Text

Description automatically generated

**Activity based**

**Project Report on**

**System programming**

**Submitted to Vishwakarma University, Pune**

**Under the Initiative of**

**Contemporary Curriculum, Pedagogy, and Practice (C2P2)**

**By**

**Merwin Pinto**

**SRN No : 202100102**

**Roll No : 01**

**Diya Oswal**

**SRN No : 202101718**

**Roll No : 40**

**Div : E**

**Third Year Engineering**

**Department of Computer Engineering**

**Faculty of Science and Technology**

**Academic Year**

**2023-2024**

Project Statement :

Implementing an Expression calculator based Interpreter in Python and Demonstrating its

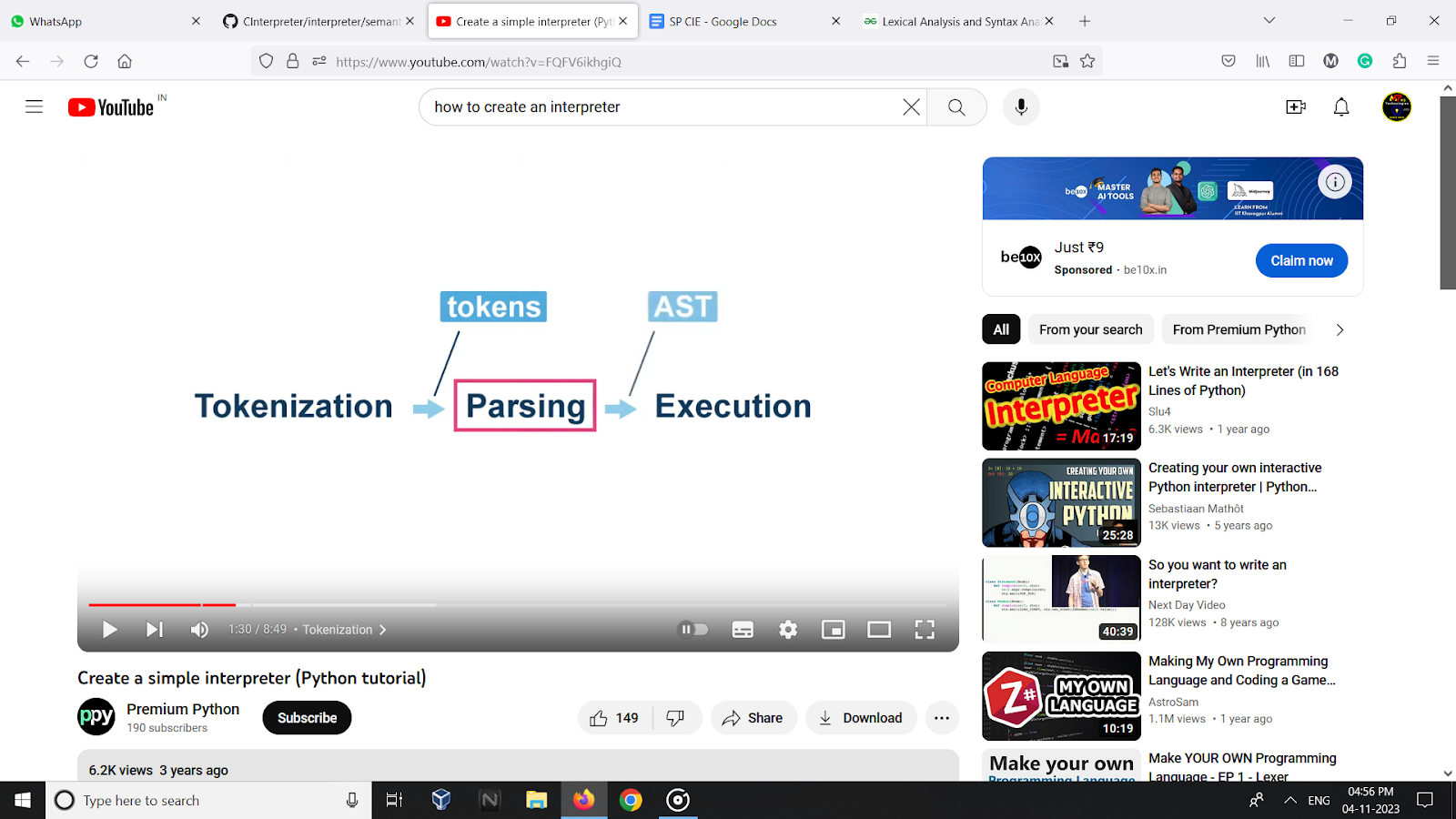
3 Phases: - Lexical , Parser and Execution.

**Project Statement :**

**Project Description :**

An Expression interpreter is a computer program that directly executes instructions written in a matematical Expression . Our main objective is to Demonstrate the Working of our Interpreter using the 3 Phases with Explanation

**Interpreter is further divided into 3 parts**

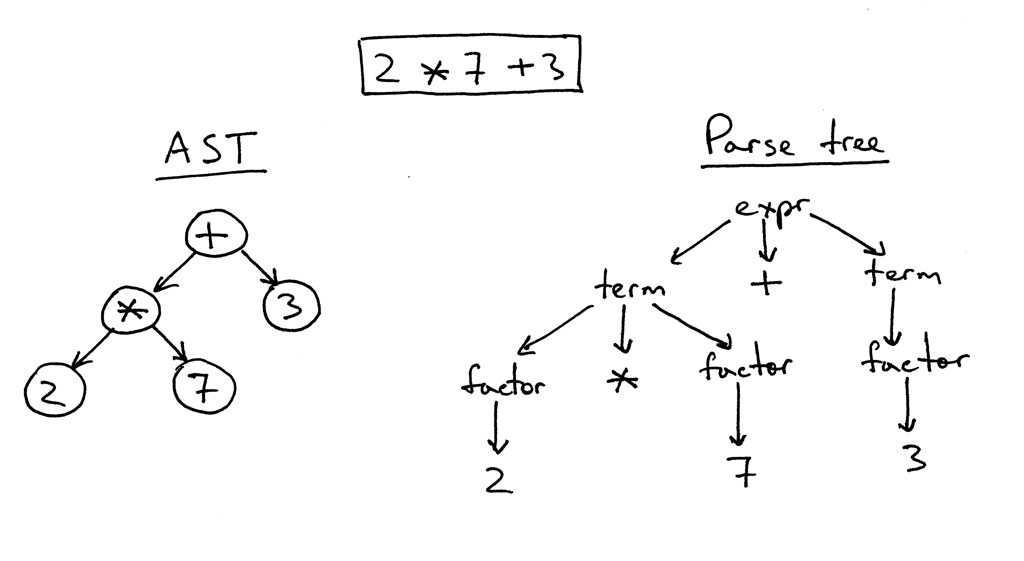
1. Lexical Analysis ( tokens )
2. Parser ( Syntax tree )
3. Execution( executes the program )

**PROJECT MODULE 2 : SYNTAX ANALYSER /PARSE TREE**

**Parser / Syntax Analysis** A parser is a program that analyzes text or code to determine if it is valid according to a set of rules, such as the grammar of a programming language or the syntax of a data format.

the parser would first need to identify the different types of tokens, such as keywords, identifiers, operators, and punctuation marks. Once the parser has identified the types of tokens, it can then start to build the AST.

**Example of Parser tree**



**Implementation :**

**Implementation procedure**

**Initialize data structures:**These stacks will be used to keep track of the order of operations and operands during parsing.

**Define operator precedence:** Create a dictionary precedence to store the precedence levels of different operators.

**Check for higher precedence:** Implement a function is\_higher\_precedence() that takes two operators as input and returns True if the first operator has higher precedence than the second, and False otherwise.

**Create binary nodes:** Define a function create\_binary\_node() that takes the op\_stack and value\_stack as input and pops the top two operands and the top operator from their respective stacks. It then creates a new binary node with the operator as its type and the two operands as its left and right children, respectively. Then Iterating over the list of tokens. For each token, check its type and take the appropriate action

**Numbers and variables:** If the token type is NUMBER or VARIABLE, push a new node with the token type and value onto the value\_stack.

**Operators:** If the token type is OPERATOR, check the op\_stack for higher-precedence operators. If there are any, pop them from the op\_stack and the value\_stack and create binary nodes until the op\_stack is empty or the next operator has lower precedence. Then, push the current operator onto the op\_stack.

**Parentheses and brackets:** If the token type is PARENTHESIS or BRACKET, check whether it is an opening or closing parenthesis/bracket. If it is opening, push it onto the op\_stack. If it is closing, pop operators from the op\_stack until you find the corresponding opening parenthesis/bracket.

**Build the parse tree:** After processing all tokens, pop remaining operators from the op\_stack and create binary nodes, pushing the resulting nodes onto the value\_stack. The remaining node on the value\_stack is the root of the parse tree.

**Print the parse tree:** Implement a recursive function print\_ast() that takes a node and an indentation level as input. It prints the node's type and value, and then recursively calls itself to print its left and right children, increasing the indentation level each time.

**Code :**

# PARSER

class Node:

def \_\_init\_\_(self, Type, value):

self.Type = Type

self.value = value

self.left = None

self.right = None

def parse\_expression(tokens):

operators = {'+', '-', '\*', '/', '%', '^'}

precedence = {

'+': 1,

'-': 1,

'\*': 2,

'/': 2,

'%': 2,

'^': 3,

}

def is\_higher\_precedence(op1, op2):

return precedence[op1] >= precedence[op2]

def create\_binary\_node(op\_stack, value\_stack):

right = value\_stack.pop()

left = value\_stack.pop()

operator = op\_stack.pop()

node = Node('OPERATOR', operator)

node.left = left

node.right = right

value\_stack.append(node)

op\_stack = []

value\_stack = []

for token\_Type, token\_value in tokens:

if token\_Type == 'NUMBER' or token\_Type == 'VARIABLE':

value\_stack.append(Node(token\_Type, token\_value))

elif token\_Type == 'OPERATOR':

while ( op\_stack and op\_stack[-1] in operators and is\_higher\_precedence(op\_stack[-1], token\_value)):

create\_binary\_node(op\_stack, value\_stack)

op\_stack.append(token\_value)

elif token\_Type in ('PARENTHESIS', 'BRACKET'):

if token\_value in '({[':

op\_stack.append(token\_value)

elif token\_value in ')}]':

while op\_stack and op\_stack[-1] not in '({[':

create\_binary\_node(op\_stack, value\_stack)

if op\_stack and op\_stack[-1] in '({[':

op\_stack.pop()

else:

raise SyntaxError(f"Unmatched closing bracket: {token\_value}")

while op\_stack:

create\_binary\_node(op\_stack, value\_stack)

return value\_stack[0]

def print\_ast(node, indent=0):

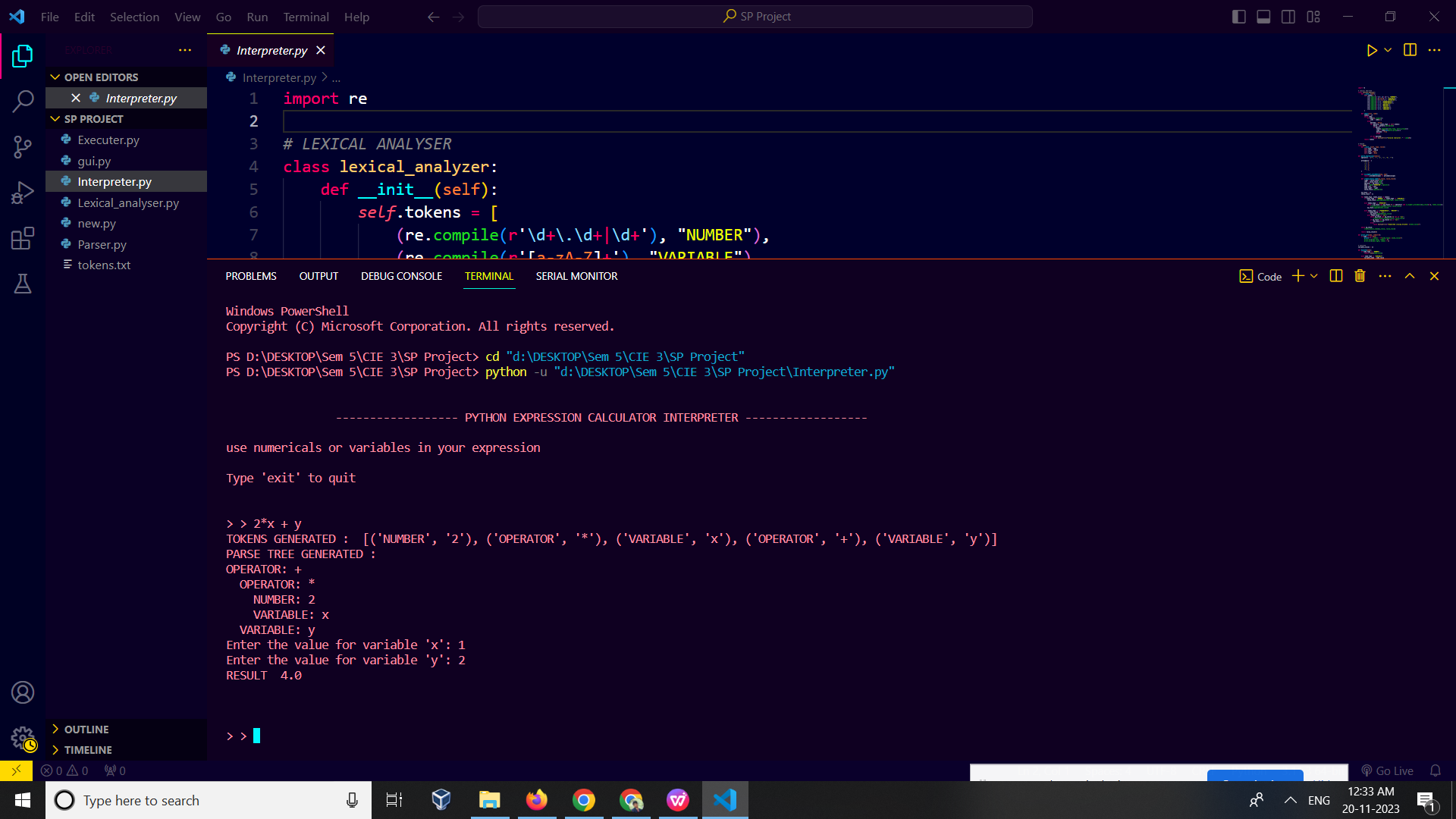
if node is not None:

print(" " \* indent + f"{node.Type}: {node.value}")

print\_ast(node.left, indent + 2)

print\_ast(node.right, indent + 2)

**Output:**



**Conclusion :**

Analysis of our parse tree :

The parse tree starts with the + operator, which is the root node of the tree. The left child of the + operator is the NUMBER node with the value 2. The right child of the + operator is another OPERATOR node with the value X. The X node is the variable node in the expression.

The OPERATOR node with the value X has two children: the VARIABLE node with the name x and the VARIABLE node with the name y. These two nodes represent the variables in the expression.

The parse tree is evaluated by traversing it from the root node to the leaf nodes. The value of each node is evaluated and then combined with the values of its children to produce the final result.

In this case, the parse tree would be evaluated as follows:

1. The value of the NUMBER node with the value 2 is retrieved.
2. The value of the VARIABLE node with the name x is retrieved.
3. The value of the VARIABLE node with the name y is retrieved.
4. The + operator is evaluated using the values of the NUMBER node and the VARIABLE node with the name x.
5. The + operator is evaluated using the result of step 4 and the value of the VARIABLE node with the name y.

The final result of the evaluation is the value of the + operator in the root node of the parse tree.