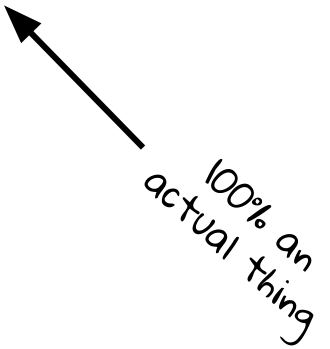

λ
λ
Haskell
GAVE MY CAT A
TOM HARDING
BEARD!
AND A
MOOG

Zygohistomorphic Prepromorphisms

~~Zygohistomorphic Prepromorphisms~~

~~Zygohistomorphic Prepromorphisms~~



OOP Is Masochism

~~OOP Is Masochism~~

Sukant Hajra
@shajra

Follow

Good OO design just leads you to FP.

RETWEETS	LIKES
57	60

9:49 am - 23 Jan 2016

57 60

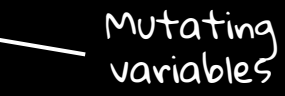

JAVASCRIPT IS
"FUNCTIONAL"
memegenerator.net

Some Ugly Imperative Code

```
var input = [1, 2, 3, 4, 5]
var total = 0

for (var i = 0; i < input.length; i++) {
  if (input[i] <= 3) {
    total += input[i] * 2
  }
}
```

Some Ugly Imperative Code

```
var input = [1, 2, 3, 4, 5]
var total = 0

for (var i = 0; i < input.length; i++) {
  if (input[i] <= 3) {
    total += input[i] * 2
  }
}
```



Some Ugly Imperative Code

```
var input = [1, 2, 3, 4, 5]
var total = 0

for (var i = 0; i < input.length; i++) {
  if (input[i] <= 3) {
    total += input[i] * 2
  }
}
```

Mutating variables

Stateful code blocks

Some Ugly Imperative Code

```
var input = [1, 2, 3, 4, 5]
var total = 0

for (var i = 0; i < input.length; i++) {
  if (input[i] <= 3) {
    total += input[i] * 2
  }
}
```

Mutating variables

Branches

Stateful code blocks

Some Ugly Imperative Code

```
var input = [1, 2, 3, 4, 5]
var total = 0

for (var i = 0; i < input.length; i++) {
  if (input[i] <= 3) {
    total += input[i] * 2
  }
}
```

Mutating variables

Branches

Requires state to connect lines

Stateful code blocks

Some Ugly Imperative Code

```
var input = [1, 2, 3, 4, 5]
var total = 0

for (var i = 0; i < input.length; i++) {
  if (input[i] <= 3) {
    total += input[i] * 2
  }
}
```

Mutating variables

Requires state to connect lines

Branches

Stateful code blocks

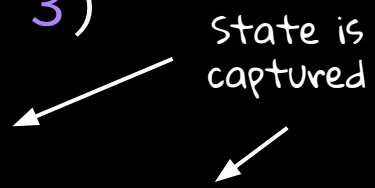
Program **must** run in the order specified

Some Functional Code

```
[1, 2, 3, 4, 5]
  .filter(x => x <= 3)
  .map(x => x * 2)
  .reduce((acc, x) => acc + x, 0)
```

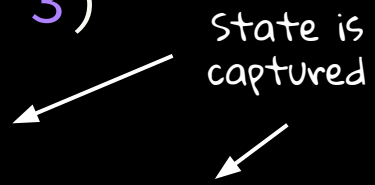
Some Functional Code

```
[1, 2, 3, 4, 5]
  .filter(x => x <= 3)
  .map(x => x * 2)
  .reduce((acc, x) => acc + x, 0)
```



Some Functional Code

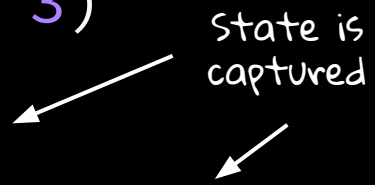
```
[1, 2, 3, 4, 5]
  .filter(x => x <= 3)
  .map(x => x * 2)
  .reduce((acc, x) => acc + x, 0)
```



Obvious program flow ↓

Some Functional Code

```
[1, 2, 3, 4, 5]
  .filter(x => x <= 3)
  .map(x => x * 2)
  .reduce((acc, x) => acc + x, 0)
```



Obvious program flow ↓

HOW DOES IT WORK?

Some Functional Code

```
[1, 2, 3, 4, 5]
  .filter(x => x <= 3)
  .map(x => x * 2)
  .reduce((acc, x) => acc + x, 0)
```

State is captured

Obvious program flow ↓

HOW DOES IT WORK?

Some Functional Code

```
[1, 2, 3, 4, 5]
  .filter(x => x <= 3)
  .map(x => x * 2)
  .reduce((acc, x) => acc + x, 0)
```

State is captured

Obvious program flow ↓

HOW DOES IT WORK?

Some Functional Code

```
[1, 2, 3, 4, 5]
  .filter(x => x <= 3)
  .map(x => x * 2)
  .reduce((acc, x) => acc + x, 0)
```

State is captured

Obvious program flow ↓

HOW DOES IT WORK?

```
[1, 2,
  .filt
  .map(
  .redu
```

How

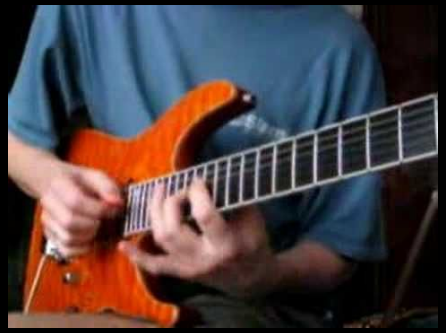
Generators?

```
[1, 2, 3, 4, 5]
  .filter(x => x <= 3)
  .map(x => x * 2)
  .reduce((acc, x) => acc + x, 0)
```

Generators?

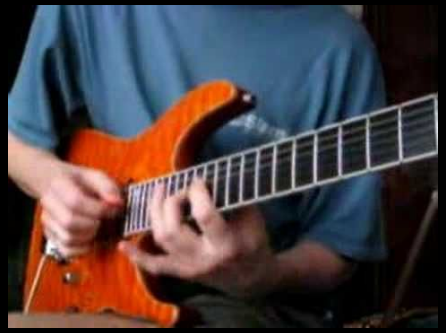
```
[1, 2, 3, 4, 5]
  .filter(x => x <= 3)
  .map(x => x * 2)
  .reduce((acc, x) => acc + x, 0)
```

Generators?



```
[1, 2, 3, 4, 5]
  .filter(x => x <= 3)
  .map(x => x * 2)
  .reduce((acc, x) => acc + x, 0)
```

Generators?



```
[1, 2, 3, 4, 5]
  .filter(x => x <= 3)
  .map(x => x * 2)
  .reduce((acc, x) => acc + x, 0)
```

We ♥ commutative/associative reducers

Example: Infinity

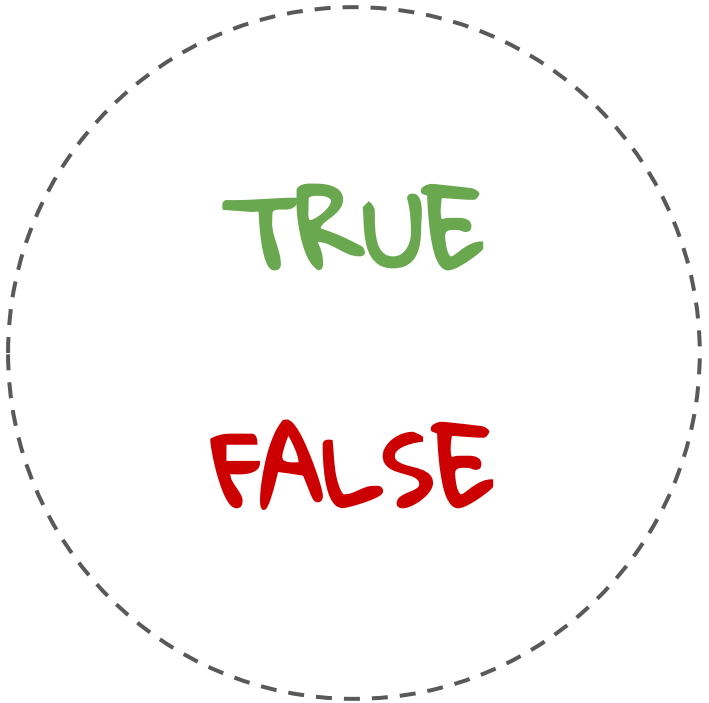
```
const inf = function *() {
    for (var i = 0; ; i++) yield i
}

const negativeInf = map(x => -x, inf())
const allEvens = filter(x => !(x % 2), inf())
```

Lazy List Operations

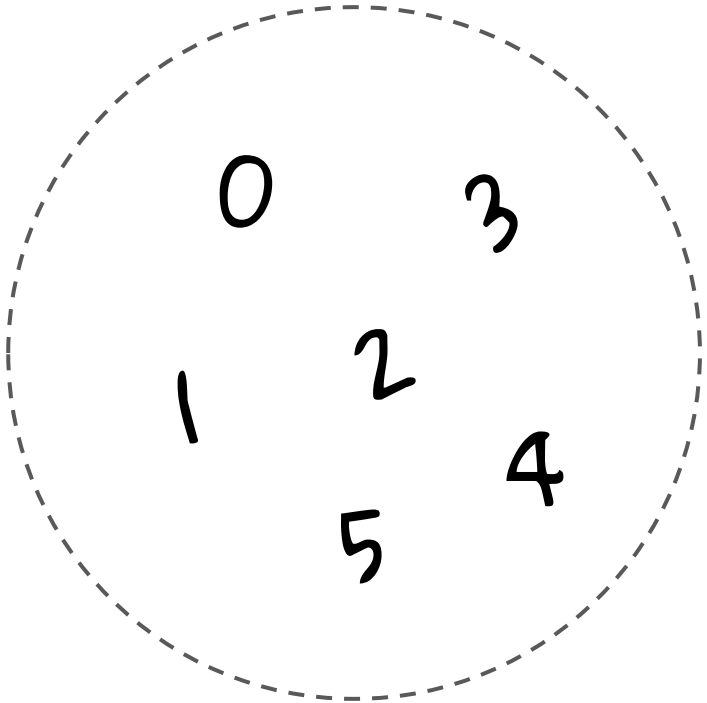
```
const map = function * (f, xs) {
  for (const x of xs) yield f (x)
}
const filter = function * (p, xs) {
  for (const x of xs) if (p (x)) yield x
}
```


KEEP PORTLAND WEIRD
Loop
Fusion

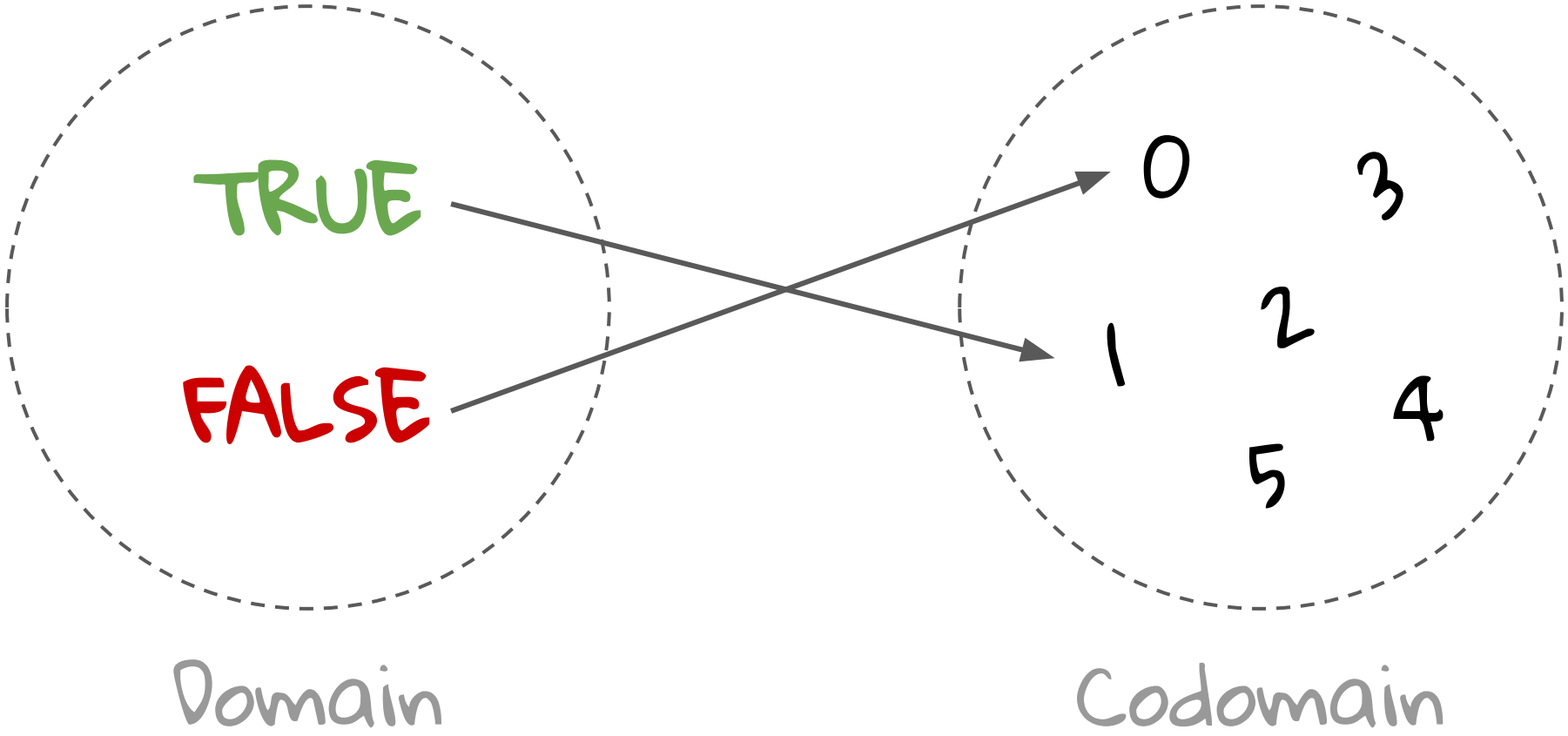

FUNCTIONS

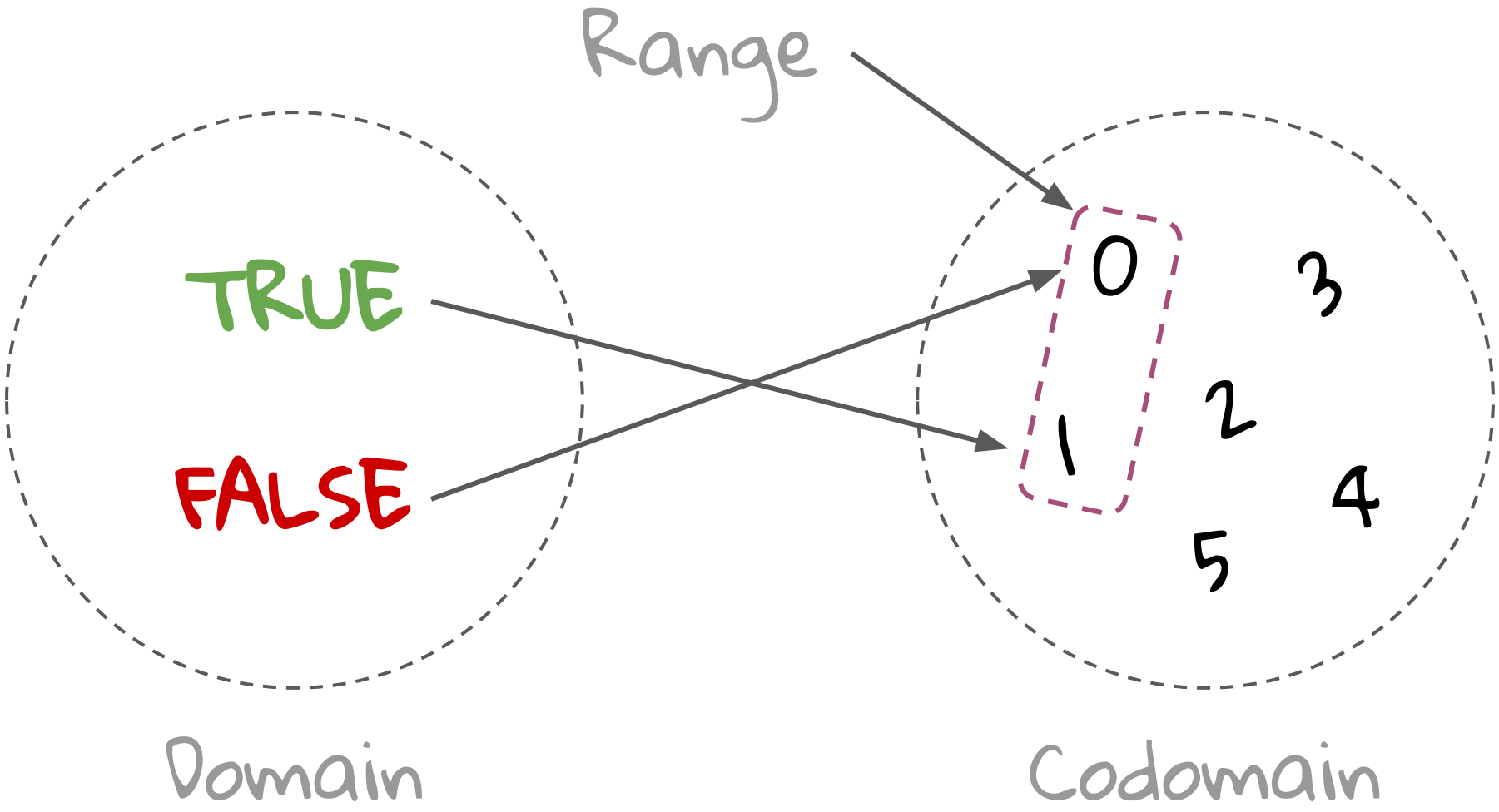


Bool



Int


TRUE
FALSE
0
3
1
2
4
5
Domain
Codomain


Range
TRUE
FALSE
0
1
2
3
4
5
Domain
Codomain

Here is a function:

```
const toBinary = b => b ? 1 : 0


toBinary(true)  == 1
toBinary(false) == 0
```

Here is a map:

```
const toBinary = { true:  1, false: 0 }
```

```
toBinary[true]  == 1
toBinary[false] == 0
```

Maps don't have side-effects

Here is a **map**:

```
const toBinary = { true:  1, false: 0 }
```

```
toBinary[true]  == 1
toBinary[false] == 0
```

Functions don't have side-effects

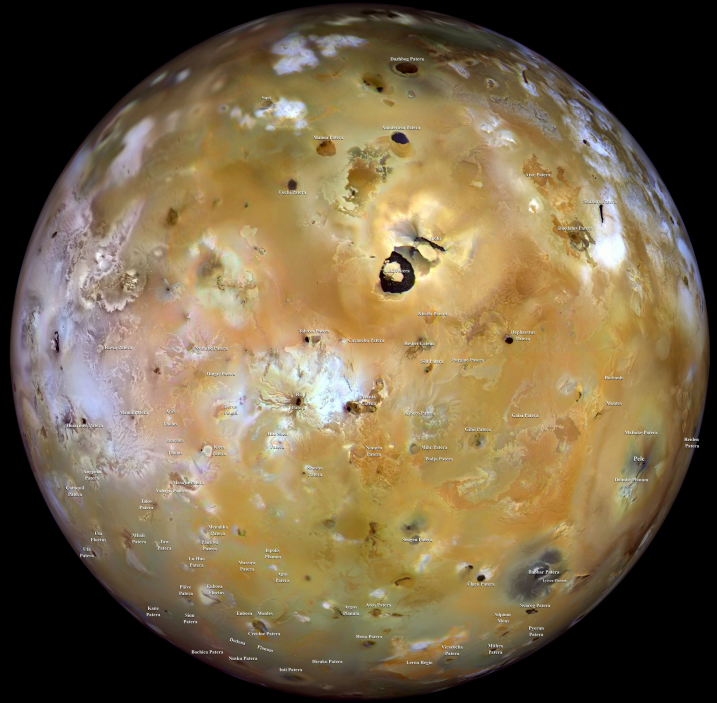
Here is a **map**:

```
const toBinary = { true:  1, false: 0 }
```

```
toBinary[true]  == 1
toBinary[false] == 0
```

Functions don't have side-effects

Io



Input/Output

Io

Input/Output

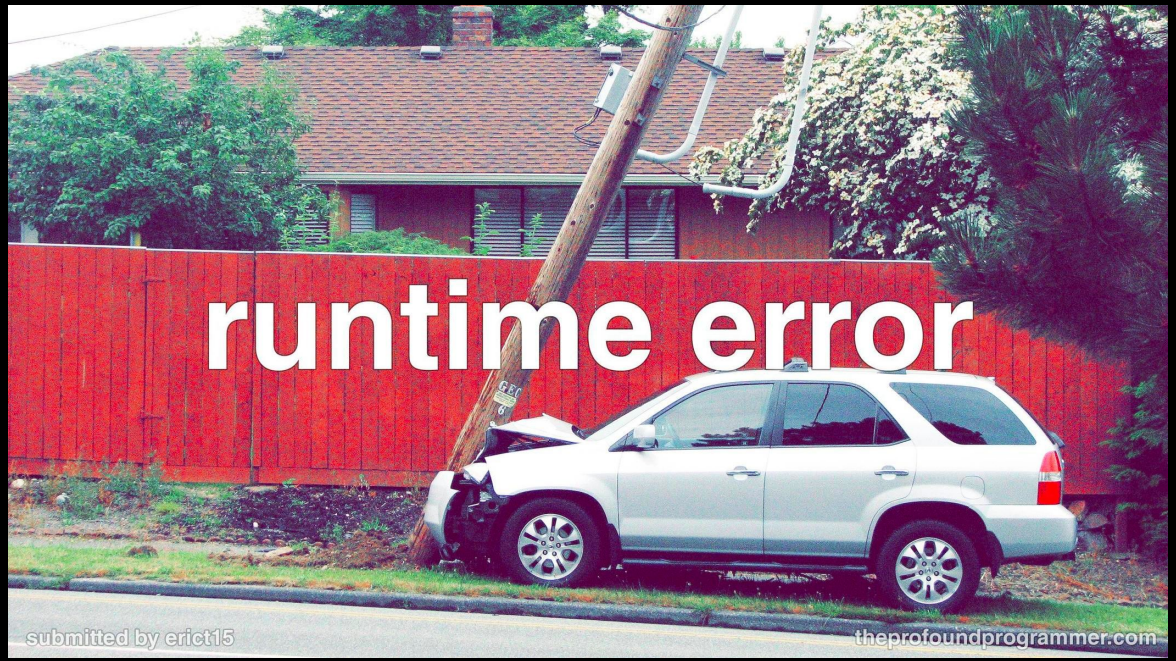
Random

Random

State Mutation

State Mutation

Exceptions


runtime error
submitted by erict15
theprofoundprogrammer.com

Exceptions

WE NEED

MONADS

WE NEED

MONADS

Another talk for
another time.

Functions are all unary

Functions are all unary

BUT inputs and outputs can be functions

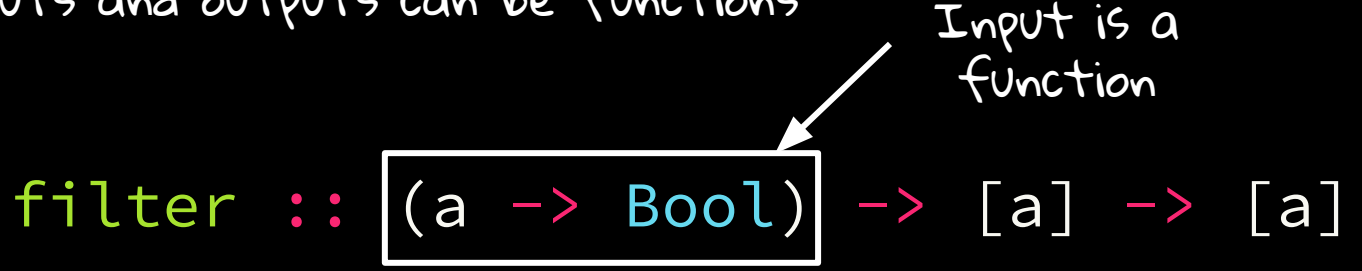
Functions are all unary

BUT inputs and outputs can be functions

```
filter :: (a -> Bool) -> [a] -> [a]
```

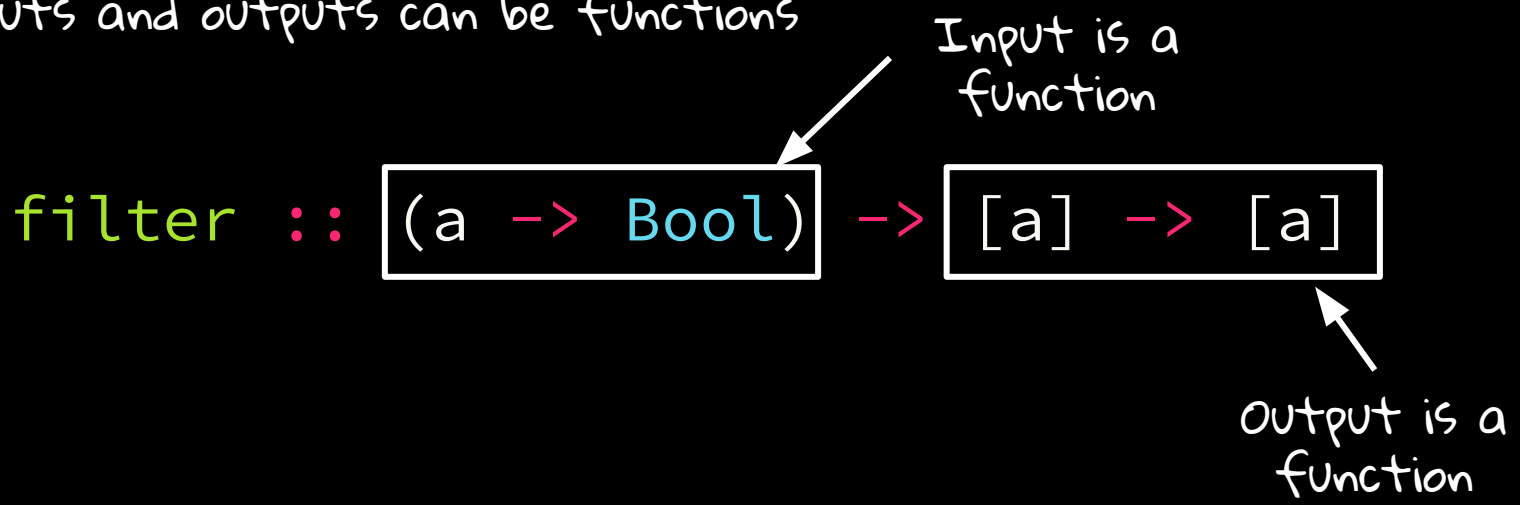
Functions are all unary

BUT inputs and outputs can be functions



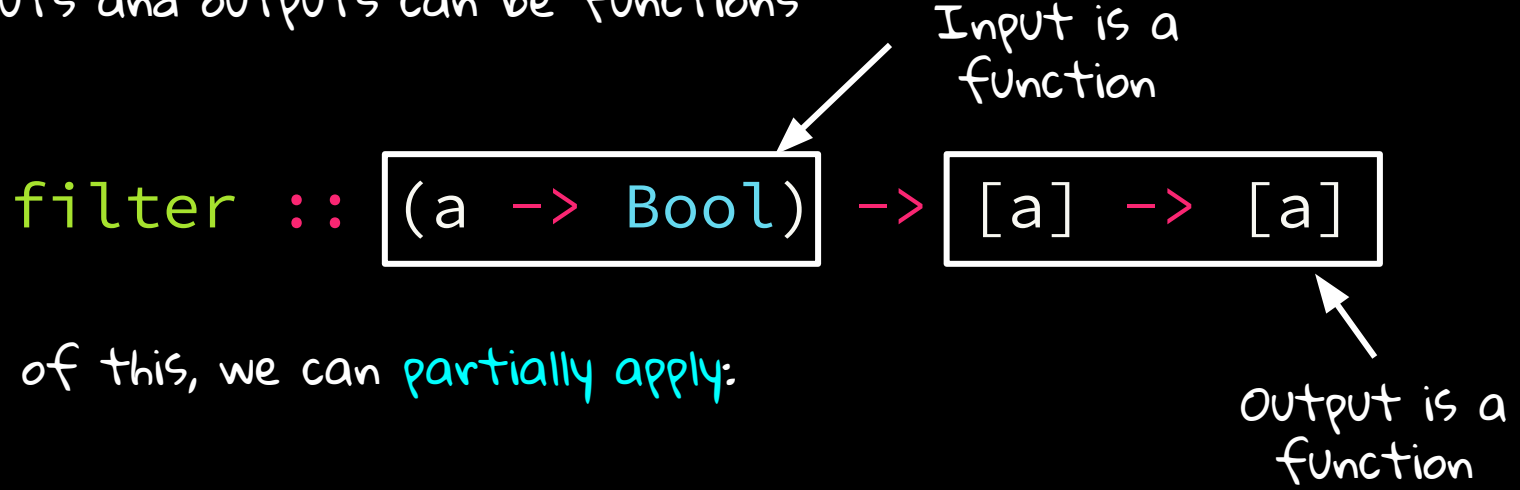
Functions are all unary

BUT inputs and outputs can be functions



Functions are all unary

BUT inputs and outputs can be functions



Because of this, we can partially apply:

```
f = filter (x => x > 3)

f ([1, 3, 4, 5]) == [4, 5]
```

Functions are all unary

BUT inputs and outputs can be functions

Input is a function

```
filter :: (a -> Bool) -> [a] -> [a]
```

Output is a function

Because of this, we can partially apply:


```
f = filter (x => x > 3)

f ([1, 3, 4, 5]) == [4, 5]
```


COMPOSER

Composition

```
const compose = (f, g) => x => f(g(x))

const uppercase = x => x.toUpperCase()
const head = xs => xs[0]

const firstCap = compose(head, uppercase)
firstCap('hello') // 'H'
```

Composition

```
const compose = (f, g) => x => f(g(x))

const uppercase = x => x.toUpperCase()
const head = xs => xs[0]

const firstCap = compose(head, uppercase)
firstCap('hello') // 'H'
```

Composition ♥ Partial Application

```
const add = x => y => x + y
const times = x => y => x * y

const incAndTwice = compose(times(2), add(1))
```

Composition ♥ Partial Application

```
const add = x => y => x + y
const times = x => y => x * y

const incAndTwice = compose(times(2), add(1))
```

Fun fact (that we already knew):

```
compose(map(f), map(g)) == map(compose(f, g))
```

Composition ♥ Partial Application

```
const add = x => y => x + y
const times = x => y => x * y

const incAndTwice = compose(times(2), add(1))
```

Fun fact (that we already knew):

```
compose(map(f), map(g)) == map(compose(f, g))
```

Super Important Business Example (TM)

```
const availablePrices = cars => {
  const available_cars = cars.filter(prop('in_stock'))

  return available_cars.map(x => {
    accounting.formatMoney(x.dollar_value)
  })
}
```

Spot the pipelines!

Thanks, @DrBoolean

Super Important Business Example™

```
const getPrice = compose(
  accounting.formatMoney,
  prop('dollar_value')
) // We have a reusable function!

const availablePrices = compose(
  map(getPrice),
  filter(prop('in_stock'))
)
```


KEEP PORTLAND WEIRD


N
VANDEVALL


N
[X]
VANDEVALL

```
const inc = x => x + 1

inc([1, 2, 3]) // Error!
```

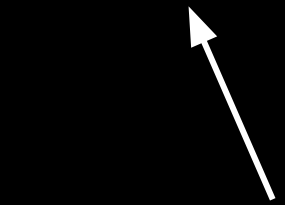
[1, 2, 3]

```
const inc = x => x + 1

map(inc)([1, 2, 3]) // [2, 3, 4]
```

```
const inc = x => x + 1

map(inc)([1, 2, 3]) // [2, 3, 4]
```

Map works without complaint for any number of values

```
const inc = x => x + 1

map(inc)([1, 2, 3]) // [2, 3, 4]
```



Map works without complaint for any number of values

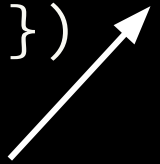
- We can do safe composition
- We can capture state
- We can do clever optimisations

Let's try

```
const MyBox = val => ({
  map: f => MyBox(f(val))
})
```

Let's try

```
const MyBox = val => ({
  map: f => MyBox(f(val))
})
```



Functors are anything with a map

Let's try

```
const MyBox = val => ({
  map: f => MyBox(f(val))
})
```

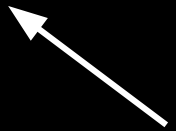
Functors are anything with a map

Safe and pretty pipelines

```
MyBox(2)
  .map(x => x + 1)
  .map(x => '' + x)
  .map(x => 'I like ' + x)
```

Lemme at it

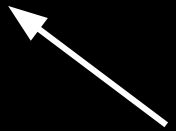
```
const MyBox = val => ({
  map: f => MyBox(f(val)),
  fold: f => f(val)
})
```



We need a fold (val) to get the value out

Lemme at it

```
const MyBox = val => ({
  map: f => MyBox(f(val)),
  fold: f => f(val)
})
```



We need a fold (`val`) to get the value out

```
fold(id, MyBox(a)) == a
```

reduce is fold for lists

So, how does this help us?


Maybe

```
const Just = val => ({
  map: f => Just(f(val)),
  fold: (_, f) => f(val)
})

const Nothing = {
  map: f => Nothing,
  fold: (n, _) => n
}
```

```
const Just = val => ({
  map: f => Just(f(val)),
  fold: (_, f) => f(val)
})

const Nothing = {
  map: f => Nothing,
  fold: (n, _) => n
}
```

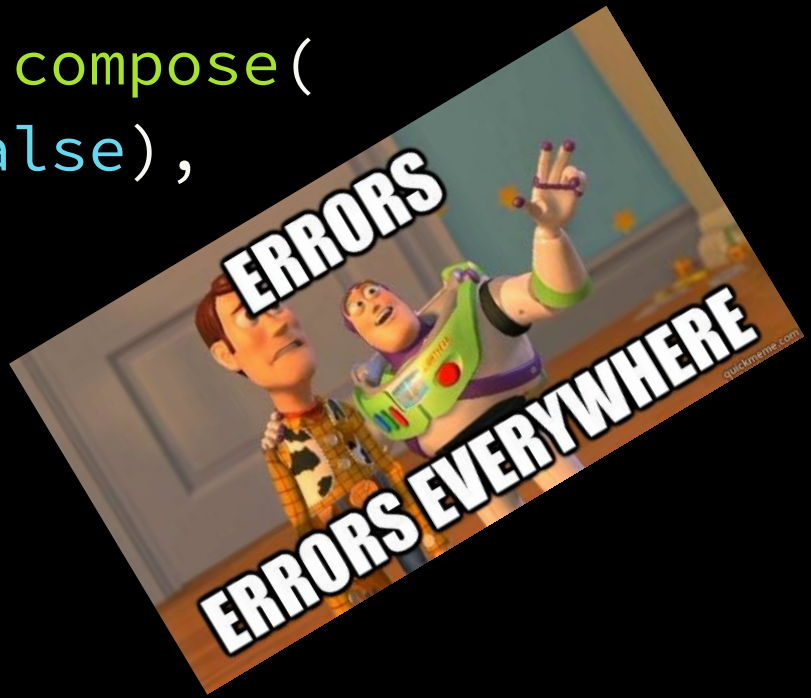
```
const Just = val => ({
  map: f => Just(f(val)),
  fold: (_, f) => f(val)
})

const Nothing = {
  map: f => Nothing,
  fold: (n, _) => n
}
```

Just like our box!

Default

```
const deleteUser = compose(
  update('live', false),
  findUser
)
```

```
const deleteUser = compose(
  update('live', false),
  findUser
)
```



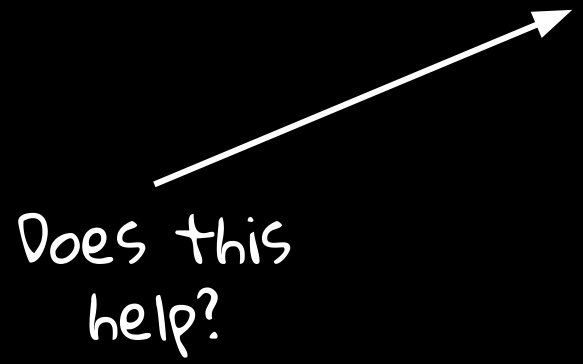
```
const deleteUser = compose(
  map(update('live', false)),
  findUser
)
```

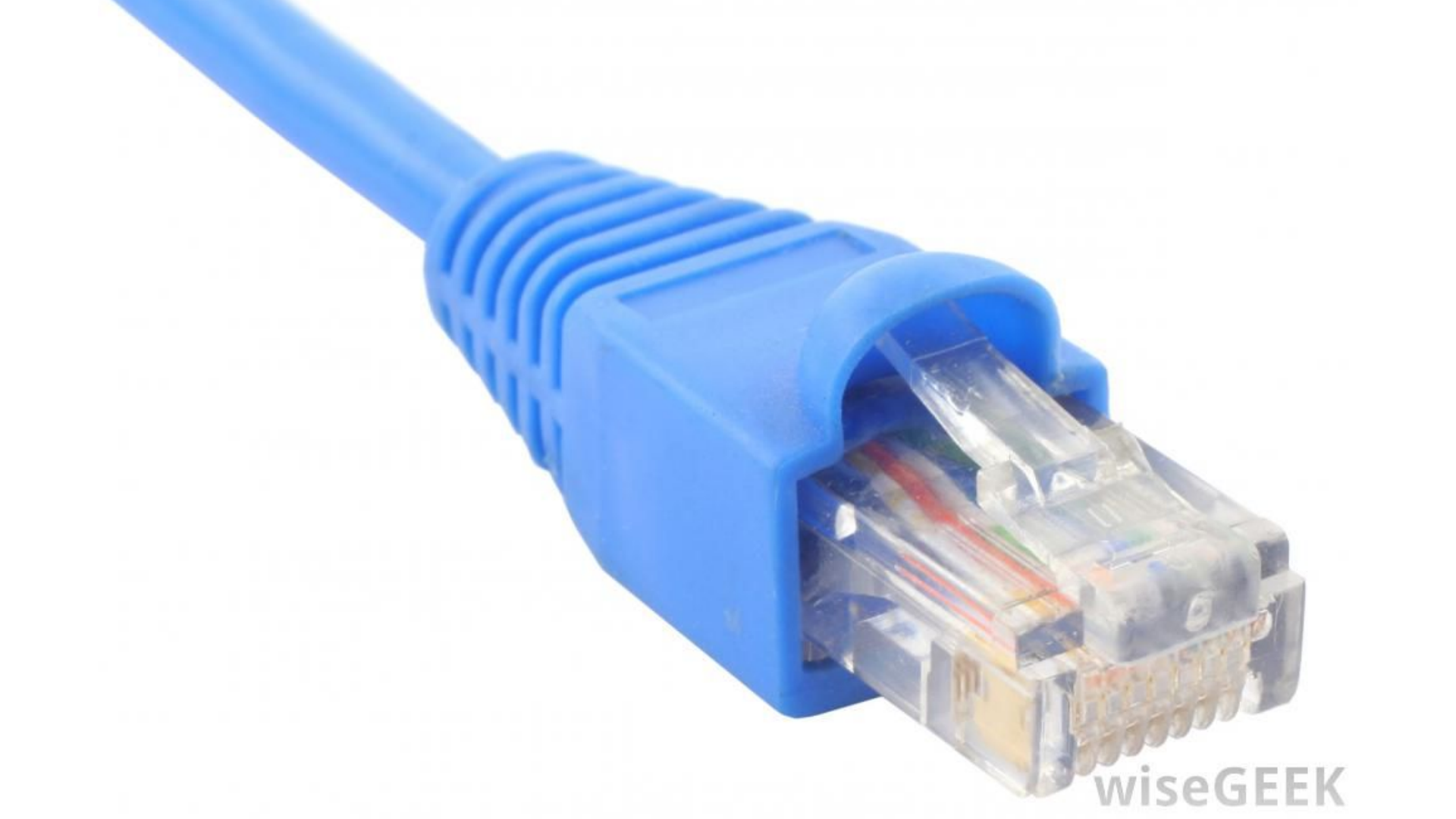
```
const deleteUser = compose(
  map(update('live', false)),
  findUser
)

findUser(validId)   // Just User
findUser(invalidId) // Nothing
```

```
  const deleteUser = compose(
    map(update('live', false)),
    findUser
  )

findUser(validId)   // Just User
findUser(invalidId) // Nothing

deleteUser(validId)   // Just User
deleteUser(invalidId) // Nothing
```

```
const getSecret = username => password =>
  username == 'test' && password === 'shh'
    ? Just('$$$') : Nothing
```

```
const getSecret = username => password =>
  username == 'test' && password === 'shh'
    ? Just('$$$') : Nothing
```




wiseGEEK


Either
wiseGEEK

```
const Right = val => ({
  map: f => Right(f(val)),
  fold: (_, g) => g(val)
})

const Left = val => ({
  map: f => Left(val),
  fold: (f, _) => f(val)
})
```

```
const Right = val => ({
  map: f => Right(f(val)),
  fold: (_, g) => g(val)
})

const Left = val => ({
  map: f => Left(val),
  fold: (f, _) => f(val)
})
```

```
const getSecret = username => password =>
  username !== 'test'
    ? Left('Incorrect username')
    : password !== 'shh'
      ? Left('Incorrect password')
      : Right('$$$')
```

Functors

- Let us capture stateful operations as pure transformations
- Let us implement error-checking (and other wizardry) at type-level
- Make for beautiful code
- Admit a window for low-level optimisations
- Come with stable and provable laws
- Allow us an easy way to modify function behaviour

TL ; RF

- Declarative code is flexible and clear.
- Functional programming is naturally declarative.
- Pure functions make for easy unit testing.
- Partial application and composition are so DRY
- Maps aren't just for arrays
- Functors can give us type-safe failure

What we didn't talk about

- Monoids
- Applicatives
- Monads
- Combinators (other than compose)
- Other folds (e.g. corecursion, catamorphisms)
- Typing (properly)
- Bifunctors (AKA completely escaping if/else)

Questions?

@whoaitstom

GitHub: i-am-tom

People you should follow:

Brian Lonsdorf

Bodil Stokke

Quildreen Motta

Fantasy Land