CSE 1729 - Introduction to Principles of Programming October 23, 2018

Laboratory Assignment 8

Objectives

- More practice with lists
- Learn about binary trees
- Have a lab that doesn't suck

Activities

1. Define a Scheme function `has-duplicates?` which takes a list and returns `#f` if no two of its members are equal, `#t` otherwise. *Hint:* Use the `member` function.

2. Define a Scheme function `num-zeroes` which takes a (possibly nested) list, and return the number of zeros in the list. You should also include any zeroes in nested sublists in the count. The list may include items of any type, and should count both 0 and 0.0 as zero.

3. (a) Define a Scheme function `(insert item lst precedes)` that inserts a value into a sorted list of values such that the list remains sorted. `item` is the thing to insert, `precedes` is a function of two variables such that `(precedes a b)` is `#t` if `a` should precede `b` in a sorted list. The key insights are that if `item` precedes the `car` of `lst`, then the result is `(cons item lst)`; if that is not the case then the result is a list containing the first element of `lst` followed by the result of inserting `item` into the tail of `lst`.

   Examples:

   ```
   > (insert 4 '() <)
   (4)
   >(insert 7 '(2 4 6 8) <)
   (2 4 6 7 8)
   >(insert 7 '(6) >)
   (7 6)
   >(insert 7 '(6) <)
   (6 7)
   ```

   (b) Using your `insert` function, define a function `(insert-all lst precedes)` which successively inserts the elements of `lst` into a result list (initially `'()`) until all elements have been inserted; that result is what the function should evaluate to.

   Examples:

   ```
   > (insert-all '(1 2 3 4 5)  >)
   (5 4 3 2 1)
   > (insert-all '(1 2 3 4 5)  <)
   (1 2 3 4 5)
   ```

4. This question involves working with binary trees (trees where a node may 0, 1, or 2 children). For these questions, we will assume that the value at each node is a number, which is necessary for part 4b below. Recall the conventions we have adopted in class for maintaining trees. We represent the empty tree with the empty list `()`; a nonempty tree is represented as a list of three objects

   ```
   (value left-subtree right-subtree)
   ```

   where `value` is the value stored at the root of the tree, and `left-subtree` and `right-subtree` are the two subtrees. We introduced some standardized functions for maintaining and accessing this structure, which you should use in your solutions below.

   ```
   (define (make-tree value left right)
    ;; produces a tree with value at the root, and
    ;; left and right as its subtrees.
      (list value left right))

   (define (value tree)
      (car tree))

   (define (left tree)
      (cadr tree))

   (define (right tree)
      (caddr tree))

   (define empty-tree? null?)
   ```

   Notably, you will need to use these functions to produce some test trees to test the following functions. For example, to produce the simple tree in Figure 1, you could use the following code:

   ```
   (define testtree
      (make-tree
         1
         (make-tree
            3
            (make-tree 7 '() '())
            (make-tree 9 '() '()))
         (make-tree 5 '() '())))
   ```

   You will probably want to make some more complex trees than that, but you get the idea.

   Finally, for all of these, figure out how to define the function recursively, as that should map fairly directly to code.

   (a) A non-empty tree is made up of nodes: the root, and all of the nodes in its subtrees. Define a Scheme procedure `(tree-node-count t)` that calculates the number of nodes in tree t. It should also work for empty trees. `(tree-node-count testtree)` would be 5.

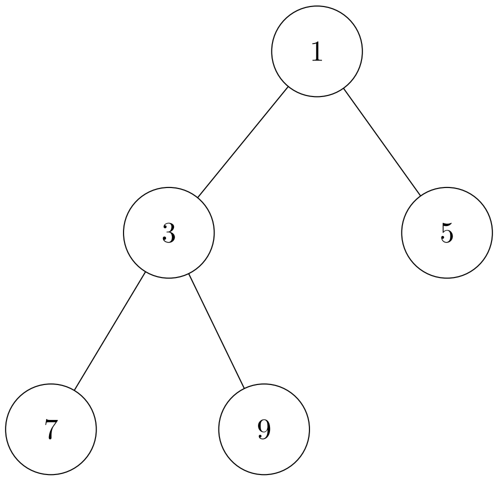

Figure 1: Tree `testtree` produced by code in question 4.

(b) Each node in a tree (in this problem) has a number as its value. Define a Scheme procedure `(tree-node-sum t)` that calculates the sum of the values of the nodes in nodes in tree t. It should work for any binary tree whose node values are numbers, as well as for empty trees. `(tree-node-sum testtree)` would be 25.

(c) The height of a node in a tree is the length of the longest path from that node to a leaf – we can consider the height of a tree to be the height of its root. The height of the empty tree is undefined, the height of a node without children (leaf) is 0, otherwise the height of a node is one more than the maximum height of the trees rooted at its children.

Define a Scheme procedure `(tree-height t)` that calculates the height of a non-empty tree t. `testtree` has height 2.

Note: you are free to choose any behavior you want for `(tree-height '())` (including bombing) as long as you get the correct answer for the height of any non-empty tree.

(d) Define a Scheme procedure, named `(tree-map f t)`, which takes two parameters, a function, `f` and a tree `t`, and is analogous to the `map` function for lists. Namely, it returns a new tree with a topology identical to that of `t` but where each node in the new tree has the value $f(v)$, where v is the value at the corresponding position in t. For example, if the input tree is `testtree` shown in Figure 1 then the tree returned by `(tree-map (lambda(x)(* 3 x)) testtree)` is shown in Figure 2.

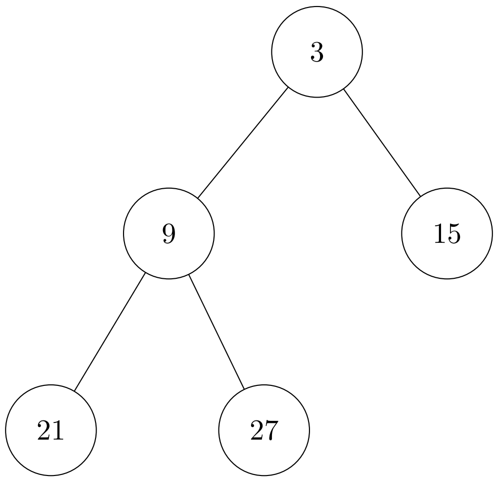

Figure 2: Tree resulting from mapping `(lambda(x)(* x 3))` over `testtree`.