CptS355 - Assignment 1 (Haskell) Su 2023

Assigned: Monday, 15 May 2023

Due: Friday, 26 May 2023 by 10 p.m.

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/downloads/.

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 posted **`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 use Visual Studio Code, since it has better support for Haskell.

In addition, you will write unit tests using `HUnit` testing package. The posted file, **`HW1Tests.hs`**, includes 3 sample test cases for each problem. You will edit this file and provide additional tests for each problem (*at least 2 additional tests per problem*).

The instructor will show how to import and run tests on `GHCI` during the lecture. Please email or visit the instructor if you run into difficulties with your system/setup.

To submit your assignment, please upload both files (`HW1.hs` and `HW1Tests.hs`) to Canvas on the Assignment1 (Haskell) section under Content->Assignments. DO NOT zip your files. 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. Otherwise you will be deducted points (around 40%).
- 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. `exists` – 10%

a) **[8pts]** Write a function **`exists`** which takes a "value" and a "list" as input. If the value is a member of the list, the function should return `True`. Otherwise it should return `False`.
The function should have type `exists :: Eq t => t -> [t] -> Bool`.

Examples:

```
> exists 1 []
False
> exists 1 [1,2,3]
True
> exists [1] [[1]]
True
> exists [1] [[3],[5]]
False
> exists '3' "CptS355"
True
```

b) **[2pts]** Explain in a comment why the type `is`
`exists :: Eq t => t -> [t] -> Bool`
but not
`exists :: t -> [t] -> Bool`

2. listUnion - 15%

Write a function `listUnion` that takes two lists as input and returns the union of those lists. Your function should have type `listUnion :: Eq a => [a] -> [a] -> [a]`.
Each value should appear in the output list only once, but the order does not matter. Please note that the input lists may have duplicate values or there may be values that appear in both input lists. All such duplicate values should be removed.

Examples:

```
> listUnion [1,3,4] [2,3,4,5]
[1,2,3,4,5]
> listUnion [1,1,2,3,3,3] [1,3,4,5]
[2,1,3,4,5]
> listUnion "CptS355" "cpts322"
"5SCcpts32"
> listUnion [[1,2],[2,3]] [[1],[2,3],[2,3]]
[[1,2],[1],[2,3]]
```

3. replace – 15%

Write a function `replace` that takes an index **n**, a value **v**, and a list **L** and returns a (new) list which is the same as **L**, except that its **n**th element is **v**. Assume 0-based indexing for **n** and **n≥0 .** (Note that **n** can be greater than the length of the list **L**.)
The type of `replace` should be:
`replace :: (Eq t1, Num t1) => t1 -> t2 -> [t2] -> [t2].`

Examples:
```
> replace 3 40 [1, 2, 3, 4, 5, 6]
[1,2,3,40,5,6]
> replace 0 'X' "abcd"
"Xbcd"
> replace 4 False [True, False, True, True, True]
[True,False,True,True,False]
> replace 5 6 [1,2,3,4,5]
[1,2,3,4,5]
> replace 6 7 [1,2,3,4,5]
[1,2,3,4,5]
```

4. prereqFor – 20%

Assume that we store the list of CptS courses and their prerequisites as a list of tuples. The first element of each tuple is the course name and the second element is the list of the prerequisites for that course. See below for an example. Please note that a course may have an arbitrary number of prerequisites.

```
prereqsList =
  [ ("CptS122" , ["CptS121"]),
    ("CptS132" , ["CptS131"]),
    ("CptS223" , ["CptS122", "MATH216"]),
    ("CptS233" , ["CptS132", "MATH216"]),
    ("CptS260" , ["CptS223", "CptS233"]),
    ("CptS315" , ["CptS223", "CptS233"]),
    ("CptS317" , ["CptS122", "CptS132", "MATH216"]),
    ("CptS321" , ["CptS223", "CptS233"]),
    ("CptS322" , ["CptS223","CptS233"]),
    ("CptS350" , ["CptS223","CptS233", "CptS317"]),
    ("CptS355" , ["CptS223"]),
    ("CptS360" , ["CptS223","CptS260"]),
    ("CptS370" , ["CptS233","CptS260"]),
    ("CptS427" , ["CptS223","CptS360", "CptS370", "MATH216", "EE234"])
  ]
```

Assume that you are creating an application for WSU. You would like to write a Haskell function `prereqFor` that takes the list of courses (similar to above) and a particular course number and returns the list of the courses which require this course as a prerequisite.
The type of `prereqFor` should be `prereqFor :: Eq t => [(a, [t])] -> t -> [a].`
(Hint: You can make use of `exists` function you defined earlier.)

Examples:
```
> prereqFor prereqsList "CptS260"
```

```
["CptS360","CptS370"]
> prereqFor prereqsList "CptS223"
["CptS260","CptS315","CptS321","CptS322","CptS350","CptS355","CptS360",
 "CptS427"]
> prereqFor prereqsList "CptS355"
[]
> prereqFor  prereqsList "MATH216"
["CptS223","CptS233","CptS317","CptS427"]
```

5. isPalindrome – 20%

A palindrome is a sentence or phrase that is the same forwards and backwards, ignoring spaces, punctuation and other special characters, and upper vs. lower case. In this problem we will consider palindromes that include only letters, digits, and spaces but don't include any punctuation or any special characters — for example "*a01 02 2010A*", "*Yreka Bakery*", and "*Doc note I dissent a fast never prevents a fatness I diet on cod*". Assume that letters are case insensitive — for example "*Madam Im Adam*"
Write a Haskell function `isPalindrome` that takes a string as argument and that returns `True` if the string is a palindrome and `False` otherwise.
(*Note*: you can make use of the following Haskell functions: `reverse` (for reversing a list), `toUpper` (for retrieving the uppercase of a given character). You can include `toUpper` from the `Data.Char` module, i.e., include the following line in the beginning of your module:
`import Data.Char (toUpper)`)

The type of `isPalindrome` should be: `isPalindrome :: [Char] -> Bool`

Examples:

```
> isPalindrome "a01 02 2010A"
True
> isPalindrome "Doc note I dissent a fast never prevents a fatness I diet on
cod"
True
> isPalindrome "Yreka Bakery"
True
> isPalindrome "top cart pop tracPOT"
True
```

6. groupSumtoN - 20%

Write a Haskell function `groupSumtoN` that takes two arguments, where the first argument is an integer (`N`) and the second is a list (`L`). The goal is to produce a result in which the elements of the original list have been collected into ordered sub-lists each containing maximum number of consecutive elements from `L` summing up to `N` or less (where `N` is the integer argument). The leftover elements (if there are any) are included as the last sub-list with a sum less than `N`. If an element in the input list `L` is greater than `N`, that element should be included in its own sublist (including that element only).

The type of `groupSumtoN` should be:

```
groupSumtoN :: (Ord a, Num a) => a -> [a] -> [[a]]
```

Note: this function is not required to be tail-recursive.

Examples:

```
> groupSumtoN 15 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[[1,2,3,4,5],[6,7],[8],[9],[10]]
> groupSumtoN 11 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[[1,2,3,4],[5,6],[7],[8],[9],[10]]
> groupSumtoN 1 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[[1],[2],[3],[4],[5],[6],[7],[8],[9],[10]]
> groupSumtoN 5 []
[[]]
> groupSumtoN 55 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[[1,2,3,4,5,6,7,8,9,10]]
```

Testing your functions

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

To setup `HUnit`:

1. First install `call-stack` package
   - Download the package from here: http://hackage.haskell.org/package/call-stack . Download the `call-stack-0.4.0.tar.gz` file and extract it.

     (To extract .gz and .tar files on Windows you may use 7zip (https://www.7-zip.org/))

     (To extract .gz and .tar files on Linux or Mac use tar—'`tar xvfz <filename>.tar.gz`')
   - 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 the last command:
     - on Windows, run the terminal or command line as administrator
     - on Mac and Linux, run the command in '`sudo`' mode
2. Next install `HUnit` package:
   - Download the package from here: http://hackage.haskell.org/package/HUnit. Download the `HUnit-1.6.2.0.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.

3. The file `HW1Tests.hs` provides 3 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 deducted *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