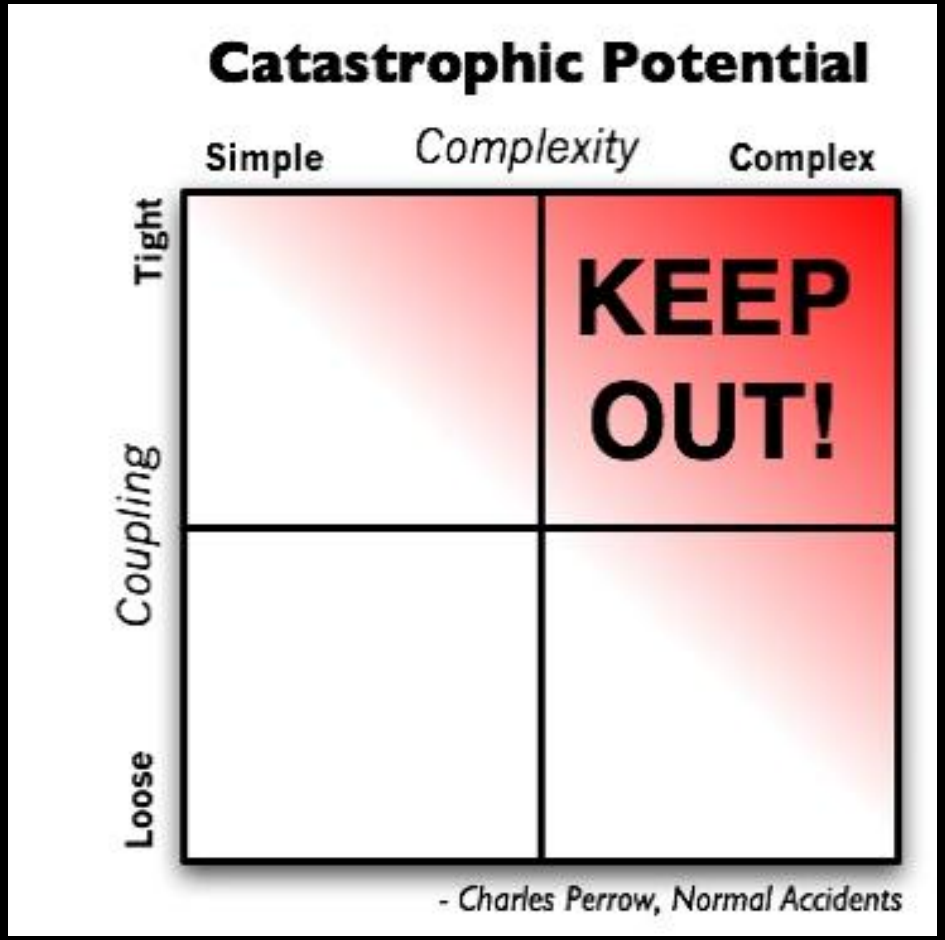
myob

Functional Programming In javascript

Ray Dai


Catastrophic Potential
Simple
Complexity
Complex
Tight
Coupling
Loose
KEEP OUT!
- Charles Perrow, Normal Accidents

Single source of truth,
State is read-only,
Changes are made with pure functions.

What is functional programming?

AWESOME

programming paradigm / style

building the structure and elements

treats computation as the evaluation of mathematical functions

avoids changing-state and mutable data.

declarative programming paradigm

programming is done with expressions or declarations instead of statements.

https://en.wikipedia.org/wiki/Functional_programming

```
// associative
add(add(x, y), z) == add(x, add(y, z))

// commutative
add(x, y) == add(y, x)

// identity
add(x, 0) == x

// distributive
add(multiply(x, y), multiply(x, z)) == multiply(x, add(y,z))
```

Javascript

Scheme

Self

FP

OO

Pure functions

Curry

Composition

Pointfree

Pure functions

A pure function is a function that, given the same input, will always return the same output and does not have any observable side effect.

Mutation, DB, IO, etc.,

Array.prototype.splice()

SEE ALSO

Standard built-in objects

The **`splice()`** method changes the content of an array by removing existing elements and/or adding new elements.

Array.prototype.slice()

SEE ALSO

Standard built-in objects

The **`slice()`** method returns a shallow copy of a portion of an array into a new array object.

https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/splice
https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/slice

```
var xs = [1,2,3,4,5]
```

```
// impure
xs.splice(0,3)
//=> [1,2,3]

xs.splice(0,3)
//=> [4,5]

xs.splice(0,3)
//=> []
```

```
// pure
xs.slice(0,3)
//=> [1,2,3]

xs.slice(0,3)
//=> [1,2,3]

xs.slice(0,3)
//=> [1,2,3]
```

```
// impure
var minimum = 21;

var checkAge = function(age) {
  return age >= minimum;
}
checkAge(21)
//=> true
minimum = 22;

checkAge(21)
//=> false
```

```
// pure
var checkAge = function(age) {
  var minimum = 21;
  return age >= minimum;
}
checkAge(21)
//=> true
minimum = 22;

checkAge(21)
//=> true
```

```
// impure
var greeting = function(name) {
  console.log("hi, " + name +
"!")
}
```

```
// pure
var greeting = function(name) {
  return "hi, " + name + "!";
}

console.log(greeting("Jonas"))
```

Let’s play pure or impure

Let’s play pure or impure

```
var birthday = function(user) {
  user.age += 1;
  return user;
}
```

Let’s play pure or impure

```
var shout = function(word) {
  return word.toUpperCase().concat("!");
}
```

Let's play pure or impure

```
var headerText = function(header_selector) {
  return $(header_selector).text();
}
```

Let's play pure or impure

```
var parseQuery = function() {
  return location.search
       .substring(1)
       .split('&').map(function(x){
       return x.split('=')
     })
   }
```

Let's play pure or impure

```
var parseQueryString = function(queryString) {
    var params = {}, queries, temp, i, l;
    queries = queryString.split("&");
    for ( i = 0, l = queries.length; i < l; i++ ) {
        temp = queries[i].split('=');
        params[temp[0]] = temp[1];
    }
    return params;
};
```

Let's play pure or impure

```
var httpGet = function(url, params){
  return function() { return $.getJSON(url, params); }
};
```

Set theoretically

Every function is

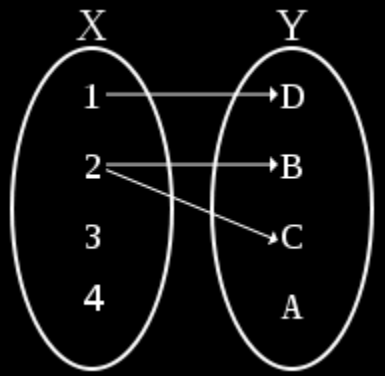
a *single-valued* collection of pairs

THIS

```
...
(-2, -1)
(0 , 0 )
(2 , 1 )
(4 , 2 )
(8 , 4 )
...
```

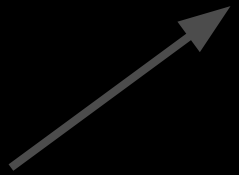
NOT THIS

```
(1, D)
(2, B)
(2, C)
```



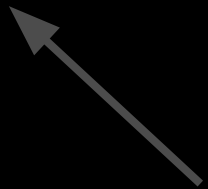
One input, one output

Input	Output
1	2
2	4
3	6



Domain

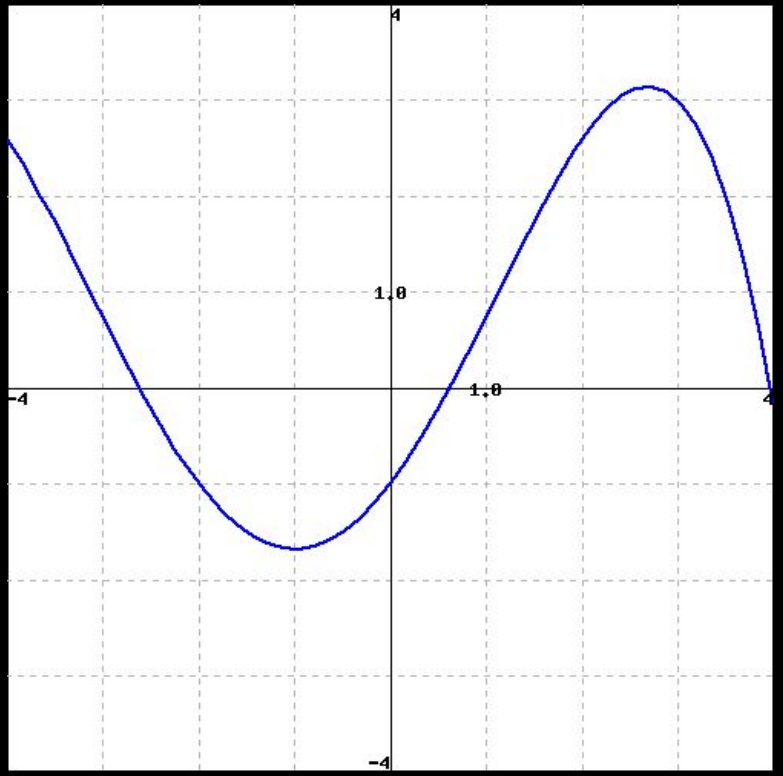
… 1,2,3 ...



Range

… 2,4,6...

One input, one output



One input, one output

```
var toLowerCase = {"A":"a", "B": "b", "C": "c", "D": "d", "E": "e"}
toLowerCase["C"]
//=> "c"
var isPrime = {1: false, 2: true, 3: true, 4: false, 5: true, 6: false}
isPrime[3]
//=> true
```

Cacheable / Memorizable

```
var greeting = memoize(function(name) {
  return "hi, " + name + "!";
})
greeting("Jonas")
//=> Hi, Jonas!
greeting("Jonas") //hit the cache
//=> Hi, Jonas!
```

Cacheable

Testable

Parallel code

Reasonable

Curried Function

A function that will return a new function until it receives all its arguments

Currying

```
//+ add :: Number -> Number -> Number
var add = curry(function(x, y) {
  return x + y
})
```

https://en.wikipedia.org/wiki/Hindley-Milner_type_system

Currying

```
add
//=> function(x,y) { return x + y }

add(2,3)
//=> 5

add(2)
//=> function(y) { return 2 + y }
```

```
01 function curry(fn) {
02   return function() {
03     if (fn.length > arguments.length) {
04       var slice = Array.prototype.slice;
05       var args = slice.apply(arguments);
06       return function() {
07         return fn.apply(
08           null, args.concat(slice.apply(arguments)));
09       };
10     }
11     return fn.apply(null, arguments);
12   };
13 }
```

```
var curry = (f, ...args) =>
    (f.length <= args.length) ?

    f.call(this, ...args) :

    (...more) => curry.call(this, f, ...args, ...more);
//# sourceURL=curry
```

Currying

```
//+ get :: String -> {String: a} -> a
var get = curry(function(prop, obj) {
  return obj[prop];
})

var user = {id: 32, name: "Gary", email: "gary@newman.com"}
get('email', user)
//=> gary@newman.com

//+ email :: {String: a} -> a
var email = get('email')
//=> function(obj){ return obj[‘email’] }
email(user)
//=> gary@newman.com
```

Currying

```
//+ modulo :: Number -> Number -> Number
var modulo = curry(function(divisor, dividend) {
  return dividend % divisor;
})
//+ isOdd :: Number -> Number
var isOdd = modulo(2)
//=> function(dividend){ return dividend % 2 }
isOdd(2)  //=> 0
isOdd(3)  //=> 1
```

Currying

```
//+ filter :: (a -> Bool) -> [a] -> [a]
var filter = curry(function(f, xs) {
  return xs.filter(f);
})
//+ odds :: [a] -> [a]
var odds = filter(isOdd)

odds([1,2,3])
//=> [1, 3]
```

Currying

```
//+ odds :: [a] -> [a]
var odds = function(xs) {
  return filter(function(x){ return isOdd(x) }, xs)
}
//+ oddz :: [a] -> [a]
var oddz = filter(isOdd)

Odds == oddz
```

Currying

```
var emails = map(email)

var users = [beyonce, martin, gary]

emails(users)
//=> ["beyonce@knowles.org", "martin@lawrence.net",
"gary@newman.com"]
```

Currying

```
//+ goodArticles :: [Article] -> [Article]
var goodArticles = function(articles) {
  return _.filter(articles, function(article){
    return _.isDefined(article)
  })
}
//+ goodArticles :: [Article] -> [Article]
var goodArticles = filter(isDefined)
```

Currying

```
//+ getChildren :: DOM -> [DOM]
var getChildren = function(el) {
  return el.childNodes
}
//+ getALLChildren :: [DOM] -> [[DOM]]
var getAllChildren = function(els) {
  return _.map(els, function(el) {
    return getChildren(el)
  })
}
```

Currying

```
var getChildren = get('childNodes') // waiting
for el
var getAllChildren = map(getChildren) // waiting
for els
```

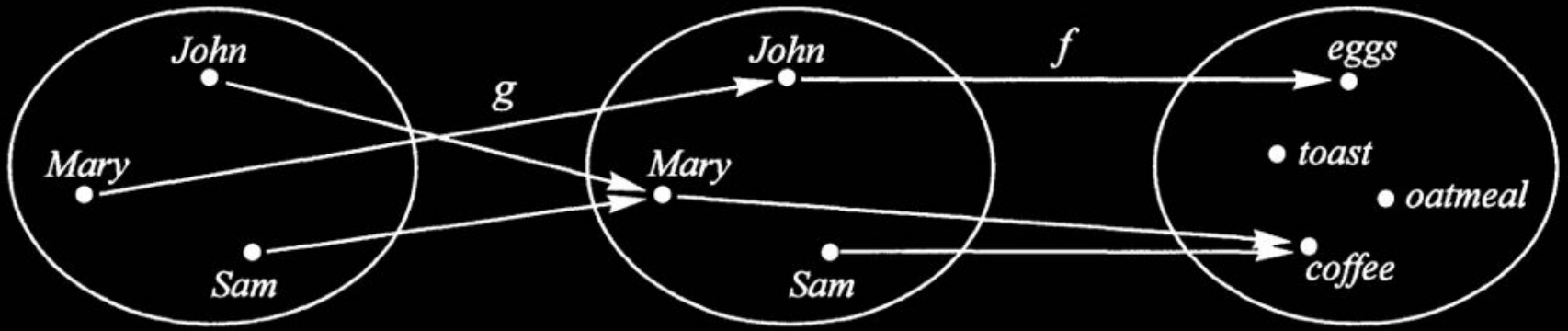
Composition

Function composition is applying one function to the results of another

Composition

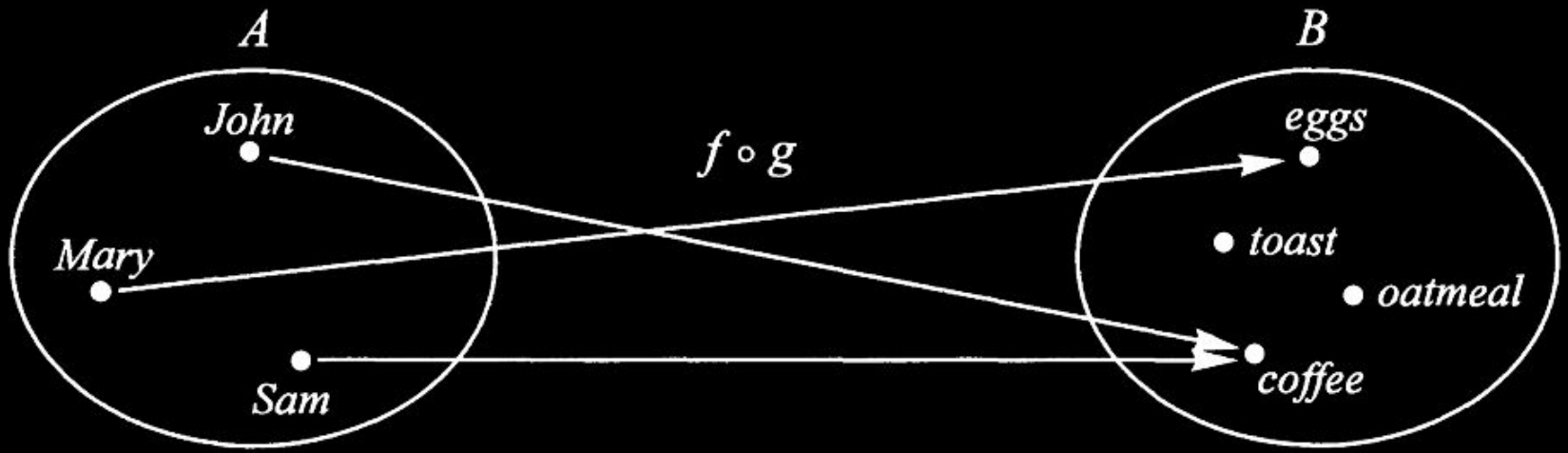
```
//+ compose :: (b -> c) -> (a -> b) -> a -> c
var compose = curry(function(f, g, x) {
  return f(g(x))
})

var compose = (...fns) => fns.reduce(
(f, g) => (...args) => f(g.apply(null, args)))
//# sourceURL=compose
```

Composition



$$A \xrightarrow{g} A \xrightarrow{f} B$$

Composition



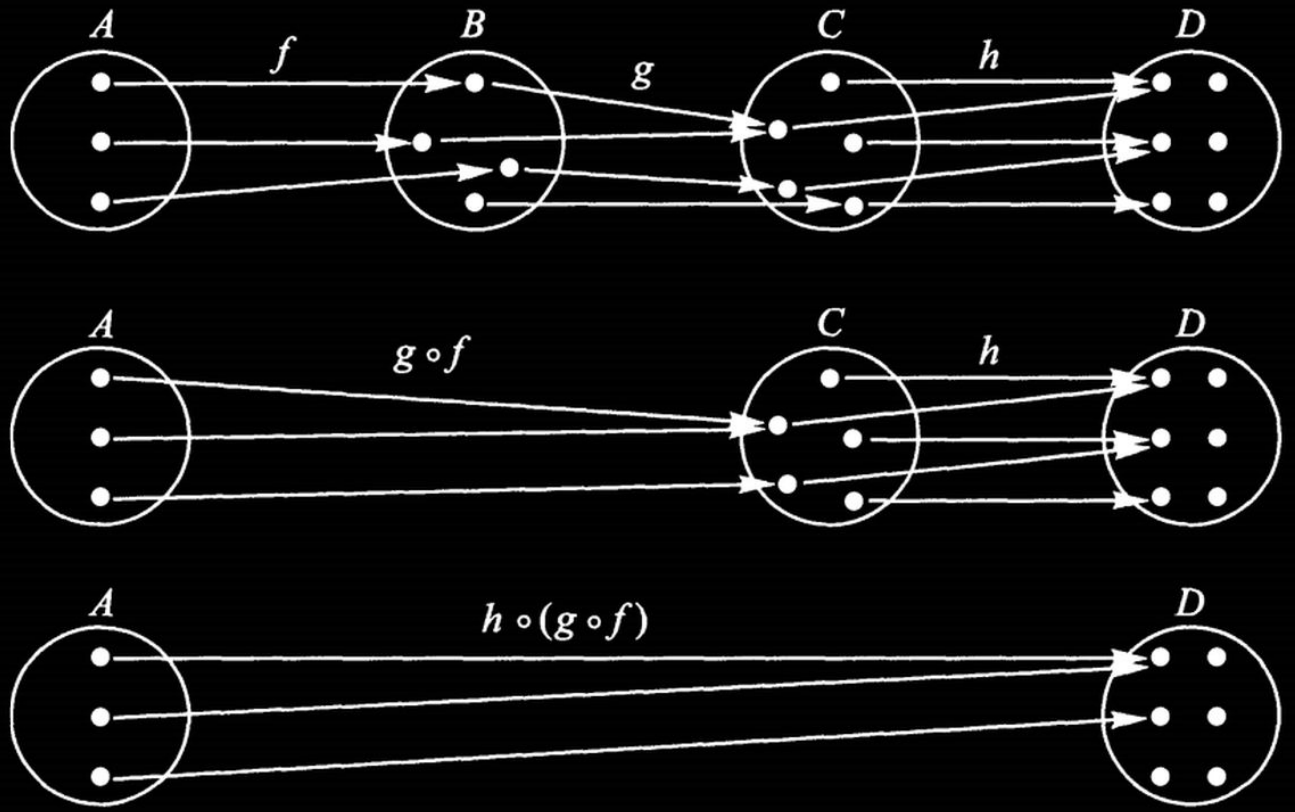
$$A \xrightarrow{f \circ g} B$$

Composition

```
var g = function(x) {
    return x.length;
}
var f = function(x) {
    return x === 3;
}
var isLetter3Word = compose(f, g);
```

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

Composition



Composition

```
//+ wordCount :: String -> Number
var wordCount = function(sentence) {
  return length(split(' ', sentence))
}
//+ length :: Array -> Number
var length = function(xs) { return xs.length }
//+ wordCount :: String -> Number
var wordCount = compose(length, split(' '))
wordCount("I am a sentence with seven words") //=> 7
```

Composition

```
//+ head :: [a] -> a
var head = function(x) { return x[0] }
//+ reverse :: [a] -> [a]
var reverse = reduce(function(acc, x) {
    return [x].concat(acc);
} ,[])
//+ last :: [a] -> a
var last = compose(head, reverse)
last(['jumpkick', 'roundhouse', 'uppercut'])
//=> 'uppercut'
```

Composition

```
'Y' <- 'y' <- 'Functional Factory'

compose(toUpperCase, last)('Functional Factory')
```

Composition

```
// associativity
compose(f, compose(g, h)) ==
compose(compose(f, g), h)
```

Composition

```
compose(toUpperCase, compose(head, reverse))

// or
compose(compose(toUpperCase, head), reverse)
```

Composition

```
var toUpperCase = function(x) {
    return x.toUpperCase();
}
var exclaim = function(x) {
    return x + '!';
}
```

Composition

```
//+ lastUpper :: [String] -> String
var lastUpper = compose(toUpperCase, head, reverse)
lastUpper(['jumpkick', 'roundhouse', 'uppercut'])
//=> 'UPPERCUT'
//+ lastUpper :: [String] -> String
var loudLastUpper = compose(exclaim, toUpperCase, head,
reverse)
loudLastUpper(['jumpkick', 'roundhouse', 'uppercut'])
//=> 'UPPERCUT!'
```

Composition

```
//+ lastUpper :: [String] -> String
var lastUpper = compose(toUpperCase, head, reverse)
lastUpper(['jumpkick', 'roundhouse', 'uppercut'])
//=> 'UPPERCUT'
//+ lastUpper :: [String] -> String
var loudLastUpper = compose(exclaim, toUpperCase, head,
reverse)
var loudLastUpper2 = compose(exclaim, lastUpper)
loudLastUpper(['jumpkick', 'roundhouse', 'uppercut'])
//=> 'UPPERCUT!'
```

Composition

```
var id = function(x) {
    return x;
}

// identity
compose(f, id) = compose(id, f) = f;
```

Composition

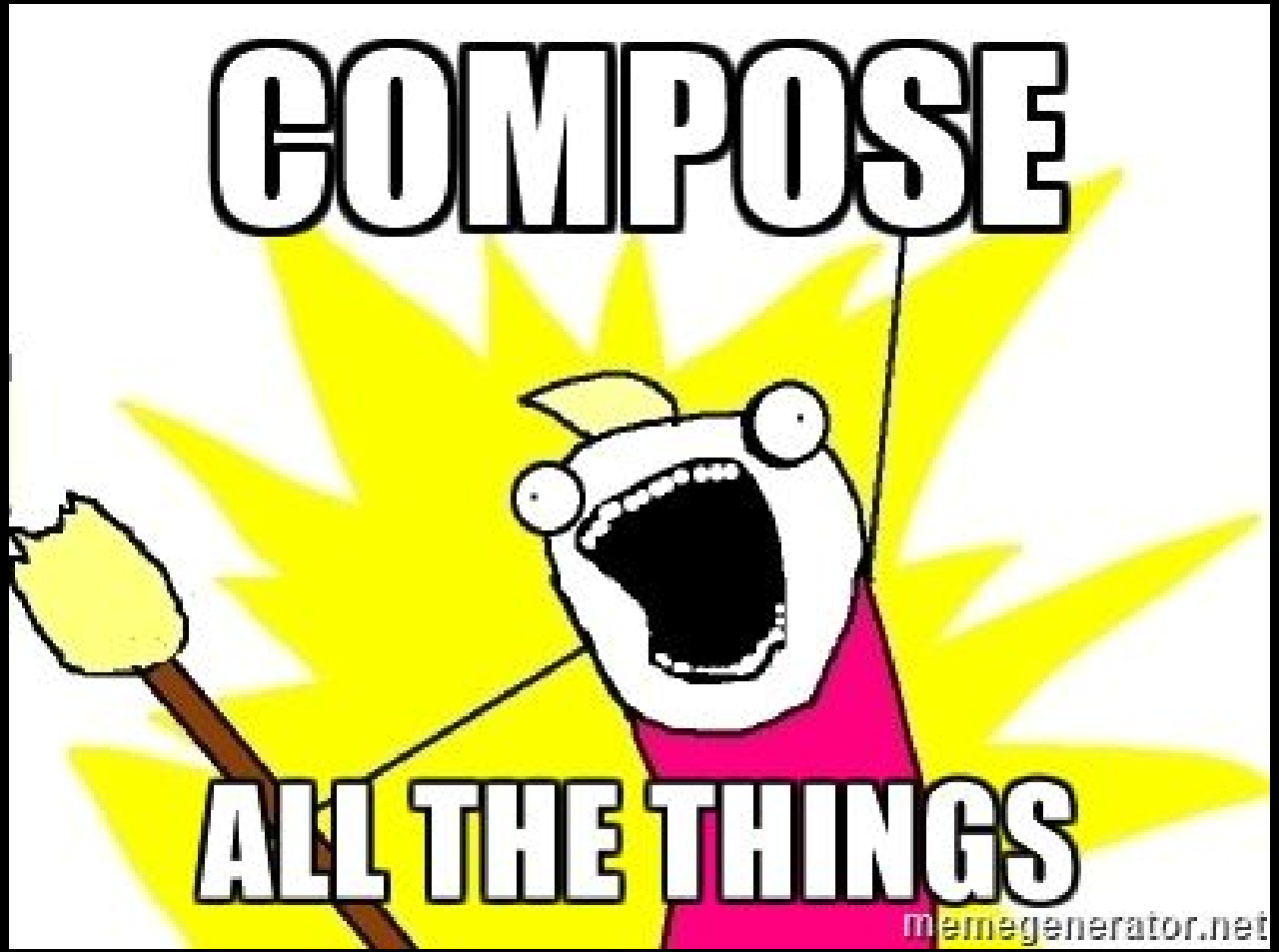
```
? <- map(['Factory', 'Function']) <- ['Factory', 'Function'] <- ['Function', 'Factory']

        compose(toUpperCase, map, reverse)(['Function', 'Factory'])
```

Composition

```
['FACTORY', 'FUNCTION'] <- ['Factory', 'Function'] <- ['Function', 'Factory']

   compose(map(toUpperCase), reverse)(['Function', 'Factory'])
```

Composition

```
// distributive
add(multiply(x, y), multiply(x, z)) == multiply(x, add(y,z))

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


COMPOSE
ALL THE THINGS
memegenerator.net

Pointfree

```
httpGet('/post/2', function(json){
  renderPost(json)
})

httpGet('/post/2', function(json, err){
  renderPost(json, err)
})

httpGet('/post/2', renderPost)
```

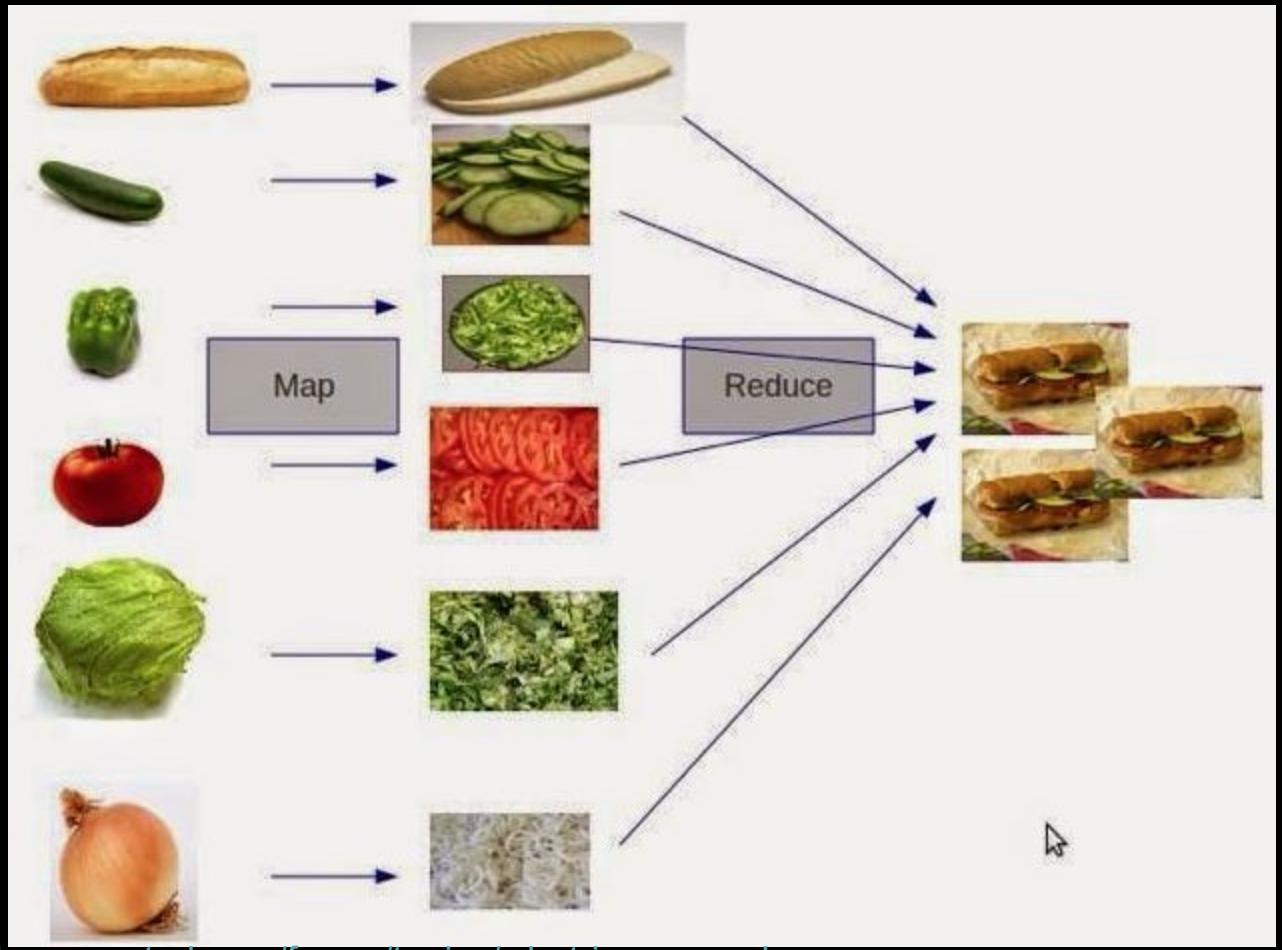
Pointfree

```
//+ clientApp :: Params -> Html
var clientApp = compose(render, doThings, httpGet('/posts'))

//+ serverApp :: Query -> JSON
var serverApp = compose(sendJSON, doThings, Db.all('posts'))

//+ shellApp :: _ -> String
var shellApp = compose(display, doThings, prompt("what's up?"))
```

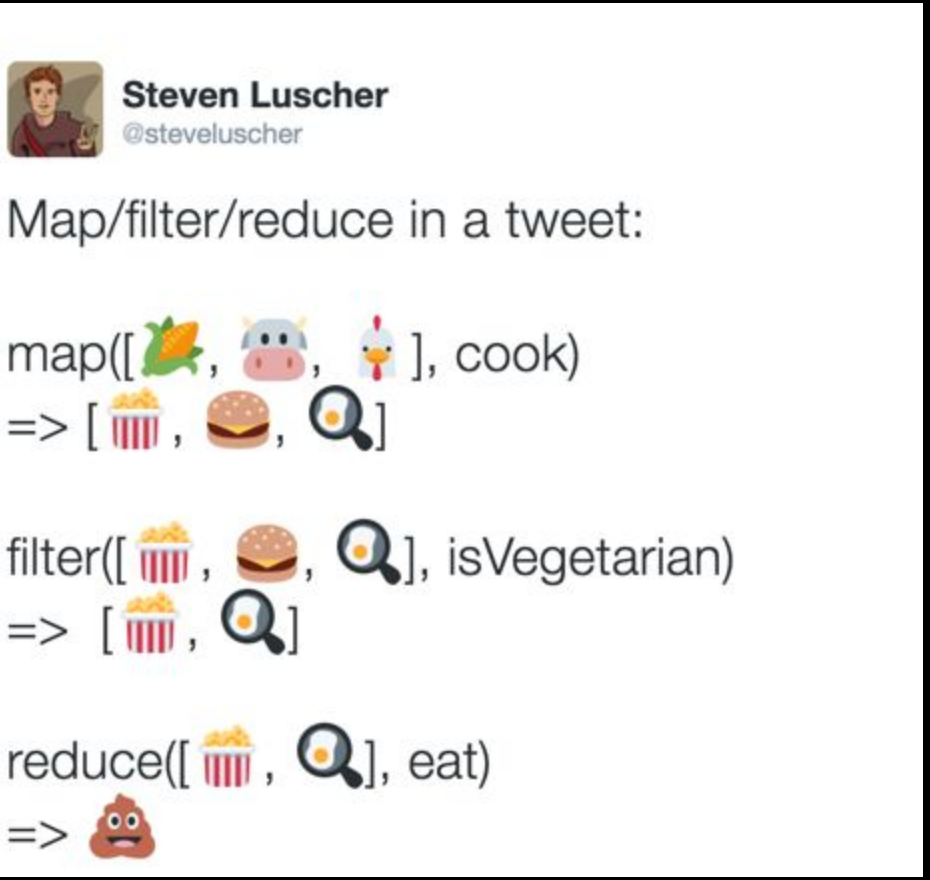
recognize most loops are one of

reduce

filter map



http://www.datasciencecentral.com/forum/topics/what-is-map-reduce



https://twitter.com/steveluscher/status/741089564329054208

```
// associative
add(add(1, 2), 4) == add(1, add(2, 4))

// commutative
add(4, 1) == add(1, 4)

// identity
add(n, 0) == n

// distributive
multiply(2, add(3,4)) == add(multiply(2, 3), multiply(2, 4))
```

Libraries

- ramdajs / ramda
- lodash / lodash/fp
- baconjs / bacon.js
- fantasyland / fantasy-io
- DrBoolean / pointfree-fantasy
- folktale / data.either, data.future

Take away

1. Parameterize all the things
2. Arguments can be provided over time, not just all at once.
3. Try not to modify outside things. Inputs etc.
4. Avoid mutability.
5. Compose without “glue” variables.

github.com/
MYOB-Technology/
AD-OnlineTax-Form-Operations

Nulls

Callbacks

Errors

Side effects

Q & A