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

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

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

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

```haskell
data Maybe a = Just a | Nothing
instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap g (Just x) = Just (g x)
```

```haskell
data Either a b = Left a | Right b
instance Functor (Either a) where
    fmap _ (Left x) = Left x
    fmap g (Right y) = Right (g y)
```

```haskell
data List a = Cons a (List a) | Nil
instance Functor List where
    fmap g (Cons x xs) = Cons (g x) (fmap g xs)
    fmap _ Nil = Nil
```
IO as a Functor

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

```
Prelude> :k IO
IO :: * -> *
```

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 IO in the Prelude: (alternative syntax)

```
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 \( \rightarrow \) takes two concrete types and produces a third (a function). This is the same kind as \texttt{Either}

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

The \((\rightarrow) \text{Int}\) type constructor takes type \(a\) and produces functions that transform Ints to \(a\)'s. \texttt{fmap} will apply a function that transforms the \(a\)'s to \(b\)'s.

\[
\text{instance } \texttt{Functor} ((\rightarrow) \text{a}) \text{ where}
\]
\[
\text{fmap } f \ g = \backslash x \rightarrow f \ (g \ x) \quad \text{-- Wait, this is just function composition!}
\]

\[
\text{instance } \texttt{Functor} ((\rightarrow) \text{a}) \text{ where}
\]
\[
\text{fmap} = (.) \quad \text{-- Much more succinct (Prelude definition)}
\]
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 \( \text{fmap} \) with two functions is like applying their composition (“applying functions to the box is like applying them in the box”):

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

\[
\text{fmap } (\lambda y \rightarrow f \ (g \ y)) \ x = \text{fmap } f \ (\text{fmap } g \ x) \quad -- \text{Equivalent}
\]
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?

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

`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 \([a -> b] -> [a] -> [b]\)
Applicative Functors: Applying Functions in a Functor

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

\textbf{infixl} 4 <$>  
( <$> ) :: Functor f => (a -> b) -> f a -> f b  
f <$> x = fmap f x  
\text{-- 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**

```haskell
instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap g (Just x) = Just (g x)

infixl 4 <$>!
f <$> x = fmap f x
```

```haskell
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    Just f <*> m = fmap f m
```

```haskell
f <$> Just x <*> Just y
= (f <$> Just x) <*> Just y  -- a <$> b <*> c = (a <$> b) <*> c
= (fmap f (Just x)) <*> Just y  -- Definition of <$>!
= (Just (f x)) <*> Just y  -- Definition of fmap Maybe
= fmap (f x) (Just y)  -- Definition of <*>!
= Just (f x y)  -- Definition of fmap Maybe
```
Lists are Applicative Functors

```haskell
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:

```haskell
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

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

Specialized to IO actions,

\[
(\ltimes) :: \text{IO } (a \rightarrow b) \\
\rightarrow \text{IO } a \\
\rightarrow \text{IO } b
\]

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 \((\rightarrow) a\) as an Applicative Functor

\[
\text{pure} :: b \to ((\rightarrow) a) b \\
:: b \to a \to b \\
\langle\ast\rangle :: ((\rightarrow) a) (b \to c) \to ((\rightarrow) a) b \to ((\rightarrow) a) c \\
:: (a \to b \to c) \to (a \to b) \to (a \to c)
\]

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

\(\langle\ast\rangle\) takes \(f :: a \to b \to c\) and \(g :: a \to b\) and should produce \(a \to c\).

Applying an argument \(x :: a\) to \(f\) and \(g\) gives \(g x :: b\) and \(f x :: b \to c\).

This means applying \(g x\) to \(f x\) gives \(c\), i.e., \(f x (g x) :: c\).

\[
\text{instance Applicative } ((\rightarrow) a) \text{ where} \\
\text{pure } x = \_ \to x \quad -- \text{a.k.a., const} \\
\text{f } \langle\ast\rangle \text{ g } = \_ \to f x (g x) \quad -- \text{Takes an } a \text{ and uses } f \text{ & } g \text{ to produce a } c
\]

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

\[
\text{instance Applicative } ((\rightarrow)) \text{ a) where } f \ast g = \lambda x \rightarrow f x (g x)
\]

\[
\text{instance Functor } ((\rightarrow)) \text{ a) where fmap = (.)}
\]

\[
f \langle$\rangle x = \text{fmap} f x
\]

Prelude> :t (+) <$> (+3) <*> (*100)
(+) <$> (+3) <*> (*100) :: \text{Num b} \Rightarrow b \rightarrow b \quad \text{-- A function on numbers}

Prelude> ( (+) <$> (+3) <*> (*100) ) 5
508 \quad \text{-- 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.

\[
\begin{align*}
( \quad (+) \ast \langle$\rangle \quad (+3) \ast \langle$\rangle \quad (*100) \quad ) \quad 5
= ( \quad ((+) \cdot (+3)) \ast \langle$\rangle \quad (*100) \quad ) \quad 5 \quad \text{-- Definition of <$\rangle}
= (\lambda x \rightarrow ((+) \cdot (+3)) x \quad ((*100) x)) \quad 5 \quad \text{-- Definition of <*\rangle}
= \quad ((+) \cdot (+3)) \quad 5 \quad ((*100) \quad 5)) \quad \text{-- Apply 5 to lambda expr.}
= \quad ((+) \quad ((+3) \quad 5)) \quad ((*100) \quad 5)) \quad \text{-- Definition of .}
= \quad (+) \quad 8 \quad 500 \quad \text{-- Evaluate (+3) 5, (*100) 5}
= \quad 508 \quad \text{-- Evaluate (+) 8 500}
\end{align*}
\]
Functions as Applicative Functors

Another example: (,,) is the “build a 3-tuple operator”

```haskell
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:

\[2 + 3 = 5\]
\[2 \times 3 = 6\]
\[2 \times 100 = 200\]

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:

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 (\f x -> f x) fs xs)
```

> 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 \( f <$> Just x <*> Just y = Just (f x y) \)

sequenceA1 [Just 3, Just 1]
= (:)$ Just 3 <*> sequenceA1 [Just 1]
= (:)$ Just 3 <*> ((:)$ Just 1 <*> sequenceA1 [])
= (:)$ Just 3 <*> ((:)$ Just 1 <*> pure [])
= (:)$ Just 3 <*> ((:)$ Just 1 <*> Just [])
= (:)$ Just 3 <*> Just [1]
= 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** :: Applicative f ⇒ (a -> b -> c) -> f a -> f b -> f c
  
  Passing (:) to `liftA2` makes b = [a] and c = [a], so
  
  ```haskell
  liftA2 (:) :: Applicative 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
  
  ```haskell
  foldr (liftA2 (:) ) :: Applicative f ⇒ f [a] -> [f a] -> f [a]
  ```

- **pure []** :: Applicative f ⇒ f [a]

  ```haskell
  foldr (liftA2 (:) ) (pure []) :: Applicative 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 \( f \) \( <*> \) \( x \) = \( \text{fmap} \ f \) \( x \) \hspace{1cm} -- \text{\( <*> \)}: \text{apply a boxed function}

pure \( \text{id} \) \( <*> \) \( x \) = \( x \) \hspace{1cm} -- \text{Because \( \text{fmap id} = \text{id} \)}

pure (\( . \)) \( <*> \) \( x \) \( <*> \) \( y \) \( <*> \) \( z \) = \( x \) \( <*> \) (\( y \) \( <*> \) \( z \)) \hspace{1cm} -- \text{\( <*> \) is left-to-right}

pure \( f \) \( <*> \) pure \( x \) = pure (\( f \) \( x \)) \hspace{1cm} -- \text{Apply a boxed function}

\( x \) \( <*> \) pure \( y \) = pure (\( \$ \) \( y \)) \( <*> \) \( x \) \hspace{1cm} -- \text{(\( \$ \) \( y \)): “apply arg. \( y \)”}
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.,</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>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:

| Prelude> 4 * 1        | Prelude> "hello" ++ []                |
| 4      -- 1 is the identity on the right | "hello"       -- [] is the right identity |
| Prelude> 1 * 4        | Prelude> [] ++ "hello"                |
| 4      -- 1 is the identity on the left     | "hello"       -- [] is the left identity |
| Prelude> 2 * (3 * 4)  | Prelude> "a" ++ ("bc" ++ "de")        |
| 24     | "abcde"                                 |
| Prelude> (2 * 3) * 4  | Prelude> ("a" ++ "bc") ++ "de"       |
| 24     -- * is associative                   | "abcde"       -- ++ is associative     |
| Prelude> 2 * 3        | Prelude> "a" ++ "b"                     |
| 6      | "ab"                                    |
| Prelude> 3 * 2        | Prelude> "b" ++ "a"                     |
| 6      -- * happens to be commutative        | "ba"          -- ++ is not commutative  |
|         |                                         |
The Monoid Type Class

class Monoid m where
  mempty :: a
  mappend :: m -> m -> m
  -- The identity value
  -- The associative binary operator

  mconcat :: [m] -> m
  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!"
*, 1 and +, 0 Can Each Make a Monoid

newtype lets us build distinct Monoids for each

In Data.Monoid,

```haskell
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)
```

```haskell
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
Ordering as a Monoid

\[
\text{data Ordering} = \text{LT} \mid \text{EQ} \mid \text{GT}
\]

In Data.Monoid,

\[
\begin{align*}
\text{instance Monoid Ordering where} \\
& \text{mempty} = \text{EQ} \\
& \text{LT} \ \text{mappend} \ _ = \text{LT} \\
& \text{EQ} \ \text{mappend} \ y = y \\
& \text{GT} \ \text{mappend} \ _ = \text{GT}
\end{align*}
\]

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

\[
\text{lcomp} :: \text{String} \rightarrow \text{String} \rightarrow \text{Ordering}
\]

"b" \ `lcomp` "aaaa" = LT \text{ -- b is shorter}
"bbbbbb" \ `lcomp` "a" = GT \text{ -- bbbbb is longer}
"avenger" \ `lcomp` "avenged" = LT \text{ -- Same length: r is after d}
A little too operational; \textit{mappend} is exactly what we want.
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)
The Foldable Type Class

What I taught you:

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

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

```
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 ++