



VISHWAKARMA
UNIVERSITY
Maximising Human Potential

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 mathematical 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)

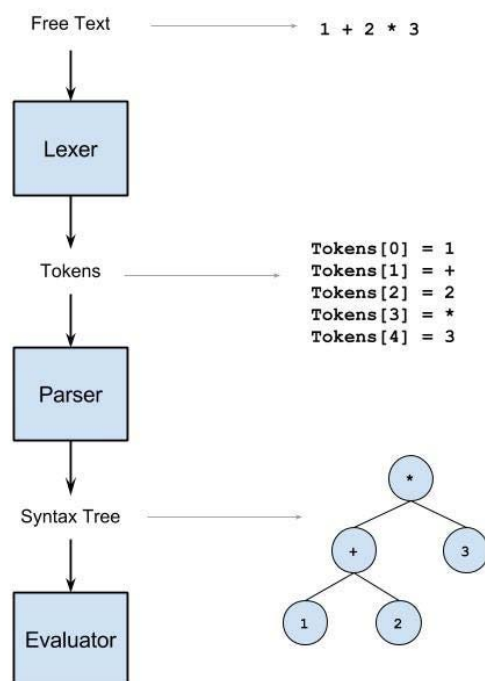


PROJECT MODULE 3 : Evaluator

Executer / Evaluator The evaluator is the final stage of our expression calculator interpreter, responsible for calculating the numerical value of a given mathematical expression (which contains both alphabets and numbers in expression). It takes the output of the parser, a parse tree, and traverses the tree to perform the arithmetic operations. The evaluator implements the order of operations, ensuring that expressions are evaluated according to the correct precedence rules.

Uses the basic rules of Bodmas and evaluates the expression to give numerical output.

The evaluator plays a crucial role in my expression calculator interpreter, transforming symbolic expressions into their numerical counterparts as shown in above image . Its ability to handle various data types.



Implementation :

Implementation procedure

Implement `Execute()` function: This function takes an AST node as input and evaluates the corresponding expression.

Numbers: If the node's type is `NUMBER`, simply return its value as a float.

Variables: If the node's type is `VARIABLE`, check if the variable name is present in the `variable_values` dictionary. If not, prompt the user to enter the value of the variable and store it in the dictionary. Then, return the value of the variable.

Operators: If the node's type is `OPERATOR`, recursively evaluate the left and right children of the node (operands) and perform the corresponding operation on their returned values.

loop: Implement a main loop that prompts the user for an expression, tokenizes it using the `lexeme` instance, parses the tokens into an AST using the `parse_expression()` function, prints the AST, evaluates the expression using the `Execute()` function, prints the result, and clears the `variable_values` dictionary for the next expression.

Run the interpreter: Invoke the main loop to continuously prompt the user for expressions, evaluate them, and display the results until the user enters 'exit'

Code :

```
# EXECUTER
```

```
variable_values = {}
```

```
def Execute(node):
```

```
    if node.Type == 'NUMBER':
```

```
        return float(node.value)
```

```
    elif node.Type == 'VARIABLE':
```

```
variable_name = node.value
if variable_name not in variable_values:
    value = input(f"Enter the value for variable
'{variable_name}': ")
    variable_values[variable_name] = float(value)
return variable_values[variable_name]
```

```
elif node.Type == 'OPERATOR':
    left = Execute(node.left)
    right = Execute(node.right)
```

```
if node.value == '+':
    return left + right
```

```
elif node.value == '-':
    return left - right
```

```
elif node.value == '*':
    return left * right
```

```
elif node.value == '/':
    if right == 0:
```

```
        raise ZeroDivisionError("Division by zero")
    return left / right
```

```
elif node.value == '%':
    return left % right
```

```
elif node.value == '^':
    return left ** right
```

```
# Main loop
```

```
if __name__ == "__main__":
```

```
    print("\n\n\t\t-----          PYTHON          EXPRESSION
CALCULATOR INTERPRETER -----\n\nuse numericals or
variables in your expression\n\nType 'exit' to quit\n\n")
```

```
lexeme = lexical_analyzer()
```

```
while True:
```

```
    code = input("> > ")
```

```
    if code.lower() == 'exit':
```

```
        break
```

```
try:
    Tokens = lexeme.lexer(code)
    print("TOKENS GENERATED : ", Tokens)

    parse_tree = parse_expression(Tokens)

    print("PARSE TREE GENERATED : ")
    print_ast(parse_tree, 0)

    result = Execute(parse_tree)
    print("RESULT ", result)
    print("\n\n")

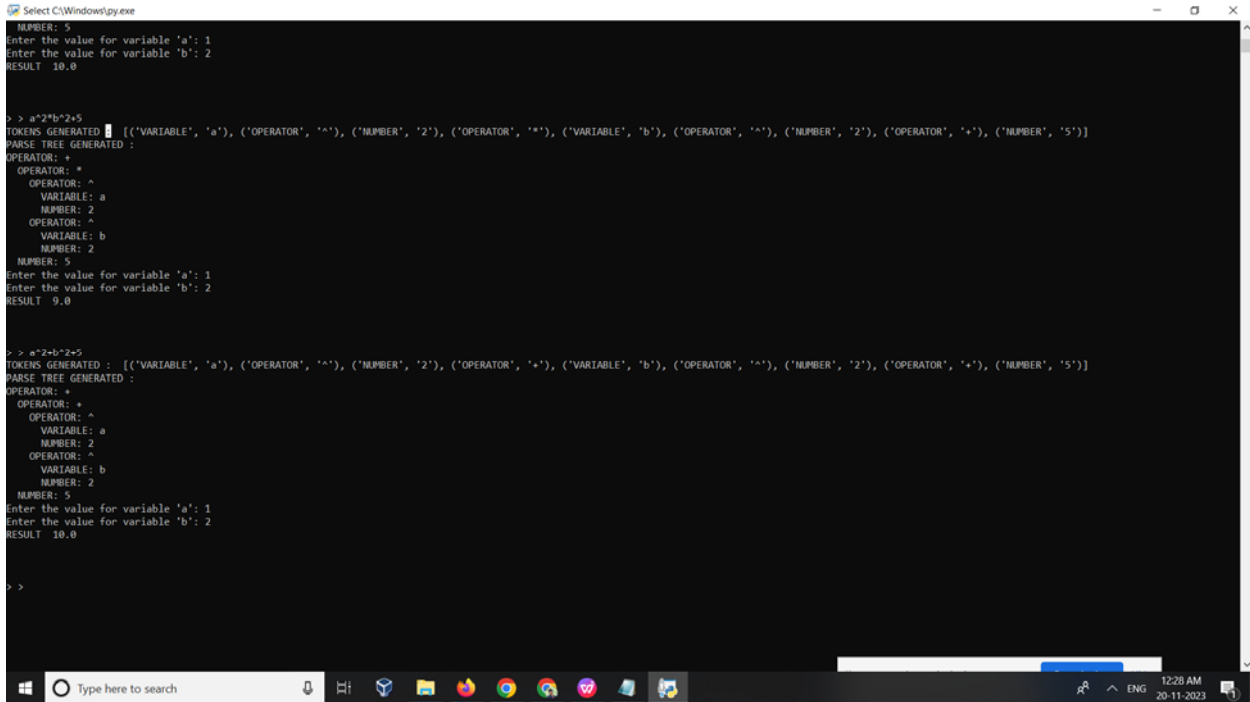
except SyntaxError as e:
    print("Syntax Error:", e)

except ValueError as e:
    print("Value Error:", e)

variable_values = {}
```

Output:

Take a screen shot of project output



```

C:\Windows\py.exe
NUMBER: 5
Enter the value for variable 'a': 1
Enter the value for variable 'b': 2
RESULT 10.0

> a^2*b^2*5
TOKENS GENERATED : [('VARIABLE', 'a'), ('OPERATOR', '^'), ('NUMBER', '2'), ('OPERATOR', '*'), ('VARIABLE', 'b'), ('OPERATOR', '^'), ('NUMBER', '2'), ('OPERATOR', '*'), ('NUMBER', '5')]
PARSE TREE GENERATED :
OPERATOR: +
  OPERATOR: *
    OPERATOR: ^
      VARIABLE: a
      NUMBER: 2
    OPERATOR: ^
      VARIABLE: b
      NUMBER: 2
  NUMBER: 5
Enter the value for variable 'a': 1
Enter the value for variable 'b': 2
RESULT 9.0

> a^2+b^2*5
TOKENS GENERATED : [('VARIABLE', 'a'), ('OPERATOR', '^'), ('NUMBER', '2'), ('OPERATOR', '+'), ('VARIABLE', 'b'), ('OPERATOR', '^'), ('NUMBER', '2'), ('OPERATOR', '*'), ('NUMBER', '5')]
PARSE TREE GENERATED :
OPERATOR: +
  OPERATOR: ^
    VARIABLE: a
    NUMBER: 2
  OPERATOR: ^
    VARIABLE: b
    NUMBER: 2
  NUMBER: 5
Enter the value for variable 'a': 1
Enter the value for variable 'b': 2
RESULT 10.0

>

```


Conclusion :

The evaluator then checks the parse tree to make sure that it is correct. It checks that the parse tree is a valid Python expression, that the operators are in the correct order, and that the variables are in the correct scope.

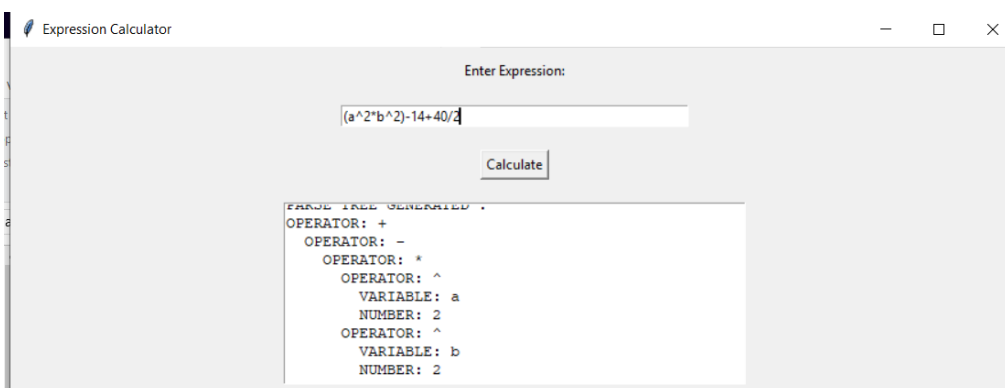
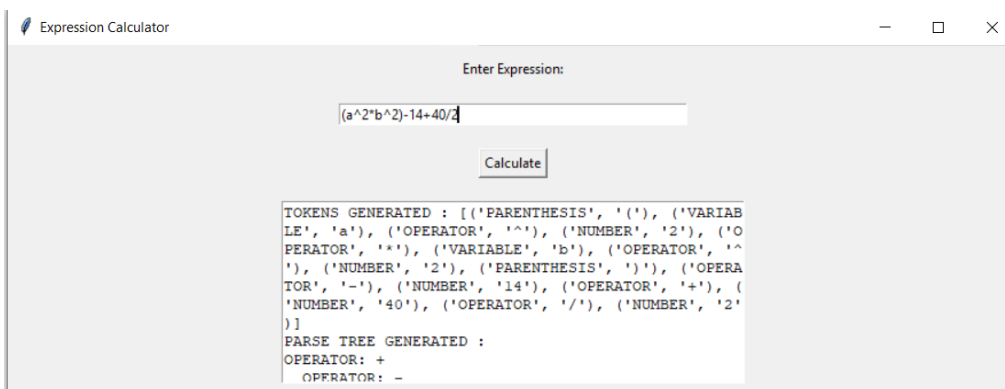
The evaluator also checks the parse tree to make sure that it is efficient. It checks that the parse tree does not contain any unnecessary nodes and that the nodes are arranged in a way that will make it easy to evaluate the expression.

Once the evaluator has checked the parse tree, it provides feedback to the parse tree generator. The feedback can be used to improve the parse tree generator and to make it more efficient.

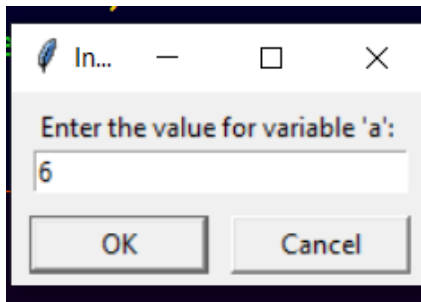
Sample values

a = 6

b = 8

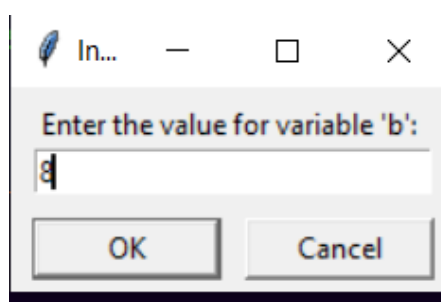


System Programming



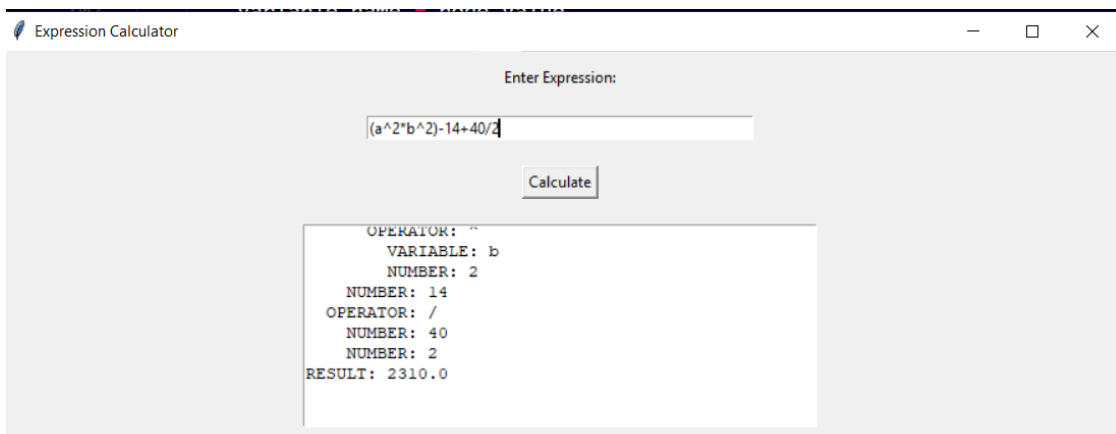
Enter the value for variable 'a':

OK Cancel



Enter the value for variable 'b':

OK Cancel



Expression Calculator

Enter Expression:

Calculate

```
OPERATOR: ^  
VARIABLE: b  
NUMBER: 2  
NUMBER: 14  
OPERATOR: /  
NUMBER: 40  
NUMBER: 2  
RESULT: 2310.0
```

Answer: 2310