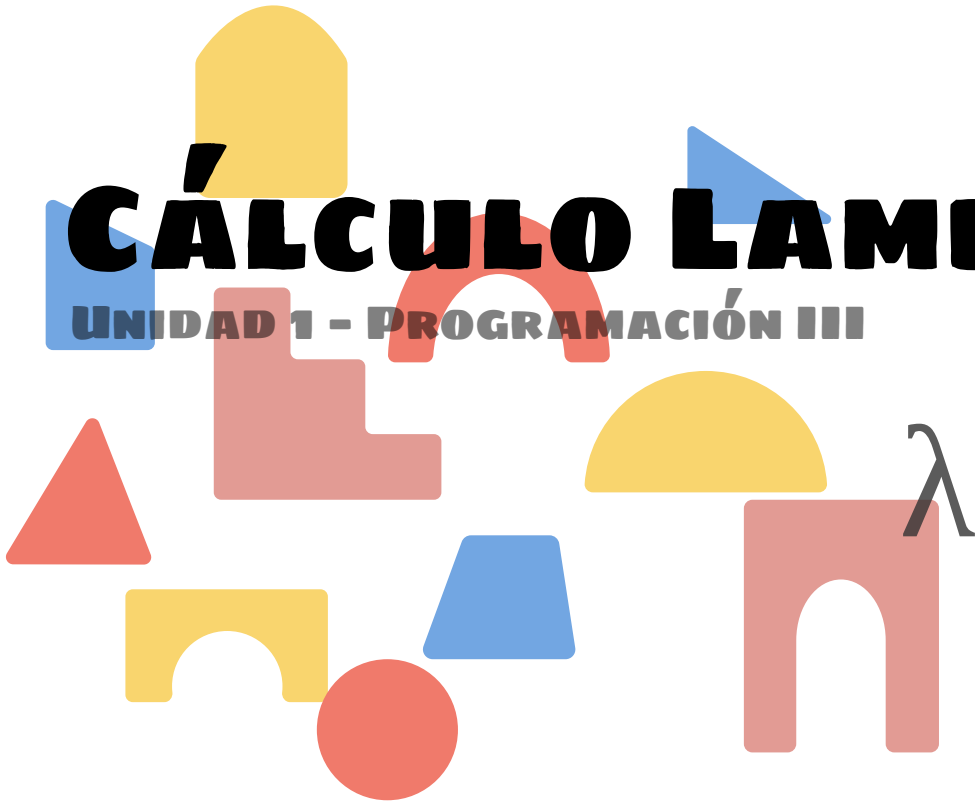
Cálculo Lambda

Unidad 1 - Programación III



INSPT - UTN - Ciclo Lectivo 2026

Alonzo Church - Creador del Cálculo Lambda

Lógico y matemático estadounidense.

- Desarrolló el **Cálculo Lambda** (década de 1930)
- Computación teórica
- Profesor de **Alan Turing**
- La **Tesis de Church-Turing** establece la equivalencia entre el cálculo lambda y

Fundamentos

Sistema Formal

El cálculo lambda es un sistema formal que sirve como base teórica de la programación funcional.

- Definición de función
- Aplicación de función
- Recursividad

Toda **función computable** puede ser expresada y evaluada mediante este sistema.

≠ Máquina de Turing

A diferencia de la máquina de Turing, el cálculo lambda se basa en:

- **Reglas de transformación** sobre expresiones simbólicas
- Sin necesidad de una **implementación real** (cinta, estados, cabezal)

Ambos modelos son **equivalentes en poder computacional** (Tesis de Church-Turing).

Programa funcional

Un programa funcional es una expresión compuesta por:

- Un **algoritmo** (las transformaciones a aplicar)
- Las **entradas** (los datos sobre los que opera)

El resultado se obtiene **evaluando** (reduciendo) la expresión.

Reglas de conversión - Reducción

$$E[P] \to E[P'], \text{ siempre y cuando } P \to P'$$

Una expresión se **reduce** reemplazando una subexpresión por su forma simplificada.

Ejemplo - Reducción aritmética

$$(7+4)\times(8+5\times3)$$

$$\rightarrow 11\times(8+5\times3)$$

$$\rightarrow 11\times(8+15)$$

$$\rightarrow 11\times23$$

$$\rightarrow 253$$

Ejemplo - Reducción funcional

```
primero (ordenar (unir (["perro", "conejo"], ordenar (["ratón", "gato"]))))
→ primero (ordenar (unir (["perro", "conejo"], ["gato", "ratón"])))
→ primero (ordenar (["perro", "conejo", "gato", "ratón"]))
→ primero (["conejo", "gato", "perro", "ratón"])
→ "conejo"
```

Definición Formal

Elementos

Una expresión lambda puede ser una de tres cosas:

- **Variable**: un nombre (x, y, z)
- **Abstracción**: una función ($\lambda x.M$)
- **Aplicación**: aplicar una función a un argumento ($M\ N$)

Sintaxis

$$\langle expresión\ \lambda \rangle ::= \langle variable \rangle \mid (\lambda \langle variable \rangle . \langle expresión\ \lambda \rangle) \mid (\langle expresión\ \lambda \rangle\ \langle expresión\ \lambda \rangle)$$

Ejemplos - Expresiones lambda

```
x
(λx.x)
((λx.x) y)
((λx.(x y)) ((λy.(y y)) z))
(((λx.(x y)) (λy.(y y))) z)
```

Convenciones

Abreviación de múltiples parámetros:

$$\lambda x.\lambda y.\lambda z.M \equiv \lambda x\ y\ z.M$$

Asociatividad izquierda de la aplicación:

$$M\ N\ P \equiv (M\ N)\ P$$

Alcance del λ se extiende lo más posible a la derecha:

$$\lambda x.M\ N \equiv \lambda x.(M\ N)$$

Ejemplo completo:

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

$$\equiv (\lambda x.(\lambda y.(y\ x)))\ a\ b$$

$$\equiv (\lambda x.\lambda y.\ y\ x)\ a\ b$$

Variables Libres y Ligadas

De funciones matemáticas a variables

En una función matemática, las variables pueden estar ligadas a un parámetro o ser libres (depender del contexto externo):

- $f(x) = x^2 + 2 \rightarrow x$ está **ligada** al parámetro
- $f(z, w) = z + w^2 + 2 \rightarrow z$ y w están **ligadas**
- $g(x) = x + y \rightarrow x$ está **ligada**, pero y es **libre**

En el cálculo lambda ocurre exactamente lo mismo con el operador λ.

Variables Libres y Ligadas - Definición

La variable x ocurre ligada en la expresión N si y solo si:

- $N = \lambda z.M$ siendo $x = z$ o cuando x ocurre ligada en M
- $N = M\ P$ donde x ocurre ligada en M y/o en P

La variable **x** ocurre **libre** en la expresión N si y solo si:

- $N = \lambda z.M$ siendo $x \neq z$ y donde x ocurre libre en M
- $N = M\ P$ donde x ocurre libre en M y en P

Ejemplo - Variables libres y ligadas

Expresión	Libres	Ligadas
$\lambda z.x\ y$	x, y	—
$\lambda x.x\ y$	y	x
$\lambda y.x\ y$	x	y
$\lambda x\ y.x\ y$	—	x, y
$(\lambda z.z\ x\ y)\ (\lambda x.x)$	x, y	z, x

Reglas de conversión

Regla de conversión Alfa (α)

Sustitución de variables

$$\lambda x.\, M =_\alpha \lambda y.\, M[y/x]$$

Permite **renombrar** la variable ligada de una abstracción sin cambiar el significado de la expresión.

Ejemplos - Conversión α

```
λx.x                    →α λy.y
λx.y x                  →α λz.y z
λx.z x x (λu x.x u) v x →α λy.z y y (λu x.x u) v y
λx y.x z y              → ?
```

Regla de conversión Beta (β)

Aplicación de un valor en una entrada

$$\beta\text{-redex} \Rightarrow (\lambda x.M)\ N$$

Consiste en realizar la **reducción** de una β-redex: sustituir todas las ocurrencias libres de x en M por N.

Ejemplo - Identificando β-redex

```
(λx.x x) z                              ← β-redex
(λx.x x) (λy.y y)                       ← β-redex
(λx.x) (λy.y y) z                       ← β-redex
(λx.(λu.u) (λv.x v)) ((λt.t t) w)       ← β-redex
```

```
x (λy.y y)    ← NO ES UNA β-redex
x (λy.y y)    ← FORMA NORMAL
```

Ejemplos - Reducción β

```
(λx.x) z =β z
```

```
(λx.(λu.u) (λv.x v)) ((λt.t t) w)
  =β (λu.u) (λv.((λt.t t) w) v)
  =β λv.((λt.t t) w) v
  =β λv.(w w) v
```

```
(λz w.w z w) w x =β ?

(λx.x x) (λy.y y) =β ?
```

Regla de conversión Eta (η)

Extensionalidad

$$\eta\text{-redex} \Rightarrow \lambda v.M\ v$$

$$\text{Conversión} \Rightarrow \lambda v.M\ v =_\eta M$$

Si una función solo aplica M a su argumento, es equivalente a M directamente.

Ejemplos - Conversión η

```
(λx.f x) y  =η f y
```

```
(λx v.x v) ((λt.t t) w)
  =β λv.((λt.t t) w) v
  =β λv.(w w) v
  =η w w
```

```
(λv.w x y v) z  =η ?

λx.x t x  =η ?
```

Estrategias de Reducción

Call-by-name

Consiste en ir reduciendo siempre la β-redex más externa desde la izquierda y que no esté ubicada dentro de una abstracción lambda, hasta llegar a una expresión en forma normal de cabecera.

```
(λu.u (λt.t) ((λy.y) u)) ((λz.z) x)
  =β (λz.z) x (λt.t) ((λy.y) ((λz.z) x))
  =β x (λt.t) ((λy.y) ((λz.z) x))
```

Orden normal

Consiste en ir reduciendo siempre la β-redex más externa desde la izquierda.

```
(λu.u (λt.t) ((λy.y) u)) ((λz.z) x)
  =β (λz.z) x (λt.t) ((λy.y) ((λz.z) x))
  =β x (λt.t) ((λy.y) ((λz.z) x))
  =β x (λt.t) ((λz.z) x)
  =β x (λt.t) x
```

(+) Si tiene forma normal, siempre permite llegar a ella

Call-by-value

Consiste en ir reduciendo siempre la β-redex más interna desde la izquierda y que no esté ubicada dentro de una abstracción lambda.

```
(λu.u (λt.t) ((λy.y) u)) ((λz.z) x)
  =β (λu.u (λt.t) ((λy.y) u)) x
  =β x (λt.t) ((λy.y) x)
  =β x (λt.t) x
```

Orden aplicativo

Consiste en ir reduciendo siempre la β-redex más interna desde la izquierda.

```
(λu.u (λt.t) ((λy.y) u)) ((λz.z) x)
  =β (λu.u (λt.t) u) ((λz.z) x)
  =β (λu.u (λt.t) u) x
  =β x (λt.t) x
```

(*) Permite llegar a su forma normal de cabecera

Comparación

Call-by-name

```
(λu.(λt.t) ((λy.y) w)) ((λv.v) r) ((λz.z z) (λx.x x))
  =β (λt.t) ((λy.y) w) ((λz.z z) (λx.x x))
  =β (λy.y) w ((λz.z z) (λx.x x))
  =β w ((λz.z z) (λx.x x))
```

Se llega a la **forma normal de cabecera**

Orden normal

```
... =β w ((λz.z z) (λx.x x))
    =β w ((λx.x x) (λx.x x)) ...
```

Se produce un **ciclo infinito**

Comparación (cont.)

Call-by-value

```
(λu.(λt.t) ((λy.y) w)) ((λv.v) r) ((λz.z z) (λx.x x))
  =β (λu.(λt.t) ((λy.y) w)) r ((λz.z z) (λx.x x))
  =β (λt.t) ((λy.y) w) ((λz.z z) (λx.x x))
  =β (λt.t) w ((λz.z z) (λx.x x))
  =β w ((λz.z z) (λx.x x))
  =β w ((λx.x x) (λx.x x)) ...  → ciclo infinito
```

Orden aplicativo

```
(λu.(λt.t) ((λy.y) w)) ((λv.v) r) ((λz.z z) (λx.x x))
  =β (λu.(λt.t) w) ((λv.v) r) ((λz.z z) (λx.x x))
  =β (λu.w) ((λv.v) r) ((λz.z z) (λx.x x))
  =β (λu.w) r ((λz.z z) (λx.x x))
  =β w ((λz.z z) (λx.x x))
  =β w ((λx.x x) (λx.x x)) ...  → ciclo infinito
```

Resumen - Estrategias de Reducción

Estrategia	β-redex elegida	¿Entra en λ?	Forma normal
Call-by-name	Más externa, izquierda	No	De cabecera
Orden normal	Más externa, izquierda	Sí	Completa (garantizada si existe)
Call-by-value	Más interna, izquierda	No	De cabecera
Orden aplicativo	Más interna, izquierda	Sí	De cabecera

Funciones Lógicas

Funciones Lógicas

Mediante abstracciones es posible definir representaciones de los valores verdadero y falso, y funciones lógicas aplicables sobre ellos.

```
// Si p entonces q, si no r
If = λp.λq.λr.p q r
```

```
// Analogía con un lenguaje
if (p) {
  q;
} else {
  r;
}
```

Otras representaciones

```
True  = λx.λy.x
False = λx.λy.y
Not   = λp.p False True
And   = λp.λq.p q False
Or    = λp.λq.p True q
```

Funciones Numéricas y Relaciones

Números de Church

También mediante abstracciones se pueden definir numerales y funciones numéricas y relacionales aplicables sobre ellos.

```
0 = λf.λx.x
1 = λf.λx.f x
2 = λf.λx.f (f x)
3 = λf.λx.f (f (f x))
4 = λf.λx.f (f (f (f x)))
5 = λf.λx.f (f (f (f (f x))))
```

Operaciones

```
Succ   = λn.λf.λx.f (n f x)
Pred   = λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)
Add    = λm.λn.λf.λx.m f (n f x)
Sub    = λm.λn.n Pred m
Mul    = λm.λn.λf.λx.m (n f) x
Pow    = λm.λn.λf.λx.n m f x
```

```
Fibo   = λn.n (λf.λa.λb.f b (Add a b))
              (λx.λy.x) (λf.λx.x) (λf.λx.f x)

IsZero = λn.n (λz.(λx.λy.y)) (λx.λy.x)
```

Ejemplo - IsZero 4

Averiguar si el número 4 es cero

```
IsZero = λn.n (λz.(λx.λy.y)) (λx.λy.x)    4 = λf.λx.f (f (f (f x)))
```

```
(λn.n (λz.λx.λy.y) (λx.λy.x)) (λf.λx.f (f (f (f x))))
```

```
=β (λf.λx.f (f (f (f x)))) (λz.λx.λy.y) (λx.λy.x)
```

```
=β (λx.(λz.λx.λy.y) ((λz.λx.λy.y) ((λz.λx.λy.y)
    ((λz.λx.λy.y) x)))) (λx.λy.x)
```

```
=β (λz.λx.λy.y) ((λz.λx.λy.y) ((λz.λx.λy.y)
    ((λz.λx.λy.y) (λx.λy.x))))
```

```
=β λx.λy.y    // False
```

Ejemplo - Add 2 1

```
Add = λm.λn.λf.λx.m f (n f x)    2 = λf.λx.f (f x)    1 = λf.λx.f x
```

```
(λm.λn.λf.λx.m f (n f x)) (λf.λx.f (f x)) (λf.λx.f x)
```

```
=β (λn.λf.λx.(λf.λx.f (f x)) f (n f x)) (λf.λx.f x)
=β (λf.λx.(λf.λx.f (f x)) f ((λf.λx.f x) f x))
```

```
=β (λf.λx.(λx.f (f x)) ((λf.λx.f x) f x))
=β (λf.λx.(f (f ((λf.λx.f x) f x))))
```

```
=β (λf.λx.(f (f ((λx.f x) x))))
=β (λf.λx.(f (f (f x))))          // En los números de Church es 3 ✓
```

Combinadores

Cálculo de combinadores SKI

Las expresiones lambda que no contienen ninguna variable libre se denominan combinadores.

Combinador	Definición	Nombre
S	$\lambda x.\lambda y.\lambda z.x\ z\ (y\ z)$	Fusión *(Verschmelzungsfunktion)*
K	$\lambda x.\lambda y.x$	Constancia *(Konstanzfunktion)*
I	$\lambda x.x$	Identidad *(Identitätsfunktion)*

El cálculo de combinadores SKI es lo suficientemente potente como para codificar cualquier función computable.

Otros combinadores

```
B  = λx.λy.λz.x (y z)
C  = λx.λy.λz.x z y
D  = λx.λy.λz.λv.x y (x v z)
M  = λx.x x
K' = λx.λy.y
Y  = λf.(λx.f (x x)) (λx.f (x x))    // Punto fijo
Ω  = (λx.x x) (λx.x x)
```

Recursividad

Se representa utilizando un combinador de punto fijo, en este caso utilizamos Y:

```
Fact = Y (λf.λx.If (IsZero x) 1 (Mul x (f (Pred x))))
```

Expansión de Fact 3:

```
Fact 3
= Y (λf.λx.If (IsZero x) 1 (Mul x (f (Pred x)))) 3
= If (IsZero 3) 1 (Mul 3 (Fact (Pred 3)))
= Mul 3 (Fact 2)
= Mul 3 (Mul 2 (Fact 1))
= Mul 3 (Mul 2 (Mul 1 (Fact 0)))
= Mul 3 (Mul 2 (Mul 1 1))
= 6
```

Pares y Listas

Par ordenado

Un par ordenado está compuesto de dos elementos, denominados primero y segundo.

$$(a, b) = \lambda s.s\ a\ b$$

```
// Función para construir pares ordenados
Pair = λx.λy.λs.s x y
```

First y Second

```
First = λp.p True
```

```
// Ejemplo → First (Pair p q)
(λp.p (λx.λy.x)) ((λx.λy.λs.s x y) p q)
  =β (λp.p (λx.λy.x)) ((λy.λs.s p y) q)
  =β (λp.p (λx.λy.x)) (λs.s p q)
  =β (λs.s p q) (λx.λy.x)
  =β (λx.λy.x) p q
  =β (λy.p) q
  =β p
```

```
Second = λp.p False
```

```
// Ejemplo → Second (Pair p q)
(λp.p (λx.λy.y)) ((λx.λy.λs.s x y) p q)
  =β (λp.p (λx.λy.y)) ((λy.λs.s p y) q)
  =β (λp.p (λx.λy.y)) (λs.s p q)
  =β (λs.s p q) (λx.λy.y)
  =β (λx.λy.y) p q
  =β (λy.y) q
  =β q
```

Listas

```
// Lista vacía
Nil = Pair True True
// O también
Nil = λz.z

// La lista (a b c)
(False . (a . (False . (b . (False . (c . (λz.z)))))))
```

```
// Verificar si una lista está vacía
Null = First

// Construir un nodo de cabeza x y cola y
Cons = λx.λy.Pair False (Pair x y)

// Obtener la cabeza y la cola de una lista
Head = λz.First (Second z)
Tail = λz.Second (Second z)
```

Referencias y recursos

Paper original

- Church, A. (1936). *An Unsolvable Problem of Elementary Number Theory* — PDF (anotado) · PDF directo

Intérpretes online

- Lambda Calculus Interpreter — reducción β paso a paso con árbol de parseo
- Lambda Calculator — evaluación eager/lazy configurable
- λ-Calculus Playground — Church numerals y combinadores predefinidos

Resumen

- **Definición formal**: variables, abstracciones y aplicaciones
- **Conversiones**: α (renombrar), β (aplicar), η (extensionalidad)
- **Estrategias de reducción**: call-by-name, orden normal, call-by-value, orden aplicativo
- **Funciones lógicas**: True, False, If, Not, And, Or
- **Números de Church**: numerales y operaciones aritméticas
- **Combinadores**: SKI y punto fijo (Y)
- **Estructuras de datos**: pares ordenados y listas

¡Muchas Gracias!