CSCE 314 Programming Languages

More Information for Problem 2 of Homework 3

Hyunyoung Lee

Consider the following data type for a binary tree:

```
data Tree a b = Leaf a | Branch b (Tree a b) (Tree a b)
```

Problem 2. (15 points) Make `Tree` an instance of `Show`. Do not use `deriving`; define the instance yourself. Make the output look somewhat nice (e.g., indent nested branches).

The data type `Tree` has two type parameters, `a` and `b`. A tree object of type `Tree` can be a leaf or a branch (internal node). A leaf is constructed using the `LeafTree` constructor, and stores a value of type `a`. A branch is constructed using the `Branch` constructor, stores a value of type `b` at the internal node, and has two subtrees – the left-subtree and the right-subtree. These subtrees are then recursively constructed to be either a leaf or another (level of) branch.

The problem statement, "Make `Tree` an instance of `Show`. Do not use `deriving`" means that you are to provide a definition for the `show` function that is a member function of the `Show` type class. It is somewhat analogous to, in C++, when you declare your own class (data type) for a binary tree, you usually overload the insertion operator `<<` for your binary tree so that when you do, for example, `cout << myTreeObject`, your tree object can be displayed on the screen as a character string but still looking a bit like a tree. If you are familiar with Java, it is analogous to overriding the toString() method in your class.

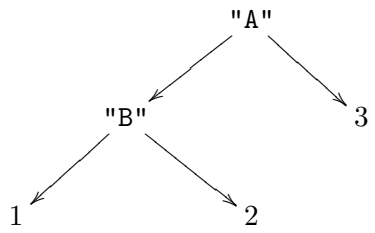
There, you give essentially a text representation of how your tree should be displayed in one string, that is, you will include `\n` for the newline character or spaces or `\t` (the tab character) for indentation in your output string to express the depth of the tree.

To understand the syntax of how to make the `Tree` data type an instance of the `Show` type class in Haskell, please read my lecture slide# 31 of haskell-05-declaring-types.pdf that gives an example of how you would define the `show` function on a recursive data type, *using a helper function*, and textbook section 8.4 to understand the *tree* data type in general, and textbook section 9.3 (page 113) for another example of defining the `show` function.

In the hw3-skeleton.hs file, I provided some example `Tree` objects, namely `tree1`, `tree2`, and `tree3`. For example, consider the `tree4` object:

```
tree4 = Branch "A"
            (Branch "B"
               (Leaf 1)
               (Leaf 2))
            (Leaf 3)
```

If I draw a picture of `tree4`, it would look like this:



But in a text format, it can be displayed as side-ways (below are possible outputs of `tree4` once you define the `show` function correctly, and trust me on that displaying the tree side-ways as below will be easier than displaying it the more natural way as in the picture above):

```
"A"
     3
     "B"
          2
          1
```

or

```
"A"
     "B"
          1
          2
     3
```

or something similar. That is, you need to express each level (depth) of the tree by use of indentation, and the nodes at the same depth (assuming that the root is at depth zero) should be output with the same amount of indentation. Furthermore, the order of left- and right-subtrees that are displayed must be consistent in each level. For example, the first output above displays right-subtree first and then left-subtree in each level, looking as though the tree in the picture above is 90° rotated counter-clock wise, whereas, the second output shows left-subtree first and then right-subtree in each level.

Here the key idea is that, in order to have the output more indented as the level gets deeper, you need to somehow be able to accumulate the indentation along the recursive calls, that is, you need to be able to pass along the amount of indentation (or the depth information) of the current level to the next recursive call of the function with the next subtree as the tree argument.

However, the `show` function takes only one argument that is the tree object to be displayed. Thus, you will need a local helper function that can take more than just the tree object as its arguments, and can pass along the current amount of indentation to the next level of recursion, so each recursive call will add more indentation, and when the recursion returns,

then the amount of indentation is decreased (to the previous level). The example in the lecture slide# 31, the `showRest` function is a local helper function that takes care of outputting the rest of the list. As such, you can define a local function that can take as many arguments as you need.

Once you define your `show` function, you can simply do in ghci,

```
*Main> tree4
```

then, it should display the `tree4` object using your implementation of the show function.

A cautionary remark:

Make sure you *don't* explicitly invoke the show function such as

```
*Main> show tree4  -- don't do this!
```

This will display the string representation that is the output of the show function itself because you are asking ghci to display the result of `show tree4` not just the value of `tree4`.

Have fun!