Chapter 04: Currying

Can't Live If Livin' Is without You

My Dad once explained how there are certain things one can live without until one acquires them. A microwave is one such thing. Smart phones, another. The older folks among us will remember a fulfilling life sans internet. For me, currying is on this list.

The concept is simple: You can call a function with fewer arguments than it expects. It returns a function that takes the remaining arguments.

You can choose to call it all at once or simply feed in each argument piecemeal.

```
const add = x => y => x + y;
const increment = add(1);
const addTen = add(10);

increment(2); // 3
addTen(2); // 12
```

Here we've made a function `add` that takes one argument and returns a function. By calling it, the returned function remembers the first argument from then on via the closure. Calling it with both arguments all at once is a bit of a pain, however, so we can use a special helper function called `curry` to make defining and calling functions like this easier.

Let's set up a few curried functions for our enjoyment. From now on, we'll summon our `curry` function defined in the Appendix A - Essential Function Support.

```
const match = curry((what, s) => s.match(what));
const replace = curry((what, replacement, s) => s.replace(what, replacement));
const filter = curry((f, xs) => xs.filter(f));
const map = curry((f, xs) => xs.map(f));
```

The pattern I've followed is a simple, but important one. I've strategically positioned the data we're operating on (String, Array) as the last argument. It will become clear as to why upon use.

(The syntax `/r/g` is a regular expression that means *match every letter 'r'*. Read more about regular expressions if you like.)

```
match(/r/g, 'hello world'); // [ 'r' ]

const hasLetterR = match(/r/g); // x => x.match(/r/g)
hasLetterR('hello world'); // [ 'r' ]
hasLetterR('just j and s and t etc'); // null

filter(hasLetterR, ['rock and roll', 'smooth jazz']); // ['rock and roll']

const removeStringsWithoutRs = filter(hasLetterR); // xs => xs.filter(x => x.match(
removeStringsWithoutRs(['rock and roll', 'smooth jazz', 'drum circle']); // ['rock

const noVowels = replace(/[aeiou]/ig); // (r,x) => x.replace(/[aeiou]/ig, r)
const censored = noVowels('*'); // x => x.replace(/[aeiou]/ig, '*')
censored('Chocolate Rain'); // 'Ch*c*l*t* R**n'
```

What's demonstrated here is the ability to "pre-load" a function with an argument or two in order to receive a new function that remembers those arguments.

I encourage you to clone the Mostly Adequate repository (`git clone https://github.com/MostlyAdequate/mostly-adequate-guide.git`), copy the code above and have a
go at it in the REPL. The curry function (and actually anything defined in the appendixes) has been made available from the `exercises/support.js` module.

More Than a Pun / Special Sauce

Currying is useful for many things. We can make new functions just by giving our base functions some arguments as seen in `hasLetterR`, `removeStringsWithoutRs`, and `censored`.

We also have the ability to transform any function that works on single elements into a function that works on arrays simply by wrapping it with `map`:

```
const getChildren = x => x.childNodes;
const allTheChildren = map(getChildren);
```

Giving a function fewer arguments than it expects is typically called *partial application*. Partially applying a function can remove a lot of boiler plate code. Consider what the above `allTheChildren` function would be with the uncurried `map` from lodash (note the arguments are in a different order):

```
const allTheChildren = elements => map(elements, getChildren);
```

We typically don't define functions that work on arrays, because we can just call `map(getChildren)` inline. Same with `sort`, `filter`, and other higher order functions (a *higher order function* is a function that takes or returns a function).

When we spoke about *pure functions*, we said they take 1 input to 1 output. Currying does exactly this: each single argument returns a new function expecting the remaining arguments. That, old sport, is 1 input to 1 output.

No matter if the output is another function - it qualifies as pure. We do allow more than one argument at a time, but this is seen as merely removing the extra `()`'s for convenience.

In Summary

Currying is handy and I very much enjoy working with curried functions on a daily basis. It is a tool for the belt that makes functional programming less verbose and tedious.

We can make new, useful functions on the fly simply by passing in a few arguments and as a bonus, we've retained the mathematical function definition despite multiple arguments.

Let's acquire another essential tool called `compose`.

Chapter 05: Coding by Composing

Exercises

Note about Exercises

Throughout the book, you might encounter an 'Exercises' section like this one. Exercises can be
done directly in-browser provided you're reading from gitbook (recommended).

Note that, for all exercises of the book, you always have a handful of helper functions available in the global scope. Hence, anything that is defined in Appendix A, Appendix B and Appendix C is available for you! And, as
if it wasn't enough, some exercises will also define functions specific to the problem they present; as a matter of fact, consider them available as well.

> Hint: you can submit your solution by doing `Ctrl + Enter` in the embedded editor!

Running Exercises on Your Machine (optional)

Should you prefer to do exercises directly in files using your own editor:

- clone the repository (`git clone git@github.com:MostlyAdequate/mostly-adequate-guide.git`)
- go in the *exercises* section (`cd mostly-adequate-guide/exercises`)
- install the necessary plumbing using npm (`npm install`)
- complete answers by modifying the files named *exercises_** in the corresponding chapter's folder
- run the correction with npm (e.g. `npm run ch04`)

Unit tests will run against your answers and provide hints in case of mistake. By the by, the answers to the exercises are available in files named *answers_**.

Let's Practice!

{% exercise %}
Refactor to remove all arguments by partially applying the function.

{% initial src="./exercises/ch04/exercise_a.js#L3;" %}
```
js const words = str => split(' ', str);
```

{% solution src="./exercises/ch04/solution_a.js" %}
{% validation src="./exercises/ch04/validation_a.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}

{% exercise %}
Refactor to remove all arguments by partially applying the functions.

{% initial src="./exercises/ch04/exercise_b.js#L3;" %}
```
js const filterQs = xs => filter(x => match(/q/i, x), xs);
```

{% solution src="./exercises/ch04/solution_b.js" %}
{% validation src="./exercises/ch04/validation_b.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}

Considering the following function:

```
js const keepHighest = (x, y) => (x >= y ? x : y);
```

{% exercise %}
Refactor `max` to not reference any arguments using the helper function `keepHighest`.

{% initial src="./exercises/ch04/exercise_c.js#L7;" %}
```
js const max = xs => reduce((acc, x) => (x >= acc ? x : acc), -Infinity, xs);
```

{% solution src="./exercises/ch04/solution_c.js" %}
{% validation src="./exercises/ch04/validation_c.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}