COMPILADORES PL2

Grupo 23

David Bachiller Vela

Víctor Sanavia Valdeolivas

Martín Mora

Contenido

[**INTRODUCCIÓN** 3](#_Toc150244942)

[**1.Primera parte: generación de árboles sintácticos para lenguajes específicos** 4](#_Toc150244943)

[**1.1** **SQLMini** 4](#_Toc150244944)

[**1.2** **Linguine** 7](#_Toc150244945)

[**2. Segunda parte: iniciación a la tabla de símbolos (1 punto)** 8](#_Toc150244946)

**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**

## **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]+)?;

STRING: '\'' ~'\''\* '\'';

COMA: ',';

AND: 'AND';

OR: 'OR';

ID: [a-zA-Z]+;

WS: [ \t\r\n]+ -> skip; // Ignoramos espacios en blanco

Aquí lo que hacemos es coger los símbolos terminales de las consultas (las palabras reservadas de SQL) y asignarlas a sus correspondientes tokens, a excepción de los tokens siguientes:

* **NÚMERO** 🡪 Aquí en número, recogemos tanto un numero entero como los decimales, ya que la parte decimal no tiene por qué aparecer, pero si aparece, tiene que haber al menos un decimal ya que no tiene sentido poner un punto decimal y dejarlo vacío.
* **STRING** 🡪 Aquí detectamos cualquier cadena de texto pero que este delimitada por comillas, aceptando cualquier carácter o ninguno entre ellas (negando las comillas de cierre)
* **ID** 🡪 Este token lo usamos para reconocer nombres de variables, ya que estamos aceptando la unión de uno o más caracteres tanto de minúsculas como de mayúsculas
* **WS** 🡪 Detectamos los espacios en blanco, de los cuales ignoramos usando *skip*.

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

parser grammar gSqlMiniParser;

options{

    tokenVocab = gSqlMiniLexer;

    language = Java;

}

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*;

Tenemos la siguiente estructura:

* **Options** 🡪 Aquí asignamos en *tokenVocab* nuestro archivo *lexer*, que es el descrito anteriormente (*gSqlMiniLexer*) y le ponemos que el lenguaje será en Java.
* **consultaSelect** 🡪 Aquí tenemos lo que contendría una consulta select, que obligatoriamente tendrá un **SELECT** seguido del **nombre de columnas**, después un **FROM** con un **nombre de tabla** y puede tener o no una sentencia WHERE o una ORDER BY
* **columnas** 🡪 Obligatoriamente tiene que haber un nombre de columna (ID) y es posible que pueda haber más separados por comas (es posible seleccionar varias columnas en un SELECT)
* **sentenciaWhere** 🡪 Un WHERE debe tener obligatoriamente expresión a evaluar, la cual puede ser anidada, es decir, puedes tener varias expresiones dentro de otra
* **sentenciaOrderBy** 🡪 un ORDER BY debe tener obligatoriamente un ORDER seguido de un BY, un nombre de columna y un método de ordenación que puede ser ASC o DESC

## **1.2** **Linguine**

**2. Segunda parte: iniciación a la tabla de símbolos (1 punto)**