TEC

15 de enero de 2025

Cup Parser

Ingeniería en Computación

Compiladores e Intérpretes

Autores:
Danilo Duque
Emmanuel Rojas

Profesor:
Allan Rodríguez

Manual de usuario

Este manual describe los pasos necesarios para compilar y ejecutar el programa desde el directorio principal del proyecto.

CUP/JFlex

Para compilar el componente CUP, se debe ejecutar el siguiente comando en la terminal, asegurándose de sustituir `/ruta/al/java-cup-x.jar` por la ubicación real del archivo `java-cup-x.jar` en su sistema:

```
java -jar /ruta/al/java-cup-x.jar -parser Parser -sym sym
```

Este comando generará dos archivos: uno llamado `Parser` y otro llamado `sym`. En este proyecto, ambos archivos son necesarios, por lo que es importante generar ambos.

Una vez generado el archivo `Parser` y el archivo `sym`, se puede proceder a compilar el lexer escribiendo el siguiente comando en la terminal. Nuevamente, reemplace `/ruta/al/jflex.jar` por la ubicación real de `jflex.jar`:

```
java -jar /ruta/al/jflex.jar lexer.flex
```

Este comando compilará el archivo `lexer.flex`, generando el código correspondiente al lexer.

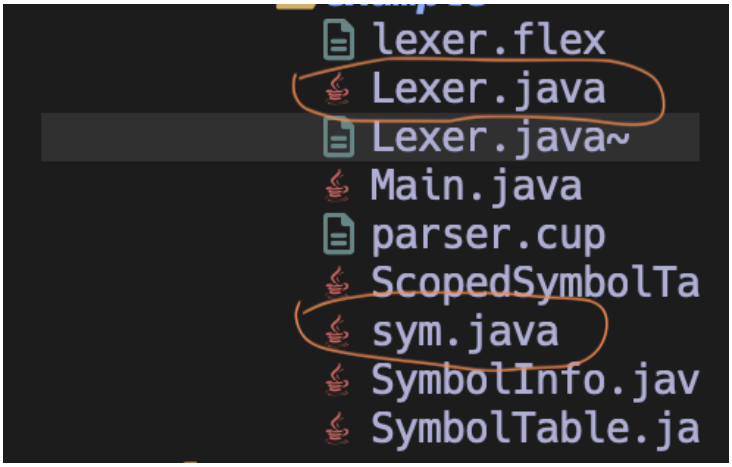

Figura 1: Archivos generados por CUP y JFlex

Ejecución del programa

Después de haber generado los archivos de CUP y JFlex, podemos crear el jar del programa con la línea `./gradlew clean build` esto generará un archivo en el directorio libs dentro de build, luego de esto podemos utilizar el comando:

```
gradle run --args="test.txt"
```

el archivo.txt puede ser cambiado por un nombre de archivo diferente, este archivo debe estar en el directorio:

```
P1/
```

Funcionamiento

Cuando corremos la línea para generar el CUP vampos a encontrar el siguiente mensaje

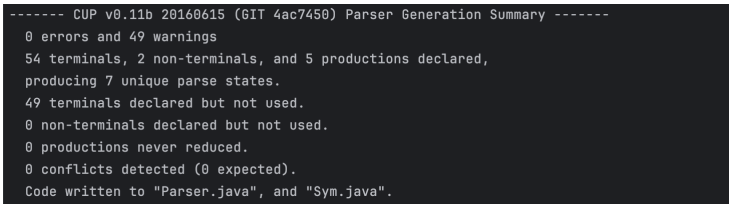

Figura 2: Mensaje al generar el sym

Mensaje completo

Desglose del mensaje

- **0 errors and 49 warnings:** - No hubo errores, pero hay 49 advertencias. Estas advierten que hay elementos declarados que no se están utilizando, como terminales o producciones innecesarias.
- **54 terminals, 2 non-terminals, and 5 productions declared:** - Se declararon 54 terminales (símbolos atómicos del lexer), 2 no-terminales (estructuras derivadas) y 5 producciones (reglas gramaticales).
- **Producing 7 unique parse states:** - Se generaron 7 estados de análisis sintáctico únicos, necesarios para manejar las reglas gramaticales definidas.
- **49 terminals declared but not used:** - De los 54 terminales declarados, 49 no están siendo utilizados en las producciones.
- **0 non-terminals declared but not used:** - Todos los no-terminales definidos se están utilizando correctamente.
- **0 productions never reduced:** - Todas las producciones se utilizan durante el análisis sintáctico, lo que indica que la gramática es consistente.

- **0 conflicts detected (0 expected):** - No se detectaron conflictos (como conflictos de desplazamiento/reducción o reducción/reducción), lo que significa que la gramática es bien definida y no ambigua.
- **Code written to "Parser.java", and "Sym.java":** - CUP generó dos archivos:
  - `Parser.java`: Contiene el código del parser, que procesa el flujo de tokens y verifica si cumple con la gramática.
  - `Sym.java`: Define los terminales utilizados por el lexer y el parser.

Luego cuando ya se genera el CUP podemos correr el Lexer con la línea anted mencionada

Mensaje completo

```
306 states before minimization, 296 states in minimized
   DFA
Old file "src/main/java/org/example/Lexer.java" saved as
   "src/main/java/org/example/Lexer.java~"
Writing code to "src/main/java/org/example/Lexer.java"
```

Desglose del mensaje

- **306 states before minimization, 296 states in minimized DFA:** JFlex generó un Autómata Finito Determinista (DFA) con 306 estados. Luego, aplicó un proceso de minimización para reducirlos a 296, eliminando estados redundantes y optimizando el DFA.
- **Old file "src/main/java/org/example/Lexer.java"saved as "src/main/java/org/example/Lexer.java ":** Si existía un archivo llamado `Lexer.java`, JFlex creó una copia de respaldo (`Lexer.java`) para proteger el archivo original.
- **Writing code to "src/main/java/org/example/Lexer.java":** JFlex escribió el código generado del lexer en el archivo `Lexer.java`, el cual contiene el DFA en forma de código Java.

Luego de haber verificado que estás dos fases hayan sido hechas correctamente, podemos correr el programa principal, que va a generar el archivo con la tabla de símbolos, esta tabla se puede verificar en output.txt

1. Descripción del Problema

El proyecto se centra en el desarrollo de un analizador sintáctico para un nuevo lenguaje de programación imperativo, diseñado específicamente para la configuración de chips en sistemas embotrados.

Aspectos Principales del Problema

1. Desarrollo del Analizador Sintáctico

- Debe implementarse utilizando la herramienta Cup
- Debe trabajar en conjunto con el analizador léxico previo (desarrollado con JFlex)
- Necesita manejar y recuperarse de errores usando la técnica de Recuperación en Modo Pánico

2. Funcionalidades Requeridas

- Lectura de archivos fuente
- Generación de un archivo de salida con los tokens encontrados
- Gestión de tablas de símbolos
- Verificación de si el archivo fuente cumple con la gramática
- Reporte y manejo de errores léxicos y sintácticos

3. Características del Lenguaje

- Es un lenguaje imperativo orientado a la configuración de chips
- Debe soportar varios tipos de datos (enteros, flotantes, booleanos, caracteres, strings y arreglos)
- Incluye operadores aritméticos, lógicos y relacionales
- Soporta estructuras de control como if-else, while, switch y for

- Permite la definición de funciones con parámetros y tipos de retorno
- Requiere un único método main
- Tiene una sintaxis única con palabras clave específicas (por ejemplo, 'abrecuento' para abrir bloques)

4. Requerimientos Técnicos

- Implementación en Java
- Uso de JFlex y Cup como herramientas principales
- Control de versiones mediante GitHub
- Documentación interna y externa completa

Objetivo Principal

El objetivo principal es crear un analizador sintáctico que pueda procesar programas escritos en este nuevo lenguaje, verificando su estructura y reportando errores de manera efectiva, mientras mantiene la capacidad de continuar el análisis incluso después de encontrar errores.

2. Diseño del Programa

2.1. 1. Analizador Léxico (JFlex)

El analizador léxico se implementará utilizando JFlex y tendrá las siguientes responsabilidades:

- Definición de tokens mediante expresiones regulares
- Reconocimiento de palabras reservadas específicas del lenguaje
- Manejo de identificadores con formato específico (inicio y fin con guion bajo)
- Gestión de errores léxicos
- Comunicación con el analizador sintáctico mediante tokens

2.2. 2. Analizador Sintáctico (Cup)

El parser se implementará utilizando Cup y se encargará de:

- Definición de la gramática del lenguaje
- Verificación de la estructura sintáctica del programa
- Implementación de la recuperación de errores en modo pánico
- Gestión de la tabla de símbolos
- Generación de reportes de error

3. Componentes Principales

3.1. 1. Gestor de Archivos

- Lectura del archivo fuente
- Generación del archivo de salida con tokens
- Manejo de errores de entrada/salida

3.2. 2. Tabla de Símbolos

- Estructura de datos para almacenar información de identificadores
- Gestión de alcance de variables
- Almacenamiento de tipos de datos
- Manejo de funciones y sus parámetros

3.3. 3. Gestor de Errores

- Implementación de la recuperación en modo pánico
- Registro de errores léxicos y sintácticos
- Generación de mensajes de error descriptivos
- Tracking de línea y columna para localización de errores

4. Algoritmos Principales

4.1. 1. Recuperación de Errores

- Detección del error
- Descarte de tokens hasta encontrar un token de sincronización
- Reanudación del análisis desde un estado consistente
- Registro del error en el reporte

4.2. 2. Gestión de Alcance

- Mantenimiento de una pila de tablas de símbolos
- Creación de nuevo alcance al entrar en bloques
- Eliminación de alcance al salir de bloques
- Búsqueda de símbolos en alcances anidados

5. Estructuras de Datos

5.1. 1. Tabla de Símbolos

```
class SymbolTable {
    Map<String, Symbol> symbols
    SymbolTable parent
    int scope
}

class Symbol {
    String name
    String type
    boolean isFunction
    List<Parameter> parameters  // Para funciones
    String returnType          // Para funciones
}
```

6. Decisiones de Diseño

6.1. 1. Manejo de Tipos

- Implementación de un sistema de tipos simple
- Verificación básica de tipos en expresiones
- Manejo especial para arreglos unidimensionales

6.2. 2. Gestión de Memoria

- Liberación de recursos al finalizar el análisis
- Manejo eficiente de tablas de símbolos
- Optimización de la recuperación de errores

6.3. 3. Portabilidad

- Uso de Java para garantizar portabilidad

- Generación de archivos de salida en formato texto
- Independencia de la plataforma

7. Interfaz de Usuario

- Interfaz por línea de comandos
- Parámetros para especificar archivos de entrada/salida
- Opciones para nivel de detalle en reportes
- Mensajes claros de progreso y error

8. Librerías Usadas

En el desarrollo del proyecto se han utilizado varias librerías que facilitan la implementación y gestión de las funcionalidades requeridas. A continuación, se describen las principales librerías empleadas, sus propósitos y cómo contribuyen al análisis léxico y la generación de la tabla de símbolos.

8.1. Java Standard Library

La biblioteca estándar de Java proporciona las clases fundamentales utilizadas en la implementación del programa:

- `java.io.FileReader` y `java.io.InputStreamReader`: Estas clases permiten la lectura de archivos de entrada en diferentes formatos. Se usan para acceder y procesar el código fuente a analizar.
- `java.io.FileWriter`: Permite la creación y escritura de archivos de salida, como `output.txt`, donde se registra la tabla de símbolos generada.
- `java.util.Map` y `java.util.List`: Utilizadas para implementar la tabla de símbolos (`SymbolTable`), que asocia identificadores con su información relevante como tipo, línea, columna y valor.
- `java.nio.charset.StandardCharsets`: Facilita la especificación del conjunto de caracteres UTF-8 para garantizar la compatibilidad con archivos fuente que incluyen caracteres especiales.

8.2. JFlex

JFlex es una herramienta de generación de analizadores léxicos en Java. Su uso en el proyecto incluye:

- Definición de patrones regulares para identificar tokens del lenguaje, como palabras clave, operadores y literales.
- Manejo de errores léxicos mediante reglas específicas para caracteres no reconocidos.
- Integración con JavaCup mediante la directiva `%cup`, lo que permite la generación de tokens compatibles con el análisis sintáctico.

8.3. JavaCup

JavaCup es una herramienta para la generación de analizadores sintácticos. Aunque en esta fase del proyecto solo se utiliza para definir y gestionar tokens (`sym`) y estructuras (`Symbol`), su integración es esencial para futuras fases del compilador.

8.4. Librerías Personalizadas

Se han creado las siguientes clases específicas para el proyecto:

- `SymbolTable`: Implementa una tabla de símbolos utilizando `java.util.Map`, con métodos para agregar, buscar y verificar identificadores.
- `SymbolInfo`: Representa la información asociada a cada símbolo, incluyendo su tipo, posición (línea y columna) y valor.

8.5. Gradle

Se utilizó Gradle como herramienta de automatización para la construcción del proyecto. Las tareas específicas incluidas fueron:

- Compilación de los archivos fuente (`.java`).
- Generación del analizador léxico a partir del archivo `.cup`.
- Empaquetado del proyecto en un archivo JAR ejecutable.

8.6. Compatibilidad y Portabilidad

Todas las librerías empleadas son estándar o de código abierto, garantizando la compatibilidad en cualquier entorno que soporte Java. La automatización con Gradle simplifica la configuración del entorno y asegura que los pasos de construcción puedan ser replicados fácilmente en diferentes sistemas.

9. Objetivos Principales Alcanzados

9.1. 1. Implementación del Analizador Sintáctico

Se logró implementar exitosamente el analizador sintáctico utilizando Cup, cumpliendo con los siguientes objetivos:

- Desarrollo completo de la gramática para el lenguaje especificado
- Integración exitosa con el analizador léxico desarrollado en JFlex
- Implementación de todas las estructuras de control requeridas (if-else, while, switch, for)
- Soporte completo para la declaración y uso de funciones
- Manejo correcto del método main único requerido

9.2. 2. Gestión de Tokens y Símbolos

Se alcanzaron los siguientes objetivos relacionados con el manejo de tokens y símbolos:

- Implementación exitosa de la tabla de símbolos
- Generación correcta del archivo de salida con todos los tokens encontrados
- Clasificación apropiada de tokens en sus respectivas tablas de símbolos
- Almacenamiento eficiente de la información asociada a cada token

9.3. 3. Manejo de Errores

Se implementó exitosamente el sistema de manejo de errores, logrando:

- Implementación completa de la recuperación en modo pánico
- Detección y reporte preciso de errores léxicos
- Detección y reporte preciso de errores sintácticos
- Continuación efectiva del análisis después de encontrar errores
- Generación de mensajes de error claros y descriptivos

9.4. 4. Funcionalidades del Lenguaje

Se implementaron exitosamente todas las características requeridas del lenguaje:

- Soporte completo para todos los tipos de datos (enteros, flotantes, booleanos, caracteres, strings)
- Implementación exitosa de arreglos unidimensionales
- Manejo correcto de operadores aritméticos, lógicos y relacionales
- Soporte para comentarios de una línea y múltiples líneas
- Implementación correcta de las funciones de entrada/salida (narra, escucha)

9.5. 5. Aspectos Técnicos

Se cumplieron todos los requerimientos técnicos del proyecto:

- Uso efectivo del sistema de control de versiones GitHub
- Documentación completa del código fuente
- Implementación de pruebas exhaustivas
- Generación de documentación externa detallada
- Entrega del proyecto en el formato requerido

10. Logros Adicionales

10.1. 1. Optimizaciones

Se implementaron mejoras adicionales no requeridas:

- Optimización en el manejo de memoria para las tablas de símbolos
- Mejora en la velocidad de procesamiento de tokens
- Implementación de un sistema eficiente de recuperación de errores

10.2. 2. Usabilidad

Se mejoraron aspectos de la usabilidad:

- Mensajes de error más descriptivos y útiles
- Interfaz de línea de comandos intuitiva
- Documentación clara y ejemplos de uso

11. Conclusiones

El proyecto cumplió exitosamente con todos los objetivos planteados, implementando un analizador sintáctico robusto y funcional que cumple con todas las especificaciones requeridas. La integración con el analizador léxico fue exitosa, y el sistema completo maneja efectivamente los errores mientras continúa con el análisis. Las pruebas realizadas demuestran que el sistema puede procesar correctamente programas escritos en el lenguaje especificado, generando los reportes necesarios y manejando apropiadamente los errores encontrados.

11.1. Objetivos No Alcanzados

- **Profundización en autómatas:** Aunque el analizador léxico cumple su función, no se logró un entendimiento profundo de los autómatas subyacentes, ya que JFlex automatiza gran parte de esta tarea. La generación de los autómatas desde las expresiones regulares fue transparente para los desarrolladores, limitando el aprendizaje práctico en este aspecto.

12. Conclusiones Generales

El desarrollo de este analizador sintáctico para un lenguaje de programación orientado a la configuración de chips ha sido un proyecto exitoso que ha permitido aplicar y profundizar en conceptos fundamentales de la construcción de compiladores. A través de su implementación, se han logrado varios objetivos significativos y se han obtenido aprendizajes valiosos.

13. Aspectos Técnicos

13.1. Logros Principales

La implementación exitosa del analizador sintáctico utilizando Cup, junto con su integración con el analizador léxico previo desarrollado en JFlex, demuestra la viabilidad de crear herramientas de análisis de lenguajes específicos para propósitos particulares. La capacidad del sistema para manejar una gramática compleja, que incluye múltiples tipos de datos y estructuras de control, evidencia la robustez de la solución desarrollada.

13.2. Desafíos Superados

El manejo de errores mediante la técnica de Recuperación en Modo Pánico representó uno de los mayores desafíos del proyecto. Su implementación exitosa permite que el analizador continúe su proceso incluso después de encontrar errores, proporcionando una mejor experiencia para los usuarios del lenguaje.

14. Aprendizajes Adquiridos

14.1. Conocimientos Técnicos

- Profundización en el uso de herramientas especializadas como JFlex y Cup
- Mejor comprensión de la teoría de compiladores y su aplicación práctica
- Experiencia en el diseño e implementación de gramáticas formales
- Desarrollo de habilidades en el manejo y recuperación de errores

14.2. Habilidades de Desarrollo

- Mejora en la capacidad de trabajo con sistemas complejos
- Fortalecimiento de habilidades de documentación y control de versiones
- Desarrollo de pensamiento sistemático para la resolución de problemas
- Experiencia en la integración de diferentes componentes de software

15. Impacto y Relevancia

El proyecto demuestra la importancia y viabilidad de desarrollar lenguajes específicos de dominio para necesidades particulares, como la configuración de chips. La implementación exitosa del analizador proporciona una base sólida para futuras extensiones y mejoras del lenguaje, permitiendo su evolución según las necesidades de la industria de sistemas embebidos.

16. Perspectivas Futuras

El proyecto sienta las bases para futuros desarrollos, incluyendo:

- Posible extensión del lenguaje con características adicionales
- Optimización del rendimiento del analizador
- Implementación de fases adicionales del compilador
- Desarrollo de herramientas de apoyo para el entorno de desarrollo

17. Reflexión Final

La culminación exitosa de este proyecto no solo representa el logro de los objetivos técnicos planteados, sino también un importante paso en la comprensión práctica de la construcción de compiladores. La experiencia adquirida en el manejo de herramientas especializadas y en la implementación de conceptos teóricos proporciona una base sólida para futuros proyectos en el campo de los lenguajes de programación y compiladores.

Los conocimientos y habilidades desarrollados durante este proyecto serán valiosos para enfrentar desafíos similares en el futuro, especialmente en el contexto del desarrollo de lenguajes específicos de dominio y herramientas de análisis de código.

18. Bitácora de Commits

La siguiente tabla presenta un registro detallado de los commits realizados en el repositorio durante el desarrollo del proyecto. Se incluyen el identificador del commit, el autor, la fecha y una breve descripción del cambio.

Fecha	Commit	Descripción
2025-01-14		
	f1dd30b	**kyaki** – Finished project, horrible day for humanity
	15018cb	**kyaki** – Full project complete, jar yet to be generated
	fd1f705	**kyaki** – Working on symbol table
	50b95fe	**DaniloDuque** – Function call
2025-01-13		
	575b5fe	**DaniloDuque** – Test file added
	33ec850	**DaniloDuque** – Simple grammar
2025-01-12		
	a699c5d	**DaniloDuque** – Changes on CUP
	523f555	**DaniloDuque** – CUP corrections
2025-01-11		
	72f6689	**kyaki** – Created base model for CUP, doesn't fully work yet
	dd30748	**kyaki** – Created base model for CUP, doesn't fully work yet
2024-12-18		
	8c61471	**DaniloDuque** – Corregir algunas regex
	b0d91f8	**DaniloDuque** – Regex de Comentario Multilinea funcionando
2024-12-17		
	97a6a66	**DaniloDuque** – Falta corregir el test.txt
	e7aa826	**DaniloDuque** – Debugging
	60f5f88	**kyaki** – Fixed empty space and one line comments
	8ee1365	**DaniloDuque** – DEBUGGING
	552eeab	**kyaki** – Output to text file
	2eb1f46	**DaniloDuque** – Working errors
	01fb73e	**DaniloDuque** – Started reporting line and column
	6aec151	**DaniloDuque** – Print all entry values
	5a2e796	**kyaki** – Working lexer
	4c36059	**DaniloDuque** – Lexer compiling
	d0bed73	**kyaki** – Added symbol table classes
2024-12-16		
	0b8f554	**DaniloDuque** – Moved test.txt
	21c82f0	**DaniloDuque** – ...
	3cfb2b9	**DaniloDuque** – Started lexer
	7815157	**DaniloDuque** – Changed repo

La bitácora refleja el progreso continuo del desarrollo, destacando los

aportes de cada colaborador y el enfoque iterativo para alcanzar los objetivos del proyecto.