WIKIPEDIA

Shunting-yard algorithm

In computer science, the **shunting-yard algorithm** is a method for parsing mathematical expressions specified in infix notation. It can produce either a postfix notation string, also known as Reverse Polish notation (RPN), or an abstract syntax tree (AST).[1] The algorithm was invented by Edsger Dijkstra and named the "shunting yard" algorithm because its operation resembles that of a railroad shunting yard. Dijkstra first described the Shunting Yard Algorithm in the Mathematisch Centrum report MR 34/61 (https://repository.cwi.nl/noauth/search/fullrecord.php?publnr=9251).

Like the evaluation of RPN, the shunting yard algorithm is stack-based. Infix expressions are the form of mathematical notation most people are used to, for instance "3 + 4" or "3 + 4 × (2 − 1)". For the conversion there are two text variables (strings), the input and the output. There is also a stack that holds operators not yet added to the output queue. To convert, the program reads each symbol in order and does something based on that symbol. The result for the above examples would be (in Reverse Polish notation) "3 4 +" and "3 4 2 1 − × +", respectively.

The shunting yard algorithm will correctly parse all valid infix expressions, but does not reject all invalid expressions. For example, "1 2 +" is not a valid infix expression, but would be parsed as "1 + 2". The algorithm can however reject expressions with mismatched parentheses.

The shunting-yard algorithm was later generalized into operator-precedence parsing.

Contents

A simple conversion

1. Input: 3 + 4
2. Push 3 to the output queue (whenever a number is read it is pushed to the output)
3. Push + (or its ID) onto the operator stack
4. Push 4 to the output queue
5. After reading the expression, pop the operators off the stack and add them to the output.

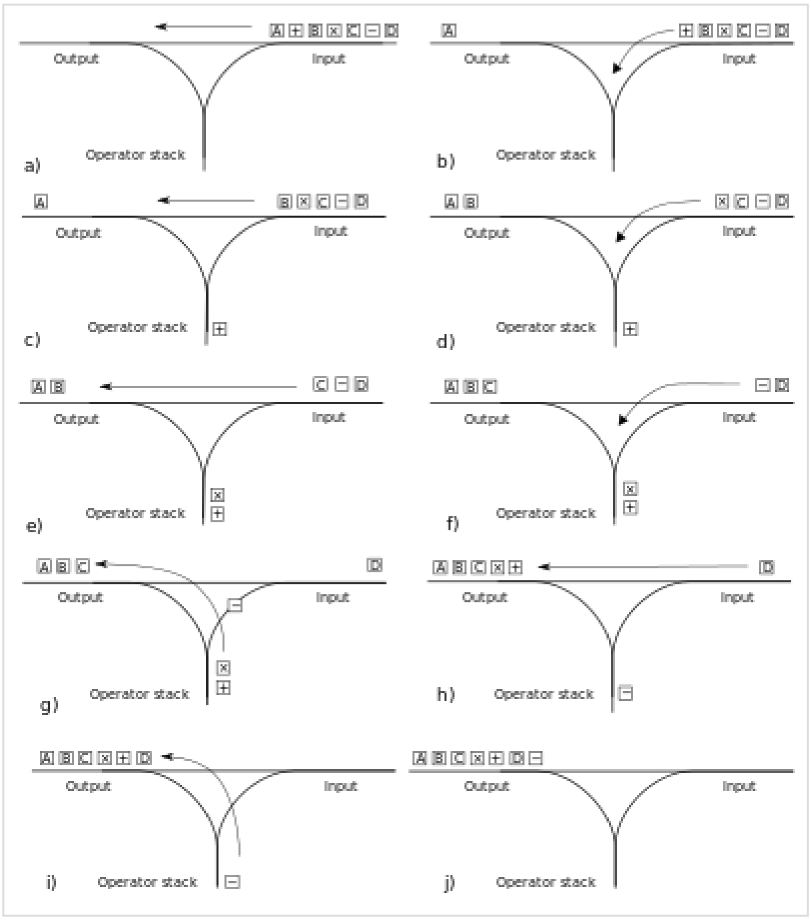
   In this case there is only one, "+".

6. Output: 3 4 +

This already shows a couple of rules:

- All numbers are pushed to the output when they are read.
- At the end of reading the expression, pop all operators off the stack and onto the output.

Graphical illustration



Graphical illustration of algorithm, using a three-way railroad junction. The input is processed one symbol at a time: if a variable or number is found, it is copied directly to the output a), c), e), h). If the symbol is an operator, it is pushed onto the operator stack b), d), f). If the operator's precedence is lower than that of the operators at the top of the stack or the precedents are equal and the operator is left associative, then that operator is popped off the stack and added to the output g). Finally, any remaining operators are popped off the stack and added to the output i).

The algorithm in detail

Important terms: Token, Function, Operator associativity, Precedence

```
/* This implementation does not implement composite functions,functions with variable number of
arguments, and unary operators. */

while there are tokens to be read:
    read a token.
    if the token is a number, then:
        push it to the output queue.
    else if the token is a function then:
        push it onto the operator stack
    else if the token is an operator then:
        while ((there is an operator at the top of the operator stack)
              and ((the operator at the top of the operator stack has greater precedence)
                  or (the operator at the top of the operator stack has equal precedence and the
token is left associative))
              and (the operator at the top of the operator stack is not a left parenthesis)):
            pop operators from the operator stack onto the output queue.
        push it onto the operator stack.
    else if the token is a left parenthesis (i.e. "("), then:
        push it onto the operator stack.
    else if the token is a right parenthesis (i.e. ")"), then:
        while the operator at the top of the operator stack is not a left parenthesis:
            pop the operator from the operator stack onto the output queue.
        /* If the stack runs out without finding a left parenthesis, then there are mismatched
parentheses. */
        if there is a left parenthesis at the top of the operator stack, then:
            pop the operator from the operator stack and discard it
        if there is a function token at the top of the operator stack, then:
            pop the function from the operator stack onto the output queue.
/* After while loop, if operator stack not null, pop everything to output queue */
if there are no more tokens to read then:
    while there are still operator tokens on the stack:
        /* If the operator token on the top of the stack is a parenthesis, then there are
mismatched parentheses. */
        pop the operator from the operator stack onto the output queue.
exit.
```

To analyze the running time complexity of this algorithm, one has only to note that each token will be read once, each number, function, or operator will be printed once, and each function, operator, or parenthesis will be pushed onto the stack and popped off the stack once—therefore, there are at most a constant number of operations executed per token, and the running time is thus O(*n*)—linear in the size of the input.

The shunting yard algorithm can also be applied to produce prefix notation (also known as Polish notation). To do this one would simply start from the end of a string of tokens to be parsed and work backwards, reverse the output queue (therefore making the output queue an output stack), and flip the left and right parenthesis behavior (remembering that the now-left parenthesis behavior should pop until it finds a now-right parenthesis). And changing the associativity condition to right.

Detailed example

Input: 3 + 4 × 2 ÷ (1 − 5) ^ 2 ^ 3

Operator	Precedence	Associativity
^	4	Right
×	3	Left
÷	3	Left
+	2	Left
−	2	Left

The symbol ^ represents the power operator.

Token	Action	Output (in RPN)	Operator stack	Notes
3	Add token to output	3		
+	Push token to stack	3	+	
4	Add token to output	3 4	+	
×	Push token to stack	3 4	× +	× has higher precedence than +
2	Add token to output	3 4 2	× +	
÷	Pop stack to output	3 4 2 ×	+	÷ and × have same precedence
	Push token to stack	3 4 2 ×	÷ +	÷ has higher precedence than +
(	Push token to stack	3 4 2 ×	(÷ +	
1	Add token to output	3 4 2 × 1	(÷ +	
−	Push token to stack	3 4 2 × 1	− (÷ +	
5	Add token to output	3 4 2 × 1 5	− (÷ +	
)	Pop stack to output	3 4 2 × 1 5 −	(÷ +	Repeated until "(" found
	Pop stack	3 4 2 × 1 5 −	÷ +	Discard matching parenthesis
^	Push token to stack	3 4 2 × 1 5 −	^ ÷ +	^ has higher precedence than ÷
2	Add token to output	3 4 2 × 1 5 − 2	^ ÷ +	
^	Push token to stack	3 4 2 × 1 5 − 2	^ ^ ÷ +	^ is evaluated right-to-left
3	Add token to output	3 4 2 × 1 5 − 2 3	^ ^ ÷ +	
end	Pop entire stack to output	3 4 2 × 1 5 − 2 3 ^ ^ ÷ +		

Input: sin (max (2, 3) ÷ 3 × π)

Token	Action	Output = (in RPN)	Operator stack	Notes
sin	Push token to stack		sin	
(	Push token to stack		(sin	
max	Push token to stack		max (sin	
(	Push token to stack		(max (sin	
2	Add token to output	2	(max (sin	
,	ignore	2	(max (sin	
3	Add token to output	2 3	(max (sin	
)	pop stack to output	2 3	(max (sin	Repeated until "(" is at the top of the stack
	Pop stack	2 3	max (sin	Discarding matching parentheses
÷	Pop stack to output	2 3 max	(sin	
	Push token to stack	2 3 max	÷ (sin	
3	Add token to output	2 3 max 3	÷ (sin	
×	Pop stack to output	2 3 max 3 ÷	(sin	
	Push token to stack	2 3 max 3 ÷	× (sin	
π	Add token to output	2 3 max 3 ÷ π	× (sin	
)	Pop stack to output	2 3 max 3 ÷ π ×	(sin	Repeated until "(" is at the top of the stack
	Pop stack	2 3 max 3 ÷ π ×	sin	Discarding matching parentheses
end	Pop entire stack to output	2 3 max 3 ÷ π × sin		

See also

- Operator-precedence parser
- Stack-sortable permutation

References

1. Theodore Norvell (1999). "Parsing Expressions by Recursive Descent" (http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm). *www.engr.mun.ca*. Retrieved 2020-12-28.

External links

- Dijkstra's original description of the Shunting yard algorithm (http://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF)
- Literate Programs implementation in C (https://web.archive.org/web/20110718214204/http://en.literateprograms.org/Shunting_yard_algorithm_(C))
- Java Applet demonstrating the Shunting yard algorithm (http://www.chris-j.co.uk/parsing.php)

- Silverlight widget demonstrating the Shunting yard algorithm and evaluation of arithmetic expressions (http://www.codeding.com/?article=11)
- Parsing Expressions by Recursive Descent (http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm) Theodore Norvell © 1999–2001. Access date September 14, 2006.
- Matlab code, evaluation of arithmetic expressions using the shunting yard algorithm (https://nl.mathworks.com/matlabcentral/fileexchange/68458-evaluation)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Shunting-yard_algorithm&oldid=1002380861"

This page was last edited on 24 January 2021, at 04:16 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.