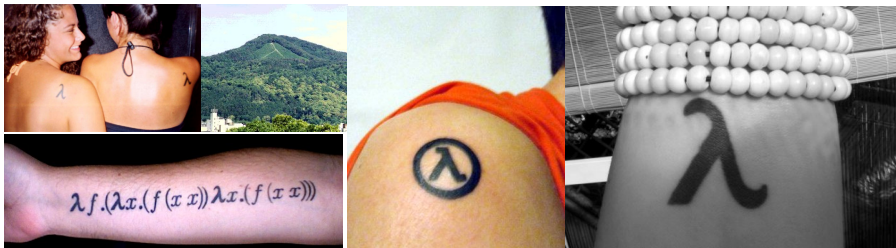


# The Lambda Calculus

Stephen A. Edwards

Columbia University

Fall 2010



# Lambda Expressions

Function application written in prefix form. “Add four and five” is

$$(+\ 4\ 5)$$

Evaluation: select a *redex* and evaluate it:

$$\begin{aligned} (+\ (*\ 5\ 6)\ (*\ 8\ 3)) &\rightarrow (+\ 30\ (*\ 8\ 3)) \\ &\rightarrow (+\ 30\ 24) \\ &\rightarrow 54 \end{aligned}$$

Often more than one way to proceed:

$$\begin{aligned} (+\ (*\ 5\ 6)\ (*\ 8\ 3)) &\rightarrow (+\ (*\ 5\ 6)\ 24) \\ &\rightarrow (+\ 30\ 24) \\ &\rightarrow 54 \end{aligned}$$

# Function Application and Currying



Function application is written as juxtaposition:

$$f x$$

Every function has exactly one argument. Multiple-argument functions, e.g.,  $+$ , are represented by *currying*, named after Haskell Brooks Curry (1900–1982). So,

$$(+ x)$$

is the function that adds  $x$  to its argument.

Function application associates left-to-right:

$$\begin{aligned} (+ 3 4) &= ((+ 3) 4) \\ &\rightarrow 7 \end{aligned}$$

# Lambda Abstraction

The only other thing in the lambda calculus is *lambda abstraction*: a notation for defining unnamed functions.

$(\lambda x. + x 1)$

(             $\lambda$              $x$     .    +    $x$     1 )  
                   $\uparrow$              $\uparrow$   $\uparrow$      $\uparrow$   $\uparrow$      $\uparrow$

That function of  $x$  that adds  $x$  to 1

# The Syntax of the Lambda Calculus

$$\begin{array}{l} \textit{expr} ::= \textit{expr expr} \\ \quad | \lambda \textit{variable} . \textit{expr} \\ \quad | \textit{constant} \\ \quad | \textit{variable} \\ \quad | (\textit{expr}) \end{array}$$

Constants are numbers and built-in functions;  
variables are identifiers.

## Beta-Reduction

Evaluation of a lambda abstraction—*beta-reduction*—is just substitution:

$$\begin{aligned}(\lambda x. + x 1) 4 &\rightarrow (+ 4 1) \\ &\rightarrow 5\end{aligned}$$

The argument may appear more than once

$$\begin{aligned}(\lambda x. + x x) 4 &\rightarrow (+ 4 4) \\ &\rightarrow 8\end{aligned}$$

or not at all

$$(\lambda x. 3) 5 \rightarrow 3$$

# Beta-Reduction

Fussy, but mechanical. Extra parentheses may help.

$$\begin{aligned}(\lambda x. \lambda y. + x y) 3 4 &= \left( \left( \lambda x. \left( \lambda y. ((+ x) y) \right) \right) 3 \right) 4 \\ &\rightarrow \left( \lambda y. ((+ 3) y) \right) 4 \\ &\rightarrow ((+ 3) 4) \\ &\rightarrow 7\end{aligned}$$

Functions may be arguments

$$\begin{aligned}(\lambda f. f 3) (\lambda x. + x 1) &\rightarrow (\lambda x. + x 1) 3 \\ &\rightarrow (+ 3 1) \\ &\rightarrow 4\end{aligned}$$

# Free and Bound Variables

$$(\lambda x. + x y) 4$$

Here,  $x$  is like a function argument but  $y$  is like a global variable.

Technically,  $x$  *occurs bound* and  $y$  *occurs free* in

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

However, both  $x$  and  $y$  occur free in

$$(+ x y)$$



## Beta-Reduction More Formally

$$(\lambda x . E) F \rightarrow_{\beta} E'$$

where  $E'$  is obtained from  $E$  by replacing every instance of  $x$  that appears free in  $E$  with  $F$ .

The definition of free and bound mean variables have scopes. Only the rightmost  $x$  appears free in

$$(\lambda x . + (- x 1)) x 3$$

so

$$\begin{aligned} (\lambda x . (\lambda x . + (- x 1)) x 3) 9 &\rightarrow (\lambda x . + (- x 1)) 9 3 \\ &\rightarrow + (- 9 1) 3 \\ &\rightarrow + 8 3 \\ &\rightarrow 11 \end{aligned}$$

## Another Example

$$\begin{aligned} (\lambda x. \lambda y. + x ((\lambda x. - x 3) y)) 5 6 &\rightarrow (\lambda y. + 5 ((\lambda x. - x 3) y)) 6 \\ &\rightarrow + 5 ((\lambda x. - x 3) 6) \\ &\rightarrow + 5 (- 6 3) \\ &\rightarrow + 5 3 \\ &\rightarrow 8 \end{aligned}$$

# Alpha-Conversion

One way to confuse yourself less is to do  $\alpha$ -conversion: renaming a  $\lambda$  argument and its bound variables.

Formal parameters are only names: they are correct if they are consistent.

$$\begin{aligned}(\lambda x. (\lambda x. + (- x 1)) x 3) 9 &\leftrightarrow (\lambda x. (\lambda y. + (- y 1)) x 3) 9 \\ &\rightarrow ((\lambda y. + (- y 1)) 9 3) \\ &\rightarrow (+ (- 9 1) 3) \\ &\rightarrow (+ 8 3) \\ &\rightarrow 11\end{aligned}$$

## Beta-Abstraction and Eta-Conversion

Running  $\beta$ -reduction in reverse, leaving the “meaning” of a lambda expression unchanged, is called *beta abstraction*:

$$+ 4 1 \leftarrow (\lambda x . + x 1) 4$$

Eta-conversion is another type of conversion that leaves “meaning” unchanged:

$$(\lambda x . + 1 x) \leftrightarrow_{\eta} (+ 1)$$

Formally, if  $F$  is a function in which  $x$  does not occur free,

$$(\lambda x . F x) \leftrightarrow_{\eta} F$$

## Reduction Order

The order in which you reduce things can matter.

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

Two things can be reduced:

$$(\lambda z . z z) (\lambda z . z z)$$

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

However,

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

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

## Normal Form

A lambda expression that cannot be  $\beta$ -reduced is in *normal form*.  
Thus,

$$\lambda y . y$$

is the normal form of

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

Not everything has a normal form. E.g.,

$$(\lambda z . z z) (\lambda z . z z)$$

can only be reduced to itself, so it never produces an non-reducible expression.

## Normal Form

Can a lambda expression have more than one normal form?

**Church-Rosser Theorem I:** If  $E_1 \leftrightarrow E_2$ , then there exists an expression  $E$  such that  $E_1 \rightarrow E$  and  $E_2 \rightarrow E$ .

**Corollary.** No expression may have two distinct normal forms.

*Proof.* Assume  $E_1$  and  $E_2$  are distinct normal forms for  $E$ :  $E \leftrightarrow E_1$  and  $E \leftrightarrow E_2$ . So  $E_1 \leftrightarrow E_2$  and by the Church-Rosser Theorem I, there must exist an  $F$  such that  $E_1 \rightarrow F$  and  $E_2 \rightarrow F$ . However, since  $E_1$  and  $E_2$  are in normal form,  $E_1 = F = E_2$ , a contradiction.

# Normal-Order Reduction

Not all expressions have normal forms, but is there a reliable way to find the normal form if it exists?

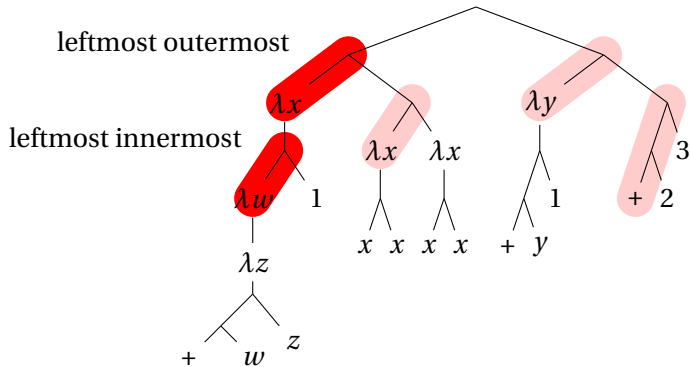
**Church-Rosser Theorem II:** If  $E_1 \rightarrow E_2$  and  $E_2$  is in normal form, then there exists a *normal order* reduction sequence from  $E_1$  to  $E_2$ .

*Normal order reduction:* reduce the leftmost outermost redex.



# Normal-Order Reduction

$$\left( \left( \lambda x. \left( \left( \lambda w. \lambda z. + w z \right) 1 \right) \right) \left( \left( \lambda x. x x \right) \left( \lambda x. x x \right) \right) \right) \left( \left( \lambda y. + y 1 \right) \left( + 2 3 \right) \right)$$



## Recursion

Where is recursion in the lambda calculus?

$$FAC = \left( \lambda n . IF (= n 0) 1 \left( * n (FAC (- n 1)) \right) \right)$$

This does not work: functions are unnamed in the lambda calculus.  
But it is possible to express recursion *as a function*.

$$\begin{aligned} FAC &= (\lambda n . \dots FAC \dots) \\ &\leftarrow_{\beta} (\lambda f . (\lambda n . \dots f \dots)) FAC \\ &= H FAC \end{aligned}$$

That is, the factorial function,  $FAC$ , is a *fixed point* of the (non-recursive) function  $H$ :

$$H = \lambda f . \lambda n . IF (= n 0) 1 (* n (f (- n 1)))$$

## Recursion

Let's invent a function  $Y$  that computes  $FAC$  from  $H$ , i.e.,  $FAC = Y H$ :

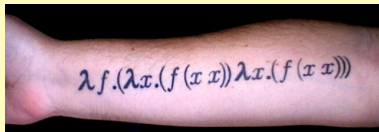
$$\begin{aligned} FAC &= H FAC \\ Y H &= H (Y H) \end{aligned}$$

$$\begin{aligned} FAC\ 1 &= Y\ H\ 1 \\ &= H\ (Y\ H)\ 1 \\ &= (\lambda f.\ \lambda n.\ IF\ (= n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))\ (Y\ H)\ 1 \\ &\rightarrow (\lambda n.\ IF\ (= n\ 0)\ 1\ (*\ n\ ((Y\ H)\ (-\ n\ 1))))\ 1 \\ &\rightarrow IF\ (= 1\ 0)\ 1\ (*\ 1\ ((Y\ H)\ (-\ 1\ 1))) \\ &\rightarrow * 1\ (Y\ H\ 0) \\ &= * 1\ (H\ (Y\ H)\ 0) \\ &= * 1\ ((\lambda f.\ \lambda n.\ IF\ (= n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))\ (Y\ H)\ 0) \\ &\rightarrow * 1\ ((\lambda n.\ IF\ (= n\ 0)\ 1\ (*\ n\ (Y\ H\ (-\ n\ 1))))\ 0) \\ &\rightarrow * 1\ (IF\ (= 0\ 0)\ 1\ (*\ 0\ (Y\ H\ (-\ 0\ 1)))) \\ &\rightarrow * 1\ 1 \\ &\rightarrow 1 \end{aligned}$$

# The Y Combinator

Here's the eye-popping part:  $Y$  can be a simple lambda expression.

$Y =$



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

$$\begin{aligned} Y H &= \left( \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \right) H \\ &\rightarrow (\lambda x. H (x x)) (\lambda x. H (x x)) \\ &\rightarrow H \left( (\lambda x. H (x x)) (\lambda x. H (x x)) \right) \\ &\leftrightarrow H \left( \left( \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \right) H \right) \\ &= H (Y H) \end{aligned}$$

“Y: The function that takes a function  $f$  and returns  $f(f(f(f(\dots))))$ ”

# Alonzo Church



1903–1995

Professor at Princeton (1929–1967)  
and UCLA (1967–1990)

Invented the Lambda Calculus

Had a few successful graduate students, including

- ▶ Stephen Kleene (Regular expressions)
- ▶ Michael O. Rabin<sup>†</sup> (Nondeterministic automata)
- ▶ Dana Scott<sup>†</sup> (Formal programming language semantics)
- ▶ Alan Turing (Turing machines)

<sup>†</sup> Turing award winners

# Turing Machines vs. Lambda Calculus



In 1936,

- ▶ Alan Turing invented the Turing machine
- ▶ Alonzo Church invented the lambda calculus

In 1937, Turing proved that the two models were equivalent, i.e., that they define the same class of computable functions.

Modern processors are just overblown Turing machines.

Functional languages are just the lambda calculus with a more palatable syntax.