CptS355 - Lab 1 (Haskell) - Spring 2021

Assigned: Monday February 1, 2021

Due: Monday February 8, 2021

Weight: Lab 1 will count for 1% of your course grade.

This lab 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 lab submission

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 `Lab1.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, `Lab1SampleTests.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.

The instructor will show how to import and run tests on `GHCI` during the lecture. Please refer to the lecture video (February 1) on Canvas.

To submit your assignment, please upload the file `HW1.hs` on the Lab1 (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.

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.)
- The type of your functions should match with the type specified in each problem. You don't need to include the "type signatures" for your 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.
- When auxiliary 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.
- 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

Lab Problems:

1. `insert`

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. The type of `insert` can be one of the following:

```
insert :: (Num t1) => t1 -> t2 -> [t2] -> [t2]
insert :: (Eq t1, Num t1) => t1 -> t2 -> [t2] -> [t2]
insert :: (Ord t1, Num t1) => t1 -> t2 -> [t2] -> [t2]
```

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.)

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 []
[]
```

2. `insertEvery`

Write a function `insertEvery` that takes an integer "`n`", a value "`item`", and a list "`iL`" and inserts the "`item`" at **every n^{th} index** in "`iL`". "`n`" is a 1-based index, i.e., "item" should be inserted after n^{th}, $2n^{th}$, $3n^{th}$, etc. elements in the list. The type of `insertEvery` can be one of the following:

```
insertEvery :: (Num t) => t -> a -> [a] -> [a]
insertEvery :: (Eq t, Num t) => t -> a -> [a] -> [a]
```

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.)

Examples:

```
> insertEvery 3 100 [1,2,3,4,5,6,7,8,9,10]
[1,2,3,100,4,5,6,100,7,8,9,100,10]
> insertEvery 8 100 [1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7,8,100]
> insertEvery 9 100 [1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7,8]
> insertEvery 3 100 []
[]
```

3. getSales

Assume that you have an online sales business and you sell products on Amazon, Ebay, Etsy, etc. and you keep track of your daily sales (in $) for each online store. You maintain the log of your sales in a Haskel list.

First consider the sales log for a single store, for example:

```
storelog = [("Mon",50),("Fri",20), ("Tue",20),("Fri",10),("Wed",25),("Fri",30)]
```

`storelog` is a list of (day, sale-amount) pairs. Note that the store may have multiple sales on the same day of the week or may not have any sales on some days.

Write a function `getSales` that takes a "day" abbreviation (e.g. "Mon", "Tue", etc.) and a store's sales log as input and returns the total sales in that store for the given day.
The type of `getSales` should be `getSales :: (Num p, Eq t) => t -> [(t, p)] -> p`

Examples:

```
> getSales "Fri" storelog
60
> getSales "Mon" storelog
50
> getSales "Sat" storelog
0
```

4. sumSales

Now we combine the sales logs for all stores into one single list. An example list is given below:

```
sales = [("Amazon",[("Mon",30),("Wed",100),("Sat",200)]),
         ("Etsy",[("Mon",50),("Tue",20),("Wed",25),("Fri",30)]),
         ("Ebay",[("Tue",60),("Wed",100),("Thu",30)]),
         ("Etsy",[("Tue",100),("Thu",50),("Sat",20),("Tue",10)])]
```

The list includes tuples where the first value in the tuple is the store name and the second value is the list of (day, sale amount) pairs. Note that the list may include multiple entries (tuples) for the same store.

Write a function, `sumSales`, that takes a store name, a day-of-week, and a sales log list (similar to "`sales`") and returns the total sales of that store on that day-of-week.
(Hint: You can make use of `getSales` function you defined in part-a.)

The type of sumSales can be:

```
sumSales:: (Num p)=> String -> String -> [(String,[(String,p)])] -> p
```

```
> sumSales "Etsy" "Tue" mysales
130
> sumSales "Etsy" "Sun" mysales
0
> sumSales "Amazon" "Mon" mysales
30
```

5. split

You should implement your own split logic for this problem. You are not allowed to use any built-in or imported split function in your solution. (For example: https://hackage.haskell.org/package/split-0.2.3.4/docs/Data-List-Split.html)

aWrite a function `split` that takes a delimiter value "c" and a list "iL", and it splits the input list with respect to the delimiter "c". The goal is to produce a result in which the elements of the original list have been collected into ordered sub-lists each containing the elements between the occurrences of the delimiter in the input list. The delimiter value should be excluded from the sub-lists.

The type of `split` can be one of the following:

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

Examples:

```
> split ',' "Courses:,CptS355,CptS322,CptS451,CptS321"
["Courses:","CptS355","CptS322","CptS451","CptS321"]

> split 0  [1,2,3,0,4,0,5,0,0,6,7,8,9,10]
[[1,2,3],[4],[5],[],[6,7,8,9,10]]
```

6. nSplit

Write a function `nSplit` that takes a delimiter value "`c`", **an integer "`n`",** and a list "`iL`", and it splits the input list with respect to the delimiter "`c`" **up to "`n`" times**. Unlike `split`, it should not split the input list at every delimiter occurrence, but only for the first "`n`" occurrences of it.

The type of `nSplit` can be one of the following:

```
nSplit :: (Ord a1, Num a1, Eq a2) => a2 -> a1 -> [a2] -> [[a2]]
nSplit :: (Num a1, Eq a2) => a2 -> a1 -> [a2] -> [[a2]]
nSplit :: (Num a1) => a2 -> a1 -> [a2] -> [[a2]]
```

Examples:

```
> nSplit ',' 1 "Courses:,CptS355,CptS322,CptS451,CptS321"
["Courses:","CptS355,CptS322,CptS451,CptS321"]

> nSplit ',' 2 "Courses:,CptS355,CptS322,CptS451,CptS321"
["Courses:","CptS355","CptS322,CptS451,CptS321"]

> nSplit ',' 4 "Courses:,CptS355,CptS322,CptS451,CptS321"
["Courses:","CptS355","CptS322","CptS451","CptS321"]

> nSplit ',' 5 "Courses:,CptS355,CptS322,CptS451,CptS321"
["Courses:","CptS355","CptS322","CptS451","CptS321"]

> nSplit 0 3 [1,2,3,0,4,0,5,0,0,6,7,8,9,10]
[[1,2,3],[4],[5],[0,6,7,89,10]]
```

Testing your functions

The `Lab1SampleTests.zip` file includes 6 .hs files where each file includes the HUnit tests for a lab problem. The tests compare the actual osutput to the expected (correct) output and raise an exception if they don't match. The test files import the `Lab1` module (`Lab1.hs` file) which will include your implementations of the lab problems.
You will write your solutions to `Lab1.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.`

Haskell resources:

- **Learning Haskell**, by Gabriele Keller and Manuel M T Chakravarty (http://learn.hfm.io/)
- **Real World Haskell**, by Bryan O'Sullivan, Don Stewart, and John Goerzen (http://book.realworldhaskell.org/)
- **Haskell Wiki:** https://wiki.haskell.org/Haskell
- **HUnit**: http://hackage.haskell.org/package/HUnit