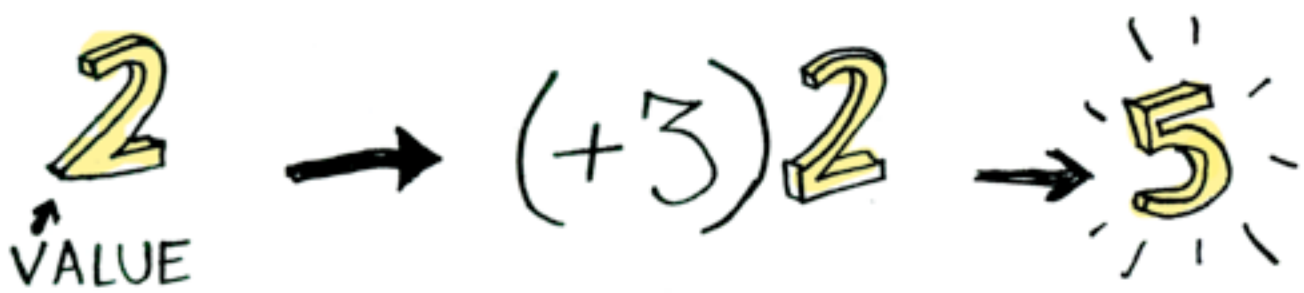
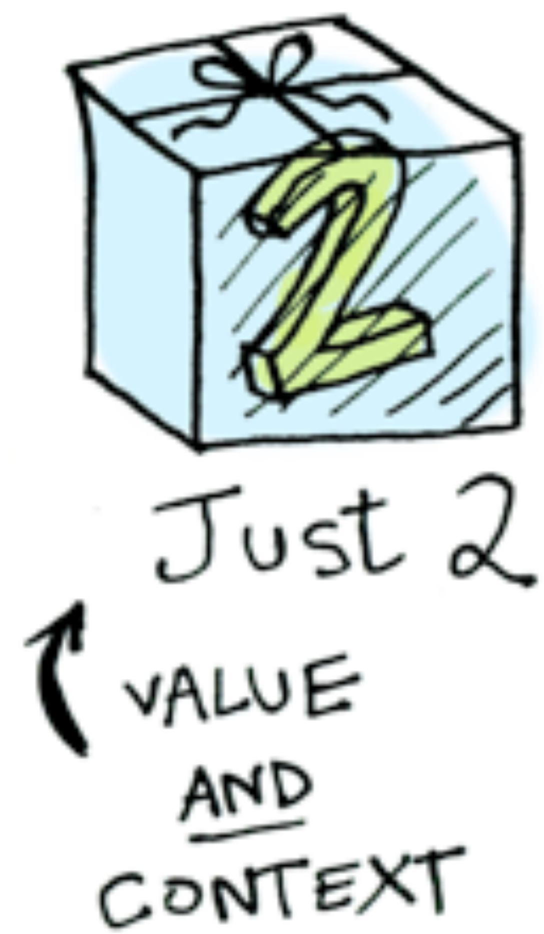
Functors
Applicative Functors
Monads

Material based upon:
http://learnyouahaskell.com
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

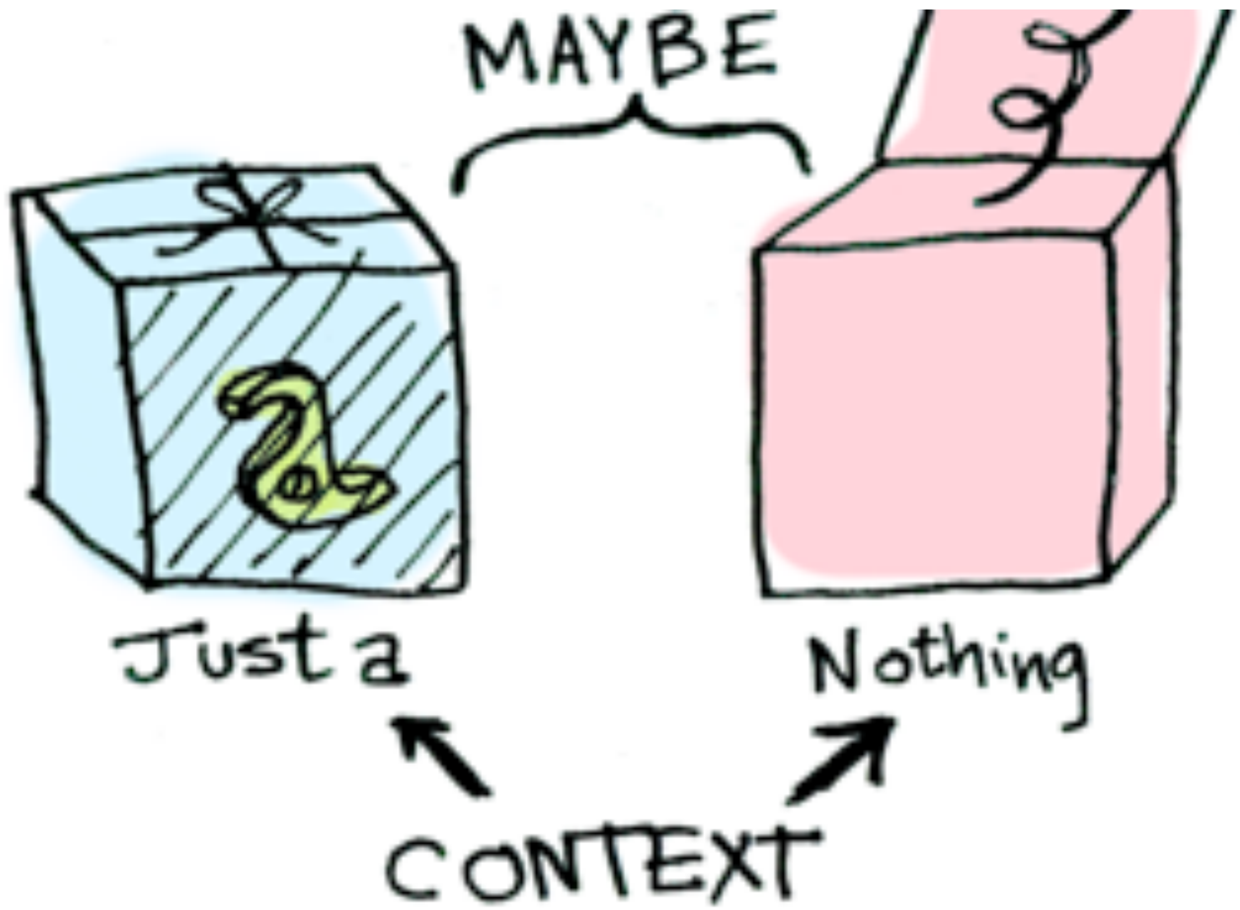
Values



Values in context

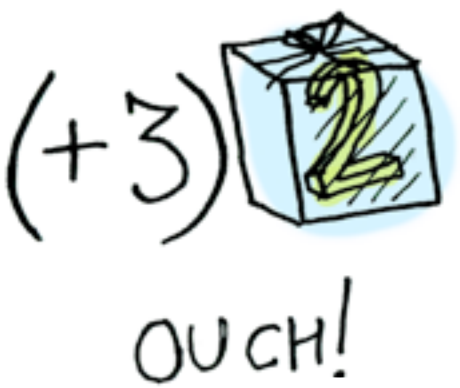


Values in context

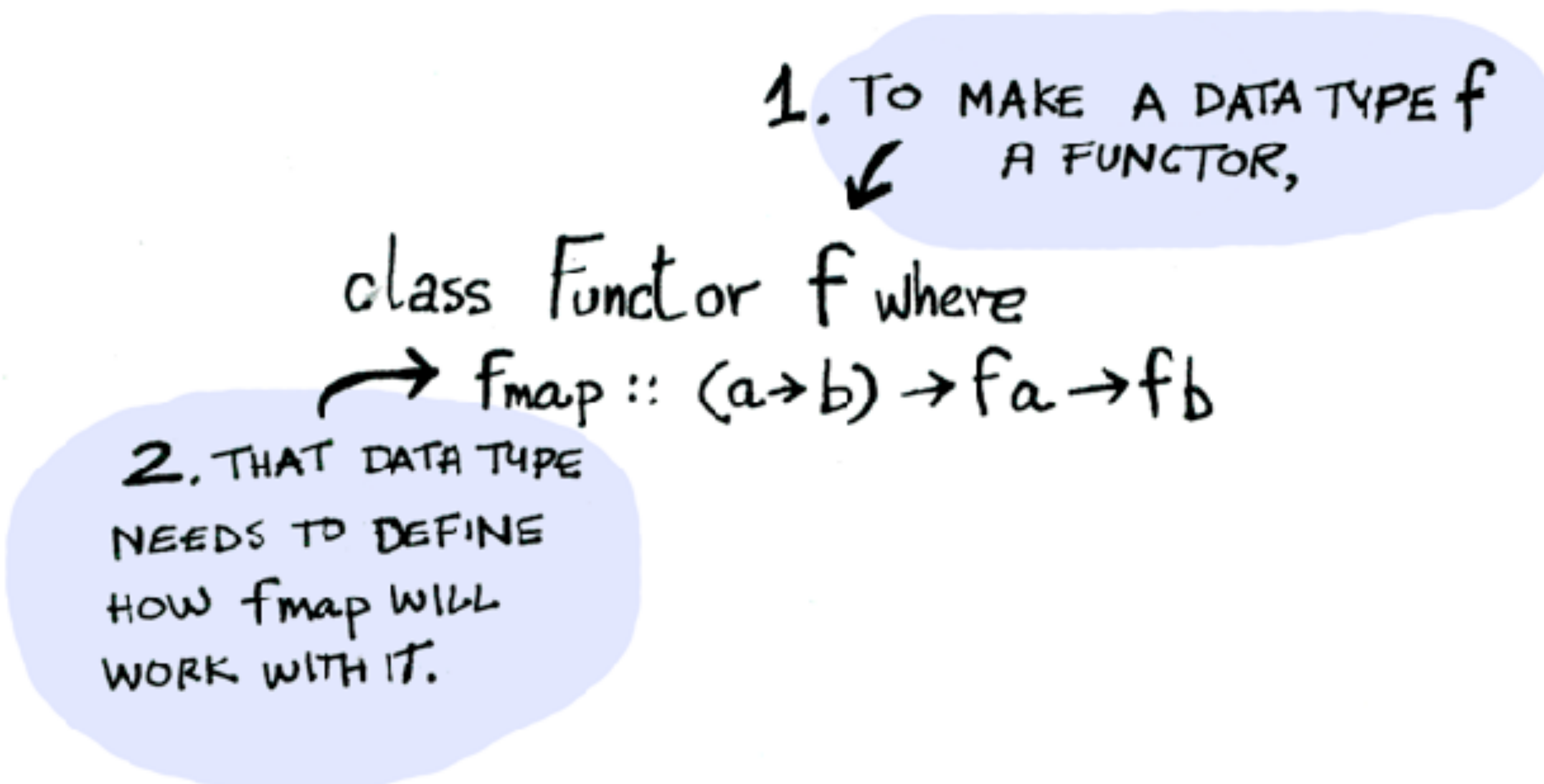



```
data Maybe a = Nothing | Just a
```

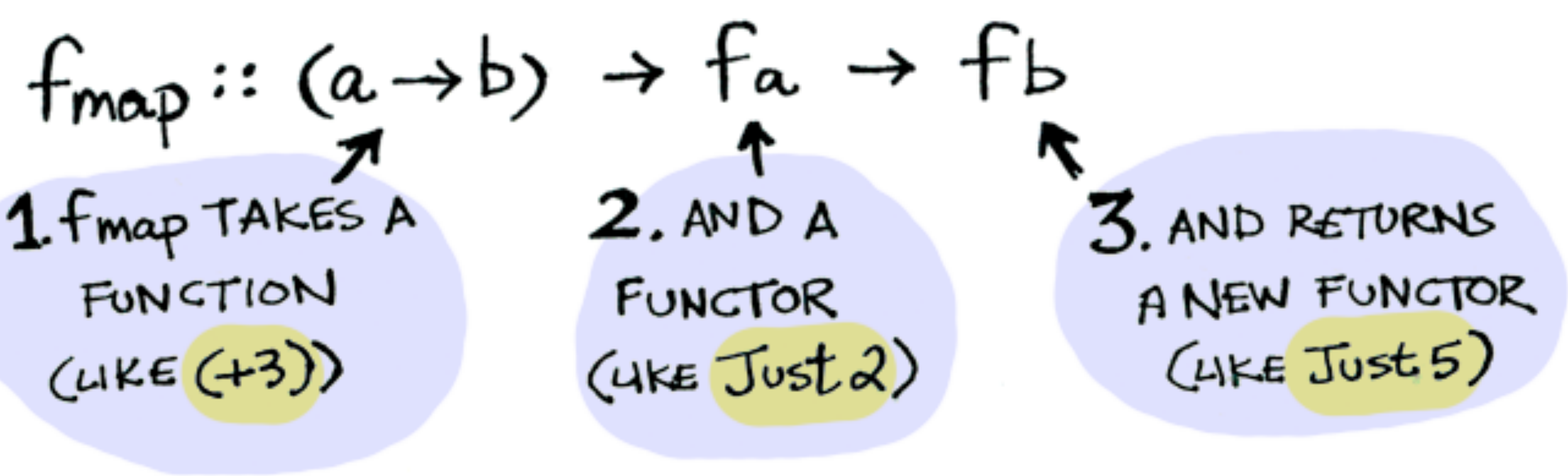
Functors




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



Maybe Functor

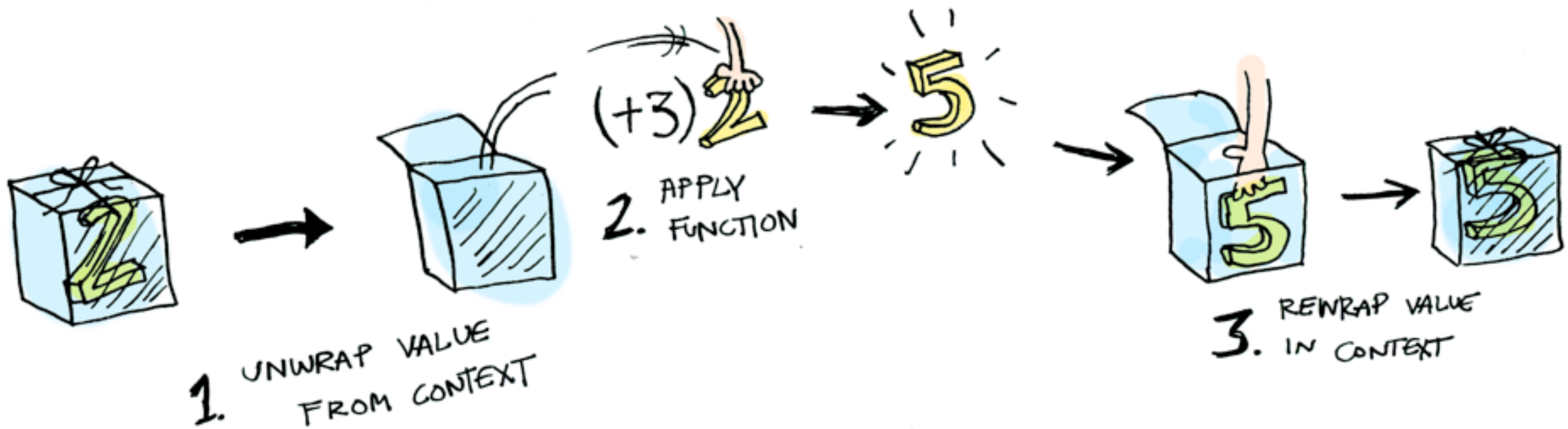



```
instance Functor Maybe where
    fmap func (Just val) = Just (func val)
    fmap func Nothing = Nothing
```

```
> (+3) <$> Just 2
Just 5
```

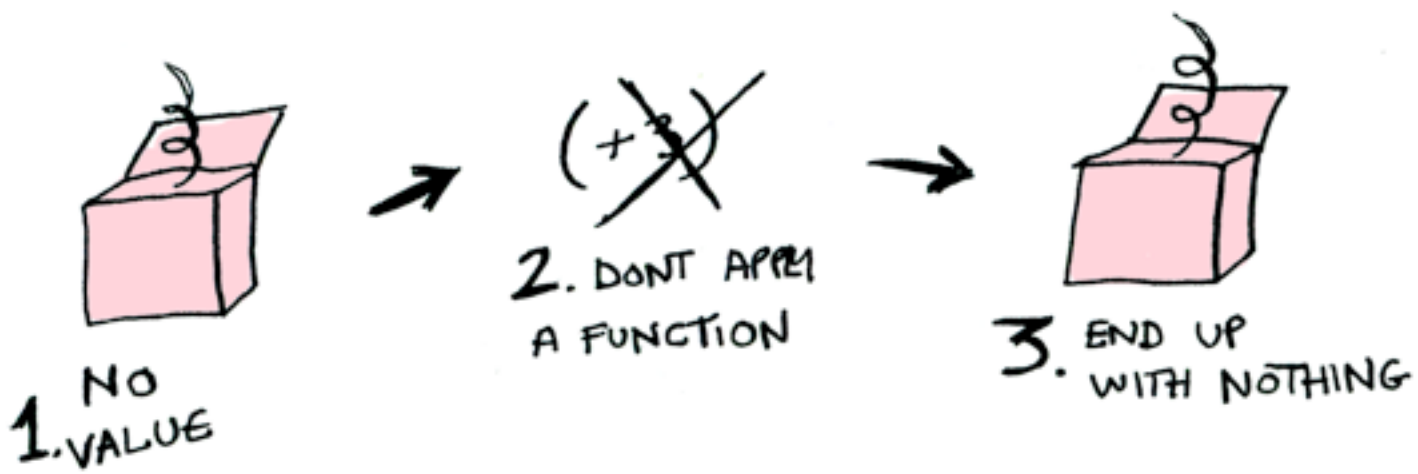
fmap

Maybe Just



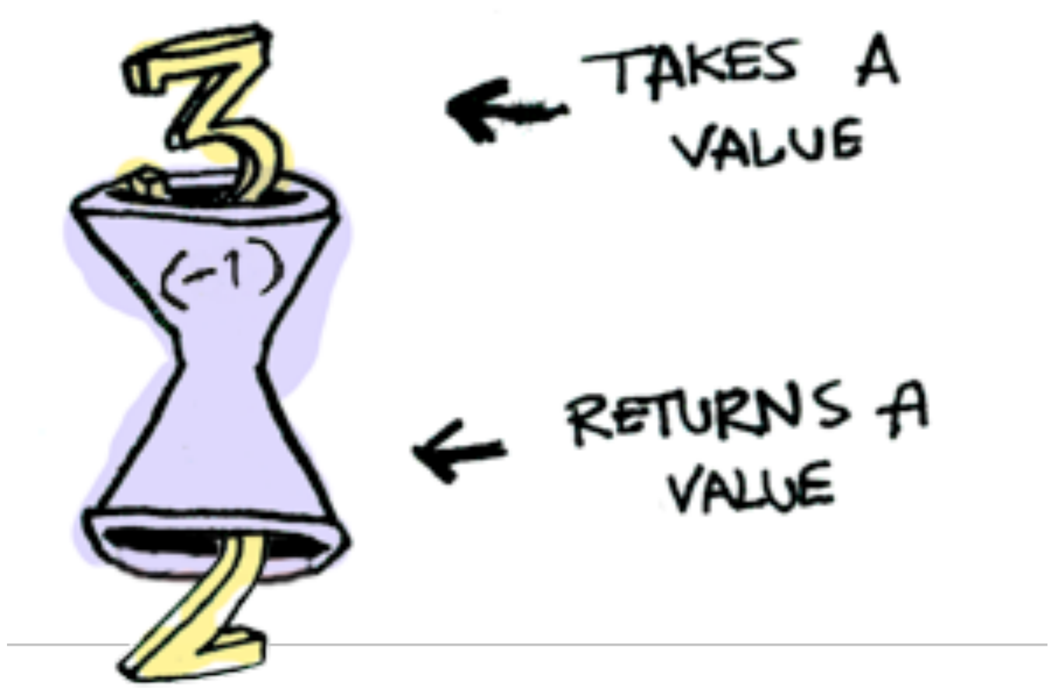
1. UNWRAP VALUE FROM CONTEXT
(+3)
2
2. APPLY FUNCTION
5
3. REWRAP VALUE IN CONTEXT

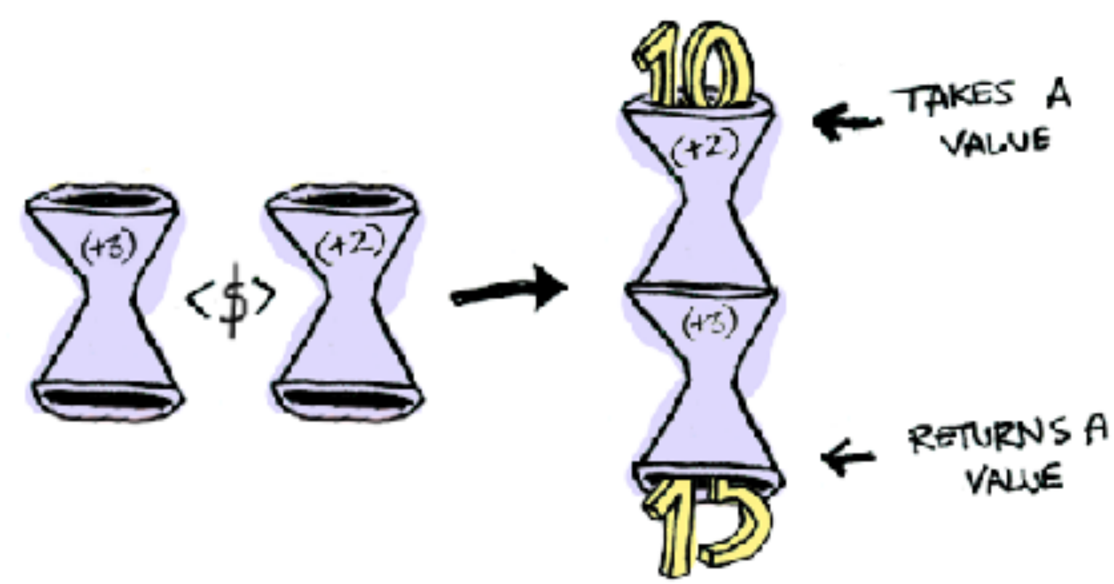
Maybe Nothing



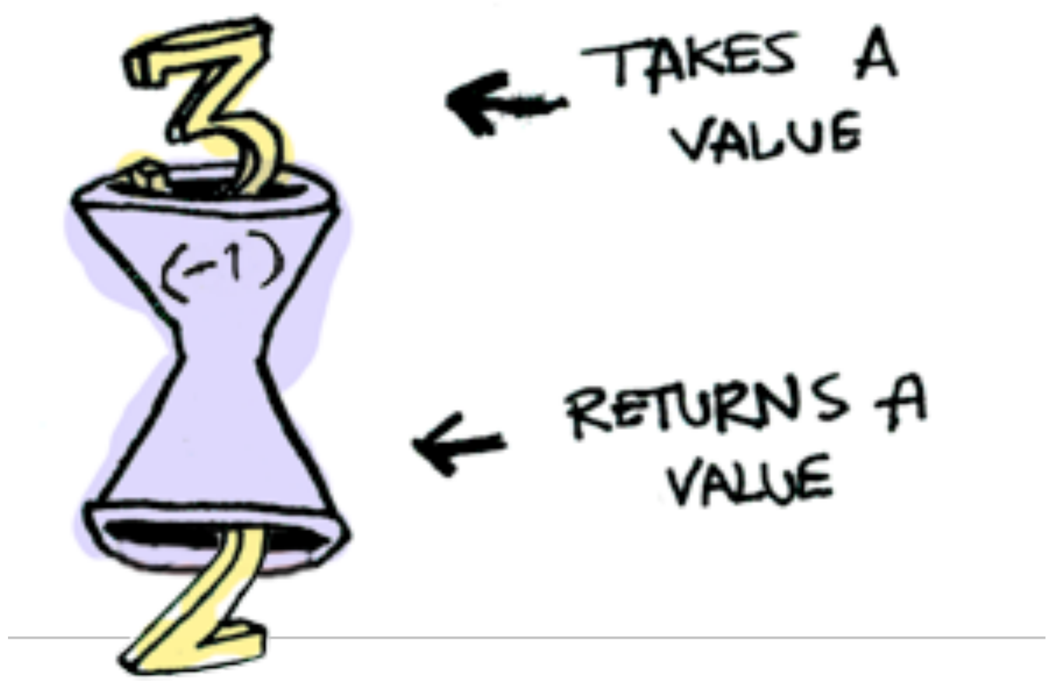

```
> fmap (+3) Nothing
Nothing
```

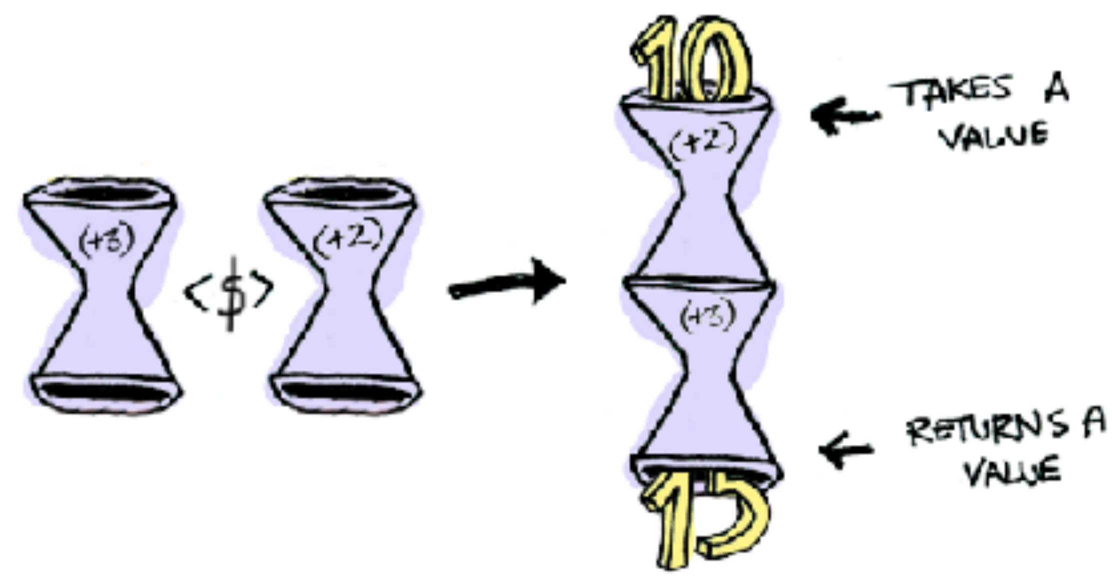
Function Functor





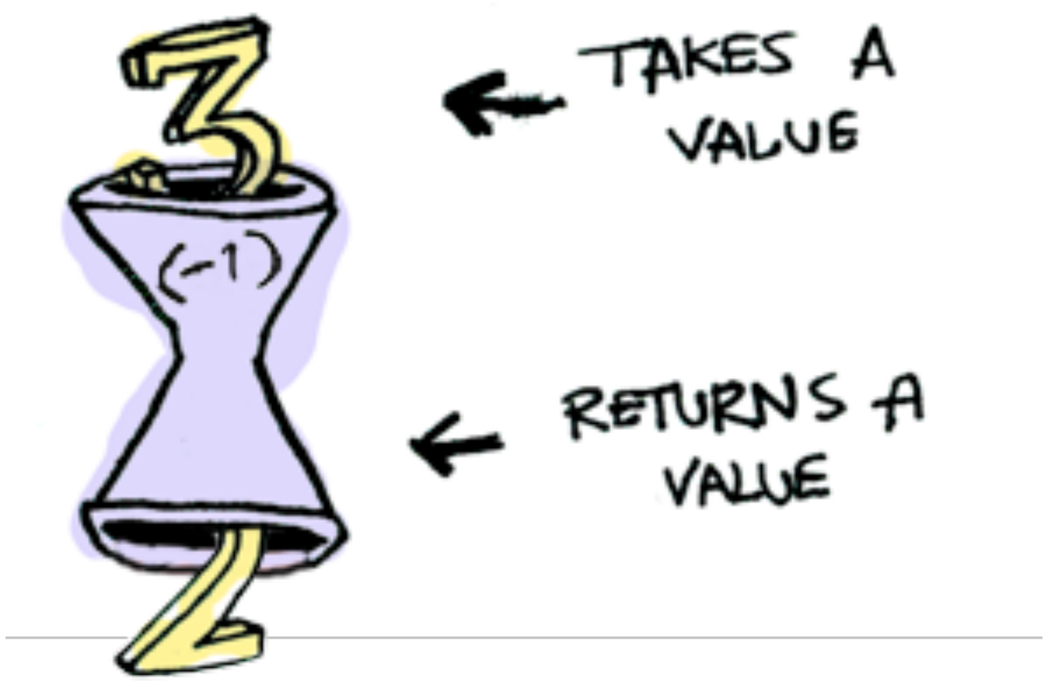
Function Functor

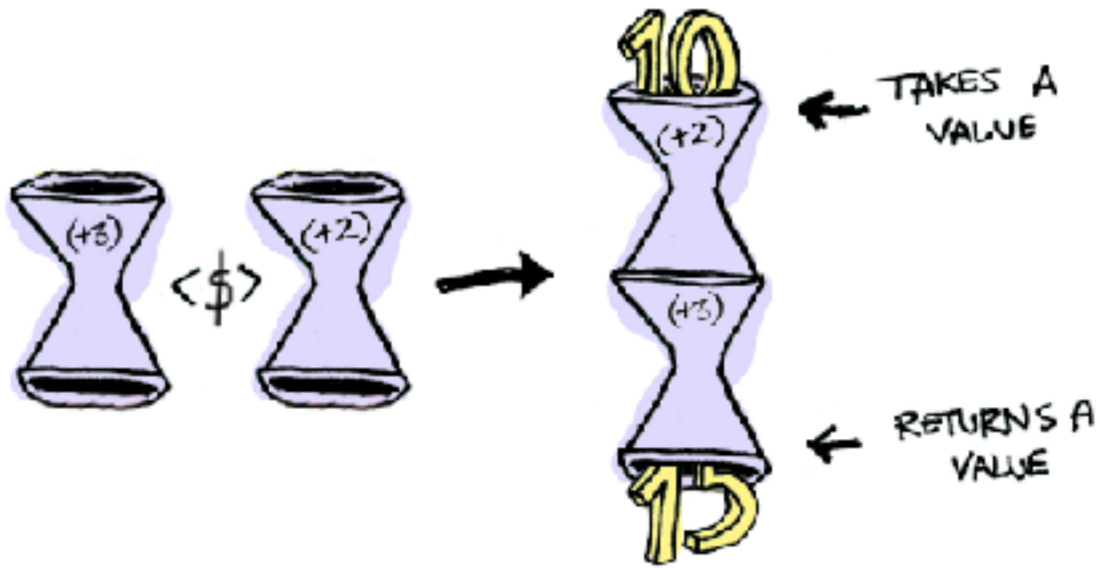





```
> let foo = fmap (+3) (+2)
> foo 10
15
```

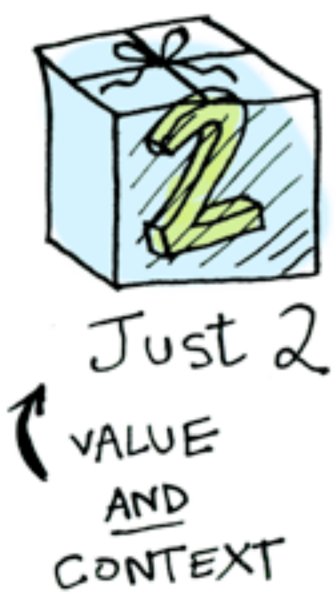
Function Functor






```haskell
instance Functor ((->) r) where
    fmap = .
```

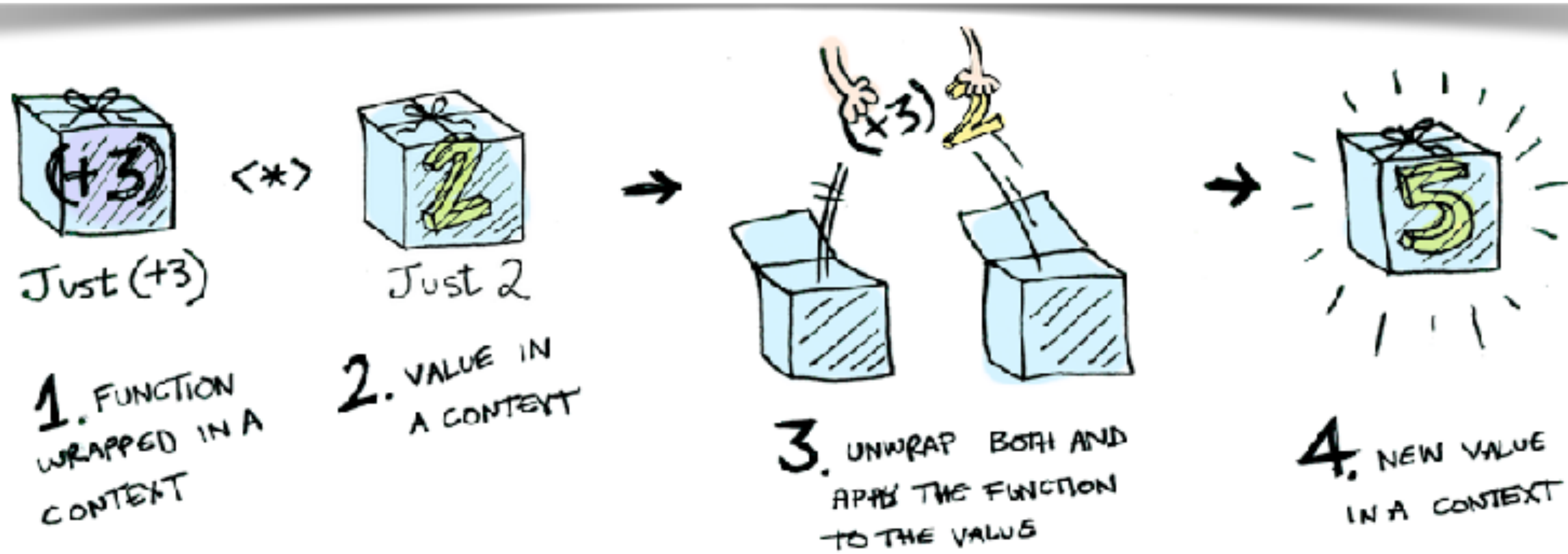
Applicative



Applicative

```
class Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
  pure  :: a -> f a
```

```
> Just (+3) <*> Just 2
Just 5
```

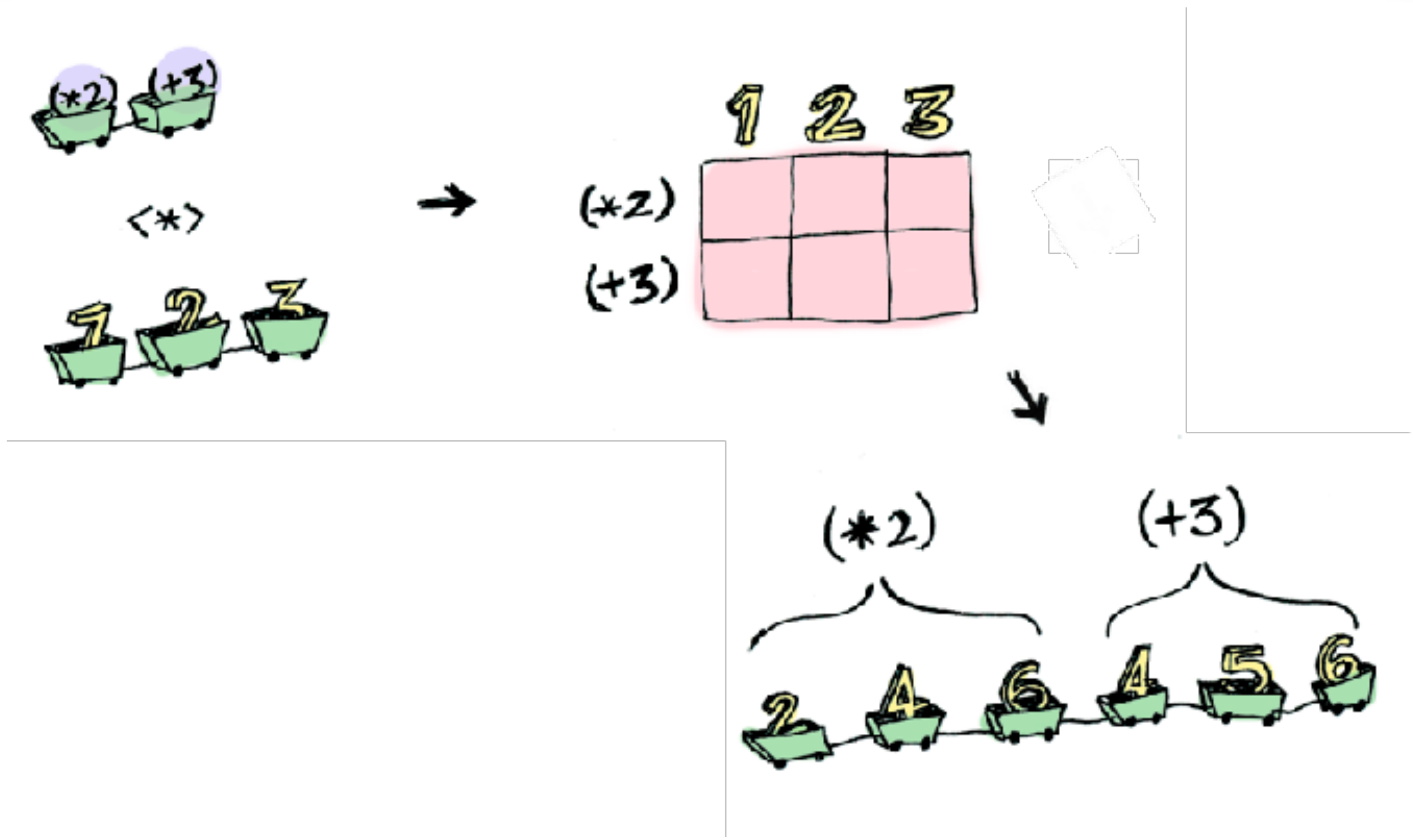


List context

```
> [(+2),(+3)] <*> [1,2,3]
```

List context

```
> [(+2),(+3)] <*> [1,2,3]
[3,4,5,4,5,6]
```



Do we understand ?!!?

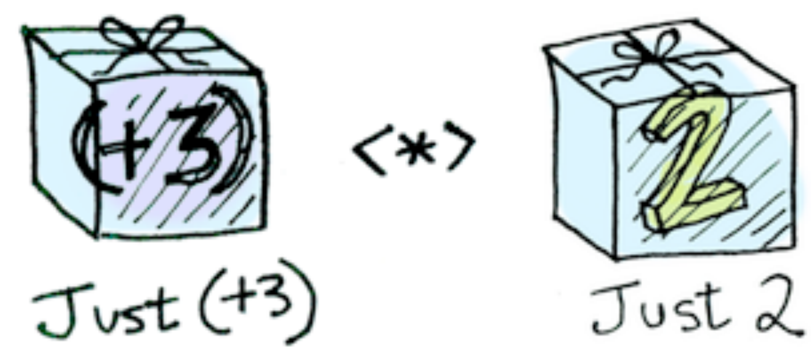
```
> (+) <$> Just 2 <*> Just 3
```

Do we understand ?!!?

```
> (+) <$> Just 2 <*> Just 3
Just 5
```

Overview till now

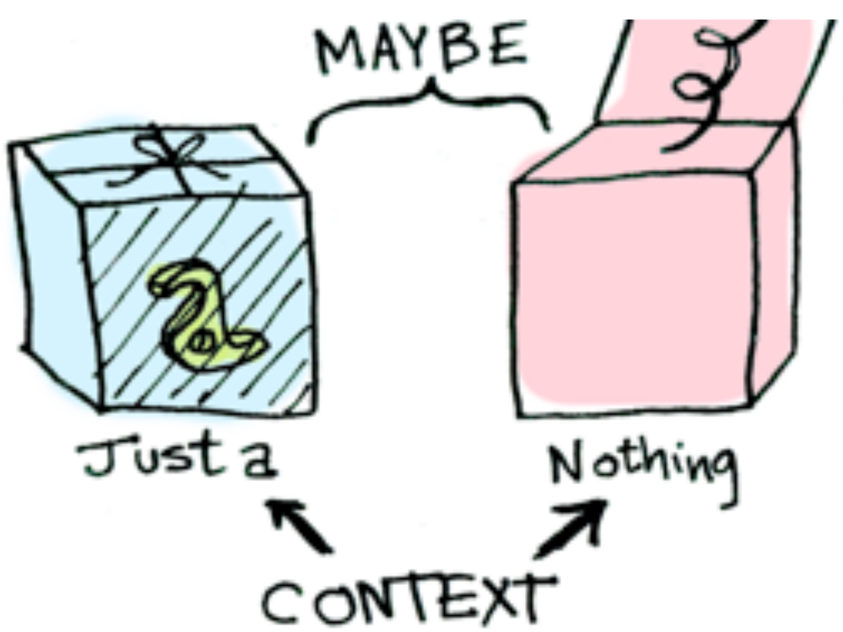
Functor

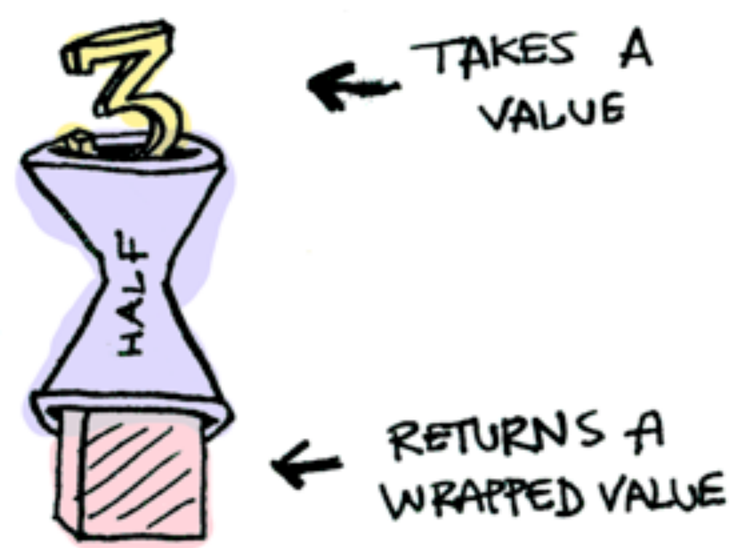


Applicative

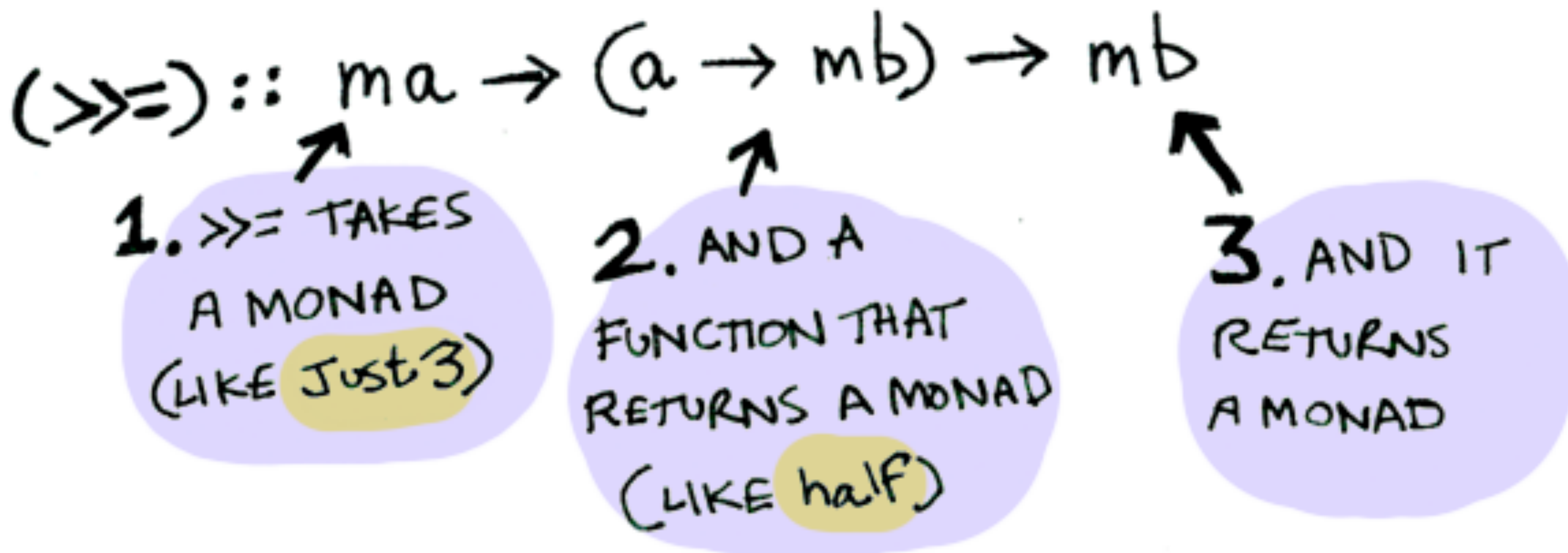
Monad

```
half x = if even x
           then Just (x `div` 2)
           else Nothing
```



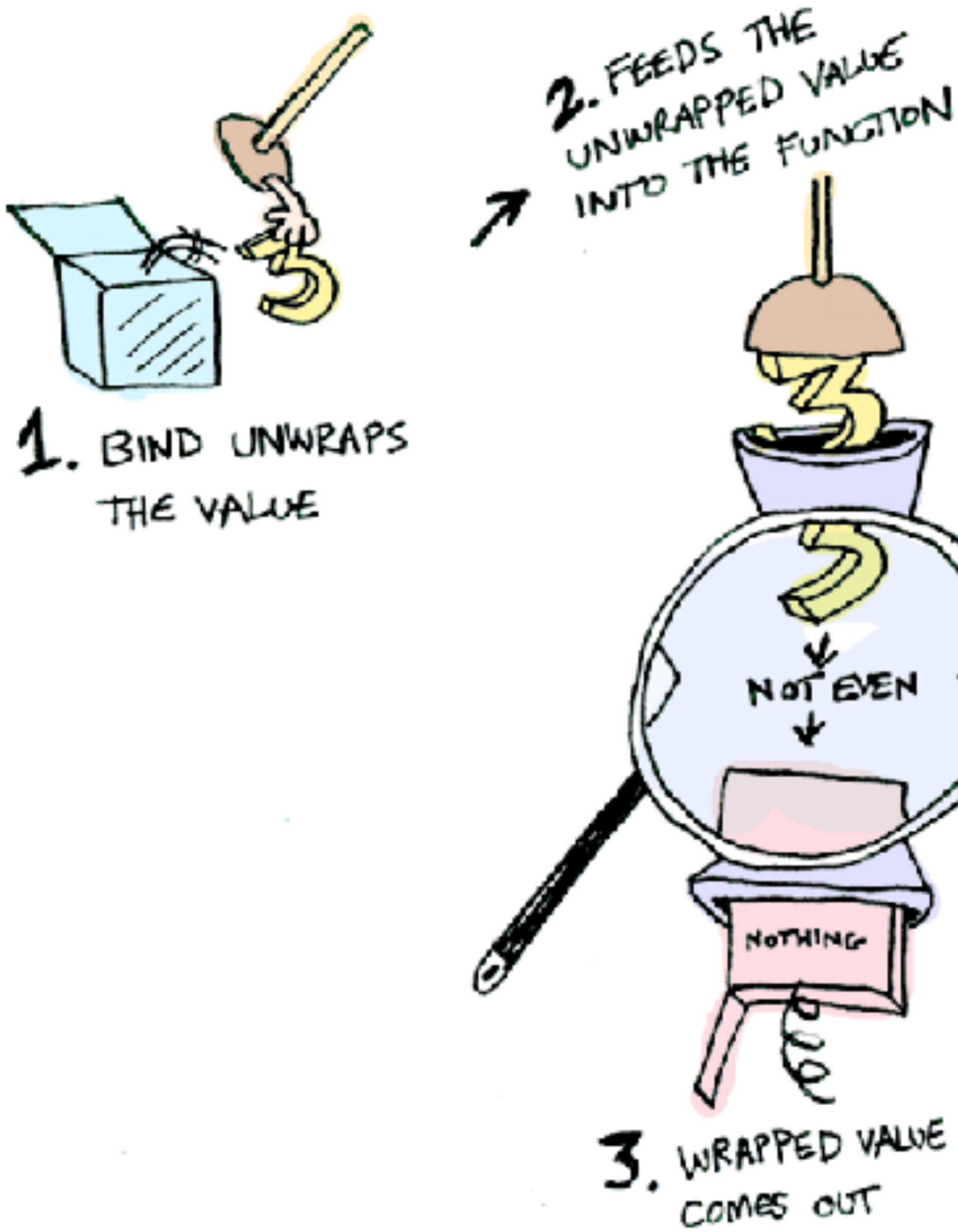


Monad



(>>=) :: ma → (a → mb) → mb
1. >>= TAKES A MONAD (LIKE Just 3)
2. AND A FUNCTION THAT RETURNS A MONAD (LIKE half)
3. AND IT RETURNS A MONAD

Maybe Monad

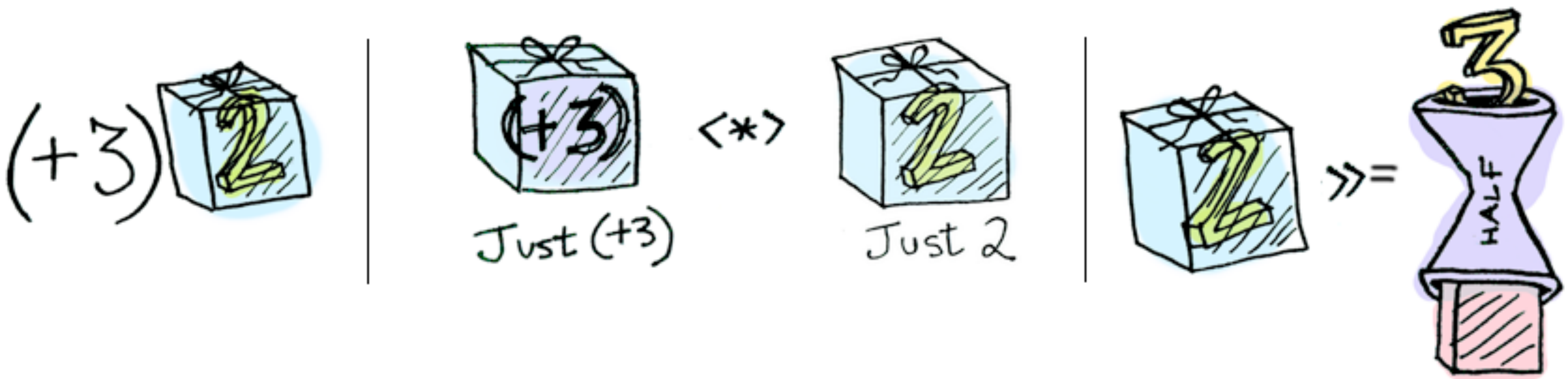



```haskell
instance Monad Maybe where
    Nothing  >>= func  = Nothing
    Just val >>= func  = func val
```

Maybe

Cool stuff! So now we know that Maybe is a **Functor**, an **Applicative**, and a **Monad**.

Overview



Functor Applicative Monad

Monads <: Applicative

```
class Applicative f where
 pure   :: a -> f a
 (<*>)  :: f (a -> b) -> f a -> f b
```

```
class Monad m  where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

Monads <: Applicative

```
class Applicative f where
   pure  :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

```
class Monad m  where
  return :: a -> m a
   (>>=) :: m a -> (a -> m b) -> m b
```

```
import Control.Monad
myap :: (Monad m) =>  (m (x->y)) -> m x -> m y
myap  m1 m2  =
```

Monads <: Applicative

```haskell
class Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

```haskell
class Monad m  where
  return :: a -> m a
   (>>=) :: m a -> (a -> m b) -> m b
```

```haskell
import Control.Monad
myap :: (Monad m) =>  (m (x->y)) -> m x -> m y
myap  m1 m2  = do { f <- m1; x2 <- m2; return (f x2) }
```

Monads <: Applicative

```
class Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

```
class Monad m  where
  return :: a -> m a
   (>>=) :: m a -> (a -> m b) -> m b
```

```
import Control.Monad
myap :: (Monad m) =>  (m (x->y)) -> m x -> m y
myap  m1 m2  = do { f <- m1; x2 <- m2; return (f x2) }
```

```
> Just (+2) `myap` Just 2
Just 4
```

Applicative <: Functor

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

```
class Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Applicative <: Functor

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

```
class Applicative f where
   pure  :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

```
myfmap :: (Applicative f) =>  (a->b) -> f a -> f b
myfmap g a1 =
```

Applicative <: Functor

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

```
class Applicative f where
   pure  :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

```
myfmap :: (Applicative f) =>  (a->b) -> f a -> f b
myfmap g a1 = (pure g) <*> a1
```

Applicative <: Functor

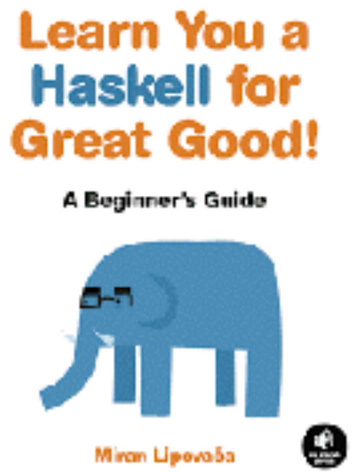
```
class Functor f where
 fmap :: (a -> b) -> f a -> f b
```

```
class Applicative f where
   pure  :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

```
myfmap :: (Applicative f) =>  (a->b) -> f a -> f b
myfmap g a1 = (pure g) <*> a1
```

```
Main> (\a -> a +1) `myfmap` Just 2
Just 3
```

In the book

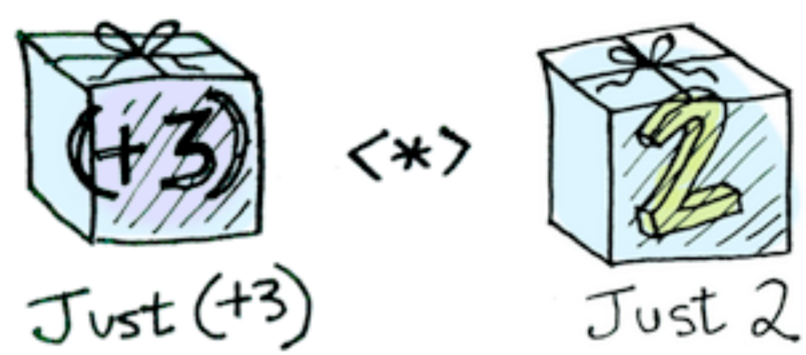


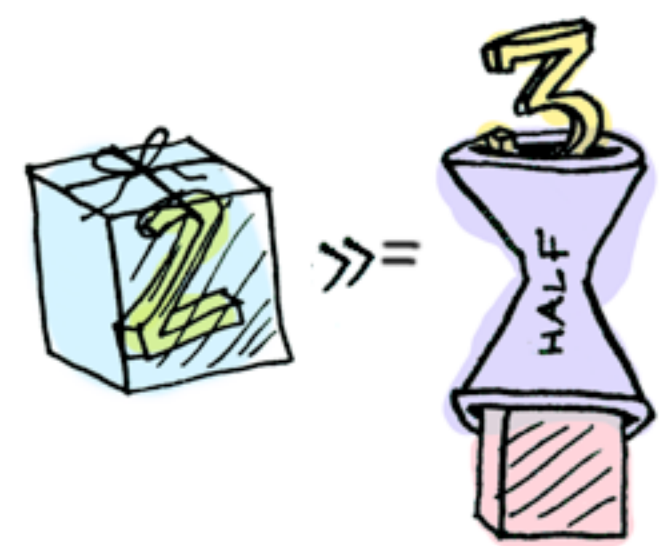
*“So every **monad** is an **applicative functor** and every **applicative functor** is a **functor**.*

The Applicative type class has a class constraint such that our type has to be an instance of Functor before we can make it an instance of Applicative.

But even though Monad should have the same constraint for Applicative, as every monad is an applicative functor, it doesn't, because the Monad type class was introduced to Haskell way before Applicative.”

GHC 7.10



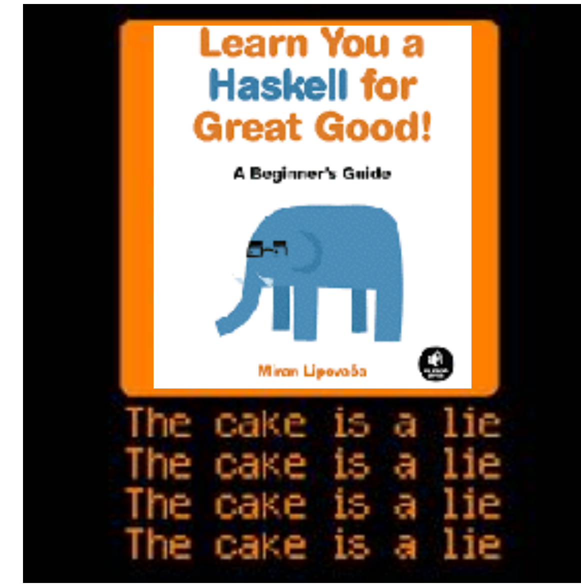


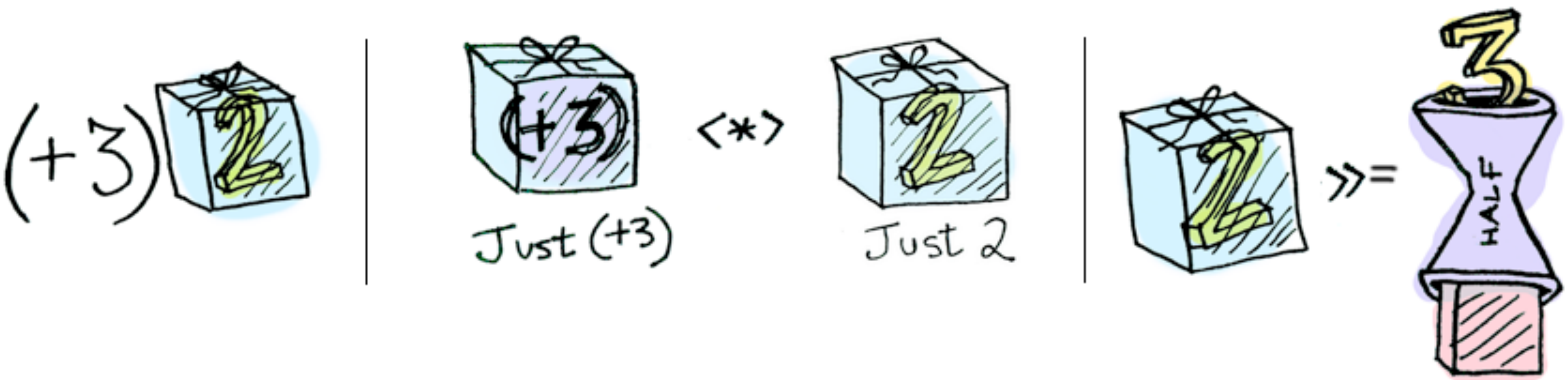

```
Functor f
```

```
=> Applicative f
```

```
=> Monad f
```

GHC 7.10






```
Functor f    => Applicative f    => Monad f
```

Errors

```haskell
data Ofwel e a =  Links e | Rechts a
```

```haskell
instance Monad (Ofwel e)  where
     return  a             = Rechts a
     (Rechts  a)  >>= f    = f a
     (Links   e)  >>= f    = Links e
```

Errors

```
data Ofwel e a =  Links e | Rechts a
```

```
instance Monad (Ofwel e)  where
     return  a            =
     (Rechts  a)  >>= f   =
     (Links   e)  >>= f   =
```

Errors

```haskell
data Ofwel e a =  Links e | Rechts a
```

```haskell
instance Monad (Ofwel e)  where
     return  a               = Rechts a
     (Rechts  a)  >>= f      = f a
     (Links   e)  >>= f      = Links e
```

```
appli.hs:25:10:
    No instance for (Applicative (Ofwel e))
      arising from the superclasses of an instance declaration
    In the instance declaration for ‘Monad (Ofwel e)’
Failed, modules loaded: none.
```

Errors

```
data Ofwel e a =  Links e | Rechts a
```

```
instance Monad (Ofwel e)  where
     return  a              = Rechts a
     (Rechts  a)  >>= f     = f a
     (Links   e)  >>= f     = Links e
```

```
instance Applicative (Ofwel e) where
       pure  = return
       (<*>) = ap
```

Errors

```
instance Monad (Ofwel e)  where
     return  a             = Rechts a
     (Rechts  a)  >>= f    = f a
     (Links   e)  >>= f    = Links e
```

```
instance Applicative (Ofwel e) where
       pure  = return
       (<*>) = ap
```

```
appli.hs:21:10:
    No instance for (Functor (Ofwel e))
      arising from the superclasses of an instance declaration
    In the instance declaration for ‘Applicative (Ofwel e)’
```

Errors

```
*Main> (Rechts 3)  >>= (\a-> Rechts $ a+2 )
Rechts 5
*Main> (Links "oeps")  >>= (\a-> Rechts $ a+2 )
Links "oeps"
```

IO & More Monads !

Christophe Scholliers

Material based upon:
http://learnyouahaskell.com
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

IO ()

```
main = putStrLn "hello, world"
```

PutStrLn returns a command

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

getLine

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!”)

ghci> :t getLine
getLine :: IO String
```

getLine

```haskell
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ "Your future is: " ++ tellFortune name
```

Correct ?

```
nameTag = "Hello, my name is " ++ getLine
```

Correct ?

```
main = do
    return ()
    return "HAHAHA"
    line <- getLine
    return "BLAH BLAH BLAH"
    return 4
    putStrLn line
```

Return for binding

```
main = do
    a <- return "hell"
    b <- return "yeah!"
    putStrLn $ a ++ " " ++ b
```

Let in do syntax

```haskell
main = do
    let a = "hell"
        b = "yeah"
    putStrLn $ a ++ " " ++ b
```

Mapping IO

```
Prelude> map print [1,2,3,4,5]
```

Mapping IO

```
Prelude> map print [1,2,3,4,5]

<interactive>:2:1:
    No instance for (Show (IO ())) arising from a use of ‘print’
    In a stmt of an interactive GHCi command: print it
```

Mapping IO

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
ghci> mapM_ print [1,2,3]
1
2
3
```

```
mapM :: (Monad m, Traversable t) => (a -> m b) -> t a -> m (t b)
```

ForM

```haskell
import Control.Monad

main = do
    colors <- forM [1,2,3,4] (\a -> do
        putStrLn $ "Which color do you associate with the number " ++ show a ++ "?"
        color <- getLine
        return color)
    putStrLn "The colors that you associate with 1, 2, 3 and 4 are: "
    mapM putStrLn colors
```

Forever !

```
getContents :: IO String
```

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l

Prelude Control.Monad> :t forever
forever :: Monad m => m a -> m b
```

Files & Streams

Get content

```haskell
getContents :: IO String
```

```haskell
import Data.Char

main = do
    contents <- getContents
    putStr (map toUpper contents)
```

Please give me some `contents`

Ok I **promise** if you need it I will give you one

Hey `map` could you caps lock all the things this **promise** will ever give back ?

Sure if you really really need it I will do it I promise !

Hey `putStr` this promise if you will ?

WHAT a promise you better give me something real to print.

Hey Map upper will you ?

Hey contents please give me a line !

Ok here you go you all the chars on the first line just call me if you would need the rest :)

```haskell
import Data.Char

main = do
    contents <- getContents
    putStr (map toUpper contents)
```

Please not too long

```haskell
main = do
    contents <- getContents
    putStr (shortLinesOnly contents)

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

```
i'm short
so am i
i am a looooooooooong line!!!
yeah i'm long so what hahahaha!!!!!!
short line
loooooooooooooooooooooooooong
short
```

```
i'm short
so am i
short
```

Interact

```
interact :: (String -> String) -> IO ()
```

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

```
i'm short
so am i
i am a looooooooooong line!!!
yeah i'm long so what hahahaha!!!!!!
short line
loooooooooooooooooooooooooooong
short
```

```
i'm short
so am i
short
```

Point Free Version

argument

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

Point Free Version

argument

```
main = interact $ unlines . filter ( (<10) . length) . lines
```

Point Free Version

argument

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

```
main = interact $ unlines . filter ((<10) . length) . lines
```

Reading Files

```haskell
openFile :: FilePath -> IOMode -> IO Handle
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
type FilePath = String
```

```haskell
import System.IO

main = do
    handle   <- openFile “Douglas_Adams.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

Automatically Closing

```haskell
import System.IO

main = do
    withFile "Douglas_Adams.txt" ReadMode
        (\handle -> do
            contents <- hGetContents handle
            putStr contents)
```

Implementing withFile

```haskell
withFile' :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile' path mode f = do
    handle <- openFile path mode
    result <- f handle
    hClose handle
    return result
```

File functions

```
hGetLine, hPutStr, hPutStrLn, hGetChar
```

Exceptions

Sometimes things fail

```
Prelude> 4 `div` 0
*** Exception: divide by zero
```

IO errors

```haskell
catch :: Exception e => IO a -> (e -> IO a) -> IO a
```

```haskell
import System.Environment
import System.IO
import System.IO.Error
import Control.Exception.Base

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e = putStrLn "Whoops, had some trouble!"
```

More monads

Logging

```haskell
mul :: Int -> Int -> Int
mul x y =  x * y

fac :: Int -> Int
fac 0 = 1
fac n = n `mul` fac (n - 1)
```

Logging

```
mul' :: Int -> Int -> (Int,String)
mul' x y = (x*y, "multiplied [" ++ (show x) ++  " *" ++ (show y) ++ "]\n ")


fac' :: Int -> (Int,String)
fac' 0 = (1,"fac of one \n")
fac' n = (y, log1 ++ log2)
        where (x,log1) = fac' (n-1)
              (y,log2) = n `mul'` x
```

The Writer Monad

```
newtype Writer w a = Writer { runWriter :: (a, w) }

instance (Monoid w) => Monad (Writer w) where
    return x = Writer (x, mempty)
    (Writer (x,v)) >>= f = let (Writer (y, v')) = f x in
                                   Writer (y, v `mappend` v')
```

Running

```
ghci> runWriter (return 3 :: Writer String Int)
(3,"")
ghci> runWriter (return 3 :: Writer (Sum Int) Int)
(3,Sum {getSum = 0})
ghci> runWriter (return 3 :: Writer (Product Int) Int)
(3,Product {getProduct = 1})
```

Running

```
import Control.Monad.Writer

logNumber :: Int -> Writer [String] Int
logNumber x = Writer (x, ["Got number: " ++ show x])

multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    return (a*b)

ghci> runWriter multWithLog
(15,["Got number: 3","Got number: 5"])
```

Running

```
tell :: MonadWriter w m => w -> m ()
```

```
multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    tell ["Gonna multiply these two"]
    return (a*b)

ghci> runWriter multWithLog
(15,["Got number: 3","Got number: 5","Gonna multiply these two"])
```

Logging in programs

```haskell
import Control.Monad.Writer

gcd' :: Int -> Int -> Writer [String] Int
gcd' a b
    | b == 0 = do
        tell ["Finished with " ++ show a]
        return a
    | otherwise = do
        tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
        gcd' b (a `mod` b)


ghci> fst $ runWriter (gcd' 8 3)
1
```

Logging in programs

```haskell
import Control.Monad.Writer

gcd' :: Int -> Int -> Writer [String] Int
gcd' a b
    | b == 0 = do
        tell ["Finished with " ++ show a]
        return a
    | otherwise = do
        tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
        gcd' b (a `mod` b)


ghci> mapM_ putStrLn $ snd $ runWriter (gcd' 8 3)
8 mod 3 = 2
3 mod 2 = 1
2 mod 1 = 0
Finished with 1
```

State

Transforming state

S

S’

f

Keeping track of state

```
s -> (a,s)
```

Input

result

new state

Statefull Stack

```
type Stack = [Int]

pop :: Stack -> (Int,Stack)
pop (x:xs) = (x,xs)

push :: Int -> Stack -> ((),Stack)
push a xs = ((),a:xs)
```

Stack operations

```haskell
stackManip :: Stack -> (Int, Stack)
stackManip stack = let
    ((),newStack1) = push 3 stack
    (a ,newStack2) = pop newStack1
    in pop newStack2
```

```
ghci> stackManip [5,8,2,1]
(5,[8,2,1])
```

What we actually want

```
stackManip = do
    push 3
    a <- pop
    pop
```

State monad !

```
newtype State s a = State { runState :: s -> (a,s) }


instance Monad (State s) where
    return x      =
  (State h) >>= f =
```

State monad !

```
newtype State s a = State { runState :: s -> (a,s) }


instance Monad (State s) where
    return x = State $ \s -> (x,s)
    (State h) >>= f = State $ \s -> let (a, newState) = h s
                                        (State g) = f a
                                    in  g newState
```

Operations

```haskell
import Control.Monad.State

pop :: State Stack Int
pop = State $ \(x:xs) -> (x,xs)

push :: Int -> State Stack ()
push a = State $ \xs -> ((),a:xs)
```

Example

```haskell
import Control.Monad.State

stackManip :: State Stack Int
stackManip = do
    push 3
    a <- pop
    pop
```

Example

```haskell
import Control.Monad.State

stackManip :: State Stack Int
stackManip = do
    push 3
    a <- pop
    pop
```

```
ghci> runState stackManip [5,8,2,1]
(5,[8,2,1])
```

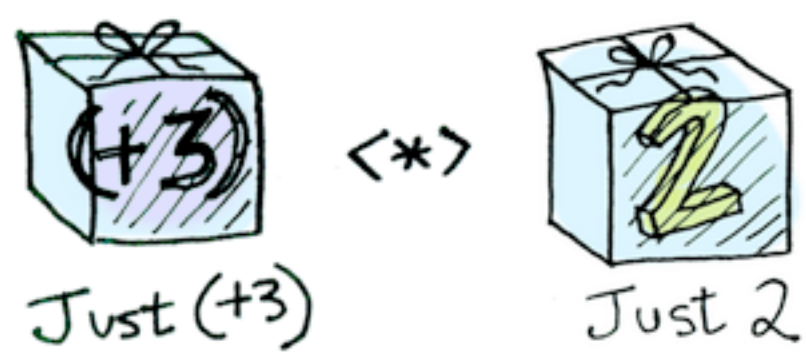
Get and Put

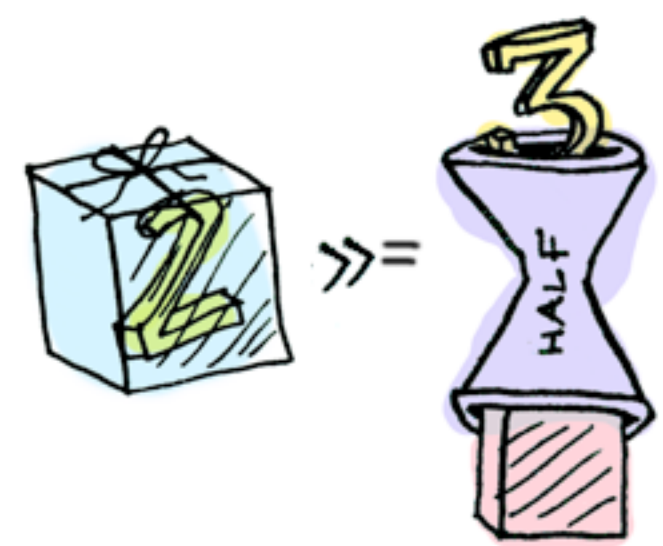
```haskell
get            = State $ \s -> (s,s)
put newState = State $ \s -> ((),newState)
```

Get and Put

```
stackyStack :: State Stack ()
stackyStack = do
    stackNow <- get
    if stackNow == [1,2,3]
        then put [8,3,1]
        else put [9,2,1]
```

Conclusion






```
Functor f
```

```
=> Applicative f
```

```
=> Monad f
```