TECNOLÓGICO NACIONAL DE MÉXICO
CAMPUS CIUDAD HIDALGO

Comparacion analizador léxico

MATERIA: Lenguajes y Automatas ll

UNIDAD 2

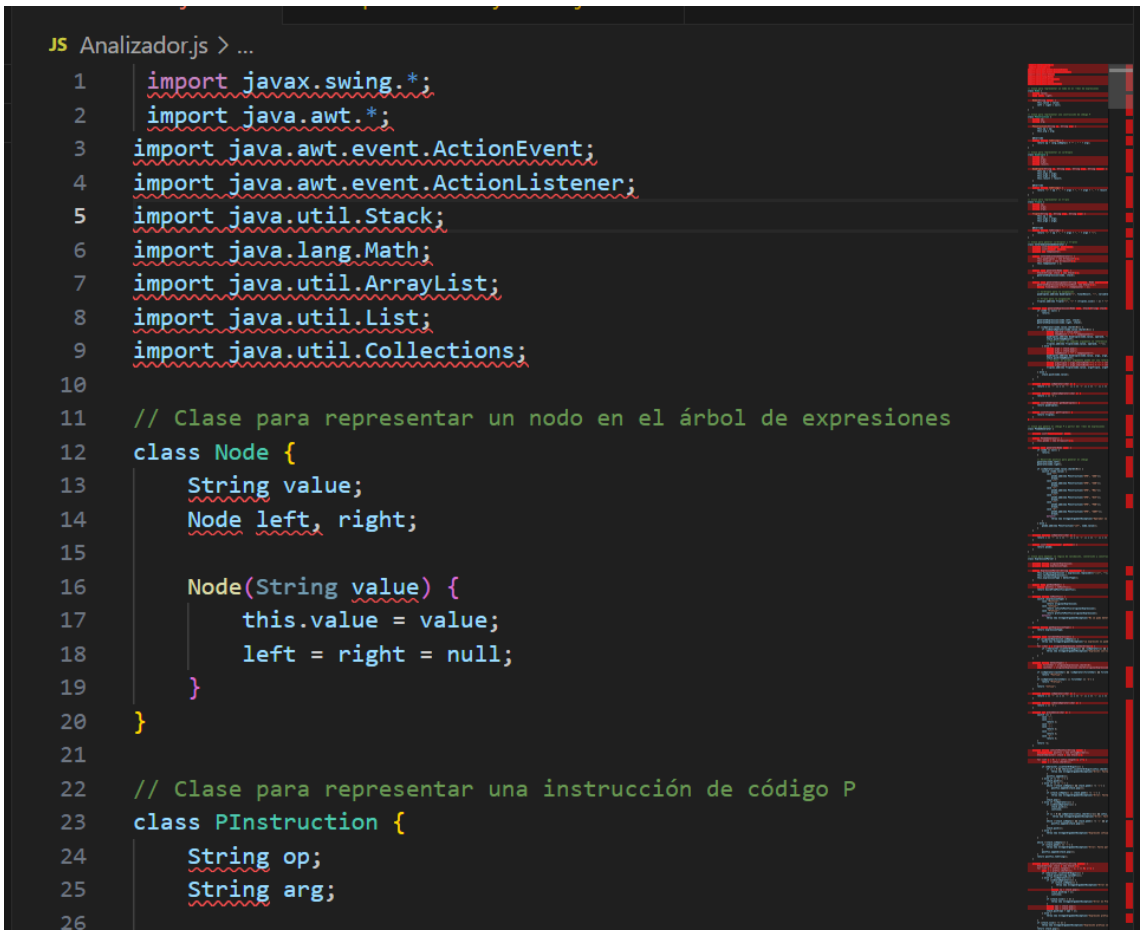
ALUMNO: Juan Pablo Salazar Rodríguez,Enrique Mauro Muños Alanis,Luis Cuahutemoc sachez,Erwin Santiago Arreaga Avila,Matias Alejandro Mendoza Gonzalez

077C

DOCENTE: Esmeralda Arreola Marin

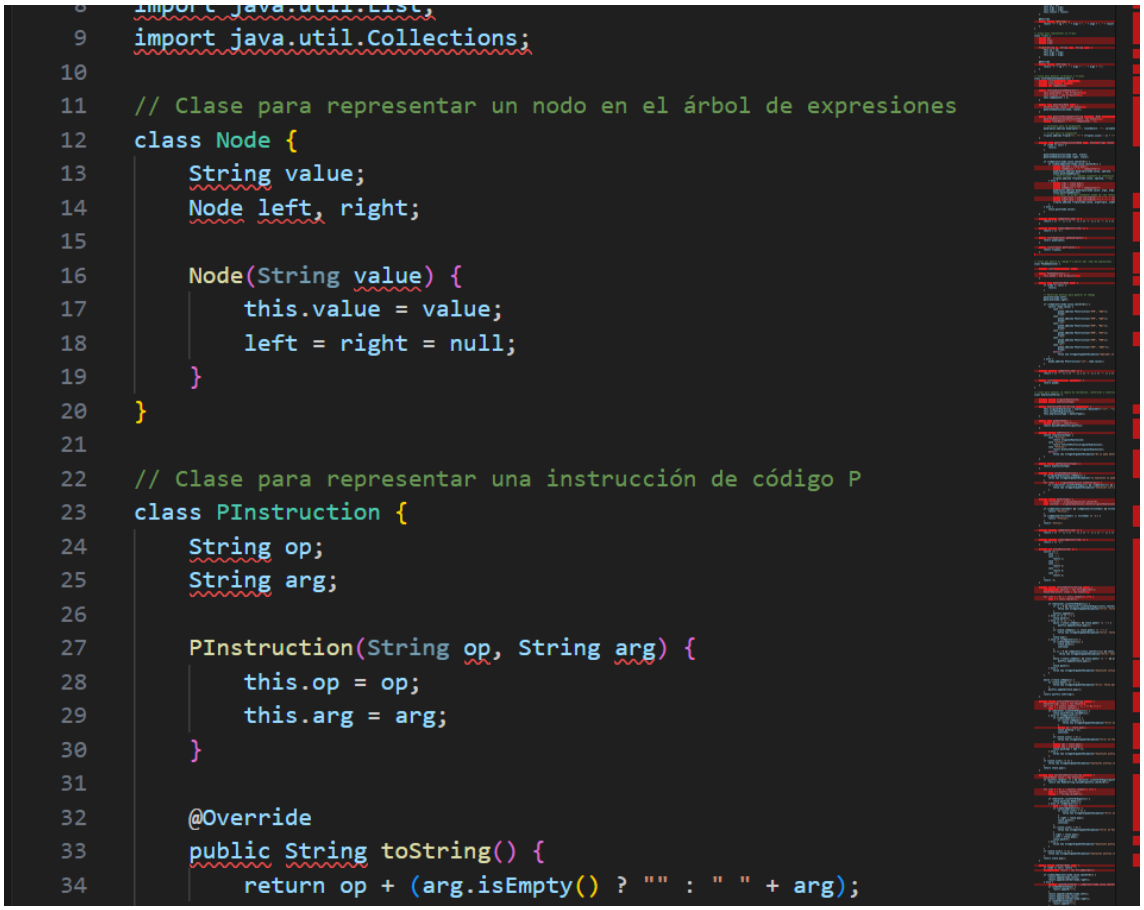
FECHA DE ENTREGA 2 de octubre Del 2025

Analizador léxico Lenguajes y autómatas I

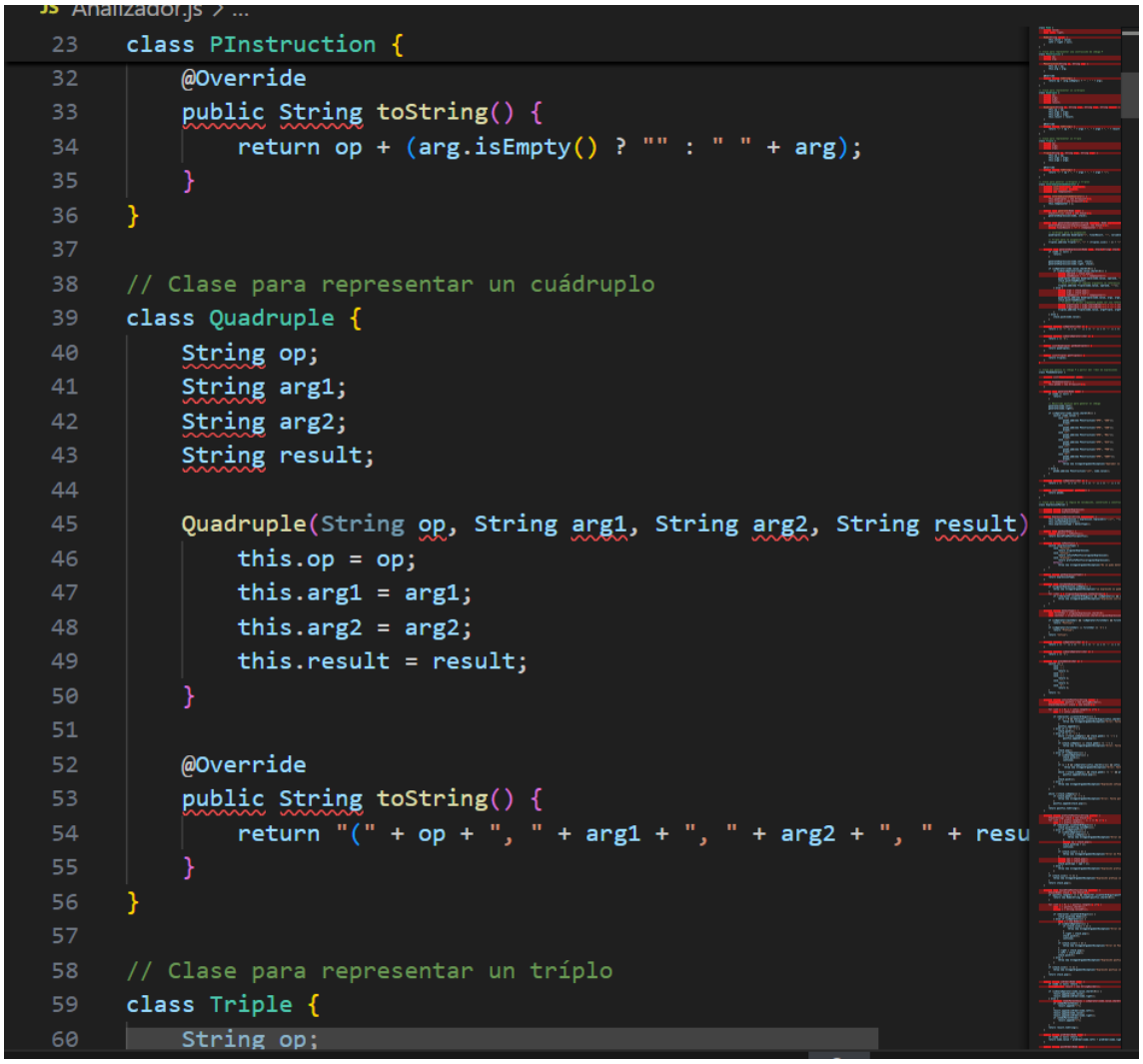


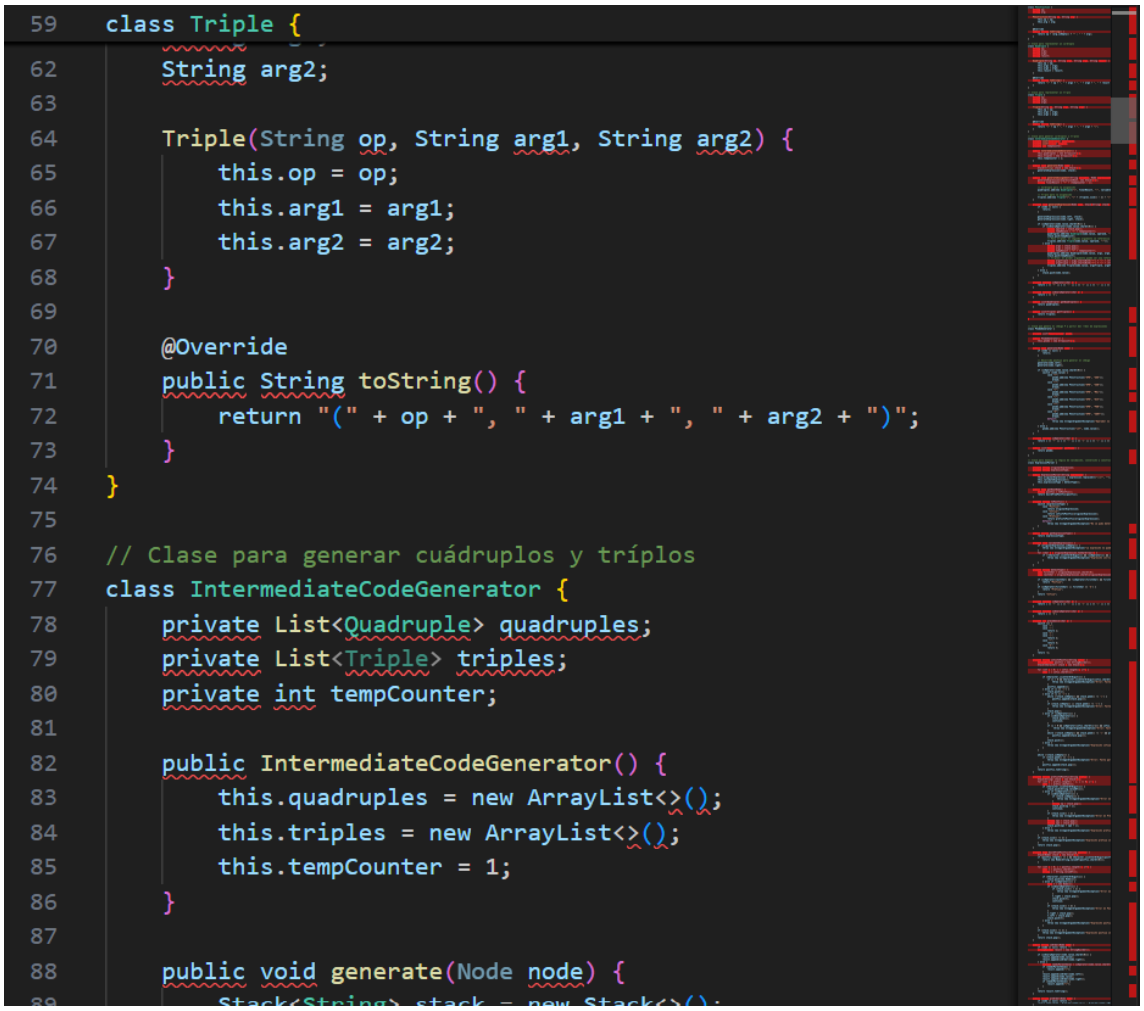
Analizador.js > ...

```
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
11  // Clase para representar un nodo en el árbol de expresiones
12  class Node {
13      String value;
14      Node left, right;
15
16      Node(String value) {
17          this.value = value;
18          left = right = null;
19      }
20  }
21
22  // Clase para representar una instrucción de código P
23  class PInstruction {
24      String op;
25      String arg;
26
```

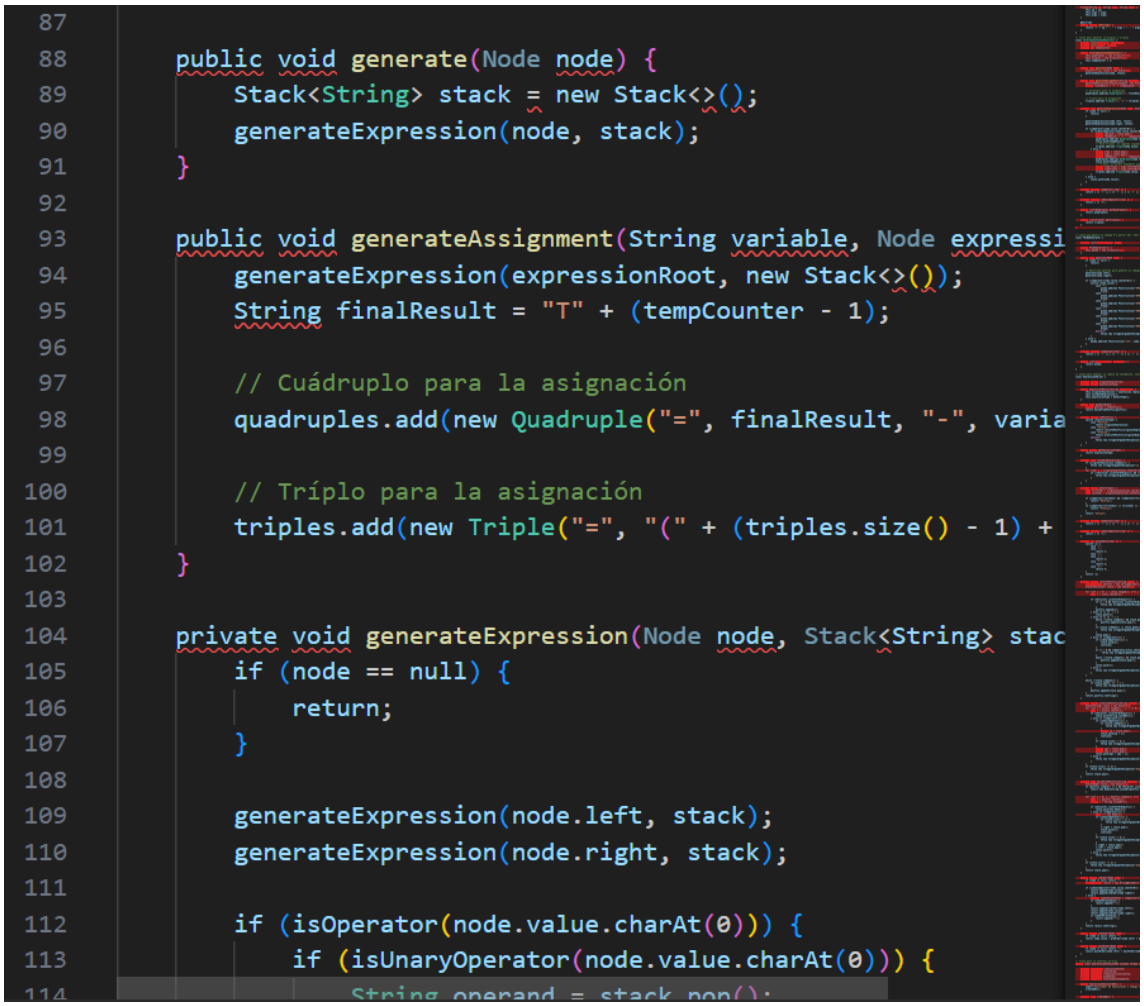


```
import java.util.List;
import java.util.Collections;

// 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 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);
```

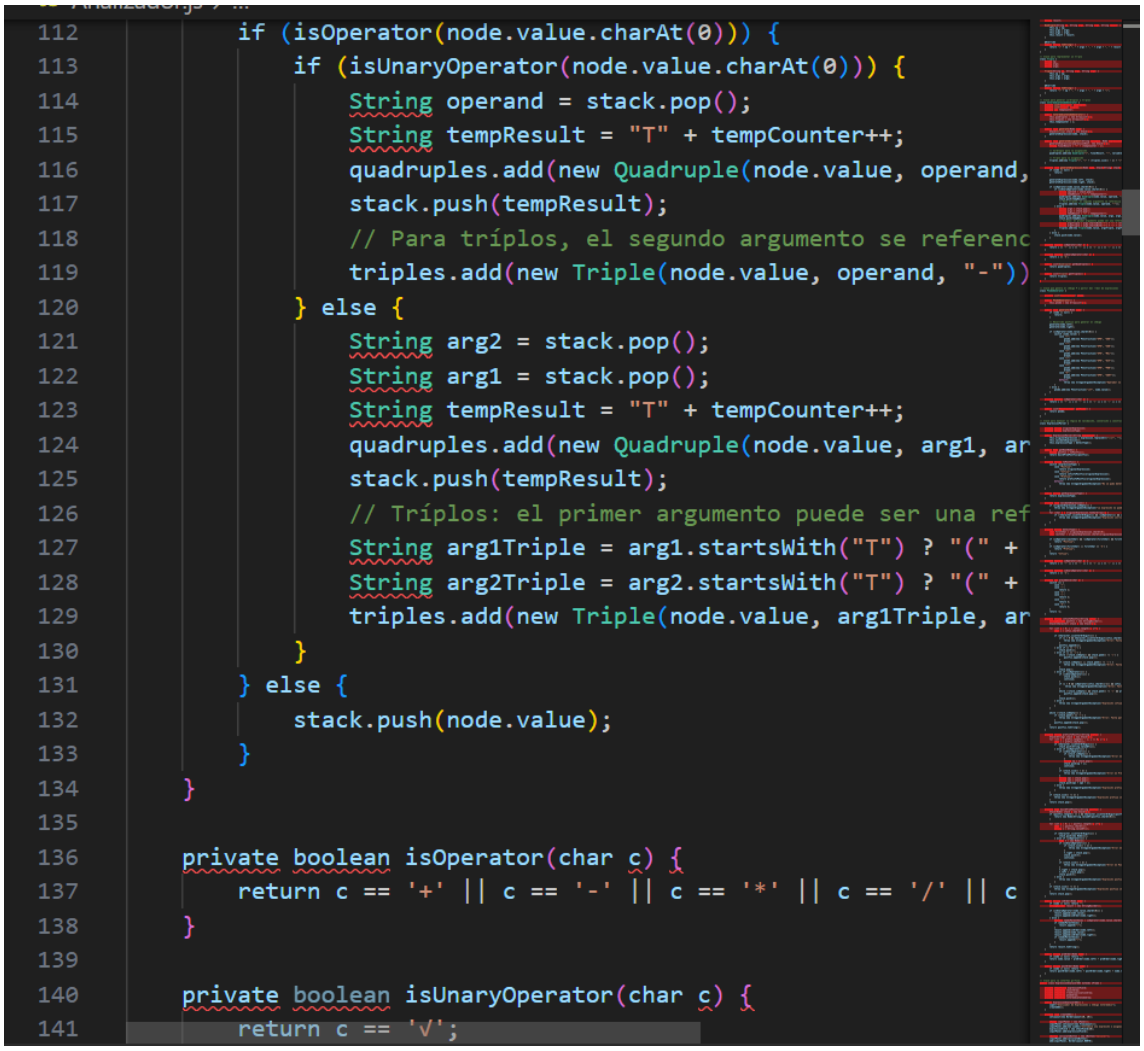

Analizador.js > ...
23 class PInstruction {
32 @Override
33 public String toString() {
34 return op + (arg.isEmpty() ? "" : " " + arg);
35 }
36 }
37
38 // Clase para representar un cuádruplo
39 class Quadruple {
40 String op;
41 String arg1;
42 String arg2;
43 String result;
44
45 Quadruple(String op, String arg1, String arg2, String result)
46 this.op = op;
47 this.arg1 = arg1;
48 this.arg2 = arg2;
49 this.result = result;
50 }
51
52 @Override
53 public String toString() {
54 return "(" + op + ", " + arg1 + ", " + arg2 + ", " + resu
55 }
56 }
57
58 // Clase para representar un tríplo
59 class Triple {
60 String op;



```
59  class Triple {
62      String arg2;
63
64      Triple(String op, String arg1, String arg2) {
65          this.op = op;
66          this.arg1 = arg1;
67          this.arg2 = arg2;
68      }
69
70      @Override
71      public String toString() {
72          return "(" + op + ", " + arg1 + ", " + arg2 + ")";
73      }
74  }
75
76  // Clase para generar cuádruplos y tríplos
77  class IntermediateCodeGenerator {
78      private List<Quadruple> quadruples;
79      private List<Triple> triples;
80      private int tempCounter;
81
82      public IntermediateCodeGenerator() {
83          this.quadruples = new ArrayList<>();
84          this.triples = new ArrayList<>();
85          this.tempCounter = 1;
86      }
87
88      public void generate(Node node) {
```

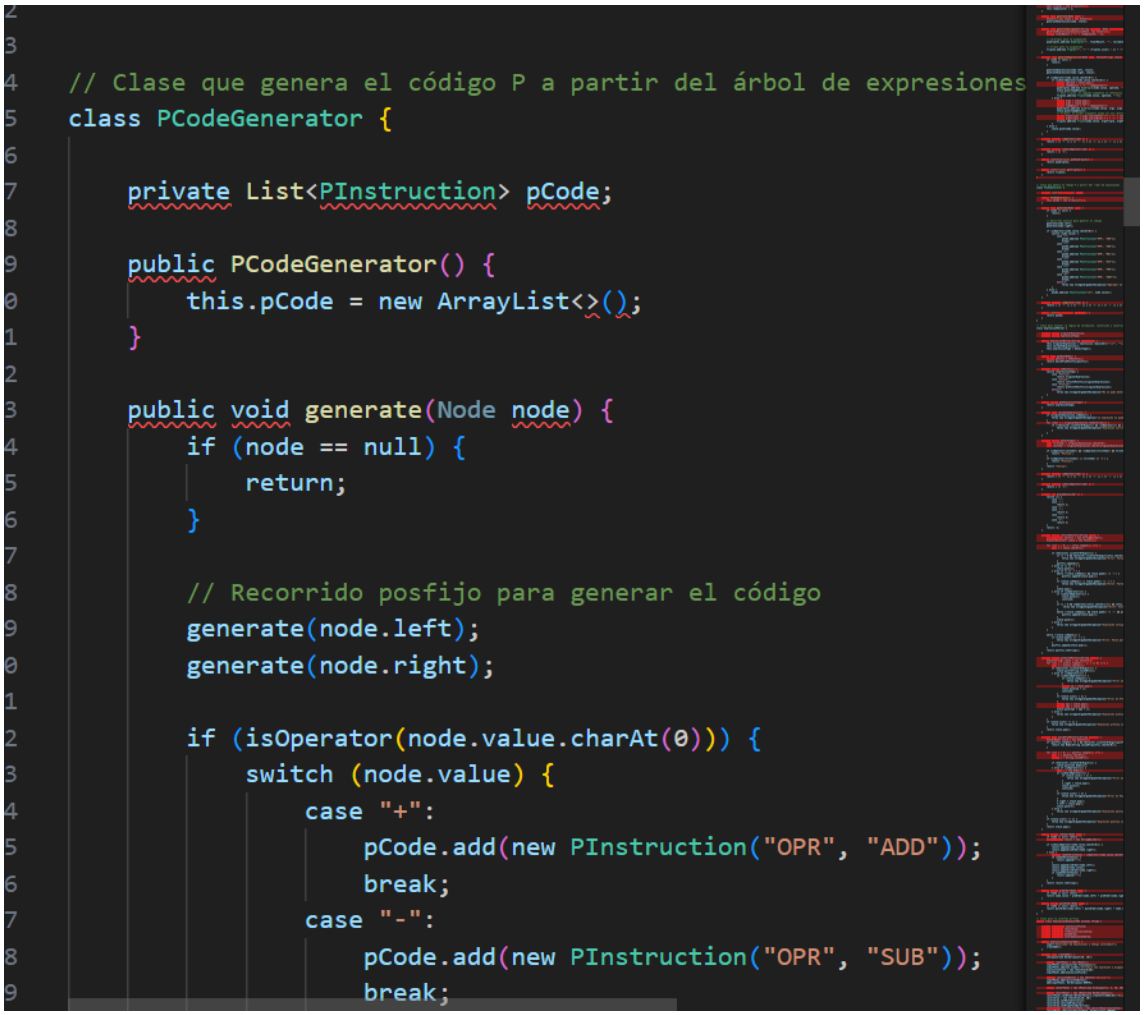


```
87
88      public void generate(Node node) {
89          Stack<String> stack = new Stack<>();
90          generateExpression(node, stack);
91      }
92
93      public void generateAssignment(String variable, Node expressi
94          generateExpression(expressionRoot, new Stack<>());
95          String finalResult = "T" + (tempCounter - 1);
96
97          // Cuádruplo para la asignación
98          quadruples.add(new Quadruple("=", finalResult, "-", varia
99
100         // Tríplo para la asignación
101         triples.add(new Triple("=", "(" + (triples.size() - 1) +
102     }
103
104     private void generateExpression(Node node, Stack<String> stac
105         if (node == null) {
106             return;
107         }
108
109         generateExpression(node.left, stack);
110         generateExpression(node.right, stack);
111
112         if (isOperator(node.value.charAt(0))) {
113             if (isUnaryOperator(node.value.charAt(0))) {
114                 String operand = stack.pop();
```



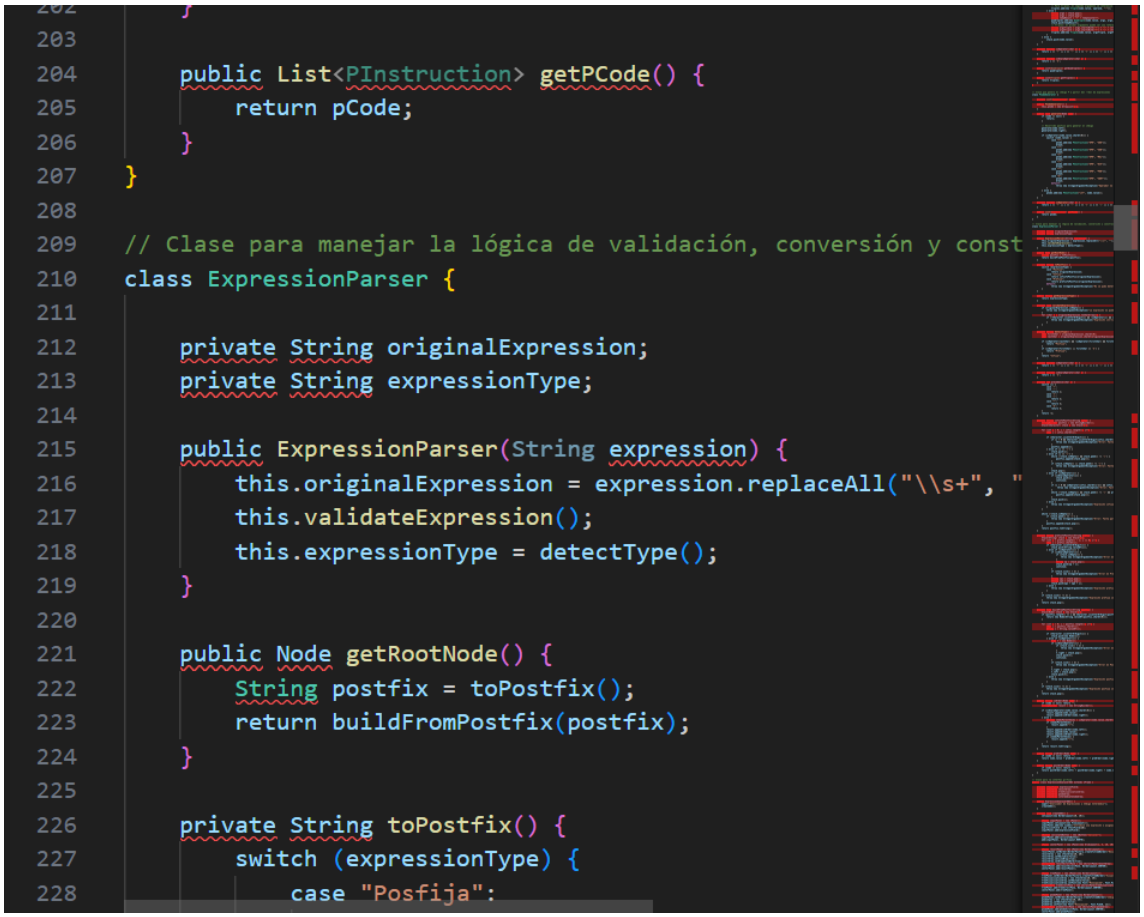
```
112         if (isOperator(node.value.charAt(0))) {
113             if (isUnaryOperator(node.value.charAt(0))) {
114                 String operand = stack.pop();
115                 String tempResult = "T" + tempCounter++;
116                 quadruples.add(new Quadruple(node.value, operand,
117                 stack.push(tempResult);
118                 // Para tríplos, el segundo argumento se referenc
119                 triples.add(new Triple(node.value, operand, "-"))
120             } else {
121                 String arg2 = stack.pop();
122                 String arg1 = stack.pop();
123                 String tempResult = "T" + tempCounter++;
124                 quadruples.add(new Quadruple(node.value, arg1, ar
125                 stack.push(tempResult);
126                 // Tríplos: el primer argumento puede ser una ref
127                 String arg1Triple = arg1.startsWith("T") ? "(" +
128                 String arg2Triple = arg2.startsWith("T") ? "(" +
129                 triples.add(new Triple(node.value, arg1Triple, ar
130             }
131         } else {
132             stack.push(node.value);
133         }
134     }
135
136     private boolean isOperator(char c) {
137         return c == '+' || c == '-' || c == '*' || c == '/' || c
138     }
139
140     private boolean isUnaryOperator(char c) {
141         return c == 'V';
```

```
128                 String arg2Triple = arg2.startsWith("t") ? "(" +
129                 triples.add(new Triple(node.value, arg1Triple, ar
130             }
131         } else {
132             stack.push(node.value);
133         }
134     }
135
136     private boolean isOperator(char c) {
137         return c == '+' || c == '-' || c == '*' || c == '/' || c
138     }
139
140     private boolean isUnaryOperator(char c) {
141         return c == '√';
142     }
143
144     public List<Quadruple> getQuadruples() {
145         return quadruples;
146     }
147
148     public List<Triple> getTriples() {
149         return triples;
150     }
151 }
152
153
```

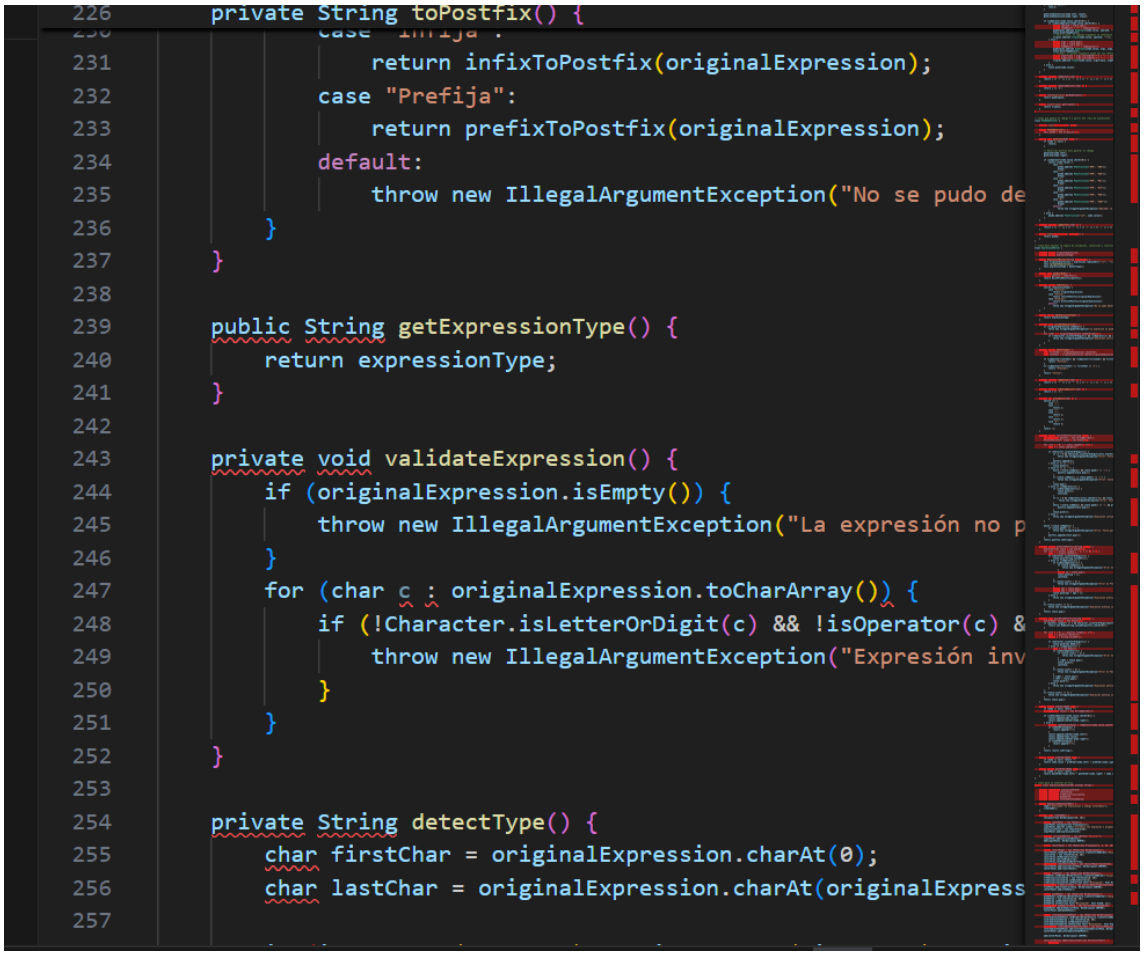


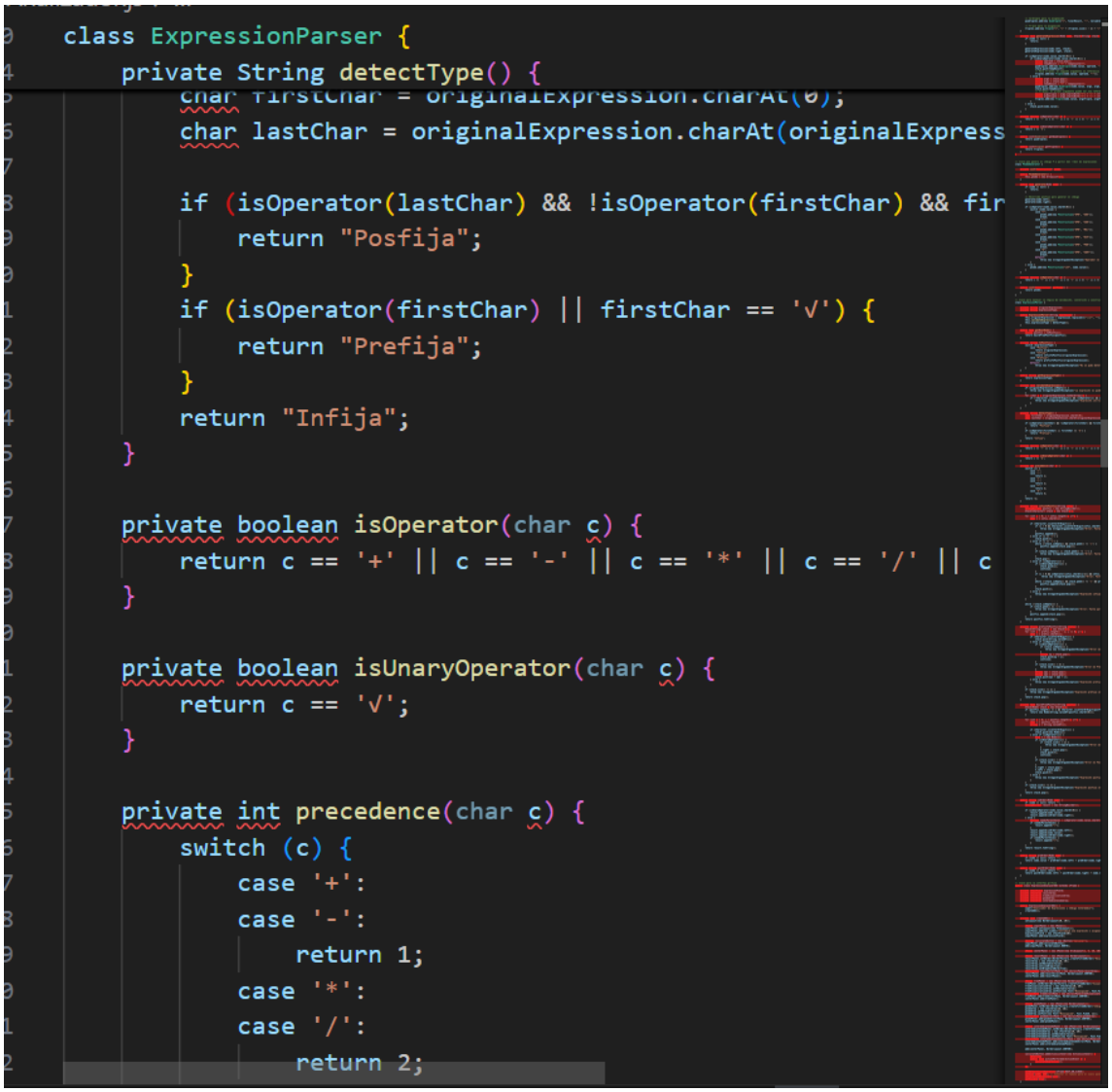
```java
// Clase que genera el código P a partir del árbol de expresiones
class PCodeGenerator {

    private List<PInstruction> pCode;

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

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

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

        if (isOperator(node.value.charAt(0))) {
            switch (node.value) {
                case "+":
                    pCode.add(new PInstruction("OPR", "ADD"));
                    break;
                case "-":
                    pCode.add(new PInstruction("OPR", "SUB"));
                    break;
```


class PCodeGenerator {
public void generate(Node node) {
break;
case "-":
pCode.add(new PInstruction("OPR", "SUB"));
break;
case "*":
pCode.add(new PInstruction("OPR", "MUL"));
break;
case "/":
pCode.add(new PInstruction("OPR", "DIV"));
break;
case "^":
pCode.add(new PInstruction("OPR", "POW"));
break;
case "√":
pCode.add(new PInstruction("OPR", "SQRT"));
break;
default:
throw new IllegalArgumentException("Operador
}
} else {
pCode.add(new PInstruction("LIT", node.value));
}
}
private boolean isOperator(char c) {
return c == '+' || c == '-' || c == '*' || c == '/' || c
}

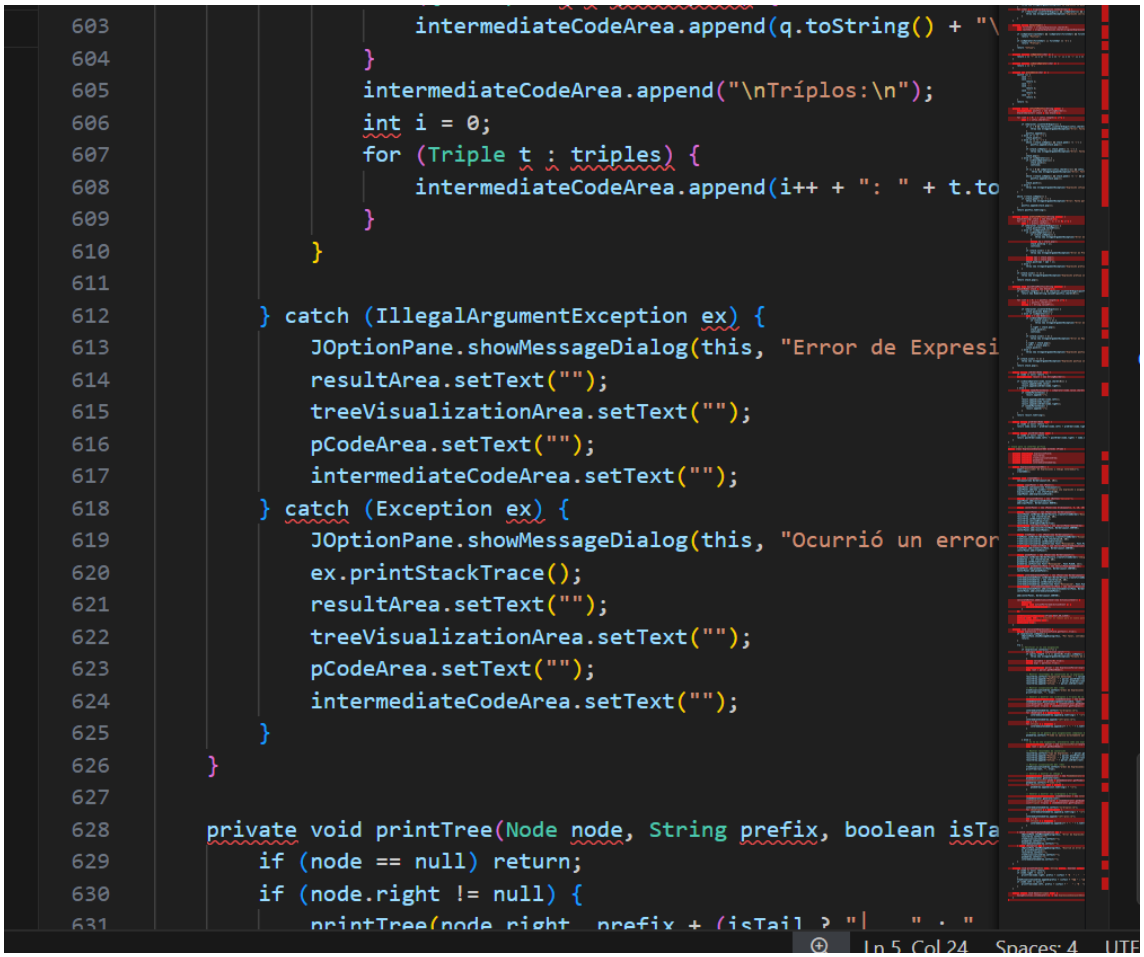


```
    }

    public List<PInstruction> getPCode() {
        return pCode;
    }
}

// Clase para manejar la lógica de validación, conversión y const
class ExpressionParser {

    private String originalExpression;
    private String expressionType;

    public ExpressionParser(String expression) {
        this.originalExpression = expression.replaceAll("\\s+", "
        this.validateExpression();
        this.expressionType = detectType();
    }

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

    private String toPostfix() {
        switch (expressionType) {
            case "Posfija":
```


226 private String toPostfix() {
231 return infixToPostfix(originalExpression);
232 case "Prefija":
233 return prefixToPostfix(originalExpression);
234 default:
235 throw new IllegalArgumentException("No se pudo de
236 }
237 }
238
239 public String getExpressionType() {
240 return expressionType;
241 }
242
243 private void validateExpression() {
244 if (originalExpression.isEmpty()) {
245 throw new IllegalArgumentException("La expresión no p
246 }
247 for (char c : originalExpression.toCharArray()) {
248 if (!Character.isLetterOrDigit(c) && !isOperator(c) &
249 throw new IllegalArgumentException("Expresión inv
250 }
251 }
252 }
253
254 private String detectType() {
255 char firstChar = originalExpression.charAt(0);
256 char lastChar = originalExpression.charAt(originalExpress
257



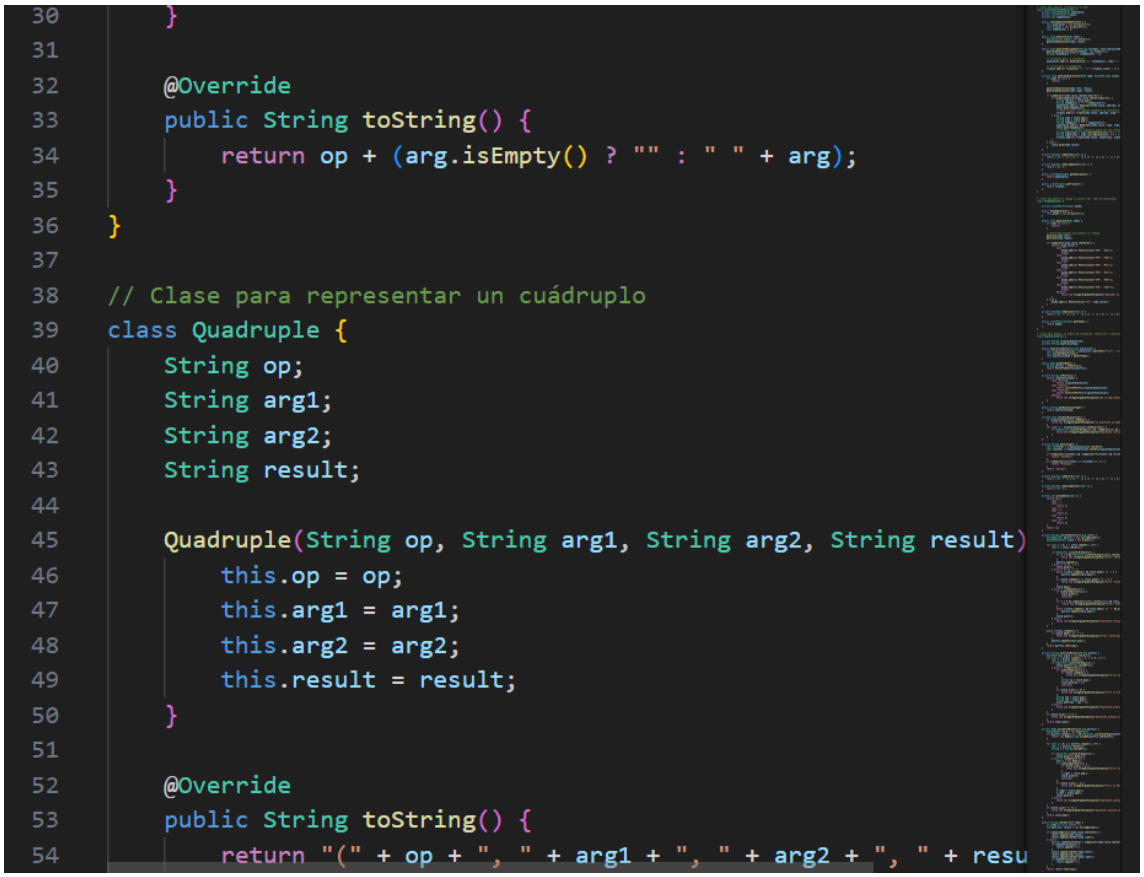
```
class ExpressionParser {
    private String detectType() {
        char firstChar = originalExpression.charAt(0);
        char lastChar = originalExpression.charAt(originalExpress

        if (isOperator(lastChar) && !isOperator(firstChar) && fir
            return "Posfija";
        }
        if (isOperator(firstChar) || firstChar == '√') {
            return "Prefija";
        }
        return "Infija";
    }

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

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

    private int precedence(char c) {
        switch (c) {
            case '+':
            case '-':
                return 1;
            case '*':
            case '/':
                return 2;
```



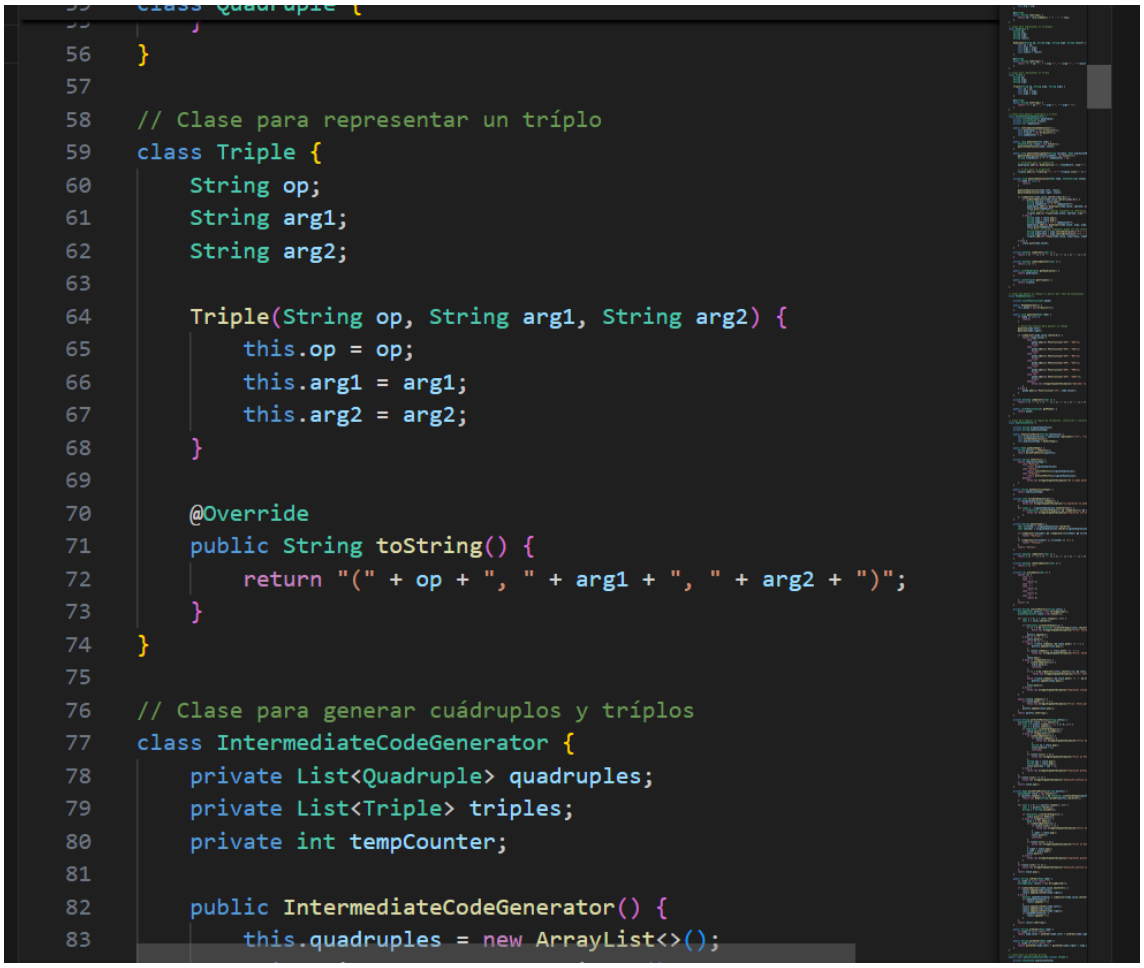
Analizador léxico II

C: > Users > Matia > Downloads > Analizador_2 > ExpressionAnalyzerGUI.java > ...

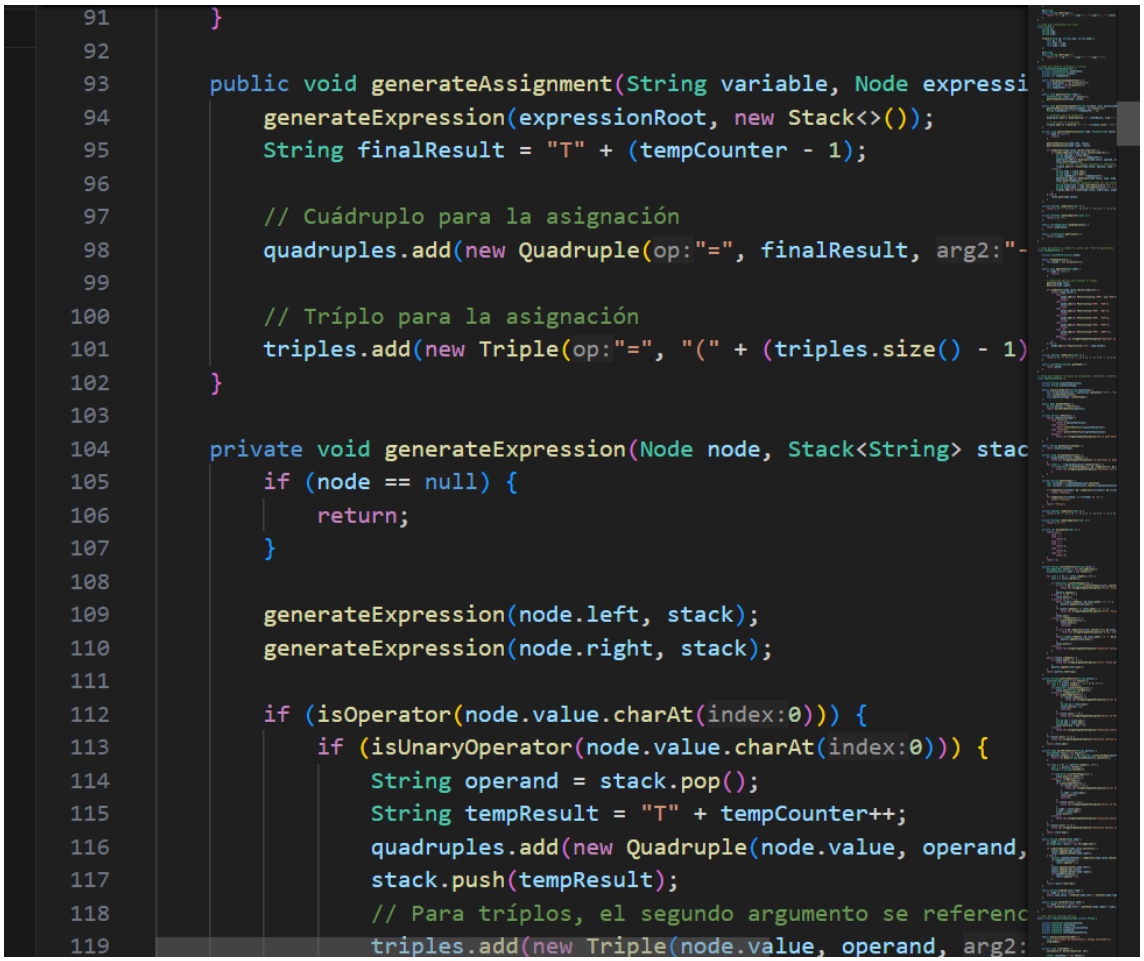
```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Stack;
import java.lang.Math;
import java.util.ArrayList;
import java.util.List;
import java.util.Collections;

// 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 de código P
class PInstruction {
    String op;
    String arg;

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



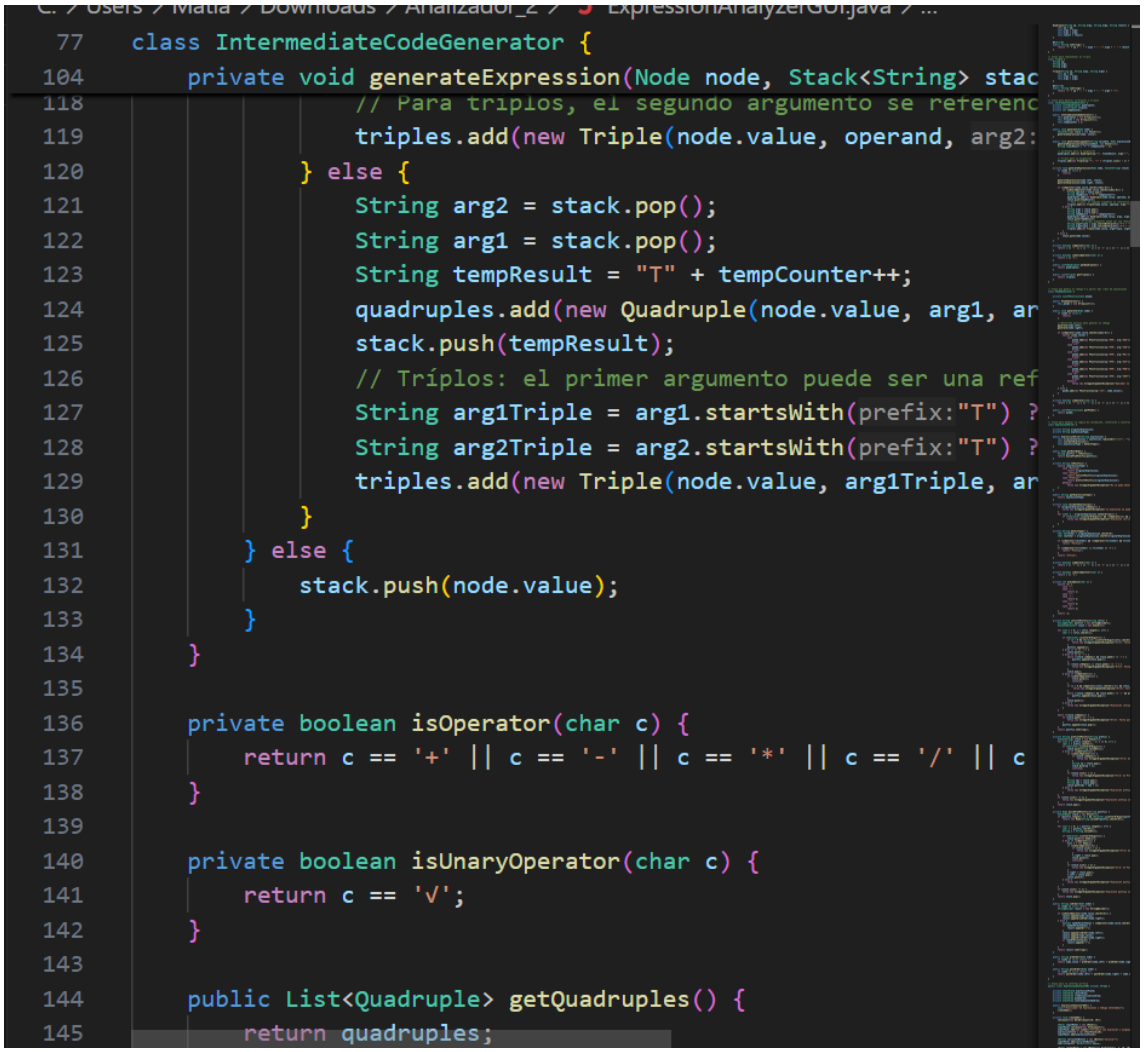
```
30      }
31
32      @Override
33      public String toString() {
34          return op + (arg.isEmpty() ? "" : " " + arg);
35      }
36  }
37
38  // Clase para representar un cuádruplo
39  class Quadruple {
40      String op;
41      String arg1;
42      String arg2;
43      String result;
44
45      Quadruple(String op, String arg1, String arg2, String result)
46          this.op = op;
47          this.arg1 = arg1;
48          this.arg2 = arg2;
49          this.result = result;
50      }
51
52      @Override
53      public String toString() {
54          return "(" + op + ", " + arg1 + ", " + arg2 + ", " + resu
```



```
56  }
57
58  // Clase para representar un tríplo
59  class Triple {
60      String op;
61      String arg1;
62      String arg2;
63
64      Triple(String op, String arg1, String arg2) {
65          this.op = op;
66          this.arg1 = arg1;
67          this.arg2 = arg2;
68      }
69
70      @Override
71      public String toString() {
72          return "(" + op + ", " + arg1 + ", " + arg2 + ")";
73      }
74  }
75
76  // Clase para generar cuádruplos y tríplos
77  class IntermediateCodeGenerator {
78      private List<Quadruple> quadruples;
79      private List<Triple> triples;
80      private int tempCounter;
81
82      public IntermediateCodeGenerator() {
83          this.quadruples = new ArrayList<>();
```



```
91      }
92
93      public void generateAssignment(String variable, Node expressi
94          generateExpression(expressionRoot, new Stack<>());
95          String finalResult = "T" + (tempCounter - 1);
96
97          // Cuádruplo para la asignación
98          quadruples.add(new Quadruple(op:"=", finalResult, arg2:"-
99
100         // Tríplo para la asignación
101         triples.add(new Triple(op:"=", "(" + (triples.size() - 1)
102     }
103
104     private void generateExpression(Node node, Stack<String> stac
105         if (node == null) {
106             return;
107         }
108
109         generateExpression(node.left, stack);
110         generateExpression(node.right, stack);
111
112         if (isOperator(node.value.charAt(index:0))) {
113             if (isUnaryOperator(node.value.charAt(index:0))) {
114                 String operand = stack.pop();
115                 String tempResult = "T" + tempCounter++;
116                 quadruples.add(new Quadruple(node.value, operand,
117                 stack.push(tempResult);
118                 // Para tríplos, el segundo argumento se referenc
119                 triples.add(new Triple(node.value, operand, arg2:
```



```
77    class IntermediateCodeGenerator {
104       private void generateExpression(Node node, Stack<String> stac
118                   // Para tripios, el segundo argumento se referenc
119                   triples.add(new Triple(node.value, operand, arg2:
120               } else {
121                   String arg2 = stack.pop();
122                   String arg1 = stack.pop();
123                   String tempResult = "T" + tempCounter++;
124                   quadruples.add(new Quadruple(node.value, arg1, ar
125                   stack.push(tempResult);
126                   // Tríplos: el primer argumento puede ser una ref
127                   String arg1Triple = arg1.startsWith(prefix:"T") ?
128                   String arg2Triple = arg2.startsWith(prefix:"T") ?
129                   triples.add(new Triple(node.value, arg1Triple, ar
130               }
131           } else {
132               stack.push(node.value);
133           }
134       }
135
136       private boolean isOperator(char c) {
137           return c == '+' || c == '-' || c == '*' || c == '/' || c
138       }
139
140       private boolean isUnaryOperator(char c) {
141           return c == 'v';
142       }
143
144       public List<Quadruple> getQuadruples() {
145           return quadruples;
```

En este segundo analizador lo que hicimos fue mejorar la parte de las expresiones y operaciones validas en la anterior tenía algunos errores en la parte de las operaciones y fue esa parte la que mejoramos en este analizador léxico también le agregamos mas funciones y operaciones a modo que el analizador sea mas eficiente a la hora de su ejecución.