Monads in Haskell

An Introduction
With drawings!

1. Unwrap value from context

2. Apply function

3. Rewrap value in context
Agenda

- Some Haskell basics
- The type system and Typeclasses
- Functors
- Monads
- The Real World
Motivation

- Haskell (and most functional languages) have:
  - No side effects
  - No concept of global state

- How do we do things that inherently have side effects or need state?
Some Haskell basics

- *Functions* are the most important construct
- Function application is just the name of the function followed by parameters:

```haskell
ghci> head [2,4,5]
2
ghci> (\x -> x * x) 2
4
ghci> let square x = x * x
```
Types

\[
\text{square} :: \text{Int} \rightarrow \text{Int} \\
\text{square} = x \times x
\]

\[
\text{head'} :: [a] \rightarrow a \\
\text{head'} (x:xs) = x
\]

\[
\text{take} :: \text{Int} \rightarrow [a] \rightarrow [a] \\
\text{take n _ | n <= 0} = [] \\
\text{take _ []} = [] \\
\text{take n (x:xs)} = x : \text{take (n-1) xs}
\]
Created and Parametric Types

data Maybe a = Just a | Nothing
Typeclasses

- NOT classes in OOP sense
- More like interfaces: define common behavior

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```
Functors

data Maybe a = Just a | Nothing
The Functor Typeclass

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

1. `fmap` takes a function (like `(+) 3`) and a functor (like `Just 2`).
2. It returns a new functor (like `Just 5`).
Example - the Maybe a type

```haskell
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
```

1. UNWRAP VALUE FROM CONTEXT
2. APPLY FUNCTION
3. REWRAP VALUE IN CONTEXT
Example - the Maybe a type

```haskell
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

1. No value
2. Don't apply a function
3. End up with Nothing
So what’s a Monad?

```haskell
class Monad m where

    return :: a -> m a

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

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

    fail :: String -> m a
    fail msg = error msg
```
class Monad m where

  return :: a -> m a

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

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

  fail :: String -> m a
  fail msg = error msg
class Monad m where

  return :: a -> m a

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

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

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

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

1. `>>=` takes a monad (like `Just 3`)
2. and a function that returns a monad (like `half`)
3. and it returns a monad
Just an Example

half :: Int -> Maybe Int
ghci> half x = if even x
    then Just x/2
    else Nothing

ghci> half 3
Nothing

ghci> half 12 >>= half
Just 3

ghci> half 50 >>= half >>= half
Nothing
Do Notation

- What if you need to pass more than just the previous result?

ghci> Just 2 >>= (\x -> Just 3 >>= (\y -> Just (x*y)))
Just 6  (Equivalent to: (\x . (\y . x * y)) Just 3 Just 2)

routine = do
  x <- Just 2
  y <- Just 3
  Just (x * y)
Do Notation

- What if you need to pass more than just the previous result?

```
ghci> Just 2 >>= (\x -> Just 3 >>= (\y -> Just (x*y))))
Just 6
```

```
routine = do
    x <- Just 3
    y <- Just 2
    return (x * y)
```
The Writer Monad

1. HALF 8

2. HALF 4

3. COMBINE BOTH LOGS AND RETURN THE FINAL VALUE

(2, )
The Writer Monad

\[
\text{writer1 >> func = do}
\]

1. Use \text{runWriter} to get the values out of the writer

\[
\text{let (val1, logs1) = runWriter writer1}
\]

\[
\text{(val2, logs2) = runWriter$ \text{func} val1 Writer (val2, logs1 ++ logs2)}
\]

2. Apply the function to the value, and get back a writer. Then use \text{runWriter} to get the values from that

3. Combining the logs and return a new writer!
import Control.Monad.Writer

mult2 :: Int -> Writer String Int
mult2 x = do
    tell("Multiplying " ++ show x ++ " by 2! ")
    return (x * 2)

add1 :: Int -> Writer String Int
add1 x = do
    tell("Adding 1 to " ++ show x ++ "! ")
    return (x + 1)
The Writer Monad

ghci> let x = return 2 >>= mult2 >>= add1 >>= mult2
(10,"Multiplying 2 by 2! Adding 1 to 4! Multiplying 5 by 2! ")
The Real World

- I/O Monad
- State Monad
- Arrows
Sources

- Learn You a Haskell
- Functors, Applicatives, And Monads In Pictures - adit.io
- Three Useful Monads - adit.io
- http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html
- Hoogle - Haskell Search Engine