# Informe de Laboratorio:

## Implementación de una Calculadora usando ANTLR v4 y el Patrón Visitor

**Materia:**
Lenguajes de Programación

**Presentado por:**
Santiago Diaz

**Fecha:**
26 de febrero de 2026

# Índice

# 1. Introducción

El presente informe documenta el proceso de construcción de una calculadora interactiva utilizando **ANTLR v4** (ANother Tool for Language Recognition), basado en el capítulo 4.2 del libro *The Definitive ANTLR 4 Reference*.

El objetivo principal fue definir una gramática para expresiones aritméticas y sentencias de asignación, generar el analizador léxico (Lexer) y sintáctico (Parser), e implementar la evaluación de las expresiones a través de un recorrido del árbol de sintaxis abstracta (AST) utilizando el patrón de diseño *Visitor* en el lenguaje Java.

# 2. Descripción del Problema

Se requiere construir un intérprete capaz de procesar un archivo de texto secuencial que contiene:

- Evaluaciones aritméticas básicas (suma, resta, multiplicación, división).
- Manejo de precedencia de operadores mediante el uso de paréntesis.
- Asignación de variables en memoria (ej. `a = 5`).
- Uso de las variables previamente asignadas en nuevas expresiones matemáticas.

# 3. Desarrollo e Implementación

La solución se dividió en tres componentes principales: la definición de la gramática, la implementación de la lógica de evaluación (Visitor) y el programa principal que orquesta la ejecución.

## 3.1. Definición de la Gramática (`LabeledExpr.g4`)

Se creó una gramática que define la estructura del lenguaje de la calculadora. Se utilizaron etiquetas (precedidas por el símbolo `#`) en las alternativas de las reglas sintácticas. Estas etiquetas son fundamentales, ya que instruyen a ANTLR para generar métodos de visita específicos para cada tipo de operación, facilitando el procesamiento en Java.

```
grammar LabeledExpr;

prog:   stat+ ;

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

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

```
    ;

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

Listing 1: Gramática LabeledExpr.g4

## 3.2. Generación del Código ANTLR

A partir del archivo de gramática, se utilizó el archivo ejecutable `antlr-4.13.2-complete.jar` para generar el código base en Java. El comando utilizado fue:

```
java -jar antlr-4.13.2-complete.jar -no-listener -visitor LabeledExpr.g4
```

El parámetro `-visitor` indica a la herramienta que genere las interfaces y clases base correspondientes al patrón Visitor, y `-no-listener` desactiva la generación del patrón Listener, el cual es el comportamiento por defecto pero no es requerido para este ejercicio.

## 3.3. Implementación del Visitor (`EvalVisitor.java`)

Se implementó la clase `EvalVisitor`, la cual hereda de la clase autogenerada `LabeledExprBaseVisit` Esta clase sobreescribe los métodos generados por las etiquetas de la gramática para dotar al analizador de semántica computacional. Se utilizó un `HashMap` de Java para almacenar las variables creadas durante el análisis.

```
import java.util.HashMap;
import java.util.Map;

public class EvalVisitor extends LabeledExprBaseVisitor<Integer> {
    Map<String, Integer> memory = new HashMap<String, Integer>();

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

    @Override
    public Integer visitPrintExpr(LabeledExprParser.PrintExprContext ctx
    ) {
        Integer value = visit(ctx.expr());
        System.out.println(value);
        return 0;
    }
    // ... Implementacion de visitMulDiv, visitAddSub, etc.
```

```
}
```

Listing 2: Fragmento de EvalVisitor.java

### 3.4. Programa Principal (`Calc.java`)

Se desarrolló una clase principal para conectar el flujo de datos: leer el archivo de texto de entrada, pasar los caracteres al Lexer, suministrar los tokens al Parser, generar el árbol de sintaxis abstracta (Parse Tree) e iniciar el recorrido utilizando el `EvalVisitor`.

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import java.io.FileInputStream;
import java.io.InputStream;

public class Calc {
    public static void main(String[] args) throws Exception {
        String inputFile = args[0];
        InputStream is = new FileInputStream(inputFile);

        ANTLRInputStream input = new ANTLRInputStream(is);
        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);
    }
}
```

Listing 3: Estructura básica de Calc.java

## 4. Compilación y Ejecución

Para probar la herramienta, se definió un archivo de entrada llamado `entrada.txt` con las siguientes expresiones:

```
193
a = 5
b = 6
a+b*2
(1+2)*3
548
5*10
```

*Nota: Se requiere un salto de línea al final del archivo para cumplir con la regla* ***`NEWLINE`*** *definida en la gramática para el último bloque* ***`stat`****.*

### 4.1. Proceso de Compilación en Linux

Para evitar conflictos con el *classpath* del sistema, la compilación se realizó enlazando explícitamente el archivo `.jar` descargado en el directorio local:

```
javac -cp ".:antlr-4.13.2-complete.jar" *.java
```

### 4.2. Ejecución de Pruebas

Se ejecutó el programa invocando la máquina virtual de Java e inyectando las librerías de tiempo de ejecución de ANTLR:

```
java -cp ".:antlr-4.13.2-complete.jar" Calc entrada.txt
```

**Salida obtenida por consola:**

```
193
17
9
548
50
```

Los resultados evidencian un correcto funcionamiento de las operaciones y del manejo de precedencia matemática implementado recursivamente en el AST por el Visitor.

## 5. Conclusiones

- **Desacoplamiento de la Gramática:** El uso del patrón Visitor con ANTLR v4 demostró ser una práctica altamente eficiente, ya que permite mantener la gramática completamente limpia e independiente del código de la aplicación. En versiones anteriores (como ANTLR v3), las acciones debían incrustarse dentro de la propia gramática.
- **Facilidad de la recursividad por la izquierda:** ANTLR v4 maneja implícitamente la recursividad por la izquierda (como se vio en la regla `expr`), lo que permitió definir la precedencia matemática de forma natural y legible.
- **Importancia del Classpath:** Durante la implementación en un entorno Linux, se evidenció que es indispensable tener control sobre las versiones de ejecución de Java, asegurando que los archivos autogenerados y el proceso de compilación usen la misma versión exacta del `jar` de ANTLR (en este caso 4.13.2).