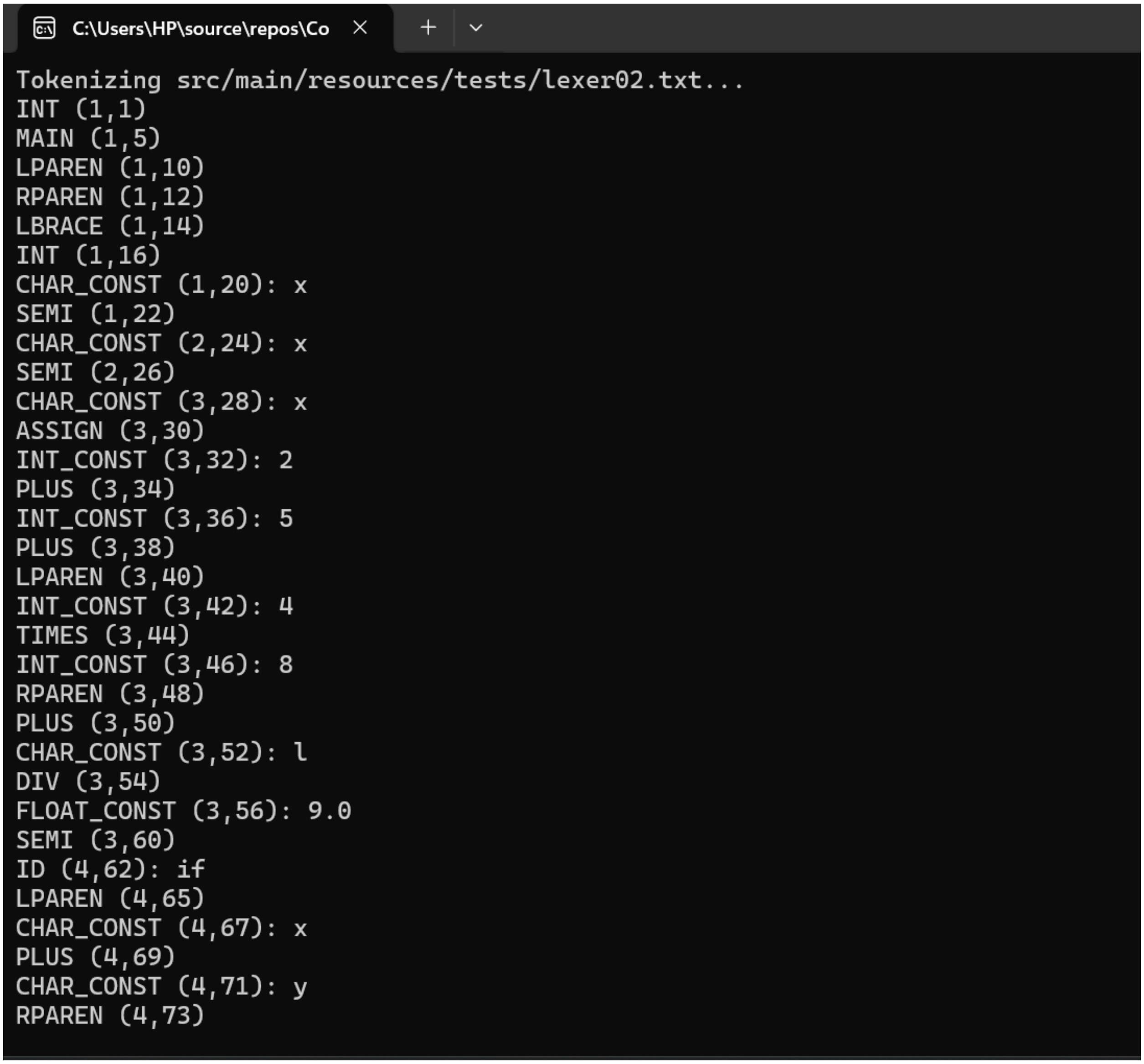
```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

namespace SemanticAnalyzerLab
{
    class Program
    {
        static List<List<string>> Symboltable = new List<List<string>>();
        static List<string> finalArray = new List<string>();
        static List<double> Constants = new List<double>();
        static Regex variable_Reg = new Regex(@"^[A-Za-z_][A-Za-z0-9]*$");
        static bool if_deleted = false;

        static void Main(string[] args)
        {
            InitializeSymbolTable();
            InitializeFinalArray();
            PrintLexerOutput();

            for (int i = 0; i < finalArray.Count; i++)
            {
                Semantic_Analysis(i);
            }

            Console.WriteLine("\nSemantic Analysis Completed.");
    Console.ReadLine();
}

static void InitializeSymbolTable()
{
    Symboltable.Add(new List<string> { "x", "id", "int", "0" });
    Symboltable.Add(new List<string> { "y", "id", "int", "0" });
    Symboltable.Add(new List<string> { "i", "id", "int", "0" });
    Symboltable.Add(new List<string> { "l", "id", "char", "0" });
}

static void InitializeFinalArray()
{
    finalArray.AddRange(new string[] {
        "int", "main", "(", ")", "{",
        "int", "x", ";",
        "x", ";",
        "x", "=", "2", "+", "5", "+", "(", "4", "*", "8", ")", "+", "l", "/", "9.0", ";",
        "if", "(", "x", "+", "y", ")", "{",
        "if", "(", "x", "!=", "4", ")", "{",
        "x", "=", "6", ";",
        "y", "=", "10", ";",
        "i", "=", "11", ";",
        "}", "}",
        "}" // <- no else provided
    });
```

```
}


static void PrintLexerOutput()
{
    Console.WriteLine("Tokenizing src/main/resources/tests/lexer02.txt...");
    int row = 1, col = 1;
    foreach (string token in finalArray)
    {
        if (token == "int")
            Console.WriteLine($"INT ({row},{col})");
        else if (token == "main")
            Console.WriteLine($"MAIN ({row},{col})");
        else if (token == "(")
            Console.WriteLine($"LPAREN ({row},{col})");
        else if (token == ")")
            Console.WriteLine($"RPAREN ({row},{col})");
        else if (token == "{")
            Console.WriteLine($"LBRACE ({row},{col})");
        else if (token == "}")
            Console.WriteLine($"RBRACE ({row},{col})");
        else if (token == ";")
            Console.WriteLine($"SEMI ({row},{col})");
        else if (token == "=")
            Console.WriteLine($"ASSIGN ({row},{col})");
        else if (token == "+")
            Console.WriteLine($"PLUS ({row},{col})");
        else if (token == "-")
            Console.WriteLine($"MINUS ({row},{col})");
        else if (token == "*")
            Console.WriteLine($"TIMES ({row},{col})");
        else if (token == "/")
            Console.WriteLine($"DIV ({row},{col})");
        else if (token == "!=")
            Console.WriteLine($"NEQ ({row},{col})");
        else if (Regex.IsMatch(token, @"^[0-9]+$"))
            Console.WriteLine($"INT_CONST ({row},{col}): {token}");
        else if (Regex.IsMatch(token, @"^[0-9]+\.[0-9]+$"))
            Console.WriteLine($"FLOAT_CONST ({row},{col}): {token}");
        else if (Regex.IsMatch(token, @"^[a-zA-Z]$"))
            Console.WriteLine($"CHAR_CONST ({row},{col}): {token}");
        else if (variable_Reg.Match(token).Success)
            Console.WriteLine($"ID ({row},{col}): {token}");
        else
            Console.WriteLine($"UNKNOWN ({row},{col}): {token}");

        col += token.Length + 1;
        if (token == ";") row++;
    }
    Console.WriteLine("EOF ({0},{1})", row, col);
}

static void Semantic_Analysis(int k)
```

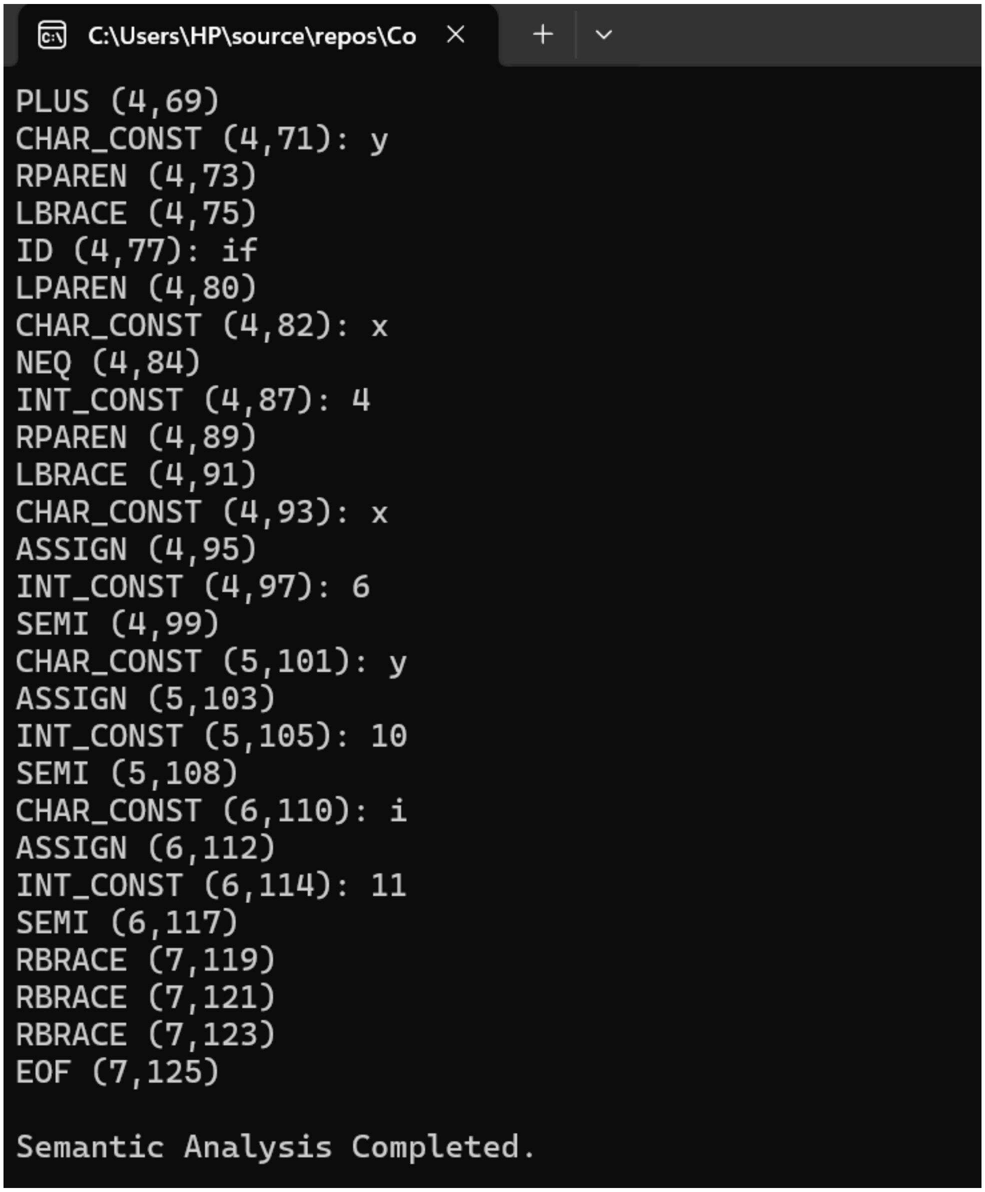
```
{
    if (k >= finalArray.Count) return;

    // Arithmetic analysis
    if ((finalArray[k] == "+" || finalArray[k] == "-") && k > 1 && k < finalArray.Count - 1)
    {
        string before = finalArray[k - 1];
        string after = finalArray[k + 1];

        if (variable_Reg.IsMatch(before) && variable_Reg.IsMatch(after))
        {
            int before_i = FindSymbol(before);
            int after_i = FindSymbol(after);

            if (before_i != -1 && after_i != -1)
            {
                string op = finalArray[k];
                double val1 = double.Parse(Symboltable[before_i][3]);
                double val2 = double.Parse(Symboltable[after_i][3]);
                double result = (op == "+") ? val1 + val2 : val1 - val2;

                Constants.Add(result);
            }
        }
    }
```

```
// Comparison
if (finalArray[k] == ">")
{
    if (k > 0 && k < finalArray.Count - 1)
    {
        string before = finalArray[k - 1];
        string after = finalArray[k + 1];
        int before_i = FindSymbol(before);
        int after_i = FindSymbol(after);

        if (before_i != -1 && after_i != -1)
        {
            double left = double.Parse(Symboltable[before_i][3]);
            double right = double.Parse(Symboltable[after_i][3]);

            if (left > right)
            {
                RemoveElseBlock();
            }
            else
            {
                RemoveIfBlock();
                if_deleted = true;
            }
        }
    }
    }
}

static int FindSymbol(string name)
{
    for (int i = 0; i < Symboltable.Count; i++)
    {
        if (Symboltable[i][0] == name)
            return i;
    }
    return -1;
}

static void RemoveElseBlock()
{
    int start = finalArray.IndexOf("else");
    if (start == -1) return;

    int braceCount = 0;
    int end = -1;
    for (int i = start; i < finalArray.Count; i++)
    {
        if (finalArray[i] == "{") braceCount++;
        if (finalArray[i] == "}") braceCount--;
        if (braceCount == 0 && finalArray[i] == "}") { end = i; break; }
    }
    if (end != -1)
        finalArray.RemoveRange(start, end - start + 1);
}

static void RemoveIfBlock()
{
    int start = -1;
    for (int i = 0; i < finalArray.Count; i++)
    {
        if (finalArray[i] == "if") { start = i; break; }
    }

    if (start == -1) return;

    int braceCount = 0, end = -1;
    for (int i = start; i < finalArray.Count; i++)
    {
        if (finalArray[i] == "{") braceCount++;
        if (finalArray[i] == "}") braceCount--;
        if (braceCount == 0 && finalArray[i] == "}") { end = i; break; }
    }

    if (end != -1)
        finalArray.RemoveRange(start, end - start + 1);
}
```

```
  }

}
```



```
Tokenizing src/main/resources/tests/lexer02.txt...
INT (1,1)
MAIN (1,5)
LPAREN (1,10)
RPAREN (1,12)
LBRACE (1,14)
INT (1,16)
CHAR_CONST (1,20): x
SEMI (1,22)
CHAR_CONST (2,24): x
SEMI (2,26)
CHAR_CONST (3,28): x
ASSIGN (3,30)
INT_CONST (3,32): 2
PLUS (3,34)
INT_CONST (3,36): 5
PLUS (3,38)
LPAREN (3,40)
INT_CONST (3,42): 4
TIMES (3,44)
INT_CONST (3,46): 8
RPAREN (3,48)
PLUS (3,50)
CHAR_CONST (3,52): l
DIV (3,54)
FLOAT_CONST (3,56): 9.0
SEMI (3,60)
ID (4,62): if
LPAREN (4,65)
CHAR_CONST (4,67): x
PLUS (4,69)
CHAR_CONST (4,71): y
RPAREN (4,73)
```



```
PLUS (4,69)
CHAR_CONST (4,71): y
RPAREN (4,73)
LBRACE (4,75)
ID (4,77): if
LPAREN (4,80)
CHAR_CONST (4,82): x
NEQ (4,84)
INT_CONST (4,87): 4
RPAREN (4,89)
LBRACE (4,91)
CHAR_CONST (4,93): x
ASSIGN (4,95)
INT_CONST (4,97): 6
SEMI (4,99)
CHAR_CONST (5,101): y
ASSIGN (5,103)
INT_CONST (5,105): 10
SEMI (5,108)
CHAR_CONST (6,110): i
ASSIGN (6,112)
INT_CONST (6,114): 11
SEMI (6,117)
RBRACE (7,119)
RBRACE (7,121)
RBRACE (7,123)
EOF (7,125)

Semantic Analysis Completed.
```