Haskell - Validating Credit Card Numbers

<DISCLAIMER>
For this lab, we provide a template **file which contains the signatures of the functions to be implemented, as well as some definitions that will be needed in order to answer the final question. We suggest you make use of this template when solving the exercises.**
</DISCLAIMER>

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 lab, you will implement a **validation algorithm for credit cards**. The algorithm 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.

First we need to find the digits of a number. Define a function

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

that takes a `n :: Integer` where `n >= 0` and returns a list of the digits of `n`. More precisely, `toDigits` should satisfy the following properties, for all `n :: Integer` where `n >= 0`:

Help

- eval (toDigits n) == n
- `all (\d -> d >= 0 && d < 10) (toDigits n)`
- `length (show n) == length (toDigits n)`

where `eval` is defined as follows:

```
eval xs = foldl (\x y -> y + (10 * x)) 0 xs
```

Note: `toDigits n` should error when n < 0.

Now we need to reverse the digits of a number. Define a function

```
toDigitsRev :: Integer -> [Integer]
```

that takes a `n :: Integer` where `n >= 0` and returns a list of the digits of `n` in reverse order. More precisely, `toDigitsRev` should satisfy the following properties, for all `n :: Integer` where `n >= 0`::

- n == evalRev (toDigitsRev n)
- `all (\d -> d >= 0 && d < 10) (toDigitsRev n)`
- `length (toDigitsRev n) == length (show n)`

where `evalRev` is defined as follows:

```
evalRev xs = foldr (\x y -> x + (10 * y)) 0 xs
```

Note: `toDigitsRev n` should error when n < 0.

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

```
doubleSecond :: [Integer] -> [Integer]
```

that doubles every second number in the input list.

Example: The result of `doubleSecond [8, 7, 6, 5] is [8, 14, 6, 10]`.

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 **individual** digits, even if a number in the list has more than 2 digits.

Example: The result of

```
sumDigits [8,14,6,10] = 8 + (1 + 4) + 6 + (1 + 0) = 20.

sumDigits [3, 9, 4, 15, 8] = 3 + 9 + 4 + (1 + 5) + 8 = 30
```

Define the function

```
isValid :: Integer -> Bool
```

that tells whether any input n :: Integer where n >= 0 could be a valid credit card number, using the algorithm outlined in the introduction.

Hint: make use of the functions defined in the previous exercises.

When you are finished with the above exercises, please proceed to the last page for the final, graded exercises.