

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: 30THMAY2025

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:

```
25 public class Lexer
26 {
27     private string _input;
28     private int _pos;
29     private char Current => _pos < _input.Length ? _input[_pos] : '\0';
30
31     public Lexer(string input) => _input = input;
32
33     public Token NextToken()
34     {
35         while (char.IsWhiteSpace(Current)) _pos++;
36
37         int start = _pos;
38
39         if (char.IsLetter(Current))
40         {
41             while (char.IsLetterOrDigit(Current)) _pos++;
42             string word = _input.Substring(start, _pos - start);
43             return new Token(word == "int" ? TokenType.Keyword : TokenType.Identifier, word, start);
44         }
45
46         if (char.IsDigit(Current))
47         {
48             while (char.IsDigit(Current)) _pos++;
49
50             return new Token(TokenType.Number, _input.Substring(start, _pos - start), start);
51         }
52
53         if ("=+-*/;".Contains(Current))
54         {
55             return new Token(TokenType.Operator, _input[_pos++].ToString(), start);
56         }
57
58         if (Current == '\0')
59             return new Token(TokenType.EOF, "", _pos);
60
61         throw new Exception($"Lexical Error at position {_pos}: Invalid character '{Current}'");
62     }
63
64     public class Parser
65     {
66         private Lexer _lexer;
67         private Token _current;
68
69         public string VariableName { get; private set; }
70         public string VariableValue { get; private set; }
71     }
}
```

```

72 1 reference
73 public Parser(Lexer lexer)
74 {
75     _lexer = lexer;
76     _current = _lexer.NextToken();
77 }
78 5 references
79 private void Eat(TokenType type)
80 {
81     if (_current.Type == type)
82         _current = _lexer.NextToken();
83     else
84         throw new Exception($"Syntax Error at position {_current.Position}: Expected {type}, ");
85 }
86 1 reference
87 public void ParseAssignment()
88 {
89     Eat(TokenType.Keyword);           // int
90     VariableName = _current.Value;
91     Eat(TokenType.Identifier);        // x
92     Eat(TokenType.Operator);         // =
93     VariableValue = _current.Value;
94     Eat(TokenType.Number);           // 5
95     Eat(TokenType.Operator);         // ;
96 }

```

```

98 public class SymbolTable
99 {
100     private Dictionary<string, string> _table = new();
101
102     1 reference
103     public void Declare(string name, string type)
104     {
105         if (_table.ContainsKey(name))
106             throw new Exception($"Semantic Error: Variable '{name}' already declared.");
107         _table[name] = type;
108     }
109
110     1 reference
111     public void Check(string name, string expectedType)
112     {
113         if (!_table.ContainsKey(name))
114             throw new Exception($"Semantic Error: Variable '{name}' not declared.");
115         if (_table[name] != expectedType)
116             throw new Exception($"Semantic Error: Type mismatch for variable '{name}'");
117     }
118
119     1 reference
120     public class Optimizer
121     {
122         1 reference
123         public string ConstantFold(string expr)

```

```

121     {
122         if (expr == "2 + 3") return "5"; // Example folding
123         return expr;
124     }
125 }
126
127 1 reference
128 public class IRGenerator
129 {
130     1 reference
131     public List<string> Generate(string id, string value)
132     {
133         return new List<string> {
134             $"t1 = {value}",
135             $"id = t1"
136         };
137     }
138
139     1 reference
140     public class TargetCodeGenerator
141     {
142         1 reference
143         public List<string> Generate(string id, string value)
144         {
145             return new List<string> {
146                 $"LOAD {value}",
147                 $"STORE {id}"
148             };
149         }
150     }
151 }

```

```

145     };
146 }
147 }
148
149 0 references
150 class Program
151 {
152     // Helper to print box with title and content lines
153     7 references
154     static void PrintBox(string title, List<string> lines)
155     {
156         int maxLength = title.Length;
157         foreach (var line in lines)
158             if (line.Length > maxLength) maxLength = line.Length;
159
160         int width = maxLength + 4; // padding + borders
161
162         string border = "+" + new string('-', width - 2) + "+";
163
164         Console.WriteLine(border);
165         Console.WriteLine("| " + title.PadRight(width - 4) + " |");
166         Console.WriteLine(border);
167         foreach (var line in lines)
168         {
169             Console.WriteLine("| " + line.PadRight(width - 4) + " |");
170         }
171         Console.WriteLine(border);
172     }
173 }

```

```

170     }
171
172     0 references
173     static void Main()
174     {
175         var symbolTable = new SymbolTable();
176
177         Console.WriteLine("Mini Compiler - Enter 5 lines of code (e.g. 'int x = 5;')");
178         List<string> codeLines = new();
179
180         for (int i = 0; i < 5; i++)
181         {
182             Console.Write($"Line {i + 1}: ");
183             string line = Console.ReadLine();
184             codeLines.Add(line);
185         }
186
187         Console.WriteLine("\n--- Compilation Start ---\n");
188
189         int lineNumber = 1;
190         foreach (var code in codeLines)
191         {
192             Console.WriteLine($"##### Line {lineNumber} #####");
193             Console.WriteLine($"[Input] {code}\n");
194
195             try
196             {

```

```

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

```

```

223     var optimizer = new Optimizer();
224     string optimizedValue = optimizer.ConstantFold(parser.VariableValue);
225     List<string> optimizationOutput = new();
226     if (optimizedValue != parser.VariableValue)
227         optimizationOutput.Add($"Constant folded '{parser.VariableValue}' → '{optimizedValue}'");
228     else
229         optimizationOutput.Add("No optimization applied");
230     PrintBox("Optimization", optimizationOutput);
231
232     // Phase 5: Intermediate Code Generation
233     var irGen = new IRGenerator();
234     var irLines = irGen.Generate(parser.VariableName, optimizedValue);
235     PrintBox("Intermediate Code Generation", irLines);
236
237     // Phase 6: Target Code Generation
238     var targetGen = new TargetCodeGenerator();
239     var targetLines = targetGen.Generate(parser.VariableName, optimizedValue);
240     PrintBox("Target Code Generation", targetLines);
241
242     Console.WriteLine($"\\nLine compiled successfully ☑\\n");
243 }
244 catch (Exception ex)
245 {
246     PrintBox("Compilation Error", new List<string> { ex.Message });
247     Console.WriteLine();
248 }

```

```

249
250     lineNumber++;
251 }
252
253 Console.WriteLine("--- Compilation Finished ---");
254 }
255 }
256 }
257

```

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 |
+-----+

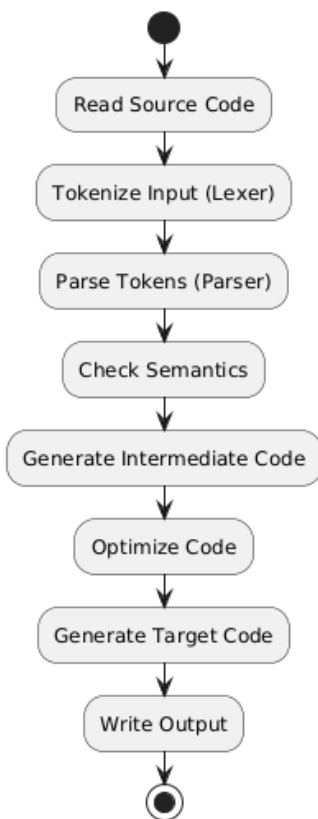
Line compiled successfully ?

--- Compilation Finished ---

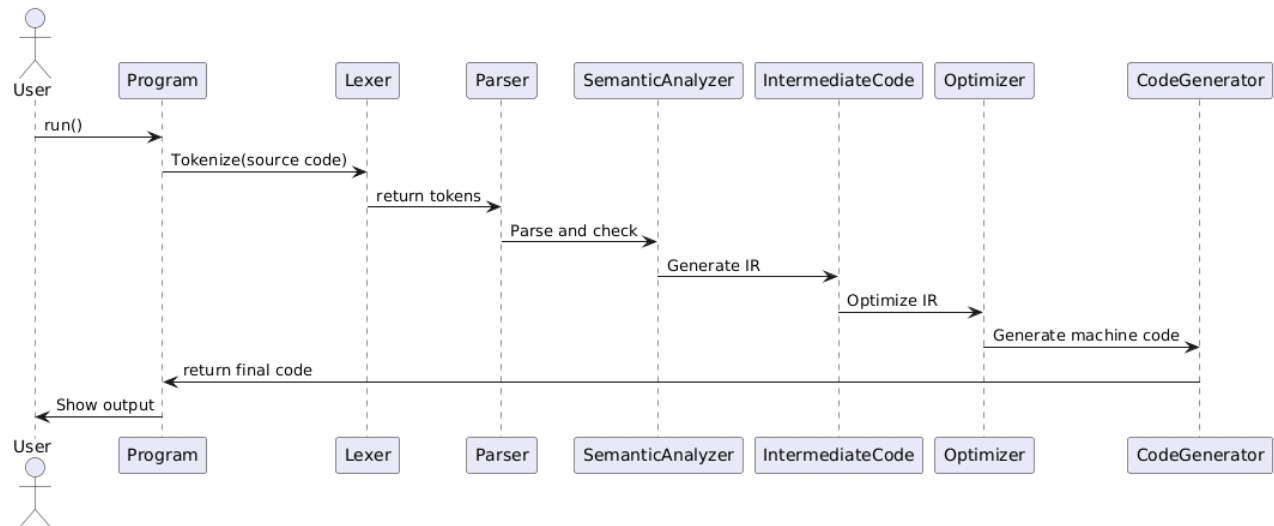
C:\Users\HP\source\repos\ConsoleApp3\minicompiler2\minicompiler2\bin\Debug\net8.0\minicompiler2.exe (process 1344) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to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

```

Activitydiagram:



Sequencediagram:



ClassDiagram:

Class Diagram

