Project #1: War! part 1

due Friday October 9

Submissions

This project will be completed in groups of size 1-3, and submitted in the usual way on our class website. One member of each group should submit the code, and all group members' names should be listed in comments at the beginning of the file.

Overview

This project is the first step in creating a Haskell implementation of the card game War. For some detail about the game and how it works, you can read a lot here: [http://www.pagat.com/war/war.html](http://www.pagat.com/war/war.html).

Our game will be designed in three parts across three projects. The parts roughly are:

1. Internals of the game engine
2. Datatypes for a more sophisticated engine
3. Interactive functions for playing the game

In this project we will do step #1: internals of the game engine. This will involve writing functions which can play the game.

To make things simple, there will always be two players, and our deck of cards will consist only of cards numbered 1 through 9. (Suits are irrelevant in standard War.) Since our cards have no suits and are numbered 1 to 9, it is easiest to store each card as an `Int`. Each player's pile of cards is a list of cards, so should be stored as a `[Int]`. Since there are two players, we always need to keep track of two piles of cards. So most of our functions will have a parameter of type `([Int],[Int])`, which represents the two piles of cards.

Standard rules

In each round the two players each play their top card. Whoever plays the higher card wins the round, and takes both their own and the other player's cards and puts them on the bottom of their own pile.

When the two players play cards of equal rank, this triggers a "war". Each player sets down three cards, and then flips one more card. Whoever wins between these last two flipped cards gets to take all the cards in the round, including the 3 cards that were set down by each player. If those flipped cards match again, another war is triggered, which is resolved in the same way.

Eventually one player will run out of cards, and then the game ends. The one with no cards left is the loser. This can happen at the end of a round, or in the middle of a war when one player doesn't have enough cards to finish the war.

Simple variation

In the simple variant, there is no war. A tie between two cards results in those two cards being permanently removed from the game. The two players continue the game without those two cards.
Russian variation

In the Russian variation, the smallest card (1 in our case), always loses except it wins against the highest card (9 in our case). Otherwise the game works the same way.

Simple Russian variation

This is a combination of the simple and Russian variants. No wars, and 1 beats 9.

Required functions

- `simpleRound` takes a pair of `[Int]` representing the two players’ piles, and returns a pair of `[Int]` which describes the resulting pair of piles after one round using the simple rules (no wars, ties remove the cards).

- `simpleFullGame` takes a pair of `[Int]` representing the two players’ starting piles, and gives a long list of pairs of `[Int]` showing the entire sequence of piles until the game ends, using the simple rules. (For some starting piles the game will never end in that case the result here will be infinite.)

- `simplePlayer1Wins` takes a pair of `[Int]` representing the two players’ starting piles and returns a `Bool` describing whether or not the first player wins the game using the simple rules. (If both players run out of cards at the same time, they both lose.)

- `simplePlayer2Wins` is similar, but tells if player 2 wins.

- `simpleRussianRound` is like `simpleRound`, but using the simple Russian rules.

- `simpleRussianFullGame` is like `simpleFullGame`, but using the simple Russian rules.

- `standardRound` is like `simpleRound`, (with the same type signature) but uses the standard rules rather than the simple ones.
  (This one is hard because during a war there is a “pot” of cards that will all go to the winner of the next matchup. I suggest making a helper function which takes one additional `[Int]` parameter representing the pot.)

- `standardFullGame` is like `simpleFullGame`, but with the standard rules rather than the simple ones.

- `russianRound` is like `standardRound`, but using the Russian rules.

- `russianFullGame` is like `standardFullGame`, but using the Russian rules.

In all of these functions you may decide however you like about the precise order that won cards are placed on the bottom of people’s piles.