Problem Set #1

due September 17

Problem 1. Your task is to implement an algorithm for checking debit and credit card numbers for simple errors. The algorithm works as follows:

(1) Scan the digits from *right to left* doubling every other digit.
(2) Subtract 9 from each number that is now greater than 9.
(3) Add all resulting digits.
(4) If sum is divisible by 10 the card number is valid, otherwise it is invalid.

Implement the following functions:

(1) `toDigits` takes an (arbitrarily large) integer, that is, the card number, and returns a list of its digits in decimal. E.g.,

```
> toDigits 1234567
[1,2,3,4,5,6,7]
```

(2) `doubleOther` takes a list of digits and doubles every other starting from the next to last digit and moving left. E.g.,

```
> doubleOther [1,2,3,4,5,6,7]
[1,4,3,8,5,12,7]

> doubleOther [1,9,3]
[1,18,3]
```

(3) `subNine` takes a list of integers and subtracts 9 from the ones greater than 9. E.g.,

```
> subNine [1,18,3]
[1,9,3]
```

(4) Finally, `validate` takes an (arbitrarily large) integer and returns True if the number is valid. It returns False otherwise. E.g.,

```
> validate 1784
True

> validate 4783
False
```

Problem 2. Your task is to implement an encoding of character strings that works as follows.

(1) Take two character strings of equal length n, where the first string is the secret word and the second one is the "encryption key".
(2) Convert the characters into their corresponding bit strings. E.g., `'a'` becomes `01100001` which is the binary representation of the Ascii value of `'a'`.
(3) Run point-wise XOR over the secret and key bit strings.
(4) Convert back into an n-character Ascii string.

Implement the following functions:

(1) `char2bin` takes a character and returns the list of 8 binary digits that make up the Ascii value of that character. Use the `ord` function from the `Data.Char` library to obtain the ordinal value of a character.

```
> char2bin 'a'
[0,1,1,0,0,0,0,1]
```

(2) `bin2char` takes a list of 8 binary digits and returns the corresponding character. For simplicity, assume that the input is a list of 8 binary digits and use `chr` from the `Data.Char` to convert the corresponding code into a character.

```
> bin2char [0,1,1,0,0,0,1,0]
'b'
```

(3) `encode` takes two strings and returns the cipher string. E.g.,

```
> encode "abc" "789"
"VZZ"
```

The bit string representation of `"abc"` is `01100001 01100010 01100011` and that of `"789"` is `00110111 00111000 00111001`. The point-wise XOR is `01010110 01011010 01011010` which corresponds to string "VZZ".

(4) Finally, `decode` reverses the encoding. It takes two strings, the cipher and the key and returns the secret string. E.g.,

```
> decode "VZZ" "789"
"abc"
```

Notes. Name your file Ps1.hs and begin with

```
{-# OPTIONS_GHC -Wall #-}

module Ps1 where

import Data.Char

toDigits:: ... --- type signature
toDigits ...  --- function definition
...
```

- Submit Ps1.hs in Submitty.
- Do not import modules other than `Data.Char` and stick to just the `ord` and `chr` external functions, everything else should be coming from the standard Prelude.
- Haskell requires explicit conversion from one numerical type to another even in cases when one would expect conversion can be done implicitly. Look up conversion primitives if you need them.

Haskell Style Guide. Adapted from Stephanie Weirich (UPenn CIS 5520).

- **Write a type signature for every function.** We will be strict about this when grading. Hint: try writing the signature *before* writing the function. If you do write the

function first, try deducing the signature and if this doesn't work, there is always the `:t` command.

- Make sure that your code produces no errors or warnings. Code with errors receives 0 on Submitty and we will mark down warnings during TA grading.

- Use consistent indentation.
- Do not use tab characters, use space for indentation. GHC should be flagging tabs, but nevertheless, be careful.
- No line should have more than **80 characters**.
- Use whitespace to make your code readable. Add whitespace on either side of binary operators, e.g., write `3 * n + 1` instead of `3*n+1`.

- Use descriptive names.
- Follow standard Haskell naming conversions: (1) use camelCase for compound names, and (2) use `x` and `xs` when you pattern-match lists.

- Use comments. Each function definition should be preceded by a comment.
- Comment should say what the function does, not how.
- Comments should be concise. Do not overcomment.
- Use full English sentences.

- Do not leave incomplete pattern matches. They will be marked down.
- Tuples, records and datatypes can be decomposed. You can also use the `@` operator if you need a reference to both the object and its components.
  For example, do not use this:

```
f arg1 arg2 = ...  where
   x = fst arg1
   y = snd arg1
   z = fst arg2
```

  Use this instead:

```
f (x,y) (z,_) = ...
```

- Combine nested case expressions.
  For example, do not use this:

```
case x of
      Red -> case y of
                Red  -> True
                Blue -> False
      Blue -> case y of
                Red  -> False
                Blue -> True
```

  Use this instead:

```
case (x,y) of
      (Red,   Red) -> True
      (Blue, Blue) -> True
      (   _,    _) -> False
```

- Use library functions, unless the assignment explicitly forbids them. Use Haskell's search engine Hoogle to look up library functions.