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

ÍNDICE GENERAL.

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 presente tabla 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 presente tabla que describe los Tokens y Patrones implementados en nuestro lenguaje. Además, se muestra el Patrón (RegEx) fomrado

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.

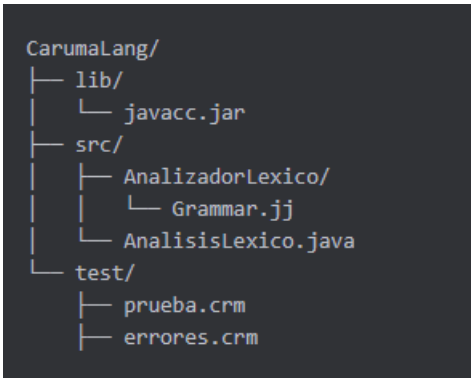

IMAGEN 1: ESTRUCTURA DE CARPETAS DEL PROYECTO.

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.

- 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.

```
options {
    IGNORE_CASE = false;
    STATIC = false;
    BUILD_PARSER = false;
}
```

IMAGEN 2: VARIABLES DE CONFIGURACIÓN.

Declaración inicial.

- 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)

```
PARSER_BEGIN(CarumaLangLexer)

package AnalizadorLexico;

public class CarumaLangLexer { }

PARSER_END(CarumaLangLexer)
```

IMAGEN 3: DECLARACIONES INICIALES ANALIZADOR.

Sección SKIP.

Ignora estos caracteres, no genera tokens y los salta automáticamente.

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

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

IMAGEN 4: SECCIÓN DE CARACTERES IGNORADOS.

Sección KEYWORDS.

Es todo el conjunto de palabras reservada que construyen el lenguaje.

```
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.

Son todo el conjunto de operadores que acepta el lenguaje. Estos realizan operaciones aritméticas, de comparación y asignación.

```
// ----------------------------------------------------------
// ---------------------- 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.

Es todo conjunto de tokens que agrupa parámetros de funciones, bloques de código y operadores para posibles bucles o estructuras.

```
// ---------------------------------------------------------
// ---------------------- 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.

Restringe las reglas del lenguaje, dictando como declara variables y tipos de datos. Emplea un símil de expresión regular.

```
// ---------------------------------------------------------
// ---------------- 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.

Este token tiene la menor prioridad. Significa lo siguiente: Cualquier carácter que NO esté en el conjunto vacío.

- 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.

Clase interna dedicada al almacenaje de la información detallada de cada error encontrado durante el 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

- Selección de Archivos: Abre un diálogo para que el usuario seleccione un archivo. Filtra para mostrar solo archivos .crm.

```
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

- Validación del archivo: Mediante una seri de if's verficamos que el archivo tenga la extensión adecuada y que se pueda leer.

```
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().

- Inicialización del lexer: Leemos el archivo línea por línea, proporcionamos una interfaz de caracteres para JavaCC e inicializamos 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.

- Separamos los tokens correctos de errores para mostrarlos posteriormente.

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

IMAGEN 14: SEPARACIÓN DE TOKENS.

- Bucle principal: En este bucle principal se obtienen los tokens del archivo, verifica si llegó al final (EOF) y clasifica el token en tres categorías.

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

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

IMAGEN 15: BUCLE PRINCIPAL PARA EL ANÁLISIS.

- Tokens Inválidos: 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.

- Tokens válidos: Se muestra una tabla con el número de tokens, contenido textual, tipo de token y la ubicación (línea y columna).

```
} 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.

- Manejo de errores: En casos extremos donde el lexer no puede crear ni siquiera un token INVALID, este cath actúa como red de seguridad. Extrae información del error, intenta avanzar un carácter para continuar el análisis y previene que el programa se detenga completamente (cumple con el requisito propuesto en clase)

```
} 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.

- Reporte de errores: Tabla organizada de todos los errores. Convierte caracteres invisibles a representación visible y muestra la ubicación exacta de cada error.

```
// 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.

- Resumen final: Muestra de forma clara sobre si el archivo cumple con la sintaxis léxica 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.