

**VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY**  
**UNIVERSITY OF SCIENCE**  
**FACULTY INFORMATION TECHNOLOGY**



**PROJECT REPORT**  
**CHALLENGE 1**

**Data Structure && Algorithms**  
**GROUP ID 273346**

**MEMBERS**

19127533 – Le Hoang Anh Quoc  
18127027 – Tran Minh Duc  
18127246 – Tran Quoc Tuan

**LECTURER**

Mr. Van Chi Nam  
Mr. Bui Huy Thong

**HO CHI MINH CITY, 16/11/2020**

## TABLE OF CONTENTS

|                             |    |
|-----------------------------|----|
| .....                       | 1  |
| I. OVERVIEW .....           | 1  |
| 1. Introduction .....       | 1  |
| 2. Members.....             | 1  |
| 3. Working Progress.....    | 1  |
| II. RESEARCH .....          | 2  |
| 1. Infix to prefix .....    | 2  |
| 2. Infix to postfix:.....   | 4  |
| 3. Prefix to postfix:.....  | 6  |
| III. CODE FRAGMENT .....    | 7  |
| 1. Class diagram.....       | 7  |
| 2. Class descriptions ..... | 8  |
| 3. Implementations.....     | 9  |
| IV. REFERENCES .....        | 13 |

## TABLE OF FIGURES

|   |    |
|---|----|
| Image 1 Class Diagram .....               | 8  |
| Image 2 Represent ArExp .....             | 9  |
| Image 3 Infix_to_postfix preprocess ..... | 10 |
| Image 4 evaluate loop token .....         | 11 |
| Image 5 Evaluate() handle stack .....     | 12 |
| Image 6 Evaluate() return result .....    | 12 |

## I. OVERVIEW

### 1. Introduction

We use infix expressions in our daily lives but computers have trouble understanding this format. In this paper, we focus on researching about some algorithms that converts among infix, prefix, postfix expression.

Besides, Our group will implement a tiny application to convert infix expression to postfix expression and calculate the expressions with float number similar as the calculator.

### 2. Members

| <i>Fullname</i>          | <i>Student ID</i> |
|--------------------------|-------------------|
| <b>Lê Hoàng Anh Quốc</b> | <b>19127533</b>   |
| <b>Trần Minh Đức</b>     | <b>18127027</b>   |
| <b>Trần Quốc Tuấn</b>    | <b>18127246</b>   |

Table 1 Members

### 3. Working Progress

| <b>Assignment</b>  | <b>Progress</b>             | <b>Note</b>            |
|--|-----------------------------|------------------------|
| <b>1.1 Research:</b>   |                             | Reported by Word.      |
| <b>Converting infix to prefix</b>                                  | <div><div></div></div> 100% |                        |
| <b>Converting infix to postfix</b>                                 | <div><div></div></div> 100% |                        |
| <b>Converting prefix to postfix</b>                                | <div><div></div></div> 100% |                        |
| <b>1.2 Programming:</b>  |                             | Floating-point option. |
| <b>Calculate the identified expressions</b>                        | <div><div></div></div> 100% |                        |
| <b>Convert the identified expressions into postfix expressions</b> | <div><div></div></div> 100% |                        |
| <b>Information of code fragment</b>                                | <div><div></div></div> 100% | Reported with 1.1      |

Table 2 Working Progress

## II. RESEARCH

| Infix Expression    | Prefix Expression | Postfix Expression |
|---------------------|-------------------|--------------------|
| $A + B$             | $+ A B$           | $A B +$            |
| $A + B * C$         | $+ A * B C$       | $A B C * +$        |
| $(A + B) * C$       | $* + A B C$       | $A B + C *$        |
| $A + B * C + D$     | $+ + A * B C D$   | $A B C * + D +$    |
| $(A + B) * (C + D)$ | $* + A B + C D$   | $A B + C D + *$    |

### 1. Infix to prefix

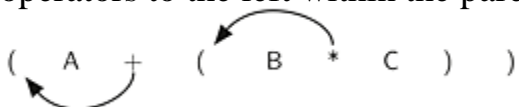
#### a) Mathematics:

- Fully parenthesized expression approach:

+ Parentheses are placed around each pair of operands and its associated operator following the precedence rule and dictate the order of operations.

Ex:  $A + B * C \Rightarrow (A + (B * C))$

+ Move the operators to the left within the parentheses covering them.

Ex: 

+ Remove all parentheses and eventually, the infix expression is converted to prefix expression.

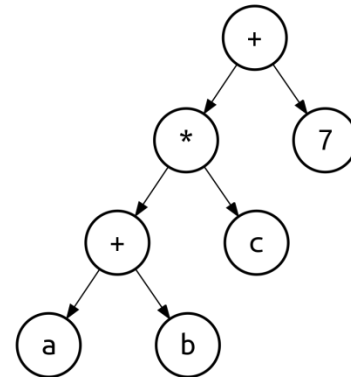
Ex:  $(+ A (* B C)) \Rightarrow + A * B C$

- Binary expression tree approach:

+ The leaves (outer vertexes) of a binary expression tree are operands and the inner vertexes contain operators. The top vertex contains the operator that has the lowest

priority. From top to bottom, the precedence of operator that the inner vertex embodies increases.

Ex:  $(a + b) * c + 7 \Rightarrow$



+ Reading the binary tree from the top node to the leaves and then from left to right, we will obtain prefix expression from infix expression.

Ex:  $+ \rightarrow * \rightarrow + \rightarrow a \rightarrow b \rightarrow c \rightarrow 7 \Rightarrow + * + a b c 7$

## b) Programming:

- To convert infix expression to prefix expression, we will convert the reversed infix expression to postfix expression by using the *shunting-yard algorithm*. After that, reverse the resultant postfix expression and the prefix expression is obtained.

- Algorithm:

+ Create an empty stack for keeping operators and an empty string list for output. Reverse the infix expression and note that each ( will become ) and each ) becomes (. Scan the result from left to right.

+ If the scanned character is an operand, add it to the list for output.

+ If the the scanned operator has precedence over the operators in the stack or the stack is empty or contains a (, push it into the stack.

+ Otherwise, pop all the operators in the stack of which precedence is greater than or equal to that of the scanned operator and put them in the list then push the scanned operator to the stack. When you encounter a parenthesis while popping then stop and push the scanned operator in the stack.

+ If the scanned character is (, push it to the stack. If it is ), pop the stack and add the elements to the list for output until a ( is encountered and discard both round brackets.

+ Repeat above steps until expression is fully scanned.

+ Finally, pop the operators from the stack and put them in the list one by one until the stack is empty then print the string list from right to left (reverse order).

c) Example:  $A * (B + C) - D$

$$A * (B + C) - D \xrightarrow{\text{reverse}} D - (C + B) * A$$

| Token    | Action                     | Output        | Stack | Notes                     |
|----------|----------------------------|---------------|-------|---------------------------|
| <b>D</b> | Add token to output        | D             |       |                           |
| <b>-</b> | Push token to stack        | D             | -     |                           |
| <b>(</b> | Add token to stack         | D             | - (   |                           |
| <b>C</b> | Push token to output       | D C           | - (   |                           |
| <b>+</b> | Add token to stack         | D C           | - ( + |                           |
| <b>B</b> | Push token to output       | D C B         | - ( + |                           |
| <b>)</b> | Pop stack to output        | D C B +       | - (   | Repeated until ( is found |
|          | Pop stack                  | D C B +       | -     | Remove the parentheses    |
| <b>*</b> | Add token to stack         | D C B +       | - *   |                           |
| <b>A</b> | Push token to output       | D C B + A     | - *   |                           |
|          | Pop entire stack to output | D C B + A * - |       |                           |

$$D C B + A * - \xrightarrow{\text{reverse}} - * A + B C D$$

## 2. Infix to postfix:

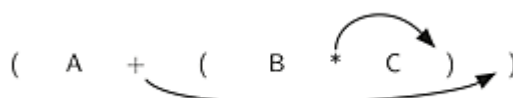
### a) Mathematics:

- Fully parenthesized expression approach:

+ Parentheses are placed around each pair of operands and its associated operator following the precedence rule and dictate the order of operations.

Ex:  $A + B * C \Rightarrow (A + (B * C))$

+ Move the operators to the right within the parentheses covering them.



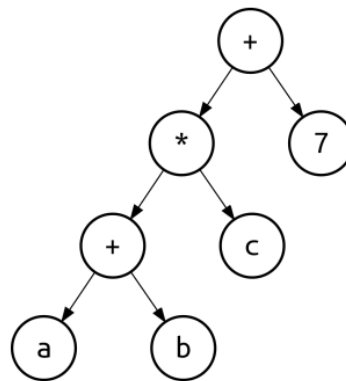
Ex:

+ Remove all parentheses and eventually, the infix expression is converted to prefix expression.

Ex:  $(A (B C *) +) \Rightarrow A B C * +$

- Binary expression tree approach:

+ The leaves (outer vertexes) of a binary expression tree are operands and the inner vertexes contain operators. The top vertex contains the operator that has the lowest priority. From top to bottom, the precedence of operator that the inner vertex embodies increases.



Ex:  $(a + b) * c + 7 \Rightarrow$

+ Reading the binary tree from left to right and then from the leaves to the top node, we will obtain postfix expression from infix expression.

Ex:  $a \rightarrow b \rightarrow + \rightarrow c \rightarrow * \rightarrow 7 \rightarrow + \Rightarrow a b + c * 7 +$

## b) Programming:

- To convert infix expression to postfix expression, we will use the *shunting-yard algorithm*. By scanning the infix expression from left to right, when we get any operand, simply add them to the postfix form and as for the operator and parenthesis, add them in the stack maintaining their precedence.

- Algorithm:

+ Create an empty stack for keeping operators and an empty string list for output. Scan the infix expression from left to right.

+ If the scanned character is an operand, add it to the list for output.

- + If the scanned operator has higher priority than that of the operator in the stack or the stack is empty or contains a (, push it into the stack.
- + Otherwise, pop all the operators in the stack of which precedence is greater than or equal to that of the scanned operator and put them in the list then push the scanned operator to the stack. When you encounter a parenthesis while popping then stop and push the scanned operator in the stack.
- + If the scanned character is (, push it to the stack. If it is ), pop the stack and add the elements to the list for output until a ( is encountered and discard both round brackets.
- + Repeat above steps until infix expression is fully scanned.
- + Finally, pop the operators from the stack and put them in the list one by one until the stack is empty then print the string list.

c) Example:  $A * (B + C) - D$

| Token                              | Action                     | Output        | Stack | Notes                       |
|------------------------------------|----------------------------|---------------|-------|-----------------------------|
| <b>A</b>                           | Add token to output        | A             |       |                             |
| <b>*</b>                           | Push token to stack        | A             | *     |                             |
| <b>(</b>                           | Add token to stack         | A             | * (   |                             |
| <b>B</b>                           | Push token to output       | A B           | * (   |                             |
| <b>+</b>                           | Add token to stack         | A B           | * ( + |                             |
| <b>C</b>                           | Push token to output       | A B C         | * ( + |                             |
| <b>)</b>                           | Pop stack to output        | A B C +       | * (   | Repeated until ( is found   |
|                                    | Pop stack                  | A B C +       | *     | Remove the parentheses      |
| <b>-</b>                           | Pop stack to output        | A B C + *     |       | - has lower priority than * |
|                                    | Push token to stack        | A B C + *     | -     |                             |
| <b>D</b>                           | Push token to output       | A B C + * D   | -     |                             |
|                                    | Pop entire stack to output | A B C + * D - |       |                             |
| <b>Postfix expression: ABC+*D-</b> |                            |               |       |                             |

### 3. Prefix to postfix:

#### a) Programming:

- To convert prefix expression to postfix expression without converting them first to infix and then to postfix, we will reverse the prefix expression and scan, push the operands found to the stack and pop two out when a operator scanned, then create a string by of them and push back to the stack.



- Algorithm:

- + Read the prefix expression in reverse order (from right to left)
- + If the character is an operand, then push it into the stack, otherwise pop two elements from the stack.
- + Create a string by concatenating two elements popped and the operator after them:  
 $string = \langle element1 \rangle + \langle element2 \rangle + \langle operator \rangle$
- + Push the resultant string back to the stack.
- + Repeat the above steps until end of prefix expression.

b) Example:  $* - A / B C - / A K L$

| Token                            | Action            | Stack         | Notes   |
|----------------------------------|-------------------|---------------|---|
| L                                | Push              | L             |   |
| K                                | Push              | L, K          |   |
| A                                | Push              | L, K, A       |   |
| /                                | Pop 2 elements    | L             | Pop 2 operands (A and K). Create a string (AK/) and push it.  |
|                                  | Push AK/          | L, AK/        |   |
| -                                | Pop 2 elements    |               | Pop 2 elements (AK/ and L). Create a string and push it.      |
|                                  | Push AK/L-        | AK/L-         |   |
| C                                | Push              | AK/L-, C      |   |
| B                                | Push              | AK/L-, C, B   |   |
| /                                | Pop 2 elements    | AK/L-         | Pop 2 operands (B and C). Create a string (BC/) and push it.  |
|                                  | Push BC/          | AK/L-, BC/    |   |
| A                                | Push              | AK/L-, BC/, A |   |
| -                                | Pop 2 elements    | AK/L-         | Pop 2 elements (A and BC/). Create a string and push it.      |
|                                  | Push ABC/-        | AK/L-, ABC/-  |   |
| *                                | Pop 2 elements    |               | Pop 2 strings (ABC/- and AK/L-). Create a string and push it. |
|                                  | Push ABC/- AK/L-* | ABC/- AK/L-*  |   |
| Postfix expression: ABC/- AK/L-* |                   |               |   |

### III. CODE FRAGMENT

#### 1. Class diagram

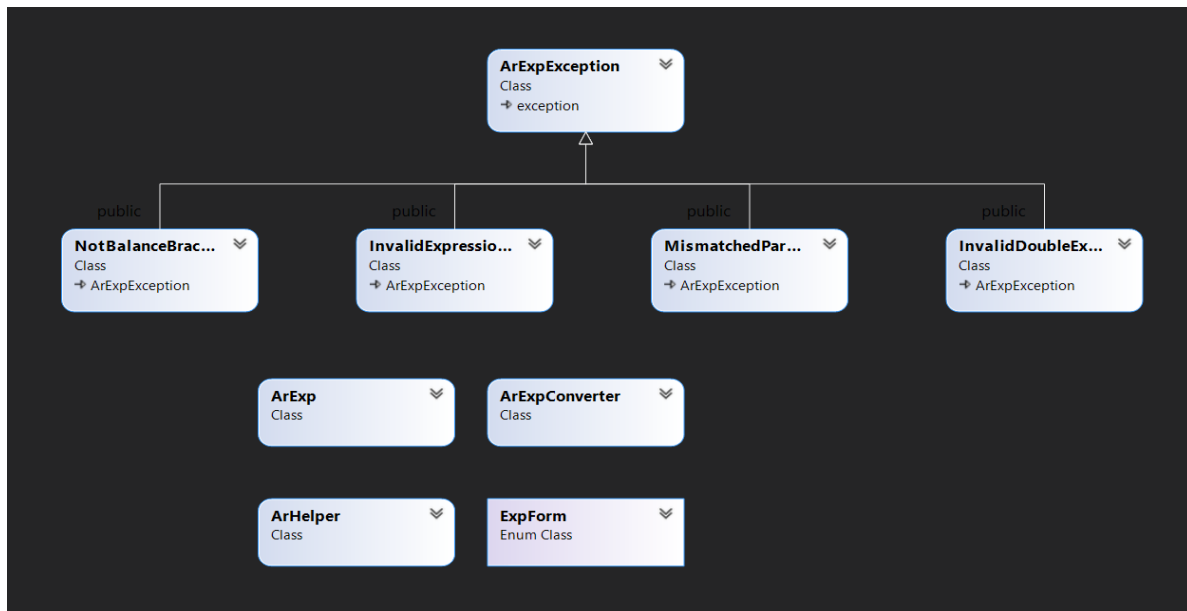


Image 1 Class Diagram

## 2. Class descriptions

Our program is constructed by 4 modules:

### - Exceptions:

- + **ArExpException**: the base class to handle exception for expressions.
- + **NotBalanceBracketExpressionException**: to handle the expression that doesn't have balanced bracket (Invalid expression) ex: { (1 + 2)
- + **InvalidExpressionException**: to handle the invalid infix expression.

Ex: 4 3 3 +, 4 4 3 + -

- + **InvalidDoubleException**: to handle the expression that has more than 2 decimal digits

- + **MismatchedParenthesisException**: to handle the expression that mismatches parenthesis symbols.

Ex: (3 + 3 + (2 + 3

### - ArHelper:

This class provides some utilities and functions to work on string, double variables.

### - Expression representation:

- + **ExpForm**: Enum class keeps type of expression

```
enum class ExpForm { INFIX_EXP, PREFIX_EXP, POSTFIX_EXP };
```

+ **ArExp**: main data structure to save the expression string and expression types. Besides, this class provides some comparison functions to use for unit testing

#### - Core convertor and evaluation:

+ **ArExpConveter**: provides the implementation to convert infix expressions to postfix expression and calculate the infix expressions.

```
ArExp infix_to_postfix()
```

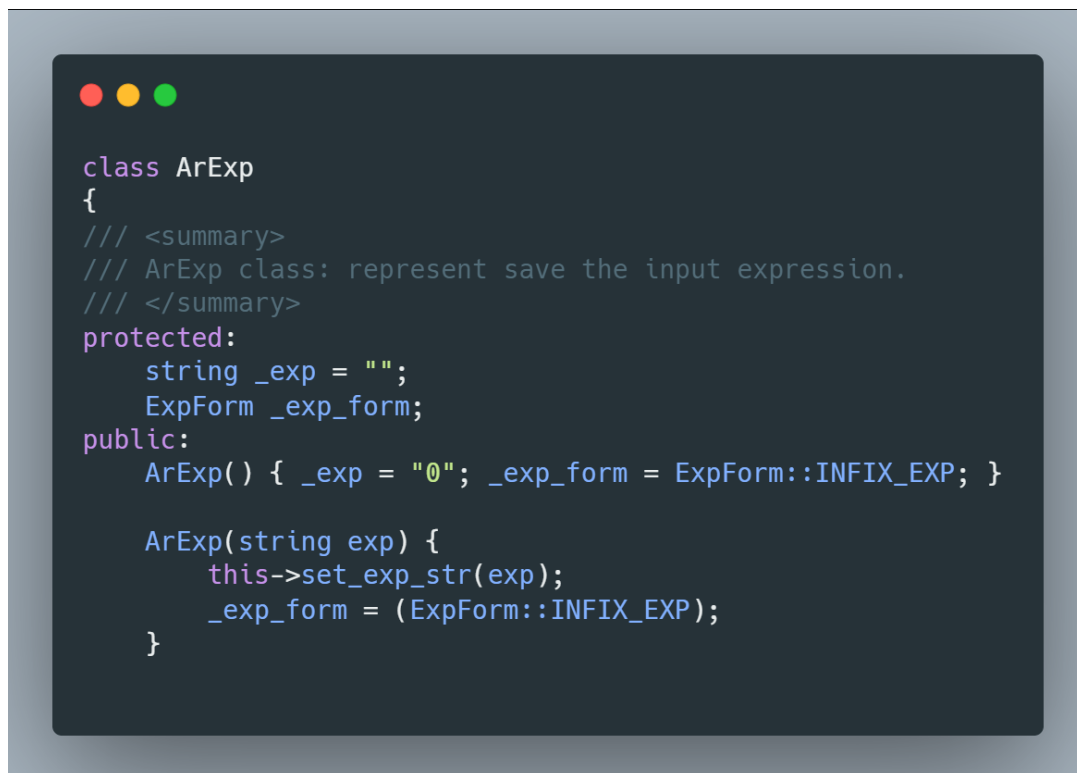
```
double evaluate()
```

#### - User Interface:

This module keeps the main entry point to process the input of users in console environment.

### 3. Implementations

#### \* Represent an infix expression by using ArExp



```
class ArExp
{
    /// <summary>
    /// ArExp class: represent save the input expression.
    /// </summary>
protected:
    string _exp = "";
    ExpForm _exp_form;
public:
    ArExp() { _exp = ""; _exp_form = ExpForm::INFIX_EXP; }

    ArExp(string exp) {
        this->set_exp_str(exp);
        _exp_form = (ExpForm::INFIX_EXP);
    }
}
```

Image 2 Represent ArExp


- Use the overloading constructors to represent a expression. Default of this class will choose **ExpForm::INFIX\_EXP** (Infix expression) as type of expression.

- If need another type, use constructor `ArExp(string exp, ExpForm form)`

**\* Implement infix\_to\_postfix()**

When our group implements the algorithm infix to postfix repersions with the researched algorithm above, we took some problems about brackets, invalid expression, double decimal points. Hence, we have completely process all exceptions

- At first, Check the expressions 's balanced expression and validate the basic rule of infix expression doesn't not allow an operator preceded by another operator (ex 1 +\* 2)



```
ArExp infix_to_postfix() {
    ArExp result;
    string expr = _arexp.to_string();

    if (areBracketsBalanced(expr) == false) {
        throw NotBalanceBracketExpressionException();
    }
    if (basic_validate_infix(expr) == false) {
        throw InvalidExpressionException();
    }
}
```

Image 3 Infix\_to\_postfix preprocess

- Then, change all “{ } [ ]” to “()” for easy to process

```
expr = replace_bracket(expr);
```

- Then, parse all token from expression string by use function

```
string next_token(string str, int& i)
while ((tok = next_token(expr, i)) != "")
```

At this loop, we will process the same with algorithms above. The while loop can be end for two reasons, the stack gets empty, or find the correspondding opening

parenthesis. If the loops end because the stack is empty, the expression has more closing parentheses than opening parentheses => invalid expression (throw an exception)

```
if (S.empty()) throw MismatchedParenthesisException();
```

Moreover, empty token => also throw InvalidExpressionException, too

\* **Implement evaluate()**

- **Step 1:** If **ArExp** is infix expression, call the function `infix_to_postfix()` to convert to postfix expression form.

- **Step 2:** Parse all token from postfix expression and store in a **vector<string> O**



Image 4 evaluate loop token

- **Step 4:** Create a stack `stack<double> evaluate;`

- **Step 5:** Loop over all token “**tok**”

+ If token is operands, push in to a stack **evaluate**

+ If operators, pop two number Val1, Val2 and calculate with operators and push into stack **evaluate** `Push(Evaluate, Val2 <operator> Val1)`

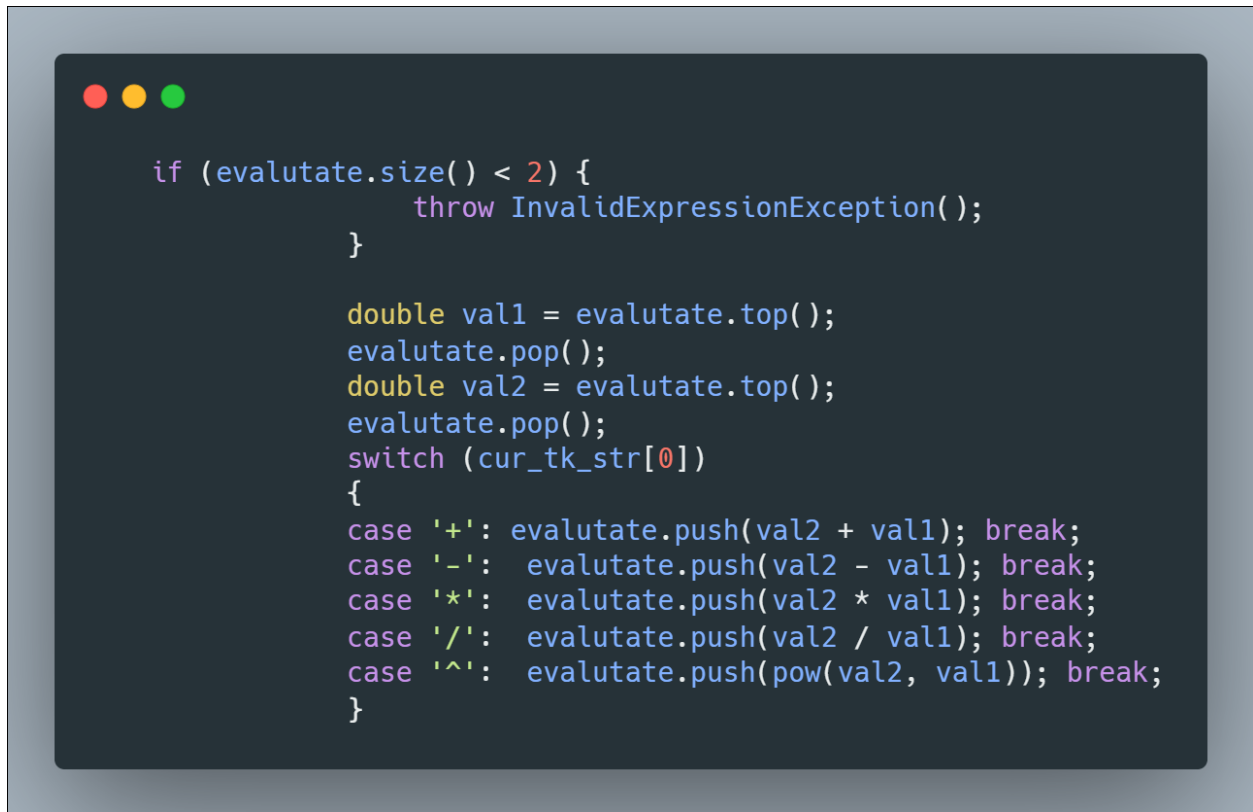


Image 5 Evaluate() handle stack

+ throw **InvalidExpressionException** if it doesn't have enough operands

- **Step 6:** Check and return the result

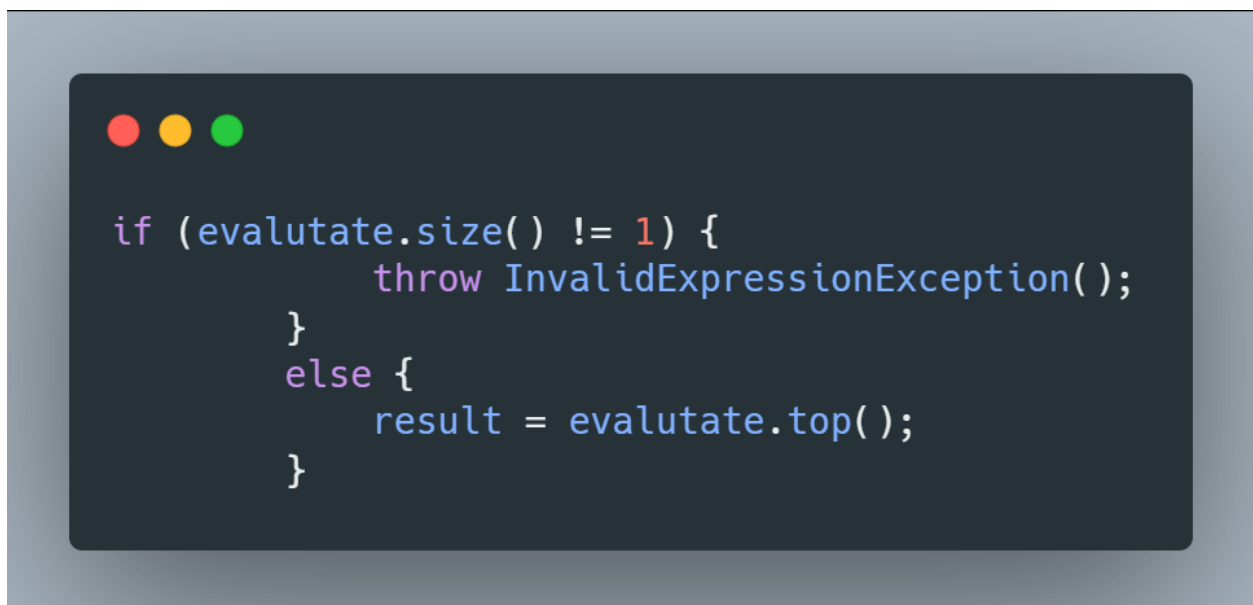


Image 6 Evaluate() return result

#### IV. REFERENCES

1. <https://www.geeksforgeeks.org/prefix-postfix-conversion>
2. <https://runestone.academy/runestone/books/published/pythonds/BasicDS/InfixPrefixandPostfixExpressions.html>
3. <https://www.geeksforgeeks.org/stack-set-2-infix-to-postfix>
4. <https://www.geeksforgeeks.org/convert-infix-prefix-notation>
5. <https://stackoverflow.com/questions/31138509/how-to-find-wrong-infix-expression>