Functors and Friends

Stephen A. Edwards

Columbia University

Fall 2023
Functors

Functor Laws

Applicative Functors

Pure and the <$> Operator

ZipList Applicative Functors

liftA2

sequenceA

Applicative Functor Laws

newtype

Monoids

Foldable
Functors: Types That Hold a Type in a Box

\[ \text{class } \text{Functor } f \text{ where} \]
\[ \quad \text{fmap} :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b \]

\( f \) is a type constructor of kind \( \star \rightarrow \star \). “A box of”

\( \text{fmap}\ g\ x \) means “apply \( g \) to every \( a \) in the box \( x \) to produce a box of \( b \)'s”

\text{data} \ \text{Maybe}\ a = \text{Just}\ a \mid \text{Nothing} \\
\text{instance} \ \text{Functor}\ \text{Maybe} \text{ where} \\
\quad \text{fmap} \ _\  \text{Nothing} = \text{Nothing} \\
\quad \text{fmap}\ g\ (\text{Just}\ x) = \text{Just}\ (g\ x) \\

\text{data} \ \text{Either}\ a\ b = \text{Left}\ a \mid \text{Right}\ b \\
\text{instance} \ \text{Functor}\ (\text{Either}\ a) \text{ where} \\
\quad \text{fmap} \ _\ (\text{Left}\ x) = \text{Left}\ x \\
\quad \text{fmap}\ g\ (\text{Right}\ y) = \text{Right}\ (g\ y) \\

\text{data} \ \text{List}\ a = \text{Cons}\ a\ (\text{List}\ a) \mid \text{Nil} \\
\text{instance} \ \text{Functor}\ \text{List} \text{ where} \\
\quad \text{fmap}\ g\ (\text{Cons}\ x\ xs) = \text{Cons}\ (g\ x)\ (\text{fmap}\ g\ xs) \\
\quad \text{fmap} \ _\ \text{Nil} = \text{Nil} \]
**IO as a Functor**

*Functor* takes a type constructor of kind \( \star \rightarrow \star \), which is the kind of \( \text{IO} \)

```
Prelude> :k IO
IO :: \( \star \rightarrow \star \)
```

IO does behave like a kind of box:

```haskell
query :: IO String
query = do line <- getLine -- getLine returns a box :: IO String
        let res = line ++ "!" -- take line out of box from getLine
        return res -- put res in an IO box
```

The definition of *Functor* \( \text{IO} \) in the Prelude: (alternative syntax)

```haskell
instance Functor IO where
    fmap f action = do result <- action -- take result from the box
                      return (f result) -- apply f; put it a box
```
Using fmap with I/O Actions

```haskell
main = do line <- getLine
         let revLine = reverse line  -- Tedious but correct
             putStrLn revLine

main = do revLine <- fmap reverse getLine  -- More direct
         putStrLn revLine

Prelude> fmap (++"!") getLine
foo
"foo!"
```
Functions are Functors

Prelude> :k (->)
(->) :: * -> * -> *  -- Like ``(+)`` (->) is a function on types

That is, the function type constructor `->` takes two concrete types and produces a third (a function). This is the same kind as `Either`

Prelude> :k ((->) Int)
((->) Int) :: * -> *

The `((->) Int)` type constructor takes type `a` and produces functions that transform Ints to `a`'s. `fmap` will apply a function that transforms the `a`'s to `b`'s.

```
instance Functor ((->) a) where
  fmap f g = \x -> f (g x)  -- Wait, this is just function composition!
```

```
instance Functor ((->) a) where
  fmap = (.)  -- Much more succinct (Prelude definition)
```
Fmapping Functions: \( \text{fmap \, f \, g = f \circ g} \)

Prelude> :t fmap (*3) (+100)
fmap (*3) (+100) :: \text{Num \, b \Rightarrow b \rightarrow b}

Prelude> fmap (*3) (+100) 1
303

Prelude> (*3) `fmap` (+100) $ 1
303

Prelude> (*3) . (+100) $ 1
303

Prelude> fmap (show . (*3)) (+100) 1
"303"
Partially Applying $fmap$

```
Prelude> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b

Prelude> :t fmap (*3)
fmap (*3) :: (Functor f, Num b) => f b -> f b
```

"$fmap (*3)$" is a function that operates on functors of the Num type class ("functors over numbers"). The function (*3) has been *lifted* to functors

```
Prelude> :t fmap (replicate 3)
fmap (replicate 3) :: Functor f => f a -> f [a]
```

"$fmap (replicate 3)$" is a function over functors that generates "boxed lists"
Functor Laws

Applying the identity function does not change the functor ("fmap does not change the box"): 

\[ \text{fmap } \text{id} = \text{id} \]

Applying \textit{fmap} with two functions is like applying their composition ("applying functions to the box is like applying them in the box"): 

\[ \text{fmap } (f \circ g) = \text{fmap } f \circ \text{fmap } g \]

\[ \text{fmap } (\lambda y \rightarrow f(g\ y)) \ x = \text{fmap } f \ (\text{fmap } g \ x) \quad \text{-- Equivalent} \]
data Maybe a = Just a | Nothing

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

{- Does Maybe follow the laws? -}

fmap id Nothing = Nothing  -- from the definition of fmap
fmap id (Just x) = Just (id x)  -- from the definition of fmap
  = Just x  -- from the definition of id

(fmap f . fmap g) Nothing = fmap f (fmap g Nothing)  -- def of .
  = fmap f Nothing  -- def of fmap
  = Nothing  -- def of fmap
  = fmap (f . g) Nothing  -- def of fmap

(fmap f . fmap g) (Just x) = fmap f (fmap g (Just x))  -- def of .
  = fmap f (Just (g x))  -- def of fmap
  = Just (f (g x))  -- def of fmap
  = Just ((f . g) x)  -- def of .
  = fmap (f . g) (Just x)  -- def of fmap
data CMaybe a = CNothing | CJust Int a

 deriving Show

instance Functor CMaybe where -- Purported
    fmap _ CNothing = CNothing
    fmap f (CJust c x) = CJust (c+1) (f x)

*Main> fmap id CNothing
CNothing -- OK: fmap id Nothing = id Nothing

*Main> fmap id (CJust 42 "Hello")
CJust 43 "Hello" -- FAIL: fmap id /= id because 43 /= 42

*Main> fmap (+1) . (+1) (CJust 42 100)
CJust 43 102

*Main> (fmap (+1) . fmap (+1)) (CJust 42 100)
CJust 44 102 -- FAIL: fmap (f . g) /= fmap f . fmap g because 43 /= 44
Multi-Argument Functions on Functors: Applicative Functors

Functions in Haskell are Curried:

\[
1 + 2 = (+) \ 1 \ 2 = ((+) \ 1) \ 2 = (1+) \ 2 = 3
\]

What if we wanted to perform 1+2 in a Functor?

\[
\text{class Functor } f \ \text{where}
\]

\[
fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b
\]

\[\text{fmap is “apply a normal function to a functor, producing a functor”}\]

Say we want to add 1 to 2 in the \([\ ]\) Functor (lists):

\[
[1] + [2] = (+) \ [1] \ [2] \quad -- \text{Infix to prefix}
\]

\[= (fmap \ (+) \ [1]) \ [2] \quad -- \text{fmap: apply function to functor}\]

\[= [(1+)] \ [2] \quad -- \text{Now what?}\]

We want to apply a Functor containing functions to another functor, e.g., something with the signature \([a \rightarrow b] \rightarrow [a] \rightarrow [b]\)
Applicative Functors: Applying Functions in a Functor

```haskell
infixl 4 <*>
class Functor f => Applicative f where
    pure :: a -> f a  -- Box something, e.g., a function
    (<<*>>) :: f (a -> b) -> f a -> f b  -- Apply boxed function to a box

instance Applicative Maybe where
    pure = Just  -- Put it in a “Just” box
    Nothing <*> _ = Nothing  -- No function to apply
    Just f <*> m = fmap f m  -- Apply function-in-a-box f

Prelude> :t fmap (+) (Just 1)
fmap (+) (Just 1) :: Num a => Maybe (a -> a)  -- Function-in-a-box

Prelude> fmap (+) (Just 1) <*> (Just 2)
Just 3
Prelude> fmap (+) Nothing <*> (Just 2)
Nothing  -- Nothing is a buzzkiller
```
Pure and the `<$>` Operator

Prelude> `pure` (10-) <$> Just 4  
Just 6

Prelude> (10-) `fmap` (Just 10) <$> Just 4  
Just 6

Prelude> (\_\_)`fmap` (Just 10) <$> Just 4  
Just 6

<$> is simply an infix `fmap` meant to remind you of the $ operator

```
infixl 4 <$>  
(<$>) :: Functor f => (a -> b) -> f a -> f b  
f <$> x = `fmap` f x  
```

--- Or equivalently, `f `fmap` x`

So  
`f <$> x <*> y <*> z` is like  
`f x y z` but on applicative functors `x, y, z`

Prelude> (+) <$> [1] <*> [2]  
[3]

Prelude> (,,) <$> Just "PFP" <*> Just "Rocks" <*> Just "Out"  
Just ("PFP","Rocks","Out")
Maybe as an Applicative Functor

\[
\text{instance Functor Maybe where}
\]
\[
\text{fmap} \; \_ \; \text{Nothing} \; = \; \text{Nothing}
\]
\[
\text{fmap} \; g \; \text{(Just} \; x) \; = \; \text{Just} \; (g \; x)
\]

\[
\text{infixl 4} \; <\$> \;
\]
\[
f \; <\$> \; x \; = \; \text{fmap} \; f \; x
\]

\[
\text{infixl 4} \; <*> \;
\]
\[
\text{instance Applicative Maybe where}
\]
\[
\text{pure} \; = \; \text{Just}
\]
\[
\text{Nothing} \; <*> \; \_ \; = \; \text{Nothing}
\]
\[
\text{Just} \; f \; <*> \; m \; = \; \text{fmap} \; f \; m
\]

\[
f \; <\$> \; \text{Just} \; x \; <*> \; \text{Just} \; y
\]
\[
= \; (f \; <\$> \; \text{Just} \; x) \; <*> \; \text{Just} \; y \quad -- \; a \; <\$> \; b \; <*> \; c \; = \; (a \; <\$> \; b) \; <*> \; c
\]
\[
= \; (\text{fmap} \; f \; \text{(Just} \; x)) \; <*> \; \text{Just} \; y \quad -- \; \text{Definition of} \; <\$>
\]
\[
= \; (\text{Just} \; (f \; x)) \; <*> \; \text{Just} \; y \quad -- \; \text{Definition of fmap Maybe}
\]
\[
= \; \text{fmap} \; (f \; x) \; \text{(Just} \; y) \quad -- \; \text{Definition of} \; <*>$
\]
\[
= \; \text{Just} \; (f \; x \; y) \quad -- \; \text{Definition of fmap Maybe}$
Lists are Applicative Functors

```
instance Applicative [] where
  pure x = [x]               -- Pure makes singleton list
  fs <*> xs = [ f x | f <- fs, x <- xs ] -- All combinations
```

<*> associates (evaluates) left-to-right, so the last list is iterated over first:

```
Prelude> [ (++"!"), (++"?"), (++".") ] <*> [ "Run", "GHC" ]
["Run!","GHC!","Run?","GHC?","Run.","GHC."]

Prelude> [ x+y | x <- [100,200,300], y <- [1..3] ]
[101,102,103,201,202,203,301,302,303]

Prelude> (+) <$> [100,200,300] <*> [1..3]
[101,102,103,201,202,203,301,302,303]

Prelude> pure (+) <*> [100,200,300] <*> [1..3]
[101,102,103,201,202,203,301,302,303]
```
IO is an Applicative Functor

<*> enables I/O actions to be used more like functions

\[
\text{instance Applicative IO where}
\]
\[
\text{pure = return}
\]
\[
a \triangleleft b = \text{do } f \leftarrow a
\]
\[
x \leftarrow b
\]
\[
\text{return } (f \; x)
\]

Specialized to IO actions,
\[
(\triangleleft \cdot) :: \text{IO } (\text{a } \rightarrow \text{ b})
\]
\[
\rightarrow \text{IO a}
\]
\[
\rightarrow \text{IO b}
\]

main = do
a <- getline
b <- getline
putStrLn $ a ++ b

main :: IO ()

main = do
a <- (++ <$> getline <*> getline)
putStrLn a

$ stack runhaskell af2.hs
One
Two
OneTwo
Function Application ((->) a) as an Applicative Functor

pure :: b -> ((->) a) b
    :: b -> a -> b

(<*>): ((->) a) (b -> c) -> ((->) a) b -> ((->) a) c
    :: (a -> b -> c) -> (a -> b) -> (a -> c)

The “box” is “a function that takes an a and returns the type in the box”

<*> takes f :: a -> b -> c and g :: a -> b and should produce a -> c.

Applying an argument x :: a to f and g gives g x :: b and f x :: b -> c.
This means applying g x to f x gives c, i.e., f x (g x) :: c.

instance Applicative ((->) a) where
    pure x = \_ -> x -- a.k.a., const
    f <$> g = \x -> f x (g x) -- Takes an a and uses f & g to produce a c

Prelude> :t \f g x -> f x (g x)
\f g x -> f x (g x) :: (a -> b -> c) -> (a -> b) -> a -> c
Functions as Applicative Functors

```
instance Applicative ((->) a) where f <*> g = \x -> f x (g x)
instance Functor ((->) a) where fmap = (.)
```

```
Prelude> :t (+) <$> (+3) <*> (*100)
(+) <$> (+3) <*> (*100) :: Num b => b -> b -- A function on numbers
Prelude> ( (+) <$> (+3) <*> (*100) ) 5
508 -- Apply 5 to +3, apply 5 to *100, and add the results
```

Single-argument functions (+3), (*100) are the boxes (arguments are “put inside”), which are assembled with (+) into a single-argument function.

```
( (+) <$> (+3) <*> (*100) ) 5
= ( (((+) . (+3)) <*> (*100)) ) 5 -- Definition of <$> 
= ( \x -> (((+) . (+3)) x ((*100) x)) ) 5 -- Definition of <*> 
= (((+) . (+3)) 5 ((*100) 5)) -- Apply 5 to lambda expr.
= (((+) ((+3) 5)) ((*100) 5)) -- Definition of .
= (+) 8 500 -- Evaluate (+3) 5, (*100) 5
= 508 -- Evaluate (+) 8 500
```
Another example: (,,) is the “build a 3-tuple operator”

Prelude> :t (,,) <$> (+3) <*> (*3) <*> (*100)
(,,) <$> (+3) <*> (*3) <*> (*100) :: Num a => a -> (a, a, a)

Prelude> ((,,) <$> (+3) <*> (*3) <*> (*100)) 2
(5,6,200)

The elements of the 3-tuple:

2 + 3 = 5
2 * 3 = 6
2 * 100 = 200

Each comes from applying 2 to the three functions.

“Generate a 3-tuple by applying the argument to (+3), (*3), and (*100)”
ZipList Applicative Functors

The usual implementation of Applicative Functors on lists generates all possible combinations:

```haskell
Prelude> [(+),(*)] <*> [1,2] <*> [10,100] [11,101,12,102,10,100,20,200]
```

Control.Applicative provides an alternative approach with zip-like behavior:

```haskell
newtype ZipList a = ZipList { getZipList :: [a] }
instance Applicative ZipList where
  pure x = ZipList (repeat x) -- Infinite list of x's
  ZipList fs <*> ZipList xs = ZipList (zipWith ( x -> f x) fs xs)
```

```haskell
> ZipList [(+),(*)] <*> ZipList [1,2] <*> ZipList [10,100]
ZipList {getZipList = [11,200]} -- [1 + 10, 2 * 100]
ZipList {getZipList = [(1,3,5),(2,4,6)]}
```
**liftA2: Lift a Two-Argument Function to an Applicative Functor**

```haskell
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (<*>) = liftA2 id  -- Default: get function from 1st arg’s box

liftA2 :: (a -> b -> c) -> f a -> f b -> f c
liftA2 f x = (<*>) (fmap f x)  -- Default implementation
```

*liftA2* takes a binary function and “lifts” it to work on boxed values, e.g.,

```haskell
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

Prelude Control.Applicative> liftA2 (:) (Just 3) (Just [4])
Just [3,4]  -- Apply (:)) inside the boxes, i.e., Just ((:) 3 [4])

```

```haskell
instance Applicative ZipList where
  pure x = ZipList (repeat x)
  liftA2 f (ZipList xs) (ZipList ys) = ZipList (zipWith f xs ys)
```
```
Turning a list of boxes into a box containing a list

sequenceA1 :: Applicative f => [f a] -> f [a] -- Prelude sequenceA
sequenceA1 [] = pure []
sequenceA1 (x:xs) = (:) <$> x <*> sequenceA1 xs

*Main> sequenceA1 [Just 3, Just 2, Just 1]
Just [3,2,1]

Recall that \( f <$> \text{Just } x <*> \text{Just } y = \text{Just } (f \, x \, y) \)

sequenceA1 [\text{Just } 3, \text{Just } 1]
= (:) <$> \text{Just } 3 <*> sequenceA1 [\text{Just } 1]
= (:) <$> \text{Just } 3 <*> (((:) <$> \text{Just } 1 <*> sequenceA1 [])
= (:) <$> \text{Just } 3 <*> (((:) <$> \text{Just } 1 <*> \text{pure } [])
= (:) <$> \text{Just } 3 <*> (((:) <$> \text{Just } 1 <*> \text{Just } [])
= (:) <$> \text{Just } 3 <*> \text{Just } [1]
= \text{Just } [3,1]
SequenceA Can Also Be Implemented With a Fold

```haskell
import Control.Applicative (liftA2)

sequenceA2 :: Applicative f => [f a] -> f [a]  -- Prelude sequenceA
sequenceA2 = foldr (liftA2 (:)) (pure [])
```

How do the types work out?

- `liftA2 :: App. f ⇒ (a → b → c) → f a → f b → f c`
- `(:) :: a → [a] → [a]`

Passing `(:)` to `liftA2` makes `b = [a]` and `c = [a]`, so

- `liftA2 (:) :: App. f ⇒ f a → f [a] → f [a]`

- `foldr :: (d → e → e) → e → [d] → e`

Passing `liftA2 (:)` to `foldr` makes `d = f a` and `e = f [a]`, so

- `foldr (liftA2 (:)) :: App. f ⇒ f [a] → [f a] → f [a]`
- `pure [] :: App. f ⇒ f [a]`
- `foldr (liftA2 (:)) (pure []) :: App. f ⇒ [f a] → f [a]`
SequenceA in Action

sequenceA :: Applicative f => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])

"Take the items from a list of boxes to make a box with a list of items"

Prelude> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
Prelude> sequenceA [Just 3, Nothing, Just 1]
Nothing  -- "Nothing" nullifies the result

Prelude> :t sequenceA [(+3), (+2), (+1)]
sequenceA [(+3), (+2), (+1)] :: Num a => a -> [a]  -- Produces a list
Prelude> sequenceA [(+3), (+2), (+1)] 10
[13,12,11]  -- Apply the argument to each function

Prelude> sequenceA [[1,2,3],[10,20]]
[[1,10],[1,20],[2,10],[2,20],[3,10],[3,20]]  -- fmap on lists
Applicative Functor Laws

pure \( f \star{} x = \text{fmap} f x \quad -- \star{}: \text{apply a boxed function} \)

pure \( \text{id} \star{} x = x \quad -- \text{Because fmap \( \text{id} = \text{id} \)} \)

pure \( (.) \star{} x \star{} y \star{} z = x \star{} (y \star{} z) \quad -- \star{} \text{is left-to-right} \)

pure \( f \star{} \text{pure} x = \text{pure} (f x) \quad -- \text{Apply a boxed function} \)

\( x \star{} \text{pure} y = \text{pure} (\$ y) \star{} x \quad -- (\$ y): \text{“apply arg. } y\text{”} \)
The **newtype** keyword: Build a New Type From an Existing Type

Say you want a version of an existing type only usable in certain contexts. *type* makes an alias with no restrictions. *newtype* is a more efficient version of *data* that only allows a single data constructor

```haskell
newtype DegF = DegF { getDegF :: Double }
newtype DegC = DegC { getDegC :: Double }

fToC :: DegF -> DegC
fToC (DegF f) = DegC $(f - 32) * 5 / 9

cToF :: DegC -> DegF
cToF (DegC c) = DegF $(c * 9 / 5) + 32

instance Show DegF where show (DegF f) = show f ++ "F"

instance Show DegC where show (DegC c) = show c ++ "C"
```
DegF and DegC In Action

*Main> fToC (DegF 32)
0.0C
*Main> fToC (DegF 98.6)
37.0C
*Main> cToF (DegC 37)
98.6F
*Main> cToF 33
   * No instance for (Num DegC) arising from the literal '33'
*Main> DegC 33 + DegC 32
   * No instance for (Num DegC) arising from a use of '+'
*Main> let t1 = DegC 33
*Main|       t2 = DegC 10 in
*Main|   getDegC t1 + getDegC t2
43.0
Newtype vs. Data: Slightly Faster and Lazier

```haskell
newtype DegF = DegF { getDegF :: Double }
data DegF = DegF { getDegF :: Double } -- Same syntax
```

A `newtype` may only have a single data constructor with a single field.

Compiler treats a `newtype` as the encapsulated type, so it’s slightly faster.

Pattern matching always succeeds for a `newtype`:

```haskell
Prelude> data DT = DT Bool
Prelude> newtype NT = NT Bool

Prelude> helloDT (DT _) = "hello"
Prelude> helloNT (NT _) = "hello"

Prelude> helloDT undefined
"*** Exception: Prelude.undefined"
Prelude> helloNT undefined
"hello" -- Just a Bool in NT's clothing
```
## Data vs. Type vs. NewType

<table>
<thead>
<tr>
<th>Keyword</th>
<th>When to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>When you need a completely new algebraic type or record, e.g., data MyTree a = Node a (MyTree a) (MyTree a)</td>
</tr>
<tr>
<td>type</td>
<td>When you want a concise name for an existing type and aren’t trying to restrict its use, e.g., type String = [Char]</td>
</tr>
<tr>
<td>newtype</td>
<td>When you’re trying to restrict the use of an existing type and were otherwise going to write data MyType = MyType t</td>
</tr>
</tbody>
</table>
Monoids

Type classes present a common interface to types that behave similarly.

A Monoid is a type with an associative binary operator and an identity value.

E.g., \(*\) and 1 on numbers, ++ and [] on lists:

```haskell
Prelude> 4 * 1
4  -- 1 is the identity on the right
Prelude> 1 * 4
4  -- 1 is the identity on the left
Prelude> 2 * (3 * 4)
24
Prelude> (2 * 3) * 4
24  -- * is associative
Prelude> 2 * 3
6
Prelude> 3 * 2
6  -- * happens to be commutative
```

```haskell
Prelude> "hello" ++ []
"hello"  -- [] is the right identity
Prelude> [] ++ "hello"
"hello"  -- [] is the left identity
Prelude> "a" ++ ("bc" ++ "de")
"abcde"
Prelude> ("a" ++ "bc") ++ "de"
"abcde"  -- ++ is associative
Prelude> "a" ++ "b"
"ab"
Prelude> "b" ++ "a"
"ba"  -- ++ is not commutative
```
The Monoid Type Class

class Monoid m where
  mempty :: a          -- The identity value
  mappend :: m -> m -> m -- The associative binary operator
  mconcat :: [m] -> m  -- Apply the binary operator to a list
  mconcat = foldr mappend mempty -- Default implementation

Lists are Monoids:

instance Monoid [a] where
  mempty = []
  mappend = (++)

Prelude> mempty :: [a]
[]
Prelude> "hello " `mappend` "world!"
"hello world!"
Prelude> mconcat ["hello ","pfp ","world!"]
"hello pfp world!"
In Data.Monoid, newtype lets us build distinct Monoids for each

```
newtype Product a = Product { getProduct :: a }
deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Product a) where
  mempty = Product 1
  Product x `mappend` Product y = Product (x * y)
```

```
newtype Sum a = Sum { getSum :: a }
deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Sum a) where
  mempty = Sum 0
  Sum x `mappend` Sum y = Sum (x + y)
```
Product and Sum In Action

Prelude Data.Monoid> `mempty` :: Sum Int
Sum {getSum = 0}
Prelude Data.Monoid> `mempty` :: Product Int
Product {getProduct = 1}

Prelude Data.Monoid> Sum 3 `mappend` Sum 4
Sum {getSum = 7}
Prelude Data.Monoid> Product 3 `mappend` Product 4
Product {getProduct = 12}

Prelude Data.Monoid> `mconcat` [Sum 1, Sum 10, Sum 100]
Sum {getSum = 111}
Prelude Data.Monoid> `mconcat` [Product 10, Product 3, Product 5]
Product {getProduct = 150}
The Any (||, False) and All (&&, True) Monoids

In Data.Monoid,

```haskell
newtype Any = Any { getAny :: Bool }
  deriving (Eq, Ord, Read, Show, Bounded)

instance Monoid Any where
  mempty = Any False
  Any x `mappend` Any y = Any (x || y)
```

```haskell
newtype All = All { getAll :: Bool }
  deriving (Eq, Ord, Read, Show, Bounded)

instance Monoid All where
  mempty = All True
  All x `mappend` All y = All (x && y)
```
Any and All

Prelude Data.Monoid> mempty :: Any
Any {getAny = False}
Prelude Data.Monoid> mempty :: All
All {getAll = True}

Prelude Data.Monoid> getAny $ Any True `mappend` Any False
True
Prelude Data.Monoid> getAll $ All True `mappend` All False
False

Prelude Data.Monoid> mconcat [Any True, Any False, Any True]
Any {getAny = True}
Prelude Data.Monoid> mconcat [All True, All True, All False]
All {getAll = False}

Yes, *any* and *all* are easier to use
data Ordering = LT | EQ | GT

In Data.Monoid,

instance Monoid Ordering where
  mempty = EQ
  LT `mappend` _ = LT
  EQ `mappend` y = y
  GT `mappend` _ = GT

Application: an \textit{lcomp} for strings ordered by length then alphabetically, e.g.,

\texttt{lcomp :: String -> String -> Ordering}

"b" \texttt{lcomp} "aaaa" = LT -- b is shorter
"bbbbbb" \texttt{lcomp} "a" = GT -- bbbbb is longer
"avenger" \texttt{lcomp} "avenged" = LT -- Same length: r is after d
lcomp :: String -> String -> Ordering
lcomp x y = case length x `compare` length y of
  LT -> LT
  GT -> GT
  EQ -> x `compare` y

A little too operational; mappend is exactly what we want

lcomp :: String -> String -> Ordering
lcomp x y = (length x `compare` length y) `mappend`
            (x `compare` y)
Maybe the Monoid

```haskell
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

```
Prelude> Nothing `mappend` Just "pfp"
Just "pfp"
Prelude> Just "fun" `mappend` Nothing
Just "fun"

Prelude> :m +Data.Monoid
Prelude Data.Monoid> Just (Sum 3) `mappend` Just (Sum 4)
Just (Sum {getSum = 7})
```
The Foldable Type Class

What I taught you:

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ _ [] = _
foldr f a (x:xs) = f x (foldr f a xs)
```

How it’s actually defined (Data.Foldable):

```haskell
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```
class Foldable t where
{-# MINIMAL foldMap | foldr #-}
foldr, foldr' :: (a -> b -> b) -> b -> t a -> b
foldr1 :: (a -> a -> a) -> t a -> a
foldl, foldl' :: (b -> a -> b) -> b -> t a -> b
foldl1 :: (a -> a -> a) -> t a -> a
fold :: Monoid m => t m -> m
      -- with mappend
foldMap :: Monoid m => (a -> m) -> t a -> m
toList :: t a -> [a]
null :: t a -> Bool
length :: t a -> Int
elem :: Eq a => a -> t a -> Bool
maximum :: Ord a => t a -> a
minimum :: Ord a => t a -> a
sum :: Num a => t a -> a
product :: Num a => t a -> a

Instance of Foldable for [] is just the usual list functions
data Tree a = Node a (Tree a) (Tree a) | Nil deriving (Eq, Read)

instance Foldable Tree where
  foldMap _ Nil = mempty
  foldMap f (Node x l r) = foldMap f l `mappend`
    f x `mappend`
    foldMap f r

> foldl (+) 0 (fromList [5,3,1,2,4,6,7] :: Tree Int)
28    -- folding the tree
> getSum $ foldMap Sum $ fromList [5,3,1,2,4,6,7]
28    -- The Sum Monoid's mappend is +
> getAny $ foldMap (\x -> Any $ x == 'w') $ fromList "brown"
True   -- Any's mappend is ||
> getAny $ foldMap (Any . (=='w')) $ fromList "brown"
True   -- More concise
> foldMap (\x -> [x]) $ fromList [5,3,1,2,4,6,7]
[1,2,3,4,5,6,7]   -- List's mappend is ++