Group 37

C++ Parser in Python

Dheeraj Kumar Singh (21114034)

Garvit Goyal (21114036)

Kulkarni Sourabh Shrinivasrao (21114053)

Problem Statement (PID = 16)

Develop a parser in Python language that accepts code in C++ and checks for syntax errors (Expected: loops, integers, if-else).

Repository Link

Description of The Code and Methods Used

The project consists of two jupyter notebook files

1. Lexer.ipynb
2. Parser.ipynb

Lexer.ipynb

This program parses the input C++ file, generating all the tokens from it. Subsequently, it stores these tokens in a list cache named "tokens".

Parser.ipynb

This file further processes the tokens generated by the lexer and simultaneously manages a symbol table to verify if any variable is undeclared within its scope. The primary loop initializes an iterator called "i," which begins at 0 and iterates through the tokens list.

In C++, a program is limited to containing variable declarations/definitions, function declarations/definitions in the global scope, as well as certain preprocessor directives and "using" statements. This structure aligns with the following BNF grammar:

Program → function def | function decl | variable decl | variable def | preprocessor | using statement

Now, I combine function declaration/definition and variable declaration/definition into a function named:

1. **checkFuncDefOrVarDef**, which returns [index, err]:
   - This function checks for function or variable declaration or definition.

Another function checks for preprocessor directives:

2. **ignorePreprocessor**, which returns [index, err]:

- This function is called when a '#' is found as the first character in a token.

- It parses the preprocessor directives.

And one more function simply ignores the "using ..." statements:

3. **ignoreTillSemicolon**, which returns [index, err]:

- This function is called when the initial token is "using".

- It ignores all statements until encountering a semicolon.

Now, we have other functions that are called while parsing the function declaration/definition and variable declarations/definitions, namely:

4. **checkVarHeader**, which returns [index, err, dtype, ident]:

- dtype : datatype of the matched identifier

- ident : matched identifier

- This function matches the pattern <dtype> <ident> , for example, 'int a' or 'int main', etc.

5. **checkScope**, which returns [index, err]:

- This function handles everything inside curly braces '{' and '}', for example, function declaration scope, if scope, else scope, while scope, etc.

- It also takes the symbol table from the parent scope to check for all declared variables in the parent scope and updates the symbol table in its own scope.

6. **checkExpr**, which returns [index, err, dtype]:

- This function parses all expressions and checks for the datatype on which the operators work (for example, we cannot add an int and a string). It also checks if the operator is valid on the operand provided, considering the lvalue or rvalue property of the operand.

7. **checkLval**, which returns [index, err, dtype, islval]:

- This function parses the lvalue in an expression and returns if the value is an actual lvalue according to the specification of C++.

The lval and expr functions together parse the expression based on the grammar defined below:

expr → lval operand expr | lval

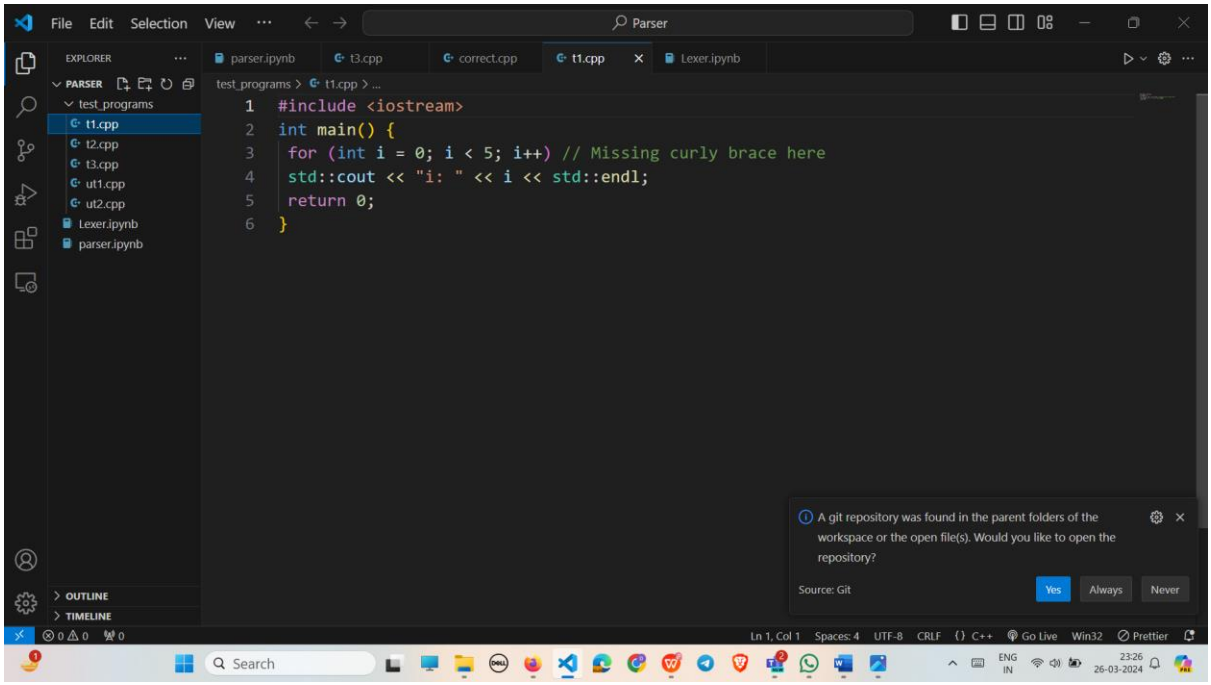
lval → unary_op lval | identifier | identifier(param_list) | lval post_op | pre_op lval

8. **checkFuncDecOrValDef**, which returns [index, err, symbol_table]:

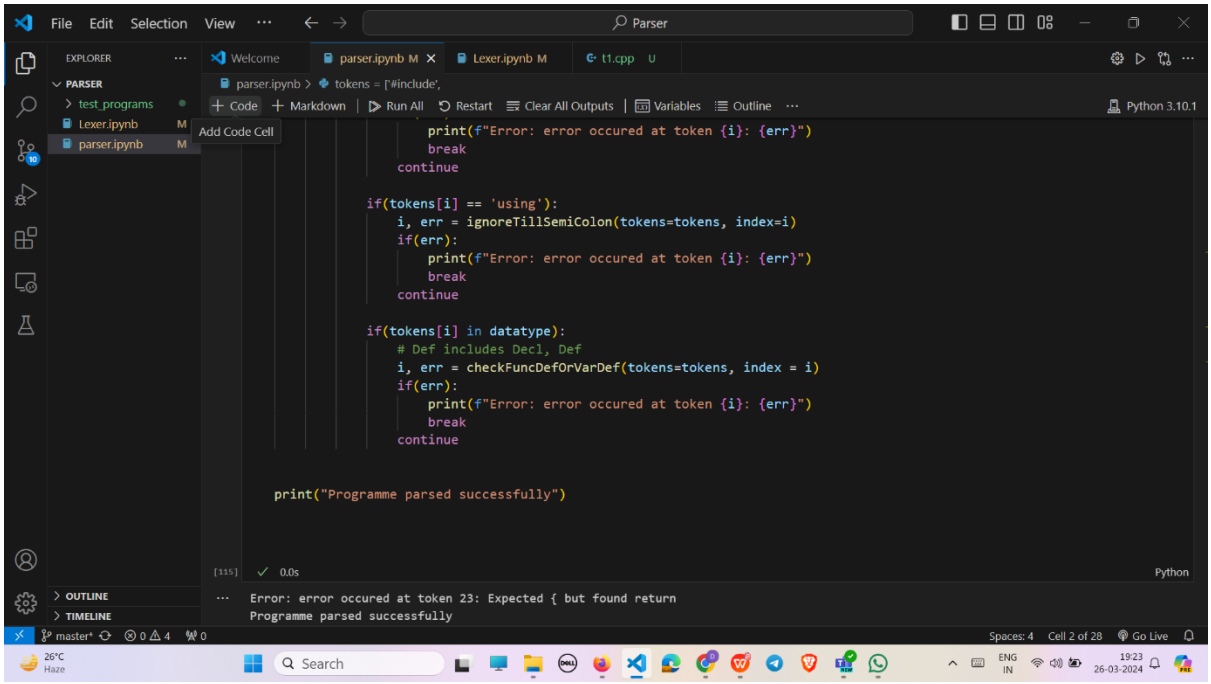
- This function parses only function declaration and variable declaration/definition, as function definition is not allowed inside any scope.

The detailed procedure for running the program is described in the readme file named “readme.md”.

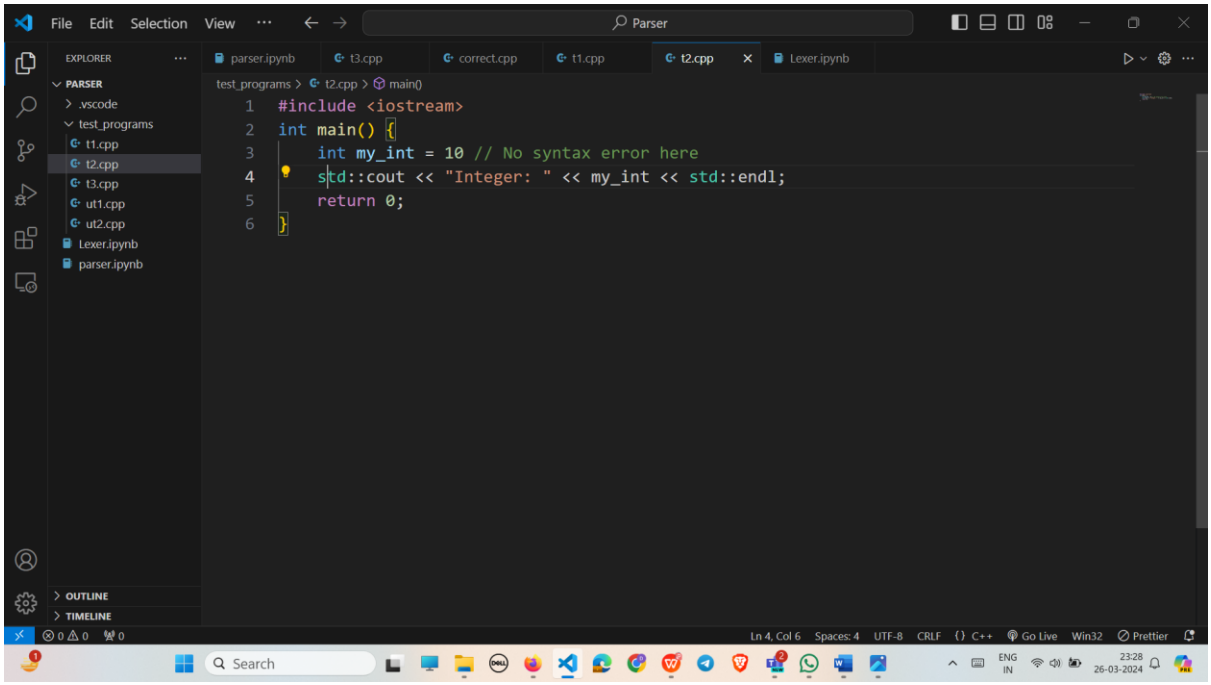
Test Case 1:



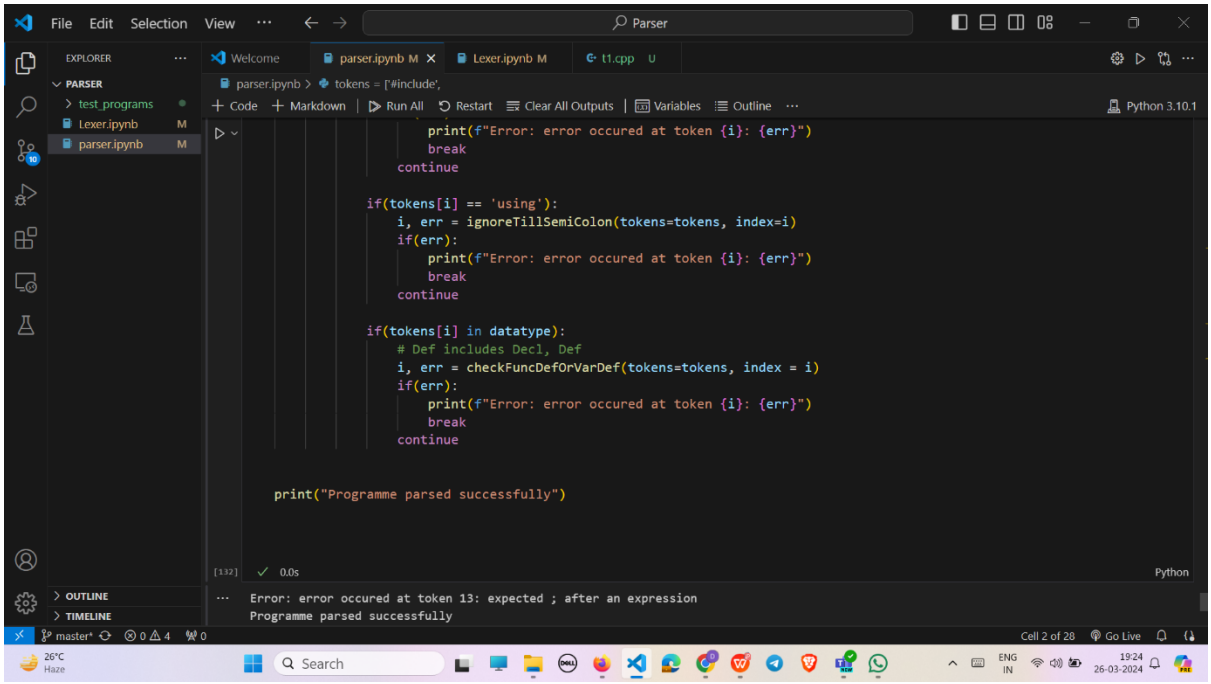
Output:



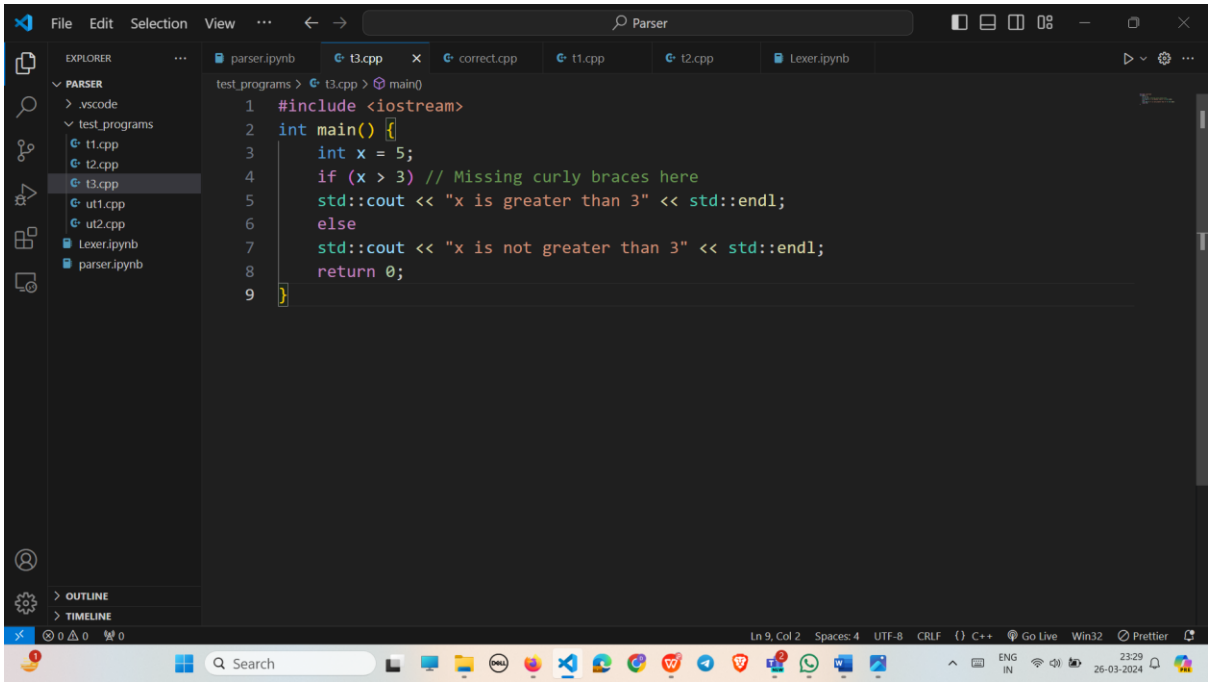
Test Case 2:



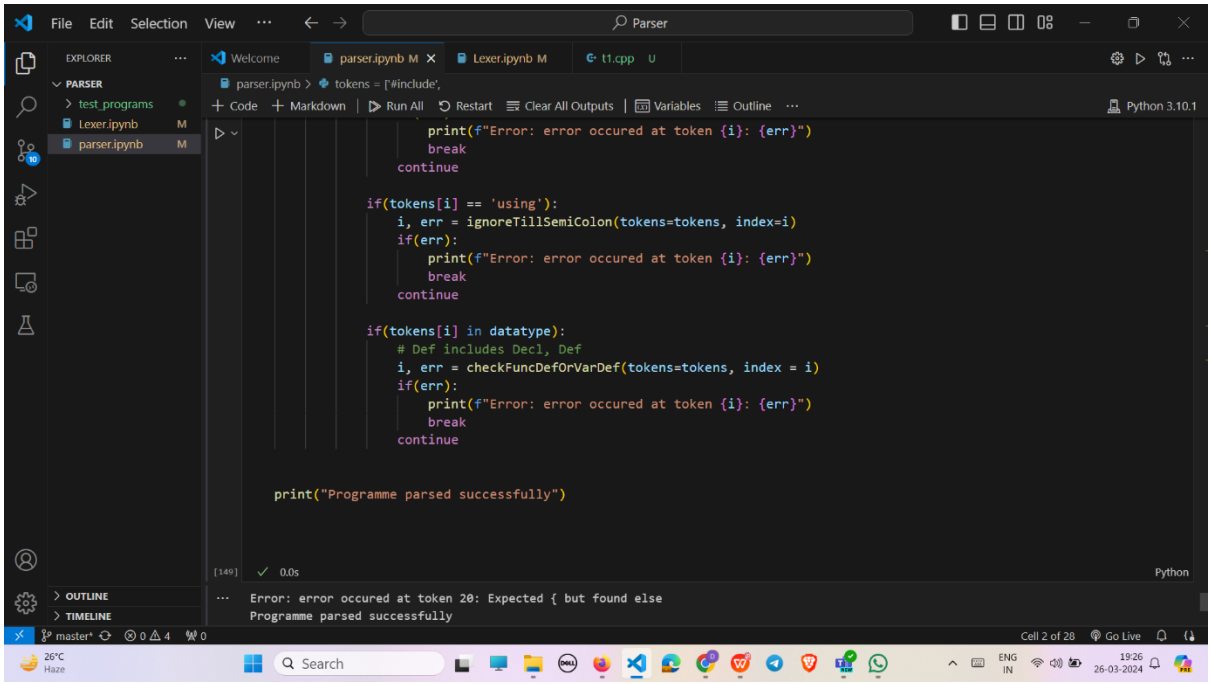
Output:



Test Case 3:

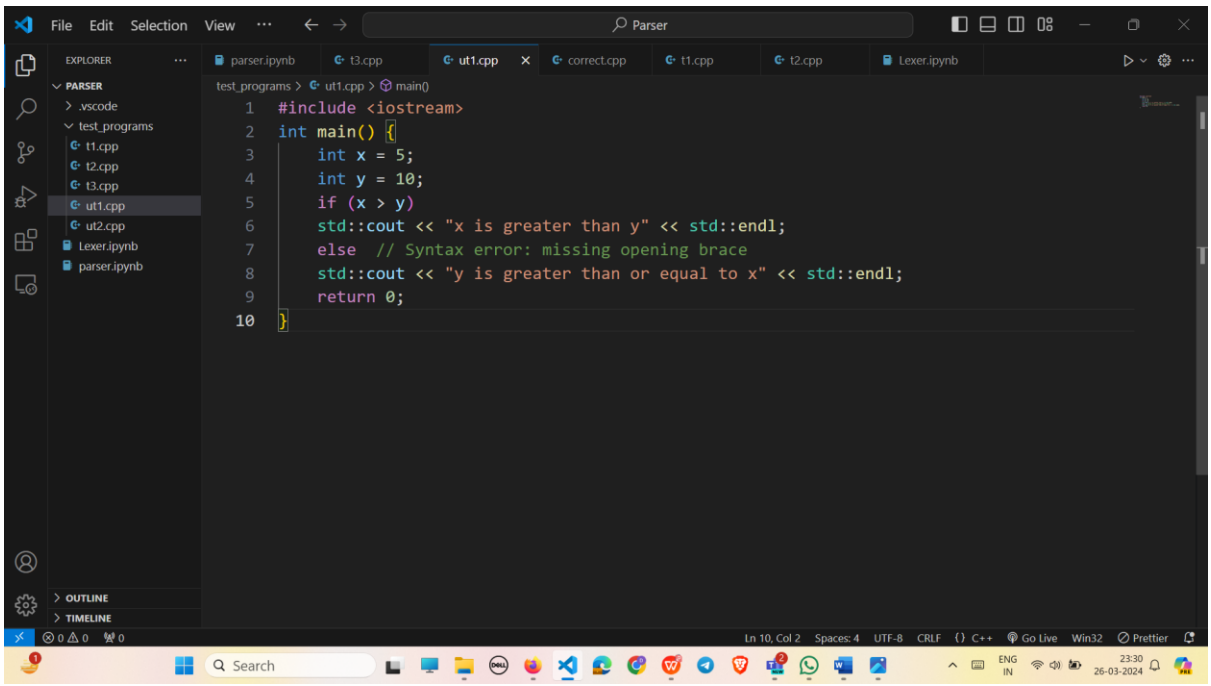


Output:

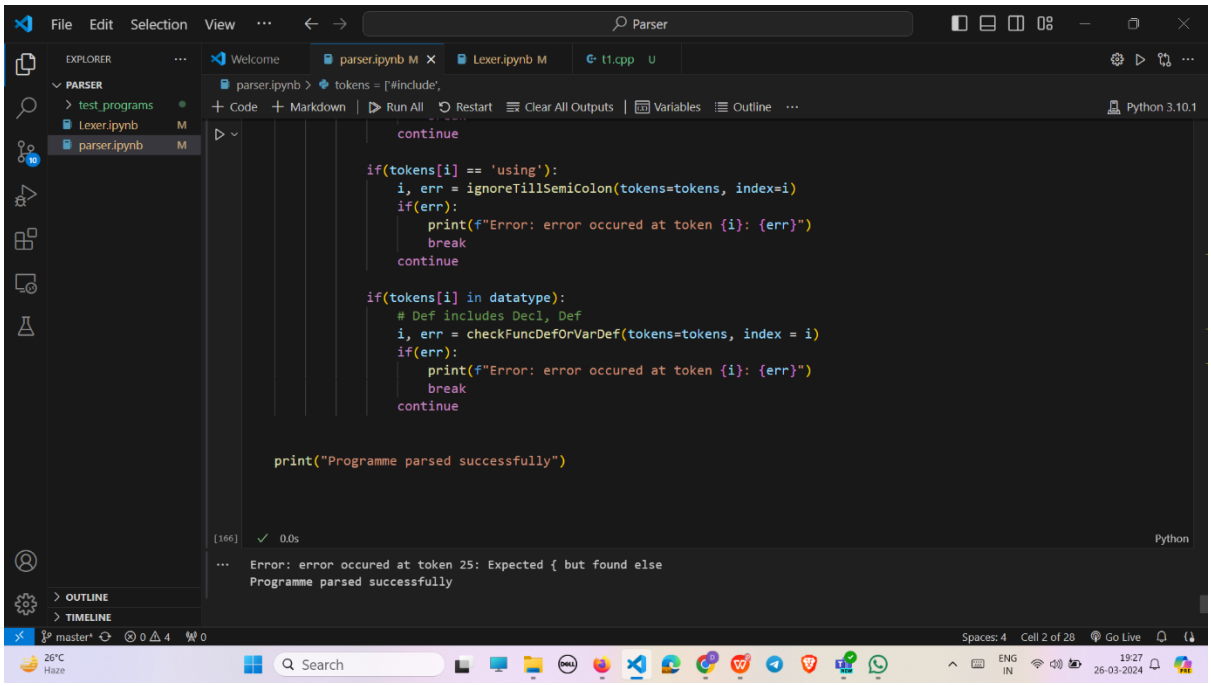


Test Cases for Evaluation:

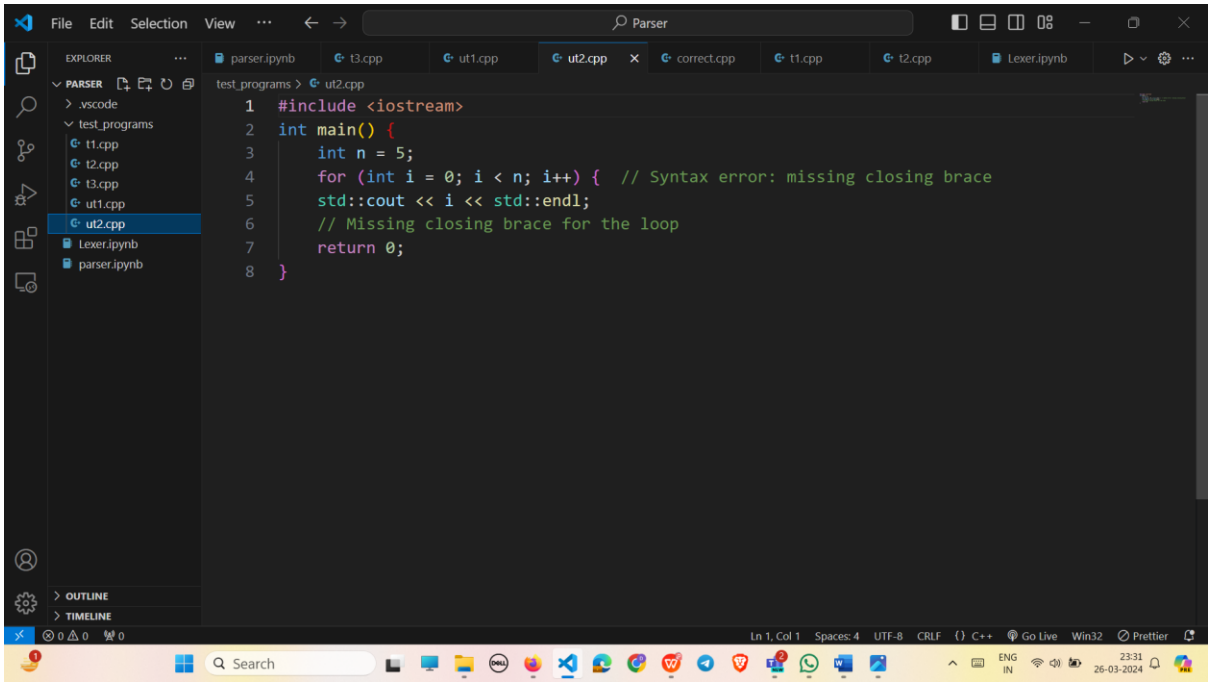
Test Case 1:



Output:



Test Case 2:



Output: