Analysis of an Arithmetic Expression

According to Tenembaum in "Data Structures Using C", consider a mathematical expression that includes several sets of grouped parentheses. For example:

7 - ((X * ((X + Y) / (J - 3)) + Y) / (4 - 2.5))

We want to ensure that the parentheses are properly grouped, meaning we want to verify that:

1. There is an equal number of left and right parentheses.

2. Every right parenthesis is preceded by a corresponding left parenthesis.

Expressions like ((A + B) or A + B{ violate criterion 1, and expressions like)A + B(-C or (A + B)) - (C + D violate criterion 2.

To solve this problem, imagine every left parenthesis as the opening of a scope, and every right parenthesis as the closing of a scope.

Now let's slightly modify the problem and suppose there are three types of scope delimiters. These types are indicated by:

- Parentheses: (and)

- Brackets: [and]

- Braces: { and }

A closing delimiter must match the type of its opening delimiter. Therefore, strings such as:

(A + B], [(A + B)], {A - (B]} are invalid.

It's necessary to track not only how many scopes were opened but also their types. This information is important because when a closing delimiter is found, we need to know the symbol used to open the scope to ensure it is closed correctly.

A stack can be used to track the types of scopes encountered. Whenever an opening delimiter is found, it is pushed onto the stack. Whenever a closing delimiter is found, the stack is examined. If the stack is empty, the closing delimiter has no corresponding opener and the string is invalid. If the stack is not empty, we pop and verify if the item popped corresponds to the closing delimiter. If it matches, we continue. Otherwise, the string is invalid. When the end of the string is reached, the stack should be empty; if not, there are still unclosed scopes and the string is invalid.

```
Algorithm: analyzeExpression
Variables:
    expression : string
    i : integer
    symbol : char
    valid : boolean
    p : Stack

Begin
    read(expression)
    i := 0
    valid := true

    while i < length(expression) do
        symbol := expression[i]
```

```
        if (symbol = '{') or (symbol = '[') or (symbol = '(') then
            p.push(symbol)
        else if (symbol = '}') or (symbol = ']') or (symbol = ')') then
            if p.isEmpty() then
                valid := false
            else if (symbol = '}') and (p.top() = '{') then
                p.pop()
            else if (symbol = ']') and (p.top() = '[') then
                p.pop()
            else if (symbol = ')') and (p.top() = '(') then
                p.pop()
            else
                break
        end if

        i := i + 1
    end while

    if p.isEmpty() and valid then
        print("Expression is Correct")
    else
        print("Expression is Incorrect")
End
```