

# Introduction to Lambda Calculus

## Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

©Hans de Nivelles, 2025

## Introduction

Lambda Calculus (after Greek letter  $\lambda$ ) was introduced by Alonzo Church in the 1930s as a calculus for reasoning about functions.

There are many variations of  $\lambda$ -calculus. The  $\lambda$ -calculus can be typed or untyped, Church or Curry typed, and there can be additional operators.

## Defining Functions

Use  $\lambda$  to define functions:

Square function:  $\lambda x (x \times x)$ .

Gaussian sum:  $\lambda x \frac{x(x+1)}{2}$ .

It is important to understand that functions are **closed objects**.

Similar notation exists in programming languages:

```
[] ( double x ) { return x * x; } // C++
```

```
(x) -> x * x; // Java
```

```
lambda x : x * x // Python
```

Moreover, every function definition is a lambda definition.

## Defining Functions

Mathematicians are defining functions all the time, but they are not good at it:

Let the function  $f$  be defined from  $f(x) = x^2$ .

The derivative of  $f$  is defined as

$$\frac{d f(x)}{d x} = \frac{d x^2}{d x} = 2x.$$

The integral of  $f$  is defined as

$$\int_0^y f(x).d x = \int_0^y x^2.d x = \frac{1}{3}y^3.$$

## $\lambda$ : The first and only binder

What is the  $dx$  in  $\frac{df(x)}{dx}$  or  $\int_0^x x^2 \cdot dx$ ?

Probable somebody told you: It is the size of an infinitely small interval.

Wrong: It is just a  $\lambda$ .

Differentiation and Integration are operations on functions. (They take a function  $\mathcal{R} \rightarrow \mathcal{R}$  and return a function  $\mathcal{R} \rightarrow \mathcal{R}$ .)

Do you know more binders?

## Two or More Variables

One can also build functions in two variables:

Same as +:

$$\lambda xy \ x + y.$$

Reverse of −:

$$\lambda xy \ y - x.$$

Just some expression:

$$\lambda xy \ x^2 + xy.$$

Three variables:

$$\lambda xyz \ \frac{x}{y^2 + z^2}.$$

## Currying

A function in two variables can be viewed as a one variable function that returns a function.

For example  $+$  (addition) is a function with signature  $\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$ .

If you give it a first number, e.g.  $3$ , it becomes a thing that waits for the second number, after which it can produce the sum.

Therefore,  $+(3)$  is a function with signature  $\mathcal{R} \rightarrow \mathcal{R}$ .

$+(3)$  is the function that adds  $3$  to any number.

$$1 + 2 = +(1, 2) = +(1)(2).$$

## Currying (2)

We have seen that all functions can be made unary. This means that  $x + y$  can be written as  $+(x)(y)$ .

Since this notation is unpleasant, we introduce an application operator  $\cdot$ .

$f \cdot t$  means: **application of  $f$  on  $t$** .

$(f \cdot t_1) \cdot t_2$  means  $f(t_1, t_2)$ .

Similarly,  $((f \cdot t_1) \cdot t_2) \cdot t_3$  means  $f(t_1, t_2, t_3)$ .

Just like multiplication, the  $\cdot$  operator is usually omitted. We write  $(f t_1 t_2 t_3)$  instead of  $((f \cdot t_1) \cdot t_2) \cdot t_3$ .

## Currying (3)

Once we have Currying, we can restrict  $\lambda$  to a single variable:

$$\lambda xy \ x^2 + xy \text{ becomes } \lambda x \ \lambda y \ x^2 + xy.$$

Types of form  $\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$  can be written as

$$\mathcal{R} \rightarrow (\mathcal{R} \rightarrow \mathcal{R}).$$

(Give it one  $x:\mathcal{R}$ , and it becomes a function  $\mathcal{R} \rightarrow \mathcal{R}$ .)

**Currying** is called after **Haskell Curry** (1900-1982).

## Excursion: Binders in Practice: Laplace Transform

(From Elementary Differential Equations and Boundary Value Problems, William E. Boyce and Richard C. DiPrima.)

*Let  $f(t)$  be given for  $t \geq 0$ , and suppose that  $f$  satisfies certain conditions to be stated a little later. Then the **Laplace transform** of  $f$ , which we will denote by  $\mathcal{L}\{f(t)\}$  or by  $F(s)$ , is defined by the equation*

$$\mathcal{L}\{f(t)\} = F(s) = \int_0^{\infty} e^{-st} f(t) dt.$$

You see that the authors are struggling with the binders. This explains their use of  $\{ \}$ . Note that  $dt$  is just  $\lambda t$ . On Page 283 of the same book, they show that  $\mathcal{L}\{\sin(at)\}$  is the function defined by  $F(s) = \frac{a}{s^2 + a^2}$ , for  $s > 0$ .

## Excursion: Binders in Practice: Derivatives

Consider the expression  $x^2y$ . One can differentiate with respect to  $x$  or with respect to  $y$ .

Better: View differentiation as an operator on functions.

One can either differentiate  $\lambda x x^2y$  or  $\lambda y x^2y$ .

## $\lambda$ -terms

**Definition:** We define  $\lambda$ -terms by recursion as follows:

- A variable is a  $\lambda$ -term.
- If  $f$  and  $t$  are  $\lambda$ -terms, then  $f \cdot t$  is a  $\lambda$ -term.
- If  $x$  is a variable, and  $t$  is a  $\lambda$ -term, then  $\lambda x t$  is a  $\lambda$ -term.

$\lambda$ -terms can be typed, for example by adding type information to  $\lambda$  when a variable is introduced.

One can also add primitive data types, for example numbers.

For the moment, we don't worry about types.

Using  $\lambda$ -notation, a function can be written as

$$\lambda x: X \ F(x),$$

which is a closed object.

Using this, one can define  $\mathcal{L}: (\text{Real} \rightarrow \text{Real}) \rightarrow (\text{Real} \rightarrow \text{Real})$  as

$$\mathcal{L} := \lambda f: (\text{Real} \rightarrow \text{Real}) \ \lambda s: \text{Real} \ \int_0^\infty \lambda t: \text{Real} \ e^{-st} f(t).$$

(It is a partial function, but we ignore this. In fact, it is so partial that it is hardly ever defined.)

The statement (from Slide 10) can be written as

$$\forall a: \text{Real} \ \mathcal{L}(\lambda t: \text{Real} \ \sin at) = \lambda s: \text{Real} \ \frac{a}{s^2 + a^2}.$$

## $\alpha$ -Equivalence

When are two  $\lambda$  terms equal?

As long as there are no  $\lambda$ -s, it is easy:

$(+ (f a) b)$  is equal to  $(+ (f a) b)$ , and not equal to  $(\times (f a) c)$ .

What about  $\lambda x \lambda y (f x y)$  and  $\lambda y \lambda x (f y x)$ ?

What about  $\lambda x \lambda y (f y y)$  and  $\lambda x \lambda x (f x x)$ ?

Names of local variables do not matter. What matters is the pattern of where they are declared, and where they are used.

## $\alpha$ -Equivalence

It is clear that in every reasonable application,  $\alpha$ -equivalent terms should have the same meaning.

$\alpha$ -equivalence must have some reasonable properties:

**Reflexivity:** Every term  $t$  is  $\alpha$ -equivalent to itself.

**Symmetry:** If  $t_1$  is  $\alpha$ -equivalent to  $t_2$ , then  $t_2$  is  $\alpha$ -equivalent to  $t_1$ .

**Transitivity:** If  $t_1$  is  $\alpha$ -equivalent to  $t_2$ , and  $t_2$  is  $\alpha$ -equivalent to  $t_3$ , then  $t_1$  is  $\alpha$ -equivalent to  $t_3$ .

Alternatively, one can say that  $\alpha$ -equivalence is **an equivalence relation**.

## Occurrences

Consider the term

$$\lambda y (+ ((\lambda x (- x z)) 3) (* x y)).$$

Variable  $x$  occurs 3 times, one time as  $\lambda x$ , one time **bound**, (by the first  $\lambda x$ ), and one time **free**.

Variable  $y$  occurs 2 times, one time as  $\lambda y$ , one time bound.

Variable  $z$  occurs 1 time, free.

We usually care only about the free occurrences, because they must have a meaning outside of the term.

Bound occurrences can always be replaced by other variables (using  $\alpha$ -equivalence).

## Free

Instead of saying 'there is a free occurrence of  $x$  in  $t$ ', one can also say ' $x$  is free in  $t$ ' or ' $x$  occurs freely in  $t$ '.

We recursively define a function **free**( $x, t$ ), with meaning 'variable  $x$  is free in term  $t$ '.

As before, the definition follows the recursive structure of  $t$  :

- If  $t$  is a variable, then **free**( $x, t$ )  $\Leftrightarrow x = t$ .
- If  $t$  has form  $t_1 \cdot t_2$ , then

$$\mathbf{free}(x, t) \Leftrightarrow \mathbf{free}(x, t_1) \text{ or } \mathbf{free}(x, t_2).$$

- If  $t$  has form  $\lambda y t'$ , then

$$\mathbf{free}(x, t) \Leftrightarrow x \neq y \text{ and } \mathbf{free}(x, t').$$

## Substitution

Substitution is the replacement of a free variable by a value.

We write  $t[x := u]$  for the term that is obtained when all free occurrences of  $x$  are replaced by  $u$ .

If  $t = \lambda y (+ ((\lambda x (- x z)) 3) (* x y))$ , then

$$t[x := (+ x 1)] = ?$$

$$t[y := (+ y 1)] = ?$$

$$t[z := (+ z 1)] = ?$$

$$t[z := (+ x y)] = ?$$

## Capture

If capture occurs, then  $t$  has to be replaced by an  $\alpha$ -variant in which no capture occurs.

It doesn't matter which variant, because  $\alpha$ -variants are equal.

$$t[ z := (+ x y) ] = \lambda u (+ ((\lambda w (- w (+ x y))) 3) (* x u)).$$

For example  $\int_0^4 xy dy = 8x$ . If one wants to apply this equation with  $x = (y + 1)$ , one must replace  $dy$  by another variable, for example  $dz$ . The result is  $\int_0^4 (y + 1)z dz = 8(y + 1)$ .

## Substitution

We write  $t[x := u]$  for the result of replacing the free occurrences of  $x$  in  $t$  by term  $u$ . As usual, we define it by recursion on the structure of  $t$ :

- If  $t$  is a variable, then  $t[x := u]$  is defined as **if**  $t = x$  **then**  $u$  **else**  $t$ .
- If  $t$  has form  $t_1 \cdot t_2$ , then  $(t_1 \cdot t_2)[x := u]$  is defined as  $t_1[x := u] \cdot t_2[x := u]$ .
- If  $t$  has form  $\lambda y t'$ , then
  - If  $x = y$ , then  $(\lambda y t')[x := u] = (\lambda y t')$ .
  - If  $x \neq y$ ,  $x$  is free in  $t'$ , and  $y$  is free in  $u$ , then find a variable  $z$  that does not occur in  $t'$  or  $u$ . Define  $(\lambda y t')[x := u]$  as  $\lambda z t'[y := z][x := u]$ .
  - Otherwise,  $(\lambda y t')[x := u]$  is defined as  $\lambda y (t'[x := u])$ .

Substitution is conceptually not difficult, but it is tricky to define it precisely, and to implement it correctly.

One possible way out is using **De Bruijn indices**:

Replace bound variables in formulas by natural numbers that indicate how many lambdas one needs to skip in order to find the lambda that binds the variable. #0 means first quantifier, #1 means second quantifier, etc.

$$\lambda y (+ ((\lambda x (- x z)) 3) (* x y))$$

becomes

$$\lambda (+ ((\lambda (- \#0 z)) 3) (* x \#0))$$

Term  $\lambda x \lambda y (f x y)$  becomes  $\lambda \lambda (f \#1 \#0)$ .

This also solves the problem of testing for  $\alpha$ -equivalence. Formulas with the Bruijn indices are  $\alpha$ -equivalent iff they are exactly equal.

## Computing with $\lambda$ -calculus

Our goal is to use  $\lambda$ -calculus as a model for computation.

The meaning of  $\lambda x (+ x 1)$  is ‘the function that returns  $(+ x 1)$ , for every  $x$ ’.

This means that whenever  $\lambda x (+ x 1)$  is applied on a term  $t$ , the result must be  $(+ t 1)$ .

The corresponding rule is called  **$\beta$ -reduction**:

$$(\lambda x f) \cdot t \Rightarrow_{\beta} f[x := t].$$

Sometimes, we don’t care about the direction. In that case, we say that  $(\lambda x f) \cdot t$  and  $f[x := t]$  are  **$\beta$ -equivalent**.

## $\beta$ -Equivalence, $\beta$ -Reduction

$\beta$ -reductions can also be made deeper inside terms, e.g.

$$(f ((\lambda x (g x)) t)) \Rightarrow_{\beta} (f (g t)) \text{ or}$$

$$\lambda y ((\lambda x (g x)) t) \Rightarrow_{\beta} \lambda y (g t).$$

$$(\lambda n (+ n n)) \cdot 2 \Rightarrow_{\beta} (+ 2 2).$$

$$(+ 1 ((\lambda n (+ n n)) 7)) \Rightarrow_{\beta} (+ 1 (+ 7 7)).$$

## Extension to Multi Argument Functions

Let's see how it works with multiple argument functions:

Let us define  $f = \lambda x \lambda y (- y x)$ .

What does this function do?

$$\begin{aligned}(f\ 2\ 3) &= ((\lambda x \lambda y (- y x) \cdot 2) \cdot 3) \Rightarrow_{\beta} ((\lambda y (- y x))[x := 2]) \cdot 3 \\ &= (\lambda y (- y 2)) \cdot 3 \Rightarrow (- 3 2).\end{aligned}$$

The final term can be simplified into 1, which is as expected.

## $\delta$ -Equivalence, $\delta$ -Reduction

Expansion of definitions is also a reduction, called  $\delta$ -reduction.

If identifier  $v$  is defined as  $v := u$ , and term  $t$  contains free occurrences of  $u$ , then some occurrences can be replaced by  $u$ .

$\delta$ -reduction is similar to substitution but not exactly the same, because there can be different outcomes dependent on which identifier is replaced.

Hence we need to define  $\Rightarrow_\delta$  as a relation.

We write  $t \Rightarrow_{\delta} t'$  if  $t'$  can be obtained from  $t$  by replacing exactly one identifier by its defined value.

$\Rightarrow_{\delta}$  is the  $\subseteq$ -minimal relation that meets the following conditions:

- If  $v$  is defined as  $u$ , then  $v \Rightarrow_{\delta} u$ .
- If  $t$  has form  $t_1 \cdot t_2$ , and  $t_1 \Rightarrow_{\delta} t'_1$ , then  $t_1 \cdot t_2 \Rightarrow_{\delta} t'_1 \cdot t_2$ .  
If  $t_2 \Rightarrow_{\delta} t'_2$ , then  $t_1 \cdot t_2 \Rightarrow_{\delta} t_1 \cdot t'_2$ .
- If  $t$  has form  $\lambda y t'$ ,  $z$  is a variable that is used *nowhere else*, and  $t'[y := z] \Rightarrow_{\delta} t''$ , then  $\lambda y t' \Rightarrow_{\delta} \lambda z t''$ .

(The last condition is hard to implement.)

## Normal Forms

Let  $t$  be a term. If there is no  $t'$ , s.t.  $t \Rightarrow_{\beta} t'$ , then we call  $t$  **in normal form**.

What happens when one keeps on applying  $\beta, \delta$ -reductions?

Either one reaches a normal form, or the rewriting goes on forever.

1. Does one always reach a normal form?
2. In case, more than one  $\beta$ -reduction is possible, does it matter which one you take?
  - (a) Could it be that different rewrite strategies result in different normal forms?
  - (b) Could it be that one rewrite strategy results in a normal form, while another one doesn't?
  - (c) Could it be that one strategy is more efficient than another strategy?

## Properties of Normalization

- We call a term  $t$  **weakly normalizing** if it can be rewritten into a normal form in a finite number of steps.
- A term  $t$  is **strongly normalizing** if every rewrite sequence starting with  $t$  will end in a normal form.
- A system is **confluent** if it has the following property: If  $t \Rightarrow t_1$  and  $t \Rightarrow t_2$ , then there exists a term  $u$ , s.t.  $t_1 \Rightarrow^* u$ , and  $t_2 \Rightarrow^* u$ .

It is also called **the diamond property**. If you draw the figure, you see why.

There exist terms that are not normalizing, for example

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

But  $\lambda$ -calculus has the diamond property. It implies that a term can have 0 or 1 normal forms.

## Properties of $\lambda$ calculus

- Topmost application of  $\beta$ -reduction is optimal (from termination point of view) strategy.

If a term  $t$  has a normal form, then topmost evaluation will find it. Haskell uses this. It allows you to do crazy things with infinite objects.

- The diamond property implies that a term cannot have more than one normal form. The proof is surprisingly easy.
- Some systems of typed lambda calculus have strong normalization.

## Definitions are Tricky

We have been using definitions implicitly, for example a few slides ago, we wrote 'let us define  $f = \lambda x \lambda y (- y x)$ '.

Definitions must be **conservative**.

This means that one could remove them, and nothing would happen. Maybe the program will get longer, but nothing more.

In order to have this, definitions must be not **circular**, if you define  $x := t$ , then  $x$  must not occur in  $t$ .

The definition above, of  $f$ , is not circular.

## Circular Definitions

Most programming languages allow circular definitions:

```
unsigned int fact( unsigned int n ) {  
    return ( n == 0 ) ? 1 : x * fact( n - 1 ); }  
}
```

```
struct list  
{  
    int head;  
    list* rest;  
};
```

Such definitions are dangerous, because the defined name cannot be replaced by its value.

For example if I define ‘let  $n$  be the natural number defined from  $n = n + 1$ ’, which number did I define?

## Limits

It is especially bad if you want to prove properties about the things that you define.

The definition is not a definition anymore, but an equation.

If the equation has multiple solutions, you may be proving things about one solution, while somebody else uses another solution.

If the equation has no solutions, you may be proving things from contradiction.

The solution to such circular definitions is the use of **fixed points**.

## Summary

$\lambda$ -calculus is a mechanism for defining computation. (Usually, one adds some more reductions, e.g. primitive operations.)

It is as fundamental as Turing machines.

Historically, it is older.

It is also more expressive.  $\lambda$ -calculus is more like a high-level language. Programming  $\lambda$ -calculus is easier than programming Turing machines.

It is derived from mathematical analysis, not from electronics.