

Universidad San Carlos de Guatemala
Centro Universitario de Occidente
División de Ciencias y Sistemas
Lenguajes Formales y de Programación
Ing. Daniel Gonzales



MANUAL TÉCNICO PROYECTO #1

Henry Josué Argueta Champet
Reg. academico: 202131261
Fecha: 02/10/2025

ÍNDICE

Descripción.....	3
Objetivos.....	3
Estructura de la práctica.....	4
Punto de Entrada Principal (Main.java).....	5
Enumeración de Tipos de Token (TokenType.java).....	6
Máquina de Estados (Clase Automata.java).....	7
Motor del Analizador Léxico (Clase Lexer.java).....	8
Interfaz Gráfica de Usuario (Clase LexerGUI.java).....	9
1. Definición de la Unidad Léxica (Clase Token.java).....	10
2. Gestión de la Configuración del Lenguaje (Clase Config.java).....	10
3. Lógica del Proceso de Análisis (Método analizar).....	11

Descripción.

Este documento ha sido creado con un propósito claro: servir como una guía detallada y completa para todos los colaboradores que deseen comprender, analizar y profundizar en el código fuente de nuestro programa. Mi objetivo es que, a través de esta explicación, puedan navegar por la estructura del proyecto con total confianza, entendiendo no sólo qué hace cada componente, sino también el porqué detrás de las decisiones de diseño y la lógica implementada.

Aquí, desglosamos cada sección del código, explicando su funcionalidad, las interacciones entre los módulos y los patrones de diseño utilizados. Esto no solo facilitará la incorporación de nuevos miembros al equipo, sino que también permitirá una colaboración más eficiente en futuras mejoras y actualizaciones.

Objetivos.

Este manual técnico ha sido diseñado como una guía esencial para ayudar a comprender y dominar todos los aspectos del software. El objetivo principal es que, como usuario, sientas completamente la capacidad para utilizar, mantener y configurar el sistema de manera eficiente.

- **Comprender el sistema:** Obtener una visión completa de la definición, el diseño, la estructura y la organización del software.
- **Gestionar el sistema:** Acceder a instrucciones claras y detalladas para configurar, mantener y resolver cualquier problema que pueda surgir.
- **Ser un usuario experto:** Familiar al usuario con las herramientas y funciones clave que te permitirán administrar, editar y configurar el software de manera efectiva.

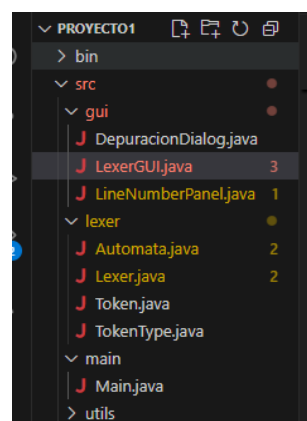
Estructura de la práctica.

Mi proyecto se organiza bajo una estructura modular que separa claramente las capas de presentación (interfaz gráfica), el núcleo del analizador léxico y las utilidades. Esta separación garantiza la modularidad, facilita el mantenimiento y permite una clara división de responsabilidades dentro del código.

La estructura principal es la siguiente:

1. bin: Este directorio contiene los archivos compilados del proyecto y no es parte de la codificación activa.
2. src: Es el directorio fuente principal, donde reside todo el código Java. Se subdivide en tres paquetes principales:
 - gui: Contiene todos los archivos responsables de la interfaz gráfica de usuario (GUI). Aquí se encuentran clases como LexerGUI.java (el frame principal de la aplicación), DepuracionDialog.java (posiblemente para mostrar el resultado del análisis o errores) y LineNumberPanel.java (un componente de utilidad para numerar las líneas de texto).
 - lexer: Este paquete es el corazón del compilador. Contiene las clases esenciales para el proceso de análisis léxico, como Lexer.java (el motor que lee el código fuente), Automata.java (la lógica de la máquina de estados para el reconocimiento de tokens), Token.java (la clase para representar una unidad léxica) y TokenType.java (la enumeración o clasificación de los posibles tokens).
 - main: Este paquete aloja la clase principal del proyecto, Main.java, que es el punto de inicio de la aplicación.
 - utils: Se utiliza para albergar cualquier clase de utilidad que pueda ser compartida entre los otros paquetes (aunque no se muestra explícitamente el contenido, su propósito es claro).

Esta arquitectura jerárquica asegura que las reglas de análisis permanezcan desacopladas de la presentación, logrando un diseño robusto y escalable.



Punto de Entrada Principal (Main.java)

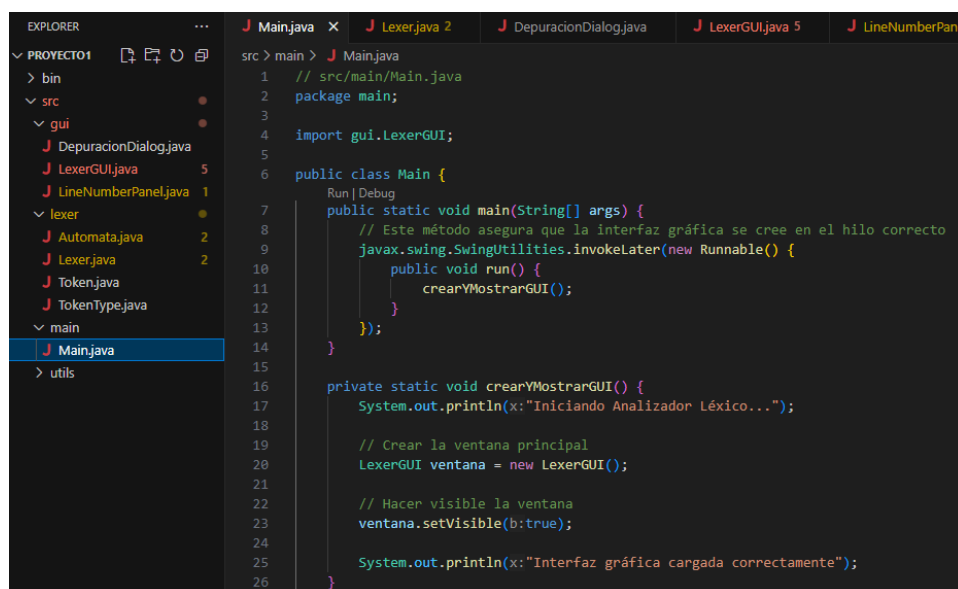
La clase Main.java opera como el punto de entrada de la aplicación, siendo responsable de inicializar y arrancar la interfaz gráfica de usuario (GUI). Para garantizar la estabilidad y la correcta gestión de la concurrencia en la aplicación Swing, implementé la lógica de arranque del GUI dentro del método main.

Garantía de Hilo de Eventos: Utilizó el método estático `javax.swing.SwingUtilities.invokeLater()` para invocar la creación y visualización de la interfaz. Esto es fundamental, ya que obliga a que todas las operaciones relacionadas con la GUI se ejecuten en el Hilo de Despacho de Eventos (Event Dispatch Thread o EDT), previniendo posibles problemas de concurrencia y asegurando que la interfaz se comporte de manera fluida y predecible.

Inicialización: El método privado `crearYMostrarGUI()` es el encargado de la inicialización. Primero, muestra un mensaje de "Iniciando Analizador Léxico..." en la consola para referencia.

Creación de la Ventana: Se instancia la clase principal de la interfaz, `LexerGUI`, y se utiliza el método `setVisible(true)` para hacer que la ventana sea visible para el usuario, finalizando el proceso de carga con el mensaje "Interfaz gráfica cargada correctamente".

Esta implementación garantiza que el analizador léxico se inicie de manera profesional, separando la lógica de arranque de la lógica de negocio del análisis.



```
src > main > J Main.java
1 // src/main/Main.java
2 package main;
3
4 import gui.LexerGUI;
5
6 public class Main {
7     public static void main(String[] args) {
8         // Este método asegura que la interfaz gráfica se cree en el hilo correcto
9         javax.swing.SwingUtilities.invokeLater(new Runnable() {
10             public void run() {
11                 crearYMostrarGUI();
12             }
13         });
14     }
15
16     private static void crearYMostrarGUI() {
17         System.out.println(x:"Iniciando Analizador Léxico...");
18
19         // Crear la ventana principal
20         LexerGUI ventana = new LexerGUI();
21
22         // Hacer visible la ventana
23         ventana.setVisible(b:true);
24
25         System.out.println(x:"Interfaz gráfica cargada correctamente");
26     }
27 }
```

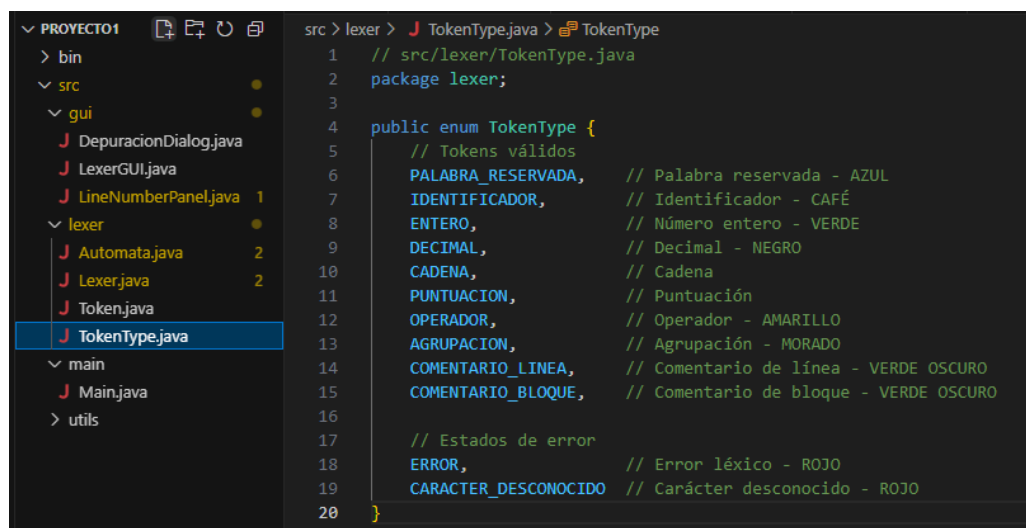
Enumeración de Tipos de Token (TokenType.java)

La clase TokenType.java se implementa como una enumeración (enum) para definir de manera estricta y controlada todos los posibles tipos de unidades léxicas que nuestro analizador puede reconocer. Esta estructura proporciona una clasificación exhaustiva que es utilizada por el motor del léxico (Lexer.java) para asignar un tipo a cada objeto Token.

La enumeración se divide en dos categorías principales para su gestión interna:

- Tokens Válidos: Definen la sintaxis reconocida del lenguaje. Incluyen categorías esenciales como PALABRA_RESERVADA, IDENTIFICADOR, tipos de datos numéricos (ENTERO, DECIMAL), CADENA, operadores, puntuación y agrupadores. También se definen explícitamente los tipos COMENTARIO_LINEA y COMENTARIO_BLOQUE.
- Estados de Error: Definen las categorías utilizadas para reportar fallos durante el análisis. Aquí se incluyen ERROR (para errores léxicos generales, como un identificador mal formado) y CARACTER_DESCONOCIDO (para símbolos que no pertenecen al alfabeto del lenguaje).

El uso de un enum garantiza que solo se utilicen tipos válidos y evita errores de escritura (typos) al momento de asignar el tipo, lo que resulta en un código más robusto y fácil de mantener.



```
1 // src/lexer/TokenType.java
2 package lexer;
3
4 public enum TokenType {
5     // Tokens válidos
6     PALABRA_RESERVADA, // Palabra reservada - AZUL
7     IDENTIFICADOR,     // Identificador - CAFÉ
8     ENTERO,             // Número entero - VERDE
9     DECIMAL,            // Decimal - NEGRO
10    CADENA,              // Cadena
11    PUNTUACION,          // Puntuación
12    OPERADOR,            // Operador - AMARILLO
13    AGRUPACION,          // Agrupación - MORADO
14    COMENTARIO_LINEA,    // Comentario de línea - VERDE OSCURO
15    COMENTARIO_BLOQUE,   // Comentario de bloque - VERDE OSCURO
16
17    // Estados de error
18    ERROR,               // Error léxico - ROJO
19    CARACTER_DESCONOCIDO // Carácter desconocido - ROJO
20 }
```

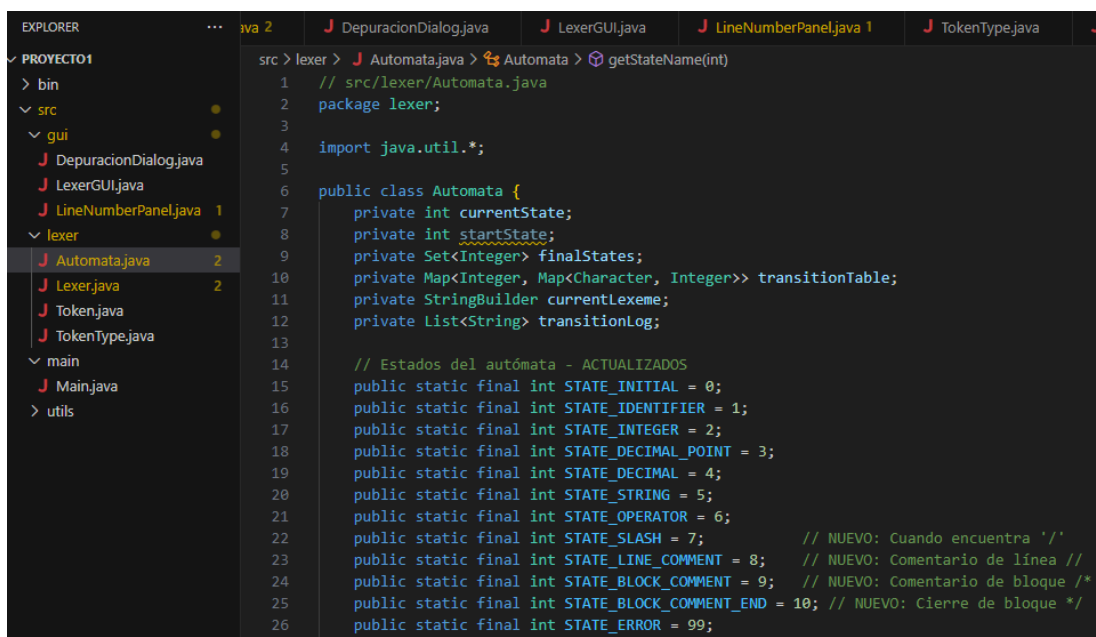
Máquina de Estados (Clase Automata.java)

La clase Automata.java representa el modelo de la máquina de estados finito (FSM) que implemento para el reconocimiento de los patrones léxicos del lenguaje. Es la pieza fundamental que permite determinar si una secuencia de caracteres constituye un identificador, un número, un operador, o cualquier otro tipo de token.

Atributos Centrales

La funcionalidad de la clase se basa en la gestión de sus atributos privados:

- **currentState**: Un entero que almacena el estado actual en el que se encuentra el autómata durante el escaneo del texto.
- **startState**: Indica el estado inicial (siempre 0) desde el cual comienza cualquier intento de reconocimiento de un nuevo token.
- **finalStates**: Un conjunto (Set) de enteros que define todos los estados que son válidos para terminar el reconocimiento de un token válido (por ejemplo, después de leer un número o una palabra reservada).
- **transitionTable**: Una estructura de datos compleja (Map<Integer, Map<Character, Integer>>) que define las reglas de transición del autómata. Esencialmente, indica a qué nuevo estado debe pasar el autómata dado un estado actual y un carácter de entrada.
- **currentLexeme**: Un StringBuilder que acumula los caracteres que componen el token que se está formando en el momento.



```
src > lexer > J Automata.java > Automata > getStateName(int)
1 // src/lexer/Automata.java
2 package lexer;
3
4 import java.util.*;
5
6 public class Automata {
7     private int currentState;
8     private int startState;
9     private Set<Integer> finalStates;
10    private Map<Integer, Map<Character, Integer>> transitionTable;
11    private StringBuilder currentLexeme;
12    private List<String> transitionLog;
13
14    // Estados del autómata - ACTUALIZADOS
15    public static final int STATE_INITIAL = 0;
16    public static final int STATE_IDENTIFIER = 1;
17    public static final int STATE_INTEGER = 2;
18    public static final int STATE_DECIMAL_POINT = 3;
19    public static final int STATE_DECIMAL = 4;
20    public static final int STATE_STRING = 5;
21    public static final int STATE_OPERATOR = 6;
22    public static final int STATE_SLASH = 7; // NUEVO: Cuando encuentra '/'
23    public static final int STATE_LINE_COMMENT = 8; // NUEVO: Comentario de línea //
24    public static final int STATE_BLOCK_COMMENT = 9; // NUEVO: Comentario de bloque /*
25    public static final int STATE_BLOCK_COMMENT_END = 10; // NUEVO: Cierre de bloque */
26    public static final int STATE_ERROR = 99;
27
```

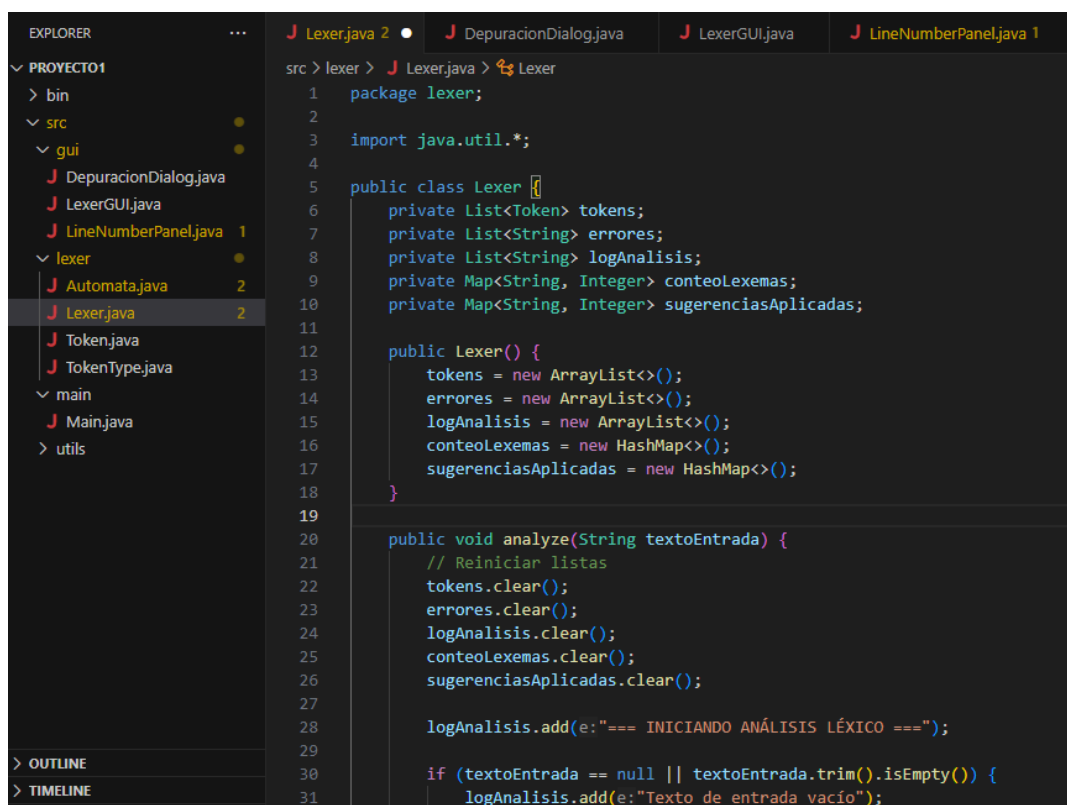
Motor del Analizador Léxico (Clase Lexer.java)

La clase Lexer.java es el motor central de nuestro analizador, responsable de tomar el texto fuente de entrada y, mediante el uso del autómata, transformarlo en una secuencia de tokens y registrar cualquier error léxico.

Atributos y Colecciones

La clase utiliza diversas colecciones para gestionar y reportar los resultados del análisis:

- tokens: Una lista (List<Token>) fundamental para almacenar todos los objetos token válidos que se reconocen durante el escaneo.
- errores: Una lista (List<String>) dedicada a almacenar los mensajes detallados de los errores léxicos encontrados.
- logAnálisis: Una lista que registra cada paso y evento significativo del proceso de análisis, útil para la depuración y para generar reportes.
- conteoLexemas y sugerenciasAplicadas: Estos mapas (Map<String, Integer>) están destinados a funcionalidades avanzadas, como el conteo de la frecuencia de cada lexema y el registro de las correcciones o sugerencias aplicadas, respectivamente.



```
1 package lexer;
2
3 import java.util.*;
4
5 public class Lexer {
6     private List<Token> tokens;
7     private List<String> errores;
8     private List<String> logAnálisis;
9     private Map<String, Integer> conteoLexemas;
10    private Map<String, Integer> sugerenciasAplicadas;
11
12    public Lexer() {
13        tokens = new ArrayList<>();
14        errores = new ArrayList<>();
15        logAnálisis = new ArrayList<>();
16        conteoLexemas = new HashMap<>();
17        sugerenciasAplicadas = new HashMap<>();
18    }
19
20    public void analyze(String textoEntrada) {
21        // Reiniciar listas
22        tokens.clear();
23        errores.clear();
24        logAnálisis.clear();
25        conteoLexemas.clear();
26        sugerenciasAplicadas.clear();
27
28        logAnálisis.add(e:"=== INICIANDO ANÁLISIS LÉXICO ===");
29
30        if (textoEntrada == null || textoEntrada.trim().isEmpty()) {
31            logAnálisis.add(e:"Texto de entrada vacío");
```

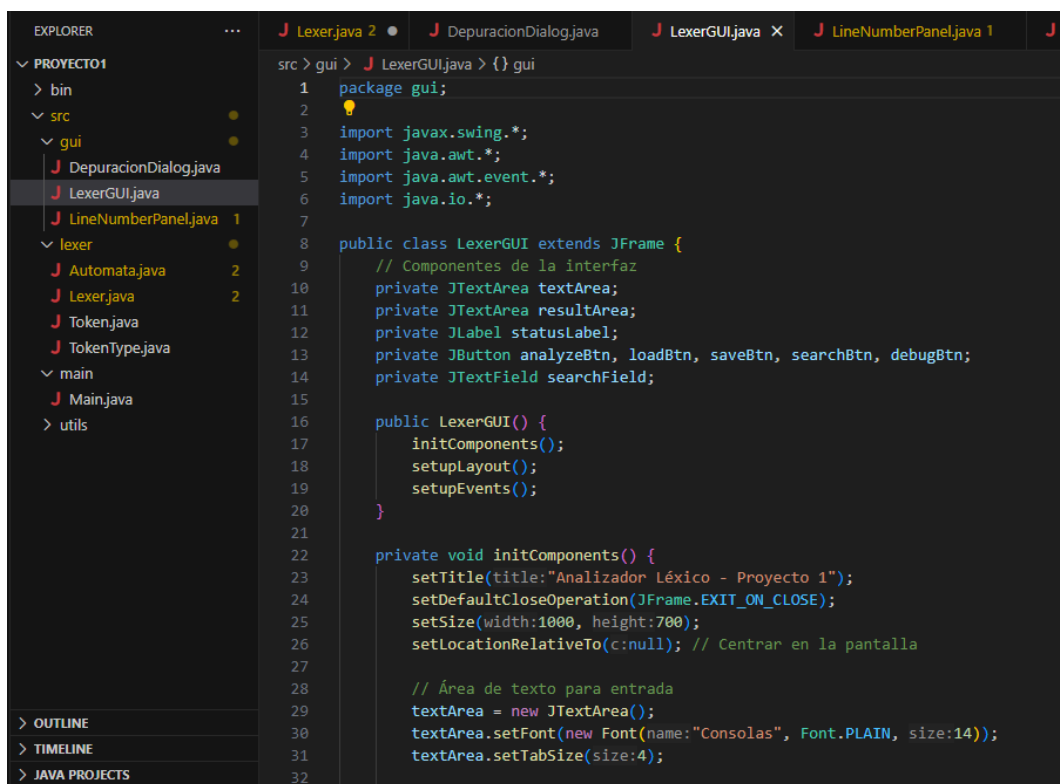

Interfaz Gráfica de Usuario (Clase LexerGUI.java)

La clase LexerGUI.java se extiende de JFrame y constituye la ventana principal de la aplicación. Su responsabilidad es gestionar todos los componentes visuales, capturar las acciones del usuario (eventos) y servir como puente entre la capa de presentación y el motor del analizador léxico (Lexer.java).

Componentes de la Interfaz

He definido los siguientes componentes privados para la manipulación de la interfaz:

- JTextArea textArea: Es el componente principal donde el usuario escribe o carga el código fuente a analizar. Lo he configurado con la fuente Consolas para una mejor visualización de código y he ajustado el tamaño de la tabulación.
- JTextArea resultArea: Destinado a mostrar la salida del análisis, incluyendo la lista de tokens generados y los errores léxicos.
- JLabel statusLabel: Se utiliza para mostrar mensajes de estado al usuario, como "Análisis completado", "Archivo cargado" o "Error al guardar".
- Botones (analyzeBtn, saveBtn, loadBtn, searchBtn, debugBtn): Componentes para ejecutar las acciones clave del usuario, como iniciar el análisis, guardar el código fuente o activar la función de depuración.



```
1 package gui;
2
3 import javax.swing.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import java.io.*;
7
8 public class LexerGUI extends JFrame {
9     // Componentes de la interfaz
10    private JTextArea textArea;
11    private JTextArea resultArea;
12    private JLabel statusLabel;
13    private JButton analyzeBtn, loadBtn, saveBtn, searchBtn, debugBtn;
14    private JTextField searchField;
15
16    public LexerGUI() {
17        initComponents();
18        setupLayout();
19        setupEvents();
20    }
21
22    private void initComponents() {
23        setTitle(title:"Analizador Léxico - Proyecto 1");
24        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25        setSize(width:1000, height:700);
26        setLocationRelativeTo(c:null); // Centrar en la pantalla
27
28        // Área de texto para entrada
29        textArea = new JTextArea();
30        textArea.setFont(new Font(name:"Consolas", Font.PLAIN, size:14));
31        textArea.setTabSize(size:4);
32    }
```

1. Definición de la Unidad Léxica (Clase Token.java)

La clase Token está diseñada para representar y gestionar una unidad léxica individual que ha sido reconocida por el analizador. Esta clase encapsula toda la información necesaria para el posterior análisis sintáctico y semántico.

Atributos

Un Token se define por los siguientes cuatro atributos privados:

- tipo (String): La clasificación léxica del token (por ejemplo, PALABRA_RESERVADA, ENTERO, OPERADOR), que se basa en la enumeración TokenType.
- lexema (String): La secuencia exacta de caracteres del código fuente que compone el token (por ejemplo, "if", "123", "=").
- fila (int) y columna (int): Las coordenadas exactas donde se inicia el token dentro del código fuente.

Constructor y Métodos

El constructor de la clase inicializa estos cuatro atributos con los valores proporcionados. Adicionalmente, implementé métodos de acceso (getters) que permiten a otras clases obtener el valor de cada atributo de forma controlada (por ejemplo, getTipo()). Finalmente, el método toString() está sobrescrito para proporcionar una representación legible del objeto ([tipo=X, lexema=Y, fila=Z, columna=W]), lo cual es invaluable para la depuración y la generación de reportes detallados.

2. Gestión de la Configuración del Lenguaje (Clase Config.java)

La clase Config es la responsable de centralizar y gestionar los elementos predefinidos del lenguaje, los cuales se cargan de forma dinámica desde un archivo de configuración externo, config.json. Esto permite modificar el alfabeto del lenguaje sin necesidad de recompilar el código fuente.

Atributos

La clase almacena los siguientes conjuntos de reglas:

- palabrasReservadas, operadores, puntuacion, agrupacion (List<String>): Listas que contienen todos los elementos válidos para cada categoría.
- comentarioLinea, comentarioBloqueInicio, comentarioBloqueFin (String): Cadenas que definen los delimitadores utilizados para ignorar secciones del código.

Inicialización

El constructor Config(String path) recibe la ruta del archivo config.json. Al ser invocado, lanza una excepción FileNotFoundException si el archivo no existe. Su implementación

utiliza librerías de parseo JSON (InputStream, JSONTokener, JSONObject) para leer el contenido del archivo y mapear los valores a las listas y variables correspondientes de la clase.

3. Lógica del Proceso de Análisis (Método analizar)

El método public void analizar(String texto) es el punto de inicio para el análisis léxico de un texto fuente específico.

Inicialización del Escaneo

Al comenzar, el método realiza las siguientes tareas de inicialización:

- Limpieza de Resultados: Se invoca la función clear() en las listas tokens y errores para asegurar que el análisis se realice en un estado limpio, sin mezclar resultados de ejecuciones anteriores.
- Contadores de Posición: Se inicializan los contadores de posición: fila y columna comienzan en 1, y posicion (el índice de recorrido del texto) comienza en 0.
- Preparación del Texto: El texto de entrada se convierte a un arreglo de caracteres (texto.toCharArray()) para facilitar el recorrido carácter por carácter.

Bucle Principal

El núcleo del método es un bucle while que itera mientras la posicion sea menor que la longitud del texto. Dentro del bucle, la lógica inicial maneja los espacios en blanco y saltos de línea:

- Si el carácter actual es un espacio o tabulación (isWhitespace): Se actualiza la columna.
- Si el carácter es un salto de línea (\n): Se incrementa la fila y la columna se reinicia a 1.

En ambos casos, se utiliza la instrucción continua para saltar el procesamiento y pasar al siguiente carácter sin generar un token, lo que permite al analizador ignorar los espacios y mantener una correcta trazabilidad de las coordenadas.