CIS 194: Homework 8

Due Friday, 31 October

- Files you should submit: `HW08.hs`.

Finger exercises

Having just learned about monads and list comprehensions, the next few exercises are an opportunity to show off what you've learned. These are meant to be straightforward.

Exercise 1 Write a function that detects whether or not a string has a certain format. The required format is as follows:

1. The string starts with a digit.
2. Say the value of this digit is n. The string next contains n `a`s.
3. After the n `a`s, either the string ends or the sequence repeats, starting with a (perhaps different) digit.

Here are some strings that match this format and some that don't:

Good strings	Bad strings
`3aaa2aa`	`3aaa2a`
`9aaaaaaaaa`	`10aaaaaaaaaa`
`0`	`1`
`001a`	`100a`
`2aa2aa`	`2bb2bb`

Your function should use the `Maybe` monad. It should look like this:

```
stringFitsFormat :: String -> Bool
stringFitsFormat = isJust . go
  where go :: String -> Maybe String
          -- go evaluates to ‘Just ""‘ on success, or Nothing otherwise
        ...
```

Hint: Use `readMaybe :: Read a => String -> Maybe a` from `Text.Read` and `stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]` from `Data.List`.

Exercise 2

Use a list comprehension to produce the list of all numbers between 1 and 100 (inclusive) that are divisible by 5 but not by 7.

```
specialNumbers :: [Int]
```

Risk

The game of *Risk* involves two or more players, each vying to "conquer the world" by moving armies around a board representing the world and using them to conquer territories. The central mechanic of the game is that of one army attacking another, with dice rolls used to determine the outcome of each battle.

The rules of the game make it complicated to determine the likelihood of possible outcomes. In this assignment, you will write a *simulator* which could be used by Risk players to estimate the probabilities of different outcomes before deciding on a course of action.

The `StdRand` *monad*

Since battles in Risk are determined by rolling dice, your simulator will need some way to access a source of non-determinism, called a random generator. (See HW07 for a discussion of pseudo-randomness.) You will have to get many pseudo-random numbers out of your random generator, and keep track of the way that the random generator changes as you query it for more randomness. This sounds like the perfect time to use a monad! The monad will keep the random generator for you and allow you to sequence queries.

The `MonadRandom` package provides just such a monad. If you don't already have it installed, just say `cabal install MonadRandom` at your command line. You can see the documentation here: `http://hackage.haskell.org/package/MonadRandom-0.3`

You import the gubbins from that package with `import Control.Monad.Random`. You can find the type `Rand` in that package, which defines a randomness monad for any type of random generator `g`. For this homework, we won't need to fiddle with the type of the generator — we'll just use the `StdGen` standard random generator. Thus, put the following line in your file to get rid of this extra parameter, unnecessary for our purposes:

```
type StdRand = Rand StdGen
```

The type `StdRand` is a monad, and it is also a member of the `MonadRandom` class, documented in the `Control.Monad.Random.Class` package.[1] Specifically, you can use the `getRandomR` function when you are operating in the `StdRand` monad. That function is the only monadic function we will be using in this assignment.

Take another look at the documentation for the `Control.Monad.Random` module, which defines various ways to "run" a `Rand` computation; in particular you will eventually (at the very end of the assignment) need to use the `evalRandIO` function.

Allowing flexibility in random generators is a good idea, in general, though. Certain computers have specialized hardware for producing randomness, and might provide a custom random generator. And, Cryptographic programs have very specific requirements for randomness and may use advanced algorithms for producing even better random numbers.

[1] You don't need to import that explicitly, because it's re-exported by `Control.Monad.Random`. You can see this re-export in the documentation for `Control.Monad.Random` by the appearance of `module Control.Monad.Random.Class` at the top. This note means that all names exported by `Control.Monad.Random.Class` are exported by `Control.Monad.Random` as well.

The Rules of Battle

The rules of attacking in Risk are as follows.

- There is an attacking army (containing some number of units) and a defending army (containing some number of units).
- The attacking player may attack with up to three units at a time. However, they must always leave at least one unit behind. That is, if they only have three total units in their army they may only attack with two, and so on.
- The defending player may defend with up to two units (or only one if that is all they have).
- To determine the outcome of a single battle, the attacking and defending players each roll one six-sided die for every unit they have attacking or defending. So the attacking player rolls one, two, or three dice, and the defending player rolls one or two dice.
- The attacking player sorts their dice rolls in descending order. The defending player does the same.
- The dice are then matched up in pairs, starting with the highest roll of each player, then the second-highest.
- For each pair, if the attacking player's roll is higher, then one of the defending player's units die. If there is a tie, or the defending player's roll is higher, then one of the attacking player's units die.

For example, suppose player A has 3 units and player B has 5. A can attack with only 2 units, and B can defend with 2 units. So A rolls 2 dice, and B does the same. Suppose A rolls a 3 and a 5, and B rolls a 4 and a 3. After sorting and pairing up the rolls, we have

A	B
5	4
3	3

A wins the first matchup (5 *vs.* 4), so one of B's units dies. The second matchup is won by B, however (since B wins ties), so one of A's units dies. The end result is that now A has 2 units and B has 4. If A wanted to attack again they would only be able to attack with 1 unit (whereas B would still get to defend with 2—clearly this would give B an advantage because the *higher* of B's two dice rolls will get matched with A's single roll.)

Some types

Include the following type definitions:

```
type Army = Int
data ArmyCounts = ArmyCounts { attackers :: Army, defenders :: Army }
  deriving Show

type DieRoll = Int
```

Exercise 3

Write an action

```
dieRoll :: StdRand DieRoll
```

that simulates rolling a fair, 6-sided die.

(You can test your function in GHCi using `evalRandIO`.)

Exercise 4

Write a function

```
battleResults :: [DieRoll] -> [DieRoll] -> ArmyCounts
```

that takes the attacker's dice rolls and the defender's dice rolls and computes the *change* in the number of armies resulting from the rolls.

Example:

```
battleResults [3,6,4] [5,5] == ArmyCounts { attackers = -1, defenders = -1 }
```

Example:

```
battleResults [3,6,4] [5,6] == ArmyCounts { attackers = -2, defenders = 0 }
```

Example:

```
battleResults [4] [3,2] == ArmyCounts { attackers = 0, defenders = -1 }
```

Hint: This function (and the next) can be much simplified if you write and use a `Monoid` instance for `ArmyCounts`. Monoids are everywhere!

Exercise 5 Write a function

```
battle :: ArmyCounts -> StdRand ArmyCounts
```

which simulates a single battle (as explained above) between two opposing armies. That is, it should simulate randomly rolling the

appropriate number of dice, interpreting the results, and updating the two armies to reflect casualties. You may assume that each player will attack or defend with the maximum number of units they are allowed.

Exercise 6 Of course, usually an attacker does not stop after just a single battle, but attacks repeatedly in an attempt to destroy the entire defending army (and thus take over its territory).

Now implement a function

```
invade :: ArmyCounts -> StdRand ArmyCounts
```

which simulates an entire invasion attempt: that is, repeated calls to `battle` until there are no defenders remaining, or fewer than two attackers.

Exercise 7 Finally, implement a function

```
successProb :: ArmyCounts -> StdRand Double
```

which runs `invade` 1000 times, and uses the results to compute a `Double` between 0 and 1 representing the estimated probability that the attacking army will completely destroy the defending army. For example, if the defending army is destroyed in 300 of the 1000 simulations (but the attacking army is reduced to 1 unit in the other 700), `successProb` should return `0.3`.

You will likely need this function, provided as we haven't talked much about numeric conversions:

```
(//) :: Int -> Int -> Double
a // b = fromIntegral a / fromIntegral b
```