.UBAfiuba
FACULTAD DE INGENIERÍA

Cálculo Lambda

TB025 - Paradigmas de Programación

Curso Essaya

Origen

En la década de 1930, varios matemáticos estaban interesados en la pregunta: ¿cuándo una función $f : \mathbb{N} \to \mathbb{N}$ es **computable**?

Es decir, ¿es posible calcular $f(n)$ para cualquier n dado, usando lápiz y papel?

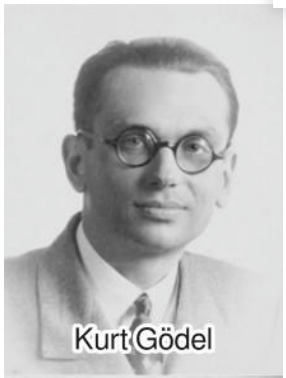
Se desarrollaron tres modelos de computabilidad por separado:

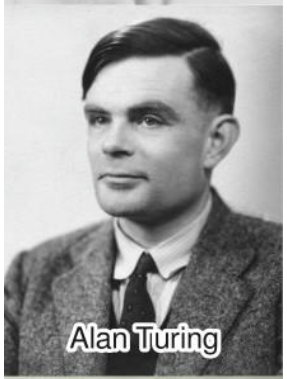
Kurt Gödel: Definió las **Funciones Recursivas Generales** y postuló que una función es computable si y solo si es recursiva general.

Alan Turing: Definió la **Máquina de Turing** y postuló que una función es computable si y solo si es computable por una máquina de Turing.

Alonzo Church: Definió el **Cálculo Lambda** y postuló que una función es computable si y solo si puede escribirse como una expresión lambda.

Luego se demostró que estos tres modelos son equivalentes (**Tesis de Church-Turing**).





El Cálculo Lambda jugó un papel importante en el desarrollo de la **teoría de los lenguajes de programación**, particularmente en el **paradigma de programación funcional**.

Funciones

El Cálculo Lambda incorpora dos simplificaciones para operar con funciones:

1. **Las funciones son anónimas.** Por ejemplo, la función

$$\text{suma\_cuadrados}(x, y) = x^2 + y^2$$

   se puede escribir como

$$(x, y) \mapsto x^2 + y^2$$

2. **Todas las funciones son de un solo argumento.** Por ejemplo, la función

$$(x, y) \mapsto x^2 + y^2$$

   se puede escribir como

$$x \mapsto (y \mapsto x^2 + y^2)$$

   Esta transformación se llama *currificación* (por Haskell Curry).

Cálculo Lambda

Consiste en escribir **expresiones lambda** con una sintaxis formal, y aplicar **reglas de reducción** para transformarlas.

Una **expresión lambda** puede ser:

- Una **variable**: `x`, `y`, `z`, ...
- Una **abstracción**: `(λx.M)` donde `M` es una expresión lambda.
- Una **aplicación**: `(M N)` donde `M` y `N` son expresiones lambda.

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

Una **abstracción** `(λx.M)` denota una **función anónima** que toma un argumento `x` y devuelve `M`, es decir $x \mapsto M$.

Una **aplicación** `(M N)` denota la acción de **invocar** la función `M` con el argumento `N`, es decir $M(N)$.

Convenciones

Para simplificar la notación:

1. Se pueden omitir los paréntesis externos: `M N ≡ (M N)`
2. Las aplicaciones se asocian a la izquierda: `M N P ≡ ((M N) P)`
3. El cuerpo de la abstracción se extiende todo lo posible hacia la derecha: `λx.M N ≡ λx.(M N)`
4. Se pueden contraer múltiples abstracciones lambda: `λx.λy.λz.N ≡ λx y z.N`
5. Cuando todas las variables son de una única letra, se pueden omitir los espacios: `MNP ≡ M N P`

Ejemplo:

`(((λx.(λy.(y x))) a) b)`

`((λx.(λy.(y x))) a) b` (1)

`(λx.(λy.(y x))) a b` (2)

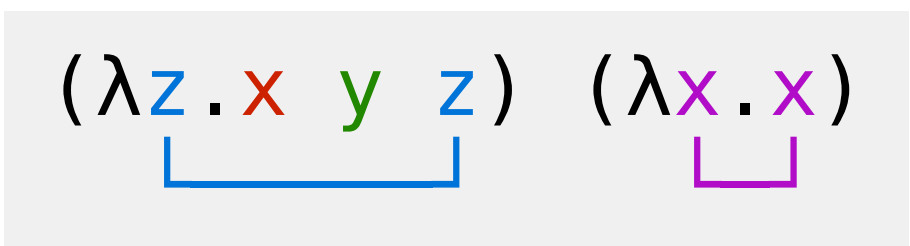
`(λx.λy.y x) a b` (3)

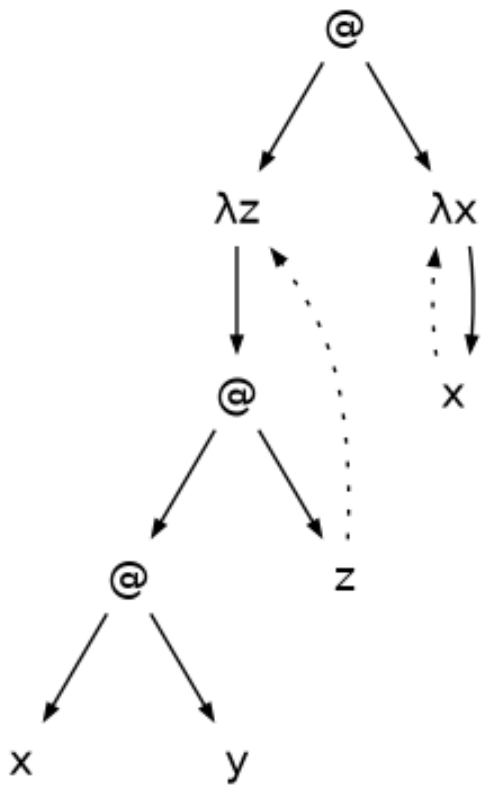
`(λx y.y x) a b` (4)

`(λxy.yx)ab` (5)

Variables libres y ligadas

Una abstracción `λx.M` **liga** las variables `x` que aparecen en el cuerpo `M`. Todas las demás variables en `M` son **libres**.





Reglas de reducción

Conversión Alfa (α): cambiar variables ligadas

> En la abstracción `λx.M`, se puede cambiar `x` por cualquier otra variable que no aparezca libre en `M`.
>
> Por ejemplo, `λx.x y =α λz.z y`.

Reducción Beta (β): aplicar una β-redex[1]

> Una **β-redex** es una aplicación de la forma `(λx.M) N`.
>
> Por ejemplo, `(λu.u z u) a =β a z a`.

Reducción Eta (η): Reducir una η-redex

> Una **η-redex** es una expresión de la forma `λx.M x` donde `x` no aparece libre en `M`.
>
> Según la regla η, `λx.M x =η M`.

[1] "redex" = expresión reducible.

Forma normal

Una expresión lambda está en:

Forma β-normal: si no contiene ninguna β-redex.

Ejemplo: `z (λx.y x) w`

Forma β-η-normal: si no contiene ninguna β-redex ni η-redex.

Ejemplo: `z (λx.x) w`

Forma normal de cabecera: si no hay una β-redex en la cabeza de la expresión.

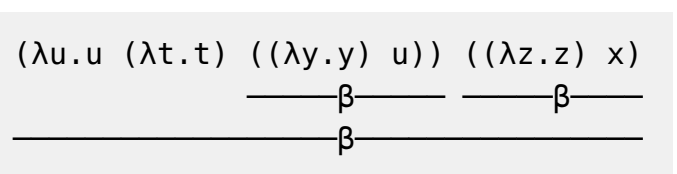
Ejemplo: `z ((λx.x) w)`

Para “evaluar” una expresión lambda, se aplican las reglas de reducción repetidamente hasta llegar a una forma normal.

> La forma normal de una expresión, si existe, es única.

Estrategias de reducción

Si una expresión contiene más de una redex, ¿cuál se debe reducir primero?



Entre las estrategias más comunes están:

Orden normal: siempre se reduce la β-redex más **externa** y a la izquierda.

> Si la expresión tiene forma normal, esta estrategia la encuentra.

Orden aplicativo: siempre se reduce la β-redex más **interna** y a la izquierda.

> Esta estrategia puede no terminar, incluso si la expresión tiene forma normal.

Call by value: Como el orden aplicativo, pero no se aplican reducciones dentro de abstracciones.

> Esta estrategia es la más común en lenguajes de programación tradiciconales como C: los argumentos de una función se evalúan antes de llamar a la función.

Call by name: Como el orden normal, pero no se aplican reducciones dentro de abstracciones.

> Esta estrategia y otras similares, son comunes en lenguajes con **evaluación perezosa** como Haskell: las funciones reciben expresiones sin evaluar, y las evalúan solo cuando las necesitan.

Codificación de tipos de datos

Booleanos

```
True  = λx y.x
False = λx y.y
```

```
Not = λp.p False True
And = λp q.p q False
Or  = λp q.p True q
If  = λp q r.p q r
```

Numerales de Church

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

```
Succ = λn.λf x.f (n f x)
Pred = λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)
```

```
Add = λm n.λf x.m f (n f x)
Sub = λm n.n Pred m
Mul = λm n.λf x.m (n f) x
Pow = λm n.λf x.n m f x
```

```
IsZero := λn.n (λx.False) True
Leq := λm n.IsZero (Sub m n)
```

Codificación de listas

Una forma de representar listas enlazadas es construyéndolas con *pares ordenados*: `(manzana, (pera, (banana, nil)))`.

Pares:

```
Pair = λx y.λs.s x y
Nil := Pair True True
```

```
First = λp.p True
Second = λp.p False
```

Listas:

La lista `[a b c]` se representa como `(False, (a, (False, (b, (False, (c, Nil))))))`

```
Null = IsEmpty = First
Cons = λh.λt.Pair False (Pair h t)
Head = λz.First (Second z)
Tail = λz.Second (Second z)
```

Así armamos la lista `[a b c]`: `(Cons a (Cons b (Cons c Nil)))`

Combinadores

> Una expresión lambda que no tiene variables libres es un **combinador**.

La **lógica combinatoria** es un modelo de computación equivalente al cálculo lambda pero más simple, ya que usa combinadores en lugar de abstracciones.

Por ejemplo, el **cálculo SKI** se basa en los combinadores:

```
I = λx.x                        I x = x
K = λx y.x                    K x y = x
S = λx y z.x z (y z)      S x y z = x z (y z)
```

Ejemplo: `SKSK = KK(SK) = K`

Recursión

¿Cómo podemos escribir funciones recursivas en el cálculo lambda?

De alguna manera necesitamos que la función pueda invocarse a sí misma. En los lenguajes tradicionales podemos usar el *nombre* de la función dentro del cuerpo de la misma función, pero en el cálculo lambda las funciones no tienen nombre, solo los argumentos.

> Un **combinador de punto fijo** es una función que recibe una función `f` y devuelve un *punto fijo*; es decir un valor `p` tal que `f p = p`.

Es decir, si `Fix` es un combinador de punto fijo:

```
Fix f = f (Fix f)
```

Recursión (cont.)

El combinador de punto fijo más conocido es el **combinador Y**:

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

Verificación:

```
Y g = (λx.g (x x)) (λx.g (x x))
    = g ((λx.g (x x)) (λx.g (x x)))
    = g (Y g)
```

El combinador Y permite escribir funciones recursivas. Por ejemplo:

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

Recursión (cont.)

El combinador Y produce *stack overflow* en lenguajes como C y Python.

El combinador Z es una variante que resuelve este problema:

```
Z = λf.(λx.f (λv.x x v)) (λx.f (λv.x x v))
```

Implementación del combinador Z en Python:

```python
Z = lambda f: (lambda x: f(lambda v: x(x)(v)))(lambda x: f(lambda v: x(x)(v)))

fact = Z(lambda f: lambda n: 1 if n == 0 else n * f(n - 1))

assert fact(5) == 120
```

```python
def Z(f):
    def g(x):
        def h(v):
            return x(x)(v)
        return f(h)
    return g(g)

def fact(f):
    def _fact(n):
        return 1 if n == 0 else n * f(n - 1)
    return _fact

fact = Z(fact)

assert fact(5) == 120
```

www.**ingenieria**.uba.ar

/ingenieriauba

/FIUBAoficial