

Luis Atencio

Functional Programming in

How to improve your JavaScript programs using functional techniques

SAMPLE CHAPTER

JavaScript



Functional Programming in JavaScript

by Luis Atencio

Chapter 3

Copyright 2016 Manning Publications

brief contents

PART 1	THINK FUNCTIONALLY	1
	1 ■ Becoming functional	3
	2 ■ Higher-order JavaScript	23
PART 2	GET FUNCTIONAL.....	55
	3 ■ Few data structures, many operations	57
	4 ■ Toward modular, reusable code	84
	5 ■ Design patterns against complexity	117
PART 3	ENHANCING YOUR FUNCTIONAL SKILLS.....	151
	6 ■ Bulletproofing your code	153
	7 ■ Functional optimizations	180
	8 ■ Managing asynchronous events and data	205



Few data structures, many operations

This chapter covers

- Understanding program control and flow
- Reasoning efficiently about code and data
- Unlocking the power of `map`, `reduce`, and `filter`
- Discovering the Lodash.js library and function chains
- Thinking recursively

Computational processes are abstract beings that inhabit computers.
As they evolve, processes manipulate other abstract things called data.

—Harold Abelson and Gerald Jay Sussman (*Structure and Interpretation of Computer Programs*, MIT Press, 1979)

Part 1 of this book accomplished two important goals: on one hand, those chapters got your feet wet by teaching you how to think functionally and introducing the tools you'll need to use functional programming. Second, you took a condensed tour of many core JavaScript features, particularly higher-order functions, that will be used frequently throughout this chapter and the rest of the book. Now that you know how to make functions pure, it's time to learn how to connect them.

In this chapter, I'll introduce you to a few useful and practical operations like `map`, `reduce`, and `filter` that allow you to traverse and transform data structures in a sequential manner. These operations are so important that virtually all functional programs use them in one way or another. They also facilitate removing manual loops from your code, because most loops are just specific cases handled by these functions.

You'll also learn to use a functional JavaScript library called `Lodash.js`. It lets you process and understand not only the structure of your application, but also the structure of your data. In addition, I'll discuss the important role recursion plays in functional programming and the advantages of being able to think recursively. Building on these concepts, you'll learn to write concise, extensible, and declarative programs that clearly separate control flow from the main logic of your code.

3.1 Understanding your application's control flow

The path a program takes to arrive at a solution is known as its *control flow*. An imperative program describes its flow or path in great detail by exposing all the necessary steps needed to fulfill its task. These steps usually involve lots of loops and branches, as well as variables that change with each statement. At a high level, you can depict a simple imperative program like this:

```
var loop = optC();
while(loop) {
  var condition = optA();
  if(condition) {
    optB1();
  }
  else {
    optB2();
  }
  loop = optC();
}
optD();
```

Figure 3.1 shows a simple flowchart of this program.

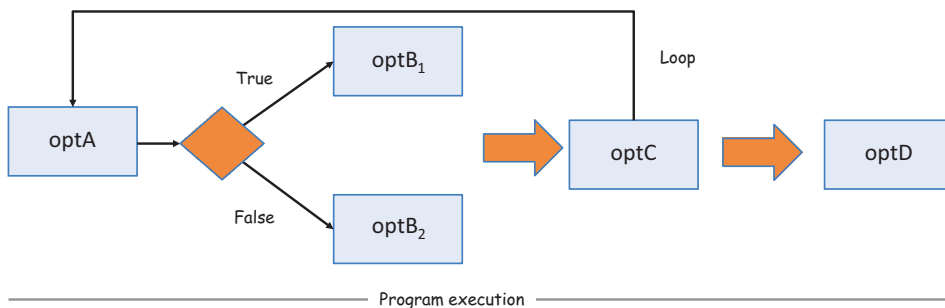


Figure 3.1 An imperative program made up of a series of operations (or statements) controlled by branches and loops

On the other hand, declarative programs, specifically functional ones, raise the level of abstraction by using a minimally structured flow made up of independent black-box operations that connect in a simple topology. These connected operations are nothing more than higher-order functions that move state from one operation to the next, as shown in figure 3.2. Working functionally with data structures such as arrays lends itself to this style of development and treats data and control flow as simple connections between high-level components.

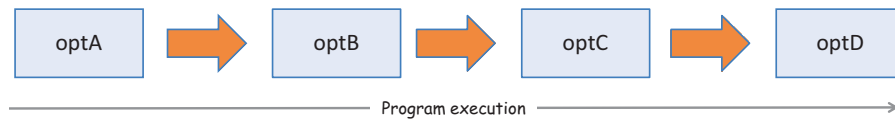


Figure 3.2 Functional control among connected black-box operations. Information flows independently from one operation to the next (the operations are individual, pure functions). Branches and iterations are effectively reduced or even eliminated in favor of high-level abstractions.

This produces code more or less like the following:

```
optA().optB().optC().optD();
```

← Connecting via dots suggests the presence of a shared object that contains these methods.

Chaining operations in this manner leads to concise, fluent, expressive programs that let you separate a program's control flow from its computational logic. Thus, you can reason about your code and your data more effectively.

3.2 Method chaining

Method chaining is an OOP pattern that allows multiple methods to be called in a single statement. When these methods all belong to the same object, method chaining is referred to as *method cascading*. Although this pattern is seen mostly in object-oriented applications, under certain conditions, such as working with immutable objects, it works just as well with functional programming. Because mutation of objects is prohibited in functional code, you may wonder how this is possible. Let's look at a string-manipulation example:

```
'Functional Programming'.substring(0, 10).toLowerCase() + ' is fun';
```

In this example, both `substring` and `toLowerCase` are string methods that operate on the owning string object (via `this`) and return new strings. The plus (+) operator is overloaded in JavaScript strings as syntactic sugar for concatenation—also producing a new string. The result of applying these transformations is a string that bears no reference to the original, which remains untouched; this is to be expected, because

strings are, by design, immutable. From an object-oriented perspective, this is taken for granted; but from the functional programming side, this is ideal—you don't require lenses to work with strings.

If you refactor the previous code into a more functional style, it looks like this:

```
concat(toLowerCase(substring('Functional Programming', 1, 10)), ' is fun');
```

This code follows the functional doctrine that all parameters should be explicitly defined in the function declaration; it has no side effects and doesn't mutate the original object. Arguably, writing this function inside out isn't as fluent as the method-chaining approach. It's also much harder to read, because you need to start peeling off the wrapped functions to understand what's truly happening.

Chaining methods belonging to a single object instance has its place in functional programming, as long as you respect the policy for change. Wouldn't it be nice to translate this pattern to work with arrays as well? The behavior we see in strings has also been extended to work with JavaScript arrays, but most people aren't familiar with it and resort to quick-and-dirty for loops.

3.3 *Function chaining*

Object-oriented programs use inheritance as the main mechanism for code reuse. Recall from the previous chapter that `Student` inherits from `Person`, and that all state and methods are inherited by the child type. You may have seen this pattern predominantly in purer object-oriented languages, especially in their data structure implementations. Java, for instance, has an explosion of concrete `List` classes for each need: `ArrayList`, `LinkedList`, `DoublyLinkedList`, `CopyOnWriteArrayList`, and others implement the basic `List` interface and derive from common parent classes, adding their own specific functionality.

Functional programming takes a different approach. Instead of creating new data structure classes to meet specific needs, it uses common ones like arrays and applies a number of coarse-grained, higher-order operations that are agnostic to the underlying representation of the data. These operations are designed to do the following:

- Accept function arguments in order to inject specialized behavior that solves your particular task
- Replace the traditional, manual looping mechanisms that contain mutations of temporary variables and side effects, thereby creating less code to maintain and fewer places where errors can occur

Let's survey these in detail. The examples in this chapter are based on a collection of `Person` objects. For brevity, I've declared only four objects, but the same concepts apply to larger collections:

```
const p1 = new Person('Haskell', 'Curry', '111-11-1111');
p1.address = new Address('US');
p1.birthYear = 1900;
```

```

const p2 = new Person('Barkley', 'Rosser', '222-22-2222');
p2.address = new Address('Greece');
p2.birthYear = 1907;

const p3 = new Person('John', 'von Neumann', '333-33-3333');
p3.address = new Address('Hungary');
p3.birthYear = 1903;

const p4 = new Person('Alonzo', 'Church', '444-44-4444');
p4.address = new Address('US');
p4.birthYear = 1903;

```

3.3.1 Understanding lambda expressions

Born from functional programming, *lambda expressions* (known as *fat-arrow functions* in the JavaScript world) encode one-line anonymous functions into a shorter syntax, compared to a traditional function declaration. You can have lambda functions with multiple lines, but one-liners are the most commonly used, as you saw in chapter 2. Whether you use lambdas or regular function syntax will depend on the readability of your code; under the hood, they're all the same. Here's a simple example of a function used to extract a person's name:

```

const name = p => p.fullname;
console.log(name(p1)); //-> 'Haskell Curry'

```

The compact notation `(p) => p.fullname` is syntactic sugar for a function that takes a parameter `p` and implicitly returns `p.fullname`. Figure 3.3 shows the structure of this new syntactic addition.

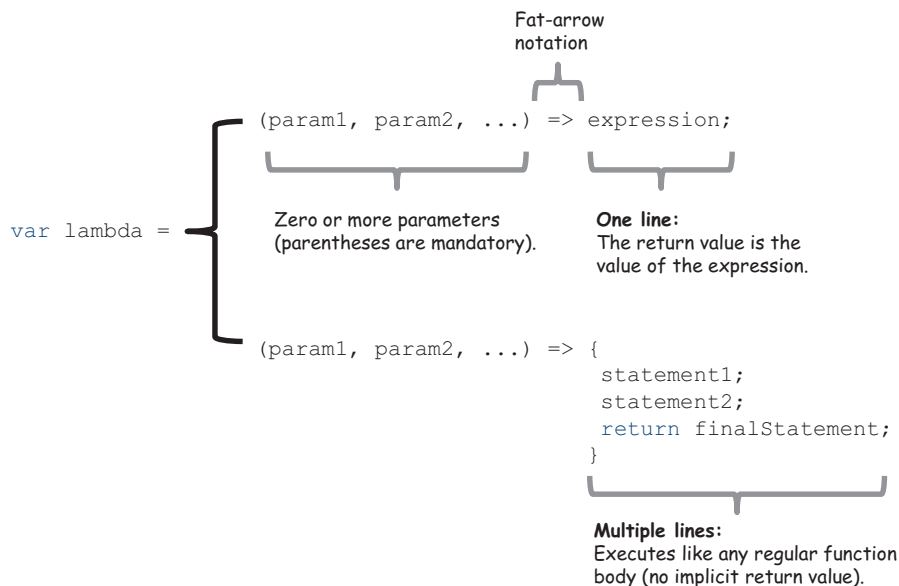


Figure 3.3 Dissecting the structure of arrow functions. The right side of a lambda function is either a single expression or an enclosed set of multiple statements.

Lambda expressions uphold the functional definition of a function because they encourage you to always return a value. In fact, for one-line expressions, the return value results from the value of the function body. Another point worth noting here is the relationship between first-class functions and lambdas. In this case, `name` points not to a concrete value, but (lazily) to a description of how to obtain it; in other words, `name` points to an arrow function that knows how to compute this data. This is why, in functional programming, you can use functions as values. I'll discuss this concept further in this chapter and lazy functions in chapter 7.

Furthermore, functional programming promotes the use of three central higher-order functions—`map`, `reduce`, and `filter`—that are designed to work well with lambda expressions. A lot of functional JavaScript code is based on processing lists of data; hence, the name of the original functional language, LISP (list processing), from which JavaScript is derived. JavaScript 5.1 provides native versions of these operations known as the functional *array extras*; but in order to create complete solutions that may involve other similar types of operations, I'll use the implementations provided in a functional library called `Lodash.js`. Its toolkit provides important artifacts that empower you to write functional programs, and it contains a rich repertoire of utility functions that handle many common programming tasks (see the appendix for details on how to install this library). Once it's installed, you can access its functionality via the global `_` (underscore or low-dash) object. Let's get started with `_.map`.

The underscore in Lodash

Lodash uses the underscore convention because it began as a fork of the famous and widely used Underscore.js project (<http://underscorejs.org/>). Lodash still tracks Underscore's API closely, to the point that it can serve as a drop-in replacement. But behind the scenes, it's a complete rewrite in favor of more elegant ways to build function chains, as well as some performance enhancements that you'll learn about in chapter 7.

3.3.2 Transforming data with `_.map`

Suppose you need to transform all the elements in a large collection of data. For instance, given a list of student objects, you want to extract each person's full name. How many times have you had to write this sequence of statements?

```
var result = [];
var persons = [p1, p2, p3, p4];
for(let i = 0; i < persons.length; i++) {
  var p = persons[i];
  if(p !== null && p !== undefined) {
    result.push(p.fullname);
  }
}
```

The imperative approach assumes fullname is a method in Student.

`map` (also known as `collect`) is a higher-order function that applies an iterator function to each element in an array, in order, and returns a new array of equal length. Here's the same program, this time using a functional style with `_.map`:

```
_.map(persons,
  s => (s !== null && s !== undefined) ? s.fullname : ''
);
```

I got rid of all var declarations using higher-order functions.

A formal definition of this operation is as follows:

```
map(f, [e0, e1, e2...]) -> [r0, r1, r2...]; where, f(dn) = rn
```

`map` is extremely useful for parsing through entire collections of elements without having to write a single loop or deal with odd scoping problems. Also, it's immutable, because the result is an entirely new array. `map` works by taking a function `f` and a collection of `n` elements as input; it returns a new array of size `n` with elements computed from applying `f` to each element in a left-to-right manner. This is depicted in figure 3.4.

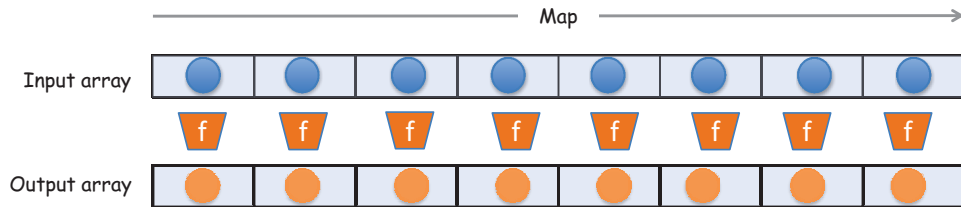


Figure 3.4 The `map` operation applies an iterator function `f` to each element in an array and returns an array of equal length.

This example of `_.map` iterates over an array of student objects and extracts their names. You use a lambda expression as the iterator function (which is common). This doesn't change the original array but, rather, returns a new one that contains the following:

```
['Haskell Curry', 'Barkley Rosser', 'John von Neumann', 'Alonzo Church']
```

Because it's always beneficial to understand one level below the abstraction layer, let's look at how `_.map` could be implemented.

Listing 3.1 Map implementation

```

function map(arr, fn) {
  let idx    = 0,
      len    = arr.length,
      result = new Array(len);

  while (++idx < len) {
    result[idx] = fn(arr[idx], idx, arr);
  }
  return result;
}

```

Takes a function and an array, applies the function to each element, and returns a new array of the same size as the original

Result: An array of the same length as the input

Applies the function fn to each element in the array and puts the result back into an array

As you can see, internally, `_.map` is based on standard loops. This function handles iteration on your behalf, so you're only responsible for administering the proper functionality in the iterator function instead of having to worry about mundane concerns like incrementing loop variables and bounds checks. This is an example of how functional libraries bring your code to the same level as purer functional languages.

`map` is exclusively a left-to-right operation; for a right-to-left sweep, you must first reverse the array. JavaScript's `Array.reverse()` operation won't work, because it mutates the original array in place, but a functional equivalent of `reverse` can be connected with `map` in a single statement:

```

_(persons).reverse().map(
  p => (p !== null && p !== undefined) ? p.fullname : ''
);

```

Notice the use of a slightly different syntax in this example. `Lodash` provides a nice, noninvasive way to integrate your code with it. All that's needed for it to be able to manage your objects is for you to wrap them in the notation `_(...)`. Afterward, you have complete control of its powerful functional arsenal to apply any transformations you need.

Mapping over containers

The concept of mapping over data structures (in this case, an array) to transform the constituent values has far-reaching implications. Just as you can map any function over an array, in chapter 5 you'll learn that you can also map a function over any object.

Now that you can apply a transformation function over your data, it's useful to be able to make conclusions or extract certain results based on the new structure. This is the work of the `reduce` function.

3.3.3 Gathering results with `_.reduce`

You know how to transform your data, but how do you gather meaningful results from it? Suppose you want to compute the country with the largest count from a collection of Person objects. You can use the `reduce` function to accomplish this.

`reduce` is a higher-order function that compresses an array of elements down to a single value. This value is computed from the accumulated result of invoking a function with an accumulator value against each element. This is easier to visualize by looking at the diagram in figure 3.5.

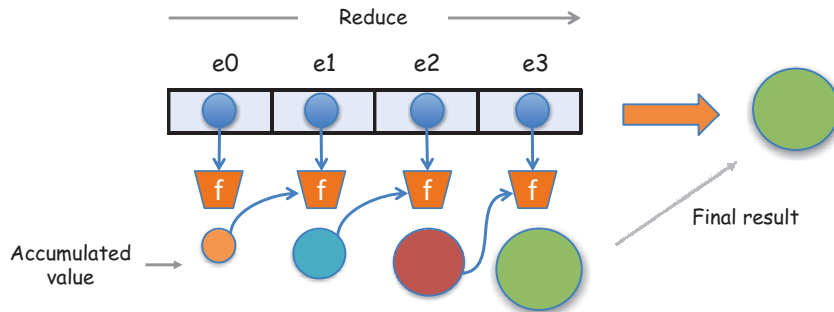


Figure 3.5 Reducing an array into a single value. Each iteration returns an accumulated value based on the previous result; this accumulated value is kept until you reach the end of the array. The final outcome of `reduce` is always a single value.

This diagram can be expressed more formally with the following notation:

`reduce(f, [e0, e1, e2, e3], accum) -> f(f(f(f(accum, e0), e1), e2), e3)) -> R`

Now, let's look at a simplified implementation of the internals of `reduce`.

Listing 3.2 Implementing `reduce`

```
function reduce(arr, fn, [accumulator]) {
  let idx = -1,
      len = arr.length;

  if (!accumulator && len > 0) {
    accumulator = arr[++idx];
  }

  while (++idx < len) {
    accumulator = fn(accumulator,
                     arr[idx], idx, arr);
  }
  return accumulator;
}
```

← If no accumulator value is provided, the first element of the array is used to initialize it.

← Invokes `fn` on each element, passing the accumulated value and the current element value

← Returns the single accumulated value

`reduce` accepts the following parameters:

- `fn`—The iterator function is executed on every value in the array and contains as parameters the accumulated value, the current value, the index, and the array.
- `accumulator`—The initial value, which is then used to store the accumulated result that's passed in to every subsequent function call.

Let's write a simple program that gathers some statistics about a set of `Person` objects. Suppose you want find the number of people who live in a particular country; see the following listing.

Listing 3.3 Computing country counts

```
_(persons).reduce(function (stat, person) {
  const country = person.address.country;
  stat[country] = _.isUndefined(stat[country]) ? 1 :
    stat[country] + 1;
  return stat;
}, {});
```

Starts the reduce process with an empty object (initializes the accumulator) (points to `{}`)

Returns the accumulated object (points to `return stat;`)

Extracts a person's country (points to `const country = person.address.country;`)

Creates a country entry that's initialized to 1 and incremented with every person living in that country (points to `stat[country] = ...`)

Running this code converts the input array into a single object containing a representation of the population by country:

```
{
  'US'      : 2,
  'Greece'  : 1,
  'Hungary' : 1
}
```

To simplify this task further, you can implement the ubiquitous `map-reduce` combination. Linking these functions, you can enhance the behavior of `map` and `reduce` by providing specialized behaviors as parameters. At a high level, this program's flow has the structure

```
_(persons).map(func1).reduce(func2);
```

where `func1` and `func2` implement the particular behavior you want. Separating the functions from the main flow, you get the code in the next listing.

Listing 3.4 Combining `map` and `reduce` to compute statistics

```
const getCountry = person => person.address.country;
const gatherStats = function (stat, criteria) {
  stat[criteria] = _.isUndefined(stat[criteria]) ? 1 :
```

```

        stat[criteria] + 1;
    return stat;
};

_(persons).map(getCountry).reduce(gatherStats, {});

```

Listing 3.4 uses `map` to preprocess the array of objects and extract all countries; then it uses `reduce` to collect the final result. This produces the same output as listing 3.3, but in a much cleaner and extensible way. Instead of direct property access, consider providing a lens (using Ramda) that focuses on the person's `address.city` property:

```

const cityPath = ['address', 'city'];
const cityLens = R.lens(R.path(cityPath), R.assocPath(cityPath));

```

And just as easily, you can compute counts based on the cities people reside in:

```

_(persons).map(R.view(cityLens)).reduce(gatherStats, {});

```

Alternatively, you can use `_.groupBy` to accomplish a similar outcome in an even more succinct way:

```

_.groupBy(persons, R.view(cityLens));

```

Unlike `map`, because `reduce` relies on an accumulated result, it can behave differently when applied left-to-right or right-to-left if not provided with a commutative operation. To illustrate this, consider a simple program that sums up the numbers in an array:

```

_([0,1,3,4,5]).reduce(_.add); // -> 13

```

The same result can be obtained by reducing in reverse with `_.reduceRight`. This works as expected because addition is a commutative operation, but it can produce significantly different results for operations that aren't, like division. Using the same notation as before, `_.reduceRight` can be viewed as follows:

```

reduceRight(f, [e0, e1, e2], accum) -> f(e0, f(e1, f(e2, f(e3, accum)))) -> R

```

For instance, these two programs using `_.divide` will compute completely different values:

```

([1,3,4,5]).reduce(_.divide) !== ([1,3,4,5]).reduceRight(_.divide);

```

Furthermore, `reduce` is an apply-to-all operation, which means there's no way for it to be short-circuited so it doesn't run through the entire array. Suppose you need to validate a list of input values. You could think of validating an array of parameters as reducing it to a single Boolean value, indicating whether all parameters are valid.

Using `reduce`, however, would be a bit inefficient because you'd have to visit all values in the list. Once you've found an invalid input, there's no point continuing to check all of them. Let's look at a more efficient validation function that uses `_.some` and other functions you'll come to know and love: `_.isUndefined` and `_.isNull`. When applied against each element in the list, `_.some` returns as soon as it finds a passing (true) value:

```
const isValid = val => !_.isUndefined(val) && !_.isNull(val);
const notAllValid = args => !_.every(args, isValid);

validate(['string', 0, null, undefined]) //-> false
validate(['string', 0, {}])               //-> true
```

Value isn't valid when undefined or null

Function `some` returns as soon as it yields true. This is useful when checking that there's at least one valid value

You can also obtain the logical inverse of `notAllValid` (called `allValid`) using `_.every`, which checks whether the given predicate returns true for all elements:

```
const isValid = val => !_.isUndefined(val) && !_.isNull(val);
const allValid = args => _.every(args, isValid);

allValid(['string', 0, null]); //-> false
allValid(['string', 0, {}]);   //-> true
```

As you saw earlier, both `map` and `reduce` attempt to traverse the entire array. Often, you aren't interested in processing all elements in your data structure and would like to skip any null or undefined objects. It would be nice if you had a mechanism to remove or filter out certain elements from the list before the computation takes place. Let's visit `_.filter` next.

3.3.4 Removing unwanted elements with `_.filter`

When processing large collections of data, it's often necessary to remove elements that don't form part of your computations. For instance, say you want to count only people living in European countries, or people born in a certain year. Instead of cluttering your code with if-else statements, you can use `_.filter`.

`filter` (also known as `select`) is a higher-order function that iterates through an array of elements and returns a new array that's a subset of the original with values for which a predicate function `p` returns a result of true. In formal notation, this looks like the following (also see figure 3.6):

```
filter(p, [d0, d1, d2, d3...dn]) -> [d0,d1,...dn] (subset of original input)
```

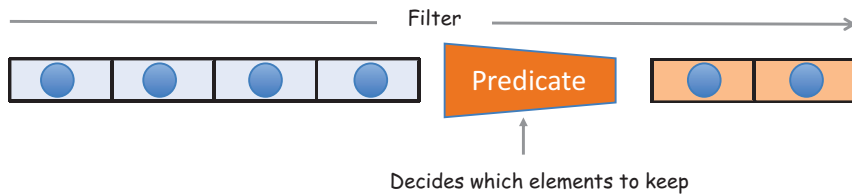


Figure 3.6 The filter operation takes an array as input and applies a selection criteria *p* that potentially yields a much smaller subset of the original array. The criteria *p* is also known as a *function predicate*.

A possible implementation of filter is shown in the following listing.

Listing 3.5 filter implementation

```
function filter(arr, predicate) {
  let idx = -1,
      len = arr.length,
      result = [];

  while (++idx < len) {
    let value = arr[idx];
    if (predicate(value, idx, this)) {
      result.push(value);
    }
  }
  return result;
}
```

Resulting array contains a subset of array values

Calls the predicate function. If the result is true, the value is kept; otherwise it's skipped.

In addition to the array, `filter` accepts a predicate function used to test each member of the array for inclusiveness. If the predicate yields `true`, the element is kept in the result; otherwise, the element is skipped. This is why `filter` is commonly used to remove invalid data from an array:

```
_(persons).filter(isValid).map(fullname);
```

But it can do much more than that. Suppose you need to extract only people born in 1903 from a collection of `Person` objects. Applying `_.filter` is much easier and cleaner than using conditional statements:

```
const bornIn1903 = person => person.birthYear === 1903;

_(persons).filter(bornIn1903).map(fullname).join(' and ');

//-> 'Alonzo Church and Haskell Curry'
```


Array comprehension

`map` and `filter` are higher-order functions that return new arrays from existing ones. They exist in many functional programming languages like Haskell, Clojure, and others. An alternative to combining `map` and `filter` is to use a concept called *array comprehension*, also known as *list comprehension*. It's a functional feature that encapsulates the functionality of `map` and `filter` into a concise syntax using the `for...of` and `if` keywords, respectively:

```
[for (x of iterable) if (condition) x]
```

At the time of this writing, there's a proposal in ECMAScript 7 to include array comprehensions. They'll let you create concise expressions to assemble new arrays (which is why the entire expression is wrapped in `[]`). For example, you can refactor the previous code in the following manner:

```
[for (p of people) if (p.birthYear === 1903) p.fullname]
  .join(' and ');
```

Applying all of these techniques based on these extensible and powerful functions allows you not only to write cleaner code but also to improve your understanding of the data. Using the declarative style, you can focus on what the output of the application will be instead of how to get there, facilitating deeper reasoning in your application.

3.4 Reasoning about your code

Recall that, in JavaScript, thousands of lines of code that share a global namespace can be loaded into a single page at once. Lately there's lots of interest in creating modules to compartmentalize business logic, but thousands of projects in production still don't do this.

What does it mean to “reason about your code”? I've used this term loosely in previous chapters to refer to the ability to look into any part of a program and easily build a mental model of what's happening. This model includes dynamic parts like the state of all variables and the outcomes of functions, as well as static parts such as the level of readability and expressiveness of your design. Both are important. You'll learn in this book that immutability and pure functions make building this model much easier.

Earlier, I highlighted the value of being able to link high-level operations together to build programs. An imperative program flow is radically different from a functional program flow. A functional flow gives you a clear picture as to the purpose of the program without revealing any of its internal details, so that you can reason more deeply about the code as well as how data flows into and out of the different stages to produce results.

3.4.1 Declarative and lazy function chains

Recall from chapter 1 that functional programs are made up of simple functions that in themselves don't accomplish much but, when put together, can solve complex tasks. In this section, you'll learn a way to build entire an program by linking a set of functions.

Functional programming's declarative model treats programs as the evaluation of independent, pure functions, which support you in building the necessary abstractions to gain fluency and expressiveness in your code. Doing so, you can form an ontology or vocabulary that clearly expresses the intent of your application. Building pure functions on top of the building blocks of `map`, `reduce`, and `filter` leads to writing in a style that makes code easy to reason about and understand at a glance.

The powerful effect of raising this level of abstraction is that you begin to think of operations as agnostic to the underlying data structures used. Theoretically speaking, whether you're working with arrays, linked lists, binary trees, or otherwise, it shouldn't change the semantic meaning of your program. For this reason, functional programming focuses on operations more than on the structure of the data.

For example, suppose you're tasked to read a list of names, normalize them, remove any duplicates, and sort the final result. First, let's write an imperative version of this program; later you'll refactor it into a functional style.

You can express the list of names as an array with unevenly formatted input strings:

```
var names = ['alonzo church', 'Haskell curry', 'stephen_kleene',
            'John Von Neumann', 'stephen_kleene'];
```

The imperative program is shown next.

Listing 3.6 Performing sequential operations on arrays (imperative approach)

```
var result = [];
for (let i = 0; i < names.length; i++) {
  var n = names[i];
  if (n !== undefined && n !== null) {
    var ns = n.replace(/_/g, ' ').split(' ');
    for(let j = 0; j < ns.length; j++) {
      var p = ns[j];
      p = p.charAt(0).toUpperCase() + p.slice(1);
      ns[j] = p;
    }
    if (result.indexOf(ns.join(' ')) < 0) {
      result.push(ns.join(' '));
    }
  }
}
result.sort();
```

Loops through all names in the array

Checks to make sure all words are valid

The array contains inconsistently formatted data; this step normalizes (fixes) each element.

Eliminates duplicates by checking whether the name exists in the result

Sorts the array

This code produces the desired output:

```
['Alonzo Church', 'Haskell Curry', 'Jon Von Neumann', 'Stephen Kleene']
```

The downside of imperative code is that it's targeted at solving a particular problem efficiently. The code in listing 3.6 can only be used to perform this particular task. Therefore, it runs at a far lower level of abstraction than functional code. The lower the level of abstraction, the lower the probability of reuse, and the greater the complexity and likelihood of errors.

On the other hand, the functional implementation merely connects black-box components together and cedes the responsibility to these well-established and tested APIs, as shown in the following listing. Notice how the cascade arrangement of function calls makes this code easier to read.

Listing 3.7 Performing sequential operations on arrays (functional approach)

```
_.chain(names)
  .filter(isValid)
  .map(s => s.replace(/_/g, ' '))
  .uniq()
  .map(_.startCase)
  .sort()
  .value();
```

```
//-> ['Alonzo Church', 'Haskell Curry', 'Jon Von Neumann', 'Stephen Kleene']
```

The `_.filter` and `_.map` functions take care of all the heavy lifting of iterating through valid indexes in the `names` array. Your only job is to supply the specialized behavior in the remaining steps. You use the `_.uniq` function to throw away duplicate entries and `_.startCase` to capitalize each word; finally, you sort all the results.

I'd much rather write and read programs that look like listing 3.7, wouldn't you? Not just because of the sheer reduction in the amount of code, but also due to its simple and clear structure.

Let's continue exploring *Lodash*. This example revisits listing 3.4, which computes counts of all countries from an array of *Person* objects. For the purpose of this example, augment the `gatherStats` function slightly:

```
const gatherStats = function (stat, country) {
  if (!isValid(stat[country])) {
    stat[country] = { 'name': country, 'count': 0 };
  }
  stat[country].count++;
  return stat;
};
```

It now returns an object with the following structure:

```
{
  'US'      : {'name': 'US', count: 2},
  'Greece'  : {'name': 'Greece', count: 1},
  'Hungary' : {'name': 'Hungary', count: 1}
}
```

Using this structure guarantees unique entries for each country. Just for fun, let's inject a few more data points into the `Person` array you began the chapter with:

```
const p5 = new Person('David', 'Hilbert', '555-55-5555');
p5.address = new Address('Germany');
p5.birthYear = 1903;

const p6 = new Person('Alan', 'Turing', '666-66-6666');
p6.address = new Address('England');
p6.birthYear = 1912;

const p7 = new Person('Stephen', 'Kleene', '777-77-7777');
p7.address = new Address('US');
p7.birthYear = 1909;
```

The next task is to build a program that returns the country with the largest number of people in this dataset. Let's do this again by linking a function with the help of `_.chain()` and a few other artifacts.

Listing 3.8 Demonstrating lazy function chains with Lodash

```
_.chain(persons)
  .filter(isValid)
  .map(_._property('address.country'))
  .reduce(gatherStats, {})
  .values()
  .sortBy('count')
  .reverse()
  .first()
  .value()
  .name;  //-> 'US'
```

Creates a lazy function chain to process the provided array

Uses `_.property` to extract the person object's `address.country` property. This is Lodash's equivalent but less feature-rich version of Ramda's `R.view()`.

Executes all functions in the chain

The `_.chain` function can be used to augment the state of an input object by connecting operations that transform the input into the desired output. It's powerful because, unlike wrapping arrays with the shorthand `_(...)` object, it explicitly makes any function in the sequence chainable. Despite this being a complex program, you can avoid creating any variables, and all looping is effectively eliminated.

Another benefit of using `_.chain` is that you can create complex programs that behave lazily, so nothing executes until that last `value()` function is called. This can have a tremendous impact in your application because you can potentially skip running

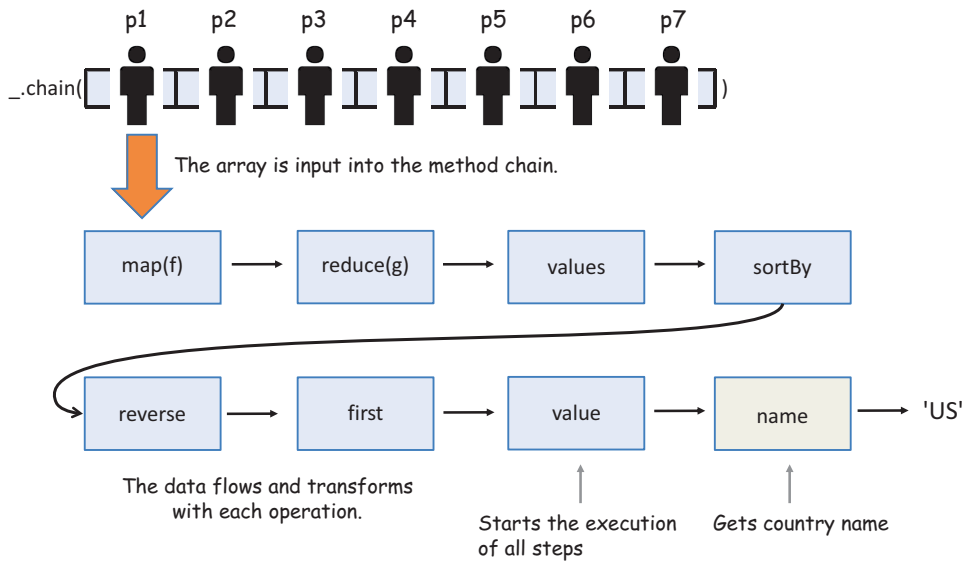


Figure 3.7 Control of a program built using Lodash function chains. The array of person objects is processed through a series of operations. Along the way, the data flows and is finally transformed into a single value.

entire functions when their results aren't needed (lazy evaluation will be discussed in chapter 7). This program's control flow is depicted in figure 3.7.

You're beginning to see why functional programs are superior. The imperative version of this task, I'll leave to your imagination. The reason listing 3.8 works so smoothly relates to the fundamental principles underlying FP—pure and side effect-free functions. Each function in the chain immutably operates on new arrays built from functions that precede it. By starting with a call to `_.chain()`, Lodash capitalizes on this pattern to provide a Swiss Army knife of utilities to satisfy most needs. This helps you transition into a style of programming called *point-free*; it's unique to functional programming, and I'll introduce it in the next chapter.

Being able to lazily define program pipelines has many more benefits than just readability. Because lazy programs are defined before they're evaluated, they can be optimized using techniques such as data-structure reuse and method fusion. These optimizations don't reduce the time it takes to execute functions per se; rather, they help to eliminate unnecessary invocations. I'll discuss this in more detail in chapter 7 when we study the performance of functional programs.

In listing 3.8, data is flowing from one node to the next in the network. Using higher-order functions declaratively makes it obvious how data transforms in each node, revealing more insights about your data.

3.4.2 SQL-like data: functions as data

Throughout this chapter, you’ve been exposed to an assortment of functions such as `map`, `reduce`, `filter`, `groupBy`, `sortBy`, `uniq`, and so on. The vocabulary formed around these functions can be used to clearly extrapolate information pertaining to your data. If you think outside the box for a second, you’ll notice that these functions resemble SQL, which isn’t accidental.

Developers are accustomed to using SQL and its features to understand and extrapolate meaning from data. For example, you can represent the collection’s person objects as shown in table 3.1.

Table 3.1 Representing the data in the person list as a table

id	firstname	lastname	country	birthYear
0	Haskell	Curry	US	1900
1	Barkley	Rosser	Greece	1907
2	John	Von Neumann	Hungary	1903
3	Alonzo	Church	US	1903
4	David	Hilbert	Germany	1862
5	Alan	Turing	England	1912
6	Stephen	Kleene	US	1909

As it turns out, thinking in terms of a query language when building programs is similar to the operations applied to arrays in functional programming, which make use of a common vocabulary or algebra, if you will, to encourage a deeper reasoning about the nature of your data and how it’s structured. The following SQL query

```
SELECT p.firstname, p.birthYear FROM Person p
WHERE p.birthYear > 1903 and p.country IS NOT 'US'
GROUP BY p.firstname, p.birthYear
```

makes it crystal clear what you should expect your data to look like after running this code. Before you implement the JavaScript version of this program, let’s implement a few function aliases to help me make this point. *Lodash* supports a feature called *mix-ins* that can be used to extend the core library with additional functions and have them chained the same way:

```
_.mixin({ 'select':  _.pluck,
         'from':    _.chain,
         'where':   _.filter,
         'groupBy':  _.sortByOrder});
```

After applying this mixin object, you can write the following program.

Listing 3.9 Writing SQL-like JavaScript

```
_.from(persons)
  .where(p => p.birthYear > 1900 && p.address.country !== 'US')
  .groupBy(['firstname', 'birthYear'])
  .select('firstname', 'birthYear')
  .value();

//-> ['Alan', 'Barkley', 'John']
```

Listing 3.9 creates aliases that map native SQL keywords to corresponding functions, so you may experience a closer realization of functional code to a query language.

JavaScript mixins

A mixin is an object that defines an abstract subset of functions relating to a particular type (in this case, a SQL command). This object isn't concretely used in code, other than to extend the behavior of another object (it's somewhat similar to a *trait* in other programming languages). The target object borrows all of functionality from the mixin.

In the object-oriented world, it's also another way to reuse code without having to use inheritance or to simulate multiple inheritance in languages that don't support it (JavaScript being one of them). I won't cover mixins in this book, but they can be powerful when used correctly. If you want to learn more about mixins, I suggest reading <https://javascriptweblog.wordpress.com/2011/05/31/a-fresh-look-at-javascript-mixins/>.

You should be convinced by now that functional programming can behave as a powerful abstraction over imperative code. What better way of processing and parsing your data than to use query language semantics? Like SQL, this JavaScript code models the data in the form of functions, also known as *functions as data*. Because it's declarative, it describes *what* the data output is and not *how* it came to be. So far, I haven't needed any conventional looping statements in this chapter—and I don't intend to use them for the rest of the book. Instead, high-level abstractions replace looping.

Another technique commonly used to replace loops is recursion, which you can use to abstract iteration when tackling problems that are “self-similar” in nature. For these types of problems, sequential function chains are inefficient and inadequate. Recursion, on the other hand, implements its own ways of processing data by yielding the heavy lifting of standard looping to the language runtime.

3.5 Learning to think recursively

Sometimes a problem is difficult and complex to tackle head on. When this occurs, you should immediately look for ways to decompose it. If the problem can be broken down into smaller versions of itself, you may be able to solve the smaller version and build it up to solve the entire problem. Recursion is essential for array traversal in pure functional programming languages like Haskell, Scheme, and Erlang because they don't have looping constructs.

In JavaScript, recursion has many applications, such as parsing XML or HTML documents, graphs, and so on. In this section, I'll explain what recursion is and then work through an exercise that will teach you how think recursively. Then we'll take a quick look at a few data structures you can parse through using recursion.

3.5.1 What is recursion?

Recursion is a technique designed to solve problems by decomposing them into smaller, self-similar problems that, when combined, arrive at the original solution. A recursive function has two main parts:

- Base cases (also known as the *terminating condition*)
- Recursive cases

The base cases are a set of inputs for which a recursive function computes a concrete result, without having to recur. The recursive case deals with a set of inputs (necessarily smaller than the original) for which the function calls itself. If the input isn't smaller, the recursion runs indefinitely until the program crashes. As the function recurs, the nature of the inputs unconditionally become smaller, finally reaching the instance for which the base case is triggered and the process terminates with a value.

Recall from chapter 2 that we used recursion to deep-freeze an entire nested object structure. The base case triggered when the object encountered was a primitive or had already been frozen; otherwise, the recursive step continued traversing the object structure as it found more unfrozen objects. Recursion was appropriate for this because at each level, the task to solve was exactly the same. Thinking recursively, though, can be a challenge, so let's begin there.

3.5.2 Learning to think recursively

Recursion isn't a simple concept to grasp. As with functional programming, the hardest part is unlearning conventional ways. The focus of this book is not to make you a master of recursion, and it's not a technique you'll use often; but it's important, and I'd like to exercise your brain and help you learn to analyze recursive problems better.

Recursive thinking takes itself or a modified version of itself into consideration. A recursive object is self-defining; for instance, think of the composition of branches in a tree. A branch has leaves as well as other branches, which in turn have more leaves and more branches. This process continues indefinitely and is halted only by a limiting external factor—the size of the tree, in this case.

With that in mind, let's do a warm-up exercise by tackling a simple problem: adding all the numbers in an array. We'll go from an imperative implementation to the most functional. The imperative side of your brain naturally visualizes a solution involving iterating through the array and keeping an accumulated value:

```
var acc = 0;
for(let i = 0; i < nums.length; i++) {
  acc += nums[i];
}
```

Your brain pushes you to consider the need for an accumulator, which is absolutely necessary when you're keeping a running total. But do you need to use a manual loop? At this point you're well aware that you have more weapons at your disposal in your functional arsenal (`_.reduce`):

```
_(nums).reduce((acc, current) => acc + current, 0);
```

Pushing manual iteration into the framework abstracts your application code from it. But you can do even better by ceding iteration entirely to the platform. The function `_.reduce` shows that you don't have to be concerned about looping or even the size of the list. You can compute the result by subsequently adding the first element to the rest and, thus, achieve recursive thinking. This thought process can be extended to picture summation as performing a sequence of operations in the following manner, which is known as *lateral thinking*:

```
sum[1,2,3,4,5,6,7,8,9] = 1 + sum[2,3,4,5,6,7,8,9]
                      = 1 + 2 + sum[3,4,5,6,7,8,9]
                      = 1 + 2 + 3 + sum[4,5,6,7,8,9]
```

Recursion and iteration are two sides of the same coin. In the absence of mutation, recursion offers a more expressive, powerful, and excellent alternative to iteration. In fact, pure functional languages don't even have standard looping constructs like `do`, `for`, and `while`, since all looping is done recursively. Recursion also leads to code that's easier to understand because it's premised on repeating the same actions multiple times on smaller input. The recursive solution in the following listing uses the `Lodash` `_.first` and `_.rest` functions to access the first element of the array or all but the first, respectively.

Listing 3.10 Performing recursive addition

```
function sum(arr) {
  if(_.isEmpty(arr)) {
    return 0;
  }
  return _.first(arr) + sum(_.rest(arr));
}
sum([]); //-> 0
sum([1,2,3,4,5,6,7,8,9]); //->45
```

← Base case (terminating condition)

← Iterative case: calls itself on a smaller input using `_.first` and `_.rest`

Adding the empty array triggers the base case, naturally returning zero. Otherwise, for non-empty arrays, you proceed to recursively extract and sum the first elements together with the rest of the array. Behind the scenes, recursive calls are stacked on top of each other. As soon as the algorithm reaches the terminating condition, all the return statements are executed as the runtime unwinds the stack to let the addition take place. This is the mechanism by which recursion cedes looping to the language runtime. Here's a step-by-step view of the sum algorithm you just implemented:

```

1 + sum[2,3,4,5,6,7,8,9]
1 + 2 + sum[3,4,5,6,7,8,9]
1 + 2 + 3 + sum[4,5,6,7,8,9]
1 + 2 + 3 + 4 + sum[5,6,7,8,9]
1 + 2 + 3 + 4 + 5 + sum[6,7,8,9]
1 + 2 + 3 + 4 + 5 + 6 + sum[7,8,9]
1 + 2 + 3 + 4 + 5 + 6 + 7 + sum[8,9]
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + sum[9]
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + sum[]
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 0      -> halts, stack unwinds
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
1 + 2 + 3 + 4 + 5 + 6 + 7 + 17
1 + 2 + 3 + 4 + 5 + 6 + 24
1 + 2 + 3 + 4 + 5 + 30
1 + 2 + 3 + 4 + 35
1 + 2 + 3 + 39
1 + 2 + 42
1 + 44
45


```

At this point, it's natural to think about the performance of recursion versus manual iteration. After all, compilers have become extremely smart at optimizing loops. ES6 JavaScript brings an optimization feature called *tail-call optimization* that can bring the performance of these two features closer together. Consider this slightly different implementation of sum:

```

function sum(arr, acc = 0) {
  if(_.isEmpty(arr)) {
    return 0;
  }
  return sum(_.rest(arr), acc + _.first(arr));
}

```


Recursive call in tail position

This version places the recursive call as the last step in the function body, or in *tail position*. We'll explore the benefits of doing this further in chapter 7 when we look at functional optimizations.

3.5.3 Recursively defined data structures

You're probably wondering about the names passed in to the person objects we've been using as sample data. Back in the 1900s, the mathematics community behind functional programming (lambda calculus, category theory, and so on) was vibrant.

Much of the work published was based on joint ideas and theorems by leading universities under the tutelage of professors like Alonzo Church. In fact, many mathematicians like Barkley Rosser, Alan Turing, and Stephen Kleene, among others, were doctoral students of Church's. They went on to have doctoral students of their own. Figure 3.8 graphs this apprenticeship relationship (or a sliver of it).

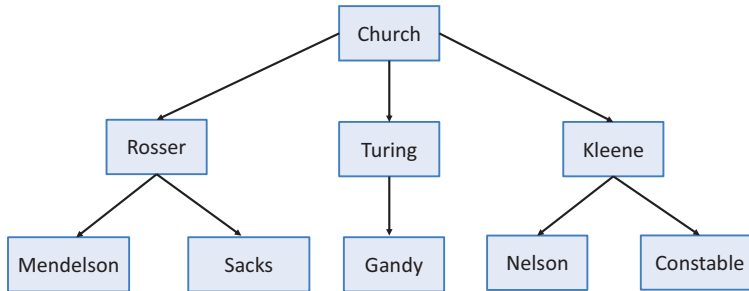


Figure 3.8 Influential mathematicians who contributed to the development of functional programming. The connected lines from parent to child nodes in the tree structure represent a “student of” relationship.

Structures like this are common in software because they can be used to model XML documents, file systems, taxonomies, categories, menu widgets, faceted navigation, social graphs, and more. So learning how to process them is vital. Figure 3.8 shows a set of nodes with connections that denote advisor-student affiliations. Up to now, you’ve used functional techniques to parse flat data structures, like arrays. But these operations won’t work on tree-like data. Because JavaScript doesn’t have a built-in tree object, you create a simple data structure based on nodes. A *node* is an object that contains a value, a reference to its parent, and an array of children. In figure 3.8, Rosser has Church as its parent node and Mendelson and Sacks as children. If a node has no parent, as is the case with Church, it’s considered the root. Here’s the definition of the Node type.

Listing 3.11 Node object

```

class Node {
  constructor(val) {
    this._val = val;
    this._parent = null;
    this._children = [];
  }

  isRoot() {
    return isValid(this._parent);
  }

  get children() {
    return this._children;
  }
}

```

← This function was created before.

```

hasChildren() {
  return this._children.length > 0;
}

get value() {
  return this._val;
}

set value(val) {
  this._val = val;
}

append(child) {
  child._parent = this;
  this._children.push(child);
  return this;
}

toString() {
  return `Node (val: ${this._val}, children:
    ${this._children.length})`;
}
}

```

Sets this node's parent

Adds this child node to the list of children

Returns this same node (convenient for method cascading)

You can create new nodes like this:

```

const church = new Node(new Person('Alonzo', 'Church', '111-11-1111'));//

```

Repeat this for every node in the tree.

Trees are recursively defined data structures that contain a root node:

```

class Tree {
  constructor(root) {
    this._root = root;
  }

  static map(node, fn, tree = null) {
    node.value = fn(node.value);
    if (tree === null) {
      tree = new Tree(node);
    }

    if (node.hasChildren()) {
      _map(node.children, function (child) {
        Tree.map(child, fn, tree);
      });
    }
    return tree;
  }
}

```

Uses a static method to avoid confusion with the more popular `Array.prototype.map`. A static method can also be used effectively as a standalone function.

Invokes the iterator function and updates the value of the node element in the tree

Similar to `Array.prototype.map`; the result is a new structure.

If the node has no children, no need to continue (base case).

Recursive call on each child node

Invokes the provided function against each child node

```

    get root() {
      return this._root;
    }
  }
}

```

The node's main logic lies in the `append` method. Appending a child to a node sets the child's parent reference to it and adds the input node to the list of children. You populate the tree by linking nodes to other child nodes in the following manner, starting with the root, church:

```

church.append(rosser).append(turing).append(kleene);
kleene.append(nelson).append(constable);
rosser.append(mendelson).append(sacks);
turing.append(gandy);

```

Each node is in charge of wrapping a person object. The recursive algorithm performs a preorder traversal of the entire tree, beginning at the root and descending to all of its children. Due to its self-similar nature, traversing the tree from the root node is exactly like traversing it from any node: a recursive definition. For this, you use `Tree.map`, a higher-order function with semantics similar to `Array.prototype.map`, which accepts a function that's evaluated against each node value. As you can see, regardless of the data structure used to model this data (a tree, in this case), the semantics of this function should remain the same. Essentially, any data type can be mapped over by preserving its structure. I'll consider this notion of mapping structure preserving functions to types more formally in chapter 5.

A preorder traversal of this tree has the following steps, starting with root:

- 1 Display the data part of the root element
- 2 Traverse the left subtree by recursively calling the preorder function
- 3 Traverse the right subtree the same way

Figure 3.9 illustrates the path the algorithm takes.

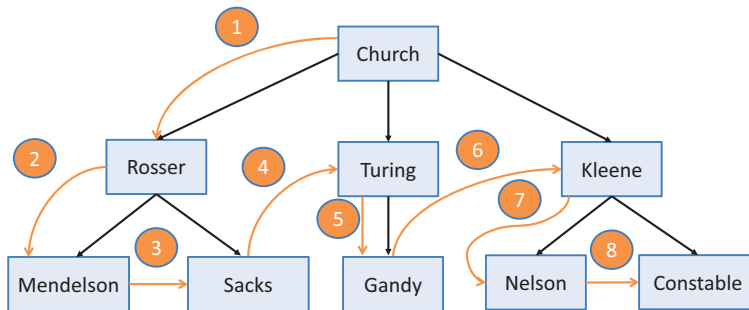


Figure 3.9 Recursive preorder traversal, starting with the root and descending all the way to the left before going to the right

The function `Tree.map` has two required inputs: the root node (which is basically the start of the tree) and the iterator function that transforms each node's value:

```
Tree.map(church, p => p.fullname);
```

This traverses the tree in preorder and applies the given function to each node, producing the following:

```
'Alonzo Church', 'Barkley Rosser', 'Elliot Mendelson', 'Gerald Sacks', 'Alan  
Turing', 'Robin Gandy', 'Stephen Kleene', 'Nels Nelson', 'Robert Constable'
```

This idea of encapsulating data to control how it's accessed is key to functional programming when working with immutability and side effect-free data types. I'll expand on this idea further in chapter 5. Parsing data structures is one of the most fundamental aspects of software and the bread and butter of functional programming. This chapter took a deeper dive into the functional style of development using JavaScript's functional capabilities encoded in an extensible functional library called *Lodash*. This style favors a streamlined and flow-based model where high-level operations can be chained together as a sequence of steps, which contain the business logic needed to arrive at your result.

It's undeniable that writing flow-based code also benefits reusability and modularization, but I've only scratched the surface. I'll take this idea of flow-based programming to the next level in chapter 4, where I'll focus on constructing real function pipelines.

3.6 Summary

- You can write extensible code with the higher-order functions `map`, `reduce`, and `filter`.
- *Lodash* is a vehicle for data processing, creating programs via control chains where data flows and transformations are clearly demarcated.
- Functional programming's declarative style creates code that's easier to reason about.
- Mapping high-level abstractions to a SQL vocabulary reveals a deeper understanding of your data.
- Recursion solves self-similar problems and is required to parse through recursively defined data structures.

Functional Programming in JavaScript

Luis Atencio

In complex web applications, the low-level details of your JavaScript code can obscure the workings of the system as a whole. As a coding style, functional programming (FP) promotes loosely coupled relationships among the components of your application, making the big picture easier to design, communicate, and maintain.

Functional Programming in JavaScript teaches you techniques to improve your web applications—their extensibility, modularity, reusability, and testability, as well as their performance. This easy-to-read book uses concrete examples and clear explanations to show you how to use functional programming in real life. If you're new to functional programming, you'll appreciate this guide's many insightful comparisons to imperative or object-oriented programming that help you understand functional design. By the end, you'll think about application design in a fresh new way, and you may even grow to appreciate monads!

What's Inside

- High-value FP techniques for real-world uses
- Using FP where it makes the most sense
- Separating the logic of your system from implementation details
- FP-style error handling, testing, and debugging
- All code samples use JavaScript ES6 (ES 2015)

Written for developers with a solid grasp of JavaScript fundamentals and web application design.

Luis Atencio is a software engineer and architect building enterprise applications in Java, PHP, and JavaScript.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/functional-programming-in-javascript



"This book transformed the way that I think about and write JavaScript."

—Andrew Meredith
Intrinsitech Corporation

"Easy to navigate, with real-life examples."

—Amy Teng, Dell

"Now, this is the way to write JavaScript!"

—William E. Wheeler, West Corporation

"After reading this book, I revisited how I approached coding and was able to retrain my mind using better methods and techniques."

—Tanner Slayton Sr.
Microsoft Corporation



MANNING

US \$ 44.99 | Can \$ 51.99 (eBook included)

ISBN-13: 978-1-61729-282-8
ISBN-10: 1-61729-282-6



9 781617 292828