File: eceval-support.scm

This file contains procedures that are taken from the Chapter 4 interpreter, and are used as machine-primitive operators in the register machines of Chapter 5.

It is loaded by

load-eceval.scm to construct the explicit-control evaluator eceval.

machine-shell.scm to construct the register machine that executes compiled code.

All operations are used by both these machines except as noted.

(load "syntax.scm")

Truth values

(define (true? x) (not (eq? x #f)))

not used by eceval itself -- used by compiled code when that is run in the eceval machine

(define (false? x) (eq? x #f))

Procedures

following compound-procedure operations not used by compiled code

(define (make-procedure parameters body env) (list 'procedure parameters body env))

(define (compound-procedure? p) (tagged-list? p 'procedure))

(define (procedure-parameters p) (cadr p))

(define (procedure-body p) (caddr p))

(define (procedure-environment p) (cadddr p))

(extend-environment vars vals base-env)

(if (= (length vars) (length vals)) (cons (make-frame vars vals) base-env)
  (if (< (length vars) (length vals))
    (error "Too many arguments supplied" vars vals)
    (error "Too few arguments supplied" vars vals)))

Looking up a variable in an environment

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals))))
    (if (eq? env the-empty-environment) (error "Unbound variable" var)
      (let ((frame (first-frame env)))
        (scan (frame-variables frame) (frame-values frame))))
  (env-loop env))

Setting a variable to a new value in a specified environment.

Note that if it is an error if the variable is not already present
(i.e., previously defined) in that environment.

(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals))))
    (if (eq? env the-empty-environment)
      (error "Unbound variable -- SET!" var)
      (let ((frame (first-frame env)))
        (scan (frame-variables frame) (frame-values frame))))
  (env-loop env))

Defining a (possibly new) variable. First see if the variable already exists. If it does, just change its value to the new value. If it does not, define the new variable in the current frame.

(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars) (add-binding-to-frame! var val frame))
            (else (scan (cdr vars) (cdr vals))))))
(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                            (primitive-procedure-objects)
                            the-empty-environment)))
    (define-variable! 'true #t initial-env)
    (define-variable! 'false #f initial-env)
    initial-env))

;; Define the primitive procedures
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

(define (primitive-implementation proc) (cadr proc))

;; Here is where we rely on the underlying Scheme implementation to
;; know how to apply a primitive procedure.
(define (apply-primitive-procedure proc args)
  (apply (primitive-implementation proc) args))

;; These are procedures in code that we will compile (or interpret)
;; that we want to regard as primitive.
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        (list '+ +)
        (list '-' -)
        (list '** *)
        (list '=' =)
        (list '/ /)
        (list '>' >)
        (list '< <)
        (list 'list list)
        ))

(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))

;; The initial environment

;; This is initialization code that is executed once, when the the
;; interpreter is invoked.

;; Support for the main driver loop

;; Simulation of new machine operations needed by eceval machine

;; Simulation of new machine operations needed by
;; eceval machine (not used by compiled code)

;; From section 5.4.1 footnote
(define (empty-arglist) '())
(define (adjoin-arg arg arglist) (append arglist (list arg)))
(define (last-operand? ops) (null? (cdr ops)))

;; From section 5.4.2 footnote, for non-tail-recursive sequences
(define (no-more-ops? seq) (null? seq))

;; From section 5.4.4 footnote
(define (get-global-environment) the-global-environment)
;; will do following when ready to run, not when load this file
;; (define the-global-environment (setup-environment))

;; Simulation of new machine operations needed for compiled code
;; and eceval/compiler interface (not used by plain eceval machine)

;; From section 5.5.2 footnote
(define (make-compiled-procedure entry env)
  (list 'compiled-procedure entry env))
(define (compiled-procedure? proc)
  (tagged-list? proc 'compiled-procedure))
(define (compiled-procedure-entry c-proc) (cadr c-proc))
(define (compiled-procedure-env c-proc) (caddr c-proc))