CIS 194: Homework 1

Due Friday, September 5

When solving the homework, strive to create not just code that works, but code that is stylish and concise. See the style guide on the website for some general guidelines. Try to write small functions which perform just a single task, and then combine those smaller pieces to create more complex functions. Don't repeat yourself: write one function for each logical task, and reuse functions as necessary.

Be sure to write functions with exactly the specified name and type signature for each exercise (to help us test your code). You may create additional helper functions with whatever names and type signatures you wish.

You are allowed to use functions in the `Data.List` standard library. You can find a list of these functions at `http://hackage.haskell.org/package/base-4.7.0.1/docs/Data-List.html`.

Administrivia

- Sign up on Piazza at `http://piazza.com/upenn/fall2014/cis194`. We will be using Piazza for Q&A and online discussions.
- Fill out the "Student Survey" on Canvas. It's listed as a quiz on that site.

Setup

To aid you in this first assignment, we have provided a skeleton *HW01.hs*. Download the file (available from the Lectures page on the course website) and make sure you can load it into GHCi. If you can't get this working, seek help! (Piazza is a good place to start.)

Validating Credit Card Numbers[1]

Have you ever wondered how websites validate your credit card number when you shop online? They don't check a massive database of numbers, and they don't use magic. In fact, most credit providers rely on a checksum formula for distinguishing valid numbers from random collections of digits (or typing mistakes).

[1] Adapted from the first practicum assigned in the University of Utrecht functional programming course taught by Doaitse Swierstra, 2008-2009.

In this section, you will implement the validation algorithm for credit cards. It follows these steps:

- 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. For the above example, the remainder would be `8`.

  If the result equals 0, then the number is valid.

Exercise 1 We first need to be able to break up a number into its last digit and the rest of the number. Write these functions:

```
lastDigit     :: Integer -> Integer
dropLastDigit :: Integer -> Integer
```

If you're stumped, look through some of the arithmetic operators mentioned in the lecture.

Example: `lastDigit 123 == 3`

Example: `lastDigit 0 == 0`

Example: `dropLastDigit 123 == 12`

Example: `dropLastDigit 5 == 0`

Exercise 2 Now, we can break apart a number into its digits. Define the function

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

`toDigits` should convert positive `Integer`s to a list of digits. (For `0` or negative inputs, `toDigits` should return the empty list.)

Example: `toDigits 1234 == [1,2,3,4]`

Example: `toDigits 0 == []`

Example: `toDigits (-17) == []`

Exercise 3 Once we have the digits in the proper order, we need to double every other one. Define a function

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

Remember that `doubleEveryOther` should double every other number *beginning from the right*, that is, the second-to-last, fourth-to-last, ... numbers are doubled.

Note that it's much easier to perform this operation on a list of digits that's in *reverse* order. You will likely need helper functions to make this work.

Example: `doubleEveryOther [8,7,6,5] == [16,7,12,5]`

Example: `doubleEveryOther [1,2,3] == [1,4,3]`

Exercise 4 The output of `doubleEveryOther` has a mix of one-digit and two-digit numbers. Define the function

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

to calculate the sum of all digits.

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

Exercise 5 Define the function

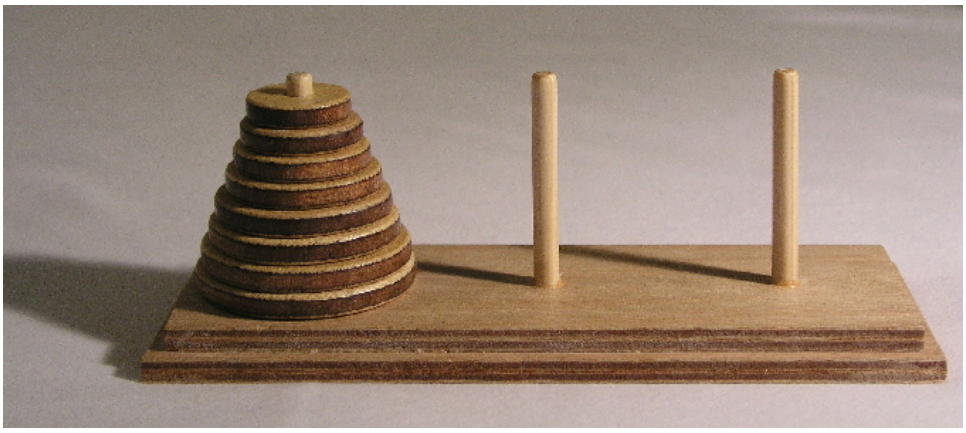
```
validate :: Integer -> Bool
```

that indicates whether an `Integer` could be a valid credit card number. This will use all functions defined in the previous exercises.

Example: `validate 4012888888881881 = True`

Example: `validate 4012888888881882 = False`

The Towers of Hanoi[2]



Exercise 6 The *Towers of Hanoi* is a classic puzzle with a solution that can be described recursively. Disks of different sizes are stacked on three pegs; the goal is to get from a starting configuration with all disks stacked on the first peg to an ending configuration with all disks stacked on the last peg, as shown in Figure 1.

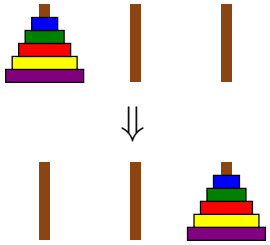


Figure 1: The Towers of Hanoi

The only rules are

- you may only move one disk at a time, and
- a larger disk may never be stacked on top of a smaller one.

For example, as the first move all you can do is move the topmost, smallest disk onto a different peg, since only one disk may be moved at a time.

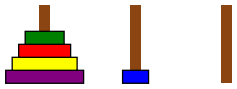


Figure 2: A valid first move.

From this point, it is *illegal* to move to the configuration shown in Figure 3, because you are not allowed to put the green disk on top of the smaller blue one.

Figure 3: An illegal configuration.

To move n discs (stacked in increasing size) from peg a to peg b using peg c as temporary storage,

1. move $n-1$ discs from a to c using b as temporary storage
2. move the top disc from a to b
3. move $n-1$ discs from c to b using a as temporary storage.

For this exercise, define a function `hanoi` with the following type:

```
type Peg = String
type Move = (Peg, Peg)
hanoi :: Integer -> Peg -> Peg -> Peg -> [Move]
```

[2] Adapted from an assignment given in UPenn CIS 552, taught by Benjamin Pierce

Given the number of discs and names for the three pegs, `hanoi` should return a list of moves to be performed to move the stack of discs from the first peg to the second.

Note that a `type` declaration, like `type Peg = String` above, makes a *type synonym*. In this case `Peg` is declared as a synonym for `String`, and the two names `Peg` and `String` can now be used interchangeably. Giving more descriptive names to types in this way can be used to give shorter names to complicated types, or (as here) simply to help with documentation.

Example: `hanoi 2 "a" "b" "c" == [("a","c"), ("a","b"), ("c","b")]`

Exercise 7 (Optional) What if there are four pegs instead of three? That is, the goal is still to move a stack of discs from the first peg to the last peg, without ever placing a larger disc on top of a smaller one, but now there are two extra pegs that can be used as "temporary" storage instead of only one. Write a function similar to `hanoi` which solves this problem in as few moves as possible.

It should be possible to do it in far fewer moves than with three pegs. For example, with three pegs it takes $2^{15} - 1 = 32767$ moves to transfer 15 discs. With four pegs it can be done in 129 moves. (See Exercise 1.17 in Graham, Knuth, and Patashnik, *Concrete Mathematics*, second ed., Addison-Wesley, 1994.)

Note: This exercise is purely for fun – no credit is associated with it. But it *is* fun!