Programming in Haskell – Homework Assignment 1

UNIZG FER, 2016/2017

Handed out: October 13, 2016. Due: October 20, 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. *(2 pts)* Tuples are a natural way of representing vectors. However, to make this representation useful, we need it to support a set of basic vector operations. You will implement vectors in 2D space using 2-tuples, as `(x, y)`, by implementing the following functions:

   (a) Define a `norm` function that computes the Euclidean norm of a vector. (Hint: see the `sqrt` function.)

   ```
   norm (3, 4) ⇒ 5.0
   norm (0, 0) ⇒ 0.0
   norm (3, 0) ⇒ 3.0
   ```

   (b) Define a `normalize` function that returns a normalized version of a given vector.

   ```
   normalize (3, 4) ⇒ (0.6, 0.8)
   normalize (1, 1) ⇒ (0.7071, 0.7071)
   normalize (0, 0) ⇒ error "Cannot normalize null vector"
   ```

   (c) Define a `scalarMult` function that takes a scalar and a vector and returns their product.

   ```
   scalarMult 0 (3, 3) ⇒ (0, 0)
   scalarMult (-1) (2, 3) ⇒ (-2, -3)
   scalarMult 2 (1, 2) ⇒ (2, 4)
   ```

   (d) Define a `dot` function that computes the dot product of two vectors.

   ```
   dot (1, 1) (1, 1) ⇒ 2
   dot (3, 5) (2, 1) ⇒ 11
   ```

```
dot (2, -1) (0, 5) ⇒ -5
```

(e) Define the `cos'` function that computes the cosine of the angle between two vectors. Use the functions you defined in previous subtasks.

```
cos' (1, 0) (1, 0) ⇒ 1.0
cos' (1, 0) (0, 1) ⇒ 0.0
cos' (3, 2) (5, 4) ⇒ 0.9962
cos' (0, 0) (1, 1) ⇒ error "Null vector given"
cos' (1, 1) (0, 0) ⇒ error "Null vector given"
```

(f) Define the `areParallel` function that checks whether two given vectors are parallel, returning a boolean.

```
areParallel (1, 1) (1, 1) ⇒ True
areParallel (1, 1) (2, 1) ⇒ False
areParalell (1, 1) (2, 2) ⇒ True
```

2. *(2 pts)* Define function `splitAt'` which splits the list at index `n` and returns the result as a tuple with the first element containing the first part, and the second element containing the second part of the list. If a list can't be split into two non-empty sublists return an error. Use `if-then-else` or guards.

```
splitAt' 5 [1..10] ⇒ ([1..5], [6..10])
splitAt' 10 [1..5] ⇒ error "n is out of range"
splitAt' 1 [1..10] ⇒ ([1], [2..9])
splitAt' (-3) [5..10] ⇒ error "n is out of range"
```

3. *(3 pts)* In this assignment you will implement the Luhn algorithm. The Luhn algorithm or Luhn formula is a simple checksum formula used to validate a variety of identification numbers, such as credit card or IMEI numbers. Pseudocode for the algorithm is given below:

   (a) Double the value of every second digit beginning from the right. That is, the last digit is unchanged; the second-to-last digit is doubled; the third-to-last digit is unchanged; and so on. For example, `[5,5,9,4]` becomes `[10,5,18,4]`.

   (b) Add the digits of the doubled values and the undoubled digits from the original number. For example, `[10,5,18,4]` becomes `(1 + 0) + 5 + (1 + 8) + 4 = 19`.

   (c) Calculate the remainder when the sum is divided by 10. For the above example, the remainder would be 9. If the result equals 0, then the number is valid.

   Given string of numbers, calculate if the number is valid or not. Write as many helper functions as you want.

   (Hint: check the digitToInt function. If you decide to use it add `import Data.Char (digitToInt)` to the beginning of your code. You can, of course, implement your own version)

```
luhn "1231252352352" ⇒ False
luhn "5555555555554444" ⇒ True
luhn "5105105105105100" ⇒ True
luhn "18" ⇒ True
luhn "17" ⇒ False
```

4. *(2 pts*)* (Note that this assignment is marked with a star and is optional.)

   Write a factorization function taking a number and returning a list of it's factors: `factorize :: Int -> [Int]`.
   You can then use this function to help you define: `primes :: Int -> [Int]`
   Don't worry about computation efficiency at this point.

   ```
   factorize 123 ⇒ [1,3,41,123]
   factorize 47 ⇒ [1,47]
   factorize 500 ⇒ [1,2,4,5,10,20,25,50,100,125,250,500]
   primes 5 ⇒ [2,3,5,7,11]
   primes 15 ⇒ [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
   ```

Corrections

"Alea iacta est" -
Gaius Julius Caesar after signing up for Haskell course.