STACKS APPLICATIONS

INFIX NOTATION

- **Infix notation** is the common arithmetic and logical formula notation, in which **operators are written** infix-style **between the operands** they act on
- E.g. **A + B**

POSTFIX NOTATION

- In Postfix notation, the **operator** comes **after the Operand**.

- For example, the Infix expression **A+B** will be written as **AB+** in its **Postfix Notation**.

- Postfix is also called ‘**Reverse Polish Notation**’

PREFIX NOTATION

- In Prefix notation, the **operator** comes **before the operand**.
- The Infix expression **A+B** will be written as **+AB** in its Prefix Notation.
- Prefix is also called ‘**Polish Notation**’

BUILDING AN ARITHMETIC EXPRESSION

Postfix Expression String Processing

*Assume 1-digit integer operands, the binary operators + - * / only, and the string to be evaluated is properly formed*

Rules for processing the postfix string:

Starting from the left hand end, inspect each character of the string

1. if it's an operand – push it on the stack
2. if it's an operator – remove the top 2 operands from the stack, perform the indicated operation, and push the result on the stack

An Example: `3*(4+5)/2` → `345+*2/` → `13`

Remaining Postfix String	int Stack (top→)	Rule Used
`345+*2/`	`empty`	
`45+*2/`	`3`	1
`5+*2/`	`3 4`	1
`+*2/`	`3 4 5`	1
`*2/`	`3 9`	2
`2/`	`27`	2
`/`	`27 2`	1
`null`	`13`	2

value of expression at top of stack

CONVERSION FROM INFIX TO POSTFIX ALGORITHM

Step1

- **Scan the Infix expression** from **left to right** for tokens (Operators, Operands & Parentheses) and perform the steps 2 to 5 for each token in the Expression

ALGORITHM

Step2

- ⋆If token is **operand, Append it** in postfix expression

Step3

- ⋆If token is a **left parentheses “(“**, **push it** in stack.

ALGORITHM

Step4

⋆If token is an **operator**,

- **Pop all the operators** which are of higher or equal precedence then the incoming token and **append them** (in the same order) to the output Expression.
- After popping out all such operators, **push the new token** on stack.

ALGORITHM

Step5

- If **“)”** right parentheses is found,
  - **Pop all the operators** from the Stack and append them to Output String, **till** you **encounter the Opening Parenthesis “(“**.
  - **Pop the left parenthesis** but don’t append it to the output string (Postfix notation does not have brackets).

ALGORITHM

Step6

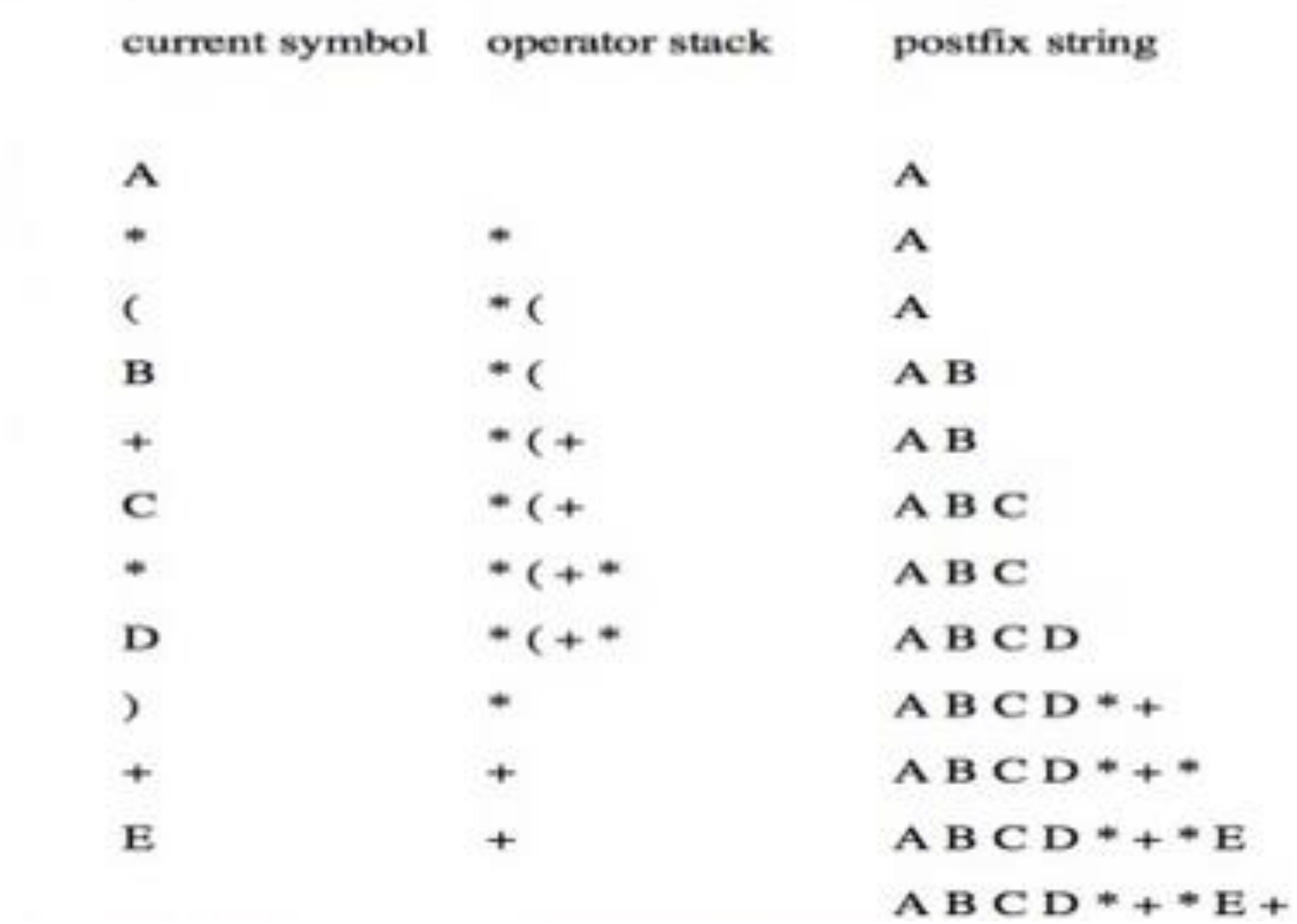
- ⋆When all tokens of Infix expression have been scanned. **Pop all the elements from the stack** and **append** them to the Output String.
- ⋆The Output string is the Corresponding **Postfix Notation**.

EXAMPLE

An Example: 7-(2*3+5)*(8-4/2) → 723*5+842/-*-

Remaining Infix String	char Stack	Postfix String	Rule Used
7-(2*3+5)*(8-4/2)	empty	null	
-(2*3+5)*(8-4/2)	empty	7	1
(2*3+5)*(8-4/2)	-	7	3
2*3+5)*(8-4/2)	-(	7	2
3+5)(8-4/2)	-(	72	1
3+5)*(8-4/2)	-(*	72	3
+5)*(8-4/2)	-(*	723	3
5)*(8-4/2)	-(+	723*	3
)*(8-4/2)	-(+	723*5	1
*(8-4/2)	-	723*5+	4
(8-4/2)	-*	723*5+	3
8-4/2)	-*(	723*5+	2
-4/2)	-*(	723*5+8	1
4/2)	-*(-	723*5+8	3
/2)	-*(-	723*5+84	1
2)	-*(-/	723*5+84	3
)	-*(-/	723*5+842	1
null	empty	723*5+842/-*-	4&5

Example A * (B + C * D) + E becomes A B C D * + * E +



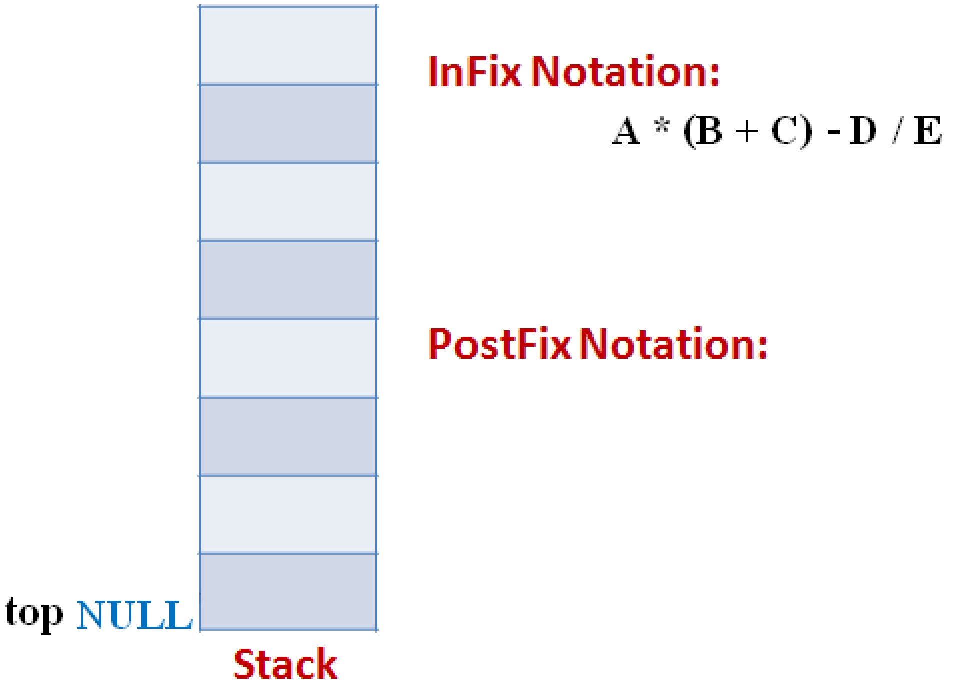
current symbol	operator stack	postfix string
A		A
*	*	A
(	* (	A
B	* (	A B
+	* (+	A B
C	* (+	A B C
*	* (+ *	A B C
D	* (+ *	A B C D
)	*	A B C D * +
+	+	A B C D * + *
E	+	A B C D * + * E
		A B C D * + * E +

EXAMPLE

- Let the incoming the Infix expression be:

A * (B + C) – D / E

Stage 1: **Stack is empty** and we only have the Infix Expression.



InFix Notation:

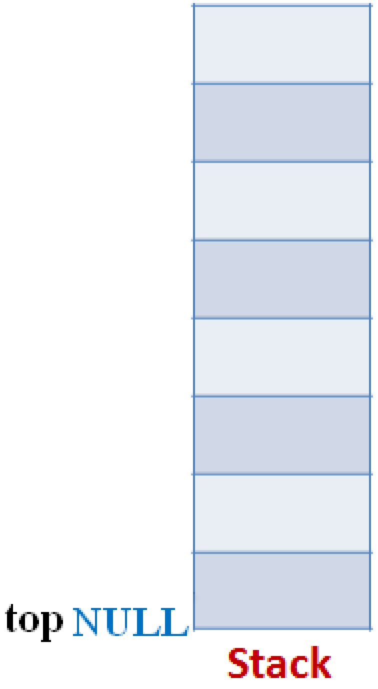
A * (B + C) - D / E

PostFix Notation:

EXAMPLE

Stage 2

- ⋆The first token is **Operand A** Operands are Appended to the Output as it is.



InFix Notation:

* (B + C) - D / E

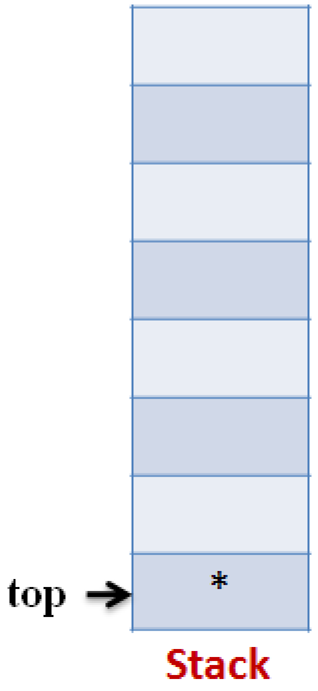
PostFix Notation:

A

EXAMPLE

Stage 3

⋆Next token is ***** Since **Stack is empty (top==NULL)** it is **pushed into the Stack**



InFix Notation:

(B + C) - D / E

PostFix Notation:

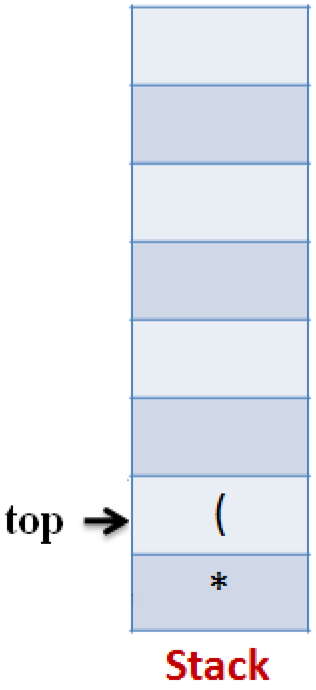
A

EXAMPLE

Stage 4

- Next token is (the precedence of open-parenthesis, when it is to go inside, is maximum.

- But when another operator is to come on the top of '(' then its precedence is least.



InFix Notation:

B + C) - D / E

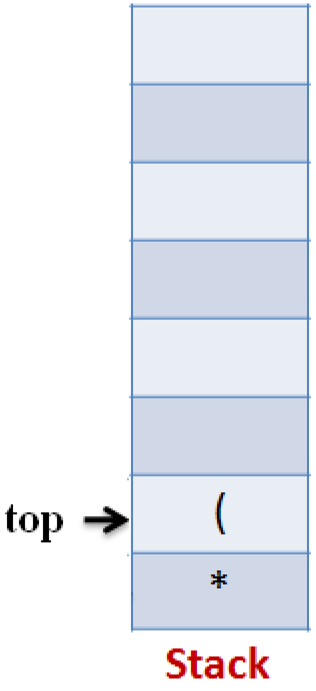
PostFix Notation:

A

EXAMPLE

Stage 5

- Next token, **B** is an operand which will go to the Output expression as it is



InFix Notation:

+ C) - D / E

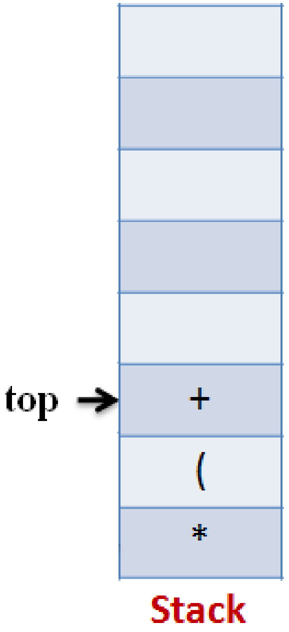
PostFix Notation:

A B

EXAMPLE

Stage 6

- ⋆Next token, **+** is operator, We consider the precedence of **top element in the Stack**, **‘(‘**. The outgoing precedence of open parenthesis is the least (refer point 4. Above). So **+** gets **pushed into the Stack**



InFix Notation:

C) - D / E

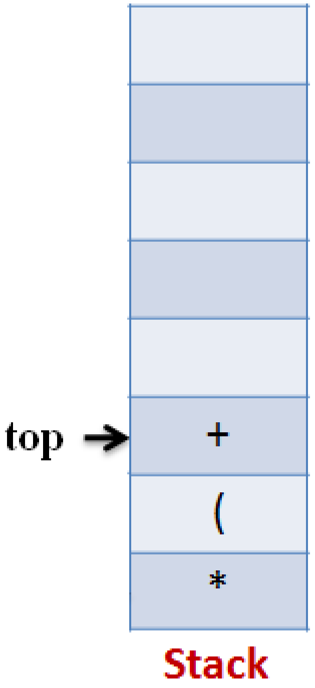
PostFix Notation:

A B

EXAMPLE

Stage 7

- Next token, **C**, is appended to the output



InFix Notation:

) - D / E

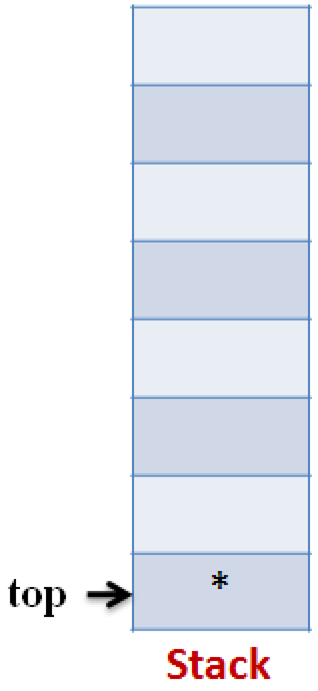
PostFix Notation:

A B C

EXAMPLE

Stage 8

- ⋆Next token **)**, means that **pop all the elements from Stack** and **append them to the output** expression till we read an opening parenthesis.



InFix Notation:

- D / E

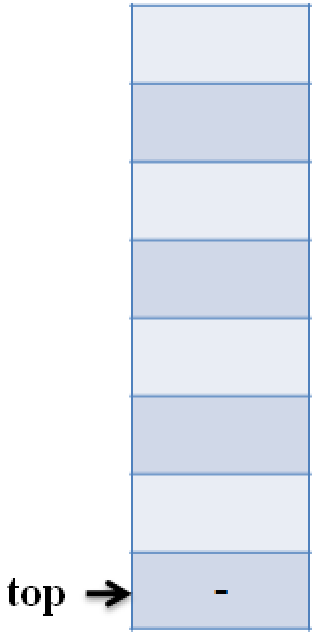
PostFix Notation:

A B C +

EXAMPLE

Stage 9

- Next token, **-**, is an operator. The precedence of operator on the top of Stack ‘*****‘ is more than that of Minus. So we **pop multiply** and **append it to output** expression. Then **push minus in the Stack**.



Stack

InFix Notation:

D / E

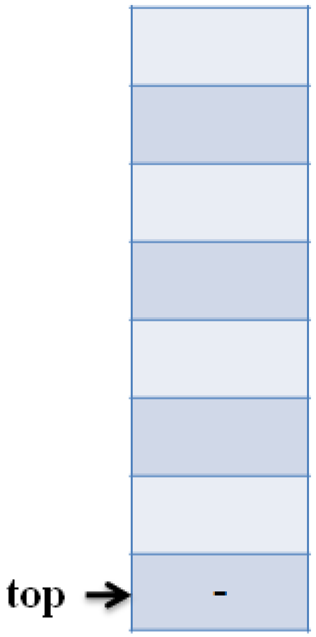
PostFix Notation:

A B C + *

EXAMPLE

Stage 10

- ⋆Next, Operand '**D**' gets **appended to the output**.



Stack

InFix Notation:

/ E

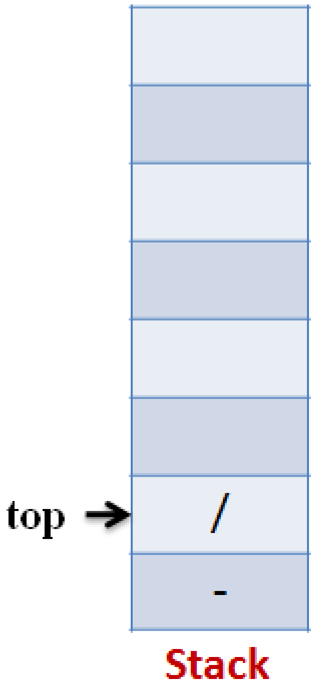
PostFix Notation:

A B C + * D

EXAMPLE

Stage 11

- Next, we will insert the **division** operator into the Stack because its precedence is more than that of minus.



InFix Notation:

E

PostFix Notation:

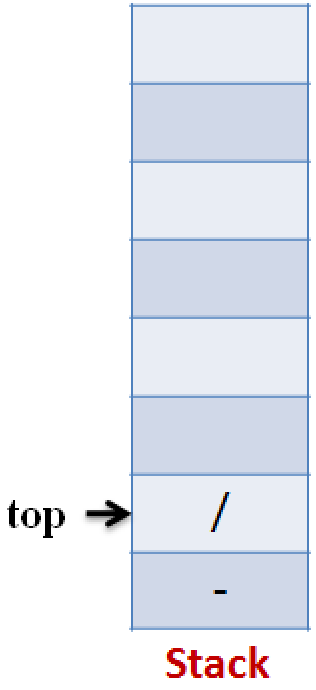
A B C + * D

EXAMPLE

Stage 12

★The last token, **E**, is an operand, so we **insert it to the output**

Expression as it is.



InFix Notation:

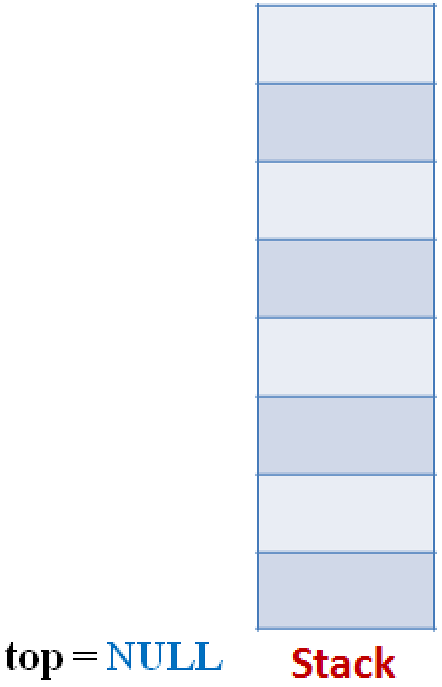
PostFix Notation:

A B C + * D E

EXAMPLE

Stage 13

★The input Expression is complete now. So we **pop the Stack** and **Append it to the Output Expression** as we pop it.



InFix Notation:

PostFix Notation:

A B C + * D E / -

EXAMPLE 2

(((A + B) * (C - E)) / (F + G))

- stack: <empty>
- output: []

EXAMPLE 2

((A + B) * (C - E)) / (F + G))

- stack: (
- output: []

EXAMPLE 2

(A + B) * (C - E)) / (F + G))

- stack: ((
- output: []

EXAMPLE 2

A + B) * (C - E)) / (F + G))

- stack: (((
- output: []

EXAMPLE 2

+ B) * (C - E)) / (F + G))

- stack: (((
- output: [A]

EXAMPLE 2

B) * (C - E)) / (F + G))

- stack: (((+
- output: [A]

EXAMPLE 2

) * (C - E)) / (F + G))

- stack: (((+
- output: [A B]

EXAMPLE 2

* (C - E)) / (F + G))

- stack: ((
- output: [A B +]

EXAMPLE 2

(C - E)) / (F + G))

- stack: ((*
- output: [A B +]

EXAMPLE 2

C - E)) / (F + G))

- stack: ((* (
- output: [A B +]

EXAMPLE 2

- E)) / (F + G))

- stack: ((* (
- output: [A B + C]

EXAMPLE 2

E)) / (F + G))

- stack: ((* (-
- output: [A B + C]

EXAMPLE 2

)) / (F + G))

- stack: ((* (-
- output: [A B + C E]

EXAMPLE 2

) / (F + G))

- stack: ((*
- output: [A B + C E -]

EXAMPLE 2

/ (F + G))

- stack: (
- output: [A B + C E - *]

EXAMPLE 2

(F + G))

- stack: (/
- output: [A B + C E - *]

EXAMPLE 2

F + G))

- stack: (/ (
- output: [A B + C E - *]

EXAMPLE 2

+ G))

- stack: (/ (
- output: [A B + C E - * F]

EXAMPLE 2

G))

- stack: (/ (+
- output: [A B + C E - * F]

EXAMPLE 2

))

- stack: (/ (+
- output: [A B + C E - * F G]

EXAMPLE 2

)

- stack: (/
- output: [A B + C E - * F G +]

EXAMPLE 2

- stack: <empty>
- output: [A B + C E - * F G + /]