Welcome

Joe Nelson
@begriffs

Brian Lonsdorf
@drboolean

Separation and Recognition

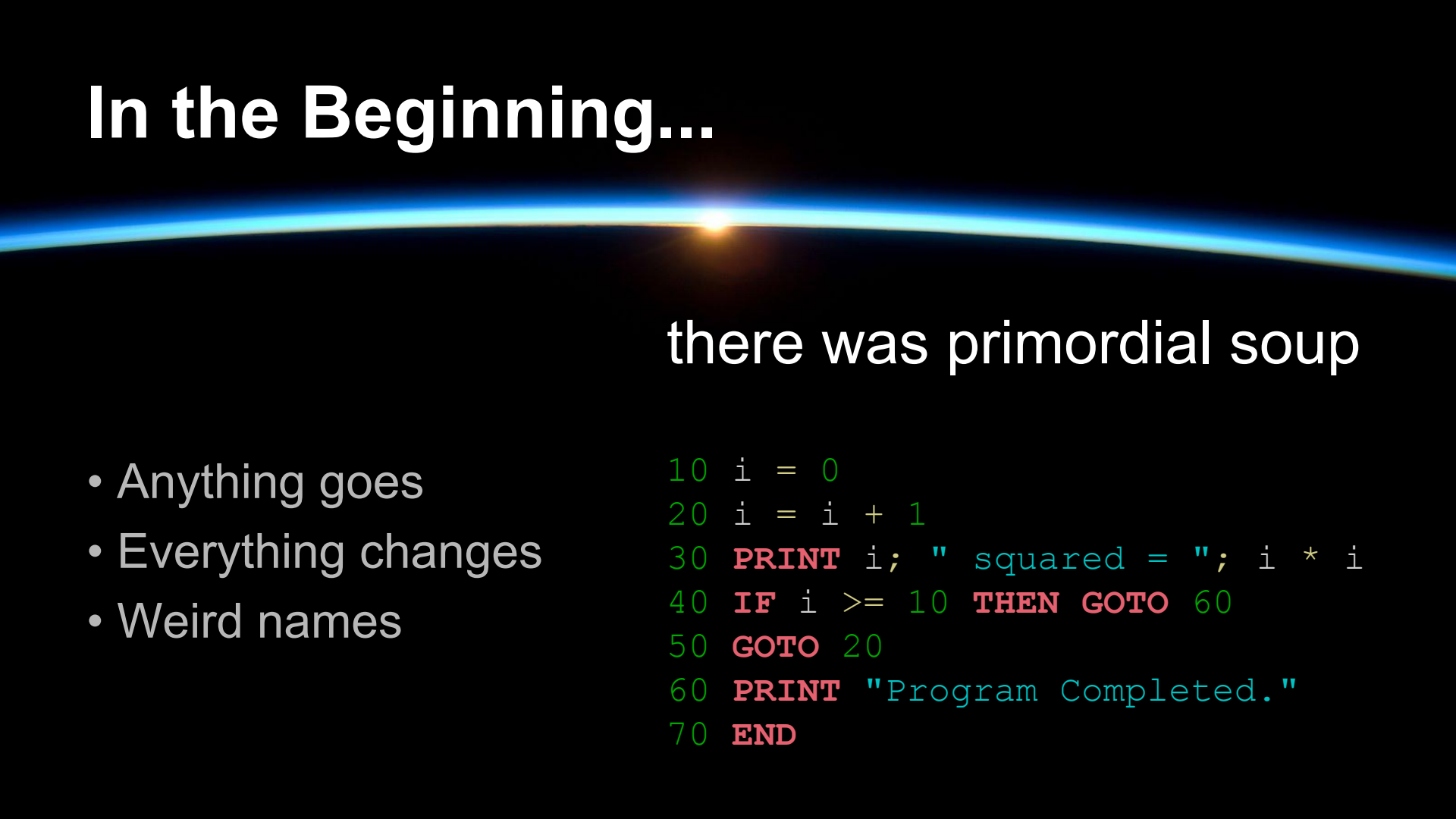
The Soul of Functional Programming

I: The Silence

III: The Demo

II: The Voyage

In the Beginning...



there was primordial soup

- Anything goes
- Everything changes
- Weird names

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

Discipline Wins

Exercising restraint while coding feels weird at first, but it’s worth it.

“The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program.”

The Symptoms

- Custom names
- Looping patterns
- Glue code
- Side effects


file
line
modified
number
content
repo
files
github
violations
name
request
full
data
head
pull
config
id
@payload
filename
payload
patch
position
validates
token
message
initialize
team
contents
additions
belongs
include
many
true
nil
memberships
lines
@github
sha
presence
api
changed
generate
admin
comment
ActiveRecord::Base
LINE
active
@modified
ActiveModel::ForbiddenAttributesProtection
attributes
removed
@custom
create
teams
@config
commit
IGNORED
Struct.new
GithubApi.new
configuration
username
ModifiedFile.new
file.modified
lines.detect
RANGE
body
self
investigate
custom
parsed
uuid
violation.line
line.line
map
violations.select
action
violation
repository
@payload.action
INFORMATION
synchronize
@pull
opened
else
style
relevant
CONFIG
file.filename
violations.any
MODIFIED
REMOVED
guide
parse
remember

Omit Needless Names

with separations
and recognitions

separate inputs from environment

Secret input: time

```
function daysThisMonth() {
  var date  = new Date()
    , y     = date.getFullYear()
    , m     = date.getMonth()
    , start = new Date(y, m, 1)
    , end   = new Date(y, m + 1, 1);
  return (end - start) / (1000 * 60 * 60 * 24);
}
```

Always works the same

```
function daysInMonth(y, m) {
  var start = new Date(y, m - 1, 1)
    , end   = new Date(y, m, 1);
  return (end - start) / (1000 * 60 * 60 * 24);
}
```

jsbin.com/yoyip

separate mutation from calculation

Teaser updates DOM

```
function teaser(size, elt) {
  setText(elt, slice(0, size, text(elt)));
}

map(teaser(50), all('p'));
```

Mutation

Merely calculates

```
var teaser = slice(0);
map(compose(setText, teaser(50), text), all('p'));
```

Mutation

recognize pure functions

Functions that don’t change anything are called “pure.”

Their purity makes them

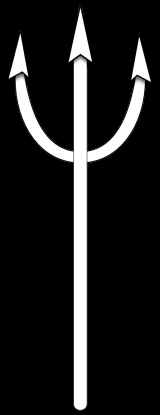
- testable
- portable
- memoizable
- parallelizable

Let’s play pure or impure

```
function getQueryVariable(variable) {
    var query = window.location.search.substring(1);
    var vars = query.split('&');
    for (var i = 0; i < vars.length; i++) {
        var pair = vars[i].split('=');
        if (decodeURIComponent(pair[0]) == variable) {
            return decodeURIComponent(pair[1]);
        }
    }
}
```

Let’s play pure or impure

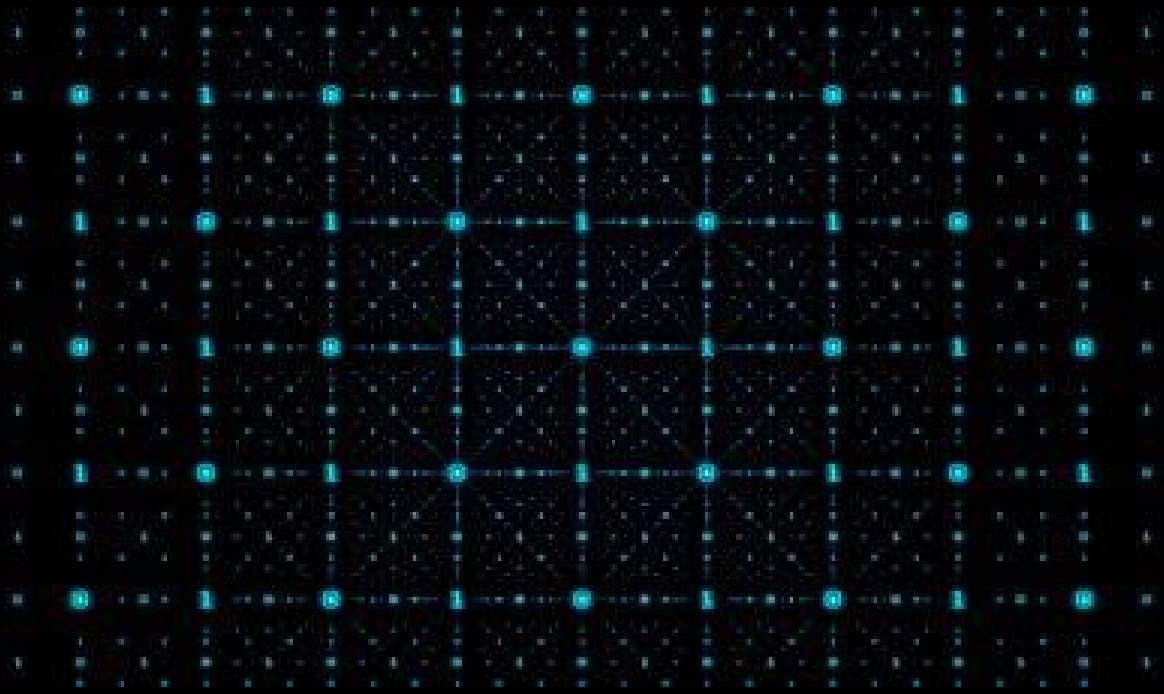
```
function random(m_w, m_z) {
  m_z = 36969 * (m_z & 65535) + (m_z >> 16);
  m_w = 18000 * (m_w & 65535) + (m_w >> 16);
  return (m_z << 16) + m_w;
}
```



Let’s play pure or impure

```
function chattyAdd(a, b, console) {
  var c = a+b;
  console.log(a, '+', b, '=', c);
  return c;
}
```

separate functions from rules



functions are nouns

separate functions from rules

Intension

```
lX8 = (lX & 0x80000000);
lY8 = (lY & 0x80000000);
lX4 = (lX & 0x40000000);
lY4 = (lY & 0x40000000);
lResult = (lX & 0x3FFFFFFF) + (lY & 0x3FFFFFFF);
if (lX4 & lY4) {
  return (lResult ^ 0x80000000 ^ lX8 ^ lY8);
}
if (lX4 | lY4) {
  if (lResult & 0x40000000) {
    return (lResult ^ 0xC0000000 ^ lX8 ^ lY8);
  } else {
```

Extension

strings	8cab5c3f5106aa48299dec51e77921ad
are	eacbe4d1b2dee530eee7460477877c4d
strung	1c6509708e542c4f27d4437b15e331cd

Set theoretically

Every function is

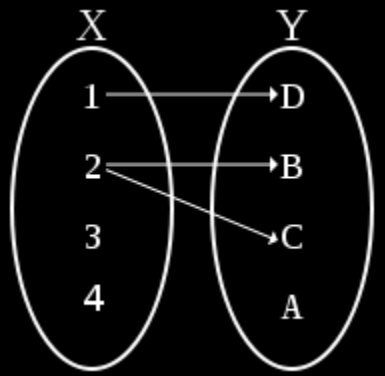
a *single-valued* collection of pairs

THIS

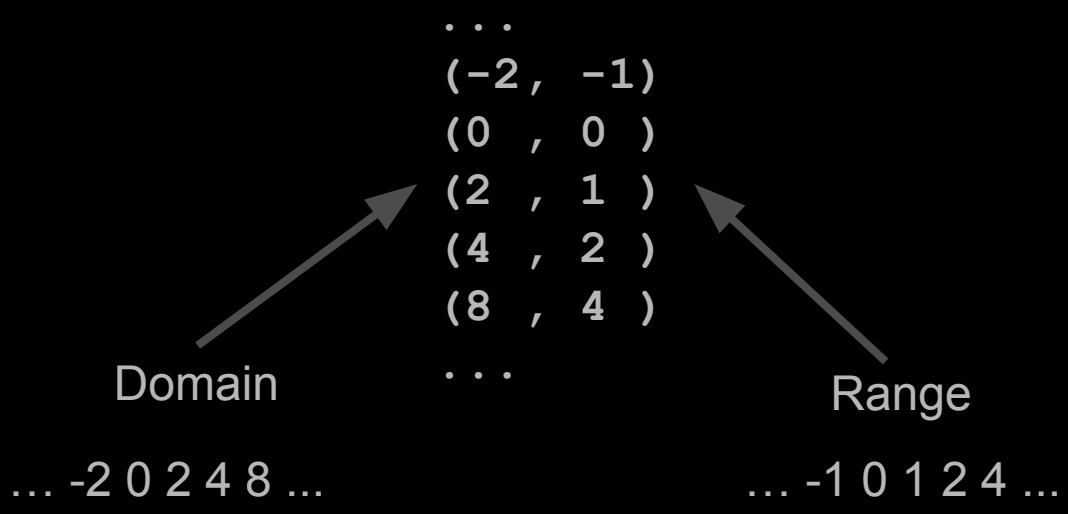
```
...
(-2, -1)
(0 , 0 )
(2 , 1 )
(4 , 2 )
(8 , 4 )
...
```

NOT THIS

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



One input, one output



Pulling it into JS

separate arity from functions

```
function get(property, object) {
  return object[property];
}

var people = [ {name: ...}, ... ];
```

Args up front

```
function getPersonName(person) {
  return get('name', person);
}

var names = people.map(getPersonName);
```

More args later

```
// ☆☆ MAGIC! ☆☆

var names = people.map(get('name'));
```

```
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 }
```

Our code becomes

```
var get = curry(function (property, object) {
  return object[property];
});

var names = people.map(get('name'));
```

jsbin.com/romun

answers: http://jsbin.com/huqif

Example currying

```
var words = function(str) {
  return split(' ', str);
};
```

```
var words = split(' ');
```

```
words("Jingle bells Batman smells");
//=> ['Jingle', 'bells', 'Batman', 'smells']
```

```
words("Jingle bells Batman smells");
//=> ['Jingle', 'bells', 'Batman', 'smells']
```

Example currying

```
var greater = function(x,y){ return x > y ? x : y; };
```

```
var max = function(xs) {
  return reduce(greater, -Infinity, xs);
};
```

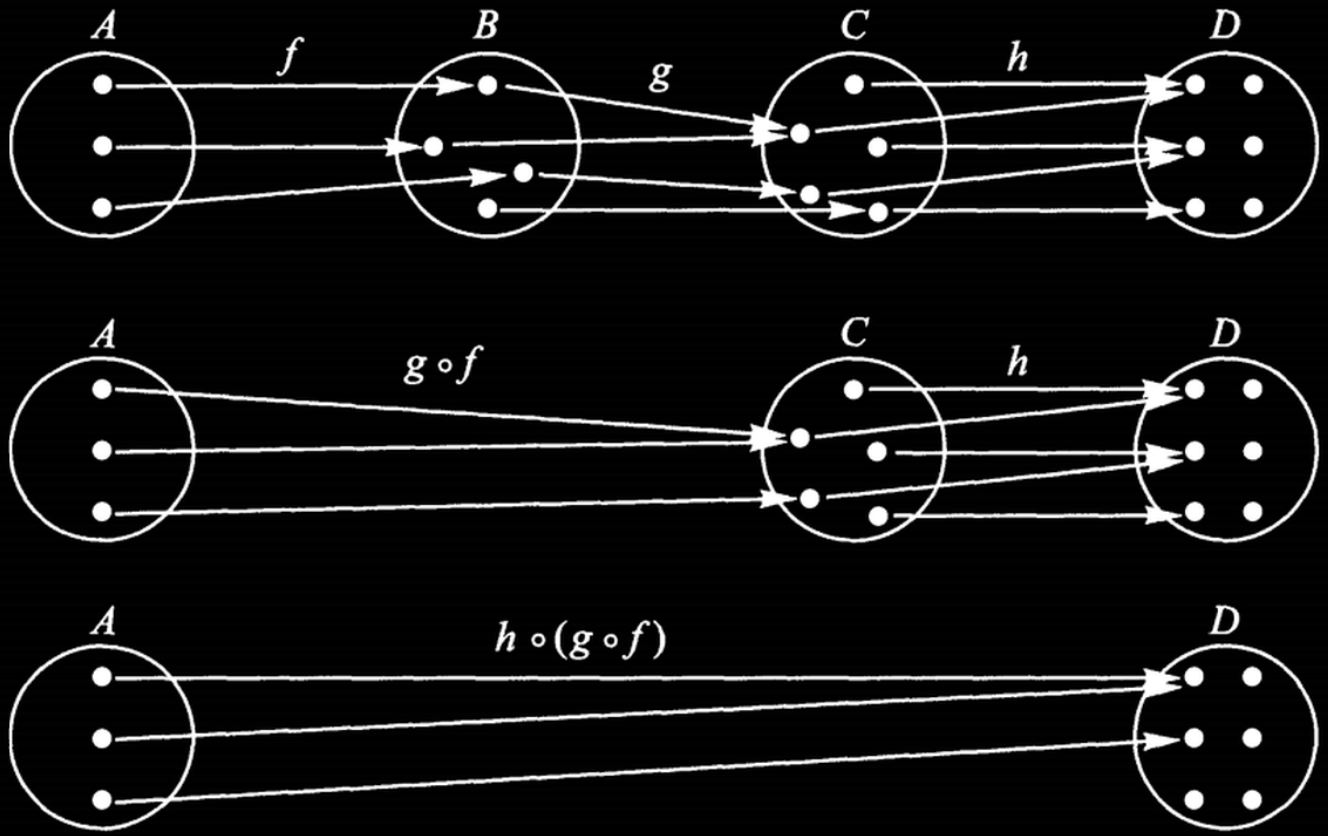
```
var max = reduce(greater, -Infinity)
```

```
max([323,523,554,123,5234])
//=> 5234
```

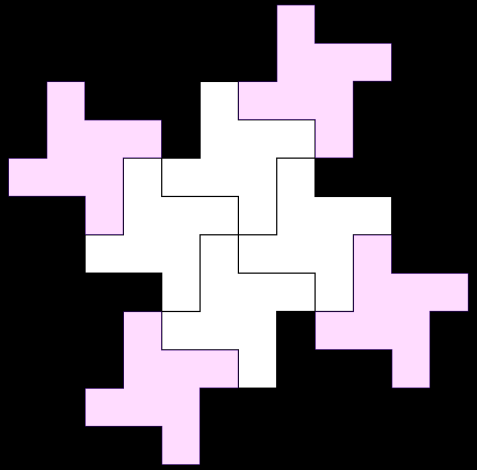
```
max([323,523,554,123,5234])
//=> 5234
```

Back to weird noun-function land

functions can “meld” aka *compose*



Aside: categories



Zooming way out from code

You start seeing similarities

Pure functions are the arrows of a “category”

Brian will show you more in The Journey

separate composition from args

Javascript code composes functions all the time

But too often it uses unnecessary arguments

example “glue” names

```
on_error(function(error) {
 log(error.message);
});
```

To compose the logging and message extraction we created a function and a “glue” name called *error*.

Simplified compose function

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

This function does not care what is inside f or g. It produces a new function that pushes a value through.

“olleH” ← “Hello” ← “hello”

compose(reverse, properNoun)

Cut out the middle man

```
on_error( compose(log, get('message')) );
```

Cut out the middle man

```
on_error( compose(log, get('message')) );
```

jsbin.com/jevag

answers: http://jsbin.com/lutug

(sorry)

recognize most loops are one of

reduce

filter map

If you write a loop you get an *F !*

Loops

In review...

1. Make all function inputs explicit as arguments.
2. These arguments can be provided over time, not just all at once.
3. Try not to modify outside things.
4. Compose without “glue” variables.

End of Part I

The Silence

Part II

The Voyage

```
add(1, 1)
//=> 2
```

```
// 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))
```

```
add("ta", "cos")
//=> tacos

add(9.2, 0.5)
//=> 9.7

add([1,2,3], [4,5,6])
//=> [1,2,3,4,5,6]
```

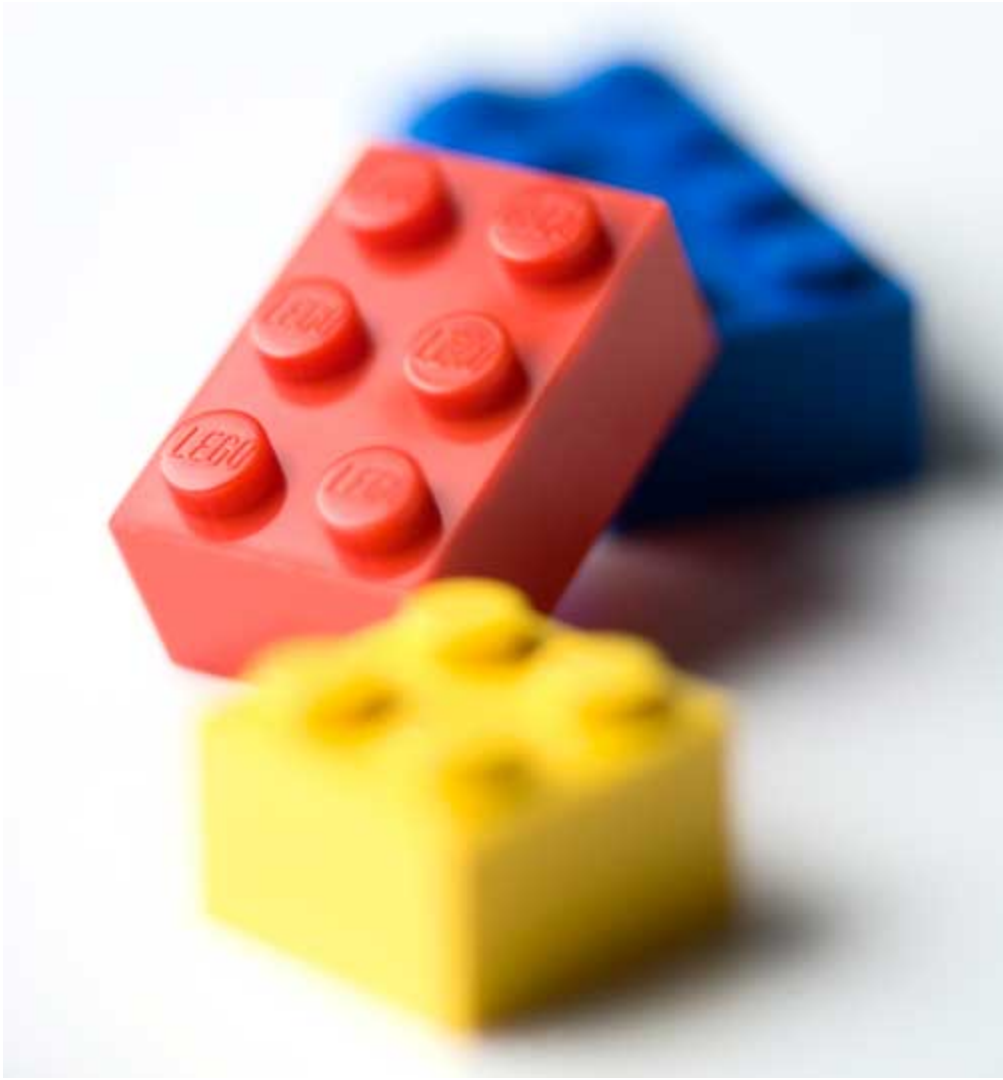
Category Theory

```
compose :: (b -> c) -> (a -> b) -> (a -> c)
```

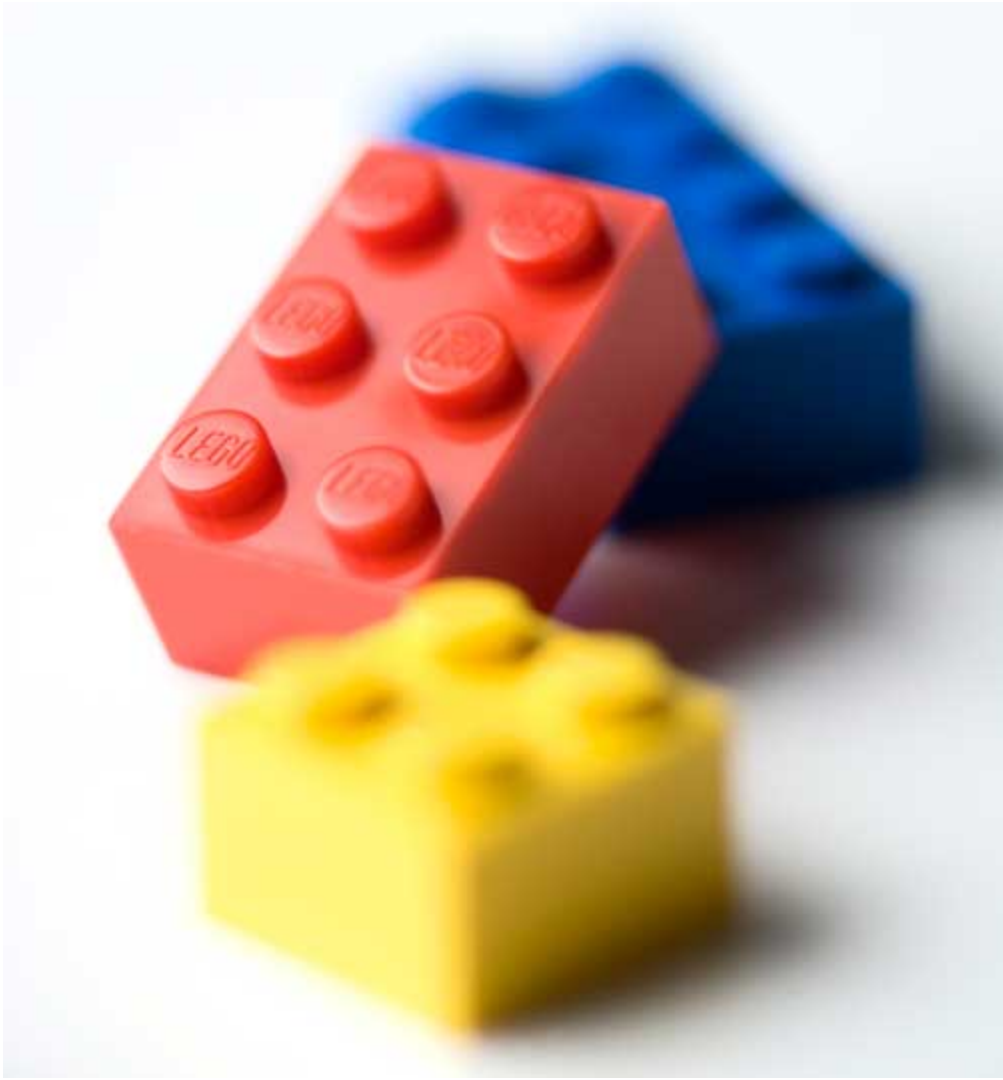
```
id :: a -> a
```

Category Laws

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

// right identity
compose(f, id)  == f

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

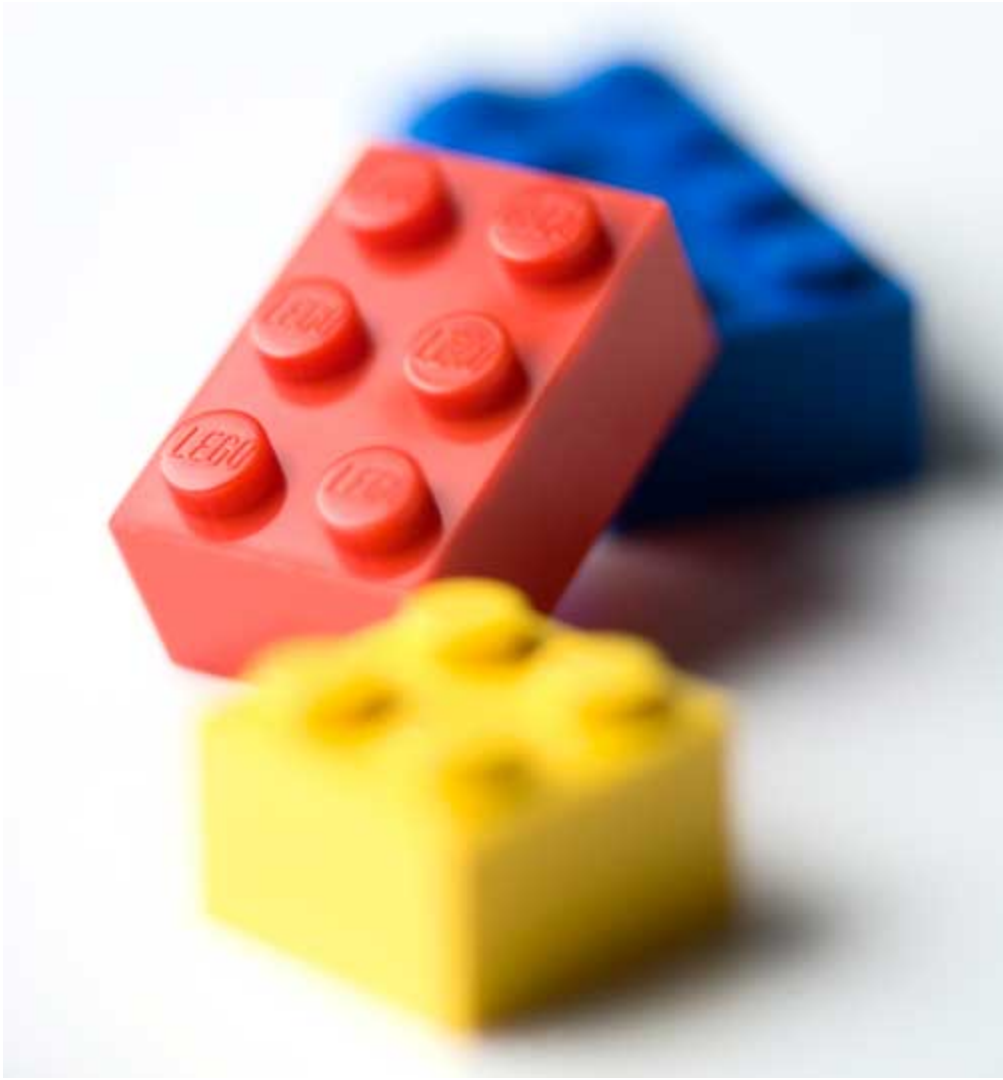


Nulls



Nulls

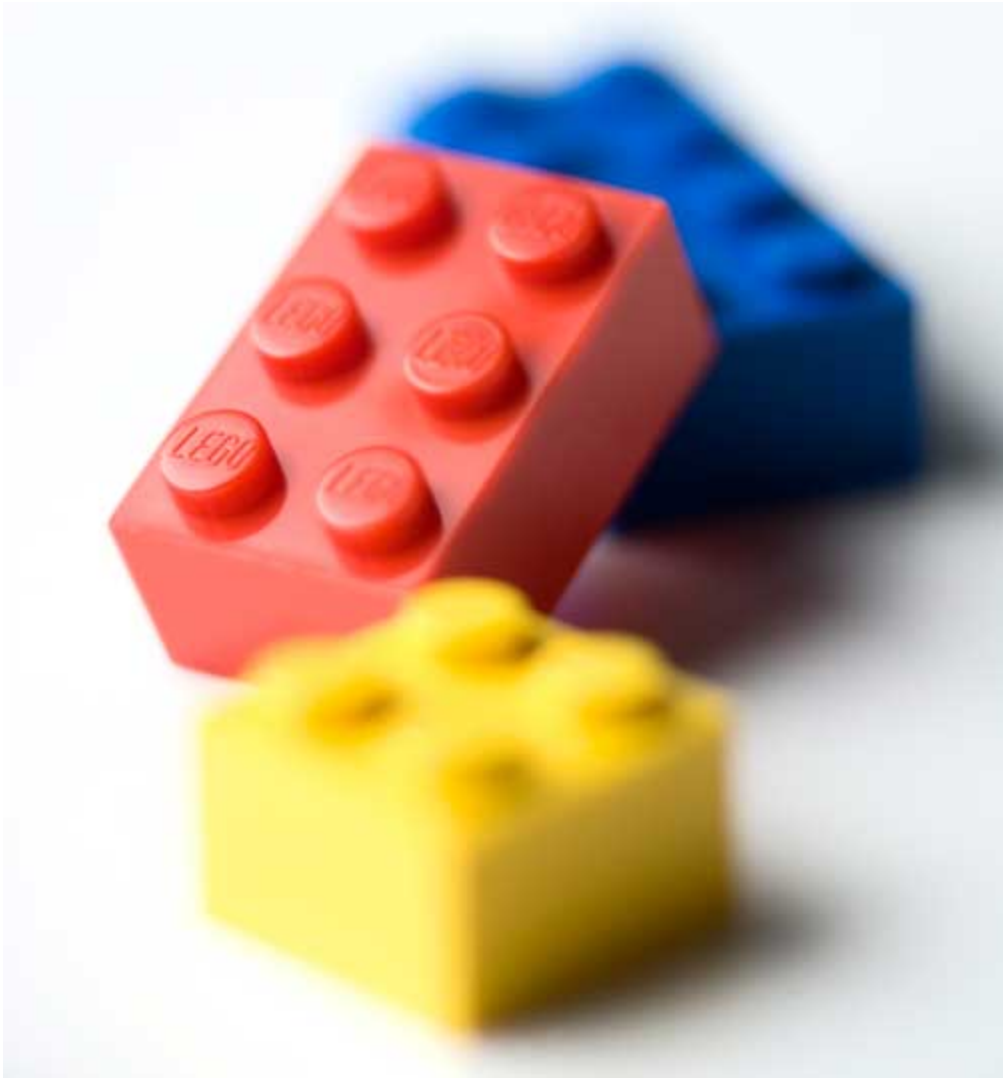
Callbacks



Nulls

Callbacks

Errors

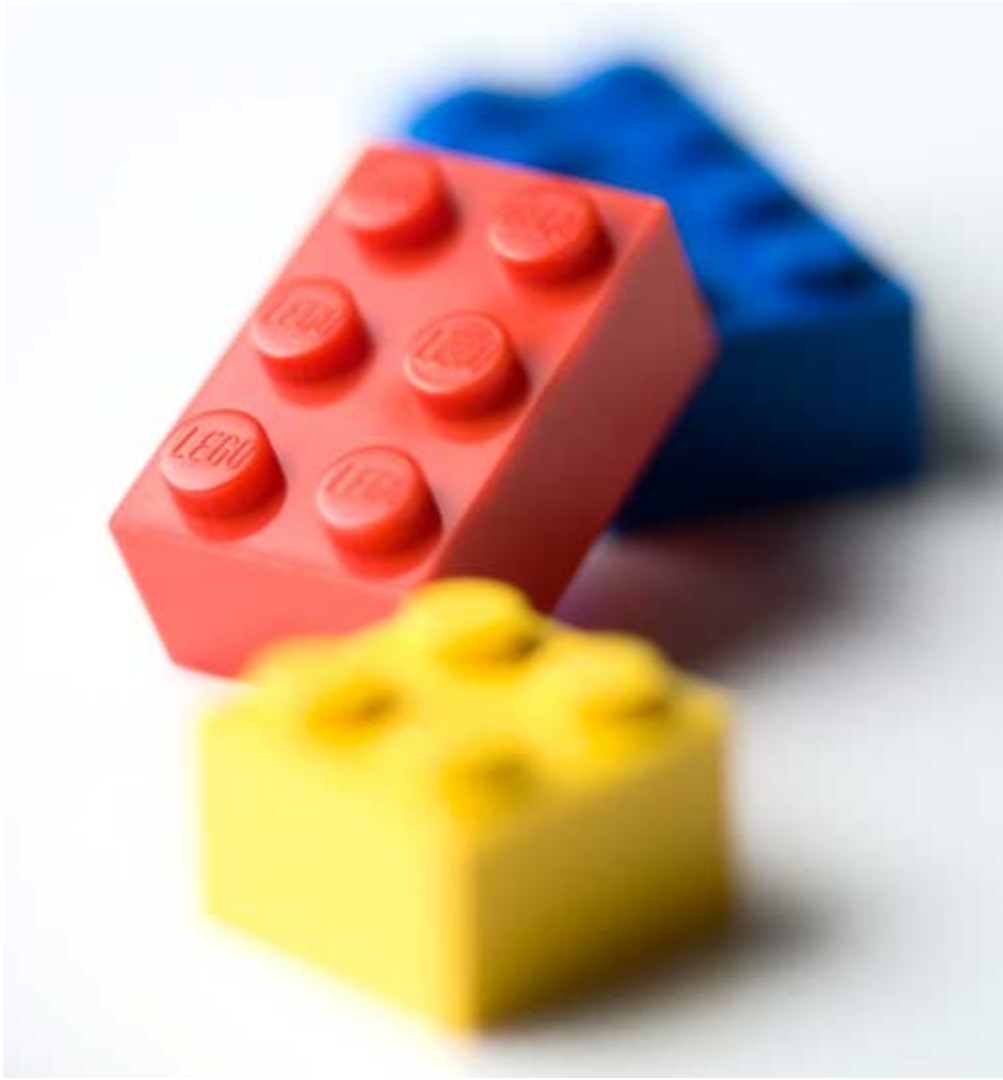


Nulls

Callbacks

Errors

Side effects



Objects

1. Containers/Wrappers for values
2. No methods
3. Not nouns
4. Probably won’t be making your own often

*think of it this way for now.

Objects

```
var _Container = function(val) {
  this.val = val;
}

var Container = function(x) { return new _Container(x); };

Container(3)
//=> _Container {val: 3}
```

Objects

```
capitalize("flamethrower")
//=> "Flamethrower"
```

```
capitalize(Container("flamethrower"))
//=> [object Object]
```

Objects

```
var _Container.prototype.map = function(f) {
  return Container(f(this.val));
}

Container("flamethrower").map(function(s){ return capitalize(s) })

//=> Container(“Flamethrower”)
```

Objects

```
var _Container.prototype.map = function(f) {
  return Container(f(this.val));
}

Container("flamethrower").map(capitalize)

//=> Container(“Flamethrower”)
```

Objects

```
Container(3).map(add(1))
//=> Container(4)

[3].map(add(1))
//=> [4]
```

the true map

goes within

Objects

```
Container([1,2,3]).map(reverse).map(first)
//=> Container(3)
```

```
Container("flamethrower").map(length).map(add(1))
//=> Container(13)
```

Objects

```
var map = _.curry(function(f, obj) {
  return obj.map(f)
})

Container(3).map(add(1)) // Container(4)

map(add(1), Container(3)) // Container(4)
```

Objects

```
map(match(/cat/g), Container("catsup"))
//=> Container([“cat”])

map(compose(first, reverse), Container("dog"))
//=> Container(“g”)
```

Functor

“An object or data structure you can map over”

functions: map

Them pesky nulls

```
var getElement = document.querySelector
var getNameParts = compose(split(' '), get('value'), getElement)


getNameParts('#full_name')
//=> ['Jonathan', 'Gregory', 'Brandis']
```

Them pesky nulls

```
var getElement = document.querySelector
var getNameParts = compose(split(' '), get('value'), getElement)


getNameParts('#fullname')
//=> Boom!
```

Maybe

Captures a null check

The value inside may not be there

Sometimes has two subclasses Just / Nothing

Sometimes called Option with subclasses Some/None

Maybe

```
var _Maybe.prototype.map = function(f) {
  return this.val ? Maybe(f(this.val)) : Maybe(null);
}
```

```
map(capitalize, Maybe("flamethrower"))
//=> Maybe(“Flamethrower”)
```

Maybe

```
var _Maybe.prototype.map = function(f) {
  return this.val ? Maybe(f(this.val)) : Maybe(null);
}
```

```
map(capitalize, Maybe(null))
//=> Maybe(null)
```

Maybe

```
var firstMatch = compose(first, match(/cat/g))


firstMatch("dogsup")
//=> Boom!
```

Maybe

```
var firstMatch = compose(map(first), Maybe, match(/cat/g))
```

```
firstMatch("dogsup")
//=> Maybe(null)
```

Maybe

```
var firstMatch = compose(map(first), Maybe, match(/cat/g))
```

```
firstMatch("catsup")
//=> Maybe(“cat”)
```

http://jsbin.com/yumog/

answers: http://jsbin.com/sopewomi

Either

Typically used for pure error handling

Like Maybe, but with an error message embedded

Has two subclasses: Left/Right

Maps the function over a Right, ignores the Left

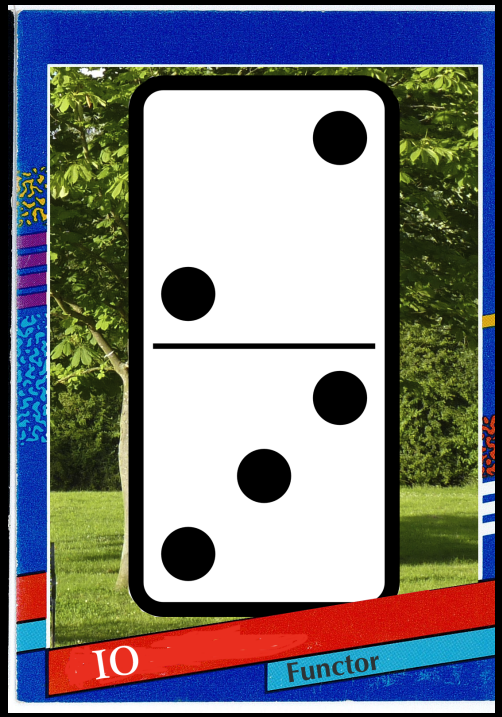
Either

```
map(function(x) { return x + 1; }, Right(2))
//=> Right(3)

map(function(x) { return x + 1; }, Left('some message))
//=> Left('some message')
```

Either

```
var determineAge = function(user){
  return user.age ? Right(user.age) : Left("couldn't get age");
}
var yearOlder = compose(map(add(1)), determineAge)

yearOlder({age: 22})
//=> Right(23)

yearOlder({age: null})
//=> Left("couldn't get age")
```

IO



A lazy computation “builder”

Typically used to contain side effects

You must *runIO* to perform the operation

Map appends the function to a list of things to run with the effectful value.

IO

```
var email_io = IO(function(){ return $("#email").val() }
var msg_io = map(concat("welcome "), email_io)

runIO(msg_io)
//=> ”welcome steve@foodie.net”
```

runIO()

IO

```
var getBgColor = compose(get("background-color"), JSON.parse)
var bgPref = compose(map(getBgColor), Store.get("preferences"))

var app = bgPref()
//=> IO()

runIO(app)
//=> #efefef
```

IO

```
var email_io = IO(function(){ return $("#email").val() }

var getValue = function(sel){ return $(sel).val() }.toIO()
```

http://jsbin.com/zegat

answers: http://jsbin.com/namewuqa

EventStream

An infinite list of results

Dual of array

Its map is sometimes lazy

Calls the mapped function each time an event happens

EventStream

```
var id_s = map(function(e) { return '#'+e.id }, Bacon.fromEventTarget(document, "click"))
//=> EventStream(String)

id_s.onValue(function(id) { alert('you clicked ' +id) })
```

EventStream

```
var id_s = map(function(e) { return  '#'+e.id }, Bacon.fromEventTarget(document, "click"))
var element_s = map(document.querySelector, id_s)
//=> EventStream(Element)

element_s.onValue(function(el) { alert('The inner html is ' +el.innerHTML) })
```

EventStream

```
var hover_s = Bacon.fromEventTarget(document, "hover")
var element_s = map(compose(document.querySelector, get('id')), hover_s)
var postid_s = map(function(el) { return el.data('post-id') }, element_s)
var future_post_s = map(Api.getProductById, postid_s)
//=> EventStream(Future(Post))
```

EventStream

```
var hover_s = Bacon.fromEventTarget(document, "hover")
var element_s = map(compose(document.querySelector, get('id')), hover_s)
var postid_s = map(function(el) { return el.data('post-id') }, element_s)
var future_post_s = map(Api.getProductById, postid_s)
//=> EventStream(Future(Post))

future_post_s.onValue(alert)
```

Future

Has an eventual value

Similar to a promise, but it’s “lazy”

You must *fork* it to kick it off

It takes a function as it’s value

Calls the function with it’s result once it’s there

Future

```
var makeHtml = function(post){ return "<div>"+post.title+"</div>"};
var page_f = map(makeHtml, http.get('/posts/2'))

page_f.fork(function(err) { throw(err) },
            function(page){ $('#container').html(page) })
```

Future

```
var makeHtml = function(title){ return "<div>"+title+"</div>"}
var createPage = compose(makeHtml, get('title'))
var page_f = compose(map(createPage), http.get('/posts/2'))

page_f.fork(function(err) { throw(err) },
            function(page){ $('#container').html(page) })
```

Future

```
var lineCount = compose(length, split(/\n/))
var fileLineCount = compose(map(lineCount), readFile)

fileLineCount("mydoc.txt").fork(log, log)
//=> 34
```

http://jsbin.com/yikoqi

answers: http://jsbin.com/suxemugi

recognize map

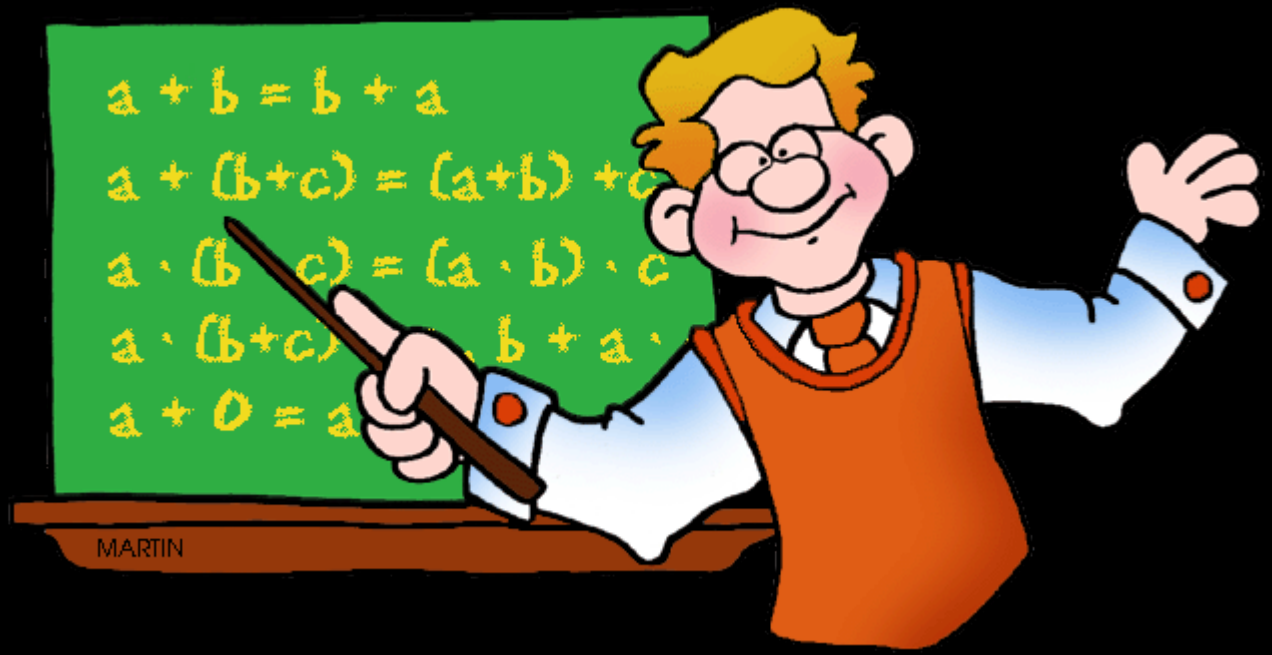
Custom names

```
[x].map(f) // [f(x)]
Maybe(x).attempt(f) // Maybe(f(x))
Promise(x).then(f) // Promise(f(x))
EventStream(x).subscribe(f) // EventStream(f(x))
```

We see it is *map*

```
map(f, [x]) // [f(x)]
map(f, Maybe(x)) // Maybe(f(x))
map(f, Promise(x)) // Promise(f(x))
map(f, EventStream(x)) // EventStream(f(x))
```

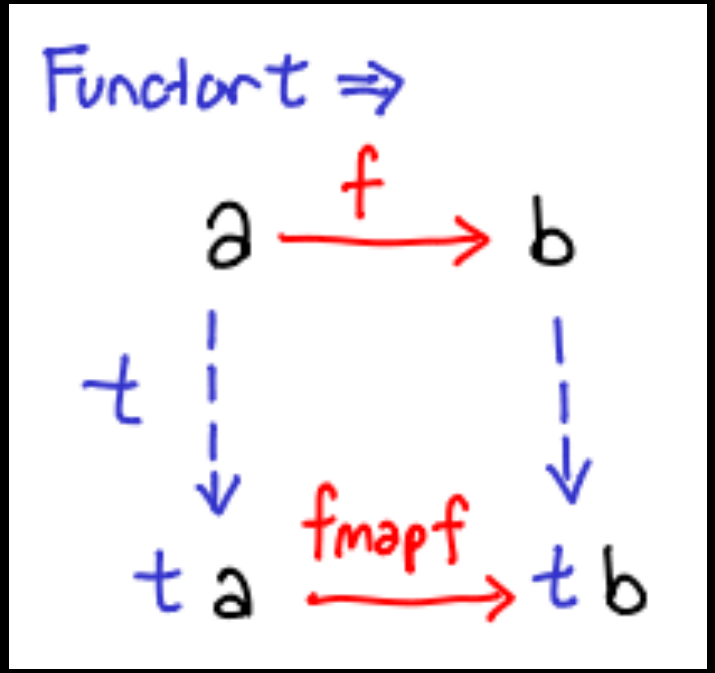
Laws & Properties are useful!



Functor Laws

```
// identity
map(id) == id

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

Functors



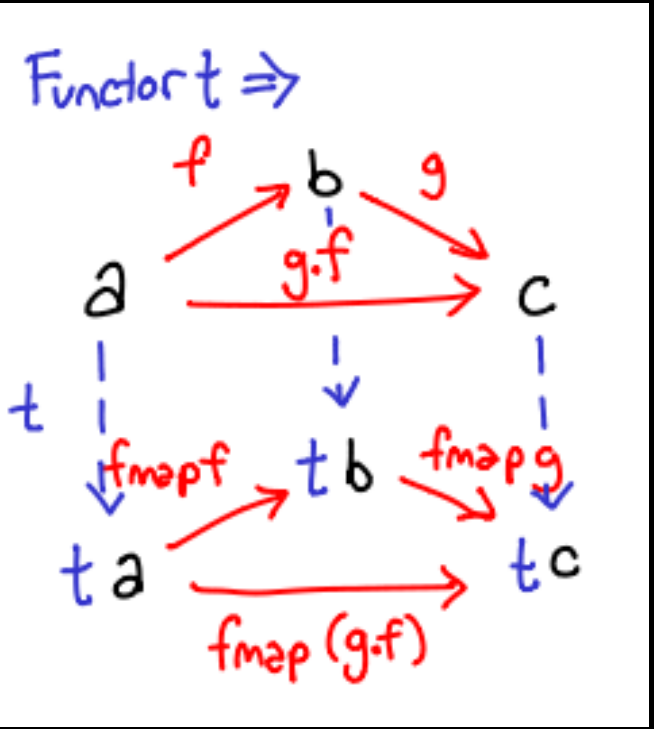

```
reverse :: String -> String

toArray :: a -> Array a
```

```
var toArray = function (x) { return [x] }
```

```
compose(toArray, reverse)("bingo")

//=> [ognib]
```

```
compose(map(reverse), toArray)("bingo")

//=> [ognib]
```

Functors




```
compose(toArray, compose(toUpper, reverse))("bingo")
//=> [ OGNIB]

compose(map(toUpper), map(reverse), toArray)("bingo")
//=> [OGNIB]

compose(map(compose(toUpper, reverse)), toArray)("bingo")
//=> [OGNIB]
```

Natural Transformations

nt :: F a -> T a

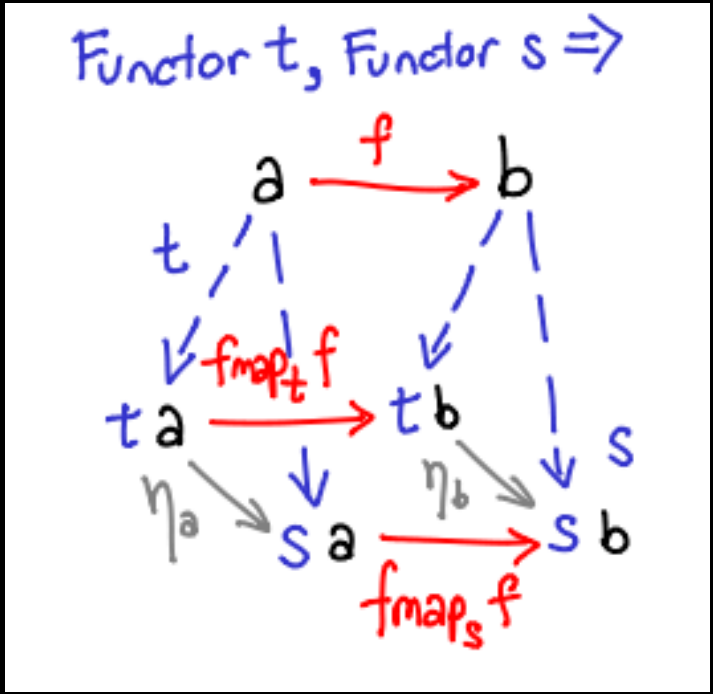
“Takes one functor to another without knowing anything about the values”

Natural Transformations

```
maybeToArray :: Maybe a -> Array a
```

```
maybeToArray(Maybe(2))
//=> [2]

maybeToArray(Maybe(null))
//=> []
```

Natural Transformations




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

```
compose(maybeToArray, map(add(1)))(Maybe(5))
// [6]
```

```
compose(map(add(1)), maybeToArray)(Maybe(5))
// [6]
```

Card Game

Card Game #1

"Make an api call with an id and possibly retrieve a post"

Card Game #1

"Make an api call with an id and possibly retrieve a post"

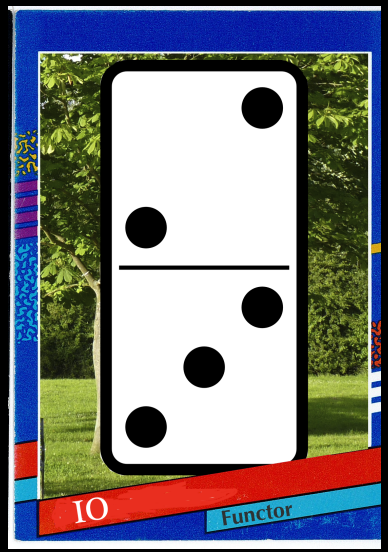
Future(Maybe(Post))

Card Game #2

"Click a navigation link and insert the corresponding html on the page"

Card Game #2

"Click a navigation link and insert the corresponding html on the page"



EventStream(IO(Dom))

Card Game #3

"Submit a signup form & return errors or make an API call that will create a user”

Card Game #3

"Submit a signup form & return errors or make an API call that will create a user”

EventStream(Either(Future(User)))

Pointed Functors

of :: a -> F a

aka: pure, return, unit, point

Pointed Functors

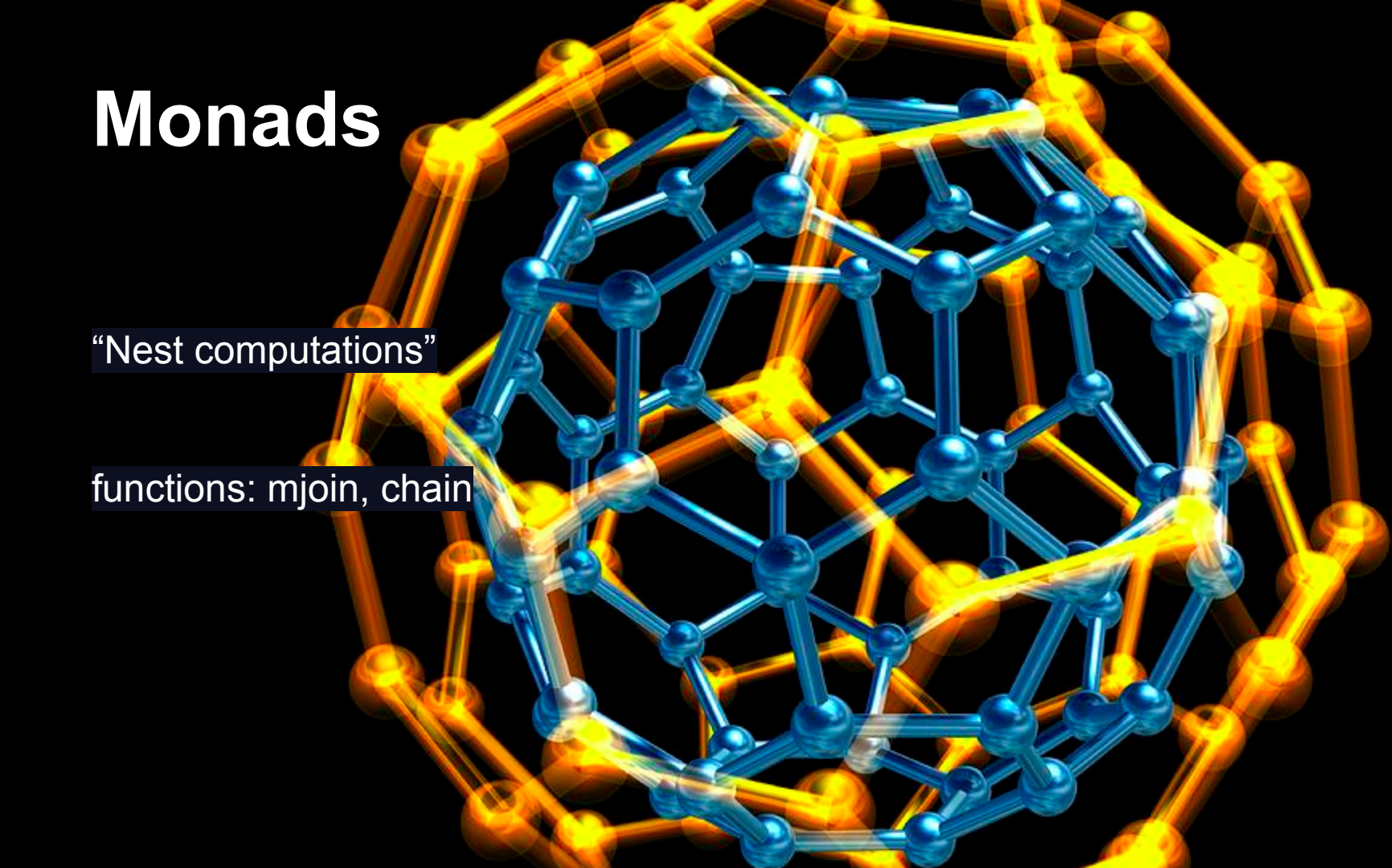
```
Container.of(split)
// Container(split)
```

```
Maybe.of(reverse)
// Maybe(reverse)
```

```
Future.of(match(/dubstep/))
// Future(match(/dubstep/))
```

```
EventStream.of(replace(/dubstep/, 'shoegaze'))
// EventStream(match(/dubstep/,  'shoegaze'))
```

Monads



“Nest computations”

functions: mjoin, chain

Monads

mjoin :: M M a -> M a

chain :: (a -> M b) -> M a -> M b

Pointed Functor + mjoin|chain = Monad

aka: pure, return, unit, point

Monads

```
mjoin(Container(Container(2)))
```

Monads

```
var getTrackingId = compose(Maybe, get("tracking_id"))
var findOrder = compose(Maybe, Api.findOrder)
var getOrderTracking = compose(map(getTrackingId),  findOrder)

var renderPage = compose(map(map(renderTemplate)), getOrderTracking)
renderPage(379)
//=> Maybe(Maybe(Html))
```

Monads

```
var getTrackingId = compose(Maybe, get("tracking_id"))
var findOrder = compose(Maybe, Api.findOrder)
var getOrderTracking = compose(mjoin, map(getTrackingId),  findOrder)

var renderPage = compose(map(renderTemplate), getOrderTracking)
renderPage(379)
//=> Maybe(Html)
```

Monads

```
var setSearchInput = function(x){ return ("#input").val(x); }.toIO()
var getSearchTerm = function(){ return getParam("term", location.search) }.toIO()
var initSearchForm = compose(map(setSearchInput),  getSearchTerm)

initSearchForm()
//=> IO(IO(Dom))

map(runIO, initSearchForm())
```

Monads

```
var setSearchInput = function(x){ return ("#input").val(x); }.toIO()
var getSearchTerm = function(){ return getParam("term", location.search) }.toIO()
var initSearchForm = compose(mjoin, map(setSearchInput),  getSearchTerm)

initSearchForm()
//=> IO(Dom)

runIO(initSearchForm())
```

Monads

```
var sendToServer = httpGet('/upload')
var uploadFromFile = compose(mjoin, map(sendToServer),  readFile)

uploadFromFile("/tmp/my_file.txt").fork(logErr, alertSuccess)
```

Monads

```
var sendToServer = httpGet('/upload')
var uploadFromFile = compose(mjoin, map(sendToServer), mjoin, map(readFile), askUser)

uploadFromFile('what file?').fork(logErr, alertSuccess)
```

Monads

```
var chain = function(f) {
  return compose(mjoin, map(f))
}
```

a.k.a: flatMap, bind

Monads

```
var sendToServer = httpGet('/upload')
var uploadFromFile = compose(chain(sendToServer), chain(readFile), askUser)

uploadFromFile('what file?').fork(logErr, alertSuccess)
```

Monads

```
var sendToServer = httpGet('/upload')

var uploadFromFile = function(what) {
  return askUser(what).chain(readFile).chain(sendToServer);
}

uploadFromFile('what file?').fork(logErr, alertSuccess)
```

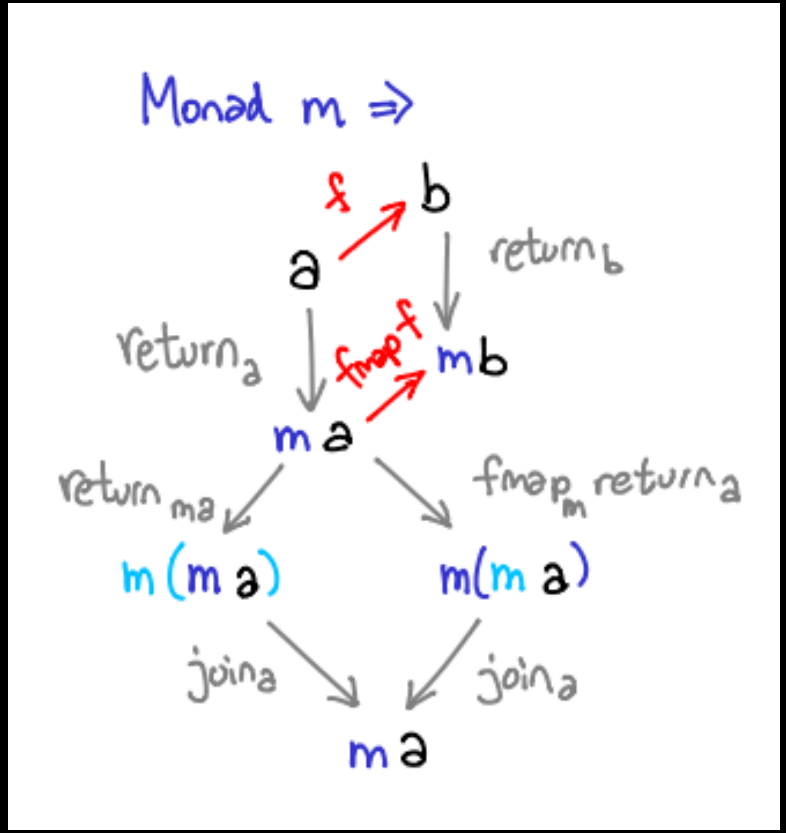
Monads

```
var chain = function(f) {
  return compose(mjoin, map(f))
}

var mjoin = chain(id)
```

http://jsbin.com/woweg

answers: http://jsbin.com/teholoko

Monads


Monad m ⇒
f
a
b
return b
return a
fmap f
m b
m a
return m a
fmap m return a
m (m a)
m (m a)
join a
join a
m a

Monad Laws

compose(mjoin, fmap(g), mjoin, fmap(f))

mcompose(g, f)

Monad Laws

```
// left identity
mcompose(M, f)  == f

// right identity
mcompose(f, M)  == f

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

Applicative Functor

“Run full computations in a context”

functions: ap, liftA2, liftA..n

Applicatives

```
map(add(1), Container(2))
//=> Container(3)

map(add, Container(2))
//=> Container(add(2))
```

Applicatives

ap :: A (a -> b) -> A a -> A b

Pointed Functor + ap = Applicative

aka: <*>

Applicatives

```
Container.of(f).ap(Container(x))
//=> Container(f(x))
```

```
Container.of(f).ap(Container(x)).ap(Container(y))
//=> Container(f(x, y))
```

Applicatives

```
Container.of(add).ap(Container(1)).ap(Container(3))
//=> Container(4)
```

Applicatives

```
Maybe.of(add).ap(Maybe(1)).ap(Maybe(3))
//=> Maybe(4)

Maybe.of(add).ap(Maybe(1)).ap(Maybe(null))
//=> Maybe(null)
```

Applicatives

```
var showLoaded = _.curry(function(tweetbtn, fbbtn){ alert('Done!') })

EventStream.of(showLoaded).ap(tweet_btn.on('load')).ap(fb_btn.on('load'))
```

Applicatives

```
var loadPage = _.curry(function(products, reviews){ render(products.zip(reviews) })

Future.of(loadPage).ap(Api.get('/products')).ap(Api.get('/reviews'))
```

Applicatives

```
var getVal = compose(Maybe, pluck('value'), document.querySelector)
var save = _.curry(function(email, pass){ return User(email, pass) })

Maybe.of(save).ap(getVal('#email')).ap(getVal('#password'))
//=> Maybe(user)
```

Applicatives

```
var getVal = compose(Maybe, pluck('value'), document.querySelector)
var save = _.curry(function(email, pass){ return User(email, pass) })

liftA2(save, getVal('#email'), getVal('#password'))
//=> Maybe(user)
```

Applicatives

```
var getVal = compose(Maybe, pluck('value'), document.querySelector)
var save = _.curry(function(nm, em, pass){ return User(nm, em, pass) })

liftA3(save,  getVal('#name'),  getVal('#email'), getVal('#password'))
//=> Maybe(user)
```

Applicatives

```
var loadPage = _.curry(function(ships, orders, receipts){
  return render(ships, orders, receipts)
})

liftA3(loadPage, http.get('/shipments'), http.get('/orders'), http.get('/receipts'))
//=> Future(Dom)
```

http://jsbin.com/zowev

answers: http://jsbin.com/gopemefe

Applicative Laws

```
// identity
A(id).ap(m) == m
// composition
A(compose).ap(f).ap(g).ap(w) == f.ap(g.ap(w)))
// homomorphism
A(f).ap(A(x)) == A(f(x))
// interchange
u.ap(A(y)) == A(function(f) { return f(y) }).ap(u)
```

Monoids

“Combination/Accumulation”

functions: mempty, mappend, mconcat

Monoids

```
reduce(function(acc, x) {
  return acc + x;
}, 0, [1,2,3])
```

Monoids

```
reduce(function(acc, x) {
  return acc * x;
}, 1, [1,2,3])
```

Monoids

```
reduce(function(acc, x) {
  return acc || x;
}, false, [false, false, true])
```

Monoids

```
reduce(function(acc, x) {
  return acc && x;
}, true, [false, false, true])
```

Monoids

Semigroup:

“Anything that has a concat (combination) method”

Monoids

Monoid:

“Any semigroup that has an empty() method”

Monoids

```
_Sum = function(v) { this.val = v; }

_Sum.prototype.concat = function(s2) {
  return Sum(this.val + s2.val)
}

_Sum.prototype.empty = function() { return Sum(0) }
```

Monoids

```
Sum(2).concat(Sum(3))
//=> Sum(5)
```

Monoids

```
Sum(2).concat(Sum(3)).concat(Sum(5))
//=> Sum(10)
```

Monoids

```
mconcat([Sum(2), Sum(3), Sum(5)])
//=> Sum(10)
```

Monoids

```
mconcat([Product(2), Product(3), Product(5)])
//=> Product(30)
```

Monoids

```
mconcat([Any(false), Any(true), Any(false)])
//=> Any(true)
```

Monoids

```
mconcat([All(false), Any(true), Any(false)])
//=> All(false)
```

Monoids

```
mconcat([Max(13), Max(2), Max(9)])
//=> Max(13)
```

Monoids

```
compose(getResult, mconcat, map(Sum))([1,2,3])
//=>6
```

Monoids

```
compose(getResult, mconcat, map(Any))([false,false,true])
//=>true
```

Monoids

```
mconcat([toUpperCase, reverse])(["bonkers"])
//=> “BONKERSsreknob”
```

http://jsbin.com/jeqof

answers: http://jsbin.com/mavugozi

Monoids

```
mconcat([Failure(["message1"]), Success(attrs), Failure(["message2"])])
//=> Failure(["message1", "message2"])
```

Monoids

```
mconcat([Success(attrs), Success(attrs)])
//=> Success(attrs)
```

Monoids

```
var checkValidations = mconcat([checkPassword, checkEmail, checkName])


checkValidations({name: "Burt"})
//=> Failure([“need a password”, “need an email”])
```

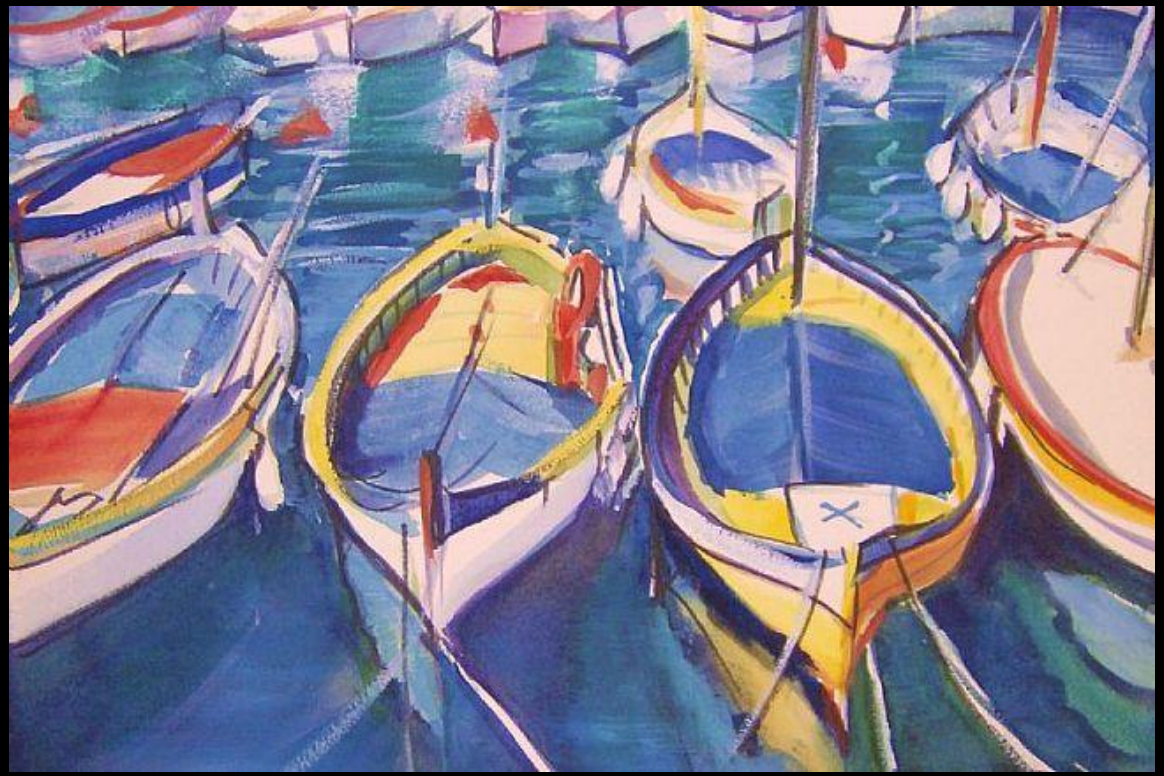
Monoid Laws

```
// left identity
concat(empty, x) == x

// right identity
concat(x, empty) == x

// associativity
concat(concat(x, y), z) == concat(x, concat(y, z))
```

Fantasy Land

End of Part II

The Journey



Part III: The Demo

Congratulations!

You’re a pioneer.

These techniques are new to the JavaScript ecosystem.

Libraries are Evolving

We'll combine the best ones

- CrossEye / ramda
- baconjs / bacon.js
- fantasyland / fantasy-io
- DrBoolean / pointfree-fantasy
- folktale / data.either