# *Expression Syntax*

Most programming languages incorporate expression syntax that uses 3 different expression notations: **infix**, **prefix**, and **postfix**. We are familiar with the **infix** notation for expressing algebraic expressions:

**A + B**

In the infix notation, a binary operator is written in between the two operands that it acts upon. When evaluating expressions in infix notation we have to be aware of the precedence and associativity of operators. For example:

A * B - C

Should evaluate multiplication first followed by subtraction. This can actually become quite complex. Two other expression notations **Prefix** notation and **Postfix** notation are parenthesis free, and do not require rules of precedence and associativity when evaluating expressions.

In **Prefix notation**, a binary operator is written in front of the two operands it affects:

**+ A B** = **A + B**

**Method** calls are a form of prefix expression, where the operator is the method name, and the variable length operand list is surrounded by parenthesis and separated by commas:

**methodname (operand1, operand2)**

And in **postfix notation**, the operator follows its operands: **A B +**

Both of these expressions facilitate the execution of expressions, and are parenthesis free. We have seen that we can use a queue to evaluate a prefix expression. As we will see, we can use a stack to quickly and readily evaluate a postfix expression, and convert an infix expression into a postfix expression.

In some cases infix expressions are translated into postfix notation, because they are more readily evaluated using a stack.

## Converting an infix expression to either prefix or postfix by hand

To convert an infix expression to either prefix or postfix expression, one technique that can be done is to fully parenthesize the original expression, and simply move the operator outside of the left or right parentheses of the expression. **During conversion the order of operands (from left to right) in the original expression is preserved.**

SO if our original expression is:

A + ( B * C ) * E – D

Using our understanding of precedence and associativity, we fully parenthesize the expression.

( ( A + ( ( B * C ) * E ) ) – D)

Next the operator is simply moved outside the parenthesis: For prefix, it is moved preceding the LEFT parenthesis, and in postfix it is moved after the RIGHT parenthesis:

**Prefix:**

**( ( A + ( ( B * C ) * E ) ) – D)**

**becomes** **– (+ ( A * (* ( B C ) E ) ) D)**

**Finally remove the parentheses**

- + A * * B C E D

- + A * * B C E D

Assuming A = 2, B = 2, C = 3, D = 4, E = 5

- + 2 * * 2 3 5 4

We evaluate the prefix expression from left to right, we extract each “token” from the string, and when we find an operator followed by two operands, we evaluate it , and put the result back into place:

Step 1: evaluate * 2 3 = 2 * 3 == 6
- + 2 * 6 5 4

Step 2, evaluate * 6 5 = 6 * 5 == 30

- + 2 30 4

Step 3, evaluate + 2 30 = 2 + 30 == 32
- 32 4

Step 4 evaluate – 32 4 = 32 - 4 == 28

***Postfix Expressions***

( ( A + ( ( B * C ) * E ) ) – D)

We fully parenthesize the expression and then move the operator outside the RIGHT parenthesis of each subexpression. The previous expression becomes:

**( ( A ( ( B C ) * E ) *)+ D) –**

remove the parentheses

A B C * E * + D –

Evaluating by hand:

**A B C * E * + D –**

As we scan the expression from left to write, when we find 2 operands followed by an operator, we evaluate the sub-expression, replace the sub-expression with its result, and continue scanning.

Assuming A = 2, B = 2, C = 3, D = 4, E = 5

2 2 3 * 5 * + 4 -

1) Evaluate 2 3 * === 2 * 3== 6
2 6 5 * + 4 -

2) Evaluate 6 5 * === 6 * 5 == 30
2 30 + 4 -

3) Evaluate 2 30 + === 2 + 30 === 32
32 4 -
4) Evaluate 32 - 4 === 28

**Using a stack to evaluate a postfix expression**

A B C * E * + D –

Postfix expressions can easily be evaluated using a stack!!!! For ANY postfix expression, we scan the expression from left to right, identifying operators and operands, using the following rules:

1. IF an operand is found, push the operand onto stack
2. Find an operator is found, we will evaluate a subexpression, **(there must be at least 2 operands on the stack)**
   a) If the size of the stack is less than 2, this is not a valid postfix expression, generate an error
   b) IF size of stack >= 2:
      - pop two operands from the stack, such that the first value popped is operand2, and the second value popped is operand1,

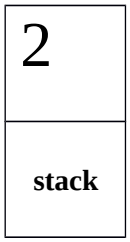
- Evaluate the subexpression as: operand1 op operand2
- push the result of the subexpression back onto the stack

3. When the end of the expression is reached, if the size of the stack is greater than 1, then an error occurred.
4. When the end of the expression is reached, if the size of the stack is 1, pop stack, this is the answer.

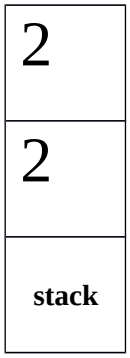
On the next page is an example of this algorithm:

**Assuming A = 2, B = 2, C = 3, D = 4, E = 5** the postfix expression
**A B C * E * + D –**
Can be evaluate as:

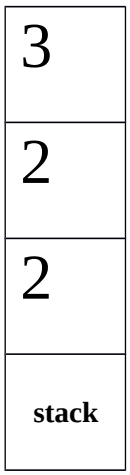
Read A, push 2 on the stack




Read B, push 2

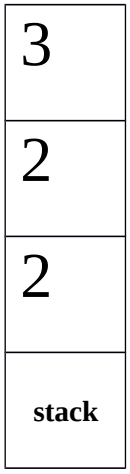



Read C , push 3

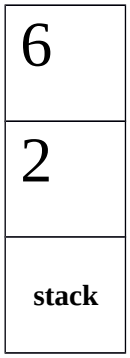



**(Assuming A = 2, B = 2, C = 3, D = 4, E = 5** the postfix expression

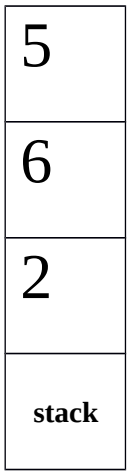
**A B C * E * + D –)**




**Read * , pop 3 into operand2, pop 2 into operand 1, evaluate 2 * 3 push 6 onto stack**

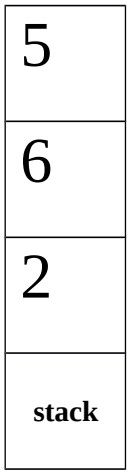



**Read E, push 5**

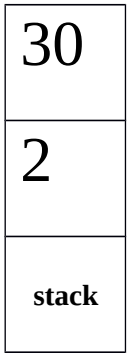



**(Assuming A = 2, B = 2, C = 3, D = 4, E = 5** the postfix expression

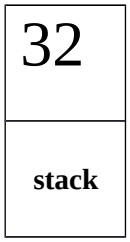
**A B C * E * + D –)**




**Read *, pop 5 into operand2, pop 6 into operand1, evaluate 6 * 5, push 30**

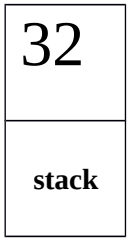



**Read + , pop  30 into operand2, pop 2 into operand 1  evaluate 2 + 30 push 32**

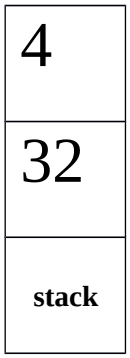



**(Assuming A = 2, B = 2, C = 3, D = 4, E = 5** the postfix expression

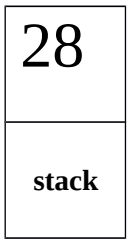
**A B C * E * + D –)**




**Read D, push 4**




**Read – pop 4 into operand2, pop 32 into operand 1, evaluate 30 – 4, push 28,**




**We've reached the end of the expression, one value is on the stack it is our answer.**

**Practice with the expression 10 5 * 6 - 11 + 3 ***
**The answer is 165**