



# LAMBDA CALCULUS: AN INTRODUCTION

Chandreyee Chowdhury

# OUTLINE

Why study lambda calculus?

Lambda calculus

- Syntax
- Evaluation
- Relationship to programming languages

# WHAT IS COMPUTABLE?

In the 1930s, two researchers from two countries Alan Turing from England and Alonzo Church from the US started analyzing the question of what can be computed.

**Godel** defined the class of *general recursive functions as the smallest set of functions*

- all the constant functions, the successor function, and closed under certain operations (such as compositions and recursion)

A function is computable (in the intuitive sense) if and only if it is general recursive

**Church** defined an idealized programming language called the *lambda calculus*,

- *a function is computable (in the intuitive sense) if and only if it can be written as a lambda term*

- Turing argued that a function is computable if and only if it can be computed on a Turing machine

# THE CONJECTURE

**Church's Thesis** : The effectively computable functions on the positive integers are precisely those functions definable in the pure lambda calculus (and computable by Turing machines).

The conjecture cannot be proved since the informal notion of “effectively computable function” is not defined precisely.

But since all methods developed for computing functions have been proved to be no more powerful than the lambda calculus, it captures the idea of computable functions

# WHY STUDY $\Lambda$ -CALCULUS

We will see a number of important concepts in their simplest possible form, which means we can discuss them in full detail

We will then reuse these notions frequently to build the different code blocks.

The notion of function is the most basic abstraction present in nearly all programming languages.

If we are to study programming languages, we therefore must strive to understand the notion of function.

# FUNCTION CREATION

Church introduced the notation

$\lambda x. E$

to denote a function with formal argument  $x$  and with body  $E$

Functions do not have names

- names are not essential for the computation

Functions have a single argument

- Only one argument functions are discussed

# FUNCTION APPLICATION

The only thing that we can do with a function is to apply it to an argument

Church used the notation

$E_1 E_2$

to denote the application of function  $E_1$  to actual argument  $E_2$   
 $E_1$  is called (ope)rator and  $E_2$  is called (ope)rand

All functions are applied to a single argument

# SIGNIFICANCE OF $\lambda$ -CALCULUS

$\lambda$ -calculus is the standard testbed for studying programming language features

- Because of its minimality
- Despite its syntactic simplicity the  $\lambda$ -calculus can easily encode:
  - numbers, recursive data types, modules, imperative features, etc.

Certain language features necessitate more substantial extensions to  $\lambda$ -calculus:

- for distributed & parallel languages:  $\pi$ -calculus
- for object oriented languages:  $\sigma$ -calculus



The central concept in  $\lambda$  calculus is the “expression”. A “name”, also called a “variable”, is an identifier which, for our purposes, can be any of the letters  $a, b, c, \dots$ . An expression is defined recursively as follows:

$$\begin{aligned}\langle \text{expression} \rangle &:= \langle \text{name} \rangle \mid \langle \text{function} \rangle \mid \langle \text{application} \rangle \\ \langle \text{function} \rangle &:= \lambda \langle \text{name} \rangle . \langle \text{expression} \rangle \\ \langle \text{application} \rangle &:= \langle \text{expression} \rangle \langle \text{expression} \rangle\end{aligned}$$

Variables  $x$

Expressions  $e ::= \lambda x. x \mid e \mid e_1 e_2$

# EXAMPLES OF LAMBDA EXPRESSIONS

```
var Identity = x => x;
```

The identity function:

$$I =_{\text{def}} \lambda x. x$$

Polymorphic expression

A function that given an argument  $y$  discards it and computes the identity function:

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

Constant function is defined as

$$\lambda y. z$$

## NOTATIONAL CONVENTIONS

Application associates to the left

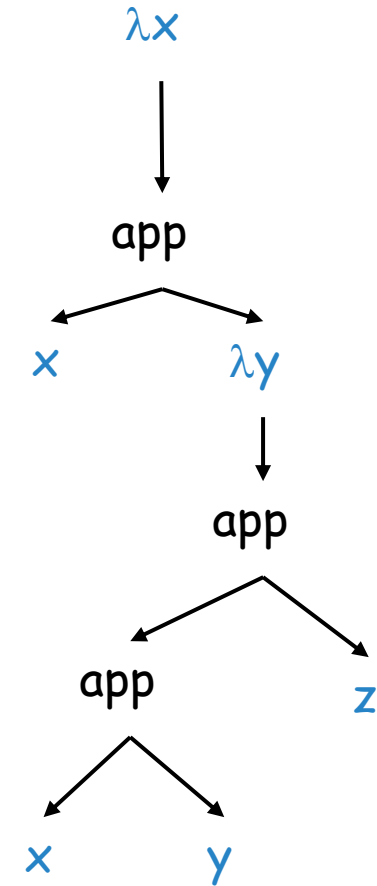
$x\ y\ z$  parses as  $(x\ y)\ z$

Abstraction extends to the right as far as possible

$\lambda x. x\ \lambda y. x\ y\ z$  parses as

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

And yields the the parse tree:



# SCOPE OF VARIABLES

As in all languages with variables, it is important to discuss the notion of scope

- Recall: the **scope** of an identifier is the portion of a program where the identifier is accessible

An abstraction  $\lambda x. E$  **binds** variable  $x$  in  $E$

- $x$  is the newly introduced variable
- $E$  is the scope of  $x$
- we say  $x$  is **bound** in  $\lambda x. E$
- Just like formal function arguments are bound in the function body

## FREE AND BOUND VARIABLES

A variable is said to be free in  $E$  if it is not bound in  $E$

Free variables are declared outside the term

We can define the free variables of an expression  $E$  recursively as follows:

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

$$\text{Free}(E_1 E_2) = \text{Free}(E_1) \cup \text{Free}(E_2)$$

$$\text{Free}(\lambda x. E) = \text{Free}(E) - \{ x \}$$

Example:  $\text{Free}(\lambda x. x (\lambda y. x y z)) = \{ ? \}$


A lambda expression with no free variables is called closed.

## FREE AND BOUND VARIABLES (CONT.)

Just like in any language with static nested scoping, we have to worry about variable shadowing

- An occurrence of a variable might refer to different things in different context

In  $\lambda$ -calculus:  $\lambda x. x (\lambda x. x) x$



# RENAMING BOUND VARIABLES

Two  $\lambda$ -terms that can be obtained from each other by a renaming of the bound variables are considered identical

Example:  $\lambda x. x$  is identical to  $\lambda y. y$  and to  $\lambda z. z$

Intuition:

- by changing the name of a formal argument and of all its occurrences in the function body, the behavior of the function does not change
- in  $\lambda$ -calculus such functions are considered identical

## RENAMING BOUND VARIABLES (CONT.)

Convention: we will always rename bound variables so that they are all unique

- e.g., write  $\lambda x. x (\lambda y.y) x$  instead of  $\lambda x. x (\lambda x.x) x$
- Variable capture or name clash problem would arise!

This makes it easy to see the scope of bindings

And also prevents serious confusion !



# SUBSTITUTION

The substitution of  $E'$  for  $x$  in  $E$  (written  $[E'/x]E$  )

- **Step 1.** Rename bound variables in  $E$  and  $E'$  so they are unique ( $\alpha$ -reduction)
- **Step 2.** Perform the textual substitution of  $E'$  for  $x$  in  $E$

This is called  $\beta$ -reduction

We write  $E \rightarrow_{\beta} E'$  to say that  $E'$  is obtained from  $E$  in one  $\beta$ -reduction step

We write  $E \rightarrow_{\beta}^* E'$  if there are zero or more steps

# FUNCTIONS WITH MULTIPLE ARGUMENTS

Consider that we extend the calculus with the `add` primitive operation

The  $\lambda$ -term  $\lambda x. \lambda y. \text{add } x \ y$  can be used to add two arguments  $E_1$  and  $E_2$ :

$$(\lambda x. \lambda y. \text{add } x \ y) E_1 E_2 \rightarrow_{\beta}$$

$$([E_1/x] \lambda y. \text{add } x \ y) E_2 =$$

$$(\lambda y. \text{add } E_1 \ y) E_2 \rightarrow_{\beta}$$

$$[E_2/y] \text{add } E_1 \ y = \text{add } E_1 \ E_2$$

The arguments are passed one at a time

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

$$((\lambda a. a) \ \lambda b. \lambda c. b) \ (x) \ \lambda e. f$$

# EVALUATION AND THE STATIC SCOPE

The definition of substitution guarantees that evaluation respects static scoping:

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



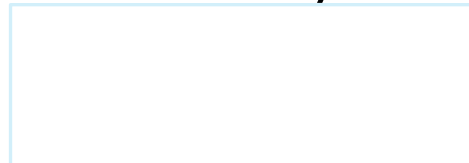
(y remains free, i.e., defined externally)

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



If we forget to rename the bound y:

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



(y was free before but is bound now)

**Definition:  $\alpha$ -reduction**

If  $v$  and  $w$  are variables and  $E$  is a lambda expression,

$$\lambda v . E \Rightarrow_{\alpha} \lambda w . E[v \rightarrow w]$$

provided that  $w$  does not occur at all in  $E$ , which makes the substitution  $E[v \rightarrow w]$  safe. The equivalence of expressions under  $\alpha$ -reduction is what makes part g) of the definition of substitution correct.

**Definition:  $\beta$ -reduction**

If  $v$  is a variable and  $E$  and  $E_1$  are lambda expressions,

$$(\lambda v . E) E_1 \Rightarrow_{\beta} E[v \rightarrow E_1]$$

provided that the substitution  $E[v \rightarrow E_1]$  is carried out according to the rules for a safe substitution.

  
 $\beta$ - redex

# ENCODING NATURAL NUMBERS IN LAMBDA CALCULUS

What can we do with a natural number?

- we can iterate a number of times

A natural number is a function that given an operation  $f$  and a starting value  $s$ , applies  $f$  a number of times to  $s$ :

$\text{one} =_{\text{def}} \lambda f. \lambda s. f \ (s)$

$\text{two} =_{\text{def}} \lambda f. \lambda s. f \ (f \ (s))$

and so on

$\text{zero} =_{\text{def}} \lambda f. \lambda s. s$

To translate the lambda expression to the programming world with datatypes

$(i \Rightarrow i+1)(0)$

$\text{two}(i \Rightarrow i+1)(0)$

# COMPUTING WITH NATURAL NUMBERS

$$\begin{aligned} 1 &\equiv \lambda sz.s(z) \\ 2 &\equiv \lambda sz.s(s(z)) \\ 3 &\equiv \lambda sz.s(s(s(z))) \end{aligned}$$

The successor function

successor  $n =_{\text{def}} \lambda n.\lambda f.\lambda x.f(nfx)$

Successor of 0 (S0) is  $_{\text{def}} (\lambda nfx.f(nfx)) (\lambda sz.z)$

$$\lambda yx.y((\lambda sz.z)yx) = \lambda yx.y((\lambda z.z)x) = \lambda yx.y(x) \equiv 1$$

Addition

$_{\text{def}} \lambda m.\lambda n.\lambda f.\lambda x.m(f)(n(f)(x))$

Multiplication

$_{\text{def}} \lambda m.\lambda n.\lambda f.\lambda x.m(n(f))(x)$

```
var anyNumber = n => f => x => (n(f))(x);  
var toNumber = n => n(i => i+1)(0);
```

# SOLVING A LAMBDA EXPRESSION

The main goal of manipulating a lambda expression is to reduce it to a “simplest form” and consider that as the value of the lambda expression.

**Definition :** A lambda expression is in **normal form** if it contains no  $\beta$ -redexes (and no  $\delta$ -rules in an applied lambda calculus), so that it cannot be further reduced using the  $\beta$ -rule or the  $\delta$ -rule. An expression in normal form has no more function applications to evaluate. ■

The only reduction possible for an expression in normal form is an  $\alpha$ -reduction.

# REDUCTION STRATEGIES

Can every lambda expression be reduced to a normal form?

Is there more than one way to reduce a particular lambda expression?

If there is more than one reduction strategy, does each one lead to the same normal form expression?

Is there a reduction strategy that will guarantee that a normal form expression will be produced?



## CAN EVERY LAMBDA EXPRESSION BE REDUCED TO A NORMAL FORM?

The identity function:

$$(\lambda x. x) E \rightarrow [E / x] x = E$$

Another example with the identity:

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

A non-terminating evaluation:

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

## IS THERE MORE THAN ONE WAY TO REDUCE A PARTICULAR LAMBDA EXPRESSION?

**Example :**  $(\lambda y . 5) ((\lambda x . x x) (\lambda x . x x))$

**Path 1 :**  $(\lambda y . 5) ((\lambda x . x x) (\lambda x . x x)) \Rightarrow_{\beta} 5$

**Path 2 :**  $(\lambda y . 5) ((\lambda x . x x) (\lambda x . x x))$

**Definition :** A **normal order** reduction always reduces the leftmost outermost  $\beta$ -redex (or  $\delta$ -redex) first. An **applicative order** reduction always reduces the leftmost innermost  $\beta$ -redex (or  $\delta$ -redex) first. ■

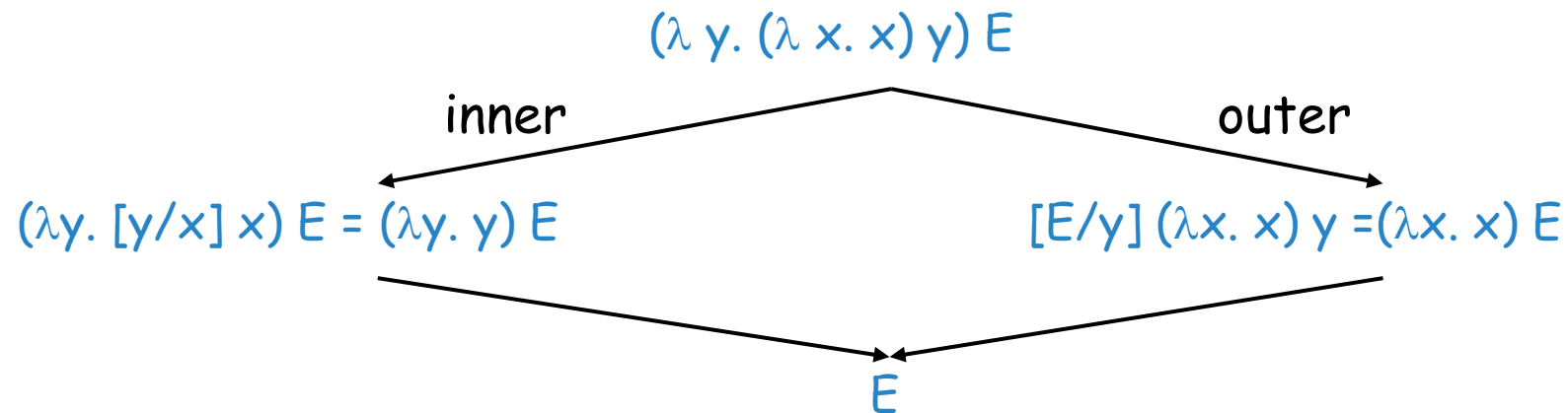
**Definition :** For any lambda expression of the form  $E = ((\lambda x . B) A)$ , we say that  $\beta$ -redex  $E$  is **outside** any  $\beta$ -redex that occurs in  $B$  or  $A$  and that these are **inside**  $E$ .

A  $\beta$ -redex in a lambda expression is **outer most** if there is no  $\beta$ -redex outside of it, and it is **inner most** if there is no  $\beta$ -redex inside of it.

IF THERE IS MORE THAN ONE REDUCTION STRATEGY, DOES EACH ONE LEAD TO THE SAME NORMAL FORM EXPRESSION?

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

- could reduce the inner or the outer  $\lambda$
- which one should we pick?



## ORDER OF EVALUATION (CONT.)

The **Church-Rosser theorem** says that any order will compute the same result

- A result is a  $\lambda$ -term that cannot be reduced further

**Church-Rosser Theorem I:** For any lambda expressions  $E$ ,  $F$ , and  $G$ , if  $E \Rightarrow^* F$  and  $E \Rightarrow^* G$ , there is a lambda expression  $Z$  such that  $F \Rightarrow^* Z$  and  $G \Rightarrow^* Z$ .

**Corollary:** For any lambda expressions  $E$ ,  $M$ , and  $N$ , if  $E \Rightarrow^* M$  and  $E \Rightarrow^* N$  where  $M$  and  $N$  are in normal form,  $M$  and  $N$  are variants of each other (equivalent with respect to  $\alpha$ -reduction).

Some evaluations may terminate while others may diverge. If two evaluations terminate, it will be to the same normal form.

DIAMOND PROPERTY (confluence)

# NORMAL ORDER REDUCTION

A normal order reduction can have either of the following outcomes

- 1. It reaches a unique (up to  $\alpha$ -conversion) normal form lambda expression
- 2. It never terminates

**Church-Rosser Theorem II:** For any lambda expressions  $E$  and  $N$ , if  $E \Rightarrow^* N$  where  $N$  is in normal form, there is a normal order reduction from  $E$  to  $N$ .

Unfortunately, there is no algorithmic way to determine for an arbitrary lambda expression which of these two outcomes will occur.

# THE CONJECTURE

- ❑ **Church's Thesis** : The effectively computable functions on the positive integers are precisely those functions definable in the pure lambda calculus (and computable by Turing machines).
- ❑ The conjecture cannot be proved since the informal notion of “effectively computable function” is not defined precisely.
- ❑ But since all methods developed for computing functions have been proved to be no more powerful than the lambda calculus, it captures the idea of computable functions

## IS THERE A REDUCTION STRATEGY THAT WILL GUARANTEE THAT A NORMAL FORM EXPRESSION WILL BE PRODUCED?

- Alan Turing proved a fundamental result, called the undecidability of the **halting problem**, which states that there is no algorithmic way to determine whether or not an arbitrary Turing machine will ever stop running. Therefore, there are lambda expressions for which it cannot be determined whether a normal order reduction will ever terminate.
- But we might want to fix the order of evaluation when we model a certain language

# CALL BY NAME

- ❑ Similar to Normal Order reduction
- ❑ Do not evaluate the argument prior to call

❑ Example:

❑  $(\lambda y. (\lambda x. x) y) ((\lambda u. u) (\lambda v. v)) \rightarrow^*_{\beta n}$

❑  $(\lambda v. v)$

- ❑ there is no evaluation of the actual parameter at the moment of the call
- ❑ The actual parameter will be evaluated only during the execution of the body, if needed

```
if (x==0 || 1/x > 0.3) ... // C++
```

- ❑ Does not give any exception for divide by zero



# CALL BY VALUE

Same as Applicative order reduction

Evaluate an argument prior to call

Example:

$(\lambda y. (\lambda x. x) y) ((\lambda u. u) (\lambda v. v)) \rightarrow^*_{\beta_v}$

$\lambda v. v$

# CALL BY NAME AND CALL BY VALUE

## CBN

- difficult to implement
- order of side effects not predictable

## CBV:

- easy to implement efficiently
- might not terminate even if CBN might terminate
- Example:  $(\lambda x. \lambda z. z) ((\lambda y. yy) (\lambda u. uu))$

Outside the functional programming language community, only CBV is used

# LAZY EVALUATION-CALL BY NEED

It is an evaluation strategy that combines on-demand evaluation with memoisation

- ❑ When a variable is bound, it is stored in the environment in an unevaluated form called a thunk, with the evaluation being delayed until the result is required to proceed
- ❑ When the thunk is evaluated, it is replaced by the result of its evaluation, avoiding any repetition of work if the value of the variable were to be needed again
- ❑ Benefits
  - ❑ A more compositional programming style
    - ❑ No need to ever construct the entire intermediate results
  - ❑ Infinite data and circular definitions
    - ❑ This would lead to non-termination in a strict language

# CALL-BY-NEED

## ❑ Problems

- ❑ The key issue with understanding lazy evaluation is that while the external interface provided to the programmer is declarative and pure, the internal implementation is stateful.
- ❑ Evaluating a term can result in updates to pre-existing environment variables
- ❑ The three benefits of better termination, faster evaluation and infinite data structures together represent the **GOOD** of laziness
- ❑ The difficulty of reasoning about efficiency represents the **BAD** of laziness.
- ❑ The need to give a stateful interpretation to an ostensibly pure language is downright **UGLY**!

## COMPARISON

- ❑ In call by need the actual parameter is evaluated only the first time it is needed
- ❑ Then the value is stored and used whenever it is needed again
- ❑ In call by name, on the contrary, the parameter is re-evaluated each time it is needed
- ❑ call-by-value is less terminating than call-by-need, as there could be some diverging subterm that a call-by-need strategy would avoid evaluating
- ❑ call-by-value is typically favoured by language designers because it is much easier to understand and reason about

### Definition: $\eta$ -reduction

If  $v$  is a variable,  $E$  is a lambda expression (denoting a function), and  $v$  has no free occurrence in  $E$ ,

$$\lambda v . (E \ v) \Rightarrow_{\eta} E.$$

$$\lambda x . (\text{sqr } x) \Rightarrow_{\eta} \text{sqr}$$

$$\lambda x . (\text{add } 5 \ x) \Rightarrow_{\eta} (\text{add } 5).$$

The  $\eta$ -reduction rule can be used to justify the extensionality of functions; namely, if  $f(x) = g(x)$  for all  $x$ , then  $f = g$

$\lambda x. (5 \ x)$  is not 5

**Extensionality Theorem:** If  $F_1 \ x \Rightarrow^* E$  and  $F_2 \ x \Rightarrow^* E$  where  $x \notin FV(F_1 \ F_2)$ , then  $F_1 \Leftrightarrow^* F_2$  where  $\Leftrightarrow^*$  includes  $\eta$ -reductions.

### Definition: $\delta$ -reduction

If the lambda calculus has predefined constants (that is, if it is not pure), rules associated with those predefined values and functions are called  $\delta$  rules; for example,  $(\text{add } 3 \ 5) \Rightarrow_{\delta} 8$  and  $(\text{not true}) \Rightarrow_{\delta} \text{false}$ .

# LAMBDA CALCULUS TO PROGRAMMING

## Data Types

- Booleans, numbers
- Collections

## Conditional expressions

## Arithmetic expressions

## Recursions

a *combinator* is a  $\lambda$ -term with no free variables

# BOOLEAN DATA TYPE

A boolean is a function that given two choices selects one of them

- $\text{true} =_{\text{def}} \lambda \text{then\_do. } \lambda \text{else\_do. then\_do}$
- $\text{false} =_{\text{def}} \lambda \text{then\_do. } \lambda \text{else\_do. else\_do}$
- $\text{if\_then\_else} =_{\text{def}} \lambda \text{cond. } \lambda \text{then\_do. } \lambda \text{else\_do. Cond (then\_do) (else\_do)}$

Example: Any\_Assignment\_Deadlines

$\text{SleepHours} = \text{if\_then\_else}(\text{Any\_Assignment\_Deadlines}) (\text{six}) (\text{ten})$

**NOT**  $=_{\text{def}} \lambda \text{boolean. } \lambda \text{then\_do. } \lambda \text{else\_do. } \underline{\text{boolean}} (\text{else\_do}) (\text{then\_do})$

$\text{Is\_even} = \lambda n. n(\text{NOT})(\text{true})$

$\text{Is\_Zero} = (\lambda n. n (\lambda x. \text{false}) \text{true})$

$\text{AND} == \lambda a. \lambda b. a \text{ b FALSE}$

$\text{OR} == \lambda a. \lambda b. a \text{ TRUE b}$

Boolean	Outcome of the expression
true	false
false	true