COMSATS UNIVERSITY ISLAMABD
ATTOCK CAMPUS

DEPARTMENT OF COMPUTER SCIENCES

SEMISTER PROJECT

PROJECT NAME:

MINI COMPILER

SUBMITTED BY:

MARYAM NOOR (SP22-BCS-002)

RIDDA ZAINAB (SP22-BCS-039)

SUBMITTED TO:

SIR BILAL

COURSE NAME:

COMPILER CONSTRUCTION

DATE:

30-May-2025

Mini Compiler?

A mini compiler is a small and simplified version of a full compiler. Its main job is to translate source code written in a programming language into machine code or an intermediate code. Mini compilers are often created for educational purposes to help students understand how compilers work. They cover the core compiler components like lexical analysis, syntax analysis, semantic analysis, and code generation, but usually handle a smaller language or provide limited functionality.

PHASES OF COMPILATION:

Lexical Analysis (Tokenization):

Lexical analysis is the first phase of a compiler. In this phase, the source code is broken down into tokens. Tokens are the smallest meaningful units of the code, such as keywords (if, while), identifiers (variable names), literals (numbers, strings), and operators (+, -, =). During this phase, whitespace and comments are removed so only meaningful code remains. The component performing this task is called the lexical analyzer or scanner. Tools like Lex are used, or programmers write their own tokenizers often based on regular expressions.

Syntax Analysis (Parsing):

Syntax analysis checks whether the sequence of tokens follows the grammar rules of the programming language. It is based on context-free grammar (CFG). If the syntax is incorrect, the compiler reports an error. The output of this phase is a parse tree or an abstract syntax tree (AST), which represents the structure of the code. A parse tree includes all tokens, while the AST removes unnecessary details and shows only the essential structure. Tools like Yacc/Bison or hand-written recursive descent parsers are used here.

Semantic Analysis:

Semantic analysis verifies the logical correctness of the code. It ensures every variable is declared before use, expressions have correct types, and functions receive appropriate arguments. For example, if you try to add an integer to a string or use a variable without declaring it, this phase will detect an error. The AST is annotated with type and other semantic information to clarify the meaning of the code. Catching semantic errors early prevents runtime problems.

Intermediate Code Generation (IR):

In this phase, the compiler converts the AST into an intermediate representation (IR) that is machine-independent. The IR makes it easier to optimize the code and later translate it into machine-specific code. A common form of IR is the three-address code (TAC), where each statement involves an operator and up to two operands. Using IR simplifies debugging and optimization processes.

Optimization (Optional):

Optimization is an optional phase aimed at making the code run more efficiently. The compiler may remove redundant calculations (constant folding) or eliminate dead code (code that never executes). Optimization results in faster, smaller code. Mini compilers usually implement only basic optimizations since their focus is on teaching the core concepts rather than producing highly efficient code.

Target Code Generation:

Here, the intermediate representation is converted into the final machine code, assembly code, or code for a virtual machine. This code can be directly executed by the hardware or run through an interpreter. During this phase, techniques like register allocation, instruction selection, and memory management are applied to ensure the executable runs well on the target machine.

Symbol Table Management

The symbol table is an important data structure in the compiler that keeps track of variable names, their types, scopes, and memory locations. When a variable is declared, its information is stored in the symbol table. When a variable is used, the compiler checks the symbol table to confirm it is valid. Symbol tables play a crucial role during semantic analysis and target code generation.

Error Handling

Error handling is present throughout all phases but focuses on detecting errors and giving clear, user-friendly messages to the programmer. The compiler identifies syntax errors (like missing semicolons or braces) and semantic errors (like type mismatches or undeclared variables). Error messages usually include the line number and error type to help developers quickly fix issues.

CODE:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MiniCompiler
{
   public enum TokenType
   {
      NUMBER, IDENTIFIER, INT, FLOAT, IF, ELSE, WHILE, PRINT,
```

```
    PLUS, MINUS, MULTIPLY, DIVIDE, ASSIGN, EQUAL, NOT_EQUAL,
    LESS_THAN, GREATER_THAN, AND, OR,
    SEMICOLON, LEFT_PAREN, RIGHT_PAREN, LEFT_BRACE, RIGHT_BRACE,
    EOF, UNKNOWN
}
public class Token
{
    public TokenType Type { get; set; }
    public string Value { get; set; }
    public int Line { get; set; }
    public int Column { get; set; }
    public Token(TokenType type, string value, int line, int column)
    {
        Type = type;
        Value = value;
        Line = line;
        Column = column;
    }
    public override string ToString()
    {
        return $"{Type,-12} | {Value,-8} | Line {Line,2}, Col {Column,2}";
    }
}
public abstract class ASTNode
{
    public abstract string ToStringIndented(int indent = 0);
    protected string GetIndent(int level) => new string(' ', level * 2);
}
public abstract class Expression : ASTNode { }
public class NumberExpression : Expression
{
    public double Value { get; set; }
    public NumberExpression(double value) { Value = value; }
```

```
    public override string ToStringIndented(int indent = 0)
    {
        return $"{GetIndent(indent)}Number: {Value}";
    }
}
public class IdentifierExpression : Expression
{
    public string Name { get; set; }
    public IdentifierExpression(string name) { Name = name; }
    public override string ToStringIndented(int indent = 0)
    {
        return $"{GetIndent(indent)}Identifier: {Name}";
    }
}
public class BinaryExpression : Expression
{
    public Expression Left { get; set; }
    public TokenType Operator { get; set; }
    public Expression Right { get; set; }
    public BinaryExpression(Expression left, TokenType op, Expression right)
    {
        Left = left;
        Operator = op;
        Right = right;
    }
    public override string ToStringIndented(int indent = 0)
    {
        var sb = new StringBuilder();
        sb.AppendLine($"{GetIndent(indent)}BinaryOp: {Operator}");
        sb.AppendLine($"{GetIndent(indent + 1)}Left:");
        sb.AppendLine(Left.ToStringIndented(indent + 2));
        sb.AppendLine($"{GetIndent(indent + 1)}Right:");
        sb.Append(Right.ToStringIndented(indent + 2));
```

```
        return sb.ToString();
    }
}
public abstract class Statement : ASTNode { }
public class DeclarationStatement : Statement
{
    public string Variable { get; set; }
    public string Type { get; set; }
    public DeclarationStatement(string variable, string type = "int")
    {
        Variable = variable;
        Type = type;
    }
    public override string ToStringIndented(int indent = 0)
    {
        return $"{GetIndent(indent)}Declaration: {Type} {Variable}";
    }
}
public class AssignmentStatement : Statement
{
    public string Variable { get; set; }
    public Expression Value { get; set; }
    public AssignmentStatement(string variable, Expression value)
    {
        Variable = variable;
        Value = value;

    public override string ToStringIndented(int indent = 0)
    {
        var sb = new StringBuilder();
        sb.AppendLine($"{GetIndent(indent)}Assignment: {Variable}");
        sb.Append(Value.ToStringIndented(indent + 1));
        return sb.ToString();
```

```
    }
}
public class PrintStatement : Statement
{
    public Expression Expression { get; set; }
    public PrintStatement(Expression expr) { Expression = expr; }
    public override string ToStringIndented(int indent = 0)
    {
        var sb = new StringBuilder();
        sb.AppendLine($"{GetIndent(indent)}Print:");
        sb.Append(Expression.ToStringIndented(indent + 1));
        return sb.ToString();
    }
}
public class IfStatement : Statement
{
    public Expression Condition { get; set; }
    public List<Statement> ThenBlock { get; set; }
    public List<Statement> ElseBlock { get; set; }
    public IfStatement(Expression condition, List<Statement> thenBlock, List<Statement> elseBlock = null)
    {
        Condition = condition;
        ThenBlock = thenBlock;
        ElseBlock = elseBlock ?? new List<Statement>();
    }
    public override string ToStringIndented(int indent = 0)
    {
        var sb = new StringBuilder();
        sb.AppendLine($"{GetIndent(indent)}If:");
        sb.AppendLine($"{GetIndent(indent + 1)}Condition:");
        sb.AppendLine(Condition.ToStringIndented(indent + 2));
        sb.AppendLine($"{GetIndent(indent + 1)}Then:");
        foreach (var stmt in ThenBlock)
```

```
            sb.AppendLine(stmt.ToStringIndented(indent + 2));
        if (ElseBlock.Any())
        {
            sb.AppendLine($"{GetIndent(indent + 1)}Else:");
            foreach (var stmt in ElseBlock)
                sb.AppendLine(stmt.ToStringIndented(indent + 2));
        }
        return sb.ToString().TrimEnd();
    }
}
public class WhileStatement : Statement
{
    public Expression Condition { get; set; }
    public List<Statement> Body { get; set; }
    public WhileStatement(Expression condition, List<Statement> body)
    {
        Condition = condition;
        Body = body;
    }
    public override string ToStringIndented(int indent = 0)
    {
        var sb = new StringBuilder();
        sb.AppendLine($"{GetIndent(indent)}While:");
        sb.AppendLine($"{GetIndent(indent + 1)}Condition:");
        sb.AppendLine(Condition.ToStringIndented(indent + 2));
        sb.AppendLine($"{GetIndent(indent + 1)}Body:");
        foreach (var stmt in Body)
            sb.AppendLine(stmt.ToStringIndented(indent + 2));
        return sb.ToString().TrimEnd();
    }
}
public class Program : ASTNode
{
```

```
    public List<Statement> Statements { get; set; }
    public Program() { Statements = new List<Statement>(); }
    public override string ToStringIndented(int indent = 0)
    {
        var sb = new StringBuilder();
        sb.AppendLine($"{GetIndent(indent)}Program:");
        foreach (var stmt in Statements)
            sb.AppendLine(stmt.ToStringIndented(indent + 1));
        return sb.ToString().TrimEnd();
    }
}
public class Symbol
{
    public string Name { get; set; }
    public string Type { get; set; }
    public double Value { get; set; }
    public bool IsInitialized { get; set; }
    public Symbol(string name, string type)
    {
        Name = name;
        Type = type;
        Value = 0.0;
        IsInitialized = false;
    }
    public override string ToString()
    {
        return $"{Name,-10} | {Type,-6} | {(IsInitialized ? "Yes" : "No"),-5} | {Value}";
    }
}
public class Lexer
{
    private readonly string source;
    private int position;
```

```
private int line = 1;
private int column = 1;
private char currentChar;
private readonly Dictionary<string, TokenType> keywords = new Dictionary<string, TokenType>
{
    { "int", TokenType.INT },
    { "float", TokenType.FLOAT },
    { "if", TokenType.IF },
    { "else", TokenType.ELSE },
    { "while", TokenType.WHILE },
    { "print", TokenType.PRINT }
};
public Lexer(string source)
{
    this.source = source;
    position = 0;
    currentChar = position < source.Length ? source[position] : '\0';
}
private void Advance()
{
    if (currentChar == '\n')
    {
        line++;
        column = 1;
    }
    else
    {
        column++;
    }
    position++;
    currentChar = position < source.Length ? source[position] : '\0';
}
private void SkipWhitespace()
```

```
{
    while (char.IsWhiteSpace(currentChar))
        Advance();

private void SkipComment()
{
    if (currentChar == '/' && position + 1 < source.Length && source[position + 1] == '/')
    {
        while (currentChar != '\n' && currentChar != '\0')
            Advance();
    }
}
private string ReadNumber()
{
    var sb = new StringBuilder();
    bool hasDecimal = false;
    while (char.IsDigit(currentChar) || (currentChar == '.' && !hasDecimal))
    {
        if (currentChar == '.') hasDecimal = true;
        sb.Append(currentChar);
        Advance();
    }
    return sb.ToString();
}
private string ReadIdentifier()
{
    var sb = new StringBuilder();
    while (char.IsLetterOrDigit(currentChar) || currentChar == '_')
    {
        sb.Append(currentChar);
        Advance();
    }
    return sb.ToString();
```

```
    }
    public List<Token> Tokenize()
    {
        var tokens = new List<Token>();
        while (currentChar != '\0')
        {
            SkipWhitespace();
            if (currentChar == '\0') break;
            if (currentChar == '/' && position + 1 < source.Length && source[position + 1] == '/')
            {
                SkipComment();
                continue;
            }
            int currentLine = line;
            int currentColumn = column;
            if (char.IsDigit(currentChar))
            {
                string number = ReadNumber();
                tokens.Add(new Token(TokenType.NUMBER, number, currentLine, currentColumn));
            }
            else if (char.IsLetter(currentChar) || currentChar == '_')
            {
                string identifier = ReadIdentifier();
                TokenType type = keywords.ContainsKey(identifier) ? keywords[identifier] :
TokenType.IDENTIFIER;
                tokens.Add(new Token(type, identifier, currentLine, currentColumn));
            }
            else
            {
                switch (currentChar)
                {
                    case '+':
                        tokens.Add(new Token(TokenType.PLUS, "+", currentLine, currentColumn));
```

```
            Advance();
            break;
        case '-':
            tokens.Add(new Token(TokenType.MINUS, "-", currentLine, currentColumn));
            Advance();
            break;
        case '*':
            tokens.Add(new Token(TokenType.MULTIPLY, "*", currentLine, currentColumn));
            Advance();
            break;
        case '/':
            tokens.Add(new Token(TokenType.DIVIDE, "/", currentLine, currentColumn));
            Advance();
            break;
        case '=':
            Advance();
            if (currentChar == '=')
            {
                tokens.Add(new Token(TokenType.EQUAL, "==", currentLine, currentColumn));
                Advance();
            }
            else
            {
                tokens.Add(new Token(TokenType.ASSIGN, "=", currentLine, currentColumn));
            }
            break;
        case '!':
            Advance();
            if (currentChar == '=')
            {
                tokens.Add(new Token(TokenType.NOT_EQUAL, "!=", currentLine, currentColumn));
                Advance();
            }
```

```
            else
            {
                tokens.Add(new Token(TokenType.UNKNOWN, "!", currentLine, currentColumn));
            }
            break;
        case '<':
            tokens.Add(new Token(TokenType.LESS_THAN, "<", currentLine, currentColumn));
            Advance();
            break;
        case '>':
            tokens.Add(new Token(TokenType.GREATER_THAN, ">", currentLine, currentColumn));
            Advance();
            break;
        case '&':
            Advance();
            if (currentChar == '&')
            {
                tokens.Add(new Token(TokenType.AND, "&&", currentLine, currentColumn));
                Advance();
            }
            else
            {
                tokens.Add(new Token(TokenType.UNKNOWN, "&", currentLine, currentColumn));
            }
            break;
        case '|':
            Advance();
            if (currentChar == '|')
            {
                tokens.Add(new Token(TokenType.OR, "||", currentLine, currentColumn));
                Advance();
            }
            else
```

```
                {
                    tokens.Add(new Token(TokenType.UNKNOWN, "|", currentLine, currentColumn));
                }
                break;
            case ';':
                tokens.Add(new Token(TokenType.SEMICOLON, ";", currentLine, currentColumn));
                Advance();
                break;
            case '(':
                tokens.Add(new Token(TokenType.LEFT_PAREN, "(", currentLine, currentColumn));
                Advance();
                break;
            case ')':
                tokens.Add(new Token(TokenType.RIGHT_PAREN, ")", currentLine, currentColumn));
                Advance();
                break;
            case '{':
                tokens.Add(new Token(TokenType.LEFT_BRACE, "{", currentLine, currentColumn));
                Advance();
                break;
            case '}':
                tokens.Add(new Token(TokenType.RIGHT_BRACE, "}", currentLine, currentColumn));
                Advance();
                break;
            default:
                tokens.Add(new Token(TokenType.UNKNOWN, currentChar.ToString(), currentLine,
currentColumn));
                Advance();
                break;
        }
    }
}
tokens.Add(new Token(TokenType.EOF, "", line, column));
```

```
        return tokens;
    }
}
public class Parser
{
    private readonly List<Token> tokens;
    private int position;
    private Token currentToken;
    public List<string> Errors { get; private set; }
    public Parser(List<Token> tokens)
    {
        this.tokens = tokens;
        position = 0;
        currentToken = tokens[0];
        Errors = new List<string>();
    }
    private void Advance()
    {
        if (position < tokens.Count - 1)
        {
            position++;
            currentToken = tokens[position];
        }
    }
    private bool Match(TokenType type)
    {
        if (currentToken.Type == type)
        {
            Advance();
            return true;
        }
        return false;
    }
```

```
    private void Expect(TokenType type, string message)
    {
        if (!Match(type))
        {
            Errors.Add($"Line {currentToken.Line}, Col {currentToken.Column}: {message}. Found '{currentToken.Value}'");
            Advance();
        }
    }
    public Program ParseProgram()
    {
        var program = new Program();
        while (currentToken.Type != TokenType.EOF)
        {
            try
            {
                var stmt = ParseStatement();
                if (stmt != null)
                    program.Statements.Add(stmt);
                else
                    Advance();
            }
            catch (Exception ex)
            {
                Errors.Add($"Line {currentToken.Line}, Col {currentToken.Column}: {ex.Message}");
                while (currentToken.Type != TokenType.SEMICOLON && currentToken.Type != TokenType.EOF)
                    Advance();
                if (currentToken.Type == TokenType.SEMICOLON)
                    Advance();
            }
        }
        return program;
    }
```

```
    private Statement ParseStatement()
    {
        switch (currentToken.Type)
        {
            case TokenType.INT:
            case TokenType.FLOAT:
                return ParseDeclaration();
            case TokenType.IDENTIFIER:
                return ParseAssignment();
            case TokenType.PRINT:
                return ParsePrintStatement();
            case TokenType.IF:
                return ParseIfStatement();
            case TokenType.WHILE:
                return ParseWhileStatement();
            default:
                Errors.Add($"Line {currentToken.Line}, Col {currentToken.Column}: Unexpected token
'{currentToken.Value}'");
                return null;
        }
    }
    private Statement ParseDeclaration()
    {
        string type = null;
        if (Match(TokenType.INT))
            type = "int";
        else if (Match(TokenType.FLOAT))
            type = "float";
        else
        {
            Errors.Add($"Line {currentToken.Line}, Col {currentToken.Column}: Expected 'int' or 'float'");
            return null;
        }
```

```
        if (currentToken.Type == TokenType.IDENTIFIER)
        {
            string varName = currentToken.Value;
            Advance();
            Expect(TokenType.SEMICOLON, "Expected ';' after declaration");
            return new DeclarationStatement(varName, type);
        }
        Errors.Add($"Line {currentToken.Line}, Col {currentToken.Column}: Expected identifier after '{type}'");
        return null;
    }
    private Statement ParseAssignment()
    {
        string varName = currentToken.Value;
        Advance();
        Expect(TokenType.ASSIGN, "Expected '='");
        var expr = ParseExpression();
        Expect(TokenType.SEMICOLON, "Expected ';' after assignment");
        return new AssignmentStatement(varName, expr);
    }
    private Statement ParsePrintStatement()
    {
        Match(TokenType.PRINT);
        Expect(TokenType.LEFT_PAREN, "Expected '(' after 'print'");
        var expr = ParseExpression();
        Expect(TokenType.RIGHT_PAREN, "Expected ')' after expression");
        Expect(TokenType.SEMICOLON, "Expected ';' after print");
        return new PrintStatement(expr);
    }
    private Statement ParseIfStatement()
    {
        Match(TokenType.IF);
        Expect(TokenType.LEFT_PAREN, "Expected '(' after 'if'");
```

```
    var condition = ParseExpression();
    Expect(TokenType.RIGHT_PAREN, "Expected ')' after condition");
    var thenBlock = ParseBlock();
    List<Statement> elseBlock = null;
    if (currentToken.Type == TokenType.ELSE)
    {
        Match(TokenType.ELSE);
        elseBlock = ParseBlock();
    }
    return new IfStatement(condition, thenBlock, elseBlock);
}
private Statement ParseWhileStatement()
{
    Match(TokenType.WHILE);
    Expect(TokenType.LEFT_PAREN, "Expected '(' after 'while'");
    var condition = ParseExpression();
    Expect(TokenType.RIGHT_PAREN, "Expected ')' after condition");
    var body = ParseBlock();
    return new WhileStatement(condition, body);
}
private List<Statement> ParseBlock()
{
    var statements = new List<Statement>();
    Expect(TokenType.LEFT_BRACE, "Expected '{'");
    while (currentToken.Type != TokenType.RIGHT_BRACE && currentToken.Type != TokenType.EOF)
    {
        var stmt = ParseStatement();
        if (stmt != null)
            statements.Add(stmt);
        else
            Advance();
    }
    Expect(TokenType.RIGHT_BRACE, "Expected '}'");
```

```
            return statements;
        }
        private Expression ParseExpression(int depth = 0)
        {
            if (depth > 100)
            {
                Errors.Add($"Line {currentToken.Line}, Col {currentToken.Column}: Excessive recursion in expression");
                return new NumberExpression(0.0);
            }
            return ParseLogicalOr(depth + 1);
        }
        private Expression ParseLogicalOr(int depth)
        {
            var expr = ParseLogicalAnd(depth);
            while (currentToken.Type == TokenType.OR)
            {
                var op = currentToken.Type;
                Advance();
                var right = ParseLogicalAnd(depth + 1);
                expr = new BinaryExpression(expr, op, right);
            }
            return expr;
        }
        private Expression ParseLogicalAnd(int depth)
        {
            var expr = ParseComparison(depth);
            while (currentToken.Type == TokenType.AND)
            {
                var op = currentToken.Type;
                Advance();
                var right = ParseComparison(depth + 1);
                expr = new BinaryExpression(expr, op, right);
```

```
        }
        return expr;
    }
    private Expression ParseComparison(int depth)
    {
        var expr = ParseTerm(depth);
        while (currentToken.Type == TokenType.EQUAL || currentToken.Type == TokenType.NOT_EQUAL
||
               currentToken.Type == TokenType.LESS_THAN || currentToken.Type ==
TokenType.GREATER_THAN)
        {
            var op = currentToken.Type;
            Advance();
            var right = ParseTerm(depth + 1);
            expr = new BinaryExpression(expr, op, right);
        }
        return expr;
    }
    private Expression ParseTerm(int depth)
    {
        var expr = ParseFactor(depth);
        while (currentToken.Type == TokenType.PLUS || currentToken.Type == TokenType.MINUS)
        {
            var op = currentToken.Type;
            Advance();
            var right = ParseFactor(depth + 1);
            expr = new BinaryExpression(expr, op, right);
        }
        return expr;
    }
    private Expression ParseFactor(int depth)
    {
        var expr = ParsePrimary();
        while (currentToken.Type == TokenType.MULTIPLY || currentToken.Type == TokenType.DIVIDE)
```

```
        {
            var op = currentToken.Type;
            Advance();
            var right = ParsePrimary();
            expr = new BinaryExpression(expr, op, right);
        }
        return expr;
    }
    private Expression ParsePrimary()
    {
        if (currentToken.Type == TokenType.NUMBER)
        {
            if (double.TryParse(currentToken.Value, out double value))
            {
                Advance();
                return new NumberExpression(value);
            }
            Errors.Add($"Line {currentToken.Line}, Col {currentToken.Column}: Invalid number
'{currentToken.Value}'");
            Advance();
            return new NumberExpression(0.0);
        }
        else if (currentToken.Type == TokenType.IDENTIFIER)
        {
            string name = currentToken.Value;
            Advance();
            return new IdentifierExpression(name);
        }
        else if (currentToken.Type == TokenType.LEFT_PAREN)
        {
            Advance();
            var expr = ParseExpression();
            Expect(TokenType.RIGHT_PAREN, "Expected ')'");
```

```
            return expr;
        }
        Errors.Add($"Line {currentToken.Line}, Col {currentToken.Column}: Expected number, identifier, or '('");
        Advance();
        return new NumberExpression(0.0);
    }
}
// Semantic Analyzer
public class SemanticAnalyzer
{
    private readonly Dictionary<string, Symbol> symbolTable = new Dictionary<string, Symbol>();
    public List<string> Errors { get; private set; } = new List<string>();
    public List<string> Warnings { get; private set; } = new List<string>();

    public Dictionary<string, Symbol> Analyze(Program program)
    {
        symbolTable.Clear();
        Errors.Clear();
        Warnings.Clear();
        foreach (var stmt in program.Statements)
            AnalyzeStatement(stmt);
        foreach (var symbol in symbolTable.Values)
        {
            if (!symbol.IsInitialized)
                Warnings.Add($"Variable '{symbol.Name}' declared but not initialized");
        }
        return new Dictionary<string, Symbol>(symbolTable);
    }

    private void AnalyzeStatement(Statement stmt)
    {
        switch (stmt)
```

```
        {
            case DeclarationStatement decl:
                if (symbolTable.ContainsKey(decl.Variable))
                    Errors.Add($"Variable '{decl.Variable}' already declared");
                else
                    symbolTable[decl.Variable] = new Symbol(decl.Variable, decl.Type);
                break;
            case AssignmentStatement assign:
                if (!symbolTable.ContainsKey(assign.Variable))
                    Errors.Add($"Variable '{assign.Variable}' not declared");
                else
                {
                    string exprType = AnalyzeExpression(assign.Value);
                    if (symbolTable[assign.Variable].Type == "int" && exprType == "float")
                        Errors.Add($"Cannot assign float to int variable '{assign.Variable}'");
                    else
                    {
                        double value = EvaluateExpression(assign.Value);
                        symbolTable[assign.Variable].IsInitialized = true;
                        symbolTable[assign.Variable].Value = symbolTable[assign.Variable].Type == "int" ?
(int)value : value;
                    }
                }
                break;
            case PrintStatement print:
                AnalyzeExpression(print.Expression);
                break;
            case IfStatement ifStmt:
                AnalyzeExpression(ifStmt.Condition);
                foreach (var s in ifStmt.ThenBlock)
                    AnalyzeStatement(s);
                foreach (var s in ifStmt.ElseBlock)
                    AnalyzeStatement(s);
```

```
            break;
        case WhileStatement whileStmt:
            AnalyzeExpression(whileStmt.Condition);
            foreach (var s in whileStmt.Body)
                AnalyzeStatement(s);
            break;
    }
}

private string AnalyzeExpression(Expression expr)
{
    switch (expr)
    {
        case NumberExpression num:
            return num.Value % 1.0 == 0.0 ? "int" : "float";
        case IdentifierExpression id:
            if (!symbolTable.ContainsKey(id.Name))
            {
                Errors.Add($"Variable '{id.Name}' not declared");
                return "int";
            }
            else if (!symbolTable[id.Name].IsInitialized)
            {
                Errors.Add($"Variable '{id.Name}' used before initialization");
                return symbolTable[id.Name].Type;
            }
            return symbolTable[id.Name].Type;
        case BinaryExpression bin:
            string leftType = AnalyzeExpression(bin.Left);
            string rightType = AnalyzeExpression(bin.Right);
            if (leftType == "float" || rightType == "float")
                return "float";
            return "int";
```

```
        default:
            return "int";
    }
}

private double EvaluateExpression(Expression expr)
{
    switch (expr)
    {
        case NumberExpression num:
            return num.Value;
        case IdentifierExpression id:
            return symbolTable.ContainsKey(id.Name) && symbolTable[id.Name].IsInitialized
                ? symbolTable[id.Name].Value
                : 0.0;
        case BinaryExpression bin:
            double left = EvaluateExpression(bin.Left);
            double right = EvaluateExpression(bin.Right);
            switch (bin.Operator)
            {
                case TokenType.PLUS: return left + right;
                case TokenType.MINUS: return left - right;
                case TokenType.MULTIPLY: return left * right;
                case TokenType.DIVIDE: return right != 0 ? left / right : 0.0;
                case TokenType.EQUAL: return Math.Abs(left - right) < 0.000001 ? 1.0 : 0.0;
                case TokenType.NOT_EQUAL: return Math.Abs(left - right) >= 0.000001 ? 1.0 : 0.0;
                case TokenType.LESS_THAN: return left < right ? 1.0 : 0.0;
                case TokenType.GREATER_THAN: return left > right ? 1.0 : 0.0;
                case TokenType.AND: return (left != 0 && right != 0) ? 1.0 : 0.0;
                case TokenType.OR: return (left != 0 || right != 0) ? 1.0 : 0.0;
                default: return 0.0;
            }
        default:
```

```
            return 0.0;
        }
    }
}
// Optimizer
public class Optimizer
{
    public Expression OptimizeExpression(Expression expr)
    {
        if (expr is BinaryExpression bin)
        {
            bin.Left = OptimizeExpression(bin.Left);
            bin.Right = OptimizeExpression(bin.Right);
            if (bin.Left is NumberExpression left && bin.Right is NumberExpression right)
            {
                double result = 0.0;
                bool valid = true;
                switch (bin.Operator)
                {
                    case TokenType.PLUS: result = left.Value + right.Value; break;
                    case TokenType.MINUS: result = left.Value - right.Value; break;
                    case TokenType.MULTIPLY: result = left.Value * right.Value; break;
                    case TokenType.DIVIDE:
                        if (right.Value != 0) result = left.Value / right.Value;
                        else valid = false;
                        break;
                    default: valid = false; break;
                }
                if (valid) return new NumberExpression(result);
            }
        }
        return expr;
    }
```

```
    public void OptimizeProgram(Program program)
    {
        for (int i = 0; i < program.Statements.Count; i++)
        {
            if (program.Statements[i] is AssignmentStatement assign)
            {
                assign.Value = OptimizeExpression(assign.Value);
            }
            else if (program.Statements[i] is PrintStatement print)
            {
                print.Expression = OptimizeExpression(print.Expression);
            }
            else if (program.Statements[i] is IfStatement ifStmt)
            {
                ifStmt.Condition = OptimizeExpression(ifStmt.Condition);
            }
            else if (program.Statements[i] is WhileStatement whileStmt)
            {
                whileStmt.Condition = OptimizeExpression(whileStmt.Condition);
            }
        }
    }
}
// Intermediate Code Generator
public class IntermediateCodeGenerator
{
    private readonly List<string> code = new List<string>();
    private int tempCounter = 0;
    public List<string> Generate(Program program)
    {
        code.Clear();
        tempCounter = 0;
        foreach (var stmt in program.Statements)
```

```
            GenerateStatement(stmt);
        return code;
    }
    private string NewTemp()
    {
        return $"t{tempCounter++}";
    }
    private void GenerateStatement(Statement stmt)
    {
        switch (stmt)
        {
            case DeclarationStatement:
                break;
            case AssignmentStatement assign:
                string result = GenerateExpression(assign.Value);
                code.Add($"{assign.Variable} = {result}");
                break;
            case PrintStatement print:
                string exprResult = GenerateExpression(print.Expression);
                code.Add($"PRINT {exprResult}");
                break;
            case IfStatement ifStmt:
                string condResult = GenerateExpression(ifStmt.Condition);
                string labelElse = $"L{tempCounter++}";
                string labelEnd = $"L{tempCounter++}";
                code.Add($"IF {condResult} GOTO L{tempCounter}");
                code.Add($"GOTO {labelElse}");
                code.Add($"L{tempCounter++}:");
                foreach (var s in ifStmt.ThenBlock)
                    GenerateStatement(s);
                code.Add($"GOTO {labelEnd}");
                code.Add($"{labelElse}:");
                foreach (var s in ifStmt.ElseBlock)
```

```
                GenerateStatement(s);
            code.Add($"{labelEnd}:");
            break;
        case WhileStatement whileStmt:
            string labelStart = $"L{tempCounter++}";
            labelEnd = $"L{tempCounter++}";
            code.Add($"{labelStart}:");
            string cond = GenerateExpression(whileStmt.Condition);
            code.Add($"IF {cond} GOTO L{tempCounter}");
            code.Add($"GOTO {labelEnd}");
            code.Add($"L{tempCounter++}:");
            foreach (var s in whileStmt.Body)
                GenerateStatement(s);
            code.Add($"GOTO {labelStart}");
            code.Add($"{labelEnd}:");
            break;
    }
}
private string GenerateExpression(Expression expr)
{
    switch (expr)
    {
        case NumberExpression num:
            return num.Value.ToString(System.Globalization.CultureInfo.InvariantCulture);
        case IdentifierExpression id:
            return id.Name;
        case BinaryExpression bin:
            string left = GenerateExpression(bin.Left);
            string right = GenerateExpression(bin.Right);
            string temp = NewTemp();
            string op = bin.Operator switch
            {
                TokenType.PLUS => "+",
```

```
                TokenType.MINUS => "-",
                TokenType.MULTIPLY => "*",
                TokenType.DIVIDE => "/",
                TokenType.EQUAL => "==",
                TokenType.NOT_EQUAL => "!=",
                TokenType.LESS_THAN => "<",
                TokenType.GREATER_THAN => ">",
                TokenType.AND => "&&",
                TokenType.OR => "||",
                _ => throw new Exception($"Unsupported operator {bin.Operator}")
            };
            code.Add($"{temp} = {left} {op} {right}");
            return temp;
        default:
            throw new Exception("Unknown expression type");
    }
  }
}
// Target Code Generator
public class TargetCodeGenerator
{
  private readonly List<string> instructions = new List<string>();
  private readonly Dictionary<string, Symbol> symbolTable;
  private int labelCounter = 0;

  public TargetCodeGenerator(Dictionary<string, Symbol> symbolTable)
  {
    this.symbolTable = symbolTable;
  }

  public List<string> Generate(Program program)
  {
    instructions.Clear();
```

```
    instructions.Add("# Stack-based VM Code");
    foreach (var stmt in program.Statements)
        GenerateStatement(stmt);
    instructions.Add("HALT");
    return instructions;
}
private void GenerateStatement(Statement stmt)
{
    switch (stmt)
    {
        case DeclarationStatement decl:
            instructions.Add($"ALLOC {decl.Variable}");
            break;
        case AssignmentStatement assign:
            GenerateExpression(assign.Value);
            instructions.Add($"STORE {assign.Variable}");
            break;
        case PrintStatement print:
            GenerateExpression(print.Expression);
            instructions.Add("PRINT");
            break;
        case IfStatement ifStmt:
            GenerateExpression(ifStmt.Condition);
            string elseLabel = $"ELSE_{labelCounter++}";
            string endLabel = $"END_{labelCounter++}";
            instructions.Add($"JZ {elseLabel}");
            foreach (var s in ifStmt.ThenBlock)
                GenerateStatement(s);
            instructions.Add($"JMP {endLabel}");
            instructions.Add($"{elseLabel}:");
            foreach (var s in ifStmt.ElseBlock)
                GenerateStatement(s);
            instructions.Add($"{endLabel}:");
```

```
                break;
            case WhileStatement whileStmt:
                string startLabel = $"LOOP_{labelCounter++}";
                string exitLabel = $"EXIT_{labelCounter++}";
                instructions.Add($"{startLabel}:");
                GenerateExpression(whileStmt.Condition);
                instructions.Add($"JZ {exitLabel}");
                foreach (var s in whileStmt.Body)
                    GenerateStatement(s);
                instructions.Add($"JMP {startLabel}");
                instructions.Add($"{exitLabel}:");
                break;
        }
    }
    private void GenerateExpression(Expression expr)
    {
        switch (expr)
        {
            case NumberExpression num:
                instructions.Add($"PUSH
{num.Value.ToString(System.Globalization.CultureInfo.InvariantCulture)}");
                break;
            case IdentifierExpression id:
                instructions.Add($"LOAD {id.Name}");
                break;
            case BinaryExpression bin:
                GenerateExpression(bin.Left);
                GenerateExpression(bin.Right);
                switch (bin.Operator)
                {
                    case TokenType.PLUS: instructions.Add("ADD"); break;
                    case TokenType.MINUS: instructions.Add("SUB"); break;
                    case TokenType.MULTIPLY: instructions.Add("MUL"); break;
```

```
                    case TokenType.DIVIDE: instructions.Add("DIV"); break;
                    case TokenType.EQUAL: instructions.Add("EQ"); break;
                    case TokenType.NOT_EQUAL: instructions.Add("NEQ"); break;
                    case TokenType.LESS_THAN: instructions.Add("LT"); break;
                    case TokenType.GREATER_THAN: instructions.Add("GT"); break;
                    case TokenType.AND: instructions.Add("AND"); break;
                    case TokenType.OR: instructions.Add("OR"); break;
                }
                break;
            }
        }
    }
    // Console Interface
    public class CompilerConsole
    {
        private const string SAMPLE_CODE = @"// Sample Program
int x;
float y;
x = 10;
y = 20.5;
int result;
result = x + y * 2; // No optimization due to mixed types
print(result);
if (x < y && y > 15.0) {
    print(1);
} else {
    print(0);
}
int i;
i = 0;
while (i < 3) {
    print(i);
    i = i + 1;
```

```
}";
        public static void Main(string[] args)
        {
            while (true)
            {
                ShowMenu();
                string choice = Console.ReadLine()?.Trim();
                Console.WriteLine();
                switch (choice)
                {
                    case "1":
                        CompileInteractive();
                        break;
                    case "2":
                        CompileSampleCode();
                        break;
                    case "3":
                        ShowLanguageReference();
                        break;
                    case "4":
                        Console.ForegroundColor = ConsoleColor.Green;
                        Console.WriteLine("👋 Thank you for using Mini Compiler! Exiting...");
                        Console.ResetColor();
                        return;
                    default:
                        Console.ForegroundColor = ConsoleColor.Red;
                        Console.WriteLine("❌ Invalid choice. Please enter 1-4.");
                        Console.ResetColor();
                        break;
                }
                Console.WriteLine("\nPress any key to continue...");
                Console.ReadKey();
                Console.Clear();
```

```
        }
    }
    private static void ShowMenu()
    {
        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.WriteLine("
╔═══════════════════════════════════════════════════════════════════════════════╗");
        Console.WriteLine("                         ║    * STARTING COMPILATION PROCESS *  ║");
        Console.WriteLine("
╚═══════════════════════════════════════════════════════════════════════════════╝");
        Console.ResetColor();
        Console.WriteLine("\n Main Menu:");
        Console.WriteLine(" 1 - Enter Custom Code");
        Console.WriteLine("       - Write your own program (end with 'END')");
        Console.WriteLine(" 2 - Run Sample Program");
        Console.WriteLine("       - Test with a pre-written example");
        Console.WriteLine(" 3 - Language Reference");
        Console.WriteLine("       - View syntax rules and examples");
        Console.WriteLine(" 4 - Exit Compiler");
        Console.Write("\n Enter your choice (1-4): ");
    }
    private static void CompileInteractive()
    {
        Console.WriteLine(" Interactive Code Entry");
        Console.WriteLine("Enter your code (type 'END' on a new line to finish):");
        Console.WriteLine(new string('-', 50));
        var codeBuilder = new StringBuilder();
        string line;
        while ((line = Console.ReadLine()) != null)
        {
            if (line.Trim().ToUpper() == "END")
                break;
            codeBuilder.AppendLine(line);
        }
```

```
    string code = codeBuilder.ToString();
    if (string.IsNullOrWhiteSpace(code))
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("\n No code entered.");
        Console.ResetColor();
        return;
    }
    Console.ForegroundColor = ConsoleColor.Green;
    Console.WriteLine("\nCode entered successfully!");
    Console.ResetColor();
    CompileCode(code);
}
private static void CompileSampleCode()
{
    Console.WriteLine("\n Compiling Sample Code");
    Console.WriteLine("Sample code:");
    Console.WriteLine(new string('-', 50));
    Console.WriteLine(SAMPLE_CODE);
    Console.WriteLine(new string('-', 50));
    CompileCode(SAMPLE_CODE);
}

private static void CompileCode(string sourceCode)
{
    Console.WriteLine("        ╔═══════════════════════════════════════════════════╗");
    Console.WriteLine("        ║    STARTING COMPILATION PROCESS       ║");
    Console.WriteLine("        ╚═══════════════════════════════════════════════════╝");
    // Lexical Analysis
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine("\n * PHASE 1 : LEXICAL ANALYSIS! * ");
    Console.ResetColor();
    Console.WriteLine(new string('=', 40));
```

```
var lexerStopwatch = System.Diagnostics.Stopwatch.StartNew();
var lexer = new Lexer(sourceCode);
var tokens = lexer.Tokenize();
lexerStopwatch.Stop();
Console.WriteLine($" Took {lexerStopwatch.ElapsedMilliseconds} ms");
Console.WriteLine($"Generated {tokens.Count} tokens:");
Console.WriteLine($"{"Type",-12} | {"Value",-8} | Position");
Console.WriteLine(new string('-', 35));
foreach (var token in tokens.Take(20))
    Console.WriteLine(token);
if (tokens.Count > 20)
    Console.WriteLine($"... and {tokens.Count - 20} more tokens");
// Syntax Analysis
Console.ForegroundColor = ConsoleColor.Yellow;
Console.WriteLine("\n * PHASE 2 : SYNTAX ANALYSIS! * ");
Console.WriteLine(new string('=', 40));
var parserStopwatch = System.Diagnostics.Stopwatch.StartNew();
var parser = new Parser(tokens);
var ast = parser.ParseProgram();
parserStopwatch.Stop();
Console.WriteLine($" Took {parserStopwatch.ElapsedMilliseconds} ms");
if (parser.Errors.Any())
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(" Syntax Errors:");
    foreach (var error in parser.Errors)
        Console.WriteLine($"  {error}");
    Console.ResetColor();
}
else
{
    Console.ForegroundColor = ConsoleColor.Green;
    Console.WriteLine("\n Syntax analysis completed!");
```

```
    Console.ResetColor();
    Console.WriteLine("\nAbstract Syntax Tree:");
    Console.WriteLine(new string('-', 30));
    Console.WriteLine(ast.ToStringIndented());
}
// Semantic Analysis
Console.ForegroundColor = ConsoleColor.Yellow;
Console.WriteLine("\n * PHASE 3 : SEMANTIC ANALYSIS! * ");
Console.WriteLine(new string('=', 40));
var semanticStopwatch = System.Diagnostics.Stopwatch.StartNew();
var analyzer = new SemanticAnalyzer();
var symbolTable = analyzer.Analyze(ast);
semanticStopwatch.Stop();
Console.WriteLine($" Took {semanticStopwatch.ElapsedMilliseconds} ms");
if (analyzer.Errors.Any())
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(" Semantic Errors:");
    foreach (var error in analyzer.Errors)
        Console.WriteLine($"  {error}");
    Console.ResetColor();
}
else
{
    Console.ForegroundColor = ConsoleColor.Green;
    Console.WriteLine("\n Semantic analysis completed!");
    Console.ResetColor();
}
if (analyzer.Warnings.Any())
{
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine("\n Warnings:");
    foreach (var warning in analyzer.Warnings)
```

```
        Console.WriteLine($"  {warning}");
    Console.ResetColor();
}
Console.WriteLine("\nSymbol Table:");
Console.WriteLine($"{"Name",-10} | {"Type",-6} | {"Init",-5} | Value");
Console.WriteLine(new string('-', 35));
foreach (var symbol in symbolTable.Values)
    Console.WriteLine(symbol);
// Optimization
Console.ForegroundColor = ConsoleColor.Yellow;
Console.WriteLine("\n * PHASE 4 : OPTIMIZATION! * ");
Console.WriteLine(new string('=', 40));
var optimizeStopwatch = System.Diagnostics.Stopwatch.StartNew();
var optimizer = new Optimizer();
optimizer.OptimizeProgram(ast);
optimizeStopwatch.Stop();
Console.WriteLine($" Took {optimizeStopwatch.ElapsedMilliseconds} ms");
Console.ForegroundColor = ConsoleColor.Green;
Console.WriteLine("\n Optimization completed!");
Console.ResetColor();
Console.WriteLine("\nOptimized AST:");
Console.WriteLine(new string('-', 30));
Console.WriteLine(ast.ToStringIndented());
// Intermediate Code Generation
Console.ForegroundColor = ConsoleColor.Yellow;
Console.WriteLine("\n * PHASE 5 : INTERMEDIATE CODE GENERATION * ");
Console.WriteLine(new string('=', 40));
if (parser.Errors.Any() || analyzer.Errors.Any())
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(" Skipped due to errors.");
    Console.ResetColor();
}
```

```
else
{
    var irStopwatch = System.Diagnostics.Stopwatch.StartNew();
    var irGenerator = new IntermediateCodeGenerator();
    var irCode = irGenerator.Generate(ast);
    irStopwatch.Stop();
    Console.WriteLine($" Took {irStopwatch.ElapsedMilliseconds} ms");
    Console.ForegroundColor = ConsoleColor.Green;
    Console.WriteLine("\n Intermediate code generated!");
    Console.ResetColor();
    Console.WriteLine("\n  Three-Address Code:  ");
    Console.WriteLine(new string('-', 35));
    for (int i = 0; i < irCode.Count; i++)
        Console.WriteLine($"{i + 1,3}: {irCode[i]}");
}
// Target Code Generation
Console.ForegroundColor = ConsoleColor.Yellow;
Console.WriteLine("\n * PHASE 6 : TARGET CODE GENERATION * ");
Console.WriteLine(new string('=', 40));
if (parser.Errors.Any() || analyzer.Errors.Any())
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(" Skipped due to errors.");
    Console.ResetColor();
}
else
{
    var targetStopwatch = System.Diagnostics.Stopwatch.StartNew();
    var targetGenerator = new TargetCodeGenerator(symbolTable);
    var targetCode = targetGenerator.Generate(ast);
    targetStopwatch.Stop();
    Console.WriteLine($" Took {targetStopwatch.ElapsedMilliseconds} ms");
    Console.ForegroundColor = ConsoleColor.Green;
```

```
            Console.WriteLine("\nTarget code generated!");
            Console.ResetColor();
            Console.WriteLine("\nStack-based VM Code:");
            Console.WriteLine(new string('-', 35));
            for (int i = 0; i < targetCode.Count; i++)
                Console.WriteLine($"{i + 1,3}: {targetCode[i]}");
        }
        // Compilation Summary
        Console.WriteLine("\n * COMPILATION SUMMARY * ");
        Console.WriteLine(new string('=', 40));
        Console.WriteLine($"Tokens generated: {tokens.Count}");
        Console.WriteLine($"Syntax errors: {parser.Errors.Count}");
        Console.WriteLine($"Semantic errors: {analyzer.Errors.Count}");
        Console.WriteLine($"Warnings: {analyzer.Warnings.Count}");
        Console.WriteLine($"Variables declared: {symbolTable.Count}");
        Console.ForegroundColor = parser.Errors.Any() || analyzer.Errors.Any() ? ConsoleColor.Red :
ConsoleColor.Green;
        Console.WriteLine(parser.Errors.Any() || analyzer.Errors.Any()
            ? " Compilation failed with errors."
            : " Compilation completed successfully!");
        Console.ResetColor();
    }
    private static void ShowLanguageReference()
    {
        Console.ForegroundColor = ConsoleColor.Magenta;
        Console.WriteLine(" Mini Language Reference");
        Console.ResetColor();
        Console.WriteLine(new string('=', 50));
        Console.WriteLine("\n Keywords:");
        Console.WriteLine("  int    - Declare integer variable");
        Console.WriteLine("  float  - Declare floating-point variable");
        Console.WriteLine("  if     - Conditional statement");
        Console.WriteLine("  else   - Alternative condition");
```

```
        Console.WriteLine("  while  - Loop statement");
        Console.WriteLine("  print  - Output expression");
        Console.WriteLine("\n Operators:");
        Console.WriteLine("  +      - Addition");
        Console.WriteLine("  -      - Subtraction");
        Console.WriteLine("  *      - Multiplication");
        Console.WriteLine("  /      - Division");
        Console.WriteLine("  =      - Assignment");
        Console.WriteLine("  ==     - Equality");
        Console.WriteLine("  !=     - Inequality");
        Console.WriteLine("  <      - Less than");
        Console.WriteLine("  >      - Greater than");
        Console.WriteLine("  &&     - Logical AND");
        Console.WriteLine("  ||     - Logical OR");
        Console.WriteLine("\n Syntax Examples:");
        Console.WriteLine("  Declaration:    int x; float y;");
        Console.WriteLine("  Assignment:     x = 10; y = 3.14;");
        Console.WriteLine("  Print:          print(x);");
        Console.WriteLine("  If-Else:        if (x > 5.0) { print(1); } else { print(0); }");
        Console.WriteLine("  While:          while (x < 10) { x = x + 1; }");
        Console.WriteLine("\n Notes:");
        Console.WriteLine("  • Statements end with ';'");
        Console.WriteLine("  • Blocks use '{' and '}'");
        Console.WriteLine("  • Comments start with '//'");
        Console.WriteLine("  • Integers and floating-point numbers are supported");
        Console.WriteLine("  • Cannot assign float to int without explicit casting");
    }
  }
}
```

OUTPUT:

```
                    * STARTING COMPILATION PROCESS *


Main Menu:
1 - Enter Custom Code
      - Write your own program (end with 'END')
2 - Run Sample Program
      - Test with a pre-written example
3 - Language Reference
      - View syntax rules and examples
4 - Exit Compiler

Enter your choice (1-4): 4

?? Thank you for using Mini Compiler! Exiting...
```

```
 Interactive Code Entry
Enter your code (type 'END' on a new line to finish):
--------------------------------------------------
int a;
float b;
a = 2.3;
print(a);
END

Code entered successfully!

          STARTING COMPILATION PROCESS


 * PHASE 1 : LEXICAL ANALYSIS! *
========================================
 Took 0 ms
Generated 16 tokens:
Type          | Value    | Position
-----------------------------------
INT           | int      | Line  1, Col  1
IDENTIFIER    | a        | Line  1, Col  5
SEMICOLON     | ;        | Line  1, Col  6
FLOAT         | float    | Line  2, Col  1
IDENTIFIER    | b        | Line  2, Col  7
SEMICOLON     | ;        | Line  2, Col  8
IDENTIFIER    | a        | Line  3, Col  1
ASSIGN        | =        | Line  3, Col  3
NUMBER        | 2.3      | Line  3, Col  5
SEMICOLON     | ;        | Line  3, Col  8
PRINT         | print    | Line  4, Col  1
LEFT_PAREN    | (        | Line  4, Col  6
IDENTIFIER    | a        | Line  4, Col  7
RIGHT_PAREN   | )        | Line  4, Col  8
SEMICOLON     | ;        | Line  4, Col  9
EOF           |          | Line  5, Col  1
```

```
 * PHASE 2 : SYNTAX ANALYSIS! *
========================================
 Took 0 ms

 Syntax analysis completed!

Abstract Syntax Tree:
-----------------------------
Program:
  Declaration: int a
  Declaration: float b
  Assignment: a
    Number: 2.3
  Print:
    Identifier: a
```

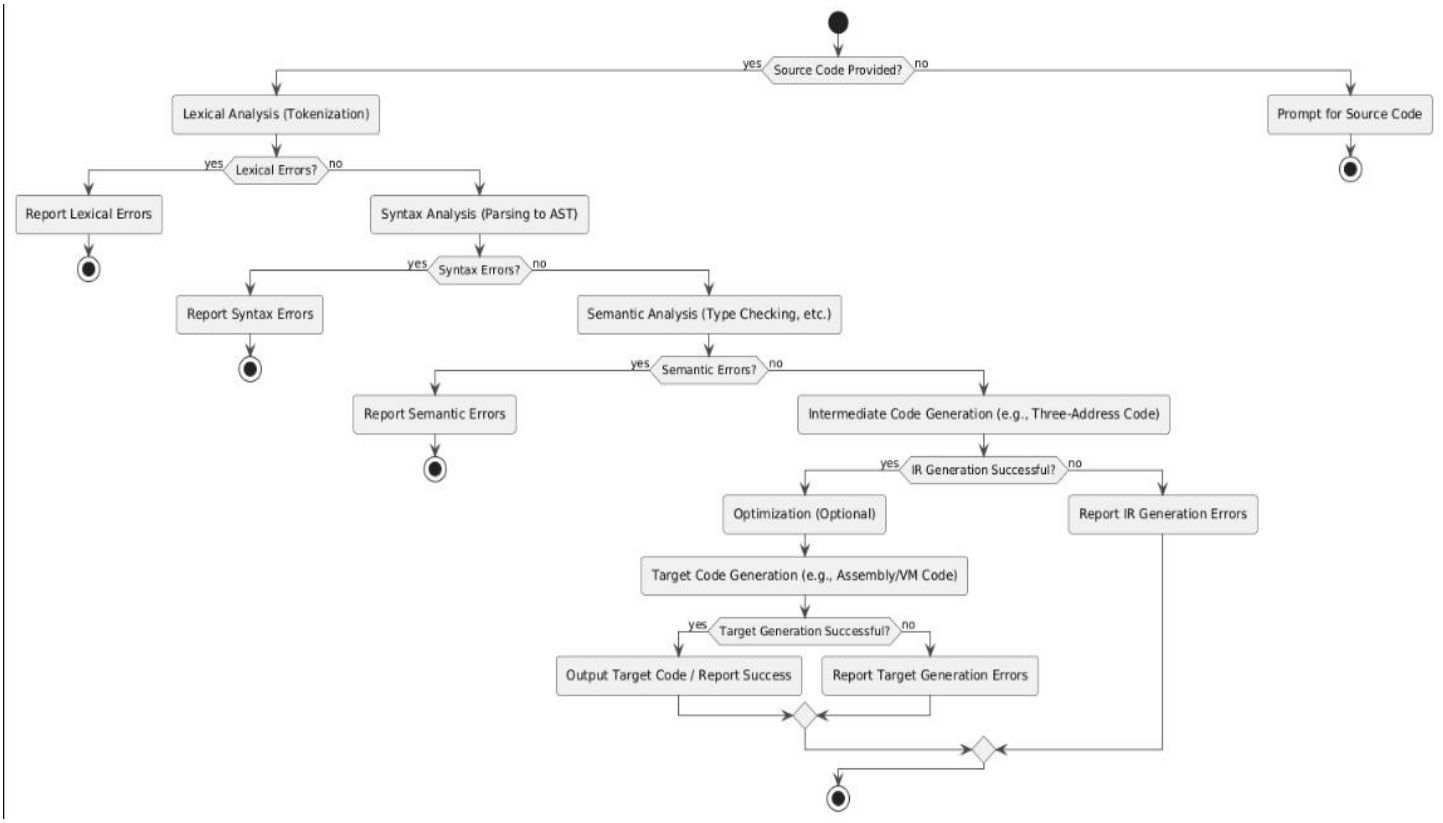
```
 * PHASE 3 : SEMANTIC ANALYSIS! *
========================================
 Took 0 ms

 Semantic analysis completed!

Symbol Table:
Name         | Type    | Init   | Value
-------------------------------------
a            | int     | Yes    | 23
b            | float   | Yes    | 3.13
```

```
 * PHASE 4 : OPTIMIZATION! *
========================================
 Took 0 ms

 Optimization completed!

Optimized AST:
-----------------------------
Program:
  Declaration: int a
  Declaration: float b
  Assignment: a
    Number: 23
  Assignment: b
    Number: 3.13
  Print:
    Identifier: a
```

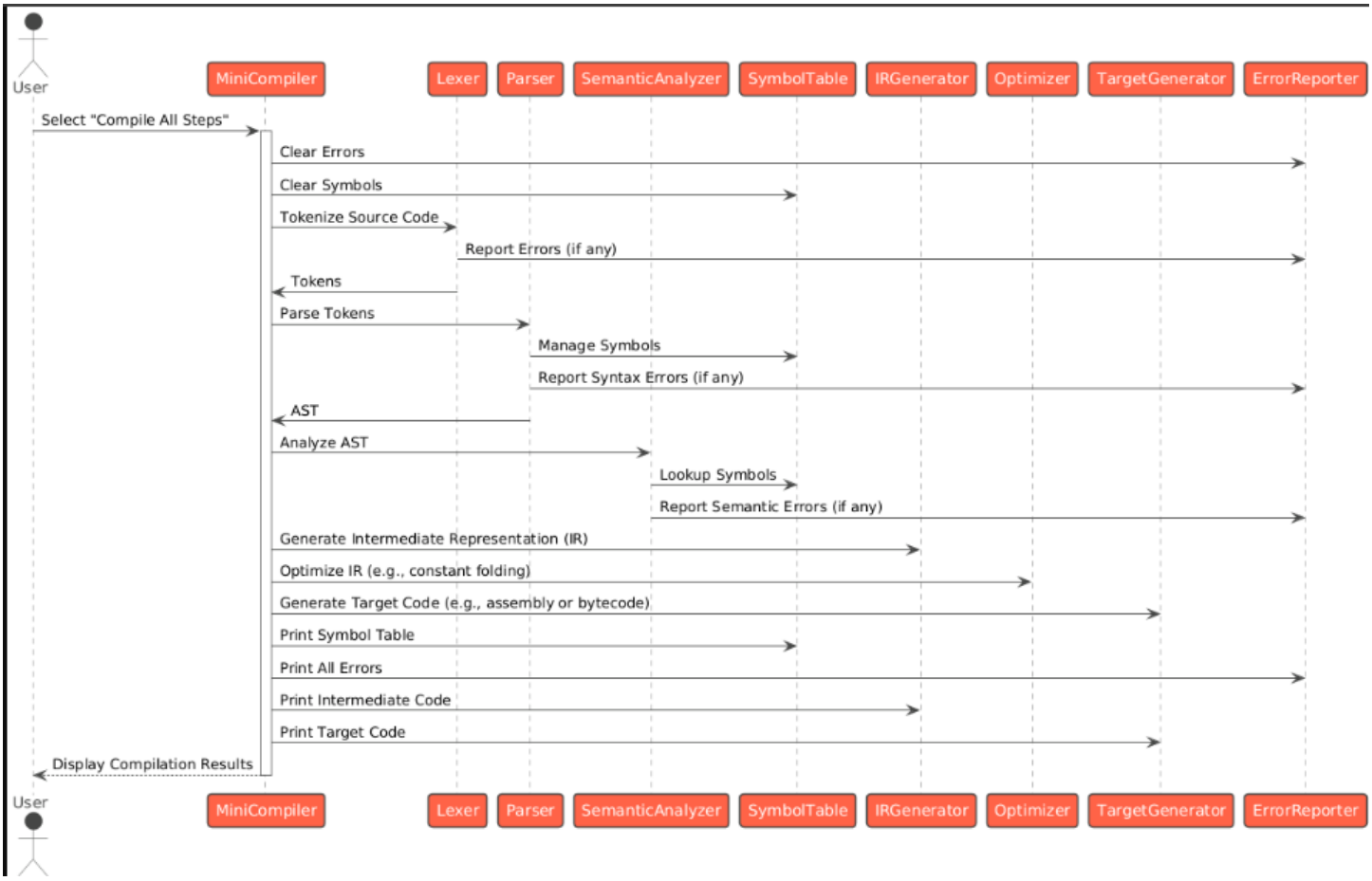
```
 * PHASE 5 : INTERMEDIATE CODE GENERATION *
==========================================
 Took 0 ms

 Intermediate code generated!

  Three-Address Code:
----------------------------------
  1: a = 23
  2: b = 3.13
  3: PRINT a
```

```
 * PHASE 6 : TARGET CODE GENERATION *
==========================================
 Took 0 ms

Target code generated!

Stack-based VM Code:
----------------------------------
  1: # Stack-based VM Code
  2: ALLOC a
  3: ALLOC b
  4: PUSH 23
  5: STORE a
  6: PUSH 3.13
  7: STORE b
  8: LOAD a
  9: PRINT
 10: HALT

 * COMPILATION SUMMARY *
==========================================
Tokens generated: 20
Syntax errors: 0
Semantic errors: 0
Warnings: 0
Variables declared: 2
 Compilation completed successfully!
```

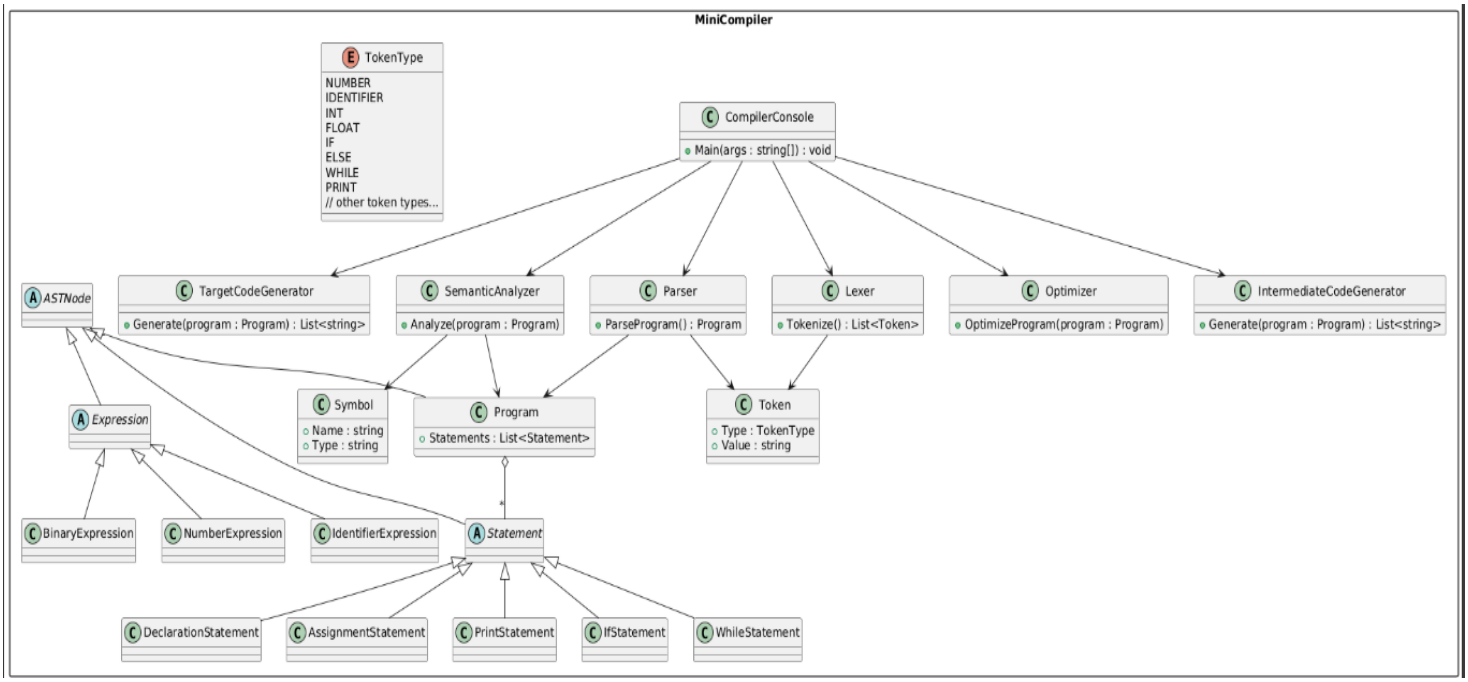
Activity Diagram:



Sequence Diagram:



Class Diagram:



Testing:

```
 Interactive Code Entry
Enter your code (type 'END' on a new line to finish):
------------------------------------------------
int a;
float b;
a = 2.3;
print(a);
END

Code entered successfully!

          STARTING COMPILATION PROCESS


 * PHASE 1 : LEXICAL ANALYSIS! *
========================================
 Took 0 ms
Generated 16 tokens:
Type          | Value    | Position
---------------------------------
INT           | int      | Line  1, Col  1
IDENTIFIER    | a        | Line  1, Col  5
SEMICOLON     | ;        | Line  1, Col  6
FLOAT         | float    | Line  2, Col  1
IDENTIFIER    | b        | Line  2, Col  7
SEMICOLON     | ;        | Line  2, Col  8
IDENTIFIER    | a        | Line  3, Col  1
ASSIGN        | =        | Line  3, Col  3
NUMBER        | 2.3      | Line  3, Col  5
SEMICOLON     | ;        | Line  3, Col  8
PRINT         | print    | Line  4, Col  1
LEFT_PAREN    | (        | Line  4, Col  6
IDENTIFIER    | a        | Line  4, Col  7
RIGHT_PAREN   | )        | Line  4, Col  8
SEMICOLON     | ;        | Line  4, Col  9
EOF           |          | Line  5, Col  1
```

```
 * PHASE 2 : SYNTAX ANALYSIS! *
========================================
 Took 0 ms

 Syntax analysis completed!

Abstract Syntax Tree:
------------------------------
Program:
  Declaration: int a
  Declaration: float b
  Assignment: a
    Number: 2.3
  Print:
    Identifier: a
```

```
 * PHASE 3 : SEMANTIC ANALYSIS! *
========================================
 Took 0 ms
 Semantic Errors:
  Cannot assign float to int variable 'a'
  Variable 'a' used before initialization

 Warnings:
  Variable 'a' declared but not initialized
  Variable 'b' declared but not initialized

Symbol Table:
Name         | Type    | Init   | Value
-----------------------------------
a            | int     | No     | 0
b            | float   | No     | 0
```

```
 * PHASE 4 : OPTIMIZATION! *
========================================
 Took 0 ms

 Optimization completed!

Optimized AST:
------------------------------
Program:
  Declaration: int a
  Declaration: float b
  Assignment: a
    Number: 2.3
  Print:
    Identifier: a
```

```
 * PHASE 5 : INTERMEDIATE CODE GENERATION *
========================================
 Skipped due to errors.

 * PHASE 6 : TARGET CODE GENERATION *
========================================
 Skipped due to errors.

 * COMPILATION SUMMARY *
========================================
Tokens generated: 16
Syntax errors: 0
Semantic errors: 2
Warnings: 2
Variables declared: 2
 Compilation failed with errors.
```