```
#lang racket
(provide (all-defined-out))

;; Variable bindings

(define (make-binding var val)
  (list var val))

(define (binding-variable binding)
  (first binding))

(define (binding-value binding)
  (first (rest binding)))

;; Frames

(define (make-frame vars vals)
  (if (not (= (length vars)
              (length vals)))
      (println "Error: lists are not of the same length!")
      (map (lambda (x y)(make-binding x y)) vars vals)))

(define (empty-frame? frame)
  (match frame
    ((list ) #t)
    (_ #f)))

(define (first-binding frame)
  (if (> (length frame) 0)
      (first frame)
      (println "Error: frame is empty!")))

(define (rest-of-bindings frame)
  (if (> (length frame) 0)
      (rest frame)
      (println "Error: frame is empty!")))

(define (adjoin-binding binding frame)
  (match binding
    ((list a b)(cons binding frame))
    (_ (println "Error: invalid binding!"))))

(define (binding-in-frame var frame)
  (cond ((= (length frame) 0) #f)
        ((equal? (binding-variable (first-binding frame)) var)
(first-binding frame))
        (else (binding-in-frame var (rest-of-bindings frame)))))

;; Environments

(define (empty-env)
```

```
  (list ))

(define (empty-env? env)
  (equal? env (list )))

(define (first-frame env)
  (if (not (empty-env? env))
      (mcar env)
      (println "Error: empty environment!")))

(define (rest-of-frames env)
  (if (not (empty-env? env))
      (mcdr env)
      (println "Error: empty environment!")))

(define (set-first-frame! env new-frame)
  (if (not (empty-env? env))
      (set-mcar! env new-frame)
      (println "Error: empty environment!")))

(define (adjoin-frame frame env)
  (mcons frame env))

(define (extend-env vars vals base-env)
  (adjoin-frame (make-frame vars vals) base-env))

(define (binding-in-env var env)
  (cond ((empty-env? env) #f)
        ((binding-in-frame var (mcar env)) (binding-in-frame var (mcar
env)))
        (else (binding-in-env var (mcdr env)))))

(define (lookup-variable var env)
  (if (binding-in-env var env)
      (binding-value (binding-in-env var env))
      (println "Error: variable not bound in environment!")))

;; Quote and quoted expressions

(define (quoted? x)
  (match x
    ((list 'quote _) #t)
    (_ #f)))

(define (text-of-quotation x)
  (match x
    ((list 'quote y) y)
    (_ (println "Error: not quoted!"))))

;; define and variables

```

```
(define (tagged-list? exp tag)
  (if (and (list? exp)
           (> (length exp) 0))
      (equal? (first exp) tag)
      #f))

(define (definition? exp)
  (tagged-list-length-n? exp 'define 3))

(define (definition-variable exp)
  (first (rest exp)))

(define (definition-value exp)
  (first (rest (rest exp))))

(define (define-variable! var val env)
   (cond ((empty-env? env)(println "Error: no frame in environment!"))
         ((binding-in-frame var (first-frame env)) (println "Error:
binding already exists!"))
         (else (set-first-frame! env (adjoin-binding (make-binding var
val) (first-frame env))) var))) ; should this return var or val?


(define (eval-definition exp env)
  (let ((var (definition-variable exp))
        (val (i-eval (definition-value exp) env)))
    (define-variable! var val env)))

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

(define (tagged-list-length-n? exp tag n)
  (if (and (list? exp)
           (= (length exp) n))
      (equal? (first exp) tag)
      #f))

(define (tagged-list-min-length-n? exp tag n)
  (if (and (list? exp) (>= (length exp) n))
      (equal? (first exp) tag)
      #f))

(define (application? exp)
  (match exp
    ((list 'quote _ ...) #f)
    ((list a _ ...) #t)
    (_ #f)))

(define (operator exp)
  (if (application? exp)
      (first exp)
```

```
      (println "Error: not an application!")))

(define (operands exp)
  (cond ((not (application? exp)) (println "Error: not an application!"))
        ((= (length exp) 1) (list ))
        (else (rest exp))))

(define (eval-operands operands env)
  (map (lambda (x)(i-eval x env)) operands))

;; Primitives

(define make-primitive
  (lambda (name proc)
    (list 'primitive name proc)))

(define (primitive-procedure? exp)
  (tagged-list-length-n? exp 'primitive 3))

(define (primitive-name proc)
  (if (primitive-procedure? proc)
      (first (rest proc))
      (println "Error: not a primitive procedure!")))

(define (primitive-implementation proc)
  (if (primitive-procedure? proc)
      (first (rest (rest proc)))
      (println "Error: not a primitive procedure!")))

;; New definition of setup-env

(define setup-env-2
  (lambda ()
    (let ((primitive-procedures (list (list '= =)(list 'equal? equal?)
                                      (list '+ +)(list '- -)
                                      (list '/ /)(list '* *)
                                      (list '< <)(list '> >)
                                      (list '<= <=)(list '>= >=)
                                      (list 'void void)
                                      (list 'cons cons)(list 'mcons mcons)
                                      (list 'printf printf)
                                      (list 'empty? empty?)(list 'null?
null?)
                                      (list 'append append)(list 'length
length)
                                      (list 'reverse reverse)(list 'floor
floor)
                                      (list 'string-join string-join)
                                      (list 'list list)(list 'list? list?)
                                      (list 'mcdr mcdr)(list 'mcar mcar)
                                      (list 'first first)(list 'rest rest)
```

```
                                        (list 'void void)(list 'string?
string?)
                                        (list 'number? number?)(list
'boolean? boolean?)
                                        (list 'list-ref list-ref)(list
'read read)
                                        (list 'string-ref string-ref)
                                        (list 'string-length string-length)
                                        (list 'string-append string-append)
                                        (list 'display display)(list
'newline newline)
                                        (list 'read-line read-line)
                                        (list 'eof-object? eof-object?)
                                        (list 'open-input-file
open-input-file)
                                        (list 'open-input-string
open-input-string)
                                        )))
        (extend-env (append (map (lambda(x)(first x))
primitive-procedures)'(null))
                    (append (map (lambda(x)(make-primitive (first
x)(second x))) primitive-procedures) '(()))
                    (empty-env)))))

(define global-env (setup-env-2))


(define (reset-global)
  (set-mcar! global-env (mcar (setup-env-2)))
  (set-mcdr! global-env (mcdr (setup-env-2))))

(define (apply-primitive-procedure proc vals)
  (if (primitive-procedure? proc)
      (apply (primitive-implementation proc) vals)
      (println "Error: not a primitive procedure!")))

;; begin

(define (begin? exp)
  (tagged-list-min-length-n? exp 'begin 1))

(define (begin-expressions exp)
  (if (begin? exp)
      (rest exp)
      (println "Error: not a begin expression!")))

(define (eval-begin exp env)
  (letrec ((helper (lambda (lst)
                     (cond ((empty? lst)(void))
                           ((= (length lst) 1)(first lst))
                           (else (helper (rest lst))))))))
```

```
    (helper (map (lambda (x)(i-eval x env)) (rest exp)))))

;; if

(define (if? exp)
  (or (tagged-list-length-n? exp 'if 4)
      (tagged-list-length-n? exp 'if 3)))

(define (test-expression exp)
  (if (if? exp)
      (first (rest exp))
      (println "Error: not an if expression!")))

(define (then-expression exp)
  (if (if? exp)
      (first (rest (rest exp)))
      (println "Error: not an if expression!")))

(define (else-expression exp)
  (if (if? exp)
      (if (= (length exp) 3)
          (void)
          (first (rest (rest (rest exp)))))
      (println "Error: not an if expression!")))

(define (eval-if exp env)
  (let ((e (else-expression exp)))
    (if (i-eval (test-expression exp) env)
        (i-eval (then-expression exp) env)
        (if (void? e)
            (void)
            (i-eval e env)))))

;; cond

(define (cond? exp)
  (match exp
    ((list 'cond (? list? a) ... ) #t)
    (_ #f)))

(define (first-cond-exp exp)
  (cond ((not (cond? exp))(println "Error: not a cond!"))
        ((= (length exp) 1)(println "Error: no conditions!"))
        (else (first (rest exp)))))

(define (rest-of-cond-exp exp)
  (cond ((not (cond? exp))(println "Error: not a cond!"))
      ((= (length exp) 1)(println "Error: no conditions!"))
      (else (rest (rest exp)))))

(define (eval-cond exp env)
```

```
  (if (empty? (rest exp))
      (void)
      (let ((curr (first-cond-exp exp))
            (curr-test (first (first-cond-exp exp)))
            (curr-res (i-eval (first (first-cond-exp exp)) env)))
        (if (and (equal? 'else curr-test)
                 (> (length (rest-of-cond-exp exp)) 0))
            (println "Error: clauses following else case")
            (cond ((= (length curr) 1)(if curr-res
                                          curr-res
                                          (eval-cond (cons 'cond
(rest-of-cond-exp exp)) env)))
                  (curr-res (i-eval (second curr) env))
                  (else (eval-cond (cons 'cond (rest-of-cond-exp exp))
env)))))))

;; Week 3

(define (lambda? exp)
  (match exp
    ((list 'lambda (list _ ... ) _ ...) #t)
    (_ #f)))

(define make-closure
   (lambda (lambda-exp env)
     (list 'closure lambda-exp env)))

(define (closure? exp)
  (match exp
    ((list 'closure (list 'lambda args _ ...) env) #t)
    (_ #f)))

(define (procedure-parameters closure)
  (if (closure? closure)
      (second (second closure))
      (println "Error: not a closure!")))

(define (procedure-body closure)
  (if (closure? closure)
      (rest (rest (second closure)))
      (println "Error: not a closure!")))

(define (procedure-env closure)
  (if (closure? closure)
      (third closure)
      (println "Error: not a closure!")))

(define (apply-closure proc vals)
  (let ((f-env (extend-env (procedure-parameters proc) vals
(procedure-env proc))))
    (i-eval (cons 'begin (procedure-body proc)) f-env)))
```

```
340
341 (define (eval-application exp env)
342   (let ((operator (i-eval (operator exp) env))
343         (operands (eval-operands (operands exp) env)))
344     (i-apply operator operands)))
345
346 (define i-apply
347    (lambda (proc vals)   ;; note: these "vals" have already been evaluated
348      (cond
349       ((primitive-procedure? proc) (apply-primitive-procedure proc vals))
350       ((closure? proc)(apply-closure proc vals))
351       (else (println "Error: unknown procedure type!")))))
352
353 (define i-eval
354   (lambda (exp env)
355     (cond
356      ((boolean? exp) exp)
357      ((number? exp) exp)
358      ((string? exp) exp)
359      ((definition? exp)(eval-definition exp env))
360      ((variable? exp)(lookup-variable exp env))
361      ((begin? exp)(eval-begin exp env))
362      ((if? exp)(eval-if exp env))
363      ((cond? exp)(eval-cond exp env))
364      ((let? exp)(eval-let exp env))
365      ((lambda? exp)(make-closure exp env))
366      ((quoted? exp)(text-of-quotation exp))
367      ((map? exp)(eval-map exp env))
368      ((foldl? exp)(eval-foldl exp env))
369      ((filter? exp)(eval-filter exp env))
370      ((and? exp)(eval-and exp env))
371      ((or? exp)(eval-or exp env))
372      ((apply? exp)(eval-apply exp env))
373      ((include? exp) (eval-include exp env))
374      ((application? exp)(eval-application exp env))
375      (else (begin (println "Error: unknown expression type"))))))
376
377 ;; i-print: give to students
378
379 (define (i-print exp)
380   (match exp
381     ((? void? exp)(void ))
382     ((? closure? exp)(println (procedure-body exp)))
383     ((? primitive-procedure? exp)(println (primitive-name exp)))
384     (_ (println exp))))
385
386 ;; let
387
388 (define (let? exp)
389   (match exp
390     ((list 'let (list (list x y) ...) rest) #t)
```

```
391     (_ #f)))
392
393 (define (let->lambda l)
394   (if (empty? (second l))
395       (list (list 'lambda (list )(third l)) (list))
396       (let ((vars (map (lambda (x)(first x)) (second l)))
397             (vals (map (lambda (x)(second x)) (second l)))
398             (exps (third l)))
399         (append (list (list 'lambda vars exps)) vals)))) ;ugh this is
399 very tricky
400
401 (define (eval-let exp env)
402   (i-eval (let->lambda exp) env))
403
404 (define (repl)
405   (read-eval-print-loop))
406
407 (define (map? exp)
408   (tagged-list-length-n? exp 'map 3))
409
410 (define (eval-map exp env)
411   (let ((func (second exp))
412         (lst (i-eval (third exp) env)))
413     (letrec ((helper (lambda (f l)
414                        (if (empty? l)
415                            (list )
416                            (cons (i-eval (list f (first l)) env)
417                                  (helper f (rest l)))))))
418       (helper func lst))))
419
420 (define (filter? exp)
421   (tagged-list-length-n? exp 'filter 3))
422
423 (define (eval-filter exp env)
424   (let ((func (second exp))
425         (lst (i-eval (third exp) env)))
426     (letrec ((helper (lambda (f l)
427                        (if (empty? l)
428                            (list )
429                            (if (i-eval (list f (first l)) env)
430                                (cons (first l) (helper f (rest l)))
431                                (helper f (rest l)))))))
432       (helper func lst))))
433
434 (define (foldl? exp)
435   (tagged-list-length-n? exp 'foldl 4))
436
437 (define (eval-foldl exp env)
438   (let ((func (second exp))
439         (init (third exp))
440         (lst (i-eval (fourth exp) env)))
```

```
    (letrec ((helper (lambda (f res l)
                       (if (empty? l)
                           res
                           (helper f (list f (first l) res) (rest l))))))
      (i-eval (helper func init lst) env)))) ; this is pretty subtle

(define (and? exp)
  (tagged-list-min-length-n? exp 'and 2))

(define (or? exp)
  (tagged-list-min-length-n? exp 'or 2))

(define (eval-or exp env)
  (letrec ((helper (lambda (clauses)
                     (if (= (length clauses) 0)
                         #f
                         (let ((res (i-eval (first clauses) env)))
                           (if res
                               res
                               (helper (rest clauses))))))))
    (helper (rest exp))))

(define (eval-and exp env)
  (letrec ((helper (lambda (clauses)
                     (if (= (length clauses) 1)
                         (i-eval (first clauses) env)
                         (let ((res (i-eval (first clauses) env)))
                           (if res
                               (helper (rest clauses))
                               #f))))))
    (helper (rest exp))))

(define (eval-apply exp env)
  (i-eval (cons (second exp) (i-eval (third exp) env)) env))

(define (apply? exp)
  (tagged-list-length-n? exp 'apply 3))

(define (include? exp)
  (tagged-list-length-n? exp 'include 2))

(define (split str c)
  (letrec ((chars (map (lambda (x) (string x)) (string->list str))))
  (reverse (foldl (lambda (x res)
         (if (equal? c x)
             (cons "" (cons (first res) (rest res)))
             (cons (string-append (first res) x) (rest res))))
       (list "")
       chars))))

;; Helpful for working towards meta-circularity:
```

```
492
493 (define (include-helper line)
494   (map (lambda (x)(string->symbol x)) (split line " ")))
495
496 (define (eval-include exp env)
497   (let ((infile (open-input-file (i-eval (second exp) env))))
498     (letrec ((helper (lambda ()
499                        (let ((line (read infile)))
500                          (cond ((eof-object? line)(void))
501                                (else (println line)
502                                      (i-print (i-eval line env))
503                                      (newline)
504                                      (helper)))))))
505       (helper))))
506
507 (define read-eval-print-loop
508   (lambda ()
509     (display "INTERPRETER> ")
510     (let ((user-input (read)))
511       (cond ((equal? user-input 'exit)(void))
512             ((equal? user-input 'exit!)(reset-global))
513             (else (i-print (i-eval user-input global-env))
514                   (newline)
515                   (read-eval-print-loop))))))
```