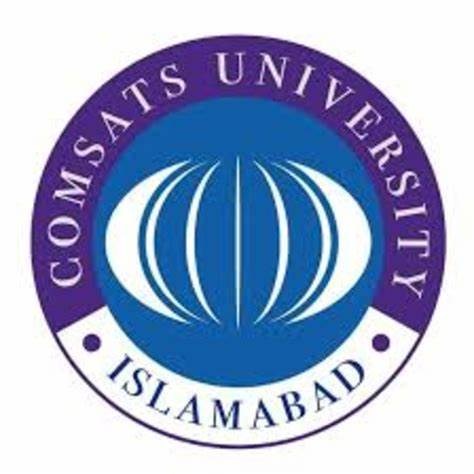
**COMSATS UNIVERSITY ISLAMABAD,**

**ATTOCK CAMPUS**

Department Of Computer Science



**Project: MINI COMPILER**

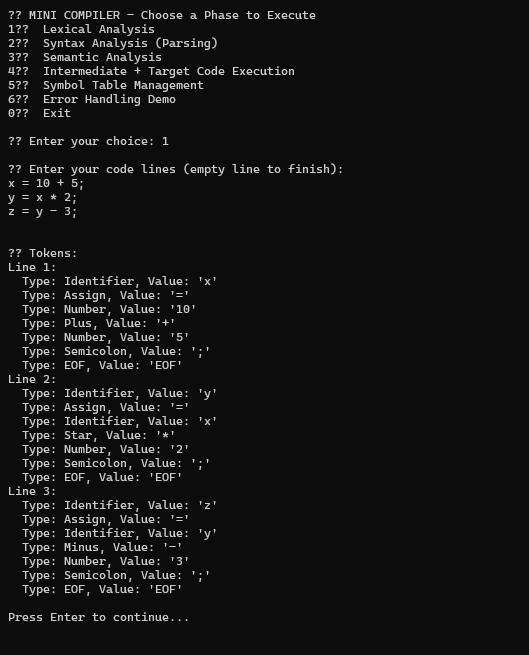
**Course Name: Compiler Construction**

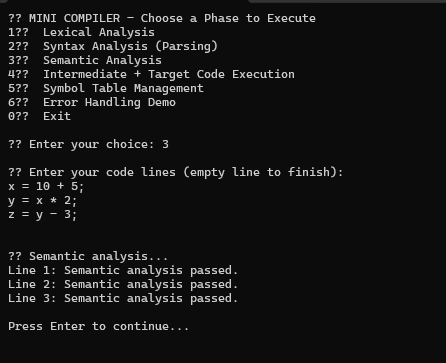
**Submitted To: Sir Bilal**

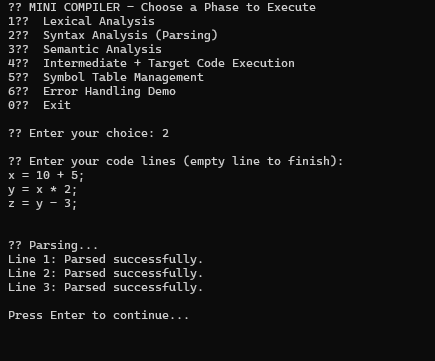
**Submitted By: Kiran Riaz**

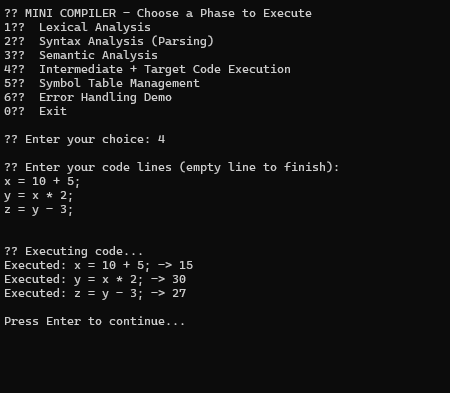
**Registration No: Sp22-Bcs-049**

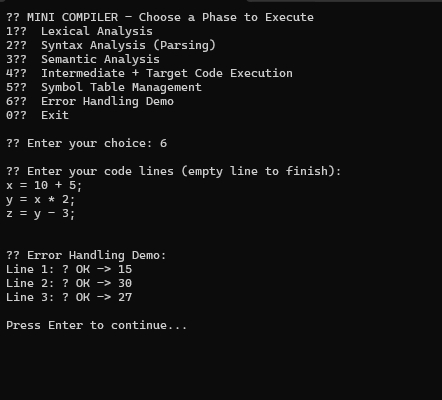
**Output:**











# **CODE:**

using System;

using System.Collections.Generic;

namespace MiniCompiler

{

// Token types

enum TokenType { Number, Identifier, Assign, Plus, Minus, Star, Slash, Semicolon, EOF }

class Token

{

public TokenType Type;

public string Value;

public Token(TokenType type, string value) {

Type = type;

Value = value;

}}

class Lexer {

private string \_input;

private int \_pos;

private List<Token> \_tokens;

public Lexer(string input){

\_input = input;

\_tokens = new List<Token>();

\_pos = 0;}

public List<Token> Tokenize()

{

while (\_pos < \_input.Length)

{

char current = \_input[\_pos];

if (char.IsWhiteSpace(current))

{

\_pos++;

}

else if (char.IsLetter(current))

{

string ident = ReadWhile(char.IsLetterOrDigit);

\_tokens.Add(new Token(TokenType.Identifier, ident));

}

else if (char.IsDigit(current))

{

string number = ReadWhile(char.IsDigit);

\_tokens.Add(new Token(TokenType.Number, number));

}

else

{

switch (current)

{

case '=': \_tokens.Add(new Token(TokenType.Assign, "=")); break;

case '+': \_tokens.Add(new Token(TokenType.Plus, "+")); break;

case '-': \_tokens.Add(new Token(TokenType.Minus, "-")); break;

case '\*': \_tokens.Add(new Token(TokenType.Star, "\*")); break;

case '/': \_tokens.Add(new Token(TokenType.Slash, "/")); break;

case ';': \_tokens.Add(new Token(TokenType.Semicolon, ";")); break;

default: throw new Exception("Unknown character: " + current);

}

\_pos++;

}

}

\_tokens.Add(new Token(TokenType.EOF, "EOF"));

return \_tokens;

}

private string ReadWhile(Func<char, bool> condition)

{

int start = \_pos;

while (\_pos < \_input.Length && condition(\_input[\_pos])) \_pos++;

return \_input.Substring(start, \_pos - start);

}

}

// AST Nodes

abstract class ASTNode { }

class NumberNode : ASTNode

{

public int Value;

public NumberNode(int value) { Value = value; }}

class VarNode : ASTNode

{

public string Name;

public VarNode(string name) { Name = name; }

}

class AssignNode : ASTNode

{

public string Name;

public ASTNode Value;

public AssignNode(string name, ASTNode value)

{

Name = name;

Value = value;

}

}

class BinOpNode : ASTNode

{

public ASTNode Left;

public string Op;

public ASTNode Right;

public BinOpNode(ASTNode left, string op, ASTNode right)

{

Left = left;

Op = op;

Right = right;

}

}

// Parser

class Parser

{

private List<Token> \_tokens;

private int \_pos = 0;

public Parser(List<Token> tokens) { \_tokens = tokens; }

private Token Peek() => \_tokens[\_pos];

private Token Consume() => \_tokens[\_pos++];

public ASTNode ParseStatement()

{

Token ident = Consume();

if (ident.Type != TokenType.Identifier)

throw new Exception("Expected identifier at start of statement.");

Token assign = Consume();

if (assign.Type != TokenType.Assign)

throw new Exception("Expected '=' after identifier.");

ASTNode expr = ParseExpression();

Token semi = Consume();

if (semi.Type != TokenType.Semicolon)

throw new Exception("Expected ';' at end of statement.");

return new AssignNode(ident.Value, expr);

}

private ASTNode ParseExpression()

{

ASTNode left = ParseTerm();

while (Peek().Type == TokenType.Plus || Peek().Type == TokenType.Minus)

{

string op = Consume().Value;

ASTNode right = ParseTerm();

left = new BinOpNode(left, op, right);

}

return left;}

private ASTNode ParseTerm()

{

ASTNode left = ParseFactor();

while (Peek().Type == TokenType.Star || Peek().Type == TokenType.Slash)

{

string op = Consume().Value;

ASTNode right = ParseFactor();

left = new BinOpNode(left, op, right);

}

return left;

}

private ASTNode ParseFactor()

{

Token tok = Consume();

if (tok.Type == TokenType.Number)

return new NumberNode(int.Parse(tok.Value));

else if (tok.Type == TokenType.Identifier)

return new VarNode(tok.Value);

throw new Exception("Unexpected token: " + tok.Value);}}

// Semantic Analyzer

class SemanticAnalyzer

{

private HashSet<string> \_symbols;

public SemanticAnalyzer(HashSet<string> symbols)

{

\_symbols = symbols;}

public void Analyze(ASTNode node)

{

if (node is AssignNode assign)

{

Analyze(assign.Value);

\_symbols.Add(assign.Name);

}

else if (node is VarNode var)

{

if (!\_symbols.Contains(var.Name))

throw new Exception("Undeclared variable: " + var.Name);

}

else if (node is BinOpNode bin)

{

Analyze(bin.Left);

Analyze(bin.Right);

}

else if (node is NumberNode)

{

// no action needed

}

else

{

throw new Exception("Unknown AST node type");}}}

// Interpreter

class Interpreter

{

private Dictionary<string, int> \_memory;

public Interpreter(Dictionary<string, int> memory)

{

\_memory = memory;

}

public int Evaluate(ASTNode node)

{

if (node is NumberNode n) return n.Value;

if (node is VarNode v)

{

if (!\_memory.ContainsKey(v.Name))

throw new Exception($"Variable '{v.Name}' is not initialized.");

return \_memory[v.Name];

}

if (node is AssignNode a)

{

int value = Evaluate(a.Value);

\_memory[a.Name] = value;

return value;

}

if (node is BinOpNode b)

{

int left = Evaluate(b.Left);

int right = Evaluate(b.Right);

switch (b.Op)

{

case "+": return left + right;

case "-": return left - right;

case "\*": return left \* right;

case "/":

if (right == 0)

throw new Exception("Division by zero");

return left / right;

default:

throw new Exception("Unknown operator");

}

}

throw new Exception("Invalid AST node");

}

}

class Program

{

private static Dictionary<string, int> symbolTable = new Dictionary<string, int>();

private static HashSet<string> declaredVars = new HashSet<string>();

static void Main(string[] args)

{

while (true)

{

Console.Clear();

Console.WriteLine("⚙️ MINI COMPILER - Choose a Phase to Execute");

Console.WriteLine("1️⃣ Lexical Analysis");

Console.WriteLine("2️⃣ Syntax Analysis (Parsing)");

Console.WriteLine("3️⃣ Semantic Analysis");

Console.WriteLine("4️⃣ Intermediate + Target Code Execution");

Console.WriteLine("5️⃣ Symbol Table Management");

Console.WriteLine("6️⃣ Error Handling Demo");

Console.WriteLine("0️⃣ Exit");

Console.Write("\n📥 Enter your choice: ");

string choice = Console.ReadLine();

if (choice == "0") break;

List<string> lines = new List<string>();

Console.WriteLine("\n📝 Enter your code lines (empty line to finish):");

while (true)

{

string line = Console.ReadLine();

if (string.IsNullOrWhiteSpace(line)) break;

lines.Add(line);

}

try

{

switch (choice)

{

case "1":

Console.WriteLine("\n🔍 Tokens:");

int lineNum1 = 1;

foreach (var line in lines)

{

Lexer lexer = new Lexer(line);

var tokens = lexer.Tokenize();

Console.WriteLine($"Line {lineNum1++}:");

foreach (var t in tokens)

{

Console.WriteLine($" Type: {t.Type}, Value: '{t.Value}'");

}

}

break;

case "2":

Console.WriteLine("\n🔧 Parsing...");

int lineNum2 = 1;

foreach (var line in lines)

{

Lexer lexer = new Lexer(line);

var tokens = lexer.Tokenize();

Parser parser = new Parser(tokens);

var ast = parser.ParseStatement();

Console.WriteLine($"Line {lineNum2++}: Parsed successfully.");

}

break;

case "3":

Console.WriteLine("\n🔍 Semantic analysis...");

declaredVars.Clear();

int lineNum3 = 1;

foreach (var line in lines)

{

Lexer lexer = new Lexer(line);

var tokens = lexer.Tokenize();

Parser parser = new Parser(tokens);

var ast = parser.ParseStatement();

SemanticAnalyzer semantic = new SemanticAnalyzer(declaredVars);

semantic.Analyze(ast);

Console.WriteLine($"Line {lineNum3++}: Semantic analysis passed.");

}

break;

case "4":

Console.WriteLine("\n▶️ Executing code...");

foreach (var line in lines)

{

Lexer lexer = new Lexer(line);

var tokens = lexer.Tokenize();

Parser parser = new Parser(tokens);

var ast = parser.ParseStatement();

SemanticAnalyzer semantic = new SemanticAnalyzer(declaredVars);

semantic.Analyze(ast);

Interpreter interpreter = new Interpreter(symbolTable);

int result = interpreter.Evaluate(ast);

Console.WriteLine($"Executed: {line} -> {result}");

}

break;

case "5":

Console.WriteLine("\n🔢 Symbol Table:");

if (symbolTable.Count == 0)

Console.WriteLine("No variables declared yet.");

else

{

foreach (var kvp in symbolTable)

{

Console.WriteLine($"{kvp.Key} = {kvp.Value}");

}

}

break;

case "6":

Console.WriteLine("\n⚠️ Error Handling Demo:");

int lineNum = 1;

foreach (var line in lines)

{

try

{

Lexer lexer = new Lexer(line);

var tokens = lexer.Tokenize();

Parser parser = new Parser(tokens);

var ast = parser.ParseStatement();

SemanticAnalyzer semantic = new SemanticAnalyzer(declaredVars);

semantic.Analyze(ast);

Interpreter interpreter = new Interpreter(symbolTable);

int result = interpreter.Evaluate(ast);

Console.WriteLine($"Line {lineNum++}: ✅ OK -> {result}");

}

catch (Exception ex)

{

Console.WriteLine($"Line {lineNum++}: ❌ Error -> {ex.Message}");

}

}

break;

default:

Console.WriteLine("Invalid choice.");

break;

}

Console.WriteLine("\nPress Enter to continue...");

Console.ReadLine();

}

catch (Exception ex)

{

Console.WriteLine($"❌ Error: {ex.Message}");

Console.WriteLine("\nPress Enter to continue...");

Console.ReadLine();

}

}

}

}

}