CCPS 506 - Assignment

For this assignment, you will implement the same general functionality using **two** of the four languages we've studied this semester. At least one of the languages chosen **MUST** be functional (Elixir or Haskell). In addition to the general requirements, each language comes with its own set of language-specific constraints which can be seen below.

Both programs are due by the end of Wednesday, June 20, 11:59pm

General assignment description:

For this assignment, you will implement functions/methods for evaluating two poker hands and choosing a winner. A description of different hands and their ranking can be seen here:

https://www.cardplayer.com/rules-of-poker/hand-rankings

The way each hand is represented will differ based on which language you're using. If you would like to represent the hands differently, please e-mail me and make a request. I will consider it so long as it doesn't significantly reduce the difficulty of the assignment. You may assume that the hands, when passed into your function, will be formed correctly. That is, you don't have to worry about error checking the data type, the number of cards in each hand, etc.

The rank of each card will be integer. To keep it simple, ace will have a rank of 14. Thus, ranks go from 2 to 14 (11 = Jack, 12 = Queen, 13 = King, 14 = Ace). This way, higher rank is always better.

The suit of each card will be String: "H" = hearts, "D" = diamonds, "S" = spades, "C" = clubs.

Assume that one hand will always clearly win. Assume that if two hands would be tied in poker, this input would not occur when testing your code. There are tie breaking mechanisms in the actual game of poker - for example, if two hands have the same high card, we check the second highest card, etc. You are **not** required to consider these cases.

Finally, indicate, in some fashion, which hand wins. It can be as simple as returning true if first hand wins, or false otherwise. You can also print a message saying which hand won, etc.

Submission

Submit all relevant source code to D2L. In the case of Smalltalk, copy all your code into a text file. For Elixir submit your .ex file, for Haskell submit your .hs file, etc.

Additionally, submit a log of your program's execution on several test runs. Test a variety of different hands, and prove that your code works. If your code doesn't work for certain types of hand – that's OK. Just show me tests where the program does work. Indicate which types of hand it works for, and which types it doesn't.

1) **Smalltalk requirements:**

Represent each hand as a heterogeneous array. Create a custom class called Poker, with a method that accepts two arrays as arguments called **`winner:vs:`** (refer to lab 1 if you forget how to do this). Calling your method might look as follows:

```
x := Poker new.
hand1 := #(3, 'H', 10, 'S', 4, 'S', 4, 'C', 5, 'C').
hand2 := #(2, 'H', 2, 'S', 5, 'S', 2, 'C', 13, 'C').
x winner: hand1 vs: hand2.
```

2) **Elixir requirements:**

Represent each hand as a heterogeneous list. Name your function for finding the winning hand "**`winner`**", and call it as follows:

```
hand1 = [3, "H", 10, "S", 4, "S", 4, "C", 5, "C"]
hand2 = [2, "H", 2, "S", 5, "S", 2, "C", 13, "C"]
Poker.winner hand1, hand2
```

Your **`winner`** function and all associated helper functions should be defined in a module called **Poker**, in a file Poker.ex. I should be able to execute my own script that calls the winner function in your Poker module.

3) **Haskell requirements:**

Represent each hand as a list of pair tuples. Name your function for finding the winning hand "**`winner`**", and call it as follows:

```
hand1 = [(3, "H"), (10, "S"), (4, "S"), (4, "C"), (5, "C")]
hand2 = [(2, "H"), (2, "S"), (5, "S"), (2, "C"), (13, "C")]
winner hand1 hand2
```

Your **`winner`** function and all associated helper functions should be defined in a module called Poker, in a file Poker.hs. I should be able to load the Poker module into GHCi and call your winner function.

4) **Rust requirements:**

Represent each hand as an array of pair tuples. Name your function for finding the winning hand "**`find_winner`**", and call it as follows:

```
let hand1 = [(3, 'H'), (10, 'S'), (4, 'S'), (4, 'C'), (5, 'C')];
let hand2 = [(2, 'H'), (2, 'S'), (5, 'S'), (2, 'C'), (13, 'C')];
let xxx = find_winner (&hand1, &hand2);
```

Call **`find_winner`** from your main function, and print the winning hand in the main function. The hands themselves may be hardcoded. I will change them and recompile when testing.

Hints and Strategy:

- Modularity will keep everything manageable. Break the problem into small parts.
    - Write a function to find the highest card in a hand.
    - Write a function to determine if there's a pair.
    - Write a function to determine if there's two pairs,
    - Etc.
- Consider the hand categories on the provided website.
    - The strength of each hand can be represented as a tuple
    - The first element can be the category of the hand
        - Pair would be category 9, for example
    - The second element can be the ***strength*** within that category
        - A pair of 10s would be (9, 10) for example.
        - (9, 10) would beat (9, 4)
        - Same category (pair) but a pair of 10s beats a pair of 4s
- Start by implementing the logic for finding the highest card in each hand, and declaring the high card hand the winner. From here, add logic to check for pairs, etc. Build up to the more complicated hands (full house, etc.).
    - I can't stress this enough – get the programs working under simple cases **first**.
- Don't be afraid to Google things! We didn't see everything in class. "Haskell largest item in list" for example, might yield useful advice.