DATA STRUCTURES & ALGORITHMS

Polish Notation

Instructor: Engr. Laraib Siddiqui

Introduction

- Polish notation was invented in 1924 by Jan Lukasiewicz, a Polish logician and philosopher.

- The idea is simply to have a parenthesis-free notation that makes each equation shorter and easier to parse in terms of defining the evaluation priority of the operators.

Example

Infix notation with parenthesis:

(3 + 2) * (5 – 1)

Polish notation:

* + 3 2 – 5 1

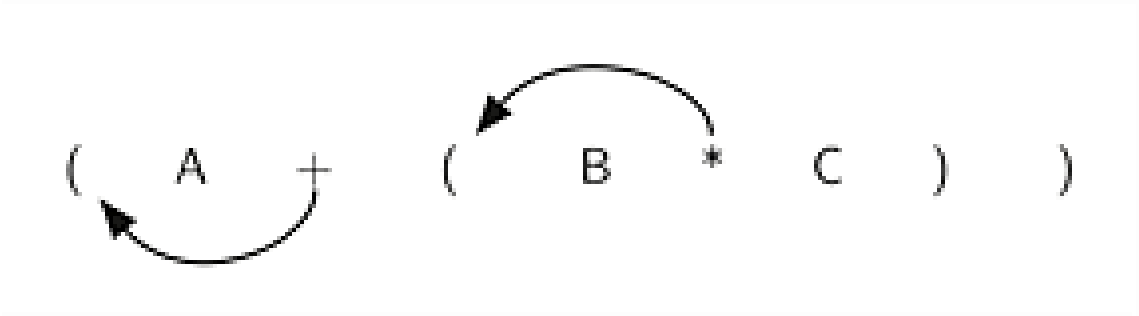
Precedence and Associativity

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.

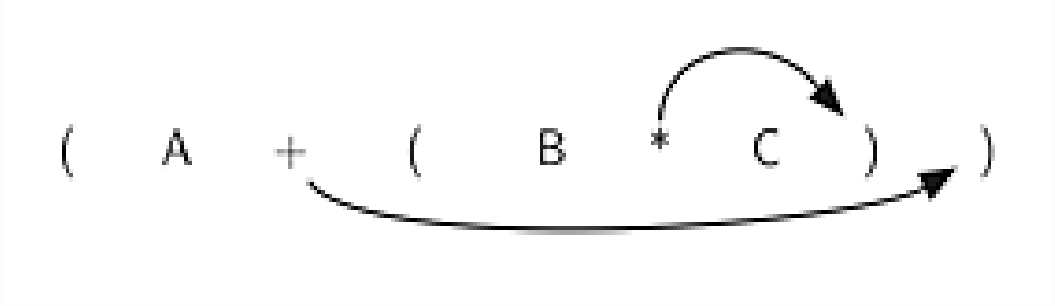
Associativity
Associativity describes the rule where operators with the same precedence appear in an expression.

S.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (–)	Lowest	Left Associative

Example



Moving Operators to the Left for Prefix Notation

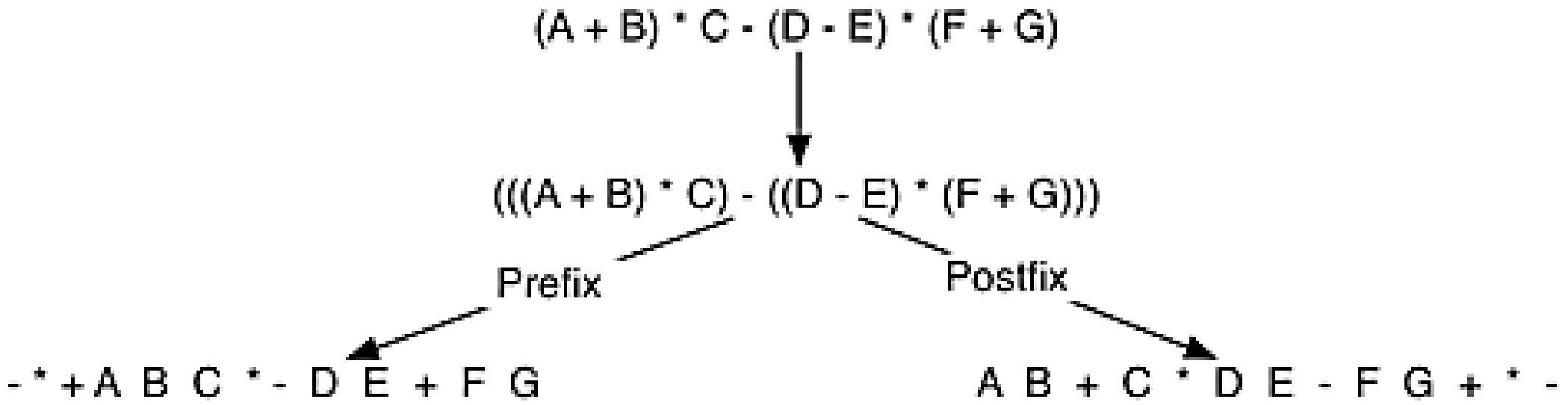


Moving Operators to the Right for Postfix Notation

Converting Infix to Prefix(Polish notation) and Postfix (Reverse Polish notation)

S.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 -

Converting Infix to Prefix and Postfix



Example

Solving equation according to their precedence:

$2\uparrow 3 + 5 * 2\uparrow 2 - 12 / 6$

(Resolving exponentials)

$8 + 5 * 4 - 12 / 6$

(Division and Multiplication)

$8 + 20 - 2$

(Addition and subtraction)

26

Evaluate

5 , 6, 2, +, *, 12, 4, /, -

Infix to postfix using Stack

1. Scan input string from left to right character by character.
2. If the character is an operand, put it into output stack.
3. If the character is an operator and operator's stack is empty, push operator into operators' stack.
4. If the operator's stack is not empty, there may be following possibilities.
    1. If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operand's stack.
    2. If the precedence of scanned operator is less than or equal to the top most operator of operator's stack, pop the operators from operand's stack until we find a low precedence operator than the scanned character. Never pop out (**'('**) or (**')'**) whatever may be the precedence level of scanned character.
    3. If the character is opening round bracket (**'('**), push it into operator's stack.
    4. If the character is closing round bracket (**')'**), pop out operators from operator's stack until we find an opening bracket (**'('**).
    5. Now pop out all the remaining operators from the operator's stack and push into output stack.

Infix to postfix using stack

A+B*C/(E-F)

Input String	Output Stack	Operator Stack
A+B*C/(E-F)	A	
A+B*C/(E-F)	A	+
A+B*C/(E-F)	AB	+
A+B*C/(E-F)	AB	+*
A+B*C/(E-F)	ABC	+*
A+B*C/(E-F)	ABC*	+/
A+B*C/(E-F)	ABC*	+/(
A+B*C/(E-F)	ABC*E	+/(
A+B*C/(E-F)	ABC*E	+/(-
A+B*C/(E-F)	ABC*EF	+/(-
A+B*C/(E-F)	ABC*EF-	+/
A+B*C/(E-F)	ABC*EF-/+	

Infix to Prefix using stack

The following algorithm must be followed for infix to prefix conversion.

1. Reverse the input string.
2. Convert the reversed string into postfix expression.
3. Now reverse the resulting infix expression obtained from the previous step. The resulting expression is prefix expression

Infix to Prefix using stack

A+B*C/(E-F)

1. Reversed string: (F-E)/C*B+A
2. Postfix of (F-E)/C*B+A: FE-C/B*A+
3. Reversed string of FE-C/B*A+: +A*B/C-EF

Input String	Operator Stack	Output Stack
(F-E)/C*B+A	(	
(F-E)/C*B+A	(	F
(F-E)/C*B+A	(-	F
(F-E)/C*B+A		FE-
(F-E)/C*B+A	/	FE-
(F-E)/C*B+A	/	FE-C
(F-E)/C*B+A	*	FE-C/
(F-E)/C*B+A	*	FE-C/B
(F-E)/C*B+A	+	FE-C/B*
(F-E)/C*B+A	+	FE-C/B*A
(F-E)/C*B+A		FE-C/B*A+
(F-E)/C*B+A		+A*B/C-EF

Postfix to Prefix

1. Scan the given postfix expression from **left to right** character by character.
2. If the character is an operand, push it into the stack.
3. But if the character is an operator, pop the top two values from stack.
4. Concatenate this operator with these two values (**operator+2nd top value+1st top value**) to get a new string.
5. Now push this resulting string back into the stack.
6. Repeat this process until the end of postfix expression. Now the value in the stack is the desired prefix expression.

Postfix to Prefix

ABC/-AK/L-*

Input String	Postfix Expression	Stack (Prefix)
ABC/-AK/L-*	BC/-AK/L-*	A
ABC/-AK/L-*	C/-AK/L-*	AB
ABC/-AK/L-*	/-AK/L-*	ABC
ABC/-AK/L-*	-AK/L-*	A/BC
ABC/-AK/L-*	AK/L-*	-A/BC
ABC/-AK/L-*	K/L-*	-A/BCA
ABC/-AK/L-*	/L-*	-A/BCAK
ABC/-AK/L-*	L-*	-A/BC/AK
ABC/-AK/L-*	-*	-A/BC/AKL
ABC/-AK/L-*	*	-A/BC-/AKL
ABC/-AK/L-*		*-A/BC-/AKL

Prefix to Postfix

1. Scan the given prefix expression from **right to left** character by character.
2. If the character is an operand, push it into the stack.
3. But if the character is an operator, pop the top two values from stack.
4. Concatenate this operator with these two values (**operator+1st top value+2nd top value**) to get a new string.
5. Now push this resulting string back into the stack.
6. Repeat this process untill the end of prefix expression. Now the value in the stack is the desired postfix expression.

Prefix to Postfix

***-A/BC-/AKL**

Input String	Prefix Expression	Stack (Postfix)
*-A/BC-/AKL	*-A/BC-/AKL	
*-A/BC-/AKL	*-A/BC-/AK	L
*-A/BC-/AKL	*-A/BC-/A	LK
*-A/BC-/AKL	*-A/BC-/	LKA
*-A/BC-/AKL	*-A/BC-	LAK/
*-A/BC-/AKL	*-A/BC	AK/L-
*-A/BC-/AKL	*-A/B	AK/L-C
*-A/BC-/AKL	*-A/	AK/L-CB
*-A/BC-/AKL	*-A	AK/L-BC/
*-A/BC-/AKL	*-	AK/L-BC/A
*-A/BC-/AKL	*	AK/L-ABC/-
*-A/BC-/AKL		ABC/-AK/L-*