### Postfix Notation

Postfix notation, also known as Reverse Polish Notation (RPN), is a way of writing mathematical expressions where operators follow their operands (numbers or subexpressions). Unlike infix notation (e.g., `3 + 4`), postfix eliminates the need for parentheses by relying on the order of operations. It's commonly used in stack-based calculators and some programming languages.

To evaluate a postfix expression:

1. Scan from left to right.

2. Push operands onto a stack.

3. When an operator is encountered, pop the required number of operands (usually two for binary operators), apply the operator, and push the result back onto the stack.

4. The final result is the single value left on the stack.

### Examples

Here are some simple to complex examples, with their infix equivalents and evaluation steps.

#### 1. Basic Addition: Infix `2 + 3`

- \*\*Postfix\*\*: `2 3 +`

- \*\*Evaluation\*\*:

- Push 2 → Stack: [2]

- Push 3 → Stack: [2, 3]

- +: Pop 3 and 2, compute 2 + 3 = 5, push 5 → Stack: [5]

- \*\*Result\*\*: 5

#### 2. Multiplication: Infix `4 \* 5`

- \*\*Postfix\*\*: `4 5 \*`

- \*\*Evaluation\*\*:

- Push 4 → Stack: [4]

- Push 5 → Stack: [4, 5]

- \*: Pop 5 and 4, compute 4 \* 5 = 20, push 20 → Stack: [20]

- \*\*Result\*\*: 20

#### 3. Mixed Operations: Infix `3 + 4 \* 2` (Precedence: multiplication first)

- \*\*Postfix\*\*: `3 4 2 \* +`

- \*\*Evaluation\*\*:

- Push 3 → Stack: [3]

- Push 4 → Stack: [3, 4]

- Push 2 → Stack: [3, 4, 2]

- \*: Pop 2 and 4, compute 4 \* 2 = 8, push 8 → Stack: [3, 8]

- +: Pop 8 and 3, compute 3 + 8 = 11, push 11 → Stack: [11]

- \*\*Result\*\*: 11

#### 4. With Parentheses: Infix `(3 + 4) \* 2` (Addition first due to parentheses)

- \*\*Postfix\*\*: `3 4 + 2 \*`

- \*\*Evaluation\*\*:

- Push 3 → Stack: [3]

- Push 4 → Stack: [3, 4]

- +: Pop 4 and 3, compute 3 + 4 = 7, push 7 → Stack: [7]

- Push 2 → Stack: [7, 2]

- \*: Pop 2 and 7, compute 7 \* 2 = 14, push 14 → Stack: [14]

- \*\*Result\*\*: 14

#### 5. More Complex: Infix `5 \* (3 + 2) - 4`

- \*\*Postfix\*\*: `5 3 2 + \* 4 -`

- \*\*Evaluation\*\*:

- Push 5 → Stack: [5]

- Push 3 → Stack: [5, 3]

- Push 2 → Stack: [5, 3, 2]

- +: Pop 2 and 3, compute 3 + 2 = 5, push 5 → Stack: [5, 5]

- \*: Pop 5 and 5, compute 5 \* 5 = 25, push 25 → Stack: [25]

- Push 4 → Stack: [25, 4]

- -: Pop 4 and 25, compute 25 - 4 = 21, push 21 → Stack: [21]

- \*\*Result\*\*: 21

#### 6. Exponentiation Example: Infix `2 ^ 3` (Using ^ for power)

- \*\*Postfix\*\*: `2 3 ^` (Note: Exponentiation is right-associative, so `2 3 4 ^ ^` is `2^(3^4)`.)

- \*\*Evaluation\*\*:

- Push 2 → Stack: [2]

- Push 3 → Stack: [2, 3]

- ^: Pop 3 and 2, compute 2^3 = 8, push 8 → Stack: [8]

- \*\*Result\*\*: 8

### Converting Infix to Postfix

Use the Shunting Yard algorithm:

- Output queue for postfix.

- Operator stack.

- Scan infix left to right:

- Operands go to output.

- Operators: While top of stack has higher/equal precedence, pop to output; then push current.

- Left parenthesis: Push to stack.

- Right parenthesis: Pop to output until left parenthesis, then pop parenthesis.

- At end, pop remaining operators to output.

These examples demonstrate how postfix simplifies evaluation without precedence rules or parentheses. For implementation, it's straightforward in code using a stack.