

# Caso T10

1) Una base de datos SQL utiliza la siguiente sintaxis para la instrucción UPDATE:

```
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]  
    SET column_name = { expression | DEFAULT } [, ...]  
    [ FROM from_list ]  
    [ WHERE condition | WHERE CURRENT OF cursor_name ] <END>
```

En esta especificación sintáctica, los corchetes ("[...]") indican que un elemento es opcional, las llaves se usan para delimitar elementos obligatorios compuestos, el carácter "|" indica elementos alternativos y los puntos suspensivos ("...") indican que el elemento anterior puede repetirse.

Estamos trabajando en el diseño de un analizador sintáctico para el compilador de SQL de la base de datos. La primera etapa de la compilación, el análisis léxico, ya está finalizada, con lo cual el analizador sintáctico recibe los tokens que ha reconocido el analizador léxico. Los tokens se etiquetarán con las palabras clave de la sentencia SQL (UPDATE, SET, WHERE, ...), los símbolos, operadores o separadores ("\*", ",", "..."), los nombres de los distintos tipos de expresiones reconocidas (table\_name, alias, column\_name, ...) o la etiqueta especial <END>, que indica el final de la sentencia SQL.

Se requiere dibujar el diagrama de transición de estados para el analizador sintáctico con la notación sugerida por METRICA o equivalente: es decir, un grafo dirigido donde los nodos son los estados (hay que distinguir el estado inicial y el estado final; el resto de los estados no necesitan diferenciarse entre ellos, aunque se les puede dar un nombre si se desea) y las aristas son las transiciones entre estados. En cada arista se deberá anotar la etiqueta del token descrita anteriormente que causa la transición del estado origen al destino.

Debe tenerse en cuenta lo siguiente:

- Cada transición consume un token proporcionado por el analizador léxico, por lo que cada arista señala la transición al estado siguiente cuando se encuentra el token con la etiqueta indicada. Así, en el nuevo estado al que se ha transicionado ya no está disponible el token anterior, sino el que le sigue en la sentencia SQL que el analizador léxico ha descompuesto en una secuencia de tokens.
- El grafo debe ser determinista, es decir, de un nodo no pueden salir dos aristas diferentes con la misma etiqueta. Tampoco pueden existir aristas sin etiqueta, es decir, que no consuman un token.
- No se debe representar una transición que no conduzca a un estado correcto dentro del análisis sintáctico de la sentencia SQL y deben representarse todas las transiciones que conduzcan a un estado correcto. Eso quiere decir que, si en un estado determinado se recibe un token que no está contemplado en sus aristas de salida, se interpreta que el compilador debería informar de un error de sintaxis.

Valor de la pregunta: 50% de la nota del caso

2) Los algoritmos recursivos son aquellos que expresan la solución de un problema en términos de una llamada a sí mismos. Los lenguajes que permiten las llamadas recursivas habitualmente las implementan con la ayuda de una pila, que va guardando el estado de la función que llama mientras se ejecuta la función llamada, de manera que, al devolver un valor una función, la llamante lo recibe y ejecuta los cálculos necesarios para poder devolver de manera análoga un valor a la función que a su vez la ha llamado.

La implementación de las llamadas recursivas usando una pila, como en las llamadas no recursivas, presenta el problema de que, cuando el nivel de llamadas es muy profundo, se puede agotar el espacio de la pila y hacer que el programa falle. Para resolver este problema, en algunos casos se pueden aplicar las siguientes técnicas:

- Conversión del algoritmo a iterativo: convertir las llamadas recursivas en un bucle, preferentemente sin consumir memoria adicional por cada iteración. Esto último no siempre es posible, puesto que hay algoritmos que requieren guardar el estado de las diferentes operaciones en curso y esto, en términos de uso de recursos, es funcionalmente equivalente a la solución de la pila que ya implementa el propio lenguaje.
- Optimización de llamadas de cola (TCO, *tail call optimization*): algunos compiladores implementan TCO, que es un tipo de optimización que se puede llevar a cabo de manera automática solamente en aquellas funciones que, después de hacer una llamada recursiva, no realizan ninguna operación adicional con el valor recibido más que devolverlo. Puesto que, en cada nivel de profundidad, todas las operaciones se realizan antes de hacer la llamada recursiva, es posible ir descartando todos los valores intermedios de la pila y al final hacer que la última ejecución de la función (cuando llega a la condición límite) devuelva el valor directamente al código que ejecutó la primera llamada, saltándose la cadena de todas las devoluciones de resultado intermedias. Puede ser necesario definir funciones auxiliares, ya que, al no devolver el control a la función llamante, la función recursiva necesita recibir todos los parámetros necesarios para efectuar su trabajo.

El caso típico de algoritmo recursivo es el cálculo del factorial, que se define de la siguiente manera para un entero positivo  $n$ :

$\text{factorial}(n)=1$  si  $n=1$ ,  
 $\text{factorial}(n)=n*\text{factorial}(n-1)$  en el resto de casos

Así pues, tenemos el siguiente fragmento de código Java que implementa la función factorial:

```
public class MiFactorial {  
    public static int factorial(int i) {  
        if (i==1) return 1;  
        return i*factorial(i-1);  
    }  
  
    public static int tcofactorial(int i) {  
        // ***** IMPLEMENTACIÓN RECURSIVA TCO*****  
    }  
  
    public static int iterfactorial(int i) {  
        // ***** IMPLEMENTACIÓN ITERATIVA *****  
    }  
  
    // ***** MÉTODOS AUXILIARES *****  
  
    public static void main(String[] args) {  
        int param=Integer.parseInt(args[0]);  
  
        System.out.println("Calculando el factorial de "+args[0]);  
        System.out.print("Resultado (recursivo)=");  
        System.out.println(factorial(param));  
        System.out.print("Resultado (recursivo TCO)=");  
        System.out.println(tcofactorial(param));  
        System.out.print("Resultado (iterativo)=");  
        System.out.println(iterfactorial(param));  
    }  
}
```

En los lugares reservados para ello, se debe completar el código, implementando el algoritmo recursivo definido en el método *factorial* en su versión recursiva con TCO (Valor: 25%) y su versión iterativa (Valor: 25%), con las funciones (métodos) auxiliares que sean necesarios.

\* **Nota 1:** el programa se ha implementado utilizando enteros (**int**) para simplificar la notación y poder utilizar la representación numérica y los operadores de los enteros. En realidad, se debería implementar utilizando la clase **BigInteger** (enteros de precisión arbitraria), puesto que un entero normal se desborda después de unas pocas iteraciones y sus resultados ya no tienen sentido mucho antes de llegar a agotar el espacio de la pila.

\* **Nota 2:** por simplicidad, no se implementa ningún control de errores ni gestión de excepciones. Se supone que el parámetro recibido siempre será un entero mayor o igual que 1.

\* **Nota 3:** la versión actual de la máquina virtual Java no soporta TCO, por lo que el comportamiento real del método *tcofactorial* será semejante al del método *factorial*.

Valor de la pregunta: 50% de la nota del caso