Integrantes:

Renata Carolina Castro Olmos

Olimpia de los Ángeles Moctezuma Juan

Carlos Alberto Ureña Andrade

Isaías de Jesús Avilés Rodríguez.

EE:

Lenguajes Formales y Compiladores.

Docente:

Primavera Argüelles Lucho

Documentación técnica:

Lenguaje CarumaLang.

Veracruz, Veracruz a 13 de noviembre del 2025.

Universidad Veracruzana
Facultad de Ingeniería Eléctrica y Electrónica
Región Veracruz

CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE IMÁGENES

1. OBJETIVO DEL LENGUAJE.

El presente documento describe el analizador léxico del lenguaje CarumaLang, desarrollado mediante JavaCC.

Este analizador identifica todos los tokens del lenguaje y detecta los errores léxicos sin detener la ejecución, generando un reporte completo de tokens válidos y errores encontrados.

2. REQUISITOS.

En la Tabla 1 se muestran los requisitos para implementar CarumaLang de forma exitosa.

Requisitos	Descripción
JDK 21	Instalado y con la variable de entorno JAVA_HOME configurada.
JavaCC	Archivo javacc.jar ubicado en la carpeta /lib.
Conocimientos previos	Haber cursado la Unidad 3 de *Lenguajes Formales y Compiladores*.

TABLA 1: REQUISITOS PARA IMPLEMENTAR CARUMALANG.

3. COMPONENTES LÉXICOS.

A continuación, se muestran los componentes léxicos y su explicación con referencia en las estructuras de control presentes en otros lenguajes de programación.

3.1 Tokens.

Entendamos por **tokens** como símbolo abstracto que representa un tipo de unidad léxica; por ejemplo, una palabra clave específica o una secuencia de caracteres de entrada que denotan un identificador.

Palabras Reservadas.

- Caruma - Inicio del programa
- holahola - Instrucción de impresión
- byebye - Fin del programa
- CaeCliente - Condicional IF
- SiNoCae - Else
- papoi - Bucle while

- paraPapoi - Bucle FOR
- stopPlease - Break
- DIOS - Valor booleano TRUE
- DIOSNO - Valor booleano FALSE
- intCHELADA - Tipo entero
- granito - Tipo decimal
- cadena - Tipo string
- caracter - Tipo char

Operadores.

- = - Asignación
- <= - Menor o igual
- >= - Mayor o igual
- == - Igualdad
- > - Mayor que
- < - Menor que
- + - Suma
- - - Resta
- * - Multiplicación
- / - División

Delimitadores.

- (- Paréntesis de apertura
-) - Paréntesis de cierre
- { - Llave de apertura
- } - Llave de cierre
- : - Dos puntos

Identificadores y Literales

- **Identificadores** - Variables/funciones que comienzan con letra seguida de letras o dígitos
- **Numeritos** - Literales numéricos enteros (ej: 42) o decimales (ej: 3.14)
- **TextoLiteral** - Cadenas de texto entre comillas dobles (ej: "Hola Mundo")
- **LetraLiteral** - Caracteres individuales entre comillas simples (ej: 'A')

3.2 Lexemas y patrones.

Un **lexema** es una secuencia de caracteres del código fuente que coincide con un patrón. Un **patrón** es una regla que define las cadenas de caracteres válidas que pueden representar un tipo específico de token.

En la Tabla 2 se describe los Tokens y Patrones implementados en nuestro lenguaje. Además, se muestra el Patrón (RegEx) formado.

CarumaLang

Significado	Token	Ejemplos de Lexemas	Patrón (RegEx informal)
<MAIN>	<Caruma>	"Caruma"	"Caruma"
<PRINT>	<holahola>	"holahola"	"holahola"
<RETURN>	<byebye>	"byebye"	"byebye"
<IF>	<CaeCliente>	"CaeCliente"	"CaeCliente"
<ELSE>	<SiNoCae>	"SiNoCae"	"SiNoCae"
<WHILE>	<papoi>	"papoi"	"papoi"
<FOR>	<paraPapoi>	"paraPapoi"	"paraPapoi"
<BREAK>	<stopPlease>	"stopPlease"	"stopPlease"
<TRUE>	<DIOS>	"DIOS"	"DIOS"
<TYPE_INT>	<intCHELADA>	"intCHELADA"	"intCHELADA"
<TYPE_FLOAT>	<granito>	"granito"	"granito"
<ID>	<mixCHELADA>	"numero", "cliente", "papoi1"	letra (letra | digito)*
<NUM>	<numerito>	"0", "21", "100", "3.14"	digito ('.'digito+)*
<TYPE_STRING>	<cadena>	"cadena"	"cadena"
<CHARACTER>	<caracter>	"caracter"	"caracter"

<STRING_LITERAL>	<TextoLiteral>	"Hola Mundo", "Caruma123"	"\"" (~["\"","\n","\r"])* "\""
<CHAR_LITERAL>	<LetraLiteral>	'a', 'Z', '5'	"'" (~["'","\n","\r"]) "'"
<ASSIGN>	<EstoEs>	"="	"="
<LE>	<MenorIgualitoQue>	"<="	"<="
<GE>	<MayorIgualitoQue>	">="	">="
<EQ>	<Igualito>	"=="	"=="
<GT>	<MayorQue>	">"	">"
<LT>	<MenorQue>	"<"	"<"
<PLUS>	<Poner>	"+"	"+"
<MINUS>	<Quitar>	"-"	"-"
<MULT>	<SaleMas>	"*"	"*"
<DIV>	<SaleMenos>	"/"	"/"
<PAR_OPEN>	<Abriendo>	"("	"("
<PAR_CLOSE>	<Cerrando>	")"	")"
<COLON>	<AhiVa>	":"	":"

TABLA 2: TABLA DE TOKENS, LEXEMAS Y PATRONES.

4. ESTRUCTURA DEL CÓDIGO Y FUNCIONAMIENTO.

4.1Organización.

La Imagen 1 muestra la organización general del proyecto, detallando cómo se distribuyen los archivos esenciales para la generación del analizador léxico.

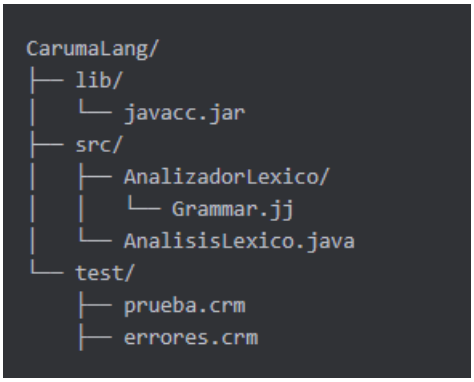

IMAGEN 1: ESTRUCTURA DE CARPETAS DEL PROYECTO.

4.2 Clases y su funcionamiento.

4.1.1 Javacc.jar.

El archivo javacc.jar es el núcleo del compilador JavaCC (Java Compiler Compiler). Su función principal es generar el código fuente Java de un analizador léxico y/o sintáctico a partir de un archivo de especificación .jj

4.1.2 Gramar.jj.

Este archivo define las reglas léxicas del lenguaje. Es procesado por JavaCC para generar automáticamente el código del analizador léxico que usa el archivo AnalizadorLexico.java

Opciones de configuración.

La Imagen 2 presenta los parámetros iniciales definidos para JavaCC.

- Estas configuraciones permiten que el lenguaje distinga entre mayúsculas y minúsculas, otorgando mayor precisión y control de código.
- Evita que las clases generadas no sean estáticas, permitiendo crear múltiples instancias del lexer simultáneamente (útil para procesar varios archivos).
- Genera solo el analizador léxico, omitiendo de momento el sintáctico.

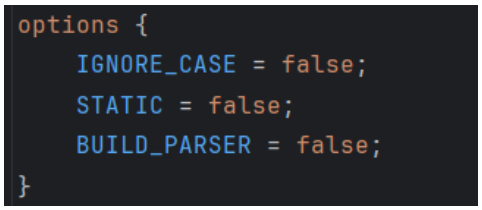

IMAGEN 2: VARIABLES DE CONFIGURACIÓN.

Declaración inicial.

En la Imagen 3 se muestra el bloque donde se establece la clase principal del lexer.

- Define el nombre de la clase principal: CarumaLangLexer
- Establece el paquete: AnalizadorLexico
- JavaCC generará CarumaLangLexerTokenManager.java (el lexer real) y CarumaLangLexerConstants.java (constantes de tokens)

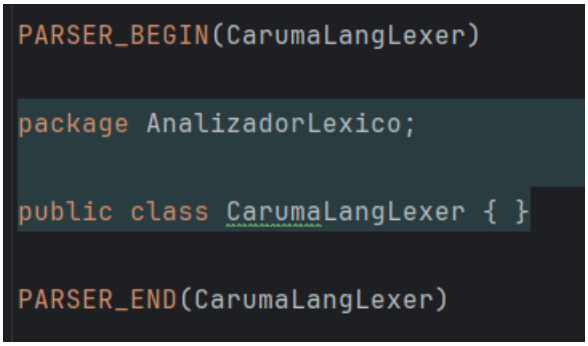

IMAGEN 3: DECLARACIONES INICIALES ANALIZADOR.

Sección SKIP.

La Imagen 4 muestra la definición de los caracteres que el analizador omite durante la lectura (como espacios, tabulaciones o saltos de línea) permitiendo avanzar sin generar tokens.

```
// ------------------------------------------------------
// ------------------ ESPACIOS Y COMENTARIOS ------------
// ------------------------------------------------------

SKIP : {
    " " | "\t" | "\r" | "\n"
}
```

IMAGEN 4: SECCIÓN DE CARACTERES IGNORADOS.

Sección KEYWORDS.

En la Imagen 5 se observa la definición de los tokens que representan las palabras reservadas del lenguaje, estableciendo los elementos básicos que permiten interpretar instrucciones y estructuras.

```
TOKEN : {
    // Estructura del Programa
    < CARUMA : "Caruma" >          // MAIN - Inicio del programa principal
|   < HOLAHOLA : "holahola" >      // PRINT - Instrucción de impresión
|   < BYEBYE : "byebye" >          // RETURN - Retorno/Fin de función

    // Control de Flujo
|   < CAECLIENTE : "CaeCliente" >  // IF - Condicional
|   < SINOCAE : "SiNoCae" >        // ELSE - Condicional alternativo
|   < PAPOI : "papoi" >            // WHILE - Bucle mientras
|   < PARAPAPOI : "paraPapoi" >    // FOR - Bucle para
|   < STOPPLEASE : "stopPlease" >  // BREAK - Romper bucle

    // Valores Booleanos
|   < DIOS : "DIOS" >              // TRUE - Valor booleano verdadero
|   < DIOSNO : "DIOSNO" >          // FALSE - Valor booleano falso

    // Tipos de Datos
|   < INTCHELADA : "intCHELADA" >  // TYPE_INT - Tipo entero
|   < GRANITO : "granito" >        // TYPE_FLOAT - Tipo decimal
|   < CADENA : "cadena" >          // TYPE_STRING - Tipo cadena
|   < CARACTER : "caracter" >      // CHARACTER - Tipo carácter
}
```

IMAGEN 5: TOKENS EMPLEADOS EN CARUMALANG.

Sección OPERADORES.

La Imagen 6 presenta el conjunto de operadores válidos, abarcando símbolos de asignación, comparación y operaciones aritméticas utilizados durante el análisis expresiones.

```
// ---------------------------------------------------
// --------------------- OPERADORES ------------------
// ---------------------------------------------------

TOKEN : {
    // Asignación
    < ESTOES : "=" >                    // ASSIGN - "="

    // Operadores Relacionales
|   < MENORIGUALITOQUE : "<=" >         // LE - "<="
|   < MAYORIGUALITOQUE : ">=" >         // GE - ">="
|   < IGUALITO : "==" >                 // EQ - "=="
|   < MAYORQUE : ">" >                  // GT - ">"
|   < MENORQUE : "<" >                  // LT - "<"

    // Operadores Aritméticos
|   < PONER : "+" >                     // PLUS - "+"
|   < QUITAR : "-" >                    // MINUS - "-"
|   < SALEMAS : "*" >                   // MULT - "*"
|   < SALEMENOS : "/" >                 // DIV - "/"
}
```

IMAGEN 6: OPERADORES UTILIZADOS.

Sección DELIMITADORES.

La Imagen 7 muestra los delimitadores empleados para estructurar bloques, parámetros y separadores dentro del lenguaje.

```
// ---------------------------------------------------
// -------------------- DELIMITADORES ----------------
// ---------------------------------------------------

TOKEN : {
    < ABRIENDO : "(" >      // PAR_OPEN - "("
|   < CERRANDO : ")" >      // PAR_CLOSE - ")"
|   < OPEN : "{" >          // BRACKET_OPEN - "{"
|   < CLOSE : "}" >         // BRACKET_CLOSE - "}"
|   < AHIVA : ":" >         // COLON - ":"
}
```

IMAGEN 7: DELIMITADORES, AGRUPADORES Y MARCADORES.

Sección IDENTIFICADORES y DELIMITADORES.

En la Imagen 8 se detallan las expresiones que definen la forma válida de identificadores, números, caracteres y cadenas, estableciendo las restricciones sintácticas del lenguaje.

```
// ---------------------------------------------------
// --------------- IDENTIFICADORES Y LITERALES ------------
// ---------------------------------------------------

TOKEN : {
    // ID - Identificadores (variables/funciones)
    // Patrón: letra (letra | dígito)*
    < MIXCHELADA : (["a"-"z", "A"-"Z"]) (["a"-"z", "A"-"Z", "0"-"9"])* >

    // NUM - Literales numéricos (enteros y decimales)
    // Patrón: dígito+ ('.' dígito+)?
|   < NUMERITO : (["0"-"9"])+ ( "." (["0"-"9"])+ )? >

    // STRING_LITERAL - Cadenas de texto entre comillas dobles
    // Patrón: "\"" (~["\"","\n","\r"])* "\""
|   < TEXTOLITERAL : "\"" ( ~["\"","\n","\r"] )* "\"" >

    // CHAR_LITERAL - Caracteres individuales entre comillas simples
    // Patrón: "'" (~["'","\n","\r"]) "'"
|   < LETRALITERAL : "'" ( ~["'","\n","\r"] ) "'" >
}
```

IMAGEN 8: REGLAS DEL LENGUAJE.

Sección ERRORES LEXICOS.

La Imagen 9 muestra la regla que clasifica cualquier carácter inválido como un token INVALID.

- JavaCC intenta hacer coincidir el input con todos los tokens definidos.
- Si ningún token coincide, crea un token `INVALID`.
- El programa Java puede detectar este token y reportarlo como error.

```
// ---------------------------------------------------------
// ---------------------- ERRORES LÉXICOS ------------------
// ---------------------------------------------------------

// Cualquier carácter no reconocido genera error
TOKEN : {
    < INVALID : ~[] >
}
```

IMAGEN 9: CARACTERES NO RECONOCIDOS.

4.1.3 AnalizadorLexico.java

Este programa lee un archivo .crm, la analiza carácter por carácter, y se separa el contenido de tokens (palabras clave, identificadores, operadores, etc) e identificando los errores léxicos.

ErrorLexico.

La Imagen 10 presenta la estructura encargada de almacenar información detallada sobre los errores detectados durante el proceso de análisis.

```
// Clase para almacenar información de errores
static class ErrorLexico {  4 usages   olimpia
    String mensaje;  1 usage
    int linea;  2 usages
    int columna;  2 usages
    String caracterInvalido;  2 usages
```

IMAGEN 10: CLASE PARA LOS ERRORES LEXICOS.

Main().

Dentro del método Main() se encuentran encapsulados las siguientes funcionalidades

- En la Imagen 11 se muestra el fragmento que gestiona la apertura y selección de archivos .crm mediante un cuadro de diálogo.

```
JFileChooser fileChooser = new JFileChooser();
FileNameExtensionFilter filter = new FileNameExtensionFilter( description: "Archivo CarumaLang",  ...extensions: "crm");
fileChooser.setFileFilter(filter);
int returnValue = fileChooser.showOpenDialog( parent: null);
```

IMAGEN 11: APERTURA DE ARCHIVO

- La Imagen 12 incluye la lógica encargada de verificar que el archivo tenga la extensión y permisos adecuados antes de iniciar el análisis.

```
if (returnValue == JFileChooser.APPROVE_OPTION) {
    String fileName = fileChooser.getSelectedFile().getAbsolutePath();

    if (!fileName.toLowerCase().endsWith(".crm")) {
        System.err.println("Error: El archivo seleccionado, no tiene una extension .crm");
        return;
    }

    try {
        analizarArchivo(fileName);
    } catch (FileNotFoundException e) {
        System.err.println("Error: No se pudo encontrar el archivo: " + fileName);
    } catch (IOException e) {
        System.err.println("Error al leer el archivo: " + e.getMessage());
    }
} else {
    System.out.println("No se seleccionó ningún archivo.");
}
}
```

IMAGEN 12: MANEJO DE ERRORES EN EL ARCHIVO.

AnalizarArchivo().

- La Imagen 13 muestra el código donde se lee el contenido del archivo línea por línea, proporcionando una interfaz de caracteres para JavaCC e inicializa el analizador léxico generado por JavaCC reconociendo los tokens.

```
BufferedReader reader = new BufferedReader(new FileReader(fileName));
SimpleCharStream stream = new SimpleCharStream(reader);
CarumaLangLexerTokenManager lexer = new CarumaLangLexerTokenManager(stream);
```

IMAGEN 13: LECTURA DEL ARCHIVO.

- En la Imagen 14 se observa la clasificación inicial de los elementos leídos, separando los tokens válidos de los errores encontrados.

```
List<Token> tokensValidos = new ArrayList<>();
List<ErrorLexico> errores = new ArrayList<>();
```

IMAGEN 14: SEPARACIÓN DE TOKENS.

- La Imagen 15 ilustra el ciclo que obtiene cada token, verifica si es el final del archivo y determina su categoría correspondiente.

```
boolean continuar = true;
while (continuar) {
    try {
        Token token = lexer.getNextToken();

        if (token.kind == CarumaLangLexerConstants.EOF) {
            continuar = false;
```

IMAGEN 15: BUCLE PRINCIPAL PARA EL ANÁLISIS.

- La Imagen 16 muestra el procedimiento para identificar tokens marcados como inválidos y registrarlos para su posterior reporte. Cuando JavaCC encuentra un carácter que no coincide con ninguna regla léxica definida, lo marca como INVALID. El programa registra el error, pero continúa analizando el resto del archivo.

```
} else if (token.kind == CarumaLangLexerConstants.INVALID) {
    // Token INVALID reconocido - tratarlo como error pero continuar
    String caracterInvalido = token.image;
    String mensaje = "Carácter no reconocido: '" + caracterInvalido +
                     "' (ASCII: " + (int)caracterInvalido.charAt(0) + ")";

    errores.add(new ErrorLexico(mensaje, token.beginLine, token.beginColumn, caracterInvalido));

    System.out.printf("ERROR | %-35s | Carácter inválido              | Línea: %d, Col: %d%n",
        caracterInvalido,
        token.beginLine,
        token.beginColumn);
```

IMAGEN 16: VERIFICACIÓN DE TOKENS VALIDOS.

- En la Imagen 17 se presenta la estructura que recopila y organiza los tokens válidos, junto con su contenido y su ubicación dentro del archivo Tokens válidos.

```
} else {
    tokensValidos.add(token);
    String tokenName = CarumaLangLexerConstants.tokenImage[token.kind];
    System.out.printf("%-5d | %-35s | %-30s | Línea: %d, Col: %d%n",
        tokensValidos.size(),
        token.image,
        tokenName,
        token.beginLine,
        token.beginColumn);
}
```

IMAGEN 17: MANEJO DE ERRORES.

- La Imagen 18 muestra el bloque que actúa como mecanismo de seguridad ante errores graves, evitando que el análisis se detenga inesperadamente. En casos extremos donde el lexer no puede crear ni siquiera un token INVALID, este catch actúa como red de seguridad. Extrae información del error e intenta avanzar un carácter para continuar el análisis.

```
} catch (TokenMgrError e) {
    // Capturar información del error (backup por si el token INVALID falla)
    String mensaje = e.getMessage();
    int linea = stream.getEndLine();
    int columna = stream.getEndColumn();

    // Extraer el carácter problemático del mensaje de error
    String caracterInvalido = "?";
    if (mensaje.contains("Encountered: \"")) {
        int start = mensaje.indexOf("Encountered: \"") + 14;
        int end = mensaje.indexOf( str: "\"", start);
        if (end > start) {
            caracterInvalido = mensaje.substring(start, end);
        }
    }

    errores.add(new ErrorLexico(mensaje, linea, columna, caracterInvalido));

    System.out.printf("ERROR | %-20s | Error léxico           | Línea: %d, Col: %d%n", caracterInvalido, linea, columna);

    // Intentar recuperarse: avanzar un carácter
    try {
        stream.readChar();
    } catch (IOException ioException) {
        continuar = false;
    }
}
```

IMAGEN 18: REPORTE DE ERRORES.

- En la Imagen 19 se muestra el formato final de presentación de errores, donde se resume la información clave de cada carácter inválido encontrado. Estos errores se organizan en forma de tabla y muestra la ubicación exacta de cada error.

```java
// Mostrar tabla de errores si los hay
if (!errores.isEmpty()) {
    System.out.println("\n----------------------------------------");
    System.out.println("     ERRORES LÉXICOS ENCONTRADOS        ");
    System.out.println("----------------------------------------");
    System.out.println();
    System.out.println("-------------------------------------------");
    System.out.println("| No. | Carácter    | Línea  | Columna |");
    System.out.println("-------------------------------------------");

    for (int i = 0; i < errores.size(); i++) {
        ErrorLexico error = errores.get(i);
        String caracterMostrar = error.caracterInvalido;
        if (caracterMostrar.equals("\n")) caracterMostrar = "\\n";
        if (caracterMostrar.equals("\t")) caracterMostrar = "\\t";
        if (caracterMostrar.equals("\r")) caracterMostrar = "\\r";

        System.out.printf("| %-4d | %-11s | %-6d | %-7d |%n",
            i + 1,
            caracterMostrar,
            error.linea,
            error.columna);
    }
    System.out.println("-------------------------------------------");
}
```

IMAGEN 19: TABLA DE ERRORES.

- La Imagen 20 presenta el resultado general del proceso, indicando si el archivo cumple o no con las reglas léxicas del lenguaje.

```
// Resumen final
System.out.println("\n--------------------------------------");
System.out.println("        RESUMEN DEL ANÁLISIS           ");
System.out.println("--------------------------------------");
System.out.println();
System.out.println("Tokens válidos reconocidos: " + tokensValidos.size());
System.out.println("Errores léxicos encontrados: " + errores.size());
System.out.println();

if (errores.isEmpty()) {
    System.out.println("Análisis léxico completado SIN ERRORES");
    System.out.println("El archivo cumple con la sintaxis léxica de CarumaLang");
} else {
    System.out.println("Análisis completado CON ERRORES");
    System.out.println("Se encontraron " + errores.size() + " caracteres no reconocidos");
    System.out.println("Revise la tabla de errores para más detalles");
}

System.out.println("\n======================================");

reader.close();
```

IMAGEN 20: RESUMEN FINAL.

generarArchivoTokens().

Una vez completado el análisis léxico y la identificación de todos los tokens y errores, el programa genera un archivo de salida con extensión .tokens que contiene un registro completo y ordenado del análisis realizado. Este archivo sirve como documentación del proceso de análisis y puede ser utilizado por fases posteriores del compilador.

Al final del método analizarArchivo(), después de completar el análisis y mostrar el resumen en consola, se invoca la función generarArchivoTokens() pasándole los siguientes parámetros, como se muestra en la Imagen 21:

```
// Generar archivo de tokens
generarArchivoTokens(fileName, tokensValidos, errores);
```

IMAGEN 21: INVOCACIÓN DE GENERARARCHIVOTOKENS.

- fileName: Ruta del archivo .crm que fue analizado
- tokensValidos: Lista de tokens válidos identificados durante el análisis
- errores: Lista de errores léxicos encontrados durante el análisis

Esta función es responsable de crear el archivo de salida con toda la información del análisis léxico. En la Imagen 22, Imagen 23 e Imagen 24 se presenta el código completo de la implementación:

```java
/**
 * Genera un archivo .tokens con la información del análisis léxico
 *
 * @param archivoFuente Ruta del archivo .crm analizado
 * @param tokensValidos Lista de tokens válidos encontrados
 * @param errores Lista de errores léxicos encontrados
 */
private static void generarArchivoTokens(String archivoFuente,
                                         List<Token> tokensValidos,
                                         List<ErrorLexico> errores) {
    try {
        // 1. Crear nombre del archivo de salida
        String nombreSalida = archivoFuente.replace(".crm", ".tokens");

        // 2. Obtener fecha y hora actual
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String fechaActual = sdf.format(new Date());

        // 3. Crear lista unificada de elementos (tokens + errores) ordenada
        List<ElementoAnalisis> elementos = new ArrayList<>();

        // Agregar tokens válidos
        for (int i = 0; i < tokensValidos.size(); i++) {
            elementos.add(new ElementoAnalisis(i + 1, tokensValidos.get(i)));
        }

        // Agregar errores
        for (ErrorLexico error : errores) {
            elementos.add(new ElementoAnalisis(error));
        }
```

IMAGEN 22: FUNCIÓN GENERARARCHIVOTOKENS PARTE 1.

```java
// Ordenar por línea y columna
Collections.sort(elementos, new Comparator<ElementoAnalisis>() {
    @Override
    public int compare(ElementoAnalisis e1, ElementoAnalisis e2) {
        if (e1.linea != e2.linea) {
            return Integer.compare(e1.linea, e2.linea);
        }
        return Integer.compare(e1.columna, e2.columna);
    }
});

// 4. Escribir archivo
try (BufferedWriter writer = new BufferedWriter(new FileWriter(nombreSalida))) {

    // ===== ENCABEZADO =====
    writer.write("# ARCHIVO DE TOKENS - CARUMALANG");
    writer.newLine();
    writer.write("# Archivo fuente: " + archivoFuente);
    writer.newLine();
    writer.write("# Fecha generacion: " + fechaActual);
    writer.newLine();
    writer.write("# Tokens validos: " + tokensValidos.size());
    writer.newLine();
    writer.write("# Errores lexicos: " + errores.size());
    writer.newLine();
    writer.newLine();

    // ===== SECCION DE TOKENS =====
    writer.write("[TOKENS]");
    writer.newLine();
```

IMAGEN 23: FUNCIÓN GENERARARCHIVOTOKENS PARTE 2.

```
        // Contador para tokens válidos (para mantener numeración correcta)
        int contadorTokens = 1;

        for (ElementoAnalisis elem : elementos) {
            if (elem.esError) {
                // Escribir error
                writer.write(String.format("ERROR|%s|%s|%d|%d|%s",
                    elem.lexema,
                    elem.tipoToken,
                    elem.linea,
                    elem.columna,
                    elem.mensajeError));
            } else {
                // Escribir token válido
                writer.write(String.format("%d|%s|%s|%d|%d|VALIDO",
                    contadorTokens++,
                    elem.lexema,
                    elem.tipoToken,
                    elem.linea,
                    elem.columna));
            }
            writer.newLine();
        }

        writer.newLine();

        // ===== SECCION DE RESUMEN =====
        writer.write("[RESUMEN]");
        writer.newLine();
        writer.write("TOKENS_VALIDOS=" + tokensValidos.size());
        writer.newLine();
        writer.write("ERRORES_LEXICOS=" + errores.size());
        writer.newLine();

        String estado = errores.isEmpty() ? "SIN_ERRORES" : "CON_ERRORES";
        writer.write("ESTADO=" + estado);
        writer.newLine();
        writer.newLine();

        // ===== FIN =====
        writer.write("[FIN]");
        writer.newLine();
```

IMAGEN 24: FUNCIÓN GENERARARCHIVOTOKENS PARTE 3.

El funcionamiento de la función es de la siguiente manera:

Paso 1: Creación del nombre del archivo: Se genera el nombre del archivo de salida reemplazando la extensión .crm por .tokens.

Paso 2: Obtención de fecha y hora: Se obtiene la fecha y hora actual del sistema para incluirla en el encabezado del archivo.

Paso 3: Unificación y ordenamiento: Se crea una lista que combina tanto los tokens válidos como los errores encontrados. Esta lista se ordena por línea y columna para mantener el orden en que aparecen en el código fuente.

Paso 4: Escritura del archivo: Se escribe el archivo con la siguiente estructura:

- Encabezado: Información general (nombre del archivo, fecha, contadores)
- Sección [TOKENS]: Lista completa de tokens y errores en orden
- Sección [RESUMEN]: Estadísticas finales del análisis
- Sección [FIN]: Marcador de fin de archivo

Paso 5: Confirmación: Se muestra un mensaje en consola confirmando la generación exitosa del archivo o informando de cualquier error.

Algunas ventajas de la generación de archivo de tokens son las siguientes:

- Persistencia: Los resultados del análisis se guardan de forma permanente.
- Trazabilidad: Permite revisar los tokens identificados sin re-ejecutar el analizador.
- Debugging: Facilita la depuración del analizador léxico y sintáctico.
- Integración: El archivo puede ser consumido por otras fases del compilador.
- Documentación: Sirve como evidencia del proceso de compilación.
- Ordenamiento: Los tokens y errores aparecen en el orden en que se encuentran en el código fuente.