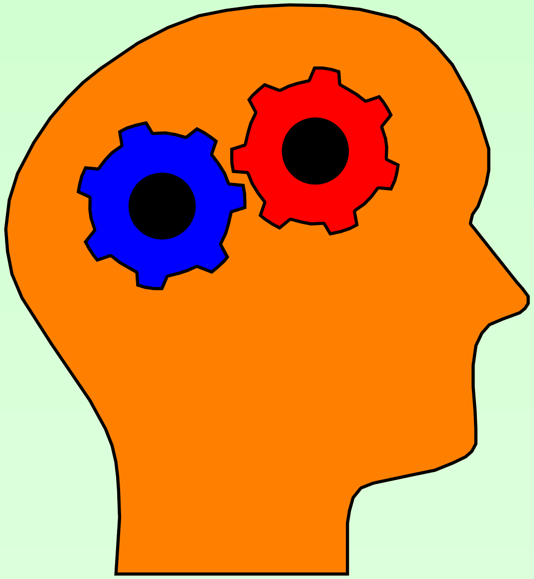
CS2104: Programming Languages Concepts

Lecture 6 : **Towards Monads**



"Imperative Programming in a Pure Language"

Lecturer : <u>Chin</u> Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Can be challenging but You are Not Alone

- The midnight Monad, a journey to enlightenment.

  https://www.lambdacat.com/the-midnight-monad-a-journey-to-enlightenment/

- Functors, Applicatives and Monads in Picture form:

  http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

- A Fistful of Monads.

  http://learnyouahaskell.com/a-fistful-of-monads

Pure vs Impure Code

- Imperative Programming (with side effects).

```
print :: String -> ()
```

```
let c = print("hello")
in c ; c
```

different

```
print("hello");
print("hello")
```

- Pure Monadic Programming.

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

```
let c = print("hello")
in c >> c
```

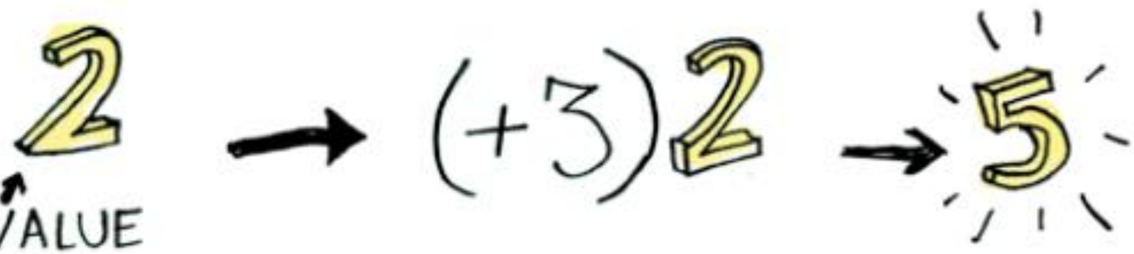
equivalent

```
print("hello") >>
print("hello")
```

Pure Value World

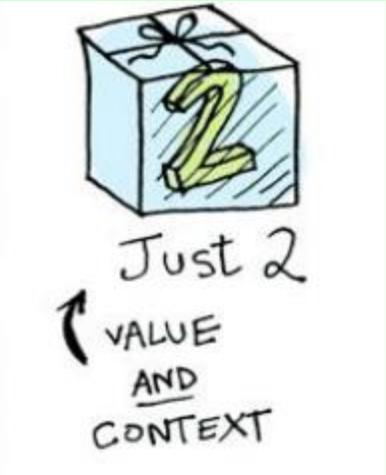
Here's a simple value:

And we know how to apply a function to this value:

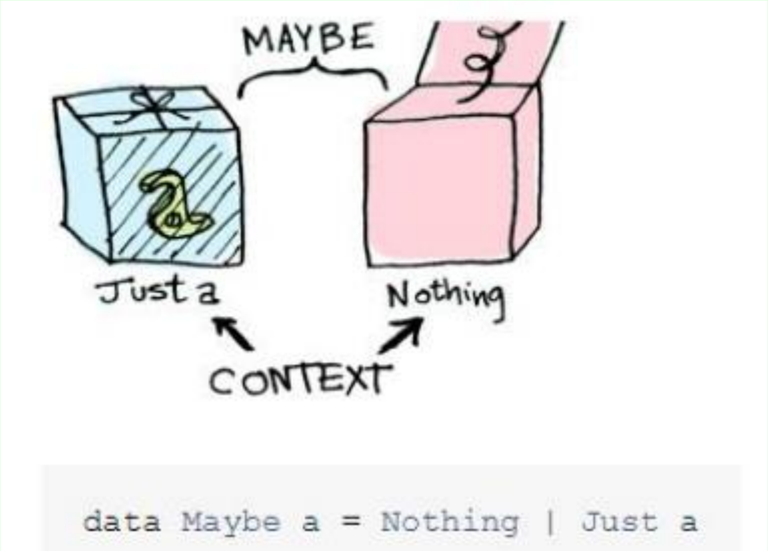


Value within Some Context

- Value and a Context.



- Maybe Type where
  where
  `Nothing`
  denotes error

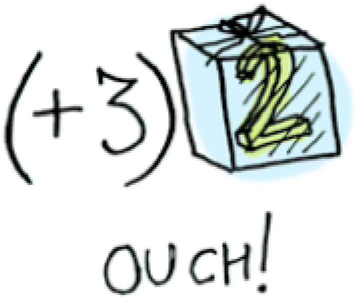


Other Examples of Context

- `[a]`
  - for non-determinism
- `state -> (state,a)`
  - for imperative state that can be updated
- `Parser a = String -> [(a,String)]`
  - For non-deterministic parsing
- `IO a`
  - for input-output interaction

Why Functor?

When a value is wrapped in a context, you can't apply a normal function to it:

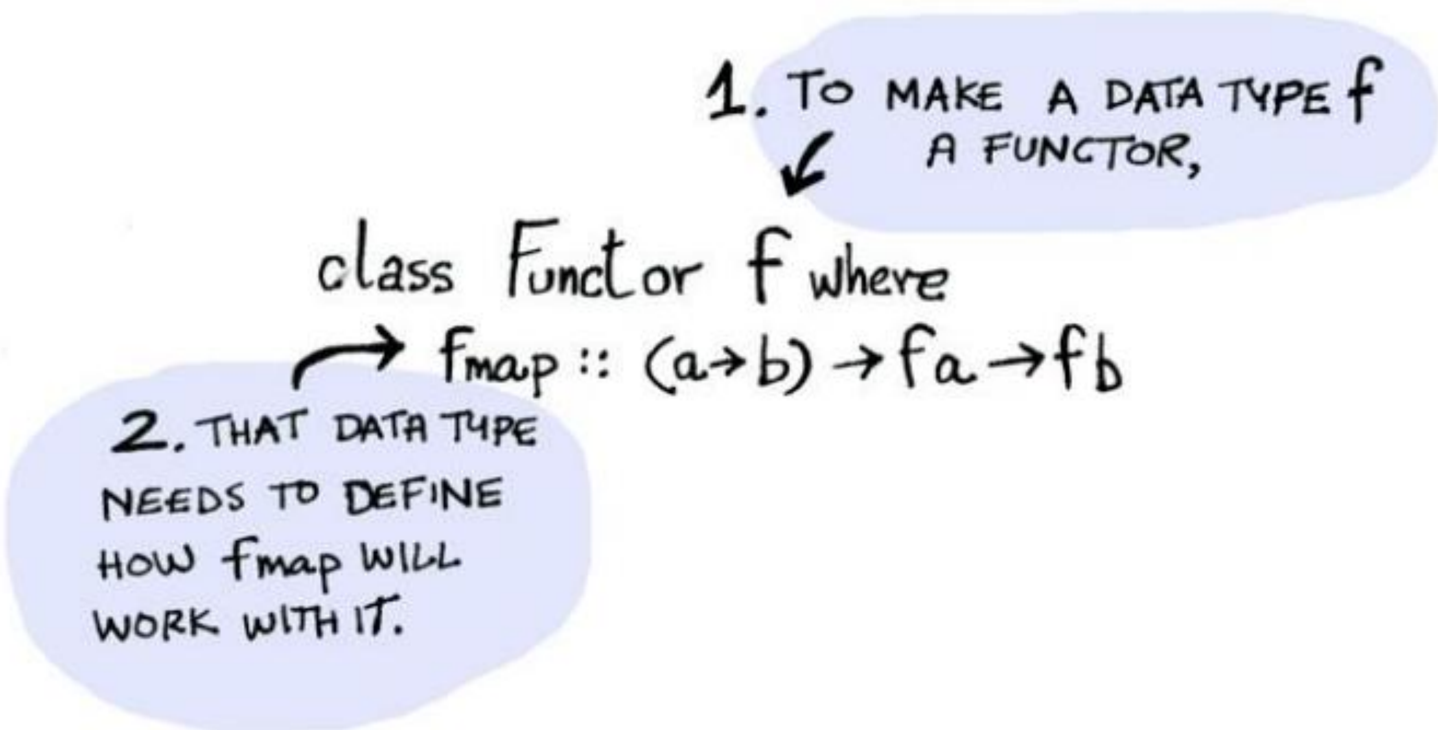


- Solution : Functor.

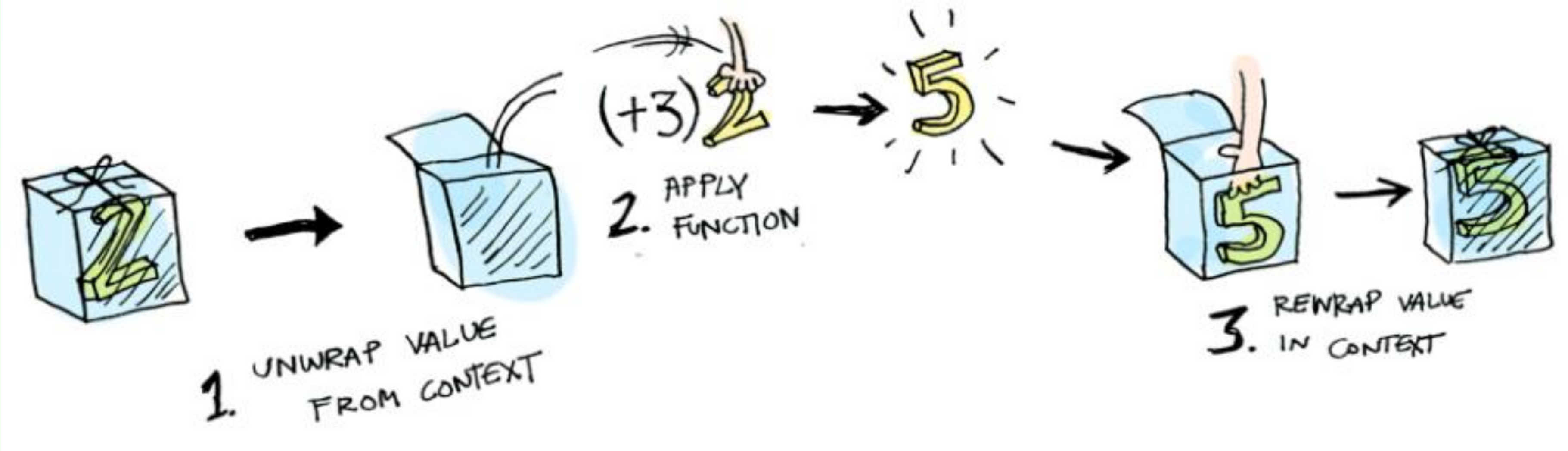
```
> fmap (+3) (Just 2)
Just 5
```

What is a Functor?

`Functor` is a typeclass. Here's the definition:

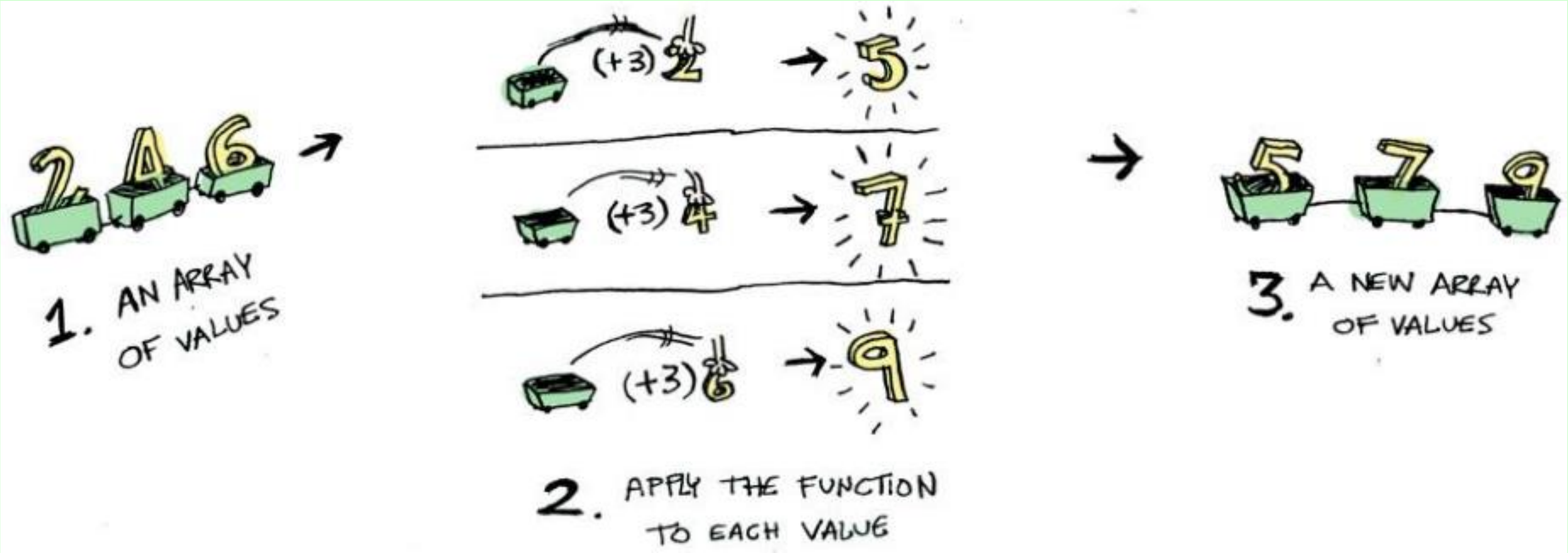


Behind the Scene




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

List/Arrays are also Functors




```
fmap   (+3)   [2,4,6]     ➔     [5,7,9]

(+3)    <$>   [2,4,6]     ➔     [5,7,9]
```

infix variant

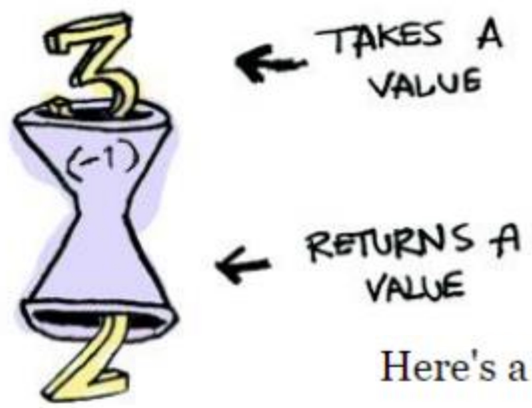
List as Functors

```haskell
instance Functor [] where
    fmap = map
```

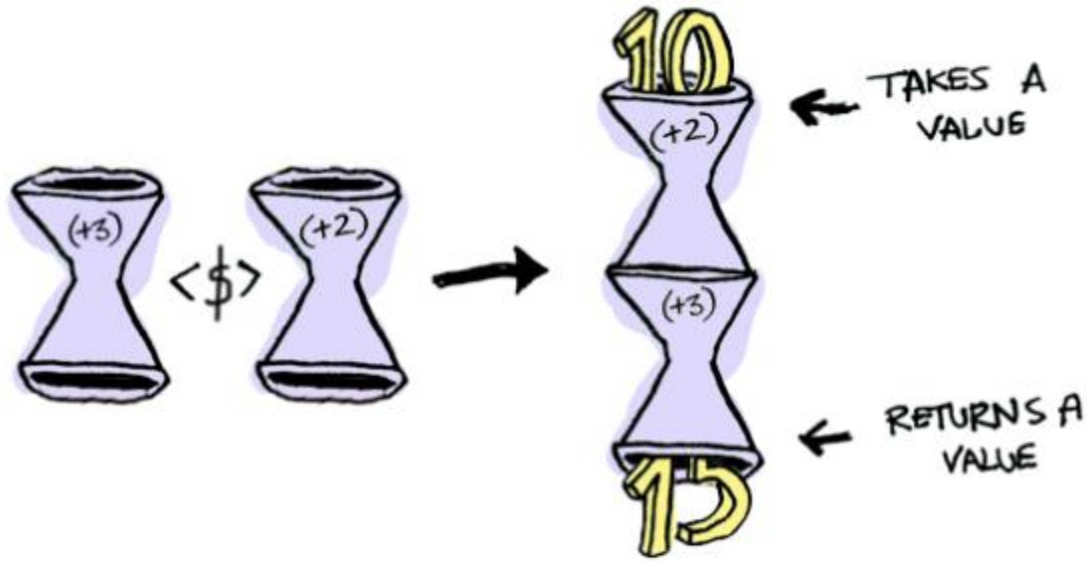
- List denotes non-determinism
- Examples:
  - `[]` means no solution
  - `[r1,r2,r3]` means three possible solutions

Functions are also Functors

Here's a function:



Here's a function applied to another function:



Functions as Functors ..

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

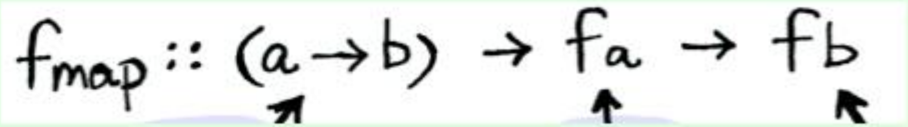
- Implementation

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

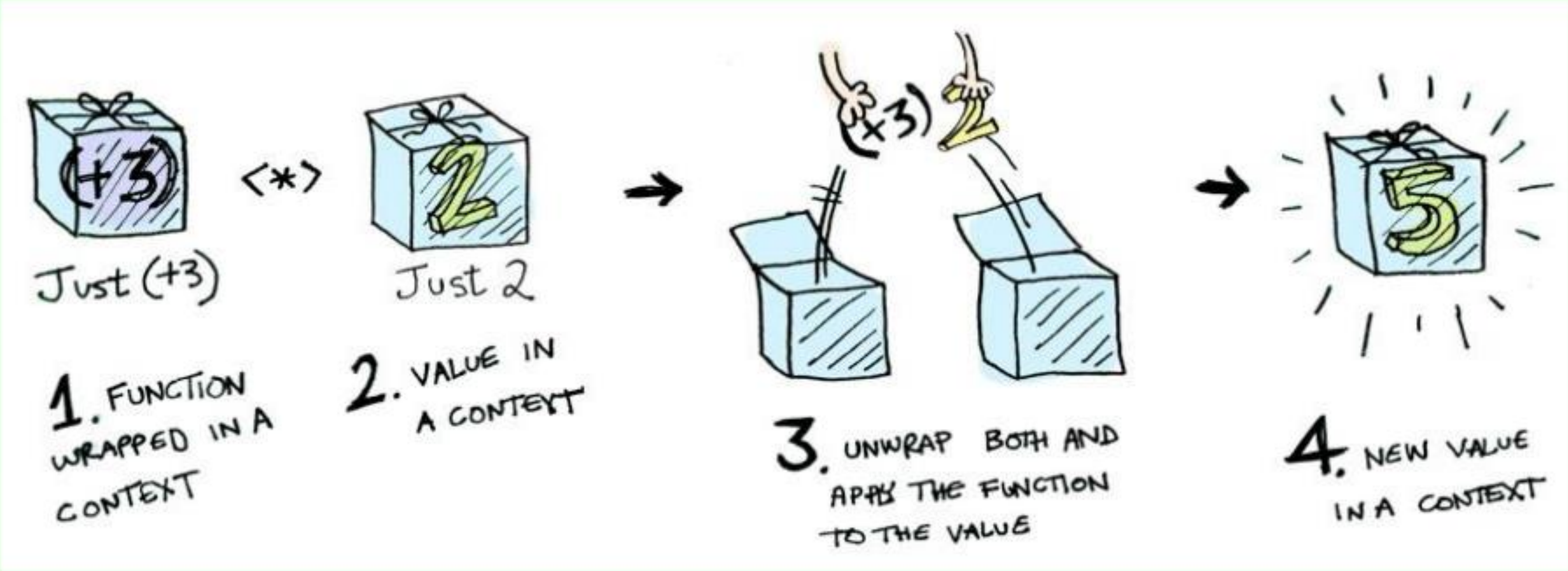
What IF Functions are Wrapped in Context?

`Maybe (Int -> Int)`　　　`Maybe (Int)`

Cannot use `fmap`

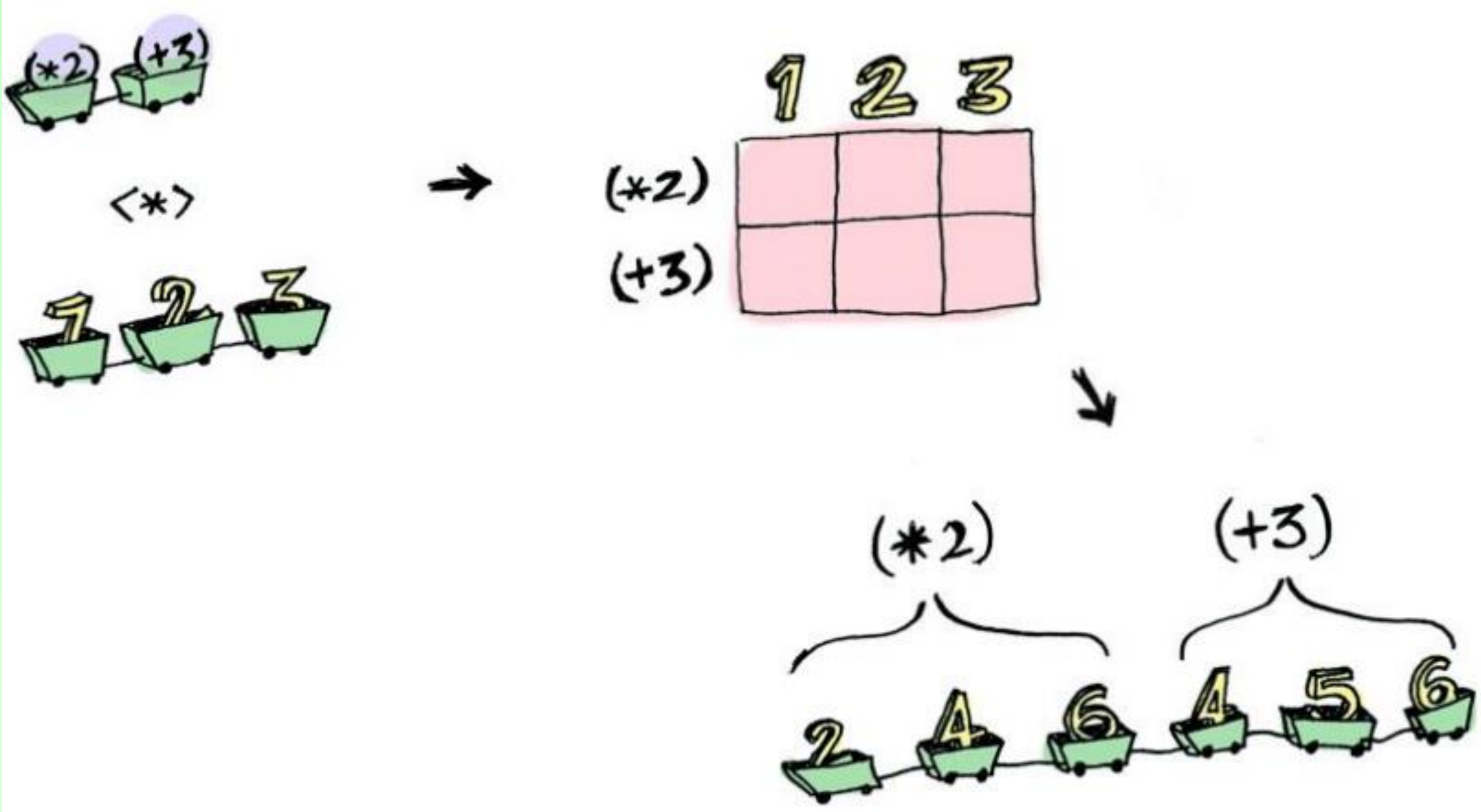


Applicative to the Rescue




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

Applicative in List Context




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

Why do we Need Applicative?

- Applicative can work with functions of any no. of arguments
  - Use fmap first

```
> let f = fmap (+) [1,2,3]
➢ :t f
➢ f :: Num a => [(a -> a)]
```

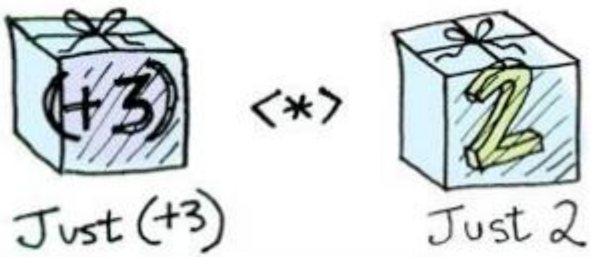
  - Use Applicative now

```
> f <*> [4,5]
➢ => [5,6,6,7,7,8]
```

Recap

Functors apply a function to a wrapped value:

Applicatives apply a wrapped function to a wrapped value:

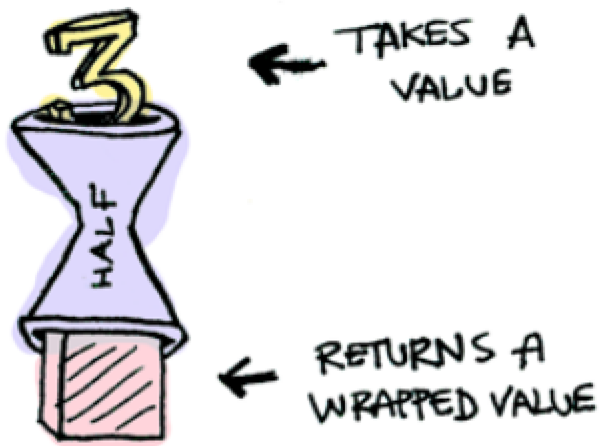


Essence of Monads

- How do we supply a wrapped value `(M a)`
  to a function which returns a wrapped value **`(a -> M b)`**
  `half :: Int -> Maybe Int`

Suppose `half` is a function that only works on even numbers:

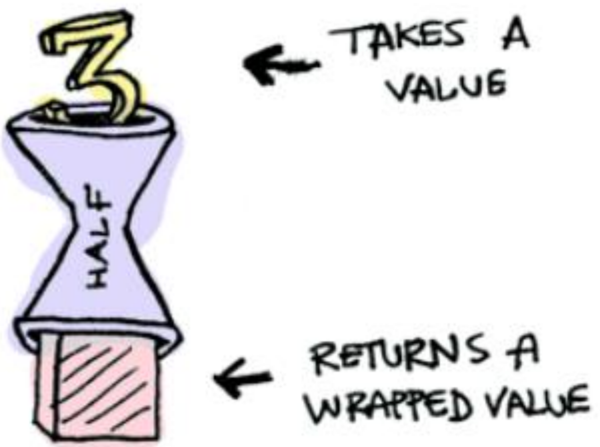
```
half x = if even x
           then Just (x `div` 2)
           else Nothing
```



What if we Apply on a Wrapped Value?

Suppose `half` is a function that only works on even numbers:

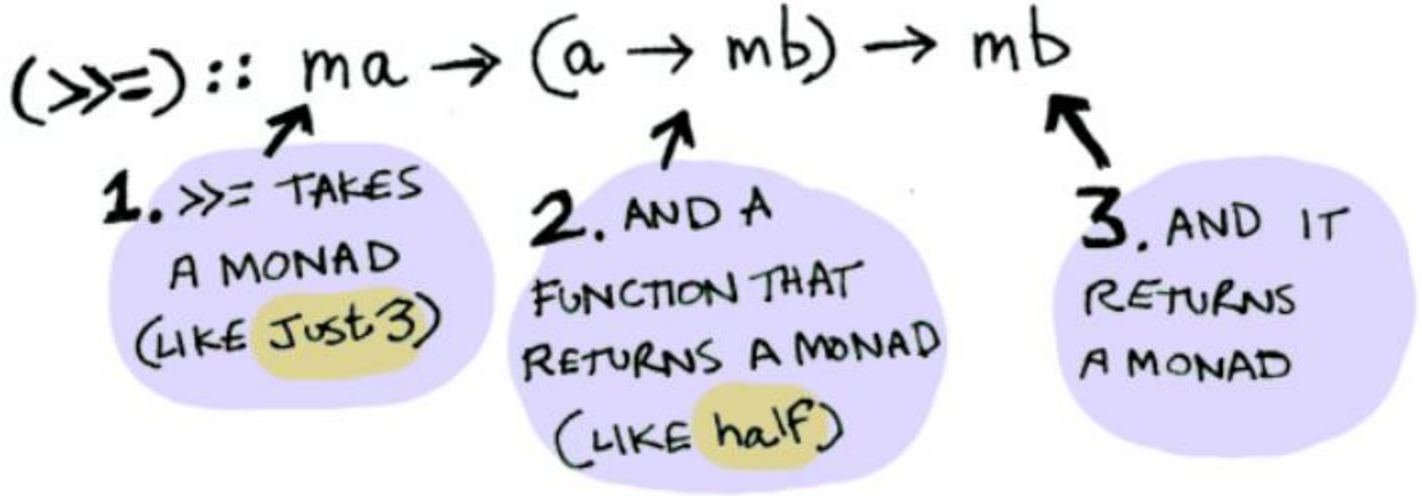
```
half x = if even x
           then Just (x `div` 2)
           else Nothing
```



Monad as a Type Class

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

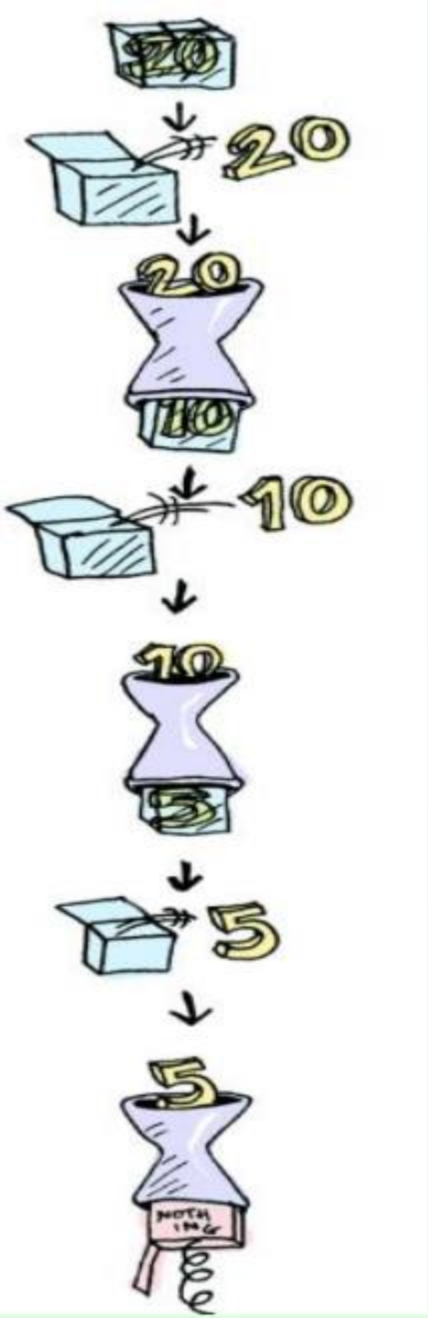
Where `>>=` is:



Chaining via Monads

```
> Just 20 >>= half >>= half >>= half
Nothing
```

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

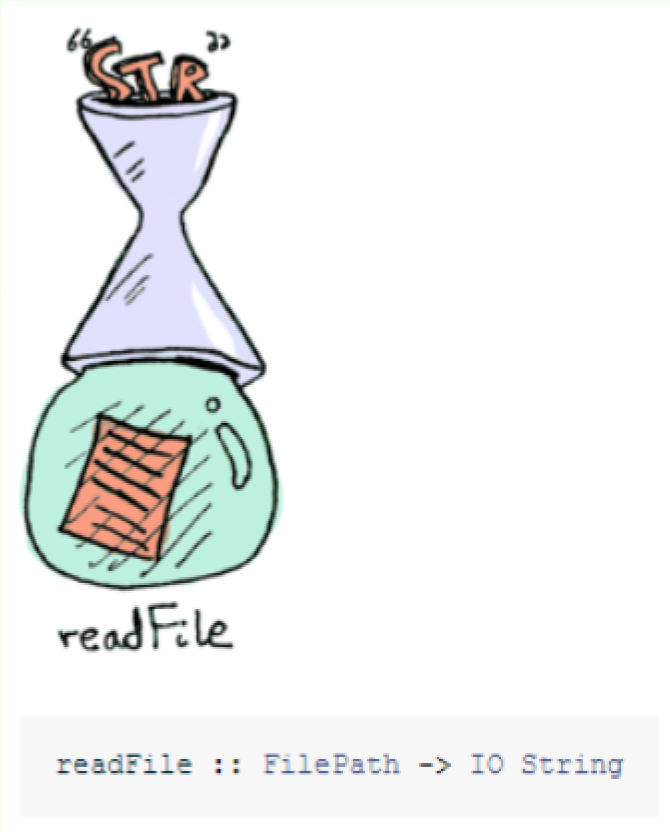


Input-Output as a Monad

`getLine` *takes no arguments and gets user input.*

IO Monad Operation

`readFile` *takes a string (a filename) and returns that file's contents*



IO Monad Operation

`putStrLn` takes a string and prints it:


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

Chaining IO Operation

Chaining

```
getLine >>= readFile >>= putStrLn
```

Syntactic Sugar

```
foo = do
    filename <- getLine
    contents <- readFile filename
    putStrLn contents
```

Do Comprehension

- Syntactic sugar notation for Monads.
- List is an instance of monad, and list comprehension is an instance of Do-comprehension!

```
[(x,y) | x <- xs, test x, y<-ys]

  do
    x <- xs
    filter test
    y <- xs
    return (a,b)

  filter test = \ x ->
      if test x then return a else empty
```

Monads

Monads Formally

- Another example of higher-order type class is:

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

  >> :: (m a) -> (m b) -> m b
  m1 >> m2    = m1 >>= (\ _ -> m2)
```

- Laws of `Monad` class:

```
(return a) >>= k     = k a
 m >>= return        = m
(m >>= (\a -> (k a) >>= (\b -> h b))
         = (m >>= (\a -> k a) >>= (\b -> h b)
```

- IO is an instance of Monad …

Input/Output

Input/Output

- The I/O system in Haskell is purely functional but has all the expressive power of conventional imperative languages.

- Actions are *defined* rather than *invoked* in an expression-oriented style.

- These actions are modelled as *monads* of type `IO t` which is a conceptual structure with some properties that supports imperative actions.

Basic I/O Operations

- Every I/O action returns a value, e.g :

```
getChar        :: IO Char
```

- Some IO actions also take input(s)

```
putChar        :: Char -> IO ()
```

- `IO` is an instance of the the `Monad` class.

- Actions are sequenced by bind operator:

```
(>>=)   :: Monad m => m a -> (a -> m b) -> m b
(>>)    :: Monad m => m a -> m b -> m b
```

Basic I/O Operations

- The do statement captures a sequence of actions, e.g :

```
main   :: IO ()
main   = do c <- getChar
            putChar c
```

- Syntactic sugar for the following:

```
main   = getChar >>=
           (\ c -> putChar c)
```

- How to return a value from sequence of actions?.

```
ready  = IO Bool
ready  = do c <- getChar
            c ==‘y’
```

```
return (c ==‘y’)
```

Bigger I/O Operations

- Function to get a string of char may use recursion, as follows:

```
getLine  :: IO String
getLine  = do c <- getChar
              if c=='\n' then return ""
              else do l <- getLine
                      return (c:l)
```

- A pure value can be converted into an action by return, but the not the converse. Illegal to use:

```
x + print y
```

- Function `f::Int -> Int -> Int` cannot do any IO at all, unless we make use of unsafe operations.

Building Actions

- IO operations are ordinary Haskell values that can be passed to functions, placed into data structures and returned as results etc.
- Example : we can build a list of actions.

```
todoList :: [IO ()]
todoList = [ putChar 'a',
             do {putChar 'b'; putChar 'c'},
             do {x <- getChar; putChar x} ]
```

- Can combine them into a single action using:

```
sequence_ :: [IO ()] -> IO ()
sequence_ = foldr (>>) (return ())
```

Imperative Programming

- I/O programming in Haskell is very close to that being done for ordinary imperative programming.

- As a comparison, imperative `getLine` is simply:

```
function getLine() {
     c := getChar();
     if c=='\n' then return ""
     else  {l:=getLine();
            return c:l} }
```

- Main difference is that no special semantics is needed and the entire code is still purely functional. Monad cleanly separates the pure from imperative.

Recap / Comparison

- Imperative `getLine` in C:

```
function getLine() {
     c := getChar();
     if c=='\n' then return ""
     else  {l:=getLine();
            return c:l} }
```

- Monadic IO in Haskell

```
getLine  :: IO String
getLine  = do c <- getChar
              if c=='\n' then return ""
              else do l <- getLine
                      return (c:l)
```