[Go to first, previous, next page; contents; index]

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 *instruction 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).

```
(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.

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))
```

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]

`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>)
```

---

[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.

[Go to first, previous, next page; contents; index]