Instituto Tecnológico de Costa Rica

Escuela de Computación

IC4810 Administración de Proyectos

Grupo: 60

2025, Semestre I

Profesor: Allan Rodriguez

Estudiantes:

Andrew López Herrera – 2021062132

Hillary Malespín Ulloa – 2021106074

Asignación:

Proyecto 01

Análisis Léxico y Sintáctico

Fecha de entrega:

Lunes 28 de marzo, 2025

Índice

Manual de usuario: instrucciones de compilación, ejecución y uso bien detalladas.

Los siguientes archivos se encuentran en la carpeta programa\OLIACCOMPILER\src\main\java\com\lexicalanalyzer del proyecto.

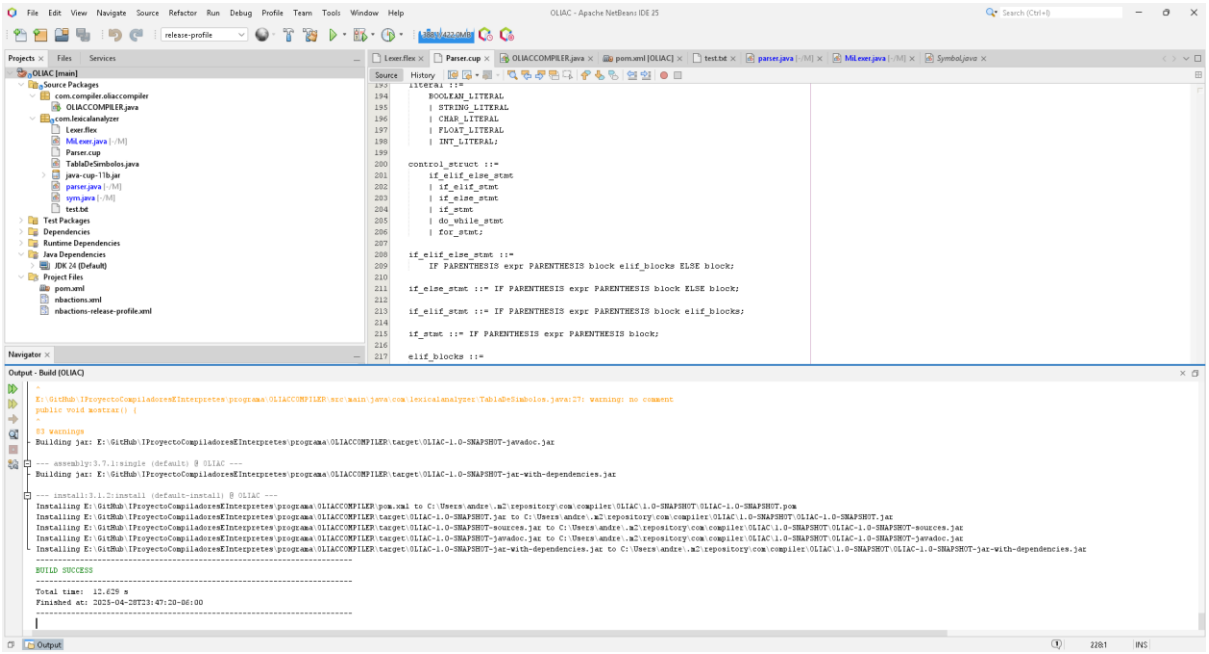
1. Se debe compilar el archivo Lexer.flex con el comando jflex Lexer.flex

```
PS E:\GitHub\IProyectoCompiladoresEInterpretes\programa\OLIACCOMPILER\src\main\java\com\lexicalanalyzer> jflex Lexer.flex
Reading "Lexer.flex"
Constructing NFA : 452 states in NFA
Converting NFA to DFA :
.....................................................................................................................
209 states before minimization, 190 states in minimized DFA
Old file "MiLexer.java" saved as "MiLexer.java~"
Writing code to "MiLexer.java"
```

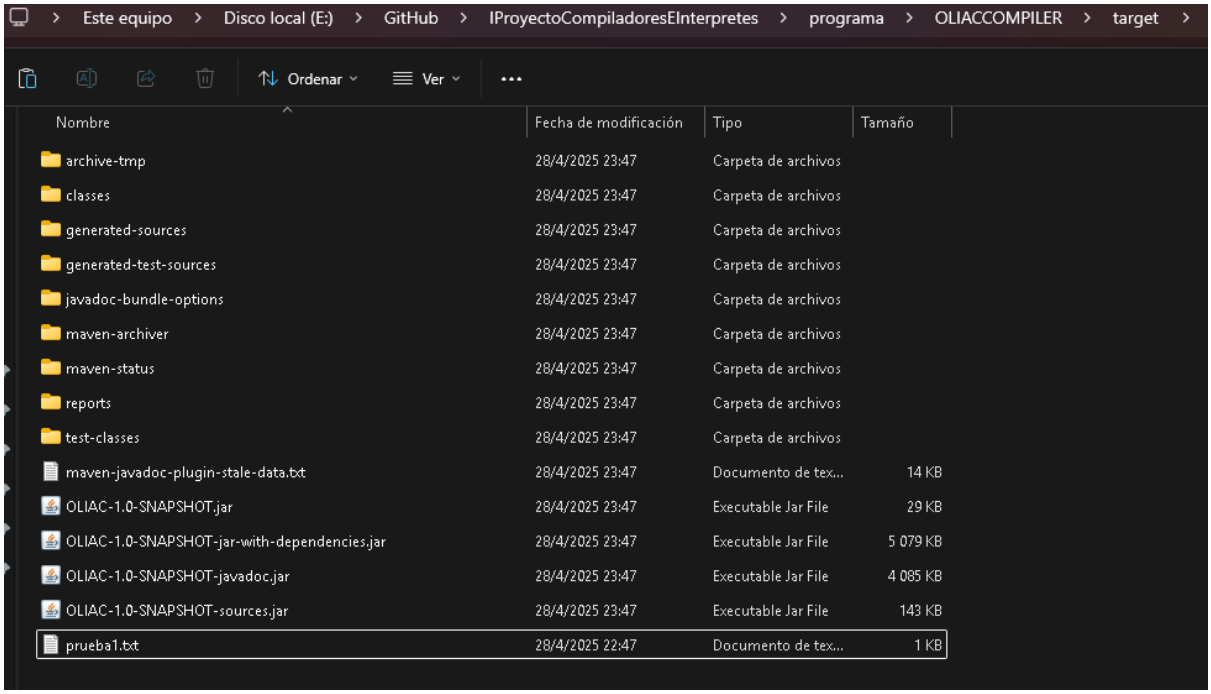
2. Se debe compilar el archivo Parser.cuo con el comando java -jar java-cup-11b.jar -parser parser -symbols sym Parser.cup

```
PS E:\GitHub\IProyectoCompiladoresEInterpretes\programa\OLIACCOMPILER\src\main\java\com\lexicalanalyzer> java -jar java-cup-11b.jar -parser parser -symbols sym Parser.cup
------- CUP v0.11b 20160615 (GIT 4ac7450) Parser Generation Summary -------
  0 errors and 0 warnings
  62 terminals, 41 non-terminals, and 121 productions declared,
  producing 219 unique parse states.
  0 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  0 conflicts detected (0 expected).
  Code written to "parser.java", and "sym.java".
--------------------------------------------------- (CUP v0.11b 20160615 (GIT 4ac7450))
```

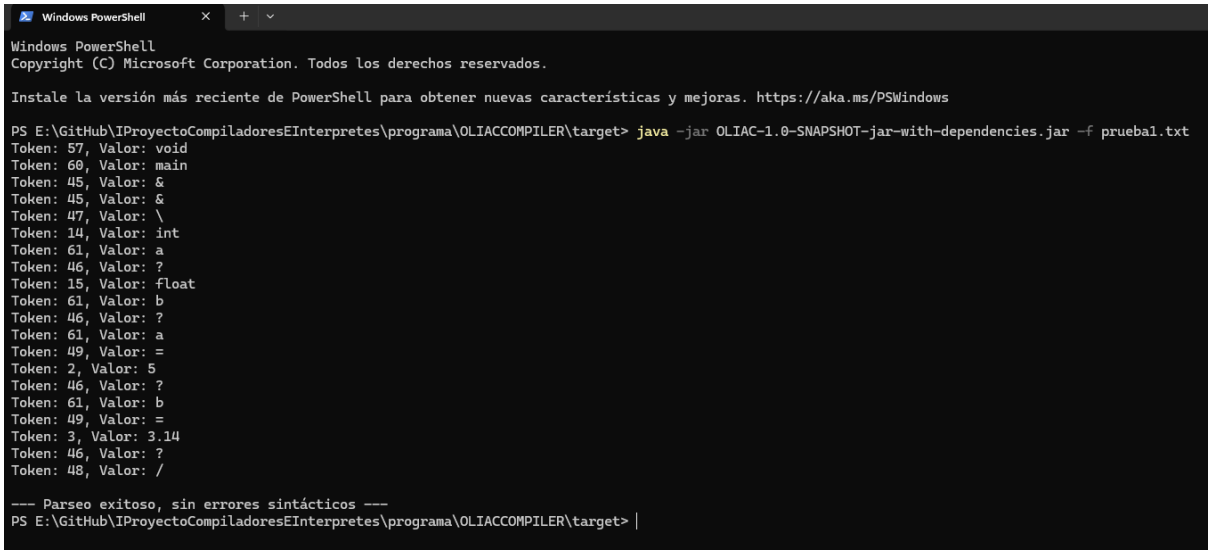
3. Se debe compilar el proyecto en NetBeans utilizando Clean and Build para generar un nuevo proyecto.



4. Se debe ir a la carpeta programa\OLIACCOMPILER\target del proyecto. Ahí se encontrará el archivo OLIAC-1.0-SNAPSHOT-jar-with-dependencies.jar que es el proyecto compilado. Ahora solo hay que ingresar un archivo de pruebas.

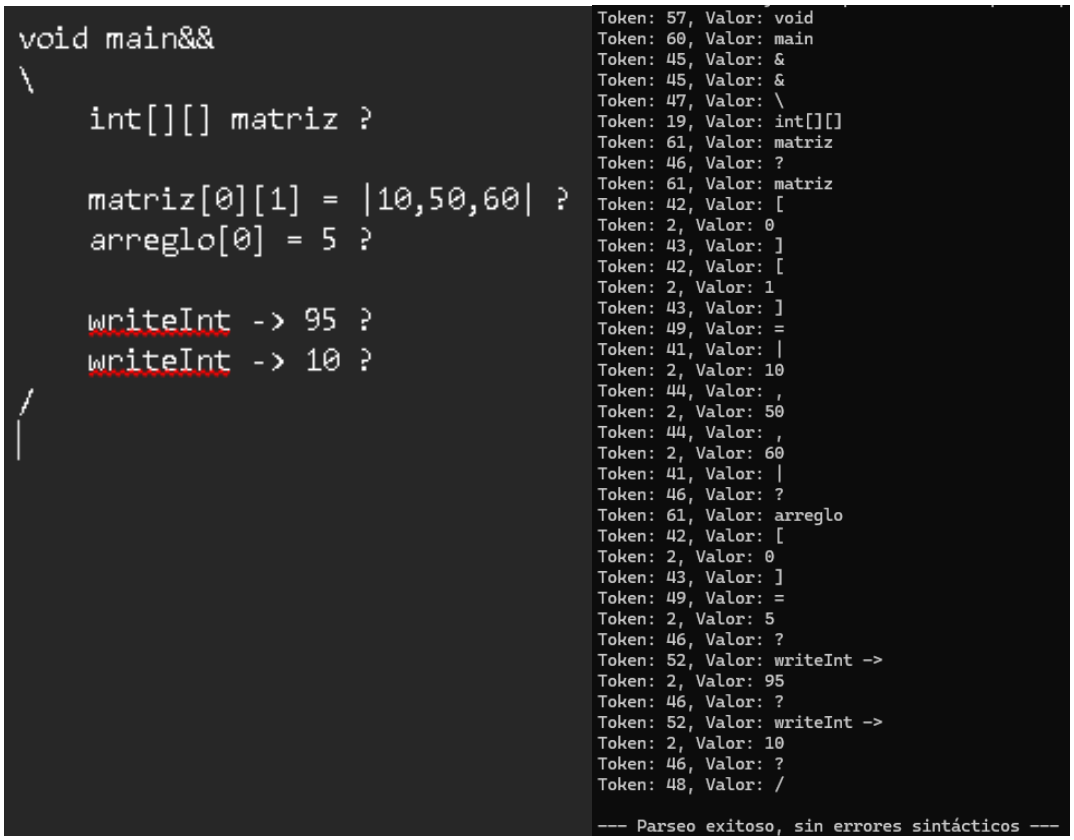


5. Cuando se haya agregado el archivo de prueba se puede utiliza el comando java -jar OLIAC-1.0-SNAPSHOT-jar-with-dependencies.jar -f prueba1.txt para realizar el análisis léxico, se puede cambiar prueba1.txt por cualquier nombre de archivo existente.



Pruebas de funcionalidad: incluir screenshots.

Prueba 1



Se puede observar que el analizador léxico maneja correctamente los arreglos, además que se puede utilizar funciones de escritura en el código.

Prueba 2

```
void main&&
\
    int i ?
    for & int i = 0 ? i < 10 ? i ++ &
    \
        writeInt -> i ?
    /
/
```

```
Token: 57, Valor: void
Token: 60, Valor: main
Token: 45, Valor: &
Token: 45, Valor: &
Token: 47, Valor: \
Token: 14, Valor: int
Token: 61, Valor: i
Token: 46, Valor: ?
Token: 11, Valor: for
Token: 45, Valor: &
Token: 14, Valor: int
Token: 61, Valor: i
Token: 49, Valor: =
Token: 2, Valor: 0
Token: 46, Valor: ?
Token: 61, Valor: i
Token: 38, Valor: <
Token: 2, Valor: 10
Token: 46, Valor: ?
Token: 61, Valor: i
Token: 24, Valor: ++
Token: 45, Valor: &
Token: 47, Valor: \
Token: 52, Valor: writeInt ->
Token: 61, Valor: i
Token: 46, Valor: ?
Token: 48, Valor: /
Token: 48, Valor: /

--- Parseo exitoso, sin errores sintácticos ---
```

Permite el uso de la estructura de control for, así mismo, se puede observar que los paréntesis cumplen con lo indicado por el profesor, además, dentro de cada estructura se puede escribir código.

Prueba 3

```
void main&&
\
    int i ?
    for & int i = 0 ? i < 10 ? i ++ &
    \
        writeInt -> i ?
    /
/

int sumar&int x, int y&
\
    return x + y ?
/

void hola&&
\
    int resultado ?
    resultado = sumar&5, 8& ?
    writeInt -> resultado ?
/
```

```
Token: 48, Valor: /
Token: 48, Valor: /
Token: 14, Valor: int
Token: 61, Valor: sumar
Token: 45, Valor: &
Token: 14, Valor: int
Token: 61, Valor: x
Token: 44, Valor: ,
Token: 14, Valor: int
Token: 61, Valor: y
Token: 45, Valor: &
Token: 47, Valor: \
Token: 56, Valor: return
Token: 61, Valor: x
Token: 27, Valor: +
Token: 61, Valor: y
Token: 46, Valor: ?
Token: 48, Valor: /
Token: 57, Valor: void
Token: 61, Valor: hola
Token: 45, Valor: &
Token: 45, Valor: &
Token: 47, Valor: \
Token: 14, Valor: int
Token: 61, Valor: resultado
Token: 46, Valor: ?
Token: 61, Valor: resultado
Token: 49, Valor: =
Token: 61, Valor: sumar
Token: 45, Valor: &
Token: 2, Valor: 5
Token: 44, Valor: ,
Token: 2, Valor: 8
Token: 45, Valor: &
Token: 46, Valor: ?
Token: 52, Valor: writeInt ->
Token: 61, Valor: resultado
Token: 46, Valor: ?
Token: 48, Valor: /

--- Parseo exitoso, sin errores sintácticos ---
```

Se puede observar que se pueden manejar diferentes funciones dentro del código fuente, así mismo, también se evalúa el código dentro de estas funciones.

Descripción del problema.

El proceso de construcción de un compilador para un lenguaje de programación personalizado presenta múltiples retos que involucran desde el análisis léxico hasta el análisis sintáctico de los programas fuente. La finalidad principal de este proyecto es desarrollar una herramienta capaz de leer archivos de texto que contienen programas escritos en un lenguaje propio, identificando correctamente los tokens que los conforman, organizándolos en una tabla de símbolos y validando que su estructura sintáctica cumpla con las reglas gramaticales definidas. Se busca facilitar la detección de errores léxicos y sintácticos desde etapas tempranas, permitiendo una retroalimentación clara y útil para los usuarios del lenguaje.

El proyecto OLIACCOMPILER implementa un analizador léxico utilizando JFlex, el cual se encarga de reconocer los distintos componentes léxicos como palabras reservadas, operadores, identificadores y literales. Posteriormente, se construye un analizador sintáctico utilizando Java CUP, el cual procesa la secuencia de tokens para verificar la correcta formación de instrucciones de acuerdo con la gramática definida. Además, se incluye la generación de una tabla de símbolos, que almacena los tokens relevantes capturados durante el proceso, sirviendo de base para etapas posteriores de compilación como la generación de código o el análisis semántico.

Al diseñar esta herramienta, se garantiza un entorno controlado donde los errores se detectan de manera eficiente, apoyando tanto en la enseñanza como en la evolución de un lenguaje de programación propio. La correcta implementación del análisis léxico, la verificación sintáctica y la gestión de símbolos constituye un paso esencial para el desarrollo de sistemas de compilación más complejos.

Diseño del programa: decisiones de diseño, algoritmos usados.

- Se eligió el uso de **JFlex** para generar el analizador léxico (MiLexer.java) a partir de un archivo de especificación (Lexer.flex)
- Java CUP para construir el analizador sintáctico (parser.java) a partir de una gramática definida en el archivo (Parser.cup).
- El diseño modular del programa también fue una decisión clave: se separaron los paquetes en com.lexicalanalyzer para los componentes del análisis léxico y sintáctico, y com.compiler.oliaccompiler para la ejecución principal del proyecto.

Librerías usadas: creación de archivos, etc.

- picocli: Esta es la librería que permite al usuario ingresar parámetros de manera más intuitiva.
- Java cup: permite realizar el parser de los tokens ingresados.
- Java flex: permite convertir los lexemas ingresados en el código fuente a tokens.

Análisis de resultados: objetivos alcanzados, objetivos no alcanzados, y razones por las cuales no se alcanzaron los objetivos (en caso de haberlos).

	Logrado	No logrado	Justificación
Gramática	Sí		
Scanner	Sí		
Parser	Sí		
El sistema debe leer un archivo fuente.	Sí		
Tabla de simbolos	Sí		
Indicar si el archivo fuente puede o no ser generado por la gramática.	Sí		
Reportar y manejar los errores léxicos y sintácticos encontrados. Debe utilizar la técnica de Recuperación en Modo Pánico (error en línea y continúa con la siguiente). En el reporte de errores es fundamental indicar la línea en que ocurre.	Sí		

Bitácora (autogenerada en git, commit por usuario incluyendo comentario).

https://github.com/AndrewLopezHerrera/IProyectoCompiladoresEInterpretes.git