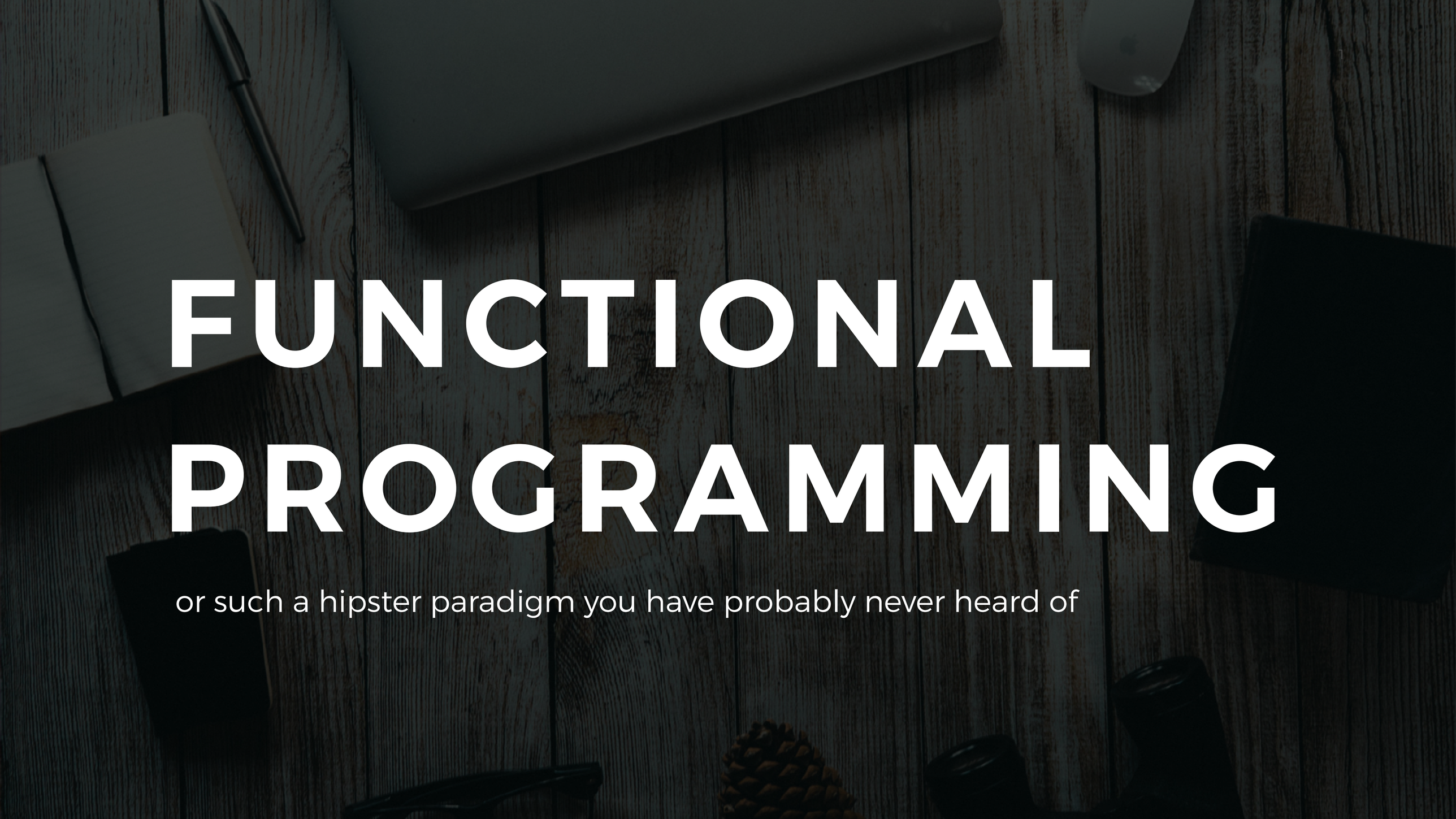


# FUNCTIONAL PROGRAMMING

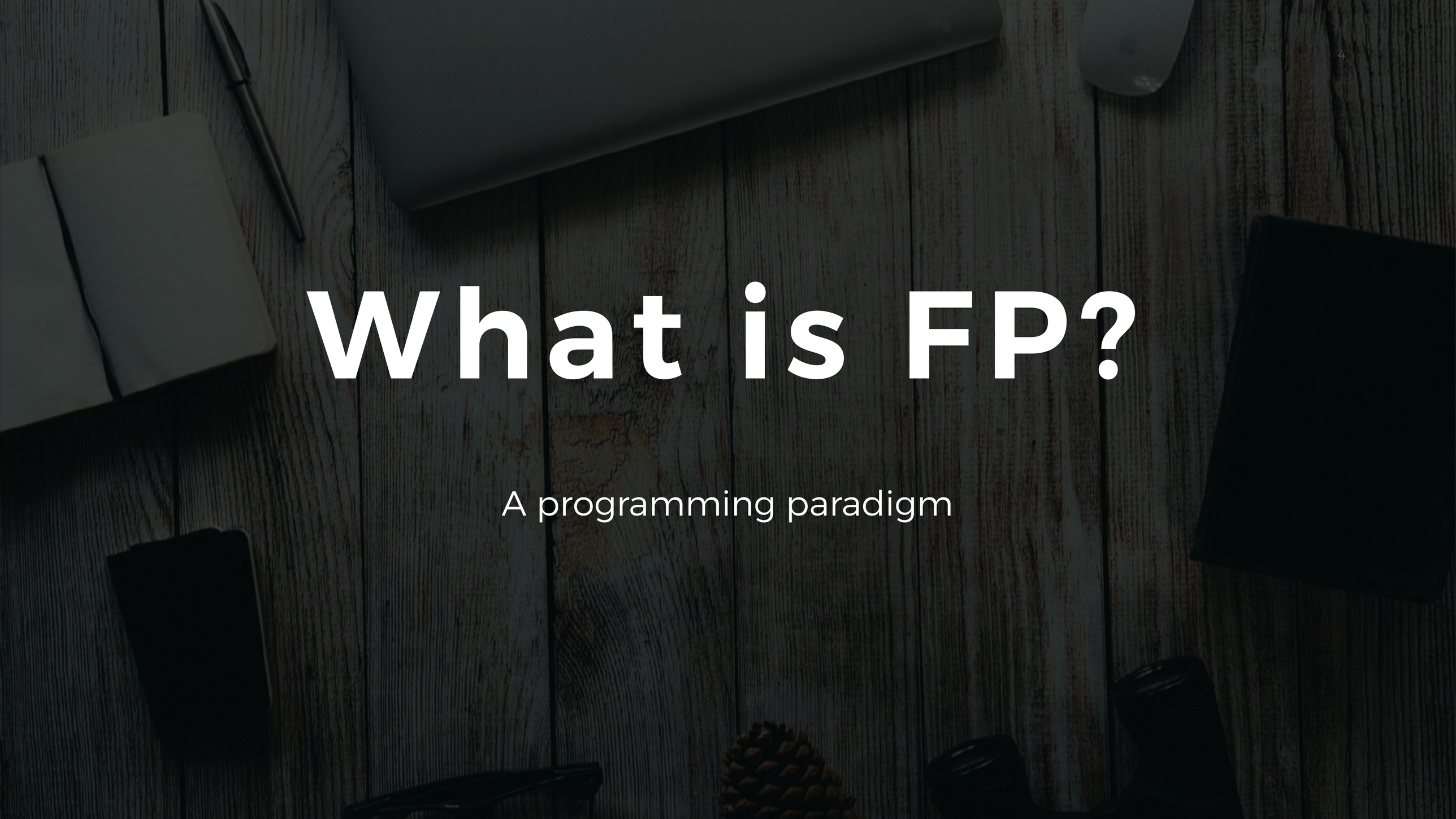
or such a hipster paradigm you have probably never heard of

# Who is this guy?

- SW Engineer at **Ooyala**
- JS fan since JQuery and MooTools were mainstream
- Developed some libraries for React Native
- **Can exit Vim**

# What is FP?

A new buzzword that is hard to pronounce



# What is FP?

A programming paradigm

- Not as new as you could think
- Has its origins in Lambda calculus (1930)
- Lisp (1962)
- Haskell (1992)

# Why FP?

# Less Bugs

- Easier to understand
- Easier to test

# Less Time

Reuse code

$$x + y + z = FP$$

(Kinda)

# FP has a steep learning curve


**Keep Calm**

# Pure Functions

# Pure functions

- A function that given the same input, will always return the same output
- Don’t have **side effects**

```
let xs = [1 , 2, 3, 4, 5];

// Slice
xs.slice(0, 3);
xs.slice(0, 3);
xs.slice(0, 3);

// Splice
xs.splice(0, 3);
xs.splice(0, 3);
xs.splice(0, 3);
```

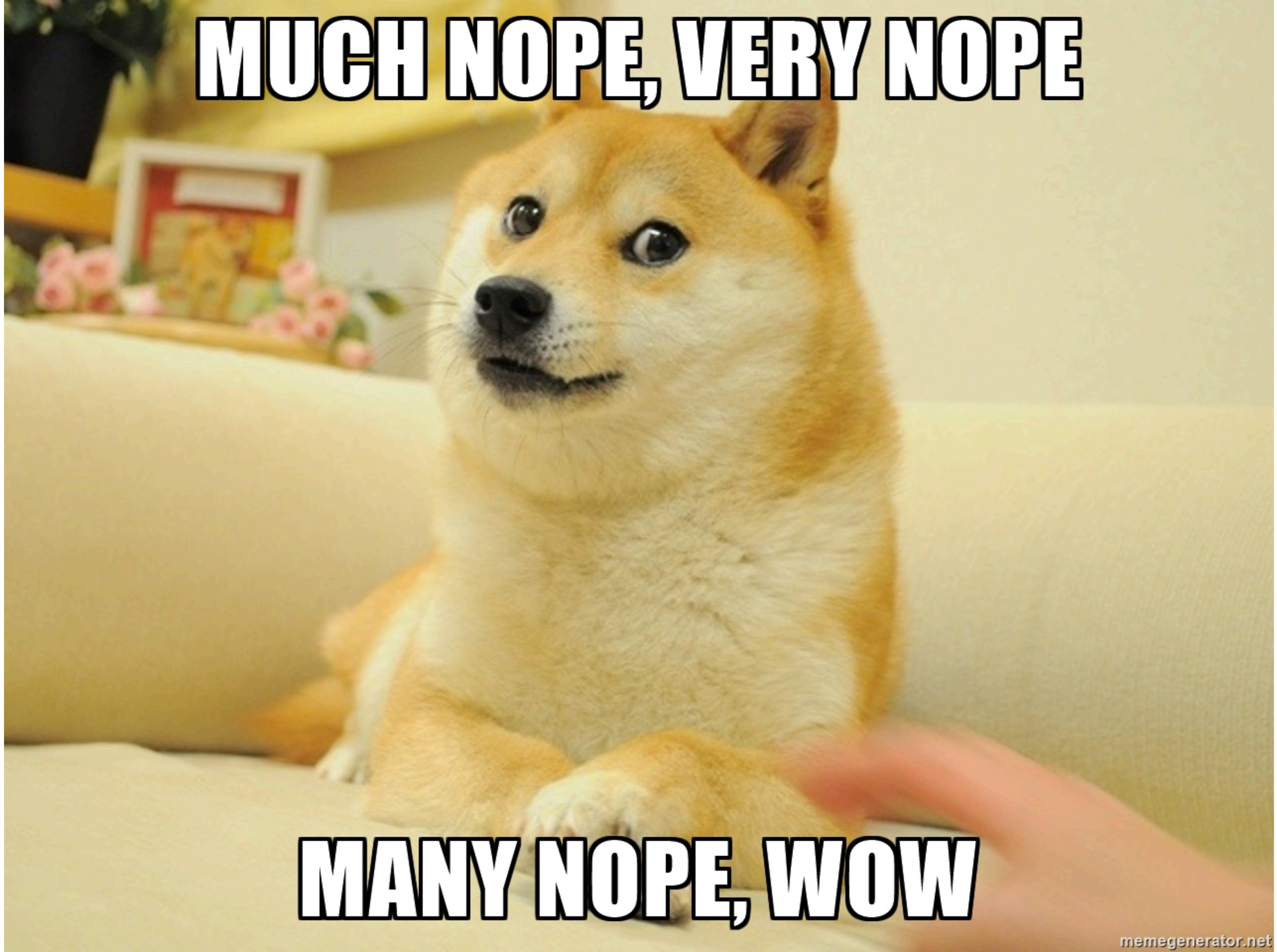
```
let xs = [1 , 2, 3, 4, 5];

// pure
xs.slice(0, 3); // -> [1, 2, 3]
xs.slice(0, 3); // -> [1, 2, 3]
xs.slice(0, 3); // -> [1, 2, 3]

//impure
xs.splice(0, 3); // -> [1, 2, 3]
xs.splice(0, 3); // -> [4, 5]
xs.splice(0, 3); // -> []
```

```
// impure
const minimumAge = 18;

let isAdult = function(age) {
  return age >= minimumAge;
}

// pure
let isAdult = function(age) {
  const minimumAge = 18;
  return age >= minimumAge;
}
```

# Side effects

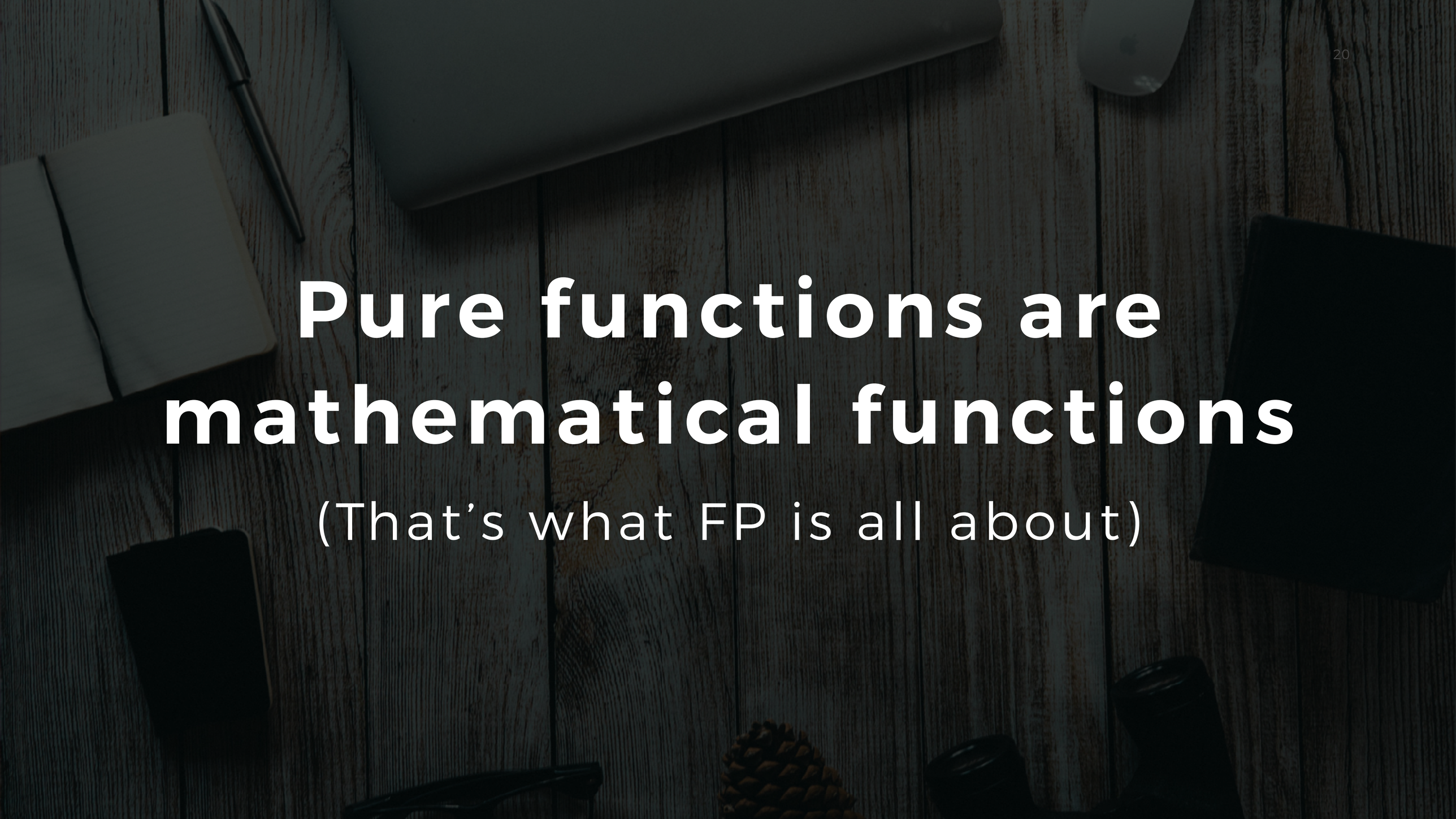
- Changing the file system
- Inserting a record in the database
- Making an http call
- Querying the DOM
- Accessing the system state

# Is it possible to live without side effects?


MUCH NOPE, VERY NOPE
MANY NOPE, WOW
memegenerator.net

# Side effects

- Side effects are a primary cause of incorrect behavior (bugs)
- They are not **forbidden**. We want to contain them and run them in a controlled way.



# Pure functions are mathematical functions

(That's what FP is all about)

# 2

# Immutability

# Immutability

- In math, $x = x + 1$ is invalid
- Once a variable takes a value, it shouldn't change again

# Is const immutable?

```
const x = 1;
```

```
const x = 1;
x = 2;

const y = [1];
y.push(2);

const z = {};
z.test = "test";
```

# Constant !== Immutable

```
const x = 1;
x = 2; // -> Error: Assignment to constant variable.

const y = [1];
y.push(2); // -> [1, 2]

const z = {};
z.test = "test"; // { test: "test" }
```

# Achieving immutability

- **Deep freeze** (Object.freeze is not enough)
- **Immutable.js**

# 3

# Higher order Functions

# Higher order functions

- A function is a first-class citizen of the language
- Functions are values!
- And they can be passed into other functions

# All non-insane programming languages have functions

```
function double(x) {
 return x * 2;
}
```

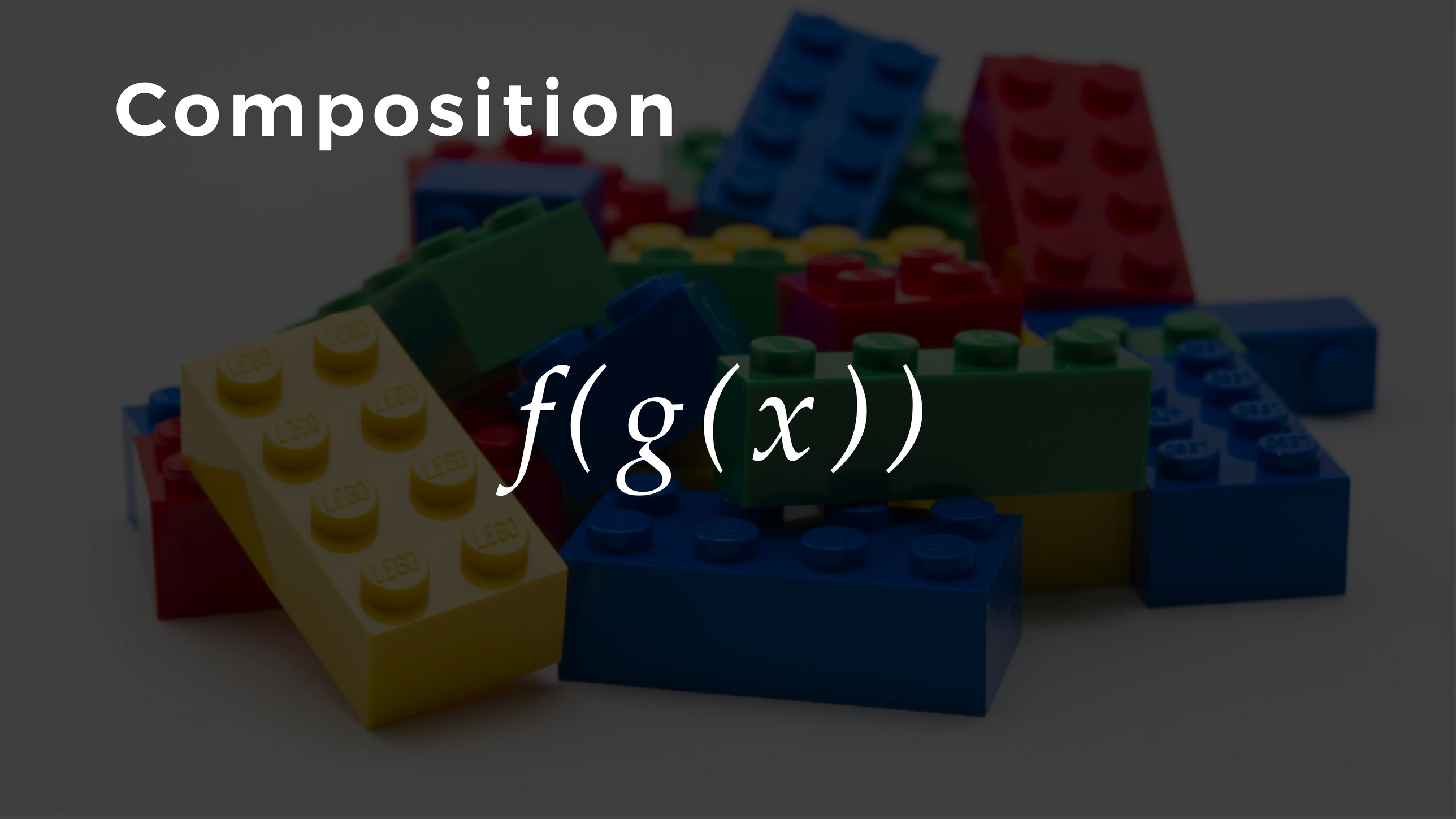
# But… not all of them can do this

```
let double = function(x) {
 return x * 2;
}
```

# Or this…

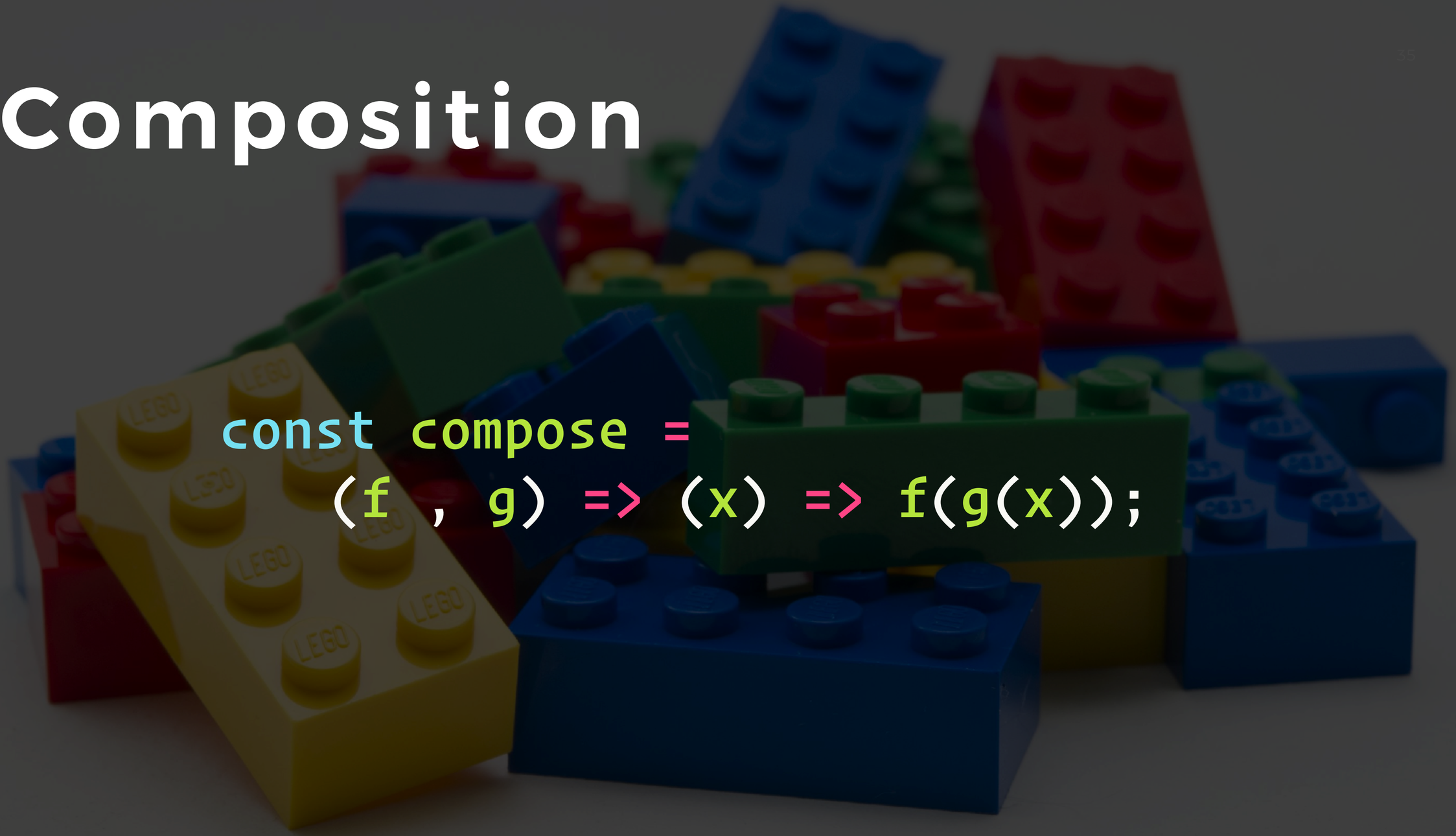
```
let double = function(x) {
  return x * 2;
}

let pizza = double;

pizza(10);
// 20
```

# Composition




$$f(g(x))$$

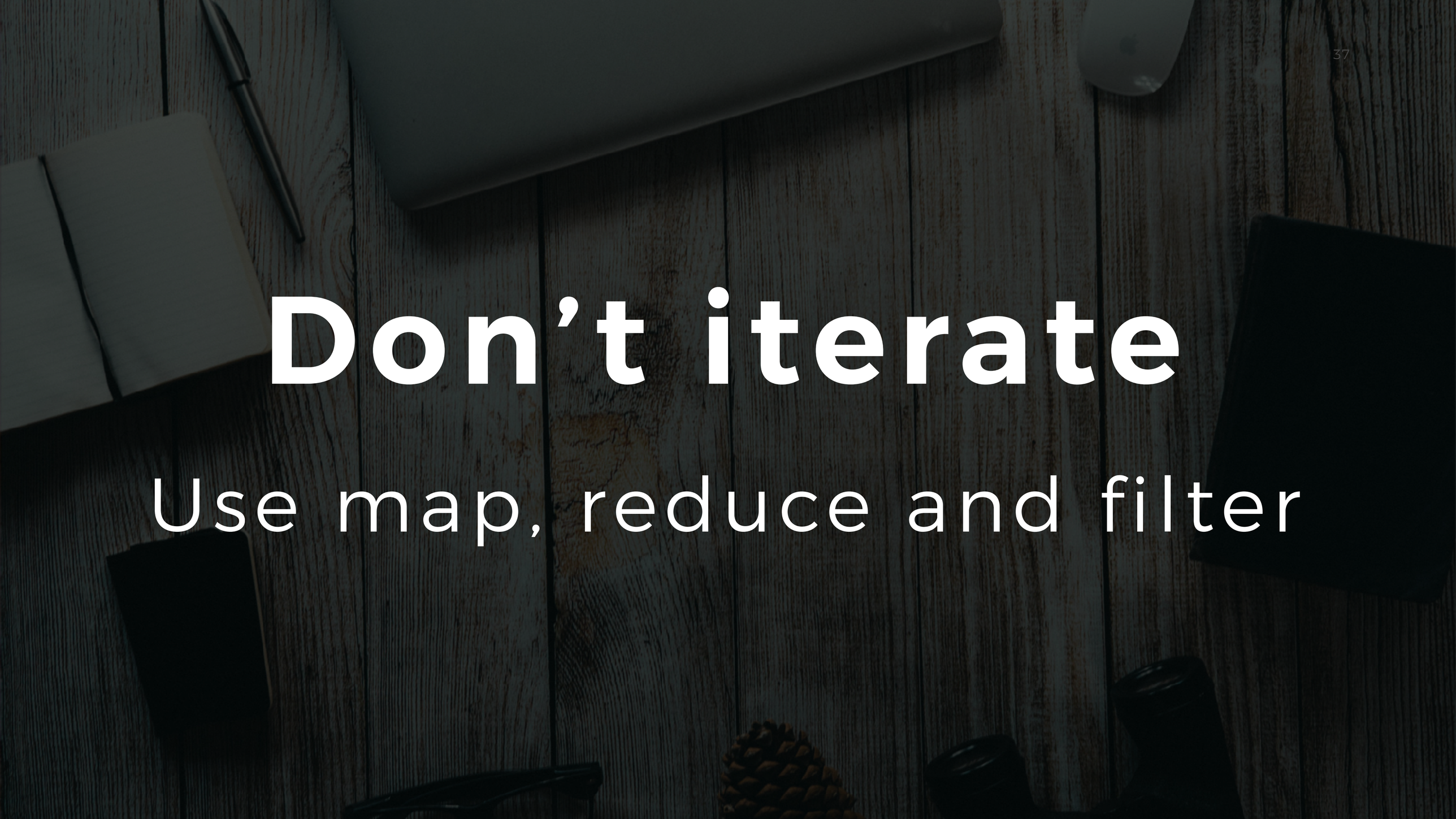
# Composition



```
const compose =
    (f , g) => (x) => f(g(x));
```

# Composition

```
const toUpperCase = x => x.toUpperCase();
const exclaim = x => `${x}!`;

// flow is the compose function from lodash
const shout = _.flow(exclaim, toUpperCase);

shout('hello world’);
// -> HELLO WORLD!
```



# Don’t iterate

Use map, reduce and filter

```
map([🌽, 🐮, 🐔], cook)
=> [🍿, 🍔, 🍳]

filter([🍿, 🍔, 🍳], isVegetarian)
=>  [🍿, 🍳]

reduce([🍿, 🍳], eat)
=> 💩
```

# Keep in your mind

```
let animals = [
  {name: 'Fido', species: 'dog'},
  {name: 'Epona', species: 'horse'},
  {name: 'Ursula', species: 'cat'},
  {name: 'Beethoven', species: 'dog'},
  {name: 'Alfred', species: 'fish'},
]
```

# Old fashioned way

```
let dogs = [];
for(let i = 0; i < animals.length; i++){
  if(animals[i].species === 'dog')
    dogs.push(animals[i]);
}
```

# Functional way

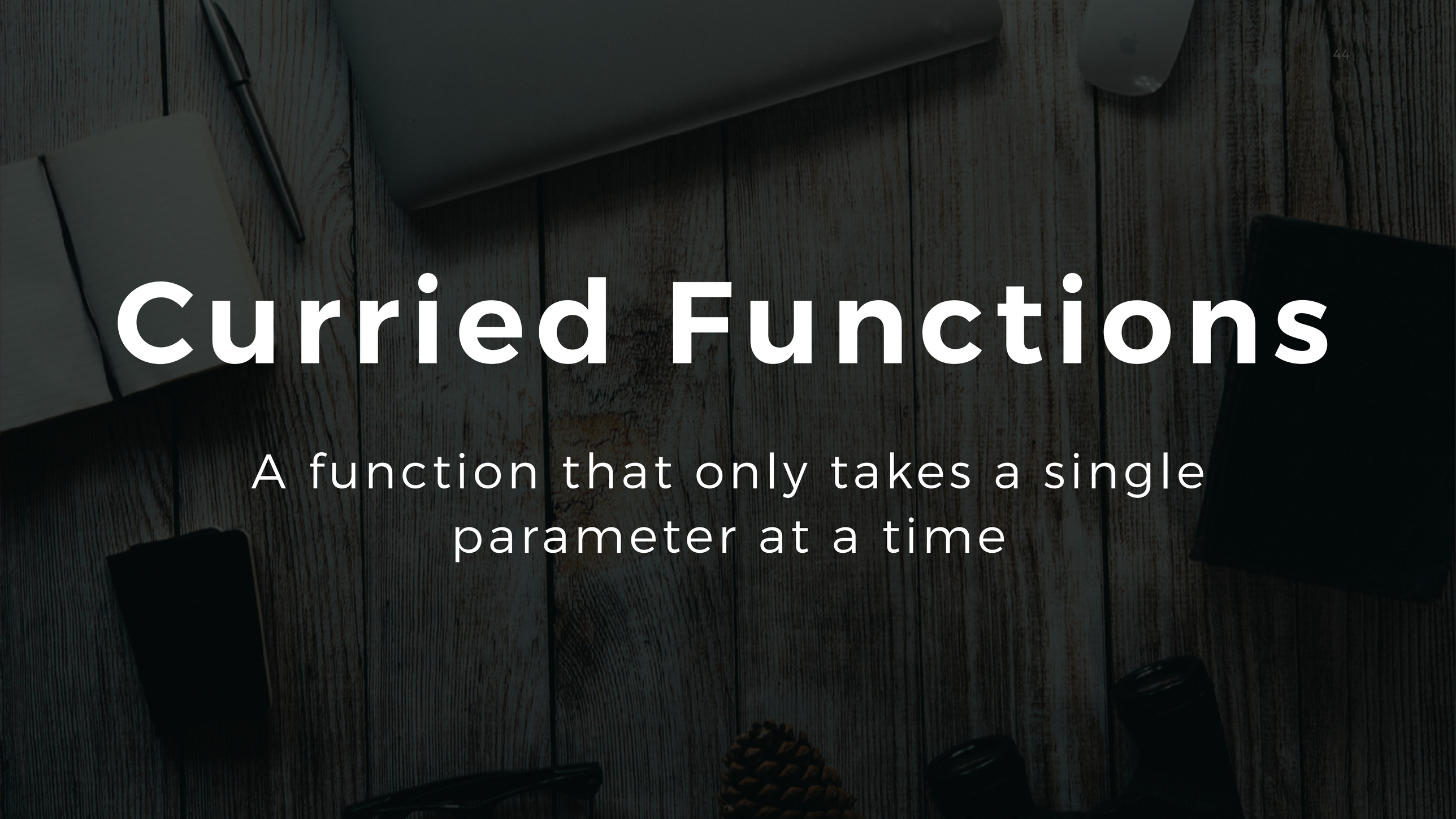
```
let dogs = animals.filter( x => x.species === 'dog' );

/*let dogs = [];
for(let i = 0; i < animals.length; i++){
if(animals[i].species === 'dog'){
dogs.push(animals[i]);
}
}*/
```

# Better Functional Way

```
const isDog = x => x.species === ‘dog';

let dogs = animals.filter(isDog);
let otherAnimals = _.reject(animals, isDog);
```

# Better Functional Way?

```
const isDog = x => x.species === "dog";
const isCat = x => x.species === "cat";

let dogs = animals.filter(isDog);
let cats = animals.filter(isCat);
```

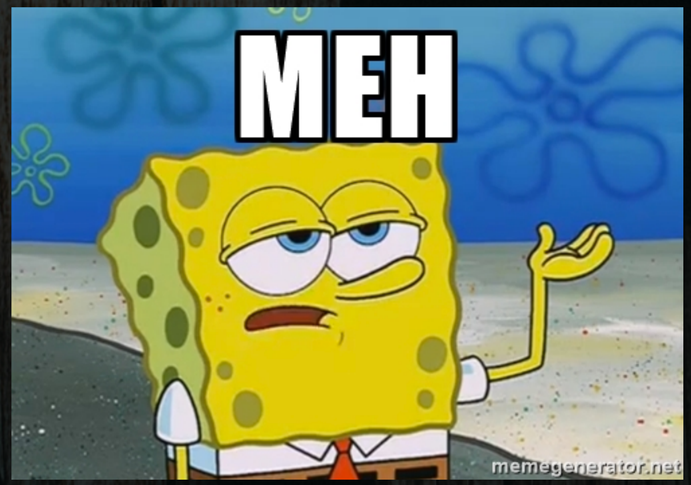


# Curried Functions

A function that only takes a single parameter at a time

# Curried Functions

```
const add = function(x) {
 return function (y) {
   return x + y;
  }
}

add(5)(10);
// Or
add(5, 10);
```



# Curried Functions

```
const add = function(x) {
 return function (y) {
   return x + y;
  }
}

const add5 = add(5);

add5(10);
```

# Better Functional Way?

```
const isDog = x => x.species === "dog";
const isCat = x => x.species === "cat";

let dogs = animals.filter(isDog);
let cats = animals.filter(isCat);
```

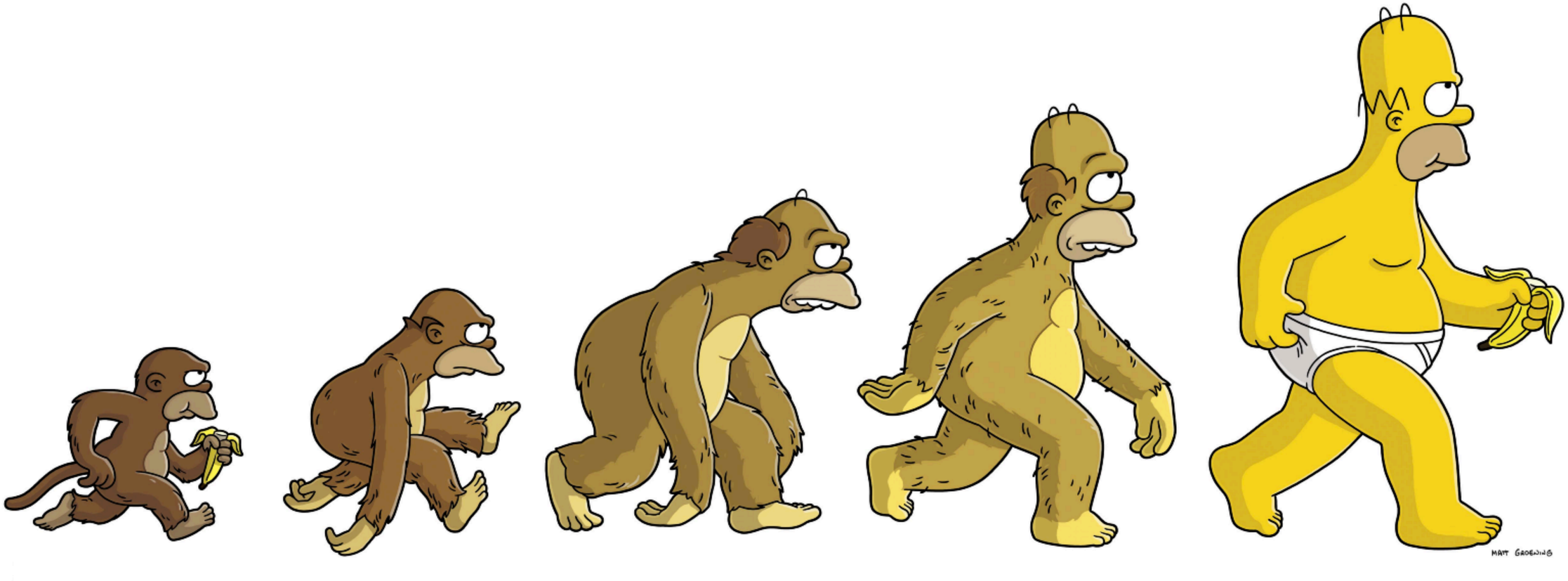
# A Better Curried Functional Way

```
const isSpecies = species => x => species === x.species;
const isDog = isSpecies('dog');
const isCat = isSpecies('cat');

let dogs = animals.filter(isDog);
let cats = animals.filter(isCat);
```

# A Better Curried Functional Way

```
const isSpecies =
    _.curry((species, x) => species === x.species);

const isDog = isSpecies('dog');
const isCat = isSpecies('cat');

let dogs = animals.filter(isDog);
let cats = animals.filter(isCat);
```


Lazy evaluation
Factories
Functors
Memoization
Monads
Recursion
Referential
transparency


MACHINE
ASSEMBLY
PROCEDURAL
OBJECT ORIENTED
FUNCTIONAL
MATT GROENING

# Keep in touch!

@carlyeah

carlyeah@me.com