1. Basic list operations in SCHEME.

   (a) Define a SCHEME function which takes two lists as parameters and returns `#t` if they are the same length and `#f` otherwise.

   Solution:

   ```
   (define (equal-length? L1 L2)
     (cond ((and (null? L1) (null? L2)) #t)
           ((null? L1) #f)
           ((null? L2) #f)
           (else (equal-length? (cdr L1) (cdr L2)))))
   ```

   (b) Consider the various SCHEME values that can be formed with integers and the `list` command. For example

   `(1 (2 3 4) (3 8) 33)`

   can be formed by evaluating

   `(list 1 (list 2 3 4) (list 3 8) 33)`

   Let us call such a value an "compoud intlist." Then`34`, `(3 1)`, and `(3 (3 (3 1) (5 5 5)))` are all compound intlists.

   Define a SCHEME function which takes two intlists as parameters and returns `#t` if they are the "same." To be precise, we define ℓ_1 and ℓ_2 to be the same if:

   1. They are the same integer,
   2. They are both the emptylist; or
   3. They are both nontrivial lists, and both their `car` and `cdr` are the same.

   You'll notice that we need a way to tell the difference between a "list-type" value and a "integer-type" value. One way to do this is to use the built-in SCHEME command `pair?` which returns `#t` if its argument was built with `cons`, and `#f` otherwise. For example,

   ```
   > (pair? 23)
   #f
   > (pair? '(3 4))
   #t
   > (pair? (cons 3 4))
   #t
   > (pair? '(3 .4))
   #t
   > (pair? (lambda (x) x))
   #f
   ```

You can use this to figure out if you need to apply case 1 above.

Solution:

```
(define (same? L1 L2)
  (cond ((and (null? L1)(null? L2)) #t)
        ((null? L1) #f)
        ((null? L2) #f)
        ((and (not (pair? L1)) (not (pair? L2)))
         (if (= L1 L2)
             #t
             #f))
        ((not (pair? L1)) #f)
        ((not (pair? L2)) #f)
        ((and (same? (car L1) (car L2))
              (same? (cdr L1) (cdr L2))) #t)
        (else #f)))
```

(c) Define a SCHEME function which takes a list of integers as a parameter and returns the average of all of the numbers in the list.

Solution:

```
(define (list-avg L)
  (define (list-sum L)
    (if (null? L)
        0
        (+ (car L) (list-sum (cdr L)))))
  (/ (list-sum L)(length L)))
(list-avg (list 1 2 3 4 5 6 7 8 9))
```

(d) Define a SCHEME function named `filter` which takes a function, f and a list, L, as parameters and returns a list composed of all elements x of L for which $f(x)$ returns `#t`.

Solution:

```
(define (filter f L)
  (if (null? L)
      (list)
      (if (f (car L))
          (cons (car L) (filter f (cdr L)))
          (filter f (cdr L)))))
(define (even? x)
  (= (modulo x 2) 0))
(filter even? (list 1 2 3 4 5 6 7 8 9))
```

(e) Write a SCHEME function (from scratch, using only `cons`, `car`, and `cdr` to manipulate lists) that appends two lists together. For example, your function, when called on (1 2 3) and (4 5), should return the list (1 2 3 4 5).

Solution:

```
(define (new-append L1 L2)
  (if (null? L1)
      L2
      (cons (car L1) (new-append (cdr L1) L2))))
(new-append (list 1 2 3 4 5) (list 6 7 8 9 10))
```

(f) Define a SCHEME function `list-sum` which, given a list of integers, computes their sum.

Solution:

```
(define (list-sum L)
  (if (null? L)
      0
      (+ (car L) (list-sum (cdr L)))))
```

(g) **Set Intersection** Define a SCHEME function named `intersection` which accepts two lists representing sets as parameters and returns a list representing the intersection of those two sets. That is, your function should return a list that includes all elements present in both lists passed as parameters. For example `(intersection (list 1 2 3 4) (list 2 4 6 8))` returns `(2 4)`.

Solution:

```
(define (intersection L1 L2)
  (define (member? x L)
    (if (null? L)
        #f
        (if (= x (car L))
            #t
            (member? x (cdr L)))))
  (if (null? L1)
      L1
      (if (member? (car L1) L2)
          (cons (car L1) (intersection (cdr L1) L2))
          (intersection (cdr L1) L2))))
```

(h) Define a SCHEME function which takes a list, `lst`, as a parameter and uses `insert` (see definitions below) to build a binary search tree containing the elements of `lst`.

Solution:

```
(define (list-to-tree L)
  (if (null? L)
      L
      (insert (car L) (list-to-tree (cdr L)))))

(define mytree (list-to-tree (list 5 4 6 3 7 2 8 1 9)))
(define mytree2 (list-to-tree (list 9 1 8 2 7 3 6 4 5)))
(define mytree3
  (list-to-tree
   (list 15 13 11 9 7 5 3 1 14 10 6 2 12 4 8)))

(define (height T)
   (if (null? T) 0
       (let ((h-left (height (left T )))
             (h-right (height (right T ))))
         (if (> h-left h-right)
             (+ 1 h-left )
             (+ 1 h-right )))))

(height mytree)
(height mytree2)
(height mytree3)
```

(i) Define a SCHEME function, named `replace`, which takes two values, `find` and `rplc`, and a list as parameters and returns a list with the first occurrence of `find` in the original list replaced with `rplc`.

Solution:

```
(define (replace find rplc L)
  (if (null? L)
      L
      (if (= find (car L))
          (cons rplc (cdr L))
          (cons (car L) (replace find rplc (cdr L))))))

(replace 6 33 (list 1 2 3 4 5 6 7 8 9 10))
```

(j) Detecting palindrome: `(list 1 2 2 1)` and `(list 1 2 1)` are palindromes, because they are the same when reversed (i.e., invariant under the operation reverse) . Write a SCHEME function `palindrome?` to determine whether a list of integers is a palindrome or not.

Solution:

```
(define (palindrome? L)
  (equal? L (reverse L)))

(define (ispalindrome? L)
  (same? L (reverse L)))
```

(k) Define a SCHEME function `prefix-sum` which, given a list $(a_1\ a_2\ \ldots\ a_k)$ of integers, returns the list $(s_1\ s_2\ \ldots\ s_k)$ where s_i is the sum of the first i of the a_i. Specifically,

$$s_1 = a_1, \quad s_2 = a_1 + a_2, \quad \ldots, \quad s_k = a_1 + a_2 + \cdots + a_k .$$

Solution:

```
(define (prefix-sum L)
  (define (prefix-sum-aux sum L)
    (if (null? L)
        L
        (let ((new-sum (+ sum (car L))))
          (cons new-sum (prefix-sum-aux new-sum (cdr L))))))
  (prefix-sum-aux 0 L))
```

TREE CONVENTIONS

As in class, we consider binary trees with the property that each node stores—in addition to its two subtrees—an integer. Recall the convention that we used for such trees: each node is represented by a list

```
(v left right),
```

where `v` is a value, and `left` and `right` are its two subtrees.

Please adopt this convention in your responses to the following questions. You may make free use of the "convenience" functions:

```
(define (maketree v left-tree right-tree)
   (list v left-tree right-tree))

(define (value T) (car T))
(define (left T)  (cadr T))
(define (right T) (caddr T))

(define (insert x T)
  (cond ((null? T) (make-tree x '() '()))
        ((eq? x (value T)) T)
        ((< x (value T)) (make-tree (value T)
                                    (insert x (left T))
                                    (right T)))
        ((> x (value T)) (make-tree (value T)
                                    (left T)
                                    (insert x (right T))))))
```

The first of these functions builds a tree from a value and a pair of trees; the following three functions extract, from a tree T, the value of the root and its left and right subtrees. Finally, we used the SCHEME empty list to represent the empty tree.

2. Repeat the set intersection question using Binary Search Trees to represent your sets: Define a SCHEME function named `intersection` which accepts two BSTs representing sets as parameters and returns a list representing the intersection of those two sets. That is, your function should return a list that includes all elements present in both BSTs passed as parameters. For example `(intersection (list 1 2 3 4) (list 2 4 6 8))` returns `(2 4)`.

Solution:

```
(define mytree3
  (list-to-tree
   (list 15 13 11 9 7 5 3 1 14 10 6 2 12 4 8)))
(define mytree4 (list-to-tree (list 2 4 6 8 10 12 14 16 18 20)))
```

```
(define (element? x T)
  (cond ((null? T) #f)
        ((eq? x (value T)) #t)
        ((< x (value T)) (element? x (left T)))
        ((> x (value T)) (element? x (right T)))))


(define (bst-set-intersection A B)
  (cond ((null? A) (list))
        ((element? (value A) B)
         (cons (value A)
               (append (bst-set-intersection (left A) B)
                       (bst-set-intersection (right A) B))))
        (else
         (append (bst-set-intersection (left A) B)
                 (bst-set-intersection (right A) B)))))

(bst-set-intersection mytree3 mytree4)
```

3. Recall that a *heap* is a binary tree with the property that *the value of any node is less than that of its children*.

   Write a SCHEME function that, given a heap H and an integer z, returns the number of elements in the heap that are less than or equal to z. For example, your function, called on the heap shown below with the number 12, should return 3, as there are 3 elements of the heap that are less than or equal to 12.

   Your solution should exploit the fact that the tree is a heap to avoid considering certain subtrees; explain.

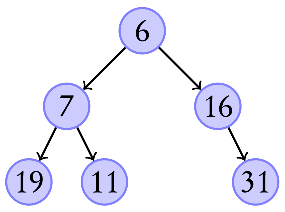

Figure 1: A heap containing 3 elements less than or equal to 12.

Solution:

```
(define (create-heap v H1 H2)
  (list v H1 H2))
(define (h-max H) (car H))
(define (h-left H)  (cadr H))
(define (h-right H) (caddr H))

(define (h-insert x H)
  (if (null? H)
      (create-heap x '() '())
      (let ((child-value (min x (h-max H)))
            (root-value  (max x (h-max H))))
        (create-heap root-value
                     (h-right H)
                     (h-insert child-value (h-left H))))))

(define myheap (h-insert 1 (h-insert 2 (h-insert 3
               (h-insert 4 (h-insert 5 (h-insert 6
               (h-insert 7 (h-insert 8 (h-insert 9
               (h-insert 10 (list)))))))))))
(define (h-num-gteq-z z H)
  (cond ((null? H) 0)
        ((< (value H) z) 0) ;all nodes in
                            ;subtrees are < value here
        (else
         (let ((num-gteq-left (h-num-gteq-z z (left H)))
               (num-gteq-right (h-num-gteq-z z (right H))))
           (+ 1 num-gteq-left num-gteq-right)))))
```

4. Polynomials: A polynomial can be represented as a list. If the polynomial is of degree n, the list will have $n+1$ elements, one for the coefficient of each of the powers of the indeterminate, say, x, including the power `0`. For example, we would represent the polynomial $3x^3-2x+1$ as the list `(3 0 -2 1)`. (Note that this polynomial has no x^2 term.)

   (a) Write a SCHEME function to add two polynomials (assumed to be in the same indeterminate). Your function should take two lists (representing polynomials) and return a list (representing a polynomial). Watch out! The polynomials may have different degree.

Solution:

```
(define (add-polynomial a b)
  (define (add-polynomial-aux a b)
    (cond ((and (null? a) (null? b))(list))
          ((null? a) b)
          ((null? b) a)
          (else
           (cons (+ (car a) (car b))
                 (add-polynomial-aux (cdr a) (cdr b))))))
  (reverse (add-polynomial-aux (reverse a) (reverse b))))
```

(b) Write a SCHEME funtion `evaluate`, which takes a polynomial (as a list) and a number n, and *evaluates* the polynomial at the number n. For example, given the list `(3 0 2 1)` and the number 3, it should return

$$3 \cdot 3^3 - 2 \cdot 3 + 1 = 76.$$

Solution:

```
(define (evaluate poly n)
  (define (evaluate-aux p degree)
    (if (null? p)
        0
        (+ (* (car p) (expt n degree))
           (evaluate-aux (cdr p) (+ degree 1)))))
  (evaluate-aux (reverse poly) 0))
(evaluate (list 3 0 -2 1) 3)
```

(c) Now, for something even trickier. Define a function `convert-to-fn`, which takes a polynomial (as a list), and returns a *function* that computes the polynomial. For example, given the list `(3 0 -2 1)`, `convert-to-fn` should return a SCHEME function `f` which, given x, returns $3x^3 - 2x + 1$.

Solution:

```
(define (convert-to-fn poly)
  (define (convert-to-fn-aux p degree)
    (lambda (x)
      (if (null? p)
          0
          (+ (* (car p) (expt x degree))
             ((convert-to-fn-aux (cdr p) (+ degree 1)) x)))))
  (convert-to-fn-aux (reverse poly) 0))
((convert-to-fn (list 3 0 -2 1)) 3)
```