Expression Parsing

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

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

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression 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, i.e. 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 style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

The following table briefly tries to show 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

As we have discussed, 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

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and − have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

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

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example –

In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c**.

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation –

```
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
```

Complete implementation

Following are the complete implementations of Expression Parsing in various programming languages –

C | C++ | Java | Python

```c
#include<stdio.h>
#include<string.h>
#include<ctype.h>
//char stack
char stack[25];
```

```
int top = -1;
void push(char item) {
   stack[++top] = item;
}
char pop() {
   return stack[top--];
}
//returns precedence of operators
int precedence(char symbol) {
   switch(symbol) {
      case '+':
      case '-':
         return 2;
         break;
      case '*':
      case '/':
         return 3;
         break;
      case '^':
         return 4;
         break;
      case '(':
      case ')':
      case '#':
         return 1;
         break;
   }
}

//check whether the symbol is operator?
int isOperator(char symbol) {

   switch(symbol) {
      case '+':
      case '-':
      case '*':
      case '/':
      case '^':
      case '(':
      case ')':
```

```
         return 1;
      break;
         default:
         return 0;
   }
}

//converts infix expression to postfix
void convert(char infix[],char postfix[]) {
   int i,symbol,j = 0;
   stack[++top] = '#';

   for(i = 0;i<strlen(infix);i++) {
      symbol = infix[i];

      if(isOperator(symbol) == 0) {
         postfix[j] = symbol;
         j++;
      } else {
         if(symbol == '(') {
            push(symbol);
         } else {
            if(symbol == ')') {

               while(stack[top] != '(') {
                  postfix[j] = pop();
                  j++;
               }

               pop();   //pop out (.
            } else {
               if(precedence(symbol)>precedence(stack[top])) {
                  push(symbol);
               } else {

                  while(precedence(symbol)<=precedence(stack[top])) {
                     postfix[j] = pop();
                     j++;
                  }
```

```
                push(symbol);
             }
          }
       }
    }

    while(stack[top] != '#') {
       postfix[j] = pop();
       j++;
    }

    postfix[j]='\0';  //null terminate string.
}

//int stack
int stack_int[25];
int top_int = -1;

void push_int(int item) {
    stack_int[++top_int] = item;
}

char pop_int() {
    return stack_int[top_int--];
}

//evaluates postfix expression
int evaluate(char *postfix){

    char ch;
    int i = 0,operand1,operand2;

    while( (ch = postfix[i++]) != '\0') {

       if(isdigit(ch)) {
          push_int(ch-'0');  // Push the operand
       } else {
          //Operator,pop two  operands
          operand2 = pop_int();
```

```
         operand1 = pop_int();

         switch(ch) {
            case '+':
               push_int(operand1+operand2);
               break;
            case '-':
               push_int(operand1-operand2);
               break;
            case '*':
               push_int(operand1*operand2);
               break;
            case '/':
               push_int(operand1/operand2);
               break;
         }
      }
   }

   return stack_int[top_int];
}

void main() {
   char infix[25] = "1*(2+3)",postfix[25];
   convert(infix,postfix);

   printf("Infix expression is: %s\n" , infix);
   printf("Postfix expression is: %s\n" , postfix);
   printf("Evaluated expression is: %d\n" , evaluate(postfix));
}
```

Output

```
Infix expression is: 1*(2+3)
Postfix expression is: 123+*
Evaluated expression is: 5
```