CHAPTER 6
Recursion

This chapter is a transitional chapter meant to smooth the way from thinking about functions to thinking about a deeper understanding of a functional style, including when to break from it and why doing so is sometimes a good idea. Specifically, this chapter covers recursion, or functions calling themselves directly or through other functions.

Self-Absorbed Functions (Functions That Call Themselves)

Historically, recursion and functional programming were viewed as related, or at least they were often taught together. Throughout this chapter, I'll explain how they're related, but for now, I can say that an understanding of recursion is important to functional programming for three reasons:

- Recursive solutions involve the use of a single abstraction applied to subsets of a common problem.
- Recursion can hide mutable state.
- Recursion is one way to implement laziness and infinitely large structures.

If you think about the essential nature of a function, `myLength`, that takes an array and returns its length (i.e., number of elements), then you might land on the following description:[1]

1. Start with a size of zero.
2. Loop through the array, adding one to the size for each entry.

1. You might actually think, "why not just use the `length` field on the array?" While this kind of pragmatic thinking is extremely important for building great systems, it's not helpful for learning recursion.

3. If you get to the end, then the size is the length.

This is a correct description of `myLength`, but it doesn't involve recursive thinking. Instead, a recursive description would be as follows:

1. An array's length is zero if it's empty.
2. Add one to the result of `myLength` with the remainder of the array.

I can directly encode these two rules in the implementation of `myLength`, as shown here:

```
function myLength(ary) {
  if (_.isEmpty(ary))
    return 0;
  else
    return 1 + myLength(_.rest(ary));
}
```

Recursive functions are very good at building values. The trick in implementing a recursive solution is to recognize that certain values are built from subproblems of a larger problem. In the case of `myLength`, the total solution can really be seen as adding the lengths of some number of single-element arrays with the length of an empty array. Since `myLength` calls itself with the result of `_.rest`, each recursive call gets an array that is one element shorter, until the last call gets an empty array (see Figure 6-1).

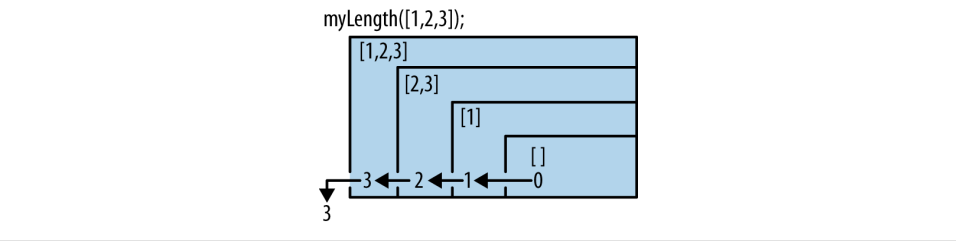

Figure 6-1. A recursive `myLength` that "consumes" an array

Observe the operation of `myLength` below:

```
myLength(_.range(10));
//=> 10

myLength([]);
//=> 0

myLength(_.range(1000));
//=> 1000
```

It's important to know that for minimal confusion, recursive functions should not change the arguments given them:

```
var a = _.range(10);

myLength(a);
//=> 10

a;
//=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

While a recursive function may *logically* consume its input arguments, it should never actually do so. While `myLength` built up an integer return value based on its input, a recursive function can build any type of legal value, including arrays and objects. Consider, for example, a function called `cycle` that takes a number and an array and builds a new array filled with the elements of the input array, repeated the specified number of times:

```
function cycle(times, ary) {
  if (times <= 0)
    return [];
  else
    return cat(ary, cycle(times - 1, ary));
}
```

The form of the `cycle` function looks similar to the `myLength` function. That is, while `myLength` "consumed" the input array, `cycle` "consumes" the repeat count. Likewise, the value built up on each step is the new cycled array. This consume/build action is shown in Figure 6-2.

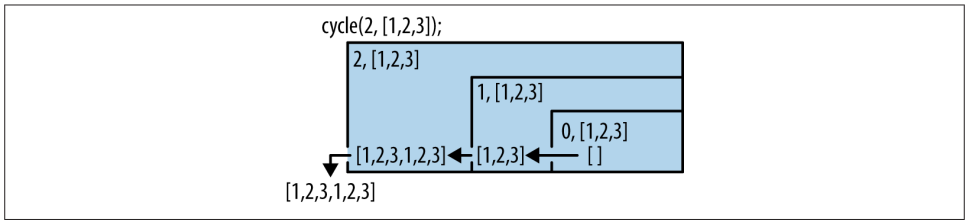

Figure 6-2. A recursive cycle that builds an array

Here's `cycle` in action:

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

_.take(cycle(20, [1,2,3]), 11);
//=> [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2]
```

Another self-recursive function that I'll create is called `unzip`, which is the inverse of Underscore's `_.zip` function, constrained to pairs, shown here:

```
_.zip(['a', 'b', 'c'], [1, 2, 3]);

//=> [['a', 1], ['b', 2], ['c', 3]]
```

Underscore's `_.zip` takes two arrays and pairs each element in the first array with each corresponding element in the second array. To implement a function that "unzips" arrays like those generated by `_.zip` requires that I think about the "pairness" of the array needing unzipping. In other words, if I think about the basic case, one array needing unzipping, then I can begin to deconstruct how to solve the problem as a whole:

```
var zipped1 = [['a', 1]];
```

Even more basic than `zipped1` would be the empty array `[]`, but an unzipped empty array is the resulting array `[[],[]]` (that seems like a good candidate for a terminating case, so put that in the back of your mind for now). The array `zipped1` is the simplest interesting case and results in the unzipped array `[['a'], [1]]`. So given the terminating case `[[],[]]` and the base case `zipped`, how can I get to `[['a'], [1]]`?

The answer is as simple as a function that makes an array of the first element in `zipped1` and puts it into the first array in the terminating case and the second element in `zipped1`, and puts that into the second array of the terminating case. I can abstract this operation in a function called `constructPair`:

```
function constructPair(pair, rests) {
  return [construct(_.first(pair), _.first(rests)),
          construct(second(pair),  second(rests))];
}
```

While the operation of `constructPair` is not enough to give me general "unzippability," I can achieve an unzipped version of `zipped1` manually by using it and the empty case:

```
constructPair(['a', 1], [[],[]]);
//=> [['a'], [1]]

_.zip(['a'], [1]);
//=> [['a', 1]]

_.zip.apply(null, constructPair(['a', 1], [[],[]]));
//=> [['a', 1]]
```

Likewise, I can gradually build up an unzipped version of a larger zipped array using `constructPair`, as shown here:

```
constructPair(['a', 1],
  constructPair(['b', 2],
    constructPair(['c', 3], [[],[]])));

//=> [['a','b','c'],[1,2,3]]
```

Graphically, these manual steps are shown in Figure 6-3.

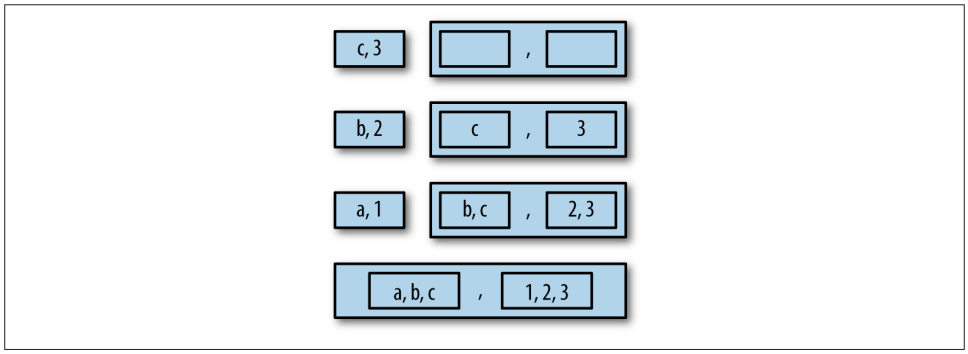

Figure 6-3. Illustrating the operation of `constructPair` graphically

So using the knowledge of how `constructPair` works, I can now build a self-recursive function `unzip`:

```
function unzip(pairs) {
  if (_.isEmpty(pairs)) return [[],[]];

  return constructPair(_.first(pairs), unzip(_.rest(pairs)));
}
```

The recursive call in `unzip` walks the given array of zipped pairs until it gets to an empty array. It then walks back down the subarrays, using `constructPair` along the way to build an unzipped representation of the array. Having implemented `unzip`, I should be able to "undo" the result of a call to `_.zip` that has built an array of pairs:

```
unzip(_.zip([1,2,3],[4,5,6]));
//=> [[1,2,3],[4,5,6]]
```

All instances of `myLength`, `cycle`, and `unzip` were examples of *self-recursion* (or, in other words, functions that call themselves). The rules of thumb when writing self-recursive functions are as follows(Touretzky 1990):

- Know when to stop
- Decide how to take one step
- Break the problem into that step and a smaller problem

Table 6-1 presents a tabular way of observing how these rules operate.

Table 6-1. The rules of self-recursion

Function	Stop When	Take One Step	Smaller Problem
`myLength`	`_.isEmpty`	`1 + ...`	`_.rest`
`cycle`	`times <= 0`	`cat(array ...`	`times - 1`
`unzip`	`_.isEmpty`	`constructPair(_.first ...`	`_.rest`

Observing these three rules will provide a template for writing your own recursive functions. To illustrate that self-recursive functions of any complexity fall into this pattern, I'll run through a more complex example and explain the similarities along the way.

Graph Walking with Recursion

Another suitable problem solved by recursion in an elegant way is the task of walking the nodes in a graph-like data structure. If I wanted to create a library for navigating a graph-like structure, then I would be hard pressed to find a solution more elegant than a recursive one. A graph that I find particularly interesting is a (partial) graph of the programming languages that have influenced JavaScript either directly or indirectly.

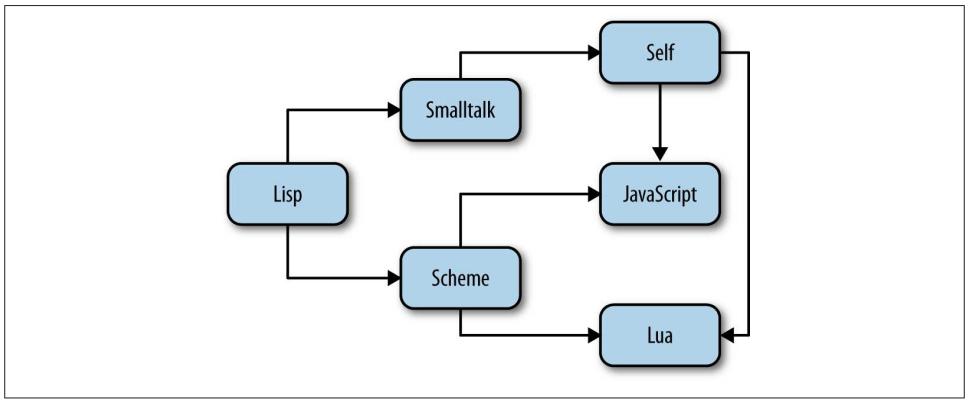

Figure 6-4. A partial graph of programming language influences

I could use a class-based or object-based representation, where each language and connection is represented by objects of type `Node` and `Arc`, but I think I'd prefer to start with something simple, like an array of arrays of strings:

```
var influences = [
  ['Lisp', 'Smalltalk'],
  ['Lisp', 'Scheme'],
  ['Smalltalk', 'Self'],
  ['Scheme', 'JavaScript'],
  ['Scheme', 'Lua'],
  ['Self', 'Lua'],
  ['Self', 'JavaScript']];
```

Each nested array in `influences` represents a connection of "influencer" to "influenced" (e.g., Lisp influenced Smalltalk) and encodes the graph shown in Figure 6-4. A recursive function, `nexts`, is defined recursively as follows(Paulson 1996):

```
function nexts(graph, node) {
  if (_.isEmpty(graph)) return [];
```

```
  var pair = _.first(graph);
  var from = _.first(pair);
  var to   = second(pair);
  var more = _.rest(graph);

  if (_.isEqual(node, from))
    return construct(to, nexts(more, node));
  else
    return nexts(more, node);
}
```

The function `nexts` walks the graph recursively and builds an array of programming languages influenced by the given node, as shown here:

```
nexts(influences, 'Lisp');
//=> ["Smalltalk", "Scheme"]
```

The recursive call within `nexts` is quite different than what you've seen so far; there's a recursive call in both branches of the `if` statement. The "then" branch of `nexts` deals directly with the target node in question, while the `else` branch ignores unimportant nodes.

Table 6-2. The rules of self-recursion according to nexts

Function	Stop When	Take One Step	Smaller Problem
`nexts`	`_.isEmpty`	`construct(...)`	`_.rest`

It would take very little work to make `nexts` take and traverse multiple nodes, but I leave that as an exercise to the reader. Instead, I'll now cover a specific type of graph-traversal recursive algorithm called depth-first search.

Depth-First Self-Recursive Search with Memory

In this section I'll talk briefly about graph searching and provide an implementation of a depth-first search function. In functional programming, you'll often need to search a data structure for a piece of data. In the case of hierarchical graph-like data (like `influences`), the search solution is naturally a recursive one. However, to find any given node in a graph, you'll need to (potentially) visit every node in the graph to see if it's the one you're looking for. One node traversal strategy called depth-first visits every leftmost node in a graph before visiting every rightmost node (as shown in Figure 6-5).

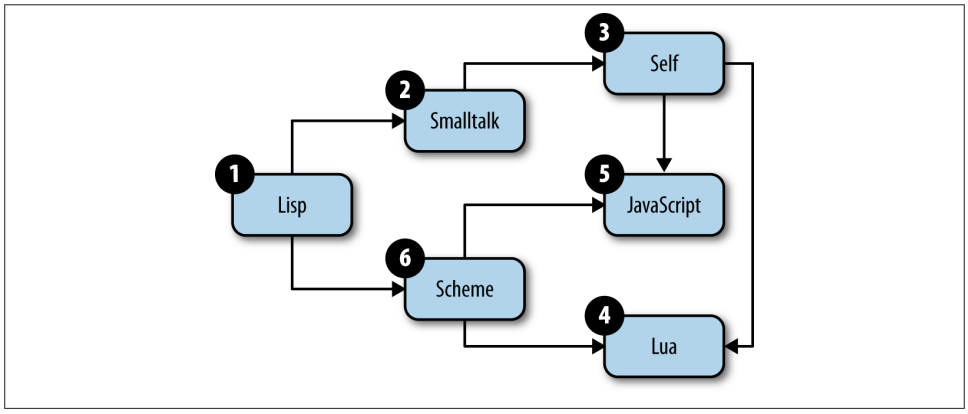

Figure 6-5. Traversing the influences graph depth-first

Unlike the previous recursive implementations, a new function `depthSearch` should maintain a memory of nodes that it's already seen. The reason, of course, is that another graph might have cycles in it, so without memory, a "forgetful" search will loop until JavaScript blows up. However, because a self-recursive call can only interact from one invocation to another via arguments, the memory needs to be sent from one call to the next via an "accumulator." An accumulator argument is a common technique in recursion for communicating information from one recursive call to the next. Using an accumulator, the implementation of `depthSearch` is as follows:

```
function depthSearch(graph, nodes, seen) {
  if (_.isEmpty(nodes)) return rev(seen);

  var node = _.first(nodes);
  var more = _.rest(nodes);

  if (_.contains(seen, node))
    return depthSearch(graph, more, seen);
  else
    return depthSearch(graph,
                       cat(nexts(graph, node), more),
                       construct(node, seen));
}
```

As you'll notice, the third parameter, `seen`, is used to hold the accumulation of seen nodes to avoid revisiting old nodes and their children. A usage example of `depthSearch` is as follows:

```
depthSearch(influences, ['Lisp'], []);
//=> ["Lisp", "Smalltalk", "Self", "Lua", "JavaScript", "Scheme"]

depthSearch(influences, ['Smalltalk', 'Self'], []);
//=> ["Smalltalk", "Self", "Lua", "JavaScript"]
```

```
depthSearch(construct(['Lua','Io'], influences), ['Lisp'], []);
//=> ["Lisp", "Smalltalk", "Self", "Lua", "Io", "JavaScript", "Scheme"]
```

You may have noticed that the `depthSearch` function doesn't actually do anything. Instead, it just builds an array of the nodes that it would do something to (if it did anything) in depth-first order. That's OK because later I'll re-implement a depth-first strategy using functional composition and mutual recursion. First, let me take a moment to talk about "tail calls."

Tail (self-)recursion

While the general form of `depthSearch` looks very similar to the functions that came before, there is one difference that is key. To highlight what I mean, consider Table 6-3.

Table 6-3. The rules of self-recursion according to `depthSearch`

Function	Stop When	Take One Step	Smaller Problem
`nexts`	`_.isEmpty`	`construct(...`	`_.rest`
`depthSearch`	`_.isEmpty`	`depthSearch(more...`	`depthSearch(cat...`

The clear difference is that the "take one step" and "smaller problem" elements of `depthSearch` are recursive calls. When either or both of these elements are recursive calls, then the function is known as tail-recursive. In other words, the last action that happens in the function (besides returning a termination element) is a recursive call. Since the last call in `depthSearch` is a recursive call, there is no way that the function body will ever be used again. A language like Scheme takes advantage of this fact to deallocate the resources used in a tail-recursive function body.

A reimplementation of `myLength` using a tail-recursive call is as follows:

```
function tcLength(ary, n) {
  var l = n ? n : 0;

  if (_.isEmpty(ary))
    return l;
  else
    return tcLength(_.rest(ary), l + 1);
}

tcLength(_.range(10));
//=> 10
```

By contrast, the recursive call in `myLength` (i.e., `1 + ...`) revisits the function body to perform that final addition. Perhaps one day JavaScript engines will optimize tail-recursive functions to preserve memory. Until that time, we're cursed to blow the call

stack on deeply recursive calls.[2] However, as you'll see later in this chapter, the tail position of a function is still interesting.

Recursion and Composing Functions: Conjoin and Disjoin

Throughout this book, I've implemented some common function composition functions—`curry1`, `partial`, and `compose`—but I didn't describe how to create your own. Fortunately, the need to create composition functions is rare, as much of the composition capabilities needed are provided by Underscore or are implemented herein. However, in this section, I'll describe the creation of two new combinators, `orify` and `andify`, implemented using recursion.

Recall that I created a `checker` function way back in Chapter 4 that took some number of predicates and returned a function that determined if they all returned truthiness for *every* argument supplied. I can implement the spirit of `checker` as a recursive function called `andify`:[3]

```
function andify(/* preds */) {
  var preds = _.toArray(arguments);

  return function(/* args */) {
    var args = _.toArray(arguments);

    var everything = function(ps, truth) {
      if (_.isEmpty(ps))
        return truth;
      else
        return _.every(args, _.first(ps))
               && everything(_.rest(ps), truth);
    };

    return everything(preds, true);
  };
}
```

Take note of the recursive call in the function returned by `andify`, as it's particularly interesting. Because the logical and operator, `&&`, is "lazy," the recursive call will never happen should the `_.every` test fail. This type of laziness is called "short-circuiting," and it is useful for avoiding unnecessary computations. Note that I use a local function, `everything`, to consume the predicates given in the original call to `andify`. Using a nested function is a common way to hide accumulators in recursive calls.

2. The ECMAScript 6 proposal currently has a section for tail-call optimization, so cross your fingers... I'll cross mine. See *http://wiki.ecmascript.org/doku.php?id=harmony:proper_tail_calls*.

3. A short-circuiting `andify` function can also be implemented via Underscore's `every` function. Can you see how?

Observe the action of `andify` here:

```
var evenNums = andify(_.isNumber, isEven);

evenNums(1,2);
//=> false

evenNums(2,4,6,8);
//=> true

evenNums(2,4,6,8,9);
//=> false
```

The implementation of `orify` is almost exactly like the form of `andify`, except for some logic reversals:[4]

```
function orify(/* preds */) {
  var preds = _.toArray(arguments);

  return function(/* args */) {
    var args = _.toArray(arguments);

    var something = function(ps, truth) {
      if (_.isEmpty(ps))
        return truth;
      else
        return _.some(args, _.first(ps))
               || something(_.rest(ps), truth);
    };

    return something(preds, false);
  };
}
```

Like `andify`, should the `_.some` function ever succeed, the function returned by `orify` short-circuits (i.e., any of the arguments match any of the predicates). Observe:

```
var zeroOrOdd = orify(isOdd, zero);

zeroOrOdd();
//=> false

zeroOrOdd(0,2,4,6);
//=> true

zeroOrOdd(2,4,6);
//=> false
```

4. The `orify` function can also be implemented via Underscore's `some` function. Can you see how?

This ends my discussion of self-recursive functions, but I'm not done with recursion quite yet. There is another way to achieve recursion and it has a catchy name: mutual recursion.

Codependent Functions (Functions Calling Other Functions That Call Back)

Two or more functions that call each other are known as mutually recursive. Two very simple mutually recursive functions are the predicates to check for even and odd numbers, shown here:

```
function evenSteven(n) {
  if (n === 0)
    return true;
  else
    return oddJohn(Math.abs(n) - 1);
}

function oddJohn(n) {
  if (n === 0)
    return false;
  else
    return evenSteven(Math.abs(n) - 1);
}
```

The mutually recursive calls bounce back and forth between each other, decrementing some absolute value until one or the other reaches zero. This is a fairly elegant solution that works as you expect:

```
evenSteven(4);
//=> true

oddJohn(11);
//=> true
```

If you adhere to the strict use of higher-order functions, then you're likely to encounter mutually exclusive functions more often. Take, for example, the `_.map` function. A function that calls `_.map` with itself is a mind-bendingly indirect way to perform mutual recursion. A function to flatten an array to one level serves as an example:[5]

```
function flat(array) {
  if (_.isArray(array))
    return cat.apply(cat, _.map(array, flat));
  else
    return [array];
}
```

5. A better solution is to use Underscore's `flatten` function.

The operation of `flat` is a bit subtle, but the point is that in order to flatten a nested array, it builds an array of each of its nested elements and recursively concatenates each on the way back. Observe:

```
flat([[1,2],[3,4]]);
//=> [1, 2, 3, 4]

flat([[1,2],[3,4,[5,6,[[[7]]],8]]]);
//=> [1, 2, 3, 4, 5, 6, 7, 8]
```

Again, this is a fairly obscure use of mutual recursion, but one that fits well with the use of higher-order functions.

Deep Cloning with Recursion

Another example where recursion seems like a good fit is to implement a function to "deep" clone an object. Underscore has a `_.clone` function, but it's "shallow" (i.e., it only copies the objects at the first level):

```
var x = [{a: [1, 2, 3], b: 42}, {c: {d: []}}];

var y = _.clone(x);

y;
//=> [{a: [1, 2, 3], b: 42}, {c: {d: []}}];

x[1]['c']['d'] = 1000000;

y;
//=> [{a: [1, 2, 3], b: 42}, {c: {d: 1000000}}];
```

While in many cases, `_.clone` will be useful, there are times when you'll really want to clone an object and all of its subobjects.[6] Recursion is a perfect task for this because it allows us to walk every object in a nested fashion, copying along the way. A recursive implementation of `deepClone`, while not robust enough for production use, is shown here:

```
function deepClone(obj) {
  if (!existy(obj) || !_.isObject(obj))
    return obj;

  var temp = new obj.constructor();
  for (var key in obj)
    if (obj.hasOwnProperty(key))
      temp[key] = deepClone(obj[key]);
```

6. I'm using the term "clone" in the way often seen in JavaScript circles. In other prototypal languages (e.g., Self or Io) a clone operation delegates to the original, cloned object until a change is made, whereby a copy occurs.

```
  return temp;
}
```

When `deepClone` encounters a primitive like a number, it simply returns it. However, when it encounters an object, it treats it like an associative structure (hooray for generic data representations) and recursively copies all of its key/value mappings. I chose to use the `obj.hasOwnProperty(key)` to ensure that I do not copy fields from the prototype. I tend to use objects as associative data structures (i.e., maps) and avoid putting data onto the prototype unless I must. The use of `deepClone` is as follows:

```
var x = [{a: [1, 2, 3], b: 42}, {c: {d: []}}];

var y = deepClone(x);

_.isEqual(x, y);
//=> true

y[1]['c']['d'] = 42;

_.isEqual(x, y);
//=> false
```

The implementation of `deepClone` isn't terribly interesting except for the fact that JavaScript's everything-is-an-object foundation really allows the recursive solution to be compact and elegant. In the next section, I'll re-implement `depthSearch` using mutual recursion, but one that actually does something.

Walking Nested Arrays

Walking nested objects like in `deepClone` is nice, but not frequently needed. Instead, a far more common occurrence is the need to traverse an array of nested arrays. Very often you'll see the following pattern:

```
doSomethingWithResult(_.map(someArray, someFun));
```

The result of the call to `_.map` is then passed to another function for further processing. This is common enough to warrant its own abstraction, I'll call it `visit`, implemented here:

```
function visit(mapFun, resultFun, array) {
  if (_.isArray(array))
    return resultFun(_.map(array, mapFun));
  else
    return resultFun(array);
}
```

The function `visit` takes two functions in addition to an array to process. The `mapFun` argument is called on each element in the array, and the resulting array is passed to `resultFun` for final processing. If the thing passed in `array` is not an array, then I just run the `resultFun` on it. Implementing functions like this is extremely useful in light

of partial application because one or two functions can be partially applied to form a plethora of additional behaviors from `visit`. For now, just observe how `visit` is used:

```
visit(_.identity, _.isNumber, 42);
//=> true

visit(_.isNumber, _.identity, [1, 2, null, 3]);
//=> [true, true, false, true]

visit(function(n) { return n*2 }, rev, _.range(10));
//=> [18, 16, 14, 12, 10, 8, 6, 4, 2, 0]
```

Using the same principle behind `flat`, I can use `visit` to implement a mutually recursive version of `depthSearch` called `postDepth`:[7]

```
function postDepth(fun, ary) {
  return visit(partial1(postDepth, fun), fun, ary);
}
```

The reason for the name `postDepth` is that the function performs a depth-first traversal of any array performing the `mapFun` on each element *after* expanding its children. A related function, `preDepth`, performs the `mapFun` call *before* expanding an element's children and is implemented as follows:

```
function preDepth(fun, ary) {
  return visit(partial1(preDepth, fun), fun, fun(ary));
}
```

There's plenty of `fun` to go around in the case of pre-order depth-first search, but the principle is sound; just perform the function call before moving onto the other elements in the array. Let's see `postDepth` in action:

```
postDepth(_.identity, influences);
//=> [['Lisp','Smalltalk'], ['Lisp','Scheme'], ...
```

Passing the `_.identity` function to the `*Depth` functions returns a copy of the `influences` array. The execution scheme of the mutually recursive functions `evenSteven`, `oddJohn`, `postDepth` and `visit` is itself a graph-like model, as shown in Figure 6-6.

7. The `JSON.parse` method takes an optional "reviver" function and operates similarly to `postDepth`. That is, after a form is parsed, `JSON.parse` passes to the reviver the associated key with the parsed data, and whatever the reviver returns becomes the new value. People have been known to use the reviver for numerous reasons, but perhaps the most common is to generate `Date` objects from date-encoded strings.

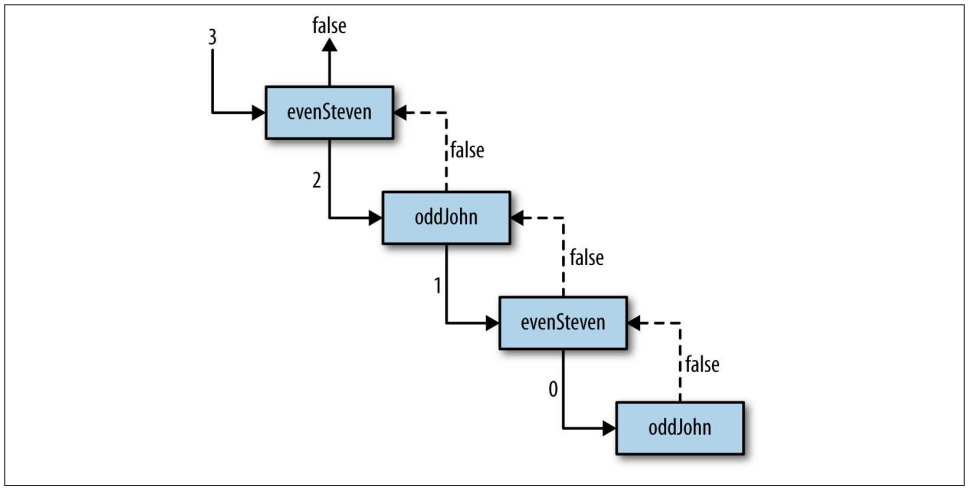

Figure 6-6. Mutually recursive functions execute in a graph-like way

What if I want to capitalize every instance of `Lisp`? There's a function to do that:

```
postDepth(function(x) {
  if (x === "Lisp")
    return "LISP";
  else
    return x;
}, influences);

//=> [['LISP','Smalltalk'], ['LISP','Scheme'], ...
```

So the rule is that if I want to change a node, then I do something with it and return the new value; otherwise, I just return the node. Of course, the original array is never modified:

```
influences;
//=> [['Lisp','Smalltalk'], ['Lisp','Scheme'], ...
```

What if I want to build an array of all of the languages that another language has influenced? I could perform this act as follows:

```
function influencedWithStrategy(strategy, lang, graph) {
  var results = [];

  strategy(function(x) {
    if (_.isArray(x) && _.first(x) === lang)
      results.push(second(x));

    return x;
  }, graph);
```

```
  return results;
}
```

The function `influencedWithStrategy` takes one of the depth-first searching functions and walks the graph, building an array of influenced languages along the way:

```
influencedWithStrategy(postDepth, "Lisp", influences);
//=> ["Smalltalk", "Scheme"]
```

Again, while I mutated an array to build the results, the action was confined to the internals of the `influencedWithStrategy` function localizing its effects.

Too Much Recursion!

As I mentioned in the earlier section about tail-recursion, current JavaScript engines do not optimize recursive calls, even if they technically could. Therefore, when using or writing recursive functions, you'll occasionally run into the following error:

```
evenSteven(100000);
// Too much recursion (or some variant)
```

The problem with this error (called "blowing the stack") is that the mutual-recursive nature of `evenSteven` and `oddJohn` causes each function to be called thousands of times before either one reaches zero. Because most JavaScript *implementations* have a limit on the number of recursive calls, functions like these can "blow the stack" fairly easily (Zakas 2010).

In this section, I'll talk briefly about a control structure called a *trampoline* that helps eliminate these types of errors. The basic principle is that instead of a deeply nested recursive call, a trampoline flattens out the calls. However, before getting into that, let me explore how I could manually fix the operation of `evenSteven` and `oddJohn` to not blow the stack with recursive calls. One possible way is to return a function that wraps the call to the mutually recursive function, instead of calling it directly. I can use `partial1` as follows to achieve just that:

```
function evenOline(n) {
  if (n === 0)
    return true;
  else
    return partial1(oddOline, Math.abs(n) - 1);
}

function oddOline(n) {
  if (n === 0)
    return false;
  else
    return partial1(evenOline, Math.abs(n) - 1);
}
```

As shown, instead of calling the mutually recursive function in the body of either `evenOline` and `oddOline`, a function wrapping those calls is returned instead. Calling either function with the termination case works as you'd expect:

```
evenOline(0);
//=> true

oddOline(0);
//=> false
```

Now I can manually flatten the mutual recursion via the following:

```
oddOline(3);
//=> function () { return evenOline(Math.abs(n) - 1) }

oddOline(3)();
//=> function () { return oddOline(Math.abs(n) - 1) }

oddOline(3)()();
//=> function () { return evenOline(Math.abs(n) - 1) }

oddOline(3)()()();
//=> true

oddOline(200000001)()()(); //... a bunch more ()s
//=> true
```

I suppose you could release these functions in a user-facing API, but I suspect that your clients would be less than happy to use them. Instead, you might want to supply another function, `trampoline`, that performs the flattening calls programmatically:

```
function trampoline(fun /*, args */) {
  var result = fun.apply(fun, _.rest(arguments));

  while (_.isFunction(result)) {
    result = result();
  }

  return result;
}
```

All that `trampoline` does is repeatedly call the return value of a function until it's no longer a function. You can see it in action here:

```
trampoline(oddOline, 3);
//=> true

trampoline(evenOline, 200000);
//=> true

trampoline(oddOline, 300000);
//=> false
```

```
trampoline(evenOline, 200000000);
// wait a few seconds
//=> true
```

Because of the indirectness of the call chain, the use of a trampoline adds some overhead to mutually recursive functions. However, slow is usually better than exploding. Again, you might not want to force your users to use `trampoline` just to avoid stack explosions. Instead, it can be hidden entirely with a functional facade:

```
function isEvenSafe(n) {
  if (n === 0)
    return true;
  else
    return trampoline(partial1(oddOline, Math.abs(n) - 1));
}

function isOddSafe(n) {
  if (n === 0)
    return false;
  else
    return trampoline(partial1(evenOline, Math.abs(n) - 1));
}
```

And these functions are used normally:

```
isOddSafe(2000001);
//=>true

isEvenSafe(2000001);
//=> false
```

Generators

Extrapolating from the nature of a trampoline, I'll end this section by showing a couple of examples of the infinite. Using recursion, I can demonstrate how to build and process infinite streams of "lazy" data, and likewise call mutual functions until the heat death of the sun. By lazy, I only mean that portions of a structure are not calculated until needed. By contrast, consider the use of the `cycle` function defined earlier in this chapter:

```
_.take(cycle(20, [1,2,3]), 11);
//=> [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2]
```

In this call, the array created by `cycle` is definitely not lazy, because it is fully constructed before being passed to `_.take`. Even though `_.take` only needed `11` elements from the cycled array, a full `60` elements were generated. This is quite inefficient, but alas, the default in Underscore and JavaScript itself.

However, a basic (and some would say base) way to view an array is that it consists of the first cell followed by the rest of the cells. The fact that Underscore provides a `_.first` and `_.rest` hints at this view. An infinite array can likewise be viewed as a "first" or

"head," and a "rest" or "tail." However, unlike a finite array, the tail of an infinite array may or may not yet exist. Breaking out the head and tail view into an object might help to conceptualize this view (Houser 2013):

```
{head: aValue, tail: ???}
```

The question arises: what should go into the `tail` position of the object? The simple answer, taken from what was shown in `oddOline`, is that a function that calculates the tail *is* the tail. Not only is the tail a normal function, it's a recursive function.

The head/tail object requires some maintenance, and is built using two functions: (1) a function to calculate the value at the current cell, and (2) another function to calculate the "seed" value for the next cell. In fact, the type of structure built in this way is a weak form of what is known as a generator, or a function that returns each subsequent value on demand. Keeping all of this in mind, the implementation of `generator` is as follows:[8]

```
function generator(seed, current, step) {
  return {
    head: current(seed),
    tail: function() {
      console.log("forced");
      return generator(step(seed), current, step);
    }
  };
}
```

As shown, the `current` parameter is a function used to calculate the value at the head position and `step` is used to feed a value to the recursive call. The key point about the `tail` value is that it's wrapped in a `function` and not "realized" until called. I can implement a couple of utility functions useful for navigating a generator:

```
function genHead(gen) { return gen.head }
function genTail(gen) { return gen.tail() }
```

The `genHead` and `genTail` functions do exactly what you think—they return the head and tail. However, the tail return is "forced." Allow me to create a generator before demonstrating its use:

```
var ints = generator(0, _.identity, function(n) { return n+1 });
```

Using the `generator` function, I can define the full range of integers. Now, using the accessor functions, I can start plucking away at the front:

```
genHead(ints);
//=> 0

genTail(ints);
```

8. The call to `console.log` is for demonstrative purposes only.

```
// (console) forced
//=> {head: 1, tail: function}
```

The call to `genHead` did not force the tail of `ints`, but a call to `genTail` did, as you might have expected. Executing nested calls to `genTail` will likewise force the generator to a depth equal to the number of calls:

```
genTail(genTail(ints));
// (console) forced
// (console) forced
//=> {head: 2, tail: function}
```

This is not terribly exciting, but using just these two functions I can build a more powerful accessor function like `genTake`, which builds an array out of the first n entries in the generator:

```
function genTake(n, gen) {
  var doTake = function(x, g, ret) {
    if (x === 0)
      return ret;
    else
      return partial(doTake, x-1, genTail(g), cat(ret, genHead(g)));
  };

  return trampoline(doTake, n, gen, []);
}
```

As shown, `genTake` is implemented using a trampoline, simply because it makes little sense to provide a function to traverse an infinite structure that explodes with a "Too much recursion" error for an unrelated reason. Using `genTake` is shown here:

```
genTake(10, ints);
// (console) forced x 10
//=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

genTake(100, ints);
// (console) forced x 100
//=> [0, 1, 2, 3, 4, 5, 6, ..., 98, 99]

genTake(1000, ints);
// (console) forced x 1000
//=> Array[1000]

genTake(10000, ints);
// (console) forced x 10000
// wait a second
//=> Array[10000]

genTake(100000, ints);
// (console) forced x 100000
// wait a minute
//=> Array[100000]
```

```
genTake(1000000, ints);
// (console) forced x 1000000
// wait an hour
//=> Array[1000000]
```

While not necessarily the fastest puppy in the litter, it's interesting to see how the "trampoline principle" works to define structures of infinite size, without blowing the stack, and while calculating values on demand. There is one fatal flaw with generators created with `generator`: while the tail cells are not calculated until accessed, they are calculated *every* time they are accessed:

```
genTake(10, ints);
// (console) forced x 10
//=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Knowing that I already called `genTake` to calculate the first `10` entries, it would have been nice to avoid performing the same actions again, but building a full-fledged generator is outside the scope of this book.[9]

Of course there is no free lunch, even when using trampolines. While I've managed to avoid exploding the call stack, I've just transferred the problem to the heap. Fortunately, the heap is orders of magnitude larger than the JavaScript call stack, so you're far less likely to run into a problem of memory consumption when using a trampoline.

Aside from the direct use of a trampoline, the idea of "trampolineness" is a general principle worth noting in JavaScript seen in the wild—something I'll discuss presently.

The Trampoline Principle and Callbacks

Asynchronous JavaScript APIs—like `setTimeout` and the XMLHttpRequest library (and those built on it like jQuery's `$.ajax`)—have an interesting property relevant to the discussion of recursion. You see, asynchronous libraries work off of an event loop that is non-blocking. That is, if you use an asynchronous API to schedule a function that might take a long time, then the browser or runtime will not block waiting for it to finish. Instead, each asynchronous API takes one or more "callbacks" (just functions or closures) that are invoked when the task is complete. This allows you to perform (effectively) concurrent tasks, some immediate and some long-running, without blocking the operation of your application.[10]

9. The ECMAScript.next activity is working through a design for generators in a future version of JavaScript. More information can be found at ECMA script's website.

10. There are caveats here. Of course you can still block your application any number of ways, but if used correctly, the event architecture of JavaScript will help you to avoid doing so.

An interesting feature of non-blocking APIs is that calls return immediately, before any of the callbacks are ever called. Instead, those callbacks occur in the not-too-distant future:[11]

```
setTimeout(function() { console.log("hi") }, 2000);
//=> returns some value right away
// ... about 2 seconds later
// hi
```

A truly interesting aspect of the immediate return is that JavaScript cleans up the call stack on every new tick of the event loop. Because the asynchronous callbacks are always called on a new tick of the event loop, even recursive functions operate with a clean slate! Observe the following:

```
function asyncGetAny(interval, urls, onsuccess, onfailure) {
  var n = urls.length;

  var looper = function(i) {
    setTimeout(function() {
      if (i >= n) {
        onfailure("failed");
        return;
      }

      $.get(urls[i], onsuccess)
        .always(function() { console.log("try: " + urls[i]) })
        .fail(function() {
          looper(i + 1);
        });
    }, interval);
  }

  looper(0);
  return "go";
}
```

You'll notice that when the call to jQuery's asynchronous `$.get` function fails, a recursive call to `looper` is made. This call is no different (in principle) than any other mutually recursive call, except that each invocation occurs on a different event-loop tick and starts with a clean stack. For the sake of completeness, the use of `asyncGetAny` is as follows:[12]

```
var urls = ['http://dsfgfgs.com', 'http://sghjgsj.biz', '_.html', 'foo.txt'];

asyncGetAny(2000,
```

11. Next Sunday, A.D.

12. I'm using the jQuery promise-based interface to perform the `GET` and to fluently build the `always` and `fail` handlers. Because of the nature of the concurrent execution, there is no guarantee that the console printing will occur before or after the `GET` result. I show them in order for the sake of expediency. I'll talk a little more about jQuery promises in Chapter 8.

```
  urls,
  function(data) { alert("Got some data") },
  function(data) { console.log("all failed") });
//=> "go"

// (console after 2 seconds) try: http://dsfgfgs.com
// (console after 2 seconds) try: http://sghjgsj.biz
// (console after 2 seconds) try: _.html

// an alert box pops up with 'Got some data' (on my computer)
```

There are better resources for describing asynchronous programming in JavaScript than this book, but I thought it worth mentioning the unique properties of the event loop and recursion. While tricky in practice, using the event loop for maximum benefit can make for highly efficient JavaScript applications.

Recursion Is a Low-Level Operation

This chapter has dealt extensively with recursion, creating recursive functions, and reasoning in the face of recursion. While this information is potentially useful, I should make one caveat to the entire discussion: recursion should be seen as a low-level operation and avoided if at all possible. The better path is to take a survey of the available higher-order functions and plug them together to create new functions. For example, my implementation of `influencedWithStrategy`, while clever in its way, was completely unnecessary. Instead, I should have known that functions already available could be mixed to produce the desired effect. First, I can create two auxiliary functions:

```
var groupFrom = curry2(_.groupBy)(_.first);
var groupTo   = curry2(_.groupBy)(second);
```

Because I'm using a simple nested array for my graph representation, creating new functions to operate on it is as simple as reusing existing array functions. I can explore the operation of `groupFrom` and `groupTo` here:

```
groupFrom(influences);
//=> {Lisp:[["Lisp", "Smalltalk"], ["Lisp", "Scheme"]],
//    Smalltalk:[["Smalltalk", "Self"]],
//    Scheme:[["Scheme", "JavaScript"], ["Scheme", "Lua"]],
//    Self:[["Self", "Lua"], ["Self", "JavaScript"]]}

groupTo(influences);
//=> {Smalltalk:[["Lisp", "Smalltalk"]],
//    Scheme:[["Lisp", "Scheme"]],
//    Self:[["Smalltalk", "Self"]],
//    JavaScript:[["Scheme", "JavaScript"], ["Self", "JavaScript"]],
//    Lua:[["Scheme", "Lua"], ["Self", "Lua"]]}
```

These are definitely fun functions (ha!), but they're not sufficient. Instead, a function—`influenced`—squares the circle in implementing my desired behavior:

```
function influenced(graph, node) {
  return _.map(groupFrom(graph)[node], second);
}
```

And this is, effectively, the same as my recursive `influencedWithStrategy` function:

```
influencedWithStrategy(preDepth, 'Lisp', influences);
//=> ["Smalltalk", "Scheme"]

influenced(influences, 'Lisp');
//=>["Smalltalk", "Scheme"]
```

Not only does the implementation of `influences` require far less code, but it's also conceptually simpler. I already know what `_.groupBy`, `_.first`, `second`, and `_.map` do, so to understand the implementation of `influenced` is to understand only how the data transforms from one function to the other. This is a huge advantage of functional programming—pieces fitting together like Lego blocks, data flowing and transforming along a pipeline of functions to achieve the desired final data form.

This is beautiful programming.

Summary

This chapter dealt with recursion, or functions that call themselves either directly or through other functions. Self-calling functions were shown as powerful tools used to search and manipulate nested data structures. For searching, I walked through tree-walking examples (no pun intended) using the `visit` function, which called out to depth-first searching functions.

Although the tree searching was a powerful technique, there are fundamental limitations in JavaScript that bound the number of recursive calls that can happen. However, using a technique called trampolining, I showed how you can build functions that call one another indirectly through an array of closures.

Finally, I felt the need to take a step back and make the point that recursion should be used sparingly. Very often, recursive functions are more confusing and less direct than higher-order or composed functions. The general consensus is to use function composition first and move to recursion and trampolines only if needed.

In the next chapter, I will cover a topic often at odds with functional programming—mutation, or the act of modifying variables in place—and how to limit or even outright avoid it.