



**UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS**

Analizador Léxico, Sintáctico y Semántico empleando JCup y JFlex.

Daniela Alexandra Martínez - 20171020006

Daniel Felipe Roa Palacios - 20171020077

Mateo Yate Gonzalez - 20171020087

*Universidad Distrital Francisco José de Caldas,
Facultad de Ingeniería.*

Ciencias de la Computación III

Profesor: Christian Jhonathan Acosta Cipagauta.

24/08/2020

Contenido

Algunas definiciones básicas	3
Descripción del proyecto y generalidades	4
Analizador Léxico	5
Lexer.flex	6
Lexer.java	7
Tokens.java	8
Analizador Sintactico	9
Intento con la librería JCup y preliminares del analizador semántico.	10
LexerCup.flex	11
LexerCup.java	12
Syntax.cup	12
Syntax.java y sym.java.	13
Problemas al momento de la implementación	13
Analizador sintáctico de autoría propia	13
Analizador Semántico	16
Bibliografía	17

Algunas definiciones básicas

Compilador : Un compilador es un software o programa que tiene como objetivo convertir un código fuente de entrada en otro código de salida como puede ser el código máquina. [1] Entre sus funciones principales está el proceso de traducción. Podemos poner de ejemplo GCC (GNU Compiler Collection) como el compilador del lenguaje C y C++ (entre otros) a código máquina. [2]

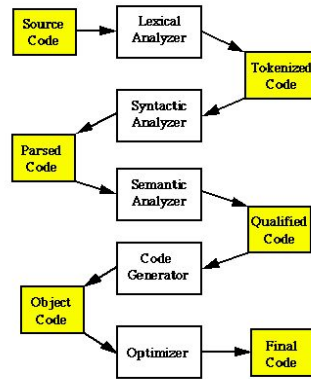


Figura 1.1, Descripción de compilador optimizado

Token: Es una cadena de caracteres que tiene un significado coherente en cierto lenguaje de programación. Son los elementos más básicos sobre los cuales se desarrolla toda traducción de un programa, surgen en la primera fase, llamada análisis léxico, sin embargo se siguen utilizando en las siguientes fases (análisis sintáctico y análisis semántico) antes de perderse en la fase de síntesis. En un lenguaje de programación, los tokens son el equivalente a las palabras y signos de puntuación en el lenguaje natural escrito. Los tokens están separados por elementos de separación que reciben el nombre genérico de separadores. [3]

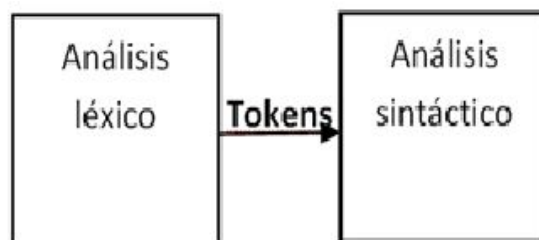


Figura 1.2, Ejemplo uso de tokens en el proceso del compilador

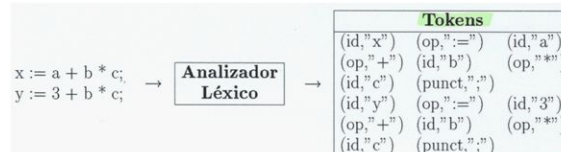


Figura 1.3, Ejemplo de cómo usar los tokens en un compilador personalizado

Descripción del proyecto y generalidades

El presente proyecto tuvo como objetivo la construcción de las fases más importantes de un compilador, el cual fue la pieza fundamental y el proyecto en general de la asignatura. Así, se decidió emplear el lenguaje de programación Java junto con las librerías JFlex y JCup para realizar las distintas fases relacionadas con los tokens y el lenguaje seleccionado.

Para este último punto, es hecho de destacar que se buscó emular un compilador que leyese sentencias de SQL en una manera más general (sin tener en cuenta algún motor de bases de datos en específico).

En este documento se exponen los aspectos más importantes relacionados con cada etapa del “semi compilador” que se realizó, empezando en la primera etapa de análisis léxico, luego la fase análisis sintáctico y finalmente en análisis semántico desde una perspectiva teórico práctica, lo que quiere decir que se abordarán aspectos dentro del código de la aplicación que permitan evidenciar cómo se aplicaron los conceptos estudiados a lo largo del presente curso.

Es importante resaltar que, para realizar la unión de las tres etapas del compilador, se hizo necesario emplear una interfaz gráfica desarrollada en Java con una serie de paneles como se identifica a continuación:

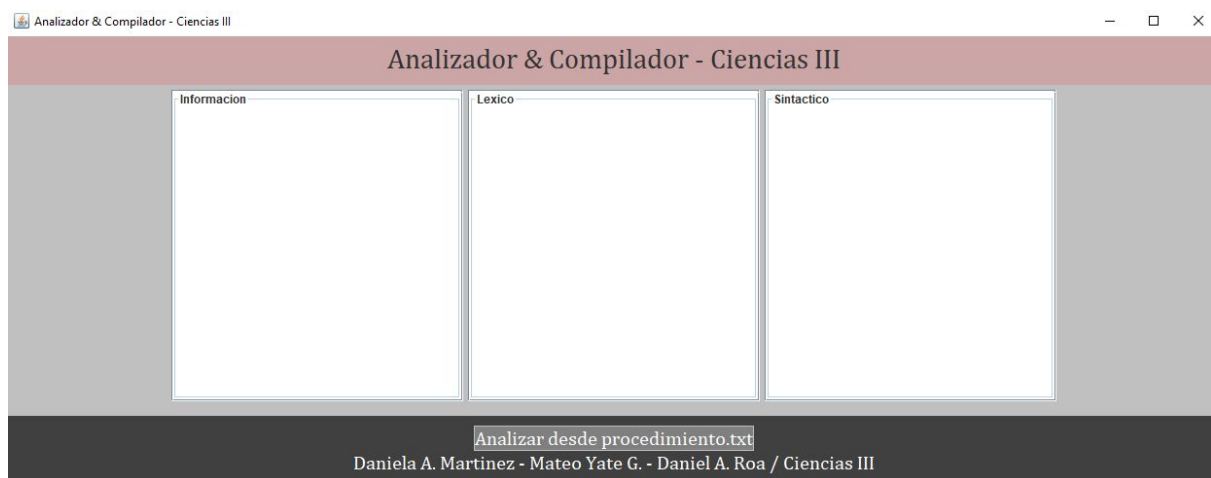


Figura 1. Interfaz gráfica desarrollada.

Entonces, en estos paneles se imprimió cada una de las etapas del proyecto más los datos leídos por el archivo procedimiento.txt, de modo que los tres analizadores se lanzaban al presionar el botón correspondiente para cargar el archivo de texto.

Analizador Léxico

El analizador léxico es la primera etapa de nuestro compilador. Es posible definir este componente como aquella herramienta que emplea el compilador para ayudar a leer los caracteres de entrada, para con ello formar una especie de “componentes” y así poder realizar su identificación y enviar dicha información al analizador sintáctico.

Como complemento a la parte teórica, esta fase es aquella que lee la secuencia del programa fuente por caracteres en el sentido de izquierda a derecha, remueve los espacios en blanco, tabulaciones y saltos en línea además de los comentarios, y finalmente es quien se encarga de agrupar estos caracteres en las unidades llamadas tokens (secuencias de caracteres que juntos forman una unidad significativa). [4]

Además, para esta fase es importante destacar que algunos elementos como los separadores, los paréntesis, las operaciones numéricas y demás son finalmente clasificaciones de tokens (llamados también componentes léxicos en nuestro idioma).

Entrando en el aspecto práctico, para nuestro analizador léxico se empleó la librería JFlex, la cual se define como un generador de analizador léxico para Java escrito en este mismo lenguaje. Además, los lexemas generados por JFlex están basados en un autómata finito determinista, lo cual cuenta con una serie de ventajas como que son rápidos sin tener un retroceso significativo. [5]

En este caso, para el proyecto se contaron con tres clases relacionadas a este componente: `Lexer.flex`, `Lexer.java` y `Tokens.java`.

Lexer.flex

Es la clase que permite reconocer expresiones como palabras o tipos de datos (admite expresiones regulares) como un token, para así poder clasificarlo y mostrarle al usuario a que se refiere cada palabra escrita.

```
package lexico;
import static lexico.Tokens.*;
%%
%class Lexer
%type Tokens
L=[a-zA-Z_]+
D=[0-9]+
espacio=[\t,\r,\n]+
%{
    public String lecturaLexica;
}%
%%
if | else | while {lecturaLexica=yytext(); return Reservada;}
{espacio} {/^Ignore*/}
"/".* {/^Ignore*/}

/* Statements */
"INSERT" {lecturaLexica=yytext();return Statement;}
"SELECT" {lecturaLexica=yytext();return Statement;}
"DELETE" {lecturaLexica=yytext();return Statement;}
"UPDATE" {lecturaLexica=yytext();return Statement;}
"CREATE" {lecturaLexica=yytext();return Statement;}

/* Tipos de datos (extraido de: https://sites.google.com/site/basededatosrelacionales/home/contenido/tipos-de-datos-en-sql-server) */
"BigInt" {lecturaLexica=yytext();return Tipo;}
"bigint" {lecturaLexica=yytext();return Tipo;}
"varchar" {lecturaLexica=yytext();return Tipo;}
"nvarchar" {lecturaLexica=yytext();return Tipo;}
"int" {lecturaLexica=yytext();return Tipo;}
"smallint" {lecturaLexica=yytext();return Tipo;}
"tinyint" {lecturaLexica=yytext();return Tipo;}
"bit" {lecturaLexica=yytext();return Tipo;}
"decimal" {lecturaLexica=yytext();return Tipo;}
"numeric" {lecturaLexica=yytext();return Tipo;}
"money" {lecturaLexica=yytext();return Tipo;}
"smallmoney" {lecturaLexica=yytext();return Tipo;}
"float" {lecturaLexica=yytext();return Tipo;}
"real" {lecturaLexica=yytext();return Tipo;}
"datetime" {lecturaLexica=yytext();return Tipo;}
"smalldatetime" {lecturaLexica=yytext();return Tipo;}
"char" {lecturaLexica=yytext();return Tipo;}
"text" {lecturaLexica=yytext();return Tipo;}
"nchar" {lecturaLexica=yytext();return Tipo;}
"ntext" {lecturaLexica=yytext();return Tipo;}
"binary" {lecturaLexica=yytext();return Tipo;}
"varbinary" {lecturaLexica=yytext();return Tipo;}
"image" {lecturaLexica=yytext();return Tipo;}
"cursor" {lecturaLexica=yytext();return Tipo;}
"timestamp" {lecturaLexica=yytext();return Tipo;}
```

Figura 2. Ilustración de una parte de la clase Lexer.flex empleada en el presente proyecto.

Lexer.java

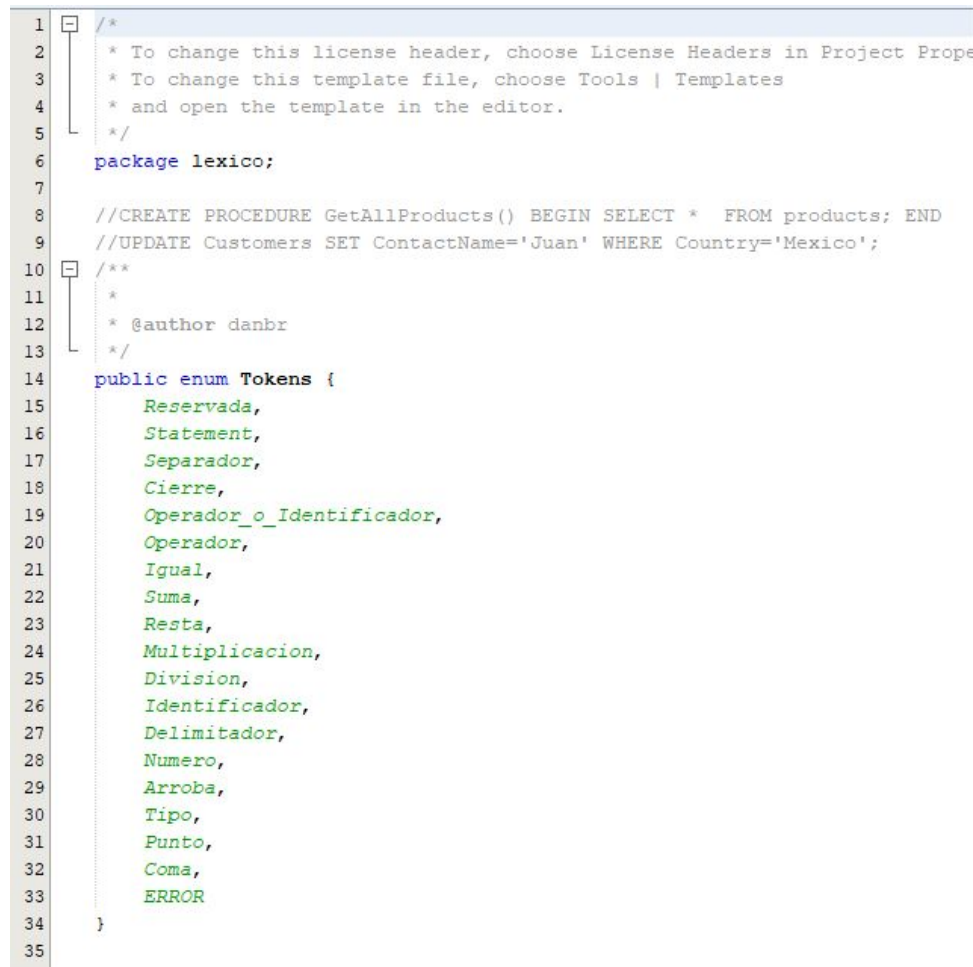
Esta clase es generada por la librería JFlex y es el puente entre el archivo .flex que contiene las instrucciones para el análisis sintáctico, el archivo Tokens.java que contiene los tokens permitidos, y el programa desarrollado comúnmente.

```
736     }
737
738     // store back cached position
739     zzMarkedPos = zzMarkedPosL;
740
741     switch (zzAction < 0 ? zzAction : ZZ_ACTION[zzAction]) {
742     case 13:
743         { lecturaLexica=yytext();return Reservada;
744         }
745     case 15: break;
746     case 8:
747         { lecturaLexica=yytext();return Punto;
748         }
749     case 16: break;
750     case 10:
751         { lecturaLexica=yytext();return Arroba;
752         }
753     case 17: break;
754     case 6:
755         { lecturaLexica=yytext();return Operador;
756         }
757     case 18: break;
758     case 1:
759         { return ERROR;
760         }
761     case 19: break;
762     case 11:
763         { lecturaLexica=yytext(); return Reservada;
764         }
765     case 20: break;
766     case 5:
767         { lecturaLexica=yytext();return Delimitador;
768         }
769     case 21: break;
770     case 14:
771         { lecturaLexica=yytext();return Statement;
772         }
773     case 22: break;
774     case 2:
775         { lecturaLexica=yytext(); return Identificador;
776         }
777     case 23: break;
778     case 9:
779         { lecturaLexica=yytext();return Cierre;
780         }
```

Figura 3. Ilustración de una parte de la clase Lexer.java empleada en el presente proyecto.

Tokens.java

Esta clase simplemente define los tokens que se emplearán y serán válidos para la lectura por parte del analizador léxico.



```
1  /*
2  * To change this license header, choose License Headers in Project Properties
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6  package lexico;
7
8  //CREATE PROCEDURE GetAllProducts() BEGIN SELECT * FROM products; END
9  //UPDATE Customers SET ContactName='Juan' WHERE Country='Mexico';
10 /**
11  *
12  * @author danbr
13  */
14 public enum Tokens {
15     Reservada,
16     Statement,
17     Separador,
18     Cierre,
19     Operador_o_Identificador,
20     Operador,
21     Igual,
22     Suma,
23     Resta,
24     Multiplicacion,
25     Division,
26     Identificador,
27     Delimitador,
28     Numero,
29     Arroba,
30     Tipo,
31     Punto,
32     Coma,
33     ERROR
34 }
35
```

Figura 4. Ilustración de una parte de la clase Lexer.java empleada en el presente proyecto.

Así, la unión de estas clases estuvo dada por la creación inicialmente del analizador léxico cuando se iniciaba el programa con la invocación del método `crearAnalziadorLexico()`, que lo que realizaba simplemente era generar el lexer a partir de la librería.

Con ello en mente, al presionar el botón de “analizar desde procedimiento.txt”, lo que se hacía respecto al analizador léxico era, cargar el archivo, general un objeto Lexer a partir de dicho archivo, y recorrer todas las filas y palabras para así poder identificar en una estructura switch-case sobre qué tipo de token se estaba tratando, o por lo contrario si la palabra correspondiente no pertenecía a ningún token.

Luego de esto, se imprimían los datos con el tipo de cada palabra o la incidencia registrada en el panel correspondiente (segundo panel de la figura 1 de izquierda a derecha) y se procedía a avanzar a las siguientes etapas del analizador.

Analizador Sintactico

Esta fase es la que se encarga de agrupar los tokens generados por el analizador léxico en frases gramaticales que el compilador utilizará en etapas posteriores. Además determina si la secuencia de tokens descrita anteriormente sigue la sintaxis definida del lenguaje.

Posteriormente, obtiene la estructura jerárquica del programa en forma de árbol (llamado árbol de Parsing) para el cual los vértices son las construcciones de alto nivel del lenguaje. Es de destacar que el objetivo de este árbol finalmente es demostrar cómo la secuencia de tokens de entrada puede ser derivada a partir de las reglas gramaticales definidas. [4, p. 34]

En este paso, se determinan las relaciones estructurales que existen entre los tokens, y a modo de símil es posible ver estas relaciones como en análisis gramatical que se puede realizar en una frase escrita en lenguaje natural.

Finalmente, con el conjunto de reglas “gramaticales” definidas en el lenguaje y con base en el árbol descrito anteriormente, el analizador sintáctico brinda a la siguiente fase la lista de reglas del lenguaje que se encuentran dadas por los tokens y el orden generado por el árbol de parsing.

Como aspecto práctico, esta etapa constó de dos subdivisiones, la primera fue la aplicación realizada empleando la librería JCup, la cual puede definirse como un “parser-generador”, lo que quiere decir que este analizador construye un parser con código de producción y asociación de fragmentos de, en este caso, código SQL. Luego, su funcionamiento es muy sencillo, cuando una producción en particular es reconocida, se genera la clase Syntax.java con un método Symbol parser() perteneciente a la librería misma que realiza las acciones del analizador sintáctico. [6]

De igual manera, la segunda etapa y que fue la que se implementó finalmente en el presente proyecto fue un analizador sintáctico realizado de una manera más manual, esto quiere decir que no se emplearon librerías y se realizaron validaciones de una manera más rudimentaria empleando expresiones de condicionales anidadas que permitieran leer toda una línea del código SQL en cuestión, como se expondrá más adelante.

Intento con la librería JCup y preliminares del analizador semántico.

Como se describió anteriormente, se realizó un intento de analizador sintáctico empleando la librería JCup y JFlex con resultados negativos. Inicialmente, se expone la estructura básica de un analizador sintáctico en JCup y JFlex y las razones del porqué esta implementación no fue posible.

Para empezar, este analizador se compone de 5 clases: LexerCup.flex, LexerCup.java, Sintax.cup, Sintax.java y sym.java.

De igual manera, es de destacar que la conexión entre el analizador sintáctico y la interfaz del usuario y lógica del programa estaban dadas por un método que generaba las clases correspondientes por medio de la librería JCup y JFlex, para así posteriormente solicitar a un objeto de tipo Sintax que realizará el parser para una línea del procedimiento almacenado en SQL, siguiendo el ciclo for expuesto a continuación que al final pintaba los resultados obtenidos en el tercer panel de la figura 1 de izquierda a derecha:

```
//Analizador sintactico
for(int i=0; i < procedimientoAlmacenado.size(); i++){
    System.out.println("Linea por analizar: " + procedimientoAlmacenado.get(i) + ", numero de linea: " + i);
    Sintax s = new Sintax(new lexico.LexerCup(new StringReader(procedimientoAlmacenado.get(i))));
    analizar(s,i);
}

taSintactico.setText(resultadoSintactico);
```

Figura 5. Conexión entre la lógica e interfaz del programa y el analizador sintáctico.

LexerCup.flex

Esta clase es de suma importancia en este intento de analizador sintáctico, dado que guarda los tipos de símbolos que puede leer nuestro analizador tales como datos, identificadores, o algunos elementos específicos como lo pueden ser las comas o los paréntesis de apertura y cierre. Para ello, emplea expresiones regulares u palabras y caracteres textuales que apunten a un tipo de símbolo que las identifiquen.

```
/* Palabra reservada FROM */
( FROM ) {return new Symbol(sym.FROM, yychar, yyline, yytext());}

/* Palabra reservada WHERE */
( WHERE ) {return new Symbol(sym.WHERE, yychar, yyline, yytext());}

/* Palabra reservada SET */
( SET ) {return new Symbol(sym.SET, yychar, yyline, yytext());}

/* Palabra reservada TABLE */
( TABLE ) {return new Symbol(sym.TABLE, yychar, yyline, yytext());}

/* Operador Igual */
( "=" ) {return new Symbol(sym.Igual, yychar, yyline, yytext());}

/* Simbolo asterisco */
( "*" ) {return new Symbol(sym.Asterisco, yychar, yyline, yytext());}

/* Parentesis de apertura */
( "(" ) {return new Symbol(sym.Parentesis_a, yychar, yyline, yytext());}

/* Parentesis de cierre */
( ")" ) {return new Symbol(sym.Parentesis_c, yychar, yyline, yytext());}

/* Coma */
( "," ) {return new Symbol(sym.Coma, yychar, yyline, yytext());}

/* Arroba */
( "@" ) {return new Symbol(sym.Arroba, yychar, yyline, yytext());}

/* Punto y coma */
( ";" ) {return new Symbol(sym.P_coma, yychar, yyline, yytext());}

/* Identificador */
{L}{L|D}* {return new Symbol(sym.Identificador, yychar, yyline, yytext());}

/* Datos */
[{l}+([']*[\w]+[']*[,*]+[^,]*)+ {return new Symbol(sym.Datos, yychar, yyline, yytext());}
```

Figura 6. Ilustración de una parte de la clase LexerCup.flex empleada en el presente proyecto.

LexerCup.java

Como se describió para la librería JFlex en el analizador léxico, esta clase se encarga de ser el puente entre el archivo .flex y la interfaz gráfica. Es de destacar que en la implementación de un analizador sintáctico no se emplea una clase de tokens como sí ocurre en el analizador léxico, esto dado que las palabras y expresiones no se caracterizan por tokens sino por símbolos.

Sintax.cup

Esta clase es la más importante del analizador sintáctico, allí se definen las estructuras que son admitidas por el lenguaje en cuestión (en este caso, SQL) por medio de los símbolos definidos en la clase LexerCup.flex.

Para ello, se definen unas expresiones terminales y no terminales. Las expresiones terminales se refieren a expresiones “irreducibles”, que pueden ser tomadas como una unidad, por ejemplo un tipo de dato o un paréntesis de apertura son caracteres y expresiones que no pueden ser escritas en términos de otras.

En sentido contrario, las expresiones no terminales son aquellas que se componen de otras expresiones definidas en el LexerCup.flex. Es aquí donde ocurre la parte más importante y es que con los nodos no terminales es posible definir la sintaxis de un lenguaje (como SQL) en cuanto a sus expresiones compuestas por expresiones terminales, como se ilustra en la siguiente imagen:

```
terminal INSERT, DELETE, UPDATE, SELECT, CREATE, PROCEDURE, BEGIN, END, INTO, VALUES,
    FROM, WHERE, SET, TABLE, Identificador, Parentesis_a, Parentesis_c, P_coma,
    Coma, Int, Cadena, Numero, Datos, T_dato, Igual, Asterisco, Comillas, CamposTabla,
    Function, Arroba, ERROR;
non terminal SENTENCIA;

start with SENTENCIA;

SENTENCIA ::=

    CREATE PROCEDURE Identificador Arroba Identificador T_dato Parentesis_a Numero Parentesis_c |
    BEGIN |
    INSERT INTO Identificador CamposTabla VALUES Datos P_coma |
    DELETE FROM Identificador WHERE Identificador Igual Datos P_coma |
    DELETE FROM Identificador P_coma |
    UPDATE Identificador SET Identificador Igual Datos WHERE Identificador Igual Datos P_coma |
    UPDATE Identificador SET Identificador Igual Datos P_coma |
    SELECT Identificador FROM Identificador P_coma |
    SELECT Asterisco FROM Identificador P_coma |
    CREATE TABLE Identificador Parentesis_a Identificador T_dato Parentesis_c P_coma |
    END

;
```

Figura 7. Ilustración de una parte de la clase Sintax.cup empleada en el presente proyecto.

De igual manera, esta clase es muy importante en el sentido en que se emplea en la etapa de análisis semántico como se verá más adelante en la presente documentación.

Syntax.java y sym.java.

Finalmente, estas clases son generadas por la librería JCup y son una analogía a la clase Lexer.java, lo que quiere decir que son el puente para unir la interfaz gráfica del programa y toda la lógica con el analizador sintáctico desarrollado en JCup.

De una manera más específica, la clase sym.java guarda una serie de identificadores para los símbolos definidos por el JFlex en la clase LexerCup.flex.

Problemas al momento de la implementación

Ahora bien, como se mencionó anteriormente, la implementación de la librería JCup con JFlex no fue posible para el analizador sintáctico, esto puesto que al momento de definir la sintaxis de las expresiones SQL en la clase Syntax.cup, no se encontró una referencia clara de qué estructura deben seguir las expresiones regulares. Sin embargo, al intentar implementar este analizador sintáctico realizando validaciones más manuales, el método que realizaba el parser en JCup no detectaba correctamente las expresiones regulares en las cadenas de texto, por lo cual se optó por volver a la idea original del analizador sintáctico de autoría propia desarrollado para entregas pasadas.

Analizador sintáctico de autoría propia

Para el desarrollo del analizador sintáctico empleado en el proyecto final, se realizaban distintas comprobaciones que verificaran que se cumpliera la sintaxis de las expresiones que permitía leer nuestro compilador (INSERT, DELETE, UPDATE, SELECT, CREATE, CREATE PROCEDURE, BEGIN, END). Así, se iniciaba por un switch para determinar qué estructura debía seguirse dependiendo del primer “statement” que presentase la línea a analizar en cuestión, como se ilustra a continuación en el fragmento de código correspondiente a la clase Sintactico.java

```
switch (opc) {
    case "INSERT":
        insert(cadenaSeparada);
        break;
    case "DELETE":
        delete(cadenaSeparada);
        break;
    case "UPDATE":
        update(cadenaSeparada);
        break;
    case "SELECT":
        select(cadenaSeparada);
        break;
    case "CREATE":
        create(cadenaSeparada);
        break;
    case "CREATE PROCEDURE":
        createProcedure(cadenaSeparada);
        break;
    case "BEGIN":
        JOptionPane.showMessageDialog(null, lineaActual + "Expresión válida");
        break;
    case "END":
        JOptionPane.showMessageDialog(null, lineaActual + "Expresión válida");
        break;
    default:
        JOptionPane.showMessageDialog(null, lineaActual + "Expresión errada: La primera palabra no es un statement");
}
```

Figura 8. Switch para determinar la estructura que debía seguir la sentencia en cuestión.

A continuación, se ilustra a manera de ejemplo la validación de una sentencia UPDATE, y los métodos relacionados como el de verificar el nombre de la expresión:

```
public boolean verificarNombre(String tbName) {

    //Verifica que el nombre no contenga palabras reservadas
    Pattern pat = Pattern.compile("UPDATE|CREATE|SELECT|INSERT|DELETE|"
        + "\\(\\|\\|\\|\\|\\|\\|\\|\\|\\|\\|\\|=|&|#|>|<|^'|'\\|\\|\\|\\|\\|");
    //Pattern pat = Pattern.compile("[^\\d]{0,}.[\"INSERT,CREATE,DELETE,if,else,for\"];");
    Matcher mat = pat.matcher(tbName);

    //Esta bien
    if (!mat.find()) {
        System.out.println(lineaActual + tbName + ": No contiene palabras reservadas");
        pat = Pattern.compile("^([\\d]*.*)");
        mat = pat.matcher(tbName);
        if (mat.matches()) {
            System.out.println(lineaActual + tbName + ": No empieza por un dígito");
            return true;
        } else {
            System.out.println(lineaActual + tbName + ": Empieza por un dígito");
            return false;
        }
    } else {
        System.out.println(lineaActual + "ERROR: " + tbName + " contiene palabras reservadas o contiene caracteres especiales");
        return false;
    }
}
```

Figura 9. Validación de una expresión UPDATE y de que el nombre de la tabla esté correctamente escrito (verificarNombre())

Es de destacar por último que la lectura se realizaba por líneas para conocer sobre la expresión en cuestión, además de que, como se evidencia en la figura 9, se emplearon algunas expresiones regulares empleando objetos de tipo Pattern para verificar que se cumpliesen las condiciones expresadas en la misma expresión.

Analizador Semántico

Finalmente, para la última etapa de nuestro compilador se realizó la implementación de un analizador semántico. Es de destacar que, a pesar de los problemas presentados en el analizador sintáctico, esta etapa fue la más complicada en cuanto a implementación puesto que es muy teórica y para ello hubo que retomar la idea del JCup del analizador sintáctico, puesto que para desarrollar correctamente el analizador semántico, era muy difícil realizar la adaptación de nuestro analizador a una sintaxis que permitiese definir las estructuras necesarias en el analizador semántico.

Respecto al componente teórico, es posible definir esta fase como aquel componente que se encarga de realizar las comprobaciones necesarias sobre el árbol sintáctico (el cual es una representación comprimida del árbol de parsing) para poder determinar cual es el correcto significado del programa. [4, p. 44].

Además, revisa las reglas que no pueden ser capturadas por ese conjunto de reglas gramaticales definido anteriormente, pero que pueden ser finalmente verificadas en tiempos de compilación. Es de aclarar que estas reglas hacen correspondencia a la semántica estática propia del lenguaje, donde cabe recalcar que es necesario hacer uso de la tabla de símbolos.

De igual manera, el análisis semántico puede agregar algunos atributos (como los tipos de datos) a la estructura del árbol semántico.

Así, es de destacar que nuestro árbol sintáctico estaría dado por las expresiones definidas en el archivo Syntax.cup. Dicho en otras palabras, la idea fue generar un árbol cuyos nodos fuesen esas partes de cada sentencia (o expresión).

De igual manera, a continuación se expone la clase .cup empleada para el analizador semántico desarrollado en el marco del proyecto:

```
terminal INSERT, DELETE, UPDATE, SELECT, CREATE, CREATE_PROCEDURE, BEGIN, END, INTO, VALUES,
FROM, WHERE, SET, TABLE, Identificador, Parentesis_a, Parentesis_c, P_coma,
Numero, Datos, T_dato, Igual, Asterisco, Comillas, function, Id_procedure, ERROR;

non terminal QUERY, SENTENCIA, CLAUSULA_INTRO, CLAUSULA_VALUES,
CLAUSULA_WHERE, CLAUSULA_FROM, CONDICIONES, TABLA, VALOR, TABLA_NODO, NOMBRE_TABLA, COLS, TBL;

start with SENTENCIA;
//pendiente clausula into clausual where clausula vales valor tablanodo
QUERY ::= SENTENCIA: n1 {
    Nodo nd = new Nodo();
    nd.setEtiqueta("SENTENCIA");
    nd.setId(parser.cont);
    parser.cont++;
    nd.AddHijos((Nodo) n1);
    parser.padre = (Nodo) nd;
    RESULT = nd;
};

SENTENCIA ::= INSERT CLAUSULA_INTRO:n1 Parentesis_a VALOR:n2 Parentesis_c CLAUSULA_VALUES:n3 P_coma {
    Nodo nd = new Nodo();
    nd.setEtiqueta("INSERT");
    nd.setId(parser.cont);
    parser.cont++;
    nd.AddHijos((Nodo) n1);
    nd.AddHijos((Nodo) n2);
    nd.AddHijos((Nodo) n3);
    parser.padre = (Nodo) nd;
    RESULT = nd;
};

SENTENCIA ::= INSERT CLAUSULA_INTRO:n1 CLAUSULA_VALUES:n2 P_coma {
    Nodo nd = new Nodo();
    nd.setEtiqueta("INSERT");
    nd.setId(parser.cont);
    parser.cont++;
    nd.AddHijos((Nodo) n1);
    nd.AddHijos((Nodo) n2);
    nd.AddHijos((Nodo) n3);
    parser.padre = (Nodo) nd;
    RESULT = nd;
};
```

Figura 10. Clase .cup empleada para el analizador semántico.

Bibliografía

- [1] ¿Qué es un compilador? (2019). Recuperado de <https://programar.best/programacion/que-es-un-compilador/>
- [2] Herramientas para generar compiladores (2012). Recuperado de www.compiladores.com.
- [3] Token . Recuperado de <https://www.ecured.cu/Tokens>
- [4] ESTRUCTURA GENERAL DE UN COMPILADOR. p. 15. Recuperado de http://www.exa.unicen.edu.ar/catedras/compila1/index_archivos/Introduccion.pdf
- [5] JFlex User's Manual (NN). Recuperado de: <https://www.jflex.de/manual.html#Intro>
- [6] CUP User's Manual (NN). Recuperado de: <http://www2.cs.tum.edu/projects/cup/docs.php#intro>