COMSATS University Islamabad

Attock Campus

Lab 1 to 12

Submitted By:

Muhammad Anas

Registration No:

Sp22-Bcs-042

Submitted To:

Sir Bilal Haider

Subject:

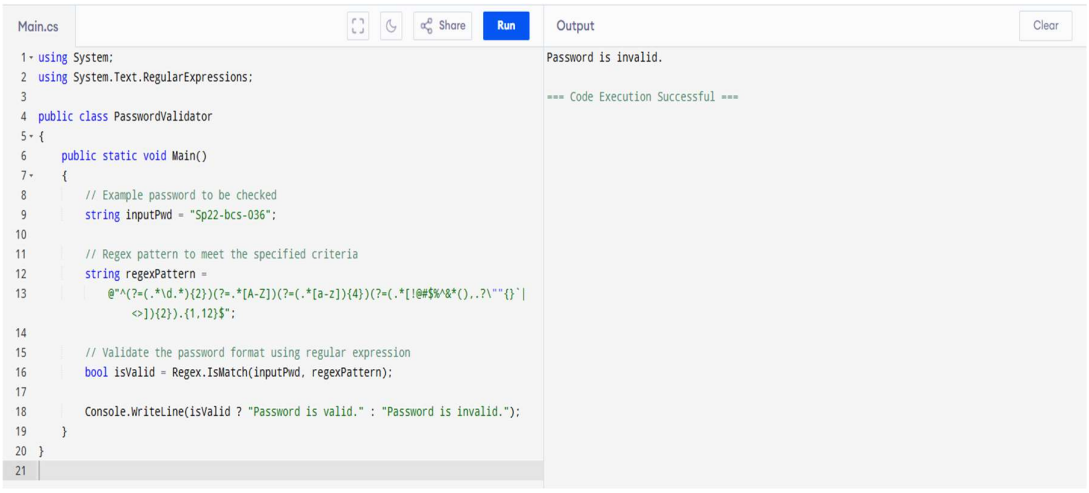
Compiler Construction

Date: 25th May,2025

Lab 1:

Task 1:

```
using System;
using System.Text.RegularExpressions;

public class PasswordValidator
{
   public static void Main()
   {
      // Example password to be checked
      string inputPwd = "Sp22-bcs-036";

      // Regex pattern to meet the specified criteria
      string regexPattern =
         @"^(?=(.*\d.*){2})(?=.*[A-Z])(?=(.*[a-
z]){4})(?=(.*[!@#$%^&*(),.?\""{}`|<>]){2}).{1,12}$";

      // Validate the password format using regular expression
      bool isValid = Regex.IsMatch(inputPwd, regexPattern);

      Console.WriteLine(isValid ? "Password is valid." :
"Password is invalid.");
   }
}
```



Task 2:

```
using System;
using System.Text;
using System.Text.RegularExpressions;

class PasswordGeneratorApp
{
   static void Main(string[] args)
   {
      // Gather user inputs
      Console.Write("First Name: ");
      string fName = Console.ReadLine();

      Console.Write("Last Name: ");
      string lName = Console.ReadLine();


      Console.Write("Registration #: ");
      string regID = Console.ReadLine();

      Console.Write("Favorite Food: ");
      string favFood = Console.ReadLine();

      Console.Write("Favorite Game: ");
      string favGame = Console.ReadLine();

      // Call function to create password
      string finalPassword = CreateSecurePassword(fName,
lName, regID, favFood, favGame);

      Console.WriteLine("Generated Password: " +
finalPassword);
   }

   static string CreateSecurePassword(string first, string last,
string reg, string food, string game)
   {
      // Merge inputs
```

```
        string combinedInput = string.Concat(first, last, reg, food,
game);

        // Remove non-alphanumeric characters
        string cleaned = Regex.Replace(combinedInput, "[^a-zA-
Z0-9]", "");

        // Add complexity
        string specialSymbols = "!@#$%^&*()_+[]{}|;:,.<>?/~`";
        StringBuilder passwordBuilder = new
StringBuilder(cleaned);
        Random rng = new Random();

        for (int i = 0; i < 4; i++)

        {
            passwordBuilder.Append(rng.Next(0, 10)); // random
digit


passwordBuilder.Append(specialSymbols[rng.Next(specialSym
bols.Length)]); // random symbol
        }

        // Ensure minimum length
        while (passwordBuilder.Length < 12)
        {
            passwordBuilder.Insert(0, 'Z');
        }

        // Shuffle final string
        string rawPassword = passwordBuilder.ToString();
        StringBuilder shuffled = new StringBuilder();
        while (rawPassword.Length > 0)
        {
            int idx = rng.Next(rawPassword.Length);
            shuffled.Append(rawPassword[idx]);
            rawPassword = rawPassword.Remove(idx, 1);
        }
```

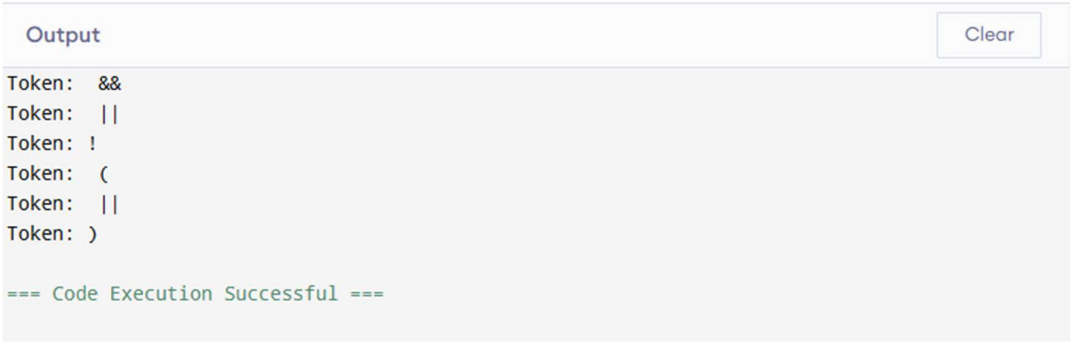
```
        return shuffled.ToString();
    }
}
```

Lab 2:

Task 1:

```
using System;
using System.Text.RegularExpressions;

class LogicalTokenParser
{
    static void Main()

    {
        // Regex pattern to identify logical operators and
parentheses
        string logicPattern = @"\s*(\&\&|\|\||!|\(|\))\s*";

        // Input string to test against
        string expression = "x && y || !z (x || y)";

        // Compile regex
        var logicRegex = new Regex(logicPattern);

        // Extract matching tokens
        var tokenMatches = logicRegex.Matches(expression);

        // Display each token found
        foreach (var match in tokenMatches)
        {
```

```
            Console.WriteLine($"Token: {(match as
Match).Value}");
        }
    }

}
```



Task 2:

```
using System;
using System.Text.RegularExpressions;

class RelationalOperatorExtractor
{
    static void Main()
    {
        // Define regex to capture relational operators
        string operatorPattern = @"\s*(==|!=|>=|<=|\>|\<)\s*";

        // Sample code string containing relational expressions
        string expressionText = "a == b && c != d || e >= f && g <
h";


        // Initialize Regex with the pattern
        var opRegex = new Regex(operatorPattern);

        // Retrieve all matched relational operators
        var foundOperators = opRegex.Matches(expressionText);

        // Display results
        foreach (Match op in foundOperators)
```

```
        {
            Console.WriteLine($"Operator detected:
'{op.Value.Trim()}'");
        }
    }
}
```

```
Output                                                    Clear

Operator detected: '=='
Operator detected: '!='
Operator detected: '>='
Operator detected: '<'

=== Code Execution Successful ===
```

Lab 3:

```
using System;
using System.Text.RegularExpressions;

class FloatValidator
{
    static void Main()
    {
        // Pattern matches numbers with optional sign, up to 3
digits before/after decimal
        string floatPattern = @"^[\+\-]?\d{1,3}(\.\d{1,3})?$|^[\+\-
]?\.\d{1,3}$";

        // Array of test inputs
        string[] samples = {
            "123",       // valid
            "-12.34",    // valid
            "+0.567",    // valid
            ".678",      // valid
            "0.5",       // valid
            "123456",    // invalid
            "1.2345",    // invalid
            "+1234",     // invalid

            ".1234"      // invalid
        };
```

```
        // Evaluate each sample
        foreach (var item in samples)
        {
            var valid = Regex.IsMatch(item, floatPattern);
            Console.WriteLine($"Input '{item}' → {(valid ? "Valid"
: "Invalid")}");
        }
    }
}
```

Output Clear

```
Input '123' ? Valid
Input '-12.34' ? Valid
Input '+0.567' ? Valid
Input '.678' ? Valid
Input '0.5' ? Valid
Input '123456' ? Invalid
Input '1.2345' ? Invalid
Input '+1234' ? Invalid
Input '.1234' ? Invalid

=== Code Execution Successful ===
```

Lab 4:

```
using System;
using System.Text.RegularExpressions;

class Program
{
    static string[] keywords = { "int", "if", "else" };

    static void Main()
    {
        Console.WriteLine("Enter some code:");
        string input = Console.ReadLine();

        Tokenize(input);
    }

    static void Tokenize(string code)
    {
        // Matches words, numbers, and symbols
```

```
        string pattern = @"\w+|[^\s\w]";
        MatchCollection tokens = Regex.Matches(code, pattern);

        foreach (Match token in tokens)
        {
            string value = token.Value;

            if (IsKeyword(value))
                Console.WriteLine($"[Keyword]     {value}");
            else if (Regex.IsMatch(value, @"^\d+$"))
                Console.WriteLine($"[Number]      {value}");
            else if (Regex.IsMatch(value, @"^[a-zA-Z_]\w*$"))
                Console.WriteLine($"[Identifier]  {value}");
            else
                Console.WriteLine($"[Operator]    {value}");
        }
    }

    static bool IsKeyword(string token)
    {
        foreach (string key in keywords)
        {
            if (token == key)
                return true;
        }
        return false;
    }
}
```

Output Clear

```
Enter some code:
int x = 10;
[Keyword]     int
[Identifier]  x
[Operator]    =
[Number]      10
[Operator]    ;

=== Code Execution Successful ===
```

Lab 5:

```
#include <stdio.h>

#include <stdlib.h>
#include <string.h>

#define TABLE_SIZE 10  // Hash table size

// Structure for a symbol table entry
typedef struct Symbol {
    char name[50];       // Identifier name
    char type[20];       // Data type (e.g., int, float)
    int scope;           // Scope level
    struct Symbol *next; // Pointer for chaining (linked list)
} Symbol;

// Hash table (Array of pointers to Symbol nodes)

Symbol *symbolTable[TABLE_SIZE];

// Hash function (Sum of ASCII values modulo table size)
int hashFunction(char *name) {
    int sum = 0;
    for (int i = 0; name[i] != '\0'; i++) {
        sum += name[i];
    }
    return sum % TABLE_SIZE;
}

// Insert a symbol into the table
void insertSymbol(char *name, char *type, int scope) {
    int index = hashFunction(name);

    // Create a new symbol node
    Symbol *newSymbol = (Symbol *)malloc(sizeof(Symbol));
    strcpy(newSymbol->name, name);
```

```
    strcpy(newSymbol->type, type);
    newSymbol->scope = scope;
    newSymbol->next = NULL;

    // Insert at the beginning of the linked list (chaining)
    if (symbolTable[index] == NULL) {
        symbolTable[index] = newSymbol;
    } else {
        newSymbol->next = symbolTable[index];
        symbolTable[index] = newSymbol;
    }
    printf("Inserted: %s (%s, scope: %d)\n", name, type, scope);
}

// Search for a symbol in the table
Symbol* searchSymbol(char *name) {
    int index = hashFunction(name);
    Symbol *temp = symbolTable[index];

    while (temp != NULL) {
        if (strcmp(temp->name, name) == 0) {

            return temp; // Found
        }
        temp = temp->next;
    }
    return NULL; // Not found
}

// Display the symbol table
void displaySymbolTable() {
    printf("\nSymbol Table:\n");
    printf("-------------------------------\n");
    printf("| Index | Name    | Type   | Scope |\n");
    printf("-------------------------------\n");

    for (int i = 0; i < TABLE_SIZE; i++) {
        Symbol *temp = symbolTable[i];
        while (temp != NULL) {
```

```
            printf("| %5d | %-7s | %-6s | %5d |\n", i, temp->name,
temp->type, temp->scope);
            temp = temp->next;
        }
    }
    printf("--------------------------------\n");
}

// Main function for testing
int main() {
    // Initializing the symbol table with NULL
    for (int i = 0; i < TABLE_SIZE; i++) {
        symbolTable[i] = NULL;
    }

    // Insert some symbols
    insertSymbol("x", "int", 1);
    insertSymbol("y", "float", 1);
    insertSymbol("sum", "int", 2);
    insertSymbol("product", "int", 2);
    insertSymbol("y", "char", 3);  // Different scope


    // Search for a symbol
    char searchName[50];
    printf("\nEnter variable name to search: ");
    scanf("%s", searchName);

    Symbol *result = searchSymbol(searchName);
    if (result) {
        printf("Found: %s (%s, scope: %d)\n", result->name,
result->type, result->scope);
    } else {
        printf("Symbol not found.\n");
    }

    // Display the symbol table
    displaySymbolTable();

    return 0;
```

}

Output

```
Inserted: x (int, scope: 1)
Inserted: y (float, scope: 1)
Inserted: sum (int, scope: 2)
Inserted: product (int, scope: 2)
Inserted: y (char, scope: 3)

Enter variable name to search: x+y=c
Symbol not found.

Symbol Table:
--------------------------------
| Index | Name    | Type   | Scope |
--------------------------------
|     0 | x       | int    |     1 |
|     1 | y       | char   |     3 |
|     1 | sum     | int    |     2 |
|     1 | y       | float  |     1 |
|     9 | product | int    |     2 |
--------------------------------
```

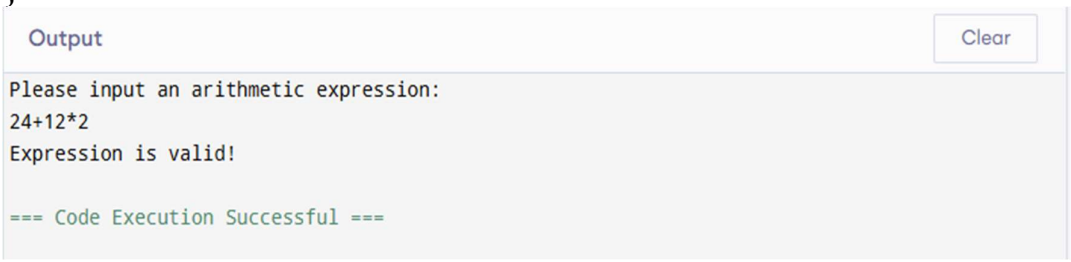
Lab 6:

```
using System;

class RecursiveParser
{
   static string expression;
   static int pos = 0;

   static void Main()
   {
      Console.WriteLine("Please input an arithmetic
expression:");
      expression = Console.ReadLine();
      expression = expression.Replace(" ", ""); // strip spaces

      try
      {
         ParseExpression();
         if (pos == expression.Length)

         {
            Console.WriteLine("Expression is valid!");
         }
```

```
        else
        {
            Console.WriteLine("Expression is invalid!");
        }
    }
    catch
    {
        Console.WriteLine("Expression is invalid!");
    }
}

// E -> T E_
static void ParseExpression()
{
    ParseTerm();
    ParseExpressionPrime();
}

// E' -> + T E' | ε
static void ParseExpressionPrime()
{
    if (Consume('+'))
    {
        ParseTerm();
        ParseExpressionPrime();
    }
}

// T -> F T_
static void ParseTerm()
{
    ParseFactor();
    ParseTermPrime();
}

// T' -> * F T' | ε

static void ParseTermPrime()
{
```

```
        if (Consume('*'))
        {
            ParseFactor();
            ParseTermPrime();
        }
    }

    // F -> (E) | number
    static void ParseFactor()
    {
        if (Consume('('))
        {
            ParseExpression();
            if (!Consume(')'))
                throw new Exception("Expected closing
parenthesis");
        }
        else if (char.IsDigit(Current()))
        {
            while (char.IsDigit(Current()))
                pos++;
        }
        else
        {
            throw new Exception("Unexpected character in factor");
        }
    }

    static bool Consume(char ch)
    {
        if (pos < expression.Length && expression[pos] == ch)
        {
            pos++;
            return true;
        }
        return false;
    }

    static char Current()
```

```
    {
        return pos < expression.Length ? expression[pos] : '\0';
    }
}
```

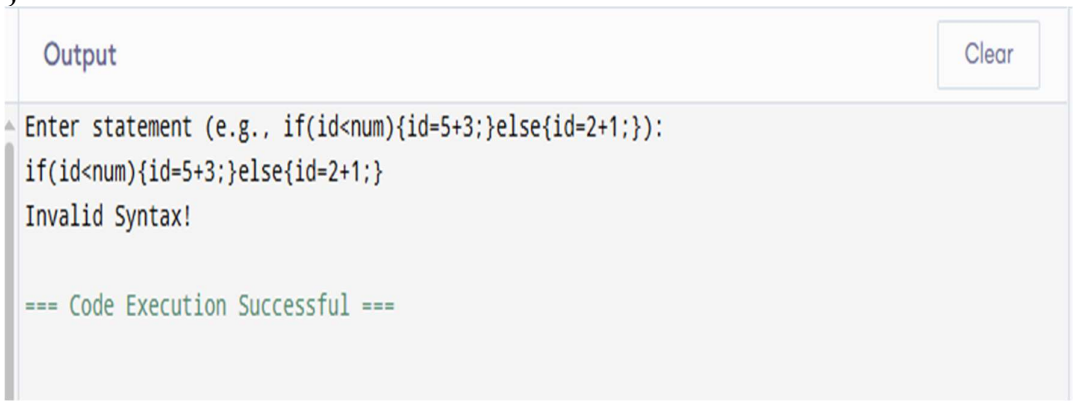


Lab 7:

```
using System;

class GrammarParser
{
    static string input;
    static int position = 0;

    static void Main()
    {
        Console.WriteLine("Enter statement (e.g.,
if(id<num){id=5+3;}else{id=2+1;}):");
        input = Console.ReadLine();
        input = input.Replace(" ", ""); // Remove spaces for easier
parsing

        try
        {
            ParseStatement();

            if (position == input.Length)
                Console.WriteLine("Valid Syntax!");
            else
                Console.WriteLine("Invalid Syntax!");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Invalid Syntax!");
```

```
    }
}

// S → if(C){S}else{S} | id=E;

static void ParseStatement()
{
    if (Match("if"))
    {
        Match("(");
        ParseCondition();
        Match(")");
        Match("{");
        ParseStatement();
        Match("}");
        Match("else");
        Match("{");
        ParseStatement();
        Match("}");
    }
    else if (Match("id"))
    {
        Match("=");
        ParseExpression();
        Match(";");
    }
    else
    {
        throw new Exception("Invalid statement");
    }
}

// C → id<num
static void ParseCondition()
{
    Match("id");
    Match("<");
    Match("num");
}
```

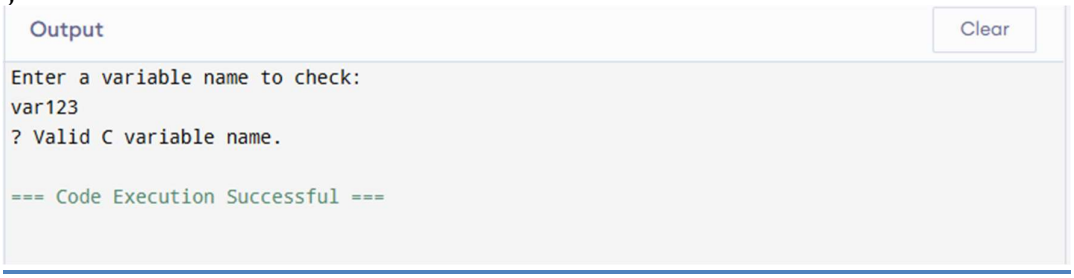
```
    // E → T + T
    static void ParseExpression()
    {
        ParseTerm();

        Match("+");
        ParseTerm();
    }

    // T → id | num
    static void ParseTerm()
    {
        if (!(Match("id") || Match("num")))
            throw new Exception("Expected id or num");
    }

    // Helper: matches the next token if it exists
    static bool Match(string token)
    {
        if (input.Substring(position).StartsWith(token))
        {
            position += token.Length;
            return true;
        }
        return false;
    }
}
```



Lab 8:

```
using System;

class DFA_CVariable
{
   static void Main()
   {
      Console.WriteLine("Enter a variable name to check:");
      string input = Console.ReadLine();

      if (IsValidVariable(input))
         Console.WriteLine("✅ Valid C variable name.");
      else
         Console.WriteLine("❌ Invalid C variable name.");
   }

   static bool IsValidVariable(string input)
   {
      int state = 0;

      for (int i = 0; i < input.Length; i++)
      {
         char ch = input[i];

         switch (state)
         {
            case 0:
               if (char.IsLetter(ch) || ch == '_')
                  state = 1;
               else
                  return false;
               break;

            case 1:
               if (char.IsLetterOrDigit(ch) || ch == '_')
                  state = 1;
               else
```

```
                return false;
            break;
        }
    }

    return state == 1;
  }
}
```



Lab 10:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace SLRParserDemo
{
  public enum TokenType { id, plus, star, lparen, rparen, dollar
}

  public class Production
  {
    public string LHS { get; }
    public string[] RHS { get; }

    public Production(string lhs, params string[] rhs)
    {
      LHS = lhs;
      RHS = rhs;
    }

    public override string ToString() => $"{LHS} →
{string.Join(" ", RHS)} }
```

```
public class SLRParser
{
    private readonly List<Production> productions = new()
    {
        new Production("E'", "E"),          // 0: augmented start
production
        new Production("E", "E", "+", "T"), // 1
        new Production("E", "T"),           // 2
        new Production("T", "T", "*", "F"), // 3
        new Production("T", "F"),           // 4
        new Production("F", "(", "E", ")"), // 5
        new Production("F", "id")           // 6
    };

    // Action table: rows = states, cols = tokens (id,+,*,(,),$)
    private readonly string[,] action = new string[12, 6]
    {
        { "S5", "",   "",   "S4", "",   ""   }, // 0
        { "",   "S6", "",   "",   "",   "acc"}, // 1
        { "",   "R2","R2",  "",   "R2","R2"},  // 2
        { "",   "R4","R4",  "",   "R4","R4"},  // 3
        { "S5", "",   "",   "S4", "",   ""   }, // 4
        { "",   "R6","R6",  "",   "R6","R6"},  // 5

        { "S5", "",   "",   "S4", "",   ""   }, // 6
        { "S5", "",   "",   "S4", "",   ""   }, // 7
        { "",   "S6", "",   "",   "S11",""   }, // 8
        { "",   "R1","S7",  "",   "R1","R1"},  // 9
        { "",   "R3","R3",  "",   "R3","R3"},  // 10
        { "",   "R5","R5",  "",   "R5","R5"}   // 11
    };

    // Goto table: rows = states, cols = non-terminals (E,T,F)
    private readonly int[,] gotoTable = new int[12, 3]
    {
        { 1,  2,  3  },  // 0
        { -1, -1, -1 },  // 1
        { -1, -1, -1 },  // 2
        { -1, -1, -1 },  // 3
```

```
            { 8,  2,  3  },  // 4
            { -1, -1, -1 },  // 5
            { -1, 9,  3  },  // 6
            { -1, -1, 10 },  // 7
            { -1, -1, -1 },  // 8
            { -1, -1, -1 },  // 9
            { -1, -1, -1 },  // 10
            { -1, -1, -1 }   // 11
        };

        private readonly Dictionary<string, int> symbolToCol =
new()
        {
            { "id", 0 }, { "+", 1 }, { "*", 2 }, { "(", 3 }, { ")", 4 }, {
"$", 5 }
        };

        private readonly Dictionary<string, int> gotoToCol =
new()
        {
            { "E", 0 }, { "T", 1 }, { "F", 2 }
        };

        // Tokenizes input string into tokens separated by spaces

        public List<string> Tokenize(string input)
        {
            var tokens = new List<string>();
            var parts = input.Split(' ',
StringSplitOptions.RemoveEmptyEntries);

            foreach (var part in parts)
            {
                if (symbolToCol.ContainsKey(part))
                    tokens.Add(part);
                else
                    throw new Exception($"Unknown token: {part}");
            }
            tokens.Add("$"); // end marker
```

```
        return tokens;
    }

    public void Parse(string input)
    {
        var tokens = Tokenize(input);
        var stateStack = new Stack<int>();
        var symbolStack = new Stack<string>();

        stateStack.Push(0);
        int ip = 0;

        Console.WriteLine("{0,-20}{1,-30}{2,-30}{3}", "State
Stack", "Symbol Stack", "Input", "Action");
        Console.WriteLine(new string('-', 100));

        while (true)
        {
            string currToken = tokens[ip];
            int state = stateStack.Peek();
            string act = action[state, symbolToCol[currToken]];

            Console.WriteLine("{0,-20}{1,-30}{2,-30}{3}",
                string.Join(" ", stateStack.Reverse()),
                string.Join(" ", symbolStack.Reverse()),

                string.Join(" ", tokens.Skip(ip)),
                string.IsNullOrEmpty(act) ? "error" : act);

            if (string.IsNullOrEmpty(act))
            {
                Console.WriteLine("Parsing error!");
                return;
            }
            else if (act.StartsWith("S"))
            {
                int nextState = int.Parse(act[1..]);
                symbolStack.Push(currToken);
                stateStack.Push(nextState);
```

```
                ip++;
            }
            else if (act.StartsWith("R"))
            {
                int prodNum = int.Parse(act[1..]);
                var prod = productions[prodNum];

                for (int i = 0; i < prod.RHS.Length; i++)
                {
                    symbolStack.Pop();
                    stateStack.Pop();
                }

                symbolStack.Push(prod.LHS);
                int gotoState = gotoTable[stateStack.Peek(),
gotoToCol[prod.LHS]];
                stateStack.Push(gotoState);
            }
            else if (act == "acc")
            {
                Console.WriteLine("{0,-20}{1,-30}{2,-30}{3}",
                    string.Join(" ", stateStack.Reverse()),
                    string.Join(" ", symbolStack.Reverse()),
                    string.Join(" ", tokens.Skip(ip)),
                    "accept");
                Console.WriteLine("\nInput accepted!");
                return;
            }
        }
    }
}

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Enter input (tokens separated by
spaces, e.g., id + id * id):");
        string input = Console.ReadLine();
```

```
            var parser = new SLRParser();

            try
            {
                parser.Parse(input);
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error: {ex.Message}");
            }
        }
    }
}
```

Output Clear

```
Enter input (tokens separated by spaces, e.g., id + id * id):
id + id * id
State Stack         Symbol Stack                Input                       Action
-------------------------------------------------------------------------------------
    --------
0                                               id + id * id $              S5
0 5                 id                          + id * id $                 R6
0 3                 F                           + id * id $                 R4
0 2                 T                           + id * id $                 R2
0 1                 E                           + id * id $                 S6
0 1 6               E +                         id * id $                   S5
0 1 6 5             E + id                      * id $                      R6
0 1 6 3             E + F                       * id $                      R4
0 1 6 9             E + T                       * id $                      S7
0 1 6 9 7           E + T *                     id $                        S5
0 1 6 9 7 5         E + T * id                  $                           R6
0 1 6 9 7 10        E + T * F                   $                           R3
0 1 6 9             E + T                       $                           R1
0 1                 E                           $                           acc
0 1                 E                           $                           accept

Input accepted!

=== Code Execution Successful ===
```

Lab 11:

Task 1:

```
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<int> Constants = new List<int>();
      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", ";",
            "}", "}",
            "}"
         });
      }

      static void PrintLexerOutput()
      {
         Console.WriteLine("Tokenizing input...");

         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})");
```

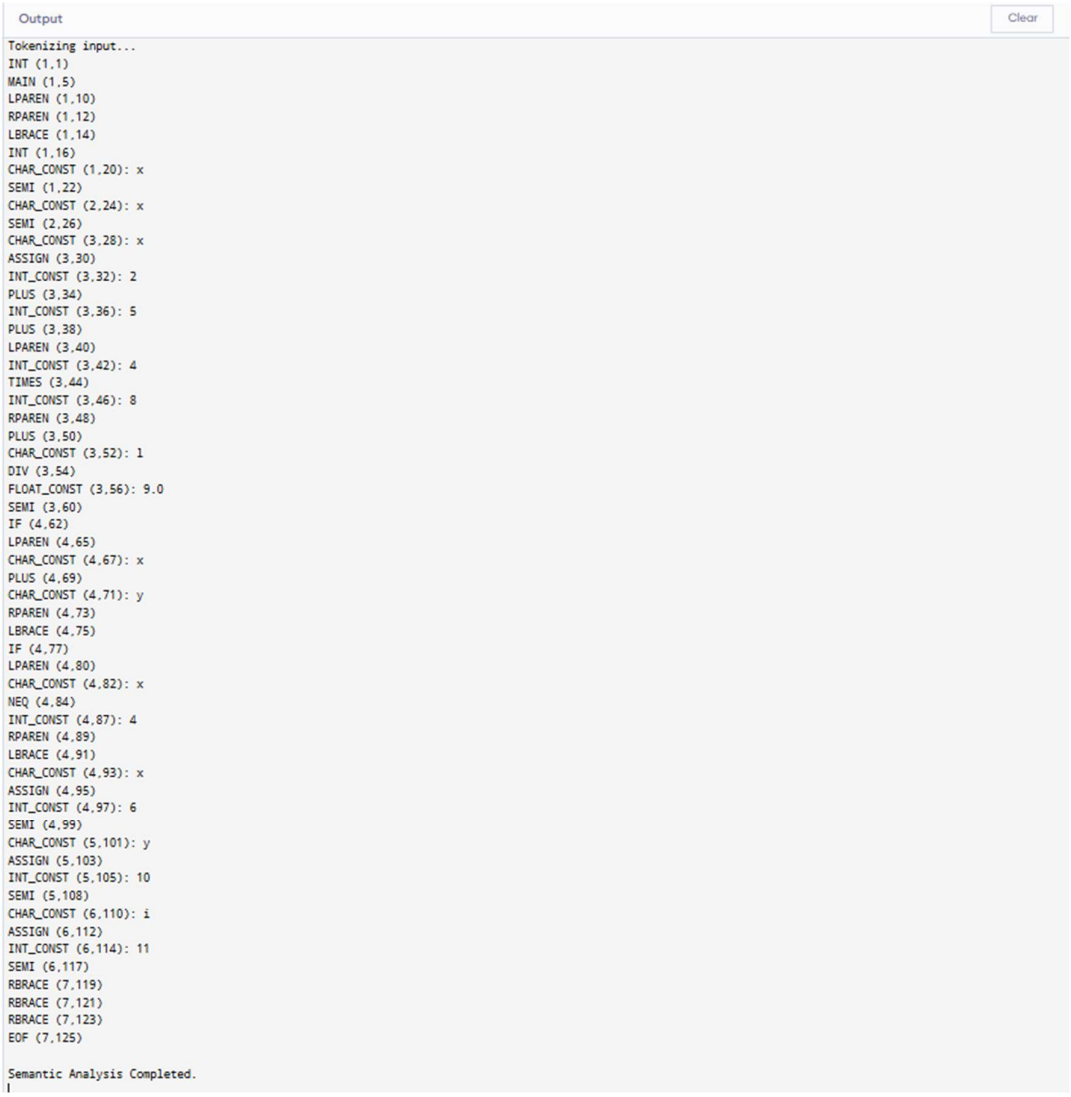
```
            else if (token == "(")
               Console.WriteLine($"LPAREN ({row},{col})");
            else if (token == ")")
               Console.WriteLine($"RPAREN ({row},{col})");
            else if (token == "{")
               Console.WriteLine($"LBRACE ({row},{col})");
            else if (token == "}")
               Console.WriteLine($"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($"NEQ ({row},{col})");
            else if (token == "if")
               Console.WriteLine($"IF ({row},{col})");
            else if (token == "else")
               Console.WriteLine($"ELSE ({row},{col})");
            else if (Regex.IsMatch(token, @"^[0-9]+$"))
               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 ({row},{col})");
    }

    static void Semantic_Analysis(int k)
    {
        if (k <= 0 || k >= finalArray.Count - 1) return;

        // Handle addition or subtraction between variables
        if (finalArray[k] == "+" || finalArray[k] == "-")
        {
            if (variable_Reg.Match(finalArray[k - 1]).Success
&& variable_Reg.Match(finalArray[k + 1]).Success)
            {
                // find type from symbol table for left operand
                int leftSymbolIndex = FindSymbol(finalArray[k -
1]);
                int rightSymbolIndex = FindSymbol(finalArray[k +
1]);
                int assignSymbolIndex = FindSymbol(finalArray[k
- 3]); // variable on LHS of assignment

                if (leftSymbolIndex == -1 || rightSymbolIndex == -
1 || assignSymbolIndex == -1)
                    return;

                string type = Symboltable[assignSymbolIndex][2];

                if (type == Symboltable[leftSymbolIndex][2] &&
type == Symboltable[rightSymbolIndex][2])
                {
                    // Calculate sum of constant values
                    int ans =
Convert.ToInt32(Symboltable[leftSymbolIndex][3]) +
```

```
Convert.ToInt32(Symboltable[rightSymbolIndex][3]);


                    Constants.Add(ans);

                    // Update symbol table for assigned variable
                    Symboltable[assignSymbolIndex][3] =
ans.ToString();

                    if (Constants.Count > 0)
                       Constants.RemoveAt(Constants.Count - 1);
                    Constants.Add(ans);
                }
            }
        }

        // Handle '>' condition in if statement (simplified)
        if (finalArray[k] == ">")
        {

            if (variable_Reg.Match(finalArray[k - 1]).Success
&& variable_Reg.Match(finalArray[k + 1]).Success)
            {
                int before_i = FindSymbol(finalArray[k - 1]);
                int after_i = FindSymbol(finalArray[k + 1]);

                if (before_i == -1 || after_i == -1)
                    return;

                if (Convert.ToInt32(Symboltable[before_i][3]) >
Convert.ToInt32(Symboltable[after_i][3]))
                {
                    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_of_else = finalArray.IndexOf("else");
        if (start_of_else == -1) return;

        int braceCount = 0;
        int end_of_else = -1;
        for (int i = start_of_else; i < finalArray.Count; i++)
        {
            if (finalArray[i] == "{") braceCount++;
            else if (finalArray[i] == "}") braceCount--;

            if (braceCount == 0 && i > start_of_else)
            {
                end_of_else = i;
                break;
            }
        }

        if (end_of_else == -1) return;

        finalArray.RemoveRange(start_of_else, end_of_else -
start_of_else + 1);
    }

    static void RemoveIfBlock()
    {
```

```
            int start_of_if = finalArray.IndexOf("if");
            if (start_of_if == -1) return;

            int braceCount = 0;
            int end_of_if = -1;
            for (int i = start_of_if; i < finalArray.Count; i++)
            {
               if (finalArray[i] == "{") braceCount++;
               else if (finalArray[i] == "}") braceCount--;

               if (braceCount == 0 && i > start_of_if)
               {
                  end_of_if = i;
                  break;
               }
            }
            if (end_of_if == -1) return;

            finalArray.RemoveRange(start_of_if, end_of_if -

start_of_if + 1);
         }
      }
   }
```



Task 2:

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

namespace InteractiveSemanticAnalyzer
{
    class Program
    {
        static List<List<string>> SymbolTable = new
List<List<string>>();
        static List<string> Tokens = new List<string>();
        static Regex variableRegex = new Regex(@"^[A-Za-
z_][A-Za-z0-9_]*$");
```

```
        static int currentTokenIndex = 0;

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

            Console.WriteLine("Enter your source code line by line
(type 'END' to finish):");

            string line;
            while ((line = Console.ReadLine()) != null &&
line.Trim() != "END")
            {
                var tokenized = Tokenize(line);
                Tokens.AddRange(tokenized);
            }

            Console.WriteLine("\nTokens:");

            foreach (var token in Tokens) Console.Write(token + "
");
            Console.WriteLine("\n\nParsing and Performing Syntax
Directed Translation...\n");

            try
            {
                ParseProgram();
                Console.WriteLine("\n✅ Semantic Analysis
Completed.");
            }
            catch (Exception ex)
            {
                Console.WriteLine("❌ Error: " + ex.Message);
            }

            Console.ReadLine();
        }

        static void InitializeSymbolTable()
        {
```

```
        // You can pre-add default variables or keep it empty
    }

    static List<string> Tokenize(string line)
    {
        var tokens = new List<string>();
        var pattern = @"\d+(\.\d+)?|[A-Za-z_][A-Za-z0-
9_]*|==|!=|<=|>=|[+\-*/=;(){}<>,]";
        foreach (Match match in Regex.Matches(line, pattern))
        {
            tokens.Add(match.Value);
        }
        return tokens;
    }

    static string Peek(int offset = 0)
    {

        if (currentTokenIndex + offset < Tokens.Count)
            return Tokens[currentTokenIndex + offset];
        return null;
    }

    static bool Match(string expected)
    {
        if (Peek() == expected)
        {
            currentTokenIndex++;
            return true;
        }
        throw new Exception($"Syntax Error: Expected
'{expected}', found '{Peek()}'");
    }

    static void ParseProgram()
    {
        Match("int");
        Match("main");
        Match("(");
```

```
            Match(")");
            ParseBlock();
        }

        static void ParseBlock()
        {
            Match("{");
            while (Peek() != "}" && Peek() != null)
                ParseStatement();
            Match("}");
        }

        static void ParseStatement()
        {
            if (Peek() == "int")
                ParseDeclaration();
            else if (Peek() == "if")
                ParseIfStatement();

            else if (variableRegex.IsMatch(Peek()))
                ParseAssignment();
            else
                throw new Exception($"Unexpected token '{Peek()}'
in statement.");
        }

        static void ParseDeclaration()
        {
            Match("int");
            string name = Peek();
            Match(name);
            Match(";");
            AddToSymbolTable(name, "int", "0");
        }

        static void ParseAssignment()
        {
            string name = Peek();
            Match(name);
            Match("=");
```

```
        int value = ParseExpression();
        Match(";");
        UpdateSymbolTable(name, value.ToString());
        Console.WriteLine($"[Semantic] {name} = {value}");
    }

    static void ParseIfStatement()
    {
        Match("if");
        Match("(");
        bool condition = ParseCondition();
        Match(")");
        if (condition)
            ParseBlock();
        else
            SkipBlock();
    }


    static bool ParseCondition()
    {
        int left = ParseExpression();
        string op = Peek();
        Match(op);
        int right = ParseExpression();

        return op switch
        {
            "==" => left == right,
            "!=" => left != right,
            ">" => left > right,
            "<" => left < right,
            ">=" => left >= right,
            "<=" => left <= right,
            _ => throw new Exception($"Unknown conditional
operator '{op}'"),
        };
    }
```

```
static int ParseExpression()
{
    int result = ParseTerm();
    while (Peek() == "+" || Peek() == "-")
    {
        string op = Peek();
        Match(op);
        int right = ParseTerm();
        result = op == "+" ? result + right : result - right;
    }
    return result;
}

static int ParseTerm()
{
    int result = ParseFactor();
    while (Peek() == "*" || Peek() == "/")
    {
        string op = Peek();
        Match(op);
        int right = ParseFactor();
        result = op == "*" ? result * right : result / right;
    }
    return result;
}

static int ParseFactor()
{
    string token = Peek();
    if (token == "(")
    {
        Match("(");
        int value = ParseExpression();
        Match(")");
        return value;
    }
    else if (int.TryParse(token, out int num))
    {
        Match(token);
```

```
            return num;
        }
        else if (variableRegex.IsMatch(token))
        {
            Match(token);
            return GetSymbolValue(token);
        }
        else
        {
            throw new Exception($"Invalid token '{token}' in
expression.");
        }
    }

    static void SkipBlock()
    {
        Match("{");
        int braceCount = 1;
        while (braceCount > 0 && currentTokenIndex <
Tokens.Count)
        {
            if (Peek() == "{") braceCount++;
            else if (Peek() == "}") braceCount--;
            currentTokenIndex++;
        }
    }

    static void AddToSymbolTable(string name, string type,
string value)
    {
        if (FindSymbol(name) == -1)
        {
            SymbolTable.Add(new List<string> { name, "id",
type, value });
            Console.WriteLine($"[Declare] {name} as {type}");
        }
        else
        {
            throw new Exception($"Variable '{name}' already
```

```
declared.");
            }
        }

        static void UpdateSymbolTable(string name, string value)
        {
            int index = FindSymbol(name);
            if (index != -1)
                SymbolTable[index][3] = value;
            else
                throw new Exception($"Variable '{name}' not
declared.");
        }

        static int GetSymbolValue(string name)
        {
            int index = FindSymbol(name);
            if (index != -1)
                return int.Parse(SymbolTable[index][3]);
            throw new Exception($"Variable '{name}' not
declared.");
        }

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

```
Output
Enter your source code line by line (type 'END' to finish):
int main(){

Tokens:
int main ( ) { int x ; x = 5 ; }

Parsing and Performing Syntax Directed Translation...

[Declare] x as int
[Semantic] x = 5

? Semantic Analysis Completed.

=== Code Execution Successful ===
```

Lab 12:

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

namespace InteractiveSemanticAnalyzer
{
   class Program
   {
      static List<List<string>> SymbolTable = new
List<List<string>>();
      static List<string> Tokens = new List<string>();
      static Regex variableRegex = new Regex(@"^[A-Za-
z_][A-Za-z0-9_]*$");
      static int currentTokenIndex = 0;

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

         Console.WriteLine("Enter your source code line by line
(type 'END' to finish):");

         string line;
         while ((line = Console.ReadLine()) != null &&
```

```
line.Trim() != "END")
            {
                var tokenized = Tokenize(line);
                Tokens.AddRange(tokenized);
            }

            Console.WriteLine("\nTokens:");
            foreach (var token in Tokens) Console.Write(token + "
");
            Console.WriteLine("\n\nParsing and Performing Syntax
Directed Translation...\n");

            try
            {
                ParseProgram();
                Console.WriteLine("\n✅ Semantic Analysis
Completed.");
            }
            catch (Exception ex)
            {
                Console.WriteLine("❌ Error: " + ex.Message);
            }

            Console.ReadLine();
        }

        static void InitializeSymbolTable()
        {

            // You can pre-add default variables or keep it empty
        }

        static List<string> Tokenize(string line)
        {
            var tokens = new List<string>();
            var pattern = @"\d+(\.\d+)?|[A-Za-z_][A-Za-z0-
9_]*|==|!=|<=|>=|[+\-*/=;(){}<>,]";
            foreach (Match match in Regex.Matches(line, pattern))
            {
```

```
            tokens.Add(match.Value);
        }
        return tokens;
    }

    static string Peek(int offset = 0)
    {
        if (currentTokenIndex + offset < Tokens.Count)
            return Tokens[currentTokenIndex + offset];
        return null;
    }

    static bool Match(string expected)
    {
        if (Peek() == expected)
        {
            currentTokenIndex++;
            return true;
        }
        throw new Exception($"Syntax Error: Expected
'{expected}', found '{Peek()}'");
    }

    static void ParseProgram()
    {
        Match("int");
        Match("main");
        Match("(");
        Match(")");
        ParseBlock();
    }

    static void ParseBlock()
    {
        Match("{");
        while (Peek() != "}" && Peek() != null)
            ParseStatement();
        Match("}");
    }
```

```
        static void ParseStatement()
        {
            if (Peek() == "int")
                ParseDeclaration();
            else if (Peek() == "if")
                ParseIfStatement();
            else if (variableRegex.IsMatch(Peek()))
                ParseAssignment();
            else
                throw new Exception($"Unexpected token '{Peek()}'
in statement.");
        }

        static void ParseDeclaration()
        {
            Match("int");
            string name = Peek();
            Match(name);
            Match(";");
            AddToSymbolTable(name, "int", "0");
        }

        static void ParseAssignment()
        {
            string name = Peek();
            Match(name);
            Match("=");
            int value = ParseExpression();
            Match(";");
            UpdateSymbolTable(name, value.ToString());
            Console.WriteLine($"[Semantic] {name} = {value}");
        }

        static void ParseIfStatement()
        {
            Match("if");
            Match("(");
            bool condition = ParseCondition();
            Match(")");
            if (condition)
```

```
            ParseBlock();
        else
            SkipBlock();
    }

    static bool ParseCondition()
    {
        int left = ParseExpression();
        string op = Peek();
        Match(op);
        int right = ParseExpression();

        return op switch
        {
            "==" => left == right,
            "!=" => left != right,
            ">" => left > right,
            "<" => left < right,
            ">=" => left >= right,
            "<=" => left <= right,
            _ => throw new Exception($"Unknown conditional
operator '{op}'"),
        };
    }

    static int ParseExpression()
    {
        int result = ParseTerm();
        while (Peek() == "+" || Peek() == "-")
        {
            string op = Peek();
            Match(op);
            int right = ParseTerm();
            result = op == "+" ? result + right : result - right;
        }
        return result;
    }

    static int ParseTerm()
    {
```

```
            int result = ParseFactor();
            while (Peek() == "*" || Peek() == "/")
            {
                string op = Peek();
                Match(op);
                int right = ParseFactor();
                result = op == "*" ? result * right : result / right;
            }
            return result;
        }

        static int ParseFactor()
        {
            string token = Peek();
            if (token == "(")
            {
                Match("(");
                int value = ParseExpression();
                Match(")");
                return value;
            }
            else if (int.TryParse(token, out int num))
            {
                Match(token);
                return num;
            }
            else if (variableRegex.IsMatch(token))
            {
                Match(token);
                return GetSymbolValue(token);
            }
            else
            {
                throw new Exception($"Invalid token '{token}' in
expression.");
            }
        }

        static void SkipBlock()
        {
```

```
            Match("{");
            int braceCount = 1;
            while (braceCount > 0 && currentTokenIndex <
Tokens.Count)
            {
                if (Peek() == "{") braceCount++;
                else if (Peek() == "}") braceCount--;
                currentTokenIndex++;
            }
        }

        static void AddToSymbolTable(string name, string type,
string value)
        {
            if (FindSymbol(name) == -1)
            {
                SymbolTable.Add(new List<string> { name, "id",
type, value });
                Console.WriteLine($"[Declare] {name} as {type}");
            }
            else
            {
                throw new Exception($"Variable '{name}' already
declared.");
            }
        }

        static void UpdateSymbolTable(string name, string value)
        {
            int index = FindSymbol(name);
            if (index != -1)
                SymbolTable[index][3] = value;
            else
                throw new Exception($"Variable '{name}' not
declared.");
        }

        static int GetSymbolValue(string name)
        {
```

```
            int index = FindSymbol(name);
            if (index != -1)
                return int.Parse(SymbolTable[index][3]);
            throw new Exception($"Variable '{name}' not
declared.");
        }

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

Output Clear

```
Enter code (end with an empty line):
int x = 5;
< keyword, int >
< id, x >
< op, = >
< digit, 5 >
< punc, ; >
< keyword, float >
< id, y >
< op, = >
< digit, 10.5 >
< punc, ; >
< id, x >
< op, = >
< id, x >
< op, + >
< digit, 3 >
< punc, ; >

=== Code Execution Successful ===
```