```scheme
;; Scheme Intrepeter with builtins quote, if, define, eq?, lambda, car, cdr,
;; cons, number?, pair?, string?, +, -, *, /, =, print, and primitive datatypes
;; symbols, numbers, strings and pairs.  Simplifications: (1) Only one
;; expression in lambda bodies, use begin for more (2) define only assign
;; variables to values, use lambda for functions: (define id (lambda (x) x), (4)
;; no set!.  Tim Finin, finin@umbc.edu, May 2010.

(require scheme/mpair)

(define (mceval exp env)
 ;; mceval evaluate expression exp in environment env
 (cond ((or (number? exp) (string? exp)               ; numbers, strings, Booleans and
        (boolean? exp) (eof-object? exp)) exp)        ;   eof evaluate to themselves
       ((symbol? exp) (lookup exp env))               ; Look up value of a variable
       ((eq? (first exp) 'quote) (second exp))        ; quote suppresses evaluation.
       ((eq? (first exp) 'begin)                      ; (begin e1 e2 ... en) evals ei
        (last (map (lambda (x)(mceval x env))         ;   in order and returns value
                   (rest exp))))                      ;   of last one
       ((eq? (first exp) 'if)                         ; (if ...) evals its args
        (if (mceval (second exp) env)                 ;   condtionally
            (mceval (third exp) env)                  ;
            (mceval (fourth exp) env)))               ;
       ((eq? (first exp) 'define)                     ; Define adds/modifies value
        (mcdefine (second exp)                        ;   of a variable in current
                  (mceval (third exp) env) env))      ;   environment (ie, top frame)
       ((eq? (first exp) 'load)                       ; Load reads and evals expressions
        (call-with-input-file (second exp) mcload))   ;  in a file
       ((eq? (first exp) 'lambda)                     ; Create a user defined function:
        (list 'LAMBDA (second exp) (third exp) env)) ;  note that it save environment
       (else (mcapply (mceval (first exp) env)       ; Apply function to evaluated args
                      (map (lambda (x)(mceval x env))
                           (rest exp))))))

(define (mcapply proc args)
 ;; apply procedure proc to arguments args
 (cond ((procedure? proc) (apply proc args))
       ((and (pair? proc) (eq? (first proc) 'LAMBDA))
        (mceval (third proc)
                (cons (make-frame (second proc) args)
                      (fourth proc))))
       (else (mcerror "mcapply: Undefined procedure" proc))))

(define (make-frame vars values)
 ;; Makes an environment frame with variables vars and initial values
 ;; values.  L2ml converts a list of pairs to one of mutable pairs.
 (mmap mcons (l2ml vars) (l2ml values)))

(define (lookup var env)
 ;; return value of variable var in environment env
 (cond ((null? env) (mcerror "unbound variable" var))
       ((massoc var (first env))
        (mcdr (massoc var (first env))))
       (else (lookup var (rest env)))))
```

```
(define (mcdefine var val env)
 ;; define variable var in environment env, gving it value val
 (let ((frame (first env)))
   (if (massoc var frame)
        ;; variable already defined, change it's value
        (set-mcdr! (massoc var frame) val)
        ;; add a new var-val cell to the end of the frame
        (set-mcdr! (mlast-pair frame)
                   (mcons (mcons var val) null))))
 (void))

(define (mlast-pair ml)
 ;; like last-pair but for mlists: returns last mpair of the mlist
 (if (null? (mcdr ml))
     ml
     (mlast-pair (mcdr ml))))

(define (mcload file)
 ;; read and mceval expressions in file w.r.t. global-env
 (if (eq? eof (mceval (read file) global-env))
     (void)
     (mcload file)))

(define (mcscheme)
 ;; mcscheme read-eval-print loop
 (printf "mcscheme> ")
 (mcprint (mceval (read) global-env))
 (mcscheme))

(define (mcprint x)
 ;; mscheme's top-level print: print x iff it's not void
 (or (void? x) (printf "~s~n" x)))

(define (l2ml l)
 ;; takes a list and returns a mutable list (mlist)
 (if (null? l) l (mcons (car l) (l2ml (cdr l)))))

(define (mcerror msg  args)
 ;; print an error message and return \#<void>
 (printf "MCERROR: ~a ~s.~n" msg args)
 (void))

;;  Primitives to define using their Scheme counterparts
(define builtins '(car cdr cons number? pair? string? eq? + - * / = < > print eof))

;; intial global environment has the builtins bound to their Scheme values
(define global-env (list (make-frame builtins (map eval builtins))))

"mcscheme0.1:, (mcscheme) to start, control-C to leave"
```