Fully Parenthesized Arithmetic Expression Evaluator – The Expression Solver Challenge

Overview

Imagine a secret mathematician who has hidden important numerical clues inside fully parenthesized arithmetic expressions. In this challenge, your mission is to create a function that can evaluate these expressions correctly. Since every operator in the expression is explicitly grouped with its operands by parentheses, your solution will carefully follow the structure dictated by the parentheses to compute the final result.

Project Objective

Develop a function that:

- Accepts a fully parenthesized arithmetic expression as a string.
- Evaluates the expression by processing the nested structure.
- Returns the computed numerical result.

Constraints

- **Language-Agnostic Design:** Your solution should be implementable in multiple programming languages (e.g., Java or Python).
- **Tokenization Assumption:** Assume that the expression's tokens (numbers, operators, and parentheses) are separated by spaces.
- **Stack Utilization:** You may not use built-in data structures but implement your own version of a stack to manage operators and operands.

Input/Output Format

- **Input:** A string that represents a fully parenthesized arithmetic expression.
  Example: `( ( 2 + 3 ) * ( 4 - 1 ) )`
- **Output:** A numerical result that is the evaluation of the expression.
  Example: `15`

Test Case Explanation

For the input expression:
`( ( 2 + 3 ) * ( 4 - 1 ) )`

1. **Parsing the Expression:**

- The sub-expression `( 2 + 3 )` is evaluated first to yield `5`.
- The sub-expression `( 4 - 1 )` is then evaluated to yield `3`.

2. **Combining Results:**
   - The two results are combined using the operator `*`, so the overall result becomes `5 * 3 = 15`.

3. **Output:**
   The evaluated result is `15`.