

Not having a good time with the mathpartir package here.

Exercise 4.1

Calls to `g` declare a reference `counter`, set its contents, then return its contents. The information between calls is lost because `counter` is different on each call. That is, the reference data structure represents a different location. The first program was made such that the environment of the closure had access to `counter`, and thus was able to reference that variable.

Exercise 4.2

$$\frac{(\text{value-of } exp_1 \rho \sigma_0) = (val_1, \sigma_1)}{(\text{value-of } (\text{zero?-exp } exp_1) \rho \sigma_0) = \\ ((\text{bool-val } \#t), \sigma_1) \text{ if } [val_1] = 0 \\ ((\text{bool-val } \#f), \sigma_1) \text{ if } [val_1] \neq 0}$$

Exercise 4.3

$$\frac{\begin{array}{l} (\text{value-of } exp_1 \rho \sigma_0) = (val_1, \sigma_1) \\ (\text{value-of } exp_2 \rho \sigma_1) = (val_2, \sigma_2) \end{array}}{(\text{value-of } (\text{call-exp } exp_1 exp_2) \rho \sigma_0) \\ = (\text{apply-procedure } (\text{expval->proc } val_1) val_2 \sigma_2)}$$
$$\frac{val_1 = (\text{procedure } var \ body \rho)}{(\text{apply-procedure } val_1 val_2 \rho_0) \\ = (\text{value-of } body [var = val_2] \rho \sigma_0)}$$

Exercise 4.4

$$\frac{\begin{array}{c} (\text{value-of } exp_1 \rho \sigma_0) = (val_1, \sigma_1) \\ \vdots \\ (\text{value-of } exp_n \rho \sigma_{n-1}) = (val_n, \sigma_n) \end{array}}{(\text{value-of } (\text{begin } exp_1 \dots exp_n) \rho \sigma_0) = (val_n, \sigma_n)}$$

Exercise 4.5

$$\frac{\begin{array}{c} (\text{value-of } exp_1 \rho \sigma_0) = (val_1, \sigma_1) \\ \vdots \\ (\text{value-of } exp_n \rho \sigma_{n-1}) = (val_n, \sigma_n) \end{array}}{\begin{array}{l} (\text{value-of } (\text{list-exp } exp_1 \dots exp_n) \rho \sigma_0) \\ = ((\text{pair-val } val_1 (\dots (\text{pair-val } val_n (\text{emptylist-val})) \dots)), \sigma_n) \end{array}}$$

Exercise 4.6

$$\frac{\begin{array}{c} (\text{value-of } exp_1 \rho \sigma_0) = (l, \sigma_1) \\ (\text{value-of } exp_2 \rho \sigma_1) = (val, \sigma_2) \end{array}}{(\text{value-of } (\text{setref-exp } exp_1 exp_2) \rho \sigma_0) = (val, [l=val]\sigma_2)}$$

Exercise 4.7

$$\frac{\begin{array}{c} (\text{value-of } exp_1 \rho \sigma_0) = (l, \sigma_1) \\ (\text{value-of } exp_2 \rho \sigma_1) = (val, \sigma_2) \end{array}}{(\text{value-of } (\text{setref-exp } exp_1 exp_2) \rho \sigma_0) = (\sigma_0(l), [l=val]\sigma_2)}$$

Exercise 4.8

`newref`, `deref`, and `setref!` take linear time.

Exercise 4.9

`newref` uses new-store-longer-by-one which takes linear time because I could not find a built-in procedure for copying vectors. `deref` takes as much time as `vector-ref`. `setref!` takes as much time as `vector-length`.

```
(define empty-store
  (lambda ()
    (make-vector 0)))

;; initialize-store! : () -> Sto
;; usage: (initialize-store!) sets the-store to the empty-store
(define initialize-store!
  (lambda ()
    (set! the-store (empty-store))))

;; reference? : SchemeVal -> Bool
(define reference?
  (lambda (v)
    (integer? v)))

;; new-store-longer-by-one : Sto -> Sto
(define new-store-longer-by-one
  (lambda (store)
    (let ((new-store (make-vector (+ 1 (vector-length store)))))
      (letrec ((inner
                (lambda (current-index stop)
                  (if (= current-index stop)
                      new-store
                      (begin (vector-set!
                               new-store
                               current-index
```

```

(vector-ref store current-index))
  (inner (+ current-index 1) stop))))))
  (inner 0 (vector-length store)))))

;; newref : ExpVal -> Ref
(define newref
  (lambda (val)
    (let* ((next-ref (vector-length the-store))
           (new-store (new-store-longer-by-one the-store)))
      (vector-set! new-store next-ref val)
      (set! the-store new-store)
      (when (instrument-newref)
        (eopl:printf
          "newref: allocating location ~s with initial contents ~s~"
          next-ref val)
        next-ref)))))

;; deref : Ref -> ExpVal
(define deref
  (lambda (ref)
    (vector-ref the-store ref)))

;; setref! : Ref * ExpVal -> Unspecified
(define setref!
  (lambda (ref val)
    (if (> (vector-length the-store) ref)
        (vector-set! the-store ref val)
        (report-invalid-reference ref the-store))))
```

Exercise 4.10

```

;; the-grammar
(expression
 ("begin" expression (arbno ";" expression) "end")
```

```
begin-exp)

;; value-of
(begin-exp (expl rest-exps)
  (letrec ((begin-inner
    (lambda (exp1 rest-exps)
      (if (null? exp1)
        rest-exps
        (begin-inner (cdr exp1)
          (value-of (car exp1) env)))))))
  (begin-inner rest-exps (value-of expl env))))
```

Exercise 4.11

```

;; the-grammar

(expression
 ("list" "(" (separated-list expression ",") ")")
 list-exp)

;; value-of

(list-exp (expss)
  (letrec ((list-inner
    (lambda (expss)
      (if (null? expss)
        (emptylist-val)
        (pair-val (value-of (car expss) env)
          (list-inner (cdr expss)))))))
    (list-inner expss)))

(define-datatype expval expval?
  (num-val
    (value number?))
  (bool-val
    )
  )

```

```

(boolean boolean?))
(proc-val
  (proc proc?))
(ref-val
  (ref reference?))
(pair-val
  (val1 expval?))
  (val2 expval?))
(emptylist-val))

```

Exercise 4.12

The fragment of the interpreter given in the book does not have enough context. The meaning of `apply-store` is unclear and `deref` takes only one argument which makes the program look like the store is still a global variable.

```

;; value-of-program : Program -> ExpVal
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (cases answer (value-of exp1
          (init-env)
          (init-store)))
        (an-answer (val store)
          val))))))

;; value-of : Exp * Env -> Answer
(define value-of
  (lambda (exp env store)
    (cases expression exp
      (const-exp (num) (an-answer (num-val num) store))
      (var-exp (var)
        (an-answer (apply-env env var))))))

```

```

                store))

(diff-exp (exp1 exp2)
           (cases answer (value-of exp1 env store)
                 (an-answer (v1 store1)
                           (cases answer (value-of exp2
                                         env
                                         store1)
                                 (an-answer (v2 store2)
                                           (an-answer
                                             (num-val
                                               (- (expval->num v1)
                                                 (expval->num v2)))
                                             store2)))))

(zero?-exp (exp1)
            (cases answer (value-of exp1 env store)
                  (an-answer (vall store1)
                            (an-answer
                              (if (zero? (expval->num vall))
                                  (bool-val #t)
                                  (bool-val #f))
                                store1)))))

(if-exp (exp1 exp2 exp3)
        (cases answer (value-of exp1 env store)
              (an-answer (vall store1)
                        (if (expval->bool vall)
                            (value-of exp2 env store1)
                            (value-of exp3 env store1)))))

(let-exp (var exp1 body)
         (cases answer (value-of exp1 env store)
               (an-answer (vall store1)
                         (value-of body
                                   (extend-env var vall env))))
```

```

                                store1)))))

(proc-exp (var body)
           (an-answer (proc-val (procedure var body env))
                      store))

(call-exp (rator rand)
           (cases answer (value-of rator env store)
                  (an-answer (proc-val store1)
                             (cases answer (value-of rand
                                                       env
                                                       store1)
                                         (an-answer (arg store2)
                                         (apply-procedure
                                         (expval->proc proc-val)
                                         arg
                                         store2))))))

(letrec-exp (p-names b-vars p-bodies letrec-body)
           (value-of letrec-body
                     (extend-env-rec* p-names
                                     b-vars
                                     p-bodies
                                     env)
                     store))

(begin-exp (expl exps)
           (letrec
              ((begin-inner
                 (lambda (e1 es store)
                   (cases answer (value-of e1 env store)
                          (an-answer (v1 store1)
                                     (if (null? es)
                                         (an-answer v1 store1)
                                         (begin-inner
                                           (car es)))))))))))

```

```

(cdr es)
store1))))))

(begin-inner expl exps store)))
(newref-exp (expl)
(cases answer (value-of expl env store)
(an-answer (vall store1)
(let ((newref-pair
(newref vall store1)))
(let ((ref (car newref-pair))
(store2 (cdr newref-pair)))
(an-answer (ref-val ref)
store2)))))

(deref-exp (expl)
(cases answer (value-of expl env store)
(an-answer (vall store1)
(let ((ref1 (expval->ref vall)))
(an-answer (deref store1 ref1)
store1)))))

(setref-exp (expl exp2)
(cases answer (value-of expl env store)
(an-answer (vall store1)
(cases answer (value-of exp2
env
store1)
(an-answer (val2 store2)
(an-answer
(num-val 23)
(setref
(expval->ref val1)
val2
store2)))))))))))

```

```

;; apply-procedure : Proc * ExpVal * Sto -> ExpVal

;; uninstrumented version
(define apply-procedure
  (lambda (proc1 arg store)
    (cases proc proc1
      (procedure (bvar body saved-env)
        (value-of body
          (extend-env bvar arg saved-env)
          store)))))

;; instrumented version
(define apply-procedure
  (lambda (proc1 arg store)
    (cases proc proc1
      (procedure (var body saved-env)
        (let ((r arg))
          (let ((new-env (extend-env var r saved-env)))
            (when (instrument-let)
              (begin
                (eopl:printf
                  "entering body of proc ~s with env =~%"
                  var)
                (pretty-print (env->list new-env))
                (eopl:printf "store =~%")
                (pretty-print
                  (store->readable
                    (get-store-as-list store))))
                (eopl:printf "~~%")))
              (value-of body new-env store)))))))

```

;; empty-store : () -> Sto

```

(define empty-store
  (lambda () '()))

;; init-store : () -> Sto
(define init-store
  (lambda ()
    (empty-store)))

;; newref : ExpVal * Sto -> Cons(Ref, Sto)
(define newref
  (lambda (val store)
    (let ((next-ref (length store)))
      (when (instrument-newref)
        (eopl:printf
          "newref: allocating location ~s with initial contents ~s~"
          next-ref val))
      (cons next-ref (append store (list val))))))

;; deref : Ref * Sto -> ExpVal
(define deref
  (lambda (store ref)
    (list-ref store ref)))

;; setref : Ref * ExpVal * Sto -> Sto
(define setref
  (lambda (ref val store)
    (letrec ((setref-inner
              (lambda (r s)
                (cond ((null? s)
                       (report-invalid-reference r store))
                      ((zero? r)
                       (cons val (cdr s))))))

```

```

        (else
         (cons
          (car s)
          (setref-inner (- r 1) (cdr s)))))))
      (setref-inner ref store)))))

(define report-invalid-reference
  (lambda (ref the-store)
    (eopl:error 'setref
                "illegal reference ~s in store ~s"
                ref the-store)))

;; get-store-as-list : Sto -> Listof(List(Ref,Expval))
(define get-store-as-list
  (lambda (store)
    (letrec
      ((inner-loop
        (lambda (sto n)
          (if (null? sto)
              '()
              (cons
               (list n (car sto))
               (inner-loop (cdr sto) (+ n 1)))))))
      (inner-loop store 0)))))

(define-datatype answer answer?
  (an-answer
   (val expval?))
  (store store?)))

;; store? : Schemeval -> Bool
(define store?

```

```
(lambda (scmval)
  ((list-of expval?) scmval)))
```

Exercise 4.13

```
; ; the-grammar
(expression
 ("proc" "(" (separated-list identifier ",") ")" expression)
 proc-exp)

(expression
 ("(" expression (arbno expression) ")")
 call-exp)

(expression
 ("letrec"
 (arbno identifier "(" (separated-list identifier ",") ")"
 "=" expression)
 "in" expression)
 letrec-exp)

; ; value-of
(call-exp (rator rands)
(letrec
  ; ; Listof(Exp) * Nil * Sto -> Cons(ExpVals, Sto)
  ((call-inner
    (lambda (expvals store)
      (if (null? exps)
          (cons (reverse vals) store)
          (cases answer (value-of (car exps) env store)
            (an-answer (vall store1)
              (call-inner (cdr exps)
                (cons vall vals
                  store1))))))))
```

```

(cases answer (value-of rator env store)
  (an-answer (value-of-rator store1)
    (let ((args/store-pair
           (call-inner rands '() store1)))
      (let ((args (car args/store-pair))
            (new-store (cdr args/store-pair)))
        (apply-procedure
         (expval->proc value-of-rator)
         args
         new-store))))))

;; apply-procedure : Proc * Listof(ExpVal) * Store -> ExpVal
;; uninstrumented version
(define apply-procedure
  (lambda (proc1 args store)
    (cases proc proc1
      (procedure (bvars body saved-env)
        (value-of body
          (extend-env* bvars args saved-env)
          store)))))

(define-datatype environment environment?
  (empty-env)
  (extend-env
   (bvar symbol?))
  (bval expval?))
  (saved-env environment?))
  (extend-env*
   (bvars (list-of symbol?))
   (bvals (list-of expval?))
   (saved-env environment?))
  (extend-env-rec*

```

```

(proc-names (list-of symbol?))
(b-vars (list-of (list-of symbol?)))
(proc-bodies (list-of expression?))
(saved-env environment?))

(define apply-env
  (lambda (env search-sym)
    (cases environment env
      (empty-env ())
        (eopl:error 'apply-env
                    "No binding for ~s"
                    search-sym)
      (extend-env (bvar bval saved-env)
                  (if (eqv? search-sym bvar)
                      bval
                      (apply-env saved-env search-sym)))
      (extend-env* (bvars bvals saved-env)
                   (cond ((location search-sym bvars)
                           => (lambda (n)
                                 (list-ref bvals n)))
                         (else (apply-env saved-env search-sym))))
      (extend-env-rec* (p-names b-vars p-bodies saved-env)
                      (cond
                        ((location search-sym p-names)
                         => (lambda (n)
                               (proc-val
                                 (procedure
                                   (list-ref b-vars n)
                                   (list-ref p-bodies n)
                                   env))))))
                      (else (apply-env saved-env search-sym))))
```

Exercise 4.14

$$\frac{(\text{value-of } exp_1 \rho \sigma_0) = (val_1, \sigma_1) \quad l \notin \text{dom}(\sigma_1)}{(\text{value-of } (\text{let-exp } var \ exp_1 \ body) \rho \sigma_0) \\ = (\text{value-of } body \ [var = l]\rho \ [l = val_1]\sigma_1)}$$

Exercise 4.15

Variables in the environment are bound to references which are plain integers.

Exercise 4.16

Assume the initial environment p is empty.

```
(value-of <let times 4 = 0 in begin set..> p)
store = ()

(value-of <begin set..> [times4=0]p)
store = ((0 (num-val 0)))

(value-of <(times4 3)> [times4=0]p)
store = ((0 (proc-val (<proc (x)..> [times4=0]p)))))

(apply-procedure (proc-val <proc (x)..>) (num-val 3))
store = ((0 (proc-val (<proc (x)..> [times4=0]p)))
         (1 (num-val 3)))

(value-of <if zero(x)..> [x=1][times4=0]p)
store = ((0 (proc-val (<proc (x)..> [times4=0]p)))
         (1 (num-val 3)))

(value-of <-((times4 -(x,1)), -4)> [x=1][times4=0]p)
store = ((0 (proc-val (<proc (x)..> [times4=0]p)))
         (1 (num-val 3)))
```

```

(apply-procedure (proc-val <proc (x)..>) (num-val 2))
store = ((0 (proc-val (<proc (x)..> [times4=0]p)))
          (1 (num-val 3)))

(value-of <if zero(x)..> [x=2][times4=0]p)
store = ((0 (proc-val (<proc (x)..> [times4=0]p)))
          (1 (num-val 3))
          (2 (num-val 2)))

(value-of <-((times4 (-x,1)), -4)>> [x=2][times4=0]p)
store = ((0 (proc-val (<proc (x)..> [times4=0]p)))
          (1 (num-val 3))
          (2 (num-val 2)))

(apply-procedure (proc-val <proc (x)..>) (num-val 1))
store = ((0 (proc-val (<proc (x)..> [times4=0]p)))
          (1 (num-val 3))
          (2 (num-val 2)))

(value-of <if zero(x)..> [x=3][times4=0]p)
store = ((0 (proc-val (<proc (x)..> [times4=0]p)))
          (1 (num-val 3))
          (2 (num-val 2))
          (3 (num-val 1)))

(value-of <-((times4 -(x,1)), -4)> [x=3][times4=0]p)
store = ((0 (proc-val (<proc (x)..> [times4=0]p)))
          (1 (num-val 3))
          (2 (num-val 2))
          (3 (num-val 1)))

(apply-procedure (proc-val <proc (x)..>) (num-val 0))

```

```

store = ((0 (proc-val (<proc (x)...> [times4=0]p)))
         (1 (num-val 3))
         (2 (num-val 2))
         (3 (num-val 1)))

(value-of <if zero(x)...> [x=4][times4=0]p)
store = ((0 (proc-val (<proc (x)...> [times4=0]p)))
         (1 (num-val 3))
         (2 (num-val 2))
         (3 (num-val 1))
         (4 (num-val 0)))

- (- (- (0, -4), -4), -4)

- (- (4, -4), -4)

- (8, -4)

```

12

Exercise 4.17

$$\begin{aligned}
 (\text{value-of } exp_1 \rho \sigma_0) &= (val_1, \sigma_1) \\
 &\vdots \\
 (\text{value-of } exp_n \rho \sigma_{n-1}) &= (val_n, \sigma_n) \\
 l_1, \dots, l_n &\notin \text{dom}(\sigma_n)
 \end{aligned}
 \hrule
 \begin{aligned}
 &(\text{value-of } (\text{let-exp } var_1 \dots var_n exp_1 \dots exp_n body) \rho \sigma_0) \\
 &= (\text{value-of } body [var_1 = l_1] \dots [var_n = l_n] \rho [l = val_1] \dots [l_n = val_n] \sigma_n)
 \end{aligned}$$

$$\begin{array}{c}
 (\text{value-of } exp_1 \rho \sigma_0) = (val_1, \sigma_1) \\
 (\text{value-of } exp_2 \rho \sigma_1) = (val_2, \sigma_2) \\
 \vdots \\
 (\text{value-of } exp_n \rho \sigma_{n-1}) = (val_n, \sigma_n)
 \end{array}
 \frac{}{\text{(value-of (call-exp } exp_1 exp_2 \dots exp_n) \rho \sigma_0) \\
 = (\text{apply-procedure (expval->proc } val_1) val_2 \dots val_n \sigma_n)}$$

$$\begin{array}{c}
 (\text{apply-procedure (procedure } var_1 \dots var_n body \rho) val_1 \dots val_n \sigma) \\
 = (\text{value-of } body [var_1 = l_1] \dots [var_n = l_n] \rho [l_1 = val_1] \dots [l_n = val_n] \sigma)
 \end{array}$$

where $l_1, \dots, l_n \notin \text{dom}(\sigma)$.

```

;; the-grammar
(expression
 ("let" (arbno identifier "=" expression) "in" expression)
 let-exp)

(expression
 ("proc" "(" (arbno identifier) ")" expression)
 proc-exp)

(expression
 ("(" expression (arbno expression) ")")
 call-exp)

;; value-of
(let-exp (vars exps body)
  (let ((vals (map (lambda (exp)
    (value-of exp env) )

```

```

        exps)))
      (value-of body
      (extend-env* vars
      (map newref vals)
      env)))))

(proc-exp (vars body)
  (proc-val (procedure vars body env)))
(call-exp (rator rands)
  (let ((proc (expval->proc (value-of rator env)))
    (args (map (lambda (rand)
      (value-of rand env))
    rands)))
  (apply-procedure proc args)))

;; apply-procedure : Proc * Listof(ExpVal) -> ExpVal
(define apply-procedure
  (lambda (proc1 vals)
    (cases proc proc1
      (procedure (vars body saved-env)
        (value-of
          body
        (extend-env* vars
        (map newref vals)
        saved-env))))))

(define-datatype proc proc?
  (procedure
    (bvar (list-of symbol?))
    (body expression?))
  (env environment?)))

;; environment datatype

```

```
(extend-env*
  (bvars (list-of symbol?))
  (bvals (list-of reference?))
  (saved-env environment?))

;; apply-env
(extend-env* (bvars bvals saved-env)
  (let ((n (location search-var bvars)))
    (if n
        (list-ref bvals n)
        (apply-env saved-env search-var))))
```