Class 7: Functor, Foldable

February 28

review: type classes

type class declaration

```
class Eq a where
  (==) :: a -> a -> Bool
```

type class instance declaration

```
instance Eq Foo where
  (==) :: Foo -> Foo -> Bool
  (A i1) == (A i2) = i1 == i2
  (B c1) == (B c2) = c1 == c2
  _ == _ = False
```

function with type class constraint

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y : ys) = x == y || elem x ys
```

generalizing map

```
map :: (a -> b) -> [a] -> [b]

treeMap :: (a -> b) -> Tree a -> Tree b

maybeMap :: (a -> b) -> Maybe a -> Maybe b
```

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

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

a digression into kinds

types have types too!
they’re called *kinds*.

```
Prelude> :k Int
Int :: *

Prelude> :k Bool
Bool :: *

Prelude> :k Char
Char :: *
```

(you can follow along in GHCi)

```
data Maybe a
  = Nothing
  | Just a
```

```
Prelude> :k Maybe Int
Maybe Int :: *

Prelude> :k Maybe
Maybe :: * -> *
```

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

```
Prelude> :k List
List :: * -> *

Prelude> :k []
[] :: * -> *
```

normal lists are defined similarly, just with special syntax

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

```
Prelude> :k Tree
Tree :: * -> *
```

generalizing map

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

```
instance Functor Int where
  fmap = …
```

```
error:
Expected kind ‘* -> *’,
but ‘Int’ has kind ‘*’
```

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

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

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

```
instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap _ [] = []
  fmap f (x : xs) = f x : fmap f xs
```

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

```
instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap = map
```

(tree functor exercise)

generalizing fold

```
listFold :: (a -> b -> b) -> b -> [a] -> b

treeFold :: (a -> b -> b) -> b -> Tree a -> b
```

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

```
instance Foldable [] where
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr _ z [] = z
  foldr f z (x : xs) = f x (foldr f z xs)
```

(tree foldable exercise)

```
any :: (a -> Bool) -> [a] -> Bool
any f = foldr ((||) . f) False
any :: (a -> Bool) -> [a] -> Bool
```

generalizes to

```
any :: Foldable t => (a -> Bool) -> t a -> Bool
```

```haskell
elem :: Eq a => a -> [a] -> Bool
elem x = any (x ==)
elem :: Eq a => a -> [a] -> Bool
```

generalizes to

```
elem :: (Foldable t, Eq a) => a -> t a -> Bool
```

```haskell
sum :: [Int] -> Int
sum = foldr (+)
sum :: [Int] -> Int
```

generalizes to

```haskell
sum :: (Foldable t, Num a) => t a -> a
```

(toList example)

today’s type classes

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

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```