****

**COMSATS University Islamabad (CUI) Attock Campus**

Compiler Construction Lab  
(Semester Project)

MiniCompiler

***By***

**Kashif Hussain CIIT/FA20-BCS-019/ATK**

***Submitted To:*Mr. Syed Bilal Haider**

Semester Project Report :

MiniCompiler

# **Introduction**:

* This compiler construction project aims to provide a basic understanding of compiler concepts by implementing a simplified compiler in C#.
* The focus is on three fundamental components of a compiler: lexical analysis (scanner), syntax analysis (parser), and code generation.

Project Overview:

Objective:

* Develop a simple compiler in C# covering lexical analysis, syntax analysis, and code generation.
* Enable users to input simple mathematical expressions and generate corresponding intermediate code.

Project Components:

* **Scanner (Lexer):** Tokenizes input expressions.
* **Parser**: Constructs an Abstract Syntax Tree (AST) from the token stream.
* **Intermediate Code Generator**: Generates simple intermediate code from the AST.

User Interface:

* The program prompts the user to input a mathematical expression.
* After processing, the program displays the generated intermediate code.

Lexical Analysis (Scanner):

* Tokenizes the input mathematical expressions.
* Recognizes operators (+, -, \*, /) and operands (numbers and identifiers).

Syntax Analysis (Parser):

* Constructs an AST from the tokenized expressions.
* Handles basic arithmetic expressions with operator precedence.

Code Generation:

* Generates simple intermediate code based on the AST.
* Supports addition, subtraction, multiplication, and division operations.

User Interaction Flow:

* The program provides a command-line interface for users to input mathematical expressions.
* After processing, the program displays the generated intermediate code.

Coding Techniques:

* Object-oriented programming in C#.
* Recursive descent parsing for syntax analysis.
* Simple data structures to represent the AST nodes.

Procedure Details:

## Lexical Analysis (Scanner):

* Identifies and categorizes tokens (operators, operands).
* Provides methods to retrieve the next token.

## Syntax Analysis (Parser):

* Constructs an AST by recursively parsing expressions.
* Handles operator precedence.

## Code Generation:

* Generates intermediate code based on the AST nodes.
* Uses a simple stack-based approach.

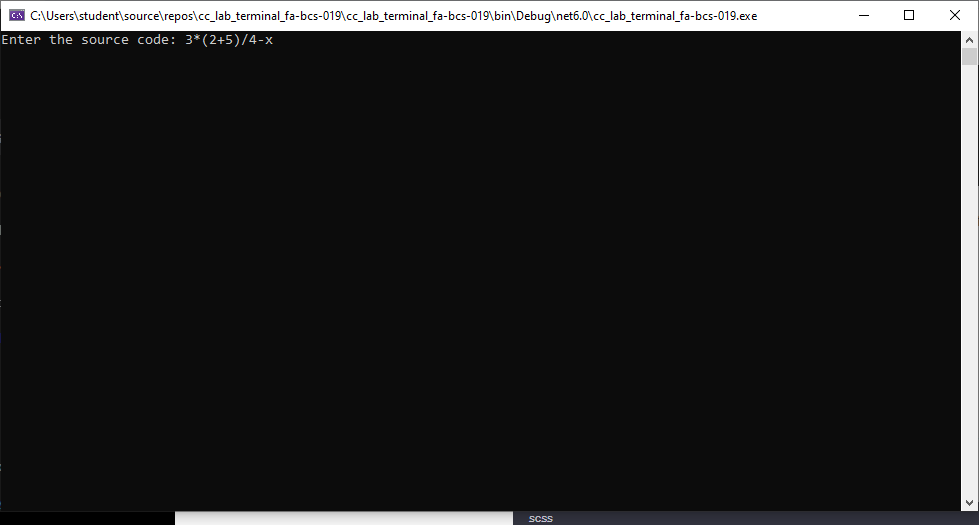
# Conclusion:

* The project successfully implements a basic compiler in C# covering lexical and syntax analysis, as well as simple code generation for mathematical expressions.
* The user-friendly interface allows users to input mathematical expressions and observe the corresponding intermediate code.
* The coding techniques used showcase the application of object-oriented principles and recursive descent parsing.

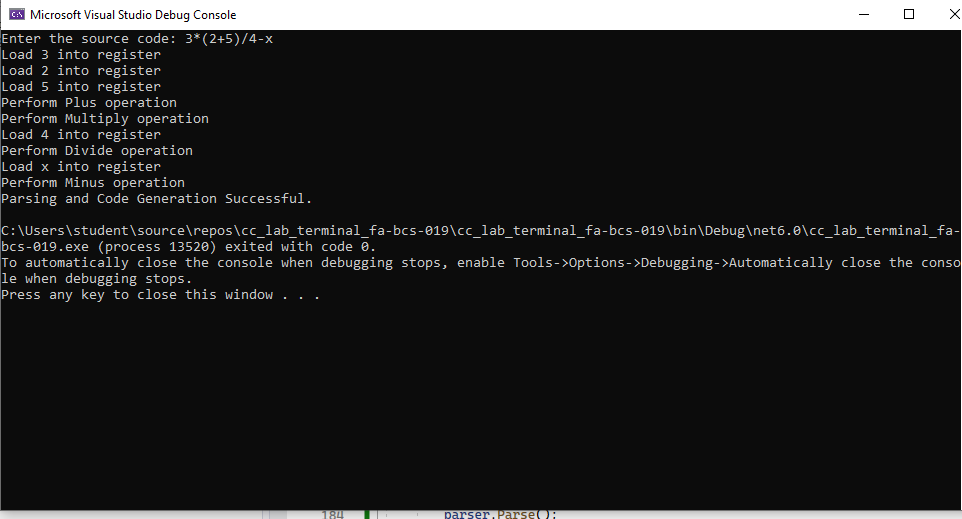
Future Improvements:

* Consider expanding language support to cover more complex expressions.
* Enhance error handling and reporting for better user experience.

INPUT:



OUTPUT:



### CODE:

using System;

// Lexical Analysis

public enum TokenType { Identifier, Number, Plus, Minus, Multiply, Divide, LeftParen, RightParen, EndOfFile, Error }

public class Token

{

public TokenType Type { get; }

public string Lexeme { get; }

// Additional properties for error handling

public char UnexpectedCharacter { get; }

public int Position { get; }

// Constructor for regular tokens

public Token(TokenType type, string lexeme)

{

Type = type;

Lexeme = lexeme;

}

// Constructor for error tokens

public Token(TokenType type, char unexpectedCharacter, int position)

{

Type = type;

UnexpectedCharacter = unexpectedCharacter;

Position = position;

}

}

public class Scanner

{

private readonly string input;

private int position = 0;

public Scanner(string input)

{

this.input = input;

}

public Token GetNextToken()

{

if (position >= input.Length)

return new Token(TokenType.EndOfFile, "");

char currentChar = input[position];

position++;

switch (currentChar)

{

case '+': return new Token(TokenType.Plus, "+");

case '-': return new Token(TokenType.Minus, "-");

case '\*': return new Token(TokenType.Multiply, "\*");

case '/': return new Token(TokenType.Divide, "/");

case '(': return new Token(TokenType.LeftParen, "(");

case ')': return new Token(TokenType.RightParen, ")");

default:

if (char.IsDigit(currentChar))

return ScanNumber(currentChar);

else if (char.IsLetter(currentChar))

return ScanIdentifier(currentChar);

else

return new Token(TokenType.Error, currentChar, position - 1);

}

}

private Token ScanNumber(char firstDigit)

{

string number = firstDigit.ToString();

while (position < input.Length && char.IsDigit(input[position]))

{

number += input[position];

position++;

}

return new Token(TokenType.Number, number);

}

private Token ScanIdentifier(char firstLetter)

{

string identifier = firstLetter.ToString();

while (position < input.Length && (char.IsLetterOrDigit(input[position]) || input[position] == '\_'))

{

identifier += input[position];

position++;

}

return new Token(TokenType.Identifier, identifier);

}

}

// Syntax Analysis and Code Generation

public class Parser

{

private readonly Scanner scanner;

private Token currentToken;

public Parser(Scanner scanner)

{

this.scanner = scanner;

currentToken = scanner.GetNextToken();

}

public void Parse()

{

Expression();

Console.WriteLine("Parsing and Code Generation Successful.");

}

private void Expression()

{

Term();

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

{

Token op = currentToken;

Match(op.Type);

Term();

// Generate Code

Console.WriteLine($"Perform {op.Type} operation");

}

}

private void Term()

{

Factor();

while (currentToken.Type == TokenType.Multiply || currentToken.Type == TokenType.Divide)

{

Token op = currentToken;

Match(op.Type);

Factor();

// Generate Code

Console.WriteLine($"Perform {op.Type} operation");

}

}

private void Factor()

{

if (currentToken.Type == TokenType.Number || currentToken.Type == TokenType.Identifier)

{

// Generate Code

Console.WriteLine($"Load {currentToken.Lexeme} into register");

Match(currentToken.Type);

}

else if (currentToken.Type == TokenType.LeftParen)

{

Match(TokenType.LeftParen);

Expression();

Match(TokenType.RightParen);

}

else if (currentToken.Type == TokenType.Error)

{

// Handle the error

Console.WriteLine($"Unexpected character '{currentToken.UnexpectedCharacter}' at position {currentToken.Position}");

// Move to the next token

currentToken = scanner.GetNextToken();

}

else

{

throw new Exception($"Unexpected token: {currentToken.Type}");

}

}

private void Match(TokenType expectedToken)

{

if (currentToken.Type == expectedToken)

currentToken = scanner.GetNextToken();

else

throw new Exception($"Expected {expectedToken}, but got {currentToken.Type}");

}

}

class Program

{

static void Main()

{

Console.Write("Enter the source code: ");

string sourceCode = Console.ReadLine();

Scanner scanner = new Scanner(sourceCode);

Parser parser = new Parser(scanner);

parser.Parse();

}

}