Team Members:

Aakash Desai: PES1UG24CS006

Aarush Muralidhara: PES1UG24CS010

Choice of Input Language: C++

Constructs Chosen:

1) Array Declaration

2) Function Definition

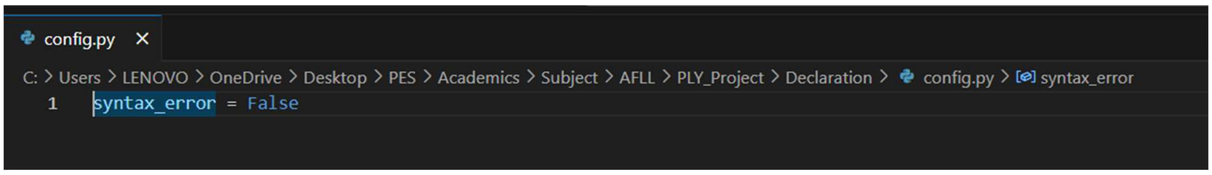
3) Function Declaration (Prototype)

4) Selection Statements (If/If-Else)

5) Simple Data-Type (Declaration)

For our project, the files named config.py is shared by the main.py, lexer.py and parser (for each construct).

It is as shown:

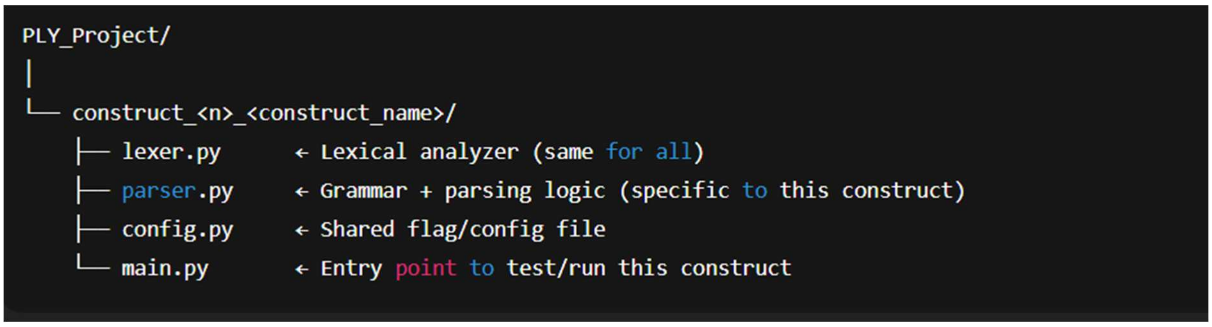


config.py ×

C: > Users > LENOVO > OneDrive > Desktop > PES > Academics > Subject > AFLL > PLY_Project > Declaration > config.py > syntax_error

```
syntax_error = False
```

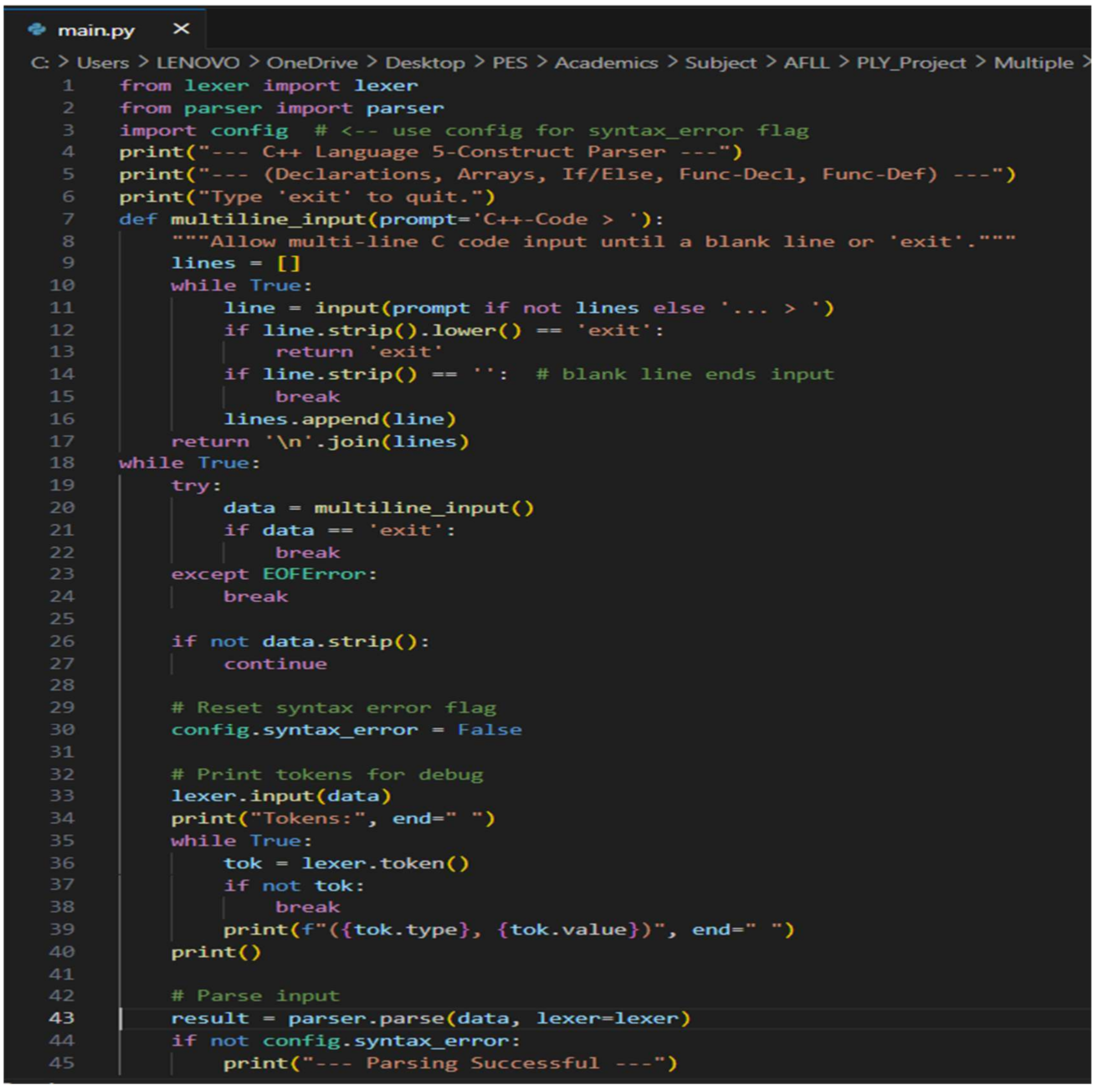
The general structure of every folder is as follows:



```
PLY_Project/
|
└── construct_<n>_<construct_name>/
    ├── lexer.py      ← Lexical analyzer (same for all)
    ├── parser.py     ← Grammar + parsing logic (specific to this construct)
    ├── config.py     ← Shared flag/config file
    └── main.py       ← Entry point to test/run this construct
```

Every construct (and construct folder) shares the same main.py file and lexer.py

Main.py file

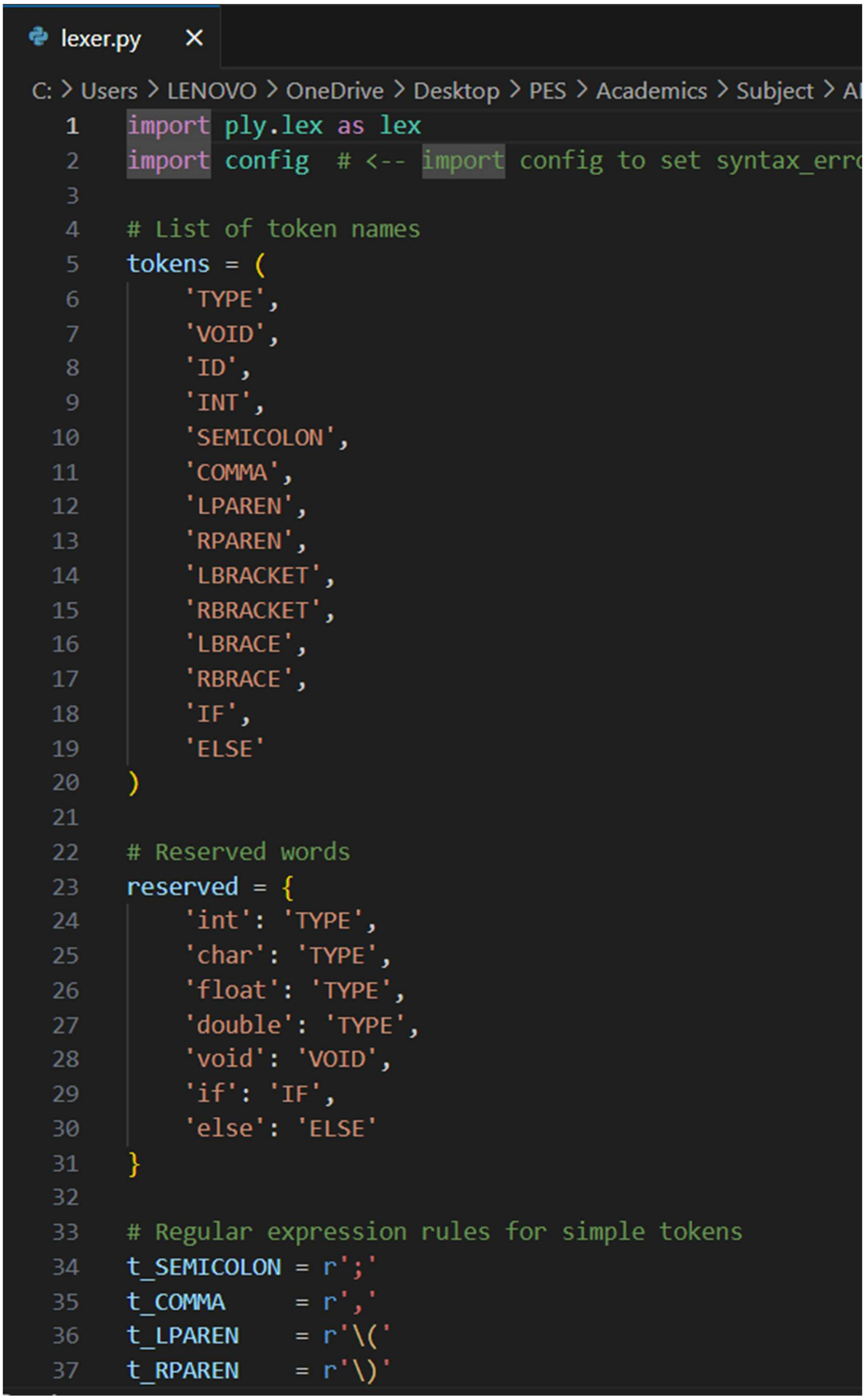

main.py
C: > Users > LENOVO > OneDrive > Desktop > PES > Academics > Subject > AFLL > PLY_Project > Multiple >

```python
from lexer import lexer
from parser import parser
import config  # <-- use config for syntax_error flag
print("--- C++ Language 5-Construct Parser ---")
print("--- (Declarations, Arrays, If/Else, Func-Decl, Func-Def) ---")
print("Type 'exit' to quit.")
def multiline_input(prompt='C++-Code > '):
    """Allow multi-line C code input until a blank line or 'exit'."""
    lines = []
    while True:
        line = input(prompt if not lines else '... > ')
        if line.strip().lower() == 'exit':
            return 'exit'
        if line.strip() == '':  # blank line ends input
            break
        lines.append(line)
    return '\n'.join(lines)
while True:
    try:
        data = multiline_input()
        if data == 'exit':
            break
    except EOFError:
        break

    if not data.strip():
        continue

    # Reset syntax error flag
    config.syntax_error = False

    # Print tokens for debug
    lexer.input(data)
    print("Tokens:", end=" ")
    while True:
        tok = lexer.token()
        if not tok:
            break
        print(f"({tok.type}, {tok.value})", end=" ")
    print()

    # Parse input
    result = parser.parse(data, lexer=lexer)
    if not config.syntax_error:
        print("--- Parsing Successful ---")
```

The main.py file serves as the driver program that connects the lexer and parser components of the project. It allows the user to enter C++-like code interactively, tokenizes the input using the lexer, and then parses it using the grammar rules defined in parser.py. The script displays the sequence of tokens generated and reports whether the parsing was successful or if any syntax errors were found. It continuously accepts input until the user types exit, making it easy to test various code snippets for different C++ constructs. In short, main.py acts as the bridge between lexical analysis and syntax analysis, providing a clear way to verify if the entered code follows the defined grammar.

The only change that is to be made in each main.py file would be in line 2, importing from the matching parser.py file.

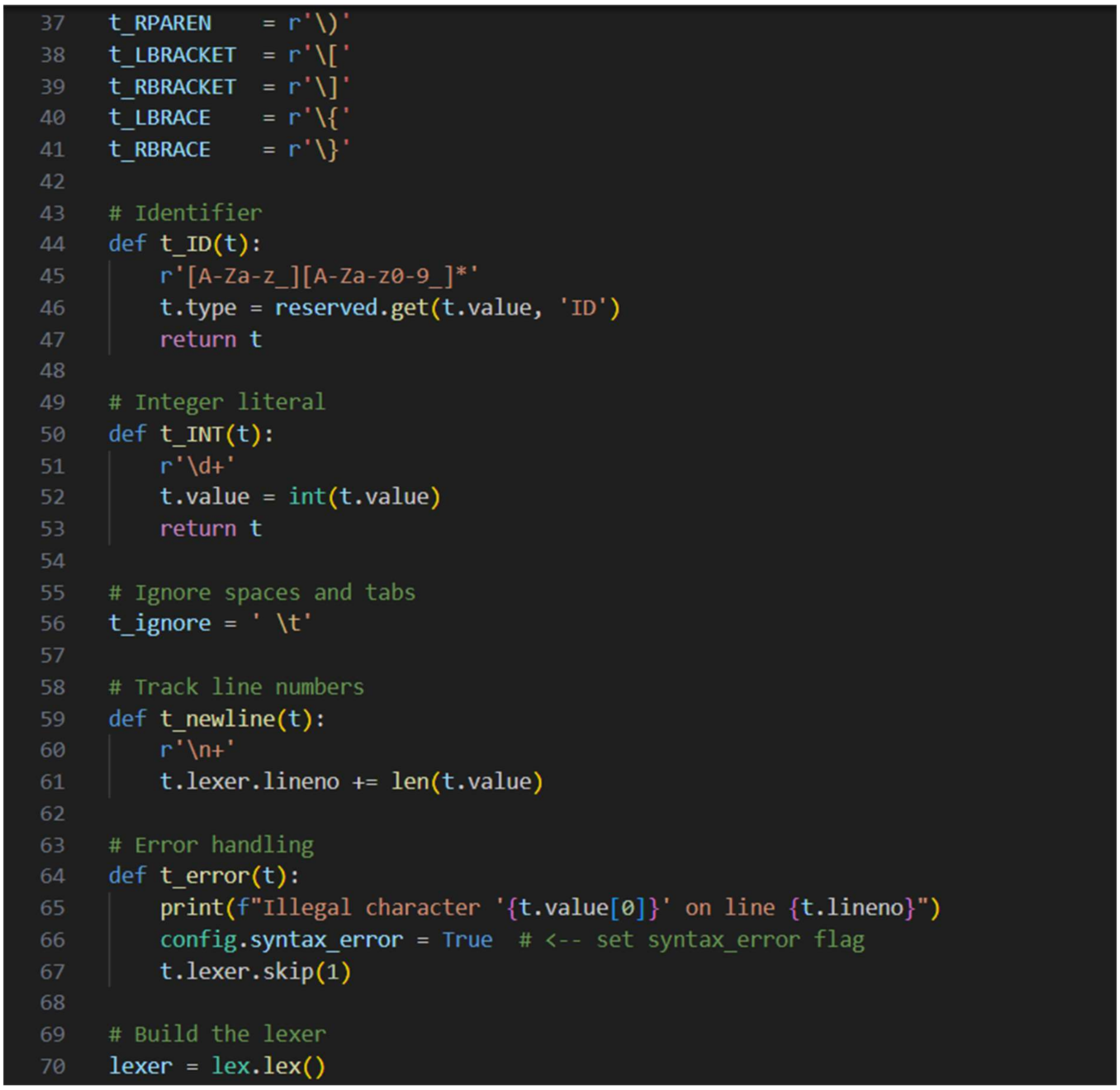
Lexer.py



lexer.py

C: > Users > LENOVO > OneDrive > Desktop > PES > Academics > Subject > AI

```python
import ply.lex as lex
import config  # <-- import config to set syntax_erro

# List of token names
tokens = (
    'TYPE',
    'VOID',
    'ID',
    'INT',
    'SEMICOLON',
    'COMMA',
    'LPAREN',
    'RPAREN',
    'LBRACKET',
    'RBRACKET',
    'LBRACE',
    'RBRACE',
    'IF',
    'ELSE'
)

# Reserved words
reserved = {
    'int': 'TYPE',
    'char': 'TYPE',
    'float': 'TYPE',
    'double': 'TYPE',
    'void': 'VOID',
    'if': 'IF',
    'else': 'ELSE'
}

# Regular expression rules for simple tokens
t_SEMICOLON = r';'
t_COMMA     = r','
t_LPAREN    = r'\('
t_RPAREN    = r'\)'
```

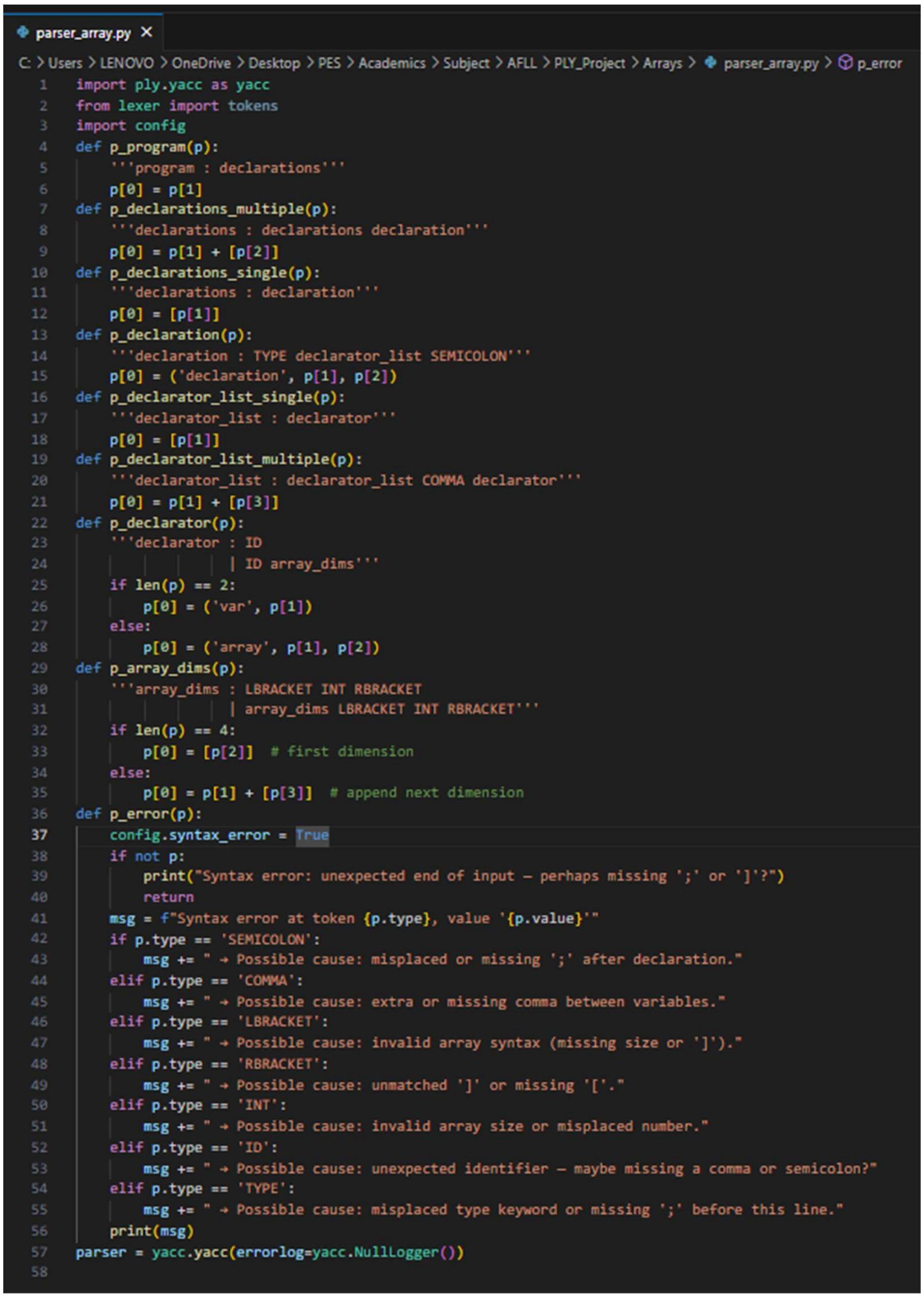


```python
t_RPAREN    = r'\)'
t_LBRACKET  = r'\['
t_RBRACKET  = r'\]'
t_LBRACE    = r'\{'
t_RBRACE    = r'\}'

# Identifier
def t_ID(t):
    r'[A-Za-z_][A-Za-z0-9_]*'
    t.type = reserved.get(t.value, 'ID')
    return t

# Integer literal
def t_INT(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Ignore spaces and tabs
t_ignore = ' \t'

# Track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# Error handling
def t_error(t):
    print(f"Illegal character '{t.value[0]}' on line {t.lineno}")
    config.syntax_error = True  # <-- set syntax_error flag
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()
```

The lexer.py file is responsible for lexical analysis, which is the first phase of the compiler process. It uses the PLY (Python Lex-Yacc) library to break the user’s C++-like source code into a sequence of tokens, such as keywords (int, if, else), identifiers, numbers, operators, and punctuation symbols. Each token is defined by a regular expression rule, allowing the lexer to recognize valid lexical patterns in the code. The lexer also ignores unnecessary elements like spaces and comments while keeping track of line numbers for accurate error reporting. In short, lexer.py transforms raw input text into structured tokens that are passed to the parser for syntax analysis.

Every Construct has its own unique parser.

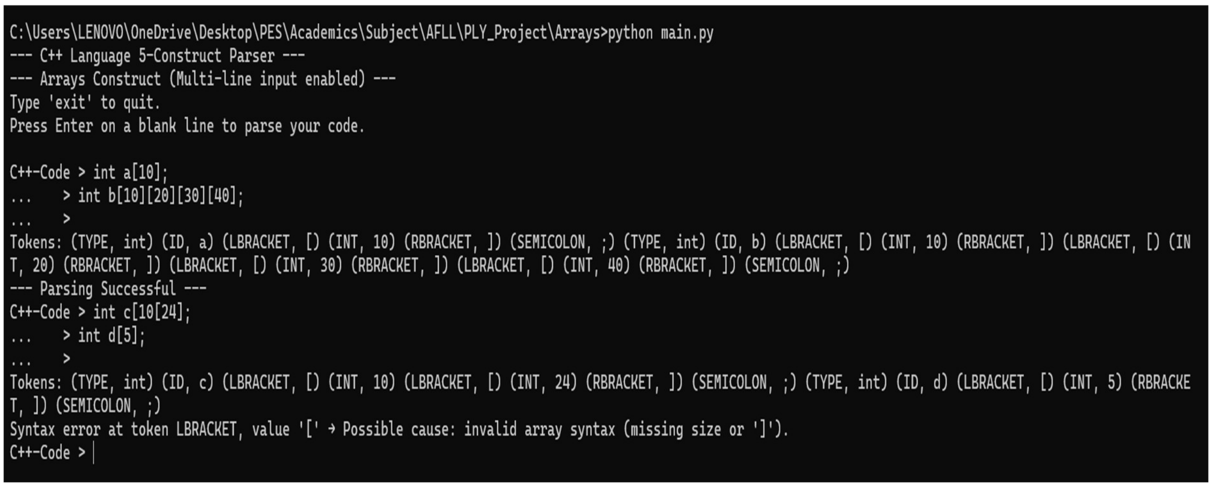
1)Array Construct



parser_array.py ×
C: > Users > LENOVO > OneDrive > Desktop > PES > Academics > Subject > AFLL > PLY_Project > Arrays > parser_array.py > p_error

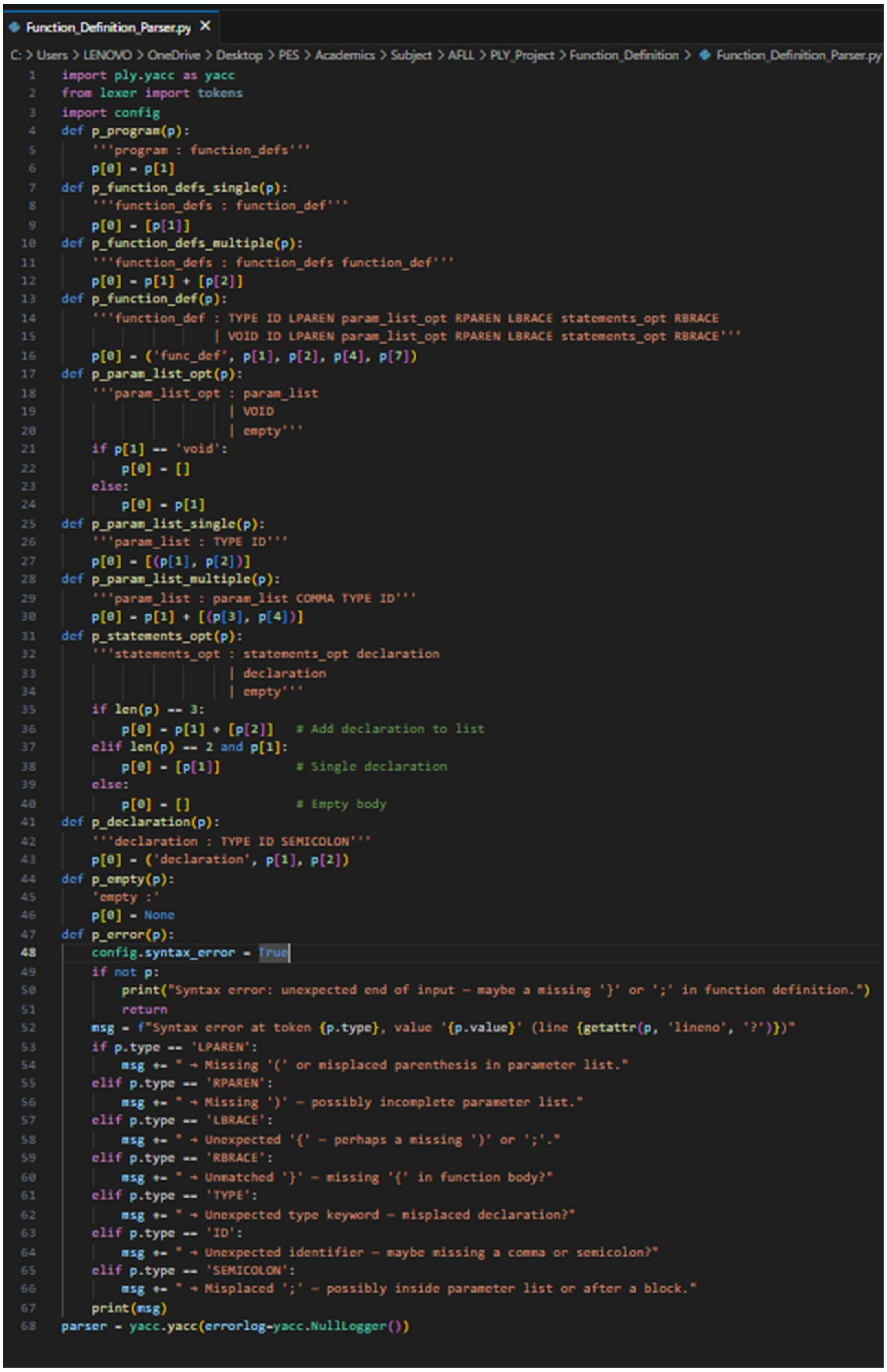
```python
import ply.yacc as yacc
from lexer import tokens
import config
def p_program(p):
    '''program : declarations'''
    p[0] = p[1]
def p_declarations_multiple(p):
    '''declarations : declarations declaration'''
    p[0] = p[1] + [p[2]]
def p_declarations_single(p):
    '''declarations : declaration'''
    p[0] = [p[1]]
def p_declaration(p):
    '''declaration : TYPE declarator_list SEMICOLON'''
    p[0] = ('declaration', p[1], p[2])
def p_declarator_list_single(p):
    '''declarator_list : declarator'''
    p[0] = [p[1]]
def p_declarator_list_multiple(p):
    '''declarator_list : declarator_list COMMA declarator'''
    p[0] = p[1] + [p[3]]
def p_declarator(p):
    '''declarator : ID
                  | ID array_dims'''
    if len(p) == 2:
        p[0] = ('var', p[1])
    else:
        p[0] = ('array', p[1], p[2])
def p_array_dims(p):
    '''array_dims : LBRACKET INT RBRACKET
                  | array_dims LBRACKET INT RBRACKET'''
    if len(p) == 4:
        p[0] = [p[2]]  # first dimension
    else:
        p[0] = p[1] + [p[3]]  # append next dimension
def p_error(p):
    config.syntax_error = True
    if not p:
        print("Syntax error: unexpected end of input — perhaps missing ';' or ']'?")
        return
    msg = f"Syntax error at token {p.type}, value '{p.value}'"
    if p.type == 'SEMICOLON':
        msg += " → Possible cause: misplaced or missing ';' after declaration."
    elif p.type == 'COMMA':
        msg += " → Possible cause: extra or missing comma between variables."
    elif p.type == 'LBRACKET':
        msg += " → Possible cause: invalid array syntax (missing size or ']')."
    elif p.type == 'RBRACKET':
        msg += " → Possible cause: unmatched ']' or missing '['."
    elif p.type == 'INT':
        msg += " → Possible cause: invalid array size or misplaced number."
    elif p.type == 'ID':
        msg += " → Possible cause: unexpected identifier — maybe missing a comma or semicolon?"
    elif p.type == 'TYPE':
        msg += " → Possible cause: misplaced type keyword or missing ';' before this line."
    print(msg)
parser = yacc.yacc(errorlog=yacc.NullLogger())
```

1)Output Verification



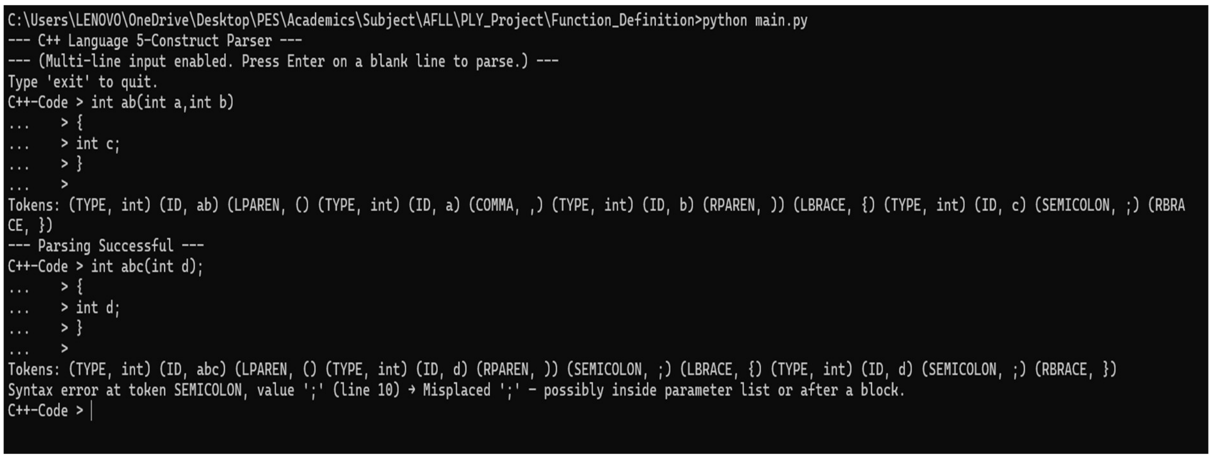
```
C:\Users\LENOVO\OneDrive\Desktop\PES\Academics\Subject\AFLL\PLY_Project\Arrays>python main.py
--- C++ Language 5-Construct Parser ---
--- Arrays Construct (Multi-line input enabled) ---
Type 'exit' to quit.
Press Enter on a blank line to parse your code.

C++-Code > int a[10];
...    > int b[10][20][30][40];
...    >
Tokens: (TYPE, int) (ID, a) (LBRACKET, [) (INT, 10) (RBRACKET, ]) (SEMICOLON, ;) (TYPE, int) (ID, b) (LBRACKET, [) (INT, 10) (RBRACKET, ]) (LBRACKET, [) (IN
T, 20) (RBRACKET, ]) (LBRACKET, [) (INT, 30) (RBRACKET, ]) (LBRACKET, [) (INT, 40) (RBRACKET, ]) (SEMICOLON, ;)
--- Parsing Successful ---
C++-Code > int c[10[24];
...    > int d[5];
...    >
Tokens: (TYPE, int) (ID, c) (LBRACKET, [) (INT, 10) (LBRACKET, [) (INT, 24) (RBRACKET, ]) (SEMICOLON, ;) (TYPE, int) (ID, d) (LBRACKET, [) (INT, 5) (RBRACKE
T, ]) (SEMICOLON, ;)
Syntax error at token LBRACKET, value '[' → Possible cause: invalid array syntax (missing size or ']').
C++-Code >
```

2)Syntax Verification for Function Definition


Function_Definition_Parser.py
C: > Users > LENOVO > OneDrive > Desktop > PES > Academics > Subject > AFLL > PLY_Project > Function_Definition > Function_Definition_Parser.py

```python
import ply.yacc as yacc
from lexer import tokens
import config
def p_program(p):
    '''program : function_defs'''
    p[0] = p[1]
def p_function_defs_single(p):
    '''function_defs : function_def'''
    p[0] = [p[1]]
def p_function_defs_multiple(p):
    '''function_defs : function_defs function_def'''
    p[0] = p[1] + [p[2]]
def p_function_def(p):
    '''function_def : TYPE ID LPAREN param_list_opt RPAREN LBRACE statements_opt RBRACE
                    | VOID ID LPAREN param_list_opt RPAREN LBRACE statements_opt RBRACE'''
    p[0] = ('func_def', p[1], p[2], p[4], p[7])
def p_param_list_opt(p):
    '''param_list_opt : param_list
                      | VOID
                      | empty'''
    if p[1] == 'void':
        p[0] = []
    else:
        p[0] = p[1]
def p_param_list_single(p):
    '''param_list : TYPE ID'''
    p[0] = [(p[1], p[2])]
def p_param_list_multiple(p):
    '''param_list : param_list COMMA TYPE ID'''
    p[0] = p[1] + [(p[3], p[4])]
def p_statements_opt(p):
    '''statements_opt : statements_opt declaration
                      | declaration
                      | empty'''
    if len(p) == 3:
        p[0] = p[1] + [p[2]]   # Add declaration to list
    elif len(p) == 2 and p[1]:
        p[0] = [p[1]]          # Single declaration
    else:
        p[0] = []              # Empty body
def p_declaration(p):
    '''declaration : TYPE ID SEMICOLON'''
    p[0] = ('declaration', p[1], p[2])
def p_empty(p):
    'empty :'
    p[0] = None
def p_error(p):
    config.syntax_error = True
    if not p:
        print("Syntax error: unexpected end of input — maybe a missing '}' or ';' in function definition.")
        return
    msg = f"Syntax error at token {p.type}, value '{p.value}' (line {getattr(p, 'lineno', '?')})"
    if p.type == 'LPAREN':
        msg += " → Missing '(' or misplaced parenthesis in parameter list."
    elif p.type == 'RPAREN':
        msg += " → Missing ')' — possibly incomplete parameter list."
    elif p.type == 'LBRACE':
        msg += " → Unexpected '{' — perhaps a missing ')' or ';'."
    elif p.type == 'RBRACE':
        msg += " → Unmatched '}' — missing '{' in function body?"
    elif p.type == 'TYPE':
        msg += " → Unexpected type keyword — misplaced declaration?"
    elif p.type == 'ID':
        msg += " → Unexpected identifier — maybe missing a comma or semicolon?"
    elif p.type == 'SEMICOLON':
        msg += " → Misplaced ';' — possibly inside parameter list or after a block."
    print(msg)
parser = yacc.yacc(errorlog=yacc.NullLogger())
```

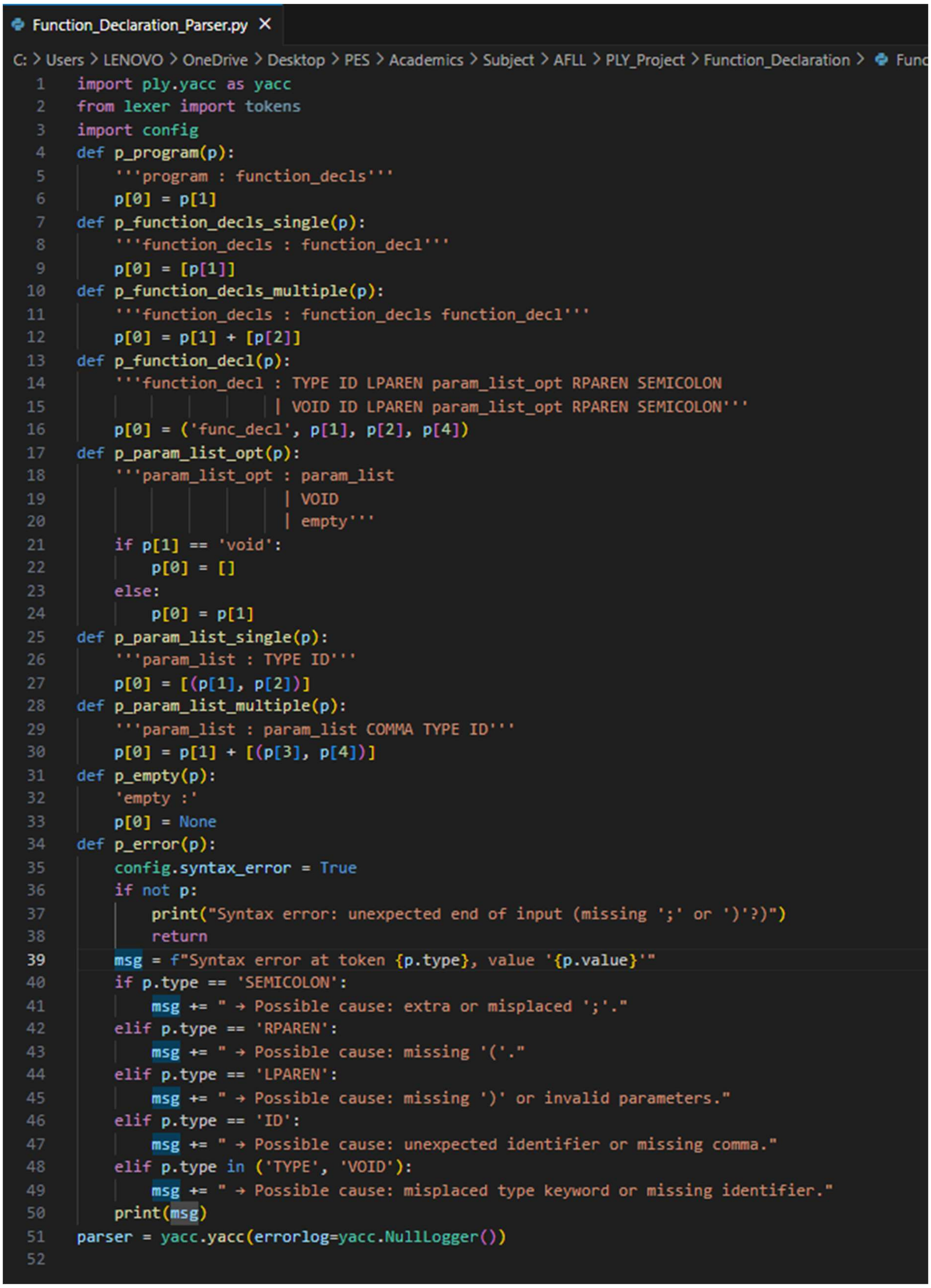
<u>2)Output Verification</u>



```
C:\Users\LENOVO\OneDrive\Desktop\PES\Academics\Subject\AFLL\PLY_Project\Function_Definition>python main.py
--- C++ Language 5-Construct Parser ---
--- (Multi-line input enabled. Press Enter on a blank line to parse.) ---
Type 'exit' to quit.
C++-Code > int ab(int a,int b)
...     > {
...     > int c;
...     > }
...     >
Tokens: (TYPE, int) (ID, ab) (LPAREN, () (TYPE, int) (ID, a) (COMMA, ,) (TYPE, int) (ID, b) (RPAREN, )) (LBRACE, {) (TYPE, int) (ID, c) (SEMICOLON, ;) (RBRA
CE, })
--- Parsing Successful ---
C++-Code > int abc(int d);
...     > {
...     > int d;
...     > }
...     >
Tokens: (TYPE, int) (ID, abc) (LPAREN, () (TYPE, int) (ID, d) (RPAREN, )) (SEMICOLON, ;) (LBRACE, {) (TYPE, int) (ID, d) (SEMICOLON, ;) (RBRACE, })
Syntax error at token SEMICOLON, value ';' (line 10) → Misplaced ';' - possibly inside parameter list or after a block.
C++-Code > 
```

___

3)Syntax verification for Function Declaration (Prototype)

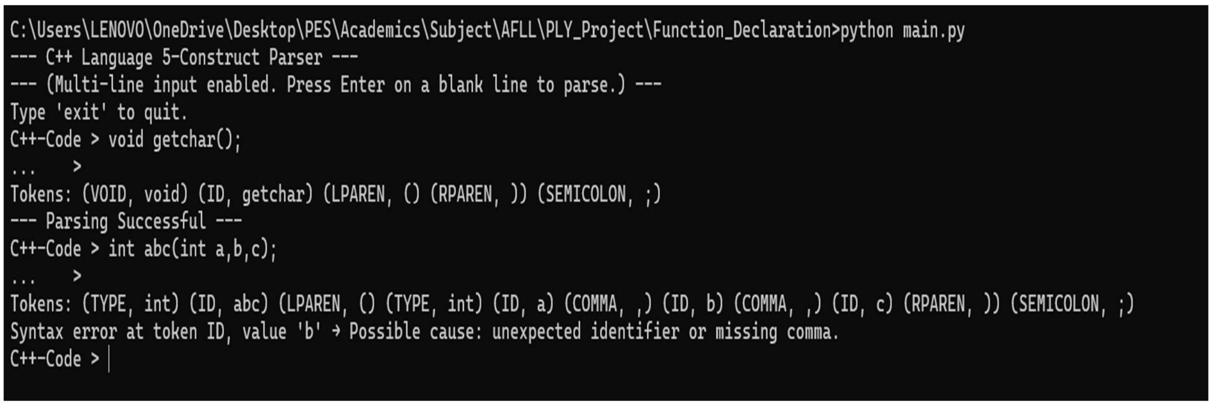


Function_Declaration_Parser.py ×

C: > Users > LENOVO > OneDrive > Desktop > PES > Academics > Subject > AFLL > PLY_Project > Function_Declaration > Func

```python
import ply.yacc as yacc
from lexer import tokens
import config
def p_program(p):
    '''program : function_decls'''
    p[0] = p[1]
def p_function_decls_single(p):
    '''function_decls : function_decl'''
    p[0] = [p[1]]
def p_function_decls_multiple(p):
    '''function_decls : function_decls function_decl'''
    p[0] = p[1] + [p[2]]
def p_function_decl(p):
    '''function_decl : TYPE ID LPAREN param_list_opt RPAREN SEMICOLON
                     | VOID ID LPAREN param_list_opt RPAREN SEMICOLON'''
    p[0] = ('func_decl', p[1], p[2], p[4])
def p_param_list_opt(p):
    '''param_list_opt : param_list
                      | VOID
                      | empty'''
    if p[1] == 'void':
        p[0] = []
    else:
        p[0] = p[1]
def p_param_list_single(p):
    '''param_list : TYPE ID'''
    p[0] = [(p[1], p[2])]
def p_param_list_multiple(p):
    '''param_list : param_list COMMA TYPE ID'''
    p[0] = p[1] + [(p[3], p[4])]
def p_empty(p):
    'empty :'
    p[0] = None
def p_error(p):
    config.syntax_error = True
    if not p:
        print("Syntax error: unexpected end of input (missing ';' or ')'?)")
        return
    msg = f"Syntax error at token {p.type}, value '{p.value}'"
    if p.type == 'SEMICOLON':
        msg += " → Possible cause: extra or misplaced ';'."
    elif p.type == 'RPAREN':
        msg += " → Possible cause: missing '('."
    elif p.type == 'LPAREN':
        msg += " → Possible cause: missing ')' or invalid parameters."
    elif p.type == 'ID':
        msg += " → Possible cause: unexpected identifier or missing comma."
    elif p.type in ('TYPE', 'VOID'):
        msg += " → Possible cause: misplaced type keyword or missing identifier."
    print(msg)
parser = yacc.yacc(errorlog=yacc.NullLogger())

```

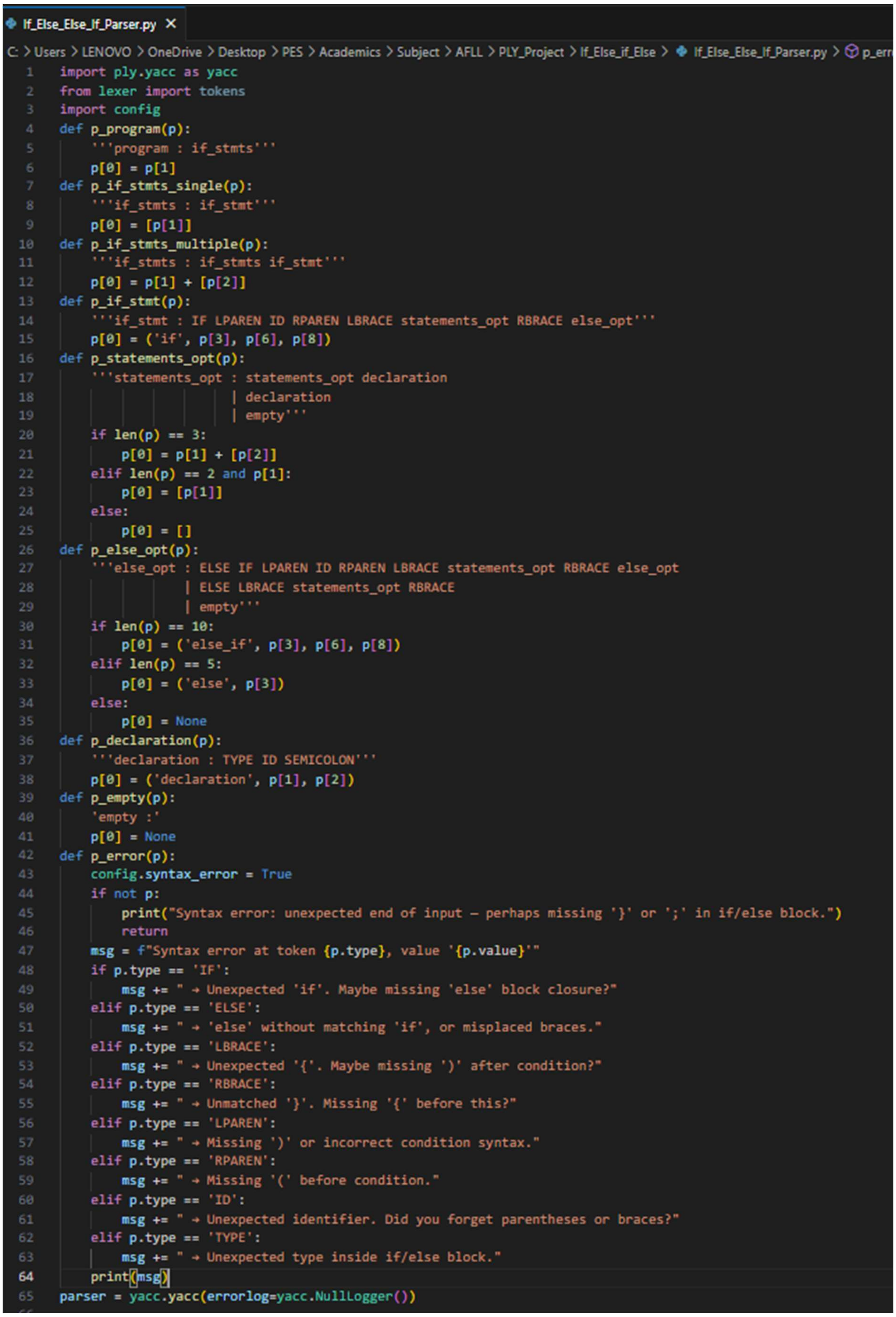
3)Output Verification



```
C:\Users\LENOVO\OneDrive\Desktop\PES\Academics\Subject\AFLL\PLY_Project\Function_Declaration>python main.py
--- C++ Language 5-Construct Parser ---
--- (Multi-line input enabled. Press Enter on a blank line to parse.) ---
Type 'exit' to quit.
C++-Code > void getchar();
...    >
Tokens: (VOID, void) (ID, getchar) (LPAREN, () (RPAREN, )) (SEMICOLON, ;)
--- Parsing Successful ---
C++-Code > int abc(int a,b,c);
...    >
Tokens: (TYPE, int) (ID, abc) (LPAREN, () (TYPE, int) (ID, a) (COMMA, ,) (ID, b) (COMMA, ,) (ID, c) (RPAREN, )) (SEMICOLON, ;)
Syntax error at token ID, value 'b' → Possible cause: unexpected identifier or missing comma.
C++-Code > 
```

____

4)If- Else Construct

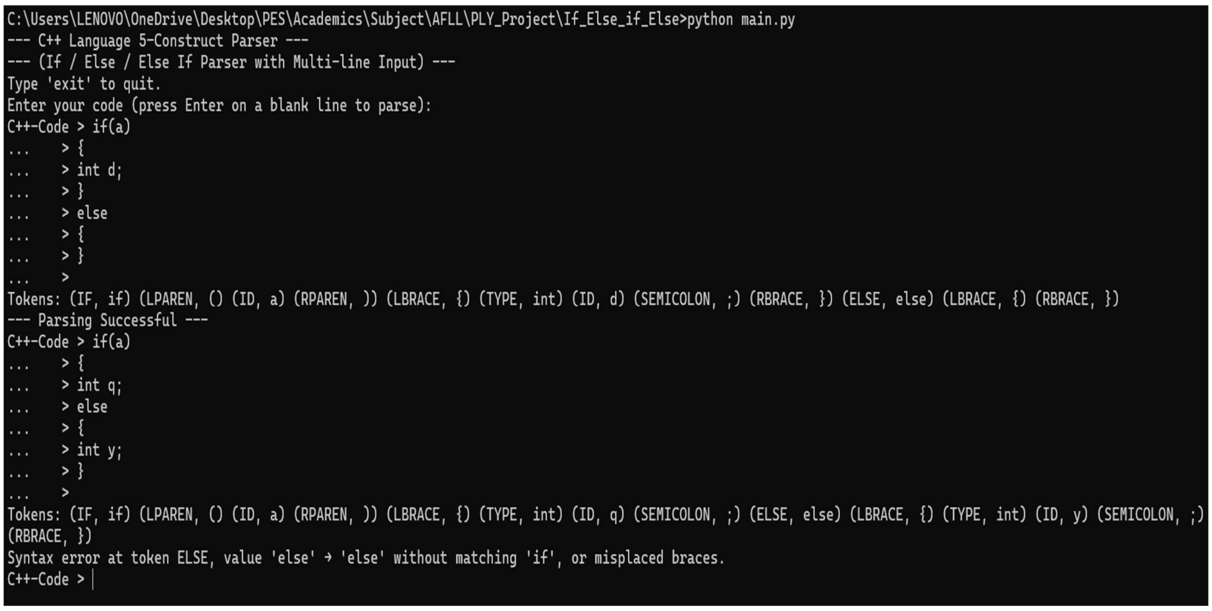


If_Else_Else_If_Parser.py ×

C: > Users > LENOVO > OneDrive > Desktop > PES > Academics > Subject > AFLL > PLY_Project > If_Else_if_Else > If_Else_Else_If_Parser.py > p_err

```python
import ply.yacc as yacc
from lexer import tokens
import config
def p_program(p):
    '''program : if_stmts'''
    p[0] = p[1]
def p_if_stmts_single(p):
    '''if_stmts : if_stmt'''
    p[0] = [p[1]]
def p_if_stmts_multiple(p):
    '''if_stmts : if_stmts if_stmt'''
    p[0] = p[1] + [p[2]]
def p_if_stmt(p):
    '''if_stmt : IF LPAREN ID RPAREN LBRACE statements_opt RBRACE else_opt'''
    p[0] = ('if', p[3], p[6], p[8])
def p_statements_opt(p):
    '''statements_opt : statements_opt declaration
                      | declaration
                      | empty'''
    if len(p) == 3:
        p[0] = p[1] + [p[2]]
    elif len(p) == 2 and p[1]:
        p[0] = [p[1]]
    else:
        p[0] = []
def p_else_opt(p):
    '''else_opt : ELSE IF LPAREN ID RPAREN LBRACE statements_opt RBRACE else_opt
                | ELSE LBRACE statements_opt RBRACE
                | empty'''
    if len(p) == 10:
        p[0] = ('else_if', p[3], p[6], p[8])
    elif len(p) == 5:
        p[0] = ('else', p[3])
    else:
        p[0] = None
def p_declaration(p):
    '''declaration : TYPE ID SEMICOLON'''
    p[0] = ('declaration', p[1], p[2])
def p_empty(p):
    'empty :'
    p[0] = None
def p_error(p):
    config.syntax_error = True
    if not p:
        print("Syntax error: unexpected end of input — perhaps missing '}' or ';' in if/else block.")
        return
    msg = f"Syntax error at token {p.type}, value '{p.value}'"
    if p.type == 'IF':
        msg += " → Unexpected 'if'. Maybe missing 'else' block closure?"
    elif p.type == 'ELSE':
        msg += " → 'else' without matching 'if', or misplaced braces."
    elif p.type == 'LBRACE':
        msg += " → Unexpected '{'. Maybe missing ')' after condition?"
    elif p.type == 'RBRACE':
        msg += " → Unmatched '}'. Missing '{' before this?"
    elif p.type == 'LPAREN':
        msg += " → Missing ')' or incorrect condition syntax."
    elif p.type == 'RPAREN':
        msg += " → Missing '(' before condition."
    elif p.type == 'ID':
        msg += " → Unexpected identifier. Did you forget parentheses or braces?"
    elif p.type == 'TYPE':
        msg += " → Unexpected type inside if/else block."
    print(msg)
parser = yacc.yacc(errorlog=yacc.NullLogger())
```

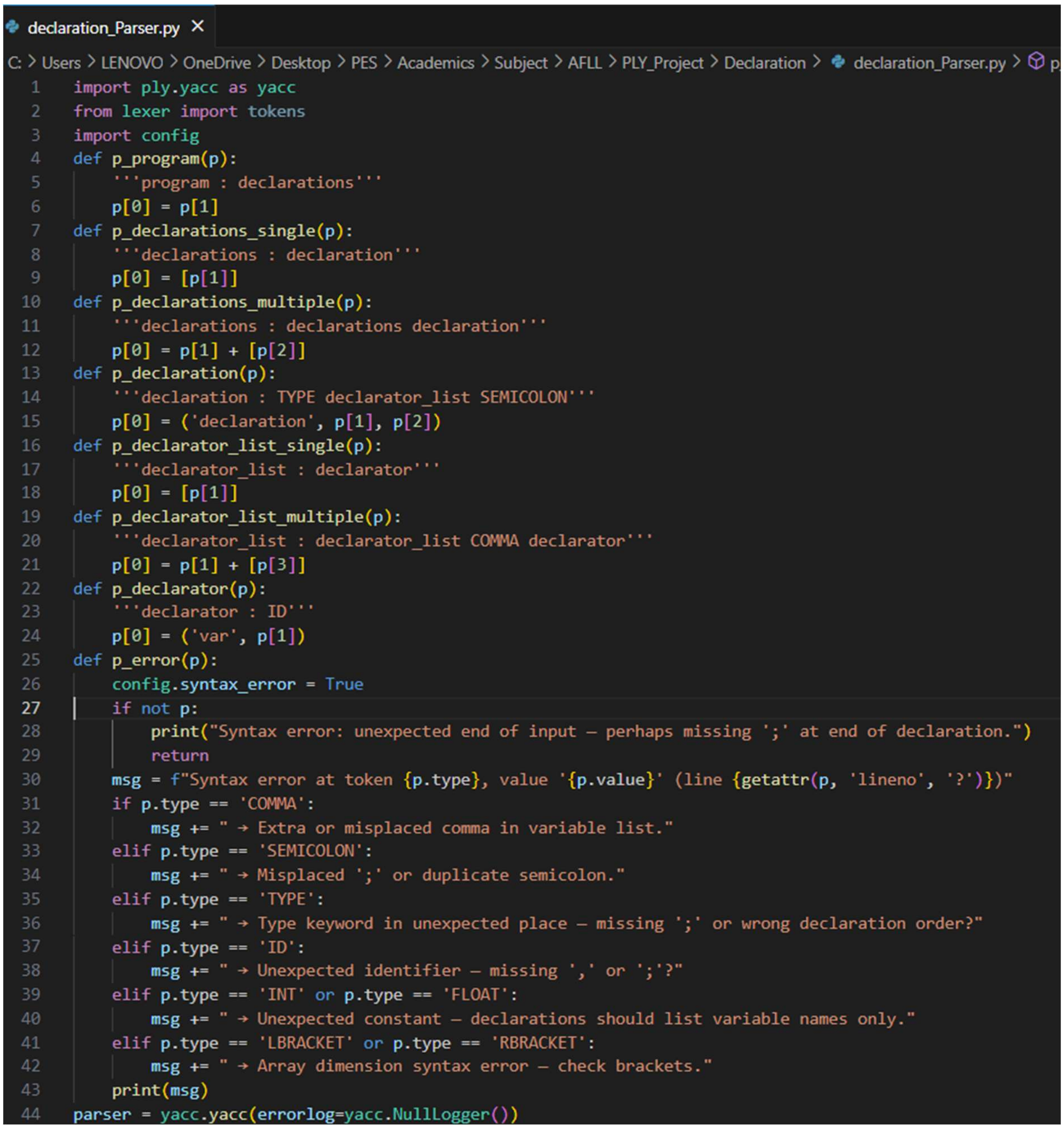
4)Output Verification



```
C:\Users\LENOVO\OneDrive\Desktop\PES\Academics\Subject\AFLL\PLY_Project\If_Else_if_Else>python main.py
--- C++ Language 5-Construct Parser ---
--- (If / Else / Else If Parser with Multi-line Input) ---
Type 'exit' to quit.
Enter your code (press Enter on a blank line to parse):
C++-Code > if(a)
...    > {
...    > int d;
...    > }
...    > else
...    > {
...    > }
...    >
Tokens: (IF, if) (LPAREN, () (ID, a) (RPAREN, )) (LBRACE, {) (TYPE, int) (ID, d) (SEMICOLON, ;) (RBRACE, }) (ELSE, else) (LBRACE, {) (RBRACE, })
--- Parsing Successful ---
C++-Code > if(a)
...    > {
...    > int q;
...    > else
...    > {
...    > int y;
...    > }
...    >
Tokens: (IF, if) (LPAREN, () (ID, a) (RPAREN, )) (LBRACE, {) (TYPE, int) (ID, q) (SEMICOLON, ;) (ELSE, else) (LBRACE, {) (TYPE, int) (ID, y) (SEMICOLON, ;)
(RBRACE, })
Syntax error at token ELSE, value 'else' → 'else' without matching 'if', or misplaced braces.
C++-Code > |
```

____

5)Simple Data-Type Declaration

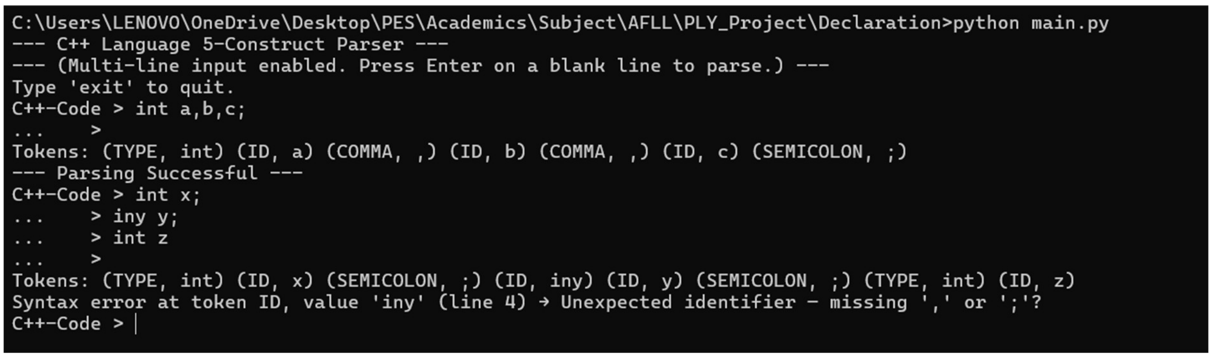


declaration_Parser.py ×

C: > Users > LENOVO > OneDrive > Desktop > PES > Academics > Subject > AFLL > PLY_Project > Declaration > declaration_Parser.py > p

```python
import ply.yacc as yacc
from lexer import tokens
import config
def p_program(p):
    '''program : declarations'''
    p[0] = p[1]
def p_declarations_single(p):
    '''declarations : declaration'''
    p[0] = [p[1]]
def p_declarations_multiple(p):
    '''declarations : declarations declaration'''
    p[0] = p[1] + [p[2]]
def p_declaration(p):
    '''declaration : TYPE declarator_list SEMICOLON'''
    p[0] = ('declaration', p[1], p[2])
def p_declarator_list_single(p):
    '''declarator_list : declarator'''
    p[0] = [p[1]]
def p_declarator_list_multiple(p):
    '''declarator_list : declarator_list COMMA declarator'''
    p[0] = p[1] + [p[3]]
def p_declarator(p):
    '''declarator : ID'''
    p[0] = ('var', p[1])
def p_error(p):
    config.syntax_error = True
    if not p:
        print("Syntax error: unexpected end of input — perhaps missing ';' at end of declaration.")
        return
    msg = f"Syntax error at token {p.type}, value '{p.value}' (line {getattr(p, 'lineno', '?')})"
    if p.type == 'COMMA':
        msg += " → Extra or misplaced comma in variable list."
    elif p.type == 'SEMICOLON':
        msg += " → Misplaced ';' or duplicate semicolon."
    elif p.type == 'TYPE':
        msg += " → Type keyword in unexpected place — missing ';' or wrong declaration order?"
    elif p.type == 'ID':
        msg += " → Unexpected identifier — missing ',' or ';'?"
    elif p.type == 'INT' or p.type == 'FLOAT':
        msg += " → Unexpected constant — declarations should list variable names only."
    elif p.type == 'LBRACKET' or p.type == 'RBRACKET':
        msg += " → Array dimension syntax error — check brackets."
    print(msg)
parser = yacc.yacc(errorlog=yacc.NullLogger())
```

5)Output Verification



```
C:\Users\LENOVO\OneDrive\Desktop\PES\Academics\Subject\AFLL\PLY_Project\Declaration>python main.py
--- C++ Language 5-Construct Parser ---
--- (Multi-line input enabled. Press Enter on a blank line to parse.) ---
Type 'exit' to quit.
C++-Code > int a,b,c;
...    >
Tokens: (TYPE, int) (ID, a) (COMMA, ,) (ID, b) (COMMA, ,) (ID, c) (SEMICOLON, ;)
--- Parsing Successful ---
C++-Code > int x;
...    > iny y;
...    > int z
...    >
Tokens: (TYPE, int) (ID, x) (SEMICOLON, ;) (ID, iny) (ID, y) (SEMICOLON, ;) (TYPE, int) (ID, z)
Syntax error at token ID, value 'iny' (line 4) → Unexpected identifier – missing ',' or ';'?
C++-Code > |
```

Conclusion

In this assignment, a modular parsing solution for C++ language constructs was successfully designed and implemented using the Python Lex-Yacc (PLY) library. The project was structured to validate each of the five chosen constructs—declarations, arrays, conditional statements, function declarations, and function definitions—independently, using a dedicated parser for each.

By creating a distinct parser.py for each construct, we were able to focus on its specific grammar and test it in isolation, ensuring correctness and extensibility. A single, shared lexer.py efficiently tokenized all C++ syntax elements, demonstrating code re-use. As shown in the output verifications, each specialized parser successfully validated its respective grammatical structure and handled syntax errors gracefully.