# CptS355 - Assignment 2 (Haskell)
# Spring 2023

**Assigned:** Tuesday, February 14, 2023

**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. `remove_every, remove_every_tail_tail`

### (a) `remove_every` – 7%

Consider the following `remove_every` function. The function takes an integer `n` and a list `lst` and removes the item after every nth element in the list `lst`.

If n is greater than the length of the input list, no elements will be removed. If n is `0`, all elements in the list will be deleted. You may assume that n `>=0`.

The below `remove_every` function will give some type errors/run time exceptions when compiled/run. Identify the problems and fix them. Explain the problems you identified/fixed in a comment.

```
remove_every n []  = []
remove_every n lst = remove_helper n lst n
  where
     remove_helper 0 (x:xs) k = (remove_helper k xs )
     remove_helper n (x:xs) k = x:(remove_helper (n-1) xs)
```

Examples:

```
> remove_every 3 "123a456b789c"
"123456789"

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

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

### (b) `remove_every_tail` – 10%

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

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

## 2. `get_outof_range` and `count_outof_range`

### (a) `get_outof_range` – 6%

Define a function `get_outof_range` which takes two values, `v1` and `v2`, and a list "`xs`", and returns the values in `xs` which are less than `v1` and greater 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.

Examples:

```
> get_outof_range (-5) 5 [10,5,0,1,2,-5,-10]
[10,-10]

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

> get_outof_range 'A' 'z' "CptS-355"
"-355"
```

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

### (b) `count_outof_range` – 10%

Using `get_outof_range` function you defined in part(a) and without using explicit recursion, define a function `count_outof_range` which takes two integer values, `v1` and `v2`, and a nested list "`xs`", and returns the total number of values in `xs` which are less than `v1` and greater 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.

Examples:

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

> count_outof_range 'A' 'z' ["Cpt S","-","355",":","HW2"]
7

> count_outof_range 1 1 [[4,1],[2,-1,3,-4],[8,0,1,5,9,4]]
10
```

## 3. find_routes – 10%

Assume the "routes" data we used in HW1.

```
routes = [
    ("Lentil", ["Chinook", "Orchard", "Valley", "Emerald","Providence", "Stadium",
      "Main", "Arbor", "Sunnyside", "Fountain", "Crestview", "Wheatland", "Walmart",
      "Bishop", "Derby", "Dilke"]),
    ("Wheat",["Chinook", "Orchard", "Valley", "Maple","Aspen", "TerreView", "Clay",
     "Dismores", "Martin", "Bishop", "Walmart", "PorchLight", "Campus"]),
    ("Silver",["TransferStation", "PorchLight", "Stadium", "Bishop","Walmart",
     "Outlet", "RockeyWay","Main"]),
    ("Blue",["TransferStation", "State", "Larry", "TerreView","Grand", "TacoBell",
     "Chinook", "Library"]),
    ("Gray",["TransferStation", "Wawawai", "Main", "Sunnyside","Crestview",
     "CityHall", "Stadium", "Colorado"]),
    ("Coffee",["TransferStation", "Grand", "Main", "Visitor","Stadium", "Spark",
     "CUB"])
   ]
```

Rewrite the `find_routes` 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 `find_routes` takes the list of bus routes and a stop name, and returns the list of the bus routes which stop at the given bus stop. You can make use of `elem` function in your solution. The order of the elements in the output can be arbitrary.

Examples:

```
> find_routes "Walmart" routes
["Lentil","Wheat","Silver"]

> find_routes "Rosauers" routes
[]

> find_routes "Main" routes
["Lentil","Silver","Gray","Coffee"]
```

## 4. add_lengths and add_nested_lengths

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

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

### (a) add_lengths - 6%

Define a Haskell function `add_lengths` 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*)

Examples:

```
> add_lengths  (INCH (-5)) (INCH 10)
(INCH 5)
```

```
>add_lengths  (INCH 5) (FOOT 10)
(INCH 125)

>(add_lengths  (YARD 1) (INCH 10)
(INCH 46)
```

### (b) add_nested_lengths - 10%

Define a Haskell function `add_nested_lengths` that takes a <u>nested</u> 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 <u>which are not recursive.</u>
(Hint: The `base` for `fold` needs to be a `LengthUnit` value. )

Examples:

```
>add_nested_lengths [[INCH (-5),INCH 10], [YARD (-1), YARD 2,FOOT 2],
                      [INCH 5],[]]
(INCH 70)

>add_nested_lengths [[INCH 5,FOOT 10],[FOOT (-10),YARD 5],[YARD (-5),
                      INCH (-5)]]
(INCH 0)

>add_nested_lengths [[YARD 2, FOOT 1], [YARD 1, FOOT 2, INCH 10],[YARD 3]]
(INCH 262)
```

## 5. sum_tree and create_sumtree

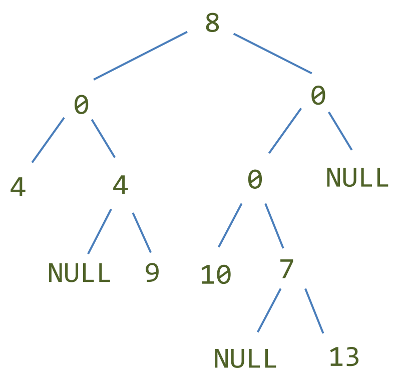
In Haskell, a polymorphic binary tree type with data both at the leaves and interior nodes might be represented as follows:

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

NULL value represent a missing child for a NODE.

### (a) sum_tree - 8%

Write a function `sum_tree` that takes a tree of type `Tree` and calculates the sum of the values stored <u>in both the leaves and interior nodes</u>.
For example:



sum_tree for the given tree will return 55 .

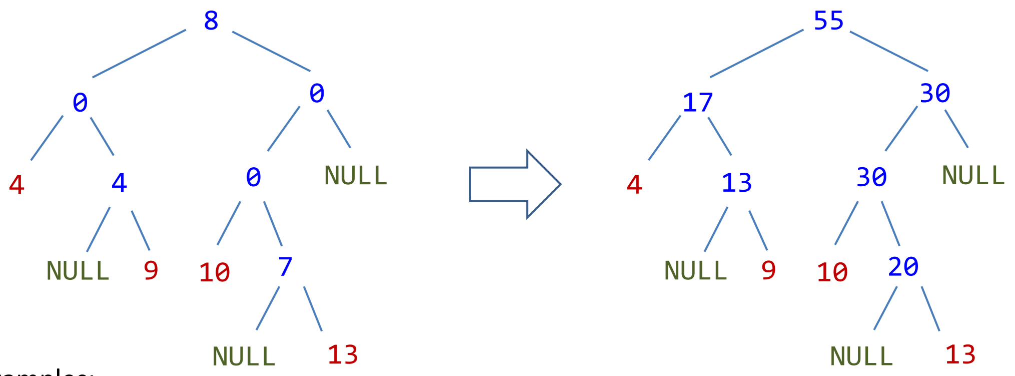
Examples:

```
> sum_tree (NODE 8 (NODE 0 (LEAF 4) (NODE 4 NULL (LEAF 9)))
                   (NODE 0 (NODE 0 (LEAF 10) (NODE 7 NULL (LEAF 13)))  NULL))
55

> sum_tree (NODE 0 (NODE 10 (LEAF 4)  NULL) (NODE (-10) (NODE (-3) NULL  (NODE 10
NULL (LEAF (-3)))) (LEAF (-4))))
4
```

**(b) create_sumtree – 10%**

Write a function `create_sumtree` takes a `Tree` value and returns a `Tree` where the interior nodes store the sum of the *leaf* and *node* values underneath them (including NODE's own values). See the example below.



Examples:

```
> sum_tree (NODE 8 (NODE 0 (LEAF 4) (NODE 4 NULL (LEAF 9)))
                   (NODE 0 (NODE 0 (LEAF 10) (NODE 7 NULL (LEAF 13)))  NULL))

NODE 55 (NODE 17 (LEAF 4) (NODE 13 NULL (LEAF 9))) (NODE 30 (NODE 30 (LEAF
10) (NODE 20 NULL (LEAF 13))) NULL)

> sum_tree (NODE 0 (NODE 10 (LEAF 4)  NULL) (NODE (-10) (NODE (-3) NULL  (NODE 10
NULL (LEAF (-3)))) (LEAF (-4))))

NODE 4 (NODE 14 (LEAF 4)  NULL) (NODE (-10) (NODE 4 NULL  (NODE 7 NULL (LEAF (-3))))
(LEAF (-4)))
```

**6. list_tree – 16%**

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):

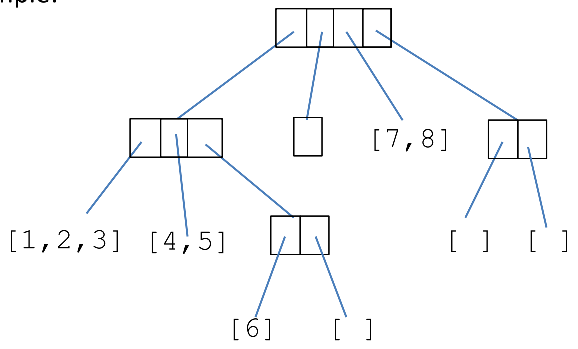
```
data ListTree a = LEAFs [a] | NODEs [(ListTree a)]
                  deriving (Show, Read, Eq)
```

Write a function `list_tree` 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). The combined values from all leaves are further combined with function `f`.

`list_tree` is invoked as:

```
        list_tree f base t
```

Example:



Examples:

```
>list_tree (+) 0
(NODEs [ NODEs [ LEAFs [1,2,3],LEAFs [4,5], NODEs [LEAFs [6], LEAFs []] ],
         NODEs [],
         LEAFs [7,8],
         NODEs [LEAFs [], LEAFs []]
       ])
36
```

```
>list_tree max 0
(NODEs [ NODEs [ LEAFs [1,2,3],LEAFs [4,5], NODEs [LEAFs [6], LEAFs []] ],
         NODEs [],
         LEAFs [7,8],
         NODEs [LEAFs [], LEAFs []]
       ])
8
```

`max` is the built-in function that returns max of two values.

## 5. Tree examples – 4%

Create two trees of type Tree. The height of both trees should be at least 4. Test your functions `sum_tree`, `create_sumtree` with those trees.

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 '`routes`' list for Problem-3.
- 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(b) should be tail-recursive.
    - Your solutions for 2(a), 2(b), 3, and 4(b) shouldn't need a recursion but should use higher order function(s) **`map`, `foldr/foldl`, or `filter`.**

## Testing your functions

The `HW2SampleTests.zip` file includes 10 `.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 P2a_HW2tests.hs
# run
```

Repeat the above for other HW problems by changing the test file name, i.e. , `P2b_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.