CptS355 - Assignment 2 (Haskell)
Fall 2021

Assigned: Tuesday, September 28, 2021

Weight: Assignment 2 will count for 8% of your course grade.

Your solutions to the assignment problems are to be your own work. Refer to the course academic integrity statement in the syllabus.

This assignment provides experience in Haskell programming. Please compile and run your code on command line using Haskell GHC compiler. You may download GHC at https://www.haskell.org/platform/.

Turning in your assignment

The problem solution will consist of a sequence of function definitions and unit tests for those functions. You will write all your functions in the attached `HW2.hs` file. You can edit this file and write code using any source code editor (Notepad++, Sublime, Visual Studio Code, etc.). We recommend you to use Visual Studio Code, since it has better support for Haskell.

In addition, you will write unit tests using `HUnit` testing package. Attached file, `HW2SampleTests.hs`, includes at least one test case for each problem. You will edit this file and provide additional tests for problems 2(a,b) and 4(a,b,c) (at least 2 tests per function). Please use test input different than those provided in the assignment prompt. Rename your test file as `HW2Tests.hs`.

To submit your assignment, please upload both files (`HW2.hs` and `HW2Tests.hs`) on the Assignment2 (Haskell) DROPBOX on Canvas (under Assignments).

The work you turn in is to be **your own personal work**. You may not copy another student's code or work together on writing code. You may not copy code from the web, or anything else that lets you avoid solving the problems for yourself. **At the top of the file in a comment, please include your name and the names of the students with whom you discussed any of the problems in this homework**. This is an individual assignment and the final writing in the submitted file should be *solely yours*.

Important rules

- Unless directed otherwise, you must implement your functions using the basic built-in functions in the Prelude library. (You are not allowed to import an additional library and use functions from there.)
- If a problem asks for a non-recursive solution, then your function should make use of the higher order functions we covered in class (`map, foldr/foldl, or filter`.) For those problems, your main functions can't be recursive. If needed, you may define non-recursive helper functions.
- Make sure that your function names match the function names specified in the assignment specification. Also, make sure that your functions work with the given tests. However, the given test inputs don't cover all boundary cases. You should generate other test cases covering the extremes of the input domain, e.g. maximum, minimum, just inside/outside boundaries, typical values, and error values.
- Question 1 requires the solution to be tail recursive. Make sure that your function is tail recursive otherwise you won't earn points for this problem.

- You will call `foldr/foldl, map, or filter` in several problems. You can use the built-in definitions of these functions.
- When auxiliary/helper functions are needed, make them local functions (inside a `let..in` or `where` blocks). You will be deducted points if you don't define the helper functions inside a `let..in` or `where` block. If you are calling a helper function in more than one function, you can define it in the main scope of your program, rather than redefining it in the let blocks of each calling function.
- Be careful about the indentation. The major rule is "*code which is part of some statement should be indented further in than the beginning of that expression*". Also, "*if a block has multiple statements, all those statements should have the same indentation*". Refer to the following link for more information: https://en.wikibooks.org/wiki/Haskell/Indentation
- Haskell comments : `-- line comment`

```
    {- multi line
       comment-}.
```

Problems

1. groupbyNTail - 10%

Consider the `groupbyN` function below. This function is similar to the `groupBy3` function we wrote in class. It takes an input list '`iL`' and a number '`n`' and it groups `iL`'s elements into sublists of length `n`. The last sublist will include the leftover elements.
The given `groupbyN` implementation is not tail-recursive. Rewrite this function in tail-recursive form; name your function `groupbyNTail`.
The type of your `groupbyNTail` function should be same as `groupbyN`'s type, i.e.,
`groupbyNTail :: [a] -> Int -> [[a]]`

```
groupbyN iL n = grouphelper iL n []
   where
     grouphelper [] n buf = (reverse buf):[]
     grouphelper (x:xs) n buf | (length buf) >=n = (reverse buf):(grouphelper xs n (x:[]))
                              | otherwise = grouphelper xs n (x:buf)
```

Examples:

```
> groupbyNTail [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] 4
[[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15]]

> groupbyNTail "abcdefghijklmnopqrstuwxyz012" 5
["abcde","fghij","klmno","pqrst","uwxyz","012"]
```

2. elemAll and stopsAt - 20%

(a) elemAll – 10%

Write a Haskell function `elemAll` that takes two lists as input and returns `True` if all elements in the first list appear in the second list. It returns `False` otherwise. Note that both input lists should have the same type.
Your function shouldn't need a recursion but should use higher order functions (`map`, `foldr/foldl`, or `filter`). Your helper functions should not be recursive as well, but they can use higher order functions.

Note: You can use the built-in "`elem`" function in your solution.

Examples:

```
> elemAll [3,5,7,10]  [1,2,3,4,5,6,7,8,9,10]
True
> elemAll [3,5,10]  [1,2,3,4,5,6,7,8,9]
False
> elemAll ["Bishop","TerreView","Walmart"] ["Chinook","Orchard","Valley", "Maple",
"Aspen", "TerreView", "Clay", "Dismores", "Martin", "Bishop", "Walmart", "PorchLight"
, "Campus"]
True
> elemAll ["Bishop", "TerreView"] ["TransferStation", "PorchLight", "Stadium",
"Bishop" , "Walmart", "Shopco", "RockeyWay"]
False
```

(b) `stopsAt` – 10%

Pullman Transit offers many bus routes in Pullman. Assume we store the bus stops for the bus routes in a list of tuples. The first element of each tuple is the bus route and the second element is the list of stops for that route (see below for an example).

```
buses = [("Wheat",["Chinook", "Orchard", "Valley", "Maple","Aspen", "TerreView", "Clay",
                   "Dismores", "Martin", "Bishop", "Walmart", "PorchLight", "Campus"]),
         ("Silver",["TransferStation", "PorchLight", "Stadium", "Bishop","Walmart", "Shopco",
                   "RockeyWay"]),
         ("Blue",["TransferStation", "State", "Larry", "TerreView","Grand", "TacoBell",
                   "Chinook", "Library"]),
         ("Gray",["TransferStation", "Wawawai", "Main", "Sunnyside","Crestview", "CityHall",
                  "Stadium", "Colorado"])
        ]
```

Write a Haskell function `stopsAt`, that takes a list of bus stop names '`stops`' and the bus route data '`buses`' (similar to the above) as input and returns the bus route names that stop at each and every stop in '`stops`'.

Your function shouldn't need a recursion but should use higher order functions (`map`, `foldr/foldl`, or `filter`). Your helper functions should not be recursive as well, but they can use higher order functions. The order of the route names in the output can be arbitrary.

Hint: You can use the "`elemAll`" function you defined in part(a) in your solution.

Examples:

```
> stopsAt ["Bishop", "TerreView", "Walmart"] buses
["Wheat"]

> stopsAt ["Bishop", "Walmart"] buses
["Wheat","Silver"]

> p2b_test3 = stopsAt ["TransferStation", "State", "Main"] buses
[ ]
```

3. **isBigger and applyRange – 25%**

You are asked to write a program that will store some timestamp data. The timestamps are either:

- 'DATE' values including the *month, day, and year* or
- 'DATETIME' values including *month, day, year , hour, and minute.*
-

You define the following Haskell datatype to represent the timestamp values.

```
data Timestamp =  DATE (Int,Int,Int) | DATETIME (Int,Int,Int,Int,Int)
                  deriving (Show, Eq)
```

(a) **isbigger - 15%**

Write a Haskell function 'isBigger' that takes two Timestamp values as input and returns True if the first timestamp value is bigger (i.e., more recent) than the second. If the second value is bigger or if the values are equal, it should return False. 'isBigger' should be able to compare:

- DATE values with DATE values ,
- DATETIME values with DATETIME values,
- DATE values with DATETIME values, and
- DATETIME values with DATE values.

When comparing a DATE value with a DATETIME value, it should ignore the time information (i.e., hour and minute). The type of the isBigger function should be:

```
isBigger :: Timestamp -> Timestamp -> Bool
```

Examples:

```
> isBigger (DATE (5,20,2021)) (DATE (5,15,2021))
True
> isBigger (DATE (6,10,2021)) (DATE (5,15,2021))
True
> isBigger (DATETIME (6,10,2021,19,30)) (DATETIME (6,10,2021,19,10))
True
> isBigger (DATETIME (6,9,2021,19,30)) (DATETIME (6,10,2021,11,10))
False
> isBigger (DATETIME (6,10,2021,11,10)) (DATETIME (6,10,2021,11,10))
False
> isBigger (DATE (6,10,2021)) (DATETIME (6,9,2021,11,10))
True
> isBigger (DATE (6,10,2021)) (DATETIME (6,10,2021,11,10))
False
> isBigger (DATETIME (6,10,2021,11,10)) (DATE (6,10,2021))
False
```

(b) **applyRange - 10%**

Define a Haskell function applyRange that takes:

- a tuple of two Timestamp values , representing a range (the first value in the tuple is the min value and the second one is the max value of the range), and
- a list of Timestamp values

as input and returns the Timestamp values from the list that are in the given range. Assume that the range is exclusive, i.e., the Timestamp values that are equal to the min or max won't be included in the output.

Your `applyRange` function **shouldn't need a recursion but should use higher order functions** (`map`, `foldr`/`foldl`, or `filter`). You may define additional non-recursive helper functions. The order of the `Timestamp` values in the output list should be same as the other they appear in the input list.
The type of the `applyRange` function should be:

```
applyRange :: (Timestamp, Timestamp) -> [Timestamp] -> [Timestamp]
```

(Hint: You may use the `isBigger` function you defined in part(a).

Examples:

```
datelist = [DATE (5,28,2021), DATETIME (6,1,2021,14,15), DATE (6,22,2021), DATE
(6,1,2021), DATETIME (6,21,2021,15,20), DATETIME (5,21,2020,14,40),
DATE (5,20,2021), DATETIME (6,9,2021,19,30), DATETIME (6,10,2021,11,10)]

> applyRange (DATE(5,20,2021) , DATETIME(6,21,2021,19,00)) datelist
[DATE (5,28,2021),DATETIME (6,1,2021,14,15),DATE (6,1,2021),DATETIME
(6,21,2021,15,20),DATETIME (6,9,2021,19,30),DATETIME (6,10,2021,11,10)]
> applyRange (DATETIME(6,1,2021,14,20) , DATE(6,25,2021)) datelist
[DATE (6,22,2021),DATETIME (6,21,2021,15,20),DATETIME (6,9,2021,19,30),DATETIME
(6,10,2021,11,10)]
```

4. `foldTree, createRTree, fastSearch – 35%`

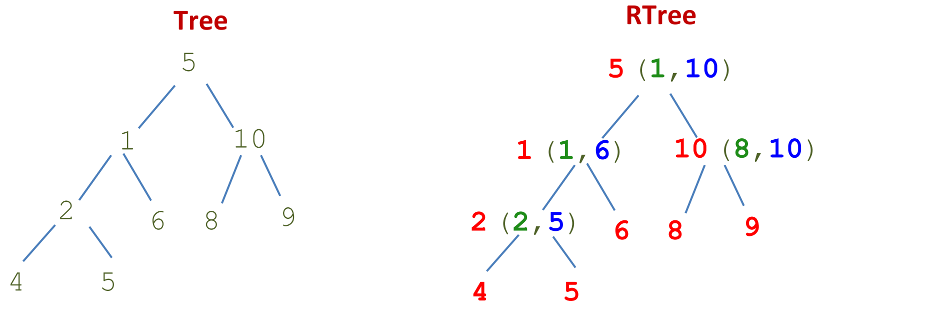
Consider the following type for a polymorphic binary tree that stores values both at the interior nodes and the leaves:

```
data Tree a = LEAF a | NODE a (Tree a) (Tree a)
                deriving (Show, Eq, Ord)
```

We now define another binary tree `RTree` type (similar to `Tree` type) where the interior nodes store both the value of the node itself and a tuple that represent the range of the values that appear in the sub-tree rooted at that node (i.e., the min and max values) .

```
data RTree a = RLEAF a | RNODE a (a,a) (RTree a) (RTree a)
                deriving (Show, Eq, Ord)
```

Example:



red : value

green : min value in the underlying sub-tree

blue : min value in the underlying sub-tree

(a) foldTree - 8%

Write a recursive function `foldTree` that takes a function 'op' and a Tree value as input and combines the values stored in NODEs and LEAFs by applying function 'op'. Note that `foldTree` doesn't take a base argument. The type of the `foldTree` function should be:

```
foldTree :: (t -> t -> t) -> Tree t -> t
```

Examples:

```
tree1 = NODE 5 (NODE 1 (NODE 2 (LEAF 4) (LEAF 5)) (LEAF 6))
               (NODE 10 (LEAF 8) (LEAF 9))

tree2 = NODE "F" (NODE "D" (LEAF "E") (NODE "C" (LEAF "B") (LEAF "G")))
                            (NODE "G" (NODE "H" (LEAF "F") (LEAF "E")) (LEAF "A"))
```

```
> foldTree (+) tree1
50
> foldTree max tree1
10
> foldTree min tree1
1
> foldTree max tree2
"H"
> foldTree min tree2
"A"
> foldTree (++) tree2
"FDECBGGHFEA"
```

(b) createRTree - 12%

Write a recursive Haskell function `createRTree` that takes a Tree value as input and creates the equivalent Rtree value. For each NODE in the input Tree value, it should calculate the range tuple , i.e., (min,max), and it should store it in an RNODE value. The type of the `createRTree` function should be: `createRTree :: Ord t => Tree t -> RTree t`

Hint: You can use `foldTree` function to calculate the min and max values of a NODE's children.

Examples:

```
tree1 = NODE 5 (NODE 1 (NODE 2 (LEAF 4) (LEAF 5)) (LEAF 6))
               (NODE 10 (LEAF 8) (LEAF 9))

tree2 = NODE "F" (NODE "D" (LEAF "E") (NODE "C" (LEAF "B") (LEAF "G")))
                            (NODE "G" (NODE "H" (LEAF "F") (LEAF "E")) (LEAF "A"))
```

```
>createRTree tree1
RNODE 5 (1,10) (RNODE 1 (1,6) (RNODE 2 (2,5) (RLEAF 4) (RLEAF 5)) (RLEAF 6))
                (RNODE 10 (8,10) (RLEAF 8) (RLEAF 9))

>createRTree tree2
RNODE "F" ("A","H") (RNODE "D" ("B","G") (RLEAF "E") (RNODE "C" ("B","G")
(RLEAF "B") (RLEAF "G"))) (RNODE "G" ("A","H") (RNODE "H" ("E","H") (RLEAF
"F") (RLEAF "E")) (RLEAF "A"))
```

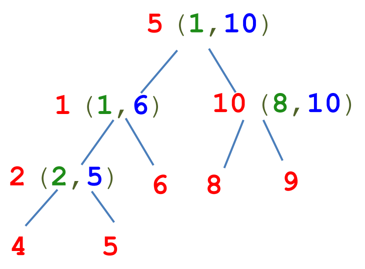
(c) fastSearch – 15%

Write a recursive Haskell function `fastSearch` that takes an RTree and a value as input and searches the tree for the given value. The search algorithm traverses the tree using "depth-first" traversal; however, it improves the search performance by eliminating (i.e., not visiting) the branches whose ranges don't overlap with the given search value. The function returns a list of tuples, where each tuple represents a node visited in the search. The first value in the tuple is a string representing the type of the visited node (either `"leaf"` or `"node"`) and the second value is the value stored in the node visited.

The type of the `fastSearch` function should be:

```
fastSearch :: Ord t => RTree t -> t -> [([Char], t)]
```

Examples:



`fastSearch` for value 8 on the left tree returns
`[("node",5),("node",1),("node",10),("leaf",8),("leaf",9)]`

Note that the search algorithm doesn't visit `RNODE 2 (2,5)`.

```
rtree1 = RNODE 5 (1,10) (RNODE 1 (1,6) (RNODE 2 (2,5) (RLEAF 4) (RLEAF 5)) (RLEAF 6))
                        (RNODE 10 (8,10) (RLEAF 8) (RLEAF 9))

rtree2 =  RNODE "F" ("A","H") (RNODE "D" ("B","G")
                                   (RLEAF "E")
                                   (RNODE "C" ("B","G") (RLEAF "B") (RLEAF "G")))
                              (RNODE "G" ("A","H")
                                   (RNODE "H" ("E","H") (RLEAF "F") (RLEAF "E"))
                                   (RLEAF "A"))
```

```
> fastSearch rtree1 6
[("node",5),("node",1),("node",2),("leaf",6),("node",10)]

> fastSearch rtree1 8
[("node",5),("node",1),("node",10),("leaf",8),("leaf",9)]

> fastSearch rtree2 "A"
[("node","F"),("node","D"),("node","G"),("node","H"),("leaf","A")]

> fastSearch rtree2 "F"
[("node","F"),("node","D"),("leaf","E"),("node","C"),("leaf","B"),("leaf","G"),("node
","G"),("node","H"),("leaf","F"),("leaf","E"),("leaf","A")]
```

5. Tree examples – 5%

Create two trees of type `Tree`. The height of both trees should be at least 4 (including the root). Test your `foldTree` and `createRTree` functions with those trees. You can use the `RTree` values returned by `createRTree` to test the `fastSearch` function. The structure of the trees you define should be different than those that are given.

Testing your functions – 5%

We will be using the `HUnit` unit testing package in CptS355. See http://hackage.haskell.org/package/HUnit for additional documentation.

The file `HW2SampleTests.hs` provides at least one sample test case comparing the actual output with the expected (correct) output for each problem. This file imports the `HW2` module (`HW2.hs` file) which will include your implementations of the given problems.

You are expected to add **at least 2 more test cases** for problems 2 (a,b) and 4 (a,b,c). You don't need to provide tests for problems 1 and 3. In your tests make sure that your test inputs cover boundary cases. Choose test input different than those provided in the assignment prompt. For problem 4 tests, you can use the trees your created in problem 5.

In `HUnit`, you can define a new test case using the `TestCase` function and the list `TestList` includes the list of all test that will be run in the test suite. So, make sure to add your new test cases to the `TestList` list. All tests in `TestList` will be run through the "`runTestTT tests`" command.

If you don't add new test cases you will be deduced at least 5 pts in this homework.

Important note about negative integer arguments:
In Haskell, the -x, where x is a number, is a special form and it is a prefix (and unary) operator negating an integer value. When you pass a negative number as argument function, you may need to enclose the negative number in parenthesis to make sure that unary (-) is applied to the integer value before it is passed to the function.
For example: `foo -5 5 [-10,-5,0,5,10]` will give a type error, but
`foo (-5) 5 [-10,-5,0,5,10]` will work