Functors and Monads

Abstracting Code Patterns

a.k.a. **Dont Repeat Yourself**

Lists

```
data List a
  = []
  | (:) a (List a)
```

Rendering the Values of a List

```
-- >>> incList [1, 2, 3]
-- ["1", "2", "3"]

showList        :: [Int] -> [String]
showList []     =  []
showList (n:ns) =  show n : showList ns
```

Squaring the values of a list

```
-- >>> sqrList [1, 2, 3]
-- 1, 4, 9

sqrList        :: [Int] -> [Int]
sqrList []     =  []
sqrList (n:ns) =  n^2 : sqrList ns
```

Common Pattern: `map` over a list

Refactor iteration into `mapList`

```
mapList :: (a -> b) -> [a] -> [b]
mapList f []     = []
mapList f (x:xs) = f x : mapList f xs
```

Reuse `map` to implement `inc` and `sqr`

```
showList xs = map (\n -> show n) xs

sqrList  xs = map (\n -> n ^ 2)  xs
```

Trees

Same "pattern" occurs in other structures!

```
data Tree a
  = Leaf
  | Node a (Tree a) (Tree a)
```

Incrementing the values of a Tree

```
-- >>> showTree (Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf))
-- (Node "2" (Node "1" Leaf Leaf) (Node "3" Leaf Leaf))

showTree :: Tree Int -> Tree String
showTree Leaf         = ???
showTree (Node v l r) = ???
```

Squaring the values of a Tree

```
-- >>> sqrTree (Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf))
-- (Node 4 (Node 1 Leaf Leaf) (Node 9 Leaf Leaf))

sqrTree :: Tree Int -> Tree Int
sqrTree Leaf         = ???
sqrTree (Node v l r) = ???
```

QUIZ: `map` over a Tree

Refactor iteration into `mapTree` ! What should the type of `mapTree` be?

```
mapTree :: ???
```

showTree :: Tree Int → Tree String

```
showTree t = mapTree (\n -> show n) t
sqrTree  t = mapTree (\n -> n ^ 2)  t

{- A -} (Int -> Int)    -> Tree Int -> Tree Int
{- B -} (Int -> String) -> Tree Int -> Tree String
{- C -} (Int -> a)      -> Tree Int -> Tree a
{- D -} (a -> a)        -> Tree a   -> Tree a
{- E -} (a -> b)        -> Tree a   -> Tree b
```

Lets write `mapTree`

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Leaf         = ???
mapTree f (Node v l r) = ???
```

QUIZ

Wait ... there is a common pattern across two *datatypes*

```
mapList :: (a -> b) -> List a -> List b    -- List
mapTree :: (a -> b) -> Tree a -> Tree b    -- Tree
```

Lets make a **`class`** for it!

```
class Mappable t where
  gmap :: ???
```

(a → b) → t a → t b

What type should we give to `gmap` ?

```
{- A -} (b -> a) -> t b    -> t a
{- B -} (a -> a) -> t a    -> t a
{- C -} (a -> b) -> [a]    -> [b]
{- D -} (a -> b) -> t a    -> t b
{- E -} (a -> b) -> Tree a -> Tree b
```

Reuse Iteration Across Types

Haskell's libraries use the name `Functor` instead of `Mappable`

```
instance Functor [] where
  fmap = mapList

instance Functor Tree where
  fmap = mapTree
```

And now we can do