Instituto Tecnológico de Costa Rica

Lenguajes de Programación

Análisis Léxico y Sintáctico

PY #01

Integrantes:

Bayron Rodriguez Centeno - 2020114659

Gadir Calderón Díaz - 2022327328

Profesor:

Allan Rodríguez Dávila

Grupo 60

I Semestre

Año 2025

Tabla de Contenidos

Tabla de Contenidos **2**
Manual de Usuario **2**
Requisitos del Sistema 2
Estructura del Proyecto 3
Instrucciones de Compilación 3
1. Generar el Analizador Léxico 3
2. Generar el Analizador Sintáctico 3
3. Compilar el Proyecto 3
Instrucciones de Ejecución 4
Salida Esperada 4
Formato del Archivo de Entrada 4
Pruebas de Funcionalidad **4**
Prueba 1: Declaración de Variables y Estructuras de control 5
Prueba 2: Funciones y Expresiones Complejas: 5
Prueba 3: Manejo de Errores 6
Descripción del problema **6**
Diseño del programa **6**
Arquitectura General 6
Decisiones de Diseño 7
Scanner (Analizador Léxico) 7
Parser (Analizador Sintáctico) 7
Main (Programa Principal) 7
Algoritmos Principales 7
Librerías usadas **7**
Herramientas Principales 7
Bibliotecas Estándar de Java 8
Análisis de Resultados **8**

Manual de Usuario

Requisitos del Sistema

- Java JDK 11 o superior
- JFlex 1.9,1
- Cub 11b
- Sistema operativo compatible con Java(Windows, Linux, macOS)

Estructura del Proyecto

```
proyecto/
├── build/              # Archivos compilados
├── lib/                # Bibliotecas necesarias
│   ├── java-cup-11b.jar
│   └── jflex-1.9.1.jar
├── src/
│   └── main/
│       ├── cup/
│       │   └── Parser.cup
│       ├── flex/
│       │   └── Scanner.flex
│       ├── java/
│       │   ├── lexer/
│       │   │   └── Scanner.java
│       │   ├── main/
│       │   │   └── Main.java
│       │   └── parser/
│       │       ├── parser.java
│       │       └── sym.java
│       └── resources/
│           └── test5.txt      # Archivo de prueba
└── README.md
```

Instrucciones de Compilación

1. Generar el Analizador Léxico

   ```
   java -jar lib/jflex-full-1.9.1.jar src/main/flex/Scanner.flex
   -d src/main/java/lexer/
   ```

2. Generar el Analizador Sintáctico

   ```
   java -jar lib/java-cup-11b.jar -parser parser -symbols sym
   -package parser src/main/cup/Parser.cup
   ```

3. Compilar el Proyecto

   ```
   javac -cp "lib/*" -d build src/main/java/main/*.java
   src/main/java/lexer/*.java src/main/java/parser/*.java
   ```

Instrucciones de Ejecución

Para ejecutar el analizador con un archivo de prueba:

```
java -cp "build;lib/*" main.java.main.Main src/main/resources/test5.txt
```

Nota: En sistemas Unix (Linux/macOS), reemplazar ';' por ':' en el classpath:

```
java -cp "build:lib/*" main.java.main.Main src/main/resources/test5.txt
```

Salida Esperada

El programa generará dos tipos de resultados:

1. **Análisis Léxico:** Se generará un archivo con el nombre del archivo de entrada seguido de "_tokens-txt" que contiene todos los tokens identificados, con el siguiente formato:
   Token: ID, Lexema: variable, Línea: 5, Columna: 1
2. **Análisis Sintáctico:** En la consola se mostrará el resultado del análisis sintáctico correcto o mostrando los errores encontrados.

Formato del Archivo de Entrada

Los archivos de entrada deben seguir la sintaxis del lenguaje definido en la gramática del proyecto. A continuación, un ejemplo básico:

```
void main ʃ ɭ \
  int x | 10?
  if ʃ x > 5 ɭ \
    write ʃ "x es mayor que 5" ɭ?
  /
/
```

Pruebas de Funcionalidad

Para ilustrar el funcionamiento del compilador, se han realizado pruebas con diferentes archivos de entrada que cubren diversas características del lenguaje:

Prueba 1: Declaración de Variables y Estructuras de control

Entrada (test1.txt):

```
void main ʃ ɭ \
  int x | 10?
  int y | 20?

  if ʃ x < y ɭ \
    write ʃ "x es menor que y" ɭ?
  /
/
```

Resultado del Análisis Léxico (test1_tokens.txt):

Token: VOID, Lexema: void, Línea: 1, Columna: 1

Token: MAIN, Lexema: main, Línea: 1, Columna: 6

Token: LPAREN, Lexema: ʃ, Línea: 1, Columna: 11

Token: RPAREN, Lexema: ɭ, Línea: 1, Columna: 12

Token: LBLOCK, Lexema: \, Línea: 1, Columna: 14

Token: INT, Lexema: int, Línea: 2, Columna: 3

Token: ID, Lexema: x, Línea: 2, Columna: 7

...

Resultados del Análisis Sintáctico:

Analizando léxicamente el archivo: src/main/resources/test1.txt

Análisis léxico completado. Tokens escritos en: src/main/resources/test1_tokens.txt

Analizando sintácticamente el archivo: src/main/resources/test1.txt

Análisis sintáctico completado sin errores.

Prueba 2: Funciones y Expresiones Complejas:

Entrada(test2.text):

```
int suma ʃ int a, int b ɭ \
  return a + b?
/

void main ʃ ɭ \
  int resultado | suma ʃ 5, 10 ɭ?
```

```
  write ∫ "El resultado es: " ⌊?
  write ∫ resultado ⌊?
/
```

Resultados similares al anterior, mostrando tokens y éxito en el análisis sintáctico.

Prueba 3: Manejo de Errores

Entrada(test3.txt):

```
void main ∫ ⌊ \
  int x | 10
  if ∫ x > 5 ⌊ \
    write ∫ "x es mayor que 5" ⌊?
  /
/
```

Resultado del Análisis Sintáctico:

Error sintáctico en línea 3, columna 3: if

Análisis sintáctico completado con errores.

Descripción del problema

Un grupo de desarrolladores desea crear un nuevo lenguaje imperativo, ligero, que le permita realizar operaciones básicas para la configuración de chips, ya que esta es una industria que sigue creciendo constantemente, y cada vez estos chips necesitan ser configurados por lenguajes más ligeros y potentes. Es por esto por lo que este grupo de desarrolladores requiere desarrollar su propio lenguaje para el desarrollo de sistemas empotrados.

Diseño del programa

Arquitectura General

El proyecto sigue una arquitectura de compilador de dos fases:

1. **Análisis Léxico:** Implementado mediante JFlex, convierte la secuencia de caracteres del archivo fuente en una secuencia de tokens.

2. **Análisis Sintáctico:** Implementado mediante Cup. analiza la secuencia de tokens para verificar que cumple con la gramática definida

Decisiones de Diseño

Scanner (Analizador Léxico)

- **Expresiones Regulares:** Se definieron expresiones regulares para cada token del lenguaje
- **Manejo de Estados:** Se implementó un sistema simple de estados para manejar comentarios multilínea
- **Recuperación de Errores:** Al encontrar un carácter no reconocido, se reporta el error y continúa con el análisis.
- **Información adicional:** Por cada token, se guarda información de su línea y columna para facilitar el reporte de errores.

Parser (Analizador Sintáctico)

- **Definición de Símbolos**: Se definieron claramente los símbolos terminales y no terminales.
- **Precedencia de Operaciones**: Se estableció la precedencia y asociatividad de los operadores siguiendo reglas matemáticas estándar.
- **Reglas de producción:** Se implementaron reglas para todas las construcciones sintácticas del lenguaje
- **Recuperación de errores:** Se implementó la técnica de Recuperación en Modo Pánico, reportando errores y continuando con el análisis.

Main (Programa Principal)

- **Gestión de archivos**: Manejo de archivos de entrada y salida
- **Integración:** Integración entre el analizador léxico y sintáctico.
- **Reporte de Resultados**: Generación de archivos de salida y mensajes de consola

Algoritmos Principales

1. **Análisis Léxico**: Se utiliza un autómata finito determinista (generado por JFlex) para reconocer los tokens.
2. **Análisis Sintáctico**: Se utiliza un analizador LR(1) (Generado por Cup) para verificar la estructura del programa.
3. **Reporte de Errores**: Implementación de handlers personalizados para reportar errores de manera informativa

Librerías usadas

Herramientas Principales

1. **JFlex(1.9.1):** Generador de analizadores léxicos para Java.
    a. Uso: Generación del Scanner.flex.
    b. Sitio web: **https://jflex.de/**
2. **Cup(11b):** Generador de analizadores sintacticos LALR para Java.
    a. Uso: Generación del Parser a partir de especificaciones en parser.cup
    b. Sitio web: **http://www2.cs.tum.edu/projects/cup/**

Bibliotecas Estándar de Java

1. **java.io:** Para manejo de archivos y streams de entrada/salida.
    a. Clases principales utilizadas: FileReader, FileWriter, PrintWriter.
2. **java.util**: Para estructuras de datos y utilidades.
    a. Clases principales utilizadas: ArrayList, HashMap.

Análisis de Resultados

Objetivos	Alcanzados	No Alcanzados	
Análisis Léxico	X		Se implementó

Completo			exitosamente el scanner capaz de reconocer todos los tokens definidos en la gramática, incluyendo: • Palabras clave • Identificadores • Literales (enteros, flotantes, caracteres, strings, booleanos) • Operadores (aritméticos, relacionales, lógicos) • Delimitadores • Comentarios
Análisis Sintáctico Robusto	X		Se logró implementar un parser que verifica correctamente la estructura sintáctica del programa según la gramática definida, incluyendo: • Declaración de variables y funciones • Expresiones aritméticas,

			relacionales y lógicas con correcta precedencia • Estructuras de control • Arreglos bidimensionales • Llamadas a funciones
Manejo de Errores	X		Se implementó un sistema de reporte de errores que proporciona información útil sobre la ubicación y naturaleza de los errores encontrados.
Generación de Salidas	X		El sistema genera correctamente los archivos de salida con la información de los tokens y reportes de análisis sintáctico.

Bitácora

https://github.com/Bayronjrc/PP1_BayronR_GadyrC.git

TEC | Tecnológico de Costa Rica