CSE 1729 - Introduction to Principles of Programming October 24, 2017

Laboratory Assignment 8

Objectives

- Mergesort and binary trees

Activities

1. `mergesort` works recursively by splitting its input in halves, sorting each half recursively using mergesort, and using merge to combine the two sorted halves. In English: if the list has zero or one element,you are done, otherwise split the list into 2 equal-sized sublists, sort each sublist, and merge them into a sorted result. Notice that unlike Quicksort there is no pivot: we split the list so the two parts are equal size (within 1) without regard to the values in the list.

   You wrote the `merge` function in Problem Set 6. Feel free to use that code, or this:

```
(define (merge la lb)
  (cond ((null? la) lb)
        ((null? lb) la)
        ((< (car la)(car lb))
         (cons (car la)(merge (cdr la) lb)))
        (else
          (cons (car lb)(merge la (cdr lb))))))
```

   (a) Write a function `(split l)` that takes a list of numbers and partitions it into two equal-sized (within one) lists, and returns a list whose first element is the first list and whose second element is the second list.

   You can use the built-in `length` function if you would like, but it should not be called more than once (in total) during `split`.

   (b) Using `split` and your `merge` function from last week (or the one listed above), write a function `(mergesort l)` as described above that takes a list of numbers and returns it sorted in ascending order.

2. This question involves working with binary trees (trees where a node may 0, 1, or 2 children). 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))
```

   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.

   (b) Each node in a tree has a 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

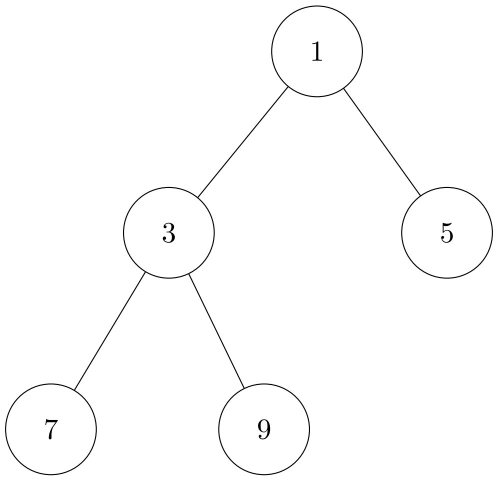

Figure 1: Tree `testtree` produced by code in question 2.

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.

(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.

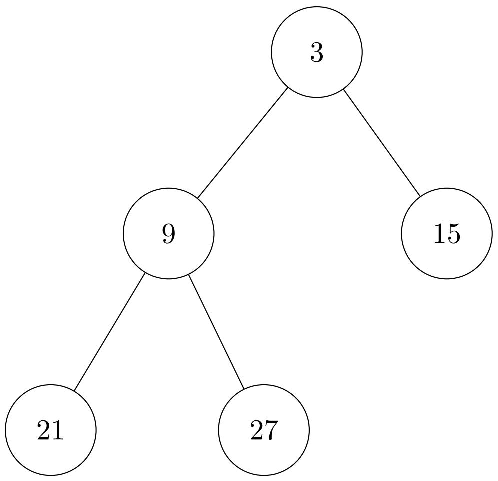

Figure 2: Tree resulting from mapping `(lambda(x)(* x 3))` over `testtree`.