Compiler Construction — Lab 1
Expression Evaluator

CS-471L Students
University of Engineering and Technology, Lahore

February 9, 2026

Overview

This document describes the implementations for Task 1 (tokenizer) and Task 3 (parser and evaluator with variables) for Lab 1. It explains how to install Python, run the scripts, and summarizes the behavior and sample outputs.

1 Environment and Requirements

- Python 3.8 or later (3.10/3.11/3.12 recommended).
- No external packages required; the code uses only Python standard library.

Installing Python (Windows)

1. Download the installer from `https://www.python.org/downloads/` (choose latest stable 3.x).
2. Run the installer and check "Add Python to PATH" before clicking Install.
3. Verify installation in PowerShell or Command Prompt:

```
python --version
# or
py -3 --version
```

4. If you use a virtual environment, create one and activate before running the lab scripts:

```
python -m venv venv
venv\Scripts\activate   % on Windows
```

2 Files

- **task1.py** – Tokenizer (breaks input into tokens, prints step-by-step tokenization).
- **task3.py** – Parser and evaluator (recursive-descent parsing, AST, variable assignment, human-readable evaluation steps).

3 How to run

Open a terminal in the folder `Lab1` and run:

```
python task1.py
python task3.py
```

Each program reads expressions from standard input (one per line). Press Enter on an empty line to quit.

4 Task 1 – Tokenizer (Brief Description)

The tokenizer reads the input string and produces a linear list of tokens. Token types used in the provided implementation are:

- **NUMBER**: integer numeric literal (e.g. 42)
- **IDENT**: identifier/variable name (e.g. x, total1)
- **OP**: operators and punctuation (e.g. +, -, *, /, (,), =)
- **EOF**: end of input marker

The tokenizer also prints each discovered token as it scans, followed by the full token list.

Sample run (task1)

```
Input expression: x = 10
Tokenizing
  Found IDENTIFIER -> x
  Found OPERATOR -> =
  Found NUMBER -> 10
  Found EOF (End of Expression)
All tokens collected: ['IDENT:x', 'OP:=', 'NUMBER:10', 'EOF']
```

5 Task 3 – Parser and Evaluator (Brief Description)

The parser implements a simple recursive-descent parser with the grammar:

```
expr   -> term ((+|-) term)*
term   -> factor ((*|/) factor)*
factor -> ( expr ) | NUMBER | IDENT
statement -> IDENT '=' expr | expr
```

The evaluator builds a small AST (Number, Var, BinOp, Assign) during parsing and then evaluates it. During evaluation the program records human-readable steps such as:

- Number literal: 3
- Lookup variable x =¿ 10
- Evaluating left side of PLUS
- Computed 3 PLUS 4 = 7

After evaluation it prints the numbered steps and the current symbol table (variables and values).

Sample run (task3)

```
Expression: x = 10
Tokens: [IDENT:x, OP:=, NUMBER:10]
AST:
Assign(x)
  Number(10)

Evaluation steps:
  Step 1: Evaluating assignment to x
  Step 2: Number literal: 10
  Step 3: Assigned x = 10

Result: Variable x = 10

Symbol table:
  x = 10
```

6 Design notes and tips

- The implementations are intentionally simple and readable: they avoid advanced Python features so students can follow the logic.
- The tokenizer uses linear scanning; the parser uses explicit recursive-descent functions for `expr`, `term`, and `factor`.
- Error handling is minimal: syntax or name errors will print an explanatory message.

7 Extending the project

Possible improvements:

- Support unary operators (unary minus)
- Add floating-point numbers to the evaluator
- Produce a graphical representation of the AST (e.g., DOT/Graphviz)
- Add unit tests (`unittest` or `pytest`) for tokenizer, parser, and evaluator

Acknowledgements

Reference: Aho, Lam, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools* (Dragon Book).