MANUAL TÉCNICO

ANALIZADOR LÉXICO
PROYECTO 1 LENGUAJES FORMALES
APLICACIÓN DE ESCRITORIO

CRISTIAN ALEJANDRO ROLDÁN LÓPEZ
202147280

Especificaciones

Versión de Java utilizado: Java 21.
Versión de JDK: 21
Sistema Operativo Utilizado: Ubuntu 24.04 LTS
IDE utilizado: Apache NetBeans IDE 21.

Metodo para obtener tokens validos de css

```
public List<Token> obtenerTokensValidosCSS(String linea) {
        List<Token> tokens = new ArrayList<>();
        int i = 0;

        while (i < linea.length()) {
            char actual = linea.charAt(i);

            // Ignorar espacios
            if (Character.isWhitespace(actual)) {
                i++;
                continue;
            }

            // Procesar comas, dos puntos, punto y coma
            if (actual == ',') {
                tokens.add(new Token(",", "", "CSS", "Coma", 0, 0));
                i++;
                continue;
            }
            if (actual == ':') {
                tokens.add(new Token(":", "", "CSS", "Dos puntos", 0,
0));
                i++;
                continue;
            }
            if (actual == ';') {
                tokens.add(new Token(";", "", "CSS", "Punto y coma",
0, 0));
                i++;
                continue;
            }

            // Procesar comillas simples
```

```
            if (actual == '\') {
                int inicio = i;
                i++;
                while (i < linea.length() && linea.charAt(i) != '\') {
                    i++;
                }
                if (i < linea.length()) {
                    String contenido = linea.substring(inicio, i + 1);
                    tokens.add(new Token(contenido, "", "CSS",
"Cadena", 0, 0));
                    i++;
                }
                continue;
            }

            // Procesar colores hexadecimales y rgba
            if (actual == '#') {
                int inicio = i;
                i++;
                while (i < linea.length() &&
(Character.isDigit(linea.charAt(i)) ||
                                        (linea.charAt(i) >= 'a' &&
linea.charAt(i) <= 'f'))) {
                    i++;
                }
                // Validar si es un color hexadecimal
                String hexColor = linea.substring(inicio, i);
                if (hexColor.length() == 4 || hexColor.length() == 7) {
                    tokens.add(new Token(hexColor, "", "CSS", "Color
Hexadecimal", 0, 0));
                    continue;
                }
            }

            // Procesar rgba
```

```
            if (linea.startsWith("rgba(", i)) {
                int inicio = i;
                i += 5; // Saltar "rgba("
                int count = 0;
                StringBuilder rgbaBuilder = new StringBuilder("rgba(");
                while (i < linea.length() && count < 4) {
                    char c = linea.charAt(i);
                    rgbaBuilder.append(c);
                    if (c == ',' || c == ')') {
                        count++;
                    }
                    i++;
                }
                if (count == 4) {
                    tokens.add(new Token(rgbaBuilder.toString(), "",
"CSS", "Color RGBA", 0, 0));
                    continue;
                }
                i = inicio; // Reiniciar si no se validó
            }

            // Identificar y procesar selectores
            if (actual == '.' || actual == '#') {
                int inicio = i;
                i = procesarSelector(linea, i, tokens);
                if (i == -1) return tokens; // Error en el procesamiento
                continue;
            }

            // Procesar llaves
            if (actual == '{') {
                tokens.add(new Token("{", "", "CSS", "Llave Apertura",
0, 0));
                i++;
                continue;
```

```
            }
            if (actual == '}') {
                tokens.add(new Token("}", "", "CSS", "Llave Cierre", 0,
0));
                i++;
                continue;
            }

            // Procesar enteros
            if (Character.isDigit(actual)) {
                int inicio = i;
                while (i < linea.length() &&
Character.isDigit(linea.charAt(i))) {
                    i++;
                }
                String entero = linea.substring(inicio, i);
                tokens.add(new Token(entero, "", "CSS", "Entero", 0,
0));
                continue;
            }

            // Procesar combinadores
            if (actual == '>' || actual == '+' || actual == '~') {
                tokens.add(new Token(String.valueOf(actual), "",
"CSS", "Combinador", 0, 0));
                i++;
                continue;
            }

            // Procesar identificadores
            if (Character.isLetter(actual)) {
                int inicio = i;
                while (i < linea.length() &&
(Character.isLetter(linea.charAt(i)) ||
Character.isDigit(linea.charAt(i)) || linea.charAt(i) == '-')) {
```

```
                i++;
            }
            String identificador = linea.substring(inicio, i);
            // Validar si es un selector de ID
            if (identificador.startsWith("#") &&
identificador.matches("#[a-z][a-z0-9-]*")) {
                tokens.add(new Token(identificador, "", "CSS",
"Selector ID", 0, 0));
            } else {
                tokens.add(new Token(identificador, "", "CSS",
"Identificador", 0, 0));
            }
            continue;
        }

        // Si llegamos aquí, hemos encontrado un carácter no
reconocido
        i++;
    }
```

Se crea una lista donde se guardaran los tokens validos de css.

Clase donde se detectan los tokens validos de js

```
package AnalizadorJS;

import Reportes.Token;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class IdentificadorJS {

    private String tokenEstadoJS = ">>[js]";
```

```
    private String[] palabrasReservadasJS = {
        "function", "const", "let", "document", "event", "alert",
        "for", "while", "if", "else", "return", "console.log", "null",
"target", "value"
    };

    // Operadores Aritméticos
    private final String[] operadoresAritmeticos = {"+", "-", "*",
"/"};

    // Operadores Relacionales
    private final String[] operadoresRelacionales = {"==", "<", ">",
"<=", ">=", "!="};

    // Operadores Lógicos
    private final String[] operadoresLogicos = {"||", "&&", "!"};

    // Operadores Incrementales
    private final String[] operadoresIncrementales = {"++", "--"};

    // Otros símbolos
    private final String[] otrosSimbolos = {"(", ")", "{", "}", "[", "]",
"=", ";", ",", ".", ":", "\"", "'", "`"};

    public String analizarJS(String cadena) {
        StringBuilder resuJs = new StringBuilder();

        // Inicializar el índice de búsqueda
        int startIndex = 0;

        // Buscar bloques de código JavaScript
        while ((startIndex = cadena.indexOf(tokenEstadoJS,
startIndex)) != -1) {
```

```
            // Extraer solo el contenido después de la etiqueta de
estado
            startIndex += tokenEstadoJS.length();
            int endIndex = cadena.indexOf(">>", startIndex); //
Suponiendo que el bloque JS finaliza con ">>"

            String contenidoJS;
            if (endIndex != -1) {
                contenidoJS = cadena.substring(startIndex,
endIndex).trim();
                startIndex = endIndex + 2; // Avanza después del bloque
encontrado
            } else {
                contenidoJS = cadena.substring(startIndex).trim(); //
Captura hasta el final si no se encuentra el final
                startIndex = cadena.length(); // Marca el final de la
cadena
            }

            if (!contenidoJS.isEmpty()) {
                resuJs.append("Lenguaje JavaScript detectado \n");
                resuJs.append(analizarTokensJS(contenidoJS)); //
Analizar el contenido
            } else {
                resuJs.append("Error: No se encontró contenido
JavaScript después de la etiqueta de estado.\n");
            }
        }

        if (resuJs.length() == 0) {
            resuJs.append("Error: Lenguaje no detectado, se requiere
la etiqueta >>[js].\n");
        }

        return resuJs.toString();
```

```
    }

    public List<Token> obtenerTokensValidosJS(String linea) {
        List<Token> tokens = new ArrayList<>();
        int i = 0;

        while (i < linea.length()) {
            // Saltar espacios en blanco
            while (i < linea.length() &&
Character.isWhitespace(linea.charAt(i))) {
                i++;
            }

            // Si encontramos el final de la línea
            if (i >= linea.length()) {
                break;
            }

            // Verificar si es un comentario de una línea
            if (i + 1 < linea.length() && linea.charAt(i) == '/' &&
linea.charAt(i + 1) == '/') {
                int inicioComentario = i;
                i += 2; // Saltar "//"
                while (i < linea.length() && linea.charAt(i) != '\n') {
                    i++;
                }
                String comentario = linea.substring(inicioComentario,
i);
                tokens.add(new Token(comentario, "", "JavaScript",
"Comentario de una línea", 0, 0));
            }
            // Verificar si es un comentario de múltiples líneas
            else if (i + 1 < linea.length() && linea.charAt(i) == '/' &&
linea.charAt(i + 1) == '*') {
                int inicioComentario = i;
```

```
                i += 2; // Saltar "/*"
                while (i + 1 < linea.length() && !(linea.charAt(i) == '*' && linea.charAt(i + 1) == '/')) {
                    i++;
                }
                if (i + 1 < linea.length()) {
                    String comentario = linea.substring(inicioComentario, i + 2);
                    tokens.add(new Token(comentario, "", "JavaScript", "Comentario de múltiples líneas", 0, 0));
                    i += 2; // Saltar "*/"
                }
            }
            // Verificar si el carácter actual es el inicio de un identificador o palabra reservada
            else if (Character.isLetter(linea.charAt(i)) || linea.charAt(i) == '_') {
                StringBuilder identificador = new StringBuilder();
                while (i < linea.length() && (Character.isLetterOrDigit(linea.charAt(i)) || linea.charAt(i) == '_')) {
                    identificador.append(linea.charAt(i));
                    i++;
                }
                String lexema = identificador.toString();
                if (esPalabraReservada(lexema)) {
                    tokens.add(new Token(lexema, "", "JavaScript", "Palabra Reservada", 0, 0));
                } else {
                    tokens.add(new Token(lexema, "", "JavaScript", "Identificador", 0, 0));
                }
            }
            // Verificar si es un número
            else if (Character.isDigit(linea.charAt(i))) {
```

```
                StringBuilder numero = new StringBuilder();
                while (i < linea.length() &&
(Character.isDigit(linea.charAt(i)) || linea.charAt(i) == '.')) {
                    numero.append(linea.charAt(i));
                    i++;
                }
                tokens.add(new Token(numero.toString(), "",
"JavaScript", "Número", 0, 0));
            }
            // Verificar cadenas
            else if (linea.charAt(i) == "" || linea.charAt(i) == '\" ||
linea.charAt(i) == '`') {
                char tipoCadena = linea.charAt(i);
                StringBuilder cadena = new StringBuilder();
                cadena.append(tipoCadena);
                i++;
                while (i < linea.length() && linea.charAt(i) !=
tipoCadena) {
                    cadena.append(linea.charAt(i));
                    i++;
                }
                if (i < linea.length()) {
                    cadena.append(tipoCadena); // Agregar el cierre de la
cadena
                    tokens.add(new Token(cadena.toString(), "",
"JavaScript", "Cadena", 0, 0));
                    i++; // Saltar el cierre de la cadena
                }
            }
            // Verificar si es un símbolo (incluyendo paréntesis,
corchetes, y llaves)
            else if (esSimbolo(Character.toString(linea.charAt(i)))) {
                StringBuilder simbolo = new StringBuilder();
                simbolo.append(linea.charAt(i));
```

```
                tokens.add(new Token(simbolo.toString(), "",
"JavaScript", "Símbolo", 0, 0));
                i++;
            }
            // Verificar operadores
            else {
                StringBuilder simbolo = new StringBuilder();
                while (i < linea.length() && !
Character.isWhitespace(linea.charAt(i))) {
                    simbolo.append(linea.charAt(i));
                    i++;
                }
                String lexema = simbolo.toString();
                if (esOperador(lexema)) {
                    tokens.add(new Token(lexema, "", "JavaScript",
"Operador", 0, 0));
                }
            }
        }
        for (Token token : tokens) {
            System.out.println("Agrego " + token.getLexema());
        }

        return tokens;
    }

    // Método para agregar tokens a la lista, asegurando que no se
dupliquen
    private void agregarToken(List<Token> tokens, Set<String>
tokensDetectados, String lexema, String tipo) {
        if (!tokensDetectados.contains(lexema)) {
            tokens.add(new Token(lexema, "", "JavaScript", tipo, 0,
0));
            tokensDetectados.add(lexema); // Añadir a los detectados
        }
```

```
}

// Verifica si una palabra es una palabra reservada de JavaScript
private boolean esPalabraReservada(String palabra) {
    for (String reservada : palabrasReservadasJS) {
        if (reservada.equals(palabra)) {
            return true;
        }
    }
    return false;
}

// Verifica si el lexema es un operador
private boolean esOperador(String lexema) {
    for (String operador : operadoresAritmeticos) {
        if (operador.equals(lexema)) return true;
    }
    for (String operador : operadoresRelacionales) {
        if (operador.equals(lexema)) return true;
    }
    for (String operador : operadoresLogicos) {
        if (operador.equals(lexema)) return true;
    }
    for (String operador : operadoresIncrementales) {
        if (operador.equals(lexema)) return true;
    }
    return false;
}

// Verifica si el lexema es un símbolo
private boolean esSimbolo(String lexema) {
    for (String simbolo : otrosSimbolos) {
        if (simbolo.equals(lexema)) return true;
    }
    return false;
```

```
    }

    // Método para analizar y retornar los tokens encontrados
    private String analizarTokensJS(String contenido) {
        List<Token> tokens = obtenerTokensValidosJS(contenido);
        StringBuilder sb = new StringBuilder();
        for (Token token : tokens) {
            sb.append(token.getLexema()).append("\n");
        }
        return sb.toString();
    }
}
```

Clase donde se analiza el codigo HTML y verificar si sera valido.

```
package AnalizadorHTML;

import Reportes.Token;
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class IdentificadorHTML {

    private String[] palabraReservada = {
        "class", "=", "href", "onClick", "id", "style", "type",
"placeholder", "required", "name"
    };
    private TraductorEtiquetas traductor = new
TraductorEtiquetas();
```

```
    private List<TraductorEtiquetas> etiNormales = new
ArrayList<>();
    private List<Token> tokenEncontrado = new ArrayList<>();

    public String analizarHTML(String cadena) {
        StringBuilder resultado = new StringBuilder();

        // Detectar y procesar el token de estado >>[html]
        if (cadena.contains(">>[html]")) {
            agregarToken(">>[html]", ">>\\[html\\]", "html", "Token
de Estado", 0, 0);
            resultado.append("Token de estado detectado: >>[html]\
n");
        }

        // Verificar si es un HTML válido
        if (esHTMLValido(cadena)) {
            resultado.append("Lenguaje HTML detectado\n");
            // Aquí llamamos al método que analiza las etiquetas
            List<Token> tokens = analizarEtiquetas(cadena);
            for (Token token : tokens) {
                resultado.append("Token encontrado:
").append(token.getLexema()).append("\n");
            }

            resultado.append(validarAtributos(cadena));
            resultado.append(validarCadenas(cadena));

            String erroresTexto =
validarTextoFueraDeEtiquetas(cadena);
            if (!erroresTexto.isEmpty()) {
                resultado.append(erroresTexto);
            }
        } else {
```

```
            resultado.append("El código no corresponde a HTML válido\n");
        }

        // Generar el reporte de tokens encontrados en la consola
        generarReporteTokens();

        return "Análisis completado."; // O cualquier mensaje que desees
    }

    private List<Token> analizarEtiquetas(String entrada) {
        List<Token> tokens = new ArrayList<>();
        List<String> etiquetas = traductor.etiquetasNormales(); // Obtiene las etiquetas válidas desde el traductor

        int i = 0; // Índice para recorrer la cadena de entrada
        while (i < entrada.length()) {
            // Verificar el token de estado antes de procesar etiquetas
            if (entrada.startsWith(">>[html]", i)) {
                tokens.add(new Token(">>[html]", ">>\\[html\\]", "html", "Token de Estado", 0, 0));
                i += 8; // Avanzamos más allá del token
                continue;
            }

            if (entrada.charAt(i) == '<') {
                int finEtiqueta = entrada.indexOf('>', i);
                if (finEtiqueta == -1) break; // No hay cierre de etiqueta

                String etiquetaCompleta = entrada.substring(i + 1, finEtiqueta).trim();
                String nombreEtiqueta;

                // Separar nombre de la etiqueta de sus atributos
```

```
            if (etiquetaCompleta.contains(" ")) {
                nombreEtiqueta = etiquetaCompleta.substring(0,
etiquetaCompleta.indexOf(" ")).trim();
            } else {
                nombreEtiqueta = etiquetaCompleta; // No tiene
atributos
            }

            // Verificar si la etiqueta es válida usando el arreglo
existente
            if (etiquetas.contains(nombreEtiqueta)) {
                // Agregar el token para la etiqueta
                tokens.add(new Token(nombreEtiqueta,
"expresión_regular_placeholder", "html", "Etiqueta Normal", 0,
0));

                // Manejo del texto interno si existe
                int inicioTexto = finEtiqueta + 1; // Mover después de
'>'
                int finTexto = entrada.indexOf('<', inicioTexto);
                if (finTexto > inicioTexto) {
                    String textoInterno = entrada.substring(inicioTexto,
finTexto).trim();
                    if (!textoInterno.isEmpty()) {
                        tokens.add(new Token(textoInterno,
"expresión_regular_placeholder", "html", "Texto Interno", 0, 0));
                    }
                }
            }
            i = finEtiqueta; // Mover el índice al final de la etiqueta
        } else {
            i++; // Avanzar al siguiente carácter
        }
    }
```

```
        return tokens;
    }



    private void generarReporteTokens() {
        if (!tokenEncontrado.isEmpty()) {
            System.out.println("\nReporte de Tokens Encontrados:");
            for (Token token : tokenEncontrado) {
                System.out.println("Token: " + token.getLexema()
                        + ", Tipo: " + token.getTipo()
                        + ", Lenguaje: " + token.getLenguaje()
                        + ", Fila: " + token.getFila()
                        + ", Columna: " + token.getColumna());
            }
        } else {
            System.out.println("No se encontraron tokens válidos.");
        }
    }

    public boolean esHTMLValido(String cadena) {
        Stack<String> stack = new Stack<>();
        int i = 0;

        while (i < cadena.length()) {
            if (cadena.charAt(i) == '<') {
                int finEtiqueta = cadena.indexOf('>', i);
                if (finEtiqueta == -1) {
                    return false; // Etiqueta no cerrada
                }
                String etiquetaCompleta = cadena.substring(i + 1,
finEtiqueta).trim();
                String nombreEtiqueta;

                if (etiquetaCompleta.contains(" ")) {
```

```
                nombreEtiqueta = etiquetaCompleta.substring(0,
etiquetaCompleta.indexOf(" ")).trim();
            } else {
                nombreEtiqueta = etiquetaCompleta; // No tiene
atributos
            }

            if (!etiquetaCompleta.startsWith("/")) { // Es una
etiqueta de apertura
                if (traductor.etiquetasNormales().contains("<" +
nombreEtiqueta + ">")) {



                    traductor.posiEtiquetas("<" + nombreEtiqueta +
">");

                    int tmp = traductor.posiEtiquetas("<" +
nombreEtiqueta + ">");



stack.push(traductor.etiquetaTraducida().get(tmp)); // Agregar
etiqueta al stack
                    System.out.println("Jalando");

                } else {
                    return false; // Etiqueta no válida
                }
            } else { // Es una etiqueta de cierre
                System.out.println("NO ES ETIQUETA
VALIDAAAAAA");
                String etiquetaDeApertura =
etiquetaCompleta.substring(1).trim(); // Extrae el nombre de la
etiqueta
```

```
                if (stack.isEmpty() || !
stack.peek().equals(etiquetaDeApertura)) {
                    return false; // No hay una etiqueta de apertura
correspondiente
                }
                stack.pop(); // Cerrar la etiqueta
            }
            i = finEtiqueta; // Mover el índice al final de la etiqueta
        }
        i++;
    }

    return stack.isEmpty(); // Asegúrate de que todas las etiquetas
están cerradas
  }

  private void validarPalabrasReservadas(String etiqueta,
StringBuilder resultado) {
    for (String palabra : palabraReservada) {
      if (etiqueta.contains(palabra)) {
        resultado.append("Palabra reservada encontrada:
'").append(palabra).append("'\n");
      }
    }
  }

  private String validarTextoFueraDeEtiquetas(String cadena) {
    StringBuilder errores = new StringBuilder();
    boolean dentroDeEtiqueta = false;
    StringBuilder textoActual = new StringBuilder();

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

      if (c == '<') {
```

```
                dentroDeEtiqueta = true;
                if (textoActual.length() > 0) {
                    errores.append("Texto fuera de etiqueta detectado:
").append(textoActual).append("\n");
                    textoActual.setLength(0);
                }
            }

            if (dentroDeEtiqueta && c == '>') {
                dentroDeEtiqueta = false;
            }

            if (!dentroDeEtiqueta) {
                textoActual.append(c);
            }
        }

        if (textoActual.length() > 0) {
            errores.append("Texto fuera de etiqueta detectado al final:
").append(textoActual).append("\n");
        }

        return errores.toString();
    }

    private String validarAtributos(String entrada) {
        StringBuilder resultado = new StringBuilder();
        int i = 0;

        while (i < entrada.length()) {
            if (entrada.charAt(i) == '<' && entrada.charAt(i + 1) != '/')
{
                int inicioEtiqueta = i;
                int finEtiqueta = entrada.indexOf('>', inicioEtiqueta);
                if (finEtiqueta == -1) {
```

```
                resultado.append("Etiqueta no válida o mal cerrada\
n");
                break;
            }

            String etiqueta = entrada.substring(inicioEtiqueta,
finEtiqueta + 1);
            resultado.append("Etiqueta de apertura encontrada:
").append(etiqueta).append("\n");

            // Validar palabras reservadas en la etiqueta
            validarPalabrasReservadas(etiqueta, resultado);

            // Validar cadenas entre comillas
            String cadenaResultado = validarCadenas(etiqueta);
            if (!cadenaResultado.isEmpty()) {
                resultado.append(cadenaResultado);
            }

            i = finEtiqueta;
        }
        i++;
    }
    return resultado.toString();
}

private String validarCadenas(String entrada) {
    StringBuilder resultado = new StringBuilder();
    int i = 0;

    while (i < entrada.length()) {
        char actual = entrada.charAt(i);

        if (actual == "" || actual == '\') {
            char delimitador = actual;
```

```
            i++;
            StringBuilder cadenaActual = new StringBuilder();

            while (i < entrada.length() && entrada.charAt(i) !=
delimitador) {
                cadenaActual.append(entrada.charAt(i));
                i++;
            }

            if (i < entrada.length()) {
                resultado.append("Cadena encontrada:
'").append(cadenaActual).append("'\n");
            }
        }
        i++;
    }
    return resultado.toString();
}

private void agregarToken(String lexema, String regex, String
lenguaje, String tipo, int fila, int columna) {
    Token nuevoToken = new Token(lexema, regex, lenguaje,
tipo, fila, columna);
    tokenEncontrado.add(nuevoToken);
}


/*
public List<Token> obtenerTokensValidos(String linea) {
    List<Token> tokens = new ArrayList<>();

    // Lógica para reconocer las etiquetas
    String regex = "<([a-zA-Z0-9]+)([^>]*)>(.*?)</\\1>"; //
Regex para etiquetas
    Pattern pattern = Pattern.compile(regex);
```

```
        Matcher matcher = pattern.matcher(linea);

        while (matcher.find()) {
            String nombreEtiqueta = matcher.group(1);
            String atributos = matcher.group(2);
            String contenido = matcher.group(3);

            // Agregar el token reconocido a la lista
            tokens.add(new Token(linea.toString(), linea.toString(),
"HTML", "",0, 0));
        }

        return tokens;
    }
    */


    public List<Token> obtenerTokensValidos(String linea) {
        List<Token> tokens = new ArrayList<>();
        int i = 0;

        while (i < linea.length()) {
            if (linea.charAt(i) == '<') {
                // Verificar si es un comentario
                if (i + 3 < linea.length() && linea.charAt(i + 1) == '!'
&& linea.charAt(i + 2) == '-' && linea.charAt(i + 3) == '-') {
                    int inicioComentario = i;
                    i += 4; // Saltamos '<!--'

                    // Buscar el cierre del comentario
                    while (i + 2 < linea.length() && !(linea.charAt(i) ==
'-' && linea.charAt(i + 1) == '-' && linea.charAt(i + 2) == '>')) {
                        i++;
                    }
```

```
                if (i + 2 < linea.length()) { // Si se encontró el cierre
                    String comentario =
linea.substring(inicioComentario, i + 3);
                    tokens.add(new Token(comentario, "", "HTML",
"Comentario", 0, 0));
                    i += 3; // Saltamos '-->'
                }
            } else {
                // Es el inicio de una etiqueta
                int inicioEtiqueta = i;
                i++; // Saltamos el '<'

                // Extraer el nombre de la etiqueta (hasta un espacio o
'>')
                StringBuilder nombreEtiqueta = new StringBuilder();
                while (i < linea.length() && linea.charAt(i) != ' ' &&
linea.charAt(i) != '>') {
                    nombreEtiqueta.append(linea.charAt(i));
                    i++;
                }

                // Saltar los atributos de la etiqueta si existen (hasta el
cierre '>')
                while (i < linea.length() && linea.charAt(i) != '>') {
                    i++;
                }

                // Si es el cierre de la etiqueta, movemos el índice
                if (i < linea.length() && linea.charAt(i) == '>') {
                    i++; // Saltamos el '>'
                }
```

```
                if (traductor.etiquetasNormales().contains("<" +
nombreEtiqueta + ">")) {


                    traductor.posiEtiquetas("<" + nombreEtiqueta +
">");

                    int tmp = traductor.posiEtiquetas("<" +
nombreEtiqueta + ">");

                     // Agregar el token para la etiqueta de apertura
                    tokens.add(new
Token(traductor.etiquetaTraducida().get(tmp), "", "HTML",
"Etiqueta", 0, 0));

                    System.out.println("Jalando");

                }

                // Verificar si hay contenido dentro de la etiqueta
                StringBuilder contenidoEtiqueta = new
StringBuilder();
                while (i < linea.length() && linea.charAt(i) != '<') {
                    contenidoEtiqueta.append(linea.charAt(i));
                    i++;
                }

                // Si se encontró contenido, agregarlo como token
                if (contenidoEtiqueta.length() > 0) {
                    tokens.add(new
Token(contenidoEtiqueta.toString().trim(), "", "HTML",
"Contenido", 0, 0));
                }
            }
        } else {
```

```
                i++; // Avanzar al siguiente carácter
            }


        }

        for (Token token : tokens) {
            System.out.println("Agregar "+token.getLexema());
        }
        return tokens;
    }


}
```

Clase donde se analiza el codigo y se identifica que analizador usar cuando se encuentre una etiqueta de estado las cuales serian: »[html], »[css], »[js]

```
package AnalizadorGeneral;

import AnalizadorCSS.IdentificadorCSS;
import AnalizadorHTML.IdentificadorHTML;
import AnalizadorJS.IdentificadorJS;
import Interfaz.VentanaTraduccion;
import Reportes.ResultadoAnalisis;
import Reportes.Token;

import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author alejandro
 */
public class AnalizarGeneral {
```

```
    private final IdentificadorHTML identificadorHTML;
    private final IdentificadorCSS identificadorCSS;
    private final IdentificadorJS identificadorJS;


    public AnalizarGeneral() {
        this.identificadorHTML = new IdentificadorHTML();
        this.identificadorCSS = new IdentificadorCSS();
        this.identificadorJS = new IdentificadorJS();
    }

    public ResultadoAnalisis analizarTexto(String texto) {
        List<Token> tokensHTML = new ArrayList<>();
        List<Token> tokensCSS = new ArrayList<>();
        List<Token> tokensJS = new ArrayList<>();

        String lenguajeActual = "";
        StringBuilder cadena = new StringBuilder();
        int fila = 1;  // Contador de líneas

        for (int i = 0; i < texto.length(); i++) {
            char actual = texto.charAt(i);

            // Detectar el inicio de un nuevo bloque de lenguaje
            if (actual == '>' && (i + 1) < texto.length() &&
texto.charAt(i + 1) == '>') {
                // Avanzar dos caracteres
                i += 2;
                lenguajeActual = obtenerLenguaje(texto, i);
                i += lenguajeActual.length();
                continue;
            }

            // Comprobar si el lenguaje actual es válido
```

```
        if (!lenguajeActual.isEmpty() && !
esLenguajeValido(lenguajeActual)) {
            System.out.println("Error: Lenguaje desconocido o
inválido: " + lenguajeActual);
            return new ResultadoAnalisis(new ArrayList<>(), new
ArrayList<>(), new ArrayList<>(), "Error: Lenguaje desconocido
o inválido.");
        }

        // Procesar líneas de texto según el lenguaje actual
        if (!lenguajeActual.isEmpty()) {
            if (actual == '\n') {
                // Procesar la línea completa si hay texto
                if (cadena.length() > 0) {
                    List<Token> tokens =
procesarLineaSegunLenguaje(lenguajeActual, cadena.toString(),
fila);
                    agregarTokens(tokensHTML, tokensCSS,
tokensJS, tokens, lenguajeActual);
                }
                // Reiniciar cadena y avanzar la línea
                cadena.setLength(0);
                fila++;
            } else {
                // Acumular caracteres en la cadena
                cadena.append(actual);
            }
        }
    }

    // Procesar la última línea si hay texto
    if (cadena.length() > 0 && !lenguajeActual.isEmpty()) {
        List<Token> tokens =
procesarLineaSegunLenguaje(lenguajeActual, cadena.toString(),
fila);
```

```
            agregarTokens(tokensHTML, tokensCSS, tokensJS,
tokens, lenguajeActual);
        }

        // Agregar etiquetas de estado al reporte
        agregarEtiquetasEstado(tokensHTML, tokensCSS,
tokensJS);

        return new ResultadoAnalisis(tokensHTML, tokensCSS,
tokensJS, "Resultado");
    }

    // Método para agregar etiquetas de estado
    private void agregarEtiquetasEstado(List<Token>
tokensHTML, List<Token> tokensCSS, List<Token> tokensJS) {
        if (!tokensHTML.isEmpty()) {
            tokensHTML.add(new Token("[html]", "html",
"HTML","Etiqueta de estado", -1, -1));
        }
        if (!tokensCSS.isEmpty()) {
            tokensCSS.add(new Token("[css]", "css", "CSS", "Etiqueta
de estado", -1, -1));
        }
        if (!tokensJS.isEmpty()) {
            tokensJS.add(new Token("[js]", "js", "JS", "Etiqueta de
estado", -1, -1));
        }
    }



    // Método para verificar si el lenguaje es válido
    private boolean esLenguajeValido(String lenguaje) {
        return "[html]".equals(lenguaje) || "[css]".equals(lenguaje) ||
"[js]".equals(lenguaje);
    }
```

```
    private String obtenerLenguaje(String texto, int index) {
        StringBuilder lenguajeBuilder = new StringBuilder();
        while (index < texto.length() && texto.charAt(index) != '\n')
{
            lenguajeBuilder.append(texto.charAt(index));
            index++;
        }
        String lenguaje = lenguajeBuilder.toString().trim();
        System.out.println("Lenguaje detectado: " + lenguaje); //
Debug
        return lenguaje;
    }

    private void agregarTokens(List<Token> tokensHTML,
List<Token> tokensCSS, List<Token> tokensJS, List<Token>
tokens, String lenguaje) {
        switch (lenguaje) {
            case "[html]":
                tokensHTML.addAll(tokens);
                break;
            case "[js]":
                tokensJS.addAll(tokens);
                break;
            case "[css]":
                tokensCSS.addAll(tokens);
                break;
            default:
                System.out.println("Lenguaje desconocido: " +
lenguaje);
                break;
        }
    }
```

```
    private List<Token> procesarLineaSegunLenguaje(String
lenguaje, String linea, int fila) {
        List<Token> tokens = new ArrayList<>();
        switch (lenguaje) {
            case "[html]":

                tokens =
identificadorHTML.obtenerTokensValidos(linea);
                break;
            case "[js]":
                System.out.println("Procesando JS: " + linea); // Debug
                tokens = identificadorJS.obtenerTokensValidosJS(linea);
                break;
            case "[css]":
                System.out.println("Procesando CSS: " + linea); //
Debug
                tokens =
identificadorCSS.obtenerTokensValidosCSS(linea);
                break;
            default:
                System.out.println("Lenguaje desconocido: " +
lenguaje);
                break;
        }
        return tokens;
    }

    /**
    * Optimiza el código eliminando comentarios y líneas que
contienen tokens.
    * @param texto El texto a optimizar.
    * @return El texto optimizado.
    */
    public String optimizarCodigo(String texto) {
      StringBuilder codigoOptimizado = new StringBuilder();
```

```
        boolean dentroDeComentarioSimple = false;
        boolean dentroDeComentarioMultiple = false;
        boolean dentroDeLineaConContenido = false;

        for (int i = 0; i < texto.length(); i++) {
            char actual = texto.charAt(i);

            // Detección de comentarios de una línea "//"
            if (!dentroDeComentarioMultiple && actual == '/' && i + 1
< texto.length() && texto.charAt(i + 1) == '/') {
                dentroDeComentarioSimple = true;
            }

            // Detección de comentarios de múltiples líneas "/*"
            if (!dentroDeComentarioSimple && actual == '/' && i + 1
< texto.length() && texto.charAt(i + 1) == '*') {
                dentroDeComentarioMultiple = true;
            }

            // Finalización de comentarios múltiples "*/"
            if (dentroDeComentarioMultiple && actual == '*' && i + 1
< texto.length() && texto.charAt(i + 1) == '/') {
                dentroDeComentarioMultiple = false;
                i++;
                continue;
            }

            // Ignorar el contenido de un comentario simple hasta el
final de la línea
            if (dentroDeComentarioSimple && actual == '\n') {
                dentroDeComentarioSimple = false;
                continue;
            }

            // Si estamos dentro de un comentario, ignorar el contenido
```

```
        if (dentroDeComentarioSimple ||
dentroDeComentarioMultiple) {
            continue;
        }

        // Detección de líneas vacías
        if (actual == '\n') {
            // Si hay contenido en la línea, no agregarla al resultado
            if (dentroDeLineaConContenido) {
                dentroDeLineaConContenido = false; // Reiniciar para
la próxima línea
            }
            continue; // Ignorar salto de línea
        }

        // Si encuentra un carácter que no sea espacio en blanco, la
línea tiene contenido
        if (!Character.isWhitespace(actual)) {
            dentroDeLineaConContenido = true;
        }

        // Agregar carácter al código optimizado si no se está en
una línea que debe ser eliminada
        if (dentroDeLineaConContenido) {
            codigoOptimizado.append(actual);
        }
      }

      return codigoOptimizado.toString().trim();
    }


    public void mostrarResultados(String texto) {
        // Obtener los tokens procesados
        ResultadoAnalisis resultado = analizarTexto(texto);
```

```
        List<Token> tokensHTML = resultado.getTokensHTML();
        List<Token> tokensCSS = resultado.getTokensCss();
        List<Token> tokensJS = resultado.getTokensJs();

        // Optimizar el código
        String textoOptimizado = optimizarCodigo(texto);

        // Generar archivo HTML
        GeneradorHTML generadorHTML = new
GeneradorHTML();
        String rutaArchivo = "resultadoAnalisis.html"; // Puedes
cambiar la ruta si lo deseas
        generadorHTML.generarArchivoHTML(tokensHTML,
tokensCSS, tokensJS, rutaArchivo);

        // Mostrar la ventana de resultados si hay tokens
        if (!tokensHTML.isEmpty() || !tokensCSS.isEmpty() || !
tokensJS.isEmpty()) {
            VentanaTraduccion ventana = new
VentanaTraduccion(tokensHTML, "Resultados",
textoOptimizado);
            ventana.setVisible(true);
        }
    }




}
```