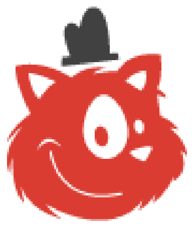


Don't Be Scared Of Functional Programming

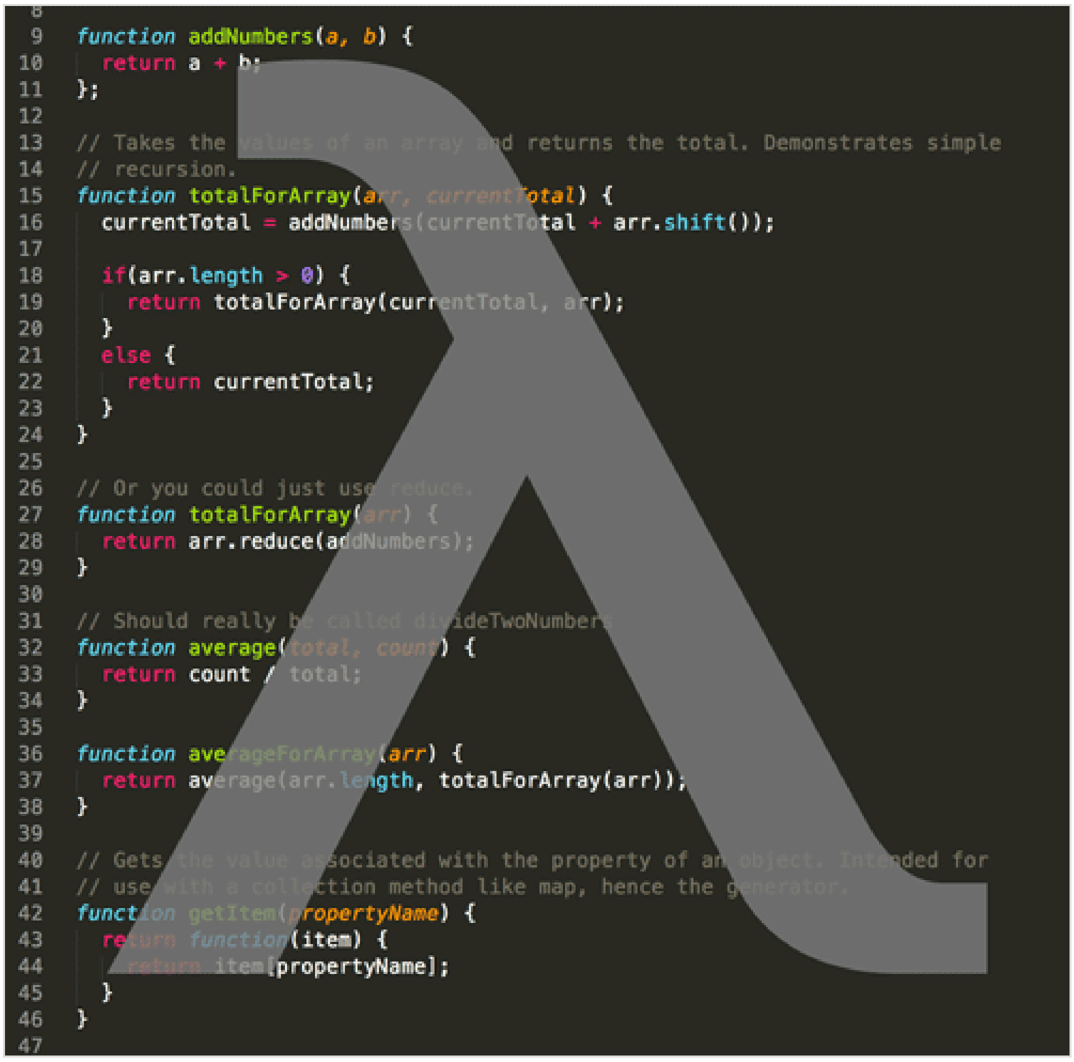
By Jonathon Morgan

Published on July 2nd, 2014 in JavaScript, Techniques with 58 Comments

Functional programming is the mustachioed hipster of programming paradigms. Originally relegated to the annals of computer science academia, functional programming has had a recent renaissance that is due largely to its utility in distributed systems (and probably also because "pure" functional languages like Haskell are difficult to grasp, which gives them a certain cachet).

Stricter functional programming languages are typically used when a system's performance and integrity are both critical — i.e. your program needs to do exactly what you expect every time and needs to operate in an environment where its tasks can be shared across hundreds or thousands of networked computers. Clojure[1], for example, powers Akamai[2], the massive content delivery network utilized by companies such as Facebook, while Twitter famously adopted[3] Scala[4] for its most performance-intensive components, and Haskell[5] is used by AT&T for its network security systems.

These languages have a steep learning curve for most front-end web developers; however, many more approachable languages incorporate features of functional programming, most notably Python, both in its core library, with functions like `map` and `reduce` (which we'll talk about in a bit), and with libraries such as Fn.py[6], along with JavaScript, again using collection methods, but also with libraries such as Underscore.js[7] and Bacon.js[8].



```
function addNumbers(a, b) {
  return a + b;
};

// Takes the values of an array and returns the total. Demonstrates simple
// recursion.
function totalForArray(arr, currentTotal) {
  currentTotal = addNumbers(currentTotal + arr.shift());

  if(arr.length > 0) {
    return totalForArray(currentTotal, arr);
  }
  else {
    return currentTotal;
  }
}

// Or you could just use reduce.
function totalForArray(arr) {
  return arr.reduce(addNumbers);
}

// Should really be called divideTwoNumbers
function average(total, count) {
  return count / total;
}

function averageForArray(arr) {
  return average(arr.length, totalForArray(arr));
}

// Gets the value associated with the property of an object. Intended for
// use with a collection method like map, hence the generator.
function getItem(propertyName) {
  return function(item) {
    return item[propertyName];
  }
}
```

Functional programming can be daunting, but remember that it isn't only for PhDs, data scientists and architecture astronauts. For most of us, the real benefit of adopting a functional style is that our programs can be broken down into smaller, simpler pieces that are both more reliable and easier to understand. If you're a front-end developer working with data, especially if you're formatting that data for visualization using D3, Raphael or the like, then functional programming will be an essential weapon in your arsenal.

Finding a consistent definition of functional programming is tough, and most of the literature relies on somewhat foreboding statements like "functions as first-class objects," and "eliminating side effects." Just in case that doesn't bend your brain into knots, at a more theoretical level, functional programming is often explained in terms of lambda calculus[9] (some actually argue[10] that functional programming is basically math) — but you can relax. From a more pragmatic perspective, a beginner needs to understand only two concepts in order to use it for everyday applications (no calculus required!).

First, data in functional programs should be **immutable**, which sounds serious but just means that it should never change. At first, this might seem odd (after all, who needs a program that never changes anything?), but in practice, you would simply create new data structures instead of modifying ones that already exist. For example, if you need to manipulate some data in an array, then you'd make a new array with the updated values, rather than revise the original array. Easy!

Secondly, functional programs should be **stateless**, which basically means they should perform every task as if for the first time, with no knowledge of what may or may not have happened earlier in the program's execution (you might say that a stateless program is ~~ignorant of the past~~[11]). Combined with immutability, this helps us think of each function as if it were operating in a vacuum, blissfully ignorant of anything else in the application besides other functions. In more concrete terms, this means that your functions will operate only on data passed in as arguments and will never rely on outside values to perform their calculations.

Immutability and statelessness are core to functional programming and are important to understand, but don't worry if they don't quite make sense yet. You'll be familiar with these principles by the end of the article, and I promise that the beauty, precision and power of functional programming will turn your applications into bright, shiny, datachomping rainbows. For now, start with simple functions that return data (or other functions), and then combine those basic building blocks to perform more complex tasks.

For example, let's say we have an API response:

```
var data = [
  {
    name: "Jamestown",
    population: 2047,
    temperatures: [-34, 67, 101, 87]
  },
  {
    name: "Awesome Town",
    population: 3568,
    temperatures: [-3, 4, 9, 12]
  }
  {
    name: "Funky Town",
    population: 1000000,
    temperatures: [75, 75, 75, 75, 75]
  }
];
```

If we want to use a chart or graphing library to compare the average temperature to population size, we'd need to write some JavaScript that makes a few changes to the data before it's formatted correctly for our visualization. Our graphing library wants an array of x and y coordinates, like so:

```
[
  [x, y],
  [x, y]
  …etc
]
```

Here, `x` is the average temperature, and `y` is the population size.

Without functional programming (or without using what's called an "imperative" style), our program might look like this:

```
var coords = [],
    totalTemperature = 0,
    averageTemperature = 0;

for (var i=0; i < data.length; i++) {
  totalTemperature = 0;

  for (var j=0; j < data[i].temperatures.length; j++) {
    totalTemperature += data[i].temperatures[j];
  }

  averageTemperature = totalTemperature / data[i].temperatures.length;

  coords.push([averageTemperature, data[i].population]);
}
```

Even in a contrived example, this is already becoming difficult to follow. Let’s see if we can do better.

When programming in a functional style, you’re always looking for simple, repeatable actions that can be abstracted out into a function. We can then build more complex features by calling these functions in sequence (also known as “composing” functions) — more on that in a second. In the meantime, let’s look at the steps we’d take in the process of transforming the initial API response to the structure required by our visualization library. At a basic level, we’ll perform the following actions on our data:

- add every number in a list,
- calculate an average,
- retrieve a single property from a list of objects.

We’ll write a function for each of these three basic actions, then compose our program from those functions. Functional programming can be a little confusing at first, and you’ll probably be tempted to slip into old imperative habits. To avoid that, here are some simple ground rules to ensure that you’re following best practices:

1. All of your functions must accept at least one argument.
2. All of your functions must return data or another function.
3. No loops!

OK, let's add every number in a list. Remembering the rules, let's make sure that our function accepts an argument (the array of numbers to add) and returns some data.

```
function totalForArray(arr) {
  // add everything
  return total;
}
```

So far so good. But how are we going to access every item in the list if we don't loop over it? Say hello to your new friend, recursion! This is a bit tricky, but basically, when you use recursion, you create a function that calls itself unless a specific condition has been met — in which case, a value is returned. Just looking at an example is probably easiest:

```
// Notice we're accepting two values, the list and the current total
function totalForArray(currentTotal, arr) {

  currentTotal += arr[0];

  // Note to experienced JavaScript programmers, I'm not using Array.shift on
  // purpose because we're treating arrays as if they are immutable.
  var remainingList = arr.slice(1);

  // This function calls itself with the remainder of the list, and the
  // current value of the currentTotal variable
  if(remainingList.length > 0) {
    return totalForArray(currentTotal, remainingList);
  }

  // Unless of course the list is empty, in which case we can just return
  // the currentTotal value.
  else {
    return currentTotal;
  }
}
```

A word of caution: Recursion will make your programs more readable, and it is essential to programming in a functional style. However, in some languages (including JavaScript), you'll run into problems when your program makes a large number of recursive calls in a single operation (at the time of writing, "large" is about 10,000 calls in Chrome, 50,000 in Firefox and 11,000 in Node.js[12]). The details are beyond the scope of this article, but the gist is that, at least until ECMAScript 6 is released[13], JavaScript doesn't support something called "~~tail recursion~~[14]," which is a more efficient form of recursion. This is an advanced topic and won't come up very often, but it's worth knowing.

With that out of the way, remember that we needed to calculate the total temperature from an array of temperatures in order to then calculate the average. Now, instead of looping over each item in the `temperatures` array, we can simply write this:

```
var totalTemp = totalForArray(0, temperatures);
```

If you're purist, you might say that our `totalForArray` function could be broken down even further. For example, the task of adding two numbers together will probably come up in other parts of your application and subsequently should really be its own function.

```
function addNumbers(a, b) {
  return a + b;
}
```

Now, our `totalForArray` function looks like this:

```
function totalForArray(currentTotal, arr) {
  currentTotal = addNumbers(currentTotal, arr[0]);

  var remainingArr = arr.slice(1);

  if(remainingArr.length > 0) {
    return totalForArray(currentTotal, remainingArr);
  }
  else {
    return currentTotal;
  }
}
```

Excellent! Returning a single value from an array is fairly common in functional programming, so much so that it has a special name, "reduction," which you'll more commonly hear as a verb, like when you "reduce an array to a single value." JavaScript has a special method just for performing this common task. Mozilla Developer Network provides a full explanation[15], but for our purposes it's as simple as this:

```
// The reduce method takes a function as its first argument, and that function
// accepts both the current item in the list and the current total result from
// whatever calculation you're performing.
var totalTemp = temperatures.reduce(function(previousValue, currentValue){
  // After this calculation is returned, the next currentValue will be
  // previousValue + currentValue, and the next previousValue will be the
  // next item in the array.
  return previousValue + currentValue;
});
```

But, hey, since we've already defined an `addNumber` function, we can just use that instead.

```
var totalTemp = temperatures.reduce(addNumbers);
```

In fact, because totalling up an array is so cool, let's put that into its own function so that we can use it again without having to remember all of that confusing stuff about reduction and recursion.

```
function totalForArray(arr) {
  return arr.reduce(addNumbers);
}

var totalTemp = totalForArray(temperatures);
```

Ah, now *that* is some readable code! Just so you know, methods such as `reduce` are common in most functional programming languages. These helper methods that perform actions on arrays in lieu of looping are often called "higher-order functions."

Moving right along, the second task we listed was calculating an average. This is pretty easy.

```
function average(total, count) {
  return total / count;
}
```

How might we go about getting the average for an entire array?

```
function averageForArray(arr) {
  return average(totalForArray(arr), arr.length);
}

var averageTemp = averageForArray(temperatures);
```

Hopefully, you're starting to see how to combine functions to perform more complex tasks. This is possible because we're following the rules set out at the beginning of this article — namely, that our functions must always accept arguments and return data. Pretty awesome.

Lastly, we wanted to retrieve a single property from an array of objects. Instead of showing you more examples of recursion, I'll cut to the chase and clue you in on another built-in JavaScript method: map[16]. This method is for when you have an array with one structure and need to map it to another structure, like so:

```
// The map method takes a single argument, the current item in the list. Check
// out the link above for more complete examples.
var allTemperatures = data.map(function(item) {
  return item.temperatures;
});
```

That's pretty cool, but pulling a single property from a collection of objects is something you'll be doing all the time, so let's make a function just for that.

```
// Pass in the name of the property that you'd like to retrieve
function getItem(propertyName) {
  // Return a function that retrieves that item, but don't execute the function.
  // We'll leave that up to the method that is taking action on items in our
  // array.
  return function(item) {
    return item[propertyName];
  }
}
```

Check it out: We've made a function that returns a function! Now we can pass it to the `map` method like this:

```
var temperatures = data.map(getItem('temperature'));
```

In case you like details, the reason we can do this is because, in JavaScript, functions are “first-class objects,” which basically means that you can pass around functions just like any other value. While this is a feature of many programming languages, it’s a requirement of any language that can be used in a functional style. Incidentally, this is also the reason you can do stuff like `$('#my-element').on('click', function(e) … )`. The second argument in the `on` method is a `function`, and when you pass functions as arguments, you’re using them just like you would use values in imperative languages. Pretty neat.

Finally, let’s wrap the call to `map` in its own function to make things a little more readable.

```
function pluck(arr, propertyName) {
  return arr.map(getItem(propertyName));
}

var allTemperatures = pluck(data, 'temperatures');
```

All right, now we have a toolkit of generic functions that we can use anywhere in our application, even in other projects. We can tally up the items in an array, get the average value of an array, and make new arrays by plucking properties from lists of objects. Last but not least, let’s return to our original problem:

```
var data = [
  {
    name: "Jamestown",
    population: 2047,
    temperatures: [-34, 67, 101, 87]
  },
  …
];
```

We need to transform an array of objects like the one above into an array of `x, y` pairs, like this:

```
[
  [75, 1000000],
  …
];
```

Here, `x` is the average temperature, and `y` is the total population. First, let's isolate the data that we need.

```
var populations = pluck(data, 'population');
var allTemperatures = pluck(data, 'temperatures');
```

Now, let's make an array of averages. Remember that the function we pass to `map` will be called on each item in the array; so, the returned value of that passed function will be added to a new array, and that new array will ultimately be assigned to our `averageTemps` variable.

```
var averageTemps = allTemperatures.map(averageForArray);
```

So far so good. But now we have two arrays:

```
// populations
[2047, 3568, 1000000]

// averageTemps
[55.25, 5.5, 75]
```

Obviously, we want only one array, so let's write a function to combine them. Our function should make sure that the item at index `0` in the first array is paired with the item at index `0` in the second array, and so on for indexes `1` to `n` (where `n` is the total number of items in the array).

```
function combineArrays(arr1, arr2, finalArr) {
  // Just so we don't have to remember to pass an empty array as the third
  // argument when calling this function, we'll set a default.
  finalArr = finalArr || [];

  // Push the current element in each array into what we'll eventually return
  finalArr.push([arr1[0], arr2[0]]);

  var remainingArr1 = arr1.slice(1),
      remainingArr2 = arr2.slice(1);

  // If both arrays are empty, then we're done
  if(remainingArr1.length === 0 && remainingArr2.length === 0) {
    return finalArr;
  }
  else {
    // Recursion!
    return combineArrays(remainingArr1, remainingArr2, finalArr);
  }
};

var processed = combineArrays(averageTemps, populations);
```

Or, because one-liners are fun:

```
var processed = combineArrays(pluck(data, 'temperatures').map(averageForArray), pluck(

// [
//  [ 55.25, 2047 ],
//  [ 5.5, 3568 ],
//  [ 75, 1000000 ]
// ]
```

Let's Get Real

Last but not least, let's look at one more real-world example, this time adding to our functional toolbelt with Underscore.js[17], a JavaScript library that provides a number of great functional programming helpers. We'll pull data from a platform for conflict and disaster information that I've been working on named CrisisNET[18], and we'll use the fantastic D3[19] library to visualize that data.

The goal is to give people coming to CrisisNET's home page a quick snapshot of the types of information in the system. To demonstrate this, we could count the number of documents from the API that are assigned to a particular category, like "physical violence" or "armed conflict." This way, the user can see how much information is available on the topics they find most interesting.

A bubble chart might be a good fit, because they are often used to represent the relative sizes of large groups of people. Fortunately, D3 has a built-in visualization named `pack` for just this purpose. So, let's create a graph with `pack` that shows the number of times that a given category's name appears in a response from CrisisNET's API.

Before we go on, note that D3 is a complex library that warrants its own tutorial (or many tutorials, for that matter). Because this article is focused on functional programming, we won't spend a lot of time on how D3 works. But don't worry — if you're not already familiar with the library, you should be able to copy and paste the code snippets specific to D3 and dig into the details another time. Scott Murray's D3 tutorials[20] are a great resource if you're interested in learning more.

Moving along, let's first make sure we have a DOM element, so that D3 has some place to put the chart it will generate with our data.

```
<div id="bubble-graph"></div>
```

Now, let's create our chart and add it to the DOM.

```
// width of chart
var diameter = 960,
    format = d3.format(",d"),
    // creates an ordinal scale with 20 colors. See D3 docs for hex values
    color = d3.scale.category20c(),

// chart object to which we'll be adding data
var bubble = d3.layout.pack()
  .sort(null)
  .size([diameter, diameter])
  .padding(1.5);

// Add an SVG to the DOM that our pack object will use to draw the
// visualization.
var svg = d3.select("#bubble-graph").append("svg")
  .attr("width", diameter)
  .attr("height", diameter)
  .attr("class", "bubble");
```

The `pack` object takes an array of objects in this format:

```
{
  children: [
    {
      className: ,
      package: "cluster",
      value:
    }
  ]
}
```

CrisisNET's data API returns information in this format:

```
{
  data: [
    {
      summary: "Example summary",
      content: "Example content",
      …
      tags: [
        {
          name: "physical-violence",
          confidence: 1
        }
      ]
    }
  ]
}
```

We see that each document has a `tags` property, and that property contains an array of items. Each tag item has a `name` property, which is what we're after. We need to find each unique tag name in CrisisNET's API response and count the number of times that tag name appears. Let's start by isolating the information we need using the `pluck` function that we created earlier.

```
var tagArrays = pluck(data, 'tags');
```

This gives us an array of arrays, like this:

```
[
  [
    {
      name: "physical-violence",
      confidence: 1
    }
  ],
  [
    {
      name: "conflict",
      confidence: 1
    }
  ]
]
```

However, what we really want is one array with every tag in it. So, let's use a handy function from Underscore.js named flatten[21]. This will take values from any nested arrays and give us an array that is one level deep.

```
var tags = _.flatten(tagArrays);
```

Now, our array is a little easier to deal with:

```
[
  {
    name: "physical-violence",
    confidence: 1
  },
  {
    name: "conflict",
    confidence: 1
  }
]
```

We can use `pluck` again to get the thing we really want, which is a simple list of only the tag names.

```
var tagNames = pluck(tags, 'name');
```

```
[
  "physical-violence",
  "conflict"
]
```

Ah, that's better.

Now we're down to the relatively straightforward tasks of counting the number of times each tag name appears in our list and then transforming that list into the structure required by the D3 `pack` layout that we created earlier. As you've probably noticed, arrays are a pretty popular data structure in functional programming — most of the tools are designed with arrays in mind. As a first step, then, we'll create an array like this:

```
[
  [ "physical-violence", 10 ],
  [ "conflict", 27 ]
]
```

Here, each item in the array has the tag name at index `0` and that tag's total count at index `1`. We want only one array for each unique tag name, so let's start by creating an array in which each tag name appears only once. Fortunately, an Underscore.js method exists just for this purpose.

```
var tagNamesUnique = _.uniq(tagNames);
```

Let's also get rid of any `false-y` (`false`, `null`, `""`, etc.) values using another handy Underscore.js function.

```
tagNamesUnique = _.compact(tagNamesUnique);
```

From here, we can write a function that generates our arrays using another built-in JavaScript collection method, named filter[22], that filters an array based on a condition.

```
function makeArrayCount(keys, arr) {

  // for each of the unique tagNames
  return keys.map(function(key) {
    return [
      key,
      // Find all the elements in the full list of tag names that match this key
      // and count the size of the returned array.
      arr.filter(function(item) { return item === key; }).length
    ]
  });

}
```

We can now easily create the data structure that `pack` requires by mapping our list of arrays.

```
var packData = makeArrayCount(tagNamesUnique, tagNames).map(function(tagArray) {
  return {
    className: tagArray[0],
    package: "cluster",
    value: tagArray[1]
  }
});
```

Finally, we can pass our data to D3 and generate DOM nodes in our SVG, one circle for each unique tag name, sized relative to the total number of times that tag name appeared in CrisisNET's API response.

```
function setGraphData(data) {
  var node = svg.selectAll(".node")
    // Here's where we pass our data to the pack object.
    .data(bubble.nodes(data)
    .filter(function(d) { return !d.children; }))
    .enter().append("g")
    .attr("class", "node")
    .attr("transform", function(d) { return "translate(" + d.x + "," + d.y + ")"; });

  // Append a circle for each tag name.
  node.append("circle")
    .attr("r", function(d) { return d.r; })
    .style("fill", function(d) { return color(d.className); });

  // Add a label to each circle, using the tag name as the label's text
  node.append("text")
    .attr("dy", ".3em")
    .style("text-anchor", "middle")
    .style("font-size", "10px")
    .text(function(d) { return d.className } );
}
```

Putting it all together, here's the `setGraphData` and `makeArray` functions in context, including a call to CrisisNET's API using jQuery (you'll need to get an API key[23]). I've also posted a fully working example on GitHub[24].

```
function processData(dataResponse) {
  var tagNames = pluck(_.flatten(pluck(dataResponse.data, 'tags')), 'name');
  var tagNamesUnique = _.uniq(tagNames);

  var packData = makeArrayCount(tagNamesUnique, tagNames).map(function(tagArray) {
    return {
      className: tagArray[0],
      package: "cluster",
      value: tagArray[1]
    }
  });

  return packData;
}

function updateGraph(dataResponse) {
  setGraphData(processData(dataResponse));
}

var apikey = // Get an API key here: http://api.crisis.net
var dataRequest = $.get('http://api.crisis.net/item?limit=100&apikey=' + apikey);

dataRequest.done( updateGraph );
```

That was a pretty deep dive, so congratulations on sticking with it! As I mentioned, these concepts can be challenging at first, but resist the temptation to hammer out `for` loops for the rest of your life.

Within a few weeks of using functional programming techniques, you'll quickly build up a set of simple, reusable functions that will dramatically improve the readability of your applications. Plus, you'll be able to manipulate data structures significantly more quickly, knocking out what used to be 30 minutes of frustrating debugging in a couple lines of code. Once your data has been formatted correctly, you'll get to spend more time on the fun part: making the visualization look awesome!

(al, il)

FOOTNOTES

1 http://clojure.org/

2 http://www.akamai.com/

3 http://www.infoq.com/articles/twitter-java-use

4 http://www.scala-lang.org/

5 http://www.haskell.org/haskellwiki/Haskell

6 https://github.com/kachayev/fn.py

7 http://underscorejs.org/

8 http://baconjs.github.io/

9 http://en.wikipedia.org/wiki/Lambda_calculus

10 http://mathoverflow.net/questions/11916/is-functional-programming-a-branch-of-mathematics

11 http://programmers.stackexchange.com/a/154523

12 http://www.2ality.com/2014/04/call-stack-size.html

13 https://people.mozilla.org/~jorendorff/es6-draft.html#sec-tail-position-calls

14 http://cs.stackexchange.com/a/7814

15 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce

16 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

17 http://underscorejs.org/

18 http://crisis.net

19 http://d3js.org

20 http://alignedleft.com/tutorials/d3

21 http://underscorejs.org/#flatten

22 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

23 http://api.crisis.net

24 https://github.com/gati/lilfunc/blob/master/smashing.html

Jonathon Morgan

Jonathon is an entrepreneur and technologist focused on platform architecture, product development, and data science. He currently leads the technology team behind CrisisNET, a platform for global crisis data from Ushahidi. Before CrisisNET Jonathon was CTO of venture-backed travel startup SA Trails, and lead technologist for design/build agency Bright & Shiny. He regularly writes and speaks about data for social good, software architecture and emerging technologies, and tweets @jonathonmorgan.