CSCI 320 Concepts of Programming Languages
G. Pothering
Assignment 2
Spring 2019 – Due February 14, 2019

The following exercises provide more exposure to Scheme.

1. Implement fully the Complex Number representation we discussed in class. In particular, besides the function `real_part`, `complex_part` and `make_complex` we defined in class, you should implement the following definitions. Whereever you see a parameter with **`num`** in it, we are assuming it is a complex number.

   `(compl num)` implements the operation $\overline{x + yi}$ from the handout.

   `(abs num)` implements the operation $|x + yi|$ from the handout.

   `(equal? num1 num2)` returns #t if num1 and num 2 represent the same complex number and returns #f otherwise.

   `(plus num1 num2)` implements the operation $(x_1 + y_1i) + (x_2 + y_2i)$from the handout.

   `(minus num1 num2)` mplements the operation $(x_1 + y_1i) - (x_2 + y_2i)$from the handout.

   `(prod num1 num2)` mplements the operation $(x_1 + y_1i) \times (x_2 + y_2i)$from the handout.

   `(quotient num1 num2)` implements the operation $\frac{x1+y1i}{x2+y2i}$from the handout.

2. Define a function `permutation?` that takes as parameters two lists, `list1` and `list2` and returns `#t` if the lists are permutations of one another, and returns `#f` otherwise. Thus

   ```
   (permutation? '(1 3 2) '(3 1 2)) would return #t
   (permutation? '(1 3 2) '(1 2)) would return #f
   (permutation? '(1 3 3) '(1 3 2)) would return #f
   (permutation? '(1 3 2) '(1 2 3 a)) would return #f
   ```

   Note that (`permutation? '() '())` would return `#t.`

   You may assume as known any functions we defined in class or in an assignment, so you do not have to redefine these.

3. We can represent a binary tree in Scheme as follows:

   - `()` will represent the empty tree; otherwise
   - Each node of a binary tree non-empty tree will be of the form `(atom list1 list2)`, where `atom` represents the nodes value and `list1` and `list2` are lists (possibly `empty`) representing the left and right subtrees respectively; for example

     ```
     (a (b () (c () ()))(d () (e (f () ()) ())))
     ```

   a. Define a function `tree?` that accepts a list as an argument and returns `#t` if the list is a tree and `#f` otherwise.

   b. Define a function `preorder` that accepts a tree as a parameter and returns a list containing the node values of the tree based on a preorder traversal. When applied to the above tree, `preorder` would return the list `(a b c d e f)`

   c. Same as b. except give an `inorder` traversal. When applied to the above tree, `inorder` would return the list `(b c a d f e)`

   d. Same as b. and c. except give a `postorder` traversal. When applied to the above tree, `postorder` would return the list `(c b f e d a)`