Solutions to EoPL3 Exercises

Release 0.1.0

Cheng Lian

May 16, 2017

Contents

Author Cheng Lian <lian.cs.zju@gmail.com>

CHAPTER 1

Contents

Chapter 1. Inductive Sets of Data

Exercise 1.1

1. $\{3n + 2 \mid n \in N\}$

   Top-down: A natural number n is in S if and only if

   (a) $n = 2$, or

   (b) $n - 3 \in S$.

   Bottom-up: Define the set S to be the smallest set contained in N and satisfying the following two properties:

   (a) $2 \in S$

   (b) if $n \in S$, then $n + 3 \in S$

   Rules of inference:

$$2 \in S$$

$$\frac{n \in S}{n + 3 \in S}$$

2. $\{2n + 3m + 1 \mid n, m \in N\}$

   Top-down: A natural number n is in S if and only if

   (a) $n = 1$, or

   (b) $n - 2 \in S$, or

   (c) $n - 3 \in S$

   Bottom-up: Define the set S to be the smallest set contained in N and satisfying the following two properties:

   (a) $1 \in S$

(b) if $n \in S$, then $n + 2 \in S$ and $n + 3 \in S$

Rules of inference:

$$1 \in S$$

$$\frac{n \in S}{n + 2 \in S, n + 3 \in S}$$

3. $\{(n, 2n + 1) \mid n \in N\}$

Top-down: A pair of natural numbers (n, m) is in S if and only if

(a) $n = 0$ and $m = 1$, or

(b) $(n - 1, m - 2) \in S$

Bottom-up: Define the set S to be the smallest set contained in $\{n, m \mid n \in N, m \in N\}$ and satisfying the following two properties:

(a) $(0, 1) \in S$

(b) if $(n, m) \in S$, then $(n + 1, m + 2) \in S$

Rules of inference:

$$(0, 1) \in S$$

$$\frac{(n, m) \in S}{(n + 1, m + 2) \in S}$$

4. $\{(n, n^2 \mid n \in N)\}$

Top-down: A pair of natural numbers (n, m) is in S if and only if

(a) $n = 0$, and $m = 0$, or

(b) $(n - 1, m - 2n + 1) \in S$

Bottom-up: Define the set S to be the smallest set contained in $\{n, m \mid n \in N, m \in N\}$ and satisfying the following two properties:

(a) $(0, 0) \in S$

(b) if $(n, m) \in S$, then $(n + 1, m + 2n + 1) \in S$

Rules of inference:

$$(0, 0) \in S$$

$$\frac{(n, m) \in S}{(n + 1, m + 2n + 1) \in S}$$

Exercise 1.2

1. $(0, 1) \in S \quad \dfrac{(n, k) \in S}{(n + 1, k + 7) \in S}$

$$\{(n, 7n + 1) \mid n \in N\}$$

2. $(0,1) \in S \quad \dfrac{(n,k) \in S}{(n+1,2k) \in S}$

$$\{(n, 2^n) \mid n \in N\}$$

3. $(0,0,1) \in S \quad \dfrac{(n,i,j) \in S}{(n+1,j,i+j) \in S}$

$$\{(n, F(n), F(n+1)) \mid n \in N\}$$

where $F(n)$ is defined as

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

4. $(0,1,0) \in S \quad \dfrac{(n,i,j) \in S}{(n+1,i+2,i+j) \in S}$

$$\{(n, 2n+1, n^2 \mid n \in N\}$$

Proof: Let (n_k, i_k, j_k) be the k-th element of S generated by the rules of inference. Then we have:

$$\begin{aligned} n_0 &= 0 \\ n_k &= n_{k-1} + 1 = n_{k-1} + 1 \times 1 \\ &= n_{k-2} + 1 + 1 = n_{k-2} + 2 \times 1 \\ &= n_{k-k} + 1 + \cdots + 1 = n_{k-k} + k \times 1 \\ &= n_0 + k \times 1 \\ &= k \end{aligned}$$

and

$$\begin{aligned} i_0 &= 1 \\ i_k &= i_{k-1} + 2 = i_{k-1} + 1 \times 2 \\ &= i_{k-2} + 2 + 2 = i_{k-1} + 2 \times 2 \\ &= i_{k-k} + 2 + \cdots + 2 = i_{k-k} + k \times 2 \\ &= i_0 + 2k \\ &= 2k + 1 \end{aligned}$$

and

$$\begin{aligned} j_0 &= 0 \\ j_k &= i_{k-1} + j_{k-1} \\ &= 2(k-1) + 1 + j_{k-1} \\ &= 2(k-1) + 1 + i_{k-2} + j_{k-2} \\ &= 2(k-1) + 1 + 2(k-2) + 1 + j_{k-2} \\ &= 2(k-1) + 1 + 2(k-2) + 1 + \cdots + 2(k-k) + 1 + j_{k-k} \\ &= \left(\sum_{x=1}^{k} 2(k-x) + 1 \right) + j_0 \\ &= 2\left(k^2 - \frac{(1+k)k}{2} \right) + k \\ &= 2k^2 - (1+k)k + k \\ &= 2k^2 - k - k^2 + k \\ &= k^2 \end{aligned}$$

Thus we have

$$S = \{(n, 2n+1, n^2) \mid n \in N\}$$

Q.E.D.

Exercise 1.3

$$0 \in T$$

$$1 \in T$$

$$\frac{n \in T}{n+3 \in T}$$

Exercise 1.4

```
    List-of-Int
=>  (Int . List-of-Int)
=>  (-7 . List-of-Int)
=>  (-7 . (Int . List-of-Int))
=>  (-7 . (3 . List-of-Int))
=>  (-7 . (3 . (Int . List-of-Int)))
=>  (-7 . (3 . (14 . List-of-Int)))
=>  (-7 . (3 . (14 . ())))
```

Exercise 1.5

Proof: This proof is by induction on the depth of e. The depth of e, $d(e)$, is defined as follows:

1. If e is $Identifier$, $d(e) = 1$
2. If e is `(lambda (`$Identifier$`) `e_1`)`, $d(e) = d(e_1) + 1$
3. If e is `(`e_1, e_2`)`, $d(e) = \max(d(e_1, e_2)$

The induction hypothesis, $IH(k)$, is that for any $e \in LcExp$ of depth $\leq k$ has the same number of left and right parentheses.

1. There are no such e that $d(e) = 0$, so $IH(0)$ holds trivially.
2. Let k be an integer such that $IH(k)$ holds.
   - If e is $Identifier$, we have $k = 1$, and there are no parentheses. So $IH(1)$ holds.
   - If e is `(lambda (`$Identifier$`) `e_1`)`, we have $k = d(e_1) + 1$. Since $d(e_1) < k$, $IH(d(e_1))$ holds, i.e., e_1 have the same number of left and right parentheses. Since e adds exactly two left parentheses and two right parentheses, $IH(k)$ holds.
   - If e is `(`e_1, e_2`)`, $d(e) = \max(d(e_1, e_2)$, we have $k = \max(e_1, e_2)$. Since $d(e_1) < k$, $IH(d(e_1)$ holds. Similarly, $IH(d(e_2))$ holds. Since e adds one left parenthesis and one right parenthesis to $e_1 e_2$, $IH(k)$ holds

Q.E.D.

Exercise 1.6

The procedure may crash when given an empty list because we are trying to apply `car` to `'()`.

Exercise 1.7

```
(define (nth-element lst n)
  (let nth-element-impl ([lst0 lst] [n0 n])
    (if (null? lst0)
      (report-list-too-short lst n)
      (if (zero? n0)
        (car lst0)
        (nth-element-impl (cdr lst0) (- n0 1))))))

(define (report-list-too-short lst n)
  (eopl:error 'nth-element
              "List ~s does not have ~s elements" lst n))
```

Exercise 1.8

If the last line is replaced, this procedure drops the first occurrence of `s` together with all the elements before it:

```
drop-until : Sym x Listof(Sym) -> Listof(Sym)
usage: (drop-until s los) returns a list with the same
       elements arranged in the same order as los,
       except that the first occurrence of the symbol s
       and all elements before it are removed.
```

Exercise 1.9

```
(define (remove s los)
  (if (null? los)
    '()
    (if (eqv? (car los) s)
      (remove s (cdr los))
      (cons (car los) (remove s (cdr los))))))
```

Exercise 1.10

Exclusive or, or "xor".

Exercise 1.11

The thing that gets smaller is the number of occurrences of `old`.

Exercise 1.12

```
(define (subst new old slist)
  (if (null? slist)
    '()
    (let ([head (car slist)] [tail (cdr slist)])
      (cons
        (if (symbol? head)
          (if (eqv? head old) new head)
          (subst new old head))
        (subst new old tail)))))
```

Exercise 1.13

```
(define (subst new old slist)
  (map (lambda (sexp)
         (subst-in-s-exp new old sexp))
       slist))

(define (subst-in-s-exp new old sexp)
  (if (symbol? sexp)
    (if (eqv? sexp old) new sexp)
    (subst new old sexp)))
```

Exercise 1.14

Proof: Translated to mathematical language, `partial-vector-sum` is equivalent to the following function $f(n)$:

$$f(n) = \begin{cases} v_0 & n = 0 \\ v_n + f(n-1) & n > 0 \end{cases}$$

Let's prove by induction on n. The induction hypothesis $IH(k)$, is

$$f(n) = \sum_{i=0}^{n} v_i$$

1. When $k = 0$, $IH(k)$ holds because $f(0) = v_0$.
2. When $k > 0$, we have

$$f(k) = v_k + f(k-1)$$

Since $IH(k-1)$ holds, we have

$$f(k) = v_k + \sum_{i=0}^{k-1} v_i = \sum_{i=0}^{k} v_i$$

Q.E.D.

Exercise 1.15

```
(define (duple n sexp)
  (if (zero? n)
    '()
    (cons sexp (duple (- n 1) sexp))))
```

Exercise 1.16

```
(define (invert lst)
  (map (lambda (pair)
         (list (cadr pair) (car pair)))
       lst))
```

Exercise 1.17

```
(define (down lst)
  (map list lst))
```

Exercise 1.18

```
(define (swapper s1 s2 slist)
  (map (lambda (sexp)
         (swapper-in-s-sexp s1 s2 sexp))
       slist))

(define (swapper-in-s-sexp s1 s2 sexp)
  (cond
    [(symbol? sexp) (cond
                      [(eqv? s1 sexp) s2]
                      [(eqv? s2 sexp) s1]
                      [else sexp])]
    [else (swapper s1 s2 sexp)]))
```

Exercise 1.19

```
(define (list-set lst n x)
  (if (zero? n)
    (cons x (cdr lst))
    (cons
      (car lst)
      (list-set (cdr lst) (- n 1) x))))
```

Exercise 1.20

```
(define (count-occurrences s slist)
  (if (null? slist)
    0
    (+ (count-occurrences-in-s-sexp s (car slist))
       (count-occurrences s (cdr slist)))))
```

```

(define (count-occurrences-in-s-sexp s sexp)
  (if (symbol? sexp)
    (if (eqv? sexp s) 1 0)
    (count-occurrences s sexp)))
```

Exercise 1.21

```
(define (product sos1 sos2)
  (flat-map (lambda (e1)
              (map (lambda (e2)
                     (list e1 e2))
                   sos2))
            sos1))

(define (flat-map proc lst)
  (if (null? lst)
    '()
    (append (proc (car lst))
            (flat-map proc (cdr lst)))))

(check-equal? (product '(a b c) '(x y))
              '((a x) (a y) (b x) (b y) (c x) (c y)))
```

Exercise 1.22

```
(define (filter-in pred lst)
  (if (null? lst)
    '()
    (let ([head (car lst)]
          [filtered-tail (filter-in pred (cdr lst))])
      (if (pred head)
        (cons head filtered-tail)
        filtered-tail))))

(check-equal? (filter-in number? '(a 2 (1 3) b 7)) '(2 7))
```

Exercise 1.23

```
(define (list-index pred lst)
  (cond [(null? lst) #f]
        [(pred (car lst)) 0]
        [else
          (let ([index (list-index pred (cdr lst))])
            (if index (+ 1 index) #f))]))
```

Exercise 1.24

```
(define (every? pred lst)
  (cond [(null? lst) #t]
        [(pred (car lst)) (every? pred (cdr lst))]
        [else #f]))
```

Exercise 1.25

```
(define (exists? pred lst)
  (cond [(null? lst) #f]
        [(pred (car lst)) #t]
        [else (exists? pred (cdr lst))]))
```

Exercise 1.26

```
(define (up lst)
  (cond [(null? lst) '()]
        [(list? (car lst)) (append (car lst)
                                   (up (cdr lst)))]
        [else (cons (car lst)
                    (up (cdr lst)))]))
```

Exercise 1.27

```
(define (flatten slist)
  (cond [(null? slist) '()]
        [(list? (car slist)) (append (flatten (car slist))
                                     (flatten (cdr slist)))]
        [else (cons (car slist)
                    (flatten (cdr slist)))]))
```

Exercise 1.28

```
(define (merge loi1 loi2)
  (cond [(null? loi1) loi2]
        [(null? loi2) loi1]
        [(< (car loi1) (car loi2)) (cons (car loi1)
                                         (merge (cdr loi1) loi2))]
        [else (cons (car loi2)
                    (merge loi1 (cdr loi2)))]))
```

Exercise 1.29

```
(define (sort loi)
  (define (merge-sort lst)
    (cond [(null? lst) '()]
          [(null? (cdr lst)) lst]
          [else (merge-sort (cons (merge (car lst) (cadr lst))
```

```
                                    (merge-sort (cddr lst))))]))
  (car (merge-sort (map list loi))))
```

Exercise 1.30

```
(define (sort/predicate pred loi)
  (define (merge-sort lst)
    (cond [(null? lst) '()]
          [(null? (cdr lst)) lst]
          [else (merge-sort (cons (merge/predicate pred (car lst) (cadr lst))
                                  (merge-sort (cddr lst))))]))
  (car (merge-sort (map list loi))))

(define (merge/predicate pred loi1 loi2)
  (cond [(null? loi1) loi2]
        [(null? loi2) loi1]
        [(pred (car loi1) (car loi2)) (cons (car loi1)
                                            (merge/predicate pred (cdr loi1) loi2))]
        [else (cons (car loi2)
                    (merge/predicate pred loi1 (cdr loi2)))]))
```

Exercise 1.31

```
(define (leaf n) n)

(define (interior-node name lson rson)
  (list name lson rson))

(define (leaf? tree)
  (number? tree))

(define lson cadr)

(define rson caddr)

(define (contents-of tree)
  (cond [(leaf? tree) tree]
        [else (car tree)]))
```

Exercise 1.32

```
(define (double-tree tree)
  (cond [(leaf? tree)
         (leaf (* 2 (contents-of tree)))]
        [else
          (interior-node (contents-of tree)
                         (double-tree (lson tree))
                         (double-tree (rson tree)))]))
```

Exercise 1.33

```
(define (mark-leaves-with-red-depth tree)
  (let mark [(n 0) (node tree)]
    (cond [(leaf? node) (leaf n)]
          [(eqv? (contents-of node) 'red)
           (interior-node 'red
                          (mark (+ n 1) (lson node))
                          (mark (+ n 1) (rson node)))]
          [else (interior-node (contents-of node)
                               (mark n (lson node))
                               (mark n (rson node)))])))
```

Exercise 1.34

```
(define (path n bst)
  (let [(value-of car)
        (lson cadr)
        (rson caddr)]
    (cond [(null? bst)
           (eopl:error 'path "Element ~s not found" n)]
          [(= (value-of bst) n)
           '()]
          [(< n (value-of bst))
           (cons 'left (path n (lson bst)))]
          [else
            (cons 'right (path n (rson bst)))])))
```

Exercise 1.35

```
(define (number-leaves tree)
  (define (number n node)
    (if (leaf? node)
      (list (+ n 1) (leaf n))
      (let* [(lson-result (number n (lson node)))
             (lson-n (car lson-result))
             (new-lson (cadr lson-result))
             (rson-result (number lson-n (rson node)))
             (rson-n (car rson-result))
             (new-rson (cadr rson-result))]
        (list rson-n (interior-node (contents-of node)
                                    new-lson
                                    new-rson)))))
  (cadr (number 0 tree)))
```

Exercise 1.36

```
(define (number-elements lst)
  (if (null? lst) '()
    (g (list 0 (car lst)) (number-elements (cdr lst)))))

(define (g head tail)
```

```
  (if (null? tail)
    (list head)
    (let* [(n (car head))
           (next (car tail))
           (new-next (cons (+ n 1) (cdr next)))]
      (cons head (g new-next (cdr tail))))))
```

Chapter 2. Data Abstraction

Exercise 2.1

Bigits implementation

```
(define base 10)

(define (set-base! n)
  (set! base n))

(define (zero) '())

(define is-zero? null?)

(define (successor n)
  (cond [(is-zero? n)
         '(1)]
        [(< (car n) (- base 1))
         (cons (+ 1 (car n)) (cdr n))]
        [else
          (cons 0 (successor (cdr n)))]))

(define (predecessor n)
  (cond [(= 1 (car n))
         (if (null? (cdr n)) (zero) (cons 0 (cdr n)))]
        [(> (car n) 0)
         (cons (- (car n) 1) (cdr n))]
        [else
          (cons (- base 1) (predecessor (cdr n)))]))

(define (integer->bigits n)
  (if (zero? n)
    (zero)
    (successor (integer->bigits (- n 1)))))

(define (bigits->integer n)
  (if (is-zero? n)
    0
    (+ 1 (bigits->integer (predecessor n)))))

(define (plus x y)
  (if (is-zero? x)
    y
    (successor (plus (predecessor x) y))))

(define (multiply x y)
```

```
  (cond [(is-zero? x) (zero)]
        [(is-zero? (predecessor x)) y]
        [else
          (plus y (multiply (predecessor x) y))]))

(define (factorial n)
  (if (is-zero? n)
    (successor (zero))
    (multiply n (factorial (predecessor n)))))
```

Factorial experiments

```
(define (with-base n thunk)
  (let [(orig-base base)]
    (begin
      (set-base! n)
      (thunk)
      (set-base! orig-base))))

(define (print-factorial base n)
  (with-base
    base
    (lambda ()
      (collect-garbage)
      (eopl:printf
        "Base ~s: ~s! = ~s~%"
        base
        n
        (bigits->integer (factorial (integer->bigits n)))))))
```

Conclusions:

1. The larger the argument is, the longer the execution time is.

   This is because of inherent time complexity of factorial.

2. The larger the base is, the shorter the execution time is.

   Because larger base takes fewer bigits to represent a given number.

Exercise 2.2

1. Unary representation:

   Pros:

   - Fully conforms to the specification, limited by physical memory capacity only.
   - Simple implementation

   Cons:

   - Poor performance
   - Hardly readable when used to represent large numbers

2. Scheme number representation

   Pros:

- Simple implementation
- The representation is easy to read
- Good performance (completely decided by the underlying concrete Scheme implementation)

Cons:

- For Scheme implementations that doesn't have built-in bignum support, calculation result may overflow, thus doesn't fully conform to the specification.

3. Bignum representation

Pros:

- Fully conforms to the specification.
- Relatively good performance. Essentially, larger base leads to better performance.
- The representation is relatively easier to read

Cons:

- Depends on a global state (base)

Exercise 2.3

Question 1

Proof $\forall n \in N$, let S be the set of all representations of $\lceil n \rceil$. Then, $\forall x \in S$, we can always construct $S' \subseteq S$ using the following generation rules:

$$x \in S$$

$$\frac{y \in S}{(\text{diff } x \text{ (diff (one) (one)))}}$$

Apparently, we have $|S'| = \infty$. Thus $|S| = \infty$.

Q.E.D.

Question 2

```
(define-datatype diff-tree diff-tree?
                 (one)
                 (diff (lhs diff-tree?)
                       (rhs diff-tree?)))

(define (zero) (diff (one) (one)))

(define (integer->diff-tree n)
  (cond [(= 1 n) (one)]
        [(> n 0) (successor (integer->diff-tree (- n 1)))]
        [else (predecessor (integer->diff-tree (+ n 1)))]))

(define (diff-tree->integer n)
  (cases diff-tree n
         [one () 1]
         [diff (lhs rhs) (- (diff-tree->integer lhs)
                            (diff-tree->integer rhs))]))
```

```
18
19 (define (diff-tree=? n m)
20   (= (diff-tree->integer n) (diff-tree->integer m)))
21
22 (define (is-zero? n)
23   (cases diff-tree n
24          (one () #f)
25          (diff (lhs rhs) (diff-tree=? lhs rhs))))
26
27 (define (successor n)
28   (diff n (diff (zero) (one))))
29
30 (define (predecessor n)
31   (diff n (one)))
```

Question 3

```
1  (define (diff-tree-plus n m)
2    (diff n (diff (zero) m)))
3
4  (check-diff-tree=? (diff-tree-plus (zero) (one))
5                     (one))
6
7  (check-diff-tree=? (diff-tree-plus (integer->diff-tree 1024)
8                                     (integer->diff-tree 2048))
9                     (integer->diff-tree 3072))
10 ;; end
```

Exercise 2.4

Constructors:

$$(\text{empty-stack}) = \lceil \emptyset \rceil$$
$$(\text{push } \lceil e \rceil \ \lceil stk \rceil) = \lceil stk' \rceil, \text{where } (\text{top } \lceil stk' \rceil) = \lceil e \rceil$$

Observers:

$$(\text{empty-stack? } \lceil stk \rceil) = \begin{cases} \text{t} & \text{if } stk = \lceil \emptyset \rceil \\ \text{f} & \text{otherwise} \end{cases}$$
$$(\text{top } \lceil stk \rceil) = \lceil e \rceil, \text{where } \lceil stk \rceil = (\text{push } \lceil e \rceil \ (\text{pop } \lceil stk \rceil))$$
$$(\text{pop } \lceil stk \rceil) = \lceil stk' \rceil, \text{where } (\text{push } (\text{top } \lceil stk \rceil) \ \lceil stk' \rceil) = \lceil stk \rceil$$

Exercise 2.5

```
1 (define (empty-env) '())
2
3 (define (extend-env var val env)
4   (cons (cons var val) env))
5
6 (define (apply-env env search-var)
7   (cond [(null? env)
8          (report-no-binding-found search-var)]
```

```
        [(eqv? (caar env) search-var)
         (cdar env)]
        [else
          (apply-env env (cdr env))]))

(define (report-no-binding-found search-var)
  (eopl:error 'apply-env "No binding for ~s" search-var))
```

Exercise 2.6

Implementation 1

Represent environments as functions.

```
(define (empty-env)
  (lambda (search-var)
    (eopl:error 'apply-env "No binding for ~s" search-var)))

(define (extend-env var val env)
  (lambda (search-var)
    (if (eqv? var search-var) val (apply-env env search-var))))

(define (apply-env env search-var)
  (env search-var))
```

Implementation 2

A representation that doesn't allow variable "shadowing":

```
(define (empty-env) '())

(define (extend-env var val env)
  (cond [(null? env)
         (list (cons var val))]
        [(eqv? var (caar env))
         (cons (cons var val)
               (cdr env))]
        [else
          (cons (car env)
                (extend-env var val (cdr env)))]))

(define (apply-env env search-var)
  (cond [(null? env)
         (report-no-binding-found search-var)]
        [(eqv? search-var (caar env))
         (cdar env)]
        [else
          (apply-env (cdr env) search-var)]))

(define (report-no-binding-found search-var)
  (eopl:error 'apply-env "No binding for ~s" search-var))
```

Implementation 3

Represent environments as "ribs": a pair consists of a list of variables and a list of values.

```
(define (empty-env) (cons '() '()))

(define (extend-env var val env)
  (let [(vars (car env))
        (vals (cdr env))]
    (cons (cons var vars)
          (cons val vals))))

(define (apply-env env search-var)
  (let* [(vars (car env))
         (vals (cdr env))]
    (cond [(null? vars)
           (report-no-binding-found search-var)]
          [(eqv? search-var (car vars))
           (car vals)]
          [else
            (apply-env (cons (cdr vars)
                             (cdr vals)))])))

(define (report-no-binding-found search-var)
  (eopl:error 'apply-env "No binding for ~s" search-var))
```

Exercise 2.7

```
(define (apply-env env search-var)
  (let apply-env-impl [(env* env)
                       (search-var* search-var)]
    (cond [(eqv? (car env*) 'empty-env)
           (report-no-binding-found search-var* env)]
          [(eqv? (car env*) 'extend-env)
           (let [(saved-var (cadr env*))
                 (saved-val (caddr env*))
                 (saved-env (cadddr env*))]
             (if (eqv? search-var* saved-var)
               saved-val
               (apply-env saved-env search-var*)))]
          [else
            (report-invalid-env env*)])))

(define (report-no-binding-found search-var env)
  (eopl:error 'apply-env "No binding for ~s in environment ~s" search-var env))
```

Exercise 2.8

```
(define empty-env? null?)
```

Exercise 2.9

```
(define (has-binding? env s)
  (cond [(empty-env? env) #f]
        [(eqv? s (caar env)) #t]
        [else (has-binding? (cdr env) s)]))
```

Exercise 2.10

```
(define (extend-env* vars vals env)
  (if (null? vars)
    env
    (extend-env* (cdr vars)
                 (cdr vals)
                 (extend-env (car vars)
                             (car vals)
                             env))))
```

Exercise 2.11

```
(define (empty-env) '())

(define (extend-env* vars vals env)
  (cons (cons vars vals) env))

(define (extend-env var val env)
  (extend-env* (list var) (list val) env))

(define (apply-env env search-var)
  (cond [(null? env)
         (report-no-binding-found search-var)]
        [else
          (let apply-ribs [(ribs (car env))]
            (let [(vars (car ribs))
                  (vals (cdr ribs))]
              (cond [(null? vars)
                     (apply-env (cdr env) search-var)]
                    [(eqv? (car vars) search-var)
                     (car vals)]
                    [else
                      (apply-ribs (cons (cdr vars) (cdr vals)))])))]))

(define (report-no-binding-found search-var)
  (eopl:error 'apply-env "No binding for ~s" search-var))
```

Exercise 2.12

```
(define (empty-stack)
  (list #t
        (lambda ()
          (eopl:error 'top "Stack is empty"))
        (lambda ()
```

```
          (eopl:error 'pop "Stack is empty"))))

(define (push e stk)
  (list #f
        (lambda () e)
        (lambda () stk)))

(define (empty-stack? stk)
  (list-ref stk 0))

(define (top stk)
  ((list-ref stk 1)))

(define (pop stk)
  ((list-ref stk 2)))

(define (stack=? s1 s2)
  (cond [(empty-stack? s1)
         (empty-stack? s2)]
        [(empty-stack? s2)
         (empty-stack? s1)]
        [else
          (and (eqv? (top s1) (top s2))
               (stack=? (pop s1) (pop s2)))]))

(define-binary-check
  (check-stack=? stack=? actual expected))
```

Exercise 2.13

```
(define (empty-env)
  (cons (lambda (search-var)
          (report-no-binding-found search-var))
        (lambda () #t)))

(define (extend-env saved-var saved-val saved-env)
  (cons (lambda (search-var)
          (if (eqv? search-var saved-var)
            saved-val
            (apply-env saved-env search-var)))
        (lambda () #f)))

(define (apply-env env search-var)
  ((car env) search-var))

(define (empty-env? env)
  ((cdr env)))

(define (report-no-binding-found search-var)
  (eopl:error 'apply-env "No binding for ~s" search-var))
```

Exercise 2.14

```
(define (empty-env)
  (list (lambda (search-var)
          (report-no-binding-found search-var))
        (lambda () #t)
        (lambda (search-var) #f)))

(define (extend-env saved-var saved-val saved-env)
  (list (lambda (search-var)
          (if (eqv? search-var saved-var)
            saved-val
            (apply-env saved-env search-var)))
        (lambda () #f)
        (lambda (search-var)
          (if (eqv? search-var saved-var)
            #t
            ((list-ref saved-env 2) search-var)))))

(define (apply-env env search-var)
  ((list-ref env 0) search-var))

(define (empty-env? env)
  ((list-ref env 1)))

(define (has-binding? env search-var)
  ((list-ref env 2) search-var))

(define (report-no-binding-found search-var)
  (eopl:error 'apply-env "No binding for ~s" search-var))
```

Exercise 2.15

```
(define (var-exp var)
  `(var-exp ,var))

(define (lambda-exp bound-var body)
  `(lambda-exp (,bound-var) ,body))

(define (app-exp rator rand)
  `(app-exp ,rator ,rand))

(define (var-exp? exp)
  (match exp
    [(list 'var-exp (? symbol?)) #t]
    [_ #f]))

(define (lambda-exp? exp)
  (match exp
    [(list 'lambda-exp (list (? var-exp?)) (? lc-exp?)) #t]
    [_ #f]))

(define (app-exp? exp)
  (match exp
    [(list 'app-exp (? lc-exp?) (? lc-exp?)) #t]
    [_ #f]))
```

```
(define (lc-exp? exp)
  (or (var-exp? exp)
      (lambda-exp? exp)
      (app-exp? exp)))

(define (var-exp->var exp)
  (match exp
    [(list 'var-exp var) var]))

(define (lambda-exp->bound-var exp)
  (match exp
    [(list 'lambda-exp (list bound-var) _) bound-var]))

(define (lambda-exp->body exp)
  (match exp
    [(list 'lambda-exp _ body) body]))

(define (app-exp->rator exp)
  (match exp
    [(list 'app-exp rator _) rator]))

(define (app-exp->rand exp)
  (match exp
    [(list 'app-exp _ rand) rand]))
```

Exercise 2.16

```
(define (var-exp var)
  `(var-exp ,var))

(define (lambda-exp bound-var body)
  `(lambda-exp ,bound-var ,body))

(define (app-exp rator rand)
  `(app-exp ,rator ,rand))

(define (var-exp? exp)
  (match exp
         [(list 'var-exp (? symbol?)) #t]
         [_ #f]))

(define (lambda-exp? exp)
  (match exp
         [(list 'lambda-exp (? var-exp?) (? lc-exp?)) #t]
         [_ #f]))

(define (app-exp? exp)
  (match exp
         [(list 'app-exp (? lc-exp?) (? lc-exp?)) #t]
         [_ #f]))

(define (lc-exp? exp)
  (or (var-exp? exp)
      (lambda-exp? exp)
      (app-exp? exp)))
```

```
29
30 (define (var-exp->var exp)
31   (match exp
32          [(list 'var-exp var) var]))
33
34 (define (lambda-exp->bound-var exp)
35   (match exp
36          [(list 'lambda-exp bound-var _) bound-var]))
37
38 (define (lambda-exp->body exp)
39   (match exp
40          [(list 'lambda-exp _ body) body]))
41
42 (define (app-exp->rator exp)
43   (match exp
44          [(list 'app-exp rator _) rator]))
45
46 (define (app-exp->rand exp)
47   (match exp
48          [(list 'app-exp _ rand) rand]))
```

Exercise 2.17

Representation 1

```
1  (define (var-exp var) var)
2
3  (define (lambda-exp bound-var body)
4    `(,bound-var ,body))
5
6  (define (app-exp rator rand)
7    `(,rator ,rand))
8
9  (define (var-exp? exp) (symbol? exp))
10
11 (define (lambda-exp? exp)
12   (match exp
13          [(list (? var-exp?) (? lc-exp?)) #t]
14          [_ #f]))
15
16 (define (app-exp? exp)
17   (match exp
18          [(list (? lc-exp?) (? lc-exp?)) #t]
19          [_ #f]))
20
21 (define (lc-exp? exp)
22   (or (var-exp? exp)
23       (lambda-exp? exp)
24       (app-exp? exp)))
25
26 (define (var-exp->var exp) exp)
27
28 (define (lambda-exp->bound-var exp)
29   (match exp
30          [(list bound-var _) bound-var]))
31
```

```
(define (lambda-exp->body exp)
  (match exp
         [(list _ body) body]))

(define (app-exp->rator exp)
  (match exp
         [(list rator _) rator]))

(define (app-exp->rand exp)
  (match exp
         [(list _ rand) rand]))
```

Representation 2

```
(define (var-exp var)
  (cons 'var-exp
        (lambda () var)))

(define (lambda-exp bound-var body)
  (cons 'lambda-exp
        (lambda (field)
          (match field
                 ['bound-var bound-var]
                 ['body body])))))

(define (app-exp rator rand)
  (cons 'app-exp
        (lambda (field)
          (match field
                 ['rator rator]
                 ['rand rand])))))

(define (var-exp? exp)
  (match exp
         [(cons 'var-exp _) #t]
         [_ #f]))

(define (lambda-exp? exp)
  (match exp
         [(cons 'lambda-exp _) #t]
         [_ #f]))

(define (app-exp? exp)
  (match exp
         [(cons 'app-exp _) #t]
         [_ #f]))

(define (lc-exp? exp)
  (or (var-exp? exp)
      (lambda-exp? exp)
      (app-exp? exp)))

(define (var-exp->var exp)
  ((cdr exp)))

(define (lambda-exp->bound-var exp)
```

```
  ((cdr exp) 'bound-var))

(define (lambda-exp->body exp)
  ((cdr exp) 'body))

(define (app-exp->rator exp)
  ((cdr exp) 'rator))

(define (app-exp->rand exp)
  ((cdr exp) 'rand))
```

Exercise 2.18

```
(define (number->sequence n)
  (list n '() '()))

(define (current-element seq)
  (car seq))

(define (move-to-left seq)
  (if (at-left-end? seq)
    (eopl:error 'move-to-left
                "Sequence ~s is already at its left end" seq)
    (let* [(n (car seq))
           (left (cadr seq))
           (right (caddr seq))
           (new-n (car left))
           (new-left (cdr left))
           (new-right (cons n right))]
      (list new-n new-left new-right))))

(define (move-to-right seq)
  (if (at-right-end? seq)
    (eopl:error 'move-to-right
                "Sequence ~s is already at its right end" seq)
    (let* [(n (car seq))
           (left (cadr seq))
           (right (caddr seq))
           (new-n (car right))
           (new-left (cons n left))
           (new-right (cdr right))]
      (list new-n new-left new-right))))

(define (insert-to-left n seq)
  (let* [(current (car seq))
         (left (cadr seq))
         (right (caddr seq))
         (new-left (cons n left))]
    (list current new-left right)))

(define (insert-to-right n seq)
  (let* [(current (car seq))
         (left (cadr seq))
         (right (caddr seq))
         (new-right (cons n right))]
    (list current left new-right)))
```

```

(define (at-left-end? seq)
  (null? (cadr seq)))

(define (at-right-end? seq)
  (null? (caddr seq)))
```

Exercise 2.19

```
(define (number->bintree n)
  `(,n () ()))

(define (current-element bintree)
  (car bintree))

(define (move-to-left-son bintree)
  (cadr bintree))

(define (move-to-right-son bintree)
  (caddr bintree))

(define (at-leaf? bintree)
  (null? bintree))

(define (insert-to-left n bintree)
  (let* [(root (car bintree))
         (lhs (move-to-left-son bintree))
         (rhs (move-to-right-son bintree))
         (lhs* `(,n ,lhs ()))]
    `(,root ,lhs* ,rhs)))

(define (insert-to-right n bintree)
  (let* [(root (car bintree))
         (lhs (move-to-left-son bintree))
         (rhs (move-to-right-son bintree))
         (rhs* `(,n ,rhs ()))]
    `(,root ,lhs ,rhs*)))
```

CHAPTER 2

Overview

This documentation is my (WIP) solutions to Essentials of Programming Languages 3rd edition exercises. All the source code can be found on GitHub.

The Scheme code is written using the `eopl` dialect provided by DrRacket 6.4. Please refer to the official DrRacket documentation for more details. Most code snippets are tested using `rackunit` (doc). Each source file is a complete Racket program, although sometimes only the interesting parts in the file are shown in the HTML page.

To run the code, first install Racket 6.4+, then install the `eopl` package:

```
$ raco pkg install eopl
```

After installing the `eopl` package, running the scheme code should be as easy as:

```
$ racket <file-path>
```