Homework Assignment 9

In this assignment you *cannot* use any imperative constructs of scheme, such as set! or the iterative loop constructs. You also are not allowed to use higher level functions, such as length to get the length of the list. Instead you must use the basic built-in functions for manipulating a list--car, cdr, cons, append--and the built-in arithmetic operators to create your functions.

---

1. Write a function in scheme called `swap` that takes two arguments and returns a cons pair, with the smaller argument first and the larger argument second. For example, the call:

   ```
   (swap 9 6)
   ```

   should return the cons pair `(6 . 9)`.

2. Write a recursive function in scheme named `avg` that takes a list of numbers and returns their average as a floating point number. For example, the call:

   ```
   (avg '(3 6 17 12 15))
   ```

   should return 10.6 (and not 53/5 which is what you'll get if you do not ensure that you have a floating point operand in one of the dividend or divisor). For efficiency reasons, please do not write two functions that separately compute the sum of the list and the length of the list (i.e., do not traverse the list twice). It is possible to write a solution to this question that traverses the list only once and simultaneously computes the sum and length of the list and then returns their quotient.

3. Write a tail-recursive version of the following:

   ```
   ;; find minimum element in a list
   (define min
     (lambda (l)
       (cond
         ((null? l) '())
         ((null? (cdr l)) (car l))
         (#t (let ((a (car l))
                   (b (min (cdr l))))
               (if (< b a) b a)))))))
   ```

4. Write a recursive function named mergesort that takes a single list argument and returns the list in sorted order. For example:

   ```
   (mergesort '(1 3 2 4 8 1 9 6 10))  ==> '(1 1 2 3 4 6 8 9 10)
   ```

   As long as you use the purely functional features of scheme (i.e., no imperative constructs), you may design your merge sort function any way you see fit. However, it helped me to define the following two helper functions, which I created and tested first, and then used them to build mergesort:

   a. (merge L1 L2): takes two lists and returns a single merged list. For example,

      ```
      (merge '(2 4 6) '(1 3 5 9 10)) ==> '(1 2 3 4 5 6 9 10)
      ```

   b. (mergesortHelper L L1 L2 whichlist?): divides a list L into two separate, equal-sized lists L1 and L2. whichlist? indicates which list the next element of L should be added to. I used cons to add the next element of L to either L1 or L2. mergesortHelper should be recursive and should use continuation-style arguments for L1 and L2 (i.e., L1 and L2 grow with each successive call to

mergesortHelper). mergesortHelper can either return L1 and L2 as a cons pair when L is empty, or it can directly implement the general case of merge sort once L is empty (i.e., call mergesort on each of the two lists L1 and L2 and call merge to merge the resulting two lists).

My eventual mergesort was very short. It implemented the two base cases where the list is either empty or has one element, in which case it simply returns the list, and it implemented the general case by calling mergesortHelper with the appropriate initial arguments.

5. Write a purely functional Scheme function to return a list containing all elements of a given list that satisfy a given predicate. For example,

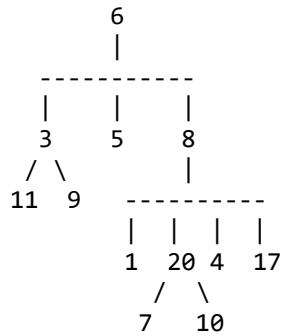
```
(filter (lambda(x) (< x 5)) '(3 9 5 8 2 4 7))
```

should return `(3 2 4)`. Your function should be named `filter`.

6. A tree can be defined in Scheme as follows:

   a. The empty list is an empty tree
   b. A list with a single element is a tree with a single node (i.e., a leaf)
   c. The first element of a list is the root of the tree. The remaining elements are sub-trees.

For example the tree:



would be represented as the list:

```
'(6 (3 (11) (9)) (5) (8 (1) (20 (7) (10)) (4) (17)))
```

Write a function named `fringe` that takes a list which represents a tree and returns a list of the leaves of the tree, in left-to-right order. For example,

```
(fringe '(6 (3 (11) (9)) (5) (8 (1) (20 (7) (10)) (4) (17))))
```

should return

```
'(11 9 5 1 7 10 4 17)
```

You must write a recursive definition of fringe that use the map and fold functions. Look at the Functional-Languges-Higher-Order-Functions lecture and the slides from that lecture if you're not sure what map and fold are. `map` is a Scheme built-in function and the Scheme implementation for `fold` can be found on the slides. I would use the map function to apply fringe to the list of sub-trees for a tree and then use fold to append the lists returned by each invocation of fringe into a single list.

7. In this problem you are going to create a "stream" that lazily generates fibonacci numbers and you are going to create a function named printFib that prints the first n elements of this list. To complete this problem, you need to write two pieces of Scheme code:

a. Define a "stream" named **fib** that lazily creates a list of fibonacci numbers, assuming that f0 is 0 and f1 is 1. Use the natural number example from the functional languages evaluation slides (slide 9) as a template. Note that you will need to use Scheme's `delay` function. Here is a good example of a fib function to modify:

```
(define fib
  (lambda (n)
    (letrec ((fib-helper
                (lambda (f1 f2 i)
                  (if (= i n)
                      f2
                      (fib-helper f2 (+ f1 f2) (+ i 1))))))
        (fib-helper 0 1 0))))
```

b. Define a function named **printFib** that takes two arguments, the **fib** stream object you defined in part a and a number n that represents the number of fibonacci numbers to print. Print these fibonacci numbers. For example:

```
(printFib fib 10) ==> 0 1 1 2 3 5 8 13 21 34
```

My solution used the display function to print the numbers and the let construct to ensure that the display functions were executed in the correct order (i.e., the display function that prints a number was executed before the display function that printed a space). My let statement used an empty list for the name-value bindings. Note that you will need to use Scheme's `force` function. Again look at the natural number example as a template for using the force function. However, your print function will need to be more sophisticated than the simple `tail` function defined in that example.

8. **Extra Credit (30 points)**: Do the online TNVoice evaluation for me and any of the TAs you wish to, and then upload a screenshot of the confirmation page to the Canvas TNVoice link by Wed. April 28 at 11:59pm.

What to Submit

1. The functions for the two parts should be placed in two files named `hw9_pt1.scm` and `hw9_pt2.scm`. We will load this file into the scheme read-eval-print loop and call your functions with test data to check them.
2. Your TNVoice confirmation page if you choose to complete it.