CSSE 304 Day 19 Summary (first we will finish the day 18 material on environment representations)

1. What is the result of executing this code?

```
> (define a 0)
> (map (lambda (x)
          (set! a (add1 a))
          a)
       '(a b c d e f g h i j k l m n o p))
```

2. Main pieces of the initial interpreter (code is on daily handout). This is a place for your notes about the procedures.

Procedure	What it does / how it works
(rep)	
(apply-env env symbol succeed fail)	
(top-level-eval parsed-form)	
(eval-exp exp)	
(eval-rands rands)	
(apply-proc proc-val args)	
(apply-prim-proc prim-proc args)	
(define init-env …	

3. Before adding **let** and **lambda** to interpreted language: Add a local **env** argument to several procedures (especially **eval-exp**) (code details are in the slides).
4. Do we need to add an **env** argument to **apply-proc**? Why or why not?

5. Add **let**, **if**, **lambda** to the interpreted language

```
(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record
   (syms (list-of symbol?))
   (vals (list-of scheme-value?))
   (env environment?)))

;; Parsed expression datatypes
(define-datatype expression expression?
  [var-exp        ; variable references
   (id symbol?)]
  [lit-exp        ; "Normal" data.
   (datum         ; Did I leave out any types?
    (lambda (x)
      (ormap
       (lambda (pred) (pred x))
       (list number? vector? boolean?
             symbol? string? pair? null?)))))]
  [app-exp        ; applications
   (rator expression?)
   (rands (list-of expression?))]
  )

; datatype for procedures.  At first only one
; kind of procedure, more kinds added later.
(define-datatype proc-val proc-val?
  [prim-proc
   (name symbol?)])

; Environment definitions. Based on EoPL section 2.3
(define empty-env
  (lambda ()
    (empty-env-record)))

(define extend-env
  (lambda (syms vals env)
    (extended-env-record syms vals env)))

(define list-find-position
  (lambda (sym los)
    (list-index (lambda (xsym) (eqv? sym xsym)) los)))

(define list-index
  (lambda (pred ls)
    (cond
     ((null? ls) #f)
     ((pred (car ls)) 0)
     (else (let ((list-index-r (list-index pred (cdr
ls))))
             (if (number? list-index-r)
                 (+ 1 list-index-r)
                 #f))))))

(define apply-env
  (lambda (env sym succeed fail) ; succeed and fail are
                                 ; applied if the var is or
                                 ; isn't found, respectively.
    (cases environment env
      (empty-env-record ()
        (fail))
      (extended-env-record (syms vals env)
        (let ((pos (list-find-position sym syms)))
          (if (number? pos)
              (succeed (list-ref vals pos))
              (apply-env env sym succeed fail)))))))

;  parse and evaluate a single expression
; (non-interactive interface)

(define eval-one-exp
  (lambda (x) (top-level-eval (parse-exp x))))

; top-level-eval evaluates a form in the global
environment

(define top-level-eval
  (lambda (parsed-form)
    ; later we may add things that are not expressions.
    (eval-exp parsed-form)))

; eval-exp is the main component of the interpreter

(define eval-exp
  (lambda (exp)
    (cases expression exp
      [lit-exp (datum) datum]
      [var-exp (id)
        (apply-env init-env id    ; look up its value.
          (lambda (x) x) ; if id is in the environment
          (lambda (); procedure to call if id not in env
             (eopl:error 'apply-env
                  "variable not found in environment: ~s"
                  id)))]
      [app-exp (rator rands)
        (let ([proc-value (eval-exp rator)]
              [args (eval-rands rands)])
          (apply-proc proc-value args))]
      [else
        (eopl:error 'eval-exp
                    "Bad abstract syntax: ~a"
                    exp)])))

; evaluate list of operands, putting results into a list
(define eval-rands
  (lambda (rands)
    (map eval-exp rands)))

;  Apply a procedure to its arguments (values, not code).
;  At this point, we only have primitive procedures.
;  User-defined procedures will be added later.

(define apply-proc
  (lambda (proc-value args)
    (cases proc-val proc-value
      [prim-proc (op) (apply-prim-proc op args)]
                      ; You will add other cases
      [else (error 'apply-proc
                   "Attempt to apply bad procedure: ~s"
                    proc-value)])))

(define *prim-proc-names* '(+ - * add1 sub1 cons =))

(define init-env  ; for now, initial global environment
                  ; only contains procedure names.
  (extend-env     ; Recall that an environment associates
     *prim-proc-names*   ;  a value (not an expression)
                         ; with an identifier.
     (map prim-proc
          *prim-proc-names*)
     (empty-env)))

; Usually an interpreter must define each built-in
; procedure individually.  Defined using the facilities
; provided by the implementation language.

(define apply-prim-proc
  (lambda (prim-proc args)
    (case prim-proc
      [(+) (+ (1st args) (2nd args))]
      [(-) (- (1st args) (2nd args))]
      [(*) (* (1st args) (2nd args))]
      [(add1) (+ (1st args) 1)]
      [(sub1) (- (1st args) 1)]
      [(cons) (cons (1st args) (2nd args))]
      [(=) (= (1st args) (2nd args))]
      [else (error 'apply-prim-proc
            "Bad primitive procedure name: ~s"
            prim-op)])))

; ---------  interactive interpreter interface -------
(define rep      ; "read-eval-print" loop.
  (lambda ()
    (display "--> ")
    ;; note that we don't change the environment yet
    (let ([answer
           (top-level-eval (parse-exp (read)))])
      ;; TODO: are there answers that should
      ;;       display differently?
      (eopl:pretty-print answer)
      (newline)
      (rep))))  ; tail-recursive, so stack doesn't grow.
```