Tema 3: Recursión

Índice

1. Recursión

Ya hemos visto algunos ejemplos de funciones recursivas. Una función es recursiva cuando se llama a si misma. Una vez que uno se acostumbra a su uso, se comprueba que la recursión es una forma mucho más natural que la iteración de expresar un gran número de funciones y procedimientos.

Recordemos el ejemplo típico de función recursiva, el factorial:

```
(define (factorial x)
   (if (= x 0)
      1
      (x * (factorial (- x 1)))))
```

La formulación matemática de la recursión es sencilla de entender, pero su implementación en un lenguaje de programación no lo es tanto. El primer lenguaje de programación que permitió el uso de expresiones recursivas fue el Lisp. En el momento de su creación existía ya el Fortran, que no permitía que una función se llamase a si misma.

En la clase de hoy veremos cómo diseñar procedimientos recursivos y cuál es el coste espacial y temporal de la recursión. Más adelante comprobaremos que no siempre una función recursiva tiene un comportamiento recursivo, sino que hay casos en los que genera un *proceso iterativo*.

Por último, en temas posteriores veremos que la recursión no sólo se utiliza para definir funciones y procedimientos sino que existen estructuras de datos cuya definición es recursiva, como las listas o los árboles.

2. Confía en la recursión

Ya sabemos cómo es un algoritmo recursivo. Hemos visto varios ejemplos utilizando números, identificadores o listas. Pero... ¿cómo diseñar un algoritmo recursivo? El consejo más importante es el que aparece en el siguiente recuadro:

Atención, aviso
Confía en la recursión

Esto es, para resolver un problema de forma recursiva debes:

1. Confiar en la recursión para resolver una versión más simple del problema
2. Obtener la solución al problema completo a partir de la solución de la versión más simple

La frase **confía en la recursión** quiere decir que cuando estés analizando el funcionamiento

de un programa recursivo y veas una llamada recursiva debes confiar en que esta llamada va a devolver el resultado que se prentende.

Vamos a utilizar estas reglas para construir algunos procedimientos recursivos.

elemento?

Por ejemplo, vamos a empezar analizando desde el punto de vista de las reglas anteriores el siguiente procedimiento, que comprueba si un carácter está en una cadena:

```
(define (elemento? x palabra)
   (cond
      ((empty? palabra) #f)
      ((equal? x (first palabra)) #t)
      (else (elemento? x (bf palabra)))))
```

El funcionamiento del procedimiento consiste en:

1. Quitamos el primer carácter de la cadena.
2. Si el carácter es la "a" devolvemos true.
3. Sino llamamos a la recursión con el resto de la cadena (un problema más sencillo, porque es una cadena más corta), confiamos en que la llamada recursiva funcione correctamente y devolvemos su resultado.

¿Cuándo para el algoritmo? Cuando el problema es "lo más simple posible" y ya no se puede simplificar más: este es el caso base. Debemos entonces devolver un valor concreto, la recursión ya ha terminado. En el caso del ejemplo se llega al caso base cuando la cadena es vacía, entonces devolvemos false.

veces

Otro ejemplo. Veamos el problema de contar el número de veces que un carácter (por ejemplo la "a") aparece en una cadena:

1. Quitamos el primer carácter de la cadena.
2. Llamamos a la recursión con el resto de la cadena (un problema más sencillo, porque es una cadena más corta). Esta llamada nos dice que en el resto de la cadena hay *n* letras "a".
3. Si el primer carácter que hemos quitado es una "a" devolvemos *n+1*, sino devolvemos *n*.
4. Caso base: cadena vacía. En ese caso devolvemos 0.

La solución en Scheme:

```
(define (veces c pal)
  (if (empty? pal)
      0
      (if (equal? (first pal) c)
         (1+ (veces c (bf pal)))
         (veces c (bf pal)))))
```

pigl

Otro problema: Pig Latin. Pig Latin es una modificación del inglés que usan los niños para hablar de forma divertida. Se trata de modificar una cadena moviendo todas las consonantes (hasta la primera vocal) de su comienzo a su final y añadiendo el sufijo "ay". Por ejemplo, en castellano:

- chaval -> avalchay
- curso -> ursocay
- problema -> oblemapray

Aquí la solución recursiva es algo más complicada. La primera idea sería hacerlo como los ejemplos anteriores:

1. Si la palabra a modificar empieza por consonante: quitar la consonante, aplicar Pig Latin al resto de cadena.
2. Confiamos en que la recursión nos devuelva el resto de cadena traducida a Pig Latin y entonces añadimos la consonante al final.
3. Caso base: si la cadena empieza por vocal devolver la cadena con el sufijo "ay" añadido.

¿Funciona bien esta solución?

Veamos un ejemplo. Apliquemos Pig Latin a la palabra "problema". Nos debería devolver "oblemapray". Sin embargo, el algoritmo anterior haría lo siguiente:

1. La palabra "problema" comienza por consonante. La quitamos y aplicamos Pig Latin a "roblema".
2. Confiamos en la recursión y devuelve la palabrama traducida: "oblemaray". Movemos la "p" al final: "oblemarayp". NO ES CORRECTO.

¿Cómo habría que plantear la recursión? Si nos fijamos en el ejemplo, nos daremos cuenta de que la "p" debería estar antes del sufijo "ay". Esto se consigue llamando a la recursión después de haber movido la consonante al final de la cadena:

1. La palabra "problema" empieza por consonante, la quitamos y la colocamos al final de la palabra. Nos queda "roblemap".
2. Confiamos en la recursión y aplicamos Pig Latin a esta palabra. Devolvería "oblemapay". !!Correcto¡¡

La solución en Scheme:

```
(define (pigl wd)
   (if (vocal? (first wd))
      (word wd 'ay)
      (pigl (word (bf wd) (first wd)))))
```

¿Dónde está aquí la regla general? ¿Estamos aplicando la recursión a un caso más sencillo? Sí, ya que la palabra "roblemap" tiene la primera vocal más a la izquierda que "problema".

nth

Vamos a ver cómo encontrar el elemento n-ésimo de una lista de forma recursiva. Este va a ser un ejemplo algo distinto a los anteriores, ya que aquí no tendremos que construir ninguna solución a partir de la llamada recursiva, sino que debemos buscar esta solución haciendo cada vez más sencillo el problema.

Vamos a llamar a la función `(nth n lista)`, donde `n` es la posición del elemento que queremos devolver (comenzando por 1) y `lista` es la lista que recorremos.

Podemos empezar por preguntarnos qué sabemos hacer con las listas:

- Devolver el primer elemento de una lista utilizando la función `car`
- Devolver el resto de la lista sin el primer elemento utilizando la función `cdr`

O sea, que si nos preguntaran por el primer elemento de la lista:

```
(nth 1 '(a b c))
```

para devolverlo habría que llamar a `car`:

```
>(car '(a b c))
a
```

¿Cómo podríamos usar la recursión para devolver un elemento que no está en primer lugar? Esto es, ¿qué hacer cuando `n` es distinto de 1? Habrá que simplificar el problema y pasarle a la llamada recursiva la lista sin el primer elemento y habiendo restado 1 a `n`:

```
(define (nth n lista)
  (if (= n 1)
    (car lista)
    (nth (- n 1) (cdr lista)))
```

¿Qué pasaría si se llama a la función con el número `n` siendo mayor que el número de elementos de la lista? Tendríamos un error, ya que la se intentaría obtener el primer elemento de una lista vacía. Para solucionarlo, basta con contemplar este caso como otra posible condición de terminación de la recursión:

```
(define (nth n lista)
  (cond
    ((null? lista) lista)
    ((= n 1) (car lista))
    (else (nth (- n 1) (cdr lista)))))
```

3. El coste espacial de la recursión

Vamos a pasar a examinar el coste espacial de la recursión. ¿Cuál es el coste espacial de una

llamada recursiva? ¿Qué información debo guardar para realizar la recursión? La respuesta es que el coste viene dado por el número de llamadas a funciones que *quedan a la espera* de que termine la recursión. Veamos algunos ejemplos.

veces

Vamos a comenzar con la función `veces` que hemos visto anteriormente. Supongamos la siguiente invocación de esta función:

```
>(veces 'a 'aaaa)
4
```

La traza de llamadas recursivas que se han generado con la llamada anterior es la siguiente:

```
(veces 'a 'aaaa)
(1+ (veces 'a 'aaa))
(1+ (1+ (veces 'a 'aa)))
(1+ (1+ (1+ (veces 'a 'a))))
(1+ (1+ (1+ 1)))
(1+ (1+ 2))
(1+ 3)
4
```

Vemos que la recursión produce una *pila de llamadas* en espera que deben almacenarse en algún lugar de la memoria del computador a la espera de ser evaluadas. En este caso el coste espacial depende linealmente de la longitud de la palabra.

fibonacci

El coste espacial no siempre tiene que ser lineal. Veamos otro ejemplo, la serie de Fibonacci.

Recordemos que la secuencia de números de Fibonacci se define con la siguiente expresión:

fib (n) = fib (n-1) + fib (n-2)
fib (1) = fib (0) = 1

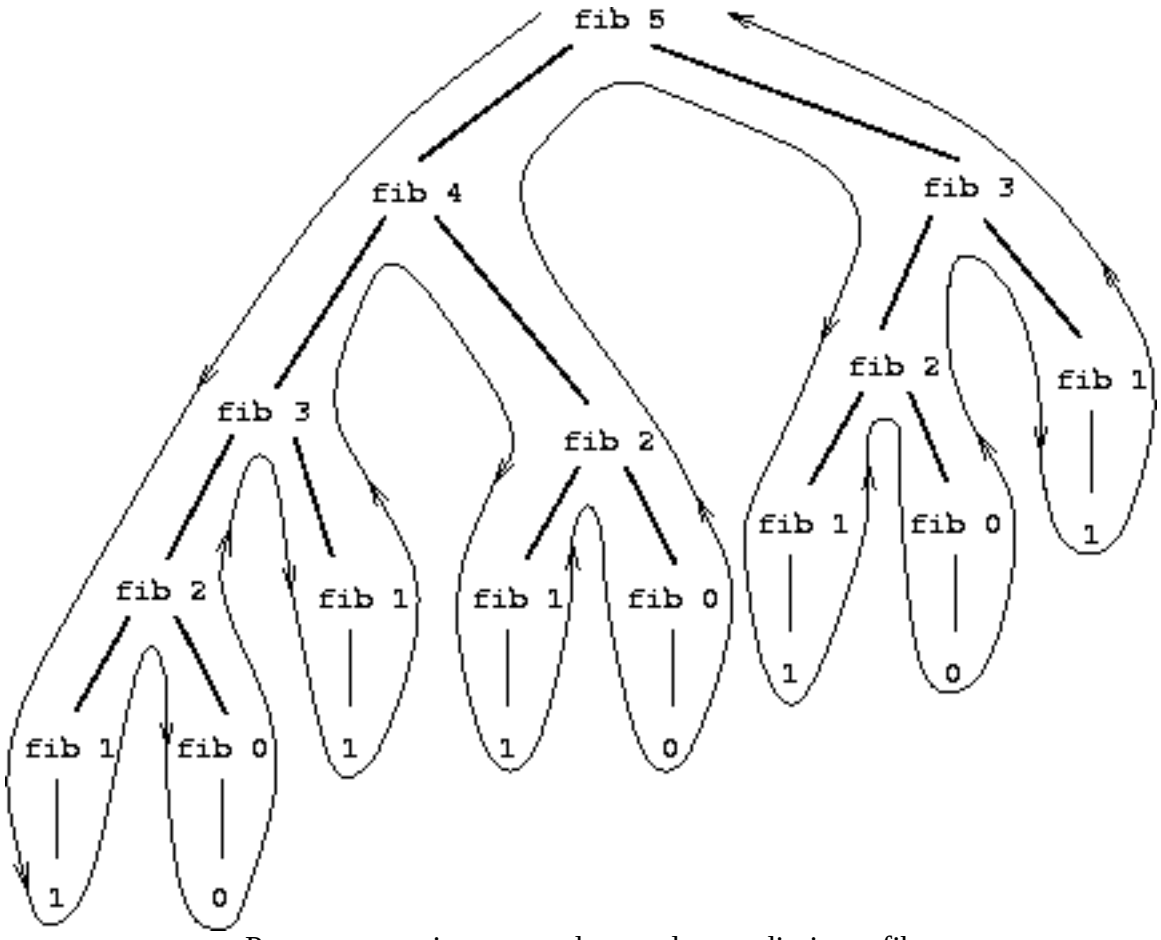
De esta forma, la secuencia de números de Fibonacci es: 1, 1, 2, 3, 5, 8, 13, 21, ...

La función recursiva que calcula el número n-ésimo de la serie es la siguiente

```
(define (fib n)
(cond ((= n 0) 0)
       ((= n 1) 1)
       (else (+ (fib (- n 1))
                (fib (- n 2))))))
```

La diferencia con las funciones recursivas vistas hasta ahora es que en el cuerpo de la función se realizan dos llamadas recursivas. Esto genera un proceso recursivo en forma de árbol,

como se comprueba en la siguiente figura, extraída del Abelson & Sussman:



Proceso recursivo generado por el procedimiento fib

Hemos visto que el coste espacial de una función recursiva depende del número de llamadas recursivas realizadas en cada llamada, que quedan en espera de devolver un resultado. ¿Existen funciones recursivas que no dejan llamadas en espera? Lo veremos mañana.

Recuerda

El coste espacial de una función recursiva depende del número de llamadas que quedan en espera en cada llamada recursiva.

4. Procesos recursivos e iterativos

En la asignatura hecho la distinción entre *programas* y los *procesos* que éstos generan. Esta distinción se hace muy importante en el caso de los programas recursivos.

Hemos visto en el apartado anterior que el proceso generado por una función recursiva genera muchas veces un coste espacial debido a las llamadas recursivas que permenacen en espera.

En este apartado comprobaremos que es posible escribir programas recursivos que generan procesos que no dejan llamadas en espera. A estos procesos los llamamos *procesos iterativos*. Veremos varios ejemplos de estos programas y analizaremos su eficiencia tanto espacial como temporal.

Vamos a introducir los conceptos de *proceso recursivo* y un *proceso iterativo*. Para diferenciar entre ambos tenemos que ocuparnos de describir los *procesos que generan* los procedimientos de Scheme. Veamos el siguiente ejemplo:

```
(define (count pal)
  (if (empty? pal)
      0
      (1+ (count (bf pal))) ))
```

Esta función cuenta el número de palabras en un frase. Es una función recursiva, porque se llama a si misma en su cuerpo. Veamos el proceso que genera la función. Para eso, veamos una traza:

```
(count 'abcdef)
(1+ (count 'bcdef))
(1+ (1+ (count 'cdef)))
(1+ (1+ (1+ (count 'def))))
(1+ (1+ (1+ (1+ (count 'ef)))))
(1+ (1+ (1+ (1+ (1+ (count 'f))))))
(1+ (1+ (1+ (1+ (1+ (1+ (count ""))))))
(1+ (1+ (1+ (1+ (1+ (1+ 0))))))
(1+ (1+ (1+ (1+ (1+ 1)))))
(1+ (1+ (1+ (1+ 2))))
(1+ (1+ (1+ 3)))
(1+ (1+ 4))
(1+ 5)
6
```

El proceso requiere un tiempo O(n), siendo n el número de caracteres de la cadena. También requiere un espacio O(n), sin contar el espacio para la frase en si misma, porque Scheme tiene que mantener el estado de n computaciones pendientes durante el procesamiento. Estas computaciones pendientes obligan a guardar el estado del proceso en el momento en que se hace la llamada recursiva. Este estado se guarda en la llamada *pila de recursión*, y se recupera cuando se obtiene el valor de la llamada recursiva.

El esquema muestra la evolución del proceso generado por la función `count`. Es un ejemplo de un proceso recursivo lineal. Cuando hemos llegado a la mitad del esquema y calculamos `(count "")`, todavía no hemos finalizado completamente el problema. Tenemos que recordar añadir 1 al resultado seís veces. Cada una de esas tareas recordadas requiere un espacio en memoria hasta que se termina el proceso.

A continuación presentamos `count-iter` un procedimiento algo más complicado que hace lo mismo sin dejar llamadas recursivas pendientes:

```
(define (count-iter pal result)
    (if (empty? pal)
        result
        (iter (bf pal) (1+ result)) ))

(define (count pal)
  (count-iter pal 0) )
```

La función `count-iter` ahora recibe un argumento adicional, denominado `result`. En él se almacena el valor parcial del cálculo de la longitud de la palabra.

En cada llamada recursiva se incrementa en 1 el resultado parcial y se elimina un carácter de la palabra. De esta forma, cuando la palabra esté vacía, el resultado habrá acumulado su longitud (siempre que inicialmente se haya pasado 0 como valor).

La función `count` se redefine para que llame a `count-iter` con los valores iniciales correctos.

Esta vez, no dejamos ninguna computación "en espera", no tenemos que recordar tareas no completas; cuando alcanzamos el caso base de la recursión tenemos la respuesta al problema completo.

```
(count 'abcdef)
(count-iter 'abcdef 0)
(count-iter 'bcdef 1)
(count-iter 'cdef 2)
(count-iter 'def 3)
(count-iter 'ef 4)
(count-iter 'f 5)
(count-iter "" 6)
6
```

Este es un ejemplo de un *proceso iterativo*. Cuando un proceso tiene esta estructura, Scheme no necesita memoria extra para recordar todas las cosas pendientes de terminar a lo largo de la computación del proceso. En otros lenguajes de programación existen bucles (`for`, `while`) para definir procesos iterativos. En un lenguaje funcional puro no existen bucles; todo lo tenemos que hacer con llamadas recursivas. Pero en este caso es un tipo de recursión

especial, que no deja llamadas en espera. Esta recursión también recibe el nombre de *tail recursión* o recursión por la cola.

Ambos algoritmos son equivalentes; incluso es posible construir la función con recursión por la cola automáticamente a partir de la función recursiva (no vamos a ver cómo se hace; lo dejamos para otras asignaturas).

4.1. Ejemplos

Otros ejemplos de procesos recursivos y procesos iterativos son los siguientes.

Factorial

Versión recursiva:

```
(define (factorial n)
   (if (= n 1)
       1
       (* n (factorial (- n 1)))))
```

Versión iterativa:

```
(define (fact-iter product counter max-count)
   (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))

(define (factorial n)
   (fact-iter 1 1 n))
```

Números de Fibonacci

Versión recursiva:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

Versión iterativa:

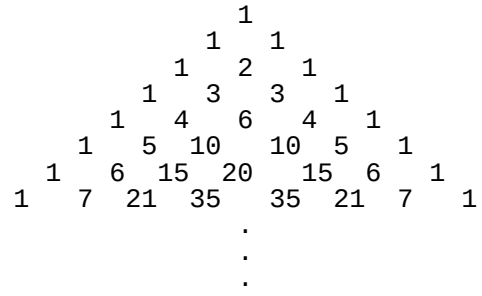
```
(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))

(define (fib n)
```

```
    (fib-iter 1 0 n))
```

Pascal

El último ejemplo de función recursiva es el triángulo de Pascal. Cada elemento del triángulo de Pascal es la suma de los dos números que hay sobre él:



Versión recursiva:

La función Scheme que calcula de forma recursiva el número de Pascal que se encuentra en una determinada fila y columna es la siguiente:

```
(define (pascal row col)
  (cond ((= col 0) 1)
        ((= col row) 1)
        (else (+ (pascal (-1+ row) (-1+ col))
                 (pascal (-1+ row) col) ))))
```

El proceso que genera esta función es un proceso que también tiene forma de árbol, ya que se hacen dos llamadas recursivas dentro del cuerpo de la función. Es similar a la función fibonacci que vimos en la clase pasada.

Versión iterativa:

Para implementar una versión iterativa iremos generando cada una de las filas del triángulo a partir de la anterior. Utilizaremos una lista para guardar la fila.

```
(define (pascal-sig-fila-central fila)
    (if (= 1 (length fila))
        '()
    (append (list (+ (car fila) (car (cdr fila))))
            (pascal-sig-fila-central (cdr fila)))))

(define (pascal-sig-fila fila)
    (append '(1)
            (pascal-sig-fila-central fila)
            '(1)))

(define (pascal-iter fila n)
```

```
      (if (= n (length fila))
          fila
          (pascal-iter (pascal-sig-fila fila) n)))

(define (pascal fila col)
   (nth col (pascal-iter '(1 1) fila)))
```

La función `pascal-sig-fila` se encarga de construir la fila siguiente del triángulo de Pascal a partir de la anterior. Por ejemplo:

```
> (pascal-sig-fila '(1 3 3 1))
(1 4 6 4 1)
```

La función `pascal-iter` va llamando de forma iterativa a `pascal-sig-fila` hasta que llegamos a la fila `n` que deseamos. La función `pascal` recoge esa fila y devuelve el elemento `col` deseado.

5. Referencias

Para saber más de los temas que hemos tratado en esta clase puedes consultar las siguientes referencias:

- Structure and Interpretation of Computer Programs (http://mitpress.mit.edu/sicp/full-text/book/book.html) , Abelson y Sussman, MIT Press 1996 (cap. 1.2.1 y 1.2.2). Disponible biblioteca politécnica (acceso al catálogo (http://gaudi.ua.es/uhtbin/boletin/285815))
- Tail recursion (http://en.wikipedia.org/wiki/Tail_recursion) (Wikipedia)