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 `* -> *`. “A box of”

`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 \( \star \to \star \), which is the kind of \( \text{IO} \)

Prelude> :k IO
IO :: \( \star \to \star \)

IO does behave like a kind of box:

\[
\begin{align*}
\text{query} &:: \text{IO String} \\
\text{query} &= \text{do} \ \text{line} \gets \text{getLine} \quad \text{-- getLine returns a box :: IO String} \\
&\quad \text{let} \ \text{res} = \text{line} + "!" \quad \text{-- take line out of box from getLine} \\
&\quad \text{return} \ \text{res} \quad \text{-- put res in an IO box}
\end{align*}
\]

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

\[
\begin{align*}
\text{instance \ Functor \ IO \ where} \\
\quad \text{fmap} \ f \ \text{action} &= \text{do} \ \text{result} \gets \text{action} \quad \text{-- take result from the box} \\
&\quad \text{return} \ (f \ \text{result}) \quad \text{-- apply f; put it a box}
\end{align*}
\]
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: \( \text{fmap } f \circ g = f \cdot g \)

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

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

Prelude> (\ast 3) `\text{fmap``} (+100) $ 1
303

Prelude> (\ast 3) . (+100) $ 1
303

Prelude> \text{fmap } (\text{show } \cdot (\ast 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"): 

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

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

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

```
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")
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

```haskell
instance Applicative IO where
  pure = return
  a <*> b = do f <- a
               x <- b
               return (f x)
```

Specialized to IO actions,

```haskell
(<>*) :: IO (a -> b)
       -> IO a
       -> IO b
```

```haskell
main = do
  a <- getLine
  b <- getLine
  putStrLn $ a ++ b
```

```haskell
main :: IO ()
main = do
  a <- (++ <$> getLine <*> getLine)
  putStrLn a
```

```
$ stack runhaskell af2.hs
One
Two
OneTwo
```
Function Application as an Applicative Functor

For the function type constructor ((->) a), the types for Applicative are

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

The definitions almost follow directly from these types:

```
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 = (.)
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 * 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:

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

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**

```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

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.,

```
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 with a list

```
sequenceA :: Applicative f => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```
Applicative Functor Laws

pure \( f \triangleright\triangleright x \) = \( \text{fmap} \ f \ x \)
pure \( \text{id} \triangleright\triangleright x \) = \( x \)
pure \( (.) \triangleright\triangleright x \triangleright\triangleright y \triangleright\triangleright z \) = \( x \triangleright\triangleright (y \triangleright\triangleright z) \)
pure \( f \triangleright\triangleright \text{pure} \ x \) = \( \text{pure} \ (f \ x) \)
x \triangleright\triangleright \text{pure} \ y = \text{pure} \ (\_ y) \triangleright\triangleright x
newtype
Monoids