Shunting Yard algorithm for parsing Maths expressions

EE24BTECH11001; Collaborated with EE24BTECH11002 and EE24BTECH11005

March 24, 2025

1 Shunting Yard Algorithm Algorithm Overview

The Shunting Yard algorithm uses two main data structures:

- A stack for temporarily storing operators
- An output queue for the final postfix expression

The algorithm processes each token in the input expression sequentially, following specific rules based on the token type[3].

1.1 Key Terms

- **Token**: A number, operator, function, or parenthesis in the expression
- **Operator**: Mathematical symbols representing operations $(+, -, *, /, \vee)$
- **Stack**: Last-In-First-Out (LIFO) data structure
- **Queue**: First-In-First-Out (FIFO) data structure

2 Algorithm Steps

1. Initialize an empty stack and an empty output queue
2. For each token in the input expression:
    - If the token is a number, add it to the output queue
    - If the token is a function, push it onto the stack
    - If the token is an operator:
        - While there is an operator at the top of the stack with greater precedence, or equal precedence and left associativity, pop it to the output queue
        - Push the current operator onto the stack
    - If the token is a left parenthesis, push it onto the stack
    - If the token is a right parenthesis:

- Pop operators from the stack to the output queue until a left parenthesis is encountered
- Discard the left parenthesis

3. After all tokens have been processed, pop any remaining operators from the stack to the output queue[6]

3 Operator Precedence and Associativity

The algorithm handles operator precedence and associativity according to standard mathematical rules. Here's a typical precedence table[2]:

Operator	**Precedence**	**Associativity**
$\vee$ (power)	4	Right
* (multiply)	3	Left
/ (divide)	3	Left
+ (add)	2	Left
- (subtract)	2	Left

Table 1: Operator Precedence and Associativity

Left-associative operators (like +, -, *, /) are evaluated from left to right, while right-associative operators (like $\vee$) are evaluated from right to left[2].

4 Example Walkthrough

Let's trace through the algorithm with the expression: $3 + 4 * 2/(1 - 5)$[1]

Token	**Action**	**Stack**	**Output Queue**
3	Add to output		3
+	Push to stack	+	3
4	Add to output	+	3 4
*	Push to stack (higher precedence than +)	+ *	3 4
2	Add to output	+ *	3 4 2
/	Pop * (equal precedence), push /	+ /	3 4 2 *
(	Push to stack	+ / (	3 4 2 *
1	Add to output	+ / (	3 4 2 * 1
-	Push to stack	+ / (-	3 4 2 * 1
5	Add to output	+ / (-	3 4 2 * 1 5
)	Pop until (	+ /	3 4 2 * 1 5 -
End	Pop remaining operators		3 4 2 * 1 5 - / +

Table 2: Step-by-step Execution of the Shunting Yard Algorithm

The resulting postfix expression is: 3 4 2 $*$ 1 5 $-$ / +[6]

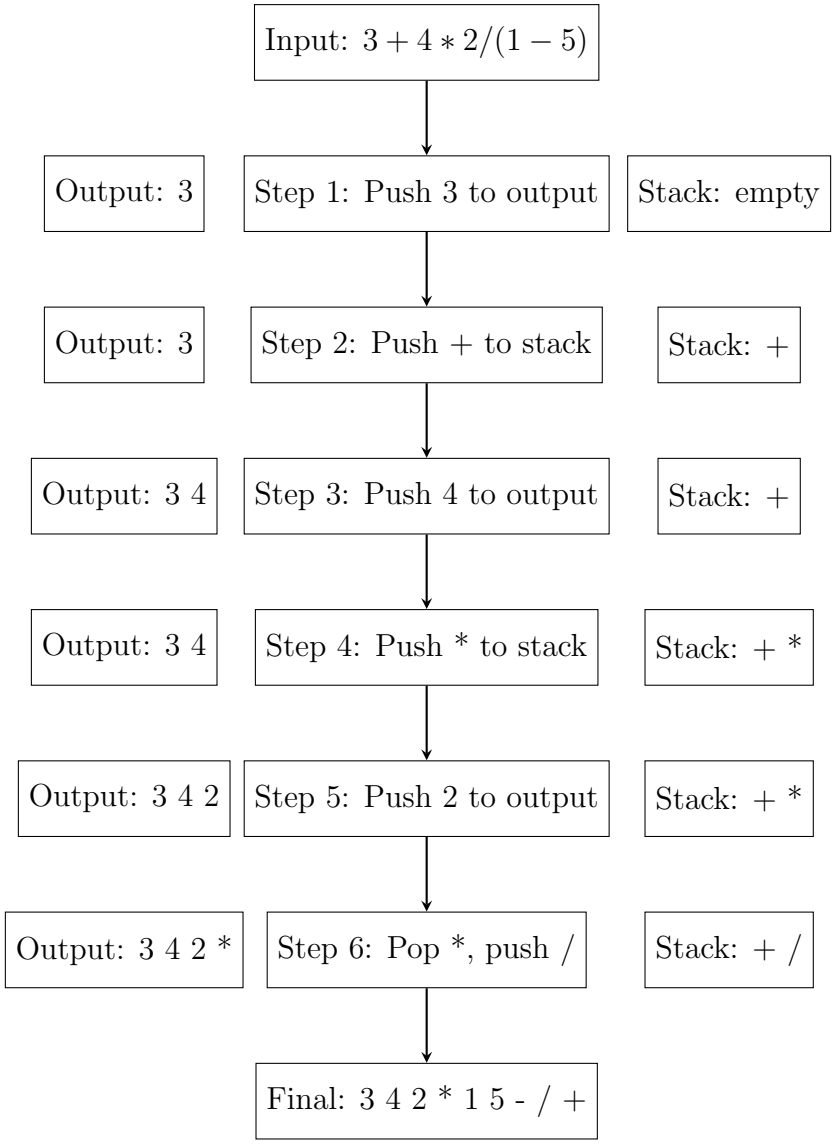

Figure 1: Visual Representation of the Example

5 Implementing BODMAS Rule

The BODMAS rule (Brackets, Orders, Division, Multiplication, Addition, Subtraction) is inherently implemented in the Shunting Yard algorithm through operator precedence. The algorithm processes operations in the correct order by:

1. Handling brackets by pushing left brackets onto the stack and popping operators until the matching right bracket is found
2. Assigning higher precedence to multiplication and division than addition and subtraction
3. Processing operators of equal precedence from left to right (for left-associative operators)[4]

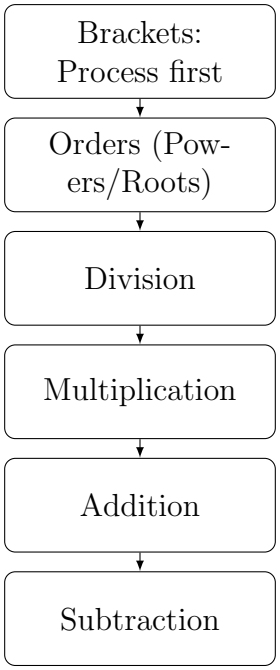

Figure 2: BODMAS Rule Implementation

6 Function Implementation

To implement functions like sin, cos, etc., the algorithm needs to be extended:

1. When a function token is encountered, push it onto the operator stack
2. When processing a function's arguments (which may be separated by commas), handle them appropriately
3. When the function's closing parenthesis is encountered, pop the function from the stack to the output queue[6]

For example, to handle $\sin(x)$ or $\max(x, y)$, the function name is pushed onto the stack when encountered, and popped to the output queue after its arguments have been processed.

7 Bracket Handling

Brackets are crucial for overriding the default operator precedence. The algorithm handles them as follows:

1. When a left bracket '(' is encountered, it is pushed onto the stack
2. When a right bracket ')' is encountered:
   - Pop operators from the stack to the output queue until a left bracket is found
   - Discard the left bracket
   - If the token at the top of the stack is a function, pop it to the output queue

This ensures that expressions within brackets are evaluated first, respecting the BODMAS rule.

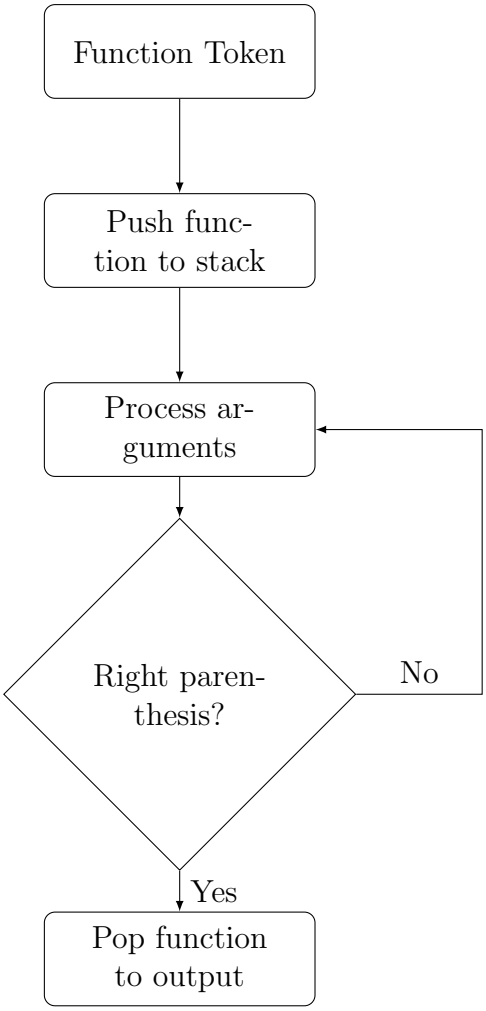

Figure 3: Function Handling in the Shunting Yard Algorithm

8 Complexity Analysis

The Shunting Yard algorithm has linear time complexity $O(n)$, where n is the number of tokens in the input expression. Each token is processed exactly once, and each operation (push, pop) takes constant time.

9 Evaluating Reverse Polish Notation

After converting an infix expression to Reverse Polish Notation (RPN) using the Shunting Yard Algorithm, the next step is to evaluate the RPN expression. This process is straightforward and efficient, making RPN particularly useful for expression evaluation in computing applications.

9.1 Understanding RPN

RPN, also known as postfix notation, is a mathematical notation where operators follow their operands. For example, the infix expression "3 + 4" is written as "3 4 +" in RPN. This notation eliminates the need for parentheses and simplifies the evaluation process.

9.2 Evaluation Algorithm

The algorithm for evaluating RPN expressions uses a stack data structure and follows these steps:

1. Initialize an empty stack.
2. Read the RPN expression from left to right:
   - If the token is an operand (number), push it onto the stack.
   - If the token is an operator, pop the required number of operands from the stack, perform the operation, and push the result back onto the stack.
3. After processing all tokens, the final result will be the only value remaining on the stack.

9.3 Example Evaluation

Let's evaluate the RPN expression resulting from our previous example:

$$3\ 4\ 2 * 1\ 5 - 2\ 3\ \hat{}\ \hat{}\ /\ +$$

Token	**Stack (after processing)**	**Explanation**
3		Push 3 onto the stack
4		Push 4 onto the stack
2		Push 2 onto the stack
		Pop 4 and 2, multiply: $4 \times 2 = 8$, pus
1		Push 1 onto the stack
5		Push 5 onto the stack
	$[3, 8, -4]$	Pop 1 and 5, subtract: $1 - 5 = -4$, pu
2	$[3, 8, -4, 2]$	Push 2 onto the stack
3	$[3, 8, -4, 2, 3]$	Push 3 onto the stack
$\vee$	$[3, 8, -4, 8]$	Pop 2 and 3, exponentiate: $2^3 = 8$, pu
$\vee$	[65536]	Pop -4 and 8, exponentiate: $(-4)^8 = 65536$
/		Pop 8 and 65536, divide: $8/65536 = 0$ (integer di
=¿	Pop 3 and 0, add: $3 + 0 = 3$, push result	

Table 3: Step-by-step evaluation of the RPN expression

Final result: 3

9.4 Advantages of RPN Evaluation

Evaluating RPN expressions offers several advantages:

1. **Simplicity**: The evaluation algorithm is straightforward and easy to implement.
2. **Efficiency**: RPN can be evaluated in a single pass from left to right, without the need for backtracking or complex parsing.

3. **No parentheses**: RPN eliminates the need for parentheses, reducing complexity and potential errors.

4. **Stack-based**: The stack-based approach aligns well with computer architecture, potentially leading to more efficient implementations.

10 Connecting Shunting Yard and RPN Evaluation

The Shunting Yard Algorithm and RPN evaluation work together to provide a complete solution for parsing and evaluating mathematical expressions:

1. The Shunting Yard Algorithm converts infix notation to RPN, handling operator precedence and associativity.

2. The RPN evaluation algorithm then processes the resulting postfix expression to compute the final result.

This two-step process allows for efficient and unambiguous evaluation of complex mathematical expressions, making it valuable in various applications such as calculators, compilers, and mathematical software.

11 Conclusion

The combination of the Shunting Yard Algorithm for infix-to-postfix conversion and the stack-based RPN evaluation method provides a robust and efficient approach to expression parsing and evaluation. This approach handles operator precedence, associativity, and nested expressions with ease, while offering a straightforward evaluation process.

By understanding and implementing these algorithms, developers can create powerful tools for mathematical expression handling, benefiting applications in fields such as computer algebra systems, calculators, and programming language design.