CptS355 - Assignment 2 (Haskell)
Fall 2019

Assigned: Friday, September 13, 2019

Due: Wednesday, September 25, 2019

Weight: Assignment 2 will count for 6% 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, `HW2Tests.hs`, includes at least one test case for each problem. You will edit this file and provide additional tests for each problem (at least 2 tests per problem). Please use test input different than those provided in the assignment prompt.

To submit your assignment, please upload both files (`HW2.hs` and `HW2Tests.hs`) on the Assignment2 (Haskell) DROPBOX on Blackboard (under Assignments). You may turn in your assignment up to 3 times. Only the last one submitted will be graded.

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 recursive definitions built up from the basic built-in functions. (You are not allowed to import an external 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 helper functions which are also not recursive.
- The type of your functions should match with the type specified in each problem. Otherwise you will be deducted points (around 40%).

- 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(b) 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` block). In this homework you will lose 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
- The assignment will be marked for good programming style (indentation and appropriate comments), as well as clean compilation and correct execution. Haskell comments are placed inside properly nested sets of opening/closing comment delimiters:
  ```
  {- multi line
  comment-}.
  ```
  Line comments are preceded by double dash, e.g., `-- line comment`

Problems

1. `merge2, merge2Tail, and mergeN` - 22%

(a) `merge2` - 6%

The function `merge2` takes two lists of integers, `l1` and `l2`, each already in ascending order, and returns a merged list that is also in ascending order. The resulting list should include the elements from both lists and may include duplicates.
The type of `merge2` should be `merge2 :: Ord a => [a] -> [a] -> [a].`

Examples:

```
> merge2 [2,5,6,8,9] [1,3,4,5,7,8,10]
[1,2,3,4,5,5,6,7,8,8,9,10]

> merge2 [1,2] [0,10,12]
[0,1,2,10,12]

> merge2 [1,3,3,5,5] [-1,2,4]
[-1,1,2,3,3,4,5,5]

> merge2 [1,2,3] []
[1,2,3]
```

(b) `merge2Tail` - 10%

Re-write the `merge2` function from part (a) as a tail-recursive function. Name your function `merge2Tail`.
(Hint: In your bases case(s), use `revAppend` (which we defined in class) to add the reverse of the accumulated merged list to the other list.)
The type of `merge2Tail` should be `merge2Tail :: Ord a => [a] -> [a] -> [a].`

(c) `mergeN` - 6%

Using `merge2` function defined above and the `foldl (or foldr)` function, define `mergeN` which takes a list of lists, each already in ascending order, and returns a new list containing all of the elements in sublists in ascending order. **Provide an answer using `foldl` (or `foldr`) ; without using explicit recursion.**
The type of `mergeN` should be one of the following:

```
mergeN :: (Foldable t, Ord a) => t [a] -> [a]   OR
mergeN:: Ord a => [[a]] -> [a]
```

Examples:

```
> mergeN [[1,2],[10,12],[2,5,6,8,9]]
[1,2,2,5,6,8,9,10,12]

> mergeN [[3,4],[-3,-2,-1],[1,2,5,8,9]]
[-3,-2,-1,1,2,3,4,5,8,9]
```

2. getInRange and countInRange - 18%

(a) getInRange – 6%

Define a function `getInRange` which takes two integer values, `v1` and `v2`, and a list `iL`, and returns the values in `iL` which are greater than `v1` and less than `v2` (exclusive). **Your function shouldn't need a recursion but should use a higher order function** (`map`, `foldr/foldl`, or `filter`). You may need to define additional helper function(s), which are also not recursive.

The type of the `getInRange` function should be:

```
getInRange :: Ord a => a -> a -> [a] -> [a]
```

Examples:

```
> getInRange 3 10 [1,2,3,4,5,6,7,8,9,10,11]
[4,5,6,7,8,9]
> getInRange (-5) 5 [-10,-5,0,5,10]
[0]
> getInRange (-1) 1 [-2,2,3,4,5]
[]
```

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: `getInRange -5 5 [-10,-5,0,5,10]` will give a type error, but
`getInRange (-5) 5 [-10,-5,0,5,10]` will work

(b) countInRange – 12%

Using `getInRange` function you defined in part(a) and without using explicit recursion, define a function `countInRange` which takes two integer values, `v1` and `v2`, and a nested list `iL`, and returns the total number of values in `iL` which are greater than `v1` and less than `v2` (exclusive). **Your function shouldn't need a recursion but should use higher order function** (`map`, `foldr/foldl`, or `filter`). You may need to define additional helper function(s), which are also not recursive.

The type of the `countInRange` function should be:

```
countInRange :: Ord a => a -> a -> [[a]] -> Int
```

Examples:

```
> countInRange 3 10 [[1,2,3,4],[5,6,7,8,9],[10,11]]
6
> countInRange (-5) 5 [[-10,-5,-4],[0,4,5],[],[10]]
3
> countInRange 1 5 [[1,5],[1],[5],[]]
0
```

3. addLengths and addAllLengths - 18%

(a) addLengths – 10%

Define the following Haskell datatype which represent the US customary length units :

```
data LengthUnit =  INCH  Int | FOOT  Int | YARD  Int
                   deriving (Show, Read, Eq)
```

Define a Haskell function `addLengths` that takes two `LengthUnit` values and calculates the sum of those in `INCH` s. (Note that 1 *foot* = 12 *inches* and 1 *yard* = 36 *inches*)

The type of the `addLengths` function should be:

```
addLengths :: LengthUnit -> LengthUnit -> LengthUnit
```

Examples:

```
> addLengths (FOOT 2) (INCH 5)
INCH 29
> addLengths (YARD 3) (INCH 3)
INCH 111
> addLengths (FOOT 3) (FOOT 5)
INCH 96
```

(b) **addAllLengths – 8%**

Now, define a Haskell function `addAllLengths` that takes a nested list of `LengthUnit` values and calculates the sum of those in `INCH` s. Your function shouldn't need a recursion but should use functions "`map`" and "`foldr` (or `foldl`)". You may define additional helper functions which are not recursive.

The type of the `addAllLengths` function should be one of the following:

```
addAllLengths :: Foldable t => [t LengthUnit] -> LengthUnit   OR
addAllLengths :: [[LengthUnit]] -> LengthUnit
```

(Hint: The `base` for `fold` needs to be a `LengthUnit` value.)

Examples:

```
> addAllLengths [[YARD 2, FOOT 1], [YARD 1, FOOT 2, INCH 10],[YARD 3]]
INCH 262
> addAllLengths [[FOOT 2], [FOOT 2, INCH 2],[]]
INCH 50
> addAllLengths []
INCH 0
```

4. sumTree and createSumTree – 23%

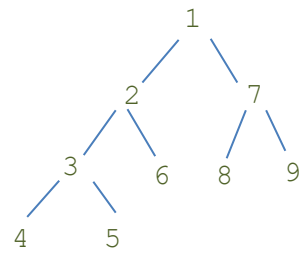
In Haskell, a polymorphic binary tree type with data both at the leaves and interior nodes might be represented as follows:

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

(a) sumTree - 8%

Write a function `sumTree` that takes a tree of type `Tree Integer` and calculates the sum of the values stored in the leaves only. The type of the `sumTree` function should be:

```
sumTree :: Num p => Tree p -> p
```



`sumTree` will return : 32

Examples:

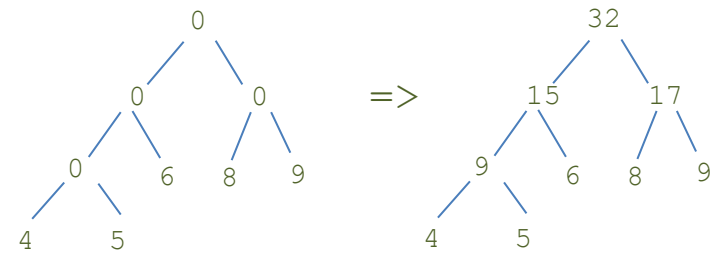
```
> t1 = NODE 1
         (NODE 2 (NODE 3 (LEAF 4) (LEAF 5)) (LEAF 6))
         (NODE 7 (LEAF 8) (LEAF 9))
> sumTree t1
32

> t2 = NODE 0
              (NODE 0 (LEAF 4) (NODE 0 (LEAF 8) (LEAF 9)))
              (NODE 0 (NODE 0 (LEAF 10) (NODE 0 (LEAF 12) (LEAF 13))) (LEAF 7))
> sumTree t2
63

> sumTree (LEAF 3)
3
```

(b) createSumTree – 15%

Write a function `createSumTree` takes an `Tree Integer` value and returns an `Tree Integer` where the interior nodes store the sum of the leaf values underneath them. See the example below.

The type of the `createSumTree` function should be

```
createSumTree :: Num a => Tree a -> Tree a
```




```
val t3 = NODE 0
          (NODE 0 (NODE 0 (LEAF 4) (LEAF 5)) (LEAF 6))
          (NODE 0 (LEAF 8) (LEAF 9))
> createSumTree t3
```

returns

```
NODE 32
   (NODE 15 (NODE 9 (LEAF 4) (LEAF 5)) (LEAF 6))
   (NODE 17 (LEAF 8) (LEAF 9))
```

5. foldListTree – 15%

A polymorphic tree type with nodes of arbitrary number of children might be represented as follows (note that the leaves store a list and interior nodes store list of "ListTree"s):

```
data ListTree a = ListLEAF [a] | ListNODE [(ListTree a)]
                  deriving (Show, Read, Eq)
```

Write a function `foldListTree` that takes a function (`f`), a base value (`base`), and a ListTree (`t`) and combines the values in the lists of the leaf notes in tree `t` by applying function `f`. (The leaves of the tree are scanned from left to right).

`foldListTree` is invoked as:

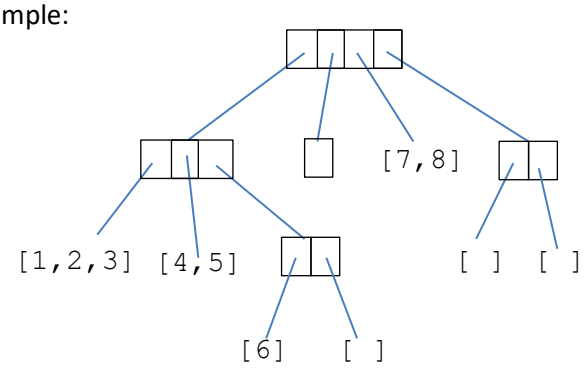
```
foldListTree f base t
```

where `f` is the combining function of type `a -> a -> a`.

The type of `foldListTree` should be:

```
foldListTree :: (a -> a -> a) -> a -> ListTree a -> a
```

Example:




```
t4 = ListNODE
    [ ListNODE [ ListLEAF [1,2,3],ListLEAF [4,5],ListNODE([ListLEAF [6], ListLEAF []]) ],
      ListNODE [],
      ListLEAF [7,8],
      ListNODE [ListLEAF [], ListLEAF []] ]

> foldListTree (+) 0 t4
36
> minValue = toInteger(minBound::Int)
> foldListTree (max) (minValue) t4
8
```

6. **Tree examples – 4%:**

Create two tree values : a "`Tree Integer`" and a "`ListTree a`" (`a` will be substituted by the type of the values stored in the tree). The height of both trees should be at least 3. Test your functions `sumTree`, `createSumTree`, `foldListTree` with those trees. The trees you define should be different than those that are given. See the example tests given in file `HW2SampleTests.hs`

Here is some additional test data for `foldListTree`.

```
> l1 = ListLEAF ["School","-","of","-","Electrical"]
> l2 = ListLEAF ["-","Engineering","-"]
> l3 = ListLEAF ["and","-","Computer","-"]
> l4 = ListLEAF ["Science"]
> l5 = ListLEAF ["-WSU"]
> n1 = ListNODE [l1,l2]
> n2 = ListNODE [n1,l3]
> t5 = ListNODE [n2,l4,l5]

> foldListTree (++) "" t5          {- ""  is empty string-}

"School-of-Electrical-Engineering-and-Computer-Science-WSU"
```

Testing your functions

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

The file `HW1SampleTests.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 each problem. Make sure that your test inputs cover all boundary cases. Choose test input different than those provided in the assignment prompt.

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% in this homework.