CIS 194: Homework 11
Due Thursday, April 5

- Files you should submit: `Risk.hs`. You should take the version we have provided and add your solutions to it.

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 `Rand StdGen` monad

Since battles in Risk are determined by rolling dice, your simulator will need some way to access a source of randomness. Many languages include standard functions for getting the output of a pseudorandom number generator. For example, in Java one can write

```
Random randGen = new Random();
int dieRoll = 1 + randGen.nextInt(6);
```

to get a random value between 1 and 6 into the variable `dieRoll`. It may seem like we can't do this in Haskell, because the output of `randGen.nextInt(6)` may be different each time it is called—and Haskell functions must always yield the same outputs for the same inputs.

However, if we think about what's going on a bit more carefully, we can see how to successfully model this in Haskell. The Java code first creates a `Random` object called `randGen`. This represents a *pseudorandom number generator*, which remembers a bit of state (a few numbers), and every time something like `nextInt` is called, it uses the state to (deterministically) generate an `Int` and then updates the state according to some (deterministic) algorithm. So the numbers which are generated are not truly random; they are in fact completely deterministic, but computed using an algorithm which generates random-seeming output. As long as we initialize (*seed*) the generator with some truly random data, this is often good enough for purposes such as simulations.

In Haskell we can cerainly have pseudorandom number generator objects. Instead of having methods which mutate them, however, we will have functions that take a generator and return the next pseudorandom value *along with a new generator*. That is, the type signature for `nextInt` would be something like

```
nextInt :: Generator -> (Int, Generator)
```

However, using `nextInt` would quickly get annoying: we have to manually pass around generators everywhere. For example, consider some code to generate three random `Int`s:

```
threeInts :: Generator -> ((Int, Int, Int), Generator)
threeInts g = ((i1, i2, i3), g''')
  where (i1, g')   = nextInt g
        (i2, g'')  = nextInt g'
        (i3, g''') = nextInt g''
```

Ugh! Fortunately, there is a much better way. The `MonadRandom` package[1] defines a *monad* which encapsulates this generator-passing behavior. Using it, `threeInts` can be rewritten as

[1] http://hackage.haskell.org/package/MonadRandom

```
threeInts :: Rand StdGen (Int, Int, Int)
threeInts =
  getRandom >>= \i1 ->
  getRandom >>= \i2 ->
  getRandom >>= \i3 ->
  return (i1,i2,i3)
```

The type signature says that `threeInts` is a computation in the `Rand StdGen` monad which returns a triple of `Int`s. `Rand StdGen` computations implicitly pass along a pseudorandom generator of type `StdGen` (which is defined in the standard Haskell library `System.Random`).

Exercise 1

Type `cabal install MonadRandom` at a command prompt (*not* the `ghci` prompt) to download and install the `MonadRandom` package from Hackage. Then visit the documentation (`http://hackage.haskell.org/package/MonadRandom`). Take a look at 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. Take a look also at the `Control.Monad.Random.Class` module, which defines a `MonadRandom` class containing methods you can use to access the random generator in a `Rand` computation. For example, this is where the `getRandom` function (used above in the `threeInts` example) comes from. However, you probably won't need to use these methods directly in this assignment.

In `Risk.hs` we have provided a type

```
newtype DieValue = DV { unDV :: Int }
```

for representing the result of rolling a six-sided die. We have also provided an instance of `Random` for `DieValue` (allowing it to be used with `MonadRandom`), and a definition

```
die :: Rand StdGen DieValue
die = getRandom
```

which represents the random outcome of rolling a fair six-sided die.

The Rules

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.)

Exercise 2

Given the definitions

```
type Army = Int
```

```
data Battlefield = Battlefield { attackers :: Army, defenders :: Army }
```

(which are also included in Risk.hs), write a function with the type

```
battle :: Battlefield -> Rand StdGen Battlefield
```

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 3

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 :: Battlefield -> Rand StdGen Battlefield
```

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

Exercise 4
   Finally, implement a function

```
successProb :: Battlefield -> Rand StdGen 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`.

Exercise 5   (Optional)
   Write a function

```
exactSuccessProb :: Battlefield -> Double
```

which computes the exact probability of success based on principles of probability, without running any simulations. (This won't give you any particular practice with Haskell; it's just a potentially interesting challenge in probability theory.)