contents of register `a` divided by the contents of register `b`. The remainder operation is a complex process, but assume for the moment that we have a primitive device that computes remainders. On each cycle of the GCD algorithm, the contents of register `a` must be replaced by the contents of register `b`, and the contents of `b` must be replaced by the remainder of the old contents of `a` divided by the old contents of `b`. It would be convenient if these replacements could be done simultaneously, but in our model of register machines we will assume that only one register can be assigned a new value at each step. To accomplish the replacements, our machine will use a third "temporary" register, which we call `t`. (First the remainder will be placed in `t`, then the contents of `b` will be placed in `a`, and finally the remainder stored in `t` will be placed in `b`.)

We can illustrate the registers and operations required for this machine by using the data-path diagram shown in figure 5.1. In this diagram, the registers (`a`, `b`, and `t`) are represented by rectangles. Each way to assign a value to a register is indicated by an arrow with an `X` behind the head, pointing from the source of data to the register. We can think of the `X` as a button that, when pushed, allows the value at the source to "flow" into the designated register. The label next to each button is the name we will use to refer to the button. The names are arbitrary, and can be chosen to have mnemonic value (for example, `a<-b` denotes pushing the button that assigns the contents of register `b` to register `a`). The source of data for a register can be another register (as in the `a<-b` assignment), an operation result (as in the `t<-r` assignment), or a constant (a built-in value that cannot be changed, represented in a data-path diagram by a triangle containing the constant).

An operation that computes a value from constants and the contents of registers is represented in a data-path diagram by a trapezoid containing a name for the operation. For example, the box marked `rem` in figure 5.1 represents an operation that computes the remainder of the contents of the registers `a` and `b` to which it is attached. Arrows (without buttons) point from the input registers and constants to the box, and arrows connect the operation's output value to registers. A test is represented by a circle containing a name for the test. For example, our GCD machine has an operation that tests whether the contents of register `b` is zero. A test also has arrows from its input registers and constants, but it has no output arrows; its value is used by the controller rather than by the data paths.

Overall, the data-path diagram shows the registers and operations that are required for the machine and how they must be connected. If we view the arrows as wires and the `X` buttons as switches, the data-path diagram is very like the wiring diagram for a machine that could be constructed from electrical components.

In order for the data paths to actually compute GCDs, the buttons must be pushed in the correct sequence. We will describe this sequence in terms of a controller diagram, as illustrated in figure 5.2. The elements of the controller diagram indicate how the data-path components should be operated. The rectangular boxes in the controller diagram identify data-path buttons to be pushed, and the arrows describe the sequencing from one step to the next. The diamond in the diagram represents a decision. One of the two sequencing arrows will be followed, depending on the value of the data-path test identified in the diamond. We can interpret the controller in terms of a physical analogy: Think of the diagram as a maze in which a marble is rolling. When the marble rolls into a box, it pushes the data-path button that is named by the box. When the marble rolls into a decision node (such as the test for $b = 0$), it leaves the node on the path determined by the result of the indicated test. Taken together, the data paths and the controller completely describe a machine for computing GCDs. We start the controller (the rolling marble) at the place marked `start`, after placing numbers in registers `a` and `b`. When the controller reaches `done`, we will find the value of the GCD in register `a`.

Exercise 5.1

Design a register machine to compute factorials using the iterative algorithm specified by the following procedure. Draw data-path and controller diagrams for this machine.

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

5.1.1 A Language for Describing Register Machines

Data-path and controller diagrams are adequate for representing simple machines such as GCD, but they are unwieldy for describing large machines such as a Lisp interpreter. To make it possible to deal with com-

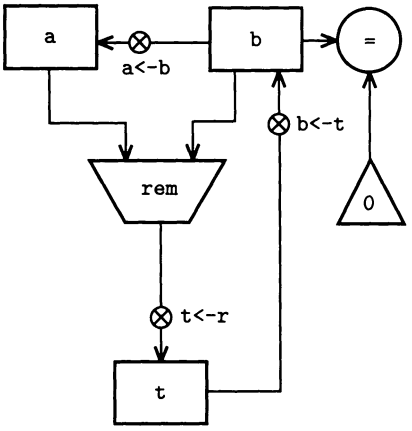


Figure 5.1 Data paths for a GCD machine.

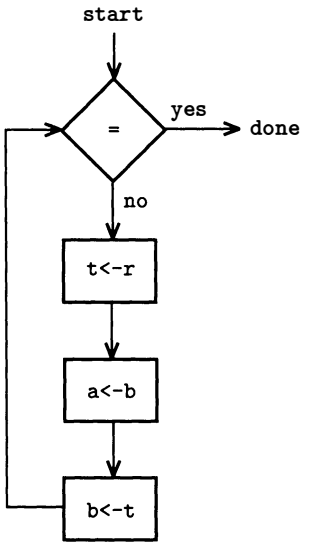


Figure 5.2 Controller for a GCD machine.

plex machines, we will create a language that presents, in textual form, all the information given by the data-path and controller diagrams. We will start with a notation that directly mirrors the diagrams.

We define the data paths of a machine by describing the registers and the operations. To describe a register, we give it a name and specify the buttons that control assignment to it. We give each of these buttons a name and specify the source of the data that enters the register under the button's control. (The source is a register, a constant, or an operation.) To describe an operation, we give it a name and specify its inputs (registers or constants).

We define the controller of a machine as a sequence of *instructions* together with *labels* that identify *entry points* in the sequence. An instruction is one of the following:

- The name of a data-path button to push to assign a value to a register. (This corresponds to a box in the controller diagram.)
- A `test` instruction, that performs a specified test.
- A conditional branch (`branch` instruction) to a location indicated by a controller label, based on the result of the previous test. (The test and branch together correspond to a diamond in the controller diagram.) If the test is false, the controller should continue with the next instruction in the sequence. Otherwise, the controller should continue with the instruction after the label.
- An unconditional branch (`goto` instruction) naming a controller label at which to continue execution.

The machine starts at the beginning of the controller instruction sequence and stops when execution reaches the end of the sequence. Except when a branch changes the flow of control, instructions are executed in the order in which they are listed.

Figure 5.3 shows the GCD machine described in this way. This example only hints at the generality of these descriptions, since the GCD machine is a very simple case: Each register has only one button, and each button and test is used only once in the controller.

Unfortunately, it is difficult to read such a description. In order to understand the controller instructions we must constantly refer back to the definitions of the button names and the operation names, and to understand what the buttons do we may have to refer to the definitions of the operation names. We will thus transform our notation to combine the information from the data-path and controller descriptions so that we see it all together.

To obtain this form of description, we will replace the arbitrary button and operation names by the definitions of their behavior. That is, instead

```
(data-paths
 (registers
  ((name a)
   (buttons ((name a<-b) (source (register b)))))
  ((name b)
   (buttons ((name b<-t) (source (register t)))))
  ((name t)
   (buttons ((name t<-r) (source (operation rem))))))

 (operations
  ((name rem)
   (inputs (register a) (register b)))
  ((name =)
   (inputs (register b) (constant 0)))))

(controller
 test-b                              ; label
   (test =)                          ; test
   (branch (label gcd-done))         ; conditional branch
   (t<-r)                            ; button push
   (a<-b)                            ; button push
   (b<-t)                            ; button push
   (goto (label test-b))             ; unconditional branch
 gcd-done)                           ; label
```

Figure 5.3 A specification of the GCD machine.

of saying (in the controller) "Push button `t<-r`" and separately saying (in the data paths) "Button `t<-r` assigns the value of the `rem` operation to register `t`" and "The `rem` operation's inputs are the contents of registers `a` and `b`," we will say (in the controller) "Push the button that assigns to register `t` the value of the `rem` operation on the contents of registers `a` and `b`." Similarly, instead of saying (in the controller) "Perform the = test" and separately saying (in the data paths) "The = test operates on the contents of register `b` and the constant 0," we will say "Perform the = test on the contents of register `b` and the constant 0." We will omit the data-path description, leaving only the controller sequence. Thus, the GCD machine is described as follows:

```
(controller
 test-b
   (test (op =) (reg b) (const 0))
   (branch (label gcd-done))
   (assign t (op rem) (reg a) (reg b))
   (assign a (reg b))
   (assign b (reg t))
   (goto (label test-b))
 gcd-done)
```

This form of description is easier to read than the kind illustrated in figure 5.3, but it also has disadvantages:

- It is more verbose for large machines, because complete descriptions of the data-path elements are repeated whenever the elements are mentioned in the controller instruction sequence. (This is not a problem in the GCD example, because each operation and button is used only once.) Moreover, repeating the data-path descriptions obscures the actual data-path structure of the machine; it is not obvious for a large machine how many registers, operations, and buttons there are and how they are interconnected.
- Because the controller instructions in a machine definition look like Lisp expressions, it is easy to forget that they are not arbitrary Lisp expressions. They can notate only legal machine operations. For example, operations can operate directly only on constants and the contents of registers, not on the results of other operations.

In spite of these disadvantages, we will use this register-machine language throughout this chapter, because we will be more concerned with understanding controllers than with understanding the elements and connections in data paths. We should keep in mind, however, that data-path design is crucial in designing real machines.

Exercise 5.2

Use the register-machine language to describe the iterative factorial machine of exercise 5.1.

Actions

Let us modify the GCD machine so that we can type in the numbers whose GCD we want and get the answer printed at our terminal. We will not discuss how to make a machine that can read and print, but will assume (as we do when we use `read` and `display` in Scheme) that they are available as primitive operations.[1]

`Read` is like the operations we have been using in that it produces a value that can be stored in a register. But `read` does not take inputs from any registers; its value depends on something that happens outside the parts of the machine we are designing. We will allow our machine's operations to have such behavior, and thus will draw and notate the use of `read` just as we do any other operation that computes a value.

[1]This assumption glosses over a great deal of complexity. Usually a large portion of the implementation of a Lisp system is dedicated to making reading and printing work.

`Print`, on the other hand, differs from the operations we have been using in a fundamental way: It does not produce an output value to be stored in a register. Though it has an effect, this effect is not on a part of the machine we are designing. We will refer to this kind of operation as an *action*. We will represent an action in a data-path diagram just as we represent an operation that computes a value—as a trapezoid that contains the name of the action. Arrows point to the action box from any inputs (registers or constants). We also associate a button with the action. Pushing the button makes the action happen. To make a controller push an action button we use a new kind of instruction called `perform`. Thus, the action of printing the contents of register `a` is represented in a controller sequence by the instruction

```
(perform (op print) (reg a))
```

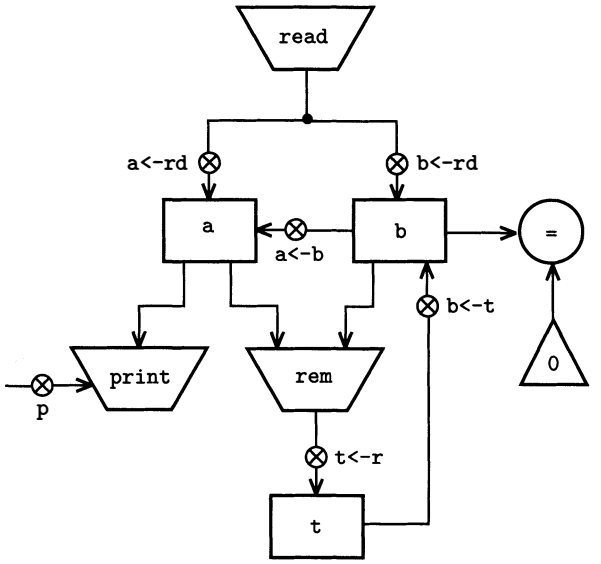
Figure 5.4 shows the data paths and controller for the new GCD machine. Instead of having the machine stop after printing the answer, we have made it start over, so that it repeatedly reads a pair of numbers, computes their GCD, and prints the result. This structure is like the driver loops we used in the interpreters of chapter 4.

5.1.2 Abstraction in Machine Design

We will often define a machine to include "primitive" operations that are actually very complex. For example, in sections 5.4 and 5.5 we will treat Scheme's environment manipulations as primitive. Such abstraction is valuable because it allows us to ignore the details of parts of a machine so that we can concentrate on other aspects of the design. The fact that we have swept a lot of complexity under the rug, however, does not mean that a machine design is unrealistic. We can always replace the complex "primitives" by simpler primitive operations.

Consider the GCD machine. The machine has an instruction that computes the remainder of the contents of registers `a` and `b` and assigns the result to register `t`. If we want to construct the GCD machine without using a primitive remainder operation, we must specify how to compute remainders in terms of simpler operations, such as subtraction. Indeed, we can write a Scheme procedure that finds remainders in this way:

```
(define (remainder n d)
  (if (< n d)
      n
      (remainder (- n d) d)))
```




```
(controller
 gcd-loop
   (assign a (op read))
   (assign b (op read))
 test-b
   (test (op =) (reg b) (const 0))
   (branch (label gcd-done))
   (assign t (op rem) (reg a) (reg b))
   (assign a (reg b))
   (assign b (reg t))
   (goto (label test-b))
 gcd-done
   (perform (op print) (reg a))
   (goto (label gcd-loop)))
```

Figure 5.4 A GCD machine that reads inputs and prints results.

We can thus replace the remainder operation in the GCD machine's data paths with a subtraction operation and a comparison test. Figure 5.5 shows the data paths and controller for the elaborated machine. The instruction

```
(assign t (op rem) (reg a) (reg b))
```

in the GCD controller definition is replaced by a sequence of instructions that contains a loop, as shown in figure 5.6.

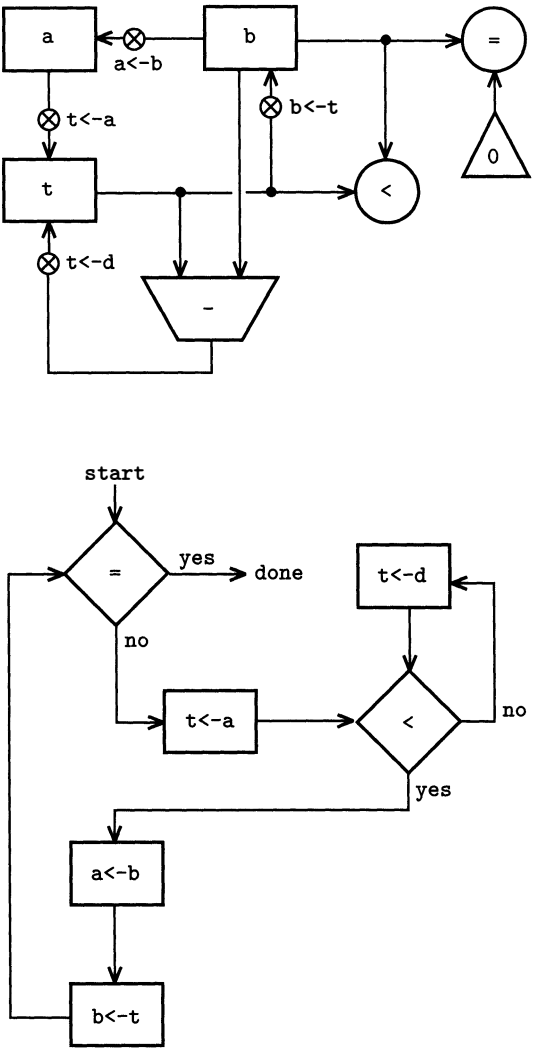


Figure 5.5 Data paths and controller for the elaborated GCD machine.

```
(controller
 test-b
   (test (op =) (reg b) (const 0))
   (branch (label gcd-done))
   (assign t (reg a))
 rem-loop
   (test (op <) (reg t) (reg b))
   (branch (label rem-done))
   (assign t (op -) (reg t) (reg b))
   (goto (label rem-loop))
 rem-done
   (assign a (reg b))
   (assign b (reg t))
   (goto (label test-b))
 gcd-done)
```

Figure 5.6 Controller instruction sequence for the GCD machine in figure 5.5.

Exercise 5.3

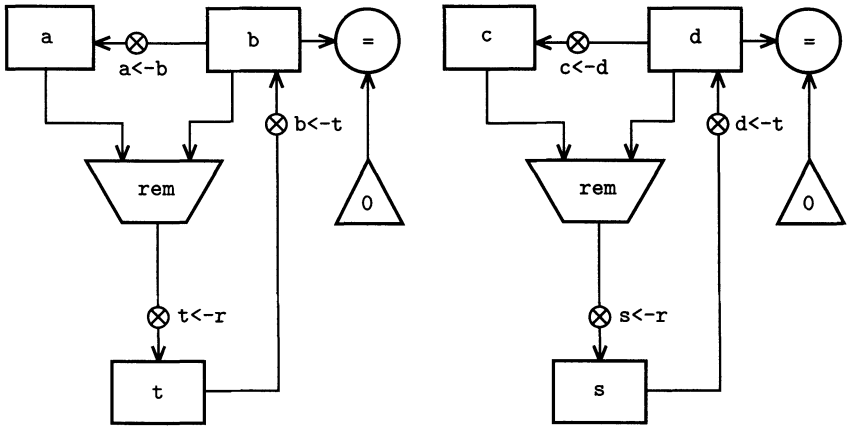
Design a machine to compute square roots using Newton's method, as described in section 1.1.7:

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

Begin by assuming that `good-enough?` and `improve` operations are available as primitives. Then show how to expand these in terms of arithmetic operations. Describe each version of the `sqrt` machine design by drawing a data-path diagram and writing a controller definition in the register-machine language.

5.1.3 Subroutines

When designing a machine to perform a computation, we would often prefer to arrange for components to be shared by different parts of the computation rather than duplicate the components. Consider a machine that includes two GCD computations—one that finds the GCD of the contents of registers `a` and `b` and one that finds the GCD of the contents of registers `c` and `d`. We might start by assuming we have a primitive `gcd` operation, then expand the two instances of `gcd` in terms of more primitive operations. Figure 5.7 shows just the GCD portions of the resulting




```
gcd-1
 (test (op =) (reg b) (const 0))
 (branch (label after-gcd-1))
 (assign t (op rem) (reg a) (reg b))
 (assign a (reg b))
 (assign b (reg t))
 (goto (label gcd-1))
after-gcd-1
   ⋮
gcd-2
 (test (op =) (reg d) (const 0))
 (branch (label after-gcd-2))
 (assign s (op rem) (reg c) (reg d))
 (assign c (reg d))
 (assign d (reg s))
 (goto (label gcd-2))
after-gcd-2
```

Figure 5.7 Portions of the data paths and controller sequence for a machine with two GCD computations.

machine's data paths, without showing how they connect to the rest of the machine. The figure also shows the corresponding portions of the machine's controller sequence.

This machine has two remainder operation boxes and two boxes for testing equality. If the duplicated components are complicated, as is the remainder box, this will not be an economical way to build the machine. We can avoid duplicating the data-path components by using the same components for both GCD computations, provided that doing so will not affect the rest of the larger machine's computation. If the values in registers `a` and `b` are not needed by the time the controller gets to `gcd-2` (or if these values can be moved to other registers for safekeeping), we can

```
gcd-1
 (test (op =) (reg b) (const 0))
 (branch (label after-gcd-1))
 (assign t (op rem) (reg a) (reg b))
 (assign a (reg b))
 (assign b (reg t))
 (goto (label gcd-1))
after-gcd-1
  ⋮
gcd-2
 (test (op =) (reg b) (const 0))
 (branch (label after-gcd-2))
 (assign t (op rem) (reg a) (reg b))
 (assign a (reg b))
 (assign b (reg t))
 (goto (label gcd-2))
after-gcd-2
```

Figure 5.8 Portions of the controller sequence for a machine that uses the same data-path components for two different GCD computations.

change the machine so that it uses registers `a` and `b`, rather than registers `c` and `d`, in computing the second GCD as well as the first. If we do this, we obtain the controller sequence shown in figure 5.8.

We have removed the duplicate data-path components (so that the data paths are again as in figure 5.1), but the controller now has two GCD sequences that differ only in their entry-point labels. It would be better to replace these two sequences by branches to a single sequence—a `gcd` *subroutine*—at the end of which we branch back to the correct place in the main instruction sequence. We can accomplish this as follows: Before branching to `gcd`, we place a distinguishing value (such as 0 or 1) into a special register, `continue`. At the end of the `gcd` subroutine we return either to `after-gcd-1` or to `after-gcd-2`, depending on the value of the `continue` register. Figure 5.9 shows the relevant portion of the resulting controller sequence, which includes only a single copy of the `gcd` instructions.

This is a reasonable approach for handling small problems, but it would be awkward if there were many instances of GCD computations in the controller sequence. To decide where to continue executing after the `gcd` subroutine, we would need tests in the data paths and branch instructions in the controller for all the places that use `gcd`. A more powerful method for implementing subroutines is to have the `continue` register hold the label of the entry point in the controller sequence at which execution should continue when the subroutine is finished. Implementing

```
gcd
 (test (op =) (reg b) (const 0))
 (branch (label gcd-done))
 (assign t (op rem) (reg a) (reg b))
 (assign a (reg b))
 (assign b (reg t))
 (goto (label gcd))
gcd-done
 (test (op =) (reg continue) (const 0))
 (branch (label after-gcd-1))
 (goto (label after-gcd-2))
  ⋮
;; Before branching to gcd from the first place where
;; it is needed, we place 0 in the continue register
 (assign continue (const 0))
 (goto (label gcd))
after-gcd-1
  ⋮
;; Before the second use of gcd, we place 1 in the continue register
 (assign continue (const 1))
 (goto (label gcd))
after-gcd-2
```

Figure 5.9 Using a `continue` register to avoid the duplicate controller sequence in figure 5.8.

```
gcd
 (test (op =) (reg b) (const 0))
 (branch (label gcd-done))
 (assign t (op rem) (reg a) (reg b))
 (assign a (reg b))
 (assign b (reg t))
 (goto (label gcd))
gcd-done
 (goto (reg continue))
   ⋮
;; Before calling gcd, we assign to continue
;; the label to which gcd should return.
 (assign continue (label after-gcd-1))
 (goto (label gcd))
after-gcd-1
   ⋮
;; Here is the second call to gcd, with a different continuation.
 (assign continue (label after-gcd-2))
 (goto (label gcd))
after-gcd-2
```

Figure 5.10 Assigning labels to the `continue` register simplifies and generalizes the strategy shown in figure 5.9.

this strategy requires a new kind of connection between the data paths and the controller of a register machine: There must be a way to assign to a register a label in the controller sequence in such a way that this value can be fetched from the register and used to continue execution at the designated entry point.

To reflect this ability, we will extend the `assign` instruction of the register-machine language to allow a register to be assigned as value a label from the controller sequence (as a special kind of constant). We will also extend the `goto` instruction to allow execution to continue at the entry point described by the contents of a register rather than only at an entry point described by a constant label. Using these new constructs we can terminate the gcd subroutine with a branch to the location stored in the `continue` register. This leads to the controller sequence shown in figure 5.10.

A machine with more than one subroutine could use multiple continuation registers (e.g., `gcd-continue`, `factorial-continue`) or we could have all subroutines share a single `continue` register. Sharing is more economical, but we must be careful if we have a subroutine (`sub1`) that calls another subroutine (`sub2`). Unless `sub1` saves the contents of `continue` in some other register before setting up `continue` for the call to `sub2`, `sub1` will not know where to go when it is finished. The mechanism developed in the next section to handle recursion also provides a better solution to this problem of nested subroutine calls.

5.1.4 Using a Stack to Implement Recursion

With the ideas illustrated so far, we can implement any iterative process by specifying a register machine that has a register corresponding to each state variable of the process. The machine repeatedly executes a controller loop, changing the contents of the registers, until some termination condition is satisfied. At each point in the controller sequence, the state of the machine (representing the state of the iterative process) is completely determined by the contents of the registers (the values of the state variables).

Implementing recursive processes, however, requires an additional mechanism. Consider the following recursive method for computing factorials, which we first examined in section 1.2.1:

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

As we see from the procedure, computing $n!$ requires computing $(n-1)!$. Our GCD machine, modeled on the procedure

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

similarly had to compute another GCD. But there is an important difference between the `gcd` procedure, which reduces the original computation to a new GCD computation, and `factorial`, which requires computing another factorial as a subproblem. In GCD, the answer to the new GCD computation is the answer to the original problem. To compute the next GCD, we simply place the new arguments in the input registers of the GCD machine and reuse the machine's data paths by executing the same controller sequence. When the machine is finished solving the final GCD problem, it has completed the entire computation.

In the case of factorial (or any recursive process) the answer to the new factorial subproblem is not the answer to the original problem. The value obtained for $(n-1)!$ must be multiplied by n to get the final answer. If we try to imitate the GCD design, and solve the factorial subproblem by decrementing the `n` register and rerunning the factorial machine, we will no longer have available the old value of `n` by which to multiply the result. We thus need a second factorial machine to work on the subproblem. This second factorial computation itself has a factorial subproblem, which requires a third factorial machine, and so on. Since each factorial machine contains another factorial machine within it, the total machine contains an infinite nest of similar machines and hence cannot be constructed from a fixed, finite number of parts.

Nevertheless, we can implement the factorial process as a register machine if we can arrange to use the same components for each nested instance of the machine. Specifically, the machine that computes $n!$ should use the same components to work on the subproblem of computing $(n-1)!$, on the subproblem for $(n-2)!$, and so on. This is plausible because, although the factorial process dictates that an unbounded number of copies of the same machine are needed to perform a computation, only one of these copies needs to be active at any given time. When the machine encounters a recursive subproblem, it can suspend work on the main problem, reuse the same physical parts to work on the subproblem, then continue the suspended computation.

In the subproblem, the contents of the registers will be different than they were in the main problem. (In this case the `n` register is decre-

mented.) In order to be able to continue the suspended computation, the machine must save the contents of any registers that will be needed after the subproblem is solved so that these can be restored to continue the suspended computation. In the case of factorial, we will save the old value of `n`, to be restored when we are finished computing the factorial of the decremented `n` register.[2]

Since there is no *a priori* limit on the depth of nested recursive calls, we may need to save an arbitrary number of register values. These values must be restored in the reverse of the order in which they were saved, since in a nest of recursions the last subproblem to be entered is the first to be finished. This dictates the use of a *stack*, or "last in, first out" data structure, to save register values. We can extend the register-machine language to include a stack by adding two kinds of instructions: Values are placed on the stack using a `save` instruction and restored from the stack using a `restore` instruction. After a sequence of values has been `saved` on the stack, a sequence of `restores` will retrieve these values in reverse order.[3]

With the aid of the stack, we can reuse a single copy of the factorial machine's data paths for each factorial subproblem. There is a similar design issue in reusing the controller sequence that operates the data paths. To reexecute the factorial computation, the controller cannot simply loop back to the beginning, as with an iterative process, because after solving the $(n-1)!$ subproblem the machine must still multiply the result by n. The controller must suspend its computation of $n!$, solve the $(n-1)!$ subproblem, then continue its computation of $n!$. This view of the factorial computation suggests the use of the subroutine mechanism described in section 5.1.3, which has the controller use a `continue` register to transfer to the part of the sequence that solves a subproblem and then continue where it left off on the main problem. We can thus make a factorial subroutine that returns to the entry point stored in the `continue` register. Around each subroutine call, we save and restore `continue` just as we

[2]One might argue that we don't need to save the old `n`; after we decrement it and solve the subproblem, we could simply increment it to recover the old value. Although this strategy works for factorial, it cannot work in general, since the old value of a register cannot always be computed from the new one.

[3]In section 5.3 we will see how to implement a stack in terms of more primitive operations.

do the n register, since each "level" of the factorial computation will use the same continue register. That is, the factorial subroutine must put a new value in continue when it calls itself for a subproblem, but it will need the old value in order to return to the place that called it to solve a subproblem.

Figure 5.11 shows the data paths and controller for a machine that implements the recursive factorial procedure. The machine has a stack and three registers, called n, val, and continue. To simplify the data-path diagram, we have not named the register-assignment buttons, only the stack-operation buttons (sc and sn to save registers, rc and rn to restore registers). To operate the machine, we put in register n the number whose factorial we wish to compute and start the machine. When the machine reaches fact-done, the computation is finished and the answer will be found in the val register. In the controller sequence, n and continue are saved before each recursive call and restored upon return from the call. Returning from a call is accomplished by branching to the location stored in continue. Continue is initialized when the machine starts so that the last return will go to fact-done. The val register, which holds the result of the factorial computation, is not saved before the recursive call, because the old contents of val is not useful after the subroutine returns. Only the new value, which is the value produced by the subcomputation, is needed.

Although in principle the factorial computation requires an infinite machine, the machine in figure 5.11 is actually finite except for the stack, which is potentially unbounded. Any particular physical implementation of a stack, however, will be of finite size, and this will limit the depth of recursive calls that can be handled by the machine. This implementation of factorial illustrates the general strategy for realizing recursive algorithms as ordinary register machines augmented by stacks. When a recursive subproblem is encountered, we save on the stack the registers whose current values will be required after the subproblem is solved, solve the recursive subproblem, then restore the saved registers and continue execution on the main problem. The continue register must always be saved. Whether there are other registers that need to be saved depends on the particular machine, since not all recursive computations need the original values of registers that are modified during solution of the subproblem (see exercise 5.4).

A double recursion

Let us examine a more complex recursive process, the tree-recursive computation of the Fibonacci numbers, which we introduced in section 1.2.2:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Just as with factorial, we can implement the recursive Fibonacci computation as a register machine with registers `n`, `val`, and `continue`. The machine is more complex than the one for factorial, because there are two places in the controller sequence where we need to perform recursive calls—once to compute Fib($n-1$) and once to compute Fib($n-2$). To set up for each of these calls, we save the registers whose values will be needed later, set the `n` register to the number whose Fib we need to compute recursively ($n-1$ or $n-2$), and assign to `continue` the entry point in the main sequence to which to return (`afterfib-n-1` or `afterfib-n-2`, respectively). We then go to `fib-loop`. When we return from the recursive call, the answer is in `val`. Figure 5.12 shows the controller sequence for this machine.

Exercise 5.4

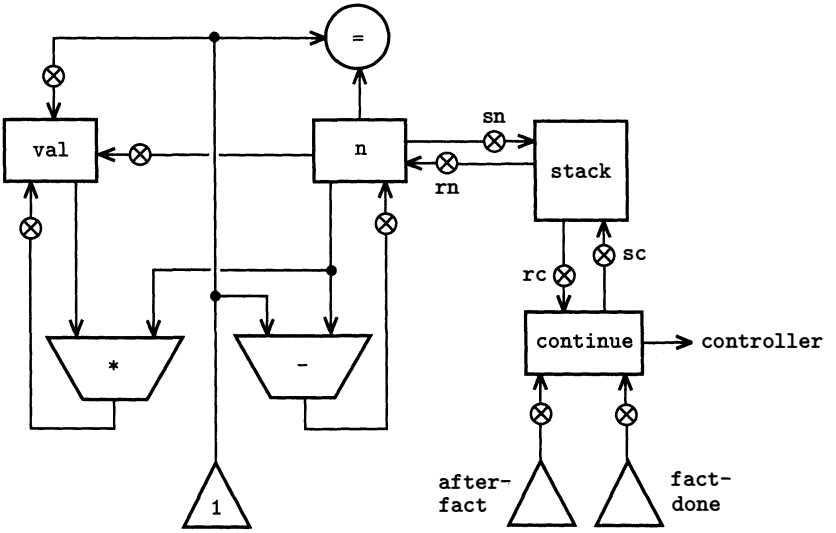
Specify register machines that implement each of the following procedures. For each machine, write a controller instruction sequence and draw a diagram showing the data paths.

a. Recursive exponentiation:

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

b. Iterative exponentiation:

```
(define (expt b n)
  (define (expt-iter counter product)
    (if (= counter 0)
        product
        (expt-iter (- counter 1) (* b product))))
  (expt-iter n 1))
```




```
(controller
   (assign continue (label fact-done))      ; set up final return address
 fact-loop
   (test (op =) (reg n) (const 1))
   (branch (label base-case))
   ;; Set up for the recursive call by saving n and continue.
   ;; Set up continue so that the computation will continue
   ;; at after-fact when the subroutine returns.
   (save continue)
   (save n)
   (assign n (op -) (reg n) (const 1))
   (assign continue (label after-fact))
   (goto (label fact-loop))
 after-fact
   (restore n)
   (restore continue)
   (assign val (op *) (reg n) (reg val))    ; val now contains n(n - 1)!
   (goto (reg continue))                    ; return to caller
 base-case
   (assign val (const 1))                   ; base case: 1! = 1
   (goto (reg continue))                    ; return to caller
 fact-done)
```

Figure 5.11 A recursive factorial machine.

```
(controller
   (assign continue (label fib-done))
 fib-loop
   (test (op <) (reg n) (const 2))
   (branch (label immediate-answer))
   ;; set up to compute Fib(n − 1)
   (save continue)
   (assign continue (label afterfib-n-1))
   (save n)                              ; save old value of n
   (assign n (op -) (reg n) (const 1)) ; clobber n to n − 1
   (goto (label fib-loop))               ; perform recursive call
 afterfib-n-1                            ; upon return, val contains Fib(n − 1)
   (restore n)
   (restore continue)
   ;; set up to compute Fib(n − 2)
   (assign n (op -) (reg n) (const 2))
   (save continue)
   (assign continue (label afterfib-n-2))
   (save val)                            ; save Fib(n − 1)
   (goto (label fib-loop))
 afterfib-n-2                            ; upon return, val contains Fib(n − 2)
   (assign n (reg val))                  ; n now contains Fib(n − 2)
   (restore val)                         ; val now contains Fib(n − 1)
   (restore continue)
   (assign val                           ; Fib(n − 1) + Fib(n − 2)
           (op +) (reg val) (reg n))
   (goto (reg continue))                 ; return to caller, answer is in val
 immediate-answer
   (assign val (reg n))                  ; base case: Fib(n) = n
   (goto (reg continue))
 fib-done)
```

Figure 5.12 Controller for a machine to compute Fibonacci numbers.

Exercise 5.5

Hand-simulate the factorial and Fibonacci machines, using some nontrivial input (requiring execution of at least one recursive call). Show the contents of the stack at each significant point in the execution.

Exercise 5.6

Ben Bitdiddle observes that the Fibonacci machine's controller sequence has an extra `save` and an extra `restore`, which can be removed to make a faster machine. Where are these instructions?

5.1.5 Instruction Summary

A controller instruction in our register-machine language has one of the following forms, where each ⟨ *input*$_i$ ⟩ is either `(reg` ⟨ *register-name* ⟩`)` or `(const` ⟨ *constant-value* ⟩`)`.

These instructions were introduced in section 5.1.1:

`(assign` ⟨*register-name*⟩ `(reg` ⟨*register-name*⟩`))`

`(assign` ⟨*register-name*⟩ `(const` ⟨*constant-value*⟩`))`

`(assign` ⟨*register-name*⟩ `(op` ⟨*operation-name*⟩`)` ⟨$input_1$⟩ ... ⟨$input_n$⟩`)`

`(perform (op` ⟨*operation-name*⟩`)` ⟨$input_1$⟩ ... ⟨$input_n$⟩`)`

`(test (op` ⟨*operation-name*⟩`)` ⟨$input_1$⟩ ... ⟨$input_n$⟩`)`

`(branch (label` ⟨*label-name*⟩`))`

`(goto (label` ⟨*label-name*⟩`))`

The use of registers to hold labels was introduced in section 5.1.3:

`(assign` ⟨*register-name*⟩ `(label` ⟨*label-name*⟩`))`

`(goto (reg` ⟨*register-name*⟩`))`

Instructions to use the stack were introduced in section 5.1.4:

`(save` ⟨*register-name*⟩`)`

`(restore` ⟨*register-name*⟩`)`

The only kind of ⟨*constant-value*⟩ we have seen so far is a number, but later we will use strings, symbols, and lists. For example, `(const "abc")` is the string `"abc"`, `(const abc)` is the symbol `abc`, `(const (a b c))` is the list `(a b c)`, and `(const ())` is the empty list.

5.2 A Register-Machine Simulator

In order to gain a good understanding of the design of register machines, we must test the machines we design to see if they perform as expected. One way to test a design is to hand-simulate the operation of the controller, as in exercise 5.5. But this is extremely tedious for all but the simplest machines. In this section we construct a simulator for machines described in the register-machine language. The simulator is a Scheme program with four interface procedures. The first uses a description of a register machine to construct a model of the machine (a data structure

whose parts correspond to the parts of the machine to be simulated), and the other three allow us to simulate the machine by manipulating the model:

`(make-machine` ⟨*register-names*⟩ ⟨*operations*⟩ ⟨*controller*⟩`)`
constructs and returns a model of the machine with the given registers, operations, and controller.

`(set-register-contents!` ⟨*machine-model*⟩ ⟨*register-name*⟩ ⟨*value*⟩`)`
stores a value in a simulated register in the given machine.

`(get-register-contents` ⟨*machine-model*⟩ ⟨*register-name*⟩`)`
returns the contents of a simulated register in the given machine.

`(start` ⟨*machine-model*⟩`)`
simulates the execution of the given machine, starting from the beginning of the controller sequence and stopping when it reaches the end of the sequence.

As an example of how these procedures are used, we can define `gcd-machine` to be a model of the GCD machine of section 5.1.1 as follows:

```
(define gcd-machine
  (make-machine
   '(a b t)
   (list (list 'rem remainder) (list '= =))
   '(test-b
       (test (op =) (reg b) (const 0))
       (branch (label gcd-done))
       (assign t (op rem) (reg a) (reg b))
       (assign a (reg b))
       (assign b (reg t))
       (goto (label test-b))
     gcd-done)))
```

The first argument to `make-machine` is a list of register names. The next argument is a table (a list of two-element lists) that pairs each operation name with a Scheme procedure that implements the operation (that is, produces the same output value given the same input values). The last argument specifies the controller as a list of labels and machine instructions, as in section 5.1.

To compute GCDs with this machine, we set the input registers, start the machine, and examine the result when the simulation terminates:

```
(set-register-contents! gcd-machine 'a 206)
done

(set-register-contents! gcd-machine 'b 40)
done

(start gcd-machine)
done

(get-register-contents gcd-machine 'a)
2
```

This computation will run much more slowly than a `gcd` procedure written in Scheme, because we will simulate low-level machine instructions, such as `assign`, by much more complex operations.

Exercise 5.7
Use the simulator to test the machines you designed in exercise 5.4.

5.2.1 The Machine Model

The machine model generated by `make-machine` is represented as a procedure with local state using the message-passing techniques developed in chapter 3. To build this model, `make-machine` begins by calling the procedure `make-new-machine` to construct the parts of the machine model that are common to all register machines. This basic machine model constructed by `make-new-machine` is essentially a container for some registers and a stack, together with an execution mechanism that processes the controller instructions one by one.

`Make-machine` then extends this basic model (by sending it messages) to include the registers, operations, and controller of the particular machine being defined. First it allocates a register in the new machine for each of the supplied register names and installs the designated operations in the machine. Then it uses an *assembler* (described below in section 5.2.2) to transform the controller list into instructions for the new machine and installs these as the machine's instruction sequence. `Make-machine` returns as its value the modified machine model.

```
(define (make-machine register-names ops controller-text)
  (let ((machine (make-new-machine)))
    (for-each (lambda (register-name)
                ((machine 'allocate-register) register-name))
              register-names)
    ((machine 'install-operations) ops)
    ((machine 'install-instruction-sequence)
     (assemble controller-text machine))
    machine))
```

Registers

We will represent a register as a procedure with local state, as in chapter 3. The procedure `make-register` creates a register that holds a value that can be accessed or changed:

```
(define (make-register name)
  (let ((contents '*unassigned*))
    (define (dispatch message)
      (cond ((eq? message 'get) contents)
            ((eq? message 'set)
             (lambda (value) (set! contents value)))
            (else
             (error "Unknown request -- REGISTER" message))))
    dispatch))
```

The following procedures are used to access registers:

```
(define (get-contents register)
  (register 'get))

(define (set-contents! register value)
  ((register 'set) value))
```

The stack

We can also represent a stack as a procedure with local state. The procedure `make-stack` creates a stack whose local state consists of a list of the items on the stack. A stack accepts requests to `push` an item onto the stack, to `pop` the top item off the stack and return it, and to `initialize` the stack to empty.

```
(define (make-stack)
  (let ((s '()))
    (define (push x)
      (set! s (cons x s)))
    (define (pop)
      (if (null? s)
          (error "Empty stack -- POP")
          (let ((top (car s)))
            (set! s (cdr s))
            top)))
    (define (initialize)
      (set! s '())
      'done)
    (define (dispatch message)
      (cond ((eq? message 'push) push)
            ((eq? message 'pop) (pop))
            ((eq? message 'initialize) (initialize))
            (else (error "Unknown request -- STACK"
                         message))))
    dispatch))
```

The following procedures are used to access stacks:

```
(define (pop stack)
  (stack 'pop))

(define (push stack value)
  ((stack 'push) value))
```

The basic machine

The `make-new-machine` procedure, shown in figure 5.13, constructs an object whose local state consists of a stack, an initially empty instruction sequence, a list of operations that initially contains an operation to initialize the stack, and a *register table* that initially contains two registers, named `flag` and `pc` (for "program counter"). The internal procedure `allocate-register` adds new entries to the register table, and the internal procedure `lookup-register` looks up registers in the table.

The `flag` register is used to control branching in the simulated machine. `Test` instructions set the contents of `flag` to the result of the test (true or false). `Branch` instructions decide whether or not to branch by examining the contents of `flag`.

The `pc` register determines the sequencing of instructions as the machine runs. This sequencing is implemented by the internal procedure `execute`. In the simulation model, each machine instruction is a data structure that includes a procedure of no arguments, called the *instruc-*

tion execution procedure, such that calling this procedure simulates executing the instruction. As the simulation runs, pc points to the place in the instruction sequence beginning with the next instruction to be executed. `Execute` gets that instruction, executes it by calling the instruction execution procedure, and repeats this cycle until there are no more instructions to execute (i.e., until pc points to the end of the instruction sequence).

As part of its operation, each instruction execution procedure modifies pc to indicate the next instruction to be executed. `Branch` and `goto` instructions change pc to point to the new destination. All other instructions simply advance pc, making it point to the next instruction in the sequence. Observe that each call to `execute` calls `execute` again, but this does not produce an infinite loop because running the instruction execution procedure changes the contents of pc.

`Make-new-machine` returns a `dispatch` procedure that implements message-passing access to the internal state. Notice that starting the machine is accomplished by setting pc to the beginning of the instruction sequence and calling `execute`.

For convenience, we provide an alternate procedural interface to a machine's `start` operation, as well as procedures to set and examine register contents, as specified at the beginning of section 5.2:

```
(define (start machine)
  (machine 'start))

(define (get-register-contents machine register-name)
  (get-contents (get-register machine register-name)))

(define (set-register-contents! machine register-name value)
  (set-contents! (get-register machine register-name) value)
  'done)
```

These procedures (and many procedures in sections 5.2.2 and 5.2.3) use the following to look up the register with a given name in a given machine:

```
(define (get-register machine reg-name)
  ((machine 'get-register) reg-name))
```

```
(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()))
    (let ((the-ops
           (list (list 'initialize-stack
                       (lambda () (stack 'initialize)))))
          (register-table
           (list (list 'pc pc) (list 'flag flag))))
      (define (allocate-register name)
        (if (assoc name register-table)
            (error "Multiply defined register: " name)
            (set! register-table
                  (cons (list name (make-register name))
                        register-table)))
        'register-allocated)
      (define (lookup-register name)
        (let ((val (assoc name register-table)))
          (if val
              (cadr val)
              (error "Unknown register:" name))))
      (define (execute)
        (let ((insts (get-contents pc)))
          (if (null? insts)
              'done
              (begin
                ((instruction-execution-proc (car insts)))
                (execute)))))
      (define (dispatch message)
        (cond ((eq? message 'start)
               (set-contents! pc the-instruction-sequence)
               (execute))
              ((eq? message 'install-instruction-sequence)
               (lambda (seq) (set! the-instruction-sequence seq)))
              ((eq? message 'allocate-register) allocate-register)
              ((eq? message 'get-register) lookup-register)
              ((eq? message 'install-operations)
               (lambda (ops) (set! the-ops (append the-ops ops))))
              ((eq? message 'stack) stack)
              ((eq? message 'operations) the-ops)
              (else (error "Unknown request -- MACHINE" message))))
      dispatch)))
```

Figure 5.13 The `make-new-machine` procedure, which implements the basic machine model.

5.2.2 The Assembler

The assembler transforms the sequence of controller expressions for a machine into a corresponding list of machine instructions, each with its execution procedure. Overall, the assembler is much like the evaluators we studied in chapter 4—there is an input language (in this case, the register-machine language) and we must perform an appropriate action for each type of expression in the language.

The technique of producing an execution procedure for each instruction is just what we used in section 4.1.7 to speed up the evaluator by separating analysis from runtime execution. As we saw in chapter 4, much useful analysis of Scheme expressions could be performed without knowing the actual values of variables. Here, analogously, much useful analysis of register-machine-language expressions can be performed without knowing the actual contents of machine registers. For example, we can replace references to registers by pointers to the register objects, and we can replace references to labels by pointers to the place in the instruction sequence that the label designates.

Before it can generate the instruction execution procedures, the assembler must know what all the labels refer to, so it begins by scanning the controller text to separate the labels from the instructions. As it scans the text, it constructs both a list of instructions and a table that associates each label with a pointer into that list. Then the assembler augments the instruction list by inserting the execution procedure for each instruction.

The `assemble` procedure is the main entry to the assembler. It takes the controller text and the machine model as arguments and returns the instruction sequence to be stored in the model. `Assemble` calls `extract-labels` to build the initial instruction list and label table from the supplied controller text. The second argument to `extract-labels` is a procedure to be called to process these results: This procedure uses `update-insts!` to generate the instruction execution procedures and insert them into the instruction list, and returns the modified list.

```
(define (assemble controller-text machine)
  (extract-labels controller-text
    (lambda (insts labels)
      (update-insts! insts labels machine)
      insts)))
```

`Extract-labels` takes as arguments a list `text` (the sequence of controller instruction expressions) and a `receive` procedure. `Receive` will be called with two values: (1) a list `insts` of instruction data structures, each containing an instruction from `text`; and (2) a table called `labels`,

which associates each label from `text` with the position in the list `insts` that the label designates.

```
(define (extract-labels text receive)
  (if (null? text)
      (receive '() '())
      (extract-labels (cdr text)
       (lambda (insts labels)
         (let ((next-inst (car text)))
           (if (symbol? next-inst)
               (receive insts
                        (cons (make-label-entry next-inst
                                                insts)
                              labels))
               (receive (cons (make-instruction next-inst)
                              insts)
                        labels)))))))
```

`Extract-labels` works by sequentially scanning the elements of the `text` and accumulating the `insts` and the `labels`. If an element is a symbol (and thus a label) an appropriate entry is added to the `labels` table. Otherwise the element is accumulated onto the `insts` list.[4]

---

[4]Using the `receive` procedure here is a way to get `extract-labels` to effectively return two values—`labels` and `insts`—without explicitly making a compound data structure to hold them. An alternative implementation, which returns an explicit pair of values, is

```
(define (extract-labels text)
  (if (null? text)
      (cons '() '())
      (let ((result (extract-labels (cdr text))))
        (let ((insts (car result)) (labels (cdr result)))
          (let ((next-inst (car text)))
            (if (symbol? next-inst)
                (cons insts
                      (cons (make-label-entry next-inst insts) labels))
                (cons (cons (make-instruction next-inst) insts)
                      labels)))))))
```

which would be called by `assemble` as follows:

```
(define (assemble controller-text machine)
  (let ((result (extract-labels controller-text)))
    (let ((insts (car result)) (labels (cdr result)))
      (update-insts! insts labels machine)
      insts)))
```

You can consider our use of `receive` as demonstrating an elegant way to return multiple values, or simply an excuse to show off a programming trick. An argument like `receive` that is the next procedure to be invoked is called a "continuation." Recall that we also used continuations to implement the backtracking control structure in the `amb` evaluator in section 4.3.3.

`Update-insts!` modifies the instruction list, which initially contains only the text of the instructions, to include the corresponding execution procedures:

```
(define (update-insts! insts labels machine)
  (let ((pc (get-register machine 'pc))
        (flag (get-register machine 'flag))
        (stack (machine 'stack))
        (ops (machine 'operations)))
    (for-each
     (lambda (inst)
       (set-instruction-execution-proc!
        inst
        (make-execution-procedure
         (instruction-text inst) labels machine
         pc flag stack ops)))
     insts)))
```

The machine instruction data structure simply pairs the instruction text with the corresponding execution procedure. The execution procedure is not yet available when `extract-labels` constructs the instruction, and is inserted later by `update-insts!`.

```
(define (make-instruction text)
  (cons text '()))

(define (instruction-text inst)
  (car inst))

(define (instruction-execution-proc inst)
  (cdr inst))

(define (set-instruction-execution-proc! inst proc)
  (set-cdr! inst proc))
```

The instruction text is not used by our simulator, but it is handy to keep around for debugging (see exercise 5.16).

Elements of the label table are pairs:

```
(define (make-label-entry label-name insts)
  (cons label-name insts))
```

Entries will be looked up in the table with

```
(define (lookup-label labels label-name)
  (let ((val (assoc label-name labels)))
    (if val
        (cdr val)
        (error "Undefined label -- ASSEMBLE" label-name))))
```

Exercise 5.8

The following register-machine code is ambiguous, because the label `here` is defined more than once:

```
start
  (goto (label here))
here
  (assign a (const 3))
  (goto (label there))
here
  (assign a (const 4))
  (goto (label there))
there
```

With the simulator as written, what will the contents of register `a` be when control reaches `there`? Modify the `extract-labels` procedure so that the assembler will signal an error if the same label name is used to indicate two different locations.

5.2.3 Generating Execution Procedures for Instructions

The assembler calls `make-execution-procedure` to generate the execution procedure for an instruction. Like the `analyze` procedure in the evaluator of section 4.1.7, this dispatches on the type of instruction to generate the appropriate execution procedure.

```
(define (make-execution-procedure inst labels machine
                                  pc flag stack ops)
  (cond ((eq? (car inst) 'assign)
         (make-assign inst machine labels ops pc))
        ((eq? (car inst) 'test)
         (make-test inst machine labels ops flag pc))
        ((eq? (car inst) 'branch)
         (make-branch inst machine labels flag pc))
        ((eq? (car inst) 'goto)
         (make-goto inst machine labels pc))
        ((eq? (car inst) 'save)
         (make-save inst machine stack pc))
        ((eq? (car inst) 'restore)
         (make-restore inst machine stack pc))
        ((eq? (car inst) 'perform)
         (make-perform inst machine labels ops pc))
        (else (error "Unknown instruction type -- ASSEMBLE"
                     inst))))
```

For each type of instruction in the register-machine language, there is a generator that builds an appropriate execution procedure. The details of these procedures determine both the syntax and meaning of the

individual instructions in the register-machine language. We use data abstraction to isolate the detailed syntax of register-machine expressions from the general execution mechanism, as we did for evaluators in section 4.1.2, by using syntax procedures to extract and classify the parts of an instruction.

`Assign` instructions

The `make-assign` procedure handles `assign` instructions:

```
(define (make-assign inst machine labels operations pc)
  (let ((target
         (get-register machine (assign-reg-name inst)))
        (value-exp (assign-value-exp inst)))
    (let ((value-proc
           (if (operation-exp? value-exp)
               (make-operation-exp
                value-exp machine labels operations)
               (make-primitive-exp
                (car value-exp) machine labels))))
      (lambda ()                ; execution procedure for assign
        (set-contents! target (value-proc))
        (advance-pc pc)))))
```

`Make-assign` extracts the target register name (the second element of the instruction) and the value expression (the rest of the list that forms the instruction) from the `assign` instruction using the selectors

```
(define (assign-reg-name assign-instruction)
  (cadr assign-instruction))

(define (assign-value-exp assign-instruction)
  (cddr assign-instruction))
```

The register name is looked up with `get-register` to produce the target register object. The value expression is passed to `make-operation-exp` if the value is the result of an operation, and to `make-primitive-exp` otherwise. These procedures (shown below) parse the value expression and produce an execution procedure for the value. This is a procedure of no arguments, called `value-proc`, which will be evaluated during the simulation to produce the actual value to be assigned to the register. Notice that the work of looking up the register name and parsing the value expression is performed just once, at assembly time, not every time the instruction is simulated. This saving of work is the reason we use

execution procedures, and corresponds directly to the saving in work we obtained by separating program analysis from execution in the evaluator of section 4.1.7.

The result returned by `make-assign` is the execution procedure for the `assign` instruction. When this procedure is called (by the machine model's `execute` procedure), it sets the contents of the target register to the result obtained by executing `value-proc`. Then it advances the `pc` to the next instruction by running the procedure

```
(define (advance-pc pc)
  (set-contents! pc (cdr (get-contents pc))))
```

`Advance-pc` is the normal termination for all instructions except `branch` and `goto`.

`Test`, `branch`, and `goto` instructions

`Make-test` handles `test` instructions in a similar way. It extracts the expression that specifies the condition to be tested and generates an execution procedure for it. At simulation time, the procedure for the condition is called, the result is assigned to the `flag` register, and the `pc` is advanced:

```
(define (make-test inst machine labels operations flag pc)
  (let ((condition (test-condition inst)))
    (if (operation-exp? condition)
        (let ((condition-proc
               (make-operation-exp
                condition machine labels operations)))
          (lambda ()
            (set-contents! flag (condition-proc))
            (advance-pc pc)))
        (error "Bad TEST instruction -- ASSEMBLE" inst))))

(define (test-condition test-instruction)
  (cdr test-instruction))
```

The execution procedure for a `branch` instruction checks the contents of the `flag` register and either sets the contents of the `pc` to the branch destination (if the branch is taken) or else just advances the `pc` (if the branch is not taken). Notice that the indicated destination in a `branch` instruction must be a label, and the `make-branch` procedure enforces this. Notice also that the label is looked up at assembly time, not each time the `branch` instruction is simulated.

```
(define (make-branch inst machine labels flag pc)
  (let ((dest (branch-dest inst)))
    (if (label-exp? dest)
        (let ((insts
               (lookup-label labels (label-exp-label dest))))
          (lambda ()
            (if (get-contents flag)
                (set-contents! pc insts)
                (advance-pc pc))))
        (error "Bad BRANCH instruction -- ASSEMBLE" inst))))

(define (branch-dest branch-instruction)
  (cadr branch-instruction))
```

A `goto` instruction is similar to a branch, except that the destination may be specified either as a label or as a register, and there is no condition to check—the `pc` is always set to the new destination.

```
(define (make-goto inst machine labels pc)
  (let ((dest (goto-dest inst)))
    (cond ((label-exp? dest)
           (let ((insts
                  (lookup-label labels
                                (label-exp-label dest))))
             (lambda () (set-contents! pc insts))))
          ((register-exp? dest)
           (let ((reg
                  (get-register machine
                                (register-exp-reg dest))))
             (lambda ()
               (set-contents! pc (get-contents reg)))))
          (else (error "Bad GOTO instruction -- ASSEMBLE"
                       inst)))))

(define (goto-dest goto-instruction)
  (cadr goto-instruction))
```

Other instructions

The stack instructions `save` and `restore` simply use the stack with the designated register and advance the `pc`:

```
(define (make-save inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (push stack (get-contents reg))
      (advance-pc pc))))
```

```
(define (make-restore inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (set-contents! reg (pop stack))
      (advance-pc pc))))

(define (stack-inst-reg-name stack-instruction)
  (cadr stack-instruction))
```

The final instruction type, handled by `make-perform`, generates an execution procedure for the action to be performed. At simulation time, the action procedure is executed and the pc advanced.

```
(define (make-perform inst machine labels operations pc)
  (let ((action (perform-action inst)))
    (if (operation-exp? action)
        (let ((action-proc
               (make-operation-exp
                action machine labels operations)))
          (lambda ()
            (action-proc)
            (advance-pc pc)))
        (error "Bad PERFORM instruction -- ASSEMBLE" inst))))

(define (perform-action inst) (cdr inst))
```

Execution procedures for subexpressions

The value of a `reg`, `label`, or `const` expression may be needed for assignment to a register (`make-assign`) or for input to an operation (`make-operation-exp`, below). The following procedure generates execution procedures to produce values for these expressions during the simulation:

```
(define (make-primitive-exp exp machine labels)
  (cond ((constant-exp? exp)
         (let ((c (constant-exp-value exp)))
           (lambda () c)))
        ((label-exp? exp)
         (let ((insts
                (lookup-label labels
                              (label-exp-label exp))))
           (lambda () insts)))
        ((register-exp? exp)
         (let ((r (get-register machine
                                (register-exp-reg exp))))
           (lambda () (get-contents r))))
        (else
         (error "Unknown expression type -- ASSEMBLE" exp))))
```

The syntax of `reg`, `label`, and `const` expressions is determined by

```
(define (register-exp? exp) (tagged-list? exp 'reg))

(define (register-exp-reg exp) (cadr exp))

(define (constant-exp? exp) (tagged-list? exp 'const))

(define (constant-exp-value exp) (cadr exp))

(define (label-exp? exp) (tagged-list? exp 'label))

(define (label-exp-label exp) (cadr exp))
```

`Assign`, `perform`, and `test` instructions may include the application of a machine operation (specified by an `op` expression) to some operands (specified by `reg` and `const` expressions). The following procedure produces an execution procedure for an "operation expression"—a list containing the operation and operand expressions from the instruction:

```
(define (make-operation-exp exp machine labels operations)
  (let ((op (lookup-prim (operation-exp-op exp) operations))
        (aprocs
         (map (lambda (e)
                (make-primitive-exp e machine labels))
              (operation-exp-operands exp))))
    (lambda ()
      (apply op (map (lambda (p) (p)) aprocs)))))
```

The syntax of operation expressions is determined by

```
(define (operation-exp? exp)
  (and (pair? exp) (tagged-list? (car exp) 'op)))

(define (operation-exp-op operation-exp)
  (cadr (car operation-exp)))

(define (operation-exp-operands operation-exp)
  (cdr operation-exp))
```

Observe that the treatment of operation expressions is very much like the treatment of procedure applications by the `analyze-application` procedure in the evaluator of section 4.1.7 in that we generate an execution procedure for each operand. At simulation time, we call the operand

procedures and apply the Scheme procedure that simulates the operation to the resulting values. The simulation procedure is found by looking up the operation name in the operation table for the machine:

```
(define (lookup-prim symbol operations)
  (let ((val (assoc symbol operations)))
    (if val
        (cadr val)
        (error "Unknown operation -- ASSEMBLE" symbol))))
```

Exercise 5.9

The treatment of machine operations above permits them to operate on labels as well as on constants and the contents of registers. Modify the expression-processing procedures to enforce the condition that operations can be used only with registers and constants.

Exercise 5.10

Design a new syntax for register-machine instructions and modify the simulator to use your new syntax. Can you implement your new syntax without changing any part of the simulator except the syntax procedures in this section?

Exercise 5.11

When we introduced `save` and `restore` in section 5.1.4, we didn't specify what would happen if you tried to restore a register that was not the last one saved, as in the sequence

```
(save y)
(save x)
(restore y)
```

There are several reasonable possibilities for the meaning of `restore`:

a. `(restore y)` puts into y the last value saved on the stack, regardless of what register that value came from. This is the way our simulator behaves. Show how to take advantage of this behavior to eliminate one instruction from the Fibonacci machine of section 5.1.4 (figure 5.12).

b. `(restore y)` puts into y the last value saved on the stack, but only if that value was saved from y; otherwise, it signals an error. Modify the simulator to behave this way. You will have to change `save` to put the register name on the stack along with the value.

c. `(restore y)` puts into y the last value saved from y regardless of what other registers were saved after y and not restored. Modify the simulator to behave this way. You will have to associate a separate stack with each register. You should make the `initialize-stack` operation initialize all the register stacks.

Exercise 5.12

The simulator can be used to help determine the data paths required for implementing a machine with a given controller. Extend the assembler to store the following information in the machine model:

- a list of all instructions, with duplicates removed, sorted by instruction type (`assign`, `goto`, and so on);
- a list (without duplicates) of the registers used to hold entry points (these are the registers referenced by `goto` instructions);
- a list (without duplicates) of the registers that are `saved` or `restored`;
- for each register, a list (without duplicates) of the sources from which it is assigned (for example, the sources for register `val` in the factorial machine of figure 5.11 are `(const 1)` and `((op *) (reg n) (reg val))`).

Extend the message-passing interface to the machine to provide access to this new information. To test your analyzer, define the Fibonacci machine from figure 5.12 and examine the lists you constructed.

Exercise 5.13

Modify the simulator so that it uses the controller sequence to determine what registers the machine has rather than requiring a list of registers as an argument to `make-machine`. Instead of pre-allocating the registers in `make-machine`, you can allocate them one at a time when they are first seen during assembly of the instructions.

5.2.4 Monitoring Machine Performance

Simulation is useful not only for verifying the correctness of a proposed machine design but also for measuring the machine's performance. For example, we can install in our simulation program a "meter" that measures the number of stack operations used in a computation. To do this, we modify our simulated stack to keep track of the number of times registers are saved on the stack and the maximum depth reached by the stack, and add a message to the stack's interface that prints the statistics, as shown below. We also add an operation to the basic machine model to print the stack statistics, by initializing `the-ops` in `make-new-machine` to

```
(list (list 'initialize-stack
            (lambda () (stack 'initialize)))
      (list 'print-stack-statistics
            (lambda () (stack 'print-statistics))))
```

Here is the new version of `make-stack`:

```
(define (make-stack)
  (let ((s '())
        (number-pushes 0)
        (max-depth 0)
        (current-depth 0))
    (define (push x)
      (set! s (cons x s))
      (set! number-pushes (+ 1 number-pushes))
      (set! current-depth (+ 1 current-depth))
      (set! max-depth (max current-depth max-depth)))
    (define (pop)
      (if (null? s)
          (error "Empty stack -- POP")
          (let ((top (car s)))
            (set! s (cdr s))
            (set! current-depth (- current-depth 1))
            top)))
    (define (initialize)
      (set! s '())
      (set! number-pushes 0)
      (set! max-depth 0)
      (set! current-depth 0)
      'done)
    (define (print-statistics)
      (newline)
      (display (list 'total-pushes  '= number-pushes
                     'maximum-depth '= max-depth)))
    (define (dispatch message)
      (cond ((eq? message 'push) push)
            ((eq? message 'pop) (pop))
            ((eq? message 'initialize) (initialize))
            ((eq? message 'print-statistics)
             (print-statistics))
            (else
             (error "Unknown request -- STACK" message))))
    dispatch))
```

Exercises 5.15 through 5.19 describe other useful monitoring and debugging features that can be added to the register-machine simulator.

Exercise 5.14

Measure the number of pushes and the maximum stack depth required to compute $n!$ for various small values of n using the factorial machine shown in figure 5.11. From your data determine formulas in terms of n for the total number of push operations and the maximum stack depth used in computing $n!$ for any $n > 1$. Note that each of these is a linear function of n and is thus determined by two constants. In order to get the statistics printed, you will have to augment the factorial machine with instructions to initialize the stack and print the statistics. You may want to also modify the machine so that it repeatedly reads a value for n, computes the factorial, and prints the result (as we did for the GCD machine in figure 5.4), so that you will not have to repeatedly invoke `get-register-contents`, `set-register-contents!`, and `start`.

Exercise 5.15

Add *instruction counting* to the register machine simulation. That is, have the machine model keep track of the number of instructions executed. Extend the machine model's interface to accept a new message that prints the value of the instruction count and resets the count to zero.

Exercise 5.16

Augment the simulator to provide for *instruction tracing*. That is, before each instruction is executed, the simulator should print the text of the instruction. Make the machine model accept `trace-on` and `trace-off` messages to turn tracing on and off.

Exercise 5.17

Extend the instruction tracing of exercise 5.16 so that before printing an instruction, the simulator prints any labels that immediately precede that instruction in the controller sequence. Be careful to do this in a way that does not interfere with instruction counting (exercise 5.15). You will have to make the simulator retain the necessary label information.

Exercise 5.18

Modify the `make-register` procedure of section 5.2.1 so that registers can be traced. Registers should accept messages that turn tracing on and off. When a register is traced, assigning a value to the register should print the name of the register, the old contents of the register, and the new contents being assigned. Extend the interface to the machine model to permit you to turn tracing on and off for designated machine registers.

Exercise 5.19

Alyssa P. Hacker wants a *breakpoint* feature in the simulator to help her debug her machine designs. You have been hired to install this feature for her. She wants to be able to specify a place in the controller sequence where the simulator will stop and allow her to examine the state of the machine. You are to implement a procedure

`(set-breakpoint` ⟨*machine*⟩ ⟨*label*⟩ ⟨*n*⟩`)`

that sets a breakpoint just before the *n*th instruction after the given label. For example,

```
(set-breakpoint gcd-machine 'test-b 4)
```

installs a breakpoint in `gcd-machine` just before the assignment to register `a`. When the simulator reaches the breakpoint it should print the label and the offset of the breakpoint and stop executing instructions. Alyssa can then use `get-register-contents` and `set-register-contents!` to manipulate the state of the simulated machine. She should then be able to continue execution by saying

`(proceed-machine` ⟨*machine*⟩`)`

She should also be able to remove a specific breakpoint by means of

`(cancel-breakpoint` ⟨*machine*⟩ ⟨*label*⟩ ⟨*n*⟩`)`

or to remove all breakpoints by means of

`(cancel-all-breakpoints` ⟨*machine*⟩`)`

5.3 Storage Allocation and Garbage Collection

In section 5.4, we will show how to implement a Scheme evaluator as a register machine. In order to simplify the discussion, we will assume that our register machines can be equipped with a *list-structured memory*, in which the basic operations for manipulating list-structured data are primitive. Postulating the existence of such a memory is a useful abstraction when one is focusing on the mechanisms of control in a Scheme interpreter, but this does not reflect a realistic view of the actual primitive data operations of contemporary computers. To obtain a more complete picture of how a Lisp system operates, we must investigate how list structure can be represented in a way that is compatible with conventional computer memories.

There are two considerations in implementing list structure. The first is purely an issue of representation: how to represent the "box-and-pointer" structure of Lisp pairs, using only the storage and addressing capabilities of typical computer memories. The second issue concerns the management of memory as a computation proceeds. The operation of a Lisp system depends crucially on the ability to continually create new data objects. These include objects that are explicitly created by the Lisp procedures being interpreted as well as structures created by the in-

terpreter itself, such as environments and argument lists. Although the constant creation of new data objects would pose no problem on a computer with an infinite amount of rapidly addressable memory, computer memories are available only in finite sizes (more's the pity). Lisp systems thus provide an *automatic storage allocation* facility to support the illusion of an infinite memory. When a data object is no longer needed, the memory allocated to it is automatically recycled and used to construct new data objects. There are various techniques for providing such automatic storage allocation. The method we shall discuss in this section is called *garbage collection.*

5.3.1 Memory as Vectors

A conventional computer memory can be thought of as an array of cubbyholes, each of which can contain a piece of information. Each cubbyhole has a unique name, called its *address* or *location.* Typical memory systems provide two primitive operations: one that fetches the data stored in a specified location and one that assigns new data to a specified location. Memory addresses can be incremented to support sequential access to some set of the cubbyholes. More generally, many important data operations require that memory addresses be treated as data, which can be stored in memory locations and manipulated in machine registers. The representation of list structure is one application of such *address arithmetic.*

To model computer memory, we use a new kind of data structure called a *vector.* Abstractly, a vector is a compound data object whose individual elements can be accessed by means of an integer index in an amount of time that is independent of the index.[5] In order to describe memory operations, we use two primitive Scheme procedures for manipulating vectors:

- `(vector-ref` ⟨*vector*⟩ ⟨*n*⟩`)` returns the *n*th element of the vector.
- `(vector-set!` ⟨*vector*⟩ ⟨*n*⟩ ⟨*value*⟩`)` sets the *n*th element of the vector to the designated value.

For example, if `v` is a vector, then `(vector-ref v 5)` gets the fifth entry in the vector `v` and `(vector-set! v 5 7)` changes the value of the fifth

[5]We could represent memory as lists of items. However, the access time would then not be independent of the index, since accessing the *n*th element of a list requires $n - 1$ `cdr` operations.

entry of the vector v to 7.[6] For computer memory, this access can be implemented through the use of address arithmetic to combine a *base address* that specifies the beginning location of a vector in memory with an *index* that specifies the offset of a particular element of the vector.

Representing Lisp data

We can use vectors to implement the basic pair structures required for a list-structured memory. Let us imagine that computer memory is divided into two vectors: `the-cars` and `the-cdrs`. We will represent list structure as follows: A pointer to a pair is an index into the two vectors. The `car` of the pair is the entry in `the-cars` with the designated index, and the `cdr` of the pair is the entry in `the-cdrs` with the designated index. We also need a representation for objects other than pairs (such as numbers and symbols) and a way to distinguish one kind of data from another. There are many methods of accomplishing this, but they all reduce to using *typed pointers*, that is, to extending the notion of "pointer" to include information on data type.[7] The data type enables the system to distinguish a pointer to a pair (which consists of the "pair" data type and an index into the memory vectors) from pointers to other kinds of data (which consist of some other data type and whatever is being used to represent data of that type). Two data objects are considered to be the same (`eq?`) if their pointers are identical.[8] Figure 5.14 illustrates the use of this method to represent the list `((1 2) 3 4)`, whose box-and-pointer diagram is also shown. We use letter prefixes to denote the data-type information. Thus, a pointer to the pair with index 5 is denoted `p5`, the empty list is denoted by the pointer `e0`, and a pointer to the number 4

[6]For completeness, we should specify a `make-vector` operation that constructs vectors. However, in the present application we will use vectors only to model fixed divisions of the computer memory.

[7]This is precisely the same "tagged data" idea we introduced in chapter 2 for dealing with generic operations. Here, however, the data types are included at the primitive machine level rather than constructed through the use of lists.

[8]Type information may be encoded in a variety of ways, depending on the details of the machine on which the Lisp system is to be implemented. The execution efficiency of Lisp programs will be strongly dependent on how cleverly this choice is made, but it is difficult to formulate general design rules for good choices. The most straightforward way to implement typed pointers is to allocate a fixed set of bits in each pointer to be a *type field* that encodes the data type. Important questions to be addressed in designing such a representation include the following: How many type bits are required? How large must the vector indices be? How efficiently can the primitive machine instructions be used to manipulate the type fields of pointers? Machines that include special hardware for the efficient handling of type fields are said to have *tagged architectures.*

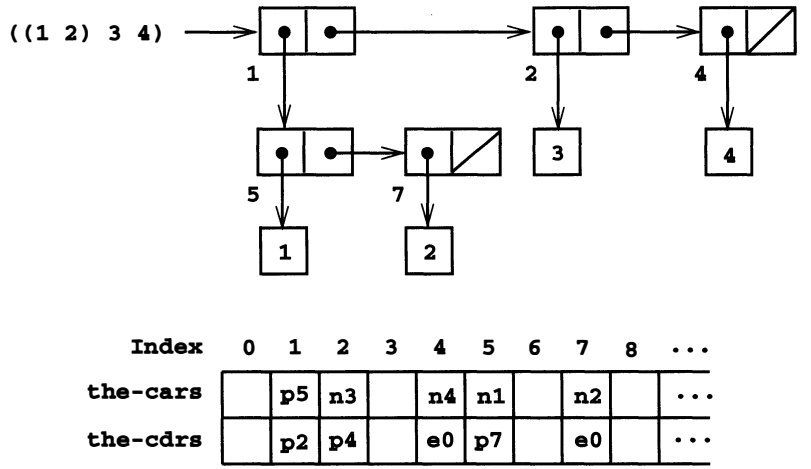


Figure 5.14 Box-and-pointer and memory-vector representations of the list `((1 2) 3 4)`.

is denoted `n4`. In the box-and-pointer diagram, we have indicated at the lower left of each pair the vector index that specifies where the `car` and `cdr` of the pair are stored. The blank locations in `the-cars` and `the-cdrs` may contain parts of other list structures (not of interest here).

A pointer to a number, such as `n4`, might consist of a type indicating numeric data together with the actual representation of the number 4.[9] To deal with numbers that are too large to be represented in the fixed amount of space allocated for a single pointer, we could use a distinct *bignum* data type, for which the pointer designates a list in which the parts of the number are stored.[10]

A symbol might be represented as a typed pointer that designates a sequence of the characters that form the symbol's printed representation. This sequence is constructed by the Lisp reader when the character string is initially encountered in input. Since we want two instances of a symbol to be recognized as the "same" symbol by `eq?` and we want `eq?` to be a simple test for equality of pointers, we must ensure that if the reader

[9]This decision on the representation of numbers determines whether `eq?`, which tests equality of pointers, can be used to test for equality of numbers. If the pointer contains the number itself, then equal numbers will have the same pointer. But if the pointer contains the index of a location where the number is stored, equal numbers will be guaranteed to have equal pointers only if we are careful never to store the same number in more than one location.

[10]This is just like writing a number as a sequence of digits, except that each "digit" is a number between 0 and the largest number that can be stored in a single pointer.

sees the same character string twice, it will use the same pointer (to the same sequence of characters) to represent both occurrences. To accomplish this, the reader maintains a table, traditionally called the *obarray*, of all the symbols it has ever encountered. When the reader encounters a character string and is about to construct a symbol, it checks the obarray to see if it has ever before seen the same character string. If it has not, it uses the characters to construct a new symbol (a typed pointer to a new character sequence) and enters this pointer in the obarray. If the reader has seen the string before, it returns the symbol pointer stored in the obarray. This process of replacing character strings by unique pointers is called *interning* symbols.

Implementing the primitive list operations

Given the above representation scheme, we can replace each "primitive" list operation of a register machine with one or more primitive vector operations. We will use two registers, `the-cars` and `the-cdrs`, to identify the memory vectors, and will assume that `vector-ref` and `vector-set!` are available as primitive operations. We also assume that numeric operations on pointers (such as incrementing a pointer, using a pair pointer to index a vector, or adding two numbers) use only the index portion of the typed pointer.

For example, we can make a register machine support the instructions

`(assign` ⟨reg_1⟩ `(op car) (reg` ⟨reg_2⟩`))`

`(assign` ⟨reg_1⟩ `(op cdr) (reg` ⟨reg_2⟩`))`

if we implement these, respectively, as

`(assign` ⟨reg_1⟩ `(op vector-ref) (reg the-cars) (reg` ⟨reg_2⟩`))`

`(assign` ⟨reg_1⟩ `(op vector-ref) (reg the-cdrs) (reg` ⟨reg_2⟩`))`

The instructions

`(perform (op set-car!) (reg` ⟨reg_1⟩`) (reg` ⟨reg_2⟩`))`

`(perform (op set-cdr!) (reg` ⟨reg_1⟩`) (reg` ⟨reg_2⟩`))`

are implemented as

```
(perform
 (op vector-set!) (reg the-cars) (reg ⟨reg1⟩) (reg ⟨reg2⟩))

(perform
 (op vector-set!) (reg the-cdrs) (reg ⟨reg1⟩) (reg ⟨reg2⟩))
```

`Cons` is performed by allocating an unused index and storing the arguments to `cons` in `the-cars` and `the-cdrs` at that indexed vector position. We presume that there is a special register, `free`, that always holds a pair pointer containing the next available index, and that we can increment the index part of that pointer to find the next free location.[11] For example, the instruction

```
(assign ⟨reg1⟩ (op cons) (reg ⟨reg2⟩) (reg ⟨reg3⟩))
```

is implemented as the following sequence of vector operations:[12]

```
(perform
 (op vector-set!) (reg the-cars) (reg free) (reg ⟨reg2⟩))
(perform
 (op vector-set!) (reg the-cdrs) (reg free) (reg ⟨reg3⟩))
(assign ⟨reg1⟩ (reg free))
(assign free (op +) (reg free) (const 1))
```

The `eq?` operation

```
(op eq?) (reg ⟨reg1⟩) (reg ⟨reg2⟩)
```

simply tests the equality of all fields in the registers, and predicates such as `pair?`, `null?`, `symbol?`, and `number?` need only check the type field.

Implementing stacks

Although our register machines use stacks, we need do nothing special here, since stacks can be modeled in terms of lists. The stack can be a list of the saved values, pointed to by a special register `the-stack`. Thus, `(save ⟨reg⟩)` can be implemented as

```
(assign the-stack (op cons) (reg ⟨reg⟩) (reg the-stack))
```

Similarly, `(restore ⟨reg⟩)` can be implemented as

```
(assign ⟨reg⟩ (op car) (reg the-stack))
(assign the-stack (op cdr) (reg the-stack))
```

---

[11] There are other ways of finding free storage. For example, we could link together all the unused pairs into a *free list*. Our free locations are consecutive (and hence can be accessed by incrementing a pointer) because we are using a compacting garbage collector, as we will see in section 5.3.2.

[12] This is essentially the implementation of `cons` in terms of `set-car!` and `set-cdr!`, as described in section 3.3.1. The operation `get-new-pair` used in that implementation is realized here by the `free` pointer.

and `(perform (op initialize-stack))` can be implemented as

```
(assign the-stack (const ()))
```

These operations can be further expanded in terms of the vector operations given above. In conventional computer architectures, however, it is usually advantageous to allocate the stack as a separate vector. Then pushing and popping the stack can be accomplished by incrementing or decrementing an index into that vector.

Exercise 5.20

Draw the box-and-pointer representation and the memory-vector representation (as in figure 5.14) of the list structure produced by

```
(define x (cons 1 2))
(define y (list x x))
```

with the `free` pointer initially `p1`. What is the final value of `free`? What pointers represent the values of `x` and `y`?

Exercise 5.21

Implement register machines for the following procedures. Assume that the list-structure memory operations are available as machine primitives.

a. Recursive `count-leaves`:

```
(define (count-leaves tree)
  (cond ((null? tree) 0)
        ((not (pair? tree)) 1)
        (else (+ (count-leaves (car tree))
                 (count-leaves (cdr tree))))))
```

b. Recursive `count-leaves` with explicit counter:

```
(define (count-leaves tree)
  (define (count-iter tree n)
    (cond ((null? tree) n)
          ((not (pair? tree)) (+ n 1))
          (else (count-iter (cdr tree)
                            (count-iter (car tree) n)))))
  (count-iter tree 0))
```

Exercise 5.22

Exercise 3.12 of section 3.3.1 presented an `append` procedure that appends two lists to form a new list and an `append!` procedure that splices two lists together. Design a register machine to implement each of these procedures. Assume that the list-structure memory operations are available as primitive operations.

5.3.2 Maintaining the Illusion of Infinite Memory

The representation method outlined in section 5.3.1 solves the problem of implementing list structure, provided that we have an infinite amount of memory. With a real computer we will eventually run out of free space in which to construct new pairs.[13] However, most of the pairs generated in a typical computation are used only to hold intermediate results. After these results are accessed, the pairs are no longer needed—they are *garbage.* For instance, the computation

```
(accumulate + 0 (filter odd? (enumerate-interval 0 n)))
```

constructs two lists: the enumeration and the result of filtering the enumeration. When the accumulation is complete, these lists are no longer needed, and the allocated memory can be reclaimed. If we can arrange to collect all the garbage periodically, and if this turns out to recycle memory at about the same rate at which we construct new pairs, we will have preserved the illusion that there is an infinite amount of memory.

In order to recycle pairs, we must have a way to determine which allocated pairs are not needed (in the sense that their contents can no longer influence the future of the computation). The method we shall examine for accomplishing this is known as *garbage collection.* Garbage collection is based on the observation that, at any moment in a Lisp interpretation, the only objects that can affect the future of the computation are those that can be reached by some succession of `car` and `cdr` operations starting from the pointers that are currently in the machine registers.[14] Any memory cell that is not so accessible may be recycled.

There are many ways to perform garbage collection. The method we shall examine here is called *stop-and-copy.* The basic idea is to divide memory into two halves: "working memory" and "free memory." When `cons` constructs pairs, it allocates these in working memory. When work-

[13]This may not be true eventually, because memories may get large enough so that it would be impossible to run out of free memory in the lifetime of the computer. For example, there are about 3×10^{13}, microseconds in a year, so if we were to `cons` once per microsecond we would need about 10^{15} cells of memory to build a machine that could operate for 30 years without running out of memory. That much memory seems absurdly large by today's standards, but it is not physically impossible. On the other hand, processors are getting faster and a future computer may have large numbers of processors operating in parallel on a single memory, so it may be possible to use up memory much faster than we have postulated.

[14]We assume here that the stack is represented as a list as described in section 5.3.1, so that items on the stack are accessible via the pointer in the stack register.

ing memory is full, we perform garbage collection by locating all the useful pairs in working memory and copying these into consecutive locations in free memory. (The useful pairs are located by tracing all the `car` and `cdr` pointers, starting with the machine registers.) Since we do not copy the garbage, there will presumably be additional free memory that we can use to allocate new pairs. In addition, nothing in the working memory is needed, since all the useful pairs in it have been copied. Thus, if we interchange the roles of working memory and free memory, we can continue processing; new pairs will be allocated in the new working memory (which was the old free memory). When this is full, we can copy the useful pairs into the new free memory (which was the old working memory).[15]

Implementation of a stop-and-copy garbage collector

We now use our register-machine language to describe the stop-and-copy algorithm in more detail. We will assume that there is a register called `root` that contains a pointer to a structure that eventually points at all accessible data. This can be arranged by storing the contents of all the machine registers in a pre-allocated list pointed at by `root` just before starting garbage collection.[16] We also assume that, in addition to the current working memory, there is free memory available into which we can

[15]This idea was invented and first implemented by Minsky, as part of the implementation of Lisp for the PDP-1 at the MIT Research Laboratory of Electronics. It was further developed by Fenichel and Yochelson (1969) for use in the Lisp implementation for the Multics time-sharing system. Later, Baker (1978) developed a "real-time" version of the method, which does not require the computation to stop during garbage collection. Baker's idea was extended by Hewitt, Lieberman, and Moon (see Lieberman and Hewitt 1983) to take advantage of the fact that some structure is more volatile and other structure is more permanent.

An alternative commonly used garbage-collection technique is the *mark-sweep* method. This consists of tracing all the structure accessible from the machine registers and marking each pair we reach. We then scan all of memory, and any location that is unmarked is "swept up" as garbage and made available for reuse. A full discussion of the mark-sweep method can be found in Allen 1978.

The Minsky-Fenichel-Yochelson algorithm is the dominant algorithm in use for large-memory systems because it examines only the useful part of memory. This is in contrast to mark-sweep, in which the sweep phase must check all of memory. A second advantage of stop-and-copy is that it is a *compacting* garbage collector. That is, at the end of the garbage-collection phase the useful data will have been moved to consecutive memory locations, with all garbage pairs compressed out. This can be an extremely important performance consideration in machines with virtual memory, in which accesses to widely separated memory addresses may require extra paging operations.

[16]This list of registers does not include the registers used by the storage-allocation system—`root`, `the-cars`, `the-cdrs`, and the other registers that will be introduced in this section.

copy the useful data. The current working memory consists of vectors whose base addresses are in registers called `the-cars` and `the-cdrs`, and the free memory is in registers called `new-cars` and `new-cdrs`.

Garbage collection is triggered when we exhaust the free cells in the current working memory, that is, when a `cons` operation attempts to increment the `free` pointer beyond the end of the memory vector. When the garbage-collection process is complete, the `root` pointer will point into the new memory, all objects accessible from the `root` will have been moved to the new memory, and the `free` pointer will indicate the next place in the new memory where a new pair can be allocated. In addition, the roles of working memory and new memory will have been interchanged—new pairs will be constructed in the new memory, beginning at the place indicated by `free`, and the (previous) working memory will be available as the new memory for the next garbage collection. Figure 5.15 shows the arrangement of memory just before and just after garbage collection.

The state of the garbage-collection process is controlled by maintaining two pointers: `free` and `scan`. These are initialized to point to the beginning of the new memory. The algorithm begins by relocating the pair pointed at by `root` to the beginning of the new memory. The pair is copied, the `root` pointer is adjusted to point to the new location, and the `free` pointer is incremented. In addition, the old location of the pair is marked to show that its contents have been moved. This marking is done as follows: In the `car` position, we place a special tag that signals that this is an already-moved object. (Such an object is traditionally called a *broken heart.*)[17] In the `cdr` position we place a *forwarding address* that points at the location to which the object has been moved.

After relocating the root, the garbage collector enters its basic cycle. At each step in the algorithm, the `scan` pointer (initially pointing at the relocated root) points at a pair that has been moved to the new memory but whose `car` and `cdr` pointers still refer to objects in the old memory. These objects are each relocated, and the `scan` pointer is incremented. To relocate an object (for example, the object indicated by the `car` pointer of the pair we are scanning) we check to see if the object has already been moved (as indicated by the presence of a broken-heart tag in the `car` position of the object). If the object has not already been moved, we copy it to the place indicated by `free`, update `free`, set up a broken

[17]The term *broken heart* was coined by David Cressey, who wrote a garbage collector for MDL, a dialect of Lisp developed at MIT during the early 1970s.

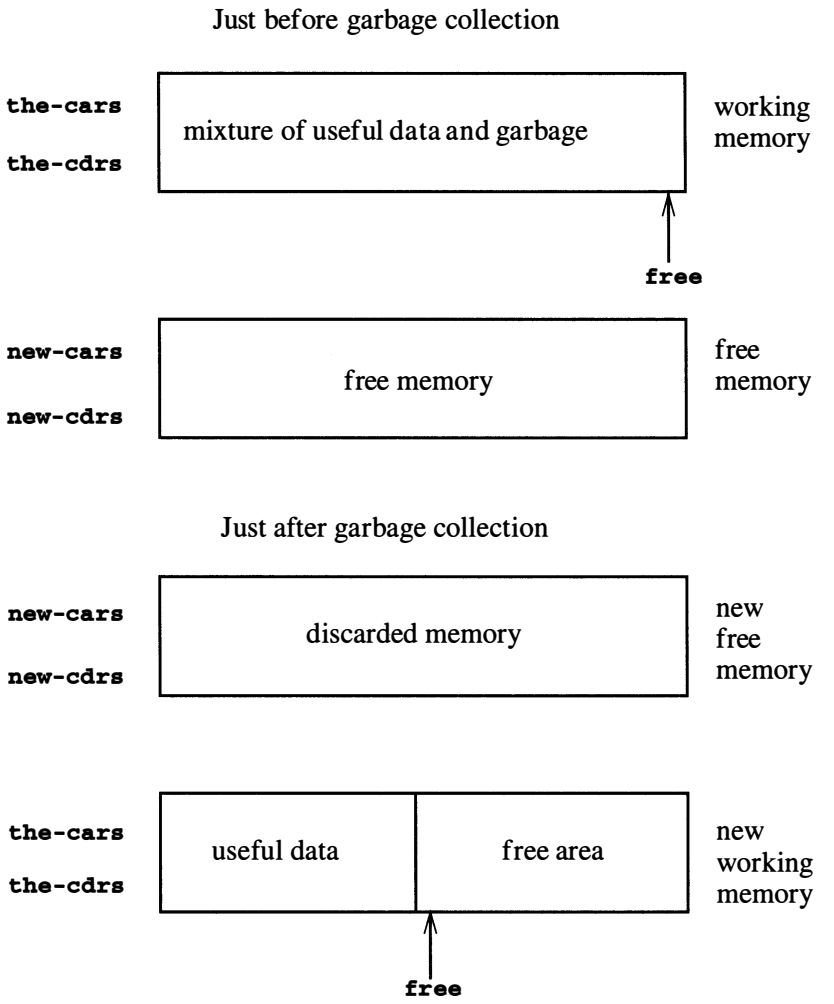


Figure 5.15 Reconfiguration of memory by the garbage-collection process.

heart at the object's old location, and update the pointer to the object (in this example, the `car` pointer of the pair we are scanning) to point to the new location. If the object has already been moved, its forwarding address (found in the `cdr` position of the broken heart) is substituted for the pointer in the pair being scanned. Eventually, all accessible objects will have been moved and scanned, at which point the `scan` pointer will overtake the `free` pointer and the process will terminate.

We can specify the stop-and-copy algorithm as a sequence of instructions for a register machine. The basic step of relocating an object is accomplished by a subroutine called `relocate-old-result-in-new`. This subroutine gets its argument, a pointer to the object to be relocated, from a register named `old`. It relocates the designated object (incrementing `free` in the process), puts a pointer to the relocated object into a register called `new`, and returns by branching to the entry point stored in the register `relocate-continue`. To begin garbage collection, we invoke this subroutine to relocate the `root` pointer, after initializing `free` and `scan`. When the relocation of `root` has been accomplished, we install the new pointer as the new `root` and enter the main loop of the garbage collector.

```
begin-garbage-collection
  (assign free (const 0))
  (assign scan (const 0))
  (assign old (reg root))
  (assign relocate-continue (label reassign-root))
  (goto (label relocate-old-result-in-new))
reassign-root
  (assign root (reg new))
  (goto (label gc-loop))
```

In the main loop of the garbage collector we must determine whether there are any more objects to be scanned. We do this by testing whether the `scan` pointer is coincident with the `free` pointer. If the pointers are equal, then all accessible objects have been relocated, and we branch to `gc-flip`, which cleans things up so that we can continue the interrupted computation. If there are still pairs to be scanned, we call the relocate subroutine to relocate the `car` of the next pair (by placing the `car` pointer in `old`). The `relocate-continue` register is set up so that the subroutine will return to update the `car` pointer.

```
gc-loop
  (test (op =) (reg scan) (reg free))
  (branch (label gc-flip))
  (assign old (op vector-ref) (reg new-cars) (reg scan))
  (assign relocate-continue (label update-car))
  (goto (label relocate-old-result-in-new))
```

At `update-car`, we modify the `car` pointer of the pair being scanned, then proceed to relocate the `cdr` of the pair. We return to `update-cdr` when that relocation has been accomplished. After relocating and updating the `cdr`, we are finished scanning that pair, so we continue with the main loop.

```
update-car
  (perform
   (op vector-set!) (reg new-cars) (reg scan) (reg new))
  (assign old (op vector-ref) (reg new-cdrs) (reg scan))
  (assign relocate-continue (label update-cdr))
  (goto (label relocate-old-result-in-new))

update-cdr
  (perform
   (op vector-set!) (reg new-cdrs) (reg scan) (reg new))
  (assign scan (op +) (reg scan) (const 1))
  (goto (label gc-loop))
```

The subroutine `relocate-old-result-in-new` relocates objects as follows: If the object to be relocated (pointed at by `old`) is not a pair, then we return the same pointer to the object unchanged (in `new`). (For example, we may be scanning a pair whose `car` is the number 4. If we represent the `car` by `n4`, as described in section 5.3.1, then we want the "relocated" `car` pointer to still be `n4`.) Otherwise, we must perform the relocation. If the `car` position of the pair to be relocated contains a broken-heart tag, then the pair has in fact already been moved, so we retrieve the forwarding address (from the `cdr` position of the broken heart) and return this in `new`. If the pointer in `old` points at a yet-unmoved pair, then we move the pair to the first free cell in new memory (pointed at by `free`) and set up the broken heart by storing a broken-heart tag and forwarding address at the old location. `Relocate-old-result-in-new` uses a register `oldcr` to hold the `car` or the `cdr` of the object pointed at by `old`.[18]

[18]The garbage collector uses the low-level predicate `pointer-to-pair?` instead of the list-structure `pair?` operation because in a real system there might be various things that are treated as pairs for garbage-collection purposes. For example, in a Scheme system that conforms to the IEEE standard a procedure object may be implemented as a special kind of "pair" that doesn't satisfy the `pair?` predicate. For simulation purposes, `pointer-to-pair?` can be implemented as `pair?`.

```
relocate-old-result-in-new
  (test (op pointer-to-pair?) (reg old))
  (branch (label pair))
  (assign new (reg old))
  (goto (reg relocate-continue))
pair
  (assign oldcr (op vector-ref) (reg the-cars) (reg old))
  (test (op broken-heart?) (reg oldcr))
  (branch (label already-moved))
  (assign new (reg free))  ; new location for pair
  ;; Update free pointer.
  (assign free (op +) (reg free) (const 1))
  ;; Copy the car and cdr to new memory.
  (perform (op vector-set!)
           (reg new-cars) (reg new) (reg oldcr))
  (assign oldcr (op vector-ref) (reg the-cdrs) (reg old))
  (perform (op vector-set!)
           (reg new-cdrs) (reg new) (reg oldcr))
  ;; Construct the broken heart.
  (perform (op vector-set!)
           (reg the-cars) (reg old) (const broken-heart))
  (perform
   (op vector-set!) (reg the-cdrs) (reg old) (reg new))
  (goto (reg relocate-continue))
already-moved
  (assign new (op vector-ref) (reg the-cdrs) (reg old))
  (goto (reg relocate-continue))
```

At the very end of the garbage-collection process, we interchange the role of old and new memories by interchanging pointers: interchanging `the-cars` with `new-cars`, and `the-cdrs` with `new-cdrs`. We will then be ready to perform another garbage collection the next time memory runs out.

```
gc-flip
  (assign temp (reg the-cdrs))
  (assign the-cdrs (reg new-cdrs))
  (assign new-cdrs (reg temp))
  (assign temp (reg the-cars))
  (assign the-cars (reg new-cars))
  (assign new-cars (reg temp))
```

5.4 The Explicit-Control Evaluator

In section 5.1 we saw how to transform simple Scheme programs into descriptions of register machines. We will now perform this transformation on a more complex program, the metacircular evaluator of sections 4.1.1–4.1.4, which shows how the behavior of a Scheme interpreter can be described in terms of the procedures `eval` and `apply`. The *explicit-control evaluator* that we develop in this section shows how the underlying procedure-calling and argument-passing mechanisms used in the evaluation process can be described in terms of operations on registers and stacks. In addition, the explicit-control evaluator can serve as an implementation of a Scheme interpreter, written in a language that is very similar to the native machine language of conventional computers. The evaluator can be executed by the register-machine simulator of section 5.2. Alternatively, it can be used as a starting point for building a machine-language implementation of a Scheme evaluator, or even a special-purpose machine for evaluating Scheme expressions. Figure 5.16 shows such a hardware implementation: a silicon chip that acts as an evaluator for Scheme. The chip designers started with the data-path and controller specifications for a register machine similar to the evaluator described in this section and used design automation programs to construct the integrated-circuit layout.[19]

Registers and operations

In designing the explicit-control evaluator, we must specify the operations to be used in our register machine. We described the metacircular evaluator in terms of abstract syntax, using procedures such as `quoted?` and `make-procedure`. In implementing the register machine, we could expand these procedures into sequences of elementary list-structure memory operations, and implement these operations on our register machine. However, this would make our evaluator very long, obscuring the basic structure with details. To clarify the presentation, we will include as primitive operations of the register machine the syntax procedures given in section 4.1.2 and the procedures for representing environments and other run-time data given in sections 4.1.3 and 4.1.4.

[19] See Batali et al. 1982 for more information on the chip and the method by which it was designed.

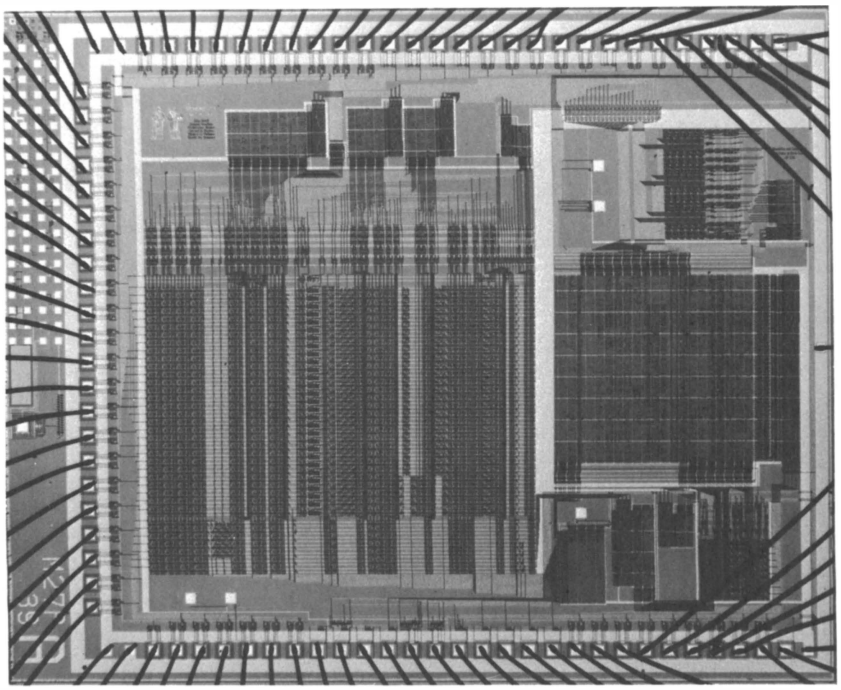


Figure 5.16 A silicon-chip implementation of an evaluator for Scheme.

In order to completely specify an evaluator that could be programmed in a low-level machine language or implemented in hardware, we would replace these operations by more elementary operations, using the list-structure implementation we described in section 5.3.

Our Scheme evaluator register machine includes a stack and seven registers: `exp`, `env`, `val`, `continue`, `proc`, `argl`, and `unev`. `Exp` is used to hold the expression to be evaluated, and `env` contains the environment in which the evaluation is to be performed. At the end of an evaluation, `val` contains the value obtained by evaluating the expression in the designated environment. The `continue` register is used to implement recursion, as explained in section 5.1.4. (The evaluator needs to call itself recursively, since evaluating an expression requires evaluating its subexpressions.) The registers `proc`, `argl`, and `unev` are used in evaluating combinations.

We will not provide a data-path diagram to show how the registers and operations of the evaluator are connected, nor will we give the complete list of machine operations. These are implicit in the evaluator's controller, which will be presented in detail.

5.4.1 The Core of the Explicit-Control Evaluator

The central element in the evaluator is the sequence of instructions beginning at `eval-dispatch`. This corresponds to the `eval` procedure of the metacircular evaluator described in section 4.1.1. When the controller starts at `eval-dispatch`, it evaluates the expression specified by `exp` in the environment specified by `env`. When evaluation is complete, the controller will go to the entry point stored in `continue`, and the `val` register will hold the value of the expression. As with the metacircular `eval`, the structure of `eval-dispatch` is a case analysis on the syntactic type of the expression to be evaluated.[20]

```
eval-dispatch
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))
  (branch (label ev-variable))
  (test (op quoted?) (reg exp))
  (branch (label ev-quoted))
  (test (op assignment?) (reg exp))
  (branch (label ev-assignment))
  (test (op definition?) (reg exp))
  (branch (label ev-definition))
  (test (op if?) (reg exp))
  (branch (label ev-if))
  (test (op lambda?) (reg exp))
  (branch (label ev-lambda))
  (test (op begin?) (reg exp))
  (branch (label ev-begin))
  (test (op application?) (reg exp))
  (branch (label ev-application))
  (goto (label unknown-expression-type))
```

Evaluating simple expressions

Numbers and strings (which are self-evaluating), variables, quotations, and `lambda` expressions have no subexpressions to be evaluated. For these, the evaluator simply places the correct value in the `val` register and continues execution at the entry point specified by `continue`. Evaluation of simple expressions is performed by the following controller code:

[20] In our controller, the dispatch is written as a sequence of `test` and `branch` instructions. Alternatively, it could have been written in a data-directed style (and in a real system it probably would have been) to avoid the need to perform sequential tests and to facilitate the definition of new expression types. A machine designed to run Lisp would probably include a `dispatch-on-type` instruction that would efficiently execute such data-directed dispatches.

```
ev-self-eval
  (assign val (reg exp))
  (goto (reg continue))
ev-variable
  (assign val (op lookup-variable-value) (reg exp) (reg env))
  (goto (reg continue))
ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto (reg continue))
ev-lambda
  (assign unev (op lambda-parameters) (reg exp))
  (assign exp (op lambda-body) (reg exp))
  (assign val (op make-procedure)
              (reg unev) (reg exp) (reg env))
  (goto (reg continue))
```

Observe how `ev-lambda` uses the `unev` and `exp` registers to hold the parameters and body of the lambda expression so that they can be passed to the `make-procedure` operation, along with the environment in `env`.

Evaluating procedure applications

A procedure application is specified by a combination containing an operator and operands. The operator is a subexpression whose value is a procedure, and the operands are subexpressions whose values are the arguments to which the procedure should be applied. The metacircular `eval` handles applications by calling itself recursively to evaluate each element of the combination, and then passing the results to `apply`, which performs the actual procedure application. The explicit-control evaluator does the same thing; these recursive calls are implemented by `goto` instructions, together with use of the stack to save registers that will be restored after the recursive call returns. Before each call we will be careful to identify which registers must be saved (because their values will be needed later).[21]

We begin the evaluation of an application by evaluating the operator to produce a procedure, which will later be applied to the evaluated operands. To evaluate the operator, we move it to the `exp` register and go to `eval-dispatch`. The environment in the `env` register is already

[21] This is an important but subtle point in translating algorithms from a procedural language, such as Lisp, to a register-machine language. As an alternative to saving only what is needed, we could save all the registers (except `val`) before each recursive call. This is called a *framed-stack* discipline. This would work but might save more registers than necessary; this could be an important consideration in a system where stack operations are expensive. Saving registers whose contents will not be needed later may also hold onto useless data that could otherwise be garbage-collected, freeing space to be reused.

the correct one in which to evaluate the operator. However, we save `env` because we will need it later to evaluate the operands. We also extract the operands into `unev` and save this on the stack. We set up `continue` so that `eval-dispatch` will resume at `ev-appl-did-operator` after the operator has been evaluated. First, however, we save the old value of `continue`, which tells the controller where to continue after the application.

```
ev-application
  (save continue)
  (save env)
  (assign unev (op operands) (reg exp))
  (save unev)
  (assign exp (op operator) (reg exp))
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch))
```

Upon returning from evaluating the operator subexpression, we proceed to evaluate the operands of the combination and to accumulate the resulting arguments in a list, held in `argl`. First we restore the unevaluated operands and the environment. We initialize `argl` to an empty list. Then we assign to the `proc` register the procedure that was produced by evaluating the operator. If there are no operands, we go directly to `apply-dispatch`. Otherwise we save `proc` on the stack and start the argument-evaluation loop:[22]

```
ev-appl-did-operator
  (restore unev)                    ; the operands
  (restore env)
  (assign argl (op empty-arglist))
  (assign proc (reg val))           ; the operator
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))
  (save proc)
```

[22]We add to the evaluator data-structure procedures in section 4.1.3 the following two procedures for manipulating argument lists:

```
(define (empty-arglist) '())

(define (adjoin-arg arg arglist)
  (append arglist (list arg)))
```

We also use an additional syntax procedure to test for the last operand in a combination:

```
(define (last-operand? ops)
  (null? (cdr ops)))
```

Each cycle of the argument-evaluation loop evaluates an operand from the list in `unev` and accumulates the result into `argl`. To evaluate an operand, we place it in the `exp` register and go to `eval-dispatch`, after setting `continue` so that execution will resume with the argument-accumulation phase. But first we save the arguments accumulated so far (held in `argl`), the environment (held in `env`), and the remaining operands to be evaluated (held in `unev`). A special case is made for the evaluation of the last operand, which is handled at `ev-appl-last-arg`.

```
ev-appl-operand-loop
  (save argl)
  (assign exp (op first-operand) (reg unev))
  (test (op last-operand?) (reg unev))
  (branch (label ev-appl-last-arg))
  (save env)
  (save unev)
  (assign continue (label ev-appl-accumulate-arg))
  (goto (label eval-dispatch))
```

When an operand has been evaluated, the value is accumulated into the list held in `argl`. The operand is then removed from the list of unevaluated operands in `unev`, and the argument-evaluation continues.

```
ev-appl-accumulate-arg
  (restore unev)
  (restore env)
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (assign unev (op rest-operands) (reg unev))
  (goto (label ev-appl-operand-loop))
```

Evaluation of the last argument is handled differently. There is no need to save the environment or the list of unevaluated operands before going to `eval-dispatch`, since they will not be required after the last operand is evaluated. Thus, we return from the evaluation to a special entry point `ev-appl-accum-last-arg`, which restores the argument list, accumulates the new argument, restores the saved procedure, and goes off to perform the application.[23]

[23]The optimization of treating the last operand specially is known as *evlis tail recursion* (see Wand 1980). We could be somewhat more efficient in the argument evaluation loop if we made evaluation of the first operand a special case too. This would permit us to postpone initializing `argl` until after evaluating the first operand, so as to avoid saving `argl` in this case. The compiler in section 5.5 performs this optimization. (Compare the `construct-arglist` procedure of section 5.5.3.)

```
ev-appl-last-arg
  (assign continue (label ev-appl-accum-last-arg))
  (goto (label eval-dispatch))
ev-appl-accum-last-arg
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore proc)
  (goto (label apply-dispatch))
```

The details of the argument-evaluation loop determine the order in which the interpreter evaluates the operands of a combination (e.g., left to right or right to left—see exercise 3.8). This order is not determined by the metacircular evaluator, which inherits its control structure from the underlying Scheme in which it is implemented.[24] Because the `first-operand` selector (used in `ev-appl-operand-loop` to extract successive operands from `unev`) is implemented as `car` and the `rest-operands` selector is implemented as `cdr`, the explicit-control evaluator will evaluate the operands of a combination in left-to-right order.

Procedure application

The entry point `apply-dispatch` corresponds to the `apply` procedure of the metacircular evaluator. By the time we get to `apply-dispatch`, the `proc` register contains the procedure to apply and `argl` contains the list of evaluated arguments to which it must be applied. The saved value of `continue` (originally passed to `eval-dispatch` and saved at `ev-application`), which tells where to return with the result of the procedure application, is on the stack. When the application is complete, the controller transfers to the entry point specified by the saved `continue`, with the result of the application in `val`. As with the metacircular `apply`, there are two cases to consider. Either the procedure to be applied is a primitive or it is a compound procedure.

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (goto (label unknown-procedure-type))
```

[24]The order of operand evaluation in the metacircular evaluator is determined by the order of evaluation of the arguments to `cons` in the procedure `list-of-values` of section 4.1.1 (see exercise 4.1).

We assume that each primitive is implemented so as to obtain its arguments from `argl` and place its result in `val`. To specify how the machine handles primitives, we would have to provide a sequence of controller instructions to implement each primitive and arrange for `primitive-apply` to dispatch to the instructions for the primitive identified by the contents of `proc`. Since we are interested in the structure of the evaluation process rather than the details of the primitives, we will instead just use an `apply-primitive-procedure` operation that applies the procedure in `proc` to the arguments in `argl`. For the purpose of simulating the evaluator with the simulator of section 5.2 we use the procedure `apply-primitive-procedure`, which calls on the underlying Scheme system to perform the application, just as we did for the metacircular evaluator in section 4.1.4. After computing the value of the primitive application, we restore `continue` and go to the designated entry point.

```
primitive-apply
  (assign val (op apply-primitive-procedure)
              (reg proc)
              (reg argl))
  (restore continue)
  (goto (reg continue))
```

To apply a compound procedure, we proceed just as with the metacircular evaluator. We construct a frame that binds the procedure's parameters to the arguments, use this frame to extend the environment carried by the procedure, and evaluate in this extended environment the sequence of expressions that forms the body of the procedure. `Ev-sequence`, described below in section 5.4.2, handles the evaluation of the sequence.

```
compound-apply
  (assign unev (op procedure-parameters) (reg proc))
  (assign env (op procedure-environment) (reg proc))
  (assign env (op extend-environment)
              (reg unev) (reg argl) (reg env))
  (assign unev (op procedure-body) (reg proc))
  (goto (label ev-sequence))
```

`Compound-apply` is the only place in the interpreter where the `env` register is ever assigned a new value. Just as in the metacircular evaluator, the new environment is constructed from the environment carried by the procedure, together with the argument list and the corresponding list of variables to be bound.

5.4.2 Sequence Evaluation and Tail Recursion

The portion of the explicit-control evaluator at `ev-sequence` is analogous to the metacircular evaluator's `eval-sequence` procedure. It handles sequences of expressions in procedure bodies or in explicit `begin` expressions.

Explicit `begin` expressions are evaluated by placing the sequence of expressions to be evaluated in `unev`, saving `continue` on the stack, and jumping to `ev-sequence`.

```
ev-begin
  (assign unev (op begin-actions) (reg exp))
  (save continue)
  (goto (label ev-sequence))
```

The implicit sequences in procedure bodies are handled by jumping to `ev-sequence` from `compound-apply`, at which point `continue` is already on the stack, having been saved at `ev-application`.

The entries at `ev-sequence` and `ev-sequence-continue` form a loop that successively evaluates each expression in a sequence. The list of unevaluated expressions is kept in `unev`. Before evaluating each expression, we check to see if there are additional expressions to be evaluated in the sequence. If so, we save the rest of the unevaluated expressions (held in `unev`) and the environment in which these must be evaluated (held in `env`) and call `eval-dispatch` to evaluate the expression. The two saved registers are restored upon the return from this evaluation, at `ev-sequence-continue`.

The final expression in the sequence is handled differently, at the entry point `ev-sequence-last-exp`. Since there are no more expressions to be evaluated after this one, we need not save `unev` or `env` before going to `eval-dispatch`. The value of the whole sequence is the value of the last expression, so after the evaluation of the last expression there is nothing left to do except continue at the entry point currently held on the stack (which was saved by `ev-application` or `ev-begin`.) Rather than setting up `continue` to arrange for `eval-dispatch` to return here and then restoring `continue` from the stack and continuing at that entry point, we restore `continue` from the stack before going to `eval-dispatch`, so that `eval-dispatch` will continue at that entry point after evaluating the expression.

```
ev-sequence
  (assign exp (op first-exp) (reg unev))
  (test (op last-exp?) (reg unev))
  (branch (label ev-sequence-last-exp))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-last-exp
  (restore continue)
  (goto (label eval-dispatch))
```

Tail recursion

In chapter 1 we said that the process described by a procedure such as

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

is an iterative process. Even though the procedure is syntactically recursive (defined in terms of itself), it is not logically necessary for an evaluator to save information in passing from one call to `sqrt-iter` to the next.[25] An evaluator that can execute a procedure such as `sqrt-iter` without requiring increasing storage as the procedure continues to call itself is called a *tail-recursive* evaluator. The metacircular implementation of the evaluator in chapter 4 does not specify whether the evaluator is tail-recursive, because that evaluator inherits its mechanism for saving state from the underlying Scheme. With the explicit-control evaluator, however, we can trace through the evaluation process to see when procedure calls cause a net accumulation of information on the stack.

Our evaluator is tail-recursive, because in order to evaluate the final expression of a sequence we transfer directly to `eval-dispatch` without saving any information on the stack. Hence, evaluating the final expression in a sequence—even if it is a procedure call (as in `sqrt-iter`, where the `if` expression, which is the last expression in the procedure

[25] We saw in section 5.1 how to implement such a process with a register machine that had no stack; the state of the process was stored in a fixed set of registers.

body, reduces to a call to `sqrt-iter`)—will not cause any information to be accumulated on the stack.[26]

If we did not think to take advantage of the fact that it was unnecessary to save information in this case, we might have implemented `eval-sequence` by treating all the expressions in a sequence in the same way—saving the registers, evaluating the expression, returning to restore the registers, and repeating this until all the expressions have been evaluated:[27]

```
ev-sequence
  (test (op no-more-exps?) (reg unev))
  (branch (label ev-sequence-end))
  (assign exp (op first-exp) (reg unev))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-end
  (restore continue)
  (goto (reg continue))
```

This may seem like a minor change to our previous code for evaluation of a sequence: The only difference is that we go through the save-restore cycle for the last expression in a sequence as well as for the others. The interpreter will still give the same value for any expression. But this change is fatal to the tail-recursive implementation, because we must now return after evaluating the final expression in a sequence in order to undo the (useless) register saves. These extra saves will accumulate during a nest of procedure calls. Consequently, processes such as `sqrt-iter` will require space proportional to the number of iterations rather than requiring constant space. This difference can be significant.

---

[26]This implementation of tail recursion in `ev-sequence` is one variety of a well-known optimization technique used by many compilers. In compiling a procedure that ends with a procedure call, one can replace the call by a jump to the called procedure's entry point. Building this strategy into the interpreter, as we have done in this section, provides the optimization uniformly throughout the language.

[27]We can define `no-more-exps?` as follows:

```
(define (no-more-exps? seq) (null? seq))
```

For example, with tail recursion, an infinite loop can be expressed using only the procedure-call mechanism:

```
(define (count n)
  (newline)
  (display n)
  (count (+ n 1)))
```

Without tail recursion, such a procedure would eventually run out of stack space, and expressing a true iteration would require some control mechanism other than procedure call.

5.4.3 Conditionals, Assignments, and Definitions

As with the metacircular evaluator, special forms are handled by selectively evaluating fragments of the expression. For an `if` expression, we must evaluate the predicate and decide, based on the value of predicate, whether to evaluate the consequent or the alternative.

Before evaluating the predicate, we save the `if` expression itself so that we can later extract the consequent or alternative. We also save the environment, which we will need later in order to evaluate the consequent or the alternative, and we save `continue`, which we will need later in order to return to the evaluation of the expression that is waiting for the value of the `if`.

```
ev-if
  (save exp)                          ; save expression for later
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg exp))
  (goto (label eval-dispatch))    ; evaluate the predicate
```

When we return from evaluating the predicate, we test whether it was true or false and, depending on the result, place either the consequent or the alternative in `exp` before going to `eval-dispatch`. Notice that restoring `env` and `continue` here sets up `eval-dispatch` to have the correct environment and to continue at the right place to receive the value of the `if` expression.

```
ev-if-decide
  (restore continue)
  (restore env)
  (restore exp)
  (test (op true?) (reg val))
  (branch (label ev-if-consequent))
```

```
ev-if-alternative
  (assign exp (op if-alternative) (reg exp))
  (goto (label eval-dispatch))
ev-if-consequent
  (assign exp (op if-consequent) (reg exp))
  (goto (label eval-dispatch))
```

Assignments and definitions

Assignments are handled by `ev-assignment`, which is reached from `eval-dispatch` with the assignment expression in `exp`. The code at `ev-assignment` first evaluates the value part of the expression and then installs the new value in the environment. `Set-variable-value!` is assumed to be available as a machine operation.

```
ev-assignment
  (assign unev (op assignment-variable) (reg exp))
  (save unev)                    ; save variable for later
  (assign exp (op assignment-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-assignment-1))
  (goto (label eval-dispatch))   ; evaluate the assignment value
ev-assignment-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
   (op set-variable-value!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))
```

Definitions are handled in a similar way:

```
ev-definition
  (assign unev (op definition-variable) (reg exp))
  (save unev)                    ; save variable for later
  (assign exp (op definition-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-definition-1))
  (goto (label eval-dispatch))   ; evaluate the definition value
ev-definition-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
   (op define-variable!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))
```

Exercise 5.23

Extend the evaluator to handle derived expressions such as `cond`, `let`, and so on (section 4.1.2). You may "cheat" and assume that the syntax transformers such as `cond->if` are available as machine operations.[28]

Exercise 5.24

Implement `cond` as a new basic special form without reducing it to `if`. You will have to construct a loop that tests the predicates of successive `cond` clauses until you find one that is true, and then use `ev-sequence` to evaluate the actions of the clause.

Exercise 5.25

Modify the evaluator so that it uses normal-order evaluation, based on the lazy evaluator of section 4.2.

5.4.4 Running the Evaluator

With the implementation of the explicit-control evaluator we come to the end of a development, begun in chapter 1, in which we have explored successively more precise models of the evaluation process. We started with the relatively informal substitution model, then extended this in chapter 3 to the environment model, which enabled us to deal with state and change. In the metacircular evaluator of chapter 4, we used Scheme itself as a language for making more explicit the environment structure constructed during evaluation of an expression. Now, with register machines, we have taken a close look at the evaluator's mechanisms for storage management, argument passing, and control. At each new level of description, we have had to raise issues and resolve ambiguities that were not apparent at the previous, less precise treatment of evaluation. To understand the behavior of the explicit-control evaluator, we can simulate it and monitor its performance.

We will install a driver loop in our evaluator machine. This plays the role of the `driver-loop` procedure of section 4.1.4. The evaluator will repeatedly print a prompt, read an expression, evaluate the expression by going to `eval-dispatch`, and print the result. The following

[28]This isn't really cheating. In an actual implementation built from scratch, we would use our explicit-control evaluator to interpret a Scheme program that performs source-level transformations like `cond->if` in a syntax phase that runs before execution.

instructions form the beginning of the explicit-control evaluator's controller sequence:[29]

```
read-eval-print-loop
  (perform (op initialize-stack))
  (perform
   (op prompt-for-input) (const ";;; EC-Eval input:"))
  (assign exp (op read))
  (assign env (op get-global-environment))
  (assign continue (label print-result))
  (goto (label eval-dispatch))
print-result
  (perform
   (op announce-output) (const ";;; EC-Eval value:"))
  (perform (op user-print) (reg val))
  (goto (label read-eval-print-loop))
```

When we encounter an error in a procedure (such as the "unknown procedure type error" indicated at `apply-dispatch`), we print an error message and return to the driver loop.[30]

```
unknown-expression-type
  (assign val (const unknown-expression-type-error))
  (goto (label signal-error))

unknown-procedure-type
  (restore continue)      ; clean up stack (from apply-dispatch)
  (assign val (const unknown-procedure-type-error))
  (goto (label signal-error))

signal-error
  (perform (op user-print) (reg val))
  (goto (label read-eval-print-loop))
```

---

[29]We assume here that `read` and the various printing operations are available as primitive machine operations, which is useful for our simulation, but completely unrealistic in practice. These are actually extremely complex operations. In practice, they would be implemented using low-level input-output operations such as transferring single characters to and from a device.

To support the `get-global-environment` operation we define

```
(define the-global-environment (setup-environment))

(define (get-global-environment)
  the-global-environment)
```

[30]There are other errors that we would like the interpreter to handle, but these are not so simple. See exercise 5.30.

For the purposes of the simulation, we initialize the stack each time through the driver loop, since it might not be empty after an error (such as an undefined variable) interrupts an evaluation.[31]

If we combine all the code fragments presented in sections 5.4.1–5.4.4, we can create an evaluator machine model that we can run using the register-machine simulator of section 5.2.

```
(define eceval
  (make-machine
   '(exp env val proc argl continue unev)
   eceval-operations
  '(
    read-eval-print-loop
      ⟨entire machine controller as given above⟩
   )))
```

We must define Scheme procedures to simulate the operations used as primitives by the evaluator. These are the same procedures we used for the metacircular evaluator in section 4.1, together with the few additional ones defined in footnotes throughout section 5.4.

```
(define eceval-operations
  (list (list 'self-evaluating? self-evaluating)
        ⟨complete list of operations for eceval machine⟩))
```

Finally, we can initialize the global environment and run the evaluator:

```
(define the-global-environment (setup-environment))

(start eceval)

;;; EC-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
;;; EC-Eval value:
ok

;;; EC-Eval input:
(append '(a b c) '(d e f))
;;; EC-Eval value:
(a b c d e f)
```

[31] We could perform the stack initialization only after errors, but doing it in the driver loop will be convenient for monitoring the evaluator's performance, as described below.

Of course, evaluating expressions in this way will take much longer than if we had directly typed them into Scheme, because of the multiple levels of simulation involved. Our expressions are evaluated by the explicit-control-evaluator machine, which is being simulated by a Scheme program, which is itself being evaluated by the Scheme interpreter.

Monitoring the performance of the evaluator

Simulation can be a powerful tool to guide the implementation of evaluators. Simulations make it easy not only to explore variations of the register-machine design but also to monitor the performance of the simulated evaluator. For example, one important factor in performance is how efficiently the evaluator uses the stack. We can observe the number of stack operations required to evaluate various expressions by defining the evaluator register machine with the version of the simulator that collects statistics on stack use (section 5.2.4), and adding an instruction at the evaluator's `print-result` entry point to print the statistics:

```
print-result
  (perform (op print-stack-statistics)) ; added instruction
  (perform
   (op announce-output) (const ";;; EC-Eval value:"))
  ... ; same as before
```

Interactions with the evaluator now look like this:

```
;;; EC-Eval input:
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
(total-pushes = 3 maximum-depth = 3)
;;; EC-Eval value:
ok

;;; EC-Eval input:
(factorial 5)
(total-pushes = 144 maximum-depth = 28)
;;; EC-Eval value:
120
```

Note that the driver loop of the evaluator reinitializes the stack at the start of each interaction, so that the statistics printed will refer only to stack operations used to evaluate the previous expression.

Exercise 5.26

Use the monitored stack to explore the tail-recursive property of the evaluator (section 5.4.2). Start the evaluator and define the iterative `factorial` procedure from section 1.2.1:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Run the procedure with some small values of n. Record the maximum stack depth and the number of pushes required to compute $n!$ for each of these values.

a. You will find that the maximum depth required to evaluate $n!$ is independent of n. What is that depth?

b. Determine from your data a formula in terms of n for the total number of push operations used in evaluating $n!$ for any $n \geq 1$. Note that the number of operations used is a linear function of n and is thus determined by two constants.

Exercise 5.27

For comparison with exercise 5.26, explore the behavior of the following procedure for computing factorials recursively:

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

By running this procedure with the monitored stack, determine, as a function of n, the maximum depth of the stack and the total number of pushes used in evaluating $n!$ for $n \geq 1$. (Again, these functions will be linear.) Summarize your experiments by filling in the following table with the appropriate expressions in terms of n:

	Maximum depth	Number of pushes
Recursive factorial		
Iterative factorial		

The maximum depth is a measure of the amount of space used by the evaluator in carrying out the computation, and the number of pushes correlates well with the time required.

Exercise 5.28

Modify the definition of the evaluator by changing `eval-sequence` as described in section 5.4.2 so that the evaluator is no longer tail-recursive. Rerun your experiments from exercises 5.26 and 5.27 to demonstrate that both versions of the `factorial` procedure now require space that grows linearly with their input.

Exercise 5.29

Monitor the stack operations in the tree-recursive Fibonacci computation:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

a. Give a formula in terms of n for the maximum depth of the stack required to compute Fib(n) for $n \geq 2$. Hint: In section 1.2.2 we argued that the space used by this process grows linearly with n.

b. Give a formula for the total number of pushes used to compute Fib(n) for $n \geq 2$. You should find that the number of pushes (which correlates well with the time used) grows exponentially with n. Hint: Let $S(n)$ be the number of pushes used in computing Fib(n). You should be able to argue that there is a formula that expresses $S(n)$ in terms of $S(n-1)$, $S(n-2)$, and some fixed "overhead" constant k that is independent of n. Give the formula, and say what k is. Then show that $S(n)$ can be expressed as $a\text{Fib}(n+1) + b$ and give the values of a and b.

Exercise 5.30

Our evaluator currently catches and signals only two kinds of errors—unknown expression types and unknown procedure types. Other errors will take us out of the evaluator read-eval-print loop. When we run the evaluator using the register-machine simulator, these errors are caught by the underlying Scheme system. This is analogous to the computer crashing when a user program makes an error.[32] It is a large project to make a real error system work, but it is well worth the effort to understand what is involved here.

a. Errors that occur in the evaluation process, such as an attempt to access an unbound variable, could be caught by changing the lookup operation to make it return a distinguished condition code, which cannot be a possible value of any user variable. The evaluator can test for this condition code and then do what is necessary to go to `signal-error`. Find all of the places in the evaluator where such a change is necessary and fix them. This is lots of work.

---

[32]Regrettably, this is the normal state of affairs in conventional compiler-based language systems such as C. In UNIX™ the system "dumps core," and in DOS/Windows™ it becomes catatonic. The Macintosh™ displays a picture of an exploding bomb and offers you the opportunity to reboot the computer—if you're lucky.

b. Much worse is the problem of handling errors that are signaled by applying primitive procedures, such as an attempt to divide by zero or an attempt to extract the `car` of a symbol. In a professionally written high-quality system, each primitive application is checked for safety as part of the primitive. For example, every call to `car` could first check that the argument is a pair. If the argument is not a pair, the application would return a distinguished condition code to the evaluator, which would then report the failure. We could arrange for this in our register-machine simulator by making each primitive procedure check for applicability and returning an appropriate distinguished condition code on failure. Then the `primitive-apply` code in the evaluator can check for the condition code and go to `signal-error` if necessary. Build this structure and make it work. This is a major project.

5.5 Compilation

The explicit-control evaluator of section 5.4 is a register machine whose controller interprets Scheme programs. In this section we will see how to run Scheme programs on a register machine whose controller is not a Scheme interpreter.

The explicit-control evaluator machine is universal—it can carry out any computational process that can be described in Scheme. The evaluator's controller orchestrates the use of its data paths to perform the desired computation. Thus, the evaluator's data paths are universal: They are sufficient to perform any computation we desire, given an appropriate controller.[33]

Commercial general-purpose computers are register machines organized around a collection of registers and operations that constitute an efficient and convenient universal set of data paths. The controller for a general-purpose machine is an interpreter for a register-machine language like the one we have been using. This language is called the *native language* of the machine, or simply *machine language.* Programs written in machine language are sequences of instructions that use the machine's data paths. For example, the explicit-control evaluator's instruction sequence can be thought of as a machine-language program for a general-purpose computer rather than as the controller for a specialized interpreter machine.

[33]This is a theoretical statement. We are not claiming that the evaluator's data paths are a particularly convenient or efficient set of data paths for a general-purpose computer. For example, they are not very good for implementing high-performance floating-point calculations or calculations that intensively manipulate bit vectors.

There are two common strategies for bridging the gap between higher-level languages and register-machine languages. The explicit-control evaluator illustrates the strategy of interpretation. An interpreter written in the native language of a machine configures the machine to execute programs written in a language (called the *source language*) that may differ from the native language of the machine performing the evaluation. The primitive procedures of the source language are implemented as a library of subroutines written in the native language of the given machine. A program to be interpreted (called the *source program*) is represented as a data structure. The interpreter traverses this data structure, analyzing the source program. As it does so, it simulates the intended behavior of the source program by calling appropriate primitive subroutines from the library.

In this section, we explore the alternative strategy of *compilation.* A compiler for a given source language and machine translates a source program into an equivalent program (called the *object program*) written in the machine's native language. The compiler that we implement in this section translates programs written in Scheme into sequences of instructions to be executed using the explicit-control evaluator machine's data paths.[34]

Compared with interpretation, compilation can provide a great increase in the efficiency of program execution, as we will explain below in the overview of the compiler. On the other hand, an interpreter provides a more powerful environment for interactive program development and debugging, because the source program being executed is available at run time to be examined and modified. In addition, because the entire library of primitives is present, new programs can be constructed and added to the system during debugging.

In view of the complementary advantages of compilation and interpretation, modern program-development environments pursue a mixed strategy. Lisp interpreters are generally organized so that interpreted procedures and compiled procedures can call each other. This enables a programmer to compile those parts of a program that are assumed to be

[34]Actually, the machine that runs compiled code can be simpler than the interpreter machine, because we won't use the `exp` and `unev` registers. The interpreter used these to hold pieces of unevaluated expressions. With the compiler, however, these expressions get built into the compiled code that the register machine will run. For the same reason, we don't need the machine operations that deal with expression syntax. But compiled code will use a few additional machine operations (to represent compiled procedure objects) that didn't appear in the explicit-control evaluator machine.

debugged, thus gaining the efficiency advantage of compilation, while retaining the interpretive mode of execution for those parts of the program that are in the flux of interactive development and debugging. In section 5.5.7, after we have implemented the compiler, we will show how to interface it with our interpreter to produce an integrated interpreter-compiler development system.

An overview of the compiler

Our compiler is much like our interpreter, both in its structure and in the function it performs. Accordingly, the mechanisms used by the compiler for analyzing expressions will be similar to those used by the interpreter. Moreover, to make it easy to interface compiled and interpreted code, we will design the compiler to generate code that obeys the same conventions of register usage as the interpreter: The environment will be kept in the `env` register, argument lists will be accumulated in `argl`, a procedure to be applied will be in `proc`, procedures will return their answers in `val`, and the location to which a procedure should return will be kept in `continue`. In general, the compiler translates a source program into an object program that performs essentially the same register operations as would the interpreter in evaluating the same source program.

This description suggests a strategy for implementing a rudimentary compiler: We traverse the expression in the same way the interpreter does. When we encounter a register instruction that the interpreter would perform in evaluating the expression, we do not execute the instruction but instead accumulate it into a sequence. The resulting sequence of instructions will be the object code. Observe the efficiency advantage of compilation over interpretation. Each time the interpreter evaluates an expression—for example, `(f 84 96)`—it performs the work of classifying the expression (discovering that this is a procedure application) and testing for the end of the operand list (discovering that there are two operands). With a compiler, the expression is analyzed only once, when the instruction sequence is generated at compile time. The object code produced by the compiler contains only the instructions that evaluate the operator and the two operands, assemble the argument list, and apply the procedure (in `proc`) to the arguments (in `argl`).

This is the same kind of optimization we implemented in the analyzing evaluator of section 4.1.7. But there are further opportunities to gain efficiency in compiled code. As the interpreter runs, it follows a process that must be applicable to any expression in the language. In contrast, a given segment of compiled code is meant to execute some particular

expression. This can make a big difference, for example in the use of the stack to save registers. When the interpreter evaluates an expression, it must be prepared for any contingency. Before evaluating a subexpression, the interpreter saves all registers that will be needed later, because the subexpression might require an arbitrary evaluation. A compiler, on the other hand, can exploit the structure of the particular expression it is processing to generate code that avoids unnecessary stack operations.

As a case in point, consider the combination `(f 84 96)`. Before the interpreter evaluates the operator of the combination, it prepares for this evaluation by saving the registers containing the operands and the environment, whose values will be needed later. The interpreter then evaluates the operator to obtain the result in `val`, restores the saved registers, and finally moves the result from `val` to `proc`. However, in the particular expression we are dealing with, the operator is the symbol `f`, whose evaluation is accomplished by the machine operation `lookup-variable-value`, which does not alter any registers. The compiler that we implement in this section will take advantage of this fact and generate code that evaluates the operator using the instruction

```
(assign proc (op lookup-variable-value) (const f) (reg env))
```

This code not only avoids the unnecessary saves and restores but also assigns the value of the lookup directly to `proc`, whereas the interpreter would obtain the result in `val` and then move this to `proc`.

A compiler can also optimize access to the environment. Having analyzed the code, the compiler can in many cases know in which frame a particular variable will be located and access that frame directly, rather than performing the `lookup-variable-value` search. We will discuss how to implement such variable access in section 5.5.6. Until then, however, we will focus on the kind of register and stack optimizations described above. There are many other optimizations that can be performed by a compiler, such as coding primitive operations "in line" instead of using a general `apply` mechanism (see exercise 5.38); but we will not emphasize these here. Our main goal in this section is to illustrate the compilation process in a simplified (but still interesting) context.

5.5.1 Structure of the Compiler

In section 4.1.7 we modified our original metacircular interpreter to separate analysis from execution. We analyzed each expression to produce an execution procedure that took an environment as argument and per-

formed the required operations. In our compiler, we will do essentially the same analysis. Instead of producing execution procedures, however, we will generate sequences of instructions to be run by our register machine.

The procedure `compile` is the top-level dispatch in the compiler. It corresponds to the `eval` procedure of section 4.1.1, the `analyze` procedure of section 4.1.7, and the `eval-dispatch` entry point of the explicit-control-evaluator in section 5.4.1. The compiler, like the interpreters, uses the expression-syntax procedures defined in section 4.1.2.[35] `Compile` performs a case analysis on the syntactic type of the expression to be compiled. For each type of expression, it dispatches to a specialized *code generator*:

```
(define (compile exp target linkage)
  (cond ((self-evaluating? exp)
         (compile-self-evaluating exp target linkage))
        ((quoted? exp) (compile-quoted exp target linkage))
        ((variable? exp)
         (compile-variable exp target linkage))
        ((assignment? exp)
         (compile-assignment exp target linkage))
        ((definition? exp)
         (compile-definition exp target linkage))
        ((if? exp) (compile-if exp target linkage))
        ((lambda? exp) (compile-lambda exp target linkage))
        ((begin? exp)
         (compile-sequence (begin-actions exp)
                           target
                           linkage))
        ((cond? exp) (compile (cond->if exp) target linkage))
        ((application? exp)
         (compile-application exp target linkage))
        (else
         (error "Unknown expression type -- COMPILE" exp))))
```

[35]Notice, however, that our compiler is a Scheme program, and the syntax procedures that it uses to manipulate expressions are the actual Scheme procedures used with the metacircular evaluator. For the explicit-control evaluator, in contrast, we assumed that equivalent syntax operations were available as operations for the register machine. (Of course, when we simulated the register machine in Scheme, we used the actual Scheme procedures in our register machine simulation.)

Targets and linkages

`Compile` and the code generators that it calls take two arguments in addition to the expression to compile. There is a *target*, which specifies the register in which the compiled code is to return the value of the expression. There is also a *linkage descriptor*, which describes how the code resulting from the compilation of the expression should proceed when it has finished its execution. The linkage descriptor can require that the code do one of the following three things:

- continue at the next instruction in sequence (this is specified by the linkage descriptor `next`),
- return from the procedure being compiled (this is specified by the linkage descriptor `return`), or
- jump to a named entry point (this is specified by using the designated label as the linkage descriptor).

For example, compiling the expression `5` (which is self-evaluating) with a target of the `val` register and a linkage of `next` should produce the instruction

```
(assign val (const 5))
```

Compiling the same expression with a linkage of `return` should produce the instructions

```
(assign val (const 5))
(goto (reg continue))
```

In the first case, execution will continue with the next instruction in the sequence. In the second case, we will return from a procedure call. In both cases, the value of the expression will be placed into the target `val` register.

Instruction sequences and stack usage

Each code generator returns an *instruction sequence* containing the object code it has generated for the expression. Code generation for a compound expression is accomplished by combining the output from simpler code generators for component expressions, just as evaluation of a compound expression is accomplished by evaluating the component expressions.

The simplest method for combining instruction sequences is a procedure called `append-instruction-sequences`. It takes as arguments any number of instruction sequences that are to be executed sequentially; it appends them and returns the combined sequence. That is, if ⟨*seq*$_1$⟩ and ⟨*seq*$_2$⟩ are sequences of instructions, then evaluating

`(append-instruction-sequences` ⟨*seq*$_1$⟩ ⟨*seq*$_2$⟩`)`

produces the sequence

⟨*seq*$_1$⟩
⟨*seq*$_2$⟩

Whenever registers might need to be saved, the compiler's code generators use `preserving`, which is a more subtle method for combining instruction sequences. `Preserving` takes three arguments: a set of registers and two instruction sequences that are to be executed sequentially. It appends the sequences in such a way that the contents of each register in the set is preserved over the execution of the first sequence, if this is needed for the execution of the second sequence. That is, if the first sequence modifies the register and the second sequence actually needs the register's original contents, then `preserving` wraps a `save` and a `restore` of the register around the first sequence before appending the sequences. Otherwise, `preserving` simply returns the appended instruction sequences. Thus, for example,

`(preserving (list` ⟨*reg*$_1$⟩ ⟨*reg*$_2$⟩`)` ⟨*seq*$_1$⟩ ⟨*seq*$_2$⟩`)`

produces one of the following four sequences of instructions, depending on how ⟨*seq*$_1$⟩ and ⟨*seq*$_2$⟩ use ⟨*reg*$_1$⟩ and ⟨*reg*$_2$⟩:

⟨*seq*$_1$⟩
⟨*seq*$_2$⟩
	`(save` ⟨*reg*$_1$⟩`)`
	⟨*seq*$_1$⟩
	`(restore` ⟨*reg*$_1$⟩`)`
	⟨*seq*$_2$⟩
		`(save` ⟨*reg*$_2$⟩`)`
		⟨*seq1*⟩
		`(restore` ⟨*reg*$_2$⟩`)`
		⟨*seq2*⟩
			`(save` ⟨*reg*$_2$⟩`)`
			`(save` ⟨*reg*$_1$⟩`)`
			⟨*seq*$_1$⟩
			`(restore` ⟨*reg*$_1$⟩`)`
			`(restore` ⟨*reg*$_2$⟩`)`
			⟨*seq*$_2$⟩

By using `preserving` to combine instruction sequences the compiler avoids unnecessary stack operations. This also isolates the details of whether or not to generate `save` and `restore` instructions within the `preserving` procedure, separating them from the concerns that arise in writing each of the individual code generators. In fact no `save` or `restore` instructions are explicitly produced by the code generators.

In principle, we could represent an instruction sequence simply as a list of instructions. `Append-instruction-sequences` could then combine instruction sequences by performing an ordinary list `append`. However, `preserving` would then be a complex operation, because it would have to analyze each instruction sequence to determine how the sequence uses its registers. `Preserving` would be inefficient as well as complex, because it would have to analyze each of its instruction sequence arguments, even though these sequences might themselves have been constructed by calls to `preserving`, in which case their parts would have already been analyzed. To avoid such repetitious analysis we will associate with each instruction sequence some information about its register use. When we construct a basic instruction sequence we will provide this information explicitly, and the procedures that combine instruction sequences will derive register-use information for the combined sequence from the information associated with the component sequences.

An instruction sequence will contain three pieces of information:

- the set of registers that must be initialized before the instructions in the sequence are executed (these registers are said to be *needed* by the sequence),
- the set of registers whose values are modified by the instructions in the sequence, and
- the actual instructions (also called *statements*) in the sequence.

We will represent an instruction sequence as a list of its three parts. The constructor for instruction sequences is thus

```
(define (make-instruction-sequence needs modifies statements)
  (list needs modifies statements))
```

For example, the two-instruction sequence that looks up the value of the variable `x` in the current environment, assigns the result to `val`, and then returns, requires registers `env` and `continue` to have been initialized, and modifies register `val`. This sequence would therefore be constructed as

```
(make-instruction-sequence '(env continue) '(val)
 '((assign val
           (op lookup-variable-value) (const x) (reg env))
   (goto (reg continue))))
```

We sometimes need to construct an instruction sequence with no statements:

```
(define (empty-instruction-sequence)
  (make-instruction-sequence '() '() '()))
```

The procedures for combining instruction sequences are shown in section 5.5.4.

Exercise 5.31

In evaluating a procedure application, the explicit-control evaluator always saves and restores the `env` register around the evaluation of the operator, saves and restores `env` around the evaluation of each operand (except the final one), saves and restores `argl` around the evaluation of each operand, and saves and restores `proc` around the evaluation of the operand sequence. For each of the following combinations, say which of these `save` and `restore` operations are superfluous and thus could be eliminated by the compiler's `preserving` mechanism:

```
(f 'x 'y)

((f) 'x 'y)

(f (g 'x) y)

(f (g 'x) 'y)
```

Exercise 5.32

Using the `preserving` mechanism, the compiler will avoid saving and restoring `env` around the evaluation of the operator of a combination in the case where the operator is a symbol. We could also build such optimizations into the evaluator. Indeed, the explicit-control evaluator of section 5.4 already performs a similar optimization, by treating combinations with no operands as a special case.

a. Extend the explicit-control evaluator to recognize as a separate class of expressions combinations whose operator is a symbol, and to take advantage of this fact in evaluating such expressions.

b. Alyssa P. Hacker suggests that by extending the evaluator to recognize more and more special cases we could incorporate all the compiler's optimizations, and that this would eliminate the advantage of compilation altogether. What do you think of this idea?

5.5.2 Compiling Expressions

In this section and the next we implement the code generators to which the `compile` procedure dispatches.

Compiling linkage code

In general, the output of each code generator will end with instructions—generated by the procedure `compile-linkage`—that implement the required linkage. If the linkage is `return` then we must generate the instruction `(goto (reg continue))`. This needs the `continue` register and does not modify any registers. If the linkage is `next`, then we needn't include any additional instructions. Otherwise, the linkage is a label, and we generate a `goto` to that label, an instruction that does not need or modify any registers.[36]

```
(define (compile-linkage linkage)
  (cond ((eq? linkage 'return)
         (make-instruction-sequence '(continue) '()
          '((goto (reg continue)))))
        ((eq? linkage 'next)
         (empty-instruction-sequence))
        (else
         (make-instruction-sequence '() '()
          `((goto (label ,linkage)))))))
```

The linkage code is appended to an instruction sequence by `preserving` the `continue` register, since a `return` linkage will require the `continue` register: If the given instruction sequence modifies `continue` and the linkage code needs it, `continue` will be saved and restored.

```
(define (end-with-linkage linkage instruction-sequence)
  (preserving '(continue)
   instruction-sequence
   (compile-linkage linkage)))
```

Compiling simple expressions

The code generators for self-evaluating expressions, quotations, and variables construct instruction sequences that assign the required value to the target register and then proceed as specified by the linkage descriptor.

```
(define (compile-self-evaluating exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '() (list target)
    `((assign ,target (const ,exp))))))
```

[36]This procedure uses a feature of Lisp called *backquote* (or *quasiquote*) that is handy for constructing lists. Preceding a list with a backquote symbol is much like quoting it, except that anything in the list that is flagged with a comma is evaluated.

For example, if the value of `linkage` is the symbol `branch25`, then the expression `` `((goto (label ,linkage))) `` evaluates to the list `((goto (label branch25)))`. Similarly, if the value of `x` is the list `(a b c)`, then `` `(1 2 ,(car x)) `` evaluates to the list `(1 2 a)`.

```
(define (compile-quoted exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '() (list target)
    `((assign ,target (const ,(text-of-quotation exp)))))))

(define (compile-variable exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '(env) (list target)
    `((assign ,target
              (op lookup-variable-value)
              (const ,exp)
              (reg env))))))
```

All these assignment instructions modify the target register, and the one that looks up a variable needs the `env` register.

Assignments and definitions are handled much as they are in the interpreter. We recursively generate code that computes the value to be assigned to the variable, and append to it a two-instruction sequence that actually sets or defines the variable and assigns the value of the whole expression (the symbol `ok`) to the target register. The recursive compilation has target `val` and linkage `next` so that the code will put its result into `val` and continue with the code that is appended after it. The appending is done preserving `env`, since the environment is needed for setting or defining the variable and the code for the variable value could be the compilation of a complex expression that might modify the registers in arbitrary ways.

```
(define (compile-assignment exp target linkage)
  (let ((var (assignment-variable exp))
        (get-value-code
         (compile (assignment-value exp) 'val 'next)))
    (end-with-linkage linkage
     (preserving '(env)
      get-value-code
      (make-instruction-sequence '(env val) (list target)
       `((perform (op set-variable-value!)
                  (const ,var)
                  (reg val)
                  (reg env))
         (assign ,target (const ok))))))))
```

```
(define (compile-definition exp target linkage)
  (let ((var (definition-variable exp))
        (get-value-code
         (compile (definition-value exp) 'val 'next)))
    (end-with-linkage linkage
     (preserving '(env)
      get-value-code
      (make-instruction-sequence '(env val) (list target)
       `((perform (op define-variable!)
                  (const ,var)
                  (reg val)
                  (reg env))
         (assign ,target (const ok))))))))
```

The appended two-instruction sequence requires `env` and `val` and modifies the target. Note that although we preserve `env` for this sequence, we do not preserve `val`, because the `get-value-code` is designed to explicitly place its result in `val` for use by this sequence. (In fact, if we did preserve `val`, we would have a bug, because this would cause the previous contents of `val` to be restored right after the `get-value-code` is run.)

Compiling conditional expressions

The code for an `if` expression compiled with a given target and linkage has the form

```
⟨compilation of predicate, target val, linkage next⟩
(test (op false?) (reg val))
(branch (label false-branch))
true-branch
⟨compilation of consequent with given target and given linkage or after-if⟩
false-branch
⟨compilation of alternative with given target and linkage⟩
after-if
```

To generate this code, we compile the predicate, consequent, and alternative, and combine the resulting code with instructions to test the predicate result and with newly generated labels to mark the true and

false branches and the end of the conditional.[37] In this arrangement of code, we must branch around the true branch if the test is false. The only slight complication is in how the linkage for the true branch should be handled. If the linkage for the conditional is `return` or a label, then the true and false branches will both use this same linkage. If the linkage is `next`, the true branch ends with a jump around the code for the false branch to the label at the end of the conditional.

```
(define (compile-if exp target linkage)
  (let ((t-branch (make-label 'true-branch))
        (f-branch (make-label 'false-branch))
        (after-if (make-label 'after-if)))
    (let ((consequent-linkage
           (if (eq? linkage 'next) after-if linkage)))
      (let ((p-code (compile (if-predicate exp) 'val 'next))
            (c-code
             (compile
              (if-consequent exp) target consequent-linkage))
            (a-code
             (compile (if-alternative exp) target linkage)))
        (preserving '(env continue)
         p-code
         (append-instruction-sequences
          (make-instruction-sequence '(val) '()
           `((test (op false?) (reg val))
             (branch (label ,f-branch))))
          (parallel-instruction-sequences
           (append-instruction-sequences t-branch c-code)
           (append-instruction-sequences f-branch a-code))
          after-if))))))
```

`Env` is preserved around the predicate code because it could be needed by the true and false branches, and `continue` is preserved because it could

---

[37] We can't just use the labels `true-branch`, `false-branch`, and `after-if` as shown above, because there might be more than one `if` in the program. The compiler uses the procedure `make-label` to generate labels. `Make-label` takes a symbol as argument and returns a new symbol that begins with the given symbol. For example, successive calls to `(make-label 'a)` would return `a1`, `a2`, and so on. `Make-label` can be implemented similarly to the generation of unique variable names in the query language, as follows:

```
(define label-counter 0)

(define (new-label-number)
  (set! label-counter (+ 1 label-counter))
  label-counter)

(define (make-label name)
  (string->symbol
    (string-append (symbol->string name)
                   (number->string (new-label-number)))))
```

be needed by the linkage code in those branches. The code for the true and false branches (which are not executed sequentially) is appended using a special combiner `parallel-instruction-sequences` described in section 5.5.4.

Note that `cond` is a derived expression, so all that the compiler needs to do handle it is to apply the `cond->if` transformer (from section 4.1.2) and compile the resulting `if` expression.

Compiling sequences

The compilation of sequences (from procedure bodies or explicit `begin` expressions) parallels their evaluation. Each expression of the sequence is compiled—the last expression with the linkage specified for the sequence, and the other expressions with linkage `next` (to execute the rest of the sequence). The instruction sequences for the individual expressions are appended to form a single instruction sequence, such that `env` (needed for the rest of the sequence) and `continue` (possibly needed for the linkage at the end of the sequence) are preserved.

```
(define (compile-sequence seq target linkage)
  (if (last-exp? seq)
      (compile (first-exp seq) target linkage)
      (preserving '(env continue)
       (compile (first-exp seq) target 'next)
       (compile-sequence (rest-exps seq) target linkage))))
```

Compiling `lambda` expressions

`Lambda` expressions construct procedures. The object code for a `lambda` expression must have the form

⟨*construct procedure object and assign it to target register*⟩
⟨*linkage*⟩

When we compile the `lambda` expression, we also generate the code for the procedure body. Although the body won't be executed at the time of procedure construction, it is convenient to insert it into the object code right after the code for the `lambda`. If the linkage for the `lambda` expression is a label or `return`, this is fine. But if the linkage is `next`, we will need to skip around the code for the procedure body by using a linkage that jumps to a label that is inserted after the body. The object code thus has the form

⟨*construct procedure object and assign it to target register*⟩
⟨*code for given linkage*⟩ *or* `(goto (label after-lambda))`
⟨*compilation of procedure body*⟩
`after-lambda`

`Compile-lambda` generates the code for constructing the procedure object followed by the code for the procedure body. The procedure object will be constructed at run time by combining the current environment (the environment at the point of definition) with the entry point to the compiled procedure body (a newly generated label).[38]

```
(define (compile-lambda exp target linkage)
  (let ((proc-entry (make-label 'entry))
        (after-lambda (make-label 'after-lambda)))
    (let ((lambda-linkage
           (if (eq? linkage 'next) after-lambda linkage)))
      (append-instruction-sequences
       (tack-on-instruction-sequence
        (end-with-linkage lambda-linkage
         (make-instruction-sequence '(env) (list target)
          `((assign ,target
                    (op make-compiled-procedure)
                    (label ,proc-entry)
                    (reg env)))))
        (compile-lambda-body exp proc-entry))
       after-lambda))))
```

`Compile-lambda` uses the special combiner `tack-on-instruction-sequence` (section 5.5.4) rather than `append-instruction-sequences` to append the procedure body to the `lambda` expression code, because the body is not part of the sequence of instructions that will be executed when the combined sequence is entered; rather, it is in the sequence only because that was a convenient place to put it.

`Compile-lambda-body` constructs the code for the body of the procedure. This code begins with a label for the entry point. Next come instructions that will cause the run-time evaluation environment to switch to the correct environment for evaluating the procedure body—namely, the definition environment of the procedure, extended to include the bindings of the formal parameters to the arguments with which the pro-

[38]We need machine operations to implement a data structure for representing compiled procedures, analogous to the structure for compound procedures described in section 4.1.3:

```
(define (make-compiled-procedure entry env)
  (list 'compiled-procedure entry env))

(define (compiled-procedure? proc)
  (tagged-list? proc 'compiled-procedure))

(define (compiled-procedure-entry c-proc) (cadr c-proc))

(define (compiled-procedure-env c-proc) (caddr c-proc))
```

cedure is called. After this comes the code for the sequence of expressions that makes up the procedure body. The sequence is compiled with linkage `return` and target `val` so that it will end by returning from the procedure with the procedure result in `val`.

```
(define (compile-lambda-body exp proc-entry)
  (let ((formals (lambda-parameters exp)))
    (append-instruction-sequences
     (make-instruction-sequence '(env proc argl) '(env)
      `(,proc-entry
        (assign env (op compiled-procedure-env) (reg proc))
        (assign env
                (op extend-environment)
                (const ,formals)
                (reg argl)
                (reg env))))
     (compile-sequence (lambda-body exp) 'val 'return))))
```

5.5.3 Compiling Combinations

The essence of the compilation process is the compilation of procedure applications. The code for a combination compiled with a given target and linkage has the form

⟨ *compilation of operator, target* `proc`, *linkage* `next` ⟩
⟨ *evaluate operands and construct argument list in* `argl` ⟩
⟨ *compilation of procedure call with given target and linkage* ⟩

The registers `env`, `proc`, and `argl` may have to be saved and restored during evaluation of the operator and operands. Note that this is the only place in the compiler where a target other than `val` is specified.

The required code is generated by `compile-application`. This recursively compiles the operator, to produce code that puts the procedure to be applied into `proc`, and compiles the operands, to produce code that evaluates the individual operands of the application. The instruction sequences for the operands are combined (by `construct-arglist`) with code that constructs the list of arguments in `argl`, and the resulting argument-list code is combined with the procedure code and the code that performs the procedure call (produced by `compile-procedure-call`). In appending the code sequences, the `env` register must be preserved around the evaluation of the operator (since evaluating the operator might modify `env`, which will be needed to evaluate the operands), and the `proc` register must be preserved around the construction of the argument list (since evaluating the operands might modify `proc`, which will be

needed for the actual procedure application). `Continue` must also be preserved throughout, since it is needed for the linkage in the procedure call.

```
(define (compile-application exp target linkage)
  (let ((proc-code (compile (operator exp) 'proc 'next))
        (operand-codes
         (map (lambda (operand) (compile operand 'val 'next))
              (operands exp))))
    (preserving '(env continue)
     proc-code
     (preserving '(proc continue)
      (construct-arglist operand-codes)
      (compile-procedure-call target linkage)))))
```

The code to construct the argument list will evaluate each operand into `val` and then `cons` that value onto the argument list being accumulated in `argl`. Since we `cons` the arguments onto `argl` in sequence, we must start with the last argument and end with the first, so that the arguments will appear in order from first to last in the resulting list. Rather than waste an instruction by initializing `argl` to the empty list to set up for this sequence of evaluations, we make the first code sequence construct the initial `argl`. The general form of the argument-list construction is thus as follows:

```
⟨compilation of last operand, targeted to val⟩
(assign argl (op list) (reg val))
⟨compilation of next operand, targeted to val⟩
(assign argl (op cons) (reg val) (reg argl))
...
⟨compilation of first operand, targeted to val⟩
(assign argl (op cons) (reg val) (reg argl))
```

`Argl` must be preserved around each operand evaluation except the first (so that arguments accumulated so far won't be lost), and `env` must be preserved around each operand evaluation except the last (for use by subsequent operand evaluations).

Compiling this argument code is a bit tricky, because of the special treatment of the first operand to be evaluated and the need to preserve `argl` and `env` in different places. The `construct-arglist` procedure takes as arguments the code that evaluates the individual operands. If there are no operands at all, it simply emits the instruction

```
(assign argl (const ()))
```

Otherwise, `construct-arglist` creates code that initializes `argl` with the last argument, and appends code that evaluates the rest of the arguments and adjoins them to `argl` in succession. In order to process the arguments from last to first, we must reverse the list of operand code sequences from the order supplied by `compile-application`.

```
(define (construct-arglist operand-codes)
  (let ((operand-codes (reverse operand-codes)))
    (if (null? operand-codes)
        (make-instruction-sequence '() '(argl)
         '((assign argl (const ())))
        (let ((code-to-get-last-arg
               (append-instruction-sequences
                (car operand-codes)
                (make-instruction-sequence '(val) '(argl)
                 '((assign argl (op list) (reg val)))))))
          (if (null? (cdr operand-codes))
              code-to-get-last-arg
              (preserving '(env)
               code-to-get-last-arg
               (code-to-get-rest-args
                (cdr operand-codes)))))))))

(define (code-to-get-rest-args operand-codes)
  (let ((code-for-next-arg
         (preserving '(argl)
          (car operand-codes)
          (make-instruction-sequence '(val argl) '(argl)
           '((assign argl
              (op cons) (reg val) (reg argl)))))))
    (if (null? (cdr operand-codes))
        code-for-next-arg
        (preserving '(env)
         code-for-next-arg
         (code-to-get-rest-args (cdr operand-codes))))))
```

Applying procedures

After evaluating the elements of a combination, the compiled code must apply the procedure in `proc` to the arguments in `argl`. The code performs essentially the same dispatch as the `apply` procedure in the metacircular evaluator of section 4.1.1 or the `apply-dispatch` entry point in the explicit-control evaluator of section 5.4.1. It checks whether the procedure to be applied is a primitive procedure or a compiled procedure. For a primitive procedure, it uses `apply-primitive-procedure`; we will see shortly how it handles compiled procedures. The procedure-application code has the following form:

```
 (test (op primitive-procedure?) (reg proc))
 (branch (label primitive-branch))
compiled-branch
 ⟨code to apply compiled procedure with given target and appropriate linkage⟩
primitive-branch
 (assign ⟨target⟩
         (op apply-primitive-procedure)
         (reg proc)
         (reg argl))
 ⟨linkage⟩
after-call
```

Observe that the compiled branch must skip around the primitive branch. Therefore, if the linkage for the original procedure call was `next`, the compound branch must use a linkage that jumps to a label that is inserted after the primitive branch. (This is similar to the linkage used for the true branch in `compile-if`.)

```
(define (compile-procedure-call target linkage)
  (let ((primitive-branch (make-label 'primitive-branch))
        (compiled-branch (make-label 'compiled-branch))
        (after-call (make-label 'after-call)))
    (let ((compiled-linkage
           (if (eq? linkage 'next) after-call linkage)))
      (append-instruction-sequences
       (make-instruction-sequence '(proc) '()
        `((test (op primitive-procedure?) (reg proc))
          (branch (label ,primitive-branch))))
       (parallel-instruction-sequences
        (append-instruction-sequences
         compiled-branch
         (compile-proc-appl target compiled-linkage))
        (append-instruction-sequences
         primitive-branch
         (end-with-linkage linkage
          (make-instruction-sequence '(proc argl)
                                     (list target)
           `((assign ,target
                     (op apply-primitive-procedure)
                     (reg proc)
                     (reg argl)))))))
       after-call))))
```

The primitive and compound branches, like the true and false branches in `compile-if`, are appended using `parallel-instruction-sequences` rather than the ordinary `append-instruction-sequences`, because they will not be executed sequentially.

Applying compiled procedures

The code that handles procedure application is the most subtle part of the compiler, even though the instruction sequences it generates are very short. A compiled procedure (as constructed by `compile-lambda`) has an entry point, which is a label that designates where the code for the procedure starts. The code at this entry point computes a result in `val` and returns by executing the instruction `(goto (reg continue))`. Thus, we might expect the code for a compiled-procedure application (to be generated by `compile-proc-appl`) with a given target and linkage to look like this if the linkage is a label

```
 (assign continue (label proc-return))
 (assign val (op compiled-procedure-entry) (reg proc))
 (goto (reg val))
proc-return
 (assign ⟨target⟩ (reg val))   ; included if target is not val
 (goto (label ⟨linkage⟩))      ; linkage code
```

or like this if the linkage is `return`.

```
 (save continue)
 (assign continue (label proc-return))
 (assign val (op compiled-procedure-entry) (reg proc))
 (goto (reg val))
proc-return
 (assign ⟨target⟩ (reg val))   ; included if target is not val
 (restore continue)
 (goto (reg continue))         ; linkage code
```

This code sets up `continue` so that the procedure will return to a label `proc-return` and jumps to the procedure's entry point. The code at `proc-return` transfers the procedure's result from `val` to the target register (if necessary) and then jumps to the location specified by the linkage. (The linkage is always `return` or a label, because `compile-procedure-call` replaces a `next` linkage for the compound-procedure branch by an `after-call` label.)

In fact, if the target is not `val`, that is exactly the code our compiler will generate.[39] Usually, however, the target is `val` (the only time the compiler specifies a different register is when targeting the evaluation of an operator to `proc`), so the procedure result is put directly into the target register and there is no need to return to a special location that copies it. Instead, we simplify the code by setting up `continue` so that

[39] Actually, we signal an error when the target is not `val` and the linkage is `return`, since the only place we request `return` linkages is in compiling procedures, and our convention is that procedures return their values in `val`.

the procedure will "return" directly to the place specified by the caller's linkage:

```
⟨set up continue for linkage⟩
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

If the linkage is a label, we set up `continue` so that the procedure will return to that label. (That is, the `(goto (reg continue))` the procedure ends with becomes equivalent to the `(goto (label ⟨linkage⟩))` at `proc-return` above.)

```
(assign continue (label ⟨linkage⟩))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

If the linkage is `return`, we don't need to set up `continue` at all: It already holds the desired location. (That is, the `(goto (reg continue))` the procedure ends with goes directly to the place where the `(goto (reg continue))` at `proc-return` would have gone.)

```
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

With this implementation of the `return` linkage, the compiler generates tail-recursive code. Calling a procedure as the final step in a procedure body does a direct transfer, without saving any information on the stack.

Suppose instead that we had handled the case of a procedure call with a linkage of `return` and a target of `val` as shown above for a non-`val` target. This would destroy tail recursion. Our system would still give the same value for any expression. But each time we called a procedure, we would save `continue` and return after the call to undo the (useless) save. These extra saves would accumulate during a nest of procedure calls.[40]

[40]Making a compiler generate tail-recursive code might seem like a straightforward idea. But most compilers for common languages, including C and Pascal, do not do this, and therefore these languages cannot represent iterative processes in terms of procedure call alone. The difficulty with tail recursion in these languages is that their implementations use the stack to store procedure arguments and local variables as well as return addresses. The Scheme implementations described in this book store arguments and variables in memory to be garbage-collected. The reason for using the stack for variables and arguments is that it avoids the need for garbage collection in languages that would not otherwise require it, and is generally believed to be more efficient. Sophisticated Lisp compilers can, in fact, use the stack for arguments without destroying tail recursion. (See Hanson 1990 for a description.) There is also some debate about whether stack allocation is actually more efficient than garbage collection in the first place, but the details seem to hinge on fine points of computer architecture. (See Appel 1987 and Miller and Rozas 1994 for opposing views on this issue.)

`Compile-proc-appl` generates the above procedure-application code by considering four cases, depending on whether the target for the call is `val` and whether the linkage is `return`. Observe that the instruction sequences are declared to modify all the registers, since executing the procedure body can change the registers in arbitrary ways.[41] Also note that the code sequence for the case with target `val` and linkage `return` is declared to need `continue`: Even though `continue` is not explicitly used in the two-instruction sequence, we must be sure that `continue` will have the correct value when we enter the compiled procedure.

```
(define (compile-proc-appl target linkage)
  (cond ((and (eq? target 'val) (not (eq? linkage 'return)))
         (make-instruction-sequence '(proc) all-regs
           `((assign continue (label ,linkage))
             (assign val (op compiled-procedure-entry)
                         (reg proc))
             (goto (reg val)))))
        ((and (not (eq? target 'val))
              (not (eq? linkage 'return)))
         (let ((proc-return (make-label 'proc-return)))
           (make-instruction-sequence '(proc) all-regs
            `((assign continue (label ,proc-return))
              (assign val (op compiled-procedure-entry)
                          (reg proc))
              (goto (reg val))
              ,proc-return
              (assign ,target (reg val))
              (goto (label ,linkage))))))
        ((and (eq? target 'val) (eq? linkage 'return))
         (make-instruction-sequence '(proc continue) all-regs
          '((assign val (op compiled-procedure-entry)
                        (reg proc))
            (goto (reg val)))))
        ((and (not (eq? target 'val)) (eq? linkage 'return))
         (error "return linkage, target not val -- COMPILE"
                target))))
```

5.5.4 Combining Instruction Sequences

This section describes the details on how instruction sequences are represented and combined. Recall from section 5.5.1 that an instruction sequence is represented as a list of the registers needed, the registers

[41] The variable `all-regs` is bound to the list of names of all the registers:

```
(define all-regs '(env proc val argl continue))
```

modified, and the actual instructions. We will also consider a label (symbol) to be a degenerate case of an instruction sequence, which doesn't need or modify any registers. So to determine the registers needed and modified by instruction sequences we use the selectors

```
(define (registers-needed s)
  (if (symbol? s) '() (car s)))

(define (registers-modified s)
  (if (symbol? s) '() (cadr s)))

(define (statements s)
  (if (symbol? s) (list s) (caddr s)))
```

and to determine whether a given sequence needs or modifies a given register we use the predicates

```
(define (needs-register? seq reg)
  (memq reg (registers-needed seq)))

(define (modifies-register? seq reg)
  (memq reg (registers-modified seq)))
```

In terms of these predicates and selectors, we can implement the various instruction sequence combiners used throughout the compiler.

The basic combiner is `append-instruction-sequences`. This takes as arguments an arbitrary number of instruction sequences that are to be executed sequentially and returns an instruction sequence whose statements are the statements of all the sequences appended together. The subtle point is to determine the registers that are needed and modified by the resulting sequence. It modifies those registers that are modified by any of the sequences; it needs those registers that must be initialized before the first sequence can be run (the registers needed by the first sequence), together with those registers needed by any of the other sequences that are not initialized (modified) by sequences preceding it.

The sequences are appended two at a time by `append-2-sequences`. This takes two instruction sequences `seq1` and `seq2` and returns the instruction sequence whose statements are the statements of `seq1` followed by the statements of `seq2`, whose modified registers are those registers that are modified by either `seq1` or `seq2`, and whose needed registers are the registers needed by `seq1` together with those registers needed by `seq2` that are not modified by `seq1`. (In terms of set operations, the new set of needed registers is the union of the set of registers needed by `seq1` with the set difference of the registers needed by `seq2` and the registers

modified by seq1.) Thus, append-instruction-sequences is implemented as follows:

```
(define (append-instruction-sequences . seqs)
  (define (append-2-sequences seq1 seq2)
    (make-instruction-sequence
     (list-union (registers-needed seq1)
                 (list-difference (registers-needed seq2)
                                  (registers-modified seq1)))
     (list-union (registers-modified seq1)
                 (registers-modified seq2))
     (append (statements seq1) (statements seq2))))
  (define (append-seq-list seqs)
    (if (null? seqs)
        (empty-instruction-sequence)
        (append-2-sequences (car seqs)
                            (append-seq-list (cdr seqs)))))
  (append-seq-list seqs))
```

This procedure uses some simple operations for manipulating sets represented as lists, similar to the (unordered) set representation described in section 2.3.3:

```
(define (list-union s1 s2)
  (cond ((null? s1) s2)
        ((memq (car s1) s2) (list-union (cdr s1) s2))
        (else (cons (car s1) (list-union (cdr s1) s2)))))

(define (list-difference s1 s2)
  (cond ((null? s1) '())
        ((memq (car s1) s2) (list-difference (cdr s1) s2))
        (else (cons (car s1)
                    (list-difference (cdr s1) s2)))))
```

Preserving, the second major instruction sequence combiner, takes a list of registers regs and two instruction sequences seq1 and seq2 that are to be executed sequentially. It returns an instruction sequence whose statements are the statements of seq1 followed by the statements of seq2, with appropriate save and restore instructions around seq1 to protect the registers in regs that are modified by seq1 but needed by seq2. To accomplish this, preserving first creates a sequence that has the required saves followed by the statements of seq1 followed by the required restores. This sequence needs the registers being saved and restored in addition to the registers needed by seq1, and modifies the registers modified by seq1 except for the ones being saved and restored.

This augmented sequence and seq2 are then appended in the usual way. The following procedure implements this strategy recursively, walking down the list of registers to be preserved:[42]

```
(define (preserving regs seq1 seq2)
  (if (null? regs)
      (append-instruction-sequences seq1 seq2)
      (let ((first-reg (car regs)))
        (if (and (needs-register? seq2 first-reg)
                 (modifies-register? seq1 first-reg))
            (preserving (cdr regs)
             (make-instruction-sequence
              (list-union (list first-reg)
                          (registers-needed seq1))
              (list-difference (registers-modified seq1)
                               (list first-reg))
              (append `((save ,first-reg))
                      (statements seq1)
                      `((restore ,first-reg))))
             seq2)
            (preserving (cdr regs) seq1 seq2)))))
```

Another sequence combiner, tack-on-instruction-sequence, is used by compile-lambda to append a procedure body to another sequence. Because the procedure body is not "in line" to be executed as part of the combined sequence, its register use has no impact on the register use of the sequence in which it is embedded. We thus ignore the procedure body's sets of needed and modified registers when we tack it onto the other sequence.

```
(define (tack-on-instruction-sequence seq body-seq)
  (make-instruction-sequence
   (registers-needed seq)
   (registers-modified seq)
   (append (statements seq) (statements body-seq))))
```

Compile-if and compile-procedure-call use a special combiner called parallel-instruction-sequences to append the two alternative branches that follow a test. The two branches will never be executed sequentially; for any particular evaluation of the test, one branch or the other will be entered. Because of this, the registers needed by the sec-

[42]Note that preserving calls append with three arguments. Though the definition of append shown in this book accepts only two arguments, Scheme standardly provides an append procedure that takes an arbitrary number of arguments.

ond branch are still needed by the combined sequence, even if these are modified by the first branch.

```
(define (parallel-instruction-sequences seq1 seq2)
  (make-instruction-sequence
   (list-union (registers-needed seq1)
               (registers-needed seq2))
   (list-union (registers-modified seq1)
               (registers-modified seq2))
   (append (statements seq1) (statements seq2))))
```

5.5.5 An Example of Compiled Code

Now that we have seen all the elements of the compiler, let us examine an example of compiled code to see how things fit together. We will compile the definition of a recursive `factorial` procedure by calling `compile`:

```
(compile
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n)))
 'val
 'next)
```

We have specified that the value of the `define` expression should be placed in the `val` register. We don't care what the compiled code does after executing the `define`, so our choice of `next` as the linkage descriptor is arbitrary.

`Compile` determines that the expression is a definition, so it calls `compile-definition` to compile code to compute the value to be assigned (targeted to `val`), followed by code to install the definition, followed by code to put the value of the `define` (which is the symbol `ok`) into the target register, followed finally by the linkage code. `Env` is preserved around the computation of the value, because it is needed in order to install the definition. Because the linkage is `next`, there is no linkage code in this case. The skeleton of the compiled code is thus

```
⟨save env if modified by code to compute value⟩
⟨compilation of definition value, target val, linkage next⟩
⟨restore env if saved above⟩
(perform (op define-variable!)
         (const factorial)
         (reg val)
         (reg env))
(assign val (const ok))
```

The expression that is to be compiled to produce the value for the variable `factorial` is a `lambda` expression whose value is the procedure that computes factorials. `Compile` handles this by calling `compile-lambda`, which compiles the procedure body, labels it as a new entry point, and generates the instruction that will combine the procedure body at the new entry point with the run-time environment and assign the result to `val`. The sequence then skips around the compiled procedure code, which is inserted at this point. The procedure code itself begins by extending the procedure's definition environment by a frame that binds the formal parameter `n` to the procedure argument. Then comes the actual procedure body. Since this code for the value of the variable doesn't modify the `env` register, the optional `save` and `restore` shown above aren't generated. (The procedure code at `entry2` isn't executed at this point, so its use of `env` is irrelevant.) Therefore, the skeleton for the compiled code becomes

```
  (assign val (op make-compiled-procedure)
              (label entry2)
              (reg env))
  (goto (label after-lambda1))
entry2
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env (op extend-environment)
              (const (n))
              (reg argl)
              (reg env))
  ⟨compilation of procedure body⟩
after-lambda1
  (perform (op define-variable!)
           (const factorial)
           (reg val)
           (reg env))
  (assign val (const ok))
```

A procedure body is always compiled (by `compile-lambda-body`) as a sequence with target `val` and linkage `return`. The sequence in this case consists of a single `if` expression:

```
(if (= n 1)
    1
    (* (factorial (- n 1)) n))
```

`Compile-if` generates code that first computes the predicate (targeted to `val`), then checks the result and branches around the true branch if the predicate is false. `Env` and `continue` are preserved around the predicate

code, since they may be needed for the rest of the `if` expression. Since the `if` expression is the final expression (and only expression) in the sequence making up the procedure body, its target is `val` and its linkage is `return`, so the true and false branches are both compiled with target `val` and linkage `return`. (That is, the value of the conditional, which is the value computed by either of its branches, is the value of the procedure.)

```
  ⟨save continue, env if modified by predicate and needed by branches⟩
  ⟨compilation of predicate, target val, linkage next⟩
  ⟨restore continue, env if saved above⟩
  (test (op false?) (reg val))
  (branch (label false-branch4))
true-branch5
  ⟨compilation of true branch, target val, linkage return⟩
false-branch4
  ⟨compilation of false branch, target val, linkage return⟩
after-if3
```

The predicate `(= n 1)` is a procedure call. This looks up the operator (the symbol `=`) and places this value in `proc`. It then assembles the arguments `1` and the value of `n` into `argl`. Then it tests whether `proc` contains a primitive or a compound procedure, and dispatches to a primitive branch or a compound branch accordingly. Both branches resume at the `after-call` label. The requirements to preserve registers around the evaluation of the operator and operands don't result in any saving of registers, because in this case those evaluations don't modify the registers in question.

```
  (assign proc
          (op lookup-variable-value) (const =) (reg env))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch17))
compiled-branch16
  (assign continue (label after-call15))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch17
  (assign val (op apply-primitive-procedure)
              (reg proc)
              (reg argl))
after-call15
```

The true branch, which is the constant 1, compiles (with target `val` and linkage `return`) to

```
(assign val (const 1))
(goto (reg continue))
```

The code for the false branch is another a procedure call, where the procedure is the value of the symbol `*`, and the arguments are `n` and the result of another procedure call (a call to `factorial`). Each of these calls sets up `proc` and `argl` and its own primitive and compound branches. Figure 5.17 shows the complete compilation of the definition of the `factorial` procedure. Notice that the possible `save` and `restore` of `continue` and `env` around the predicate, shown above, are in fact generated, because these registers are modified by the procedure call in the predicate and needed for the procedure call and the `return` linkage in the branches.

Exercise 5.33

Consider the following definition of a factorial procedure, which is slightly different from the one given above:

```
(define (factorial-alt n)
  (if (= n 1)
      1
      (* n (factorial-alt (- n 1)))))
```

Compile this procedure and compare the resulting code with that produced for `factorial`. Explain any differences you find. Does either program execute more efficiently than the other?

Exercise 5.34

Compile the iterative factorial procedure

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Annotate the resulting code, showing the essential difference between the code for iterative and recursive versions of `factorial` that makes one process build up stack space and the other run in constant stack space.

Exercise 5.35

What expression was compiled to produce the code shown in figure 5.18?

Exercise 5.36

What order of evaluation does our compiler produce for operands of a combination? Is it left-to-right, right-to-left, or some other order? Where in the compiler is this order determined? Modify the compiler so that it produces some other order of evaluation. (See the discussion of order of evaluation for the explicit-control evaluator in section 5.4.1.) How does changing the order of operand evaluation affect the efficiency of the code that constructs the argument list?

Exercise 5.37

One way to understand the compiler's `preserving` mechanism for optimizing stack usage is to see what extra operations would be generated if we did not use this idea. Modify `preserving` so that it always generates the `save` and `restore` operations. Compile some simple expressions and identify the unnecessary stack operations that are generated. Compare the code to that generated with the `preserving` mechanism intact.

Exercise 5.38

Our compiler is clever about avoiding unnecessary stack operations, but it is not clever at all when it comes to compiling calls to the primitive procedures of the language in terms of the primitive operations supplied by the machine. For example, consider how much code is compiled to compute `(+ a 1)`: The code sets up an argument list in `argl`, puts the primitive addition procedure (which it finds by looking up the symbol + in the environment) into `proc`, and tests whether the procedure is primitive or compound. The compiler always generates code to perform the test, as well as code for primitive and compound branches (only one of which will be executed). We have not shown the part of the controller that implements primitives, but we presume that these instructions make use of primitive arithmetic operations in the machine's data paths. Consider how much less code would be generated if the compiler could *open-code* primitives—that is, if it could generate code to directly use these primitive machine operations. The expression `(+ a 1)` might be compiled into something as simple as [43]

```
(assign val (op lookup-variable-value) (const a) (reg env))
(assign val (op +) (reg val) (const 1))
```

In this exercise we will extend our compiler to support open coding of selected primitives. Special-purpose code will be generated for calls to these primitive

[43] We have used the same symbol + here to denote both the source-language procedure and the machine operation. In general there will not be a one-to-one correspondence between primitives of the source language and primitives of the machine.

```
;; construct the procedure and skip over code for the procedure body
  (assign val
          (op make-compiled-procedure) (label entry2) (reg env))
  (goto (label after-lambda1))

entry2      ; calls to factorial will enter here
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env
          (op extend-environment) (const (n)) (reg argl) (reg env))
;; begin actual procedure body
  (save continue)
  (save env)

;; compute (= n 1)
  (assign proc (op lookup-variable-value) (const =) (reg env))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch17))
compiled-branch16
  (assign continue (label after-call15))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch17
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call15    ; val now contains result of (= n 1)
  (restore env)
  (restore continue)
  (test (op false?) (reg val))
  (branch (label false-branch4))
true-branch5   ; return 1
  (assign val (const 1))
  (goto (reg continue))

false-branch4
;; compute and return (* (factorial (- n 1)) n)
  (assign proc (op lookup-variable-value) (const *) (reg env))
  (save continue)
  (save proc)    ; save * procedure
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op list) (reg val))
  (save argl)    ; save partial argument list for *

;; compute (factorial (- n 1)), which is the other argument for *
  (assign proc
          (op lookup-variable-value) (const factorial) (reg env))
  (save proc)   ; save factorial procedure
```

Figure 5.17 Compilation of the definition of the `factorial` procedure (continued on next page).

```
;; compute (- n 1), which is the argument for factorial
  (assign proc (op lookup-variable-value) (const -) (reg env))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch8))
compiled-branch7
  (assign continue (label after-call6))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch8
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call6    ; val now contains result of (- n 1)
  (assign argl (op list) (reg val))
  (restore proc)  ; restore factorial
;; apply factorial
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch11))
compiled-branch10
  (assign continue (label after-call9))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch11
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call9       ; val now contains result of (factorial (- n 1))
  (restore argl)  ; restore partial argument list for *
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)  ; restore *
  (restore continue)
;; apply * and return its value
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch14))
compiled-branch13
;; note that a compound procedure here is called tail-recursively
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch14
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
after-call12
after-if3
after-lambda1
;; assign the procedure to the variable factorial
  (perform
   (op define-variable!) (const factorial) (reg val) (reg env))
  (assign val (const ok))
```

Figure 5.17 (continued)

```
  (assign val (op make-compiled-procedure) (label entry16)
                                           (reg env))
  (goto (label after-lambda15))
entry16
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env
          (op extend-environment) (const (x)) (reg argl) (reg env))
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (save continue)
  (save proc)
  (save env)
  (assign proc (op lookup-variable-value) (const g) (reg env))
  (save proc)
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (assign val (const 2))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const x) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch19))
compiled-branch18
  (assign continue (label after-call17))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch19
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call17
  (assign argl (op list) (reg val))
  (restore proc)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch22))
compiled-branch21
  (assign continue (label after-call20))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch22
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
```

Figure 5.18 An example of compiler output (continued on next page). See exercise 5.35.

procedures instead of the general procedure-application code. In order to support this, we will augment our machine with special argument registers `arg1` and `arg2`. The primitive arithmetic operations of the machine will take their inputs from `arg1` and `arg2`. The results may be put into `val`, `arg1`, or `arg2`.

The compiler must be able to recognize the application of an open-coded primitive in the source program. We will augment the dispatch in the `compile` procedure to recognize the names of these primitives in addition to the reserved

```
after-call20
  (assign argl (op list) (reg val))
  (restore env)
  (assign val (op lookup-variable-value) (const x) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)
  (restore continue)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch25))
compiled-branch24
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch25
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
after-call23
after-lambda15
  (perform (op define-variable!) (const f) (reg val) (reg env))
  (assign val (const ok))
```

Figure 5.18 (continued)

words (the special forms) it currently recognizes.[44] For each special form our compiler has a code generator. In this exercise we will construct a family of code generators for the open-coded primitives.

a. The open-coded primitives, unlike the special forms, all need their operands evaluated. Write a code generator `spread-arguments` for use by all the open-coding code generators. `Spread-arguments` should take an operand list and compile the given operands targeted to successive argument registers. Note that an operand may contain a call to an open-coded primitive, so argument registers will have to be preserved during operand evaluation.

b. For each of the primitive procedures `=`, `*`, `-`, and `+`, write a code generator that takes a combination with that operator, together with a target and a linkage descriptor, and produces code to spread the arguments into the registers and then perform the operation targeted to the given target with the given linkage. You need only handle expressions with two operands. Make `compile` dispatch to these code generators.

c. Try your new compiler on the `factorial` example. Compare the resulting code with the result produced without open coding.

d. Extend your code generators for `+` and `*` so that they can handle expressions with arbitrary numbers of operands. An expression with more than two operands will have to be compiled into a sequence of operations, each with only two inputs.

---

[44] Making the primitives into reserved words is in general a bad idea, since a user cannot then rebind these names to different procedures. Moreover, if we add reserved words to a compiler that is in use, existing programs that define procedures with these names will stop working. See exercise 5.44 for ideas on how to avoid this problem.

5.5.6 Lexical Addressing

One of the most common optimizations performed by compilers is the optimization of variable lookup. Our compiler, as we have implemented it so far, generates code that uses the `lookup-variable-value` operation of the evaluator machine. This searches for a variable by comparing it with each variable that is currently bound, working frame by frame outward through the run-time environment. This search can be expensive if the frames are deeply nested or if there are many variables. For example, consider the problem of looking up the value of `x` while evaluating the expression `(* x y z)` in an application of the procedure that is returned by

```
(let ((x 3) (y 4))
  (lambda (a b c d e)
    (let ((y (* a b x))
          (z (+ c d x)))
      (* x y z))))
```

Since a `let` expression is just syntactic sugar for a `lambda` combination, this expression is equivalent to

```
((lambda (x y)
   (lambda (a b c d e)
     ((lambda (y z) (* x y z))
      (* a b x)
      (+ c d x))))
 3
 4)
```

Each time `lookup-variable-value` searches for `x`, it must determine that the symbol `x` is not `eq?` to `y` or `z` (in the first frame), nor to `a`, `b`, `c`, `d`, or `e` (in the second frame). We will assume, for the moment, that our programs do not use `define`—that variables are bound only with `lambda`. Because our language is lexically scoped, the run-time environment for any expression will have a structure that parallels the lexical structure of the program in which the expression appears.[45] Thus, the compiler can know, when it analyzes the above expression, that each time the procedure is applied the variable `x` in `(* x y z)` will be found two frames out from the current frame and will be the first variable in that frame.

We can exploit this fact by inventing a new kind of variable-lookup operation, `lexical-address-lookup`, that takes as arguments an en-

[45] This is not true if we allow internal definitions, unless we scan them out. See exercise 5.43.

vironment and a *lexical address* that consists of two numbers: a *frame number*, which specifies how many frames to pass over, and a *displacement number*, which specifies how many variables to pass over in that frame. `Lexical-address-lookup` will produce the value of the variable stored at that lexical address relative to the current environment. If we add the `lexical-address-lookup` operation to our machine, we can make the compiler generate code that references variables using this operation, rather than `lookup-variable-value`. Similarly, our compiled code can use a new `lexical-address-set!` operation instead of `set-variable-value!`.

In order to generate such code, the compiler must be able to determine the lexical address of a variable it is about to compile a reference to. The lexical address of a variable in a program depends on where one is in the code. For example, in the following program, the address of `x` in expression ⟨*e1*⟩ is (2,0)—two frames back and the first variable in the frame. At that point `y` is at address (0,0) and `c` is at address (1,2). In expression ⟨*e2*⟩, `x` is at (1,0), `y` is at (1,1), and `c` is at (0,2).

```
((lambda (x y)
   (lambda (a b c d e)
     ((lambda (y z) ⟨e1⟩)
      ⟨e2⟩
      (+ c d x))))
 3
 4)
```

One way for the compiler to produce code that uses lexical addressing is to maintain a data structure called a *compile-time environment*. This keeps track of which variables will be at which positions in which frames in the run-time environment when a particular variable-access operation is executed. The compile-time environment is a list of frames, each containing a list of variables. (There will of course be no values bound to the variables, since values are not computed at compile time.) The compile-time environment becomes an additional argument to `compile` and is passed along to each code generator. The top-level call to `compile` uses an empty compile-time environment. When a `lambda` body is compiled, `compile-lambda-body` extends the compile-time environment by a frame containing the procedure's parameters, so that the sequence making up the body is compiled with that extended environment. At each point in the compilation, `compile-variable` and `compile-assignment` use the compile-time environment in order to generate the appropriate lexical addresses.

Exercises 5.39 through 5.43 describe how to complete this sketch of the lexical-addressing strategy in order to incorporate lexical lookup into the compiler. Exercise 5.44 describes another use for the compile-time environment.

Exercise 5.39

Write a procedure `lexical-address-lookup` that implements the new lookup operation. It should take two arguments—a lexical address and a run-time environment—and return the value of the variable stored at the specified lexical address. `Lexical-address-lookup` should signal an error if the value of the variable is the symbol `*unassigned*`.[46] Also write a procedure `lexical-address-set!` that implements the operation that changes the value of the variable at a specified lexical address.

Exercise 5.40

Modify the compiler to maintain the compile-time environment as described above. That is, add a compile-time-environment argument to `compile` and the various code generators, and extend it in `compile-lambda-body`.

Exercise 5.41

Write a procedure `find-variable` that takes as arguments a variable and a compile-time environment and returns the lexical address of the variable with respect to that environment. For example, in the program fragment that is shown above, the compile-time environment during the compilation of expression ⟨ *e1* ⟩ is `((y z) (a b c d e) (x y))`. `Find-variable` should produce

```
(find-variable 'c '((y z) (a b c d e) (x y)))
(1 2)

(find-variable 'x '((y z) (a b c d e) (x y)))
(2 0)

(find-variable 'w '((y z) (a b c d e) (x y)))
not-found
```

Exercise 5.42

Using `find-variable` from exercise 5.41, rewrite `compile-variable` and `compile-assignment` to output lexical-address instructions. In cases where `find-variable` returns `not-found` (that is, where the variable is not in the compile-time environment), you should have the code generators use the evaluator operations, as before, to search for the binding. (The only place a variable that is not found at compile time can be is in the global environment, which

[46]This is the modification to variable lookup required if we implement the scanning method to eliminate internal definitions (exercise 5.43). We will need to eliminate these definitions in order for lexical addressing to work.

is part of the run-time environment but is not part of the compile-time environment.[47] Thus, if you wish, you may have the evaluator operations look directly in the global environment, which can be obtained with the operation `(op get-global-environment)`, instead of having them search the whole run-time environment found in `env`.) Test the modified compiler on a few simple cases, such as the nested `lambda` combination at the beginning of this section.

Exercise 5.43

We argued in section 4.1.6 that internal definitions for block structure should not be considered "real" `define`s. Rather, a procedure body should be interpreted as if the internal variables being defined were installed as ordinary `lambda` variables initialized to their correct values using `set!`. Section 4.1.6 and exercise 4.16 showed how to modify the metacircular interpreter to accomplish this by scanning out internal definitions. Modify the compiler to perform the same transformation before it compiles a procedure body.

Exercise 5.44

In this section we have focused on the use of the compile-time environment to produce lexical addresses. But there are other uses for compile-time environments. For instance, in exercise 5.38 we increased the efficiency of compiled code by open-coding primitive procedures. Our implementation treated the names of open-coded procedures as reserved words. If a program were to rebind such a name, the mechanism described in exercise 5.38 would still open-code it as a primitive, ignoring the new binding. For example, consider the procedure

```
(lambda (+ * a b x y)
  (+ (* a x) (* b y)))
```

which computes a linear combination of `x` and `y`. We might call it with arguments `+matrix`, `*matrix`, and four matrices, but the open-coding compiler would still open-code the `+` and the `*` in `(+ (* a x) (* b y))` as primitive `+` and `*`. Modify the open-coding compiler to consult the compile-time environment in order to compile the correct code for expressions involving the names of primitive procedures. (The code will work correctly as long as the program does not `define` or `set!` these names.)

5.5.7 Interfacing Compiled Code to the Evaluator

We have not yet explained how to load compiled code into the evaluator machine or how to run it. We will assume that the explicit-control-

[47] Lexical addresses cannot be used to access variables in the global environment, because these names can be defined and redefined interactively at any time. With internal definitions scanned out, as in exercise 5.43, the only definitions the compiler sees are those at top level, which act on the global environment. Compilation of a definition does not cause the defined name to be entered in the compile-time environment.

evaluator machine has been defined as in section 5.4.4, with the additional operations specified in footnote 38. We will implement a procedure `compile-and-go` that compiles a Scheme expression, loads the resulting object code into the evaluator machine, and causes the machine to run the code in the evaluator global environment, print the result, and enter the evaluator's driver loop. We will also modify the evaluator so that interpreted expressions can call compiled procedures as well as interpreted ones. We can then put a compiled procedure into the machine and use the evaluator to call it:

```
(compile-and-go
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n))))
 ;;; EC-Eval value:
ok

 ;;; EC-Eval input:
(factorial 5)
;;; EC-Eval value:
120
```

To allow the evaluator to handle compiled procedures (for example, to evaluate the call to `factorial` above), we need to change the code at `apply-dispatch` (section 5.4.1) so that it recognizes compiled procedures (as distinct from compound or primitive procedures) and transfers control directly to the entry point of the compiled code:[48]

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (test (op compiled-procedure?) (reg proc))
  (branch (label compiled-apply))
  (goto (label unknown-procedure-type))

compiled-apply
  (restore continue)
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
```

[48]Of course, compiled procedures as well as interpreted procedures are compound (non-primitive). For compatibility with the terminology used in the explicit-control evaluator, in this section we will use "compound" to mean interpreted (as opposed to compiled).

Note the restore of `continue` at `compiled-apply`. Recall that the evaluator was arranged so that at `apply-dispatch`, the continuation would be at the top of the stack. The compiled code entry point, on the other hand, expects the continuation to be in `continue`, so `continue` must be restored before the compiled code is executed.

To enable us to run some compiled code when we start the evaluator machine, we add a `branch` instruction at the beginning of the evaluator machine, which causes the machine to go to a new entry point if the `flag` register is set.[49]

```
  (branch (label external-entry))        ; branches if flag is set
read-eval-print-loop
  (perform (op initialize-stack))
  ...
```

`External-entry` assumes that the machine is started with `val` containing the location of an instruction sequence that puts a result into `val` and ends with `(goto (reg continue))`. Starting at this entry point jumps to the location designated by `val`, but first assigns `continue` so that execution will return to `print-result`, which prints the value in `val` and then goes to the beginning of the evaluator's read-eval-print loop.[50]

```
external-entry
  (perform (op initialize-stack))
  (assign env (op get-global-environment))
  (assign continue (label print-result))
  (goto (reg val))
```

---

[49] Now that the evaluator machine starts with a `branch`, we must always initialize the `flag` register before starting the evaluator machine. To start the machine at its ordinary read-eval-print loop, we could use

```
(define (start-eceval)
  (set! the-global-environment (setup-environment))
  (set-register-contents! eceval 'flag false)
  (start eceval))
```

[50] Since a compiled procedure is an object that the system may try to print, we also modify the system print operation `user-print` (from section 4.1.4) so that it will not attempt to print the components of a compiled procedure:

```
(define (user-print object)
  (cond ((compound-procedure? object)
         (display (list 'compound-procedure
                        (procedure-parameters object)
                        (procedure-body object)
                        '<procedure-env>)))
        ((compiled-procedure? object)
         (display '<compiled-procedure>))
        (else (display object))))
```

Now we can use the following procedure to compile a procedure definition, execute the compiled code, and run the read-eval-print loop so we can try the procedure. Because we want the compiled code to return to the location in `continue` with its result in `val`, we compile the expression with a target of `val` and a linkage of `return`. In order to transform the object code produced by the compiler into executable instructions for the evaluator register machine, we use the procedure `assemble` from the register-machine simulator (section 5.2.2). We then initialize the `val` register to point to the list of instructions, set the `flag` so that the evaluator will go to `external-entry`, and start the evaluator.

```
(define (compile-and-go expression)
  (let ((instructions
         (assemble (statements
                    (compile expression 'val 'return))
                   eceval)))
    (set! the-global-environment (setup-environment))
    (set-register-contents! eceval 'val instructions)
    (set-register-contents! eceval 'flag true)
    (start eceval)))
```

If we have set up stack monitoring, as at the end of section 5.4.4, we can examine the stack usage of compiled code:

```
(compile-and-go
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n))))

(total-pushes = 0 maximum-depth = 0)
 ;;; EC-Eval value:
ok

 ;;; EC-Eval input:
(factorial 5)
(total-pushes = 31 maximum-depth = 14)
;;; EC-Eval value:
120
```

Compare this example with the evaluation of `(factorial 5)` using the interpreted version of the same procedure, shown at the end of section 5.4.4. The interpreted version required 144 pushes and a maximum stack depth of 28. This illustrates the optimization that results from our compilation strategy.

Interpretation and compilation

With the programs in this section, we can now experiment with the alternative execution strategies of interpretation and compilation.[51] An interpreter raises the machine to the level of the user program; a compiler lowers the user program to the level of the machine language. We can regard the Scheme language (or any programming language) as a coherent family of abstractions erected on the machine language. Interpreters are good for interactive program development and debugging because the steps of program execution are organized in terms of these abstractions, and are therefore more intelligible to the programmer. Compiled code can execute faster, because the steps of program execution are organized in terms of the machine language, and the compiler is free to make optimizations that cut across the higher-level abstractions.[52]

The alternatives of interpretation and compilation also lead to different strategies for porting languages to new computers. Suppose that we wish to implement Lisp for a new machine. One strategy is to begin with the explicit-control evaluator of section 5.4 and translate its instructions to instructions for the new machine. A different strategy is to begin with the compiler and change the code generators so that they generate code for the new machine. The second strategy allows us to run any Lisp program on the new machine by first compiling it with the compiler running on our original Lisp system, and linking it with a compiled version of the

---

[51] We can do even better by extending the compiler to allow compiled code to call interpreted procedures. See exercise 5.47.

[52] Independent of the strategy of execution, we incur significant overhead if we insist that errors encountered in execution of a user program be detected and signaled, rather than being allowed to kill the system or produce wrong answers. For example, an out-of-bounds array reference can be detected by checking the validity of the reference before performing it. The overhead of checking, however, can be many times the cost of the array reference itself, and a programmer should weigh speed against safety in determining whether such a check is desirable. A good compiler should be able to produce code with such checks, should avoid redundant checks, and should allow programmers to control the extent and type of error checking in the compiled code.

Compilers for popular languages, such as C and C++, put hardly any error-checking operations into running code, so as to make things run as fast as possible. As a result, it falls to programmers to explicitly provide error checking. Unfortunately, people often neglect to do this, even in critical applications where speed is not a constraint. Their programs lead fast and dangerous lives. For example, the notorious "Worm" that paralyzed the Internet in 1988 exploited the UNIX™ operating system's failure to check whether the input buffer has overflowed in the finger daemon. (See Spafford 1989.)

run-time library.[53] Better yet, we can compile the compiler itself, and run this on the new machine to compile other Lisp programs.[54] Or we can compile one of the interpreters of section 4.1 to produce an interpreter that runs on the new machine.

Exercise 5.45

By comparing the stack operations used by compiled code to the stack operations used by the evaluator for the same computation, we can determine the extent to which the compiler optimizes use of the stack, both in speed (reducing the total number of stack operations) and in space (reducing the maximum stack depth). Comparing this optimized stack use to the performance of a special-purpose machine for the same computation gives some indication of the quality of the compiler.

a. Exercise 5.27 asked you to determine, as a function of n, the number of pushes and the maximum stack depth needed by the evaluator to compute $n!$ using the recursive factorial procedure given above. Exercise 5.14 asked you to do the same measurements for the special-purpose factorial machine shown in figure 5.11. Now perform the same analysis using the compiled `factorial` procedure.

Take the ratio of the number of pushes in the compiled version to the number of pushes in the interpreted version, and do the same for the maximum stack depth. Since the number of operations and the stack depth used to compute $n!$ are linear in n, these ratios should approach constants as n becomes large. What are these constants? Similarly, find the ratios of the stack usage in the special-purpose machine to the usage in the interpreted version.

Compare the ratios for special-purpose versus interpreted code to the ratios for compiled versus interpreted code. You should find that the special-purpose machine does much better than the compiled code, since the hand-tailored controller code should be much better than what is produced by our rudimentary general-purpose compiler.

b. Can you suggest improvements to the compiler that would help it generate code that would come closer in performance to the hand-tailored version?

---

[53] Of course, with either the interpretation or the compilation strategy we must also implement for the new machine storage allocation, input and output, and all the various operations that we took as "primitive" in our discussion of the evaluator and compiler. One strategy for minimizing work here is to write as many of these operations as possible in Lisp and then compile them for the new machine. Ultimately, everything reduces to a small kernel (such as garbage collection and the mechanism for applying actual machine primitives) that is hand-coded for the new machine.

[54] This strategy leads to amusing tests of correctness of the compiler, such as checking whether the compilation of a program on the new machine, using the compiled compiler, is identical with the compilation of the program on the original Lisp system. Tracking down the source of differences is fun but often frustrating, because the results are extremely sensitive to minuscule details.

Exercise 5.46

Carry out an analysis like the one in exercise 5.45 to determine the effectiveness of compiling the tree-recursive Fibonacci procedure

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

compared to the effectiveness of using the special-purpose Fibonacci machine of figure 5.12. (For measurement of the interpreted performance, see exercise 5.29.) For Fibonacci, the time resource used is not linear in n; hence the ratios of stack operations will not approach a limiting value that is independent of n.

Exercise 5.47

This section described how to modify the explicit-control evaluator so that interpreted code can call compiled procedures. Show how to modify the compiler so that compiled procedures can call not only primitive procedures and compiled procedures, but interpreted procedures as well. This requires modifying `compile-procedure-call` to handle the case of compound (interpreted) procedures. Be sure to handle all the same `target` and `linkage` combinations as in `compile-proc-appl`. To do the actual procedure application, the code needs to jump to the evaluator's `compound-apply` entry point. This label cannot be directly referenced in object code (since the assembler requires that all labels referenced by the code it is assembling be defined there), so we will add a register called `compapp` to the evaluator machine to hold this entry point, and add an instruction to initialize it:

```
  (assign compapp (label compound-apply))
  (branch (label external-entry))       ; branches if flag is set
read-eval-print-loop
  ...
```

To test your code, start by defining a procedure `f` that calls a procedure `g`. Use `compile-and-go` to compile the definition of `f` and start the evaluator. Now, typing at the evaluator, define `g` and try to call `f`.

Exercise 5.48

The `compile-and-go` interface implemented in this section is awkward, since the compiler can be called only once (when the evaluator machine is started). Augment the compiler-interpreter interface by providing a `compile-and-run` primitive that can be called from within the explicit-control evaluator as follows:

```
;;; EC-Eval input:
(compile-and-run
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n))))
;;; EC-Eval value:
ok

;;; EC-Eval input:
(factorial 5)
;;; EC-Eval value:
120
```

Exercise 5.49

As an alternative to using the explicit-control evaluator's read-eval-print loop, design a register machine that performs a read-compile-execute-print loop. That is, the machine should run a loop that reads an expression, compiles it, assembles and executes the resulting code, and prints the result. This is easy to run in our simulated setup, since we can arrange to call the procedures `compile` and `assemble` as "register-machine operations."

Exercise 5.50

Use the compiler to compile the metacircular evaluator of section 4.1 and run this program using the register-machine simulator. (To compile more than one definition at a time, you can package the definitions in a `begin`.) The resulting interpreter will run very slowly because of the multiple levels of interpretation, but getting all the details to work is an instructive exercise.

Exercise 5.51

Develop a rudimentary implementation of Scheme in C (or some other low-level language of your choice) by translating the explicit-control evaluator of section 5.4 into C. In order to run this code you will need to also provide appropriate storage-allocation routines and other run-time support.

Exercise 5.52

As a counterpoint to exercise 5.51, modify the compiler so that it compiles Scheme procedures into sequences of C instructions. Compile the metacircular evaluator of section 4.1 to produce a Scheme interpreter written in C.