MANUAL TECNICO
ANALIZADOR LEXICO Y SINTACTICO LENGUAJE SQL

DESCRIPCIÓN

El proyecto se centra en el desarrollo de un analizador léxico y un analizador sintáctico, utilizando herramientas automatizadas para el procesamiento del lenguaje en Java. Para el análisis léxico se empleó **JFlex**, una herramienta generadora de analizadores léxicos que permite definir y generar un escáner que identifica y categoriza los elementos básicos (tokens) en un lenguaje de programación.

El objetivo del analizador léxico es leer el código de entrada y dividirlo en tokens, como palabras clave, identificadores, operadores, literales y otros símbolos fundamentales en el lenguaje. Una vez identificados estos tokens, el **analizador sintáctico** se encarga de estructurarlos según las reglas gramaticales predefinidas, comprobando que la secuencia y relación entre los tokens formen construcciones válidas en el lenguaje.

Tecnologías usadas

1. Backend: Lógica del programa desarrollada en Java 21,Jflex.

2. Frontend: Interfaz gráfica implementada con Java Swing.

3. Generación de Gráficas: Uso de Dot - GraphViz versión 2.43.0.

4. Distribución: Archivo ejecutable empaquetado en formato JAR.

5. Documentación: Manuales generados utilizando LibreOffice.

6. Repositorio en Github: trabajado en repositorio remoto y local, cuenta con 9 ramas de funcionalidades y una rama de actualizaciones en código.

7. Nota: No se utilizaron expresiones regulares como Regex o Matcher en el desarrollo.

8. Nota: Los automatas que se describen acontinuacion son de tipo finito Determinista (AFD).

9, Nota: No se usaron herramientas para el analizador sintactico

ANALIZADOR LEXICO EN JFLEX

El analizador léxico desarrollado en JFlex tiene como objetivo identificar y clasificar los distintos componentes del lenguaje, conocidos como tokens, que servirán como base para el análisis sintáctico. A continuación, se explican los elementos principales del archivo Jflex.

```
/* Definición de la clase del analizador léxico */
%public
%class AnalizadorLexico
%unicode
%line
%column
%caseless
```

Definición de Patrones Regulares

En esta sección, se definen patrones comunes para simplificar las expresiones regulares. Estos patrones se referencian en las reglas de tokens, facilitando la legibilidad y el mantenimiento del código.

```
/* Definición de patrones regulares */
   Digito = [0-9]
   FECHA = "'"{Digito}{Digito}{Digito}{Digito}"-"{Digito}{Digito}"-"{Digito}{Digito}"'"
   DECIMAL = "-?" {Digito}+ "(\\." {Digito}+ ")?"
   ERROR_DECIMAL = "-?({Digito}*\\.){2,}{Digito}*"
   NUMERO = "-"?[0-9]+
   Letra = [a-zA-Z]
   LetterLower = [a-z]
   Identificador = {LetterLower}({LetterLower}|{Digito}|_{LetterLower}|_{Digito})*
   CADENA = "'"( [^'] | "''" )*"'" //cadenas de texto
   PALABRA = [a-zA-Z]+
   SPACE = " "
%%
```

Digito: Representa cualquier dígito numérico ([0-9]).

- **FECHA**: Reconoce una cadena con formato de fecha AAAA-MM-DD.
- **DECIMAL**: Representa números decimales con o sin signo negativo.
- **ERROR_DECIMAL**: Identifica errores en decimales, como tener más de un punto decimal.
- **NUMERO**: Reconoce números enteros con o sin signo negativo.
- **Identificador**: Define los identificadores válidos, que pueden comenzar con una letra minúscula seguida de letras, dígitos o guiones bajos.
- **CADENA**: Permite identificar cadenas de texto entre comillas simples (')

Reglas de Tokens

Esta sección define cómo el analizador debe actuar cuando encuentra un patrón. Cada regla retorna un Token de un tipo específico, indicando el tipo de dato y su posición.

```
%%

/* Definición de reglas de tokens */
/* Entero , decimal */

 {FECHA}          { return new Token(TipoToken.FECHA, yyline, yycolumn, yytext()); }
  {ERROR_DECIMAL}  { return new Token(TipoToken.NO_RECONOCIDO, yyline, yycolumn, yytext()); }
 {DECIMAL}        { return new Token(TipoToken.DECIMAL, yyline, yycolumn, yytext()); }
 {NUMERO}         { return new Token(TipoToken.ENTERO,  yyline, yycolumn, yytext()); }
 {CADENA}          { return new Token(TipoToken.CADENA,  yyline, yycolumn, yytext()); }

/* LLAVE ESPECIAL */

"PRIMARY KEY"  { return new Token(TipoToken.LlaveEspecial,  yyline, yycolumn, yytext()); }
"NOT NULL" { return new Token(TipoToken.LlaveEspecial,  yyline, yycolumn, yytext()); }
"UNIQUE" { return new Token(TipoToken.LlaveEspecial,  yyline, yycolumn, yytext()); }
"AUTO INCREMENT" { return new Token(TipoToken.LlaveEspecial,  yyline, yycolumn, yytext()); }

"PRIMARY_KEY"  { return new Token(TipoToken.LlaveEspecial,  yyline, yycolumn, yytext()); }
"NOT_NULL" { return new Token(TipoToken.LlaveEspecial,  yyline, yycolumn, yytext()); }
"UNIQUE" { return new Token(TipoToken.LlaveEspecial,  yyline, yycolumn, yytext()); }
"AUTO_INCREMENT" { return new Token(TipoToken.LlaveEspecial,  yyline, yycolumn, yytext()); }
```

```
/*    TOKENS CREATE */
"CREATE"          { return new Token(TipoToken.CREATE, yyline, yycolumn,yytext()); }
"DATABASE"        { return new Token(TipoToken.CREATE, yyline, yycolumn,yytext()); }
"TABLE"           { return new Token(TipoToken.CREATE, yyline, yycolumn,yytext()); }
"KEY"             { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"NULL"            { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"PRIMARY"         { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"UNIQUE"          { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"FOREIGN"         { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"REFERENCES"      { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"ALTER"           { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"ADD"             { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"COLUMN"          { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"TYPE"            { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"DROP"            { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"CONSTRAINT"      { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"IF"              { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"EXIST"           { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"CASCADE"         { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"ON"              { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
"DELETE"          { return new Token(TipoToken.CREATE, yyline, yycolumn,yytext()); }
"SET"             { return new Token(TipoToken.CREATE, yyline, yycolumn,yytext()); }
"UPDATE"          { return new Token(TipoToken.CREATE, yyline, yycolumn,yytext()); }
"INSERT"          { return new Token(TipoToken.CREATE, yyline, yycolumn,yytext()); }
```

```
  "INTO"            { return new Token(TipoToken.CREATE, yyline, yycolumn,yytext()); }
  "VALUES"          { return new Token(TipoToken.CREATE, yyline, yycolumn,yytext()); }
  "SELECT"          { return new Token(TipoToken.CREATE, yyline, yycolumn,yytext()); }
  "FROM"            { return new Token(TipoToken.CREATE, yyline, yycolumn,yytext()); }
  "WHERE"           { return new Token(TipoToken.CREATE, yyline, yycolumn,yytext()); }
  "AS"              { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
  "GROUP"           { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
  "ORDER"           { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
  "BY"              { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
  "ASC"             { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
  "DESC"            { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
  "LIMIT"           { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
  "JOIN"            { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
  "DEFAULT"         { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
  "EXISTS"          { return new Token(TipoToken.CREATE, yyline, yycolumn, yytext()); }
  /* Tipos de datos */
  "INTEGER"         { return new Token(TipoToken.TIPO_DE_DATO, yyline, yycolumn, yytext()); }
  "BIGINT"          { return new Token(TipoToken.TIPO_DE_DATO, yyline, yycolumn, yytext()); }
  "VARCHAR"         { return new Token(TipoToken.TIPO_DE_DATO, yyline, yycolumn, yytext()); }
  "DECIMAL"         { return new Token(TipoToken.TIPO_DE_DATO, yyline, yycolumn, yytext()); }
  "NUMERIC"         { return new Token(TipoToken.TIPO_DE_DATO, yyline, yycolumn, yytext()); }
  "INT"             { return new Token(TipoToken.TIPO_DE_DATO, yyline, yycolumn, yytext()); }
  "SERIAL"          { return new Token(TipoToken.TIPO_DE_DATO, yyline, yycolumn, yytext()); }
```

```
"DATE"            { return new Token(TipoToken.TIPO_DE_DATO, yyline, yycolumn, yytext()); }

"TEXT"            { return new Token(TipoToken.TIPO_DE_DATO, yyline, yycolumn, yytext()); }

"BOOLEAN"         { return new Token(TipoToken.TIPO_DE_DATO, yyline, yycolumn, yytext()); }



/* Booleanos */
 "TRUE"            { return new Token(TipoToken.BOOLEANO, yyline, yycolumn, yytext()); }
 "FALSE"           { return new Token(TipoToken.BOOLEANO, yyline, yycolumn, yytext()); }

/* Funcion Agregacion */
 "SUM"             { return new Token(TipoToken.FUNCION_AGREGACION, yyline, yycolumn, yytext()); }
 "AVG"             { return new Token(TipoToken.FUNCION_AGREGACION, yyline, yycolumn, yytext()); }
 "COUNT"           { return new Token(TipoToken.FUNCION_AGREGACION, yyline, yycolumn, yytext()); }
 "MAX"             { return new Token(TipoToken.FUNCION_AGREGACION, yyline, yycolumn, yytext()); }
 "MIN"             { return new Token(TipoToken.FUNCION_AGREGACION, yyline, yycolumn, yytext()); }

/* signos*/
 "("               { return new Token(TipoToken.SIGNOS, yyline, yycolumn, yytext()); }
 ")"               { return new Token(TipoToken.SIGNOS, yyline, yycolumn, yytext()); }
 ","               { return new Token(TipoToken.SIGNOS, yyline, yycolumn, yytext()); }
 ";"               { return new Token(TipoToken.SIGNOS, yyline, yycolumn, yytext()); }
 "."               { return new Token(TipoToken.SIGNOS, yyline, yycolumn, yytext()); }
 "="               { return new Token(TipoToken.SIGNOS, yyline, yycolumn, yytext()); }
```

```
/* signos*/
 "("                { return new Token(TipoToken.SIGNOS, yyline, yycolumn, yytext()); }
 ")"                { return new Token(TipoToken.SIGNOS, yyline, yycolumn, yytext()); }
 ","                { return new Token(TipoToken.SIGNOS, yyline, yycolumn, yytext()); }
 ";"                { return new Token(TipoToken.SIGNOS, yyline, yycolumn, yytext()); }
 "."                { return new Token(TipoToken.SIGNOS, yyline, yycolumn, yytext()); }
 "="                { return new Token(TipoToken.SIGNOS, yyline, yycolumn, yytext()); }

"--".*  { return new Token(TipoToken.COMENTARIO, yyline, yycolumn, yytext()); }

/* Aritmeticos*/
 "+"                { return new Token(TipoToken.ARITMETICO, yyline, yycolumn, yytext()); }
 "-"                { return new Token(TipoToken.ARITMETICO, yyline, yycolumn, yytext()); }
 "*"                { return new Token(TipoToken.ARITMETICO, yyline, yycolumn, yytext()); }
 "/"                { return new Token(TipoToken.ARITMETICO, yyline, yycolumn, yytext()); }

/* relacionales*/
 "<"                { return new Token(TipoToken.RELACIONAL, yyline, yycolumn, yytext()); }
 ">"                { return new Token(TipoToken.RELACIONAL, yyline, yycolumn, yytext()); }
 "<="               { return new Token(TipoToken.RELACIONAL, yyline, yycolumn, yytext()); }
 ">="               { return new Token(TipoToken.RELACIONAL, yyline, yycolumn, yytext()); }

 /* logicos */
 "AND"              { return new Token(TipoToken.LOGICO, yyline, yycolumn, yytext()); }
 "OR"               { return new Token(TipoToken.LOGICO, yyline, yycolumn, yytext()); }
 "NOT"              { return new Token(TipoToken.LOGICO, yyline, yycolumn, yytext()); }
```

home > kevin-mushin > SistemasCunoc > SS_2024 > Lenguajes > ProyectoFinal > ProyectoFinal_LFP > src > main > java > org > example > Flex > SqlLexer.jflex

```
134   /* relacionales*/
141    "AND"              { return new Token(TipoToken.LOGICO, yyline, yycolumn, yytext()); }
142    "OR"               { return new Token(TipoToken.LOGICO, yyline, yycolumn, yytext()); }
143    "NOT"              { return new Token(TipoToken.LOGICO, yyline, yycolumn, yytext()); }
144
145
146   /* Identificadores */
147   {Identificador}   { return new Token(TipoToken.IDENTIFICADOR, yyline, yycolumn, yytext()); }
148
149   /* Comentarios (Ejemplo de comentarios de línea SQL) */
150
151   /* Ignorar espacios en blanco y saltos de línea */
152   [\t\n\r ]+        { /* Ignorar espacios y saltos de línea */ }
153
154   [a-zA-Z]+[A-Z0-9_]*[a-zA-Z0-9]* { return new Token(TipoToken.NO_RECONOCIDO, yyline, yycolumn, yytext()); }
155
156   .                 { return new Token(TipoToken.NO_RECONOCIDO, yyline, yycolumn, yytext()); }
157
158   /* EOF */
159   <<EOF>> { return new Token(TipoToken.EOF, yyline, yycolumn, "EOF"); }
160
```

Espacios en blanco: Ignora espacios, tabulaciones y saltos de línea.

- **Tokens no reconocidos**: Todos los patrones que no coinciden con los definidos se etiquetan como NO_RECONOCIDO, indicando un posible error léxico.

ANALIZADOR SINTACTICO: SQLANALIZADOR

Estructura General de SQLAnalizador

- **Atributos**:
    - erroresGenerados: Almacena tokens de errores de sintaxis.
    - indice: Marca la posición actual en la lista de tokens.
    - tokens: Contiene la lista de tokens generada por el analizador léxico.
    - tablasCreadas, schemasCreados, modificaciones, insersionesHechas: Listas para almacenar las entidades creadas o modificadas durante el análisis.

Métodos Principales

Control de Tokens:

- siguienteToken(): Avanza al siguiente token si el índice es menor que el tamaño total de la lista de tokens.
- tokenActual(): Devuelve el token actual sin avanzar el índice.

Análisis Sintáctico General (analizar Sintaxis):
Recorre la lista de tokens y, dependiendo del lexema, redirige el análisis a métodos específicos de acuerdo con el tipo de instrucción SQL (CREATE, ALTER, INSERT, etc.). Si encuentra un error, agrega el token a la lista de errores y continúa buscando el siguiente ;.

Métodos de Análisis Específicos:

- **analizarDDLCreate()**: Maneja la creación de bases de datos y tablas, redirigiendo el flujo a AnalizadorDatabase y Analizador table.

- **analizarDDLAlter()**: Gestiona modificaciones en tablas (ALTER TABLE) verificando si el siguiente token es TABLE y si se proporciona un identificador de tabla.

- **analizarDML()**: Analiza las instrucciones INSERT y SELECT, utilizando clases de análisis adicionales (AnalizadorInsercion, AnalizadorSelect) para manejar inserciones y consultas.

- **analizarDCL()**: Se utiliza para el comando UPDATE, mediante AnalizadorUpdate.

Manejo de Errores:

- **buscarProximoPuntoYcoma()**: Avanza hasta encontrar un ;, permitiendo al analizador ignorar tokens incorrectos hasta el siguiente punto de reanudación.

Validación y Getters:

- **esTokenValidoColumna()**: Verifica si el token corresponde a un identificador de columna válido.
- Métodos como getModificaciones(), getErroresGenerados() y obtnerTablasValidas() exponen datos para su posterior uso o revisión.

Estado	Token Actual	Acción	Siguiente Estado
0	CREATE	Llama a analizarDDLCreate()	Dependiendo del token (DATABASE o TABLE)
1	ALTER	Llama a analizarDDLAlter()	Busca siguiente token: TABLE
2	DROP	Llama a analizarDDLAlter()	Busca siguiente token: TABLE
3	INSERT	Llama a analizarDML()	Dependiendo del token (INTO, *, IDENTIFICADOR)
4	SELECT	Llama a analizarDML()	Dependiendo del token (* o columna)
5	UPDATE	Llama a analizarDCL()	Analiza la estructura del comando UPDATE
6	EOF o Fin de tokens	Termina el análisis	-
7	Otro	Agrega a erroresGenerados y busca ;	Salta al siguiente comando SQL

Cada sección del código SQL es analizada y descompuesta según su estructura gramatical mediante estados que siguen un flujo bien definido. A continuación, se proporciona una explicación detallada de los estados y transiciones, junto con los métodos específicos usados para procesar comandos SQL.

TABLA DE TRANSICIONES:

analizarDDLCreate(): Comprueba si el comando es CREATE DATABASE o CREATE TABLE y redirige a subanalizadores (AnalizadorDatabase, AnalizadorTable). Este método permite crear entidades de base de datos, generando objetos que se agregan a las listas schemasCreados y tablasCreadas.

- **analizarDDLAlter()**: Procesa modificaciones (ALTER) o eliminaciones (DROP) de tablas. Este método valida si el token siguiente es TABLE y luego verifica si el token de identificación es válido, gestionando las modificaciones a través de AnalizadorModificadores o condiciones especiales con AnalizadorIf.

- **analizarDML()**: Encargado de las operaciones INSERT y SELECT, controla el flujo hacia el AnalizadorInsercion para capturar valores de inserción en insersionesHechas o hacia AnalizadorSelect para consultas.

- **analizarDCL()**: Dirigido a procesar UPDATE, usa AnalizadorUpdate para capturar y validar la tabla, los campos a modificar y cualquier condición WHERE.

- **buscarProximoPuntoYcoma()**: Avanza hasta el próximo ;, permitiendo al analizador ignorar errores y recuperar la ejecución en la siguiente instrucción. Es clave para gestionar instrucciones con errores o incompletas.

ANALIZADORES ESPECÍFICOS:

Clase AnalizadorDatabase

Esta clase se encarga de analizar y procesar el comando CREATE DATABASE. Su objetivo es verificar que la instrucción siga la sintaxis adecuada y crear una instancia de Database en caso de ser válida.

Atributos

- tokens: Lista de tokens que representan los componentes de la instrucción.
- baseCreada: Objeto Database que representa la base de datos creada al finalizar el análisis.
- indice: Índice actual en la lista de tokens.

Métodos

1. **Constructor (AnalizadorDatabase)**
    - Inicializa la lista de tokens y el índice.
    - Crea un objeto Database que se almacenará en baseCreada si el análisis es exitoso.

2. **tokenActual()**
    - Devuelve el token en la posición actual, útil para evaluar el contenido sin modificar el índice.

3. **siguienteToken()**
    - Avanza al siguiente token en la lista mientras el índice no supere el tamaño de la lista.

4. **analizar()**
    - Este método realiza el análisis principal de CREATE DATABASE:
        - Llama a siguienteToken() para avanzar al siguiente token.
        - Si el token es un IDENTIFICADOR, se interpreta como el nombre de la base de datos, que se almacena en baseCreada.
        - Si el próximo token es ;, se considera que el análisis es exitoso y el índice se actualiza.
        - Si el token no cumple con el formato, lanza una excepción ErrorSintacticoException.

5. **evaluarToken(String tokenEsperado)**
    - Verifica si el token actual coincide con el valor esperado (; en este caso).
    - Si no es así, lanza una excepción ErrorSintacticoException.
    - Si es correcto, avanza al siguiente token.

6. **getBaseCreada() y setBaseCreada(Database baseCreada)**
    - Getters y setters para acceder y modificar baseCreada externamente.

Clase AnalizadorTable

La clase AnalizadorTable es responsable de analizar la instrucción CREATE TABLE. Evalúa la estructura completa de una tabla SQL, incluyendo nombres de columnas, tipos de datos y restricciones.

Atributos

- tokens: Lista de tokens que representan la instrucción CREATE TABLE.
- indice: Posición actual en la lista de tokens.
- tabla: Objeto Tabla que almacena la estructura de la tabla creada.
- tokensTable: Almacena los tokens que definen la estructura de la tabla.

Métodos:

1. **Constructor (AnalizadorTable)**
    - Inicializa la lista de tokens, el índice y el objeto tabla.
    - Prepara tokensTable para almacenar los tokens de la estructura de la tabla.

2. **tokenActual() y siguienteToken()**
    - Funcionan igual que en AnalizadorDatabase, devolviendo y avanzando tokens sin modificar el índice general.

3. **analizar()**
    - Realiza el análisis completo de la instrucción CREATE TABLE:
        - Avanza un token para verificar si el próximo es un IDENTIFICADOR.
        - Almacena el nombre de la tabla en tabla y en tokensTable.
        - Valida la apertura de paréntesis (y llama a analizarDefinicionDeTabla() para interpretar la estructura interna.
        - Verifica el cierre del paréntesis y la presencia de ; al final de la instrucción.

4. **evaluarSiguiente(String tokenEsperado)**
    - Similar a evaluarToken en AnalizadorDatabase, verifica si el siguiente token coincide con el valor esperado, avanzando al siguiente token si es correcto.

5. **analizarDefinicionDeTabla()**
    - Analiza la definición interna de la tabla, evaluando columnas y restricciones:
        - Si el token es CONSTRAINT, llama a analizarRestriccion() para interpretar las restricciones.
        - Si no, llama a analizarColumna() para identificar nombres de columnas y sus tipos de datos.
        - Avanza con cada coma encontrada, separando las definiciones de columnas o restricciones.

6. **analizarTipoDato()**
    - Evalúa el tipo de datos de una columna:
        - Reconoce tipos simples como INTEGER, BIGINT, DATE, TEXT, BOOLEAN y tipos complejos como VARCHAR, DECIMAL, NUMERIC.
        - Para tipos complejos, valida parámetros adicionales como longitud o precisión, y escala en el caso de DECIMAL y NUMERIC.

7. **analizarColumna()**
    - Analiza el nombre de una columna y su tipo de datos.
    - Llama a analizarTipoDato() para definir el tipo de la columna.
    - Verifica restricciones como PRIMARY KEY, NOT NULL, o UNIQUE si el token actual las define.

8. **analizarRestriccion()**
    - Procesa restricciones CONSTRAINT, especialmente claves externas (FOREIGN KEY).
    - Verifica el nombre de la restricción, columna de clave externa, tabla de referencia, y columna de referencia.

9. **getTabla()**
    - Devuelve la instancia tabla, con todos los tokens y estructura que representan la definición de la tabla.