COMSATS UNIVERSITY ISLAMABAD , ATTOCK CAMPUS

CONSTRUCTION COMPILER TASK MINI COMPILER

GROUP MEMBERS: FAJAR AAMIR SHEIKH(SP22-BCS-031)

SARA ARSHAD(SP22-BCS-025)

SUBMITTED TO: SIR BILAL BUKHARI

SUBMISSION DATE: 30TH MAY2025

DEPARTMENT : COMPUTER SCIENCE

Description:

This mini compiler is designed to process and compile simple variable declarations such as:

int x = 5;

It performs the main phases of compilation, typically found in a real-world compiler, in the following order:

1. Lexical Analysis

Purpose: Breaks the input code into tokens (basic units like keywords, identifiers, operators, numbers).

Component:Lexer class.

Example Token Types:

Keyword (e.g., int)

Identifier (e.g., x)

Number (e.g., 5)

Operator (e.g., = or ;)

2. Syntax Analysis

Purpose:Checks if the sequence of tokens follows the correct grammar or structure of the language.

Component:Parser class.

Example Rule:

Must match pattern: int <identifier> = <number>;

3. Semantic Analysis

Purpose: Validates the meaning of the code (e.g., variable is declared correctly and types match).

Component: SymbolTable class.

Checks:

Variable is not redeclared.

Variable types are consistent.

4. Optimization

Purpose: Improves the code by simplifying expressions (e.g., folding constant expressions).

Component:Optimizer class.

Example:

Replaces 2 + 3 with 5.

5. Intermediate Code Generation (IR)

Purpose: Translates code into a simpler intermediate representation for easier processing.

Component: IRGenerator class.

Example IR:

t1 = 5

x = t1

6. Target Code Generation

Purpose:Produces low-level code similar to machine instructions.

Component:TargetCodeGenerator class.

Example Output:

LOAD 5

STORE x

7. Output Presentation

Purpose:Displays results of each compilation phase in a formatted console output using boxes.

Component:PrintBox method.

Execution Flow (Main Program)

1. Takes 5 lines of code as input from the user.

2. For each line, it executes all 7 compiler phases.

3. Displays each step with results or errors.

CODE:

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

// Token types
public enum TokenType
{
   Keyword, Identifier, Operator, Number, Semicolon
}

// Token class
public class Token
{
   public TokenType Type;
   public string Value;

   public Token(TokenType type, string value)
   {
      Type = type;
      Value = value;
   }

   public override string ToString() => $"[{Type}: {Value}]";
}

// 1. Lexical Analysis
public class Lexer
{
   public List<Token> Tokenize(string input)
   {
      var tokens = new List<Token>();
      var regex = new Regex(@"\s*(int|[a-zA-Z_]\w*|=|\d+|;)\s*");

      var matches = regex.Matches(input);
      foreach (Match match in matches)
      {
         string value = match.Groups[1].Value;
```

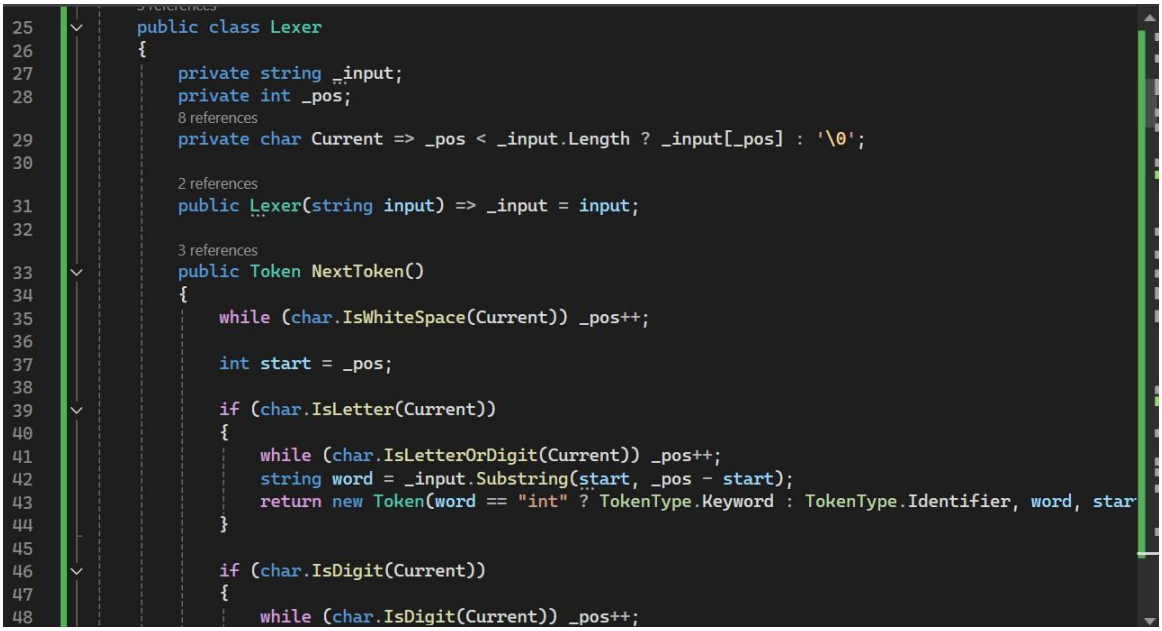
```
        if (value == "int") tokens.Add(new Token(TokenType.Keyword, value));
        else if (Regex.IsMatch(value, @"^\d+$")) tokens.Add(new
Token(TokenType.Number, value));
        else if (value == "=") tokens.Add(new Token(TokenType.Operator, value));
        else if (value == ";") tokens.Add(new Token(TokenType.Semicolon, value));
        else tokens.Add(new Token(TokenType.Identifier, value));
      }

      return tokens;
    }
}

// 2. Syntax Analysis
public class Parser
{
    public bool Parse(List<Token> tokens)
    {
      // Expected pattern: int <id> = <number> ;
      if (tokens.Count != 5) return false;
      return tokens[0].Type == TokenType.Keyword &&
          tokens[1].Type == TokenType.Identifier &&
          tokens[2].Type == TokenType.Operator &&
          tokens[3].Type == TokenType.Number &&
          tokens[4].Type == TokenType.Semicolon;
    }
}

// 3. Semantic Analysis
public class SymbolTable
{
    private Dictionary<string, int> table = new Dictionary<string, int>();

    public bool AddVariable(string name, int value)
    {
      if (table.ContainsKey(name))
      {
        Console.WriteLine($"Semantic Error: Variable '{name}' already declared.");
        return false;
      }

      table[name] = value;
```

```
        return true;
    }

    public void PrintTable()
    {
        Console.WriteLine("\nSymbol Table:");
        foreach (var kv in table)
            Console.WriteLine($"  {kv.Key} = {kv.Value}");
    }
}

// 4. Optimizer
public class Optimizer
{
    public int Optimize(string expression)
    {
        // Simple constant folding
        return int.Parse(expression);
    }
}

// 5. Intermediate Code Generator
public class IRGenerator
{
    public void Generate(string var, int value)
    {
        Console.WriteLine($"\nIntermediate Code:\nt1 = {value}\n{var} = t1");
    }
}

// 6. Target Code Generator
public class TargetCodeGenerator
{
    public void Generate(string var, int value)
    {
        Console.WriteLine($"\nTarget Code:\nLOAD {value}\nSTORE {var}");
    }
}

// 7. Output Presenter
public class Printer
```

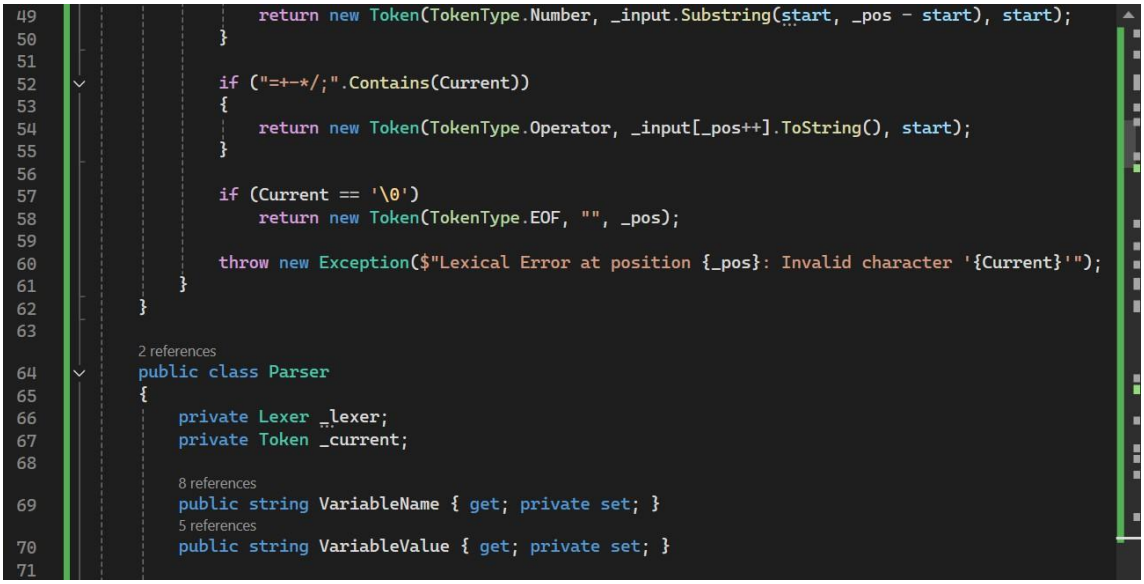
```
{
    public static void PrintBox(string title, List<Token> tokens)
    {
        Console.WriteLine($"\n--- {title} ---");
        foreach (var token in tokens)
            Console.WriteLine(token);
    }
}

// Main compiler driver
public class MiniCompiler
{
    static void Main()
    {
        var lexer = new Lexer();
        var parser = new Parser();
        var semantic = new SymbolTable();
        var optimizer = new Optimizer();
        var ir = new IRGenerator();
        var target = new TargetCodeGenerator();

        Console.WriteLine("Enter up to 5 lines of code (e.g., int x = 5;):");
        for (int i = 0; i < 5; i++)
        {
            Console.Write($"\nLine {i + 1}: ");
            string line = Console.ReadLine();
            if (string.IsNullOrWhiteSpace(line)) break;

            var tokens = lexer.Tokenize(line);
            Printer.PrintBox("Tokens", tokens);

            if (!parser.Parse(tokens))
            {
                Console.WriteLine("Syntax Error: Invalid statement structure.");
                continue;
            }

            string varName = tokens[1].Value;
            string valueStr = tokens[3].Value;
            int value = optimizer.Optimize(valueStr);
```

```
        if (!semantic.AddVariable(varName, value)) continue;

        ir.Generate(varName, value);
        target.Generate(varName, value);
    }

    semantic.PrintTable();
  }
}
```

CODE SCREENSHOT:



```
public class Lexer
{
    private string _input;
    private int _pos;
    8 references
    private char Current => _pos < _input.Length ? _input[_pos] : '\0';

    2 references
    public Lexer(string input) => _input = input;

    3 references
    public Token NextToken()
    {
        while (char.IsWhiteSpace(Current)) _pos++;

        int start = _pos;

        if (char.IsLetter(Current))
        {
            while (char.IsLetterOrDigit(Current)) _pos++;
            string word = _input.Substring(start, _pos - start);
            return new Token(word == "int" ? TokenType.Keyword : TokenType.Identifier, word, star
        }

        if (char.IsDigit(Current))
        {
            while (char.IsDigit(Current)) _pos++;
```



```
            return new Token(TokenType.Number, _input.Substring(start, _pos - start), start);
        }

        if ("=+-*/;".Contains(Current))
        {
            return new Token(TokenType.Operator, _input[_pos++].ToString(), start);
        }

        if (Current == '\0')
            return new Token(TokenType.EOF, "", _pos);

        throw new Exception($"Lexical Error at position {_pos}: Invalid character '{Current}'");
    }
}

2 references
public class Parser
{
    private Lexer _lexer;
    private Token _current;

    8 references
    public string VariableName { get; private set; }
    5 references
    public string VariableValue { get; private set; }
```

```
72      public Parser(Lexer lexer)
73      {
74          _lexer = lexer;
75          _current = _lexer.NextToken();
76      }
77
78      private void Eat(TokenType type)
79      {
80          if (_current.Type == type)
81              _current = _lexer.NextToken();
82          else
83              throw new Exception($"Syntax Error at position {_current.Position}: Expected {type},
84      }
85
86      public void ParseAssignment()
87      {
88          Eat(TokenType.Keyword);                // int
89          VariableName = _current.Value;
90          Eat(TokenType.Identifier);             // x
91          Eat(TokenType.Operator);               // =
92          VariableValue = _current.Value;
93          Eat(TokenType.Number);                 // 5
94          Eat(TokenType.Operator);               // ;
95      }
96  }
```

```
98  public class SymbolTable
99  {
100     private Dictionary<string, string> _table = new();
101
102     public void Declare(string name, string type)
103     {
104         if (_table.ContainsKey(name))
105             throw new Exception($"Semantic Error: Variable '{name}' already declared.");
106         _table[name] = type;
107     }
108
109     public void Check(string name, string expectedType)
110     {
111         if (!_table.ContainsKey(name))
112             throw new Exception($"Semantic Error: Variable '{name}' not declared.");
113         if (_table[name] != expectedType)
114             throw new Exception($"Semantic Error: Type mismatch for variable '{name}'");
115     }
116 }
117
118 public class Optimizer
119 {
120     public string ConstantFold(string expr)
```

```
        {
            if (expr == "2 + 3") return "5"; // Example folding
            return expr;
        }
    }

    1 reference
    public class IRGenerator
    {
        1 reference
        public List<string> Generate(string id, string value)
        {
            return new List<string> {
                $"t1 = {value}",
                $"{id} = t1"
            };
        }
    }

    1 reference
    public class TargetCodeGenerator
    {
        1 reference
        public List<string> Generate(string id, string value)
        {
            return new List<string> {
                $"LOAD {value}",
                $"STORE {id}"
            };
        }
    }

    0 references
    class Program
    {
        // Helper to print box with title and content lines
        7 references
        static void PrintBox(string title, List<string> lines)
        {
            int maxLength = title.Length;
            foreach (var line in lines)
                if (line.Length > maxLength) maxLength = line.Length;

            int width = maxLength + 4; // padding + borders

            string border = "+" + new string('-', width - 2) + "+";

            Console.WriteLine(border);
            Console.WriteLine("| " + title.PadRight(width - 4) + " |");
            Console.WriteLine(border);
            foreach (var line in lines)
            {
                Console.WriteLine("| " + line.PadRight(width - 4) + " |");
            }
            Console.WriteLine(border);
```

```
170         }
171
            0 references
172         static void Main()
173         {
174             var symbolTable = new SymbolTable();
175
176             Console.WriteLine("Mini Compiler - Enter 5 lines of code (e.g. 'int x = 5;'):");
177             List<string> codeLines = new();
178
179             for (int i = 0; i < 5; i++)
180             {
181                 Console.Write($"Line {i + 1}: ");
182                 string line = Console.ReadLine();
183                 codeLines.Add(line);
184             }
185
186             Console.WriteLine("\n--- Compilation Start ---\n");
187
188             int lineNumber = 1;
189             foreach (var code in codeLines)
190             {
191                 Console.WriteLine($"########## Line {lineNumber} ##########");
192                 Console.WriteLine($"[Input] {code}\n");
193
194                 try
195                 {
```

```
196                     // Phase 1: Lexical Analysis
197                     var lexer = new Lexer(code);
198                     List<string> tokensOutput = new();
199                     Token token;
200                     while ((token = lexer.NextToken()).Type != TokenType.EOF)
201                     {
202                         tokensOutput.Add(token.ToString());
203                     }
204                     PrintBox("Lexical Analysis", tokensOutput);
205
206                     // Reinit lexer for parsing
207                     lexer = new Lexer(code);
208
209                     // Phase 2: Syntax Analysis
210                     var parser = new Parser(lexer);
211                     parser.ParseAssignment();
212                     PrintBox("Syntax Analysis", new List<string> { $"Parsed assignment: int {parser.V
213
214                     // Phase 3 & 7: Semantic Analysis + Symbol Table
215                     symbolTable.Declare(parser.VariableName, "int");
216                     symbolTable.Check(parser.VariableName, "int");
217                     PrintBox("Semantic Analysis & Symbol Table", new List<string> {
218                         $"Variable '{parser.VariableName}' declared as 'int'",
219                         $"Type check passed for '{parser.VariableName}'"
220                     });
221
222                     // Phase 4: Optimization
```

```
                    var optimizer = new Optimizer();
                    string optimizedValue = optimizer.ConstantFold(parser.VariableValue);
                    List<string> optimizationOutput = new();
                    if (optimizedValue != parser.VariableValue)
                        optimizationOutput.Add($"Constant folded '{parser.VariableValue}' → '{optimiz
                    else
                        optimizationOutput.Add("No optimization applied");
                    PrintBox("Optimization", optimizationOutput);

                    // Phase 5: Intermediate Code Generation
                    var irGen = new IRGenerator();
                    var irLines = irGen.Generate(parser.VariableName, optimizedValue);
                    PrintBox("Intermediate Code Generation", irLines);

                    // Phase 6: Target Code Generation
                    var targetGen = new TargetCodeGenerator();
                    var targetLines = targetGen.Generate(parser.VariableName, optimizedValue);
                    PrintBox("Target Code Generation", targetLines);

                    Console.WriteLine("\nLine compiled successfully ☑\n");
                }
                catch (Exception ex)
                {
                    PrintBox("Compilation Error", new List<string> { ex.Message });
                    Console.WriteLine();
                }
```

```
                lineNumber++;
            }

            Console.WriteLine("--- Compilation Finished ---");
        }
    }
}
```

output:

```
Mini Compiler - Enter 5 lines of code (e.g. 'int x = 5;'):
Line 1: int x = 5;
Line 2: int y = 2 + 3;
Line 3:  int x = 10;
Line 4: int z = a;
Line 5: int total = 10;

--- Compilation Start ---

########## Line 1 ##########
[Input] int x = 5;

+------------------+
| Lexical Analysis |
+------------------+
| Keyword: int     |
| Identifier: x    |
| Operator: =      |
| Number: 5        |
| Operator: ;      |
+------------------+
+-------------------------------+
| Syntax Analysis               |
+-------------------------------+
| Parsed assignment: int x = 5; |
+-------------------------------+
+---------------------------------+
| Semantic Analysis & Symbol Table |
+---------------------------------+
| Variable 'x' declared as 'int'   |
| Type check passed for 'x'        |
+---------------------------------+
+-------------------------+
```

```
+-------------------------+
| Optimization            |
+-------------------------+
| No optimization applied |
+-------------------------+
+------------------------------+
| Intermediate Code Generation |
+------------------------------+
| t1 = 5                       |
| x = t1                       |
+------------------------------+
+-------------------------+
| Target Code Generation |
+-------------------------+
| LOAD 5                  |
| STORE x                 |
+-------------------------+

Line compiled successfully ?

########## Line 2 ##########
[Input] int y = 2 + 3;

+------------------+
| Lexical Analysis |
+------------------+
| Keyword: int     |
| Identifier: y    |
| Operator: =      |
| Number: 2        |
| Operator: +      |
| Number: 3        |
| Operator: ;      |
```

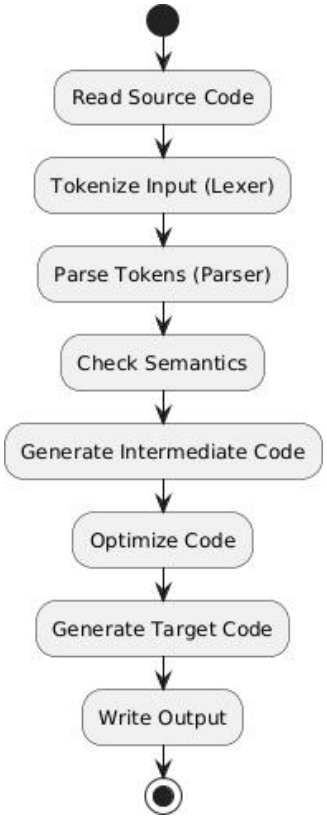
```
+------------------+
+--------------------------------+
| Syntax Analysis                |
+--------------------------------+
| Parsed assignment: int y = 2; |
+--------------------------------+
+----------------------------------+
| Semantic Analysis & Symbol Table |
+----------------------------------+
| Variable 'y' declared as 'int'   |
| Type check passed for 'y'        |
+----------------------------------+
+-------------------------+
| Optimization            |
+-------------------------+
| No optimization applied |
+-------------------------+
+------------------------------+
| Intermediate Code Generation |
+------------------------------+
| t1 = 2                       |
| y = t1                       |
+------------------------------+
+-------------------------+
| Target Code Generation |
+-------------------------+
| LOAD 2                  |
| STORE y                 |
+-------------------------+

Line compiled successfully ?

########## Line 3 ##########
```

```
[Input]  int x = 10;

+-----------------+
| Lexical Analysis |
+-----------------+
| Keyword: int     |
| Identifier: x    |
| Operator: =      |
| Number: 10       |
| Operator: ;      |
+-----------------+
+-------------------------------+
| Syntax Analysis               |
+-------------------------------+
| Parsed assignment: int x = 10; |
+-------------------------------+
+-----------------------------------------------+
| Compilation Error                             |
+-----------------------------------------------+
| Semantic Error: Variable 'x' already declared. |
+-----------------------------------------------+

########## Line 4 ##########
[Input] int z = a;

+-----------------+
| Lexical Analysis |
+-----------------+
| Keyword: int     |
| Identifier: z    |
| Operator: =      |
| Identifier: a    |
| Operator: ;      |
```

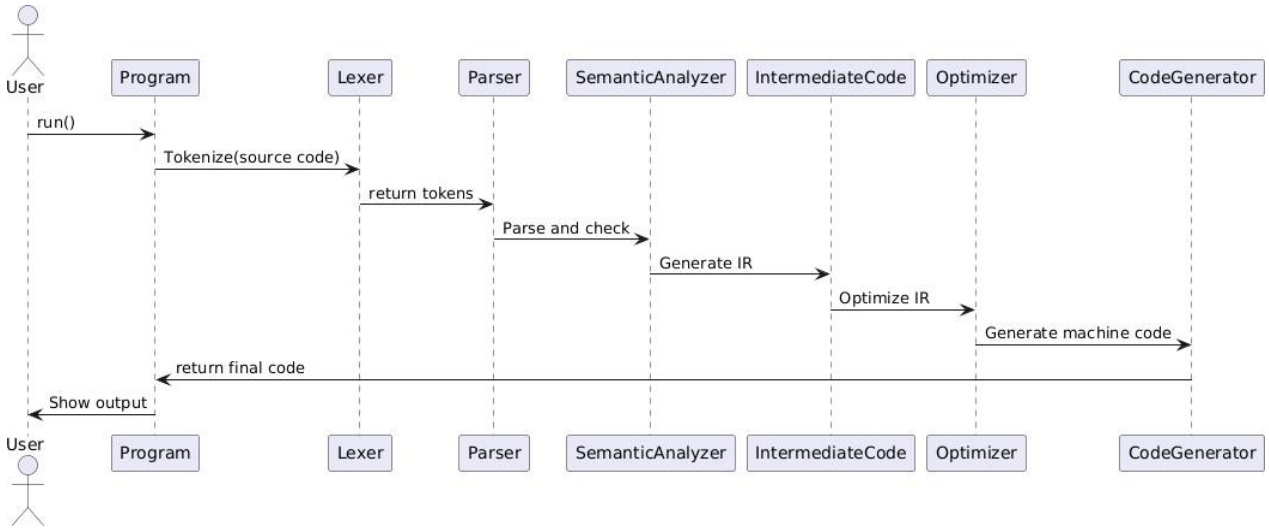
```
+------------------+
+-------------------------------------------------------------+
| Compilation Error                                           |
+-------------------------------------------------------------+
| Syntax Error at position 8: Expected Number, got Identifier |
+-------------------------------------------------------------+

########## Line 5 ##########
[Input] int total = 10;

+-------------------+
| Lexical Analysis  |
+-------------------+
| Keyword: int      |
| Identifier: total |
| Operator: =       |
| Number: 10        |
| Operator: ;       |
+-------------------+
+-----------------------------------+
| Syntax Analysis                   |
+-----------------------------------+
| Parsed assignment: int total = 10; |
+-----------------------------------+
+-----------------------------------+
| Semantic Analysis & Symbol Table  |
+-----------------------------------+
| Variable 'total' declared as 'int' |
| Type check passed for 'total'     |
+-----------------------------------+
+-------------------------+
| Optimization            |
+-------------------------+
```

```
+-----------------------------------+
| Semantic Analysis & Symbol Table  |
+-----------------------------------+
| Variable 'total' declared as 'int' |
| Type check passed for 'total'     |
+-----------------------------------+
+-------------------------+
| Optimization            |
+-------------------------+
| No optimization applied |
+-------------------------+
+-------------------------------+
| Intermediate Code Generation |
+-------------------------------+
| t1 = 10                       |
| total = t1                    |
+-------------------------------+
+-------------------------+
| Target Code Generation |
+-------------------------+
| LOAD 10                 |
| STORE total             |
+-------------------------+

Line compiled successfully ?

--- Compilation Finished ---

C:\Users\HP\source\repos\ConsoleApp3\minicompiler2\minicompiler2\bin\Debug\net8.0\minicompiler2.exe (process 1344) exited with code 0
 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debug
ging stops.
Press any key to close this window . . .
```

Activitydiagram:



Sequencediagram:



ClassDiagram: