

FP's Secret Sauce

An in-depth look at currying and its advantages. (5 min. read)

Currying

I hope the last section gave you a taste of currying, as we're dipping a bit more into it here.

Its main benefit is code reuse. Curried functions have a better chance of being useful in different scenarios, thus making them **DRY**er.

Let's say you have an **add** function

```
const add = (x, y) => {  
  console.log(`Adding ${x} and ${y}`);  
  
  return x + y;  
};  
  
const result = add(2, 3);  
  
console.log({ result });
```



What happens if you don't give **add** its required parameters? What if you give too little?

```
const add = (x, y) => {  
  console.log(`Adding ${x} and ${y}`);  
  
  return x + y;  
};  
  
const result = add(2);  
  
console.log({ result });
```



Not fun. You know this leads to weird bugs, especially in JavaScript where we do anything and everything. The problem sort of goes away when you curry `add`.

```
import { curry } from 'ramda';

const add = (x, y) => {
  console.log(`Adding ${x} and ${y}`);

  return x + y;
};

const curriedAdd = curry(add);

// returns a function
const result = curriedAdd(2);

console.log({ result });
```

Your eyes don't deceive you, `curriedAdd(2)` returns a function. It will keep returning a function until you supply the last parameter, `y`.

```
import { curry } from 'ramda';

const add = (x, y) => {
  console.log(`Adding ${x} and ${y}`);

  return x + y;
};

const curriedAdd = curry(add);
const result = curriedAdd(2);

console.log(
  result(),
  result(),
  result()
);

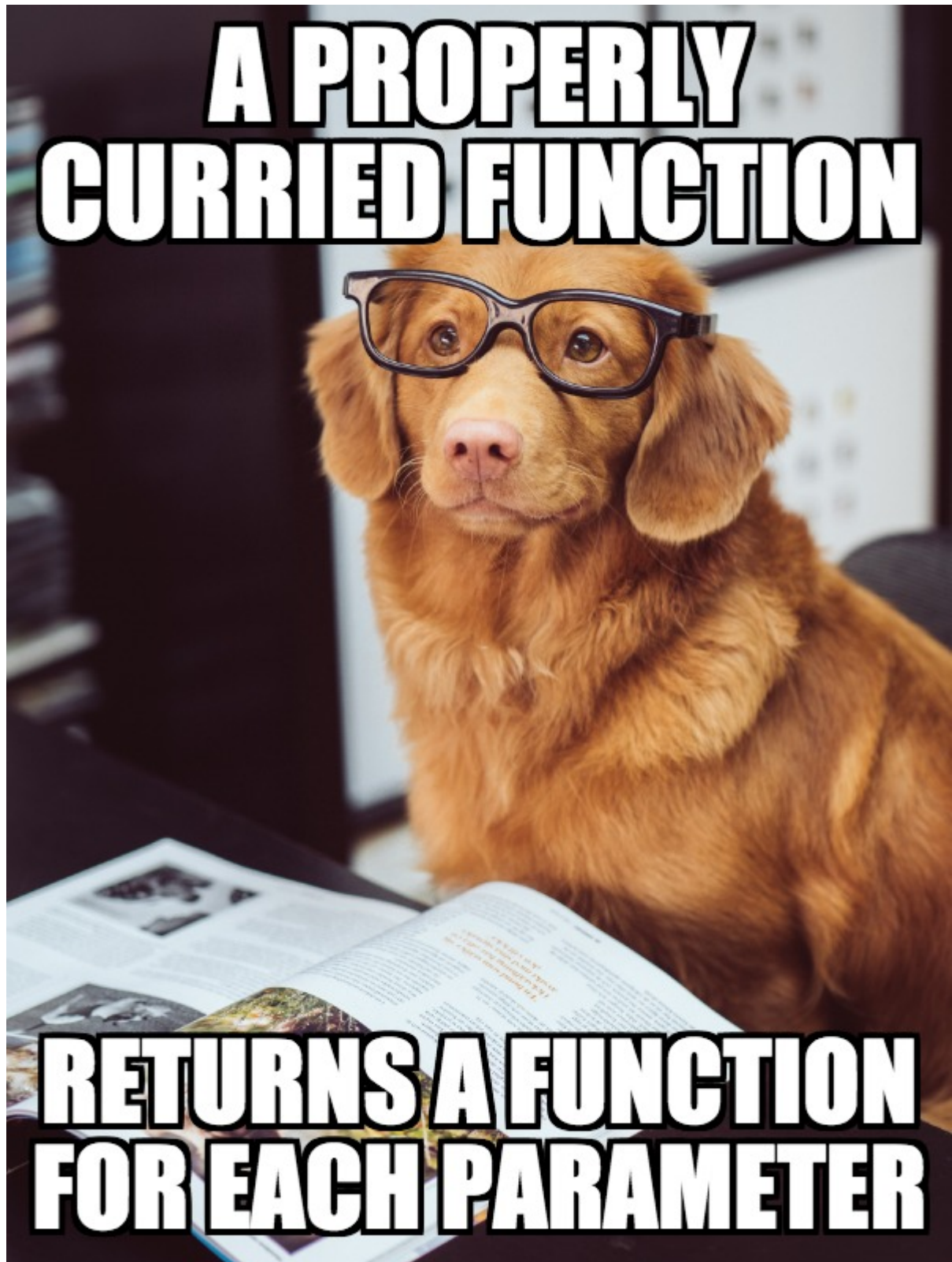
console.log("I won't stop returning functions until you give me Y! \n")
console.log("Okay dude, here's y = 3...\n");

console.log(result(3));

console.log(`\n-_-`)
```

Not Entirely Textbook

According to the FP books, currying means you return one new function per expected parameter.



So if you have an `add3` or `greet` function

```
const add3 = (x, y, z) => x + y + z;  
const greet = (greeting, first, last) => `${greeting} ${first} ${last}`;
```

```
const greet = (greeting, first, last) => `${greeting}, ${first} ${last}`;
```

“Properly” currying them looks like this

```
const curriedAdd3 = (x) => (y) => (z) => x + y + z;  
const curriedGreet = (greeting) => (first) => (last) => `${greeting}, ${first} ${last}`;
```

But let’s be honest, this sucks in JavaScript. I don’t want to call my functions like this

```
curriedAdd3(1)(2)(3); // 6  
curriedGreet('Hello')('John')('Doe'); // Hello, John Doe
```

Thankfully Ramda’s `curry` supports either style. You can use them like above or give all the arguments at once.

```
import { curry } from 'ramda';  
  
const greet = (greeting, first, last) => `${greeting}, ${first} ${last}`;  
const curriedGreet = curry(greet);  
  
// Supply all arguments at once  
const result1 = curriedGreet('Hello', 'John', 'Doe');  
  
// Or break it up  
const result2 = curriedGreet('Hello')('John')('Doe');  
  
// Get weird if you want  
const result3 = curriedGreet('Hello')()()('John', 'Doe');  
  
// Store curried function as variable  
const greetJohn = curriedGreet('Hello', 'John')  
const result4 = greetJohn('Smith');  
  
console.log({  
  result1,  
  result2,  
  result3,  
  result4  
});
```



Summary

- Curried functions keep returning functions until all required parameters are given.

- This lets you partially apply a function and create more specialized versions.
- “Properly” curried functions return one function per argument. `(a) => (b) => (c)` instead of `(a, b, c)`.
- But they’re a bit impractical so Ramda lets you supply them normally, or one at a time.