CptS355 - Assignment 2 (Haskell)
Fall 2020

Assigned: Tuesday, September 22, 2020

Due: Monday, October 5, 2020

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, `HW2SampleTests.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. 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 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` - 20%

(a) `merge2` - 5%

The function `merge2` takes two lists, `l1` and l2, and returns a merged list where the elements from l1 and l2 appear interchangeably. The resulting list should include the leftovers from the longer list and it may include duplicates.
The type of `merge2` should be `merge2 :: [a] -> [a] -> [a].`

Examples:

```
> merge2 [3,2,1,6,5,4] [1,2,3]
[3,1,2,2,1,3,6,5,4]
> merge2 "Ct 5" "pS35"
"CptS 355"
> merge2 [(1,2),(3,4)] [(5,6),(7,8),(9,10)]
[(1,2),(5,6),(3,4),(7,8),(9,10)]
> 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`.
The type of `merge2Tail` should be `merge2Tail :: [a] -> [a] -> [a].`

You can use `reverse` or `revAppend` in your solution. We defined `revAppend` in class.

(c) `mergeN` - 5%

Using `merge2` function defined above and the `foldl` function, define `mergeN` which takes a list of lists and returns a new list containing all the elements in sublists. The sublists should be merged left to right, i.e., first two lists should be merged first and the merged list should further be merged with the third list, etc. **Provide an answer using `foldl`; without using explicit recursion.**
The type of `mergeN` should be: `mergeN:: [[a]] -> [a]`

Examples:

```
> mergeN ["ABCDEF","abcd","123456789","+=?$"]
"A+1=a?2$B3b4C5c6D7d8E9F"
> mergeN [[3,4],[-3,-2,-1],[1,2,5,8,9],[10,20,30]]
[3,10,1,20,-3,30,2,4,5,-2,8,-1,9]
> mergeN [[],[],[1,2,3]]
[1,2,3]
```

2. `removeDuplicates, count, and histogram - 25%`

(a) `removeDuplicates – 10%`

Define a function `removeDuplicates` which takes a list as input and it eliminates the duplicate values from the list. The unique elements in the output list may appear in arbitrary order. **Your function shouldn't need a recursion but should use a higher order function** (`map`, `foldr/foldl`, or `filter`). Your helper functions <u>should not be recursive</u> as well, but they can use higher order functions. You may use the `elem` function in your implementation.

The type of the `removeDuplicates` function should be:

```
removeDuplicates:: Eq a => [a] -> [a]
```

Examples:

```
> removeDuplicates [5,4,3,2,1,1,2,3,4,5,6,7]
[1,2,3,4,5,6,7]
> removeDuplicates "CptS322 - CptS322 -  CptS 321"
"-CptS 321"
> removeDuplicates [[1,2],[1],[],[3],[1],[]]
[[1,2],[3],[1],[]]
```

(b) count – 5%

Define a function `count` which takes a value and a list as input and it count the number of occurrences of the value in the input list. **Your function should not need a recursion but should use a higher order function** (`map`, `foldr/foldl`, or `filter`). Your helper functions should not be recursive as well, but they can use higher order functions. You may use the `length` function in your implementation.

The type of the `count` function should be: `count :: Eq a => a -> [a] -> Int`

Examples:

```
> count [] [[],[1],[1,2],[]]
2
> count (-5) [1,2,3,4,5,6,7]
0
> count 'i' "incomprehensibilities"
5
```

(c) histogram – 10%

The function `histogram` creates a histogram for a given list. The histogram will be a list of tuples (pairs) where the first element in each tuple is an item from the input list and the second element is the number of occurrences of that item in the list. **Your function shouldn't need a recursion but should use a higher order function** (`map`, `foldr/foldl`, or `filter`). Your helper functions should not be recursive as well, but they can use higher order functions. You may use the `count` and `removeDuplicates` functions you defined in parts (a) and (b).

The order of the tuples in the histogram can be arbitrary. The type of the function should be:

```
histogram :: Eq a => [a] -> [(a, Int)]
```

Examples:

```
> histogram [[],[1],[1,2],[1],[],[]]
[([1,2],1),([1],2),([],3)]
> histogram "macadamia"
[('c',1),('d',1),('m',2),('i',1),('a',4)]
> histogram (show 122333444455555)
[('1',1),('2',2),('3',3),('4',4),('5',5)]
```

3. **concatAll, concat2Either, and concat2Str – 19%**

(a) **concatAll - 4%**

Function concatAll is given a nested list of strings and it returns the concatenation of all strings in all sublists of the input list. Your function should not need a recursion but should use functions “map” and “foldr”. You may define additional helper functions <u>which are not recursive.</u>
The type of the concatAll function should be:

```
concatAll ::  [[String]] -> String
```

Examples:

```
> concatAll [["enrolled"," ","in"," "],["CptS","-","355"],[" ","and","
"],["CptS","-","322"]]
"enrolled in CptS-355 and CptS-322"
> concatAll [[],[]]
""
```

(b) **concat2Either - 9%**

Define the following Haskell datatype:

```
data AnEither  = AString String | AnInt Int
                deriving (Show, Read, Eq)
```

Define a Haskell function concat2Either that takes a nested list of AnEither values and it returns an AString, which is the concatenation of all values in all sublists of the input list. The parameter of the AnInt values should be converted to string and included in the concatenated string. You may use the **show** function to convert an integer value to a string.

Your concat2Either function shouldn’t need a recursion but should use functions “map” and “foldr”. You may define additional helper functions <u>which are not recursive.</u> The type of the concat2Either function should be:

```
concat2Either:: [[AnEither]] -> AnEither
```

(Note: To implement concat2Either, change your concatAll function and your helper function in order to handle AnEither values instead of strings.)

Examples:

```
> concat2Either [[AString "enrolled", AString " ", AString "in", AString " "],[AString
"CptS", AString "-", AnInt 355], [AString " ", AString "and", AString " "], [AString
"CptS", AString "-", AnInt 322]]
AString "enrolled in CptS-355 and CptS-322"
> concat2Either [[AString "", AnInt 0],[]]
AString "0"
> concat2Either []
AString ""
```

(c) **concat2Str - 6%**

Re-define your concat2Either function so that it returns a concatenated string value instead of an AString value. Similar to concat2Either, the parameter of the AnInt values should be converted to string and included in the concatenated string.

Your concat2Str function shouldn't need a recursion but should use functions "map" and "foldr". You may define additional helper functions which are not recursive. The type of the concat2Str function should be:

```
concat2Str:: [[AnEither]] -> String
```

(Note: To implement concat2Str, change your concat2Either function and your helper function in order to return a string value instead of an AnEither value.)

```
> concat2Str [[AString "enrolled", AString " ", AString "in", AString " "],[AString
"CptS", AString "-", AnInt 355], [AString " ", AString "and", AString " "], [AString
"CptS", AString "-", AnInt 322]]
"enrolled in CptS-355 and CptS-322"
> concat2Str [[AString "", AnInt 0],[]]
"0"
> concat2Str []
""
```

4. evaluateTree, printInfix, createRTree – 32%

Consider the following Haskell type Op that defines the major arithmetic operations on integers.

```
data Op = Add | Sub | Mul | Pow
          deriving (Show, Read, Eq)
```

The following function "evaluate" takes an Op value as argument and evaluates the operation on the integer arguments x and y.

```
evaluate:: Op -> Int -> Int -> Int
evaluate Add x y =  x+y
evaluate Sub   x y =  x-y
evaluate Mul x y =  x*y
evaluate Pow x y = x^y
```

Now, we define an expression tree as a Haskell polymorphic binary tree type with data at the leaves and Op operators at the interior nodes:

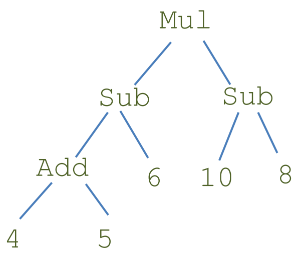
```
data ExprTree a = ELEAF a | ENODE Op (ExprTree a) (ExprTree a)
                  deriving (Show, Read, Eq)
```

(a) evaluateTree - 10%

Write a function evaluateTree that takes a tree of type (ExprTree Int) as input and evaluates the tree from bottom-up.

The type of the evaluateTree function should be evaluateTree :: ExprTree Int -> Int

For example:



evaluateTree on the left tree returns 6.

```
>evaluateTree (ENODE Mul (ENODE Sub (ENODE Add (ELEAF 4) (ELEAF 5)) (ELEAF 6))
                         (ENODE Sub (ELEAF 10) (ELEAF 8)))
6
>evaluateTree (ENODE Add (ELEAF 10)
                        (ENODE Sub (ELEAF 50) (ENODE Mul (ELEAF 3) (ELEAF 10))))
30
>evaluateTree (ELEAF 4)
4
```

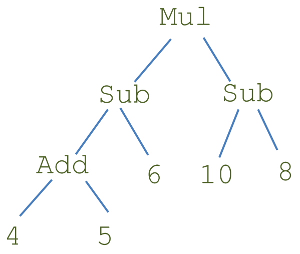
(b) **printInfix - 10%**

Write a function `printInfix` that takes a tree of type `(ExprTree a)` as input and prints the operands in the interior nodes and the values in the leaf nodes in "in-fix" order to a string. The expressions lower in the tree are enclosed in parenthesis.

The type of the `printInfix` function should be:

```
printInfix:: Show a => ExprTree a -> String
```

For example:



`printInfix` on the left tree returns :
`"(((4 `Add` 5) `Sub` 6) `Mul` (10 `Sub` 8))"`

```
>printInfix (ENODE Mul (ENODE Sub (ENODE Add (ELEAF 4) (ELEAF 5)) (ELEAF 6))
                      (ENODE Sub (ELEAF 10) (ELEAF 8)))
"(((4 `Add` 5) `Sub` 6) `Mul` (10 `Sub` 8))"

>printInfix (ENODE Add (ELEAF 10)
                        (ENODE Sub (ELEAF 50) (ENODE Mul (ELEAF 3) (ELEAF 10))))
"(10 `Add` (50 `Sub` (3 `Mul` 10)))"

>printInfix (ELEAF 4)
"4"
```

(c) **createRTree – 12%**

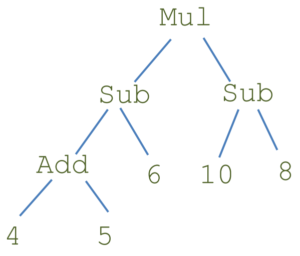
Consider the following Haskell tree type.

```
data ResultTree a  = RLEAF a | RNODE a (ResultTree a) (ResultTree a)
                     deriving (Show, Read, Eq)
```

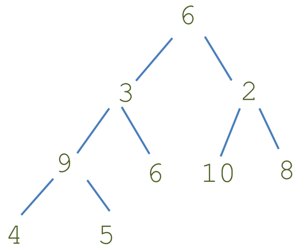
Write a function createRTree that takes a tree of type (ExprTree Int) as input and creates a tree of type (ResultTree Int). createRTree recursively evaluates each subtree in the input tree and store the evaluated values in the corresponding nodes in the output ResultTree.
The type of the createRTree function should be:

```
createRTree :: ExprTree Int -> ResultTree Int
```

For example:



createRTree on the left tree returns :



```
>createRTree (ENODE Mul (ENODE Sub (ENODE Add (ELEAF 4) (ELEAF 5)) (ELEAF 6))
             (ENODE Sub (ELEAF 10) (ELEAF 8)))
RNODE 6 (RNODE 3 (RNODE 9 (RLEAF 4) (RLEAF 5)) (RLEAF 6)) (RNODE 2 (RLEAF 10) (RLEAF 8))

>createRTree (ENODE Add (ELEAF 10) (ENODE Sub (ELEAF 50) (ENODE Mul (ELEAF 3) (ELEAF 10))))
RNODE 30 (RLEAF 10) (RNODE 20 (RLEAF 50) (RNODE 30 (RLEAF 3) (RLEAF 10)))

>createRTree (ELEAF 4)
RLEAF 4
```

5. Tree examples - 4%

Create two trees of type ExprTree. The height of both trees should be at least 4 (including the root). Test your functions evaluateTree, printInfix, and createRTree with those trees. The trees you define should be different than those that are given.

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 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 3 (a,b,c) and 4 (abc). You don't need to provide tests for problems 1 and 2. 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% 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