Universidad Rafael Landívar
Facultad de Ingeniería
Compiladores
Sección 01
Ing. Pedro David Gabriel Wong

PROYECTO DE APLICACIÓN
" ANALIZADOR LÉXICO"

Jhonatan Velasquez Fuentes - 1137521
José Rodrigo Valle Riveiro - 1149123
Esly Merileydi Ajsivinac Cacatzí - 1308723
Melanie Hernández Prado -1111622

Guatemala, 19 de Marzo del 2025

ÍNDICE

ANÁLISIS

1. **Nombre del Programa:**

   Compilador - Análisis Léxico

2. **Objetivo General:**

Desarrollar la primera fase de un compilador, implementando un analizador léxico que procese código fuente en un lenguaje de programación nuevo y genere como salida una lista de tokens y una tabla de símbolos, identificando errores léxicos en el proceso.
Aplicar los conceptos de análisis léxico para la traducción de un lenguaje. Así mismo implementar el uso de expresiones regulares para la evaluación de patrones.

3. **Alcance del programa.**

Entrada de código fuente**:** Permitir a los usuarios introducir código de forma manual o cargarlo desde un archivo.

Análisis Léxico:

- Reconocimiento de tokens que incluyen variables, operadores, palabras clave y caracteres especiales.
- Creación de una tabla de símbolos que contenga información relevante.
- Generación de un informe sobre los errores léxicos detectados.

Interfaz Gráfica: Desarrollo de una aplicación web que ofrezca un entorno interactivo para visualizar los resultados del análisis así mismo, un entorno interactivo para visualizar la salida del análisis.

Arquitectura Cliente-Servidor:

- **Frontend (React):** Permite la interacción con el usuario y la visualización de resultados.
- **Backend (Node.js con Java):** Procesa el análisis léxico y devuelve la información estructurada.

Escalabilidad: Este módulo es la base para futuras etapas del compilador, como el análisis sintáctico y semántico.

ANÁLISIS DE REQUISITOS

4. Requisitos funcionales.

La lógica dentro del analizador léxico se elaboró utilizando el lenguaje java, el programa cuenta también con una interfaz gráfica la cual permite la interacción entre el cliente y el servidor, el cliente se encarga únicamente de darle uso a la interfaz gráfica, mientras que el servidor contiene la lógica, clases y procedimientos que hacen que el programa funcione de forma correcta. Dicha interfaz permite accionar el proceso de análisis léxico del código fuente, provee los siguientes reportes:

- Reporte de tokens generados.
- Reporte de tabla de símbolos.
- Reporte de errores léxicos encontrados.

El programa es insensible a mayúsculas y minúsculas, acepta el uso de comentarios dentro del código, contempla los diferentes tipos de datos para las variables, por ejemplo; enteros, real, booleano, caracter, cadena, etc. Reconoce los números negativos y sin embargo, toda variable que se requiera utilizar en el lenguaje fuente debe ser previamente declarada.

Otro requisito funcional del sistema es el manejo de operadores aritméticos, estos son; suma, resta, multiplicación, división, potencia, módulo, autoincremento, autodecremento. De igual manera los operadores relacionales generan un resultado booleano, cuyo objetivo es comparar entre dos expresiones y devolver el valor lógico correspondiente, estos comparan entre valores booleanos y devuelven el resultado también como booleano (true/false).

Signos especiales, agrupación y control. Como signo de agrupación se utilizan los paréntesis “(” y “)”, como signo de control se tendrá el fin de una sentencia por punto y coma “;”. Para la agrupación de expresiones como if, for, while, se utilizan las llaves “{” y “}”

Para las sentencias de control se sigue una estructura, por ejemplo, IF:

$$if\ (< expresión\ booleana >)\{< instrucciones >\}$$
$$if(< expr >)\{< instr >\}\ else\ \{< instr >\}$$
$$if\ (< expr >)\{< instr >\}\ else\ if\ (< expr >)\{< instr >\}$$

Por otro lado, las sentencias cíclicas que se pueden utilizar son for y while. Para la sentencia FOR tenemos la siguiente estructura:

$$for\ (< def.variable >; < condición >; < paso >)\{< instr >\}$$

Para la sentencia DO - WHILE, tenemos la siguiente estructura:

$$do\ \{< instrucciones >\}\ while\ (< expresión\ booleana >);$$

Por otro lado el compilador contiene funciones reservadas del compilador estas son:
Instrucciones de salida:

- EscribirLinea: función que permite escribir en la consola un texto en específico ingresado como un parámetro de la función, su estructur es la siguiente:
  - $EscribirLinea("Hola\ Mundo")$
- Escribir: funcion similar a EscribirLinea, pero no genera un cambio de línea en la consola, su estructura es la siguiente:
  - $Escribir("Hola")$

Funciones predefinidas:

- Longitud (<var>);
  Devuelve la longitud del factor indicado.
- aCadena(<var>);
  Devuelve un tipo string con el valor que contenga el factor pasado como parámetro.

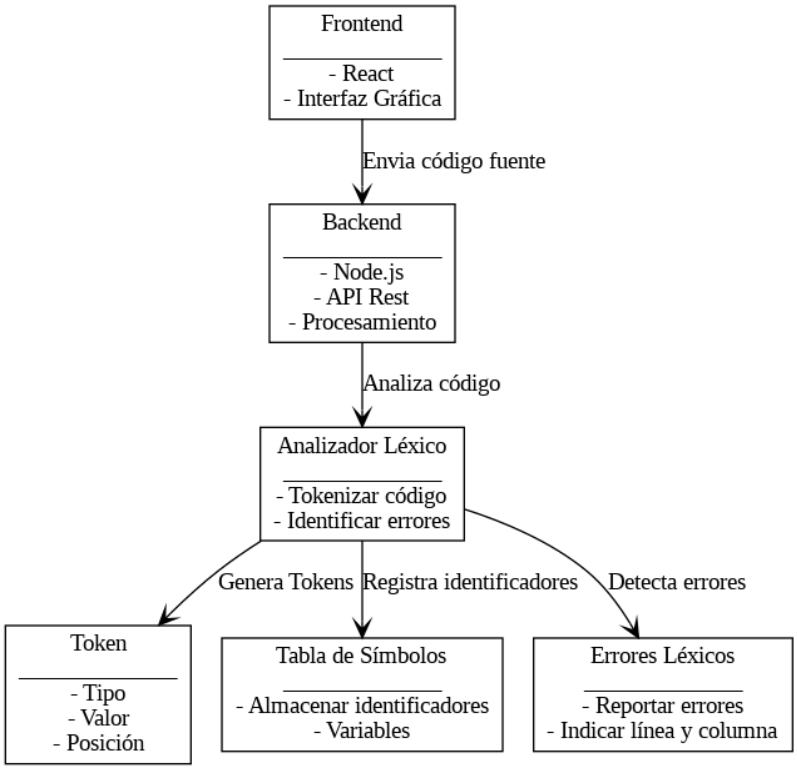
Tabla de Símbolos.
En el proceso del análisis léxico del programa fuente, se debe de crear y alimentar la tabla de símbolos respectivos de los compiladores. La estructura mínima de la tabla debe de ser:

Columna	**Objetivo**
Identificador	Nombre del identificador
Tipo de Token	Nombre del token
Línea	Número de línea donde se encontró
Columna	Posición del primer carácter donde se encontró

Por último, el control de errores es fundamental dentro del analizador léxico, es dar visibilidad de forma detallada de los errores léxicos que se han encontrado durante el análisis del código. Por tal motivo se genera una salida al programador donde se indican los errores encontrados en el código fuente.

MANUAL TÉCNICO

5. Diagramas de Clases Principales.



6. Diseño de pantalla.

Pantalla 1: Interfaz inicial que el usuario observa al ingresar. Tenemos una ventana para que el usuario visualice el analizador léxico, los tokens, la tabla de símbolos y los errores léxicos. Esta cambia dependiendo de los datos que el analizador reciba.

7. Algoritmos.

Algoritmo 1 - Análisis Léxico General.

- Iniciar posición, línea y columna en 0, 1 y 1 respectivamente.
- Mientras no se llegue al final de la entrada:
- Leer el carácter actual.
- Si es espacio, tabulación o salto de línea, avanzar.
- Si es comentario de una línea (//), avanzar hasta el salto de línea.
- Si es comentario multilínea (/* */), avanzar hasta el cierre o reportar error si no cierra.
- Si es letra, formar un identificador o palabra reservada.
- Si es dígito, formar un número (entero o real).
- Si es ', leer un carácter literal.
- Si es ", leer una cadena literal.
- Si es un operador (simple o compuesto), reconocerlo.
- Si no es reconocible, registrar error léxico.
- Al finalizar, devolver lista de tokens, tabla de símbolos y errores léxicos.

Algoritmo 2 - Reconocimiento de Identificadores o Palabras Reservadas.

- Detectar si el carácter es una letra.
- Formar un lexema uniendo letras, dígitos y guiones bajos consecutivos.
- Comparar el lexema (en minúsculas) con la lista de palabras reservadas.
- Si coincide, registrar un token de tipo PALABRA_RESERVADA.
- Si no, registrar un token de tipo IDENTIFICADOR y agregarlo a la tabla de símbolos si no está.

Algoritmo 3 - Reconocimiento de números (enteros y reales).

- Detectar si el carácter es un dígito.
- Formar un número entero leyendo dígitos consecutivos.
- Si se encuentra un punto (.), continuar leyendo dígitos y formar un número real.
- Registrar el token como ENTERO o REAL según corresponda.

Algoritmo 4 - Reconocimiento de Literales de Carácter.

- Detectar comilla simple (').
- Leer un carácter después de la comilla.
- Verificar si el siguiente carácter es una comilla de cierre (').
- Si cierra correctamente, registra un token de tipo CARACTER.
- Si no cierra, registra un error léxico.

Algoritmo 5 - Reconocimiento de Literales de Cadena.

- Detectar comilla doble (").
- Leer todos los caracteres hasta encontrar otra comilla doble.
- Si encuentra un salto de línea antes de cerrar, registrar error de cadena en múltiples líneas.
- Si cierra correctamente, registra un token de tipo CADENA.
- Si no cierra, registra un error léxico.

Algoritmo 6 - Reconocimiento de Operadores.

- Detectar si el carácter es un símbolo de operador (+, -, *, /, ^, #, !, <, >, =, &, |).
- Verificar si puede ser un operador compuesto (==, >=, <=, !=, ++, --, ||, &&).
- Si es compuesto, avanzar dos caracteres y registrar token de tipo OPERADOR.
- Si es simple, avanzar un carácter y registrar token de tipo OPERADOR.

8. Casos de prueba.

#	**Entrada (Código fuente)**	**Salida Esperada**
1	entero x = 10;	Tokens: entero (PALABRA_RESERVADA), x (IDENTIFICADOR), = (OPERADOR), 10 (ENTERO), ; (OPERADOR)
2	// Esto es un comentario	Ningún token generado (se ignora el comentario).
3	if (x >= 5) { escribir("Hola"); }	Tokens: if, (, x, >=, 5,), {, escribir, (, "Hola",), }, ;
4	"Cadena sin cerrar	Error: Símbolo no reconocido, Línea: x, Columna: x, Caracter: "
6	10.5.6	Error: Símbolo no reconocido, Línea: x, Columna: x, Caracter: .

DISEÑO Y ARQUITECTURA

- Lógica.

El analizador léxico lee línea por línea un archivo de texto que contiene posibles tokens del lenguaje. Para cada línea, el sistema identifica si la entrada corresponde a palabras reservadas, identificadores válidos o símbolos no reconocidos.
El analizador utiliza expresiones regulares y reglas de validación para determinar si un elemento es un identificador válido, un número, un símbolo o una palabra reservada.
El resultado de la validación se muestra por consola indicando la categoría del token y si es válido o inválido.

- Expresiones Regulares Utilizadas.

Identificadores:
Letras o combinación de letras, dígitos y guiones bajos, iniciando siempre con una letra.
Regla: ^[a-zA-Z_][a-zA-Z0-9_]*$

Números enteros:
Secuencia de dígitos.
Regla: ^\d+$

Palabras reservadas:
Coincidencia exacta con palabras clave predefinidas (no se usa expresión regular, sino comparación directa).

- Listado de palabras reservadas.

El analizador reconoce las siguientes palabras reservadas:

- include
- void
- int
- main
- for
- if
- while
- return
- break
- continue

- Librerías y dependencias.

Librerías estándar utilizadas:
re (expresiones regulares)
os (gestión de archivos)
No se requiere instalación de dependencias externas

- Arquitectura de la solución.

El proyecto está estructurado en un solo script que sigue una arquitectura simple pero clara:

Entrada de datos:
El sistema solicita al usuario la ruta de un archivo de texto con líneas a analizar.

Procesamiento:
El archivo se abre y procesa línea por línea.
Cada línea pasa por el proceso de validación para determinar su categoría: identificador, palabra reservada o símbolo desconocido.

Validación:
Se utiliza una combinación de expresiones regulares y listas predefinidas de palabras reservadas para categorizar cada token.

Salida:
Por cada línea se imprime en consola el resultado de la evaluación (token reconocido, tipo y validez).

MANUAL DE USUARIO

Introducción: Este manual está diseñado para guiar al usuario en el uso del Compilador - Análisis Léxico. Este programa permite analizar código fuente en un lenguaje específico, generando una lista de tokens, una tabla de símbolos y reportando errores léxicos.

9. Requisitos del Sistema

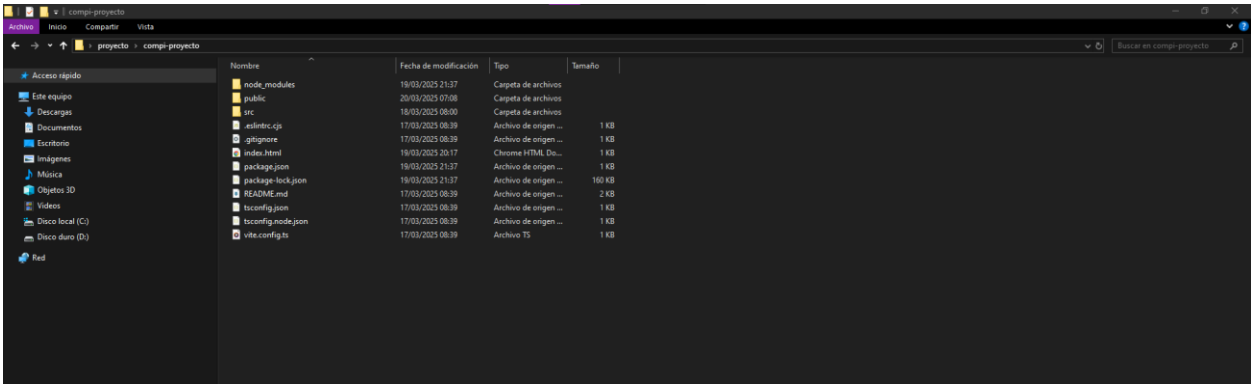
Para ejecutar correctamente el programa, se recomienda contar con:

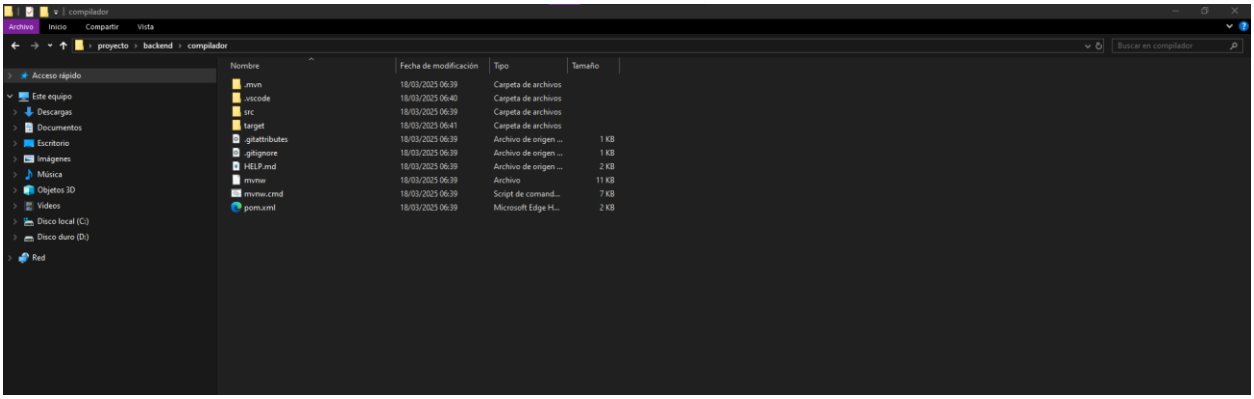
- Navegador web actualizado (Google Chrome, Mozilla Firefox, Microsoft Edge).
- Conexión a Internet para el acceso al frontend.
- Backend ejecutándose en un servidor local o remoto con Node.js y Java.

10. Instalación y Configuración

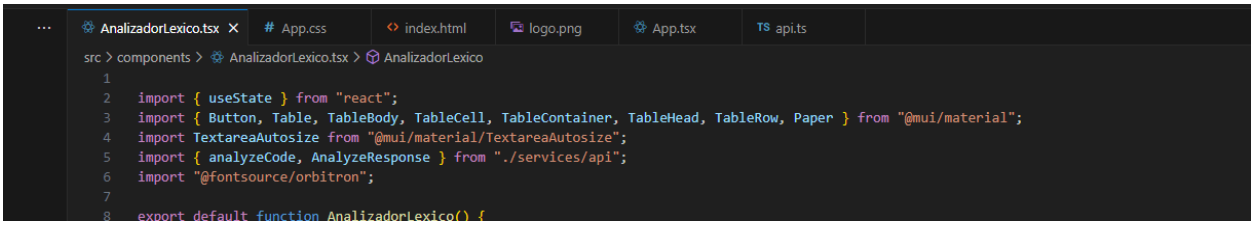
Si el usuario desea ejecutarlo localmente debe:

Clonar el repositorio desde GitHub:
Acceder al directorio del proyecto:

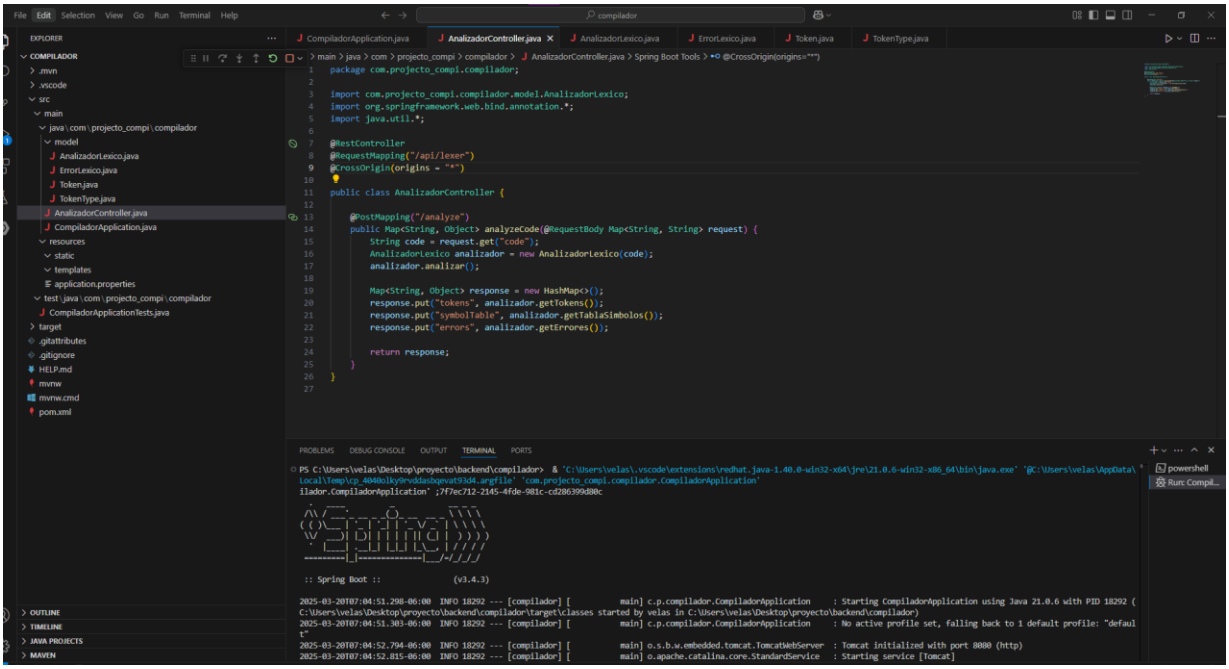




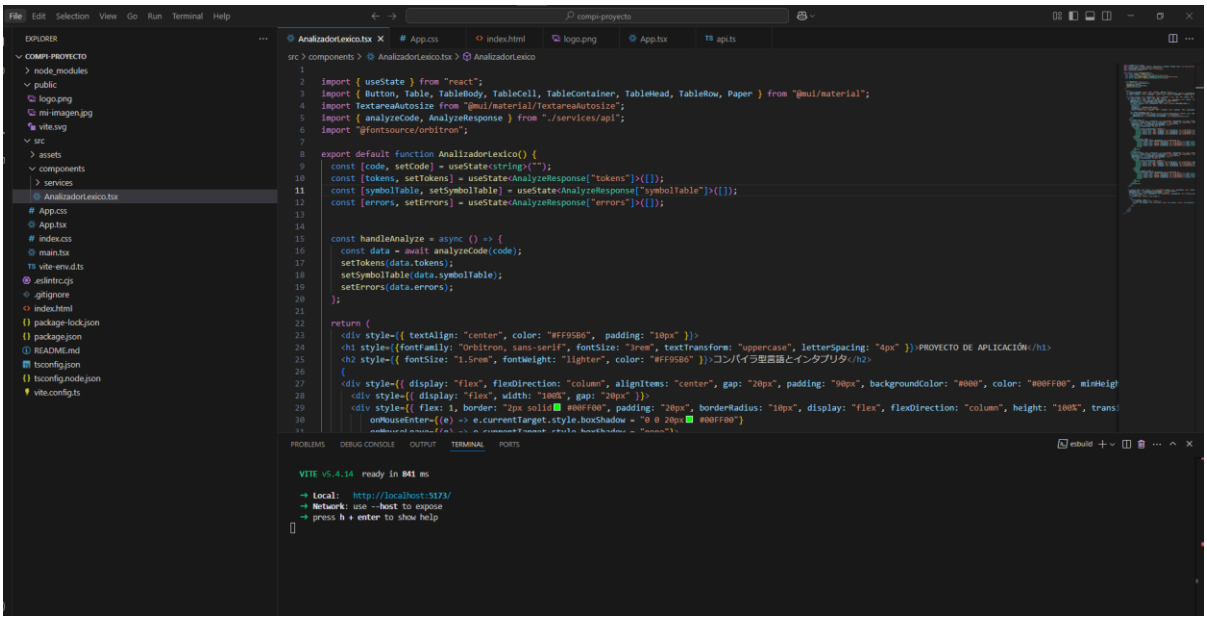
Instalar dependencias necesarias:



Iniciar el servidor backend:



Iniciar la interfaz frontend:



Acceder al programa desde el navegador en: http://localhost:5173

USO DEL PROGRAMA

11. Interfaz Principal

La interfaz consta de las siguientes secciones:

- **Área de Ingreso de Código**: Un cuadro de texto donde el usuario puede escribir o pegar código fuente.
- **Botón "Analizar Código"**: Ejecuta el análisis léxico.
- **Salida de Tokens**: Muestra los tokens generados.
- **Tabla de Símbolos**: Despliega la tabla de símbolos con identificadores.
- **Errores Léxicos**: Lista los errores detectados durante el análisis.

12. Procedimiento de Análisis

1. Ingrese el código fuente en el área de texto.
2. Haga clic en el botón **"Analizar Código"**.
3. Espere unos segundos mientras el programa procesa la información.
4. Revise los resultados en las secciones de **Tokens, Tabla de Símbolos y Errores Léxicos**.

14. **Interpretación de Resultados**

- **Tokens**: Lista de elementos reconocidos en el código, con su tipo.
- **Tabla de Símbolos**: Identificadores utilizados en el código con su tipo y ubicación.
- **Errores Léxicos**: Muestra caracteres o patrones no reconocidos en el código fuente.

15. Solución de Problemas

Errores Comunes

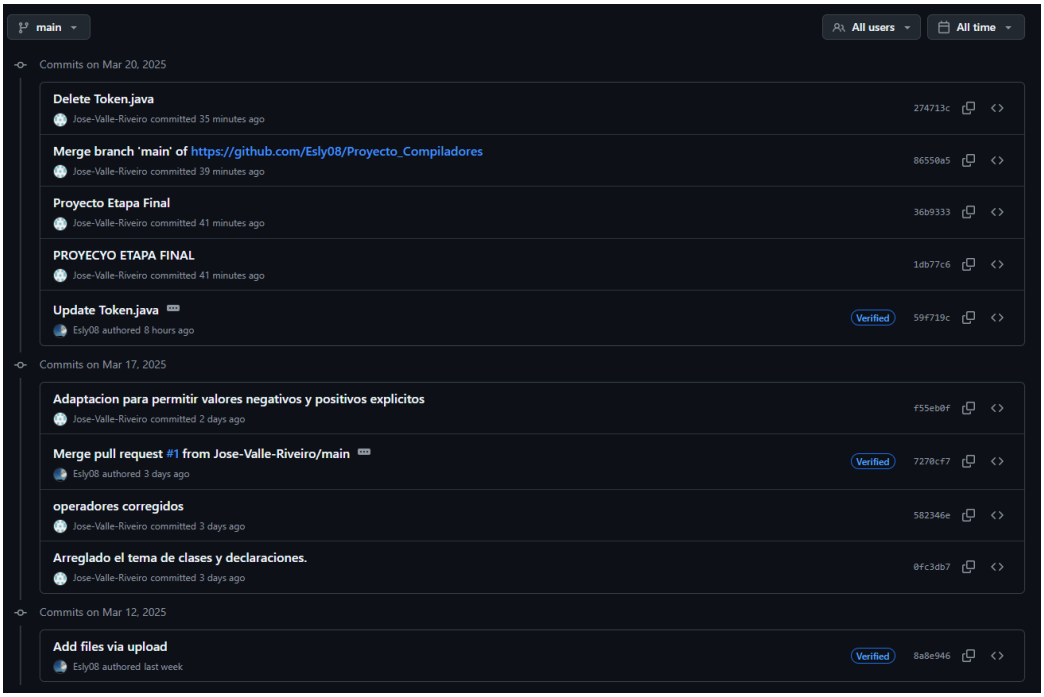
Error	Posible Causa	Solución
No se generan tokens	Código vacío o incorrecto	Verifique el código ingresado
Error de conexión	Backend no está corriendo	Asegúrese de ejecutar el servidor
Error en tabla de símbolos	Variables no declaradas correctamente	Revise la sintaxis del código

DESARROLLO DE GITHUB

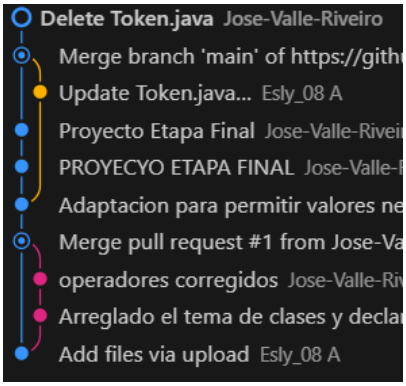
El desarrollo se realizó colaborativamente en la plataforma de gitHub, inicialmente se desarrolló la parte del backend y posteriormente se añadió toda la infraestructura del frontend.

[Enlace al repositorio.]

Historial de commits:



Historial de commits desde la rama principal en github.



Historial de ramas usando el source control de VS Code

CONCLUSIONES

1. El desarrollo del analizador léxico permite identificar y clasificar los elementos de un código fuente de manera automatizada, facilitando la detección de tokens y errores léxicos, lo cual es fundamental en la construcción de un compilador.

2. La implementación de una interfaz gráfica en React brinda a los usuarios una experiencia accesible para ingresar código, ejecutar el análisis y visualizar los resultados de manera clara y estructurada, mejorando la comprensión del proceso de compilación.

3. Este proyecto constituye el primer paso en la construcción de un compilador completo. La información generada en esta fase será crucial para etapas posteriores, como el análisis sintáctico y semántico, permitiendo la evolución del sistema hacia una solución más robusta y eficiente.

REFERENCIAS

- Node.js. (2025). *Run JavaScript everywhere.* https://nodejs.org/en
- Spring Boot. (2025). *Spring Boot.* https://spring.io/projects/spring-boot
- GitHub. (2025). *Proyecto_Compiladores.*

https://github.com/Esly08/Proyecto_Compiladores