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

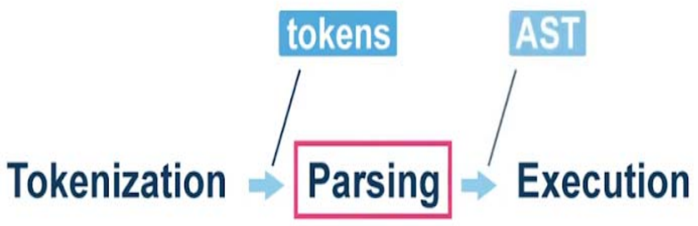
Project Statement :
Implementing an Expression calculator based Interpreter in Python and Demonstrating its
3 Phases: - Lexical , Parser and Execution.

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)

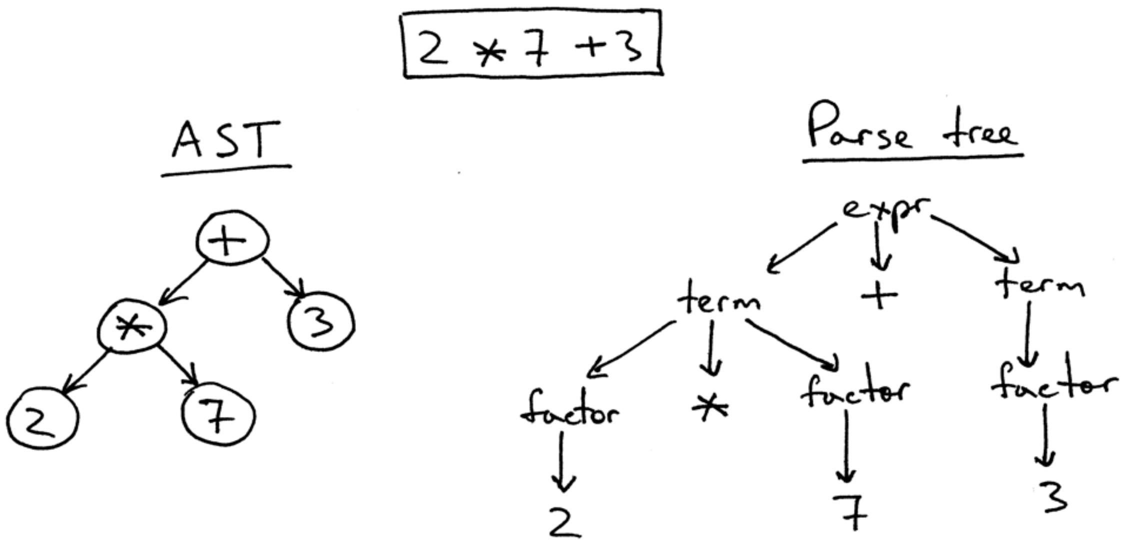


PROJECT MODULE 2 : SYNTAX ANALYSER /PARSE TREE

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.

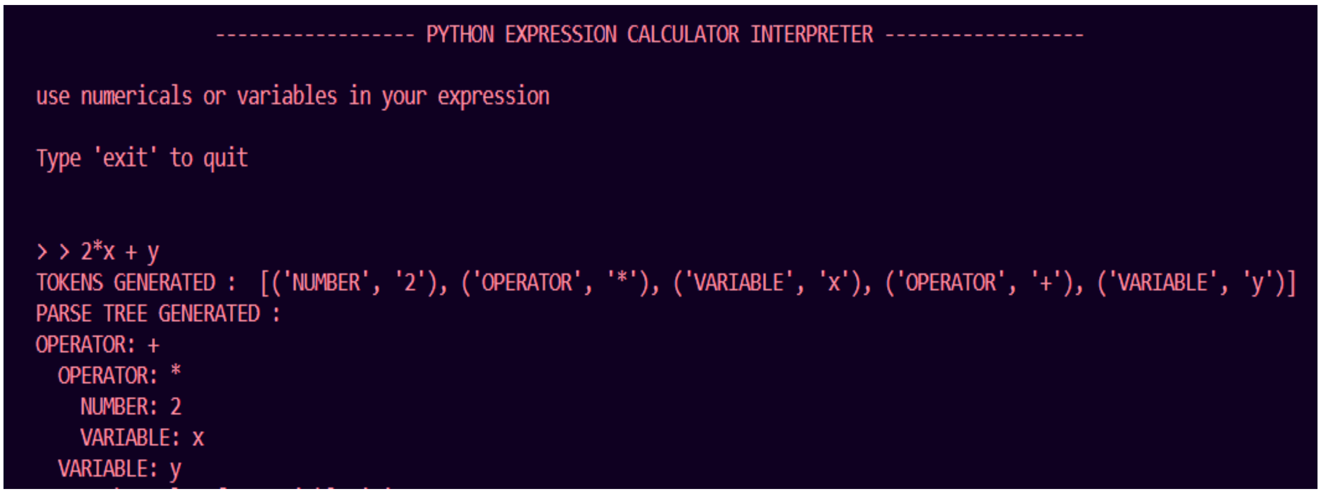
Code :

```
# 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:



```
                 ----------------- PYTHON EXPRESSION CALCULATOR INTERPRETER -----------------

  use numericals or variables in your expression

  Type 'exit' to quit


  > > 2*x + y
  TOKENS GENERATED :  [('NUMBER', '2'), ('OPERATOR', '*'), ('VARIABLE', 'x'), ('OPERATOR', '+'), ('VARIABLE', 'y')]
  PARSE TREE GENERATED :
  OPERATOR: +
    OPERATOR: *
      NUMBER: 2
      VARIABLE: x
    VARIABLE: y
```

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.