Poker Hand

by ToF

November 12, 2011

$\lambda \quad \lambda \quad \lambda$

1 Solving the problem

Did we solve our problem?

1 Not yet.

What do we need to do, then?

2 Mark the lines from the input with the ranking of the hand, and suffix the best one with "(winner)".

What test should I write?

3 Write the simplest test you can think of.

What is the trivial case for a function that should mark lines?

4 No hand at all.

Ok

```
markResults [Nothing] ~?= [""]
```

If we don't have a hand, then there is no mark.

5 ■ I see. Here's the function:

```
markResults :: [Maybe Ranking] → [String]
markResults _ = [""]
```

■ Done.

Here's my next case.

```
markResults [Nothing, Just Pair]
            ~?= ["", "Pair  (winner)"]
```

6 ■ Ok. I'll just add a pattern:

```
markResults :: [Maybe Ranking] → [String]
markResults [Nothing]   = [""]
markResults [Nothing, Just Pair] = ["", "Pair  (winner)"]
```

■ It's a *fake*, as usual.

Do you see a possible refactoring here?

7 I see a map:

```
markResults :: [Maybe Ranking] → [String]
markResults = map mark
    where mark Nothing = ""
          mark (Just Pair) = "Pair  (winner)"
```

■ Refactoring done.

Here's a new case:

```
markResults [Nothing, Just Pair, Just HighCard] ~?=
                    ["","Pair (winner)","High Card"]
```

■ We can have several hands. The best one is the winner. There's *non-exaustive patterns* error in our code, now.

8 ■ Sure. Here's a fix:

```
markResults :: [Maybe Ranking] → [String]
markResults = map mark
    where mark Nothing = ""
          mark (Just Pair) = "Pair (winner)"
          mark (Just r) = show r
```

■ It's still a *fake*.

How can we remove the *fake*?

9 By comparing each value in the list with the maximum value in the list.

```
markResults :: [Maybe Ranking] → [String]
markResults rs = map mark rs
    where mark Nothing = ""
          mark v@(Just r)
              | v == maximum rs = show r ++ " (winner)"
          mark (Just r) = show r
```

■ It works!

Can you remove duplication?

10 Yes.

```
markResults :: [Maybe Ranking] → [String]
markResults rs = map mark rs
    where mark Nothing = ""
          mark v@(Just r) = (show r) ++ if (v ==
                maximum rs) then " (winner)" else ""
```

■ Done.

Could we have pattern in lieu of the `if then else`?

11 Yes.

```
markResults :: [Maybe Ranking] → [String]
markResults rs = map mark rs
    where mark Nothing = ""
          mark (Just r) = (show r) ++ winner (Just r)
          winner v | v == maximum rs = " (winner)"
          winner _ = ""
```

■ Done.

And we could avoid computing the `maximum` at each line.

12 You are right.

```
markResults :: [Maybe Ranking] → [String]
markResults rs = map mark rs
    where mark Nothing = ""
          mark (Just r) = (show r) ++ winner (Just r)
          winner v | v == m = " (winner)"
          winner _ = ""
          m = maximum rs
```

■ And now we are done with marking results.

What else is missing?

13 Our program will have to reproduce and complete the input lines.

Ok. Here a test:

```
scores ["6♥ 6♦ 6♠ 6♣",
        "6♣ 4♦ A♣ 3♠ K♠ 5♦ T♠",
        "6♣ 6♦ A♣ 3♠ K♠",
        "9♣ A♥ K♠ 3♣ K♦ 9♦ 6♦"] ~?=
    ["6♥ 6♦ 6♠ 6♣",
     "6♣ 4♦ A♣ 3♠ K♠ 5♦ T♠ High Card",
     "6♣ 6♦ A♣ 3♠ K♠" ,
     "9♣ A♥ K♠ 3♣ K♦ 9♦ 6♦ Two Pairs (winner)"]
```

And an important one, for that matter. Can we make it pass?

14 Wow. This is a big test!

15 Let's try. First We have to find the max ranking for each hand:

```
scores :: [String] → [String]
scores input = let rs = map maxRanking input
```

Then we have to compute the marks:

```
ms = markResults rs
```

Then we join them with a concatenation operation:

```
in zipWith (++) input ms
```

■ Does it work?

No. The resulting lines lack a space between the input and the marks: We expect:
"6♣ 4♦ A♣3♠ K♠5♦ T♠ High Card"
and we get "6♣ 4♦ A♣3♠ K♠5♦ T♠High Card".

16 ■ Then (++) is not the good operation to zip the lists with. Let's write our own function:

```
scores :: [String] → [String]
scores input = let rs = map (maxRanking . cards) input
                   ms = markResults rs
               in zipWith join input ms
                   where join a b = a ++ ' ': b
```

■ Does it work now?

No. We have a supplementary space on the first line:
We expect "6♣ 6♦ A♣3♠ K♠"
and we get "6♣ 6♦ A♣3♠ K♠ ".

17 ■ Sure: when there is no mark, we shouldn't add that space. Let's add a pattern.

```
scores :: [String] → [String]
scores input = let rs = map (maxRanking . cards) input
                   ms = markResults rs
               in zipWith join input ms
                   where join a "" = a
                         join a b = a ++ ' ': b
```

■ And we're done!

Are we? Here's the final test case.

```
K♣ 9♠ K♠ K♦ 9♦ 3♣ 6♦
9♣ A♥ K♠ K♦ 9♦ 3♣ 6♦
A♣ Q♣ K♠ K♦ 9♦ 3♣
9♥ 5♠
4♦ 2♦ K♠ K♦ 9♦ 3♣ 6♦
7♠ T♠ K♠ K♦ 9♦
```

I put it in a file named *game.txt*.

18 Ok. We just have to create a main program which would process this file and compute the scores.

```
module Main
where
import PokerHand
```

Let's call this program *Scores.hs*.

Ok. How does the program work?

19 Very simple. First we get the text from the input. We have to separate this text into lines, calculate the scores, assemble the result back into a text, which we display on the output:

```
main = getContents
       >>= lines
       >>. scores
       >>. unlines
       >>. putStrLn
```

How do we try it?

20 just type:

```
runghc Scores <game.txt
```

Here's the output:

```
K♣ 9♠ K♠ K♦ 9♦ 3♣ 6♦ Full House (winner)
9♣ A♥ K♠ K♦ 9♦ 3♣ 6♦ Two Pair
A♣ Q♣ K♠ K♦ 9♦ 3♣
9♥ 5♠
4♦ 2♦ K♠ K♦ 9♦ 3♣ 6♦ Flush
7♠ T♠ K♠ K♦ 9♦
```

So, I guess it works.

21 ■ ■ ■ It works! Hurray!

Let's ship our program to the customer

22 And have a fantastic dinner!

```
module Tests
where
import Test.HUnit
import PokerHand
import Data.Ord (comparing)
import Data.List (sort,sortBy)

ud = words "A♣ 2♣ T♣ K♣ 9♣ Q♣ J♣"
sd = words "2♣ 9♣ T♣ J♣ Q♣ K♣ A♣"

main = runTestTT $ TestList
       [sortBy (comparing card) ud ~?= sd
       ,map suit (cards "A♣ A♦ A♥ A♠") ~?= ['♣','♦','♥','♠']
       ,flush (cards "A♣ T♣ 3♣ 4♣ 2♣") ~?= True
       ,flush (cards "A♠ T♣ 3♣ 4♣ 2♣") ~?= False
       ,flush (cards "A♠ T♠ 3♠ 4♠ 2♠") ~?= True
       ,"6♣ 4♦ A♣ 3♠ K♠" `beat` "8♥ J♥ 7♦ 5♥ 6♣"
       ,"5♥ 2♦ 3♥ 4♦ 2♥" `beat` "A♥ K♥ Q♦ J♦ 9♥"
       ,"5♥ 4♦ 3♥ 2♦ 3♣" `beat` "A♥ K♥ Q♦ J♦ 9♥"
       ,"5♥ 4♦ 3♥ 3♣ 2♥" `beat` "7♦ 5♥ 3♦ 2♠ 2♦"
       ,"2♦ 2♣ 3♣ 3♠ 4♥" `beat` "A♥ A♠ K♣ Q♦ J♠"
       ,"2♦ 2♣ 2♠ 3♥ 4♦" `beat` "A♥ A♠ K♣ K♦ J♠"
       ,"2♦ 2♠ 2♥ 2♣ 3♦" `beat` "A♥ A♦ A♠ K♥ K♠"
       ,"6♠ 5♦ 4♣ 3♦ 2♥" `beat` "A♣ A♥ A♦ K♣ Q♠"
       ,"5♠ 4♦ 3♣ 2♦ A♥" `beat` "A♣ A♥ A♦ K♣ Q♠"
       ,"6♥ 4♥ 3♥ 2♥ A♥" `beat` "A♠ K♣ Q♥ J♠ T♦"
       ,"5♥ 4♥ 3♥ 2♥ A♥" `beat` "A♦ A♠ A♥ A♠ K♥"
       ,"6♥ 5♥ 4♥ 3♥ 2♥" `beat` "A♦ A♠ A♥ A♠ K♥"
       , TestList [show HighCard ~?= "High Card",
                   show Pair ~?= "Pair",
```

```
                  show TwoPairs ~?= "Two Pairs",
                  show ThreeOfAKind ~?= "Three of a Kind",
                  show Straight ~?= "Straight",
                  show Flush ~?= "Flush",
                  show FullHouse ~?= "Full House",
                  show FourOfAKind ~?= "Four of a Kind",
                  show StraightFlush ~?= "Straight Flush"]
    ,maxRank "6♥ 6♦ 6♠ 6♣ K♠ K♦" ~?= Nothing
    ,maxRank "6♣ 4♦ A♣ 3♠ K♠ T♦ 8♣" ~?= Just HighCard
    ,maxRank "6♣ 6♦ A♣ 3♠ K♠ T♥ 8♦" ~?= Just Pair
    ,markResults [Nothing] ~?= [""]
    ,markResults [Nothing, Just Pair] ~?= ["", "Pair (winner)"]
    ,markResults [Nothing, Just Pair, Just HighCard] ~?=
                    ["", "Pair (winner)", "High Card"]
    ,scores ["6♥ 6♦ 6♠ 6♣",
             "6♣ 4♦ A♣ 3♠ K♠ 5♦ T♠",
             "6♣ 6♦ A♣ 3♠ K♠",
             "9♣ A♥ K♠ 3♣ K♦ 9♦ 6♦"] ~?=
                ["6♥ 6♦ 6♠ 6♣",
                 "6♣ 4♦ A♣ 3♠ K♠ 5♦ T♠ High Card",
                 "6♣ 6♦ A♣ 3♠ K♠" ,
                 "9♣ A♥ K♠ 3♣ K♦ 9♦ 6♦ Two Pairs (winner)"]
  ]
  where beat h g = comparing (hand . cards) h g ~?= GT
        maxRank = maxRanking . cards
```

Listing 1: Tests.hs

```
module PokerHand
where
import Char
import Data.Ord
import Data.List

data Card = C { value :: Value, suit :: Suit }
          deriving (Ord,Eq)
type Value = Int
type Suit = Char

data Hand = H Ranking [Card]
          deriving (Ord,Eq)

ranking :: Hand → Ranking
ranking (H r _) = r

data Ranking = HighCard
             | Pair
             | TwoPairs
             | ThreeOfAKind
             | Straight
             | Flush
             | FullHouse
             | FourOfAKind
             | StraightFlush
             deriving (Ord,Eq)

instance (Show) Ranking
  where
    show HighCard     = "High Card"
    show Pair         = "Pair"
    show TwoPairs     = "Two Pairs"
    show ThreeOfAKind = "Three of a Kind"
    show Straight     = "Straight"
    show Flush        = "Flush"
    show FullHouse    = "Full House"
```

```
    show FourOfAKind = "Four of a Kind"
    show StraightFlush = "Straight Flush"

card :: String → Card
card [v,s] = C (toValue v) s
  where
    toValue 'A' = 14
    toValue 'K' = 13
    toValue 'Q' = 12
    toValue 'J' = 11
    toValue 'T' = 10
    toValue c = ((ord c) - (ord '0'))

same :: (Eq a) => (t → a) → t → t → Bool
same f a b = f a == f b

flush :: [Card] → Bool
flush (c:cs) = all (same suit c) cs


rSortBy :: (Ord a) => (a → a → Ordering) → [a] → [a]
rSortBy f = sortBy (flip f)

(>>.) :: (a → b) → (b → c) → (a → c)
(>>.) = flip (.)

scores :: [String] → [String]
scores input = let rs = map (maxRanking . cards) input
                   ms = markResults rs
               in zipWith join input ms
                   where join a "" = a
                         join a b = a ++ ' ': b

markResults :: [Maybe Ranking] → [String]
markResults rs = map mark rs
  where mark Nothing = ""
        mark (Just r) = (show r) ++ winner (Just r)
        winner v | v == m = " (winner)"
        winner _ = ""
        m = maximum rs

maxRanking :: [Card] → Maybe Ranking
maxRanking cs | length cs < 7 = Nothing
maxRanking cs = Just $ max (subLists cs)
  where
    max = maximum . map (ranking . hand)
    subLists = filter ((5==).length) . subsequences

hand :: [Card] → Hand
hand = rSortBy (comparing value)
     >>. groupBy (same value)
     >>. rSortBy (comparing length)
     >>. rank
     >>. promoteStraight
     >>. promoteFlush

rank :: [[Card]] → Hand
rank gs = H (calcRank gs) (concat gs)
  where calcRank [[_,_,_,_],_] = FourOfAKind
        calcRank [[_,_,_],_] = FullHouse
        calcRank [[_,_,_],_,_] = ThreeOfAKind
        calcRank [[_,_],[_,_],_] = TwoPairs
        calcRank [[_,_],_,_,_] = Pair
        calcRank [_,_,_,_,_] = HighCard
```

```
cards :: String → [Card]
cards = map card . words

promoteStraight :: Hand → Hand
promoteStraight (H r [a,b,c,d,e])
    | value a − value e == 4 =
        H Straight [a,b,c,d,e]
promoteStraight (H HighCard [a,b,c,d,e])
    | value a == 14 && value b == 5 =
        H Straight [b,c,d,e,a]
promoteStraight h = h


promoteFlush :: Hand → Hand
promoteFlush (H HighCard cs)
    | flush cs = H Flush cs
promoteFlush (H Straight cs)
    | flush cs = H StraightFlush cs
promoteFlush h = h
```

Listing 2: Tests.hs