# Math evaluator



## Tips

1. Read the document at least once in full before writing code.
2. Try performing the algorithms by hand so that you understand the algorithm
3. When implementing, test the small steps. **Even** when using the template check if the splitting works as you expect.
4. Use "git commit" whenever you have achieved a good result.

# Background

## Goal

The goal is to create an application that can evaluate math written in the “common way”, which is officially called the “infix notation”. By evaluating the expression we want to calculate an answer. We will calculate the answer by:

1. By splitting the expression into an array of tokens
2. Converting the tokens from an infix notation to a reverse Polish notation
3. Evaluate the Reverse Polish Notation to a result

| Infix notation | Result |
|---|---|
| 9 | 9 |
| 1 + 2 | 3 |
| 1 + 2 × 3 | 7 |
| 1 + 2 × 3 − 4 ÷ 5 | 6,2 |
| 2 ^ 5 × 2 | 64 |
| (1 + 2 ) × ( 2 + 1 ) | 9 |
| ( 1 + ( 2 + 3 − 4 ) ) × ( 5 + 6 ) | 22 |

# Steps

## Step 1: Expression splitting

Before we can process the expression we must split the tokens. If you use the template then you already have the method to split the expression. The below examples are examples of how the values are split.

Once you have accomplished this step (either by writing your own splitter or by using the template) you can move on to the next step. Try and test it with below methods just to be sure.

| Infix notation | Tokens |
|---|---|
| `9` | `[ "9" ]` |
| `1 + 2` | `[ "1", "+", "2" ]` |
| `1 + 2 × 3` | `[ "1", "+", "2", "*", "3" ]` |
| `1 + 2 × 3 − 4 ÷ 5` | `[ "1", "+", "2", "*", "3", "−", "4", "/", "5" ]` |
| `2 ^ 5 × 2` | `[ "2", "^", "5", "*", "2" ]` |
| `(1 + 2 ) × ( 2 + 1 )` | `[ "(", "1", "+", "2", ")", "*", "(", "2", "+", "1", ")" ]` |
| `(1+(2+3−4))×(5+6)` | `[ "(", "1", "+", "(", "2", "+", "3", "−", "4", ")", ")", "*", "(", "5", "+", "6", ")" ]` |

# Step 2: Converting Infix notation to RPN

One way to convert “infix notation” to “reverse polish notation” (hereafter RPN) is by using the Shunting Yard algorithm that takes operator precedence into account. An operator with higher precedence takes priority over an operator with lower precedence.

| Operator symbol | Operator name | Precedence |
|---|---|---|
| **+, -** | Addition, Subtraction | 1 |
| ***, /** | Multiplication, Division | 2 |
| **^** | Power | 3 |

Below is an example in which we also use parentheses to clearly demonstrate the order in which we expect the expression to be solved. The goal is to get the reverse Polish notation on the right.

| Infix notation | Infix with parentheses | Reverse Polish Notation |
|---|---|---|
| `9` | `9` | `9` |
| `1 + 2` | `1 + 2` | `1 2 +` |
| `1 + 2 × 3` | `1 + ( 2 × 3 )` | `1 2 3 × +` |
| `1 + 2 × 3 − 4 ÷ 5` | `1 + ( 2 × 3 ) − ( 4 ÷ 5 )` | `1 2 3 × + 4 5 ÷ −` |
| `2 ^ 5 × 2` | `( 2 ^ 5 ) × 2` | `2 5 ^ 2 ×` |
| `(1 + 2 ) × ( 2 + 1 )` | `(1 + 2 ) × ( 2 + 1 )` | `1 2 + 2 1 + ×` |
| `(1+(2+3−4))×(5+6)` | `(1+(2+3−4))×(5+6)` | `1 2 3 + 4 − + 5 6 + ×` |

Once you have tested all the expressions, you can move on to the next step evaluation.

---

## The procedure

Once split perform the following algorithm by reading the tokens.

1. If the token is numeric put it on the output queue
2. If the token is "`(`" then put it on the operator stack
3. If the token is "`)`" then
    1. *Pop* the top operator of the operator stack
    2. If the top operator is "`(`" then stop and move to the next "token"
    3. Else put the top operator in the output queue and then pop the next operator from the operator stack to repeat.
4. If the token is a normal math operator
    1. Then *Peek* at the top operator from the operator stack
    2. If no top operator is found then add the token to the operator stack and continue to the next token.
    3. If the top operator is "`(`" then add the token to the operator stack and continue to the next token.
    4. Get the operator precedence for the current token and the top operator
    5. If the top operator has a lower precedence than the current operator then put the current operator on the operator stack and continue to the next token.

6. Else put the top operator in the output queue and peek/pop the next top operator.

5. Once all tokens have been parsed
    1. Pop the operator stack and put it in the output queue, repeat this until the operator stack is empty

---

## Example conversion

| # | Infix | Output Queue (OQ) | Operator Stack (OS) | Remarks |
|---|---|---|---|---|
| 0 | (1+(2+3−4))×(5+6) | [ ] | [ ] | Initial state |
| 1 | (1+(2+3−4))×(5+6) | [ ] | [ "(" ] | Put "(" on the stack |
| 2 | (1+(2+3−4))×(5+6) | [ "1" ] | [ "(" ] | Put "1" on the queue |
| 3 | (1+(2+3−4))×(5+6) | [ "1" ] | [ "(", "+" ] | Put "+" on the stack since the topmost was "(" |
| 4 | (1+(2+3−4))×(5+6) | [ "1" ] | [ "(", "+", "(" ] | Put "(" on the stack |
| 5 | (1+(2+3−4))×(5+6) | [ "1", "2" ] | [ "(", "+", "(" ] | Put "2" on the queue |
| 6 | (1+(2+3−4))×(5+6) | [ "1", "2" ] | [ "(", "+", "(", "+" ] | Put "+" on the stack since the topmost was "(" |
| 7 | (1+(2+3−4))×(5+6) | [ "1", "2", "3" ] | [ "(", "+", "(", "+" ] | Put "3" on the queue |
| 8 | (1+(2+3−4))×(5+6) | [ "1", "2", "3", "+" ] | [ "(", "+", "(", "−" ] | Move "+" from stack to queue since "-" has the same precedence, then the topmost is "(" so put "-" on stack |
| 9 | (1+(2+3−4))×(5+6) | [ "1", "2", "3", "+", "4" ] | [ "(", "+", "(", "−" ] | Put "4" on the queue |
| 10 | (1+(2+3−4))×(5+6) | [ "1", "2", "3", "+", "4", "−" ] | [ "(", "+" ] | Pop stack and move to queue until "(" is encountered. Only "-" was stacked before "(" |
| 11 | (1+(2+3−4))×(5+6) | [ "1", "2", "3", "+", "4", "−", "+" ] | [ ] | Pop stack and move to queue until "(" is encountered. Only "+" was stacked before "(" |

| # | Infix | Output Queue (OQ) | Operator Stack (OS) | Remarks |
|---|---|---|---|---|
| 12 | (1+(2+3−4))×(5+6) | [ "1", "2", "3", "+", "4", "−", "+" ] | [ "*" ] | Put "*" on the stack |
| 13 | (1+(2+3−4))×(5+6) | [ "1", "2", "3", "+", "4", "−", "+" ] | [ "*", "(" ] | Put "(" on the stack |
| 14 | (1+(2+3−4))×(5+6) | [ "1", "2", "3", "+", "4", "−", "+", "5" ] | [ "*", "(" ] | Put "5" on the queue |
| 15 | (1+(2+3−4))×(5+6) | [ "1", "2", "3", "+", "4", "−", "+", "5" ] | [ "*", "(", "+" ] | Put "+" on the stack since the topmost was "(" |
| 16 | (1+(2+3−4))×(5+6) | [ "1", "2", "3", "+", "4", "−", "+", "5", "6" ] | [ "*", "(", "+" ] | Put "6" on the queue |
| 17 | (1+(2+3−4))×(5+6) | [ "1", "2", "3", "+", "4", "−", "+", "5", "6", "+" ] | [ "*" ] | Pop stack and move to queue until "(" is encountered. Only "+" was stacked before "(" |
| 18 | (1+(2+3−4))×(5+6) | [ "1", "2", "3", "+", "4", "−", "+", "5", "6", "+", "*" ] | [ ] | No infix tokens, pop stack and move to queue until stack is empty. Only "*" was encountered. |

Notes:
* The "(" and ")" should never be put in the output

## Step 3: Evaluating RPN to result

Once converted to RPN we can calculate the result by reading from left to right using the following rules:

1. If digit, put digit on stack
2. If operator, then pull digits from stack, perform the operation and put the result on the stack

Once every token has of the RPN queue has been processed only a single digit should be on the stack.

---

### Example

| Step | RPN | Stack | |
|---|---|---|---|
| **0** | 1 2 3 × + 4 5 ÷ − | [ ] | Initial state |
| **1** | 1 2 3 × + 4 5 ÷ − | [ 1 ] | Put digit on stack (1) |
| **2** | 1 2 3 × + 4 5 ÷ − | [ 1, 2 ] | Put digit on stack (2) |
| **3** | 1 2 3 × + 4 5 ÷ − | [ 1, 2, 3 ] | Put digit on stack (3) |
| **4** | 1 2 3 × + 4 5 ÷ − | [ 1, 6 ] | Pull last 2 digits (2, 3)<br>Multipy (2 times 3 equals 6)<br>Put back on stack (6) |
| **5** | 1 2 3 × + 4 5 ÷ − | [ 7 ] | Pull last 2 digits (1, 6)<br>Add (1 and 6 equals 7)<br>Put back on stack (7) |
| **6** | 1 2 3 × + 4 5 ÷ − | [ 7, 4 ] | Put digit on stack (4) |
| **7** | 1 2 3 × + 4 5 ÷ − | [ 7, 4, 5 ] | Put digit on stack (5) |
| **8** | 1 2 3 × + 4 5 ÷ − | [ 7, 0.8 ] | Pull last 2 digits (4, 5)<br>Divide (4 divided by 5 equals 0.8)<br>Put back on stack (0.8) |
| **9** | 1 2 3 × + 4 5 ÷ − | [ 6.2 ] | Pull last 2 digits (7, 0.8)<br>Subtract (7 minus 0.8 equals 6.2)<br>Put back on stack (6.2) |

# Code Template

The below code is perfect to start with. It even includes a function to perform the first step.

```
using System.Linq;
using System.Text.RegularExpressions;

namespace Calculator;

public static class Program
{
    public static void Main(string[] args)
    {
        // string[] tokens = GetTokens("1 + 2 * 3 - 4 / 5");
        // string[] rpn = ToRpn(tokens);
        // double result = EvaluateRpn(rpn);
        // Console.WriteLine(result);
    }

    private static string[] GetTokens(string input)
    {
        return InfixMathExpression()
            .Matches(input)
            .Select(x => x.Value.Trim())
            .Where(x => !string.IsNullOrWhiteSpace(x))
            .ToArray();
    }

    private static Regex InfixMathExpression() =>
        new Regex("" +
                  "([0-9]+(?:\\.[0-9]+)?)" + // Numbers
                  "|(\\+)" +                 // + operator
                  "|(-)" +                   // - operator
                  "|(\\*)" +                 // * operator
                  "|(\\/)" +                 // / operator
                  "|(\\^)" +                 // ^ operator
                  "|(\\()" +                 // Open parenthesis
                  "|(\\))"                   // Close parenthesis
        );
}
```