Programación funcional II

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Abstracciones funcionales **2**
1.1. Expresiones lambda . . . 2
1.1.1. Parámetros y cuerpos . . . 2
1.1.2. Aplicación funcional . . . 2
1.1.3. Variables ligadas y libres . . . 4
1.1.4. Ámbito de una variable ligada . . . 5
1.1.5. Variables *sombreadas* . . . 6
1.1.6. Expresiones lambda y entornos . . . 6
1.1.7. Renombrado de parámetros . . . 8
1.2. Estrategias de evaluación . . . 9
1.2.1. Orden de evaluación . . . 9
1.2.2. Evaluación estricta y no estricta . . . 10
1.3. Composición de funciones . . . 12

2. Tipos de datos compuestos **12**
2.1. Cadenas . . . 12
2.2. Listas . . . 12

3. Computabilidad **13**
3.1. Funciones y procesos . . . 13
3.2. Funciones recursivas . . . 13
3.2.1. Definición . . . 13
3.2.2. Casos base y casos recursivos . . . 14
3.2.3. El factorial . . . 14
3.2.4. Recursividad lineal . . . 15
3.2.5. Recursividad en árbol . . . 15
3.3. Un lenguaje Turing-completo . . . 15

4. Funciones de orden superior **15**
4.1. `map()` . . . 15
4.2. `filter()` . . . 15
4.3. `reduce()` . . . 15

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 producía 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 denota 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**, el **orden de evaluación** de cualquier expresión es **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)
= (4 + 3) * suma(2, 7)
= 7 * suma(2, 7)
= 7 * (lambda x, y: x + y)(2, 7)
= 7 * (2 + 7)
= 7 * 9
= 63
```

1.1.3. Variables ligadas y libres

- Si un identificador aparece en la lista de parámetros de la 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.

- 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:

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

da error porque `z` no está definido (no está ligado a ningún valor en el entorno).

- En cambio:

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

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

- Observar que no es necesario que las variables libres estén ligadas en el entorno cuando se *crea* la expresión lambda, sino cuando se *aplica*.
- Una expresión lambda cuyo cuerpo sólo contiene variables ligadas, es una expresión cuyo valor sólo va a depender de los argumentos que se usen cuando se aplique la expresión lambda.
- En cambio, el valor de una expresión lambda que contenga variables libres dependerá no sólo de los valores de sus argumentos, sino también de los valores a los que estén ligadas las variables libres al evaluar la expresión lambda.
- Por ejemplo, podemos escribir una expresión lambda que calcule la suma de tres números a partir de otra expresión lambda que calcule la suma de dos números:

```
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 (por lo que es una variable libre).

Por tanto, el valor de la expresión lambda anterior dependerá de lo que valga `suma` (de lo que haga, de lo que devuelva...).

1.1.4. Ámbito de una variable ligada

- Recordemos que el **ámbito de una ligadura** es la porción del programa en la que dicha ligadura tiene validez.
- 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.
- Por contraste, los identificadores que no tienen ámbito local se dice que tienen un **ámbito 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.

1.1.5. Variables *sombreadas*

- ¿Qué ocurre cuando una expresión lambda contiene como parámetros nombres que ya están definidos (ligados) en el entorno?
- 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 diferente a la que aparece en la lista de parámetros de la expresión lambda de la línea 2.
- En este caso, decimos que **el parámetro x *hace sombra*** al identificador x que, en el entorno, está ligado al valor 4.
- Por tanto, 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).
- 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.
- Si necesitáramos acceder 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:

```
1 x = 4
2 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).

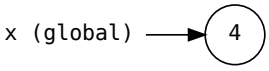
1.1.6. Expresiones lambda y entornos

- Recordemos que el **entorno** es el conjunto de todas las ligaduras que son accesibles en un punto concreto de un programa.

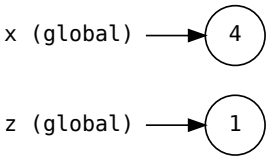
- Para calcular el entorno en un punto dado, debemos tener en cuenta las ligaduras, así como los ámbitos de dichas ligaduras y las variables ligadas que hagan sombra a otras situadas en el mismo ámbito.
- Por ejemplo:

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

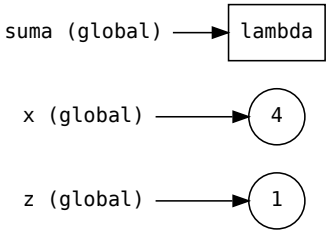
- En cada línea tendríamos los siguientes entornos:



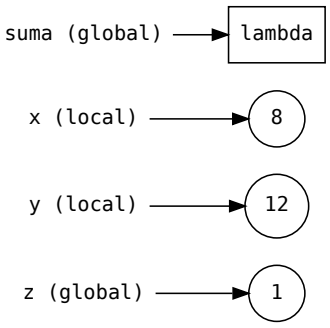
Entorno en la línea 1



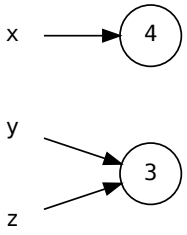
Entorno en la línea 2



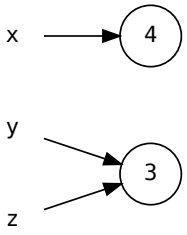
Entorno en la línea 3 fuera de la expresión lambda



Entorno en la línea 3 en el cuerpo de la expresión lambda



Entorno en la línea 4



Entorno en la línea 5

1.1.7. 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**.

1.2. Estrategias de evaluación

- A la hora de evaluar una 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 fuera adentro o de dentro afuera).
  - La necesidad o no de evaluar dicha 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**.
  - **Orden normal**: reducir siempre el *redex* más **externo**.
- **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**.
- Corresponde a lo que en muchos lenguajes de programación se denomina **paso de argumentos por valor**.

- Ejemplo:

```
cuadrado = lambda x, y: x * x
```

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

```
cuadrado(3 + 4)
= cuadrado(7)
= (lambda x, y: x * x)(7)
= (7 * 7)
= 49
```

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

```
cuadrado = lambda x, y: x * x
```

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

```
cuadrado(3 + 4)
= (lambda x, y: x * x)(3 + 4)
= (3 + 4) * (3 + 4)
= 7 * (3 + 4)
= 7 * 7
= 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 `3`.
- 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
    ```
- En **Python** la evaluación es **estricta**, salvo algunas excepciones:
  - El operador ternario:

    ```
    <expresión_condicional> ::= <valor_si_verdadero> if <condición> else <valor_si_falso>
    ```

    evalúa perezosamente *<valor_si_verdadero>* 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 se basan en la evaluación estricta y el paso de argumentos por valor (siguen el orden aplicativo).
- **Haskell**, por ejemplo, es un lenguaje funcional puro que se basa en la 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(12)` se evaluaría así según el *orden aplicativo*:

```
area(12)                                        # definición de area
= (lambda r: 3.1416 * cuadrado(r))(12)          # definición de cuadrado
= (lambda r: 3.1416 * (lambda x: x * x)(r))(12) # aplicación
= (3.1416 * (lambda x: x * x)(12))              # aplicación
= (3.1416 * (12 * 12))                          # aritmética
= 452.3904
```

- Y según el *orden normal*:

```
area(12)                                # definición de area
= (lambda r: 3.1416 * cuadrado(r))(12)  # aplicación
= (3.1416 * cuadrado(12))               # definición de cuadrado
= (3.1416 * (lambda x: x * x)(12))      # aplicación
= (3.1416 * (12 * 12))                  # aritmética
= 452.3904
```

- En ambos casos se obtiene el mismo resultado, ya que en todo momento hemos usado *funciones puras*.

2. Tipos de datos compuestos

2.1. Cadenas

- Las **cadenas** se pueden considerar datos compuestos de otros más simples.
- Por ahora consideraremos que una cadena `c` está formada por dos partes:
  - El **primer carácter** de la cadena, si existe (al que se accede mediante `c[0]`).
  - El **resto** de la cadena (al que se accede mediante `c[1:]`).
- 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'
```

2.2. Listas

- Las **listas** son una generalización de las cadenas.

- Una lista es una **secuencia de elementos** que no tienen por qué ser caracteres, sino que pueden ser **de cualquier tipo** (números, cadenas, booleanos, incluso otras listas).
- Los literales de tipo lista se representan enumerando sus elementos separados por comas y encerrados entre corchetes.
- Por ejemplo:

```
lista = [27, 'hola', True, 73.4, ['a', 'b', 'c'], 99]
```

- Con las listas usaremos las mismas operaciones de acceso que con las cadenas (`lista[0]` es el primer elemento de la lista, `lista[1:]` es el resto de la lista, etcétera).

3. Computabilidad

3.1. 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 con sus conjuros.
- 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.

3.2. Funciones recursivas

3.2.1. 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:

  GNU significa **GNU No es Unix**.

  Por tanto, GNU = GNU No es Unix = GNU No es Unix No es Unix...

  Y así hasta el infinito.

3.2.2. 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**).
- Para ello, en algún momento, la recursión debe alcanzar un punto en el que la función no se llame a sí misma.
- La función, en cada paso recursivo, debe ir acercándose cada vez más a ese punto.
- A ese punto o puntos en los que la función recursiva no se llama a sí misma, se les denomina **casos base**.
- Es decir: la función recursiva, ante ciertos valores de sus argumentos, debe devolver directamente un valor y no llamarse de nuevo recursivamente.
- Los demás casos, que sí provocan llamadas recursivas, se denominan **casos recursivos**.

3.2.3. El factorial

- El ejemplo más típico de función recursiva es el **factorial**.
- El factorial de un número natural n se representa $n!$ 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 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:

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

3.2.4. Recursividad lineal

3.2.4.1. Procesos lineales recursivos

3.2.4.2. Procesos lineales iterativos

3.2.5. Recursividad en árbol

3.3. Un lenguaje Turing-completo

4. Funciones de orden superior

4.1. `map()`

4.2. `filter()`

4.3. `reduce()`