AoPL Chapter 6
Error Checking & Monads

William Harrison

CS 4450

Principles of Programming Languages

Some slides originally from William Cook

Resources

Lecture covers:

- Chapter 6 of “Anatomy of Programming Languages”

http://www.cs.utexas.edu/~wcook/anatomy/anatomy.htm

Interpreter so far

- Our current JavaScript-like interpreter already supports a number of features:
  - basic expressions (arithmetic & conditionals)
  - variable declarations
  - function definitions & first-class functions
  - recursion

Interpreter so far

- This allows us to write some programs, such as:

```
var fact = function(n){
   if (n == 0) 1; else n * fact(n-1)
};
var x = 10;
fact(x)
```

JavaScript Interpreter so far

- However, what happens if we write?

```
var fact = function(n){
   if (n == 0) 1; else n * fatn(n-1)
};
fact(10)
```

JavaScript Interpreter so far

- However, what happens if we write?

```
var fact = function(n){
   if (n == 0) 1; else n * fatn(n-1)
};
fact(10)
```

If we try in ghci:

```
*Main> execute fact2
*** Exception: Maybe.fromJust: Nothing
```

JavaScript Interpreter so far

```
*Main> execute fact2
*** Exception: Maybe.fromJust: Nothing
```

Not very helpful or informative as an error message!

Errors

- Errors are an important aspect of computation.
- They are typically a pervasive feature of a language, because they affect the way every expression is evaluated. For example, consider the expression:

  a + b

- If a or b raise errors then we need to deal with this possibility.

Errors

- Because errors are so pervasive they are a notorious problem in programming and programming languages.
- When coding in C the convention is to check the return codes of all system calls.
- However this is often not done.
- Java’s exception handling mechanism provides a more robust way to deal with errors.

Errors

- During the course so far we have already encountered some ways to deal with errors:

  - We used Haskell's "error" function

  - Also, we've seen uses of the "Maybe" datatype

```
data Maybe a = Just a | Nothing
```

Maybe

- The Maybe datatype provides a useful mechanism to deal with errors:

```
data Maybe a = Nothing | Just a
```

Error!

Good result!

Maybe

- However, sometimes we would like to track some more information about what went wrong.
- For example, perhaps we would like to report an error message.
- The Maybe datatype is limiting in this case, because Nothing does not track any information.
- How can we improve on the Maybe datatype to provide more useful information?

Representing Errors

- We can create a datatype Checked, provides a constructor Error to be used instead of Nothing

```
data Checked a = Good a | Error String
```

A good value!

Error with an error message!

Interpreter that deals with Errors

- Using Checked, we can reimplement the eval function to deal with errors.

```
eval :: Exp -> Env -> Checked Value
```

What kind of errors can we have in the current interpreter?

Kinds of Errors

- Three kinds of errors:
  - Division by zero

    3/0

  - Undefined variable errors

    var x = 3; x + y

  - Type-errors

    3 + True

Implementing Error Handling

Undefined Variables

- Dealing with undefined variables:

```
eval (Variable x) env =
    case lookup x of
         Nothing -> Error ("Unbound " ++ x)
         Just v  -> Good v
```

Now we can provide an error message

Currently

```
binary :: Op -> Value -> Value -> Value
binary Add (IntV a)   (IntV b)   = IntV (a + b)
…
binary Div (IntV a)   (IntV b)   = IntV (a `div` b)
binary And (BoolV a) (BoolV b) = BoolV (a && b)
…
binary EQ  a          b          = BoolV (a == b)
```

Type Errors

- Dealing with type errors:

```
checked_binary :: Op -> Value -> Value -> Checked Value
checked_binary Add (IntV a) (IntV b)
                          = Good (IntV (a + b))
…
checked_binary Div (IntV a) (IntV b)
                          = Good (IntV (a `div` b))
checked_binary And (BoolV a) (BoolV b)
                          = Good (BoolV (a && b))
…
checked_binary EQ a b     = Good (BoolV (a == b))
checked_binary _ _ _      = Error “Type Error!”
```

Division by zero

- Division by zero:

```
checked_binary Div (IntV _)   (IntV 0)
        = Error “Division by Zero!”
checked_binary Div (IntV a)   (IntV b)
        = Good (IntV (a `div` b))
```

Propagating errors

- Propagating errors:

```
eval (Binary op a b) env =
  case eval a env of
    Error msg -> Error msg
    Good av ->
      case eval b env of
        Error msg -> Error msg
        Good bv ->
          checked_binary op av bv
```

A bit longwinded though!

Before Checked

- Evaluating binary operators before Checked:

```
eval (Binary op a b) env =
  binary op (eval a env) (eval b env)
```

After Checked

- Evaluating binary operators after Checked:

```
eval (Binary op a b) env =
  case eval a env of
    Error msg -> Error msg
    Good av ->
      case eval b env of
        Error msg -> Error msg
        Good bv ->
          checked_binary op av bv
```

Can we make the code less messy?

Spotting the pattern

- Evaluating binary operators after errors:

```
eval (Binary op a b) env =
  case eval a env of
    Error msg -> Error msg
    Good av ->
      case eval b env of
        Error msg -> Error msg
        Good bv ->
          checked_binary op av bv
```

There seems to be a pattern here.

Spotting the pattern

- We seem to have something like this:

```
case first-part of
    Error msg -> Error msg
    Good v    -> next-part v
```

How to capture this pattern as reusable code?

Spotting the pattern

- Use a higher-order function!

first-part >>= *next-part* =
  `case` *first-part* `of`
    `Error msg -> Error msg`
    `Good v    ->` *next-part* `v`

Revising the Implementation

Creating auxiliary definitions

- The higher-order function capturing error propagation:

```
(>>=) :: Checked a -> (a -> Checked b) -> Checked b
x >>= f =
  case x of
    Error msg -> Error msg
    Good v -> f v
```

We will call this function bind (since it binds a value v)

- A function that creates checked values:

```
return :: a -> Checked a
return v = Good v
```

Rewriting Evaluation

- Here is the new version (4 cases) of evaluation:

```
evalM (Literal v) env = return v
evalM (Variable x) env =
  case lookup x env of
    Nothing -> Error ("Variable " ++ x ++ " undefined")
    Just v  -> return v
evalM (Unary op a) env =
  evalM a env >>= checked_unary op
evalM (Binary op a b) env =
  evalM a env >>=
    \v1 -> evalM b env >>=
       \v2 -> checked_binary op v1 v2
```

Propagating errors

- Compare:

```
evalM (Binary op a b) env =
  evalM a env >>=
    \v1 -> evalM b env >>=
       \v2 -> checked_binary op v1 v2
```

Propagating errors

- with

```
eval (Binary op a b) env =
  case eval a env of
    Error msg -> Error msg
    Good av ->
      case eval b env of
        Error msg -> Error msg
        Good bv ->
          checked_binary op av bv
```

This code is definitely longer.

Propagating errors

- FYI:

```
evalM (Binary op a b) env =
   (evalM a env) >>=
     (\v1 -> (evalM b env) >>=
        (\v2 -> (checked_binary op v1 v2)))
```

- I would typically write this as:

```
evalM (Binary op a b) env =
      evalM a env >>= \v1 ->
      evalM b env >>= \v2 ->
      checked_binary op v1 v2
```

Propagating errors

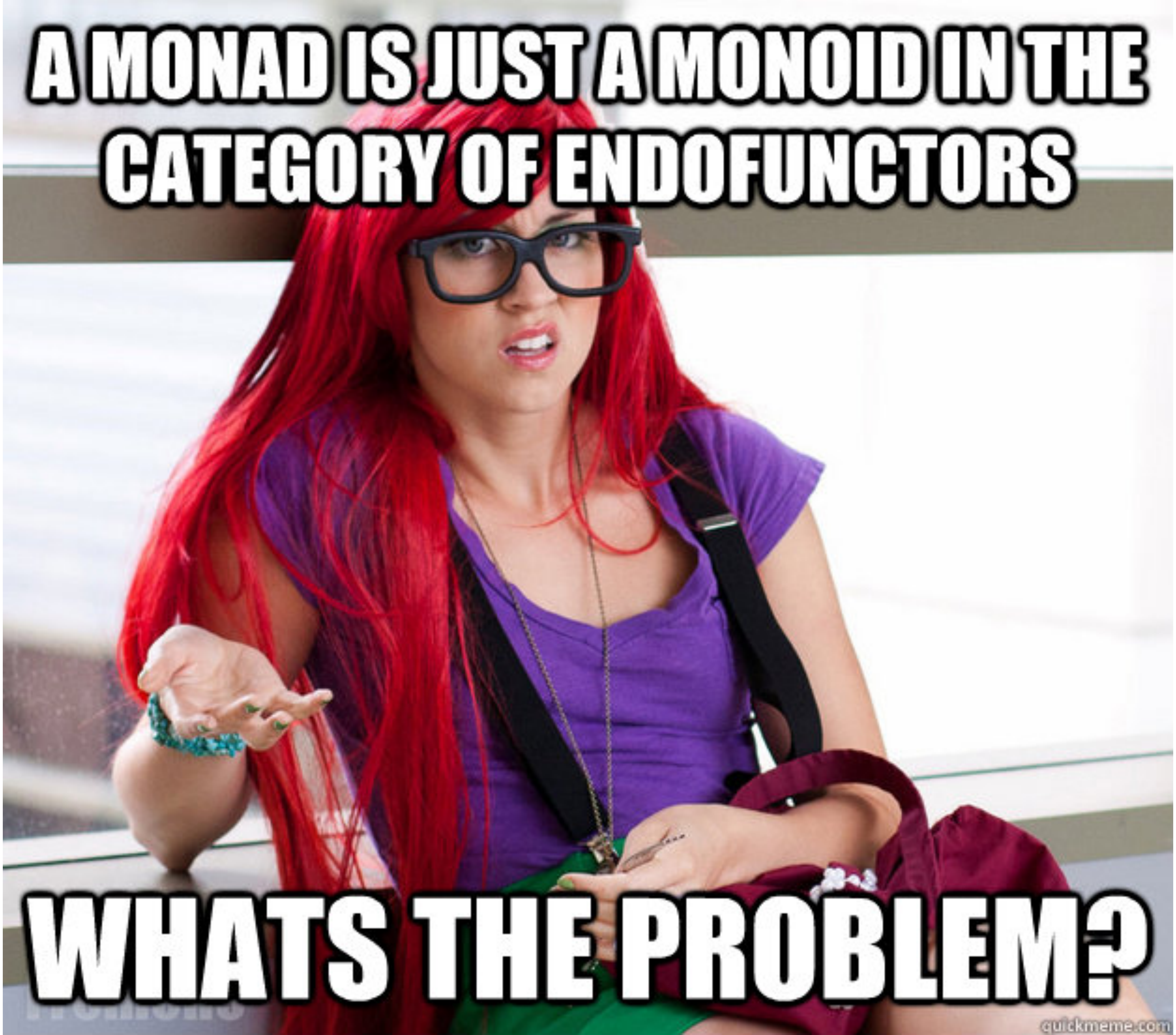
- Compare:

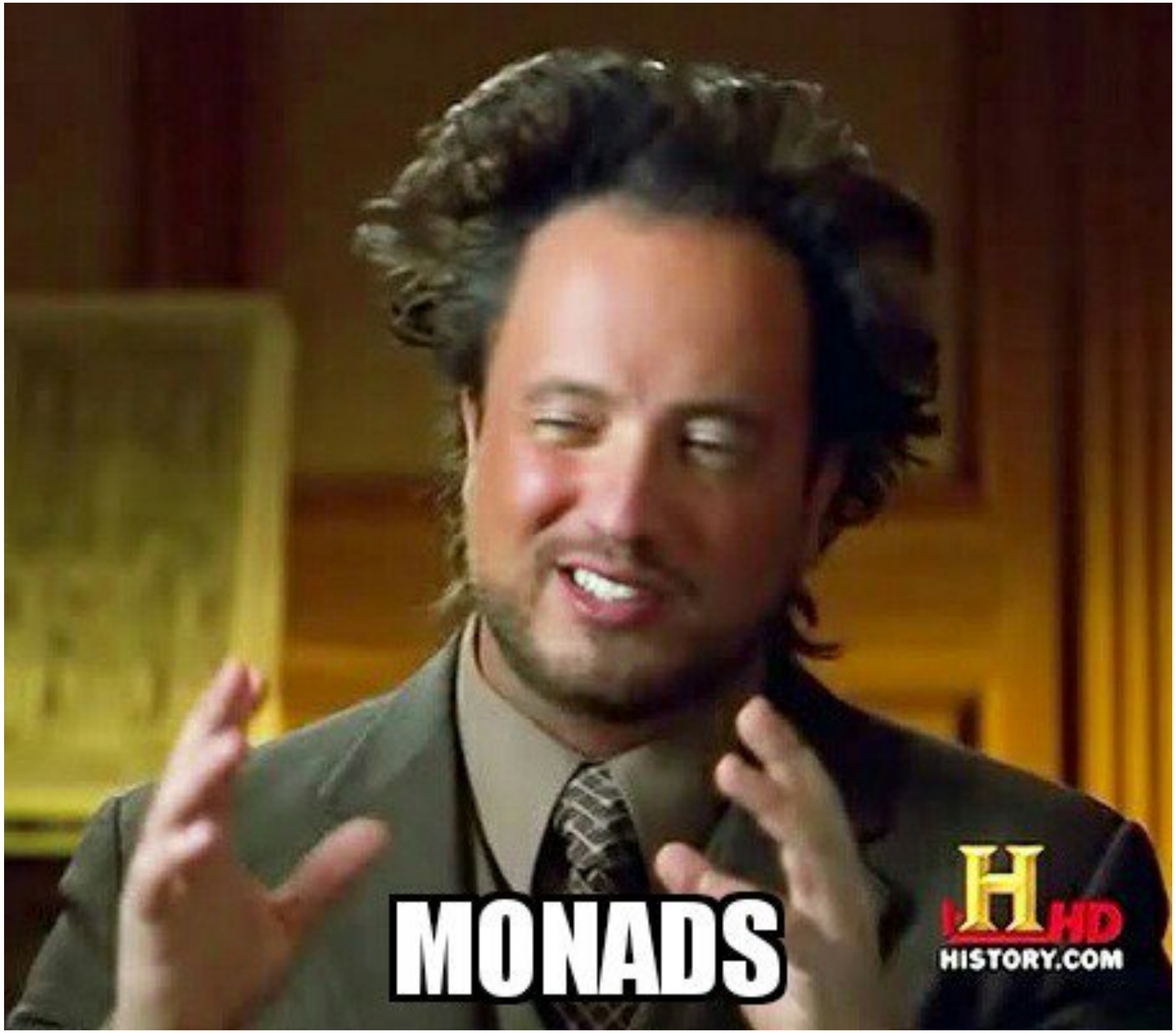
```
evalM (Binary op a b) env =
   evalM a env >>=
      \v1 -> evalM b env >>=
         \v2 -> checked_binary op v1 v2
```

Still, the use of bind may not immediately intuitive.

Monads


DO I LOOK LIKE I KNOW
WHAT A MONAD IS?
made on imgur


A MONAD IS JUST A MONOID IN THE CATEGORY OF ENDOFUNCTORS
WHATS THE PROBLEM?
quickmeme.com

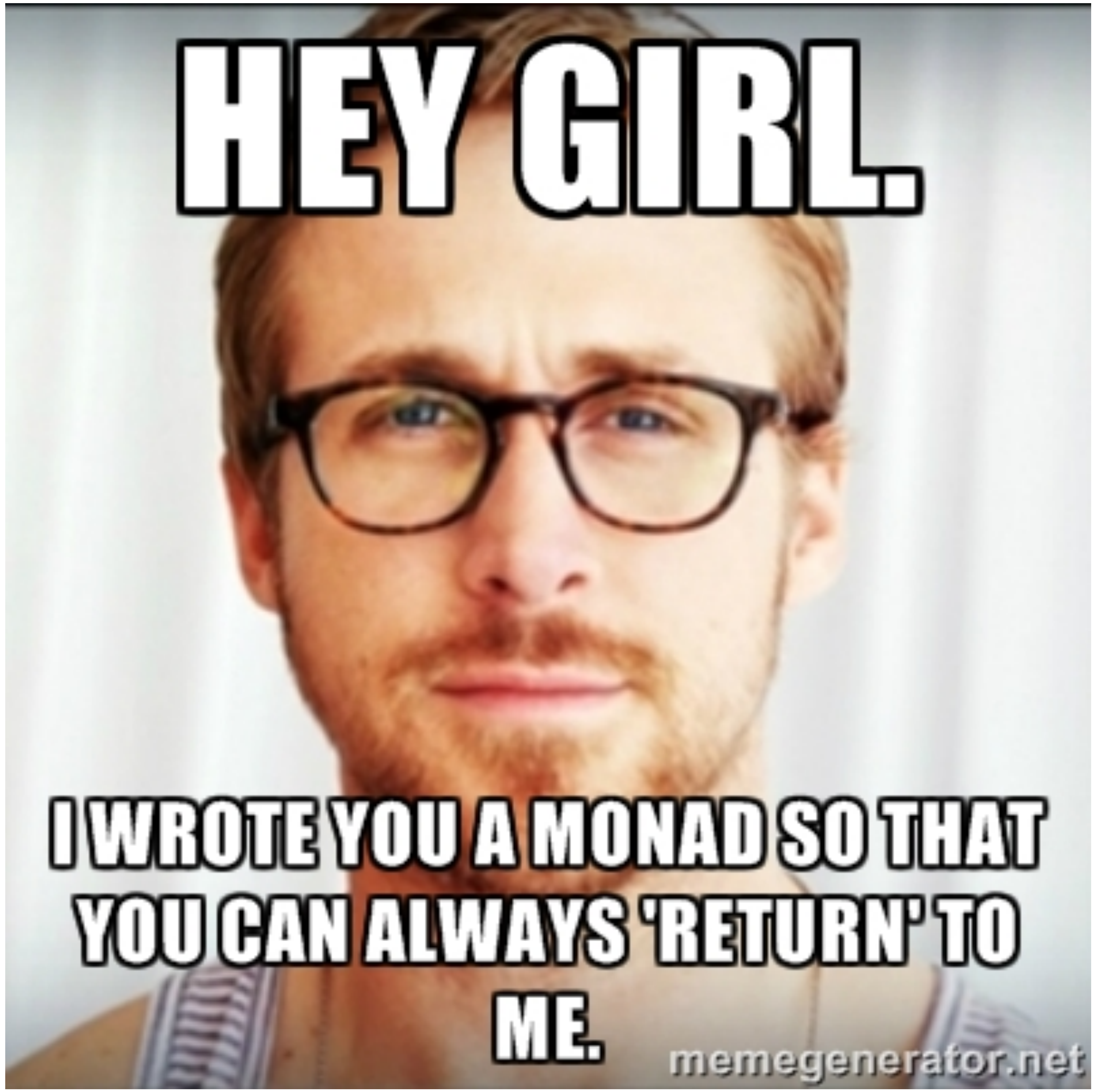

MONADS
HD
HISTORY.COM

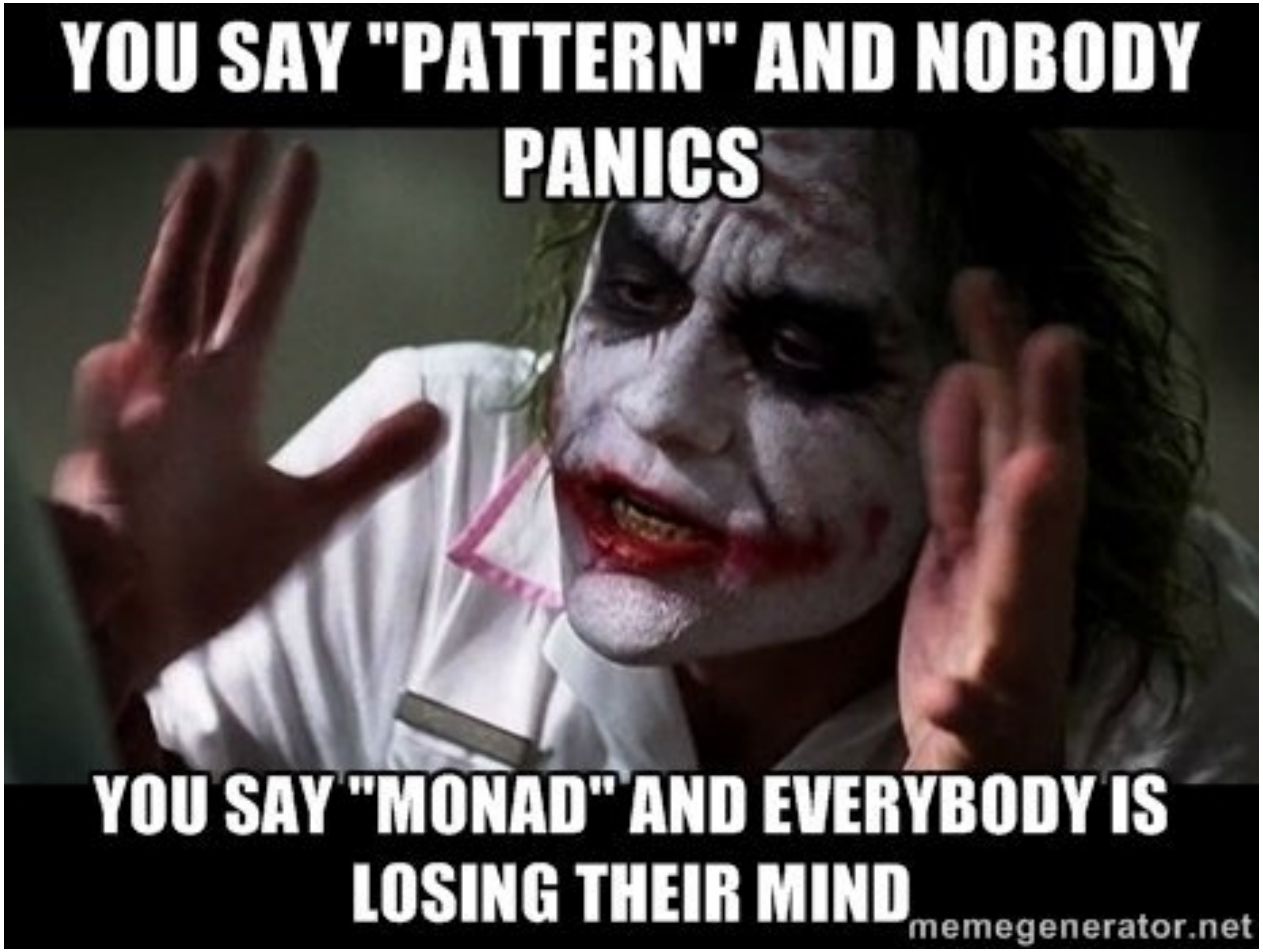

SAY MONAD
ONE MORE TIME
memegenerator.net

Monads in Haskell

- Monads are a structure composed of two basic operations (bind and return), which capture a common pattern that occurs in many types.
- In Haskell Monads are implemented using type classes:

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


HEY GIRL.
I WROTE YOU A MONAD SO THAT YOU CAN ALWAYS 'RETURN' TO ME.
memegenerator.net


YOU SAY "PATTERN" AND NOBODY PANICS
YOU SAY "MONAD" AND EVERYBODY IS LOSING THEIR MIND
memegenerator.net

Checked as a Monad

Because Checked can implement return and bind it can be made an instance of Monad

```
instance Monad Checked where
   return v = Good v
   x >>= f  =
     case x of
       Error msg -> Error msg
       Good v -> f v
```

Rewriting Code Again

- Using bind and return from the Monad class does not affect the code:

```
evalM :: Monad m => Exp -> Env -> m Value
evalM (Binary op a b) env =
  evalM a env >>=
    \v1 -> evalM b env >>=
      \v2 -> checked_binary op v1 v2
```

Rewriting Code Again

- However, because monads are so pervasive, Haskell supports a special notation for monads (do-notation).
- With the do-notation we can re-write the program as follows:

```
evalM (Binary op a b) env =
  do v1 <- eval a env
     v2 <- eval b env
     checked_binary op v1 v2
```

Do-notation

- In Haskell, code using the do-notation, such as:

```
do  pattern <- exp
    morelines
```

Is converted to code using this transformation:

```
exp >>= (\pattern -> do morelines)
```

Monad Laws

- It is not enough to implement bind and return. A proper monad is also required to satisfy some laws:

```
return a >>= k ==  k a
m >>= return   ==  m
m >>= (\x -> k x >>= h) ==  (m >>= k) >>= h
```