LAMBDA λ CALCULUS

EST. 1936

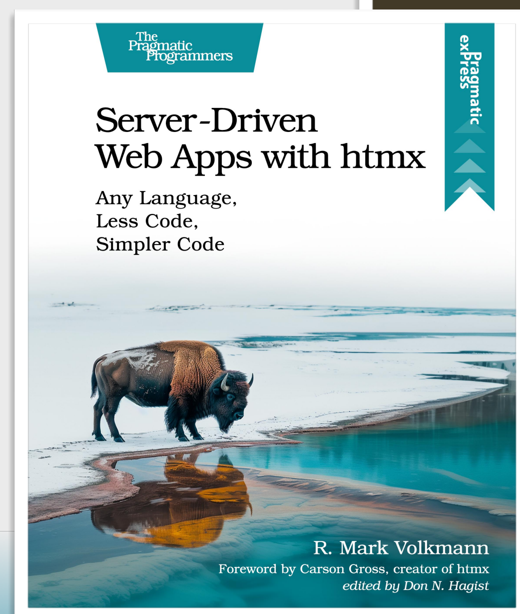
R. Mark Volkmann
https://mvolkmann.github.io
r.mark.volkmann@gmail.com

OBJECT COMPUTING
YOUR OUTCOMES ENGINEERED

Slides at **https://github.com/mvolkmann/talks/**

About Me

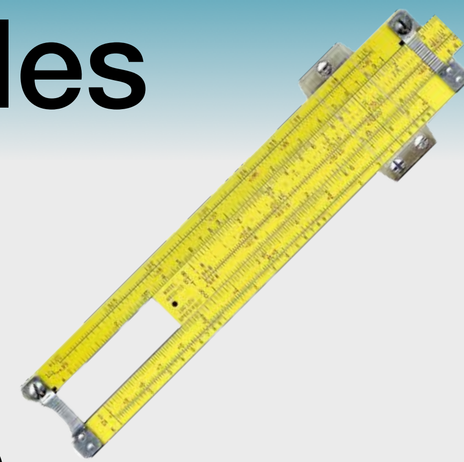
- Partner and Distinguished Software Engineer
  at Object Computing, Inc. in St. Louis, Missouri USA
- 45 years of professional software development experience
- Writer and speaker
- Blog at https://mvolkmann.github.io/blog/
- Author of Manning book “Svelte ... in Action”
- Author of Pragmatic Bookshelf book
  “Server-Driven Web Apps with htmx”



λ-Calculus Overview

- Describes concepts that are fundamental to **functional programming**
  - **first-class functions** take other functions as arguments and can return functions
  - **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 λ-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 λ-calculus

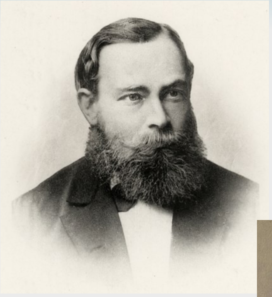
History

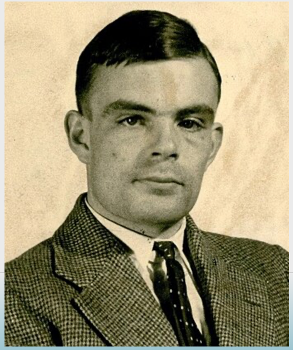
- **Gottlob Frege** (1848-1925)
  - studied use of functions in logic in 1893
- **Moses Schönfinkel** (1888-1942)
  - studied how combinators can be applied to formal logic in 1920s
  - “Combinator” has two meanings, both of which describe a kind of function.
    The first describes functions that have 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.

pure functions

function composition

- **Alonzo Church** (1903-1985)
  - invented Lambda Calculus in the 1930s
    before computers existed!
  - was PhD advisor of Alan Turing (1912-1954)





JavaScript Functions

```
function add(a, b) {
  const result = a + b;
  return result;
}
```

```
function add(a, b) {
  return a + b;
}
```

```
const add = (a, b) => {
  const result = a + b;
  return result;
};
```

```
const add = (a, b) => {
  return a + b;
};
```

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

```
const result = add(2, 3); // 5
```

```
const curriedAdd = a => {
  return b => a + b;
};
```

```
const curriedAdd = a => b => a + b;
```

```
const add2 = curriedAdd(2);
const result = add2(3); // 5
```

```
const result = curriedAdd(2)(3); // 5
```

Concepts

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

λ-calculus	JavaScript
λ<parameter>.<body>	<parameter> => <body>

λx.a b c x is evaluated as **λx.(((a b) c) x)**

expressions like **λx.λy.λz.a b c** are sometimes written in the shorthand form **λxyz.a b c**

- **Application**
  - calls a function (or sequence of them) with arguments

λ-calculus	JavaScript
(λ<parameter>.<body>) <arguments>	(<parameter> => <body>)(<arguments>)
(λxyz.<body>) a b c	(x => y => z => <body>)(a)(b)(c)

λ-calculus Does Not Define

- Does not define
  - syntax for values such as Booleans, numbers, and strings
  - operators on those types
  - any built-in functions
- However, alternatives to those can be defined using only concepts on previous slide, which is the amazing thing about λ-calculus!
- All you have are functions
  - they take a single argument that is a function
  - they return a single function

Bound vs. Free Variables

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

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

function bodies are underlined in blue

λ-calculus	JavaScript	Bound Variables	Free Variables
`λx.(+ x 1)`	`x => x + 1`	x	none
`λx.(+ y 1)`	`x => y + 1`	none	y
`λx.x λx.(+ x 1)`	`x => x((x => x + 1))`	rename 2nd x as shown below	
`λx.x λy.(+ y 1)`	`x => x((y => y + 1))`	x and y	none

Evaluation Rules

- λ-calculus defines four evaluation rules
  - α**-conversion** (alpha)
  - **β-reduction** (beta)
  - **δ-rule** (delta)
  - **η-conversion** (eta)

α-conversion (alpha)

- Changes the name of a bound variable, resulting in an equivalent function
- Examples
  - function `λx.x` is equivalent to `λy.y`
  - function `λfx.f (+ x 1)` is equivalent to `λgy.g (+ y 1)`

δ-rule (delta)

δ

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

β-reduction (beta)

β

- Applies arguments to a function
- Result is determined by substituting argument value for all occurrences of a function parameter in body
- Examples
  - `(λx.(+ x 3)) 2` evaluates to `(+ 2 3)` which evaluates to `5`
  - `(λfx.f (+ x 1))` evaluates two arguments, a function and a number
    - apply both arguments with `(λfx.f (+ x 1)) (λx.(* x 2)) 3`
    - apply β-reduction to obtain `(λx.(* x 2)) (+ 3 1)`
    - apply δ-rule to second term to get `(λx.(* x 2)) 4`
    - apply β-reduction again to obtain `(* 4 2)`
    - apply δ-rule to obtain `8`

1st arg for `f`

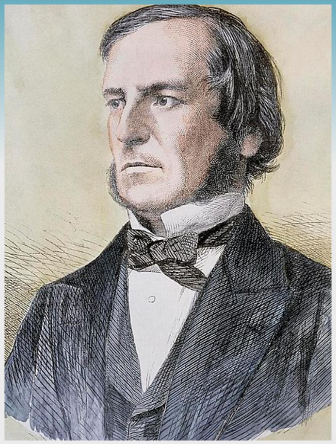
2nd arg for `x`

η-conversion (eta)

η

- Replaces function that has an explicit parameter with one that takes an implicit parameter, creating a **point-free** version of the function
  - a way of defining a function without explicitly mentioning arguments
- Example
  - **`λx.(+ 1 x)`** is equivalent to **`(+ 1)`**
    because **`+`** is a a function that takes two arguments,
    but only one is supplied

Boolean Values



George Boole
(1815-1864)

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

λ-calculus	JavaScript
`λt. λf. t`	`t => f => t`

parameter names are arbitrary

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

λ-calculus	JavaScript
`λt. λf. f`	`t => f => f`

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

Not

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

λ-calculus	JavaScript
λb.b false true	b => b(false_)(true_)

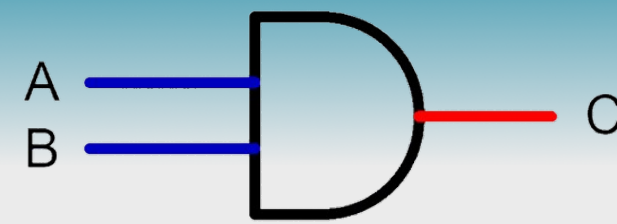
```
const not = b => b(false_)(true_);
```

- Examples
  - **(λb.b false true) (λt. λf. t)**
    (λt. λf. t) false true
    false
  - **(λb.b false true) (λt. λf. f)**
    (λt. λf. f) false true
    true

function that represents true

function that represents false

And



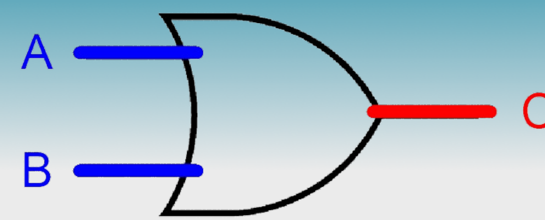
- Function to return “and” of two Boolean values, where **x** and **y** the true or false function

λ-calculus	JavaScript
`λx. λy. x y false`	`x => y => x(y)(false_)`

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

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

Or



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

λ-calculus	JavaScript
λx. λy. x true y	x => y => x(true_)(y)

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

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

Church Numerals

Alonzo Church (1903-1985)

- Represent whole numbers [non-negative integers] by functions that take another function and a value
- The passed function is called some number of times

Number	λ term	JavaScript
0	`λfx.x`	`f => x => x`
1	`λfx.f x`	`f => x => f(x)`
2	`λfx.f (f x)`	`f => x => f(f(x))`
3	`λfx.f (f (f x))`	`f => x => 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)));
```

- Also used to repeat an operation **`n`** times

Successor

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

λ-calculus	JavaScript
`λn (λf. λx. f (n f x))`	`n => f => x => f(n(f)(x))`

n
times

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

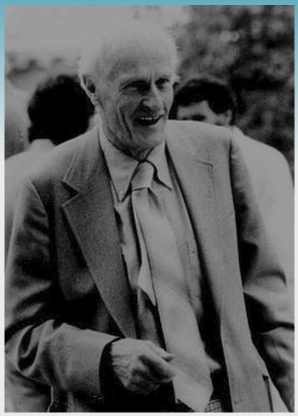
one
more
time

- Example
  - **`succ(two)`** is **`three`**
  - from **`f(f(x))`** we get **`f(f(f(x)))`**
- O(n)

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

Giuseppe Peano (1858-1932)

Predecessor ...



Stephen Kleene
(1909-1994)

- 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

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`**)

λ-calculus	JavaScript
`λx.λy.λf.f x y`	`x => y => f => f(x)(y)`

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

λ-calculus	JavaScript
`λp.p TRUE`	`p => p(true_)`

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

λ-calculus	JavaScript
`λp.p FALSE`	`p => p(false_)`

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

λ-calculus	JavaScript
`λp.pair (snd p) (succ (snd p))`	`p => pair(snd(p))(succ(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

λ-calculus	JavaScript
`λn.fst (n phi (pair zero zero))`	`n => fst(n(phi)(pair(zero)(zero)))`

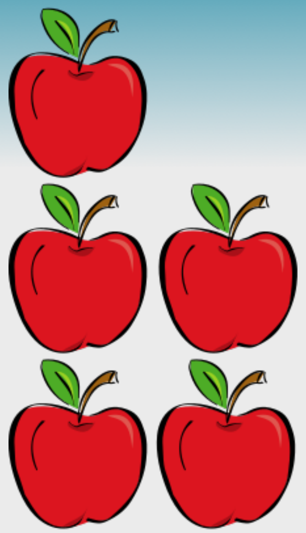
```
const pred = n => fst(n(phi)(pair(zero)(zero)));
```

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

- O(n)
- Another way to write this that is harder to follow, but doesn't need helper functions

```
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

λ-calculus	JavaScript
`λmn. (m succ) n`	`m => n => m(succ)(n)`

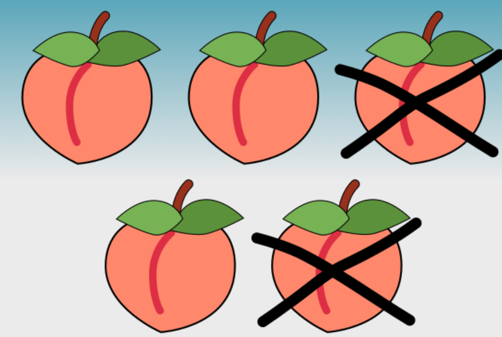
```
const add = m => n => m(succ)(n);
```

```
3 + 2 ?

add(three)(two)
three(succ)(two)
succ(succ(succ(two)))
succ(succ(three))
succ(four)
five
```

- O(n)

Subtraction



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

λ-calculus	JavaScript
`λmn. (n pred) m`	`m => n => n(pred)(m)`

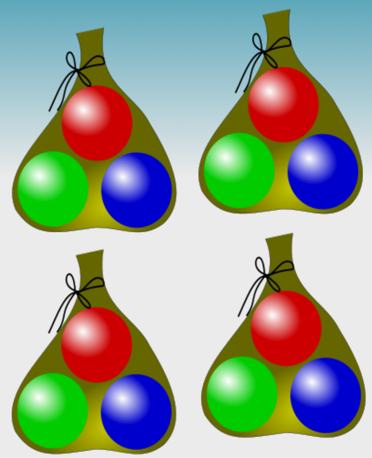
```
const sub = m => n => n(pred)(m);
```

```
5-2?

sub(five)(two)
two(pred)(five)
pred(pred(five))
pred(four)
three
```

- O(n^2)
  - because **`pred`** is O(n) and is called n times

Multiplication



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

λ-calculus	JavaScript
λmn. m (add n) 0	m => n => m(add(n))(zero)

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

- O(n^2)

```
4 * 3 ?

mul(four)(three)
four(add(three))(zero)
(add(three)(add(three)(add(three)(add(three)))))(zero)
(add(three)(add(three)(add(three))))(three)
(add(three)(add(three)))(six)
(add(three))(nine)
twelve
```

Exponentiation

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

λ-calculus	JavaScript
`λmn. n (mul m) 1`	`m => n => n(mul(m))(one)`

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

```
2^3 ?

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

- O(n^3)

- Alternative definition

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

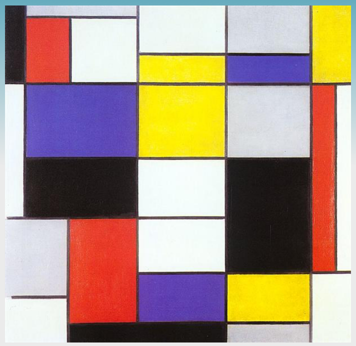
passes a number function to another number function

```
2^3 ?

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(f(x))))))))
- result is function for eight
```

applies the function for two a total of three times

Function Composition



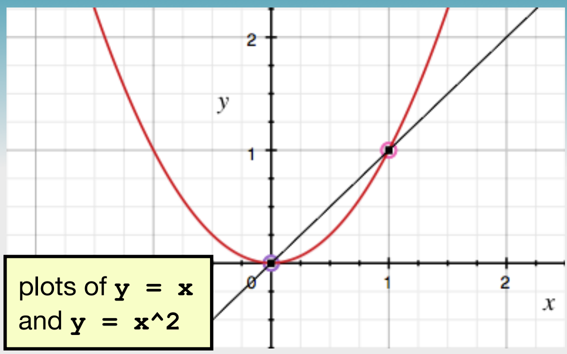
- Function that combines other functions
- Example
  - composing a function that adds 2 with a function that multiplies by 3

λ-calculus	JavaScript
`λfgx.f (g x)`	`f => g => x => f(g(x))`

```
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

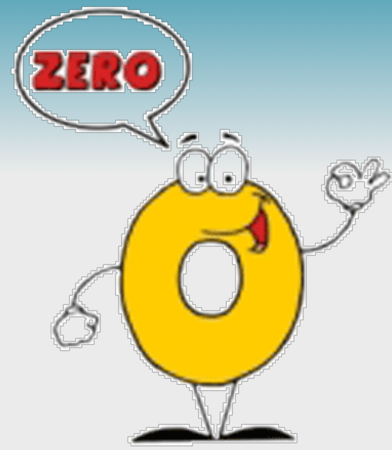
Fixed Points



x = f(x) = y

- Some functions have a fixed point
  - value that can be passed to it that results in same value
  - ex. **`sqrt(1) = 1`** and **`cos(0.739085) ≅ 0.739085`**
- λ-calculus function **`λx.x x`** has a fixed point of **`λx.x x`**
  - **`(λx.x x)(λx.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” (Ω)
- We need this to implement recursion

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

λ-calculus	JavaScript
λn.n (λx.FALSE) TRUE	n => n(x => false_)(true_)

```
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_**.

- O(n)
- We need this to implement recursion

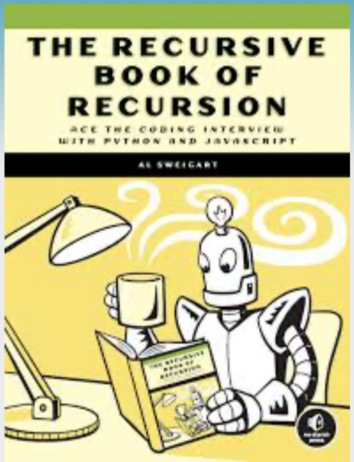
Recursion

- λ-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

```
λf.(λx.x x) (λx.f (x x))
```

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

  - discovered by Haskell Curry



Haskell Curry (1900-1982)

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);
```

defined on previous slide

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

1st function

2nd function

calls selected function

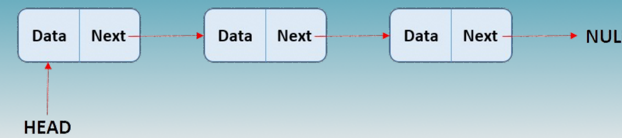
3! ?

```
mul(three)(f(pred(three))
mul(three)(mul(two)(f(pred(two))))
mul(three)(mul(two)(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

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

When car or cdr are called, true_ or false_ becomes the value of **f** here

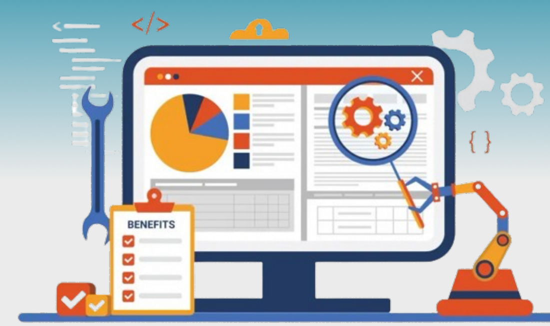
- Examples

code from unit tests

```
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);
```

Testing



- It’s useful to convert λ-calculus representations to actual boolean and number values
- Can be accomplished with following JavaScript functions where
  - **b** is a λ-calculus function that represents true or false
  - **n** is a λ-calculus function that represents a whole number

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

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`**
  - **amazingly fast!** only 95ms to run all the tests

See Smalltalk version at https://github.com/mvolkmann/Cuis-Smalltalk-LambdaCalculus

Resources

- **Lambda calculus page on Wikipedia**
  - 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/D4PYwNtYNwsgoixGa/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

My Latest Effort

- See npm package **wrec** which greatly simplifies creating web components
  - https://www.npmjs.com/package/wrec
- Name is acronym for **W**eb **RE**active **C**omponents

```
import Wrec, {html} from './wrec.min.js';

class HelloWorld extends Wrec {
  static properties = {
    name: {type: String, value: 'World'}
  };

  static html = html`<div>Hello, <span>this.name</span>!</div>`;
}

HelloWorld.register();
```

```
<hello-world name="Mark"></hello-world>
```