The 6.001 Lazy Meta-Circular Evaluator

The Core Evaluator

```
(define (l-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) (eval (cond->if exp) env))
        ((and? exp) (l-eval (and->if exp) env))
        ((until? exp) (eval-until exp env))
        ((application? exp)
         (l-apply (actual-value (operator exp) env)
                  (operands exp)
                  env))
        (else (error "Unknown expression type -- L-EVAL" exp))))

(define (l-apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure
                                    (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment (procedure-parameters procedure)
                              (list-of-delayed-args arguments env)
                              (procedure-environment procedure))))
        (else (error "Unknown procedure type -- L-APPLY" procedure))))


(define (actual-value exp env)
  (force-it (l-eval exp env)))

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

(define (list-of-delayed-args exps env)
  (if (no-operands? exps) '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps) env))))

(define (eval-if exp env)
  (if (actual-value (if-predicate exp) env)
      (l-eval (if-consequent exp) env)
      (l-eval (if-alternative exp) env)))

(define (eval-sequence exps env)
```

```
  (cond ((last-exp? exps) (l-eval (first-exp exps) env))
        (else (l-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (l-eval (assignment-value exp) env)
                       env))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
    (l-eval (definition-value exp) env)
    env))
```

Representing Expressions

```
(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 (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) ; formal params
                   (cddr exp)))) ; body

(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)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))

(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 (and? exp) (tagged-list? exp 'and))

(define (and-clauses exp) (cdr exp))

(define (until? exp) (tagged-list? exp 'until))

(define (until-test exp) (cadr exp))

(define (until-body exp) (cddr exp))

(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))
```

Representing procedures

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

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

(define (procedure-parameters p) (list-ref p 1))

(define (procedure-body p) (list-ref p 2))

(define (procedure-environment p) (list-ref p 3))
```

Representing environments

```
;; Implement environments as a list of frames; parent environment is
;; the cdr of the list. Each frame will be implemented as a list
;; of variables and a list of corresponding values.

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

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

(define the-empty-environment '())

(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))))

(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))))

(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 -- LOOKUP" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame) (frame-values frame)))))
  (env-loop env))

(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)) ; Same as lookup except for this
            (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))


(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))))
```

Primitive Procedures and the Initial Environment

```
(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?)
        (list 'list list)
        (list '+ +)
        (list '> >)
        (list '< <)
        (list '= =)
        (list '* *)
        (list '- -)
        ; ... more primitives
        ))

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

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

(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-global-environment (setup-environment))

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

The Read-Eval-Print Loop

```
(define input-prompt ";;; L-Eval input:")

(define output-prompt ";;; L-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (actual-value input the-global-environment)))
      (announce-output output-prompt)
      (display output)))
  (driver-loop))

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

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

Representing Thunks

```
(define (delay-it exp env) (list 'thunk exp env))

(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))

(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value (thunk-exp obj)
                                     (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj)))

(define (actual-value exp env)
  (force-it (l-eval exp env)))

(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))

(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))
```