**Notations**

An arithmetic expression can be written in three different but equivalent notation −

* Infix Notation
* Prefix (Polish) Notation
* Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression.

**Infix Notation**

An expression is written in **infix** notation, e.g. **a - b + c,** where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

**Prefix Notation**

In this notation, operator is **prefix**ed to operands. Operator is written ahead of operands. For example: **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

**Postfix Notation**

This notation style is known as **Reversed Polish Notation**. In this notation, the operator is **postfix**ed to the operands. The operator is written after the operands. For example: **ab+**. This is equivalent to its infix notation **a + b**.

The following table shows the difference in all three notations −

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.No.** | **Infix Notation** | **Prefix Notation** | **Postfix Notation** |
| 1 | a + b | + a b | a b + |
| 2 | (a + b) ∗ c | ∗ + a b c | a b + c ∗ |
| 3 | a ∗ (b + c) | ∗ a + b c | a b c + ∗ |
| 4 | a / b + c / d | + / a b / c d | a b / c d / + |
| 5 | (a + b) ∗ (c + d) | ∗ + a b + c d | a b + c d + ∗ |
| 6 | ((a + b) ∗ c) - d | - ∗ + a b c d | a b + c ∗ d - |

**Parsing Expressions**

It is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed. To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.No.** | **Operator** | **Precedence** | **Associativity** |
| 1 | Exponentiation ^ | Highest | Right Associative |
| 2 | Multiplication ( ∗ ) & Division ( / ) | Second Highest | Left Associative |
| 3 | Addition ( + ) & Subtraction ( − ) | Lowest | Left Associative |

Examples:

|  |  |  |  |
| --- | --- | --- | --- |
| **Infix** | **Postfix** | **Prefix** | **Notes** |
| A \* B + C / D | A B \* C D / + | + \* A B / C D | multiply A and B, divide C by D, add the results |
| A \* (B + C) / D | A B C + \* D / | / \* A + B C D | add B and C, multiply by A, divide by D |
| A \* (B + C / D) | A B C D / + \* | \* A + B / C D | divide C by D, add B, multiply by A |

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

**Parse trees:** A similar trick to convert to and from parse trees - each bracketed triplet of an operator and its two operands (or sub-expressions) corresponds to a node of the tree. The corresponding parse trees are:

/ \*

+ / \ / \

/ \ \* D A +

/ \ / \ / \

\* / A + B /

/ \ / \ / \ / \

A B C D B C C D

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

**Postfix Evaluation Algorithm**

Step 1 − scan the expression from left to right

Step 2 − if it is an operand push it to stack

Step 3 − if it is an operator pull operand from stack and perform operation

Step 4 − store the output of step 3, back to stack

Step 5 − scan the expression until all operands are consumed

Step 6 − pop the stack and perform operation

## Infix to Postfix Conversion Algorithm

Let Q be any infix expression and we have to convert it to postfix expression P.

**Step1:** Push left parenthesis onto [STACK](https://www.thecrazyprogrammer.com/2013/12/c-program-for-array-representation-of-stack-push-pop-display.html) and add right parenthesis at the end of Q.

**Step2:** Scan Q from left to right and repeat step 3 to 6 for each element of Q until the STACK is empty.

**Step3:** If an operand is encountered add it to P.

**Step4:** If a left parenthesis is encountered push it onto the STACK.

**Step51:** If an operator is encountered, then

* Repeatedly pop from STACK and add to P each operator  
  which has same precedence as or higher precedence than the operator  
  encountered.
* Push the encountered operator onto the STACK.

**Step6:** If a right parenthesis is encountered, then

* Repeatedly pop from the STACK and add to P each operator  
  until a left parenthesis is encountered.
* Remove the left parenthesis; do not add it to P.

**Step7:** Exit

**Example:**

Infix Expression: 5+3\*2

Postfix Expression: 5 3 2\*+.

Infix Expression: **A+ (B\*C-(D/E^F)\*G)\*H**,

**Resultant Postfix Expression: ABC\*DEF^/G\*-H\*+**

For better understanding, let us trace out an example **A \* B – (C + D) + E**

|  |  |  |  |
| --- | --- | --- | --- |
| **INPUT CHARACTER** | **OPERATION ON STACK** | **STACK** | **POSTFIX EXPRESSION** |
| A |  | Empty | A |
| \* | Push | \* | A |
| B |  | \* | A B |
| – | Check and Push | – | A B \* |
| ( | Push | – ( | A B \* |
| C |  | – ( | A B \* C |
| + | Check and Push | – ( + | A B \* C |
| D |  | – ( + | A B \* C D |
| ) | Pop and append to postfix till ‘(‘ | – | A B \* C D + |
| + | Check and push | + | A B \* C D + – |
| E |  | + | A B \* C D + – E |
| End of Input | Pop till Empty | Empty | A B \* C D + – E + |

**POSTFIX Expression :- A B \* C D + – E +**

**Algorithm to Convert Infix to Prefix**

**Step 1.** Push “)” onto STACK, and add “(“ to end of the A  
**Step 2.** Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty  
**Step 3.** If an operand is encountered add it to B  
**Step 4.** If a right parenthesis is encountered push it onto STACK  
**Step 5.** If an operator is encountered then:

**a.** Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator.

**b.** Add operator to STACK

**Step 6.** If left parenthesis is encountered then

**a.** Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encountered)

**b.** Remove the left parenthesis

**Step 7.** Exit