\[ \lambda \]

Matt Might
University of Utah
matt.might.net
@mattmmight
Lambda is everywhere!

All you need is lambda.

Lambda multiplies you.
Sapir-Whorf Hypothesis

Edward Sapir

Language limits thought.

Benjamin Whorf
Challenge
Write factorial without using recursion.
Or iteration.
Or conditionals.
Or any numbers.
What is λ?
\( \lambda \) is a notation
for anonymous functions
```python
>>> f = lambda x: x + 1
>>> f(3)
4
```
$\lambda$ is a language
λ is a proof theory
λ is a way of life
Lambda is everywhere!

All you need is lambda.

Lambda multiplies you.
Lambda is everywhere!
lambda \( \nu_1, \ldots, \nu_n \): exp
(lambda (v1 ... vn) exp)
\((\lambda (v_1 \ldots v_n) \; exp)\)
function \((v_1, \ldots, v_n)\) { return \textit{exp} ; }
new Procedure () {
    public T run (T_1 v_1, ..., T_n v_n) {
        return exp ;
    }
}
\( (T_1 \ nu_1, \ldots, T_n \ nu_n) \rightarrow \exp \)
function \((v_1, \ldots, v_n)\) return \(exp\) end
function (v_1, \ldots, v_n) \text{ use (free)} \{ \text{return } exp \; ; \}
lambda \{|v_1, \ldots, v_n| \text{return } \text{exp} \}
$(v_1 : t_1, \ldots, v_2 : t_2) => \\text{exp}$
\[ \text{free}(T_1 \ v_1, \ldots, T_n \ v_n) \{ \text{return } \exp; \} \]
Origins of $\lambda$
Origins of notation
(Supposedly) Euclid

300 BC
230104 ΒΑΣΙΛΕΩΣ

Δύο αριθμοί διδέντων μή πρῶτον πρὸς ἀλλήλους τὸ μέγιστον αὐτῶν κοινὸν μέτρον εὑρέθην.

gcd(a, b) = (b == 0) ? a : gcd(b, a mod b)

'Εστωσαν οἱ διδέντες δύο αριθμοὶ μὴ πρῶτοι πρὸς ἀλλήλους οἱ ΑΒ, ΓΔ. δεὶ δὴ τῶν ΑΒ, ΓΔ τὸ μέγιστον κοινὸν μέτρον εὑρέθην.

Εἰ μὲν οὖν οἱ ΓΔ τῶν ΑΒ μετρεῖ, μετρεῖ δὲ καὶ εὑστόν, οἱ ΓΔ ἀρα τῶν ΓΔ, ΑΒ κοινὸν μέτρον ἔστιν. καὶ φανερὸν, ὅτι καὶ μέγιστον συνεδίδει γὰρ μεῖζὸν τοῦ ΓΔ τὸν ΓΔ μετρήσει.

Εἰ δὲ οὐ μετρεῖ οἱ ΓΔ τῶν ΑΒ, τῶν ΑΒ, ΓΔ ἀνδυραρρομένου δεῖ τὸν ἐλάσσονον ἁπὸ τοῦ μεῖζονς λειψάνηται τὶς ἀρίθμοις, δε μετρήσει τὸν πρὸς εὑστὸ. μονάς μὲν γὰρ οὐ λειψάνηται: εἰ δὲ μὴ, ἔστωσαν οἱ ΑΒ, ΓΔ πρῶτοι πρὸς ἀλλήλους· ὅπερ οὐχ ὑπόκειται. λειψάνηται τὶς ἀρα ἀρίθμοις, δε μετρήσει τὸν πρὸς εὑστὸ. καὶ ο μὲν ΓΔ τὸν ΒΕ μετρῶν λειπέτω εὑστόν ἐλάσσονα τὸν ΕΑ, ο δὲ ΕΑ τὸν ΔΖ μετρῶν λειπέτω εὑστόν ἐλάσσονα τὸν ΖΓ, ο δὲ ΓΖ τὸν ΑΕ μετρήσει. ἐπεὶ οὖν οἱ ΓΖ τὸν ΑΕ μετρεῖ, ο δὲ ΑΕ τὸν ΔΖ μετρεῖ, καὶ ο ΓΖ ἀρα τὸν ΔΖ μετρήσει. μετρεῖ δὲ καὶ εὑστόν καὶ ἄλογον ἄρα τὸν ΓΔ μετρήσει. ο δὲ ΓΔ τὸν ΒΕ μετρεῖ· καὶ ο ΓΖ ἀρα τὸν ΒΕ μετρεῖ· μετρεῖ δὲ καὶ τὸν ΕΑ· καὶ ἄλογον ἄρα τὸν ΒΕ μετρήσει: μετρεῖ δὲ καὶ τὸν ΓΔ· ο ΓΖ ἀρα τῶν ΑΒ, ΓΔ μετρεῖ. ο ΓΖ ἀρα τῶν ΑΒ, ΓΔ κοινὸν μέτρον ἔστιν. λέγω δὴ, ὅτι καὶ μέγιστον. εἰ γὰρ μὴ ἔστιν ο ΙΔ τῶν ΑΒ, ΓΔ μέγιστον κοινὸν μέτρον, μετρήσει τις τοὺς ΑΒ, ΓΔ ἀρίθμοις ἀρίθμοις μεῖζον ἄν τοῦ ΓΖ. μετρίται, καὶ ἔστω ὁ Η. καὶ ἐπεί οἱ Η τὸν ΓΔ μετρεῖ, ο δὲ ΓΔ τὸν ΒΕ μετρεῖ, καὶ ο Η ἀρα τὸν BE μετρεῖ· μετρεῖ δὲ καὶ ἄλογον τὸν ΒΕ· καὶ λοιπὸν ἀρα τὸν ΑΕ μετρήσει. ο δὲ ΑΕ τὸν ΔΖ μετρεῖ· καὶ ο Η ἁρα τὸν ΔΖ μετρήσει· μετρεῖ δὲ καὶ ἄλογον τὸν ΔΖ· καὶ λοιπὸν ἁρα τὸν ΓΔ· καὶ λοιπὸν ἁρα τὸν ΓΖ· ὃς εἶ σεν ἀδύνατον· οὐκ ἁρα τοὺς ΑΒ, ΓΔ ἀρίθμοις ἀρίθμοις τις μετρήσει μεῖζον ἄν τοῦ ΓΖ· ὃς εἶ σεν ἀδύνατον· ὃς εἶ σεν ἀδύνατον· ὃς εἶ σεν ἀδύνατον. εἰς τὸν ΑΒ, ΓΔ μετρίταιν ἐστὶ κοινὸν μέτρον [ὅπερ έδει δεῖξαι].

Πόρισμα.

'Εκ δὴ τούτου φανερὸν, ὅτι ἢν ἀρίθμος δύο ἀρίθμοις μετρήσῃ, καὶ τὸ μέγιστον αὐτῶν κοινὸν μέτρον μετρήσει· ὅπερ έδει δεῖξαι.
“Limitations”
No zero
No variables/unknowns
No operators
Long division needed Ph.D.
(Supposedly) Hippasus
BURN THE WITCH!
$1 + 3x = x^2$
The number such that adding one to three of the root of that number is equal to that number.
The number such that adding one to three of the root of that number is equal to that number.

Brahmagupta (?)

Indian numerals (596)
The number such that adding 1 to 3 of the root of that number is equal to that number.

Brahmagupta (?)

Indian numerals (596)

Nagari numerals around 11th century A.D.
et

The number such that adding 1 to 3 of the root of that number is equal to that number.
The number such that adding 1 to 3 of the root of that number is equal to that number.
The number such that adding 1 to 3 of the root of that number is equal to that number.
The number such that adding $1$ to $3$ of the root of that number is equal to that number.
The number such that adding $1$ to $3$ of the root of that number is equal to that number.
The number such that $1 + 3$ of the root of that number is equal to that number.
The number such that $1 + \frac{3}{3}$ of the root of that number is equal to that number.
The number such that adding one to three of the root of that number is equal to that number.

$$1 + 3x$$ is equal to $x$.

(Probably) François Viète

Variables (1570)
1 + 3\pi \text{ is equal to } \pi
$1 + 3\pi$ is equal to $\frac{10\pi}{3}$
1 + 3\pi is equal to \frac{17}{6}
Early 1600s

William Oughtred

\[ 1 + 3x = \pi \]
Isaac Newton

Principia (late 1600s)

\[ 1 + 3x = x^2 \]
1 + 3x = x^2
formae generalis autem sumendo $n = 3n$ praebeat
\[\frac{\int \frac{1}{x^{i-1}} dx}{\int \frac{1}{x^{i+1}} dx} = k \int \frac{x^{2i-1}}{(1-x^i)^{i+1}} dx,\]

quibus coniungendis adipiscimur
\[\frac{\left(\int \frac{1}{x^{i-1}} dx\right)^4}{\int \frac{1}{x^{i+1}} dx} = \rho \int \frac{x^{2i-1}}{(1-x^i)^{i+1}} \int \frac{x^{2i-1}}{(1-x^i)^{i+1}} \int \frac{x^{2i-1}}{(1-x^i)^{i+1}} dx.\]

Sit nunc $n = \frac{1}{4}$ et sumatur $k = 4$ etique
\[\frac{\int \frac{1}{x^{i-1}} dx}{\int \frac{1}{x^{i+1}} dx} = \rho \int \frac{x^{2i-1}}{(1-x^i)^{i+1}} \int \frac{x^{2i-1}}{(1-x^i)^{i+1}} \int \frac{x^{2i-1}}{(1-x^i)^{i+1}} dx.\]

**COROLLARIUM 1**

35. Si igitur sit $i = 1$, habebimus
\[\int dx \frac{1}{x^{i-1}} = \rho \int \frac{dx}{x^i} \int \frac{x^{2i}}{(1-x^i)^{i+1}} \int \frac{x^{2i}}{(1-x^i)^{i+1}} \int \frac{x^{2i}}{(1-x^i)^{i+1}} dx.\]

qua expressio si littera $P$ designetur, erit in genere
\[\int dx \frac{1}{x^{i-1}} = \frac{1}{4} \cdot \frac{5}{4} \cdot \frac{9}{4} \cdot \ldots \cdot \frac{4n-3}{4} P.\]

**COROLLARIUM 2**

36. Pro altero casu principali sumamus $i = 3$ etique
\[\int dx \frac{1}{x^{i-1}} = \rho \int \frac{dx}{x^i} \int \frac{x^{2i}}{(1-x^i)^{i+1}} \int \frac{x^{2i}}{(1-x^i)^{i+1}} \int \frac{x^{2i}}{(1-x^i)^{i+1}} dx.\]

seu facta reductione ad simpliciores formas
\[\int dx \frac{1}{x^{i-1}} = \rho \int \frac{dx}{x^i} \int \frac{x^{2i}}{(1-x^i)^{i+1}} \int \frac{x^{2i}}{(1-x^i)^{i+1}} \int \frac{dx}{x^i}.\]

\[f(x) = 1 + 3x - x^2\]
f(x)
\( f^2 \)
$x \quad \rightarrow \quad x^2 + 3$
\texttt{lambda x: x*x + 3}
What else is out there?
\{1, 2\}
Unification of math & logic
Every integer greater than two can be written as the sum of two primes.
\( \forall n : (n > 2) \implies \exists a, b : p(a) \land p(b) \land a + b = n \)
Gottlob Frege

Bertrand Russell

Sets all the way down!

Nope.
Alonzo Church
(My great\textsuperscript{5} grand advisor!)
Why not functions?
\[ f(x) = x^2 - 4 \]
\( f = \lambda x. x^2 - 4 \)
\( \lambda x. x^2 - 4 \)
\((\lambda x. x^2 - 4)(2)\)
λ-calculus
\( e_1(e_2) \)
λν.ε
\[ u \ | \ \lambda u. e \ | \ e_1(e_2) \]
\( v \)

\( \lambda v. e \)

\( e_1(e_2) \)
\( \lambda v : e \) \( e_1(e_2) \)
Lambda is everywhere!
All you need is lambda.
Lambda multiplies you.
All you need is lambda.
lambda => language
Compiling to lambda-calculus: Turtles all the way down

My compilers class always starts with a full lecture on the lambda-calculus.
It leaves behind only the dedicated.

\[ \lambda f. (\lambda x. (f (x x)) \lambda x. (f (x x))) \]

_Barendregt is a helluva drug._

The lambda-calculus is a minimal programming language.
The only constructs are free and bound function applications.
Variables are all bound.

My compilers class always starts with a full lecture on the lambda-calculus.
It leaves behind only the dedicated.

_Barendregt is a helluva drug._

The lambda-calculus is a minimal programming language.
The only constructs are free and bound function applications.
Variables are all bound.
Multi-argument functions
f(y, y)
lambda $x: e$
lambda $x, y : e$
\texttt{lambda } x : \texttt{lambda } y : e
\( f(x, y) \)
f(x)(y)
$ python

>>> f = lambda x,y: x + y

>>> g = lambda x: lambda y: x + y

>>> f(1,2)
3

>>> f(1,2)
3

>>> g(1)(2)
3

>>> g(1)(2)
3
Booleans
IF (cond) (true) (false)
IF (TRUE) (true) (false)
true
IF (FALSE) (true) (false)
false
IF (cond) (true) (false)
cond (true) (false)
IF (cond) (true) (false)
$$\text{IF} = (\lambda c : 
    \lambda t : 
    \lambda f : 
    c(t)(f))$$
TRUE = (lambda t:
    lambda f:
    t)
FALSE = (lambda t:
    lambda f:
    f)
Natural numbers
What is a natural?
Zero
Another natural + 1
How do we use naturals?
Iteration!
ZERO \( (f) (z) \)
ZERO \ (f) \ (z) \ = \ = \ z
ONE \((f)\) \((z)\)
ONE \( (f) (z) = f(z) \)
TWO \((f)(z)\) \(=\) \(f(f(z))\)
THREE \((f)(z) == f(f(f(z)))\)
ZERO \( (f) (z) = z \)
ZERO = (lambda f :
            lambda z :
                f(z))
SUCC(n)
SUCC(n)(f)(z) = f(n(f)(z))
SUCC = n f z f(n(f)(f)(z))
SUCC = ( lambda n:
            lambda f:
                lambda z: f(n(f)(z)))
SUM = (lambda a:
    lambda b:
        lambda f:
            lambda z:
                a(f)(b(f)(z)))
MUL = (lambda a:
    lambda b:
        lambda f:
            lambda z:
                a(lambda z:b(f)(z))(z))
Pairs
PAIR (a) (b)
\text{LEFT} \ (\text{PAIR} \ (a) \ (b)) = a
\texttt{RIGHT (PAIR (a) (b)) = b}
PAIR = (lambda a:
    lambda b:
        lambda f: f(a)(b))
RIGHT = (lambda p:
           p(lambda l: lambda r: r))
LEFT = (lambda p:
    p(lambda l: lambda r: l))
Lists
NIL
CONS (hd) (tl)
\text{HEAD} \ (\text{CONS} \ (hd) \ (tl)) = hd
TAIL (CONS (hd) (tl)) = tl
\text{CONSP} \ (\text{CONS} \ (hd) \ (tl)) \ = \ \text{TRUE}
NILP (CONS (hd) (tl)) = FALSE
NILP (NIL) = TRUE
CONSP (NIL) = FALSE
VOID = lambda _:_

NIL = lambda oncons: lambda onnil: onnil(VOID)

CONS = (lambda hd: lambda tl:
    lambda oncons: lambda onnil:
        oncons(hd)(tl))

CONSP = lambda l: l (lambda hd: lambda tl: TRUE) (lambda void: FALSE)

NILP = lambda l: l (lambda hd: lambda tl: FALSE) (lambda void: TRUE)

HEAD = lambda l: l (lambda hd: lambda tl: hd) (VOID)

TAIL = lambda l: l (lambda hd: lambda tl: tl) (VOID)
Recursion
Non-termination
(lambda f:f(f))(lambda g:g(g))
\[ U = \lambda f: f(f(f)) \]
U(U)
Recursion is self-reference
self-application

=>

self-reference
\( f = \ldots f \ldots \)
\( f = \ldots U(h) \ldots \)
\[ f = U(\text{lambda } h: \ldots U(h) \ldots) \]
\[ f = U(\lambda h: \ldots \ U(h) \ldots) \]
fact = U(lambda h:
lambda n: 1 if n <= 0 else n * (U(h))(n-1))
We can do better!
fact = U(lambda h:
    lambda n: 1 if n <= 0 else n * (U(h))(n-1))
fact = Y(lambda f:
    lambda n: 1 if n <= 0 else n * f(n-1))
Define recursive function...
...as fixed point...
...of non-recursive function.
Define fixed point finder...
...without recursion.
$x$ is a **fixed point** of $F$ if $F(x) = x$. 
Y(F) = x such that x = F(x)
\[ Y(F) = x \quad \text{such that} \quad x = F(Y(F)) \]
\[ Y(F) = F(Y(F)) \]
Y = lambda F: F(Y(F))
\[ Y = \lambda F : F(\lambda x : Y(F)(x)) \]
\[ Y = \mathcal{U}(\lambda h: \lambda F: F(\lambda x: \mathcal{U}(h)(F)(x))) \]
\[ Y = U(\lambda h:\lambda F:F(\lambda x:h(h)(F)(x))) \]
\[ Y = ((\lambda h: \lambda F: F(\lambda x: h(h)(F)(x)))) \\
(\lambda h: \lambda F: F(\lambda x: h(h)(F)(x)))) \]
$ python
>>> Y = ((lambda h: lambda F: F(lambda x: h(h)(F)(x))))
    (lambda h: lambda F: F(lambda x: h(h)(F)(x))))

>>> fact = Y(lambda f: lambda n: 1 if n <= 0 else n * f(n-1))

>>> fact(8)
40320
Fixed-point combinators in JavaScript: Memoizing recursive functions

It comes as a surprise to many programmers that it is possible to express a "recursive" function like factorial without using recursion or iteration.

The technique involved is subtle but powerful: the recursive function is computed as the "fixed point" of a non-recursive function. To compute the fixed point, we can use the **Y combinator**, which is itself a non-recursive function that computes fixed points.

*That this manages to work is truly remarkable.*

--Sussman and Steele on the Y Combinator

As a practical application of this theory, recursive functions expressed as fixed points allow the use of a memoizing fixed-point combinator. The combinator approach to recursion makes it possible to cache the internal calls to a recursive function automatically.

For example, this caching turns the naive, exponential implementation of Fibonacci into the optimized, linear-time version *for free.*

Read below to see how to do this all of this in JavaScript, courtesy of its anonymous function construct.
Yacc is dead: An update

[article index] [email me] [@mattmight] [+mattmight] [rss]

The draft of "Yacc is Dead" that David Darais and I posted on arXiv received some attention, so we'd like to share updates and feedback.

(You don't have to read the draft to read this post; it's self-contained.)

The draft claims that Brzozowski's derivative eases parser implementation; that it handles all CFGs (yes, including left-recursive); that it seems to be efficient in practice; and that it should eventually be efficient in theory too.

We reply to Russ Cox's "Yacc is Not Dead" by running the example that he criticized with our parser, and find that it does indeed generate a parser as fast as -- almost as fast as -- a nonrecursive parser.
Lambda multiplies you.
λ is a gateway drug
Higher-order programming (map, fold, reduce, …)

Functional programming

Laziness

Monads → relational programming

Continuation-passing style → nodejs

Continuations → time travel, logic programming

Type theory

Curry-Howard isomorphism → dependent types

theorem proving
Thanks!
matt.might.net
TRUE = (lambda t:
           lambda f:
           t)
TRUE = (lambda t:
    lambda f:
    t())
FALSE = (lambda t:
    lambda f:
    f)
FALSE = (lambda t:
    lambda f:
        f())
IF (cond)
(lambda: true)
(lambda: false)