

# CS96si Assignment 2a: Haskell Poker Scorer

Joseph Victor

May 14, 2012

## Introduction

The goal of this assignment is to write Haskell functions that determine if a list of poker cards satisfy a given hand. The assignment should be, in terms of questions like “where do I start?”, very straightforward; the actual code you need to write consists only of one function for each hand (plus any auxillary functions you may need). You will write notoriously easy/gross functions for checking poker hands, but I think you’ll find that obliterating them functionally is quite fun. Before getting started, take a look at the links for getting Haskell on the website. Now, without further ado, lets get started.

## pokerdata.hs

The file pokerdata.hs contains definitions for the following data types: Rank, Suit, Card, Hand and ScoredHand, and some basic functionality for them. Go ahead, play with them. Type “Heart”, or “Ten” (without the quotes) into the ghci, and see the Suit and Rank types at work. Type “:t Heart” and see that Haskell knows about the type. Type “Eight < King”, to see that the typeclass Ord is working, and type “[Two .. Jack]” to see that the typeclass enum is working. Play around, get comfy, figure out how to use your types. Notice that Ace is high, that is “Ace > Two” returns True.

Next, type “Card Two Heart” to create a value of type Card. Again, try things like “:t Card Ace Club” or “(Card Jack Diamond) > (Card Ten Heart)”. Try saying “[Card Two Club .. Card Ace Diamond]” to get the whole deck. Understand these calls, they are easy, but important tools for writing simple hand checkers.

The next thing you will notice in pokerdata.hs is a data declaration called ScoredHand. An element of type ScoredHand is a category of hand (for

instance, Flush or FullHouse or TwoPair) and some additional data from which a tie can be broken. For instance, given the hand

4 Heart, 5 Heart, 6 Diamond, 7 Club, 8 Spade

We could say, hey, thats a straight, so we would associate with that the ScoredHand value

Straight Eight

since to reconstruct a straight you need only the highest rank (you don't care about Suits).

If you forget the rules for hands in Poker, just remind yourself on Wikipedia, or send me an email, or ask someone in your house.

Your job, then, will be to try to associate ScoredHands with sorted lists of cards, or report that the hand does not satisfy the category.

## The Utils.hs File

This is where you can put any general purpose utility functions you may or may not need. Write em as they come up.

I put some functions for writing the test benches here. You don't really need to understand what's already there, but they aren't hard.

## The Tofu (or if you're not a vegetarian, The Meat)

For each hand category, there is an unwritten function in poker.hs, waiting for you to write it. The functions all have the same type

[Card] -> Maybe ScoredHand

Recall that for any type a, Maybe a is a type whos values are either (Just x) or Nothing, where x is some value of type a. So this means a Maybe ScoredHand is either a Just ScoredHand or the symbol Nothing. These functions are supposed to return some Just value if the hand satisfies the conditions, and Nothing otherwise. For instance, given the hand from before

4 Heart, 5 Heart, 6 Diamond, 7 Club, 8 Spade

if you call `flush` on it, you should return `Nothing`, but if you call `straight` on it, you should return

Just (Straight Eight)

What can you assume about the `hand` argument? You may assume that the argument has exactly 5 cards, that it is sorted low-to-high, that all the cards are unique, and that the hand does not satisfy any higher ranked category. For instance, if you are checking to see if something is a 3-of-a-kind, you do NOT have to check that there are exactly three, (obviously you do need to check that there are at least 3), since you can assume it does not satisfy 4-of-a-kind.

So the behavior of these functions you have to write is simple: check to see if the input hand satisfies the category, and if it does return `Just` what it is and if it doesn't return `Nothing`.

`highCard` has been done for you.

Once all these have been written, you are done. There is a function written for you called `scoreHand`, which takes a `Hand` (remember, a `Hand` is a structure with exactly 5 unsorted `Card` values in it), converts that `Hand` to a `[Card]` and calls the functions you wrote on it, in order, to score the hand. If you call `scoreHand` before writing these functions, it may yell at you.

## Strategies and Tactics

How to write this? We have a list, and we want to use the high order functions we have to figure out what it is. Think about what you want to do. Want to get rid of all the cards which don't satisfy something? Try `filter`. Want to modify each `Card` in a uniform way. Try `map`. Want to make sure all the `Cards` satisfy a certain predicate. Consider using a filter on a negated predicate. If the whole list satisfies the predicate, then nothing satisfies the negated predicate, so the result will be `[]`. To see if everything in the hand is a `Heart`, for instance, you could write this.

```
allHearts :: [Card] -> Bool
allHearts cards = [] == (filter (\(Card r s) -> s /= Heart) cards)
```

Remember that lambdas can do pattern matching in Haskell, so if you are making a lambda which takes a card object as its argument, if you are going to need the `Rank` and `Suit` of the card you may as well take the argument as a pattern.

As a side note, if you like that idea, you could define

```
all :: (a -> Bool) -> [a] -> Bool
all pred lst = [] == (filter (not.pred) lst)

allHearts :: [Card] -> Bool
allHearts hand = all (\(Card r s) -> s == Heart) hand
```

The laziness will take care of the fact that we don't want the rest of the list looked at once we've found one thing which satisfies `f`. Actually, though, `all` is part of the standard prelude, so you can just use it, but the idea of defining your own high order functions is a good one.

Write helper functions. The `n-of-a-kind` functions, `full house` and `two pair` are quite similar. Try to reuse code, just like with anything.

Another strategy might be this: think about how you might implement it in Java, and then start getting rid of loops one at a time by turning them into function calls. Pretty soon you will find this step is no longer required, but the idiom is different enough that it might help. Remember, you can always store “local variables” in a `where` statement, and `where` does pattern matching.

## Testing

Think it works? Well, I wrote a cute test bench, so it should be pretty clear if it does or not. Just load up `Main.hs`, and either type “`main`” into `ghci` or, in a terminal, type “`ghc -make Main.hs`”, or compile the project in whatever way you'd like, and run it, probably with “`./Main`”. This will run the testing script, which basically tries your code on a metric crap-ton of random hands. It will tell you if you failed and why, or yell at you if something is not yet implemented. (There is some finite chance the test benches are wrong, although they agreed with my implementation of the assignment, so if they are wrong it might be because I don't understand poker, in which case yell at me).

## Reminders, PAQ, More Advice, Etc

Types mean everything. Try to load things into `ghci` as often as possible. If something is wrong, Haskell will alert you. If there is one thing Haskell is good at, its yelling at you if you mess up the types.

If you have something of type `Card` and you need something of type `Suit`, use pattern matching. Remember that `let`, `where`, `lambda` (the `\` thing is supposed to be `lambda`, `λ`) and `case` all do pattern matching, so you can

extract data like that. The same thing goes if you have something of type Maybe Int and you want an Int (or possibly Nothing).

The examples from class are all online, or refer to Learn You A Haskell For Great Good.

You have to write 9 functions. You have 9 days to do the assignment. Now, they won't all take the same amount of time, in fact the first few will be hardest, but you can imagine writing one a day. So just do that. Start early, and give yourself time to have fun with it.

## Deliverables

I need the code from poker.hs and Utils.hs if you modify it, as well as a brief write-up saying what you learned, emailed to me by the due date.

I will be somewhat stricter about the due date this time. Or am I bluffing? This is poker, after all... In all seriousness, do try to start this before the due date please.

I want to offer as much early help as possible. I will be holding a sort of YEAH hours on Tuesday May 15, as well as OH on Monday May 21 and Tuesday May 22 in Synergy from 8 till 10.

Good luck, and enjoy.