Project Report

Project Title: MiniCompiler Design

Name: Nazrana Nahreen
Student ID : C231444
Semester : 5TH (spring-2025)
Section : 5BF

Course Code : CSE-3528
Course Title : Compiler Lab
Course Instructor: Farzana Tasnim
(Lecturer at International Islamic University Chittagong)

Submission Date:

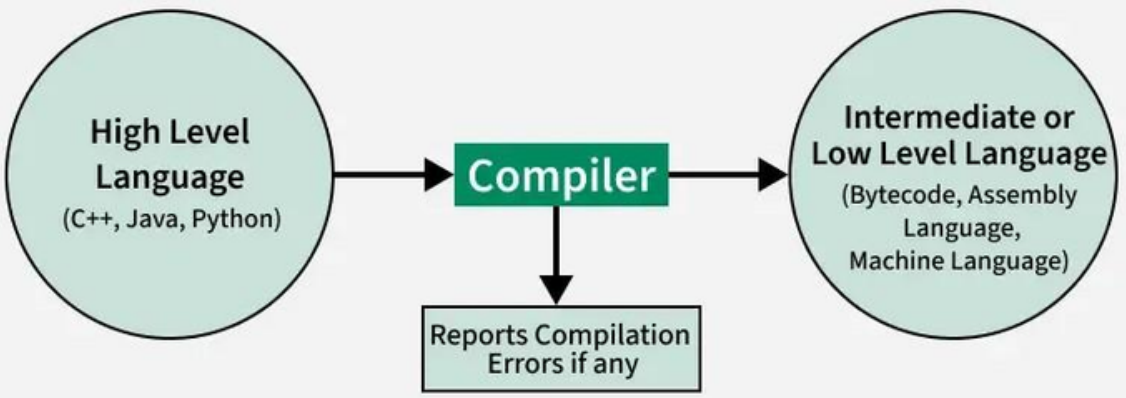
Instructor's Signature:

Introduction: A compiler is software that translates or converts a program written in a high-level language (source code) into a low-level language (typically machine code or bytecode). This MiniCompiler is a simplified educational version of a compiler, built using Python and the PLY (Python Lex-Yacc) library. It interprets a subset of a C-like language and demonstrates core compiler phases such as lexical analysis, syntax analysis (parsing), AST generation, and code interpretation.

Objectives: It supports essential language features including variable assignments, arithmetic operations, logical operations, conditional statements, loops, and print functionality. This project aims to showcase the fundamental design of a compiler.

Compiler Construction Used Tools:

- PLY (Python Lex-Yacc): For lexical analysis and parsing.
- Python: As the host language and interpreter for the generated AST.



Phases of the MiniCompiler:

1. Lexical Analysis

Lexical analysis is the first phase of the compiler. It converts the input source code into a stream of tokens. This phase is implemented using the PLY lexer.

- Detects identifiers, numbers, operators, and delimiters.
- Recognizes keywords like if, else, for, while, print, and, or, not.
- Ignores comments using //, whitespace, and newlines.

2. Syntax Analysis (Parsing)

Parsing checks whether the sequence of tokens follows the grammar of the language. This is done using PLY's yacc module.

- Builds a parse tree or AST from the token stream.
- Supports constructs like if-else, for, while, expressions, and print statements.
- Detects and reports syntax errors.

3. Abstract Syntax Tree (AST):

The AST represents the hierarchical structure of the source program. Each node corresponds to a language construct:

- Assignment, Print
- IfElse, ForLoop, WhileLoop
- BinOp, UnOp, Var, Number

4. Code Execution (Interpretation):

Instead of converting code to machine language, the MiniCompiler directly executes the AST using a custom interpreter. This phase mimics code generation by walking the AST and performing actions.

- Executes conditionals and loops.
- Evaluates expressions.
- Stores and updates variable values in a symbol table.

Symbol Table in MiniCompiler:

The symbol table is a dictionary that maps variable names to their current values during execution. It supports dynamic scoping and variable reuse.

Error Detection and Handling:

- Syntax errors during parsing are reported with line numbers and offending tokens.
- Lexical errors (illegal characters) are caught by the lexer.

Language Features Supported:

- Variable assignments (e.g., x = 5;)
- Arithmetic: +, -, *, /
- Logical: and, or, not
- Comparison: <, >, <=, >=, ==, !=
- Conditional: if, else
- Loops: for, while
- Output: print
- Single-line comments: // this is ignored

Example Program

```
x = 5;
y = 10;
if x < y {
    print x + y;
} else {
    print x - y;
}

while x < 8 {
    print x;
    x = x + 1;
}
```

Output:

```
15
5
6
7
```

```
a = 5;
b = 10;
c = 0;

print a + b;        // addition
print b - a;        // subtraction

c = (a < b) and (b > 0);
print c;            // true

c = (a > b) or (b > 0);
print c;            // true

c = not (a > b);
print c;            // true

while a < 8 {
    print a;
    a = a + 1;
}
```

Technologies Used:

- Python 3.x
- PLY (Python Lex-Yacc)
- PyCharm IDE
- CLI-based execution

Conclusion:

This MiniCompiler project demonstrates how basic compiler phases work together to translate and execute a high-level language. It simulates the key parts of a compiler's frontend including lexical analysis, parsing, AST construction, and code execution.

Understanding these steps lays a strong foundation for deeper study into full compiler construction, optimizations, and language implementation.