Slices of Functional Programming

The Texas Hold'em Kata

Xavier Detant & Christophe Thibaut

April 2, 2017

`git@github.com:ToF-/SlicesOfFP.git`
`SlicesOfFP/code/`

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

Patterns allow for expressing distinct cases

Pattern Matching

Write a function `label` that passes this test:

```haskell
import Test.Hspec

main = hspec $ do
    describe "label" $ do
        it "should be an english label" $ do
            label "WO" `shouldBe` "Wool"
            label "CO" `shouldBe` "Cotton"
            label "PA" `shouldBe` "Nylon"
            label "PC" `shouldBe` "Acrylic"
            label "XX" `shouldBe` "--- unknown label ---"
            label "YY" `shouldBe` "--- 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:

```
import Test.Hspec

main = hspec $ do
    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 :

```
import Test.Hspec

main = hspec $ do
    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:

```
import Test.Hspec

main = hspec $ do
    describe "compare" $ do
        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:

```
import Test.Hspec

main = hspec $ do
    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

```haskell
import Test.Hspec

main = hspec $ do
    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` False
            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:

```
import Test.Hspec

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

And a new function from String to Card :

```
card :: String → Card
```

Comparing cards, improved

Write the function: card :: String → Card so that the test pass

```
import Test.Hspec

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

Comparing cards, improved

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

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

Better because:

- once conversion is done, the comparing takes care of itself
- bad input is detected at conversion, not in comparisons

But:

- you can still do silly things like rank (4807,'@')

Type Class = a way to define type conformity

Saying that

```
data Rank = Two | Three | Four | Five | Six | Seven |
    Eight
          | Nine | Ten | Jack | Queen | King | Ace
    deriving (Eq, Ord, Enum, Show)
```

means that values of type Rank

- can be compared with == and /=
- can be compared with compare , < , <= ...
- can be converted to and fron Int with fromEnum and toEnum
- can be converted to String with show

Types = a way to think about a problem

Let's create new types:

```
data Suit = Hearts | Clubs | Diamonds | Spades
    deriving (Eq, Show)

data Rank = Two | Three | Four | Five | Six | Seven |
    Eight
          | Nine | Ten | Jack | Queen | King | Ace
    deriving (Eq, Ord, Enum, Show)

type Card = (Rank, Suit)
```

Rewrite the card function so that the tests still pass Hint:

```
card [r,s] = (charToRank c, charToSuit s)

charToRank 'A' = toEnum 12
charToRank 'K' = toEnum 11
...
```

Type Class = a way to define type conformity

Ghci:

```
:load PokerHand.hs ↩

Two < Three ↩
Ace > King ↩
show Queen ↩
card "8d" ↩
```

Better design:

- the type `Card` can have only 52 values.
- once conversion is done, you can only
  - compare by rank order (no illegal rank allowed)
  - compare on equality by suit (no illegal suit allowed)

Checkpoint #1

We have the proper types to describe our values

We have our first feature: comparing cards

Well Done!!

Organizing Code in Modules

Let's move the production code into its own *module*:

```
module PokerHand
where
```

and use it in the Specs script:

```
import Test.Hspec
import PokerHand
```

Passing Functions to Functions

Ghci:

```
import Data.Ord ↩

:type compare ↩

:type comparing ↩

comparing abs (-4) 3 ↩

:load PokerHand.hs ↩

comparing rank (card "8c") (card "5d") ↩
```

the function `rank` is passed to the `comparing` function

Combining Functions

Ghci:

```
:type (.) ↩

(length . words) "time flies like an arrow" ↩

comparing (rank . card) "8c" "5d" ↩
```

(f . g) x == f (g x)

Combining Functions

Refactor the test using `comparing` and the `.` operator

```
import Test.Hspec

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

Mapping a function to a list of values

Ghci:

```
:type map ↩
map negate [-34,42,17] ↩

map sqrt [1,2,3,4,5] ↩
```

Collecting Cards

Write the function `cards` such that

```
import Test.Hspec

main = hspec $ do
    describe "cards" $ do
        it "should collect cards from a string" $ do
            cards "8d Ah Qc" `shouldBe`
                 [(Eight,Diamonds),(Ace,Hearts),(Queen,
    Clubs)]
```

Sorting

```
import Data.List ↩

sort [42,3,17,1,22,4,38] ↩

sortBy compare "HELLO" ↩

sortBy (comparing length) (words "time flies like an arrow") ↩

:type flip ↩

flip compare 4 5 ↩

sortBy (flip compare) "HELLO" ↩
```

Ranks of a hand

Write the function ‘ranks‘ such that

```
import Test.Hspec

main = hspec $ do
    describe "ranks" $ do
        it "should give the sorted ranks of a hand" $ do
            ranks (cards "8d Ah Qc")  `shouldBe` [Ace,
    Queen, Eight]
```

Grouping

```
group "HELLO"

(group . sort) "Cats and Dogs"
```

Groups of Cards

Write the function ‘groups‘ such that

```
import Test.Hspec

main = hspec $ do
    describe "groups" $ do
        it "should group and sort the ranks of a hand" $
    do
            groups (cards "8d Ah Qc 8h 8s")  `shouldBe`
                [[Eight,Eight,Eight],[Ace],[Queen]]

            groups (cards "8d Ah Qc 8h As")  `shouldBe`
                [[Ace,Ace],[Eight,Eight],[Queen]]
```

Hint: use

- sort
- sortBy
- comparing
- group
- reverse

Categorizing groups of Cards

A data type for Category

```
data Category = HighCard | OnePair | TwoPairs |
    ThreeOfAKind
              | Straight | Flush | FullHouse | FourOfAKind
              | StraightFlush | RoyalFlush
    deriving (Eq,Ord,Show)
```

Categorizing groups of Cards

Write the function category :: [[Rank]] → Category

```haskell
import Test.Hspec

main = hspec $ do
    describe "category" $ do
        it "should determine the category of a hand" $ do
            let hs = ["4s 5d Kc Tc 3d"
                     ,"4s Kd Kc Tc 3d"
                     ,"4s Kd Kc Tc Td"
                     ,"Ts Kd Kc Kc 8d"
                     ,"Ts Kd Kc Tc Td"
                     ,"Ts Kd Kc Kc Kd"]
            map (category.groups.cards) hs ==
                    [HighCard, OnePair, TwoPairs
                    ,ThreeOfAKind, FullHouse, FourOfAKind]
```

Hint:

```haskell
category [_,_,_,_,_]   = HighCard
category [[_,_],_,_,_] = OnePair
...
```

Special categories

A Straight is like a HighCard with ranks forming a sequence

e.g. Th 9d 8c 7s 6s

A Flush is like a HighCard with all cards of same suit

e.g. Kh Jh 9h 7h 6h

Guards

Pattern matching can be applied with conditions, called guards

```
power n m | m >= 0     = product (replicate m n)
          | otherwise = error "negative exponent"

sign n | n < 0 = -1
       | n > 0 =  1
       | _     =  0
```

Detecting a Flush

Write the function isFlush :

```
import Test.Hspec

main = hspec $ do
    describe "isFlush" $ do
        it "should detect when all cards have the same
    suit" $ do
            isFlush (cards "8d Ah 4d 3d Ad") `shouldBe`
    False
            isFlush (cards "8h Ah 4h 3h Kh") `shouldBe`
    True
```

Hint: use

- group
- length
- pattern matching with guards

The Enum Type Class

Ghci:

```
fromEnum False ↩
fromEnum True ↩

:load PokerHand.hs ↩

fromEnum Ace ↩
fromEnum King ↩
```

Detecting a Straight

Method:

- Given a list of 5 distinct groups of 1 rank each,
- And the first rank value = the last rank value + 4
- Then the category is Straight

```
isStraight :: [Rank] → Bool
isStraight [a,_,_,_,b] = fromEnum a == 4 + fromEnum b
isStraight _                = False
```

Lexicographic Order

Tuples, like Lists can be compared according to lexicographic order:

$$(a, b) < (c, d) \equiv (a < c) \vee (a = c) \wedge (b < d)$$

$$[a, b] < [c, d] \equiv (a < c) \vee (a = c) \wedge (b < d)$$

This allows for comparing hand by category then ranks:

- If two hands have the same category, the winner is the hand with the highest rank in the category.
- If two hands have the same category and rank, the winner is the hand with the highest remaining cards.

Comparing two hands

Comparing two hands involves comparing their category, and if their categories are equal, comparing the ranks in the order given by the groups.

Creating values of type `Ranking` allows for such comparisons, provided that the ranks are sorted in reverse order.

```
import Test.Hspec

type Ranking = (Category, [Rank])

main = hspec $ do
    describe "Ranking" $ do
        it "should correctly compare two ranking values" $
     do
            (OnePair,[Ace,Ace,Ten,Eight,Five])
                > (OnePair, [Ace,Ace,Eight,Seven,Two])
                 `shouldBe` True
```

Determining a Ranking

Create the function:

```
ranking :: [Card] → Ranking
```

```
import Test.Hspec

main = hspec $ do
    describe "ranking" $ do
        it "should keep the ranking of a hand" $ do
            ranking (cards "2c 2s 3s 3c 4h")
                `shouldBe` (TwoPairs, [Three,Three,Two,Two
    ,Four])

            ranking (cards "2c 2s As 3c 4h")
                `shouldBe` (OnePair, [Two,Two,Ace,Four,
    Three])
```

Hint:

```
ranking cs = (cat,rs)
where
    cat = category gs
    rs  = concat gs
```

Special Categories (cont.)

A *Straight Flush* is a *Straight* and a *Flush*

e.g `Th 9h 8h 7h 6h`

A *Royal Flush* is a *Straight Flush* starting with an Ace

e.g. `Ah Kh Qh Jh Th`

Promoting to special categories

```
promote :: Ranking → Ranking
promote (HighCard,[Ace,Five,_,_,_]) = (Straight,
                                        [Five,Four,Three,Two
    ,Ace])
promote (HighCard,rs) | isStraight rs = (Straight, rs)
promote r = r

flushes :: Ranking → Bool → Ranking
flushes True (HighCard,rs) = (Flush, rs)
flushes True (Straight,[Ace,_,_,_,_]) = (RoyalFlush, [Ace,
    King,Queen,Jack,Ten])
flushes True (Straight,rs) = (StraightFlush rs)
flushes False r = r
```

Ranking Final Test

```
it "should correctly order a list by ranking" $ do
    let s = ["7s 5c 4d 3d 2c" ,"As Kc Qd Jd 9c"
            ,"2h 2d 5c 4c 3c" ,"Ah Ad Kc Qc Jc"
            ,"2c 2s 3s 3c 4h" ,"Ac As Ks Kc Jh"
            ,"2h 2d 2c 4c 3c" ,"Ah Ad Ac Qc Jc"
            ,"5h 4s 3d 2c Ah" ,"Ah Ks Qd Jc Th"
            ,"7c 5c 4c 3c 2c" ,"Ac Kc Qc Jc 9c"
            ,"2h 2d 2c 3h 3c" ,"Ah Ad Ac Kh Kc"
            ,"2c 2s 2h 2d 3c" ,"Ac As Ah Ad Jc"
            ,"5c 4c 3c 2c Ac" ,"Ah Kh Qh Jh Th"]
        isOrdered [_] = True
        isOrdered (x:y:xs) = x < y && isOrdered (y:xs)
        r = map (ranking.cards) s
    isOrdered r `shouldBe` True
```

Ranking Final Test

Hint:

```
ranking cs = flushes (isFlush cs) (promote (cat, rs))
where
cat = category gs
rs  = concat   gs
gs  = groups (ranks cs)
```

Checkpoint #2

We can compare two hands in Texas Hold'em

Well Done!!