

ARITHMETIC EXPRESSIONS

One of the most important applications of stacks is the evaluation of arithmetic expressions.

Consider the simple arithmetic expression;

$$A+B$$

We have three possibilities for the positioning of the operator;

1. Before the operands as

$$+AB$$

which is called **Prefix notation** or **Polish notation**.
(after the Polish logician Jan Lu Lukasiewicz)

2. Between the operands

$$A+B$$

which is called **Infix notation** .

3. After the operands

$$AB+$$

which is called **Postfix notation** or **Reverse Polish notation** .

How to convert

→ from infix to postfix

→ from infix to prefix

General rules for conversion

1. Completely parenthesize the infix expression according to the order of precedence to specify the order of operation.

2. Move each operator to its corresponding

→right (for postfix)

→left (for prefix)

3. Remove all parentheses

Consider the arithmetic operators and their precedence as

<u>operators</u>	<u>priority</u>
\wedge	highest
$*$, $/$	↓
$+$, $-$	lowest

Example

$A/B^{\wedge}C + D * E - A * C$ ← infix form;

infix to postfix conversion

$A / B^{\wedge} C + D * E - A * C$ ← infix form

$((A / (B^{\wedge} C)) + (D * E) - (A * C))$

$((((ABC^{\wedge} /) + (DE*)) - (AC*)))$

$(((ABC^{\wedge} / DE*) +) - (AC*))$

$ABC^{\wedge} / DE* + AC* -$

← postfix form

Use the standard convention that when unparenthesised operators of the same precedence are scanned, the order from left to right, except in the case of exponentiation which is from right to left.

$$A + B - C \rightarrow (A + B) - C$$

$$A / B * C \rightarrow (A / B) * C$$

$$A \wedge B \wedge C \rightarrow A \wedge (B \wedge C)$$

convert infix to prefix

$$A / B^{\wedge} C + D * E - A * C \quad \leftarrow \text{infix form}$$

$$(((A / (B^{\wedge} C)) + (D * E)) - (A * C))$$

$$(((A / (^{\wedge} BC)) + (*DE)) - (*AC))$$

$$((/A^{\wedge} BC) + (*DE)) - (*AC)$$

$$(+ /A^{\wedge} BC * DE) - (*AC)$$

$$\boxed{-+ /A^{\wedge} BC * DE * AC} \quad \leftarrow \text{prefix form}$$

Advantage

1. The need for parantheses is eliminated
2. The precedence rule is no longer relevant
3. The expression is evaluated by making

→ left to right scan for postfix

→ right to left scan for prefix
by the use of stacking.

Example

$$(B^2 - 4 * A * C)^{(1/2)}$$

infix to postfix

$$(B^2 - 4 * A * C)^{(1/2)}$$

$$((B^2) - ((4A*) * C))^{(1/2)}$$

$$((B^2) - ((4A*C*))^{(1/2)}$$

$$(B^2 4A*C* -)^{(1/2)}$$

$$(B^2 4A*C* - 1/2)^{\wedge}$$
 ← postfix notation

infix to prefix

$$(B^2 - 4 * A * C)^{(1/2)}$$

$$((B^2) - ((4 * A) * C))^{(1/2)}$$

$$((^B2) - ((* 4 A) * C))^{(/12)}$$

$$((^B2) - (** 4 A C))^{(/12)}$$

$$((-^B2**4AC))^{(/12)}$$

$$(^-^B2**4AC/12)$$
 ← prefix notation

compilers use "postfix notation".

Algorithm to convert from infix to RPN (postfix)

1. Initialize an empty stack of operators
2. While no error and not to end of infix expression
 - a. Get the next input token (const,variable,arithmetic operator, left paranthesis, right paranthesis) in the infix expression.
 - b. If the token is
 - i) A left paranthesis: Push it onto the stack
 - ii) A right paranthesis: Pop and display stack element until a left paranthesis is encountered. But do not display the left parentheses. It is an error if the stack becomes empty with no left parentheses found.
 - iii) An operator: If the stack is empty or input token has higher priority than the top stack, push onto the stack.

Otherwise, pop and display the top stack element, then, repeat the comparison of input token with the new top stack item.
3. When end of infix expression is reached, pop and display stack items until the stack is empty.

infix to postfix

an array for infix expr.	an array for postfix expr.	operator stack
A + B * C #	empty	# (empty)
+ B * C #	A	#
B * C #	A	# +
* C #	A B	# +
C #	A B	# + *
#	A B C	# + *
	A B C *	# +
	A B C * +	#
A * B + C #	empty	# (empty)
* B + C #	A	#
B + C #	A	# *
+ C #	A B	# *
C #	A B *	#
C #	A B *	# +
#	A B * C	# +
	A B * C +	#
	A B * C +	

If incoming operator (*) has higher priority than top stack element (+) then, push it.

pop and display until incoming operator (+) lower priority than stack top or empty.

Example:


If the table of priorities


Incoming priority	3	2	2	1	1	4	0	0
Symbol (operator)	\wedge	*	/	+	-	()	#
Instack priority	3	2	2	1	1	0	?	0

→ Operators are taken out of stack as long as in stack priority is greater or equal to the incoming new operator priority.

→ Priority of ')' is said to be undefined (?) because it is never pushed onto the stack.

infix	postfix	operator stack
A * (B+C-D/E)/F#	(empty)	#(empty)
* (B+C-D/E)/F#	A	#
(B+C-D/E)/F#	A	# *
B+C-D/E)/F#	A	# *(
+C-D/E)/F#	AB	# *(
C-D/E)/F#	AB	# *(+
-D/E)/F#	ABC	# *(+ -
	ABC+	# *(
D/E)/F#	ABC+	# *(-
/E)/F#	ABC+D	# *(-
E)/F#	ABC+D	# *(-/
)/F#	ABC+DE	# *(-/
	ABC+DE/	# *(-
	ABC+DE/-	# *
/F#	ABC+DE/-	# * / = *
	ABC+DE/- *	#
F#	ABC+DE/- *	#/
#	ABC+DE/- *F	#/
	ABC+DE/- *F/	#(empty)

infix	postfix	operator
(A+B) * C #	(empty)	#(empty)
A+B) * C #	(empty)	# (
+B) * C #	A	# (
B) * C #	A	# (+
) * C #	AB	# (+  Pop and display until left paranthesis
* C #	AB+	# (
C #	AB+	# (
#	AB+	# (*
	AB+C	# (*
	AB+C *	#(empty)
	AB+C *	

infix	postfix	operator stack
A * B + (C - D/E) #	(empty)	#(empty)
*B + (C - D/E) #	A	#
B + (C - D/E) #	A	# * (+ = *)  pop
+ (C - D/E) #	AB	#
(C - D/E) #	AB *	# +
C - D/E) #	AB *	# + (
- D/E) #	AB * C	# + (
D/E) #	AB * C	# + (-
/E) #	AB * CD	# + (-
E) #	AB * CD	# + (-/
) #	AB * CDE	# + (-/
#	AB * CDE/	# + (-
#	AB * CDE/-	# +
#	AB * CDE/-+	#
	AB * CDE/-+	

Operators in stack are in increasingly order!

Evaluating postfix and prefix expressions

Postfix

1. Scan the expression from left to right.
 2. Each time an operator is encountered, apply it to the immediately preceding operands.
 3. Replace the operands and the operator with the result.
 4. Continue scanning to the right
- use stack for operand tokens and results.

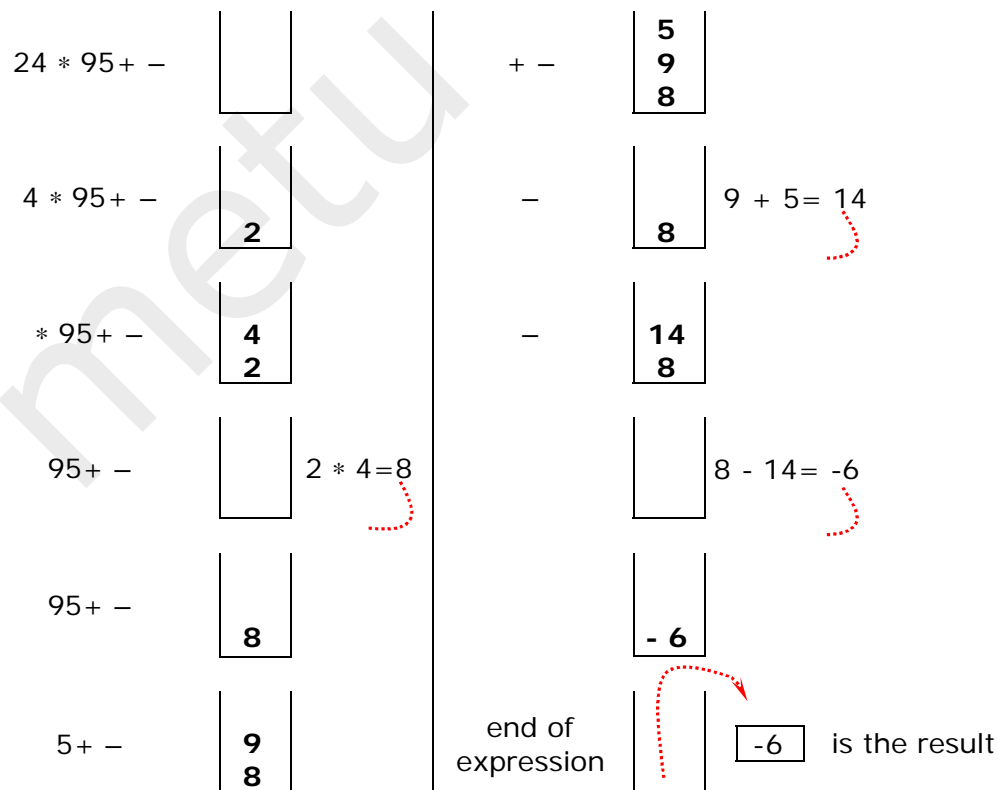
<u>infix</u>	<u>postfix</u>	
3+4 * 5	345 * +	
	45 * +	3
	5 * +	4 3
	* +	5 4 3
	+	20 3
		23
		end of expression Empty the stack.

<u>infix</u>	<u>postfix</u>	
(7+2) * 3	72+3 *	
	2+3 *	7
	+3 *	2 7
	3 *	9
	*	3 9
		27
		end of expression Empty it.

Algorithm to evaluate postfix (RPN) expressions

1. Initialize an empty stack.
2. Repeat the following until the end of expression.
 - a. Get the next token(const, variable, operator) in RPN expression.
 - b. If the token is an operand, push it onto the stack. If it is an operand, then do the following
 - i. Pop the top two values from the stack. If the stack does not contain two items, an error due to a malformed RPN expression has occurred, and evaluation terminated.
 - ii. Apply the operator to these two values.
 - iii. Push the resulting value back onto the stack.
3. When the end of expression is encountered, its value is on top of the stack (and, in fact , must be the only value in the stack)

Example: $24 * 95 + -$



Evaluation of prefix expressions

1. Scan the expression from right to left.
2. Each time an operator is encountered, apply it to the two operands that immediately follow the operator.
3. Replace the operands and the operator with the result.
4. Continue scanning the expression to the left.

Example:

infix

$3*(2-6)$

prefix

$*3\ 2\ 6$

$*3(-2\ 6)$

$*3(2\ 6)$

$*3(-4)$

$(-4)\ *3$

-12
result

infix

$(5+3)\ * (6-4)$

prefix

$*+53\ 6\ 4$

$*+53(-6\ 4)$

$*+53(6\ 4)$

$*+53(2)$

$* (+53)(2)$

$* (5+3)(2)$

$* (8)(2)$

$(8)\ * (2)$

16
result