MANUAL TÉCNICO

Aplicación Android – Intérprete de Pseudocódigo con Generación de Diagramas de Flujo

Autor: Alex Domingo

Curso: Compiladores

1. Organización del Proyecto

El proyecto fue desarrollado utilizando Android Studio con lenguaje Kotlin. La arquitectura está organizada bajo una separación clara entre interfaz de usuario, compilador (análisis léxico y sintáctico) y modelos de datos.

1.1 Estructura General

```
alex-domingo-practica---compi/
├── app/
│   ├── src/main/java/com/example/decompi/
│   │   ├── MainActivity.kt
│   │   ├── compiler/
│   │   │   ├── Compiler.kt
│   │   │   ├── lexer.flex
│   │   │   ├── Lexer.java
│   │   │   ├── parser.cup
│   │   │   ├── sym.java
│   │   │   └── models/
│   │   │       ├── AstNodes.kt
│   │   │       └── ReportModels.kt
```

Descripción de componentes:
- MainActivity.kt: Interfaz principal.
- Compiler.kt: Orquestador del proceso de compilación.
- lexer.flex: Definición del analizador léxico (JFlex).
- parser.cup: Definición del analizador sintáctico (CUP).
- AstNodes.kt: Definición de nodos del Árbol Sintáctico Abstracto (AST).
- ReportModels.kt: Modelos de reportes y errores.

2. Análisis Léxico (JFlex)

El analizador léxico fue desarrollado utilizando JFlex. Su función es convertir la cadena de entrada en una secuencia de tokens que posteriormente serán utilizados por el analizador sintáctico.

2.1 Tokens Reconocidos

Palabras reservadas:
VAR, SI, ENTONCES, FINSI, MIENTRAS, HACER, FINMIENTRAS, MOSTRAR, LEER, INICIO, FIN

Operadores aritméticos:
*+, -, *, /*

Operadores relacionales:
==, !=, >, <, >=, <=

Operadores lógicos:
&&, ||, !

Símbolos especiales:
( ), , , |, %%%%

Literales:
- NUMBER (Double)
- STRING
- HEX_COLOR
- ID

2.2 Expresiones Regulares Definidas

```
Integer = [0-9]+
Decimal = [0-9]+ '.' [0-9]* | '.' [0-9]+
Identifier = [a-zA-Z][a-zA-Z0-9_]*
StringLiteral = " ([^"\\] | \\.)* "
HexColor = 'H' [0-9a-fA-F]{6}
Comment = '#' [^\r\n]*
```

El analizador léxico también mantiene información de línea y columna para la generación de reportes de errores.

3. Análisis Sintáctico (CUP)

El analizador sintáctico fue desarrollado utilizando Java CUP. Se definieron terminales, no terminales y reglas de producción que construyen un Árbol Sintáctico Abstracto (AST).

3.1 Símbolo Inicial

```
program ::= INICIO full_instruction_list FIN SEPARATOR config_list
```

3.2 Reglas Principales

```
instruction ::= simple_instruction
        | SI ( condition ) ENTONCES simple_instruction_list FINSI
        | MIENTRAS ( condition ) HACER simple_instruction_list FINMIENTRAS
```

expression ::= expression + expression
| expression - expression
*| expression * expression*
| expression / expression
| NUMBER
| ID

3.3 Precedencia de Operadores

1. OR
2. AND
3. NOT
4. Operadores relacionales
5. + y -
*6. * y /*

3.4 Manejo de Errores

Se implementó el método syntax_error para capturar errores sintácticos y almacenarlos en una lista de CompilationError.

El diseño sigue el patrón de Árbol Sintáctico Abstracto (AST), permitiendo separar el análisis sintáctico de la ejecución o generación del diagrama.