CS450 - Structure of Higher Level Languages

Register Machines

November 23, 2020

Computing with Register Machines

- We have seen how we can simulate a scheme evaluator using scheme as the underlying language.
- However, some questions about the lower-level behavior of the language are still unanswered.
- 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.
- Let us 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*.

Computing with Register Machines

- 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, we will examine several Lisp procedures and design a specific register machine to execute each procedure.
- Most of the primitive operations of our register machines are very simple: an operation might add the numbers fetched from two registers, storing the result into a third register.
- In order to deal with lists we will also use the memory operations `car`, `cdr`, and `cons`, which require an elaborate storage-allocation mechanism.

Example – GCD

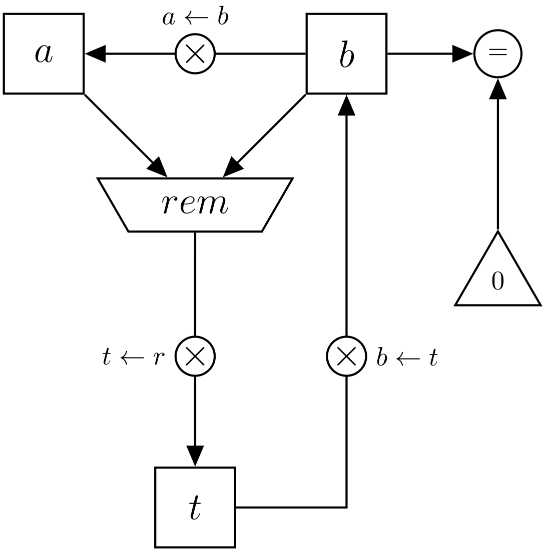
- To design a register machine, we must design its *data paths* (registers and operations) and the *controller* that sequences these operations.
- To illustrate the design of a simple register machine, let us examine Euclid's Algorithm for the GCD of two integers.
- Euclid's Algorithm can be carried out by an iterative process, as specified by the following procedure:

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

GCD Example

- We must keep track of two numbers, a and b, so let us assume that these numbers are stored in two registers with those names.
- We have to test whether the contents of register b is 0 and compute the remainder of the contents of register a divided by the contents of register b.
- Assume for now that we have a primitive device that computes remainders.
- On each cycle of the algorithm, the contents of a must be replaced by the contents of 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.
- In our model 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 t.

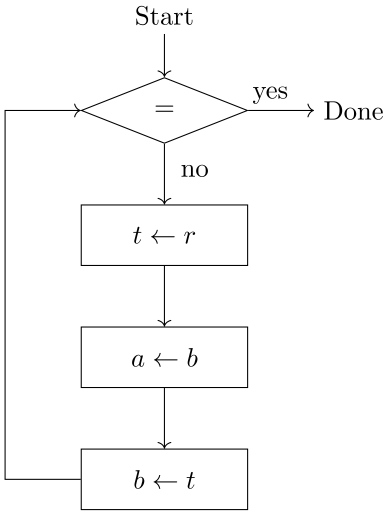
Data Path Illustration



Data Path Illustration

- The registers 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 source of data for a register can be another register (as in the $a \leftarrow b$ assignment), an operation result (as in the $t \leftarrow r$ assignment), or a constant (a built-in value that cannot be changed, represented by a triangle).
- 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.

Data Controller Illustration



Data Controller Illustration

- In order for the data paths to actually compute GCDs, the buttons must be pushed in the correct sequence.
- The elements of the controller diagram indicate how the data-path components should be operated.
- The rectangular boxes identify data-path buttons to be pushed, and the arrows describe the sequencing from one step to the next.
- The diamond 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.
- Together, the data paths and the controller completely describe a machine for computing GCDs.
- We start the controller at the place marked start, after placing numbers in registers a and b. When the controller reaches done, the value of the GCD is in register a.

Register Machine Language

- We need an adequate way to describe more complicated procedures.
- 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.
- 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.
- To describe an operation, we give it a name and specify its inputs.

Register Machine Language

- 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 to a location indicated by a controller label, based on the result of the previous test. (The test and branch 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.
- We start at the beginning of the controller instruction sequence and stop when execution reaches the end of the sequence.

GCD Example

```
(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)))))
```

GCD Example

```
(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
```

GCD Example

- This description forces us to go back and forth between the controller and path.
- We will leave only the controller and give informative names to the buttons and actions.
- 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)
```

GCD Example

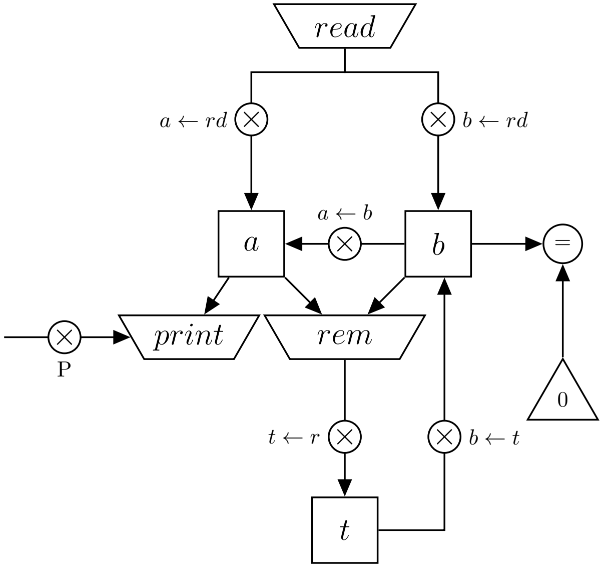
- Let us modify the GCD machine so that we can type in the input numbers and get the answer printed at our terminal.
- Let us assume (as we do when we use read and display in Scheme) that `read` and `print` are available as primitive operations.
- `Read` produces a value that can be stored in a register, but its value depends on something that happens outside the machine.
- `Print` does not produce an output value to be stored in a register. We will refer to this kind of operation as an `action`.
- We will represent an action in a data-path diagram as a trapezoid that contains the name of the action.

GCD Example

- 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))
  ```
- We can modify the diagram, so that 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.

GCD Example



read
$a \leftarrow rd$
$b \leftarrow rd$
a
$a \leftarrow b$
b
=
print
rem
P
0
$t \leftarrow r$
$b \leftarrow t$
t

GCD Example

```
(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)))
```

Using Stacks to Implement Recursion

- We can implement iterative processes 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 is completely determined by the contents of the registers (the values of the state variables).
- Implementing recursive processes, however, requires an additional mechanism.
- See for example the recursive factorial function:

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

Using Stacks to Implement Recursion

- The value obtained for (n - 1)! must be multiplied by n to get the final answer.
- We need the values from previous calls to compute the result.
- We would need "nested" instances of machines inside machines...
- However, only one instance is "active" at any given moment.
- 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 mean time, the values we save must be restored in reverse order of 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*.

Using Stacks to Implement Recursion

- 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.
- In particular, the `factorial` machine has a stack and three registers, called `n`, `val`, and `continue`.
- The `continue` register transfers 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.

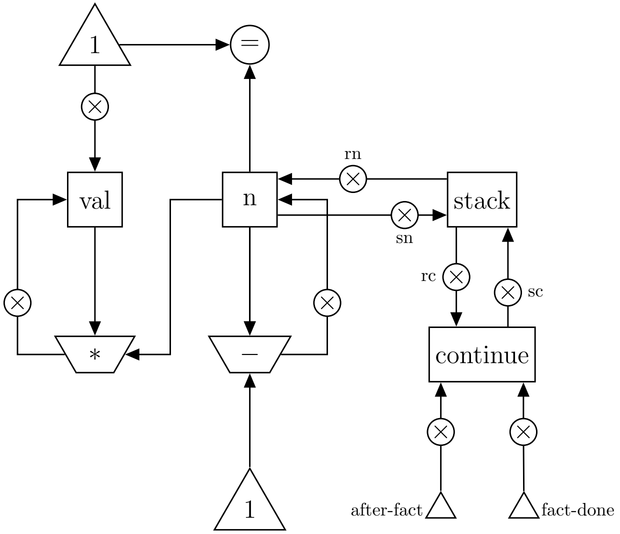
Controller for Recursive Factorial

```
controller
   ; set up final return address
   (assign continue (label fact-done))
  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))
```

Controller for Recursive Factorial (Cont.)

```
after-fact
  (restore n)
  (restore continue)
   ; val now contains n(n - 1)!
  (assign val (op *) (reg n) (reg val))
  ; return to caller
  (goto (reg continue))
base-case
; base case: 1! = 1
  (assign val (const 1))
 ; return to caller
  (goto (reg continue))
fact-done)
```

Controller for Recursive Factorial



Instruction Summary

```
(assign <register-name> (reg <register-name>))
(assign <register-name> (const <constant-value>))
(assign <register-name> (op <operation-name>)
<input1> ... <inputn>)
(perform (op <operation-name>) <input1> ... <inputn>)
(test (op <operation-name>) <input1> ... <inputn>)
(branch (label <label-name>))
(goto (label <label-name>))

(assign <register-name> (label <label-name>))
(goto (reg <register-name>))

(save <register-name>)
(restore <register-name>)
```

The Register Machine Simulator

- We must test the machines we design to see if they perform as expected.
- We will 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 and the other three allow us to simulate the machine by manipulating the model.

The Register Machine Simulator

```
(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.

The Register Machine Simulator – GCD Example

```
(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 Register Machine Simulator – GCD Example

- The first argument is a list of register names.
- The next argument is a table (a list of two-element lists) that pairs each operation name with its Scheme implementation.
- The last argument specifies the controller as a list of labels and machine instructions.
- 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
```

The Machine Model

- The machine model is represented using message-passing.
- The procedure `make-new-machine` constructs the parts of the machine model that are common to all register machines.
- It's essentially a container for some registers and a stack, with an execution mechanism that processes the controller instructions one by one.
- The basic model is then extended to include the specific machine by allocating the registers, installing the operations and installing the instructions.
- The assembler transforms the controller list into instructions.
- The returned value is the machine itself.

The 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))
```

The Registers

Registers are a function with a local state.

```
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))

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

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

The Stack

```
(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 Machine Itself

- The `make-new-machine` procedure constructs an object with a stack, an initially empty instruction sequence, a list of operations, and a register table that initially contains two registers, named `flag` and `pc` (for "program counter").
- 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), and branches decide whether or not to branch by examining the contents of flag.
- The `pc` register determines the sequencing of instructions as the machine runs, implemented by the internal procedure `execute`.
- Each machine instruction is a data structure that includes a procedure of no arguments and calling this procedure simulates executing the instruction.

The Machine Itself

- 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 until pc points to the end of the instruction sequence.
- 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 to the next instruction in the sequence.
- Each call to `execute` calls `execute` again, but this does not produce an infinite loop because we stop when pc runs out,

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.
- 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 produces 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.

The Assembler Code

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

extract-labels Code

```
(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)))))))
```

update-insts! Code

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)))
```

make-instruction

- Here we pair 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))
```

make-instruction

- The assembler calls make-execution-procedure to generate the execution procedure for an instruction.
- This procedure dispatches on the type of instruction to generate the appropriate execution procedure.
- 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.

make-instruction

```
(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))))
```

Assign instruction

```
(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)))))
```

test instruction

- 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))))
```

branch instruction

- We check the contents of the flag register and either set the contents of the pc to the branch destination (if the branch is taken) or else just advance 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 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))))
```

goto instruction

Similar to a branch, except that the destination may be either a label or as a register, and there is no condition.

```
(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)))))
```

Stack save and restore instructions

```
(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))
```

make-perform

`make-perform` generates an execution procedure for the action to be performed.

```
(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 or for input to an operation

```
(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))))
```

Making an operation expression

- Assign, perform, and test instructions may include the application of a machine operation to some operands.
- 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)))))
```