CSSE 304 Assignment 7 162 points Updated for Spring, 2016

Objectives: You should learn

- More about list processing.
- More about the use of let, letrec, named let, map, and apply.
- How to base recursive programs on recursive datatype definitions.

This is divided into two parts, but each part is substantial. Start early. Especially problems 5 and 7

.

Details for these instructions are in the previous assignment, so not repeated here: Individual assignment**.** Comments at beginning, before each problem, and when you do anything non-obvious. Submit to server (test offline first). Your code must not mutate (unless a particular problem calls for it), read, or write anything. Assume arguments have correct form unless problem says otherwise.

Abbreviations for the textbooks:

EoPL - Essentials of Programming Languages, 3rd Edition
TSPL - The Scheme Programming Language, 4rd Edition
EoPL-1 - Essentials of Programming Languages, 1st Edition (handout)

Reading Assignment: See the schedule page. Have you been keeping up with the reading?

Problems to turn in: For many of these, you will want to write one or more helper procedures.

Some of the problems come from Chapter 1 of EoPL (3rd edition), which you should have been reading already.

#1 (10 points) `vector-append-list` `(vector-append-list v lst)` returns a new vector with the elements of `lst` attached to the end of `v`. Do this without using `vector->list`, `list->vector`, or `append`.
For this problem only, you can and should use mutation: namely the vector-set! procedure. Note that vector-set! does not return a value.

#2 (20 points) Write `qsort`. `(qsort pred ls)`, a Scheme procedure whose arguments are
a predicate (total ordering) which takes two arguments `x` and `y`, and returns `#t` if `x` is "less than" `y`, `#f` otherwise.
a list whose items can be compared using this predicate.
`qsort` should produce the sorted list using a QuickSort algorithm (write your own; do not use Scheme's `sort` procedure).

For example:

```
(qsort < '(4 2 4 3 2 4 1 8 2 1 3 4)) ➔ (1 1 2 2 2 3 3 4 4 4 4 8)

(qsort (lambda (x y) (< (abs (- x 10)) (abs (- y 10))))
       '(5 1 10 8 16 17 23 -1))
     ➔ (10 8 5 16 17 1 -1 23)
```

If you do not remember how QuickSort works, see http://en.wikipedia.org/wiki/Quicksort or Chapter 7 of the Weiss book used for CSSE230. There are quicksort algorithms that do fancy things when choosing the pivot in order to attempt to avoid the worst case. You do not need to do any of those things here; you can simply use the `car` of the list as the pivot. Since mutation is not allowed, your algorithm cannot do the sort in-place. Furthermore, you are not allowed to copy the list elements to a vector, then sort the vector and copy back to a list. All of your work should be done with lists.

#3 (15 points) Write a Scheme predicate `(connected? g)` that takes an undirected graph (represented in the same way as in previous assignments) and determines whether it is connected. A graph is connected if every vertex can be reached from every other vertex *via* a sequence of edges. Starting point: the null graph and the graph with one vertex are connected. For example:

```
(connected? '((a (c)) (b (c)) (c (a b))))          ➔ #t
(connected? '((a ()) (b (c)) (c (b))))             ➔ #f
(connected? '((a (b)) (b (a)) (c (d)) (d (c))))    ➔ #f
```

#4 (10 points) Write `(reverse-it lst)` that takes a single-level list `lst` and returns a new list that has the elements of `lst` in reverse order. The original list should not be changed. Can you do this in O(n) time? You probably cannot if you use `append`. Obviously, you should not use Scheme's `reverse` procedure in your implementation.

#5 (40 points) **Examples are in the test cases**. A Binary Search Tree (BST) datatype is defined on page 10 of EoPL. Define the following procedures:

1. `(empty-BST)` takes no arguments and creates an empty tree, which is represented by an empty list.
2. `(empty-BST? obj)` takes a Scheme object `obj`. It returns `#t` if `obj` is an empty BST and `#f` otherwise.
3. `(BST-insert num bst)` returns a BST *result*. If `num` is already in `bst`, *result* is structurally equivalent to `bst`. If `num` is not already in `bst`, *result* adds `num` in its proper place relative to the other nodes in a tree whose shape is the same as the original. Like any BST insertion, this should have a worst-case running time that is O(height(`bst`)).
4. `(BST-inorder bst)` should (can we do it in in O(N) time?) produce an ordered list of the values in `bst`.
5. `(BST? obj)` returns `#t` if Scheme object `obj` is a BST and `#f` otherwise.
6. `BST-element`, `BST-left`, `BST-right`. Accessor procedures for the parts of a node.
7. `(BST-insert-nodes bst nums)` starts with tree `bst` and inserts each integer from the list `nums`, in the given order, returning the tree that includes all of the inserted nodes. The original tree is not changed, and this procedure does no mutation.
8. `(BST-contains? bst num)` determines, in time that is O(height(`bst`)), whether `num` is in `bst`.

#6 (10 points) Write (`map-by-position fn-list arg-list`) where

- `fn-list` and `arg-list` have the same length,
- and each element of the list `fn-list` is a procedure that can take one argument,
- and each element of the list `arg-list` has a type that is appropriate to be an argument of the corresponding element of `fn-list`.

`map-by-position` returns a list (in their original order) of the results of applying each function from `fn-list` to the corresponding value from `arg-list`. **You must use `map` to solve this problem; no explicit recursion is allowed.**

```
(map-by-position (list cadr - length (lambda(x)(- x 3)))
                 '((1 2) -2 (3 4) 5))                    ➔ (2 2 2 2)
```

#7 (57 points) Consider the following syntax definition from page 9 of EoPL:

```
<bintree> ::= <integer>  | ( <symbol> <bintree> <bintree> )
```

Write the following procedures:

- `(bt-leaf-sum T)` finds the sum of all of the numbers in the leaves of the `bintree` `T`.
- `(bt-inorder-list T)` creates a list of the symbols from the *interior* nodes of T, in the order that they would be visited in an inorder traversal of the binary tree.
- `(bt-max T)` returns the largest integer in the tree.
- `(bt-max-interior T)` takes a binary tree with at least one interior node, and returns (in O(N) time, where N is the number of nodes) the symbol associated with an interior node whose subtree has a maximal leaf sum (at least as large as the sum from any other interior node in the tree). If multiple nodes in the tree have the same maximal leaf-sum, return the symbol associated with the leftmost maximal node.

This `bt-max-interior` procedure is trickier than it looks at first!

You may not use mutation.
You may not traverse any subtree twice (such as by calling `bt-leaf-sum` on every interior node).
You may not create any additional size-O(N) data structures that you then traverse to get the answer.
Think about how to return enough info from each recursive call to solve this without doing another traversal.

Note: We will revisit this linear-time max-interior problem several times later. If you do not get this version, the later versions will be harder for you, so you should do what it takes to get this one.