PROCESADORES DEL LENGUAJES Septiembre 2019

NOMBRE: _____ GRUPO: _____

Lenguaje de Programación B (Análisis léxico-sintáctico)

B es un lenguaje diseñado para programar relaciones binarias con argumentos que pueden ser constantes o variables.

B tiene las siguientes *restricciones sintácticas*: (a) el programa está constituido por un conjunto de declaraciones. (b) La declaración puede ser: (b.1) primitiva, por ejemplo, `_es_padre(jose,X)` o (b.2) no primitiva, por ejemplo, `_es_tio(jose,X) = _es_hermano(jose,Y), _es_madre(Y,X)`

B tiene las siguientes *restricciones léxicas*: (a) las constantes son cadenas de caracteres en minúsculas y las variables son cadenas de caracteres en mayúsculas, (b) la coma representa la conjunción lógica, (c) el punto y coma se usa para terminar declaraciones y (d) los nombres de las relaciones deben comenzar con el subrayado.

A continuación, se muestra un programa B de ejemplo:

```
_es_padre(jose,X);
_es_hermano(jose,Y);
_es_tio(jose,X) = _es_hermano(jose,Y),_es_padre(Y,X);
_es_padre(juan,ines);
_es_tio(jose,X) = _es_hermano(jose,Y),_es_madre(Y,X);
```

SE PIDE: **(2,5 puntos)**

[DISEÑO] Gramática independiente de tecnología que describa la sintaxis de B.

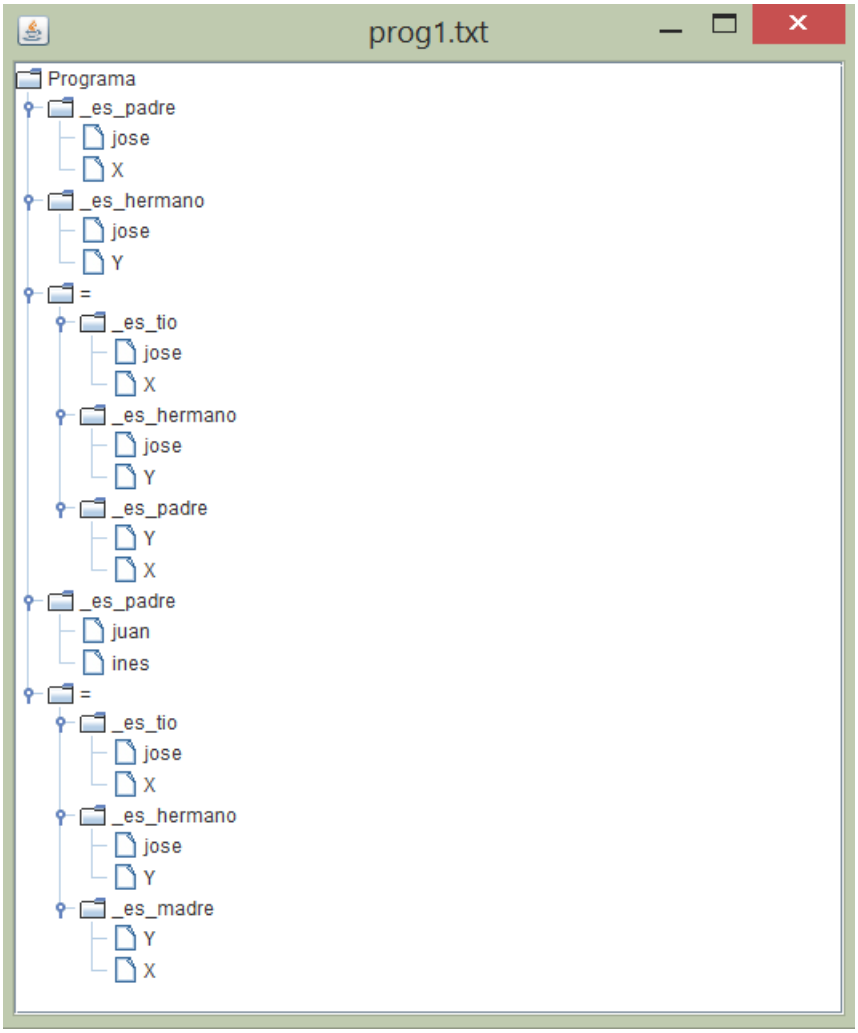
```
programa : (declaracion PyC)*    ;

declaracion : declaracion_no_primitiva
            | declaracion_primitiva
            ;

declaracion_primitiva : atomo  ;

atomo : R PARENTESISABIERTO term COMA term PARENTESISCERRADO  ;

declaracion_no_primitiva : atomo IGUAL secuencia_atomos  ;

secuencia_atomos : atomo COMA secuencia_atomos
                 | atomo
                 ;

term : VAR
     | CONST
     ;
```

[IMPLEMENTACIÓN] Lexer y Parser Antlr para B.

El parser Antlr debe generar árbol de sintaxis abstracta (AST). Los símbolos de puntuación (paréntesis, comas y puntos y comas) deben ser eliminados del AST. El resto de la información se considera relevante y debe ser incluida en el AST. Dibuje el AST generado por su parser para el programa de ejemplo propuesto.

```
class Analex extends Lexer;
options{
      importVocab=Anasint;
}
protected NL : "\r\n" {newline();};
protected LETRAMIN : 'a'..'z';
protected LETRAMAY : 'A'..'Z';

BTF:(' '|'\t'| NL) {$setType(Token.SKIP);};

R: '_'(LETRAMIN|LETRAMAY|'_')+ ;

VAR: (LETRAMAY)+ ;

CONST: (LETRAMIN)+ ;

PARENTESISABIERTO : '(';

PARENTESISCERRADO :')';

COMA:',';

PyC:';';

IGUAL:'=';
```

```
class Anasint extends Parser;
options{
      buildAST=true;
}
tokens{
      PROGRAMA;
}
programa : (declaracion PyC!)* EOF!
   { #programa = #(#[PROGRAMA,"Programa"], ##);}
         ;

declaracion : (atomo IGUAL)=>declaracion_no_primitiva
            | declaracion_primitiva
            ;

declaracion_primitiva : atomo  ;

atomo : R^ PARENTESISABIERTO! term COMA! term PARENTESISCERRADO!  ;

declaracion_no_primitiva : atomo IGUAL^ secuencia_atomos  ;

secuencia_atomos : (atomo COMA)=>atomo COMA! secuencia_atomos
                 | atomo
                 ;

term : VAR
     | CONST
     ;
```


prog1.txt
Programa
_es_padre
jose
X
_es_hermano
jose
Y
=
_es_tio
jose
X
_es_hermano
jose
Y
_es_padre
Y
X
_es_padre
juan
ines
=
_es_tio
jose
X
_es_hermano
jose
Y
_es_madre
Y
X

PROCESADORES DEL LENGUAJES Septiembre 2019

NOMBRE: _____ GRUPO: _____

Lenguaje de Programación B (Análisis semántico)

B es un lenguaje diseñado para programar relaciones binarias con argumentos que pueden ser constantes o variables. La sintaxis de B está formalizada en la siguiente gramática abstracta:

```
programa : #(PROGRAMA (declaracion)*)

declaracion : declaracion_no_primitiva  | declaracion_primitiva  ;

declaracion_primitiva : atomo ;

atomo : #(R term term) ;

declaracion_no_primitiva : #(IGUAL atomo secuencia_atomos)  ;

secuencia_atomos : (atomo)+ ;

term : VAR  | CONST    ;
```

Un programa B de ejemplo es el siguiente:

```
_es_padre(jose,X);                                  //declaración primitiva
_es_hermano(jose,Y);                                //declaración primitiva
_es_tio(jose,X) = es_hermano(jose,Y),es_padre(Y,X); //declaración no primitiva
_es_padre(juan,ines);                               //declaración primitiva
```

Se denomina *definición* de una relación `R` al conjunto de todas las declaraciones primitivas de `R` y las no primitivas en las que `R` ocurre en la parte izquierda del lexema `IGUAL (=)`. Una relación es primitiva si todas las declaraciones de su definición son primitivas. Una relación es no primitiva si alguna de las declaraciones de su definición es no primitiva. Por ejemplo, la relación `_es_padre` en el programa de ejemplo es primitiva y la relación `_es_tio` es no primitiva.

Un `programa` se dice que es *seguro* si y sólo si todas sus relaciones son seguras. Una relación `R` en un programa se dice que es *segura* si y sólo si: **(a)** `R` es primitiva o **(b)** `R` es no primitiva y toda declaración no primitiva de `R` cumple la siguiente restricción: las variables que sólo ocurren en la parte derecha de la declaración, ocurren en algún átomo de relación primitiva.
Por ejemplo, el programa mostrado anteriormente es seguro porque su única relación no primitiva (`_es_tio`) es segura: su única declaración cumple **(b)** dado que la variable `Y` ocurre en átomos de relaciones primitivas (`_es_hermano`, `_es_padre`) .
Sin embargo, el siguiente programa

```
_es_tio(jose,X) = es_hermano(jose,Z),es_padre(Y,X);
_es_padre(jose,X);
_es_padre(juan,ines);
_es_hermano(jose,Y);
_es_hermano(X,Y) = es_hermano(Y,X);
```

no es seguro porque la relación no primitiva `_es_tio` no es segura: su única declaración no cumple **(b)** porque incluye una variable `Z` que sólo ocurre en átomo de la relación no primitiva `_es_hermano`.

SE PIDE: **(2,5 puntos)**

[DISEÑO] Diseño abstracto del analizador semántico. Use la gramática abstracta dada. Aclare qué información necesita almacenar y en qué localización de la gramática necesita almacenarla. ¿El cálculo de programa seguro.se puede realizar al vuelo? Justifique su respuesta.

```
{     // Se necesita almacenar las relaciones no primitivas definidas en el programa:
         no_prim_rels
```

```
      no_prim_rels = { _es_hermano, _es_tio}

    // Se necesita identificar cada declaración en el programa: id_decl

        decl 1, decl 2, ...

    //Se necesita almacenar por cada declaración no primitiva y por cada átomo
    //de su cuerpo, la relación y sus variables: decls_no_prim_cuerpo.
    //La relación se necesita para saber si es primitiva o no primitiva.

        decl 1      _es_hermano  Z
                    _es_padre    X,Y

        decl 5      _es_hermano  X,Y

    //Se necesita almacenar por cada declaración no primitiva, las variables
    //del átomo de su cabeza: decls_no_prim_cabeza


        decl 1      X
        decl 5      X,Y


    //Operaciones: (No se pueden realizar al vuelo)
    // Hasta concluir el procesamiento no podemos conocer cuáles son las relaciones
    // no primitivas.

    vars_atomos_prim
    // Una vez conocidas las relaciones no primitivas, hay que calcular
    // para cada declaración en decls_no_prim_cuerpo, el conjunto de variables
    // de sus átomos con definición primitiva

       { Y, X } = vars_atomos_prim(decl 1)

    chequeo_decl_segura
    // Para cada declaración no primitiva,
    //  para cada átomo con relación con definición no primitiva en el
    //    cuerpo(decls_no_prim_cuerpo),
    //    consultar si alguna de sus
    //    variables no se encuentra entre las variables de la cabeza de dicha
    //       declaración (decls_no_prim_cabeza) o entre las variables
    //       de los átomos con definición primitiva (vars_atomos_prim)

       no = chequeo_decl_segura (decl 1) porque

                        decl 1      X                (en la cabeza)
                        decl 1      _es_hermano  Z   (en el cuerpo)
                                    _es_padre    X,Y

                        { Y, X } = vars_atomos_prim(decl 1)

                        Y por tanto, Z ∉ { Y, X }
    chequeo_prog_seguro
    // Dado un programa,
    //  para cada declaración no primitiva en el
    //    cuerpo(decls_no_prim_cuerpo),
    //    comprobar si es segura

}


programa : #(PROGRAMA ({id_decl = id_decl + 1} declaracion)*)
             { resultado = chequeo_prog_seguro(); };

declaracion : declaracion_no_primitiva
            | declaracion_primitiva
            ;
```

```
declaracion_primitiva : r=atomo {rels = rels ∪ {r}} ;

atomo[localizacion] returns [R] :
   #(R v1=term v2=term) {si (localización no es cabeza) entonces
                           almacenar en decls_no_prim_cuerpo id_decl,R,v1,v2
                        sino
                           almacenar en decls_no_prim_cabeza id_decl,v1,v2 } ;

declaracion_no_primitiva :
   #(IGUAL r=atomo[cabeza]  secuencia_atomos) {rels = rels ∪ {r}} ;

secuencia_atomos : (atomo[cuerpo])+ ;

term returns [VAR]:  VAR
     | CONST
     ;
```

[IMPLEMENTACIÓN] Implementación del diseño propuesto mediante Treeparser Antlr. Use la gramática abstracta dada.

```
header{
      import java.util.*;
}
class Anasem extends TreeParser;
options{
      importVocab=Anasint;
}
{
      Integer id_decl=0;

      Set<String>no_prim_rels=new HashSet<String>();

      Map<Integer,Map<String,Set<String>>>decls_no_prim_cuerpo =
         new HashMap<Integer,Map<String,Set<String>>>();

      Map<Integer,Set<String>>decls_no_prim_cabeza =
         new HashMap<Integer,Set<String>>();

      void almacenar_decls_no_prim_cuerpo(Integer n, String r, String v1, String v2){
            Map<String,Set<String>>aux=new HashMap<String,Set<String>>();
            Set<String>aux2=new HashSet<String>();
            if (decls_no_prim_cuerpo.keySet().contains(n)){
               aux=decls_no_prim_cuerpo.get(n);
               if (aux.keySet().contains(r))
                  aux2=aux.get(r);
            }
            if (v1!=null) aux2.add(v1);
            if (v2!=null) aux2.add(v2);
            aux.put(r,aux2);
            decls_no_prim_cuerpo.put(n,aux);
      }

      void almacenar_decls_no_prim_cabeza(Integer n, String v1, String v2){
            Set<String>aux=new HashSet<String>();
            if (v1!=null) aux.add(v1);
            if (v2!=null) aux.add(v2);
        decls_no_prim_cabeza.put(n,aux);
      }


      Set<String> vars_atomos_prim(Integer decl){
       Set<String>resultado=new HashSet<String>();
       Map<String,Set<String>>aux=decls_no_prim_cuerpo.get(decl);
       for(String r:aux.keySet())
          if (!no_prim_rels.contains(r))
             resultado.addAll(aux.get(r));
```

```
        return resultado;
      }

      boolean chequeo_decl_segura(Integer decl){
            Map<String,Set<String>>aux=decls_no_prim_cuerpo.get(decl);
            Set<String>s1=decls_no_prim_cabeza.get(decl);
            System.out.println("Declaración: "+decl);
            Set<String>s2=vars_atomos_prim(decl);
            System.out.println("Cuerpo: "+aux);
            System.out.println("Vars a la izq de la declaración: "+s1);
            System.out.println("Vars a la dcha de la declaración en átomos con relación
primitiva: "+s2);
            for(String r:aux.keySet())
                  if (no_prim_rels.contains(r)){
                        Set<String>s0=aux.get(r);
                        s0.removeAll(s1);
                        s0.removeAll(s2);
                        if (!s0.isEmpty()) {System.out.println("Variables
problemáticas: "+s0); return false;}
                  }
            return true;
      }

      boolean chequeo_prog_seguro(){
            boolean resultado=true;
            boolean aux=true;
            for(Integer d:decls_no_prim_cuerpo.keySet()){
               aux=chequeo_decl_segura(d);
               if (!aux)
                  System.out.println("   >>>> Declación "+d+" no es segura");
               resultado = resultado && aux;
               aux=true;
            }
            return resultado;
      }


}

programa : #(PROGRAMA ({id_decl++;} declaracion)*)
               { System.out.println(no_prim_rels);
                   System.out.println(decls_no_prim_cabeza);
                 System.out.println(decls_no_prim_cuerpo);
                 System.out.println("¿Programa seguro?: "+chequeo_prog_seguro());};

declaracion : declaracion_no_primitiva
            | declaracion_primitiva
            ;

declaracion_primitiva {String r;}: r=atomo[null]  ;

atomo[Boolean localizacion_cuerpo] returns[String r=null;] {String v1,v2;}:
   #(a:R v1=term v2=term) { r=new String(a.getText());
                            if (localizacion_cuerpo!=null){
                               if (localizacion_cuerpo)
almacenar_decls_no_prim_cuerpo(id_decl,r,v1,v2);
                               else almacenar_decls_no_prim_cabeza(id_decl,v1,v2);
                            }} ;

declaracion_no_primitiva {String r;}:
   #(IGUAL r=atomo[false] secuencia_atomos) {no_prim_rels.add(r);} ;

secuencia_atomos {String r;}: (r=atomo[true])+ ;

term returns [String s=null;]:  a:VAR {s=new String(a.getText());}
     | CONST
     ;
```

PROCESADORES DEL LENGUAJES Septiembre 2019

NOMBRE: _____ GRUPO: _____

Lenguaje de Programación S (Interpretación)

S es un lenguaje de programación que permite programar árboles de grado n. La declaración de variable en S crea un árbol sin nodos. Los nodos del árbol contienen información de tipo entera. Los variables enteras usadas en el programa no se declaran y toman valor 0 por defecto. El valor de éstas puede alterarse mediante el uso de asignaciones.
La semántica de las operaciones suma, resta y producto de enteros son las convencionales. No se consideran otras operaciones.
Se contemplan las siguientes operaciones para el tipo árbol: (1) `root`, (2) `child`, (3) `sibling` (4) `root_node` y (5) `sequence`. La operación `root` tiene dos argumentos, un árbol y un entero, y permite establecer como nodo raíz del árbol dicho entero. La operación `child` tiene dos argumentos, un nodo y un árbol, y permite establecer el árbol como hijo único del nodo. La operación `sibling` tiene como argumentos dos árboles y permite establecer el segundo como hermano del primero. La función `root_node` tiene como argumento un árbol con al menos un nodo y devuelve el nodo raíz de dicho árbol. La operación `sequence` tiene como argumentos un árbol y permite mostrar por pantalla la secuenciación de sus nodos mediante un recorrido en profundidad. A continuación, se muestra un programa S de ejemplo con anotaciones para aclarar su interpretación:

```
VARIABLES a,b,c,d;            //declaracion de 4 arboles: a, b, c y d.
INSTRUCCIONES
   x=2;                       //x vale 2
   root(a,x);                 //raiz de a vale 2
   x=3;                       //x vale 3
   root(b,1);                 //raiz de b vale 1
   root(c,x+2);               //raiz de c vale 5
   child(root_node(a),b);     //el hijo de la raiz de a es b
   sibling(b,c);              //b tiene un hermano c
   child(root_node(b),d);     //el hijo de la raiz de b es d
   sequence(a)                //muestra por pantalla: 2,1,5
   root(d,0);                 //raiz de d vale 0
   sequence(a)                //muestra por pantalla: 2,1,0,5
```

SE PIDE: **(2,5 puntos)**

[DISEÑO] Diseño del intérprete usando la gramática abstracta dada en el Anexo. Los programas S se suponen sin errores.

Arbol: Las operaciones consideradas en el diseño son las siguientes:

```
// Establece info como raíz del árbol a
   root(a,info)

//Establece h como árbol hijo del árbol a
  child(a, h)

//Establece h como árbol hermano de árbol a
  sibling(a, h)

//Secuenciación de árbol a siguiendo criterio primero en profundidad
  sequence()
```

Diseño Intérprete:

```
{
      //Memoria para las variables enteras: int_vars

      //Memoria para las variables árbol: arb_vars
}

programa : #(PROGRAMA decl_vars instrs) ;

decl_vars : #(VARIABLES (ID {almacenar árbol ID sin nodos en arb_vars })*) ;
```

```
instrs : #(INSTRUCCIONES  (instr)*) ;

instr : asig | op_arbol ;

asig : #(ASIG ID expr) {almacenar ID en int_vars con valor expr} ;

op_arbol : root | child | sibling | sequence ;

root : #(ROOT ID expr) {t=recuperar ID desde arb_vars ;
                        root(t, expr);
                        almacenar t en arb_vars ;} ;

child : #(CHILD a=node b:ID) {t=recuperar a desde arb_vars ;
                              child(t, b);
                              almacenar t en arb_vars ;} ;

sibling : #(SIBLING a:ID b:ID) {t=recuperar a desde arb_vars ;
                                sibling(t, b);
                                almacenar t en arb_vars ;} ;

node returns [s] : #(ROOT_NODE ID) { s = ID }

sequence : #(SEQUENCE ID) {t=recuperar ID desde arb_vars ;
                           sequence(t); } ;

expr returns [r] :  #(MAS a=expr b=expr){r=a+b;}
     | #(MENOS a=expr b=expr){r=a-b;}
     | #(POR a=expr b=expr){r=a*b;}
     | NUM {r=NUM;}
     | ID {consultar valor r de ID en int_vars, si no está devolver 0}
     ;
```

[IMPLEMENTACIÓN] Implementación del intérprete previamente diseñado mediante un Treeparser Antlr (use las reglas de la gramática dada en Anexo). Aclare especialmente la implementación del tipo árbol.

```
public class Arbol {
   Integer info;
   Arbol hijo;
   Arbol hermano;

   public Arbol(){
         info=null;
         hijo=null;
         hermano=null;
   }

   public void root(Integer i){
         info=i;
   }

   public void child(Arbol a){
         hijo=a;
   }

   public void sibling(Arbol a){
         hermano=a;
   }

   public void sequence(){
         System.out.print(info+";");
         if (hijo!=null)        this.hijo.sequence();
         if (hermano!=null) this.hermano.sequence();
   }
}
```

```
header{
      import java.util.*;
}
class Interprete3 extends TreeParser;
options{
      importVocab=Interprete;
}
{
      Map<String,Integer>int_vars=new HashMap<String,Integer>();
      Map<String,Arbol>arb_vars=new HashMap<String,Arbol>();

      public void crear_arbol(String a){
            Arbol aux=new Arbol();
            arb_vars.put(a,aux);
      }
}

programa : #(PROGRAMA decl_vars instrs) ;

decl_vars : #(VARIABLES (a:ID {crear_arbol(a.getText());})*) ;

instrs : #(INSTRUCCIONES  (instr)*) ;

instr : asig | op_arbol ;

asig {Integer i;} : #(ASIG a:ID i=expr) {int_vars.put(a.getText(),i);} ;

op_arbol {String s;}: root | child | sibling | s=node | sequence ;

root {Integer i; Arbol aux;} : #(ROOT a:ID i=expr) {aux=arb_vars.get(a.getText());
aux.root(i);} ;

child {Arbol aux1,aux2;String s;}: #(CHILD s=node b:ID)
                           {aux1=arb_vars.get(s); aux2=arb_vars.get(b.getText());
                             aux1.child(aux2);} ;

sibling {Arbol aux1,aux2;}: #(SIBLING a:ID b:ID)
                            {aux1=arb_vars.get(a.getText());
aux2=arb_vars.get(b.getText());
                              aux1.sibling(aux2);} ;

node returns[String s=null]: #(ROOT_NODE a:ID)  {s=new String(a.getText());} ;


sequence {Arbol aux;}: #(SEQUENCE a:ID)
                       {aux=arb_vars.get(a.getText()); System.out.print("\n");
                        aux.sequence();} ;

expr returns [Integer r=null] {Integer a,b;}:  #(MAS a=expr b=expr){r=a+b;}
     | #(MENOS a=expr b=expr){r=a-b;}
     | #(POR a=expr b=expr){r=a*b;}
     | n:NUM {r=new Integer(n.getText());}
     | i:ID {r=0;if (int_vars.keySet().contains(i.getText()))
r=int_vars.get(i.getText());}
     ;
```

PROCESADORES DEL LENGUAJES Septiembre 2019

NOMBRE: _____ GRUPO: _____

Lenguaje de Programación S (Compilación)

S es un lenguaje de programación que permite programar árboles de grado n. La declaración de variable en S crea un árbol sin nodos. Los nodos del árbol contienen información de tipo entera. Los variables enteras usadas en el programa no se declaran y toman valor 0 por defecto. El valor de éstas puede alterarse mediante el uso de asignaciones.
La semántica de las operaciones suma, resta y producto de enteros son las convencionales. No se consideran otras operaciones.
Se contemplan las siguientes operaciones para el tipo árbol: (1) `root`, (2) `child`, (3) `sibling` (4) `root_node` y (5) `sequence`. La operación `root` tiene dos argumentos, un árbol y un entero, y permite establecer como nodo raíz del árbol dicho entero. La operación `child` tiene dos argumentos, un nodo y un árbol, y permite establecer el árbol como hijo único del nodo. La operación `sibling` tiene como argumentos dos árboles y permite establecer el segundo como hermano del primero. La función `root_node` tiene como argumento un árbol con al menos un nodo y devuelve el nodo raíz de dicho árbol. La operación `sequence` tiene como argumentos un árbol y permite mostrar por pantalla la secuenciación de sus nodos mediante un recorrido en profundidad. A continuación, se muestra un programa S de ejemplo con anotaciones para aclarar su interpretación:

```
VARIABLES a,b,c,d;              //declaracion de 4 arboles: a, b, c y d.
INSTRUCCIONES
   x=2;                         //x vale 2
   root(a,x);                   //raiz de a vale 2
   x=3;                         //x vale 3
   root(b,1);                   //raiz de b vale 1
   root(c,x+2);                 //raiz de c vale 5
   child(root_node(a),b);       //el hijo de la raiz de a es b
   sibling(b,c);                //b tiene un hermano c
   child(root_node(b),d);       //el hijo de la raiz de b es d
   sequence(a)                  //muestra por pantalla: 2,1,5
   root(d,0);                   //raiz de d vale 0
   sequence(a)                  //muestra por pantalla: 2,1,0,5
```

SE PIDE: Compilador de lenguaje S a lenguaje Java. **(2,5 puntos)**

[DISEÑO] Diseño del compilador usando la gramática abstracta dada en Anexo. Aporte la compilación resultante del programa de ejemplo. Suponga que el programa S no tiene errores y se compila como una función Java main().

```
{
      abrir_fichero()

      cerrar_fichero()

      escribir(codigo)

    //Almacen para variables enteras: vars_declaradas;

    //Generar código para declaración de variables árbol
      declarar_arboles(vars)

   //Generar código asignación
     cod_asig(var, expr)
}

programa[String nombre] : {abrir_fichero(); escribir("public class nombre {");}
      #(PROGRAMA { escribir("public static void main(String[] args) {");}
           decl_vars instrs { escribir("}");})
      { escribir("}");cerrar_fichero();}  ;

decl_vars:
   #(VARIABLES (ID {añadir ID a vars;})*) {declarar_arboles(vars);} ;

instrs : #(INSTRUCCIONES  (instr)*) ;
```

```
instr : asig | op_arbol ;

asig  : #(ASIG ID expr) {cod_asig(ID,expr);} ;

op_arbol : root | child | sibling | sequence ;

root : #(ROOT ID expr) {escribir(ID+".root("+expr+");");} ;

child : #(CHILD s=node b:ID) {escribir(s+".child("+b+");");} ;

sibling : #(SIBLING a:ID b:ID) {escribir(a+".sibling("+b+");");} ;

node returns [s] : #(ROOT_NODE ID) { s=ID}

sequence {Arbol aux;}: #(SEQUENCE ID) {escribir(ID+".sequence();");} ;

expr returns [cod] :  #(MAS a=expr b=expr){cod=a+"+"+b;}
     | #(MENOS a=expr b=expr){cod=a+"-"+b;}
     | #(POR a=expr b=expr){cod=a+"*"+b;}
     | NUM {cod=NUM;}
     | ID {cod=ID;}
     ;
```

Compilación resultante del programa de ejemplo:

```
   public static void main(String[] args) {
      Arbol a, b, c, d;
      a=new Arbol();
      b=new Arbol();
      c=new Arbol();
      d=new Arbol();
      Integer x;
      x=2;
      a.root(x);
      x=3;
      b.root(1);
      c.root(x+2);
      a.child(b);
      b.sibling(c);
      a.sequence();
      System.out.print("\n");
      d.root(0);
      b.child(d);
      a.sequence();
      System.out.print("\n");
   }
```

[IMPLEMENTACIÓN] Implementación del compilador previamente diseñado mediante Treeparser Antlr (use las reglas de la gramática dada en Anexo).

```
header{
      import java.util.*;
      import java.io.*;
}
class Compilador extends TreeParser;
options{
      importVocab=Interprete;
}
{
      int ind=0;

      FileWriter f;

      String nombre_fichero;
```

```
      public void abrir_fichero(){
            try{
               nombre_fichero=nombre_fichero.substring(0,nombre_fichero.length()-4);
               f=new FileWriter("C:\\Docencia\\PL\\PL2019-
20\\Examenes\\sept19\\src\\"+nombre_fichero+".java");
            }catch(IOException e){}
      }

      public void cerrar_fichero(){
            try{
               f.close();
            }catch(IOException e){}
      }

      public void escribir(String cod){
            try{
            for(int i=0;i<ind;i++)
               f.write(' ');
            f.write(cod+"\n");
            }catch(IOException e){}
      }

      public void comienzo_clase(){
            escribir("public class "+nombre_fichero+"{");
            ind+=3;
      }

      public void comienzo_main(){
            escribir("public static void main(String[] args) {");
            ind+=3;
      }

      public void fin_elemento(){
            ind-=3;
            escribir("}");
      }

      Set<String>vars_declaradas=new HashSet<String>();

      public void declarar_arboles(Set<String>vars){
         String cod = new String();
         cod+="Arbol ";
         int cont=0;
         for(String s:vars){
              cont++;
            if (cont>1)
               cod+=", "+s;
            else
               cod+=s;
         }
         cod+=";";
         escribir(cod);
         for(String s:vars)
            escribir(s+"=new Arbol();");
      }

      public void cod_asig(String var, String expr){
         if (!vars_declaradas.contains(var))
            escribir("Integer "+var+";");
          vars_declaradas.add(var);
            escribir(var+"="+expr+";");
      }
}

programa[String nombre] :
   {nombre_fichero=new String(nombre);abrir_fichero(); comienzo_clase();}
      #(PROGRAMA {comienzo_main();} decl_vars instrs {fin_elemento();})
```

```
   {fin_elemento();cerrar_fichero();}  ;

decl_vars {Set<String>vars=new HashSet<String>();}:
   #(VARIABLES (a:ID {vars.add(a.getText());})*) {declarar_arboles(vars);} ;

instrs : #(INSTRUCCIONES  (instr)*) ;

instr : asig | op_arbol ;

asig {String e;} : #(ASIG a:ID e=expr) {cod_asig(a.getText(),e);} ;

op_arbol {String s;}: root | child | sibling | s=node | sequence ;

root {String e;} : #(ROOT a:ID e=expr) {escribir(a.getText()+".root("+e+");");} ;

child {Arbol aux1,aux2;String s;}: #(CHILD s=node b:ID)
                            {escribir(s+".child("+b.getText()+");");} ;

sibling {Arbol aux1,aux2;}: #(SIBLING a:ID b:ID)
                             {escribir(a.getText()+".sibling("+b.getText()+");");} ;

node returns[String s=null]: #(ROOT_NODE a:ID) {s=new String(a.getText());};

sequence {Arbol aux;}: #(SEQUENCE a:ID)
                       {escribir(a.getText()+".sequence();");
                        escribir("System.out.print(\"\\n\");");} ;

expr returns [String r=new String()] {String a,b;}:  #(MAS a=expr b=expr){r=a+"+"+b;}
     | #(MENOS a=expr b=expr){r=a+"-"+b;}
     | #(POR a=expr b=expr){r=a+"*"+b;}
     | n:NUM {r=new String(n.getText());}
     | i:ID {r=new String(i.getText());}
     ;
```