
# Department of Computer Science and Engineering

## University of Asia Pacific

### Compiler Project Report

Course Title: Compiler Design Lab

Course Code: CSE 430

Semester: 4.2

Submitted by:

1. Name: Md. Abdur Rashid , ID: 19101008
2. Name: Nor Mohammad Nasim , ID: 19101010
3. Name: Tahamid Khan , ID: 19101016

Submitted to,

Baivab Das,

Lecturer,

Department of Computer Science and Engineering,

University of Asia Pacific.

## 1. Design:

The offered code was created using the PLY (Python Lex-Yacc) library and is developed in the Python programming language to implement a straightforward calculator. Let's dissect the overall structure and outline the code modules:

**1.Language: Python:**

The code is written in Python, which is a high-level, interpreted programming language known for its simplicity and readability.

**2.IDE: PyCharm:**

- PyCharm is an integrated development environment (IDE) for Python programming.
- While the specific IDE used to develop the code is not mentioned in the code itself, PyCharm is a popular choice for Python development.

**3.Modules:**

The code consists of the following modules:

- re: This module provides support for regular expressions. It is used to define the regular expressions for the lexer tokens.
- ply.lex: This module is part of the PLY library and provides the functionality for lexical analysis or tokenization.
- ply.yacc: This module is also part of the PLY library and handles the parsing or syntactic analysis using the defined grammar rules.

**4.Lexer:**

The lexer module defines the tokens and their corresponding regular expressions using the tokens list and the t_TOKEN functions. The t_error function handles any syntax errors encountered during tokenization.

**5.Parser:**

The parser module defines the grammar rules for the calculator using the p_RULE functions. Each rule specifies how to handle different expressions and operators. The p_error function handles syntax errors during parsing.

**6.Main Execution:**

The main execution part prompts the user to input an expression to evaluate. It then uses the lexer and parser to tokenize and parse the input expression. The result is printed, and the process continues until an EOFError is encountered.

Overall, the code uses the PLY library to implement a lexer and parser for a simple calculator. The lexer tokenizes the input expression, and the parser applies grammar rules to evaluate the expression and perform the corresponding calculations.

## 2. Implementation:

In this part here we explain full code of our project:

Part 1.

```
# Import the necessary modules
import re

import ply.lex as lex

import ply.yacc as yacc
```

Explanation: The lexer and parser for a basic arithmetic expression evaluator are implemented in our project code. For lexical analysis and parsing, it makes use of PLY (Python Lex-Yacc), a Python implementation of the well-known Lex and Yacc tools.

Part 2.

```
# Define the tokens for the lexer
tokens = [
   'NUMBER', # Integer or decimal number
   'PLUS',   # Addition operator
   'MINUS',  # Subtraction operator
   'TIMES',   # Multiplication operator
   'DIVIDE',  # Division operator
   'LPAREN',  # Left parenthesis
   'RPAREN',   # Right parenthesis
]

# Define the regular expressions for each token
t_PLUS    = r'\+'  # Plus sign
t_MINUS   = r'-'   # Minus sign
t_TIMES   = r'\*'   # Asterisk
t_DIVIDE  = r'/'    # Forward slash
t_LPAREN  = r'\('   # Left parenthesis
t_RPAREN  = r'\)'   # Right parenthesis
t_NUMBER  = r'\d+'   # Match one or more digits
```

Explanation: In part 2, the code first specifies the tokens that the lexer will understand, such as parentheses (LPAREN and RPAREN), numbers (NUMBER), and operators like PLUS, MINUS, TIMES, and DIVIDE. Using the t_X functions, it also specifies regular expressions for every token.

Part 3.

```
# Define how to handle whitespace
t_ignore = ' \t'  # Ignore spaces and tabs

# Define what to do when a syntax error is encountered
def t_error(t):
    print(f"Invalid character '{t.value[0]}'")
    t.lexer.skip(1)  # Skip the erroneous character

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

Explanation: In part 3, we ignore the white space and handle invalid character. Here, Lex.lex() is then used to create the lexer, creating an object that can be used to tokenize input strings.

Part 4.

```
def p_expression_plus(p):
    'expression : expression PLUS term'
# Match expression + term
    p[0] = p[1] + p[3]  # Compute the result
    print(f"{p[1]} + {p[3]} = {p[0]}")  # Print the result
```

Explanation:

This function defines that, the grammar rule for the addition of two expressions is defined by the function p_expression_plus(p). It computes the outcome by adding the values of the two expressions and checks for the pattern "expression + term". It records the outcome of this grammar rule in p[0], where it is stored. It prints the computation result before concluding.

Part 5.

```
def p_expression_minus(p):
    'expression : expression MINUS term'
# Match expression - term
    p[0] = p[1] - p[3]  # Compute the result
    print(f"{p[1]} - {p[3]} = {p[0]}") # Print the result
```

Explanation:

This function defines that , The grammar rule for the subtraction of two expressions is defined by the function p_expression_minus(p). It computes the outcome by matching the "expression - term" pattern.

Part 6.

```
def p_expression_term(p):
    'expression : term'      # Match a term by itself
    p[0] = p[1]              # Set the result to the term
```

Explanation:

This function defines that, the grammar rule for an expression that consists only of a term. By itself, it matches a term's pattern and sets the result to the value of that term.

Part 7.

```
def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]
    print(f"{p[1]} * {p[3]} = {p[0]}")
```

Explanation:

The grammar rule for multiplying two terms is defined by this function. It multiplies the values of the two terms to calculate the outcome and matches the pattern of "term * factor". This grammar rule's output, which is stored in p[0], is the result. The computation result is then printed.

Part 8.

```
def p_term_divide(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]
    print(f"{p[1]} / {p[3]} = {p[0]}")
```

Explanation:

This function defines that, specifies the grammar rule for dividing two terms. It computes the outcome by dividing the term's value by the factor's value and checks to see if the pattern of "term / factor" is present. It records the outcome of this grammar rule in p[0], where it is stored. It prints the computation result before concluding.

Part 9.

```
def p_term_factor(p):
    'term : factor'
    p[0] = p[1]
```

Explanation:

The function p_term_factor(p) specifies the grammar rule for a term that is only a factor. It sets the outcome to the value of that factor after matching the pattern of a single factor.

Part 10.

```
def p_factor_num(p):
    'factor : NUMBER'
    p[0] = int(p[1])
    print(f"Number: {p[0]}")
```

Explanation:

This function, p_factor_num(p), specifies the grammar rule for a factor that is a number. When a number's pattern is matched, the result is set to that number's value.

Part 11.

```
def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]
```

Explanation:

With the help of the function p_factor_expr(p), we can determine the grammar requirements for factors that are parenthetical expressions. It sets the outcome to the value of that expression by matching the pattern of "(expression)".

Part 12.

```
# Define what to do when a syntax error is encountered
def p_error(p):
    print("Syntax error in input!")
```

Explanation:

When a syntax error is encountered by the parser, p_error(p) is called. There is a syntax error in the input, and it prints an error message indicating this.

Part 13.

```
# Create the parser using YACC
parser = yacc.yacc()
```

Explanation:

YACC parser objects are created by the yacc.yacc() function. YACC (Yet Another Compiler Compiler) is a parser generator that creates a parser for a language from a formal description of its syntax.

Part 14.

```
while True:
    try:
        s = input('enter number: ')
    except EOFError:
        break

    # Parse and evaluate the expression
    print("Parsing:")
```

```
result = parser.parse(s)
print(f"Result: {result}\n")
```

Explanation:

While True: This initiates an endless loop that won't stop unless specifically broken.

try: This is the first statement in a try block, which is used to handle any exceptions that might arise from user-supplied expressions.

When the user is prompted to enter a number, the command s = input('enter number:') assigns the user's expression to the variable s.

except EOFError: If the user inputs the end of file (EOF) signal, this exception is caught.

Break: The while loop is broken here.

The string "Parsing:" is printed to the console using print("Parsing:").

result = parser.parse(s): This parses the user-inputted expression using the previously created YACC parser object and assigns the outcome to the variable result.

Printing "Result:  resultn"  This formatted string prints the parsed expression's outcome to the console. After the result is printed, a new line is added using the "n" character.

## 3. Result:

Input: (8*9)+(5*6)+24 and output will be = 126

**Output explain:**

The application evaluates arithmetic expressions entered by the user and functions as a straightforward calculator.

The user enters the expression "(89)+(56)+24" for this particular run. The lexer tokenizes this input, picking up on the symbols + and * as well as the digits 8, 9, 5, 6, and 24. The parser then employs these tokens to create a parse tree that accurately depicts the expression's structure.

The sub-expression "89" is the first factor that the parser recognizes in the expression. "8 * 9 = 72" is printed after this subexpression has been evaluated. Next, it advances to the "56" sub-expression in the parse tree. It also evaluates this subexpression, printing "5 * 6 = 30" as a result. Finally, the top-level expression "(89)+(56)+24" is recognized by the parser. When evaluating this expression, it prints "72 + 30 = 102" and "102 + 24 = 126" after adding the values of the two subexpressions and the number 24.

The expression's final result, 126, is printed.

## 4. Reference

1. Parser generator

https://planetcalc.com/5600/

2. Mathematical Expression Parser

https://itnext.io/writing-a-mathematical-expression-parser-35b0b78f869e

3. YACC program to implement a Calculator

https://www.geeksforgeeks.org/yacc-program-to-implement-a-calculator-and-recognize-a-valid-arithmetic-expression/

4. text parsers with yacc and lex

https://developer.ibm.com/tutorials/au-lexyacc/

5. Parsing an arithmetic expression and building a tree from it

https://stackoverflow.com/questions/4589951/parsing-an-arithmetic-expression-and-building-a-tree-from-it-in-java