CptS355 - Assignment 1 (Haskell)
Fall 2020

Assigned: Monday September 7, 2020

Due: Thursday September 17, 2020

Weight: Assignment 1 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 `HW1.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, `HW1SampleTests.hs`, includes 2 to 4 sample unit tests for each problem. You will edit this file and provide additional tests (add at least 2 tests per problem). Please rename your test file as `HW1Tests.hs`.

The instructor will show how to import and run tests on `GHCI` during the lecture. Please refer to the lecture video (September 8) on Blackboard.

To submit your assignment, please upload both files (`HW1.hs` and `HW1Tests.hs`) on the Assignment1 (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.)
- 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
- 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. `biggerDate` and `maxDate` – 20%

a) **[8pts]** Assume we represent a "date" value as a Haskell tuple of type `(Int,Int,Int)` where the first part is the day, the second part is the month, and the third part is the year.

Write a function **`biggerDate`** which takes two "date" values as input and returns the bigger (i.e. the more recent) date.
The function should have type:

```
biggerDate::(Ord a1,Ord a2,Ord a3)=>(a3,a2,a1)->(a3,a2,a1)->(a3,a2,a1)
```

Examples:

```
> biggerDate (1,8,2020) (20,7,2020)
(1,8,2020)
> biggerDate (6,7,2020) (30,12,2019)
(6,7,2020)
```

b) **[12pts]** Write a function **`maxDate`** that takes a list of dates and returns the maximum date value in the input list. If the input list is empty, it raises an error.
You can make use of `biggerDate` function you defined in part-a.
The type of your function should be :

```
maxDate :: (Ord a1, Ord a2, Ord a3) => [(a3, a2, a1)] -> (a3, a2, a1)
```

Examples:

```
> maxDate [(31,8,2020),(20,7,2020),(1,9,2020),(30,12,2019),(6,7,2018),
(6,7,2020)]
(1,9,2020)
> maxDate [(31,8,2020)]
(31,8,2020)
```

2. `ascending` - 12%

Write a function `ascending` that takes a list of comparable values (i.e., having types from Ord typeclass) and it returns "`True`" if the elements in the list are in increasing order. It returns "`False`" otherwise. Consecutive duplicate values will not violate the ascending condition.

Your function should have type `ascending :: Ord t => [t] -> Bool`.

Examples:

```
> ascending [1]
True
> ascending [1,2,3,4,5,3,6,7]
False
> ascending [1,3,4,4,5,6,7,10,100,200]
True
```

3. insert and insertEvery – 20%

a) **(6 pts)** 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 []
[]
```

b) **(14 pts)** 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 []
[]
```

4. getSales and sumSales – 20%

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.

a) **(6 pts)** 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
```

b) **(14 pts)** 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` and `nSplit` - 23%

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)

a) **(8 pts)** Write 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]]
```

b) **(15 pts)** 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]]
```

(5%) Testing your functions

We will be using the `HUnit` unit testing package in CptS355. See http://hackage.haskell.org/package/HUnit for additional documentation.

You may install `HUnit` in 3 different ways:

Option1 (using cabal installer) :

Run the following commands on the terminal.

```
cabal update
cabal v1-install HUnit
```

Option2 (from source) :

1. First install `call-stack` package
   - Download the package from here: http://hackage.haskell.org/package/call-stack . Download the `call-stack-0.2.0.tar.gz` file and extract it.

   (To extract .gz and .tar files on Windows you may use 7zip (https://www.7-zip.org/))
   - Then switch to the `call-stack` directory and install `call-stack` using the following commands at the terminal/command prompt:

     ```
     runhaskell Setup configure
     runhaskell Setup build
     runhaskell Setup install
     ```
   - If you get "permission denied" errors:
     - on Windows, run the terminal or command line as administrator.
     - on Mac and Linux, run the command in `'sudo'` mode (or login as root).
2. Next install `HUnit` package:
   - Download the package from here: http://hackage.haskell.org/package/HUnit. Download the `HUnit-1.2.5.2.tar.gz` file and extract it.
   - Then switch to the `HUnit` directory and install `HUnit` using the following commands at the terminal/command prompt:

     ```
     runhaskell Setup configure
     runhaskell Setup build
     runhaskell Setup install
     ```

   If you get "permission denied" errors, follow the above guideline.

Option3 (using stack – use this option if you already installed stack) :

Run the following command on the terminal.

```
stack install HUnit
```

Running Tests

The file `HW1SampleTests.hs` provides 2 to 4 sample test cases comparing the actual output with the expected (correct) output for each problem. This file imports the `HW1` module (`HW1.hs` file) which will include your implementations of the given problems.

You are expected to add **at least 2 more test cases** for each problem. Make sure that your test inputs cover all boundary cases.

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. The instructor will further explain this during the lecture.

If you don't add new test cases you will be deduced at least 5% in this homework.

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