Functional Programming 1 — 1DL330

Assignment 3

Lab: Friday, 27 September

Submission Deadline: 18:00, Wednesday, 2 October, 2019

This assignment should be solved in **groups of two** students. Please join a group in the *Assignment 3* group division on the Student Portal. If you have not found a group partner by the start of the lab for this assignment, please inform the lab assistant. If you would prefer to solve this assignment individually due to special circumstances, please contact the course teacher, Tjark Weber.

Instructions

- Start by reading the General Lab Instructions (on the Student Portal).
- Download the file `Lab3.hs` from the Student Portal. Modify this file as needed to add your solutions. Do *not* rename the file.
- The file that you submit must be a valid Haskell file, so the answers to some of the questions need to be given as comments using `--...` or `{- ... -}`.
- Remember to follow our Coding Convention (also on the Student Portal), and provide a specification for every function that you write. Also provide a variant for every recursive function.
- Make sure that your solutions pass the tests in `Lab3Tests.hs` before you submit. A submission that does not pass these tests will get grade K. See the *Testing* section near the end of this assignment for details.

1 Fruit

We consider three kinds of fruit: apples, bananas and lemons. Bananas and apples are sold by the kilogram, but a lemon always has the same price, regardless of weight.

1. Give a datatype declaration for the type `Fruit`. The datatype should reflect that there are three kinds of fruit, so the datatype definition should have three cases. For apples (`Apple`) and bananas (`Banana`) there should be an associated weight (given as a value of type **`Double`**). For lemons (`Lemon`) there should be an associated number of units (given as a value of type **`Integer`**).

2. Define a function
   ```
   sumPrice :: [Fruit] -> Double -> Double -> Double -> Double
   ```
   This function should take a list of fruit, the price of apples (per kilogram), bananas (per kilogram), and the price of lemons (per unit), and return the total cost of the items in the list.

2 Binary Search Trees

The nodes of a binary search tree are ordered from left to right according to their key value. Consider the following datatype definition of binary search trees:

```
data BSTree = Void | BSNode BSTree Integer BSTree
```

Define a function

```
subTree :: Integer -> Integer -> BSTree -> BSTree
```

where `subTree a b t` yields a binary search tree containing the keys in `t` that are greater than or equal to `a` but smaller than `b`.

Example: Assume that `t` is defined as follows.

```
t = BSNode (BSNode (BSNode Void 0 (BSNode Void 2 Void))
                   3
                   (BSNode Void 5 Void))
           6
           (BSNode Void
                   7
                   (BSNode Void 8 (BSNode Void 9 Void)))
```

We now have

```
subTree 5 8 t == BSNode (BSNode Void 5 Void)
                        6
                        (BSNode Void 7 Void)
```

(The tree produced by your solution might be shaped differently, but it should also contain the keys 5, 6 and 7, and no other keys.) We also have

```
subTree 10 20 t == Void
```

This question can (and should) be solved without introducing intermediate data structures (such as lists).

3 Trees

1. Define a datatype `Tree a` of (finitely branching) labeled trees. Each node (`Node`) carries a label (of polymorphic type `a`) and may have an arbitrary (non-negative) number of children. Different nodes may have different numbers of children. Each tree has at least one node (so there should be no **Void** case in your datatype definition).

   See `https://en.wikipedia.org/wiki/Tree_%28data_structure%29` for further definitions and some hints.

2. Define functions to

   (a) compute the number of nodes in such a tree (`count :: Tree a -> Integer`),

   (b) compute the list of all node labels in such a tree (`labels :: Tree a -> [a]`),

   (c) compute the height of such a tree (`height :: Tree a -> Integer`).

   Hint: a possible solution is to use helper functions and mutual recursion. Another solution uses higher-order functions such as **map**.

4 Higher-Order Functions

You have already seen recursive definitions for the functions (++), **elem**, **last**, **reverse** and **filter** in class. Now your task is to give non-recursive definitions for these functions, using the higher-order functions **foldl** or **foldr**. Define

1. `(++) :: [a] -> [a] -> [a]`
2. `elem :: Eq a => a -> [a] -> Bool`
3. `last :: [a] -> a`
4. `reverse :: [a] -> [a]`
5. `filter :: (a -> Bool) -> [a] -> [a]`

In all cases, the definition using **foldl** or **foldr** should be quite simple.

Obviously, the function **last** cannot yield a meaningful result for the empty list. Make sure that applying **last** to the empty list gives an error.

Testing

Use the test cases provided in `Lab3Tests.hs` (from the Student Portal) to test your solution. See the instructions for Assignment 1 for further information on running the tests.

Some tests might fail because they assume other datatype definitions or function signatures than what you are using. If necessary, *change your solutions* to make all tests compile and run successfully.

Good luck!