HASKELL WRAP-UP

Curt Clifton
Rose-Hulman Institute of Technology

RECALL...

MONADS ARE A GENERAL SOLUTION TO LOTS OF PROBLEMS

GENERAL IDEA

- A computation with a certain type of result
- A certain type of structure in its result
- Need to pass the result of one
  of these computations to another

MONAD TYPECLASS

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

return takes a value of the inner type and wraps it in a computation

binding operator takes a computation

and feeds its value to a function

that makes a another computation

MONADS WE HAVE KNOWN

- Maybe
- List
- State s

get, put, runState

TRAPPED IN A MONAD

- How do we get results from computation?
  - Pattern match
  - Could use support functions if provided
- Without these the result is trapped!

http://www.flickr.com/photos/snugglepup/

DE DO DO DO

do is just sugar

http://www.flickr.com/photos/hopefoote/

MONAD BINDING

```
c >>= \x -> ...
```

```
do
    x <- c
    ...
```

MONAD BINDING

```
evalBinA op lt rt =
  eval lt
  >>= \lv -> eval rt
  >>= \rv -> return (lv `op` rv)
```

```
evalBinA op lt rt = do
  lv <- eval lt
  rv <- eval rt
  return (lv `op` rv)
```

MONAD SEQUENCING

```
eval (Set x t t') =
   eval t
   >>= \xv -> get
   >>= \env -> put (insert x xv env)
   >> eval t'
```

```
eval (Set x t t') = do
   xv <- eval t
   env <- get
   put (insert x xv env)
   eval t'
```

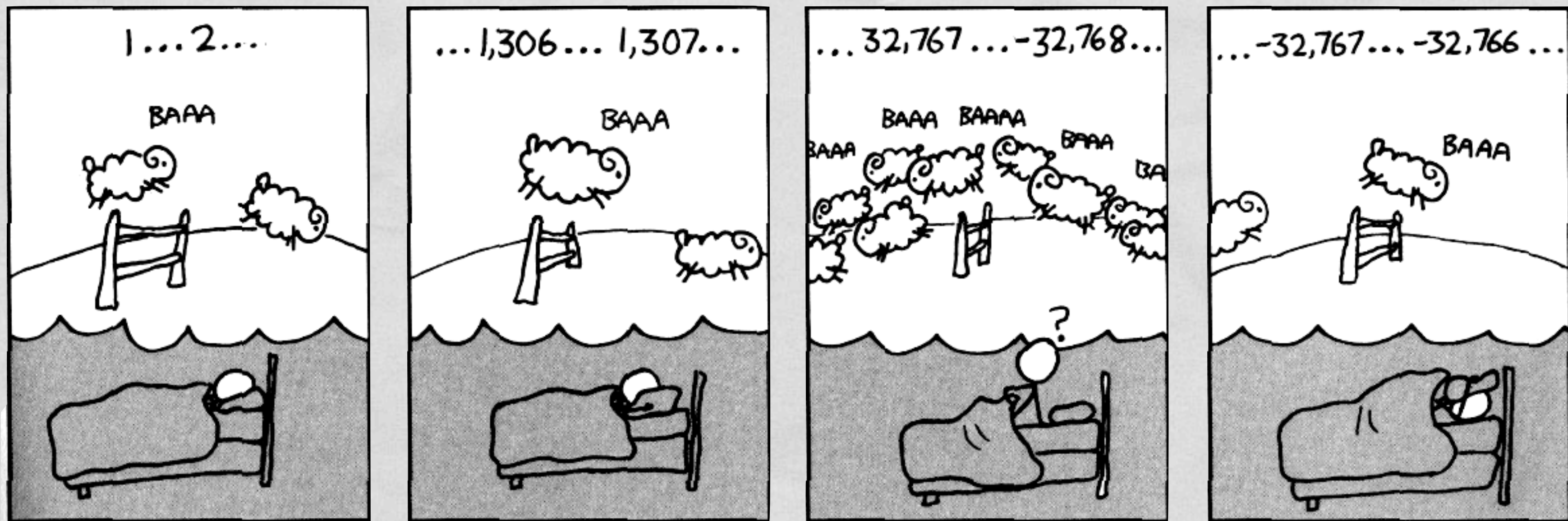
Sugar is just a newline

```
eval :: Term -> EnvState Value
eval (Const v) = return v
eval (Div lt rt) = evalBinA div lt rt
eval (Mult lt rt) = evalBinA (*) lt rt
eval (Sum lt rt) = evalBinA (+) lt rt
eval (Get x) =
   get
   >>= \env -> case (Data.Map.lookup x env) of
               Just v -> return v
               Nothing -> fail (x : " unbound")
eval (Set x t t') =
   eval t
   >>= \xv -> get
   >>= \env -> put (insert x xv env)
   >> eval t'

evalBinA op lt rt =
   eval lt
   >>= \lv -> eval rt
   >>= \rv -> return (lv `op` rv)
```

```
eval :: Term -> EnvState Value
eval (Const v) = return v
eval (Div lt rt) = evalBinA div lt rt
eval (Mult lt rt) = evalBinA (*) lt rt
eval (Sum lt rt) = evalBinA (+) lt rt
eval (Get x) = do
   env <- get
   case (Data.Map.lookup x env) of
      Just v  -> return v
      Nothing -> fail (x : " unbound")
eval (Set x t t') = do
   xv <- eval t
   env <- get
   put (insert x xv env)
   eval t'

evalBinA op lt rt = do
   lv <- eval lt
   rv <- eval rt
   return (lv `op` rv)
```

Q1,2

CARTOON OF THE DAY



If androids someday do dream of electronic sheep don't forget to declare *sheepCount* as a long int.

THE IO MONAD

Thought experiment

- type IO = State Universe

- ex1 = do

```
    putStr "WHAT is your name? "
    inpStr1 <- getLine
    putStr "WHAT is your quest? "
    inpStr2 <- getLine
    putStrLn ("Good luck with that, " ++ inpStr1 ++ "!")
```

putStr :: String -> IO ()

getLine :: IO String

THE IO MONAD

Thought experiment

- type IO = State Universe
- strangeDays :: IO ()

```
strangeDays :: IO ()
strangeDays = do
    world0 <- get
    putStrLn "The cat is dead"
    put world0
    putStrLn "The cat is free"
    put world0
```

runState

IO MONAD

```
> :i IO
newtype IO a
  = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
               -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
```

POSITIVE CHARACTERISTICS

Q3

NEGATIVE CHARACTERISTICS

Q4