COMSATS University Islamabad

Attock Campus

Semester Project
(Mini Compiler)

Group Member:

Muhammad Anas (Sp22-Bcs-042)

Moazzam Azam (Sp22-Bcs-010)

Submitted To:

Sir Bilal Haider

Subject:

Compiler Construction

Date: 30th May,2025

GitHub Repository Link

https://github.com/Muhmmad-Anas/Mini-Compiler.git

Overview:

Mini Compiler Pro is a complete compiler implementation written in C# that demonstrates all phases of compilation from source code to assembly generation. It features a professional Windows Forms GUI and supports a simple programming language with variables, arithmetic operations, conditional statements, and loops.

Key Features

- Complete Compilation Pipeline: Lexical Analysis → Parsing → Semantic Analysis → IR Generation → Optimization → Code Generation
- Professional GUI: Tabbed interface showing each compilation phase
- Error Handling: Comprehensive error reporting with line/column information
- Symbol Table Management: Variable declaration and usage tracking
- Code Optimization: Constant folding and dead code elimination
- Assembly Generation: Target code generation in assembly format.

Architecture:

The compiler follows the traditional multi-pass architecture: Source Code → Lexer → Parser → Semantic Analyzer → IR Generator → Optimizer → Code Generator → Assembly

Design Patterns Used:

- Visitor Pattern: For AST traversal (IASTVisitor)
- Composite Pattern: For AST node hierarchy
- Strategy Pattern: For different compilation phases
- Observer Pattern: For GUI updates

Language Grammar:

The Mini Compiler supports a simple C-like language with the following grammar:

Tokens

Keywords:
if, else, while, for, return, int, float, string, bool, true, false, function

Operators:
+, -, *, /, %, =, ==, !=, <, >, <=, >=, &&, ||

Delimiters:
(,), {, }, ;, ,

Literals:
numbers, floating-point numbers, string literals

Identifiers:
variable and function names

Grammar Rules

Program → Statement*
Statement → Assignment | IfStatement | WhileStatement | Block
Assignment → IDENTIFIER '=' Expression ';'
IfStatement → 'if' '(' Expression ')' Statement ('else' Statement)?
WhileStatement → 'while' '(' Expression ')' Statement
Block → '{' Statement* '}'
Expression → Comparison
Comparison → Term (('==' | '!=' | '<' | '>' | '<=' | '>=') Term)*
Term → Factor (('+' | '-') Factor)*
Factor → Primary (('*' | '/') Primary)*
Primary → NUMBER | IDENTIFIER | '(' Expression ')'

Core Components

1. Token Class

Represents individual tokens with position information:

```
public class Token
{
   public string Type { get; set; }      // Token type
(IDENTIFIER, NUMBER, etc.)
```

```
    public string Value { get; set; }     // Token value
    public int Line { get; set; }         // Line number
    public int Column { get; set; }       // Column position
}
```

2. Lexer (Lexical Analyzer)

- Purpose: Converts source code into a stream of tokens
- Features:
  - Regular expression-based tokenization
  - Line/column tracking for error reporting
  - Support for keywords, operators, literals, and identifiers
  - Comment and whitespace handling

Key Methods:

- Tokenize(): Main tokenization method
- Pattern matching using Dictionary<string, string> TokenPatterns

3. AST (Abstract Syntax Tree) Nodes

Hierarchical representation of the program structure:

```
// Base class
public abstract class ASTNode
{
    public abstract string Accept(IASTVisitor visitor);
}
```

// Node types:

- ProgramNode: Root of the AST
- AssignmentNode: Variable assignments
- BinaryOpNode: Binary operations (+, -, *, /, comparisons)
- NumberNode: Numeric literals
- IdentifierNode: Variable references
- IfNode: Conditional statements
- WhileNode: Loop statements
- BlockNode: Code blocks

4. Parser (Syntax Analyzer)

- Purpose: Builds AST from token stream
- Method: Recursive descent parsing

- Features:
    - Operator precedence handling
    - Left-associative operators
    - Error recovery and reporting

Key Methods:

- Parse(): Entry point
- ParseStatement(), ParseExpression(), etc.: Grammar rule implementations

5. Symbol Table

Manages variable declarations and type information:

```
public class SymbolInfo
{
   public string Name { get; set; }
   public string Type { get; set; }
   public int Line { get; set; }
   public bool IsInitialized { get; set; }
}
```

6. Semantic Analyzer

- Purpose: Type checking and semantic validation
- Features:
    - Variable declaration checking
    - Usage before initialization detection
    - Type compatibility verification

7. IR Generator (Intermediate Representation)

Generates three-address code:

```
public class ThreeAddressCode
{
   public string Operator { get; set; }   // Operation (+, -, =, etc.)
   public string Operand1 { get; set; }   // First operand
   public string Operand2 { get; set; }   // Second operand
   public string Result { get; set; }     // Result variable
}
```

8. Optimizer

Performs code optimizations:

- Constant Folding: Evaluates constant expressions at compile time

9. Code Generator

Generates target assembly code from optimized IR:

- Supports basic instruction set (MOV, ADD, SUB, MUL, DIV, JMP, etc.)
- Label generation for control flow
- Register allocation simulation

Compilation Pipeline

Phase 1: Lexical Analysis
Input: Source code string
Process: Tokenization using regex patterns
Output: List<Token>

Phase 2: Syntax Analysis (Parsing)
Input: List<Token>
Process: Recursive descent parsing
Output: AST (ProgramNode)

Phase 3: Semantic Analysis
Input: AST
Process: Symbol table construction, type checking
Output: Error list, Symbol table

Phase 4: IR Generation
Input: AST
Process: AST traversal with visitor pattern
Output: List<ThreeAddressCode>

Phase 5: Optimization
Input: IR code
Process: Constant folding, dead code elimination
Output: Optimized IR code

Phase 6: Code Generation
Input: Optimized IR
Process: Assembly instruction generation
Output: List<string> (assembly code)

User Interface

Main Window Components

1. **Menu Bar**
   - File: New, Load Sample, Exit
   - Help: About
2. **Source Code Panel**
   - Multi-line text editor with syntax highlighting support
   - Consolas font for better code readability
3. **Control Buttons**
   - Compile: Execute full compilation pipeline
   - Clear: Clear all panels
   - Load Sample: Load example program
4. **Results Tabs**
   - 🔤 Tokens: Lexical analysis results
   - 🌳 AST: Abstract syntax tree visualization
   - 🧠 Semantic: Semantic analysis results
   - 💾 IR: Intermediate representation
   - ⚡ Optimized IR: Optimized intermediate code
   - 🛠 Assembly: Generated assembly code
   - ❌ Errors: Compilation errors
   - 📋 Symbol Table: Variable information
5. **Status Bar**
   - Shows current compilation status

Usage Guide

Getting Started

1. **Launch the Application**
   - Run the executable or compile from Visual Studio
   - The main window will appear with an empty source code editor
2. **Write Code**
   - Enter your program in the source code panel
   - Use the supported language syntax
3. **Compile**
   - Click the "Compile" button
   - Check each tab to see the compilation results

Technical Implementation

Error Handling Strategy

- Lexical Errors: Unknown characters, invalid tokens
- Syntax Errors: Unexpected tokens, missing semicolons
- Semantic Errors: Undeclared variables, type mismatches
- Custom Exception: CompilerException with detailed messages

Memory Management

- Efficient token storage
- AST node lifecycle management
- String interning for identifiers

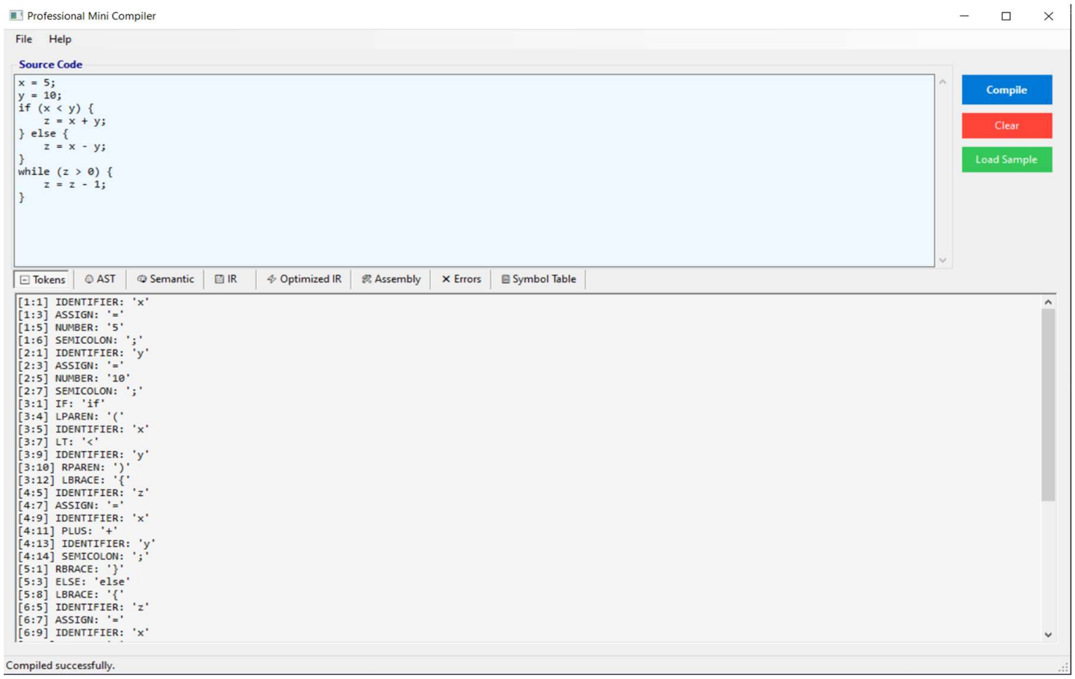
Performance Considerations

- Single-pass lexing
- Recursive descent parsing with minimal backtracking
- Efficient symbol table lookups using Dictionary
- Lazy evaluation in optimization passes

Extensibility Points

1. New Language Features: Add tokens, AST nodes, parser rules
2. Additional Optimizations: Extend Optimizer class
3. Different Target Architectures: Modify CodeGenerator
4. Enhanced UI: Add more visualization tabs

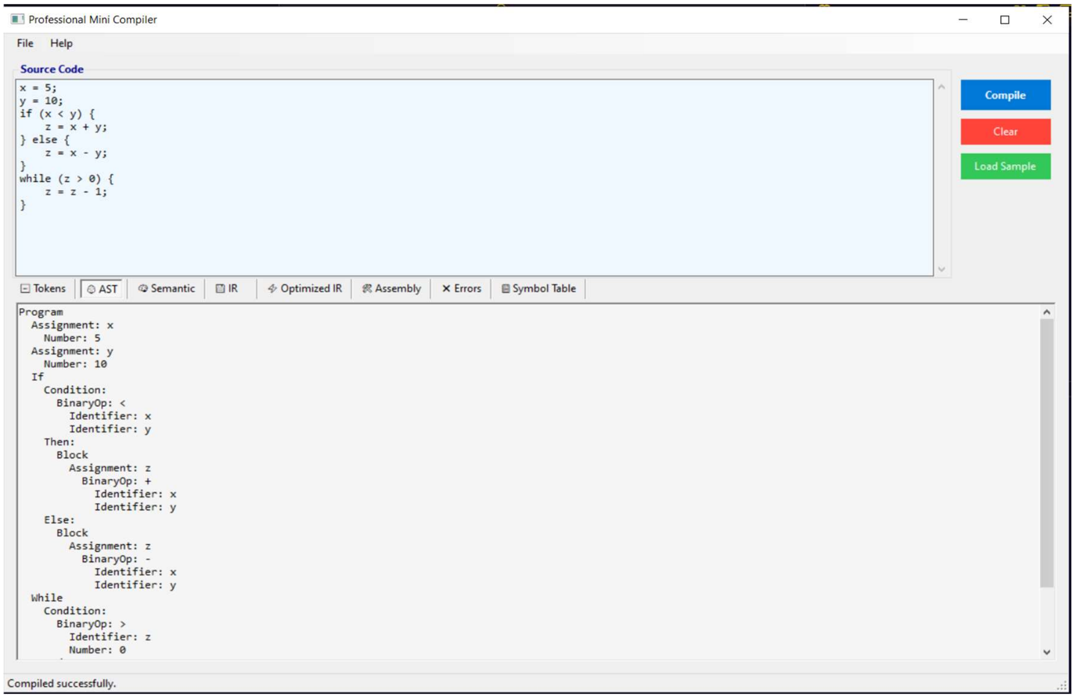
Output:



```
x = 5;
y = 10;
if (x < y) {
    z = x + y;
} else {
    z = x - y;
}
while (z > 0) {
    z = z - 1;
}
```

```
[1:1] IDENTIFIER: 'x'
[1:3] ASSIGN: '='
[1:5] NUMBER: '5'
[1:6] SEMICOLON: ';'
[2:1] IDENTIFIER: 'y'
[2:3] ASSIGN: '='
[2:5] NUMBER: '10'
[2:7] SEMICOLON: ';'
[3:1] IF: 'if'
[3:4] LPAREN: '('
[3:5] IDENTIFIER: 'x'
[3:7] LT: '<'
[3:9] IDENTIFIER: 'y'
[3:10] RPAREN: ')'
[3:12] LBRACE: '{'
[4:5] IDENTIFIER: 'z'
[4:7] ASSIGN: '='
[4:9] IDENTIFIER: 'x'
[4:11] PLUS: '+'
[4:13] IDENTIFIER: 'y'
[4:14] SEMICOLON: ';'
[5:1] RBRACE: '}'
[5:3] ELSE: 'else'
[5:8] LBRACE: '{'
[6:5] IDENTIFIER: 'z'
[6:7] ASSIGN: '='
[6:9] IDENTIFIER: 'x'
```

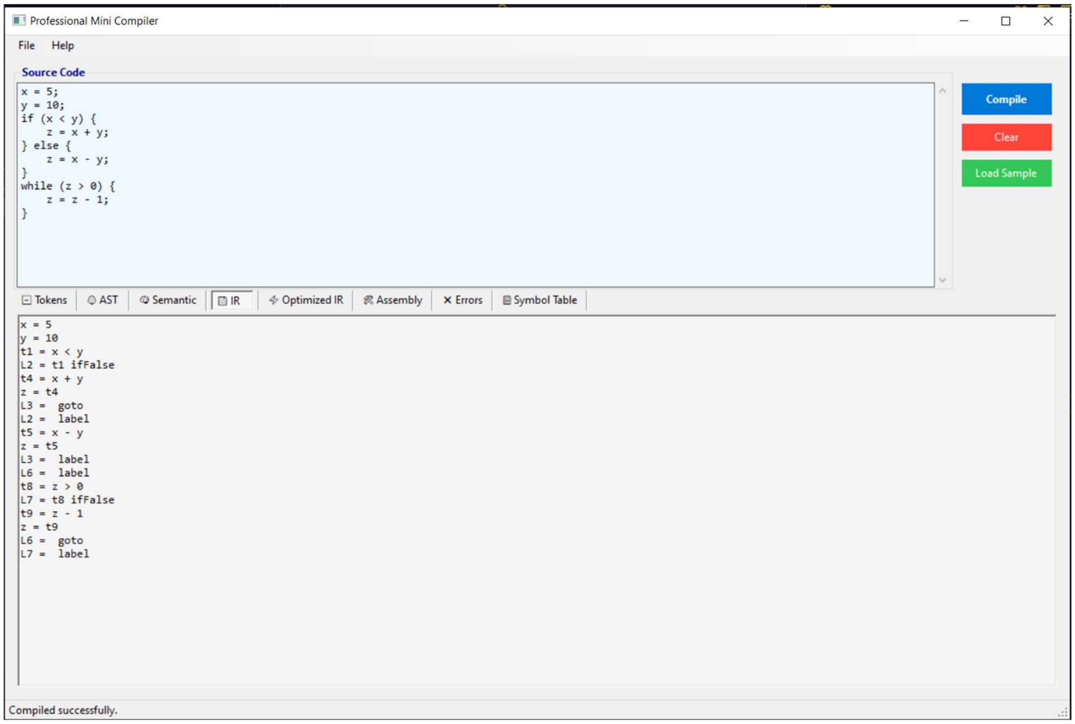
Compiled successfully.



```
x = 5;
y = 10;
if (x < y) {
    z = x + y;
} else {
    z = x - y;
}
while (z > 0) {
    z = z - 1;
}
```

```
Program
  Assignment: x
    Number: 5
  Assignment: y
    Number: 10
  If
    Condition:
      BinaryOp: <
        Identifier: x
        Identifier: y
    Then:
      Block
        Assignment: z
          BinaryOp: +
            Identifier: x
            Identifier: y
    Else:
      Block
        Assignment: z
          BinaryOp: -
            Identifier: x
            Identifier: y
  While
    Condition:
      BinaryOp: >
        Identifier: z
        Number: 0
```

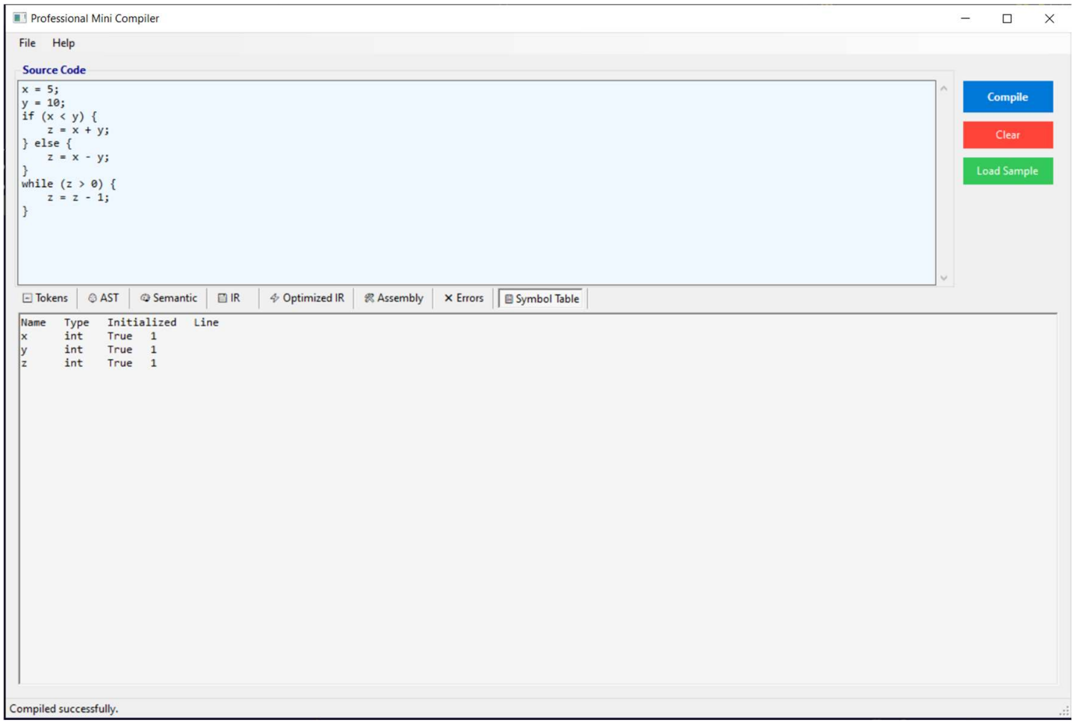
Compiled successfully.



```
x = 5;
y = 10;
if (x < y) {
    z = x + y;
} else {
    z = x - y;
}
while (z > 0) {
    z = z - 1;
}
```

```
x = 5
y = 10
t1 = x < y
L2 = t1 ifFalse
t4 = x + y
z = t4
L3 =  goto
L2 =  label
t5 = x - y
z = t5
L3 =  label
L6 =  label
t8 = z > 0
L7 = t8 ifFalse
t9 = z - 1
z = t9
L6 =  goto
L7 =  label
```

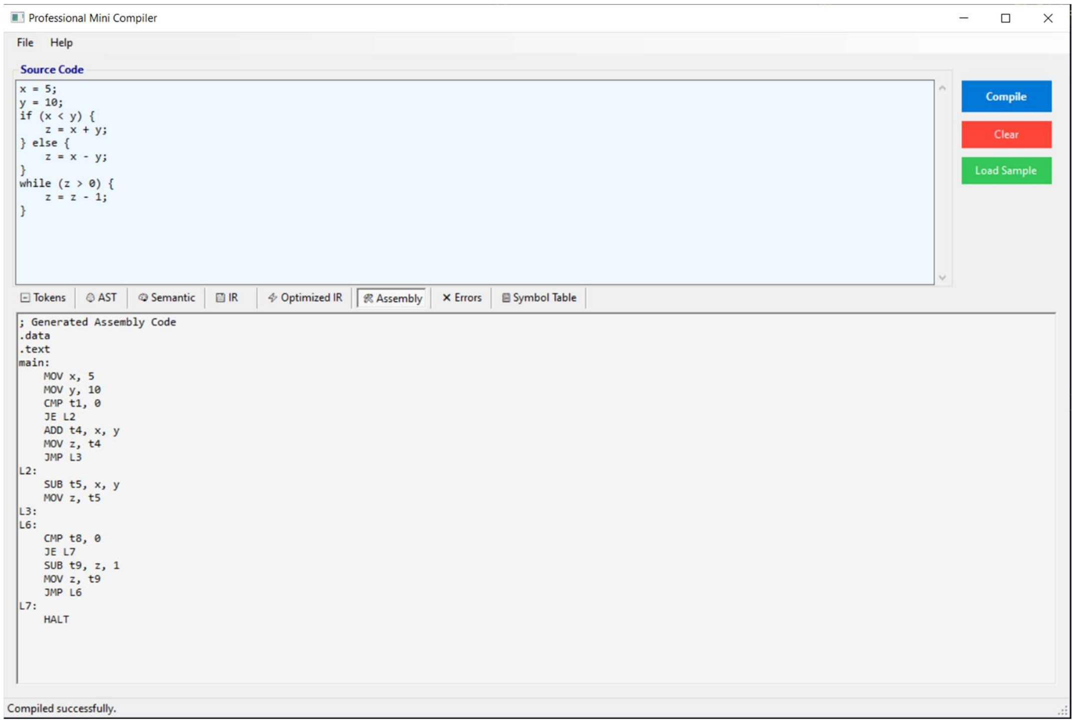
Compiled successfully.



```
x = 5;
y = 10;
if (x < y) {
    z = x + y;
} else {
    z = x - y;
}
while (z > 0) {
    z = z - 1;
}
```

```
Name   Type   Initialized   Line
x      int    True    1
y      int    True    1
z      int    True    1
```

Compiled successfully.

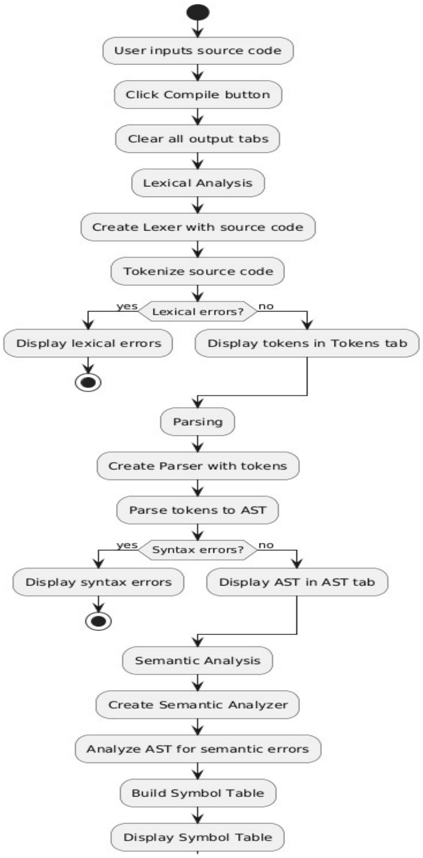


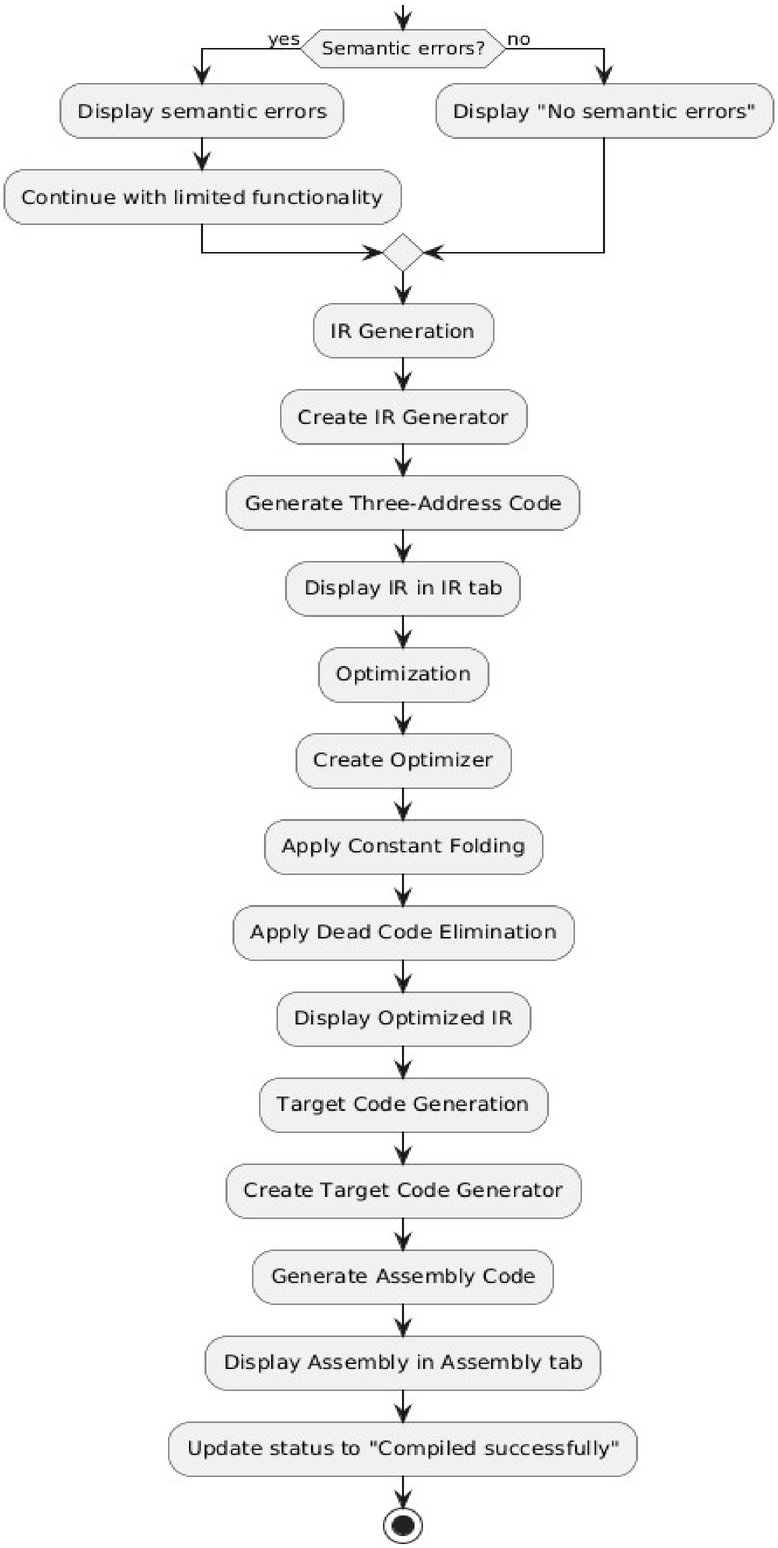
```
x = 5;
y = 10;
if (x < y) {
    z = x + y;
} else {
    z = x - y;
}
while (z > 0) {
    z = z - 1;
}
```

```
; Generated Assembly Code
.data
.text
main:
    MOV x, 5
    MOV y, 10
    CMP t1, 0
    JE L2
    ADD t4, x, y
    MOV z, t4
    JMP L3
L2:
    SUB t5, x, y
    MOV z, t5
L3:
L6:
    CMP t8, 0
    JE L7
    SUB t9, z, 1
    MOV z, t9
    JMP L6
L7:
    HALT
```

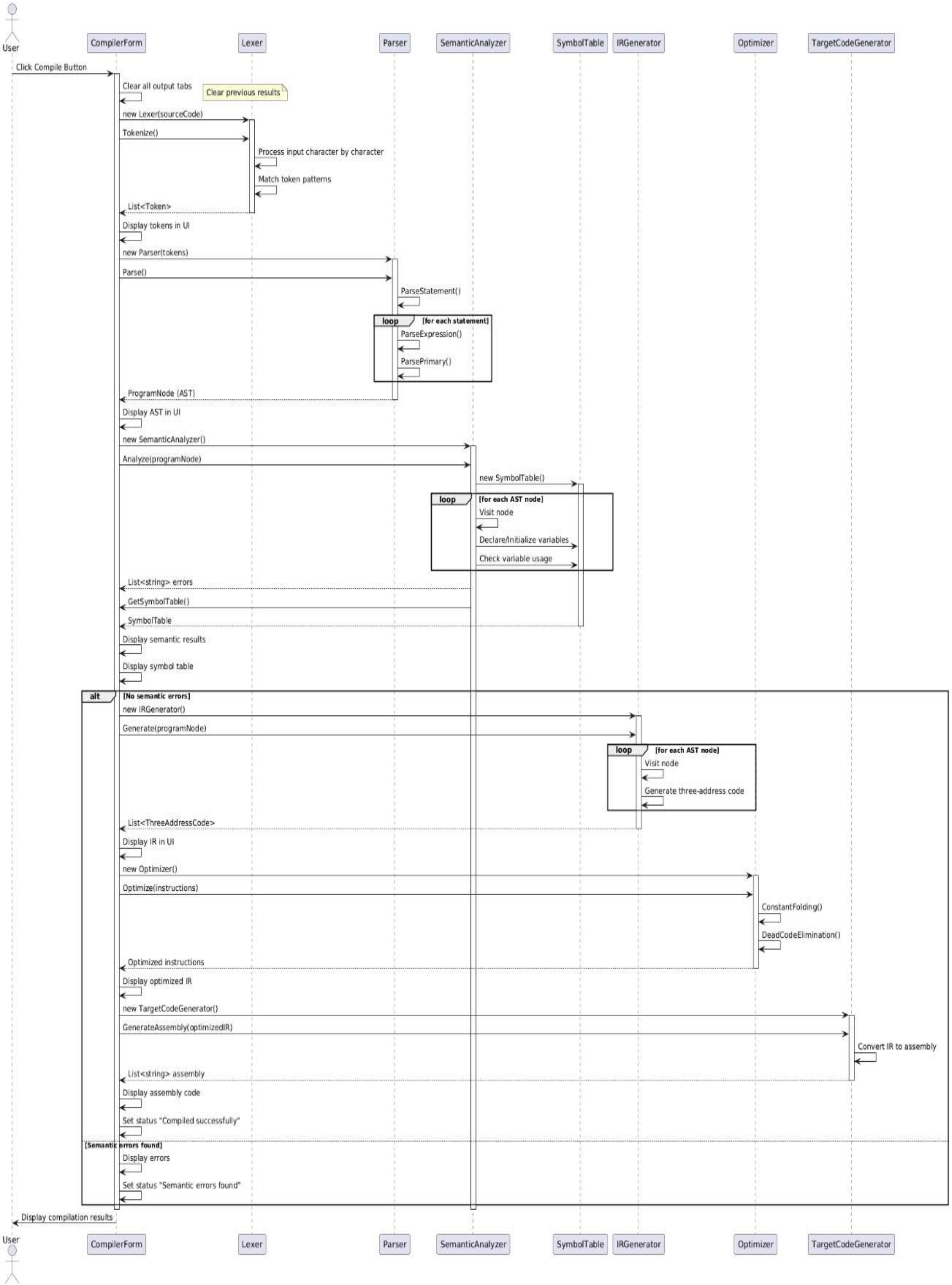
Compiled successfully.

Activity Diagram:

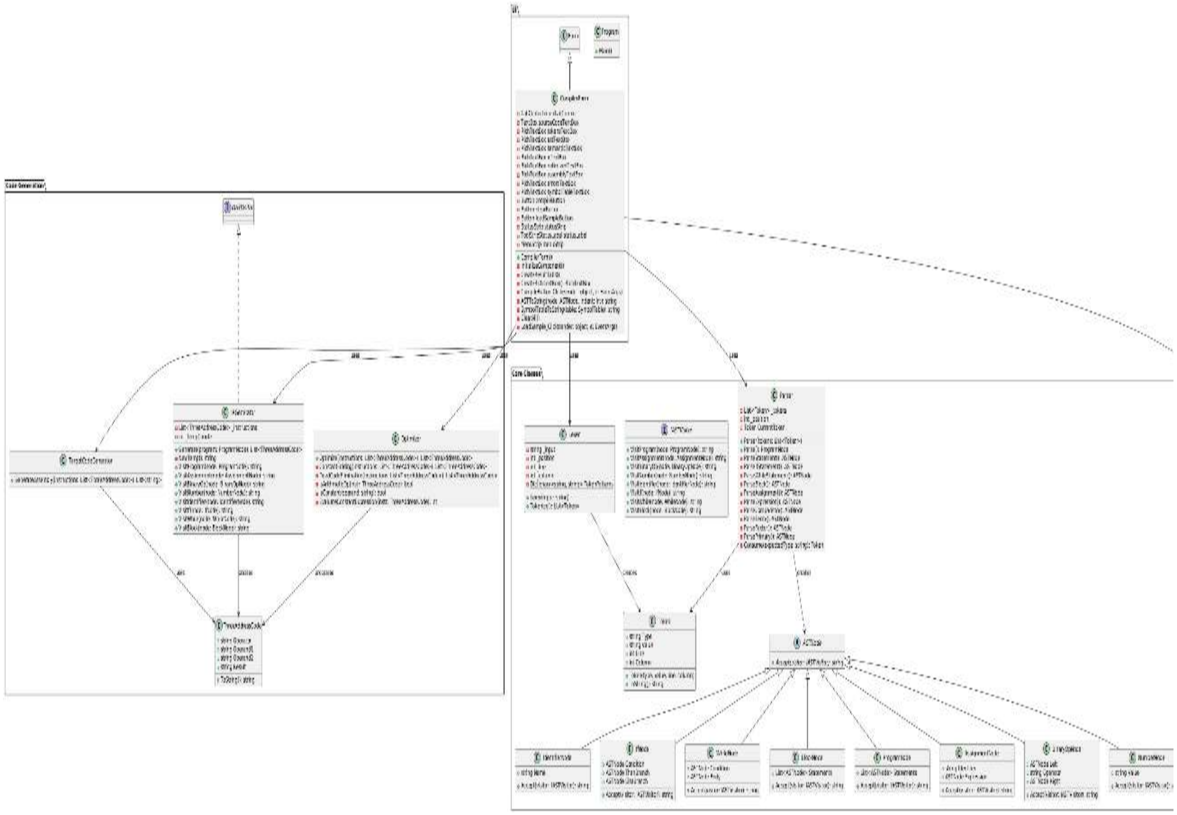



Semantic errors?
yes
no
Display semantic errors
Display "No semantic errors"
Continue with limited functionality
IR Generation
Create IR Generator
Generate Three-Address Code
Display IR in IR tab
Optimization
Create Optimizer
Apply Constant Folding
Apply Dead Code Elimination
Display Optimized IR
Target Code Generation
Create Target Code Generator
Generate Assembly Code
Display Assembly in Assembly tab
Update status to "Compiled successfully"

Sequence Diagram:



Class Diagram:


Program
Code Generation
Core Classes
Optimizer
ThreeAddressCode
Parser
ASTNode
uses
creates
produces