005. Hallar el factorial de un número natural

Escriba una función factorial que toma un número natural como argumento y devuelve el factorial de dicho número.

El factorial de un número natural se define de forma recurrente como:

$$n! = \begin{cases} 1 & \text{si } (n = 0) \\ n \times (n-1)! & \text{si } (n > 0) \end{cases}$$

También se lo puede calcular como: $n! = n \times (n-1) \times (n-2) \times ... \times 3 \times 2 \times 1$

Implementación iterativa en C

```
#include <stdio.h>
unsigned long long int iterfac(unsigned int n)
{
        unsigned long long int res = 1;
        for(unsigned int i=n; i > 0; --i)
                res *= i;
        return res;
}

int main(void)
{
        unsigned int x = 65;
        printf("factorial(%u) = %llu\n", x, iterfac(x));
        return 0;
}
```

El máximo número que podemos calcular su factorial es:
unsigned long -> factorial(65) = 9223372036854775808
unsigned long long -> factorial(65) = 9223372036854775808

Utilizamos **res** como acumulador de los productos sucesivos, desde **n** hasta 1; inicializamos **res** con 1 que es el elemento neutro del producto. Como se trata de lenguaje C, empleamos **unsigned long long int**, para disponer la mayor cantidad posible de bits para el resultado (aunque el máximo valor es el mismo que con **unsigned long int** en GNU/Linux).

Implementación recursiva en C

```
#include <stdio.h>
unsigned long long int recurfac(unsigned int n)
{
        if (n == 0)
                return 1;
        else
                return n * recurfac(n-1);
}

int main(void)
{
        unsigned int x = 65;
        printf("factorial(%u) = %llu\n", x, recurfac(x));
        return 0;
}
```

En este caso por cada llamada recursiva, se genera un nuevo stack frame para almacenar el parámetro, la dirección de retorno, etc. Como en C solamente podemos llegar hasta 65 con el factorial, usando los números definidos en el lenguaje, eso no es un problema, pues 65 frames no es nada en términos de memoria actual.

Implementación recursiva en Scheme

El código Scheme recursivo sigue de cerca el código recursivo de C y la definición matemática, que copiamos nuevamente a continuación para comparar con el Scheme correspondiente:

$$n! = \begin{cases} 1 & \text{si } (n = 0) \\ n \times (n-1)! & \text{si } (n > 0) \end{cases}$$

```
(define factorial (lambda (n)

 (if (= n 0)   1

     (* n  (factorial (- n 1)))
 )))
```

Esta implementación recursiva tiene los mismos problemas que la de C, pero agravados porque Scheme implementa precisión infinita (o casi) para los números:

`(display (factorial 10000))` el resultado tiene 35660 dígitos y lo calcula en medio segundo; y no es el más grande que se puede calcular. ¿Problemas agravados dije? No parece un problema para Scheme.

Ya tenemos una conclusión interesante: nuestro programa en Scheme es mucho más capaz que nuestro programa en C, a pesar que tienen la misma estructura y complejidad. Es decir, que usando la misma inteligencia de programador, tengo mejores prestaciones si escribo en Scheme que en C, en el caso de problemas que usan profusamente estructuras y programas recursivos.

¿Podemos evitar la acumulación de stack frames en Scheme?

Implementación iterativa en Scheme

Debemos hacer una distinción importante, entre programas y procesos. Un programa es lo que escribimos en C o Scheme. Un proceso es lo que se ejecuta en la computadora al seguir un programa dado.

En C los programas iterativos generan procesos iterativos; los programas recursivos generan procesos iterativos.

En Scheme podemos generar procesos recursivos a partir de programas recursivos, pero también, y esto es lo interesante, podemos generar procesos iterativos a partir de programas recursivos. ¿Cómo es esto? Cuando en un proceso recursivo, la llamada recursiva se da como última instrucción en el programa, y el programa retorna el valor obtenido de la recursión, sin utilizarlo en computaciones subsiguientes, la llamada recursiva actúa como una reasignación de valores a los parámetros y un salto (GOTO) al comienzo de la rutina recursiva. En Scheme el compilador detecta que éste es el caso, y reemplaza la manipulación del stack para crear una llamada, por un salto incondicional al principio del programa.

Vemos esto para el caso del programa de factorial.

```
(define factorial
  (lambda (n acum)
    (if (= n 0)
        acum
        (factorial (- n 1) (* n acum)))))

(display (factorial 10000 1))
```

Note que ahora el **`if`** tiene dos posibles valores de salida: el valor de **`acum`** cuando **`n`** es cero, o el valor de la llamada (salto) recursiva. No hay productos ni ninguna computación adicional luego de la llamada recursiva.

El compilador de Scheme encuentra estas situaciones, y genera un programa en el cual la recursión se elimina por optimización de llamada de cola (*tail-call optimization*). Es una estrategia muy poderosa, que nos permite aprovechar la expresividad de la recursión, y disponer de procesos que corren con las ventajas de memoria de los programas iterativos.