

**COMSATS UNIVERSITY ISLAMABAD**  
**ATTOCK CAMPUS**



**DEPARTMENT OF COMPUTER SCIENCE**

**Lab 1 to 12**

**Submitted by:-**

**Moazzam Azam**

**Reg No:-**

**SP22-BCS-010**

**Submitted to:-**

**Syed Bilal Haider**

**Subject:-**

**Compiler construction**

**Date:-**

**05/25/2025**

## Lab 1

```
using System;

using System.Text.RegularExpressions;

public class Program
{
    public static void Main()
    {
        // Hardcoded password for validation
        string password = "Sp22-bcs-036"; // Example password

        // Regular expression pattern for the requirements
        string pattern = @"^(?=.*\d.){2})(?=.*[A-Z])(?=.*[a-z]){4})(?=.*[!@#$%^&*()_.,?\"{ }|<>]){2}).{1,12}$";

        // Check if the password matches the pattern
        if (Regex.IsMatch(password, pattern))
        {
            Console.WriteLine("Password is valid.");
        }
        else
        {
            Console.WriteLine("Password is invalid.");
        }
    }
}
```

## Output

```
Password is valid.  
  
=== Code Execution Successful ===
```

## Task 2

```
using System;  
  
using System.Text;  
using System.Text.RegularExpressions;  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        // Sample inputs  
        Console.WriteLine("Enter your first name: ");  
        string firstName = Console.ReadLine();  
  
        Console.WriteLine("Enter your last name: ");  
        string lastName = Console.ReadLine();  
  
        Console.WriteLine("Enter your registration number: ");  
        string regNumber = Console.ReadLine();  
  
        Console.WriteLine("Enter your favorite food: ");  
        string food = Console.ReadLine();  
  
        Console.WriteLine("Enter your favorite game: ");  
        string game = Console.ReadLine();
```

```

// Generate the password
string password = GeneratePassword(firstName, lastName, regNumber, food, game);

// Display the generated password
Console.WriteLine("Generated Password: " + password);
}

static string GeneratePassword(string firstName, string lastName, string regNumber, string food,
string game)
{
    // Combine all input values
    string combined = firstName + lastName + regNumber + food + game;

    // Regular expression to remove any unwanted characters (non-alphanumeric)
    string sanitized = Regex.Replace(combined, @"[^a-zA-Z0-9]", "");

    // Make the string more complex by adding special characters and digits
    string complexPassword = sanitized;

    // Add some random numbers and special characters
    Random rand = new Random();
    string specialChars = "!@#$%^&*()_+[]{}|;:,<>?/~`";
    for (int i = 0; i < 4; i++)
    {
        // Add random number
        complexPassword += rand.Next(0, 10).ToString();

        // Add random special character

```

```

        complexPassword += specialChars[rand.Next(specialChars.Length)];
    }

    // Ensure password length is at least 12 characters
    if (complexPassword.Length < 12)
    {
        complexPassword = complexPassword.PadLeft(12, 'X'); // Add filler 'X' if too short
    }

    // Randomly shuffle the password to increase complexity
    StringBuilder shuffledPassword = new StringBuilder();
    while (complexPassword.Length > 0)
    {
        int index = rand.Next(complexPassword.Length);
        shuffledPassword.Append(complexPassword[index]);
        complexPassword = complexPassword.Remove(index, 1);
    }

    return shuffledPassword.ToString();
}
}

```

### Output

```

Enter your first name:
usman
Enter your last name:
malik
Enter your registration number:
036
Enter your favorite food:
cake
Enter your favorite game:
football
Generated Password: k4a2omnkafs11ja6mit!e939luca0?ob!

=== Code Execution Successful ===

```

## LAB 2

## TASK 1

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

class Program
{
    static void Main()
    {
        // The regular expression for logical operators and parentheses
        string pattern = @"s*(&|\\|\\|!|\\(|\\))s*";

        // Test string with logical operators and parentheses
        string input = "x && y || !z (x || y)";

        // Create a Regex object with the pattern
        Regex regex = new Regex(pattern);

        // Find all matches
        MatchCollection matches = regex.Matches(input);

        // Output the matches
        foreach (Match match in matches)
        {
            Console.WriteLine($"Found: {match.Value}");
        }
    }
}
```

## Output

Found: &&

Found: ||

Found: !

Found: (

Found: ||

Found: )

=== Code Execution Successful ===

## TASK 2

```
using System;
```

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

```
class Program
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        // The regular expression for relational operators
```

```
        string pattern = @"\s*(==|!=|>=|<=|>|<)\s*";
```

```
        // Test string with relational operators
```

```
        string input = "a == b && c != d || e >= f && g < h";
```

```
        // Create a Regex object with the pattern
```

```
        Regex regex = new Regex(pattern);
```

```
        // Find all matches
```

```
        MatchCollection matches = regex.Matches(input);
```

```

// Output the matches
foreach (Match match in matches)
{
    Console.WriteLine($"Found: {match.Value}");
}
}

```

### Output

```

Found:  ==
Found:  !=
Found:  >=
Found:  <

```

```

=== Code Execution Successful ===

```

## LAB 3

### TASK 1

```

using System;

using System.Text.RegularExpressions;

class Program
{
    static void Main()
    {
        // Regular expression for floating point numbers with length <= 6
        string pattern = @"^[+-]?\d{1,3}(\.\d{1,3})?$/^[+-]?\.\d{1,3}$/";

        // Test strings
        string[] testStrings = {

```



```
"123",    // valid
"-12.34", // valid
"+0.567", // valid
".678",   // valid
"0.5",    // valid
"123456", // invalid
"1.2345", // invalid
"+1234",  // invalid
".1234"   // invalid
};

// Check each string against the regex
foreach (var test in testStrings)
{
    bool isMatch = Regex.IsMatch(test, pattern);
    Console.WriteLine($"{test}: {(isMatch ? "Valid" : "Invalid")}");
}
}
```

### Output

```
123: Valid
-12.34: Valid
+0.567: Valid
.678: Valid
0.5: Valid
123456: Invalid
1.2345: Invalid
+1234: Invalid
.1234: Invalid

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

## LAB 4

### TASK 1

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define BUFFER_SIZE 1024
#define KEYWORD_COUNT 3

char buffer1[BUFFER_SIZE], buffer2[BUFFER_SIZE];
char *lexeme_start, *forward;
int active_buffer = 1;

char *keywords[KEYWORD_COUNT] = {"int", "if", "else"};

void switch_buffer() {
    if (active_buffer == 1) {
        forward = buffer2;
```

```
        active_buffer = 2;
    } else {
        forward = buffer1;
        active_buffer = 1;
    }
}
```

```
int is_keyword(char *lexeme) {
    for (int i = 0; i < KEYWORD_COUNT; i++) {
        if (strcmp(lexeme, keywords[i]) == 0)
            return 1;
    }
    return 0;
}
```

```
void lexical_analyzer() {
    char lexeme[100];
    int lexeme_length = 0;

    while (*forward != '\0') {
        if (isspace(*forward)) {
            forward++;
            continue;
        }

        lexeme_start = forward;

        if (isalpha(*forward)) {
            while (isalnum(*forward)) {
```

```

        lexeme[lexeme_length++] = *forward;
        forward++;
    }
    lexeme[lexeme_length] = '\0';

    if (is_keyword(lexeme))
        printf("Keyword: %s\n", lexeme);
    else
        printf("Identifier: %s\n", lexeme);
}

else if (isdigit(*forward)) {
    while (isdigit(*forward)) {
        lexeme[lexeme_length++] = *forward;
        forward++;
    }
    lexeme[lexeme_length] = '\0';
    printf("Number: %s\n", lexeme);
}

else {
    printf("Operator: %c\n", *forward);
    forward++;
}

lexeme_length = 0;
}
}

int main() {
    printf("Enter input code: ");

```

```

fgets(buffer1, BUFFER_SIZE, stdin); // Take input from user

forward = buffer1;

active_buffer = 1;


lexical_analyzer();

return 0;

}

```

### Output

```

Enter input code: a+b=c
Identifier: a
Operator: +
Identifier: b
Operator: =
Identifier: c

=== Code Execution Successful ===

```

## Lab 5

### Task 1

```

#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

### Task 1

using System;

class RecursiveDescentParser

{

static string input;

static int index = 0;

static void Main()

{

Console.WriteLine("Enter an arithmetic expression");

input = Console.ReadLine();

input = input.Replace(" ", ""); // remove spaces

try

{

E();

if (index == input.Length)

{

Console.WriteLine("Expression is valid!");

}

else

{

Console.WriteLine("Invalid Expression!");

}

}

catch

{

```
        Console.WriteLine("Invalid Expression!");
    }
}
```

```
//  $E \rightarrow T E'$ 
static void E()
{
    T();
    EPrime();
}
```

```
//  $E' \rightarrow + T E' \mid \epsilon$ 
static void EPrime()
{
    if (Match('+'))
    {
        T();
        EPrime();
    }
}
```

```
//  $T \rightarrow F T'$ 
static void T()
{
    F();
    TPrime();
}
```

```
//  $T' \rightarrow * F T' \mid \epsilon$ 
```

```
static void TPrime()
```

```
{
```

```
    if (Match('*'))
```

```
    {
```

```
        F();
```

```
        TPrime();
```

```
    }
```

```
}
```

```
//  $F \rightarrow (E) \mid id$ 
```

```
static void F()
```

```
{
```

```
    if (Match('('))
```

```
    {
```

```
        E();
```

```
        if (!Match(' '))
```

```
            throw new Exception("Missing ");
```

```
    }
```

```
    else if (Char.IsDigit(CurrentChar()))
```

```
    {
```

```
        while (Char.IsDigit(CurrentChar()))
```

```
            index++; // consume the number
```

```
    }
```

```
    else
```

```
    {
```

```
        throw new Exception("Invalid character in F()");
```

```
    }
```

```
}
```

```

static bool Match(char expected)
{
    if (index < input.Length && input[index] == expected)
    {
        index++;
        return true;
    }
    return false;
}

static char CurrentChar()
{
    return index < input.Length ? input[index] : '\0';
}
}

```

Output
<pre> Enter an arithmetic expression 2+3*4 Expression is valid!  === Code Execution Successful === </pre>

## Lab 7

### Task 1

```
using System;
```

```
class GrammarParser
```

```

{
    static string input;

```

```
static int index = 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
```

```
    try
```

```
    {
```

```
        S();
```

```
        if (index == input.Length)
```

```
        {
```

```
            Console.WriteLine("Valid Syntax!");
```

```
        }
```

```
    else
```

```
    {
```

```
        Console.WriteLine("Invalid Syntax!");
```

```
    }
```

```
}
```

```
catch
```

```
{
```

```
    Console.WriteLine("Invalid Syntax!");
```

```
}
```

```
}
```

```
// S → if(C){S}else{S} | id=E;
```

```
static void S()
```

```
{
```

```

if (Match("if"))
{
    Match("(");
    C();
    Match(")");
    Match("{");
    S();
    Match("}");
    Match("else");
    Match("{");
    S();
    Match("}");
}
else if (Match("id"))
{
    Match("=");
    E();
    Match(";");
}
else
{
    throw new Exception("Invalid Statement");
}
}

```

//  $C \rightarrow id<num$

```

static void C()
{
    Match("id");
}

```

```
    Match("<");  
    Match("num");  
}
```

```
// E → T+T  
static void E()  
{  
    T();  
    Match("+");  
    T();  
}
```

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

```
// Helpers  
static bool Match(string token)  
{  
    if (input.Substring(index).StartsWith(token))  
    {  
        index += token.Length;  
        return true;  
    }  
    return false;  
}
```



```
}
```

### Output

```
Enter statement (e.g., if(id<num){id=5+3;}else{id=2+1;}):  
if(id<num){id=5+3;}else{id=2+1;}  
Invalid Syntax!
```

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

## Lab 08

### Task 1

```
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; // ✓ add break to fix the error
    }
}

return state == 1;
}
```

## Output

Enter a variable name to check:

my\_var1

? Valid C variable name.

=== Code Execution Successful ===

## Lab 10

### Task 1

```
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
```

```
            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
};

private readonly string[,] action = new string[12, 6]
{
    // id + * ( ) $
    { "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
};

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}
};

```

```

public List<string> Tokenize(string input)
{
    var tokens = new List<string>();
    var parts = input.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
    foreach (var part in parts)
    {
        if (part == "id") tokens.Add("id");
        else if (part == "+") tokens.Add("+");
        else if (part == "*") tokens.Add("*");
        else if (part == "(") tokens.Add("(");
        else if (part == ")") tokens.Add(")");
    }
}

```

```

        else throw new Exception($"Unknown token: {part}");
    }
    tokens.Add("$");
    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)),
            act == "" ? "error" : act);
    }
}

```

```

if (act == "")
{
    Console.WriteLine("Parsing error!");
    return;
}
else if (act[0] == 'S')
{
    int nextState = int.Parse(act.Substring(1));
    symbolStack.Push(currToken);
    stateStack.Push(nextState);
    ip++;
}
else if (act[0] == 'R')
{
    int prodNum = int.Parse(act.Substring(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();
        parser.Parse(input);
    }
}
}

```



Output

Clear

Enter input (tokens separated by spaces, e.g., id + id \* id):  
id + id

State	Stack	Symbol Stack	Input
Action			
-----			
0			id + id \$
	S5		
0 5		id	+ id \$
	R6		
0 3		F	+ id \$
	R4		
0 2		T	+ id \$
	R2		
0 1		E	+ id \$
	S6		
0 1 6		E +	id \$
	S5		
0 1 6 5		E + id	\$
	R6		.
0 1 6 3		E + F	\$
	R4		
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 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;

    if (finalArray[k] == "+" || finalArray[k] == "-")
    {
        if (k > 0 && k < finalArray.Count - 1 &&
            variable_Reg.Match(finalArray[k - 1]).Success &&
            variable_Reg.Match(finalArray[k + 1]).Success)
        {
            string type = finalArray[k - 4];
            string left_side = finalArray[k - 3];
            string before = finalArray[k - 1];
            string after = finalArray[k + 1];

```

```

int left_side_i = FindSymbol(left_side);

int before_i = FindSymbol(before);

int after_i = FindSymbol(after);


if (type == Symboltable[before_i][2] && type == Symboltable[after_i][2])
{
    int ans = Convert.ToInt32(Symboltable[before_i][3]) +
Convert.ToInt32(Symboltable[after_i][3]);
    Constants.Add(ans);
}


if (Symboltable[left_side_i][2] == Symboltable[before_i][2] &&
    Symboltable[left_side_i][2] == Symboltable[after_i][2])
{
    int ans = Convert.ToInt32(Symboltable[before_i][3]) +
Convert.ToInt32(Symboltable[after_i][3]);
    if (Constants.Count > 0) Constants.RemoveAt(Constants.Count - 1);
    Constants.Add(ans);
    Symboltable[left_side_i][3] = ans.ToString();
}
}

if (finalArray[k] == ">")
{
    if (k > 0 && k < finalArray.Count - 1 &&
        variable_Reg.Match(finalArray[k - 1]).Success &&
        variable_Reg.Match(finalArray[k + 1]).Success)

```

```

{
    string before = finalArray[k - 1];
    string after = finalArray[k + 1];

    int before_i = FindSymbol(before);
    int after_i = FindSymbol(after);

    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");
    int end_of_else = finalArray.Count - 1;
    for (int i = end_of_else; i >= start_of_else; i--)
    {
        if (finalArray[i] == "{") { end_of_else = i; }
    }

    for (int i = start_of_else; i <= end_of_else; i++)
    {
        finalArray.RemoveAt(start_of_else);
    }
}

static void RemoveIfBlock()
{
    int start_of_if = finalArray.IndexOf("if");
    int end_of_if = finalArray.IndexOf("}");

    for (int i = start_of_if; i <= end_of_if; i++)
    {
        finalArray.RemoveAt(start_of_if);
    }
}
}
```



## Output

```
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)
```

## 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")
        ParseSelfStatement();
    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 ParseSelfStatement()
```

```
{
    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

### TASK 1

```
using System;
```

```
using System.Collections;
```

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

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

```
namespace LexicalAnalyzerV1
```

```
{
```

```
    class Token
```

```
    {
```

```
        public string Type;
```

```
        public string Value;
```

```
        public int Line;
```

```
public int Column;
```

```
public Token(string type, string value, int line, int column)
```

```
{
```

```
    Type = type;
```

```
    Value = value;
```

```
    Line = line;
```

```
    Column = column;
```

```
}
```

```
}
```

```
class Program
```

```
{
```

```
    static List<string> keywordList = new List<string> { "int", "float", "while", "main", "if", "else", "new"
};
```

```
    static Regex variable_Reg = new Regex(@"^[A-Za-z_][A-Za-z0-9]*$");
```

```
    static Regex constants_Reg = new Regex(@"^[0-9]+([.][0-9]+)?([e]([+|-]?[0-9]+)?$");
```

```
    static Regex operators_Reg = new Regex(@"^[~*+>/<&|=]$");
```

```
    static Regex Special_Reg = new Regex(@"^[.,'\[\]\{\}\(\);:~]$");
```

```
    static List<Token> tokens = new List<Token>();
```

```
    static void Main(string[] args)
```

```
{
```

```
    Console.WriteLine("Enter code (end with an empty line):");
```

```
    string userInput = "";
```

```
    string line;
```

```
    // Read multi-line input
```

```
while ((line = Console.ReadLine()) != null && line != "")
{
    userInput += line + "\n";
}

// Lexical Analysis
TokenizeAndPrint(userInput);

// Parser
ParseTokens();
}
```

```
static void TokenizeAndPrint(string input)
{
    tokens.Clear();
    int line_num = 1;
    int col_num = 1;
    int i = 0;

    while (i < input.Length)
    {
        char c = input[i];

        if (c == '\n')
        {
            line_num++;
            col_num = 1;
            i++;
            continue;
        }
    }
}
```

```

    }

    if (char.IsWhiteSpace(c))
    {
        col_num++;

        i++;

        continue;
    }

    // Identifier or keyword
    if (char.IsLetter(c) || c == '_')
    {
        int start = i;
        int startCol = col_num;
        while (i < input.Length && (char.IsLetterOrDigit(input[i]) || input[i] == '_'))
        {
            i++; col_num++;
        }
        string word = input.Substring(start, i - start);
        if (keywordList.Contains(word))
        {
            Console.WriteLine($"< keyword, {word} >");
            tokens.Add(new Token("keyword", word, line_num, startCol));
        }
        else
        {
            Console.WriteLine($"< id, {word} >");
            tokens.Add(new Token("id", word, line_num, startCol));
        }
        continue;
    }

```

```

    }

    // Number
    if (char.IsDigit(c))
    {
        int start = i;

        int startCol = col_num;

        while (i < input.Length && (char.IsDigit(input[i]) || input[i] == '.'))
        {
            i++; col_num++;
        }

        string num = input.Substring(start, i - start);

        Console.WriteLine($"< digit, {num} >");

        tokens.Add(new Token("digit", num, line_num, startCol));

        continue;
    }

    // Operator
    if (operators_Reg.IsMatch(c.ToString()))
    {
        Console.WriteLine($"< op, {c} >");

        tokens.Add(new Token("op", c.ToString(), line_num, col_num));

        i++; col_num++;

        continue;
    }

    // Punctuation/Special
    if (Special_Reg.IsMatch(c.ToString()))
    {

```

```

        Console.WriteLine($"< punc, {c} >");

        tokens.Add(new Token("punc", c.ToString(), line_num, col_num));

        i++; col_num++;

        continue;
    }

    // Unknown character

    Console.WriteLine($"ERROR: {c} at line {line_num}");

    tokens.Add(new Token("error", c.ToString(), line_num, col_num));

    i++; col_num++;
}
}

// --- PARSER ---

static int currentToken = 0;

static Token Peek() => currentToken < tokens.Count ? tokens[currentToken] : null;

static Token Next()
{
    if (currentToken < tokens.Count) return tokens[currentToken++];

    return null;
}

static void Expect(string type, string value = null, string expected = null)
{
    var t = Peek();

    if (t == null || t.Type != type || (value != null && t.Value != value))
    {
        string found = t == null ? "EOF" : t.Value;
    }
}

```



```
        string errMsg = $"ERROR: {found} at line {(t?.Line ?? tokens[tokens.Count - 1].Line)}, column  
        {(t?.Column ?? tokens[tokens.Count - 1].Column)}; Expected {expected ?? type.ToUpper()}";
```

```
        Console.WriteLine(errMsg);
```

```
        throw new Exception(); // stop further parsing after first error
```

```
    }
```

```
    Next();
```

```
}
```

```
static void ParseTokens()
```

```
{
```

```
    currentToken = 0;
```

```
    try
```

```
    {
```

```
        while (currentToken < tokens.Count)
```

```
        {
```

```
            ParseStatement();
```

```
        }
```

```
    }
```

```
    catch
```

```
    {
```

```
        // Stop after first syntax error
```

```
    }
```

```
}
```

```
static void ParseStatement()
```

```
{
```

```
    var t = Peek();
```

```
    if (t == null) return;
```

```

if (t.Type == "keyword" && t.Value == "int")
{
    Next();
    Expect("id", null, "IDENTIFIER");
    Expect("op", "=", "EQUALS SIGN"); // FIXED: Expect '=' as operator, not punctuation
    ParseExpression();
    Expect("punc", ";", "SEMICOLON");
}
else if (t.Type == "id")
{
    Next();
    if (Peek() != null && Peek().Type == "op" && Peek().Value == "=")
    {
        Next();
        ParseExpression();
        Expect("punc", ";", "SEMICOLON");
    }
    else
    {
        var next = Peek();
        string errMsg = $"ERROR: {next?.Value} at line {next?.Line}, column {next?.Column}; Expected
'='";

        Console.WriteLine(errMsg);
        throw new Exception();
    }
}
else
{
    string errMsg = $"ERROR: {t.Value} at line {t.Line}, column {t.Column}; Unexpected token";

```

```

        Console.WriteLine(errMsg);
        throw new Exception();
    }
}

static void ParseExpression()
{
    var t = Peek();
    if (t != null && (t.Type == "id" || t.Type == "digit"))
    {
        Next();
        if (Peek() != null && Peek().Type == "op")
        {
            Next();
            ParseExpression();
        }
    }
    else
    {
        string errMsg = $"ERROR: {t?.Value ?? "EOF"} at line {t?.Line ?? tokens[tokens.Count - 1].Line},
column {t?.Column ?? tokens[tokens.Count - 1].Column}; Expected EXPRESSION";

        Console.WriteLine(errMsg);
        throw new Exception();
    }
}
}

```

## Output

Enter code (end with an empty line):

```
int a=4;
```

```
< keyword, int >
```

```
< id, a >
```

```
< op, = >
```

```
< digit, 4 >
```

```
< punc, ; >
```

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