

# Cálculo $\lambda$ y programación funcional

## Programación funcional para la física computacional

21 de febrero de 2023

### Máquinas de Turing

La máquina de Turing es uno de los modelos de computación más usados, es lo más cercano a una computadora (como la que usaron para bajar este pdf, y quizá hasta para leerlo), pero no es el único modelo. Ahora lo mencionó sólo para dar entrada a la idea del cálculo  $\lambda$ , para entender las cosas. No sé si será el método más pedagógico, o el más sencillo, pero es el estándar en la mayoría de libros del tema (además de que Turing es el más famoso de los matemáticos que trabajaron inicialmente en el área). Los modelos son equivalentes, así que no perderemos generalidad.

Una máquina de Turing se compone de un conjunto finito de estados  $Q$ , una cinta semi-infinita limitada por la izquierda con el símbolo  $\vdash$  e ilimitada por la derecha (este límite izquierdo es para saber donde empieza la cinta) y una cabeza que puede moverse a izquierda y derecha, capaz de leer y escribir caracteres en la cinta<sup>1</sup>. Las palabras de entrada, de longitud finita, se escriben sobre la cinta de izquierda a derecha (como escribimos nosotros). Al terminar la palabra de entrada en el resto de casillas de la cinta para distinguir que no contienen caracter alguno está pre-escrito el símbolo  $\_$ . Un esquema ejemplificando esta disposición se muestra en la figura ??.

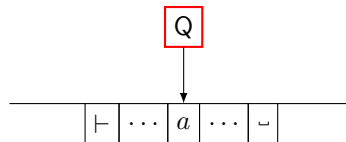


Figura 1: Esquema mecánico de una máquina de Turing.

Podemos dar una definición formal:

**Definición 1** Una máquina de Turing determinista, de cinta única es una 9-tupla (yo lo traduciría como un noneto o enéada, pero quizá no es la terminología) descrita como:

$$M = (Q, \Sigma, \Gamma, \vdash, \_, \delta, s, t, r)$$

con:

<sup>1</sup>Son muy jóvenes para recordar los casetes quizá, aunque por ahí hay unos intentos retros de revivirlos, pero si alguna vez han visto uno con su respectivo reproductor podrán notar que cuenta con una cinta magnética de color café o negro, que es leída por una cabeza también magnética. Era un formato de no muy buena calidad, de riesgo pues un imán de potencia suficiente podía dañar la cinta y los carretes podían provocar accidentes como enrollarse o atorarse. Hubo computadoras que usaban casetes para leer programas, para una crónica al respecto echen un ojo a: <https://www.xataka.com/historia-tecnologica/cuando-los-videojuegos-venian-en-cassette-y-habia-que-rebobinarlos-para-poder-jugar>

- $Q$  es el conjunto finito de estados
- $\Sigma$  es el alfabeto de entrada (finito)
- $\Gamma$  es el alfabeto de cinta (finito), con  $\Sigma \subseteq \Gamma$
- $\vdash \in \Gamma - \Sigma$  el símbolo de inicio de la cinta
- $\sqcup \in \Gamma - \Sigma$  el símbolo de espacio en blanco
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\rightarrow, \leftarrow\}$  la función de transición
- $s \in Q$  el estado inicial
- $t \in Q$  el estado de aceptación
- $r \in Q$  el estado de rechazo.

### Un ejemplo

Sea la máquina de Turing que decida el lenguaje  $A = \{0^{2^n} | n \geq 0\}$ , es decir, el lenguaje con puras cadenas de 0's que su longitud sea una potencia de 2.

	$\vdash$	0	$x$	$\sqcup$
$s$	$(s, \vdash, \rightarrow)$	$(q_1, \sqcup, \rightarrow)$	$(r, -, -)$	$(r, -, -)$
$q_1$		$(q_2, x, \rightarrow)$	$(q_1, x, \rightarrow)$	$(t, -, -)$
$q_2$		$(q_3, 0, \rightarrow)$	$(q_2, x, \rightarrow)$	$(q_4, \sqcup, \leftarrow)$
$q_3$		$(q_2, x, \rightarrow)$	$(q_3, x, \rightarrow)$	$(r, -, -)$
$q_4$		$(q_4, 0, \leftarrow)$	$(q_4, x, \leftarrow)$	$(q_1, \sqcup, \rightarrow)$

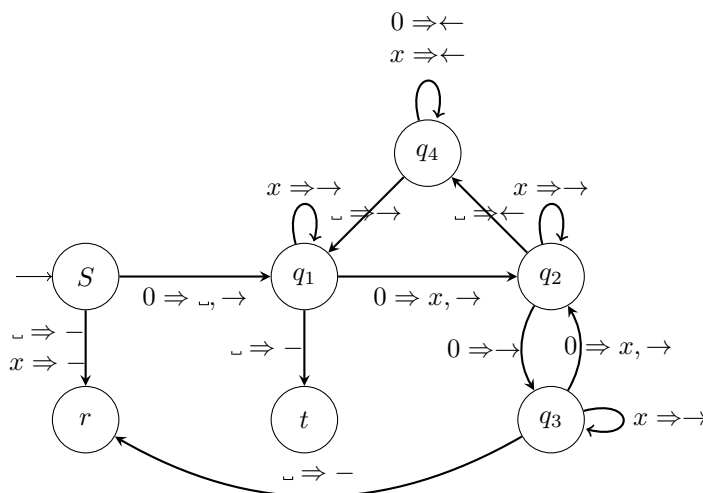


Figura 2: Diagrama de estados de la máquina de Turing descrita.

## Otros modelos computacionales

El modelo de máquina de Turing ajusta muy bien para cuando se trabaja de manera teórica con computadoras, pero quizá pueda parecer demasiado mecánico y no se parece a como operamos con computadoras hoy en día. Existen más modelos que sonm equivalentes, sabemos de esa equivalencia por la **tesis Church-Turing**.

**Tesis de Church-Turing 1** *Toda función es efectivamente calculable sí y sólo sí es calculable por una máquina de Turing.*

En esos otros modelos se encuentran las funciones recursivas  $\mu$ , relacionadas directamente al trabajo de Gödel al preguntarse ¿cuál es el mínimo de funciones necesarias para definir a todas las funciones computables?

1. *Sucesor*. La función  $s : \mathbb{N} \rightarrow \mathbb{N}$  dadas por  $s(x) = x + 1$  es computable.
2. *Cero*. La función  $z : \mathbb{N}^0 \rightarrow \mathbb{N}$  dada por  $F() = 0$  es computable.
3. *Proyecciones*. Las funciones  $\pi_k^n : \mathbb{N}^n \rightarrow \mathbb{N}$  dadas por  $\pi_k^n(x_1, \dots, x_n) = x_k$ ,  $1 \leq k \leq n$ , son computables.
4. *Composición*. Si  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  y  $g_1, \dots, g_k : \mathbb{N}^n \rightarrow \mathbb{N}$  son computables, entonces también lo es la función  $f \circ (g_1, \dots, g_k) : \mathbb{N}^n \rightarrow \mathbb{N}$  que en la entrada  $\bar{x} = x_1, \dots, x_n$ , da

$$f(g_1(\bar{x}), \dots, g_k(\bar{x}))$$

5. *Recursión primitiva*. Si  $h_i : \mathbb{N}^n \rightarrow \mathbb{N}$  y  $g_i : \mathbb{N}^{n+k+1} \rightarrow \mathbb{N}$  son computables,  $1 \leq i \leq k$ , entonces también lo son las funciones  $f_i : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ ,  $1 \leq i \leq k$ , definidas por inducción mutua de la siguiente manera:

$$f_i(0, \bar{x}) \stackrel{def}{=} h_i(x)$$
$$f_i(x + 1, \bar{x}) \stackrel{def}{=} g_i(x, \bar{x}, f_1(x, \bar{x}), \dots, f_k(x, \bar{x})),$$

donde  $\bar{x} = x_1, \dots, x_n$ .

6. *Minimización no acotada*. Si  $g : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  es computable, entonces también lo es la función  $f : \mathbb{N}^n \rightarrow \mathbb{N}$  que con la entrada  $\bar{x} = x_1, \dots, x_n$  de al menos  $y$  tal que  $g(z, \bar{x})$  esté definida para todas las  $z \leq y$  y  $g(y, \bar{x}) = 0$  si tal  $y$ , y está indefinida de otra manera. Esto se denota como:

$$f(\bar{x}) = \mu y. (g(y, \bar{x}) = 0)$$

## Cálculo $\lambda$

Ya en la introducción habíamos mencionado una forma de ver a las funciones matemáticas como cajas negras, figura . Ahora vamos un paso adelante a definir de manera abstracta lo que es una función, una *abstracción funcional* y como aplicamos las funciones desde una perspectiva abstracta y meramente matemática.

Las características del cálculo  $\lambda$  pueden listarse:

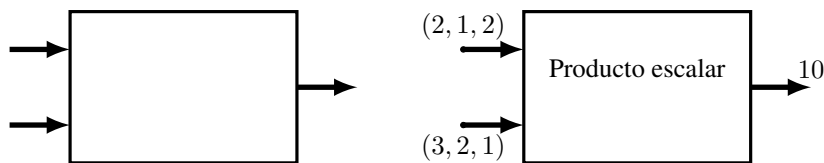


Figura 3: Una función representada como caja negra.

- Solo depende de funciones. Si no se puede escribir como función, no se puede incluir.
- No tiene estado o efectos laterales
- El orden de evaluación no es relevante
- Todas las funciones son unitarias, sólo toman un argumento

El cálculo  $\lambda$  fue iniciado en la década de los 30s del siglo  $XX$  por Alonso Church. No fue la única persona trabajando en el área, Haskell Curry trabajo en la lógica combinatoria, Gödel en las funciones recursivas  $\mu$  en las que podemos encontrar algunas similitudes, Emil Post en el sistema canónico Post, o sistema de reescritura y Alan Turing con las máquinas de Turing. Todos ellos de alguna manera trataron de definir qué era un algoritmo y darle una estructura formal<sup>2</sup>. Las funciones recursivas  $\mu$  y las máquinas de Turing ya las vimos de rápido, en lo que sigue nos centraremos en la parte del trabajo de Church y Curry. Pero para ser justos hay que mencionar como funciona el sistema canónico de Post.

Se parte de una cantidad finitamente grande de cadenas de caracteres que serán manipuladas, de manera repetida son transformadas al aplicar un conjunto finito de reglas, formando un lenguaje regular. Al ser Turing completo, es equivalente a trabajar con máquinas de Turing. Ya sólo se estudia con interés histórico pues hay formalismos que incluyen las ideas de Post más usados hoy en día.

La idea de Post al crear el cálculo  $\lambda$  era demostrar que el problema de la decisión de Hilbert no era resoluble por la aritmética de Peano. Lo que obtuvo fue un sistema para estudiar las matemáticas de una forma más general.

Con las reglas antes mencionadas sólo resta decir que para construir todas las funciones matemáticas en el cálculo  $\lambda$  basta usar tres operaciones:

- Crear funciones que recibirán variables
- Asociar una variable a la expresión
- Aplicar una función a un argumento

¿Porque la diferencia entre asociar una variable y aplicar a un argumento? Tal como cuando definimos una función matemática del estilo  $f(x) = x^2 - 3x$  la variable  $x$  es más un marco donde eventualmente puede entrar un valor, a evaluar la ecuación en un número es que estamos aplicando la función a un argumento. Las variables son términos  $\lambda$  y son la base de las definiciones deductivas del cálculo  $\lambda$ .

<sup>2</sup>Como vimos en la sección pasada los algoritmos existen incluso desde hace 2000 años, pero no había una teoría formal de ellos.

En el trato normal de las variables en el cálculo  $\lambda$  los tipos no entran en juego, está des-tipado (no sé si así se dice, pero suena cagado).

Imaginemos que se tienen dos términos  $\lambda M$  y  $N$ ,  $MN$  también es un término  $\lambda$ , podría ser por ejemplo que  $M$  es una función y  $N$  una entrada (podría ser también una función). La forma más formal de escribirlo es  $(M)N$  donde se dice que  $N$  es una entrada aplicada a la función  $M$ . El paréntesis es operador *aplicación* y se asocia a la izquierda  $(MN)P$ .

Pero realmente el orden de la aplicación es de izquierda a derecha, por ejemplo, si tenemos  $\lambda x.E(x)$  lo que quiere decir es que con la entrada  $x$  la función calcula  $E(x)$ .

Veamos un ejemplo concreto. Se tiene:

1 `inc(x) = x+1`

Lo que quiere decir que para incrementar  $x$  se le suma 1

2 `(x) -> x+1`

Pero usando ya directamente la abstracción

$$\lambda x.(x + 1)$$

Si ahora tenemos la función

3 `sumcuad(x, y) = x^2+y^2`

Que se traduce a

4 `(x, y) -> x^2+y^2`

¿Cómo se pondría en la abstracción?

$$\lambda x.(\lambda y.(x^2 + y^2))$$

¡Muy bien! ¿Y ahora como se evalúa (aplicar a un argumento)? Para el caso del incremento al número 8:

$$\lambda x.(x + 1)8$$

Vamos a un caso más abstracto, sea la composición de las funciones  $f$  y  $g$

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

Pero en ese caso solo estamos dando como entrada a  $x$  ¿y si quiero dejarlo como el esqueleto de una composición done también puedo darle como entrada las funciones  $f$  y  $g$ ?

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

Con ese esqueleto ahora hagamos la composición de la función sucesor:

$$\begin{aligned} & (\lambda f.\lambda g.\lambda x.f(gx))(\lambda y.(y + 1))(\lambda z.(z + 1)) \\ & \rightarrow (\lambda g.\lambda x.((\lambda y.(y + 1))(gx)))(\lambda z.(z + 1)) \\ & \rightarrow \lambda x.((\lambda y.(y + 1))((\lambda z.(z + 1))x)) \\ & \rightarrow \lambda x.((\lambda y.(y + 1))(x + 1)) \\ & \rightarrow \lambda x.((x + 1) + 1) \end{aligned}$$

## Calculo $\lambda$ tipado

Una variante del cálculo  $\lambda$  admite que se especifique el tipo de las variables a dar como argumento, así las funciones definidas se restringen al dominio dado.

Si en la función incremento sólo queremos operar sobre naturales

$$\lambda x : \nu.(x + 1)$$

Eso es el estilo Church, se infiere que al ser la entrada naturales la salida son naturales, pero el estilo Curry hace que toda la función sea definida dentro de los naturales

$$(\lambda x.(x + 1))\nu \rightarrow \nu$$

Si hay más de una variable la versión Post sería

$$\lambda x : \nu.(\lambda y : \nu.(x^2 + y^2))$$

Y la versión Curry

$$\lambda x.(\lambda y.(x^2 + y^2)) : \nu \rightarrow \nu \rightarrow \nu$$

## Operaciones de reducción

### Programación funcional en *Python*

Declaraciones en lugar de procedimientos paso a paso.

Le decimos que hacer pero no como hacerlo.

Haskell: datos inmutables

Vemos en python:

```
1 x=1
2 oldID=id(x)
3 x=x+1
4 id(x)==oldID
```

La falta de estado, como sucede en Haskell, conlleva la desventaja de la falta de memoria.

## Funciones en Haskell

Dos opciones, *curriado* y *no curriado*.

La definición de una *\*non-curried\**

```
5 suma (x,y) = x+y
```

A partir de la función suma definamos la función sucesor:

```
6 suc (x) = suma (x,1)
```

Si esto mismo tratamos de hacer en *python*, para definir una operación que deja sin cambio una variable, se tiene que hacer:

```

1 def DoChange(x, y):
2     x = x.__add__(y)
3     return x
4
5 x = 1
6 print(x)
7 print(DoChange(x, 2))
8 print(x)

```

Es una definición especial de la suma que lo permite, pero no es la norma, por ejemplo:

```

1 def DoChange(aList):
2     aList.append(4)
3     return aList
4
5 aList = [1, 2, 3]
6 print(aList)
7 print(DoChange(aList))
8 print(aList)

```

Regresando a Haskell, definamos las operaciones de suma e incremento de forma *curriada*, es decir, definimos una función que sólo necesita un argumento y regresa otra función. Cierra **GHCi** y vuelve a abrir. Ahora define:

```

7 suma x y = x+y

```

Aquí parece que solo quitamos los paréntesis, pero no. Para notar la diferencia ahora definimos la función incremento como:

```

8 inc = suma 1

```

Se puede usar *\*curry\** y *\*uncurry\** para pasar de una a la otra. Aprovecha para jugar con *map*.

Como haríamos esto mismo en *python*:

```

5 def suma(x,y):
6     return x+y

```

Para definir el incremento en uno:

```

7 def inc(x):
8     return suma (x,1)

```

## Referencias

- [1] Thompson, Simon J.. "Haskell - the craft of functional programming." International computer science series (1996).
- [2] Van Roy, Peter. "Programming paradigms for dummies: what every programmer should know." (2009).
- [3] Van Roy, Peter, Haridi, Seif. "Concepts, Techniques, and Models of Computer Programming". \*The MIT Press\*, (2004). ISBN: 0262220695
- [4] Kowalski, Robert. "The early years of logic programming", *Communications of the ACM*, **31**, 1, (1988).
- [5] <https://realpython.com/python-concurrency/>, revisado el 15 de febrero de 2023.