Register-machine simulator from section 5.2 of STRUCTURE AND INTERPRETATION OF COMPUTER PROGRAMS

This file can be loaded into Scheme as a whole.
Then you can define and simulate machines as shown in section 5.2

**NB** there are two versions of make-stack below. Choose the monitored or unmonitored one by reordering them to put the one you want last, or by commenting one of them out. Also, comment in/out the print-stack-statistics op in make-new-machine. To find this stack code below, look for comments with **Commented and reformatted by C. Offner (Spring/2002-Fall/2003)**

Changed "primitive" to "machine-primitive" to avoid confusion with Scheme primitives. (This term is used only in the comments.) The machine-primitives are those operators that are used in (op ...) expressions or (perform ...) instructions.

Changed lookup-prim to lookup-machine-primitive, and changed make-primitive-exp to make-elementary-exp, for the same reason.

The register machine

make-machine is what the user invokes to create a new machine. It takes three arguments:

register-names: a list of register names that are needed by the instructions in the machine.
ops: a list of the machine’s "machine-primitive" operations, and how to perform them.
controller-text: the actual sequence of instructions of the machine. The machine starts execution with the first instruction.

make-machine performs three actions:

1. It calls make-new-machine (defined below) to construct the skeleton of the machine.
2. It adds the registers specified (in register-names) to the machine.
3. It adds to the machine-primitive operators specified (in ops) to the machine.
4. Finally, it calls assemble (defined below) to assemble each instruction (in controller-text) so that it can actually be executed. It then adds these assembled instructions to the machine.

(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))
;; Making and accessing registers

(define (make-register)
  (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))

;; Creating a stack

;; **original (unmonitored) version from section 5.2.1
(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))

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

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

;; **monitored version from section 5.2.4
(define (make-stack)
  (let ((s '())
        (number-pushes 0)
        (max-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))
      (newline))
    (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))

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

(define (push stack value)
  ((stack 'push) value))
Making the skeleton of a machine

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(make-new-machine)

(let ((pc (make-register))
      (flag (make-register))
      (stack (make-stack))
      (the-instruction-sequence '()))

(let ((the-ops
        (list (list 'initialize-stack
                  (lambda () (stack 'initialize)))
              ;;; next for monitored stack (as in section 5.2.4)
              ;;; -- comment out if not wanted
              (list 'print-stack-statistics
                    (lambda () (stack 'print-statistics))))))

(allocate-register name)

(if (assoc name register-table)
    (error "Multiply defined register: " name)
    (set! register-table                    (cons (list name (make-register))                          register-table)))

'register-allocated)

(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))))))

(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))

;;;;;;;;;;;;;;;; Access functions for the machine:

(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)

(define (get-register machine reg-name)
  ((machine 'get-register) reg-name))
The assembler

assemble first calls extract-labels. This creates two lists:

1. a list of labels. Each label is paired with the instruction it refers to.
2. a list of instruction. The labels have been removed from this list. Each instruction is paired (initially) with the empty list.

Then update-insts! is called to replace the empty list paired with each instruction with the actual code to be generated to implement that instruction.

These routines are written in continuation-passing style. So instead of something like this:

(define (assemble ...) (update-insts! (extract-labels ...) ...))

update-insts! becomes the kernel of the continuation of extract-labels.

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

(define (extract-labels text receive)
  (if (null? text) (receive () '()) ;; This is where everything really happens, at the end. (extract-labels (cdr text) (lambda (insts labels) ;; This is where labels and insts are accumulated (as on a stack):
  (let ((next-inst (car text))) ;; either like this
    (receive insts
      (cons (make-label-entry next-inst insts) labels))
      (receive (cons (make-instruction next-inst insts) labels))))))

(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 ;; Generate the machine code.
          (instruction-text inst) labels machine
          (pc flag stack ops)) insts)))))

;; make-instruction will later be filled in by update-insts! using set-instruction-execution-proc!

;; instruction text and whose second element is initially empty. The second element will later be filled in by update-insts! using set-instruction-execution-proc!

(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-car! inst proc))

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

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

;; make-instruction creates a pair whose first element is the second element is the "rest of the instruction sequence". Thus, branching to that label amounts to resuming execution at the second element of the pair. (One could think of the second element of the pair as the address in instruction memory referred to by the label.)

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

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

;; make-execution-procedure dispatches on the type of instruction to the specific machine code generation routines. Each such routine returns a lambda expression which when executed, performs the action specified by the instruction being assembled.

(define (make-execution-procedure inst labels machine)
  (cond ((eq? (car inst) 'assign)
      (make-assign inst machine labels ops pc))
    ((eq? (car inst) 'test)
      (make-test inst machine labels ops pc))
    ((eq? (car inst) 'branch)
      (make-branch inst machine labels flag pc))
    ((eq? (car inst) 'save)
      (make-save inst machine stack pc))
    ((eq? (car inst) 'restore)
      (make-revert inst machine stack pc))
    ((eq? (car inst) 'perform)
      (make-perform inst machine labels ops pc))
    (else (error "Unknown instruction type -- ASSEMBLE " insts)))))
```
(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-elementary-exp
                (car value-exp) machine labels)))
         (lambda ()
          ; execution procedure for assign
          (set-contents! target (value-proc))
          (advance-pc pc))))))

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

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

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

(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))

(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))

(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)))
            (error "Unknown expression type -- ASSEMBLE " exp)))
      ((register-exp? dest)
       (let ((reg (get-register machine (register-exp-reg dest)))
            (lambda ()
             (set-contents! reg (pop stack))
             (advance-pc pc))))
      (error "Bad GOTO instruction -- ASSEMBLE " inst))))

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

(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))

(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))

(define (make-elementary-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))
               (error "Unknown expression type -- ASSEMBLE " 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))

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

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

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

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

;; from 4.1
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      #f))

'(REGISTER SIMULATOR LOADED)