**Analysis For Dijkstra Two Stack Calculator**

Overview

A solution must be found to evaluate arithmetic expressions that are fully parenthesised (in brackets) using an algorithm. The expression may contain integers, decimals, parentheses and operators (the operators being the +, -, \*, and /) meaning add, subtract, multiply and divide respectively. There should be no operator precedence meaning that no operation should take priority over another due to its operator (i.e. in BIDMAS division takes precedence over subtraction). Instead, all precedence should be defined by parentheses which show the order of operations to happen.

Objectives Of The Project:  
  
The objective is to implement a calculator using Dijkstra’s two-stack algorithm. It should be able to read the infix expression (meaning the operators are in between the operands) by processing expressions by reading left to right. It should then return a final result based on the evaluated sub-expressions.

Why Dijkstra’s Two-Stack Algorithm?

Dijkstra’s Two-Stack Algorithm is actually designed for evaluating fully parenthised infix expressions, hence, using an algorithm which is designed to solve the problem at hand will likely be the most suitable. It guarantees correct evaluation order of sub-expressions without the need of precedence rules and instead uses the parentheses, which is also required in the problem statement. Furthermore, this algorithm is simple and stack-based. This is because logic is procedural - you go from left to right and perform actions based on each token. There is no need for operator precedence handling because the parentheses explicitly define order of operations. Additionally, there are only a few conditional checks per token. For example, “Is this a number, is this an operator, is this a “(“, is this a “)”.

User requirements.

The user should be able to input an arithmetic expression that is fully parenthesised. This means that every single operation is enclosed in brackets, so that it clearly shows the order in which calculations should happen. The expression will include integers and basic operators (as mentioned above) and should return the correct answer after evaluating it.

The program should be efficient, and should be able to work in a time that matches the size of the input which is O(n). It should also be easy to understand and well-organised, with clear sections for reading input, doing calculations, and showing results. This is because it makes it easier to test and update later if more features are added.  
  
Constraints and limitations:

All of the expressions must be fully parenthesised and also the program will be assuming valid inputs. This means that it wont handle negative numbers on its own like -5 or extra symbols like ^ for exponentiation (or powers).  
  
The calculator should return correct answers for all test cases and handle complex nested brackets. It must not crash during normal use and should give instant results. The code should be clean, well-structured by using a modular approach.