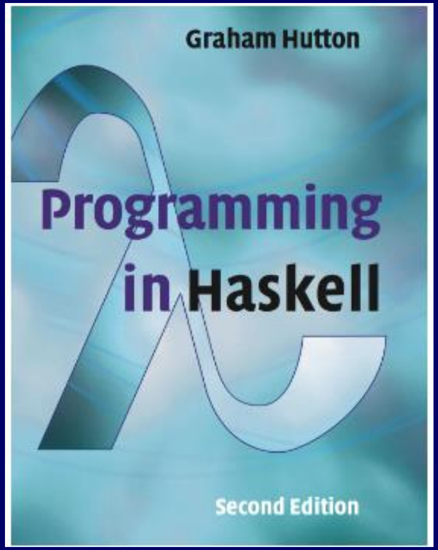
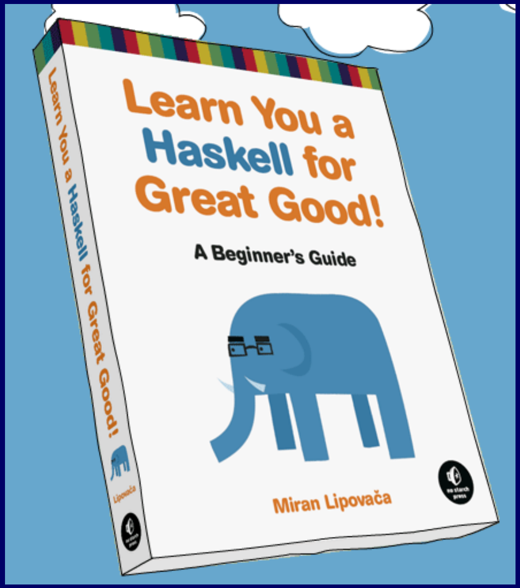
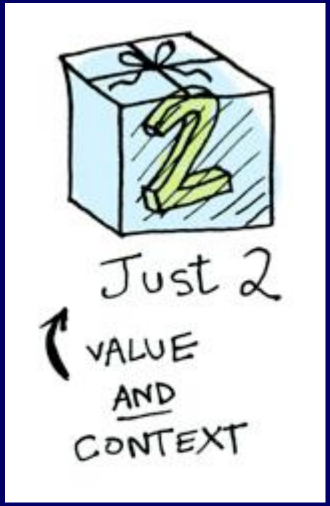
PROGRAMMING IN HASKELL

Monads



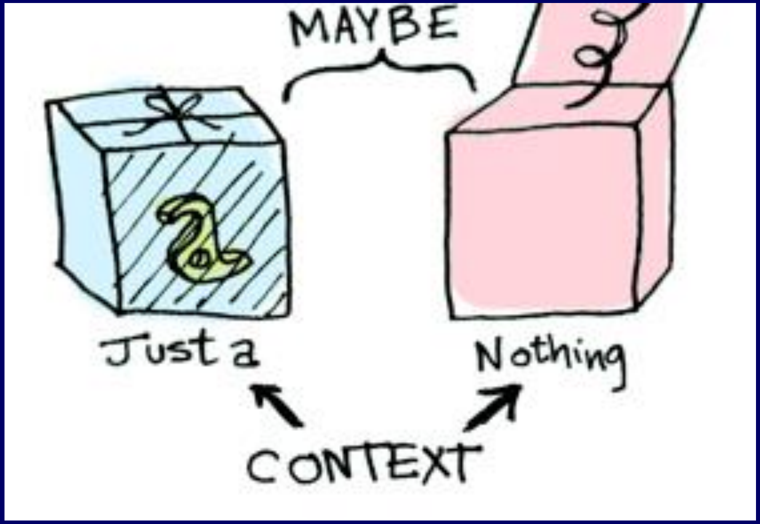




Based on lecture notes by Graham Hutton,
the book "Learn You a Haskell for Great Good",
pictures from Aditya Bhargava

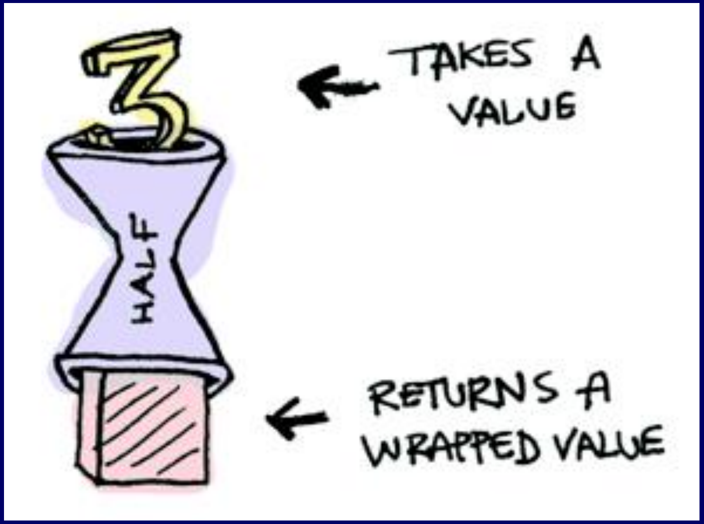
Monads

We look at Maybe to illustrate



Monads

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




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

Monads

What if we feed it a wrapped value?

We need to use >>= to shove our wrapped value into the function. Here's a photo of >>=:

Monads

How does it work?

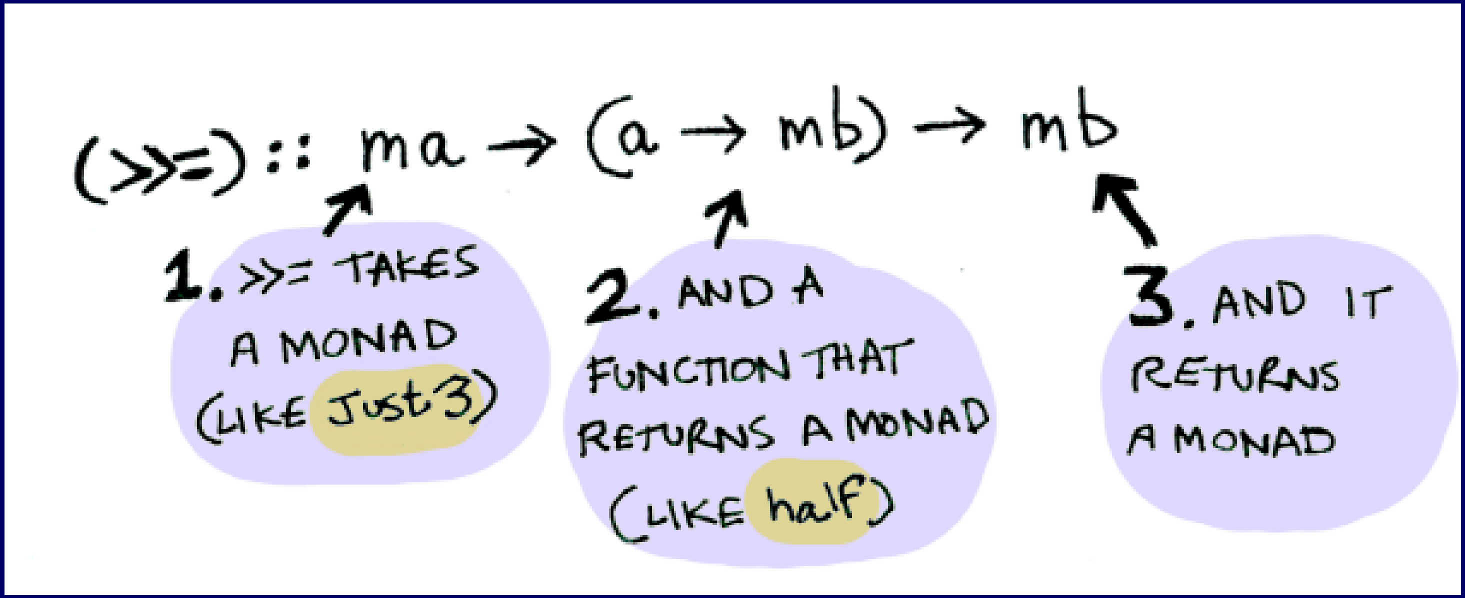
```
> Just 3 >>= half Nothing
> Just 4 >>= half Just 2
> Nothing >>= half Nothing
```

Monads

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

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

Where >>= is:



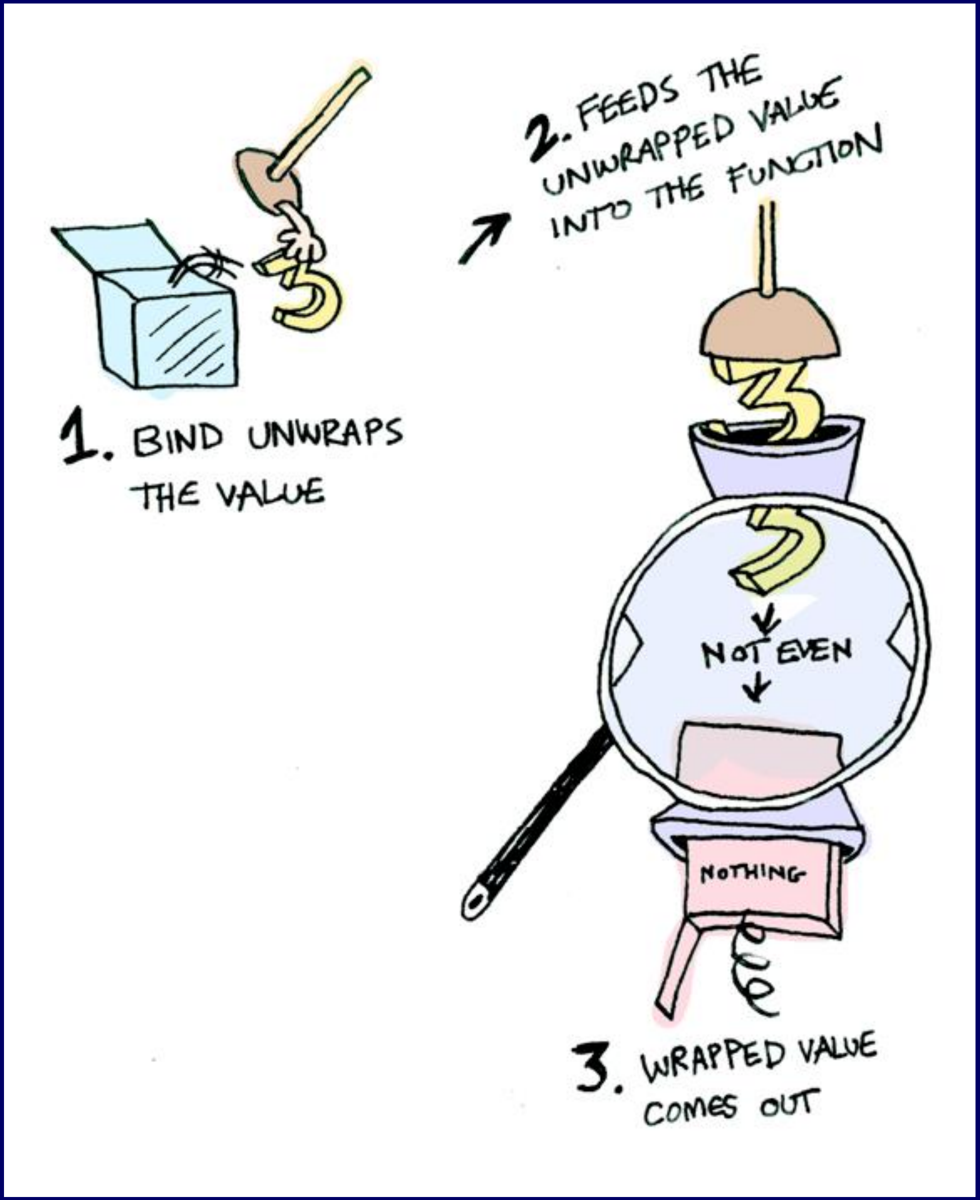
Monads

So Maybe is a Monad:

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

Monads

Here it is
in action
with
a Just 3!



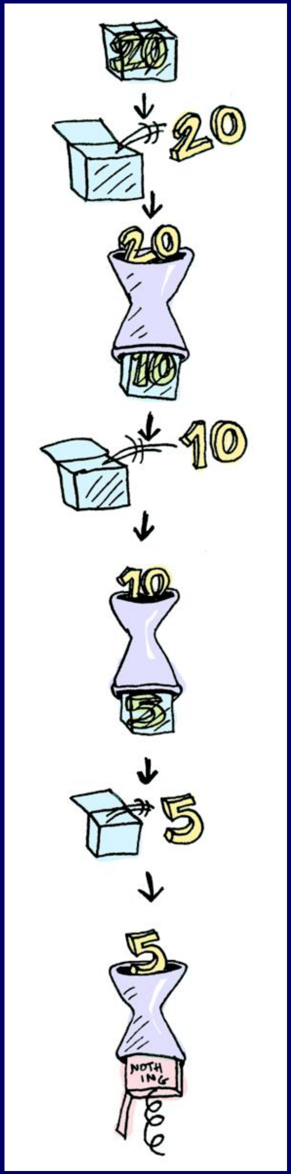
Monads

And if you pass in a Nothing it's even simpler:

Monads

You can also chain calls:

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



What is a Monad? – Computerphile
Graham Hutton
Computer Science at the University of Nottingham

Monads

A monad is defined by 3 things:

- A type constructor m
- A function return
- An operator (>>=) – read as “bind”

Originally, these were introduced for IO, but they can be much more powerful (and difficult to understand).

The function and operator are methods of the Monad typeclass and have types themselves, which are required to obey 3 laws (more later):

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

Monads example: Maybe

```
return :: a -> Maybe a
   return x  = Just x

   (>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
   m >>= g = case m of
                Nothing -> Nothing
                Just x  -> g x
```

- Maybe is the monad, and return brings a value into it by wrapping it with Just.
- As for (>>=), it takes a m :: Maybe a value and a g :: a -> Maybe b function. If m is Nothing, there is nothing to do and the result is Nothing. Otherwise, in the Just x case, g is applied to x, the underlying value wrapped in Just, to give a Maybe b result.
- To sum it all up, if there is an underlying value of type a in m, we apply g to it, which brings the underlying value back into the Maybe monad.

Monads: motivation

Continuing this example:
Imagine a family database with two functions, which look up the parents of a person:

```
father :: Person -> Maybe Person
mother :: Person -> Maybe Person
```

The Maybe is there in case our database is missing some information.

We then might want to build other functions, to look up grandparents, etc (next slide)

Monads: motivation

Now, let’s find grandparents:

```
maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
    case mother p of
        Nothing -> Nothing
        Just mom -> father mom
```

But there’s a better way! Maybe is a monad, so can do:

```
maternalGrandfather p = mother p >>= father
```

Monads:

Even more drastic:

```
bothGrandfathers :: Person -> Maybe (Person, Person)
   bothGrandfathers p =
       case father p of
           Nothing -> Nothing
           Just dad ->
               case father dad of
                   Nothing -> Nothing
                   Just gf1 ->                          -- found first
grandfather
                       case mother p of
                           Nothing -> Nothing
                           Just mom ->
                               case father mom of
                                   Nothing -> Nothing
                                   Just gf2 ->          -- found second
grandfather
                                       Just (gf1, gf2)
```

Becomes:

```
bothGrandfathers p =
      father p >>=
          (\dad -> father dad >>=
              (\gf1 -> mother p >>=   -- gf1 is only used in the
final return
                  (\mom -> father mom >>=
                      (\gf2 -> return (gf1,gf2) ))))
```

Monads: nuts and bolts

Monad the typeclass lives in Control.Monad, and has the following methods:

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

       (>>)   :: m a -> m b -> m b
       fail   :: String -> m a
```

- Aside from return and bind, there are two additional methods, (>>) and fail.
  - Both of them have default implementations, and so you don't need to provide them when writing an instance.

Monads: nuts and bolts

Monad is a subclass of Applicative, which is a subclass of Functor. So these are also functors!

Note: (>>) and fail both have default implementations, and so you don't need to provide them when writing an instance.

(>>) sequences two monadic actions when the second action does not involve the result of the first, which is a common scenario for monads such as IO. Simple example:

```
printSomethingTwice :: String -> IO ()
printSomethingTwice str = putStrLn str >> putStrLn str
```

Monads: back to that example

In fact, that grandfather example can be made even better if we use the do notation with braces and semi-colons.

This will look more like imperative code that we're used to! (Like IOs.)

Here, father and mother are functions that might fail to produce results, i.e. raise an exception, and when that happens, the whole do-block will fail, i.e. terminate with an exception.

```
bothGrandfathers p = do {
        dad <- father p;
        gf1 <- father dad;
        mom <- mother p;
        gf2 <- father mom;
        return (gf1, gf2);
      }
```

Monad laws

Monads are required to obey 3 laws:

```
m >>= return      =  m                           -- right unit
return x >>= f    =  f x                         -- left unit

(m >>= f) >>= g   =  m >>= (\x -> f x >>= g)     -- associativity
```

Monads originally come from a branch of mathematics called Category Theory.

(Fortunately, it is entirely unnecessary to understand category theory in order to understand and use monads in Haskell.)

Monadic Programming

1. Having designed a Monad (i.e. 'I need a Monad with this shape), we define an instance of a Monad. This involves essentially writing what
    1. result and
    2. >>= mean in this Monad

1. We can now write programs using >>=, return etc. allowing us to delineate effectful programming

2. Having written the Monad, we can now use the do notation which will include the failure / effectual side-effects.

Note : we can write the programs using 2. or 3. above. do is easier to read !

Monadic Programming

Having designed a Monad (i.e. ‘I need a Monad with this shape), we define an instance of a Monad. This involves essentially writing what

1. result and
2. >>= mean in this Monad

1.For the purposes of this module, we concentrate on the instance of Monad. However, in order to write the full code, we need to remember that

1. Monad is an Applicative
2. Applicative is a Functor

So we need to write the instances down the chain

We look at MyMaybe (you may remember this from Maybe!)

MyMaybe from scratch

```
data MyMaybe a = MyJust a | Nuttin deriving (Eq, Show)

instance Functor MyMaybe where
fmap f (MyJust x) = MyJust (f x)
fmap _ _ = Nuttin

instance Applicative MyMaybe where
pure = MyJust
Nuttin <*> _ =Nuttin
(MyJust f) <*> something = fmap f something

instance Monad MyMaybe where
return = MyJust
Nuttin >>= _ = Nuttin
MyJust x >>= f = f x
```

Using MyMaybe

```haskell
-- use this to test this with ghci
mayf :: Num a => a -> MyMaybe a
mayf x = MyJust (x+1)
```

Call as :

```
*Main> MyJust 4 >>= mayf
MyJust 5
*Main> Nuttin >>= mayf
Nuttin
*Main>
```

Desugaring do blocks from Graham Hutton's Computerphile video.

```
data Expr = Val Int | Div Expr Expr

safediv :: Int-> Int -> Maybe Int
safediv n m = if m == 0 then
                             Nothing
                           else
                             Just (n `div` m)
-- First attempt
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
                      Nothing -> Nothing
                      Just n -> case eval y of
                                   Nothing -> Nothing
                                   Just m -> safediv n m
```

Desugaring do blocks

```
-- Second  attempt using >>= binding
eval :: Expr -> Maybe Int
eval (Val n) = return n
eval (Div x y) =    eval x >>= (\n ->
                    eval y >>= (\m ->
                    safediv n m ))
```

```
-- Third attempt using do block
eval :: Expr -> Maybe Int
eval (Val n)    =    return n
eval (Div x y) = do n <- eval x
                    m <- eval y
                    safediv n m
```

Desugaring do blocks

Bind Operator

```
do x <- foo          ===          foo >>= (\x -> bar)
   bar
```

Sequence Operator

```
do act1              ===          act1 >> act2
   act2
```