# *CIS 194: Homework 1*

*Due Wednesday, January 28*

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/spring2015/cis194`. We will be using Piazza for Q&A and online discussions.

## *Setup*

To aid you in this first assignment, we have provided a skeleton *HW01.hs*, *HW01Tests.hs*, and *Testing.hs*. Download the files (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 called the Luhn Algorithm for distinguishing valid numbers from random collections of digits (or typing mistakes).

In this section, you will implement the Luhn Algorithm. Pseudocode for the algorithm is provided below:

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

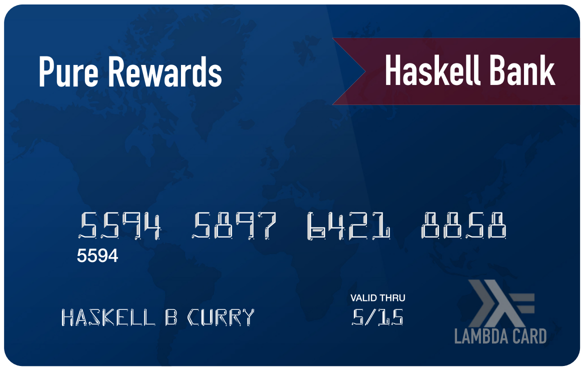

Figure 1: The personal credit card of Haskell Curry who was kind enough to publish his information on the CIS 194 course site. Note, this is for *educational purposes only*. Please do not violate his trust and steal his credit card information!

- 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]`.
- 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`.
- 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.

**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`

Note that some test cases for these functions have been provided in *HW01Tests.hs*. To run the tests for this exercise, load *HW01Tests.hs* in to GHCi and type `runTests ex1Tests`. The result is a list of failures; if the list is empty then all of the tests passed. You should add your own test cases for the remaining exercises.

**Exercise 2** Now, we can break apart a number into its digits. It is actually easier to break a number in to a list of its digits in reverse order (can you figure out why?). Your task is to define the function

```
toRevDigits :: Integer -> [Integer]
```

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

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

*Example*: `toRevDigits 0 == []`

*Example*: `toRevDigits (-17) == []`

It is easy to define a function that gets a list of digits in the proper order in terms of `toRevDigits`:

It is a good exercise to write the function `toDigits` from scratch in order to determine why it is slightly less elegant than `toRevDigits`.

```
toDigits :: Integer -> [Integer]
toDigits n = reverse (toRevDigits n)
```

However, you will likely not need to use it for this homework.

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

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

Remember that the Luhn algorithm should double every other digit *beginning from the right*, that is, the second-to-last, fourth-to-last, . . . numbers are doubled. It's much easier to perform this operation on a list of digits that's in *reverse* order. Conveniently, the function you defined in the previous exercise gives you the digits in reverse order. The function `doubleEveryOther` should thus take in a list that is already in reverse order and double every other number starting with the second one.

*Example*: `doubleEveryOther [4, 9, 5, 5]` = `[4, 18, 5, 10]`

*Example*: `doubleEveryOther [0, 0]` = `[0, 0]`

**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 [10, 5, 18, 4]` = `1 + 0 + 5 + 1 + 8 + 4` = `19`

**Exercise 5** Define the function

```
luhn :: Integer -> Bool
```

that indicates whether an `Integer` could be a valid credit card number. This should use all functions defined in the previous exercises. Don't repeat youself!

Remember that your implementation of `doubleEveryOther` gives you a list in reverse order. Do you need to account for this?

*Example*: `luhn 5594589764218858` = `True`

*Example*: `luhn 1234567898765432` = `False`

If you want more credit card numbers to test on you can get some at `http://www.getcreditcardnumbers.com`. You can test on your own credit cards as well, but you probably don't want to submit test cases with your credit card information.

## *The Towers of Hanoi*[2]

Figure 2: Real life Towers of Hanoii set

**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 3.

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.

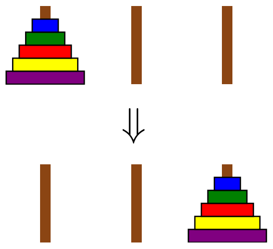

Figure 3: The Towers of Hanoi objective

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.

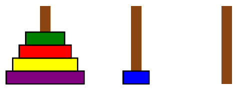

Figure 4: A valid first move.

From this point, it is *illegal* to move to the configuration shown in Figure 5, because you are not allowed to put the green disk on top of the smaller blue one.

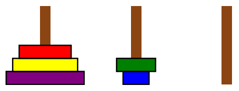

Figure 5: An illegal configuration.

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

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

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

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!