

COMPILADORES PL2

Grupo 23

David Bachiller Vela

Víctor Sanavia Valdeolivas

Martín Mora

Contenido

INTRODUCCIÓN	3
1. Primera parte: generación de árboles sintácticos para lenguajes específicos	4
1.1 SQLMini	4
1.1.2 Programa SQL	15
1.2 Linguine	21
1.2.2 Programa Linguine	35
2. Segunda parte: iniciación a la tabla de símbolos	43

INTRODUCCIÓN

En el ámbito de la informática, la capacidad de construir analizadores léxicos y sintácticos para lenguajes de programación es fundamental. En este proyecto, nos adentramos en el mundo de la generación de árboles sintácticos utilizando la herramienta ANTLR. A través de la definición de gramáticas, el desarrollo de lexers y parsers, y la construcción de Árboles de Sintaxis Abstracta (AST), exploramos la esencia de la interpretación de lenguajes específicos.

Esta práctica se divide en dos partes: en la primera, nos enfocamos en la creación de un analizador capaz de identificar automáticamente los elementos de un lenguaje dado, mientras que, en la segunda, nos sumergimos en la visualización y manipulación de los ASTs resultantes. Además, siempre se podrán proponer mejoras que enriquezcan la gramática y la funcionalidad del analizador.

Para entender todo este proceso ahí que entender que lo que hace el analizador léxico es analizar una cadena de entrada, a partir de una Expresión Regular o definiciones regulares. Generando un token con un cierto valor, mientras que el analizador sintáctico le llegan todos estos tokens en un determinado orden y su función es verificar si el orden en el que llegan los tokens es válido, con sus determinadas reglas gramaticales, con lo que acabamos generando un árbol en el que se puede ver toda esta secuencia.

1. Primera parte: generación de árboles sintácticos para lenguajes específicos

1.1 SQLMini

En esta parte de la práctica se pedía construir una gramática capaz de detectar distintas consultas SQL, las cuales se encontraban en el enunciado.

Primero que todo, debemos entender cómo funciona una gramática, la cual consta de dos partes:

- **Lexer** → Es el encargado de almacenar los símbolos terminales y de pasar los tokens a la siguiente parte
- **Parser** → Se encarga de leer la entrada y asignarlos usando los tokens que le llegan del *lexer*.

Nosotros hemos decidido dividir el *lexer* y el *parser* en dos archivos distintos.

En cuanto al *lexer* tenemos el siguiente código fuente:

```
lexer grammar gSqlMiniLexer;

//Aquí pondremos los simbolos terminales que tiene nuestro SqlMini

SELECT: 'SELECT';
FROM: 'FROM';
WHERE: 'WHERE';
ORDER: 'ORDER';
BY: 'BY';
ASC: 'ASC';
DESC: 'DESC';
MAYORQUE: '>';
MAYORIGUALQUE: '>=';
IGUALQUE: '=';
ABREPARENTESIS: '(';
CIERRAPARENTESIS: ')';
NUMERO: [0-9]+ ('.' [0-9]+)?; //Ahora tambien evaluamos los numeros
decimales
STRING: '\'' ~'\''* '\'';
COMA: ',';
AND: 'AND';
OR: 'OR';
ID: ([a-zA-Z]+) | '*'; // Identificadores (en este caso, nombres de
columnas o tablas)
WS: [ \t\r\n]+ -> skip; // Ignorar espacios en blanco
```

Palabras Clave y Operadores:

Las palabras clave como **SELECT**, **FROM**, **WHERE**, **ORDER**, **BY**, **ASC**, **DESC**, **AND**, y **OR**, así como los operadores de comparación como **MAYORQUE**, **MAYORIGUALQUE**, y **IGUALQUE**, son elementos fundamentales en cualquier consulta SQL.

Símbolos de Puntuación:

Los símbolos como, (**coma**), ((**paréntesis de apertura**) y) (**paréntesis de cierre**) son parte esencial de la sintaxis de SQLMini. Estos símbolos se utilizan para separar elementos en la consulta y para delimitar expresiones.

Números:

La regla NUMERO está diseñada para reconocer tanto números enteros como números decimales. Por ejemplo, esta regla puede identificar tanto 123 como 12.34.

Cadenas de Texto:

La regla STRING permite reconocer cadenas de texto encerradas entre comillas simples ('). Esto es crucial para manipular texto en las consultas SQL.

Identificadores:

La regla ID se utiliza para reconocer identificadores, que en SQLMini suelen ser nombres de columnas o tablas. Por ejemplo, si tenemos una columna llamada nombre, en la base de datos, esta regla permitirá identificarla.

Espacios en Blanco:

La regla WS se encarga de ignorar los espacios en blanco, tabulaciones, retornos de carro y saltos de línea. Esto es importante para que el analizador léxico pueda omitir caracteres que no afectan la estructura de la consulta.

Al definir estas reglas léxicas, establecemos la base para que el analizador léxico pueda reconocer los elementos individuales del código fuente SQLMini. Estos tokens serán utilizados posteriormente por el analizador sintáctico para construir la estructura de la consulta en forma de árbol sintáctico abstracto (AST).

Una vez introducido el *lexer* nos vamos al *parser*, el cual tiene el siguiente código fuente:

Vamos a utilizar los tokens que hemos definido en el *lexer* para poder declarar el *parser* con la sintaxis que se nos ha indicado que se define en programa.

```
parser grammar gSqlMiniParser;

options{
    tokenVocab = gSqlMiniLexer;
    language = Java;
}

programa: (consultaSelect)*;

consultaSelect : SELECT columnas FROM nombreTabla sentenciaWhere?
sentenciaOrderBy?;

columnas: nombreColumna (COMA nombreColumna)*;

nombreColumna: ID;

nombreTabla: ID;

sentenciaWhere: WHERE expresion;

sentenciaOrderBy: ORDER BY nombreColumna metodoOrdenacion;

metodoOrdenacion: ASC | DESC;

expresion: ABREPARENTESIS expresion CIERRAPARENTESIS
        | expresion AND expresion
        | expresion OR expresion
        | nombreColumna IGUALQUE (NUMERO | STRING)
        | nombreColumna (MAYORQUE | MAYORIGUALQUE) NUMERO;
```

consultaSelect: Esta es la regla principal que representa una consulta SELECT en SQLMini. Incluye todas las posibles partes de una consulta: la **selección de columnas**, la **tabla de donde se seleccionan los datos**, la **cláusula WHERE (opcional)** y la **cláusula ORDER BY (opcional)**.

columnas: Define la lista de columnas que se seleccionarán en la consulta. Puede haber una o más columnas separadas por comas.

nombreColumna: Representa el nombre de una columna. Esto es esencial para especificar qué datos se deben recuperar de la tabla.

nombreTabla: Define el nombre de la tabla desde donde se seleccionarán los datos.

sentenciaWhere: Representa la cláusula WHERE que filtra los resultados de la consulta según una condición dada.

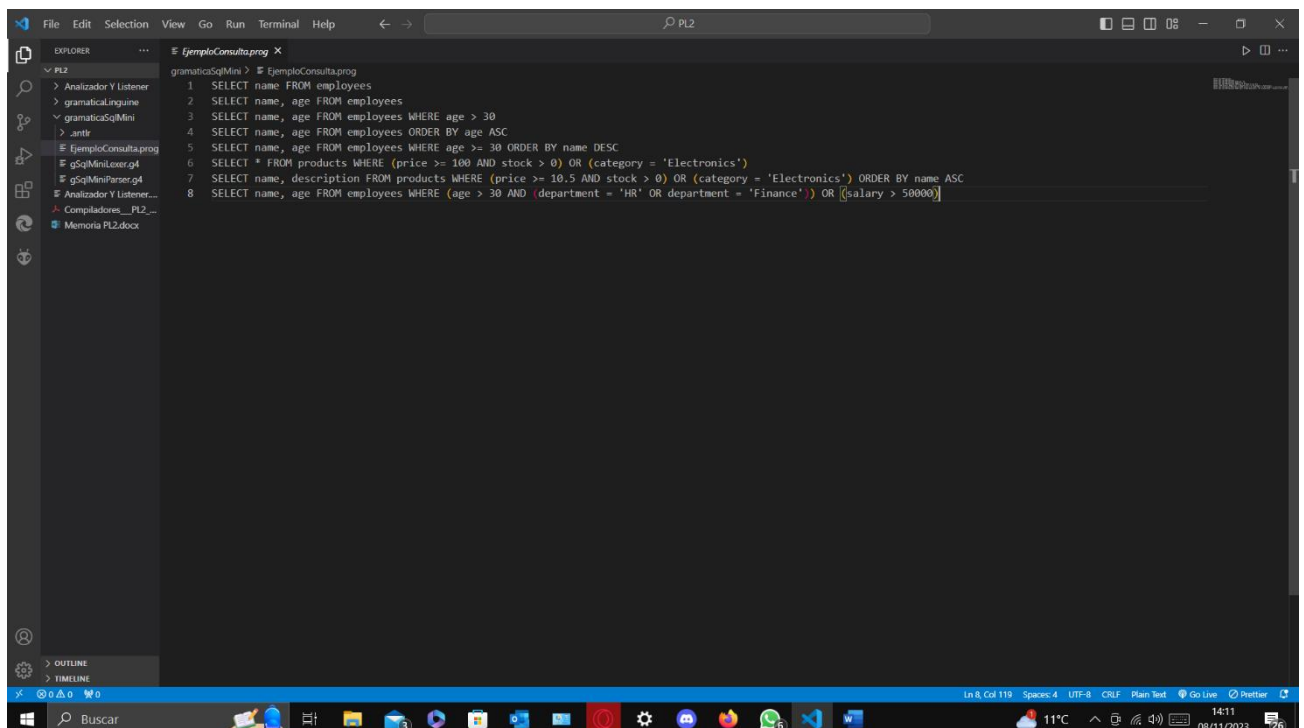
sentenciaOrderBy: Representa la cláusula ORDER BY que permite ordenar los resultados de la consulta.

metodoOrdenacion: Define si se ordena de forma ascendente (ASC) o descendente (DESC) según una columna específica.

expresion: Esta regla define las diferentes expresiones que pueden aparecer en la cláusula WHERE. Pueden ser comparaciones, operaciones lógicas (AND y OR) o expresiones entre paréntesis.

Reglas alternativas en expresion: Se incluyen diferentes formas de expresiones, como comparaciones entre columnas y valores, así como operaciones lógicas.

Una vez definido el *lexer* y *parser* vamos a declarar todas las consultas de las que se va a generar el árbol en un programa el cual hemos llamado **EjemploConsulta.prog**, donde pondremos las consultas SQL de ejemplo:



```
1 SELECT name FROM employees
2 SELECT name, age FROM employees
3 SELECT name, age FROM employees WHERE age > 30
4 SELECT name, age FROM employees ORDER BY age ASC
5 SELECT name, age FROM employees WHERE age >= 30 ORDER BY name DESC
6 SELECT * FROM products WHERE (price >= 100 AND stock > 0) OR (category = 'Electronics')
7 SELECT name, description FROM products WHERE (price >= 10.5 AND stock > 0) OR (category = 'Electronics') ORDER BY name ASC
8 SELECT name, age FROM employees WHERE (age > 30 AND (department = 'HR' OR department = 'Finance')) OR (salary > 50000)
```

Una vez creados estos tres elementos, pasamos a la compilación para poder mostrar el resultado del árbol, los pasos para esto son:

antlr gSqlMiniLexer.g4

antlr gSqlMiniParser.g4

javac *.java

grun gSqlMini programa -tokens -gui [EjemploConsulta.prog]

grun gSqlMini consultaSelect: Estás ejecutando la regla consultaSelect definida en tu parser.

-tokens: Muestra los tokens reconocidos por el lexer.

-gui: Muestra una interfaz gráfica para visualizar el árbol de análisis.

EjemploConsulta.prog: archivo de entrada que contiene una consulta SQLMini.

```
cs: Administrador: Símbolo del sistema - grun gSqlMini programa -tokens -gui EjemploConsulta.prog
C:\antlr\gramaticas\gramaticaSqlMini>antlr gSqlMiniLexer.g4

C:\antlr\gramaticas\gramaticaSqlMini>java org.antlr.v4.Tool gSqlMiniLexer.g4
C:\antlr\gramaticas\gramaticaSqlMini>antlr gSqlMiniParser.g4

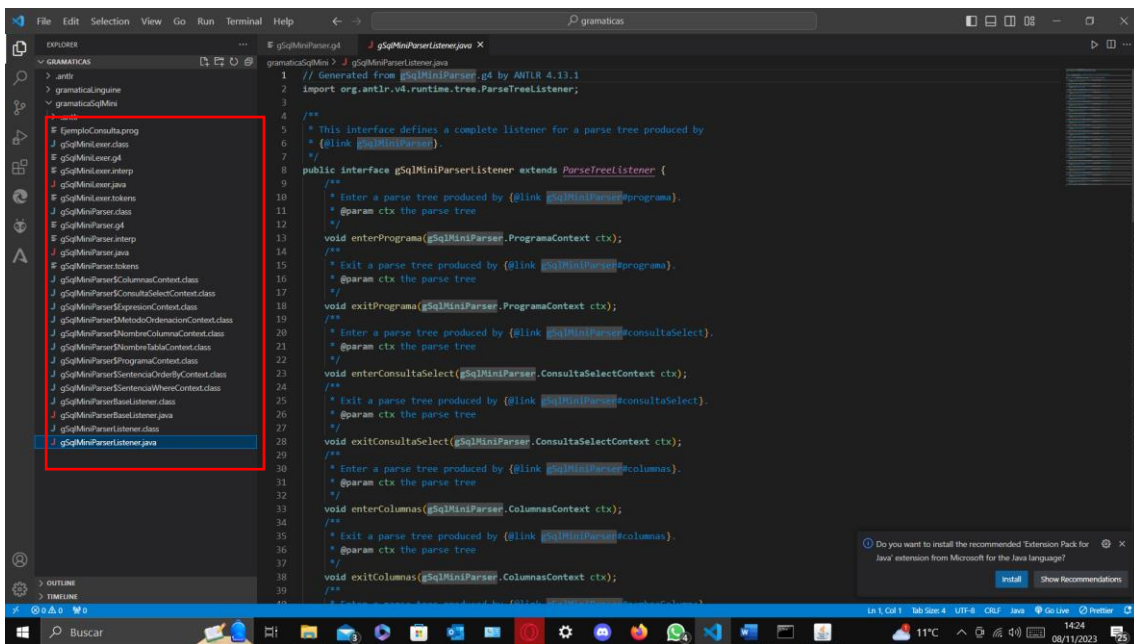
C:\antlr\gramaticas\gramaticaSqlMini>java org.antlr.v4.Tool gSqlMiniParser.g4
C:\antlr\gramaticas\gramaticaSqlMini>javac *.java

C:\antlr\gramaticas\gramaticaSqlMini>grun gSqlMini programa -tokens -gui EjemploConsulta.prog

C:\antlr\gramaticas\gramaticaSqlMini>java org.antlr.v4.gui.TestRig gSqlMini programa -tokens -gui EjemploConsulta.prog
[@0,0:5='SELECT',<'SELECT'>,1:0]
[@1,7:10='name',<ID>,1:7]
[@2,12:15='FROM',<'FROM'>,1:12]
[@3,17:25='employees',<ID>,1:17]
[@4,28:33='SELECT',<'SELECT'>,2:0]
[@5,35:38='name',<ID>,2:7]
[@6,39:39='',<'>',2:11]
[@7,41:43='age',<ID>,2:13]
[@8,45:48='FROM',<'FROM'>,2:17]
[@9,50:58='employees',<ID>,2:22]
[@10,61:66='SELECT',<'SELECT'>,3:0]
[@11,68:71='name',<ID>,3:7]
[@12,72:72='',<'>',3:11]
[@13,74:76='age',<ID>,3:13]
[@14,78:81='FROM',<'FROM'>,3:17]
[@15,83:91='employees',<ID>,3:22]
[@16,93:97='WHERE',<'WHERE'>,3:32]
[@17,99:101='age',<ID>,3:38]
[@18,103:103='>',<'>',3:42]
[@19,105:106='30',<NUMERO>,3:44]
[@20,109:114='SELECT',<'SELECT'>,4:0]
[@21,116:119='name',<ID>,4:7]
[@22,120:120='',<'>',4:11]
[@23,122:124='age',<ID>,4:13]
[@24,126:129='FROM',<'FROM'>,4:17]
[@25,131:139='employees',<ID>,4:22]
[@26,141:145='ORDER',<'ORDER'>,4:32]
[@27,147:148='BY',<'BY'>,4:38]
[@28,150:152='age',<ID>,4:41]
[@29,154:156='ASC',<'ASC'>,4:45]
[@30,159:164='SELECT',<'SELECT'>,5:0]
[@31,166:169='name',<ID>,5:7]
[@32,170:170='',<'>',5:11]
[@33,172:174='age',<ID>,5:13]
[@34,176:179='FROM',<'FROM'>,5:17]
[@35,181:189='employees',<ID>,5:22]
[@36,191:195='WHERE',<'WHERE'>,5:32]
[@37,197:199='age',<ID>,5:38]
[@38,201:202='>=>',<'>=>',5:42]
```

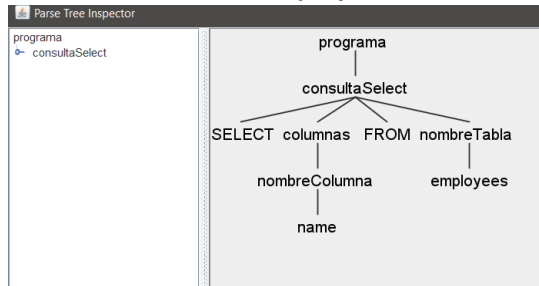

Administrador: Símbolo del sistema - grun gSqlMini program

Una vez que se despliegan todos los tokens y se compila todo vemos como se nos generan los archivos .class, .java,

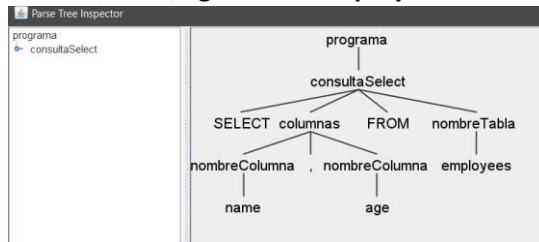


Ahora vamos a mostrar el árbol consulta por consulta:

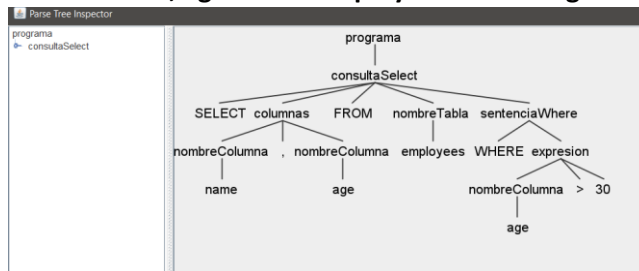
1) **SELECT name FROM employees**



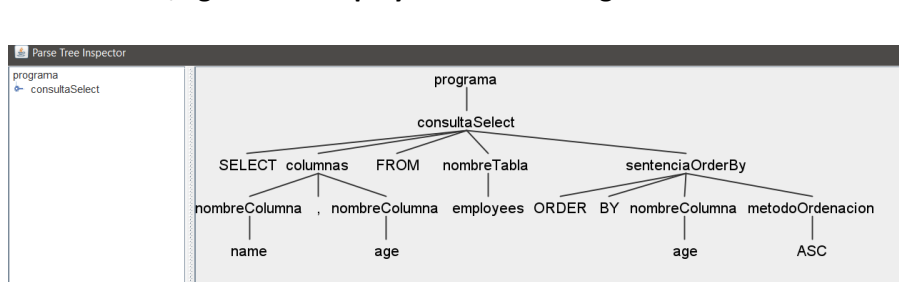
2) **SELECT name, age FROM employees**



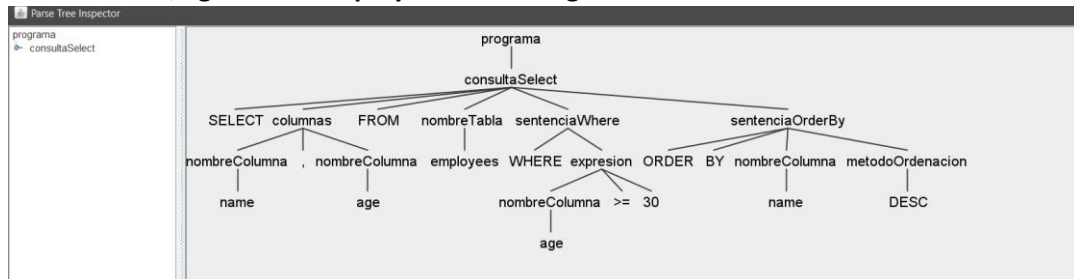
3) **SELECT name, age FROM employees WHERE age > 30**



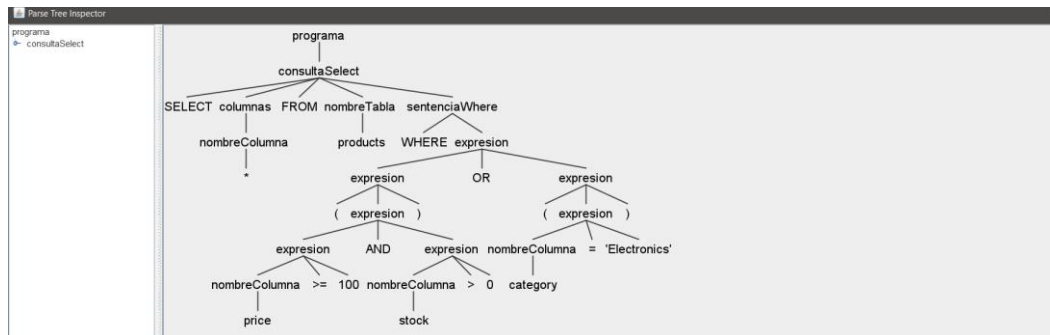
4) **SELECT name, age FROM employees ORDER BY age ASC**



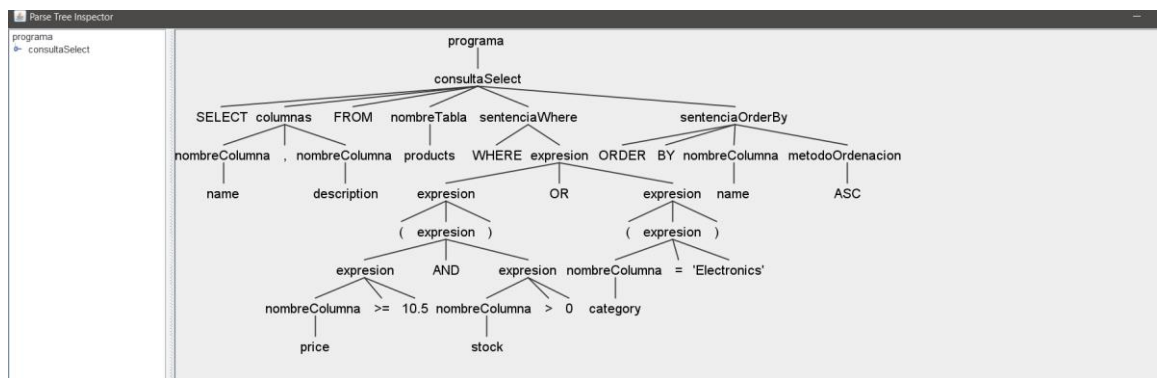
5) **SELECT name, age FROM employees WHERE age >= 30 ORDER BY name DESC**



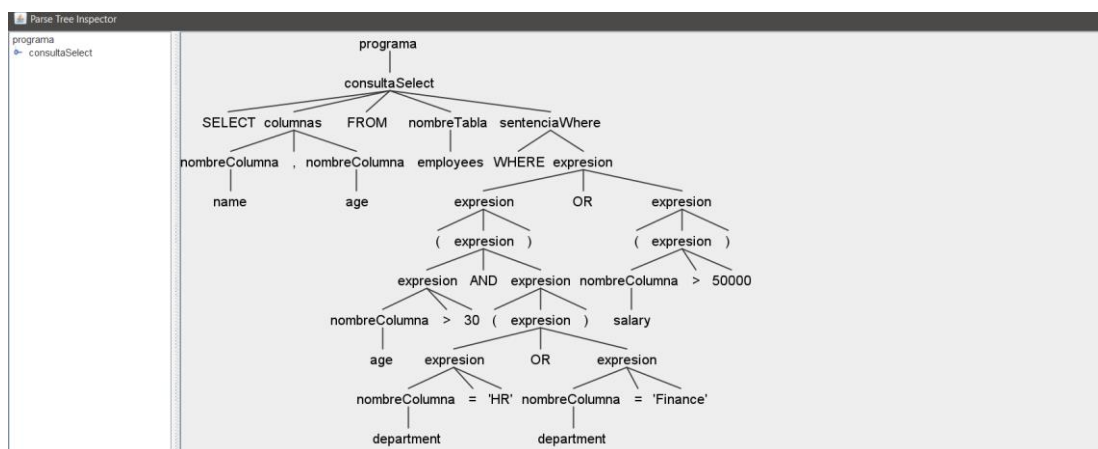
- 6) **SELECT * FROM products WHERE (price >= 100 AND stock > 0) OR (category = 'Electronics')**



- 7) **SELECT name, description FROM products WHERE (price >= 10.5 AND stock > 0) OR (category = 'Electronics') ORDER BY name ASC**



- 8) **SELECT name, age FROM employees WHERE (age > 30 AND (department = 'HR' OR department = 'Finance')) OR (salary > 50000)**



MEJORAS

En este caso hemos implementado 4 mejoras a nuestro código original que son necesarios para poder implementar la mayoría de las consultas de sql.

La primera mejora se basa en una búsqueda de elementos de las columnas por ejemplo si tenemos la columna de clientes y quiere solo su nombre le pasamos **clientes.nombre** , así podemos ser mas selectivos a la hora de hacer las consultas y poder acceder a atributos exclusivos.

La segunda y tercera mejora se basa en la implementación de **JOIN** y **ON** a nuestra sintaxis, **Join** se utiliza para combinar filas de dos o más tablas basándose en una condición especificada en la cláusula **ON**. La condición **ON** define la relación entre las columnas de las tablas involucradas en la unión, estableciendo cómo se deben emparejar las filas entre las tablas.

La cuarta mejora es la implementación del elemento **COUNT** que se utiliza para contar el número de filas que cumplen con ciertas condiciones en una tabla.

Vamos a ver las modificaciones que se han hecho en el lexer y en el parser:

```
lexer grammar gSqlMiniLexer;

//Aquí pondremos los simbolos terminales que tiene nuestro SqlMini

SELECT: 'SELECT';
FROM: 'FROM';
WHERE: 'WHERE';
ORDER: 'ORDER';
BY: 'BY';
ASC: 'ASC';
DESC: 'DESC';
COUNT: 'COUNT';
JOIN: 'JOIN';
ON: 'ON';
PUNTO: '.';
MAYORQUE: '>';
MAYORIGUALQUE: '>=';
IGUALQUE: '=';
ABREPARENTESIS: '(';
CIERRAPARENTESIS: ')';
NUMERO: '[0-9]+ ('[.]' [0-9]+)?; //Ahora tambien evaluamos los numeros decimales
STRING: '\\' ~'\\'* '\\';
COMA: ',';
AND: 'AND';
```

```
OR: 'OR';
ID: ([a-zA-Z]+) | '*'; // Identificadores (en este caso, nombres de columnas
o tablas)
WS: [ \t\r\n]+ -> skip; // Ignorar espacios en blanco
```

Se han declarado los 4 nuevos tokens count, join, on y Punto.

El parser quedaría:

```
parser grammar gSqlMiniParser;

options{
    tokenVocab = gSqlMiniLexer;
    language = Java;
}

programa: (consultaSelect)*;

consultaSelect : SELECT (columnas|count) FROM nombreTabla joinStatement?
sentenciaWhere? sentenciaOrderBy?;

columnas: nombreColumna (COMA nombreColumna )*;

nombreColumna: ID|nombreTabla (PUNTO nombreColumna);

nombreTabla: ID;

sentenciaWhere: WHERE expresion;

sentenciaOrderBy: ORDER BY nombreColumna metodoOrdenacion;

metodoOrdenacion: ASC | DESC;

count: COUNT ABREPARENTESIS nombreColumna CIERRAPARENTESIS;

joinStatement: JOIN nombreTabla ON nombreColumna IGUALQUE nombreColumna ;

expresion: ABREPARENTESIS expresion CIERRAPARENTESIS
    | expresion AND expresion
    | expresion OR expresion
    | nombreColumna IGUALQUE (NUMERO | STRING)
    | nombreColumna (MAYORQUE | MAYORIGUALQUE) NUMERO;
```

SELECT y FROM: La regla consultaSelect ahora permite seleccionar count o columnas específicas, y también puede incluir la cláusula JOIN.

Columnas y COUNT: La regla columnas ahora puede contener referencias a columnas de tablas específicas utilizando la notación nombreTabla.nombreColumna.

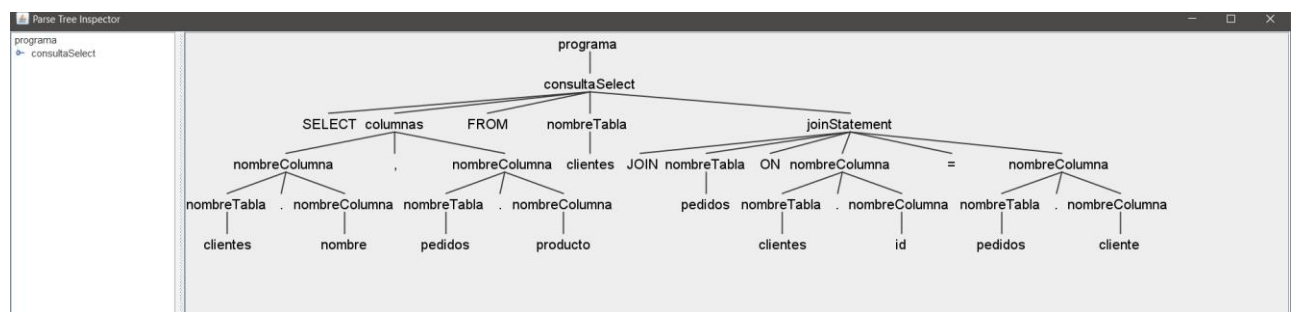
JOIN: Se ha agregado la regla joinStatement para manejar la cláusula JOIN en las consultas SELECT. Esta regla especifica cómo unir tablas en la consulta.

Expresiones: Las reglas de expresiones (expresion) se han actualizado para incluir referencias a columnas de tablas específicas.

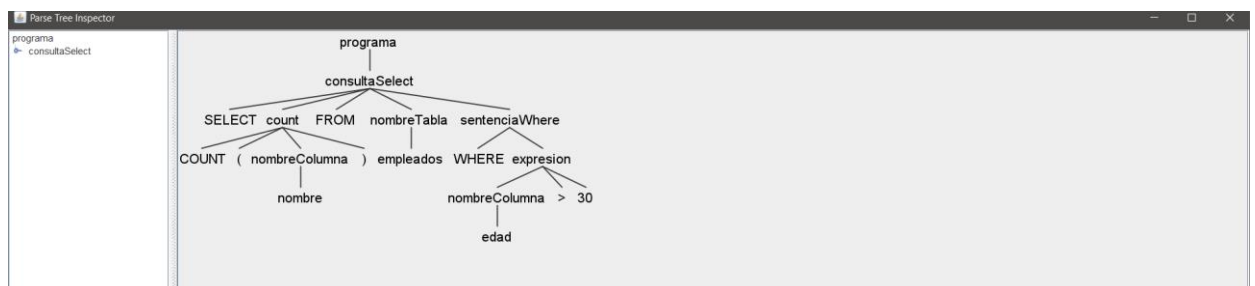
Ahora, este parser ampliado puede manejar consultas SELECT más complejas que involucran la selección de columnas de tablas específicas, la agregación de conteos, y la inclusión de la cláusula JOIN para combinar filas de diferentes tablas según una condición especificada.

Una vez compilado todo y analizado vamos a pasarle dos consultas sql para ver el funcionamiento:

SELECT clientes.nombre, pedidos.producto FROM clientes JOIN pedidos ON clientes.id = pedidos.cliente_id WHERE clientes.edad >= 18 ORDER BY clientes.nombre ASC



SELECT COUNT(nombre) FROM empleados WHERE edad > 30



1.1.2 Programa SQL

Para esta parte de la práctica se nos pedía la implementación de un programa codificado en el lenguaje de Java que, en primer lugar, lea una entrada cualquiera desde la consola, es decir, le escribamos en el terminal una de las posibles consultas y a continuación imprima en la consola el árbol generado correspondiente y a su vez que genere un archivo de texto plano con el árbol dibujado. Para realizar esta tarea se han implementado dos clases Java.

AnalizadorSQL.java

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

//import java.io.FileInputStream;
//import java.io.InputStream;

public class AnalizadorSQL {

    //Método de entrada por defecto

    public static void main(String[] args) throws Exception{

        CharStream input;
        if (args.length > 0) {
            // Si se proporciona un argumento en la línea de comandos, usarlo
            como entrada
            input = CharStreams.fromString(args[0]);
        } else {
            // Si no se proporciona un argumento, leer desde la entrada
            estándar
            input = CharStreams.fromStream(System.in);
        }

        //conectamos con el lexer
        gSqlMiniLexer lexer = new gSqlMiniLexer(input);

        //Inicializamos el canal de tokens
        CommonTokenStream tokens = new CommonTokenStream(lexer);

        //preparamos el parser
        gSqlMiniParser parser = new gSqlMiniParser(tokens);

        //generar arbol a partir del axioma de la gramatica
        ParseTree tree = parser.programa();
```

```
//Mostrar el arbol por consola:
//System.out.println(tree.toStringTree(parser));

//Recorrer el arbol:
//1-Inicializar un recorredor
//2- Inicializar mi escuchador
//3 Recorrer el arbol

//1
ParseTreeWalker walker = new ParseTreeWalker();
//2
ListenerSQL escuchador = new ListenerSQL(parser);
//
walker.walk(escuchador,tree);

}
}
```

Definimos la clase `AnalizadorSQL`: Esta clase contiene el método `main`, que es el punto de entrada del programa.

Determina la entrada de datos: El programa toma una cadena de entrada que representa una sentencia SQL, que puede ser proporcionada como argumento en la línea de comandos. Si se proporciona un argumento, utiliza esa cadena como entrada. Si no se proporciona ningún argumento, el programa lee la entrada estándar.

Inicializa el analizador léxico (Lexer): Se crea un objeto de la clase `gSqlMiniLexer` que se utiliza para convertir la cadena de entrada en tokens reconocibles.

Inicializa el canal de tokens: Los tokens generados por el lexer se almacenan en un `CommonTokenStream` para que el analizador sintáctico (parser) pueda consumirlos.

Prepara el analizador sintáctico (Parser): Se crea un objeto de la clase `gSqlMiniParser`, que se utiliza para construir el árbol de análisis sintáctico a partir de los tokens.

Genera el árbol de análisis: Se llama al método `programa()` del parser para construir el árbol de análisis a partir de la entrada SQL. El árbol representa la estructura de la sentencia SQL.

Prepara un recorrido del árbol: Se configura un recorredor de árbol (`ParseTreeWalker`) que permitirá explorar el árbol sintáctico.

Inicializa un escuchador (Listener): Se crea un objeto de la clase `ListenerSQL`, que es una implementación personalizada de un escuchador para el árbol sintáctico. Este escuchador se utilizará para realizar acciones específicas cuando se recorra el árbol.

Recorre el árbol sintáctico: Finalmente, se utiliza el recorredor para recorrer el árbol sintáctico y aplicar la lógica definida en el escuchador. Esto puede implicar realizar acciones específicas basadas en la estructura de la sentencia SQL, como validaciones o extracciones de información.

ListenerSQL.java

```
import java.io.FileWriter;
import java.io.PrintWriter;
import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.tree.ErrorNode;
import org.antlr.v4.runtime.tree.TerminalNode;

public class ListenerSQL extends gSqlMiniParserBaseListener{

    gSqlMiniParser parser;

    private int depth = 0;
    public ListenerSQL(gSqlMiniParser parser) {
        this.parser = parser;
    }
    @Override
    public void enterEveryRule(ParserRuleContext ctx) {
        String ruleName = parser.getRuleNames()[ctx.getRuleIndex()];
        String indentation = "    ".repeat(depth);
        String salida = indentation + ruleName + " ->";
        System.out.println(salida);
        escribirAST(salida);
        depth++;
    }
    @Override
    public void exitEveryRule(ParserRuleContext ctx) {
        depth--;
    }
    @Override
    public void visitTerminal(TerminalNode node) {
        String indentation = "    ".repeat(depth);
        String salida = indentation + node.getText();
        System.out.println(salida);
        escribirAST(salida);
    }
    public void escribirAST(String rule)
    {
```

```
        FileWriter fichero = null;
        PrintWriter pw = null;
        try
        {
            String ruta = "C:\\ANTLR\\gramaticas\\ArbolSQL.txt";
            fichero = new FileWriter(ruta,true);
            //pw = new PrintWriter(fichero);

            fichero.write(rule + "\n");

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                // Nuevamente aprovechamos el finally para
                // asegurarnos que se cierra el fichero.
                if (null != fichero)
                    fichero.close();
            } catch (Exception e2) {
                e2.printStackTrace();
            }
        }
    }
}
```

Clase ListenerSQL: Esta clase extiende gSqlMiniParserBaseListener y se utiliza como un escuchador personalizado para procesar eventos durante el recorrido del árbol sintáctico generado por ANTLR.

Constructor: La clase tiene un constructor que recibe un objeto gSqlMiniParser como argumento, que se utiliza para acceder al parser y obtener información sobre las reglas gramaticales.

Manejo de la profundidad del árbol: La variable depth se utiliza para realizar un seguimiento de la profundidad actual en el árbol sintáctico. Esto se usa para formatear la salida y la representación visual del árbol.

Método enterEveryRule: Este método se llama al entrar en cada regla del árbol sintáctico. Imprime el nombre de la regla y aumenta la profundidad.

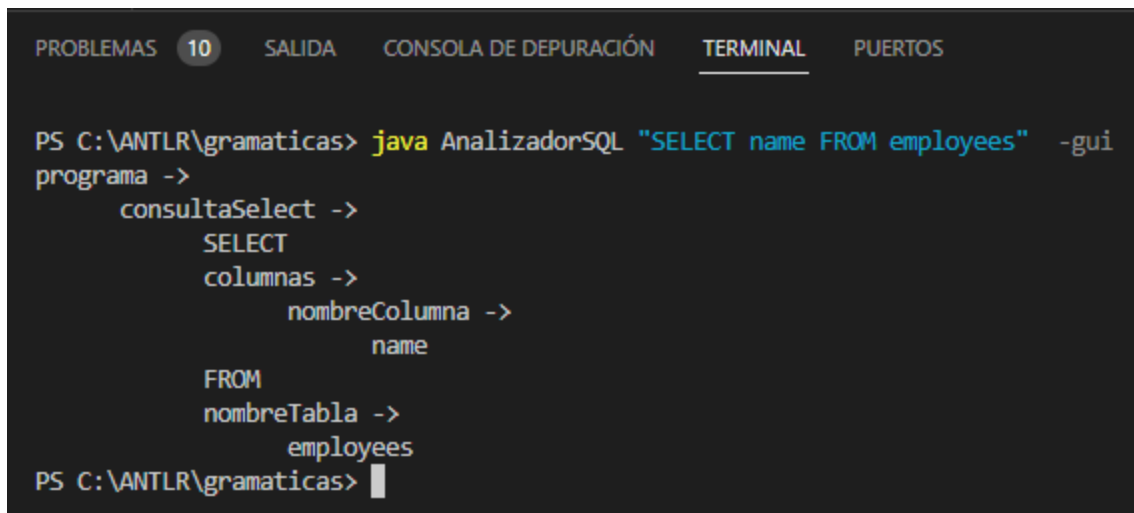
Método exitEveryRule: Este método se llama al salir de una regla del árbol sintáctico. Disminuye la profundidad.

Método visitTerminal: Este método se llama al visitar un nodo terminal del árbol sintáctico (por ejemplo, un token). Imprime el texto del nodo terminal.

Método escribirAST: Este método se utiliza para escribir información sobre las reglas y nodos visitados en el árbol sintáctico en un archivo llamado "ArbolSQL.txt" en la ruta "C:\ANTLR\gramaticas\". Utiliza un FileWriter y un PrintWriter para escribir la información en el archivo.

La clase ListenerSQL se utiliza para imprimir y almacenar información sobre el árbol de análisis sintáctico de sentencias SQL procesadas por ANTLR. Ayuda a visualizar la estructura del árbol y la información contenida en él, lo que puede ser útil para depurar y comprender el proceso de análisis sintáctico.

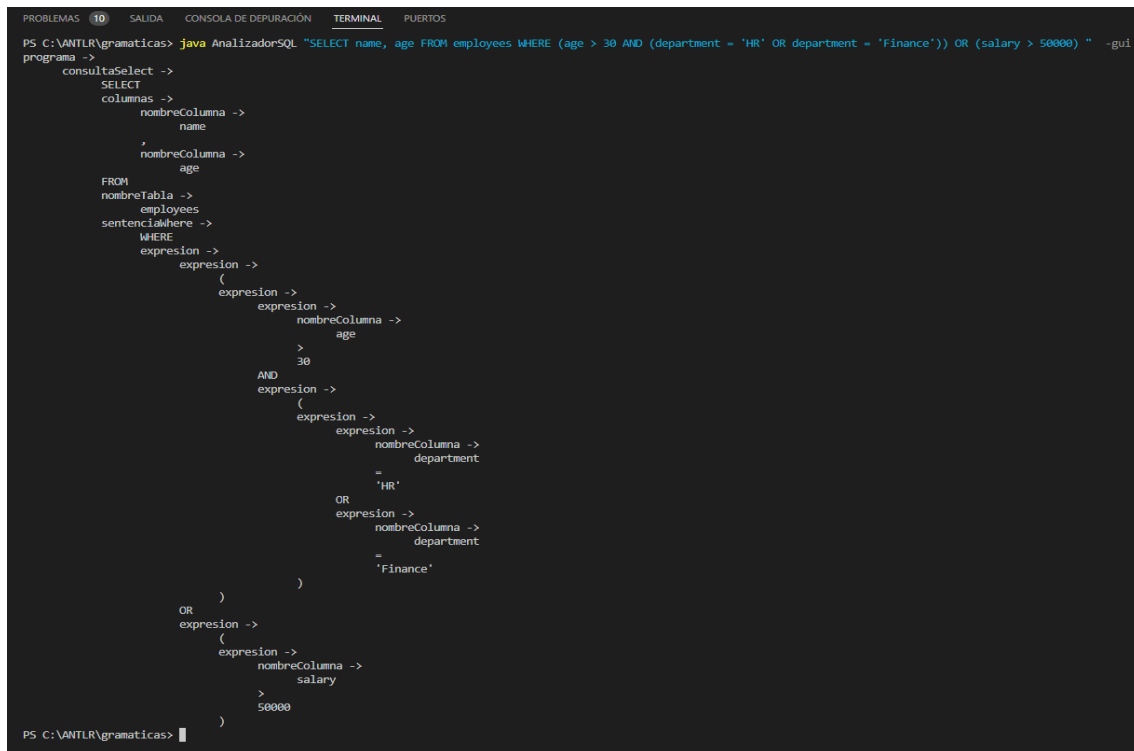
Ejemplo del funcionamiento:



```
PROBLEMAS 10 SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS

PS C:\ANTLR\gramaticas> java AnalizadorSQL "SELECT name FROM employees" -gui
programa ->
  consultaSelect ->
    SELECT
    columnas ->
      nombreColumna ->
        name
    FROM
    nombreTabla ->
      employees
PS C:\ANTLR\gramaticas>
```

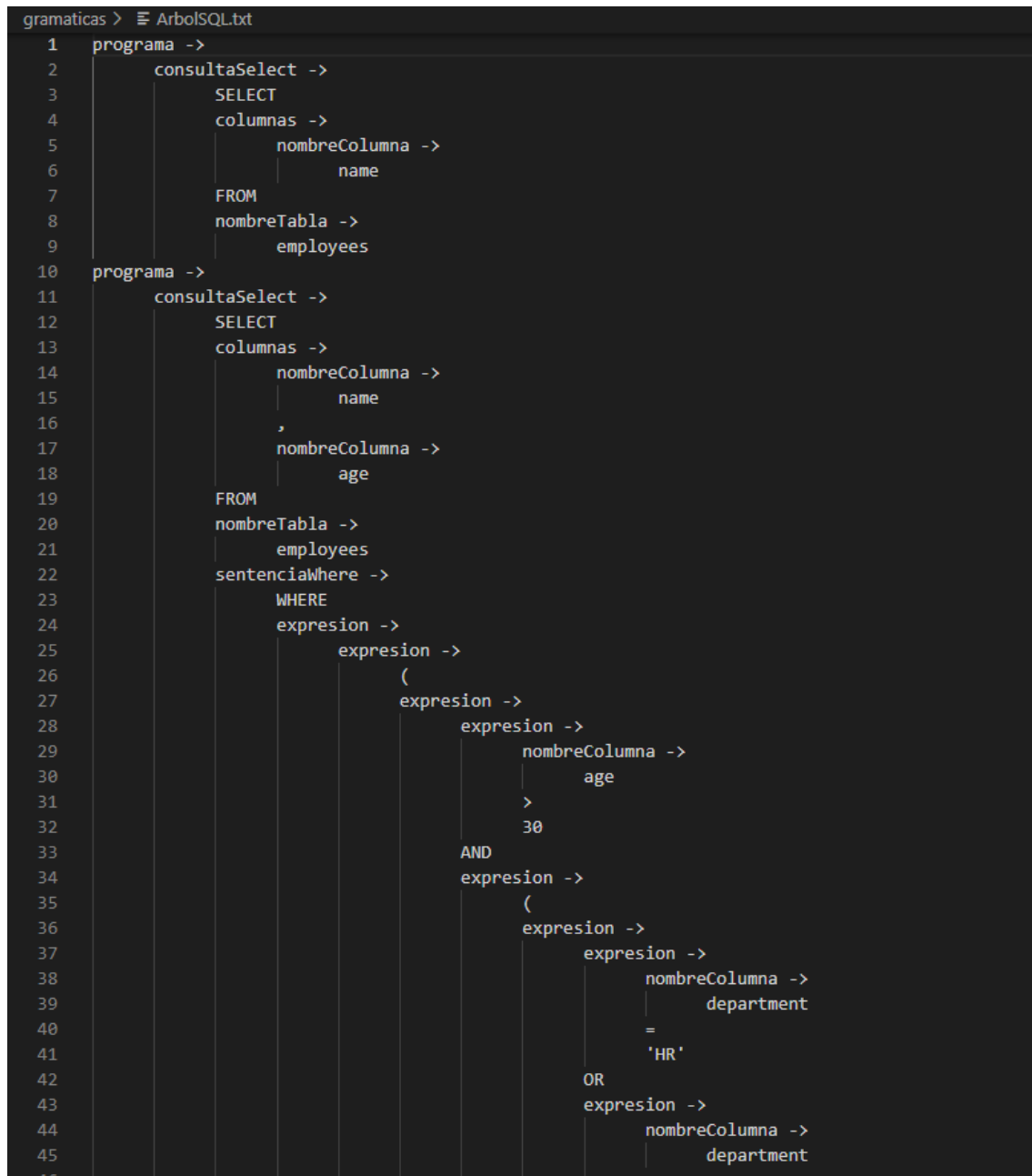
Aquí vemos como se imprime el árbol en consola tras pasarle una consulta de forma manual.



```
PROBLEMAS 10 SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS

PS C:\ANTLR\gramaticas> java AnalizadorSQL "SELECT name, age FROM employees WHERE (age > 30 AND (department = 'HR' OR department = 'Finance')) OR (salary > 50000) " -gui
programa ->
  consultaSelect ->
    SELECT
    columnas ->
      nombreColumna ->
        name
      nombreColumna ->
        age
    FROM
    nombreTabla ->
      employees
    sentenciaWhere ->
      WHERE
      expresion ->
        expresion ->
          (
            expresion ->
              expresion ->
                nombreColumna ->
                  age
                >
                30
              AND
              expresion ->
                (
                  expresion ->
                    expresion ->
                      nombreColumna ->
                        department
                      =
                      'HR'
                    OR
                    expresion ->
                      nombreColumna ->
                        department
                      =
                      'Finance'
                  )
                )
            )
          OR
          expresion ->
            (
              expresion ->
                nombreColumna ->
                  salary
                >
                50000
            )
        )
  )
PS C:\ANTLR\gramaticas>
```

Lo mismo con otro ejemplo.



Y aquí tenemos el fichero ArbolSQL.txt que se genera cuando hacemos las consultas. Como podemos observar, se han escrito en él las dos consultas que se han realizado previamente.

1.2 Linguine

Linguine es un lenguaje inventado que tendremos que reconocer creando un *lexer* y un *parser* adaptados a las sentencias del lenguaje para poder reconocer con éxito las distintas sentencias del mismo.

En cuanto al *lexer*, tenemos el siguiente código:

```
lexer grammar gLinguineLexer;  
lexer grammar gLinguineLexer;  
  
ASINGACION: 'let';  
IF: 'if';  
THEN: 'then';  
ELSE: 'else';  
FUNCION: 'fun';  
SHOW: 'show';  
MATCH: 'match';  
WITH: 'with';  
DEFAULT: '?';  
FLECHA: '->';  
COMILLAS: '"';  
OR: '|';  
IDENTIFICADOR: [a-zA-Z]+;  
IGUALQUE: '=';  
MAYORQUE: '>';  
MENORIGUALQUE: '<=';  
SUMA: '+';  
RESTA: '-';  
MULTIPLICACION: '*';  
DIVISION: '/';  
ABREPARENTESIS: '(';  
CIERRAPARENTESIS: ')';  
ENTERO: [0-9]+;  
COMA: ',';  
PUNTOYCOMA: ';';  
INTRO: '\r\n' -> skip;  
ESPACIO: [ \t\r\n]+ -> skip;
```

Lo que hacemos en esta parte, es definir todos los símbolos terminales que contiene el lenguaje Linguine.

Una vez que el *lexer* está completado, podremos manejar los tokens enviados por éste con el *parser*, el cual tiene el siguiente código:

```
parser grammar gLinguineParser;  
  
options{  
    tokenVocab = gLinguineLexer;  
    language = Java;
```

}

```

    tokenVocab = gLinguineLexer;
    language = Java;
}

programa: (instruccion PUNTOYCOMA saltoInstruccion?)*;

instruccion: asignacion
            | condicional
            | declaracionFuncion
            | llamadaFuncion
            | show
            | match
            | expresion
            | operando;

            | condicional
            | declaracionFuncion
            | llamadaFuncion
            | show
            | match
            | expresion
            | operando;

saltoInstruccion: INTRO;

asignacion: ASINGACION IDENTIFICADOR IGUALQUE (expresion | condicional |
match | llamadaFuncion);

condicional: IF ABREPARENTESIS expresion CIERRAPARENTESIS sentenciaThen
sentenciaElseIf sentenciaElse;

sentenciaElseIf: ELSE IF ABREPARENTESIS expresion CIERRAPARENTESIS
sentenciaThen;

sentenciaThen: THEN ABREPARENTESIS? instruccion CIERRAPARENTESIS?
(operador ABREPARENTESIS instruccion CIERRAPARENTESIS)*;

sentenciaElse: ELSE ABREPARENTESIS? instruccion CIERRAPARENTESIS?;

declaracionFuncion: FUNCION IDENTIFICADOR ABREPARENTESIS parametros?
CIERRAPARENTESIS FLECHA instruccion;

llamadaFuncion: IDENTIFICADOR ABREPARENTESIS parametros?
CIERRAPARENTESIS;

```

```
match: MATCH expresion WITH (INTRO? OR (operando | DEFAULT) | FLECHA
expresion)+;
```

```
match: MATCH (expresion | llamadaFuncion) WITH (INTRO? OR (operando |
DEFAULT) | FLECHA expresion)+;
```

```
parametros: (operando (COMA operando)*
| expresion);
| expresion);
```

```
string: COMILLAS IDENTIFICADOR COMILLAS;
```

```
show: SHOW ABREPARENTESIS instruccion CIERRAPARENTESIS;
```

```
expresion: (operando | llamadaFuncion | string) (operador (operando |
llamadaFuncion))*;
```

```
operador: SUMA | RESTA | MULTIPLICACION | DIVISION | MAYORQUE |
MENORIGUALQUE;
```

```
operando: IDENTIFICADOR | ENTERO;
```

El funcionamiento del **parser** es recibir los tokens del **lexer** y de ahí relacionarlos con las distintas reglas que tiene el mismo, las cuales son las siguientes:

- **Programa** --> Es la regla principal, lo que espera es una **instrucción** terminada por un **punto y coma** y un salto de instrucción que sería lo mismo que un intro (**\r\n**)
- **Instrucción** --> Una instrucción se encarga de relacionar todas las “acciones” que se pueden llevar a cabo en Linguine, que serían: **asignacion, condicional, declaracion de una función, llamada a una funcion, show, match, expresion cualquiera y operando**
- **Salto de instrucción** --> Básicamente con esto detectamos que hay otra instrucción más para analizar debido a que hay un **intro** (**\r\n**)
- **Asignación** --> Una asignación está definida por el token de **asignacion** (**let**) seguido de un nombre de variable que está definido mediante el token **identificador** seguido de un signo de igual, el cual vendría identificado por el token **igual que**, y detrás de ese igual podemos poner una **expresión, condicional, match** o una **llamada a una función**
- **Condicional** --> Representa la estructura de un **if** en Linguine. Debe empezar obligatoriamente por un token **if** seguido de una apertura de paréntesis identificado mediante el token de **abrir paréntesis**, que lo usaríamos para meter una **expresión** dentro, una vez cerrado el paréntesis con el token cerrar paréntesis, seguido de una **sentencia then** y después una **sentencia else**
- **Sentencia then** --> Define la forma que tiene un **then** dentro un **if**. Para ello debemos empezar por la palabra **then** que esta identificada por el token **then**,

con una **instrucción** cualquiera entre paréntesis (que pueden o no estar) y que también puede operar sobre una **instrucción**

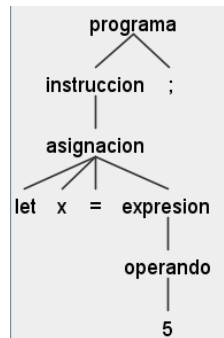
- **Sentencia else** --> Define la estructura de un *else* dentro de un *if*. Debemos empezar por un *else*, reconocido por el token **else**, y una **instrucción** que puede estar o no **entre paréntesis**
- **Declaración de una función** --> Para declarar una función en Linguine debemos empezar por la palabra *fun*, que está reconocida por el token **función**, un nombre de función, que está como un token **identificador**, y unos **paréntesis** donde pueden haber o no **parámetros**, seguidos de una flecha identificada por el token **flecha** que indicaría el código de la misma, que podremos meter ahí cualquier **instrucción** que queramos.
- **Llamada a una función** --> Para llamar a una función en Linguine tenemos que poner, en primer lugar, el nombre de la función, que sería un token **identificador** seguido de unos **paréntesis** que pueden o no tener **parámetros** dentro
- **Match** --> Representa la estructura de una instrucción *match* en Linguine. Primero debemos empezar introduciendo la palabra *match* que está reconocida por el token **match** seguido de una **expresión** a evaluar, después tendremos que añadir un *with* que está identificado por el token **with**, y detrás podemos poner mediante **intros** (o no) y un símbolo **OR** un operador con una **flecha** apuntando a una **expresión**
- **Parámetros** --> Aquí definimos los parámetros que le podemos meter a las funciones, que puede ser un **operando** o más (si están separados por **comas**) o bien una **expresión**.
- **String** --> Aquí definimos la forma de un string, que sería un **identificador** entre **comillas**.
- **Show** --> Para añadir un show tenemos que poner la palabra *show* que está identificado por el token **show** y después entre **paréntesis** una **instrucción** cualquiera.
- **Expresion** --> Una expresión puede ser un **operando** o una **llamada a función** junto con una **operación** a otro **operando** o a una **llamada a función**
- **Operador** --> Aquí recogemos qué puede ser un operador, es decir, una **suma**, **resta**, **multiplicación**, **división**, **mayor que** y **menor o igual que**
- **Operando** --> Un operando puede ser un **identificador** (letra) o bien un numero **entero**

Una vez hecho el **parser**, podemos compilar ambos archivos y podremos ejecutar todas las diferentes consultas visualizando sus respectivos tokens. A continuación, se muestran los tokens y ASTs respectivos de los ejemplos:

- **Ejemplo 1** --> *let x = 5;*




```
[@0,0:2='let',<'let'>,1:0]  
[@1,4:4='x',<IDENTIFICADOR>,1:4]  
[@2,6:6='=',<'='>,1:6]  
[@3,8:8='5',<ENTERO>,1:8]  
[@4,9:9=';',<'>',1:9]  
[@5,12:11='<EOF>',<EOF>,2:0]
```



- **Ejemplo 2** --> *let y = 2 + 3;*

```

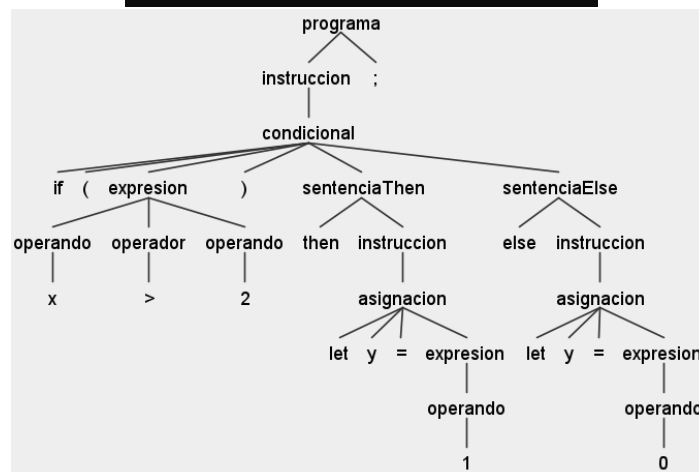
[ @0,0:2='let', <'let'>, 1:0 ]
[ @1,4:4='y', <IDENTIFICADOR>, 1:4 ]
[ @2,6:6='=', <'='>, 1:6 ]
[ @3,8:8='2', <ENTERO>, 1:8 ]
[ @4,10:10='+', <'+'>, 1:10 ]
[ @5,12:12='3', <ENTERO>, 1:12 ]
[ @6,13:13=';', <'>', 1:13 ]
[ @7,16:15='<EOF>', <EOF>, 2:0 ]
  
```



- **Ejemplo 3** --> *if (x > 2) then let y = 1 else let y = 0;*

```

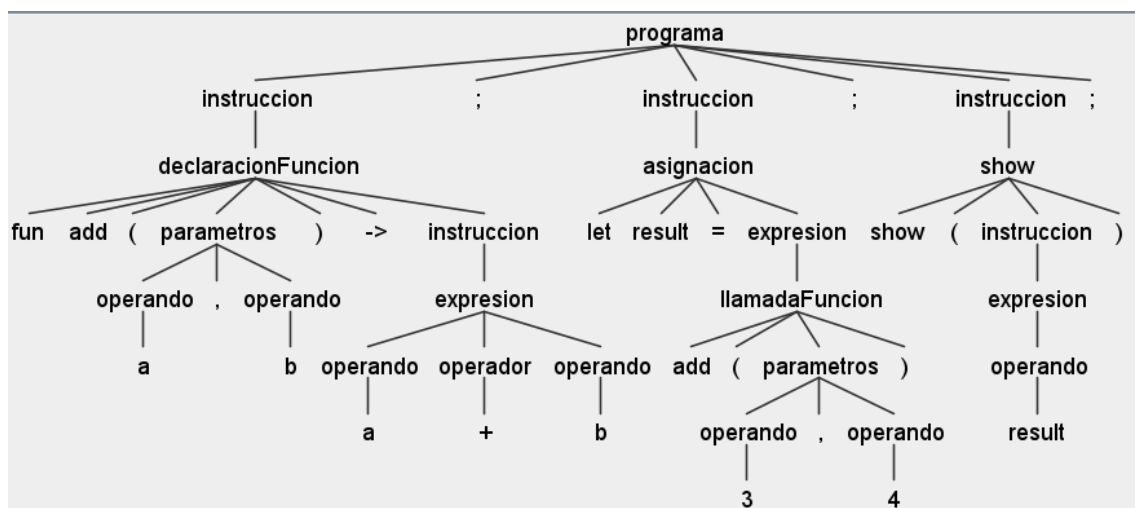
[ @0,0:1='if', <'if'>, 1:0 ]
[ @1,3:3='(', <'('>, 1:3 ]
[ @2,4:4='x', <IDENTIFICADOR>, 1:4 ]
[ @3,6:6='>', <'>'>, 1:6 ]
[ @4,8:8='2', <ENTERO>, 1:8 ]
[ @5,9:9=')', <')'>, 1:9 ]
[ @6,11:14='then', <'then'>, 1:11 ]
[ @7,16:18='let', <'let'>, 1:16 ]
[ @8,20:20='y', <IDENTIFICADOR>, 1:20 ]
[ @9,22:22='=', <'='>, 1:22 ]
[ @10,24:24='1', <ENTERO>, 1:24 ]
[ @11,26:29='else', <'else'>, 1:26 ]
[ @12,31:33='let', <'let'>, 1:31 ]
[ @13,35:35='y', <IDENTIFICADOR>, 1:35 ]
[ @14,37:37='=', <'='>, 1:37 ]
[ @15,39:39='0', <ENTERO>, 1:39 ]
[ @16,40:40=';', <'>', 1:40 ]
[ @17,43:42='<EOF>', <EOF>, 2:0 ]
  
```



- **Ejemplo 4** --> fun add(a, b) -> a + b; let result = add(3, 4); show(result)

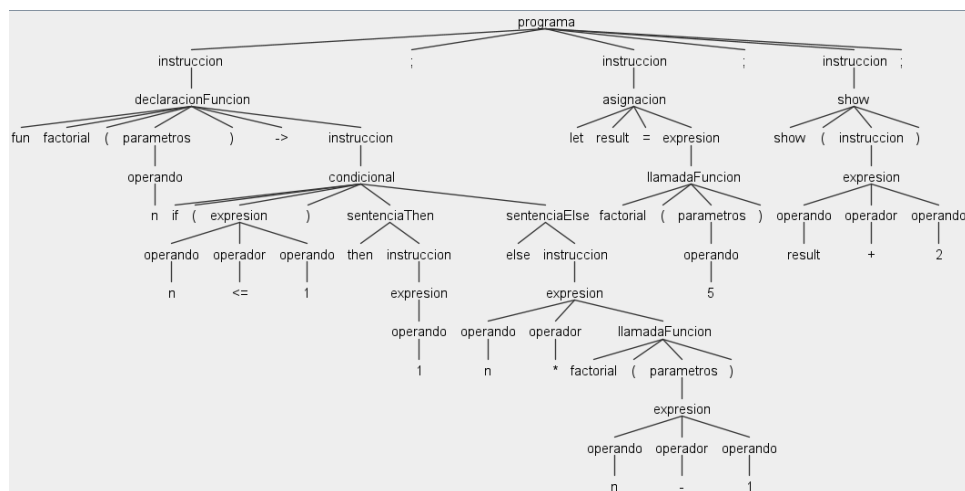
```

[ @0,0:2='fun', <'fun'>, 1:0 ]
[ @1,4:6='add', <IDENTIFICADOR>, 1:4 ]
[ @2,7:7='(', <'('>, 1:7 ]
[ @3,8:8='a', <IDENTIFICADOR>, 1:8 ]
[ @4,9:9=',', <','>, 1:9 ]
[ @5,11:11='b', <IDENTIFICADOR>, 1:11 ]
[ @6,12:12=')', <')'>, 1:12 ]
[ @7,14:15='->', <'>'>, 1:14 ]
[ @8,17:17='a', <IDENTIFICADOR>, 1:17 ]
[ @9,19:19='+', <'+'>, 1:19 ]
[ @10,21:21='b', <IDENTIFICADOR>, 1:21 ]
[ @11,22:22=';', <';'>, 1:22 ]
[ @12,25:27='let', <'let'>, 2:0 ]
[ @13,29:34='result', <IDENTIFICADOR>, 2:4 ]
[ @14,36:36='=', <'='>, 2:11 ]
[ @15,38:40='add', <IDENTIFICADOR>, 2:13 ]
[ @16,41:41='(', <'('>, 2:16 ]
[ @17,42:42='3', <ENTERO>, 2:17 ]
[ @18,43:43=',', <','>, 2:18 ]
[ @19,45:45='4', <ENTERO>, 2:20 ]
[ @20,46:46=')', <')'>, 2:21 ]
[ @21,47:47=';', <';'>, 2:22 ]
[ @22,50:53='show', <'show'>, 3:0 ]
[ @23,54:54='(', <'('>, 3:4 ]
[ @24,55:60='result', <IDENTIFICADOR>, 3:5 ]
[ @25,61:61=')', <')'>, 3:11 ]
[ @26,62:62=';', <';'>, 3:12 ]
[ @27,65:64='<EOF>', <EOF>, 4:0 ]
  
```



- **Ejemplo 5** --> fun factorial(n) -> if (n <= 1) then 1 else n * factorial(n - 1); let result = factorial(5); show(result + 2);

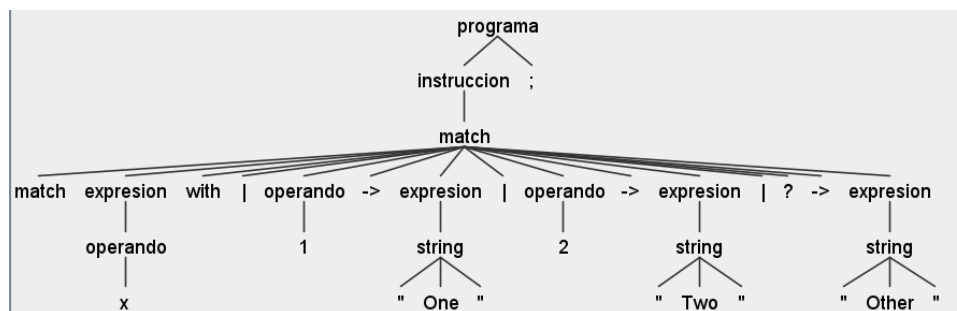
```
[@0,0:2='fun',<'fun'>,1:0]
[1,4:12='factorial',<IDENTIFICADOR>,1:4]
[2,13:13='(',<'('>,1:13]
[3,14:14='n',<IDENTIFICADOR>,1:14]
[4,15:15=')',<'>'>,1:15]
[5,17:18='->',<'>'>,1:17]
[6,20:21='if',<'if'>,1:20]
[7,23:23='(',<'('>,1:23]
[8,24:24='n',<IDENTIFICADOR>,1:24]
[9,26:27='<=','<='>,1:26]
[10,29:29='1',<ENTERO>,1:29]
[11,30:30=')',<'>'>,1:30]
[12,32:35='then',<'then'>,1:32]
[13,37:37='1',<ENTERO>,1:37]
[14,39:42='else',<'else'>,1:39]
[15,44:44='n',<IDENTIFICADOR>,1:44]
[16,46:46='*',<'*>,1:46]
[17,48:56='factorial',<IDENTIFICADOR>,1:48]
[18,57:57='(',<'('>,1:57]
[19,58:58='n',<IDENTIFICADOR>,1:58]
[20,60:60='-',<'-'>,1:60]
[21,62:62='1',<ENTERO>,1:62]
[22,63:63=')',<'>'>,1:63]
[23,64:64=';',<'>'>,1:64]
[24,67:69='let',<'let'>,2:0]
[25,71:76='result',<IDENTIFICADOR>,2:4]
[26,78:78='=',<'='>,2:11]
[27,80:88='factorial',<IDENTIFICADOR>,2:13]
[28,89:89='(',<'('>,2:22]
[29,90:90='5',<ENTERO>,2:23]
[30,91:91=')',<'>'>,2:24]
[31,92:92=';',<'>'>,2:25]
[32,95:98='show',<'show'>,3:0]
[33,99:99='(',<'('>,3:4]
[34,100:105='result',<IDENTIFICADOR>,3:5]
[35,107:107='+',<'+'>,3:12]
[36,109:109='2',<ENTERO>,3:14]
[37,110:110=')',<'>'>,3:15]
[38,111:111=';',<'>'>,3:16]
[39,114:113='<EOF>',<EOF>,4:0]
```



- Ejemplo 6** --> match x with | 1 -> "One" | 2 -> "Two" | ? -> "Other";

```

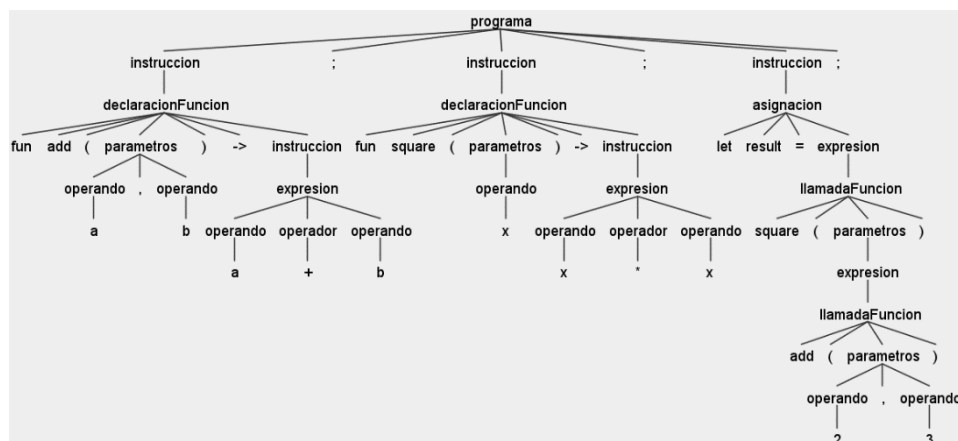
[ @0, 0:4= 'match', <'match'>, 1:0 ]
[ @1, 6:6= 'x', <IDENTIFICADOR>, 1:6 ]
[ @2, 8:11= 'with', <'with'>, 1:8 ]
[ @3, 14:14= '|', <'|'>, 2:0 ]
[ @4, 16:16= '1', <ENTERO>, 2:2 ]
[ @5, 18:19= '->', <'>'>, 2:4 ]
[ @6, 21:21= '"', <'"'>, 2:7 ]
[ @7, 22:24= 'One', <IDENTIFICADOR>, 2:8 ]
[ @8, 25:25= '"', <'"'>, 2:11 ]
[ @9, 28:28= '|', <'|'>, 3:0 ]
[ @10, 30:30= '2', <ENTERO>, 3:2 ]
[ @11, 32:33= '->', <'>'>, 3:4 ]
[ @12, 35:35= '"', <'"'>, 3:7 ]
[ @13, 36:38= 'Two', <IDENTIFICADOR>, 3:8 ]
[ @14, 39:39= '"', <'"'>, 3:11 ]
[ @15, 42:42= '|', <'|'>, 4:0 ]
[ @16, 44:44= '?', <'?'>, 4:2 ]
[ @17, 46:47= '->', <'>'>, 4:4 ]
[ @18, 49:49= '"', <'"'>, 4:7 ]
[ @19, 50:54= 'Other', <IDENTIFICADOR>, 4:8 ]
[ @20, 55:55= '"', <'"'>, 4:13 ]
[ @21, 56:56= ';', <'>'>, 4:14 ]
[ @22, 59:58= '<EOF>', <EOF>, 5:0 ]
  
```



- Ejemplo 7** --> *fun add(a, b) -> a + b; fun square(x) -> x * x; let result = square(add(2, 3));*

```

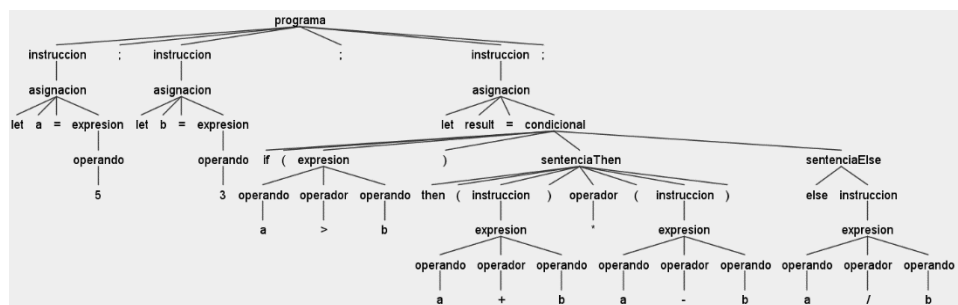
[ @0,0:2='fun', <'fun'>, 1:0 ]
[ @1,4:6='add', <IDENTIFICADOR>, 1:4 ]
[ @2,7:7='(', <'('>, 1:7 ]
[ @3,8:8='a', <IDENTIFICADOR>, 1:8 ]
[ @4,9:9=',', <','>, 1:9 ]
[ @5,11:11='b', <IDENTIFICADOR>, 1:11 ]
[ @6,12:12=')', <')'>, 1:12 ]
[ @7,14:15='->', <'>'>, 1:14 ]
[ @8,17:17='a', <IDENTIFICADOR>, 1:17 ]
[ @9,19:19='+', <'+'>, 1:19 ]
[ @10,21:21='b', <IDENTIFICADOR>, 1:21 ]
[ @11,22:22=';', <';'>, 1:22 ]
[ @12,25:27='fun', <'fun'>, 2:0 ]
[ @13,29:34='square', <IDENTIFICADOR>, 2:4 ]
[ @14,35:35='(', <'('>, 2:10 ]
[ @15,36:36='x', <IDENTIFICADOR>, 2:11 ]
[ @16,37:37=')', <')'>, 2:12 ]
[ @17,39:40='->', <'>'>, 2:14 ]
[ @18,42:42='x', <IDENTIFICADOR>, 2:17 ]
[ @19,44:44='*', <'*'>, 2:19 ]
[ @20,46:46='x', <IDENTIFICADOR>, 2:21 ]
[ @21,47:47=';', <';'>, 2:22 ]
[ @22,50:52='let', <'let'>, 3:0 ]
[ @23,54:59='result', <IDENTIFICADOR>, 3:4 ]
[ @24,61:61='=', <'='>, 3:11 ]
[ @25,63:68='square', <IDENTIFICADOR>, 3:13 ]
[ @26,69:69='(', <'('>, 3:19 ]
[ @27,70:72='add', <IDENTIFICADOR>, 3:20 ]
[ @28,73:73='(', <'('>, 3:23 ]
[ @29,74:74='2', <ENTERO>, 3:24 ]
[ @30,75:75=',', <','>, 3:25 ]
[ @31,77:77='3', <ENTERO>, 3:27 ]
[ @32,78:78=')', <')'>, 3:28 ]
[ @33,79:79=')', <')'>, 3:29 ]
[ @34,80:80=';', <';'>, 3:30 ]
[ @35,83:82='<EOF>', <EOF>, 4:0 ]
  
```



- Ejemplo 8** --> *let a = 5; let b = 3; let result = if (a > b) then (a + b) * (a - b) else a / b;*

```

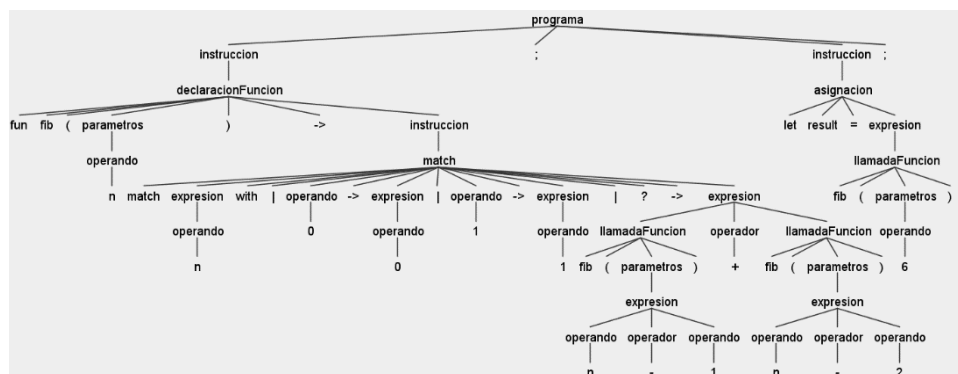
[ @0,0:2='let', <'let'>, 1:0 ]
[ @1,4:4='a', <IDENTIFICADOR>, 1:4 ]
[ @2,6:6='=', <'='>, 1:6 ]
[ @3,8:8='5', <ENTERO>, 1:8 ]
[ @4,9:9=';', <'>', 1:9 ]
[ @5,12:14='let', <'let'>, 2:0 ]
[ @6,16:16='b', <IDENTIFICADOR>, 2:4 ]
[ @7,18:18='=', <'='>, 2:6 ]
[ @8,20:20='3', <ENTERO>, 2:8 ]
[ @9,21:21=';', <'>', 2:9 ]
[ @10,24:26='let', <'let'>, 3:0 ]
[ @11,28:33='result', <IDENTIFICADOR>, 3:4 ]
[ @12,35:35='=', <'='>, 3:11 ]
[ @13,37:38='if', <'if'>, 3:13 ]
[ @14,40:40='(', <'('>, 3:16 ]
[ @15,41:41='a', <IDENTIFICADOR>, 3:17 ]
[ @16,43:43='>', <'>'>, 3:19 ]
[ @17,45:45='b', <IDENTIFICADOR>, 3:21 ]
[ @18,46:46=')', <')'>, 3:22 ]
[ @19,48:51='then', <'then'>, 3:24 ]
[ @20,53:53='(', <'('>, 3:29 ]
[ @21,54:54='a', <IDENTIFICADOR>, 3:30 ]
[ @22,56:56='+', <'+'>, 3:32 ]
[ @23,58:58='b', <IDENTIFICADOR>, 3:34 ]
[ @24,59:59=')', <')'>, 3:35 ]
[ @25,61:61='*', <'*'>, 3:37 ]
[ @26,63:63='(', <'('>, 3:39 ]
[ @27,64:64='a', <IDENTIFICADOR>, 3:40 ]
[ @28,66:66='-', <'-'>, 3:42 ]
[ @29,68:68='b', <IDENTIFICADOR>, 3:44 ]
[ @30,69:69=')', <')'>, 3:45 ]
[ @31,71:74='else', <'else'>, 3:47 ]
[ @32,76:76='a', <IDENTIFICADOR>, 3:52 ]
[ @33,78:78='/', <'/'>, 3:54 ]
[ @34,80:80='b', <IDENTIFICADOR>, 3:56 ]
[ @35,81:81=';', <'>', 3:57 ]
[ @36,84:83='<EOF>', <EOF>, 4:0 ]
  
```



- Ejemplo 9** --> `fun fib(n) -> match n with | 0 -> 0 | 1 -> 1 | ? -> fib(n - 1) + fib(n - 2); let result = fib(6);`

```

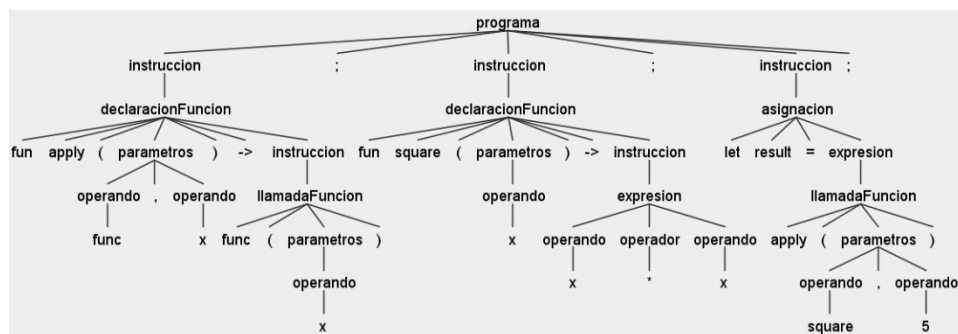
[0,0:2='fun',<'fun'>,1:0]
[1,4:6='fib',<IDENTIFICADOR>,1:4]
[2,7:7='(',<'(>',1:7]
[3,8:8='n',<IDENTIFICADOR>,1:8]
[4,9:9=')',<'>',1:9]
[5,11:12='->',<'>',1:11]
[6,14:18='match',<'match'>,1:14]
[7,20:20='n',<IDENTIFICADOR>,1:20]
[8,22:25='with',<'with'>,1:22]
[9,28:28='|',<'|'>,2:0]
[10,30:30='0',<ENTERO>,2:2]
[11,32:33='->',<'>',2:4]
[12,35:35='0',<ENTERO>,2:7]
[13,38:38='|',<'|'>,3:0]
[14,40:40='1',<ENTERO>,3:2]
[15,42:43='->',<'>',3:4]
[16,45:45='1',<ENTERO>,3:7]
[17,48:48='|',<'|'>,4:0]
[18,50:50='?',<'?'>,4:2]
[19,52:53='->',<'>',4:4]
[20,55:57='fib',<IDENTIFICADOR>,4:7]
[21,58:58='(',<'(>',4:10]
[22,59:59='n',<IDENTIFICADOR>,4:11]
[23,61:61='-',<'>',4:13]
[24,63:63='1',<ENTERO>,4:15]
[25,64:64=')',<'>',4:16]
[26,66:66='+',<'>',4:18]
[27,68:70='fib',<IDENTIFICADOR>,4:20]
[28,71:71='(',<'(>',4:23]
[29,72:72='n',<IDENTIFICADOR>,4:24]
[30,74:74='-',<'>',4:26]
[31,76:76='2',<ENTERO>,4:28]
[32,77:77=')',<'>',4:29]
[33,78:78=';',<'>',4:30]
[34,81:83='let',<'let'>,5:0]
[35,85:90='result',<IDENTIFICADOR>,5:4]
[36,92:92='=',<'='>,5:11]
[37,94:96='fib',<IDENTIFICADOR>,5:13]
[38,97:97='(',<'(>',5:16]
[39,98:98='6',<ENTERO>,5:17]
[40,99:99=')',<'>',5:18]
[41,100:100=';',<'>',5:19]
[42,103:102='<EOF>',<EOF>,6:0]
  
```



- Ejemplo 10** --> *fun apply(func, x) -> func(x); fun square(x) -> x * x; let result = apply(square, 5);*

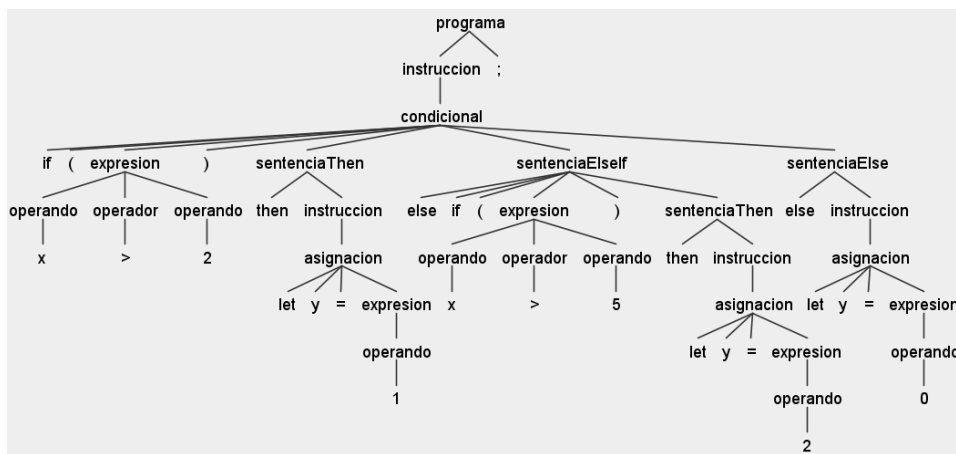
```

[0,0:2='fun',<'fun'>,1:0]
[1,4:8='apply',<IDENTIFICADOR>,1:4]
[2,9:9='(',<'('>,1:9]
[3,10:13='func',<IDENTIFICADOR>,1:10]
[4,14:14=',',<','>,1:14]
[5,16:16='x',<IDENTIFICADOR>,1:16]
[6,17:17=')',<')'>,1:17]
[7,19:20='->',<'>'>,1:19]
[8,22:25='func',<IDENTIFICADOR>,1:22]
[9,26:26='(',<'('>,1:26]
[10,27:27='x',<IDENTIFICADOR>,1:27]
[11,28:28=')',<')'>,1:28]
[12,29:29=';',<';'>,1:29]
[13,32:34='fun',<'fun'>,2:0]
[14,36:41='square',<IDENTIFICADOR>,2:4]
[15,42:42='(',<'('>,2:10]
[16,43:43='x',<IDENTIFICADOR>,2:11]
[17,44:44=')',<')'>,2:12]
[18,46:47='->',<'>'>,2:14]
[19,49:49='x',<IDENTIFICADOR>,2:17]
[20,51:51='*',<'*'>,2:19]
[21,53:53='x',<IDENTIFICADOR>,2:21]
[22,54:54=';',<';'>,2:22]
[23,57:59='let',<'let'>,3:0]
[24,61:66='result',<IDENTIFICADOR>,3:4]
[25,68:68='=',<'='>,3:11]
[26,70:74='apply',<IDENTIFICADOR>,3:13]
[27,75:75='(',<'('>,3:18]
[28,76:81='square',<IDENTIFICADOR>,3:19]
[29,82:82=',',<','>,3:25]
[30,84:84='5',<ENTERO>,3:27]
[31,85:85=')',<')'>,3:28]
[32,86:86=';',<';'>,3:29]
[33,89:88='<EOF>',<EOF>,4:0]
  
```

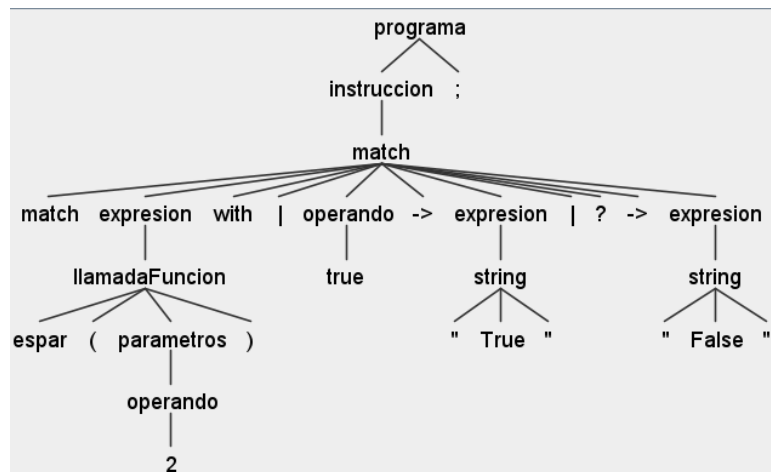


Para la **parte avanzada**, hemos implementado las siguientes **mejoras** a nuestro lenguaje Linguine:

- Hemos implementado la opción de poner **else if** dentro de los ifs para evaluar varias condiciones, hemos añadido lo siguiente:
 - En la regla de *condicional* dentro del *parser* hemos indicado que entre un bloque then y un bloque else, puede haber una sentencia else if, que contiene un else y un condicional detrás: `sentenciaElseIf: ELSE condicional;`
 - Para demostrar que la mejora funciona he puesto el siguiente ejemplo: *if (x > 2) then let y = 1 else if (x > 5) then let y = 2 else let y = 0;*. Que genera el siguiente árbol:



- Como segunda parte de la mejora, hemos implementado que un **match pueda usar el resultado de una función como parámetro de entrada**, para ello simplemente hemos añadido a la regla de match que también pueda leer una llamada a una función
 - La regla del match quedaría tal que así: `match: MATCH (expresion | llamadaFuncion) WITH (INTRO? OR (operando | DEFAULT) | FLECHA expresion)+;`
 - Como ejemplo he puesto la siguiente instrucción: *match espar(2) with | true -> "True" | ? -> "False";*
 - Esto genera el siguiente árbol:



1.2.2 Programa Linguine

Analizador Linguine.java

```

import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

import java.io.FileInputStream;
import java.io.InputStream;

public class AnalizadorLinguine {

    //Método de entrada por defecto

    public static void main(String[] args) throws Exception{

        //Inicializamos la entrada de datos
        String inputFile=null;
        if (args.length>0) inputFile=args[0];

        //inicializamos los streams de datos
        InputStream is = System.in;
        if (inputFile!=null) is = new FileInputStream(inputFile);

        //Inicializo ANTLR con el fichero
        ANTLRInputStream input = new ANTLRInputStream (is);

        //conectamos con el lexer
        glinguineLexer lexer = new glinguineLexer(input);
  
```

```
//Inicializamos el canal de tokens
CommonTokenStream tokens = new CommonTokenStream(lexer);

//preparamos el parser
glinguineParser parser = new gLinguineParser(tokens);

//generar arbol a partir del axioma de la gramatica
ParseTree tree = parser.programa();

//Mostrar el arbol por consola:
System.out.println(tree.toStringTree(parser));

//Recorrer el arbol:
//1-Inicializar un recorredor
//2- Inicializar mi escuchador
//3 Recorrer el arbol

//1
ParseTreeWalker walker = new ParseTreeWalker();
//2
ListenerLinguine escuchador = new ListenerLinguine(parser);
//
walker.walk(escuchador,tree);

}
}
```

Define la clase AnalizadorLinguine: Esta clase contiene el método main, que es el punto de entrada del programa.

Inicializa la entrada de datos: El programa determina si se proporciona un archivo de entrada como argumento en la línea de comandos y, si es así, lo utiliza. Si no se proporciona un archivo de entrada, lee desde la entrada estándar. Este punto es el que es diferente en el Analizador Linguine con respecto al Analizador SQL, aquí utilizamos el `FileInputStream`, ya que estamos utilizando un archivo como entrada.

Inicializa los flujos de entrada: Se inicializa un flujo de entrada (`InputStream`) que se utilizará para obtener los datos de entrada. Dependiendo de si se proporciona un archivo de entrada o no, el flujo se configura para leer desde el archivo o desde la entrada estándar.

Inicializa ANTLR con la entrada de datos: Se crea un objeto `ANTLRInputStream` a partir del flujo de entrada. Este objeto se utiliza como entrada para el analizador léxico ANTLR.

Conecta con el lexer: Se crea un objeto de la clase `gLinguineLexer`, que se utiliza para convertir la entrada en tokens reconocibles.

Inicializa el canal de tokens: Los tokens generados por el lexer se almacenan en un `CommonTokenStream` para que el analizador sintáctico (parser) pueda consumirlos.

Prepara el parser: Se crea un objeto de la clase `gLinguineParser`, que se utiliza para construir el árbol de análisis sintáctico a partir de los tokens.

Genera el árbol de análisis: Se llama al método `programa()` del parser para construir el árbol de análisis a partir de la entrada del lenguaje de programación. El árbol representa la estructura del programa.

Prepara un recorrido del árbol: Se configura un recorridor de árbol (`ParseTreeWalker`) que permitirá explorar el árbol sintáctico.

Inicializa un escuchador (Listener): Se crea un objeto de la clase `ListenerLinguine`, que es una implementación personalizada de un escuchador para el árbol sintáctico. Este escuchador se utilizará para realizar acciones específicas cuando se recorra el árbol.

ListenerLinguine.java

```
import java.io.FileWriter;
import java.io.PrintWriter;

import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.tree.ErrorNode;
import org.antlr.v4.runtime.tree.TerminalNode;

public class ListenerLinguine extends gLinguineParserBaseListener{

    gLinguineParser parser;

    private int depth = 0;
    public ListenerLinguine(gLinguineParser parser) {
        this.parser = parser;
    }

    @Override
    public void enterEveryRule(ParserRuleContext ctx) {
        String ruleName = parser.getRuleNames()[ctx.getRuleIndex()];
        String indentation = "    ".repeat(depth);
        String salida = indentation + ruleName + " ->";
        System.out.println(salida);
        escribirAST(salida);
        depth++;
    }
}
```

```
}

@Override
public void exitEveryRule(ParserRuleContext ctx) {
    depth--;
}

@Override
public void visitTerminal(TerminalNode node) {
    String indentation = "    ".repeat(depth);
    String salida = indentation + node.getText();
    System.out.println(salida);
    escribirAST(salida);
}

public void escribirAST(String rule)
{
    FileWriter fichero = null;
    PrintWriter pw = null;
    try
    {
        String ruta = "C:\\ANTLR\\gramaticas\\ArbolLinguine.txt";
        fichero = new FileWriter(ruta,true);
        //pw = new PrintWriter(fichero);

        fichero.write(rule + "\n");

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            // Nuevamente aprovechamos el finally para
            // asegurarnos que se cierra el fichero.
            if (null != fichero)
                fichero.close();
        } catch (Exception e2) {
            e2.printStackTrace();
        }
    }
}
```

Ejemplo del funcionamiento:

Este es el fichero Prueba.txt que le pasaremos al Analizador.

```
gramaticas > Prueba.txt
1  let x = 4 + 1;
2
3  fun restar(x) -> 5 - x;
4
5  match x with
6  | 1 -> "One"
7  | 2 -> "Two"
8  | ? -> "Other";
9
10 fun add(a, b) -> a + b;
11 fun square(x) -> x * x;
12 let result = square(add(2, 3));
13
14 let a = 5;
15 let b = 3;
16 let result = if (a > b) then (a + b) * (a - b) else a / b;
```

PROBLEMAS 10 SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS

)

PS C:\VANTLR\gramaticas> java AnalizadorLinguine Prueba.txt -gui

Y este es el fichero ArbolLinguine.txt que nos genera el árbol en un fichero de texto plano.

```

gramaticas > ArbolLinguine.txt
1  programa ->
2      instruccion ->
3          asignacion ->
4              let
5              x
6              =
7              instruccion ->
8                  expresion ->
9                      operando ->
10                         4
11                     operador ->
12                         +
13                     operando ->
14                         1
15      ;
16      instruccion ->
17          declaracionFuncion ->
18              fun
19              restar
20              (
21              parametros ->
22                  operando ->
23                      x
24              )
25      ->
26      instruccion ->
27          expresion ->
28              operando ->
29                  5
30              operador ->
31                  -
32              operando ->
33                  x
34      ;
35      instruccion ->
36          match ->
37              match
38              expresion ->
39                  operando ->
40                      x
41              with
42
43              |
44              operando ->
45                  1
46              ->
47              expresion ->
48                  operando ->
49                      0
50

```



```

49         |           |           |           |           |           |           |           |           |           |
50         |           |           |           |           |           |           |           |           |           |
51         |           |           |           |           |           |           |           |           |           |
52         |           |           |           |           |           |           |           |           |           |
53         |           |           |           |           |           |           |           |           |           |
54         |           |           |           |           |           |           |           |           |           |
55         |           |           |           |           |           |           |           |           |           |
56         |           |           |           |           |           |           |           |           |           |
57         |           |           |           |           |           |           |           |           |           |
58         |           |           |           |           |           |           |           |           |           |
59         |           |           |           |           |           |           |           |           |           |
60         |           |           |           |           |           |           |           |           |           |
61         |           |           |           |           |           |           |           |           |           |
62         |           |           |           |           |           |           |           |           |           |
63         |           |           |           |           |           |           |           |           |           |
64         |           |           |           |           |           |           |           |           |           |
65         |           |           |           |           |           |           |           |           |           |
66         |           |           |           |           |           |           |           |           |           |
67         |           |           |           |           |           |           |           |           |           |
68         |           |           |           |           |           |           |           |           |           |
69         |           |           |           |           |           |           |           |           |           |
70         |           |           |           |           |           |           |           |           |           |
71         |           |           |           |           |           |           |           |           |           |
72         |           |           |           |           |           |           |           |           |           |
73         |           |           |           |           |           |           |           |           |           |
74         |           |           |           |           |           |           |           |           |           |
75         |           |           |           |           |           |           |           |           |           |
76         |           |           |           |           |           |           |           |           |           |
77         |           |           |           |           |           |           |           |           |           |
78         |           |           |           |           |           |           |           |           |           |
79         |           |           |           |           |           |           |           |           |           |
80         |           |           |           |           |           |           |           |           |           |
81         |           |           |           |           |           |           |           |           |           |
82         |           |           |           |           |           |           |           |           |           |
83         |           |           |           |           |           |           |           |           |           |
84         |           |           |           |           |           |           |           |           |           |
85         |           |           |           |           |           |           |           |           |           |
86         |           |           |           |           |           |           |           |           |           |
87         |           |           |           |           |           |           |           |           |           |
88         |           |           |           |           |           |           |           |           |           |
89         |           |           |           |           |           |           |           |           |           |
90         |           |           |           |           |           |           |           |           |           |
91         |           |           |           |           |           |           |           |           |           |
92         |           |           |           |           |           |           |           |           |           |
93         |           |           |           |           |           |           |           |           |           |
94         |           |           |           |           |           |           |           |           |           |
95         |           |           |           |           |           |           |           |           |           |
96         |           |           |           |           |           |           |           |           |           |
97         |           |           |           |           |           |           |           |           |           |
98         |           |           |           |           |           |           |           |           |           |

```

```

gramaticas > ≡ ArbolLinguine.txt
168      result
169      =
170      instruccion ->
171      |
172      |   condicional ->
173      |   |
174      |   |   if
175      |   |   |
176      |   |   |   (
177      |   |   |   |
178      |   |   |   |   expresion ->
179      |   |   |   |   |
180      |   |   |   |   |   operando ->
181      |   |   |   |   |   |
182      |   |   |   |   |   |   a
183      |   |   |   |   |   |   |
184      |   |   |   |   |   |   |   operador ->
185      |   |   |   |   |   |   |   |
186      |   |   |   |   |   |   |   |   >
187      |   |   |   |   |   |   |   |   |
188      |   |   |   |   |   |   |   |   |   operando ->
189      |   |   |   |   |   |   |   |   |   |
190      |   |   |   |   |   |   |   |   |   |   b
191      |   |   |   |   |   |   |   |   |   |
192      |   |   |   |   |   |   |   |   |   |   )
193      |   |   |   |   |   |   |   |   |   |
194      |   |   |   |   |   |   |   |   |   |   sentenciacThen ->
195      |   |   |   |   |   |   |   |   |   |   |
196      |   |   |   |   |   |   |   |   |   |   |   then
197      |   |   |   |   |   |   |   |   |   |   |   |
198      |   |   |   |   |   |   |   |   |   |   |   |   (
199      |   |   |   |   |   |   |   |   |   |   |   |   |
200      |   |   |   |   |   |   |   |   |   |   |   |   |   instruccion ->
201      |   |   |   |   |   |   |   |   |   |   |   |   |   |
202      |   |   |   |   |   |   |   |   |   |   |   |   |   |   expresion ->
203      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
204      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   operando ->
205      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
206      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   a
207      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
208      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   operador ->
209      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
210      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   +
211      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
212      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   operando ->
213      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
214      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   b
215      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
216      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   )
217      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   sentenciaElse ->
218      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
219      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   else
220      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
221      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   instruccion ->
222      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
223      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   expresion ->
224      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
225      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   operando ->
226      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
227      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   a
228      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
229      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   operador ->
230      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
231      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   /
232      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
233      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   operando ->
234      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
235      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   b
236      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
237      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   ;

```

2. Segunda parte: iniciación a la tabla de símbolos

Este apartado sirve únicamente para hacer una introducción a la tabla de símbolos, que será usada más extensivamente en la última práctica. Consiste en añadir una funcionalidad al programa *linguine* que devuelva, tras analizar un fragmento de código, una lista con las variables que se han declarado en él y los valores que tienen asignadas.

LinguineSymbolsListener.java

```
public class LinguineSymbolsListener extends
    LinguineParserBaseListener {

    // Sobrescribe el método enterAssignment para procesar las
    // sentencias de asignación.
    @Override
    public void enterAssignment(LinguineParser.AssignmentContext
    ctx) {
        // Obtiene el texto de la sentencia de asignación y
        // elimina la palabra "let".
        String rawText = ctx.getText().replaceAll("let", "");
        try {
            // Divide el texto en dos partes: la variable y su
            // valor, usando el signo "=" como separador.
            String variable = rawText.split("=")[0].trim(); //
            // Utiliza trim() para eliminar espacios al principio y al final.
            String value = rawText.split("=")[1].trim();

            // Imprime la variable en azul y su valor en verde
            // en la consola.
            System.out.println("\u001B[34m" + variable
                + "\u001B[0m" + " -> " + "\u001B[32m"
                + value + "\u001B[0m");
        } catch (Exception e) {
            // Maneja la excepción si ocurre algún problema
            // durante el procesamiento.
            e.printStackTrace();
        }
    }
}
```

Vamos a explicar los métodos mas más destacados:

enterAssignment: Este método es llamado cuando el analizador sintáctico (parser) entra en una regla de asignación (Assignment).

ctx.getText().replaceAll("let", ""): Obtiene el texto de la sentencia de asignación y reemplaza la palabra "let" con una cadena vacía para quitarla.

try { ... } catch (Exception e) { ... }: Utiliza un bloque try-catch para manejar posibles excepciones que puedan ocurrir durante el procesamiento de la sentencia de asignación.

String variable = rawText.split("=")[0].trim();: Divide el texto en dos partes, la variable y su valor, utilizando el signo "=" como separador. El método trim() se utiliza para eliminar espacios en blanco al principio y al final de la variable.

System.out.println("\u001B[34m" + variable + "\u001B[0m" + " -> " + "\u001B[32m" + value + "\u001B[0m");: Imprime la variable en azul y su valor en verde en la consola, utilizando códigos de color ANSI para cambiar el color del texto en la consola.

e.printStackTrace();: Si ocurre una excepción durante el procesamiento, imprime la traza de la excepción para ayudar en la depuración.

Está diseñado para procesar sentencias de asignación, imprimir el nombre de la variable en azul y su valor en verde en la consola, y manejar posibles excepciones.