Part I

Exercises 7.2, 7.3, 7.6, 7.7, 7.8, 7.12, 7.18, 7.22, 7.30, 7.31

Part II

1. Consider the following three examples:

;;;; Example 1
(define fact
 (lambda (x)
 (letrec
   ((loop
     (lambda (x acc)
      (if (= x 0)
       acc
       (loop (sub1 x) (* x acc)))))
 (loop x 1)))))

;;;; Example 2
(define reverse
 (lambda (x)
 (letrec
   ((loop
     (lambda (x acc)
      (if (null? x)
       acc
       (loop (cdr x) (cons x acc))))))
 (loop x ’())))

;;;; Example 3
(define iota
 (lambda (x)
 (letrec
   ((loop
     (lambda (x acc)
      (if (= x 0)
       acc
       (loop x acc)))))
 (loop x 1))))
The higher-order function tail-recur takes the following arguments

- *bpred* - a procedure of x which returns true if the terminating condition is satisfied and false otherwise
- *xproc* - a procedure of x which updates x
- *aproc* - a procedure of x and acc which updates acc
- *acc0* - an initial value for acc

and returns a tail recursive function of x. It can be used to write the function, factorial as follows:

> (define fact (tail-recur zero? sub1 * 1))
> (fact 10)
3628800

(a) Give a definition for tail-recur.
(b) Use tail-recur to define reverse.
(c) Use tail-recur to define iota.

2. Write a function, disjunction2, which takes two predicates as arguments and returns the predicate which returns #t if either predicate does not return #f. For example:

> ((disjunction2 symbol? procedure?) +)
#t
> ((disjunction2 symbol? procedure?) (quote +))
#t
> (filter (disjunction2 even? (lambda (x) (< x 4))) (iota 8))
(1 2 3 4 6 8)

3. Now write disjunction, which takes an arbitrary number (> 0) of predicates as arguments.

4. A matrix, \[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\], can be represented in Scheme as a list of lists: ((1 2) (3 4)). Without using recursion, write a function, matrix-map, which takes a function, f, and a matrix, A, as arguments and returns the matrix, B, consisting of f applied to the elements of A, i.e., \(B_{ij} = f(A_{ij})\).

> (matrix-map (lambda (x) (* x x)) ’((1 2) (3 4)))
((1 4) (9 16))
5. Consider the following definition for fold (called flat-recur in your text):

```
(define fold
  (lambda (seed proc)
    (letrec
     ((pattern
       (lambda (ls)
         (if (null? ls)
             seed
             (proc (car ls)
                 (pattern (cdr ls))))))
      pattern)))
```

(a) Use fold to write a function delete-duplicates which deletes all duplicate items from a list. For example,

```
> (delete-duplicates '(a b a b a b a b))
(a b)
> (delete-duplicates '(1 2 3 4))
(1 2 3 4)
```

(b) Use fold to write a function assoc which takes an item and a list of pairs as arguments and returns the first pair in the list with a car car which is equal to item. If there is no such pair then assoc should return false. For example,

```
> (assoc 'b '((a 1) (b 2)))
(b 2)
> (assoc 'c '((a 1) (b 2)))
#f
```

Part III

Using the functions, apply, select, map, filter, outer-product and iota, and without using recursion, give definitions for the following functions:

1. length - returns the length of a list.
2. sum-of-squares - returns the sum of the squares of its arguments.
3. avg - returns the average of its arguments.
4. avg-odd - returns the average of its odd arguments.
5. *shortest* - returns the shortest of its list arguments.

6. *avg-fact* - returns the average of the factorials of its arguments.

7. *tally* - takes a predicate and a list and returns the number of list elements which satisfy the predicate.

8. *list-ref* - takes a list and an integer, \textit{n}, and returns the \textit{n}-th element of the list.