



R. Mark Volkmann  
Object Computing, Inc.  
 <https://objectcomputing.com>  
 mark@objectcomputing.com  
 @mark\_volkmann



# $\lambda$ -calculus Overview



- Describes concepts that are fundamental to **functional programming**
  - **first-class functions** take other functions as arguments and can return a function
  - **currying** implements a function with multiple parameters as a sequence of functions that each have a single parameter
- **Purpose**
  - study how functions can interact with each other, not to calculate results in a useful or efficient way
- **Turing complete**
  - capable of performing any calculation or solving any computational problem, given enough time and memory

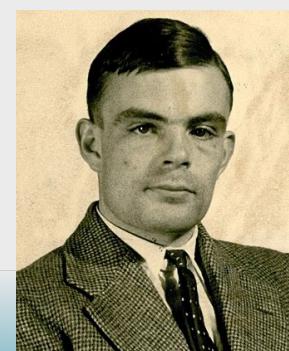
# Compared to Slide Rules



- Calculators are an excellent replacement for slide rules
  - It's not necessary to understand how to use a slide rule in order to use a calculator
  - But it's fascinating to learn how slide rules work (logarithms)
- 
- Likewise, it's not necessary to understand  $\lambda$ -calculus in order to be productive in modern programming languages
  - But it's fascinating to learn how much can be accomplished within the constraints of  $\lambda$ -calculus

# History

- **Gottlob Frege** (1848-1925)
  - studied use of functions in logic in 1893
- **Moses Schönfinkel** (1888-1942)
  - studied how combinators can be applied to formal logic in the 1920s
  - “Combinator” has two meanings, both of which describe a kind of function.  
The first describes functions that have no free variables.  
It combines only its arguments to produce a result.  
The second describes functions that take other functions  
and combine them to create a new function.
- **Alonzo Church** (1903-1985)
  - invented Lambda Calculus in the 1930s
  - was PhD advisor of Alan Turing (1912-1954)



# Concepts



- **Variable**
  - placeholder for a term represented by a single-letter name
  - two kinds of variables, bound and free ... discussed later
- **Lambda Abstraction**
  - defines anonymous functions that have exactly one parameter

$\lambda$ -calculus	JavaScript
$\lambda<\text{parameter}>.<\text{body}>$	$<\text{parameter}> => <\text{body}>$

$\lambda x. a \ b \ c \ x$  is evaluated as  $\lambda x. (((a \ b) \ c) \ x)$

expressions like  $\lambda x. \lambda y. \lambda z. a \ b \ c$  are sometimes written in the shorthand form  $\lambda xyz. a \ b \ c$

- **Application**
  - calls functions with an argument

$\lambda$ -calculus	JavaScript
$(\lambda<\text{parameter}>.<\text{body}>) \ <\text{arguments}>$	$(<\text{parameter}> => <\text{body}>)(<\text{arguments}>)$
$(\lambda xyz.<\text{body}>) \ a \ b \ c$	$(x => y => z => <\text{body}>)(a)(b)(c)$

$<\text{arguments}>$  is a whitespace-separated list of expressions

# $\lambda$ -calculus Does Not Define

- Any built-in functions
  - Syntax for values such as Booleans, numbers, and strings
  - Operators on those types
- 
- However, alternatives to those can be defined using only concepts on previous slide, which is the amazing thing about  $\lambda$ -calculus



# Bound vs. Free Variables

- Bound variables
  - bound by a specific abstraction (function)
  - appear as function parameters and represent an input value
- Free variables
  - appear in function definitions and are not parameters
  - can represent any value from the “environment”



pretending to have a  
+ function for now;  
will define later

$\lambda$ -calculus	JavaScript	Bound Variables	Free Variables
$\lambda x. (+ x 1)$	$x \Rightarrow x + 1$	x	none
$\lambda x. (+ y 1)$	$x \Rightarrow y + 1$	none	y
$\lambda x. x \lambda x. (+ x 1)$	$x \Rightarrow x((x \Rightarrow x + 1))$	rename 2nd x as shown below	
$\lambda x. x \lambda y. (+ y 1)$	$x \Rightarrow x((y \Rightarrow y + 1))$	x and y	none

# Evaluation Rules

- $\lambda$ -calculus defines four evaluation rules
  - **$\alpha$ -conversion** (alpha)
  - **$\beta$ -reduction** (beta)
  - **$\delta$ -rule** (delta)
  - **$\eta$ -conversion** (eta)



# $\alpha$ -conversion

$\alpha$

- Changes names of bound variables, resulting in equivalent functions
- Examples
  - function  $\lambda x . x$  is equivalent to  $\lambda y . y$
  - function  $\lambda f x . f (+ x 1)$  is equivalent to  $\lambda g y . g (+ y 1)$

# $\delta$ -rule (delta)

$\delta$

- Evaluates functions that are assumed to be built-in
- Example
  - `(+ 1 2)` can be evaluated to 3

# $\beta$ -reduction

$\beta$

- Applies arguments to a function
- Result is determined by substituting argument values for all occurrences of the function parameter
- Examples
  - $(\lambda x. (+ x 3)) 2$  evaluates to  $(+ 2 3)$  which evaluates to 5
  - $(\lambda f x. f (+ x 1))$  takes two arguments, a function and a number
    - apply two arguments with  $(\lambda f x. f (+ x 1)) (\underline{\lambda x. (* x 2)}) \underline{3}$
    - result is  $(\lambda x. (* x 2)) (+ 3 1)$
    - apply  $\delta$  rule to second term to get  $(\lambda x. (* x 2)) 4$
    - apply  $\beta$  rule again to obtain  $(* 4 2)$
    - one more use of  $\delta$ -rule, gives 8

# $\eta$ -conversion



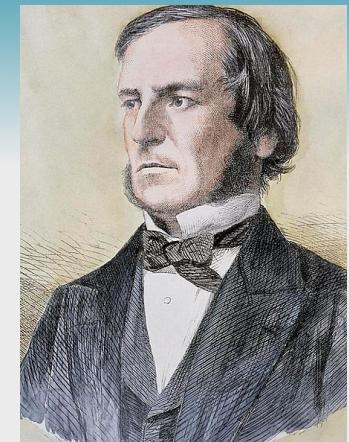
- Replaces function that has an explicit parameter with one that takes an implicit parameter, creating a **point-free** version of the function
- Example
  - $\lambda x. (+ x 1)$  is equivalent to  $(+ x)$  because + is a function that takes two arguments, but only one is supplied

# Boolean Values

- **True** - function that takes two arguments and always returns the first

$\lambda$ -calculus	JavaScript
$\lambda t. \lambda f. t$	$t \Rightarrow f \Rightarrow t$

parameter names  
are arbitrary



George Boole  
(1815-1864)

- **False** - function that takes two arguments and always returns the second

$\lambda$ -calculus	JavaScript
$\lambda t. \lambda f. f$	$t \Rightarrow f \Rightarrow f$

```
// Adding underscores to avoid
// conflicting with JavaScript keywords.
const true_ = x => y => x;
const false_ = x => y => y;
```

# Not

- Function to return “not” of a Boolean value, where **b** is either the true or false function

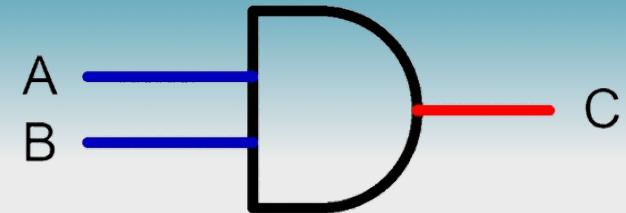
$\lambda$ -calculus	JavaScript
$\lambda b.b \text{ false true}$	<code>b =&gt; b(false_)(true_)</code>

```
const not = b => b(false_)(true_);
```



- Examples
  - $(\lambda b.b \text{ false true}) (\lambda t. \lambda f. t)$  evaluates to false
  - $(\lambda b.b \text{ false true}) (\lambda t. \lambda f. f)$  evaluates to true

# And



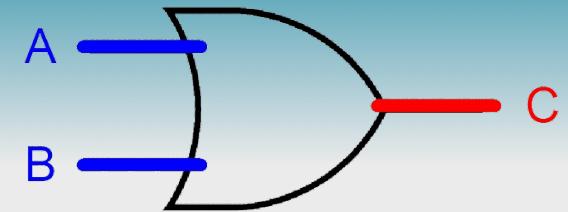
- Function to return “and” of two Boolean values, where **x** and **y** the true or false function

$\lambda$ -calculus	JavaScript
$\lambda x. \lambda y. x \ y \ \text{false}$	<code>x =&gt; y =&gt; x(y)(false_)</code>

```
const and = x => y => x(y)(false_);
```

- If first argument is false, that is result
- Otherwise second argument is result

# Or



- Function to return “or” of two Boolean values, where **x** and **y** the true or false function

$\lambda$ -calculus	JavaScript
$\lambda x. \lambda y. x \text{ true } y$	$x \Rightarrow y \Rightarrow x(\text{true\_})(y)$

```
const or = x => y => x(true_)(y);
```

- If first argument is true, that is result
- Otherwise second argument is result

# Church Numerals



- Represent natural numbers by functions that take another function and a value
- The passed function is called some number of times

Number	$\lambda$ term	JavaScript
0	$\lambda f x. x$	$f \Rightarrow x \Rightarrow x$ ←
1	$\lambda f x. f \ x$	$f \Rightarrow x \Rightarrow f(x)$
2	$\lambda f x. f \ (f \ x)$	$f \Rightarrow x \Rightarrow f(f(x))$
3	$\lambda f x. f \ (f \ (f \ x))$	$f \Rightarrow x \Rightarrow f(f(f(x)))$

same as function for false

```
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)))));
```

Alonzo Church  
(1903-1985)

- Also used to repeat an operation  $n$  times

# Successor

- Function to return next number after a given number
- Applies function passed to a number one more time

$\lambda$ -calculus	JavaScript
$\lambda n \ (\lambda f. \ \lambda x. \ f \ (n \ f \ x))$	$n \Rightarrow f \Rightarrow x \Rightarrow f(n(f)(x))$

```
const succ = n => f => x => f(n(f)(x));
```

- Example

- **succ(two)** is **three**
- **f(f(x)) -> f(f(f(x)))**

**Peano numbers** are numbers that include zero and the results of repeatedly applying the successor function.

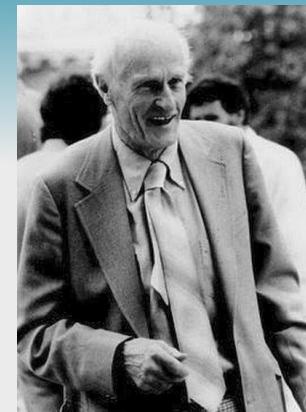


Giuseppe Peano  
(1858-1932)

# Predecessor ...

- Function to return previous number before a given number
- Alonzo Church couldn't find a solution,  
but his student Stephen Kleene did
- Recipe
  - start with pair  $(0, 0)$
  - to get predecessor of  $n$ ,  
create a new pair from previous pair  $n$  times
    - 1st number in new pair is 2nd number in previous pair
    - 2nd number in new pair is successor of its 1st number
  - take first number of final pair

while in a dentist chair  
waiting to have  
wisdom teeth removed



Stephen Kleene  
(1909-1994)

## Predecessor of 5?

initial:  $(0, 0)$   
1st:  $(0, 1)$   
2nd:  $(1, 2)$   
3rd:  $(2, 3)$   
4th:  $(3, 4)$   
5th:  $(4, 5)$

works because  
of delay in  
incrementing  
the first number

# ... Predecessor ...



- Requires 4 helper functions
  - represent a pair (**pair**)

$\lambda$ -calculus	JavaScript
$\lambda x. \lambda y. \lambda f. f \ x \ y$	$x \Rightarrow y \Rightarrow f \Rightarrow f(x)(y)$

- get 1st element of pair (**fst**)

$\lambda$ -calculus	JavaScript
$\lambda p. p \text{ TRUE}$	$p \Rightarrow p(\text{true}_\circ)$

- get 2nd element of pair (**snd**)

$\lambda$ -calculus	JavaScript
$\lambda p. p \text{ FALSE}$	$p \Rightarrow p(\text{false}_\circ)$

- create new pair from existing (**phi**)

$\lambda$ -calculus	JavaScript
$\lambda p. \text{pair } (\text{snd } p) \ (\text{succ } (\text{snd } p))$	$p \Rightarrow \text{pair}(\text{snd}(p))(\text{succ}(\text{snd}(p)))$

```
const pair = x => y => f => f(x)(y);
const fst = p => p(true_);
const snd = p => p(false_);
const phi = p => pair(snd(p))(succ(snd(p)));
```

# ... Predecessor



- Putting it all together

$\lambda$ -calculus	JavaScript
$\lambda n.fst (n\ \phi\ (\text{pair}\ zero\ zero))$	$n \Rightarrow \text{fst}(n(\phi)(\text{pair(zero)}(\text{zero})))$

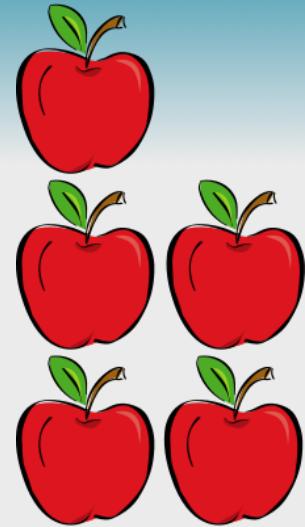
```
const pred = n => fst(n(phi)(pair(zero)(zero)));
```

applies **phi** function **n** times  
to the pair (0 , 0)

- Another way to write this that is harder to follow

```
const pred = n => f => x => n(g => h => h(g(f)))(u => x)(u => u);
```

# Addition



- Can be seen as iterated successors
- To find  $m + n$ , start with  $n$  and call **succ** on it  $m$  times

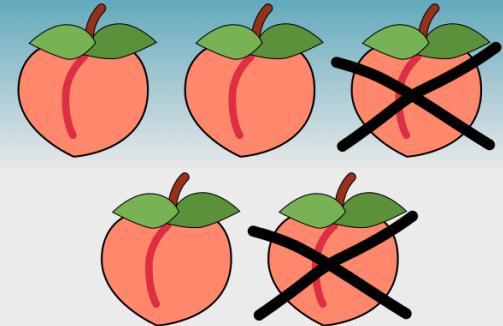
$\lambda$ -calculus	JavaScript
$\lambda mn. (m \text{ succ})\ n$	$m \Rightarrow n \Rightarrow m(\text{succ})(n)$

```
const add = m => n => m(succ)(n);
```

- $O(n)$

```
3 + 2 ?  
  
add(three)(two)  
three(succ)(two)  
succ(succ(succ(two)))  
succ(succ(three))  
succ(four)  
five
```

# Subtraction



- Can be seen as iterated predecessors
- To find  $m - n$ , start with  $m$  and call **pred** on it  $n$  times

$\lambda$ -calculus	JavaScript
$\lambda mn. (n \text{ pred}) m$	$m \Rightarrow n \Rightarrow n(\text{pred})(m)$

```
const sub = m => n => n(pred)(m);
```

- $O(n)$

```
5 - 2 ?  
  
sub(five)(two)  
two(pred)(five)  
pred(pred(five))  
pred(four)  
three
```

# Multiplication

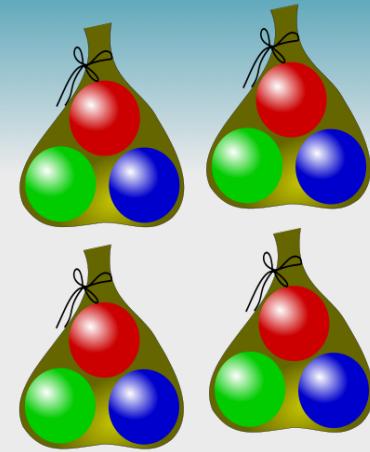
- Can be seen as iterated addition
- To find  $m * n$ , start with **zero** and call **add(n)** on it  $m$  times

$\lambda$ -calculus	JavaScript
$\lambda mn. m (add\ n)\ 0$	$m \Rightarrow n \Rightarrow m(\text{add}(n))(\text{zero})$

```
const mul = m => n => m(add(n))(zero);
```

- $O(n^2)$

```
4 * 3 ?  
  
mul(four)(three)  
four(add(three))(zero)  
add(add(add(add(three))))(zero)  
add(add(add(three)))(three)  
add(add(three))(six)  
add(three)(nine)  
twelve
```



# Exponentiation

**5<sup>3</sup>**  
base exponent

- Can be seen as iterated multiplication
- To find  $m^n$ , start with **one** and call **mul(m)** on it **n** times

$\lambda$ -calculus	JavaScript
$\lambda mn. n (\text{mul } m) \ 1$	$m \Rightarrow n \Rightarrow n(\text{mul}(m))(\text{one})$

- $O(n^3)$
- Alternative definition

```
const exp = m => n => n(m);
```

```
const exp = m => n => n(mul(m))(one);
```

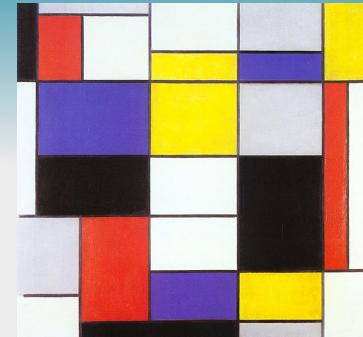
2<sup>3</sup> ?

```
three(mul(two))(one)
(mul(two)(mul(two)(mul(two))))(one)
(mul(two)(mul(two)))(two)
(mul(two))(four)
eight
```

2<sup>3</sup> ?

```
1st: f(f(x))
- replace each f with two calls to f
2nd: f(f(f(f(x))))
- replace each f with two calls to f
3rd: f(f(f(f(f(f(f(x)))))))
- result is function for 8
```

# Function Composition



- Function that combines other functions
- Example
  - composing a function that adds 2 with a function that multiplies by 3

λ-calculus	JavaScript
$\lambda f g x. f (g x)$	<code>f =&gt; g =&gt; x =&gt; f(g(x))</code>

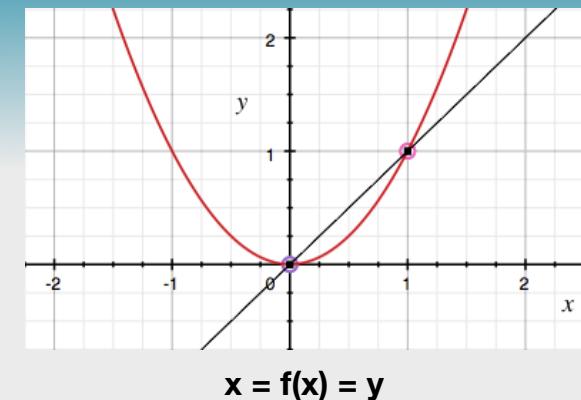
```
const compose = f => g => x => f(g(x));
```

- When functions representing two numbers are composed, the result is their product (multiplication)
  - so this function can be used in place of the one we saw for multiplication

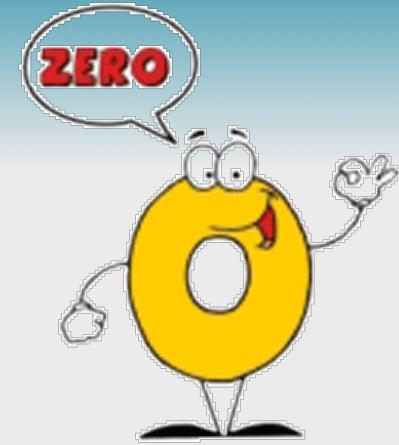
When `f` and `g` are number functions, this returns a function that calls a given function `g` times and repeats this `f` times resulting in `f * g` calls which is the number function for their product.

# Fixed Points

- Some functions have a fixed point
  - value that can be passed to it that results in same value
  - ex.  $\text{sqrt}(1) = 1$  and  $\cos(0.739085) \approx 0.739085$
- $\lambda$ -calculus function  $\lambda x.x$  has a fixed point of  $\lambda x.x$ 
  - $(\lambda x.x)(\lambda x.x)$  - left term is a function and right is argument
  - substituting argument for all  $x$  in function body gives same thing, so evaluating this never ends
  - called “Omega Combinator” ( $\Omega$ )



# Is Zero



- Function to determine if a given number is zero
- Recall that **zero** function is same as **false** function, which always returns its second argument

$\lambda$ -calculus	JavaScript
$\lambda n. n (\lambda x. \text{FALSE}) \text{ TRUE}$	<code>n =&gt; n(x =&gt; false_)(true_)</code>

```
const iszero = n => n(x => false_)(true_);
```

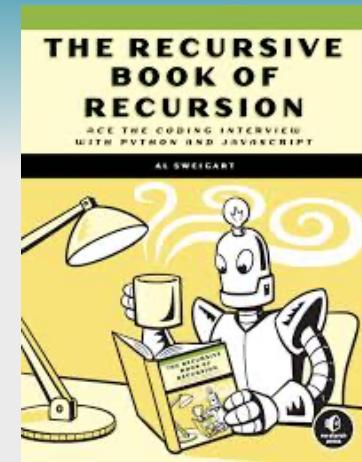
If **n** is zero, this will return the 2nd argument which is **true\_**. Otherwise it will call 1st argument function **n** times and every call will return **false\_**.

# Recursion

- $\lambda$ -calculus functions do not have names, so they can't call themselves recursively
- Enter **Y Combinator!**
  - function that adds a function parameter to Omega combinator in order to call a provided function repeatedly

```
 $\lambda f. (\lambda x. x\ x) \ (\lambda x. f\ (x\ x))$ 
```

```
const Y = f => (x => x(x))(x => f(y => x(x)(y))) ;
```



# Factorial

- Can use Y Combinator to implement a factorial function

```
const facgen = f => n => iszero(n) () => one() => mul(n)(f(pred(n))))();  
const factorialY = Y(facgen);
```

tests n and returns  
either the 1st  
or 2nd function

1st  
function

2nd  
function

calls  
selected  
function

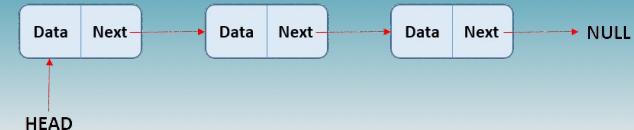
3! ?

```
mul(three)(f(pred(three))  
mul(three)(mul(two)(f(pred(two))))  
mul(three)(mul(two)(f(one)))  
mul(three)(mul(two)(mul(one)(f(pred(one))))  
mul(three)(mul(two)(mul(one)(f(zero))))  
mul(three)(mul(two)(mul(one)((() => one))))  
mul(three)(mul(two)(one))  
mul(three)(two)  
six
```

Technically this uses **Z Combinator**  
rather than **Y Combinator**  
so 1st and 2nd functions are  
called **lazily** rather than eagerly.

**Haskell** is an example of a programming language  
that automatically performs lazy evaluation  
and can use Y Combinator.

# Linked Lists



- Can simulate linked lists with “cons cells” short for “construct”
  - each cell holds a pair of values
  - **cons** function returns a cons cell
  - **car** function takes a cons cell and returns first element
  - **cdr** function takes a cons cell and returns last element
  - **nil** function is used to mark end of a linked list
- Examples

```
const cons = a => b => f => f(a)(b);
const car = p => p(true_);
const cdr = p => p(false_);
const nil = f => x => null;
```

```
const pair = cons(one)(two);
expect(car(pair)).toBe(one);
expect(cdr(pair)).toBe(two);

const list = cons(one)(cons(two)(cons(three)(nil)));
expect(car(list)).toBe(one);
expect(car(cdr(list))).toBe(two);
expect(car(cdr(cdr(list)))).toBe(three);
expect(cdr(cdr(cdr(list)))).toBe(nil);
```

code from unit tests

# Testing



- It's useful to convert  $\lambda$ -calculus representations to actual boolean values and numbers
- Can be accomplished with following JavaScript functions where
  - **b** is a  $\lambda$ -calculus function that represents true or false
  - **n** is a  $\lambda$ -calculus function that represents a natural number

```
const jsbool = b => b(true)(false);
const jsnum = n => n(x => x + 1)(0);
```

# Resources

- **Lambda calculus page on Wikipedia**
  - [https://en.wikipedia.org/wiki/Lambda\\_calculus](https://en.wikipedia.org/wiki/Lambda_calculus)
- **Learn X in Y minutes Where X=Lambda Calculus**
  - <https://learnxinyminutes.com/docs/lambda-calculus/>
- **Intro to hacking with the lambda calculus - blog post by L Rudolf L**
  - <https://www.lesswrong.com/posts/D4PYwNtYNwsogoixGa/intro-to-hacking-with-the-lambda-calculus>
- **Fundamentals of Lambda Calculus & Functional Programming in JavaScript - YouTube talk by Gabriel Lebec**
  - <https://www.youtube.com/watch?v=3VQ382QG-y4>



# Wrap Up

- You don't need to know this
- But it's super cool to see how much can be represented and implemented using only functions!
- Use this knowledge to impress your programmer friends
- See my runnable code at  
<https://github.com/mvolkmann/lambda-calculus/>
  - in file `lambda-calculus.test.ts`
  - install Bun and enter `bun test`

