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

$$\frac{\texttt{(value-of}\ \ exp_1\ \ \rho\ \ \sigma_0\texttt{)} = \texttt{(}val_1\texttt{,}\ \sigma_1\texttt{)}}{\begin{array}{c}\texttt{(value-of (zero?-exp}\ \ exp_1\texttt{)}\ \ \rho\ \ \sigma_0\texttt{)} = \\ \texttt{((bool-val \#t),}\ \sigma_1\texttt{)}\ \ \text{if}\ \ \lfloor val_1 \rfloor = 0 \\ \texttt{((bool-val \#f),}\ \sigma_1\texttt{)}\ \ \text{if}\ \ \lfloor val_1 \rfloor \neq 0\end{array}}$$

Exercise 4.3

$$\frac{\begin{array}{c}\texttt{(value-of}\ \ exp_1\ \ \rho\ \ \sigma_0\texttt{)} = \texttt{(}val_1\texttt{,}\ \sigma_1\texttt{)} \\ \texttt{(value-of}\ \ exp_2\ \ \rho\ \ \sigma_1\texttt{)} = \texttt{(}val_2\texttt{,}\ \sigma_2\texttt{)}\end{array}}{\begin{array}{c}\texttt{(value-of (call-exp}\ \ exp_1\ \ exp_2\texttt{)}\ \ \rho\ \ \sigma_0\texttt{)} \\ = \texttt{(apply-procedure (expval->proc}\ \ val_1\texttt{)}\ \ val_2\ \ \sigma_2\texttt{)}\end{array}}$$

$$\frac{val_1 = \texttt{(procedure}\ \ var\ \ body\ \ \rho\texttt{)}}{\begin{array}{c}\texttt{(apply-procedure}\ \ val_1\ \ val_2\ \ \rho_0\texttt{)} \\ = \texttt{(value-of}\ \ body\ \ [var = val_2]\rho\ \ \sigma_0\texttt{)}\end{array}}$$

Exercise 4.4

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

Exercise 4.5

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

Exercise 4.6

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

Exercise 4.7

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

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 (exp1 rest-exps)
           (letrec ((begin-inner
                     (lambda (exps final-val)
                       (if (null? exps)
                           final-val
                           (begin-inner (cdr exps)
                                        (value-of (car exps)
                                                  env))))))
             (begin-inner rest-exps (value-of exp1 env))))
```

Exercise 4.11

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

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

(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))
```