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

**class Functor f where**

\[
\text{fmap :: (} \text{a} \rightarrow \text{b}) \rightarrow \text{f a} \rightarrow \text{f b}
\]

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

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

**data Maybe a = Just a | Nothing**

**instance Functor Maybe where**

\[
\text{fmap _ Nothing} = \text{Nothing}
\]

\[
\text{fmap g (Just x)} = \text{Just (g x)}
\]

**data Either a b = Left a | Right b**

**instance Functor (Either a) where**

\[
\text{fmap _ (Left x)} = \text{Left x}
\]

\[
\text{fmap g (Right y)} = \text{Right (g y)}
\]

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

**instance Functor List where**

\[
\text{fmap g (Cons x xs)} = \text{Cons (g x) (fmap g xs)}
\]

\[
\text{fmap _ Nil} = \text{Nil}
\]
**IO as a Functor**

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

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

IO does behave like a kind of box:

```haskell
query :: IO String
query = do line <- getLine
          let res = line ++ "!"
          return res
```

The definition of Functor IO in the Prelude: (alternative syntax)

```haskell
instance Functor IO where
  fmap f action = do result <- action
                      return (f result)
```
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)
```
Fmappings Functions: \( \text{fmap} \ f \ g = f \ . \ g \)

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

Prelude> \text{fmap} \ (*3) \ (+100) \ 1
303

Prelude> \((\ast3) \ `\text{fmap}` \ (+100) \) \$ \ 1
303

Prelude> \((\ast3) \ . \ (+100) \) \$ \ 1
303

Prelude> \text{fmap} \ (\text{show} \ . \ (*3)) \ (+100) \ 1
"303"
Partially Applying \textit{fmap}

\begin{verbatim}
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 \textit{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"
\end{verbatim}
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”):

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

\[
\text{fmap } (\lambda y \rightarrow f (g y)) \; x = \text{fmap } f (\text{fmap } g \; x) \quad -- \text{Equivalent}
\]
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
My So-Called Functor

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 Hakell are Curried:

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

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

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

\(\text{fmap}\) 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

\textbf{infixl 4 <*>}

\begin{verbatim}
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
\end{verbatim}

\textbf{instance} Applicative Maybe where
\begin{verbatim}
  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
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}
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

    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

\[
\text{instance Functor Maybe where}
\begin{align*}
\text{fmap } \_ \_ \text{ Nothing } &= \text{Nothing} \\
\text{fmap } g \ (\text{Just } x) &= \text{Just } (g \ x)
\end{align*}
\]

\[
\text{infixl 4 } \langle\$\rangle \\
f \langle\$\rangle x = \text{fmap } f \ x
\]

\[
\text{instance Applicative Maybe where}
\begin{align*}
\text{pure} &= \text{Just} \\
\text{Nothing } \langle\*\rangle \_ &= \text{Nothing} \\
\text{Just } f \ \langle\*\rangle m &= \text{fmap } f \ m
\end{align*}
\]

\[
f \langle\$\rangle \text{Just } x \ \langle\*\rangle \text{Just } y
\]
\[
= (f \langle\$\rangle \text{Just } x) \ \langle\*\rangle \text{Just } y \\
= (\text{fmap } f (\text{Just } x)) \ \langle\*\rangle \text{Just } y \\
= \ (\text{Just } (f \ x)) \ \langle\*\rangle \text{Just } y \\
= \ \text{fmap } (f \ x) \ (\text{Just } y) \\
= \ \text{Just } (f \ x \ y)
\]
**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

\[
\text{instance } \text{Applicative } \text{IO where}
\]
\[
\text{pure } = \text{return}
\]
\[
a <*> b = \text{do } f <- a
\]
\[
x <- b
\]
\[
\text{return } (f \ x)
\]

Specialized to IO actions,

\[
(<*>): \text{IO (a } \rightarrow \text{ b)} \\
\rightarrow \text{ IO a} \\
\rightarrow \text{ IO b}
\]

```
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

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

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

\langle*\rangle\;\text{takes}\; f :: a \rightarrow b \rightarrow c\;\text{and}\; g :: a \rightarrow b\;\text{and should produce}\; a \rightarrow c.

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

\textbf{instance} Applicative ((->) a) \textbf{where}

\[
\text{pure} \; x \; = \; \_ \rightarrow x
\]
\[
\quad -- \text{a.k.a., const}
\]
\[
f \; \langle*\rangle \; g \; = \; \_ \rightarrow f \; x \; (g \; x)
\]
\[
\quad -- \text{takes an a and uses f & g to produce a c}
\]

Prelude> \(\text{:t} \; f \; g \; x \rightarrow f \; x \; (g \; x)\)
\(\text{\textbackslash}f \; g \; x \rightarrow f \; x \; (g \; x) :: (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c\)
Functions as Applicative Functors

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

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
```
Functions as Applicative Functors

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:

- $2 + 3 = 5$
- $2 \times 3 = 6$
- $2 \times 100 = 200$

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]
[11,110,12,120,101,1010,20,200]

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

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

class Functor \( f \Rightarrow \) Applicative \( f \) where

\[
\begin{align*}
\text{pure} & \quad : \quad a \Rightarrow f\, a \\
(\text{<*>}) & \quad : \quad f\, (a \Rightarrow b) \Rightarrow f\, a \Rightarrow f\, b \\
\text{(<>*)} & = \text{liftA2 \ id} \quad -- \text{Default: get function from 1st arg’s box}
\end{align*}
\]

\[
\begin{align*}
\text{liftA2} & \quad : \quad (a \Rightarrow b \Rightarrow c) \Rightarrow f\, a \Rightarrow f\, b \Rightarrow f\, c \\
\text{liftA2} \, f \, x & = (\text{<*>}) \, (\text{fmap} \, f \, x) \quad -- \text{Default implementation}
\end{align*}
\]

\textit{liftA2} takes a binary function and “lifts” it to work on boxed values, e.g.,

\[
\begin{align*}
\text{liftA2} & \quad : \quad (a \Rightarrow b \Rightarrow c) \Rightarrow (f\, a \Rightarrow f\, b \Rightarrow f\, c)
\end{align*}
\]

\textbf{Prelude Control.Applicative> liftA2 (:) \ (Just \ 3) \ (Just \ [4])} \\
\textbf{Just \ [3,4]} \quad -- \text{Apply (:) inside the boxes, i.e., Just ((:) \ 3 \ [4])}

\textbf{instance Applicative ZipList where}

\[
\begin{align*}
\text{pure} \ x & = \text{ZipList} \, (\text{repeat} \, x) \\
\text{liftA2} \, f \, (\text{ZipList} \, xs) \, (\text{ZipList} \, ys) & = \text{ZipList} \, (\text{zipWith} \, f \, xs \, ys)
\end{align*}
\]
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?

```haskell
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

```haskell
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

```haskell
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 \( f \triangleleft\triangleright x \) = \( \text{fmap} \ f \ x \quad -- \triangleleft\triangleright: \text{apply a boxed function} \)

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

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

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

\( x \triangleleft\triangleright \text{pure } y = \text{pure} \ ($ y$) \triangleleft\triangleright x \quad -- ($ y$): “apply arg. y” \)
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., 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:

```
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
```

```
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

```haskell
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!"
*, 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)

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**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prelude Data.Monoid&gt; <code>mempty :: Sum Int</code></td>
<td>Sum {getSum = 0}</td>
</tr>
<tr>
<td>Prelude Data.Monoid&gt; <code>mempty :: Product Int</code></td>
<td>Product {getProduct = 1}</td>
</tr>
<tr>
<td>Prelude Data.Monoid&gt; <code>Sum 3 </code>mappend<code> Sum 4</code></td>
<td>Sum {getSum = 7}</td>
</tr>
<tr>
<td>Prelude Data.Monoid&gt; <code>Product 3 </code>mappend<code> Product 4</code></td>
<td>Product {getProduct = 12}</td>
</tr>
<tr>
<td>Prelude Data.Monoid&gt; <code>mconcat [Sum 1, Sum 10, Sum 100]</code></td>
<td>Sum {getSum = 111}</td>
</tr>
</tbody>
</table>
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

```
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 `lcomp` for strings ordered by length then alphabetically, e.g.,

```
lcomp :: String -> String -> Ordering

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

\begin{verbatim}
lcomp :: String \rightarrow String \rightarrow Ordering
lcomp x y = case length x `compare` length y of
  LT \rightarrow LT
  GT \rightarrow GT
  EQ \rightarrow x `compare` y
\end{verbatim}
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 _ a [] = a
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 ++