CptS355 - Assignment 1 (Haskell)
Fall 2021

Assigned: Sunday September 12, 2021
Weight: Assignment 1 will count for 6.5% 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 GHCI. You may download Haskell Platform 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 (Sublime, Visual Studio Code, etc.). We recommend you 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.
To submit your assignment, please upload both files (`HW1.hs` and `HW1Tests.hs`) on the Assignment1 (Haskell) DROPBOX on Canvas (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.)
- 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. everyOther – 10%

a) **[10pts]** Define a recursive function `everyOther` which takes a list as input and returns a list including "every other value" from the input list, starting with the first element.
The type of the `everyOther` function should be compatible with the following:
`everyOther :: [a] -> [a]`

Examples:

```
> everyOther "AaBbCcDdEeFfGgH"
"ABCDEFGH"["yes","oui","ja","evet","ye","shi","ie","nai"]
> everyOther [1,2,3,4,5,6,7,8,9,10]
[1,3,5,7,9]
> everyOther ["yes","no","oui","non","ja","nein","evet","hayir","ye","ani","shi",
"hai","ie","meiyou","nai","ochi" ]
["yes","oui","ja","evet","ye","shi","ie","nai"]
> everyOther ['A']
"A"
```

2. eliminateDuplicates, getAllSeconds, and clusterCommon – 30%

a) **[5pts]** Define a recursive function `eliminateDuplicates` which takes a list as input, and it returns the unique values from this list. The unique elements in the output can be in arbitrary order. You may use the `elem` function in your implementation.
The type of the `eliminateDuplicates` function should be compatible with the following:
`eliminateDuplicates:: Eq a => [a] -> [a]`

Examples:

```
> eliminateDuplicates [6,5,1,6,4,2,2,3,7,2,1,1,2,3,4,5,6,7]
[1,2,3,4,5,6,7]
> eliminateDuplicates "CptS322 - CptS322 - CptS 321"
"-CptS 321"
> eliminateDuplicates [[1,2],[1],[],[3],[1],[]]
[[1,2],[3],[1],[]]
> eliminateDuplicates ["Let","it","snow","let","it","rain","let","it","hail"]
["Let","snow","rain","let","it","hail"]
```

b) **[5pts]** Define a recursive function `matchingSeconds` which takes a value `v` and a list of tuples as input and returns the list of second values from the tuples in the input list whose first values match `v`. The order of the elements in the output list should be same as the order they appear in the input list.
The type of the `matchingSeconds` function should be compatible with the following:
`matchingSeconds :: Eq t => t -> [(t, a)] -> [a]`

Examples:

```
> matchingSeconds "cat" [("cat",5),("dog",9),("parrot",3),("cat",3),("fish",1)]
[5,3]
> matchingSeconds "hamster" [("cat",5),("dog",9),("parrot",3),("cat",3),("fish",1)]
[]
> matchingSeconds "CptS" [("EE",214),("CptS",355),("CptS",302), ("CptS",322)]
[355,302,322]
```

c) **[20pts]** Define a recursive function `clusterCommon` which takes a list of tuples as input and combines the tuples with common first elements into a single tuple where the first value is the common element, and the second value is the list of their second values. It returns a list of such combined tuples. The first elements of the tuples in the output should be unique. The output list can have arbitrary order. The type of the `clusterCommon` function should be compatible with the following:

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

Note: *You can use the functions* `matchingSeconds` *and* `eliminateDuplicates` *in your solution.*

Examples:

```
> clusterCommon [("cat",5),("dog",10),("parrot",3),("dog",5),("dog",7),("cat",3),
("fish",1)]
[("parrot",[3]),("dog",[10,5,7]),("cat",[5,3]),("fish",[1])]
> clusterCommon [(1,10),(4,400),(3,3),(2,20),(3,30),(1,1),(4,40),(3,300)]
[(2,[20]),(1,[10,1]),(4,[400,40]),(3,[3,30,300])]
> clusterCommon []
[]
```

3. maxNumCases – 15%

Assume you work for a "Healthcare Data Analytics" company and you write scripts to process various dataset. In your analysis, you use the CDC's COVID-19 dataset.
For example, the following dataset reports the monthly new COVID cases for some counties in WA.

```
cdcData =[("King",[("Mar",2706),("Apr",3620),("May",1860),("Jun",2157),("July",5014),
                   ("Aug",4327),("Sep",2843)]),
          ("Pierce", [("Mar",460),("Apr",965),("May",522),("Jun",2260),("July",2470),
                   ("Aug",1776),("Sep",1266)]),
          ("Snohomish",[("Mar",1301),("Apr",1145),("May",532),("Jun",568),
                   ("July",1540),("Aug",4360),("Sep",811)]),
          ("Spokane",[("Mar",147),("Apr",4000),("May",233),("Jun",794),("July",2412),
                   ("Aug",1530),("Sep",1751)]),
          ("Whitman" , [("Apr",7),("May",5),("Jun",19),("July",51),("Aug",514),
                   ("Sep",732), ("Oct",278)]) ]
```

`cdcData` is a list of tuples where the first value in the tuple is the county name and the second is the list of tuples. Each tuple in this list includes the month and number of cases in that month.
Note that some counties may not have any new cases in some months. If there is no data for a given month, you should assume that the number of new cases for that month is `0`.
Define a function, `maxNumCases`, which calculates the *maximum number of new cases* among all counties *for a given month*.
The type of the `maxNumCases` function should be compatible with the following:

```
maxNumCases :: (Num p, Ord p, Eq t) => [(a, [(t, p)])] -> t -> p
```

Examples:

```
> maxNumCases cdcData "Jun"
2260
> maxNumCases cdcData "Apr"
4000
> maxNumCases cdcData "Jan"
0
```

4. groupIntoLists – 20%

Write a function `groupIntoLists` that takes a list as input and returns a nested list (i.e., list of lists). The function groups the input list elements into sublists of increasing length. The last sublist will include the leftover elements.

The type of `groupIntoLists` should be compatible with the following:

```
groupIntoLists :: [a] -> [[a]]
```

Examples:

```
> groupIntoLists [1,2,3,4,5,6,7,8,9,10,11,12]
[[1],[2,3],[4,5,6],[7,8,9,10],[11,12]]
> groupIntoLists "abcdefghijklmnopqrstuwxyz012"
["a","bc","def","ghij","klmno","pqrstu","wxyz012"]
> groupIntoLists ""
[]
```

5. getSlice – 20%

Write a function `getSlice` that takes a 2-tuple, which include two delimiter elements and a list, and it returns the slice of the list enclosed between those delimiters. The function should return the slice for the first occurrence of the delimiter characters in the list. If the first delimiter doesn't exist in the list, it should return `[]`. Also, if the second delimiter doesn't exist, it should return the slice of the list from the first delimiter to the end of the list.

The type of `getSlice` should be compatible with the following:

```
getSlice :: Eq a => (a, a) -> [a] -> [a]
```

Examples:

```
> getSlice  ('(',')') "I got the (Covid-19) vaccine!"
"Covid-19"
> getSlice ('(',')') "I hope this year (2021) will be better than last year (2020)."
"2021"
> getSlice  ('*','*') "Start the assignment *early*!"
"early"
> getSlice (0,9)  [1,2,3,4,0,5,6,7,8,9,10,11]
[5,6,7,8]
> getSlice (0,9)  [1,2,3,4,5,6,7,8,9,10,11]
[]
> getSlice (0,9)  [1,2,3,4,0,5,6,7,8,10,11]
[5,6,7,8,10,11]
```

(5%) Testing your functions

Install `HUnit`

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

Windows (using cabal installer) :

Run the following commands on the terminal.

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

Mac (using stack installer)
Run the following commands on the terminal.

```
stack install HUnit
```

Check the attached `HUnit_HowtoInstall.pdf` document for other options to 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