

# Lambda Calculi and You

Clarissa Littler

June 22, 2017

Introduction

Syntax and  
calculation

Code is data

Control flow

Lessons learned

# What you'll learn

What the  $\lambda$  calculus is, how to calculate with it, and lessons to draw from it

Introduction

Syntax and  
calculation

Code is data

Control flow

Lessons learned

# History of $\lambda$ calculus



$$\hat{\ } \rightarrow \Lambda \rightarrow \lambda$$

# What is it?

- $\lambda x.M$
- $MN$
- $x$

# Let's break it down

$\lambda x.M$

```
function (x) {  
  M  
}
```

```
(lambda x: M)
```

```
{ |x| M }
```

# Sample programs

$$\text{id} = \lambda x. x$$
$$\text{double} = \lambda f. \lambda x. f(f\ x)$$
$$\text{if} = \lambda b. \lambda t. \lambda f. b\ t\ f$$

# Substitution: where computation happens

$$(\lambda x.M)N \rightarrow N[M/x]$$

You've seen this

```
def sillyFun(x):  
    y = x + 2  
    print(y)  
    return (x*x)
```



You've seen this

```
sillyFun(3):  
  y = 3 + 2  
  print(y)  
  return (3 * 3)
```

# Evaluation order

Functions before arguments

$$MN \rightarrow (\lambda x. I)N$$

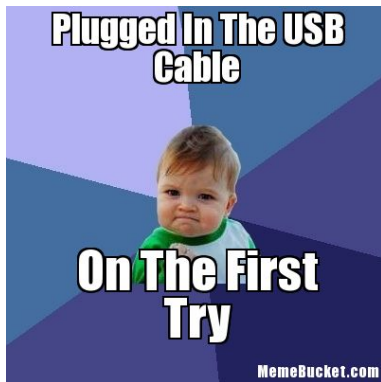
# Capture avoidance

$(\lambda x. \lambda y. x y) y \rightarrow \lambda y. y y$   
But *that* can't be right!

# Rename variables

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

Can't get it right



# Is this a real language?

Believe it or not, everything we need is here

# Church encodings

Church encodings are representations of *data* as *functions* that use the data

# Natural numbers

Numbers are functions of the form  $\lambda s.\lambda z.??$

$$0 := \lambda s.\lambda z.z$$
$$S := \lambda n.\lambda s.\lambda z.s(n\ s\ z)$$



# How are **these** numbers?

$$1 := S(0) = \lambda s. \lambda z. s(0 \ s \ z) = \lambda s. \lambda z. s \ z$$

$$2 := S(1) = \lambda s. \lambda z. s(1 \ s \ z) = \lambda s. \lambda z. s(s \ z)$$

# The meaning of a natural number

The number  $N$  represents doing *something*  $N$  times

$\text{double} = 2$

# Definite iteration

Natural numbers encapsulate the act of definite iteration

$$m + n := m(S)(n)$$

$$1 + 1 = 1(S)(1) = S(1) = 2$$

$$2 + 2 = 2(S)(2) = S(S(2)) = 4$$

$$3 + 5 = 3(S)(5) = S(S(S(5))) = 8$$

$$m * n := m(n(S))(0)$$

$$\begin{aligned} 1 * 1 &= 1(1(S))(0) \\ &= 1(S)(0) = S(0) = 1 \end{aligned}$$

$$\begin{aligned} 2 * 2 &= 2(2(S))(0) \\ &= 2(S)(2(S)(0)) = 2(S)(2) = 4 \end{aligned}$$

# Booleans

We represent true and false as *functions*

$$\text{true} := \lambda t. \lambda f. t$$
$$\text{false} := \lambda t. \lambda f. f$$

# The bool is the choice

if-expression:

$$\text{if} := \lambda b. \lambda t. \lambda f. b \ t \ f$$

examples:

$$\text{if}(\text{true})(x)(y) = \text{true}(x)(y) = x$$

$$\text{if}(\text{false})(x)(y) = \text{false}(x)(y) = y$$

the choice is built into the booleans themselves

Pair types (two things joined together):

$$\text{pair} := \lambda l. \lambda r. \lambda p. p(l)(r)$$
$$\text{fst} := \lambda p. p(\lambda l. \lambda r. l)$$
$$\text{snd} := \lambda p. p(\lambda l. \lambda r. r)$$

We could also have written:

$$\text{fst} := \lambda p. p(\text{true})$$
$$\text{snd} := \lambda p. p(\text{false})$$

A list is empty, or an element followed by a list

$$\text{nil} := \lambda c. \lambda n. n$$
$$\text{cons}(x, xs) := \lambda c. \lambda n. c\ x(xs\ c\ n)$$



# Understanding reduce/fold

List with three elements

$$\begin{aligned}
 \text{ourList} &:= \text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil}))) \\
 \text{ourList}(+, 0) &= 1 + \text{cons}(2, \text{cons}(3, \text{nil}))(+, 0) \\
 &= 1 + 2 + \text{cons}(3, \text{nil})(+, 0) \\
 &= 1 + 2 + 3 + \text{nil}(+, 0) \\
 &= 1 + 2 + 3 + 0
 \end{aligned}$$

`[1,2,3].reduce(function (x,y) {return x + y},0)`

# A lesson from Church encodings

Thinking inductively  $\Rightarrow$  modular code

# What about control flow?

We've *almost* shown Turing completeness

# Recursion

$$Y := \lambda f. (\lambda x. f(x x)) (\lambda x. f(x x))$$

# A simple proof it works

$$\begin{aligned} Y(g) &= (\lambda x. g(xx))(\lambda x. g(xx)) \\ &= g((\lambda x. g(xx))(\lambda x. g(xx))) \\ &= g(Y(g)) \end{aligned}$$

# Sequencing code

$$l_1; l_2 \Rightarrow (\lambda x. l_2) l_1$$

# Variable binding\*

$$\text{let } x = v \text{ in } M \Rightarrow (\lambda x.M)v$$

# Global variable binding\*

Easiest with variable hoisting

$$\text{var } x = v; M \Rightarrow (\lambda x. M)v$$



# Compilation as language design

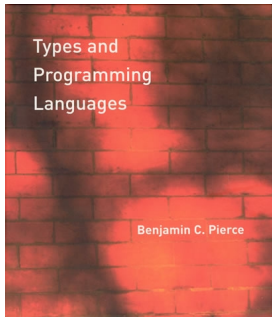
You can experiment with features via compilation  
between languages

# $\lambda$ : a common language

The  $\lambda$  calculus can be found inside many languages

# $\lambda$ : a PL toolkit

The common language  
of PL researchers



# $\lambda$ : a way to understand computation

Formal mathematical models let us get at the heart of  
computation

# Questions

Any Questions?

## Bonus slides

GUESS WE HAD MORE  
TIME!

Mutable  
variables can be  
*simulated*

What about *mutable  
variables*?

## The secret origin of monads

### Notions of Computation and Monads

EUGENIO MOGGI\*

*Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, UK*

The  $\lambda$ -calculus is considered a useful mathematical tool in the study of programming languages, since programs can be *identified* with  $\lambda$ -terms. However, if one goes further and uses  $\beta\eta$ -conversion to prove equivalence of programs, then a gross simplification is introduced (programs are identified with total functions from values to values) that may jeopardise the applicability of theoretical results. In this paper we introduce *calculus*, based on a categorical semantics for *computations*, that provide a correct basis for proving equivalence of programs for a wide range of *notions of computation*. © 1991 Academic Press, Inc.

# Closures + state

$\lambda + \text{state} = \text{everything}$

