WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Čeština
Deutsch
Español
한국어
Nederlands
日本語
Русский
Svenska
Українська
中文
Edit links

Shunting-yard algorithm

From Wikipedia, the free encyclopedia
(Redirected from Shunting yard algorithm)

This article includes a list of references, related reading or external links, **but its sources remain unclear because it lacks inline citations**. Please improve this article by introducing more precise citations. *(August 2013)*

In computer science, the **shunting-yard algorithm** is a method for parsing mathematical expressions specified in infix notation. It can be used to produce output in Reverse Polish notation (RPN) or as an abstract syntax tree (AST). 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.

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 shunting-yard algorithm has been later generalized into operator-precedence parsing.

Contents [hide]

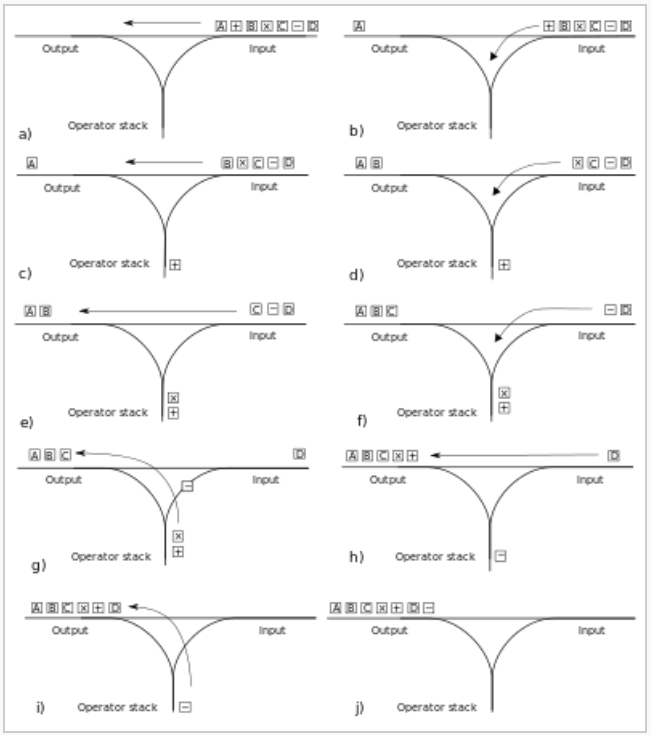
A simple conversion [edit]

Input: 3+4

1. Add 3 to the output queue (whenever a number is read it is added to the output)
2. Push + (or its ID) onto the operator stack
3. Add 4 to the output queue
4. After reading the expression, pop the operators off the stack and add them to the output.
5. In this case there is only one, "+".
6. Output 3 4 +

This already shows a couple of rules:

- All numbers are added 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 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 direct 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 less than that of the operators at the top of the stack or the

The algorithm in detail [edit]

- While there are tokens to be read:
  - Read a token.
  - If the token is a number, then add it to the output queue.

precedences 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).

- If the token is a function token, then push it onto the stack.
- If the token is a function argument separator (e.g., a comma):
  - Until the token at the top of the stack is a left parenthesis, pop operators off the stack onto the output queue. If no left parentheses are encountered, either the separator was misplaced or parentheses were mismatched.
- If the token is an operator, o_1, then:
  - while there is an operator token, o_2, at the top of the operator stack, and either
    o_1 is left-associative and its precedence is *less than or equal* to that of o_2, or
    o_1 is right associative, and has precedence *less than* that of o_2,

    then pop o_2 off the operator stack, onto the output queue;
  - push o_1 onto the operator stack.
- If the token is a left parenthesis (i.e. "("), then push it onto the stack.
- If the token is a right parenthesis (i.e. ")"):
  - Until the token at the top of the stack is a left parenthesis, pop operators off the stack onto the output queue.
  - Pop the left parenthesis from the stack, but not onto the output queue.
  - If the token at the top of the stack is a function token, pop it onto the output queue.
  - If the stack runs out without finding a left parenthesis, then there are mismatched parentheses.
- When there are no more tokens to read:
  - While there are still operator tokens in the stack:
    - If the operator token on the top of the stack is a parenthesis, then there are mismatched parentheses.
    - Pop the operator 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).

Detailed example [edit]

Input: 3 + 4 * 2 / (1 - 5) ^ 2 ^ 3

operator	precedence	associativity
^	4	Right
*	3	Left
/	3	Left
+	2	Left
−	2	Left

Note: The symbol ^ represents the power operator (it doesn't represent XOR).

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 * 3.1415

Token	Action	Output (in RPN)	Operator Stack	Notes
sin	Push token to stack		sin	
max	Push token to stack		max sin	
2	Add token to output	2	max sin	
3	Add token to output	2 3	max sin	
/	Pop token to output	2 3 max	/ sin	
3	Add token to output	2 3 max 3	/ sin	
*	Pop token to output	2 3 max 3 /	* sin	
3.1415	Add token to output	2 3 max 3 / 3.1415	* sin	
end	Pop entire stack to output	2 3 max 3 / 3.1415 * sin		

If one was writing an interpreter, this output would be tokenized and written to a compiled file to be later interpreted. Conversion from infix to RPN can also allow for easier simplification of expressions. To do this, act like one is solving the RPN expression, however, whenever one come to a variable its value is null, and whenever an operator has a null value, it and its parameters are written to the output (this is a simplification, problems arise when the parameters are operators). When an operator has no null parameters its value can simply be written to the output. This method obviously doesn't include all the simplifications possible: It's more of a constant folding optimization.

A good problem to try on this is on SPOJ : http://www.spoj.com/problems/ONP/

See also [edit]

- Operator-precedence parser

External links [edit]

- Dijkstra's original description of the Shunting yard algorithm
- Literate Programs implementation in C
- Implementation in various languages, including C and Python
- Java Applet demonstrating the Shunting yard algorithm
- Silverlight widget demonstrating the Shunting yard algorithm and evaluation of arithmetic expressions
- Parsing Expressions by Recursive Descent Theodore Norvell © 1999–2001. Access date September 14, 2006.
- Extension to the 'Shunting Yard' algorithm to allow variable numbers of arguments to functions
- Java implementation of the Shunting yard algorithm
- Another Java implementation of the Shunting yard algorithm
- A Python implementation of the Shunting yard algorithm
- A Swift implementation of the Shunting yard algorithm

- A GNU Guile implementation of the Shunting yard algorithm

Categories: Parsing algorithms | Dutch inventions

This page was last modified on 10 August 2015, at 21:48.

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.

Privacy policy About Wikipedia Disclaimers Contact Wikipedia Developers Mobile view