

A Parser

Report by: Rohan Venkatesha

Phase 1: Specification:

Language Specification: The self-designed language is called cminus and it has the following features:

- It is a case-sensitive language that uses ASCII characters.
- It supports only two data types: basic type int and a standard data type float.
- It supports arithmetic, logical, relational, and assignment operators.
- It supports if-else, while, and compound statements for control flow.
- It supports single-line and multi-line comments that start with /* and end with */
- It supports identifiers that start with a letter and can contain alphanumeric characters.
- It supports literals that are enclosed in single quotes for strings, and supports only decimal notation for floating point numbers.
- It supports keywords that are reserved for the language and cannot be used as identifiers. The keywords are: int, float, if, else, exit, while, read, write, and return.

Ply is a Python library that provides a set of tools to write lexical and syntactic analyzers. You will now add the parsing code to generate parse tree (abstract syntax tree - AST) of the given cminus code snippet.

You can use the following hints to write the parser rules:

- Use the `p_<name>` function syntax to define a rule. The `<name>` part should match the corresponding non-terminal in the grammar.
- Use the `p[0]`, `p[1]`, ..., `p[n]` syntax to access the symbols on the right-hand side of the rule. The `p[0]` symbol is the result of the rule.
- Use the `p.slice[n]` syntax to access the token object of the n-th symbol. The token object has attributes such as `type`, `value`, `lineno`, and `lexpos`.
- Use the `p_error` function to handle syntax errors. The function takes a single argument which is the token object where the error occurred. You can print an error message and skip the rest of the input.
- Use the `yacc` function to create the parser object. You can pass the lexer object and the start symbol as arguments.

Phase 2: Design:

1. lexer using the PLY (Lex – YACC) library in Python.

```
import ply.lex as lex
```

```
# List of token names
```

```
tokens =
```

```
['IDENTIFIER','INTCON','FLOATCON','STRING','PLUS','MINUS','TIMES','DIVIDE','ASSIGN','EQ','NE','LE','LT','GE','GT','AND','OR','NOT','LP','RP','LBK','RBK','LBR','RBR','SC','CM','IF','ELSE','EXIT','FLOAT','INT','READ','RETURN','WHILE','WRITE']
```

```
# Dictionary of keywords
```

```
keywords = {
```

```
'if': 'IF',  
'else': 'ELSE',  
'exit': 'EXIT',  
'float': 'FLOAT',  
'int': 'INT',  
'read': 'READ',  
'return': 'RETURN',  
'while': 'WHILE',  
'write': 'WRITE',  
}
```

Regular expression rules for simple tokens

```
t_PLUS = r'\+'  
t_MINUS = r'\-'  
t_TIMES = r'\*'  
t_DIVIDE = r'\/'  
t_ASSIGN = r'='  
t_EQ = r'=='  
t_NE = r'!='  
t_LE = r'<='  
t_LT = r'<'  
t_GE = r'>='  
t_GT = r'>'  
t_AND = r'&&'  
t_OR = r'\|\|'  
t_NOT = r'!'  
t_LP = r'\('  
t_RP = r'\)'  
t_LBK = r'\['  
t_RBK = r'\]'  
t_LBR = r'\{'  
t_RBR = r'\}'  
t_SC = r';'  
t_CM = r','
```

Regular expression rules with actions

def t_IDENTIFIER(t):

 r'[a-zA-Z_]\w*'

 # Check if the identifier is a keyword

 t.type = keywords.get(t.value, 'IDENTIFIER')

 return t

def t_FLOATCON(t):

 r'\d+\.\d+'

 t.value = float(t.value)

 return t

def t_INTCON(t):

 r'\d+'

 t.value = int(t.value)

 return t

def t_STRING(t):

 r\"'[^\']*'|\"[^\"]*\"'

 t.value = t.value[1:-1] # remove the quotes

 return t

Define a rule to track line numbers

def t_newline(t):

 r'\n+'

 t.lexer.lineno += len(t.value)

Define a rule to ignore whitespace and tabs

t_ignore = ' \t'

Define a rule to handle comments

def t_COMMENT(t):

 r'\\"[\\s\\S]*?\\\"|\\/\\\"[\\s\\S]*\$'

 pass

Error handling rule

def t_error(t):

 print(f"Lexical error: Unexpected character '{t.value[0]}')"

 t.lexer.skip(1)

Build the lexer

```
lexer = lex.lex()
```

2. Overview of Lexer:

Token Names: The tokens include basic elements like identifiers, constants (integer, float, string), arithmetic operators (+, -, *, /), comparison operators (==, !=, <=, <, >=, >), logical operators (&&, ||, !), parentheses, brackets, braces, semicolon, comma, and keywords for control flow and variable types.

Keywords: Dictionary called keywords to map reserved words in your language to their corresponding token types.

Regular Expressions for Simple Tokens: Regular expressions are used to define the patterns for simple tokens like operators, parentheses, brackets, braces, semicolon, and comma.

Regular Expressions with Actions: Regular expressions along with corresponding actions to convert the token's string representation into the appropriate Python type.

Line Numbers and Whitespace Handling: Rules to track line numbers and ignore whitespace.

Comments: Rule to handle comments, which are enclosed within /* ... */.

Error Handling: Error handling rule to print a message when an unexpected character is encountered.

Build the Lexer: Finally, you build the lexer using the lex.lex() function.

3. Build the parser using the PLY (Lex – YACC) library in Python :

Parser Rules

```
import ply.yacc as yacc

def p_program(p):
    """Program : DeclList Procedures
               | Procedures"""
def p_Procedures(p):
    """Procedures : ProcedureDecl Procedures
                  | ProcedureDecl"""
def p_ProcedureDecl(p):
    """ProcedureDecl : ProcedureHead ProcedureBody"""
def p_ProcedureHead(p):
    """ProcedureHead : FunctionDecl DeclList
                     | FunctionDecl"""
def p_FunctionDecl(p):
    """FunctionDecl : Type IDENTIFIER LP RP LBR"""
def p_ProcedureBody(p):
```

```

    ""ProcedureBody : StatementList RBR""
def p_DeclList(p):
    ""DeclList : Type IdentifierList SC
        | DeclList Type IdentifierList SC""
def p_IdentifierList(p):
    ""IdentifierList : VarDecl
        | IdentifierList CM VarDecl ""
def p_VarDecl(p):
    ""VarDecl : IDENTIFIER
        | IDENTIFIER LBK INTCON RBK""
def p_Type(p):
    ""Type : INT
        | FLOAT""
def p_StatementList(p):
    ""StatementList : Statement
        | StatementList Statement""
def p_Statement(p):
    ""Statement : Assignment
        | IfStatement
        | WhileStatement
        | IOStatement
        | ReturnStatement
        | ExitStatement
        | CompoundStatement""
def p_Assignment(p):
    ""Assignment : Variable ASSIGN Expr SC""
def p_IfStatement(p):
    ""IfStatement : IF Test CompoundStatement
        | IF Test CompoundStatement ELSE CompoundStatement""
def p_Test(p):
    ""Test : LP Expr RP""
def p_WhileStatement(p):
    ""WhileStatement : while_token WhileExpr Statement""

```

```

def p_while_token(p):
    """while_token : WHILE"""

def p_WhileExpr(p):
    """WhileExpr : LP Expr RP"""

def p_IOStatement(p):
    """IOStatement : READ LP Variable RP SC
        | WRITE LP Expr RP SC
        | WRITE LP StringConstant RP SC"""

def p_ReturnStatement(p):
    """ReturnStatement : RETURN Expr SC"""

def p_ExitStatement(p):
    """ExitStatement : EXIT SC"""

def p_CompoundStatement(p):
    """CompoundStatement : LBR StatementList RBR"""

def p_Expr(p):
    """Expr : Expr AND SimpleExpr
        | Expr OR SimpleExpr
        | SimpleExpr
        | NOT SimpleExpr"""

def p_SimpleExpr(p):
    """SimpleExpr : SimpleExpr EQ AddExpr
        | SimpleExpr NE AddExpr
        | SimpleExpr LE AddExpr
        | SimpleExpr LT AddExpr
        | SimpleExpr GE AddExpr
        | SimpleExpr GT AddExpr
        | AddExpr"""

def p_AddExpr(p):
    """AddExpr : AddExpr PLUS MulExpr
        | AddExpr MINUS MulExpr
        | MulExpr"""

def p_MulExpr(p):
    """MulExpr : MulExpr TIMES Factor

```

```

        | MulExpr DIVIDE Factor
        | Factor'''
def p_Factor(p):
    '''Factor : Variable
        | Constant
        | IDENTIFIER LP RP
        | LP Expr RP'''
def p_Variable(p):
    '''Variable : IDENTIFIER
        | IDENTIFIER LBK Expr RBK'''
def p_StringConstant(p):
    '''StringConstant : STRING'''
def p_Constant(p):
    '''Constant : INTCON
        | FLOATCON'''
# Error rule for syntax errors
def p_error(p):
    print(f'Syntax error at line {p.lineno}')
parser = yacc.yacc()

```

4. Overview of Parser Rules:

Program Rule: The top-level rule for the program, which can consist of a declaration list followed by procedures or just procedures.

Procedures Rule: Rules for defining procedures, which can be a single procedure or a list of procedures.

ProcedureDecl Rule: Combining the procedure head and body.

ProcedureHead Rule: Rules for defining the head of a procedure, including function declarations and declaration lists.

FunctionDecl Rule: Defining the type of the function.

ProcedureBody Rule: Defining the body of a procedure.

DeclList Rule: Rules for declaring variables, including their types and identifiers.

IdentifierList Rule: Lists of identifiers, possibly with variable declarations.

VarDecl Rule: Rules for declaring variables, including array declarations.

Type Rule: Rules for defining variable types, like INT or FLOAT.

StatementList Rule: Lists of statements.

Statement Rule: Rules for various types of statements, including assignments, if statements, while statements, IO statements, return statements, exit statements, and compound statements.

Assignment Rule: Rules for variable assignments.

IfStatement Rule: Rules for if statements, possibly with an else clause.

Test Rule: Rules for the test expression in an if statement.

WhileStatement Rule: Rules for while statements.

IOStatement Rule: Rules for input/output statements.

ReturnStatement Rule: Rules for return statements.

ExitStatement Rule: Rules for exit statements.

CompoundStatement Rule: Rules for compound statements enclosed in curly braces.

Expr, SimpleExpr, AddExpr, MulExpr, Factor Rules: Rules for expressions, including logical and arithmetic operations.

Variable Rule: Rules for variables, including array references.

StringConstant Rule: Rules for string constants.

Constant Rule: Rules for constants, including integers and floats.

Error Handling Rule: An error rule to handle syntax errors.

Build the Parser: Using yacc.yacc() to build the parser.

Test the Parser: Parsing input from a file.

5. Generate the Tree:

```
function print_tree(node, indent):
```

```
    if node is a list:
```

```
        for item in node:
```

```
            print_tree(item, indent)
```

```
    else if node is a dictionary:
```

```
        for key, value in node:
```

```
            print "|" * repeated indent times + key + ":"
```

```
            print_tree(value, indent + 1)
```

```
    else:
```

```
        print "|" * repeated indent times + ": " + node
```

6. Overview of Parse Tree:

print_tree: This function is used to print a tree-like structure representing the parsed code. It takes a node as input, which can be a list or a dictionary. If it's a list, it iterates over its items and prints them recursively. If it's a dictionary, it prints each key-value pair, and for each value, it calls itself recursively with increased indentation.

parse_source_code:

```
result = parser.parse(source_code, lexer=lexer)
```

parser: The parser object you've built using the `yacc.yacc()` function. This object is responsible for parsing the input source code based on the grammar rules you've defined.

source_code: The source code of your programming language that you want to parse.

lexer: The lexer object responsible for tokenizing the source code. It breaks down the source code into tokens, which the parser then uses to understand the structure of the code.

result: This variable holds the result of the parsing operation. It could be an abstract syntax tree (AST) or any other representation of the parsed code, depending on how your parser is designed.

Phase 3: Testing and Output:

Input 1:

```
int main() {  
    int x, y;  
    read(x); read(y);  
    while ((x != 0) || (y != 0)) {  
        write (x*y);  
        read (x); read (y);  
    }  
    exit;  
}
```

Output 1:

Lexical Output

```
[('INT', 'int'), ('IDENTIFIER', 'main'), ('LP', '('), ('RP', ')'), ('LBR', '{'), ('INT', 'int'), ('IDENTIFIER', 'x'), ('CM', ','),  
('IDENTIFIER', 'y'), ('SC', ';'), ('READ', 'read'), ('LP', '('), ('IDENTIFIER', 'x'), ('RP', ')'), ('SC', ';'), ('READ', 'read'), ('LP', '('),  
('IDENTIFIER', 'y'), ('RP', ')'), ('SC', ';'), ('WHILE', 'while'), ('LP', '('), ('LP', '('), ('IDENTIFIER', 'x'), ('NE', '!='), ('INTCON',  
0), ('RP', ')'), ('OR', '||'), ('LP', '('), ('IDENTIFIER', 'y'), ('NE', '!='), ('INTCON', 0), ('RP', ')'), ('RP', ')'), ('LBR', '{'), ('WRITE',
```

```
'write'), ('LP', '('), ('IDENTIFIER', 'x'), ('TIMES', '*'), ('IDENTIFIER', 'y'), ('RP', ')'), ('SC', ';'), ('READ', 'read'), ('LP', '('), ('IDENTIFIER', 'x'), ('RP', ')'), ('SC', ';'), ('READ', 'read'), ('LP', '('), ('IDENTIFIER', 'y'), ('RP', ')'), ('SC', ';'), ('RBR', '}'), ('EXIT', 'exit'), ('SC', ';'), ('RBR', '}')]
```

YACC output

```
{'Program': {'Procedures': {'ProcedureDecl': {'ProcedureHead': {'FunctionDecl': {'Type': 'int', 'Name': 'main', 'LP': '(', 'RP': ')', 'LBR': '{', 'DeclList': {'Type': 'int', 'IdentifierList': {'Variable Declaration': 'x'}, 'Variable Declaration': 'y'}, 'Semicolon': ';'}, 'ProcedureBody': {'StatementList': {'StatementList': {'StatementList': {'StatementList': {'Statement': {'IOStatement': {'type': 'read_statement', 'LP': '(', 'variable': {'Variable': 'x'}, 'RP': ')', 'Semicolon': ';'}}}, 'Statement': {'IOStatement': {'type': 'read_statement', 'LP': '(', 'variable': {'Variable': 'y'}, 'RP': ')', 'Semicolon': ';'}}}, 'Statement': {'WhileStatement': {'While Expression': {'WhileExpr': {'Expr': {'operator': '| |', 'left': {'Expr': {'SimpleExpr': {'AddExpr': {'MulExpr': {'Factor': {'Expr': {'SimpleExpr': {'operator': '!=', 'left': {'SimpleExpr': {'AddExpr': {'MulExpr': {'Factor': {'Variable': 'x'}}}}}, 'right': {'AddExpr': {'MulExpr': {'Factor': {'Constant': 0}}}}}}}}}, 'right': {'SimpleExpr': {'AddExpr': {'MulExpr': {'Factor': {'Expr': {'SimpleExpr': {'operator': '!=', 'left': {'SimpleExpr': {'AddExpr': {'MulExpr': {'Factor': {'Variable': 'y'}}}}}, 'right': {'AddExpr': {'MulExpr': {'Factor': {'Constant': 0}}}}}}}}}}}, 'Statement': {'Statement': {'CompoundStatement': {'StatementList': {'StatementList': {'StatementList': {'Statement': {'IOStatement': {'type': 'write_statement', 'LP': '(', 'Expression': {'Expr': {'SimpleExpr': {'AddExpr': {'MulExpr': {'operator': '*', 'left': {'MulExpr': {'Factor': {'Variable': 'x'}}}, 'right': {'Factor': {'Variable': 'y'}}}}}}}, 'RP': ')', 'Semicolon': ';'}}}, 'Statement': {'IOStatement': {'type': 'read_statement', 'LP': '(', 'variable': {'Variable': 'x'}, 'RP': ')', 'Semicolon': ';'}}}, 'Statement': {'IOStatement': {'type': 'read_statement', 'LP': '(', 'variable': {'Variable': 'y'}, 'RP': ')', 'Semicolon': ';'}}}}}}}, 'Statement': {'ExitStatement': 'exit'}}}, 'Right Braces': '}}}}}
```

Abstract Syntax Tree

Program:

| Procedures:

| | ProcedureDecl:

| | | ProcedureHead:

```

| | | | FunctionDecl:
| | | | | Type:
| | | | | : int
| | | | | Name:
| | | | | : main
| | | | | LP:
| | | | | : (
| | | | | RP:
| | | | | : )
| | | | | LBR:
| | | | | : {
| | | | DeclList:
| | | | | Type:
| | | | | : int
| | | | | IdentifierList:
| | | | | Variable Declaration:
| | | | | : x
| | | | | Variable Declaration:
| | | | | : y
| | | | Semicolon:
| | | | : ;
| | | ProcedureBody:
| | | | StatementList:
| | | | | StatementList:
| | | | | | StatementList:
| | | | | | | StatementList:
| | | | | | | | Statement:
| | | | | | | | | IOStatement:
| | | | | | | | | type:
| | | | | | | | | : read_statement
| | | | | | | | | LP:
| | | | | | | | | : (
| | | | | | | | | variable:

```

| | | | | | | | | Variable:
| | | | | | | | | : x
| | | | | | | | | RP:
| | | | | | | | | :)
| | | | | | | | | Semicolon:
| | | | | | | | | : ;
| | | | | | | | | Statement:
| | | | | | | | | IOStatement:
| | | | | | | | | type:
| | | | | | | | | : read_statement
| | | | | | | | | LP:
| | | | | | | | | : (
| | | | | | | | | variable:
| | | | | | | | | Variable:
| | | | | | | | | : y
| | | | | | | | | RP:
| | | | | | | | | :)
| | | | | | | | | Semicolon:
| | | | | | | | | : ;
| | | | | | | | | Statement:
| | | | | | | | | WhileStatement:
| | | | | | | | | While Expression:
| | | | | | | | | WhileExpr:
| | | | | | | | | Expr:
| | | | | | | | | operator:
| | | | | | | | | : ||
| | | | | | | | | left:
| | | | | | | | | Expr:
| | | | | | | | | SimpleExpr:
| | | | | | | | | AddExpr:
| | | | | | | | | MulExpr:
| | | | | | | | | Factor:
| | | | | | | | | Expr:

| | | | | | | | | | | | | | | | | SimpleExpr:
| | | | | | | | | | | | | | | | | operator:
| | | | | | | | | | | | | | | | | : !=
| | | | | | | | | | | | | | | | | left:
| | | | | | | | | | | | | | | | | SimpleExpr:
| | | | | | | | | | | | | | | | | AddExpr:
| | | | | | | | | | | | | | | | | MulExpr:
| | | | | | | | | | | | | | | | | Factor:
| | | | | | | | | | | | | | | | | Variable:
| | | | | | | | | | | | | | | | | : x
| | | | | | | | | | | | | | | | | right:
| | | | | | | | | | | | | | | | | AddExpr:
| | | | | | | | | | | | | | | | | MulExpr:
| | | | | | | | | | | | | | | | | Factor:
| | | | | | | | | | | | | | | | | Constant:
| | | | | | | | | | | | | | | | | : 0
| | | | | | | | | | | | | | | | | right:
| | | | | | | | | | | | | | | | | SimpleExpr:
| | | | | | | | | | | | | | | | | AddExpr:
| | | | | | | | | | | | | | | | | MulExpr:
| | | | | | | | | | | | | | | | | Factor:
| | | | | | | | | | | | | | | | | Expr:
| | | | | | | | | | | | | | | | | SimpleExpr:
| | | | | | | | | | | | | | | | | operator:
| | | | | | | | | | | | | | | | | : !=
| | | | | | | | | | | | | | | | | left:
| | | | | | | | | | | | | | | | | SimpleExpr:
| | | | | | | | | | | | | | | | | AddExpr:
| | | | | | | | | | | | | | | | | MulExpr:
| | | | | | | | | | | | | | | | | Factor:
| | | | | | | | | | | | | | | | | Variable:
| | | | | | | | | | | | | | | | | : y
| | | | | | | | | | | | | | | | | right:

```
| | | | | | | | | | | | | | | | AddExpr:  
| | | | | | | | | | | | | | | | MulExpr:  
| | | | | | | | | | | | | | | | Factor:  
| | | | | | | | | | | | | | | | Constant:  
| | | | | | | | | | | | | | | | : 0  
  
| | | | | | | | Statement:  
| | | | | | | | Statement:  
| | | | | | | | CompoundStatement:  
| | | | | | | | StatementList:  
| | | | | | | | StatementList:  
| | | | | | | | StatementList:  
| | | | | | | | Statement:  
| | | | | | | | IOStatement:  
| | | | | | | | type:  
| | | | | | | | : write_statement  
| | | | | | | | LP:  
| | | | | | | | : (  
| | | | | | | | Expression:  
| | | | | | | | Expr:  
| | | | | | | | SimpleExpr:  
| | | | | | | | AddExpr:  
| | | | | | | | MulExpr:  
| | | | | | | | operator:  
| | | | | | | | : *  
| | | | | | | | left:  
| | | | | | | | MulExpr:  
| | | | | | | | Factor:  
| | | | | | | | Variable:  
| | | | | | | | : x  
| | | | | | | | right:  
| | | | | | | | Factor:  
| | | | | | | | Variable:  
| | | | | | | | : y
```

| | | | | | | | | | | | | | | | | | RP:
| | | | | | | | | | | | | | | | | :)
| | | | | | | | | | | | | | | | | Semicolon:
| | | | | | | | | | | | | | | | | : ;
| | | | | | | | | | | | | | | | | Statement:
| | | | | | | | | | | | | | | | | IOStatement:
| | | | | | | | | | | | | | | | | type:
| | | | | | | | | | | | | | | | | : read_statement
| | | | | | | | | | | | | | | | | LP:
| | | | | | | | | | | | | | | | | : (
| | | | | | | | | | | | | | | | | variable:
| | | | | | | | | | | | | | | | | Variable:
| | | | | | | | | | | | | | | | | : x
| | | | | | | | | | | | | | | | | RP:
| | | | | | | | | | | | | | | | | :)
| | | | | | | | | | | | | | | | | Semicolon:
| | | | | | | | | | | | | | | | | : ;
| | | | | | | | | | | | | | | | | Statement:
| | | | | | | | | | | | | | | | | IOStatement:
| | | | | | | | | | | | | | | | | type:
| | | | | | | | | | | | | | | | | : read_statement
| | | | | | | | | | | | | | | | | LP:
| | | | | | | | | | | | | | | | | : (
| | | | | | | | | | | | | | | | | variable:
| | | | | | | | | | | | | | | | | Variable:
| | | | | | | | | | | | | | | | | : y
| | | | | | | | | | | | | | | | | RP:
| | | | | | | | | | | | | | | | | :)
| | | | | | | | | | | | | | | | | Semicolon:
| | | | | | | | | | | | | | | | | : ;
| | | | | Statement:
| | | | | ExitStatement:
| | | | | : exit

$$| \quad | \quad | \quad | \quad : \quad \}$$
[illegible]

```

{
  "ProcedureBody": {
    "StatementList": {
      "StatementList": {
        "Statement": {
          "IDStatement": {
            "type": "read_statement",
            "lp": {
              "variable": {
                "Variable": {
                  "name": "x"
                },
                "RP": {}
              },
              "Semicolon": {}
            },
            "Statement": {
              "IDStatement": {
                "type": "read_statement",
                "lp": {
                  "variable": {
                "Variable": {
                  "name": "y"
                },
                "RP": {}
              },
              "Semicolon": {}
            },
            "Statement": {
              "WhileStatement": {
                "whileExpression": {
                  "Expr": {
                    "operator": ":",
                    "left": {
                      "Expr": {
                        "SimpleExpr": {
                          "AddExpr": {
                            "AddExpr": {
                              "Factor": {
                                "Expr": {
                                  "SimpleExpr": {
                                    "operator": "+",
                                    "left": {
                                      "SimpleExpr": {
                                        "AddExpr": {
                                          "AddExpr": {
                                            "Factor": {
                                              "Variable": {
                                                "name": "x"
                                              }
                                            }

```



```
[('INT', 'int'), ('IDENTIFIER', 'main'), ('LP', '('), ('RP', ')'), ('LBR', '{'), ('INT', 'int'), ('IDENTIFIER', 'x'), ('CM', ','), ('IDENTIFIER', 'y'), ('SC', ';'), ('EXIT', 'exit'), ('SC', ';'), ('RBR', '}')]
```

YACC output

```
{'Program': {'Procedures': {'ProcedureDecl': {'ProcedureHead': {'FunctionDecl': {'Type': 'int', 'Name': 'main', 'LP': '(', 'RP': ')', 'LBR': '{', 'DeclList': {'Type': 'int', 'IdentifierList': {'Variable Declaration': 'x'}, 'Variable Declaration': 'y'}, 'Semicolon': ';'}, 'ProcedureBody': {'StatementList': {'Statement': {'ExitStatement': 'exit'}}}, 'Right Braces': '}}}}}}
```

Abstract Syntax Tree

Program:

| Procedures:

| | ProcedureDecl:

| | | ProcedureHead:

| | | | FunctionDecl:

| | | | | Type:

| | | | | : int

| | | | | Name:

| | | | | : main

| | | | | LP:

| | | | | : (

| | | | | RP:

| | | | | :)

| | | | | LBR:

| | | | | : {

| | | | DeclList:

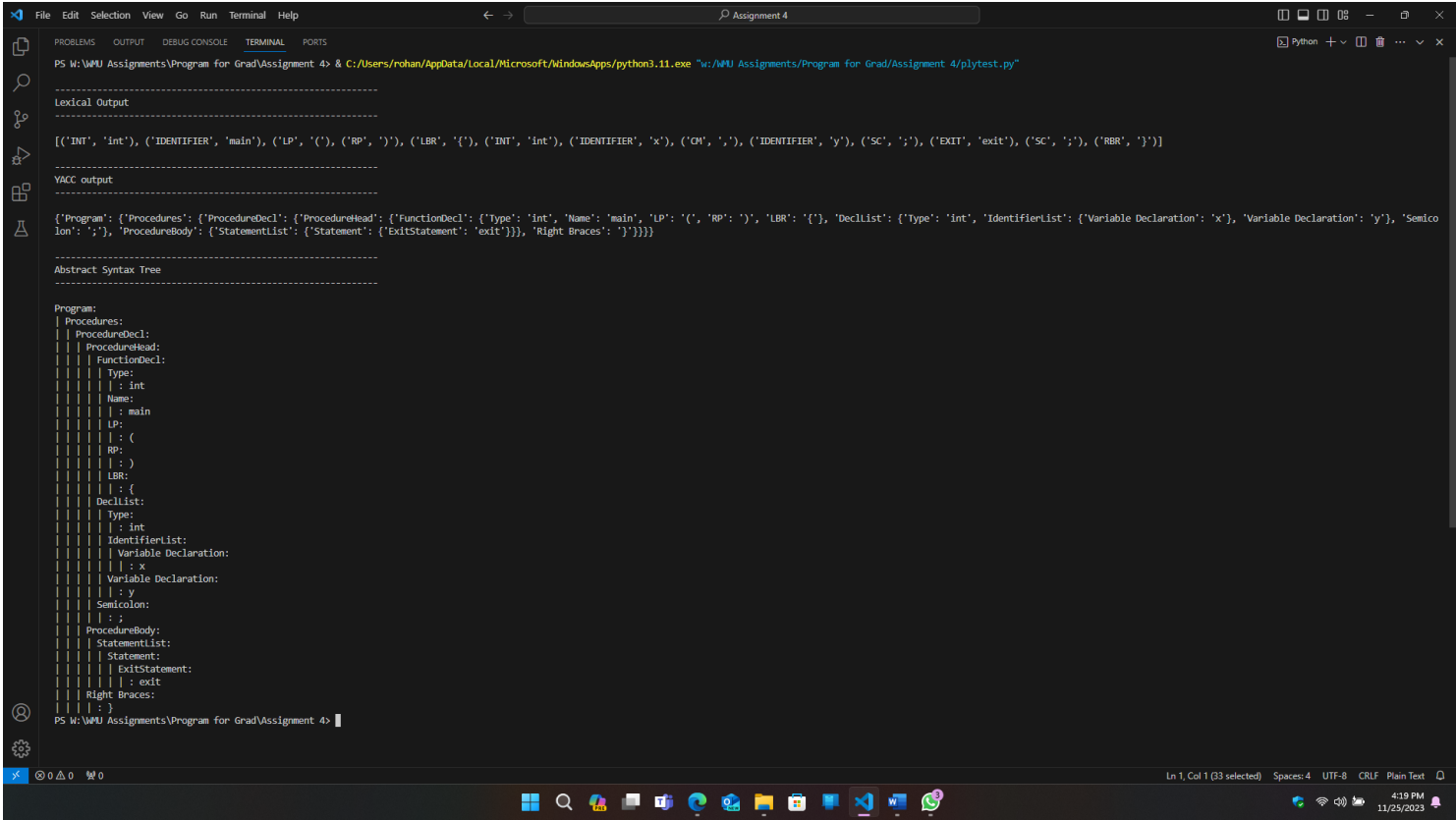
| | | | | Type:

```

| | | | : int
| | | | IdentifierList:
| | | | Variable Declaration:
| | | | : x
| | | | Variable Declaration:
| | | | : y
| | | Semicolon:
| | | :;
| | ProcedureBody:
| | | StatementList:
| | | Statement:
| | | | ExitStatement:
| | | | : exit
| | Right Braces:
| | | :}

```

Screenshot:



Input 3: (Testing Error Conditions)

```
int main() {  
int x, y;  
read(x);  
read(y)  
exit;  
}
```

Output 3:

Lexical Output

[('INT', 'int'), ('IDENTIFIER', 'main'), ('LP', '(', ('RP', ')), ('LBR', '{'), ('INT', 'int'), ('IDENTIFIER', 'x'), ('CM', ','), ('IDENTIFIER', 'y'), ('SC', ';'), ('READ', 'read'), ('LP', '(', ('IDENTIFIER', 'x'), ('RP', ')), ('SC', ';'), ('READ', 'read'), ('LP', '(', ('IDENTIFIER', 'y'), ('RP', ')), ('EXIT', 'exit'), ('SC', ';'), ('RBR', '}')]

Syntax error at line 10: Unexpected token 'exit'

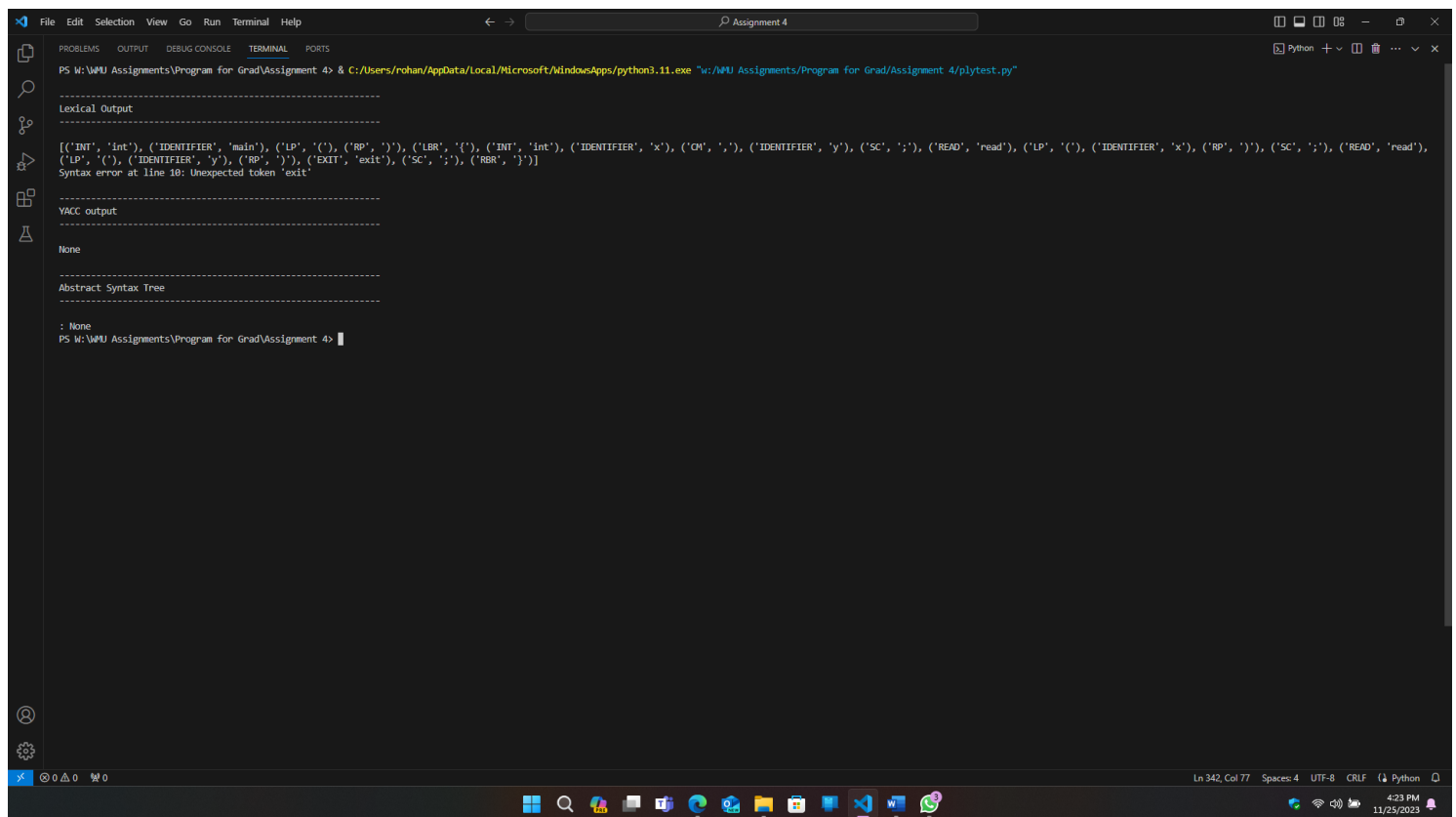
YACC output

None

Abstract Syntax Tree

: None

Screenshot:



```
PS M:\WMU Assignments\Program for Grad\Assignment 4> & C:\Users\rohan\AppData\Local\Microsoft\WindowsApps\python3.11.exe "M:\WMU Assignments\Program for Grad\Assignment 4\plytest.py"

Lexical Output
-----

[('INT', 'int'), ('IDENTIFIER', 'main'), ('LP', '('), ('RP', ')'), ('LBR', '{'), ('INT', 'int'), ('IDENTIFIER', 'x'), ('OP', ','), ('IDENTIFIER', 'y'), ('SC', ';'), ('READ', 'read'), ('LP', '('), ('IDENTIFIER', 'x'), ('RP', ')'), ('SC', ';'), ('READ', 'read'), ('LP', '('), ('IDENTIFIER', 'y'), ('RP', ')'), ('EXIT', 'exit'), ('SC', ';'), ('RBR', '}')]
Syntax error at line 10: Unexpected token 'exit'

YACC output
-----

None

Abstract Syntax Tree
-----

: None
PS M:\WMU Assignments\Program for Grad\Assignment 4> |
```

References:

- ChatGPT, prompt, November 25, 2023, OpenAI, <https://chat.openai.com>.
- PLY (Lex-Yacc) <https://github.com/dabeaz/ply>
- Video Guide - PA3 Parser – Python, <https://youtu.be/kearNtiYWr8?feature=shared>
- <https://stackoverflow.com/questions/77499081/how-to-properly-implement-an-abstract-syntax-tree-in-ply>