Programación funcional (II)

Ricardo Pérez López

IES Doñana, curso 2024/2025

Generado el 2025/07/09 a las 16:10:00

Índice

1. Computabilidad

1.1. 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 llamado **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**, es decir, cómo se debe comportar el proceso durante la ejecución de la función.

En cada paso de la ejecución 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.

1.2. Funciones *ad-hoc*

Supongamos que queremos diseñar una función llamada `permutas` que reciba un número entero n y que calcule cuántas permutaciones distintas podemos hacer con n elementos.

Por ejemplo: si tenemos 3 elementos (digamos, A, B y C), podemos formar con ellos las siguientes permutaciones:

$$ABC, ACB, BAC, BCA, CAB, CBA$$

y, por tanto, con 3 elementos podemos formar 6 permutaciones distintas. En consecuencia, `permutas(3)` debe devolver `6`.

La implementación de esa función deberá satisfacer la siguiente especificación:

$$\begin{cases} \textbf{Pre}: & n \geq 0 \\ & \texttt{permutas(}n\texttt{: int) -> int} \\ \textbf{Post}: & \texttt{permutas(}n\texttt{)} = \text{el número de permutaciones que} \\ & \text{podemos formar con } n \text{ elementos} \end{cases}$$

Un programador con poca idea de programación (o muy listillo) se podría plantear una implementación parecida a la siguiente:

```
permutas = lambda n: 1 if n == 0 else 1 if n == 1 else 2 if n == 2 else ...
```

que se puede escribir mejor usando la barra invertida (\) para poder separar una instrucción en varias líneas:

```
permutas = lambda n: 1 if n == 0 else \
                     1 if n == 1 else \
                     2 if n == 2 else \
                     6 if n == 3 else \
                     24 if n == 4 else \
                     ...                    # sigue y sigue
```

Pero este algoritmo en realidad es *tramposo*, porque no calcula nada, sino que se limita a asociar el dato de entrada con el de salida, que se ha tenido que calcular previamente usando otro procedimiento.

Este tipo de algoritmos se denominan **algoritmos *ad-hoc***, y las funciones que los implementan se denominan **funciones *ad-hoc***.

Las funciones *ad-hoc* **no son convenientes** porque:

- Realmente son **tramposos** (no calculan nada).
- **No son útiles**, porque al final el cálculo se tiene que hacer con otra cosa.
- Generalmente resulta **imposible** que una función de este tipo abarque todos los posibles datos de entrada, ya que, en principio, puede haber **infinitos** y, por tanto, su código fuente también tendría que ser infinito.

Usar algoritmos y funciones *ad-hoc* se penaliza en esta asignatura.

1.3. 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.

2. Tipos de datos recursivos

2.1. Concepto

Un **tipo de dato recursivo** es aquel que puede definirse en términos de sí mismo.

Un **dato recursivo** es un dato que pertenece a un tipo recursivo. Por tanto, es un dato que se construye sobre otros datos del mismo tipo.

Como toda estructura recursiva, un tipo de dato recursivo tiene casos base y casos recursivos:

- En los casos base, el tipo recursivo se define directamente, sin referirse a sí mismo.
- En los casos recursivos, el tipo recursivo se define sobre sí mismo.

La forma más natural de manipular un dato recursivo es usando funciones recursivas.

2.2. Cadenas

Las **cadenas** se pueden considerar **tipos de datos recursivos**, 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*).

En tal caso, se cumple que `c == c[0] + 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'
```

2.3. Tuplas

Las **tuplas** (datos de tipo `tuple`) 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]
27
>>> tupla[1:]
('hola', True, 73.4, ('a', 'b', 'c'), 99)
>>> tupla[1:][0]
'hola'
```

Junto a las operaciones `t[0]` y `t[1:]`, 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)
(1, 2, 3, 4, 5, 6)
```

Eso significa que, si `t` es una tupla no vacía, se cumple que `t == (t[0],) + t[1:]`.

Esta propiedad es similar (aunque no exactamente igual) a la que se cumple en las cadenas no vacías.

2.4. Rangos

Los rangos (datos de tipo `range`) son valores que representan **sencuencias de números enteros**.

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

```
range([start: int,] stop: int [, step: int]) -> range
```

Cuando se omite *start*, se entiende que es `0`.

Cuando se omite *step*, se entiende que es `1`.

El valor de *stop* no se alcanza nunca.

Cuando *start* y *stop* son iguales, representa el *rango vacío*.

step debe ser siempre distinto de cero.

Cuando *start* es mayor que *stop*, el valor de *step* 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.

La **forma normal** de un rango es una expresión en la que se llama a la función `range` con los argumentos necesarios para construir el rango:

```
>>> range(2, 3)
range(2, 3)
>>> range(4)
range(0, 4)
```

```
>>> range(2, 5, 1)
range(2, 5)
>>> range(2, 5, 2)
range(2, 5, 2)
```

El **rango vacío** es un valor que no tiene expresión canónica, ya que cualquiera de las siguientes expresiones representan al rango vacío tan bien como cualquier otra:

- `range(0)`.
- `range(a, a)`, donde a es cualquier entero.
- `range(a, b, c)`, donde $a \geq b$ y $c > 0$.
- `range(a, b, c)`, donde $a \leq b$ y $c < 0$.

```
>>> range(3, 3) == range(4, 4)
True
>>> range(4, 3) == range(3, 4, -1)
True
```

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]
4
>>> rango[1:]
range(5, 7)
>>> rango[1:][0]
5
```

2.5. 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))
()
```

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.