FUNCTIONAL PROGRAMMING

Haskell

LAMBDAS

- ***Lambdas* are anonymous functions** that we use when we need a function only once.
- Normally, we make a lambda with the sole purpose of passing it to a higher-order function.
- To declare a lambda, we write a \ (because it kind of looks like the Greek letter lambda (λ) if you squint hard enough), and then we write the function's parameters, separated by spaces.

LAMBDAS

- zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]

> [153.0,61.5,31.0,15.75,6.6]

- map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]

> [3,8,9,8,7]

- filter (\x -> (mod x 2) == 0) [1, 2, 3, 4, 5, 6]

> [2,4,6]

FOLDS

- Folds can be used to implement any function where you traverse a list once, element by element, and then return something based on that.

- Whenever you want to traverse a list to return something, chances are you want a fold.

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

```
mydouble [] = []
mydouble (x:xs) = [2 * x] ++ mydouble xs
```

FUNCTOR

- A Functor in Category Theory is a mapping.

- Thus, if I have a type **a**, a type **b** and an arrow (i.e. function) **a -> b**, then a **Functor f** does the following mappings:

i) **f** maps type **a** to type **f a**.

ii) **f** maps type **b** to type **f b**.

iii) **f** maps function **a -> b** to function **f a -> f b**.

Please note that the types **a** and **b** can be the same type or different types. Also, please note that here, **f** represents the Functor and not a function. The functions here are denoted by **->**

FUNCTOR (CONTINUED..)

```
module Main where

data MyFunctor a = MySomething a deriving (Show)

instance Functor MyFunctor where
  fmap f (MySomething x) = MySomething (f x)

main :: IO ()
main  =
  do
    putStrLn "Program begins."

    let thing1 = MySomething 45
    print thing1
    print (fmap (*2) thing1)
    print (fmap (+1) thing1)
    print (fmap (:[]) thing1)
    print (fmap (\x -> 2*x+1:[]) thing1)
    print thing1
    putStrLn "Program ends."
```

FUNCTOR (CONTINUED..)

```
module Main where

data MyFunctor a = MySomething a
           | MySomethingElse [a]
           | MySomethingOther (a,a)
           | MyYetAnother a
           deriving (Show)

instance Functor MyFunctor where
  fmap f (MySomething x)      = MySomething (f x)
  fmap f (MySomethingElse xs) = MySomethingElse (map f xs)
  fmap f (MySomethingOther x) = MySomethingOther (f (fst x), f (snd x))
  fmap f (MyYetAnother x)     = MySomething (f x)
```

To be continued…

FUNCTOR (CONTINUED..)

```
main :: IO ()
main  =
  do
    putStrLn "Program begins."

    let thing1 = MySomething 10
    print thing1
    print (fmap (*2) thing1)
    print thing1

    let thing2 = MySomethingElse [100,1000,10000,100000]
    print thing2
    print (fmap (*2) thing2)
    print thing2

    let thing3 = MySomethingOther (400,500)
    print thing3
    print (fmap (*2) thing3)
    print thing3

    let thing4 = MyYetAnother 200
    print thing4
    print (fmap (*2) thing4)
    print thing4

    putStrLn "Program ends."
```

MONOIDS

The formal definition of monoid is that it is a set **with an associative binary operator** and **an identity element**. That's a lot to take in, but breaking the definition down into two parts and demonstrating each with an example or two will make it clear.

```
module BradyBunch where
data Family =
  Mike       { dad :: String
             , kids :: [String]
             } |
  Carol      { mom :: String
             , kids :: [String]
             } |
  BradyBunch { dad :: String
             , mom :: String
             , kids :: [String]
             }
             deriving (Eq, Show)
instance Semigroup Family where
  (Mike dad kids) <> (Carol mom kids') = BradyBunch
dad mom (kids <> kids')
mikesFam :: Family
mikesFam = Mike "Mike" ["Greg", "Peter", "Bobby"]
carolsFam :: Family
carolsFam = Carol "Carol" ["Marcia", "Jan", "Cindy"]
```

Output:

```
1.mikesFam <> carolsFam
2.> BradyBunch {dad = "Mike", mom = "Carol", kids =
["Greg","Peter","Bobby","Marcia","Jan","Cindy"]}
```

MONOIDS

Output:

```
bradyBunch <> alice
> BradyBunch {dad = "Mike", mom = "Carol", kids =
["Greg","Peter","Bobby","Marcia","Jan","Cindy"]}

bradyBunch == (bradyBunch <> alice)
> True
```

```haskell
module BradyBunch where
data Family =
  Mike       { dad :: String
             , kids :: [String]
             } |
  Carol      { mom :: String
             , kids :: [String]
             } |
  BradyBunch { dad :: String
             , mom :: String
             , kids :: [String]
             } |
  Alice
             deriving (Eq, Show)
instance Monoid Family where
  mempty = Alice
  mappend = (<>)
instance Semigroup Family where
  (Mike dad kids) <> (Carol mom kids') = BradyBunch dad mom (kids <>
kids')
  fam     <> Alice     = fam
  Alice   <> fam       = fam
mikesFam :: Family
mikesFam = Mike "Mike" ["Greg","Peter","Bobby"]
carolsFam :: Family
carolsFam = Carol "Carol" ["Marcia","Jan","Cindy"]
bradyBunch :: Family
bradyBunch = mikesFam <> carolsFam
alice :: Family
alice = Alice
```

REDUCE

- const array = [1, 2, 3, 4, 5];

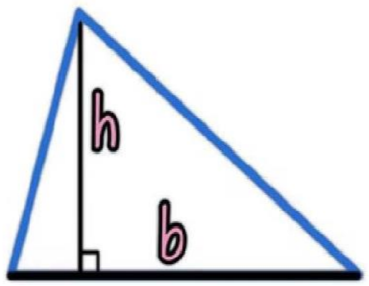
```
array.reduce((accumulative, current) => accumulative + current, 0);
> 15

array.reduce((accumulative, current) => accumulative * current, 1);
> 120
```

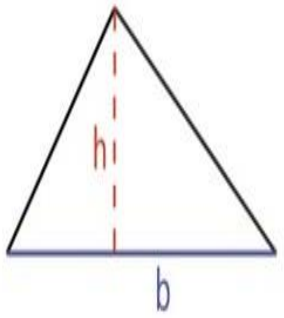
- const array = [[1, 2], [2, 3], [3, 4]];

```
array.reduce((accumulative, current) => accumulative.concat(current), []);
> [1, 2, 2, 3, 3, 4]
```

PRACTICE

- We have two triangles, where the bases of the triangles are 5cm and 3 cm, heights of the triangles are 3 cm and 7 cm respectively. Find the areas of two triangles **using LAMBDAS.**





- **Use FOLDS** to create the list of squares of some numbers.

- Write a program to solve $x^2 + 2x + 5$ or any x **using FUNCTOR.**

- Suppose we have three sets of fruits A = [“mango“, “melon“, “apple“], B= [“berry“, “banana“, “kiwi“, “pine apple“] and C = [“grapes“, “orange“]. Write a program to concat three sets in to one using **MONOIDS.**