Documentación Técnica
Calculadora con ANTLR4

26 de febrero de 2026

Índice

1. Introducción

El presente documento describe el funcionamiento de una calculadora implementada utilizando ANTLR4 y el patrón de diseño *Visitor*. El sistema permite:

- Evaluar expresiones aritméticas.
- Asignar valores a variables.
- Imprimir resultados.
- Manejar precedencia de operadores.

La implementación está compuesta por:

- Archivo de gramática: `LabeledExpr.g4`
- Programa principal: `Calc.java`
- Visitador evaluador: `EvalVisitor.java`

2. Gramática: LabeledExpr.g4

La gramática define la estructura sintáctica del lenguaje de la calculadora.

2.1. Regla inicial

```
prog: stat+ ;
```

La regla `prog` indica que un programa consiste en una o más sentencias.

2.2. Reglas de sentencias

```
stat:   expr NEWLINE                # printExpr
    |   ID '=' expr NEWLINE         # assign
    |   NEWLINE                     # blank
    ;
```

Se definen tres tipos de sentencias:

- **printExpr**: evalúa e imprime una expresión.
- **assign**: asigna el valor de una expresión a una variable.
- **blank**: línea en blanco.

2.3. Reglas de expresiones

```
expr:   expr op=('*'|'/') expr      # MulDiv
    |   expr op=('+'|'-') expr      # AddSub
    |   INT                         # int
    |   ID                          # id
    |   '(' expr ')'                # parens
    ;
```

La gramática soporta:

- Multiplicación y división.
- Suma y resta.
- Números enteros.
- Variables.
- Paréntesis.

ANTLR4 maneja la precedencia de operadores gracias al orden de definición de las reglas recursivas.

2.4. Tokens léxicos

```
MUL : '*' ;
DIV : '/' ;
ADD : '+' ;
SUB : '-' ;
ID  : [a-zA-Z]+ ;
INT : [0-9]+ ;
NEWLINE:'\r'? '\n' ;
WS  : [ \t]+ -> skip ;
```

Estos tokens permiten identificar operadores, identificadores, números enteros y espacios en blanco.

3. Generación del Parser

Se ejecutó el siguiente comando:

```
antlr4 -no-listener -visitor LabeledExpr.g4
```

Este comando genera:

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

La opción `-visitor` genera las clases necesarias para implementar el patrón Visitor.

La opción `-no-listener` evita generar clases Listener, ya que no se utilizan en esta implementación.

4. Programa Principal: Calc.java

El flujo principal es:

1. Leer la entrada (archivo o consola).
2. Crear el lexer.
3. Crear el parser.
4. Generar el árbol sintáctico.
5. Ejecutar el visitor.

Fragmento relevante:

```
ParseTree tree = parser.prog();
EvalVisitor eval = new EvalVisitor();
eval.visit(tree);
```

5. Evaluación con Visitor: EvalVisitor.java

La clase extiende:

```
public class EvalVisitor extends LabeledExprBaseVisitor<Integer>
```

Esto indica que cada visita devuelve un valor de tipo `Integer`.

5.1. Memoria de Variables

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

Se utiliza un mapa para almacenar pares variable-valor.

5.2. Asignación

```
public Integer visitAssign(...) {
    String id = ctx.ID().getText();
    int value = visit(ctx.expr());
    memory.put(id, value);
    return value;
}
```

Evalúa la expresión y guarda el resultado en memoria.

5.3. Impresión

```
public Integer visitPrintExpr(...) {
    Integer value = visit(ctx.expr());
    System.out.println(value);
    return 0;
}
```

Evalúa la expresión y muestra el resultado.

5.4. Operaciones Aritméticas

Multiplicación y división:

```
public Integer visitMulDiv(...) {
    int left = visit(ctx.expr(0));
    int right = visit(ctx.expr(1));
    if (ctx.op.getType() == LabeledExprParser.MUL)
        return left * right;
    return left / right;
}
```

Suma y resta:

```
public Integer visitAddSub(...) {
    int left = visit(ctx.expr(0));
    int right = visit(ctx.expr(1));
    if (ctx.op.getType() == LabeledExprParser.ADD)
        return left + right;
    return left - right;
}
```

6. Flujo de Ejecución Completo

El flujo de ejecución del programa utilizando un archivo de entrada es el siguiente:

1. Se crea un archivo de texto (por ejemplo, `t.expr`) que contiene las expresiones a evaluar.
2. El programa se ejecuta mediante el comando:

   ```
   java Calc t.expr
   ```

3. La clase `Calc` abre el archivo y lo utiliza como flujo de entrada (`InputStream`).
4. El Lexer analiza el contenido del archivo y lo divide en tokens.
5. El Parser procesa los tokens y construye el árbol sintáctico (Parse Tree).
6. El *EvalVisitor* recorre el árbol sintáctico.
7. Las expresiones se evalúan recursivamente.

8. Los resultados se imprimen en consola conforme se procesan las sentencias.

De esta manera, toda la ejecución depende del contenido del archivo de entrada y no de interacción directa por consola.

7. Ejecución con Archivo de Entrada

Para realizar pruebas con múltiples expresiones, se creó un archivo de texto utilizando:

```
nano t.expr
```

El contenido del archivo `t.expr` es el siguiente:

```
193
a = 5
b = 6
a+b*2
(1+2)*3
```

Posteriormente, se ejecutó el programa utilizando el archivo como entrada:

```
java Calc t.expr
```

7.1. Salida Obtenida

La salida generada por el programa fue:

```
193
17
9
```

7.2. Análisis de Resultados

- **193**: Se imprime directamente el número ingresado.
- **17**: Resultado de la expresión $a + b * 2$.

  Dado que:

  $$a = 5, \quad b = 6$$
  $$b * 2 = 12$$
  $$a + 12 = 17$$

  Se respeta correctamente la precedencia de operadores (multiplicación antes que suma).
- **9**: Resultado de $(1 + 2) * 3$.

  Primero se resuelve el paréntesis:

  $$1 + 2 = 3$$

  Luego:

  $$3 * 3 = 9$$

Esto confirma que:

- El archivo es leído correctamente por `Calc.java`.
- El Lexer y Parser funcionan adecuadamente.
- El Visitor evalúa correctamente las expresiones.
- Se respeta la precedencia y el uso de paréntesis.