COMSATS UNIVERSITY ISLAMABAD,

ATTOCK CAMPUS

Department Of Computer Science

MINI COMPILER

Course Name:	**Compiler Construction**
Submitted To:	**Sir Bilal**
Submitted By:	**Ayesha Arif(Sp22-bcs-030)**
	Ifra Habib(Sp22-bcs-032)

MINI COMPILER:

Introduction

A compiler is a specialized software tool that translates code written in a high-level programming language into a lower-level language, typically machine code that can be directly executed by a computer's CPU. This translation process enables developers to write programs in human-readable languages, which are then converted into executable instructions for the hardware.

The main purpose of a compiler is to bridge the gap between human logic and machine execution. It performs multiple tasks during this transformation process, which are organized into several phases. These phases include lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and final code generation. Additionally, a symbol table is maintained throughout compilation to keep track of variable names, types, and scopes, and error handling mechanisms are employed to ensure the correctness of the source code.

Compilers play an essential role in software development, enabling the creation of efficient, portable, and error-free software. By optimizing code during compilation, they also contribute to improved application performance and reduced resource consumption.

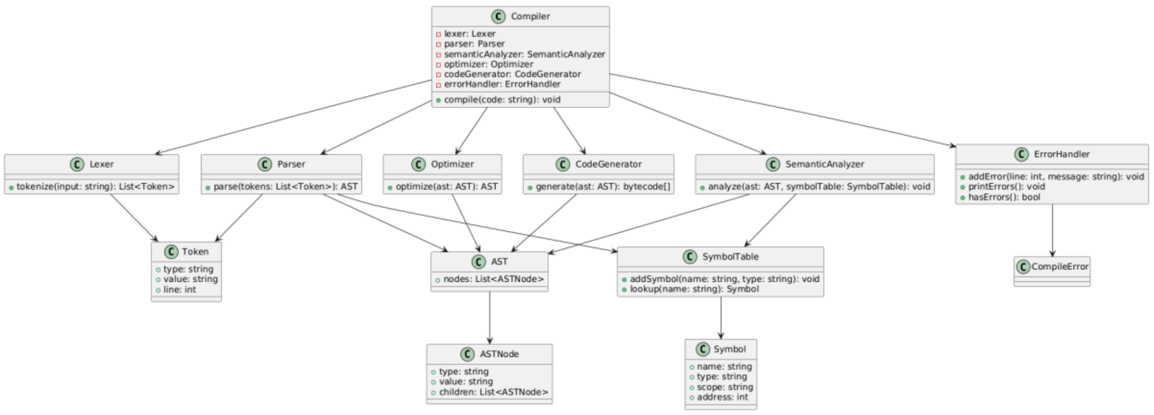
Compiler Phases

Phase	Element	Description
⍰	Lexical Analysis	• Tokenizes the input source code (e.g., identifies keywords, identifiers, operators) • Removes whitespaces and comments Use tools like Lex or write your own tokenizer
⍰	Syntax Analysis (Parsing)	• Validates the syntax based on grammar rules (e.g., via CFG) • Produces a parse tree or abstract syntax tree (AST) Use tools like Yacc/Bison or write recursive descent parsers
⍰	Semantic Analysis	• Checks for semantic errors (type checking, undeclared variables, etc.) • Annotates the AST with type information

?	Intermediate Code Generation	• Converts AST into an intermediate representation (IR), e.g., three-address code
?	Optimization (Optional)	• Performs basic optimizations (e.g., constant folding, dead code elimination)
?	Target Code Generation	• Converts IR to assembly/machine code or to a stack-based virtual machine code
?	Symbol Table Management	• Keeps track of variables, their types, scopes, and addresses
?	Error Handling	• Detects and reports syntax and semantic errors with line numbers and helpful messages

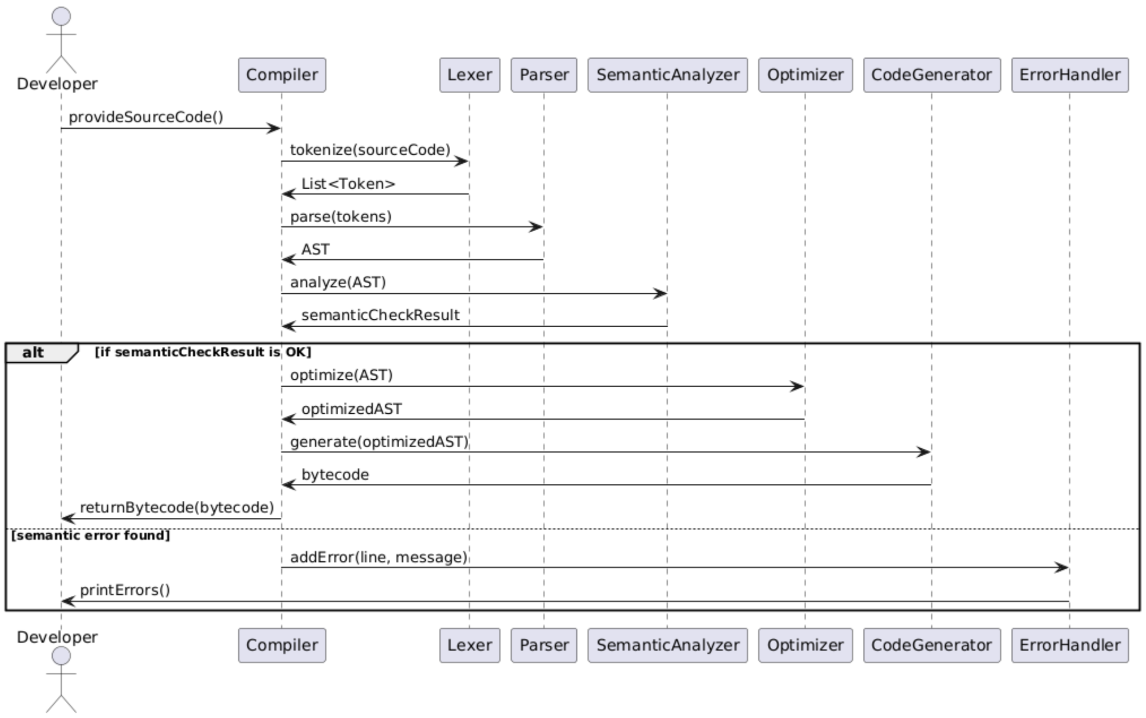
CLASS DIAGRAM:

- This **class diagram** shows the core components of a mini compiler and how they interact.
- Lexer: Tokenizes input source code into a list of Tokens.
- Parser: Uses the tokens to generate an AST (Abstract Syntax Tree).
- SymbolTable: Manages variable names, types, and scope; accessed during parsing.
- CodeGenerator: Consumes the AST to produce bytecode or target machine code.


Compiler
lexer: Lexer
parser: Parser
semanticAnalyzer: SemanticAnalyzer
optimizer: Optimizer
codeGenerator: CodeGenerator
errorHandler: ErrorHandler
compile(code: string): void
Lexer
tokenize(input: string): List<Token>
Parser
parse(tokens: List<Token>): AST
Optimizer
optimize(ast: AST): AST
CodeGenerator
generate(ast: AST): bytecode[]
SemanticAnalyzer
analyze(ast: AST, symbolTable: SymbolTable): void
ErrorHandler
addError(line: int, message: string): void
printErrors(): void
hasErrors(): bool
Token
type: string
value: string
line: int
AST
nodes: List<ASTNode>
SymbolTable
addSymbol(name: string, type: string): void
lookup(name: string): Symbol
CompileError
ASTNode
type: string
value: string
children: List<ASTNode>
Symbol
name: string
type: string
scope: string
address: int

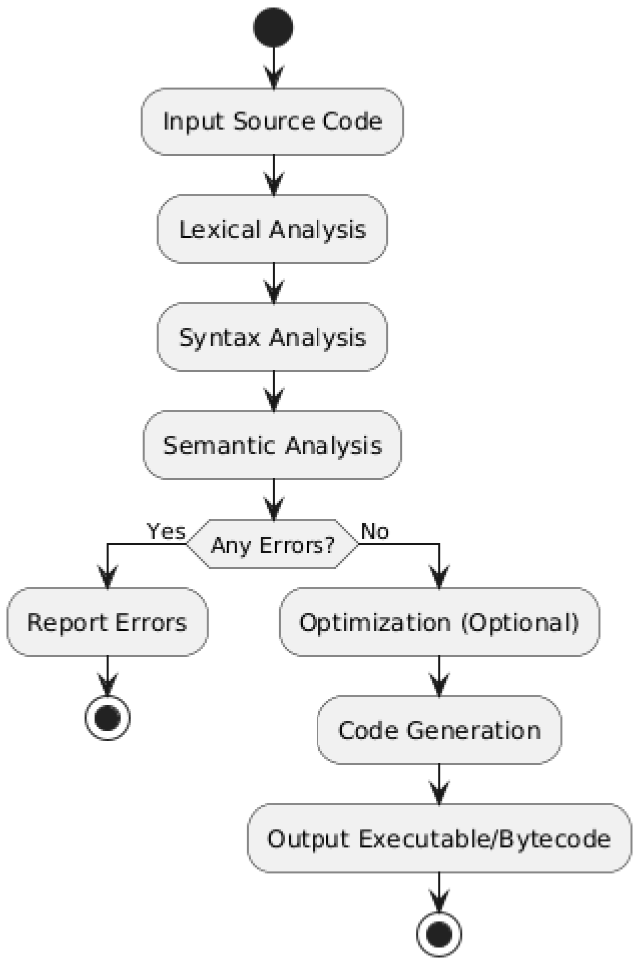
SEQUENCE DIAGRAM:

- This **sequence diagram** describes the **runtime interaction** between components.
- The Developer provides source code to the compiler.
- The flow then proceeds through Lexer, Parser, and CodeGenerator.
- Each component passes its output to the next until bytecode is returned to the user



ACTIVITY DIAGRAM:

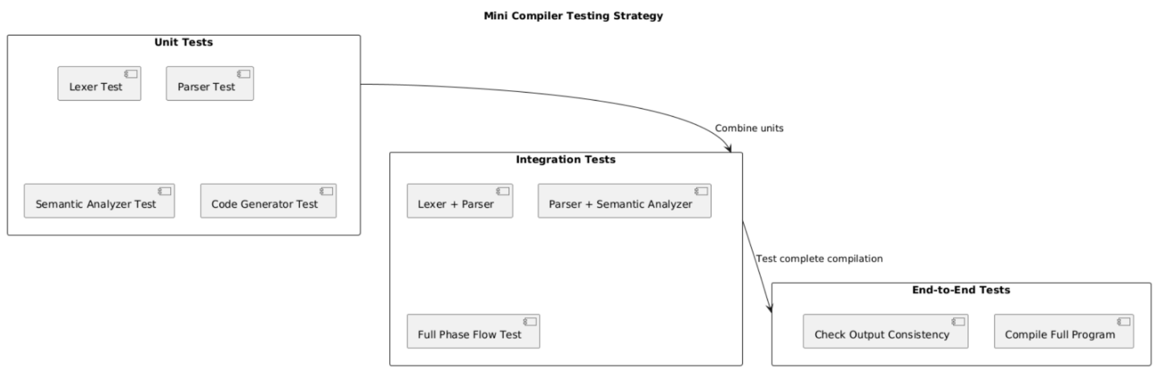
- This **activity diagram** models the **workflow** of the compiler.
- It starts with source code input and moves through:
- **Lexical**, **Syntax**, and **Semantic** analysis
- **Error checking**: if errors are found, report and stop
- If no errors: proceed to **Optimization** and **Code Generation**
- Ends with the generated output (machine code or bytecode).



TESTING:

Importance of Testing:

- Testing ensures **reliability, correctness, and maintainability**.
- Helps catch bugs early, reduce regression, and improve quality.
- Enables confident code changes and feature additions.


Mini Compiler Testing Strategy
Unit Tests
Lexer Test
Parser Test
Semantic Analyzer Test
Code Generator Test
Combine units
Integration Tests
Lexer + Parser
Parser + Semantic Analyzer
Full Phase Flow Test
Test complete compilation
End-to-End Tests
Check Output Consistency
Compile Full Program