

**TECNOLÓGICO NACIONAL DE MÉXICO
CAMPUS CIUDAD HIDALGO**



Manual Tecnico

MATERIA: Lenguajes y Automatas II

UNIDAD 2

**ALUMNO: Juan Pablo Salazar Rodríguez, Enrique
Mauro Muños Alanis, Luis Cuahutemoc sachez, Erwin
Santiago Arreaga Avila, Matias Alejandro Mendoza
Gonzalez**

077C

DOCENTE: Esmeralda Arreola Marin

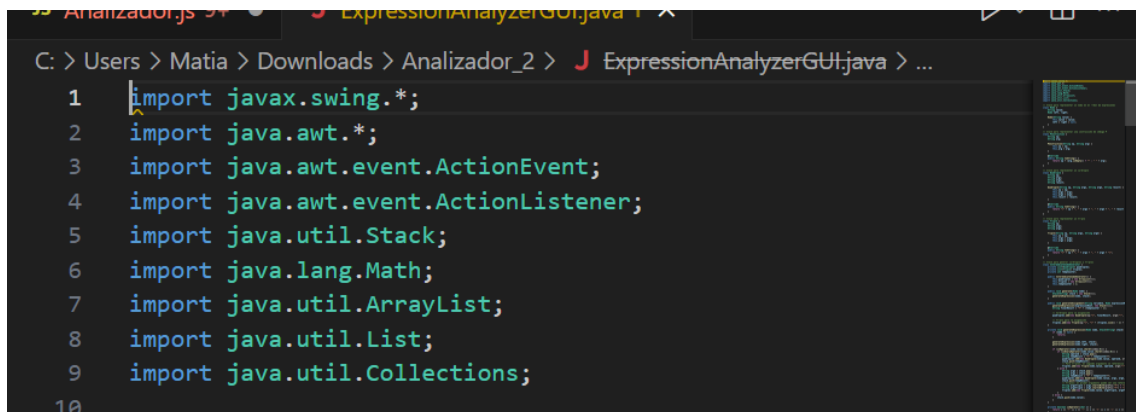
FECHA DE ENTREGA 2 de octubre Del 2025

Introduccion:

En este documento se mostrarán las librerías tanto las operaciones y otras cosas del analizador léxico 2

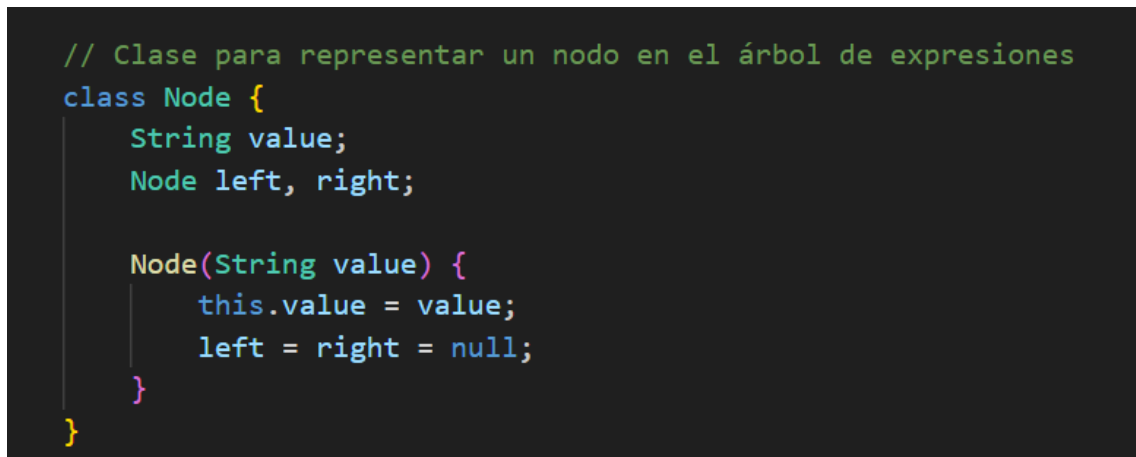
Fragmentos del código

Libreras: aquí se muestran lo que son las librerías de java.



```
C: > Users > Matia > Downloads > Analizador_2 > J ExpressionAnalyzerGUI.java > ...  
1  import javax.swing.*;  
2  import java.awt.*;  
3  import java.awt.event.ActionEvent;  
4  import java.awt.event.ActionListener;  
5  import java.util.Stack;  
6  import java.lang.Math;  
7  import java.util.ArrayList;  
8  import java.util.List;  
9  import java.util.Collections;  
10
```

Clases para representar expresiones



```
// Clase para representar un nodo en el árbol de expresiones  
class Node {  
    String value;  
    Node left, right;  
  
    Node(String value) {  
        this.value = value;  
        left = right = null;  
    }  
}
```

Clase para representar una instrucción del código P

```

}

// Clase para representar una instrucción de código P
class PInstruction {
    String op;
    String arg;

    PInstruction(String op, String arg) {
        this.op = op;
        this.arg = arg;
    }

    @Override
    public String toString() {
        return op + (arg.isEmpty() ? "" : " " + arg);
    }
}

```

Clase para representar un cuádruplo

```

3 // Clase para representar un cuádruplo
4 class Quadruple {
5     String op;
6     String arg1;
7     String arg2;
8     String result;
9
10    Quadruple(String op, String arg1, String arg2, String result) {
11        this.op = op;
12        this.arg1 = arg1;
13        this.arg2 = arg2;
14        this.result = result;
15    }
16
17    @Override
18    public String toString() {
19        return "(" + op + ", " + arg1 + ", " + arg2 + ", " + resu
20    }
21 }

```

Clase para representar un triplo

```
// Clase para representar un tríplo
class Triple {
    String op;
    String arg1;
    String arg2;

    Triple(String op, String arg1, String arg2) {
        this.op = op;
        this.arg1 = arg1;
        this.arg2 = arg2;
    }

    @Override
    public String toString() {
        return "(" + op + ", " + arg1 + ", " + arg2 + ")";
    }
}
```

Clase para generar cuádruplos y triplos

```
// Clase para generar cuádruplos y triplos
class IntermediateCodeGenerator {
    private List<Quadruple> quadruples;
    private List<Triple> triples;
    private int tempCounter;

    public IntermediateCodeGenerator() {
        this.quadruples = new ArrayList<>();
        this.triples = new ArrayList<>();
        this.tempCounter = 1;
    }

    public void generate(Node node) {
        Stack<String> stack = new Stack<>();
        generateExpression(node, stack);
    }

    public void generateAssignment(String variable, Node expression) {
        generateExpression(expressionRoot, new Stack<>());
        String finalResult = "T" + (tempCounter - 1);
    }
}
```

Clase para triplos de segundo argumento

```
triples.add(new Triple(node.value, operand, arg2:
} else {
    String arg2 = stack.pop();
    String arg1 = stack.pop();
    String temp String tempResult - IntermediateCodeG
    quadruples. Stack<String>
    stack.push(tempResult);
```

Clase para generar la clase de triplos de primer argumento

```

class IntermediateCodeGenerator {
    private void generateExpression(Node node, Stack<String> stack, List<Triple> triples, List<Quadruple> quadruples) {
        // Para tríplos, el segundo argumento se referencia
        triples.add(new Triple(node.value, operand, arg2));
    } else {
        String arg2 = stack.pop();
        String arg1 = stack.pop();
        String tempResult = "T" + tempCounter++;
        quadruples.add(new Quadruple(node.value, arg1, arg2, tempResult));
        stack.push(tempResult);
        // Tríplos: el primer argumento puede ser una referencia
        String arg1Triple = arg1.startsWith(prefix:"T") ? arg1 : "T" + tempCounter++;
        String arg2Triple = arg2.startsWith(prefix:"T") ? arg2 : "T" + tempCounter++;
        triples.add(new Triple(node.value, arg1Triple, arg2Triple));
    }
} else {
    stack.push(node.value);
}
}

private boolean isOperator(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/' || c == '^';
}

private boolean isUnaryOperator(char c) {
    return c == 'v';
}

public List<Quadruple> getQuadruples() {
    return quadruples;
}
}

```

Clase para generar código P a partir del árbol de expresiones

```
// Clase que genera el código P a partir del árbol de expresiones
class PCodeGenerator {

    private List<PInstruction> pCode;

    public PCodeGenerator() {
        this.pCode = new ArrayList<>();
    }

    public void generate(Node node) {
        if (node == null) {
            return;
        }
    }
}
```

Recorrido del árbol en postfijo

```
// Recorrido posfijo para generar el código
generate(node.left);
generate(node.right);

if (isOperator(node.value.charAt(index:0))) {
    switch (node.value) {
        case "+":
            pCode.add(new PInstruction(op:"OPR", arg:"ADD"));
            break;
        case "-":
            pCode.add(new PInstruction(op:"OPR", arg:"SUB"));
            break;
        case "*":
            pCode.add(new PInstruction(op:"OPR", arg:"MUL"));
            break;
        case "/":
            pCode.add(new PInstruction(op:"OPR", arg:"DIV"));
            break;
        case "^":
            pCode.add(new PInstruction(op:"OPR", arg:"POW"));
            break;
        case "√":
            pCode.add(new PInstruction(op:"OPR", arg:"SQRT"));
            break;
        default:
            throw new IllegalArgumentException("Operador no válido");
    }
}
```

Recorrido posfijo para generar el código

```

// Recorrido post-ijo para generar el código
generate(node.left);
generate(node.right);

if (isOperator(node.value.charAt(index:0))) {
    switch (node.value) {
        case "+":
            pCode.add(new PInstruction(op:"OPR", arg:"ADD
            break;
        case "-":
            pCode.add(new PInstruction(op:"OPR", arg:"SUB
            break;
        case "*":
            pCode.add(new PInstruction(op:"OPR", arg:"MUL
            break;
        case "/":
            pCode.add(new PInstruction(op:"OPR", arg:"DIV
            break;
        case "^":
            pCode.add(new PInstruction(op:"OPR", arg:"POW
            break;
        case "√":
            pCode.add(new PInstruction(op:"OPR", arg:"SQR
            break;
        default:
            throw new IllegalArgumentException("Operador
    }

```

Clase para manejar la lógica de validación, conversión y construcción del árbol


```

class ExpressionParser {

    private String originalExpression;
    private String expressionType;

    public ExpressionParser(String expression) {
        this.originalExpression = expression.replaceAll(regex:"\\
this.validateExpression();
        this.expressionType = detectType();
    }

    public Node getRootNode() {
        String postfix = toPostfix();
        return buildFromPostfix(postfix);
    }

    private String toPostfix() {
        switch (expressionType) {

```

Clase para la interfaz gráfica

```

// Clase para la interfaz gráfica
public class ExpressionAnalyzerGUI extends JFrame {

    private JTextField expressionField;
    private JTextArea resultArea;
    private JTextArea treeVisualizationArea;
    private JTextArea pCodeArea;
    private JTextArea intermediateCodeArea;

    public ExpressionAnalyzerGUI() {
        super(title:"Analizador de Expresiones y Código Intermedi
        createGUI();
    }

    private void createGUI() {
        setLayout(new BorderLayout(hgap:10, vgap:10));

        JPanel inputPanel = new JPanel();
        inputPanel.setLayout(new FlowLayout());
        inputPanel.add(new JLabel(text:"Introduce una expresión o
        expressionField = new JTextField(columns:20);
        inputPanel.add(expressionField);

```

Código para Verificar si es una asignación

```
// Verificar si es una asignación
if (expression.contains(s:"=")) {
    String[] parts = expression.split(regex:"=");
    if (parts.length != 2 || parts[0].trim().isEmpty() || parts[1].trim().isEmpty()) {
        throw new IllegalArgumentException(s:"Formato incorrecto");
    }
    String variable = parts[0].trim();
    String expr = parts[1].trim();

    ExpressionParser parser = new ExpressionParser(expr);
    Node root = parser.getRootNode();

    // Mostrar resultados de conversión de la expresión
    resultArea.setText("Asignación detectada: " + variable + " = " + expr);
}
```

Código para Mostrar resultados de conversión de la expresión

```
// Mostrar resultados de conversión de la expresión
resultArea.setText("Asignación detectada: " + variable + " = " + expr);
resultArea.append("Posfija: " + parser.postOrder(root));
resultArea.append("Prefija: " + parser.preOrder(root));
resultArea.append("Infija: " + parser.inOrder(root));
```

Código para Mostrar visualización del árbol

```
treeVisualizationArea.setText("Árbol de Expresión");
printTree(root, prefix:"", isTail:true);
```

Generar y mostrar los cuádruplos y trípteros de la asignación

```

// Generar y mostrar los cuádruplos y tríplos de
IntermediateCodeGenerator iCodeGenerator = new In
iCodeGenerator.generateAssignment(variable, root)
List<Quadruple> quadruples = iCodeGenerator.getQu
List<Triple> triples = iCodeGenerator.getTriples(

intermediateCodeArea.setText(t:"Cuádruplos:\n");
for (Quadruple q : quadruples) {
    intermediateCodeArea.append(q.toString() + "\n"
}
intermediateCodeArea.append(str:"\nTríplos:\n");
int i = 0;
for (Triple t : triples) {

```

Generar y mostrar el código P

```

PCodeGenerator pCodeGenerator = new PCodeGenerator();
pCodeGenerator.generate(root);
List<PInstruction> pCode = pCodeGenerator.getPCode();
pCodeArea.setText(t:"P-Code:\n");
for (PInstruction inst : pCode) {
    pCodeArea.append(inst.toString() + "\n");
}

```

Generar y mostrar los cuádruplos y tríplos

```
// Generar y mostrar los cuádruplos y tríplos
IntermediateCodeGenerator iCodeGenerator = new IntermediateCodeGenerator();
iCodeGenerator.generate(root);
List<Quadruple> quadruples = iCodeGenerator.getQuadruples();
List<Triple> triples = iCodeGenerator.getTriples();

intermediateCodeArea.setText(t:"Cuádruplos:\n");
for (Quadruple q : quadruples) {
    intermediateCodeArea.append(q.toString() + "\n");
}
intermediateCodeArea.append(str:"\nTríplos:\n");
int i = 0;
for (Triple t : triples) {
    intermediateCodeArea.append(i++ + ": " + t.toString() + "\n");
}
```