Poker Hand

by ToF

October 29, 2011

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

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 wich, 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 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]
```

What is the result?

■ 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 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]
```

What now?

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

```
module PokerHand
where

card :: String → Card
```

■ 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
```

Can you make it pass ?

■ I don't think so.

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:

```
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
```

This last test is a bit long.

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 a 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 flipping 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
```

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

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

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 thant *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♦ T♥"
    ,"5♥ 4♦ 3♥ 2♦ 3♣" `beat` "A♥ K♥ Q♦ J♦ T♥"]
```

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

■ And this is the tested code: first the type declarations:

```
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)
```

What about the functions ?

Here they are:

```
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 Comparing Pairs

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♦"]
```

■ Those 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
```

■ Still failing.

■ 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*. Everything would be fine if we always 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, descending:

```
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
```

■ Now we have errors.

That's because we now have non-exhaustive patterns in our three last tests.

■ Of course. Let's put the only remaining possible pair pattern:

```
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 everything is fine again.