HASKELL MONADS

Curt Clifton
Rose-Hulman Institute of Technology

Please SVN Update your *HaskellInClass* folder, then open *sugar.hs*, *Eddie*.hs*

THE IO MONAD

CAN WE BE JUST A LITTLE BIT IMPURE?

- How are we getting side effects if Haskell is a pure language?
- Solution: Pass along an object to be "mutated"
- Original: *f :: Tree → Int*
- New: *f :: (Tree, State) → (Int, State)*

Original State

"Mutated" State

Monads automate this pattern

MONADIC MAPS

```
strToMessage :: String -> String
strToMessage s = "… sir: " ++ s

putMessage :: String -> IO ()
putMessage = putStrLn . strToMessage

strings = ["Lancelot", "Robin"]

ex3 = do
    putMessage "Start me up"
    mapM_ putMessage strings
    putMessage "That's all folks!"
```

```
ghci> :type mapM
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
ghci> :type mapM_
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m ()
```

Q1,2

THE MONAD TYPECLASS

Sequences two expressions that have Monad results

Sequences two Monad expressions binding result of first for use in second

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

Wrap pure value in Monad

DA DO DO DO

- The *do* expression in Haskell is just a sugar for Monad sequencing

Inside *do*	Monad notation
`e1` `e2`	`e1 >>= \_ -> e2` or `e1 >> e2`

`x <- e1`
`e2`
	`e1 >>= \x -> e2`

`return e1`	`return e1`

SUGAR FREE!

```
        ex4 = do
            putStr "WHAT is your name? "
            inpStr1 <- getLine
            putStrLn ("Bugger off, " ++ inpStr1 ++ "!")
```

desugar

```
ex5 =
    putStr "What is your name? " >>
    getLine >>=
        (\inpStr -> putStrLn ("Bugger off, " ++ inpStr ++ "!"))
```

desugar

```
ex6 =
    putStr "What is your name? " >>=
        (\_ -> getLine >>=
                (\inpStr -> putStrLn ("Bugger off, " ++ inpStr ++ "!")))
```

IMPLEMENTING AN INTERPRETER USING MONADS

THE LANGUAGE: EDDIE

- Syntax:
  - 42
  - 30 + 12
  - 6 * 7
  - 85 / 2
  - x
  - x = 2; y = x * 3; x = y * 7; x

Typical semantics, except integer division

imperative (non-functional) assignment

Q4

IMPLEMENTING EDDIE

- *EddieTypes.hs:*
  - Defines the data types
- *EddieParse.hs:*
  - Defines a parser for Eddie using the Parsec module
- *EddieEval.hs:*
  - Where we'll define an interpreter for Eddie