Lab 2: Functional Programming

Instructors: Zoran Budimlić, Mackale Joyner

Course wiki: `http://comp322.rice.edu`

Staff Email: comp322-staff@rice.edu

Goals for this lab

- Experiment with the functional programming concepts we learned in class, specifically the implementation of higher-order functions.

Due: Wednesday, January 26th at 4:30pm

Downloads

As with lab 1, the provided template project is accessible through your private GitHub classroom repo at: `https://classroom.github.com/a/mImrs6Kk`

For instructions on checking out this repo through IntelliJ or through the command-line, please see the Lab 1 handout. The below instructions will assume that you have already checked out the lab2 folder, and that you have imported it as a Maven Project if you are using IntelliJ.

1 Functional Trees

In this lab, you are given starter code for functional trees. The functional trees are implemented using a very similar approach to the one we used in class to implement functional lists. A `Tree` is a generic interface that has two implementations: `Node` and `Empty`. A `Node` is a class that represents a node in a tree. Each node has some generic value (`private final T value`), and a list of children (`private final GList<Tree<T>> children`). Each child is another tree. If the list of children is empty, then the node is a leaf node.

You are also given a `GList` implementation that we discussed in class, both to use as a template to see how these higher-order functions could be implemented on generic functional data structures, and to use in your implementation (since the children in each node are implemented as a `GList`.

In this lab, you will need to edit `Tree.java` to produce a correct implementation for `map` and `filter` methods on generic functional trees.

In `Tree.java` you will find that the `map` and `filter` are already implemented for the class `Empty` (they return an empty tree). Running `LabCorrectnessTest.java` will verify the correctness of your solution.

1.1 Implementing `Map`

This is the easier of the two problems, as the structure of the tree does not need to change. All you need to do is map every value in every node to the new value, using the function `f` that is passed as the argument to `map`.

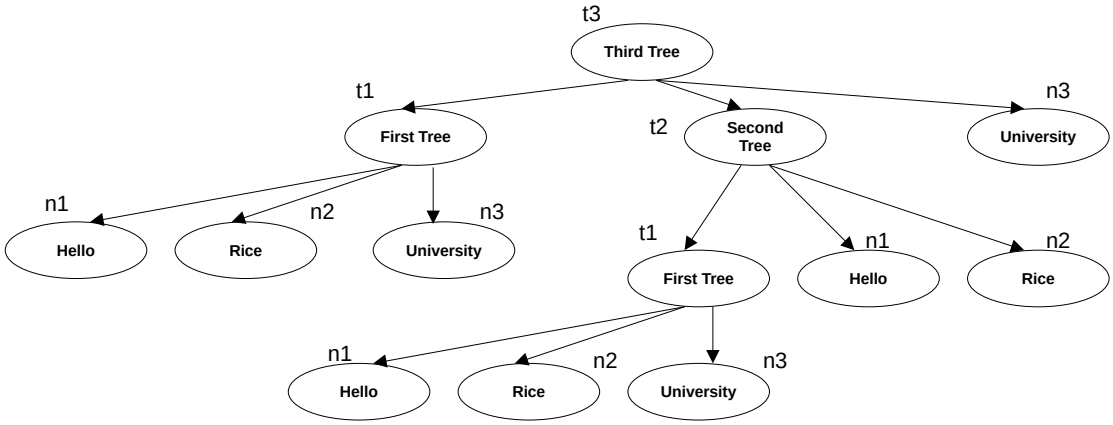
`map` is purely functional. Its result should be a new tree that has the exact same structure as the original tree, but the values in each node of the result tree are the result of applying the function `f` on the corresponding node in the original tree. There is no mutation here, you should not modify the original tree in any way. You may find the `Tree.makeNode` method useful for creating new nodes for the result tree.

Make sure you traverse the whole tree to apply the mapping function. You may find the method `GList.map` useful when traversing the children of a node.

Take a look at the `Lab2CorrectnessTest.testMap()` method. Here, we are creating a tree to test the map function as follows:

```
var n1 = makeNode("Hello", GList.empty());
var n2 = makeNode("Rice", GList.empty());
var n3 = makeNode("University", GList.empty());
var t1 = makeNode("First Tree", GList.of(n1, n2, n3));
var t2 = makeNode("Second Tree", GList.of(t1, n1, n2));
var t3 = makeNode("Third Tree", GList.of(t1, t2, n3));
```

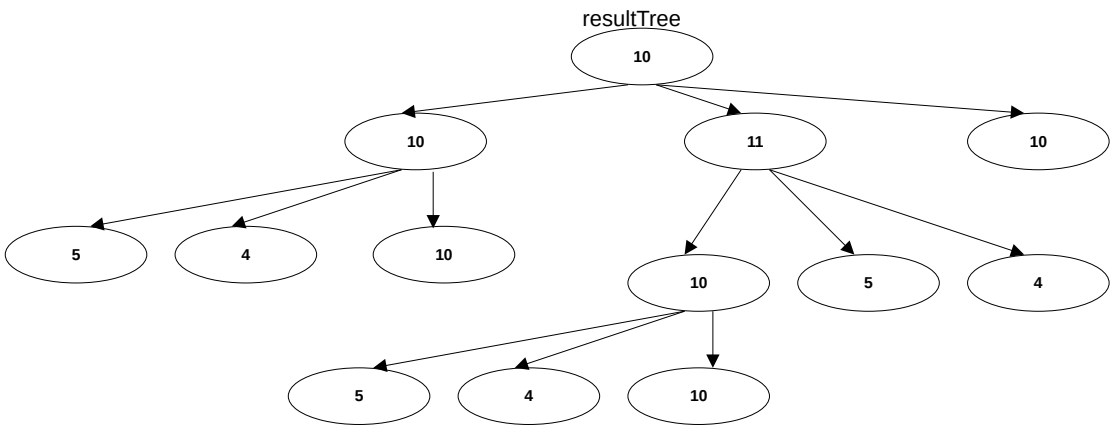
This creates a tree `t3` that looks like this:



Now, when we map this tree using the following call:

```
var resultTree = t3.map(String::length)
```

We should get a new tree `resultTree` that looks like this:



Notice that the `resultTree` has the exact same structure as `t3`, but the value of each node in `resultTree` is the lenght of the string in the corresponding node in `t3`.

1.2 Implementing `Filter`

Implementing filter is going to be a bit trickier, since we will have to restructure the resulting tree. If the predicate `p` returns `false` when applied to a value in a node, then that node should not appear in the result tree. But what happens at the position that that node was occupying? We have to find a replacement for that node, if one exists.

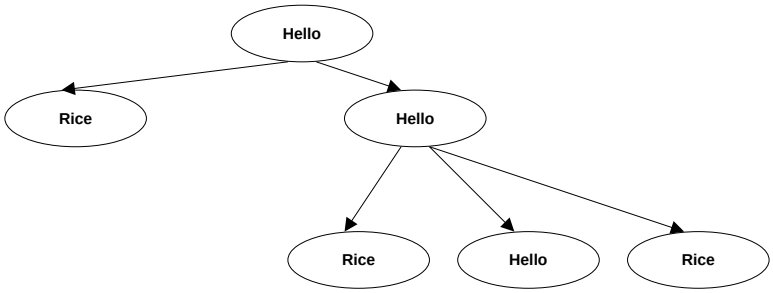
We will be using a very simple approach for restructuring the tree. If a node is filtered out, then we will replace it with its first child.

What if that first child needs to be filtered out as well? Well, it will get replaced by its own first child. And so on... Hopefully, you see the recursive structure of the approach here. If all the children and their subtrees of a node that is filtered out are also filtered out, then the node and its whole subtree do not appear in the output.

For example, let's say we want to filter out all the nodes in the tree `t3` above that contain a string of length of more than 5. I.e., we want to keep all the nodes containing strings of less than 6. We would do this using filter like so:

```
filteredTree = t3.filter(s -> s.length() < 6);
```

The resulting `filteredTree` should look like this:



Let's see how we got this result. The node at the root ("Third Tree") needs to be filtered out, so it gets replaced by the first child. But the first child ("First Tree") is also filtered out, so it gets replaced by its own first child ("Hello"). That's now the root of the result tree. Only node `n2` ("Rice") is what's left over from the whole `t1` subtree.

Similarly, the node `t2` ("Second Tree") is filtered out, and needs to be replaced by its first child. But the first child, node `t1` ("First Tree") is also filtered out, so it's replaced by its own first child, node `n1` ("Hello").

The third child of the original root, node `n3` ("University") gets filtered out.

To help you out, we have given you a helper function `makeTreeHelper`, that, given a `GList` of trees, makes one single tree out of them, by making the fist tree in the list the root tree, and appending all the remaining trees to the children of the first tree.

Remember, think recursively! We filter the tree by filtering all its children. Filtering some of those children may result in empty trees, so we'll need to get rid of those. Then, if needed (if the root of the tree needs to be filtered out as well), we can just make one single tree out of the list of children by calling the provided helper function.

The reference solution is 2 lines of code, one to compute the filtered children of the node, and the other one to compute the final filtered result for the current node.

2 Demonstrating your lab work

Show your work to an instructor or TA to get credit for this lab during lab or office hours by Wednesday, January 26th at 4:30pm. They will want to see your updated files committed and pushed to GitHub in your web browser, and the passing unit tests on your laptop.