

# Lambda Calculus

Nagender Parimi  
Department of Computer Science,  
Rensselaer Polytechnic Institute

[parimi@cs.rpi.edu](mailto:parimi@cs.rpi.edu)



---

---

---

---

---

---

---

---

## Lambda Calculus - Introduction

- Lambda Calculus is a language itself, based on the functional model
- Syntax is very simple and has only three constructs:
  - $M$  is a *term*
  - $M ::= x \mid (M M) \mid (\lambda x. M)$
- $\lambda x. M$  is a notation denoting an anonymous function with parameter  $x$  and body  $M$



---

---

---

---

---

---

---

---

## Introduction

- $\lambda$  in  $(\lambda x. M)$  is similar to the use of  $\$$  as a placeholder in Oz
  - `proc { $ x } M end` is an anonymous function
- $M ::= x \mid (M N) \mid (\lambda x. M)$ 
  - $x$  is a *variable*
  - $(M N)$  is the *application* of function  $M$  to  $N$
  - $(\lambda x. M)$  is referred to as an *abstraction*



---

---

---

---

---

---

---

---

## Syntactic Conventions



- In the absence of parenthesis, function application groups from left to right
  - $x\ y\ z$  abbreviates  $((x\ y)\ z)$
  - $x\ (y\ z)$  is application of  $x$  to  $(y\ z)$
- Function application has higher precedence than abstraction
  - $\lambda x. x\ z$  abbreviates  $(\lambda x. (x\ z))$
- $\lambda x. \lambda y. M$  can be written as  $\lambda xy. M$

---

---

---

---

---

---

---

---

## Free and Bound Variables



- Variable  $x$  is said to be bound in  $\lambda x. M$  else it is free
- The occurrence of  $x$  in  $\lambda x. M$  is called its *binding occurrence*, or a *binding* of  $x$
- The *scope* of this binding is  $M$ ; all occurrences of  $x$  are bound within this scope

---

---

---

---

---

---

---

---

## Free and Bound Variables



- Let  $\text{free}(M)$  denote set of free variables in a term  $M$ 
  - $\text{free}(x) = \{x\}$
  - $\text{free}(M\ N) = \text{free}(M) \cup \text{free}(N)$
  - $\text{free}(\lambda x. M) = \text{free}(M) - \{x\}$
- Which variables are free in  $(\lambda x. y)$   $(\lambda y. y)$  ?

---

---

---

---

---

---

---

---

## Free and Bound Variables



- Which variables are free in

$$T = (\lambda x. y) (\lambda y. y) ?$$

- Recall:

$$\text{free}(M N) = \text{free}(M) \cup \text{free}(N)$$

$$\text{free}(\lambda x. M) = \text{free}(M) - \{x\}$$

$$\text{free}(x) = \{x\}$$

---

---

---

---

---

---

---

---

## Free and Bound Variables



- Which variables are free in

$$T = (\lambda x. y) (\lambda y. y) ?$$

- $\text{free}(\lambda x. y) = \{y\}$
- $\text{free}(\lambda y. y) = \emptyset$
- $\text{free}(T) = \{y\} \cup \emptyset = \{y\}$

---

---

---

---

---

---

---

---

## Substitution



- Substitution is a tool employed in *reduction* (explained later)
- It is denoted as  $\{N/x\}M$   
N is substituted for x in M
- The bigger picture: function execution
  - The result of applying an abstraction  $\lambda x. M$  to an argument N,  
 $(\lambda x. M) N$   
is formalized by “substituting” N for x in M

---

---

---

---

---

---

---

---

## Substitution: $\{N/x\} M$



- An informal definition:
  1. If the free variables of  $N$  have no bound occurrences in  $M$ , then  $\{N/x\}M$  is formed by replacing all free occurrences of  $x$  in  $M$  by  $N$ ;
  2. else the bound variables in  $M$  are renamed to fresh variables until the above rule applies. This renaming is done to prevent a *name clash* i.e. when a free variable in  $N$  is bound in  $M$
- Substitution has the effect of replacing only **free** occurrences of  $x$  in  $M$

---

---

---

---

---

---

---

---

## Substitution: $\{N/x\} M$



- Examples
  - $\{u/x\} x = u$  (rule 1)
  - $\{u/x\} (x y) = (u y)$  (rule 1)
- $M$  has no free occurrences in the following examples
  - $\{u/x\} y = y$
  - $\{u/x\} (\lambda x. x) = (\lambda x. x)$
- Rule 2 applies here:
  - $\{u/x\} (\lambda u. xu) = \{u/x\} (\lambda z. xz) = (\lambda z. uz)$

---

---

---

---

---

---

---

---

## Computation in Lambda Calculus



- Computation is achieved by a sequence of reductions
- A term is “reduced” into as simple a form as possible
- Reductions take place through systematic application of defined rules: the  $\alpha$  and  $\beta$  reductions

---

---

---

---

---

---

---

---

## β Reductions



- $(\lambda x.M) N \Rightarrow_{\beta} \{N/x\} M$   
is called a **β reduction**
- A term of the form  $(\lambda x.M) N$  is called a *redex*, short for “reduction expression”
- A term that cannot be β-reduced further is said to be in β-normal form, or simply **normal form**
- e.g.  $(\lambda x.x)$  is in normal form

---

---

---

---

---

---

---

---

## β Reductions



- Let us try to reduce  
 $T = (\lambda x.xx) (\lambda x.xx)$
- It is of the form  $(\lambda x.M) N$   
with  $M=xx$ ,  $N=(\lambda x.xx)$

---

---

---

---

---

---

---

---

## β Reduction:

$$(\lambda x.M) N \Rightarrow_{\beta} \{N/x\} M$$



- $T = (\lambda x.xx) (\lambda x.xx)$
- For clarity, we rewrite  
 $T = (\lambda y.yy) (\lambda x.xx)$
- $T \Rightarrow_{\beta} \{(\lambda x.xx)/y\} yy$   
 $\Rightarrow_{\beta} (\lambda x.xx) (\lambda x.xx)$
- We are back where we started!  
**Corollary:** reductions can be non-terminating

---

---

---

---

---

---

---

---

## $\alpha$ -Conversions



- The renaming step used previously is formalized as an  $\alpha$ -conversion:  
 $\lambda x.M \Rightarrow_{\alpha} \lambda y.\{y/x\}M; y \notin \text{free}(M)$
- $\lambda x.x$  and  $\lambda y.y$  are the same function
- A *reduction sequence* is defined as any sequence of  $\beta$ -reductions and  $\alpha$ -conversions

---

---

---

---

---

---

---

---

## Reduction Sequences



- We write  $P \Rightarrow Q$  if  $P \Rightarrow_{\beta} Q$  or  $P \Rightarrow_{\alpha} Q$ .
- $P \Rightarrow^* Q$  if there exist terms  $P_0, P_1 \dots P_n$  for some  $n \geq 0$ ,  
 $P = P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_n = Q$

---

---

---

---

---

---

---

---

## Church-Rosser Theorem



- For terms  $M, P$  &  $Q$ :  
If  $M \Rightarrow^* P$  and  $M \Rightarrow^* Q$ , then there must exist a term  $R$  such that  $P \Rightarrow^* R$  and  $Q \Rightarrow^* R$
- Thus, *result of a computation does not depend on the order in which reductions are applied*

---

---

---

---

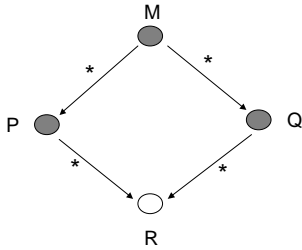
---

---

---

---

## Church-Rosser Theorem



If M reduces to P and Q then both can reach some common R

---

---

---

---

---

---

---

---

## Reduction Strategy & Parameter Passing



- Given a term, there may be several redex choices for reduction. How do we decide which one to pick first?
- In other words, how do function applications get evaluated? And in what order?
- Concept is similar to that in parameter passing strategies

---

---

---

---

---

---

---

---

## Recap: Parameter Passing



- Call by value: argument is evaluated before entering procedure body
- Call by name: the procedure is given a function for each passed parameter; the function is evaluated each time parameter is required

---

---

---

---

---

---

---

---

## Reduction Strategies



- Call by name strategy
  - Also called *normal order reduction*
  - Chooses the **leftmost-outermost** redex in a term: among the outermost redexes, pick the leftmost one
  - It is guaranteed to reach a normal form, if one exists

---

---

---

---

---

---

---

---

## Reduction Strategies



- Call by name strategy
- Example  
 $(\lambda x. (\lambda y. y) x) z$
- Outermost redex:  
 $(\lambda x. (\lambda y. y) x) z$
- Call by name evaluates enclosed function(s) to obtain argument values

---

---

---

---

---

---

---

---

## Reduction Strategies



- Call by value strategy
  - Picks the **leftmost innermost** redex
- Example: innermost redex  
 $(\lambda x. (\lambda y. y) x) z$

---

---

---

---

---

---

---

---

## Reduction Strategies



- Call by value strategy
  - It evaluates an argument before passing it to the enclosing function
  - Can get stuck in a cycle of non-terminating reductions, evaluating an argument that will never be used
  - However, if the reduction terminates it may do so in fewer  $\beta$  reductions

---

---

---

---

---

---

---

---

## Reduction Strategies



- Example  
 $T = (\lambda xy. x) z N = (\lambda x. \lambda y. x) z N$
  - Call by name:  
 $(\lambda xy. x) z N \Rightarrow_{\beta} (\lambda y. z) N \Rightarrow_{\beta} z$
- Reduction terminates irrespective of what N precisely is!

---

---

---

---

---

---

---

---

## Reduction Strategies



- Example  
 $(\lambda xy. x) z N$ ;  $N = (\lambda x. xx) (\lambda x. xx)$
- Recall: N by itself is a non terminating computation
- Call by value:  
 $(\lambda xy. x) z N \Rightarrow_{\beta} (\lambda y. z) ((\lambda x. xx) (\lambda x. xx))$   
 $\Rightarrow_{\beta} (\lambda y. z) ((\lambda x. xx) (\lambda x. xx))$   
 $\Rightarrow_{\beta} \dots$

---

---

---

---

---

---

---

---

## Reduction Strategies



- Call by value
  - Is stuck in the non terminating reduction of  $N = (\lambda x. xx) (\lambda x. xx)$  though N is not needed by the actual computation
- Another example:  $(\lambda x. xx) T$ 
  - Call by name shall evaluate T twice, once for each copy of x
  - Call by value wins by first reducing T to a normal form

---

---

---

---

---

---

---

---

## Currying



- Every function accepts a single argument
- A function with two arguments is expressed as a function of one argument which returns a one-argument function e.g.  
 $f(x, y) = x * y$   
would be written in lambda calculus as  
 $\lambda x. \lambda y. x * y$   
 $= \lambda xy. x * y$

---

---

---

---

---

---

---

---

## Currying



- Extending it to multiple arguments, function g is a **curried form** of function f if:  
 $f(x_1, x_2, \dots, x_n) = g x_1 x_2 \dots x_n$   
where f has  $n \geq 0$  arguments, which g takes one at a time
- $\lambda xy. x * y$  is a curried form of the binary multiplication operator  $f(x, y) = x * y$

---

---

---

---

---

---

---

---

## Currying



- The expression  $2 * 3$  corresponds to  $(\lambda xy. x * y) 2 3$
- Partial Evaluation: The function  $\lambda xy$  expects two arguments,  $x$  and  $y$ . What if only  $x$  were provided?  
 $(\lambda xy. x * y) 2$

---

---

---

---

---

---

---

---

## Currying



- Partial Evaluation: evaluation of a term when not all of its arguments are available  
 $(\lambda xy. x * y) 2 \Rightarrow_{\beta} (\lambda y. 2 * y)$   
is a function that multiplies its argument  $y$  by 2

---

---

---

---

---

---

---

---

## Applications of $\lambda$ Calculus



- Simple model for studying functional languages
- Tool for study of computability
  - Halting Problem
  - Undecidability of whether two lambda calculus terms are equivalent

---

---

---

---

---

---

---

---

## Applications of $\lambda$ Calculus



- Parameter passing
  - Call by name
  - Call by value
- Church-Rosser theorem: independence of result and order of evaluation of terms

---

---

---

---

---

---

---

---

## Church Numerals



- Popular construction of natural numbers in lambda calculus
- Intuitively, the number  $n$  in lambda calculus is a function that takes a function  $f$  as argument and returns the  $n$ -th power of  $f$
- Higher order programming: each Church integer is a higher order function

---

---

---

---

---

---

---

---

## Church Numerals



$0 = \lambda f. \lambda x. x$   
 $1 = \lambda f. \lambda x. f x$   
 $2 = \lambda f. \lambda x. f(f x)$   
 $3 = \lambda f. \lambda x. f(f(f x))$   
...  
 $n = \lambda f. \lambda x. f^n(x)$

---

---

---

---

---

---

---

---

## Church Numerals



- We can define arithmetic functions on Church numerals
  - Successor: function which takes a Church integer  $n$  and returns  $n+1$   
 $\text{SUCC} = \lambda n. \lambda f. \lambda x. f (n f x)$
  - Verify that  $\text{SUCC } 0 = 1$

---

---

---

---

---

---

---

---

## Drawbacks of $\lambda$ calculus



- Too simplistic for real world programming
- Performance Measurement: call-by-value (if it terminates) is faster than call-by-name but by how much?

---

---

---

---

---

---

---

---

## Your turn!



---

---

---

---

---

---

---

---