WEEK 10 PRESENTATION

Agenda:

1. Quick context week 10 - Higher Order Functions
2. Group practice problem
3. Explain Exercise07

Higher Order Functions

4. Functions are DATA in Haskell, and can be treated just like data of any other type.
    - This is known as first class functions
    - Functions can be used as arguments for functions -- essentially what a higher order function is
    - Functions which make use of other functions

What examples can you think of that we have already touched upon, that is a higher order function?

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f xs     []     = []
zipWith f []     ys     = []
zipWith f (x:xs) (y:ys) = (f x y) : zipWith f xs ys
```

Functional Programming - makes extensive use of Higher Order Function

5. Programs are constructed by applying and composing functions

Partial Application

add:: int -> (int -> int)
`add x y = x + y` ====> 2-ary function (meaning it takes two inputs, or two arguments)

We can define a unary function using partial application :

incr:: Int - > Int
incr = add 1

--- the argument that 'incr' takes in is IMPLICIT since Haskell automatically knows add takes two arguments, this function will take as a input an Int and add 1 to it

Currying:

6. Process of transforming a function that takes multiple arguments into a function that takes just a single argument, and returns another function, which accepts further arguments
7. Haskell functions are curried by default.

```
add x y = x + y
```

8. -- int -> (int -> int)
9. -- function takes an int, and returns another function.

10. add5 = add 5
11. returns a function with only ONE argument

i.e. what Haskell does in the background:

12. add x y z = x + y + z   ; int ->(int -> (int -> int))

Takes as input x, and outputs the below function:

add y z = 5 + y + z   ; int -> (int - > int)

Takes as input y and outputs the below function:

add z = 5 + 2 + z   ; int -> int

To specify an uncurried function:
Use tuples: i.e. add: (int->int) -> int ; you cant specify one argument at a time

```
flip :: (a -> b -> c) -> b -> a -> c
flip f = (\x y -> f y x)
```

13. flip f = (\x y -> f y x)   ; (a->b->c) ->(b -> (a -> c))

i.e. if f was add, it will do the same currying as before with the 'add' function.

add y z = y + z   ; int -> (int - > int)

-- take an 'int' and return the add function below with just z as parameter.
i.e. let's give y = 2
Takes as input y and outputs the below function:

add z = 2 + z   ; int -> int

As you can see it does this: ((add y) z)

<u>Function Application ==> LEFT ASSOCIATIVE</u>

F x y == (f x) y =====> brackets implicitly there
-- Haskell applies the first argument to the function

Function Symbol -> ====> RIGHT ASSOCIATIVE

a -> b -> c === a -> (b -> c)

But
a -> b -> c === (a -> b) -> c THIS MEANS YOU ARE TAKING AS INPUT A FUNCTION like shown in flip.

Function Composition

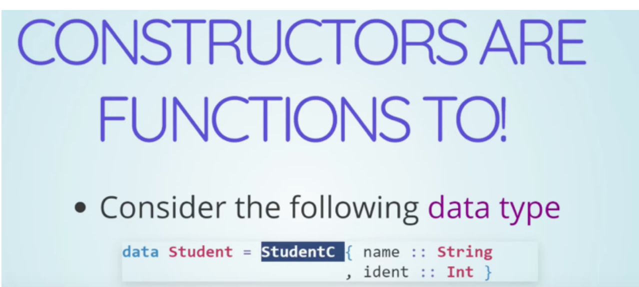
```
-- When input parameter of a function f, is another function (i.e. g). The output type of
                        Z = f . g   == Z = f (g(x) )
-- when using this operator, argument is implicitly there.
-- also known as "point free form"
```

Parenthesis Buildup

c$ -> replaces parenthesis , therefore the parenthesis must run until the end.

```
            fun (x:xs) = foldr (+) 0 (map (+2) (reverse (init(x:xs))))
            fun1 (x:xs) = foldr (+) 0 $ map (+2) $ reverse $ init $ x:xs
```

Can't get the parenthesis in the middle, since it does not run until the end!



This would be like defining:

```
                                    Data Student = StudentC String Int

                                    name :: Student -> String
                                    Name1 = "Lucy"

                                    ident :: Student -> Int

                                    ident1 = 1

                                    Student1 :: Student
                                    Student1 = StudentC Name1 Ident1

 In ghci ==>  name Student1  === Name1 === "Lucy"
```

QUESTIONS:

14.
```haskell
data FamilyTree = Person { name   :: String
, mother :: Maybe FamilyTree
, father :: Maybe FamilyTree }
  deriving (Show,Eq)

person2 :: FamilyTree
person2 = Person "Lucy" (Just mother1) (Just father1)
mother1:: FamilyTree
mother1 = Person "Lina" (Just (Person "Lola" Nothing Nothing)) Nothing
father1:: FamilyTree
father1 = Person "John" Nothing Nothing
```

15. Using the above Family Tree, output the greatGrandmother of any person, from both the father and mothers side, in a list. If the grandmother is not found, or you're given "nothing", replace what would be the name with "nothing"

```haskell
greatGrandmother :: FamilyTree -> [String]
```

```
*Lib> greatGrandmother person2
["Lola","Nothing"]
```

16. Define the function composition operator (.)

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

(.) :: (b->c) -> (a->b) -> a -> c

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
```

17. Implement the function composition operator

(a->b)->a->b

```haskell
($) :: (a -> b) -> a -> b
```

Solutions

Monday, November 22, 2021

12:36 AM

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

```haskell
($) :: (a -> b) -> a -> b
f $ x = f x
```

```haskell
-- FOR GROUP QUESTIONS! ---


data FamilyTree = Person { name   :: String
, mother :: Maybe FamilyTree
, father :: Maybe FamilyTree }
  deriving (Show,Eq)

person2 :: FamilyTree
person2 = Person "Lucy" (Just mother1) (Just father1)
mother1:: FamilyTree
mother1 = Person "Lina" (Just (Person "Lola" Nothing Nothing)) Nothing
father1:: FamilyTree
father1 = Person "John" Nothing Nothing


fromM :: Maybe FamilyTree -> FamilyTree
fromM (Just a) = a
fromM Nothing = Person "Nothing" Nothing Nothing

greatGrandmother :: FamilyTree -> [String]
greatGrandmother (Person name t1 t2) =  let
                                         greatGrandmother' n (Person name t t0)
                                                | n == 0 = name
                                                |otherwise = greatGrandmother' (n-1) (fromM t)
                                         in [greatGrandmother' 2 (Person name t1 t2)]  ++ [greatGrandmother' 2 (Person name t2 t1)]
```