Project #2: War! part 2

due Friday November 20

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 #2: datatypes for a more sophisticated engine. This time the cards will not be represented by the `Int` type, but by custom types. The cards will have “suits” and “ranks” (though the suits are usually irrelevant).

In the previous project, even if you wrote your functions as nicely as possible, you probably ended up writing big chunks of your definitions several times in different places (for example the definitions of `standardRound` and `russianRound` are almost but not quite the same).

This time we will create higher order functions so that there will be almost no duplication of code, and creating new game variations will require only a few new lines of code.

Cards, suits, and ranks

Each card has a suit and a rank. The suit can be one of: Spade, Heart, Diamond, or Club. The rank is either an integer 2 through 10, or it is one of the special ranks Jack, Queen, King, Ace. The usual ordering of ranks from lowest to highest is 2, 3, ..., 10, Jack, Queen, King, Ace.

In the standard variant of war, the suits of the cards are completely ignored. The ranks of the two cards are compared, and the card with the higher rank is the winner. If the ranks are equal, a war is triggered.

In the Russian variant, the game is the same except for the exceptional rule that a card with rank 2 will beat a card with rank Ace. Still the suits are ignored.

A new game variant: Spades trump

In this variant the suits matter a bit. Spades (♠) will always beat cards from any other suit regardless of ranks. Thus 4♠ will beat J♣. In a round between two spades, the higher ranked card will win as usual.

(Because of this rule, there can be no war in a round where one of the cards is a spade. The ace of spades is undefeatable, but it can change hands during a war.)
Different sets of rules: Comparison and round

All of our game variations can be classified as one of two changes to basic rules. There are “comparison rules”, which tell us which cards beat other cards, and there are “round rules” which say what to do in each round and how to handle a tie.

Comparison rules express things like the standard ordering for the cards, or the russian variation in which 2 beats Ace, or the spades trump rules in which spades are special. The “round rules” tell us what to do based on which card wins the round, or what to do in the case of a tie.

We’re going to express these types of rules as higher order functions. A comparison rule should take two cards and return a \( \text{Bool} \) saying whether or not the first card wins. So we don’t have to continually refer to this type, make a type synonym:

\[
\text{type ComparisonRule} = \text{Card} \to \text{Card} \to \text{Bool}
\]

Round rules function like our various “round” functions from Project 1. They say what should happen in each round based on comparing the values of the cards. In Project 1 there were two variations: “war” or “no war” (the “simple” rules). These rules should functions which take a pair of lists of cards and return another pair of lists of cards. Thus we can express these using this type synonym:

\[
\text{type RoundRule} = ([\text{Card}], [\text{Card}]) \to ([\text{Card}], [\text{Card}])
\]

Required functions and types

- Define three types called Suit, Rank, and Card. A suit is simply one of the four suits. A rank is either a number 2 through 10, or one of the 4 special ranks. A card consists of a rank together with a suit.

- Put your Card type in the Show class so that it displays nicely. Like \([4 \spadesuit, A \heartsuit, J \diamondsuit]\) might show as something like:

  \[4 \text{ of Clubs}, \text{Ace of Diamonds}, \text{Jack of Hearts}\]

- Define at least one test pair of type \([\text{Card}, \text{Card}]\) called testPair that you (and I) can use to test your functions.

- Put your Rank type in the Ord and Enum classes.
  
  Suggestion: Use your enum functions to define the ordering.

- Define deck :: [Card] which is a list of all the cards in the standard 52-card deck.
  
  Suggestion: use a two-variable list comprehension where the ranges look something like:

  \[s \leftarrow [\spadesuit, \heartsuit, \diamondsuit, \clubsuit], r \leftarrow [2..\text{Ace}]\]

  You’ll need to change some things slightly in my suggestion to make it work with your types.

- Put these type synonyms in your code:

  \[
  \text{type ComparisonRule} = \text{Card} \to \text{Card} \to \text{Bool}
  \]

  \[
  \text{type RoundRule} = ([\text{Card}], [\text{Card}]) \to ([\text{Card}], [\text{Card}])
  \]

- Define three comparison rules called: standardComparison, russianComparison, spadeTrumpComparison. The result of each of these should be \( \text{True} \) if the first card wins, and \( \text{False} \) otherwise (including ties).
  
  Suggestion: you can use one of your Enum functions to make some of the comparisons easier.

- Define two functions called: roundWithoutWar and roundWithWar with type \( \text{ComparisonRule} \to \text{RoundRule} \) which take a comparison rule and use it to create a round rule, either with or without wars.

  Hint: The type \( \text{RoundRule} \) is the same as \( ([\text{Card}], [\text{Card}]) \to ([\text{Card}], [\text{Card}]) \), so these functions' types are \( \text{ComparisonRule} \to ([\text{Card}], [\text{Card}]) \to ([\text{Card}], [\text{Card}]) \), so write them as functions of two parameters, the first being a \( \text{ComparisonRule} \).
If you do this properly, you should be able to make a very short definition of one of your Project 1 functions:

\[
\text{simpleRound} = \text{roundWithoutWar standardComparison}
\]

- Give similar very short definitions for the Project 1 functions \text{standardRound}, \text{russianRound}, \text{simpleRound}, and \text{simpleRussianRound}.

- Define a function \text{fullGame} which takes a function of type \text{RoundRule} and creates a function that plays the full game by repeating the given round function.

  Using this, you can make definitions like:

  \[
  \text{simpleFullGame} = \text{fullGame simpleRound}
  \]

- Give similar very short definitions for the Project 1 functions \text{standardFullGame}, \text{russianFullGame}, \text{simpleFullGame}, and \text{simpleRussianFullGame}.

- Use your new functions to make very short definitions for \text{spadeTrumpRound} and \text{spadeTrumpFullGame} (have these ones use wars).

- Imagine a new rule variation, and create new round and fullgame functions for your variation. Call your functions \text{myRound} and \text{myFullGame}. Your variant can be as simple or as complicated as you like, and can involve a new comparison rule, or a new round rule, or both, or a change in some other way. Include a comment in your code that explains to me exactly how your variation is supposed to work.