

Poker

Alex Elyasov, Jan Rochel

deadline: before October 17

The purpose of this assignment is to implement a console version of the poker card game with a basic AI player. As usual you can refer to <http://en.wikipedia.org/wiki/Poker> for details.

1 Introduction

It is time to become a “bad guy”: learn how to play poker and write a bit more Haskell code. For those who are unfamiliar with this well-known game, we will give all necessary information during the assignment formulation. From the many variations of poker, we chose the *Five-Card Draw* variant for this exercise. Logically, the assignment can be split up into the following modules: Cards, Deck, Combo, AI and Main. Use this partitioning in your implementation.

2 Cards

1. A poker deck consists of 52 cards (see Figure 1), with the ranks 1 to 10, plus Jack, Queen, King and Ace, where for every rank there are four cards belonging to the suites Spades, Clubs, Diamonds, and Hearts, respectively.

In this exercise you need to define a data type representing the cards. There are many ways to do this either using auxiliary data types, or using constructors with parameters.

```
data Card ...
```

2. Haskell provides a mechanism for automatic instances derivation which you can use by adding a `deriving` clause to your data type declaration. A `deriving` clause is parametrized with the class names you intend the data type to be an instance of. The `deriving` mechanism is provided only for a limited set of classes. In that case you need to provide an explicit instance declaration.





















































Suit	Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King
Clubs													
Diamonds													
Hearts													
Spades													

Figure 1: List of poker's cards

The next step is to make the Card data type an instance of the Show type class, such that it can be displayed with the show function in a nice way.

```
instance Show Card where show = ...
```

So if card1 and card2 are two variables type of Card, with the corresponding values "Six Hearts" and "Ace Spades" they should be printed in the following way:

```
*Main> card1
6H
*Main> card2
AS
```

3. Also you need to derive instances for the following type classes Eq, Ord and Enum from data type Card. You can do this using automatic deriving, but be aware that the automatically derived order corresponds to the order in which the constructors occur in the data type definition. Make sure that the order corresponds to the relative strength of the cards (we do not require an order for the suites):

```
1 < ... < 10 < Jack < Queen < King < Ace
```

3 Deck

The module Deck should import the module Cards. The aim of this module is to implement the functionality which is responsible for generating full deck, shuffling it and dealing cards to the players.

Deck is represented as:

```
type Deck = [Card]
```

4. Define a function

```
fullDeck :: Deck
```

which generates the full deck (52 cards).

5. One way to shuffle the deck is to generate two random numbers within the range $[0, 51]$ and then swap the cards on the positions corresponding to these numbers. Repeating this process k -times we can get a relatively good shuffled deck.

The next function is responsible for the deck shuffling:

```
shuffleDeck :: Deck -> IO Deck
```

You can use the following function to generate a random number in the interval $[0, 51]$:

```
genPos :: IO Int
genPos = getStdRandom (randomR (0, 51))
```

You also need to import `System.Random` from the Haskell standard libraries.

6. The last function in this module deals N cards to the player from the deck and returns the updated deck.

```
dealNCards :: Int -> Deck -> ([Card], Deck)
```

4 Poker Combinations

7. A complete *hand* (5 cards) is dealt to each player. Define a type synonym or data type `Hand`. Each player has the aim of collecting the best possible combination of cards (Fig. 2), as some hands are stronger than others. A 'straight flush' for example is much stronger than 'two pairs'.

In this exercise you need to define a data type representing the possible categories of poker hands:

```
data Combo ...
```

You should take into account, for instance, that a pair of sixes is weaker than a pair of Jacks or that a straight starting from six is stronger than a straight starting from five. Refer to http://en.wikipedia.org/wiki/Poker_hands for the exact rules. Make the data type an instance of `Show` and `Ord`, where the ordering should correspond to the strength of the combo.

8 (difficult). For every combination you need to provide the corresponding recognizer, which is a function of the type `Hand -> Maybe Combo`. For instance, the function `checkFlush`

```
checkFlush :: Hand -> Maybe Combo
```

checks if the given hand is a Flush, and if it is not returns `Nothing`.

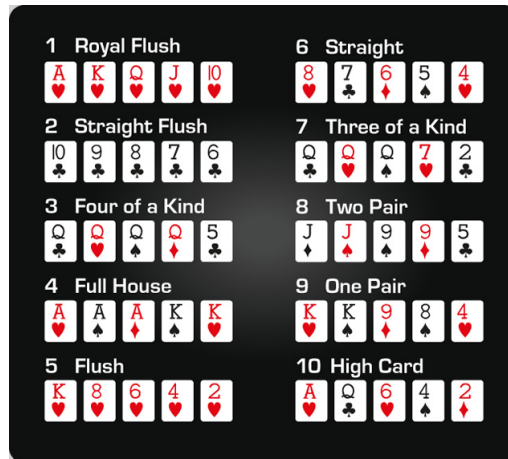


Figure 2: List of poker's hands

9. Because of the ordering of all poker combinations we can sequentially check whether the hand is Straight Flush, Four of a Kind, and so on. That is, the first successful comparison gives you the highest combination the hand constitutes. You need to write a function `bestCombo` which implements the above-mentioned scheme.

```
bestCombo :: Hand -> Combo
```

Hint: It makes things easier if you ensure that a Hand is always sorted.

10. The game can have the result one of the following types: you win, you lose or draw. Using a data type representing the result you need to define a function `whoWin`, which is comparing two hands and return the result.

```
data Result ...
whoWin :: Hand -> Hand -> Result
```

5 AI

11 (difficult). Now we are going to implement an AI player that decides on which cards of a hand to exchange for new cards. The aim is to maximise the probability to win against another player. We are going to exploit the ability of the computer to explore large combinatorial spaces.

Hint: Use `undefined` to leave gaps during the implementation. Also use `undefined` if you are not able to come up with a working implementation of a function. That way you can still submit a partial, type-correct solution. If you want to skip forward to the easier exercise of implementing the main function you can define `pickSubst = const []` for now.

The overall goal is to define a selection function that picks the cards to discard from the hand.

```
pickSubst :: Hand -> [Card]
```

The poker variant you are to implement it is not permitted to exchange more than three cards. Thus, we first compute the set of all subsets of the given hand of length not greater than three. They are the candidates we consider for substitution.

```
candidates :: [[Card]]
```

Note, that this function does not have a `Hand` as a parameter, from which you can conclude that it must be a local function of `pickSubst`. Always prudently consider whether you want to define a function top-level or locally inside of another function.

`candidates` should be defined in terms of

```
sublists :: Int -> [a] -> [[a]]
```

The result of `sublists 3 [1,2,3,4]` should be `[[1,2,3], [1,2,4], [1,3,4], [2,3,4]]`. Reason carefully about what the result should be in case 0 is used as a first parameter. You can verify the function by asserting that there are indeed 2,598,960 different poker hands.

Our choice will be based on how strong the hands are that might result from substituting the selected cards by the same number of cards from the *remaining* deck.

```
possibleHandsAfterReplacing :: [Card] -> [Hand]
```

Define `possibleHandsAfterReplacing` in terms of `sublists`, a function that generates all sublists of a specific length.

While the `Ord` relation on `Combo` is sufficient to compare the strength of two hands, it doesn't immediately give us the means to compare two sets of hands. But if we could assign a numeric weight for the strength of a hand then we could compare the average weight of the two sets. So, what are we going to use as a weight? Since we want to maximise the chances of winning against the opponent let us simply take the probability of beating the opponent with a given hand.

```
winProb :: Hand -> Float
```

A game is won if the enemy has a hand worse than our own. Therefore we need to compute the probabilities of the enemy having a hand worse than a given combo. Define a mapping from combos to the probability that the opponent will have no stronger hand than a given combo.

```
oppNotStrongerThan :: Map Combo Float
```

The `Map` data type is defined in the `Data.Map` module from the `containers` package.¹ The probability that the opponent has no stronger hand than a given combo is the sum of all the probabilities of him having a weaker (or equally strong) combo.

¹Installation via `cabal update`; `cabal install containers` from the terminal

Define `oppNotStrongerThan` in terms of a mapping from combos to their probabilities of occurring in the opponents hand

```
comboProbs :: Map Combo Float
```

which in turn should make use of the list of all hands the enemy could have.

```
possibleHands :: [Hand]
```

Hints:

- Be clever and look for useful functions in the Haskell standard libraries (<http://www.haskell.org/hoogle/>).
- To avoid waiting for minutes while testing the AI you can have it operate on a small subset of the poker deck (say 20 cards) to avoid combinatory explosion.
- Haskell programs run much faster when compiled with GHC to machine code than interpreted with GHCi. Compile via `ghc -rtsopts Main` and execute the resulting executable with `./Main +RTS -K999999999`. The options ensure that the runtime system has enough stack space for the complex computation.

Bonus: Make the AI run in feasible time. Under one minute is possible without any neat tricks or parallelisation. Ask the assistants how to profile your implementation.

6 Main

This is the main module for the poker. The module is responsible for game initialization, and user interaction functionality.

12. An execution of a program starts with the function `main` in the `Main` module. So this function should print a welcoming message for the user and ask whether he wants to continue the game. That is you need to create a loop in which the game will be executed until you stop it with the answer "no". The program should not be sensitive to the register, it means that all possible answers: "No", "no", "NO", "nO" (the same for the answer "yes") should be recognized as equivalent.

```
main :: IO ()
main = do ...
    if (cond) -- check the quit condition
    then do  -- play the game
        game
        main
    else    -- quit
```

13. A function `game` represents the sequence of actions, which the program needs to execute during one poker game. Use the functions from the exercises below, in order to implement the logic described in the comments to the function `game`.

```
game :: IO ()
game = do
    -- shuffle deck, for instance, 1000 times
    -- deal 5 cards (hand) to the player
    -- deal 5 cards (hand) to the computer
    -- show player's hand
    -- ask player which cards he wants to discard (max. 3)
    -- deal new cards instead of discarded
    -- run exchanging AI for computer's hand (this step is optional)
    -- check whose combination is better
```

14. A function `showHand` prints a hand as the sequence of cards separated by spaces.

```
showHand :: Hand -> IO ()
```

15. Write a function which requests the cards you want to exchange.

```
askCardsForExchange :: IO [Card]
```

For example, if your hand consists of the cards `[AH,KH,QH,2D,3C]` and you want to exchange 2D and 3C, your input should be the following string `"2D 3C"`. But the resulting value from `IO` in that case will be the list `[c1,c2]`, where `c1` and `c2` are the values of the type `Card`.

Hint: Implement an auxiliary function `readCards :: String -> [Cards]`

16. The function below accepts a hand, list of the cards you want to exchange and deck, and it returns the pair with the updated hand and the deck. The new hand was obtained from the original hand by the substitution of fresh generated cards from the current deck instead of exchanged cards.

```
exchangeCards :: Hand -> [Card] -> Deck -> (Hand, Deck)
```

17. The last function is responsible for making decision who win, and printing the result.

```
checkResult :: Hand -> Hand -> IO ()
```