DOCUMENTACIÓN TÉCNICA

Desarrollo con ANTLR4 y Visitor
Fecha: 26 de febrero de 2026
Índice

1. Visión General del Proyecto

El presente documento describe la implementación de un intérprete de expresiones aritméticas desarrollado con ANTLR4 y el patrón de diseño Visitor.

A diferencia de una calculadora tradicional, este sistema:

- Procesa múltiples instrucciones desde archivo
- Permite definir y reutilizar variables
- Aplica precedencia de operadores automáticamente
- Evalúa expresiones mediante recorrido de árbol sintáctico

La arquitectura se divide en tres componentes principales:

- Gramática del lenguaje (LabeledExpr.g4)
- Programa ejecutor (Calc.java)
- Evaluador semántico (EvalVisitor.java)

2. Diseño del Lenguaje

El lenguaje implementado permite:

- Operaciones básicas: +, -, *, /
- Agrupación mediante paréntesis
- Asignación de variables
- Evaluación línea por línea

Ejemplo válido:

```
x = 10
y = 4
x + y * 2
(3 + 1) * 5
```

3. Definición de la Gramática

Regla Principal

La regla inicial establece que un programa está compuesto por una secuencia de instrucciones:

```
prog : stat+ ;
```

Esto indica que debe existir al menos una instrucción.

Tipos de Instrucciones

```
stat
   : expr NEWLINE              # printStatement
   | ID '=' expr NEWLINE       # assignment
   | NEWLINE                   # emptyLine
   ;
```

Se contemplan tres casos:

- Evaluación directa
- Asignación de variable
- Línea vacía

Expresiones Aritméticas

```
expr
   : expr op=('*'|'/') expr    # multiplication
   | expr op=('+'|'-') expr    # addition
   | INT                       # number
   | ID                        # variable
   | '(' expr ')'              # grouped
   ;
```

La precedencia se controla gracias al orden de definición recursiva.

Multiplicación y división tienen mayor prioridad que suma y resta.

4. Generación Automática del Parser

El siguiente comando permite generar las clases necesarias:

antlr4 -visitor -no-listener LabeledExpr.g4

Archivos generados:

- LabeledExprLexer.java
- LabeledExprParser.java
- LabeledExprBaseVisitor.java
- LabeledExprVisitor.java

La opción -visitor habilita la generación del patrón Visitor.
La opción -no-listener evita código innecesario.

5. Programa Principal (Calc.java)

Responsabilidades principales:

- Leer archivo de entrada
- Crear el Lexer
- Crear el Parser
- Generar el árbol sintáctico
- Ejecutar el Visitor evaluador

Flujo simplificado:

```
CharStream input = CharStreams.fromFileName(args[0]);
LabeledExprLexer lexer = new LabeledExprLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
LabeledExprParser parser = new LabeledExprParser(tokens);
ParseTree tree = parser.prog();

EvalVisitor eval = new EvalVisitor();
eval.visit(tree);
```

6. Implementación del Visitor

La clase:

public class EvalVisitor extends LabeledExprBaseVisitor<Integer>

Cada visita retorna un valor entero.

Gestión de Variables

Se utiliza:

Map<String, Integer> memory = new HashMap<>();

Permite almacenar valores asignados.

Evaluación de Asignación

Se evalúa la expresión.

Se guarda el resultado en el mapa.

No se imprime automáticamente.

Evaluación de Expresiones

Las operaciones se resuelven recursivamente:

Ejemplo multiplicación:

```
if (ctx.op.getType() == LabeledExprParser.MUL)
   return left * right;
```

7. Flujo Completo de Ejecución

Se crea archivo entrada.expr

Se ejecuta:

java Calc entrada.expr

El lexer tokeniza

El parser construye el árbol

El visitor recorre y evalúa

Se imprimen resultados

8. Caso de Prueba

Contenido del archivo:

100
a = 7
b = 3
a + b * 2
(2 + 3) * 4

Salida esperada:

100
13
20

9. Validación

✓ Precedencia respetada
✓ Variables almacenadas correctamente
✓ Paréntesis evaluados primero
✓ Ejecución desde archivo funcional

10. Conclusión

La combinación de ANTLR4 con el patrón Visitor permite:

- Separar sintaxis de semántica
- Construir mini lenguajes fácilmente

- Evaluar expresiones con claridad estructural
- Mantener código modular y escalable

El proyecto demuestra cómo implementar un intérprete simple pero robusto utilizando herramientas modernas de generación de analizadores sintácticos.