```scheme
;;; file: s450.scm
;;;
;;; Metacircular evaluator from chapter 4 of STRUCTURE AND
;;; INTERPRETATION OF COMPUTER PROGRAMS (2nd edition)
;;;
;;; Modified by kwn, 3/4/97
;;; Modified and commented by Carl Offner, 10/21/98 -- 10/12/04
;;;
;;; This code is the code for the metacircular evaluator as it appears
;;; in the textbook in sections 4.1.1-4.1.4, with the following
;;; changes:
;;;
;;; 1.  It uses #f and #t, not false and true, to be Scheme-conformant.
;;;
;;; 2.  Some function names were changed to avoid conflict with the
;;; underlying Scheme:
;;;
;;;       eval => xeval
;;;       apply => xapply
;;;       extend-environment => xtend-environment
;;;
;;; 3.  The driver-loop is called s450.
;;;
;;; 4.  The booleans (#t and #f) are classified as self-evaluating.
;;;
;;; 5.  These modifications make it look more like UMB Scheme:
;;;
;;;        The define special form evaluates to (i.e., "returns") the
;;;          variable being defined.
;;;        No prefix is printed before an output value.
;;;
;;; 6.  I changed "compound-procedure" to "user-defined-procedure".
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      xeval and xapply -- the kernel of the metacircular evaluator
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (xeval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (xeval (cond->if exp) env))
        ((application? exp)
         (xapply (xeval (operator exp) env)
                 (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- XEVAL " exp))))

(define (xapply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((user-defined-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (xtend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- XAPPLY " procedure))))

;;; Handling procedure arguments

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (xeval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))

;;; These functions, called from xeval, do the work of evaluating some
;;; of the special forms:

(define (eval-if exp env)
  (if (true? (xeval (if-predicate exp) env))
      (xeval (if-consequent exp) env)
      (xeval (if-alternative exp) env)))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (xeval (first-exp exps) env))
        (else (xeval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

(define (eval-assignment exp env)
  (let ((name (assignment-variable exp)))
    (set-variable-value! name
                         (xeval (assignment-value exp) env)
                         env)
  name))    ;; A & S return 'ok

(define (eval-definition exp env)
  (let ((name (definition-variable exp)))
    (define-variable! name
      (xeval (definition-value exp) env)
      env)
  name))     ;; A & S return 'ok
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      Representing expressions
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Numbers, strings, and booleans are all represented as themselves.
;;; (Not characters though; they don’t seem to work out as well
;;; because of an interaction with read and display.)

(define (self-evaluating? exp)
  (or (number? exp)
      (string? exp)
      (boolean? exp)
      ))

;;; variables -- represented as symbols

(define (variable? exp) (symbol? exp))

;;; quote -- represented as (quote <text-of-quotation>)

(define (quoted? exp)
  (tagged-list? exp ’quote))

(define (text-of-quotation exp) (cadr exp))

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

;;; assignment -- represented as (set! <var> <value>)

(define (assignment? exp)
  (tagged-list? exp ’set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))

;;; definitions -- represented as
;;;    (define <var> <value>)
;;;  or
;;;    (define (<var> <parameter_1> <parameter_2> ... <parameter_n>) <body>)
;;;
;;; The second form is immediately turned into the equivalent lambda
;;; expression.

(define (definition? exp)
  (tagged-list? exp ’define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)
                   (cddr exp))))

;;; lambda expressions -- represented as (lambda ...)
;;;
;;; That is, any list starting with lambda.  The list must have at
;;; least one other element, or an error will be generated.

(define (lambda? exp) (tagged-list? exp ’lambda))

(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp))

(define (make-lambda parameters body)
  (cons ’lambda (cons parameters body)))

;;; conditionals -- (if <predicate> <consequent> <alternative>?)

(define (if? exp) (tagged-list? exp ’if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      #f))

(define (make-if predicate consequent alternative)
  (list ’if predicate consequent alternative))


;;; sequences -- (begin <list of expressions>)

(define (begin? exp) (tagged-list? exp ’begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))

(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))

(define (make-begin seq) (cons ’begin seq))


;;; procedure applications -- any compound expression that is not one
;;; of the above expression types.

(define (application? exp) (pair? exp))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))


;;; Derived expressions -- the only one we include initially is cond,
;;; which is a special form that is syntactically transformed into a
;;; nest of if expressions.

(define (cond? exp) (tagged-list? exp ’cond))
```

```scheme
(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))

(define (cond-predicate clause) (car clause))

(define (cond-actions clause) (cdr clause))

(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))

(define (expand-clauses clauses)
  (if (null? clauses)
      #f                          ; no else clause -- return #f
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF "
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      Truth values and procedure objects
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Truth values

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

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


;;; Procedures

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

(define (user-defined-procedure? p)
  (tagged-list? p 'procedure))


(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (cadddr p))
```

```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      Representing environments
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; An environment is a list of frames.

(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))

(define the-empty-environment '())

;;; Each frame is represented as a pair of lists:
;;;   1.  a list of the variables bound in that frame, and
;;;   2.  a list of the associated values.

(define (make-frame variables values)
  (cons variables values))

(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))

;;; Extending an environment

(define (xtend-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))
```

```scheme
;;; Setting a variable to a new value in a specified environment.
;;; Note that 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))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      The initial environment
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

(define (setup-environment)
  (let ((initial-env
         (xtend-environment (primitive-procedure-names)
                            (primitive-procedure-objects)
                            the-empty-environment)))
    initial-env))

;;; Define the primitive procedures

(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

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

(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
;;      more primitives
        ))

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

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

;;; 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))
```

```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      The main driver loop
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Note that (read) returns an internal representation of the next
;;; Scheme expression from the input stream.  It does NOT evaluate
;;; what is typed in -- it just parses it and returns an internal
;;; representation.  It is the job of the scheme evaluator to perform
;;; the evaluation.  In this case, our evaluator is called xeval.

(define input-prompt "s450==> ")

(define (s450)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (xeval input the-global-environment)))
      (user-print output)))
  (s450))

(define (prompt-for-input string)
  (newline) (newline) (display string))

;;; Note that we would not want to try to print a representation of the
;;; <procedure-env> below -- this would in general get us into an
;;; infinite loop.

(define (user-print object)
  (if (user-defined-procedure? object)
      (display (list 'user-defined-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      Here we go:  define the global environment and invite the
;;;        user to run the evaluator.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(define the-global-environment (setup-environment))

(display "... loaded the metacircular evaluator. (s450) runs it.")
(newline)
```