

# Lambda Calculus 1

Sven Tennie

July 8, 2018

# Lambda Calculus

- ▶ Invented by Alonzo Church (1920s)
- ▶ Equally expressive to the Turing Machine(s)
- ▶ Formal Language
- ▶ Computational Model
  - ▶ Lisp (1950s)
  - ▶ ML
  - ▶ Haskell
- ▶ "Lambda Expressions" in almost every modern programming language

# Why should I care?

- ▶ Simple Computational Model
  - ▶ to describe structure and behaviour (E.g. Operational Semantics)
  - ▶ to reason and prove
- ▶ Explains why things in FP are like they are
  - ▶ pure functions
  - ▶ higher-order functions
  - ▶ currying
  - ▶ lazy evaluation
- ▶ Understand FP Compilers
  - ▶ Good starting-point when you want to introduce FP stuff into other languages
  - ▶ Good base when you what to write your own compiler
  - ▶ GHC uses an enriched Lambda Calculus internally

# Untyped Lambda Calculus

$t ::= x$

Variable

$\lambda x.t$

Abstraction

$t\ t$

Application

# Untyped Lambda Calculus

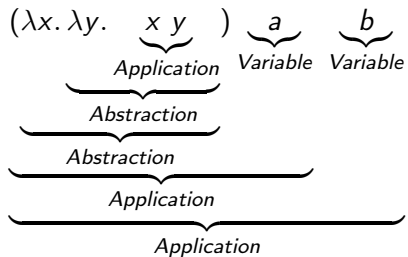
$t ::= x$	Variable
$\lambda x.t$	Abstraction
$t\ t$	Application

## Example

### ► Identity

$$\underbrace{\underbrace{\lambda x.x}_{\text{Abstraction}} \underbrace{y}_{\text{Variable}}}_{\text{Application}} \rightarrow y$$

# Evaluation / Reduction



# Evaluation / Reduction

$$\begin{array}{c} (\lambda x. \lambda y. \underbrace{x \ y}_{\text{Application}}) \underbrace{a}_{\text{Variable}} \underbrace{b}_{\text{Variable}} \\ \underbrace{\hspace{1.5cm}}_{\text{Abstraction}} \\ \underbrace{\hspace{1.5cm}}_{\text{Abstraction}} \\ \underbrace{\hspace{2.5cm}}_{\text{Application}} \\ \underbrace{\hspace{3.5cm}}_{\text{Application}} \end{array}$$

$$(\lambda \boxed{x}. \lambda y. \boxed{x} \ y) \boxed{a} \boxed{b}$$

# Evaluation / Reduction

$$\begin{array}{c} (\lambda x. \lambda y. \underbrace{x \ y}_{\text{Application}}) \underbrace{a}_{\text{Variable}} \underbrace{b}_{\text{Variable}} \\ \underbrace{\hspace{1.5cm}}_{\text{Abstraction}} \\ \underbrace{\hspace{1.5cm}}_{\text{Abstraction}} \\ \underbrace{\hspace{2.5cm}}_{\text{Application}} \\ \underbrace{\hspace{3.5cm}}_{\text{Application}} \end{array}$$

$$\begin{array}{l} (\lambda \boxed{x}. \lambda y. \boxed{x} \ y) \boxed{a} \boxed{b} \\ \rightarrow (\lambda \boxed{y}. \boxed{a} \ \boxed{y}) \boxed{b} \end{array}$$



# Evaluation / Reduction

$$\begin{array}{c} (\lambda x. \lambda y. \underbrace{x \ y}_{\text{Application}}) \underbrace{a}_{\text{Variable}} \underbrace{b}_{\text{Variable}} \\ \underbrace{\hspace{1.5cm}}_{\text{Abstraction}} \\ \underbrace{\hspace{1.5cm}}_{\text{Abstraction}} \\ \underbrace{\hspace{2.5cm}}_{\text{Application}} \\ \underbrace{\hspace{3.5cm}}_{\text{Application}} \end{array}$$

$$\begin{aligned} & (\lambda \textcolor{brown}{x}. \lambda y. \textcolor{brown}{x} \ y) \textcolor{brown}{a} \textcolor{brown}{b} \\ \rightarrow & (\lambda \textcolor{teal}{y}. a \ \textcolor{teal}{y}) \textcolor{teal}{b} \\ \rightarrow & a \ b \end{aligned}$$

# Evaluation / Reduction

$$\begin{array}{c} (\lambda x. \lambda y. \underbrace{x \ y}_{\text{Application}}) \underbrace{a}_{\text{Variable}} \underbrace{b}_{\text{Variable}} \\ \underbrace{\hspace{1.5cm}}_{\text{Abstraction}} \\ \underbrace{\hspace{1.5cm}}_{\text{Abstraction}} \\ \underbrace{\hspace{2.5cm}}_{\text{Application}} \\ \underbrace{\hspace{3.5cm}}_{\text{Application}} \end{array}$$

$$\begin{aligned} & (\lambda \boxed{x}. \lambda y. \boxed{x} \ y) \boxed{a} \boxed{b} \\ \rightarrow & (\lambda \boxed{y}. \boxed{a} \ \boxed{y}) \boxed{b} \\ \rightarrow & \boxed{a} \ \boxed{b} \end{aligned}$$

## ► Remarks

- Lambda Expressions expand as much to the right as possible
- We use parentheses to clarify what's meant
  - Even though I didn't specify them in the grammar ...

# Higher Order Functions

- ▶ Functions that take or return functions
  - ▶ Are there "by definition"

$$\underbrace{\underbrace{\lambda x.x}_{\text{Abstraction}} \underbrace{\lambda y.y}_{\text{Abstraction}}}_{\text{Application}} \rightarrow \underbrace{\lambda y.y}_{\text{Abstraction}}$$

# Currying

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

- ▶ Example

- ▶ (+1) Section in Haskell

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

- ▶ Partial Application is there "by definition"

# Remarks

- ▶ Everything (Term) is an Expression
  - ▶ No statements
- ▶ No "destructive" Variable Assignments
  - ▶ The reason why FP Languages promote pure functions

# Some Vocabulary

$\lambda x.(x\ y)$

- ▶  $x$  is *bound* by the surrounding abstraction
- ▶  $y$  is *free*
  - ▶ E.g. part of the environment

# Reductions and Conversions

- ▶ Alpha conversion

$$\lambda x.x \rightarrow_{\alpha} \lambda y.y$$

# Reductions and Conversions

- ▶ Alpha conversion

$$\lambda x.x \rightarrow_{\alpha} \lambda y.y$$

- ▶ Beta reduction

$$(\lambda x.x)y \rightarrow_{\beta} y$$



# Reductions and Conversions

- ▶ Alpha conversion

$$\lambda x.x \rightarrow_{\alpha} \lambda y.y$$

- ▶ Beta reduction

$$(\lambda x.x)y \rightarrow_{\beta} y$$

- ▶ Eta conversion

- ▶ iff (if and only if)  $x$  is not free in  $f$

$$(\lambda x.f\ x) \rightarrow_{\eta} f$$

$$(\lambda x.(\lambda y.y)\ x) \rightarrow_{\eta} \lambda y.y$$

- ▶  $x$  is not free in  $f$

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

# Church Encodings

- ▶ Encode Data into the Lambda Calculus
- ▶ To simplify our formulas, let's say that we have declarations

$$id \equiv \lambda x.x$$

$$id\ y \rightarrow y$$

# Booleans

$$true \equiv \lambda t. \lambda f. t$$

$$false \equiv \lambda t. \lambda f. f$$

$$if\_then\_else \equiv \lambda c. \lambda b_{true}. \lambda b_{false}. c \ b_{true} \ b_{false}$$

## Example

$$\begin{aligned} & if\_then\_else \ true \ a \ b \\ \equiv & (\lambda c. \lambda b_{true}. \lambda b_{false}. c \ b_{true} \ b_{false}) \ true \ a \ b \\ \rightarrow & true \ a \ b \\ \equiv & (\lambda t. \lambda f. t) \ a \ b \\ \rightarrow & (\lambda f. a) \ b \\ \rightarrow & a \end{aligned}$$

# And

$$\text{true} \equiv \lambda t. \lambda f. t$$
$$\text{false} \equiv \lambda t. \lambda f. f$$
$$\text{and} \equiv \lambda p. \lambda q. p \ q \ p$$

## ► Example

$$\text{and true false}$$
$$\equiv (\lambda p. \lambda q. p \ q \ p) \ \text{true} \ \text{false}$$
$$\rightarrow (\lambda q. \text{true} \ q \ \text{true}) \ \text{false}$$
$$\rightarrow \text{true false true}$$
$$\equiv (\lambda t. \lambda f. t) \ \text{false} \ \text{true}$$
$$\rightarrow (\lambda f. \text{false}) \ \text{true}$$
$$\rightarrow \text{false}$$

Or

$\lambda p. \lambda q. p p q$

# Pairs

$$pair \equiv \lambda x. \lambda y. \lambda z. z \ x \ y$$

$$first \equiv (\lambda p. p)(\lambda x. \lambda y. x)$$

$$second \equiv (\lambda p. p)(\lambda x. \lambda y. y)$$

## Example

$$\begin{aligned} pair_{AB} &\equiv pair && a \ b \\ &\equiv && (\lambda x. \lambda y. \lambda z. z \ x \ y) \ a \ b \\ &\rightarrow && (\lambda y. \lambda z. z \ a \ y) b \\ &\rightarrow && \lambda z. z \ a \ b \\ &\equiv && pair'_{ab} \end{aligned}$$

## Pair Example (continued)

$$\begin{aligned} pair'_{ab} &\equiv \lambda z. z \ a \ b \\ first &\equiv (\lambda p. p)(\lambda x. \lambda y. x) \end{aligned}$$

$$\begin{aligned} first \ pair'_{ab} &\equiv (\lambda p. p)(\lambda x. \lambda y. x) pair'_{ab} \\ &\rightarrow pair'_{ab}(\lambda x. \lambda y. x) \\ &\equiv (\lambda z. z \ a \ b)(\lambda x. \lambda y. x) \\ &\rightarrow (\lambda x. \lambda y. x) \ a \ b \\ &\rightarrow (\lambda y. a) \ b \\ &\rightarrow a \end{aligned}$$

# Numerals

- ▶ Peano axioms
  - ▶ Every natural number can be defined with 0 and a successor function

$$0 \equiv \lambda f. \lambda x. x$$

$$1 \equiv \lambda f. \lambda x. f \ x$$

$$2 \equiv \lambda f. \lambda x. f \ (f \ x)$$

$$3 \equiv \lambda f. \lambda x. f \ (f \ (f \ x))$$

- ▶ Meaning

0  $f$  is evaluated 0 times

1  $f$  is evaluated once

$x$  can be every lambda term



# Numerals Example - Successor

$$0 \equiv \lambda f. \lambda x. x$$

$$1 \equiv \lambda f. \lambda x. f \ x$$

$$\text{successor} \equiv \lambda n. \lambda f. \lambda x. f \ (n \ f \ x)$$

$$\text{successor} 1 \equiv (\lambda n. \lambda f. \lambda x. f \ (n \ f \ x)) 1$$

$$\rightarrow \lambda f. \lambda x. f \ (1 \ f \ x)$$

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

$$\text{to} \lambda f. \lambda x. f \ ((\lambda x. f \ x) \ x)$$

$$\text{to} \lambda f. \lambda x. f \ (f \ x)$$

$$\equiv 2$$

# Numerals Example - $0 + 0$

$0 \equiv \lambda f. \lambda x. x$

$plus \equiv \lambda m. \lambda n. \lambda f. \lambda x. mf(nfx)$

$plus\ 0\ 0 \equiv (\lambda m. \lambda n. \lambda f. \lambda x. mf(nfx))\ 0\ 0$

$\rightarrow (\lambda n. \lambda f. \lambda x. 0f(nfx))\ 0$

$\rightarrow (\lambda f. \lambda x. 0f(0fx))$

$\equiv (\lambda f. \lambda x. (\lambda f. \lambda x. x)f(0fx))$

$\rightarrow (\lambda f. \lambda x. (\lambda x. x)(0fx))$

$\rightarrow (\lambda f. \lambda x. (0fx))$

$\equiv (\lambda f. \lambda x. ((\lambda f. \lambda x. x)fx))$

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

$\rightarrow (\lambda f. \lambda x. x$

$\equiv 0$

The implementation of programming languages Type Systems

# Thanks

- ▶ Hope you enjoyed this talk and learned something new.
- ▶ Hope it wasn't too much math and dusty formulas ... :)