Discrete Math & Functional Programming— CSCI 054— Spring 2024
Instructor: Osborn
Homework 3 (23 point(s))
Due: 10:00PM on Sunday

- For this assignment you should continue working with the same partner(s) that you worked with for week02-ps. Only one of you should turn in the code on Gradescope, but that person should make sure to add everyone else as a collaborator.
- Please make sure you understand the collaboration rules. In particular, a given pair/triple should **never** be looking at any one else's working code or anything functionally equivalent (e.g. listening to them read out their code, looking for samples on whiteboards or online, asking the TA to check against the solutions, etc). If you have any questions, you are expected to check with Professor Osborn beforehand!
- The template file `week03-ps-template.hs` has the types for each of the functions. There is a comment above each that simply names the function. Make sure to augment that comment with a description of what the function does. This is in addition to other comments you might want to include in your file.
- The only list functions that you are allowed to use in your code are: `head`, `tail`, `length`, and `reverse`.

1. **[14 point(s)] Luhn algorithm**

   The Luhn Algorithm is an algorithm for verifying the validity of a credit card. Given a number, the algorithm works as follows:

   - Write the number as a sequence of decimal digits in the usual left-to-right order. Starting at the right, double every second value. (That is, start by doubling the next-to-last digit.)
   - Add the digits of the resulting sequence.
   - Compute the remainder when the above sum is divided by 10.
   - The number is valid if the remainder is 0.

   For example, here is how the algorithm checks the number 13573:

separate the digits	1		3		5		7		3
double every other one	1		6		5		14		3
add the digits	1		6		5		$1+4$		3
compute the total	1	+	6	+	5	+	5	+	3	$=20$

Therefore the number `13573` is valid because 20 is a multiple of 10. Usually, the last (rightmost) digit of an account number is the *check digit*. It is chosen to make the Luhn algorithm work out correctly. In the above example, 1357 is the "real" account number and 3 is the check digit. [1]

For this problem you will write implement a function that verifies if a given number satisfies the Luhn algorithm. We'll break this down into several functions as follows:

(a) Write a function `toDigits :: Integer -> [Integer]` that converts a positive integer to a list of digits. If the given integer is less than 1, the function should return an empty list.

(b) Write a function `toDigitsRev :: Integer -> [Integer]` that does the same thing, but with the digits in reverse order.

(c) Write a function `doubleEveryOther :: [Integer] -> [Integer]` which doubles every other number **starting from the right**. In other words, the last digit is not doubled, but the second-to-last digit is doubled, the third-from-last digit is not doubled, and so on.

(d) The output of `doubleEveryOther` is a list of integers between 0 and 18 (inclusive). Write a function `sumDigits :: [Integer] -> Integer` that returns the sum of all the digits (note that `sumDigits [1, 14]` should evaluate to `6`).

(e) Finally, write a function `validate :: Integer -> Bool` which takes an integer and returns `True` if it's a valid credit card number and `False` otherwise. This function should use most (but possibly not all, depending on how you choose to implement `validate`) of the previous functions.

2. **[9 point(s)] Substitution Ciphers**

Encryption takes *plaintext*, which is a human-readable message that is to be kept secret, and turns it into *ciphertext*, which is the encoded version that is hopefully inaccessible to anyone who does not know how to decrypt it. Often there is an additional piece of information called the *key* that allows someone to decode the *ciphertext* back into its human-readable *plaintext* form.

The Caesar cipher is one of the simplest forms of encryption. We choose a constant shift distance k and replace each letter by its kth successor. The following show the substitutions for a shift of 4.

ABCDEFGHIJKLMNOPQRSTUVWXYZ␣

EFGHIJKLMNOPQRSTUVWXYZ␣ABCD

[1] Security warning: Be cautious with real credit card numbers and avoid typing them into files. There are websites like `http://www.getcreditcardnumbers.com/` from which you can get "valid looking" numbers for testing. Life warning: Do not try to buy anything with a card number from one of these sites.

In encrypting the plaintext, the letter `A` becomes `E`, `B` becomes `F`, and so on. When we reach the end of the alphabet, we wrap around to the beginning. To keep things simple, we use only uppercase letters plus the blank space, denoted ␣. When we encrypt the blank space, we hide the word structure of the message and make it harder to decrypt. With a shift of size 4, `SAGEHEN` becomes `WEKILIR`, and `MEET AT MIDNIGHT` becomes `QIIXDEXDQMHRMKLX`.

Caesar ciphers are easy to decipher because all you need to know is the translation of one letter. That reveals the shift and all the letter translations are known. With our alphabet, including the blank space, there are 27 different shifts. One of them, the zero shift, does not change the message at all and is useless for secrecy. Thus there are 26 possible keys — much too small a number. An adversary could easily try all 26 possibilities and decrypt a message. We'll see something more sophisticated in the next problem.

Naturally we'll implement encrypt (and decrypt). To do this you should implement the following functions:

- `sanitize :: [Char] -> [Char]`: this function converts a list of characters to a new list from which all characters other than blanks and letters have been removed and in which all letters have been shifted to uppercase. For example, `sanitize "I recurse, therefore I am."` should evaluate to `"I RECURSE THEREFORE I AM"`.

  Note: this is a beautiful fit for `map` and `filter`!

- `caesar :: Integer -> [Char] -> [Char]`: this function takes an integer and a string and encodes the string using the shift specified by the integer. For example, `caesar 7 "I recurse, therefore I am."` should evaluate to `"PGYLJAYZLG OLYLMVYLGPGHT"`.

  Note: there are different ways to write this functon. One possibility would be to implement a helper function with a type signature `Int -> Char -> Char` that shifts a single character by the given amount. Then you could use `map` to apply this helper function to the entire message.

  If the integer shift n is greater than 27 then `caesar n` should do the same thing as `caesar (n-27)`. If the integer shift n is less than 0 then `caesar n` should do the same thing as `caesar (n+27)`.