COMSATS UNIVERSITY ISLAMABAD

ATTOCK CAMPUS

Submitted To

Sir Bilal Haider

Submitted By

Muhammad Ayan Irfan

(SP22-BCS-060)

/Zohaib Alam

(SP22-BCS-028)

30th May, 2025.

MINI-COMPILER

Introduction and Overview

Project Overview

This mini compiler is a complete implementation of a programming language compiler written in C#. It demonstrates all major phases of compilation from source code to executable instructions, providing a practical example of compiler construction principles.

Supported Language Features

- **Data Types**: int, float, bool

- **Variables**: Declaration and assignment

- **Expressions**: Arithmetic, logical, and comparison operations

- **Control Flow**: if/else statements, while loops • **Built-in Functions**: print statement for output

- **Operators**:

  ○ Arithmetic: +, -, *, /, % ○ Comparison:

  ==, !=, <, >, <=, >= ○

  Logical: &&, ||, !

Target Platform

The compiler generates code for a custom stack-based virtual machine (VM), making it platform-independent while maintaining educational clarity.

Educational Value

This implementation serves as a learning tool for understanding:

- Compiler construction phases

- Abstract Syntax Tree (AST) generation

- Symbol table management

- Intermediate code generation

- Virtual machine execution

Compiler Architecture and Design

Seven-Phase Architecture

The compiler follows the traditional compilation pipeline with seven distinct phases:

Phase 1: Lexical Analysis (Tokenization)

- **Component**: Lexer class
- **Purpose**: Converts source code into tokens
- **Output**: List of tokens with type, value, and position information
- **Features**: Handles keywords, identifiers, literals, operators, and comments

Phase 2: Syntax Analysis (Parsing)

- **Component**: Parser class
- **Purpose**: Builds Abstract Syntax Tree from tokens
- **Algorithm**: Recursive descent parser
- **Grammar**: Supports operator precedence and associativity

Phase 3: Semantic Analysis

- **Component**: SemanticAnalyzer class
- **Purpose**: Type checking and symbol table management
- **Features**: Scope management, type compatibility verification, initialization checking

Phase 4: Intermediate Code Generation

- **Component**: IRGenerator class
- **Purpose**: Generates platform-independent intermediate representation
- **Format**: Three-address code style instructions

Phase 5: Optimization

- **Component**: Optimizer class
- **Purpose**: Code improvement (basic implementation)

- **Note**: Currently minimal, designed for future enhancement

Phase 6: Code Generation

- **Component**: CodeGenerator class
- **Purpose**: Translates IR to target machine code
- **Target**: Stack-based VM instructions

Phase 7: Execution

- **Component**: StackVM class
- **Purpose**: Executes generated code
- **Architecture**: Stack-based virtual machine with memory management

Design Patterns Used

- **Visitor Pattern**: For AST traversal in semantic analysis
- **Strategy Pattern**: For different optimization techniques
- **Interpreter Pattern**: For VM instruction execution

Language Specification and Grammar

Lexical Specification Keywords int, float, bool, void, if, else, while, for, function, return, print, true, false

Operators and Delimiters

Arithmetic: + - * / %

Assignment: =

Comparison: == != < > <= >=

Logical: && || !

Delimiters: ; , ( ) { } .

Literals

- **Integer**: Sequences of digits (e.g., 42, 0, 123)
- **Float**: Numbers with decimal points (e.g., 3.14, 0.5)

- **Boolean**: true or false
- **String**: Quoted text with escape sequences **Grammar Specification (BNF-style)** bnf program → statement*

statement → varDecl | assignment | ifStmt | whileStmt
| printStmt | block | ";"

varDecl → type IDENTIFIER ("=" expression)? ";" assignment →
IDENTIFIER "=" expression ";" ifStmt → "if" "(" expression ")"
statement ("else" statement)?

whileStmt → "while" "(" expression ")" statement

printStmt → "print" expression ";" block →
"{" statement* "}"

expression → logicalOr logicalOr → logicalAnd ("||"
logicalAnd)* logicalAnd → equality ("&&" equality)*

equality → comparison (("==" | "!=") comparison)*

comparison → term ((">" | ">=" | "<" | "<=") term)* term
→ factor (("+" | "-") factor)* factor → unary (("*" | "/"
| "%") unary)* unary → ("!" | "-") unary | primary

primary → NUMBER | FLOAT | BOOLEAN | IDENTIFIER
| "(" expression ")"

type → "int" | "float" | "bool"

Operator Precedence (Highest to Lowest)

1. Unary operators (!, -)

2. Multiplicative (*, /, %)

3. Additive (+, -)

4. Relational (<, >, <=, >=)

5. Equality (==, !=)

6. Logical AND (&&)

7. Logical OR (||)

Type System

- **Static Typing**: All variables must be declared with explicit types

- **Type Compatibility**: Automatic promotion from int to float in mixed expressions

- **Type Checking**: Compile-time verification of type consistency

Implementation Details

Key Data Structures Token Structure

csharp public class Token {   public

TokenType Type { get; set; }   public

string Value { get; set; }   public int

Line { get; set; }   public int Column

{ get; set; }

}

AST Node Hierarchy

- **Base Class**: ASTNode with type and position information

- **Expression Nodes**: BinaryOpNode, UnaryOpNode, NumberNode, etc.

- **Statement Nodes**: VarDeclarationNode, AssignmentNode, IfNode,

etc. **Symbol Table** csharp public class Symbol {   public string Name { get;

set; }   public DataType Type { get; set; }   public int Scope { get; set; }

public int Address { get; set; }   public bool IsInitialized { get; set; }

}

Error Handling Strategy

- **Custom Exception**: CompilerException with line/column information

- **Error Recovery**: Continues parsing after errors when possible

- **Semantic Errors**: Detailed type mismatch and undeclared variable messages

Memory Management

- **Stack-based VM**: Uses evaluation stack for expression computation

- **Variable Storage**: Array-based memory with address allocation

- **Scope Management**: Hierarchical symbol table with scope stack

Intermediate Representation

The IR uses a simple instruction set:

- **Data Movement**: LOAD_CONST, LOAD_VAR, STORE_VAR

- **Arithmetic**: ADD, SUB, MUL, DIV, MOD

- **Comparison**: CMP_EQ, CMP_NEQ, CMP_LT, etc.

- **Control Flow**: JUMP, JUMP_IF_FALSE, LABEL

- **I/O**: PRINT

Virtual Machine Architecture

- **Stack-based Execution**: Operands pushed/popped from evaluation stack

- **Memory Model**: Linear array for variable storage

- **Instruction Pointer**: Sequential execution with jump capabilities

- **Type Handling**: Dynamic type conversion during arithmetic operations

Diagrams

Activity Diagram:

```
                    ●
              Read Source Code
           Phase 1: Lexical Analysis
              Create Lexer
            Tokenize source code
            Generate tokens list ─────┤ Converts source code into tokens
           Phase 2: Syntax Analysis
              Create Parser
            Parse tokens into AST ─────┤ Builds Abstract Syntax Tree
          Phase 3: Semantic Analysis
            Create Symbol Table
          Create Semantic Analyzer
        Analyze AST for type checking
            Build annotated AST ─────┤ Type checking and symbol resolution
           Phase 4: IR Generation
            Create IR Generator
        Generate intermediate code ─────┤ Creates platform-independent IR
           Phase 5: Optimization
              Create Optimizer
            Optimize IR code ─────┤ Basic optimizations (currently minimal)
          Phase 6: Code Generation
          Create Code Generator
          Generate target code ─────┤ Converts IR to VM instructions
            Phase 7: Execution
              Create Stack VM
            Execute target code
              Display output ─────┤ Runtime execution on stack-based VM
        yes ◇ Compilation successful? ◇ no
  Display "Compilation Successful"   Display compilation error
              ◉                  Show line and column info
                                        ◉
```
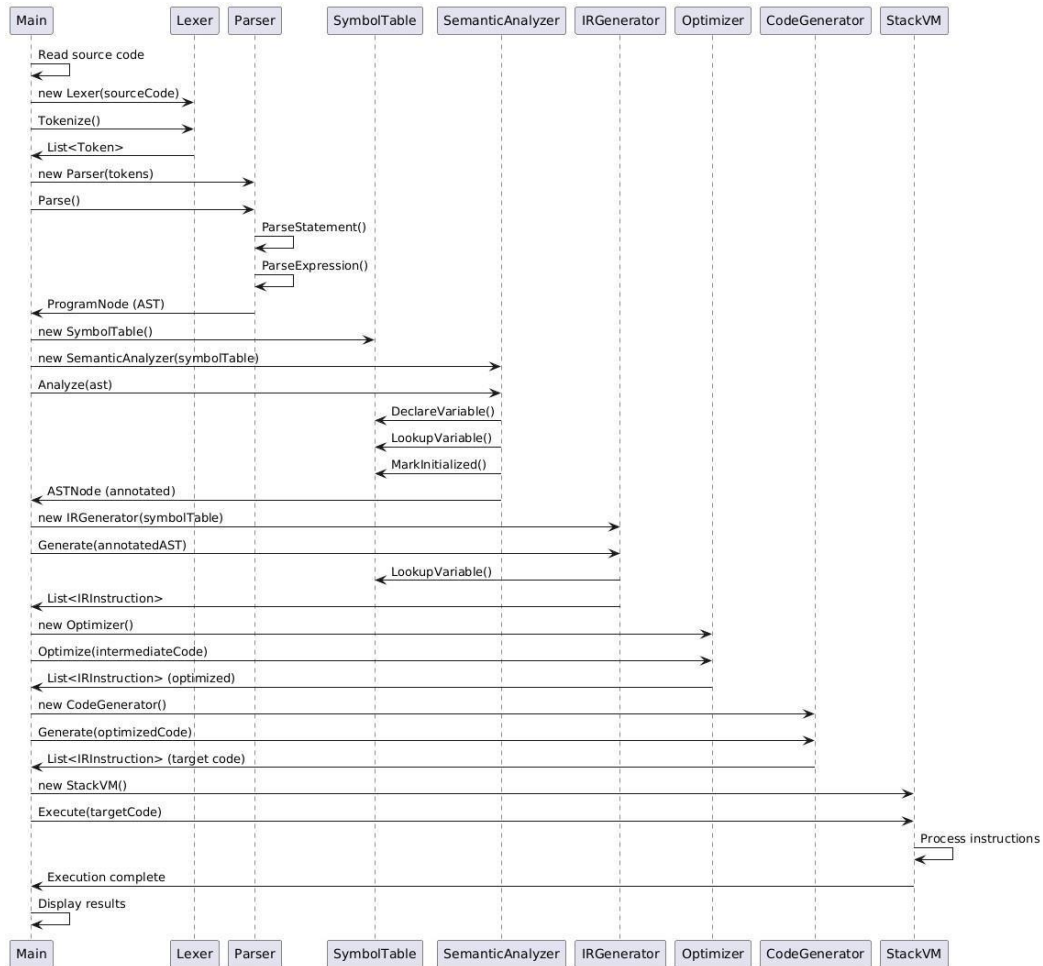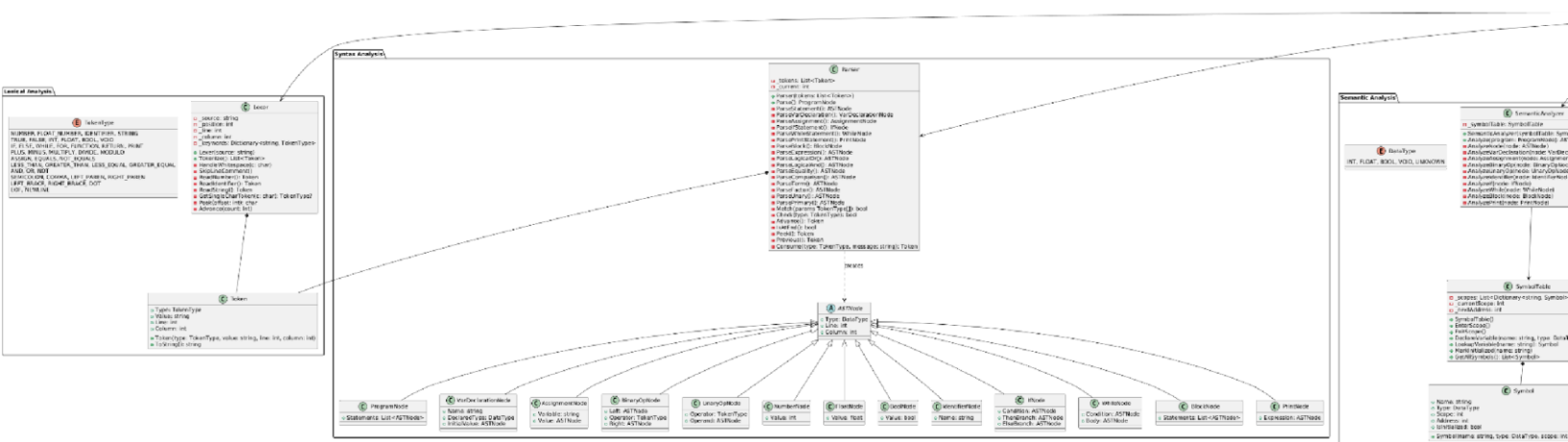
Sequence Diagram:

Diagram:

GitHub Repository

https://github.com/UMAR-CUI/Mini-Compiler-SP22.git