Programación funcional II

Ricardo Pérez López

IES Doñana, curso 2020/2021

Índice general

1. Abstracciones funcionales

1.1. Expresiones lambda

Las **expresiones lambda** (también llamadas **abstracciones lambda** o **funciones anónimas** en algunos lenguajes) son expresiones que capturan la idea abstracta de «**función**».

Son la forma más simple y primitiva de describir funciones en un lenguaje funcional.

Su sintaxis (simplificada) es:

⟨*expr_lambda*⟩ ::= **`lambda`** [⟨*lista_parámetros*⟩]**`:`** ⟨*expresión*⟩
⟨*lista_parámetros*⟩ := ⟨*identificador*⟩ (**`,`** ⟨*identificador*⟩)*

Por ejemplo:

```
lambda x, y: x + y
```

1.1.1. Parámetros y cuerpos

Los identificadores que aparecen entre la palabra clave `lambda` y el carácter de dos puntos (`:`) son los **parámetros** de la expresión lambda.

La expresión que aparece tras los dos puntos (`:`) es el **cuerpo** de la expresión lambda.

En el ejemplo anterior:

```
lambda x, y: x + y
```

- Los parámetros son `x` e `y`.
- El cuerpo es `x + y`.
- Esta expresión lambda captura la idea general de sumar dos valores (que en principio pueden ser de cualquier tipo, siempre y cuando admitan el operador `+`).

1.1.2. Aplicación funcional

De la misma manera que decíamos que podemos aplicar una función a unos argumentos, también podemos aplicar una expresión lambda a unos argumentos.

Recordemos que *aplicar* una función a unos argumentos produce el valor que la función asocia a esos argumentos en el conjunto imagen.

Por ejemplo, la aplicación de la función *max* sobre los argumentos 3 y 5 se escribe como $max(3, 5)$ y eso denota el valor 5.

Igualmente, la aplicación de una expresión lambda como

```
lambda x, y: x + y
```

sobre los argumentos `4` y `3` se representa así:

```
(lambda x, y: x + y)(4, 3)
```

1.1.2.1. Llamadas a funciones

Si hacemos la siguiente definición:

```
suma = lambda x, y: x + y
```

a partir de ese momento podemos usar `suma` en lugar de su valor (la expresión lambda), por lo que podemos hacer:

```
suma(4, 3)
```

en lugar de

```
(lambda x, y: x + y)(4, 3)
```

Cuando aplicamos a sus argumentos una función así definida también podemos decir que estamos **invocando** o **llamando** a la función. Por ejemplo, en `suma(4, 3)` estamos *llamando* a la función `suma`, o hay una *llamada* a la función `suma`.

1.1.2.2. Evaluación de una aplicación funcional

En nuestro modelo de sustitución, la **evaluación de la aplicación de una expresión lambda** consiste en **sustituir**, en el cuerpo de la expresión lambda, **cada parámetro por su argumento correspondiente** (por orden) y devolver la expresión resultante *parentizada* (entre paréntesis).

A esta operación se la denomina **aplicación funcional** o **β-reducción**.

Siguiendo con el ejemplo anterior:

```
(lambda x, y: x + y)(4, 3)
```

sustituimos en el cuerpo de la expresión lambda los parámetros `x` e `y` por los argumentos `4` y `3`, respectivamente, y parentizamos la expresión resultante, lo que da:

```
(4 + 3)
```

que simplificando (según las reglas del operador `+`) da `7`.

Lo mismo podemos hacer si definimos previamente la expresión lambda ligándola a un identificador:

```
suma = lambda x, y: x + y
```

Así, la aplicación de la expresión lambda resulta más fácil y clara de escribir:

```
suma(4, 3)
```

En ambos casos, el resultado es el mismo (7).

Importante:

En **Python**, salvo excepciones, los operandos y los argumentos de las funciones se evalúan **de izquierda a derecha**.

1.1.2.3. Ejemplos

Dado el siguiente código:

```
suma = lambda x, y: x + y
```

¿Cuánto vale la expresión siguiente?

```
suma(4, 3) * suma(2, 7)
```

Según el modelo de sustitución, reescribimos:

```
suma(4,3) * suma(2, 7)
= (lambda x, y: x + y)(4, 3) * suma(2, 7)  # definición de suma
= (4 + 3) * suma(2, 7)                     # aplicación a 4, 3
= 7 * suma(2, 7)                           # aritmética
= 7 * (lambda x, y: x + y)(2, 7)           # definición de suma
= 7 * (2 + 7)                              # aplicación a 2, 7
= 7 * 9                                    # aritmética
= 63
```

1.1.3. Variables ligadas y libres

Si un identificador aparece en la lista de parámetros de una expresión lambda, a ese identificador le llamamos **variable ligada** de la expresión lambda.

En caso contrario, le llamamos **variable libre** de la expresión lambda.

En el ejemplo anterior:

```
lambda x, y: x + y
```

los dos identificadores que aparecen en el cuerpo (x e y) son variables ligadas, ya que ambos aparecen en la lista de parámetros de la expresión lambda.

En cambio, en la expresión lambda:

```
lambda x, y: x + y + z
```

x e y son variables ligadas mientras que z es libre.

1.1.4. Ámbitos

Recordemos que el **ámbito de una ligadura** es la porción del programa en la que dicha ligadura tiene validez.

Ampliaremos ahora el concepto de *ámbito* para incluir los aspectos nuevos que incorporan las expresiones lambda.

1.1.4.1. Ámbito de una variable ligada

Hemos visto que **un parámetro** de una expresión lambda **es una variable ligada** en el cuerpo de dicha expresión lambda.

En realidad, lo que hace la expresión lambda es **ligar al parámetro con la variable ligada que está dentro del cuerpo**, y esa ligadura existe únicamente en el cuerpo de la expresión lambda.

Por tanto, **el ámbito de una variable ligada es el cuerpo de la expresión lambda** que la liga con su parámetro.

También se dice que la variable ligada tiene un **ámbito local** a la expresión lambda o que es **local** a dicha expresión lambda.

Por contraste, los identificadores (y ligaduras) que no tienen ámbito local se dice que tienen un **ámbito no local** o, a veces, un **ámbito más global**.

Si, además, ese ámbito resulta ser el **ámbito global**, decimos directamente que el identificador (o la ligadura) es **global**.

Por ejemplo:

```
1 # Aquí empieza el script (no hay más definiciones antes de esta línea):
2 producto = lambda x: x * x
3 y = producto(3)
4 z = x + 1       # da error
```

La expresión lambda de la línea 2 tiene un parámetro (x) ligado a la variable ligada x situada en el cuerpo de la expresión lambda.

Por tanto, el ámbito de la variable ligada x es el **cuerpo** de la expresión lambda (x * x).

Eso quiere decir que, fuera de la expresión lambda, no es posible acceder al valor de la variable ligada, al encontrarnos **fuera de su ámbito**.

Por ello, la línea 4 dará un error al intentar acceder al valor de un identificador no ligado.

1.1.4.2. Ámbitos, marcos y entornos

Recordemos que un marco es un conjunto de ligaduras.

Y que un entorno es una secuencia de marcos que contienen todas las ligaduras válidas en un punto concreto de la ejecución del programa.

Cuando la ejecución del programa entra dentro de un ámbito, **se crea un nuevo marco asociado a ese ámbito**.

Ahora hemos visto que **cada expresión lambda define un nuevo ámbito**.

Por tanto, cuando se aplica una expresión lambda a unos argumentos, se crea un nuevo marco que contiene las ligaduras que define dicha expresión lambda con sus argumentos.

Ese nuevo marco se enlaza con el marco del ámbito que lo contiene (el marco más interno *apunta* al más externo), de manera que el último siempre es el marco global.

El marco desaparece cuando el flujo de control del programa se sale del ámbito, ya que cada marco va asociado a un ámbito.

Se va formando así una cadena de marcos que representa el **entorno** del programa en un punto dado del mismo.

A partir de ahora ya no vamos a tener un único marco (el *marco global*) sino que tendremos, además, al menos uno más cada vez que se aplique una expresión lambda a unos argumentos.

El **ámbito** es un concepto *estático*: es algo que existe y se reconoce simplemente leyendo el código del programa, sin tener que ejecutarlo.

El **marco** es un concepto *dinámico*: es algo que se crea y se destruye a medida que vamos entrando o saliendo de un ámbito, y contiene las ligaduras que se definen dentro de ese ámbito.

Por ejemplo:

```
suma = lambda x, y: x + y
```

el cuerpo de la función `suma` define un nuevo ámbito, y cada vez que se llama a `suma` con unos argumentos concretos se crea un nuevo marco que liga sus argumentos con sus parámetros.

El concepto de **entorno** refleja el hecho de que los ámbitos se contienen unos a otros (están anidados unos dentro de otros).

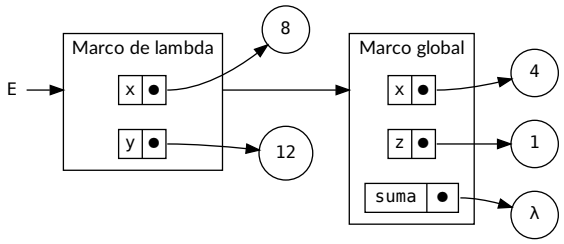
Si un marco *A* apunta a un marco *B*, significa que el ámbito de *A* está contenido en el ámbito de *B*.

Asimismo, el primer marco de la cadena de marcos del entorno representa el ámbito actual de la porción de código donde se está calculando el entorno.

El último marco siempre es el marco global.

En realidad, el marco global apunta, a su vez, al marco donde se encuentran las definiciones internas predefinidas del lenguaje (como la función `max`), pero lo ignoraremos de aquí en adelante por simplicar.

Si en un determinado punto del programa tenemos el siguiente entorno:



Podemos afirmar que:

- El ámbito de la expresión lambda está contenido en el ámbito global.
- El marco actual es el marco de la expresión lambda.
- El ámbito actual es el cuerpo de la expresión lambda.
- Por tanto, el programa se encuentra actualmente ejecutando el cuerpo de la expresión lambda.

1.1.4.3. Ligaduras *sombreadas*

¿Qué ocurre cuando una expresión lambda contiene como parámetros nombres que ya están definidos (ligados) en el entorno, en un ámbito más global?

Por ejemplo:

```
1 x = 4
2 total = (lambda x: x * x)(3)  # Su valor es 9
```

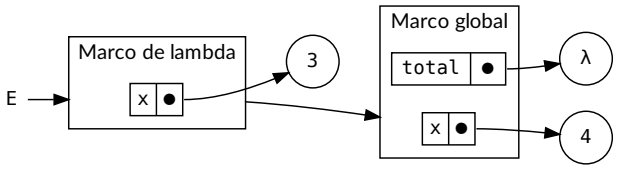
La x que aparece en la línea 1 es distinta a la que aparece en la línea 2.

El identificador x que aparece en el cuerpo de la expresión lambda **hace referencia al parámetro x de la expresión lambda**, y **no** al identificador x que está fuera de la expresión lambda (y que aquí está ligado al valor 4).

En este caso, decimos que **el parámetro x *hace sombra*** al identificador x global, y decimos que ese identificador está **sombreado** o que su ligadura está **sombreada**.

Que el parámetro haga sombra al identificador de fuera significa que no podemos acceder a ese identificador externo desde el cuerpo de la expresión lambda como si fuera una variable libre.

Esto es así porque la primera ligadura del identificador x que nos encontramos al recorrer la secuencia de marcos del entorno es la que está en el marco de la expresión lambda.

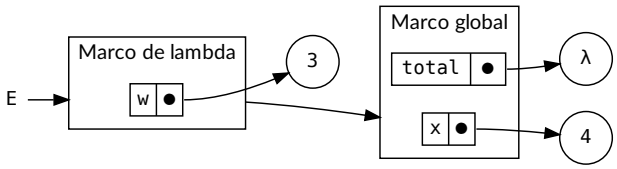


Entorno en el cuerpo de la expresión lambda, con variable sombreada

Si necesitáramos acceder, desde el cuerpo de la expresión lambda, al valor de la x que está fuera de la expresión lambda, lo que podemos hacer es **cambiar el nombre** al parámetro x. Por ejemplo:

```
x = 4
total = (lambda w: w * x)(3)  # Su valor es 12
```

Así, tendremos en la expresión lambda una variable ligada (el parámetro w) y una variable libre (el identificador x ligado en el ámbito global) al que ahora sí podemos acceder al no estar sombreada.



Entorno en el cuerpo de la expresión lambda, sin variable sombreada

1.1.4.4. Renombrado de parámetros

Los parámetros se pueden *renombrar* (siempre que se haga de forma adecuada) sin que se altere el significado de la expresión lambda.

A esta operación se la denomina **α-conversión**.

Un ejemplo de α-conversión es la que hicimos antes.

La α-conversión hay que hacerla correctamente para evitar efectos indeseados. Por ejemplo, en:

```
lambda x, y: x + y + z
```

si renombramos x a z tendríamos:

```
lambda z, y: z + y + z
```

lo que es claramente incorrecto. A este fenómeno indeseable se le denomina **captura de variables**.

1.1.4.5. Expresiones lambda y entornos

Para encontrar el valor ligado a un identificador en el entorno, buscamos **en el primer marco del entorno** una ligadura para ese identificador, y si no la encontramos, **vamos subiendo por la cadena de marcos** hasta encontrarla. Si no aparece en ningún marco, querrá decir que el identificador no está ligado, o que su ligadura está fuera del entorno, en otro ámbito.

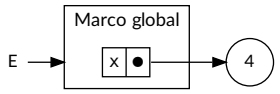
Debemos tener en cuenta también, por tanto, las posibles **variables sombreadas** que puedan aparecer.

Si un identificador en un ámbito más local *hace sombra* a otro en un ámbito más global, al buscar una ligadura en la cadena de marcos (en el entorno) se encontrará primero la ligadura más local, ignorando las otras.

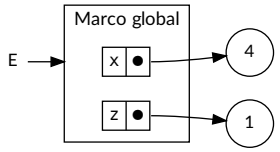
Por ejemplo:

```
1 x = 4
2 z = 1
3 suma = (lambda x, y: x + y + z)(8, 12)
4 y = 3
5 w = 9
```

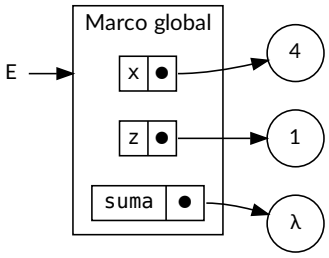
A medida que vamos ejecutando cada línea del código, tendríamos los siguientes entornos:



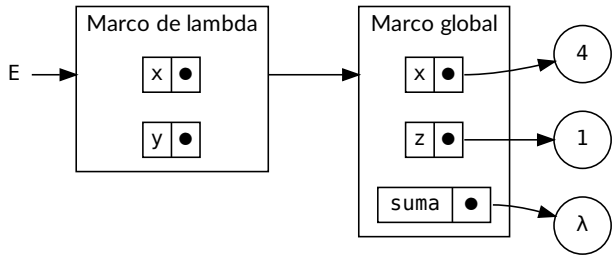
Entorno en la línea 1



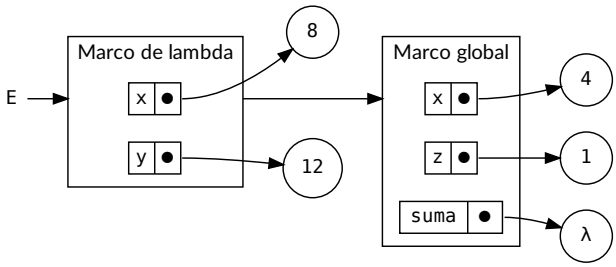
Entorno en la línea 2



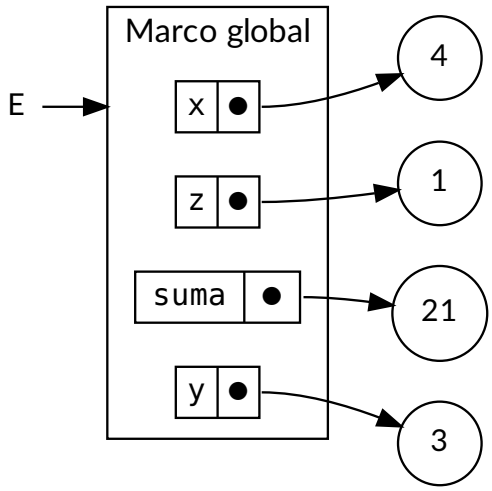
Entorno en la línea 3 fuera de la expresión lambda



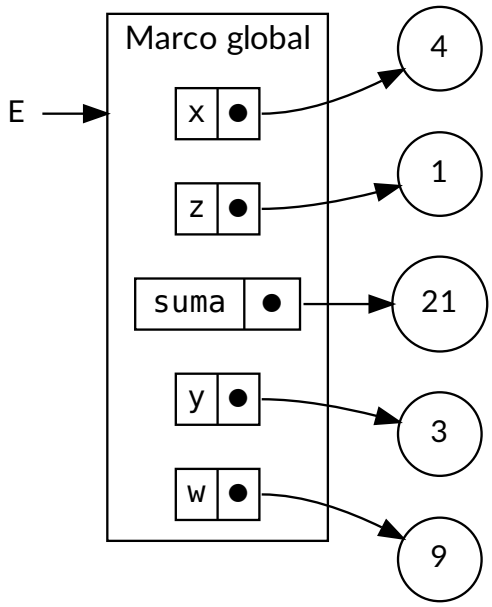
Entorno en la línea 3 en el cuerpo de la expresión lambda, **antes** de aplicar los argumentos



Entorno en la línea 3 en el cuerpo de la expresión lambda, **después** de aplicar los argumentos



Entorno en la línea 4



Entorno en la línea 5

1.1.4.6. Evaluación de expresiones lambda con entornos

Para que una expresión lambda funcione, sus variables libres deben estar ligadas a algún valor en el entorno **en el momento de *evaluar* una aplicación de la expresión lambda sobre unos argumentos**.

Por ejemplo:

```
1 >>> prueba = lambda x, y: x + y + z  # aquí no da error
2 >>> prueba(4, 3)                    # aquí sí
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   File "<stdin>", line 1, in <lambda>
6 NameError: name 'z' is not defined
```

da error porque `z` no está definido (no está ligado a ningún valor en el entorno) en el momento de llamar a `prueba` en la línea 2.

En cambio:

```
1 >>> prueba = lambda x, y: x + y + z
2 >>> z = 9
3 >>> prueba(4, 3)
4 16
```

sí funciona (y devuelve `16`) porque, en el momento de evaluar la aplicación de la expresión lambda (en la línea 3), el identificador `z` está ligado a un valor en el entorno (en este caso, `9`).

Observar que no es necesario que las variables libres estén ligadas en el entorno cuando *se crea* la expresión lambda, sino cuando **se evalúa el cuerpo de la expresión lambda**, o sea, cuando se aplica la expresión lambda a unos argumentos.

1.1.5. Pureza

Si el cuerpo de una expresión lambda no contiene variables libres, el valor que obtendremos al aplicarla a unos argumentos dependerá únicamente del valor que tengan esos argumentos (no dependerá de nada más que sea «*exterior*» a la expresión lambda).

En cambio, si el cuerpo de una expresión lambda sí contiene variables libres, el valor que obtendremos al aplicarla a unos argumentos no sólo dependerá del valor de esos argumentos, sino también de los valores a los que estén ligadas las variables libres en el momento de evaluar la aplicación de la expresión lambda.

Es el caso del ejemplo anterior, donde tenemos una expresión lambda que contiene una variable libre (`z`) y, por tanto, cuando la aplicamos a los argumentos `4` y `3` obtenemos un valor que depende, no sólo de los valores de `x` e `y`, sino también del valor de `z`:

```
>>> prueba = lambda x, y: x + y + z
>>> z = 9
>>> prueba(4, 3)
16
```

En este otro ejemplo, escribimos una expresión lambda que calcula la suma de tres números a partir de otra expresión lambda que calcula la suma de dos números:

```
suma = lambda x, y: x + y
suma3 = lambda x, y, z: suma(x, y) + z
```

En este caso, hay un identificador (`suma`) que no aparece en la lista de parámetros de la expresión lambda `suma3`, por lo que es una variable libre en el cuerpo de la expresión lambda de `suma3`.

En consecuencia, el valor de dicha expresión lambda dependerá de lo que valga `suma` en el entorno actual.

Se dice que una expresión lambda es **pura** si, siempre que la apliquemos a unos argumentos, el valor obtenido va a depender únicamente del valor de esos argumentos, es decir, de sus parámetros o variables ligadas.

También se dice que una expresión lambda que contiene sólo variables ligadas es **más pura** que otra que también contiene variables libres.

Y también podemos hablar de **grados de pureza**:

- Podemos decir que hay **más pureza** si una variable libre representa una **función** a aplicar en el cuerpo de la expresión lambda, que si representa cualquier otro tipo de valor.
- En el ejemplo anterior, tenemos que la expresión lambda de `suma3`, sin ser *totalmente pura*, a efectos prácticos se la puede considerar **pura**, ya que su única variable libre (`suma`) se usa como una **función**, y las funciones tienden a no cambiar durante la ejecución del programa, al contrario que los demás tipos de valores.

Por ejemplo, las siguientes expresiones lambda están ordenadas de mayor a menor pureza, siendo la primera totalmente **pura**:

```
# producto es una expresión lambda totalmente pura:
producto = lambda x, y: x * y
# cuadrado es casi pura; a efectos prácticos se la puede
# considerar pura ya que sus variables libres (en este
# caso, sólo una: producto) son funciones:
cuadrado = lambda x: producto(x, x)
# suma es impura, porque su variable libre no es una función:
suma = lambda x, y: x + y + z
```

La pureza de una función es un rasgo deseado y que hay que tratar de alcanzar siempre que sea posible, ya que facilita el desarrollo y mantenimiento de los programas, además de simplificar el razonamiento sobre los mismos, permitiendo aplicar directamente nuestro modelo de sustitución.

Es más incómodo trabajar con `suma` porque hay que *recordar* que depende de un valor que está *fuera* de la expresión lambda, cosa que no resulta evidente a no ser que mires en el cuerpo de la expresión lambda.

1.2. Estrategias de evaluación

A la hora de evaluar una expresión (cualquier expresión) existen varias **estrategias** diferentes que se pueden adoptar.

Cada lenguaje implementa sus propias estrategias de evaluación que están basadas en las que vamos a ver aquí.

Básicamente se trata de decidir, en cada paso de reducción, qué sub-expresión hay que reducir, en función de:

- El orden de evaluación:
  - De fuera adentro o de dentro afuera.
  - De izquierda a derecha o de derecha a izquierda.
- La necesidad o no de evaluar dicha sub-expresión.

1.2.1. Orden de evaluación

En un lenguaje de programación funcional puro se cumple la **transparencia referencial**, según la cual el valor de una expresión depende sólo del valor de sus sub-expresiones (también llamadas *redexes*).

Pero eso también implica que **no importa el orden en el que se evalúen las sub-expresiones**: el resultado debe ser siempre el mismo.

Gracias a ello podemos usar nuestro modelo de sustitución como modelo computacional.

Hay dos **estrategias básicas de evaluación**:

- **Orden aplicativo**: reducir siempre el *redex* más **interno** (y más a la izquierda).
- **Orden normal**: reducir siempre el *redex* más **externo** (y más a la izquierda).

Python usa el orden aplicativo, salvo excepciones.

1.2.1.1. Orden aplicativo

El **orden aplicativo** consiste en evaluar las expresiones *de dentro afuera*, es decir, empezando siempre por el *redex* más **interno** y más a la izquierda.

Corresponde a lo que en muchos lenguajes de programación se denomina **paso de argumentos por valor**.

Ejemplo:

```
cuadrado = lambda x: x * x
```

Según el orden aplicativo, la expresión `cuadrado(3 + 4)` se reduciría así:

```
cuadrado(3 + 4)
= cuadrado(7)              # aritmética
= (lambda x, y: x * x)(7)  # definición de cuadrado
= (7 * 7)                  # aplicación a 7
= 49                       # aritmética
```

alcanzando la forma normal en 4 pasos de reducción.

1.2.1.2. Orden normal

El **orden normal** consiste en evaluar las expresiones *de fuera adentro*, es decir, empezando siempre por el *redex* más **externo** y más a la izquierda.

Corresponde a lo que en muchos lenguajes de programación se denomina **paso de argumentos por nombre**.

Ejemplo:

```
cuadrado = lambda x: x * x
```

Según el orden normal, la expresión `cuadrado(3 + 4)` se reduciría así:

```
cuadrado(3 + 4)
= (lambda x, y: x * x)(3 + 4)  # definición de cuadrado
= ((3 + 4) * (3 + 4))          # aplicación a (3 + 4)
= 7 * (3 + 4)                  # aritmética
= 7 * 7                        # aritmética
= 49
```

alcanzando la forma normal en 5 pasos de reducción.

1.2.2. Evaluación estricta y no estricta

Existe otra forma de ver la evaluación de una expresión:

- **Evaluación estricta**: Reducir todos los *redexes* aunque no hagan falta.

- **Evaluación no estricta**: Reducir sólo los *redexes* que sean estrictamente necesarios para calcular el valor de la expresión.

  A esta estrategia de evaluación se la denomina también **evaluación perezosa**.

Por ejemplo:

Sabemos que la expresión `1/0` da un error de *división por cero*:

```
>>> 1/0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Supongamos que tenemos la siguiente definición:

```
primero = lambda x, y: x
```

de forma que `primero` es una función que simplemente devuelve el primero de sus argumentos.

Es evidente que la función `primero` no necesita evaluar nunca su segundo argumento, ya que no lo utiliza (simplemente devuelve el primero de ellos). Por ejemplo, `primero(4, 3)` devuelve `4`.

Sabiendo eso... ¿qué valor devolvería la siguiente expresión?

```
primero(4, 1/0)
```

Curiosamente, el resultado dependerá de si la evaluación es estricta o perezosa:

- **Si es estricta**, el intérprete evaluará todos los argumentos de la expresión lambda aunque no se utilicen luego en su cuerpo. Por tanto, al evaluar `1/0` devolverá un error.

  Es lo que ocurre cuando se evalúa siguiendo el **orden aplicativo**.

- En cambio, **si es perezosa**, el intérprete evaluará únicamente aquellos argumentos que se usen en el cuerpo de la expresión lambda, y en este caso sólo se usa el primero, así que dejará sin evaluar el segundo, no dará error y devolverá directamente `4`.

  Es lo que ocurre cuando se evalúa siguiendo el **orden normal**:

  ```
  primero(4, 1/0) = (lambda x, y: x)(4, 1/0) = (4) = 4
  ```

Hay un resultado teórico que avala lo que acabamos de observar:

Teorema:

Si una expresión tiene forma normal, el **orden normal** de evaluación conduce seguro a la misma.

En cambio, el orden aplicativo es posible que no encuentre la forma normal de la expresión.

En **Python** la evaluación es **estricta**, salvo algunas excepciones:

- El operador ternario:

⟨*expr_condicional*⟩ ::= ⟨*valor_si_cierto*⟩ **if** ⟨*condición*⟩ **else** ⟨*valor_si_falso*⟩

evalúa perezosamente ⟨*valor_si_cierto*⟩ y ⟨*valor_si_falso*⟩.

- Los operadores lógicos `and` y `or` también son perezosos (se dice que evalúan **en cortocircuito**):
  * `True or x` siempre es igual a `True`.
  * `False and x` siempre es igual a `False`.

  En ambos casos no es necesario evaluar `x`.

La mayoría de los lenguajes de programación usan evaluación estricta y paso de argumentos por valor (siguen el orden aplicativo).

Haskell, por ejemplo, es un lenguaje funcional puro que usa evaluación perezosa y sigue el orden normal.

1.3. Composición de funciones

Podemos crear una función que use otra función. Por ejemplo, para calcular el área de un círculo usamos otra función que calcule el cuadrado de un número:

```
cuadrado = lambda x: x * x
area = lambda r: 3.1416 * cuadrado(r)
```

La expresión `area(11 + 1)` se evaluaría así según el *orden aplicativo*:

```
1 area(11 + 1)                                  # aritmética
2 = area(12)                                    # definición de area
3 = (lambda r: 3.1416 * cuadrado(r))(12)        # aplicación
4 = (3.1416 * cuadrado(12))                     # definición de cuadrado
5 = (3.1416 * (lambda x: x * x)(12))            # aplicación
6 = (3.1416 * (12 * 12))                        # aritmética
7 = (3.1416 * 144)                              # aritmética
8 = 452.3904
```

En detalle:

- **Línea 1**: Se evalúa primero el argumento.
- **Línea 3**: El *redex* más interno es `r`, pero no puede reducirse más en este momento porque se desconoce su valor. El siguiente más interno es `cuadrado(r)` pero tampoco puede reducirse ya que se desconoce el valor de `r`. El siguiente más interno es el `*`, que tampoco se puede reducir porque su operando derecho (el `cuadrado(r)`) aún no está evaluado ni se puede evaluar, así que el *redex* que queda es la aplicación del `lambda` sobre su argumento `12`.
- **Línea 4**: El *redex* más interno es `cuadrado(12)`, que se evalúa reescribiéndolo con su definición.
- **Línea 5**: El *redex* más interno es la aplicación del `lambda` sobre su argumento `12`.

La expresión `area(11 + 1)` se evaluaría así según el *orden normal*:

```
1 area(11 + 1)                                  # definición de area
2 = (lambda r: 3.1416 * cuadrado(r))(11 + 1)    # aplicación
3 = (3.1416 * cuadrado(11 + 1))                 # definición de cuadrado
4 = (3.1416 * (lambda x: x * x)(11 + 1))        # aplicación
5 = (3.1416 * ((11 + 1) * (11 + 1)))            # aritmética
6 = (3.1416 * (12 * (11 + 1)))                  # aritmética
7 = (3.1416 * (12 * 12))                        # aritmética
8 = (3.1416 * 144)                              # aritmética
9 = 452.3904
```

En ambos casos (orden aplicativo y orden normal) se obtiene el mismo resultado.

En detalle:

- **Línea 1**: Se evalúa el *redex* más externo, que es `area` (se reescribe con su definición).
- **Línea 2**: El *redex* más externo es la aplicación de la expresión `lambda` a su argumento `11 + 1`.
- **Línea 3**: El *redex* más externo es el `*`, pero para evaluarlo hay que evaluar primero todos sus argumentos, por lo que ahora hay que evaluar `cuadrado`, reescribiéndolo con su definición.
- **Línea 4**: Igual que en la línea 3, para poder evaluar el `*` más externo hay que evaluar primero la aplicación del `lambda` sobre su argumento `11 + 1`.

1.4. Las funciones como abstracciones

Aunque es muy sencilla, la función `area` ejemplifica la propiedad más potente de las funciones definidas por el programador: la **abstracción**.

La función `area` está definida sobre la función `cuadrado`, pero se basa sólo en la relación que `cuadrado` establece entre sus argumentos de entrada y su resultado de salida.

Podemos escribir `area` sin preocuparnos de cómo calcular el cuadrado de un número, porque eso ya lo hace la función `cuadrado`.

Los detalles sobre cómo se calcula el cuadrado están **ocultos dentro de la definición** de `cuadrado`. Esos detalles **se ignoran en este momento** para considerarlos más tarde.

De hecho, por lo que respecta a `area`, `cuadrado` no representa una definición concreta de función, sino más bien la abstracción de una función, lo que se denomina una **abstracción funcional**. A este nivel de abstracción, cualquier función que calcule el cuadrado de un número es igual de buena y le serviría igual de bien a `area`.

Por tanto, considerando únicamente los valores que devuelven, las dos funciones siguientes son indistinguibles e igual de válidas para `area`. Ambas reciben un argumento numérico y devuelven el cuadrado de ese número:

```
cuadrado = lambda x: x * x
cuadrado = lambda x: x * (x - 1) + x
```

En otras palabras: la definición de una función debe ser capaz de **ocultar sus detalles de implementación**.

Un programador no debe necesitar saber cómo está implementada una función por dentro para poder usarla. Eso es lo que ocurre, por ejemplo, con las funciones predefinidas del lenguaje: sabemos

qué hacen pero no necesitamos saber *cómo* lo hacen.

Incluso puede que el usuario de una función no sea el mismo que la haya escrito, sino que la puede haber recibido de otro programador como una «**caja negra**».

1.4.1. Especificaciones de funciones

Técnicamente, se dice que para poder **usar una abstracción funcional** nos basta con conocer su **especificación**, que es la descripción de qué hace esa función.

La especificación de una abstracción funcional está formada por tres propiedades fundamentales:

- El **dominio**: el conjunto de argumentos válidos.
- El **rango**: el conjunto de posibles valores que devuelve.
- El **propósito**: qué hace la función, es decir, la relación entre su entrada y su salida.

Nosotros hasta ahora, al especificar programas, hemos llamado **entrada** al dominio y hemos agrupado el rango y el propósito en una sola propiedad que hemos llamado **salida**.

Por ejemplo, cualquier función `cuadrado` que usemos para implementar `area` debe satisfacer esta especificación:

$$\begin{cases} \textbf{Entrada} : n \in \mathbb{R} \\ \textit{cuadrado} \\ \textbf{Salida} : n^2 \end{cases}$$

La especificación **no concreta cómo** se debe llevar a cabo el propósito. Ese es un detalle de implementación que se abstrae a este nivel.

Este esquema es el que hemos usado hasta ahora para especificar programas, y se podría seguir usando para especificar funciones, ya que éstas son consideradas *subprogramas*.

Pero para especificar una función, en cambio, resulta más adecuado usar el siguiente esquema, al que llamaremos **especificación funcional**:

$$\begin{cases} \textbf{Pre} : \textit{True} \\ \textit{cuadrado}\ (n : \textit{float})\ \texttt{->}\ \textit{float} \\ \textbf{Post} : \textit{cuadrado}(n) = n^2 \end{cases}$$

Pre representa la **precondición**: la propiedad que debe cumplirse justo *antes* de llamar a la función.

Post representa la **postcondición**: la propiedad que debe cumplirse justo *después* de llamar a la función.

Lo que hay en medio es la **signatura**: el nombre de la función, el nombre y tipo de sus parámetros y el tipo del valor de retorno.

La especificación se lee así: si se llama a una función cumpliendo con su signatura en un estado que satisface su precondición, la llamada termina y lo hace en un estado que satisface su postcondición.

En este caso, la precondición es *True*, que equivale a decir que cualquier condición de entrada es buena para usar la función.

Dicho de otra forma, no hace falta que se dé ninguna condición especial para usar la función. Siempre que la llamada cumpla con la signatura de la función, el parámetro *n* puede tomar cualquier valor real y no hay ninguna restricción adicional.

Tanto la precondición como la postcondición son **predicados**, es decir, expresiones lógicas que se escriben usando el lenguaje de las matemáticas y la lógica.

La signatura se escribe usando la sintaxis del lenguaje de programación que vayamos a usar para implementar la función (en este caso, Python).

Las pre y postcondiciones no es necesario escribirlas de una manera **formal y rigurosa**, usando el lenguaje de las Matemáticas o la Lógica.

Si la especificación se escribe en *lenguaje natural* y se entiende bien, completamente y sin ambigüedades, no hay problema.

El motivo de usar un lenguaje formal es que, normalmente, resulta **mucho más conciso y preciso que el lenguaje natural**.

El lenguaje natural suele ser:

- **Más prolijo**: necesita más palabras para decir lo mismo que diríamos matemáticamente usando menos caracteres.
- **Más ambiguo**: lo que se dice en lenguaje natural se puede interpretar de distintas formas.
- **Menos completo**: quedan flecos y situaciones especiales que no se tienen en cuenta.

Otro ejemplo más completo:

$$\begin{cases} \textbf{Pre} : car \neq \text{" "} \land len(car) = 1 \\ cuenta\ (cadena : str,\ car : str) \text{ -> } int \\ \textbf{Post} : cuenta(cadena, car) \geq 0 \land cuenta(cadena, car) = cadena.count(car) \end{cases}$$

count es una **función *oculta* o *auxiliar*** (en este caso, un *método auxiliar*). Las funciones auxiliares se puede usar en la especificación pero está prohibido usarlas en la implementación.

Con esto estamos diciendo que *cuenta* es una función que recibe una cadena y un carácter (otra cadena con un único carácter dentro).

Además, estamos diciendo que devuelve el mismo resultado que devuelve el método `count` (que ya existe en Python).

Es decir: cuenta el número de veces que el carácter *car* aparece en *cadena*.

En realidad, las condiciones de la especificación anterior se podrían simplificar aprovechando las propiedades de las expresiones lógicas, quedando así:

$$\begin{cases} \textbf{Pre} : len(car) = 1 \\ cuenta\ (cadena : str,\ car : str) \text{ -> } int \\ \textbf{Post} : cuenta(cadena, car) = cadena.count(car) \end{cases}$$

Ejercicio

1. ¿Por qué?

Finalmente, podríamos escribir la misma especificación en lenguaje natural:

$$\begin{cases} \textbf{Pre} : car \text{ debe ser un único carácter} \\ cuenta\ (cadena : str,\ car : str) \text{ -> } int \\ \textbf{Post} : cuenta(cadena, car) \text{ devuelve el número de veces} \\ \qquad \text{que aparece el carácter } car \text{ en la cadena } cadena. \\ \qquad \text{Si } cadena \text{ es vacía o } car \text{ no aparece nunca en la} \\ \qquad \text{cadena } cadena, \text{ debe devolver 0.} \end{cases}$$

Probablemente resulta más fácil de leer (sobre todo para los novatos), pero también es más largo y prolijo.

Es como un contrato escrito por un abogado en lenguaje jurídico.

2. Computabilidad

2.1. Funciones y procesos

Los **procesos** son entidades abstractas que habitan los ordenadores.

Conforme van evolucionando, los procesos manipulan otras entidades abstractas llamadas **datos**.

La evolución de un proceso está dirigida por un patrón de reglas llamada **programa**.

Los programadores crean programas para **dirigir** a los procesos.

Es como decir que los programadores son magos que invocan a los espíritus del ordenador (los procesos) con sus conjuros (los programas) escritos en un lenguaje mágico (el lenguaje de programación).

Una **función** describe la *evolución local* de un **proceso**.

En cada paso se calcula el *siguiente estado* del proceso basándonos en el estado actual y en las reglas definidas por la función.

Nos gustaría ser capaces de visualizar y de realizar afirmaciones sobre el comportamiento global del proceso cuya evolución local está definida por la función.

Esto, en general, es muy difícil, pero al menos vamos a describir algunos de los modelos típicos de evolución de los procesos.

2.2. Funciones recursivas

2.2.1. Definición

Una **función recursiva** es aquella que se define en términos de sí misma.

En general, eso quiere decir que la definición de la función contiene una o varias referencias a ella misma y que, por tanto, se llama a sí misma dentro de su cuerpo.

Las definiciones recursivas son el mecanismo básico para ejecutar **repeticiones de instrucciones** en un lenguaje de programación funcional.

Por ejemplo:

$$f(n) = n + f(n+1)$$

Por tanto,

$$f(1) = 1 + f(2) = 1 + 2 + f(3) = 1 + 2 + 3 + f(4) = \dots$$

Cada vez que una función se llama a sí misma decimos que se realiza una **llamada recursiva** o **paso recursivo**.

2.2.2. Casos base y casos recursivos

Resulta importante que una definición recursiva se detenga alguna vez y proporcione un resultado, ya que si no, no sería útil (tendríamos lo que se llama una **recursión infinita**).

Por tanto, en algún momento, la recursión debe alcanzar un punto en el que la función no se llame a sí misma y se detenga.

Para ello, es necesario que la función, en cada paso recursivo, se vaya acercando cada vez más a ese punto.

A ese punto en el que la función recursiva no se llama a sí misma, se le denomina **caso base**, y puede haber más de uno.

Los casos base, por tanto, determinan bajo qué condiciones la función no se llamará a sí misma, o dicho de otra forma, con qué valores de sus argumentos la función devolverá directamente un valor y no provocará una nueva llamada recursiva.

Los demás casos, que sí provocan llamadas recursivas, se denominan **casos recursivos**.

2.2.3. El factorial

El ejemplo más típico de función recursiva es el **factorial**.

El factorial de un número natural n se representa por $n!$ y se define como el producto de todos los números desde 1 hasta n:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

Por ejemplo:

$$6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$$

Pero para calcular 6! también se puede calcular 5! y después multiplicar el resultado por 6, ya que:

$$6! = 6 \cdot \overbrace{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}^{5!}$$

$$6! = 6 \cdot 5!$$

Por tanto, el factorial se puede definir de forma **recursiva**.

Tenemos el **caso recursivo**, pero necesitamos al menos un **caso base** para evitar que la recursión se haga *infinita*.

El caso base del factorial se obtiene sabiendo que el factorial de 0 es directamente 1 (no hay que llamar al factorial recursivamente):

$$0! = 1$$

Combinando ambos casos tendríamos:

$$n! = \begin{cases} 1 & \text{si } n = 0 \quad \text{(caso base)} \\ n \cdot (n-1)! & \text{si } n > 0 \quad \text{(caso recursivo)} \end{cases}$$

2.2.4. Diseño de funciones recursivas

El diseño de funciones recursivas se basa en:

- Pensamiento optimista
- Descomposición (reducción) del problema
- Identificación de problemas no reducibles (mínimos)

2.2.4.1. Pensamiento optimista

Consiste en suponer que la función deseada ya existe y es capaz de resolver ejemplares más pequeños del problema (este paso se denomina **hipótesis inductiva**).

Se trata de encontrar el patrón común de forma que resolver el problema principal implique el mismo patrón en un problema más pequeño.

Ejemplo:

- Queremos diseñar una función que calcule el factorial de un número.
- Para ello, supongamos que ya contamos con una función que calcula el factorial de un número más pequeño. Tenemos que creer y confiar en que es así, aunque ahora mismo no sea verdad.

  Es decir: si queremos calcular el factorial de n, suponemos que tenemos ya una función *fact* que no sabe calcular el factorial de n, pero sí el de $(n-1)$. *Ésta es nuestra hipótesis inductiva*.

2.2.4.2. Descomposición del problema

Reducimos el problema de forma que así tendremos un ejemplar más pequeño del mismo problema y, por tanto, podremos usar la función *fact* anterior para poder resolver ese ejemplar más pequeño.

A continuación, usamos dicha solución *parcial* para obtener la solución al problema original.

Ejemplo:

- Sabemos que $n! = n \cdot (n-1)!$
- Sabemos que la función *fact* sabe calcular el factorial de $(n-1)$ (por pensamiento optimista).
- Por tanto, lo único que tenemos que hacer para obtener el factorial de n es multiplicar n por el resultado de $fact(n-1)$.

> Dicho de otra forma: ***si yo supiera* calcular el factorial de $(n-1)$, me bastaría con multiplicarlo por n para obtener el factorial de n.**

2.2.4.3. Identificación de problemas no reducibles

Debemos identificar los ejemplares más pequeños (los que no se pueden reducir más) para los cuales hay una solución explícita y directa que no necesita recursividad: los *casos base*.

Es importante comprobar que la reducción que le hemos realizado al problema en el paso anterior produce ejemplares que están más cerca del caso base.

Ejemplo:

- En nuestro caso, sabemos que $0! = 1$, por lo que nuestra función podría devolver directamente 1 cuando se le pida calcular el factorial de 0.
- Además, en la reducción obtenida en el paso anterior, pasamos de calcular el factorial de n a calcular el factorial de uno menos, con lo cual, cada vez estaremos más cerca del caso base, que es el factorial de 0. Al final siempre acabaremos alcanzando el caso base.

Combinando todos los pasos, obtenemos la solución general:

$$fact(n) = \begin{cases} 1 & \text{si } n = 0 \quad \text{(caso base)} \\ n \cdot fact(n-1) & \text{si } n > 0 \quad \text{(caso recursivo)} \end{cases}$$

2.2.5. Recursividad lineal

Una función tiene **recursividad lineal** si cada llamada a la función recursiva genera, como mucho, otra llamada recursiva a la misma función.

El factorial definido en el ejemplo anterior es un caso típico de recursividad lineal.

2.2.5.1. Procesos lineales recursivos

La forma más directa y sencilla de definir una función que calcule el factorial de un número a partir de su definición recursiva podría ser la siguiente:

```
factorial = lambda n: 1 if n == 0 else n * factorial(n - 1)
```

Utilizaremos el modelo de sustitución para observar el funcionamiento de esta función al calcular 6!:

```
factorial(6)
= (6 * factorial(5))
= (6 * (5 * factorial(4)))
= (6 * (5 * (4 * factorial(3))))
= (6 * (5 * (4 * (3 * factorial(2)))))
= (6 * (5 * (4 * (3 * (2 * factorial(1))))))
= (6 * (5 * (4 * (3 * (2 * (1 * factorial(0)))))))
= (6 * (5 * (4 * (3 * (2 * (1 * 1))))))
= (6 * (5 * (4 * (3 * (2 * 1)))))
= (6 * (5 * (4 * (3 * 2))))
= (6 * (5 * (4 * 6)))
= (6 * (5 * 24))
= (6 * 120)
= 720
```

Podemos observar un perfil de **expansión** seguido de una **contracción**:

- La **expansión** ocurre conforme el proceso construye una cadena de operaciones a realizar *posteriormente* (en este caso, una cadena de multiplicaciones).
- La **contracción** se realiza conforme se van ejecutando realmente las multiplicaciones.

Llamaremos **proceso recursivo** a este tipo de proceso caracterizado por una cadena de **operaciones pendientes de completar**.

Para poder ejecutar este proceso, el intérprete necesita **memorizar**, en algún lugar, un registro de las multiplicaciones que se han dejado para más adelante.

En el cálculo de $n!$, la longitud de la cadena de operaciones pendientes (y, por tanto, la información que necesita almacenar el intérprete), crece *linealmente* con n, al igual que el número de pasos de reducción.

- A este tipo de procesos lo llamaremos **proceso recursivo *lineal***.

2.2.5.2. Procesos lineales iterativos

A continuación adoptaremos un enfoque diferente.

Podemos mantener un producto acumulado y un contador desde n hasta 1, de forma que el contador y el producto cambien de un paso al siguiente según la siguiente regla:

$$\begin{cases} acumulado_{nuevo} = acumulado_{viejo} \cdot contador_{viejo} \\ contador_{nuevo} = contador_{viejo} - 1 \end{cases}$$

Su traducción a Python podría ser la siguiente, usando una función auxiliar `fact_iter`:

```
fact_iter = lambda cont, acc: acc if cont == 0 else \
                              fact_iter(cont - 1, cont * acc)
fact = lambda n: fact_iter(n, 1)
```

Al igual que antes, usaremos el modelo de sustitución para visualizar el proceso del cálculo de 6!:

```
fact(6)
= fact_iter(6, 1)
= fact_iter(5, 6)
= fact_iter(4, 30)
= fact_iter(3, 120)
= fact_iter(2, 360)
= fact_iter(1, 720)
= fact_iter(0, 720)
= 720
```

Este proceso no tiene expansiones ni contracciones ya que, en cada instante, toda la información que se necesita almacenar es el valor actual de los parámetros `cont` y `acc`, por lo que el tamaño de la memoria necesaria es constante.

A este tipo de procesos lo llamaremos **proceso iterativo**.

El número de pasos necesarios para calcular *n*! usando esta función crece *linealmente* con *n*.

- A este tipo de procesos lo llamaremos **proceso iterativo *lineal***.

Tipo de proceso	Número de reducciones	Memoria necesaria
Recursivo	Proporcional a `n`	Proporcional a `n`
Iterativo	Proporcional a `n`	Constante

* * *

Tipo de proceso	Número de reducciones	Memoria necesaria
Recursivo lineal	Linealmente proporcional a `n`	Linealmente proporcional a `n`
Iterativo lineal	Linealmente proporcional a `n`	Constante

En general, un **proceso iterativo** es aquel que está definido por una serie de **variables de estado** junto con una **regla** fija que describe cómo actualizar dichas variables conforme cambia el proceso de un estado al siguiente.

La **diferencia entre los procesos recursivo e iterativo** se puede describir de esta otra manera:

- En el **proceso iterativo**, las variables ligadas dan una descripción completa del estado del proceso en cada instante.

Así, si parásemos el cálculo entre dos pasos, lo único que necesitaríamos hacer para seguir con el cálculo es darle al intérprete el valor de los dos parámetros.

- En el **proceso recursivo**, el intérprete tiene que mantener cierta información *oculta* que no está almacenada en ningún parámetro y que indica en qué punto se encuentra el proceso dentro de la cadena de operaciones pendientes.

No debe confundirse un **proceso recursivo** con una **función recursiva**:

- Cuando hablamos de *función recursiva* nos referimos al hecho sintáctico de que la definción de la función hace referencia a sí misma (directa o indirectamente).
- Cuando hablamos de *proceso recursivo* nos referimos a la forma en como se desenvuelve la ejecución de la función.

Puede parecer extraño que digamos que una función recursiva (por ejemplo, `fact_iter`) genera un proceso iterativo.

Sin embargo, el proceso es realmente iterativo porque su estado está definido completamente por dos variables ligadas, y para ejecutar el proceso sólo se necesita almacenar esas dos variables.

2.2.6. Recursividad en árbol

La **recursividad en árbol** se produce cuando la función tiene **recursividad múltiple**.

Una función tiene **recursividad múltiple** cuando una llamada a la función recursiva puede generar más de una llamada recursiva a la misma función.

El ejemplo clásico es la función que calcula los términos de la **sucesión de Fibonacci**.

La sucesión comienza con los números 0 y 1, y a partir de éstos, cada término es la suma de los dos anteriores:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, ...

Podemos definir una función que devuelva el n-ésimo término de la sucesión de Fibonacci:

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \quad \text{(caso base)} \\ 1 & \text{si } n = 1 \quad \text{(caso base)} \\ fib(n-1) + fib(n-2) & \text{si } n > 1 \quad \text{(caso recursivo)} \end{cases}$$

Que traducida a Python sería:

```
fib = lambda n: 0 if n == 0 else 1 if n == 1 else fib(n - 1) + fib(n - 2)
```

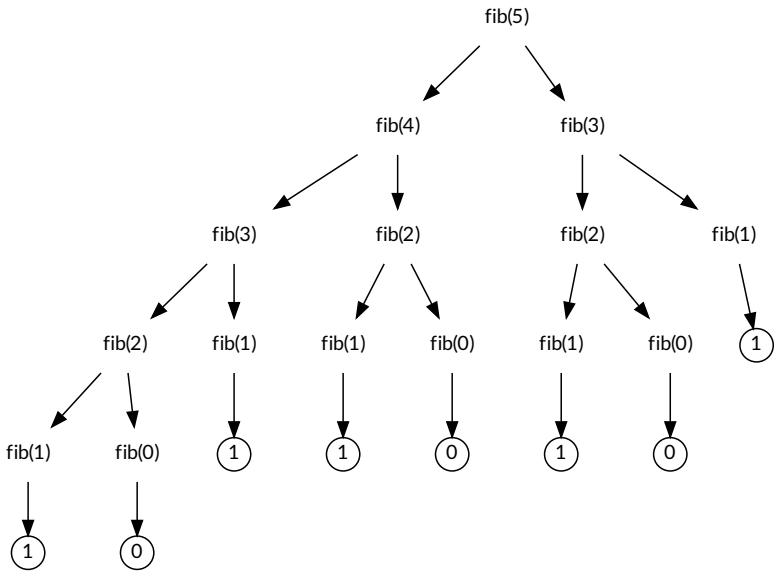
o bien:

```
fib = lambda n: 0 if n == 0 else \
                1 if n == 1 else \
                fib(n - 1) + fib(n - 2)
```

Si vemos el perfil de ejecución de `fib(5)`, vemos que:

- Para calcular `fib(5)`, antes debemos calcular `fib(4)` y `fib(3)`.
- Para calcular `fib(4)`, antes debemos calcular `fib(3)` y `fib(2)`.
- Así sucesivamente hasta poner todo en función de `fib(0)` y `fib(1)`, que se pueden calcular directamente (son los casos base).

En general, el proceso resultante parece un árbol.



La función anterior es un buen ejemplo de recursión en árbol, pero desde luego es un método *horrible* para calcular los números de Fibonacci, por la cantidad de **operaciones redundantes** que efectúa.

Para tener una idea de lo malo que es, se puede observar que $fib(n)$ crece exponencialmente en función de n.

Por lo tanto, el proceso necesita una cantidad de tiempo que crece **exponencialmente** con n.

Por otro lado, el espacio necesario sólo crece **linealmente** con n, porque en un cierto momento del cálculo sólo hay que memorizar los nodos que hay por encima.

En general, en un proceso recursivo en árbol el tiempo de ejecución crece con el número de nodos mientras que el espacio necesario crece con la altura máxima del árbol.

Se puede construir un **proceso iterativo** para calcular los números de Fibonacci.

La idea consiste en usar dos variables de estado `a` y `b` (con valores iniciales `1` y `0`, respectivamente) y aplicar repetidamente la siguiente transformación:

$$\begin{cases} a_{nuevo} = a_{viejo} + b_{viejo} \\ b_{nuevo} = a_{viejo} \end{cases}$$

Después de n pasos, `a` y `b` contendrán, respectivamente, $fib(n+1)$ y $fib(n)$.

En Python sería:

```
fib_iter = lambda cont, a, b: b if cont == 0 else fib_iter(cont - 1, a + b, a)
fib = lambda n: fib_iter(n , 1, 0)
```

Esta función genera un proceso iterativo lineal, por lo que es mucho más eficiente.

2.3. La pila de control

La **pila de control** es una estructura de datos que utiliza el intérprete para llevar la cuenta de las **llamadas *activas*** en un determinado momento, incluyendo el valor de sus parámetros y el punto de retorno al que debe devolverse el control cuando finalice la ejecución de la función.

- Las **llamadas activas** son aquellas llamadas a funciones que aún no han terminado de ejecutarse.

La pila de control es, básicamente, un **almacén de marcos**.

Cada vez que se hace una nueva llamada a una función, **su marco** correspondiente **se almacena en la cima de la pila** sobre los demás marcos que pudiera haber.

Ese marco es el primero de la cadena de marcos que forman el entorno de la función, que deberán estar almacenados más abajo en la pila.

Los marcos se enlazan entre sí para representar los entornos que actúan en las distintas llamadas activas.

El intérprete puede, además, almacenar ahí cualquier otra información que necesite para gestionar las llamadas a funciones.

El marco de la función, junto con toda esa información adicional, se denomina **registro de activación**.

Por tanto, **la pila de control almacena registros de activación**.

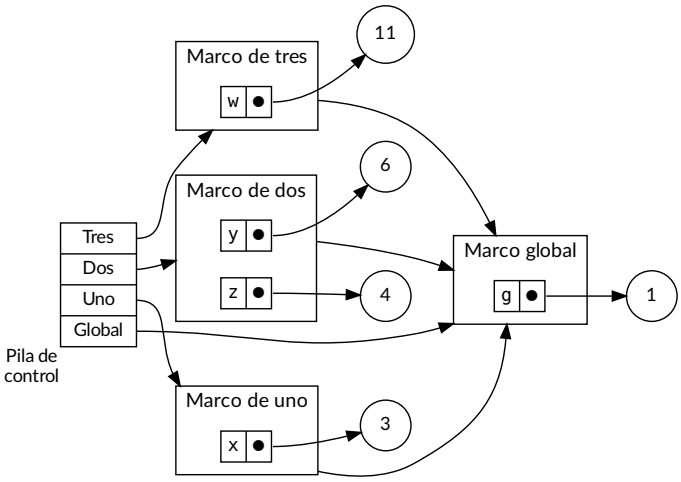
Cada llamada activa está representada por su correspondiente registro de activación en la pila, y cada registro de activación va asociado a un marco.

En cuanto la llamada finaliza, su registro de activación se saca de la pila y se transfiere el control a la llamada que está inmediatamente debajo (si es que hay alguna).

Cuando desaparece un registro de activación, también se elimina de la memoria su marco asociado (hay una excepción a esto, que veremos en posteriores temas cuando hablemos de las *clausuras*).

Supongamos el siguiente código:

```
g = 1
uno = lambda x: 1 + dos(2 * x, 4)
dos = lambda y, z: tres(y + z + g)
tres = lambda w: "W vale " + str(w)
uno(3)
```



Pila de control con la función `tres` activada

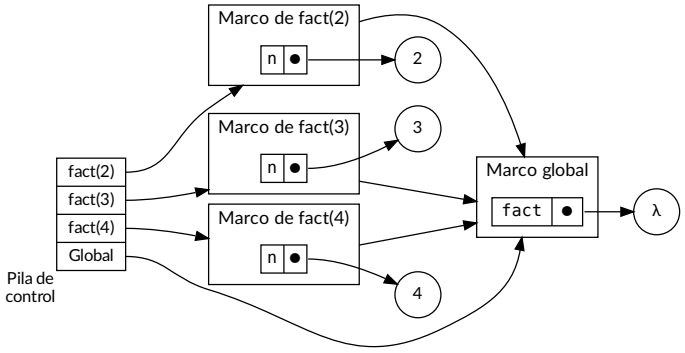
Del análisis del diagrama del ejemplo anterior se pueden deducir las siguientes conclusiones:

- En un momento dado, dentro del ámbito global se ha llamado a la función `uno`, la cual ha llamado a la función `dos`, la cual ha llamado a la función `tres`, la cual aún no ha terminado de ejecutarse.
- El entorno en la función `uno` empieza por el marco de `uno`, el cual apunta al marco global.
- El entorno en la función `dos` empieza por el marco de `uno`, el cual apunta al marco global.
- El entorno en la función `tres` empieza por el marco de `uno`, el cual apunta al marco global.

Hemos dicho que habrá un registro de activación por cada nueva llamada que se realice a una función, y que ese registro se mantendrá en la pila hasta que la llamada finalice.

Por tanto, en el caso de una función recursiva, tendremos un registro de activación por cada llamada recursiva.

```
fact = lambda n: 1 if n == 0 else n * fact(n - 1)
fact(4)
```



Pila de control de `fact` tras tres activaciones desde `fact(4)`

2.4. Un lenguaje Turing-completo

El paradigma funcional que hemos visto hasta ahora (uno que nos permite definir funciones, componer dichas funciones y aplicar recursividad, junto con el operador ternario condicional) es un lenguaje de programación **completo**.

Decimos que es **Turing completo**, lo que significa que puede computar cualquier función que pueda computar una máquina de Turing.

Como las máquinas de Turing son los ordenadores más potentes que podemos construir (ya que describen lo que cualquier ordenador es capaz de hacer), esto significa que nuestro lenguaje puede calcular todo lo que pueda calcular cualquier ordenador.

3. Tipos de datos recursivos

3.1. Cadenas

Las **cadenas** se pueden considerar **datos recursivos compuestos**, ya que podemos decir que toda cadena `c`:

- o bien es la cadena vacía `''` (*caso base*),
- o bien está formada por dos partes:
  - El **primer carácter** de la cadena, al que se accede mediante `c[0]`.
  - El **resto** de la cadena (al que se accede mediante `c[1:]`), que también es una cadena (*caso recursivo*).

Eso significa que podemos acceder al segundo carácter de la cadena (suponiendo que exista) mediante `c[1:][0]`.

```
cadena = 'hola'
cadena[0]       # devuelve 'h'
cadena[1:]      # devuelve 'ola'
cadena[1:][0]   # devuelve 'o'
```

3.2. Tuplas

Las **tuplas** son una generalización de las cadenas.

Una tupla es una **secuencia de elementos** que no tienen por qué ser caracteres, sino que cada uno de ellos pueden ser **de cualquier tipo** (números, cadenas, booleanos, ..., incluso otras tuplas).

Los literales de tipo tupla se representan enumerando sus elementos separados por comas y encerrados entre paréntesis.

Por ejemplo:

```
tupla = (27, 'hola', True, 73.4, ('a', 'b', 'c'), 99)
```

Si sólo tiene un elemento, hay que poner una coma detrás:

```
tupla = (35,)
```

Las tuplas también pueden verse como un **tipo de datos recursivo**, ya que toda tupla `t`:

- o bien es la tupla vacía, representada mediante `()` (*caso base*),
- o bien está formada por dos partes:
  - El **primer elemento** de la tupla (al que se accede mediante `t[0]`), que hemos visto que puede ser de cualquier tipo.
  - El **resto** de la tupla (al que se accede mediante `t[1:]`), que también es una tupla (*caso recursivo*).

Según el ejemplo anterior:

```
tupla = (27, 'hola', True, 73.4, ('a', 'b', 'c'), 99)
tupla[0]       # devuelve 27
tupla[1:]      # devuelve ('hola', True, 73.4, ('a', 'b', 'c'), 99)
tupla[1:][0]   # devuelve 'hola'
```

Junto a las operaciones `t[0]` (primer elemento de la tupla) y `t[1:]` (resto de la tupla), tenemos también la operación `+` (**concatenación**), al igual que ocurre con las cadenas.

Con la concatenación se pueden crear nuevas tuplas a partir de otras tuplas.

Por ejemplo:

```
(1, 2, 3) + (4, 5, 6)  # devuelve (1, 2, 3, 4, 5, 6)
```

3.3. Rangos

Un rango es un tipo de dato cuyos valores representan **sencuencias de números enteros**.

Los rangos se crean con la función `range`, cuya sintaxis es:

⟨*rango*⟩ ::= **`range(`**[⟨*inicio*⟩`,`] ⟨*fin*⟩[`,` ⟨*paso*⟩]**`)`**

⟨*inicio*⟩, ⟨*fin*⟩ y ⟨*paso*⟩ deben ser números enteros.

Cuando se omite ⟨*inicio*⟩, se entiende que es `0`.

El valor de ⟨*fin*⟩ no se alcanza nunca.

Cuando ⟨*inicio*⟩ y ⟨*fin*⟩ son iguales, representa el *rango vacío*.

Cuando ⟨*inicio*⟩ es mayor que ⟨*fin*⟩, el ⟨*paso*⟩ debería ser negativo. En caso contrario, también representaría el rango vacío.

Ejemplos:

- `range(10)` representa la secuencia $0, 1, 2, \ldots, 9$
- `range(3, 10)` representa la secuencia $3, 4, 5, \ldots, 9$
- `range(0, 10, 2)` representa la secuencia $0, 2, 4, 6, 8$
- `range(4, 0, -1)` representa la secuencia $4, 3, 2, 1$
- `range(3, 3)` representa el rango vacío
- `range(4, 3)` también representa el rango vacío

Los rangos también pueden verse como un **tipo de datos recursivo**, ya que todo rango `r`:

- o bien es el rango vacío (*caso base*),
- o bien está formado por dos partes:
  - El **primer elemento** del rango (al que se accede mediante `r[0]`), que hemos visto que tiene que ser un número entero.
  - El **resto** del rango (al que se accede mediante `r[1:]`), que también es un rango (*caso recursivo*).

Según el ejemplo anterior:

```
rango = range(4, 7)
rango[0]       # devuelve 4
rango[1:]      # devuelve range(5, 7)
rango[1:][0]   # devuelve 5
```

3.4. Conversión a tupla

Las cadenas y los rangos se pueden convertir fácilmente a tuplas usando la función `tuple`:

```
>>> tuple('hola')
('h', 'o', 'l', 'a')
>>> tuple('')
()
```

```
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> tuple(range(1, 11))
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> tuple(range(0, 30, 5))
(0, 5, 10, 15, 20, 25)
>>> tuple(range(0, 10, 3))
(0, 3, 6, 9)
>>> tuple(range(0, -10, -1))
(0, -1, -2, -3, -4, -5, -6, -7, -8, -9)
>>> tuple(range(0))
()
>>> tuple(range(1, 0))
()
```

4. Funciones de orden superior

4.1. Concepto

Hemos visto que **las funciones son**, en realidad, **abstracciones** en la medida en que nos permiten usarlas sin tener que conocer los detalles internos del procesamiento que realizan.

Por ejemplo, si queremos usar la función `cubo`, nos da igual que dicha función esté implementada de cualquiera de las siguientes maneras:

```
cubo = lambda x: x * x * x
cubo = lambda x: x ** 3
cubo = lambda x: x * x ** 2
```

A efectos de **usar** la función, nos basta con saber que calcula el cubo de un número, sin necesitar saber qué cálculo concreto realizar para obtener el resultado. Los detalles de implementación quedan ocultos y por eso también decimos que `cubo` es una abstracción.

Las funciones también son abstracciones en la medida en que describen operaciones compuestas a realizar sobre ciertos valores sin importar cuáles sean esos valores en concreto.

Por ejemplo, cuando definimos:

```
cubo = lambda x: x * x * x
```

no estamos hablando del cubo de un número en particular, sino más bien de un **método** para calcular el cubo de un número.

Por supuesto, nos la podemos arreglar sin definir el cubo, escribiendo siempre expresiones explícitas (como `3*3*3`, `y*y*y`, etc.) sin usar la palabra «cubo», pero eso nos obligaría siempre a expresarnos en términos de las operaciones primitivas de nuestro lenguaje (como `*`), en vez de poder usar términos de más alto nivel.

Es decir: **nuestros programas podrían calcular el cubo de un número, pero no tendrían la habilidad de expresar el concepto de *elevar al cubo***.

Una de las habilidades que deberíamos pedir a un lenguaje potente es la posibilidad de **construir abstracciones** asignando un nombre a los patrones más comunes, y luego trabajar directamente en términos de dichas abstracciones.

Las funciones nos permiten esta habilidad y esa es la razón de que todos los lenguajes (salvo los más primitivos) incluyan mecanismos para definir funciones.

Por ejemplo: en el caso anterior, vemos que hay un patrón (multiplicar algo por sí mismo tres veces) que se repite con frecuencia, y a partir de él construimos una abstracción que asigna un nombre a ese patrón (*elevar al cubo*). Esa abstracción la definimos como una función que describe la *regla* necesaria para elevar algo al cubo.

Muchas veces observamos el mismo patrón en funciones muy diferentes.

Para poder abstraer, de nuevo, lo que tienen en común dichas funciones, deberíamos ser capaces de manejar funciones que acepten a otras funciones como argumentos o que devuelvan otra función como resultado. A estas funciones que manejan otras funciones las llamaremos **funciones de orden superior**.

Por ejemplo, supongamos las dos funciones siguientes:

```
# Suma los enteros comprendidos entre a y b:
suma_enteros = lambda a, b: 0 if a > b else a + suma_enteros(a + 1, b)

# Suma los cubos de los enteros comprendidos entre a y b:
suma_cubos = lambda a, b: 0 if a > b else cubo(a) + suma_enteros(a + 1, b)
```

Estas dos funciones comparten claramente un patrón subyacente común. Se diferencian solamente en:

- El nombre de la función
- La función de a que se utiliza para calcular cada término

Podríamos haber escrito las funciones anteriores rellenando los «casilleros» del siguiente *patrón general*:

```
⟨nombre⟩ = lambda a, b: 0 if a > b else ⟨término⟩(a) + ⟨nombre⟩(a + 1, b)
```

La existencia de este patrón común nos demuestra que hay una abstracción esperando que la saquemos a la superficie.

De hecho, los matemáticos han identificado hace mucho tiempo esta abstracción llamándola **suma de una serie**, y la expresan así:

$$\sum_{n=a}^{b} f(n) = f(a) + \ldots + f(b)$$

La ventaja que tiene usar la notación anterior es que se puede trabajar directamente con el concepto de sumatorio en vez de trabajar con sumas concretas, y podemos sacar conclusiones generales sobre los sumatorios independientemente de la serie particular que estemos tratando.

Igualmente, como programadores estamos interesados en que nuestro lenguaje tenga la suficiente potencia como para describir directamente el concepto de *sumatorio*, en vez de funciones particulares que calculen sumas concretas.

En programación funcional lo conseguimos creando funciones que conviertan los «casilleros» en parámetros:

```
suma = lambda term, a, b: 0 if a > b else term(a) + suma(term, a + 1, b)
```

De esta forma, las dos funciones `suma_enteros` y `suma_cubos` anteriores se podrían definir en términos de esta `suma`:

```
suma_enteros = lambda a, b: suma(lambda x: x, a, b)
suma_cubos = lambda a, b: suma(lambda x: x * x * x, a, b)
# O mejor aún:
suma_cubos = lambda a, b: suma(cubo, a, b)
```

¿Se podría generalizar aún más la función `suma`?

4.2. `map`

Supongamos que queremos escribir una función que, dada una tupla de números, nos devuelva otra tupla con los mismos números elevados al cubo.

Inténtalo primero como ejercicio.

Una forma de hacerlo sería:

```
elevar_cubo = lambda t: () if t == () else \
                        (cubo(t[0]),) + elevar_cubo(t[1:])
```

¿Y elevar a la cuarta potencia?

```
elevar_cuarta = lambda t: () if t == () else \
                          ((lambda x: x ** 4)(t[0]),) + elevar_cuarta(t[1:])
```

Es evidente que hay un patrón subyacente que se podría abstraer creando una función de orden superior que aplique una función `f` a los elementos de una tupla y devuelva la tupla resultante.

Esa función se llama `map`, y viene definida en Python:

```
map(⟨función⟩, ⟨iterable⟩) -> ⟨iterador⟩
```

donde ⟨*iterable*⟩ puede ser cualquier cosa compuesta de elementos que se puedan recorrer de uno en uno, como una **tupla**, una **cadena** o un **rango** (cualquier *secuencia* de elementos vale).

Podemos usarla así:

```
>>> map(cubo, [1, 2, 3, 4])
<map object at 0x7f22b25e9d68>
```

Lo que devuelve no es una lista, sino un objeto *iterador* que examinaremos con más detalle en posteriores temas.

Por ahora, nos basta con saber que un iterador es un flujo de datos que se pueden recorrer de uno en uno.

Lo que haremos aquí será simplemente transformar ese iterador en la lista correspondiente usando la función `list` sobre el resultado de `map`:

```
>>> list(map(cubo, [1, 2, 3, 4]))
[1, 8, 27, 64]
```

Además de una lista, también podemos usar un rango:

```
>>> list(map(cubo, range(1, 5)))
[1, 8, 27, 64]
```

¿Cómo definirías la función `map`?

Podríamos definirla así:

```
map = lambda f, l: [] if l == [] else [f(l[0])] + map(f, l[1:])
```

4.3. `filter`

`filter` es una **función de orden superior** que devuelve aquellos elementos de una lista (o cualquier cosa *iterable*) que cumplen una determinada condición.

Su sintaxis es:

filter(⟨*función*⟩, ⟨*iterable*⟩) -> ⟨*iterador*⟩

Por ejemplo:

```
>>> list(filter(lambda x: x > 0, [-4, 3, 5, -2, 8, -3, 9]))
[3, 5, 8, 9]
```

4.4. `reduce`

`reduce` es una **función de orden superior** que aplica, de forma acumulativa, una función a todos los elementos de una lista (o cualquier cosa *iterable*).

Las operaciones se hacen agrupándose **por la izquierda**.

Captura un **patrón muy frecuente** de recursión sobre listas de elementos.

Por ejemplo, para calcular la suma de todos los elementos de una lista, haríamos:

```
suma = lambda l: 0 if l == [] else l[0] + suma(l[1:])
```

Y para calcular el producto:

```
producto = lambda l: 1 if l == [] else l[0] * producto(l[1:])
```

Como podemos observar, la estrategia de cálculo es esencialmente la misma (sólo se diferencian en la operación a realizar (`+` o `*`) y en el valor inicial o *elemento neutro* (`0` o `1`).

Si abstraemos ese patrón común podemos crear una función de orden superior que capture la idea de **reducir todos los elementos de una lista a un único valor**.

Eso es lo que hace la función `reduce`.

Su sintaxis es:

```
reduce(⟨función⟩, ⟨iterable⟩[, ⟨valor_inicial⟩]) -> ⟨valor⟩
```

El ⟨*valor_inicial*⟩, si existe, se usará como primer elemento de la lista en el cálculo y sirve como valor por defecto cuando la lista está vacía.

La ⟨*función*⟩ debe recibir dos argumentos y devolver un valor.

Para usarla, tenemos que *importarla* previamente del *módulo* `functools`.

- No es la primera vez que importamos un módulo. Ya lo hicimos con el módulo `math`.
- En su momento estudiaremos con detalle qué son los módulos. Por ahora nos basta con lo que ya sabemos: que contienen definiciones que podemos incorporar a nuestros *scripts*.

Por ejemplo, para calcular la suma y el producto de `[1, 2, 3, 4]`:

```
from functools import reduce
lista = [1, 2, 3, 4]
suma_de_numeros = reduce(lambda x, y: x + y, lista, 0)
producto_de_numeros = reduce(lambda x, y: x * y, lista, 1)
```

¿Cómo podríamos definir la función `reduce`?

Una forma (con valor inicial obligatorio) podría ser así:

```
reduce = lambda fun, lista, ini: ini if lista == [] else \
                                 lista[0] if lista[1:] == [] else \
                                 fun(lista[0], reduce(fun, lista[1:], ini))
```

4.5. Listas por comprensión

Dos operaciones que se realizan con frecuencia sobre un iterador son:

- Realizar alguna operación sobre cada elemento (`map`)
- Seleccionar un subconjunto de elementos que cumplan alguna condición (`filter`)

Las listas por comprensión son una notación copiada del lenguaje Haskell que nos permite realizar ambas operaciones de una forma muy concisa:

```
>>> [x ** 3 for x in [1, 2, 3, 4]]
[1, 8, 27, 64]
# equivale a:
>>> list(map(lambda x: x ** 3, [1, 2, 3, 4]))
[1, 8, 27, 64]

>>> [x for x in [-4, 3, 5, -2, 8, -3, 9] if x > 0]
[3, 5, 8, 9]
# equivale a:
>>> list(filter(lambda x: x > 0, [-4, 3, 5, -2, 8, -3, 9]))
[3, 5, 8, 9]
```

Su sintaxis es:

⟨*lista_comp*⟩ ::= **[**⟨*expresión*⟩ (**for** ⟨*identificador*⟩ **in** ⟨*secuencia*⟩ [**if** ⟨*condición*⟩])+**]**

Los elementos de la salida generada serán los sucesivos valores de ⟨*expresión*⟩.

Las cláusulas **if** son opcionales. Si están, la ⟨*expresión*⟩ sólo se evaluará y añadirá al resultado cuando se cumpla la ⟨*condición*⟩.

Por ejemplo:

```
>>> sec1 = 'abc'
>>> sec2 = (1, 2, 3)
>>> [(x, y) for x in sec1 for y in sec2]
[('a', 1), ('a', 2), ('a', 3),
 ('b', 1), ('b', 2), ('b', 3),
 ('c', 1), ('c', 2), ('c', 3)]
```

Bibliografía

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, Mass. : New York: MIT Press ; McGraw-Hill.