Chapter - 7 : More functional Patterns.

*) Functions are defined by the fact that they can be applied to an argument & return a result.

*) A value that can be used as an argument to a fn is a first-class value. In Haskell, even functions can be passed around as arguments.

*) for type inference, the most polymorphic type that works is inferred.

*) we can use let expressions to declare & bind variables. for example

```
bindExp :: Integer → String
bindExp x =
    let y = 5 in
        "the integer was: " ++ show x
        ++ " and y was " ++ show y
```

y is only in scope for the let expression.

*) Haskell is lexically scoped, hence the definition of any variable that is innermost in the code takes precedence.

→ Anonymous function : using the lambda syntax represented by the backslash.

```
(\x → x * 3) :: Integer → Integer
```

to apply an anonymous fn, we'll often need to wrap it in parantheses so that our intent is clear.

```
(\x -> x * 3) 5
```

*) In GHCi error msgs, it refers to the last expression we entered.

*) The lambda based syntax is used a lot. Generally when we are passing functions as arguments to a higher order function. The way anonymous functions are evaluated might be another reason to use them in certain contexts.

→ Pattern Matching : an integral & ubiquitous feature of Haskell. It is a way of matching values against patterns, and where appropriate, binding values to successful matches.

↓

Patterns can include things as diverse as undefined variables, numeric literals, and list syntax i.e. pattern matching matches on any and all data constructors.

↓

Pattern matching allows us to expose data and dispatch different behaviours based on that data in our function definitions by deconstructing values to expose their innerworkings.

↓

Patterns are matched against values, or data constructors,

And not types. Matching a pattern might fail, proceeding to the next available pattern to match. When the match succeeds, the variables exposed in the pattern are bound. Pattern matching proceeds from left to right & outside to inside.

↓

can be done on numbers as well.

*) Use :{ & :} block syntax to write functions in GHCi.

```
:{
let isItTwo :: Integer → Bool
    isItTwo 2 = True
    isItTwo _ = False
:}
```

*) Ordering patterns from most specific to least specific.

*) Incomplete pattern matches applied to data they don't handle will return bottom, a non-value used to denote that the program cannot return a value or result. This will throw an exception, which if left unhandled, will make our program to fail.

*) Using -wall flag allows us to know at compile time, if pattern matching conditions are all exhaustive or missing some cases.

*) Pattern matching allows us to unpack & expose the contents of our data.

*) The values True & False don't have any other data to expose, but there exist some data constructors that have parameters, and pattern matching can let us expose & make use of the data in their arguments.

*) newType : special case of data declarations. newType is different in that it only permits only one constructor & only one field.

```haskell
-- registeredUser1.hs
module RegisteredUser where

newtype Username =
  Username String

newtype AccountNumber =
  AccountNumber Integer

data User =
    UnregisteredUser
  | RegisteredUser Username AccountNumber
```

*) With User, we can use pattern matching to accomplish two things. Since User is a sum with two constructors, UnregisteredUser and RegisteredUser, we can pattern match to dispatch our function differently depending on what value we get.

↓

Then with the RegisteredUser constructor, it's a product of two newtypes, Username & AccountNumber. We can use pattern matching to unpack its contents.

↓

for example :

```
printUser :: User -> IO ()
printUser UnregisteredUser =
    putStrLn "UnregisteredUser"
printUser (RegisteredUser
    (Username name)
    (AccountNumber acctNum)) =
    putStrLn $ name ++ " " ++ show AcctNum
```

(*) The type of RegisteredUser is a function that constructs a User of two arguments: Username & AccountNumber. This is what we mean when we refer to a value as a "data constructor".

(*) Another example:

```
data WherePenguinsLive =
    Galapagos
  | Antarctica
  | Australia
  | SouthAfrica
  | SouthAmerica
  deriving (Eq, Show)

data Penguin =
  Peng WherePenguinsLive
  deriving (Eq, Show)
```

product type

```
isSouthAfrica :: WherePenguinsLive → Bool
isSouthAfrica SouthAfrica = True
isSouthAfrica _ = False
```

used to indicate an unconditional match on a value, we can generalise over, or don't really care about.

*) using pattern matching to unpack Penguin values to get at the WherePenguinsLive value it contains:

```
gimmeWhereTheyLive :: Penguin
                   → WherePenguinsLive
gimmeWhereTheyLive (Peng whereItLives) =
    whereItLives
```

---

*) more elaborate example: expose the contents of Peng and match on what WherePenguinsLive values we care about in one pattern matching

```
galapagosPenguin :: Penguin → Bool
galapagosPenguin (Peng Galapagos) = True
galapagosPenguin _ = False
```

→ unpacking + specifying the value we want to unpack for.

```
antarcticPenguin :: Penguin → Bool
antarcticPenguin (Peng Antarctic) = True
```

```
antarcticPenguin _ = False

antarcticOrGalagpos :: Penguin -> Bool
antarcticorGalagpos a = (galapagosPenguin a) ||
                        (antarcticPenguin a)
```

*) Here we are using pattern matching to unpack the Penguin datatype, and specifying which where Penguins live value we want to match on.

→ Pattern matching tuple

```
f :: (a,b) -> (c,d) -> ((b,d),(a,c))
f (a,b) (c,d) = ((b,d),(a,c))

addEmUp2 :: Num a => (a,a) -> a
addEmp2 (x,y) = x + y

fst3 :: (a,b,c) -> a
fst3 (x,_,_) -> x

third 3 :: (a,b,c) -> c
third 3 (_,_,x) -> x
```

*) :browse to see a list of the type signatures and functions we looded from the module Tuplefunctions.

→ Exercises : variety Pack

① k (x, y) = x

k :: (a, b) → a

k2 = k ("three", (1+2)) :: String

② f :: (a, b, c)

→ (d, e, f)

→ ((a, d), (c, f))

f (a, b, c) (d, e, f) = ((a, d), (c, f))

→ Case Expressions : Similar to if-then-else expressions, of making the function return a different result based on different inputs. We can use case expressions with any datatype that has visible data constructors.

```
funcZ x =
  case x + 1 == 1 of
    True → "Awesome"
    False → "wut"

pal xs =
  case xs == reverse xs of
    True → "yes"
    False → "no"
```

↓

this can be rewritten as

↓

```
pal xs =
  case y of
    True → "yes"
    False → "no"
  where y = xs == (reverse xs)
```

*) Another example

```
greetIfCool :: String → IO()
greetIfCool coolness =
  case cool of
    True → putStrLn "eyyy. What's shakin'?"
    False → putStrLn "pshhh"
  where cool =
    coolness == "downright frosty yo"
```

→ Execcise : Case Practice

1.
```
function C x y =
  case x > y of
    True → x
    False → y
```

2.
```
ifEvenAdd2 n =
  case c of
    True → (n+2)
    False → n
```

where c = even n

→ Higher order functions : functions that accept functions as arguments. for example : flip

```
flip :: (a → b → c) → b → a → c
```

↓

takes a fn, as argument along on with two arguments b & a, and then the fn is applied after flipping i.e. a → b → c rather than b → a → c., to return c.

↓

```
flip f x y = f y x
```

the parantheses must be used to express a function argument in the function type

*) making use of compare as a HOF

```
data Employee = Coder
              | Manager
              | VeeP
              | CEO
              deriving (Eq, Ord, Show)
```

defines the order from top to bottom in ascending order

```
reportBoss :: Employee → Employee → IO()
reportBoss e e' = putStrLn $ show e ++ "is the boss of"
                  ++ show e'
```

```
employeeRank :: Employee → Employee → IO ()
employeeRank e e' =
    case compare e e' of
        GT → reportBoss e e'
        EQ → putStrLn "Equal"
        LT → reportBoss e e'
```

---

*) employeeRank function can be changed to the following, to apply something neat :)

```
employeeRank :: (Employee → Employee → Ordering)
                → Employee → Employee → IO ()
employeeRank f e e' =
    case f e e' of
        GT → reportBoss e e'
        EQ → putStrLn "Equal"
        _  → (flip reportBoss) e e'
```

(function argument) ←

this function can how be compare or a cheeky function to implement some specific functionality

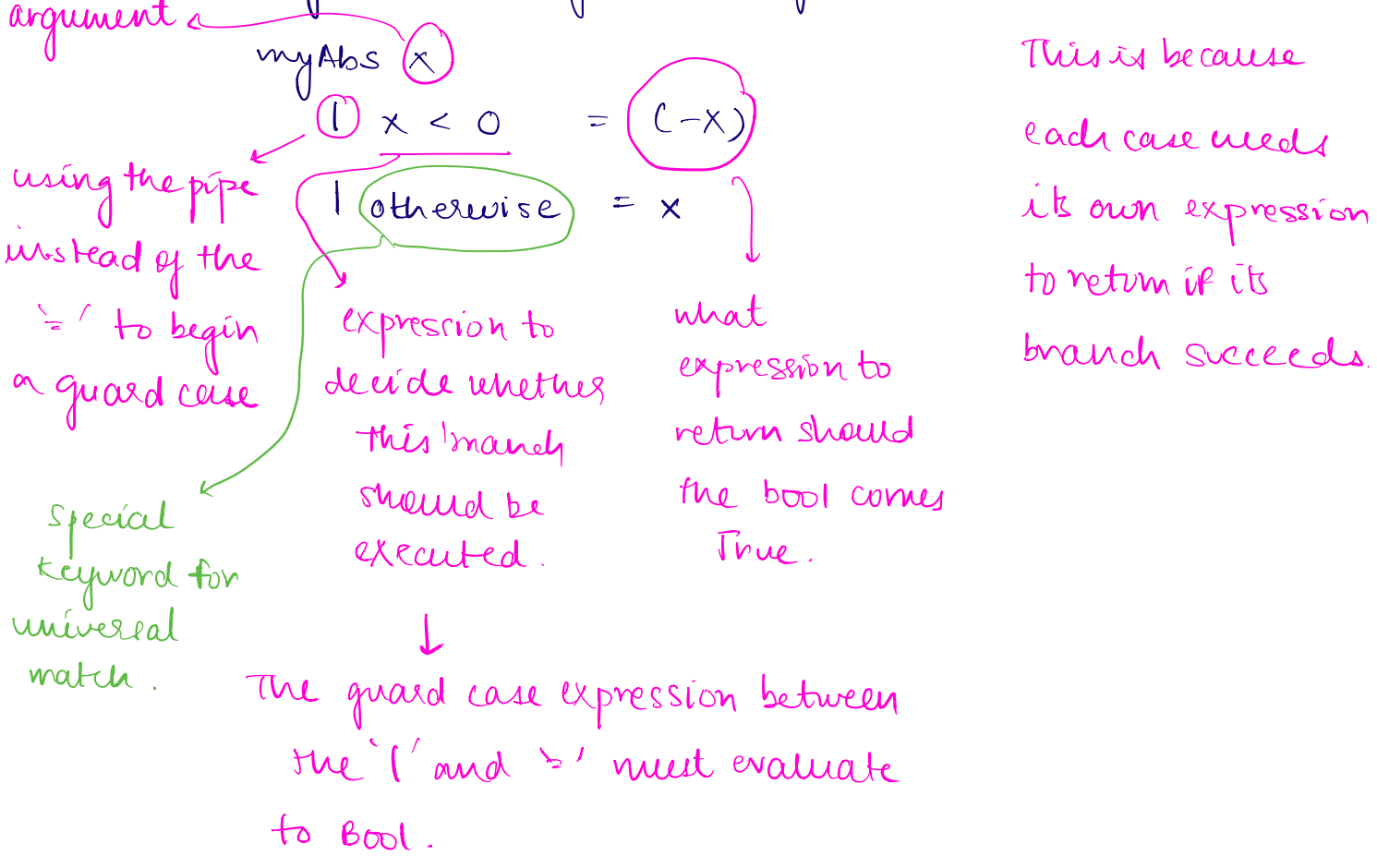
trying to drive the concept home imo.

```
codeesRuleCEOsDrool :: Employee →
                       Employee → Ordering
codeesRuleCEOsDrool codeE codeE' = EQ
codeesRuleCEOsDrool codeE _ = GT
codeesRuleCEOsDrool _ codeE' = LT
codeesRuleCEOsDrool e e' = compare e e'
```

default behaviour incase codee isn't an argument ←

*) Hence HOFs give us a big amount of flexibility on how to use & write code, in terms of reusability and composing of code.

→ Guards: another syntactic pattern called guards that relies on truth values to decide between two or more possible results. example.



↓

The guard case expression between the '|' and '=' must evaluate to Bool.

*) Guards always evaluate sequentially, so our guards should be ordered accordingly.

*) we can also use 'where' declarations within guard blocks.

```
avgGrade :: (Fractional a, Ord a)
         => a -> Char

avgGrade x
  | y >= 0.9 = 'A'
  | y >= 0.8 = 'B'
  | y >= 0.7 = 'C'
  | y >= 0.59 = 'D'
  | otherwise = 'F'
  where y = x / 100
```

→ in scope for all guards.

→ function composition : type of HOF that allows us to combine functions such that the result of applying one function gets passed to the next function as argument.

$$(.) :: (b \to c) \to (a \to b) \to a \to c$$

The result of $(a \to b)$ is argument to $(b \to c)$ to get the final resultant c.

*) Basic syntax around function composition

$$(f \circ g)\ x = f\ (g\ x)$$

composition operator (.) takes two functions here, f corresponds to $(b \to c)$ in the type signature whereas g refers to $(a \to b)$. g is applied to x, and

then its result is sent to f to reach the final argument.

↓

The composition operator can be seen as a way of pipelining data through multiple functions.

↓

example : negate . sum $ [1, 2, 3, 4, 5]

↓

even parenthesis could have been used here

*) The need for $: Because function application has a higher precedence than the composition operator, that function application would happen before the two functions composed. we would be passing a numeric value where our composition operator needs a function. By using the $ we signal that application to the arguments should happen after the functions are already composed.

We can even parenthesize the expression as :

(negate . sum) [1, 2, 3, 4, 5]

*) We may define it this way as well, which is more similar to how composition is written in source files.

let f x = take 5 . enumFrom $ 3

One reason to do this is that makes it easy to compose more than 2 functions. for example

take 5 . filter odd . enumFrom $ 3

*) As we compose more functions, nesting the functions might get tiresome.

*) It also allows us to write in an even more terse style known as "pointfree".

→ Pointfree style : Pointfree refers to a style of composing functions without specifying their arguments. This style is tidier on the page and easier to read as it helps the reader focus on the functions rather than the data that is being passed around.

(f . g . h) x easier to read than f (g (h (x))),
and it also brings the focus to the functions rather than the arguments.

↓

Pointfree is an extension of that idea but now we drop the argument altogether.

f . g = \x → f (g x)

f . g . h = \x → f (g (h (x)))

↓

```
let f = negate . sum
f [1, 2, 3, 4, 5]
-15
```

↓

When we define our function f, we don't specify that there will be any arguments.

↓

```
>> f :: Int -> [Int] -> Int
>> f z xs = foldr (+) z xs
```

↓ Point free

```
>> f = foldr (+)
>> f 0 [1..5]
   15
```

↓

```
>> let f = length . filter (== 'a')
>> f "abracadabra"
5
```

filter with the bool operator, matches the elements of the list to filter out elements.

↓

```
add :: Int -> Int -> Int
add x y = x + y

addPf :: Int -> Int -> Int
addPf = (+)
```

```
addOne :: Int → Int
addOne = \x+1 = x+1

addOnePF :: Int → Int
addOnePF = (+1)
```

↓

```
main :: IO()
main = do
        print ( 0 :: Int )
        print (add 1 0)          # 1
        print (addOne 0)         # 1
        print  (addOnePF 0)      # 1
        print ( (addOne . addOne ) 0)        # 2
        print ( (addOnePF . addOne ) 0)      # 2
        print ( (addOne . addOnePF )0)       # 2
        print ( (addOnePF . addOnePF)0)      # 2
        print ( negate (addOne 0) )          # -1
        print ( (negate . addOne ) 0 )       # -1
        print ( (addOne . addOne . addOne .  # 2
                                negate . addOne ) 0 )
```

*) Remember that the functions in composition are applied from right to left.

→ Demonstrating composition

print & putStr → earlier said that they seem similar on the surface but behave differently because they have different underlying types.

↓

```
putStr :: String -> IO()
putStrLn :: String -> IO()
print :: Show a => a -> IO()
```

↓

First two take Strings as arguments, which print has a constrained polymorphic parameter, Show a ⇒ a.

The first two work fine to display values that are already in String format, but what about values not in the String format. First we need to convert them to String format and then print them.

↓

Show has a fn. 'show', that got an argument with Show class constraint, and converted to String. This would be sent to putStr to print to screen.

↓

fortunately, it was understood that this would be a common pattern, so the function named print is the composition of putStrLn & show.

↓

when we are working with functions primarily in terms of composition rather than application, the pointfree version

Can sometimes be more elegant.