CptS355 - Lab 2 (Haskell) – Fall 2023

Weight: Lab2 will count for 1.5% of your course grade.

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/platforms/.

Turning in your lab submission

This lab involves 7 programming problems and your solution will consist of a sequence of function definitions for the given problems. You should implement as many of the problems as you can. As explained in the "Grading" section, 4 correct solutions are sufficient to earn full points in the lab. Note that some problems have sub-parts; you should complete all parts to get credit for those problems.

You will write all your functions in the attached `Lab2.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.

Attached file, `Lab2SampleTests.zip`, includes HUnit unit tests for each problem. Make sure to test your code before you submit. Please refer to the "Testing your functions" section at the end of this document.

To submit your assignment, please upload the file `Lab2.hs` on the Lab2 (Haskell) DROPBOX on Canvas (under Assignments). You may turn in your lab submission up to 3 times. Only the last one submitted will be graded.

Grading

The lab assignments will be auto-graded using automated tests. Your lab grade will be calculated as follows:

- When you submit the lab, you will automatically earn 60 points.
- For every correct solution that passes all test cases, you will earn 10 points. If your function fails even a single test case, you will not be able to earn points for that question.
- The lab includes 7 programming problems. You can implement as many of the lab problems as you want. However, 4 correct answers are sufficient to earn full points (100) in the lab. No extra credit will be given for additional solutions.
- However, submitting solutions for more than 4 problems will improve your chance to obtain 100% score (in case one of the test cases fail for the other problems).
- The Lab Zoom sessions are optional. You can work on the lab problems yourself or with your friends and submit your solutions on Canvas.

Important rules

- In your solutions, you can use any of the built-in Haskell functions included in the lecture notes. However, 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.
- Make sure that the type of your functions match with the type specified in each problem.

- 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.
- 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). 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.

Lab Problems

1. `merge2, merge2Tail, and mergeN`

(a) `merge2`

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.

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`

Re-write the `merge2` function from part (a) as a tail-recursive function. Name your function `merge2Tail`.

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

(c) `mergeN`

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 compatible with : `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. count and histogram

(a) count

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.

Examples:

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

(b) histogram

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` function you defined in part (a) and `eliminateDuplicates` function you defined in HW1.

The order of the tuples in the histogram can be arbitrary.

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

(a) **concatAll**

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 which are not recursive.

Examples:

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

(b) **concat2Either**

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 which are not recursive.

(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 ""
```

4. concat2Str

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.

(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 []
""
```

evaluateTree, printInfix, createRTree

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
```

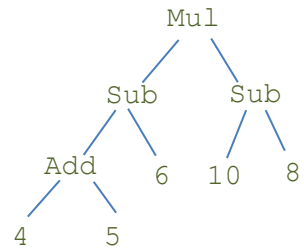
Now, we define an expression tree as a Haskell polymorphic binary tree type with data at the leaves and `Op` operators at the interior nodes:

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

5. evaluateTree

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

For example:



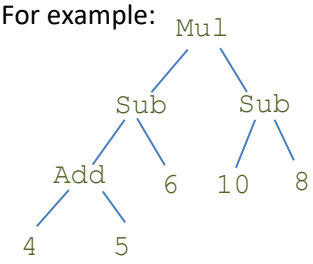
`evaluateTree` on the left tree returns 6.

```
> 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
```

6. 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.

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"
```

7. createRTree

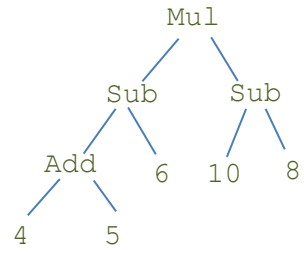
Consider the following Haskell tree type.

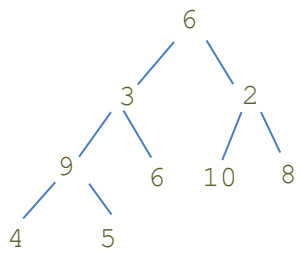
```
data ResultTree a  = RLEAF a | RNODE a (ResultTree a) (ResultTree a)
                     deriving (Show, Read, Eq)
```

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`.

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
```

Testing your functions

The `Lab2SampleTests.zip` file includes 7 `.hs` files where each one includes the HUnit tests for a different lab problem. The tests compare the actual output to the expected (correct) output and raise an exception if they don't match. The test files import the `Lab2` module (`Lab2.hs` file) which will include your implementations of the lab problems.

You will write your solutions to `Lab2.hs` file. To test your solution for the first lab problem run the following commands on the command line window (i.e., terminal):

```
$ ghci
$ :l Q1_tests.hs
*Q1_tests> main
```

Repeat the above for other lab problems by changing the test file name, i.e. , `Q2_tests.hs`, `Q3_tests.hs`, etc.