Exercises

Exercise 1.15
duple : $Int \times SchemeVal \rightarrow Listof(SchemeVal)$
usage:

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

Exercise 1.16

```
(define invert
  (lambda (lst)
    (if (null? lst)
        '()
        (cons (list (cadar lst) (caar lst))
              (invert (cdr lst))))))
```

Exercise 1.17

```
(define down
  (lambda (lst)
    (if (null? lst)
        '()
        (cons (list (car lst))
              (down (cdr lst))))))
```

Exercise 1.18

```
(define swap-in-s-exp
  (lambda (s1 s2 sexp)
    (if (symbol? sexp)
        (cond ((eqv? sexp s1) s2)
              ((eqv? sexp s2) s1)
              (else sexp))
```

```
        (swapper s1 s2 sexp))))

(define swapper
  (lambda (s1 s2 slist)
    (if (null? slist)
        '()
        (cons (swap-in-s-exp s1 s2 (car slist))
              (swapper s1 s2 (cdr slist))))))
```

Exercise 1.19

```
(define report-list-too-short
  (lambda (n proc-name)
    (eopl:error proc-name
                "List too short by ~s elements.~%"
                (+ n 1))))

(define list-set
  (lambda (lst n x)
    (cond ((null? lst) (report-list-too-short n 'list-set))
          ((zero? n) (cons x (cdr lst)))
          (else (cons (car lst)
                      (list-set (cdr lst) (- n 1) x))))))
```

Exercise 1.20

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

(define count-occurrences-in-s-exp
  (lambda (s sexp)
```

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

Exercise 1.21

```
(define product
  (lambda (sos1 sos2)
    (if (null? sos1)
        '()
        (append (product-exp (car sos1) sos2)
                (product (cdr sos1) sos2)))))

(define product-exp
  (lambda (s sos2)
    (if (null? sos2)
        '()
        (cons (list s (car sos2))
              (product-exp s (cdr sos2))))))
```

Exercise 1.22

```
(define filter-in
  (lambda (pred lst)
    (cond ((null? lst) '())
          ((pred (car lst))
           (cons (car lst) (filter-in pred (cdr lst))))
          (else (filter-in pred (cdr lst))))))
```

Exercise 1.23

```
(define list-index
  (lambda (pred lst)
    (list-i pred lst 0)))

(define list-i
  (lambda (pred lst n)
```

```
    (cond ((null? lst) #f)
          ((pred (car lst)) n)
          (else (list-i pred (cdr lst) (+ n 1))))))
```

Exercise 1.24

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

Exercise 1.25

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

Exercise 1.26

```
(define up
  (lambda (lst)
    (cond ((null? lst) '())
          ((not (pair? (car lst)))
           (cons (car lst) (up (cdr lst))))
          (else (append (car lst)
                        (up (cdr lst)))))))
```

Exercise 1.27

```
(define flatten
  (lambda (slist)
    (if (null? slist)
        '()
        (append (flatten-in-s-exp (car slist))
```

```
                (flatten (cdr slist))))))
```

```
(define flatten-in-s-exp
  (lambda (sexp)
    (if (symbol? sexp)
        (list sexp)
        (flatten sexp))))
```

Exercise 1.28

Exercise 1.29

Exercise 1.30

Exercise 1.31

Exercise 1.