CptS355 - Assignment 2 (Haskell)
Fall 2022

Assigned: Wednesday, September 28, 2022

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.

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.

To submit your assignment, please upload `HW2.hs` file to 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(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` 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. `insert, insert_tail`

(a) `insert` – 3%

Write a function `insert` that takes an integer n, a value `item`, and a list `iL` and inserts the `item` at index n in the list `iL`. n is a 1-based index, i.e., item should be inserted after n^{th} element in the list. If `n` is greater than the length of the input list, the `item` will not be inserted. If n is `0`, `item` will be inserted to the beginning of the list. (You may assume that n >=0.)
This is the `insert` function you implemented in Lab1.

Examples:

```
> insert 3 100 [1,2,3,4,5,6,7,8]
[1,2,3,100,4,5,6,7,8]

> insert 8 100 [1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7,8,100]

> insert 9 100 [1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7,8]

> insert 3 100 []
[]
```

(b) `insert_tail` – 10%

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

You may use the same test cases provided above to test your function.

2. game_scores and wins_by_year

Assume the "wsu_games" data we used in HW1.

```
wsu_games_test = [
    (2019, [("NMSU",(58,7)), ("UNCO",(59,17)), ("HOU",(31,24)), ("UCLA",(63,67)),
            ("UTAH",(13,38)), ("ASU",(34,38)), ("COLO",(41,10)), ("ORE",(35,37)),
            ("CAL",(20,33)), ("STAN",(49,22)), ("ORST",(54,53)), ("WASH",(13,31)),
            ("AFA",(21,31))]),
    (2020, [("ORST",(38,28)), ("ORE",(29,43)), ("USC",(13,38)), ("UTAH",(28,45))]),
    (2021, [("USU",(23,26)), ("PORT ST.",(44,24)), ("USC",(14,45)), ("UTAH",(13,24)),
            ("CAL",(21,6)), ("ORST",(31,24)), ("STAN",(34,31)), ("BYU",(19,21)),
            ("ASU",(34,21)), ("ORE",(24,38)), ("ARIZ",(44,18)), ("WASH",(40,13)),
            ("CMU",(21,24))] )
   ]
```

(a) game_scores – 12%

Rewrite the game_scores function in HW1 **using higher order functions (map, foldr/foldl, or filter) and without using recursion.** Your helper functions should not be recursive as well, but they can use higher order functions.

Remember that game_scores takes the game list (similar to wsu_games above) and an opponent team name (e.g., "USC") as input and returns the list of the game scores that WSU played against the given opponent team.

Examples:

```
> game_scores wsu_games "USC"
[(13,38),(14,45)]
> game_scores wsu_games "ORST"
[(54,53),(38,28),(31,24)]
> game_scores wsu_games "YALE"
[ ]
```

(b) wins_by_year – 12%

Rewrite the wins_by_year function in HW1 **using higher order functions (map, foldr/foldl, or filter) and without using recursion.** Your helper functions should not be recursive as well, but they can use higher order functions.

Remember that wins_by_year takes the WSU game data as input, and it returns a list of tuples where each tuple includes the year and the number wins (of WSU team) in that year.

Example:

```
> wins_by_year wsu_games
[(2019,6),(2020,1),(2021,7)]
```

3. sum_nested_int, sum_nested_item, and sum_my_nested

Consider the following Haskell datatype:

```
data NestedList  = Item Int
                 | Array [Int]
                 deriving (Show, Read, Eq)
```

Note that, Array's parameter is an [Int] value.

(a) sum_nested_int - 8%

Function sum_nested_int takes a list of NestedItem values as input and it sums all the parameter values of Item and Array elements in the input list. **It returns the overall sum as an Int value.**
Your function shouldn't need a recursion but should use higher order function(s) **map, foldr/foldl, or filter.** You may define additional helper functions which are not recursive.
For example,
sum_nested_int [Item 10, Item 5, Array [7,3,12,11], Item 5, Array [9], Array []]
returns 10 + 5 + 7 + 3 + 12 + 11 + 5 + 9 + 0 = 62

Examples:

```
> sum_nested_int [Item 10, Item 5, Array [7, 3, 12, 11], Item 5, Array [9], Array []]
62

> sum_nested_int [Array [-3,-5,-6],Array [10, 11],Item (-5),Array [7],Item 1,Item 0]
10

> sum_nested_int [ ]
0
```

(b) sum_nested_item - 5%

Function sum_nested_item takes a list of NestedList values as input and it sums all the parameter values in Item and Array elements in the input list. **It returns the overall sum as an Item value.**
Your function shouldn't need a recursion but should use higher order function(s) **map, foldr/foldl, or filter.** You may define additional helper functions which are not recursive. Also, your function should not use sum_nested_int that you defined in part(a).
For example,
sum_nested_item [Item 10, Item 5, Array [7,3,12,11], Item 5, Array [9], Array []]
returns (Item 62).

Examples:

```
> sum_nested_item [Item 10, Item 5, Array [7,3,12,11], Item 5, Array [9], Array []]
Item 62

> sum_nested_item [Array [-3,-5,-6],Array [10, 11],Item (-5),Array [7],Item 1,Item 0]
Item 10

> sum_nested_item [ ]
Item 0
```

(c) **sum_my_nested - 12%**

Now assume we revise the `NestedItem` datatype and define the following:

```
data MyNested  = MyItem Int
               | MyArray [MyNested]
               deriving (Show, Read, Eq)
```

Note that, `MyArray`'s parameter is a `[MyNested]` value.

Define a Haskell function `sum_my_nested` that takes a list of `MyNested` values and it returns the sum of all parameter values of `MyItem` and `MyArray` values. Since the parameter of `MyArray` is a list of `MyNested` values, it should recursively add all parameter values in that list. **The function should return the overall sum as an `Int` value.**
Your function can be recursive and may use higher order functions.

Note: A possible solution will use foldr and a helper function which is mutually recursive with sum_my_nested, i.e., sum_my_nested calls helper and helper calls sum_my_nested.

Examples:

```
> sum_my_nested [MyItem 10, MyItem 5, MyArray
                 [MyItem 7, MyArray [MyItem 3, MyItem 12], MyItem 11],
                 MyItem 5, MyArray [MyItem 9], MyArray []]
62
> sum_my_nested [MyArray [MyItem (-2), MyArray [MyItem 3, MyArray
                  [MyItem 12, MyItem (-5), MyArray [MyItem (-8), MyArray[MyItem 3]],
                   MyArray []], MyItem 11], MyItem 5], MyArray [MyItem 9], MyArray[]]
28

> sum_my_nested [ ]
Item 0
```

4. tree_height, create_htree, tree_paths

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)
```

Assume we modify the above datatype and have each interior node (i.e., Node value) store the height of the sub-tree rooted at that node. We call this new datatype HTree, defined as follows:

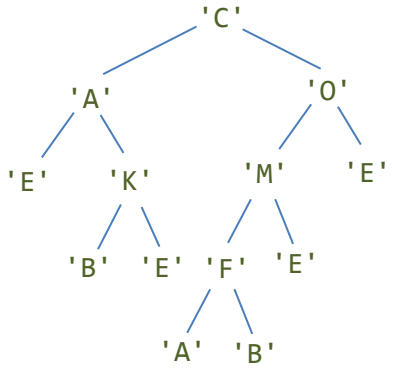
```
data HTree a = HLeaf a | HNode a Int (HTree a) (HTree a)
               deriving (Show, Read, Eq)
```

The Int value in the HNode is the height of the subtree.

(a) tree_height - 8%

Write a function tree_height that takes a tree of type (Tree a) and calculates the height of the tree, i.e., returns the length of the longest path from root to a Leaf node.

For example:



tree_height for the given tree will return 5.

Examples:

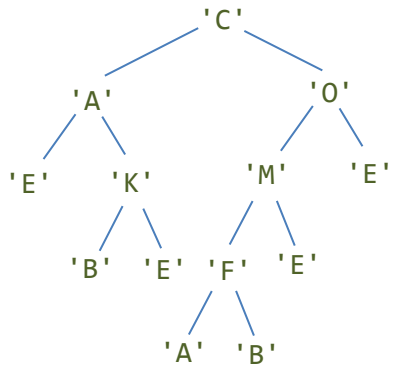
```
> tree_height (Node 'C' (Node 'A' (Leaf 'E') (Node 'K' (Leaf 'B') (Leaf 'E'))) (Node
'O' (Node 'M' (Node 'F' (Leaf 'A') (Leaf 'B')) (Leaf 'E')) (Leaf 'E')))
5
```

```
> tree_height (Node 3 (Node 10 (Node 1 (Leaf 4) (Leaf 5)) (Leaf 6)) (Node 5 (Leaf 8)
(Leaf 4)))
4
```

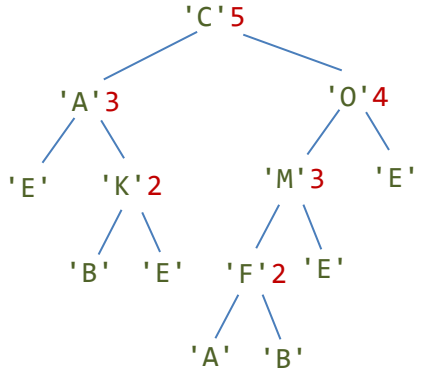
(b) create_htree – 10%

Write a function `create_htree` that takes a tree of type (`Tree a`) and converts the input tree to a `HTree`. The function should recursively traverse the input tree and create a `HLeaf` value for each `Leaf`, and a `HNode` value for each `Node`. It should calculate the height of the subtree rooted at that node and store that value in the created `HNode`.

For example:



`create_htree` for the left tree will return the tree of the right.
The red values are the heights of the nodes.



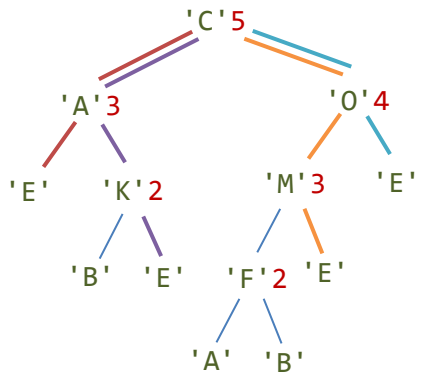
Examples:

```
t1 = Node 3 (Node 10 (Node 1 (Leaf 4) (Leaf 5)) (Leaf 6))
            (Node 5 (Leaf 8) (Leaf 4))

t2 = Node 'C' (Node 'A' (Leaf 'E') (Node 'K' (Leaf 'B') (Leaf 'E')))
              (Node 'O' (Node 'M' (Node 'F' (Leaf 'A') (Leaf 'B')) (Leaf 'E')) (Leaf 'E'))
```

```
> create_htree t2
HNode 'C' 5 (HNode 'A' 3 (HLeaf 'E') (HNode 'K' 2 (HLeaf 'B') (HLeaf 'E'))) (HNode 'O' 4
(HNode 'M' 3 (HNode 'F' 2 (HLeaf 'A') (HLeaf 'B')) (HLeaf 'E')) (HLeaf 'E'))

> tree_height t1
HNode 3 4 (HNode 10 3 (HNode 1 2 (HLeaf 4) (HLeaf 5)) (HLeaf 6)) (HNode 5 2 (HLeaf 8)
(HLeaf 4))
```

(c) find_paths – 13%

Write a function `find_paths` that takes a tree `t` of type (`HTree a`) and a value `v` of type `a`, and it finds all paths from the root to the leaf nodes having the value `v`. `find_paths` returns a nested list where each sublist corresponds to a path from root to a (`Leaf v`) node.
For example:



`find_paths` for the left tree with target value 'E' will return:
`["CAE","CAKE","COME","COE"]`

The paths to target leaf nodes are marked using colored lines.

```
ht1 = HNode 3 4 (HNode 10 3 (HNode 1 2 (HLeaf 4) (HLeaf 5)) (HLeaf 6))
                (HNode 5 2 (HLeaf 8) (HLeaf 4))
ht2 = HNode 'C' 5 (HNode 'A' 3 (HLeaf 'E') (HNode 'K' 2 (HLeaf 'B') (HLeaf 'E')))
                  (HNode 'O' 4 (HNode 'M' 3 (HNode 'F' 2 (HLeaf 'A') (HLeaf 'B'))
                  (HLeaf 'E')) (HLeaf 'E'))
```

```
>find_paths ht2 'E'
["CAE","CAKE","COME","COE"]
```

```
>find_paths ht2 'A'
["COMFA"]
```

```
>find_paths ht1 4
[[[3,10,1,4],[3,5,4]]
```

5. Tree examples – 4%

Create two trees of type Tree. The height of both trees should be at least 4. Test your functions `tree_height`, `create_htree` with those trees. You can further test your `create_htree` function using the HTree values returned from `create_htree`.
The trees you define should be different than those that are given. Make sure to change the shape of the trees; just changing the values will not make your trees different.

Include your example trees at the end of your HW2.hs file - under the comment INCLUDE YOUR TREE EXAMPLES HERE .
In this assignment you won't submit any test files.

Assignment rules – 3%

Make sure that your assignment submission complies with the following. :

- The module name of your `HW2.hs` files should be `HW2`. Please don't change the module name in your submission file.
- The function names in your solutions should match the names in the assignment prompt. Running the given tests will help you identify typos in function names.
- Make sure to remove all test data from the `HW2.hs` file, e.g. , tree examples provided in the assignment prompt , the test files and the '`wsu_games`' list for Problem-2.
- Make sure to define your helper functions inside a let..in or where block.
- Make sure that your solutions meet the specified requirements:
    - Your solution for 1(a) should be tail-recursive.
    - Your solutions for 2(a), 2(b), 3(a), and 3(b) shouldn't need a recursion but should use higher order function(s) **`map`, `foldr`/`foldl`, or `filter`.**
    - Your solutions for 3(b) should not use `sum_nested_item` function you defined in 3(a).

Testing your functions

The `HW2SampleTests.zip` file includes 4 `.hs` files where each one includes the HUnit tests for a different HW 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 `HW2` module (`HW2.hs` file) which will include your implementations of the HW problems.

You will write your solutions to `HW2.hs` file. To test your solution for each HW problem run the following commands on the command line window (i.e., terminal):

```
$ ghci
$ :l P1_HW2tests.hs
P1_HW2tests> run
```

Repeat the above for other HW problems by changing the test file name, i.e. , `P2_HW2tests.hs, P3_HW2tests.hs, etc.`

You don't need to submit any tests for this assignment. However, you should still test your solutions using additional input. Make sure to test your code for boundary cases.

Note : For problem 4(abc), you can use the trees you created in problem-5. See above.

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