COMSATS University Islamabad

Attock Campus



**Semester Project (Mini Compiler)**

# Group Member:

Jamil Askari (Sp22-Bcs-007)

Muhammad Faizan (Sp22-Bcs-044)

# Submitted To:

**Subject:**

Sir Bilal Haider

Compiler Construction



**Date:** 30th May,2025



# GitHub Repository Link

[**https://github.com/Jamil**](https://github.com/Jamil) **Askari /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

}

# 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

# 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

# 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

# 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; }

}

# Semantic Analyzer

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

1. **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

}

# Optimizer

Performs code optimizations:

* + Constant Folding: Evaluates constant expressions at



compile time

# 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

# Menu Bar

* + File: New, Load Sample, Exit
  + Help: About

# Source Code Panel

* + Multi-line text editor with syntax highlighting support
  + Consolas font for better code readability

# Control Buttons

* + Compile: Execute full compilation pipeline
  + Clear: Clear all panels
  + Load Sample: Load example program

# Results Tabs

* + ⬛ᐻ⬛ Tokens: Lexical analysis results
  + ׸׺׷. AST: Abstract syntax tree visualization
  + C'⎒ç Semantic: Semantic analysis results
  + .ߵ˙\_o߱t IR: Intermediate representation
  + /f Optimized IR: Optimized intermediate code
  + tᢊᢉᢈ Assembly: Generated assembly code

o + Errors: Compilation errors

* + çാˆ഻ഽ† Symbol Table: Variable information

# Status Bar

* + Shows current compilation status

# Usage Guide

Getting Started

# Launch the Application

* + Run the executable or compile from Visual Studio
  + The main window will appear with an empty source code editor

# Write Code

* + Enter your program in the source code panel
  + Use the supported language syntax

# 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

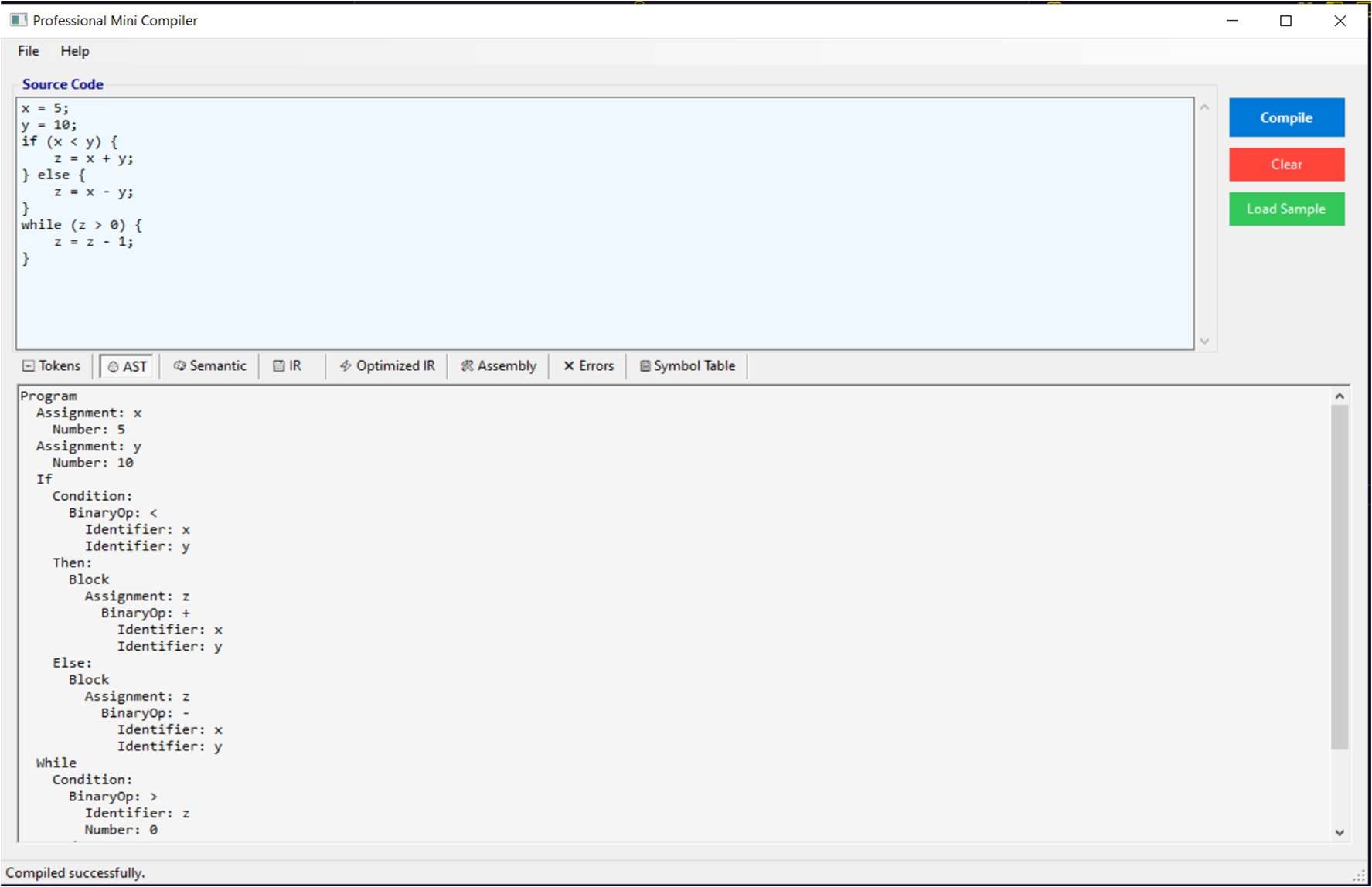
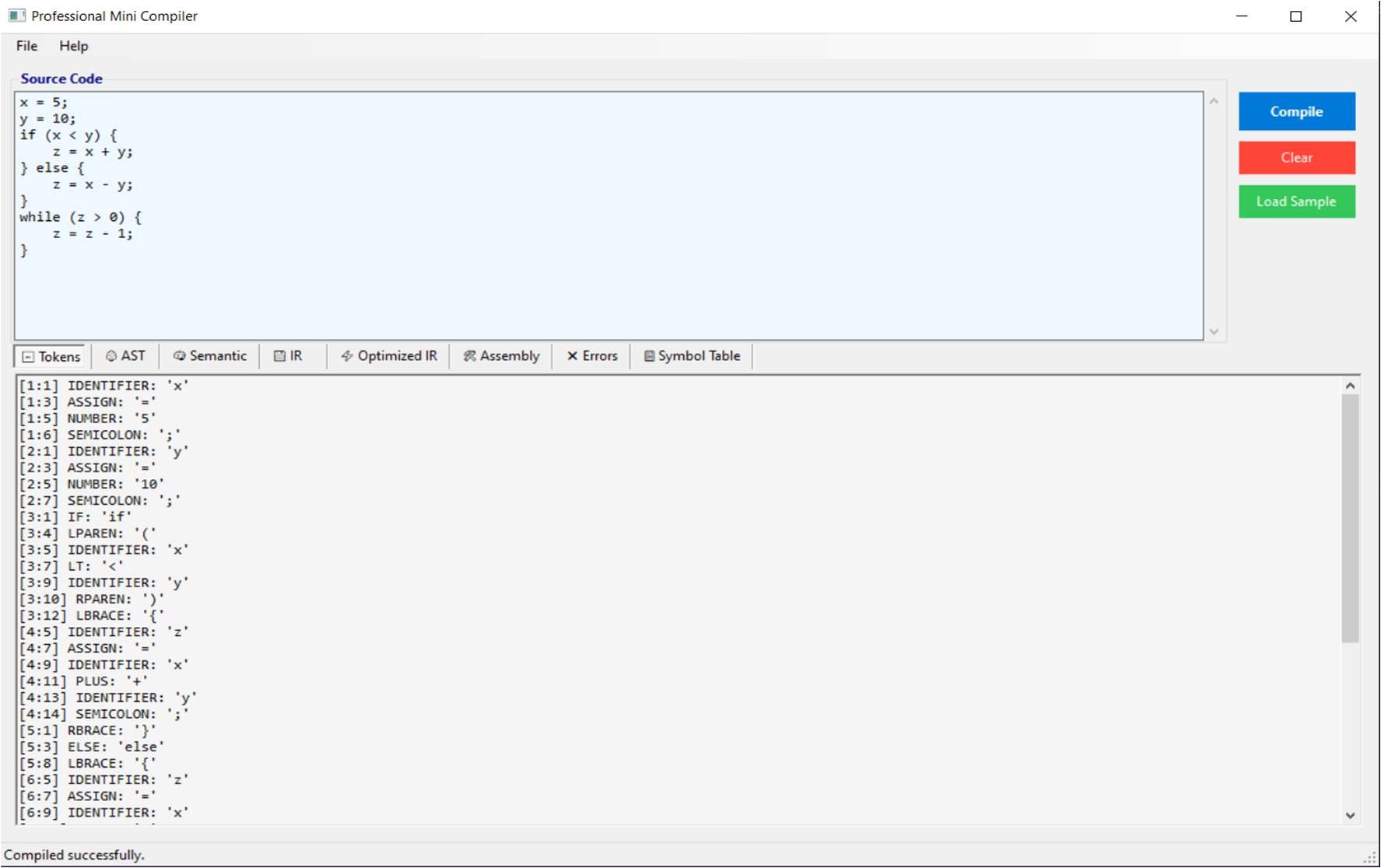
* Single-pass lexing
* Recursive descent parsing with minimal backtracking
* Efficient symbol table lookups using Dictionary
* Lazy evaluation in optimization passes

# Extensibility Points

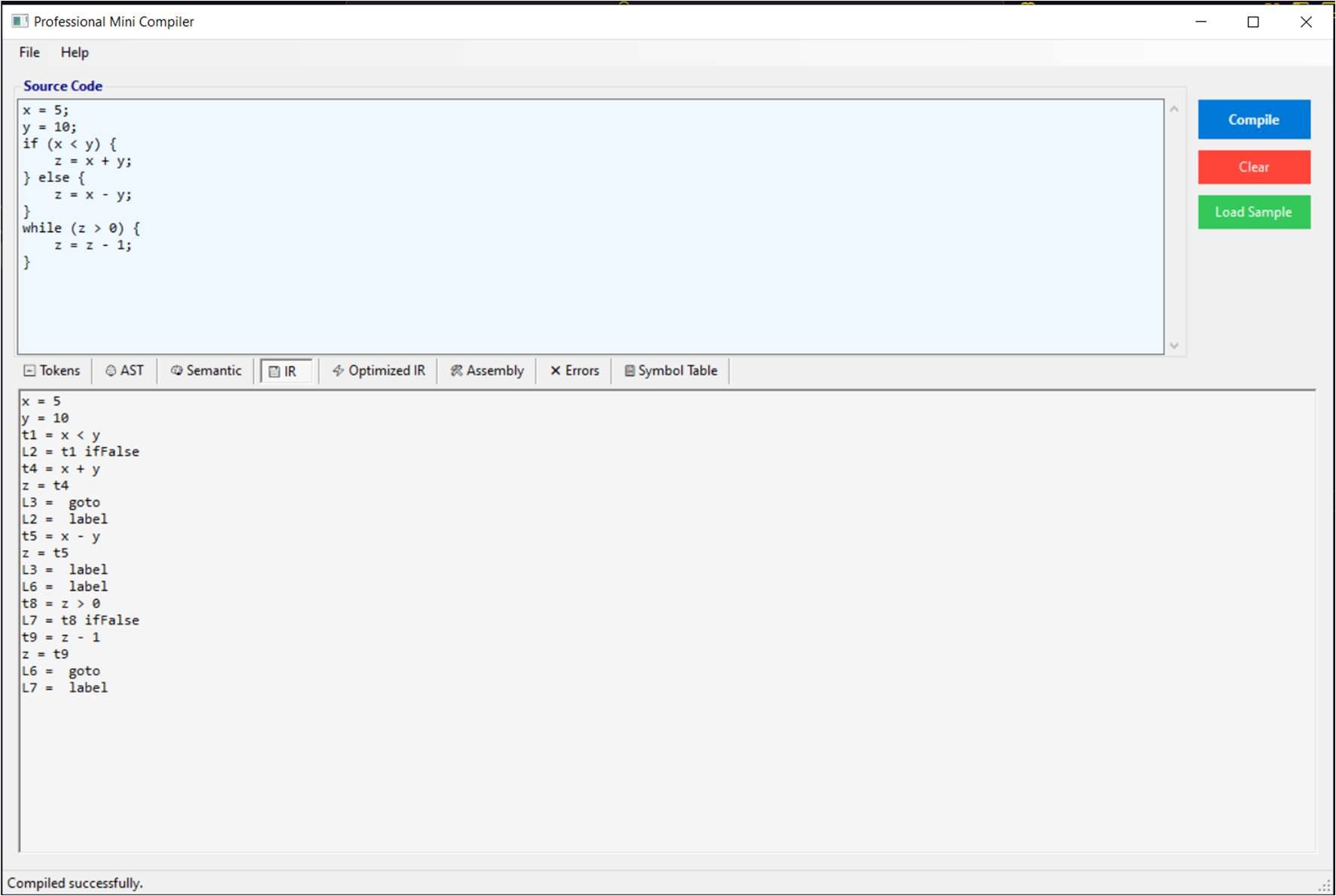
1. New Language Features: Add tokens, AST nodes, parser rules
2. Additional Optimizations: Extend Optimizer class
3. Different Target Architectures: Modify CodeGenerator
4. Enhanced UI: Add more visualization tabs

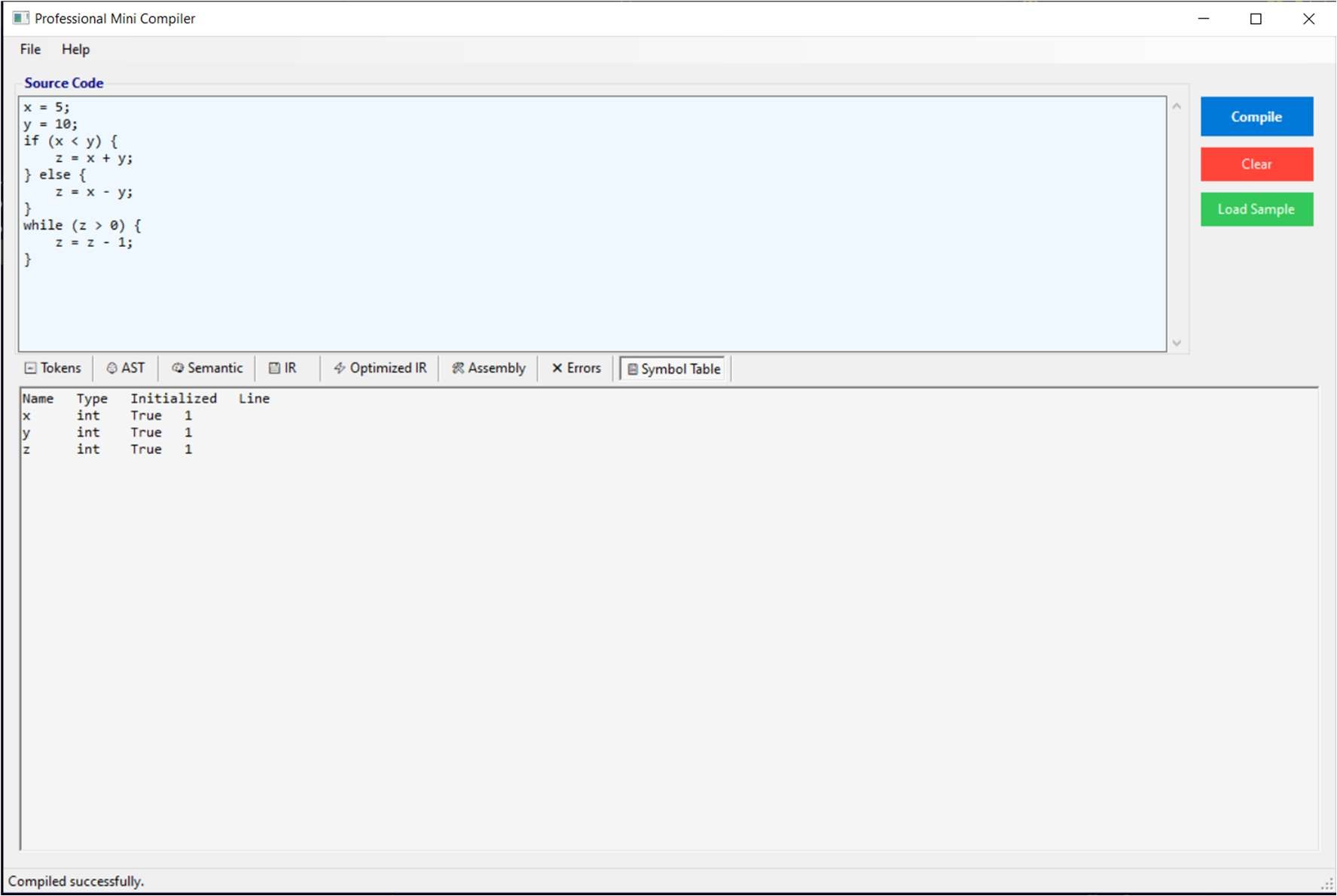


**Output:**

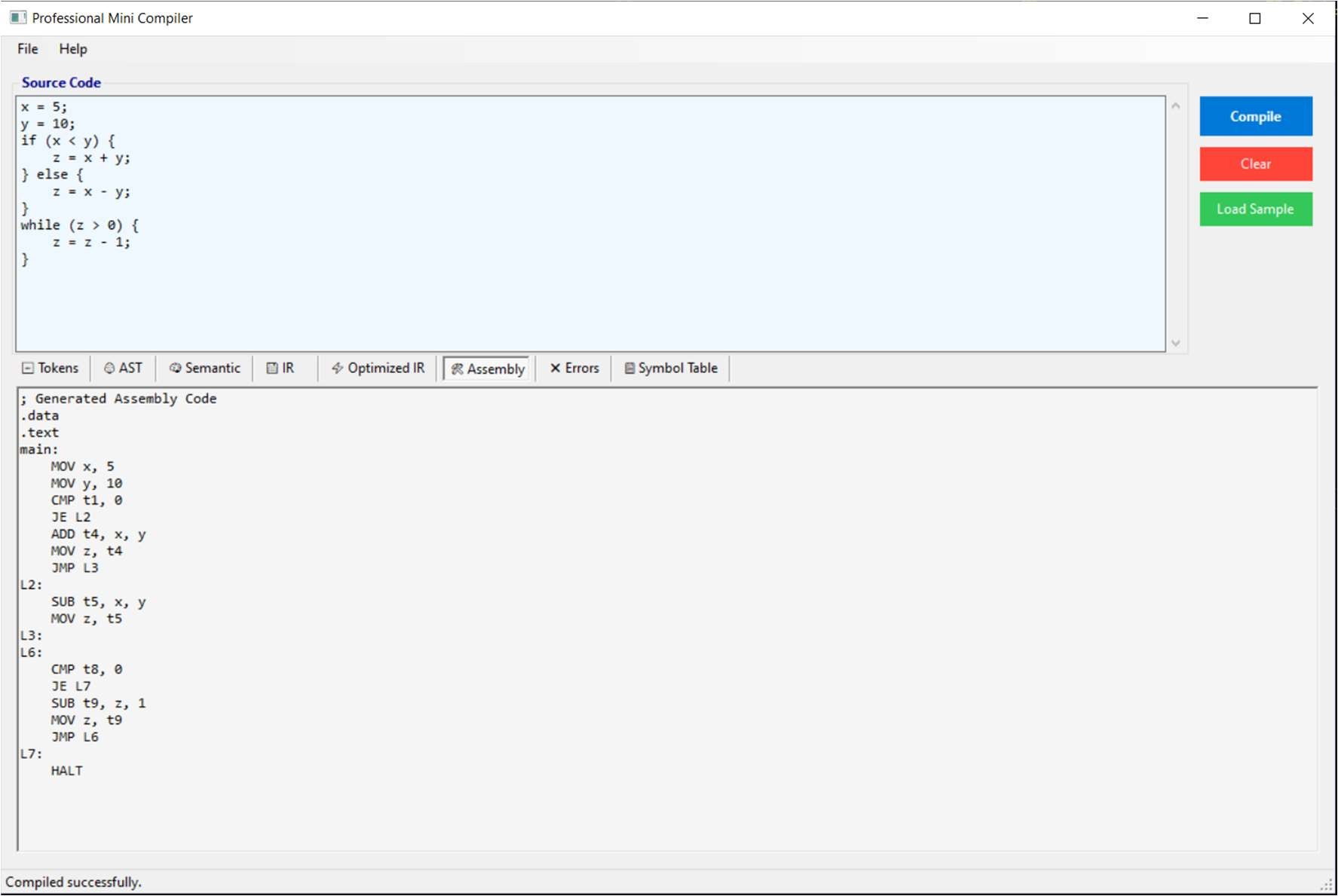
****



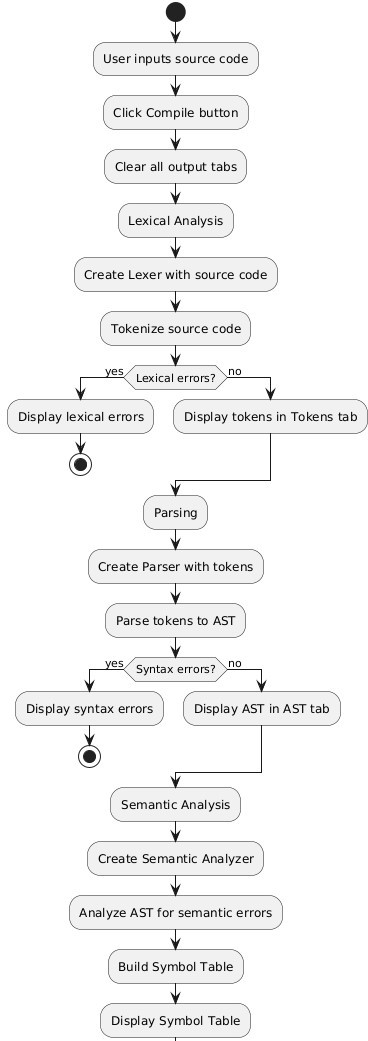


****

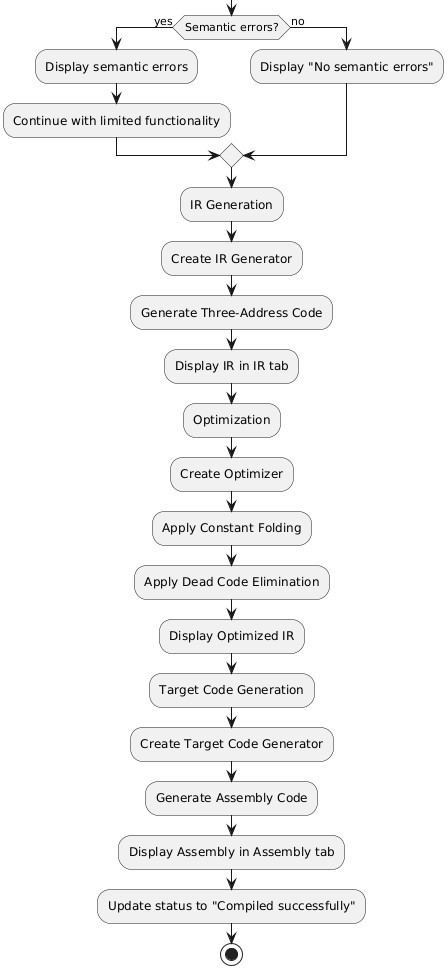




**Activity Diagram:**

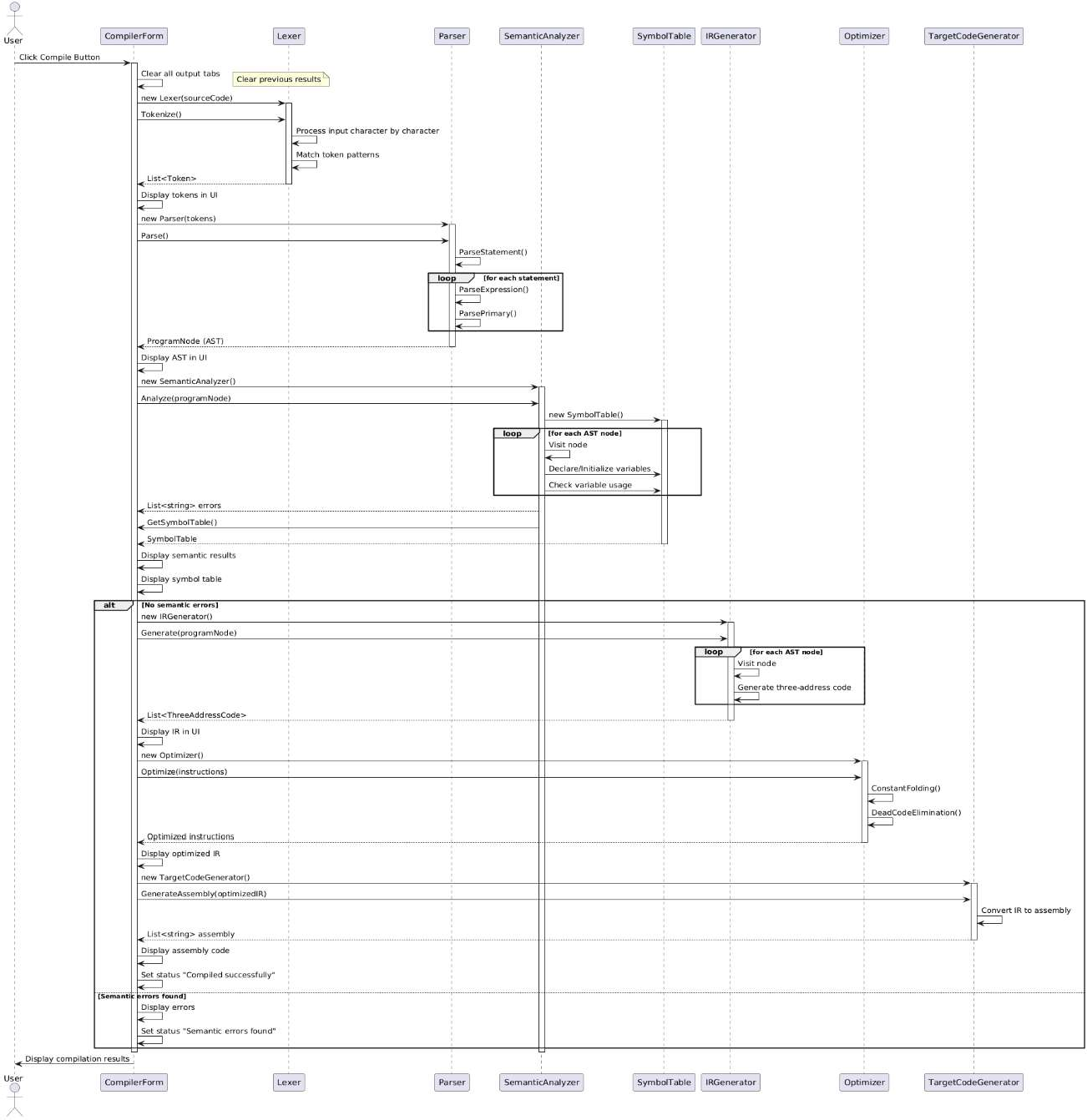
****





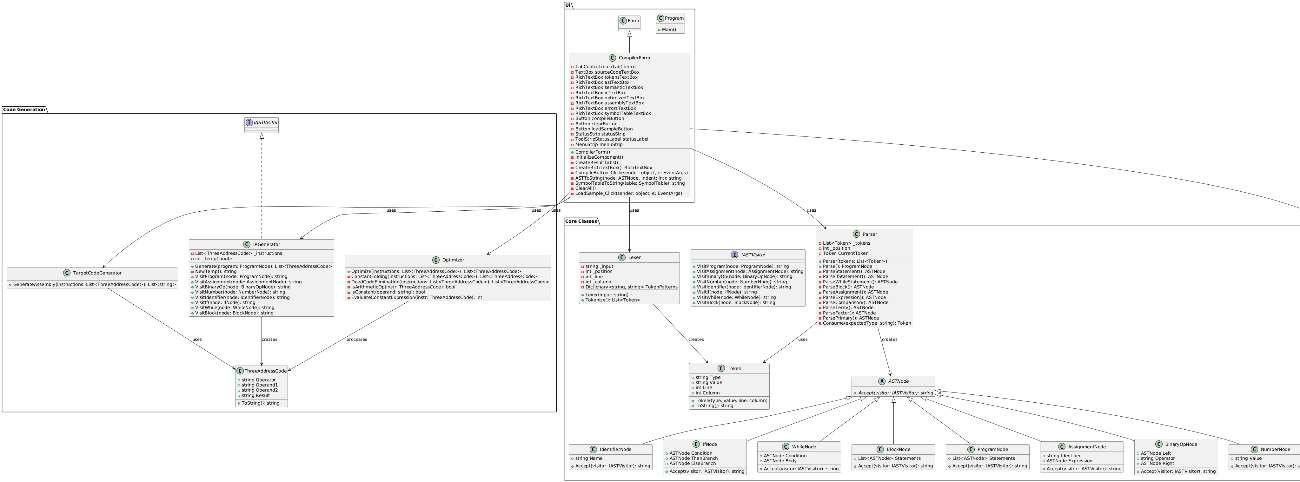


**Sequence Diagram:**

****



**Class Diagram:**

****