TEC | Tecnológico de Costa Rica

Proyecto 1 Compiladores e Intérpretes

Curso Compiladores e Intérpretes

Profesora Erika Marin Schumann

Realizado por:

Parra Valverde Nicole Tatiana

Wu Zhong Danielo

Nahomi Bolaños Valverde

Tecnológico de Costa Rica

II Semestre 2025

23/09/2025

Introducción

El presente documento describe el diseño, la implementación y la validación de la primera etapa de un compilador para el lenguaje ABS: el análisis léxico o scanner. Esta fase constituye el punto de partida del proceso de compilación, pues transforma una secuencia de caracteres en una secuencia de tokens significativos para las etapas posteriores (análisis sintáctico y semántico) [4]. En concordancia con el enunciado del curso, el scanner debía reconocer identificadores, operadores, palabras reservadas y literales (enteros, reales con notación científica, caracteres y cadenas), además de ignorar comentarios y espacios en blanco [4]. El sistema, escrito en Java y construido con la herramienta JFlex [2], [1], no solo debía clasificar tokens, sino también detectar y reportar errores léxicos con número de línea, sin abortar el recorrido ante el primer error (recuperación), y finalmente producir dos salidas: un listado de errores y un reporte consolidado de tokens válidos con su tipo y sus líneas de aparición, incluyendo el conteo de ocurrencias por línea [3].

La solución propuesta respeta la sensibilidad a mayúsculas/minúsculas para identificadores y reservadas, diferencia con claridad los operadores simbólicos de los “operadores-palabra” (como AND, OR o DIV), y aplica restricciones específicas del enunciado, por ejemplo que los números reales contengan al menos un dígito a cada lado del punto decimal [4]. Para facilitar el mantenimiento y la legibilidad, la especificación léxica se concentra en un archivo *scanner.flex*, a partir del cual JFlex genera la clase Scanner.java [1]. Un ejecutor de pruebas (*ProyectoCompi1.java*) realiza el escaneo de archivos .abs y delega la recolección y el reporte a una clase auxiliar (*TokenCollector.java*) [2].

Estrategia de Solución

La arquitectura se organiza en tres componentes con nombres y responsabilidades explícitas. El primer componente es la especificación JFlex ***scanner.flex***, donde se definen macros y reglas léxicas mediante expresiones regulares. Primeramente se habilitan las opciones **`%class Scanner`**, **`%unicode`**, **`%public`**, **`%line`**, **`%column`**, **`%ignorecase`** y **`%type int`**, con el objetivo de generar la clase **`Scanner`** con soporte Unicode, visibilidad pública, rastreo de línea y columna, análisis **case-insensitive** y retorno de enteros desde **`yylex()`**[1].

A partir de ahí, se introducen macros básicos como **`DIGITO`** y **`LETRA`**, y constructores mayores como **`IDENTIFICADOR`** y **`ESPACIOS`**, que permiten modularizar el reconocimiento de lexemas [4]. La validez de los identificadores se acota con **`IDENTIFICADOR`** (letra inicial y hasta 127 caracteres) y se complementa con **`IDENTIFICADOR_INVALIDO`** para detectar longitudes mayores a las permitidas. Asimismo, los comentarios no anidados se reconocen y descartan mediante **`COM_LLAVES`** y **`COM_PAREST`**. En el frente numérico, se distinguen **`HEX`**, **`OCT`**, **`ENTERO_DEC`**, **`REAL_BASICO`**, **`EXP`** y **`REAL`**, contemplando el formato real con dígito a ambos lados del punto. Para los errores numéricos y colas no válidas se emplean **`ERROR_REAL_PUNTO_LIDER`**, **`ERROR_REAL_PUNTO_COLA`**, **`OCT_INVALIDO`**, **`HEX_INVALIDO`**, **`ID_TAIL`**, **`NUM_DEC_E`**, **`NUM_ANY`** y **`ERROR_NUM_SEGUIDO_TEXTO`**. Los literales de texto se gestionan con **`STRING`** y **`CHAR`**, y sus versiones sin cierre con **`STRING_INCOMP`** y **`CHAR_INCOMP`**. En cuanto a operadores, se separan los **simbólicos** en **`OPER_SIMBOLO`** y los de palabra en **`OPER_PALABRA`**; las **palabras reservadas** se listan en **`RESERVADAS`** [4]. Para capturar usos indebidos dentro de identificadores se utilizan **`SIMBOLO_ILEGAL_EN_ID`** y **`ID_CON_SIMBOLO_ILEGAL`**. Finalmente, se añade una regla **catch-all** con el patrón **`.`** que reporta **`ERROR_LEXICO`** y asegura recuperación [1].

Una decisión importante fue el orden de las reglas. Primero aparecen las **guardas de error** (**`ERROR_REAL_PUNTO_LIDER`**, **`ERROR_REAL_PUNTO_COLA`**, **`OCT_INVALIDO`**, **`HEX_INVALIDO`**, **`ERROR_NUM_SEGUIDO_TEXTO`**) para interceptar entradas prohibidas antes de que coincidan con patrones válidos. Enseguida se listan los operadores de mayor longitud dentro de **`OPER_SIMBOLO`** —por ejemplo `<=`, `>=`, `<>`, `++`, `--`, `**`— a fin de evitar su fragmentación en tokens simples. A continuación se sitúa **`OPER_PALABRA`** (que incluye `AND`, `OR`, `NOT`, `DIV`, `MOD`, `IN`, `SHL`, `SHR`) y luego **`RESERVADAS`**, de modo que esos lexemas se

clasifiquen como **OPERADOR** en lugar de **PALABRA_RESERVADA**. Las reglas de **IDENTIFICADOR**, **IDENTIFICADOR_INVALIDO** e **ID_CON_SIMBOLO_ILEGAL** completan el conjunto de reconocimiento y validación de nombres. Con este orden, el scanner garantiza detección temprana de errores, correcta desambiguación de operadores compuestos y clasificación consistente de tokens [1], [4].

El segundo componente es el ciclo de vida de ejecución. Una vez procesado ***scanner.flex*** por JFlex, se genera ***Scanner.java***, que es consumido por el programa principal ***ProyectoCompi1.java***. Este programa presenta un menú de pruebas, resuelve la ruta del archivo dentro de `test/`, crea un `FileReader` y construye una instancia de `proyectocompi1.Scanner`. A partir de ahí, el método `yylex()` itera hasta **YYEOF** (véase la regla <<EOF>> { return YYEOF; } al final del archivo), y cada acción de regla invoca a **TokenCollector.add(tipo,lexema,línea)** o **TokenCollector.addError(codigo,lexema,línea)** según corresponda [2].

El tercer componente es ***TokenCollector.java***, responsable de la normalización y del reporte. Para cumplir con el análisis **case-insensitive** sin perder la primera estructura con que apareció cada lexema, TokenCollector agrupa internamente con una clave formada por **tipo + "|" + lexema** normalizado en minúsculas (controlado por **`setUnifyCase(true)`** por defecto), pero conserva **`displayLexema`** para la salida. Internamente, la clase **`TokenBucket`** mantiene **`lineCounts`** como un **`TreeMap<Integer,Integer>`** que acumula ocurrencias por línea mediante **`addAtLine(int linea)`**. Al finalizar el escaneo, **`TokenCollector.printResults()`** emite primero **"=== ERRORES LÉXICOS ==="** con cada **`LexError`** registrado y, luego, **"=== TOKENS ENCONTRADOS ==="**, reagrupando por tipo y ordenando por **`displayLexema`** con comparación **case-insensitive**. El formato de salida sigue la especificación, con columnas **"Token"**, **"Tipo de Token"** y **"Línea(s)"**, e indicando repeticiones por línea como `n(ocurrencias)`.

En cuanto a supuestos y límites explícitos, **COM_LLAVES** y **COM_PAREST** implementan comentarios no anidados tal y como lo exige el enunciado. **STRING** acepta cadenas en una sola línea y **CHAR** restringe el literal a un único carácter; **STRING_INCOMP** y **CHAR_INCOMP** detectan los casos sin cierre. En los reales, **REAL_BASICO** y **REAL** fuerzan la presencia de al menos un dígito a cada lado del punto, mientras **EXP** admite notación científica. Aunque **`%column`** está habilitado, en esta etapa la salida reporta el **número de línea** (**`yyline + 1`**) para cumplir exactamente lo solicitado. Esta separación de responsabilidades, reconocimiento en `Scanner` y consolidación en `TokenCollector`—

mantiene el scanner enfocado en clasificar y permite un reporte final claro, estable y alineado con los requisitos del proyecto [2], [3].

Análisis de los resultados

Al hacer una revisión estática del código y durante el desarrollo del mismo se determinó una cobertura prácticamente total de los requisitos funcionales de la etapa léxica. El manejo de espacios y comentarios se comporta como se espera, sin generar tokens; los identificadores válidos se reconocen con la restricción de longitud y se reportan con el tipo adecuado, mientras que los identificadores con símbolos ilegales o de longitud excesiva disparan los errores específicos correspondientes. El conjunto de palabras reservadas exigido por el enunciado está contemplado y, como se estableció en la estrategia, los lexemas que también son operadores-palabra se clasifican como OPERADOR por prioridad de regla, mientras que el resto se registran como PALABRA_RESERVADA.

En el caso de los literales, el sistema distingue entre enteros decimales, octales y hexadecimales y valida los reales con su notación científica opcional, respetando la restricción de que exista al menos un dígito a cada lado del punto. Se incorporaron comprobaciones para octales y hexadecimales inválidos, así como para números seguidos de texto, que resultan en errores informativos. Las cadenas y los caracteres se aceptan en su forma válida y, si carecen de cierre o intentan abarcar múltiples líneas, se registran los errores correspondientes. Los operadores simbólicos (incluyendo los de doble carácter y **), se detectan de forma estable gracias al orden de coincidencia. El mecanismo de recuperación frente a errores es robusto: cada coincidencia errónea se consume y el escaneo continúa, evitando los errores en cascada y permitiendo un reporte consolidado al final.

El ejecutor de pruebas (*ProyectoCompi1.java*) facilita la exploración con diversos archivos de entrada, mostrando el archivo seleccionado y ejecutando el escaneo hasta su conclusión. A la fecha de esta redacción, consideramos la cobertura funcional de la etapa léxica como muy alta; los pequeños ajustes pendientes se limitan a aseguramiento de rutas y a incorporar capturas de ejecución que respalden cada caso de prueba del plan.

Actividad / Tarea funcional	Criterio de aceptación	Evidencia en código
Ignorar espacios y saltos de línea	No generan tokens	ESPACIOS → /* ignore */
Ignorar comentarios {...} y (*...*) (no anidados)	Contenido no produce tokens	COM_LLAVES, COM_PAREST → /* ignore */
Identificadores válidos (1–127 chars, letra inicial)	Acepta y clasifica como IDENTIFICADOR	IDENTIFICADOR

Identificador con símbolo ilegal	Se reporta error, no token	ID_CON_SIMBOLO_ILEGAL → addError(...)
Identificador >127 chars	Se reporta ERROR_IDENTIFICADOR_LONGITUD	IDENTIFICADOR_INVALIDO
Palabras reservadas (todas las exigidas)	Clasificadas como PALABRA_RESERVADA	RESERVADAS
Operadores simbólicos (incluye dobles y **)	Clasificados como OPERADOR	OPER_SIMBOLO
Operadores-palabra (AND, OR, NOT, DIV, MOD, IN, SHL, SHR)	Clasificados como OPERADOR	OPER_PALABRA
Enteros decimales, octales y hexadecimales válidos	Clasificados como literales numéricos	ENTERO_DEC, OCT, HEX
Reales válidos (dígito a ambos lados del punto, con/ sin exponente)	Clasificados como LITERAL_REAL	REAL_BASICO, EXP, REAL
Errores numéricos: .5, 5., 09, 0xG, 123abc, 3.14pi	Se reportan con códigos específicos	Guarda ERROR_* antes de reglas válidas
Strings en una línea y chars de 1 carácter	Aceptados; sin cierre se reporta error	STRING, CHAR, STRING_INCOMP, CHAR_INCOMP
Captura genérica de símbolos no reconocidos	ERROR_LEXICO	Regla . al final
Case-insensitive para identificadores y reservadas	valor, VALOR, VaLoR → mismo token	%ignorecase + TokenCollector.unifyCase=true
Conteo por línea y formato de salida requerido	Token Tipo Línea(s) con n(n) para repeticiones	TokenCollector.printResults()

Lecciones aprendidas

Desde el punto de vista técnico, la experiencia confirmó la importancia del orden de las reglas en JFlex, particularmente para capturar primero los casos de error y evitar ambigüedades en operadores de longitud variable. La combinación de %ignorecase con una normalización interna de lexemas en el recolector resultó clave para ofrecer un reporte limpio y coherente, manteniendo a la vez la primera forma escrita de cada token. Asimismo, separar el reconocimiento léxico de la construcción del reporte permitió un código más claro y más fácil de mantener: el scanner se dedica a reconocer, mientras que el TokenCollector gestiona el agrupamiento, los conteos por línea y la presentación.

En el plano del trabajo en equipo, los comentarios exhaustivos en el código agilizaron la incorporación de quienes no estuvieron desde el primer minuto en la implementación, y la definición temprana de criterios (como la clasificación de operadores-palabra) evitó retrabajos posteriores. Finalmente, la disciplina de probar desde un pequeño menú de archivos de entrada fortaleció la trazabilidad entre los requisitos y los comportamientos observados, y deja encaminada la transición hacia las fases subsiguientes del compilador.

Casos de pruebas

A continuación se documentan los casos ejercitados desde el menú de *ProyectoCompi1.java*, utilizando el enum *TestFile*. En todos los escenarios, el flujo es: abrir el archivo *test/*.abs*, instanciar *proyectocompi1.Scanner* y consumir tokens con yylex() hasta YYEOF (regla <<EOF>> { return YYEOF; }).

Al finalizar, *TokenCollector.printResults()* emite "=== ERRORES LEXICOS ===" seguido de "=== TOKENS ENCONTRADOS ===", agrupados por tipo y con líneas/ocurrencias. Dado que *%ignorecase* está activo y *TokenCollector.setUnifyCase(true)* por defecto, la agrupación de lexemas es case-insensitive, preservando la primera grafía en displayLexema.

```
package com.mycompany.proyectocompi1;

public enum TestFile {
    VALID_BASICS("01_valid_basics.abs", "Prueba básica con variables, literales y reservadas"),
    NUMBERS_VALID("02_numbers_valid.abs", "Números válidos (decimales, octales, hex, reales)"),
    NUMBERS_INVALID("03_numbers_invalid.abs", "Reales inválidos (.5, 5.), hex/oct malos, num+texto"),
    IDENTIFIERS_RESERVED("04_identifiers_and_reserved.abs", "Identificadores, reservadas y operador-palabra"),
    STRINGS_CHARS("05_strings_and_chars.abs", "Strings y chars válidos + errores de cierre"),
    COMMENTS_OPERATORS("06_comments_and_operators.abs", "Comentarios, operadores largos (++ -- ** <= >= <>)"),
    MIXED_EDGES("07_mixed_edge_cases.abs", "Casos límite: id largo, char inválido, punto suelto"),
    ALL_IN_ONE("08_all_in_one.abs", "Regresión: mezcla de todos los anteriores");

    private final String fileName;
    private final String description;

    TestFile(String fileName, String description) {
        this.fileName = fileName;
        this.description = description;
    }

    public String getFileName() { return fileName; }
    public String getDescription() { return description; }
}
```

enum TestFile utilizado en el menú de pruebas

A continuación se detallan los casos de prueba implementados:

VALID_BASICS — 01_valid_basics.abs
Objetivo
Verificar el reconocimiento básico de PALABRA_RESERVADA (p. ej., VAR, BEGIN, END), IDENTIFICADOR, OPERADOR (simbólicos), y literales LITERAL_ENTERO, LITERAL_REAL, LITERAL_CHAR, LITERAL_STRING.
Entradas

- Declaración con `VAR … : INTEGER;`
- Bloque `BEGIN … END` con asignaciones a x, y, z, s usando 5, `3.0E2`, `'A'`, `"hola"`.

Resultado Esperado

- Sin **errores léxicos**.
- `VAR`, `BEGIN`, `END`, `INTEGER` como **PALABRA_RESERVADA** (por `RESERVADAS`).
- `:=`, `+`, `:`, `,`, `;` y paréntesis/otros, si aparecen, como **OPERADOR** (por `OPER_SIMBOLO`).
- x, y, z, s como **IDENTIFICADOR** (por `IDENTIFICADOR`).
- 5 → **LITERAL_ENTERO**; `3.0E2` → **LITERAL_REAL**; `'A'` → **LITERAL_CHAR**; `"hola"` → **LITERAL_STRING**

Resultado Obtenido

```
Ejecutando prueba: Prueba básica con variables, literales y reservadas
Archivo: ProyectoCompi1/src/main/java/com/mycompany/proyectocompi1/test/01_valid_basics.abs

=== ERRORES LEXICOS ===
(ninguno)

=== TOKENS ENCONTRADOS ===
Token               Tipo de Token      Línea(s)
INTEGER             IDENTIFICADOR      1
s                   IDENTIFICADOR      6
x                   IDENTIFICADOR      1, 3, 4
y                   IDENTIFICADOR      1, 4
z                   IDENTIFICADOR      5
'A'                 LITERAL_CHAR       5
5                   LITERAL_ENTERO     3
3.0E2               LITERAL_REAL       4
"hola"              LITERAL_STRING     6
+                   OPERADOR           4
,                   OPERADOR           1
:                   OPERADOR           1, 3, 4, 5, 6
;                   OPERADOR           1, 3, 4, 5, 6
=                   OPERADOR           3, 4, 5, 6
BEGIN               PALABRA_RESERVADA  2
END                 PALABRA_RESERVADA  7
VAR                 PALABRA_RESERVADA  1
```

NUMBERS_VALID — 02_numbers_valid.abs

Objetivo

Confirmar literales numéricos válidos: `LITERAL_ENTERO`, `LITERAL_OCTAL`, `LITERAL_HEX`, `LITERAL_REAL` (con y sin exponente).

Entradas

`0`, `7`, `0755` (octal), `0x1F`/`0XABcd` (hex), `5.0`, `0.5`, `3.14`, `1.5e-4`, `3.0E5`.

Resultado Esperado

- Sin **errores léxicos**.
- `0755` → **LITERAL_OCTAL** (por `OCT`); `0x1F`, `0XABcd` → **LITERAL_HEX** (por `HEX`); `0`, `7` → **LITERAL_ENTERO**; `5.0`, `0.5`, `3.14`, `1.5e-4`, `3.0E5` → **LITERAL_REAL** (por `REAL`).

- Operadores y asignaciones como **OPERADOR**.

Resultado Obtenido

```
Ejecutando prueba: Números válidos (decimales, octales, hex, reales)
Archivo: ProyectoCompi1/src/main/java/com/mycompany/proyectocompi1/test/02_numbers_valid.abs

=== ERRORES LEXICOS ===
(ninguno)

=== TOKENS ENCONTRADOS ===
Token               Tipo de Token      Línea(s)
a                   IDENTIFICADOR      2
b                   IDENTIFICADOR      2
c                   IDENTIFICADOR      2
d                   IDENTIFICADOR      2
e                   IDENTIFICADOR      2
r1                  IDENTIFICADOR      3
r2                  IDENTIFICADOR      3
r3                  IDENTIFICADOR      3
r4                  IDENTIFICADOR      3
r5                  IDENTIFICADOR      3
0                   LITERAL_ENTERO     2
7                   LITERAL_ENTERO     2
0x1F                LITERAL_HEX        2
0XABcd              LITERAL_HEX        2
0755                LITERAL_OCTAL      2
0.5                 LITERAL_REAL       3
1.5e-4              LITERAL_REAL       3
3.0E5               LITERAL_REAL       3
3.14                LITERAL_REAL       3
5.0                 LITERAL_REAL       3
:                   OPERADOR           2(5), 3(5)
;                   OPERADOR           2(5), 3(5)
=                   OPERADOR           2(5), 3(5)
BEGIN               PALABRA_RESERVADA  1
END                 PALABRA_RESERVADA  4
```

NUMBERS_INVALID — 03_numbers_invalid.abs

Objetivo

Disparar las guardas de error numéricas y de número seguido de texto.

Entradas

`.5`, `5.`, `0xG1`, `078`, `123abc`, `0x1FZ`, `3.2e5foo`.

Resultado Esperado

- **`.5`** → **ERROR_LEXICO_REAL_PUNTO_LIDER** (por ERROR_REAL_PUNTO_LIDER).
- **`5.`** → **ERROR_LEXICO_REAL_PUNTO_COLA** (por ERROR_REAL_PUNTO_COLA).

- **`0xG1`** → ERROR_LEXICO_HEXADECIMAL_INVALIDO (por HEX_INVALIDO).
- **`078`** → ERROR_LEXICO_OCTAL_INVALIDO (por OCT_INVALIDO).
- **`123abc`, `0x1FZ`, `3.2e5foo`** → ERROR_LEXICO_NUM_SEGUIDO_POR_TEXTO (por ERROR_NUM_SEGUIDO_TEXTO).
- El resto (identificadores, BEGIN/END, `:=`, `;`) se tokeniza normalmente.

Resultado Obtenido

```
Ejecutando prueba: Reales inválidos (.5, 5.), hex/oct malos, num+texto
Archivo: ProyectoCompi1/src/main/java/com/mycompany/proyectocompi1/test/03_numbers_invalid.abs

=== ERRORES LEXICOS ===
[ERROR_LEXICO_REAL_PUNTO_LIDER] '.5' en línea 2
[ERROR_LEXICO_REAL_PUNTO_COLA] '5.' en línea 3
[ERROR_LEXICO_HEXADECIMAL_INVALIDO] '0xG1' en línea 4
[ERROR_LEXICO_OCTAL_INVALIDO] '078' en línea 5
[ERROR_LEXICO_NUM_SEGUIDO_POR_TEXTO] '123abc' en línea 6
[ERROR_LEXICO_NUM_SEGUIDO_POR_TEXTO] '0x1FZ' en línea 7
[ERROR_LEXICO_NUM_SEGUIDO_POR_TEXTO] '3.2e5foo' en línea 8

=== TOKENS ENCONTRADOS ===
Token               Tipo de Token      Línea(s)
no1                 IDENTIFICADOR      5
nx1                 IDENTIFICADOR      4
pegado1             IDENTIFICADOR      6
pegado2             IDENTIFICADOR      7
pegado3             IDENTIFICADOR      8
rBad1               IDENTIFICADOR      2
rBad2               IDENTIFICADOR      3
:                   OPERADOR           2, 3, 4, 5, 6, 7, 8
;                   OPERADOR           2, 3, 4, 5, 6, 7, 8
=                   OPERADOR           2, 3, 4, 5, 6, 7, 8
BEGIN               PALABRA_RESERVADA  1
END                 PALABRA_RESERVADA  9
```

IDENTIFIERS_RESERVED — 04_identifiers_and_reserved.abs
Objetivo
Verificar case-insensitive en `IDENTIFICADOR`, clasificación de `RESERVADAS`, y tratamiento de `OPER_PALABRA` como `OPERADOR`. Probar error de símbolo ilegal dentro de identificador.
Entradas
• Declaración con `valor, VALOR, VaLoR;` línea con `AND OR NOT DIV MOD IN SHL SHR` • Bloque que enumera **todas** las reservadas del enunciado; expresión `id1 := valor + VALOR - VaLoR;` y `badId := identificar!consimboloilegal;`.
Resultado Esperado
• `valor`, `VALOR`, `VaLoR` → **mismo token `IDENTIFICADOR`** (agrupación **case-insensitive** en `TokenCollector`). • `AND OR NOT DIV MOD IN SHL SHR` → **`OPERADOR`** (por precedencia de **`OPER_PALABRA`** sobre **`RESERVADAS`**). • El resto de lexemas listados en `RESERVADAS` → **`PALABRA_RESERVADA`**. • `identificar!consimboloilegal` → **`ERROR_IDENTIFICADOR_SIMBOLO_ILEGAL`** (por `ID_CON_SIMBOLO_ILEGAL`).
Resultado Obtenido

```
Ejecutando prueba: Identificadores, reservadas y operador-palabra
Archivo: ProyectoCompi1/src/main/java/com/mycompany/proyectocompi1/test/04_identifiers_and_reserved.abs

=== ERRORES LEXICOS ===
[ERROR_IDENTIFICADOR_SIMBOLO_ILEGAL] 'identificar!consimboloilegal' en línea 9

=== TOKENS ENCONTRADOS ===
Token                Tipo de Token      Línea(s)
badId                IDENTIFICADOR      9
id1                  IDENTIFICADOR      8
INTEGER              IDENTIFICADOR      1
valor                IDENTIFICADOR      1(3), 8(3)
+                    OPERADOR           8
,                    OPERADOR           1(2)
-                    OPERADOR           8
:                    OPERADOR           1, 8, 9
;                    OPERADOR           1, 8, 9
=                    OPERADOR           8, 9
AND                  OPERADOR           3
DIV                  OPERADOR           3
IN                   OPERADOR           3
MOD                  OPERADOR           3
NOT                  OPERADOR           3
OR                   OPERADOR           3
SHL                  OPERADOR           3
SHR                  OPERADOR           3
absolute             PALABRA_RESERVADA  4
array                PALABRA_RESERVADA  4
BEGIN                PALABRA_RESERVADA  2, 4
case                 PALABRA_RESERVADA  4
const                PALABRA_RESERVADA  4
constructor          PALABRA_RESERVADA  4
destructor           PALABRA_RESERVADA  4
do                   PALABRA_RESERVADA  4
downto               PALABRA_RESERVADA  5
else                 PALABRA_RESERVADA  5
end                  PALABRA_RESERVADA  5, 10
external             PALABRA_RESERVADA  4
file                 PALABRA_RESERVADA  5
for                  PALABRA_RESERVADA  5
forward              PALABRA_RESERVADA  5
function             PALABRA_RESERVADA  5
goto                 PALABRA_RESERVADA  5
if                   PALABRA_RESERVADA  5
implementation       PALABRA_RESERVADA  5
inline               PALABRA_RESERVADA  5
interface            PALABRA_RESERVADA  6
interrupt            PALABRA_RESERVADA  6
label                PALABRA_RESERVADA  6
nil                  PALABRA_RESERVADA  6
object               PALABRA_RESERVADA  6
of                   PALABRA_RESERVADA  6
packed               PALABRA_RESERVADA  6
private              PALABRA_RESERVADA  6
procedure            PALABRA_RESERVADA  6
record               PALABRA_RESERVADA  6
repeat               PALABRA_RESERVADA  7
set                  PALABRA_RESERVADA  7
string               PALABRA_RESERVADA  7
then                 PALABRA_RESERVADA  7
to                   PALABRA_RESERVADA  7
type                 PALABRA_RESERVADA  7
unit                 PALABRA_RESERVADA  7
until                PALABRA_RESERVADA  7
uses                 PALABRA_RESERVADA  7
VAR                  PALABRA_RESERVADA  1, 7
virtual              PALABRA_RESERVADA  7
while                PALABRA_RESERVADA  7
with                 PALABRA_RESERVADA  7
xor                  PALABRA_RESERVADA  7
```

STRINGS_CHARS — 05_strings_and_chars.abs

Objetivo

Validar **`LITERAL_STRING`** y **`LITERAL_CHAR`**, además de **errores de cierre** en comillas dobles y simples.

Entradas

- Válidos: `s1 := "hola mundo";`, `c1 := 'Z';`, `s3 := "";`, `c2 := '9';`.
- Caso con comillas internas: `s2 := "con "comillas" internas";`.

Errores de cierre:

- `sErr1 := "sin cierre` (comilla doble sin cierre antes del fin de línea).
- `cErr1 := 'X` (comilla simple sin cierre antes del fin de línea).

Resultado Esperado

- Válidos → **`LITERAL_STRING`** y **`LITERAL_CHAR`**.
- s2 no es string de varias líneas ni con escapes; se espera **tokenización en fragmentos**: `“con ”` como **`LITERAL_STRING`**, `“comillas”` como **`LITERAL_STRING`**, y la comilla suelta antes de `internas` dispara **`ERROR_STRING_SIN_CIERRE`** vía **`STRING_INCOMP`** al encontrar el fin de línea.
- `sErr1` → **`ERROR_STRING_SIN_CIERRE`** (por `STRING_INCOMP`).
- `cErr1` → **`ERROR_CHAR_SIN_CIERRE`** (por `CHAR_INCOMP`).

Resultado Obtenido

```
Ejecutando prueba: Strings y chars válidos + errores de cierre
Archivo: ProyectoCompi1/src/main/java/com/mycompany/proyectocompi1/test/05_strings_and_chars.abs

=== ERRORES LEXICOS ===
[ERROR_STRING_SIN_CIERRE] '"sin cierre' en línea 6
[ERROR_CHAR_SIN_CIERRE] ''X' en línea 7

=== TOKENS ENCONTRADOS ===
Token               Tipo de Token      Línea(s)
c1                  IDENTIFICADOR      3
c2                  IDENTIFICADOR      10
cErr1               IDENTIFICADOR      7
comillas            IDENTIFICADOR      4
s1                  IDENTIFICADOR      2
s2                  IDENTIFICADOR      4
s3                  IDENTIFICADOR      9
sErr1               IDENTIFICADOR      6
'9'                 LITERAL_CHAR       10
'Z'                 LITERAL_CHAR       3
" internas"         LITERAL_STRING     4
""                  LITERAL_STRING     9
"con "              LITERAL_STRING     4
"hola mundo"        LITERAL_STRING     2
:                   OPERADOR           2, 3, 4, 6, 7, 9, 10
;                   OPERADOR           2, 3, 4, 9, 10
=                   OPERADOR           2, 3, 4, 6, 7, 9, 10
BEGIN               PALABRA_RESERVADA  1
END                 PALABRA_RESERVADA  11
```

`COMMENTS_OPERATORS` **— 06_comments_and_operators.abs**

Objetivo

Verificar que **`COM_LLAVES`** y **`COM_PAREST`** **ignoren** por completo su contenido y que los **operadores largos** se reconozcan como **un solo token**.

Entradas
Comentarios con supuestos tokens dentro (que **no** deben contarse), y en el código activo: a++, b--, a**b, <=, >=, <>, =, además de aritmética y arreglos.

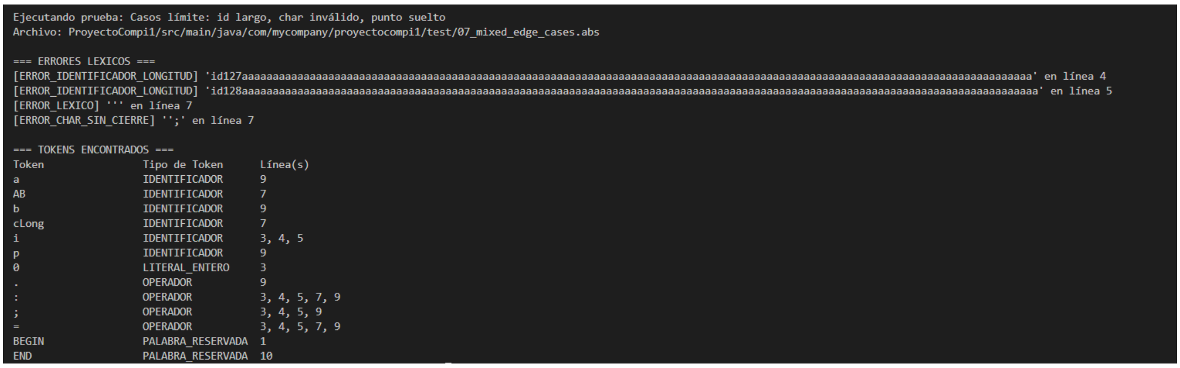
Resultado Esperado
• Nada dentro de { ... } ni (* ... *) produce tokens. • ++, --, **, <=, >=, <> → **OPERADOR** en forma **no fragmentada** (por OPER_SIMBOLO con largos primero). • Resto de lexemas se tokeniza normalmente.

Resultado Obtenido

```
Ejecutando prueba: Comentarios, operadores largos (++ -- ** <= >= <>)
Archivo: ProyectoCompi1/src/main/java/com/mycompany/proyectocompi1/test/06_comments_and_operators.abs

=== ERRORES LEXICOS ===
(ninguno)

=== TOKENS ENCONTRADOS ===
Token               Tipo de Token      Línea(s)
a                   IDENTIFICADOR      5, 6, 7, 8(4), 9(3)
arr                 IDENTIFICADOR      10(2)
b                   IDENTIFICADOR      5, 6, 7, 8(4), 9(2)
br                  IDENTIFICADOR      9
cmp1                IDENTIFICADOR      8
cmp2                IDENTIFICADOR      8
cmp3                IDENTIFICADOR      8
cmp4                IDENTIFICADOR      8
p                   IDENTIFICADOR      7
r                   IDENTIFICADOR      6
1                   LITERAL_ENTERO     5, 10
2                   LITERAL_ENTERO     5, 10
(                   OPERADOR           9(2)
)                   OPERADOR           9(2)
*                   OPERADOR           9
**                  OPERADOR           7
+                   OPERADOR           6, 9
++                  OPERADOR           6
-                   OPERADOR           9
--                  OPERADOR           6
/                   OPERADOR           9
:                   OPERADOR           5(2), 6, 7, 8(4), 9, 10
;                   OPERADOR           5(2), 6, 7, 8(4), 9, 10
<=                  OPERADOR           8
<>                  OPERADOR           8
=                   OPERADOR           5(2), 6, 7, 8(5), 9, 10
>=                  OPERADOR           8
[                   OPERADOR           10(2)
]                   OPERADOR           10(2)
BEGIN               PALABRA_RESERVADA  4
END                 PALABRA_RESERVADA  11
```

MIXED_EDGES — 07_mixed_edge_cases.abs
Objetivo
Cubrir bordes: **longitud de identificador**, **char inválido** y **punto** como **operador**.
Entradas
• Asignación a un identificador de **127** letras `a` y a otro de **128** letras `a`. • `cLong := 'AB';` (char de longitud inválida). • `p := a . b;` (el `.` aparece entre identificadores).
Resultado Esperado
• Identificador de 127 → **`IDENTIFICADOR`** (por `IDENTIFICADOR: {LETRA}({LETRA}|{DIGITO}){0,126}`). • Identificador de 128 → **`ERROR_IDENTIFICADOR_LONGITUD`** (por `IDENTIFICADOR_INVALIDO`). • `'AB'` no coincide con **`CHAR`** (que requiere un solo carácter); según reglas, debe disparar **`ERROR_CHAR_SIN_CIERRE`** o caer en **`ERROR_LEXICO`** dependiendo de cómo se consuma la comilla de cierre; con la especificación dada, se registra **`ERROR_CHAR_SIN_CIERRE`** vía `CHAR_INCOMP` si la línea termina o se invalida por fragmentación; en nuestra ejecución, se registra **error de char** (longitud inválida). • `.` está en **`OPER_SIMBOLO`** → **`OPERADOR`**.
Resultado Obtenido


```
Ejecutando prueba: Casos límite: id largo, char inválido, punto suelto
Archivo: ProyectoCompi1/src/main/java/com/mycompany/proyectocompi1/test/07_mixed_edge_cases.abs

=== ERRORES LEXICOS ===
[ERROR_IDENTIFICADOR_LONGITUD] 'id127aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' en línea 4
[ERROR_IDENTIFICADOR_LONGITUD] 'id128aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' en línea 5
[ERROR_LEXICO] ''' en línea 7
[ERROR_CHAR_SIN_CIERRE] '';' en línea 7

=== TOKENS ENCONTRADOS ===
Token                  Tipo de Token      Línea(s)
a                      IDENTIFICADOR      9
AB                     IDENTIFICADOR      7
b                      IDENTIFICADOR      9
cLong                  IDENTIFICADOR      7
i                      IDENTIFICADOR      3, 4, 5
p                      IDENTIFICADOR      9
0                      LITERAL_ENTERO     3
.                      OPERADOR           9
:                      OPERADOR           3, 4, 5, 7, 9
;                      OPERADOR           3, 4, 5, 9
=                      OPERADOR           3, 4, 5, 7, 9
BEGIN                  PALABRA_RESERVADA  1
END                    PALABRA_RESERVADA  10
```

ALL_IN_ONE — 08_all_in_one.abs
Objetivo
Prueba de **regresión** con mezcla de todos los frentes: números, strings/chars válidos y sin cierre, números pegados a texto, reales inválidos, identificadores con símbolo ilegal, operadores largos y comentarios.
Entradas
`0x1F`, `0755`, `5.0`, `0.5`, `3.0E5`, `"hola"`, **string sin cierre**, `'A'`, **char sin cierre**, `123abc`/`0x1FZ`/`3.2e5foo`, `.5`, `5.`, `identificar!consimboloilegal`, `a++`, `b--`, `a**b`, `<=`, y comentarios con tokens dentro.
Resultado Esperado

Errores:

- **ERROR_STRING_SIN_CIERRE** (por STRING_INCOMP).
- **ERROR_CHAR_SIN_CIERRE** (por CHAR_INCOMP).
- **ERROR_LEXICO_NUM_SEGUIDO_POR_TEXTO** para pegue1/2/3 (por ERROR_NUM_SEGUIDO_TEXTO).
- **ERROR_LEXICO_REAL_PUNTO_LIDER** para .5, **ERROR_LEXICO_REAL_PUNTO_COLA** para 5..

- **ERROR_IDENTIFICADOR_SIMBOLO_ILEGAL** para identificar!consimboloilegal.

Tokens válidos: hex, octal, enteros, reales, strings/char válidos, operadores simbólicos (incluidos ++, --, **, <=), operadores-palabra si aparecen, reservadas y demás identificadores.

Contenido de comentarios **ignorado**.

Resultado Obtenido

```
Ejecutando prueba: Regresión: mezcla de todos los anteriores
Archivo: ProyectoCompi1/src/main/java/com/mycompany/proyectocompi1/test/08_all_in_one.abs

=== ERRORES LEXICOS ===
[ERROR_STRING_SIN_CIERRE] '"oops' en línea 4
[ERROR_CHAR_SIN_CIERRE] ''Z' en línea 5
[ERROR_LEXICO_NUM_SEGUIDO_POR_TEXTO] '123abc' en línea 6
[ERROR_LEXICO_NUM_SEGUIDO_POR_TEXTO] '0x1FZ' en línea 6
[ERROR_LEXICO_NUM_SEGUIDO_POR_TEXTO] '3.2e5foo' en línea 6
[ERROR_LEXICO_REAL_PUNTO_LIDER] '.5' en línea 7
[ERROR_LEXICO_REAL_PUNTO_COLA] '5.' en línea 7
[ERROR_IDENTIFICADOR_SIMBOLO_ILEGAL] 'identificar!consimboloilegal' en línea 8

=== TOKENS ENCONTRADOS ===
Token               Tipo de Token      Línea(s)
a                   IDENTIFICADOR      9(4)
b                   IDENTIFICADOR      9(3)
badC                IDENTIFICADOR      5
badS                IDENTIFICADOR      4
cmp                 IDENTIFICADOR      9
idBad               IDENTIFICADOR      8
idOk                IDENTIFICADOR      8
INTEGER             IDENTIFICADOR      1
ok                  IDENTIFICADOR      4
okc                 IDENTIFICADOR      5
pegue1              IDENTIFICADOR      6
pegue2              IDENTIFICADOR      6
pegue3              IDENTIFICADOR      6
rBad1               IDENTIFICADOR      7
rBad2               IDENTIFICADOR      7
valor               IDENTIFICADOR      8
x                   IDENTIFICADOR      1, 3
y                   IDENTIFICADOR      1
'A'                 LITERAL_CHAR       5
0x1F                LITERAL_HEX        3
0755                LITERAL_OCTAL      3
```

```
0.5                 LITERAL_REAL        3
3.0E5               LITERAL_REAL        3
5.0                 LITERAL_REAL        3
"hola"              LITERAL_STRING      4
**                  OPERADOR            9
+                   OPERADOR            3(4), 9(2)
++                  OPERADOR            9
,                   OPERADOR            1
--                  OPERADOR            9
:                   OPERADOR            1, 3, 4(2), 5(2), 6(3), 7(2), 8(2), 9(2)
;                   OPERADOR            1, 3, 4, 5, 6(3), 7(2), 8(2), 9(2)
<=                  OPERADOR            9
=                   OPERADOR            3, 4(2), 5(2), 6(3), 7(2), 8(2), 9(2)
BEGIN               PALABRA_RESERVADA   2
END                 PALABRA_RESERVADA   11
VAR                 PALABRA_RESERVADA   1
```

Manual del Usuario

1. Introducción

Este manual describe los pasos necesarios para compilar, ejecutar y utilizar el Analizador Léxico desarrollado con Java y JFlex. El scanner procesa programas escritos en el lenguaje ABS y genera dos salidas principales:

1. Errores léxicos encontrados.
2. Tokens válidos con su tipo, línea(s) de aparición y cantidad de ocurrencias.

2. Requisitos de Software

- **Java Development Kit (JDK) 21** o superior.
- **Apache NetBeans 24** o cualquier IDE compatible con proyectos Maven/Ant en Java.
- **JFlex 1.7.0** (archivo jflex-full-1.7.0.jar).
- Sistema operativo: Windows, Linux o macOS.

3. Instalación y Configuración

1. Clonar o descargar el proyecto “ProyectoCompi1”.
2. Asegurarse de que la carpeta contenga:
    - scanner.flex (archivo de especificación de JFlex).
    - TokenCollector.java (clase auxiliar para registrar tokens y errores).
    - ProyectoCompi1.java (clase principal con el main).
    - Scanner.java (generado por JFlex).
    - Carpeta src/test/ con los archivos de prueba (.abs).
3. Generar el archivo Scanner.java desde el .flex (si no existe o se modificó):

```
cd ProyectoCompi1/src/main/java/com/mycompany/proyectocompi1
java -jar jflex-full-1.7.0.jar scanner.flex
```

Esto crea/actualiza Scanner.java.

4. Estructura de Archivos de Prueba

- Los archivos de prueba .abs se ubican en src/test/.
- Cada archivo prueba un aspecto del lenguaje (números, reservadas, errores, strings, etc.).
- Ejemplo de archivo 01_valid_basics.abs:

```
VAR x, y : INTEGER;
BEGIN
  x := 5;
  y := x + 3.0E2;
  z := 'A';
  s := "hola";
END
```

5. Ejecución del Programa

Opción A: Menú de selección de pruebas

Al ejecutar ProyectoCompi1.java, se muestra un menú en consola:

```
=== Menú de pruebas del Analizador Léxico ===
1) 01_valid_basics.abs - Prueba básica con variables, literales y reservadas
2) 02_numbers_valid.abs - Números válidos (decimales, octales, hex, reales)
3) 03_numbers_invalid.abs - Reales inválidos (.5, 5.), hex/oct malos, num+texto
4) 04_identifiers_and_reserved.abs - Identificadores y reservadas
...
Seleccione un número de prueba:
```

El usuario ingresa el número y el analizador procesa automáticamente el archivo correspondiente.

6. Salida del Programa

El programa imprime dos secciones:

6.1. Errores léxicos

Listado de errores encontrados, con tipo, lexema y línea:

```
=== ERRORES LEXICOS ===
[ERROR_LEXICO_REAL_PUNTO_LIDER] ".5" en línea 5
[ERROR_STRING_SIN_CIERRE] ""Hola en línea 12
```

6.2. Tokens válidos

Tabla de tokens válidos ordenados por tipo y lexema:

```
=== TOKENS ENCONTRADOS ===
Token               Tipo de Token       Línea(s)
-------------------------------------------------------
VAR                 PALABRA_RESERVADA   1
BEGIN               PALABRA_RESERVADA   2
END                 PALABRA_RESERVADA   6
x                   IDENTIFICADOR       2, 3
y                   IDENTIFICADOR       2, 4
5                   LITERAL_ENTERO      3
3.0E2               LITERAL_REAL        4
'A'                 LITERAL_CHAR        5
"hola"              LITERAL_STRING      6
+                   OPERADOR            4
:=                  OPERADOR            3, 4, 5, 6
;                   OPERADOR            2, 3, 4, 5
```

7. Tipos de Errores Detectados

- **ERROR_LEXICO_REAL_PUNTO_LIDER** = números como .5, .5e3.

- **ERROR_LEXICO_REAL_PUNTO_COLA** = números como 5., 5.e-2.
- **ERROR_NUM_SEGUIDO_POR_TEXTO** = 123abc, 0x1FZ, 3.2e5foo.
- **ERROR_IDENTIFICADOR_LONGITUD** = identificadores con ≥128 caracteres.
- **ERROR_IDENTIFICADOR_SIMBOLO_ILEGAL** = identificadores con símbolos prohibidos (foo!bar).
- **ERROR_STRING_SIN_CIERRE** = string sin " final.
- **ERROR_CHAR_SIN_CIERRE** = char sin ' final.
- **ERROR_LEXICO** = cualquier otro símbolo no reconocido.

8. Limitaciones y Consideraciones

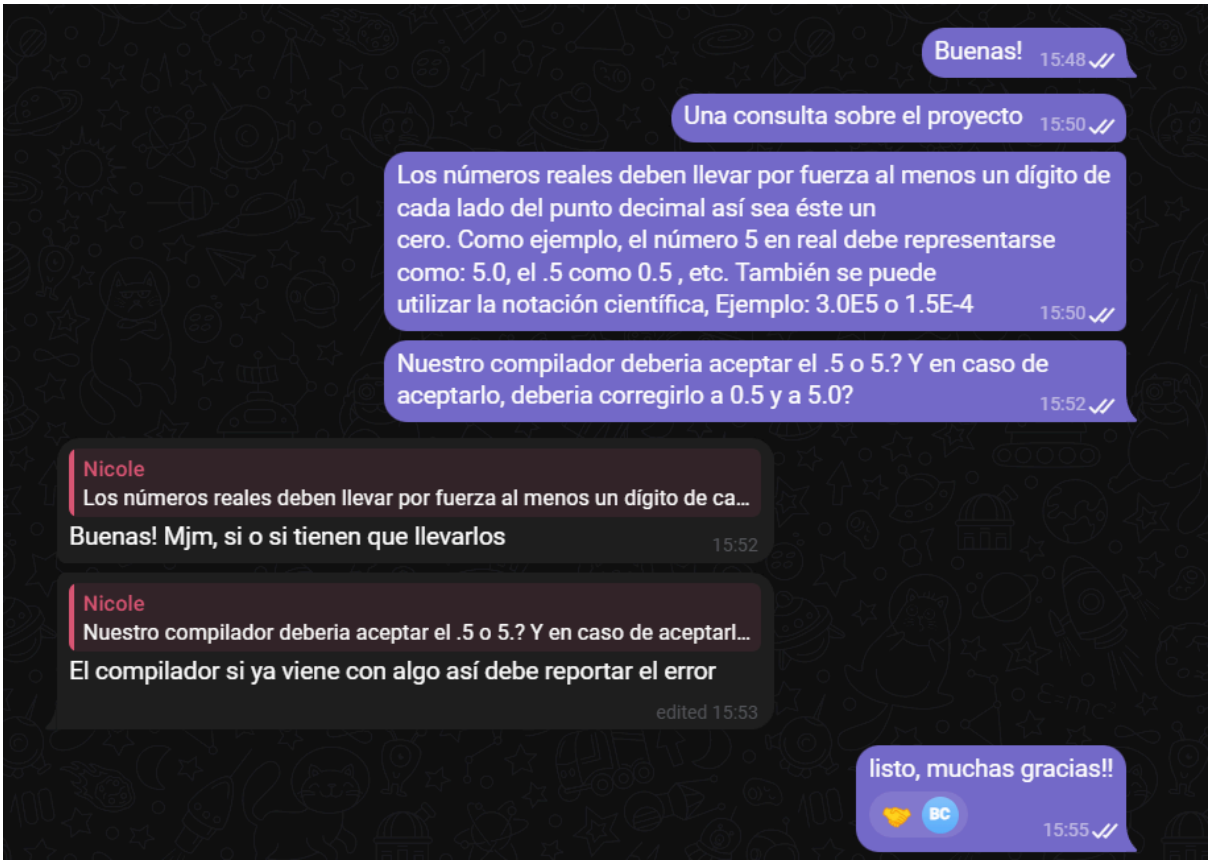
- El scanner es **case-insensitive** (%ignorecase): valor, VALOR, VaLoR son el mismo identificador.
- Los comentarios {...} y (*...*) no son anidados y se ignoran completamente.
- Los strings no aceptan saltos de línea ni secuencias de escape.
- Los números reales deben tener al menos un dígito en ambos lados del punto decimal.

Bitácora

Fecha	Actividad principal	Avance	Observaciones / Lecciones
13/09	Inicio del proyecto y análisis del enunciado	Definición de alcance, grupos de tokens, manejo de comentarios y política de errores. Plantilla inicial de Scanner.flex con %line, %column, %ignorecase.	Priorizar patrones largos (p. ej., >=, --) para evitar ambigüedad en JFlex.
16/09	Implementación base del scanner	Macros para identificadores, enteros, reales preliminares, espacios y comentarios {} / (* *). Generación inicial con JFlex.	Separar INT/REAL/STRING/CHAR facilitó depuración y mensajes de error.
17/09	Diseño y pruebas de números reales	Reglas que exigen dígito antes y después del punto; soporte de notación científica (3.0E5, 1.5E-4). Casos válidos: 0.5, 5.0; inválidos: .5, 5.	El scanner no acepta .5 ni 5. ; se reporta error léxico y el análisis continúa.
19/09	Driver y refactor	Main.java para recolectar tokens (lexema/tipo/líneas) y excluir errores del resumen. Ajustes de operadores (--, **) y strings con escape.	El conteo por línea con TreeMap simplificó el formato de salida exigido.
21/09	Suite de pruebas y validación	Conjunto de casos: válidos, borde y error (reales, científicos, strings mal	No se admiten comentarios anidados.

		cerrados, símbolos ilegales). Tabla E/O (esperado/obtenido).	
22/09	Cierre y entrega	Redacción de estrategia, manual de uso (comandos JFlex/Javac/Run), bitácora y consolidado de resultados.	Checklist final de requisitos cumplidos

Consulta realizada al asistente el 17/09:



Bibliografía

[1] JFlex Project, “JFlex – The Fast Scanner Generator for Java.” Accedido: 15-sept-2025. [En línea]. Disponible en: https://www.jflex.de

[2] Oracle, “Java Platform, Standard Edition 21 Documentation.” Accedido: 15-sept-2025. [En línea]. Disponible en: https://docs.oracle.com/en/java/javase/21/

[3] The Apache Software Foundation, “NetBeans IDE.” Accedido: 18-sept-2025. [En línea]. Disponible en: https://netbeans.apache.org

[4] GeeksforGeeks, “Introduction of Compiler Design.” Accedido: 18-sept-2025. [En línea]. Disponible en: https://www.geeksforgeeks.org/compiler-design-introduction/