- Convert Infix Expression to Post-Fix Expression

Conventional notation is called infix notation. The arithmetic operators appears between two operands. Parentheses are required to specify the order of the operations. For example: a + (b * c).

Post fix notation (also, known as reverse Polish notation) eliminates the need for parentheses. There are no precedence rules to learn, and parenthese are never needed. Because of this simplicity, some popular hand-held calculators use postfix notation to avoid the complications of multiple sets of parentheses. The operator is placed directly after the two operands it needs to apply. For example: a b c * +

This short example makes the move from infix to postfix intuitive. However, as expressions get more complicated, there will have to be rules to follow to get the correct result:

Simple heuristic algorithm to visually convert infix to postfix

- Fully parenthesize the expression, to reflect correct operator precedence
- Move each operator to replace the right parenthesis to which it belongs
- Remove parentheses

Evaluating expressions

A stack is used in two phases of evaluating an expression such as

3 * 2 + 4 * (A + B)

•Convert the infix form to postfix using a stack to store operators and then pop them in correct order of precedence.
•Evaluate the postfix expression by using a stack to store operands and then pop them when an operator is reached.

Infix to postfix conversion
Scan through an expression, getting one token at a time.

1 Fix a priority level for each operator. For example, from high to low:

3. - (unary negation)
2. * /
1. + - (subtraction)

Thus, high priority corresponds to high number in the table.

2 If the token is an operand, do not stack it. Pass it to the output.

3 If token is an operator or parenthesis, do the following:

-- Pop the stack until you find a symbol of lower priority number than the current one. An incoming left parenthesis will be considered to have higher priority than any other symbol. A left parenthesis on the stack will not be removed unless an incoming right parenthesis is found.
The popped stack elements will be written to output.

--Stack the current symbol.

-- If a right parenthesis is the current symbol, pop the stack down to (and including) the first left parenthesis. Write all the symbols except the left parenthesis to the output (i.e. write the operators to the output).

-- After the last token is read, pop the remainder of the stack and write any symbol (except left parenthesis) to output.

Example:

Convert A * (B + C) * D to postfix notation.

Move	Curren Ttoken	Stack	Output
1	A	empty	A

2	*	*	A
3	(	(*	A
4	B	(*	A B
5	+	+(*	A B
6	C	+(*	A B C
7	)	*	A B C +
8	*	*	A B C + *
9	D	*	A B C + * D
10		empty	

Notes:
In this table, the stack grows toward the left. Thus, the top of the stack is the leftmost symbol.
In move 3, the left paren was pushed without popping the * because * had a lower priority then "(".
In move 5, "+" was pushed without popping "(" because you never pop a left parenthesis until you get an incoming right parenthesis. In other words, there is a distinction between incoming priority, which is very high for a "(", and instack priority, which is lower than any operator.
In move 7, the incoming right parenthesis caused the "+" and "(" to be popped but only the "+" as written out.
In move 8, the current "*" had equal priority to the "*" on the stack. So the "*" on the stack was popped, and the incoming "*" was pushed onto the stack.

Evaluating Postfix Expressions

Once an expression has been converted to postfix notation it is evaluated using a stack to store the operands.

- Step through the expression from left to right, getting one token at a time.
- Whenever the token is an operand, stack the operand in the order encountered.
- When an operator is encountered:
- If the operator is binary, then pop the stack twice
- If the operator is unary (e.g. unary minus), pop once
- Perform the indicated operation on the operator(s)
- Push the result back on the stack.
- At the end of the expression, the top of the stack will have the correct value for the expression.

Example:

Evaluate the expression 2 3 4 + * 5 * which was created by the previous algorithm for infix to postfix.

Move	Current Token	Stack (grows toward left)
1	2	2
2	3	3 2
3	4	4 3 2
4	+	7 2
5	*	14
6	5	5 14
7	*	70

Notes:
Move 4: an operator is encountered, so 4 and 3 are popped, summed, then pushed back onto stack.
Move 5: operator * is current token, so 7 and 2 are popped, multiplied, pushed back onto stack.
Move 7: stack top holds correct value.
Notice that the postfix notation has been created to properly reflect operator precedence. Thus, postfix expressions never need parentheses.