Classic algorithms for expression evaluation

The classic approach to writing code to evaluate arithmetic expressions comprises 3 steps:

- Parsing an infix expression
- Converting an infix expression to a postfix expression
- Evaluating the postfix expression

Note that our discussion of this classic approach will be somewhat simplified: arithmetic expressions will contain only operands, binary operators, and one kind of parentheses. Additionally, all operands and operators will be represented by a single character, making parsing trivial.

Expression notations

The three most common forms of notation in arithmetic expressions are *infix, prefix,* and *postfix* notations. Infix notation is a common way of writing expressions, while prefix and postfix notations are primarily used in computer science.

Infix notation
Infix notation is the conventional notation for arithmetic expressions. It is called *infix* notation because each operator is placed between its operands, which is possible only when an operator has exactly two operands (as in the case with binary operators such as addition, subtraction, multiplication, division, and modulo). When parsing expressions written in infix notation, you need parentheses and precedence rules to remove ambiguity.

```
Syntax: operand1 operator operand2
Example: (A+B)*C-D/(E+F)
```

Prefix notation
In prefix notation, the operator is written before its operands. This notation is frequently used in computer science, especially in compiler design. It is also called *Polish notation* in honor of its inventor, Jan Lukasiewicz.

```
Syntax  : operator operand1 operand2
Example : -*+ABC/D+EF
```

Postfix notation
In postfix notation, the operator comes after its operands. Postfix notation is also known as *reverse Polish notation* (RPN) and is commonly used because it enables easy evaluation of expressions.

```
Syntax  : operand1 operand2 operator
Example : AB+C*DEF+/-
```

Prefix and postfix notation have three common features:

- The operands are in the same order that they would be in the equivalent infix expression.
- Parentheses are not needed.
- The priority of the operators is irrelevant.

Converting infix notation to postfix notation

To convert an expression in an infix expression to its equivalent in postfix notation, we must know the precedence and associativity of operators. *Precedence* or operator strength determines order of evaluation; an operator with higher precedence is evaluated before one of lower precedence. If the operators all have the same precedence, then the order of evaluation depends on their *associativity.* The associativity of an operator defines the order in which operators of the same precedence are grouped (right-to-left or left-to-right).

```
Left associativity  : A+B+C = (A+B)+C
Right associativity : A^B^C = A^(B^C)
```

The conversion process involves reading the operands, operators, and parentheses of the infix expression using the following algorithm:

1. Initialize an empty stack and empty output queue.
2. Read the infix expression from left to right, one symbol at a time.
3. If the symbol is an operand, append it to the output queue.
4. If the symbol is an operator, pop operators from the stack and append them one by one to the output queue until you reach: an opening parenthesis, a left associative operator of (strictly) lower precedence or a right associative operator of lower or equal precedence. Push the operator onto the stack.
5. If the symbol is an opening parenthesis, push it onto the stack.
6. If the symbol is a closing parenthesis, pop all operators until you reach an opening parenthesis and append them to the output queue.
7. If the end of the infix is found, pop all remaining operators from the stack and append them to the output queue.

Postfix expression evaluation

Evaluating a postfix expression is simpler than directly evaluating an infix expression. In postfix notation, the need for parentheses is eliminated and the priority of the operators is no longer relevant. You can use the following algorithm to evaluate postfix expressions:

1. Initialize an empty stack.
2. Read the postfix expression from left to right.
3. If the symbol is an operand, push it onto the stack.
4. If the symbol is an operator, pop two operands, perform the appropriate operation, and then push the result onto the stack. If you could not pop two operators, the syntax of the postfix expression was not correct.
5. At the end of the postfix expression, pop a result from the stack. If the postfix expression was correctly formed, the stack should be empty.

Operators precedence table

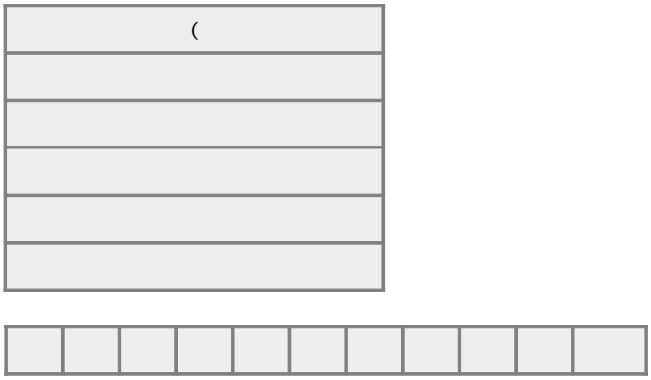
Operator	Precedence	Associativity
-	3	left
+	3	left
*	4	left
/	4	left
^	5	right

Parsing example:

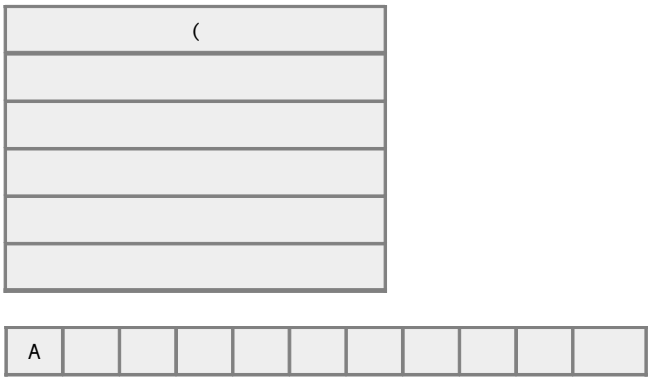
```
Infix: (A+B)*C-D/(E+F)

Postfix: AB+C*DEF+/-
```

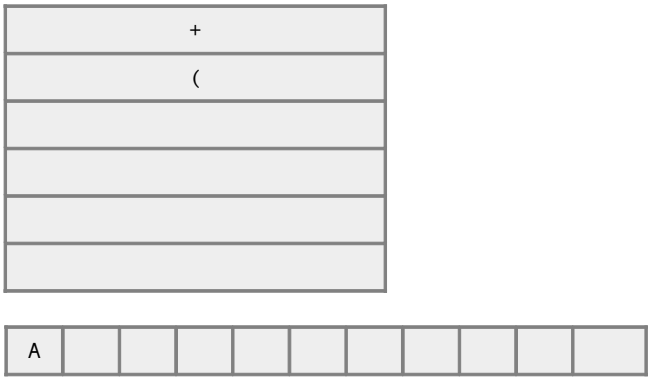
Step 1: (



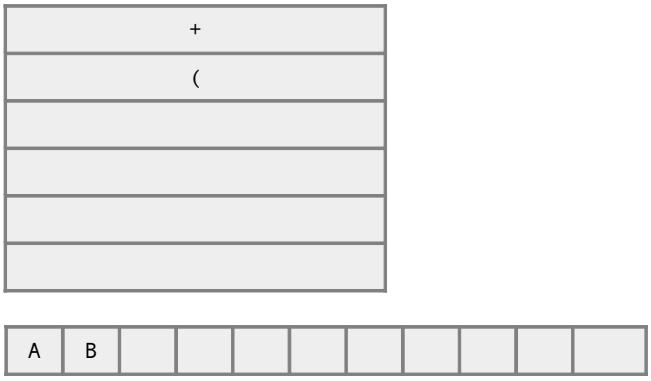
Step 2: A



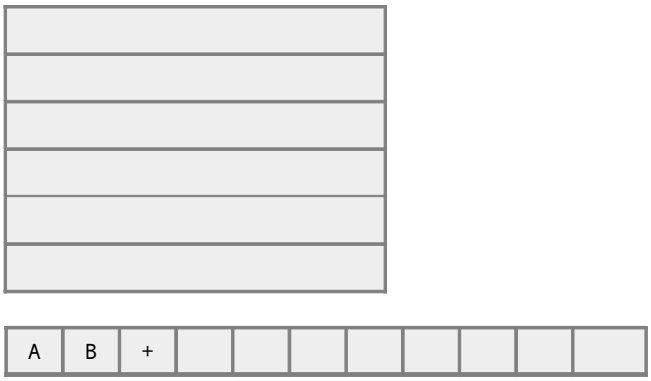
Step 3: +



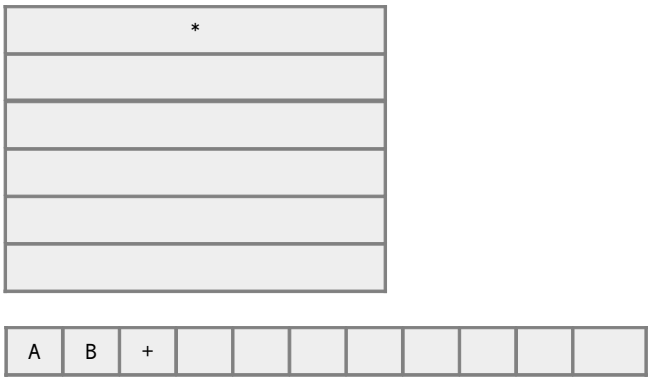
Step 4: B



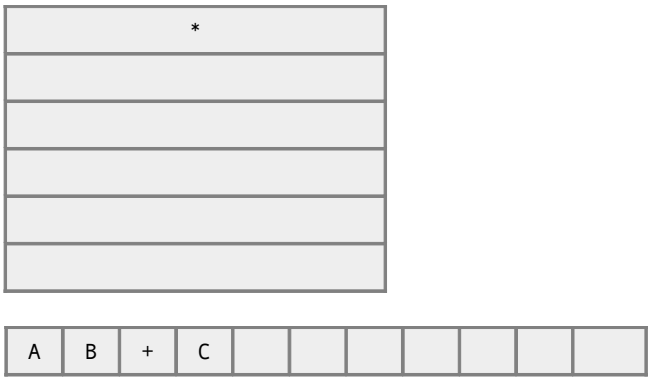
Step 5:)



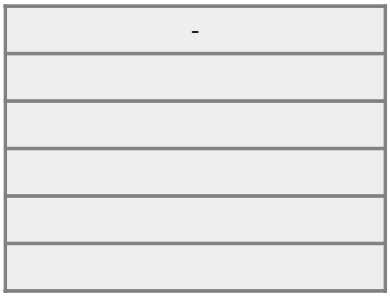
Step 6: *



Step 7: C

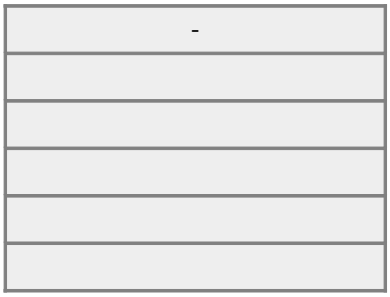


Step 8: -



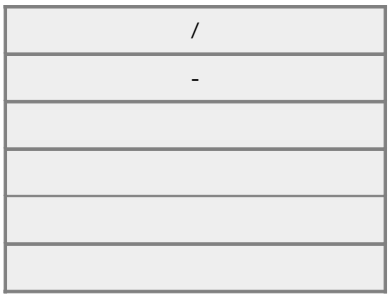
A	B	+	C	*						

Step 9: D



A	B	+	C	*	D					

Step 10: /



A	B	+	C	*	D					

Step 11: (

(
/
-

A	B	+	C	*	D					

Step 12: E

(
/
-

A	B	+	C	*	D	E				

Step 13: +

+
(
/
-

A	B	+	C	*	D	E				

Step 14: F

+
(
/
-

A	B	+	C	*	D	E	F			

Step 15:)

/
-

A	B	+	C	*	D	E	F	+		

Final step:

A	B	+	C	*	D	E	F	+	/	-