There is a Bluebird in my Talk that Wants to Get out

Rewriting JavaScript

THEWIZARDLUCAS

I'm Lucas

THEWIZARDLUCAS

I'm Lucas, The Wizard

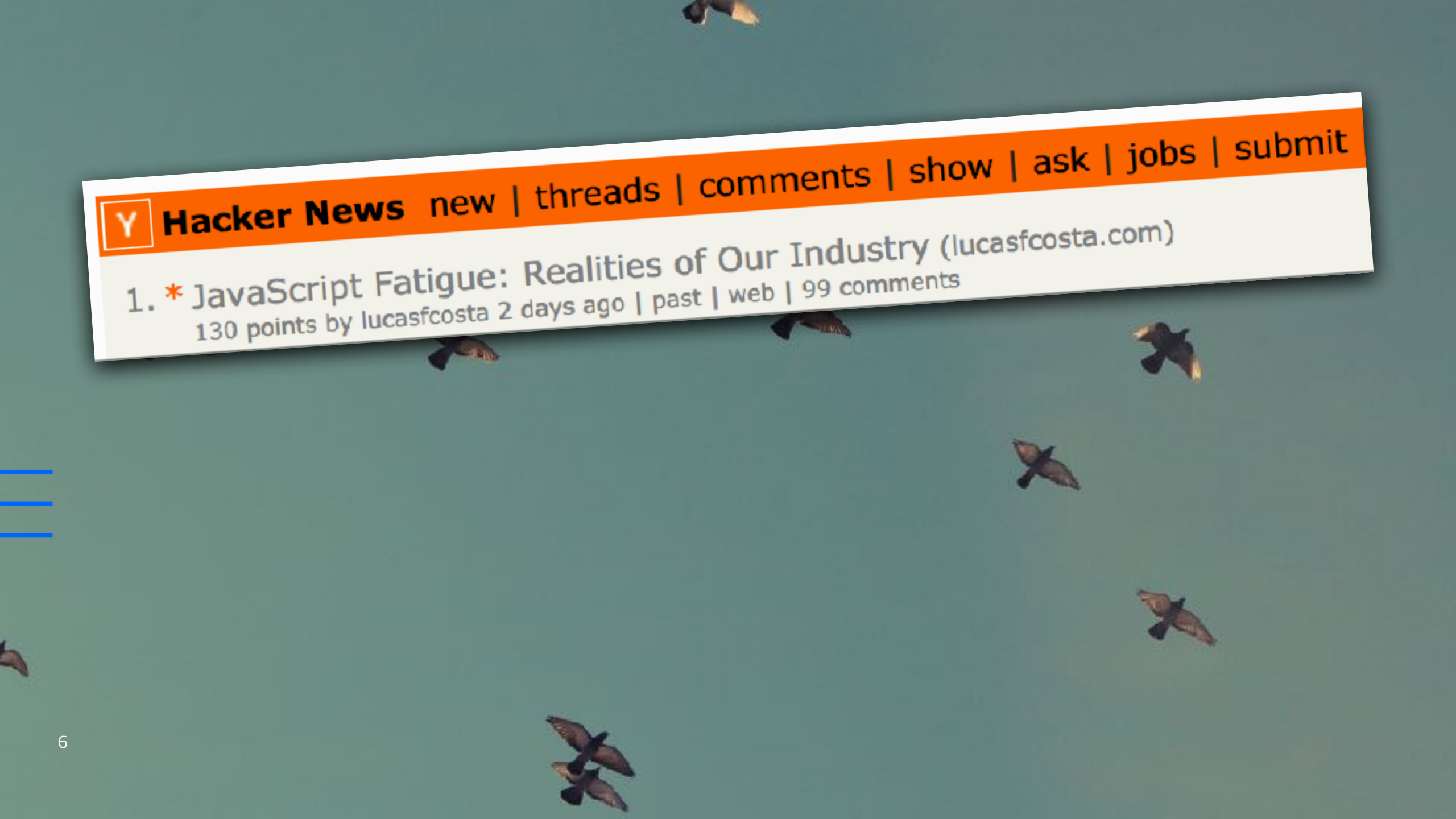
THEWIZARDLUCAS

@THEWIZARDLUCAS

lucasfcosta.com

@THEWIZARDLUCAS

lucasfcosta.com


Hacker News new | threads | comments | show | ask | jobs | submit
1. * JavaScript Fatigue: Realities of Our Industry (lucasfcosta.com)
130 points by lucasfcosta 2 days ago | past | web | 99 comments

rv77ax 2 days ago [-]

I bet the author use space instead of tab.

reply


Right, but will you talk about birds?

This is a talk about Lambda Calculus

Calculus !== Numbers

Fun

Fun

Functions

Lambda Calculus is what is behind Functional Programming


Is it useful?


stackoverflow

How helpful is knowing lambda calculus? [closed]

69

To all the people who know lambda calculus: What benefit has it bought you, regarding programming? Would you recommend that people learn it?

math | functional-programming | computer-science | lambda-calculus

26

If you want to program in any functional programming language, it's essential. I mean, how useful is it to know about Turing machines? Well, if you write C, the language paradigm is quite close to Turing machines -- you have an instruction pointer and a current instruction, and the machine takes some action in the current state, and then ambles along to the next instruction.

In a functional language, you simply can't think like that -- that's not the language paradigm. You have to think back to lambda calculus, and how terms are evaluated there. It will be much harder for you to be effective in a functional language if you don't know lambda calculus.

share improve this answer

answered Sep 22 '08 at 12:55

EfForEffort

54.7k ● 4 ● 26 ● 38

If you want to program in any functional programming language, it's essential.


essential.


λ

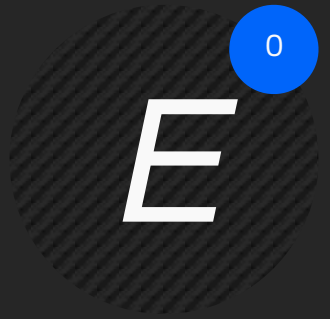
In Lambda Calculus

Everything is an expression

In Lambda Calculus

Everything evaluates to a value

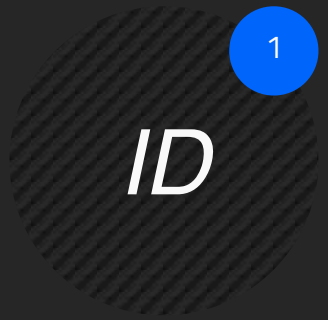
This is Lambda Calculus



Expression

RUINING JAVASCRIPT

This is Lambda Calculus



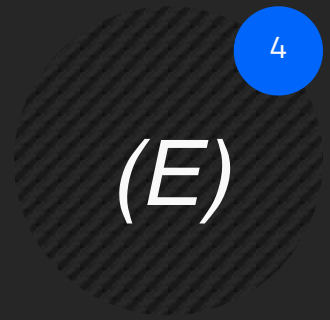
Identifier

Abstraction

Function definitions

Application

Invocation



Grouping

This is a lambda

$$\lambda a.\ a$$

This is a lambda

This is a lambda

a(arg)

a arg

Applying arg to a

This is a lambda

λa. a

arg

We get back arg

This is a lambda

$$\lambda a.\ a$$

It can only take a single argument!

This is a lambda

$$\lambda a.\ \lambda b.\ b$$

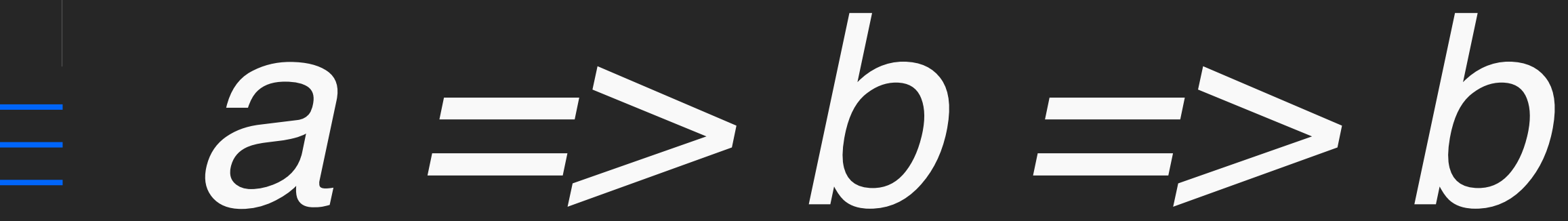
Lambdas can return other lambdas

This is a lambda

$$\lambda a.\ \lambda b.\ b$$

Currying

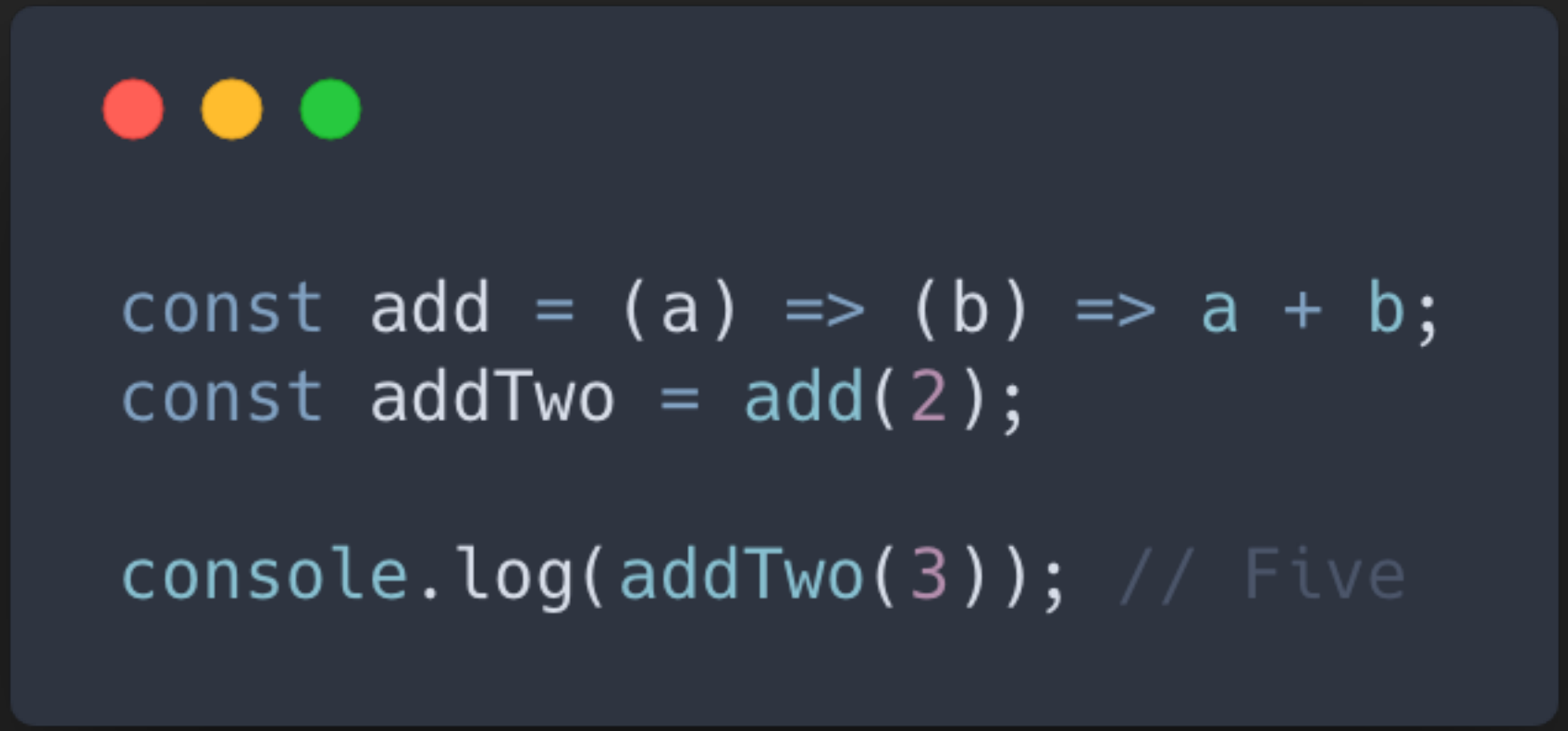
This is a lambda



In JavaScript...

RUINING JAVASCRIPT

This is a lambda



```
const add = (a) => (b) => a + b;
const addTwo = add(2);

console.log(addTwo(3)); // Five
```

This is a lambda

```
DivideThreeNumbers :: Int -> Int -> Int -> Int
```

In Haskell...

This is a lambda

$$(\lambda x.\ x\ b)\ [x \rightarrow y]$$

evaluation or substitution

This is a lambda

(y b)

evaluation or substitution

This is a lambda

$$(\lambda x.\ x\ b)\ [b \rightarrow y]$$

This is a lambda

$$(\lambda x.\ x\ b)\ [b \rightarrow y]$$

b is not an argument!

This is a lambda

Each evaluation step is called a beta-reduction

This is a lambda

We can only do beta reduction when expressions have beta-reduxes

This is a lambda

We can only do beta reduction when expressions contain an applicative form

This is a lambda

When an expression cannot be further evaluated it is said to be in its beta-normal form

This is a lambda

$$\lambda x.\ \lambda y.\ +\ y\ (+\ x\ 1)$$

This is a lambda

$$\lambda x.\ \lambda y.\ +\ y\ (+\ x\ 1)\ 4\ 2$$

This is a lambda

$$(\lambda x.\ \lambda y.\ +\ y\ (+\ x\ 1))\ 4\ 2$$

$$(\lambda x.\ \lambda y.\ +\ y\ (+\ 4\ 1))\ [x \rightarrow 4]$$

This is a lambda

$$(\lambda x.\ \lambda y.\ +\ y\ (+\ x\ 1))\ 4\ 2$$

$$(\lambda x.\ \lambda y.\ +\ y\ (+\ 4\ 1))\ [x \rightarrow 4]$$

$$(\lambda y.\ +\ y\ (+\ 4\ 1))$$

This is a lambda

$$(\lambda y.\ +\ y\ (+\ 4\ 1))\ 2$$

This is a lambda

$$(\lambda y.\ +\ y\ (+\ 4\ 1))\ 2$$

$$(\lambda y.\ +\ y\ (+\ 4\ 1))\ [y \rightarrow 2]$$

This is a lambda

$$(\lambda y.\ +\ y\ (+\ 4\ 1))\ 2$$

$$(\lambda y.\ +\ y\ (+\ 4\ 1))\ [y \rightarrow 2]$$

$$(+\ 2\ (+\ 4\ 1))\ [y \rightarrow 2]$$

This is a lambda

$$(\lambda y.\ +\ y\ (+\ 4\ 1))\ 2$$

$$(\lambda y.\ +\ y\ (+\ 4\ 1))\ [y \rightarrow 2]$$

$$(+\ 2\ (+\ 4\ 1))\ [y \rightarrow 2]$$

$$(+\ 2\ (+\ 4\ 1))$$

This is a lambda

```
(+ 2 (+ 4 1))
(+ 2 5)
(7)
```

This is a lambda

This is a lambda

$$x\ y\ z$$

How do we disambiguate?

This is a lambda

$$x(y\ z) \text{ or } (x\ y)\ z$$

?

This is a lambda

$$(x\ y)\ z$$

applications are left associative

This is a lambda

fn(first)(second)

just like when you write proper functional code

This is a lambda

$$\lambda x. x\ y$$

How do we disambiguate?

This is a lambda

$$(\lambda x.\ x)\ y \quad \text{or} \quad (\lambda x.\ x\ y)$$

?

This is a lambda

$$(\lambda x. x\ y)$$

abstractions extend as much to the far-right as possible

This is a lambda

this is why you might need parenthesis to disambiguate

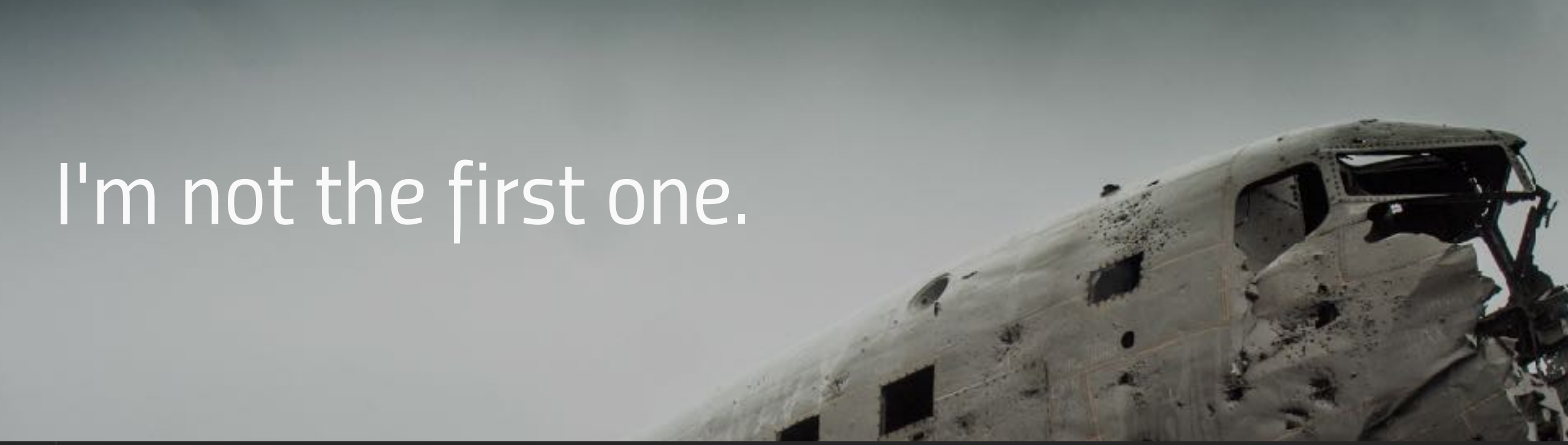
Lambda Calculus is another way of representing computation

Lambda Calculus is Turing Complete

We can replace any programming language with Lambda Calculus

In this talk I'm gonna write JavaScript with nothing but functions

I'm not the first one.

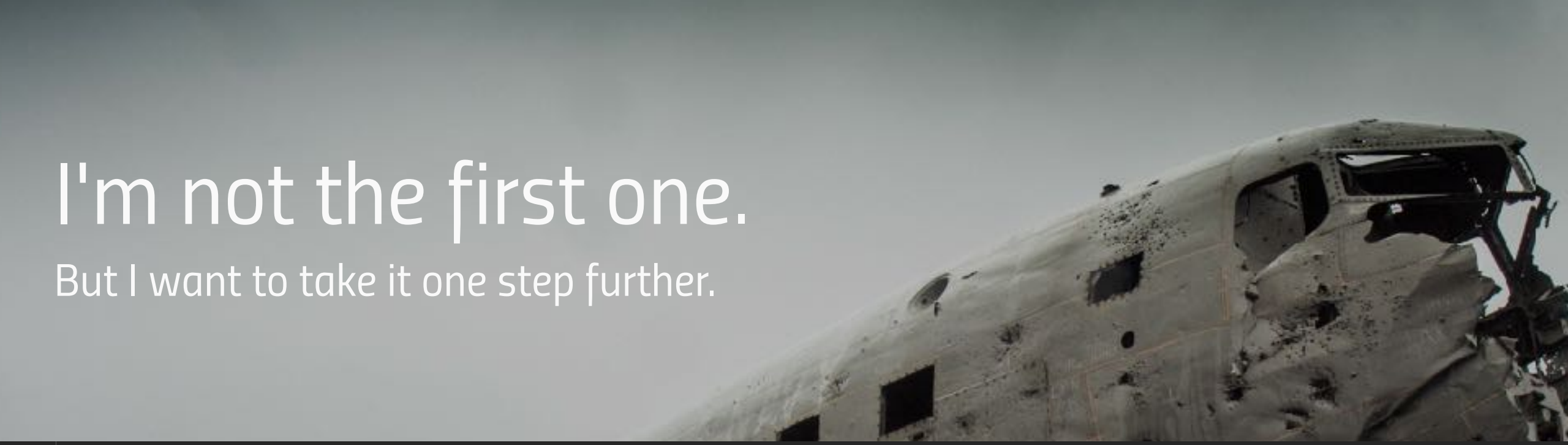


Programming with nothing

By Tom Stuart

I'm not the first one.

But I want to take it one step further.



Programming with nothing

By Tom Stuart

Part 1

Part 1

Part 2

Part 1

Part 2

Part 3

Part 1

Part 2

Part 3

Part 4

Rewrite JavaScript

With nothing but functions

- All functions must take only one argument
- Functions can return other functions
- We will use assignments to make things easier to explain


Ruining JavaScript
Part 1

Part 1

Replacing Booleans

Part 2

Replacing Numbers

What is a number, anyway?

What is a number, anyway?

2

What is a number, anyway?

2

What is a number, anyway?

Representation

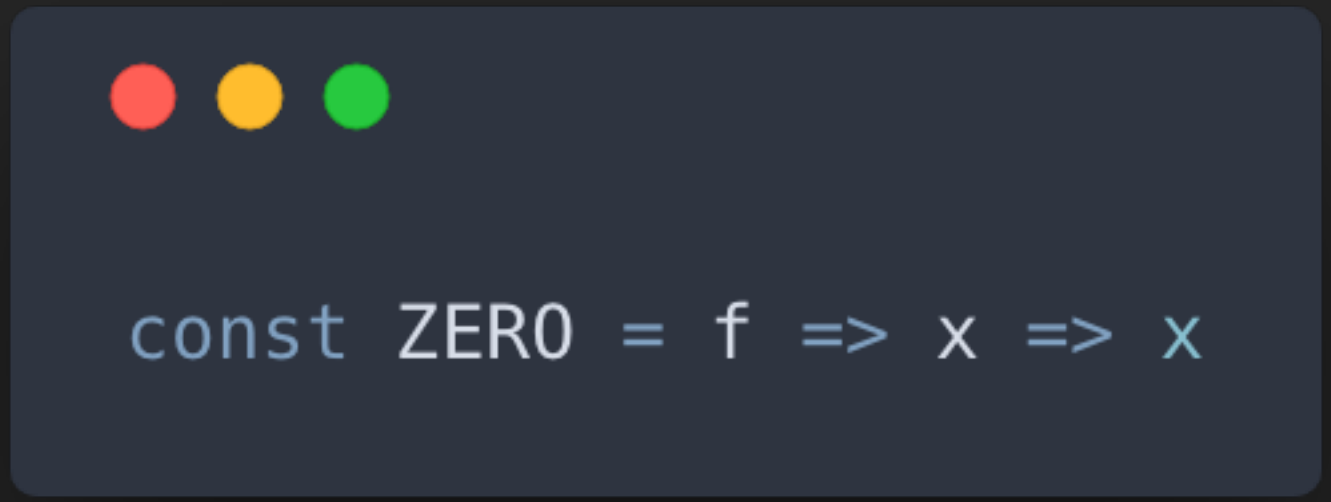
2

Meaning

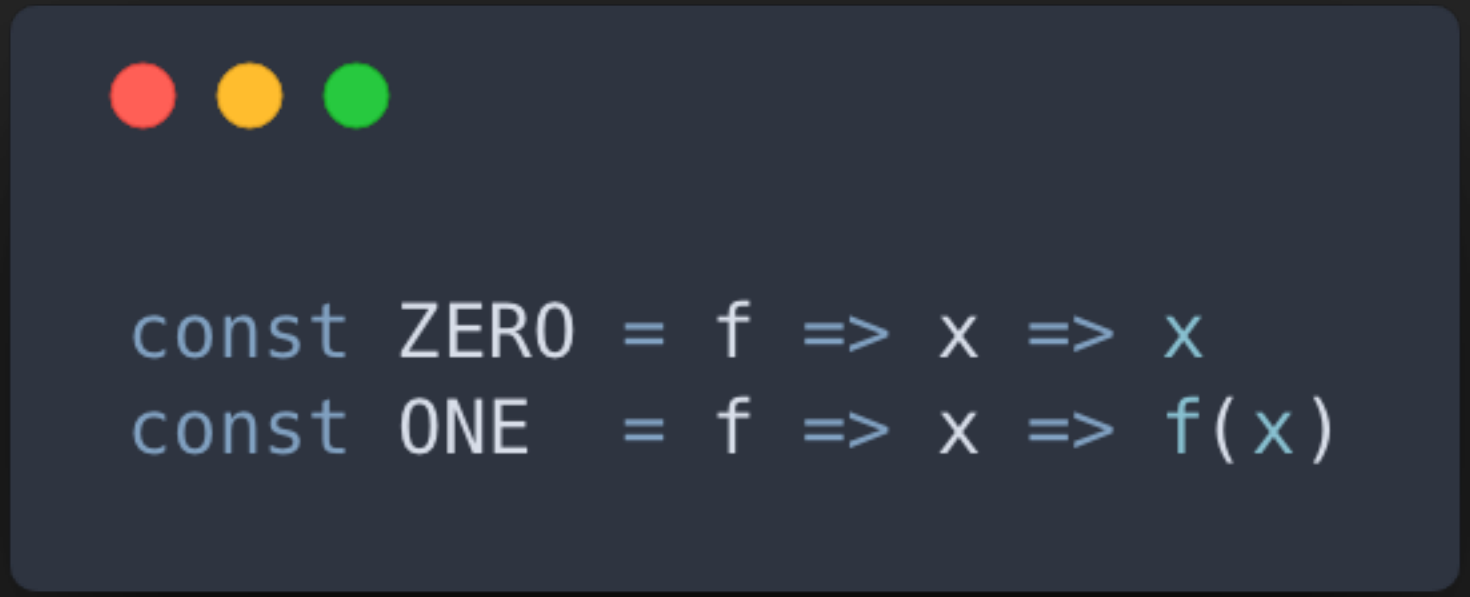
Meaning

How can we represent quantities with functions?

How can we represent quantities with functions? Function applications!



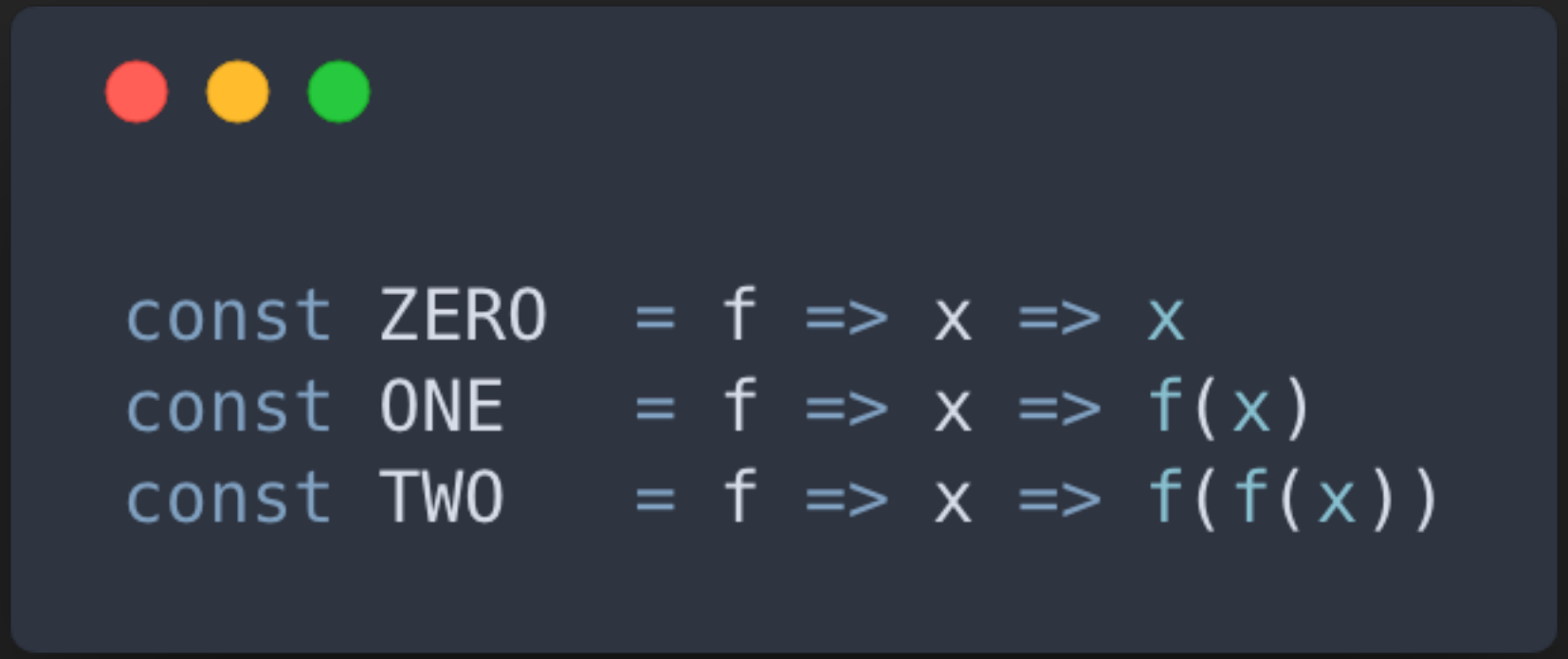
How can we represent quantities with functions? Function applications!



```
const ZERO = f => x => x
const ONE  = f => x => f(x)
```

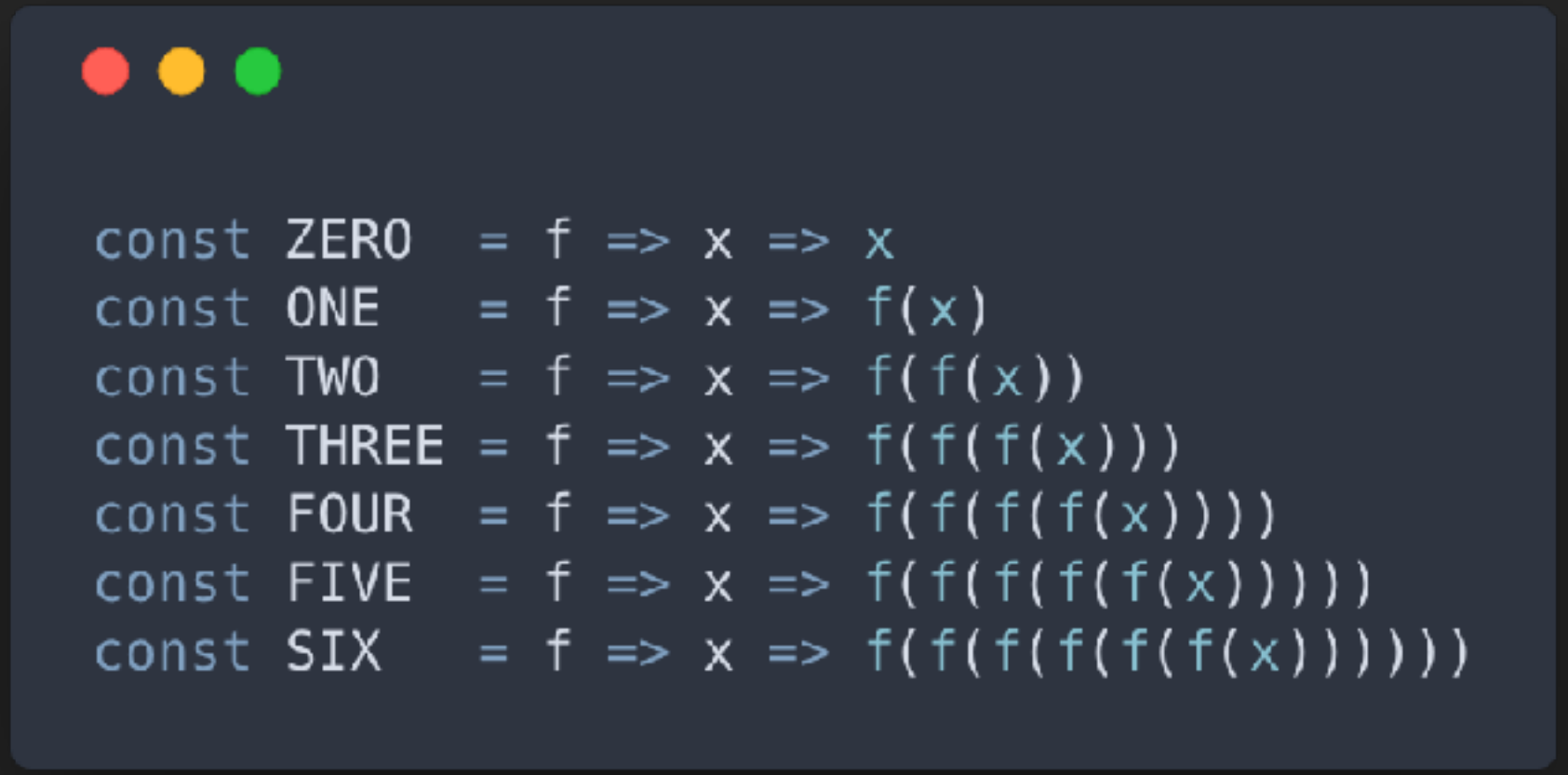
How can we represent quantities with functions?

Function applications!



```
const ZERO  = f => x => x
const ONE   = f => x => f(x)
const TWO   = f => x => f(f(x))
```

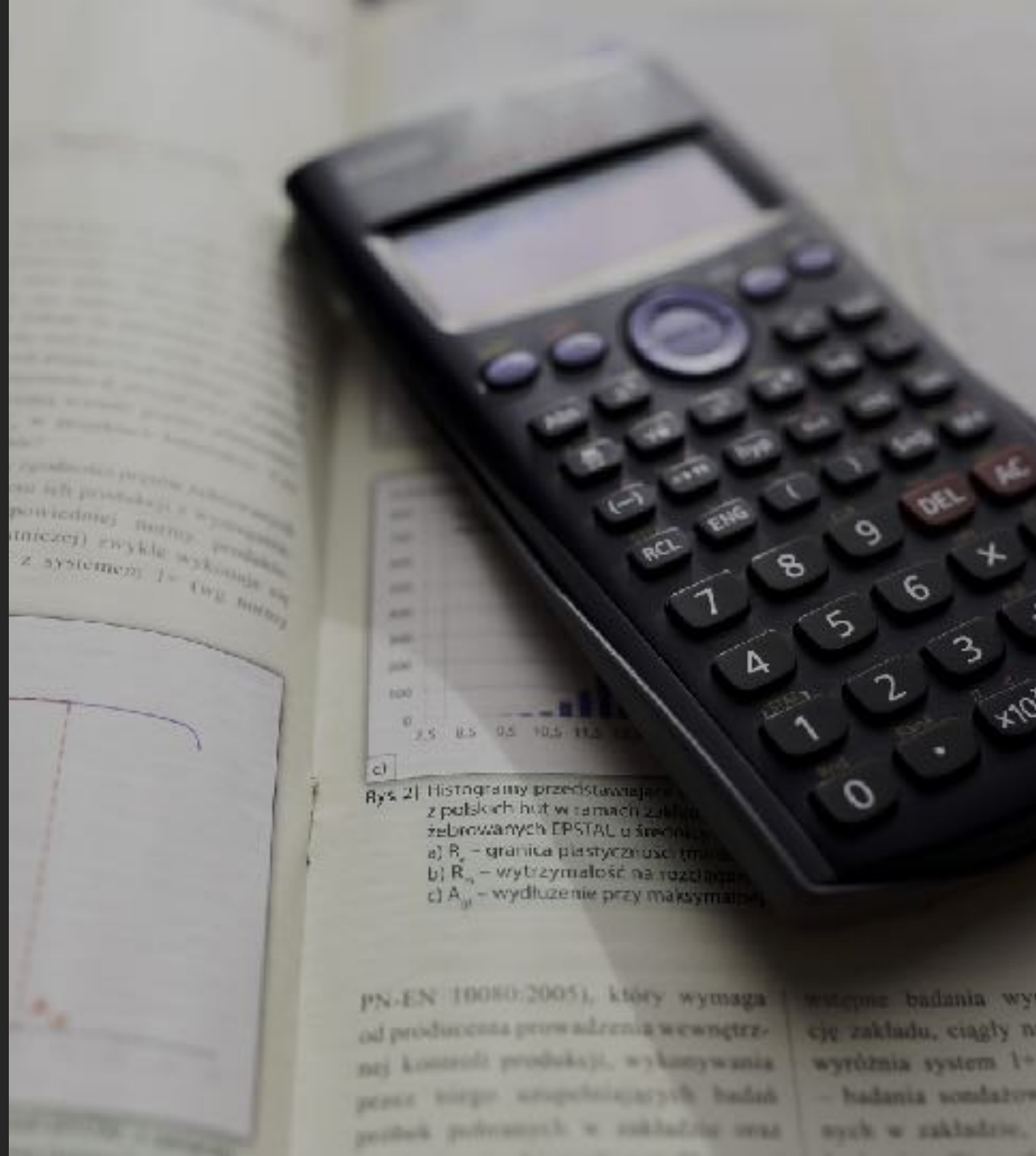
How can we represent quantities with functions? Function applications!



```
const ZERO  = f => x => x
const ONE   = f => x => f(x)
const TWO   = f => x => f(f(x))
const THREE = f => x => f(f(f(x)))
const FOUR  = f => x => f(f(f(f(x))))
const FIVE  = f => x => f(f(f(f(f(x)))))
const SIX   = f => x => f(f(f(f(f(f(x))))))
```

Part 3

Replacing Arithmetics




WRAPPERS

PAIRS

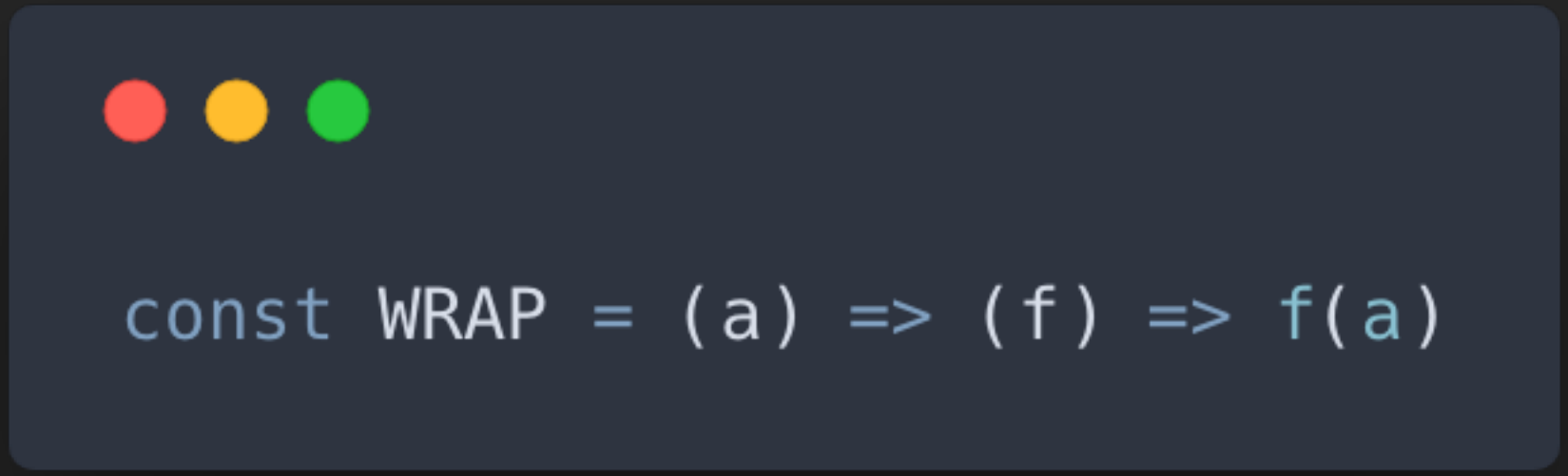
INCREMENTING PAIRS

PREDECESSOR


WRAPPERS

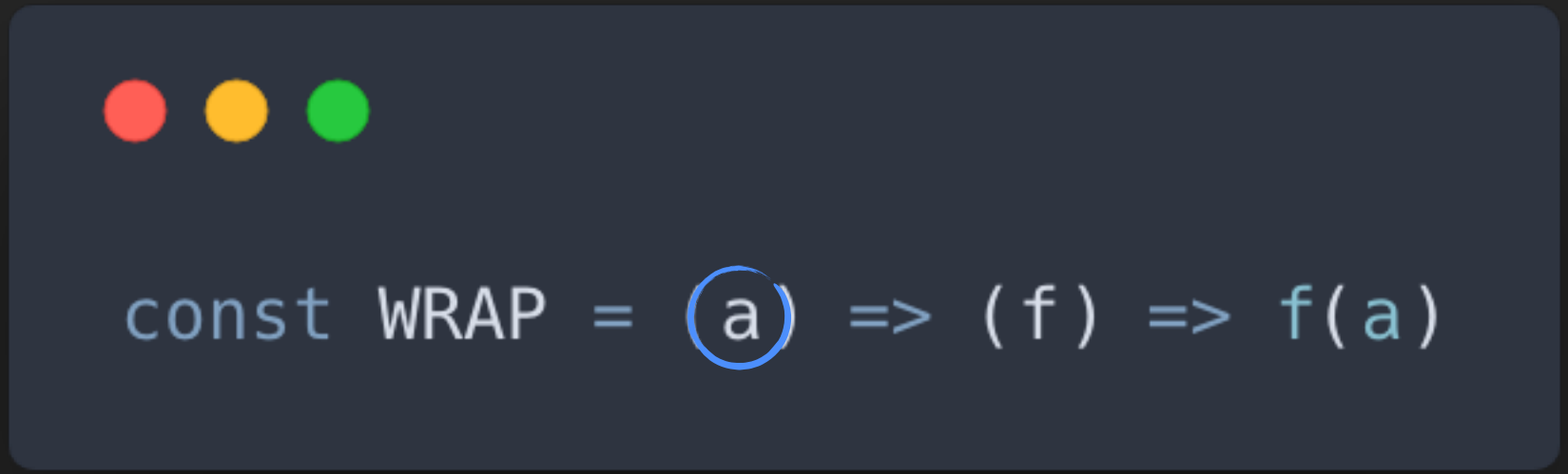
Replacing arithmetics

Pairs



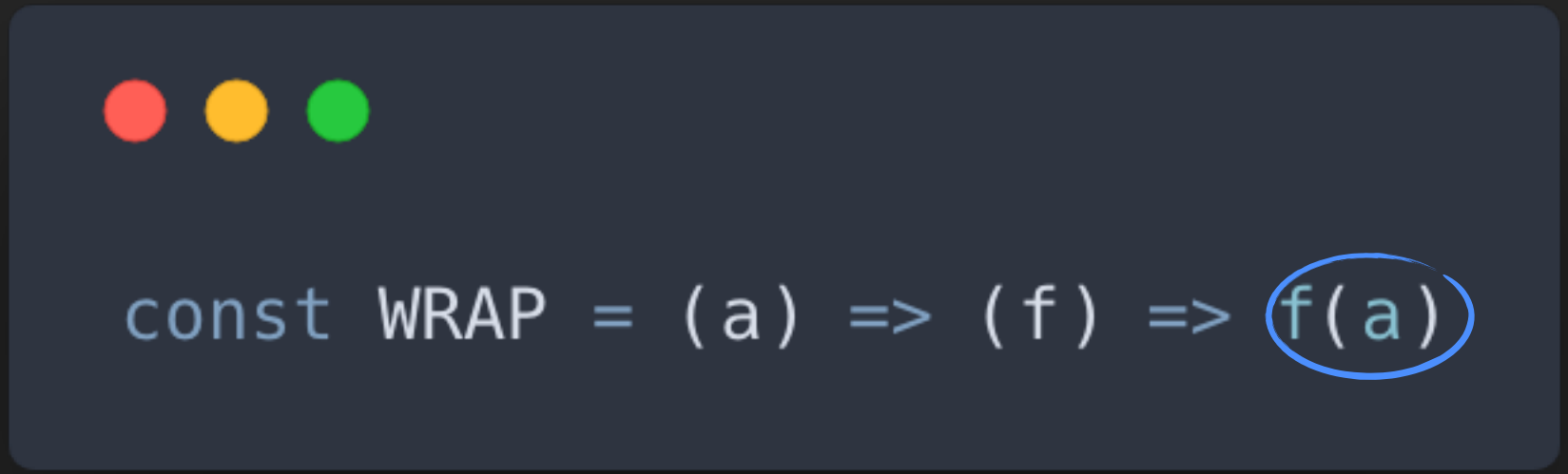
```
const WRAP = (a) => (f) => f(a)
```

Pairs



First we store something.

Pairs

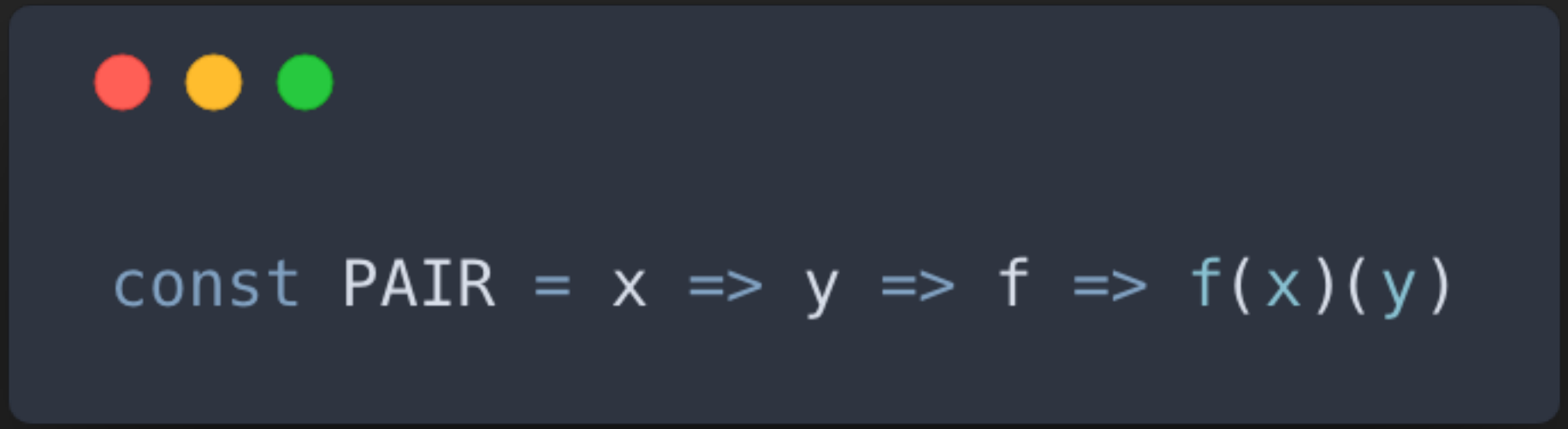


Then we apply whatever is stored to a function

PAIRS

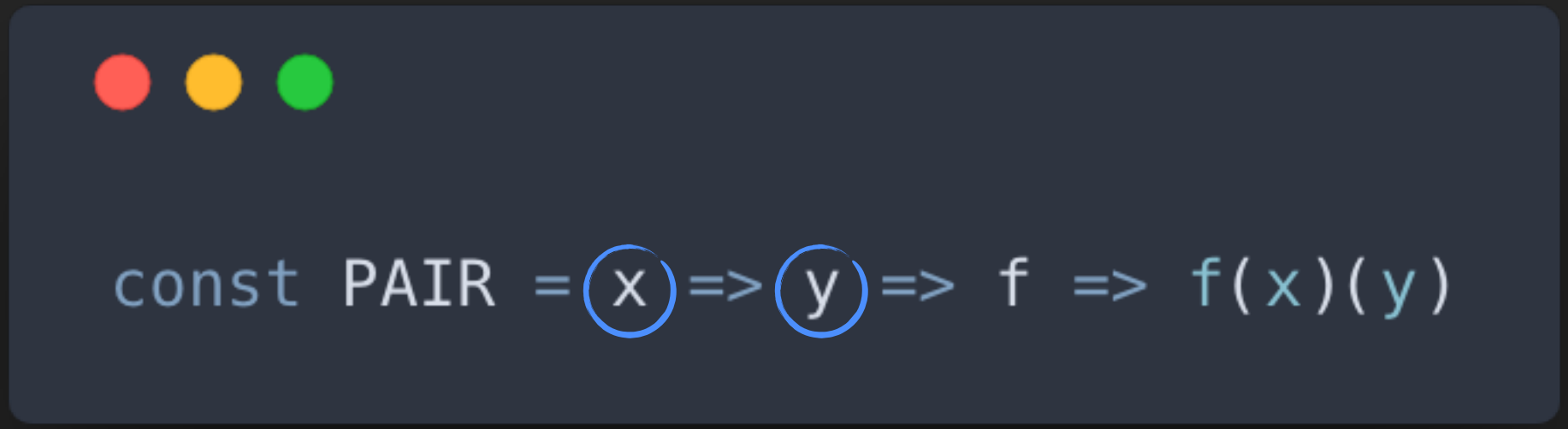
Replacing arithmetics

Pairs



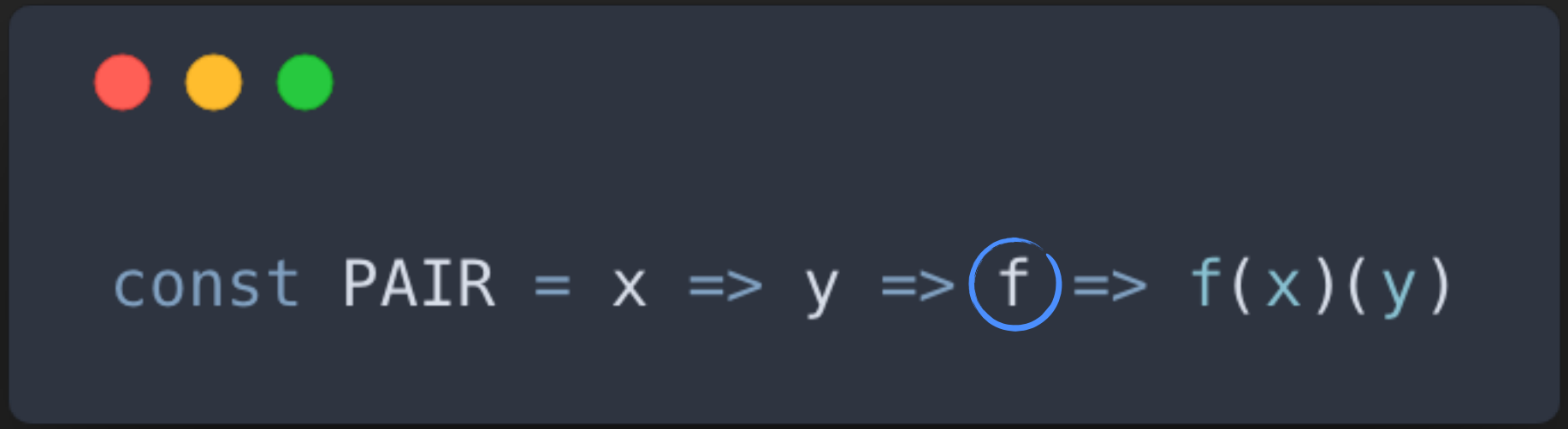
```
const PAIR = x => y => f => f(x)(y)
```

Pairs



First we store two values.

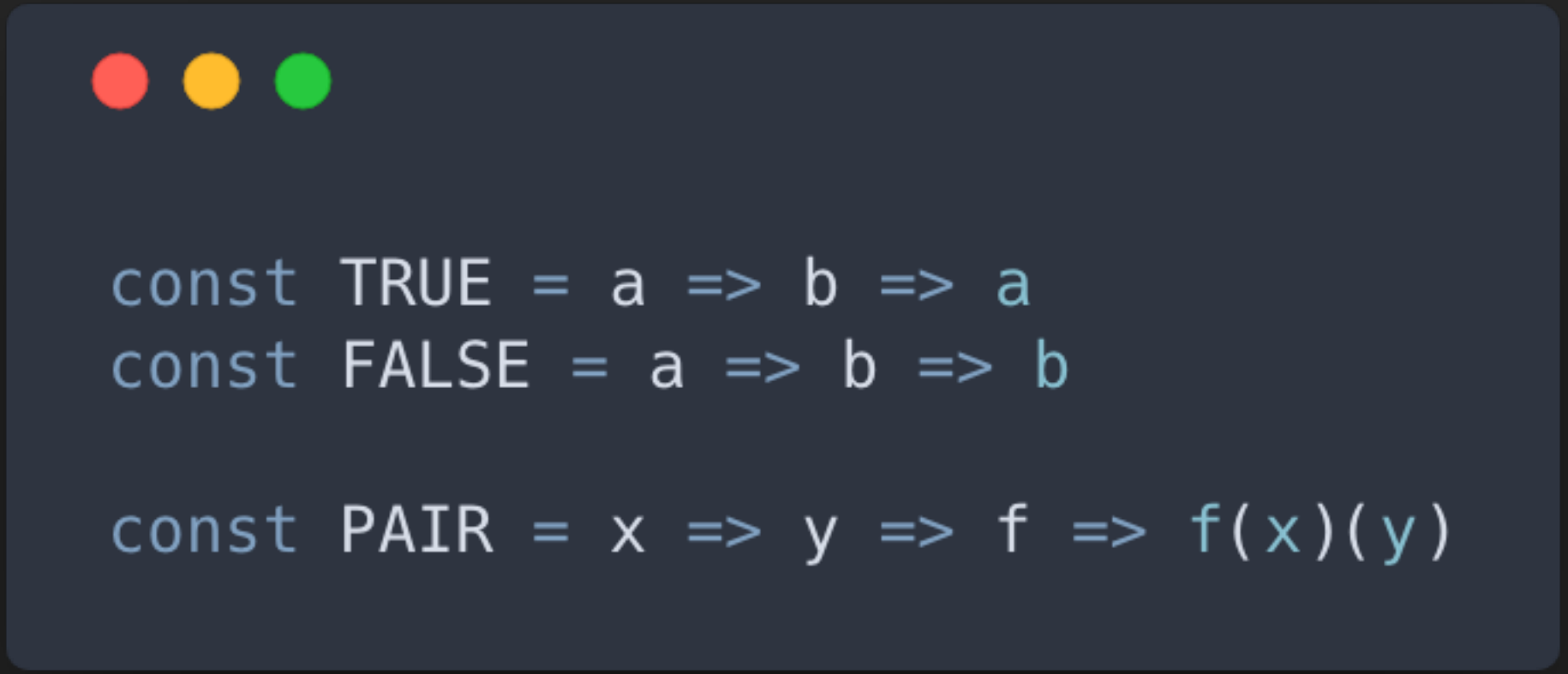
Pairs



Then we tell which one we want.

Replacing arithmetics

Pairs

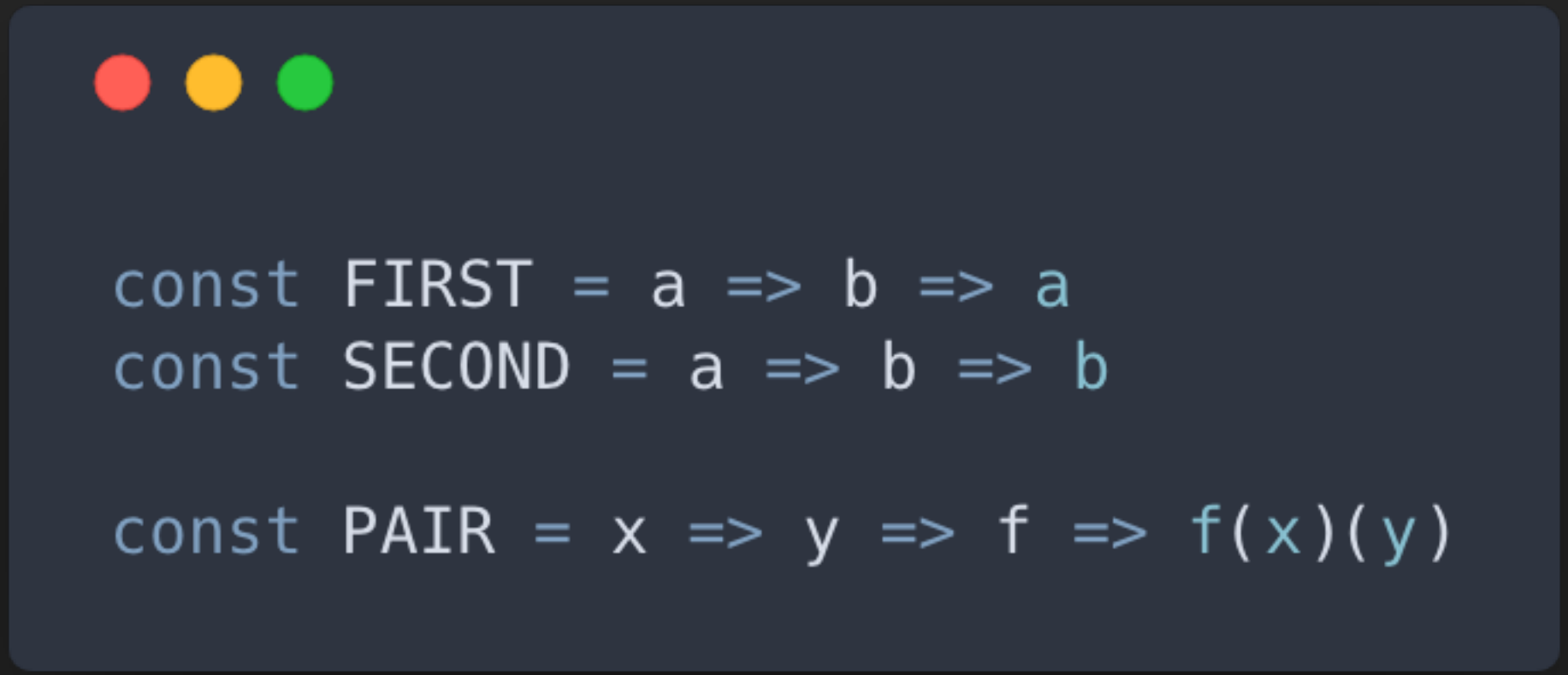


```
const TRUE = a => b => a
const FALSE = a => b => b

const PAIR = x => y => f => f(x)(y)
```

Replacing arithmetics

Pairs



```
const FIRST = a => b => a
const SECOND = a => b => b

const PAIR = x => y => f => f(x)(y)
```

INCREMENTING PAIRS

Incrementing Pairs

$$\{0, 0\} \qquad \{0, 1\}$$

Incrementing Pairs

$$\{0, 0\} \quad \{0, 1\}$$
$$\{1, 2\}$$

Incrementing Pairs

$$\{0, 0\} \quad \{0, 1\}$$
$$\{1, 2\} \quad \{2, 3\}$$

Incrementing Pairs

$$\{0, 0\} \quad \{0, 1\}$$
$$\{1, 2\} \quad \{2, 3\}$$
$$\{4, 5\}$$

Incrementing Pairs

{ 0, 0 } { 0, 1 }
{ 1, 2 } { 2, 3 }
{ 4, 5 } { 5, 6 }

Incrementing Pairs

$$\{0, 0\} \quad \{0, 1\}$$
$$\{1, 2\} \quad \{2, 3\}$$
$$\{4, 5\} \quad \{5, 6\}$$

Five $\{0, 0\}$

Incrementing Pairs

{ 0, 0 } { 0, 1 }
{ 1, 2 } { 2, 3 }
{ 4, 5 } { 5, 6 }

Five { 0, 0 } { 0, 1 } x1

Incrementing Pairs

$$\{0, 0\} \quad \{0, 1\}$$
$$\{1, 2\} \quad \{2, 3\}$$
$$\{4, 5\} \quad \{5, 6\}$$

Five $\{0, 0\} \quad \{1, 2\}$ x2

Incrementing Pairs

{ 0, 0 } { 0, 1 }

{ 1, 2 } { 2, 3 }

{ 4, 5 } { 5, 6 }

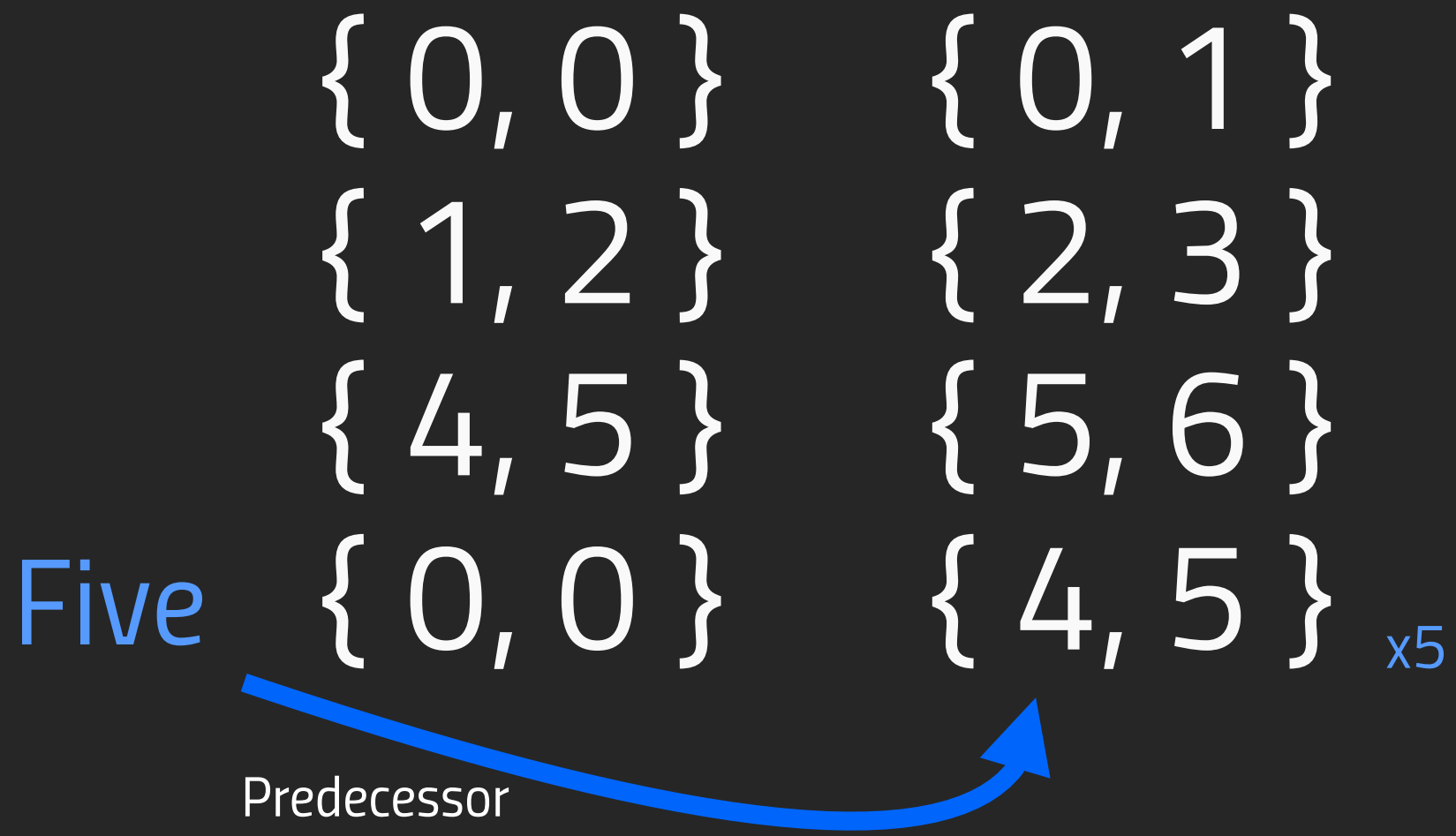
Five { 0, 0 } { 2, 3 } x3

Incrementing Pairs

{ 0, 0 } { 0, 1 }
{ 1, 2 } { 2, 3 }
{ 4, 5 } { 5, 6 }

Five { 0, 0 } { 3, 4 } x4

Incrementing Pairs



PREDECESSOR

Replacing arithmetics

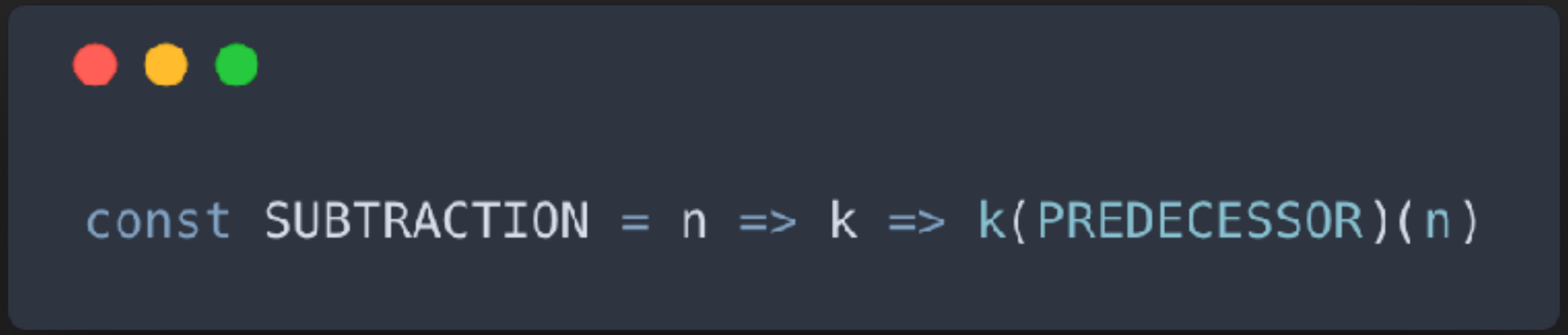
The Predecessor Function

```
const PREDECESSOR = n => n(INCREMENT_PAIR)(PAIR(ZERO)(ZERO))(TRUE)
```

First of N x INCREMENT_PAIR(0, 0)

Replacing arithmetics

The Subtraction Function



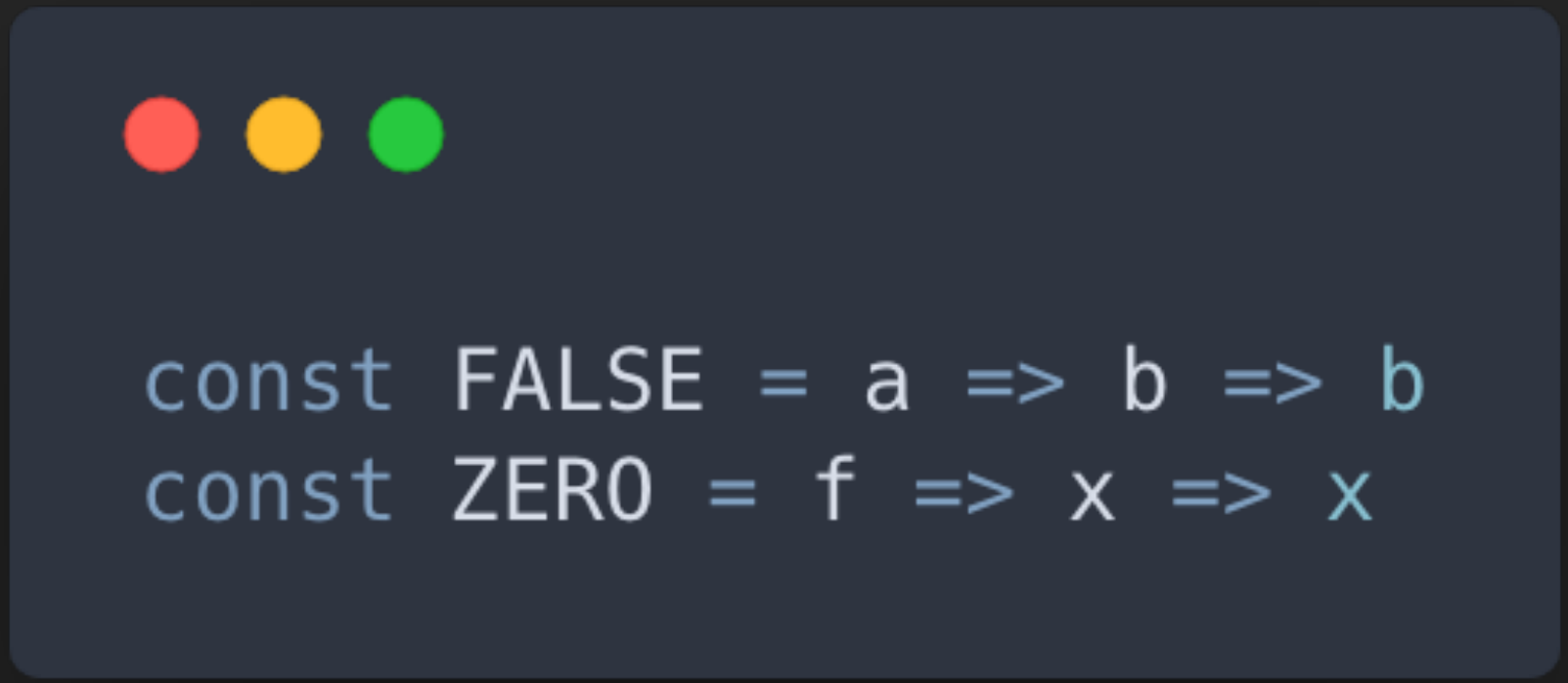
```
const SUBTRACTION = n => k => k(PREDECESSOR)(n)
```

Predecessor of N K times

Part 4

Replacing Boolean Operators

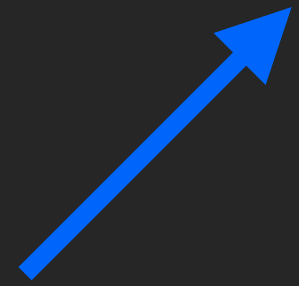
Alpha Equivalence



```
const FALSE = a => b => b
const ZERO = f => x => x
```

Renaming

```
λx. λy. x(y(z)) {a/x}
```



Renaming "x" to "a"

Renaming

```
λa. λy. a(y(z))
```

Renaming

```
λa. λy. a(y(z)) {b/y}
```

Renaming "y" to "b"

Renaming

```
λa. λb. a(b(z))
```

Renaming

```
λa. λy. a(y(z)) {c/z}
```

Renaming "z" to "c"

Renaming

```
λa. λy. a(y(z))
```

Same thing! Z is not an argument!

Ruining JavaScript With Birds

Part 2

Our Birds

Combinators

Functions that don't have free variables.

```
const withContext = a => b(a)
```

a is bound and b is free

```
const combinator  = a => b => a
```

both a and b are bound

Our Birds

Combinators

```
const one = x => x - x
```

Our Birds

Combinators

```
const two = x => x + y
```

Our Birds

Combinators

```
const three = x => y + z
```

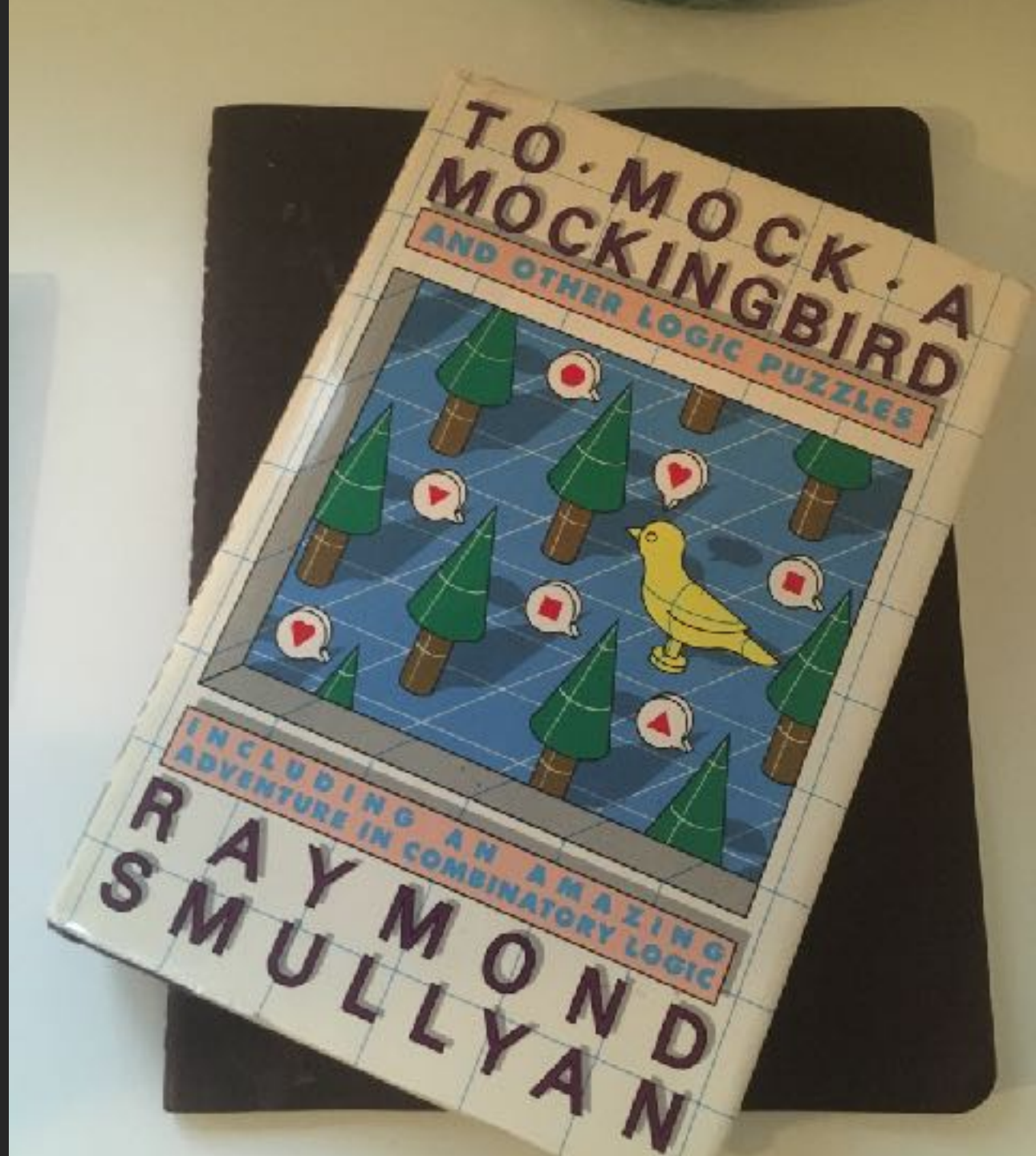
Our Birds

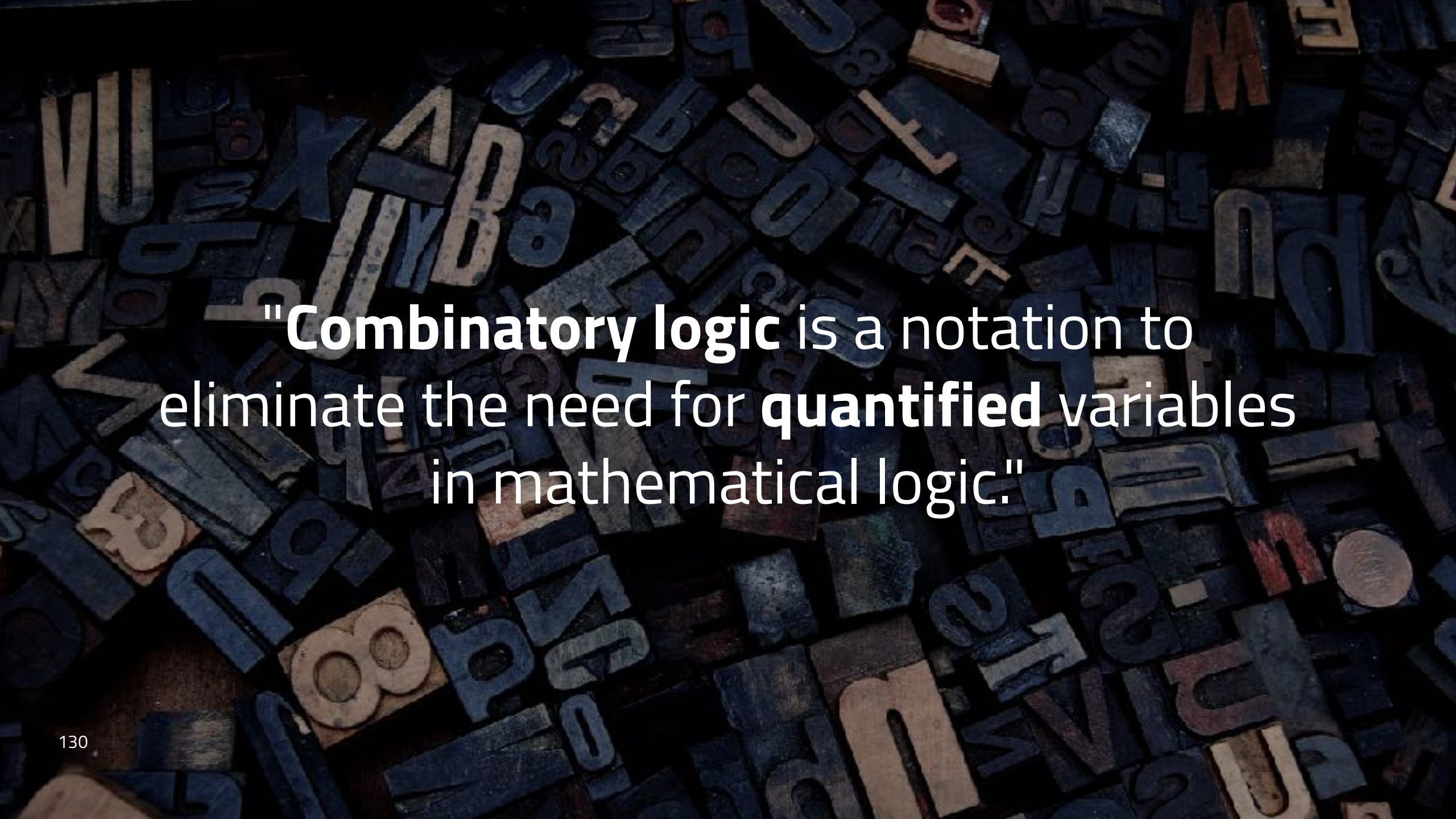
Combinators

```
const four = (x) => (y) => x + y
```

Combinators and Birds

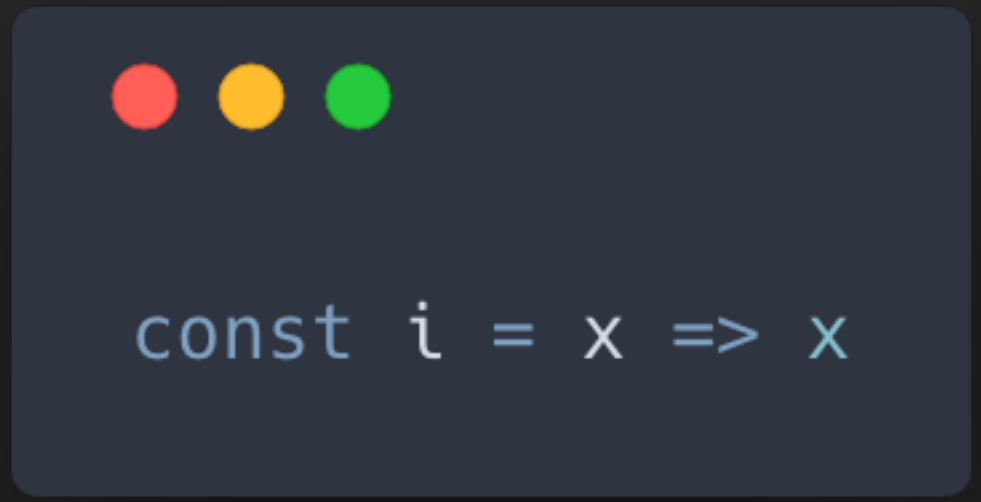
To Mock a Mockingbird

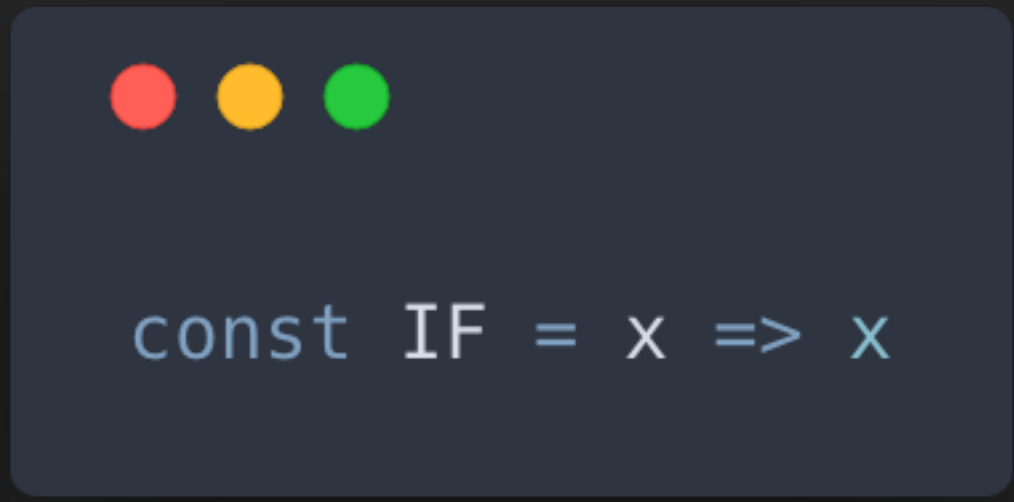



"**Combinatory logic** is a notation to eliminate the need for **quantified** variables in mathematical logic."

Combinators

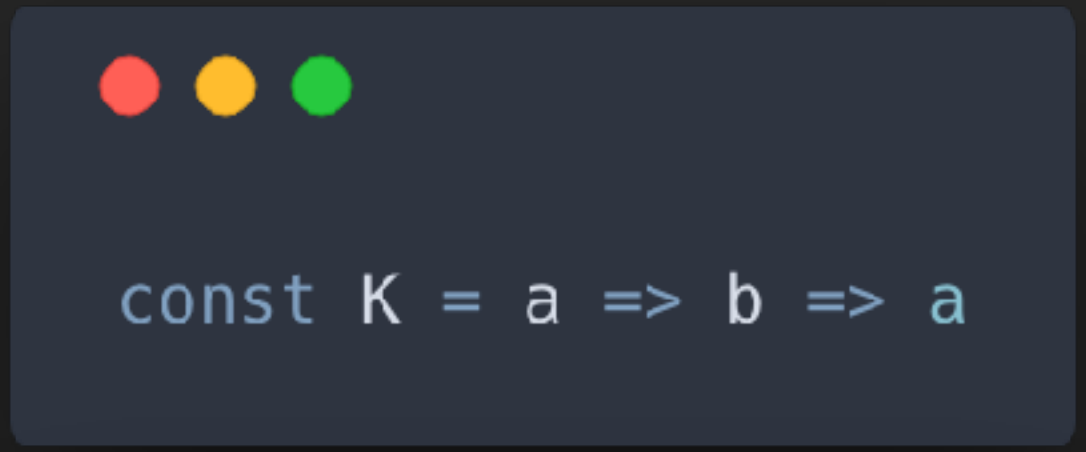
The Idiot Bird

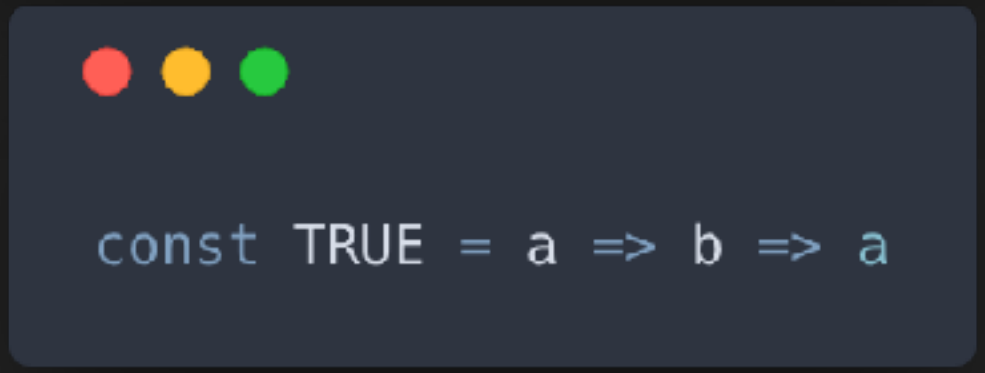




Combinators

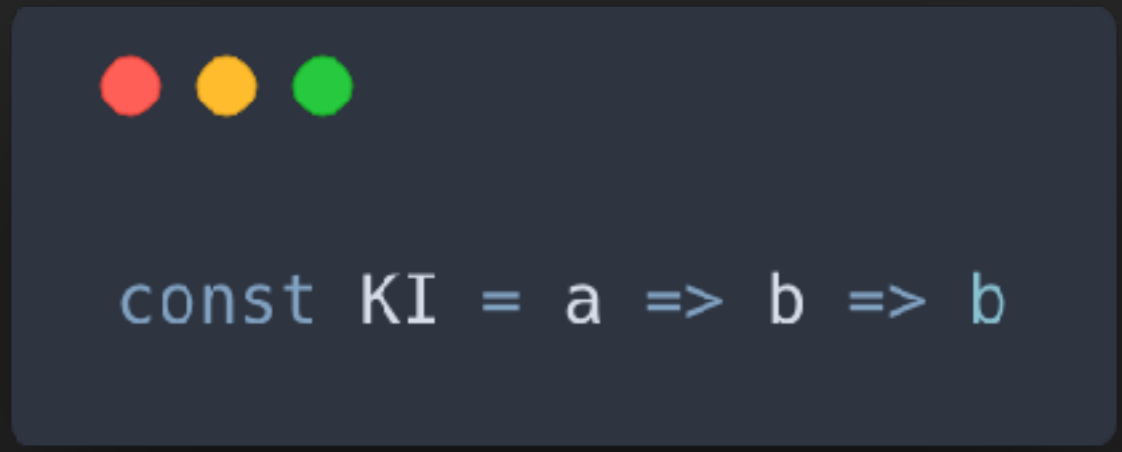
The Kestrel

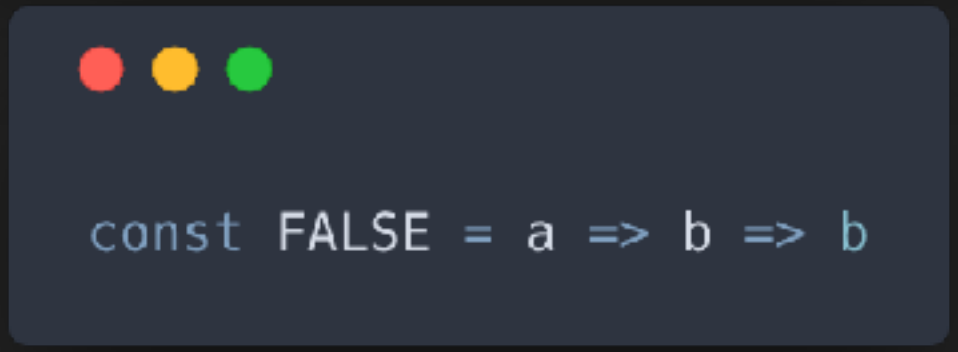




Combinators

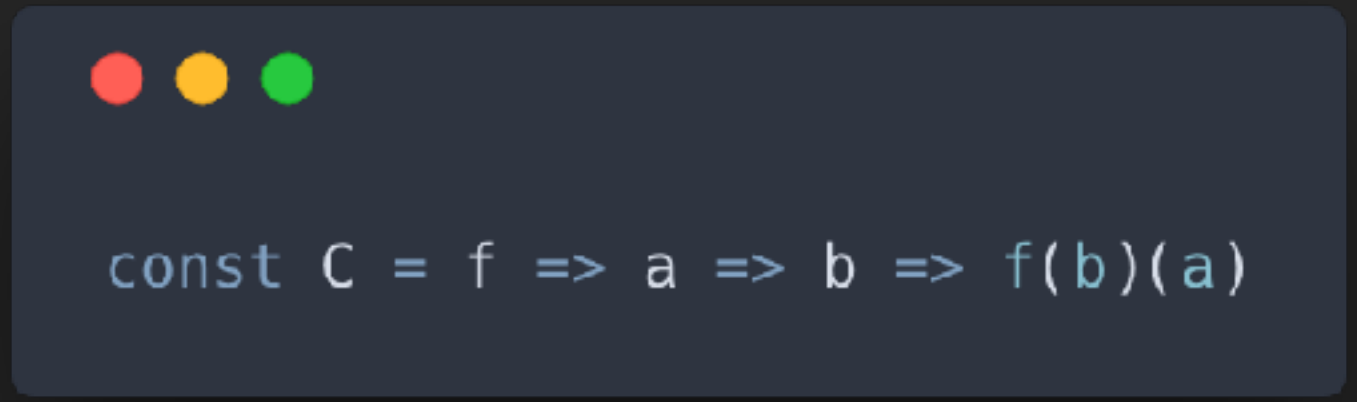
The Kite



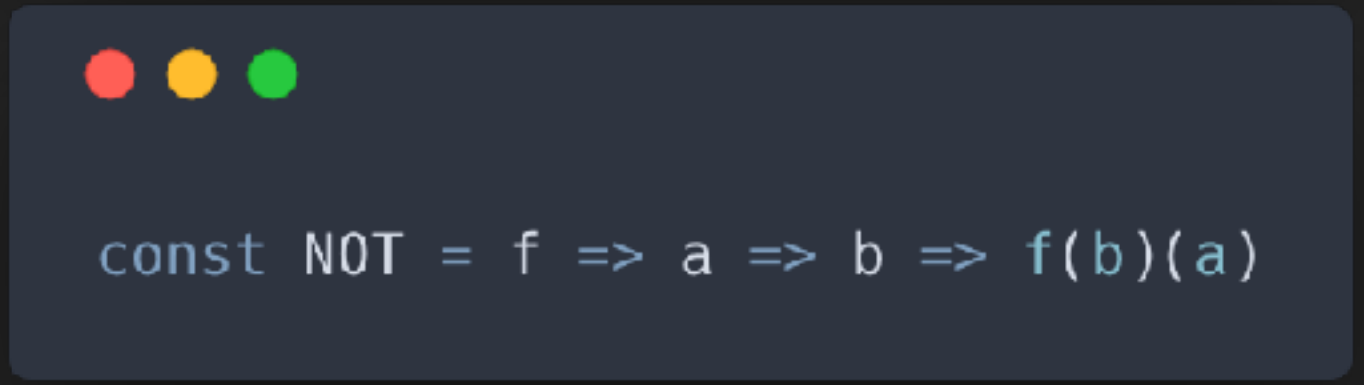


Combinators

The Cardinal



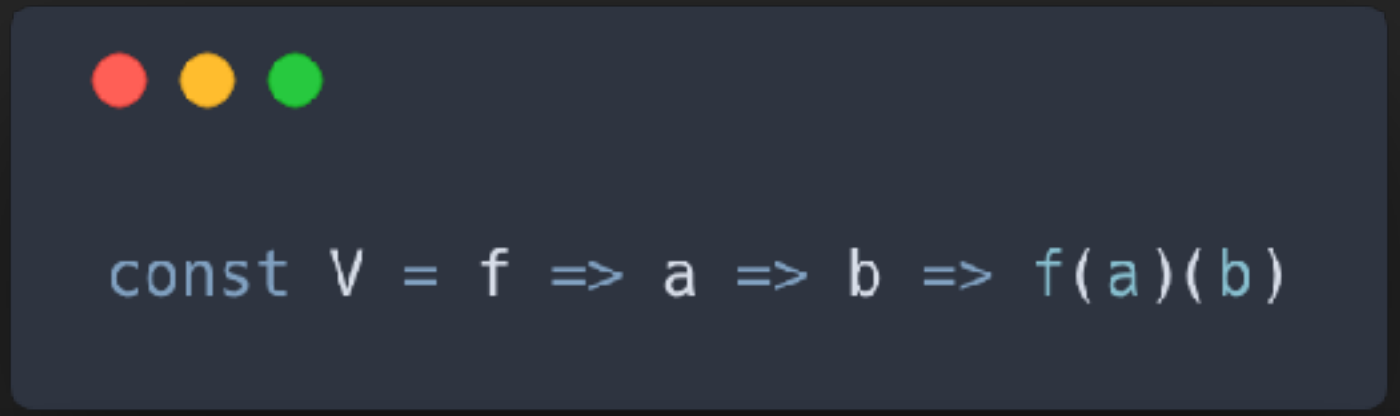
```
const C = f => a => b => f(b)(a)
```



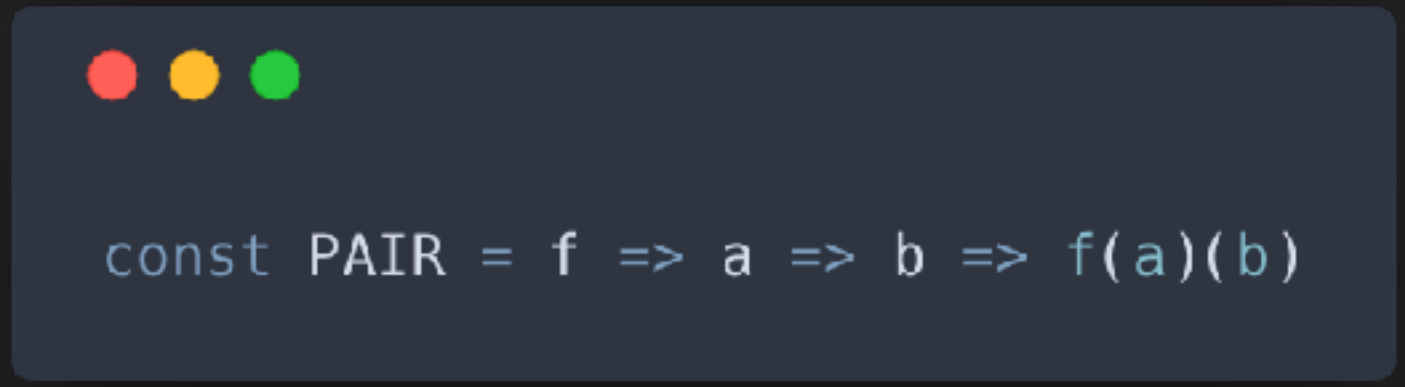
```
const NOT = f => a => b => f(b)(a)
```

Combinators

The Vireo



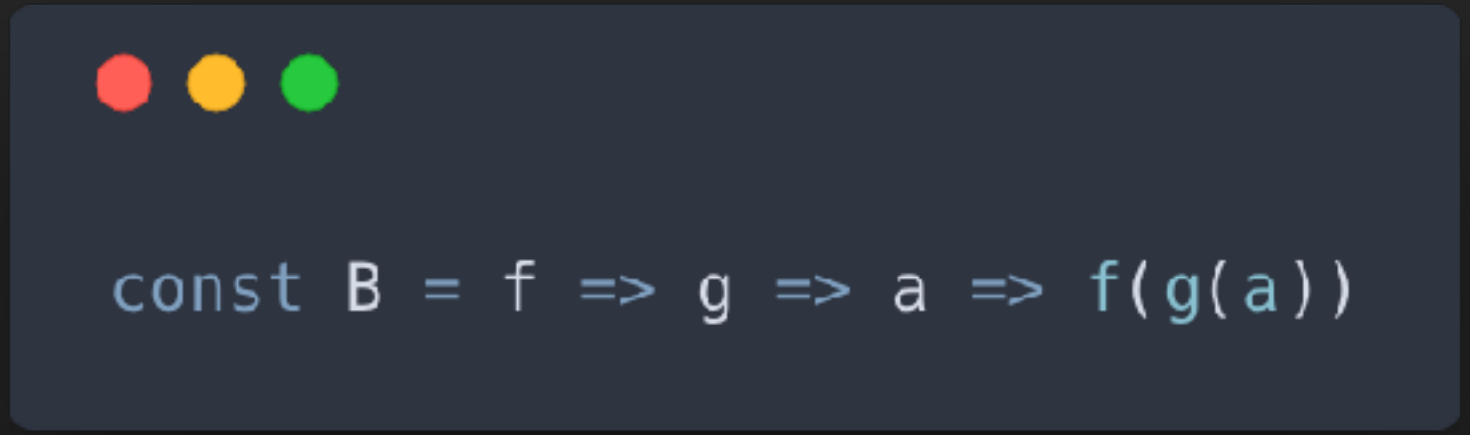
```
const V = f => a => b => f(a)(b)
```



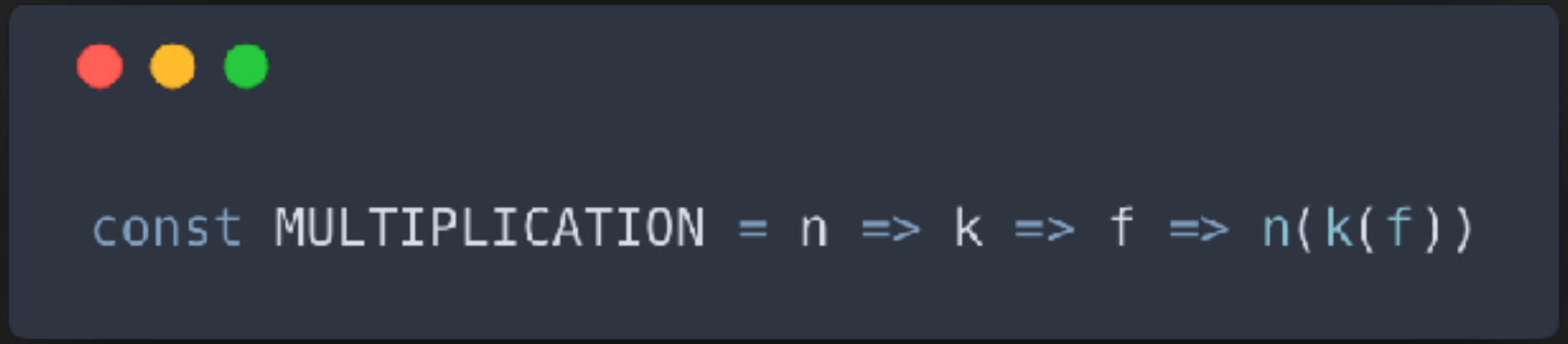
```
const PAIR = f => a => b => f(a)(b)
```

Combinators

The Bluebird



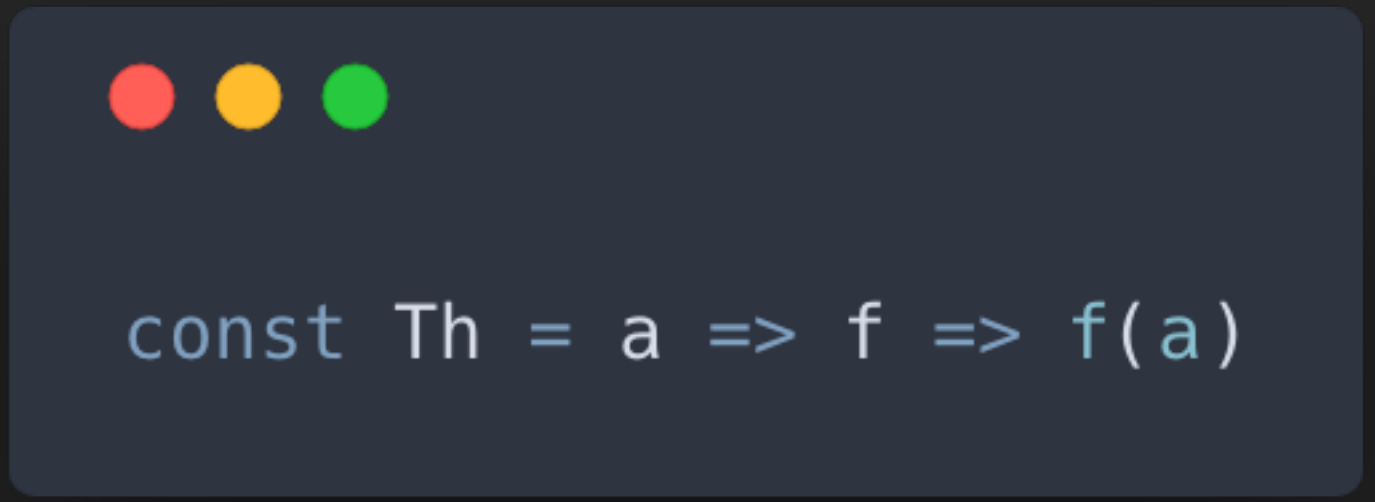
```
const B = f => g => a => f(g(a))
```



```
const MULTIPLICATION = n => k => f => n(k(f))
```

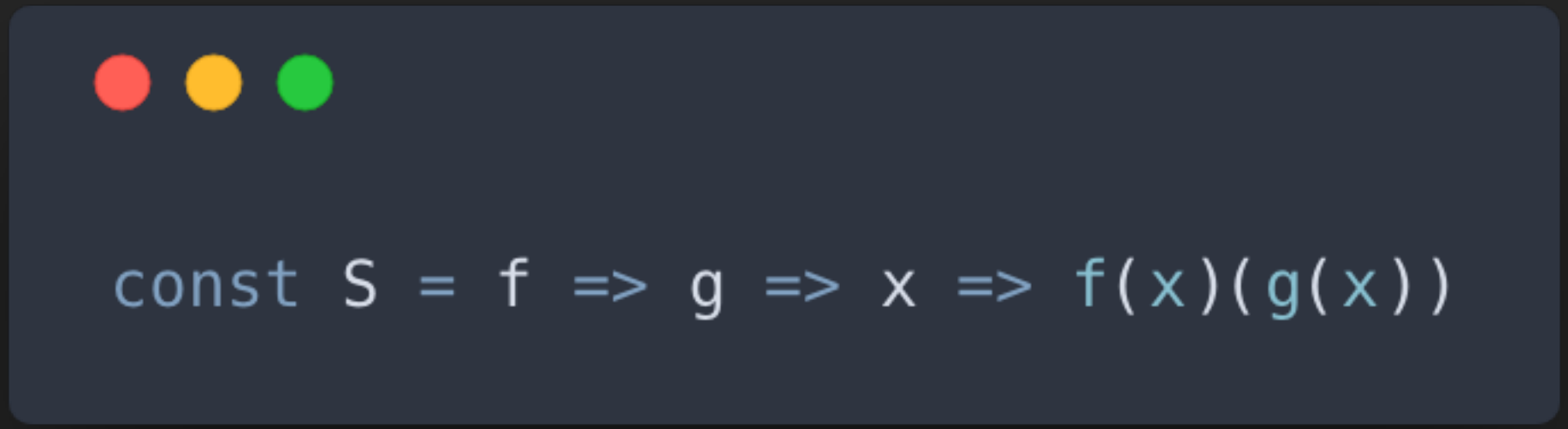
Combinators

The Thrush



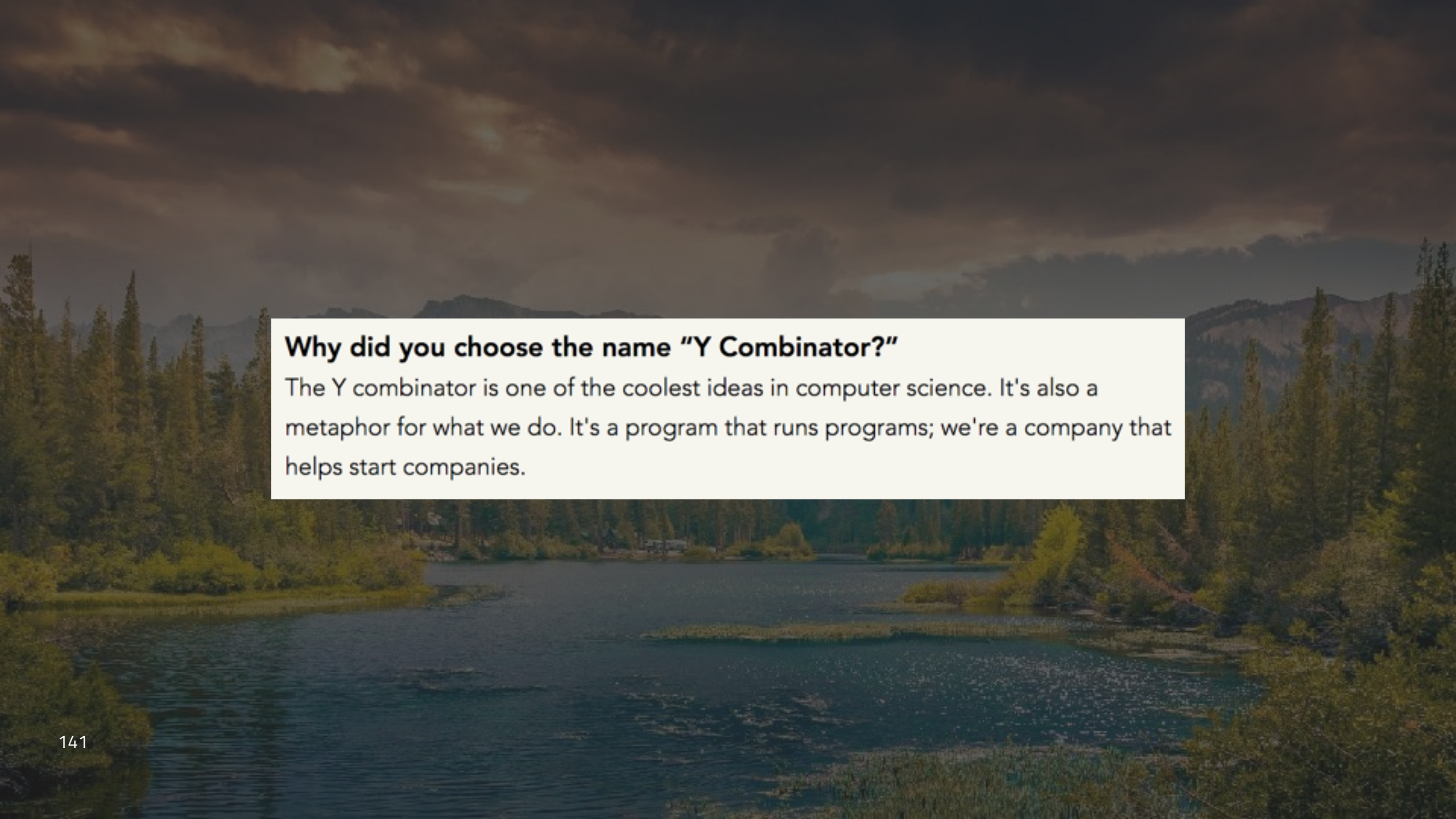
Combinators

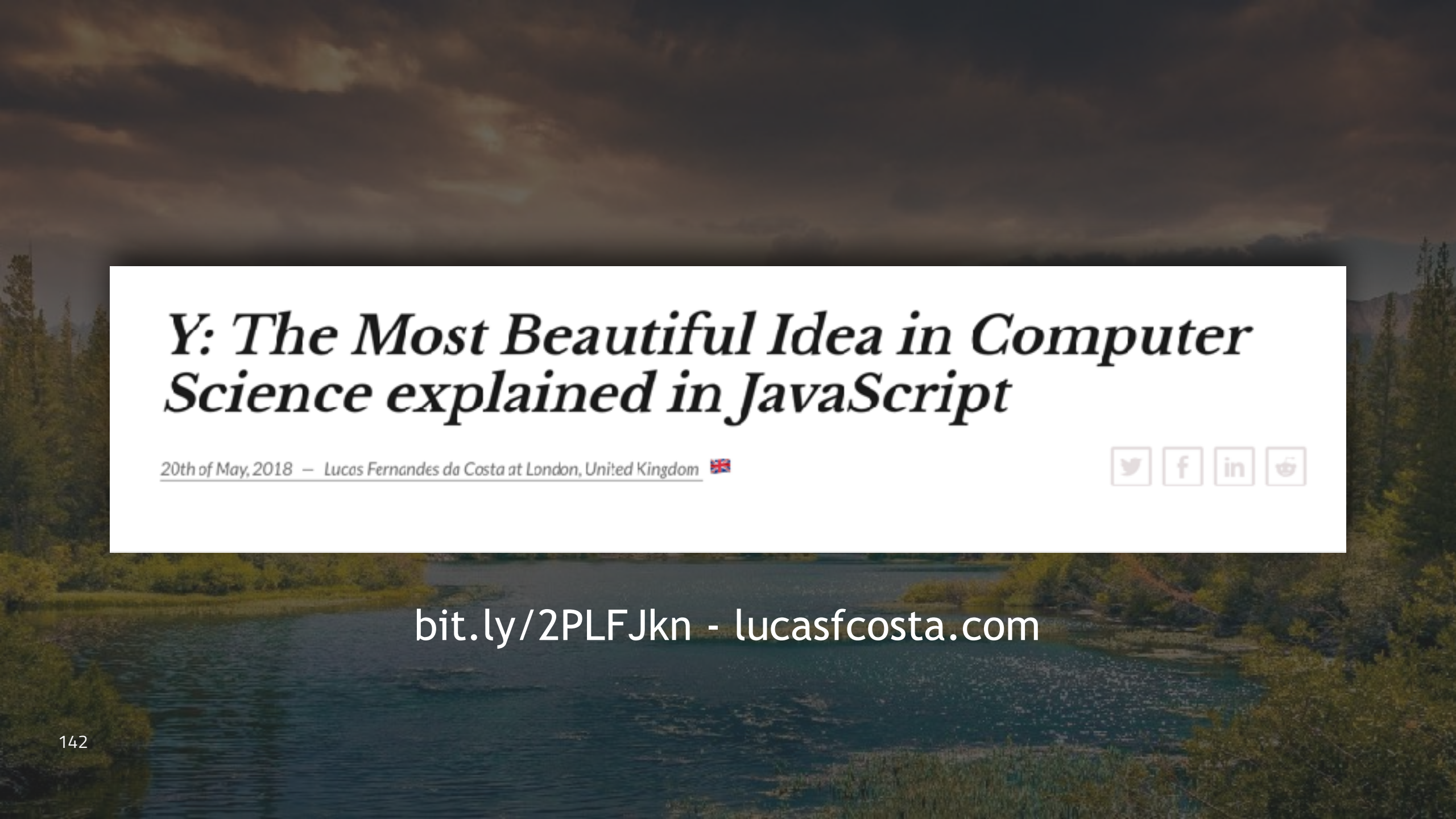
The Starling




One last combinator

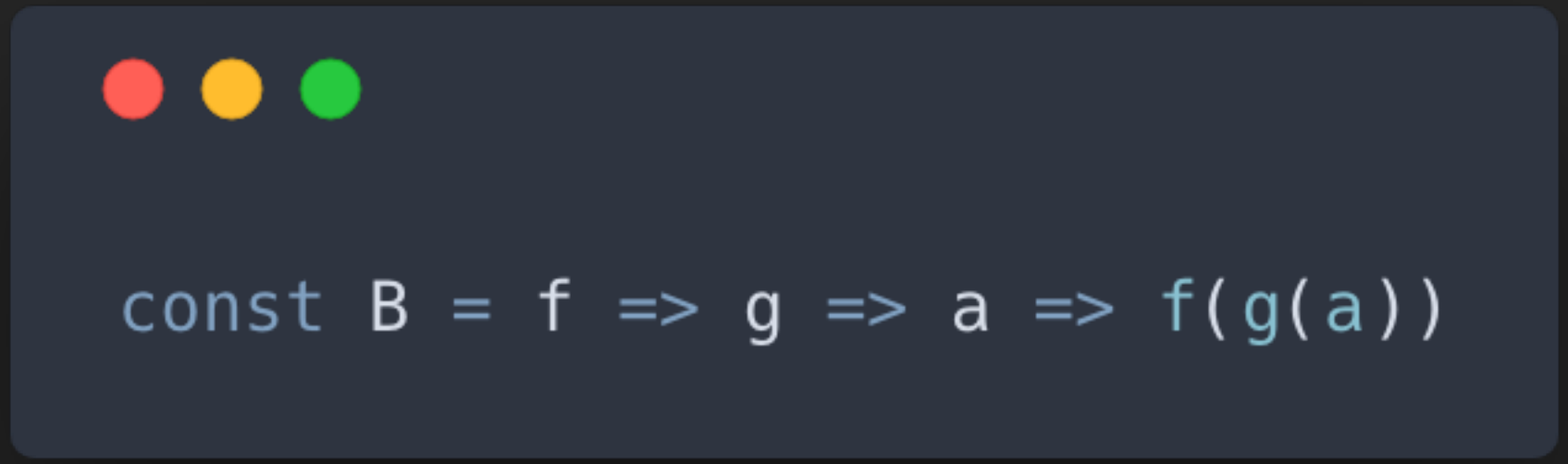

Y
Combinator


Why did you choose the name “Y Combinator?”
The Y combinator is one of the coolest ideas in computer science. It's also a metaphor for what we do. It's a program that runs programs; we're a company that helps start companies.



bit.ly/2PLFJkn - lucasfcosta.com

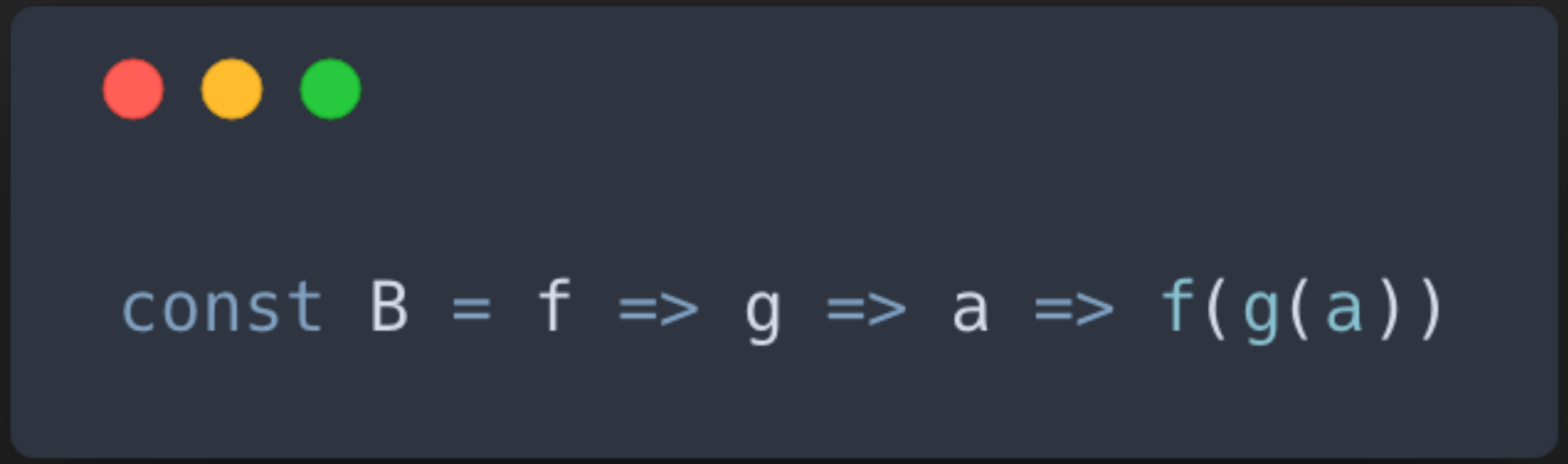
Replacing Functions (Successor)



```
const B = f => g => a => f(g(a))
```

```
const SUCCESSOR = n => f => x => f(n(f)(x))
```

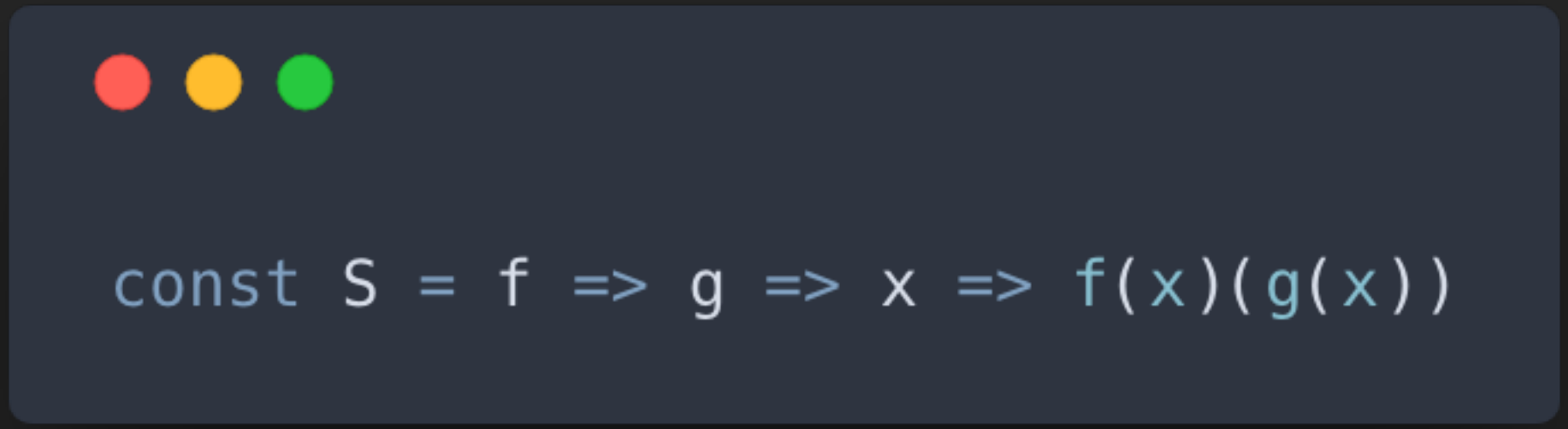
Replacing Functions (Successor)



```
const B = f => g => a => f(g(a))
```

```
const SUCCESSOR = n => f => B(f)(n(f))
```

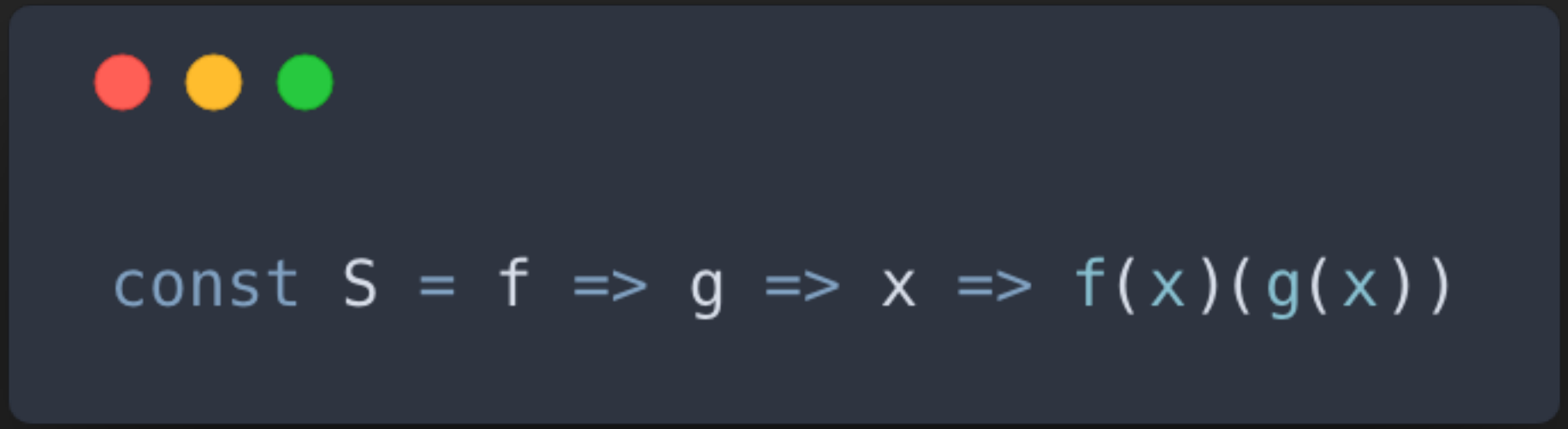
Replacing Functions (Successor)



```
const S = f => g => x => f(x)(g(x))
```

```
const SUCCESSOR = n => f => B(f)(n(f))
```

Replacing Functions (Successor)

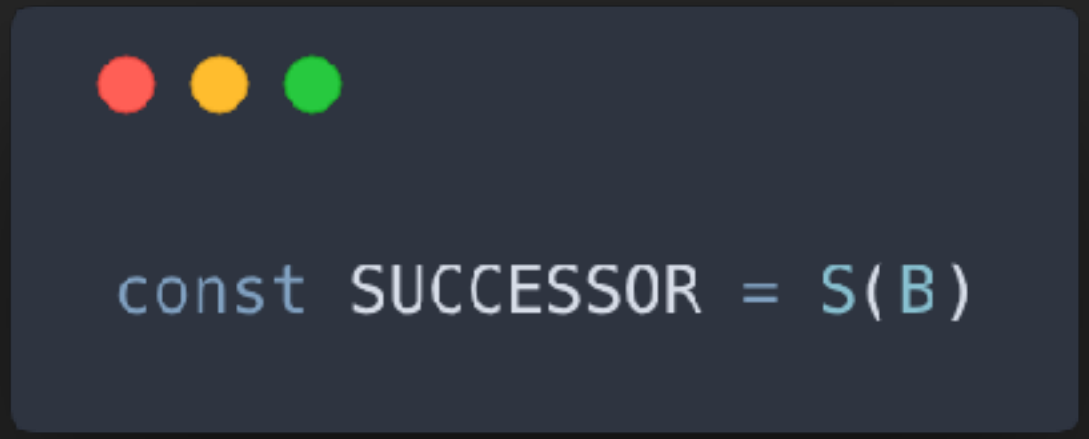


```
const S = f => g => x => f(x)(g(x))
```

```
const SUCCESSOR = n => f => B(f)(n(f))
```

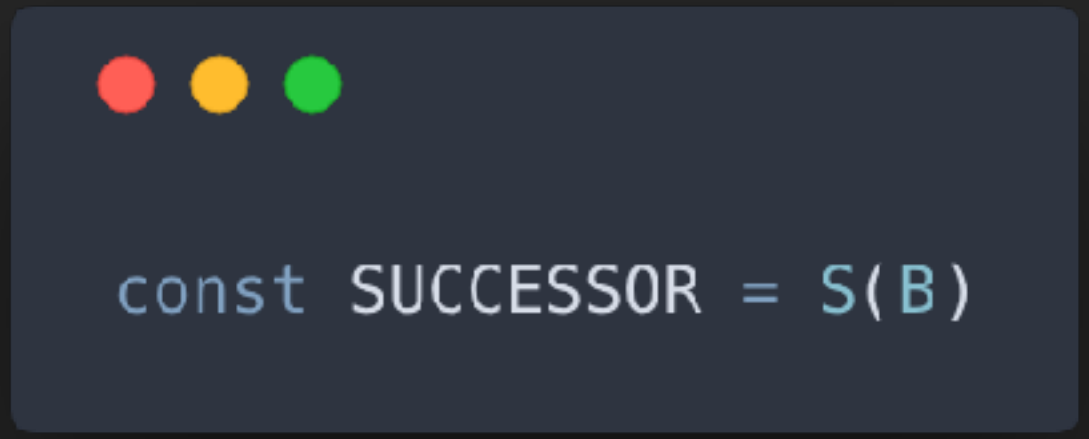
```
const SUCCESSOR = S(B)
```

Replacing Functions (Addition)



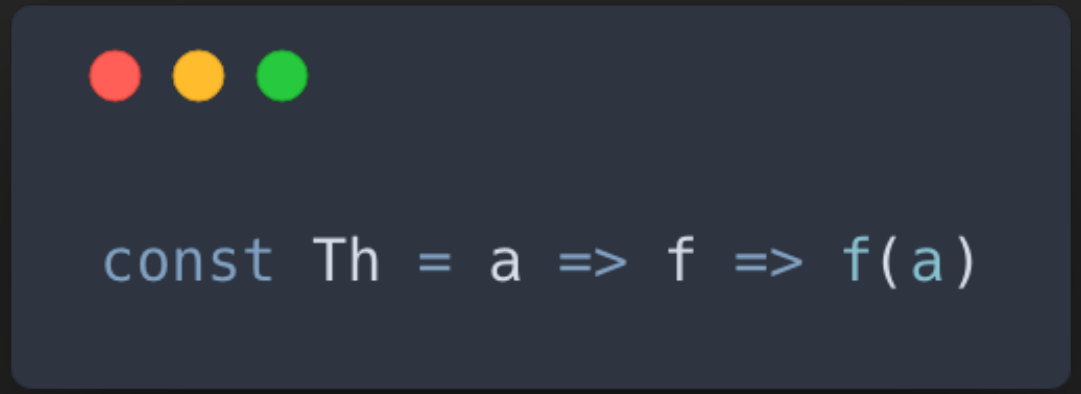

```
const ADDITION = n => k => n(SUCESSOR)(k)
```

Replacing Functions (Addition)




```
const ADDITION = n => k => n(S(B))(k)
```

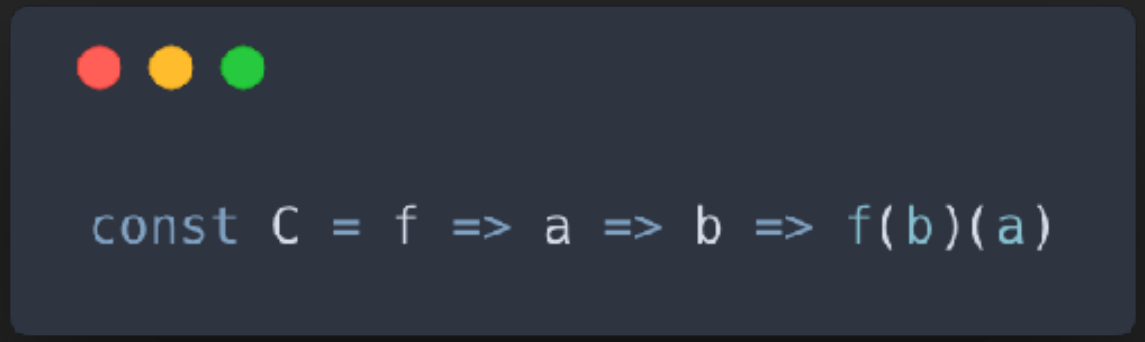
Replacing Functions (Addition)



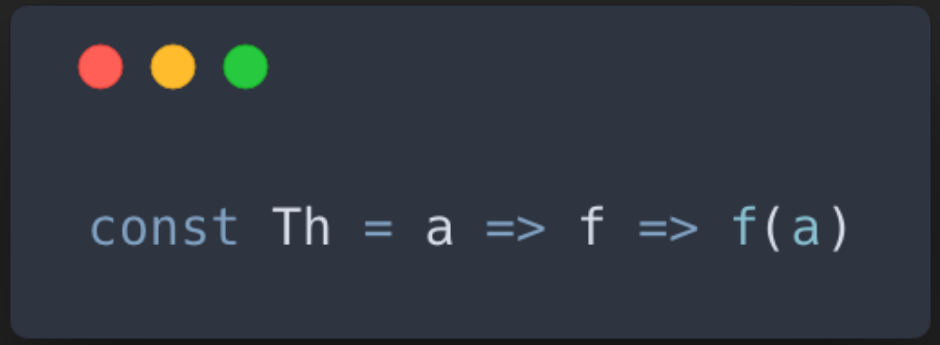
```
const Th = a => f => f(a)
```

```
const ADDITION = k => Th(S(B))(k)
```

Replacing Functions (Addition)



```
const C = f => a => b => f(b)(a)
```



```
const Th = a => f => f(a)
```

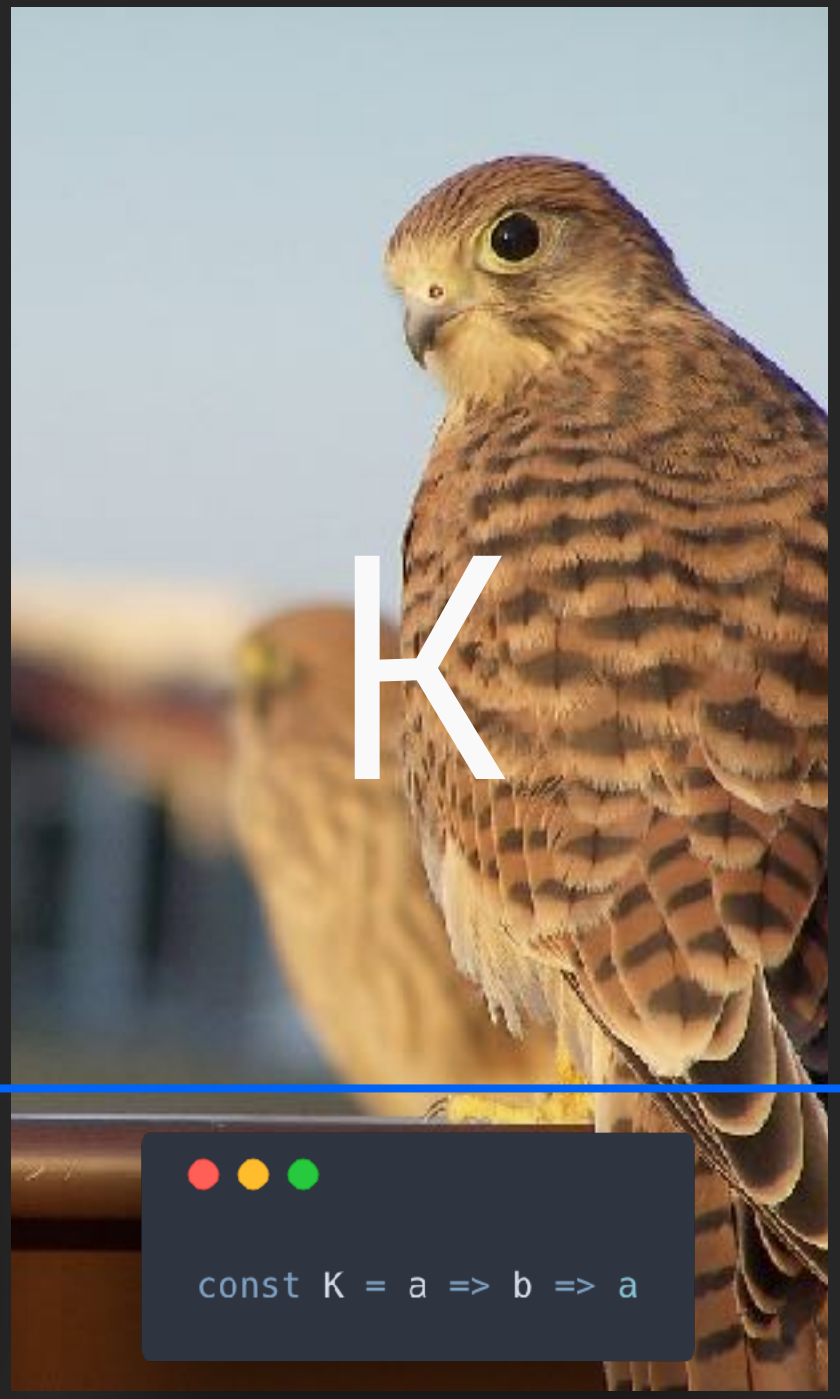
```
const ADDITION = C(Th)(Th(S(B)))
```

Ruining JavaScript and the Birds

Part 3


WHAT IF I TOLD YOU
YOU ONLY NEED
TWO COMBINATORS?
Yes, that's right. Two.


S
const S = f => g => x => f(x)(g(x))


K
const K = a => b => a


SK
SK Calculus

SK Calculus

People also call it SKI Calculus

SK Calculus

People also call it SKI Calculus

Because it's more convenient to have I

SK Calculus

People also call it SKI Calculus

Because it's more convenient to have I

```
const I = S(K)(K)
```

Replacing Numbers

SK Calculus

```
const KI = K(S(K)(K))
```

Replacing Numbers

SK Calculus

```
const KI = K(I)
```

SK Calculus

```
const KI = K(I)
const B = S(K(S))(K)
```

SK Calculus

```
const KI = K(I)
const B = S(K(S))(K)
const C =
((S((S(K((S(KS))K)))S))(KK))
```

Of course!

SK Calculus

http://www.angelfire.com/tx4/cus/combinator/birds.html

Combinator Birds

Function Abstraction	Symbol	Bird	Combinator	SK Combinator
λabc.a(bc)	B	Bluebird	S(KS)K	((S(KS))K)
λabcd.a(bcd)	B_1	Blackbird	BBB	[illegible]
λabcde.a(bcde)	B_2	Bunting	B(BBB)B	[illegible]
λabcd.a(b(cd))	B_3	Becard	B(BB)B	[illegible]
λabc.acb	C	Cardinal	S(BBS)(KK)	[illegible]
λabcd.ab(cd)	D	Dove	BB	[illegible]
λabcde.abc(de)	D_1	Dickcissel	B(BB)	[illegible]
λabcde.a(bc)(de)	D_2	Dovekies	BB(BB)	[illegible]
λabcde.ab(cde)	E	Eagle	B(BBB)	[illegible]
λabcdefg.a(bcd)(efg)	Ê	Bald Eagle	B(BBB)(B(BBB))	[illegible]
λabc.cba	F	Finch	ETTET	[illegible]
λabcd.ad(bc)	G	Goldfinch	BBC	[illegible]
λabc.abcb	H	Hummingbird	BW(BC)	[illegible]
λa.a	I	Identity Bird (aka Idiot)	SKK	((SK)K)
λabcd.ab(adc)	J	Jay	B(BC)(W(BC)(B(BBB)))	[illegible]
λab.a	K	Kestrel (True)	K	K
λab.a(bb)	L	Lark	CBM	[illegible]
λa.aa	M	Mockingbird	SII	((S((SK)K))((SK)K))
λab.ab(ab)	M_2	Double Mockingbird	BM	[illegible]
λab.b(ab)	O	Owl	SI	[illegible]
λabc.b(ac)	Q	Queer Bird	CB	((S(K(S((S(KS))K))))K)
λabc.a(cb)	Q_1	Quixotic Bird	BCB	[illegible]
λabc.b(ca)	Q_2	Quizzical Bird	C(BCB)	[illegible]
λabc.c(ab)	Q_3	Quirky Bird	BT	[illegible]
λabc.c(ba)	Q_4	Quacky Bird	F*B	[illegible]
λabc.bca	R	Robin	BBT	((S(K((S(KS))K)))((S(K(S((SK)K))))K))
λabc.ac(bc)	S	Starling	S	S
λab.ba	T	Thrush	CI	((S(K(S((SK)K))))K)
λab.b(aab)	U	Turing	LO	[illegible]
λabc.cab	V	Vireo (aka Pairing)	BCT	[illegible]

What does this mean?

What does this mean?

If we can replace all code by functions

What does this mean?

If we can replace all code by functions
Replace all functions by combinators

What does this mean?

If we can replace all code by functions
Replace all functions by combinators
And replace all combinators by S and K

What does this mean?

If we can replace all code by functions
Replace all functions by combinators
And replace all combinators by S and K

Then we can replace all code by S and K

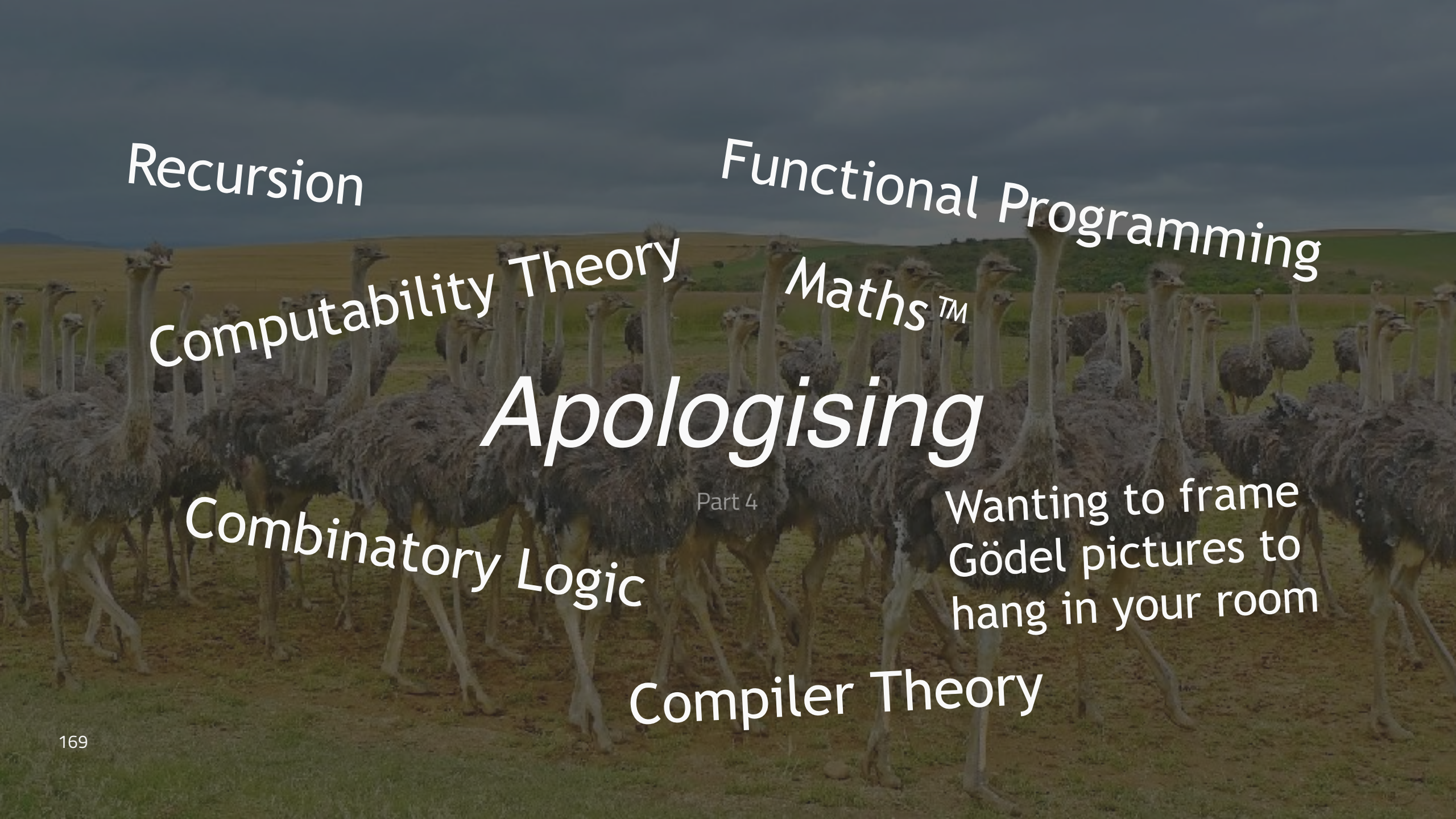
What does this mean?

https://crypto.stanford.edu/~blynn/lambda/sk.html

http://xn--wxak1a.com/blog/Combinators.html

Apologising

Part 4


Recursion
Functional Programming
Computability Theory
Maths™
Apologising
Part 4
Combinatory Logic
Wanting to frame Gödel pictures to hang in your room
Compiler Theory

References

http://codon.com/programming-with-nothing The blog post for the talk I mentioned in the beginning

https://speakerdeck.com/tomstuart/programming-with-nothing Slides for the talk "Programming with Nothing"

http://www.angelfire.com/tx4/cus/combinator/birds.html List of Notorious Combinators

https://bit.ly/2xpcPKn A Flock of Functions - Gabriel Lebec

https://amzn.to/2BVVsa1 To Mock a Mockingbird - Raymond Smullyan

http://computationbook.com Understanding Computation - Tom Stuart

https://www.youtube.com/watch?v=_kYGDJSm0gE - ASU Lectures - Lambda Calculus by Adam Doupé

Thank you!

@THEWIZARDLUCAS (TWITTER)

@LUCASFCOSTA (GITHUB)

LUCASFCOSTA.COM

Shhh, no tears. Only lambdas now.