COMSATS UNIVERSITY ISLAMABAD

ATTOCK CAMPUS

DEPARTMENT OF COMPUTER SCIENCE

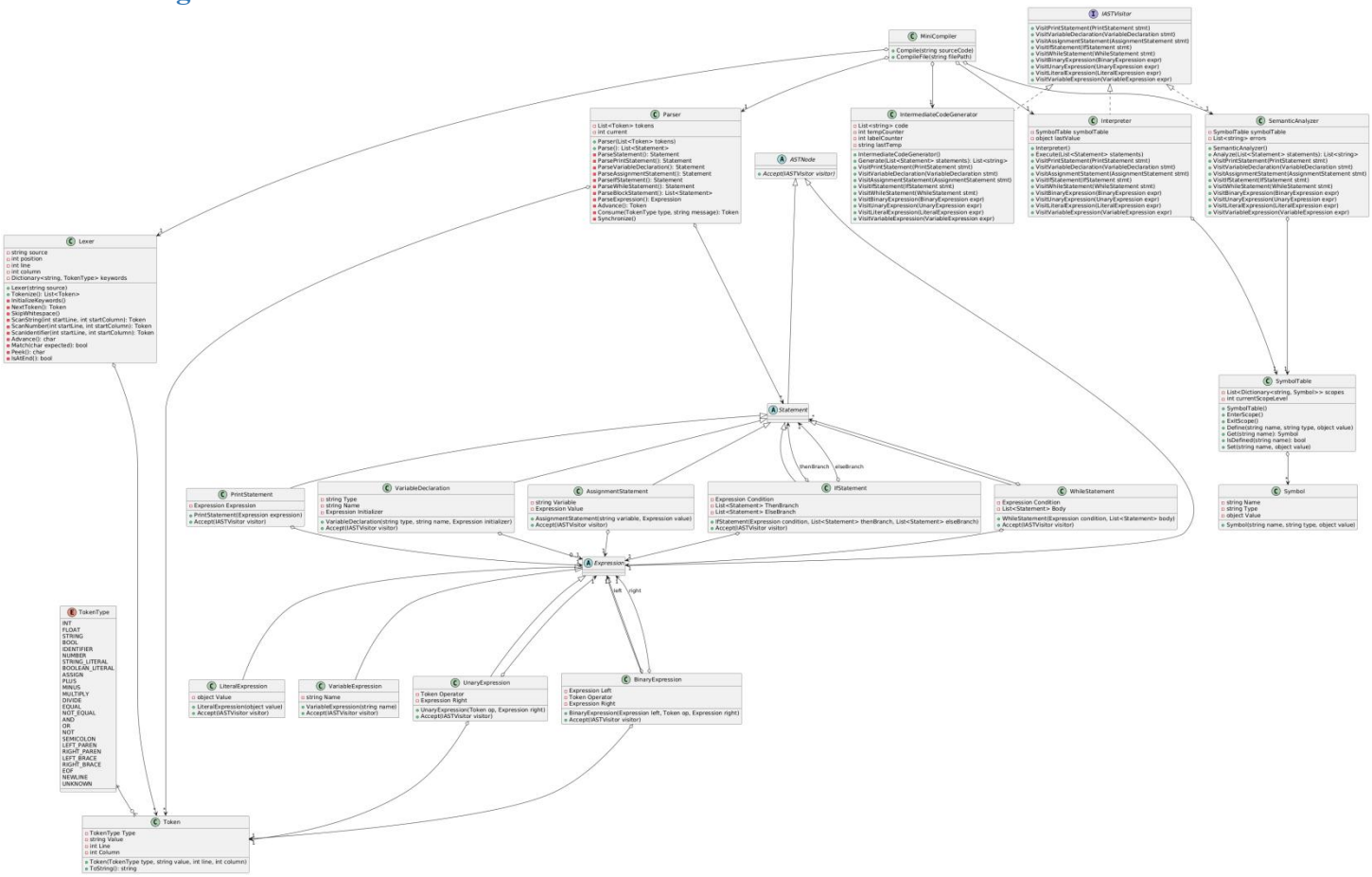
Mini Compiler

NAME	MUHAMMAD FAHAD QASIM
REGISTRATION	SP22-BCS-034
SUBJECT	COMPILER CONSTRCUTION
LAB	SEMSTER PROJECT
SUBMITTED TO	SIR BILAL BUKHARI

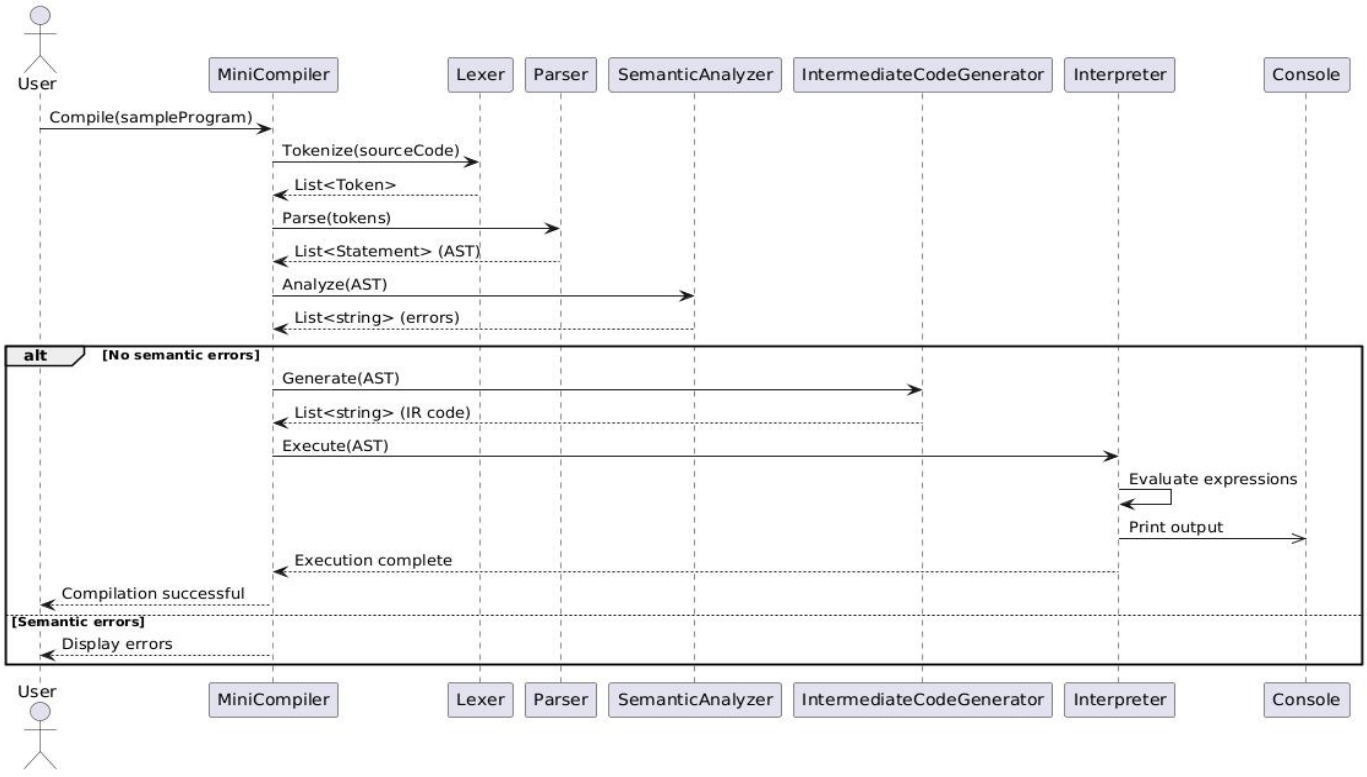
May 30, 2025

Mini Compiler

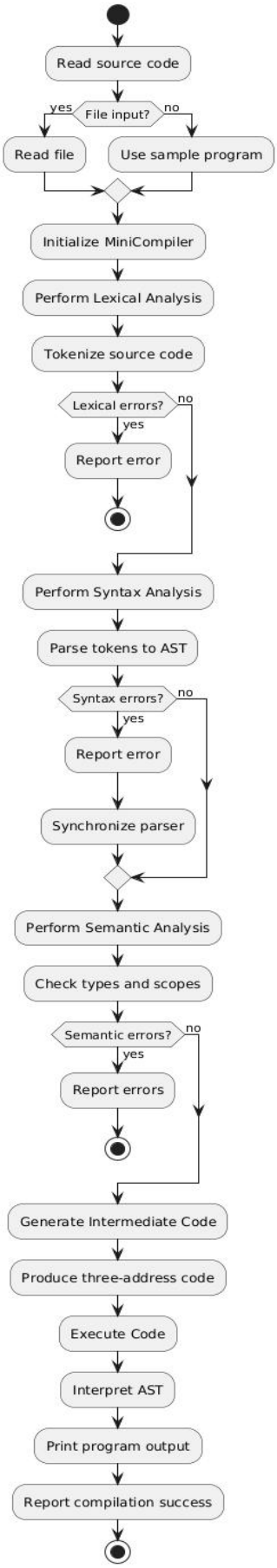
✓ Class Diagram



✓ Sequence Diagram



✓ Activity Diagram



Code

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
// Token Types for the Lexical Analyzer
public enum TokenType
{
    // Keywords
    INT, FLOAT, STRING, BOOL, IF, ELSE, WHILE, FOR, RETURN, PRINT,

    // Identifiers and Literals
    IDENTIFIER, NUMBER, STRING_LITERAL, BOOLEAN_LITERAL,

    // Operators
    ASSIGN, PLUS, MINUS, MULTIPLY, DIVIDE, MODULO,
    EQUAL, NOT_EQUAL, LESS_THAN, GREATER_THAN, LESS_EQUAL, GREATER_EQUAL,
    AND, OR, NOT,

    // Delimiters
    SEMICOLON, COMMA, LEFT_PAREN, RIGHT_PAREN, LEFT_BRACE, RIGHT_BRACE,

    // Special
    EOF, NEWLINE, UNKNOWN
}
```

```
// Token class
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 $"Token({Type}, '{Value}', {Line}:{Column})";
    }
}
```

```
// Lexical Analyzer (Scanner)
public class Lexer
{
    private string source;
    private int position;
    private int line;
    private int column;
    private Dictionary<string, TokenType> keywords;
```

```
    public Lexer(string source)
    {
        this.source = source;
        this.position = 0;
        this.line = 1;
        this.column = 1;

        InitializeKeywords();
    }
```

```
    private void InitializeKeywords()
    {
        keywords = new Dictionary<string, TokenType>
        {
            {"int", TokenType.INT},
            {"float", TokenType.FLOAT},
            {"string", TokenType.STRING},
            {"bool", TokenType.BOOL},
            {"if", TokenType.IF},
            {"else", TokenType.ELSE},
            {"while", TokenType.WHILE},
            {"for", TokenType.FOR},
            {"return", TokenType.RETURN},
            {"print", TokenType.PRINT},
            {"true", TokenType.BOOLEAN_LITERAL},
            {"false", TokenType.BOOLEAN_LITERAL}
        };
    }
```

```
    public List<Token> Tokenize()
    {
        List<Token> tokens = new List<Token>();

        while (!IsAtEnd())
        {
            Token token = NextToken();
            if (token != null && token.Type != TokenType.NEWLINE)
            {
                tokens.Add(token);
            }
        }

        tokens.Add(new Token(TokenType.EOF, "", line, column));
        return tokens;
    }
```

```
    private Token NextToken()
    {
        SkipWhitespace();

        if (IsAtEnd()) return null;
```

```
        int startLine = line;
        int startColumn = column;
        char c = Advance();
```

```
        // Single character tokens
        switch (c)
        {
            case '(': return new Token(TokenType.LEFT_PAREN, "(", startLine, startColumn);
            case ')': return new Token(TokenType.RIGHT_PAREN, ")", startLine, startColumn);
            case '{': return new Token(TokenType.LEFT_BRACE, "{", startLine, startColumn);
```

```
            case '}': return new Token(TokenType.RIGHT_BRACE, "}", startLine, startColumn);
            case ';': return new Token(TokenType.SEMICOLON, ";", startLine, startColumn);
            case ',': return new Token(TokenType.COMMA, ",", startLine, startColumn);
            case '+': return new Token(TokenType.PLUS, "+", startLine, startColumn);
            case '-': return new Token(TokenType.MINUS, "-", startLine, startColumn);
            case '*': return new Token(TokenType.MULTIPLY, "*", startLine, startColumn);
            case '/': return new Token(TokenType.DIVIDE, "/", startLine, startColumn);
            case '%': return new Token(TokenType.MODULO, "%", startLine, startColumn);
            case '\n':
                line++;
                column = 1;
                return new Token(TokenType.NEWLINE, "\\n", startLine, startColumn);
        }
```

```
        // Two character tokens
        if (c == '=')
        {
            if (Match('='))
                return new Token(TokenType.EQUAL, "==", startLine, startColumn);
            else
                return new Token(TokenType.ASSIGN, "=", startLine, startColumn);
        }

        if (c == '!')
        {
            if (Match('='))
                return new Token(TokenType.NOT_EQUAL, "!=", startLine, startColumn);
            else
                return new Token(TokenType.NOT, "!", startLine, startColumn);
        }

        if (c == '<')
        {
            if (Match('='))
                return new Token(TokenType.LESS_EQUAL, "<=", startLine, startColumn);
            else
                return new Token(TokenType.LESS_THAN, "<", startLine, startColumn);
        }

        if (c == '>')
        {
            if (Match('='))
                return new Token(TokenType.GREATER_EQUAL, ">=", startLine, startColumn);
            else
                return new Token(TokenType.GREATER_THAN, ">", startLine, startColumn);
        }
```

```
        if (c == '&' && Match('&'))
            return new Token(TokenType.AND, "&&", startLine, startColumn);

        if (c == '|' && Match('|'))
            return new Token(TokenType.OR, "||", startLine, startColumn);
```

```
        // String literals
        if (c == '"')
        {
            return ScanString(startLine, startColumn);
        }
```

```
        // Numbers
        if (char.IsDigit(c))
        {
```

```
            return ScanNumber(startLine, startColumn);
        }

        // Identifiers and keywords
        if (char.IsLetter(c) || c == '_')
        {
            return ScanIdentifier(startLine, startColumn);
        }

        return new Token(TokenType.UNKNOWN, c.ToString(), startLine, startColumn);
    }

    private Token ScanString(int startLine, int startColumn)
    {
        StringBuilder value = new StringBuilder();

        while (!IsAtEnd() && Peek() != '"')
        {
            if (Peek() == '\n')
            {
                line++;
                column = 1;
            }
            value.Append(Advance());
        }

        if (IsAtEnd())
        {
            throw new Exception($"Unterminated string at line {startLine}");
        }

        // Consume closing quote
        Advance();
        return new Token(TokenType.STRING_LITERAL, value.ToString(), startLine, startColumn);
    }

    private Token ScanNumber(int startLine, int startColumn)
    {
        StringBuilder value = new StringBuilder();
        value.Append(source[position - 1]);

        while (!IsAtEnd() && char.IsDigit(Peek()))
        {
            value.Append(Advance());
        }

        // Check for decimal point
        if (!IsAtEnd() && Peek() == '.' && position + 1 < source.Length && char.IsDigit(source[position + 1]))
        {
            value.Append(Advance()); // consume '.'
            while (!IsAtEnd() && char.IsDigit(Peek()))
            {
                value.Append(Advance());
            }
        }

        return new Token(TokenType.NUMBER, value.ToString(), startLine, startColumn);
    }

    private Token ScanIdentifier(int startLine, int startColumn)
    {
        StringBuilder value = new StringBuilder();
```

```
        value.Append(source[position - 1]);
```

```
        while (!IsAtEnd() && (char.IsLetterOrDigit(Peek()) || Peek() == '_'))
        {
            value.Append(Advance());
        }
```

```
        string text = value.ToString();
        TokenType type = keywords.ContainsKey(text) ? keywords[text] : TokenType.IDENTIFIER;
        return new Token(type, text, startLine, startColumn);
    }
```

```
    private void SkipWhitespace()
    {
        while (!IsAtEnd())
        {
            char c = Peek();
            if (c == ' ' || c == '\r' || c == '\t')
            {
                Advance();
            }
            else if (c == '/' && position + 1 < source.Length && source[position + 1] == '/')
            {
                // Handle single-line comments
                while (!IsAtEnd() && Peek() != '\n')
                {
                    Advance();
                }
            }
            else if (c == '/' && position + 1 < source.Length && source[position + 1] == '*')
            {
                // Handle multi-line comments
                Advance(); // Consume '/'
                Advance(); // Consume '*'
                while (!IsAtEnd() && !(Peek() == '*' && position + 1 < source.Length && source[position + 1] == '/'))
                {
                    if (Peek() == '\n')
                    {
                        line++;
                        column = 1;
                    }
                    Advance();
                }
                if (!IsAtEnd())
                {
                    Advance(); // Consume '*'
                    Advance(); // Consume '/'
                }
            }
            else
            {
                break;
            }
        }
    }
```

```
    private char Advance()
    {
        column++;
        return source[position++];
    }
```

```
    private bool Match(char expected)
    {
        if (IsAtEnd() || source[position] != expected) return false;
        position++;
        column++;
        return true;
    }
```

```
    private char Peek()
    {
        if (IsAtEnd()) return '\0';
        return source[position];
    }
```

```
    private bool IsAtEnd()
    {
        return position >= source.Length;
    }
}
```

```
// Abstract Syntax Tree Nodes
public abstract class ASTNode
{
    public abstract void Accept(IASTVisitor visitor);
}
```

```
public abstract class Statement : ASTNode { }
public abstract class Expression : ASTNode { }
```

```
public class PrintStatement : Statement
{
    public Expression Expression { get; set; }

    public PrintStatement(Expression expression)
    {
        Expression = expression;
    }

    public override void Accept(IASTVisitor visitor)
    {
        visitor.VisitPrintStatement(this);
    }
}
```

```
public class VariableDeclaration : Statement
{
    public string Type { get; set; }
    public string Name { get; set; }
    public Expression Initializer { get; set; }

    public VariableDeclaration(string type, string name, Expression initializer = null)
    {
        Type = type;
        Name = name;
        Initializer = initializer;
    }

    public override void Accept(IASTVisitor visitor)
    {
        visitor.VisitVariableDeclaration(this);
    }
}
```

```
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 void Accept(IASTVisitor visitor)
    {
        visitor.VisitAssignmentStatement(this);
    }
}
```

```
public class IfStatement : Statement
{
    public Expression Condition { get; set; }
    public List<Statement> ThenBranch { get; set; }
    public List<Statement> ElseBranch { get; set; }

    public IfStatement(Expression condition, List<Statement> thenBranch, List<Statement> elseBranch = null)
    {
        Condition = condition;
        ThenBranch = thenBranch;
        ElseBranch = elseBranch ?? new List<Statement>();
    }

    public override void Accept(IASTVisitor visitor)
    {
        visitor.VisitIfStatement(this);
    }
}
```

```
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 void Accept(IASTVisitor visitor)
    {
        visitor.VisitWhileStatement(this);
    }
}
```

```
// Expression nodes
public class BinaryExpression : Expression
{
    public Expression Left { get; set; }
    public Token Operator { get; set; }
    public Expression Right { get; set; }

    public BinaryExpression(Expression left, Token op, Expression right)
```

```
    {
        Left = left;
        Operator = op;
        Right = right;
    }

    public override void Accept(IASTVisitor visitor)
    {
        visitor.VisitBinaryExpression(this);
    }
}
```

```
public class UnaryExpression : Expression
{
    public Token Operator { get; set; }
    public Expression Right { get; set; }

    public UnaryExpression(Token op, Expression right)
    {
        Operator = op;
        Right = right;
    }

    public override void Accept(IASTVisitor visitor)
    {
        visitor.VisitUnaryExpression(this);
    }
}
```

```
public class LiteralExpression : Expression
{
    public object Value { get; set; }

    public LiteralExpression(object value)
    {
        Value = value;
    }

    public override void Accept(IASTVisitor visitor)
    {
        visitor.VisitLiteralExpression(this);
    }
}
```

```
public class VariableExpression : Expression
{
    public string Name { get; set; }

    public VariableExpression(string name)
    {
        Name = name;
    }

    public override void Accept(IASTVisitor visitor)
    {
        visitor.VisitVariableExpression(this);
    }
}
```

```
// Visitor interface for AST traversal
public interface IASTVisitor
{
```

```
    void VisitPrintStatement(PrintStatement stmt);
    void VisitVariableDeclaration(VariableDeclaration stmt);
    void VisitAssignmentStatement(AssignmentStatement stmt);
    void VisitIfStatement(IfStatement stmt);
    void VisitWhileStatement(WhileStatement stmt);
    void VisitBinaryExpression(BinaryExpression expr);
    void VisitUnaryExpression(UnaryExpression expr);
    void VisitLiteralExpression(LiteralExpression expr);
    void VisitVariableExpression(VariableExpression expr);
}
```

```
// Recursive Descent Parser
public class Parser
{
    private List<Token> tokens;
    private int current;
```

```
    public Parser(List<Token> tokens)
    {
        this.tokens = tokens;
        this.current = 0;
    }
```

```
    public List<Statement> Parse()
    {
        List<Statement> statements = new List<Statement>();

        while (!IsAtEnd())
        {
            try
            {
                Statement stmt = ParseStatement();
                if (stmt != null)
                    statements.Add(stmt);
            }
            catch (Exception e)
            {
                Console.WriteLine($"Parse error at line {Peek().Line}: {e.Message}");
                Synchronize();
            }
        }

        return statements;
    }
```

```
    private Statement ParseStatement()
    {
        if (Match(TokenType.PRINT)) return ParsePrintStatement();
        if (Match(TokenType.INT, TokenType.FLOAT, TokenType.STRING, TokenType.BOOL)) return ParseVariableDeclaration();
        if (Match(TokenType.IF)) return ParseIfStatement();
        if (Match(TokenType.WHILE)) return ParseWhileStatement();
        if (Check(TokenType.IDENTIFIER) && CheckNext(TokenType.ASSIGN)) return ParseAssignmentStatement();

        // Expression statement
        Expression expr = ParseExpression();
        Consume(TokenType.SEMICOLON, "Expected ';' after expression");
        return new PrintStatement(expr);
    }
```

```
    private Statement ParsePrintStatement()
    {
        Consume(TokenType.LEFT_PAREN, "Expected '(' after 'print'");
```

```
        Expression expr = ParseExpression();
        Consume(TokenType.RIGHT_PAREN, "Expected ')' after expression");
        Consume(TokenType.SEMICOLON, "Expected ';' after print statement");
        return new PrintStatement(expr);
    }
```

```
    private Statement ParseVariableDeclaration()
    {
        Token typeToken = Previous();
        Token name = Consume(TokenType.IDENTIFIER, "Expected variable name");

        Expression initializer = null;
        if (Match(TokenType.ASSIGN))
        {
            initializer = ParseExpression();
        }

        Consume(TokenType.SEMICOLON, "Expected ';' after variable declaration");
        return new VariableDeclaration(typeToken.Value, name.Value, initializer);
    }
```

```
    private Statement ParseAssignmentStatement()
    {
        Token name = Advance();
        Consume(TokenType.ASSIGN, "Expected '=' in assignment");
        Expression value = ParseExpression();
        Consume(TokenType.SEMICOLON, "Expected ';' after assignment");
        return new AssignmentStatement(name.Value, value); // Fixed typo: Changed DefinitionStatement to AssignmentStatement
    }
```

```
    private Statement ParseIfStatement()
    {
        Consume(TokenType.LEFT_PAREN, "Expected '(' after 'if'");
        Expression condition = ParseExpression();
        Consume(TokenType.RIGHT_PAREN, "Expected ')' after if condition");

        Consume(TokenType.LEFT_BRACE, "Expected '{' before if body");
        List<Statement> thenBranch = ParseBlockStatement();

        List<Statement> elseBranch = null;
        if (Match(TokenType.ELSE))
        {
            Consume(TokenType.LEFT_BRACE, "Expected '{' before else body");
            elseBranch = ParseBlockStatement();
        }

        return new IfStatement(condition, thenBranch, elseBranch);
    }
```

```
    private Statement ParseWhileStatement()
    {
        Consume(TokenType.LEFT_PAREN, "Expected '(' after 'while'");
        Expression condition = ParseExpression();
        Consume(TokenType.RIGHT_PAREN, "Expected ')' after while condition");

        Consume(TokenType.LEFT_BRACE, "Expected '{' before while body");
        List<Statement> body = ParseBlockStatement();

        return new WhileStatement(condition, body);
    }
```

```
    private List<Statement> ParseBlockStatement()
```

```
    {
        List<Statement> statements = new List<Statement>();

        while (!Check(TokenType.RIGHT_BRACE) && !IsAtEnd())
        {
            statements.Add(ParseStatement());
        }

        Consume(TokenType.RIGHT_BRACE, "Expected '}' after block");
        return statements;
    }
```

```
    private Expression ParseExpression()
    {
        return ParseLogicalOr();
    }
```

```
    private Expression ParseLogicalOr()
    {
        Expression expr = ParseLogicalAnd();

        while (Match(TokenType.OR))
        {
            Token op = Previous();
            Expression right = ParseLogicalAnd();
            expr = new BinaryExpression(expr, op, right);
        }

        return expr;
    }
```

```
    private Expression ParseLogicalAnd()
    {
        Expression expr = ParseEquality();

        while (Match(TokenType.AND))
        {
            Token op = Previous();
            Expression right = ParseEquality();
            expr = new BinaryExpression(expr, op, right);
        }

        return expr;
    }
```

```
    private Expression ParseEquality()
    {
        Expression expr = ParseComparison();

        while (Match(TokenType.NOT_EQUAL, TokenType.EQUAL))
        {
            Token op = Previous();
            Expression right = ParseComparison();
            expr = new BinaryExpression(expr, op, right);
        }

        return expr;
    }
```

```
    private Expression ParseComparison()
    {
        Expression expr = ParseTerm();
```

```
        while (Match(TokenType.GREATER_THAN, TokenType.GREATER_EQUAL, TokenType.LESS_THAN, TokenType.LESS_EQUAL))
        {
            Token op = Previous();
            Expression right = ParseTerm();
            expr = new BinaryExpression(expr, op, right);
        }

        return expr;
    }
```

```
    private Expression ParseTerm()
    {
        Expression expr = ParseFactor();

        while (Match(TokenType.MINUS, TokenType.PLUS))
        {
            Token op = Previous();
            Expression right = ParseFactor();
            expr = new BinaryExpression(expr, op, right);
        }

        return expr;
    }
```

```
    private Expression ParseFactor()
    {
        Expression expr = ParseUnary();

        while (Match(TokenType.DIVIDE, TokenType.MULTIPLY, TokenType.MODULO))
        {
            Token op = Previous();
            Expression right = ParseUnary();
            expr = new BinaryExpression(expr, op, right);
        }

        return expr;
    }
```

```
    private Expression ParseUnary()
    {
        if (Match(TokenType.NOT, TokenType.MINUS))
        {
            Token op = Previous();
            Expression right = ParseUnary();
            return new UnaryExpression(op, right);
        }

        return ParsePrimary();
    }
```

```
    private Expression ParsePrimary()
    {
        if (Match(TokenType.BOOLEAN_LITERAL))
        {
            return new LiteralExpression(Previous().Value == "true");
        }

        if (Match(TokenType.NUMBER))
        {
            string value = Previous().Value;
            if (value.Contains('.'))
```

```
                return new LiteralExpression(double.Parse(value));
            else
                return new LiteralExpression(int.Parse(value));
        }

        if (Match(TokenType.STRING_LITERAL))
        {
            return new LiteralExpression(Previous().Value);
        }

        if (Match(TokenType.IDENTIFIER))
        {
            return new VariableExpression(Previous().Value);
        }

        if (Match(TokenType.LEFT_PAREN))
        {
            Expression expr = ParseExpression();
            Consume(TokenType.RIGHT_PAREN, "Expected ')' after expression");
            return expr;
        }

        throw new Exception($"Unexpected token: {Peek().Value} at line {Peek().Line}");
    }
```

```
    private bool Match(params TokenType[] types)
    {
        foreach (TokenType type in types)
        {
            if (Check(type))
            {
                Advance();
                return true;
            }
        }
        return false;
    }
```

```
    private bool Check(TokenType type)
    {
        if (IsAtEnd()) return false;
        return Peek().Type == type;
    }

    private bool CheckNext(TokenType type)
    {
        if (current + 1 >= tokens.Count) return false;
        return tokens[current + 1].Type == type;
    }
```

```
    private Token Advance()
    {
        if (!IsAtEnd()) current++;
        return Previous();
    }
```

```
    private bool IsAtEnd()
    {
        return Peek().Type == TokenType.EOF;
    }
```

```
    private Token Peek()
```

```
    {
        return tokens[current];
    }
```

```
    private Token Previous()
    {
        return tokens[current - 1];
    }
```

```
    private Token Consume(TokenType type, string message)
    {
        if (Check(type)) return Advance();
        throw new Exception($"{message}. Got {Peek().Type} at line {Peek().Line}");
    }
```

```
    private void Synchronize()
    {
        Advance();

        while (!IsAtEnd())
        {
            if (Previous().Type == TokenType.SEMICOLON) return;

            switch (Peek().Type)
            {
                case TokenType.IF:
                case TokenType.WHILE:
                case TokenType.FOR:
                case TokenType.RETURN:
                case TokenType.INT:
                case TokenType.FLOAT:
                case TokenType.STRING:
                case TokenType.BOOL:
                    return;
            }

            Advance();
        }
    }
}
```

```
// Symbol Table for semantic analysis
public class Symbol
{
    public string Name { get; set; }
    public string Type { get; set; }
    public object Value { get; set; }

    public Symbol(string name, string type, object value = null)
    {
        Name = name;
        Type = type;
        Value = value;
    }
}
```

```
public class SymbolTable
{
    private List<Dictionary<string, Symbol>> scopes;
    private int currentScopeLevel;
```

```
    public SymbolTable()
```

```
    {
        scopes = new List<Dictionary<string, Symbol>>();
        scopes.Add(new Dictionary<string, Symbol>()); // Global scope
        currentScopeLevel = 0;
    }
```

```
    public void EnterScope()
    {
        scopes.Add(new Dictionary<string, Symbol>());
        currentScopeLevel++;
    }
```

```
    public void ExitScope()
    {
        if (currentScopeLevel > 0)
        {
            scopes.RemoveAt(currentScopeLevel);
            currentScopeLevel--;
        }
    }
```

```
    public void Define(string name, string type, object value = null)
    {
        scopes[currentScopeLevel][name] = new Symbol(name, type, value);
    }
```

```
    public Symbol Get(string name)
    {
        for (int i = currentScopeLevel; i >= 0; i--)
        {
            if (scopes[i].ContainsKey(name))
                return scopes[i][name];
        }
        return null;
    }
```

```
    public bool IsDefined(string name)
    {
        for (int i = currentScopeLevel; i >= 0; i--)
        {
            if (scopes[i].ContainsKey(name))
                return true;
        }
        return false;
    }
```

```
    public void Set(string name, object value)
    {
        for (int i = currentScopeLevel; i >= 0; i--)
        {
            if (scopes[i].ContainsKey(name))
            {
                scopes[i][name].Value = value;
                return;
            }
        }
    }
}
```

```
// Semantic Analyzer
public class SemanticAnalyzer : IASTVisitor
{
```

```
    private SymbolTable symbolTable;
    private List<string> errors;
```

```
    public SemanticAnalyzer()
    {
        symbolTable = new SymbolTable();
        errors = new List<string>();
    }
```

```
    public List<string> Analyze(List<Statement> statements)
    {
        errors.Clear();
        symbolTable.EnterScope(); // Enter global scope
        foreach (Statement stmt in statements)
        {
            stmt.Accept(this);
        }
        symbolTable.ExitScope();
        return errors;
    }
```

```
    public void VisitPrintStatement(PrintStatement stmt)
    {
        stmt.Expression.Accept(this);
    }
```

```
    public void VisitVariableDeclaration(VariableDeclaration stmt)
    {
        if (symbolTable.IsDefined(stmt.Name))
        {
            errors.Add($"Variable '{stmt.Name}' is already declared in this scope");
            return;
        }
```

```
        string exprType = null;
        if (stmt.Initializer != null)
        {
            stmt.Initializer.Accept(this);
            exprType = GetExpressionType(stmt.Initializer);
            if (!IsCompatibleType(stmt.Type, exprType))
            {
                errors.Add($"Type mismatch: Cannot assign {exprType} to {stmt.Type} variable '{stmt.Name}'");
            }
        }
```

```
        symbolTable.Define(stmt.Name, stmt.Type, null);
    }
```

```
    public void VisitAssignmentStatement(AssignmentStatement stmt)
    {
        if (!symbolTable.IsDefined(stmt.Variable))
        {
            errors.Add($"Undefined variable '{stmt.Variable}'");
            return;
        }
```

```
        stmt.Value.Accept(this);
        string exprType = GetExpressionType(stmt.Value);
        Symbol symbol = symbolTable.Get(stmt.Variable);
        if (!IsCompatibleType(symbol.Type, exprType))
        {
            errors.Add($"Type mismatch: Cannot assign {exprType} to {symbol.Type} variable '{stmt.Variable}'");
```

```
        }
    }
```

```
    public void VisitIfStatement(IfStatement stmt)
    {
        stmt.Condition.Accept(this);
        string condType = GetExpressionType(stmt.Condition);
        if (condType != "bool")
        {
            errors.Add($"If condition must be boolean, got {condType}");
        }
```

```
        symbolTable.EnterScope();
        foreach (Statement s in stmt.ThenBranch)
        {
            s.Accept(this);
        }
        symbolTable.ExitScope();
```

```
        symbolTable.EnterScope();
        foreach (Statement s in stmt.ElseBranch)
        {
            s.Accept(this);
        }
        symbolTable.ExitScope();
    }
```

```
    public void VisitWhileStatement(WhileStatement stmt)
    {
        stmt.Condition.Accept(this);
        string condType = GetExpressionType(stmt.Condition);
        if (condType != "bool")
        {
            errors.Add($"While condition must be boolean, got {condType}");
        }
```

```
        symbolTable.EnterScope();
        foreach (Statement s in stmt.Body)
        {
            s.Accept(this);
        }
        symbolTable.ExitScope();
    }
```

```
    public void VisitBinaryExpression(BinaryExpression expr)
    {
        expr.Left.Accept(this);
        expr.Right.Accept(this);
        string leftType = GetExpressionType(expr.Left);
        string rightType = GetExpressionType(expr.Right);
```

```
        switch (expr.Operator.Type)
        {
            case TokenType.PLUS:
            case TokenType.MINUS:
            case TokenType.MULTIPLY:
            case TokenType.DIVIDE:
            case TokenType.MODULO:
                if (!(leftType == "int" || leftType == "float") || !(rightType == "int" || rightType == "float"))
                {
                    errors.Add($"Operator {expr.Operator.Value} requires numeric operands, got {leftType} and {rightType}");
                }
```

```
            break;
        case TokenType.EQUAL:
        case TokenType.NOT_EQUAL:
            if (leftType != rightType)
            {
                errors.Add($"Cannot compare {leftType} with {rightType} using {expr.Operator.Value}");
            }
            break;
        case TokenType.AND:
        case TokenType.OR:
            if (leftType != "bool" || rightType != "bool")
            {
                errors.Add($"Operator {expr.Operator.Value} requires boolean operands, got {leftType} and {rightType}");
            }
            break;
        case TokenType.GREATER_THAN:
        case TokenType.GREATER_EQUAL:
        case TokenType.LESS_THAN:
        case TokenType.LESS_EQUAL:
            if (!(leftType == "int" || leftType == "float") || !(rightType == "int" || rightType == "float"))
            {
                errors.Add($"Operator {expr.Operator.Value} requires numeric operands, got {leftType} and {rightType}");
            }
            break;
    }
}
```

```
public void VisitUnaryExpression(UnaryExpression expr)
{
    expr.Right.Accept(this);
    string rightType = GetExpressionType(expr.Right);
    if (expr.Operator.Type == TokenType.NOT && rightType != "bool")
    {
        errors.Add($"Operator ! requires boolean operand, got {rightType}");
    }
    else if (expr.Operator.Type == TokenType.MINUS && !(rightType == "int" || rightType == "float"))
    {
        errors.Add($"Operator - requires numeric operand, got {rightType}");
    }
}
```

```
public void VisitLiteralExpression(LiteralExpression expr)
{
    // Nothing to check for literals
}
```

```
public void VisitVariableExpression(VariableExpression expr)
{
    if (!symbolTable.IsDefined(expr.Name))
    {
        errors.Add($"Undefined variable '{expr.Name}'");
    }
}
```

```
private string GetExpressionType(Expression expr)
{
    if (expr is LiteralExpression lit)
    {
        if (lit.Value is int) return "int";
        if (lit.Value is double) return "float";
        if (lit.Value is string) return "string";
        if (lit.Value is bool) return "bool";
```

```
        }
        else if (expr is VariableExpression var)
        {
            Symbol symbol = symbolTable.Get(var.Name);
            return symbol?.Type ?? "unknown";
        }
        else if (expr is BinaryExpression bin)
        {
            string leftType = GetExpressionType(bin.Left);
            string rightType = GetExpressionType(bin.Right);
            switch (bin.Operator.Type)
            {
                case TokenType.PLUS:
                    if (leftType == "string" || rightType == "string") return "string";
                    return (leftType == "float" || rightType == "float") ? "float" : "int";
                case TokenType.MINUS:
                case TokenType.MULTIPLY:
                case TokenType.DIVIDE:
                    return (leftType == "float" || rightType == "float") ? "float" : "int";
                case TokenType.MODULO:
                    return "int";
                case TokenType.EQUAL:
                case TokenType.NOT_EQUAL:
                case TokenType.GREATER_THAN:
                case TokenType.GREATER_EQUAL:
                case TokenType.LESS_THAN:
                case TokenType.LESS_EQUAL:
                case TokenType.AND:
                case TokenType.OR:
                    return "bool";
            }
        }
        else if (expr is UnaryExpression un)
        {
            if (un.Operator.Type == TokenType.NOT) return "bool";
            return GetExpressionType(un.Right);
        }
        return "unknown";
    }
```

```
    private bool IsCompatibleType(string varType, string exprType)
    {
        if (varType == exprType) return true;
        if (varType == "float" && exprType == "int") return true; // Allow int to float
        return false;
    }
}
```

```
// Intermediate Code Generator (Three-Address Code)
public class IntermediateCodeGenerator : IASTVisitor
{
    private List<string> code;
    private int tempCounter;
    private int labelCounter;
    private string lastTemp;
```

```
    public IntermediateCodeGenerator()
    {
        code = new List<string>();
        tempCounter = 0;
        labelCounter = 0;
    }
```

```
public List<string> Generate(List<Statement> statements)
{
    code.Clear();
    tempCounter = 0;
    labelCounter = 0;
    foreach (Statement stmt in statements)
    {
        stmt.Accept(this);
    }
    return code;
}
```

```
public void VisitPrintStatement(PrintStatement stmt)
{
    stmt.Expression.Accept(this);
    code.Add($"print {lastTemp}");
}
```

```
public void VisitVariableDeclaration(VariableDeclaration stmt)
{
    if (stmt.Initializer != null)
    {
        stmt.Initializer.Accept(this);
        code.Add($"{stmt.Name} = {lastTemp}");
    }
    else
    {
        code.Add($"{stmt.Name} = 0");
    }
}
```

```
public void VisitAssignmentStatement(AssignmentStatement stmt)
{
    stmt.Value.Accept(this);
    code.Add($"{stmt.Variable} = {lastTemp}");
}
```

```
public void VisitIfStatement(IfStatement stmt)
{
    stmt.Condition.Accept(this);
    string elseLabel = $"L{labelCounter++}";
    string endLabel = $"L{labelCounter++}";
    code.Add($"if not {lastTemp} goto {elseLabel}");

    foreach (Statement s in stmt.ThenBranch)
    {
        s.Accept(this);
    }
    code.Add($"goto {endLabel}");
    code.Add($"{elseLabel}:");

    foreach (Statement s in stmt.ElseBranch)
    {
        s.Accept(this);
    }
    code.Add($"{endLabel}:");
}
```

```
public void VisitWhileStatement(WhileStatement stmt)
{
    string startLabel = $"L{labelCounter++}";
```

```
        string endLabel = $"L{labelCounter++}";
        code.Add($"{startLabel}:");

        stmt.Condition.Accept(this);
        code.Add($"if not {lastTemp} goto {endLabel}");

        foreach (Statement s in stmt.Body)
        {
            s.Accept(this);
        }
        code.Add($"goto {startLabel}");
        code.Add($"{endLabel}:");
    }
```

```
    public void VisitBinaryExpression(BinaryExpression expr)
    {
        expr.Left.Accept(this);
        string leftTemp = lastTemp;
        expr.Right.Accept(this);
        string rightTemp = lastTemp;
        string resultTemp = $"t{tempCounter++}";
        code.Add($"{resultTemp} = {leftTemp} {expr.Operator.Value} {rightTemp}");
        lastTemp = resultTemp;
    }
```

```
    public void VisitUnaryExpression(UnaryExpression expr)
    {
        expr.Right.Accept(this);
        string rightTemp = lastTemp;
        string resultTemp = $"t{tempCounter++}";
        code.Add($"{resultTemp} = {expr.Operator.Value}{rightTemp}");
        lastTemp = resultTemp;
    }
```

```
    public void VisitLiteralExpression(LiteralExpression expr)
    {
        lastTemp = expr.Value.ToString();
    }
```

```
    public void VisitVariableExpression(VariableExpression expr)
    {
        lastTemp = expr.Name;
    }
}
```

```
// Code Generator (Interpreter)
public class Interpreter : IASTVisitor
{
    private SymbolTable symbolTable;
    private object lastValue;

    public Interpreter()
    {
        symbolTable = new SymbolTable();
    }

    public void Execute(List<Statement> statements)
    {
        symbolTable.EnterScope();
        foreach (Statement stmt in statements)
        {
            stmt.Accept(this);
```

```
        }
        symbolTable.ExitScope();
    }

    public void VisitPrintStatement(PrintStatement stmt)
    {
        stmt.Expression.Accept(this);
        Console.WriteLine(lastValue?.ToString() ?? "null");
    }

    public void VisitVariableDeclaration(VariableDeclaration stmt)
    {
        object value = null;
        if (stmt.Initializer != null)
        {
            stmt.Initializer.Accept(this);
            value = lastValue;
        }

        symbolTable.Define(stmt.Name, stmt.Type, value);
    }

    public void VisitAssignmentStatement(AssignmentStatement stmt)
    {
        stmt.Value.Accept(this);
        symbolTable.Set(stmt.Variable, lastValue);
    }

    public void VisitIfStatement(IfStatement stmt)
    {
        stmt.Condition.Accept(this);
        bool condition = IsTruthy(lastValue);

        symbolTable.EnterScope();
        if (condition)
        {
            foreach (Statement s in stmt.ThenBranch)
            {
                s.Accept(this);
            }
        }
        else
        {
            foreach (Statement s in stmt.ElseBranch)
            {
                s.Accept(this);
            }
        }
        symbolTable.ExitScope();
    }

    public void VisitWhileStatement(WhileStatement stmt)
    {
        symbolTable.EnterScope();
        while (true)
        {
            stmt.Condition.Accept(this);
            if (!IsTruthy(lastValue)) break;

            foreach (Statement s in stmt.Body)
            {
                s.Accept(this);
```

```
            }
        }
        symbolTable.ExitScope();
    }

    public void VisitBinaryExpression(BinaryExpression expr)
    {
        expr.Left.Accept(this);
        object left = lastValue;

        expr.Right.Accept(this);
        object right = lastValue;

        switch (expr.Operator.Type)
        {
            case TokenType.PLUS:
                if (left is string || right is string)
                    lastValue = left?.ToString() + right?.ToString();
                else if (left is double || right is double)
                    lastValue = Convert.ToDouble(left) + Convert.ToDouble(right);
                else
                    lastValue = Convert.ToInt32(left) + Convert.ToInt32(right);
                break;

            case TokenType.MINUS:
                if (left is double || right is double)
                    lastValue = Convert.ToDouble(left) - Convert.ToDouble(right);
                else
                    lastValue = Convert.ToInt32(left) - Convert.ToInt32(right);
                break;

            case TokenType.MULTIPLY:
                if (left is double || right is double)
                    lastValue = Convert.ToDouble(left) * Convert.ToDouble(right);
                else
                    lastValue = Convert.ToInt32(left) * Convert.ToInt32(right);
                break;

            case TokenType.DIVIDE:
                lastValue = Convert.ToDouble(left) / Convert.ToDouble(right);
                break;

            case TokenType.MODULO:
                lastValue = Convert.ToInt32(left) % Convert.ToInt32(right);
                break;

            case TokenType.GREATER_THAN:
                if (left is double || right is double)
                    lastValue = Convert.ToDouble(left) > Convert.ToDouble(right);
                else
                    lastValue = Convert.ToInt32(left) > Convert.ToInt32(right);
                break;

            case TokenType.GREATER_EQUAL:
                if (left is double || right is double)
                    lastValue = Convert.ToDouble(left) >= Convert.ToDouble(right);
                else
                    lastValue = Convert.ToInt32(left) >= Convert.ToInt32(right);
                break;

            case TokenType.LESS_THAN:
                if (left is double || right is double)
```

```
                lastValue = Convert.ToDouble(left) < Convert.ToDouble(right);
            else
                lastValue = Convert.ToInt32(left) < Convert.ToInt32(right);
            break;

        case TokenType.LESS_EQUAL:
            if (left is double || right is double)
                lastValue = Convert.ToDouble(left) <= Convert.ToDouble(right);
            else
                lastValue = Convert.ToInt32(left) <= Convert.ToInt32(right);
            break;

        case TokenType.EQUAL:
            lastValue = IsEqual(left, right);
            break;

        case TokenType.NOT_EQUAL:
            lastValue = !IsEqual(left, right);
            break;

        case TokenType.AND:
            lastValue = IsTruthy(left) && IsTruthy(right);
            break;

        case TokenType.OR:
            lastValue = IsTruthy(left) || IsTruthy(right);
            break;
    }
}

public void VisitUnaryExpression(UnaryExpression expr)
{
    expr.Right.Accept(this);

    switch (expr.Operator.Type)
    {
        case TokenType.MINUS:
            if (lastValue is double)
                lastValue = -(double)lastValue;
            else
                lastValue = -Convert.ToInt32(lastValue);
            break;

        case TokenType.NOT:
            lastValue = !IsTruthy(lastValue);
            break;
    }
}

public void VisitLiteralExpression(LiteralExpression expr)
{
    lastValue = expr.Value;
}

public void VisitVariableExpression(VariableExpression expr)
{
    Symbol symbol = symbolTable.Get(expr.Name);
    lastValue = symbol?.Value;
}

private bool IsTruthy(object obj)
{
```

```
        if (obj == null) return false;
        if (obj is bool) return (bool)obj;
        return true;
    }

    private bool IsEqual(object a, object b)
    {
        if (a == null && b == null) return true;
        if (a == null) return false;
        return a.Equals(b);
    }
}
```

```
// Main Compiler Class
public class MiniCompiler
{
    public void Compile(string sourceCode)
    {
        Console.WriteLine("=== COMPILATION PROCESS ===\n");

        try
        {
            // Phase 1: Lexical Analysis
            Console.WriteLine("Phase 1: Lexical Analysis (Tokenization)");
            Console.WriteLine("--------------------------------------");
            Lexer lexer = new Lexer(sourceCode);
            List<Token> tokens = lexer.Tokenize();

            Console.WriteLine("Tokens generated:");
            foreach (Token token in tokens.Take(20))
            {
                Console.WriteLine($"  {token}");
            }
            if (tokens.Count > 20)
                Console.WriteLine($"  ... and {tokens.Count - 20} more tokens");
            Console.WriteLine();

            // Phase 2: Syntax Analysis (Parsing)
            Console.WriteLine("Phase 2: Syntax Analysis (Parsing)");
            Console.WriteLine("----------------------------------");
            Parser parser = new Parser(tokens);
            List<Statement> ast = parser.Parse();
            Console.WriteLine($"AST generated with {ast.Count} statements");
            Console.WriteLine("Parse completed successfully!\n");

            // Phase 3: Semantic Analysis
            Console.WriteLine("Phase 3: Semantic Analysis");
            Console.WriteLine("--------------------------");
            SemanticAnalyzer analyzer = new SemanticAnalyzer();
            List<string> semanticErrors = analyzer.Analyze(ast);

            if (semanticErrors.Count > 0)
            {
                Console.WriteLine("Semantic errors found:");
                foreach (string error in semanticErrors)
                {
                    Console.WriteLine($"  Error: {error}");
                }
                Console.WriteLine();
                return;
            }
            else
```

```
            {
                Console.WriteLine("Semantic analysis passed!\n");
            }

            // Phase 4: Intermediate Code Generation
            Console.WriteLine("Phase 4: Intermediate Code Generation");
            Console.WriteLine("-------------------------------------");
            IntermediateCodeGenerator irGenerator = new IntermediateCodeGenerator();
            List<string> irCode = irGenerator.Generate(ast);
            Console.WriteLine("Three-Address Code generated:");
            foreach (string instruction in irCode)
            {
                Console.WriteLine($"  {instruction}");
            }
            Console.WriteLine();

            // Phase 5: Code Generation & Execution
            Console.WriteLine("Phase 5: Code Generation & Execution");
            Console.WriteLine("-------------------------------------");
            Console.WriteLine("Program output:");
            Console.WriteLine("---------------");

            Interpreter interpreter = new Interpreter();
            interpreter.Execute(ast);

            Console.WriteLine("\n=== COMPILATION SUCCESSFUL ===");
        }
        catch (Exception e)
        {
            Console.WriteLine($"Compilation failed: {e.Message}");
        }
    }

    public void CompileFile(string filePath)
    {
        try
        {
            string sourceCode = File.ReadAllText(filePath);
            Console.WriteLine($"Compiling file: {filePath}");
            Console.WriteLine($"Source code length: {sourceCode.Length} characters\n");
            Compile(sourceCode);
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine($"Error: File '{filePath}' not found.");
        }
        catch (Exception e)
        {
            Console.WriteLine($"Error reading file: {e.Message}");
        }
    }
}
```

```
// Program Entry Point
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("=== MINI COMPILER ===");
        Console.WriteLine("A Custom Compiler Implementation in C#\n");

        MiniCompiler compiler = new MiniCompiler();
```

```
        if (args.Length > 0)
        {
            // Compile from file
            compiler.CompileFile(args[0]);
        }
        else
        {
            // Interactive mode with sample program
            Console.WriteLine("No file specified. Running sample program...\n");

            string sampleProgram = @"
// Sample Program for Mini Compiler
int x = 10; // Declare x
int y = 20; /* Declare y */
int result = x + y * 2;
print(result);
```

```
string message = ""Hello, World!"";
print(message);
```

```
bool flag = true;
if (flag) {
    print(""Flag is true"");
    int counter = 0;
    while (counter < 3) {
        print(counter);
        counter = counter + 1;
    }
} else {
    print(""Flag is false"");
}
```

```
float pi = 3.14159;
float area = pi * 5.0 * 5.0;
print(area);
";

            Console.WriteLine("Sample source code:");
            Console.WriteLine("--------------------");
            Console.WriteLine(sampleProgram);
            Console.WriteLine();

            compiler.Compile(sampleProgram);
        }
    }
}
```

```
Outpu :
t
=== MINI COMPILER ===
A Custo  Compiler Implementatio  in C#
  m              n
No file specified. Running sample progra
                              m...
Sampl  source cod :
e-----e---------- -
-
// Sample Program for Mini Compile
int x = 10; // Declare
int y = 20; x* Declare y
int result =*x + y * 2;
print(result);

string message = "Hello, Worl   ;
print(message);  d!"

bool flag = true;
if (flag) {
    print("Flag is tru  );
    int counter = 0;
    while (counter < 3) {
        print(counter);
        counter = counter + 1;
    }
} else {
    print("Flag is fals  );
}        e"

float pi = 3.14159;
float area = pi * 5.0 * 5.0;
print(area);


=== COMPILATION PROCESS ===

Phase 1: Lexical Analysis (Tokenizatio )
---------------------------n----------
Token  generated:
s Token(INT, 'int', 3:1)
  Token(IDENTIFIER, 'x', 3:5)
  Token(ASSIG , '=', 3:7)
  Token(NUMBE , '10', 3:9)
  Token(BEMICOLON, ';', 3:11)
  Token(INT, 'int', 4:1)
  Token(IDENTIFIER, 'y', 4:5)
  Token(ASSIG , '=', 4:7)
  Token(NUMBE , '20', 4:9)
  Token(BEMICOLON, ';', 4:11)
  Token(INT, 'int', 5:1)
  Token(IDENTIFIER, 'result', 5:5)
  Token(ASSIG , '=', 5:12)
  Token(IDENTIFIER, 'x', 5:14)
  Token(PLUS, '+', 5:16)
  Token(IDENTIFIER, 'y', 5:18)
  Token(MULTIPLY, '*', 5:20)
  Token(NUMBE , '2', 5:22)
  Token(BEMICOLON, ';', 5:23)
  Token(PRINT, 'print', 6:1)
  ... and 82 more token
           s
Phase 2: Synta  Analysis (Parsing)
---------x---------------------- -
AST generated with 11 statements
Parse completed successfull !
    y
Phase 3: Semantic Analysis
-------------------------
Semantic analysis passe !
       d
Phase 4: Intermediat  Code Generation
---------g------------------------ -
Three-Address Code generated:
  x = 10
  y = 20
  t0 = y * 2
  t1 = x + t0
  result = t1
  print resul
t message = Hello, World!
  print messag
e flag = True
  if not flag goto L0
  print Flag is true
  counter = 0
  L2:
  t2 = counter < 3
  if not t2 goto L3
  print counte
r t3 = counter + 1
  counter = t3
  goto L2
  L3:
  goto L1
  L0:
  print Flag is false
  L1:
  pi = 3.14159
  t4 = pi * 5
  t5 = t4 * 5
  area = t5
  print are
a
Phase 5: Code Generation & Execution
------------------------------------
Program outpu :
--------t----- -
50
Hello, World!
Flag is true
0
1
2
78.53975

=== COMPILATION SUCCESSFUL ===
```