

# Poker Hand

by ToF

November 12, 2011

$\lambda \ \lambda \ \lambda$

## 1 We have a problem...

---

What is this : "8♥"?

A **String**.

---

Yes. What does it represent?

An eight of hearts, or 8♥.

---

What does "7♣ 6♦ 9♠" represent?

Some others cards: 7♣, 6♦, and 9♠.

---

Right. And what does "A♥ K♥ Q♥ J♥ T♥" represent?

It represents victory: it's a royal flush.

---

What is the best we can do with the following:  
4♦ 2♦ K♠ K♦ 9♦ 3♣ 6♦?

A flush.

---

And with 9♣ A♥ K♠ K♦ 9♦ 3♣ 6♦?

Two pairs.

---

Correct. And with A♣ Q♣ K♠ K♦ 9♦ 3♣?

Nothing, because there are less than seven cards.

---

And with 9♥ 5♠?

Nothing, for the same reason.

---

That's right.  
What about K♣ 9♠ K♠ K♦ 9♦ 3♣ 6♦?

It's a full house. Say, why are you showing me all these cards?

---

Because we have a problem, and I wanted to be sure you know the basics about *Poker*.

Show me what the problem is.

---

We have to write a program with, given this input:

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♦

These are the cards of some players in a game of *Texas Hold'em*. Right?

– right – ..would output this:

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♦

I see.

What do you see?

Some lines are just left as they are.  
 Some lines are marked with the ranking of the best possible hand given the cards on the line.  
 The line with the best ranking is marked as the winner.

Do you think we can solve the problem?

Yes, provided we have the good tools.

What is the value of: `filter even [4,8,0,7]`?

[4,8,0]

What is the value of: `subsequences "abc"`?

[ "", "a", "b", "ab", "c", "ac", "bc", "abc" ]

And of the expression: `maximum [4,8,0,7]`?

8

What about: `zip [3,5,2] "abc"`?

[ (3,' a' ) ,(5,' b' ) ,(2,' c' ) ]

and the expression: `zipWith (*) [3,5,2] [4,9,7]`?

[12, 45, 14]

What is the value of:  
`words "time flies like an arrow"`?

[ "time", " flies ", " like ", "an", "arrow" ]

What is the value of: `compare "time" "arrow"`?

GT, because "time" > "arrow"

What is the value of:  
`comparing length "time" "arrow"`?

LT, because (length "time") < (length "arrow")

Do you want to solve the problem?

Let's make some tea first.

## 2 Dealing with Cards

---

What is something simple we could begin to solve?

Comparing cards.

---

How do we proceed?

Write a failing test.

---

Ok. Let's compare a 6♣ and a 6♠. These two cards should be considered equals in value.

```
module Tests
where
import Test.HUnit

main = runTestTT $ TestList
    [compare "6♣" "6♠" ~?= EQ]
```

■ The result is failure:

```
expected: EQ
but got: LT
```

But it's not a big matter, since we're comparing [Strings](#) when we should compare [Cards](#).

What is the result?

---

What is a [Card](#)?

It's a new data type.

---

How do I create values of this type?

Pretend you have a function from [String](#) to [Card](#).

---

Ok. I'll just call that function *card* :

```
main = runTestTT $ TestList
    [compare (card "6♣") (card "6♠") ~?= EQ]
```

■ Error, as expected. Let me just write the function.

```
module PokerHand
where

card :: String → Card
```

What now?

---

■ This results in two errors:

The type signature for 'card' lacks an accompanying binding

Not in scope: type constructor or class 'Card'

Can you write provide the missing parts?

■ OK, this is the `Card` type:

```
data Card = C
```

It has just a single value, `C`. And we implement the function

```
card :: String → Card
card _ = C
```

which is just producing the single value.

■ Now we have another error:

No instance for (Ord Card) arising from a use of 'compare'

Possible fix: add an instance declaration for (Ord Card)

Should we make the suggested fix?

■ Sure:

```
data Card = C deriving (Ord)

card :: String → Card
card _ = C
```

■ Now we have this:

No instance for (Eq Card) arising from a use of 'compare'

Possible fix: add an instance declaration for (Eq Card)

■ Again, let's do what the compiler suggests

```
data Card = C deriving (Ord,Eq)

card :: String → Card
card _ = C
```

■ And the test passes.

Of course, this is just a *fake* implementation of the function `card`.

Then write another test.

Here you go:

```
main = runTestTT $ TestList
  [compare (card "6♣") (card "6♠") ~?= EQ
  ,compare (card "6♣") (card "5♠") ~?= GT]
```

How do we make it pass?

■ We have to compare the rank values of the cards, so we should store this value in the `Card` type:

```
data Card = C Value deriving (Ord,Eq)
type Value = Int

card :: String → Card
card _ = C 0
```

■ Of course, the test now fails, as we must calculate the real value instead of returning zero. Let's think..

■ Just make the test pass. I don't like having to think on a red bar.

■ Let's play "*fake it 'til you make it*" then:

```
card :: String → Card
card ['6', _] = C 6
card ['5', _] = C 5
```

■ Now it's obvious.

■ Indeed, just convert from `Char` to `Int`, using the `ord` function. Do it.

■ Ok!

```
module PokerHand
where
import Char

data Card = C Value deriving (Ord,Eq)
type Value = Int

card :: String → Card
card [c, _] = C $ (ord c) - (ord '0')
```

■ Done.

Done? I think I have a new test to write. But first I'll do some refactoring, too.

```
main = runTestTT $ TestList
  [compare (card "6♣") (card "6♠") ~?= EQ
  ,compare (card "6♣") (card "5♠") ~?= GT]
```

You know about `comparing` right?

Yes, and so does *GHCI*:

```
comparing :: (Ord a) => (b -> a) -> b -> b -> Ordering
--Defined in Data.Ord
```

`comparing` takes a function from a type `b` to an ordered type `a`, two values of type `b` and gives the comparison using the given function.

Yes, so I can compare `Strings` using the `card` function:

```
import Data.Ord (comparing)

main = runTestTT $ TestList
  [comparing card "6♣" "6♠" ~?= EQ
  ,comparing card "6♣" "5♠" ~?= GT]
```

■ Nice!

Now for my new test:

```
main = runTestTT $ TestList
  [comparing card "6♣" "6♠" ~?= EQ
  ,comparing card "6♣" "5♠" ~?= GT
  ,comparing card "T♣" "J♠" ~?= LT]
```

■ We're expecting `LT` but get `GT`. Can you make it pass?

■ Sure:

```
card :: String -> Card
card ['J',_] = C 11
card ['T',_] = C 10
card [c, _] = C $ (ord c) - (ord '0')
```

■ We just have to add special cases.

Good. Here's a new one:

```
main = runTestTT $ TestList
  [comparing card "6♣" "6♠" ~?= EQ
  ,comparing card "6♣" "5♠" ~?= GT
  ,comparing card "T♣" "J♠" ~?= LT
  ,comparing card "K♣" "A♠" ~?= LT]
```

■ Ok.

```
card :: String -> Card
card ['A',_] = C 14
card ['K',_] = C 13
card ['J',_] = C 11
card ['T',_] = C 10
card [c, _] = C $ (ord c) - (ord '0')
```

■ That's easy: give each card its value.

We forgot the Queen value:

```
main = runTestTT $ TestList
  [comparing card "6♣" "6♠" ~?= EQ
  ,comparing card "6♣" "5♠" ~?= GT
  ,comparing card "T♣" "J♠" ~?= LT
  ,comparing card "K♣" "A♠" ~?= LT
  ,comparing card "Q♣" "K♠" ~?= LT]
```

Can you add it?

■ Sure:

```
card :: String -> Card
card ['A',_] = C 14
card ['K',_] = C 13
card ['Q',_] = C 12
card ['J',_] = C 11
card ['T',_] = C 10
card [c, _] = C $ (ord c) - (ord '0')
```

■ And we are done with card values.

We are, but these tests are a bit heavy. Can you think of a way to avoid repeating all these comparisons?

Yes: we could test the sorting of a deck.

```

module Tests
where
import Test.HUnit
import PokerHand
import Data.List (sort)

ud = map card ["A♠","2♠","T♠","K♠","9♠","Q♠","J♠"]
sd = map card ["2♠","9♠","T♠","J♠","Q♠","K♠","A♠"]

main = runTestTT $ TestList
      [sort ud ~?= sd]

```

Is that what you mean?

■ Yes, but we have a new problem.

■ Indeed:

No instance for (Show Card) arising from a use of '~?='

Possible fix: add an instance declaration for (Show Card)

Should we follow the suggestion?

■ No. I don't think the `Card` type should derive the `Show` class just for testing reasons.

Then should we get back to the previous version of our tests?

I have a better idea: instead of comparing lists of `Cards` we can compare lists of `Strings`.

■ Comparing the `Strings`? Ok:

```

ud = ["A♠","2♠","T♠","K♠","9♠","Q♠","J♠"]
sd = ["2♠","9♠","T♠","J♠","Q♠","K♠","A♠"]

main = runTestTT $ TestList
      [sort ud ~?= sd]

```

■ But now the test fails:

expected: ["2♠","9♠","T♠","J♠","Q♠","K♠","A♠"]  
but got: ["2♠","9♠","A♠","J♠","K♠","Q♠","T♠"]

Do you see why?

■ Of course: we don't use `Cards` any more! We should compare the `Strings` using the `card` function. The function

`sortBy :: (a → a → Ordering) → [a] → [a]`

allows us to do that.

You mean like this:

```

import Data.Ord (comparing)
import Data.List (sortBy)

ud = ["A♠","2♠","T♠","K♠","9♠","Q♠","J♠"]
sd = ["2♠","9♠","T♠","J♠","Q♠","K♠","A♠"]

main = runTestTT $ TestList
      [sortBy (comparing card) ud ~?= sd]

```

■ Yes!

I wonder what would the test show if it failed. Let's falsify it:

```
import Data.Ord (comparing)
import Data.List (sortBy)

ud = ["3♣", "2♣", "T♣", "K♣", "9♣", "Q♣", "J♣"]
sd = ["2♣", "9♣", "T♣", "J♣", "Q♣", "K♣", "A♣"]

main = runTestTT $ TestList
      [sortBy (comparing card) ud ~?= sd]
```

I just changed the first value of the unsorted desk.

■ Here is what the message says:

```
expected: ["2♣", "9♣", "T♣", "J♣", "Q♣", "K♣", "A♣"]
but got:  ["2♣", "3♣", "9♣", "T♣", "J♣", "Q♣", "K♣"]
```

The test properly outputs the results as a list of [Strings](#). You can un-falsify the test now.

Yes.

Oh, and using [words](#) for the definition of our decks would make the code prettier.

You are right. So this is the test code:

```
module Tests
where
import Test.HUnit
import PokerHand
import Data.Ord (comparing)
import Data.List (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]
```

■ And this is the tested code:

```
module PokerHand
where
import Char

data Card = C Value deriving (Ord, Eq)
type Value = Int

card :: String → Card
card ['A', _] = C 14
card ['K', _] = C 13
card ['Q', _] = C 12
card ['J', _] = C 11
card ['T', _] = C 10
card [c, _] = C $ (ord c) - (ord '0')
```

Are we done with comparing *Cards*?

Not yet, but it's time for a break.



### 3 Looking for a Flush

---

What is the next task with regard to card comparison?

We need to compare suits so that we can find a *flush*.

---

Ok I'll write a test:

```
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
  , flush (cards "A♠ T♠ 3♠ 4♠ 2♠") ~= True]
  where cards = map card . words
```

■ Let's write a function *flush*

```
flush :: [Card] → Bool
flush _ = True
```

■ Done.

---

I see. Still the *fake it 'til you make it* approach.

This is the simplest thing that makes the test pass.

---

Ok. Here is another test:

```
main = runTestTT $ TestList
  [sortBy (comparing card) ud ~= sd
  , flush (cards "A♠ T♠ 3♠ 4♠ 2♠") ~= True
  , flush (cards "A♠ T♠ 3♠ 4♠ 2♠") ~= False ]
  where cards = map card . words
```

■ I don't think so.

Can you make it pass?

---

What is missing?

The `Card` type doesn't include suits.

---

How can we change that?

Add a failing test on getting **Suits** from **Cards**.

Ok, then I'll replace my last test with this one:

```
main = runTestTT $ TestList
  [sortBy (comparing card) ud ~?= sd
  , flush (cards "A♣ T♣ 3♣ 4♣ 2♣") ~?= True
  , map suit (cards "A♣ A♦ A♣ A♣") ~?= ['♣','♦','♣','♣']
  ]
  where cards = map card . words
```

Can you make this one pass?

■ First we need a **suit** function:

```
type Suit = Char

suit :: Card → Suit
suit _ = '♣'
```

■ Now the test is failing.

What else is needed?

■ We must store the suit into to the **Card** type:

```
data Card = C Value Suit deriving (Ord,Eq)
```

And then we have to capture the suit in the **card** function:

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

■ The code in the **card** function is a bit tedious, don't you think?

■ I'll refactor it when the bar is green. I still have to remove the *fake* on **suit**:

```
suit :: Card → Suit
suit (C _ s) = s
```

■ And now we can get **Suits** from **Cards**.

Good. Refactor the code, now.

■ Allright. First I can discard the **suit** function by declaring labels:

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

Then I can separate concerns in the **card** function:

```
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'))
```

■ Done.

Can I add my test on **flush** now?

Yes.

Here it is:

```
main = runTestTT $ TestList
  [sortBy (comparing card) ud ~?= sd
  , flush (cards "A♠ T♠ 3♠ 4♠ 2♠") ~?= True
  , map suit (cards "A♠ A♠ A♠ A♠") ~?= ['♠','♠','♠','♠']
  , flush (cards "A♠ T♠ 3♠ 4♠ 2♠") ~?= False]
  where cards = map card . words
```

Do you see how to make it pass?

■ Sure:

```
flush :: [Card] → Bool
flush (c:_) = suit c == '♠'
```

■ As you see, it's a *fake*.

In that case, I'll add a new test :

```
main = runTestTT $ TestList
  [sortBy (comparing card) ud ~?= sd
  , flush (cards "A♠ T♠ 3♠ 4♠ 2♠") ~?= True
  , map suit (cards "A♠ A♠ A♥ A♠") ~?= ['♠','♠','♦','♠']
  , flush (cards "A♠ T♠ 3♠ 4♠ 2♠") ~?= False
  , flush (cards "A♠ T♠ 3♠ 4♠ 2♠") ~?= True]
  where cards = map card . words
```

■ Ok. I think I can take a more general approach:

```
flush :: [Card] → Bool
flush (c:cs) = all (\x → suit x == suit c) cs
```

■ Of course, we're assuming that the *flush* function will always consume non-empty lists.

Ok. This are the tests so far:

```
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]
  where cards = map card . words
```

And this is the tested code:

```
module PokerHand
where
import Char

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

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'))

flush :: [Card] → Bool
flush (c:cs) = all (\x → suit x == suit c) cs
```

Are we done with comparing cards?

I think so. Let's have lunch.

## 4 “Pair” Programming

---

Now that we have suitable tools to compare cards, what should we do?

Compare hands.

---

How do we form a [Hand](#)?

We’ll write a function:

```
type Hand = [Card]
hand :: String → Hand
```

---

Good. But we should write a test before writing code.

Go on.

---

What is the simplest hand comparison we could write a test for?

Let’s try comparing simple “High Cards” hands.

---

Ok. Here is a new test:

This last test is a bit long.

```
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
  ,comparing hand "6♣ 4♦ A♠ 3♠ K♠" "8♠ J♥ 7♦ 5♥ 6♠" ~?= GT]
  where cards = map card . words
```

---

Ok, let's rephrase it this way:

```
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♣"]
  where cards = map card . words
        beat h g = comparing hand h g ~?= GT
```

■ OK. We need to create the *hand* function. But first I will borrow your *cards* utility function.

Sure, take it to your side.

```
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♣"]
  where beat h g = comparing hand h g ~?= GT
```

■ Thanks

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

In fact forming a hand is just making *Cards* from *Strings* and sorting them:

```
hand :: String → Hand
hand = sort . cards
```

■ Except we get *LT* instead of *GT*.

Of course: we're sorting in the wrong order. How can we change the sorting order?

We can use *sortBy* and give it the proper comparison function.

Given what *GHCI* tells us about *sort*, *sortBy* and *compare*:

```
:type sort
sort :: (Ord a) => [a] -> [a]
:type sortBy
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
:type compare
compare :: (Ord a) => a -> a -> Ordering
```

We know that *sortBy compare* is equivalent to *sort*. How can we reverse the result given by *compare*?

By *flip* ping its arguments. *flip f a b* is equivalent to *f b a*. Thus:

```
hand :: String → Hand
hand = sortBy (flip compare) . cards
```

■ will do the trick.

Ok. What is the next hand that can beat a *High Card*?

A *Pair*.

Then I'll write this test:

```
, "5♥ 2♦ 4♦ 3♥ 2♥" `beat` "A♥ K♥ Q♦ J♦ 9♥"]
where beat h g = comparing hand h g ~?= GT
```

■ The test fails. We have to detect that the hand is a pair, and use that information to trump the usual card comparison.

Meaning: the lowest *Pair* should beat the highest *High Card*.

How do we do that?

■ We declare that, within the *Hand* type, a *Pair* is always greater than a *High Card*.

How do we order values within a type?

■ We declare it as an algebraic type, saying we either have a `HighCard` followed by a list of `Cards`, or a `Pair`:

```
data Hand = HighCard [Card]
          | Pair
          deriving (Ord,Eq)
```

■ Of course, now the implementation of `hand` doesn't yield a correct `Hand` value.

The compiler says:

Couldn't match expected type 'Hand'  
against inferred type '[Card]'  
In the expression:  
    sortBy (flip compare) . cards  
In the definition of 'hand':  
    hand = sortBy (flip compare) . cards  
Can you arrange this?

Yes. Let's begin by forcing the function to a `HighCard` value:

```
hand :: String → Hand
hand = HighCard . sortBy (flip compare) . cards
```

■ and we're back with a failing test instead of a compiler error.

Can you *fake* the correct construction that would make the test pass?

Yes. Let's just insert a special case:

```
hand :: String → Hand
hand "5♥ 2♦ 4♣ 3♥ 2♥" = Pair
hand s = HighCard $ sortBy (flip compare) $ cards s
```

■ And the test is passing, because given the declaration of `Hand`, `Pair` is a higher `Hand` value than `HighCard`.

Now we need to triangulate, so I'm adding a new test about a *Pair* beating a *High Card*:

```
, "5♥ 2♦ 3♥ 4♦ 2♥" 'beat' "A♥ K♥ Q♦ J♦ 9♥"
, "5♥ 4♦ 3♥ 2♦ 3♣" 'beat' "A♥ K♥ Q♦ J♦ 9♥"]
```

■ I'll aggravate my *fake* with a new pattern:

```
hand :: String → Hand
hand "5♥ 2♦ 3♥ 4♦ 2♥" = Pair
hand "5♥ 4♦ 3♥ 2♦ 3♣" = Pair
hand s = HighCard $ sortBy (flip compare) $ cards s
```

■ And now we have to think.

How can we get rid of these *fake* implementations?

By writing a function from `String` to `Bool` that detects a *Pair*.

If you had this function, what would the `hand` function look like?

It would look like this:

```
hand :: String → Hand
hand s | hasAPair s = Pair
hand s = HighCard $ sortBy (flip compare) $ cards s
```

■ The code is broken, now.

Can you write the function `hasAPair`?

Yes:

```
hasAPair :: String → Bool
hasAPair "5♥ 2♦ 3♥ 4♦ 2♥" = True
hasAPair "5♥ 4♦ 3♥ 2♦ 3♣" = True
hasAPair _ = False
```

■ Done.

There’s a bit of noise in these patterns. Do we really need to deal with `Strings`?

■ No, we can match patterns on the card `Values`:

```
hand :: String → Hand
hand s | hasAPair $ map value $ cards s = Pair
hand s = HighCard $ sortBy (flip compare) $ cards s

hasAPair :: [Value] → Bool
hasAPair [5,2,3,4,2] = True
hasAPair [5,4,3,2,3] = True
hasAPair _ = False
```

Would it help if we sorted the values?

■ That would clarify the patterns, so let’s do it:

```
hand :: String → Hand
hand s | hasAPair $ sort $ map value $ cards s = Pair
hand s = HighCard $ sortBy (flip compare) $ cards s

hasAPair :: [Value] → Bool
hasAPair [2,2,3,4,5] = True
hasAPair [2,3,3,4,5] = True
hasAPair _ = False
```

Do you see something common between the first two patterns of `hasAPair`?

Apart from the fact they both end with `3,4,5`, no.

Can you group the values after sorting them?

■ Ok. We have to change the signature for the function.

```
hand :: String → Hand
hand s | hasAPair $ group $ sort $ map value $ cards s = Pair
hand s = HighCard $ sortBy (flip compare) $ cards s

hasAPair :: [[Value]] → Bool
hasAPair [[2,2],[3],[4],[5]] = True
hasAPair [[2],[3,3],[4],[5]] = True
hasAPair _ = False
```

■ Oh. Now I see something.

What do you see?

Each list contains four groups. So that would be a way to detect any *Pair*!

How would write the function, then?

■ Like this:

```
hasAPair :: [[Value]] → Bool
hasAPair gs = length gs == 4
```

■ The code is still quite messy, though.

How can we refactor?

First, factorize parts of the expression, like `cards s`

```
hand :: String → Hand
hand s | hasAPair $ group $ sort $ map value $ cs = Pair
hand s = HighCard $ sortBy (flip compare) $ cs
      where cs = cards s
```

■ Oops. That doesn’t work

The compiler says:

Not in scope: ‘cs’

Your `cs` variable should be declared for the first pattern too.

Ok. Let’s go back to green.

```
hand :: String → Hand
hand s | hasAPair $ group $ sort $ map value $ cs = Pair
      where cs = cards s
hand s = HighCard $ sortBy (flip compare) $ cs
      where cs = cards s
```

■ Now we can continue to refactor.

How can you write only one pattern in this function?

■ By using an `if`:

```
hand :: String → Hand
hand s = if hasAPair $ group $ sort $ map value $ cs
      then Pair
      else HighCard $ sortBy (flip compare) $ cs
      where cs = cards s
```

Now, add legibility.

■ Let’s have more auxiliary functions, and bring `hasAPair` where it belongs:

```
hand :: String → Hand
hand s = if hasAPair (groups cs) then Pair
      else HighCard $ sortBy (flip compare) $ cs
      where cs = cards s
            groups = group . sort . map value
            hasAPair gs = length gs == 4
```

In this function, we sort the cards twice. Would the `grouping` still work if it used `sortBy (flip compare)` instead of `sort`?

■ Let’s ask the code:

```
hand :: String → Hand
hand s = if hasAPair (groups cs) then Pair
      else HighCard $ sortBy (flip compare) $ cs
      where cs = cards s
            groups = group . sortBy (flip compare) .
                  map value
            hasAPair gs = length gs == 4
```

■ Yes, the criteria of having four groups still holds, whatever the order in which sort the cards.

So we can factorize the sorting.

■ Right. Now `cs` represent the sorted cards:

```
hand :: String → Hand
hand s = if hasAPair (groups cs) then Pair
      else HighCard cs
      where cs = sortBy (flip compare) $ cards s
            groups = group . map value
            hasAPair gs = length gs == 4
```

■ But, this code is still too long.



Maybe we can get rid of *hasAPair*

■ Let’s try:

```
hand :: String → Hand
hand s = case length $ groups cs of
  4 → Pair
  5 → HighCard cs
  where cs = sortBy (flip compare) $ cards s
        groups = group . map value
```

■ Right.

And harmonize variable names, like *gs* instead of *groups*...

■ You mean like this:

```
hand :: String → Hand
hand s = case length gs of
  4 → Pair
  5 → HighCard cs
  where cs = sortBy (flip compare) $ cards s
        gs = group $ map value $ cs
```

■ Yeah, that’s a bit clearer.

Can you add symmetry? Using *groupBy* instead of *group* and *map*.

■ Sure:

```
hand :: String → Hand
hand s = case length gs of
  4 → Pair
  5 → HighCard cs
  where cs = sortBy (flip compare) $ cards s
        gs = groupBy (same value) cs
        same f a b = f a == f b
```

■ That’s even clearer.

Hey, that *same* function is interesting. Do you see where we met a case for it before?

No.

Look at the *flush* function.

Here it is:

```
flush :: [Card] → Bool
flush (c:cs) = all (\x → suit x == suit c) cs
```

Can you use something similar to the function *same* here?

Let’s try:

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

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

■ You are right.

Simplify, then!

Ok.

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

hand :: String → Hand
hand s = case length gs of
    4 → Pair
    5 → HighCard cs
    where cs = sortBy (flip compare) $ cards s
          gs = groupBy (same value) cs
```

Ok. Here's is the test code:

```
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♥"]
    where beat h g = comparing hand h g ~?= GT
```

■ And this is the tested code:

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

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

data Hand = HighCard [Card] | Pair deriving (Ord,Eq)

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'))

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

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

hand :: String → Hand
hand s = case length gs of
    4 → Pair
    5 → HighCard cs
    where cs = sortBy (flip compare) $ cards s
          gs = groupBy (same value) cs

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

What should we do now?

Have some rest!

## 5 Grouping Cards

So far, our hand comparisons are correct as long as we compare *High Cards* hands or compare a *High Card* to a *Pair*. What's the next step?

Comparing *Pairs*.

Ok. Here's a test:

```
, "5♥ 4♦ 3♥ 2♦ 3♣" 'beat' "5♥ 2♦ 3♥ 4♦ 2♥"]
  where beat h g = comparing hand h g ~= GT
```

The hand on the left should win, since a pair of 3 beats a pair of 2s. But the test fails, we get `EQ` instead of `GT`.

■ We can solve this by storing cards along with the `Pair` value in the `Hand` type:

```
data Hand = HighCard [Card]
          | Pair      [Card]
          deriving (Ord,Eq)
```

And we must complete the `hand` function, too.

■ Yes:

```
hand :: String → Hand
hand s = case length gs of
    4 → Pair cs
    5 → HighCard cs
  where cs = sortBy (flip compare) $ cards s
        gs = groupBy (same value) cs
```

■ And now the test passes.

We have a problem, though. Can you see it?

Not yet.

Look at the `hand` function.

What is the value of `cs` when `s` equals `"5♥ 4♦ 3♥ 2♦ 3♣"`?

That's `[5, 4, 3, 3, 2]`.

And what would be the value of `cs` if `s` was equal to  
`"5♥2♦3♥7♦2♦"`?

`[7,5,3,2,2]`. Ouch.

Let's write a new test:

```
, "5♥4♦3♥3♣2♥" 'beat' "7♦5♥3♦2♠2♦"]
```

■ and sure enough the test is failing.

■ I see. The value of the pair should beat the value of the remaining cards.

Do you know how to solve this?

No.

What is the simplest possible thing that would make the tests pass?

Using the *fake it* strategy. We can arrange the cards according to their place in the groups list.

Well, do this, then.

I want to refactor the code, first.

Ok I'm removing my last test

■ Thanks. Here's my refactoring:

```
hand :: String → Hand
hand s = case gs of
    [_,_,_,_] → Pair cs
    [_,_,_,_] → HighCard cs
    where cs = sortBy (flip compare) $ cards s
          gs = groupBy (same value) cs
```

■ Everything is still working fine.

What's the use of these patterns?

■ Describing the two cases of *Pair* that we have so far:

```
hand :: String → Hand
hand s = case gs of
    [[a],[b],[c],[d,e]] → Pair cs
    [[a],[b],[c,d],[e]] → Pair cs
    [_,_,_,_] → HighCard cs
    where cs = sortBy (flip compare) $ cards s
          gs = groupBy (same value) cs
```

■ Please put your last test back in the code.

Here it is :

```
, "5♥4♦3♥3♣2♥" 'beat' "7♦5♥3♦2♠2♦"]
```

■ Still failing.

■ The `a,b,c,d,e` variables will be used to rearrange the *Pair* value.

```
hand :: String → Hand
hand s = case gs of
    [[a],[b],[c],[d,e]] → Pair [d,e,a,b,c]
    [[a],[b],[c,d],[e]] → Pair [c,d,a,b,e]
    [_,_,_,_] → HighCard cs
    where cs = sortBy (flip compare) $ cards s
          gs = groupBy (same value) cs
```

■ And now the pairs are correctly compared.

Ok. What if we have pairs on the highest values? It wouldn't match our two patterns.

I told you it was a *fake*. In fact, comparing pairs would always work if we had only one pattern for pairs: `[[a,b],[c],[d],[e]]`

How can we ensure we always have this pattern for pairs?

By sorting the groups by size, in reverse order:

```
hand :: String → Hand
hand s = case gs of
    [[a],[b],[c],[d,e]] → Pair [d,e,a,b,c]
    [[a],[b],[c,d],[e]] → Pair [c,d,a,b,e]
    [_,_,_,_] → HighCard cs
  where cs = sortBy (flip compare) $ cards s
        gs = sortBy (flip groupSize) $ groupBy (
            same value) cs
        groupSize = comparing length
```

■ That's what the `sortBy (flip groupSize)` does. But we're still in red.

Yes, we now have non-exhaustive patterns in our three last tests.

■ Let's replace the previous pair patterns with the only remaining possible one:

```
hand :: String → Hand
hand s = case gs of
    [[a,b],[c],[d],[e]] → Pair [a,b,c,d,e]
    [_,_,_,_] → HighCard cs
  where cs = sortBy (flip compare) $ cards s
        gs = sortBy (flip groupSize) $ groupBy (
            same value) cs
        groupSize = comparing length
```

■ And we're back to green.

How can we make this code more legible?

We can put some symmetry into the patterns:

```
hand :: String → Hand
hand s = case gs of
    [[a,b],[c],[d],[e]] → Pair [a,b,c,d,e]
    [[a],[b],[c],[d],[e]] → HighCard [a,b,c,d,e]
  where cs = sortBy (flip compare) $ cards s
        gs = sortBy (flip groupSize) $ groupBy (same
            value) cs
        groupSize = comparing length
```

The function is quite long; can you split it into two parts, one for grouping cards, one for finding the ranking?

■ Sure:

```
hand :: String → Hand
hand s = ranking gs
  where cs = sortBy (flip compare) $ cards s
        gs = sortBy (flip groupSize) $ groupBy (same
            value) cs
        groupSize = comparing length

ranking :: [[Card]] → Hand
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]
```

■ Done.

Then, write a clearer version of *hand*.

Here we go:

```
hand :: String → Hand
hand = ranking .
    sortBy (flip (comparing length)) .
    groupBy (same value) .
    sortBy (flip compare) .
    cards

ranking :: [[Card]] → Hand
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]
```

■ Done.

The `sortBy (flip ..)` construct is a bit complicated. Can you make the code more legible?

Yes.

```
hand :: String → Hand
hand = ranking .
    rSortBy (comparing length) .
    groupBy (same value) .
    rSortBy (comparing value) . cards

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

■ For Cards, `compare` and `comparing value` are equivalent, so we can use the latter form for symmetry.

The function we use for ranking is quite powerful. How easily do you think it could handle new rankings?

Write a new test, and we will see.

Allright. Here's a test saying that the lowest *Two Pairs* can beat the highest possible *Pair*.

```
, "2♦ 2♣ 3♣ 3♠ 4♥" 'beat' "A♥ A♠ K♣ Q♦ J♠"]
```

This test is in error with the following message:

non-exhaustive pattern in function ranking

Can you make it pass?

■ Let's begin by adding the pattern for *Three of a Kind*:

```
ranking :: [[Card]] → Hand
ranking [[a,b],[c,d],[e]] = TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]
```

■ We're not done yet.

Here's what the error message says:

Not in scope: data constructor 'TwoPairs'

Let's insert the constructor into the `Hand` type:

```
data Hand = HighCard [Card]
          | Pair [Card]
          | TwoPairs [Card]
          deriving (Ord,Eq)
```

■ And now we can detect and compare *Two Pairs* hands.

Good. Now, here's a test saying that the lowest *Three of a Kind* can beat the highest possible *Two Pairs*.

```
, "2♦ 2♣ 2♠ 3♥ 4♦" 'beat' "A♥ A♠ K♣ K♦ J♠"]
```

The fails with a message similar to the previous one:

non-exhaustive pattern in function ranking

■ Let's add the pattern for *Three of a Kind*:

```
ranking :: [[Card]] → Hand
ranking [[a,b,c],[d],[e]] = ThreeOfAKind [a,b,c,d,e]
ranking [[a,b],[c,d],[e]] = TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]
```

■ And you should have a new message.

Indeed:

Not in scope: data constructor 'ThreeOfAKind'

Here I go:

```
data Hand = HighCard [Card]
          | Pair [Card]
          | TwoPairs [Card]
          | ThreeOfAKind [Card]
          deriving (Ord,Eq)
```

■ And now we can also detect and compare *Three of a Kind* hands.

Great. What other ranking can we add that would use different group patterns?

Let's go for *Full House* and *Four of a Kind*.

What about *Straight* and *Flush*?

There's no new grouping involved in those. We can add them later.

Ok. Here's a test:

```
, "2♦ 2♣ 2♥ 2♠ 3♦" 'beat' "A♥ A♦ A♠ K♥ K♠"]
```

It states that the lowest *Four of a Kind* beats the highest *Full House*.

■ Let's begin with adding the constructors:

```
data Hand = HighCard [Card]
          | Pair [Card]
          | TwoPairs [Card]
          | ThreeOfAKind [Card]
          | FullHouse [Card]
          | FourOfAKind [Card]
          deriving (Ord,Eq)
```

Good.

■ Then we add the group patterns:

```
ranking :: [[Card]] → Hand
ranking [[a,b,c,d],[e]] = FourOfAKind [a,b,c,d,e]
ranking [[a,b,c],[d,e]] = FullHouse [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = ThreeOfAKind [a,b,c,d,e]
ranking [[a,b],[c,d],[e]] = TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]
```

■ And it's done.

Great. Here's the test code so far:

```
module Tests
where
import Test.HUnit
import PokerHand
import Data.Ord (comparing)
import Data.List (sortBy, 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♠"]
  where beat h g = comparing hand h g ~?= GT
```

And here's the tested code:

```
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 = HighCard [Card]
          | Pair [Card]
          | TwoPairs [Card]
          | ThreeOfAKind [Card]
          | FullHouse [Card]
          | FourOfAKind [Card]
              deriving (Ord, Eq)

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

hand :: String → Hand
hand = ranking .
  rSortBy (comparing length) .
  groupBy (same value) .
  rSortBy (comparing value) . cards

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

ranking :: [[Card]] → Hand
ranking [[a,b,c,d],[e]] = FourOfAKind [a,b,c,d,e]
ranking [[a,b,c],[d,e]] = FullHouse [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = ThreeOfAKind [a,b,c,d,e]
ranking [[a,b],[c,d],[e]] = TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]

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

Now is a good time to pause.

I'll have a *croque monsieur* with salad.



## 6 Straight & Flush

So far, our *hand* function is still not correct with regard to the rules of Poker.

Agreed. At least three rankings are missing.

What are they?

The *Straight*, the *Flush*, and the *Straight Flush*.

What about the *Royal Flush*?

It's another name for the highest *Straight Flush*.

I'll begin with a test for a *Straight* beating any *Three of a Kind*. What is an example of the lowest possible *Straight*?

5♦ 4♣ 3♦ 2♥ 1♠. This is a special case, though, because the ace is not the highest value in that hand.

Then, let's begin with the general case and use 6♠ 5♦ 4♣ 3♦ 2♥ instead:

```
, "6♠ 5♦ 4♣ 3♦ 2♥" 'beat' "A♣ A♥ A♦ K♣ Q♣"]
```

■ We'll use the same routine as before. First, describe the new *Hand* value:

```
data Hand = HighCard [Card]
          | Pair      [Card]
          | TwoPairs  [Card]
          | ThreeOfAKind [Card]
          | Straight  [Card]
          | FullHouse [Card]
          | FourOfAKind [Card]
          deriving (Ord, Eq)
```

■ then completing the *hand* function.

Do you know how to recognize a *Straight*?

■ Yes: it's like a *HighCard*, meaning that every value is distinct, but the values are in sequence, meaning that the highest value minus the lowest should equal 4. I'll add this criteria as guard.

Go on.

```
ranking :: [[Card]] → Hand
ranking [[a,b,c,d],[e]] = FourOfAKind [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = FullHouse [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = ThreeOfAKind [a,b,c,d,e]
ranking [[a,b],[c,d],[e]] = TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]]
  | value a - value e == 4 = Straight [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]
```

■ And now the test is passing.

Good. What about the special case where the ace is the lowest? I'll add the test:

```
, "5♠ 4♦ 3♣ 2♥ A♥" 'beat' "A♣ A♥ A♦ K♣ Q♠"]
```

■ The test fails. Can you make it pass?

■ Yes, we just have to add the same pattern with a new guard for the case where the highest card is an ace and the next one is a five:

```
ranking :: [[Card]] → Hand
ranking [[a,b,c,d],[e]] = FourOfAKind [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = FullHouse [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = ThreeOfAKind [a,b,c,d,e]
ranking [[a,b],[c,d],[e]] = TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]]
  | value a - value e == 4 = Straight [a,b,c,d,e]
  | value a == 14 && value b == 5 = Straight [b,c,d,e,a]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]
```

■ Note that I order the cards in the value differently, so that the ace is at the last position when we print it.

Ok. Now for the *Flush*. Here is a new test:

```
, "6♥ 4♥ 3♥ 2♥ A♥" 'beat' "A♣ K♣ Q♥ J♠ T♦"]
```

■ The lowest *Flush* should beat the highest *Straight*.

First, create the value:

```
data Hand = HighCard [Card]
          | Pair [Card]
          | TwoPairs [Card]
          | ThreeOfAKind [Card]
          | Straight [Card]
          | Flush [Card]
          | FullHouse [Card]
          | FourOfAKind [Card]
          deriving (Ord,Eq)
```

We already have a function to detect a *Flush*.

■ Yes, I'll just use it within a pattern similar to a *High Card*:

```
ranking :: [[Card]] → Hand
ranking [[a,b,c,d],[e]] = FourOfAKind [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = FullHouse [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = ThreeOfAKind [a,b,c,d,e]
ranking [[a,b],[c,d],[e]] = TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]]
  | value a - value e == 4 = Straight [a,b,c,d,e]
  | value a == 14 && value b == 5 = Straight [b,c,d,e,a]
  | flush [a,b,c,d,e] = Flush [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]
```

■ And we are done with *Flush*.

We now have the *Straight Flush* case. Do you know how to handle it?

Yes. Write a test.

Here it is.

```
, "5♥ 4♥ 3♥ 2♥ A♥" 'beat' "A♦ A♠ A♥ A♠ K♥"]
```

■ I started with the lowest *Straight Flush*.

■ Ok. I'll create the value, same as usual:

```
data Hand = HighCard [Card]
         | Pair [Card]
         | TwoPairs [Card]
         | ThreeOfAKind [Card]
         | Straight [Card]
         | Flush [Card]
         | FullHouse [Card]
         | FourOfAKind [Card]
         | StraightFlush [Card]
         deriving (Ord, Eq)
```

■ Then I'll add the case:

```
ranking :: [[Card]] → Hand
ranking [[a,b,c,d],[e]] = FourOfAKind [a,b,c,d,e]
ranking [[a,b,c],[d,e]] = FullHouse [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = ThreeOfAKind [a,b,c,d,e]
ranking [[a,b],[c,d],[e]] = TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]]
  | value a - value e == 4 = Straight [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]]
  | value a == 14 && value b == 5 && flush [a,b,c,d,e]
    = StraightFlush [b,c,d,e,a]
ranking [[a],[b],[c],[d],[e]]
  | value a == 14 && value b == 5 = Straight [b,c,d,e,a]
ranking [[a],[b],[c],[d],[e]]
  | flush [a,b,c,d,e] = Flush [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]
```

■ It works, but it's ugly.

Yes. How could you avoid repeating yourself?

I don't know.

If we add the case for a general *Straight Flush*, it will make things worse.

I know.

Do you notice something specific about uses of the *flush* function?

It occurs for only one group pattern: `[[a],[b],[c],[d],[e]]`.

What `Hand` values this group pattern produce?

`HighCard` or `Straight`.

And what should it produce when the function *flush* yields `True` for the cards in the groups?

`Flush` or `StraightFlush`.

What should be produced for other groups when the function *flush* yields `True`?

That's not possible. There's no two cards of the same value in a flush.

Can you draw a table?

initial hand	flush	result
HighCard	True	Flush
Straight	True	StraightFlush
other	True	impossible
HighCard	False	HighCard
Straight	False	Straight
other	False	unchanged

Can you transform this table into a function?

Yes:

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

■ if that's what you mean.

That's what I mean. Can you use it in the *hand* function now?

Yes:

```
hand :: String → Hand
hand = promoteFlush . ranking .
      rSortBy (comparing length) .
      groupBy (same value) .
      rSortBy (comparing value) . cards
```

■ The code is still working.

Now we can get rid of the flush tests in the *ranking* function.

You are right:

```
ranking :: [[Card]] → Hand
ranking [[a,b,c,d],[e]] = FourOfAKind [a,b,c,d,e]
ranking [[a,b,c],[d,e]] = FullHouse [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = ThreeOfAKind [a,b,c,d,e]
ranking [[a,b],[c,d],[e]] = TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]]
  | value a - value e == 4 = Straight [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]]
  | value a == 14 && value b == 5 = Straight [b,c,d,e,a]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]
```

■ Nicely done.

And this should also work for the general case of *Straight Flush*, as this test will show:

```
, "6♥ 5♥ 4♥ 3♥ 2♥" 'beat' "A♦ A♠ A♥ A♣ K♥"]
```

■ Already passing! I think we should keep it, thought.

I agree.

Can you draw another decision table for detecting the *Straight* hand?

Here it is:

initial hand	a-e == 4	a==14 && b==5	result
HighCard	True	False	Straight [a,b,c,d,e]
HighCard	False	True	Straight [b,c,d,e,a]
other	True	False	unchanged
other	False	True	unchanged

Can you design a function from this table ?

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

■ Done.

Then use it in the *hand* function ?

```
hand :: String → Hand
hand = promoteFlush . promoteStraight . ranking .
  rSortBy (comparing length) .
  groupBy (same value) .
  rSortBy (comparing value) . cards
```

■ Done.

Then simplify the *ranking* function.

```
ranking :: [[Card]] → Hand
ranking [[a,b,c,d],[e]] = FourOfAKind [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = FullHouse [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = ThreeOfAKind [a,b,c,d,e]
ranking [[a,b],[c,d],[e]] = TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]
```

■ Done.

How could we make the *hand* function more legible?

Maybe by stating a step per line:

```
hand :: String → Hand
hand = promoteFlush
  . promoteStraight
  . ranking
  . rSortBy (comparing length)
  . groupBy (same value)
  . rSortBy (comparing value)
  . cards
```

■ Like this.

Maybe writing the steps in reverse order would read more naturally?

I'm not sure if we can do that.

We can if we reverse the `(.)` function. *GHCI* shows us how to do:

```
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
> :t (flip (.))
(flip (.)) :: (a -> b) -> (b -> c) -> a -> c
```

We need a new operator, then:

```
(>>.) :: (a -> b) -> (b -> c) -> (a -> c)
(>>.) = flip (.)
```

That's right.

■ And we can apply it to *hand*:

```
hand :: String -> Hand
hand = cards
    >>. rSortBy (comparing value)
    >>. groupBy (same value)
    >>. rSortBy (comparing length)
    >>. ranking
    >>. promoteStraight
    >>. promoteFlush
```

■ Now the code is clearer.

Yes. Here's the test code:

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

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♥"]
    where beat h g = comparing hand h g ~?= GT
```

And this is the tested code:

```
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 = HighCard [Card]
           | Pair [Card]
           | TwoPairs [Card]
           | ThreeOfAKind [Card]
           | Straight [Card]
           | Flush [Card]
           | FullHouse [Card]
           | FourOfAKind [Card]
           | StraightFlush [Card]
              deriving (Ord, Eq)

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 (.)

hand :: String -> Hand
hand = cards
    >>. rSortBy (comparing value)
    >>. groupBy (same value)
    >>. rSortBy (comparing length)
    >>. ranking
    >>. promoteStraight
    >>. promoteFlush

ranking :: [[Card]] -> Hand
ranking [[a,b,c,d],[e]] = FourOfAKind [a,b,c,d,e]
ranking [[a,b,c],[d,e]] = FullHouse [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = ThreeOfAKind [a,b,c,d,e]
ranking [[a,b],[c,d],[e]] = TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = HighCard [a,b,c,d,e]

cards :: String -> [Card]
cards = map card . words

promoteStraight :: Hand -> Hand
promoteStraight (HighCard [a,b,c,d,e])
    | value a - value e == 4 = Straight [a,b,c,d,e]
promoteStraight (HighCard [a,b,c,d,e])
    | value a == 14 && value b == 5 = Straight [b,c,d,e,a]
promoteStraight h = h

promoteFlush :: Hand -> Hand
promoteFlush (HighCard cs) | flush cs = Flush cs
promoteFlush (Straight cs) | flush cs = StraightFlush cs
promoteFlush h = h

```

---

We have a done a lot of work! What would you like to do now ?

Let's take a walk.

## 7 Printing

---

What should we work on, now ?

Let's do something that is easy, for a change.

---

What about printing the rankings ?

That will be short and sweet.

---

Do you remember what the program is expected to print?

Not much.

---

Here's an example:

```
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♦
```

I see. We need to print:

- the line of cards we have in input
- the ranking of the hand found in the line
- the mention "(winner)" along with the best ranking

---

Let's take care of your second item: showing the ranking.

Ok.

---

Here's a test:

```
showRanking (hand "6♣ 4♦ A♠ 3♠ K♠") ~?= "High Card"
```

Easy:

```
showRanking :: Hand → String
showRanking _ = "High Card"
```

■ And the test passes.

---



Here's another test, then:

```
showRanking (hand "5♥ 2♦ 3♥ 4♦ 2♥") ~?= "Pair"
```

```
showRanking :: Hand → String
showRanking (Pair _) = "Pair"
showRanking _ = "High Card"
```

■ Done. That's easy.

Yes, easy, and tedious. Could we skip the testing part on that feature?

Not if we abide by the rule #1 of TTD.

Which is?

You are not allowed to write any production code unless it is to make a failing unit test pass.

But I don't want to create all these hands just so that we can test the label given to the ranking.

Then just test the label given to the ranking.

You mean I should write my tests like this:

```
,showRanking HighCard ~?= "High Card"
,showRanking Pair ~?= "Pair"
```

■ No, that's not right. You can't use these data constructor without a list of `Cards`. But an empty list should do the trick.

It doesn't sound right, though. Look at the message:

```
Couldn't match expected type 'Hand'
against inferred type '[Card] -> Hand'
```

Let's try:

```
,showRanking (HighCard []) ~?= "High Card"
,showRanking (Pair []) ~?= "Pair"
```

■ Yes, that's better.

In that case, I'd rather create a single test for all ranking labels:

```
map showRanking [HighCard [],
                  Pair [],
                  TwoPairs [],
                  ThreeOfAKind [],
                  Straight [],
                  Flush [],
                  FullHouse [],
                  FourOfAKind [],
                  StraightFlush []] ~?=
["High Card", "Pair", "Two Pairs", "Three of a Kind",
 "Straight", "Flush", "Full House",
 "Four of a Kind", "Straight Flush"]
```

Ok. Here the function `showRanking`:

```
showRanking :: Hand → String
showRanking (HighCard) = "High Card"
showRanking (Pair _) = "Pair"
showRanking (TwoPairs _) = "Two Pairs"
showRanking (ThreeOfAKind _) = "Three of a Kind"
showRanking (Straight _) = "Straight"
showRanking (Flush _) = "Flush"
showRanking (FullHouse _) = "Full House"
showRanking (FourOfAKind _) = "Four of a Kind"
showRanking (StraightFlush _) = "Straight Flush"
```

■ And your big test is passing. But this is not quite satisfying.

Agreed. The test is not as expressive as it should be. What we want to express is that, for example: *the keyword `FourOfAKind` should be displayed as "Four of a Kind".*

Then you can change the tests.

Allright.

```
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"]
```

■ This provokes an error:

No instance for (Show ([Card] -> Hand))

■ Data constructor like *HigCard* or *Pair* are really functions. And we cannot make a function *Showable*.

What should we do then?

Create a data type for these values:

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

■ This is only the first step.

Indeed. Now we have *multiple declarations* errors: each value is declared in both *Hand* type and *Ranking* type.

We don't need any more to have them in the *Hand* type.

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

■ Now creating a *Hand* is done with the data constructor *H*, followed by a *Ranking* and a list of *Cards*. Of course this is only the second step.

The *ranking* function is broken:

Couldn't match expected type '[Card] -> Hand'  
against inferred type 'Ranking'  
In the expression: FourOfAKind [a, b, c, d, ....]

■ To fix this, we need to use *H*, the new data constructor:

```
ranking :: [[Card]] -> Hand
ranking [[a,b,c,d],[e]] = H FourOfAKind [a,b,c,d,e]
ranking [[a,b,c],[d,e]] = H FullHouse [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = H ThreeOfAKind [a,b,c,d,e]
ranking [[a,b],[c,d],[e]] = H TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = H Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = H HighCard [a,b,c,d,e]
```

We have the same error in functions *promoteStraight* and *promoteFlush*.

■ We'll apply the same fix:

```
promoteStraight :: Hand -> Hand
promoteStraight (H HighCard [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
```

Yes:

```

Couldn't match expected type 'Hand'
  against inferred type 'Ranking'
In the pattern: HighCard _
In the definition of 'showRanking':
  showRanking (HighCard _) = "High Card"

```

Yes, *showRanking* is not correct any more. First we have to declare *Ranking* to be an instance of the class *Show*. Then we have to override the *show* function for *Ranking* values.

```

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"

```

■ And we're done.

Now that the tests are passing, we should refactor the code.

You are right. Let's begin with the *ranking* function:

```

ranking :: [[Card]] → Hand
ranking [[a,b,c,d],[e]] = H FourOfAKind [a,b,c,d,e]
ranking [[a,b,c],[d,e]] = H FullHouse [a,b,c,d,e]
ranking [[a,b,c],[d],[e]] = H ThreeOfAKind [a,b,c,d,e]
ranking [[a,b],[c,d],[e]] = H TwoPairs [a,b,c,d,e]
ranking [[a,b],[c],[d],[e]] = H Pair [a,b,c,d,e]
ranking [[a],[b],[c],[d],[e]] = H HighCard [a,b,c,d,e]

```

We should change its name, because a function called *ranking* should be about extracting the *Ranking* value from a *Hand*.

I agree.

What would be a good name for a function that ranks a list of cards ?

That would be *rank*:

```

hand :: String → Hand
hand = cards
      >>. rSortBy (comparing value)
      >>. groupBy (same value)
      >>. rSortBy (comparing length)
      >>. rank
      >>. promoteStraight
      >>. promoteFlush

rank :: [[Card]] → Hand
rank [[a,b,c,d],[e]] = H FourOfAKind [a,b,c,d,e]
rank [[a,b,c],[d,e]] = H FullHouse [a,b,c,d,e]
rank [[a,b,c],[d],[e]] = H ThreeOfAKind [a,b,c,d,e]
rank [[a,b],[c,d],[e]] = H TwoPairs [a,b,c,d,e]
rank [[a,b],[c],[d],[e]] = H Pair [a,b,c,d,e]
rank [[a],[b],[c],[d],[e]] = H HighCard [a,b,c,d,e]

```

■ There.

Something is bothering me: the *rank* function is not *DRY*.

Yes. Since we list the cards along with every *Ranking* value, we can do that once in the main body of the function, and calculate the ranking in an auxiliary function. Thus we are separating concerns.

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

■ As you probably know, *concat* concatenates several lists into one.

Ok. Here's the test code:

```
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"]
  ]
where beat h g = comparing hand h g ~?= GT
```

And here's the tested code:

```
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)

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 (.)

hand :: String → Hand
hand = cards
  >>. 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 Straight [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

```

## 8 Finding Hands

---

We know how to compute a hand's ranking, and print that ranking. What do we need to do now?

We need to find the five card hand with the best ranking in a list of cards.

---

How do we do that?

Just write a failing test.

Ok. Here we go:

```
maxRanking "6♥ 6♦ 6♣ 6♠ K♠ K♦" ~?= Nothing
```

In that case, the result is `Nothing` because there are less than 7 cards in the string. You know about `Nothing`, right ?

■ Yes.

```
maxRanking :: String → Maybe Ranking
maxRanking _ = Nothing
```

■ Your test is implying that `maxRanking` consumes a `String` and returns, `Maybe`, a `Ranking`.

That is correct. Here's another one:

```
maxRanking "6♣ 4♦ A♣ 3♠ K♠ T♦ 8♣" ~?= Just HighCard
```

■ I'll make it pass as fast as I can:

```
maxRanking :: String → Maybe Ranking
maxRanking s | length (cards s) < 7 = Nothing
maxRanking _ = Just HighCard
```

■ We just ignore lists of less than 7 cards.

Ok. But there is still a *fake*. Here's a new test:

```
maxRanking "6♣ 6♦ A♣ 3♠ K♠ T♥ 8♦" ~?= Just Pair
```

Easy: we just return the ranking of the list of cards

```
maxRanking :: String → Maybe Ranking
maxRanking s | length (cards s) < 7 = Nothing
maxRanking s = Just $ ranking $ hand s
```

■ Uh oh.

---

Not in scope: ranking

We don't have a function *ranking*. We had one, but we renamed it.

Ok, here's the needed function:

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

■ But the test still doesn't pass.

Non-exhaustive patterns in function calcRank

We have several messages like this.

Of course, that function recognizes only five card patterns:

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

■ and I fed it with a seven card list.

What should we do then?

To make this test pass, let's cut the list and take only five cards. This means taking only the first 14 chars of the string:

```
maxRanking :: String → Maybe Ranking
maxRanking s | length (cards s) < 7 = Nothing
maxRanking s = Just $ ranking $ hand $ take 14 s
```

■ See? It works.

That's not very clear. Couldn't we just say:  
*hand* \$ take 5 \$ cards s  
instead?

We can always write it:

```
maxRanking :: String → Maybe Ranking
maxRanking s | length (cards s) < 7 = Nothing
maxRanking s = Just $ ranking $ hand $ take 5 $ cards s
```

■ But it doesn't work because *hand* consumes *Strings*, not lists of *Cards*.

Then make it accept a list of *Cards*.

■ Ok. Heres the *hand* function:

```
hand :: String → Hand
hand = cards
  >>. rSortBy (comparing value)
  >>. groupBy (same value)
  >>. rSortBy (comparing length)
  >>. rank
  >>. promoteStraight
  >>. promoteFlush
```

■ So, I change the signature, and get rid of the first call to *cards*, this call will be made somewhere above.:

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

■ We still have an error.

Couldn't match expected type 'Card' against inferred type 'Char'

It's in the tests. I need to change this line:

```
beat h g = comparing hand h g ~?= GT
```

like this:

```
beat h g = comparing (hand . cards) h g ~?= GT
```

■ And everything is back to normal.

Good.

We should also change the signature of our function `maxRanking` so that it also receive a `[Card]`. I will change my test to that effect:

```
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]
where beat h g = comparing (hand . cards) h g ~?= GT
      maxRank = maxRanking . cards
```

■ I now change the code of the function:

```
maxRanking :: [Card] → Maybe Ranking
maxRanking cs | length cs < 7 = Nothing
maxRanking cs = Just $ ranking $ hand $ take 5 $ cs
```

■ And the refactoring is done. Now we need to replace our *fake* with the general implementation.

There are several possible 5 card hands we can form with a list of 7 cards. Do you know how much?

Yes,  $\binom{7}{5} = \frac{7 \times 6 \times 5 \times 4 \times 3}{5 \times 4 \times 3 \times 2 \times 1} = \frac{2520}{120} = 21$

Do you know how to find them?

Yes. I can use the `subsequences` function. For example `subsequences "CAT"` gives:

```
[ "", "C", "A", "CA", "T", "CT", "AT", "CAT"]
```

What if we want only two letters subsequences ?

I suppose applying `filter ((2==) . length)` on the list would do the trick.

Then we have to find the best hand.

Oh, that's the simplest part.

Let's begin with that part, then.

Allright. Suppose we have computed some sublists already:

```
maxRanking :: [Card] → Maybe Ranking
maxRanking cs | length cs < 7 = Nothing
maxRanking cs =
  let sl = [drop 2 cs, take 5 (drop 1 cs), take 5 cs]
  in Just $ ranking $ hand $ take 5 $ cs
```

■ Here we don't compute all the 21 sublists, we create 3 of them.

Given our last test:

```
maxRank "6♣ 6♦ A♠ 3♠ K♠ T♥ 8♦" ~?= Just Pair
```

the 3 sublists would give these hands:

A♠ 3♠ K♠ T♥ 8♦

6♦ A♠ 3♠ K♠ T♥

6♣ 6♦ A♠ 3♠ K♠.

Only the last hand would rank as a *Pair*.

That's right.



Go on.

Then, to find the best ranking hand from this list is easy:

```
maxRanking :: [Card] → Maybe Ranking
maxRanking cs | length cs < 7 = Nothing
maxRanking cs =
  let sl = [drop 2 cs, take 5 (drop 1 cs), take 5 cs]
      max = maximum . map ranking . map hand
  in Just $ max sl
```

■ And we're done for that part.

Ok. Now how do we compute the sublists?

Hmm. First we need a helper function to create 5 item sublists from a list

Ok. What about a function such as:

```
subLists 2 "CAT" ~?= ["CA","CT","AT"]
```

I introduce a variable, because I don't really want to write the case for 21 sublists.

■ Nice idea. Here's the function:

```
subLists :: Int → [a] → [[a]]
subLists n = filter ((n ==) . length) . subsequences
```

■ It's a bit more general than needed, thought.

Ok. Now use the function to find hands.

Allright:

```
maxRanking :: [Card] → Maybe Ranking
maxRanking cs | length cs < 7 = Nothing
maxRanking cs =
  let sl = subLists 5 cs
      max = maximum . map ranking . map hand
  in Just $ max sl
```

■ It works!

Yes. Can you clean up the code? I will delete the test about `subLists`, as I think this function is just a helper.

Right:

```
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
```

■ And we're done.

## 9 Solving the problem

---

Did we solve our problem?

Not yet.

---

What do we need to do, then?

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

---

What test should I write?

Write the simplest test you can think of.

---

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

No hand at all.

Ok

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

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

■ I see. Here's the function:

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

■ Done.

---

Here's my next case.

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

■ 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?

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-exhaustive patterns* error in our code, now.

■ 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*?

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?

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`?

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.

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?

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)"]
```

Wow. This is a big test!

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

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"
```

■ 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♠ "

■ 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*.

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?

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?

just type:

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

---

Here's the output:

■ ■ ■ It works! Hurray!

```
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.

---

Let's ship our program to the customer

And have a fantastic dinner!