12/26/2024

TROY UNIVERSITY

**CS 2250**

**Computer Science I**

**Fall 2024**



**Memebers**

Hoang van hiep - 1690518

Ly Ba Hoang - 1695172

Pham Thanh Tung - 1694591

**TUTOR:**

PH. d BUI THI MAI ANH

FINAL PROJECT CS 2550

Group 1

# Table of contents

[Table of contents 2](#_Toc186101182)

[A. Criteria: 3](#_Toc186101183)

[B. Topic: Prefix Expression 3](#_Toc186101184)

[C. Problem analysis 4](#_Toc186101185)

[1. What is infix notation? 4](#_Toc186101186)

[2. What is prefix notation? 4](#_Toc186101187)

[D. Program design 5](#_Toc186101188)

[1. Program description: 5](#_Toc186101189)

[2. Functions (Modules) of the program: 5](#_Toc186101190)

[3. The pseudocode of the whole program: 6](#_Toc186101191)

[E. The infixValidationAndTokenize function design and documentation 7](#_Toc186101192)

[1. Fundamental basics to validate an infix expression: 7](#_Toc186101193)

[2. The algorithm of the function: 8](#_Toc186101194)

[3. Modules used in the function algorithm 10](#_Toc186101195)

[F. The infixToPrefix Function design and documentation 16](#_Toc186101196)

[1. Algorithm for the conversion of infix to prefix expression: 16](#_Toc186101197)

[2. The pseudocode of the function 17](#_Toc186101198)

[G. Prefix calculation desgin 17](#_Toc186101199)

[H. Some test cases and demo examples of the program 19](#_Toc186101200)

GROUP 1 - PROJECT - CS 255

TOPIC 2: INFIX TO PREFIX

# Criteria:

1. The program runs correctly, has all the required functions

2. The program source code is presented beautifully. The comments are complete and clear.

3. The program is capable of handling cases where the input data is not standardized or has errors.

4. Tests have been conducted to ensure the program runs stably.

5. The work is divided reasonably within the group.

6. Submit the major assignment: Submit to Teams – Assignment MiniProject: analysis and installation report + source code (.zip)

# Topic: Prefix Expression

**Prefix Expression in Programming**

A prefix expression is a syntax in mathematical representation where the operators are placed before their operands.

For example: If the infix notation of a mathematical expression like (a + b) \* (c + d)

The prefix notation is going to place operators in front of the operands, the prefix notation of the above infix expression is \* + a b + c d

**Program Requirements:**

Create a program that performs the following tasks:

1. **Input Validation and Syntax Checking**  
   Allow the user to input a string representing a mathematical expression in infix form. The operands will be integers (including negative numbers).  
   Example: (12 + 10) \* (17 – 8)  
   If the input has incorrect syntax, such as (12 10) – 7, the program should notify the user about the syntax error.
2. **Convert Infix to Prefix**  
   Convert the infix expression to its corresponding prefix form.
3. **Evaluate the Prefix Expression**  
   Compute the result of the prefix expression.
4. **Menu Design**  
   The program should include a menu with the following options:
   * **Enter an Infix Expression**:  
     The program reads and validates the infix expression. If there’s a syntax error, it should notify the user and display the menu again. If the expression is valid, it should display the equivalent prefix expression and the menu for further actions.
   * **Evaluate the Prefix Expression**:  
     The program calculates the value of the prefix expression. If no infix expression has been entered yet, it should prompt the user to input an infix expression first and return to the menu.
   * **Exit**:  
     Exit the program.

# Problem analysis

## What is infix notation?

Operators are written in-between their operands. This is the usual way we write expressions. An expression such as

A \* ( B + C ) / D

is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets ( ) to allow users to override these rules. For example, the usual rules for associativity say that we perform operations from left to right, so the multiplication by A is assumed to come before the division by D. Similarly, the usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction.

## What is prefix notation?

Operators are written before their operands. The expressions given above are equivalent to

/ \* A + B C D

The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order.

Operators use values to their right, and if these values themselves involve other computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication).

Operators act on the two values immediately to the right of them. For example, the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit:

(/ (\* A (+ B C)) D)

Let's evaluate the above expression in more detail: / \* A + B C D

Step 1: Starting with the operator ‘ / ’

Step 2: Notice the next operator is ‘ \* ‘

Step 3: Notice the last operator is ‘ + ‘ → evaluate the expression + B C = B + C first

Step 4: Now, evaluate the expression \* A + B C = A \* (B + C)

Step 5: Evaluate the expression / \* A + B C D = A \* (B + C) / D

Some examples:

|  |  |
| --- | --- |
| **Infix notation** | **Prefix notation** |
| A + B \* C | + A \* B C |
| (A + B \* C ) / ( D - E ) | / + A \* B C - D E |
| ( A + B ) \* ( C - D ) | \* + A B - C D |

# Program design

## Program description:

1. Display the Menu with 3 options
2. If the user chooses option 1
   * Take the infix expression
   * Check if the infix expression is valid or not
     + If valid, display the corresponding prefix expression, and return to part a.
     + If invalid, display the error message and return to part a.
3. If the user chooses option 2:
   * If there is already an infix expression, then calculate and display the value of the infix expression, and return to part a.
   * If there is no infix expression, display a message to inform the user of entering an infix expression, return to part a.
4. If the user chooses option 3: Exit the program.

## Functions (Modules) of the program:

* The **displayMenu** function (Hoang Van Hiep): The function only displays 3 options of the menu including (i) Enter an infix expression (ii) Calculate the value of the infix expression (iii) Exit the program. The function should take no parameters and return no value (a void function).
* The **infixValidationAndTokenize** function (Ly Ba Hoang): The function takes the infix expression entered by the user, and then checks if the infix expression is valid or invalid. If the expression is valid, the function returns a ‘true’ value, otherwise, returns a ‘false’ value. At the same time, the function also processes the infix expression to split it into tokens (operands, operators, and parentheses) and puts them into a vector, which is then used in the function infixToPrefix to convert infix into prefix.
* The **infixToPrefix** function (Pham Thanh Tung): The function takes the tokens vector above and converts the infix into prefix. It stores the tokens of the prefix into a vector in the main function. The vector of prefix tokens then is used to display the prefix to the screen, and used in the prefix evaluation.
* The **prefixCalculation** Function (Pham Thanh Tung): The function takes the prefix vector, which contains tokens of the prefix expression, and calculates the prefix expression, then returns the result to the main function.

## The pseudocode of the whole program:

|  |  |
| --- | --- |
| Step 1 | Declare a vector of string to hold the tokens of the infix expression |
| Step 2 | Declare a vector of prefix to hold the tokens of the prefix expression |
| Step 3 | Display the introduction of the program |
| **Step 4** | **Do** |
| Step 5 | Display the menu with 3 options |
| Step 6 | Prompt for user’s option |
| Step 7 | If (user’s choice is invalid) then |
| Step 8 | Display an error message and and go back to step 4 |
| Step 9 | Else if (user’s choice is 1 - Enter an infix expression) then |
| Step 10 | Clear the two vectors of the previous infix expression |
| Step 11 | Get the infix expression from user |
| Step 12 | Call the function **infixValidationAndTokenize** to check the input |
| Step 13 | If (the infix expression is not valid) then |
| Step 14 | Display error message and go back to Step 4 |
| Step 15 | Else |
| Step 16 | Call the function **infixToPrefix** to convert |
| Step 17 | Display the prefix expression and go back to Step 4 |
| Step 18 | Else if (user’s choice is 2 - Calculate the infix expression) |
| Step 19 | If (there is no prefix expression, or vector prefix is empty) then |
| Step 20 | Display an error message to indicate there is no prefix |
| Step 21 | Else |
| Step 22 | Call the function **prefixCalculation** to calculate the prefix |
| Step 23 | Display the calculation result and go back to Step 4 |
| Step 24 | Clear the two vectors of the previous infix expression |
| Step 25 | While (the user’s choice is not “3 - EXIT” or invalid) |

# The infixValidationAndTokenize function design and documentation

## Fundamental basics to validate an infix expression:

* 1. **Parentheses matching rules:** 
     + If there is an open parenthesis, then it must have a matching close parenthesis.

Example: Wrong infix expression:

(2 + 5) / 7); ((3 + 10) – (4 + 5);…

* + - Open parentheses cannot appear at the end of an expression. For example, some invalid infix expressions:

2 + 5 ((; 3 \* (5 + 6) (

* + - Close parentheses cannot appear at the beginning of an expression. For example, some invalid infix expressions:

) 2 + 5; ) 9 / (4 – 1)

* + - An open parenthesis cannot follow right after a close parenthesis ) (, and a close parenthesis also cannot follow right after an open parenthesis ( ).
  1. **Operators rules:**
     + If there is a sequence of of plus and minus signs, the function also handles it into a single plus sign or single minus sign. For example:

+---+ = -; ++++ = +;…

* + - Two or more operators cannot be consecutive in an expression to avoid ambiguous. For example, some ambiguous infix expressions:

3 +/ 5; 9 \* \* (2 + 3); …

* + - Operators cannot appear at the beginning and at the end of an expression. For example, some invalid infix expressions:

\* 4 + 5; / 15 / 5; 20 + 4 \*; (100 + 3) / 3 + 1 –;….

* + - Operators cannot appear right after an open parenthesis. For example, some invalid infix expressions:

3 \* (+ 6 + 6);….

* 1. **Operands rules:**
     + For a negative number, if it appears at the beginning of the expression, then it is not necessary to enclose the number with a pair of parentheses; however, if the number appears inside the expression, it must be enclosed by a pair of parentheses. If not, the negative sign will be interpreted as an operator. For example:

-3 + 5 \* (-2); (-9) \* 10 + (-5); 9 --- -5 = 9 + 5;….

* + - Between two operands must have an operator, cannot be a parenthesis. For example, some invalid infix expressions:

(12 10) / 3; (80 + 100) 20;…

* + - Operands cannot immediately follows a close parenthesis. For example, some invalid infix expressions:

12 / (3 – 2) 3; 100 \* (3 + 3) / (2 + 2) 9;…

* + - Whitespaces in an infix expression may not make an expression to be wrong. For example, some valid infix expressions:

12/ 3: There is a whitespace between the operator / and operand 3

(2 +3)/ 2: There are whitespaces between 2 and ‘+’ and between ‘/’ and 2

* 1. **Convention of negative integers:**
     + If the negative sign represents a negative number, then it must appear right after the number without any whitespaces in between. For example:

Valid cases: -9; -10;…

Other cases: - 9; - 10;… The minus sign is going to be intepreted as a minus operator.

* + - If the negative integers appear at somewhere inside the expression, then it must be enclosed by a pair of parentheses and also to avoid more clear and not ambiguous. Otherwise, it is going to be intepreted as a minus operator. For example:

Valid cases: 10 + (-9); (25 / (-5)) + 2;…

Invalid cases: 10 + -9; (25 / -5) + 2;…

* + - If a number with multiple plus and minus signs are enclosed by a pair of parenthesis, then these multiple plus and minus signs will be intepreted as the sign of the number. For example:

10 + (-+-- 9) = 10 + (-9); 10 + (-+---9) = 10 + (+9) = 10 + 9;…

## The algorithm of the function:

**Firstly,** the function checks whether the infix expression is empty, or contains only whitespace characters (spaces or tabs) by using a function named **isEmptyExpression**. If it is, then the function display an error message to indicate an empty string and return false to the main function.

**Secondly,** the function checks whether the infix expression contains two consecutive operands without any operators in between by using a function named **areConsecutiveOperands**. If it is, the function display an error message to indicate that there exists two consecutive operands and return false to the main function.

**Thirdly,** the function removes all spaces in the infix expression and store each character in the expression as an individual element into a vector by using a function named **removeSpaces**.

**Next,** the function handles all sequences of plus and minus signs to convert it into a single plus or a single minus sign, and store the character into a new vector containing all processed characters (do not contain whitespace characters and sequence of plus and minus signs) by using a function named **simplifySigns**.

**Then,** the function runs a loop through the vector containing characters (after removing spaces and simplify squences of plus and minus signs) in the infix expression to validate. For each iteration, the current character is validated based its right previous character. If the current character is valid, then it is stored as the previous character, push it into another vector of string as tokens or parses of the infix expression, and the function moves to the next iteration. The vector of tokens will be then used in the process of infix-to-prefix conversion.

**Finally,** when all characters of the infix expression are validated, the function also needs to check two more things to ensure that the expression is valid. Firstly, the function must check whether the number of open parentheses is equal to zero. If it is not, then the function displays an error message to indicate unmatched parentheses and returns false to the main function. Secondly, if the last character (or the previous character) is an operator, or an open parentheses, the function displays an error message and returns false to the main function.

**Consequently,** the result of the function is that it successfully **validates the infix expression** and **returns a vector of string containing tokens** (parentheses, operands, operators) in order, which then is used to convert the infix expression into prefix expression.

Here is the infixValidationAndTokenize function’s pseudocode:

|  |  |
| --- | --- |
| Step 1 | Inititalize previous character to a null character |
| Step 2 | Inititalize the number of open parentheses to 0 |
| Step 3 | Initialize a vector of character named rawCharacters to hold non-whitespace characters |
| Step 4 | Declara a boolean variable to hold the checking result |
| Step 5 | If (**isEmptyExpression**) then |
| Step 6 | Display error message and return false |
| Step 7 | If (**areConsecutiveOperands**) there exists two consecutive operands |
| Step 8 | Display error message and return false |
| Step 9 | Remove all whitespace characters and store each character into the rawCharacters by calling the function **removeSpaces** |
| Step 10 | Simplify sequences of plus and minus signs by calling the function **simplifySigns** |
| **Step 11** | **A loop iterate through each element of the rawCharacters vector** |
| Step 11.1 | If (**isOpeningParethesis**(current character)) the current character is an open aprenthesis |
| Step 11.2 | Call the function **isOpenParenthesisValid** |
| Step 11.3 | If (result is false) |
| Step 11.4 | Return false to the main function |
| Step 11.5 | Else if (**isClosingParenthesis**(current character)) the current character is a close parenthesis |
| Step 11.6 | Call the function **isCloseParenthesisValid** |
| Step 11.7 | If (result is false) |
| Step 11.8 | Return false to the main function |
| Step 11.9 | Else if (**isOperator**(current character)) the current character is an operator |
| Step 11.10 | Call the function **isOperatorValid** |
| Step 11.11 | If (result is false) |
| Step 11.12 | Return false to the main function |
| Step 11.13 | Else if (**isdigit**(current character)) the current character is a digit |
| Step 11.14 | Call the function **isDigitValid** |
| Step 11.15 | If (result is false) |
| Step 11.16 | Return false to the main function |
| Step 11.17 | Else |
| Step 11.18 | Display error message of invalid character |
| Step 11.19 | Return false to the main function |
| Step 12 | If the number of open parentheses is greater than 0 |
| Step 13 | Display error message and return false to the main function |
| Step 14 | If the last character is an open parenthesis |
| Step 15 | Display an error message and return false to the main function |
| Step 16 | Return true to the main function |

## Modules used in the function algorithm

The function uses 7 functions as shown below:

1. **isEmptyExpression(string):** This is a boolean function. The function takes one parameter which is the string of the infix expression, and checks whether the expression is empty or only contains whitespace characters (spaces or tabs). If the expression is empty or only contains whitespace characters, the function returns true; otherwise returns false to the infixValidationAndTokenize function.

Here is the function’s psedocode:

|  |  |
| --- | --- |
| Step 1 | If the string is empty (using empty() built-in function) then |
| Step 2 | Display an error message and return true |
| Step 3 | A loop to iterate through each character in the expression |
| Step 4 | If there is a character is different from spaces and tabs |
| Step 5 | Return false and break the loop |
| Step 6 | Return true |

1. **areConsecutiveOperands(string):** This is a bool function. The function takes one parameter, which is the string of the infix expression, and then checks whether the expression contains two consecutive operands without an operator in between. If it is, the function returns true; otherwise the function returns false to the infixValidationAndTokenize function.

Here is the function’s pseudocode:

|  |  |
| --- | --- |
| Step 1 | Inititalize and set the flag **isPreviousDigit** and **isPreviousSpace** to false |
| Step 2 | A loop to iterate through each character in the expression |
| Step 3 | If the character is a whitespace character |
| Step 4 | Set the flag **isPreviousSpace** to true |
| Step 5 | Else if the character is a digit |
| Step 6 | If **isPreviousDigit** and **isPreviousSpace** then |
| Step 7 | Return true // Contains two consecutive operands |
| Step 6 | Set the flag **isPreviousSpace** to false |
| Step 7 | Set the flag **isPreviousDigit** to true |
| Step 8 | Else |
| Step 9 | Set the flag **isPreviousDigit** and **isPreviousSpace** to false |
| Step 10 | Return false // No contain two consecutive operands |

1. **removeSpaces(string, vector<char> &):** This is a void function. The function takes two parameters: A string of the infix expression and a vector of char data type. The function removes all whitespace characters in the expression and at the same time store each character in the expression into the vector.

Here is the function’s pseudocode:

|  |  |
| --- | --- |
| Step 1 | A loop to iterate through each character in the expression |
| Step 2 | If the character is not a whitespace character (spaces or tabs) |
| Step 3 | Push the character back to the vector |

1. **simplifySigns(vector<char>, vector<char> &):** (Advanded new feature) This is a void-return type function. The function takes two parameters including

* vector<char> rawCharacters: Contain raw characters of the infix expression including a sequence of plus and minus signs.
* vector<char> &processCharacters: Contain characters of the infix expression after simplifying sequences of plus and minus signs.

The function detects sequences of plus and minus signs, and counts the number of minus signs. If there is an odd number of minus signs, then the sequence is converted into a single minus sign; otherwise the sequence is converted into a single plus sign. Then, the function adds it into the processCharacters vector.

Here is the function’s pseudocode:

|  |  |
| --- | --- |
| Step 1 | Set minusCount to 0 |
| Step 2 | A loop to interate each character in the rawCharacters vector |
| Step 3 | If (the character is a minus or a plus) |
| Step 4 | If (the chacter is a minus) |
| Step 5 | Increment minusCount |
| Step 6 | While (the next character is a minus or a plus) |
| Step 7 | Increment index |
| Step 8 | If (next character is a minus) |
| Step 9 | Increment minusCount |
| Step 10 | Append to the processCharacters (minus sign if minusCount is odd, plus sign if minusCount is even) |
| Step 11 | Set minusCount to 0 |
| Step 12 | Else |
| Step 13 | Append the character to the processCharacters vector |

1. **isOpenParenthesisValid(char, char &, int &, vector<string> &, bool &):** This is a void-return-type funciton. The function takes five parameters:

* char currentCharacter: this parameter is the current character, which is an open parenthesis.
* char & previousCharacter: this reference parameter is the previous character and plays a role as a condition to check the current character.
* int &numOfOpenParentheses: this reference parameter is the number of open parentheses. It is passed into the function so that we can update and keep track of the open parentheses to validate parentheses.
* vector<string> &tokens: this is a vector, where the parses or tokens of the expression (parentheses, operands, operators) are stored into.
* bool &result: this is a reference variable to hold the checking result.

The function checks whether the open parenthesis is valid. If it is valid, then the function pushes it into the vector as a string, increments the number of open parenthesis, updates the previous character to an open parenthesis, and sets result to true; otherwise the function only sets result to false.

Here is the function’s pseudocode:

|  |  |
| --- | --- |
| Step 1 | If the previous character is a close parenthesis or a number |
| Step 2 | Display error message |
| Step 3 | Set result to false |
| Step 4 | Else |
| Step 5 | Push the open parenthesis to the vector as a string |
| Step 6 | Increment the number of open parentheses |
| Step 7 | Set the previous character to open parenthesis |
| Step 8 | Set the result to true |

1. **isCloseParenthesisValid(char, char &, int &, vector<string> &, bool &):** This is a void-return-type function. The function takes five parameters including

* char currentCharacter: this parameter is the current character, which is an open parenthesis.
* char & previousCharacter: this reference parameter is the previous character and plays a role as a condition to check the current character.
* int &numOfOpenParentheses: this reference parameter is the number of open parentheses. It is passed into the function so that we can update and keep track of the open parentheses to validate parentheses.
* vector<string> &tokens: this is a vector, where the parses or tokens of the expression (parentheses, operands, operators) are stored into.
* bool &result: this is a reference variable to hold the checking result.

The function checks whether the close parenthesis is valid. If it is valid, then the function pushes it into the vector as a string, decrements the number of open parenthesis, updates the previous character to a close parenthesis, and sets result to true; otherwise the function only sets result to false.

Here is the function’s pseudocode:

|  |  |
| --- | --- |
| Step 1 | If the previous character is a null character, or an opertor, or an open parenthesis, or the number of open parenthesis equals to 0 |
| Step 2 | Display error message |
| Step 3 | Set result to false |
| Step 4 | Else |
| Step 5 | Push the open parenthesis to the vector as a string |
| Step 6 | Decrement the number of open parentheses |
| Step 7 | Set the previous character to open parenthesis |
| Step 8 | Set the result to true |

1. **isOperatorValid(char, char &, vector<string> &, bool &, int &, vector<char>)**: This is a void-return-type function. The function takes six parameters including

* char currentCharacter: this parameter is the current character, which is an open parenthesis.
* char & previousCharacter: this reference parameter is the previous character and plays a role as a condition to check the current character.
* vector<string> &tokens: this is a vector, where the parses or tokens of the expression (parentheses, operands, operators) are stored into.
* bool &result: this is a reference variable indicating the success or failure of the validation.
* int & index: this is a reference variable referring to the subscript of the current character in the rawCharacters vector. The index is passed into the function so that it can checks negative multiple-digit operands (E.g. -259;…)
* vector<char> rawCharacters: This is the vector obtained after the removeSpaces function, which contains non-whitespace character in the infix expression.

The function isOperatorValid checks the validity of a given operator (currentChar) in a mathematical expression. It examines the context of the operator by looking at the preceding character (previousChar), the position in the expression (index), and the sequence of raw characters (rawCharacters). Depending on the context, it either processes the operator (e.g., adds it to the list of tokens) or raises an error. The function also handles specific cases such as negative numbers, unary operators, and operators at invalid positions.

Here is the function’s pseudocode:

|  |  |
| --- | --- |
| **Step 1** | **If the previous character is a an opertor, or the current character is at the end of the expression** |
| Step 1.1 | Display error message |
| Step 1.2 | Set result to false |
| **Step 2** | **Else if the operator is at the beginning of the expression** |
| Step 2.1 | If the operator is a minus sign and the next character is a digit |
| Step 2.2 | Take the negative number and push it into vector tokens |
| Step 2.3 | Set the previous character to a number and result to true |
| Step 2.4 | Else if the opeartor is a minus sign |
| Step 2.5 | Push “0” and the negative operator (unary) into the vector |
| Step 2.6 | Set previous character to operators and result to true |
| Step 2.6 | Else if the operator is a plus sign |
| Step 2.7 | Set previous character to a null character and result to true |
| Step 2.8 | Else |
| Step 2.9 | Display error message and set result to false |
| **Step 3** | **Else if the previous character is an open parenthesis** |
| Step 3.1 | If the current character is a minus sign followed by a digit |
| Step 3.1 | Take the negative number and push it into vector tokens |
| Step 3.2 | Set the previous character to a number and result to true |
| Step 3.3 | Else if the current character is a minus |
| Step 3.4 | Push “0” and the negative operator (unary) into the vector |
| Step 3.5 | Set previous character to operators and result to true |
| Step 3.6 | Else if the current character is not a plus |
| Step 3.7 | Display error message and set result to false |
| **Step 4** | **Else** // The current operator is valid |
| Step 4.1 | Push the operator to vector |
| Step 4.2 | Set previous character to an operator and result to true |

1. **isDigitValid(char, char &, vector<string> &, bool &result, int &index, vector<char>):** This is a void-return-type function. The function takes six parameters including

* char currentCharacter: this parameter is the current character, which is an open parenthesis.
* char & previousCharacter: this reference parameter is the previous character and plays a role as a condition to check the current character.
* vector<string> &tokens: this is a vector, where the parses or tokens of the expression (parentheses, operands, operators) are stored into.
* bool &result: this is a reference variable indicating the success or failure of the validation.
* int & index: this is a reference variable referring to the subscript of the current character in the rawCharacters vector. The index is passed into the function so that it can checks negative multiple-digit operands (E.g. -259;…)
* vector<char> rawCharacters: This is the vector obtained after the removeSpaces function, which contains non-whitespace character in the infix expression.

Then, the function checks the validity of a digit (currentChar) in the infix expression and processes multi-digit numbers. It ensures that digits are in valid positions and not directly following a close parenthesis (which would indicate a syntax error). If the digit is part of a multi-digit number, it accumulates the digits into a single token. The token is then added to the list of tokens (tokens). The function also updates the previous character (previousChar) and sets the result (result) to indicate success or failure.

Here is the function’s pseudocode:

|  |  |
| --- | --- |
| **Step 1** | **If the previous character is a close parenthesis** |
| Step 1.1 | Display error message |
| Step 1.2 | Set result to false |
| **Step 2** | **Else** |
| Step 2.1 | Set number to currentChar (as string) |
| Step 2.2 | Increment index |
| Step 2.3 | While index is within bounds and current character is a digit |
| Step 2.4 | Append current character to number and increment index |
| Step 2.5 | Decrement index |
| Step 2.6 | Push number to token |
| Step 2.7 | Set previous character to number and result to true |

# The infixToPrefix Function design and documentation

## Algorithm for the conversion of infix to prefix expression:

The function takes two parameter, which is the vecter of string consisting of infix tokens and a vector of string to store the tokens of the prefix.

* + - Firstly, the function declare two vector of string: one to hold the output (named **output**) and one to hold the operators and parentheses (named **opStack**). The function reverses the vecter of infix tokens.
    - Then, it runs a loop to scan the reversed vector from left to right.
    - Whenever an operand is encoutered, they are put into a vector **output**.
    - If an operator is encountered and the vector **opStack** is found to be empty, then simply push the operator into the back of the vector **opStack**.
    - If the incoming operator has higher precedence than the last operator in **opStack**, push the incoming operator into the back of **opStack**.
    - If the incoming operator has the same precedence with the last operator in the **opStack**, push the incoming operator into the **opStack**. However, if the incoming operator has the same precedence with the last operator in the **opStack** and the incoming operator is “^”, then pop the last operator in the **opStack** till the condition is true. If the condition is not true, push the “^” operator into the back of **opStack**
    - If the incoming operator has lower precedence than the last operator in the **opStack**, pop, and push the last operator in **opStack** into the back of **output**. Test the incoming operator against the last operator in **opStack** again and pop the operator from the **opStack** till it finds the operator of a lower precedence or same precedence.
    - When we reach the end of the expression, pop, and push all operators in **opStack** into the back of **output**.
    - If the operator is ')', then push it into the back of **opStack**.
    - If the operator is '(', then pop all the operators from the **opStack** till it finds ‘)’ opening bracket in the **opStack**.
    - If the the last element in the **opStack** is ')', push the operator into the back of **opStack**.
    - At the end, reverse the vector of string **output**, and runs a loop to copy the reversed vector into the vector of string storing the prefix token.

## The pseudocode of the function

|  |  |
| --- | --- |
| Step 1 | Declare the **output** vector and **opStack** vector |
| Step 2 | Reverse the vector tokens of infix expression |
| Step 3 | A loop to iterate through each token of the reversed vector |
| Step 4 | If the token is an operand, then add it to **output** vector |
| Step 5 | Else if the token is “ ) ” push it into the back of the **opStack** vector |
| Step 6 | Else if the token is “ ( ” |
| Step 6.1 | Pop back operators in the **opStack** vector until “ ) ” |
| Step 6.2 | Pop back the “ ) ” in the **opStack** vector |
| Step 7 | Else if the token is an operator |
| Step 7.1 | If the opStack is empty, push the token into the back of **opStack** |
| Step 7.2 | Else if the precedence(token) > precedence(last in **opStack**) then |
| Step 7.3 |  |
| Step 7.2 | Else if the precedence(token) = precedence(last in **opStack**) then |
| Step 7.3 | While token is “^”, pop the last operator in **opStack** and push it into the back of **output** |
| Step 7.4 | Push the token into the back of **opStack** |
| Step 7.5 | Else if the precedence(token) precedence( |

# Prefix calculation desgin

* **Initialize an Operands List**: Create a list to hold operands during evaluation.
* **Iterate Over the Prefix Expression**: Traverse the prefix expression from right to left.
* **Handle Operands**: If an element is an operand, convert it to an integer and push it onto the operands list.
* **Handle Operators**:
  + If an element is an operator, pop the top two operands from the list.
  + Perform the operation (addition, subtraction, multiplication, division, or exponentiation) on these operands.
  + Push the result back onto the operands list.
* **Return the Final Result**: After processing all elements, the final result is the remaining element in the operands list.

|  |  |
| --- | --- |
| **Step** | **Implementation** |
| 1 | DECLARE operands AS EMPTY LIST OF INTEGERS |
| 2 | LOOP to run through each element from right to left:  FOR index FROM (size of prefix - 1) TO 0 DO |
| 3 | SET lastElement TO element at index in prefix |
| 4 | IF lastElement is operand THEN |
| 5 | CONVERT lastElement to integer |
| 6 | PUSH converted integer TO operands |
| 7 | ELSE  SET operand\_1 TO top element of operands |
| 8 | REMOVE top element from operands |
| 9 | SET operand\_2 TO top element of operands |
| 10 | REMOVE top element from operands |
| 11 | DECLARE result AS INTEGER |
| 12 | IF lastElement IS "+" THEN  SET result TO operand\_1 + operand\_2 |
| 13 | ELSE IF lastElement IS "-" THEN  SET result TO operand\_1 - operand\_2 |
| 14 | ELSE IF lastElement IS "\*" THEN  SET result TO operand\_1 \* operand\_2 |
| 15 | ELSE IF lastElement IS "/" THEN  SET result TO operand\_1 / operand\_2 |
| 16 | ELSE IF lastElement IS "^" THEN  SET result TO operand\_1 raised to the power of operand\_2 |
| 17 | PUSH result TO operands |
| 18 | END FOR |
| 19 | RETURN top element of operands |

# 

# Some test cases and demo examples of the program

## Example 1:

Infix expression: 2 + 3 \* (1 + 2) + 2^2 → Prefix expression: + + 2 \* 3 + 1 2 ^ 2 2

Prefix calculation: 15

A screenshot of a computer

Description automatically generated

## Example 2:

Infix expression: -2 \* (-2) + 3 \* (-2) → Prefix expression: + \* -2 -2 \* 3 -2

Prefix calculation: -2

A screenshot of a computer

Description automatically generated

## Example 3:

Display an error message if user chooses 2 but there is no prefix expression

A black screen with white text

Description automatically generated

## Example 4: Handle a squence of plus and minus signs

Infix expression: 5 -+-- 3 \* 2 = 5 – 3 \* 2 → Prefix expression: - 5 \* 3 2

A screenshot of a computer program

Description automatically generated

## Example 5: An error infix expression

Invalid infix expression:

* 3 \*- 3 → Display ERROR: Two invalid consecutive operators
* A screenshot of a computer program

  Description automatically generated3 \* (2 + 3)) → Display ERROR: Unmatched parentheses