Computer Science 320 (Fall, 2016)
Concepts of Programming Languages

Homework Assignment 3: Recursive Data Types

Out: Tuesday, September 27, 2016
Due: Wednesday, October 12, 2016, by 11:59 pm

In this assignment, you will implement a binary tree data structure in two languages: one in Haskell and one in Python.

The portion of your solution in Haskell should be defined within a module named `Tree`, and you should submit the file `Tree.hs` (or `Tree.lhs`). **File names are case sensitive.** The portion of your solution in Python should be defined within a file named `tree.py`. Verbal responses to non-programming questions should be in the form of comments in your code.

Note: The solution to each part of the Haskell problems is meant to be extremely short (one to four lines). You may (and should) use functions from earlier problems in this assignment to solve parts of later problems in this assignment. Unless otherwise specified, your solutions may utilize functions from the standard prelude as well as material from lecture notes.

(20 pts) **Problem 1.**

(a) Define a polymorphic data type called `Tree a` that has two constructors: `Leaf :: Tree a` and `Node :: a -> Tree a -> Tree a -> Tree a`. Trees of this type will contain a single piece of data of type `a` at each node, and no data at their leaves.

(b) Modify your data type definition so that the `show` and `(==)` functions are both defined for it.[1]

(c) Define a function `mapT :: (a -> b) -> Tree a -> Tree b` that operates on trees the same way that `map` operates on lists. In other words, `mapT` should take a function and apply it to every data item of type `a` within a tree of type `Tree a`.

(d) Define a function `foldT :: (a -> b -> b -> b) -> b -> Tree a -> b` that operates on trees the same way that `foldr` operates on lists. In other words, the base item of type `b` should replace the leaf constructor in the tree, and the function of type `a -> b -> b -> b` should replace the node constructor in the tree.

[1]Hint: This should be done as part of the original `data` expression by adding a `deriving` clause. See, for example, Chapters 14 and 16 in the book *The Craft of Functional Programming* by Simon Thompson.

(20 pts) **Problem 2.**

(a) Define a function `leafCount :: Tree a -> Integer` that returns an integer representing the total number of leaves in a tree. To get full credit, you must use `foldT` and may define at most one helper function.

(b) Define a function `nodeCount :: Tree a -> Integer` that returns an integer representing the total number of non-leaf nodes in a tree. To get full credit, you must use `foldT` and may define at most one helper function.

(c) Define a function `height :: Tree a -> Integer` that returns an integer representing the height of the tree. Trees consisting of only a leaf have height `0`. To get full credit, you must use `foldT` and may define at most one helper function.

(30 pts) **Problem 3.**

(a) A tree is *perfect* if all the leaves of the tree are at the same depth. Define a function `perfect :: Tree a -> Bool` that returns `True` if the tree supplied is perfect and `False` otherwise. You may use any approach to implement this function.

(b) A tree is a *degenerate* if all the nodes are arranged in a single path. Equivalently, a tree is degenerate if all nodes have at least one leaf child. Define a function `degenerate :: Tree a -> Bool` that returns `True` if and only if the tree supplied is degenerate.[2]

(c) Define a function `list :: Tree a -> Maybe [a]`. If the supplied tree is degenerate the function should return `Just l`, where `l` corresponds to a list constructed by replacing the `Node` constructors with `(:)` constructors and the `Leaf` constructors have been replaced with the `[]` constructor where appropriate. If the supplied tree is not degenerate, the function should return `Nothing`. You are encouraged to use `degenerate` and `foldT` to implement this function.[3]

(30 pts) **Problem 4.**

(a) Implement in an imperative language a data structure definition similar to `Tree a`.

(b) Add functions or members to the data structure definition corresponding to the functions `(==)`, `leafCount`, `nodeCount`, `height`, `perfect`, `degenerate`, and `list` (the last function can return an array or other list-like data structure instance if possible, and `null` or something equivalent otherwise). You may use any approach to implement these functions.

(c) Identify two advantages of higher-order functions (such as `foldT`) within the context of processing tree data structures.

[2]Hint: You do not need to make your function recursive if you use functions you have already defined. How many leaves does a degenerate tree have? How many nodes?

[3]Hint: Do not make your use of `foldT` more complicated than necessary. Do you need to check that the tree is degenerate more than once? Use your `nodeCount` implementation as a guide.

(15 pts) **Problem 5.** (Bonus, which you do not need to solve for the maximum of 100 pts)

(a) In your `Tree` module, define a new data type called `Color` that has four constructors: `Red::Color`, `Green::Color`, `Blue::Color`, and `Yellow::Color`.

(b) Define a value `infColorTree :: Tree Color`, an infinite tree in which every node has a color, and the three neighbors of that node (the parent and its two children) have three distinct colors which are different from the color of the node. (That is, for every node v, the two children of that node must have different colors from each other as well as from v, and the parent of v must not share a color with either of the two children of v, nor with v itself).