Functional and Logic Programming

Home Assignment 1

Due: Saturday, 21.4.2018 - 23:55

Instructions

- Please create a source file called ***hw1.hs*** and put all the answers there.
  The file should start with a comment which contains your **full name** (in English) and **ID**
  (see also example: ***hwexample.hs*** in the Moodle)

  ```
  -- John Doe
  -- 654321987
  ```

- Make sure the file **is valid** by loading it into GHCi.
  A valid file will load without any errors or warnings.

- If you need a function but you don't know how to implement it - just write it's signature (name and type) and put `undefined` in the function's body.
  That way you'll be able to load the file even though it contains references to undefined names.
  (see also example: ***hwexample.hs*** in the Moodle)

- When writing a function - write both the **type** and the **body** of the function.

- Be sure to write functions with **exactly the specified name** (and **type signature** - if it is provided) for each exercise.
  You may create additional auxiliary/helper functions with whatever names and type signatures you wish.

- Try to write **small functions** which perform just **a single task**, and then **combine** them to create more complex functions.

Exercises

1. Write a function `isPalindrome`, which takes a `String` and returns `True` if that `String` is a palindrome and `False` otherwise.

```
isPalindrome ""        = True
isPalindrome "x"       = True
isPalindrome "radar"   = True
isPalindrome "racecar" = True
isPalindrome "lambda"  = False
isPalindrome "haskell" = False
```

2. Write a function `isPrefix`, which takes 2 `String`s and returns `True` if the first `String` is a prefix of the second `String`, and `False` otherwise.

```
isPrefix "pre"   "prefix" = True
isPrefix ""      "word"   = True
isPrefix "abc"   "abcde"  = True
isPrefix "abcde" "abc"    = False
isPrefix "abc"   "def"    = False
```

3. a) Write a function `squareList`, which takes an `Int` `n` and produces a list of the first `n` squares in **descending** order.
(You may assume the input `n` is always non-negative.)

```
squareList 0 = [0]
squareList 3 = [9, 4, 1, 0]
squareList 5 = [25, 16, 9, 4, 1, 0]
```

   b) Write a function `listSquare`, which takes an `Int` `n` and produces a list of the first `n` squares in **ascending** order, using `++`.
(You may assume the input `n` is always non-negative.)

```
listSquare 0 = [0]
listSquare 3 = [0, 1, 4, 9]
listSquare 5 = [0, 1, 4, 9, 16, 25]
```

4. Write a function `fact`, which calculates the factorial of a given number by **accumulating** the result.
The function should take an `Integer` as input and return an `Integer` as output.
(You may assume the input is always non-negative)
(**Hint:** use an auxiliary function with 2 inputs)

```
fact 0 = 1
fact 3 = 6
fact 5 = 120
```

5. In this exercise we will implement a **credit card validation function** using a **checksum**. The checksum algorithm is as follows:

   - 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, `[1, 3, 8, 6]` becomes `[2, 3, 16, 6]`.
   - Add the digits of the doubled values and the undoubled digits from the original number.
   For example, `[2, 3, 16, 6]` becomes `2 + 3 + 1 + 6 + 6 = 18`.
   - Calculate the remainder when the sum is divided by `10`. If the result equals `0`, then the number is valid.

   Implement the following steps of the validation algorithm:

   a) Create a function:

   ```
   toDigits :: Integer -> [Integer]
   ```

   which converts positive `Integer`s to a list of its digits.
   (For `0` and negative numbers it should return the empty list)

   ```
   toDigits 1234  = [1, 2, 3, 4]
   toDigits 0     = []
   toDigits (-17) = []
   ```

   b) Create a function:

   ```
   doubleEveryOther :: [Integer] -> [Integer]
   ```

   that doubles every other number in a list *beginning from the right* - so it doubles the second-to-last number, the fourth-to-last number, etc.

   ```
   doubleEveryOther [8, 7, 6, 5] = [16, 7, 12, 5]
   doubleEveryOther [1, 2, 3]    = [1, 4, 3]
   ```

c) Create a function:

```
sumDigits :: [Integer] -> Integer
```

that takes a list of numbers (not necessarily one-digit numbers) and returns the sum of all the digits of all the numbers in it.
If the list is empty it returns `0`.

```
sumDigits [16, 7, 12, 5] = 1 + 6 + 7 + 1 + 2 + 5 = 22
```

d) Create a function:

```
validate :: Integer -> Bool
```

that indicates whether a given `Integer` is a valid credit card number by checking if the remainder of the given sum divided by `10` is equal to `0`.
This will use all the functions defined in the previous sections.

```
validate 4012888888881881 = True
validate 4012888888881882 = False
```