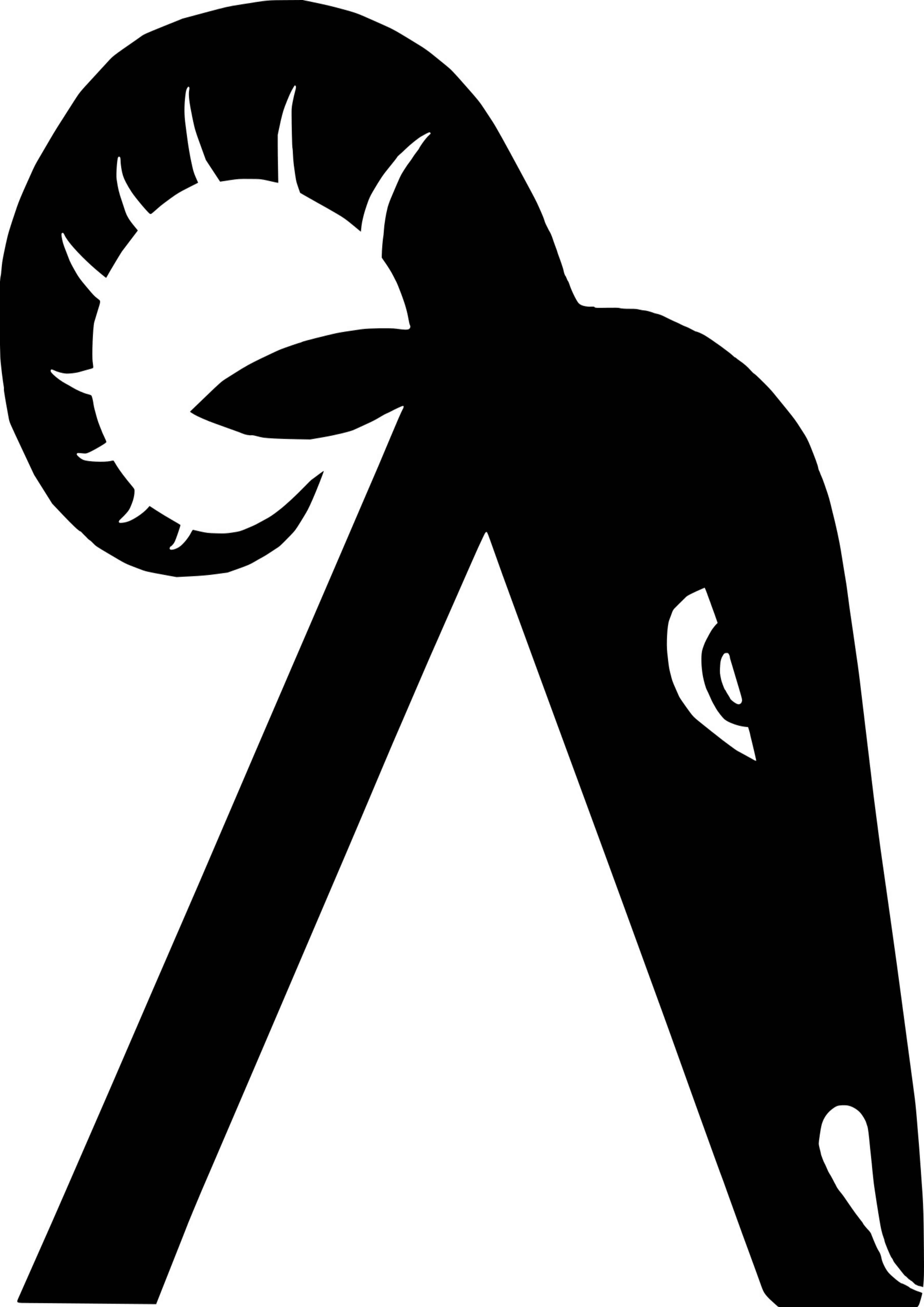

Table of Contents

Ramda

Ramda is a "practical functional library for Javascript programmers."

It is designed to bring many of the usual techniques of functional programming to the world of Javascript. Its goals are modest: to make it easy to work with small, standard functions against consistent data types such as lists, to make it easy to work with immutable data, to make it easy to compose applications function-by-function, composing simpler functions into more sophisticated workflows.

Ramda also has a few more advanced techniques up its sleeve. Its delegation mechanism allows us to turn many OO-style APIs into more functional versions, its lenses allow us to simply focus attention on some specific part of a data structure, and its transducers allow us to combine many operations on every element of a list of data into a single operation per element.

But at its heart, Ramda is a simple library. It is not a framework that dictates how we build our applications. Instead it is a low-level toolkit that offers a collection of functional tools we believe to be helpful.

Installation

To use with node:

```
$ npm install ramda
```

Then in the console:

```
var R = require('ramda');
```

Or use any individual functions via

```
var map = require('ramda')/src/map;
```

Note that internal dependencies will be resolved for you here.

To use directly in the browser:

```
<script src="path/to/yourCopyOf/ramda.js"></script>
```

or the minified version:

```
<script src="path/to/yourCopyOf/ramda.min.js"></script>
```

or from a CDN, either cdnjs:

```
<script src="//cdnjs.cloudflare.com/ajax/libs/ramda/0.16.0/ramda.min.js"></script>
```

or one of the below links from jsDelivr:

```
<script src="//cdn.jsdelivr.net/ramda/0.16.0/ramda.min.js"></script>
<script src="//cdn.jsdelivr.net/ramda/0.16/ramda.min.js"></script>
<script src="//cdn.jsdelivr.net/ramda/latest/ramda.min.js"></script>
```

(note that using `latest` is taking a significant risk that ramda API changes could break your code.)

These script tags add the variable `ramda` on the browser's global scope.

Or you can inject ramda into virtually any unsuspecting website using the bookmarklet.

Build

- on Unix-based platforms, `npm run build` updates **dist/ramda.js** and **dist/ramda.min.js**
- on Windows, write the output of `scripts/build --complete` to a temporary file, then rename the temporary file **dist/ramda.js**.

Partial Builds

It is possible to build Ramda with a subset of the functionality to reduce its file size. Ramda's build system supports this with command line flags. For example if you're using `R.compose`, `R.reduce`, and `R.filter` you can create a partial build with:

```
./scripts/build -- src/compose.js src/reduce.js src/filter.js > dist/ramda.custom.js
```

This requires having Node/io.js installed.

A Note on Signatures

TODO

Core Features

Ramda does not try to be everything to everyone. Although there are some Advanced features, for the most part Ramda focuses on some simple ideas, and leaves it to the user to decide how to put them together. These features are not everything that one would want in a functional language, but they do constitute a solid functional programming core.

In the section on Small Functions on common structures, we discuss how having a large number of common functions that will run consistently against simple resuable data structures -- an idea very much at odds with Object Oriented programming's notion of having many data structures, each with a small number of custom functions -- how this idea makes thinking about our systems significantly easier.

Javascript does little to help us work with immutable data. It's all too easy to write functions that maintain reference equality but alter the actual contents of user objects beyond recognition. Ramda helps by guaranteeing that its functions never mutate any input data. We consider the implications of this in Immutability of user data.

Another core functional programming concern is for functions to be referentially transparent[1]. While Ramda, does not try to enforce this for you, its functions themselves have no side-effects, and very few Ramda functions are written with side-effects in mind. We investigate this more fully in Avoiding side-effects.

In Composing functions we hit the core of Ramda's strength: how to chain together multiple functions to make increasingly sophisticated systems out of simple parts, analyzing a number of different forms of functional composition supported by Ramda.

In order to effectively compomse functions, it will help a great deal to be able to easily change one function into another by partially applying some of its parameters. We will discuss more sophisticated ways to do this in Chapter 2: Functions, but in Currying everywhere we evaluate the most common mechanism for this in Ramda, the simple `curry` function.

Finally, in Dispatching we examine how Ramda wraps certain other APIs in a functional façade, to allow you to more easily integrate into a functional programming workflow.

Small functions on common structures

> It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures. -- *Alan J. Perlis*

Functional programming is best paired with a few consistent data structures. The largest set of Ramda's functions operate on *lists*. We'll discuss lists in much more detail in Chapter 2, but at the moment, we can just think about simple dense arrays. Let's start with this data:

```
var lineItems = [
  {
    productId: "783490",
    description: "Rubik's Cube, 3x3x3",
    price: 12.99,
    quantity: 1
  },
  {
    productId: "134672",
    description: "Sudoku Book",
    price: 6.50,
    quantity: 3
  },
  {
    productId: "162075",
    description: "Sam Lloyd's 15 puzzle",
    price: 3.25,
    quantity: 5
  },

];
```

We can run various functions against this structure:

```
R.pluck('description')(lineItems);
//=> ["Rubik's Cube 3x3x3", "Sudoku Book", "Sam Lloyd's 15 puzzle"]

R.map(item => item.quantity * item.price)(lineItems);
//=> [12.99, 19.50, 16.25]

R.reduce((total, item) => total + item.price * item.quantity, 0)(lineItems);
//=> 48.74

R.filter(item => item.price < 10.00, lineItems);
//=> [
//     {"description": "Sudoku Book", "price": 6.5,
//        "productId": "134672", "quantity": 3},
//     {"description": "Sam Lloyd's 15 puzzle", "price": 3.25,
//        "productId": "162075", "quantity": 5}
// ]

R.compose(R.reduce(R.add, 0), R.pluck('quantity'))(lineItems);
//=> 9

R.project(['productId', 'quantity'])(lineItems);
//=> [
//    {productId: "783490", quantity: 1},
//    {productId: "134672", quantity: 3},
//    {productId: "162075", quantity: 5}
// ]
```

But we can also make reusable functions out of these:

```
// descriptions :: [LineItem] -> [String]
```

```
const descriptions = R.pluck('description');

// lineItemTotals :: [LineItem] -> [Number]
const lineItemTotals = R.map(item => item.quantity * item.price);

// invoiceTotal :: [LineItem] -> Number
const invoiceTotal = reduce((total, item) => total + item.price * item.quantity,
0)

// inexpensive :: [LineItem] -> [LineItem]
const inexpensive = R.filter(item => item.price < 10.00);

// totalItems :: [LineItem] -> Number
const totalItems = R.compose(R.reduce(R.add, 0), R.pluck('quantity'));

// warehoustPickList :: [LineItem] -> [{productId, quantity}]
const warehousePickList = R.project(['productId', 'quantity'])
```

(If these Haskell-style signatures are confusing, we have a short primer available.)

Now we can use these functions to work with *any* list of `LineItem`s. Some, such as `inexpensive` even further return new lists of `LineItem`s. Many others return different sorts of lists, such as lists of descriptions, of item totals, or of simplified versions of a `LineItem`, as displayed by `warehousePickList`.

Working with common structures like this make it very easy to build complex systems through combinations of simple tools. Let's take another look at our totaling fuctions. These functions are not too complicated, but they really combine several steps into one. We might be better off breaking them down:

```
// itemTotal :: LineItem -> Number
const itemTotal = (item) => item.price * item.quantity;

// lineItemTotals :: [LineItem] -> [Number]
const lineItemTotals = R.map(itemTotal);

// invoiceTotal :: [LineItem] -> Number
const invoiceTotal = R.pipe(lineItemTotals, R.sum);
```

Now we're getting to a set of very easily reusable functions with clear purposes. And although each one operates on either a `LineItem` or a list of `LineItem`s, the Ramda functions they use (`map` and `sum`) operate on lists. (We'll see more about `pipe` in the section on Composition.) Because we keep getting lists back from intermediate functions, we can continue to use familiar functions to operate on them.

The largest set of Ramda functions operate on lists of elements. But there are others for Strings, for plain Objects, for Functions, and some for logical combinations. The main point is that Ramda tries to create many useful functions for these few types, generally returning another element from one of these types. These common interfaces is how functional systems are pieced together.

Immutability

TODO

Avoiding side-effects

TODO

Composition

TODO

Currying

TODO

Dispatching

TODO

TODO

Currying

TODO

TODO

TODO

TODO

TODO

Working with lists

Arrays as lists

Strings as (sometimes) close-enough

Building lists: `range`, `repeat`, `times`, `keys`, `toPairs`

Slicing and dicing lists: `take`, `drop`, `append`, `prepend`, `concat`

Tweaking lists

List to list transformations: `map`, `pluck`, `filter`, `aperture`, `uniq`, `zip`, `partition`

Transforming lists to single values: `reduce`, `find`, `nth`, `all/any`