Universidad San Carlos de Guatemala “CUNOC”

Lenguajes Formales y de Programación

Ing. Daniel Gonzales

Manual Técnico

Mario Raúl Yancor Ocaña

201930761

Quetzaltenango, 9 de octubre del 2024

MANUAL TECNICO

Índice

1. Introducción

Este manual describe en detalle el funcionamiento de un proyecto de Análisis Léxico que procesa fragmentos de código en JavaScript, CSS, y HTML, separándolos y organizándolos en un archivo HTML. Además, el sistema genera reportes de tokens y optimización, así como un reporte de errores cuando sea necesario.

El sistema está implementado en Java, y utiliza una interfaz gráfica creada con Swing para recibir la entrada de texto, realizar el análisis y mostrar los resultados al usuario.

2. Arquitectura del Sistema

El sistema está basado en una arquitectura MVC (Modelo-Vista-Controlador).

- Vista (GUI): Implementada en la clase AnalizadorFrame, es la interfaz gráfica que permite al usuario interactuar con el sistema.
- Controlador (Manejador): La clase Manejador es el corazón del sistema, donde se realiza el análisis léxico y la generación de archivos y reportes.
- Modelo (Datos): La clase Reporte gestiona los datos que se generan durante el análisis, incluidos los reportes de tokens y de optimización.

Componentes Clave

- Entrada del Usuario: El usuario ingresa fragmentos de código en un área de texto (JTextArea) que es procesada por el controlador.
- Análisis Léxico: El sistema separa el código en CSS, HTML y JavaScript y lo organiza.
- Generación de Archivos: El sistema genera un archivo HTML que contiene las secciones de CSS, HTML y JavaScript, organizadas adecuadamente.

- Reportes: El sistema genera reportes de los tokens reconocidos, la optimización y posibles errores en el análisis.

3. Diseño de la Interfaz de Usuario (GUI)

La interfaz gráfica del sistema está diseñada usando el framework Swing de Java. Los principales componentes son:

- JTextArea (textoIntro): Área de texto donde el usuario introduce los fragmentos de código.
- JTextArea (textoResultado): Área de texto donde se muestran los resultados del análisis.
- JButton (analizarBoton): Botón que inicia el análisis léxico cuando se presiona.
- JButton (limpiarBoton): Botón que limpia las áreas de texto de entrada y resultados.
- JButton (btnReporte): Botón que genera el reporte de tokens.
- JButton (btnReporteOpti): Botón que genera el reporte de optimización.
- JButton (btnReporteError): Botón para la generación de reportes de errores.

Distribución de la Interfaz

La interfaz está distribuida en dos columnas:

- Columna Izquierda: Contiene el área de texto donde el usuario ingresa el código.
- Columna Derecha: Muestra los resultados del análisis en formato HTML organizado.

Flujo de Interacción

1. El usuario ingresa el código en el área de entrada y presiona el botón "Analizar".
2. El sistema procesa el código, separa las secciones y genera un archivo HTML con el resultado.

3. El usuario puede limpiar el área de texto o generar reportes adicionales (tokens, optimización, errores).

4. Clases Principales

4.1. Clase AnalizadorFrame

Esta clase representa la interfaz gráfica del programa. Es responsable de recibir las entradas del usuario y mostrar los resultados. Además, maneja los eventos relacionados con los botones de la interfaz.

Variables de instancia principales:

- textoIntro (JTextArea): Donde el usuario introduce el código a ser analizado.
- textoResultado (JTextArea): Donde se muestran los resultados del análisis.
- analizador (Manejador): Instancia de la clase Manejador, que realiza el análisis léxico.

Eventos clave:

- analizarBotonActionPerformed(): Este método se activa cuando el usuario presiona el botón "Analizar". Envía el texto introducido al Manejador para su procesamiento.
- limpiarBotonActionPerformed(): Limpia los campos de entrada y resultado.
- btnReporteActionPerformed(): Genera un reporte HTML de los tokens identificados.
- btnReporteOptiActionPerformed(): Genera un reporte de optimización basado en el análisis.
- btnReporteErrorActionPerformed(): (A implementar) Genera un reporte de errores si se encuentran.

4.2. Clase Manejador

La clase Manejador es la responsable de procesar el texto de entrada, analizar el contenido y generar los archivos HTML y reportes.

Variables de instancia:

- textoEntrada: Almacena el texto de entrada para ser procesado.
- cssContent: Almacena el contenido CSS extraído del texto.
- jsContent: Almacena el contenido JavaScript.
- htmlContent: Almacena el contenido HTML.
- reporte: Instancia de la clase Reporte, utilizada para manejar los datos del análisis.

Métodos clave:

- analizar(): Llama al método separarLenguajes() para dividir el texto en CSS, HTML y JS.
- separarLenguajes(): Realiza la separación de lenguajes dentro del texto proporcionado por el usuario.
- getHtmlOrdenado(): Genera el contenido HTML final con las secciones ordenadas de CSS, JS y HTML.
- generarArchivoHtml(): Genera un archivo HTML con el contenido ordenado en una carpeta específica.
- obtenerReporteTokens(): Devuelve un reporte de los tokens identificados durante el análisis.
- generarReporteOptimizacion(): Genera un archivo HTML que contiene un reporte de optimización.

4.3. Clase Reporte

La clase Reporte se encarga de gestionar los reportes que genera el sistema, como el reporte de tokens y de optimización.

Métodos clave:

- generarReporteTokens(): Identifica y genera un reporte de los tokens presentes en el texto.
- generarReporteErrores(): (A implementar) Genera un reporte de los errores encontrados durante el análisis.

5**. Detalle de los Métodos**

5.1. Métodos en AnalizadorFrame

- analizarBotonActionPerformed(): Procesa el texto de entrada utilizando la clase Manejador, llama a getHtmlOrdenado() para obtener el contenido HTML procesado y lo muestra en textoResultado.
- limpiarBotonActionPerformed(): Borra el contenido de las áreas de texto textoIntro y textoResultado.
- btnReporteActionPerformed(): Llama al método obtenerReporteTokens() de la clase Manejador para generar el reporte de tokens en un archivo HTML.
- btnReporteOptiActionPerformed(): Invoca generarReporteOptimizacion() para crear el reporte de optimización.
- btnReporteErrorActionPerformed(): Genera un reporte de errores si es necesario.

5.2. Métodos en Manejador

- analizar(): Llama a separarLenguajes() para analizar y separar el texto en CSS, JS y HTML.

- separarLenguajes(): Implementa la lógica para identificar y extraer las secciones de CSS, JS y HTML.
- getHtmlOrdenado(): Devuelve un string con el contenido HTML final, organizando CSS, JS y HTML.
- generarArchivoHtml(): Escribe el contenido HTML en un archivo en la ruta especificada.
- obtenerReporteTokens(): Genera un reporte de los tokens identificados en el análisis.
- generarReporteOptimizacion(): Genera un reporte sobre las optimizaciones realizadas durante el análisis.

5.3. Métodos en Reporte

- generarReporteTokens(): Identifica los tokens válidos en el texto de entrada y genera un reporte HTML.
- generarReporteErrores(): (A implementar) Identifica y reporta errores de sintaxis o análisis.

6. Generación de Archivos HTML

El sistema genera varios archivos HTML como resultado del análisis. Estos archivos incluyen:

- Archivo principal HTML: Contiene el contenido procesado, con las secciones de CSS, JavaScript y HTML organizadas correctamente.
- Reporte de Tokens: Un archivo HTML separado que contiene los tokens identificados en el código.
- Reporte de Optimización: Describe las optimizaciones aplicadas al código para mejorar su eficiencia o corregir problemas.

7. Gestión de Errores

La gestión de errores es una parte crucial del sistema, ya que garantiza que los posibles problemas en el análisis léxico puedan ser detectados y reportados correctamente.

7.1. Tipos de Errores

El sistema está diseñado para identificar diferentes tipos de errores durante el análisis léxico:

- Errores de Sintaxis: Estos errores ocurren cuando el código de entrada contiene fragmentos mal formados o símbolos inesperados.
- Errores de Tipificación de Datos: Se generan cuando un valor no coincide con el tipo esperado. Por ejemplo, si una regla CSS espera un valor numérico y encuentra una cadena de texto.
- Errores de Estructura: Ocurren cuando las estructuras HTML, CSS o JavaScript están incompletas o mal balanceadas. Ejemplos incluyen etiquetas HTML sin cerrar, corchetes faltantes en CSS, o paréntesis sin su par correspondiente en JavaScript.

7.2. Mecanismo de Detección de Errores

El proceso de detección de errores se realiza dentro de la clase Manejador mediante la lógica del método analizar(), que inspecciona el código de entrada para identificar patrones incorrectos. La detección de errores se gestiona de la siguiente manera:

1. Reconocimiento de Patrones: El sistema compara cada fragmento de código con las expresiones regulares establecidas para identificar los componentes válidos de HTML, CSS y JavaScript.
2. Reporte de Errores: Cuando se detecta un error, se almacena en una lista de errores, que posteriormente puede ser procesada por la clase Reporte para generar un archivo HTML que enumere y describa los problemas encontrados.

7.4. Reporte de Errores

Los errores se recopilan y se presentan en un archivo HTML generado por el método generarReporteErrores() en la clase Reporte. Este archivo contiene una lista detallada de los errores encontrados, junto con sus ubicaciones dentro del texto de entrada.

8. Optimizaciones Posibles

El proyecto puede beneficiarse de diversas optimizaciones, tanto a nivel de rendimiento como en la mejora de la experiencia del usuario y la robustez del sistema.

8.1. Mejora del Análisis Léxico

El actual análisis léxico puede optimizarse a través de:

- Mejoras en las Expresiones Regulares: Refactorizar las expresiones regulares utilizadas para la detección de tokens en HTML, CSS y JavaScript, de manera que sean más eficientes y flexibles.
- Análisis Predictivo: Implementar técnicas de análisis predictivo para anticipar errores y proporcionar sugerencias en tiempo real mientras el usuario escribe el código.

8.2. Manejo de Grandes Volúmenes de Código

Para mejorar el rendimiento al procesar grandes volúmenes de código, se pueden aplicar las siguientes optimizaciones:

- Procesamiento Paralelo: Implementar procesamiento paralelo o multihilo para dividir el análisis léxico entre varios hilos de ejecución, acelerando el tiempo de respuesta.
- Carga Diferida (Lazy Loading): En lugar de cargar y analizar todo el código en una sola pasada, se podría implementar una carga diferida que procese el código por secciones, mejorando el uso de memoria.

8.3. Interfaz de Usuario Mejorada

Algunas mejoras que se pueden realizar en la interfaz de usuario incluyen:

- Autocompletado y Validación en Tiempo Real: Añadir autocompletado y resaltado de sintaxis para que el usuario pueda corregir errores a medida que escribe, sin necesidad de esperar al análisis final.
- Historial de Análisis: Incorporar una funcionalidad para guardar y cargar análisis previos, permitiendo al usuario trabajar en varias sesiones y revisar análisis antiguos.

8.4. Reportes Avanzados

El sistema de generación de reportes podría ampliarse con nuevas funcionalidades:

- Reporte de Rendimiento: Un reporte que evalúe el impacto del código en el rendimiento web, identificando posibles cuellos de botella en el CSS, HTML o JavaScript.
- Sugerencias de Mejora: Incluir recomendaciones automáticas en los reportes de optimización, basadas en buenas prácticas de programación web.

9. Conclusiones

El sistema de análisis léxico desarrollado proporciona una solución efectiva para el procesamiento de fragmentos de código en CSS, JavaScript, y HTML, generando archivos organizados y reportes que facilitan la detección de errores y optimizaciones. Las características más destacadas del sistema incluyen:

- Análisis preciso y detallado: El sistema separa correctamente los lenguajes de entrada, generando archivos HTML organizados.
- Generación de reportes: El sistema crea reportes útiles tanto para los tokens identificados como para las optimizaciones realizadas.
- Interfaz intuitiva: La interfaz gráfica permite al usuario ingresar código fácilmente y visualizar resultados en tiempo real.

Sin embargo, existen áreas donde el sistema puede mejorar, particularmente en la gestión de grandes volúmenes de código y en la detección proactiva de errores. Con las optimizaciones sugeridas, el sistema puede evolucionar hacia una herramienta más robusta y versátil.