Cas S10

1) Una base de dades SQL utilitza la següent sintaxi per a la instrucció UPDATE:

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

En aquesta especificació sintàctica, els claudàtors ("[...]") indiquen que un element és opcional, les claus s'utilitzen per delimitar elements obligatoris compostos, el caràcter "|" indica elements alternatius i els punts suspensius ("...") indiquen que l'element anterior pot repetir-se.

Estam treballant en el disseny d'un analitzador sintàctic per al compilador d'SQL de la base de dades. La primera etapa de la compilació, l'anàlisi lèxica, ja està finalitzada, amb la qual cosa l'analitzador sintàctic rep els tokens que ha reconegut l'analitzador lèxic. Els tokens s'etiquetaran amb les paraules clau de la sentència SQL (UPDATE, SET, WHERE ...), els símbols, operadors o separadors ("*", ","), els noms dels diferents tipus d'expressions reconegudes (table_name, alias, column_name...) o l'etiqueta especial <END>, que indica el final de la sentència SQL.

Es requereix dibuixar el diagrama de transició d'estats per a l'analitzador sintàctic amb la notació suggerida per METRICA o equivalent: és a dir, un graf dirigit on els nodes són els estats (cal distingir l'estat inicial i els estats finals; la resta dels estats no necessiten diferenciar-se entre ells, encara que se'ls pot donar un nom si es desitja) i les arestes són les transicions entre estats. A cada aresta s'haurà d'anotar l'etiqueta del token descrita anteriorment que causa la transició de l'estat origen al de destinació.

S'ha de tenir en compte el següent:

- Cada transició consumeix un token proporcionat per l'analitzador lèxic, per la qual cosa cada aresta assenyala la transició a l'estat següent quan es troba el token amb l'etiqueta indicada. Així, en el nou estat al que s'ha transicionat ja no està disponible el token anterior, sinó el que li segueix en la sentència SQL que l'analitzador lèxic ha descompost en una seqüència de tokens.

- El graf ha de ser determinista, és a dir, d'un node no poden sortir dues arestes diferents amb la mateixa etiqueta. Tampoc no poden existir arestes sense etiqueta, és a dir, que no consumeixin un token.

- No s'ha de representar una transició que no condueixi a un estat correcte dins de l'anàlisi sintàctica de la sentència SQL i s'han de representar totes les transicions que condueixin a un estat correcte. Això vol dir que, si en un estat determinat es rep un token que no està previst a les seves arestes de sortida, s'interpreta que el compilador hauria d'informar d'un error de sintaxi.

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

2) Els algorismes recursius són aquells que expressen la solució d'un problema en termes d'una cridada a si mateixos. Els llenguatges que permeten les cridades recursives habitualment les implementen amb l'ajuda d'una pila, que va guardant l'estat de la funció que crida mentre s'executa la funció cridada, de manera que, en tornar un valor una funció, la que l'ha cridat el rep i executa els càlculs necessaris per poder tornar de manera anàloga un valor a la funció que a la seva vegada l'ha cridat.

La implementació de les cridades recursives utilitzant una pila, com en les cridades no recursives, presenta el problema que, quan el nivell de cridades és molt profund, es pot esgotar l'espai de la pila i fer que el programa falli. Per resoldre aquest problema, en alguns casos es poden aplicar les següents tècniques:

- Conversió de l'algorisme a iteratiu: convertir les cridades recursives en un bucle, preferentment sense consumir memòria addicional per cada iteració. Això darrer no sempre és possible, ja que hi ha algorismes que requereixen guardar l'estat de les diferents operacions en curs i això, en termes d'ús de recursos, és funcionalment equivalent a la solució de la pila que ja implementa el propi llenguatge.

- Optimització de cridades de cua (TCO, tail call optimization): alguns compiladors implementen TCO, que és un tipus d'optimització que es pot dur a terme de manera automàtica només en aquelles funcions que, després de fer una cridada recursiva, no realitzen cap operació addicional amb el valor rebut més que tornar-lo. Atès que, en cada nivell de profunditat, totes les operacions es realitzen abans de fer la cridada recursiva, és possible anar descartant tots els valors intermedis de la pila i al final fer que la darrera execució de la funció (quan arriba a la condició límit) torni el valor directament al codi que va executar la primera cridada, saltant-se la cadena de totes les devolucions de resultat intermèdies. Pot ser necessari definir funcions auxiliars, ja que, en no retornar el control a la funció que crida, la funció recursiva necessita rebre tots els paràmetres necessaris per efectuar la seva feina.

El cas típic d'algorisme recursiu és el càlcul del factorial, que es defineix de la següent manera per a un enter positiu n:

factorial(n)=1 si n=1,
factorial(n)=n*factorial(n-1) en la resta de casos

Així doncs, tenim el següent fragment de codi Java que implementa la funció 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));
    }
}
```

Als llocs reservats per a això, s'ha de completar el codi, implementant l'algorisme recursiu definit en el mètode *factorial* en la seva versió recursiva amb TCO *(Valor: 25%)* i la seva versió iterativa *(Valor: 25%)*, amb les funcions (mètodes) auxiliars que siguin necessaris.

* **Nota 1**: el programa s'ha implementat utilitzat enters (**int**) per simplificar la notació i poder utilitzar la representació numèrica i els operadors dels enters. En realitat, s'hauria d'implementar utilitzant la classe **BigInteger** (enters de precisió arbitrària), ja que un enter normal es desborda després d'unes quantes iteracions i els seus resultats ja no tenen sentit molt abans d'arribar a esgotar l'espai de la pila.

* **Nota 2**: per simplicitat, no s'implementa cap control d'errors ni gestió d'excepcions. Se suposa que el paràmetre rebut sempre serà un enter major o igual que 1.

* **Nota 3**: la versió actual de la màquina virtual Java no suporta TCO, per la qual cosa el comportament real del mètode *tcofactorial* serà semblant al del mètode *factorial*.

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