CIS 194

MONADS

"MONADS ARE LIKE BURRITOS"

- By examples in the wild
- As a logical extension of Functor
- Kleisli arrows
- Haskell definition

I'M JUST YOUR FRIENDLY NEIGHBORHOOD MONAD

CODING DEMO

THE FUNCTOR APPROACH

LET'S TALK ABOUT FUNCTORS

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

```
The Functor laws:
  fmap id == id
  fmap (f . g) == fmap f . fmap g
```

LIST IS A FUNCTOR

```
data List a = Nil | Cons a (List a)

instance Functor List where
  fmap :: (a -> b) -> List a -> List b
  fmap f Nil          = Nil
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

- `fmap (\x -> x * x) [7,9,4] == [49,81,16]`
- `fmap even [19,2,5] == [False, True, False]`

MAYBE IS A FUNCTOR

```
instance Functor Maybe where
  fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

- `fmap (\x -> x * x) (Just 6) == Just 36`
- `fmap even Nothing == Nothing`

TREES ARE A FUNCTOR

```
data BinTree a
  = E
  | B a (BinTree a) (BinTree a)
  deriving (Show)

instance Functor BinTree where
  fmap f E          = E
  fmap f (B x l r) = B (f x) (fmap f l) (fmap f r)


t = B "super" (B "wow" E E) (B "rad" E (B "cool" E E))
fmap (\s -> s ++ "!") t
```

WHAT DO THEY HAVE IN COMMON?

```
instance Functor List where
  fmap f Nil         = Nil
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)

instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)

instance Functor BinTree where
  fmap f E         = E
  fmap f (B x l r) = B (f x) (fmap f l) (fmap f r)
```

REMEMBER THE DISTRIBUTIVE PROPERTY?

a * (b + c) == a * b + a * c

```
       (a *) ((+)     b   c)   == (+)     (a * b)  (a * c)
fmap f    (Cons x xs) == Cons (f x) (fmap f xs)
```

- `fmap` is kind of distributing `f` over the data
- Data has the same shape but values are changed

- Not exactly the same but there is something there…

DO YOU EVEN LIFT BRO?

- `fmap :: (a -> b) -> f a -> f b`

  ...is equivalent to...

  `fmap :: (a -> b) -> (f a -> f b)`

- `fmap` takes a function that operates on normal values and lifts it to operate on containers of that value
- "fmap extends a function with super powers"

MONADS IN TERMS OF FUNCTORS

```
class Context m where
  inject :: a -> m a
  lift :: (a -> b) -> m a -> m b
  join :: m (m a) -> m a

andThen m f = join (lift f m)
```

But what is `join`?

FLATTENING CONTEXT

- What does a Maybe of a Maybe mean?
- What does a List of a List mean?
- What does a Logger of a Logger mean?

- How can we combine nested contexts?
- Depends on the details of what we are representing

KLEISLI ARROWS

KLEIS-A-WHO?

- Arrows are things of type `a -> m b`
- Think of Arrows as “actions” or “commands”
- Similar to normal functions of type `a -> b`
- But the output value is wrapped in additional context

MONADS AS ARROW COMPOSITION

How do we compose functions?

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

How can we compose Arrows?

```
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = \x -> g x `andThen` f
```

MONADS IN HASKELL

PRELUDE DEFINITION

```
class Applicative m => Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg
```

(RELEVANT) TYPECLASS LAWS

- `return a >>= k == k a`
- `m >>= return == m`
- `m >>= (\x -> k x >>= h) == (m >>= k) >>= h`

Or equivalently…

- `return >=> g = g`
- `f >=> return == f`
- `(f >=> g) >=> h == f >=> (g >=> h)`