JAVASCRIPT

THE GOOD-ER PARTS

A NORMAL CONVERSATION

Me

HEY HONEY, CAN YOU PICK UP SOME DINNER ON YOUR WAY HOME FROM WORK?

HOW ABOUT CAFE RIO?

Wife

SURE, WHAT WOULD YOU LIKE?

SOUNDS GOOD. (GETS CAFE RIO)

A WEIRD, INFLAMMATORY CONVERSATION

Me

TAKE YOUR FOOT OFF THE ACCELERATOR, MOVE IT 6 INCHES TO THE LEFT, AND SLOWLY PLACE IT DOWN ON THE BRAKE. NOW, MOVE THE STEERING WHEEL SLIGHTLY TO THE RIGHT SO THAT YOU CAN TAKE EXIT 332.

Wife

ARE YOU MAD AT ME? DO YOU THINK I'M DUMB? WHY ARE YOU TALKING DOWN TO ME?

JAVASCRIPT: THE GOOD PARTS

- In 2008, Douglas Crockford published JavaScript: The Good Parts.
- You can read it online here: http://bdcampbell.net/javascript/book/javascript_the_good_parts.pdf

JAVASCRIPT: THE GOOD PARTS

- first-class functions
- prototypical inheritance
- object literals
- array literals

JAVASCRIPT: THE BAD PARTS

- == (double equals)
- with statement
- eval
- typed wrappers (e.g. `Boolean`, `Number`, `String`, `Object`, `Array`)
- void operator
- bitwise operators (e.g. &, |, >>, <<)

I WAS RECRUITED TO NETSCAPE WITH THE PROMISE OF "DOING SCHEME" IN THE BROWSER.

. . . (AFTER ARRIVING AT NETSCAPE) . . .

THE DIKTAT FROM UPPER ENGINEERING MANAGEMENT WAS THAT THE LANGUAGE MUST LOOK LIKE JAVA.

Brendan Eich (creator of JavaScript)
https://brendaneich.com/2008/04/popularity/

QUOTE

JAVASCRIPT HAS MORE IN COMMON WITH FUNCTIONAL LANGUAGES LIKE LISP OR SCHEME THAN WITH C OR JAVA.

Douglas Crockford

http://javascript.crockford.com/javascript.html

JAVASCRIPT: THE GOOD-ER PARTS (I.E. FUNCTIONAL PARTS)

- Pure functions
- Immutability
- Expressions (instead of statements)
- Function composition
- Recursion
- Higher-order functions
- Currying

JAVASCRIPT: THE PROCEDURAL PARTS

- Loops: `while`, `do…while`, `for`, `for…of`, `for…in`
- Variable declarations using `var` or `let`
- `if` and `switch` statements (*caveat – explained later)
- Void functions
- Object mutation (e.g. `person.name = 'Joe';`)
- Array mutator methods: `push`, `pop`, `shift`, `unshift`, etc
- Map mutator methods: `set`, `delete`, `clear`
- Set mutator methods: `add`, `delete`, `clear`

PURE FUNCTIONS

PURE FUNCTIONS MUST SATISFY TWO PROPERTIES

- **Referential transparency**: The function always gives the same return value for the same arguments. This means that the function cannot depend on mutable state.
- **Side-effect free**: The function cannot cause any side-effects. Side-effects include logging to the console, mutating an object or reassigning a variable.

PURE FUNCTIONS

```
// Pure
function multiply(a, b) {
  return a * b;
}

// Impure
let heightRequirement = 46;

function canRide(height) {
  return height >= heightRequirement;
}

// Impure
function multiplyWithLogging(a, b) {
  console.log('Arguments: ', a, b);
  return a * b;
}
```

WHY IS EACH ONE IMPURE?

- console.log
- element.addEventListener
- Math.random()
- Date.now()
- $.get ($ can be jQuery, axios, angular 2 Http, etc)

IMMUTABILITY

MUTABILITY MAKES LIFE HARD

```
let heightRequirement = 46;

function canRide(height) {
  return height >= heightRequirement;
}
```

```
// Every half second, set heightRequirement to a
// random number between 0 and 200.
setInterval(() => {
  heightRequirement = Math.floor(Math.random() * 201);
}, 500);

const mySonsHeight = 47;

// Every half second, check if my son can ride.
// Sometimes it will be true and sometimes it will be false.
setInterval(() => console.log(canRide(mySonsHeight)), 500);
```

PROBLEM SOLVED

```
const heightRequirement = 46;

function canRide(height) {
  return height >= heightRequirement;
}
```

ARE THERE ANY PROBLEMS HERE?

```
const constants = {
  heightRequirement: 46,
  // ... other constants go here
};
```

```
function canRide(height) {
  return height >= constants.heightRequirement;
}
```

WHICH OBJECT IS COMPLETELY UN-CHANGEABLE?

	Reassignable	Not Reassignable
Mutable	`let o1 = { foo: 'bar' };`	`const o2 = { foo: 'bar' };`
Immutable	`let o3 = Object.freeze({ foo: 'bar' });`	`const o4 = Object.freeze({ foo: 'bar' });`

AVOIDING OBJECT MUTATION

```
const fruity1 = Object.freeze({
  a: 'apple',
  b: 'banana',
  c: 'cherry'
});

// ES2015
const fruity2 = Object.assign({}, fruity1, { b: 'blueberry' });
// fruity2: { a: 'apple', b: 'blueberry', c: 'cherry' }

// ES2018
const fruity3 = { ...fruity1, c: 'cantaloupe' };
// fruity3: { a: 'apple', b: 'banana', c: 'cantaloupe' }
```

AVOIDING ARRAY MUTATION

```
const a = [4, 5, 6];

// Instead of: a.push(7, 8, 9);
const b = [...a, 7, 8, 9]; // or a.concat(7, 8, 9);

// Instead of: a.pop();
const c = a.slice(0, -1);
```

```
// Instead of: a.unshift(1, 2, 3);
const d = [1, 2, 3, ...a]; // or [1, 2, 3].concat(a);

// Instead of: a.shift();
const e = a.slice(1);

// R = Ramda (alternatively you can use lodash/fp)

// Instead of: a.sort(myCompareFunction);
const f = R.sort(myCompareFunction, a);

// Instead of: a.reverse();
const g = R.reverse(a);
```

AVOIDING MAP MUTATION

```
const map = new Map([
  [1, 'one'],
  [2, 'two'],
  [3, 'three']
]);

// Instead of: map.set(4, 'four');
const map2 = new Map([...map, [4, 'four']]);

// Instead of: map.delete(1);
const map3 = new Map([...map].filter(([key]) => key !== 1));

// Instead of: map.clear();
const map4 = new Map();
```

AVOIDING SET MUTATION

```
const set = new Set(['A', 'B', 'C']);

// Instead of: set.add('D');
const set2 = new Set([...set, 'D']);

// Instead of: set.delete('B');
const set3 = new Set([...set].filter(key => key !== 'B'));

// Instead of: set.clear();
const set4 = new Set();
```

TYPESCRIPT READONLY INTERFACE

```
// `person1` is inferred to be readonly.
// There is a runtime cost for using `Object.freeze`.
const person1 = Object.freeze({ name: 'Mary', age: 31 });
person1.age = 51;

// vs

// More verbose to explicitly declare `person2` as readonly.
// No runtime cost since we did not call `Object.freeze`.
const person2: Readonly<{ name: string, age: number }> = { name: 'Mary', age: 31 };
person2.age = 51;
```

TYPESCRIPT READONLY ARRAY INTERFACE

```
// Runtime cost for using `Object.freeze`.
const instruments1 = Object.freeze(['guitar', 'keyboard', 'drums']);
instruments1.push('kazoo');

// or

// More verbose, but no runtime cost
const instruments2: ReadonlyArray<string> = ['guitar', 'keyboard', 'drums'];
instruments2.push('kazoo');
```

TYPESCRIPT READONLY MAP INTERFACE

```ts
// Does not work!!!!
const numberMap1 = Object.freeze(new Map([
  ['one', 1],
  ['two', 2],
  ['three', 3]
]));
numberMap1.set('four', 4); // I can still mutate the map!

// vs

// You must do it like this:
const numberMap2: ReadonlyMap<string, number> = new Map([
  ['one', 1],
  ['two', 2],
  ['three', 3]
]);
numberMap2.set('four', 4);
```

TYPESCRIPT READONLY SET INTERFACE

```
// Does not work!!!! (same reason as Map)
const turtleSet1 = Object.freeze(new Set([
  'Leonardo',
  'Donatello',
  'Michelangelo',
  'Raphael'
]));
turtleSet1.add('Shredder');

// vs

// You must do it like this:
const turtleSet2: ReadonlySet<string> = new Set([
  'Leonardo',
  'Donatello',
  'Michelangelo',
  'Raphael'
]);
turtleSet2.add('Shredder');
```

IMMUTABLE.JS

- Q: Is it expensive to keep creating new objects instead of mutating existing ones?
- A: Yes, in terms of both CPU and RAM usage.
- Solution: Immutable.js is a library that provides us with persistent, immutable data structures.
- Functional programming languages such as Clojure and Scala have persistent data structures built-in, but we will need a library to help us in JavaScript.

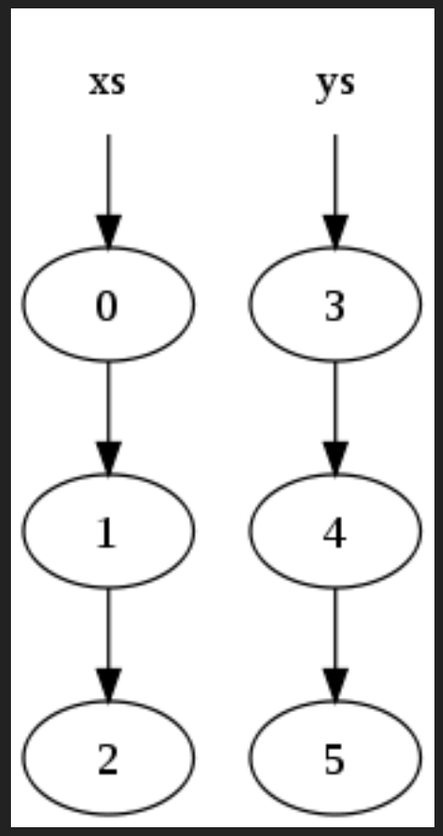
PERSISTENT DATA STRUCTURE:
A DATA STRUCTURE THAT ALWAYS PRESERVES THE PREVIOUS VERSION OF ITSELF WHEN IT IS MODIFIED. SUCH DATA STRUCTURES ARE EFFECTIVELY IMMUTABLE, AS THEIR OPERATIONS DO NOT (VISIBLY) UPDATE THE STRUCTURE IN-PLACE, BUT INSTEAD ALWAYS YIELD A NEW UPDATED STRUCTURE.

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

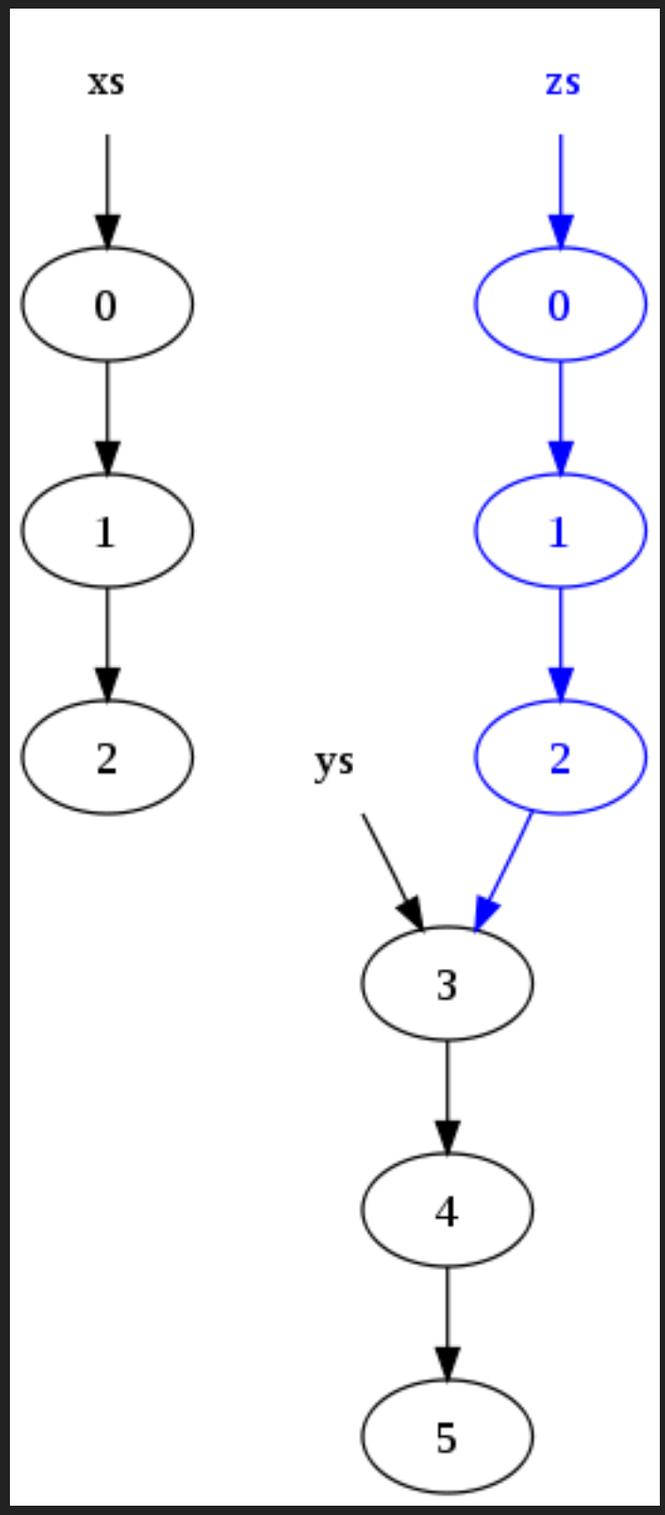
PERSISTENT DATA STRUCTURES

xs = [0, 1, 2]

ys = [3, 4, 5]



zs = xs.concat(ys)



Source: https://en.wikipedia.org/wiki/Persistent_data_structure

IMMUTABLE.JS: LIST

```
import { List } from 'immutable';

// Use in place of `[]`.
const list1 = List(['A', 'B', 'C']);
const list2 = list1.push('D', 'E');

console.log([...list1]); // ['A', 'B', 'C']
console.log([...list2]); // ['A', 'B', 'C', 'D', 'E']
```

IMMUTABLE.JS: MAP

```
import { Map } from 'immutable';

// Use in place of `new Map()`
const map1 = Map<string, number>([
  ['one', 1],
  ['two', 2],
  ['three', 3]
]);
const map2 = map1.set('four', 4);

console.log([...map1]); // [['one', 1], ['two', 2], ['three', 3]]
console.log([...map2]); // [['one', 1], ['two', 2], ['three', 3],
['four', 4]]
```

IMMUTABLE.JS: SET

```
import { Set } from 'immutable';

// Use in place of `new Set()`
const set1 = Set([1, 2, 3, 3, 3, 3, 3, 4]);
const set2 = set1.add(5);

console.log([...set1]); // [1, 2, 3, 4]
console.log([...set2]); // [1, 2, 3, 4, 5]
```

EXPRESSIONS

A STATEMENT IS A COMPLETE LINE OF CODE THAT PERFORMS SOME ACTION, WHILE AN EXPRESSION IS ANY SECTION OF THE CODE THAT EVALUATES TO A VALUE. EXPRESSIONS CAN BE COMBINED "HORIZONTALLY" INTO LARGER EXPRESSIONS USING OPERATORS, WHILE STATEMENTS CAN ONLY BE COMBINED "VERTICALLY" BY WRITING ONE AFTER ANOTHER, OR WITH BLOCK CONSTRUCTS. EVERY EXPRESSION CAN BE USED AS A STATEMENT (WHOSE EFFECT IS TO EVALUATE THE EXPRESSION AND IGNORE THE RESULTING VALUE), BUT MOST STATEMENTS CANNOT BE USED AS EXPRESSIONS. HERE.

Anders Kaseorg

https://www.quora.com/Whats-the-difference-between-a-statement-and-an-expression-in-Python

WHICH ARE EXPRESSIONS?

- if
- switch
- while (loop)
- variable declaration
- variable assignment
- +, -, *, / operators
- function invocation

IF STATEMENT VS IF EXPRESSION

```
const number = 11;

// if statement.
let message1;
if (number > 10) {
  message1 = "I can't count that high";
} else {
  message1 = `You chose ${number}`;
}


// "if" expression (ternary operator).
const message2 = (number > 10) ? "I can't count that high" : `You chose ${number}`;
```

PRAGMATIC IF EXPRESSION

```
const number = 11;

// Each branch must have a `return` statement.
const message = (() => {
  if (number > 10) {
    return "I can't count that high";
  } else {
    return `You chose ${number}`;
  }
})();

console.log(message);
```

PRAGMATIC SWITCH EXPRESSION

```
const polygon = 'octagon';

// Again, each branch must have a `return` statement.
const numberOfSides = (() => {
  switch(polygon) {
    case 'triangle':
      return 3;
    case 'square':
    case 'rectangle':
      return 4;
    case 'pentagon':
      return 5;
    case 'hexagon':
      return 6;
    case 'heptagon':
      return 7;
    case 'octagon':
      return 8;
    default:
      // I can't count that high, so we'll just default it to Infinity.
      return Infinity;
  }
})();

console.log(numberOfSides);
```

FUNCTION COMPOSITION

$$(F \circ G)(X) = F(G(X))$$

Your high school math teacher

THE RIGHT-HAND SIDE: F(G(X))

```
function h(x) {
  return x + 1;
}

function g(x) {
  return x * x;
}

function f(x) {
  return x.toString();
}

const y = f(g(h(1))); // y = (f ∘ g ∘ h)(1)
console.log(y); // '4'
```

THE LEFT-HAND SIDE: (F ∘ G)(X)

```
import * as R from 'ramda';

function h(x) {
  return x + 1;
}

function g(x) {
  return x * x;
}

function f(x) {
  return x.toString();
}

// R = Ramda
const composite = R.compose(f, g, h);
const y = composite(1);
console.log(y); // '4'
```

RECURSION

TO ITERATE IS HUMAN, TO RECURSE, DIVINE.

L Peter Deutsch

ITERATIVE FACTORIAL

```
function iterativeFactorial(n) {
  let product = 1;
  for (let i = 1; i <= n; i++) {
    product *= i;
  }
  return product;
}
```

RECURSIVE FACTORIAL

```
function recursiveFactorial(n) {
  // Base case -- stop the recursion
  if (n === 0) {
    return 1; // 0! is defined to be 1.
  }
  return n * recursiveFactorial(n - 1);
}
```

UH-OH!

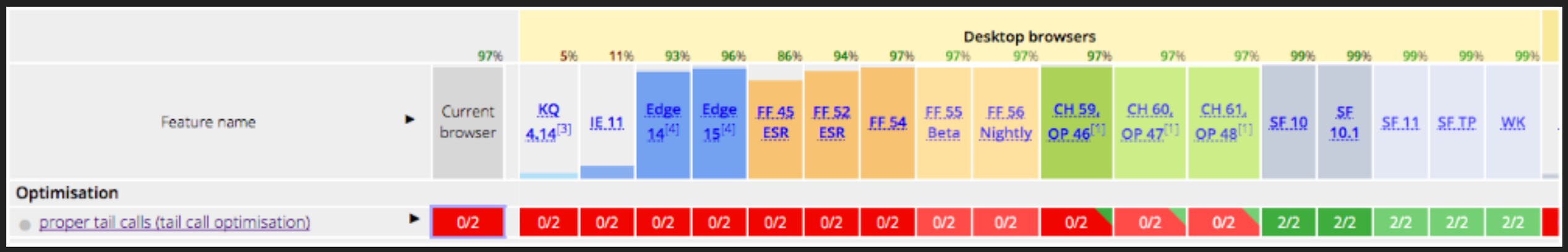
```
recursiveFactorial(20000);
```

```
▶Uncaught RangeError: Maximum call stack size exceeded
    at recursiveFactorial (index.html:266)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
    at recursiveFactorial (index.html:270)
```

PROPER TAIL CALLS (PTC) OPTIMIZATION

- Good news if 100% of your users use Safari! Bad news for everyone else
- Competing standard: Syntactic Tail Calls (STC)

Optimization: https://github.com/tc39/proposal-ptc-syntax#syntactic-tail-calls-stc



OPTIMIZED RECURSIVE FACTORIAL

```
'use strict';

// Optimized for tail call optimization.
function factorial(n, product = 1) {
  if (n === 0) {
    return product;
  }
  return factorial(n - 1, product * n)
}

factorial(20000); // Infinity (only in Safari)
```

HIGHER-ORDER FUNCTIONS

NATIVE

```
const vehicles = [
  { make: 'Honda', model: 'CR-V', type: 'suv', price: 24045 },
  { make: 'Honda', model: 'Accord', type: 'sedan', price: 22455 },
  { make: 'Mazda', model: 'Mazda 6', type: 'sedan', price: 24195 },
  { make: 'Mazda', model: 'CX-9', type: 'suv', price: 31520 },
  { make: 'Toyota', model: '4Runner', type: 'suv', price: 34210 },
  { make: 'Toyota', model: 'Sequoia', type: 'suv', price: 45560 },
  { make: 'Toyota', model: 'Tacoma', type: 'truck', price: 24320 },
  { make: 'Ford', model: 'F-150', type: 'truck', price: 27110 },
  { make: 'Ford', model: 'Fusion', type: 'sedan', price: 22120 },
  { make: 'Ford', model: 'Explorer', type: 'suv', price: 31660 }
];

const averageSUVPrice = vehicles
  .filter(v => v.type === 'suv')
  .map(v => v.price)
  .reduce((sum, price, i, array) => sum + price / array.length, 0);

console.log(averageSUVPrice); // 33399
```

COMPOSITION

```
import * as R from 'ramda';

const vehicles = [
  { make: 'Honda', model: 'CR-V', type: 'suv', price: 24045 },
  { make: 'Honda', model: 'Accord', type: 'sedan', price: 22455 },
  { make: 'Mazda', model: 'Mazda 6', type: 'sedan', price: 24195 },
  { make: 'Mazda', model: 'CX-9', type: 'suv', price: 31520 },
  { make: 'Toyota', model: '4Runner', type: 'suv', price: 34210 },
  { make: 'Toyota', model: 'Sequoia', type: 'suv', price: 45560 },
  { make: 'Toyota', model: 'Tacoma', type: 'truck', price: 24320 },
  { make: 'Ford', model: 'F-150', type: 'truck', price: 27110 },
  { make: 'Ford', model: 'Fusion', type: 'sedan', price: 22120 },
  { make: 'Ford', model: 'Explorer', type: 'suv', price: 31660 }
];

// Using `compose` executes the functions from bottom-to-top.
const averageSUVPrice2 = R.compose(
  R.mean,
  R.map(v => v.price),
  R.filter(v => v.type === 'suv')
)(vehicles);

console.log(averageSUVPrice2); // 33399
```

PIPE

```
import * as R from 'ramda';

const vehicles = [
  { make: 'Honda', model: 'CR-V', type: 'suv', price: 24045 },
  { make: 'Honda', model: 'Accord', type: 'sedan', price: 22455 },
  { make: 'Mazda', model: 'Mazda 6', type: 'sedan', price: 24195 },
  { make: 'Mazda', model: 'CX-9', type: 'suv', price: 31520 },
  { make: 'Toyota', model: '4Runner', type: 'suv', price: 34210 },
  { make: 'Toyota', model: 'Sequoia', type: 'suv', price: 45560 },
  { make: 'Toyota', model: 'Tacoma', type: 'truck', price: 24320 },
  { make: 'Ford', model: 'F-150', type: 'truck', price: 27110 },
  { make: 'Ford', model: 'Fusion', type: 'sedan', price: 22120 },
  { make: 'Ford', model: 'Explorer', type: 'suv', price: 31660 }
];

// Using `pipe` executes the functions from top-to-bottom.
const averageSUVPrice1 = R.pipe(
  R.filter(v => v.type === 'suv'),
  R.map(v => v.price),
  R.mean
)(vehicles);

console.log(averageSUVPrice1); // 33399
```

CURRYING

UN-CURRIED FUNCTION

```
function dot(vector1, vector2) {
  return vector1.reduce((sum, element, index) => sum += element * vector2[index], 0);
}
```

```
const v1 = [1, 3, -5];
const v2 = [4, -2, -1];

console.log(dot(v1, v2)); // 1(4) + 3(-2) + (-5)(-1) = 4 - 6 + 5 = 3
```

MANUALLY CURRYING THE FUNCTION

```
function curriedDot(vector1) {
  return function (vector2) {
    return vector1.reduce((sum, element, index) => sum += element * vector2[index], 0);
  }
}

// Taking the dot product of any vector with [1, 1, 1]
// is equivalent to summing up the elements of the vector.
const sumElements = curriedDot([1, 1, 1]);

console.log(sumElements([1, 3, -5])); // -1
console.log(sumElements([4, -2, -1])); // 1
```

USING A LIBRARY TO DO THE CURRYING FOR US!

```
import * as R from 'ramda';

function dot(vector1, vector2) {
  return vector1.reduce((sum, element, index) => sum += element * vector2[index], 0);
}
```

```
const v1 = [1, 3, -5];
const v2 = [4, -2, -1];

// Use Ramda to do the currying for us!
const curriedDot = R.curry(dot);
const sumElements = curriedDot([1, 1, 1]);

console.log(sumElements(v1)); // -1
console.log(sumElements(v2)); // 1

// This works! You can still call the curried function with two arguments.
console.log(curriedDot(v1, v2)); // 3
```

PARTIAL APPLICATION WITHOUT CURRYING

```
function dot(vector1, vector2) {
  return vector1.reduce((sum, element, index) => sum += element * vector2[index], 0);
}
```

```
// We can use the `bind` method to do partial application -- bypassing currying.
const sumElements = dot.bind(null, [1, 1, 1]);

console.log(sumElements([1, 3, -5])); // -1
console.log(sumElements([4, -2, -1])); // 1
```

THANK YOU!

- email: matt.banz@gmail.com
- Git Hub: https://github.com/battmanz/functional-javascript-examples
- Open Source Article: https://opensource.com/article/17/6/functional-javascript