DIVISIÓN DE CIENCIAS DE LA INGENIERÍA
LENGUAJES FORMALES DE PROGRAMACION

Manual Técnico - "MULTIPLE ANALIZADOR LEXICO"

El IDE utilizado fue: NetBeans 20.

Con la versión de java: 17.0.6

CLASE TOKEN: es un componente fundamental para el analizador léxico. Cada instancia de la clase encapsula la información necesaria sobre un token identificado en el código fuente, incluyendo su lexema, traducción, tipo, posición dentro del código, y en este caso, también una expresión regular asociada para representar la estructura del token.

Atributos:

1. **lexema:** Representa la secuencia de caracteres que conforman el token en el código fuente original.
2. **traduccion:** Este campo almacena la traducción del token en otro lenguaje (útil si se está haciendo traducción entre lenguajes de programación o natural).
3. **lenguaje:** Indica el lenguaje de programación o de entrada donde se originó este token.
4. **tipo:** Define el tipo de token, que puede ser un identificador, palabra reservada, operador, número, etc.
5. **expresionRegular: E**s la expresión regular que define la estructura esperada del token. Esto es útil para identificar y validar el formato del token en el código.
6. **fila:** Representa la posición en la fila (línea) del código fuente donde fue encontrado el token.
7. **columna:** Representa la posición en la columna dentro de la línea donde comienza el token.

Métodos:

1. **getLexema()**:
    a. **Descripción**: Devuelve el lexema del token.
    b. **Retorno**: String (el lexema del token).

2. **getTraduccion()**:
    a. **Descripción**: Devuelve la traducción del token.
    b. **Retorno**: String (la traducción del token).
3. **getLenguaje()**:

   a. **Descripción**: Devuelve el lenguaje de origen del token.
   b. **Retorno**: String (el lenguaje del token).
4. **getTipo()**:
   a. **Descripción**: Devuelve el tipo de token (identificador, operador, palabra reservada, etc.).
   b. **Retorno**: String (el tipo del token).
5. **getExpresionRegular()**:
   a. **Descripción**: Devuelve la expresión regular asociada al token.
   b. **Retorno**: String (la expresión regular que representa el formato del token).
6. **getFila()**:
   a. **Descripción**: Devuelve la fila en la que se encuentra el token en el código fuente.
   b. **Retorno**: int (la fila donde está el token).
7. **getColumna()**:
   a. **Descripción**: Devuelve la columna donde comienza el token en el código fuente.
   b. **Retorno**: int (la columna donde inicia el token).
8. **toString()**:
   a. **Descripción**: Proporciona una representación en forma de cadena del token, mostrando todos sus atributos.
   b. **Retorno**: String (una cadena que contiene la información sobre el lexema, traducción, lenguaje, tipo, expresión regular, fila y columna).

Uso:

La clase Token es esencial para la representación y manipulación de los diferentes elementos léxicos dentro del código fuente del analizador léxico. Al utilizar los atributos como la expresión regular y las posiciones dentro del código, esta clase puede facilitar la validación y el análisis del código fuente, además de permitir una fácil traducción entre lenguajes o la categorización de elementos.

CLASE OPTMIZAR CODIGO: se encarga de realizar optimizaciones básicas sobre el fragmento de código fuente, como la eliminación de comentarios y de líneas vacías. Este tipo de optimización es útil para mejorar la legibilidad y reducir la cantidad de líneas innecesarias en el código, manteniendo su funcionalidad.

Atributos:

1. **codigoFuente**: Es una cadena de texto que contiene el código fuente original que será optimizado. Este es el código que se pasa a la clase en el constructor.

Métodos:

1. **optimizar():**
    - Descripción: Este método realiza el proceso de optimización del código fuente. Aplica dos optimizaciones clave:
        1. Eliminación de comentarios de línea: Las líneas que comienzan con // son consideradas comentarios y se eliminan.
        2. Eliminación de líneas vacías: Las líneas que están vacías o solo contienen espacios se eliminan.
    - Retorno: String (devuelve el código fuente optimizado).

Proceso de Optimización:

- Dividir en líneas: Se utiliza el método split("\n") para dividir el código en un arreglo de líneas.
- Eliminar comentarios y líneas vacías: Para cada línea:
    - Si la línea comienza con //, se ignora.
    - Si la línea está vacía (después de eliminar espacios en blanco), también se ignora.
- Agregar líneas válidas: Si la línea contiene código válido, se añade al código optimizado.

Uso:

es sencilla pero eficaz para realizar optimizaciones básicas en el código fuente. Su propósito principal es eliminar comentarios y líneas vacías, dejando solo las líneas que contienen código ejecutable.

CLASE ANALIZADOR_JS: es una subclase de Analizador_Lexico, que está diseñada para realizar un análisis léxico de código JavaScript. Utiliza una tabla de palabras reservadas específicas de JavaScript y varios métodos para identificar tokens según el tipo (como operadores lógicos, aritméticos, cadenas, comentarios, etc.).

Atributos:

- **palabrasReservadasJS:** Es un HashMap que almacena palabras reservadas de JavaScript. Este mapa se utiliza para reconocer palabras clave específicas del lenguaje durante el análisis léxico.

Métodos:

1. **Constructor:**
   - El constructor Analizador_JS() inicializa la tabla de palabras reservadas para JavaScript. Algunas palabras reservadas incluidas son function, let, const, if, else, return, entre otras.
2. **Método Analizar(String linea, int fila):**
   - Este método sobreescribe el método Analizar de la clase base Analizador_Lexico y analiza línea por línea el código JavaScript.
   - Procesa cadenas, delimitadores, operadores y palabras, añadiendo los tokens correspondientes a una lista de tokens.
   - Los delimitadores (como ;, ,, {, }, []) y el punto (.) se identifican y clasifican como tokens específicos.
   - Maneja correctamente las cadenas delimitadas por comillas simples, dobles o backticks, para asegurarse de que los tokens se procesen adecuadamente sin dividir la cadena en partes incorrectas.
3. **Método procesarPalabra(String palabra, int fila, int columna):**
   - Este método procesa cada palabra identificada en el análisis léxico.
   - Dependiendo de la palabra, se determina su tipo (palabra reservada, identificador, número entero, decimal, cadena, booleano, operador, etc.) y se crea el token correspondiente, que se agrega a la lista de tokens.
   - Si no se reconoce la palabra, se añade un error al reporte con una sugerencia del analizador adecuado (por ejemplo, CSS o HTML).
4. **Métodos auxiliares para la validación de diferentes tipos de tokens:**
   - EsComentario: Determina si una palabra es un comentario (comienza con //).
   - EsOperadorLogico: Verifica si la palabra es un operador lógico (&&, ||, !).
   - EsIdentificador: Valida si la palabra es un identificador que cumple con las reglas de JavaScript (comienza con una letra o guion bajo, y los caracteres restantes son letras, dígitos o guiones bajos).
   - EsEntero: Comprueba si la palabra es un número entero.
   - EsDecimal: Verifica si la palabra es un número decimal.
   - EsCadena: Valida si la palabra es una cadena delimitada por comillas simples, dobles o backticks.
   - EsOperadorRelacional: Determina si la palabra es un operador relacional (==, !=, <, <=, >, >=).

- EsOperadorAritmetico: Valida si la palabra es un operador aritmético (+, -, *, /).
- EsIcrementales: Comprueba si la palabra es un operador de incremento o decremento (++, --).
- EsAsignacion: Verifica si la palabra es un operador de asignación (=).

5. **Método sugerirAnalizador(String palabra):**
   - Este método se utiliza cuando una palabra no puede ser identificada como un token válido. Intenta sugerir a qué analizador podría pertenecer la palabra (JavaScript, CSS o HTML).

Uso:

proporciona una funcionalidad completa para identificar y clasificar tokens de código JavaScript, tales como palabras reservadas, operadores, cadenas, números y más. Además, maneja comentarios y genera errores para palabras no reconocidas, proporcionando sugerencias sobre el analizador al que podrían pertenecer esos tokens no válidos.

CLASE ANALIZOR_HTML: extiende de la clase Analizador_Lexico y se encarga de analizar texto que simula ser código HTML, incluyendo etiquetas personalizadas que luego son traducidas a etiquetas HTML estándar. Además, la clase identifica palabras reservadas y genera un contenido HTML traducido junto con un reporte de tokens y errores durante el análisis.

Atributos Principales:

- **TraduccionesHTML:** Un HashMap que almacena las traducciones de etiquetas personalizadas a etiquetas estándar de HTML.
- **PalabrasReservadas:** Un HashMap que contiene palabras reservadas relacionadas con atributos HTML.
- **tokens:** Una lista de objetos Token donde se almacenan los tokens válidos identificados durante el análisis.
- **errors:** Una lista de objetos Error donde se almacenan los errores encontrados durante el análisis.
- **htmlContent:** Un StringBuilder que almacena el contenido HTML traducido.
- **tokenCount:** Contador que lleva el número de tokens válidos encontrados.
- **tokenCountE: C**ontador que lleva el número de tokens de error identificados.

Métodos Principales:

1. Analizador_HTML():
Constructor que inicializa los mapas de traducciones HTML y palabras reservadas, así como las listas de tokens y errores, y los contadores de tokens.

2. inicializarTraduccionesHTML():
Inicializa el HashMap con las etiquetas personalizadas y sus correspondientes traducciones a HTML estándar.

3. inicializarPalabrasReservadas():
Define las palabras reservadas utilizadas en el análisis HTML, que se relacionan con los atributos de las etiquetas (como class, id, href, etc.).

4. Analizar(String linea, int fila):
Método principal de análisis léxico que procesa una línea de texto identificando tokens como etiquetas HTML o palabras reservadas. Utiliza un ciclo para recorrer la línea de texto, identificando etiquetas (entre < y >) y procesando cada palabra o símbolo identificado.

5. procesarPalabra(String palabra, int fila, int columna):
Procesa una palabra específica y decide si es una etiqueta HTML personalizada, un atributo, texto o un error. Si es una etiqueta personalizada, la traduce a HTML y la agrega al contenido HTML generado. Si es un error, se registra en la lista de errores.

6. AgregarToken(Token token):
Agrega un token válido a la lista de tokens y muestra un mensaje de depuración con el tipo y lexema del token.

7. AgregarTokenError(Error errorr):
Registra un token de error en la lista de errores e incrementa el contador de errores.

8. getHtmlContent():
Devuelve el contenido HTML generado hasta el momento como una cadena.

9. getTokens():
Devuelve la lista de tokens identificados.

10. getError():
Devuelve la lista de errores identificados durante el análisis.

11. determinarTipo(String etiqueta):
Determina el tipo de una etiqueta HTML personalizada (si es de apertura, cierre o si corresponde a un texto de una sola línea).

12. sugerirAnalizador(String palabra):
Intenta determinar si una palabra no reconocida pertenece a CSS o JavaScript, basándose en ciertas pistas (como el uso de : o .).

USO:

toma líneas de texto que representan código HTML y las procesa para identificar etiquetas personalizadas o reservadas, generando un contenido HTML traducido y almacenando los tokens identificados. También maneja errores y los reporta si una palabra no pertenece al lenguaje HTML analizado.

CLASE ANALIZADOR_CSS: extiende Analizador_Lexico y está diseñada para analizar código CSS línea por línea, identificando tokens CSS y almacenándolos en una lista.

Atributos principales:

- **ReglasCSS:** Un HashMap que almacena las propiedades CSS válidas que el analizador puede reconocer.
- **cssContent:** Un StringBuilder que almacena el contenido CSS que se genera o traduce a partir del análisis.
- **tokenCount y tokenCountE:** Contadores que probablemente se utilizan para llevar un registro del número de tokens procesados y errores encontrados.
- **tokens:** Una lista (List<Token>) que almacena todos los tokens válidos que el analizador ha identificado en el código CSS.
- **errors:** Una lista (List<Error>) que almacena los errores encontrados durante el proceso de análisis.

Métodos principales:

- **inicializarReglasCSS():** Inicializa el HashMap con una lista de propiedades CSS válidas, como color, font-size, width, entre muchas otras.
- **Analizar(String linea, int fila):** Realiza el análisis de una línea del código CSS, identificando palabras clave, propiedades, valores, y caracteres especiales como : y ;. Procesa cada carácter de la línea y acumula las palabras, para luego enviarlas al método procesarPalabra.
- **procesarPalabra(String palabra, int fila, int columna):** Identifica el tipo de la palabra que ha sido procesada. Verifica si la palabra es un selector, una propiedad CSS, un valor (medidas, colores, etc.), un comentario, o algún otro token relevante del lenguaje CSS. Si no es reconocida, el sistema sugiere un posible error de análisis.
- **EsComentario(String palabra):** Identifica si la palabra corresponde a un comentario CSS.
- **EsSelectorUniversal(String palabra):** Verifica si el selector es universal (*).
- **EsSelectorEtiqueta(String palabra):** Verifica si una palabra es una etiqueta HTML válida, como div, p, header, etc.
- **EsSelectorClase(String palabra):** Verifica si la palabra es un selector de clase válido (comienza con . y sigue una estructura adecuada).
- **EsSelectorID(String palabra)**: Verifica si la palabra es un selector de ID válido (comienza con # y sigue ciertas reglas).
- **EsCombinador(String palabra):** Verifica si la palabra es un combinador CSS (>, +, ~).

- **EsPseudoclase(String palabra):** Identifica si la palabra es una pseudoclase CSS como :hover, :before, :after, entre otras.
- **EsValorMedida(String palabra):** Identifica si la palabra es un valor de medida en CSS, como 100px, 50%, etc.

USO:

Es una herramienta de análisis léxico enfocada en identificar y clasificar correctamente las partes de un archivo CSS. Esto incluye propiedades CSS, selectores, valores, combinadores, pseudoclases y otros elementos que forman parte de las reglas de estilo CSS. Si se encuentra algún error o palabra que no es reconocida como válida en CSS, el analizador lo reporta como un error y sugiere un posible origen o corrección.

CLASE ANALIZAERO_LEXICO: es una clase abstracta que sirve como base para crear analizadores léxicos en diferentes contextos o lenguajes. Un analizador léxico es responsable de descomponer una entrada de texto en componentes más pequeños llamados tokens, que son las unidades fundamentales de sintaxis (como palabras clave, operadores, literales, etc.). Además, también detecta y reporta errores léxicos cuando encuentra entradas no válidas.

Atributos

1. List<Token> tokens

- Almacena todos los tokens que se identifican como válidos durante el análisis léxico.
- La lista es inicializada en el constructor, permitiendo que cada vez que se use un analizador léxico derivado, la lista de tokens esté vacía al inicio.

2. List<Error> errors

- Almacena los errores léxicos que se encuentran durante el análisis.
- La lista es inicializada de manera similar a la lista de tokens, garantizando que comience vacía.

Métodos

1. Analizador_Lexico() (Constructor)

- **Propósito**: Inicializa las listas de tokens y errors cuando se crea una instancia de una subclase.
- **Descripción**: El constructor crea listas vacías para tokens y errors, asegurando que se pueda agregar información conforme se realice el análisis léxico en las subclases.

2. abstract void Analizar(String linea, int fila)

- **Propósito**: Este método abstracto es donde el análisis léxico se lleva a cabo.
- **Descripción**: Las subclases que extienden de Analizador_Lexico deben implementar este método, que se encargará de analizar una línea de código y descomponerla en tokens o encontrar errores.
- **Parámetros**:
    - String linea: Es la línea de código que se va a analizar.
    - int fila: Es el número de fila en la que se encuentra la línea de código (útil para reportar errores y ubicaciones exactas de los tokens).

3. List<Token> getTokens()

- **Propósito**: Retorna la lista de tokens válidos que se han identificado durante el análisis léxico.

- **Descripción**: Este método devuelve la lista tokens, permitiendo que otras partes del programa accedan a los tokens generados. Es un método esencial para obtener el resultado del análisis.

4. List<Error> getErrors()

- **Propósito**: Retorna la lista de errores que se encontraron durante el análisis léxico.
- **Descripción**: Similar al método anterior, pero en este caso, se utiliza para devolver los errores léxicos que se encontraron durante el análisis. Permite al sistema saber qué errores fueron detectados y dónde ocurrieron en el código.

USO:

actúa como una plantilla general para construir los analizadores léxicos que puedan procesar diferentes tipos de lenguaje. Se espera que cada lenguaje que necesite ser analizado extienda esta clase y defina el comportamiento específico de cómo analizar el código fuente, clasificando las entradas en **tokens** o reportando **errores** cuando no se pueda identificar una entrada válida.