Functors and Monads

Abstracting Code Patterns
a.k.a. Don't 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: \texttt{map} over a list

Refactor iteration into \texttt{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?

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

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

```haskell
{- 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
```
Let's write `mapTree`

```haskell
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 :: ???

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

- only list
- only tree
Reuse Iteration Across Types

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

```haskell
instance Functor [] where
  fmap = mapList

instance Functor Tree where
  fmap = mapTree
```

And now we can do

```haskell
-- >>> fmap (\n -> n + 1) (Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf))
-- (Node 4 (Node 1 Leaf Leaf) (Node 9 Leaf Leaf))
```

```haskell
-- >>> fmap show [1,2,3]
-- ["1", "2", "3"]
```

A Type to Represent Expressions
data Expr
    = Number Int        -- ^ 0,1,2,3,4
    | Plus  Expr Expr   -- ^ e1 + e2
    | Minus Expr Expr   -- ^ e1 - e2
    | Mult  Expr Expr   -- ^ e1 * e2
    | Div   Expr Expr   -- ^ e1 / e2

deriving (Show)

(5+6) * (3-1) / (3-3)

Some Example Expressions

e1
e1 = Plus (Number 2) (Number 3)  -- 2 + 3

e2 = Minus (Number 10) (Number 5)  -- 10 - 4

e3 = Mul e1 e2  -- (2+3) * (10-4)

e4 = Div e3 (Number 3)  -- ((2+3) * (10-4)) / 3
**EXERCISE: An Evaluator for Expressions**

Fill in an implementation of `eval`

```haskell
eval :: Expr -> Int
eval e = ???
```

so that when you’re done we get

```plaintext
-- >>> eval e1
-- 5
-- >>> eval e2
-- 6
-- >>> eval e3
-- 30
-- >>> eval e4
-- 10
```
QUIZ

What does the following evaluate to?

\[
\text{quiz} = \text{eval} (\text{Div} (\text{Number 60}) (\text{Minus} (\text{Number 5}) (\text{Number 5})))
\]

A. 0  B. 1  C. Type error  D. Runtime exception  E. NaN

\[
60 \div 0
\]

To avoid crash, return a Result

Lets make a data type that represents Ok or Error