Escuela Técnica Superior de Ingeniería Informática

Procesadores de Lenguajes

Departamento de Lenguajes y Sistemas Informáticos

Miguel Angel Cifredo Campos

TERCERO

--- REVERSO DE PORTADA ---

1 Introducción.

1.1 Objetivo.

Todo lenguaje posee sintaxis y semántica. La sintaxis se refiere a la *forma* de las sentencias del lenguaje. La semántica se refiere al *significado* de tales sentencias. La especificación de lenguaje se realizará de manera informal y basada en ejemplos típicos.

El diseño del procesador está basado en un patrón arquitectónico denominado *tubería-filtro*. Sobre una *tubería* se desplazan distintas representaciones de datos o de programas que son procesadas por *filtros* especializados.

Hay múltiples tecnologías para construir procesadores de lenguajes. En el presente curso utilizaremos una tecnología llamada **ANTLR** (acrónimo de *ANother Tool for Language Recognition*).

1.2 Especificación del lenguaje.

La especificación del lenguaje a procesar estará basada en definiciones informales y ejemplos típicos.

Ejemplo 1. Especificación de un lenguaje de programación.

Definición informal de un lenguaje hipotético de programación al que llamaremos LF: Se trata de un lenguaje de programación secuencial con variables enteras y dos tipos de instrucciones: Definición de variables con expresión entera (DEF) y, Evaluación de variables (EVAL). No se acepta el uso de variables sin declarar. La declaración de variables asocia valor 0 por defecto. Cuando se define una variable se le asocia una expresión entera "sin evaluar". Esta expresión se evalúa justo al ejecutar la instrucción EVAL. Toda variable definida sobre sí misma se le asocia un valor indefinido.

A continuación se muestra un programa de ejemplo para ver una sintaxis del lenguaje LF.

```
VARIABLES x, y, z, a, b;
INSTRUCCIONES
      a DEF -1;
      b DEF (a+1);
      EVAL b;
      a DEF 2;
      EVAL b;
      b DEF b + 1;
      z DEF 2*y;
      EVAL z;
      EVAL b;
```

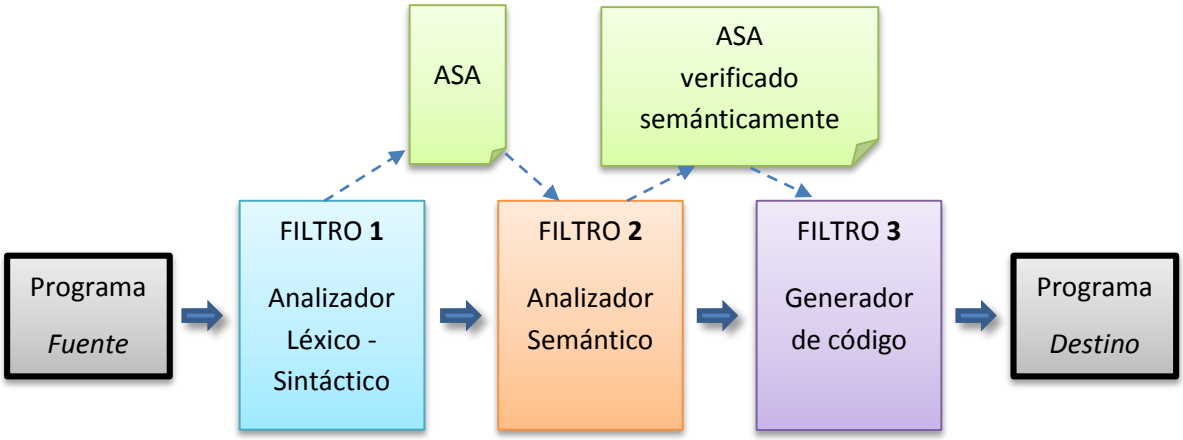
Para precisar la semántica del lenguaje LF, se aporta la ejecución del programa anterior:

```
b = 0              // primera instrucción EVAL
b = 3              // segunda instrucción EVAL
z = 0              // tercera instrucción EVAL
b = indefinido     // cuarta  instrucción EVAL
```

1.3 Diseño del procesador.

El diseño del procesador se basará en el patrón arquitectónico denominado *tubería-filtro*. Sobre la tubería se desplazan distintas representaciones de datos o programas que son procesadas por filtros especializados. La naturaleza y número de filtros dependerá del problema a resolver.

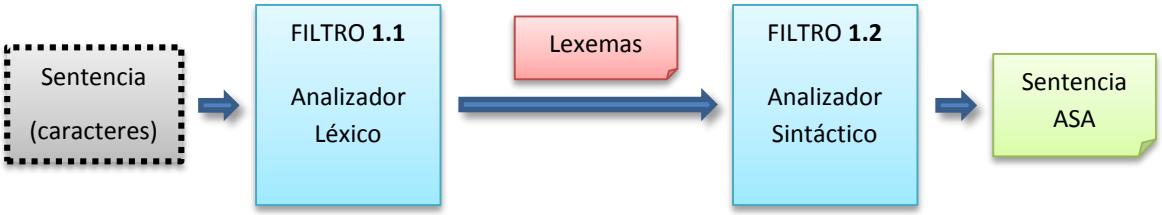
El siguiente gráfico muestra la arquitectura de un compilador con tres filtros: Analizador léxico-sintáctico, Analizador semántico y Generador de código. Partiendo de un programa fuente, en forma de secuencia de caracteres, se generará un árbol de sintaxis abstracta (asa) (Filtro 1). A continuación se comprueba, con un analizador semántico (Filtro 2), si el árbol de sintaxis abstracta (asa) cumple condiciones semánticas suficientes para asignar semántica al programa fuente. Si esto es así, el árbol de sintaxis abstracta (asa), verificado semánticamente, se puede usar como entrada a un generador de código (Filtro 3). Este último filtro traducirá el programa original a un programa en el lenguaje destino.



1.4 Filtro 1: Análisis Léxico-Sintáctico.

Su objetivo es decidir si la descripción textual de entrada tiene la forma adecuada.

El analizador léxico-sintáctico es en realidad un doble filtro. El primer filtro es el *analizador léxico* que toma la sentencia entrada en forma de secuencia de caracteres y produce una salida en forma de secuencia de lexemas. El segundo filtro es el *analizador sintáctico* que toma la secuencia de lexemas producido por el analizador léxico y produce una versión de la entrada en forma de árbol de sintaxis abstracta (asa).



Lexema: Unidad textual (palabra o signo de puntuación) utilizados en el lenguaje a procesar. Los lexemas son unidades mínimas desde las que se construyen estructuras sintácticas. El ejemplo 2 muestra la secuencia de lexemas en la que se transforma el programa mostrado en el ejemplo 1. Entre paréntesis se muestra la palabra o signo de puntuación asociado al lexema. Se indicarán en mayúsculas.

Ejemplo 2. Programa del lenguaje LF como secuencia de lexemas.

```
VARIABLES(VARIABLES), VAR(x), COMA(,), VAR(y), COMA(,), VAR(z), COMA(,), VAR(a), COMA(,), VAR(b),
PyC(;), INSTRUCCIONES(INSTRUCCIONES), VAR(a), DEF(DEF), MENOS(-), NUMERO(1), PyC (;), VAR(b),
lexema:10(DEF), PA((), VAR(a), MAS(+), NUMERO (1), PC()), PyC (;), EVAL(EVAL), VAR(b), PyC(;),
VAR(a), DEF(DEF), NUMERO (2), PyC (;), EVAL(EVAL), VAR(b), PyC (;), VAR(b), DEF(DEF), VAR(b),
MAS(+), NUMERO(1), PyC (;), VAR(z), DEF(DEF), NUMERO (2), POR(*), VAR(y), PyC (;), EVAL(EVAL),
VAR(z), PyC (;), EVAL(EVAL), VAR(b), PyC (;)
```

Ejemplo 3. Analizador Léxico.

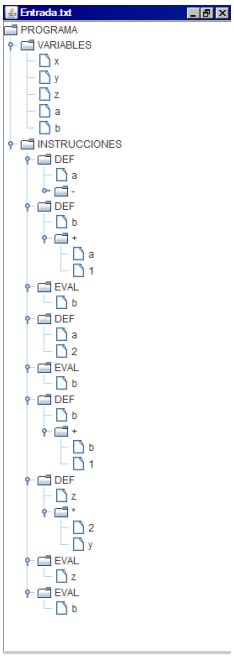
El siguiente ejemplo muestra el código ANTLR del *analizador léxico* que genera la secuencia de lexemas mostrada en el ejemplo 2.

```
class Analex extends Lexer;

options{
        importVocab    = Anasint;
}

tokens{
        VARIABLES      = "VARIABLES";
        INSTRUCCIONES  = "INSTRUCCIONES";
        EVAL           = "EVAL";
        DEF            = "DEF";
}

protected NL     : "\r\n"  {newline();} ;
protected DIGITO : '0'..'9';
protected LETRA  : 'a'..'z'|'A'..'Z';

BTF    : (' '|'\t'|NL)     {$setType(Token.SKIP);} ;
NUMERO : (DIGITO)+ ;
VAR    : (LETRA)+ ;
PA     : '(' ;
PC     : ')' ;
PyC    : ';' ;
COMA   : ',' ;
MAS    : '+' ;
MENOS  : '-' ;
POR    : '*' ;
```

Árbol de Sintaxis Abstracta (asa): Es un árbol construido desde lexemas de acuerdo a las reglas sintácticas del lenguaje.

Ejemplo 4. Programa LF como árbol de sintaxis abstracta (asa).

Se muestra el árbol de sintaxis abstracta (asa) construido desde la secuencia de lexemas mostrado en el ejemplo 2 para el programa mostrado en ejemplo 1.



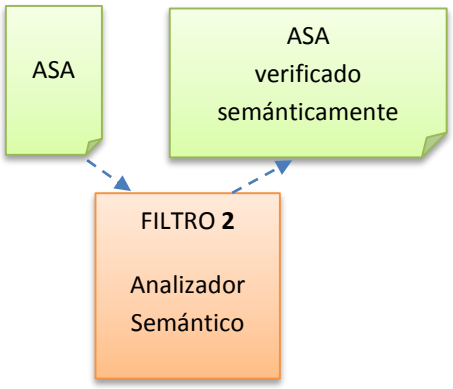
Ejemplo 5. Analizador sintáctico.

El ejemplo 5 muestra el código ANTLR del *analizador sintáctico* que genera el árbol de sintaxis abstracta (asa) mostrado en el ejemplo 4.

```
class Anasint extends Parser;
options{
        buildAST = true;
}
tokens{
        PROGRAMA;
}
programa      : variables instrucciones EOF!
                {#programa = #(#[PROGRAMA, "PROGRAMA"], ##);}
              ;
variables     : VARIABLES^ idents PyC!
              ;
idents        : (VAR COMA) => VAR COMA! idents
              | VAR
              |
              ;
instrucciones : INSTRUCCIONES^ (definicion|evaluacion)*
              ;
definicion    : VAR DEF^ expr PyC!
              ;
evaluacion    : EVAL^ idents PyC!
              ;
expr          : (expr1 MAS)   => expr1 MAS^ expr
              | (expr1 MENOS) => expr1 MENOS^ expr
              | expr1
              ;
expr1         : (expr2 POR)   => expr2 POR^ expr1
              | expr2
              ;
expr2         : NUMERO
              | MENOS^ expr
              | VAR
              | PA! expr PC!
              ;
```

Tanto el *analizador léxico* como el *analizador sintáctico* pueden detectar errores en sus procesamientos respectivos. Los errores detectados por el analizador léxico se denominan *errores léxicos* y los detectados por el analizador sintáctico se denominan *errores sintácticos*.

1.5 Filtro 2: Análisis Semántico.



¿Podemos asociarle significado a un programa que ha superado el filtro léxico-sintáctico? Generalmente no. El filtro léxico-semántico no es capaz de procesar condiciones sintácticas dependientes del contexto tales como el uso de variables no declaradas o condiciones semánticas tales como la corrección del tipo asociado a las expresiones. El objetivo del filtro semántico es comprobar el cumplimiento de tales condiciones.

Ejemplo 6. Analizador semántico.

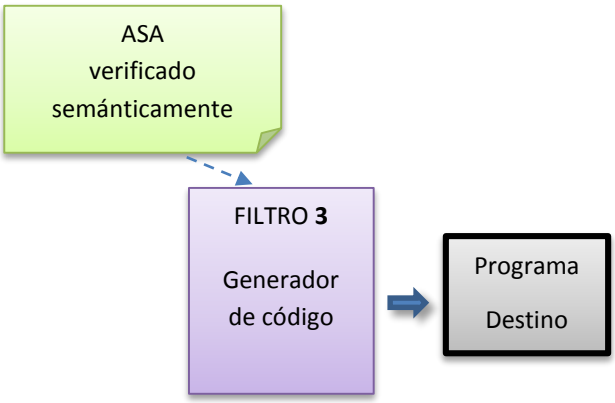
El ejemplo 6 muestra el código ANTLR del *analizador semántico* que verifica el asa mostrado en el ejemplo 4.

```
header{
      import java.util.*;
      import antlr.*;
}
class Anasint4 extends TreeParser;

options {
      importVocab = Anasint;
}
{
      Set<String> variables = new HashSet<String>();
      ASTFactory factory    = new ASTFactory();
      public void almacenar(String var){
            variables.add(var);
      }
      public boolean ocurrencia(String var, AST expr){
            switch(expr.getType()){
                  case NUMERO:      return false;
                  case VAR:         return expr.getText().equals(var);
                  case MAS:
                  case POR:         return ocurrencia(var, expr.getFirstChild())
                                       || ocurrencia(var, expr.getFirstChild().getNextSibling());
                  case MENOS: if (expr.getFirstChild().getNextSibling()==null)
                                    return ocurrencia(var, expr.getFirstChild());
                              else
                                    return ocurrencia(var, expr.getFirstChild())
                                       || ocurrencia(var, expr.getFirstChild().getNextSibling());
            }
            return false;
      }
}
programa      : #(PROGRAMA variables instrucciones )
              ;
variables     : #(VARIABLES (v:VAR {almacenar(v.getText());} )*)
              ;
instrucciones : #(INSTRUCCIONES (definicion|evaluacion)*)
              ;
definicion    : #(DEF v:VAR e:expr)
              { if (!variables.contains(v.getText()))
                   System.out.println("Variable no declarada: " + #v.getText());
                if (ocurrencia(#v.getText(), #e))
                   System.out.println("Variable indefinida: " + #v.getText());
              } ;
evaluacion    : #(EVAL v:VAR)
              ;
expr          : #(MAS expr expr)
              | #(POR expr expr)
              | NUMERO
              | VAR
              | (#(MENOS expr expr)) => #(MENOS expr expr)
              | #(MENOS expr)
              ;
```

Los errores detectados por el analizador semántico se denominan *errores semánticos*.

1.6 Filtro 3: Generador de Código.

El objetivo de la generación de código es traducir el programa a otro lenguaje (normalmente ejecutable). El programa que supera el filtro semántico es un programa al que se le puede asociar significado. Por tanto, este programa se puede ejecutar, bien interpretándolo directamente o bien a través de una traducción a otro lenguaje. El generador de código no produce errores, es una función total.



En nuestro ejemplo, la generación de código consiste en convertir el programa original en una versión Java de éste.

Ejemplo 7. Generador de código.

```
header{
      import java.util.*;
      import antlr.*;
      import java.io.*;
}

class Anasint3 extends TreeParser;

options
{
      importVocab = Anasint;
}


{
      FileWriter fichero;

      private void open_file(){
            try{
                  fichero = new FileWriter("_Programa.java");
            }catch(IOException e)
            { System.out.println("open_file (exception): " + e.toString()); }
      }


      private void close_file(){
            try{
                  fichero.close();
            }catch(IOException e)
               { System.out.println("close_file (exception): " + e.toString()); }
      }


      int espacios = 0;
      private void gencode_espacios(){
            try{
                  for (int i = 1; i <= espacios; i++)
                        fichero.write(" ");
            }catch(IOException e)
                 { System.out.println("gencode_espacios (exception): " + e.toString()); }
      }
```

```
ASTFactory             factory    = new ASTFactory();
Hashtable<String, AST> vars       = new Hashtable<String, AST>();
Hashtable<String, AST> vars2      = new Hashtable<String, AST>();
List<String>           evals_pend = new LinkedList<String>();
Integer                num_evals  = new Integer(0);

public void declarar_variable(String var){
      AST nodo = new CommonAST();
      nodo.setType(NUMERO);
      nodo.setText("0");
      vars.put(var,nodo);
}

public void definir_variable(String var, AST expr){
      vars.put(var,expr);
}

public void evaluar_variable(String var){
      Integer i;
      String aux = new String(var);
      num_evals++;
      Enumeration<String> v = vars.keys();
      while (v.hasMoreElements()){
            String key = v.nextElement();
            if (cierre(var).contains(key)){
                  aux = key + num_evals.toString();
                  vars2.put(aux,sustitucion_expr(vars.get(key)));
            }
      }
      try{
            v = vars2.keys();
            while (v.hasMoreElements()){
                  String key = v.nextElement();
                  gencode_espacios();
                  fichero.write("private static int " + key + "(){\n");
                  espacios++;
                  gencode_espacios();
                  fichero.write("return " + gencode_exp(vars2.get(key)) + ";\n");
                  espacios--;
                  gencode_espacios();
                  fichero.write("}\n");
            }
      }catch(IOException e){}
      vars2.clear();
      evals_pend.add(var+num_evals.toString());
}

public Set<String> cierre(String var){
      Set<String> resultado = new HashSet<String>();
      int size_ant = 0;
      int size_act = 0;
      resultado.add(var);
      size_act = resultado.size();
      while (size_ant != size_act){
            resultado.addAll(calcular_variables(vars.get(var)));
            size_ant = size_act;
            size_act = resultado.size();
      }
      return resultado;
}

public Set<String> calcular_variables(AST expr){
      Set<String> aux = new HashSet<String>();
      switch(expr.getType()){
            case NUMERO: break;
            case VAR:          aux.add(expr.getText()); break;
            case MAS:
            case POR:          aux.addAll(calcular_variables(expr.getFirstChild()));
                               aux.addAll(calcular_variables(expr.getFirstChild().getNextSibling()));
                         break;
            case MENOS:        aux.addAll(calcular_variables(expr.getFirstChild()));
                         if (expr.getFirstChild().getNextSibling() != null)
                               aux.addAll(calcular_variables(expr.getFirstChild().getNextSibling()));
                         break;
            default:           aux = null;
      }
      return aux;
}
```

```
public AST sustitucion_expr(AST expr){
      AST nodo;
      String aux;
      switch(expr.getType()){
            case NUMERO: nodo = factory.dupTree(expr);
                         break;
            case VAR   : aux = expr.getText();
                         nodo = new CommonAST();
                         aux = aux + num_evals.toString();
                         nodo.setType(VAR);
                         nodo.setText(aux);
                         break;
            case MAS   : nodo = new CommonAST();
                         nodo.setType(MAS);
                         nodo.setText("+");
                         nodo.setFirstChild(sustitucion_expr(expr.getFirstChild()));
                         nodo.getFirstChild().setNextSibling(
                              sustitucion_expr(expr.getFirstChild().getNextSibling())
                         );
                         break;
            case POR   : nodo = new CommonAST();
                         nodo.setType(POR);
                         nodo.setText("*");
                         nodo.setFirstChild(sustitucion_expr(expr.getFirstChild()));
                         nodo.getFirstChild().setNextSibling(
                              sustitucion_expr(expr.getFirstChild().getNextSibling())
                         );
                         break;
            case MENOS : nodo = new CommonAST();
                         nodo.setType(MENOS);
                         nodo.setText("-");
                         nodo.setFirstChild(sustitucion_expr(expr.getFirstChild()));
                         if (expr.getFirstChild().getNextSibling()!=null)
                            nodo.getFirstChild().setNextSibling(
                                 sustitucion_expr(expr.getFirstChild().getNextSibling())
                            );
                         break;
            default:     nodo = null;
      }
      return nodo;
}

private void gencode_begin_class(){
      try{
            gencode_espacios();
            fichero.write("import java.io.*;\n");
            gencode_espacios();
            fichero.write("public class _Programa" + "\n");
            gencode_espacios();
            fichero.write("{\n");
            espacios++;
      }catch(IOException e){}
}

private void gencode_main(){
      try{
            gencode_espacios();
            fichero.write("public static void main(String[] args) {\n");
            espacios++;
            Iterator<String> it = evals_pend.listIterator();
            while (it.hasNext()){
                  gencode_espacios();
                  fichero.write("System.out.println(" + it.next() + "());\n");
            }
            espacios--;
            gencode_espacios();
            fichero.write("}\n");
      }catch(IOException e){}
}

private void gencode_end_class(){
      try{
            espacios--;
            gencode_espacios();
            fichero.write("}");
      }catch(IOException e){}
}
```

```
      public String gencode_exp(AST expr){
            switch(expr.getType()){
                  case NUMERO: return expr.getText();
                  case VAR   : return expr.getText() + "()";
                  case MAS   : return
gencode_exp(expr.getFirstChild()) + "+" + gencode_exp(expr.getFirstChild().getNextSibling());
                  case POR   : return
gencode_exp(expr.getFirstChild()) + "*" + gencode_exp(expr.getFirstChild().getNextSibling());
                  case MENOS : if (expr.getFirstChild().getNextSibling() != null)
                                    return
gencode_exp(expr.getFirstChild()) + "-" + gencode_exp(expr.getFirstChild().getNextSibling());
                               else
                                    return "-" + gencode_exp(expr.getFirstChild());
                  default    : return null;
             }
      }

}

programa      : {open_file(); gencode_begin_class();}
                #(PROGRAMA variables instrucciones)
                {gencode_main(); gencode_end_class(); close_file();}
              ;
variables     : #(VARIABLES (v:VAR {declarar_variable(v.getText());} )*)
              ;
instrucciones : #(INSTRUCCIONES (definicion|evaluacion)*)
              ;
definicion    : #(DEF v:VAR e:expr) {definir_variable(v.getText(), e);}
              ;
evaluacion    : #(EVAL v:VAR)       {evaluar_variable(v.getText());}
              ;
expr          : #(MAS expr expr)
              | #(POR expr expr)
              | NUMERO
              | VAR
              | (#(MENOS expr expr)) => #(MENOS expr expr)
              | #(MENOS expr)
              ;
```

Ejemplo 8. Código Java generado para el programa del lenguaje LF.

A continuación se presenta el código Java generado para el programa del lenguaje LF del ejemplo 1.

```
import java.io.*;

public class _Programa {
     private static int a1() {
        return -1;
     }
     private static int b1() {
        return a1() + 1;
     }
     private static int a2() {
        return 2;
     }
     private static int b2() {
        return a2() + 1;
     }
     private static int y3() {
        return 0;
     }
     private static int z3() {
        return 2 * y3();
     }
     private static int b4() {
        return b4() + 1;
     }
     public static void main(String[] args) {
        System.out.println(b1());
        System.out.println(b2());
        System.out.println(z3());
        System.out.println(b4());
     }
}
```

2 Análisis Léxico-Sintáctico.

2.1 Objetivo.

Dar respuesta a las siguientes preguntas:

- ✓ ¿Qué es el análisis léxico-sintáctico de un lenguaje?
- ✓ ¿Cómo se diseña y se construye un analizador léxico-sintáctico?

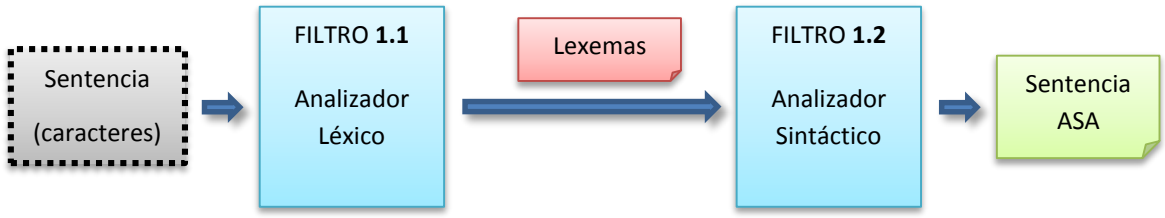
2.2 Lenguaje de ejemplo.

Para presentar este tema utilizaremos un lenguaje de programación hipotético al que llamaremos LF descrito en el apartado 1.2.

2.3 ¿Qué es el análisis léxico-sintáctico de un lenguaje?

El análisis léxico-sintáctico tiene por objeto reconocer la forma de las sentencias de un lenguaje; o sea, decidir si la descripción textual de entrada tiene la forma correcta.

Reconocer la forma de una sentencia implica reconocer sus lexemas y las estructuras sintácticas. El resultado del análisis léxico-sintáctico puede ser bien un error de reconocimiento o bien una versión de la sentencia reconocida en forma de árbol de sintaxis abstracta (asa).



Para reconocer los lexemas de un lenguaje usaremos *expresiones regulares* y, para reconocer estructuras sintácticas, usaremos *gramáticas independientes de contexto* (*gramática* en adelante).

Una **gramática** es un conjunto de reglas donde cada una de ellas es de la siguiente forma genérica:

$$\text{cabeza} : \text{cuerpo}_1 \mid \text{cuerpo}_2 \mid \ldots \mid \text{cuerpo}_N \qquad \text{siendo } N \geq 1$$

La cabeza de la regla es un símbolo llamado *no terminal* que representa una estructura sintáctica. El cuerpo de la regla está compuesto por símbolos terminales (lexemas) y símbolos no terminales. La composición de estos símbolos se consigue haciendo uso de alternativas, iteraciones y yuxtaposiciones.

Ejemplo 1. Gramática del lenguaje LF.

A continuación se muestra la gramática del lenguaje LF.

```
programa      : variables instrucciones
              ;
variables     : VARIABLES idents PyC
              ;
idents        : VAR COMA idents
              | VAR
              |
              ;
instrucciones : INSTRUCCIONES (definicion|evaluacion)*
              ;
definicion    : VAR DEF expr PyC
              ;
evaluacion    : EVAL idents PyC
              ;
expr          : expr1 MAS expr
              | expr1 MENOS expr
              | expr1
              ;
expr1         : expr2 POR expr1
              | expr2
              ;
expr2         : NUMERO
              | MENOS expr
              | VAR
              | PA expr PC
              ;
```

El carácter * indica que la estructura que lo precede es repetible de 0 a *n* veces.

Dada una gramática *G*, el análisis sintáctico se puede realizar aplicando reglas de *G* de izquierda a derecha (análisis descendente). La aplicación de una regla se denomina *derivación*. En el análisis sintáctico descendente, el objetivo es encontrar una derivación que reconozca la sentencia. Los símbolos no terminales son símbolos generativos porque producen derivaciones. Sin embargo, los lexemas son símbolos no generativos porque no producen derivaciones.

Ejemplo 2. Derivación.

A continuación veremos derivaciones producidas desde la gramática del ejemplo 1.

```
programa -> variables instrucciones -> VARIABLES idents PyC instrucciones -> VARIABLES VAR COMA
idents PyC instrucciones -> …
```

Si no es posible construir una derivación que genere la sentencia completa, como secuencia de lexemas, entonces dicha sentencia no pertenece al lenguaje especificado en la gramática.

Una gramática es *determinista* si toda sentencia perteneciente al lenguaje especificado por dicha gramática tiene una única derivación. La gramática propuesta para el lenguaje LF (ejemplo 1) es *no determinista*. El no determinismo se debe a la existencia de prefijos comunes para distintas alternativas en una misma regla. Por ejemplo, la regla `idents : VAR COMA idents | VAR | ;` tiene el prefijo común `VAR` en sus dos primeras alternativas. Los prefijos comunes pueden producirse sobre símbolos *no terminales*. Por ejemplo sobre los símbolos *expr1* y *expr2* en las reglas correspondientes a la definición de expresión.

Una forma de resolver el problema del prefijo común es usar *predicados sintácticos*. Los predicados sintácticos son condiciones añadidas a las alternativas para reconocer de manera tentativa un determinado prefijo de lexemas. Si es así, se selecciona la alternativa correspondiente. Si no, se selecciona la siguiente alternativa en la regla (y así sucesivamente). La siguiente gramática resuelve el problema de la gramática original (ejemplo 1) usando prefijos comunes.

Ejemplo 3. Gramática determinista.

El ejemplo 1 tiene dos fallos: { Falta de validez: `VARIABLES x,;` ; Falta de completitud: { `(VARIABLES;)` ; `(VARIABLES,),` } } que se solucionan como:

```
variables     : VARIABLES idents
              ;
idents        : VAR idents2.
              |
              ;
idents2       : COMA VAR idents2
              | PyC
              ;
```

El problema de la gramática determinista queda resuelto por el siguiente código:

```
programa      : variables instrucciones
              ;
variables     : VARIABLES idents PyC
              ;
idents        : (VAR COMA) => VAR COMA idents      // regla con predicado sintáctico.
              | VAR
              |
              ;
instrucciones : INSTRUCCIONES (definicion|evaluacion)*
              ;
definicion    : VAR DEF expr PyC
              ;
evaluacion    : EVAL idents PyC
              ;
expr          : (expr1 MAS)   => expr1 MAS expr    // regla con predicado sintáctico.
              | (expr1 MENOS) => expr1 MENOS expr  // regla con predicado sintáctico.
              | expr1
              ;
expr1         : (expr2 POR)   => expr2 POR expr1   // regla con predicado sintáctico.
              | expr2
              ;
expr2         : NUMERO
              | MENOS expr
              | VAR
              | PA expr PC
              ;
```

Los **lexemas** o **tokens** son los *símbolos terminales* que aparecen en la gramática y representan las distintas categorías de palabras y símbolos de puntuación que podemos encontrar en una sentencia del lenguaje. Los lexemas de un lenguaje se definen con reglas de la forma:

`LEXEMA : expresión regular`

donde *expresión regular* puede ser:

carácter	ej. '('
concatenación de caracteres	ej. "INSTRUCCIONES"
rango de caracteres	ej. '0' .. '9'
conjunto negativo de caracteres	ej. ~ ('0'|'1')
opción	ej. ('a'..'z') ?
alternativa	ej. 'a'..'z' | 'A'..'Z'
cierre	ej. (LETRA) +

Ejemplo 4. Lexemas de LF.

Las reglas que definen los lexemas del lenguaje LF se muestran a continuación.

```
// Lexemas auxiliares (se usan en la definición de otros lexemas)
protected NL     : "\r\n";            //concatenación
protected DIGITO : '0'..'9';          //rango
protected LETRA  : 'a'..'z'|'A'..'Z'; //alternativa de rangos

// Lexemas no auxiliares
BTF          : (' '|'\t'|NL);         //alternativa
NUMERO       : (DIGITO)+;             //cierre
VAR          : (LETRA)+;              //cierre
VARIABLES    : "VARIABLES";           //concatenación
INSTRUCCIONES: "INSTRUCCIONES";       //concatenación
EVAL         : "EVAL";                //concatenación
DEF          : "DEF" ;                //concatenación
PA           : '('   ;                //carácter
PC           : ')'   ;                //carácter
PyC          : ';'   ;                //carácter
COMA         : ','   ;                //carácter
MAS          : '+'   ;                //carácter
MENOS        : '-'   ;                //carácter
POR          : '*'   ;                //carácter
```

Un **árbol de sintaxis abstracta** (asa) es un árbol que refleja la estructura léxico-sintáctica de una sentencia. Los nodos del árbol de sintaxis abstracta (asa) son *lexemas*. Se distinguen dos tipos de lexemas en el árbol de sintaxis abstracta asa: Lexemas producidos por el analizador léxico y Lexemas producidos por el analizador sintáctico. Estos últimos sirven para identificar estructuras sintácticas.

Para construir un árbol de sintaxis abstracta (asa) tenemos que anotar la gramática. El siguiente ejemplo 5 muestra la gramática del ejemplo 1 con las anotaciones requeridas para construir su árbol de sintaxis abstracta (asa) correspondiente.

Ejemplo 5. Gramática que construye árboles de sintaxis abstracta.

```
programa      : variables instrucciones EOF!
                {#programa = #(#[PROGRAMA, "PROGRAMA"], ##);}
              ;
variables!    : a:VARIABLES b:idents PyC
                {#variables = #(#a, #b);}
              ;
idents        : (VAR COMA) => VAR COMA! idents
              | VAR
              |
              ;
instrucciones : INSTRUCCIONES^ (definicion|evaluacion)*
              ;
definicion    : VAR DEF^ expr PyC!
              ;
evaluacion    : EVAL^ idents PyC!
              ;
expr          : (expr1 MAS)   => expr1 MAS^ expr
              | (expr1 MENOS) => expr1 MENOS^ expr
              | expr1
              ;
expr1         : (expr2 POR)   => expr2 POR^ expr1
              | expr2
              ;
expr2         : NUMERO
              | MENOS^ expr
              | VAR
              | PA! expr PC!
              ;
```

El árbol de sintaxis abstracta (asa) es una expresión del tipo: `#(nodo, subárbol1 , … , subarbolk)` donde *nodo* es un árbol de sintaxis abstracta (asa) elemental formado por un único lexema, y donde subárbol$_1$, …, subarbol$_k$ son los árbol asas hijos de *nodo*.

El árbol asa elemental o *nodo* se puede expresar de diferentes maneras:

`#[LEXEMA, "txt…"]`	Donde `LEXEMA` es el identificador del lexema y `"txt…"` es el texto asociado al lexema. Por ejemplo, `#[PROGRAMA, "PROGRAMA"]`
`#[LEXEMA]`	Donde `LEXEMA` es el identificador del lexema.
`LEXEMA`	Donde `LEXEMA` es el identificador del lexema.

Cada símbolo *no terminal* de la gramática puede sintetizar un árbol asa o una secuencia de árboles asas. Por ejemplo, el símbolo `idents` sintetiza una secuencia de árboles asas y el símbolo `programa` sintetiza un único árbol asa. Cada lexema de la gramática, por el contrario, sólo puede sintetizar un único árbol asa con un único nodo.

Podemos referenciar los árboles asas sintetizados con etiquetas. Por ejemplo, la siguiente regla `variables! : a:VARIABLES b:idents PyC;` define dos etiquetas: la etiqueta *a* para referirse al árbol asa `#[VARIABLES]` y la etiqueta *b* para referirse al árbol asa `#idents`.

Para referirnos a todos los árboles asas sintetizados por el cuerpo de una regla se tiene la expresión: `##`.

La **programación de árboles asas** es una actividad importante en el desarrollo de un procesador. ANTLR construye árboles asas de forma automática según el orden de derivación. Por ejemplo, la regla `definicion : VAR DEF expr PyC;` construiría un árbol asa `#definicion` con nodo raíz `VAR` y con tres árboles asas hijos (de izquierda a derecha: `#[DEF]` , `#expr` y `#[PyC]`).

Podemos cambiar la forma de sintetizar el árbol asa haciendo uso de los operadores `^` y `!` .

El operador `^` sólo se puede usar como sufijo de un lexema y sirve para enraizar.

El operador `!` se puede usar como sufijo de cualquier símbolo y se evitar la construcción de árboles asas.

Por ejemplo, la regla `definicion: VAR DEF^ expr PyC!;` construye un árbol asa llamado `#definicion` donde `DEF` es la raíz, `VAR` es el primer árbol asa hijo (con un único nodo) y `expr` es el segundo árbol asa hijo. Así mismo, `PyC` es un lexema que no forma parte del árbol asa. Cuando este operador `!` se usa como sufijo en la cabeza de la regla, la regla no construye ningún árbol asa de forma automática.

ANTLR proporciona una biblioteca para programar árboles de sintaxis abstractas (asas) llamada `antlr`. Esta biblioteca incluye una interfaz `AST` para el tipo árbol sintaxis abstracta (asa) y la clase `CommonAST` para implementarlo. El nodo del árbol asa tiene dos campos principales: el lexema y el texto asociado al lexema. Podemos consultar estos campos con las operaciones: `getType()` y `getText()`. Podemos modificar estos campos con las operaciones: `setType(int)` y `setText(String)`.

Las operaciones `getFirstChild()` y `getNextSibling()` servirán para visitar los nodos del árbol asa y las operaciones `setFirstChild(AST)` and `setNextSibling(AST)` para construir el árbol asa.

ANTLR también proporciona la clase `ASTFactory` que permite hacer una copia de un árbol asa mediante `dupTree(AST)` y una copia de una secuencia de árboles asas mediante `dupList(AST)`.

2.4 ¿Cómo se diseña y construye un analizador Léxico-Sintáctico?

Recomendamos los siguientes pasos para el diseño de un analizador léxico-sintáctico.

Dado un lenguaje,

1) Diseñar una gramática determinista.
2) Diseñar los lexemas definidos en la gramática.
3) Implementar una primera versión del analizador léxico-sintáctico en ANTLR.
4) Testar dicha versión.
5) Anotar la gramática para construir árboles de sintaxis abstracta (asa).
6) Implementar la segunda versión del analizador léxico-sintáctico en ANTLR.
7) Testar el analizador con distintas clases de sentencias del lenguaje.

El **diseño de una gramática** se basa en un conjunto de recomendaciones:

- ✓ La estructura sintáctica del lenguaje debe interpretarse de forma jerárquica. La cabeza de la regla pone nombre a la estructura sintáctica y el cuerpo define dicha estructura.
- ✓ Las estructuras alternativas del lenguaje pueden expresarse con cuerpos alternativos. Por ejemplo, `definicion|evaluacion`.
- ✓ Las estructuras repetitivas del lenguaje pueden expresarse con reglas iterativas o recursivas. Por ejemplo, `INST (definicion|evaluacion)*` y `expr: expr1 MAS expr | expr1 MENOS expr | expr1`.
- ✓ Las piezas elementales de una estructura se formalizan con lexemas. Por ejemplo, `INSTRUCCIONES`.
- ✓ Los cuerpos de las reglas deben ser simples de interpretar. De esta manera, resultará más inteligible la estructura sintáctica del lenguaje.
- ✓ Una vez diseñada las reglas de la gramática, nos planteamos si permite realizar *análisis descendentes deterministas*. Si no es así, hay que transformar la gramática haciendo uso de predicados sintácticos.

Una vez diseñada la gramática se tiene constancia de los lexemas del lenguaje. El **diseño de los lexemas** se basa en un conjunto de recomendaciones:

- ✓ Los lexemas con formas alternativas de caracteres pueden expresarse con expresiones regulares alternativas. Por ejemplo, `BTF: (' '|'\t'|NL);`.
- ✓ Los lexemas con formas iterativas de caracteres pueden expresarse con expresiones regulares de tipo cierre. Por ejemplo, `NUMERO: (DIGITO)+;`.
- ✓ Los lexemas con partes opcionales pueden expresarse con expresiones regulares de tipo opción.
- ✓ Los lexemas con forma de rango de caracteres pueden expresarse con expresiones regulares de tipo rango. Por ejemplo, `LETRA: ('a'..'z'|'A'..'Z');`.
- ✓ Los lexemas con forma de secuencia de caracteres pueden expresarse con expresiones regulares de tipo concatenación. Por ejemplo, `INSTRUCCIONES: "INSTRUCCIONES";`
- ✓ Los lexemas con 1 carácter pueden expresarse con expresiones regulares de tipo carácter. Por ejemplo, `MAS: '+' ;`
- ✓ El uso de lexemas auxiliares facilita la legibilidad de las expresiones regulares. Por ejemplo, `VAR: (LETRA)+;` siendo `LETRA : 'a'..'z'|'A'..'Z';` un lexema auxiliar.

Ejemplo 6. Parser Antlr para el lenguaje LF.

En este punto, se propone la implementación de la primera versión del analizador Léxico-Sintáctico en ANTLR. La gramática mostrada en ejemplo 5 constituirá el núcleo principal del parser ANTLR.

```
class Anasint extends Parser;

programa      : variables instrucciones EOF
              ;
variables     : VARIABLES idents PyC
              ;
idents        : (VAR COMA) => VAR COMA idents
              | VAR
              |
              ;
instrucciones : INSTRUCCIONES (definicion|evaluacion)*
              ;
definicion    : VAR DEF expr PyC
              ;
evaluacion    : EVAL idents PyC
              ;
expr          : (expr1 MAS)   => expr1 MAS   expr
              | (expr1 MENOS) => expr1 MENOS expr
              | expr1
              ;
expr1         : (expr2 POR)   => expr2 POR   expr1
              | expr2
              ;
expr2         : NUMERO
              | MENOS expr
              | VAR
              | PA expr PC
              ;
```

Ejemplo 7. Lexer Antlr para el lenguaje LF.

De forma similar, los lexemas mostrados en ejemplo 4 constituirá el núcleo principal de la implementación del lexer ANTLR.

```
class Analex extends Lexer;

options{
      importVocab = Anasint;
}

tokens{
      VARIABLES     = "VARIABLES";
      INSTRUCCIONES = "INSTRUCCIONES";
      EVAL          = "EVAL";
      DEF           = "DEF";
}

protected NL     : "\r\n"        {newline();} ;
protected DIGITO : '0'..'9';
protected LETRA  : 'a'..'z'|'A'..'Z';

          BTF    : (' '|'\t'|NL) {$setType(Token.SKIP);} ;
          NUMERO : (DIGITO)+     {$setType(NUMBER)    ;} ;
          VAR    : (LETRA)+ ;
          PA     : '(' ;
          PC     : ')' ;
          PyC    : ';' ;
          COMA   : ',' ;
          MAS    : '+' ;
          MENOS  : '-' ;
          POR    : '*' ;
```

Ejemplo 8. Programa para testar el analizador léxico-sintáctico.

Construida la primera versión del analizador Léxico-Sintáctico, se procede a testarlo. El siguiente programa Java servirá para realizar el *testing*. La idea es configurar en Eclipse un entorno de ejecución con programas con el lenguaje LF de pruebas pasados como argumento. Los programas con el lenguaje LF de pruebas deben ser de dos tipos: *programas correctos* y *programas incorrectos*. El analizador debe aceptar los programas correctos y elevar excepciones de reconocimiento para los incorrectos.

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.FileNotFoundException;
import antlr.RecognitionException;
import antlr.TokenStreamException;

public class Principal {
  public static void main(String[] args) {
     try {
        FileInputStream f = new FileInputStream(args[0]);
        Analex  analex    = new Analex(f);
        Anasint anasint   = new Anasint(analex);
        anasint.programa();    // Análisis léxico-sintáctico
        f.close();
     }
     catch (FileNotFoundException e) { System.out.println("Error in file");             }
     catch (TokenStreamException  e) { System.out.println("Error in lexical analysis"); }
     catch (RecognitionException  e) { System.out.println("Error in parser analysis");  }
     catch (IOException           e) { System.out.println("Error in file");             }
  }
}
```

Ejemplo 9. Programa para testar el analizador léxico.

Si lo que se quiere testar es sólo el analizador léxico entonces el programa de prueba para hacerlo se muestra en el siguiente ejemplo.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import antlr.Token;
import antlr.TokenStreamException;

public class Principal {
  public static void main(String[] args) {
     try {
        FileInputStream f = new FileInputStream(args[0]);
        Analex analex = new Analex(f);
        Token t = analex.nextToken();
        while (t.getType() != Token.EOF_TYPE) {
           System.out.print("lexema:" + t.getType() + "(" + t.getText() + ")" + ", ");
           t = analex.nextToken();
        }
     }
     catch (FileNotFoundException e) { System.out.println("Error in file");             }
     catch (TokenStreamException  e) { System.out.println("Error in lexical analysis"); }
  }
}
```

Una vez testada la primera versión del analizador, se anota la gramática para que sea capaz de construir árboles de sintaxis abstracta (asa). Para anotar la gramática, primero hay que tener claro cuál es la estructura del árbol.

Ejemplo 11. Parser Antlr para el lenguaje LF con la capacidad de construir árboles ASAs.

```
class Anasint extends Parser;

options{
      buildAST = true; // activar la síntesis de asas durante el análisis.
}

tokens{
      PROGRAMA;
}

programa      : variables instrucciones EOF!
                {#programa = #(#[PROGRAMA, "PROGRAMA"], ##);}
              ;
variables!    : a:VARIABLES b:idents PyC {#variables = #(#a, #b);}
              ;
idents        : (VAR COMA) => VAR COMA! idents
              | VAR
              |
              ;
instrucciones : INSTRUCCIONES^ (definicion|evaluacion)*
              ;
definicion    : VAR DEF^ expr PyC!
              ;
evaluacion    : EVAL^ idents PyC!
              ;
expr          : (expr1 MAS)   => expr1 MAS^ expr
              | (expr1 MENOS) => expr1 MENOS^ expr
              | expr1
              ;
expr1         : (expr2 POR)   => expr2 POR^ expr1
              | expr2
              ;
expr2         : NUMERO
              | MENOS^ expr
              | VAR
              | PA! expr PC!
              ;
```

Ejemplo 12. Programa para testar el analizador léxico-sintáctico.

Construida la segunda versión del analizador léxico-sintáctico, se procede a testarlo. El siguiente programa Java servirá para realizar el *testing*.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;

public class Principal {
  public static void main(String[] args) {
     try {
        FileInputStream f = new FileInputStream(args[0]);
        Analex analex = new Analex(f);
        Anasint anasint = new Anasint(analex);
        anasint.programa();  // Iniciar el análisis léxico-sintáctico

        AST a = anasint.getAST();
        ASTFrame af = new ASTFrame(args[0], a);
        af.setVisible(true); // Mostrar el árbol asa por pantalla
     }
     catch (FileNotFoundException e) { System.out.println("Error in file"); }
     catch (RecognitionException  e) { System.out.println("Error in parser analysis"); }
     catch (TokenStreamException  e) { System.out.println("Error in lexical analysis"); }
  }
}
```

3 Análisis Semántico.

3.1 Objetivo.

Dar respuesta a las siguientes preguntas:

- ✓ ¿Qué es el análisis semántico de un lenguaje?
- ✓ ¿Cómo se diseña y se construye un analizador semántico?

3.2 El problema del cálculo de tipo de una expresión.

Supongamos un lenguaje de expresiones enteras y booleanas con la siguiente gramática:

```
expr    :( expr1 op_log ) => expr1 op_log expr
        |( NO ) => NO expr1
        | expr1
        ;
op_log  : Y | O
        ;
expr1   :( expr2 op_rel ) => expr2 op_rel expr2
        | expr2
        ;
op_rel  : MAYOR| MENOR| IGUAL
        ;
expr2   :( expr3 op_arit ) => expr3 op_arit expr2
        | expr3
        ;
op_arit : MAS | MENOS | POR| DIV
        ;
expr3   : IDENT| NUMERO | TRUE | FALSE | PA expr PC
        ;
```

Calcular el tipo de una expresión significa decidir si la expresión es entera o booleana. Por ejemplo, `2+5` es una expresión entera y, `cierto` y `falso`, es una expresión booleana. Sin embargo, hay expresiones para las que no es posible decidir el tipo basándonos sólo en la sintaxis. Por ejemplo, no podemos calcular el tipo de expresiones tales como `(x+3)` o `(z y v <= 5)` a menos que conozcamos el tipo de sus variables. Para superar esta limitación disponemos de las *gramáticas atribuidas*.

3.3 Gramáticas atribuidas.

Una gramática atribuida es una gramática con variables. Los atributos (o variables) pueden ser **sintetizados** o **heredados**. Si el **valor del atributo se calcula** en una regla, se dice que el atributo es **sintetizado**. Si la regla no produce **el valor del atributo sino que sólo lo utiliza** entonces se dice que el atributo es **heredado** en dicha regla.

El tipo de una expresión se puede calcular recursivamente:

- ✓ Caso base: Las constantes enteras son de tipo entero, las constantes booleanas son de tipo booleano, el tipo de una variable depende de su declaración.
- ✓ Caso recursivo: Una expresión con un operador aritmético y con sub-expresiones enteras es una expresión *entera*, una expresión con un operador booleano y con sub-expresiones booleanas es una expresión *booleana*. Una expresión que mezcle sub-expresiones de distinto tipo u operadores de distinto tipo es una expresión de tipo *indefinido*.

El siguiente ejemplo muestra una gramática atribuida ANTLR para resolver el problema del cálculo de tipo de una expresión. El tipo de la expresión se codifica con el valor de un atributo sintetizado llamado `t`. El valor 0 de este atributo significa tipo *indefinido*, el 1 significa tipo *entero* y 2 significa tipo *booleano*.

```
header{ import java.util.*; }

class Anasint extends Parser;
{
      //Tabla con el tipo de cada variable.
      //tipo = 1 => tipo de la variable es entero
      //tipo = 2 => tipo de la variable es booleano
      Hashtable<String,Integer> vars = new Hashtable<String, Integer>();
}
// Gramática atribuída.
// t es un atributo sintetizado. Valores de t: 0: indefinido, 1: entero, 2: booleano
sentencia returns [int t=0;]       : variables t=expr
                                   ;
variables                          : decl_vars PyC
                                   ;
// v es un atributo heredado.
tipo [String v]  : ENTERO  { vars.put(v, new Integer(1)); }
                 | BOOLEAN { vars.put(v, new Integer(2)); }
                 ;

decl_vars        : (IDENT tipo[new String("")] COMA) => a:IDENT tipo[a.getText()] COMA decl_vars
                 | b:IDENT tipo[b.getText()]
                 |
                 ;

// t es un atributo sintetizado. Valores de t: 0: indefinido, 1: entero, 2: booleano
expr returns [int t=0;] {int a, b;}  :
                  (expr1 (Y|O)) => a=expr1 (Y b=expr | O b=expr)
                      { if (a==b & a==2) t=a; else t=0; }
                 |( NO ) => NO a=expr
                      { if (a==2) t=a; else t=0; }
                 | t=expr1
                 ;

expr1 returns [int t=0;] {int a, b;}  :
                (expr2 (MAYOR | MENOR | IGUAL)) => a=expr2 (MAYOR b=expr2 | MENOR b=expr2 | IGUAL b=expr2)
                      { if (a==b & a==1) t=2; else t=0; }
                 | t=expr2
                 ;

expr2 returns [int t=0;] {int a, b;}  :
                  (expr3 (MAS | MENOS | POR ) ) => a=expr3 (MAS b=expr2 | MENOS b=expr2 | POR b=expr2)
                      { if (a==b & b==1) t=a; else t=0; }
                 | ( expr3 DIV) => a=expr3 (DIV b=expr2)
                      { if (a==b & b==1) t=a; else t=0; }
                 | t=expr3
                 ;

expr3 returns [int t=0;]                :
                              // a es una etiqueta para acceder a lexema IDENT.
                              // vars permite acceder a información contextual.
                   a:IDENT  { t = vars.get(a.getText()); }
                 | b:NUMERO { t = 1; }
                 | TRUE     { t = 2; }
                 | FALSE    { t = 2; }
                 | PA t=expr PC
                 ;
```

3.4 ¿Qué es el análisis semántico?

La definición de un lenguaje no se basa sólo en aspectos sintácticos. Por ejemplo, una restricción usual en los lenguajes de programación es que toda variable en un programa debe estar declarada o que el tipo de una expresión debe estar definido. Este tipo de restricciones no se pueden implementar con gramáticas independientes de contexto. Para resolverlos, las gramáticas independientes de contexto se extienden con *atributos* (gramáticas atribuidas).

Supongamos un lenguaje llamado LPROP diseñado para expresar interpretaciones y fórmulas proposicionales. La proposición en LPROP se expresa con una letra minúscula. Los operadores lógicos en LPROP son la equivalencia (<=>), la implicación (=>), la conjunción (&), la disyunción (|) y la negación (-). Por interpretación en LPROP se entiende un conjunto de pares (proposición, valor de verdad). En LPROP, las proposiciones pueden ser ciertas (T) o falsas (F).

```
class Anasint extends Parser;
options{
      buildAST = true;
}
tokens{
      ENTRADA;
}
entrada        : interpretacion formula EOF!
               {#entrada = #(#[ENTRADA,"ENTRADA"],##);}
               ;
interpretacion : LLA^ asignaciones LLC!
               ;
asignaciones   : (asignacion COMA) => asignacion COMA! asignaciones
               | asignacion
               ;
asignacion     : PA! PROP^ COMA! (TRUE|FALSE) PC!
               ;
formula        : (formula1 IMPL) => formula1 IMPL^ formula
               | formula1
               ;
formula1       : (formula2 EQUIV) => formula2 EQUIV^ formula1
               | formula2
               ;
formula2       : (formula3 OR) => formula3 OR^ formula2
               | formula3
               ;
formula3       : (formula4 AND) => formula4 AND^ formula3
               | formula4
               ;
formula4       : (NEG) => NEG^ formula5
               | formula5
               ;
formula5       : PROP
               | TRUE
               | FALSE
               | PA! formula PC!
               ;
```

Un ejemplo de sentencia en el lenguaje LPROP es el siguiente:

`{(p,F),(q,T),(r,F)}` `p => (q | r & v)`

La "forma" de esta sentencia puede ser reconocida con la gramática del ejemplo 2. Sin embargo, la proposición v no tiene interpretación y por ello la formula `p => (q | r & v)` no tiene valor de verdad. ¿La sentencia `p => (q | r & v)` pertenece al lenguaje LPROP?

Si por pertenencia al lenguaje LPROP se entiende que `p => (q | r & v)` tiene que tener un valor de verdad definido entonces la respuesta sería NO. Por tanto, **el reconocimiento de las sentencias de un lenguaje puede implicar algo más que un simple reconocimiento sintáctico**.

La definición de un lenguaje o, equivalentemente, el criterio para reconocer sus sentencias dependerá del propósito del propio lenguaje. Por ejemplo, si el propósito de LPROP es evaluar sentencias entonces necesitaríamos imponer restricciones adicionales. Una posible restricción sería la de imponer una interpretación para cada proposición en la fórmula o reforzar esta restricción con la de aceptar sólo fórmulas para la que existe una interpretación que la haga cierta. Responsabilizarse de estas restricciones adicionales a la sintaxis es el propósito del analizador semántico.

3.5 ¿Cómo se diseña y se construye un analizador semántico?

Dada una gramática del lenguaje, se recomienda los siguientes pasos para diseñar un analizador semántico:

1) Especificar las restricciones semánticas del lenguaje.
2) Diseñar una gramática atribuida que cubra las restricciones semánticas especificadas en (1).
3) Implementar un parser ANTLR para la gramática atribuida, diseñada en (2) y testarlo.

Especificar las restricciones semánticas del lenguaje. Si una sentencia no supera estas restricciones, no puede pertenecer al lenguaje. Por ejemplo, una sentencia es semánticamente correcta en LPROP si lo es sintácticamente y su fórmula tiene valor de verdad para la interpretación dada.

Decidir sobre la corrección sintáctica de una sentencia LPROP se consigue con la gramática propuesta en el ejemplo 2. Dicha gramática genera un árbol de sintaxis abstracta para mostrar en pantalla la estructura de la sentencia reconocida.

Diseñar una gramática atribuida que cubra las restricciones semánticas. Hay que decidir cuáles son los atributos necesarios para resolver el problema. Por ejemplo, para decidir en LPROP si una fórmula tiene valor de verdad según la interpretación dada en su sentencia es suficiente con comprobar que toda proposición de la fórmula tiene asignado un valor de verdad en la interpretación. En una gramática atribuida, el atributo sería el conjunto de las proposiciones presentes en la interpretación. A continuación mostramos dicha gramática.

Implementar un parser ANTLR para la gramática atribuida y testarlo.

Ejemplo 3. Gramática atribuida para el lenguaje LPROP.

```
{ Set<String> s = new HashSet<String>(); }     // Atributo

entrada        : #(ENTRADA interpretacion formula)
               ;
interpretacion : #(LLA (asignacion)*)
               ;
asignacion     : #(a:PROP { s.add(a.getText()); } (TRUE | FALSE))
               ;
formula        : #(EQUIV formula formula)
               | #(IMPL  formula formula)
               | #(OR    formula formula)
               | #(AND   formula formula)
               | #(NEG   formula)
               | p:PROP { if (s.contains(p.getText()) == false)         // s es un atributo heredado.
                              System.out.println("Error semántico");
                        }
               | TRUE
               | FALSE
               ;
```

A continuación mostramos el parser para la gramática atribuida del ejemplo 3.

Ejemplo 4. Tree-parser para el lenguaje LPROP.

```
header{
      import java.util.*;
}

class Anasint2 extends TreeParser;    //Parser de asas

options{
      importVocab = Anasint;
}

{ Set<String> s = new HashSet<String>(); }

entrada        : #(ENTRADA interpretacion formula)
               ;
interpretacion : #(LLA (asignacion)*)
               ;
asignacion     : #(a:PROP { s.add(a.getText()); } (TRUE | FALSE))
               ;
formula        : #(EQUIV formula formula)
               | #(IMPL  formula formula)
               | #(OR    formula formula)
               | #(AND   formula formula)
               | #(NEG   formula)
               | p:PROP { if (s.contains(p.getText()) == false)
                              System.out.println("Error semántico");
                        }
               | TRUE
               | FALSE
               ;
```

El parser propuesto en el ejemplo 4 es un tree-parser. Los asas generados por el parser del ejemplo 2 son consumidos por el tree-parser del ejemplo 4.

Un **tree-parser** es un parser para reconocer árboles de sintaxis abstracta. Las reglas del tree-parser se diferencian de las del parser convencional porque en la parte derecha tienen un patrón asa.

Por ejemplo, la regla `entrada: #(ENTRADA interpretacion formula)` tiene un asa en la parte derecha con nodo raíz ENTRADA y dos sub-asas (*interpretacion* y *formula*).

Los operadores cierre y alternativa se pueden utilizar en los patrones asa. Por ejemplo, el patrón `#(LLA (asignacion)*)` representa un asa con nodo raíz `LLA` y una secuencia de asas de tipo *asignacion*.

Las etiquetas son el recurso ANTLR para acceder a los nodos del asa. Por ejemplo, el patrón `#(a:PROP { s.add(a.getText()); } (TRUE | FALSE))` define una etiqueta a para acceder al nodo `PROP`.

4 Interpretación, compilación.

4.1 Objetivo.

Dar respuesta a las siguientes preguntas:

- ✓ ¿Qué es la interpretación de un lenguaje?
- ✓ ¿Qué es la compilación de un lenguaje?
- ✓ ¿Cómo se diseña y se construye un intérprete?
- ✓ ¿Cómo se diseña y se construye un compilador?

El reconocimiento del lenguaje ya no es el problema. Se supone que el análisis léxico-sintáctico y el análisis semántico han resuelto el problema del reconocimiento. El problema ahora es ¿qué hacer con las sentencias del lenguaje?

4.2 ¿Qué es la interpretación de un lenguaje?

Interpretar un lenguaje significa ejecutar directamente sus sentencias.

Supongamos el lenguaje LF definido en el Tema 1. Para interpretar el lenguaje LF tenemos que definir qué es la ejecución de un programa en lenguaje LF. Sabemos que LF es un lenguaje de programación secuencial. Las instrucciones se ejecutan una a una desde el comienzo del programa hasta el final. Además, tiene la capacidad de expresar variables enteras y dos tipos de instrucciones: (a) Definición de variables con expresión entera (DEF) y (b) Evaluación de variables (EVAL). No se acepta el uso de variables sin declarar. La declaración de variables asocia valor 0 por defecto a éstas.

Ejecutar una instrucción DEF significa vincular una expresión a una variable para una eventual evaluación. Toda definición de una variable solapará sus definiciones anteriores en el programa. Toda variable definida sobre sí misma se le asocia un valor indefinido.

Ejecutar una instrucción EVAL *x* significa evaluar el valor de la expresión asociada a *x* mostrando por pantalla el resultado.

A continuación mostramos un programa en lenguaje LF y su correspondiente interpretación.

Ejemplo 1: Programa en LF

```
VARIABLES x, y, z, a, b;
INSTRUCCIONES
   a DEF -1;
   b DEF (a+1);
   EVAL b;
   a DEF 2;
   EVAL b;
   b DEF b + 1;
   z DEF 2*y;
   EVAL z;
   EVAL b;
```

Ejemplo 2: Interpretación del Programa en LF

```
b --> 0
b --> 3
z --> 0
b --> INDEFINIDO
```

4.3 ¿Qué es la compilación de un lenguaje?

Compilar un lenguaje significa ser capaz de compilar cada sentencia de dicho lenguaje. Compilar una sentencia es traducirla a otro lenguaje preservando, en lo posible, el significado de la sentencia. A diferencia de la interpretación, la compilación involucra dos lenguajes: el **lenguaje fuente** y el **lenguaje destino**. El lenguaje *fuente* suele ser un lenguaje expresivo para el que no se dispone de intérprete. El lenguaje *destino* suele ser un lenguaje menos expresivo pero que tiene un intérprete. Un ejemplo de lenguaje compilado es Java. El lenguaje destino es un lenguaje ensamblador interpretable, después del ensamblado (CodeByte) en la llamada máquina virtual de Java.

La construcción de un compilador obliga a especificar correspondencias entre los recursos expresivos del lenguaje fuente y los recursos expresivos del lenguaje destino. Las principales correspondencias en nuestro problema son: (1) el programa en lenguaje LF se corresponde con una clase Java con m*a*in, (2) la instrucción EVAL se corresponde con la definición de un método (sin parámetros) y la correspondiente llamada a dicho método en el *main* para presentar por pantalla el resultado, (3) la instrucción DEF se corresponde con la definición de un método (sin parámetros).

Ejemplo 3: Compilación en Java de un programa en lenguaje LF en ejemplo 1

```
import java.io.*;

public class _Programa {

     private static int a1() {
        return -1;
     }

     private static int b1() {
        return a1() + 1;
     }

     private static int a2() {
        return 2;
     }

     private static int b2() {
        return a2() + 1;
     }

     private static int y3() {
        return 0;
     }

     private static int z3() {
        return 2 * y3();
     }

     private static int b4() {
        return b4() + 1;
     }

     public static void main(String[] args) {
        System.out.println(b1());
        System.out.println(b2());
        System.out.println(z3());
        System.out.println(b4());
     }
}
```

4.4 ¿Cómo se diseña y se construye un intérprete?

No hay una respuesta general a esta pregunta. La construcción del intérprete es fuertemente dependiente de la semántica del lenguaje. El paso clave del diseño es clarificar la semántica del lenguaje. Una forma de clarificar esta semántica es poner sentencias de ejemplos y decir qué significan (ver ejemplos 1 y 2). Por ejemplo, la ejecución mostrada en el ejemplo 2 facilita la comprensión del lenguaje LF. Las dos primeras evaluaciones de la variable *b* producen resultados diferentes sin cambiar su definición. Esto se debe a que las expresiones se asocian a las variables justo en el momento de su evaluación no en el momento de su definición (*late binding*). En nuestro programa de ejemplo (ver ejemplo 1), la variable *a* modifica su definición a lo largo del programa y la definición de la variable *b* depende de la definición de la variable *a*. Esto hace que la evaluación de la variable *b* pueda ser distinta sin modificar su definición.

Una vez comprendida la semántica del lenguaje LF, diseñamos un intérprete. Para evaluar una variable es necesario almacenar su definición. Dado que la evaluación no se produce hasta alcanzar una instrucción EVAL, estas definiciones se almacenan en una tabla para recuperarlas en el momento necesario (atributo heredado y sintetizado).

A continuación se muestra un intérprete para el lenguaje LF en forma de *tree-parser*.

Ejemplo 4: Intérprete para el lenguaje LF.

```
header{
	import java.util.*;
	import antlr.*;
}

class Anasint extends TreeParser;

options
{
	importVocab = Anasint;
}

tokens{
	INDEF;
}

{
	Hashtable<String, AST> variables = new Hashtable<String, AST>();
	ASTFactory factory = new ASTFactory();

	public void imprimir(){
		System.out.println(variables);
	}

	public void declarar(String var){
		AST n = nodo("0", NUMERO);
		variables.put(var, n);
	}

	public AST nodo(String texto, int tipo){
		AST resultado = new CommonAST();
		resultado.setType(tipo);
		resultado.setText(texto);
		return resultado;
	}

	public int convertir(AST numero){
		return (new Integer(numero.getText())).intValue();
	}

	public void almacenar(String var, AST expr){
		variables.put(var, expr);
	}
```

```
	public boolean ocurrencia(String var, AST expr){
		switch(expr.getType()){
			case NUMERO: return false;
			case VAR   : return expr.getText().equals(var);
			case MAS   :
			case POR   : return ocurrencia(var, expr.getFirstChild())
					|| ocurrencia(var, expr.getFirstChild().getNextSibling());
			case MENOS : if (expr.getFirstChild().getNextSibling()==null)
					return ocurrencia(var, expr.getFirstChild());
				else
					return ocurrencia(var, expr.getFirstChild())
					    || ocurrencia(var, expr.getFirstChild().getNextSibling());
			}
		return false;
	}



	public AST evaluar(AST expr){
		AST resultado = null;
		AST a,b;

		switch(expr.getType()){

		  case NUMERO: resultado = factory.dupTree(expr);
				break;

		  case VAR  : resultado = evaluar(variables.get(expr.getText()));
				break;

		  case MAS  : a = evaluar(expr.getFirstChild());
				b = evaluar(expr.getFirstChild().getNextSibling());
				if (a.getType()==INDEF || b.getType()==INDEF)
				    resultado = nodo("INDEF",INDEF);
				else
				    resultado = nodo((new Integer(convertir(a) + convertir(b))).toString(),NUMERO);
				break;

		  case POR  : a = evaluar(expr.getFirstChild());
				b = evaluar(expr.getFirstChild().getNextSibling());
				if (a.getType()==INDEF || b.getType()==INDEF)
				    resultado = nodo("INDEF",INDEF);
				else
				    resultado = nodo((new Integer(convertir(a) * convertir(b))).toString(),NUMERO);
				break;

		  case MENOS: if (expr.getFirstChild().getNextSibling()==null){
				a = evaluar(expr.getFirstChild());
				if (a.getType()==INDEF)
				    resultado = nodo("INDEF",INDEF);
				else
				    resultado = nodo((new Integer(-1*convertir(a))).toString(),NUMERO);
				}
				else{
				    a = evaluar(expr.getFirstChild());
				    b = evaluar(expr.getFirstChild().getNextSibling());
				    if (a.getType()==INDEF || b.getType()==INDEF)
				      resultado = nodo("INDEF",INDEF);
				    else
				      resultado = nodo((new Integer(convertir(a) - convertir(b))).toString(),NUMERO);
				}
				break;

		  case INDEF: resultado = factory.dupTree(expr);
				break;

		}
		return resultado;
	}
}
```

```
programa      : #(PROGRAMA variables instrucciones)
              ;
variables     : #(VARIABLES (v:VAR {declarar(#v.getText());} )*)
              ;
instrucciones : #(INSTRUCCIONES (definicion|evaluacion)*)
              ;
definicion    : #(DEF v:VAR e:expr)
                { if (!ocurrencia(#v.getText(),#e))
                     almacenar(#v.getText(), #e);
                  else{
                        AST n = new CommonAST();
                        n.setType(INDEF);
                        n.setText("INDEF");
                        almacenar(#v.getText(), n);
                       }
                }
              ;
evaluacion    : #(EVAL v:VAR)
                { System.out.println(#v.getText() + " --> " + evaluar(#v).toStringTree()); }
              ;
expr          : #(MAS expr expr)
              | #(POR expr expr)
              | NUMERO
              | VAR
              | (#(MENOS expr expr)) => #(MENOS expr expr)
              | #(MENOS expr)
              ;
```

4.5 ¿Cómo se diseña y se construye un compilador?

Tampoco hay respuesta general a esta pregunta. La construcción del compilador es fuertemente dependiente de los lenguajes involucrados (lenguaje *fuente* y lenguaje *destino*). Los pasos claves del diseño son (1) Clarificar las semánticas del lenguaje fuente y del lenguaje destino y (2) Clarificar las correspondencias entre ambos lenguajes.

Tal y como ya hemos visto, el enlace dinámico de expresiones a variables constituye el principal problema a la hora de traducir el lenguaje LF a Java (*late binding*). Las variables Java son imperativas: el enlace del valor a la variable se produce al ejecutar una asignación y se mantiene hasta que otra asignación la solape. Por tanto, no podemos usar asignaciones imperativas para implementar definiciones del lenguaje LF.

Hemos visto que las principales correspondencias entre el lenguaje LF y el lenguaje Java son:

1) el programa en lenguaje LF se corresponde con una clase Java con *main*.
2) la instrucción EVAL se corresponde con la definición de un método (sin parámetros) y la correspondiente llamada a dicho método en el *main* para presentar por pantalla el resultado.
3) la instrucción DEF se corresponde con la definición de un método (sin parámetros).

Una forma de resolver el problema del enlace variable/expresión es codificar cada evaluación con la declaración de un método. El cuerpo del método implementará el enlace dinámico. Por ejemplo, los métodos *b1*() y *b2*() en el ejemplo 3 implementan las dos primeras evaluaciones de la variable *b* en el programa en lenguaje LF del ejemplo 2. Cada evaluación además generará una llamada al método correspondiente en el *main*.

El siguiente ejemplo muestra el compilador del lenguaje LF a lenguaje Java.

Ejemplo 4: Compilador del lenguaje LF.

```
header{
	import java.util.*;
	import antlr.*;
	import java.io.*;
}

class Anasint extends TreeParser; //Compilador

options
{
	importVocab = Anasint;
}

{

//////////////////

	FileWriter fichero;

	private void open_file(){
		try{
			fichero = new FileWriter("_Programa.java");
		}catch(IOException e)
			{System.out.println("open_file (exception): " + e.toString());}
	}

	private void close_file(){
		try{
			fichero.close();
		}catch(IOException e)
			{System.out.println("close_file (exception): " + e.toString());}
	}


//////////////////

	int espacios = 0;

	private void gencode_espacios(){
		try{
			for (int i = 1; i<=espacios; i++)
				fichero.write(" ");
		}catch(IOException e)
			{System.out.println("gencode_espacios (exception): " + e.toString());}
	}


//////////////////

	ASTFactory factory = new ASTFactory();
	Hashtable<String, AST> vars = new Hashtable<String, AST>();
	Hashtable<String, AST> vars2 = new Hashtable<String, AST>();
	List<String> evals_pend = new LinkedList<String>();
	Integer num_evals = new Integer(0);

	public void declarar_variable(String var){
		AST nodo = new CommonAST();
		nodo.setType(NUMERO);
		nodo.setText("0");
		vars.put(var, nodo);
	}

	public void definir_variable(String var, AST expr){
		vars.put(var, expr);
	}
```

```
public void evaluar_variable(String var){
	Integer i;
	String aux = new String(var);
	num_evals++;
	Enumeration<String> v = vars.keys();
	while (v.hasMoreElements()){
		String key = v.nextElement();
		if (cierre(var).contains(key)){ //Se calcula el cierre de var.
			// se calcula una variable indexada para var.
			aux = key+num_evals.toString();
			// se sustituye en la expresión de la definición cada
			// variable original por una nueva variable indexada
			// por el número de evaluaciones
			vars2.put(aux,sustitucion_expr(vars.get(key)));
		}
	}
	try{ // A cada variable presente en la definición de la variable
	     // evaluada (las incluidas en vars2) se genera un método.
		v = vars2.keys();
		while (v.hasMoreElements()){
			String key = v.nextElement();
			gencode_espacios();
			fichero.write("private static int " + key + "(){\n");
			espacios++;
			gencode_espacios();
			fichero.write("return " + gencode_exp(vars2.get(key)) + ";\n");
			espacios--;
			gencode_espacios();
			fichero.write("}\n");
		}
	}catch(IOException e){}
	vars2.clear();
	evals_pend.add(var + num_evals.toString()); //Queda pendiente generar
	// llamada en el main.
}


public Set<String> cierre(String var){
	Set<String> resultado = new HashSet<String>();
	int size_ant = 0;
	int size_act = 0;
	resultado.add(var);
	size_act = resultado.size();
	while (size_ant != size_act){
		resultado.addAll(calcular_variables(vars.get(var)));
		size_ant = size_act;
		size_act = resultado.size();
	}
	return resultado;
}


public Set<String> calcular_variables(AST expr){
	Set<String> aux = new HashSet<String>();
	switch(expr.getType()){
		case NUMERO:
			break;
		case VAR:
				aux.add(expr.getText());
			break;
		case MAS:
		case POR:
				aux.addAll(calcular_variables(expr.getFirstChild()));
				aux.addAll(calcular_variables(expr.getFirstChild().getNextSibling()));
			break;
		case MENOS:
				aux.addAll(calcular_variables(expr.getFirstChild()));
				if (expr.getFirstChild().getNextSibling()!= null)
				    aux.addAll(calcular_variables(expr.getFirstChild().getNextSibling()));
			break;
		default: return null;
	}
	return aux;
}
```

```
public AST sustitucion_expr(AST expr){
      AST nodo;
      String aux;
      switch(expr.getType()){
            case NUMERO: return factory.dupTree(expr);
            case VAR   : aux = expr.getText();
                         nodo = new CommonAST();
                         aux = aux + num_evals.toString();
                         nodo.setType(VAR);
                         nodo.setText(aux);
                         return nodo;
            case MAS   : nodo = new CommonAST();
                         nodo.setType(MAS);
                         nodo.setText("+");
                         nodo.setFirstChild(sustitucion_expr(expr.getFirstChild()));
                         nodo.getFirstChild().setNextSibling(
                         sustitucion_expr(expr.getFirstChild().getNextSibling()));
                         return nodo;
            case POR   : nodo = new CommonAST();
                         nodo.setType(POR);
                         nodo.setText("*");
                         nodo.setFirstChild(sustitucion_expr(expr.getFirstChild()));
                         nodo.getFirstChild().setNextSibling(
                         sustitucion_expr(expr.getFirstChild().getNextSibling()));
                         return nodo;
            case MENOS : nodo = new CommonAST();
                         nodo.setType(MENOS);
                         nodo.setText("-");
                         nodo.setFirstChild(
                         sustitucion_expr(expr.getFirstChild()));
                         if (expr.getFirstChild().getNextSibling() != null)
                              nodo.getFirstChild().setNextSibling(
                                    sustitucion_expr(expr.getFirstChild().getNextSibling()));
                         return nodo;
            default    : return null;
      }
}


private void gencode_begin_class(){
      try{
            gencode_espacios();
            fichero.write("import java.io.*;\n");
            gencode_espacios();
            fichero.write("public class _Programa"+"\n");
            gencode_espacios();
            fichero.write("{\n");
            espacios++;
      }catch(IOException e){}
}


private void gencode_main(){
      try{
            gencode_espacios();
            fichero.write("public static void main(String[] args) {\n");
            espacios++;
            Iterator<String> it = evals_pend.listIterator();
            while (it.hasNext()){
                  gencode_espacios();
                  fichero.write("System.out.println(" + it.next() + "());\n");
            }
            espacios--;
            gencode_espacios();
            fichero.write("}\n");
      }catch(IOException e){}
}


private void gencode_end_class(){
      try{
            espacios--;
            gencode_espacios();
            fichero.write("}");
      }catch(IOException e){}
}
```

```
      public String gencode_exp(AST expr){
            switch(expr.getType()){
                  case NUMERO: return expr.getText();
                  case VAR   : return expr.getText()+"()";
                  case MAS   : return gencode_exp(expr.getFirstChild()) + "+"
                                    +gencode_exp(expr.getFirstChild().getNextSibling());
                  case POR   : return gencode_exp(expr.getFirstChild()) + "*"
                                    +gencode_exp(expr.getFirstChild().getNextSibling());
                  case MENOS : if (expr.getFirstChild().getNextSibling() != null)
                                    return gencode_exp(expr.getFirstChild()) + "-"
                                          +gencode_exp(expr.getFirstChild().getNextSibling());
                               else
                                    return "-" + gencode_exp(expr.getFirstChild());
                  default: return null;
            }
      }
}
```

```
programa      : {open_file(); gencode_begin_class();}
                #(PROGRAMA variables instrucciones)
                {gencode_main(); gencode_end_class(); close_file();}
              ;

variables     : #(VARIABLES (v:VAR {declarar_variable(v.getText());} )*)
              ;

instrucciones : #(INSTRUCCIONES (definicion|evaluacion)*)
              ;

definicion    : #(DEF v:VAR e:expr)
                {definir_variable(v.getText(),e);}
              ;

evaluacion    : #(EVAL v:VAR)
                {evaluar_variable(v.getText());}
              ;

expr          : #(MAS expr expr)
              | #(POR expr expr)
              | NUMERO
              | VAR
              | (#(MENOS expr expr)) => #(MENOS expr expr)
              | #(MENOS expr)
              ;
```

5 Laboratorio.

5.1 Análisis Léxico-Sintáctico.

Objetivo: Problemas para consolidar el diseño y construcción de analizadores léxico-sintácticos.

5.1.1 EXPL

Supongamos un lenguaje de expresiones lógicas llamado EXPL con los siguientes recursos expresivos: variables, constantes booleanas y enteras, predicados, funciones, negación, conjunción, disyunción, implicación y cuantificadores universal y existencial. Los predicados y funciones son símbolos prefijos con las excepciones de los predicados relacionales igual, mayor que, menor que, menor o igual que, mayor o igual que y distinto y de las funciones suma, resta, producto y división que son infijos.

Algunas de las expresiones de EXPL son:

```
1. (resultado == falso Y i<=numeroElementos(v))
2. PARATODO x,y: (p(x) Y NO r(x,y) => z == cierto)
3. EXISTE z: (z+2 == 15)
```

En la sentencia 1, *resultado* es una variable, *falso* es una constante, *Y* es la conjunción lógica, *i* es una variable y *numeroElementos* es una función.

En la sentencia 2, *PARATODO* es el cuantificador universal, *x,y* son las variables cuantificadas, *NO* es la negación lógica, *=>* es la implicación lógica, *z* es una variable y *cierto* es una constante.

En la sentencia 3, *EXISTE* es el cuantificador universal, *z* es la variable cuantificada, *+* representa la suma, *2* y *15* son constantes enteras.

SE PIDE: Analizador léxico-sintáctico para EXPL con capacidad para construir árboles de sintaxis abstracta. Testar el analizador sobre un conjunto de expresiones pertenecientes a EXPL y sobre un conjunto de expresiones no pertenecientes a EXPL.

Tests:

Resultado del test realizado con sentencias pertenecientes al lenguaje:

```
1. (resultado == falso Y i <= numeroElementos(v))      // OK
2. PARATODO x,y: (p(x) Y NO r(x,y) => z==cierto)       // OK
3. EXISTE z:(z+2 == 15)                                // OK
```

Resultado del test realizado con sentencias no pertenecientes al lenguaje:

```
1. (resultado == falso NO i <= numeroElementos(v))     // ERROR SINTÁCTICO.
2. PARATODO x,y p(x) Y NO r(x,y) => z==cierto)         // ERROR SINTÁCTICO.
3. EXISTE z:(z+2 = 15)                                 // ERROR LÉXICO.
```

Solución:

Analizador léxico:

```
class Analex extends Lexer;

options{
        importVocab = Anasint;
        k = 2;
}

tokens{
        Y        = "Y";
        O        = "O";
        NO       = "NO";
        PARATODO = "PARATODO";
        EXISTE   = "EXISTE";
        FALSO    = "falso";
        CIERTO   = "cierto";
}

protected NL        : "\r\n" {newline();};

protected LETRA     : 'a'..'z'|'A'..'Z';

protected DIGITO    : '0'..'9';

          BTF       : (' '|'\t'|NL) {$setType(Token.SKIP);};
          IDENT     : LETRA(LETRA|DIGITO)*;
          NUMERO    : (DIGITO)+;

          IMPLICA   : "=>";
          MAYOR     : '>' ;
          MENOR     : '<' ;
          MAYORIGUAL: ">=";
          MENORIGUAL: "<=";
          IGUAL     : "==";
          DISTINTO  : "!=";

          MAS       : '+' ;
          MENOS     : '-' ;
          POR       : '*' ;
          DIVISION  : '/' ;

          PA        : '(' ;
          PC        : ')' ;
          DP        : ':' ;
          COMA      : ',' ;
```

Analizador sintáctico:

```
class Anasint extends Parser;

options{
        buildAST = true;
}

expresion               : exprBooleana
                        ;

exprBooleana            : ( subexprBooleana IMPLICA ) => subexprBooleana IMPLICA^ exprBooleana
                        | subexprBooleana
                        ;

subexprBooleana         : ( subexprBooleana2 ( Y | O ) ) => subexprBooleana2 ( Y^ | O^ ) subexprBooleana
                        | subexprBooleana2
                        ;

subexprBooleana2        : PARATODO^ variablesCuantificadas DP! PA! exprBooleana PC!
                        | EXISTE^   variablesCuantificadas DP! PA! exprBooleana PC!
                        | ( NO PA ) => NO^ PA! exprBooleana PC!
                        | NO^ atomo
                        | (PA exprBooleana) => PA! exprBooleana PC!
                        | atomo
                        ;

variablesCuantificadas : (IDENT COMA) => IDENT COMA! variablesCuantificadas
                        | IDENT
                        ;

atomo                   :( termino ( MAYOR  |MENOR  |MAYORIGUAL  |MENORIGUAL  |IGUAL  |DISTINTO  )  ) =>
                           termino ( MAYOR^ |MENOR^ |MAYORIGUAL^ |MENORIGUAL^ |IGUAL^ |DISTINTO^ ) termino
                        | IDENT^ PA! terminos PC!
                        ;

terminos                : ( termino COMA ) => termino COMA! terminos
                        | termino
                        |
                        ;

termino_base            : ( IDENT PA ) => IDENT^ PA! terminos PC!
                        | IDENT
                        | NUMERO
                        | FALSO
                        | CIERTO
                        ;

termino                 : termino_base ((MAS^|MENOS^|POR^|DIVISION^) termino)?
                        | PA! termino PC!
                        ;
```

5.1.2 LEXCHANGE

Supongamos un lenguaje llamado LEXCHANGE para programar transferencias de datos entre dos bases de datos relacionales. Un programa típico en LEXCHANGE incluye: (a) Esquema de datos fuente, (b) Datos fuente, (c) Esquema de datos destino y (d), un conjunto de restricciones especificando la transferencia de datos entre el fuente y el destino. Todos los datos son de tipo cadena de caracteres. Las restricciones son implicaciones lógicas con antecedente formado por un átomo definido sobre una relación del esquema fuente y un consecuente formado por un átomo definido sobre una relación del esquema destino (ver programa de ejemplo). Hay dos clases de variables en una restricción, las que sólo aparecen en el consecuente y las demás. Las primeras se suponen cuantificadas existencialmente y las segundas universalmente.

Ejemplo. Programa LEXCHANGE.

```
ESQUEMA FUENTE
       estudiante(NOMBRE, NACIMIENTO, DNI)
       empleado(NOMBRE, DNI, TELEFONO)
DATOS FUENTE
       estudiante(Axel,1980,12122345)
       estudiante(Lorenzo,1982,10345321)
       estudiante(Antonio,1979,87654456)
       empleado(Axel,12122345,616234345)
       empleado(Manuel,50545318,617876654)
ESQUEMA DESTINO
       persona(NOMBRE, NACIMIENTO, DNI, TELEFONO)
RESTRICCIONES
       VAR x,y,z,u;
       estudiante(x,y,z) implica persona(x,y,z,u)
       VAR x,y,z,u;
       empleado(x,y,z) implica persona(x,u,y,z)
```

La ejecución del anterior programa transfiere los siguientes datos a la base de datos destino. Cada término X_i presente en los datos representa la instanciación de una variable cuantificada existencialmente en una restricción.

```
persona(Axel, 1980, 12122345, X1)
persona(Lorenzo, 1982, 10345321, X2)
persona(Antonio, 1979, 87654456, X3)
persona(Axel, X4, 12122345, 616234345)
persona(Manuel, X5, 50545318, 617876654)
```

SE PIDE: Intérprete de programas LEXCHANGE. Se debe construir la solución con un *tree-parser*.

Solución:

Analizador léxico:

```
class Analex extends Lexer;

options{
      importVocab = Anasint;
}
tokens{
      ESQUEMA       = "ESQUEMA";
      FUENTE        = "FUENTE";
      DATOS         = "DATOS";
      DESTINO       = "DESTINO";
      RESTRICCIONES = "RESTRICCIONES";
      VAR           = "VAR";
      IMPLICA       = "implica";
}
protected NL     : "\r\n" {newline();};
protected DIGITO : '0'..'9';
protected LETRA  : 'a'..'z'|'A'..'Z';
          BTF    : (' '|'\t'|NL) {$setType(Token.SKIP);};
          IDENT  : LETRA(LETRA|DIGITO)*;
          NUMERO : (DIGITO)+;
          PA     : '(';
          PC     : ')';
          COMA   : ',';
          PyC    : ';';
```

Analizador sintáctico:

```
class Anasint extends Parser;
options{
      buildAST = true;
}
tokens{
      ENTRADA;
      ESQUEMAFUENTE;
      ESQUEMADESTINO;
      DATOSFUENTE;
      RESTRICCION;
}
entrada        : esquema_fuente datos_fuente esquema_destino restricciones EOF!
                 {#entrada = #(#[ENTRADA, "ENTRADA"], ##);}
               ;
esquema_fuente : ESQUEMA! FUENTE! (signatura)+
                 {#esquema_fuente = #(#[ESQUEMAFUENTE, "ESQUEMA FUENTE"], ##);}
               ;
signatura      : IDENT^ PA! atributos PC!
               ;
atributos      :(IDENT COMA) => IDENT COMA! atributos
               | IDENT
               ;
datos_fuente   : DATOS! FUENTE! (a:atomo)+
                 {#datos_fuente = #(#[DATOSFUENTE, "DATOS FUENTE"], ##);}
               ;
esquema_destino: ESQUEMA! DESTINO! (signatura)+
                 {#esquema_destino = #(#[ESQUEMADESTINO, "ESQUEMA DESTINO"], ##);}
               ;
restricciones  : RESTRICCIONES^ (restriccion)+ ;
restricción    : variables implicacion
                 {#restriccion = #(#[RESTRICCION, "RESTRICCION"], ##);}
               ;
variables      : VAR^ vars PyC!
               ;
vars           : (IDENT COMA) => IDENT COMA! vars
               | IDENT
               ;
implicación    : atomo IMPLICA^ atomo ;
atomo          : IDENT^ PA! terminos PC! ;
terminos       : (termino COMA)=> termino COMA! terminos
               | termino
               ;
termino        : IDENT
               | NUMERO
               ;
```

Solución alternativa al Analizador sintáctico (mía).

```
class Anasint extends Parser;

options { buildAST =  true; }

tokens{ PROGRAMA; ESQUEMA; ESQUEMAFUENTE; ESQUEMADESTINO; DATOSFUENTE; RESTRICCION; }

programa     : esq_fuente dat_fuente esq_destino restricciones  {#programa = #(#[PROGRAMA, "Programa LEXCHANGE "], ##);};

esq_fuente   : ESQUEMA! FUENTE! (registro)+                     {#esq_fuente = #(#[ESQUEMAFUENTE, "Esquema Fuente"], ##);};
registro     : IDEN^ PA! campo PC!;
campo        : (IDEN COMA!) => (IDEN COMA! campo) | IDEN ;

dat_fuente   : DATOS! FUENTE! (tupla)+                          {#dat_fuente = #(#[DATOSFUENTE, "Datos Fuente"], ##);};
tupla        : IDEN^ PA! dato PC!;
dato         : (valor COMA!) => (valor COMA! dato) | valor ;
valor        : IDEN | NUMERO;

esq_destino  : ESQUEMA! DESTINO! registro                       {#esq_destino = #(#[DATOSFUENTE, "Esquema Destino"], ##);};

restricciones: RESTRICCIONES! (variables implicaciones)+        {#restricciones = #(#[RESTRICCION, "Restricciones"], ##);};
variables    : VAR^ campo PyC!;
implicaciones: registro IMPLICA^ registro;
```

5.1.3 CALCPROG

Supongamos un lenguaje llamado CALCPROG para programar secuencias de órdenes. Las órdenes pueden ser: Expresión entera, Asignación o Declaración de función. Las declaraciones de funciones están restringidas a un parámetro. A continuación se muestra un programa de ejemplo CALCPROG y su interpretación.

```
3 + (4 + 1);            // expresión
a = 1 + 1;              // asignación
f(a) = 10 * a;          // declaración de función
f(a);                   // expresión
g(x) = 10*f(x) + a;     // declaración de función
g(3);                   // expresión
f(f(2));                // expresión
f(a+1);                 // expresión
```

```
Expresión : ( + 3 ( + 4 1 ) )                 vale 8
Asignación: a                                 vale 2
Función   : f( a )                            vale ( * 10 a )
Expresión : ( LLAMADA f a )                   vale 20
Función   : g( x )                            vale ( + ( * 10 ( LLAMADA f x ) ) a )
Expresión : ( LLAMADA g 3 )                   vale 302
Expresión : ( LLAMADA f ( LLAMADA f 2 ) )     vale 200
Expresión : ( LLAMADA f ( + a 1 ) )           vale 30
```

SE PIDE: Analizador léxico-sintáctico para CALCPROG con capacidad para construir árboles de sintaxis abstracta (asa).

Solución:

Analizador léxico:

```
class Analex extends Lexer;

options { importVocab = Anasint; }

BT: (' '|'\t') {$setType(Token.SKIP);}            ;  // Blancos y Tabuladores
SL: "\r\n"     { newline(); $setType(Token.SKIP);} ;  // Salto de línea

// Lexemas auxiliares
protected DIGITO: ('0'..'9');
protected LETRA : ('a'..'z');
          IDENT : LETRA(LETRA|DIGITO)* ;  // Identificador
          NUMERO: (DIGITO)+ ;             // Número
          ASIG  : '=' ;                   // Operador de asignación
          PyC   : ';' ;                   // Punto y coma
          PA    : '(' ;                   // Paréntesis abierto
          PC    : ')' ;                   // Paréntesis cerrado
          MAS   : '+' ;                   // Operador aritmético +
          MENOS : '-' ;                   // Operador aritmético -
          POR   : '*' ;                   // Operador aritmético x
          DIV   : '/' ;                   // Operador aritmético /
```

Analizador sintáctico:

```
class Anasint extends Parser;

options { buildAST = true; }

tokens {
        ORDENES;
        DEF_FUNCION;
        LLAMADA;
}

// Símbolo inicial de la gramática
ordenes    : (orden PyC!)+ EOF!
             {#ordenes = #(#[ORDENES,"ORDENES"], ##);}
           ;

// Tipos de órdenes: (a) Definición de funciones (def_función)
//                   (b) Asignaciones            (asignación)
//                   (c) Expresiones             (expresión)
orden      : (IDENT PA IDENT PC ASIG) => def_funcion
           | (IDENT ASIG) => asignacion
           | expresion
           ;

// Definición de función.
def_funcion: IDENT PA! IDENT PC! ASIG! expresion
             {#def_funcion = #(#[DEF_FUNCION,"DEF_FUNCION"], ##);}
           ;

// Asignación.
asignacion : IDENT ASIG^ expresión
           ;

// Expresión.
expresion  : expresion1 ((MAS^|MENOS^) expresion)?
           ;
expresion1 : expresion2 ((POR^|DIV^) expresion1)?
           ;
expresion2 : NUMERO
           | (IDENT PA)=> IDENT PA! expresion PC!
             {#expresion2 = #(#[LLAMADA, "LLAMADA"], ##);}
           | IDENT
           | PA! expresion PC!
           ;
```

Solución alternativa:

```
header{ import java.util.*; }

class Anasint2 extends TreeParser; //Parser de asas

options{ importVocab = Anasint; }


 {
       Hashtable<String,AST> funcion  = new Hashtable<String,AST>();
       Set<String>           ciclicas = new HashSet<String>();

       //Recorre todas las funciones de la tabla funcion y les aplica el metodo ciclo
       public void comprueba(){
          Set<String> funciones = funcion.keySet();
          for (String e:funciones)
             ciclo (e, funcion.get(e));
       }

       //Comprueba si la definición de una función es cíclica o no
       public void ciclo(String fun, AST expr){
          switch(expr.getType()){
             case NUMERO:
             case IDENT:
                   break;
             case MAS:
             case POR:
             case DIV:
                ciclo(fun, expr.getFirstChild());
                ciclo(fun, expr.getFirstChild().getNextSibling());
                   break;
             case MENOS:
                if (expr.getFirstChild().getNextSibling() == null)
                   ciclo(fun,expr.getFirstChild());
                else{
                   ciclo(fun,expr.getFirstChild());
                   ciclo(fun,expr.getFirstChild().getNextSibling());
                }
                   break;
             case LLAMADA:
                if (expr.getFirstChild().getText().equals(fun) ||
                   ciclicas.contains(expr.getFirstChild().getText())){
                   System.out.println("Definición cíclica: " + fun + " = " +expresion(funcion.get(fun)));
                   ciclicas.add(fun);
                }else{
                   if(funcion.keySet().contains(expr.getFirstChild().getText())){
                      expr = funcion.get(expr.getFirstChild().getText());
                      ciclo(fun, expr);
                   }else
                      System.out.println("Función indefinida: " + expr.getFirstChild().getText() +
                      " en " + fun + " = " + expresion(funcion.get(fun)));
                }
          }
       }

       //Transforma un arbol AST a cadena de texto en formato original
       public String expresion(AST expr){
          String res="";
          if (expr.getText().equals("+") || expr.getText().equals("*") || expr.getText().equals("/")) {
             res = res + expresion(expr.getFirstChild()) + expr.getText()
                 + expresion(expr.getFirstChild().getNextSibling());
          } else if (expr.getText().equals("-")) {
             if(expr.getFirstChild().getNextSibling() == null)
                res = res + "-" + expr.getFirstChild().getText();
             else
                res = res + expresion(expr.getFirstChild()) + expr.getText()
                     + expresion(expr.getFirstChild().getNextSibling());
          }else if (expr.getText().equals("LLAMADA")) {
                res = res + expr.getFirstChild().getText()
                     + "(" + expr.getFirstChild().getNextSibling().getText() + ")";
          }else
             res = res + expr.getText();
          return res;
       }
 }
```

```
ordenes :  #(ORDENES (orden)+)
        ;

orden   : #(DEF_FUNCION a:IDENT IDENT b:expr) {funcion.put(a.getText(),b);}
        | #(ASIG IDENT expr)
        | expr
        ;

expr    :  #(MAS     expr expr)
        | (#(MENOS   expr expr)) => #(MENOS expr expr)
        |  #(POR     expr expr)
        |  #(DIV     expr expr)
        |  #(LLAMADA expr expr)
        | NUMERO
        | IDENT
        |  #(MENOS   expr)
;
```

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;

public class Principal {

  public static void main(String[] args) {
     try{

        FileInputStream f = new FileInputStream("Entrada.txt");
        Analex analex = new Analex(f);
        Anasint anasint = new Anasint(analex);
        anasint.ordenes();

        AST a = anasint.getAST();
        ASTFrame af = new ASTFrame("Entrada.txt", a);
        af.setVisible(true);
        Anasint2 anasint2 = new Anasint2();
        anasint2.ordenes(a);
        anasint2.comprueba();

     }
     catch(FileNotFoundException e)      { System.out.println("Error in file");          }
     catch(RecognitionException  e)      { System.out.println("Error sintactico " + e); }
     catch(TokenStreamException  e)      { System.out.println("Error Lexico " + e);     }
  }

}
```

5.1.4 UMLTEXT

Supongamos un lenguaje llamado UMLTEXT diseñado para expresar textualmente diagramas de clase y diagramas de objetos UML. Los diagramas de clase tienen la siguiente expresividad: Declaración de clase, Declaración de asociación y Declaración de restricciones de multiplicidad sobre los extremos de la asociación. Un ejemplo de sentencia UMLTEXT es la siguiente:

```
BEGIN_CLASS_DIAGRAM //Diagrama de clase.
        (Class A) //Clase A
        (Class B) //Clase B
        (Association R between A and B)                    // Asociación entre A y B
        (Multiplicity R on A is = 1)                       // Restricción multiplicidad sobre R en A
        (Multiplicity R on B is >= 1)                      // Restricción multiplicidad sobre R en B
END_CLASS_DIAGRAM

BEGIN_OBJECT_DIAGRAM //Diagrama de objetos.
        (Object o1 Class A)                                // Objeto clase A
        (Object o2 Class B)                                // Objeto clase B
        (Object o3 Class A)                                // Objeto clase A
        (Object o4 Class B)                                // Objeto clase B
        (Link l1 Association R LinkedObjects o1 o2)        // Enlace de R
        (Link l2 Association R LinkedObjects o3 o2)        // Enlace de R
END_OBJECT_DIAGRAM
```

SE PIDE: Analizador léxico-sintáctico para UMLTEXT con capacidad para construir árboles asas.

Analizador léxico:

```
class Analex extends Lexer;
options{
        importVocab   = Anasint;  // Importación del vocabulario de tokens desde el analizador Anasit.g
        testLiterals  = false;    // Por defecto no se activa la comprobación de literales declarados
}
tokens{
        CLASS         = "Class";
        ASSOCIATION   = "Association";
        MULTIPLICITY  = "Multiplicity";
        OBJECT        = "Object";
        LINK          = "Link";
        LINKEDOBJECTS = "LinkedObjects";
        BETWEEN       = "between";
        AND           = "and";
        IS            = "is";
        ON            = "on";
        BEGIN_CD      = "BEGIN_CLASS_DIAGRAM";
        BEGIN_OD      = "BEGIN_OBJECT_DIAGRAM";
        END_CD        = "END_CLASS_DIAGRAM";
        END_OD        = "END_OBJECT_DIAGRAM";
}

BT     : (' '|'\t') {$setType(Token.SKIP);} ; // No sirve para análisis sintáctico (B)lancos y (T)tabuladores
SL     : "\r\n" {newline(); $setType(Token.SKIP);} ; // (S)altos de (L)ínea
PA     : '(';                                        // (P)aréntesis (A)bierto
PC     : ')';                                        // (P)aréntesis (C)errado
OP_REL : "<="   {$setType(LE);}                      //
       | ">="   {$setType(GE);}                      // Operadores relacionales
       | "="    {$setType(EQ);}                      //
       ;
protected DIGITO : ('0'..'9')                    ;
protected LETRA  : ('a'..'z'|'A'..'Z')           ;
          NUMERO : (DIGITO)+ {$setType(NUMBER);} ;

   // Lexema IDENT (Identificadores con o sin prefijo)
   // Se activa la comprobación de palabras reservadas que tienen preferencia a cualquier otro identificador.
   // IDENT options {testLiterals=true;}: AUX_IDENT ('.' AUX_IDENT)* ;
IDENT options {testLiterals=true;}: LETRA(LETRA|DIGITO|'_')*
                                  ;
```

Analizador sintáctico:

```
header{
        import java.util.*;
        import antlr.*;
}

class Anasint extends Parser;

options{
        buildAST=true;
}

tokens{
        SPEC;
        CLASSD;
        OBJECTD;
}
{
        AST generateSpec(AST dclass, AST dobject){
              ASTFactory f = new ASTFactory();
              AST result = new CommonAST();
              result.setType(SPEC);
              result.setText("SPECIFICATION");
              AST dcAST = new CommonAST();
              dcAST.setType(CLASSD);
              dcAST.setText("CLASS DIAGRAM");
              AST dcAST2 = f.dupList(dclass);
              dcAST.setFirstChild(dcAST2);
              AST doAST = new CommonAST();
              doAST.setType(OBJECTD);
              doAST.setText("OBJECT DIAGRAM");
              AST doAST2 = f.dupList(dobject);
              doAST.setFirstChild(doAST2);
              result.setFirstChild(dcAST);
              AST aux = result.getFirstChild();
              aux.setNextSibling(doAST);
              return result;
        }

        AST generateElement(AST elem){
              ASTFactory f = new ASTFactory();
              AST result = f.dupTree(elem);
              return result;
        }
}

// REGLAS:

specification!        : a:class_diagram b:object_diagram EOF!     {#specification = generateSpec(#a,#b); }
                      ;
class_diagram         : BEGIN_CD! (class_diagram_element)* END_CD!
                      ;
class_diagram_element : (PA CLASS) => a:class_def!                {## = generateElement(#a);}
                      | (PA ASSOCIATION) => b:association_def     {## = generateElement(#b);}
                      | c:multiplicity_def                        {## = generateElement(#c);}
                      ;
class_def             : PA! CLASS^ IDENT PC!
                      ;
association_def       : PA! ASSOCIATION^ IDENT BETWEEN! IDENT AND! IDENT PC!
                      ;
multiplicity_def      : PA! MULTIPLICITY^ IDENT ON! IDENT IS! op_rel NUMBER PC!
                      ;
object_diagram        : BEGIN_OD! (object_diagram_element)* END_OD!
                      ;
object_diagram_element: (PA OBJECT) => a:object_def               {## = generateElement(#a);}
                      | b:link_def                                {## = generateElement(#b);}
                      ;
object_def            : PA! OBJECT^ IDENT CLASS! IDENT PC!
                      ;
link_def              : PA! LINK^ IDENT ASSOCIATION! IDENT LINKEDOBJECTS! IDENT IDENT PC!
                      ;
op_rel                : EQ
                      | LE
                      | GE
                      ;
```

5.1.5 LPROC

Supongamos un lenguaje llamado LPROC diseñado para expresar procedimientos como el mostrado en el siguiente ejemplo.

```
buscar(entero e, vector(entero)[1..n] v) devuelve (booleano, entero)
variables locales:
	booleano resultado;
	entero elemento, i;
instrucciones:
	(resultado, i) = (falso, 1);
	mientras (resultado == falso y i<=numeroElementos(v)) hacer
		elemento = v[i];
		si (elemento == e) entonces
			resultado = cierto;
		sino
			i = i + 1;
		finsi
	finmientras
	devuelve (resultado,i);
fin
```

El perfil de un procedimiento consta de Nombre, Parámetros y Resultados. Internamente el procedimiento se estructura con un conjunto de variables locales y una secuencia de instrucciones. Los tipos elementales en LPROC son el tipo lógico y el tipo entero. El único tipo no elemental es el tipo vector (de elementos enteros o lógicos). La declaración de un vector siempre incluirá el tipo de sus elementos y el rango de sus índices. LPROC tiene 5 tipos de instrucciones: Asignaciones simples, Asignaciones múltiples, Iteraciones, Condicionales y Devolución de resultados.

SE PIDE: Analizador léxico-sintáctico para LPROC con capacidad para construir árboles de sintaxis abstracta.

Solución:

Analizador léxico:

```
class Analex extends Lexer;

options{
        importVocab = Anasint;
        k = 2;
}

tokens{
        VARIABLES     = "variables";
        LOCALES       = "locales";
        INSTRUCCIONES = "instrucciones";
        DEV           = "devuelve";
        BOOLEANO      = "booleano";
        ENTERO        = "entero";
        VECTOR        = "vector";
        FIN           = "fin";
        MIENTRAS      = "mientras";
        SI            = "si";
        FINMIENTRAS   = "finmientras";
        FINSI         = "finsi";
        SINO          = "sino";
        ENTONCES      = "entonces";
        HACER         = "hacer";
        Y             = "y";
        O             = "o";
        NO            = "no";
}

protected NL        : "\r\n"         {newline();} ;

protected DIGITO    : '0'..'9' ;

protected LETRA     : 'a'..'z'|'A'..'Z' ;


          BTN       : (' '|'\t'|NL)  {$setType(Token.SKIP);} ;
          NUMERO    : (DIGITO)+ ;
          IDENT     : (LETRA)(LETRA|DIGITO)* ;

          OP        : ("==") => "==" {$setType(IGUAL);}
                    | "="            {$setType(ASIG);}
                    ;

          MAYOR     : '>'  ;
          MENOR     : '<'  ;
          MAYORIGUAL: ">=" ;
          MENORIGUAL: "<=" ;

          DP        : ':'  ;
          RANGO     : ".." ;
          DISTINTO  : "!=" ;
          COMA      : ','  ;
          PA        : '('  ;
          PC        : ')'  ;
          CA        : '['  ;
          CC        : ']'  ;
          PyC       : ';'  ;

          MAS       : '+'  ;
          MENOS     : '-'  ;
          POR       : '*'  ;
          DIV       : '/'  ;
```

Analizador sintáctico:

```
class Anasint extends Parser;

options{ buildAST = true; }

tokens{
	PROCEDURE;
	PERFIL;
	NOMBRE;
	PARAMETROS;
	RESULTADOS;
	VARIABLESLOCALES;
	INSTRS;
	VARS;
	EXPRS;
}

procedure!		: a:perfil b:variableslocales c:instrucciones EOF
			  {#procedure = #(#[PROCEDURE,"PROCEDURE"], #a, #b, #c);}
			;

perfil!			: a:IDENT PA b:parametros PC DEV PA c:tipos PC
			  {#perfil = #(#[PERFIL    ,"PERFIL"    ],
			               #(#[NOMBRE    ,"NOMBRE"    ], #a),
			               #(#[PARAMETROS,"PARAMETROS"], #b),
			               #(#[RESULTADOS,"RESULTADOS"], #c)    );}
			;

parametros		: (parametro COMA) => parametro COMA! parametros
			| parametro
			|
			;

parametro		: tipo IDENT^
			;

tipos			: (tipo COMA)=> tipo COMA! tipos
			| tipo
			|
			;

variableslocales!	: VARIABLES LOCALES DP a:variables
			  {#variableslocales = #(#[VARIABLESLOCALES,"VARIABLESLOCALES"], #a);}
			;

variables		: decl_vars variables
			|
			;

decl_vars!		: a:tipo b:vars PyC
			  {#decl_vars = #(#a, #b);}
			;

vars			: (IDENT COMA) => IDENT COMA! vars
			| IDENT
			;

instrucciones		: INSTRUCCIONES! DP! (instruccion)* FIN!
			  {#instrucciones = #(#[INSTRS,"INSTRUCCIONES"], ##);}
			;

instruccion		: asignacion
			| iteracion
			| seleccion
			| retorno
			;

asignacion		: asignacionsimple
			| asignacionmultiple
			;
asignacionsimple	: IDENT ASIG^ expr PyC!
			;

asignacionmultiple!	: PA a:vars PC b:ASIG PA c:exprs PC PyC!
			  {#asignacionmultiple = #(#b,#(#[VARS,"VARIABLES"],#a), #(#[EXPRS,"EXPRESIONES"],#c));}
			;
```

```
iteracion   : MIENTRAS^ PA! expr PC! HACER! bloque FINMIENTRAS!
            ;

seleccion   : (SI PA expr PC ENTONCES bloque SINO) => SI^ PA! expr PC! ENTONCES! bloque SINO! bloque FINSI!
            | SI^ PA! expr PC! ENTONCES! bloque FINSI!
            ;

bloque      : (instruccion)*
            ;

retorno     : DEV^ PA! exprs PC! PyC!
            ;

tipo        : VECTOR^ PA! tipo PC! CA! indice RANGO! indice CC!
            | ENTERO
            | BOOLEANO
            ;

indice      : NUMERO
            | IDENT
            ;

exprs       : (expr COMA) => expr COMA! exprs
            | expr
            |
            ;

expr        : (expr2 (Y|O)) => expr2 (Y^|O^) expr
            | expr2
            ;

expr2       : (termino (MAYOR|MENOR|MAYORIGUAL|MENORIGUAL|IGUAL|DISTINTO)) =>
              termino (MAYOR^|MENOR^|MAYORIGUAL^|MENORIGUAL^|IGUAL^|DISTINTO^) termino
            | termino
            ;

termino     : (termino2 (MAS|MENOS|POR|DIV)) => termino2 (MAS^|MENOS^|POR^|DIV^) termino
            | termino2
            ;

termino2    : (IDENT CA) => IDENT^ CA! expr CC!
            | (IDENT PA) => IDENT^ PA! exprs PC!
            | IDENT
            | NUMERO
            | FALSO
            | CIERTO
            ;
```

5.2 Análisis semántico.

Objetivo: Problemas para consolidar el diseño y construcción de analizadores semánticos.

5.2.1 La ruptura del control inalcanzable

Supongamos un lenguaje de programación secuencial con instrucción de ruptura de control. El objetivo del problema es detectar en los programas rupturas de control inalcanzables (y por tanto innecesarias). Los siguientes programas muestran rupturas inalcanzables.

Programa 1:

```
i=1;
while (i<=10){
      if (v==x){
            break;
            i=i+3;
            break; //Inalcanzable
      }
      break;
      while (i>9)
            break; //Inalcanzable
}
```

Programa 2:

```
i=1;
while (i<=10){
      if (v==x){
            break;
            i=i+3;
            break; //Inalcanzable
      }
      if (v==2) break;
      if (v==x) break;
      while (i>9)
            break;
}
```

Programa 3:

```
i=1;
while (i<=10){
      if (v==x){
            break;
            i=i+3;
            break; //Inalcanzable
      }
      break;
      break; //Inalcanzable
      while (i>9)
            break; //Inalcanzable
}
```

SE PIDE: Analizador semántico para detectar rupturas de control inalcanzables en un programa. Para implementarlo, se suministra el siguiente parser ANTLR para el lenguaje de programación:

Analizador sintáctico:

```
class Anasint extends Parser;

instrucciones : (instruccion)* ;
instruccion   : asignacion
              | iteracion
              | seleccion
              | ruptura
              ;
asignacion    : IDENT ASIG expr PyC
              ;
iteracion     : WHILE PA expr PC bloque
              ;
seleccion     : IF PA expr PC bloque
              ;
ruptura       : BREAK PyC
              ;
bloque        : instruccion
              | LLA instrucciones LLC
              ;
expr          : expr_suma ((MENOR|MAYOR|MENORIGUAL|MAYORIGUAL|IGUAL|DISTINTO) expr_suma)?
              ;
expr_suma     : expr_mult ((MAS|MENOS) expr_mult)*
              ;
expr_mult     : expr_base ((POR|DIV) expr_base)*
              ;
expr_base     : NUMERO
              | IDENT
              | PA expr PC
              ;
```

Solución:

Analizador semántico:

```
class Anasint extends Parser;

instrucciones[Pila p] : (instruccion[p])*
                      ;
instruccion[Pila p]   : asignacion
                      | iteracion[p] { p.quitar(); }
                      | seleccion[p] { p.quitar(); }
                      | ruptura  [p]
                      ;
asignacion            : IDENT ASIG expr PyC
                      ;
iteracion[Pila p]     : WHILE PA expr PC {if (!p.cima())  p.poner(false);  else  p.poner(true);}  bloque[p]
                      ;
seleccion[Pila p]     : IF PA expr PC    {if (!p.cima())  p.poner(false);  else  p.poner(true);}  bloque[p]
                      ;
ruptura[Pila p]       : a:BREAK PyC
                        {
                         if (!p.cima())
                            System.out.println( "Linea " + a.getLine()
                                              + ", Columna " + a.getColumn()
                                              + ": Break inalcanzable"
                                              );
                         else {
                            p.quitar();
                            p.poner(false);}
                         }
                      ;
bloque[Pila p]        : instruccion[p]
                      | LLA instrucciones[p] LLC
                      ;
expr                  : expr_suma ((MENOR|MAYOR|MENORIGUAL|MAYORIGUAL|IGUAL|DISTINTO) expr_suma)?
                      ;
expr_suma             : expr_mult ((MAS|MENOS) expr_mult)*
                      ;
expr_mult             : expr_base ((POR|DIV) expr_base)*
                      ;
expr_base             : NUMERO
                      | IDENT
                      | PA expr PC
                      ;
```

Clase auxiliar:

```
import java.util.*;

public class Pila {
     private Stack<Boolean> p = new Stack<Boolean>();

     public Pila() {
     }

     public boolean cima() {
        return (p.peek()).booleanValue();
     }
     public void poner(boolean elem) {
        p.push(new Boolean(elem));
     }
     public boolean quitar() {
        return (p.pop()).booleanValue();
     }
     public boolean vacia() {
        return p.empty();
     }
}
```

5.2.2 LEXCHANGE

Supongamos un lenguaje llamado LEXCHANGE para programar transferencias de datos entre dos bases de datos relacionales. Un programa típico en LEXCHANGE incluye: (a) esquema de datos fuente, (b) datos fuente, (c) esquema de datos destino y (d), un conjunto de restricciones especificando la transferencia de datos entre la fuente y el destino. Todos los datos son de tipo cadena de caracteres. Las restricciones son implicaciones lógicas con antecedente formado por un átomo definido sobre una relación del esquema fuente y un consecuente formado por un átomo definido sobre una relación del esquema destino (ver programa de ejemplo). Hay dos clases de variables en una restricción, las que sólo aparecen en el consecuente y las demás. Las primeras se suponen cuantificadas existencialmente y las segundas universalmente.

Ejemplo. Programa LEXCHANGE.

```
ESQUEMA FUENTE
        estudiante(NOMBRE, NACIMIENTO, DNI)
        empleado(NOMBRE, DNI, TELEFONO)
DATOS FUENTE
        estudiante(Axel,1980,12122345)
        estudiante(Lorenzo,1982,10345321)
        estudiante(Antonio,1979,87654456)
        empleado(Axel,12122345,616234345)
        empleado(Manuel,50545318,617876654)
ESQUEMA DESTINO
        persona(NOMBRE, NACIMIENTO, DNI, TELEFONO)
RESTRICCIONES
        VAR x,y,z,u;
        estudiante(x,y,z) implica persona(x,y,z,u)
        VAR x,y,z,u;
        empleado(x,y,z) implica persona(x,u,y,z)
```

La ejecución del anterior programa transfiere los siguientes datos a la base de datos destino:

```
persona(Axel, 1980, 12122345, X1)
persona(Lorenzo, 1982, 10345321, X2)
persona(Antonio, 1979, 87654456, X3)
persona(Axel, X4, 12122345, 616234345)
persona(Manuel, X5, 50545318, 617876654)
```

Cada dato X_i representa la instanciación de una variable cuantificada existencialmente en una restricción.

SE PIDE Analizador semántico para decidir si un programa LEXCHANGE es semánticamente correcto. Por programa semánticamente correcto se entiende un programa cuyos datos fuente son instancias de relaciones en el esquema fuente. Es decir, no hay ningún dato fuente declarado sobre una relación no declarada en el esquema fuente y los datos fuente presentan un número de argumentos coincidente con el número de argumentos de las relaciones declaradas en el esquema fuente. Resolver el problema anotando el siguiente parser para el lenguaje LEXCHANGE:

Analizador sintáctico:

```
class Anasint extends Parser;

options{
      buildAST = true; // Construcción de asa
}

tokens{
      ENTRADA;
      ESQUEMAFUENTE;
      ESQUEMADESTINO;
      DATOSFUENTE;
      RESTRICCION;
}


entrada        : esquema_fuente datos_fuente esquema_destino restricciones EOF!
                 {#entrada = #(#[ENTRADA, "ENTRADA"], ##);}
               ;

esquema_fuente : ESQUEMA! FUENTE! (signatura)+
                 {#esquema_fuente = #(#[ESQUEMAFUENTE, "ESQUEMA FUENTE"], ##);}
               ;

signatura      : IDENT^ PA! atributos PC!
               ;

atributos      :(IDENT COMA) => IDENT COMA! atributos
               | IDENT
               ;

datos_fuente   : DATOS! FUENTE! (atomo)+
                 {#datos_fuente = #(#[DATOSFUENTE, "DATOS FUENTE"], ##);}
               ;

esquema_destino: ESQUEMA! DESTINO! (signatura)+
                 {#esquema_destino = #(#[ESQUEMADESTINO, "ESQUEMA DESTINO"], ##);}
               ;

restricciones  : RESTRICCIONES^ (restriccion)+
               ;

restriccion    : variables implicacion
                 {#restriccion = #(#[RESTRICCION, "RESTRICCION"], ##);}
               ;

variables      : VAR^ vars PyC!
               ;

vars           : (IDENT COMA) => IDENT COMA! vars
               | IDENT
               ;

implicacion    : atomo IMPLICA^ atomo ;

atomo          : IDENT^ PA! terminos PC! ;

terminos       : (termino COMA) => termino COMA! terminos
               | termino
               ;

termino        : IDENT
               | NUMERO
               ;
```

Solución:

Analizador semántico.

```
header{
	import java.util.*;
}

class Anasint extends Parser;

options{
	buildAST = true;
}
tokens{
	ENTRADA;
	ESQUEMAFUENTE;
	ESQUEMADESTINO;
	DATOSFUENTE;
	RESTRICCION;
}

{
	Hashtable predicadoNumeroArgumentos = new Hashtable<String, Integer>();
	void chequear(AST atom){
		AST argumentos = atom.getFirstChild();
		int n =0;
		while (argumentos != null){
		   n++;
		   argumentos = argumentos.getNextSibling();
		}
		if (predicadoNumeroArgumentos.get(atom.getText()) == null)
		   System.out.println("Error en un átomo sin definición: " + atom.toStringTree());
		else
		   if (n != ((Integer)predicadoNumeroArgumentos.get(atom.getText())).intValue())
		      System.out.println("Error en número de argumentos en el átomo: " + atom.toStringTree());
	}
}

entrada           : esquema_fuente datos_fuente esquema_destino restricciones EOF!
                    {#entrada = #(#[ENTRADA, "ENTRADA"], ##);}
                  ;
esquema_fuente    : ESQUEMA! FUENTE! (signatura)+
                     {#esquema_fuente = #(#[ESQUEMAFUENTE, "ESQUEMA FUENTE"], ##);}
                  ;
signatura {int n;}: i:IDENT^ PA! n=atributos PC!
                    {predicadoNumeroArgumentos.put(i.getText(), n);}
                  ;
atributos returns [int n=0;]:(IDENT COMA) => IDENT COMA! n=atributos { n++; }
                            | IDENT {n = 1;}
                            ;
datos_fuente      : DATOS! FUENTE! (a:atomo {chequear(#a);})+
                    {#datos_fuente = #(#[DATOSFUENTE, "DATOS FUENTE"], ##);}
                  ;
esquema_destino   : ESQUEMA! DESTINO! (signatura)+
                    {#esquema_destino = #(#[ESQUEMADESTINO, "ESQUEMA DESTINO"], ##);}
                  ;
restricciones     : RESTRICCIONES^ (restriccion)+
                  ;
restriccion       : variables implicacion
                    {#restriccion = #(#[RESTRICCION, "RESTRICCION"], ##);}
                  ;
variables         : VAR^ vars PyC!
                  ;
vars              : (IDENT COMA) => IDENT COMA! vars
                  | IDENT
                  ;
implicacion       : atomo IMPLICA^ atomo
                  ;
atomo             : IDENT^ PA! terminos PC!
                  ;
terminos          : (termino COMA) => termino COMA! terminos
                  | termino
                  ;
termino           : IDENT
                  | NUMERO
                  ;
```

Analizador semántico. Solución alternativa con atributos heredados y sintetizados.

```
header{
   import java.util.*;
}

class Anasint extends TreeParser;

options{
   importVocab = Anasint;
}


entrada          :
                 {Hashtable<String, Integer> almacen;}
                 #(ENTRADA almacen=esquema_fuente
                                  datos_fuente   [almacen]
                                  esquema_destino[almacen]
                                  restricciones  [almacen])
                 ;

esquema_fuente   returns [Hashtable<String, Integer> almacen = new Hashtable<String, Integer>()]
                                                  : #(ESQUEMAFUENTE  (signatura[almacen])+)
                 ;

datos_fuente     [Hashtable<String, Integer> almacen] : #(DATOSFUENTE    (atomo[almacen])+)
                 ;

esquema_destino  [Hashtable<String, Integer> almacen] : #(ESQUEMADESTINO (signatura[almacen])+)
                 ;

restricciones    [Hashtable<String, Integer> almacen] : #(RESTRICCIONES  (restriccion[almacen])+)
                 ;

restriccion      [Hashtable<String, Integer> almacen] : #(RESTRICCION variables  implicacion[almacen])
                 ;

implicacion      [Hashtable<String, Integer> almacen] : #(IMPLICA     atomo[almacen]  atomo[almacen])
                 ;

signatura        [Hashtable<String, Integer> almacen] : #(a:IDENT {int cont = 0;} (IDENT {cont++;})+)
                                                        {almacen.put(a.getText(), cont);}
                 ;

atomo            [Hashtable<String, Integer> almacen] : #(a:IDENT {int cont=0;} (termino {cont++;} )+)
                 {
                   if (almacen.containsKey(a.getText()))
                      if (almacen.get(a.getText()) == cont)
                          ;
                      else
                          System.out.println("ERROR (número de atributos incorrecto): " + a.getText());
                   else
                      System.out.println("ERROR (tabla no declarada): " + a.getText());
                 }
                 ;

variables        : #(VAR  (IDENT)+)
                 ;

termino          : IDENT
                 | NUMERO
                 ;
```

Analizador semántico. Solución alternativa con mensajes de error precisos.

```
header{ import java.util.*; }

class Anasint extends TreeParser;

options{ importVocab = Anasint; }


{
  String astToString(AST asa){
        String resultado = new String("");
        resultado += asa.getText();
        resultado += "(";
        AST aux = asa.getFirstChild();
        while (aux != null)
        {
           resultado += aux.getText();
           aux = aux.getNextSibling();
           if (aux != null) resultado += ",";
        }
        resultado += ")";
        return resultado;
  }

  void verificar(int error, AST asa){
     switch(error){
        case 0: break;
        case 1: System.out.println("ERROR (tabla no declarada): " + asa.getText());
                break;
        case 2: System.out.println("ERROR (número de atributos incorrecto) en tupla: " + astToString(asa));
                break;
        default:
     }
  }
}



entrada         {Hashtable<String, Integer> almacen;}
                :#(ENTRADA almacen=esquema_fuente
                                   datos_fuente   [almacen]
                                   esquema_destino[almacen]
                                   restricciones  [almacen]
                  )
                {System.out.println(almacen.toString());}
                ;


esquema_fuente  returns [Hashtable<String, Integer> almacen = new Hashtable<String, Integer>()]
                :#(ESQUEMAFUENTE (signatura[almacen])+)
                ;


datos_fuente    [Hashtable<String, Integer> almacen]
                {int error;}
                :#(DATOSFUENTE (error=a:atomo[almacen] {verificar(error, a);})+)
                ;


esquema_destino [Hashtable<String, Integer> almacen]
                :#(ESQUEMADESTINO (signatura[almacen])+)
                ;


restricciones   [Hashtable<String, Integer> almacen]
                :#(RESTRICCIONES (restriccion[almacen])+)
                ;

signatura       [Hashtable<String, Integer> almacen]
                :#(a:IDENT {int cont = 0;} (IDENT {cont++;})+)
                {almacen.put(a.getText(), cont);}
                ;
```

```
atomo           [Hashtable<String, Integer> almacen]
                returns [int error=0]
                :#(a:IDENT {int cont = 0;} (termino {cont++;})+)
                {
                  if (almacen.containsKey(a.getText()))
                     if (almacen.get(a.getText()) == cont)
                        error = 0;
                     else
                        error = 2;
                  else
                     error = 1;
                }
                ;


restriccion     [Hashtable<String, Integer> almacen]
                :#(RESTRICCION variables implicacion[almacen])
                ;


variables       :#(VAR (IDENT)+)
                ;


implicacion     [Hashtable<String, Integer> almacen]
                {int error;}
                :#(IMPLICA error=atomo[almacen]  error=atomo[almacen])
                ;


termino         : IDENT
                | NUMERO
                ;
```

5.2.3 PROP

Supongamos un lenguaje de proposiciones lógicas llamado PROP. El lenguaje tiene la siguiente capacidad expresiva:

(a) La variable proposicional en PROP se expresa mediante una letra minúscula.
(b) PROP dispone de los operadores lógicos: equivalencia (<=>), implicación (=>), conjunción (&), disyunción (|) y negación (-).
(c) PROP dispone de constantes lógicas cierto y falso.

Para precisar la sintaxis de una proposición, se suministra el siguiente parser:

```
class Anasint extends Parser;

options{
	buildAST = true;
}

tokens{
	FORMULA;
}

proposicion! : a:formula EOF {#proposicion = #(#[FORMULA, "FORMULA"], #a);}
             ;

formula      : (formula1 IMPL) => formula1 IMPL^ formula
             | formula1
             ;

formula1     : (formula2 EQUIV) => formula2 EQUIV^ formula1
             | formula2
             ;

formula2     : (formula3 OR) => formula3 OR^ formula2
             | formula3
             ;

formula3     : (formula4 AND) => formula4 AND^ formula3
             | formula4
             ;

formula4     : (NEG) => NEG^ formula5
             | formula5
             ;

formula5     : PROP
             | TRUE
             | FALSE
             | PA! formula PC!
             ;
```

Un ejemplo de fórmula proposicional es el siguiente: `p => (q | r & v)`

SE PIDE: Construir un analizador semántico (*TreeParser* ANTLR) que sea capaz de decidir si una proposición PROP (en formato de árbol de sintaxis abstracta generado por el parser dado) es *consistente*.

Una proposición PROP es consistente si existe alguna asignación de valores a sus variables que la haga cierta. Si no es así, entonces la fórmula se dice que es *inconsistente*.

Solución:

Analizador semántico.

```
header{
     import java.util.*;
}

class Anasem extends TreeParser;

options{
     importVocab = Anasint;
}


{


  // Lista de proposiciones en la fórmula analizada
  List<String> vars = new LinkedList<String>();



  // Generador de tuplas de valores de verdad (tabla de verdad de tamaño 'vars')
  List<List<Boolean>> generate(){
         List<Boolean> aux = new LinkedList<Boolean>();
         List<List<Boolean>> sols = new LinkedList<List<Boolean>>();
         Boolean elem = false;
         for (int i = 1; i <= (int)Math.pow(2, vars.size()); i++)
            sols.add(new LinkedList<Boolean>());
            for (int i = 1; i <= vars.size(); i++){
               int k = (int)Math.pow(2, i - 1);
               for (int j = 0; j <= (int)Math.pow(2, vars.size()) - 1; j++){
                  aux = sols.remove(j);
                  aux.add(elem);
                  sols.add(j, aux);
                  if (j % k == 0) elem = !elem;
               }
            }
            return sols;
  }



  // Asignación de valores de verdad a la variables de la fórmula (interpretaciones)
  List<Hashtable<String,Boolean>> interpretaciones(List<List<Boolean>> valores){
         List<Hashtable<String,Boolean>> r = new LinkedList<Hashtable<String,Boolean>>();
         List<Boolean> aux = new LinkedList<Boolean>();
         for(int i = 0; i < valores.size(); i++){
            aux = valores.get(i);
            Hashtable<String, Boolean> taux = new Hashtable<String, Boolean>();
            for(int j = 0; j < aux.size(); j++)
               taux.put(vars.get(j), aux.get(j));
            r.add(taux);
         }
         return r;
  }



  // Evaluador de interpretaciones
  boolean consistente(AST formula, List<Hashtable<String,Boolean>> interpretaciones){
         boolean r = false;
         int i = 0;
         while(i < interpretaciones.size() && !r){
            r = evalua(formula, interpretaciones.get(i));
            if (r)
             System.out.println("Consistente (centinela): "+ interpretaciones.get(i).toString());
            i=i+1;
         }
         return r;
  }
```

```
  boolean evalua(AST formula, Hashtable<String,Boolean> interpretacion){
        boolean r = true;
        switch(formula.getType()){
           case TRUE : r = true;                                                  break;
           case FALSE: r = false;                                                 break;
           case PROP : r = interpretacion.get(formula.getText());                 break;
           case NEG  : r = !evalua(formula.getFirstChild(), interpretacion);      break;
           case AND  : r = evalua(formula.getFirstChild(), interpretacion)
                           &&
                           evalua(formula.getFirstChild().getNextSibling(), interpretacion);
                       break;
           case OR   : r = evalua(formula.getFirstChild(),interpretacion)
                           ||
                           evalua(formula.getFirstChild().getNextSibling(), interpretacion);
                       break;
           case IMPL : r = !evalua(formula.getFirstChild(), interpretacion)
                           ||
                           evalua(formula.getFirstChild().getNextSibling(), interpretacion);
                       break;
           case EQUIV: r = (evalua(formula.getFirstChild(), interpretacion)
                           ==
                           evalua(formula.getFirstChild().getNextSibling(), interpretacion));
                       break;
        }
        return r;
  }
}
```

```
proposicion : #(FORMULA f:formula)
              { System.out.println(consistente(#f, interpretaciones(generate()))); }
            ;

formula     : #(EQUIV formula formula)
            | #(IMPL  formula formula)
            | #(OR    formula formula)
            | #(AND   formula formula)
            | #(NEG   formula)
            | p:PROP  { if (!vars.contains(p.getText())) vars.add(p.getText()); }
            | TRUE
            | FALSE
            ;
```

Solución alternativa

```
class Analex extends Lexer;

options{
  importVocab = Anasint;
}

BLANCO   : ' '          {$setType(Token.SKIP);}           ;
FIN_LINEA: "\r\n"       {newline(); $setType(Token.SKIP);};
TABULADOR: '\t'         {$setType(Token.SKIP);}           ;
PROP     :  'a'..'z';
EQUIV    : "<=>";
IMPL     : "=>";
AND      : '&';
OR       : '|';
NEG      : '-';
PA       : '(';
PC       : ')';
PyC      : ';';
TRUE     : 'T';
FALSE    : 'F';
```

```
class Anasint1 extends Parser;

options{
   buildAST = true;
}

tokens{
   FORMULA;
}

proposicion! : a:formula EOF {#proposicion = #(#[FORMULA, "FORMULA"],#a);}
             ;

formula      : (formula1 IMPL) => formula1 IMPL^ formula
             | formula1
             ;

formula1     : (formula2 EQUIV) => formula2 EQUIV^ formula1
             | formula2
             ;

formula2     : (formula3 OR) => formula3 OR^ formula2
             | formula3
             ;

formula3     : (formula4 AND) => formula4 AND^ formula3
             | formula4
             ;

formula4     : (NEG) => NEG^ formula5
             | formula5
             ;

formula5     : PROP
             | TRUE
             | FALSE
             | PA! formula PC!
             ;
```

```
header{
  import java.util.*;
}

class Anasint2 extends TreeParser;

options{
  importVocab = Anasint;
}

proposicion returns [Set<String> vars = new HashSet<String>()]
                           : #(FORMULA formula[vars])
                           ;

formula [Set<String> vars] : #(IMPL  formula[vars] formula[vars])
                           | #(EQUIV formula[vars] formula[vars])
                           | #(OR    formula[vars] formula[vars])
                           | #(AND   formula[vars] formula[vars])
                           | #(NEG   formula[vars])
                           | a:PROP {vars.add(a.getText());}
                           | TRUE
                           | FALSE
;
```

```
header{
  import java.util.*;
}

class Anasint3 extends TreeParser;

options{
  importVocab = Anasint;
}

proposicion [Hashtable<String,Boolean> interpretacion]
            returns [Boolean evaluacion = true]
        : #(FORMULA evaluacion=formula[interpretacion])
        ;

formula [Hashtable<String,Boolean> interpretacion]
        returns [Boolean evaluacion = true ]
        {Boolean v1, v2;}
        : #(IMPL  v1=formula[interpretacion] v2=formula[interpretacion])        {evaluacion = (!v1 | v2) ;}
        | #(EQUIV v1=formula[interpretacion] v2=formula[interpretacion])        {evaluacion = (v1 == v2) ;}
        | #(OR    v1=formula[interpretacion] v2=formula[interpretacion])        {evaluacion = (v1 | v2)  ;}
        | #(AND   v1=formula[interpretacion] v2=formula[interpretacion]         {evaluacion = (v1 & v2)  ;}
        | #(NEG   v1=formula[interpretacion])                                   {evaluacion = (!v1)      ;}
        | a:PROP                                          {evaluacion = interpretacion.get(a.getText());}
        | TRUE                                                                  {evaluacion = true       ;}
        | FALSE                                                                 {evaluacion = false      ;}
        ;
```

5.3 Interpretación y compilación.

Objetivo: Problemas para consolidar el diseño y construcción de intérpretes y compiladores.

5.3.1 CALCPROG

Supongamos un lenguaje llamado CALCPROG para programar secuencias de órdenes. La orden puede ser: Expresión entera, Asignación o Declaración de función. Las declaraciones de funciones están restringidas a un parámetro. A continuación se muestra un programa de ejemplo CALCPROG y su interpretación.

```
3 + (4 + 1);            // expresión
a = 1 + 1;              // asignación
f(a) = 10 * a;          // declaración de función
f(a);                   // expresión
g(x) = 10*f(x) + a;     // declaración de función
g(3);                   // expresión
f(f(2));                // expresión
f(a+1);                 // expresión
```

```
Expresión : ( + 3 ( + 4 1 ) )                 vale 8
Asignación: a                                 vale 2
Función   : f( a )                            vale ( * 10 a )
Expresión : ( LLAMADA f a )                   vale 20
Función   : g( x )                            vale ( + ( * 10 ( LLAMADA f x ) ) a )
Expresión : ( LLAMADA g 3 )                   vale 302
Expresión : ( LLAMADA f ( LLAMADA f 2 ) )     vale 200
Expresión : ( LLAMADA f ( + a 1 ) )           vale 30
```

SE PIDE: Intérprete para programas CALCPROG. La solución se debe construir con un tree-parser. Suponer que el análisis léxico-sintáctico se ha resuelto con los lexer y parser propuestos en la solución del problema 3 del laboratorio de Análisis Léxico-Sintáctico.

Solución:

```
header{
     import java.io.*;
     import java.util.*;
     import antlr.*;
}

class Anasint extends TreeParser;

options{
     importVocab = Anasint;
}



{
     Hashtable tabla_variables  = new Hashtable();
     Hashtable tabla_parametros = new Hashtable();
     Hashtable tabla_funciones  = new Hashtable();

  ASTFactory af = new ASTFactory();



  void sustituir(AST expresion, AST parametro_formal, int parametro_real){
     //  Pre: 'expresion'  es el AST asociado a una expresión
     // 'parametro_formal' es el AST asociado a un parámetro formal
     // 'parametro_real'   parámetro real de tipo entero
     switch(expresion.getType()){
        case MAS:
        case MENOS:
        case POR:
        case DIV:
                  sustituir(expresion.getFirstChild(), parametro_formal, parametro_real);
                  sustituir(expresion.getFirstChild().getNextSibling(), parametro_formal, parametro_real);
                  break;
        case LLAMADA:
                  sustituir(expresion.getFirstChild().getNextSibling(), parametro_formal, parametro_real);
                  break;
        case IDENT:
                  if (expresion.getText().equals(parametro_formal.getText())){
                     expresion.setType(NUMERO);
                     expresion.setText(new Integer(parametro_real).toString());
                  }
                  break;
        default:
                  break;
     }
  }



  // Post: toda ocurrencia de 'parametro_formal' en 'expresion' se sustituye por 'valor'.
  int evaluarVariable(AST variable){
     // Pre: 'variable' es una variable en formato AST.
     if (tabla_variables.containsKey(variable.getText()))
        try{
           return this.expresion((AST)tabla_variables.get(variable.getText()));
        }catch(RecognitionException e){
           System.out.println("error sintáctico" + e.getMessage());
           return 0;
        }
     else
        return 0;
     // Post: se devuleve el valor de 'variable' almacenado en 'tabla_variables'.
     // Si la variable no se almecenó, se devuelve 0.
  }
```

```
   int evaluarFuncion(AST def_funcion, AST parametro_formal, int parametro_real){
      // Pre: 'def_funcion' es la expresión correspondiente a la definición de una función.
      // 'parametro_formal' es el parámetro formal de la función.
      // 'parametro_real'   es el parámetro real de tipo entero.
      try{
         switch(def_funcion.getType()){
            case MAS:
               return evaluarFuncion(def_funcion.getFirstChild(), parametro_formal, parametro_real)
               +
               evaluarFuncion(def_funcion.getFirstChild().getNextSibling(), parametro_formal, parametro_real);
            case MENOS:
               return evaluarFuncion(def_funcion.getFirstChild(), parametro_formal, parametro_real)
               -
               evaluarFuncion(def_funcion.getFirstChild().getNextSibling(), parametro_formal, parametro_real);
            case POR:
               return evaluarFuncion(def_funcion.getFirstChild(), parametro_formal, parametro_real)
               *
               evaluarFuncion(def_funcion.getFirstChild().getNextSibling(), parametro_formal, parametro_real);
            case DIV:
               return evaluarFuncion(def_funcion.getFirstChild(), parametro_formal, parametro_real)
               /
               evaluarFuncion(def_funcion.getFirstChild().getNextSibling(), parametro_formal, parametro_real);
            case LLAMADA:
               if (tabla_funciones.containsKey(def_funcion.getFirstChild().getText())) {
                  AST aux1 = af.dupTree((AST)tabla_funciones.get(def_funcion.getFirstChild().getText()));
                  AST aux2 = af.dupTree((AST)tabla_parametros.get(def_funcion.getFirstChild().getText()));
                  sustituir(aux1, aux2, parametro_real);
                  return this.expresion(aux1);
               }
               else return 0;
            case IDENT:
               if (def_funcion.getText().equals(parametro_formal.getText()))
                  return parametro_real;
               else
                  return this.expresion(def_funcion);
            case NUMERO:
               return this.expresion(def_funcion);
            default:
               return 0;
         }
      }catch(RecognitionException e)
         { System.out.println("error sintáctico" + e.getMessage());
           return 0;
         }
      }




   // Post: Se devuelve la evaluación de 'def_funcion' previa sustitución de toda ocurrencia del
   // 'parametro_formal' por 'parametro_real'.
   void actualizarVariables(AST variable, AST expresion){
      // Pre: 'variable' es el AST de una variable y 'valor' es el AST de una expresión.
      tabla_variables.put(variable.getText(), af.dupTree(expresion));
      // Post: 'tabla_variables' se actualiza con la entrada (variable, expresion).
   }




   void actualizarFunciones(AST nombre_funcion, AST parametro_formal, AST expresion){
      // Pre: 'nombre_funcion' es el AST correspondiente al nombre de la función.
      // 'parametro_formal' es el AST correspondiente al parámetro formal de la función.
      // 'expresion' es la expresión que define la función.
      tabla_funciones.put(nombre_funcion.getText(), af.dupTree(expresion));
      tabla_parametros.put(nombre_funcion.getText(), af.dupTree(parametro_formal));
      // Post: 'tabla_funciones' se actualiza con la entrada (nombre_funcion, expresion).
      // 'tabla_parametros' se actualiza con la entrada (nombre_funcion, parametro_formal).
   }



}
```

```
ordenes     : #(ORDENES (orden)+)
            ;

orden       {int r;}
            : def_funcion
            | asignacion
            | r=e:expresión
              { System.out.println( "Expresión: " + e.toStringTree()
                                  + " vale " + r
                                  );
              }
            ;

def_funcion {int r;}
            :#(DEF_FUNCION a:IDENT b:IDENT r=c:expresion)
            {
              actualizarFunciones(a, b, c);
              System.out.println( "Función: " + a.getText() + "(" + b.toStringTree() + " )"
                                + " vale " + c.toStringTree()
                                );
            }
            ;

asignacion  {int r;}
            :#(ASIG a:IDENT r=b:expresion)
            {
              actualizarVariables(a, b);
              System.out.println( "Asignación: " + a.getText()
                                + " vale " + r
                                );
            }
            ;

expresion   returns [int valor=0;]
            {int r, s;}
            : #(MAS     r=expresion s=expresion) { valor = r + s; }
            | #(MENOS   r=expresion s=expresion) { valor = r - s; }
            | #(POR     r=expresion s=expresion) { valor = r * s; }
            | #(DIV     r=expresion s=expresion) { valor = r / s; }
            | #(LLAMADA f:IDENT     r=expresion) { valor = evaluarFuncion((AST)tabla_funciones.get(f.getText())
                                                                         ,(AST)tabla_parametros.get(f.getText())
                                                                         ,r
                                                                         );
                                                 }
            | i:IDENT                            { valor = evaluarVariable(i); }
            | j:NUMERO                           { valor = new Integer(j.getText()).intValue(); }
            ;


exception catch [ArithmeticException e] { valor = 0; }
```

5.3.2 LEXCHANGE

Supongamos un lenguaje llamado LEXCHANGE para programar transferencias de datos entre dos bases de datos relacionales. Un programa típico en LEXCHANGE incluye: (a) esquema de datos fuente, (b) datos fuente, (c) esquema de datos destino y un conjunto de restricciones especificando la transferencia de datos entre la fuente y el destino. Todos los datos son de tipo cadena de caracteres. Las restricciones son implicaciones lógicas con antecedente formado por un átomo definido sobre una relación del esquema fuente y un consecuente formado por un átomo definido sobre una relación del esquema destino (ver programa de ejemplo). Hay dos clases de variables en una restricción, las que sólo aparecen en el consecuente y las demás. Las primeras se suponen cuantificadas existencialmente y las segundas universalmente.

Ejemplo. Programa LEXCHANGE.

```
ESQUEMA FUENTE
        estudiante(NOMBRE, NACIMIENTO, DNI)
        empleado(NOMBRE, DNI, TELEFONO)
DATOS FUENTE
        estudiante(Axel,1980,12122345)
        estudiante(Lorenzo,1982,10345321)
        estudiante(Antonio,1979,87654456)
        empleado(Axel,12122345,616234345)
        empleado(Manuel,50545318,617876654)
ESQUEMA DESTINO
        persona(NOMBRE, NACIMIENTO, DNI, TELEFONO)
RESTRICCIONES
        VAR x,y,z,u;
        estudiante(x,y,z) implica persona(x,y,z,u)
        VAR x,y,z,u;
        empleado(x,y,z) implica persona(x,u,y,z)
```

La ejecución del anterior programa transfiere los siguientes datos a la base de datos destino.

```
persona(Axel, 1980, 12122345, X1)
persona(Lorenzo, 1982, 10345321, X2)
persona(Antonio, 1979, 87654456, X3)
persona(Axel, X4, 12122345, 616234345)
persona(Manuel, X5, 50545318, 617876654)
```

Cada dato X_i representa la instanciación de una variable cuantificada existencialmente en una restricción.

SE PIDE: Intérprete de programas LEXCHANGE. Se debe construir la solución con un *Tree-Parser*. Suponer que el análisis léxico-sintáctico se ha resuelto con los lexer y parser propuestos en la solución del problema 2 del laboratorio Análisis Léxico-sintáctico y el análisis semántico se ha resuelto con el parser propuesto en la solución del problema 2 del laboratorio Análisis Semántico.

Solución:

Análisis léxico:

```
class Analex extends Lexer;

options{ importVocab = Anasint; }

tokens{
      ESQUEMA       = "ESQUEMA";
      FUENTE        = "FUENTE";
      DATOS         = "DATOS";
      DESTINO       = "DESTINO";
      RESTRICCIONES = "RESTRICCIONES";
      VAR           = "VAR";
      IMPLICA       = "implica";
}
protected NL     : "\r\n"        {newline();};
protected DIGITO : '0'..'9';
protected LETRA  : 'a'..'z'|'A'..'Z';
          BTF    : (' '|'\t'|NL) {$setType(Token.SKIP);};
          IDENT  : LETRA(LETRA|DIGITO)*;
          NUMERO : (DIGITO)+;
          PA     : '(';
          PC     : ')';
          COMA   : ',';
          PyC    : ';';
```

Análisis sintáctico:

```
class Anasint extends Parser;

options{ buildAST = true; }// Construcción de asa
tokens {
      ENTRADA;
      ESQUEMAFUENTE;
      ESQUEMADESTINO;
      DATOSFUENTE;
      RESTRICCION;
}
entrada        : esquema_fuente datos_fuente esquema_destino restricciones EOF!
                 {#entrada = #(#[ENTRADA, "ENTRADA"], ##);}
               ;
esquema_fuente : ESQUEMA! FUENTE! (signatura)+
                 {#esquema_fuente = #(#[ESQUEMAFUENTE, "ESQUEMA FUENTE"], ##);}
               ;
signatura      : IDENT^ PA! atributos PC!
               ;
atributos      :(IDENT COMA) => IDENT COMA! atributos
               | IDENT
               ;
datos_fuente   : DATOS! FUENTE! (atomo)+
                 {#datos_fuente = #(#[DATOSFUENTE, "DATOS FUENTE"], ##);}
               ;
esquema_destino: ESQUEMA! DESTINO! (signatura)+
                 {#esquema_destino = #(#[ESQUEMADESTINO, "ESQUEMA DESTINO"], ##);}
               ;
restricciones  : RESTRICCIONES^ (restriccion)+ ;
restriccion    : variables implicacion
                {#restriccion = #(#[RESTRICCION, "RESTRICCION"], ##);}
               ;
variables      : VAR^ vars PyC! ;
vars           : (IDENT COMA) => IDENT COMA! vars
               | IDENT
               ;
implicacion    : atomo IMPLICA^ atomo ;
atomo          : IDENT^ PA! terminos PC! ;
terminos       : (termino COMA) => termino COMA! terminos
               | termino
               ;
termino        : IDENT
               | NUMERO
               ;
```

Análisis semántico:

```
header{
     import java.util.*;
     import java.io.*;
     import antlr.*;
}

class Anasem extends TreeParser;

options { importVocab = Anasint; }



{

  FileWriter f;
  int cont=1;


  void abrirFichero(){
     try { f = new FileWriter("salida.txt"); }
     catch(IOException e) { System.out.println(e.toString()); }
  }


  void cerrarFichero(){
     try { f.close(); }
     catch(IOException e) { System.out.println(e.toString()); }
  }


  void escribirFicheroSalida(AST consecuente, Hashtable<String,String> sustituciones){
     //Pre: sustituciones representa las sustituciones llevadas a cabo en el antecedente de la restricción.
     try{
        f.write(consecuente.getText());
        f.write("(");
        AST argumento = consecuente.getFirstChild();
        while (argumento != null){
           boolean enc = false;
           if (sustituciones.containsKey(argumento.getText()))
              f.write(sustituciones.get(argumento.getText()));
           else{
              f.write("X"+cont);
              cont++;
           }
           argumento = argumento.getNextSibling();
           if (argumento != null)
              f.write(",");
        }
        f.write(")\n");
     } catch(IOException e) { System.out.println(e.toString()); }
     //Post: se escribe en salida.txt la instanciación de consecuente según las sustituticiones
     //contenidas en sustituciones.
  }


  Hashtable<String, String> matchingAntecedenteAtomo(AST antecedente, AST atomo){
     //Pre: antecedente y atomo definido sobre el mismo predicado
     Hashtable<String, String> resultado = new Hashtable<String, String>();
     AST termino = atomo.getFirstChild();
     AST variable = antecedente.getFirstChild();
     while (variable != null) {
        resultado.put(variable.getText(), termino.getText());
        System.out.println(variable.getText() + " " + atomo.getText());
        variable = variable.getNextSibling();
        termino = termino.getNextSibling();
     }
     return resultado;
     //Post: cada variable de antecedente se le asocia el término correspondiente de atomo
  }
```

```
  void matchingRestriccionDatosFuentes(AST implicacion, AST datos){
     //Pre: dada una implicación y el conjunto de datos fuente
     System.out.println(implicacion.toStringList());
     System.out.println(datos.toStringList());
     Hashtable<String, String> aux;
     AST atomoFuente = datos.getFirstChild();
     while (atomoFuente != null){
        if ((atomoFuente.getText()).equals((implicacion.getFirstChild()).getText())) {
           aux = matchingAntecedenteAtomo(implicacion.getFirstChild(), atomoFuente);
           System.out.println(aux.toString());
           escribirFicheroSalida(implicacion.getFirstChild().getNextSibling(), aux);
           aux.clear();
        }
        atomoFuente = atomoFuente.getNextSibling();
     }
     //Post: genera el fichero salida.txt aplicado la restricción a cada átomo en datos.
     //Las variables existenciales generan en el fichero de salida variables Xi.
  }
}
```

```
entrada                         : { abrirFichero(); }
                                  #(ENTRADA esquema_fuente a:datos_fuente esquema_destino restricciones[a])
                                  { cerrarFichero(); }
                                ;

esquema_fuente                  : #(ESQUEMAFUENTE (signatura)+)
                                ;

datos_fuente                    : #(DATOSFUENTE (atomo)*)
                                ;

esquema_destino                 : #(ESQUEMADESTINO (signatura)+)
                                ;

restricciones                     [AST datosFuentes]
                                : #(RESTRICCIONES (restriccion[datosFuentes])+)
                                ;

restriccion                       [AST datosFuentes]
                                : #(RESTRICCION variables a:implicacion)
                                  { matchingRestriccionDatosFuentes(a, datosFuentes); }
                                ;

signatura                       : #(IDENT (IDENT)*)
                                ;

variables                       : #(VAR (IDENT)*)
                                ;

implicacion                     : #(IMPLICA atomo atomo)
                                ;

atomo                           : #(IDENT (termino)*)
                                ;

termino                         : IDENT
                                | NUMERO
                                ;
```

Análisis semántico (Solución alternativa):

```
header{ import java.util.*; }

class Anasem extends TreeParser;

options{ importVocab = Anasint; }

{
   String astToString(AST asa){
      String resultado = new String("");
      resultado += asa.getText();
      resultado += "(";
      AST aux = asa.getFirstChild();
      while (aux != null) {
         resultado += aux.getText();
         aux = aux.getNextSibling();
         if (aux != null) resultado += ",";
      }
      resultado += ")";
      return resultado;
   }

   void verificar(int error, AST asa){
      switch(error){
         case 0: break;
         case 1: System.out.println("ERROR (tabla no declarada): " + asa.getText());
                 break;
         case 2: System.out.println("ERROR (número de atributos incorrecto) en tupla: " + astToString(asa));
                 break;
      }
   }
}

entrada         {Hashtable<String,Integer> almacen;}
                : #(ENTRADA almacen=esquema_fuente datos_fuente[almacen] ...)
                ;
esquema_fuente  returns [Hashtable<String,Integer> almacen = new Hashtable<String,Integer>()]
                : #(ESQUEMAFUENTE (signatura[almacen])+)
                ;
datos_fuente    [Hashtable<String,Integer> almacen]
                {int error;}
                : #(DATOSFUENTE (error=a:atomo[almacen] {verificar(error,a);})+)
                ;
esquema_destino [Hashtable<String,Integer> almacen]
                : #(ESQUEMADESTINO (signatura[almacen])+)
                ;
restricciones   [Hashtable<String,Integer> almacen]
                : #(RESTRICCIONES  (restriccion[almacen])+)
                ;
signatura       [Hashtable<String,Integer> almacen]
                : #(a:IDENT {int cont=0;} (IDENT {cont++;})+)
                {almacen.put(a.getText(),cont);}
                ;
atomo           [Hashtable<String,Integer> almacen] returns [int error=0]
                : #(a:IDENT {int cont=0;} (termino {cont++;})+)
                {  if (almacen.containsKey(a.getText()))
                      if (almacen.get(a.getText()) == cont)
                          error = 0;
                      else
                          error = 2;
                   else
                      error = 1;
                }
                ;
restriccion     [Hashtable<String,Integer> almacen]
                : #(RESTRICCION variables implicacion[almacen])
                ;
variables       : #(VAR (IDENT)+)
                ;
implicacion     [Hashtable<String,Integer> almacen]
                {int error;}
                : #(IMPLICA error=atomo[almacen] error=atomo[almacen])
                ;
termino         : IDENT
                | NUMERO
                ;
```

5.3.3 OWL

OWL es un lenguaje diseñado para expresar ontologías en la web. Una ontología es un conjunto de declaraciones que describen un dominio de interés. En este ejercicio se consideran 4 tipos de declaraciones:

Asertos de clase: Formalizan la pertenencia de un individuo a una clase de individuos.

Ejemplo:
```
ClassAssertion(Person Mary) // Mary es una persona
ClassAssertion(Woman Mary)  // Mary es una mujer
```

Asertos de propiedad: Formalizan una determinada relación entre dos individuos.

Ejemplo:
```
ObjectPropertyAssertion(hasWife John Mary) // Mary es la esposa de John
```

Jerarquía de clase: Formalizan una relación de inclusión entre dos clases de individuos.

Ejemplo:
```
SubClassOf(Mother Woman) //Toda madre es una mujer
```

Definición de clase: Formaliza la definición de una clase primitiva.

Ejemplo:
```
EquivalentClasses(Person Human) //persona (clase primitiva) equivale a ser humano
```

Aparte de las clases primitivas tales como `Woman` o `Person`, OWL dispone de operadores adicionales para formalizar el concepto clase de individuo:

- Clase de individuo expresada mediante operadores lógicos de *intersección*, *unión* y *complemento*:

  Ejemplos:
  ```
  ObjectIntersectionOf(Woman Mother)       // individuos que son mujeres y madres
  ObjectUnionOf(Father Mother GrandFather) // individuos que son padres, madres o abuelos
  ObjectComplementOf(Father)               // individuos que no son padres
  ```

- Clase expresada mediante *restricción sobre propiedad*:

  Ejemplos:
  ```
  ObjectSomeValuesFrom(hasChild Happy)     // individuos con algún hijo feliz
  ObjectAllValuesFrom(hasChild Happy)      // individuos con todos sus hijos felices
  ```

Ejemplos de ontología OWL:

```
ClassAssertion(Person Mary) // Mary es una persona

SubClassOf(Woman Person)    // Toda mujer es una persona

SubClassOf(Mother Woman)    // Toda madre es una mujer

ClassAssertion(Person John) // John es una persona

SubClassOf(HappyMother ObjectIntersectionOf(Mother ObjectAllValuesFrom(hasChild Happy)))
// Madre feliz es una madre con todos sus hijos felices
```

SE PIDE:

(1) Construir un *TreeParser* ANTLR que reconozca los árboles de sintaxis abstracta generados por el *Parser* ANTLR dado.

(2) Anotar el *TreeParser* del apartado (1) para que compile en formulas lógicas una ontología OWL. La compilación de una ontología OWL se define como la compilación de cada una de sus declaraciones. La compilación de una declaración se define formalmente de la siguiente manera:

(a) `ClassAssertion(C I)` Se compila a la formula lógica: `C(I).`

(b) `ObjectPropertyAssertion(P I1 I2)` Se compila a la formula lógica: `P(I1,I2).`

(c) `SubClassOf(C1 C2)` Se compila a la formula lógica: `forall x (F(C1,x) implies F(C2,x)).`

(d) `EquivalentClasses(C1 C2)` Se compila a la formula lógica: `forall x (F(C1,x) equiv F(C2,x)).`

Siendo:

```
F(C, x)                              =   C(x)                                  // si C una clase primitiva.
F(ObjectSomeValuesFrom(P C), x)      =   exists y (P(x,y) and F(C,y))          // siendo y una variable nueva.
F(ObjectAllValuesFrom(P C), x)       =   forall y (P(x,y) implies F(C,y))      // siendo y una variable nueva.
F(ObjectUnionOf(C1,…,Cn), x)         =   F(C1, x) or … or F(Cn, x)
F(ObjectIntersectionOf(C1,…,Cn), x) =   F(C1, x) and … and F(Cn, x)
F(ObjectComplementOf(C), x)          =   not F(C, x)
```

Ejemplo de traducción:

Ontología original:

```
ClassAssertion(Person Mary)
SubClassOf(Woman Person)
SubClassOf(Mother Woman)
ClassAssertion(Person John)
EquivalentClasses(HappyMother ObjectIntersectionOf(Mother ObjectAllValuesFrom(hasChild Happy)))
```

Fórmulas lógicas equivalentes:

```
Person(Mary)
forall x0 (Woman(x0) implies Person(x0))
forall x0 (Mother(x0) implies Woman(x0))
Person(John)
forall x0 (HappyMother(x0) equiv Mother(x0) and forall x1 (hasChild(x0,x1) implies Happy(x1)))
```

Dado el siguiente *parser* ANTLR para OWL:

```
class Anasint extends Parser;

options{
      buildAST = true;
}

tokens{
      ONTOLOGY;
}

ontologia               : (declaracion)* EOF!
                          { #ontologia = #(#[ONTOLOGY,"Ontology"], ##); }
                        ;

declaracion             : aserto_clase
                        | aserto_propiedad
                        | subclase
                        | definicion
                        ;

aserto_clase            : CLASS_ASSERTION^ PA! clase_primitiva individuo PC!
                        ;

aserto_propiedad        : OBJECT_PROPERTY_ASSERTION^ PA! propiedad individuo individuo PC!
                        ;

subclase                : SUB_CLASS_OF^ PA! clase clase PC!
                        ;

definicion              : EQUIVALENT_CLASSES^ PA! clase_primitiva clase PC!
                        ;

individuo               : IDENT
                        ;

propiedad               : IDENT
                        ;

clase                   : clase_primitiva
                        | restriccion_existencial
                        | restriccion_universal
                        | union
                        | interseccion
                        | complementario
                        ;

clase_primitiva         : IDENT
                        ;

restriccion_existencial : OBJECT_SOME_VALUES_FROM^ PA! propiedad clase PC!
                        ;

restriccion_universal   : OBJECT_ALL_VALUES_FROM^ PA! propiedad clase PC!
                        ;

union                   : OBJECT_UNION_OF^ PA! clase (clase)+ PC!
                        ;

interseccion            : OBJECT_INTERSECTION_OF^ PA! clase (clase)+ PC!
                        ;

complementario          : OBJECT_COMPLEMENT_OF^ PA! clase PC!
                        ;
```

Solución:

Apartado (1):

```
class Anasint extends TreeParser;

options{
      importVocab = Anasint;
}

ontologia               : #(ONTOLOGY (declaracion)*)
                        ;

declaracion             : aserto_clase
                        | aserto_propiedad
                        | subclase
                        | definicion
                        ;

aserto_clase            : #(CLASS_ASSERTION individuo clase)
                        ;

aserto_propiedad        : #(OBJECT_PROPERTY_ASSERTION propiedad individuo individuo)
                        ;

subclase                : #(SUB_CLASS_OF clase clase)
                        ;

definicion              : #(EQUIVALENT_CLASSES clase_primitiva clase)
                        ;

individuo               : IDENT
                        ;

propiedad               : IDENT
                        ;

clase                   : clase_primitiva
                        | restriccion_existencial
                        | restriccion_universal
                        | union
                        | interseccion
                        | complementario
                        ;

clase_primitiva         : IDENT
                        ;

restriccion_existencial : #(OBJECT_SOME_VALUES_FROM propiedad clase)
                        ;

restriccion_universal   : #(OBJECT_ALL_VALUES_FROM propiedad clase)
                        ;

union                   : #(OBJECT_UNION_OF clase (clase)+)
                        ;

interseccion            : #(OBJECT_INTERSECTION_OF clase (clase)+)
                        ;

complementario          : #(OBJECT_COMPLEMENT_OF clase)
                        ;
```

Apartado (2):

```
header{ import java.util.*;
        import java.io.*;
        import antlr.*;
}
class Anasint extends TreeParser;
options { importVocab = Anasint; }

{ FileWriter f; }

ontologia  {String r;}
           { try{f = new FileWriter("output");} catch(IOException e){} }
    : #(ONTOLOGY ( r=declaracion { try{f.write(r);} catch(IOException e){} } )*)
           { try{f.close();} catch(IOException e){} }
    ;
declaracion       returns [String r = null]
    : r = aserto_clase
    | r = aserto_propiedad
    | r = subclase[0]
    | r = definicion[0]
    ;
aserto_clase      returns [String r = null]
    : #(CLASS_ASSERTION c:IDENT i:IDENT)
                  { r = new String(c.getText() + "(" + i.getText() + ")\n");}
    ;
aserto_propiedad  returns [String r = null]
    : #(OBJECT_PROPERTY_ASSERTION p:IDENT i1:IDENT i2:IDENT)
                  { r = new String(p.getText() + "(" + i1.getText() + "," + i2.getText() + ")\n");}
    ;
subclase          [int cont]   returns [String r = null]  {String c1, c2;}
    : #(SUB_CLASS_OF c1=clase[cont] c2=clase[cont])
                  {r = new String("forall x" + cont + "(" + c1 + " implies " + c2 + ")\n");}
    ;
definicion        [int cont]   returns [String r = null]  {String c1, c2;}
    : #(EQUIVALENT_CLASSES c1=clase_primitiva[cont] c2=clase[cont])
                  {r = new String("forall x" + cont + "(" + c1 + " equiv " + c2 + ")\n");}
    ;
individuo         returns [String r = null]
    : a:IDENT
                  {r = new String(a.getText());}
    ;
propiedad         returns [String r = null]
    : a:IDENT
                  {r = new String(a.getText());}
    ;
clase [int cont] returns [String r = null] : r = clase_primitiva[cont]
                                           | r = restriccion_existencial[cont]
                                           | r = restriccion_universal[cont]
                                           | r = union[cont]
                                           | r = interseccion[cont]
                                           | r = complementario[cont]
    ;
clase_primitiva    [int cont]   returns [String r = null]
    : a:IDENT
                   {r = new String(a.getText() + "(x"+cont+")");}
    ;
restriccion_existencial [int cont]   returns [String r = null]  {String p, c;}
    : #(OBJECT_SOME_VALUES_FROM p=propiedad c=clase[cont+1])
                  {r = new String("exists x" + (cont+1)+"("+p+"(x"+cont+", x"+(cont+1)+") and "+c +")");}
    ;
restriccion_universal   [int cont]   returns [String r = null]  {String p, c;}
    : #(OBJECT_ALL_VALUES_FROM p=propiedad c=clase[cont+1])
                  {r=new String("forall x" + (cont+1)+"("+p+"(x"+cont+", x"+(cont+1)+") implies "+c+")");}
    ;
union             [int cont]   returns [String r = null]  {String c1, c2, c3, aux;}
    : #(OBJECT_UNION_OF c1=clase[cont] {c2 = new String();} (c2=clase[cont] {c2 = " or " + c2;} )+)
                  {r = new String("(" + c1 + c2 + ")");}
    ;
interseccion      [int cont]   returns [String r = null]  {String c1, c2, c3, aux;}
    : #(OBJECT_INTERSECTION_OF c1=clase[cont] {c2 = new String();} (c2=clase[cont] {c2 = " and " + c2;} )+)
                  {r = new String("(" + c1 + c2 + ")");}
    ;
complementario    [int cont]   returns [String r = null]
    : #(OBJECT_COMPLEMENT_OF r=clase[cont])
                  {r = new String("not" + r);}
    ;
```

6 Exámenes.

6.1 Exámen 2012-2013 Turno mañana.

ANÁLISIS SEMÁNTICO

Supongamos un lenguaje de programación llamado DIN. La principal característica de DIN es que sus variables pueden cambiar de tipo en tiempo de ejecución (es lo que se conoce como sistema de tipos dinámico). Un programa DIN es una secuencia de asignaciones cumpliendo las siguientes reglas.

(Regla 1) La variable a la que se le asigna una expresión adopta el tipo de dicha expresión.

(Regla 2) Una expresión DIN es de tipo entero si y sólo si sus operadores (si los tiene) pertenecen al conjunto {MAS, MENOS, POR}, sus constantes (si las tiene) son enteras y sus variables (si las tiene) son enteras o indefinidas.

(Regla 3) Una expresión DIN es de tipo booleana si y sólo si sus operadores (si los tiene) pertenecen al conjunto {Y, O, NO}, sus constantes (si las tiene) son booleanas y sus variables (si las tiene) son booleanas o indefinidas.

(Regla 4) Toda expresión DIN que no es de tipo entero ni booleano es de tipo indefinido.

(Regla 5) Si una expresión es de tipo entero entonces todas las variables indefinidas de dicha expresión cambian su tipo a tipo entero.

(Regla 6) Si una expresión es de tipo booleano entonces todas las variables indefinidas de dicha expresión cambian su tipo a tipo booleano.

(Regla 7) Una variable a la que no se le ha asignado nunca ninguna expresión o no se ha utilizado en ninguna expresión de tipo entero o booleano es de tipo indefinido.

Ejemplo de programa DIN :

```
PROGRAMA
x = 1;            // 1 es una expresión de tipo entero por Regla 2.
                      // x es de tipo entero por la Regla 1.
v = FALSO;        // FALSO es una expresión de tipo booleano por la Regla 3.
                      // v es de tipo booleano por la Regla 1.
z = v Y r;        // la expresión v Y r es de tipo booleano por las Regla 3.
                      // el tipo de r cambia de indefinido (Regla 7) a booleano por la Regla 6.
                      // z es de tipo booleano por la Regla 1.
a = v MAS c;      // la expresión v MAS c es de tipo indefinido por la Regla 4.
                      // el tipo de c es indefinido por la Regla 7.
                      // a es de tipo indefinido por la Regla 1.
v = 1;            // 1 es una expresión de tipo entero por Regla 2.
                      // v es de tipo entero por la Regla 1.
a = v MENOS c;    // la expresión v MENOS c es de tipo entero por la Regla 2.
                      // a es de tipo entero por la Regla 1.
t = w;            // la expresión w es de tipo indefinido por la Regla 4.
                      // el tipo de w es indefinido por la Regla 7.
                      // t es de tipo indefinido por la Regla 1.
```

Se pide: anotar DE FORMA LEGIBLE el siguiente tree-parser para que detecte expresiones de tipo indefinido. Al detectar una expresión indefinida se debe dar aviso por pantalla.

```
programa   : #(PROGRAMA (asignacion)*)
           ;
asignacion : #(ASIG IDENT expresion)
           ;
expresion  : #(MAS   expresion expresion)
           | #(MENOS expresion expresion)
           | #(POR   expresion expresion)
           | #(Y     expresion expresion)
           | #(O     expresion expresion)
           | #(NO    expresion)
           | NUMERO
           | IDENT
           | FALSO
           | CIERTO
           ;
```

INTERPRETACIÓN

Supongamos un lenguaje LIST diseñado para expresar listas de números.

Ejemplo de listas LIST:

```
([1,2])++[3,5]++[4]
[2|[3]]++[5]
[1|[2|[1,3]]]
[2|([3|[4]]++[5])]
[2,3,5]
[]
```

La reducción de una lista LIST produce una lista sin operadores.

++ Concatena listas.

| Añade un número como primer elemento de la lista.

Las reglas para reducir una lista LIST L son las siguientes:

(Regla 1) Si L = [] entonces su reducción es [].

(Regla 2) Si L = [n_0 ,n_1, …, n_k], siendo n_0, n_1, …,n_k números, entonces su reducción es [n_0, n_1, …, n_k].

(Regla 3) Si L = [n_0|s], siendo s una lista, entonces la reducción de L es [n_0, n_1, …, n_k] siendo la reducción de s igual a [n_1, …, n_k].

(Regla 4) Si L = m++s, siendo m y s listas, entonces la reducción de L es [n_0, …, n_i, n_j, …, n_k] siendo la reducción de m igual a [n_0, …, n_i] y la reducción de s igual a [n_j, …, n_k].

Ejemplo de reducciones:

```
([1,2])++[3,5]++[4]     se reduce a [1,2,3,5,4]
[2|[3]]++[5]            se reduce a [2,3,5]
[1|[2|[1,3]]]           se reduce a [1,2,1,3]
[2|([3|[4]]++[5])]      se reduce a [2,3,4,5]
[2,3,5]                 se reduce a [2,3,5]
[]                      se reduce a []
```

Se pide: atribuir DE FORMA LEGIBLE el siguiente tree-parser para que sintetice una lista Java de números enteros (List<Integer>) con la reducción de una lista LIST.

Para programar esta lista Java puede utilizar los siguientes métodos declarados en List<Integer>:

- Añadir un entero *e* al final de la lista L: L.add(*e*)
- Añadir un entero *e* a la lista L en la posición *p*: L.add(*p*, *e*)
- Añadir al final de la lista L todos los enteros de la lista L1: L.addAll(L1)

```
lista        : #(CONCATENAR lista lista)
             | #(ANADIR NUMERO lista)
             | (#(CA NUMERO)) => #(CA lista_numeros CC)
             | CA CC
             ;

lista_numeros: (NUMERO)+
             ;
```

Solución:

ANÁLISIS SEMÁNTICO

```
header{
   import java.util.*;
}

class Anasint2Semantico extends TreeParser;

options{
   importVocab = AnasintSemantico;
}

{
   // almacén para recordar el tipo de cada variable
   //    0: indefinido, 1: entero y 2: booleano.
   Map<String,Integer> variables = new HashMap<String,Integer>();

   //Calcular el tipo de una expresión con operador binario.
   //    si el operador binario es entero entonces tipo = 1
   //    si el operador binario es logico entonces tipo = 2
   Integer calcular_tipo(Integer tipo1, Integer tipo2, Integer tipo){
      Integer r = 0;
      if (tipo == 1)
         if (tipo1==2 | tipo2==2)  r = 0;
         else  r = 1;
      else
         if (tipo1==1 | tipo2==1)  r = 0;
         else  r = 2;

        return r;
   }

   void actualizarTipoVariablesIndefinidas(Set<String> varsIndef, Integer tipo){
      for(String v: varsIndef)
         variables.put(v, tipo);
   }
}



programa : #(PROGRAMA (asignacion)*)
         ;

asignacion {Integer tipo; Set<String> varsIndef = new HashSet<String>();}
          :#(ASIG i:IDENT tipo=e:expresion[varsIndef])
           { actualizarTipoVariablesIndefinidas(varsIndef, tipo);
             variables.put(i.getText(), tipo);
             if (tipo == 0) System.out.println("Expresión " + e.toStringTree() + " es de tipo indefinido");
           }
          ;

expresion [Set<String> varsIndef] returns[Integer tipo = 0;] {Integer tipo1, tipo2;}
       : #(MAS   tipo1=expresion[varsIndef] tipo2=expresion[varsIndef]) {tipo=calcular_tipo(tipo1, tipo2, 1);}
       | #(MENOS tipo1=expresion[varsIndef] tipo2=expresion[varsIndef]) {tipo=calcular_tipo(tipo1, tipo2, 1);}
       | #(POR   tipo1=expresion[varsIndef] tipo2=expresion[varsIndef]) {tipo=calcular_tipo(tipo1, tipo2, 1);}
       | #(Y     tipo1=expresion[varsIndef] tipo2=expresion[varsIndef]) {tipo=calcular_tipo(tipo1, tipo2, 2);}
       | #(O     tipo1=expresion[varsIndef] tipo2=expresion[varsIndef]) {tipo=calcular_tipo(tipo1, tipo2, 2);}
       | #(NO    tipo1=expresion[varsIndef])                            {tipo=calcular_tipo(tipo1, 2,     2);}
       | NUMERO  {tipo = 1;}
       | i:IDENT {
                   if (!variables.containsKey(i.getText())) tipo = 0;
                   else                                     tipo = variables.get(i.getText());
                   if (tipo == 0) varsIndef.add(i.getText());
                 }
       | FALSO   {tipo = 2;}
       | CIERTO  {tipo = 2;}
       ;
```

```
header{
   import java.util.*;
}

class Anasint2Interprete extends TreeParser;

options{
   importVocab = AnasintInterprete;
}

lista              returns [List<Integer> l = new LinkedList<Integer>()]
                   {List l1, l2;}
                :  #(CONCATENAR l1=lista l2=lista)
                   {
                    l.addAll(l1);
                    l.addAll(l2);
                   }
                |  #(ANADIR n:NUMERO l1=lista)
                   {
                    l.add(new Integer(n.getText()));
                    l.addAll(l1);
                   }
                |  (#(CA NUMERO)) => #(CA lista_numeros[l] CC)
                |  CA CC
                ;

lista_numeros      [List l]
                :  (n:NUMERO {l.add(new Integer(n.getText()));} )+
                ;
```

6.2 Exámen 2012-2013 Turno tarde.

ANÁLISIS SEMÁNTICO

Supongamos un lenguaje de programación llamado BLQ. La principal característica de BLQ es la capacidad de programar con bloques. Un bloque está formado por declaraciones de variables y una secuencia de instrucciones. La instrucción pueden ser un bloque o una asignación.

Todo programa BLQ cumple las siguientes reglas.

(Regla 1) Toda variable declarada en un bloque es visible para las instrucciones contenidas en éste. En caso de ambigüedad se sigue el criterio de "la declaración más próxima": si una variable x es visible en un punto del programa con dos o más declaraciones entonces se considera sólo la declaración más cercana a dicho punto.

(Regla 2) Toda variable declarada en un bloque no es visible en el exterior a dicho bloque.

(Regla 3) Si una variable no es visible en un bloque entonces su tipo es indefinido en dicho bloque.

(Regla 4) Una expresión es de tipo entero si y sólo si sus operadores (si los tiene) pertenecen al conjunto {MAS, MENOS, POR}, sus constantes (si las tiene) son de tipo entero y sus variables (si las tiene) son de tipo entero.

(Regla 5) Una expresión es de tipo booleano si y sólo si sus operadores (si los tiene) pertenecen al conjunto {Y, O, NO}, sus constantes (si las tiene) son de tipo booleano y sus variables (si las tiene) son de tipo booleano.

(Regla 6) Si una expresión no es entera ni booleana entonces es de tipo indefinido.

Una asignación var = expr es correcta si y sólo si el tipo de var coincide con el tipo de expr y no es indefinido.

Ejemplo de programa BLQ:

```
PROGRAMA
BLOQUE
    ENTERO x,y,z;
    y = 2+x+z;      // sí correcta (y es entera  por Regla 1, 2+x+z es entera por Reglas 1 y 4)
    BLOQUE
        ENTERO m;
        BOOLEANO x,y;
        y = 1;      // no correcta (y es booleana por Regla 1, 1 es entera por Regla  4)
        x = z;      // no correcta (x es booleana por Regla 1, z es entera por Reglas 1 y 4)
        m = z;      // sí correcta (m es entera   por Regla 1, z es entera por Reglas 1 y 4)
        m = 2;
    FBLOQUE
    m = 2;          // no correcta (m es indefinida por Regla 2)
FBLOQUE
```

Se pide:

Anotar DE FORMA LEGIBLE el siguiente tree-parser para que detecte asignaciones no correctas. Al detectar una asignación no correcta se debe dar aviso por pantalla.

```
programa       : #(PROGRAMA variables instrucciones) ;
variables      : (declaracion)* ;
declaracion    : #(ENTERO listaVariables) | #(BOOLEANO listaVariables) ;
listaVariables : (IDENT)+ ;
instrucciones  : (instruccion)* ;
instruccion    : asignacion | bloque ;
asignacion     : #(ASIG IDENT expresion) ;
bloque         : #(BLOQUE variables instrucciones) ;
expresion      : #(MAS expresion expresion)
               | #(MENOS expresion expresion)
               | #(POR expresion expresion)
               | #(Y expresion expresion)
               | #(O expresion expresion)
               | #(NO expresion)
               | NUMERO | IDENT | FALSO | CIERTO
               ;
```

INTERPRETACIÓN

Supongamos un lenguaje STACK diseñado para expresar pilas de números.

Ejemplo de pilas STACK:

```
1+(3,5)
(2+(3))-
1+(2+(1,3))-
(2,3,5)
()
```

La reducción de una pila STACK produce una pila sin operadores +(apilar un número en la pila) y -(desapilar).

Las reglas para reducir una pila STACK p son las siguientes:

(Regla 1) Si p = ( ) entonces su reducción es ( ).

(Regla 2) Si p = (n_1, …, n_k), siendo n_1, …, n_k números, entonces su reducción es (n_1, …, n_k).

(Regla 3) Si p = n_0+s, siendo s una pila, entonces la reducción de p es igual a(n_0, n_1, …, n_k), siendo (n_1, …, n_k), la reducción de s.

(Regla 4) Si p = s-, siendo s = (n_0 ,…, n_k), entonces la reducción de p es igual a (n_1 ,…, n_k).

Ejemplo de reducciones:

```
1+(3,5)          se reduce a (1,3,5)
(2+(3))-         se reduce a (3)
1+(2+(1,3))-     se reduce a (1,1,3)
(2,3,5)          se reduce a (2,3,5)
()               se reduce a ()
```

Se pide: atribuir DE FORMA LEGIBLE el siguiente tree-parser para que sintetice una lista Java de números enteros (List<Integer>) con la reducción de una pila STACK.

Para programar esta lista Java puede utilizar los siguientes métodos declarados en List<Integer>:

- Añadir un entero *e* a la lista L en la posición *p*: L.add(*p*, *e*)
- Eliminar el entero de la lista L de la posición *p*: L.remove(*p*)
- Calcular el tamaño de la lista L: L.size()

```
pila         : #(APILAR NUMERO pila)
             | #(DESAPILAR pila)
             | (#(PA NUMERO)) => #(PA lista_numeros PC)
             | #(PA PC)
             ;

lista_numeros: (NUMERO)+
             ;
```

Solución:

ANÁLISIS SEMÁNTICO

```
header{
   import java.util.*;
}

class Anasint2Semantico extends TreeParser;

options{
   importVocab = AnasintSemantico;
}

tokens{
   INDEF;
}


{
   // lista que almacena las variables visibles en cada punto del programa
   List<Map<String,Integer>> lvars = new LinkedList<Map<String,Integer>>();

   Integer calcularTipo(Integer t1, Integer t2){
      if (t1 == t2)
         if (t1 != INDEF)
            return t1;
         else
            return INDEF;
      else
         return INDEF;
   }


   Integer buscarTipo(AST ident){
     Iterator<Map<String,Integer>> it = lvars.listIterator();
     boolean enc = false;
     Integer tipo = 0;
     while (!enc & it.hasNext()){
        Map<String,Integer> declaracion = it.next();
        if (declaracion.containsKey(ident.getText())){
           enc = true;
           tipo = declaracion.get(ident.getText());
        }
     }
     if (!enc) tipo = INDEF;
     return tipo;
   }

}
```

```
programa                    : #(PROGRAMA variables instrucciones)
                              {lvars.remove(0);}
                            ;

variables                     {Map<String,Integer> vars = new HashMap<String,Integer>();}
                            : (declaracion[vars])*
                              {lvars.add(0,vars);}
                            ;

declaracion                   [Map<String, Integer> vars]
                            : #(a:ENTERO   listaVariables[vars, a])
                            | #(b:BOOLEANO listaVariables[vars, b])
                            ;

listaVariables                [Map<String, Integer> vars, AST tipo]
                            : (a:IDENT {vars.put(a.getText(), tipo.getType());} )+
                            ;

instrucciones               : (instruccion)*
                            ;

instruccion                 : asignacion
                            | bloque
                            ;

asignacion                    {Integer t1,t2;}
                            : #(ASIG a:IDENT t2=e:expresion)
                              {
                               t1 = buscarTipo(a);
                               if (t1!=t2 | t1==INDEF | t2 ==INDEF)
                                  System.out.println( "Asignación no correcta: " + a.toStringTree()
                                                      + " = " + e.toStringTree()
                                                      );
                               }
                            ;

bloque                      : #(BLOQUE variables instrucciones)
                              {lvars.remove(0);}
                            ;

Expresión                   returns [Integer tipo=0;]
                            {Integer t1, t2;}
                            : #(MAS    t1=expresion t2=expresion) {tipo = calcularTipo(t1, t2)      ;}
                            | #(MENOS  t1=expresion t2=expresion) {tipo = calcularTipo(t1, t2)      ;}
                            | #(POR    t1=expresion t2=expresion) {tipo = calcularTipo(t1, t2)      ;}
                            | #(Y      t1=expresion t2=expresion) {tipo = calcularTipo(t1, t2)      ;}
                            | #(O      t1=expresion t2=expresion) {tipo = calcularTipo(t1, t2)      ;}
                            | #(NO     t1=expresion)              {tipo = calcularTipo(t1, BOOLEANO);}
                            | NUMERO  {tipo = ENTERO       ;}
                            | a:IDENT {tipo = buscarTipo(a);}
                            | FALSO   {tipo = BOOLEANO     ;}
                            | CIERTO  {tipo = BOOLEANO     ;}
                            ;
```

INTERPRETACIÓN

```
header{
   import java.util.*;
}
class Anasint2Interprete extends TreeParser;

options{
   importVocab = AnasintInterprete;
}

pila returns[List<Integer> s=new LinkedList<Integer>()] :
                          #(APILAR n:NUMERO s=pila) {s.add(0, new Integer(n.getText()));}
                        |  #(DESAPILAR s=pila)       {if (s.size()>0) s.remove(0);}
                        | (#(PA NUMERO)) => #(PA lista_numeros[s] PC)
                        |  #(PA PC)
                        ;

lista_numeros [List s]: (n:NUMERO {s.add(new Integer(n.getText()));} )+
                        ;
```

6.3 Exámen 2013-2014 Tema 2.

EL LENGUAJE DE LAS EXPRESIONES CONTEXTUALIZADAS (LEC).

ANÁLISIS LÉXICO-SINTÁCTICO.

El siguiente ejemplo muestra un programa en LEC:

```
CONTEXTO 1
    x = 5;
    y = 7;
    z = 1;

CONTEXTO 2
    x = 9;
    z = 3;

DEFECTO 1

EXPRESIONES
    (x{10} - y{2}) * z;
    (x{1} + y{1}) * z{2};
    (x - y) * z;
```

- Un programa LEC es una secuencia de expresiones aritméticas contextualizadas.
- Los contextos son secciones del programa en las que se inicializan las variables con constantes numéricas (ej. x = 5).
- Un programa LEC puede tener uno o más contextos.
- Todo programa LEC tiene un contexto por defecto (ej. DEFECTO 1).
- Las expresiones LEC se construyen con operadores aritméticos, constantes numéricas y variables.
- Los operadores aritméticos son la suma (+), la resta (-) y el producto (*).
- Las constantes numéricas son enteros positivos o cero.
- La variable LEC puede referenciar a un contexto (ej. z{2} es una variable del contexto 2) o al contexto por defecto (ej. x).

SE PIDE:

(1) Gramática independiente de tecnología que especifique la sintaxis de un programa LEC.

(2) Analizador Sintáctico ANTLR para reconocer un programa LEC.

(3) Analizador Léxico ANTLR para reconocer los lexemas de un programa LEC.

Solución:

(1)

```
programa     : contextos defecto expresiones
             ;
contextos    : (contexto)+
             ;
contexto     : CONTEXTO NUMERO asignaciones
             ;
asignaciones : (asignacion)+
             ;
asignacion   : VARIABLE ASIG NUMERO PUNTOYCOMA
             ;
defecto      : DEFECTO NUMERO
             ;
expresiones  : EXPRESIONES (expresion PUNTOYCOMA)*
             ;
expresion    : expresion1 ((MAS|MENOS) expresion)?
             ;
expresion1   : expresion2 (POR expresion1)?
             ;
expresion2   : VARIABLE LLAVEABIERTA NUMERO LLAVECERRADA
             | VARIABLE
             | NUMERO
             | PARENTESISABIERTO expresion PARENTESISCERRADO
             ;
```

(2)

```
class Anasint extends Parser;

programa     : contextos defecto expresiones EOF
             ;
contextos    : (contexto)+
             ;
contexto     : CONTEXTO NUMERO asignaciones
             ;
asignaciones : (asignacion)+
             ;
asignacion   : VARIABLE ASIG NUMERO PUNTOYCOMA
             ;
defecto      : DEFECTO NUMERO
             ;
expresiones  : EXPRESIONES (expresion PUNTOYCOMA)*
             ;
expresion    : expresion1 ((MAS|MENOS) expresion)?
             ;
expresion1   : expresion2 (POR expresion1)?
             ;
expresion2   : (VARIABLE LLAVEABIERTA) => VARIABLE LLAVEABIERTA NUMERO LLAVECERRADA
             | VARIABLE
             | NUMERO
             | PARENTESISABIERTO expresion PARENTESISCERRADO
             ;
```

(3)

```
class Analex extends Lexer;
options{
     importVocab = Anasint;
}
protected NL     : "\r\n"         {newline();};
BTF              : (' '|'\t'| NL) {$setType(Token.SKIP);};
VARIABLE         : 'a'..'z'     ;
CONTEXTO         : "CONTEXTO"   ;
DEFECTO          : "DEFECTO"    ;
EXPRESIONES      : "EXPRESIONES";
NUMERO           : ('0'..'9')+  ;
PARENTESISABIERTO: '(';
PARENTESISCERRADO: ')';
MAS              : '+';
MENOS            : '-';
POR              : '*';
ASIG             : '=';
LLAVEABIERTA     : '{';
LLAVECERRADA     : '}';
PUNTOYCOMA       : ';';
```

6.4 Exámen 2013-2014 Tema 3.

EL LENGUAJE DE LAS EXPRESIONES CONTEXTUALIZADAS (LEC).

ANÁLISIS SEMÁNTICO.

El siguiente ejemplo muestra un programa en LEC:

```
CONTEXTO 1
    x = 5;
    y = 7;
    z = 1;

CONTEXTO 2
    x = 9;
    z = 3;

DEFECTO 1

EXPRESIONES
    (x{10} - y{2}) * z;
    (x{1} + y{1}) * z{2};
    (x - y) * z;
```

- Un programa LEC es una secuencia de expresiones aritméticas contextualizadas.
- Los contextos son secciones del programa en las que se inicializan las variables con constantes numéricas (ej. x = 5).
- Un programa LEC puede tener uno o más contextos.
- Todo programa LEC tiene un contexto por defecto (ej. DEFECTO 1).
- Las expresiones LEC se construyen con operadores aritméticos, constantes numéricas y variables.
- Los operadores aritméticos son la suma (+), la resta (-) y el producto (*).
- Las constantes numéricas son enteros positivos o cero.
- La variable LEC puede referenciar a un contexto (ej. z{2} es una variable del contexto 2) o al contexto por defecto (ej. x).

El lenguaje LEC define un conjunto de restricciones semánticas:

(1) Ninguna variable puede inicializarse más de una vez en un mismo contexto.
(2) No se permite seleccionar un contexto por defecto inexistente (ej. DEFECTO 7 en nuestro programa de ejemplo).
(3) No se permiten el uso de variables con contexto inexistentes (ej. x{10} en nuestro programa de ejemplo).
(4) No se permite el uso de variables inexistentes (ej. y{2} en nuestro programa de ejemplo).

Para el programa de ejemplo, el analizador semántico dará los siguientes mensajes de error por pantalla:

```
Error: contexto inexistente 10 para variable x
Error: variable inexistente y{2}
```

SE PIDE: Atribuir DE FORMA LEGIBLE el siguiente tree-parser para que genere mensajes de error.

```
programa     : #(PROGRAMA contextos defecto expresiones) ;
contextos    : #(CONTEXTOS (contexto)+) ;
contexto     : #(CONTEXTO NUMERO asignaciones) ;
asignaciones : (asignacion)+ ;
asignacion   : #(ASIG VARIABLE NUMERO) ;
defecto      : #(DEFECTO NUMERO) ;
expresiones  : #(EXPRESIONES (expresion)*) ;
expresion    : #(MAS expresion expresion)
             | #(MENOS expresion expresion)
             | #(POR expresion expresion)
             | #(LLAVEABIERTA VARIABLE NUMERO)
             | VARIABLE
             | NUMERO
             ;
```

Solución:

```
header{  import java.util.*;  }

class Anasint2 extends TreeParser;

options{ importVocab = Anasint; }

{  // Tabla para almacenar la identificación de cada contexto y las variables definidas en dicho contexto.
   // Hashable<identificacion del contexto, variables del contexto>
   Hashtable<String, Set<String>> ctxts = new Hashtable<String, Set<String>>();

   // Contexto por defecto
   String contextoPorDefecto = new String();
}

programa     : #(PROGRAMA contextos defecto expresiones) ;

contextos    : #(CONTEXTOS (contexto)+) ;

contexto       {Set<String> vars;}
             : #(CONTEXTO n:NUMERO vars=asignaciones)
               {ctxts.put(#n.getText(), vars);}
             ;

asignaciones   returns [Set<String> vars = new HashSet<String>()]
             : (asignacion[vars])+
             ;

asignacion     [Set<String> vars]
             : #(ASIG v:VARIABLE n:NUMERO)
             { if (vars.contains(#v.getText()))
                 System.out.println("Error: variable redefinida");
               else
                 vars.add(v.getText());
             }
             ;

defecto      : #(DEFECTO n:NUMERO)
             { if (!ctxts.keySet().contains(#n.getText()))
                 System.out.println("Error: contexto por defecto inexistente: " + #n.getText());
               else
                 contextoPorDefecto = #n.getText();
             }
             ;

expresiones  : #(EXPRESIONES (expresion)*) ;

expresion    : #(MAS   expresion expresion)
             | #(MENOS expresion expresion)
             | #(POR   expresion expresion)
             | #(LLAVEABIERTA v:VARIABLE n:NUMERO)
             { if (!ctxts.containsKey(#n.getText()))
                  System.out.println( "Error: contexto inexistente " + n.getText()
                                    + " para variable " + v.getText());
               else
                  if (!(ctxts.get(#n.getText())).contains(#v.getText()))
                     System.out.println("Error: variable inexistente " + v.getText() + "{" +#n.getText()+ "}");
             }
             | w:VARIABLE
             { if (!ctxts.containsKey(contextoPorDefecto))
                  System.out.println( "Error: contexto por defecto inexistente " + n.getText()
                                    + " para variable" + w.getText());
               else
                  if (!(ctxts.get(contextoPorDefecto)).contains(w.getText()))
                     System.out.println("Error: variable inexistente " + w.getText());
             }
             | NUMERO
             ;
```

6.5 Exámen 2013-2014 Tema 4.

EL LENGUAJE DE LAS EXPRESIONES CONTEXTUALIZADAS (LEC).

INTERPRETACIÓN.

El siguiente ejemplo muestra un programa en LEC:

```
CONTEXTO 1
    x = 5;
    y = 7;
    z = 1;

CONTEXTO 2
    x = 9;
    z = 3;

DEFECTO 1

EXPRESIONES
    (x{10} - y{2}) * z;
    (x{1} + y{1}) * z{2};
    (x - y) * z;
```

- Un programa LEC es una secuencia de expresiones aritméticas contextualizadas.
- Los contextos son secciones del programa en las que se inicializan las variables con constantes numéricas (ej. x = 5).
- Un programa LEC puede tener uno o más contextos.
- Todo programa LEC tiene un contexto por defecto (ej. DEFECTO 1).
- Las expresiones LEC se construyen con operadores aritméticos, constantes numéricas y variables.
- Los operadores aritméticos son la suma (+), la resta (-) y el producto (*).
- Las constantes numéricas son enteros positivos o cero.
- La variable LEC puede referenciar a un contexto (ej. z{2} es una variable del contexto 2) o al contexto por defecto (ej. x).

SE PIDE: Atribuir DE FORMA LEGIBLE el siguiente tree-parser para que interprete programas LEC.

```
programa     : #(PROGRAMA contextos defecto expresiones) ;
contextos    : #(CONTEXTOS (contexto)+) ;
contexto     : #(CONTEXTO NUMERO asignaciones) ;
asignaciones : (asignacion)+ ;
asignacion   : #(ASIG VARIABLE NUMERO) ;
defecto      : #(DEFECTO NUMERO) ;
expresiones  : #(EXPRESIONES (expresion)*) ;
expresion    : #(MAS expresion expresion)
             | #(MENOS expresion expresion)
             | #(POR expresion expresion)
             | #(LLAVEABIERTA VARIABLE NUMERO)
             | VARIABLE
             | NUMERO
             ;
```

La interpretación del programa de ejemplo mostrará por pantalla los siguientes mensajes:

```
Resultado expresión:  -2
Resultado expresión:  36
Resultado expresión:  -2
```

Solución:

```
header{
   import java.util.*;
}

class Anasint3 extends TreeParser;

options{
   importVocab = Anasint;
}

{
      // Tabla para almacenar el valor de cada variable en cada contexto
      Hashtable<String,Hashtable<String,Integer>> ctxts = new Hashtable<String,Hashtable<String,Integer>>();

      // Contexto por defecto
      String ctxt_defecto = new String();
}


programa     : #(PROGRAMA contextos defecto expresiones)
             ;

contextos    : #(CONTEXTOS (contexto)+)
             ;

contexto       {Hashtable<String,Integer> vars;}
             : #(CONTEXTO n:NUMERO vars=asignaciones)
               {ctxts.put(#n.getText(),vars);}
             ;

asignaciones   returns [Hashtable<String,Integer> vars = new Hashtable<String,Integer>()]
             : (asignacion[vars])+
             ;

asignacion     [Hashtable<String,Integer> vars]
             : #(ASIG v:VARIABLE n:NUMERO)
               { vars.put(v.getText(), new Integer(n.getText())); }
             ;

defecto      : #(DEFECTO n:NUMERO)
               { ctxt_defecto = #n.getText(); }
             ;

expresiones    {int r;}
             : #(EXPRESIONES (r=expresión {System.out.println("Resultado expresión: " + r);} )*  )
             ;

expresion      returns [int r = 0]
               {int a, b;}
             : #(MAS   a=expresion b=expresion)       { r = a + b; }
             | #(MENOS a=expresion b=expresion)       { r = a - b; }
             | #(POR   a=expresion b=expresion)       { r = a * b; }
             | #(LLAVEABIERTA v:VARIABLE n:NUMERO)    { r = ctxts.get(n.getText()).get(v.getText());  }
             | w:VARIABLE                            { r = ctxts.get(ctxt_defecto).get(w.getText()); }
             | m:NUMERO                              { r = new Integer(m.getText()).intValue();      }
             ;
```

6.6 Exámen 2014-2015 Turno mañana.

El lenguaje F.

F es el lenguaje formado por las siguientes fórmulas:

(1) Un proposición (ej. p) es una fórmula F
(2) Si p y q son proposiciones entonces p = q es una fórmula F
(3) La negación de una proposición (ej. -p) es una fórmula F
(4) Las constantes T y F son fórmulas F
(5) Si f y g son fórmulas F también lo son su conjunción (ej. f&g), disyunción (ej. f|g) e implicación (ej. f=>g). Se asume que & es un operador más prioritario que |y que |es más prioritario que =>
(6) Si f es una fórmula F también lo es f entre paréntesis (ej. (f)).

SE PIDE: (4 puntos)

(a) Analizador léxico ANTLR para reconocer los lexemas de una fórmula F.

(b) Analizador sintáctico ANTLR para reconocer la sintaxis de una fórmula F. Dicho analizador debe construir árboles de sintaxis abstracta (ASA) conteniendo la información relevante para evaluar una fórmula F. Dibuje el ASA construido por su analizador para la fórmula (-p => q) & r | s = t.

Solución:

Apartado (a) Analizador Léxico:

```
class Analex extends Lexer;

options{
   importVocab = Anasint;
   k = 2;
}

BLANCO           : ' '    {$setType(Token.SKIP);};
TABULADOR        : '\t'   {$setType(Token.SKIP);};
FIN_LINEA        : "\r\n" {$setType(Token.SKIP); newline();};

PROPOSICION      : 'a'..'z';
Y                : '&' ;
O                : '|' ;
IMPLICA          : "=>";
NO               : '-' ;
IGUAL            : '=' ;
PARENTESISABIERTO:'('  ;
PARENTESISCERRADO:')'  ;
CIERTO           : 'T' ;
FALSO            : 'F' ;
```

Apartado (b) Analizador Sintáctico:

```
class Anasint extends Parser;

options{
   buildAST = true;
   k = 2;
}

formula : (formula1 IMPLICA) => formula1 IMPLICA^ formula
        | formula1
        ;

formula1: (formula2 O) => formula2 O^ formula1
        | formula2
        ;

formula2: (formula3 Y) => formula3 Y^ formula2
        | formula3
        ;

formula3: PROPOSICION
        | CIERTO
        | FALSO
        | NO^ PROPOSICION
        | PROPOSICION IGUAL^ PROPOSICION
        | PARENTESISABIERTO! formula PARENTESISCERRADO!
        ;
```

El lenguaje de programación SET.

SET es un lenguaje diseñado para programar conjuntos de números naturales. Inicialmente, los conjuntos declarados están vacíos. Un conjunto puede modificarse asignándole una expresión. Esta expresión puede ser:

(a) Conjunto de elementos por extensión (ej. {9, 3, 5}).
(b) Conjunto unión (ej. union(s, t)).
(c) Conjunto intersección (ej. interseccion(s, t))
(d) Conjunto diferencia (ej. diferencia(r, s)).

SET dispone de operaciones para añadir/eliminar un elemento a/de un conjunto (ej. incluir(s, 10) / eliminar(s,10)). Finalmente, SET dispone de una instrucción para mostrar los elementos de un conjunto por pantalla (ej. mostrar(s)).

Ejemplo de programa SET:

```
PROGRAMA
CONJUNTOS s,t,r;
INSTRUCCIONES
      s = {9, 3, 5};
      t = {5, 4};
      r = interseccion(union(s, t), t);
      mostrar(r);
      r =union(s, t);
      mostrar(r);
      r =diferencia(r, s);
      mostrar(r);
      incluir(s, 10);
      mostrar(s);
      eliminar(s, 10);
      mostrar(s);
```

Mensajes mostrados por pantalla al interpretar el programa anterior:

```
r=[4, 5]
r=[3, 4, 5, 9]
r=[4]
s=[3, 5, 9, 10]
s=[3, 5, 9]
```

SE PIDE: (6 puntos)

Intérprete de programas SET. Construya el intérprete con un tree-parser ANTLR. Dicho tree-parser debe procesar los árboles de sintaxis abstracta generados por el siguiente parser. Justifique los atributos propuestos y explique su implementación en Java.

Solución:

```
class Anasint extends Parser;

options{
   k = 2;
   buildAST = true;
}

programa     : PROGRAMA^ conjuntos instrucciones
             ;

conjuntos    : CONJUNTOS^ lista_vars PUNTOyCOMA!
             ;

lista_vars   : (VARIABLE COMA) => VARIABLE COMA! lista_vars
             | VARIABLE
             ;

instrucciones: INSTRUCCIONES^ (instruccion PUNTOyCOMA!)*
             ;

instruccion  : asignacion | incluir | eliminar | mostrar
             ;

asignacion   : VARIABLE ASIGNACION^ expresion
             ;

incluir      : INCLUIR^ PARENTESISABIERTO! VARIABLE COMA! NUMERO PARENTESISCERRADO!
             ;

eliminar     : ELIMINAR^ PARENTESISABIERTO! VARIABLE COMA! NUMERO PARENTESISCERRADO!
             ;

mostrar      : MOSTRAR^ PARENTESISABIERTO! VARIABLE PARENTESISCERRADO!
             ;

expresion    : UNION^ PARENTESISABIERTO! expresion COMA! expresion PARENTESISCERRADO!
             | INTERSECCION^ PARENTESISABIERTO! expresion COMA! expresion PARENTESISCERRADO!
             | DIFERENCIA^ PARENTESISABIERTO! expresion COMA! expresion PARENTESISCERRADO!
             | LLAVEABIERTA^ nums LLAVECERRADA!
             | LLAVEABIERTA^ LLAVECERRADA!
             | VARIABLE
             ;

nums         : (NUMERO COMA) => NUMERO COMA! nums
             | NUMERO
             ;
```

```
header{
   import java.util.*;
}

class Anasint extends TreeParser;

options{
   importVocab = AnasintProb;
}

{
   Hashtable<String,Set<Integer>> vars = new Hashtable<String,Set<Integer>>();

   void declarar(String var){
      vars.put(var, new HashSet<Integer>());
   }

}

programa     : #(PROGRAMA conjuntos instrucciones)
             ;

conjuntos    : #(CONJUNTOS lista_vars)
             ;

lista_vars   : (a:VARIABLE {declarar(a.getText());} )+
             ;

instrucciones: #(INSTRUCCIONES (instruccion)*)
             ;

instruccion  : asignacion | incluir | eliminar | mostrar
             ;

asignacion     {Set<Integer> s;}
             : #(ASIGNACION v:VARIABLE s=expresion)
               {vars.put(v.getText(), s);}
             ;

incluir      : {Set<Integer> aux;}
               #(INCLUIR v:VARIABLE n:NUMERO)
               {aux = vars.get(v.getText());
                aux.add(new Integer(n.getText()));
                vars.put(v.getText(), aux);
               }
             ;

eliminar     : {Set<Integer> aux;}
               #(ELIMINAR v:VARIABLE n:NUMERO)
               {aux = vars.get(v.getText());
                aux.remove(new Integer(n.getText()));
                vars.put(v.getText(), aux);
               }
             ;

mostrar      : #(MOSTRAR v:VARIABLE)
               {System.out.println( v.getText() + "=" + vars.get(v.getText()).toString() );}
             ;

expresion      returns [Set<Integer> s = new HashSet<Integer>()]
               {Set<Integer> a, b;}:
               #(UNION        s=expresion b=expresion) {s.addAll(b)   ;}
             | #(INTERSECCION s=expresion b=expresion) {s.retainAll(b);}
             | #(DIFERENCIA   s=expresion b=expresion) {s.removeAll(b);}
             | #(LLAVEABIERTA a=nums {s.addAll(a);}  )
             | v:VARIABLE            {s.addAll(vars.get(v.getText())) ;}
             ;

nums           returns [Set<Integer> s = new HashSet<Integer>()]
             : (a:NUMERO {s.add(new Integer(a.getText()));} )*
             ;
```

6.7 Exámen 2014-2015 Turno tarde.

El lenguaje TREE.

TREE es el lenguaje formado por las siguientes expresiones:

(1) `tvacio` es una expresión TREE.
(2) Si `n` es un nodo y a_1, …, a_n con n > 0 son expresiones TREE entonces `enraizar(n, a1, …, an)` es una expresión TREE.
(3) Si `a` es una expresión TREE entonces `hijo(a)` es una expresión TREE.
(4) Si `a` una expresión TREE entonces `hermano(a)` es una expresión TREE.
(5) Si `a` es una expresión TREE entonces `raiz(a)` es un nodo.
(6) Si `n` es un número natural entonces `nodo(n)` es un nodo.

SE PIDE: (4 puntos)

(a) Analizador léxico ANTLR para reconocer los lexemas de una expresión TREE.

(b) Analizador sintáctico ANTLR para reconocer la sintaxis de una expresión TREE. Dicho analizador debe construir árboles de sintaxis abstracta (ASA) conteniendo la información relevante de una expresión TREE.

(c) Dibuje el ASA construido por su analizador para la expresión de ejemplo:
`enraizar(nodo(3), hijo(enraizar(nodo(10), tvacio, tvacio)), tvacio).`

Solución:

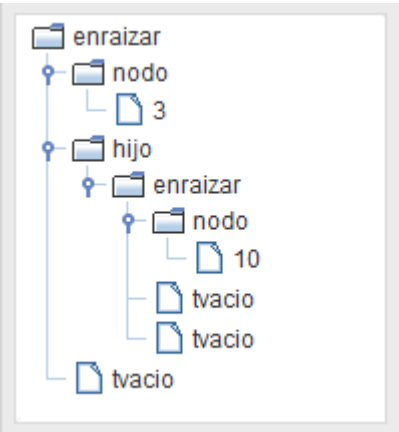
Apartado (a) Analizador Léxico.

```
class AnalexProb extends Lexer;
options{
   importVocab = AnasintProb;
   k = 2;
}
          BLANCO    : ' '    {$setType(Token.SKIP);           };
          TABULADOR : '\t'   {$setType(Token.SKIP);           };
          FIN_LINEA : "\r\n" {$setType(Token.SKIP); newline();};
protected DIGITO    : '0'..'9'  ;
NUMERO              : (DIGITO)+ ;
ENRAIZAR            : "enraizar";
HIJO                : "hijo"    ;
HERMANO             : "hermano" ;
RAIZ                : "raiz"    ;
NODO                : "nodo"    ;
COMA                : ','       ;
TVACIO              : "tvacio"  ;
PARENTESISABIERTO   : '('       ;
PARENTESISCERRADO   : ')'       ;
```

Apartado (b) Analizadir Sintáctico.

```
class AnasintProb extends Parser;
options{
   buildAST = true;
   k = 2;
}
ltree : ENRAIZAR^ PARENTESISABIERTO! nodo COMA! ltrees PARENTESISCERRADO!
      | HIJO^     PARENTESISABIERTO! ltree PARENTESISCERRADO!
      | HERMANO^  PARENTESISABIERTO! ltree PARENTESISCERRADO!
      | TVACIO
      ;
nodo  : RAIZ^ PARENTESISABIERTO! ltree  PARENTESISCERRADO!
      | NODO^ PARENTESISABIERTO! NUMERO PARENTESISCERRADO!
      ;
ltrees: (ltree COMA)=> ltree COMA! ltrees
      | ltree
      ;
```

Apartado (c) Dibujo ASA.



El lenguaje de programación CTXT.

CTXT es un lenguaje de programación basado en el concepto de bloque. El programa CTXT está formado por un bloque conteniendo una secuencia de instrucciones. Las instrucciones puede ser de tres tipos:

(a) Asignación.
(b) Bloque
(c) Mostrar por pantalla.

Ejemplo de programa CTXT:

```
{
   x = 2;
   y = 7;
   z = 0;
   mostrar(x);
   {
      x = 9;
      mostrar(x);
      y = 9;
      mostrar(y);
   }
   mostrar(y);
   x = x + 1;
   mostrar(x);
}
```

Mensajes mostrados por pantalla al interpretar el programa anterior:

```
x = 2
x = 9
y = 9
y = 7
x = 3
```

SE PIDE: (6 puntos)

Intérprete de programas CTXT. Construya el intérprete con un tree-parser ANTLR que procese los árboles de sintaxis abstracta generados por el siguiente parser. Justifique los atributos propuestos y explique su implementación en Java.

Solución:

```
class Anasint extends Parser;

options{
   buildAST = true;
   k = 2;
}

tokens{
   PROGRAMA;
}

programa   : b:bloque {#programa = #(#[PROGRAMA,"PROGRAMA"], b);}
           ;

bloque     : LLAVEABIERTA^ (instruccion)* LLAVECERRADA!
           ;

instruccion: asignacion | bloque | mostrar
           ;

asignacion : IDENT ASIGNACION^ expresion PUNTOYCOMA!
           ;

mostrar    : MOSTRAR^ PARENTESISABIERTO! IDENT PARENTESISCERRADO! PUNTOYCOMA!
           ;

expresion  : (expresion1 (MAS|MENOS)) => expresion1 (MAS^|MENOS^) expresion
           | expresion1
           ;

expresion1 : (expresion2 (POR|DIV))   => expresion2 (POR^|DIV^) expresion1
           | expresion2
           ;

expresion2 : NUMERO
           | IDENT
           | PARENTESISABIERTO! expresion PARENTESISCERRADO!
           ;
```

```
header{
   import java.util.*;
}

class Anasint extends TreeParser;

options{
   importVocab = Anasint;
}

{
   Stack<Hashtable<String,Integer>> datos = new Stack<Hashtable<String,Integer>>();

   void crear_ctxt_vars(){
      Hashtable<String, Integer> vars = new Hashtable<String, Integer>();
      if (!datos.isEmpty())
         vars.putAll(datos.peek());
      datos.push(vars);
   }

   void asignar(String var, Integer valor){
      Hashtable<String,Integer> vars = datos.pop();
      vars.put(var, valor);
      datos.push(vars);
   }
}



programa   : #(PROGRAMA bloque)
           ;

bloque     : #(LLAVEABIERTA {crear_ctxt_vars();} (instruccion)*)
             { datos.pop(); }
           ;

instruccion: asignacion | bloque | mostrar
           ;

asignacion   { Integer r; }
           : #(ASIGNACION a:IDENT r=expresion)
             { asignar(a.getText(),r); }
           ;

mostrar    : #(MOSTRAR b:IDENT)
             { System.out.println(b.getText() + " = " + datos.peek().get(b.getText())); }
           ;

expresion    returns [Integer r=0]
             { Integer r1, r2; }
           : #(MAS   r1=expresion r2=expresion)   {r = r1 + r2;}
           | #(MENOS r1=expresion r2=expresion)   {r = r1 - r2;}
           | #(POR   r1=expresion r2=expresion)   {r = r1 * r2;}
           | #(DIV   r1=expresion r2=expresion)   {r = r1 / r2;}
           | a:NUMERO {r = new Integer(a.getText());}
           | b:IDENT  {r = datos.peek().get(b.getText());}
           ;
```

6.8 Exámen 2014-2015 Convocatoria final Enero.

El lenguaje ASSERTION

ASSERTION es un lenguaje de programación. Sus programas pueden incluir asertos. Un aserto es una condición lógica construida con las siguientes reglas:

(1) Si f y g son expresiones enteras entonces f == g, f != g, f > g y f < g son asertos.
(2) Las constantes T y F son asertos.
(3) Si f y g son asertos también lo son su conjunción (ej. f&g), disyunción (ej. f|g) y negación (ej. –f, -g).
(4) Si f es un aserto también lo es f entre paréntesis (ej. (f)).

Sintácticamente, un programa ASSERTION está compuesto por una declaración de variables y una secuencia de instrucciones. Las instrucciones pueden ser:

(a) Asignaciones.
(b) Instrucciones condicionales.
(c) Iteraciones.

Un programa ASSERTION puede incluir un aserto entre llaves en cualquier punto de su secuencia de instrucciones.

Ejemplo de programa ASSERTION:

```
PROGRAMA
   VARIABLES x, y, z;
   LEER(x);
   { x > 0 }
   z = 1;
   { z < x/2 | z == x/2 }
   y = 1;
   MIENTRAS (y < x/2 | y == x/2) HACER
      SI ((x/y)*y == x) ENTONCES
         z = y;
      FINSI
      y = y + 1;
      { (x/z) * z == x & (z == 1 | z > 1) & (z < x/2 | z == x/2) }
   FINMIENTRAS
   { (x/z) * z == x & (z == 1 | z > 1) & (z < x/2 | z == x/2) }
```

SE PIDE: (3 puntos)

(a) Analizador léxico ANTLR del lenguaje ASSERTION

(b) Analizador sintáctico ANTLR del lenguaje ASSERTION

(c) Dibujo ASA

Solución:

Apartado (a) Analizador Léxico.

```
class AnalexAssertion extends Lexer;
options{
    importVocab = AnasintAssertion;
    k = 2;
}

tokens{
    PROGRAMA    = "PROGRAMA"   ;
    VARIABLES   = "VARIABLES"  ;
    SI          = "SI"         ;
    ENTONCES    = "ENTONCES"   ;
    FINSI       = "FINSI"      ;
    MIENTRAS    = "MIENTRAS"   ;
    HACER       = "HACER"      ;
    FINMIENTRAS = "FINMIENTRAS";
    LEER        = "LEER"       ;
}
protected LETRA :'a'..'z'|'A'..'Z';
protected DIGITO:'0'..'9';
BLANCO  : ' '    {$setType(Token.SKIP);}            ;
TAB     : '\t'   {$setType(Token.SKIP);}            ;
NLINEA  : "\r\n" {$setType(Token.SKIP); newline();} ;
VAR     : LETRA(LETRA|NUMERO)*;
NUMERO  : (DIGITO)+;
ASIG    : '=';
MAS     : '+';
MENOS   : '-';
POR     : '*';
DIV     : '/';
Y       : '&';
O       : '|';
PA      : '(';
PC      : ')';
LLA     : '{';
LLC     : '}';
IGUAL   : "==";
DISTINTO: "!=";
MAYOR   : '>';
MENOR   : '<';
PyC     : ';';
COMA    : ',';
```

Apartado (b) Analizador Sintáctico.

```
class AnasintAssertion extends Parser;

    programa      : PROGRAMA variables instrucciones EOF
                  ;

    variables     : VARIABLES vars PyC
                  ;

    vars          : (VAR COMA)=> VAR COMA vars
                  | VAR
                  ;

    instrucciones : (instruccion | aserto)*
                  ;

    instruccion   : lectura
                  | asignación
                  | condicional
                  | iteracion
                  ;

    lectura       : LEER PA VAR PC PyC
                  ;

    asignacion    : VAR ASIG expr PyC
                  ;

    condicional   : SI PA cond PC ENTONCES instrucciones FINSI
                  ;

    iteracion     : MIENTRAS PA cond PC HACER instrucciones FINMIENTRAS
                  ;

    aserto        : LLA cond LLC
                  ;

    expr          : (expr1 (MAS|MENOS))=> expr1 (MAS|MENOS) expr
                  | expr1
                  ;

    expr1         : (expr2 (POR|DIV))=> expr2 (POR|DIV) expr1
                  | expr2
                  ;

    expr2         : NUMERO
                  | VAR
                  | PA expr PC
                  ;

    cond          : (cond1 O)=> cond1 O cond
                  | cond1
                  ;

    cond1         : (cond2 Y)=> cond2 Y cond1
                  | cond2
                  ;

    cond2         : (expr IGUAL)   => expr IGUAL expr
                  | (expr DISTINTO)=> expr DISTINTO expr
                  | (expr MAYOR)   => expr MAYOR expr
                  | (expr MENOR)   => expr MENOR expr
                  | CIERTO
                  | FALSO
                  | MENOS cond
                  | PA cond PC
                  ;
```

El lenguaje BLQ

BLQ es un lenguaje de programación basado en el concepto de bloque. Un bloque BLQ es una construcción formada por una declaración de variables y una secuencia de instrucciones. Las instrucciones en BLQ son de dos tipos: (a) Asignaciones y (b) Bloques. Las declaraciones de variables en un bloque se propagan a los bloques internos a éste pero no al revés.

Ejemplo de programa BLQ:

```
BLOQUE x, z;
   x = 1;
   m = 2 * (x + 7);
   BLOQUE x, m;
      x = m + z;
      n = 3;
   FBLOQUE
   z = m + 1;
FBLOQUE
```

SE PIDE: (3 puntos)

Analizador semántico ANTLR que detecte el uso de variables no declaradas. Una variable se dice no declarada si no tiene declaración en el punto en el que ésta ocurre. Por ejemplo, la ejecución del analizador semántico sobre el programa anterior emitirá los siguientes mensajes por pantalla:

```
Variable m no declarada: instrucción m = 2 * (x + 7);
Variable n no declarada: instrucción n = 3;
Variable m no declarada: instrucción z = m + 1;
```

La implementación del analizador semántico se realizará con un treeparser. Dicho analizador debe procesar los árboles de sintaxis abstracta generados por el siguiente parser ANTLR. Justifique los atributos propuestos y explique su implementación en Java.

Solución:

```
class AnasintBlq extends Parser;
   options{
      buildAST = true;
   }

   programa     : bloque EOF!
                ;

   bloque       : BLOQUE^ variables instrucciones FBLOQUE!
                ;

   variables    : vars PyC!
                ;

   vars         : (VAR COMA) => VAR COMA! vars
                | VAR
                ;

   instrucciones: (asignacion | bloque)*
                ;

   asignacion   : VAR ASIG^ expr PyC!
                ;

   expr         : (expr1 (MAS|MENOS)) => expr1 (MAS^|MENOS^) expr
                | expr1
                ;

   expr1        : (expr2 (POR|DIV)) => expr2 (POR^|DIV^) expr1
                | expr2
                ;

   expr2        : NUMERO
                | VAR
                | PA! expr PC!
                ;



header{
   import java.util.*;
}
class Anasint2Blq extends TreeParser;
   options{
      importVocab = AnasintBlq;
   }
{

Stack<Set<String>> memoria = new Stack<Set<String>>();

void inicializar_memoria_bloque(){
   Set<String> m = new HashSet<String>();
   if (!memoria.isEmpty())
   m.addAll(memoria.peek());
   memoria.push(m);
}

void eliminar_memoria_bloque(){
   memoria.pop();
}

void declarar_variable(String var){
   Set<String> m = memoria.pop();
   m.add(var);
   memoria.push(m);
}

Boolean comprobar_declaracion_variable(String var){
   Boolean resultado = true;
   if (!memoria.peek().contains(var))
   resultado = false;
   return resultado;
}
```

```
String expresionASTtoString(AST e, int op_padre){
   String r = new String();
   String s;
   switch(e.getType())
      {
      case MAS:    r = expresionASTtoString(e.getFirstChild(), MAS);
                   s = expresionASTtoString(e.getFirstChild().getNextSibling(), MAS);
                   if (op_padre==POR | op_padre==DIV)
                      r = '(' + r + e.getText() + s +')';
                   else
                      r = r + e.getText() + s;
                   break;
      case MENOS:  r = expresionASTtoString(e.getFirstChild(), MENOS);
                   s = expresionASTtoString(e.getFirstChild().getNextSibling(), MENOS);
                   if (op_padre==POR | op_padre==DIV)
                      r = '('+ r + e.getText() + s + ')';
                   else
                      r = r + e.getText() + s;
                   break;
      case POR:    r = expresionASTtoString(e.getFirstChild(), POR);
                   s = expresionASTtoString(e.getFirstChild().getNextSibling(), POR);
                   r = r + e.getText() + s;
                   break;
      case DIV:    r = expresionASTtoString(e.getFirstChild(), DIV);
                   s = expresionASTtoString(e.getFirstChild().getNextSibling(), DIV);
                   r = r + e.getText() + s;
                   break;
      case NUMERO: r = e.getText();
                   break;
      case VAR:    r = e.getText();
                   break;
      default:
      }
   return r;
}

void comprobar(String var, AST exp, Set<String> vars){
   if (!comprobar_declaracion_variable(var))
      System.out.println("Variable " + var + " no declarada: instrucción " + var + " = " +
      expresionASTtoString(exp, -5) + ";");
   else
      for (String s:vars){
         if (!comprobar_declaracion_variable(s))
            System.out.println("Variable " + s + " no declarada: instrucción " + var + " = " +
            expresionASTtoString(exp, -5) + ";");
      }
   }

}



bloque       : {inicializar_memoria_bloque();}
               #(BLOQUE variables instrucciones)
               {eliminar_memoria_bloque();}
             ;

variables    : (v:VAR {declarar_variable(v.getText());})+
             ;

instrucciones: (asignacion | bloque)*
             ;

asignacion     {Set<String> l;}
             : #(ASIG v:VAR l=e:expr)
               {comprobar(v.getText(), e, l);}
             ;

expr          returns [Set<String> r = new HashSet<String>();]
              {Set<String> s;}
             : #(MAS   r=expr s=expr) {r.addAll(s);}
             | #(MENOS r=expr s=expr) {r.addAll(s);}
             | #(POR   r=expr s=expr) {r.addAll(s);}
             | #(DIV   r=expr s=expr) {r.addAll(s);}
             | n:NUMERO
             | v:VAR {if (!comprobar_declaracion_variable(v.getText()))
                          r.add(v.getText());
                      }
             ;
```

El lenguaje SEQ

Un programa SEQ está formado por una secuencia de instrucciones. Estas instrucciones pueden ser de 3 tipos:

(1) Instrucción para mostrar por pantalla el valor de una variable.
(2) Instrucción para asignar una expresión entera a una variable.
(3) Instrucción condicional.

Ejemplo de programa SEQ:

```
x = 5 + 7;
mostrar(x);
si (x >= 0 & x <= 10)
   y = 5;
   mostrar(y);
   si (x < 5)
      y = 0;
      mostrar(y);
   sino
      y = 10;
      mostrar(y);
   finsi
sino
   si (x < 0)
      y = 1;
      mostrar(y);
   sino
      y = 11;
      mostrar(y);
   finsi
finsi
x = x + 1;
mostrar(x);
```

La interpretación del anterior programa muestra por pantalla los siguientes mensajes:

```
x = 12
y = 11
x = 13
```

SE PIDE: (4 puntos)

Interprete de programas SEQ. Explique las principales decisiones de diseño y aclare la forma de implementarlas. Se supone que los programas SEQ no contienen errores que impidan su interpretación.

La implementación del intérprete se hará atribuyendo el siguiente parser ANTLR:

```
class Anasint extends Parser;

programa    : bloque EOF
            ;

bloque      : (instruccion)*
            ;

instruccion : asignacion
            | condicional
            | mostrar
            ;

asignacion  : IDENT ASIGNACION termino PUNTOYCOMA
            ;

condicional : SI PARENTESISABIERTO condicion PARENTESISCERRADO
                bloque
              SINO
                bloque
              FINSI
            ;

mostrar     : MOSTRAR PARENTESISABIERTO IDENT PARENTESISCERRADO PUNTOYCOMA
            ;

condicion   : (condicion1 O) => condicion1 O condicion
            | condicion1
            ;

condicion1  : (condicion2 Y) => condicion2 Y condicion1
            | condicion2
            ;

condicion2  : (termino MAYORIGUAL) => termino MAYORIGUAL termino
            | (termino MAYOR)      => termino MAYOR termino
            | (termino MENORIGUAL) => termino MENORIGUAL termino
            | (termino MENOR)      => termino MENOR termino
            | (termino IGUAL)      => termino IGUAL termino
            | termino DISTINTO termino
            ;

termino     : (termino1 MAS)   => termino1 MAS termino
            | (termino1 MENOS) => termino1 MENOS termino
            | (termino1 POR)   => termino1 POR termino
            | (termino1 DIV)   => termino1 DIV termino
            | termino1
            ;

termino1    : IDENT
            | NUMERO
            | PARENTESISABIERTO termino PARENTESISCERRADO
            ;
```

Los lexemas usados en el anterior parser se han definido en el siguiente lexer ANTLR:

```
class Analex extends Lexer;

   options{
      importVocab = Anasint;
      k = 2;
   }

   tokens{
      MOSTRAR = "mostrar";
      SI      = "si"     ;
      SINO    = "sino"   ;
      FINSI   = "finsi"  ;
   }

BLANCO           : ' '    {$setType(Token.SKIP);}           ;
TABULADOR        : '\t'   {$setType(Token.SKIP);}           ;
FIN_LINEA        : "\r\n" {$setType(Token.SKIP); newline();};
protected DIGITO : '0'..'9';
protected LETRA  : 'a'..'z';
Y                : '&';
O                : '|';
IDENT            : LETRA(LETRA|DIGITO)*;
NUMERO           : (DIGITO)+;
ASIGNACION       : '=';
MAS              :' +';
MENOS            : '-';
POR              : '*';
DIV              : '/';
MAYOR            : '>';
MENOR            :' <';
MAYORIGUAL       : ">=";
MENORIGUAL       : "<=";
IGUAL            : "==";
DISTINTO         : "!=";
COMA             : ',';
PUNTOYCOMA       : ';';
PARENTESISABIERTO: '(';
PARENTESISCERRADO: ')';
```

```
header{   import antlr.*;
          import java.util.*;   }

class Anasint2Seq extends Parser;
   {
      ASTFactory                 factory    = new ASTFactory();
      Hashtable<String, Integer> vars       = new Hashtable<String, Integer>();
      Stack<Boolean>             centinelas = new Stack<Boolean>();
   }

programa    : {centinelas.push(true);} bloque {centinelas.pop();} EOF
            ;
bloque      : (instruccion)*
            ;
instruccion : asignacion
            | condicional
            | mostrar
            ;
asignacion    {Integer r;}
            : a:IDENT ASIGNACION r=termino PUNTOYCOMA
              {if (centinelas.peek()) vars.put(a.getText(),r);}
            ;
condicional   {Boolean b1=centinelas.peek(); Boolean b2;}
            : SI PARENTESISABIERTO b2=condicion PARENTESISCERRADO
                  { if (b1)
                     if (b2)
                        centinelas.push(true);
                     else
                        centinelas.push(false);
                  else
                     centinelas.push(false);
                  }
                bloque
                  {centinelas.pop();}
              SINO
                  { if (b1)
                     if (!b2)
                        centinelas.push(true);
                     else
                        centinelas.push(false);
                  else
                     centinelas.push(false);
                  }
                bloque
                  {centinelas.pop();}
              FINSI
            ;
mostrar     : MOSTRAR PARENTESISABIERTO a:IDENT PARENTESISCERRADO PUNTOYCOMA
              {if (centinelas.peek())
                           System.out.println(a.getText() + " = " + vars.get(a.getText()));}
            ;
condicion     returns [Boolean r = true]   {Boolean r1, r2;}
            : (condicion1 O)=> r1=condicion1 O r2=condicion {r=(r1|r2);}
            | r=condicion1
            ;
condicion1    returns [Boolean r = true]   {Boolean r1, r2;}
            : (condicion2 Y)=> r1=condicion2 Y r2=condicion1 {r=(r1&r2);}
            | r=condicion2
            ;
condicion2    returns [Boolean r = true]   {Integer r1, r2;}
            : (termino MAYORIGUAL) => r1=termino MAYORIGUAL r2=termino {r = (r1 >= r2);}
            | (termino MAYOR)      => r1=termino MAYOR      r2=termino {r = (r1 >  r2);}
            | (termino MENORIGUAL) => r1=termino MENORIGUAL r2=termino {r = (r1 <= r2);}
            | (termino MENOR)      => r1=termino MENOR      r2=termino {r = (r1 <  r2);}
            | (termino IGUAL)      => r1=termino IGUAL      r2=termino {r = (r1 == r2);}
            | r1=termino DISTINTO r2=termino                           {r = (r1 != r2);}
            ;
termino       returns [Integer r = 0]      {Integer r1, r2;}
            : (termino1 MAS)   => r1=termino1 MAS   r2=termino {r = r1 + r2;}
            | (termino1 MENOS) => r1=termino1 MENOS r2=termino {r = r1 - r2;}
            | (termino1 POR)   => r1=termino1 POR   r2=termino {r = r1 * r2;}
            | (termino1 DIV)   => r1=termino1 DIV   r2=termino {r = r1 / r2;}
            | r=termino1
            ;
termino1      returns [Integer r = 0]
            : a:IDENT  {r = vars.get(a.getText());}
            | b:NUMERO {r = new Integer(b.getText());}
            | PARENTESISABIERTO r=termino PARENTESISCERRADO
            ;
```