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
                acc))))))
The higher-order function \textit{tail-recur} takes the following arguments

- \textit{bpred} - a procedure of \(x\) which returns true if the terminating condition is satisfied and false otherwise
- \textit{xproc} - a procedure of \(x\) which updates \(x\)
- \textit{aproc} - a procedure of \(x\) and \(acc\) which updates \(acc\)
- \textit{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:

\begin{verbatim}
> (define fact (tail-recur zero? sub1 * 1))
> (fact 10)
362880
\end{verbatim}

(a) Give a definition for \textit{tail-recur}.

(b) Use \textit{tail-recur} to define \textit{reverse}.

(c) Use \textit{tail-recur} to define \textit{iota}.

2. Consider the following three examples:

\begin{verbatim}
;; Example 1
(define longer
  (lambda (ls1 ls2)
    (if (> (length ls1) (length ls2))
        ls1
        ls2)))

(define longest
  (lambda args
    (cond ((null? args) ())
          ((null? (cdr args)) (car args))
          (else
           (longer (car args)
                    (apply longest (cdr args)))))))

;; Example 2
(define either?
  (lambda (x y) (or x y)))
\end{verbatim}
(define any?
  (lambda args
    (cond ((null? args) #f)
           ((null? (cdr args)) (car args))
           (else
             (either? (car args)
                       (apply any? (cdr args)))))))

;; Example 3
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))

(define compose-many
  (lambda args
    (cond ((null? args) (lambda (x) x))
          ((null? (cdr args)) (car args))
          (else
            (compose (car args)
                     (apply compose-many (cdr args)))))))

In each of the three examples above, a function of two arguments is generalized to a function of an arbitrary number of arguments. Write a higher-order function generalize which takes a function of two arguments, proc2, and an identity element, id, as arguments, and returns the generalized function. Your function should behave as follows:

(define longest (generalize longer ()))
(define any? (generalize either? #f))
(define compose-many (generalize compose (lambda (x) x)))

3. 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)))))))

(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, n, and returns the n-th element of the list.