The Hassle with Monads

Panicz Maciej Godek

@PaniczGodek

`https://github.com/panicz/writings/tree/master/talks/datamass`

datamass.io summit, 28.09.2018

Before we begin - questions to the audience:

- do you program?
- do you not program?

Before we begin - questions to the audience:

- do you program?
- do you not program?

Before we begin - questions to the audience:

- do you program?
- do you not program?

Which languages do you know?

- Java/C#
- Python
- JavaScript
- C/C++
- Scala
- Erlang/Elixir
- OCaml/F#
- Haskell
- some other?

Which languages do you know?

- Java/C#
- Python
- JavaScript
- C/C++
- Scala
- Erlang/Elixir
- OCaml/F#
- Haskell
- some other?

Which languages do you know?

- Java/C#
- Python
- JavaScript
- C/C++
- Scala
- Erlang/Elixir
- OCaml/F#
- Haskell
- some other?

Which languages do you know?

- Java/C#
- Python
- JavaScript
- C/C++
- Scala
- Erlang/Elixir
- OCaml/F#
- Haskell
- some other?

Which languages do you know?

- Java/C#
- Python
- JavaScript
- C/C++
- Scala
- Erlang/Elixir
- OCaml/F#
- Haskell
- some other?

Which languages do you know?

- Java/C#
- Python
- JavaScript
- C/C++
- Scala
- Erlang/Elixir
- OCaml/F#
- Haskell
- some other?

Which languages do you know?

- Java/C#
- Python
- JavaScript
- C/C++
- Scala
- Erlang/Elixir
- OCaml/F#
- Haskell
- some other?

Which languages do you know?

- Java/C#
- Python
- JavaScript
- C/C++
- Scala
- Erlang/Elixir
- OCaml/F#
- Haskell
- some other?

Which languages do you know?

- Java/C#
- Python
- JavaScript
- C/C++
- Scala
- Erlang/Elixir
- OCaml/F#
- Haskell
- some other?

Which languages do you know?

- Java/C#
- Python
- JavaScript
- C/C++
- Scala
- Erlang/Elixir
- OCaml/F#
- Haskell
- some other?

On to the topic

- do you know functional programming?
- have you heard about monads?
- do you understand monads?
- did you try to understand monads and failed?

On to the topic

- do you know functional programming?
- have you heard about monads?
- do you understand monads?
- did you try to understand monads and failed?

On to the topic

- do you know functional programming?
- have you heard about monads?
- do you understand monads?
- did you try to understand monads and failed?

On to the topic

- do you know functional programming?
- have you heard about monads?
- do you understand monads?
- did you try to understand monads and failed?

On to the topic

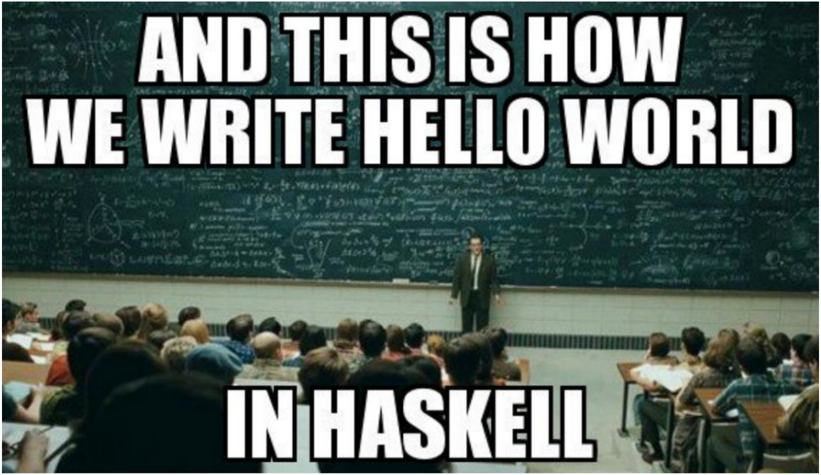
- do you know functional programming?
- have you heard about monads?
- do you understand monads?
- did you try to understand monads and failed?

Why learn monads?

Yes, monads seem to be a form of AspectOrientedProgramming, since they serve to isolate a generalized computational strategy from the specifics of an algorithm. For example, in HaskellLanguage you can write a graph-searching procedure that can either do a depth-first search and return the first result, or do a breadth-first search and return a list of results, merely by running it in a different monad.

`http://wiki.c2.com/?AspectOrientedProgramming`

Why learn monads?



Why (lazy) functional programming?

Why (lazy) functional programming?

```
numbersFrom n = n:numbersFrom(n+1)
numbersFrom 0 = [0,1,2,3,4,5,6,7,8,9,10,...]

sieve (h:t) = h:(sieve [x|x<-t,x`mod`h/=0])
primes = sieve (numbersFrom 2)
primes = [2,3,5,7,11,13,17,19,23,29,31,37,...]
```

Jerzy Karczmarczuk, *Generating Power of Lazy Semantics*
John Hughes, *Why Functional Programming Matters*

Why (lazy) functional programming?

```
numbersFrom n = n:numbersFrom(n+1)
numbersFrom 0 = [0,1,2,3,4,5,6,7,8,9,10,...]
```

```
sieve (h:t) = h:(sieve [x|x<-t,x`mod`h/=0])
primes = sieve (numbersFrom 2)
primes = [2,3,5,7,11,13,17,19,23,29,31,37,...]
```

Jerzy Karczmarczuk, *Generating Power of Lazy Semantics*
John Hughes, *Why Functional Programming Matters*

Why (lazy) functional programming?

```
numbersFrom n = n:numbersFrom(n+1)
numbersFrom 0 = [0,1,2,3,4,5,6,7,8,9,10,...]

sieve (h:t) = h:(sieve [x|x<-t,x`mod`h/=0])
primes = sieve (numbersFrom 2)
primes = [2,3,5,7,11,13,17,19,23,29,31,37,...]
```

Jerzy Karczmarczuk, *Generating Power of Lazy Semantics*
John Hughes, *Why Functional Programming Matters*

Why (lazy) functional programming?

```
numbersFrom n = n:numbersFrom(n+1)
numbersFrom 0 = [0,1,2,3,4,5,6,7,8,9,10,...]

sieve (h:t) = h:(sieve [x|x<-t,x`mod`h/=0])
primes = sieve (numbersFrom 2)
primes = [2,3,5,7,11,13,17,19,23,29,31,37,...]
```

Jerzy Karczmarczuk, *Generating Power of Lazy Semantics*
John Hughes, *Why Functional Programming Matters*

Why (lazy) functional programming?

```
numbersFrom n = n:numbersFrom(n+1)
numbersFrom 0 = [0,1,2,3,4,5,6,7,8,9,10,...]

sieve (h:t) = h:(sieve [x|x<-t,x`mod`h/=0])
primes = sieve (numbersFrom 2)
primes = [2,3,5,7,11,13,17,19,23,29,31,37,...]
```

Jerzy Karczmarczuk, *Generating Power of Lazy Semantics*
John Hughes, *Why Functional Programming Matters*

Why (lazy) functional programming?

```
numbersFrom n = n:numbersFrom(n+1)
numbersFrom 0 = [0,1,2,3,4,5,6,7,8,9,10,...]

sieve (h:t) = h:(sieve [x|x<-t,x`mod`h/=0])
primes = sieve (numbersFrom 2)
primes = [2,3,5,7,11,13,17,19,23,29,31,37,...]
```

Jerzy Karczmarczuk, *Generating Power of Lazy Semantics*
John Hughes, *Why Functional Programming Matters*

Why (lazy) functional programming?

```
numbersFrom n = n:numbersFrom(n+1)
numbersFrom 0 = [0,1,2,3,4,5,6,7,8,9,10,...]

sieve (h:t) = h:(sieve [x|x<-t,x`mod`h/=0])
primes = sieve (numbersFrom 2)
primes = [2,3,5,7,11,13,17,19,23,29,31,37,...]
```

Jerzy Karczmarczuk, *Generating Power of Lazy Semantics*
John Hughes, *Why Functional Programming Matters*

The essence of laziness

```
square x = x * x
```

Applicative order (evaluate arguments before expansion):

square (2*3) = square 6 $=_{def}$ 6 * 6 = 36

Normal order (evaluate arguments after expansion):

square (2*3) $=_{def}$ (2*3) * (2*3) = 6 * 6 = 36

The essence of laziness

```
square x = x * x
```

Applicative order (evaluate arguments before expansion):

square (2*3) = square 6 $=_{def}$ 6 * 6 = 36

Normal order (evaluate arguments after expansion):

square (2*3) $=_{def}$ (2*3) * (2*3) = 6 * 6 = 36

The essence of laziness

```
square x = x * x
```

Applicative order (evaluate arguments before expansion):

square (2*3) = square 6 $=_{def}$ 6 * 6 = 36

Normal order (evaluate arguments after expansion):

square (2*3) $=_{def}$ (2*3) * (2*3) = 6 * 6 = 36

The essence of laziness

```
square x = x * x
```

Applicative order (evaluate arguments before expansion):

square (2*3) = square 6 $=_{def}$ 6 * 6 = 36

Normal order (evaluate arguments after expansion):

square (2*3) $=_{def}$ (2*3) * (2*3) = 6 * 6 = 36

The essence of laziness

```
square x = x * x
```

Applicative order (evaluate arguments before expansion):

square (2*3) = square 6 $=_{def}$ 6 * 6 = 36

Normal order (evaluate arguments after expansion):

square (2*3) $=_{def}$ (2*3) * (2*3) = 6 * 6 = 36

The essence of laziness

```
square x = x * x
```

Applicative order (evaluate arguments before expansion):

square (2*3) = square 6 $=_{def}$ 6 * 6 = 36

Normal order (evaluate arguments after expansion):

square (2*3) $=_{def}$ (2*3) * (2*3) = 6 * 6 = 36

The essence of laziness

```
square x = x * x
```

Applicative order (evaluate arguments before expansion):

square (2*3) = square 6 $=_{def}$ 6 * 6 = 36

Normal order (evaluate arguments after expansion):

square (2*3) $=_{def}$ (2*3) * (2*3) = 6 * 6 = 36

The essence of laziness

```
square x = x * x
```

Applicative order (evaluate arguments before expansion):

square (2*3) = square 6 $=_{def}$ 6 * 6 = 36

Normal order (evaluate arguments after expansion):

square (2*3) $=_{def}$ (2*3) * (2*3) = 6 * 6 = 36

The essence of laziness

```
square x = x * x
```

Applicative order (evaluate arguments before expansion):

`square (2*3) = square 6` $=_{def}$ `6 * 6 = 36`

Normal order (evaluate arguments after expansion):

`square (2*3)` $=_{def}$ `(2*3) * (2*3) = 6 * 6 = 36`

The essence of laziness

```
square x = x * x
```

Applicative order (evaluate arguments before expansion):

square (2*3) = square 6 $=_{def}$ 6 * 6 = 36

Normal order (evaluate arguments after expansion):

square (2*3) $=_{def}$ (2*3) * (2*3) = 6 * 6 = 36

The essence of laziness

```
square x = x * x
```

Applicative order (evaluate arguments before expansion):

square (2*3) = square 6 $=_{def}$ 6 * 6 = 36

Normal order (evaluate arguments after expansion):

square (2*3) $=_{def}$ (2*3) * (2*3) = 6 * 6 = 36

Lambda the Ultimate

```
square x = x * x
square = λ x -> x * x
square = function(x) { return x * x; }

distance x y = abs(x-y)
distance = λ x -> λ y -> abs(x-y)
```
(currying)

```
let name = value in expression
(λ name -> expression) value
```

Lambda the Ultimate

```
square x = x * x
square = λ x -> x * x
square = function(x) { return x * x; }

distance x y = abs(x-y)
distance = λ x -> λ y -> abs(x-y)
```
(currying)

```
let name = value in expression
(λ name -> expression) value
```

Lambda the Ultimate

```
square x = x * x
square = λ x -> x * x
square = function(x) { return x * x; }

distance x y = abs(x-y)
distance = λ x -> λ y -> abs(x-y) (currying)

let name = value in expression
(λ name -> expression) value
```

Lambda the Ultimate

```
square x = x * x
square = λ x -> x * x
square = function(x) { return x * x; }

distance x y = abs(x-y)
distance = λ x -> λ y -> abs(x-y)
```
(currying)

```
let name = value in expression
(λ name -> expression) value
```

Lambda the Ultimate

```
square x = x * x
square = λ x -> x * x
square = function(x) { return x * x; }

distance x y = abs(x-y)
distance = λ x -> λ y -> abs(x-y)
```
(currying)

```
let name = value in expression
(λ name -> expression) value
```

Lambda the Ultimate

```
square x = x * x
square = λ x -> x * x
square = function(x) { return x * x; }

distance x y = abs(x-y)
distance = λ x -> λ y -> abs(x-y)
```
(currying)

```
let name = value in expression
(λ name -> expression) value
```

Lambda the Ultimate

```
square x = x * x
square = λ x -> x * x
square = function(x) { return x * x; }

distance x y = abs(x-y)
distance = λ x -> λ y -> abs(x-y)
```
(currying)

```
let name = value in expression
(λ name -> expression) value
```

The problem with I/O

```
1*3 + 2*0

readNumber()*3 + 2*readNumber()
< 1
< 0
```

The problem with I/O

```
1*3 + 2*0

readNumber()*3 + 2*readNumber()
< 1
< 0
```

The problem with I/O

```
1*3 + 2*0

readNumber()*3 + 2*readNumber()
< 1
< 0
```

Attempted solution

```
let   a      = readNumber(   ) in
  let   b      = readNumber(   ) in
     a*2 + 3*b
```

Attempted solution

```
let  a     = readNumber(  ) in
  let  b     = readNumber(  ) in
    a*2 + 3*b
```

Attempted solution

```
let  a     = readNumber(  ) in
  let  b     = readNumber(  ) in
     a*2 + 3*b
```

Actual solution

```
let (a,w1) = readNumber(w0) in
  let (b,w2) = readNumber(w1) in
     a*2 + 3*b
```

Better (composable) solution

```
let (a,w1) = readNumber(w0) in
  let (b,w2) = readNumber(w1) in
    (a*2 + 3*b, w2)
```

A function

```
myOperation w0 =
let (a,w1) = readNumber(w0) in
  let (b,w2) = readNumber(w1) in
    (a*2 + 3*b, w2)
```

A function

```
myOperation :: RealWorld -> (Int, RealWorld)
myOperation w0 =
let (a,w1) = readNumber(w0) in
  let (b,w2) = readNumber(w1) in
    (a*2 + 3*b, w2)
```

https://wiki.haskell.org/IO_inside

Downsides

- need to pass additional parameter
- prone to errors (e.g. `w0` instead of `w1`)
- nesting level increases

Could we do something to make `w0` passed implicitly?

Downsides

- need to pass additional parameter
- prone to errors (e.g. `w0` instead of `w1`)
- nesting level increases

Could we do something to make `w0` passed implicitly?

Downsides

- need to pass additional parameter
- prone to errors (e.g. `w0` instead of `w1`)
- nesting level increases

Could we do something to make `w0` passed implicitly?

Downsides

- need to pass additional parameter
- prone to errors (e.g. `w0` instead of `w1`)
- nesting level increases

Could we do something to make `w0` passed implicitly?

Downsides

- need to pass additional parameter
- prone to errors (e.g. `w0` instead of `w1`)
- nesting level increases

Could we do something to make `w0` passed implicitly?

How could this look like?

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

How could this look like?

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

How could this look like?

```
pass readNumber
     (λ a -> pass readNumber
                   (λ b -> return a*2 + 3*b))

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

How could this look like?

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

How could this look like?

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

Does this really work?

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
    continuation result w1
```

Does this really work?

```
pass readNumber
     (λ a -> pass readNumber
                 (λ b -> return a*2 + 3*b))

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

Does this really work?

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

Does this really work?

```
pass readNumber
     (λ a -> pass readNumber
              (λ b -> λ w -> (a*2 + 3*b, w)))

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

Does this really work?

```
pass readNumber
     (λ a -> pass readNumber
               (λ b -> λ w -> (a*2 + 3*b, w)))

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

Does this really work?

```
pass readNumber (λ a -> λ w1 ->
  let (y, w2) = readNumber(w1) in
        (λ b -> λ w -> (a*2 + 3*b, w)) y w2)

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

Does this really work?

```
pass readNumber (λ a -> λ w1 ->
  let (y, w2) = readNumber(w1) in
         (λ b -> λ w -> (a*2 + 3*b, w)) y w2)

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

Does this really work?

```
pass readNumber (λ a -> λ w1 ->
   let (y, w2) = readNumber(w1) in
          (λ w -> (a*2 + 3*y, w)) w2)

return value world = (value, world)

pass value continuation w0 =
   let (result, w1) = value w0 in
       continuation result w1
```

Does this really work?

```
pass readNumber (λ a -> λ w1 ->
  let (y, w2) = readNumber(w1) in
        (a*2 + 3*y, w2))

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

Does this really work?

```
pass readNumber (λ a -> λ w1 ->
  let (y, w2) = readNumber(w1) in
        (a*2 + 3*y, w2))

return value world = (value, world)

pass value continuation w0 =
  let (result, w1) = value w0 in
     continuation result w1
```

Does this really work?

```
λ w0 -> let (x,w3) = readNumber(w0) in
   (λ a -> λ w1 -> let (y, w2) = readNumber(w1)
        in (a*2 + 3*y, w2)) x w3

return value world = (value, world)

pass value continuation w0 =
   let (result, w1) = value w0 in
      continuation result w1
```

Does this really work?

```
λ w0 -> let (x,w3) = readNumber(w0) in
   (λ a -> λ w1 -> let (y, w2) = readNumber(w1)
        in (a*2 + 3*y, w2)) x w3

return value world = (value, world)

pass value continuation w0 =
   let (result, w1) = value w0 in
       continuation result w1
```

Does this really work?

```
λ w0 -> let (x,w3) = readNumber(w0) in
   (λ w1 -> let (y, w2) = readNumber(w1) in
             (x*2 + 3*y, w2)) w3

return value world = (value, world)

pass value continuation w0 =
   let (result, w1) = value w0 in
      continuation result w1
```

Does this really work?

```
λ w0 -> let (x,w3) = readNumber(w0) in
   let (y, w2) = readNumber(w3) in
          (x*2 + 3*y, w2)

return value world = (value, world)

pass value continuation w0 =
   let (result, w1) = value w0 in
       continuation result w1
```

Does this really work?

```
λ w0 -> let (x,w3) = readNumber(w0) in
  let (y, w2) = readNumber(w3) in
         (x*2 + 3*y, w2)
```

```
let (a,w1) = readNumber(w0) in
  let (b,w2) = readNumber(w1) in
    (a*2 + 3*b, w2)
```

It works!

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))
```

But typing λ and the increased indentation level is annoying!

```
pass readNumber (λ a
-> pass readNumber (λ b
-> return a*2 + 3*b))
```

It works!

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))
```

But typing λ and the increased indentation level is annoying!

```
pass readNumber (λ a
-> pass readNumber (λ b
-> return a*2 + 3*b))
```

It works!

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))
```

But typing λ and the increased indentation level is annoying!

```
pass readNumber (λ a
-> pass readNumber (λ b
-> return a*2 + 3*b))
```

Introduce new syntax (`do`-notation):

```
do result <- action
   actions ...
```

can be interpreted as:

```
pass action (λ result -> do actions ...)
```

Note: In Haskell, `pass` function is spelled »= and pronounced "bind".

Introduce new syntax (`do`-notation):

```
do result <- action
   actions ...
```

can be interpreted as:

```
pass action (λ result -> do actions ...)
```

Note: In Haskell, `pass` function is spelled »= and pronounced "bind".

Introduce new syntax (`do`-notation):

```
do result <- action
   actions ...
```

can be interpreted as:

```
pass action (λ result -> do actions ...)
```

Note: In Haskell, `pass` function is spelled »= and pronounced "bind".

Introduce new syntax (`do`-notation):

```
do result <- action
   actions ...
```

can be interpreted as:

```
pass action (λ result -> do actions ...)
```

Note: In Haskell, `pass` function is spelled »= and pronounced "bind".

Emperor's new clothes

Now we can write our program as:

```
do a <- readNumber
   b <- readNumber
   return a*2 + 3*b
```

The sequencing pattern

A monad (sequencing pattern) consists of:

1. a »= (bind, pass, chain) function that takes some (decorated) value and a function (*continuation*) and passes that value to the function
2. a `return` function that takes some (raw) value and lifts (decorates) it, so that it can be chained using the »= operator

The monad laws:

`((return v) »= f) = (f v)` – *left identity*

`(m »= return) = m` – *right identity*

The sequencing pattern

A monad (sequencing pattern) consists of:

1. a »= (bind, pass, chain) function that takes some (decorated) value and a function (*continuation*) and passes that value to the function
2. a `return` function that takes some (raw) value and lifts (decorates) it, so that it can be chained using the »= operator

The monad laws:

`((return v) »= f) = (f v)` – *left identity*

`(m »= return) = m` – *right identity*

The sequencing pattern

A monad (sequencing pattern) consists of:

1. a »= (bind, pass, chain) function that takes some (decorated) value and a function (*continuation*) and passes that value to the function
2. a `return` function that takes some (raw) value and lifts (decorates) it, so that it can be chained using the »= operator

The monad laws:

`((return v) »= f) = (f v)` – *left identity*

`(m »= return) = m` – *right identity*

The sequencing pattern

A monad (sequencing pattern) consists of:

1. a »= (bind, pass, chain) function that takes some (decorated) value and a function (*continuation*) and passes that value to the function
2. a `return` function that takes some (raw) value and lifts (decorates) it, so that it can be chained using the »= operator

The monad laws:

`((return v) »= f) = (f v)` – *left identity*

`(m »= return) = m` – *right identity*

The sequencing pattern

A monad (sequencing pattern) consists of:

1. a »= (bind, pass, chain) function that takes some (decorated) value and a function (*continuation*) and passes that value to the function
2. a `return` function that takes some (raw) value and lifts (decorates) it, so that it can be chained using the »= operator

The monad laws:

`((return v) »= f) = (f v)` – *left identity*

`(m »= return) = m` – *right identity*

The sequencing pattern

A monad (sequencing pattern) consists of:

1. a »= (bind, pass, chain) function that takes some (decorated) value and a function (*continuation*) and passes that value to the function
2. a `return` function that takes some (raw) value and lifts (decorates) it, so that it can be chained using the »= operator

The monad laws:

`((return v) »= f) = (f v)` – *left identity*

`(m »= return) = m` – *right identity*

The sequencing pattern

A monad (sequencing pattern) consists of:

1. a »= (bind, pass, chain) function that takes some (decorated) value and a function (*continuation*) and passes that value to the function
2. a `return` function that takes some (raw) value and lifts (decorates) it, so that it can be chained using the »= operator

The monad laws:

`((return v) »= f) = (f v)` – *left identity*

`(m »= return) = m` – *right identity*

The sequencing pattern

A monad (sequencing pattern) consists of:

1. a `»=` (bind, pass, chain) function that takes some (decorated) value and a function (*continuation*) and passes that value to the function
2. a `return` function that takes some (raw) value and lifts (decorates) it, so that it can be chained using the `»=` operator

The monad laws:

`((return v) »= f) = (f v)` – *left identity*

`(m »= return) = m` – *right identity*

The sequencing pattern

A monad (sequencing pattern) consists of:

1. a `»=` (bind, pass, chain) function that takes some (decorated) value and a function (*continuation*) and passes that value to the function
2. a `return` function that takes some (raw) value and lifts (decorates) it, so that it can be chained using the `»=` operator

The monad laws:

`((return v) »= f) = (f v)` – *left identity*

`(m »= return) = m` – *right identity*

In Haskell

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

Monads with the `do` notation provide a general and systematic solution to the common anti-pattern known as *the Pyramid of Doom*.

In Haskell

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

Monads with the `do` notation provide a general and systematic solution to the common anti-pattern known as *the Pyramid of Doom*.

In Haskell

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

Monads with the `do` notation provide a general and systematic solution to the common anti-pattern known as *the Pyramid of Doom*.

In Haskell

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

Monads with the `do` notation provide a general and systematic solution to the common anti-pattern known as *the Pyramid of Doom*.

In Haskell

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

Monads with the `do` notation provide a general and systematic solution to the common anti-pattern known as *the Pyramid of Doom*.

The Pyramid of Doom

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

The Pyramid of Doom

*In computer programming, the **pyramid of doom** is a common problem that arises when a program uses many levels of nested indentation to control access to a function. It is commonly seen when checking for null pointers or handling callbacks.*

Wikipedia/Pyramid_of_doom_(programming)

The Pyramid of Doom – example

```
theWidth = windows("Main").views(5).size().width();

if windows.contains("Main") {
  if windows("Main").views.contains(5) {
    theWidth = windows("Main").views(5).size().width();
    //more code that works with theWidth
  }
}
```

With “optional chaining”/“null-conditional”/“safe navigation” operator:

```
theWidth = windows("Main")?.views(5)?.size.width;
```

The Pyramid of Doom – example

```
theWidth = windows("Main").views(5).size().width();
```

```
if windows.contains("Main") {
  if windows("Main").views.contains(5) {
    theWidth = windows("Main").views(5).size().width();
    //more code that works with theWidth
  }
}
```

With “optional chaining”/“null-conditional”/“safe navigation” operator:

```
theWidth = windows("Main")?.views(5)?.size.width;
```

The Pyramid of Doom – example

```
theWidth = windows("Main").views(5).size().width();

if windows.contains("Main") {
  if windows("Main").views.contains(5) {
    theWidth = windows("Main").views(5).size().width();
    //more code that works with theWidth
  }
}
```

With “optional chaining”/“null-conditional”/“safe navigation” operator:

```
theWidth = windows("Main")?.views(5)?.size.width;
```

The Pyramid of Doom – example

```
theWidth = windows("Main").views(5).size().width();

if windows.contains("Main") {
  if windows("Main").views.contains(5) {
    theWidth = windows("Main").views(5).size().width();
    //more code that works with theWidth
  }
}
```

With "optional chaining"/"null-conditional"/"safe navigation" operator:

```
theWidth = windows("Main")?.views(5)?.size.width;
```

The Pyramid of Doom – example

```
theWidth = windows("Main").views(5).size().width();

if windows.contains("Main") {
  if windows("Main").views.contains(5) {
    theWidth = windows("Main").views(5).size().width();
    //more code that works with theWidth
  }
}
```

With “optional chaining”/“null-conditional”/“safe navigation” operator:

```
theWidth = windows("Main")?.views(5)?.size.width;
```

Dealing with “no values” in Haskell

```
data Maybe a = Nothing | Just a

let theWidth = do window <- windows("Main")
                  view <- views 5 window
                  return width (size view)

instance Monad Maybe where
  (Nothing »= f)  = Nothing
  (Just a »= f) = (f a)
  return a = Just a
```

Dealing with “no values” in Haskell

```
data Maybe a = Nothing | Just a

let theWidth = do window <- windows("Main")
                  view <- views 5 window
                  return width (size view)

instance Monad Maybe where
  (Nothing »= f) = Nothing
  (Just a »= f) = (f a)
  return a = Just a
```

Dealing with “no values” in Haskell

```
data Maybe a = Nothing | Just a

let theWidth = do window <- windows("Main")
                  view <- views 5 window
                  return width (size view)

instance Monad Maybe where
  (Nothing »= f) = Nothing
  (Just a »= f) = (f a)
  return a = Just a
```

Dealing with “no values” in Haskell

```
data Maybe a = Nothing | Just a

let theWidth = do window <- windows("Main")
                  view <- views 5 window
                  return width (size view)

instance Monad Maybe where
  (Nothing »= f)  = Nothing
  (Just a »= f) = (f a)
  return a = Just a
```

Dealing with “no values” in Haskell

```
data Maybe a = Nothing | Just a

let theWidth = do window <- windows("Main")
                  view <- views 5 window
                  return width (size view)

instance Monad Maybe where
  (Nothing »= f) = Nothing
  (Just a »= f) = (f a)
  return a = Just a
```

Dealing with “no values” in Haskell

```
data Maybe a = Nothing | Just a

let theWidth = do window <- windows("Main")
                  view <- views 5 window
                  return width (size view)

instance Monad Maybe where
  (Nothing >>= f) = Nothing
  (Just a >>= f) = (f a)
  return a = Just a
```

Dealing with “no values” in Haskell

```
data Maybe a = Nothing | Just a

let theWidth = do window <- windows("Main")
                  view <- views 5 window
                  return width (size view)

instance Monad Maybe where
  (Nothing »= f) = Nothing
  (Just a »= f) = (f a)
  return a = Just a
```

The List Monad

```
instance Monad List where
  (x »= f) = concatMap f x
  return a = [a]

>>> concatMap (λ n -> [1..n]) [1,2,3]
 [1, 1,2, 1,2,3]

>>> do a <- [1,2,3]
       b <- [4,5]
       return (a, b)

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

The List Monad

```
instance Monad List where
  (x »= f) = concatMap f x
  return a = [a]

>>>  concatMap (λ n -> [1..n]) [1,2,3]
 [1, 1,2, 1,2,3]

>>>  do a <- [1,2,3]
        b <- [4,5]
        return (a, b)

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

The List Monad

```
instance Monad List where
  (x »= f) = concatMap f x
  return a = [a]
```

```
>>> concatMap (λ n -> [1..n]) [1,2,3]
 [1, 1,2, 1,2,3]

>>> do a <- [1,2,3]
       b <- [4,5]
       return (a, b)

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

The List Monad

```
instance Monad List where
  (x »= f) = concatMap f x
  return a = [a]

>>>  concatMap (λ n -> [1..n]) [1,2,3]
 [1, 1,2, 1,2,3]

>>>  do a <- [1,2,3]
        b <- [4,5]
        return (a, b)

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

The List Monad

```
instance Monad List where
  (x »= f) = concatMap f x
  return a = [a]
```

```
>>>  concatMap (λ n -> [1..n]) [1,2,3]
 [1, 1,2, 1,2,3]
```

```
>>>  do a <- [1,2,3]
        b <- [4,5]
        return (a, b)

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

The List Monad

```
instance Monad List where
  (x »= f) = concatMap f x
  return a = [a]

>>>  concatMap (λ n -> [1..n]) [1,2,3]
 [1, 1,2, 1,2,3]

>>>  do a <- [1,2,3]
        b <- [4,5]
        return (a, b)

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

The List Monad

```
instance Monad List where
  (x »= f) = concatMap f x
  return a = [a]

>>>  concatMap (λ n -> [1..n]) [1,2,3]
 [1, 1,2, 1,2,3]

>>>  do a <- [1,2,3]
        b <- [4,5]
        return (a, b)

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

The Amb Monad

```
import Control.Monad
import Control.Monad.Amb

pyTriple n = do a <- anIntegerBetween 1 n
                b <- anIntegerBetween (a+1) n
                c <- anIntegerBetween (b+1) n
                when (a*a + b*b /= c*c) empty
                return (a,b,c)

>>>  oneValue (pyTriple 20)
(3,4,5)

>>>  allValues (pyTriple 20)
[(3,4,5),(5,12,13),(6,8,10),(8,15,17),
 (9,12,15),(12,16,20)]
```

The Amb Monad

```
import Control.Monad
import Control.Monad.Amb

pyTriple n = do a <- anIntegerBetween 1 n
                b <- anIntegerBetween (a+1) n
                c <- anIntegerBetween (b+1) n
                when (a*a + b*b /= c*c) empty
                return (a,b,c)
```

```
>>>  oneValue (pyTriple 20)
(3,4,5)

>>>  allValues (pyTriple 20)
[(3,4,5),(5,12,13),(6,8,10),(8,15,17),
 (9,12,15),(12,16,20)]
```

The Amb Monad

```
import Control.Monad
import Control.Monad.Amb

pyTriple n = do a <- anIntegerBetween 1 n
                b <- anIntegerBetween (a+1) n
                c <- anIntegerBetween (b+1) n
                when (a*a + b*b /= c*c) empty
                return (a,b,c)

>>>  oneValue (pyTriple 20)
(3,4,5)

>>>  allValues (pyTriple 20)
[(3,4,5),(5,12,13),(6,8,10),(8,15,17),
 (9,12,15),(12,16,20)]
```

The Amb Monad

```
import Control.Monad
import Control.Monad.Amb

pyTriple n = do a <- anIntegerBetween 1 n
                b <- anIntegerBetween (a+1) n
                c <- anIntegerBetween (b+1) n
                when (a*a + b*b /= c*c) empty
                return (a,b,c)

>>>  oneValue (pyTriple 20)
(3,4,5)

>>>  allValues (pyTriple 20)
[(3,4,5),(5,12,13),(6,8,10),(8,15,17),
 (9,12,15),(12,16,20)]
```

The Amb Monad

```
import Control.Monad
import Control.Monad.Amb

pyTriple n = do a <- anIntegerBetween 1 n
                b <- anIntegerBetween (a+1) n
                c <- anIntegerBetween (b+1) n
                when (a*a + b*b /= c*c) empty
                return (a,b,c)

>>>  oneValue (pyTriple 20)
(3,4,5)

>>>  allValues (pyTriple 20)
[(3,4,5),(5,12,13),(6,8,10),(8,15,17),
 (9,12,15),(12,16,20)]
```

The Amb Monad

```
import Control.Monad
import Control.Monad.Amb

pyTriple n = do a <- anIntegerBetween 1 n
                b <- anIntegerBetween (a+1) n
                c <- anIntegerBetween (b+1) n
                when (a*a + b*b /= c*c) empty
                return (a,b,c)

>>>  oneValue (pyTriple 20)
(3,4,5)

>>>  allValues (pyTriple 20)
[(3,4,5),(5,12,13),(6,8,10),(8,15,17),
 (9,12,15),(12,16,20)]
```

Other instances of monads

Phil Wadler, *The First Monad Tutorial*

`https://www.youtube.com/watch?v=yjmKMhJOJos`

Rob Norris, *Functional Programming with Effects*

`https://www.youtube.com/watch?v=po3wmq4S15A`

Other instances of monads

Phil Wadler, *The First Monad Tutorial*
`https://www.youtube.com/watch?v=yjmKMhJOJos`

Rob Norris, *Functional Programming with Effects*
`https://www.youtube.com/watch?v=po3wmq4S15A`

Monad Laws revisited

Function composition:

```
(f . g) x = f (g x)

function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

`f . (g . h) = (f . g) . h` – *associativity of (.)*

```
(f . (g . h)) x = f ((g . h) x) = f (g (h x))
(((f . g) h)) x = (f . g) (h x) = f (g (h x))
```

Monad Laws revisited

Function composition:

`(f . g) x = f (g x)`

```
function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

`f . (g . h) = (f . g) . h` – *associativity of (.)*

```
(f . (g . h)) x = f ((g . h) x) = f (g (h x))
(((f . g) h)) x = (f . g) (h x) = f (g (h x))
```

Monad Laws revisited

Function composition:

```
(f . g) x = f (g x)

function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

`f . (g . h) = (f . g) . h` – *associativity of (.)*

```
(f . (g . h)) x = f ((g . h) x) = f (g (h x))
(((f . g) h)) x = (f . g) (h x) = f (g (h x))
```

Monad Laws revisited

Function composition:

```
(f . g) x = f (g x)
```

```
function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

`f . (g . h) = (f . g) . h` – *associativity of (.)*

```
(f . (g . h)) x = f ((g . h) x) = f (g (h x))
(((f . g) h)) x = (f . g) (h x) = f (g (h x))
```

Function composition:

```
(f . g) x = f (g x)

function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

`f . (g . h) = (f . g) . h` – *associativity of (.)*

```
(f . (g . h)) x = f ((g . h) x) = f (g (h x))
(((f . g) h)) x = (f . g) (h x) = f (g (h x))
```

Monad Laws revisited

Function composition:

```
(f . g) x = f (g x)

function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

`f . (g . h) = (f . g) . h` – *associativity of (.)*

```
(f . (g . h)) x = f ((g . h) x) = f (g (h x))
(((f . g) h)) x = (f . g) (h x) = f (g (h x))
```

Monad Laws revisited

Function composition:

```
(f . g) x = f (g x)

function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

`f . (g . h) = (f . g) . h` – *associativity of (.)*

```
(f . (g . h)) x = f ((g . h) x) = f (g (h x))
(((f . g) h)) x = (f . g) (h x) = f (g (h x))
```

Monad Laws revisited

Function composition:

```
(f . g) x = f (g x)

function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

`f . (g . h) = (f . g) . h` – *associativity of (.)*

```
(f . (g . h)) x = f ((g . h) x) = f (g (h x))
(((f . g) h)) x = (f . g) (h x) = f (g (h x))
```

Function composition:

```
(f . g) x = f (g x)

function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

`f . (g . h) = (f . g) . h` – *associativity of (.)*

```
(f . (g . h)) x = f ((g . h) x) = f (g (h x))
(((f . g) h)) x = (f . g) (h x) = f (g (h x))
```

Function composition:

```
(f . g) x = f (g x)

function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

`f . (g . h) = (f . g) . h` – *associativity of (.)*

```
(f . (g . h)) x = f ((g . h) x) = f (g (h x))
(((f . g) h)) x = (f . g) (h x) = f (g (h x))
```

Function composition:

```
(f . g) x = f (g x)

function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

`f . (g . h) = (f . g) . h` – *associativity of (.)*

```
(f . (g . h)) x = f ((g . h) x) = f (g (h x))
(((f . g) h)) x = (f . g) (h x) = f (g (h x))
```

Monad Laws revisited

Identity function:

```
id x = x

function identity(x) { return x; }
```

`id . f = f` – *left identity*

`(id . f) x = id (f x) = f x`

`f . id = f` – *right identity*

`(f . id) x = f (id x) = f x`

Monad Laws revisited

Identity function:

```
id x = x
```

```
function identity(x) { return x; }
```

`id . f = f` – *left identity*

`(id . f) x = id (f x) = f x`

`f . id = f` – *right identity*

`(f . id) x = f (id x) = f x`

Monad Laws revisited

Identity function:

```
id x = x

function identity(x) { return x; }
```

`id . f = f` – *left identity*

`(id . f) x = id (f x) = f x`

`f . id = f` – *right identity*

`(f . id) x = f (id x) = f x`

Monad Laws revisited

Identity function:

```
id x = x

function identity(x) { return x; }

id . f = f
```
– *left identity*

```
(id . f) x = id (f x) = f x
f . id = f
```
– *right identity*

```
(f . id) x = f (id x) = f x
```

Monad Laws revisited

Identity function:

```
id x = x

function identity(x) { return x; }
```

`id . f = f` – *left identity*

`(id . f) x = id (f x) = f x`

`f . id = f` – *right identity*

`(f . id) x = f (id x) = f x`

Monad Laws revisited

Identity function:

```
id x = x

function identity(x) { return x; }
```

`id . f = f` – *left identity*

`(id . f) x = id (f x) = f x`

`f . id = f` – *right identity*

`(f . id) x = f (id x) = f x`

Monad Laws revisited

Identity function:

```
id x = x

function identity(x) { return x; }
```

`id . f = f` – *left identity*

`(id . f) x = id (f x) = f x`

`f . id = f` – *right identity*

`(f . id) x = f (id x) = f x`

Monad Laws revisited

Identity function:

```
id x = x

function identity(x) { return x; }
```

`id . f = f` – *left identity*

`(id . f) x = id (f x) = f x`

`f . id = f` – *right identity*

`(f . id) x = f (id x) = f x`

Monad Laws revisited

Identity function:

```
id x = x

function identity(x) { return x; }
```

`id . f = f` – *left identity*

`(id . f) x = id (f x) = f x`

`f . id = f` – *right identity*

`(f . id) x = f (id x) = f x`

Monad Laws revisited

Identity function:

```
id x = x

function identity(x) { return x; }
```

`id . f = f` – *left identity*

`(id . f) x = id (f x) = f x`

`f . id = f` – *right identity*

`(f . id) x = f (id x) = f x`

Monad Laws revisited

Identity function:

```
id x = x

function identity(x) { return x; }
```

`id . f = f` – *left identity*

`(id . f) x = id (f x) = f x`

`f . id = f` – *right identity*

`(f . id) x = f (id x) = f x`

Monad Laws revisited

Identity function:

```
id x = x

function identity(x) { return x; }
```

`id . f = f` – *left identity*

`(id . f) x = id (f x) = f x`

`f . id = f` – *right identity*

`(f . id) x = f (id x) = f x`

Monad Laws revisited

Identity function:

```
id x = x

function identity(x) { return x; }
```

`id . f = f` – *left identity*

`(id . f) x = id (f x) = f x`

`f . id = f` – *right identity*

`(f . id) x = f (id x) = f x`

Monad Laws revisited

associativity + identity = monoid (semi-group with neutral element)

$(\circ, id)$ is a monoid. Other examples:

- $(+, 0)$
- $(*, 1)$
- $(min, +\infty)$
- $(max, -\infty)$

Monad Laws revisited

associativity + identity = monoid (semi-group with neutral element)

$(\circ, id)$ is a monoid. Other examples:

- $(+, 0)$
- $(*, 1)$
- $(min, +\infty)$
- $(max, -\infty)$

Monad Laws revisited

associativity + identity = monoid (semi-group with neutral element)

$(\circ, id)$ is a monoid. Other examples:

- $(+, 0)$
- $(*, 1)$
- $(min, +\infty)$
- $(max, -\infty)$

Monad Laws revisited

associativity + identity = monoid (semi-group with neutral element)

$(\circ, id)$ is a monoid. Other examples:

- $(+, 0)$
- $(*, 1)$
- $(min, +\infty)$
- $(max, -\infty)$

Monad Laws revisited

associativity + identity = monoid (semi-group with neutral element)

$(\circ, id)$ is a monoid. Other examples:

- $(+, 0)$
- $(*, 1)$
- $(min, +\infty)$
- $(max, -\infty)$

Monad Laws revisited

associativity + identity = monoid (semi-group with neutral element)

$(\circ, id)$ is a monoid. Other examples:

- $(+, 0)$
- $(*, 1)$
- $(min, +\infty)$
- $(max, -\infty)$

Monad Laws revisited

Kleisli composition:

```
(f >=> g) x = do y <- f x
                 g y

(f >=> g) x = (f x) »= g

(return >=> f) = f
(f >=> return) = f
((f >=> g) >=> h) = (f >=> (g >=> h))
```

Monad Laws revisited

Kleisli composition:

```
(f >=> g) x = do y <- f x
                 g y
```

```
(f >=> g) x = (f x) »= g
```

```
(return >=> f) = f
(f >=> return) = f
((f >=> g) >=> h) = (f >=> (g >=> h))
```

Monad Laws revisited

Kleisli composition:

```
(f >=> g) x = do y <- f x
                 g y
```

```
(f >=> g) x = (f x) »= g
```

```
(return >=> f) = f
(f >=> return) = f
((f >=> g) >=> h) = (f >=> (g >=> h))
```

Monad Laws revisited

Kleisli composition:

```
(f >=> g) x = do y <- f x
                 g y

(f >=> g) x = (f x) »= g

(return >=> f) = f
(f >=> return) = f
((f >=> g) >=> h) = (f >=> (g >=> h))
```

Kleisli composition:

```
(f >=> g) x = do y <- f x
                 g y

(f >=> g) x = (f x) »= g

(return >=> f) = f
(f >=> return) = f
((f >=> g) >=> h) = (f >=> (g >=> h))
```

Monad Laws revisited

Kleisli composition:

```
(f >=> g) x = do y <- f x
                 g y

(f >=> g) x = (f x) »= g

(return >=> f) = f
(f >=> return) = f
((f >=> g) >=> h) = (f >=> (g >=> h))
```

Monad Laws revisited

Functors, Applicatives, Monads

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

class Functor f => Applicative f where
  pure ::  a -> f a
  (<*>) ::  f (a -> b) -> f a -> f b

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

Functors, Applicatives, Monads

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

class Functor f => Applicative f where
  pure ::  a -> f a
  (<*>) ::  f (a -> b) -> f a -> f b

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

Functors, Applicatives, Monads

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

class Functor f => Applicative f where
  pure ::  a -> f a
  (<*>) ::  f (a -> b) -> f a -> f b

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

Functors, Applicatives, Monads

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

class Functor f => Applicative f where
  pure ::  a -> f a
  (<*>) ::  f (a -> b) -> f a -> f b

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

Free monads

```
data IO a =
  PutStrLn String (IO a)
  | GetStrLn (String -> IO a)
  | Sleep Int (IO a)
  | DeleteFile String (IO a)
  | LaunchTheMissles (IO a)
  ...
  | forall a0.  Chain (IO a0) (a0 -> IO a)
  | Return a
```

Problems:

- `IO` can do really anything
- we may want to restrict it to a smaller number of capabilities
- we may want to simulate (mock) the behavior

Free monads

```
data IO a =
  PutStrLn String (IO a)
  | GetStrLn (String -> IO a)
  | Sleep Int (IO a)
  | DeleteFile String (IO a)
  | LaunchTheMissles (IO a)
  ...
  | forall a0.  Chain (IO a0) (a0 -> IO a)
  | Return a
```

Problems:

- `IO` can do really anything
- we may want to restrict it to a smaller number of capabilities
- we may want to simulate (mock) the behavior

Free monads

```
data IO a =
  PutStrLn String (IO a)
  | GetStrLn (String -> IO a)
  | Sleep Int (IO a)
  | DeleteFile String (IO a)
  | LaunchTheMissles (IO a)
  ...
  | forall a0.  Chain (IO a0) (a0 -> IO a)
  | Return a
```

Problems:

- `IO` can do really anything
- we may want to restrict it to a smaller number of capabilities
- we may want to simulate (mock) the behavior

Free monads

```
data IO a =
  PutStrLn String (IO a)
  | GetStrLn (String -> IO a)
  | Sleep Int (IO a)
  | DeleteFile String (IO a)
  | LaunchTheMissles (IO a)
  ...
  | forall a0.  Chain (IO a0) (a0 -> IO a)
  | Return a
```

Problems:

- `IO` can do really anything
- we may want to restrict it to a smaller number of capabilities
- we may want to simulate (mock) the behavior

Free monads

```
data IO a =
  PutStrLn String (IO a)
  | GetStrLn (String -> IO a)
  | Sleep Int (IO a)
  | DeleteFile String (IO a)
  | LaunchTheMissles (IO a)
  ...
  | forall a0.  Chain (IO a0) (a0 -> IO a)
  | Return a
```

Problems:

- `IO` can do really anything
- we may want to restrict it to a smaller number of capabilities
- we may want to simulate (mock) the behavior

Free monads

```
data IO a =
  PutStrLn String (IO a)
  | GetStrLn (String -> IO a)
  | Sleep Int (IO a)
  | DeleteFile String (IO a)
  | LaunchTheMissles (IO a)
  ...
  | forall a0.  Chain (IO a0) (a0 -> IO a)
  | Return a
```

Problems:

- `IO` can do really anything
- we may want to restrict it to a smaller number of capabilities
- we may want to simulate (mock) the behavior

Free monads

John De Goes, *FP to the Max*
`https://www.youtube.com/watch?v=sxudIMiOo68`

Igal Tabachnik, *Journey to Functional Programming*
`https://www.youtube.com/watch?v=g1EvM4CbUvM`

Kelley Robinson, *Why the Free Monad isnt' Free*
`https://www.youtube.com/watch?v=U0lK0hnbc4U`

Free monads

John De Goes, *FP to the Max*
`https://www.youtube.com/watch?v=sxudIMiOo68`

Igal Tabachnik, *Journey to Functional Programming*
`https://www.youtube.com/watch?v=g1EvM4CbUvM`

Kelley Robinson, *Why the Free Monad isnt' Free*
`https://www.youtube.com/watch?v=U0lK0hnbc4U`

Free monads

John De Goes, *FP to the Max*
https://www.youtube.com/watch?v=sxudIMiOo68

Igal Tabachnik, *Journey to Functional Programming*
https://www.youtube.com/watch?v=g1EvM4CbUvM

Kelley Robinson, *Why the Free Monad isnt' Free*
https://www.youtube.com/watch?v=U0lK0hnbc4U

Monad transformers

Problem: different monads do not stack together well.

For example, `Future (Maybe a)` cannot be composed with the single »= operator.
Gabriele Petronella, *Monad transformers down to earth*
https://www.youtube.com/watch?v=jd5e71nFEZM

Oleg Kiselyov, Hiromi Ishii, *Freer Monads, More Extensible Effects*
http://okmij.org/ftp/Haskell/extensible/more.pdf

Daniel Spiewak, *Emm: A Sane Alternative to Monad Transformers in Scala*
https://www.youtube.com/watch?v=E5Tri3Yow0U

Monad transformers

Problem: different monads do not stack together well.

For example, `Future (Maybe a)` cannot be composed with the single »= operator.

Gabriele Petronella, *Monad transformers down to earth*
https://www.youtube.com/watch?v=jd5e71nFEZM

Oleg Kiselyov, Hiromi Ishii, *Freer Monads, More Extensible Effects*
http://okmij.org/ftp/Haskell/extensible/more.pdf

Daniel Spiewak, *Emm: A Sane Alternative to Monad Transformers in Scala*
https://www.youtube.com/watch?v=E5Tri3Yow0U

Monad transformers

Problem: different monads do not stack together well.

For example, `Future (Maybe a)` cannot be composed with the single »= operator.
Gabriele Petronella, *Monad transformers down to earth*
`https://www.youtube.com/watch?v=jd5e71nFEZM`

Oleg Kiselyov, Hiromi Ishii, *Freer Monads, More Extensible Effects*
`http://okmij.org/ftp/Haskell/extensible/more.pdf`

Daniel Spiewak, *Emm: A Sane Alternative to Monad Transformers in Scala*
`https://www.youtube.com/watch?v=E5Tri3Yow0U`

Monad transformers

Problem: different monads do not stack together well.

For example, `Future (Maybe a)` cannot be composed with the single »= operator.
Gabriele Petronella, *Monad transformers down to earth*
`https://www.youtube.com/watch?v=jd5e71nFEZM`

Oleg Kiselyov, Hiromi Ishii, *Freer Monads, More Extensible Effects*
`http://okmij.org/ftp/Haskell/extensible/more.pdf`

Daniel Spiewak, *Emm: A Sane Alternative to Monad Transformers in Scala*
`https://www.youtube.com/watch?v=E5Tri3Yow0U`

Monad transformers

Problem: different monads do not stack together well.

For example, `Future (Maybe a)` cannot be composed with the single »= operator.
Gabriele Petronella, *Monad transformers down to earth*
https://www.youtube.com/watch?v=jd5e71nFEZM

Oleg Kiselyov, Hiromi Ishii, *Freer Monads, More Extensible Effects*
http://okmij.org/ftp/Haskell/extensible/more.pdf

Daniel Spiewak, *Emm: A Sane Alternative to Monad Transformers in Scala*
https://www.youtube.com/watch?v=E5Tri3Yow0U

Using the IO monad is emulating imperative programming.

Ron Pressler, *Please Stop Polluting our Imperative Languages with Pure Concepts*
`https://www.youtube.com/watch?v=449j7oKQVkc`

Conal Elliott, *The C language is purely functional*
`http://conal.net/blog/posts/the-c-language-is-purely-functional`

Criticism

Using the IO monad is emulating imperative programming.

Ron Pressler, *Please Stop Polluting our Imperative Languages with Pure Concepts*
`https://www.youtube.com/watch?v=449j7oKQVkc`

Conal Elliott, *The C language is purely functional*
`http://conal.net/blog/posts/the-c-language-is-purely-functional`

Criticism

Using the IO monad is emulating imperative programming.

Ron Pressler, *Please Stop Polluting our Imperative Languages with Pure Concepts*
`https://www.youtube.com/watch?v=449j7oKQVkc`

Conal Elliott, *The C language is purely functional*
`http://conal.net/blog/posts/the-c-language-is-purely-functional`

Questions?

@PaniczGodek

`https://www.quora.com/profile/Panicz-Godek`

`https://github.com/panicz/writings/tree/master/talks/datamass`