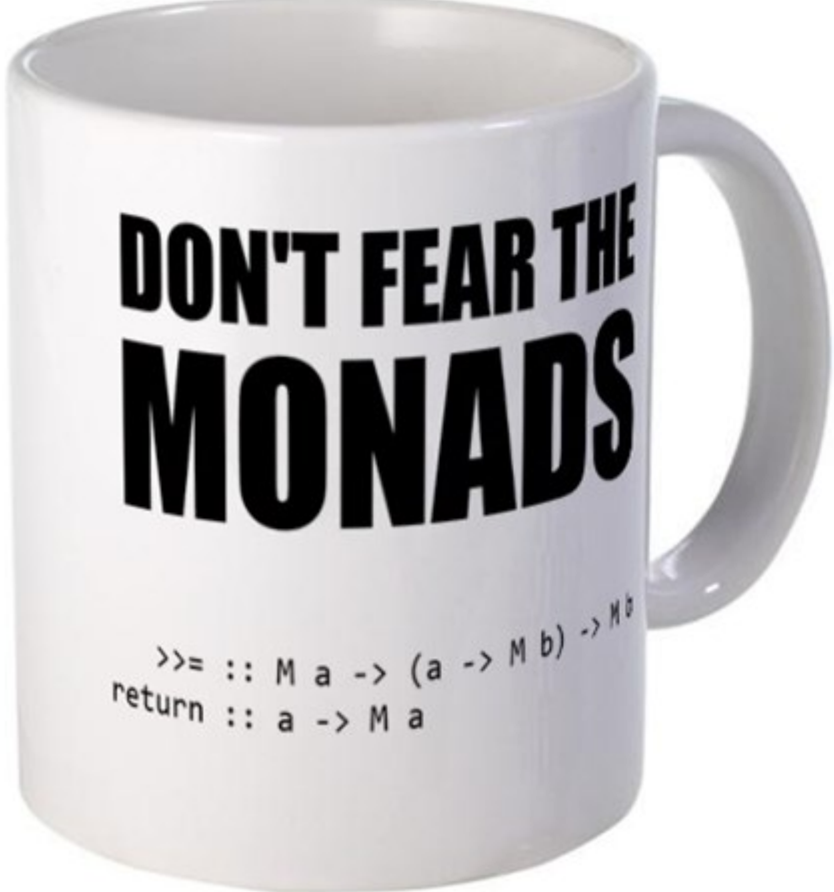


Monads

Bonus lecture
2017

David Sands

Our version of the story, so far….

Monad is the class of "instructions". Instructions can be built using "do" notation. We have seen two kinds of instructions i.e. two instances of Monad:

IO

```
do
 putStr "File name?"
 f <- getLine
 c <- readFile f
 return $ f ++ c
```

Gen

```
do
 n <- choose (2,10)
 r <- elements[Clubs,Spades]
 return $ Card r (Numeric n)
```

IO vs Gen

IO T

- Instructions to build a value of type **T** by interacting with the operating system
- Run by the ghc runtime system

Gen T

- Instructions to create a random value of type **T**
- Run by the QuickCheck library functions to perform random tests

Repeating Instructions

```
doTwice i =
  do a <- i
     b <- i
     return (a,b)
```

```
Main> doTwice $ putStrLn "hello”
hello
hello
((),())
Main>
```

Repeating Instructions

```
doTwice i =
  do a <- i
     b <- i
     return (a,b)
```

```
Main> sample $ doTwice (choose ('a','z'))
('m','c')
('b','j')
('h','l')
('y','q')
('k','f')
('w','q')
('p','h')
Main>
```

Monads = Instructions

- What is the type of doTwice?

```
Main> :t doTwice
doTwice :: Monad a => a b -> a (b,b)
```

Even the *kind of instructions* can vary.

Different kinds of instructions, depending on who obeys them.

Whatever kind of result it produces, we get a pair of them

IO means operating system.

Plan

1. One more example of a Monad:
   – Instructions for Parsing (a parsing library)

2. Rolling your own Monads
   – The Truth about "do"
   – The Parser Monad
   – Maybe is also a Monad (and list, and...

A Simple Parsing Library

A library for building parsers containing:

- An abstract data type `Parser a`
- A function

```
parse ::
 Parser a -> String -> Maybe (a,String)
```

- Basic building blocks for building parsers

Example: Phone numbers

Two ways of writing phone numbers:

+46 317721000   0317721000

```
data PhoneNr = Global Int Int | Local Int
  deriving Show

showNr (Global cc n) = "+" ++ show cc ++ " " ++ show n
showNr (Local n)     = show n

work = Local 7372076
home = Global 46 73720766
```

IO

```
do
  s <- getLine
  c <- readFile s
  return $ s ++ c
```

Gen

```
do
  n <- elements[1..9]
  m <- vectorOf n arbitrary
  return $ n:m
```

Parser

```
do
  c <- sat (`elem` ”;,:”)
  ds <- chain digit (char c)
  return $ map digitToInt ds
```

IO t	**Gen t**	**Parser t**
• Instructions for interacting with operating system • Run by GHC runtime system produce value of type t	• Instructions for building random values • Run by **quickCheck** to generate random values of type t	• Instructions for parsing • Run by **parse** to parse a string and produce a **Maybe** (t,String)

Terminology

- A *"monadic value"* is just an expression whose type is an instance of class Monad
- *"t is a monad"* means t is an instance of the class Monad
- We have often called a monadic value an *"instruction"*. This is not standard terminology
  - but sometimes they are called "actions"

Monads and do notation

- To be an instance of class Monad you need (as a minimal definition) operations **>>=** and **return**

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

  (>>) :: m a -> m b -> m b
  x >> y = x >>= \_ -> y

  fail :: String -> m a
  fail msg = error msg
```

Default implementations

Update, As of GHC 7.10

OK that's a bit old school. Nowadays Monad is a subclass of Applicative (which is a subclass of Functor)

The class itself is a bit simpler – you just need to define >>=

But I'll define it the "old" way and ignore the rest

Boilerplate to make your monad an instance of Applicative

```
import Control.Applicative (Applicative(..))
import Control.Monad(liftM, ap)

instance Functor MyMonad where fmap = liftM
instance Applicative MyMonad where
    pure  = return
    (<*>) = ap
```

See ”Learn you a Haskell…” for more info on Functor and Applicative

The truth about Do

- Do syntax is just a shorthand:

```
do act1
   act2
```

==

```
act1 >> act2
```

==

```
act1 >>= \_ -> act2
```

```
do v <- act1
   act2
```

==

```
act1 >>= \v -> act2
```

The Parser Monad

- To be an instance of class Monad you need two operations: **>>=** and **return**

```
instance Monad Parser where
  return = succeed
  (>>=)  = (>*>)
```

- Why bother?

Our first example of a home-grown monad
Can understand do notation

Example

```haskell
foo :: IO ()
foo = do
    filename <- getLine
    contents <- readFile filename
    putStrLn contents
```

The truth about Do

Full translation (I)

```
do act1
   …
   actn
```

==

```
act1 >> do …
           actn
```

```
do v <- act1
   …
   actn
```

==

```
act1 >>= \v -> do …
                  actn
```

```
do actn
```

==

```
actn
```

The truth about Do

Full Translation (II): Let and pattern matching

```
do let p = e
   …
   actn
```

==

```
let p = e in
do …
   actn
```

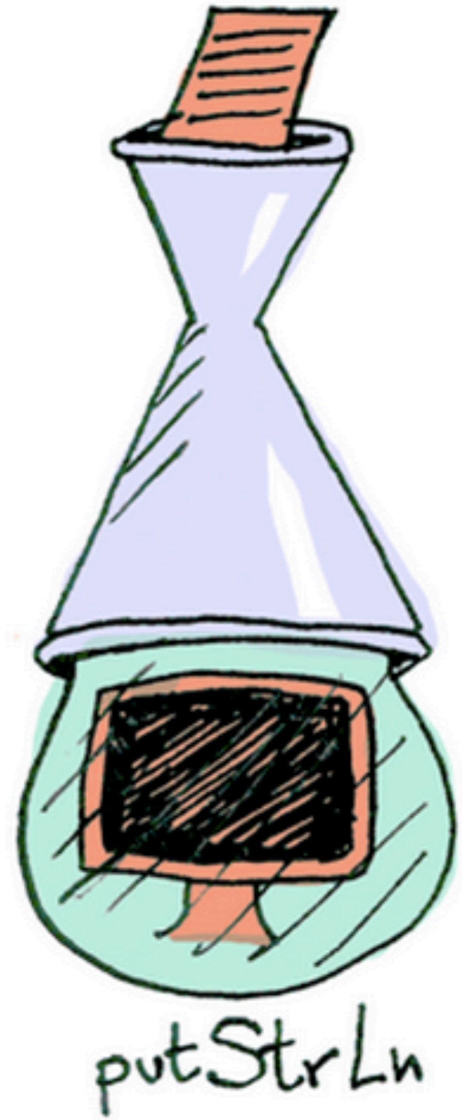
```
do pattern <- act1
   …
   actn
```

==

```
let f pattern = do …
                   actn
    f _       = fail “Error”
in  act1 >>= f
```


```
getLine :: IO String
```


```
readFile :: FilePath -> IO String
```

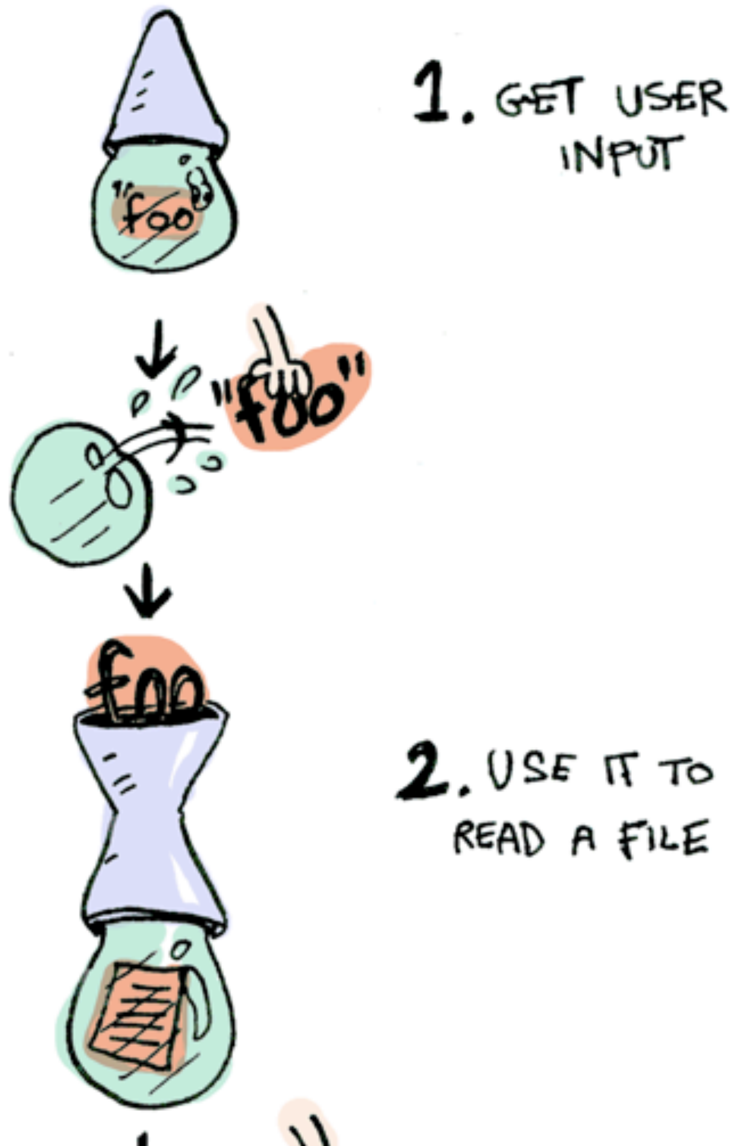



```haskell
putStrLn :: String -> IO ()
```

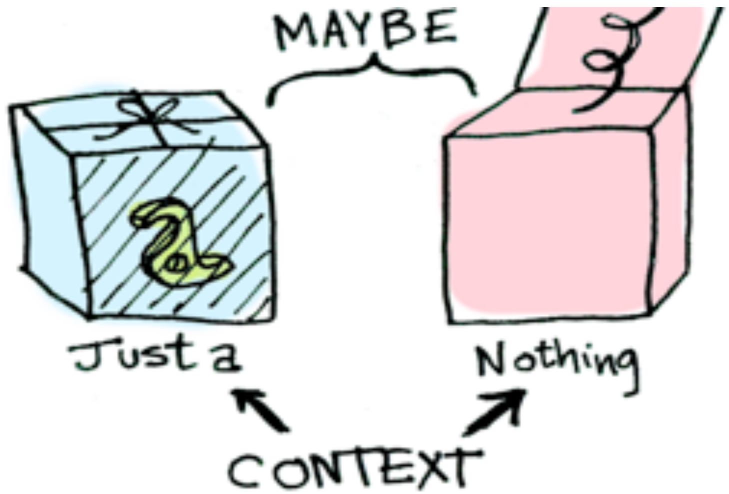
All three functions take a value (or no value) and produce an IO “wrapped” value

The function >>= allows us to join them together

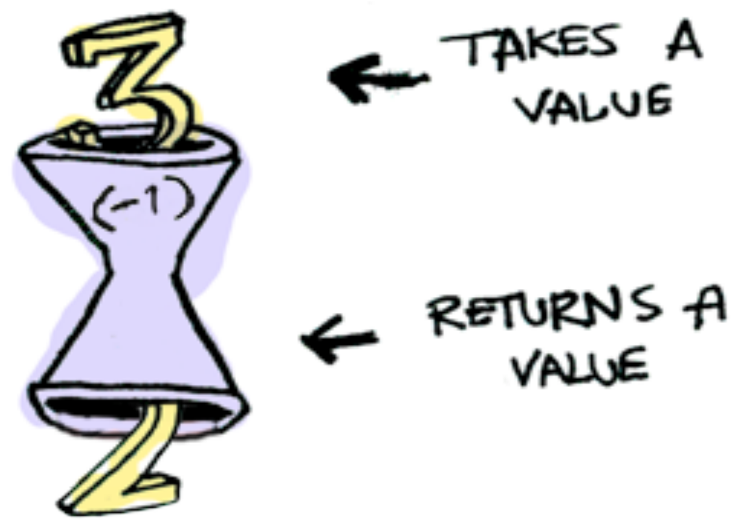
getLine >>= readFile >>= putStrLn


"foo"
"foo"
foo
1. GET USER INPUT
2. USE IT TO READ A FILE

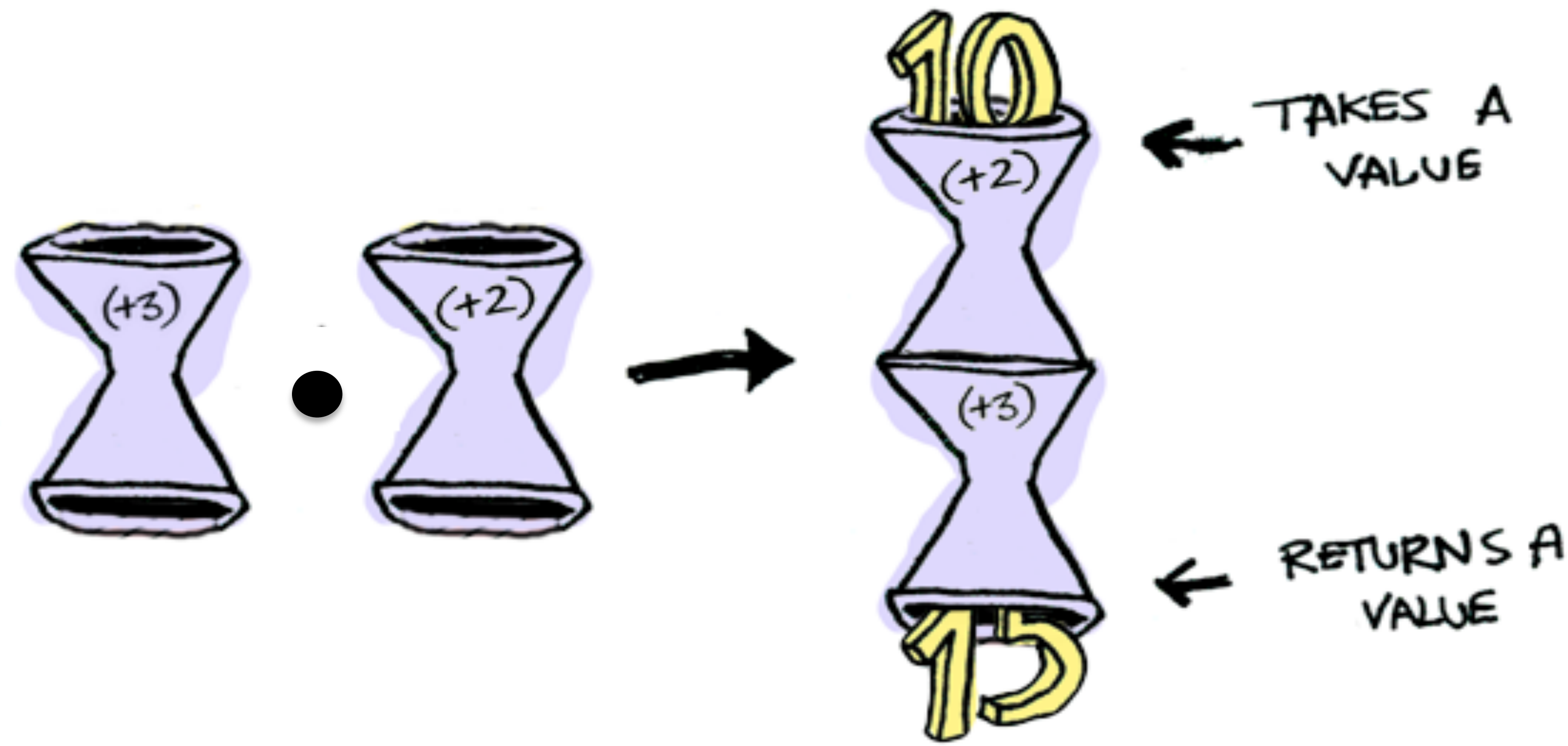
Maybe



Here is a function

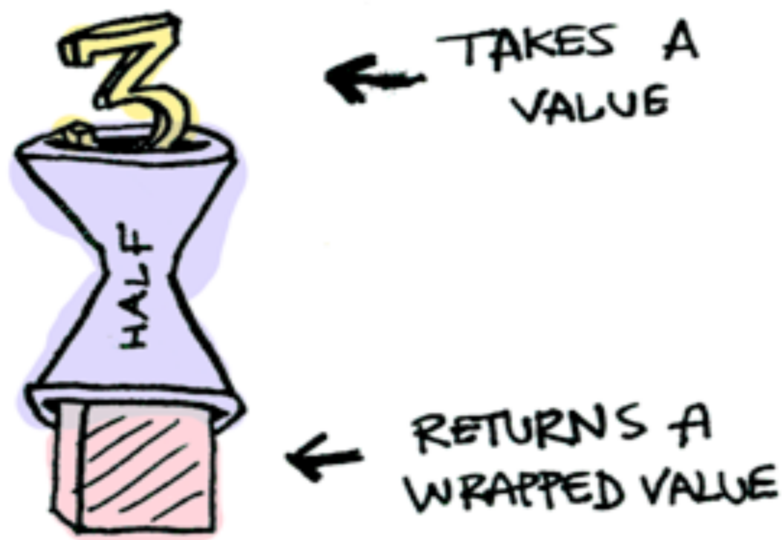


They can be composed



Here is a function

```
half x
    | even x = Just (x `div` 2)
    | odd x  = Nothing
```



What if we feed it a wrapped value?

We need to use >>= to shove our wrapped value into the function

>>=

>>=

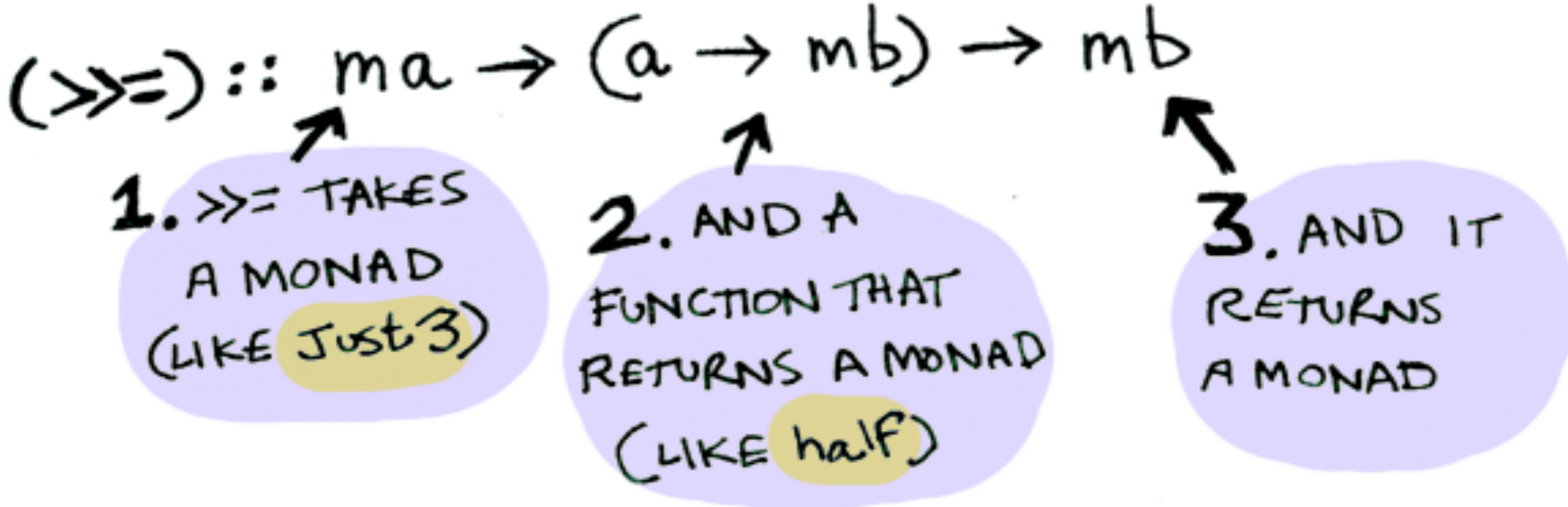
Here's how it works:

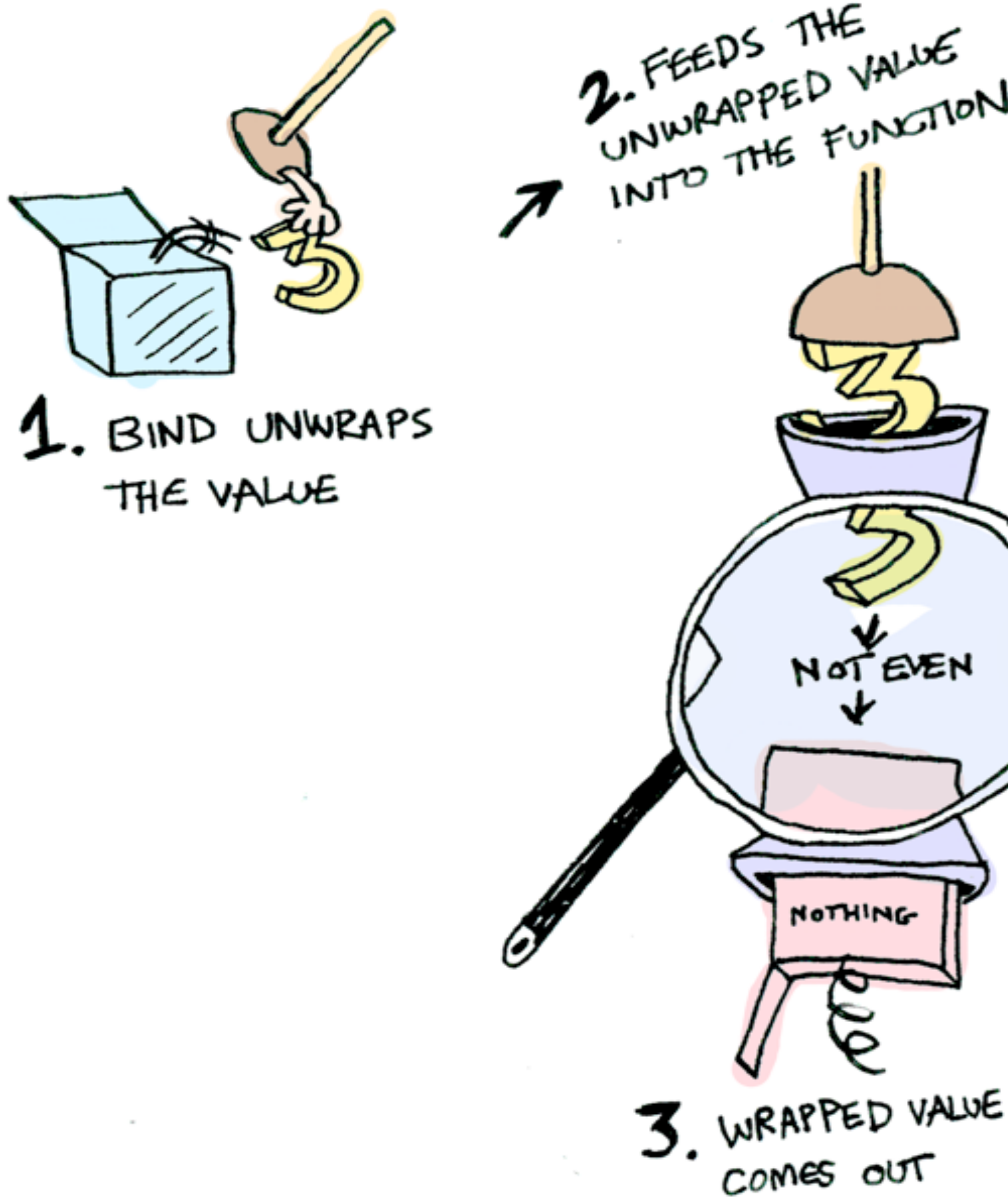
```
> Just 3 >>= half
Nothing
> Just 4 >>= half
Just 2
> Nothing >>= half
Nothing
```

What's happening inside? `Monad` is another typeclass. Here's a partial definition:

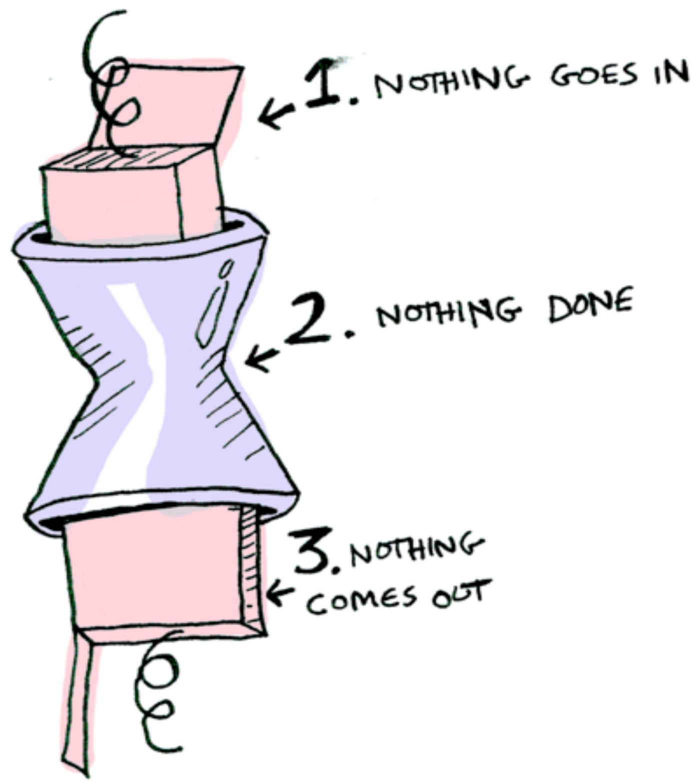
```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
```

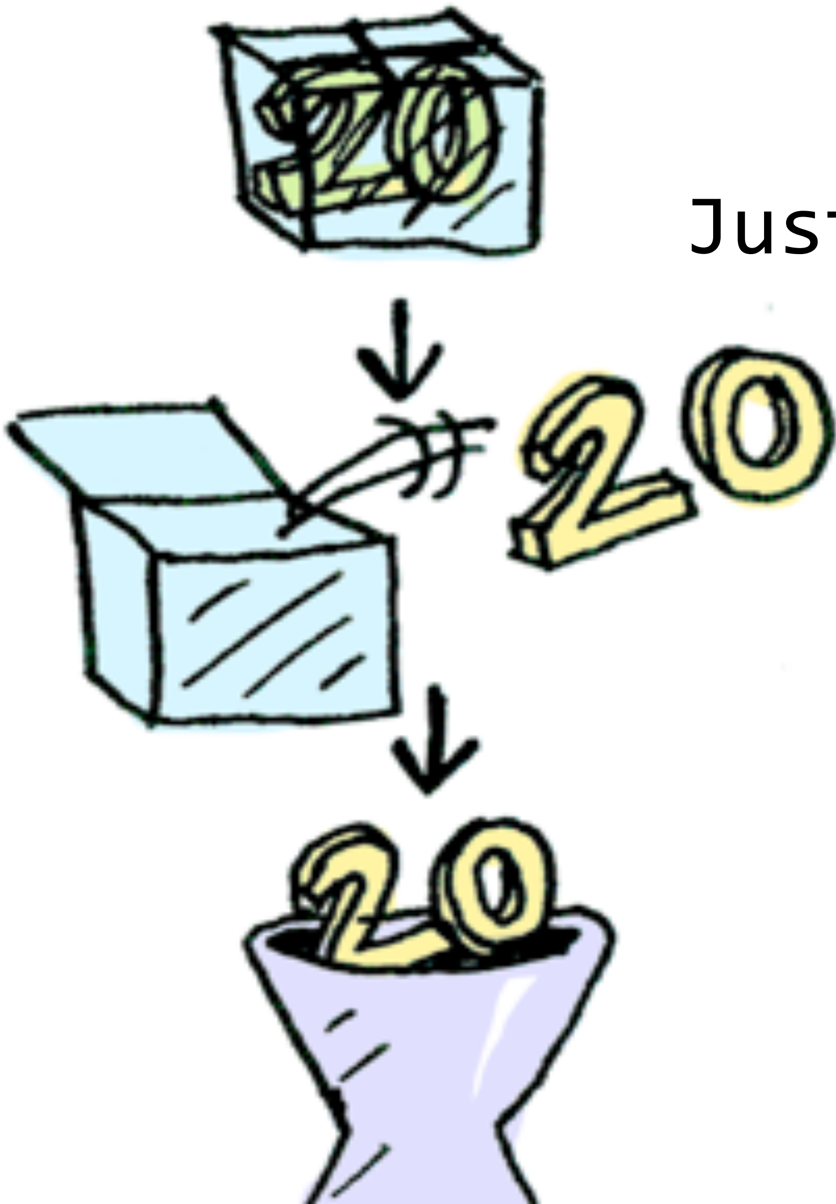
>>=


(>>=) :: 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

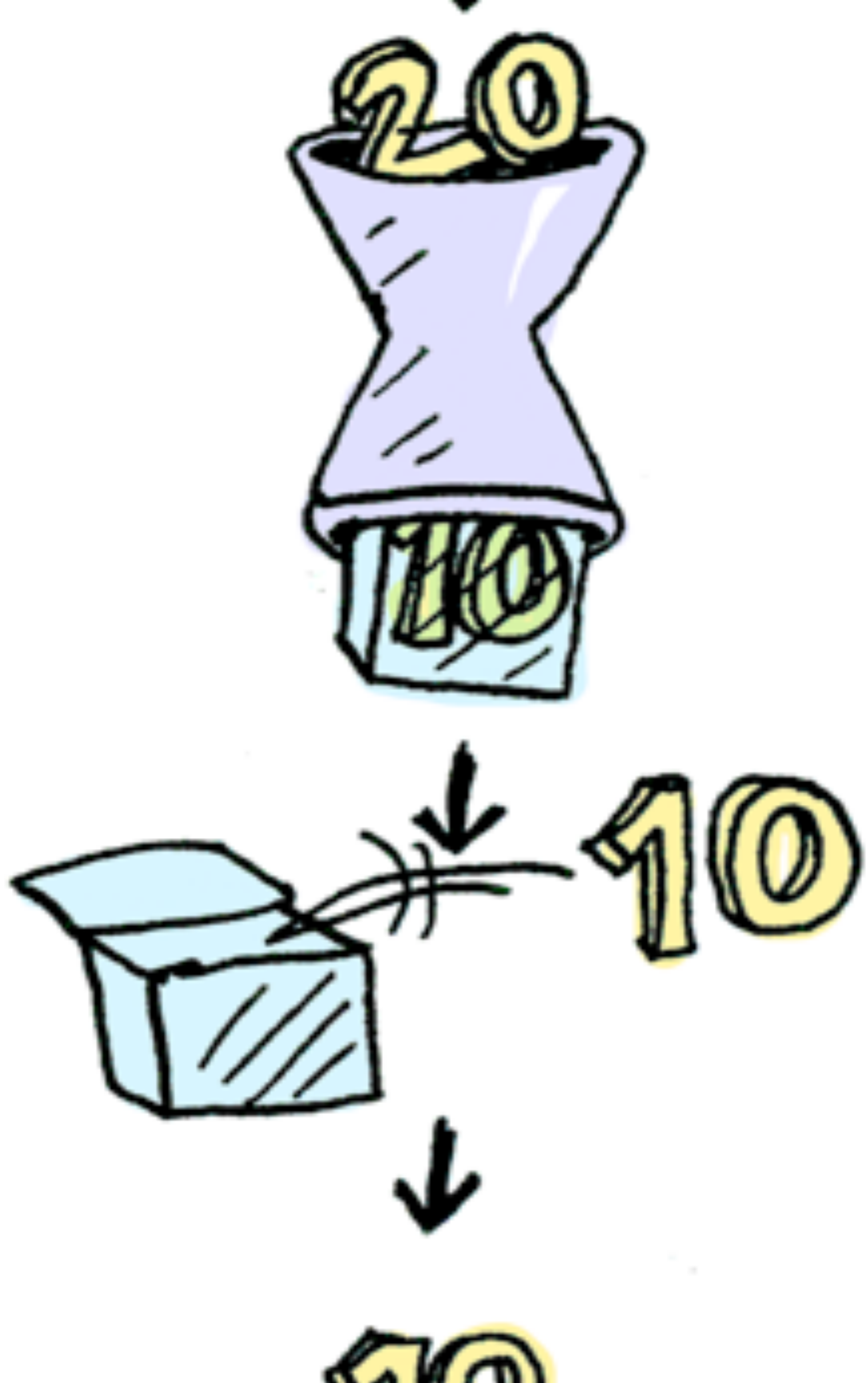

1. BIND UNWRAPS THE VALUE
2. FEEDS THE UNWRAPPED VALUE INTO THE FUNCTION
NOT EVEN
NOTHING
3. WRAPPED VALUE COMES OUT

>>=


1. NOTHING GOES IN
2. NOTHING DONE
3. NOTHING COMES OUT



```
Just 20 >>= half >>= half
        >>= half
```


20
10
10


10
5
5

Instance Monad Maybe

- Maybe is a very simple monad

```
instance Monad Maybe where
   Just x  >>= k = k x
   Nothing >>= _ = Nothing

   return        = Just
   fail s        = Nothing
```

Although simple it can be useful…

Congestion Charge Billing

Congestion Charge Billing

Registration number used to find the Personnummer of the owner

```
carRegister :: [(RegNr,PNr)]
```

Personnummer used to find the name of the owner

```
nameRegister :: [(PNr,Name)]
```

Name used to find the address of the owner

```
addressRegister :: [(Name,Address)]
```

Example:
Congestion Charge Billing

```
type CarReg = String ; type PNr = String
type Name = String ; type Address = String

carRegister :: [(CarReg,PNr)]
carRegister
 = [("JBD 007","750408-0909"), ...]

nameRegister :: [(PNr,Name)]
nameRegister
 = [("750408-0909","Dave“), ... ]

addressRegister :: [((Name,PNr),Address)]
addressRegister =
  [(("Dave","750408-0909"),"42 Streetgatan\n Askim")
  , ... ]
```

Example: Congestion Charge Billing

With the help of

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```

we can return the address of car owners

```
billingAddress :: CarReg -> Maybe (Name, Address)
billingAddress car =
 case lookup car carRegister of
   Nothing -> Nothing
   Just pnr -> case lookup pnr nameRegister of
        Nothing -> Nothing
        Just name ->
           case lookup (name,pnr) addressRegister of
                Nothing -> Nothing
                Just addr -> Just (name,addr)
```

Example: Congestion Charge Billing

Using the fact that Maybe is a member of class Monad we can avoid the spaghetti and write:

```
billingAddress car = do
  pnr  <- lookup car carRegister
  name <- lookup pnr nameRegister
  addr <- lookup (name,pnr) addressRegister
  return (name,addr)
```

Example: Congestion Charge Billing

Unrolling one layer of the do syntactic sugar:

```
billingAddress car ==
 lookup car carRegister >>= \pnr ->
 do
  name <- lookup pnr nameRegister
  addr <- lookup (name,pnr) addressRegister
  return (name,addr)
```

- `lookup car carRegister` gives `Nothing` then the definition of `>>=` ensures that the whole result is `Nothing`
- `return` is `Just`

Summary

- We can use higher-order functions to build Parsers from other more basic Parsers.
- Parsers can be viewed as an instance of Monad
- We have seen how we can build our own Monads!
  - A lot of ”plumbing” is nicely hidden away
  - The implementation of the Monad is not visible and can thus be changed or extended

IO t

- Instructions for interacting with operating system
- Run by GHC runtime system produce value of type t

Gen t

- Instructions for building random values
- Run by **quickCheck** to generate random values of type t

Parser t

- Instructions for parsing
- Run by **parse** to parse a string and **Maybe** produce a value of type t

+ Maybe = Four Monads

Code

- Parsing.hs
  - module containing the parser monad and simple parser combinators.

See course home page

- We can build our own Monads!
  - A lot of ”plumbing” is nicely hidden away
  - A powerful pattern, used widely in Haskell
  - A pattern that can be used in other languages, but syntax support helps
    - F# computation expressions
    - Scala

More examples

- http://adit.io/posts/2013-06-10-three-useful-monads.html
- stack (slides/video from last year)

Another Example: A Stack

- A Stack is a stateful object
- Stack operations can push values on, pop values off, add the top elements

```
type Stack = [Int]
newtype StackOp t = StackOp (Stack -> (t,Stack))

-- the type of a stack operation that produces
-- a value of type t
pop :: StackOp Int
push :: Int -> StackOp ()
add :: StackOp ()
```

Running a StackOp

```
type Stack = [Int]
newtype StackOp t = StackOp (Stack -> (t,Stack))

run (StackOp f) = f

-- run (StackOp f) state = f state
```

Operations

```haskell
pop :: StackOp Int
pop = StackOp $ \(x:xs) -> (x,xs) -- can fail

push :: Int -> StackOp ()
push i = StackOp $ \s -> ((),i:s)

add :: StackOp ()
add = StackOp $ \(x:y:xs) -> ((),x+y:xs) -- can fail
```

Building a new StackOp…

```
swap :: StackOp ()
swap = StackOp $ \s ->
          let (x,s') = run pop s
              (y,s'') = run pop s'
              (_,s''') = run (push x) s''
              (_,s'''') = run (push y) s'''
           in (_, s'''')
```

No thanks!

StackOp is a Monad

- Stack instructions for producing a value

```
-- (>>=) :: StackOp a -> (a -> StackOp b) -> StackOp b
instance Monad StackOp
  where return n = StackOp $ \s -> (n,s)
        sop >>= f  = StackOp $ \s ->
                             let (i,s') = run sop s
                             in run (f i) s'
```

So now we can write…

```
swap = do
    a <- pop
    b <- pop
    push a
    push b
```

Stack t

- Stack instructions producing a value of type t
- Run by **run**

Maybe t

- Instructions for either producing a value or nothing
- Run by ?? (not an abstract data type)

Two More Monads

Pictures from a blog post about functors, applicatives and monads

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

Aditya Y. Bhargava