Mini Compiler Project Documentation

Project Overview

- **Project Title**

Mini Compiler for C# language.

Description

This project implements a complete compiler system that translates a C# programming language into stack-based virtual machine code. The compiler follows the traditional compilation pipeline with six distinct phases: Lexical Analysis, Syntax Analysis, Semantic Analysis, Optimization, Intermediate Code Generation, and Target Code Generation.

Programming Language

- **Implementation Language**: C# (.NET Framework)
- **Target Language**: Custom mini programming language
- **Output Format**: Stack-based Virtual Machine instructions

Key Objectives

- Demonstrate understanding of compiler construction principles
- Implement all major phases of compilation
- Provide error detection and reporting capabilities
- Generate optimized intermediate and target code
- Offer an interactive console interface for testing

System Architecture

Overall Design

The compiler follows a **multi-phase architecture** where each phase performs a specific transformation on the input:

```
Source Code → Lexer → Parser → Semantic Analyzer → Optimizer → IR Generator →
Target Generator
```

Design Patterns Used

- **Abstract Syntax Tree (AST)** pattern for program representation
- **Visitor Pattern** for tree traversal operations
- **Strategy Pattern** for different code generation strategies
- **Symbol Table** for identifier management

Compilation Phases

- **Phase 1: Lexical Analysis**

Component: `Lexer` class

Functionality:

- Converts source code into sequence of tokens
- Handles whitespace and comment elimination
- Provides line and column position tracking
- Supports single-line comments (`//`)
- Performs keyword recognition

- **Phase 2: Syntax Analysis (Parsing)**

Component: `Parser` class

Functionality:

- Builds Abstract Syntax Tree from token stream
- Implements recursive descent parsing
- Enforces grammar rules and syntax constraints
- Provides comprehensive error reporting

- **Phase 3: Semantic Analysis**

Component: `SemanticAnalyzer` class

Functionality:

- Type checking and compatibility verification
- Variable declaration and usage validation
- Symbol table construction and management
- Initialization status tracking

- <u>**Phase 4: Optimization**</u>

Component: `Optimizer` class

Functionality:

- Constant folding for arithmetic expressions
- Dead code elimination
- Expression simplification

- <u>**Phase 5: Intermediate Code Generation**</u>

Component: `IntermediateCodeGenerator` class

Functionality:

- Generates three-address code representation
- Handles temporary variable creation
- Implements control flow translation

- <u>**Phase 6: Target Code Generation**</u>

Component: `TargetCodeGenerator` class

Functionality:

- Converts intermediate code to stack-based VM instructions
- Implements stack-based expression evaluation
- Generates executable virtual machine code