**VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY**

**UNIVERSITY OF TECHNOLOGY**

🙞···☼···🙜



**FACULTY OF COMPUTER SCIENCE AND ENGINEERING**

**COURSE: COMPUTER ARCHITECTURE LAB (CO2008)**

**ASSIGNMENT REPORT  
CALCULATOR**

**Class: CC02 – Semester 232**

***Instructor:* NGUYỄN THIÊN ÂN**

**Student: Phạm Nguyễn Minh Hiếu**

**ID: 2252215**

*Ho Chi Minh City – 2024*

**TABLE OF CONTENTS:**

[**I.** **BACKGROUND:** 3](#_Toc166023338)

[**1.** **Introduce** 3](#_Toc166023339)

[**2.** **Theory of Infix expression** 3](#_Toc166023340)

[**3.** **Theory of Postfix expression** 5](#_Toc166023341)

[**4.** **Theory of Stack** 6](#_Toc166023342)

[**II.** **OVERVIEW ABOUT ALGORITHM** 7](#_Toc166023343)

[**III.** **MORE DETAILS ON ALGORITHM PROCESSING** 9](#_Toc166023344)

[**Function block 1:** 9](#_Toc166023345)

[**Function block 2** 12](#_Toc166023346)

[**Function block 3** 15](#_Toc166023347)

[**IV.** **TEST RUN OF PROGRAM** 20](#_Toc166023348)

[**V.** **SOME NOTES FOR USERS** 22](#_Toc166023349)

[**REFERENCE** 23](#_Toc166023350)

# 

# 

# **BACKGROUND:**

## **Introduce**

Mathematical formulas often involve complex expressions that require a clear understanding of the order of operations. To represent these expressions, we use different notations, each with its own advantages and disadvantages. In this article, we will explore three common expression notations: infix, prefix, and postfix.

In this assignment, we will mention infix and postfix, let's talk about them now.

## **Theory of Infix expression**

Infix expressions are mathematical expressions where the operator is placed between its operands. This is the most common mathematical notation used by humans.

*For example*: the expression “” is an infix expression, where the operator “+” is placed between the operands “2” and “3”.

Infix notation is easy to read and understand for humans, but it can be difficult for computers to evaluate efficiently. This is because the order of operations must be taken into account, and parentheses can be used to override the default order of operations.

Common way of writing Infix expressions:

Infix notation is the notation that we are most familiar with. For example, the expression “” is written in infix notation.

In infix notation, operators are placed between the operands they operate on. For example, in the expression “”, the addition operator “+” is placed between the operands “2” and “3”.

Parentheses are used in infix notation to specify the order in which operations should be performed. For example, in the expression “”, the parentheses indicate that the addition operation should be performed before the multiplication operation.

***Operator precedence rules:***

Infix expressions follow operator precedence rules, which determine the order in which operators are evaluated. For example, multiplication and division have higher precedence than addition and subtraction. This means that in the expression “”, the multiplication operation will be performed before the addition operation.

Here’s the table summarizing the operator precedence rules for common mathematical operators:

|  |  |
| --- | --- |
| Operator | Precedence |
| Parentheses () | Highest |
| Exponents ^ | High |
| Multiplication \* | Medium |
| Division / | Medium |
| Addition + | Low |
| Subtraction - | Low |

***Advantages of Infix Expressions:***

* More natural and easier to read and understand for humans.
* Widely used and supported by most programming languages and calculators.

***Disadvantages Infix Expressions:***

* Requires parentheses to specify the order of operations.
* Can be difficult to parse and evaluate efficiently.

## **Theory of Postfix expression**

Postfix expressions are also known as Reverse Polish Notation *(RPN)*, are a mathematical notation where the operator follows its operands. This differs from the more common infix notation, where the operator is placed between its operands.

In postfix notation, operands are written first, followed by the operator. For example, the infix expression “” would be written as “5 2 +” in postfix notation.

***Advantages of Postfix Notation:***

Also eliminates the need for parentheses.

Easier to read and understand for humans.

More commonly used than prefix notation.

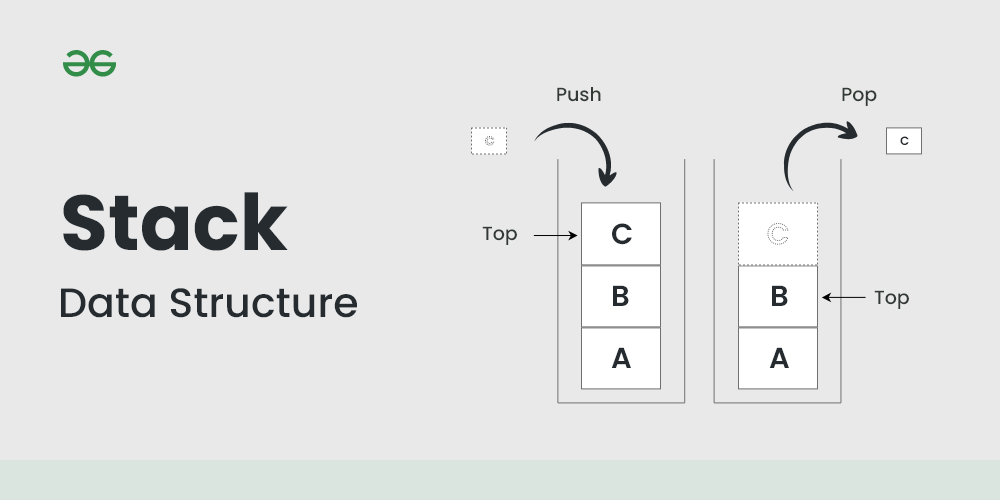
***Disadvantages of Postfix Expressions:***

Requires a stack-based algorithm for evaluation.

Can be less efficient than prefix notation in certain situations.

## **Theory of Stack**

A Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.



# **OVERVIEW ABOUT ALGORITHM**

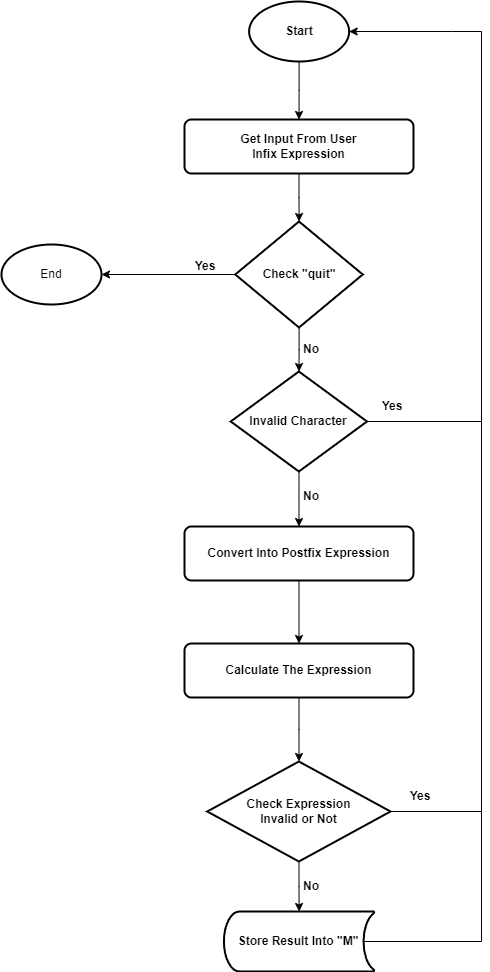
*In this exercise*, we will create a simple computer with simple calculation functions. The algorithm idea for mathematical expressions after being entered from the user is to convert the original expression - infix expression to a postfix expression.

After converting from infix expression to postfix expression, we will proceed to calculate from the expression in postfix. This process involves pulling out characters, converting them to real numbers, and calculating. Finally, the result will be saved in variable M for further calculations. This process will be repeated until the user enters the "quit" command line.

*In more detail in checking*, the algorithm will first check for "Valid" characters. If they are invalid, the program will ask the user to re-enter the expression. Otherwise, the program will continue to compile until the second round of testing. In this round, the algorithm will check the reasonableness in the order of expressions. If the expression is reasonable in the order, it will be calculated and saved in value “M”, otherwise, the program will ask the user to re-enter a new expression.

*Regarding writing to the file,* all input expressions as well as the results of those expressions into the “calc\_log\_txt” file, the program will write all user input expressions case, including incorrect character entries, incorrect math expressions errors due to the order of calculations, exiting the program “quit”, and the results of each calculation.

This is the flowchart about the overall algorithm:



# **MORE DETAILS ON ALGORITHM PROCESSING**

To facilitate a better understanding of how the program actually works, the functions of each block in the program, as well as special cases in algorithm processing, we will go deeper into the blocks in the program.

***The program will operate in 3 main blocks:***

*Function block 1:* Convert expression from infix to postfix.

*Function block 2:* Calculate from transformed expression.

*Function block 3:* Convert results to string format to write to file.

## **Function block 1:**

***Step 1:*** Read the input expression and save it into an array with a maximum size of 100bytes.

***Step 2:*** Check the character sequence, if the input sequence is "quit", the program will end, otherwise the program will continue.

***Step 3:*** After having the input expression, we will use a loop to convert the expression to postfix form.

During the process of iterating to each character, the program will simultaneously check the validity of the character, as well as the validity of the order of mathematical expressions. The program will ask the user to re-enter if such cases occur, otherwise it will continue to complete the conversion.

*Note:* Considered valid characters: "0 1 2 3 4 5 6 7 8 9 + - \* / . M ˆ ! ( )"

Some math expressions are considered errors: ""; ""; ""; ""; “”; etc.

***Step 4:*** Proceed to set the numbers and characters in postfix format. In this step, the program will perform the conversion. And in “Postfix\_stack”, we will add a “space” between operand – operand or operand – operator to mark for easy calculation.

* "Postfix\_stack" will be the array containing the expression after conversion.
* "operator\_stack" will be the array containing the operators.

Scan the symbols of the expression from left to right and for each symbol, do the following:

1. ***If symbol is an operand or a dot***

* Put that symbol into “Posfix\_stack”

1. If symbol is a left parenthesis

* Push it on the “operator\_stack”

1. ***If symbol is a right parenthesis***

* Pop all the operators from the stack upto the first left parenthesis and put it into “Postfix\_stack”
* Discard the left and right parentheses

1. ***If symbol is an operator***

* If the precedence of the operator in the stack are greater than or equal to the current operator, then

Pop the operators out of the stack and put them onto the

“Postfix\_stack”, and push the current operator onto the stack.

* else

Push the current operator onto the “operator\_stack”.

* A special operator is “!”, I will consider it like a number, so I will add it onto “Postfix\_stack” without checking precedence.

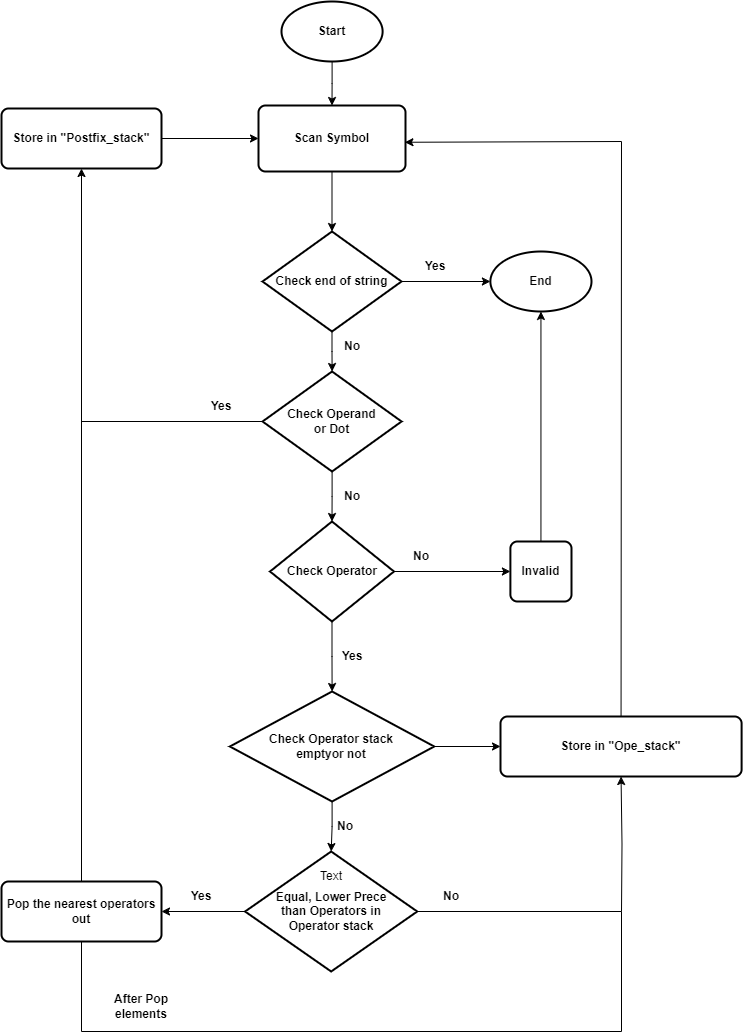
*In addition*, I also create an array for saving status of factorization. As you know, the expression “(-4)!” and “-4!” is different, one is error, another is negative result. So the array will assign 1 to the expression which have open parenthesis before for checking negative number when calculate.

I also created a few functions to handle continuous plus - minus signs, such as

will become ;

“ will become , etc.

*Flowchart algorithm for function block 1: Convert into Postfix expression*

**

## **Function block 2**

In the second function block, we will calculate the results based on the postfix expression after completing function block 1. In this block, we will create a new stack named “calculation” for saving the real value converted from string in “Postfix\_stack”.

***Step 1:*** Scan symbol from left to right. The “space” will distinguish elements from each other.

***Step 2:*** Check symbol

If they are numbers, convert them to real numbers and save them to the “calculation” stack. The conversion process is performed as follows.

* If the sign is negative, we proceed to mark it with a register to finally do the "zero - expression" and store into “calculation”.
* *If it is an integer,* for the first digit, we convert it to a real number by adding 48 according to the ASCII code table, then we take "" and add it to that number. From the 2nd digit onwards, we continue to convert through the ASCII code table, multiply the previously found number by 10 and add it to the current number. Finally, check the negative number and save the results to stack calculation.

*For example:* and . Positive number with 3 steps, negative number with 4 steps.

|  |  |  |
| --- | --- | --- |
| **Step** | **Expression** | **Result** |
| 1 |  | 1 |
| 2 |  | 12 |
| 3 |  | 123 |
| 4 |  | -123 |

* *If it is a decimal number,* we do the same for the integer part as for integers number. But in the decimal part, instead of multiplying by 10 like in the integer part, for the first decimal number we will divide the current number by 10 and add it with the integer part. From the 2nd decimal number onwards, we will divide by , and at the nth decimal number, we will divide by and repeat the addition as above. Finally, check the negative number and save the results to stack calculation.

*For example:*  and Positive number with 5 steps, negative number with 6 steps.

|  |  |  |
| --- | --- | --- |
| **Step** | **Expression** | **Result** |
| 1 |  | 1 |
| 2 |  | 12 |
| 3 |  | 123 |
| 4 |  | 123.4 |
| 5 |  | 123.45 |
| 6 |  | -123.45 |

* *If they are operators,* take the 2 closest elements in the stack calculation to perform the calculation by calculation functions. Then, save the newly received result back to the stack calculation.

*For example: 2.34 43 +*

|  |  |
| --- | --- |
| **Step** | **Explain** |
| 1 |  |
| 2 |  |
| 3 |  |

* *For operator "!",* we will only take the nearest 1 element to calculate. Before calculating, we will check whether the expression satisfies or not, as mentioned above, for example "-4!" would be satisfied but "(-4)!" it's the opposite. To verify this, we will check whether the mark is 1 or 0 in the register used for the mark. Then, the newly received results will be saved into the stack calculation.

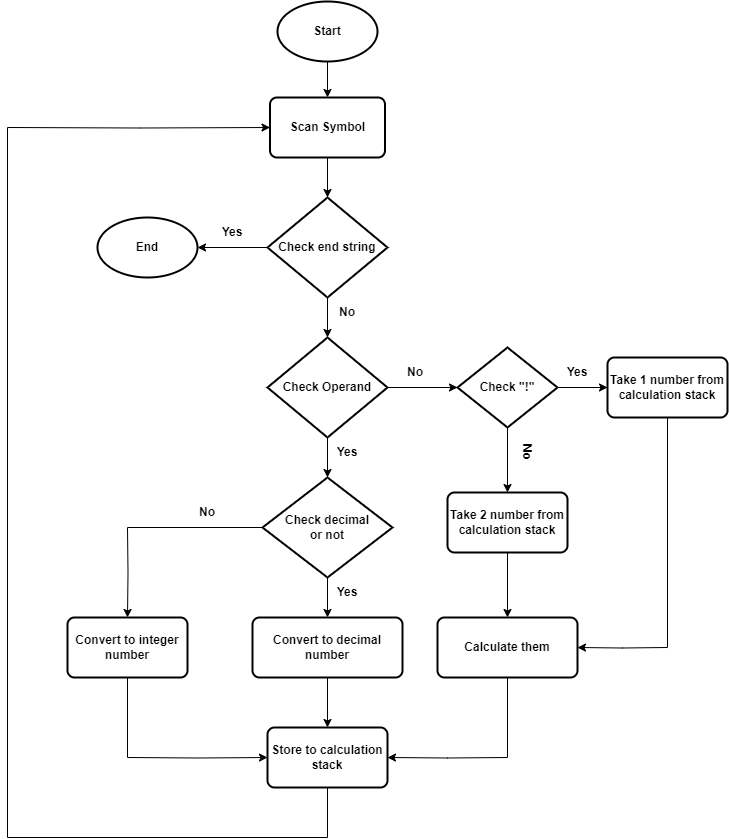
*For example: 3!*

|  |  |
| --- | --- |
| **Step** | **Explain** |
| 1 |  |
| 2 |  |

***Step 3:*** Save the final result to value "M".

***Note:*** We check a number is a decimal number or not by convert it into integer, the program will round it to the integer part. After that, we subtract the initial number with the number has just been converted. It the result is zero, it means the initial number is in integer format, otherwise, it is in decimal number format.

*Flowchart algorithm for function block 2: Calculate from transformed expression.*

**

## **Function block 3**

In function block 3, after obtaining the value of M, I converted the M value to string and wrote it to the “calc\_log.txt” file, and at the same time output the M value to the screen.

***Step 1:*** Screen the M value to the screen.

***Step 2:*** Proceed to convert the format of M

* *First,* check whether M is negative or not. If M is negative, save the minus sign to write to the file.
* *Then,* check if M is an integer or a decimal number, the way to check is as mentioned in function block 2. After checking, if it is a decimal number, we use subtraction between the original number and the rounded number. Store the subtraction value into the register (this is decimal part).

*For example: 12.345*

|  |  |  |  |
| --- | --- | --- | --- |
| **Step** | **Explain** | **Expression** | **Result** |
| 1 | Convert into integer |  | 12 |
| 2 | Subtract |  | 0.345 |
| 3 | Save the integer part |  | 12 |
| 4 | Save the decimal part |  | 0.345 |

* *Proceed to save the integer part:* Take the integer part and divide it by 10, the remainder will be saved in the "to\_write\_1" array, the result will be used to divide by 10. If we get the result as 0, the loop end.
* *Then,* take the elements in "to\_write\_1" and save them in "to\_write\_2" in reverse order.

*For example:* Integer = 12. After the first division, we get a remainder of 2, the result is 1. After division by 2, we get a remainder of 1, the result is 0. The "to\_write\_1" array will now be is "2, 1". We have to change the position to "1, 2" to be satisfied.

|  |  |  |  |
| --- | --- | --- | --- |
| **Step** | **Expression** | **Remainder** | **Result** |
| 1 | 12/10 | 2 | 1 |
| 2 | 1/10 | 1 | 0 = end |
| 3 | to\_write\_1 = (21) |  |  |
| 4 | to\_write\_2=(12 |  |  |

* *Proceed to save the decimal part:* First, we save the “.” sign in "to\_write\_2".
* *Then,* multiply the result of the first subtraction by 10, we will get a new number, round and subtract to get the next decimal part. The number after rounding will be converted into characters according to the ASCII code table and saved directly to "to\_write\_2", the process continues until there are 16 characters in the decimal part.

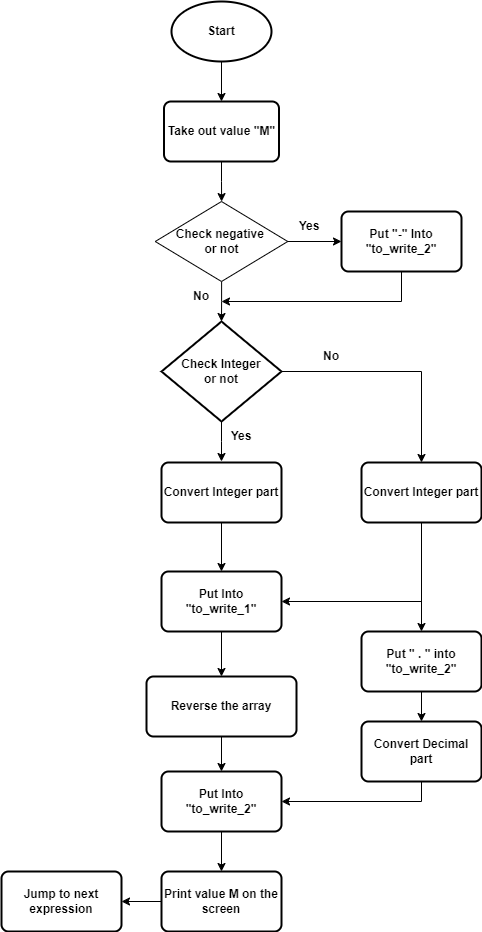
*For example:* Decimal = 0.345. The result is the number in Convert column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Step** | **Expression** | **Result 1** | **Convert** | **Subtract** | **Result 2** |
| 1 |  | 3.45 | 3 |  | 0.45 |
| 2 | 0.45\*10 | 4.5 | 4 |  | 0.5 |
| 3 | 0.5\*10 | 5 | 5 | 0 | 0 |
| 4 | to\_write\_2=(.345) |  |  |  |  |

***Step 3:*** Write "to\_write\_2" into "calc\_log.txt"

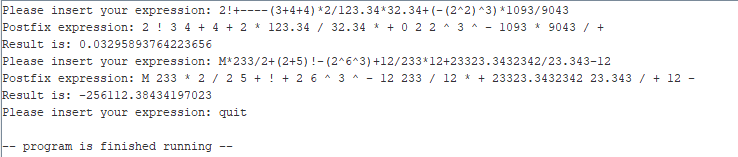
***Note:*** During the conversion process, it is necessary to create a variable to count the number of characters to avoid font errors during saving.

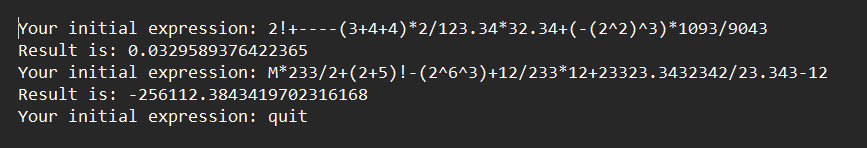
*Flowchart algorithm for function block 3: Convert results to string format to write to file.*

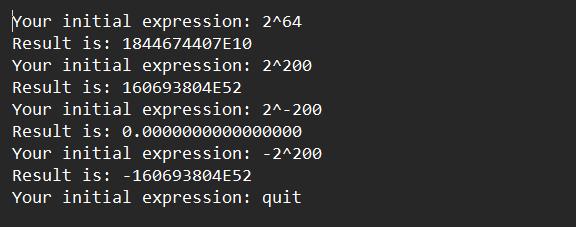
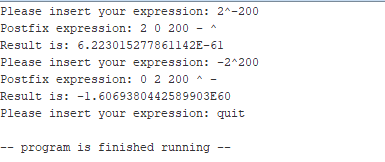
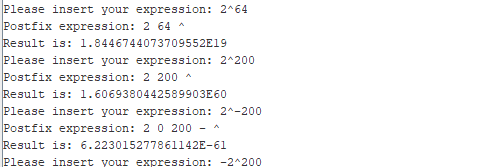


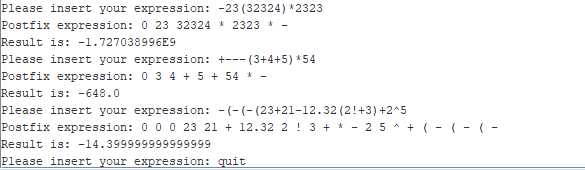
# **TEST RUN OF PROGRAM**

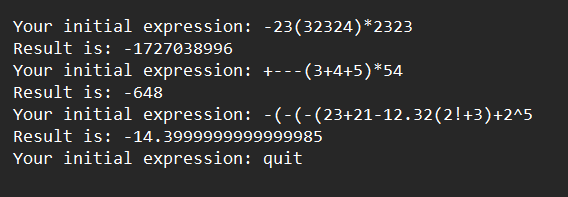
Some test I have try with this program, I will screen the postfix expression onto screen for easier checking.

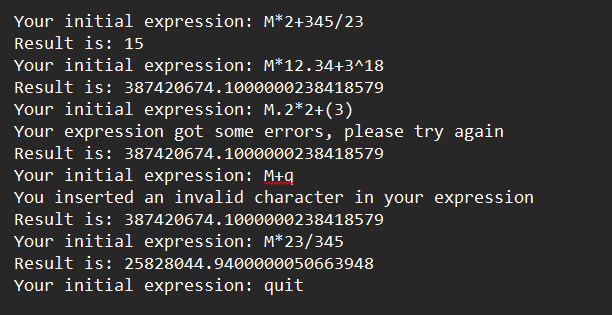
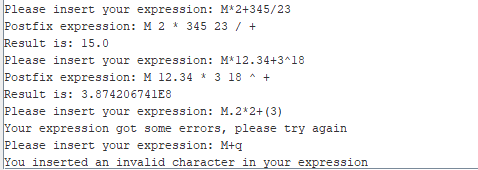












# **SOME NOTES FOR USERS**

Although the program is designed with diverse cases such as:

or

or

or , etc.

But we encourage users to enter the necessary expressions correctly and in the appropriate order because errors may still occur during the calculation process.

Pay special attention to the opening-closing parentheses in expressions.

For example, if you want to calculate: in computer, you have to write 2^(2^3) in MIPS.

# **REFERENCE**

1. *Infix, Postfix and Prefix Expressions/Notations, 21 Mar, 2024*

<https://www.geeksforgeeks.org/infix-postfix-prefix-notation/>

1. *Stack Data Structure, 02 May, 2024* <https://www.geeksforgeeks.org/stack-data-structure/>
2. *Neso Academy - Application of Stacks (Infix to Postfix)* [*https://www.youtube.com/@nesoacademy*](https://www.youtube.com/@nesoacademy)