Scanner - Documentacion

Este proyecto consiste en construir un compilador para el lenguaje Decaf. La primera fase del compilador es el análisis léxico, que se realiza mediante un **scanner** generado con JFlex. Este scanner identifica y clasifica los tokens del lenguaje Decaf, maneja errores léxicos, y filtra comentarios y espacios en blanco.

2. Herramientas y Tecnologías

- **JFlex**: Generador de scanners basado en especificaciones de expresiones regulares.
- **CUP**: Generador de parsers que se integra con el scanner para realizar el análisis sintáctico.
- **Java**: Lenguaje de programación utilizado para implementar el scanner y el compilador.

3. Estructura del Proyecto

El proyecto se organiza en varios paquetes y directorios según la funcionalidad:

- **/class**: Contiene clases para representar estructuras de datos del lenguaje.
- **/compiler**: Contiene la clase `Compiler.java`, que sirve como interfaz de línea de comandos (CLI) para ejecutar el compilador.
- **/scanner**: Contiene el archivo `Scanner.java`, generado por JFlex, y el archivo de especificación `Scanner.flex`.
- **/parser**: Contiene el archivo `Parser.java`, generado por CUP, que realiza el análisis sintáctico.
- **/ast**: Contiene la clase `Ast.java` para representar el árbol de sintaxis abstracta.
- **/semantic**: Contiene la clase `Semantic.java` para realizar el análisis semántico.
- **/irt**: Contiene `Irt.java`, que se encarga de la representación intermedia.
- **/opt**: Contiene clases para optimización, como `Algebraic.java` y `ConstantF.java`.
- **/codegen**: Contiene `Codegen.java`, que genera el código final a partir de la representación intermedia.

4. Implementación del Scanner

Archivo de Especificación: `scanner/Scanner.flex`

Archivo Generado: `scanner/Scanner.java`

4.1. Definición de Tokens

El archivo `Scanner.flex` utiliza expresiones regulares para definir los tokens del lenguaje Decaf. Los tokens incluyen:

- **Palabras clave**: `if`, `else`, `while`, `for`, `int`, `float`, `boolean`, etc.
- **Operadores**: `=`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `+`, `-`, `*`, `/`.
- **Delimitadores**: `(`, `)`, `{`, `}`, `[`, `]`, `;`, `,`.
- **Literales**: Números enteros y flotantes, identificadores, literales de cadena.

4.2. Manejo de Errores

El scanner debe identificar y manejar errores léxicos de manera robusta:

- **Caracteres Ilegales**: Detecta caracteres no reconocidos y muestra un mensaje de error sin detener el análisis.
- **Comillas Faltantes**: Detecta literales de cadena con comillas no cerradas y emite un mensaje de error.
- **Errores en Literales de Flotantes**: Detecta literales de flotante mal formateados (por ejemplo, `123.`) y muestra un mensaje de error.

4.3. Actualización de Posición

El scanner mantiene información sobre la posición de línea y columna del texto analizado:

- **Método `updatePosition`**: Actualiza la línea y columna basándose en el texto escaneado.
- **Métodos `yyline` y `yycolumn`**: Devuelven la línea y columna actuales para proporcionar información precisa en los mensajes de error.

4.4. Producción de Tokens

El scanner produce tokens utilizando la clase `Symbol` del parser CUP. Cada token incluye:

- **Tipo (`type`)**: Identificador del tipo de token según la clase `sym`.
- **Valor (`value`)**: Valor léxico del token.
- **Posición (`left`, `right`)**: Posiciones de inicio y fin del token en el archivo de entrada.

4.5. Ejemplo de Definiciones en `Scanner.flex`

```java
// Palabras clave

"if" { updatePosition(yytext()); return createSymbol(sym.IF,
"if"); }
```

```
"else" { updatePosition(yytext()); return createSymbol(sym.ELSE,
"else"); }


// Operadores

"=" { updatePosition(yytext()); return createSymbol(sym.EQUALS,
"="); }

"==" { updatePosition(yytext()); return
createSymbol(sym.EQUALS_EQUALS, "=="); }


// Literales

[0-9]+ {

    updatePosition(yytext());

    String processedValue = Algebraic.processInteger(yytext());

    return createSymbol(sym.INTLIT, processedValue);

}
```

5. Integración con el CLI

Archivo: `compiler/Compiler.java`

5.1. Opciones del CLI

El CLI permite ejecutar diferentes etapas del compilador mediante las siguientes opciones:

- **`o <outname>`**: Especifica el archivo de salida.
- **`target <stage>`**: Define hasta qué etapa debe llegar el compilador. Las etapas disponibles son: `scan`, `parse`, `ast`, `semantic`, `irt`, `codegen`.
- **`opt <opt_stage>`**: Aplica optimizaciones específicas. Las etapas de optimización son: `constant`, `algebraic`.

- **`debug <stage>`**: Imprime información de depuración para las etapas especificadas. Las etapas de depuración pueden ser múltiples y se separan con `:`.

5.2. Ejemplo de Ejecución del CLI

```
Unset

java compiler -target scan -o output.txt input.txt
```

En este comando:

- **`target scan`**: Ejecuta solo la fase de escaneo.
- **`o output.txt`**: Guarda la salida en `output.txt`.

6. Consideraciones Adicionales

- **Errores de JFlex**: JFlex genera código fuente para el scanner pero no verifica su corrección. Asegúrate de que el archivo `.flex` esté libre de errores antes de compilar.
- **Integración con CUP**: Asegúrate de que los tokens generados por el scanner sean compatibles con los que espera el parser CUP.
- **Pruebas**: Realiza pruebas exhaustivas para verificar que el scanner maneja todos los tokens y errores correctamente. Utiliza una variedad de archivos de prueba para cubrir diferentes casos de uso.

7. Conclusión

La implementación del scanner para el lenguaje Decaf es crucial para el análisis léxico del compilador. Debe ser capaz de identificar correctamente los tokens, manejar errores léxicos y integrarse adecuadamente con el parser CUP. La documentación proporcionada detalla la estructura del proyecto, la implementación del scanner, y cómo integrarlo con el CLI para garantizar un análisis léxico efectivo.