Lecture 6-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**

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 prefixed 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 postfixed 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 −

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.

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) −

Operator Precedence Associativity

1 Exponentiation ^ Highest Right Associative

2 Multiplication (*) & Division (/) Second Highest Left Associative

3 Addition (+) & Subtraction (−) Lowest Left Associative

Why using postfix notation?

1. It is the most suitable notation system to calculate any expression duo to its reversing characteristic and the absence of using parenthesis.
2. It is universally accepted notation for designing the arithmetic logic unit (ALU) of the processor.
3. It is the way computers look toward any arithmetic expression.

Conversion Rules

There are a set of rules need to be performed when using the stack in converting infix notation to postfix notation, these rules are:

1. When reading a number or data item, perform pop immediately.
2. When reading an operator, pop until the top of the stack has an element of lower precedence and push this element.
3. When any ending parenthesis or brackets like) or] is pushed, then perform pop until reaching the matching (or [.
4. The parenthesis or brackets have the lowest precedence inside the stack.
5. When the end of the input is reached, pop until the stack is empty.

Operator Precedences

Type	*Operators*
scope resolution	namespace_name :: member
selection/subscripting function call postfix operators	class_name.member pointer–>member array[exp] function(args) var++ var−−
prefix operators dereference/address	++var −−var +exp −exp ˜exp !exp *pointer &var
multiplication/division	* / %
addition/subtraction	+ −
shift	<< >>
comparison	< <= > >=
equality	== !=
bitwise and	&
bitwise exclusive-or	^
bitwise or	|
logical and	&&
logical or	||
conditional	bool_exp ? true_exp : false_exp
assignment	= += −= *= /= %= >>= <<= &= ^= |=

Example:

Convert the following infix expression to the equivalent postfix notation.

1. A + B * C

Solution:

step	Input	Stack	Output
1	A		A
2	+	+	A
3	B	+	AB
4	*	+ *	AB
5	C	+ *	ABC

6			ABC*+

The postfix expression is: ABC*+

2. 3 + 4 * 5 / 6

Solution:

step	Input	Stack	Output
1	3		3
2	+	+	3
3	4	+	34
4	*	+*	34
5	5	+*	345
6	/	+/	345*
7	6	+/	345*6
8			345*6/+

The postfix expression is: 345*6/+

3. (30+23)*(43-21)/(84+7)

Solution:

step	Input	Stack	Output
1	(	(	
2	30	(	30
3	+	(+	30
4	23	(+	30 23
5	)		30 23 +
6	*	*	30 23 +
7	(	*(	30 23 +
8	43	*(	30 23 + 43
9	-	*(-	30 23 + 43
10	21	*(-	30 23 + 43 21
11	)	*	30 23 + 43 21 -
12	/	/	30 23 + 43 21 - *
13	(	/(	30 23 +43 21 - *
14	84	/(	30 23 +43 21 - * 84
15	+	/(+	30 23 +43 21 - * 84
16	7	/(+	30 23 +43 21 - * 84 7
17	)	/	30 23 + 43 21 - * 84 7 +

18			30 23 + 43 21 - * 84 7+ /

The postfix expression is: 30 23 + 43 21 - * 84 7+ /

4. (4+8)*(6-5)/((3-2)*(1+2))

Solution:

step	Input	Stack	Output
1	(	(	
2	4	(	4
3	+	(+	4
4	8	(+	4 8
5	)		4 8 +
6	*	*	4 8 +
7	(	*(	4 8 +
8	6	*(	4 8 + 6
9	-	*(-	4 8 + 6
10	5	*(-	4 8 + 6 5
11	)	*	4 8 + 6 5 -
12	/	/	4 8 + 6 5 - *
13	(	/(	4 8 + 6 5 - *
14	(	/((	4 8 + 6 5 - *

15	3	/((	4 8 + 6 5 - * 3
16	-	/(-	4 8 + 6 5 - * 3
17	2	/(-	4 8 + 6 5 - * 3 2
18	)	/(	4 8 + 6 5 - * 3 2 -
19	*	/(*	4 8 + 6 5 - * 3 2 -
20	(	/(*(	4 8 + 6 5 - * 3 2 -
21	1	/(*(	4 8 + 6 5 - * 3 2 - 1
22	+	/(*(+	4 8 + 6 5 - * 3 2 - 1
23	2	/(*(+	4 8 + 6 5 - * 3 2 – 1 2
24	)	/(*	4 8 + 6 5 - * 3 2 – 1 2 +
25	)	/	4 8 + 6 5 - * 3 2 – 1 2 + *
26			4 8 + 6 5 - * 3 2 – 1 2 + * /

The postfix expression is: 4 8 + 6 5 - * 3 2 – 1 2 + * /

Tutorials:

1. What is the obtained number after performing the following sequence of push and pop:

   PUSH(1), POP, PUSH(2), POP, PUSH(3), PUSH(4), POP, PUSH(5), POP, POP

2. What is the required sequence of push and pop to obtain the string (25431) from the initial input (12345).

3. Convert the following infix expression to the equivalent postfix notation:

    - 1 + 2 - 3
    - A * B + C * D
    - (A + B) * C / D
    - (A + B) * C / D + E ^ F / G
    - A + [(B + C) + (D + E) * F] / G