Programming in Haskell – Homework Assignment 2

UNIZG FER, 2016/2017

Handed out: October 20, 2016. Due: October 27, 2016 at 17:00

Note: Define each function with the exact name specified. You can (and in most cases you should) define each function using a number of simpler functions. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message. Problems marked with a star ($\star$) are optional.

Each problem is worth a certain number of points. The points are given at the beginning of each problem or subproblem (if they are scored independently). These points are scaled, together with a score for the in-class exercises, if any, to 10. Problems marked with a star ($\star$) are scored on top of the mandatory problems, before scaling. The score is capped at 10, but this allows for a perfect score even with some problems remaining unsolved.

1. *(3 pts)* We can use a list of lists, where all sublists are of equal length, to represent a matrix. Define the following functions over such a matrix:

   (a) The function `isWellFormed` that checks whether the matrix has all rows of equal length.

   ```
   isWellFormed ::  [[Int]] -> Bool

   isWellFormed [[1,2,3],[4,5,6],[7,8,9]] ⇒ True
   isWellFormed [[1,2,3],[4,5]] ⇒ False
   isWellFormed [[]] ⇒ False
   ```

   (b) The function `size` that returns the dimensions of a $n \times m$ matrix as a pair `(n,m)`.

   ```
   size ::  [[Int]] -> (Int, Int)

   size [[1,2,3],[4,5,6]] ⇒ (2,3)
   size [[5,0,5],[2]] ⇒ error "Matrix is malformed"
   size [[]] ⇒ error "Matrix is malformed"
   ```

   (c) The function `getElement` that returns the element at the given position in matrix.

   ```
   getElement ::  [[Int]] -> Int -> Int -> Int

   getElement [[9,8,7],[6,5,4],[3,2,1]] 1 0 ⇒ 6
   getElement [[9,8,7],[6,5,4],[3,2,1]] 3 0
   ```

```
⇒ error "Index out of bounds"
getElement [[3,2,1],[5,4]] 0 0 ⇒ error "Matrix is malformed"
getElement [[1],[2,3]] (-1) 0 ⇒ either of the errors above
```

(d) The function `getRow` that returns the `i`–th row of a matrix.

```
getRow ::  [[Int]] -> Int -> [Int]

getRow [[1,2],[3,4],[5,6]] 0 ⇒ [1,2]
getRow [[1,2,3],[4,5]] 1 ⇒ error "Matrix is malformed"
getRow [[1,2,3]] (-3) ⇒ error "Index out of bounds"
getRow [[1,2],[3,4,5]] 2 ⇒ either of the errors above
```

(e) The function `getCol` that returns the `i`–th column of a matrix.

```
getCol ::  [[Int]] -> Int -> [Int]

getCol [[1,2,3],[4,5,6]] 0 ⇒ [1,4]
getCol [[1,2,3],[4,5]] 1 ⇒ error "Matrix is malformed"
getCol [[1,2,3]] (-3) ⇒ error "Index out of bounds"
getCol [[1,2],[3,4,5]] 2 ⇒ either of the errors above
```

(f) The function `addMatrices` that returns the sum of two given matrices.

```
addMatrices ::  [[Int]] -> [[Int]] -> [[Int]]

addMatrices [[1,2,3],[4,5,6]] [[7,8,9],[10,11,12]]
⇒ [[8,10,12],[14,16,18]]
addMatrices [[1,2,3],[4,5,6]] [[1]]
⇒ error "Matrices are not of equal size"
addMatrix [[1,2],[3,4]] [[5,6],[7]] ⇒ error "Matrix is malformed"
```

(g) The function `transpose'` that returns a transposed version of the given matrix. It may not use the `Data.List.transpose` function.

```
transpose' ::  [[Int]] -> [[Int]]

transpose' [[1,2,3],[4,5,6]] ⇒ [[1,4],[2,5],[3,6]]
transpose' [[1,2,3],[4,5]] ⇒ error "Matrix is malformed"
```

(h) The function `multMatrices` that multiplies two matrices.

```
multMatrices ::  [[Int]] -> [[Int]] -> [[Int]]

multMatrices [[1,2],[3,4],[5,6]] [[7,8],[9,10]]
⇒ [[25,28],[57,64],[89,100]]
multMatrices [[1,0],[0,1]] [[3,5],[9,2]] ⇒ [[3,5],[9,2]]
multMatrices [[1,2],[3,4]] [[5]]
⇒ error "Incompatible matrix dimensions"
multMatrices [[1,2],[3]] [[4]] ⇒ error "Matrix is malformed"
```

2. *(3 pts)* Even though this is avoided because of inefficiency, a dictionary can be represented with a list of key-value pairs. For instance, [(12, "aardvark"), (37, "elephant")] could be dictionary containing two items: "aardvark" for key 12 and "elephant" for key 37. Its type is [(Int, String)]. For this assignment, implement the following functions:

```
type Key = Int
type Value = String
type Entry = (Key, Value)
type Dictionary = [Entry]
type Frequency = [(Value, Int)]
```

   (a) `exists` function which checks whether a given key exists in a dictionary.

```
exists :: Key -> Dictionary -> Bool
exists 5 [(0, "cow")] => False
exists 5 [(0, "panthera"), (5, "skunk")] => True
exists 5 [] => False
```

   (b) `get` function which returns the value associated with the given key.

```
get :: Dictionary -> Key -> Value
get [(5, "lynx"), (4, "skunk"), (12, "wallaby")] 5 => "lynx"
get [(5, "lynx"), (4, "skunk"), (12, "wallaby")] 8
=> error "key 8 not found"
```

   (c) `insert` function which returns the dictionary with the inserted entry. If the key is already present in the dictionary the function should alter the current value for that key.

```
insert :: Entry -> Dictionary -> Dictionary
insert (3, "alpaca") [] => [(3, "alpaca")]
insert (7, "lion") [(12, "goose"), (7, "crow)]
=> [(12, "goose"), (7, "lion")]
```

   (d) `delete` function which removes the entry for a given key.

```
delete :: Key -> Dictionary -> Dictionary
delete 12 [(4, "woodcock"), (12, "beaver")] => [(4, "woodcock")]
delete 1 [(4, "woodcock"), (12, "beaver")]
=> [(4, "woodcock", (12, "beaver")]
```

   (e) `freq` function which returns the number of appearances of every value in the dictionary.

```
freq :: Dictionary -> Frequency
freq [(12, "beaver"), (4, "skunk"), (17, "beaver)] =>
 [("beaver", 2), ("skunk", 1)]
freq [] => error "dictionary is empty"
freq [(4, "hare"), (12, "hare"), (11, "hare")] => [("hare", 3)]
```

3. *(3 pts)* John celebrated his 30th birthday among his best friends with large quantities of grape juice. He got many presents from his friends but he was most amazed with the gift from Jake. Jake got him a natural number `N`. John immediately started discovering properties of this amazing number `N` but he got stuck trying to find out what is the largest multiple of the number 30 which can be constructed by permuting the digits of the number `N`. Write a program that can help John find such a number, or report an error if there is no such number. John also told you that he really likes Haskell, especially the `Data.List` module.

```
largestMultiple ::  String -> Int

largestMultiple "120" ⇒ 210
largestMultiple "2391" ⇒ *** Exception:  No such number
largestMultiple "30" ⇒ 30
largestMultiple "303" ⇒ 330
```

4. ($[0, \infty\rangle$ *pts*)* (Note that this assignment is marked with a star and is optional.)

   Last week we talked about types. Types are very important in Haskell and they are of great help to the programmer when reasoning about his program. So, let's do some more thinking about types.

   Consider the function `undefined ::  a`. This might seem like some special language construct, but like many other things in Haskell it's actually just "syntactic sugar". The purpose of this function is to use it as a placeholder when you want you program to compile for the moment, but you will provide the actual implementation later. When undefined is evaluated it's just throws an `*** Exception: Prelude.undefined` error. Think about it's type and why it works. Write your own implementations, `undefined'` which throws a similar error, and `undefined''` which results in bottom ("infinite loop"). To get the maximum learning effect, try to figure out why this works without Googling first, and then seek further explanation.

   One more example. Consider the function with the type signature `foo ::  a -> b -> a`. Write as many implementations of such function as you can think of. Your functions must evaluate in finite time and cannot cause an error. You will get a point for every function you write. Please provide comment on every function you write containing your deepest thoughts about why it works.

Corrections

There are no Easter eggs down here, go away.