The Texas Hold'em Kata

given this file: `input.txt` :

```
kc 9s ks kd 9d 3c 6d
9c ah ks kd 9d 3c 6d
ac qc ks kd 9d 3c
9h 5s
4d 2d ks kd 9d 3c 6d
7s ts ks kd 9d
```

after command

`runhaskell pokerhands.hs <input.txt`

then the output is

```
kc 9s ks kd 9d 3c 6d full house (winner)
9c ah ks kd 9d 3c 6d two pair
ac qc ks kd 9d 3c
9h 5s
4d 2d ks kd 9d 3c 6d flush
7s ts ks kd 9d
```

The Texas Hold'em Kata

in the line:

```
8s 9d Th Js Qd Kc Ah
```

`T`,`J`,`Q`,`K`,`A` stand for *Ten*, *Jack*, *Queen*, *King*, *Ace*, and
`h`,`s`,`d`,`c` stand for *Hearts*, *Spades*, *Diamonds*, *Clubs*

Texas Hold'em in five steps:

1. Interpret Strings in terms of Cards
2. Compare Cards (by Rank or by Suit)
3. Find the Category of a Hand (Hand = group of 5 Cards)
4. Find the best possible Hand in a group of 7 Cards
5. Find the best player in a game

program = function evaluation

Lauch *ghci* and try some functions:

```
sqrt 1764 ↩

Data.List.subsequences "ABCD" ↩

subtract 2 44 ↩

 2 `subtract` 44 ↩

subtract 1 (subtract 1 44) ↩

6 * (3 + 4) ↩

(*) 6 ((+) 3 4) ↩

Data.List.insert 42 [1,32,87] ↩
```

Writing a test

A short program named `Specs.hs`:

```
import Test.Hspec

main = hspec
(describe "a test"
 (it "should pass"
  (2+2  `shouldBe` 4)))
```

Running the test:

```
runhaskell Specs.hs ↩
```

Writing a suite of tests

Sequencing actions with `do`:

```
import Test.Hspec

main = hspec $ do
    describe "a suite" $ do
        it "should pass" $ do
            2+2 `shouldBe` 4
        it "should not fail" $ do
            2*2 `shouldBe` 4
```

- the `$` operator is an alternative to parentheses:
- `f $ x y z` $\equiv$ `f (x y z)`
- the `do` construct allows for sequencing of actions
- the actions must be indented under their sequencing do
- we will use `do` and actions only in the tests

Let's write some functions

Write a function *response* that passes this test:

```
import Test.Hspec

main = hspec \$ do
    describe "response" $ do
        it "should be a yes or a no" \$ do
            response 'N' `shouldBe` False
            response 'n' `shouldBe` False
            response 'Y' `shouldBe` True
            response 'y' `shouldBe` True
```

Pattern Matching

```
response 'Y' = True
response 'y' = True
response 'N' = False
response 'n' = False
```

Patterns allow for expressing distinct cases

Pattern Matching

Write a function label that passes this test:

```
import Test.Hspec

main = hspec $ do
    describe "label" $ do
        it "should be a yes or a no" $ do
            label "WO" = "Wool"
            label "CO" = "Cotton"
            label "PA" = "Nylon"
            label "PC" = "Acrylic"
            label "XX" = "--- unknown label ---"
            label "YY" = "--- unknown label ---"
```

Pattern Matching

```
label "WO" = "Wool"
label "CO" = "Cotton"
label "PA" = "Nylon"
label "PC" = "Acrylic"
label   _  = "--- unknown label ---"
```

The underscore symbol in the left part of the equality denotes *any value that is distinct from the values in the preceding patterns*.

Lists

A way to collect values of the same type

Ghci:

```
1 : 2 : 3 : [] ↩

'a' : 'b' : 'c' : "" ↩

[4,8] ++ [0,7] ↩

head [4,8,0,7] ↩

tail [4,8,0,7] ↩

reverse "Hello World" ↩

concat ["A","List","Of","Lists"] ↩
```

Let's write some functions

Write a function *average* that passes this test:

```
describe "average" $ do
    it "should calculate the average" $ do
        average [ ]          `shouldBe` 0
        average [2, 4, 12] `shouldBe` 6
```

Let's write some functions

using Pattern Matching to denote cases:

```
average [ ]  = 0
average xs   = sum xs `div` length xs
```

A variable defined in the left part of the equality receives the argument value and can be used in the right part.

Pattern Matching

```
ordered [a,b]   = a <= b
ordered [a,b,c] = ordered [a,b] && ordered [b,c]

product []     = 1
product (x:xs) = x * product xs
```

Patterns also allow for deconstructing data:

- elements of a list
- head of a list and remaining list

Comparing values

Some useful checks about compare :

```
describe "compare" $ do
    it "should compare values of any type of class
    Ord" $ do
        compare 42 17         `shouldBe` GT
        compare 'A' 'B'       `shouldBe` LT
        compare 11.3 11.3     `shouldBe` EQ
        compare "cat" "dog"   `shouldBe` LT
```

Strings are not Cards!

There's no way that this test can pass:

```
describe "using Strings as Cards" $ do
    it "cannot give satisfactory comparisons" $ do
        compare "Td" "Jc"  `shouldBe` LT
        compare "8d" "8c"  `shouldBe` EQ
        compare "Ah" "Jc"  `shouldBe` GT
```

unless we rewrite `compare`

How to compare cards by rank ?

Write a function rank that passes this test:

```
describe "comparing card by rank" $ do
    it "should follow the rules of poker" $ do
        compare (rank "8d") (rank "6h") `shouldBe` GT
        compare (rank "4d") (rank "4h") `shouldBe` EQ
        compare (rank "9d") (rank "Th") `shouldBe` LT
        compare (rank "Td") (rank "Jh") `shouldBe` LT
        compare (rank "Jd") (rank "Qh") `shouldBe` LT
        compare (rank "Qd") (rank "Kh") `shouldBe` LT
        compare (rank "Kd") (rank "Ah") `shouldBe` LT
```

Hint:

```
rank ['A',_] = 14
rank ['K',_] = 13
    . . .
```

How to compare cards by suit

Write a function `suit` that passes this test

```
describe "comparing card by suit" $ do
    it "should follow the rules of poker" $ do
        suit "8d" == suit "6d" `shouldBe` True
        suit "4d" == suit "4h" `shouldBe` False
        suit "9d" == suit "Tc" `shouldBe` True
        suit "Td" == suit "Js" `shouldBe` False
```

Types

Types are a way to check the meaning of programs
All expressions, all function definitions have a type.
Although Haskell can infer our types, we can explicitely declare function signatures:

```
rank :: String →  Int
suit :: String →  Char
```

Types

Thanks to types, expressions like

- rank False
- rank 3.1415

are not legal But:

- rank "Foo" is still legal
- compare (rank "!*") (rank "18") == ... ?
- every String value is not a valid Card value
- only when comparing fails we know we had incorrect data

Tuples

A way to gather values of different types Ghci:

```
:type (EQ,'@', False) ↩

:type ('A',True) ↩

:type fst ↩

:type snd ↩

fst ('A', True) ↩

snd ('A', True)  ↩
```

a way to think about the problem

Let's define types synonyms:

```
type Card = (Rank, Suit)
type Rank = Int
type Suit = Char

rank :: Card →  Rank
suit :: Card →  Suit
```

And a new function from String to Card :

```
card :: String →  Card
```