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

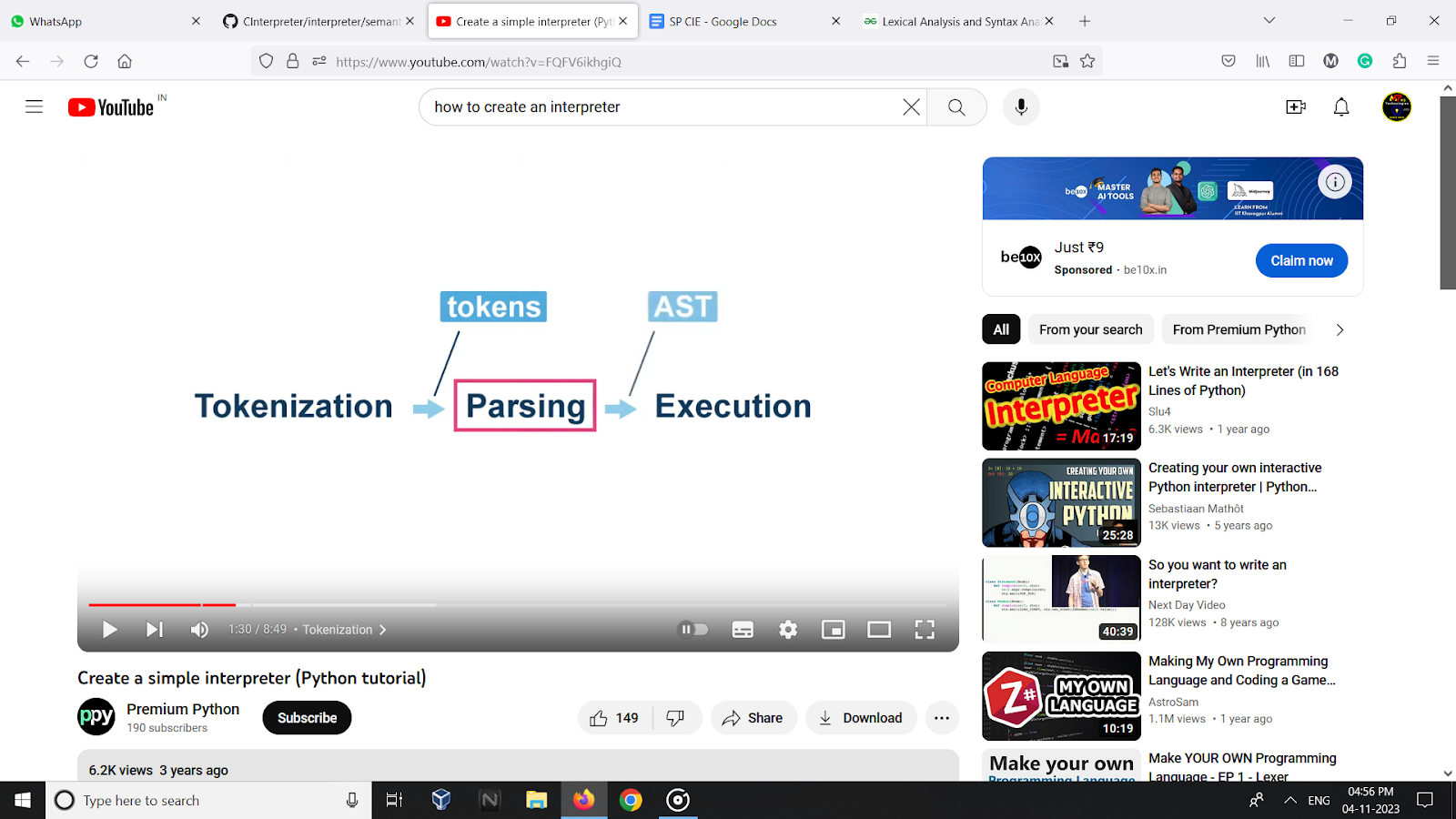
3 Phases: - Lexical , Parser and Execution.

**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**

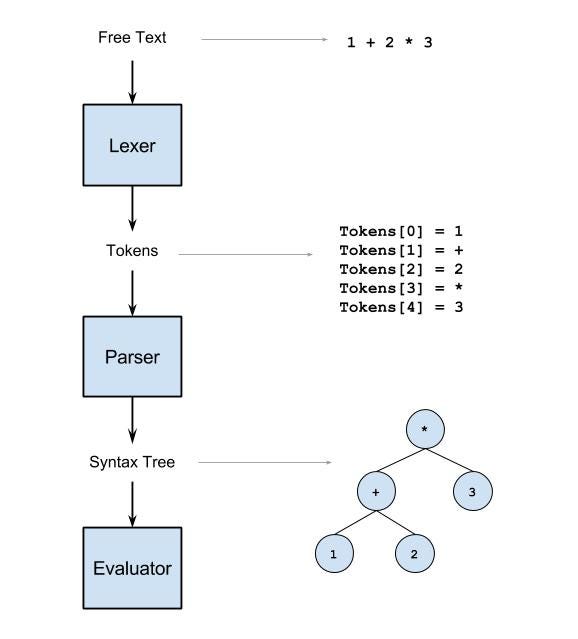
1. Lexical Analysis ( tokens )
2. Parser ( Syntax tree )
3. Execution( executes the program )

**PROJECT MODULE 3 : Evaluator**

**Executer / Evaluator** The evaluator is the final stage of ourexpression 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)

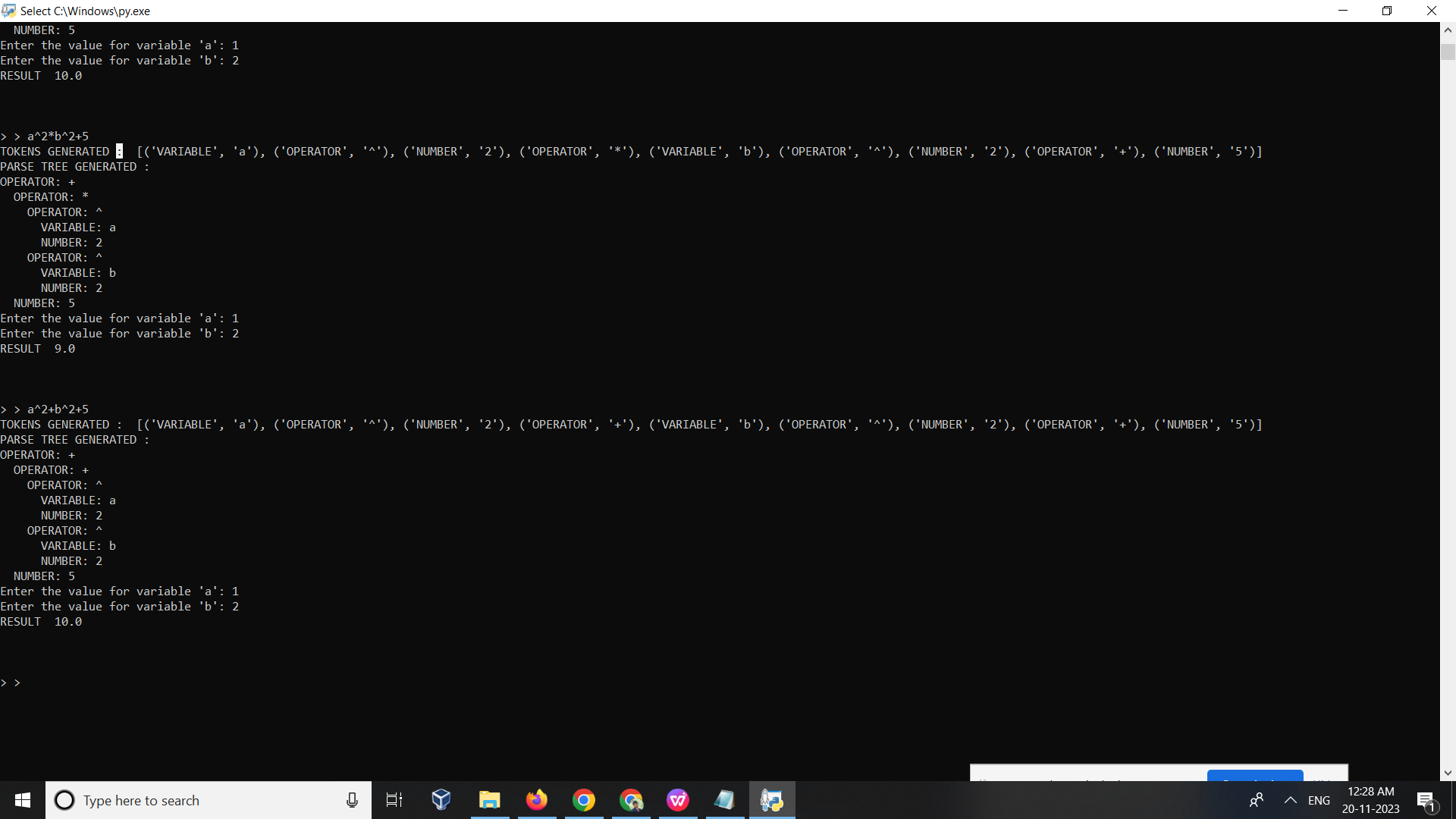
except ValueError as e:

print("Value Error:", e)

variable\_values = {}

**Output:**

**Take a screen shot of project output**



**Conclusion :**

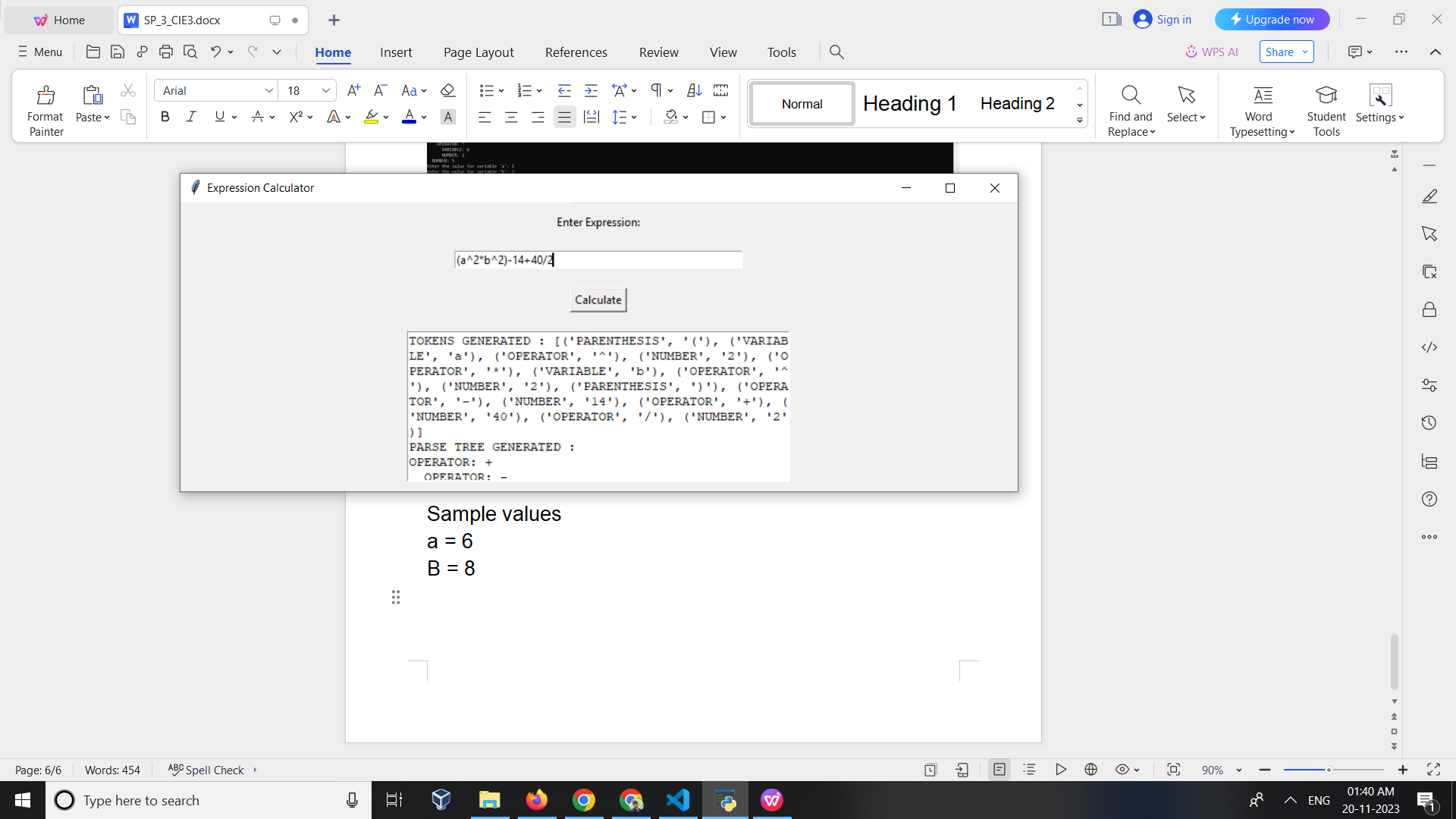
The evaluator then checks the parse tree to make sure that it is correct. It checks that the parse tree is a valid Python expression, that the operators are in the correct order, and that the variables are in the correct scope.

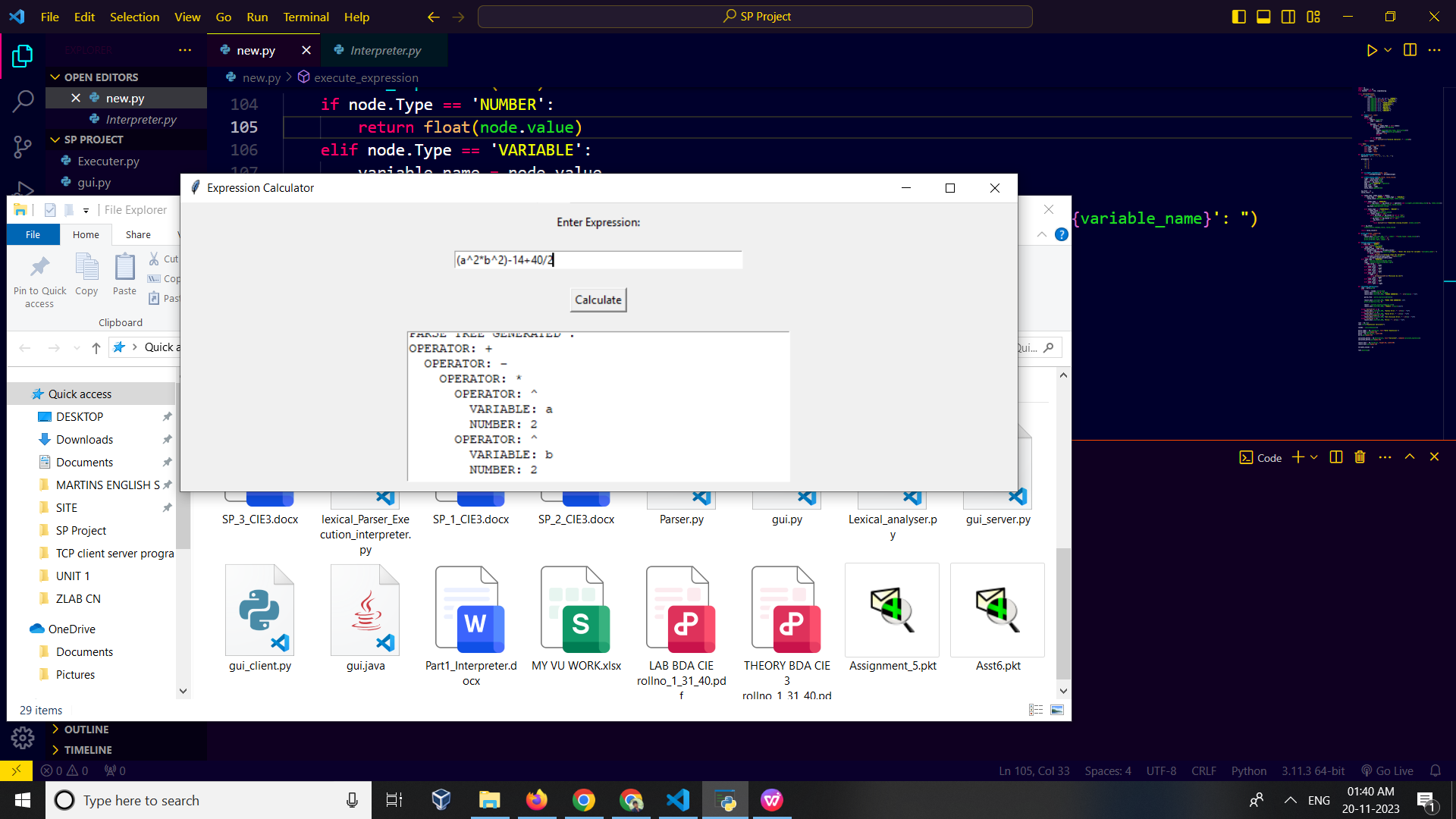
The evaluator also checks the parse tree to make sure that it is efficient. It checks that the parse tree does not contain any unnecessary nodes and that the nodes are arranged in a way that will make it easy to evaluate the expression.

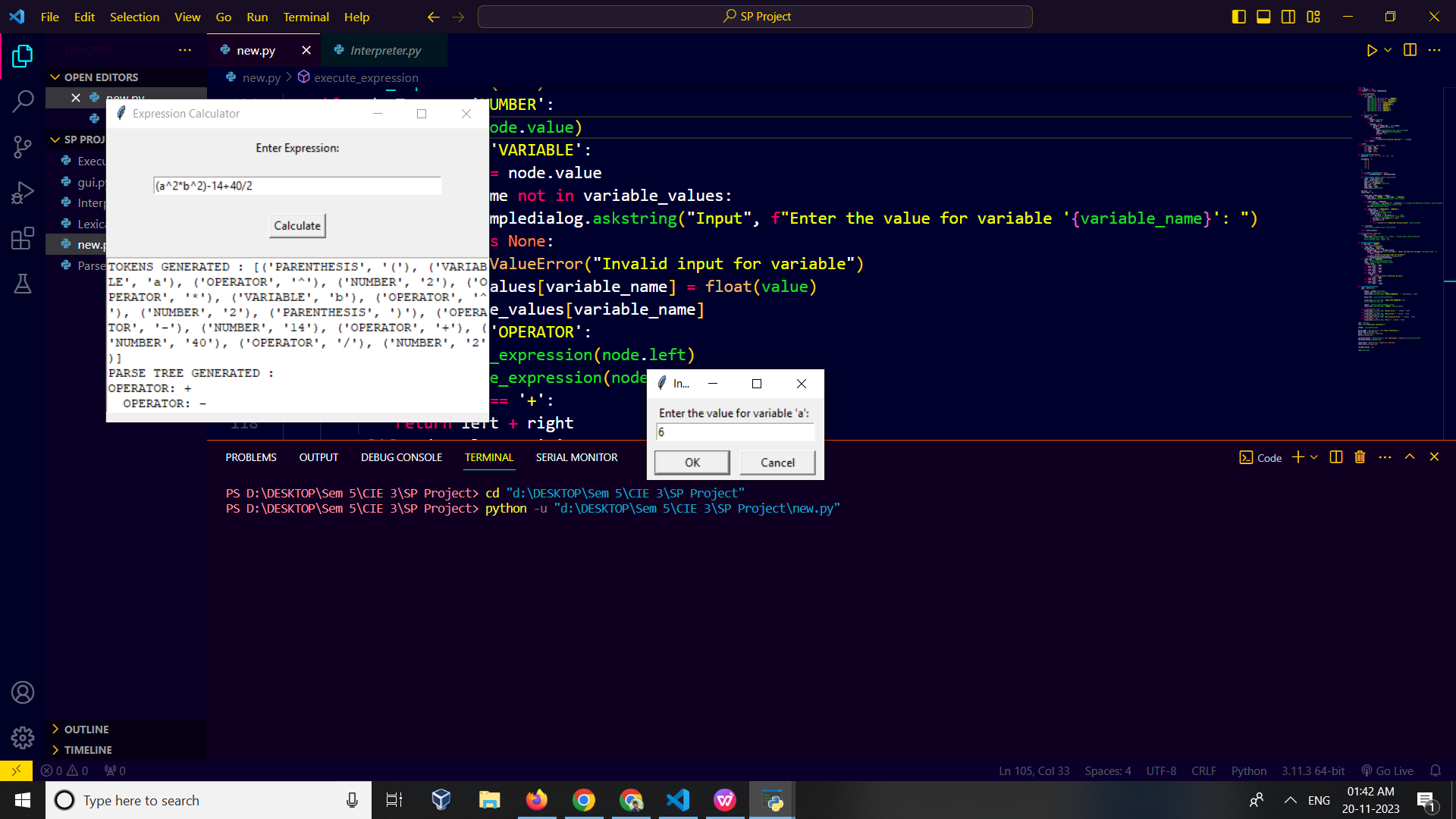
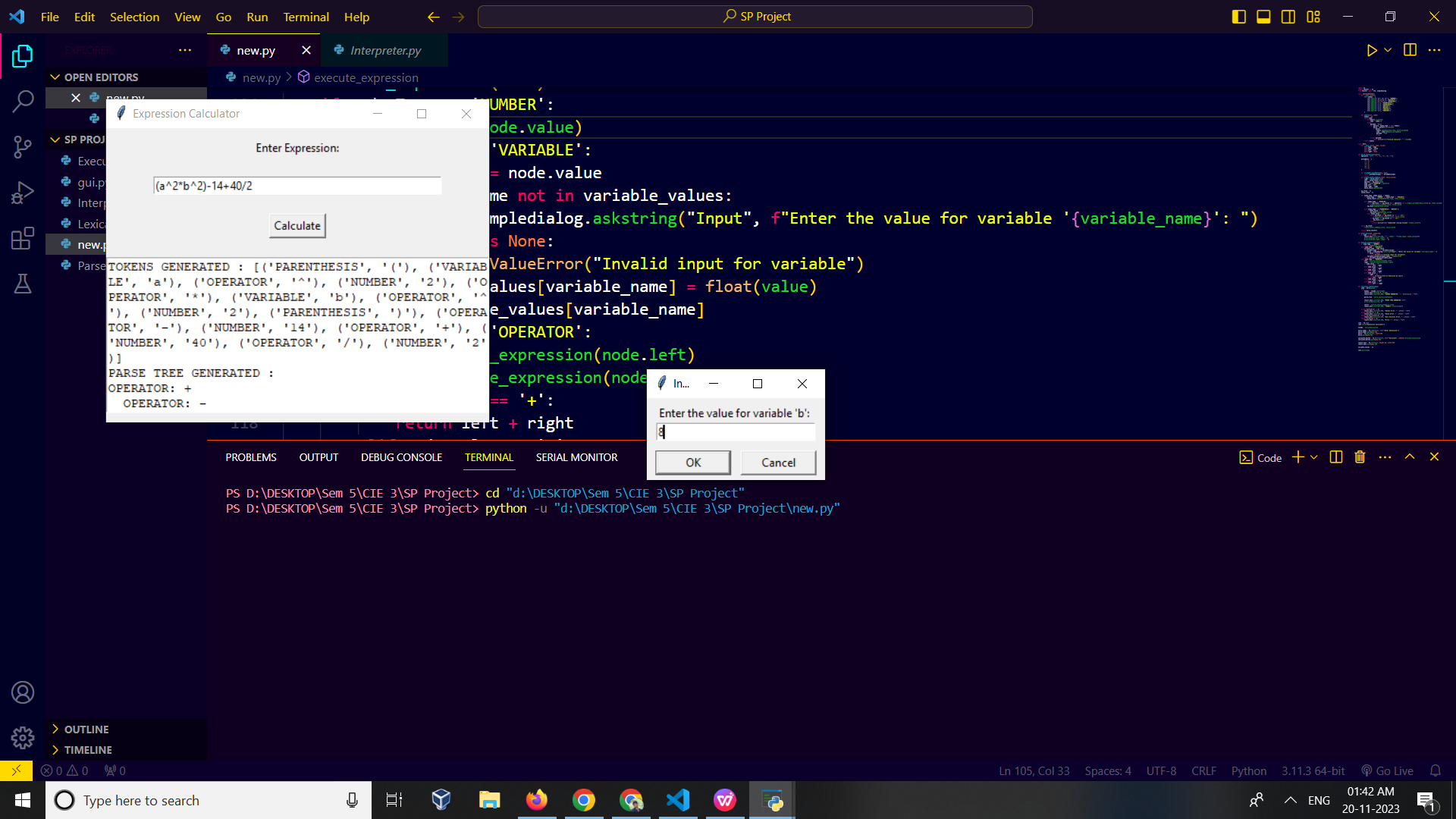
Once the evaluator has checked the parse tree, it provides feedback to the parse tree generator. The feedback can be used to improve the parse tree generator and to make it more efficient.

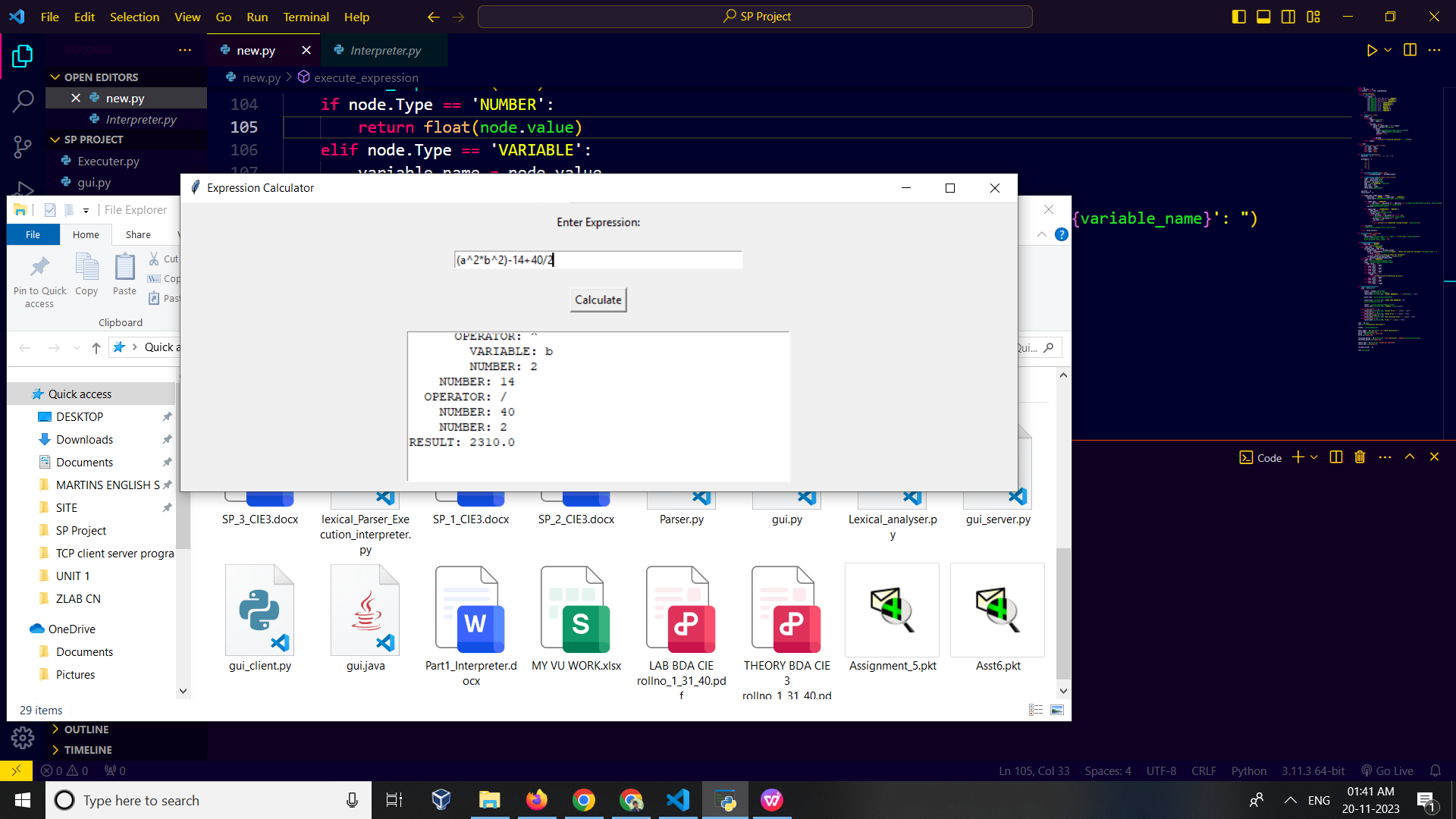
Sample values   
a = 6

b = 8









**Answer: 2310**