Poker Hand

by ToF

October 26, 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 fail:

```
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.