Evaluación

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2025/07/31 a las 15:44:00

Índice

1. El modelo de entorno

1.1. Entorno (*environment*)

El **entorno** es una extensión del concepto de *marco*, usado por los lenguajes interpretados en la **resolución de identificadores**, ya que:

> El **entorno** nos da acceso a ***todas* las ligaduras (almacenadas en marcos, es decir, no de atributos de objetos) que son *visibles* en un momento concreto de la ejecución de un programa interpretado**.

El intérprete usa el entorno para resolver los identificadores que se encuentran ligados mediante ligaduras cuya visibilidad depende de un ámbito y que estén, por tanto, almacenadas en un marco.

Por tanto, no lo usa para resolver los identificadores asociados a atributos de objetos.

Durante la ejecución del programa, se van creando y destruyendo marcos a medida que se van entrando y saliendo de ciertos ámbitos; en concreto, a medida que se van ejecutando *scripts*, funciones o métodos.

Asimismo, en esos marcos se van almacenando ligaduras.

Según se van creando en memoria, esos marcos van enlazándose unos con otros creando una **secuencia de marcos** que se denomina **entorno** (del inglés, *environment*).

En un momento dado, el entorno contendrá más o menos marcos dependiendo de por dónde haya pasado la ejecución del programa hasta ese momento.

El entorno, por tanto, es un concepto ***dinámico*** que **depende del momento en el que se calcule**, es decir, de por dónde va la ejecución del programa.

Más concretamente: depende de qué *scripts*, funciones, métodos y definiciones se han ejecutado hasta ahora.

Por tanto, el entorno depende de qué partes del programa se han ido ejecutando hasta llegar a la instrucción actual.

El entorno **siempre contendrá**, al menos, un marco: el *marco global*, que **siempre será *el último* de la secuencia de marcos** que forman el entorno.

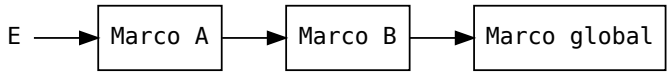
Asimismo, el primer marco del entorno se denomina el **marco actual**.

Si el marco global es el único que existe, entonces el marco actual será el marco global.

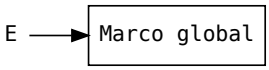
Gráficamente, representaremos los entornos como una **lista enlazada de marcos** conectados entre sí formando secuencias, de manera que:

- Usaremos la letra *E* como un indicador que siempre apunta al primer marco de la lista.

  Ese primer marco es el **marco actual**.
- El último marco siempre será el marco global.



Si sólo hay un marco en el entorno, ése será necesariamente el marco global, el cual será también al mismo tiempo el marco actual:

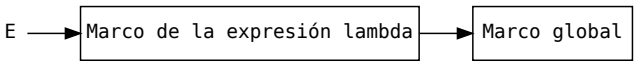


Por ejemplo:

- Cuando entramos a ejecutar un *script*, se crea su *marco global*.

- Si dentro de ese *script* llamamos a una expresión lambda, se creará un marco para esa ejecución concreta de la expresión lambda, por lo que en ese caso habrá dos marcos en la memoria: el *global* y el de esa llamada a la expresión lambda.

  El marco de la expresión lambda será el marco actual, que será el primer marco del entorno y apuntará a su vez al marco global.



- El marco de la expresión lambda se eliminará de la memoria cuando termine esa ejecución de la expresión lambda.
- A su vez, el marco global sólo se eliminará de la memoria cuando se finalice la ejecución del *script*.

1.1.1. Ámbitos, marcos y entornos

Hagamos un resumen rápido de todo lo visto hasta ahora.

El entorno contiene todas las ligaduras visibles en un punto concreto de la ejecución del programa interpretado, siempre que sean ligaduras cuya visibilidad dependa de un ámbito y estén, por tanto, almacenadas en un marco (o sea, no es el caso de los atributos de objetos).

Un marco contiene un conjunto de ligaduras (ya que es un *espacio de nombres*), y un entorno es una secuencia de marcos.

Los marcos se van creando y destruyendo a medida que se van ejecutando y terminando de ejecutar ciertas partes del programa (*scripts*, funciones o métodos).

Una expresión lambda representa una función.

Cuando se llama a una función, se crea un nuevo marco que contiene las ligaduras que ligan a los parámetros con los valores de esos argumentos.

El cuerpo de una expresión lambda determina su propio ámbito, de forma que las ligaduras que ligan a los parámetros con los argumentos se definen dentro de ese ámbito y son, por tanto, *locales* a ese ámbito.

Es decir: los parámetros (y las ligaduras entre los parámetros y los argumentos) tienen **un ámbito local** al cuerpo de la expresión lambda y sólo son visibles dentro de él.

Además, esas ligaduras tienen un **almacenamiento local** al **marco** que se crea al llamar a la expresión lambda.

Ese **marco** y ese **ámbito** van ligados:

- Cuando **se *empieza*** a ejecutar el cuerpo de la expresión lambda, **se *entra*** en el ámbito y, por tanto, **se *crea*** el marco en la memoria.

- Cuando **se *termina*** de ejecutar el cuerpo de la expresión lambda, **se *sale*** del ámbito y, por tanto, **se *elimina*** el marco de la memoria.

Todo *marco* lleva asociado un *ámbito* (lo contrario no siempre es cierto).

Cuando se crea el nuevo marco, éste se enlaza con el marco que hasta ese momento había sido el marco actual, en cadena.

El último marco de la cadena es siempre el marco global.

Se va formando así una **secuencia de marcos** que representa el **entorno** del programa allí donde se está ejecutando la instrucción actual.

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 llame a una expresión lambda y mientras dure la ejecución de la misma.

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 se van ejecutando y terminando de ejecutar ciertas partes del programa: *scripts*, funciones y métodos.

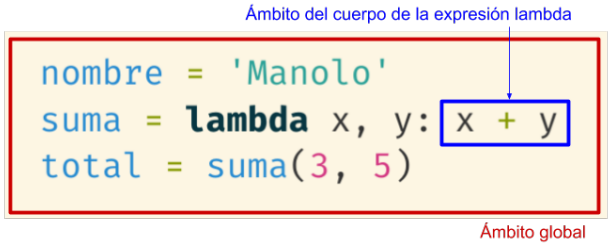
Un marco se crea cuando se **entra** en el **ámbito** de un *script*, función o método, y **se destruye** cuando se **sale** de ese **ámbito**.

Por ejemplo, en el siguiente código:

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

el cuerpo de la función `suma` define un nuevo ámbito.

Por tanto, en el siguiente código tenemos dos ámbitos: el ámbito global (más externo) y el ámbito del cuerpo de la expresión lambda (más interno y anidado dentro del ámbito global):



Además, cada vez que se llama a `suma`, la ejecución del programa entra en su cuerpo, lo que crea un nuevo marco que almacena las ligaduras entre sus parámetros y los argumentos usados en esa llamada.

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

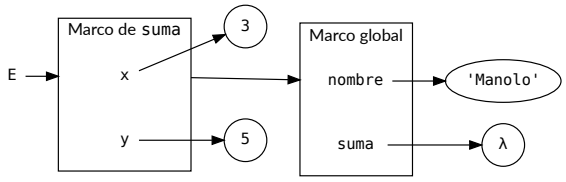
Si un marco *A* apunta a un marco *B* en el entorno, significa que el ámbito de *A* está contenido en el ámbito de *B*.

El **primer marco** en la cadena del entorno siempre será el último marco que se ha creado y que todavía no se ha destruido.

Ese marco es el **marco actual**, y se corresponde con el ámbito actual, es decir, con el ámbito más interno de la instrucción actual.

Por otra parte, el **último marco** del entorno siempre es el *marco global*.

Por ejemplo, si en cierto momento de la ejecución del programa anterior tenemos el siguiente entorno:



Podemos afirmar que:

- El marco de la función `suma` apunta al marco global en el entorno.
- El ámbito de la expresión lambda a la que está ligado `suma` está contenido en el ámbito global.
- El marco actual es el marco de la expresión lambda.
- Por tanto, el programa se encuentra actualmente ejecutando el cuerpo de la expresión lambda.
- De hecho, está ejecutando la llamada `suma(3, 5)`.

1.1.2. Evaluación de expresiones con entornos

Al evaluar una expresión, el intérprete **buscará en el entorno el valor al que está ligado cada identificador** que aparezca en la expresión.

Para ello, el intérprete buscará **en el primer marco del entorno** (el *marco actual*) una ligadura para ese identificador y, si no la encuentra, **irá pasando por toda la secuencia de marcos** hasta encontrarla.

Si no aparece en ningún marco, querrá decir que:

- o bien el identificador **no está ligado** (porque aún no se ha creado la ligadura),
- o bien su ligadura **está fuera del entorno** y por tanto no es visible actualmente (al encontrarse en otro ámbito inaccesible desde el ámbito actual).

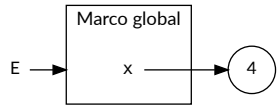
En cualquiera de estos casos, **generará un error** de tipo `NameError` («*nombre no definido*»).

Por ejemplo:

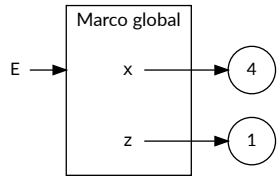
```
1 x = 4
2 z = 1
3 suma = (lambda x, y: x + y + z)(8, 12)
4 y = 3
```

```
5 w = 9
```

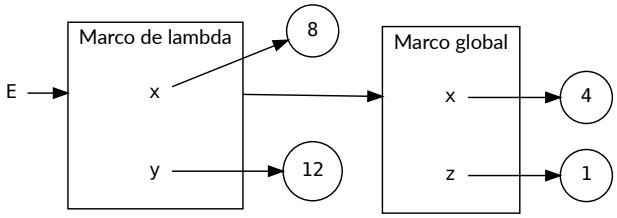
A medida que vamos ejecutando cada línea del código, tendríamos los siguientes entornos:



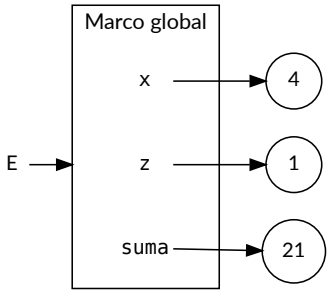
Entorno justo tras ejecutar la línea 1



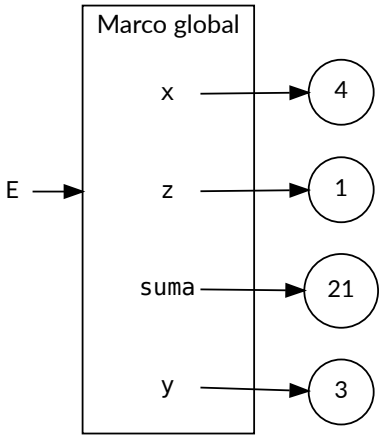
Entorno justo tras ejecutar la línea 2



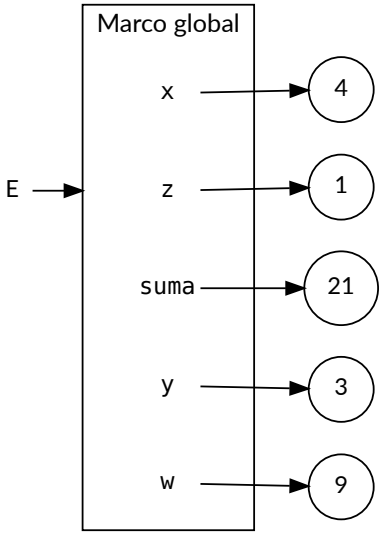
Entorno en la línea 3 en el cuerpo de la expresión lambda, después de aplicar los argumentos y **durante** la ejecución del cuerpo



Entorno en la línea 3, **después** de ejecutar el cuerpo y devolver el resultado



Entorno justo tras ejecutar la línea 4



Entorno justo tras ejecutar la línea 5

1.1.3. Evaluación de expresiones lambda con entornos

Para que una expresión lambda funcione, todos los identificadores que aparezcan en el cuerpo deben estar ligados a algún valor en el entorno **en el momento de *evaluar la aplicación* de la expresión lambda sobre unos argumentos**.

Por ejemplo:

```
>>> prueba = lambda x, y: x + y + z  # aquí no da error
>>> prueba(4, 3)                     # aquí sí
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <lambda>
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:

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

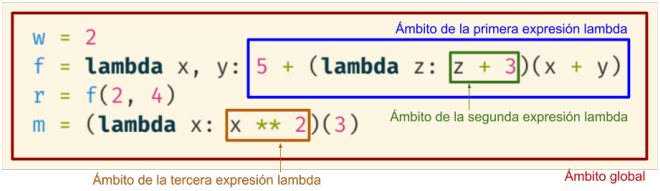
Observar que no es necesario que los identificadores que aparecen en el cuerpo estén ligados en el entorno cuando *se crea* la expresión lambda, sino cuando ***se evalúa* el cuerpo de la expresión lambda**, o sea, cuando se llama a la expresión lambda.

Ejemplo

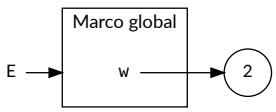
En el siguiente *script*:

```
w = 2
f = lambda x, y: 5 + (lambda z: z + 3)(x + y)
r = f(2, 4)
m = (lambda x: x ** 2)(3)
```

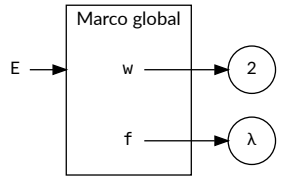
existen cuatro ámbitos:



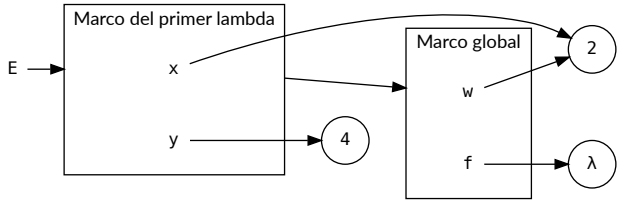
Su ejecución, línea a línea, produce los siguientes entornos:



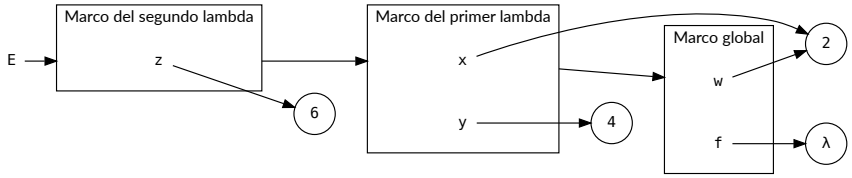
Entorno justo tras ejecutar la línea 1



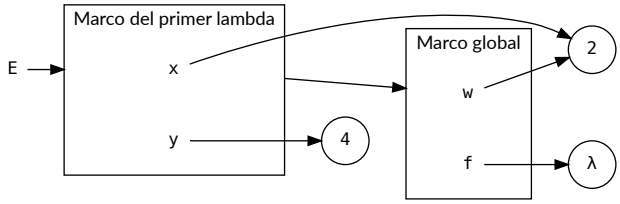
Entorno justo tras ejecutar la línea 2



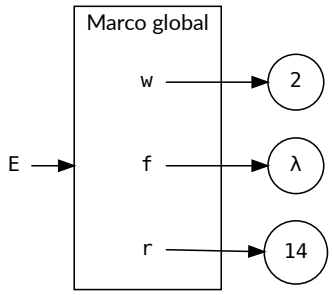
Entorno en la línea 3 en el cuerpo de la primera expresión lambda, después de aplicar sus argumentos y durante la ejecución de su cuerpo



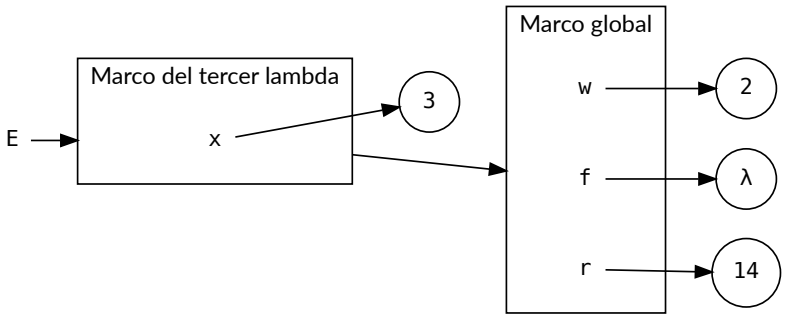
Entorno en la línea 3 en el cuerpo de la segunda expresión lambda, después de aplicar sus argumentos y durante la ejecución de su cuerpo



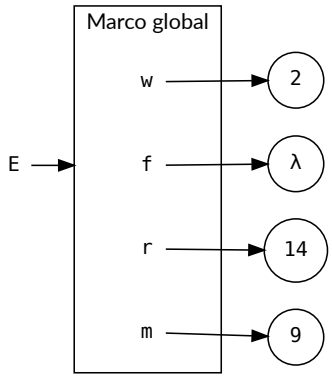
Entorno en la línea 3 en el cuerpo de la segunda expresión lambda, después de ejecutar su cuerpo y devolver su resultado



Entorno en la línea 3 en el cuerpo de la primera expresión lambda, después de ejecutar su cuerpo y devolver su resultado



Entorno en la línea 4 en el cuerpo de la tercera expresión lambda, después de aplicar sus argumentos y durante la ejecución de su cuerpo



Entorno justo tras ejecutar la línea 4

1.1.3.1. Ligaduras *sombreadas*

Recordemos que la **resolución de identificadores** es el proceso por el cual el compilador o el intérprete determinan qué ligadura se corresponde con una aparición concreta de un determinado identificador.

¿Qué ocurre cuando una expresión lambda contiene como parámetros algunos identificadores que ya están ligados en el entorno, en un espacio de nombres asociado a un ámbito más global?

Por ejemplo:

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

¿Cómo resuelve el intérprete de Python las distintas x que aparecen en el código? ¿Son la misma x? ¿Se corresponden con la misma ligadura? ¿Están todas esas x ligadas al mismo valor?

La x que aparece en la línea 1 es distinta a las que aparecen en la 2:

- La x de la línea 1 es un identificador ligado a un valor en el ámbito global (el ámbito de esa ligadura es el ámbito global). Esa ligadura, se almacena en el marco global, y por eso decimos que esa x es *global*.

  Por tanto, la aparición de la x en la línea 1 representa a la x cuya ligadura se encuentra almacenada en el marco global (es decir, la x global) y que está ligada al valor 4.

- Las x de la línea 2 representan al parámetro de la expresión lambda. Ese parámetro está ligado al argumento de la llamada, el ámbito de esa ligadura es el cuerpo de la expresión lambda y esa ligadura se almacena en el marco de la llamada a la expresión lambda.

  En consecuencia, las apariciones de la x en la línea 2 representan a la x *local* a la expresión lambda, cuya ligadura se encuentra almacenada en el marco de la llamada a la expresión lambda y que está ligada a 3.

Por tanto, la x que aparece en el cuerpo de la expresión lambda **no** se refiere al identificador x que está fuera de la expresión lambda (y que aquí está ligado al valor 4), sino al parámetro x que, en la llamada de la línea 2, está ligado al valor 3 (el argumento de la llamada).

Es decir:

- Dentro del cuerpo de la expresión lambda, x vale 3.
- Fuera del cuerpo de la expresión lambda, x vale 4.

Para determinar cuánto vale cada aparición de la x en ese código (es decir, para *resolver* la aparición de cada x), el intérprete de Python consulta el **entorno**.

La x que está en la línea 1 y las x que están en la línea 2 son apariciones distintas que se corresponden con ligaduras distintas que tienen ámbitos distintos y se almacenan en espacios de nombres distintos.

Por tanto, el identificador x podrá tener valores distintos dependiendo de qué aparición concreta de la x estamos evaluando.

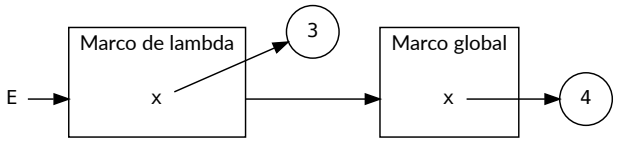
Cuando un mismo identificador está ligado dos veces en dos ámbitos anidados uno dentro del otro, decimos que:

- El identificador que aparece en el ámbito más externo está **sombreado** (y su ligadura está **sombreada**) por el del ámbito más interno.
- El identificador que aparece en el ámbito más interno **hace sombra** al identificador sombreado (y también se dice que su ligadura **hace sombra** a la ligadura sombreada) que aparece en el ámbito más externo.

En nuestro ejemplo, podemos decir que el parámetro x de la expresión lambda hace sombra al identificador x que aparece en el ámbito global.

Eso significa que no podemos acceder a ese identificador x global desde dentro del cuerpo de la expresión lambda, porque la x dentro del cuerpo siempre se referirá a la x local (el parámetro de la expresión lambda).

Esto ocurre así porque, al buscar un valor para x, la primera ligadura que se encuentra el intérprete para el identificador x al recorrer la secuencia de marcos del entorno, es precisamente la que está en el marco de la expresión lambda, que es el marco actual cuando se está ejecutando su cuerpo.

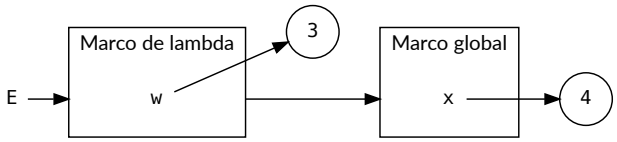


Entorno en el cuerpo de la expresión lambda, con ligadura sombreada

Si desde dentro de la expresión lambda necesitáramos acceder al valor de la x que está fuera de ese expresión lambda, lo que podríamos hacer es **cambiarle el nombre** al parámetro x. Por ejemplo:

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

Así, en la expresión lambda tendríamos el parámetro w y el identificador libre x, éste último ligado en el ámbito global, y a cuyo valor ahora sí podemos acceder al no estar sombreado y encontrarse dentro del entorno.



Entorno en el cuerpo de la expresión lambda, sin identificador sombreado

1.1.3.2. Renombrado de parámetros

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

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 identificadores**.

1.1.3.3. Visualización en *Pythontutor*

Pythontutor es una herramienta online muy interesante y práctica que nos permite ejecutar un *script* paso a paso y visualizar sus efectos.

Muestra la pila de control, los marcos dentro de ésta, las ligaduras dentro de éstos y los datos almacenados en el montículo.

Entrando en http://pythontutor.com/visualize.html se abre un área de texto donde se puede teclear (o copiar y pegar) el código fuente del *script* a ejecutar.

Pulsando en «*Visualize Execution*» se pone en marcha, pudiendo ejecutar todo el *script* de una vez o hacerlo paso a paso.

Conviene elegir las siguientes opciones:

- *Hide exited frames (default)*
- *Render all objects on the heap (Python/Java)*
- *Draw pointers as arrows (default)*

Visualizar el *script* anterior en Pythontutor

Ejercicio

1. En el *script* anterior:

```
w = 2
f = lambda x, y: 5 + (lambda z: z + 3)(x + y)
r = f(2, 4)
m = (lambda x: x ** 2)(3)
```

   indicar:

a. Los identificadores.
b. Los ámbitos.
c. Los entornos, marcos y ligaduras en cada línea de código.
d. Los ámbitos de cada ligadura.
e. La visibilidad de cada ligadura.
f. El tiempo de vida de cada ligadura.
g. El almacenamiento de cada ligadura.
h. Los ámbitos de cada aparición de cada identificador.
i. Las ligaduras sombreadas y los identificadores sombreados.
j. Los identificadores y ligaduras que hacen sombra.

1.2. Pureza

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

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

Es el caso del siguiente ejemplo, donde tenemos una expresión lambda que contiene un identificador 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` en el entorno:

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

En este otro ejemplo, tenemos 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 ligada a `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 o, lo que es lo mismo, del valor de sus parámetros en la llamada.

Podemos decir que hay distintos **grados de pureza**:

- Una expresión lambda en cuyo cuerpo no hay ningún identificador libre es **más pura** que otra que contiene identificadores libres.

- Una expresión lambda cuyos **identificadores libres** representan **funciones** que se usan en el cuerpo de la expresión lambda, es **más pura** que otra cuyos identificadores libres representan 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 único identificador libre (`suma`) se usa como una **función**.

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 identificadores libres (en este
# caso, sólo una: producto) son funciones:
cuadrado = lambda x: producto(x, x)
# suma es impura, porque su identificador libre (z) 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.

2. Resolución de atributos de objetos

2.1. Resolución de atributos de objetos

Ya estudiamos que el acceso a un atributo de un objeto suponía buscar la correspondiente ligadura únicamente en el espacio de nombres asociado a ese objeto, y no en ningún otro.

Por tanto, dicha resolución requiere de un mecanismo algo distinto a lo visto hasta ahora, ya que las ligaduras que ligan el nombre del atributo con su valor se almacenan en el propio objeto, no en un marco.

En consecuencia, el acceso a un atributo de un objeto usando el operador punto (`.`), como en la expresión `math.pi` de este ejemplo:

```
import math
x = math.pi * 2
```

no requiere usar el entorno.

De hecho, el lenguaje ni siquiera tiene por qué tener entornos. Recordemos que los lenguajes compilados no usan entornos para resolver identificadores y pueden resolver perfectamente los atributos de los objetos.

Concretamente, resolver el identificador `pi` en la expresión `math.pi` requerirá de los siguientes pasos:

1. Se busca el valor de `math` en el entorno, que devuelve el objeto que representa al módulo `math`.
2. Una vez que sabemos que el operando izquierdo del operador punto (`.`) es un objeto, procedemos a resolver el identificador `pi`, pero para ello sólo se considera el espacio de nombres asociado al objeto `math`.

   Es decir: buscamos el valor de `pi` en el espacio de nombres de `math`, y sólo ahí.
3. Una vez localizado, se devolverá el valor ligado al nombre `pi` en el espacio de nombres de `math`, o se lanzará un error `NameError` en caso de que no haya ninguna ligadura para `pi` en `math`.

Como se puede observar, en ningún momento se usa el entorno para resolver el identificador `pi` dentro de `math`.

3. La pila de control

3.1. La pila de control

La **pila de control** es una estructura de datos que utiliza el intérprete para llevar la cuenta de las **ejecuciones *activas*** que hay en un determinado momento de la ejecución del programa.

Las **ejecuciones activas** son aquellas llamadas a funciones (o ejecuciones de *scripts*) que aún no han terminado de ejecutarse.

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

El marco que hay en el fondo de la pila siempre es el *marco global* y se corresponde con el espacio de nombres del *script* actual.

Cada vez que se hace una nueva llamada a una función:

1. la ejecución actual se detiene,
2. el marco correspondiente a esa llamada **se almacena en la cima de la pila** sobre los demás marcos que pudiera haber y
3. se continúa la ejecución en la función llamada.

Ese marco representa por dónde va la ejecución del programa en este momento.

Según ésto, si un marco *A* está justo debajo de otro marco *B*, es porque el código correspondiente a *A* está esperando a que termine el código correspondiente a *B* (normalmente, una función).

Ese marco además es el primero de la secuencia de marcos que forman el entorno de la función, que también estarán almacenados en la pila, más abajo.

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

El intérprete almacena en el marco cualquier información que necesite para gestionar las llamadas a funciones, incluyendo:

- Las ligaduras entre los parámetros y sus valores (por supuesto).

- La ligadura que apunta al valor de retorno de la función.
- Cuál es el siguiente marco que le sigue en el entorno.
- El punto de retorno, dentro del programa, al que debe devolverse el control cuando finalice la ejecución de la función.

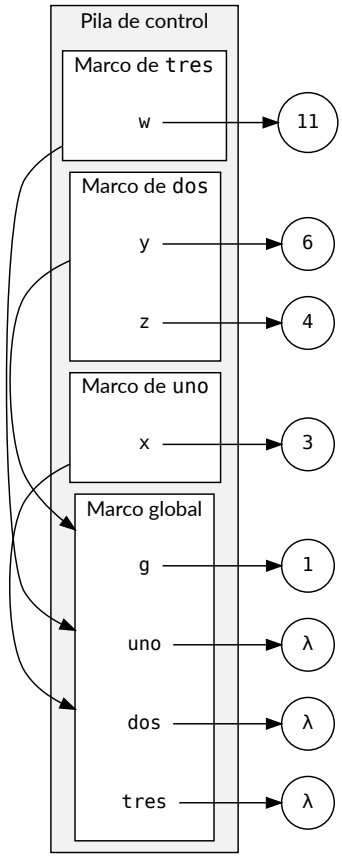
Un marco almacenado en la pila también se denomina **registro de activación**. Por tanto, también podemos decir que la pila de control almacena registros de activación.

Cada llamada activa está representada por su correspondiente marco en la pila.

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

Ejemplos

```
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)
```



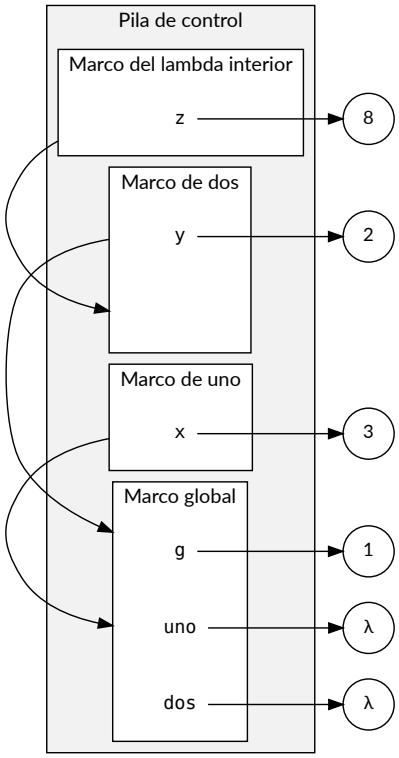
Pila de control con la llamada a la función `tres` activada

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 `dos`, el cual apunta al marco global.
- El entorno en la función `tres` empieza por el marco de `tres`, el cual apunta al marco global.

Si tenemos ámbitos anidados, los marcos se apuntarán entre sí en el entorno. Por ejemplo:

```
g = 1
uno = lambda x: dos(x - 1)
dos = lambda y: 1 + (lambda z: z * 2)(y ** 3)
uno(3)
```

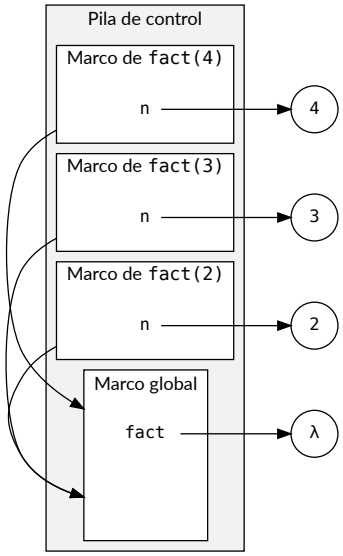


Pila de control con ámbitos anidados y la función `dos` activada

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

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

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



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

Los **traductores que optimizan la recursividad final** lo que hacen es sustituir cada llamada recursiva por la nueva llamada recursiva a la misma función.

De esta forma, el marco que genera cada nueva llamada recursiva no se apila sobre los marcos anteriores en la pila, sino que sustituye al marco de la llamada que la ha llamado a ella.

Por ejemplo, en el siguiente caso:

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

fact(5)
= fact_iter(5, 1)
= fact_iter(4, 5)
= fact_iter(3, 20)
= fact_iter(2, 60)
= fact_iter(1, 120)
= fact_iter(0, 120)
= 120
```

`fact_iter(4, 5)` llama a `fact_iter(3, 20)` y devuelve directamente el resultado de ésta.

Es decir: `fact_iter(4, 5) == fact_iter(3, 20)`, así que hacer `fact_iter(4, 5)` es lo mismo que hacer `fact_iter(3, 20)`.

Por tanto, la llamada a `fact_iter(4, 5)` se puede sustituir por la llamada a `fact_iter(3, 20)`.

Un intérprete que optimiza la recursividad final no apilaría el marco de la segunda llamada sobre el marco de la primera, sino que el marco de la segunda sustituiría al marco de la primera dentro de la pila.

Así se haría también con las demás llamadas recursivas a `fact_iter(2, 60)`, `fact_iter(1, 120)` y `fact_iter(0, 120)`.

De este modo, la pila no crecería con cada nueva llamada recursiva.

4. Estrategias de evaluación

4.1. 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 subexpresión 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 subexpresión.

4.1.1. 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 subexpresiones (también llamadas *redexes*, del inglés, *reducible expression*).

Pero eso también implica que **no importa el orden en el que se evalúen las subexpresiones**: 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.

4.1.1.1. Orden aplicativo

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

El *redex* más interno es el que no contiene a otros *redexes*. Si existe más de uno que cumpla esa condición, se elige el que está más a la izquierda.

Eso implica que los operandos y los argumentos se evalúan **antes** que los operadores y las aplicaciones de funciones.

Corresponde a lo que en muchos lenguajes de programación se denomina **paso de argumentos por valor** (*call-by-value*).

Por ejemplo, si tenemos la siguiente función:

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

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

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

4.1.1.2. Orden normal

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

El *redex* más externo es el que no está contenido en otros *redexes*. Si existe más de uno que cumpla esa condición, se elige el que está más a la izquierda.

Eso implica que los operandos y los argumentos se evalúan **después** de las aplicaciones de los operadores y las funciones.

Por tanto, los argumentos que se pasan a las funciones lo hacen **sin evaluarse** previamente.

Corresponde a lo que en muchos lenguajes de programación se denomina **paso de argumentos por nombre** (*call-by-name*).

Por ejemplo, si tenemos la siguiente función:

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

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

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

4.1.2. 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)                                # definición de area
2  = (lambda r: 3.1416 * cuadrado(r))(11 + 1)  # evalúa 11 y devuelve 11
3  = (lambda r: 3.1416 * cuadrado(r))(11 + 1)  # evalúa 1 y devuelve 1
4  = (lambda r: 3.1416 * cuadrado(r))(11 + 1)  # evalúa 11 + 1 y devuelve 12
5  = (lambda r: 3.1416 * cuadrado(r))(12)      # aplicación a 12
6  = (3.1416 * cuadrado(12))                   # evalúa 3.1416 y devuelve 3.1416
7  = (3.1416 * cuadrado(12))                   # definición de cuadrado
8  = (3.1416 * (lambda x: x * x)(12))          # aplicación a 12
9  = (3.1416 * (12 * 12))                      # evalúa (12 * 12) y devuelve 144
10 = (3.1416 * 144)                            # evalúa (3.1416 * 11) y...
11 = 452.3904                                  # ... devuelve 452.3904
```

En detalle:

- **Línea 1**: Se evalúa `area`, que devuelve su definición (una expresión lambda).
- **Líneas 2–4**: Lo siguiente a evaluar es la aplicación de `area` sobre su argumento, por lo que primero evaluamos éste (es el *redex* más interno).
- **Línea 5**: Ahora se aplica la expresión lambda a su argumento `12`.
- **Línea 6**: El *redex* más interno y a la izquierda es el `3.1416`, que ya está evaluado.
- **Línea 7**: El *redex* más interno que queda por evaluar es la aplicación de `cuadrado` sobre `12`. Primero se evalúa `cuadrado`, sustituyéndose por su definición...
- **Línea 8**: ... y ahora se aplica la expresión lambda a su argumento `12`.
- Lo que queda es todo aritmética.

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 a (11 + 1)
3  = (3.1416 * cuadrado(11 + 1))               # evalúa 3.1416 y devuelve 3.1416
4  = (3.1416 * cuadrado(11 + 1))               # definición de cuadrado
5  = (3.1416 * (lambda x: x * x)(11 + 1))      # aplicación a (11 + 1)
6  = (3.1416 * ((11 + 1) * (11 + 1)))          # evalúa (11 + 1) y devuelve 12
7  = (3.1416 * (12 * (11 + 1)))                # evalúa (11 + 1) y devuelve 12
8  = (3.1416 * (12 * 12))                      # evalúa (12 * 12) y devuelve 144
9  = (3.1416 * 144)                            # evalúa (3.1416 * 144) y...
10 = 452.3904                                  # ... devuelve 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(11 + 1)`. Para ello, se reescribe la definición de `area`...
- **Línea 2**: ... y se aplica la expresión lambda al argumento `11 + 1`.
- **Línea 3**: El *redex* más externo es el `*`, pero para evaluarlo hay que evaluar primero todos sus argumentos, por lo que primero se evalúa el izquierdo, que es `3.1416`.
- **Línea 4**: Ahora hay que evaluar el derecho (`cuadrado(11 + 1)`), por lo que se reescribe la definición de `cuadrado`...
- **Línea 5**: ... y se aplica la expresión lambda al argumento `11 + 1`.
- Lo que queda es todo aritmética.

A veces no resulta fácil determinar si un *redex* es más interno o externo que otro, sobre todo cuando se mezclan funciones y operadores en una misma expresión.

En ese caso, puede resultar útil reescribir los operadores como funciones, cuando sea posible.

Por ejemplo, la siguiente expresión:

```
abs(-12) + max(13, 28)
```

se puede reescribir como:

```
from operator import add
add(abs(-12), max(13, 28))
```

lo que muestra claramente que la suma es más externa que el valor absoluto y el máximo (que están, a su vez, al mismo nivel de profundidad).

Un ejemplo más complicado:

```
abs(-12) * max((2 + 3) ** 5), 37)
```

se reescribiría como:

```
from operator import add, mul
mul(abs(-12), max(pow(add(2, 3), 5), 37))
```

donde se aprecia claramente que el orden de las operaciones, de más interna a más externa, sería:

1. Suma (`+` o `add`).
2. Potencia (`**` o `pow`).
3. Valor absoluto (`abs`) y máximo (`max`) al mismo nivel.
4. Producto (`*` o `mul`).

4.1.3. Evaluación estricta y no estricta

Existe otra forma de ver la evaluación de una expresión:

- **Evaluación estricta o *impaciente***: Reducir todos los *redexes* aunque no hagan falta para calcular el valor de la expresión.
- **Evaluación no estricta o *perezosa***: Reducir sólo los *redexes* que sean estrictamente necesarios para calcular el valor de la expresión.

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 de estandarización:

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*⟩ dependiendo del valor de la ⟨*condición*⟩.

- Los operadores lógicos **`and`** y **`or`** también son perezosos (se dice que evalúan **en cortocircuito**):

  * `True or` *x*

    siempre es igual a `True`, valga lo que valga *x*.

  * `False and` *x*

    siempre es igual a `False`, valga lo que valga *x*.

  En ambos casos no es necesario evaluar *x*.

En Java también existe un operador ternario (`? :`) y unos operadores lógicos (`||` y `&&`) que se evalúan de igual forma que en Python.

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.

La evaluación perezosa en Haskell permite resultados muy interesantes, como la posibilidad de manipular estructuras de datos infinitas.

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.