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 18 de diciembre del 2025.

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.

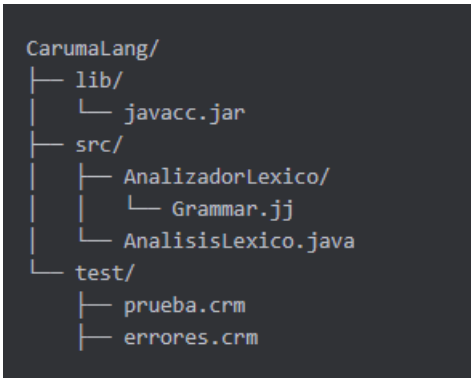


IMAGEN 1: ESTRUCTURA DE CARPETAS DEL PROYECTO.

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.

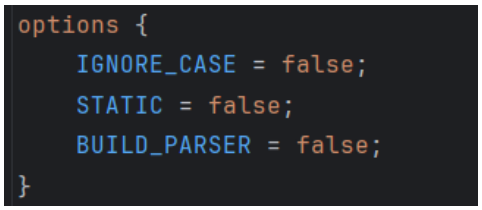


IMAGEN 2: VARIABLES DE CONFIGURACIÓN.

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)

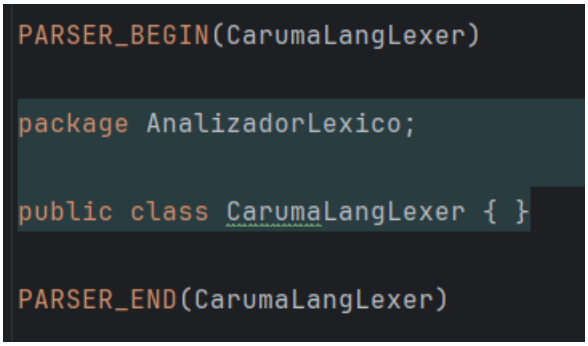


IMAGEN 3: DECLARACIONES INICIALES ANALIZADOR.

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.

5. ANALIZADOR SINTÁCTICO

El analizador sintáctico de CarumaLang es el encargado de verificar que la secuencia de tokens generada por el analizador léxico cumpla con las reglas gramaticales del lenguaje. Esta fase valida el orden y la estructura de las instrucciones.

5.1 Estructura del código y funcionamiento

El análisis sintáctico se implementa principalmente a través de dos archivos clave generados y gestionados mediante JavaCC.

5.1.1 Grammar.jj (Definición Sintáctica)

Este archivo contiene la lógica central del parser. A diferencia de la fase léxica, aquí se definen las producciones gramaticales y la jerarquía de las estructuras.

Funcionamiento clave:

- **PARSER_BEGIN / PARSER_END:** Define la clase CarumaLangParser.
- **Producciones:** Métodos como void Programa(), void Declaracion(), etc., que representan los no-terminales de la gramática.
- **Consumo de Tokens:** Utiliza las definiciones léxicas (como <INTCHELADA>, <MIXCHELADA>) para validar la entrada.

5.1.2 AnalisisSintactico.java (Ejecución y Modo Pánico)

Esta clase extiende la funcionalidad del parser generado para agregar una gestión robusta de errores. En lugar de detenerse al primer error, implementa una estrategia de recuperación.

Componentes principales:

- **ParserConRecuperacion:** Una clase interna que hereda de CarumaLangParser.
- **recuperarHastaToken():** Método clave para el "Modo Pánico". Si encuentra un error, consume tokens hasta encontrar uno seguro (como un punto y coma o cierre de llave) para continuar analizando.
- **Lista de Errores:** Almacena todos los ErrorSintactico encontrados para mostrarlos al final, en lugar de abortar la compilación.

5.2 Clases y su funcionamiento: Grammar.jj

El archivo Grammar.jj es el componente central del analizador sintáctico. Este archivo contiene la especificación formal de la gramática de CarumaLang y es procesado por JavaCC para generar el parser (analizador sintáctico) en Java. A continuación, se detalla el propósito y funcionamiento de cada una de sus secciones.

5.2.1 Configuración y Definición del Parser

Al inicio del archivo se encuentran las opciones de configuración y la definición de la clase principal del parser.

La sección options establece parámetros cruciales para la generación del código:

- IGNORE_CASE = false: Indica que el lenguaje distingue entre mayúsculas y minúsculas (case-sensitive).
- STATIC = false: Permite crear múltiples instancias del parser, lo cual es útil si se necesita analizar varios archivos simultáneamente o en diferentes hilos.

- BUILD_PARSER = true: Instruye a JavaCC para que genere los archivos .java correspondientes al analizador sintáctico (CarumaLangParser.java, etc.).

Posteriormente, los bloques PARSER_BEGIN y PARSER_END delimitan la definición de la clase Java que encapsulará el parser. Aquí se define el paquete AnalizadorSintactico y se incluye un método main básico para pruebas iniciales.

En la **Imagen 25** se muestra el bloque de configuración y definición de la clase del parser:

```
1  // ==========================================
2  // ANALIZADOR SINTÁCTICO - CARUMALANG
3  // Grammar.jj - Parser LL(1) Descendente Recursivo
4  // ==========================================
5
6  options {
7      IGNORE_CASE = false;
8      STATIC = false;
9      BUILD_PARSER = true;  // IMPORTANTE: true para análisis sintáctico
10 }
11
12 PARSER_BEGIN(CarumaLangParser)
13
14 package AnalizadorSintactico;
15
16 public class CarumaLangParser {
17     public static void main(String[] args) {
18         System.out.println("Analizador Sintáctico CarumaLang - Parser LL(1)");
19     }
20 }
21
22 PARSER_END(CarumaLangParser)
```

IMAGEN 25: CONFIGURACIÓN DE OPCIONES Y DEFINICIÓN DE LA CLASE CARUMALANGPARSER

5.2.2 Análisis Léxico: SKIP y Tokens

Aunque este archivo está enfocado en la sintaxis, también define las reglas léxicas que el parser utilizará para consumir la entrada.

Sección SKIP: Esta sección especifica los caracteres que el analizador debe ignorar y no considerar como tokens significativos. En CarumaLang, se ignoran los espacios en blanco, tabulaciones (\t), retornos de carro (\r) y saltos de línea (\n). Esto permite que el programador formatee su código libremente sin afectar la lógica del programa.

En la **Imagen 26** se detalla la sección SKIP:

```
28  SKIP : {
29      " "
30  |   "\t"
31  |   "\r"
32  |   "\n"
33  }
```

IMAGEN 26: SECCIÓN SKIP: CARACTERES IGNORADOS POR EL ANALIZADOR.

Sección TOKEN (Palabras Reservadas): Aquí se definen los tokens correspondientes a las palabras clave del lenguaje. Cada token tiene un nombre simbólico (como CARUMA, CAECLIENTE) y su representación literal en el código fuente. Esta sección cubre la estructura del programa, control de flujo, valores booleanos y tipos de datos.

En la **Imagen 27** se observan las definiciones de las palabras reservadas:

```
39  TOKEN : {
40      // Estructura del Programa
41      < CARUMA : "Caruma" >
42  |   < HOLAHOLA : "holahola" >
43  |   < BYEBYE : "byebye" >
44
45      // Control de Flujo
46  |   < CAECLIENTE : "CaeCliente" >
47  |   < SINOCAE : "SiNoCae" >
48  |   < PAPOI : "papoi" >
49  |   < PARAPAPOI : "paraPapoi" >
50  |   < STOPPLEASE : "stopPlease" >
51
52      // Valores Booleanos
53  |   < DIOS : "DIOS" >
54  |   < DIOSNO : "DIOSNO" >
55
56      // Tipos de Datos
57  |   < INTCHELADA : "intCHELADA" >
58  |   < GRANITO : "granito" >
59  |   < CADENA : "cadena" >
60  |   < CARACTER : "caracter" >
61  }
```

IMAGEN 27: DEFINICIÓN DE TOKENS: PALABRAS RESERVADAS DEL SISTEMA.

Sección TOKEN (Operadores y Delimitadores): A continuación, se definen los operadores aritméticos, relacionales y de asignación, así como los delimitadores (paréntesis, llaves, comas, etc.). Estos símbolos son fundamentales para construir expresiones y definir bloques de código.

En la **Imagen 28** se muestra la definición de operadores y delimitadores:

```
63  // ------------------------------------------------------
64  // --------------------- OPERADORES ---------------------
65  // ------------------------------------------------------
66
67  TOKEN : {
68      // Asignación
69      < ESTOES : "=" >
70
71      // Operadores Relacionales
72  |   < MENORIGUALITOQUE : "<=" >
73  |   < MAYORIGUALITOQUE : ">=" >
74  |   < IGUALITO : "==" >
75  |   < MAYORQUE : ">" >
76  |   < MENORQUE : "<" >
77
78      // Operadores Aritméticos
79  |   < PONER : "+" >
80  |   < QUITAR : "-" >
81  |   < SALEMAS : "*" >
82  |   < SALEMENOS : "/" >
83  }
84
85  // ------------------------------------------------------
86  // -------------------- DELIMITADORES -------------------
87  // ------------------------------------------------------
88
89  TOKEN : {
90      < ABRIENDO : "(" >
91  |   < CERRANDO : ")" >
92  |   < OPEN : "{" >
93  |   < CLOSE : "}" >
94  |   < AHIVA : ":" >
95  |   < COMA : "," >
96  }
```

IMAGEN 28: DEFINICIÓN DE TOKENS: OPERADORES Y DELIMITADORES.

Sección TOKEN (Identificadores y Literales): Finalmente, se definen los tokens dinámicos mediante expresiones regulares.

- MIXCHELADA: Para identificadores de variables y funciones.
- NUMERITO: Para números enteros y decimales.
- TEXTOLITERAL: Para cadenas de texto entre comillas dobles.
- LETRALITERAL: Para caracteres individuales entre comillas simples.

En la **Imagen 29** se presentan los patrones para identificadores y literales:

```
102  TOKEN : {
103      < MIXCHELADA : (["a"-"z", "A"-"Z"]) (["a"-"z", "A"-"Z", "0"-"9"])* >
104  |   < NUMERITO : (["0"-"9"])+ ( "." (["0"-"9"])+ )? >
105  |   < TEXTOLITERAL : "\"" ( ~["\"","\n","\r"] )* "\"" >
106  |   < LETRALITERAL : "'" ( ~["'","\n","\r"] ) "'" >
107  }
```

IMAGEN 29: DEFINICIÓN DE TOKENS: IDENTIFICADORES Y LITERALES DINÁMICOS.

5.2.3 Gramática Sintáctica: Estructura General

A partir de este punto comienza la definición de la Gramática Libre de Contexto (GLC) que describe la estructura del lenguaje.

Programa y Declaraciones: La regla Programa() es el punto de entrada (símbolo inicial) de la gramática. Establece que todo archivo debe comenzar con el token CARUMA, seguido de una serie de Declaraciones, terminar con BYEBYE y finalmente el fin de archivo (EOF).

La regla Declaraciones() permite una secuencia de cero o más instrucciones. Cada Declaracion() puede ser una declaración de variable, una asignación, una estructura de control o una impresión. Se utiliza LOOKAHEAD(2) en Declaracion() para resolver la ambigüedad entre una declaración de variable (que empieza con un tipo) y una asignación (que empieza con un identificador), ya que ambos podrían parecer similares al analizador con solo un token de anticipación.

En la **Imagen 30** se muestra la gramática para la estructura general del programa:

```
113   // ---------------------------
114   // 1. PROGRAMA COMPLETO
115   // ---------------------------
116
117   void Programa() : {}
118   {
119       <CARUMA>
120       Declaraciones()
121       <BYEBYE>
122       <EOF>
123   }
124
125   // ---------------------------
126   // 2. DECLARACIONES
127   // ---------------------------
128
129   void Declaraciones() : {}
130   {
131       (Declaracion())*
132   }
133
134   void Declaracion() : {}
135   {
136       LOOKAHEAD(2)
137       DeclaracionVariable()
138   |   Asignacion()
139   |   EstructuraControl()
140   |   Impresion()
141   }
```

IMAGEN 30: GRAMÁTICA SINTÁCTICA: ESTRUCTURA GENERAL DEL PROGRAMA.

5.2.4 Declaración de Variables y Tipos

Estas reglas definen cómo se declaran las variables.

- DeclaracionVariable(): Estructura una declaración como: Tipo + Lista de Identificadores + (Opcional) Inicialización.
- Tipo(): Define los tipos de datos válidos (intCHELADA, granito, etc.).
- ListaIdentificadores(): Permite declarar múltiples variables separadas por comas.
- InicializacionOpt(): Maneja la asignación inicial de valores si está presente.

En la **Imagen 31** se detalla la gramática para la declaración de variables:

```
void DeclaracionVariable() : {}
{
    Tipo() ListaIdentificadores() [InicializacionOpt()]
}

void Tipo() : {}
{
    <INTCHELADA>
|   <GRANITO>
|   <CADENA>
|   <CARACTER>
}

void ListaIdentificadores() : {}
{
    <MIXCHELADA> (<COMA> <MIXCHELADA>)*
}

void InicializacionOpt() : {}
{
    <ESTOES> ListaExpresiones()
}

void ListaExpresiones() : {}
{
    Expresion() (<COMA> Expresion())*
}
```

IMAGEN 31: GRAMÁTICA SINTÁCTICA: DECLARACIÓN DE VARIABLES Y TIPOS DE DATOS.

5.2.5 Estructuras de Control (If, While, For)

Esta sección define la sintaxis para el flujo de control del programa.

- EstructuraControl(): Actúa como un despachador hacia EstructuraIf, EstructuraWhile o EstructuraFor.
- EstructuraIf(): Define la sintaxis del condicional CaeCliente, incluyendo la condición entre paréntesis, el bloque de código entre llaves y la parte opcional ElseOpt (SiNoCae).
- EstructuraWhile(): Define el bucle papoi, similar al while tradicional.
- EstructuraFor(): Define el bucle paraPapoi, con sus tres partes (inicialización, condición, incremento) separadas por el token AHIVA (:). La regla Inicializacion()

dentro del for también usa LOOKAHEAD(2) para distinguir entre declarar una variable nueva o usar una existente.

En la **Imagen 32** se observa la gramática para las estructuras de control:

```
// ----------------------------------------------------
// 5. ESTRUCTURAS DE CONTROL
// ----------------------------------------------------

void EstructuraControl() : {}
{
    EstructuraIf()
|   EstructuraWhile()
|   EstructuraFor()
}

// ----------------------------------------------------
// 5.1 ESTRUCTURA IF
// ----------------------------------------------------

void EstructuraIf() : {}
{
    <CAECLIENTE> <ABRIENDO> Condicion() <CERRANDO>
    <OPEN> Declaraciones() <CLOSE>
    [ElseOpt()]
}

void ElseOpt() : {}
{
    <SINOCAE> <OPEN> Declaraciones() <CLOSE>
}

// ----------------------------------------------------
// 5.2 ESTRUCTURA WHILE
// ----------------------------------------------------

void EstructuraWhile() : {}
{
    <PAPOI> <ABRIENDO> Condicion() <CERRANDO>
    <OPEN> Declaraciones() <CLOSE>
}

// ----------------------------------------------------
// 5.3 ESTRUCTURA FOR
// ----------------------------------------------------

void EstructuraFor() : {}
{
    <PARAPAPOI> <ABRIENDO> Inicializacion() <AHIVA>
    Condicion() <AHIVA> Incremento() <CERRANDO>
    <OPEN> Declaraciones() <CLOSE>
}

void Inicializacion() : {}
{
    LOOKAHEAD(2)
    Tipo() <MIXCHELADA> <ESTOES> Expresion()
|   <MIXCHELADA> <ESTOES> Expresion()
}

void Incremento() : {}
{
    <MIXCHELADA> <ESTOES> Expresion()
}
```

IMAGEN 32: GRAMÁTICA SINTÁCTICA: ESTRUCTURAS DE CONTROL (IF, WHILE, FOR).

5.2.6 Condiciones y Expresiones Lógicas

Estas reglas permiten construir condiciones complejas. La jerarquía Condicion -> ExpresionRelacional -> Expresion asegura la correcta evaluación. Se definen operadores relacionales (<, >, ==, etc.) y lógicos (DIOS, DIOSNO).

En la **Imagen 33** se muestra la gramática para condiciones y operadores relacionales:

```
244   // ------------------------------------------------------
245   // 6. CONDICIONES
246   // ------------------------------------------------------
247
248   void Condicion() : {}
249   {
250       ExpresionRelacional() (OperadorLogico() ExpresionRelacional())*
251   }
252
253   void ExpresionRelacional() : {}
254   {
255       Expresion() OperadorRelacional() Expresion()
256   }
257
258   void OperadorRelacional() : {}
259   {
260       <MENORQUE>
261   |   <MAYORQUE>
262   |   <MENORIGUALITOQUE>
263   |   <MAYORIGUALITOQUE>
264   |   <IGUALITO>
265   }
266
267   void OperadorLogico() : {}
268   {
269       <DIOS>
270   |   <DIOSNO>
271   }
```

IMAGEN 33: GRAMÁTICA SINTÁCTICA: CONDICIONES Y OPERADORES RELACIONALES.

5.2.7 Expresiones Aritméticas

Para manejar las operaciones matemáticas respetando la precedencia de operadores, la gramática se estructura en niveles:

- Expresion(): Maneja sumas y restas (PONER, QUITAR).
- Termino(): Maneja multiplicaciones y divisiones (SALEMAS, SALEMENOS), que tienen mayor precedencia.
- Factor(): Representa la unidad más básica (números, variables, paréntesis) y tiene la mayor precedencia.

Esta estructura jerárquica garantiza que 2 + 3 * 4 se interprete correctamente como 2 + (3 * 4).

En la **Imagen 34** se detalla la gramática para expresiones aritméticas:

```
273   // ---------------------------------------------------------
274   // 7. EXPRESIONES ARITMÉTICAS
275   // ---------------------------------------------------------
276
277   void Expresion() : {}
278   {
279       Termino() ((<PONER> | <QUITAR>) Termino())*
280   }
281
282   void Termino() : {}
283   {
284       Factor() ((<SALEMAS> | <SALEMENOS>) Factor())*
285   }
286
287   void Factor() : {}
288   {
289       <NUMERITO>
290   |   <TEXTOLITERAL>
291   |   <LETRALITERAL>
292   |   <MIXCHELADA>
293   |   <DIOS>
294   |   <DIOSNO>
295   |   <ABRIENDO> Expresion() <CERRANDO>
296   }
```

IMAGEN 34: GRAMÁTICA SINTÁCTICA: EXPRESIONES ARITMÉTICAS Y PRECEDENCIA.

5.3 Clases y su funcionamiento: AnalisisSintactico.java

El archivo AnalisisSintactico.java actúa como el controlador principal del analizador. A diferencia de los archivos generados automáticamente por JavaCC, esta clase es escrita manualmente para implementar la lógica de "Modo Pánico", gestión de archivos y reporte de errores amigables para el usuario. Extiende la funcionalidad básica del parser para que no se detenga ante el primer fallo.

5.3.1 Estructura de Datos para Errores

Para gestionar los errores sin interrumpir la compilación, se define una clase interna estática ErrorSintactico. Esta estructura permite almacenar la información detallada de cada fallo encontrado (mensaje, ubicación, token encontrado y esperado) para su posterior reporte en una tabla.

Además, se define la clase ParserConRecuperacion, que hereda de la clase generada CarumaLangParser. Esta clase contenedora mantiene una lista de errores y un contador para evitar bucles infinitos de reportes.

En la **Imagen 35** se muestra la definición de la clase de error y la extensión del parser:

```
15  /**
16   * Analizador Sintáctico con Modo Pánico
17   * Detecta TODOS los errores sintácticos sin detenerse
18   */
19  public class AnalisisSintactico {
20
21      // Clase para almacenar información de errores sintácticos
22      static class ErrorSintactico {
23          String mensaje;
24          int linea;
25          int columna;
26          String tokenEncontrado;
27          String tokenEsperado;
28
29          ErrorSintactico(String mensaje, int linea, int columna, String tokenEncontrado, String tokenEsperado) {
30              this.mensaje = mensaje;
31              this.linea = linea;
32              this.columna = columna;
33              this.tokenEncontrado = tokenEncontrado;
34              this.tokenEsperado = tokenEsperado;
35          }
36      }
37
38      // Parser personalizado con recuperación de errores
39      static class ParserConRecuperacion extends CarumaLangParser {
40          private List<ErrorSintactico> errores = new ArrayList<>();
41          private int contadorErrores = 0;
42          private static final int MAX_ERRORES = 50;
```

IMAGEN 35: CLASE ERRORSINTACTICO Y EXTENSIÓN DEL PARSER.

5.3.2 Implementación del Modo Pánico

El núcleo de la recuperación de errores reside en dos métodos clave: verificarYConsumirToken y recuperarHastaToken.

- verificarYConsumirToken(): Comprueba si el siguiente token es el esperado. Si no lo es, registra un error pero permite que el análisis continúe, en lugar de lanzar una excepción fatal.
- recuperarHastaToken(): Es el mecanismo de sincronización. Cuando el parser se "pierde" debido a un error de sintaxis, este método descarta tokens de entrada uno por uno hasta encontrar un "token de sincronización" seguro (como un punto y coma,

una llave de cierre o una palabra reservada de inicio de sentencia), permitiendo que el parser se realinee.

En la **Imagen 36** se detallan los métodos de recuperación y sincronización:

```java
    /**
     * Verifica y consume un token esperado
     */
    private boolean verificarYConsumirToken(int tipoEsperado, String nombreToken) {
        Token tok = getToken(1);
        if (tok.kind == tipoEsperado) {
            getNextToken();
            return true;
        } else {
            registrarError("Se esperaba '" + nombreToken + "'",
                           tok.beginLine, tok.beginColumn,
                           tok.image, nombreToken);
            return false;
        }
    }

    /**
     * Recupera hasta encontrar un token de sincronización
     */
    private void recuperarHastaToken(int... tokensSincronizacion) {
        Token tok = getToken(1);

        while (tok.kind != EOF && contadorErrores < MAX_ERRORES) {
            for (int tokenSinc : tokensSincronizacion) {
                if (tok.kind == tokenSinc) {
                    return;
                }
            }
            getNextToken();
            tok = getToken(1);
        }
    }
```

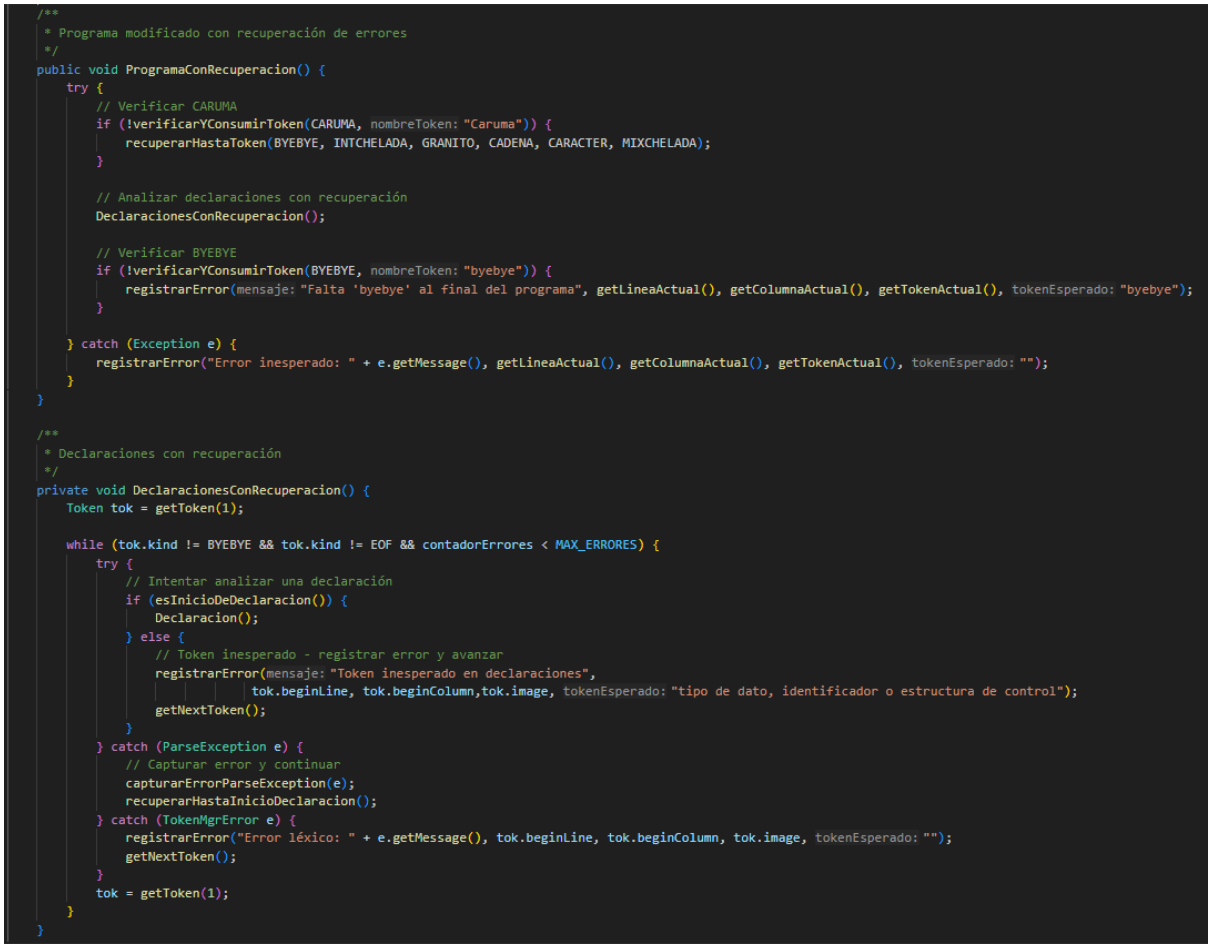
IMAGEN 36: MÉTODOS DE RECUPERACIÓN EN MODO PÁNICO.

5.3.3 Reglas con Recuperación

Para aplicar la estrategia de recuperación, se reescriben los puntos de entrada de la gramática. Métodos como ProgramaConRecuperacion y DeclaracionesConRecuperacion envuelven las llamadas a las reglas gramaticales originales dentro de bloques try-catch.

Si ocurre una ParseException (lanzada por el parser generado) o un TokenMgrError (error léxico), estos métodos capturan la excepción, registran el error utilizando capturarErrorParseException, y luego invocan recuperarHastaInicioDeclaracion para saltar hasta la siguiente instrucción válida y continuar el análisis.

En la **Imagen 37** se observa cómo se envuelven las reglas gramaticales para manejar excepciones:



```java
/**
 * Programa modificado con recuperación de errores
 */
public void ProgramaConRecuperacion() {
    try {
        // Verificar CARUMA
        if (!verificarYConsumirToken(CARUMA, nombreToken: "Caruma")) {
            recuperarHastaToken(BYEBYE, INTCHELADA, GRANITO, CADENA, CARACTER, MIXCHELADA);
        }

        // Analizar declaraciones con recuperación
        DeclaracionesConRecuperacion();

        // Verificar BYEBYE
        if (!verificarYConsumirToken(BYEBYE, nombreToken: "byebye")) {
            registrarError(mensaje: "Falta 'byebye' al final del programa", getLineaActual(), getColumnaActual(), getTokenActual(), tokenEsperado: "byebye");
        }

    } catch (Exception e) {
        registrarError("Error inesperado: " + e.getMessage(), getLineaActual(), getColumnaActual(), getTokenActual(), tokenEsperado: "");
    }
}

/**
 * Declaraciones con recuperación
 */
private void DeclaracionesConRecuperacion() {
    Token tok = getToken(1);

    while (tok.kind != BYEBYE && tok.kind != EOF && contadorErrores < MAX_ERRORES) {
        try {
            // Intentar analizar una declaración
            if (esInicioDeDeclaracion()) {
                Declaracion();
            } else {
                // Token inesperado - registrar error y avanzar
                registrarError(mensaje: "Token inesperado en declaraciones",
                               tok.beginLine, tok.beginColumn,tok.image, tokenEsperado: "tipo de dato, identificador o estructura de control");
                getNextToken();
            }
        } catch (ParseException e) {
            // Capturar error y continuar
            capturarErrorParseException(e);
            recuperarHastaInicioDeclaracion();
        } catch (TokenMgrError e) {
            registrarError("Error léxico: " + e.getMessage(), tok.beginLine, tok.beginColumn, tok.image, tokenEsperado: "");
            getNextToken();
        }
        tok = getToken(1);
    }
}
```

IMAGEN 37: IMPLEMENTACIÓN DE REGLAS CON RECUPERACIÓN DE ERRORES.

5.3.4 Procesamiento de Excepciones

El parser generado por JavaCC lanza excepciones genéricas que son difíciles de entender para un usuario final. El método capturarErrorParseException descompone estas excepciones para extraer información útil, como la lista de tokens que se esperaban en ese punto. El método auxiliar extraerTokensEsperados formatea esta lista para generar mensajes de error claros como "Se esperaba 'hola' o 'adiós'".

En la **Imagen 38** se muestra la lógica para procesar y traducir las excepciones técnicas:

```java
/**
 * Captura y procesa ParseException
 */
private void capturarErrorParseException(ParseException e) {
    String mensaje = e.getMessage();
    Token tokError = e.currentToken.next;
    int linea = tokError.beginLine;
    int columna = tokError.beginColumn;
    String tokenEncontrado = tokError.image;

    // Extraer tokens esperados
    String esperado = extraerTokensEsperados(e);

    registrarError(mensaje, linea, columna, tokenEncontrado, esperado);
}

/**
 * Extrae tokens esperados del ParseException
 */
private String extraerTokensEsperados(ParseException e) {
    StringBuilder esperados = new StringBuilder();

    if (e.expectedTokenSequences != null && e.expectedTokenSequences.length > 0) {
        for (int i = 0; i < e.expectedTokenSequences.length && i < 5; i++) {
            int[] secuencia = e.expectedTokenSequences[i];
            if (secuencia.length > 0) {
                String tokenImg = e.tokenImage[secuencia[0]];
                if (esperados.length() > 0) {
                    esperados.append(str: ", ");
                }
                esperados.append(tokenImg);
            }
        }
    }

    return esperados.length() > 0 ? esperados.toString() : "token válido";
}
```

IMAGEN 38: CAPTURA Y PROCESAMIENTO DE EXCEPCIONES DE PARSEO.

5.3.5 Ejecución y Pre-análisis

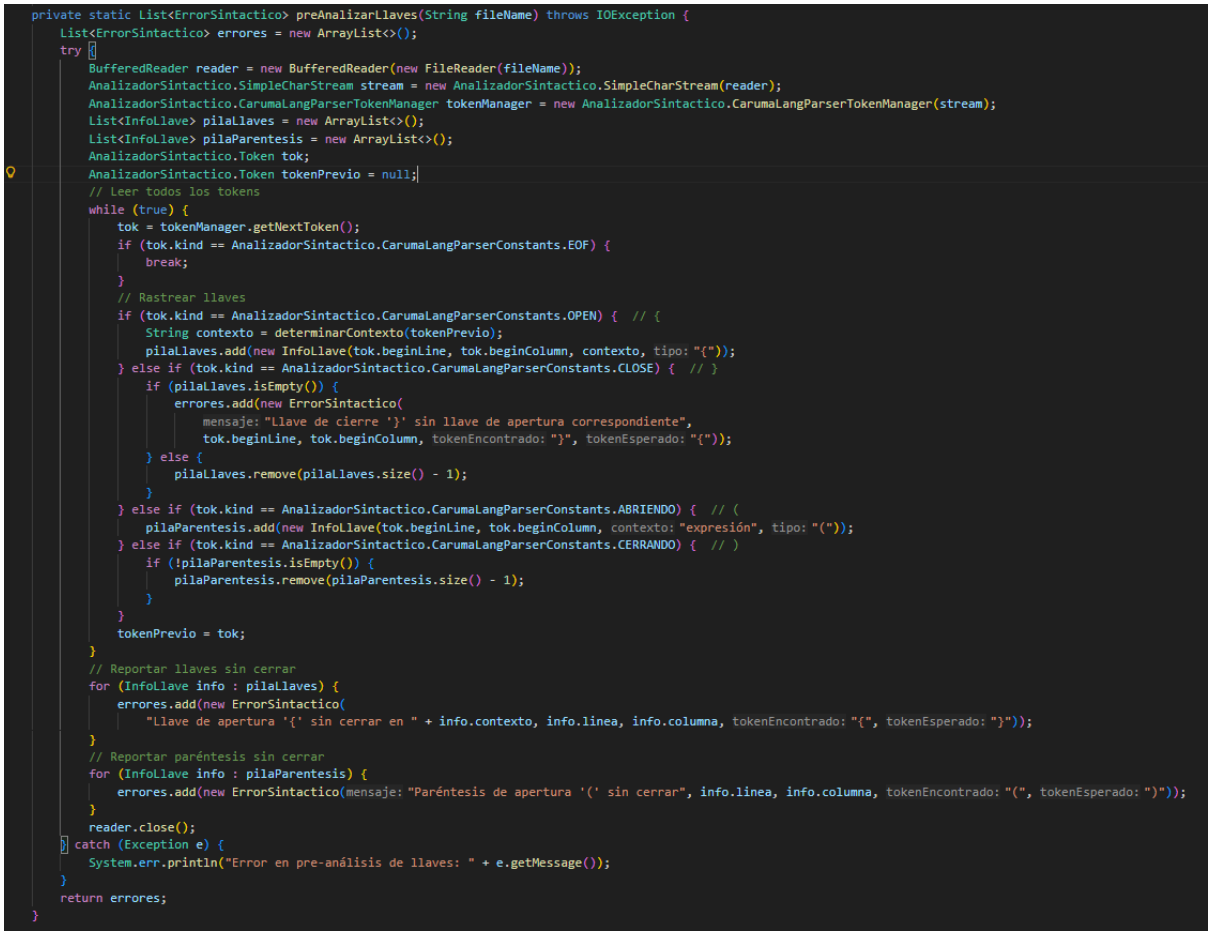
El método main y analizarArchivo orquestan todo el proceso. Una característica distintiva de esta implementación es el uso de un **Pre-análisis de Llaves** (preAnalizarLlaves).

Antes de ejecutar el análisis sintáctico completo (que es LL(1)), el sistema hace una lectura rápida del archivo para verificar únicamente el balanceo de delimitadores ({, }, (,)). Esto es crucial porque en un parser recursivo, una llave faltante puede causar errores en cascada que confunden al resto del análisis. Este paso preliminar detecta y reporta desbalanceos de bloques antes de intentar analizar la lógica interna.

En la **Imagen 39** se presenta el flujo principal y en la **Imagen 40** se presenta el pre-análisis:

```java
private static void analizarArchivo(String fileName) throws IOException {
    System.out.println(x: "========================================");
    System.out.println(x: "   ANALIZADOR SINTACTICO - CARUMALANG");
    System.out.println(x: "     MODO PANICO - TODOS LOS ERRORES");
    System.out.println(x: "========================================");
    System.out.println("Archivo: " + fileName + "\n");

    // PASO 1: Pre-análisis para detectar llaves sin cerrar
    List<ErrorSintactico> erroresLlaves = preAnalizarLlaves(fileName);

    // PASO 2: Análisis sintáctico normal
    BufferedReader reader = new BufferedReader(new FileReader(fileName));
    ParserConRecuperacion parser = new ParserConRecuperacion(reader);

    System.out.println(x: "Iniciando analisis sintactico en modo panico...\n");

    // Ejecutar análisis con recuperación de errores
    parser.ProgramaConRecuperacion();

    List<ErrorSintactico> errores = parser.getErrores();

    // PASO 3: Combinar errores de llaves con errores sintácticos
    errores.addAll(erroresLlaves);

    // Mostrar resultados
    if (errores.isEmpty()) {
        System.out.println(x: "========================================");
        System.out.println(x: "         ANALISIS EXITOSO");
        System.out.println(x: "========================================");
        System.out.println(x: "\nEl programa cumple con la sintaxis de CarumaLang");
        System.out.println(x: "No se encontraron errores sintacticos");
        System.out.println(x: "\n========================================");
    } else {
        mostrarErrores(errores);
    }

    // Generar archivo de errores
    generarArchivoErrores(fileName, errores);

    reader.close();
}
```

IMAGEN 39: GESTIÓN DE ARCHIVOS.



```
private static List<ErrorSintactico> preAnalizarLlaves(String fileName) throws IOException {
    List<ErrorSintactico> errores = new ArrayList<>();
    try {
        BufferedReader reader = new BufferedReader(new FileReader(fileName));
        AnalizadorSintactico.SimpleCharStream stream = new AnalizadorSintactico.SimpleCharStream(reader);
        AnalizadorSintactico.CarumaLangParserTokenManager tokenManager = new AnalizadorSintactico.CarumaLangParserTokenManager(stream);
        List<InfoLlave> pilaLlaves = new ArrayList<>();
        List<InfoLlave> pilaParentesis = new ArrayList<>();
        AnalizadorSintactico.Token tok;
        AnalizadorSintactico.Token tokenPrevio = null;
        // Leer todos los tokens
        while (true) {
            tok = tokenManager.getNextToken();
            if (tok.kind == AnalizadorSintactico.CarumaLangParserConstants.EOF) {
                break;
            }
            // Rastrear llaves
            if (tok.kind == AnalizadorSintactico.CarumaLangParserConstants.OPEN) {  // {
                String contexto = determinarContexto(tokenPrevio);
                pilaLlaves.add(new InfoLlave(tok.beginLine, tok.beginColumn, contexto, tipo: "{"));
            } else if (tok.kind == AnalizadorSintactico.CarumaLangParserConstants.CLOSE) {  // }
                if (pilaLlaves.isEmpty()) {
                    errores.add(new ErrorSintactico(
                        mensaje: "Llave de cierre '}' sin llave de apertura correspondiente",
                        tok.beginLine, tok.beginColumn, tokenEncontrado: "}", tokenEsperado: "{"));
                } else {
                    pilaLlaves.remove(pilaLlaves.size() - 1);
                }
            } else if (tok.kind == AnalizadorSintactico.CarumaLangParserConstants.ABRIENDO) {  // (
                pilaParentesis.add(new InfoLlave(tok.beginLine, tok.beginColumn, contexto: "expresión", tipo: "("));
            } else if (tok.kind == AnalizadorSintactico.CarumaLangParserConstants.CERRANDO) {  // )
                if (!pilaParentesis.isEmpty()) {
                    pilaParentesis.remove(pilaParentesis.size() - 1);
                }
            }
            tokenPrevio = tok;
        }
        // Reportar llaves sin cerrar
        for (InfoLlave info : pilaLlaves) {
            errores.add(new ErrorSintactico(
                "Llave de apertura '{' sin cerrar en " + info.contexto, info.linea, info.columna, tokenEncontrado: "{", tokenEsperado: "}"));
        }
        // Reportar paréntesis sin cerrar
        for (InfoLlave info : pilaParentesis) {
            errores.add(new ErrorSintactico(mensaje: "Paréntesis de apertura '(' sin cerrar", info.linea, info.columna, tokenEncontrado: "(", tokenEsperado: ")"));
        }
        reader.close();
    } catch (Exception e) {
        System.err.println("Error en pre-análisis de llaves: " + e.getMessage());
    }
    return errores;
}
```

IMAGEN 40: PRE-ANÁLISIS DE DELIMITADORES.

6. GRAMÁTICAS

En esta sección se establece la definición rigurosa del lenguaje CarumaLang desde una perspectiva teórica-matemática. Se describe la estructura del lenguaje mediante la notación formal de una gramática *G = (N, T, P, S)*, donde se detallan explícitamente los conjuntos de símbolos No Terminales, Terminales (tokens), las reglas de Producción y el Símbolo inicial. Esta especificación sirve como la base fundamental sobre la cual se construye la sintaxis del lenguaje, independientemente de su implementación tecnológica.

1. Gramática para Programa Completo

Gramática G_programa

P → Caruma D byebye EOF

D → S D | ε

S → DV | A | EC | H

Desglose G = (N, T, P, S)

- **N** = {P, D, S, DV, A, EC, H}
- **T** = {Caruma, byebye, EOF, intCHELADA, granito, cadena, caracter, MIXCHELADA, CaeCliente, SiNoCae, papoi, paraPapoi, holahola, (,), {, }, :, ,, =, +, -, *, /, <, >, <=, >=, ==, DIOS, DIOSNO, NUMERITO, TEXTOLITERAL, LETRALITERAL}
- **P** = {
    - P → Caruma D byebye EOF
    - D → S D
    - D → ε
    - S → DV
    - S → A
    - S → EC
    - S → H }
- **S** = P

---

2. Gramática para Declaración de Variables

Gramática G_declaracion

DV → tipo LI

DV → tipo LI IO

IO → = LE

LI → MIXCHELADA

LI → MIXCHELADA , LI

LE → E

LE → E , LE

tipo → intCHELADA | granito | cadena | caracter

E → T

E → T + T

E → T - T

T → F

T → F * F

T → F / F

F → NUMERITO | TEXTOLITERAL | LETRALITERAL | MIXCHELADA | DIOS | DIOSNO

F → (E)

Desglose G = (N, T, P, S)

- **N** = {DV, IO, LI, LE, tipo, E, T, F}
- **T** = {intCHELADA, granito, cadena, caracter, MIXCHELADA, =, ,, +, -, *, /, (,), NUMERITO, TEXTOLITERAL, LETRALITERAL, DIOS, DIOSNO}
- **P** = {
    - DV → tipo LI
    - DV → tipo LI IO
    - IO → = LE
    - LI → MIXCHELADA
    - LI → MIXCHELADA , LI
    - LE → E
    - LE → E , LE
    - tipo → intCHELADA
    - tipo → granito
    - tipo → cadena
    - tipo → caracter
    - E → T
    - E → T + T
    - E → T - T
    - T → F
    - T → F * F

    - T → F / F
    - F → NUMERITO
    - F → TEXTOLITERAL
    - F → LETRALITERAL
    - F → MIXCHELADA
    - F → DIOS
    - F → DIOSNO
    - F → (E) }
- **S** = DV

Ejemplos válidos:

intCHELADA x

intCHELADA x, y, z

granito pi = 3.14

cadena nombre, apellido = "Juan", "Pérez"

---

3. Gramática para Asignación

Gramática G_asignacion

A → MIXCHELADA = E

E → T

E → T + T

E → T - T

T → F

T → F * F

T → F / F

F → NUMERITO | TEXTOLITERAL | LETRALITERAL | MIXCHELADA | DIOS | DIOSNO

F → (E)

Desglose G = (N, T, P, S)

- **N** = {A, E, T, F}
- **T** = {MIXCHELADA, =, +, -, *, /, (,), NUMERITO, TEXTOLITERAL, LETRALITERAL, DIOS, DIOSNO}
- **P** = {
    - A → MIXCHELADA = E

    - E → T
    - E → T + T
    - E → T - T
    - T → F
    - T → F * F
    - T → F / F
    - F → NUMERITO
    - F → TEXTOLITERAL
    - F → LETRALITERAL
    - F → MIXCHELADA
    - F → DIOS
    - F → DIOSNO
    - F → (E) }
- **S** = A

Ejemplos válidos:

x = 10

resultado = a + b * c

nombre = "CarumaLang"

activo = DIOS

---

4. Gramática para Estructuras de Control

Gramática G_estructuras

EC → I | W | FOR

Desglose G = (N, T, P, S)

- **N** = {EC, I, W, FOR}
- **T** = {CaeCliente, SiNoCae, papoi, paraPapoi}
- **P** = {
    - EC → I
    - EC → W
    - EC → FOR }
- **S** = EC

5. Gramática para IF (CaeCliente)

Gramática G_if

I → CaeCliente (C) { D }

I → CaeCliente (C) { D } EO

EO → SiNoCae { D }

C → ER

C → ER OL ER

ER → E OR E

OR → < | > | <= | >= | ==

OL → DIOS | DIOSNO

D → S D | ε

S → DV | A | EC | H

E → T

E → T + T

E → T - T

T → F

T → F * F

T → F / F

F → NUMERITO | TEXTOLITERAL | LETRALITERAL | MIXCHELADA | DIOS | DIOSNO

F → (E)

Desglose G = (N, T, P, S)

- **N** = {I, EO, C, ER, OR, OL, D, S, DV, A, EC, H, E, T, F}
- **T** = {CaeCliente, SiNoCae, (,), {, }, <, >, <=, >=, ==, DIOS, DIOSNO, +, -, *, /, NUMERITO, TEXTOLITERAL, LETRALITERAL, MIXCHELADA, intCHELADA, granito, cadena, caracter, =, ,, holahola, papoi, paraPapoi}
- **P** = {
    - I → CaeCliente (C) { D }
    - I → CaeCliente (C) { D } EO
    - EO → SiNoCae { D }
    - C → ER
    - C → ER OL ER
    - ER → E OR E

- OR → <
- OR → >
- OR → <=
- OR → >=
- OR → ==
- OL → DIOS
- OL → DIOSNO
- D → S D
- D → ε
- S → DV
- S → A
- S → EC
- S → H
- E → T
- E → T + T
- E → T - T
- T → F
- T → F * F
- T → F / F
- F → NUMERITO
- F → TEXTOLITERAL
- F → LETRALITERAL
- F → MIXCHELADA
- F → DIOS
- F → DIOSNO
- F → (E) }

- **S** = I

Ejemplos válidos:

```
CaeCliente(x > 10) {
   holahola("Mayor que 10")
}
```

```
CaeCliente(x == y) {
    holahola("Iguales")
} SiNoCae {
    holahola("Diferentes")
}

CaeCliente(x > 5 DIOS y < 10) {
    holahola("Entre 5 y 10")
}
```

---

6. Gramática para WHILE (papoi)

Gramática G_while

W → papoi (C) { D }

C → ER

C → ER OL ER

ER → E OR E

OR → < | > | <= | >= | ==

OL → DIOS | DIOSNO

D → S D | ε

S → DV | A | EC | H

E → T

E → T + T

E → T - T

T → F

T → F * F

T → F / F

F → NUMERITO | TEXTOLITERAL | LETRALITERAL | MIXCHELADA | DIOS | DIOSNO

F → (E)

Desglose G = (N, T, P, S)

- **N** = {W, C, ER, OR, OL, D, S, DV, A, EC, H, E, T, F}

- **T** = {papoi, (,), {, }, <, >, <=, >=, ==, DIOS, DIOSNO, +, -, *, /, NUMERITO, TEXTOLITERAL, LETRALITERAL, MIXCHELADA, intCHELADA, granito, cadena, caracter, =, ,, holahola, CaeCliente, SiNoCae, paraPapoi}
- **P** = {
    - W → papoi (C) { D }
    - C → ER
    - C → ER OL ER
    - ER → E OR E
    - OR → <
    - OR → >
    - OR → <=
    - OR → >=
    - OR → ==
    - OL → DIOS
    - OL → DIOSNO
    - D → S D
    - D → ε
    - S → DV
    - S → A
    - S → EC
    - S → H
    - E → T
    - E → T + T
    - E → T - T
    - T → F
    - T → F * F
    - T → F / F
    - F → NUMERITO
    - F → TEXTOLITERAL
    - F → LETRALITERAL
    - F → MIXCHELADA

- F → DIOS
- F → DIOSNO
- F → (E) }

- **S** = W

Ejemplos válidos:

```
papoi(contador < 10) {
    holahola(contador)
    contador = contador + 1
}


papoi(x > 0 DIOS y > 0) {
    x = x - 1
    y = y - 1
}
```

NOTA: La implementación actual NO incluye stopPlease dentro del cuerpo del while, aunque está definido como token.

---

7. Gramática para FOR (paraPapoi)

Gramática G_for

FOR → paraPapoi (INIT : C : INC) { D }

INIT → tipo MIXCHELADA = E

INIT → MIXCHELADA = E

C → ER

C → ER OL ER

ER → E OR E

INC → MIXCHELADA = E

OR → < | > | <= | >= | ==

OL → DIOS | DIOSNO

D → S D | ε

S → DV | A | EC | H

tipo → intCHELADA | granito | cadena | caracter

E → T

E → T + T

E → T - T

T → F

T → F * F

T → F / F

F → NUMERITO | TEXTOLITERAL | LETRALITERAL | MIXCHELADA | DIOS | DIOSNO

F → (E)

Desglose G = (N, T, P, S)

- **N** = {FOR, INIT, C, ER, INC, OR, OL, D, S, DV, A, EC, H, tipo, E, T, F}
- **T** = {paraPapoi, (,), :, {, }, =, <, >, <=, >=, ==, +, -, *, /, DIOS, DIOSNO, intCHELADA, granito, cadena, caracter, MIXCHELADA, NUMERITO, TEXTOLITERAL, LETRALITERAL, ,, holahola, CaeCliente, SiNoCae, papoi}
- **P** = {
    - FOR → paraPapoi (INIT : C : INC) { D }
    - INIT → tipo MIXCHELADA = E
    - INIT → MIXCHELADA = E
    - C → ER
    - C → ER OL ER
    - ER → E OR E
    - INC → MIXCHELADA = E
    - OR → <
    - OR → >
    - OR → <=
    - OR → >=
    - OR → ==
    - OL → DIOS
    - OL → DIOSNO
    - D → S D
    - D → ε
    - S → DV
    - S → A
    - S → EC
    - S → H

- tipo → intCHELADA
- tipo → granito
- tipo → cadena
- tipo → caracter
- E → T
- E → T + T
- E → T - T
- T → F
- T → F * F
- T → F / F
- F → NUMERITO
- F → TEXTOLITERAL
- F → LETRALITERAL
- F → MIXCHELADA
- F → DIOS
- F → DIOSNO
- F → (E) }

- **S** = FOR

Ejemplos válidos:

```
paraPapoi(intCHELADA i = 0 : i < 10 : i = i + 1) {
   holahola(i)
}

paraPapoi(contador = 0 : contador < 100 : contador = contador + 5) {
   holahola("Conteo de 5 en 5")
}
```

NOTA IMPORTANTE: La inicialización permite declarar una nueva variable CON tipo, o usar una variable ya existente SIN tipo.

---

8. Gramática para PRINT (holahola)

Gramática G_print

H → holahola ( )

H → holahola (ARGS)

ARGS → E

ARGS → E , ARGS

E → T

E → T + T

E → T - T

T → F

T → F * F

T → F / F

F → NUMERITO | TEXTOLITERAL | LETRALITERAL | MIXCHELADA | DIOS | DIOSNO

F → (E)

Desglose G = (N, T, P, S)

- **N** = {H, ARGS, E, T, F}
- **T** = {holahola, (,), ,, +, -, *, /, NUMERITO, TEXTOLITERAL, LETRALITERAL, MIXCHELADA, DIOS, DIOSNO}
- **P** = {
    - H → holahola ( )
    - H → holahola (ARGS)
    - ARGS → E
    - ARGS → E , ARGS
    - E → T
    - E → T + T
    - E → T - T
    - T → F
    - T → F * F
    - T → F / F
    - F → NUMERITO
    - F → TEXTOLITERAL
    - F → LETRALITERAL
    - F → MIXCHELADA
    - F → DIOS
    - F → DIOSNO

- F → (E) }

- **S** = H

Ejemplos válidos:

holahola()

holahola("Hola Mundo")

holahola(x)

holahola("Resultado: ", x + y)

holahola("Nombre: ", nombre, " Edad: ", edad)

---

9. Gramática para Expresiones Aritméticas

Gramática G_expresiones

E → T

E → T + T

E → T - T

T → F

T → F * F

T → F / F

F → NUMERITO | TEXTOLITERAL | LETRALITERAL | MIXCHELADA | DIOS | DIOSNO

F → (E)

Desglose G = (N, T, P, S)

- **N** = {E, T, F}
- **T** = {+, -, *, /, (,), NUMERITO, TEXTOLITERAL, LETRALITERAL, MIXCHELADA, DIOS, DIOSNO}
- **P** = {
  - E → T
  - E → T + T
  - E → T - T
  - T → F
  - T → F * F
  - T → F / F
  - F → NUMERITO
  - F → TEXTOLITERAL

- F → LETRALITERAL
- F → MIXCHELADA
- F → DIOS
- F → DIOSNO
- F → (E) }

- **S** = E

Precedencia de Operadores:

1. **Mayor precedencia:** *, / (multiplicación, división)
2. **Menor precedencia:** +, - (suma, resta)
3. **Paréntesis:** ( ) para alterar precedencia

Ejemplos válidos:

5 + 3 * 2 → evaluado como: 5 + (3 * 2) = 11

(5 + 3) * 2 → evaluado como: (5 + 3) * 2 = 16

a - b / c → evaluado como: a - (b / c)

x + y - z → evaluado izquierda a derecha

---

10. Gramática para Condiciones

Gramática G_condiciones

C → ER

C → ER OL ER

ER → E OR E

OR → < | > | <= | >= | ==

OL → DIOS | DIOSNO

E → T

E → T + T

E → T - T

T → F

T → F * F

T → F / F

F → NUMERITO | TEXTOLITERAL | LETRALITERAL | MIXCHELADA | DIOS | DIOSNO

F → (E)

Desglose G = (N, T, P, S)

- **N** = {C, ER, OR, OL, E, T, F}
- **T** = {<, >, <=, >=, ==, DIOS, DIOSNO, +, -, *, /, (,), NUMERITO, TEXTOLITERAL, LETRALITERAL, MIXCHELADA}
- **P** = {
    - C → ER
    - C → ER OL ER
    - ER → E OR E
    - OR → <
    - OR → >
    - OR → <=
    - OR → >=
    - OR → ==
    - OL → DIOS
    - OL → DIOSNO
    - E → T
    - E → T + T
    - E → T - T
    - T → F
    - T → F * F
    - T → F / F
    - F → NUMERITO
    - F → TEXTOLITERAL
    - F → LETRALITERAL
    - F → MIXCHELADA
    - F → DIOS
    - F → DIOSNO
    - F → (E) }
- **S** = C

Ejemplos válidos:

x > 10

x + y < 100

a * b >= c / d

x > 5 DIOS y < 10 (x > 5 AND y < 10)

edad >= 18 DIOSNO activo (edad >= 18 OR activo)

NOTA: DIOS funciona como operador lógico AND, DIOSNO funciona como operador lógico OR.

NOTAS IMPORTANTES SOBRE LA IMPLEMENTACIÓN

Características Clave del Parser LL(1)

1. **LOOKAHEAD(2):**
    - Se usa en Declaracion() para distinguir entre declaración y asignación
    - Se usa en Inicializacion() del FOR para distinguir si hay tipo o no
2. **Sin Recursión Izquierda:**
    - Todas las gramáticas han sido transformadas para eliminar recursión izquierda
    - Las expresiones usan iteración en lugar de recursión
3. **Case Sensitive:**
    - CarumaLang distingue mayúsculas y minúsculas
    - DIOS ≠ dios, intCHELADA ≠ intchelada
4. **Estructura Obligatoria:**
    - Todo programa DEBE iniciar con Caruma
    - Todo programa DEBE terminar con byebye
5. **Tokens NO Implementados en Sentencias:**
    - stopPlease: Definido como token pero NO usado en el cuerpo de estructuras

Tipos de Datos Soportados

- intCHELADA: Enteros
- granito: Números decimales
- cadena: Cadenas de texto
- caracter: Caracteres individuales

Valores Especiales

- DIOS: Valor booleano TRUE / Operador lógico AND
- DIOSNO: Valor booleano FALSE / Operador lógico OR

Delimitadores

- ( ): Paréntesis para expresiones y parámetros
- { }: Llaves para bloques de código
- :: Separador en estructura FOR
- ,: Separador de elementos en listas

7. GLC

Esta sección detalla la adaptación práctica de las reglas formales para la construcción del analizador sintáctico en JavaCC. Se presentan las Gramáticas Libres de Contexto (GLC) transformadas para cumplir con los requisitos de un parser **LL(1) Descendente Recursivo**. A continuación, se describen las reglas de producción tal como fueron implementadas en el código fuente, explicando las decisiones de diseño tomadas para resolver conflictos sintácticos, como la eliminación de la recursividad por la izquierda, la factorización de reglas y el uso de LOOKAHEAD para la desambiguación de instrucciones.

1. PROGRAMA COMPLETO

Gramática G_1: Programa Principal

Programa → Caruma Declaraciones byebye EOF

Producción:

- Programa deriva en la palabra reservada Caruma, seguida de cero o más declaraciones, la palabra reservada byebye y el fin de archivo.

Descripción:

- Todo programa en CarumaLang debe comenzar con Caruma y terminar con byebye.
- Entre estas palabras clave puede haber cualquier número de declaraciones (incluyendo ninguna).

---

2. DECLARACIONES

Gramática G_2: Secuencia de Declaraciones

Declaraciones → Declaracion Declaraciones

Declaraciones → ε

Producciones:

1. Una declaración seguida de más declaraciones (recursión)
2. Cadena vacía (caso base)

Descripción:

- Permite cero o más declaraciones en secuencia.
- Implementada mediante recursión a la derecha para mantener LL(1).

Gramática G_3: Tipos de Declaración

Declaracion → DeclaracionVariable [LOOKAHEAD(2)]

Declaracion → Asignacion

Declaracion → EstructuraControl

Declaracion → Impresion

Producciones:

1. Declaración de variable(s) con o sin inicialización
2. Asignación a variable existente
3. Estructura de control (IF, WHILE, FOR)
4. Instrucción de impresión

LOOKAHEAD(2):

- Se requiere mirar 2 tokens adelante para distinguir entre:
    - intCHELADA x (DeclaracionVariable)
    - x = 5 (Asignacion)

---

3. DECLARACIÓN DE VARIABLES

Gramática G_4: Declaración de Variables

DeclaracionVariable → Tipo ListaIdentificadores

DeclaracionVariable → Tipo ListaIdentificadores InicializacionOpt

InicializacionOpt → = ListaExpresiones

Producciones:

1. Tipo seguido de uno o más identificadores sin inicialización
2. Tipo seguido de uno o más identificadores con inicialización

Descripción:

- Permite declarar múltiples variables del mismo tipo.
- La inicialización es opcional pero, si se incluye, debe tener valores para todas las variables declaradas.

Gramática G_5: Tipos de Datos

Tipo → intCHELADA

Tipo → granito

Tipo → cadena

Tipo → caracter

Producciones:

- intCHELADA: Tipo entero
- granito: Tipo decimal/flotante

- cadena: Tipo cadena de texto
- caracter: Tipo carácter individual

Gramática G_6: Lista de Identificadores

ListaIdentificadores → MIXCHELADA

ListaIdentificadores → MIXCHELADA , ListaIdentificadores

Producciones:

1. Un solo identificador (caso base)
2. Identificador seguido de coma y más identificadores (recursión)

Descripción:

- Permite declarar múltiples variables en una sola línea.
- Recursión a la derecha para mantener LL(1).

Ejemplos:

x → un identificador

x, y → dos identificadores

x, y, z → tres identificadores

Gramática G_7: Lista de Expresiones

ListaExpresiones → Expresion

ListaExpresiones → Expresion , ListaExpresiones

Producciones:

1. Una sola expresión (caso base)
2. Expresión seguida de coma y más expresiones (recursión)

Descripción:

- Usada para inicializar múltiples variables.
- Debe coincidir en cantidad con ListaIdentificadores.

Ejemplos:

10 → una expresión

10, 20 → dos expresiones

10, 20, 30 → tres expresiones

---

4. ASIGNACIÓN

Gramática G_8: Asignación

Asignacion → MIXCHELADA = Expresion

Producción:

- Un identificador seguido del operador de asignación y una expresión.

Descripción:

- Asigna el valor de una expresión a una variable previamente declarada.
- La variable debe existir (verificación semántica fuera del alcance sintáctico).

Ejemplos:

x = 10

resultado = a + b * c

nombre = "CarumaLang"

---

5. ESTRUCTURAS DE CONTROL

Gramática G_9: Estructuras de Control

EstructuraControl → EstructuraIf

EstructuraControl → EstructuraWhile

EstructuraControl → EstructuraFor

Producciones:

1. Estructura condicional IF
2. Estructura repetitiva WHILE
3. Estructura repetitiva FOR

---

6. ESTRUCTURA IF (CaeCliente)

Gramática G_{10}: Estructura IF

EstructuraIf → CaeCliente (Condicion) { Declaraciones }

EstructuraIf → CaeCliente (Condicion) { Declaraciones } ElseOpt

ElseOpt → SiNoCae { Declaraciones }

Producciones:

1. IF sin ELSE: Ejecuta bloque si la condición es verdadera
2. IF con ELSE: Ejecuta un bloque u otro según la condición

Descripción:

- CaeCliente es la palabra reservada para IF.
- SiNoCae es la palabra reservada para ELSE (opcional).
- La condición debe ir entre paréntesis.
- El bloque de código debe ir entre llaves.

Ejemplos:

```
CaeCliente(x > 10) {
    holahola("Mayor que 10")
}

CaeCliente(edad >= 18) {
    holahola("Mayor de edad")
} SiNoCae {
    holahola("Menor de edad")
}
```

---

7. ESTRUCTURA WHILE (papoi)

Gramática G_{11}: Estructura WHILE

EstructuraWhile → papoi (Condicion) { Declaraciones }

Producción:

- Bucle WHILE con condición entre paréntesis y cuerpo entre llaves.

Descripción:

- papoi es la palabra reservada para WHILE.
- Ejecuta el bloque mientras la condición sea verdadera.
- La condición se evalúa antes de cada iteración.

Ejemplo:

```
papoi(contador < 10) {
    holahola(contador)
    contador = contador + 1
}
```

---

8. ESTRUCTURA FOR (paraPapoi)

Gramática G_{12}: Estructura FOR

EstructuraFor → paraPapoi (Inicializacion : Condicion : Incremento) { Declaraciones }

Producción:

- Bucle FOR con tres componentes separados por dos puntos.

Descripción:

- paraPapoi es la palabra reservada para FOR.
- **Inicialización:** Se ejecuta una sola vez al inicio.
- **Condición:** Se evalúa antes de cada iteración.
- **Incremento:** Se ejecuta al final de cada iteración.

Gramática G_{13}: Inicialización del FOR

Inicializacion → Tipo MIXCHELADA = Expresion [LOOKAHEAD(2)]

Inicializacion → MIXCHELADA = Expresion

Producciones:

1. Declaración de variable nueva con inicialización
2. Asignación a variable existente

LOOKAHEAD(2):

- Se requiere para distinguir entre:
    - intCHELADA i = 0 (declaración nueva)
    - contador = 0 (variable existente)

Ejemplos:

intCHELADA i = 0 → declaración nueva

contador = 0 → variable existente

Gramática G_{14}: Incremento del FOR

Incremento → MIXCHELADA = Expresion

Producción:

- Asignación que modifica la variable de control.

Descripción:

- Típicamente incrementa o decrementa el contador.
- Puede ser cualquier expresión válida.

Ejemplos:

i = i + 1 → incremento de 1

contador = contador + 5 → incremento de 5

j = j - 1 → decremento de 1

---

9. CONDICIONES

Gramática G_{15}: Condiciones

Condicion → ExpresionRelacional

Condicion → ExpresionRelacional OperadorLogico ExpresionRelacional

Condicion → ExpresionRelacional OperadorLogico ExpresionRelacional OperadorLogico ExpresionRelacional

...

Forma Simplificada:

Condicion → ExpresionRelacional (OperadorLogico ExpresionRelacional)*

Producciones:

1. Una sola expresión relacional
2. Múltiples expresiones relacionales conectadas con operadores lógicos

Descripción:

- Permite condiciones simples y compuestas.
- Los operadores lógicos conectan expresiones relacionales.

Ejemplos:

x > 10 → condición simple

x > 5 DIOS y < 10 → condición compuesta con AND

edad >= 18 DIOSNO activo → condición compuesta con OR

Gramática G_{16}: Expresión Relacional

ExpresionRelacional → Expresion OperadorRelacional Expresion

Producción:

- Dos expresiones conectadas por un operador relacional.

Descripción:

- Compara dos valores usando operadores relacionales.
- Retorna un valor booleano (verdadero o falso).

Gramática G_{17}: Operadores Relacionales

OperadorRelacional → <

OperadorRelacional → >

OperadorRelacional → <=

OperadorRelacional → >=

OperadorRelacional → ==

Producciones:

- <: Menor que
- >: Mayor que
- <=: Menor o igual que
- >=: Mayor o igual que
- ==: Igual a

Gramática G_{18}: Operadores Lógicos

OperadorLogico → DIOS

OperadorLogico → DIOSNO

Producciones:

- DIOS: Operador lógico AND (conjunción)
- DIOSNO: Operador lógico OR (disyunción)

Descripción:

- DIOS retorna verdadero si AMBAS condiciones son verdaderas.
- DIOSNO retorna verdadero si AL MENOS UNA condición es verdadera.

---

10. EXPRESIONES ARITMÉTICAS

Gramática G_{19}: Expresiones

Expresion → Termino

Expresion → Termino + Termino

Expresion → Termino - Termino

Expresion → Termino + Termino + Termino

...

Forma Simplificada:

Expresion → Termino ((+ | -) Termino)*

Producciones:

1. Un solo término

2. Términos conectados por suma o resta

Descripción:

- Operadores de suma y resta tienen la misma precedencia.
- Asociatividad izquierda: a - b + c se evalúa como (a - b) + c.
- Implementada con iteración (no recursión) para mantener LL(1).

Gramática G_{20}: Términos

Termino → Factor

Termino → Factor * Factor

Termino → Factor / Factor

Termino → Factor * Factor * Factor

...

Forma Simplificada:

Termino → Factor ((* | /) Factor)*

Producciones:

1. Un solo factor
2. Factores conectados por multiplicación o división

Descripción:

- Operadores de multiplicación y división tienen mayor precedencia que suma/resta.
- Asociatividad izquierda: a / b * c se evalúa como (a / b) * c.

Gramática G_{21}: Factores

Factor → NUMERITO

Factor → TEXTOLITERAL

Factor → LETRALITERAL

Factor → MIXCHELADA

Factor → DIOS

Factor → DIOSNO

Factor → (Expresion)

Producciones:

1. **NUMERITO**: Literal numérico (entero o decimal)
2. **TEXTOLITERAL**: Literal de cadena entre comillas dobles

3. **LETRALITERAL**: Literal de carácter entre comillas simples
4. **MIXCHELADA**: Identificador de variable
5. **DIOS**: Valor booleano TRUE
6. **DIOSNO**: Valor booleano FALSE
7. **Paréntesis**: Expresión entre paréntesis para alterar precedencia

Descripción:

- Los factores son los elementos atómicos de las expresiones.
- Los paréntesis permiten forzar el orden de evaluación.

Ejemplos:

42 → literal numérico

"Hola" → literal de cadena

'A' → literal de carácter

x → identificador

DIOS → booleano true

(a + b) → expresión entre paréntesis

Precedencia de Operadores (de mayor a menor):

Nivel	Operadores	Descripción	Asociatividad
1	( )	Paréntesis	-
2	*, /	Multiplicación, División	Izquierda
3	+, -	Suma, Resta	Izquierda
4	<, >, <=, >=, ==	Relacionales	-
5	DIOS, DIOSNO	Lógicos (AND, OR)	Izquierda

Ejemplos de Evaluación:

5 + 3 * 2 → 5 + (3 * 2) = 11

(5 + 3) * 2 → (5 + 3) * 2 = 16

10 - 4 + 2 → (10 - 4) + 2 = 8

8 / 2 * 3 → (8 / 2) * 3 = 12

---

11. IMPRESIÓN (holahola)

Gramática G_{22}: Impresión

Impresion → holahola ( )

Impresion → holahola (Argumentos)

Producciones:

1. Impresión sin argumentos (línea vacía)
2. Impresión con uno o más argumentos

Descripción:

- holahola es la palabra reservada para imprimir.
- Puede no tener argumentos o tener múltiples argumentos separados por comas.

Ejemplos:

holahola() → imprime línea vacía

holahola("Hola Mundo") → imprime cadena

holahola("Resultado: ", x + y) → imprime cadena y expresión

Gramática G_{23}: Argumentos de Impresión

Argumentos → Expresion

Argumentos → Expresion , Argumentos

Producciones:

1. Una sola expresión (caso base)
2. Expresión seguida de coma y más argumentos (recursión)

Descripción:

- Permite múltiples argumentos separados por comas.
- Cada argumento es una expresión que puede ser evaluada.

Ejemplos:

x → un argumento

"Nombre: ", nombre → dos argumentos

x, y, z → tres argumentos

"Suma: ", a + b, " Resta: ", a - b → cuatro argumentos

---

8. ÁRBOLES

Los árboles de derivación, también conocidos como árboles de análisis sintáctico (*parse trees*), son la representación gráfica y jerárquica de la estructura gramatical del código fuente. Estos diagramas ilustran explícitamente cómo el analizador aplica las reglas de producción de las Gramáticas Libres de Contexto (GLC) para descomponer una cadena de entrada en sus componentes constituyentes, partiendo desde el símbolo inicial hasta llegar a los tokens terminales.

En el contexto de CarumaLang, estos árboles permiten visualizar la precedencia de operadores, la anidación de estructuras de control y la recursividad definida en el parser LL(1). A continuación, se presentan los diagramas correspondientes a las estructuras fundamentales del lenguaje.

8.1 Estructura General del Programa

La estructura base de todo archivo en CarumaLang está definida por la **Gramática G_1**, la cual establece que un programa válido debe estar encapsulado entre las palabras reservadas de inicio y fin.

En la **Imagen 41**, se muestra el diagrama del árbol de derivación de la gramática correspondiente al **Programa Completo**. Este árbol tiene como raíz el no-terminal Programa, del cual se derivan obligatoriamente el token de inicio Caruma, el cuerpo de Declaraciones (que puede derivar en múltiples instrucciones o en vacío), el token de cierre byebye y el marcador de fin de archivo EOF.

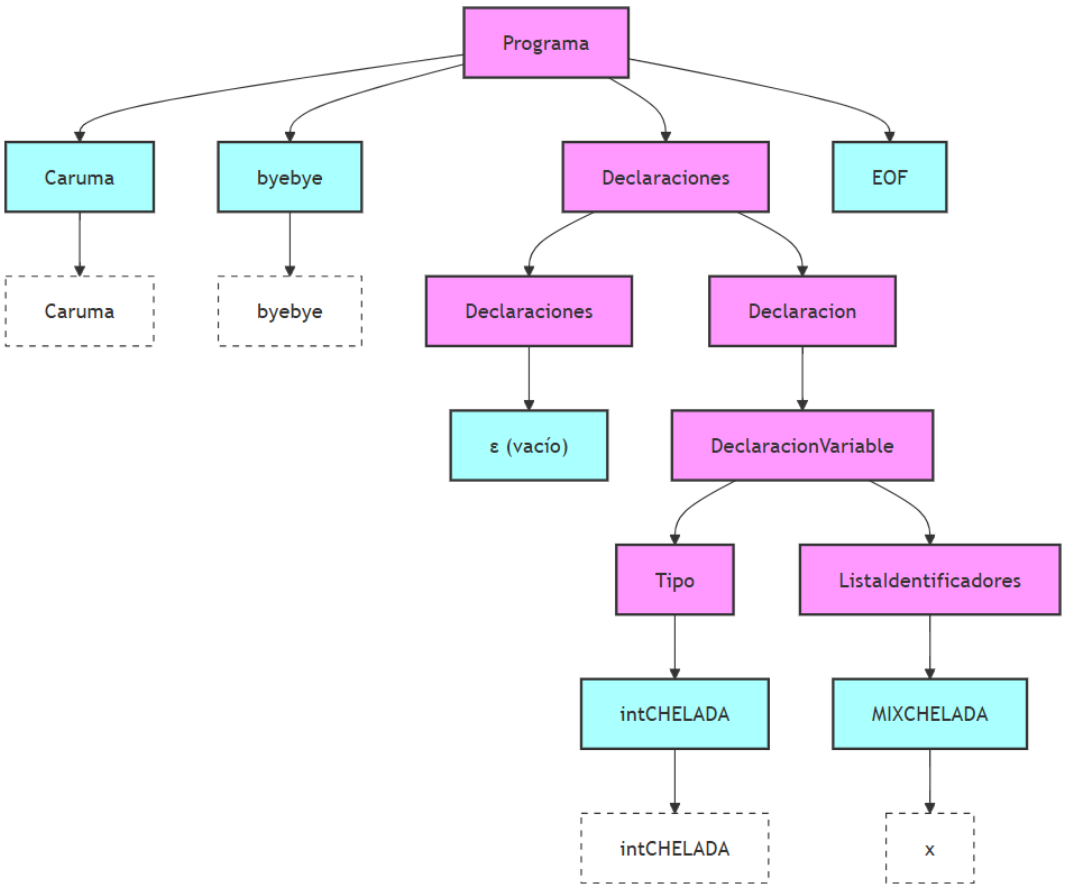


IMAGEN 41: ÁRBOL DE DERIVACIÓN G_1 - ESTRUCTURA DEL PROGRAMA PRINCIPAL.

8.2 Secuencia de Declaraciones

El cuerpo del programa se construye mediante la **Gramática G_2**, la cual utiliza recursividad por la derecha para permitir una secuencia indefinida de instrucciones.

En la **Imagen 42**, se muestra el diagrama del árbol de derivación de la gramática correspondiente a la **Secuencia de Declaraciones**. El nodo Declaraciones se expande en una Declaracion individual y un nuevo nodo Declaraciones recursivo. Este ciclo continúa hasta que se deriva en *e* (cadena vacía), indicando que no hay más instrucciones por procesar.

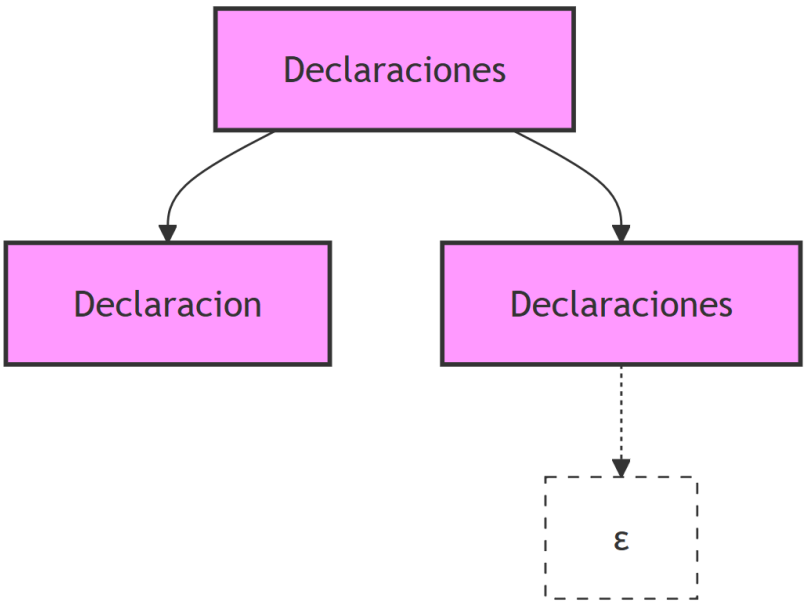


IMAGEN 42: ÁRBOL DE DERIVACIÓN G_2 - SECUENCIA DE DECLARACIONES.

8.3 Tipos de Declaración (Gramática G_3)

Esta regla actúa como un despachador que decide qué tipo de instrucción se está procesando.

En la **Imagen 43**, se muestra el árbol para **G_3: Declaración**. El parser evalúa el token actual (o los siguientes dos con LOOKAHEAD) para decidir si deriva en DeclaracionVariable, Asignacion, EstructuraControl o Impresión.

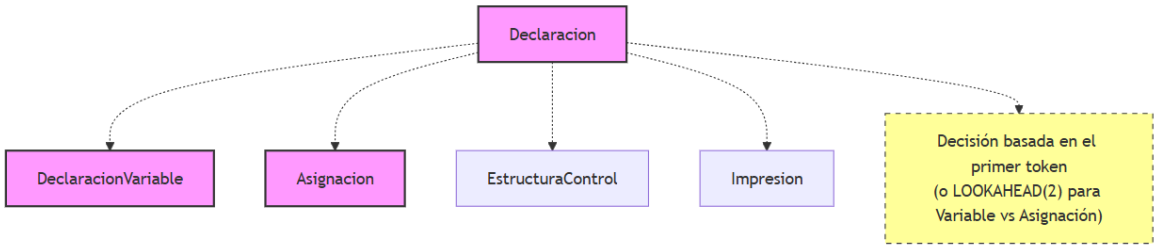


IMAGEN 43: ÁRBOL DE DERIVACIÓN G_3 - TIPOS DE DECLARACIÓN (DESPACHADOR).

8.4 Estructura de Variables (Gramática G_4)

Define la sintaxis para declarar nuevas variables, con o sin valor inicial.

En la **Imagen 44**, se presenta el árbol para **G_4: Declaración de Variables**. El nodo DeclaracionVariable se compone de un Tipo obligatorio, una ListaIdentificadores (para una o más variables) y una InicializacionOpt opcional para asignar valores.

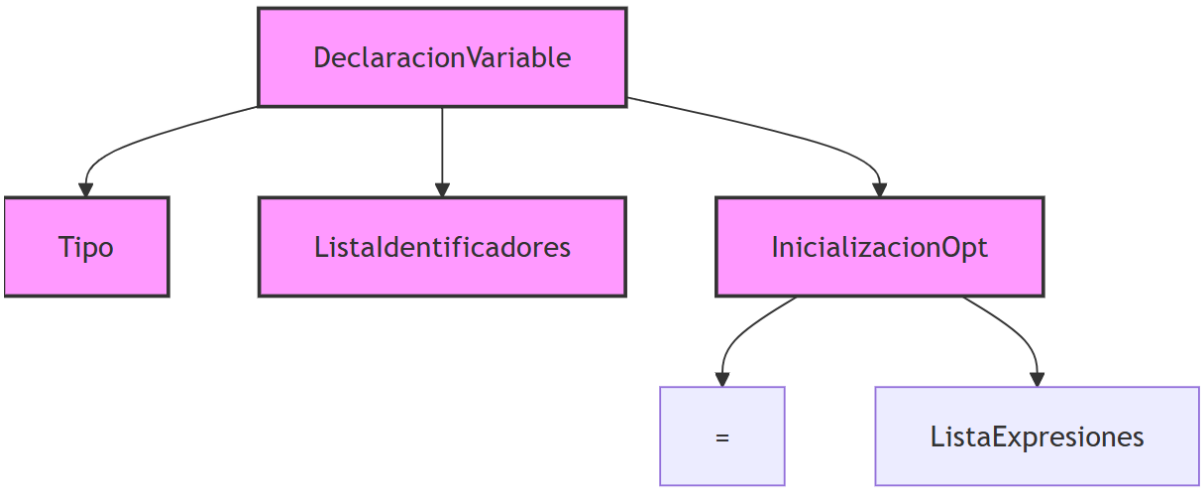


IMAGEN 44: ÁRBOL DE DERIVACIÓN G_4 - DECLARACIÓN DE VARIABLES.

8.5 Tipos de Datos (Gramática G_5)

Especifica los tipos primitivos soportados por el lenguaje.

En la **Imagen 45**, se muestra el árbol de selección para **G_5: Tipos**. El nodo Tipo deriva directamente en uno de los tokens reservados que definen el tipo de dato: intCHELADA (entero), granito (flotante), cadena (string) o caracter (char)

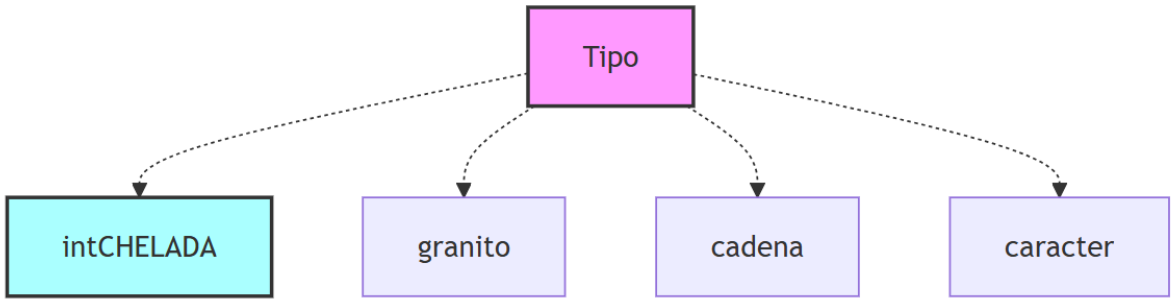


IMAGEN 45: ÁRBOL DE DERIVACIÓN G_5 - TIPOS DE DATOS PRIMITIVOS.

8.6 Listas de Identificadores (Gramática G_6)

Permite la declaración múltiple de variables en una sola línea (ej. x, y, z).

En la **Imagen 46**, se ilustra la recursividad de **G_6: Lista de Identificadores**. El nodo consume un identificador MIXCHELADA y, si encuentra una coma, se llama a sí mismo para procesar el siguiente identificador, permitiendo listas de longitud arbitraria

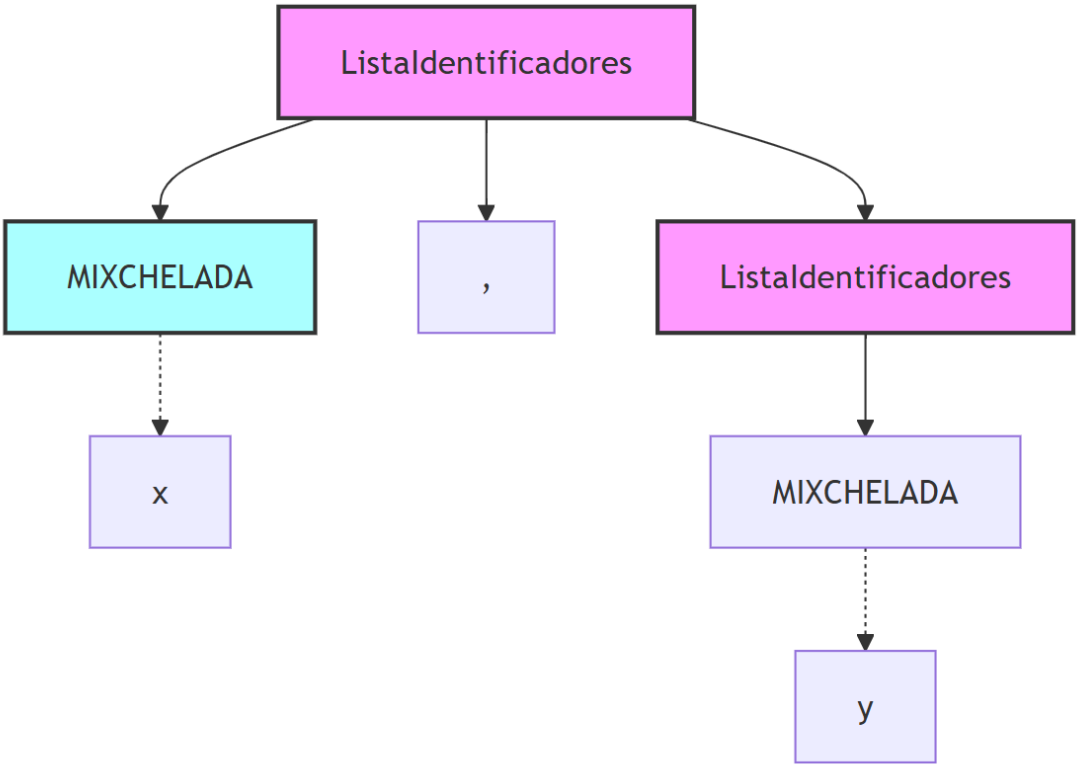


IMAGEN 46: ÁRBOL DE DERIVACIÓN G_6 - LISTA RECURSIVA DE IDENTIFICADORES.

8.7 Listas de Expresiones (Gramática G_7)

Estructura complementaria a la lista de identificadores, usada para asignar valores múltiples.

En la **Imagen 47**, se muestra el árbol para **G_7: Lista de Expresiones**. Similar a la lista de identificadores, este nodo consume una Expresion y, mediante recursividad separada por comas, permite encadenar múltiples valores para inicializar variables

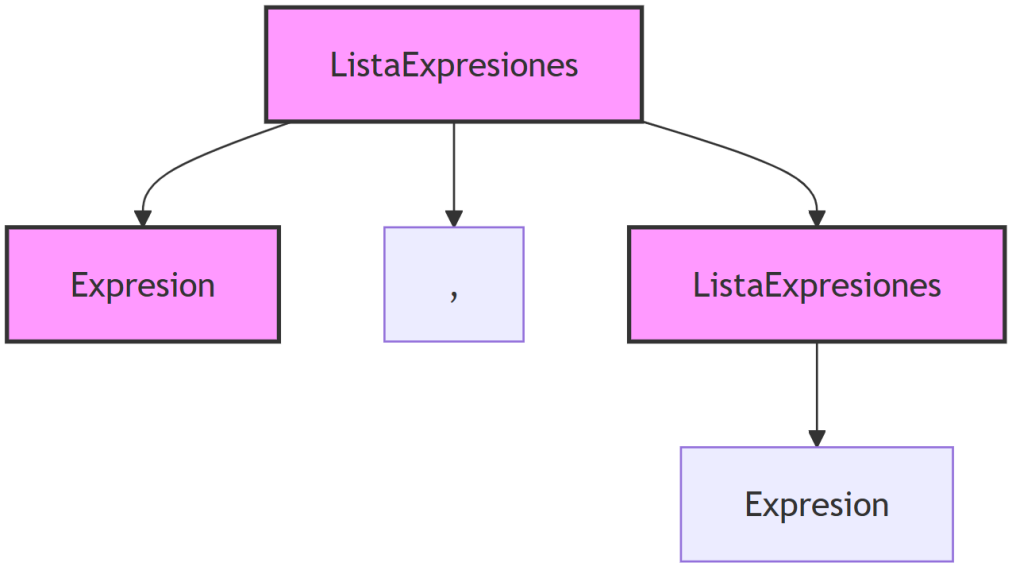


IMAGEN 47: ÁRBOL DE DERIVACIÓN G_7 - LISTA RECURSIVA DE EXPRESIONES.

8.8 Asignación (Gramática G_8)

Define la modificación de valor de una variable existente.

En la **Imagen 48**, se presenta el árbol para **G_8: Asignación**. La estructura es lineal: comienza con el identificador destino <MIXCHELADA>, seguido del operador = y finaliza con la Expresion cuyo resultado será almacenado

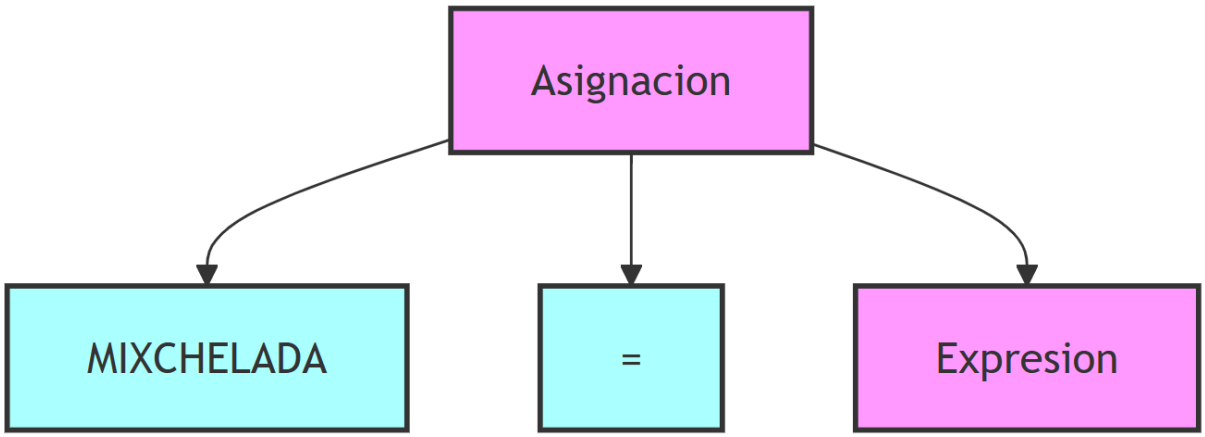


IMAGEN 48: ÁRBOL DE DERIVACIÓN G_8 - ASIGNACIÓN DE VALORES.

8.9 Estructuras de Control (Gramática G_9)

Nodo despachador para el flujo lógico del programa.

En la **Imagen 49**, se muestra el árbol de decisión para **G_9: Estructuras de Control**. Este no-terminal no consume tokens por sí mismo, sino que deriva en EstructuraIf, EstructuraWhile o EstructuraFor dependiendo de la palabra reservada encontrada (CaeCliente, papoi, paraPapoi)

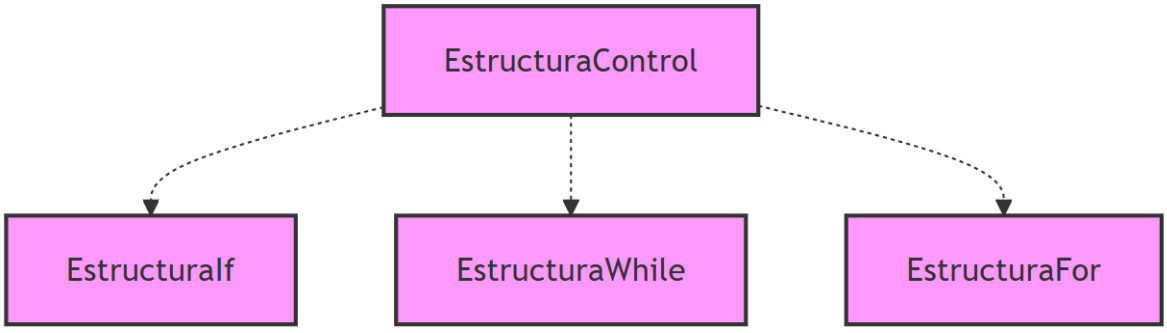


IMAGEN 49: ÁRBOL DE DERIVACIÓN G_9 - ESTRUCTURAS DE CONTROL.

8.10 Estructura IF (Gramática G_{10})

Define la ejecución condicional mediante CaeCliente.

En la **Imagen 50**, se ilustra el árbol de **G_{10}: Estructura IF**. Se observan los componentes obligatorios: token CaeCliente, Condicion entre paréntesis y bloque de código entre llaves. Adicionalmente, se muestra la rama opcional ElseOpt que maneja el caso contrario (SiNoCae)

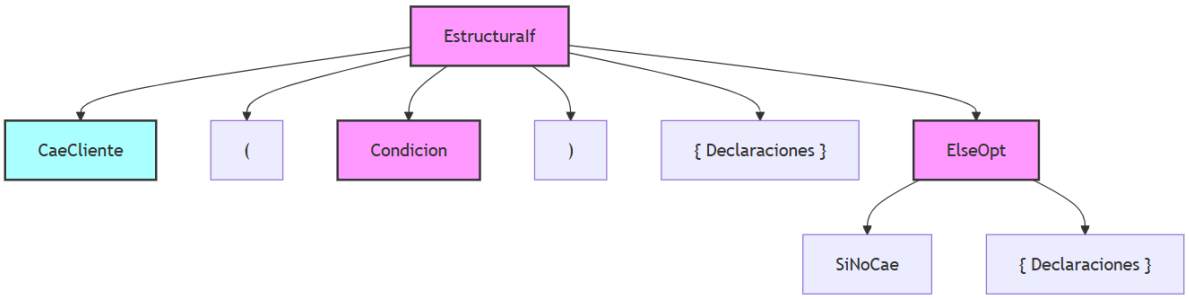


IMAGEN 50: ÁRBOL DE DERIVACIÓN G_{10} - ESTRUCTURA CONDICIONAL IF (CAECLIENTE).

8.11 Estructura WHILE (Gramática G_{11})

Define el bucle de repetición papoi.

En la **Imagen 51**, se muestra el árbol para **G_{11}: Estructura WHILE**. La raíz deriva en la palabra reservada papoi, una Condicion de control y el cuerpo del bucle, representando la ejecución iterativa basada en una expresión booleana

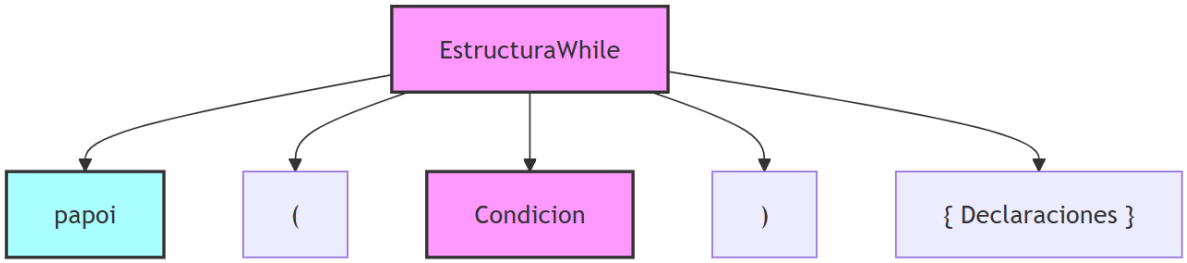


IMAGEN 51: ÁRBOL DE DERIVACIÓN G_{11} - ESTRUCTURA DE REPETICIÓN WHILE (PAPOI).

8.12 Estructura FOR (Gramática G_{12})

Define el bucle controlado paraPapoi.

En la **Imagen 52**, se presenta el árbol de **G_{12}: Estructura FOR**. Este diagrama detalla los tres componentes de control separados por el token especial : (AHIVA): Inicializacion, Condicion e Incremento, seguidos por el bloque de código a repetir

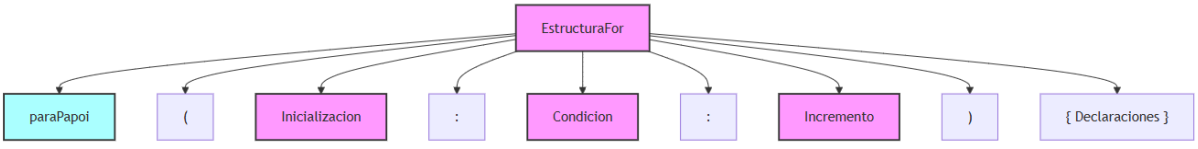


IMAGEN 52: ÁRBOL DE DERIVACIÓN G_{12} - ESTRUCTURA DE REPETICIÓN FOR (PARAPAPOI).

8.13 Inicialización del FOR (Gramática G_{13})

Regla especial para permitir declaración o asignación en el inicio del bucle.

En la **Imagen 53**, se muestra el árbol para **G_{13}: Inicialización del FOR**. Este nodo utiliza LOOKAHEAD para bifurcarse en dos opciones: declarar una nueva variable de control (Rama Izquierda: Tipo + MIXCHELADA) o utilizar una variable existente (Rama Derecha: MIXCHELADA)

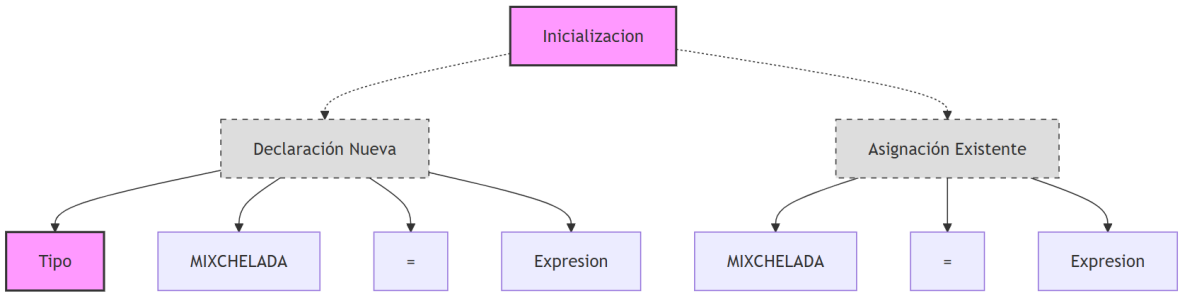


IMAGEN 53: ÁRBOL DE DERIVACIÓN G_{13} - INICIALIZACIÓN DEL BUCLE FOR.

8.14 Incremento del Bucle (Gramática 14)

La **Gramática G_{14}** define la tercera parte de la estructura FOR, encargada de actualizar la variable de control al final de cada iteración.

En la **Imagen 54**, se muestra el árbol de derivación para una instrucción de **Incremento** típica (i = i + 1). El nodo raíz deriva en el identificador <MIXCHELADA>, el operador de asignación = y la Expresion que calcula el nuevo valor

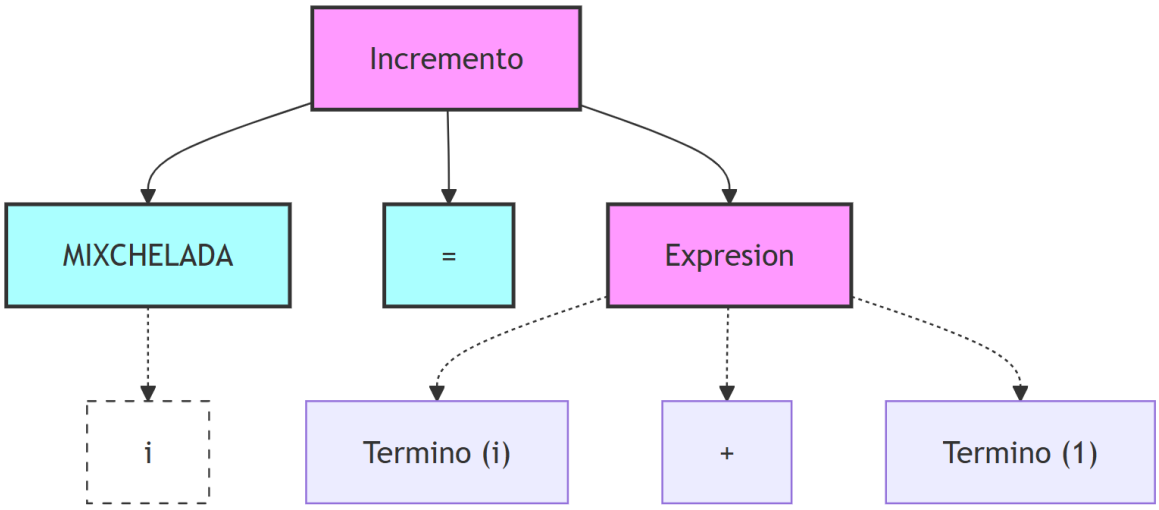


IMAGEN 54: ÁRBOL DE DERIVACIÓN G_{14} - INCREMENTO DEL BUCLE FOR.

8.15 Lógica de Condiciones Compuestas (Gramática 15)

La **Gramática G_{15}** permite la creación de condiciones complejas uniendo múltiples comparaciones mediante operadores lógicos.

En la **Imagen 55**, se ilustra el árbol para una **Condición Compuesta** (a > b DIOS c < d). El nodo Condicion se expande para conectar dos ExpresionRelacional a través de un OperadorLogico (en este caso, DIOS que corresponde al AND), permitiendo evaluar estructuras booleanas complejas

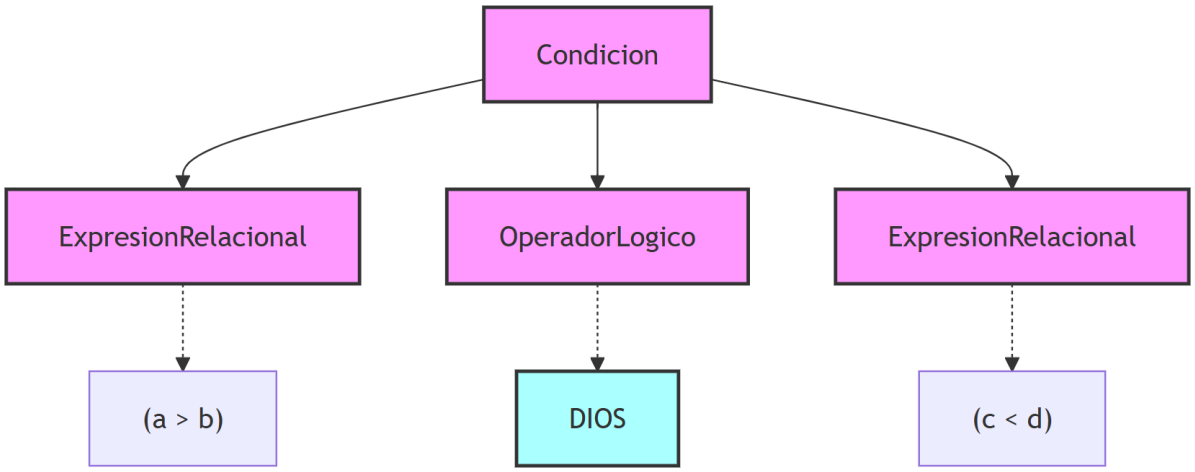


IMAGEN 55: ÁRBOL DE DERIVACIÓN G_{15} - CONDICIONES LÓGICAS.

8.16 Expresiones Relacionales (Gramática 16)

La **Gramática G_{16}** es el bloque constructor básico de las condiciones, encargada de comparar dos valores numéricos.

En la **Imagen 56**, se presenta el árbol de derivación para una **Expresión Relacional** (edad >= 18). El diagrama muestra cómo el no-terminal conecta dos Expresiones (izquierda y derecha) mediante un OperadorRelacional central.

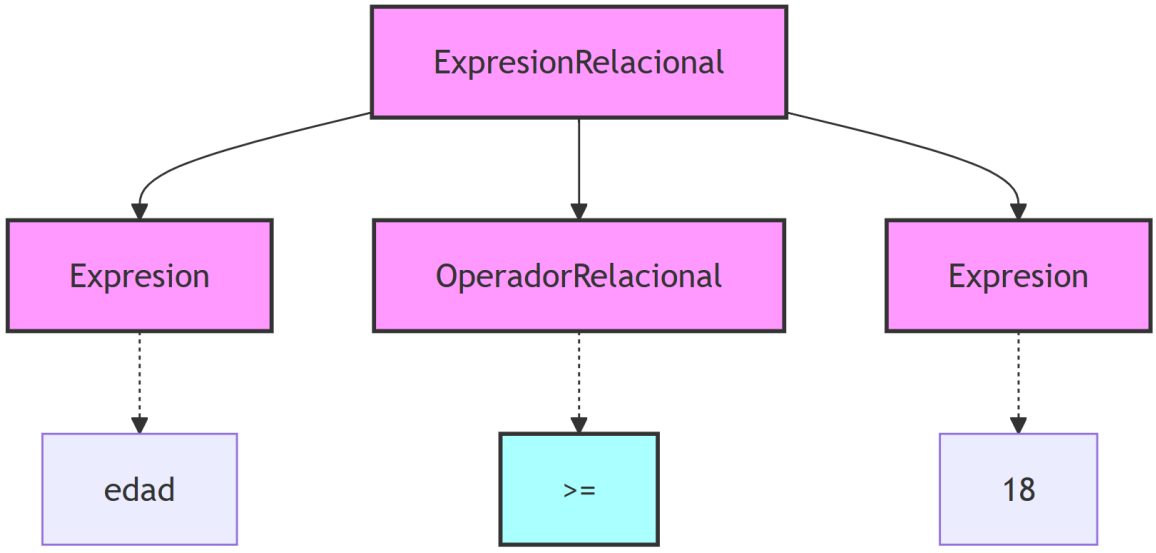


IMAGEN 56: ÁRBOL DE DERIVACIÓN G_{16} - EXPRESIONES RELACIONALES.

8.17 y 8.18 Selección de Operadores (Gramáticas 17 y 18)

Estas gramáticas definen los terminales válidos para las comparaciones y la lógica booleana.

En la **Imagen 57**, se muestran los árboles de selección para **Operadores Relacionales (G_{17})** y en la **imagen 58** los **Operadores Lógicos (G_{18})**. Estos nodos no tienen estructura compleja, sino que derivan directamente en uno de los tokens reservados permitidos (<, >, DIOS, DIOSNO, etc.)

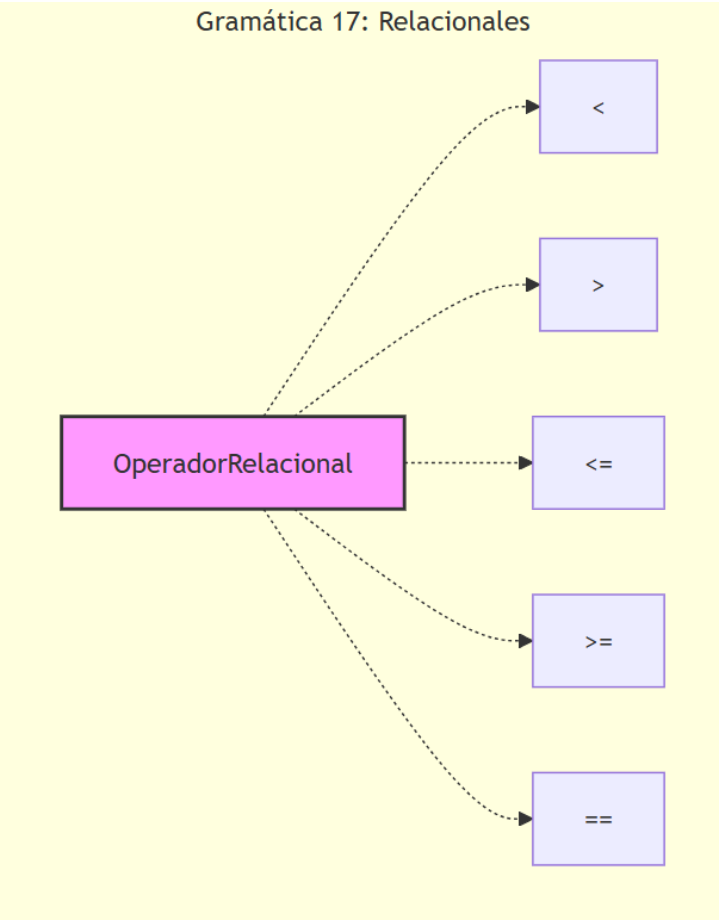


IMAGEN 57: ÁRBOL DE DERIVACIÓN G_{17} - OPERADORES RELACIONALES.

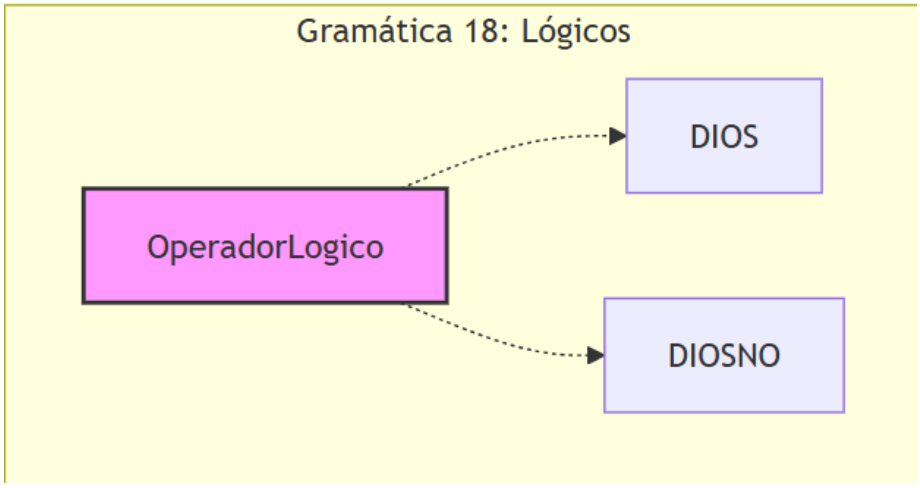


IMAGEN 58: ÁRBOL DE DERIVACIÓN G_{18} - OPERADORES LÓGICOS

8.19 Estructura de Expresiones (Gramática 19)

La **Gramática G_{19}** maneja las operaciones de menor precedencia (suma y resta) y es la entrada a cualquier cálculo matemático.

En la **Imagen 59**, se visualiza el árbol para la **Expresión** a + b. Para evitar la recursividad izquierda infinita en el parser LL(1), esta regla se implementa de forma iterativa: un Termino inicial seguido de cero o más pares de (Operador Termino)

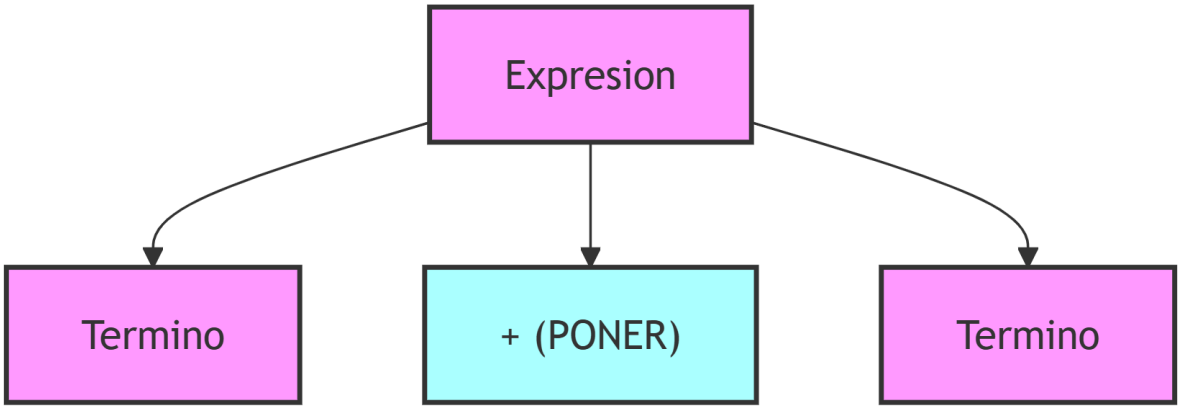


IMAGEN 59: ÁRBOL DE DERIVACIÓN G_{19} - EXPRESIONES ARITMÉTICAS (SUMA/RESTA).

8.20 Jerarquía de Términos (Gramática 20)

La **Gramática G_{20}** gestiona las operaciones de mayor precedencia (multiplicación y división).

En la **Imagen 60**, se muestra el árbol para un **Término** x * y. Al igual que la expresión, utiliza una estructura iterativa descendente hacia Factor, asegurando que estas operaciones se agrupen antes que las sumas en el árbol sintáctico general

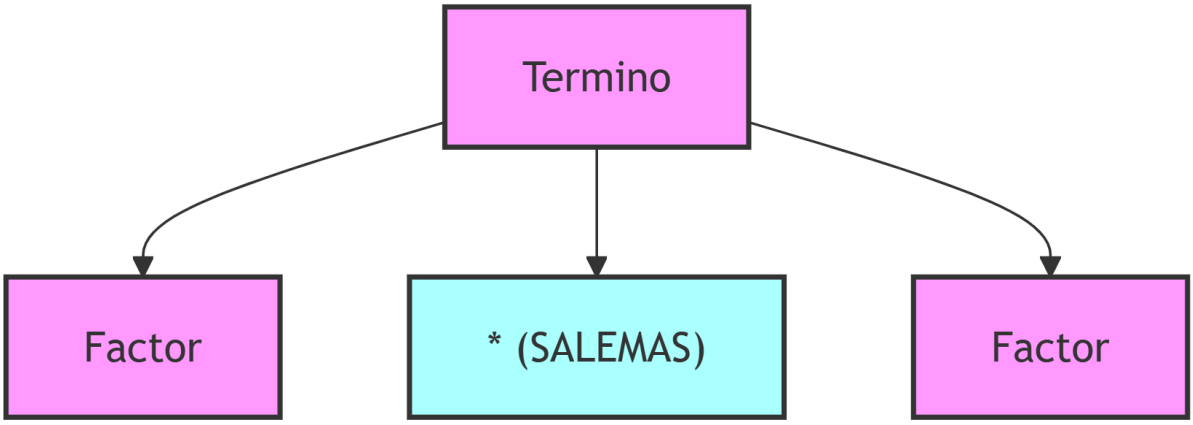


IMAGEN 60: ÁRBOL DE DERIVACIÓN G_{20} - TÉRMINOS (MULTIPLICACIÓN/DIVISIÓN).

8.21 Factores y Atomicidad (Gramática 21)

La **Gramática G_{21}** define los elementos atómicos que pueden operar (números, variables) y permite alterar la precedencia mediante paréntesis.

En la **Imagen 61**, se presenta el árbol de selección para **Factores**. El nodo puede derivar en un literal (NUMERITO, TEXTOLITERAL), un identificador (MIXCHELADA) o, crucialmente, en una nueva Expresion encerrada entre paréntesis, lo que permite reiniciar la jerarquía de evaluación

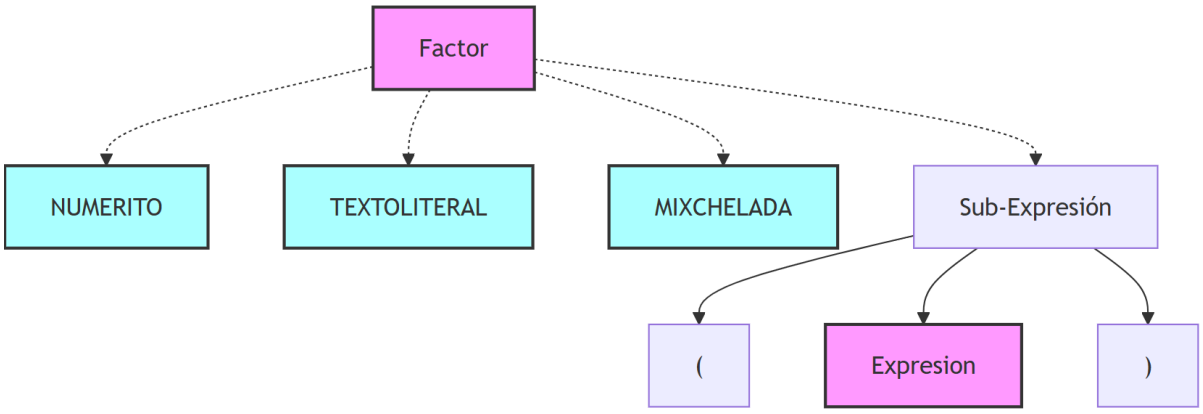


IMAGEN 61: ÁRBOL DE DERIVACIÓN G_{21} - FACTORES Y ATOMICIDAD.

8.22 Sentencia de Impresión (Gramática 22)

La **Gramática G_{22}** define la llamada a la función de salida del sistema.

En la **Imagen 62**, se muestra el árbol de derivación para **Impresión**. La estructura consiste en la palabra reservada holahola, seguida de paréntesis que pueden contener (opcionalmente) una lista de Argumentos a imprimir.

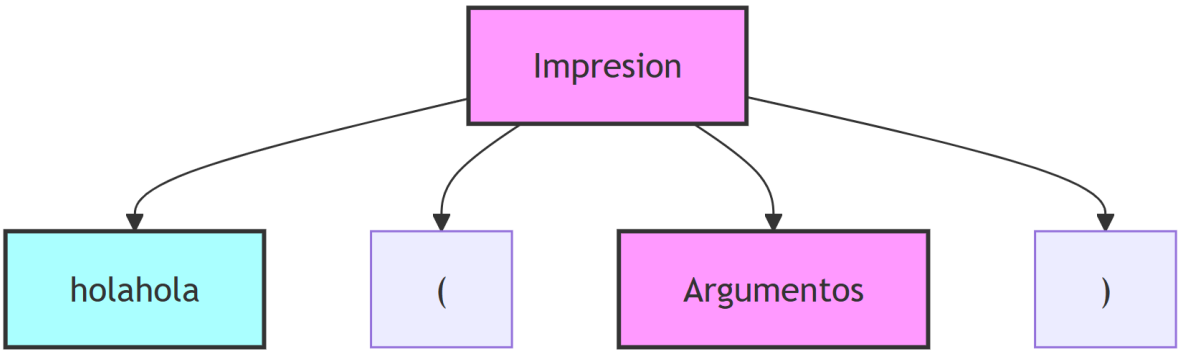


IMAGEN 62: ÁRBOL DE DERIVACIÓN G_{22} - INSTRUCCIÓN DE IMPRESIÓN (HOLAHOLA).

8.23 Listas de Argumentos (Gramática 23)

Finalmente, la **Gramática G_{23}** permite pasar múltiples valores a la función de impresión.

En la **Imagen 63**, se ilustra el árbol recursivo para **Argumentos de Impresión** ("Hola", nombre). El nodo consume una Expresion y, si encuentra una coma, invoca nuevamente a Argumentos, permitiendo imprimir cadenas concatenadas o listas de variables en una sola instrucción.

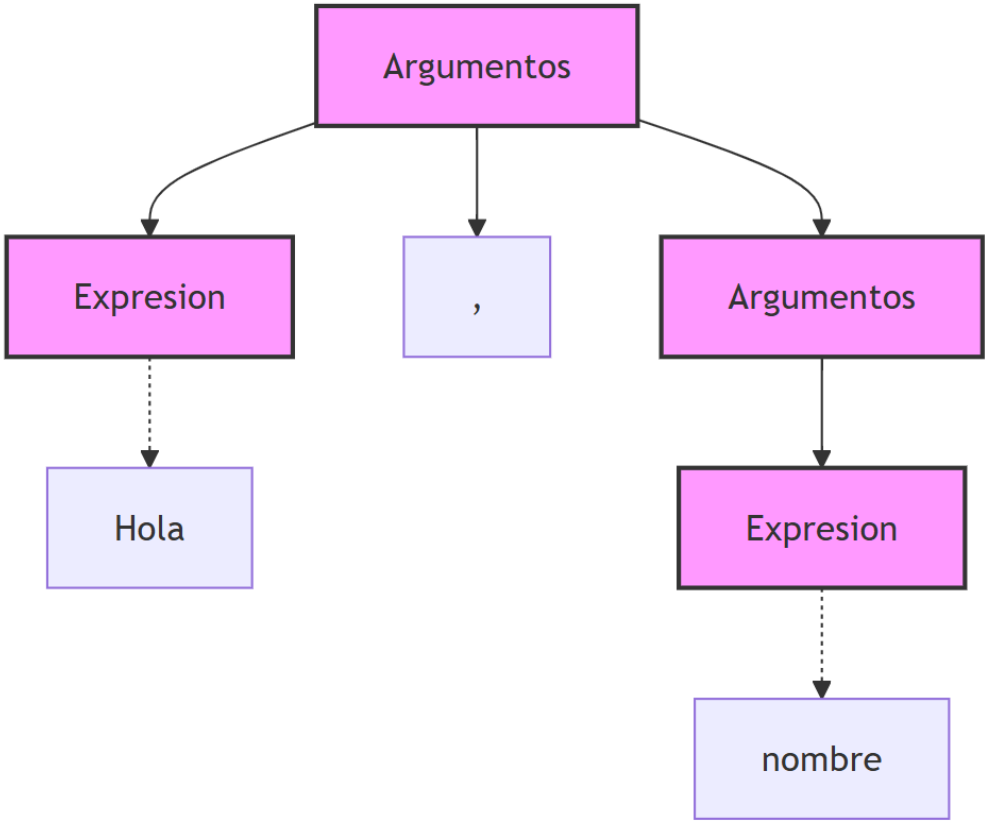


IMAGEN 63: ÁRBOL DE DERIVACIÓN G_{23} - ARGUMENTOS DE IMPRESIÓN.

9. AMBIGÜEDAD

Durante el diseño del analizador sintáctico **LL(1)**, se identificaron situaciones donde la gramática original presentaba ambigüedades o conflictos de decisión (elección de reglas). Dado que un parser descendente recursivo necesita decidir qué camino tomar basándose en el siguiente token disponible, se aplicaron dos estrategias principales para resolver estos conflictos: el uso de LOOKAHEAD y la reescritura de la gramática para eliminar la recursividad izquierda.

9.1 Conflicto Declaración vs. Asignación (Uso de LOOKAHEAD)

El Problema:

En la gramática de CarumaLang, tanto una Declaración de Variable como una Asignación pueden aparecer como sentencias válidas dentro de Declaraciones().

- Una declaración comienza con un *Tipo de Dato* (ej. intCHELADA).
- Una asignación comienza con un *Identificador* (ej. numero).

Debido a la estructura del lenguaje, el parser podría enfrentar ambigüedad si solo observa 1 token de anticipación, especialmente si el diseño del lenguaje permitiera solapamientos léxicos. Aunque conceptualmente los tokens son distintos, en la implementación práctica se optó por forzar la decisión para asegurar la robustez.

La Solución:

Se implementó LOOKAHEAD(2) en la regla Declaracion(). Esto instruye al parser a "mirar" dos tokens adelante antes de tomar una decisión.

- Si ve Tipo + Identificador (ej. intCHELADA x), sabe que es una **Declaración**.
- Si ve Identificador + = (ej. x =), sabe que es una **Asignación**.

Caso Similar en Estructura FOR:

La misma ambigüedad se presenta en la inicialización del bucle paraPapoi, donde se puede declarar una variable nueva (intCHELADA i = 0) o usar una existente (i = 0). Se resolvió igualmente con LOOKAHEAD(2) en la regla Inicializacion().

9.2 Recursividad por la Izquierda (Expresiones Aritméticas)

El Problema:

Las gramáticas formales para expresiones aritméticas suelen definirse con recursividad por la izquierda para respetar la asociatividad natural (ej. E -> E + T). Sin embargo, esta estructura es incompatible con los analizadores LL(1) descendentes recursivos como el generado por JavaCC, ya que provoca un bucle infinito (el parser intenta llamar a E indefinidamente sin consumir tokens).

Gramática Ambigua (Recursiva por Izquierda):

E -> E + T | T

La Solución:

Se resolvió transformando la gramática a una forma iterativa utilizando la Notación Extendida de Backus-Naur (EBNF). Se eliminó la recursividad directa sustituyéndola por cierres de Kleene (bucles *), lo que permite procesar múltiples términos secuenciales sin recursión infinita.

Gramática Resuelta (Iterativa para LL(1)):

E -> T \ (("+" | "-") T)*

Esta transformación se aplicó a todas las reglas de precedencia:

1. **Expresiones:** Manejan sumas y restas iterativamente.
2. **Términos:** Manejan multiplicaciones y divisiones iterativamente.

10. RECORRIDO

El analizador sintáctico de CarumaLang clasifica como un parser **Descendente Recursivo (Recursive Descent Parser)** que opera bajo la lógica **LL(1)**. Este diseño fue seleccionado por su eficiencia, facilidad de implementación manual y correspondencia directa con la estructura de las Gramáticas Libres de Contexto definidas.

10.1 Desglose de la Clasificación LL(1)

La denominación LL(1) describe matemáticamente el comportamiento del algoritmo de análisis:

- **L (Left-to-right):** El analizador escanea la cadena de entrada (tokens) de izquierda a derecha, consumiendo los símbolos en el orden en que aparecen en el código fuente.
- **L (Leftmost derivation):** Construye el árbol de análisis sintáctico generando siempre la *derivación más a la izquierda*. Esto significa que, al expandir los no-terminales, el parser siempre procesa el no-terminal que se encuentra más a la izquierda en la regla de producción actual.
- **(1):** Utiliza **un token de anticipación (Lookahead)** para determinar qué regla de producción aplicar. El parser observa el siguiente token disponible (jj_nt) sin consumirlo para decidir qué camino tomar en la gramática.

10.2 Implementación Descendente Recursiva

En la práctica, este tipo de recorrido se implementa mediante un conjunto de procedimientos (métodos en Java) que se llaman mutuamente de forma recursiva.

- **Correspondencia Método-Regla:** Cada **No Terminal** de la gramática (como Programa, Declaracion, EstructuraIf) tiene asociado un método específico en el código (void Programa(), void Declaracion(), etc.).
- **Flujo Top-Down:** El análisis comienza en el símbolo inicial (la raíz del árbol, Programa) y "desciende" hacia las hojas (los tokens terminales) llamando a los métodos correspondientes a medida que avanza la estructura.
- **Consumo de Terminales:** Cuando la gramática espera un símbolo terminal (un token como <CAECLIENTE> o <ABRIENDO>), el método llama a una función de consumo (jj_consume_token) que verifica si el token actual coincide. Si coincide, avanza al siguiente; si no, reporta un error.

10.3 Excepción: Lookahead Local

Aunque la arquitectura general es LL(1), existen puntos específicos en la gramática de CarumaLang donde un solo token de anticipación no es suficiente para desambiguar la instrucción (por ejemplo, al distinguir entre el inicio de una declaración de variable y una asignación).

Para estos casos, el parser utiliza una capacidad extendida de **LOOKAHEAD(2)**. Esto permite al analizador inspeccionar dos tokens adelante antes de tomar una decisión, resolviendo conflictos locales sin alterar la naturaleza descendente recursiva del sistema global.

11. MANEJO DE ERRORES

El compilador de CarumaLang implementa un sistema robusto de gestión de errores diseñado para identificar, clasificar y reportar múltiples fallos en una sola ejecución. A diferencia de un parser tradicional que se detiene ante la primera inconsistencia, este sistema utiliza una estrategia de recuperación conocida como **"Modo Pánico"** combinada con un **Pre-análisis de Estructura**.

11.1 Clasificación de Errores

El sistema distingue y gestiona dos categorías principales de errores:

1. **Errores Léxicos (TokenMgrError):**
    - **Origen:** Ocurren cuando el analizador léxico encuentra una secuencia de caracteres que no coincide con ningún patrón de token definido (por ejemplo, símbolos ilegales como @ o #).
    - **Manejo:** Aunque son detectados en la fase léxica, el analizador sintáctico los captura para incluirlos en el reporte unificado, permitiendo que el análisis continúe.
2. **Errores Sintácticos (ParseException):**
    - **Origen:** Ocurren cuando la secuencia de tokens válida no cumple con las reglas de producción de la gramática (por ejemplo, olvidar un punto y coma, una llave de cierre o escribir una instrucción en orden incorrecto).
    - **Manejo:** Se utiliza la clase personalizada ParserConRecuperacion para interceptar estas excepciones, registrar el error y resincronizar el análisis.

11.2 Estrategia de Recuperación (Modo Pánico)

Para evitar la terminación abrupta del análisis, se implementó la clase ParserConRecuperacion en el archivo AnalisisSintactico.java. Esta clase extiende el parser generado y añade mecanismos de tolerancia a fallos.

Mecanismo de Sincronización: Cuando se detecta un token inesperado, el parser entra en "estado de pánico" y ejecuta el método recuperarHastaToken(). Este método consume y descarta tokens de la entrada hasta encontrar un **Token de Sincronización** seguro.

- **Tokens de Sincronización:** Son elementos que indican inequívocamente el inicio o fin de una estructura mayor, permitiendo al parser "reubicarse". En CarumaLang, estos incluyen:
    - Inicio de declaraciones: intCHELADA, granito, cadena.
    - Estructuras de control: CaeCliente, papoi, paraPapoi.
    - Delimitadores finales: byebye.

11.3 Pre-análisis de Delimitadores

Una innovación clave en esta implementación es la fase de **Pre-análisis de Llaves**. Antes de ejecutar el parser sintáctico completo (LL(1)), el sistema realiza un recorrido ligero del archivo fuente exclusivamente para verificar el balanceo de bloques ({ y }).

- **Objetivo:** Los errores de llaves faltantes suelen causar "errores en cascada" que confunden al parser sintáctico, haciendo que reporte errores falsos en el resto del código.

- **Funcionamiento:** Se utiliza una pila para rastrear cada apertura de bloque y su contexto (si pertenece a un CaeCliente, papoi, etc.). Si al finalizar el archivo la pila no está vacía o se encuentra un cierre inesperado, se reporta el error específico antes de intentar analizar la lógica interna.

11.4 Estructura de Reporte y Tabla de Errores

Todos los errores detectados se almacenan en una estructura de datos unificada (List<ErrorSintactico>), que actúa como una **Tabla de Errores**. Cada registro en esta tabla contiene información detallada para facilitar la depuración:

- **Ubicación:** Línea y columna exactas donde ocurrió el fallo.
- **Token Encontrado:** El lexema que provocó el error.
- **Token Esperado:** La lista de tokens que la gramática permitía en esa posición (extraída del expectedTokenSequences de la excepción).
- **Mensaje Descriptivo:** Una explicación humana del error (ej. "Falta cierre de paréntesis").

Al finalizar la ejecución, si la tabla contiene elementos, el sistema genera dos salidas:

1. **Consola:** Un resumen tabular visual para el usuario.
2. **Archivo .errores:** Un reporte persistente y detallado.

12. MANEJO DE SÍMBOLOS

El analizador utiliza estructuras de datos en memoria para gestionar la información recopilada durante el proceso de compilación. Estas estructuras son fundamentales para el reporte de fallos y la validación de las declaraciones del programador.

12.1 Tabla de Errores (Implementación)

La "Tabla de Errores" no es una estructura estática, sino una colección dinámica que acumula las incidencias detectadas durante el recorrido del parser. Se implementó mediante la clase interna ErrorSintactico y una lista ArrayList.

- **Estructura del Registro:** Cada entrada en la tabla de errores es un objeto que encapsula toda la información necesaria para el diagnóstico:
    - mensaje: Descripción textual del problema.
    - linea y columna: Coordenadas exactas en el fichero fuente.
    - tokenEncontrado: El lexema que generó el conflicto.
    - tokenEsperado: El símbolo que la gramática anticipaba en esa posición.

Esta lista permite que el compilador, al finalizar el análisis, vuelque toda la información de manera estructurada tanto en la consola como en el archivo de reporte .errores, en lugar de abortar la ejecución con una excepción no controlada.

12.2 Tabla de Símbolos (Identificación Sintáctica)

En la fase de análisis sintáctico, el "Manejo de Símbolos" se centra en la identificación y validación estructural de los identificadores (variables). El parser verifica que cada símbolo introducido en el programa cumpla con las reglas de declaración antes de ser utilizado.

Validación de Declaraciones: El parser rastrea la creación de nuevos símbolos mediante la regla DeclaracionVariable. Esta regla asegura que todo símbolo (<MIXCHELADA>) esté precedido obligatoriamente por un tipo de dato válido (intCHELADA, granito, cadena, etc.).

- **Mecanismo:** Cuando el parser encuentra un token de Tipo, anticipa que el siguiente token debe ser un identificador. Esta secuencia Tipo -> ID constituye la entrada básica para lo que posteriormente sería la Tabla de Símbolos Semántica.
- **Gramática:** La regla asegura que se pueden declarar múltiples símbolos del mismo tipo separados por comas (ej. intCHELADA x, y, z), validando la estructura de la lista de identificadores.