```
;;;;EXPLICIT-CONTROL EVALUATOR FROM SECTION 5.4 OF
;;;; STRUCTURE AND INTERPRETATION OF COMPUTER PROGRAMS

;;; To use it
;;; -- load "load-eceval.scm", which loads this file and the
;;;    support it needs (including the register-machine simulator)
;;;    and then defines the global environment and starts the machine.

;; To restart, just do
;; (start eceval)
;;;;;;;;;;


;;**NB. To [not] monitor stack operations, comment in/[out] the line after
;; print-result in the machine controller below
;;**Also choose the desired make-stack version in regsim.scm

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      The built-in (machine-primitive) operations of the machine
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Each machine automatically has two built-in operations:
;;;    initialize-stack
;;;    print-stack-statistics

;;; All other built-in operations have to be declared here.
;;; Implementations for them have to be provided elsewhere, as noted
;;; below.

(define eceval-operations
  (list
   ;; primitive Scheme operations
   (list 'read read)

   ;; operations in syntax.scm
   (list 'self-evaluating? self-evaluating?)
   (list 'quoted? quoted?)
   (list 'text-of-quotation text-of-quotation)
   (list 'variable? variable?)
   (list 'assignment? assignment?)
   (list 'assignment-variable assignment-variable)
   (list 'assignment-value assignment-value)
   (list 'definition? definition?)
   (list 'definition-variable definition-variable)
   (list 'definition-value definition-value)
   (list 'lambda? lambda?)
   (list 'lambda-parameters lambda-parameters)
   (list 'lambda-body lambda-body)
   (list 'if? if?)
   (list 'if-predicate if-predicate)
   (list 'if-consequent if-consequent)
   (list 'if-alternative if-alternative)
   (list 'begin? begin?)
   (list 'begin-actions begin-actions)
   (list 'last-exp? last-exp?)
   (list 'first-exp first-exp)
   (list 'rest-exps rest-exps)
   (list 'application? application?)
   (list 'operator operator)
   (list 'operands operands)
   (list 'no-operands? no-operands?)
   (list 'first-operand first-operand)
   (list 'rest-operands rest-operands)

   ;; operations in eceval-support.scm
   (list 'true? true?)
   (list 'make-procedure make-procedure)
   (list 'compound-procedure? compound-procedure?)
   (list 'procedure-parameters procedure-parameters)
   (list 'procedure-body procedure-body)
   (list 'procedure-environment procedure-environment)
   (list 'extend-environment extend-environment)
   (list 'lookup-variable-value lookup-variable-value)
   (list 'set-variable-value! set-variable-value!)
   (list 'define-variable! define-variable!)
   (list 'primitive-procedure? primitive-procedure?)
   (list 'apply-primitive-procedure apply-primitive-procedure)
   (list 'prompt-for-input prompt-for-input)
   (list 'announce-output announce-output)
   (list 'user-print user-print)
   (list 'empty-arglist empty-arglist)
   (list 'adjoin-arg adjoin-arg)
   (list 'last-operand? last-operand?)
   (list 'no-more-exps? no-more-exps?)  ;for non-tail-recursive machine
   (list 'get-global-environment get-global-environment))
   )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      The machine itself
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define eceval
  (make-machine
   '(exp env val proc argl continue unev)
   eceval-operations
   '(
;;; Note that the read-eval-print-loop comes first.  This ensures that
;;; the program starts by entering this loop.
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
;;**following instruction optional -- if use it, need monitored stack
  (perform (op print-stack-statistics))
  (perform
   (op announce-output) (const ";;; EC-Eval value:"))
  (perform (op user-print) (reg val))
  (goto (label read-eval-print-loop))

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

unknown-procedure-type
  (restore continue)
  (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))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      The main dispatch routine
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

eval-dispatch
  ;; On entry,
  ;;   exp contains the expression to be evaluated.
  ;;   env contains the environment in which to evaluate the expression.
  ;;   continue contains the label at which to continue execution.
  ;; On exit,
  ;;   val holds the result of evaluating the expression.
  ;;   Execution continues at the label specified in continue.
  (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))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      self-evaluating expressions
;;;      variable names
;;;      quoted expressions
;;;      LAMBDA expressions
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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))

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

;;;      procedure applications
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

ev-application
  ;; This ultimately ends up in either
  ;;   primitive-apply (which restores continue), or
  ;;   compound-apply (which winds up in ev-sequence, which restores continue)
  (save continue)
  ;; We save env to evaluate all the operands in -- the operator and all its
  ;; operands must be evaluated in the same environment.
  (save env)
  (assign unev (op operands) (reg exp))
  ;; We save unev (the list of remaining unevaluated operands) because
  ;; this register tends to be used as a temporary.
  (save unev)
  (assign exp (op operator) (reg exp))
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch))
ev-appl-did-operator
  ;; The evaluated procedure is now in the val register.  We move it (below)
  ;; into the proc register, and save that register (if there are any
  ;; arguments to evaluate) to protect against subsidiary procedure
  ;; applications.
  ;;
  ;; argl holds the list of evaluated arguments.
  ;; unev holds the list of remaining unevaluated arguments.
  (restore unev)
  (restore env)
  (assign argl (op empty-arglist))
  (assign proc (reg val))
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))
  (save proc)
ev-appl-operand-loop
  ;; Save argl to protect agains subsidiary procedure calls.
  (save argl)
  (assign exp (op first-operand) (reg unev))
  ;; Are we evaluating the last operand?  If so, don't bother saving any
  ;; registers; just go evaluate it.
  (test (op last-operand?) (reg unev))
  (branch (label ev-appl-last-arg))
  ;; This is not the last operand.  Again save env and unev.  Call
  ;; eval-dispatch to evaluate the operand.
  (save env)
  (save unev)
  (assign continue (label ev-appl-accumulate-arg))
  (goto (label eval-dispatch))
ev-appl-accumulate-arg
  ;; Restore all the registers saved around the call, move the evaluated
  ;; argument into the argl list, and truncate the unev list.  Then go
  ;; back around the loop again.
  (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))
ev-appl-last-arg
  ;; We're evaluating the last operand.  Just call eval-dispatch.
  (assign continue (label ev-appl-accum-last-arg))
  (goto (label eval-dispatch))
ev-appl-accum-last-arg
  ;; Now restore the argl list, accumulate the last (evaluated) argument
```

```
  ;; into it, and restore the proc register.
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore proc)
  (goto (label apply-dispatch))  ; This is not needed; could just fall
                                 ; through.  (The label is needed,
                                 ; though, since apply-dispatch is
                                 ; jumped to from other places.)
apply-dispatch
  ;; On entry,
  ;;   proc contains the procedure to apply.
  ;;   argl contains the argument list.
  ;;   The continuation is at the top of the stack.
  ;;
  ;; On exit (from either primitive-apply or compound-apply),
  ;;   val will hold the result of the procedure application.
  ;;   The code will exit to the continuation (popped from the stack).
  (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))

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

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))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      BEGIN expressions
;;;      sequences
;;;      IF expressions
;;;      assignment expressions
;;;      definitions
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

;;; Sequences occur in two places:
;;;   a) The body of a procedure is a sequence.
;;;   b) A BEGIN expression is a stand-alone sequence.
;;; So ev-sequence is jumped to initially from one of those two places.

ev-sequence
  ;; On entry,
  ;;   unev contains the (unevaluated) sequence elements.  The first
  ;;     one will be put in the exp register.
  ;;   env contains the environment in which to evaluate the sequence
  ;;     elements.
  ;;   The continuation is on top of the stack.
  (assign exp (op first-exp) (reg unev))
  ;; If this is the last element of the sequence, we just go evaluate
  ;; it.  We don't need to save anything on the stack.
  (test (op last-exp?) (reg unev))
  (branch (label ev-sequence-last-exp))
  ;; Otherwise, we need to save unev (to keep the remainder of the
  ;; unevaluated sequence) and env (so all elements of the sequence
  ;; can be evaluated in the same environment).
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  ;; Now we're back from evaluating the sequence element.  Restore env and
  ;; unev, truncate unev, and go around the loop again.
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-last-exp
  ;; Just go evaluate the last element of the sequence tail-recursively.
  (restore continue)
  (goto (label eval-dispatch))

ev-if
  (save exp)
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg exp))
  (goto (label eval-dispatch))
ev-if-decide
  (restore continue)  ; Since this is where eval-dispatch returns to,
  (restore env)       ;   this is where the restores have to go.
  (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))

ev-assignment
  (assign unev (op assignment-variable) (reg exp))
  (save unev)
  (assign exp (op assignment-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-assignment-1))
  (goto (label eval-dispatch))
ev-assignment-1
  (restore continue)  ; Since this is where eval-dispatch returns to,
  (restore env)       ;   this is where the restores have to go.
  (restore unev)
  (perform
   (op set-variable-value!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))

ev-definition
  (assign unev (op definition-variable) (reg exp))
```

```
  (save unev)
  (assign exp (op definition-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-definition-1))
  (goto (label eval-dispatch))
ev-definition-1
  (restore continue)  ; Since this is where eval-dispatch returns to,
  (restore env)       ;   this is where the restores have to go.
  (restore unev)
  (perform
   (op define-variable!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))
   )))

'(EXPLICIT CONTROL EVALUATOR LOADED)
```