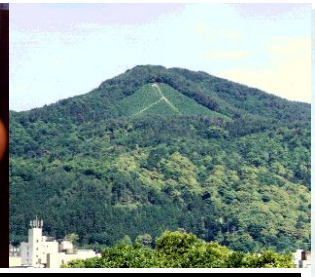
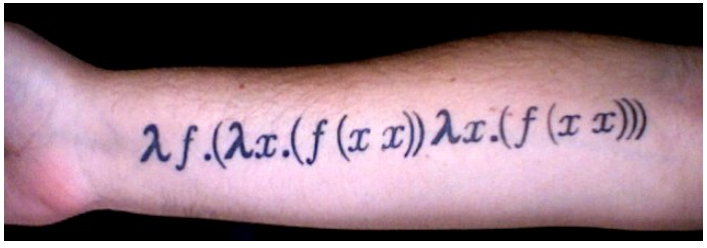
The Lambda Calculus

Stephen A. Edwards

Columbia University

Fall 2014





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:

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

Simon Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

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.

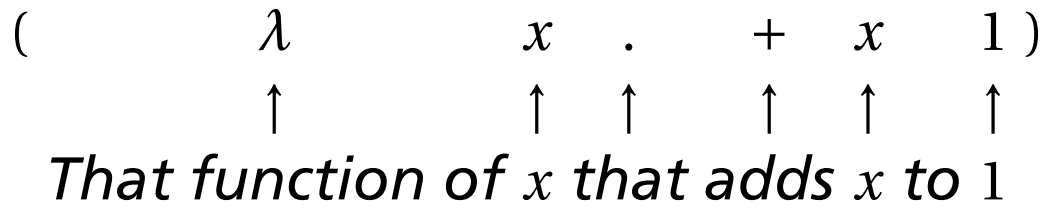
Function application associates left-to-right:

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

Lambda Abstraction

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

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



The Syntax of the Lambda Calculus

expr	::=	*expr expr*
	|	λ *variable* . *expr*
	|	*constant*
	|	*variable*
	|	(*expr*)

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

Function application binds more tightly than λ:

$$\lambda x.f\,g\,x = \left(\lambda x.(f\,g)\,x\right)$$

Beta-Reduction

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

$$(\lambda x\ .\ +\ x\ 1)\ 4 \rightarrow (+\ 4\ 1) \rightarrow 5$$

The argument may appear more than once

$$(\lambda x\ .\ +\ x\ x)\ 4 \rightarrow (+\ 4\ 4) \rightarrow 8$$

or not at all

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

Beta-Reduction

Fussy, but mechanical. Extra parentheses may help.

$$
\begin{aligned}
(\lambda x\,.\,\lambda y\,.\,+\;x\;y)\;3\;4 &= \Big(\big(\lambda x\,.\,\big(\lambda y\,.\,\big((+\;x)\;y\big)\big)\big)\;3\Big)\;4 \\
&\to \Big(\quad \lambda y\,.\,\big((+\;3)\;y\big) \quad\Big)\;4 \\
&\to \big((+\;3)\;4\big) \\
&\to 7
\end{aligned}
$$

Functions may be arguments

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

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

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

Alpha-Conversion

One way to confuse yourself less is to do α-conversion: renaming a λ argument and its bound variables.

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

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

You've probably done this before in C or Java:

```
int add(int x, int y)
{
  return x + y;
}
```

$\leftrightarrow$

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

Beta-Abstraction and Eta-Conversion

Running β-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,

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

```
int f(int y) { ... }
int g(int x) {  return f(x);  }

g(w); ← can be replaced with f(w)
```

Reduction Order

The order in which you reduce things can matter.

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

Two things can be reduced:

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

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

However,

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

$$(\lambda x\,.\,\lambda y\,.\,y)\ (\cdots) \rightarrow\ (\lambda y\,.\,y)$$

Normal Form

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

$$\lambda y \,.\, y$$

is the normal form of

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

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.

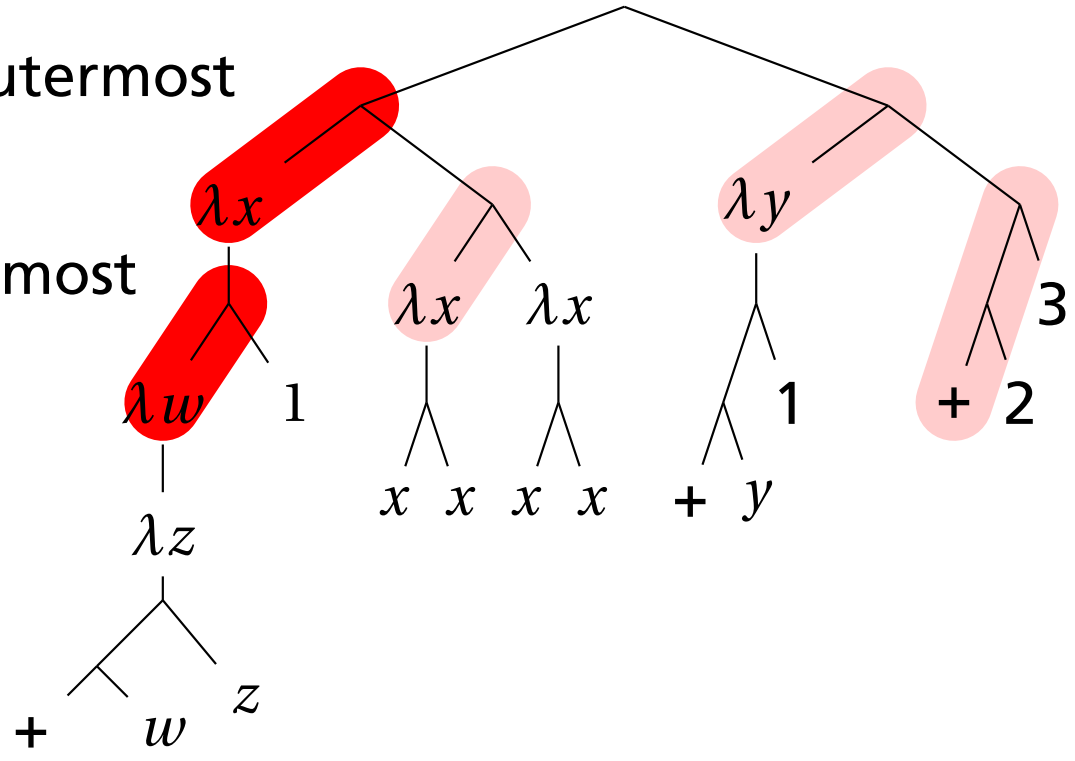
Normal order reduction: reduce the leftmost outermost redex.

Normal-Order Reduction

$$\Big(\big(\lambda x\,.\,\big((\lambda w\,.\,\lambda z\,.\,+\;w\;z)\;1\big)\big)\;\big((\lambda x\,.\,x\;x)\;(\lambda x\,.\,x\;x)\big)\Big)\;\big((\lambda y\,.\,+\;y\;1)\;(+\;2\;3)\big)$$

leftmost outermost

leftmost innermost



Boolean Logic in the Lambda Calculus

"Church Booleans"

$$\begin{aligned} \text{true} &= \lambda x\,.\,\lambda y\,.\,x \\ \text{false} &= \lambda x\,.\,\lambda y\,.\,y \end{aligned}$$

Each is a function of two arguments: true is "select first;" false is "select second." If-then-else uses its predicate to select *then* or *else*:

$$\text{ifelse} = \lambda p\,.\,\lambda a\,.\,\lambda b\,.\,p\ a\ b$$

E.g.,

$$\begin{aligned} \text{ifelse true } 42\ 58 &= \text{true } 42\ 58 \\ &\rightarrow (\lambda x\,.\,\lambda y\,.\,x)\ 42\ 58 \\ &\rightarrow (\lambda y\,.\,42)\ 58 \\ &\rightarrow 42 \end{aligned}$$

Boolean Logic in the Lambda Calculus

Logic operators can be expressed with if-then-else:

$$\begin{aligned} \text{and} &= \lambda p\,.\,\lambda q\,.\,p\ q\ p \\ \text{or} &= \lambda p\,.\,\lambda q\,.\,p\ p\ q \\ \text{not} &= \lambda p\,.\,\lambda a\,.\,\lambda b\,.\,p\ b\ a \end{aligned}$$

$$\begin{aligned} \text{and true false} &= (\lambda p\,.\,\lambda q\,.\,p\ q\ p)\ \text{true false} \\ &\rightarrow \text{true false true} \\ &\rightarrow (\lambda x\,.\,\lambda y\,.\,x)\ \text{false true} \\ &\rightarrow \text{false} \end{aligned}$$

$$\begin{aligned} \text{not true} &= (\lambda p\,.\,\lambda a\,.\,\lambda b\,.\,p\ b\ a)\ \text{true} \\ &\rightarrow_\beta \lambda a\,.\,\lambda b\,.\,\text{true}\ b\ a \\ &\rightarrow_\beta \lambda a\,.\,\lambda b\,.\,b \\ &\rightarrow_\alpha \lambda x\,.\,\lambda y\,.\,y \\ &= \text{false} \end{aligned}$$

Arithmetic: The Church Numerals

$$
\begin{aligned}
0 &= \lambda f\,.\,\lambda x\,.\,x \\
1 &= \lambda f\,.\,\lambda x\,.\,f\ x \\
2 &= \lambda f\,.\,\lambda x\,.\,f\ (f\ x) \\
3 &= \lambda f\,.\,\lambda x\,.\,f\ \big(f\ (f\ x)\big)
\end{aligned}
$$

I.e., for $n = 0, 1, 2, \ldots$, $n f x = f^{(n)}(x)$. The successor function:

$$
\mathsf{succ} = \lambda n\,.\,\lambda f\,.\,\lambda x\,.\,f\ (n\ f\ x)
$$

$$
\begin{aligned}
\mathsf{succ}\ 2 &= \big(\lambda n\,.\,\lambda f\,.\,\lambda x\,.\,f\ (n\ f\ x)\big)\ 2 \\
&\to \lambda f\,.\,\lambda x\,.\,f\ (2\ f\ x) \\
&= \lambda f\,.\,\lambda x\,.\,f\ \Big(\big(\lambda f\,.\,\lambda x\,.\,f\ (f\ x)\big)\ f\ x\Big) \\
&\to \lambda f\,.\,\lambda x\,.\,f\ \big(f\ (f\ x)\big) \\
&= 3
\end{aligned}
$$

Adding Church Numerals

Finally, we can add:

$$\text{plus} = \lambda m.\lambda n.\lambda f.\lambda x.\ m\ f\ (\ n\ f\ x)$$

$$\begin{aligned}
\text{plus } 3\ 2 &= \big(\lambda m.\lambda n.\lambda f.\lambda x.\ m\ f\ (\ n\ f\ x)\big)\ 3\ 2 \\
&\rightarrow \lambda f.\lambda x.\ 3\ f\ (\ 2\ f\ x) \\
&\rightarrow \lambda f.\lambda x.\ f\ (f\ (f\ (2\ f\ x))) \\
&\rightarrow \lambda f.\lambda x.\ f\ (f\ (f\ (f\ (\ f\ x)))) \\
&= 5
\end{aligned}$$

Not surprising since $f^{(m)} \circ f^{(n)} = f^{(m+n)}$

Multiplying Church Numerals

$$\text{mult} = \lambda m.\lambda n.\lambda f.m\ (n\ f)$$

$$\begin{aligned}
\text{mult}\ 2\ 3 &= \big(\lambda m.\lambda n.\lambda f.m\ (n\ f)\big)\ 2\ 3 \\
&\to \lambda f.2\ (3\ f) \\
&= \lambda f.\ 2\ (\lambda x.\ f(f(f\ x))) \\
&\leftrightarrow_\alpha \lambda f.\ 2\ (\lambda y.\ f(f(f\ y))) \\
&\to \lambda f.\ \lambda x.\ (\lambda y.\ f(f(f\ y)))\ ((\lambda y.\ f(f(f\ y)))\ x) \\
&\to \lambda f.\ \lambda x.\ (\lambda y.\ f(f(f\ y)))\ (f(f(f\ x))) \\
&\to \lambda f.\ \lambda x.\qquad f(f(f\ (f(f(f\ x)))\ )) \\
&= 6
\end{aligned}$$

The *predecessor* function is trickier since there aren't negative numbers.

Recursion

Where is recursion in the lambda calculus?

$$\mathsf{fac} = \left(\lambda n \,.\, \mathsf{if}\ (=\ n\ 0)\ 1\ \left(*\ n\ \left(\mathsf{fac}\ (-\ n\ 1)\right)\right)\right)$$

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

$$\begin{aligned} \mathsf{fac} &= (\lambda n \,.\, \ldots\ \mathsf{fac}\ \ldots) \\ &\leftarrow_\beta (\lambda f \,.\, (\lambda n \,.\, \ldots\ f\ \ldots))\ \mathsf{fac} \\ &= H\ \mathsf{fac} \end{aligned}$$

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

$$H = \lambda f \,.\, \lambda n \,.\, \mathsf{if}\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))$$

Recursion

Let's invent a Y that computes fac from H, i.e., $\mathsf{fac} = Y\ H$:

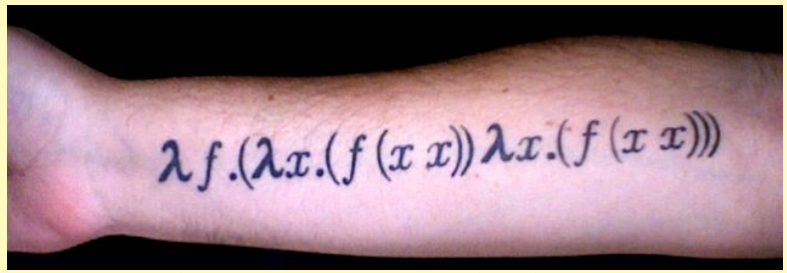
$$\begin{aligned} \mathsf{fac} &= H\ \mathsf{fac} \\ Y\ H &= H\ (Y\ H) \end{aligned}$$

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

The Y Combinator

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

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



$$= \lambda f\,.\,\left(\lambda x\,.\,f\,(x\,x)\right)\,\left(\lambda x\,.\,f\,(x\,x)\right)$$

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

"Y: The function that takes a function f and returns $f(f(f(f(\cdots))))$"

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† (Nondeterministic automata)
- Dana Scott† (Formal programming language semantics)
- Alan Turing (Turing machines)

† 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.