Functional Programming

COMPOSITION
PURE FUNCTIONS
AVOID SIDE EFFECTS
AVOID SHARED STATE
AVOID MUTATING STATE
DECLARATIVE VS IMPERATIVE

CONCISE

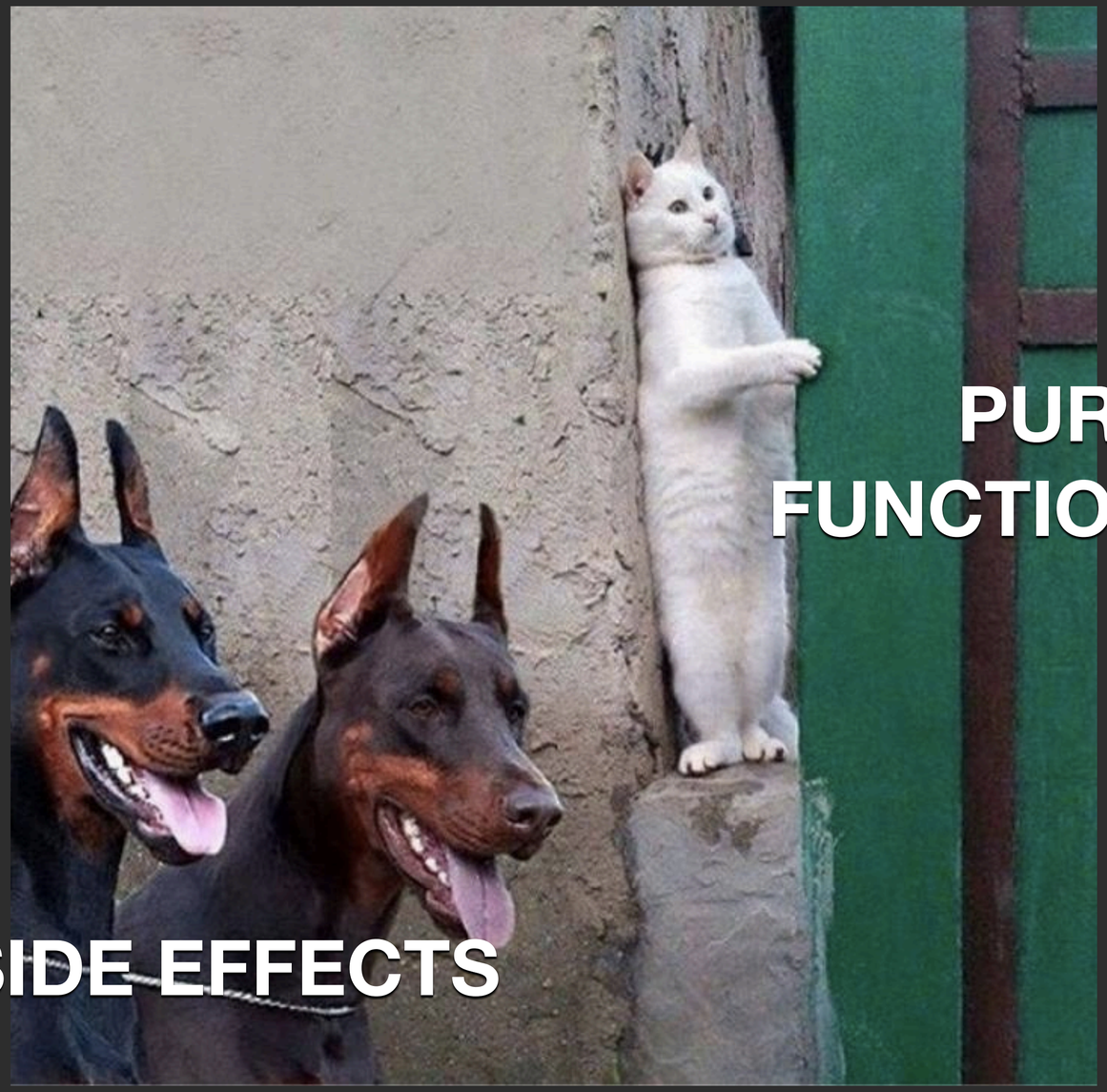
PREDICTABLE

TESTABLE


SPOT
MET
CBM
NYC

Pure functions

No side effects
Don’t assign to any outside variable
Don’t make any I/O operations
Don’t modify passed parameters
Return same output when given the same input


PURE
FUNCTION
SIDE EFFECTS

Pure functions

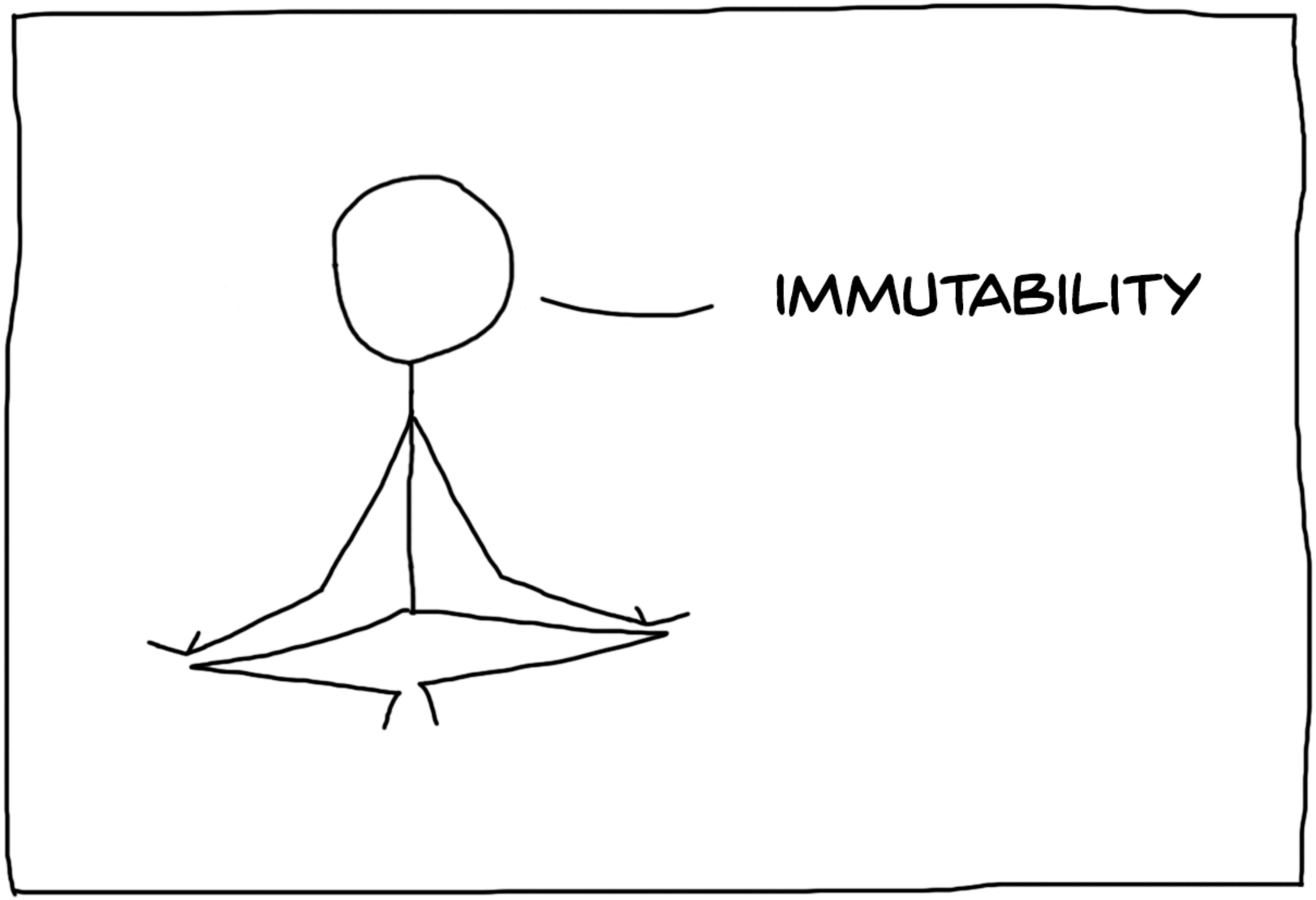
Referential Transparency

```
var x = Math.pow(2, 3) + 2;

var x = 8 + 2;
```

Pure functions

Easy to test
They don’t rely on any external state
Only input -> output

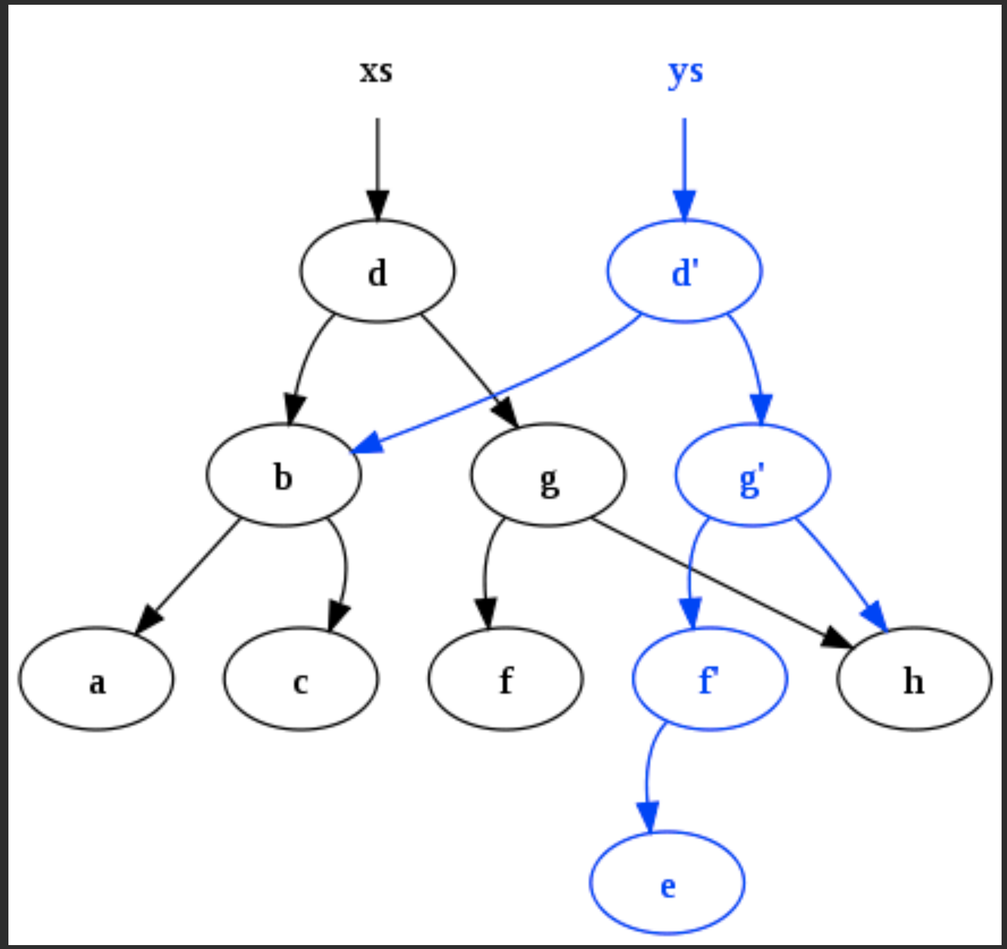
No side effects
Can be used in any context


IMMUTABILITY

Immutability

Immutability

Immutability



Declarative vs Imperative

Expressions over statements
“What to do“ instead of “How to do“

```
SELECT * FROM orders WHERE created_date < 1574792357;
```

Declarative vs Imperative

```
const apples = fruits.filter(f => f.name === 'apple')

const apples = []
for (let i = 0; i < fruits.length; i++) {
  if (fruits[i].name === 'apple') {
    apples.push(fruits[i])
  }
}
```

Currying

```
const add = (a, b) => a + b;
const curried = a => b => a + b;

const addTen = curried(10);

console.log(addTen(32)) // 42
```

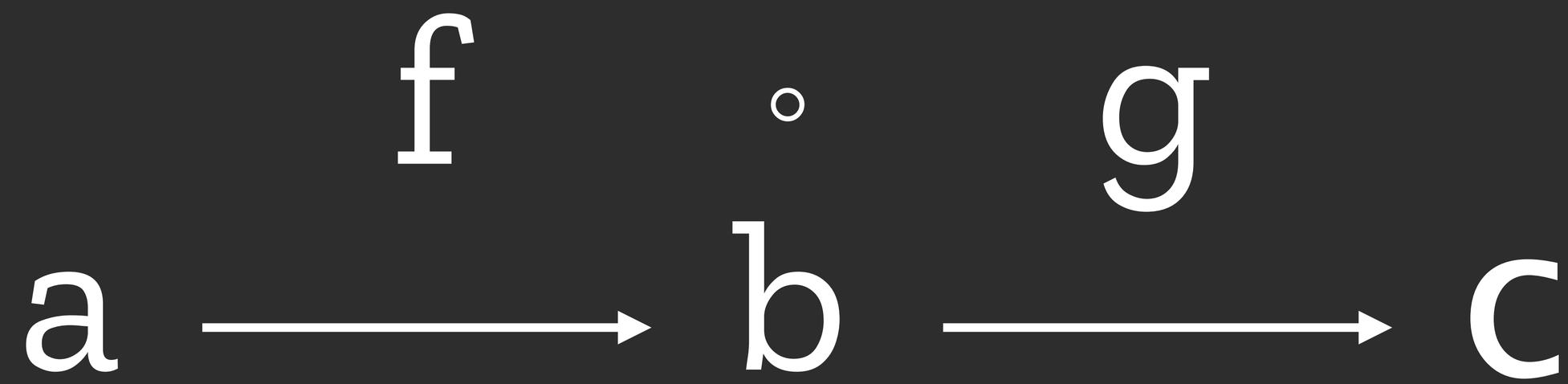
Currying

```
add(20)(22);

function curry(fn) {
  ...
}

const add = curry((a, b) => a + b);

console.log(add(1, 41)) // 42
console.log(add(22)(20)) // 42
```

Currying

```
add :: Int -> Int -> Int
add a b = a + b
```

compose(all(the(things())))


f
∘
g
a
b
c

Composition

```
function doubleSay(str) {
  return str + ', ' + str;
}
function capitalize(str) {
  return str[0].toUpperCase() + str.slice(1);
}
function exclaim(str) {
  return str + '!';
}
```

Composition

```
const result = exclaim(capitalize(doubleSay('hello')));

result //=> 'Hello, hello!'
```

Composition

```
const fn = compose(
  exclaim,
  capitalize,
  doubleSay
)
```

```
const fn = pipe(
  doubleSay,
  capitalize,
  exclaim
)
```

Composition

```
const result = 'hello'
  |> doubleSay
  |> capitalize
  |> exclaim;
```

STAGE 1: Pipeline operator proposal

Partial Application

```
const names = ['Alonzo Church', 'Haskell Curry', 'Alan Turing']

const join = curry((glue, xs) => xs.join(glue))
const split = curry((sep, s) => s.split(sep))
const map = curry((fn, xs) => xs.map(fn))
const tail = xs => xs[xs.length - 1]
const toLower = s => s.toLowerCase()

const fn = compose(
  join(''),
  map(toLower),
  map(tail),
  map(split(' '))
)

console.log(fn(names)) // alonzo-haskell-alan
```

Partial Application

```
const fn2 = compose(toLower, tail, split(' '))

const fn = compose(
  join(''),
  map(fn2)
)
```

Data last pattern

```
const names = ['Alonzo Church', 'Haskell Curry', 'Alan Turing']

const join = curry((glue, xs) => xs.join(glue))
const split = curry((sep, s) => s.split(sep))
const map = curry((fn, xs) => xs.map(fn))
const tail = xs => xs[xs.length - 1]
const toLower = s => s.toLowerCase()

const fn = compose(
  join(''),
  map(toLower),
  map(tail),
  map(split(' '))
)

console.log(fn(names)) // alonzo-haskell-alan
```

Data last pattern

```
const join = curry((xs, glue) => xs.join(glue))
const split = curry((s, sep) => s.split(sep))
const map = curry((xs, fn) => xs.map(fn))
const toLower = s => s.toLowerCase()

const fn = compose(
  xs => join(xs, ''),
  xs => map(xs, toLower),
  xs => map(xs, tail),
  xs => map(xs, s => split(s, ' '))
)
```

Point-free style

```
const fn = compose(
  join(''),
  map(toLower),
  map(tail),
  map(split(' '))
)
```

```
const fn = compose(
  exclaim,
  capitalize,
  doubleSay
)
```

```
const fn2 = compose(toLower, tail, split(' '))
```

How to ~~draw an owl~~

functional programming

1.

2.

write the rest of
the fucking program

f(x) = x

1. ~~Draw some circles~~

2. ~~Draw the rest of the fucking owl~~

function curry(fn)

```
function curry(fn) {
    const arity = fn.length;
    return function curried(...args) {
        if (args.length < arity) {
            return curried.bind(null, ...args);
        }

        return fn.call(null, ...args);
    };
}
```

function compose(...fns)

```
const compose = (...fns) => (...args) =>
    fns.reduceRight((res, fn) => fn(...res), args);
```