UNIVERDIDAD MARIANO GÁLVEZ DE GUATEMALA

Ingeniería en sistemas de la información y ciencias de la computación

Kevin Roberto Lacán Tzul

1490-21-13789

Cécily Karina Guinea Mejía

1490-21-3264

Trabajo: Proyecto Fase 2

6to. Semestre

Contenido

INTRODUCCIÓN

El presente proyecto consiste en el desarrollo de un **Analizador Léxico**, una herramienta fundamental en el proceso de compilación de lenguajes de programación. Su objetivo principal es leer un código fuente escrito en un lenguaje específico y descomponerlo en una serie de tokens, que son las unidades léxicas más pequeñas con significado. Estos tokens son utilizados posteriormente por el analizador sintáctico para construir la estructura del programa.

El analizador léxico implementado en este proyecto está diseñado para reconocer una variedad de elementos del lenguaje, incluyendo:

- **Números**: Soporta enteros cortos y largos, así como números en formato monetario y real.
- **Cadenas de texto**: Permite la identificación de cadenas delimitadas por comillas dobles.
- **Fechas**: Reconoce fechas en el formato **'dd\mm\yyyy'**, lo que permite su uso en aplicaciones que requieren manipulación de datos temporales.
- **Identificadores**: Permite la identificación de variables y funciones, siguiendo las reglas de nomenclatura del lenguaje.
- **Palabras reservadas**: Reconoce las palabras clave del lenguaje, que tienen un significado especial y no pueden ser utilizadas como identificadores.
- **Operadores y delimitadores**: Identifica operadores aritméticos, lógicos y de comparación, así como delimitadores como paréntesis, llaves y punto y coma.

El analizador está implementado en Java y cuenta con una interfaz gráfica de usuario (GUI) que permite a los usuarios ingresar el código fuente y visualizar los resultados del análisis léxico de manera clara y organizada. La herramienta proporciona retroalimentación inmediata sobre los tokens reconocidos, facilitando así el proceso de depuración y comprensión del código.

Este proyecto no solo tiene aplicaciones en el ámbito académico, sino que también puede ser utilizado como base para el desarrollo de lenguajes de programación personalizados o para la creación de herramientas de análisis de código en entornos de desarrollo.

A través de este documento, se detallarán las características del analizador léxico, su funcionamiento interno, así como ejemplos de uso y posibles extensiones futuras.

JUSTIFICACIÓN

La creación de un analizador léxico es un paso fundamental en el desarrollo de lenguajes de programación y herramientas de análisis de código. La justificación de este proyecto se basa en varios aspectos clave que destacan su relevancia y utilidad:

1. **Fundamento de la Compilación**: El análisis léxico es la primera fase del proceso de compilación, donde el código fuente se transforma en una secuencia de tokens. Esta transformación es esencial para que el compilador pueda entender y procesar el código de manera efectiva. Al desarrollar un analizador léxico, se sienta la base para la creación de compiladores y otros sistemas de procesamiento de lenguajes.
2. **Facilitación del Aprendizaje**: Este proyecto proporciona una herramienta educativa valiosa para estudiantes y desarrolladores que deseen comprender los principios del análisis de lenguajes. Al implementar un analizador léxico, los usuarios pueden aprender sobre la estructura de los lenguajes de programación, las reglas de formación de tokens y la importancia de la sintaxis en la programación.
3. **Mejora de la Productividad**: Al contar con un analizador léxico, los desarrolladores pueden identificar y corregir errores en el código fuente de manera más eficiente. La retroalimentación inmediata sobre los tokens reconocidos permite a los programadores detectar problemas de sintaxis y semántica en etapas tempranas del desarrollo, lo que reduce el tiempo de depuración y mejora la calidad del código.
4. **Personalización y Extensibilidad**: Este analizador léxico está diseñado para ser flexible y extensible, lo que permite a los usuarios adaptarlo a sus necesidades específicas. Los desarrolladores pueden modificar y ampliar el analizador para soportar nuevos tipos de tokens, palabras reservadas y reglas de sintaxis, lo que lo convierte en una herramienta versátil para el desarrollo de lenguajes personalizados.
5. **Aplicaciones Prácticas**: Además de su uso educativo, el analizador léxico tiene aplicaciones prácticas en la industria del software. Puede ser utilizado en la creación de herramientas de análisis de código, editores de texto avanzados y entornos de desarrollo integrados (IDEs) que requieren un análisis sintáctico y semántico del código fuente.
6. **Contribución a la Comunidad**: Al documentar y compartir este proyecto, se contribuye al conocimiento colectivo en el campo del desarrollo de lenguajes de programación y herramientas de análisis. Otros desarrolladores pueden

beneficiarse de la implementación y los conceptos presentados, fomentando la colaboración y el intercambio de ideas en la comunidad de programación.

BREVE EXPLICACIÓN DE LA PRIMERA FASE Y AJUSTES REALIZADOS

En la primera fase del desarrollo del analizador léxico, se implementó un conjunto inicial de tokens que, tras la revisión y realimentación del ingeniero a cargo del proyecto, resultaron ser insuficientes y, en algunos casos, incorrectos. Esta etapa inicial se centró en la identificación de elementos léxicos básicos, como números, identificadores y operadores, utilizando patrones de expresiones regulares que no abarcaban completamente la diversidad y complejidad del lenguaje objetivo.

Contexto de la Primera Fase

Durante la implementación inicial, se definieron patrones de tokens que, aunque funcionales en un contexto limitado, no consideraban adecuadamente las variaciones y excepciones que pueden presentarse en un código fuente real. Por ejemplo, se identificaron problemas en la detección de ciertos tipos de números, cadenas y fechas, lo que llevó a la generación de tokens erróneos o a la omisión de tokens válidos. Esta situación no solo comprometía la precisión del análisis léxico, sino que también afectaba la calidad del análisis sintáctico posterior.

Ajustes Realizados

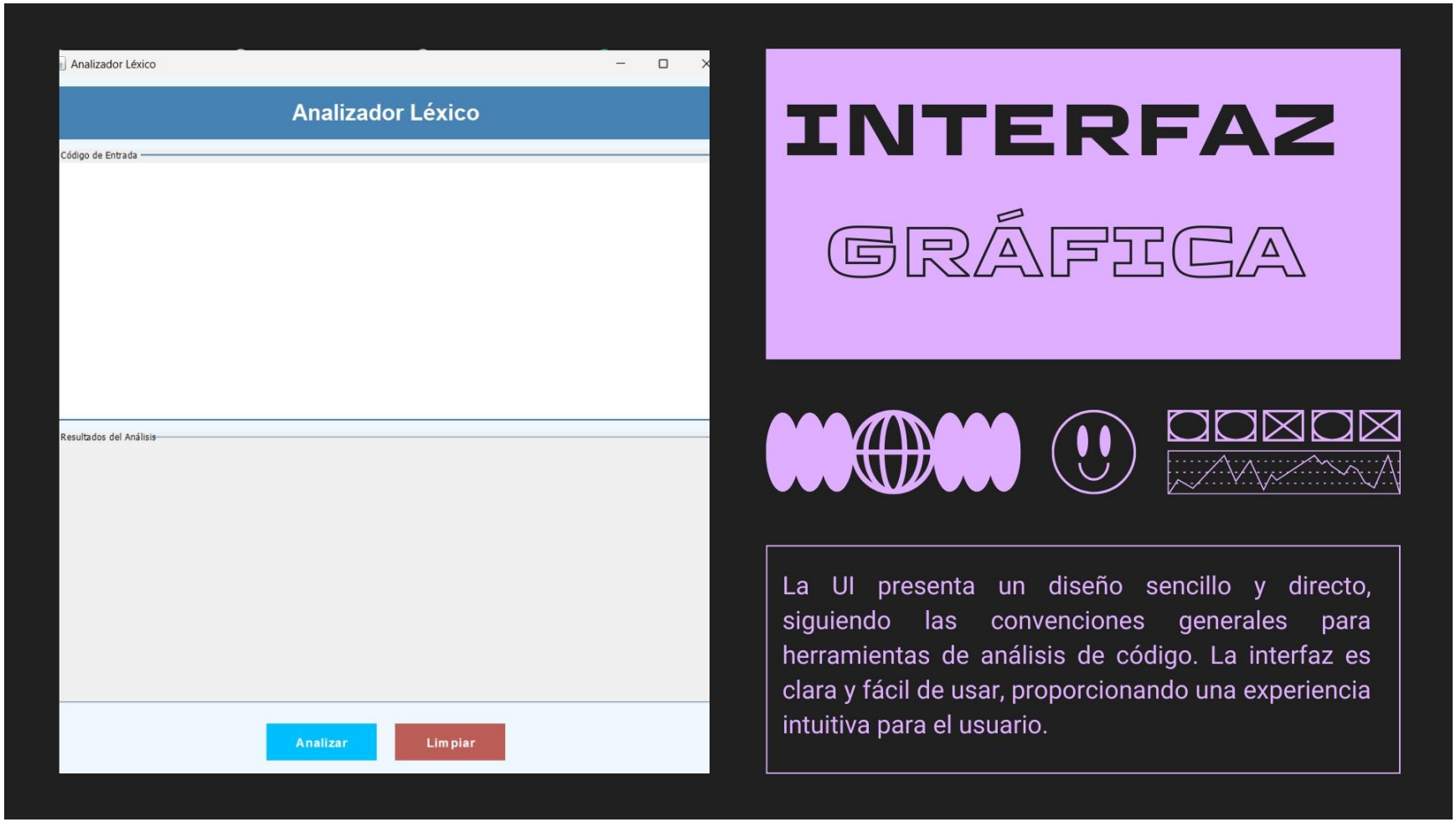
A raíz de la retroalimentación recibida, se llevó a cabo un replanteamiento exhaustivo de la lógica de tokenización. Los ajustes realizados incluyeron:

1. **Revisión de Patrones de Expresión Regular**: Se revisaron y ajustaron los patrones de expresiones regulares utilizados para la identificación de tokens. Se incorporaron nuevos patrones que permiten una mejor detección de números en diferentes formatos, así como de cadenas y fechas, asegurando que se alineen con las especificaciones del lenguaje.
2. **Ampliación de la Cobertura de Tokens**: Se amplió la lista de tokens reconocidos para incluir elementos adicionales, como palabras reservadas, comentarios y delimitadores. Esto no solo mejora la precisión del análisis, sino que también permite una mayor flexibilidad en el manejo de diferentes estilos de codificación.
3. **Validación y Pruebas Exhaustivas**: Se implementaron pruebas exhaustivas para validar la correcta identificación de tokens en una variedad de casos de prueba, incluyendo ejemplos de código que abarcan diferentes estructuras y estilos. Esto garantizó que el analizador léxico pudiera manejar adecuadamente situaciones complejas y errores comunes en el código fuente.
4. **Documentación y Comentarios**: Se mejoró la documentación interna del código para facilitar la comprensión de la lógica de tokenización y los patrones

utilizados. Esto es fundamental para el mantenimiento futuro del analizador y para facilitar la colaboración con otros desarrolladores.

Estos ajustes no solo han permitido mejorar la precisión y la robustez del analizador léxico, sino que también han sentado las bases para un análisis sintáctico más efectivo en las fases posteriores del proyecto. La realimentación del ingeniero ha sido invaluable en este proceso, guiando la evolución del analizador hacia una herramienta más completa y confiable.

CAPTURAS DEL SOFTWARE CREADO, CON UNA BREVE EXPLICACIÓN



Código de Entrada

```
123_c
456_1
Q123456
15:30
0xFF
'31\12\2023'
"Hola Mundo!"
inicio
fin
true false
miVariable_1
*/
+-
abc
+=
||
```

CÓDIGO DE ENTRADA

El código de entrada muestra una serie de elementos que podrían ser interpretados como variables, valores o incluso palabras clave en un lenguaje de programación.

Resultados del Análisis

Línea	Columna	Tipo	Valor
1	0	Entero Corto	123_c
2	0	ENTERO_LARGO	456_l
3	0	Valor Monetario	Q123456
4	0	Hora	15:30
5	0	Número Hexadecimal	0xFF
6	0	Fecha	'31\12\2023'
7	0	Cadena de Texto	"Hola Mundo!"
8	0	INICIO	inicio
9	0	FIN	fin
10	1	BOOLEANO	true

RESULTADO DE ANÁLISIS

La tabla muestra diferentes líneas de análisis, cada una con su respectivo tipo de dato y valor. esto es el resultado que se ingreso en el espacio anterior.

CÓDIGO FUENTE

Este código representa el componente principal del analizador léxico, el cual se encarga del procesamiento y clasificación de tokens. Su funcionamiento está estrechamente integrado con otras clases complementarias que se encuentran en la carpeta adjunta del proyecto. La interrelación entre estos componentes es fundamental e importante. Caber recalcar que para una ejecución exitosa del programa es necesario mantener la estructura de archivos y la relación entre las clases complementarias, ya que estas proporcionan funcionalidades esenciales para el proceso de análisis.

```
package modelo;

import java.util.*;
import java.util.regex.Pattern;

public class AnalizadorManager {
    // Patrones básicos según las convenciones
    private static final String L = "[a-zA-Z]";
    private static final String M = "[A-Z]";
    private static final String m = "[a-z]";
    private static final String N = "[0-9]";
    private static final String H = "[0-9A-Fa-f]";

    // Patrones de expresiones regulares para cada tipo de token
    private static final Pattern PATRON_REAL_CIENTIFICO = Pattern.compile("[0-9]+(\\.[0-9]+)?[Ee][+-]?[0-9]+");
    private static final Pattern PATRON_ENTERO = Pattern.compile(N + "+");
    private static final Pattern PATRON_BOOLEANO = Pattern.compile("true|false");
```

```
    private static final Pattern PATRON_ENTERO_CORTO = Pattern.compile(N + "+_c");

    private static final Pattern PATRON_ENTERO_LARGO = Pattern.compile(N + "+_l");

    private static final Pattern PATRON_DINERO = Pattern.compile("Q" + N + "+(?:," + N + "{3})*(?:\\." + N + "{2})?");

    private static final Pattern PATRON_HORA = Pattern.compile("(?:[01]" + N + "|2[0-3]):[0-5]" + N);

    private static final Pattern PATRON_HEXADECIMAL = Pattern.compile("0[xX]" + H + "+");

    private static final Pattern PATRON_REAL = Pattern.compile("[0-9]+\\.[0-9]+");

    private static final Pattern PATRON_FECHA = Pattern.compile("'(0?[1-9]|[12][0-9]|3[01])\\\\(0?[1-9]|1[012])\\\\[0-9]{4}'");

    private static final Pattern PATRON_IDENTIFICADOR = Pattern.compile("(" + L + "|_)(" + L + "|" + N + "|_)*");

    private static final Pattern PATRON_CADENA = Pattern.compile("\"[^\"]*\"");

    private static final Pattern PATRON_INICIO = Pattern.compile("inicio");

    private static final Pattern PATRON_FIN = Pattern.compile("fin");


    // Conjuntos de operadores

    private static final Set<String> OPERADORES_ADITIVOS = Set.of("+", "-");

    private static final Set<String> OPERADORES_MULTIPLICATIVOS = Set.of("*", "/", "%");

    private static final Set<String> OPERADORES_RELACIONALES = Set.of("<", ">", "<=", ">=", "==", "!=");

    private static final Set<String> OPERADORES_LOGICOS = Set.of("&&", "||", "!");

    private static final Set<String> OPERADORES_ASIGNACION = Set.of("=", "+=", "-=", "*=", "/=", "%=");

    private static final Set<String> DELIMITADORES = Set.of("{", "}", "[", "]", ",", ";", ".");
```

```
    public List<Token> analizar(String codigo) {
        List<Token> tokens = new ArrayList<>();
        String[] lineas = codigo.split("\n");

        for (int numLinea = 0; numLinea < lineas.length; numLinea++) {
            String linea = lineas[numLinea].trim();
            if (!linea.isEmpty()) {
                analizarLinea(linea, numLinea + 1, tokens);
            }
        }

        return tokens;
    }

    private void analizarLinea(String linea, int numeroLinea, List<Token>
tokens) {
        StringBuilder tokenActual = new StringBuilder();
        boolean enCadena = false;
        int columnaActual = 0;

        for (int i = 0; i < linea.length(); i++) {
            char c = linea.charAt(i);
            columnaActual++;

            if (c == "") {
                if (!enCadena) {
                    procesarTokenPendiente(tokenActual,          numeroLinea,
columnaActual - tokenActual.length(), tokens);
                    enCadena = true;
                } else {
```

```
                tokenActual.append(c);
                procesarToken(tokenActual.toString(), numeroLinea, columnaActual - tokenActual.length(), tokens);
                tokenActual.setLength(0);
                enCadena = false;
                continue;
            }
        }

        if (enCadena) {
            tokenActual.append(c);
            continue;
        }

        if (Character.isWhitespace(c)) {
            procesarTokenPendiente(tokenActual, numeroLinea, columnaActual - tokenActual.length(), tokens);
        } else if (esCaracterEspecial(c)) {
            procesarTokenPendiente(tokenActual, numeroLinea, columnaActual - tokenActual.length(), tokens);
            if (c == '.' && !tokenActual.isEmpty() && Character.isDigit(tokenActual.charAt(tokenActual.length() - 1))) {
                tokenActual.append(c);
            } else {
                procesarOperadorCompuesto(c, linea, i, numeroLinea, columnaActual, tokens);
            }
        } else {
            tokenActual.append(c);
        }
    }
```

```
        procesarTokenPendiente(tokenActual, numeroLinea, columnaActual - tokenActual.length(), tokens);
    }

    private void procesarTokenPendiente(StringBuilder tokenActual, int numeroLinea, int columna, List<Token> tokens) {
        if (tokenActual.length() > 0) {
            procesarToken(tokenActual.toString(), numeroLinea, columna, tokens);
            tokenActual.setLength(0);
        }
    }

    private void procesarToken(String token, int numeroLinea, int columna, List<Token> tokens) {
        if (token.isEmpty()) return;

        // Verificar primero números científicos y reales
        if (PATRON_REAL_CIENTIFICO.matcher(token).matches()) {
            tokens.add(new Token("REAL_CIENTIFICO", token, numeroLinea, columna));
        }
        // Verificar identificadores especiales y palabras reservadas
        else if (PATRON_INICIO.matcher(token).matches()) {
            tokens.add(new Token("INICIO", token, numeroLinea, columna));
        }
        else if (PATRON_FIN.matcher(token).matches()) {
            tokens.add(new Token("FIN", token, numeroLinea, columna));
        }
        else if (token.equals("if") || token.equals("else")) {
```

```
            tokens.add(new Token("PALABRA_RESERVADA", token, numeroLinea, columna));
        }
        // Verificar tipos numéricos específicos
        else if (PATRON_ENTERO_CORTO.matcher(token).matches()) {
            tokens.add(new Token("ENTERO_CORTO", token, numeroLinea, columna));
        }
        else if (PATRON_ENTERO_LARGO.matcher(token).matches()) {
            tokens.add(new Token("ENTERO_LARGO", token, numeroLinea, columna));
        }
        else if (PATRON_DINERO.matcher(token).matches()) {
            tokens.add(new Token("DINERO", token, numeroLinea, columna));
        }
        else if (PATRON_HORA.matcher(token).matches()) {
            tokens.add(new Token("HORA", token, numeroLinea, columna));
        }
        else if (PATRON_HEXADECIMAL.matcher(token).matches()) {
            tokens.add(new Token("HEXADECIMAL", token, numeroLinea, columna));
        }
        else if (PATRON_REAL.matcher(token).matches()) {
            tokens.add(new Token("REAL", token, numeroLinea, columna));
        }
        else if (PATRON_FECHA.matcher(token).matches()) {
            tokens.add(new Token("FECHA", token, numeroLinea, columna));
        }
        else if (PATRON_ENTERO.matcher(token).matches()) {
```

```
            tokens.add(new Token("ENTERO", token, numeroLinea, columna));
        }
        else if (PATRON_CADENA.matcher(token).matches()) {
            tokens.add(new Token("CADENA", token, numeroLinea, columna));
        }
        else if (PATRON_BOOLEANO.matcher(token).matches()) {
            tokens.add(new Token("BOOLEANO", token, numeroLinea, columna));
        }
        else if (PATRON_IDENTIFICADOR.matcher(token).matches()) {
            tokens.add(new Token("IDENTIFICADOR", token, numeroLinea, columna));
        }
        else {
            String tipoOperador = getTipoOperador(token);
            if (!tipoOperador.equals("NO_RECONOCIDO")) {
                tokens.add(new Token(tipoOperador, token, numeroLinea, columna));
            } else {
                tokens.add(new Token("NO_RECONOCIDO", token, numeroLinea, columna));
            }
        }
    }

    private void procesarOperadorCompuesto(char c, String linea, int pos, int numeroLinea, int columna, List<Token> tokens) {
        String operador = String.valueOf(c);

        // Verificar primero si es un paréntesis
```

```
        if (operador.equals("(")) {
            tokens.add(new Token("PARENTESIS_INICIO", operador,
numeroLinea, columna));
            return;
        }
        if (operador.equals(")")) {
            tokens.add(new Token("PARENTESIS_FINAL", operador,
numeroLinea, columna));
            return;
        }

        if (pos + 1 < linea.length()) {
            char nextChar = linea.charAt(pos + 1);
            String posibleOperador = operador + nextChar;

            if  (OPERADORES_RELACIONALES.contains(posibleOperador)
||
                OPERADORES_LOGICOS.contains(posibleOperador) ||
                OPERADORES_ASIGNACION.contains(posibleOperador)) {
                tokens.add(new Token(getTipoOperador(posibleOperador),
posibleOperador, numeroLinea, columna));
                return;
            }
        }

        if (OPERADORES_ADITIVOS.contains(operador) ||
            OPERADORES_MULTIPLICATIVOS.contains(operador) ||
            OPERADORES_RELACIONALES.contains(operador) ||
            DELIMITADORES.contains(operador)) {
            tokens.add(new Token(getTipoOperador(operador), operador,
numeroLinea, columna));
```

```
        }
    }

    private String getTipoOperador(String operador) {
        if (OPERADORES_ADITIVOS.contains(operador)) return
"OPERADOR_ADITIVO";
        if (OPERADORES_MULTIPLICATIVOS.contains(operador)) return
"OPERADOR_MULTIPLICATIVO";
        if (OPERADORES_RELACIONALES.contains(operador)) return
"OPERADOR_RELACIONAL";
        if (OPERADORES_LOGICOS.contains(operador)) return
"OPERADOR_LOGICO";
        if (OPERADORES_ASIGNACION.contains(operador)) return
"OPERADOR_ASIGNACION";
        if (DELIMITADORES.contains(operador)) return "DELIMITADOR";
        if (operador.equals("(")) return "PARENTESIS_INICIO";
        if (operador.equals(")")) return "PARENTESIS_FINAL";
        return "NO_RECONOCIDO";
    }

    private boolean esCaracterEspecial(char c) {
        return "+-*/<>=!&|(){}[],.;%".indexOf(c) != -1;
    }
}
```

Formalización de la gramática regular

Expresiones regulares principales

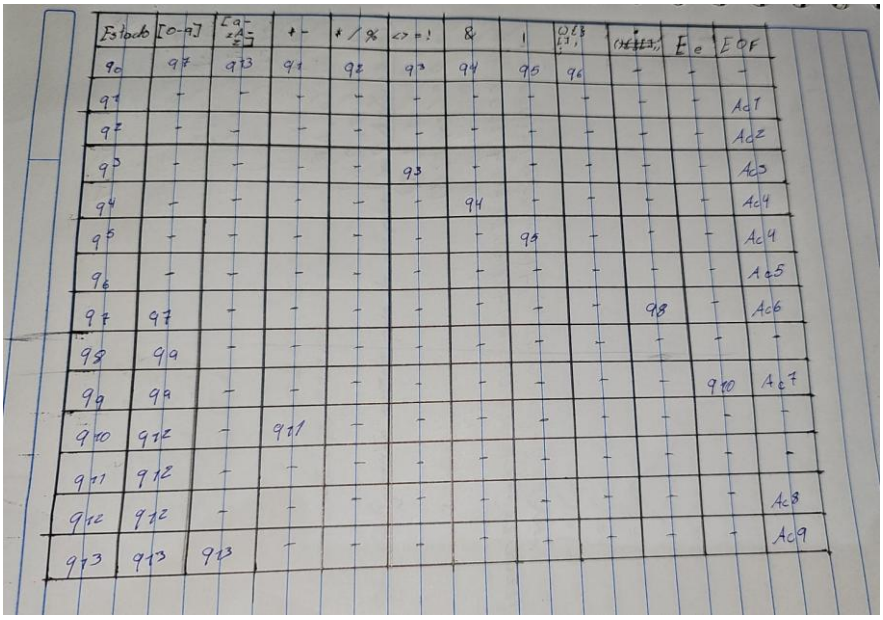
- ENTERO: [0-9]+
- REAL: [0-9]+\.[0-9]+
- REAL_CIENTIFICO: [0-9]+\.[0-9]+[eE][+-]?[0-9]+
- IDENTIFICADOR: [a-zA-Z_][a-zA-Z0-9_]*
- OPERADOR_ADITIVO: [+-]
- OPERADOR_MULTIPLICATIVO: [*/%]
- OPERADOR_RELACIONAL: [<>]=?|==|!=
- OPERADOR_LOGICO: &&|\|\|
- DELIMITADOR: [()\[\]{},;]

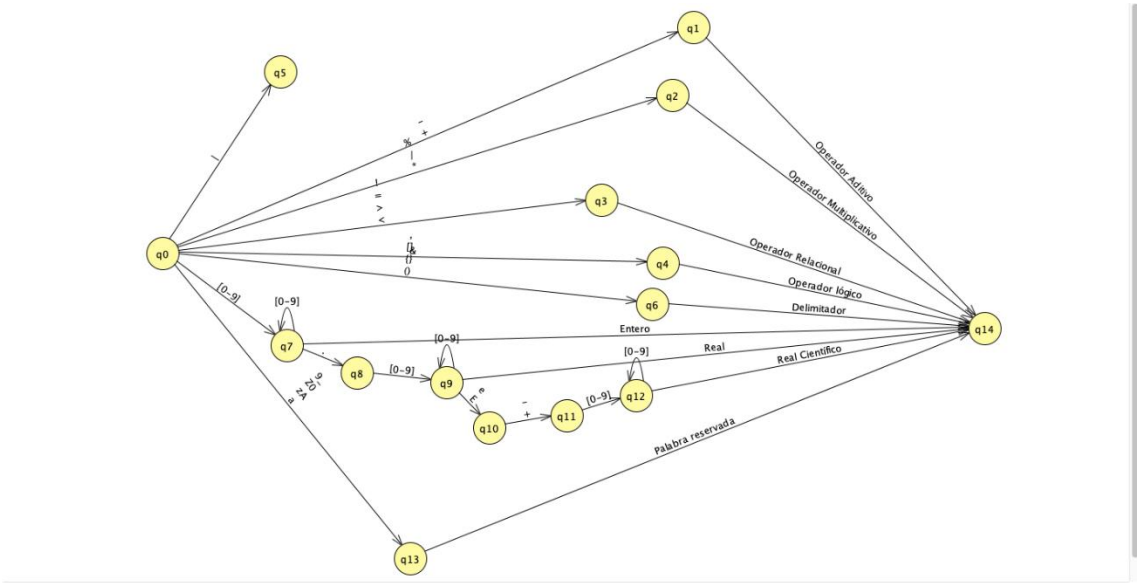
Convenciones:

- Los estados están etiquetados como q0, q1, q2, etc.
- El estado inicial es q0
- Las transiciones están etiquetadas con los símbolos de entrada
- Los estados finales están marcados y asociados con el tipo de token

3. Proceso de Conversión de AFN a AFD:

AFN



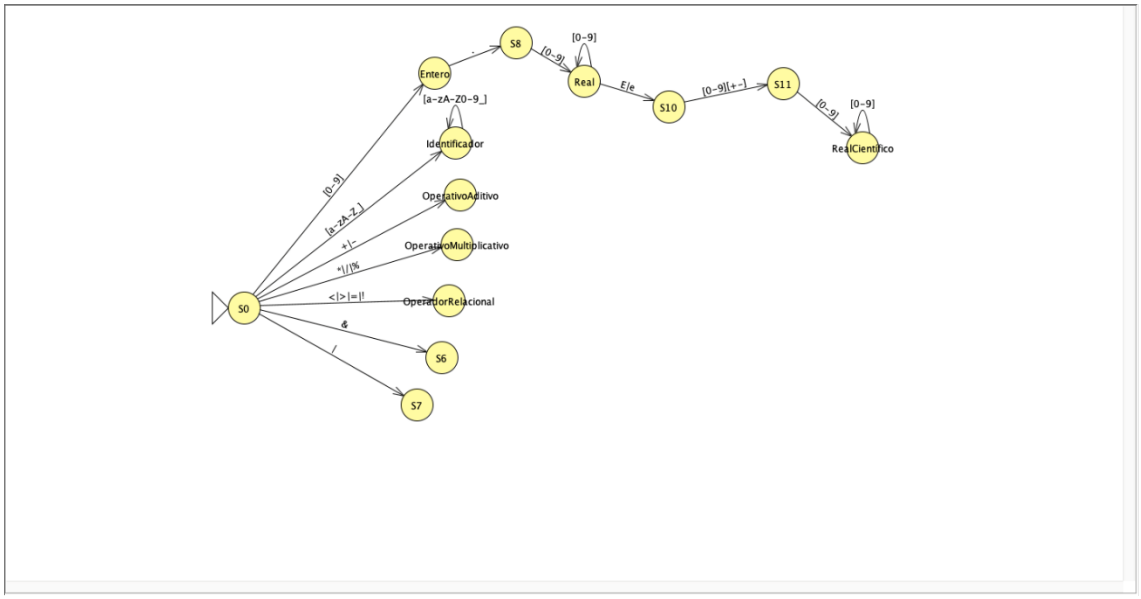


Donde:

- Ac1: Acepta OPERADOR_ADITIVO
- Ac2: Acepta OPERADOR_MULTIPLICATIVO
- Ac3: Acepta OPERADOR_RELACIONAL
- Ac4: Acepta OPERADOR_LOGICO
- Ac5: Acepta DELIMITADOR
- Ac6: Acepta ENTERO
- Ac7: Acepta REAL
- Ac8: Acepta REAL_CIENTIFICO

- Ac9: Acepta IDENTIFICADOR/PALABRA_RESERVADA

AFD



Donde:

- Ac1: Acepta ENTERO
- Ac2: Acepta IDENTIFICADOR
- Ac3: Acepta OPERADOR_ADITIVO
- Ac4: Acepta OPERADOR_MULTIPLICATIVO
- Ac5: Acepta OPERADOR_RELACIONAL
- Ac6: Acepta OPERADOR_LOGICO
- Ac7: Acepta DELIMITADOR
- Ac8: Acepta REAL
- Ac9: Acepta REAL_CIENTIFICO

CONCLUSIONES

1. El desarrollo del analizador léxico demuestra la importancia fundamental de la fase de análisis léxico en el proceso de compilación, permitiendo la identificación y clasificación precisa de los diferentes tipos de tokens.
2. La implementación del programa logró reconocer exitosamente diversos tipos de datos como enteros, valores monetarios, fechas, horas y cadenas de texto, cumpliendo con los objetivos planteados inicialmente.
3. La estructura modular del código facilita el mantenimiento y la escalabilidad del sistema, permitiendo la incorporación de nuevos tipos de tokens según sea necesario.
4. El manejo de errores implementado en el analizador permite una identificación clara de problemas en el código fuente, facilitando la depuración y corrección.
5. La integración entre las diferentes clases del proyecto demuestra la importancia de una arquitectura bien planificada en el desarrollo de compiladores.

RECOMENDACIONES

Se recomienda:

1. Implementar el reconocimiento de números decimales en el analizador léxico para ampliar la capacidad de procesamiento de datos numéricos, lo cual permitirá manejar una gama más amplia de valores en las aplicaciones futuras.
2. Desarrollar un sistema más robusto para el manejo de errores que incluya la identificación precisa de la línea donde ocurre el error, facilitando así la depuración del código fuente analizado.
3. Optimizar los algoritmos de procesamiento de tokens monetarios y fechas, con el fin de mejorar el rendimiento general del analizador léxico cuando procese archivos de gran tamaño.
4. Reestructurar el código fuente implementando una mejor separación de responsabilidades en los métodos de validación, lo que facilitará el mantenimiento y la escalabilidad del sistema.
5. Incorporar el reconocimiento de comentarios multilínea en el analizador léxico, permitiendo así un análisis más completo del código fuente y mejorando la compatibilidad con diferentes estilos de programación.