Validation

Sean Leather, adapted by Andres Löh

Difficulty: easy

Use the file `Practicum1.hs` linked from the wiki. All of the functions are "undefined" to start with, allowing you to load the file into GHCi even though you haven't completed all the exercises.

Validating Credit Card Numbers

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 collection of digits (or typing mistakes).

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

- Double the value of every second digit beginning with the rightmost.
- Add the digits of the doubled values and the undoubled digits from the original number.
- Calculate the modulus of the sum divided by 10.

If the result equals 0, then the number is valid. Here is an example of the results of each step on the number 4012888888881881.

- In order to start with the rightmost digit, we produce a reversed list of digits. Then, we double every second digit.

  Result: $[1, 16, 8, 2, 8, 16, 8, 16, 8, 16, 8, 16, 2, 2, 0, 8]$.

- We sum all of the digits of the resulting list above. Note that we must again split the elements of the list into their digits (e.g. 16 becomes $[1, 6]$).

  Result: 90.

- Finally, we calculate the modulus of 90 over 10.

  Result: 0.

Since the final value is 0, we know that the above number is a valid credit card number.
If we make a mistake in typing the credit card number and instead provide 4012888888881891, then the result of the last step is 2, proving that the number is invalid.

1. We need to first find the digits of a number. Define a function

```
toDigitsRev :: Integer → [Integer]
```

that returns a list of positive, decimal (base-10) digits in reverse order. (Recall that we start doubling from the rightmost digit.) You may define toDigitsRev directly or with the function

```
toDigits :: Integer → [Integer]
```

such that toDigitsRev is defined as

```
toDigitsRev = reverse . toDigits
```

Example: The result of toDigitsRev 1234 is [4,3,2,1].
Good programming style: While this may not be necessary for credit card numbers, make toDigitsRev correctly handle inputs that are negative or zero.

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

```
doubleSecond :: (Num a) ⇒ [a] → [a]
```

that doubles every second number in the input list.
Example: The result of doubleSecond [8, 7, 6, 5] is [8, 14, 6, 10].

3. The output of doubleSecond has a mix of one-digit and two-digit numbers. Define a function

```
sumDigits :: [Integer] → Integer
```

to calculate the sum of all digits.
Example: The result of sumDigits [8, 14, 6, 10] is 20.

4. Define the function

```
validate :: Integer → Bool
```

that tells whether any positive input could be a valid credit card number. This will use all functions defined in the previous exercises.
Explain: Why do we use Integer here instead of Int or even Integral a ⇒ a?

Reading and Showing Credit Card Numbers

It's fine to use an Integer for a credit card number internally, but we (as consumers) are accustomed to seeing the number with a certain formatting. In the following exercises, we want to translate between the integer value 4012888888881881 and the string `"4012 8888 8888 1881"`, so that we have a space after every fourth digit from the right. We'll assume all credit card numbers are at most 16 digits. For numbers that have fewer digits, use zeroes to fill out the remaining digits. Thus, 123456789 becomes `"0000 0001 2345 6789"`.

5. Define the function

```
readCC :: String → Integer
```

that parses the number in the format described above. Note that you can use the function read :: (Read a) ⇒ String → a to convert strings to values. Refer to the Prelude documentation for other useful functions.

Explain: How can your function fail?

6. Define the function

```
showCC :: Integer → String
```

that prints the number in the format described above. Note that you can use the function show :: (Show a) ⇒ a → String to convert values to strings. Refer to the Prelude documentation for other useful functions.

Explain: How can your function fail?

Identifying Credit Card Type

Credit cards not only have a formula for validating the digits; they also have formulas for determining the type of card. The type is distinguished by issuer and length:

- A prefix up to six digits long that serves as a unique identification number for the issuer.
- The length can vary within precise limits specific to each prefix.

In this section, you will implement the identification of a credit card number. In order to collect the information for identification, you will need to define I/O operations.

7. Define the function

```
lookupIssuer :: String → Integer → IO String
```

that reads in card type data from a file (whose name is given in the first argument) and returns the issuer of the card number (in the second argument).

A sample file that we'll call `data.txt` appears as follows:

```
34 15 American Express
37 15 American Express
560221 16 Bankcard
6011 16 Discover Card
65 16 Discover Card
51 16 Master Card
52 16 Master Card
4 13 Visa
4 16 Visa
417500 16 Visa Electron
```

Each line contains a prefix, a space, a length (number of digits), a space, and the name of an issuer.

If the number matches a prefix and length, return the third field. If the look-up fails, return the string `"Unknown"`.

Example: The result of lookupIssuer `"data.txt"` 4012888888881881 is `"Visa"`.

8. This exercise uses all of the functions you have written in this practicum. Define the function

```
checkCC :: String → IO ()
```

that takes the name of a file containing card type data. This function provides an interactive program at the terminal and follows these steps:

1. Ask the user for a credit card number in the format used for Ex. 5.
2. Validate the number.
3. Look up the credit card issuer.
4. Print the following:
   - The reformatted number as defined in Ex. 6
   - Status of validation
   - Name of issuer or error message
5. Go back to the first step.

You can end the program by typing Ctrl-C.

Example: This is what a session in this interactive program might look like:

```
ghci> checkCC "data.txt"
Enter credit card number: 0004 2222 2222 2222
The number 0004 2222 2222 2222 is valid and the type is Visa.
Enter credit card number: _
```

In the case of an invalid card, there is no need to look up the type.

```
Enter credit card number: 0004 2222 2222 2223
The number 0004 2222 2222 2223 is not a valid credit card number.
Enter credit card number: _
```

9 (optional)**.** Define a function

$$\mathsf{toDigitsRevG} :: (\mathsf{Integral\ a}) \Rightarrow \mathsf{a} \rightarrow \mathsf{a} \rightarrow [\mathsf{a}]$$

that takes a base and an integer and returns a list of positive digits.

The base (any integer greater than 1) represents the radix of the number. In Ex. 1, we assumed numbers were in base 10 (decimal), but now we allow for other bases, e.g. 2 (binary), 8 (octal), 16 (hexadecimal), and many others that do not have names.

The function toDigitsRevG is more general than toDigitsRev, because it supports other types of integers such as Int. What makes the definition of this function different from toDigitsRev? Describe this in your comments.

In general, we want Haskell functions to be *total*. That is, the function should not produce any errors or incorrect values. How do you make toDigitsRevG a total function? Often, an empty list ($[\,]$) output indicates that the arguments are invalid. Thus, if the function checks for invalid arguments and returns an empty list, it will still be total. Explain how you do this for toDigitsRevG in your comments.