Funciones recursivas

Juan Álvarez Rubio

En matemáticas, algunas funciones se pueden definir a través de recurrencias. El ejemplo emblemático es la función que calcula el factorial de un número entero positivo n. La notación es n! y se define como el producto de todos los números enteros entre 1 y n, es decir, n! =1x 2x … x n, por ejemplo, 4!=1 x 2 x 3 x 4=24. La función factorial también se puede definir a través de la recurrencia matemática n!=(n-1)! x n. Por ejemplo, 4!=3! x 4 y 3!=2! x 3 y 2!=1! x 2 y 1!=0! x 1 y se define 0!=1.

Función para calcular factorial de un número

La versión computacional de una función definida con una recurrencia matemática se denomina función recursiva:

```
#factorial: int -> int
#factorial de entero n >= 0
#ej: factorial(4)->24, factorial(0)->1
def factorial(n):
    assert type(n)==int and n>=0
    if n == 0 :
        return 1
    else:
        return factorial(n-1) * n
assert factorial(4)==24
assert factorial(0)==1
```

Una función recursiva es una función que se invoca a sí misma. Por ejemplo, la función factorial contiene la llamada factorial(n-1). Al respecto, es muy importante que la llamada contenga un argumento menor que el original, de manera que se acerque al "caso base". Caso base es la situación en que la función devuelve un resultado sin llamarse a sí misma, por ejemplo, if n==0: return 1. En otras palabras, las invocaciones o llamadas recursivas deben acercarse o converger al caso base, de manera que el cálculo en algún momento finalice.

La función recursiva factorial contiene una instrucción if con else, por lo que debe probarse al menos en dos casos: el caso base (ejemplo, cero), y un caso general (con un argumento mayor que cero). Por otra parte, la función tiene una precondición que asegura la validez del parámetro: número entero mayor o igual a cero.

La invocación factorial(4) significa que la función tiene que devolver como resultado factorial(3)*4. Pero factorial(3) implica calcular factorial(2)*3. Y factorial(2) debe calcular factorial(1)*2 y factorial(1) será factorial(0)*1. Finalmente, factorial(0) devolverá el resultado 1 en la llamada factorial(0)*1 que había quedado pendiente. A su vez, el resultado 1 se reemplazará en

factorial(1)*2 produciendo un 2. El 2 se reemplazará en factorial(2)*3 que entregará un 6. Y con el 6, finalmente se calculará factorial(3)*4 para devolver finalmente el valor 24, que corresponde al factorial de 4. Los resultados de las llamadas o invocaciones a la función quedan en suspenso hasta llegar al caso base, y su resultado (ejemplo un 1) desencadena el cálculo de los valores pendientes.

Cada llamada a la función factorial crea su propio parámetro n. Es decir, se crearán tantas variables de nombre n como invocaciones a la función, y entre ellas no habrá confusión. La implementación computacional subyacente maneja adecuadamente la situación, y el programador no tiene que preocuparse del funcionamiento interno de las funciones recursivas.

No hay una manera única de escribir una función recursiva. Por ejemplo, la función para calcular el factorial puede escribirse de la siguiente manera:

```
def factorial(n):
    assert type(n)==int and n>=0
    if n <= 1 : return 1
    return n * factorial(n-1)
```

La función se implementó con una instrucción if sin else y el caso general como una instrucción incondicional e independiente. El caso base con la condición n<=1 evita una llamada recursiva adicional considerando que 1! y 0! entregan un resultado 1. Por otra parte, como la multiplicación es una operación aritmética conmutativa, entonces el caso general se calcula con la instrucción return n*factorial(n-1). Esta alternativa se conoce como recursión "al final" del cálculo (*tail recursion*) y return factorial(n-1)*n es una recursión "al comienzo" del cálculo (*head recursion*).

Función para contar dígitos de un número entero

Para calcular la cantidad de dígitos de un número entero n se puede plantear la recurrencia dígitos(n) = 1 + dígitos(n sin el último dígito) e implementarla con la siguiente función recursiva:

```
#digitos: int -> int
#cantidad de digitos de n
#ej: digitos(245)->3, digitos(-4)->1
def digitos(n):
    assert type(n)==int
    if abs(n) < 10 :
        return 1
    else:
        return 1 + digitos(n//10)
assert digitos(245)==3
assert digitos(-4)==1  #caso base
```

La precondición asegura que el parámetro de la función sea un número entero. El caso base se implementa con la instrucción if abs(n)<10 que detecta si el número, positivo o negativo, tiene un

solo dígito. Por otra parte, la expresión n//10 entrega el número original pero sin el último dígito. Las pruebas deben contemplar el caso base y el caso general. Por ejemplo, dígitos(-4) entrega 1 y dígitos(245) se calcula como 1+digitos(24) y digitos(24) es 1+dígitos(2) y dígitos(2) entrega 1 y por lo tanto el resultado final es 3 (1+1+1).

Las funciones factorial y dígitos se pueden usar en forma conjunta en un programa que lea un entero y escriba el valor de su factorial y su cantidad de dígitos:

```
from factorial import *
from digitos import *
n=int(input("n° entero positivo?"))
fact=factorial(n)
print("factorial:", fact)
print("dígitos:", digitos(fact))
```

El programa supone que las funciones están grabadas en los archivos factorial.py y digitos.py que se consideran módulos con una sola función. La instrucción from nombre import * permite usar las funciones directamente con su nombre. El programa lee un número y escribe el valor de su factorial y la cantidad de dígitos a través de un diálogo como el que se muestra en el siguiente ejemplo:

```
n° entero positivo?20
factorial: 2432902008176640000
dígitos: 19
```

Función para calcular una potencia entera

Calcular x^y se puede lograr con x**y o con math.pow(x,y). El operador de elevación a potencia y la función predefinida math.pow están diseñadas para cualquier valor y tipo del exponente. Si el exponente es un número entero mayor o igual que cero, entonces el cálculo de x^y se puede lograr con la recurrencia $x^y = x \cdot x^{y-1}$ con $x^0=1$. La función recursiva para la recurrencia es:

```
#potencia: num int -> num
#x elevado a y, y >= 0
#ej: potencia(2,3)->8, potencia(1.5,2)->2.25
def potencia(x,y):
    assert type(x)==int or type(x)==float
    assert type(y)==int and y>=0
    if y==0:
        return 1
    else:
        return x * potencia(x,y-1)
```

```
assert potencia(5,0)==1
assert potencia(2,3)==8
assert potencia(1.5,2)==2.25
```

La invocación potencia(0,0) entrega erróneamente el resultado 1, situación que también se produce con 0**0 y math.pow(0,0). Como 0^0 no está definido, entonces se debe agregar la precondición assert not (x==0 and y==0) para evitar entregar un resultado incorrecto.

Una segunda versión de la función puede optimizar el cálculo de una potencia positiva a través de la recurrencia $x^y=x^{y/2} \cdot x^{y/2}$, por ejemplo, $x^{32}=x^{16} \cdot x^{16}$. En caso que el exponente sea impar, por ejemplo, x^{33} se puede calcular como $x \cdot x^{16} \cdot x^{16}$. El operador de división entera // puede usarse para exponentes pares o impares porque 32//2 y 33//2 entregan el resultado 16. La ventaja de esta implementación es que se realizan $\log_2 y$ llamadas recursivas.

```
def potencia(x,y):
    assert type(x)==int or type(x)==float
    assert type(y)==int and y>=0
    assert not(x==0 and y<=0)
    if y==0:
        return 1
    else:
        p=potencia(x,y//2)
        if y%2==0:  #exponente par?
            return p*p
        else:
            return x*p*p
assert potencia(2,-3)==0.125
assert potencia(1.5,2)==2.25
```

Una última versión de la función evita la evaluación reiterada de las precondiciones y la condición de exponente negativo usando una función interna:

```
def potencia(x,y):
    assert type(x)==int or type(x)==float
    assert type(y)==int and y>=0
    assert not(x==0 and y==0)
    def _potencia(x,y):
        if y==0: return 1
        p=_potencia(x,y//2)
        if y%2==0:
            return p*p
        else:
            return x*p*p
    return _potencia(x,y)
```

La función recursiva interna _potencia (el nombre puede comenzar con _) no necesita receta de diseño ni precondiciones porque sólo se invoca desde la función potencia (que ya no es recursiva) después que se comprueba la validez de los parámetros en las precondiciones.

Cálculo del máximo común divisor

Para calcular el máximo común divisor entre dos enteros positivos se debe encontrar el mayor número que divida a los dos números. Conviene comenzar con un divisor igual al menor entre los dos números y decrementarlo hasta encontrar un divisor que divida a los dos números.

```
#mcd: int int -> int
#máximo común divisor entre x e y positivos
#ej: mcd(18,24)->6, mcd(4,9)->1, mcd(7,7)->7
def mcd(x,y):
    assert type(x)==int and x>0
    assert type(y)==int and y>0
    def _mcd(x,y,divisor):
        if x%divisor==0 and y%divisor==0:
            return divisor
        else:
            return _mcd(x,y,divisor-1)
    return _mcd(x,y,min(x,y))
assert mcd(18,24)==6
assert mcd(4,9)==1
assert mcd(7,7)==7
```

Nótese que mcd(18,24) se calcula con _mcd(18,24,18), _mcd(18,24,17,), _mcd(18,24,16), etc, hasta _mcd(18,24,6) que entrega finalmente el resultado 6. Por otra parte, mcd(4,9) se calcula con _mcd(4,9,4), _mcd(4,9,3), _mcd(4,9,2), _mcd(4,9,1) que entrega 1. Y mcd(7,7) se convierte en _mcd(7,7,7) que entrega inmediatamente 1.

El máximo común divisor (mcd) se puede calcular de manera más eficiente con el algoritmo de Euclídes que se define con la siguiente recurrencia:

```
          x si x=y
mcd(x,y): mcd(x-y,y) si x>y
          mcd(x,y-x) si x<y
```

Por ejemplo, mcd(18,24) se calcula con mcd(18,6), mcd(12,6) y mcd(6,6) que entrega un 6. Si x e y no tienen divisores comunes, por ejemplo, mcd(9,4), se realizan las siguientes invocaciones

recursivas: mcd(9,4), mcd(5,4), mcd(1,4), mcd(1,3), mcd(1,2) y mcd(1,1) que entrega 1. La función recursiva que implementa la recurrencia es:

```
def mcd(x,y):
    assert type(x)==int and x>0
    assert type(y)==int and y>0
    if x==y:
        return x
    elif x>y:
        return mcd(x-y,y)
    else:
        return mcd(x,y-x)
```

La función mcd contiene dos invocaciones recursivas y ambas cumplen con la condición de acercarse y converger al caso base. Por ejemplo, mcd(18,24) se calcula con mcd(18,6), mcd(12,6) y mcd(6,6) que entrega el resultado 6. Por otra parte, mcd(9,4) se calcula con mcd(5,4), mcd(1,4), mcd(1,3), mcd(1,2) y mcd(1,1) que entrega un 1.