Problem Set #3

due September 24

Note: Use only functions defined in the standard Prelude. Construct readable solutions!

Problem 1. Implement `altMap :: (a -> b) -> (a -> b) -> [a] -> [b]` which takes two argument functions and a list, and applies the argument functions on the list elements in turn in order. E.g.,

```
> altMap (+1) (+100) [0,1,2,3,4]
[1,101,3,103,5]
```

Problem 2. Now implement the *luhn* algorithm from PS#1 for checking credit card numbers in as "wholemeal" style as possible, but most importantly, the solution should be readable.

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

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

Note: Use `iterate`.

(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]
```

Note: Naturally, use `altMap` for full credit.

(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 3. Rewrite the following functions in "wholemeal style".

(1)
```
fun1 :: [Integer] -> Integer
fun1 [] = 1
fun1 (x:xs)
    | even x = (x - 2) * fun1 xs
    | otherwise = fun1 xs
```

(2)
```
fun2 :: Integer -> Integer
fun2 1 = 0
fun2 n
    | even n = n + fun2 (n `div` 2)
    | otherwise = fun2 (3 * n + 1)
```

Name your functions `fun1'` and `fun2'` respectively and use `takeWhile` and `iterate` in `fun2'`.

Problem 4. Use `foldl` or `foldr` to write the `digitsToInt` function that converts a list of digits (integers) into an integer:

```
> digitsToInt [1,2,3,4,5,6]
123456

> digitsToInt $ replicate 30 8
888888888888888888888888888888
```

Problem 5. Write function `transpose::[[a]] -> [[a]]` that transposes its argument. E.g.,

```
> transpose [[1,2,3],[4,5,6]]
[[1,4],[2,5],[3,6]]

> transpose [[1,2],[4,5,6]]
[[1,4],[2,5]]
```

Note: For full credit, use two calls to `foldr`.

Some notes. Name your file Ps3.hs and begin with

```
module Ps3 where

altMap:: ... --- type signature
altMap ...  --- function definition

...
```

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

- **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.