```
#lang racket
;;
;; eval.scm − 6.037
;;

(require r5rs)
(define first car)
(define second cadr)
(define third caddr)
(define fourth cadddr)
(define rest cdr)

(define (tagged−list? exp tag)
  (and (pair? exp) (eq? (car exp) tag)))

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

(define (quoted? exp) (tagged−list? exp 'quote))
(define (text−of−quotation exp) (cadr exp))

(define (variable? exp) (symbol? exp))
(define (assignment? exp) (tagged−list? exp 'set!))
(define (assignment−variable exp) (cadr exp))
(define (assignment−value exp) (caddr exp))
(define (make−assignment var expr)
  (list 'set! var expr))

(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))))  ; formal params, body
(define (make−define var expr)
  (list 'define var expr))

(define (lambda? exp) (tagged−list? exp 'lambda))
(define (lambda−parameters lambda−exp) (cadr lambda−exp))
(define (lambda−body lambda−exp) (cddr lambda−exp))
(define (make−lambda parms body) (cons 'lambda (cons parms body)))

(define (if? exp) (tagged−list? exp 'if))
(define (if−predicate exp) (cadr exp))
(define (if−consequent exp) (caddr exp))
(define (if−alternative exp) (cadddr exp))
(define (make−if pred conseq alt) (list 'if pred conseq alt))

(define (cond? exp) (tagged−list? exp 'cond))
(define (cond−clauses exp) (cdr exp))
(define first−cond−clause car)
(define rest−cond−clauses cdr)
(define (make−cond seq) (cons 'cond seq))

(define (let? expr) (tagged−list? expr 'let))
(define (let−bound−variables expr) (map first (second expr)))
(define (let−values expr) (map second (second expr)))
(define (let−body expr) (cddr expr)) ;differs from lecture−−body may be a sequence
(define (make−let bindings body)
  (cons 'let (cons bindings body)))

(define (begin? exp) (tagged−list? exp 'begin))
(define (begin−actions begin−exp) (cdr begin−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 exp) (cons 'begin exp))

(define (application? exp) (pair? exp))
(define (operator app) (car app))
(define (operands app) (cdr app))
(define (no−operands? args) (null? args))
(define (first−operand args) (car args))
(define (rest−operands args) (cdr args))
(define (make−application rator rands)
  (cons rator rands))

(define (and? expr) (tagged−list? expr 'and))
(define and−exprs cdr)
(define (make−and exprs) (cons 'and exprs))
(define (or? expr) (tagged−list? expr 'or))
(define or−exprs cdr)
(define (make−or exprs) (cons 'or exprs))
```



```
;;
;; this section is the actual implementation of meval
;;

(define (m−eval 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) (m−eval (cond−>if exp) env))
        ((let? exp) (m−eval (let−>application exp) env))
        ((application? exp)
         (m−apply (m−eval (operator exp) env)
                  (list−of−values (operands exp) env)))
        (else (error "Unknown expression type −− EVAL" exp))))

(define (m−apply procedure arguments)
  (cond ((primitive−procedure? procedure)
         (apply−primitive−procedure procedure arguments))
        ((compound−procedure? procedure)
         (eval−sequence
          (procedure−body procedure)
          (extend−environment (procedure−parameters procedure)
                              arguments
                              (procedure−environment procedure))))
        (else (error "Unknown procedure type −− APPLY" procedure))))

(define (list−of−values exps env)
  (cond ((no−operands? exps) '())
        (else (cons (m−eval (first−operand exps) env)
                    (list−of−values (rest−operands exps) env)))))

(define (eval−if exp env)
  (if (m−eval (if−predicate exp) env)
      (m−eval (if−consequent exp) env)
      (m−eval (if−alternative exp) env)
      ))

(define (eval−sequence exps env)
  (cond ((last−exp? exps) (m−eval (first−exp exps) env))
        (else (m−eval (first−exp exps) env)
              (eval−sequence (rest−exps exps) env))))

(define (eval−assignment exp env)
  (set−variable−value! (assignment−variable exp)
                       (m−eval (assignment−value exp) env)
                       env))

(define (eval−definition exp env)
  (define−variable! (definition−variable exp)
                    (m−eval (definition−value exp) env)
                    env))

(define (let−>application expr)
  (let ((names (let−bound−variables expr))
        (values (let−values expr))
        (body (let−body expr)))
    (make−application (make−lambda names body)
                      values)))

(define (cond−>if expr)
  (let ((clauses (cond−clauses expr)))
    (if (null? clauses)
        #f
        (if (eq? (car (first−cond−clause clauses)) 'else)
            (make−begin (cdr (first−cond−clause clauses)))
            (make−if (car (first−cond−clause clauses))
                     (make−begin (cdr (first−cond−clause clauses)))
                     (make−cond (rest−cond−clauses clauses)))))))

(define input−prompt ";;; M−Eval input:")
(define output−prompt ";;; M−Eval value:")

(define (driver−loop)
  (prompt−for−input input−prompt)
  (let ((input (read)))
    (if (eq? input '**quit**)
        'meval−done
        (let ((output (m−eval input the−global−environment)))
          (announce−output output−prompt)
          (pretty−display output)
          (driver−loop))))))
```

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

(define (announce-output string)
  (newline) (display string) (newline))

(define *meval-warn-define* #t) ; print warnings?
(define *in-meval* #f)          ; evaluator running


;;
;;
;; implementation of meval environment model
;;

; double bubbles
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
(define (compound-procedure? proc)
  (tagged-list? proc 'procedure))
(define (procedure-parameters proc) (second proc))
(define (procedure-body proc) (third proc))
(define (procedure-environment proc) (fourth proc))


; bindings
(define (make-binding var val)
  (list var val))
(define binding-variable car)
(define binding-value cadr)
(define (binding-search var frame)
  (if (null? frame)
      #f
      (if (eq? var (first (first frame)))
          (first frame)
          (binding-search var (rest frame)))))
(define (set-binding-value! binding val)
  (set-car! (cdr binding) val))

; frames
(define (make-frame variables values)
  (cons 'frame (map make-binding variables values)))
(define (frame-variables frame) (map binding-variable (cdr frame)))
(define (frame-values frame) (map binding-value (cdr frame)))
(define (add-binding-to-frame! var val frame)
  (set-cdr! frame (cons (make-binding var val) (cdr frame))))
(define (find-in-frame var frame)
  (binding-search var (cdr frame)))

; environments
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())

(define (find-in-environment var env)
  (if (eq? env the-empty-environment)
      #f
      (let* ((frame (first-frame env))
             (binding (find-in-frame var frame)))
        (if binding
            binding
            (find-in-environment var (enclosing-environment env))))))

; drop a frame
(define (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 args supplied" vars vals)
          (error "Too few args supplied" vars vals))))

; name rule
(define (lookup-variable-value var env)
  (let ((binding (find-in-environment var env)))
    (if binding
        (binding-value binding)
        (error "Unbound variable -- LOOKUP" var))))

(define (set-variable-value! var val env)
  (let ((binding (find-in-environment var env)))
    (if binding
        (set-binding-value! binding val)
        (error "Unbound variable -- SET" var))))

(define (define-variable! var val env)
  (let* ((frame (first-frame env))
         (binding (find-in-frame var frame)))
```



```
    (if binding
        (set-binding-value! binding val)
        (add-binding-to-frame! var val frame))))

; primitives procedures - hooks to underlying Scheme procs
(define (make-primitive-procedure implementation)
  (list 'primitive implementation))
(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 'set-car! set-car!)
        (list 'set-cdr! set-cdr!)
        (list 'null? null?)
        (list '+ +)
        (list '- -)
        (list '< <)
        (list '> >)
        (list '= =)
        (list 'display display)
        (list 'not not)
        ; ... more primitives
        ))

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

(define (primitive-procedure-objects)
  (map make-primitive-procedure (map cadr (primitive-procedures))))

(define (apply-primitive-procedure proc args)
  (apply (primitive-implementation proc) args))

; used to initialize the environment
(define (setup-environment)
  (let ((initial-env (extend-environment (primitive-procedure-names)
                                         (primitive-procedure-objects)
                                         the-empty-environment))
        (oldwarn *meval-warn-define*))
    (set! *meval-warn-define* #f)
    (define-variable! 'true #t initial-env)
    (define-variable! 'false #f initial-env)
    (set! *meval-warn-define* oldwarn)
    initial-env))

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

(define (refresh-global-environment)
  (set! the-global-environment (setup-environment))
  'done)
```