CPE360

Infix to Postfix conversion

Stack Applications

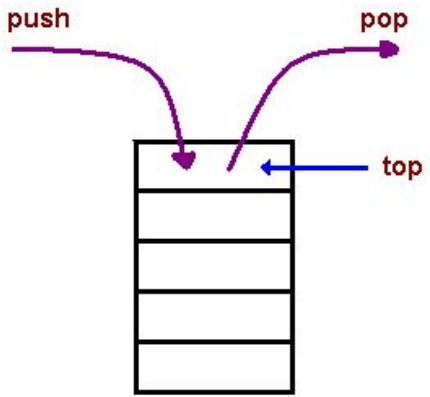
LIFO Philosophy

Subroutines

Recommendation Engines

Memory Management modules

Etc.,



Stack Applications

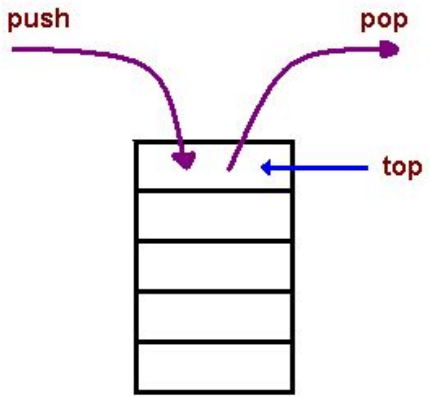
LIFO Philosophy

Subroutines

Recommendation Engines

Memory Management modules

Etc.,



Infix to Postfix Expressions!

What is postfix?

2 + 3

2 + 3 * 5

Postfix

A way of “preserving” the order of operations

Postfix implies that the *operation* is written at the *end* of the expression (‘post’) in a way that captures the desired *order* for a machine that reads from left to right.

a + b

ab+

a + b * c

abc*+

What are we going to do?

We’ll use a **stack** to convert infix expressions **to postfix expressions**

Recall PEMDAS

Before we proceed, we assume you know this rule

Parentheses
Exponents
Multiplication
Division
Addition
Subtraction

Using Stacks to build Postfix

a+b*c = abc*+

What we want

Input: a+b*c

Output: abc*+

Few ground rules

Our program is constrained by the following:

- We read one symbol at a time, left to right
- A symbol is either an operand or an operator
- When we encounter *operands*, we just write them to postfix (e.g., a b c d)
- But, every time we encounter an *operation*, we have a job to do.. (e.g., +, -, \, *, ^)

Terminology

a + b * c - d / e ^ f + g

Operands: a, b, c, d, e, f, g
Operators: +, *, -, /, ^, +

(in that order)

The algorithm

Step 1: Scan the next symbol, terminate if NULL (i.e., no more symbol)
Step 2: If symbol is an *operand*, write straight to *postfix* expression
Step 3: If symbol is an operator:

.....**3.1** If the precedence of the scanned operator is *greater* than the precedence of the operator in the stack (or the stack is empty), **push** it.

.....**3.2** Else, **Pop all the operators** from the stack which are greater than or equal to in precedence than that of the scanned operator. **After doing that Push** the scanned operator to the stack.

Step 4: Repeat 1-3 for all symbols. When all symbols are scanned, pop the contents of the Stack onto the postscript.

Let's start simple

A + B + C

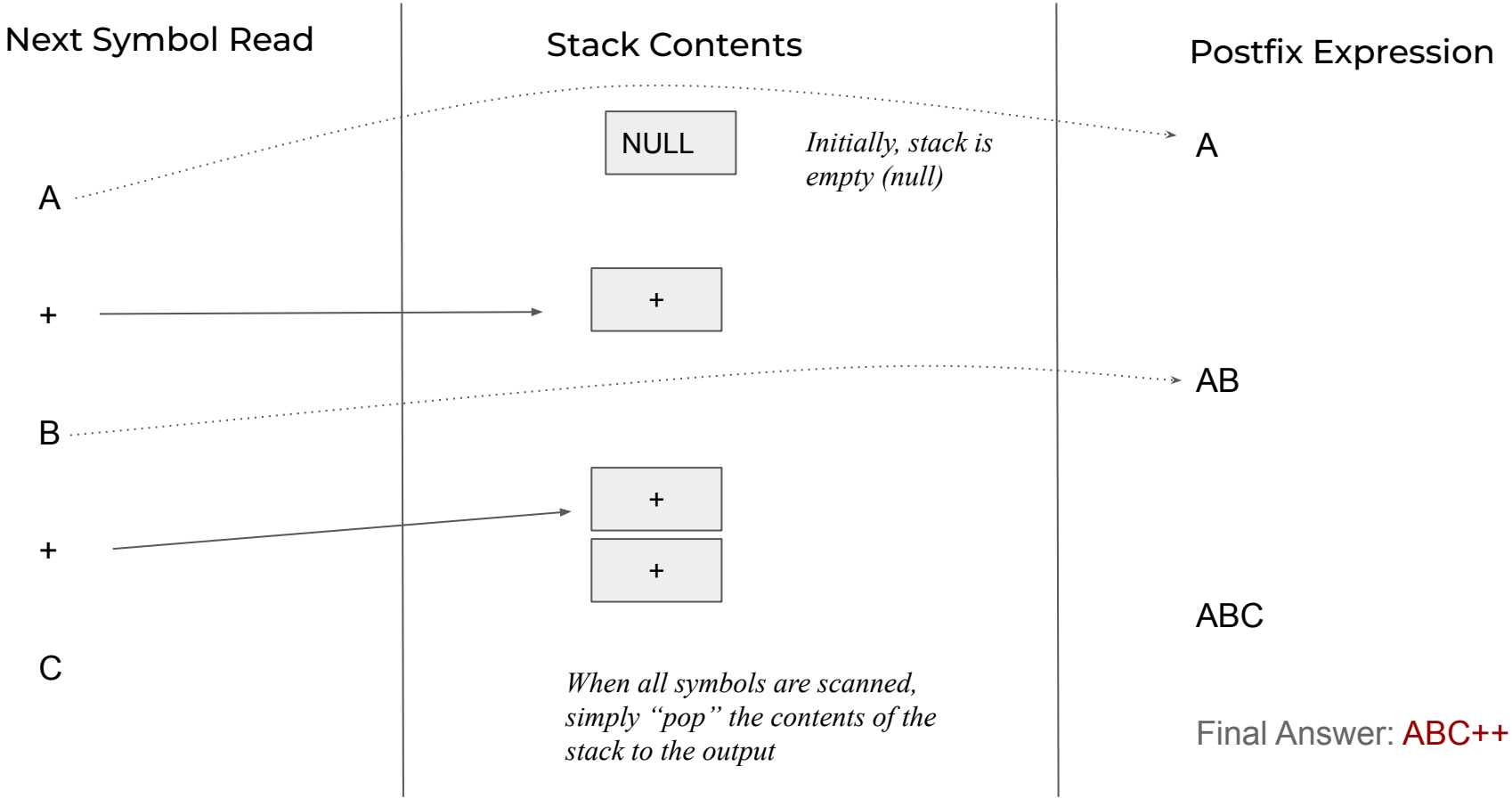
A + B + C

Next Symbol Read	Stack Contents	Postfix Expression
	NULL *Initially, stack is empty (null)*	

A + B + C

Next Symbol Read	Stack Contents	Postfix Expression
A	NULL *Initially, stack is empty (null)*	
+		
B		
+		
C		

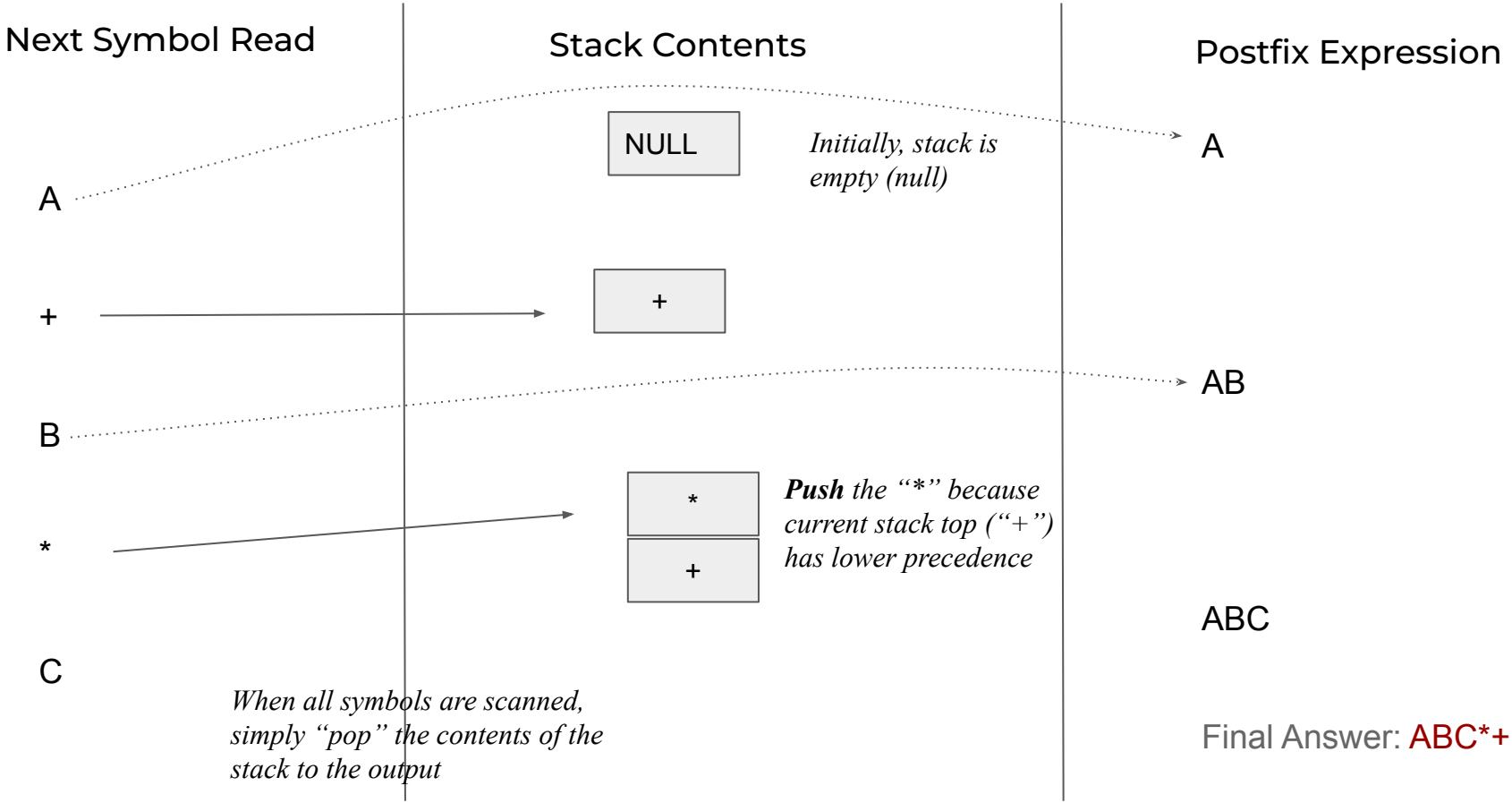
A + B + C


Next Symbol Read
Stack Contents
Postfix Expression
A
+
B
+
C
NULL
Initially, stack is empty (null)
+
+
+
A
AB
ABC
When all symbols are scanned, simply "pop" the contents of the stack to the output
Final Answer: ABC++

Let's try something more tricky

A + B * C

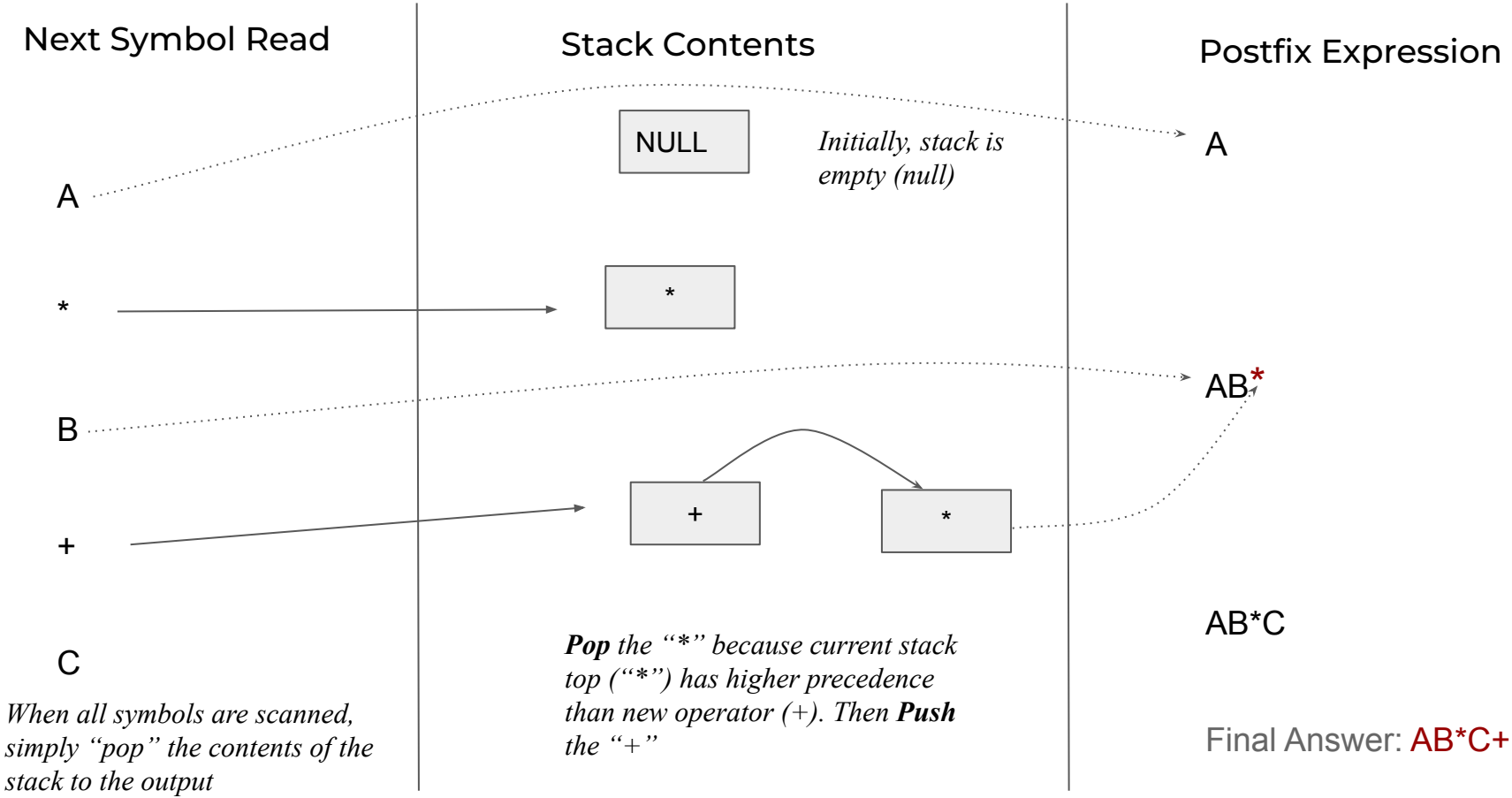
A + B * C


Next Symbol Read
Stack Contents
Postfix Expression
A
+
B
*
C
NULL
Initially, stack is empty (null)
+
A
AB
*
+
Push the "*" because current stack top ("+") has lower precedence
ABC
When all symbols are scanned, simply "pop" the contents of the stack to the output
Final Answer: ABC*+

If it were this instead

A * B + C

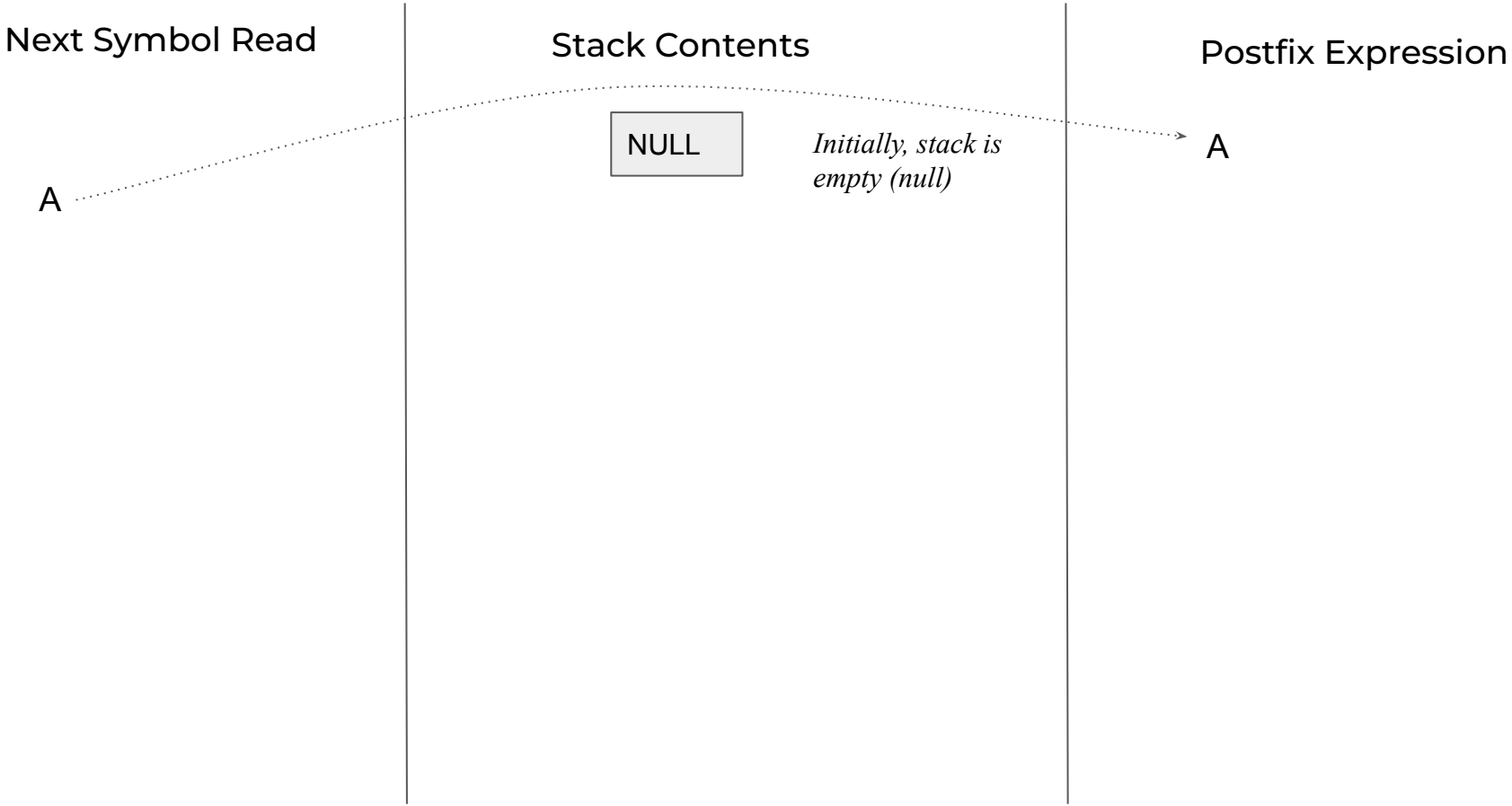
A * B + C


Next Symbol Read
Stack Contents
Postfix Expression
NULL
Initially, stack is empty (null)
A
*
B
+
C
*
+
*
A
AB*
AB*C
Pop the "*" because current stack top ("*") has higher precedence than new operator (+). Then Push the "+"
When all symbols are scanned, simply "pop" the contents of the stack to the output
Final Answer: AB*C+

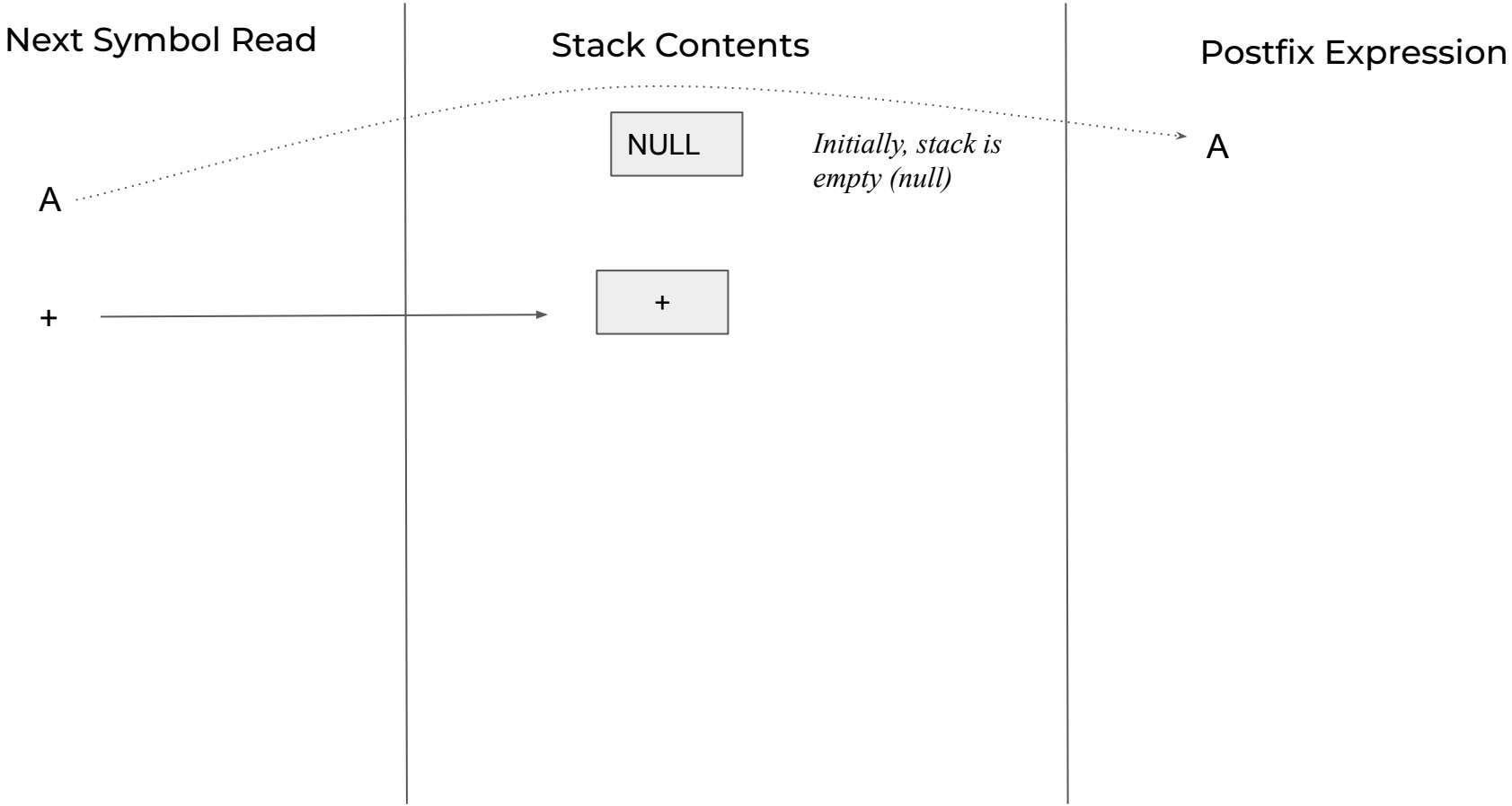
Alright, let's try this one now

A + B * C - D/F

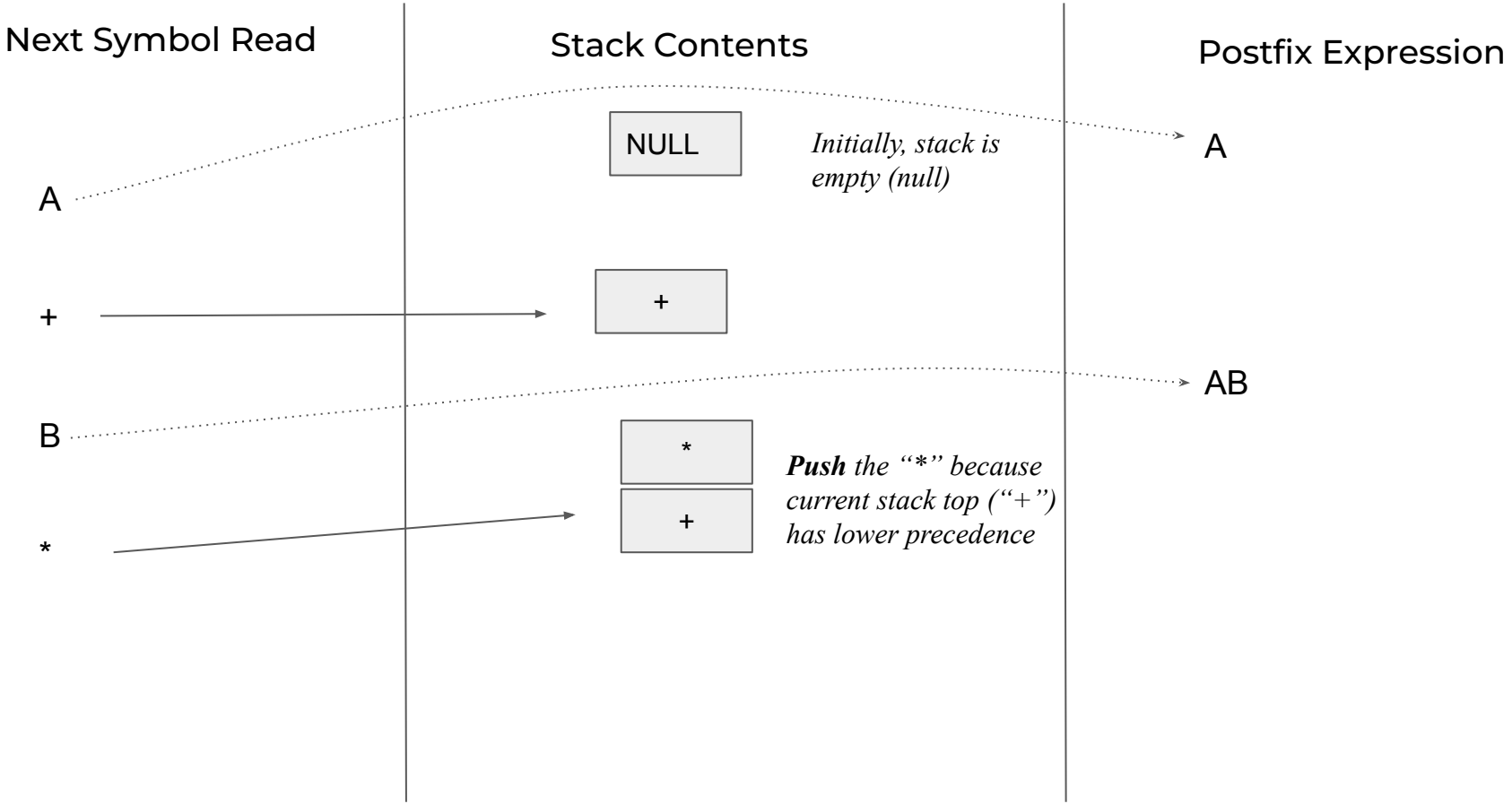
A + B*C - D/F


Next Symbol Read
Stack Contents
Postfix Expression
A
NULL
Initially, stack is empty (null)
A

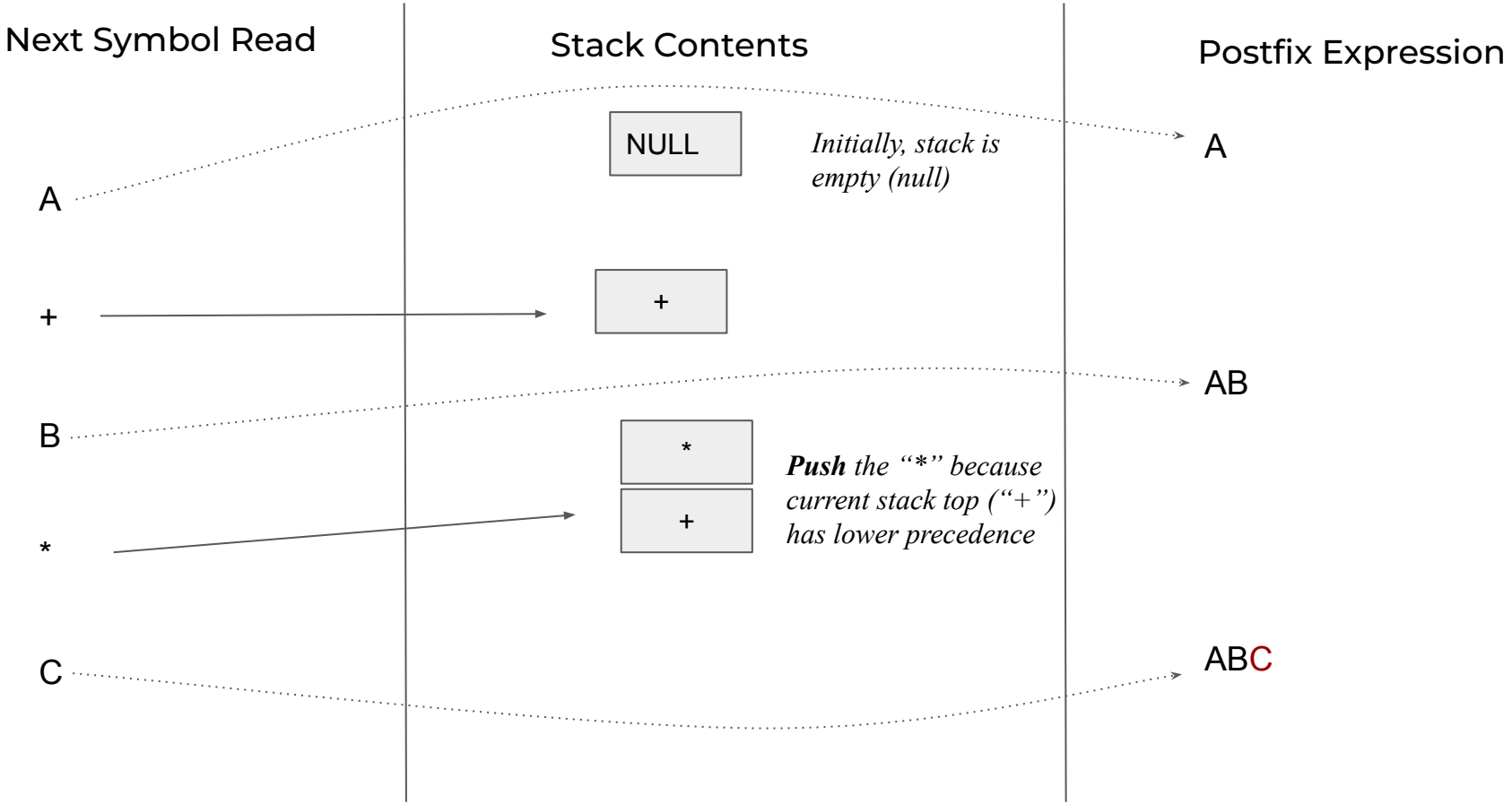
A + B*C - D/F


Next Symbol Read
Stack Contents
Postfix Expression
NULL
Initially, stack is empty (null)
A
A
+
+

A + B*C - D/F


Next Symbol Read
Stack Contents
Postfix Expression
NULL
Initially, stack is empty (null)
A
A
+
+
B
AB
*
*
+
Push the "*" because current stack top ("+") has lower precedence

A + B*C - D/F

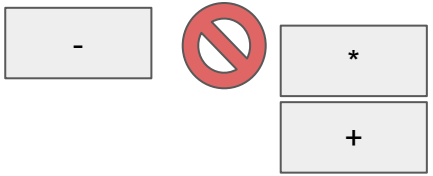

Next Symbol Read
Stack Contents
Postfix Expression
NULL
Initially, stack is empty (null)
A
A
+
+
AB
B
*
+
Push the "*" because current stack top ("+") has lower precedence
*
C
ABC

A + B*C - D/F

Next Symbol Read

Stack Contents

Postfix Expression

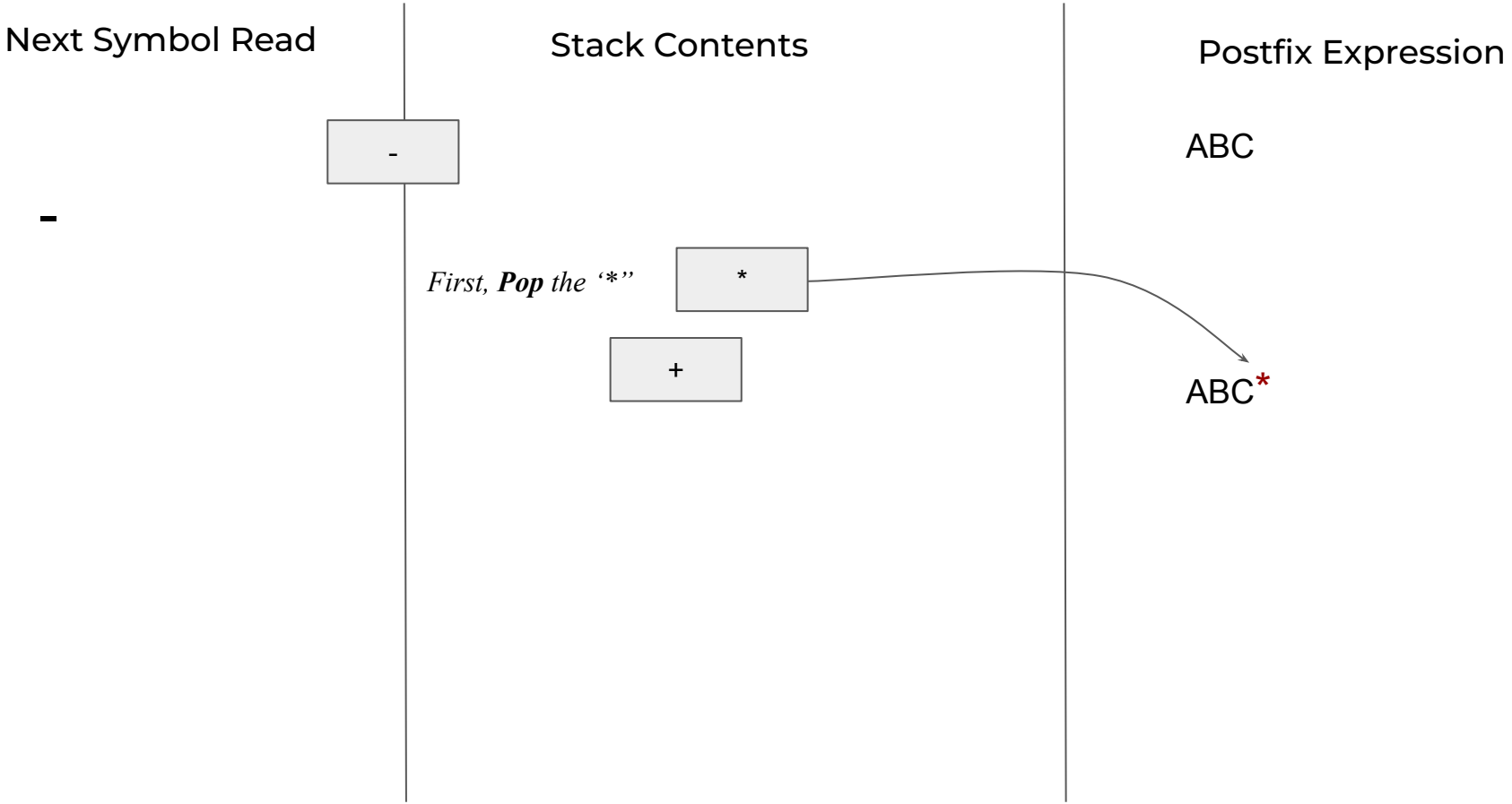


"-" has a lower precedence than "", and "+".*

*We have to pop both the * and + to make room for "-"*

ABC

A + B*C - D/F

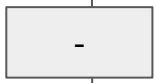

Next Symbol Read
Stack Contents
Postfix Expression
-
ABC
-
First, Pop the "*"
*
+
ABC*

A + B*C - D/F

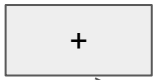
Next Symbol Read

-

Stack Contents



*Next, **Pop** the '+"*



Postfix Expression

ABC*

ABC*+

A + B*C - D/F

Next Symbol Read

Stack Contents

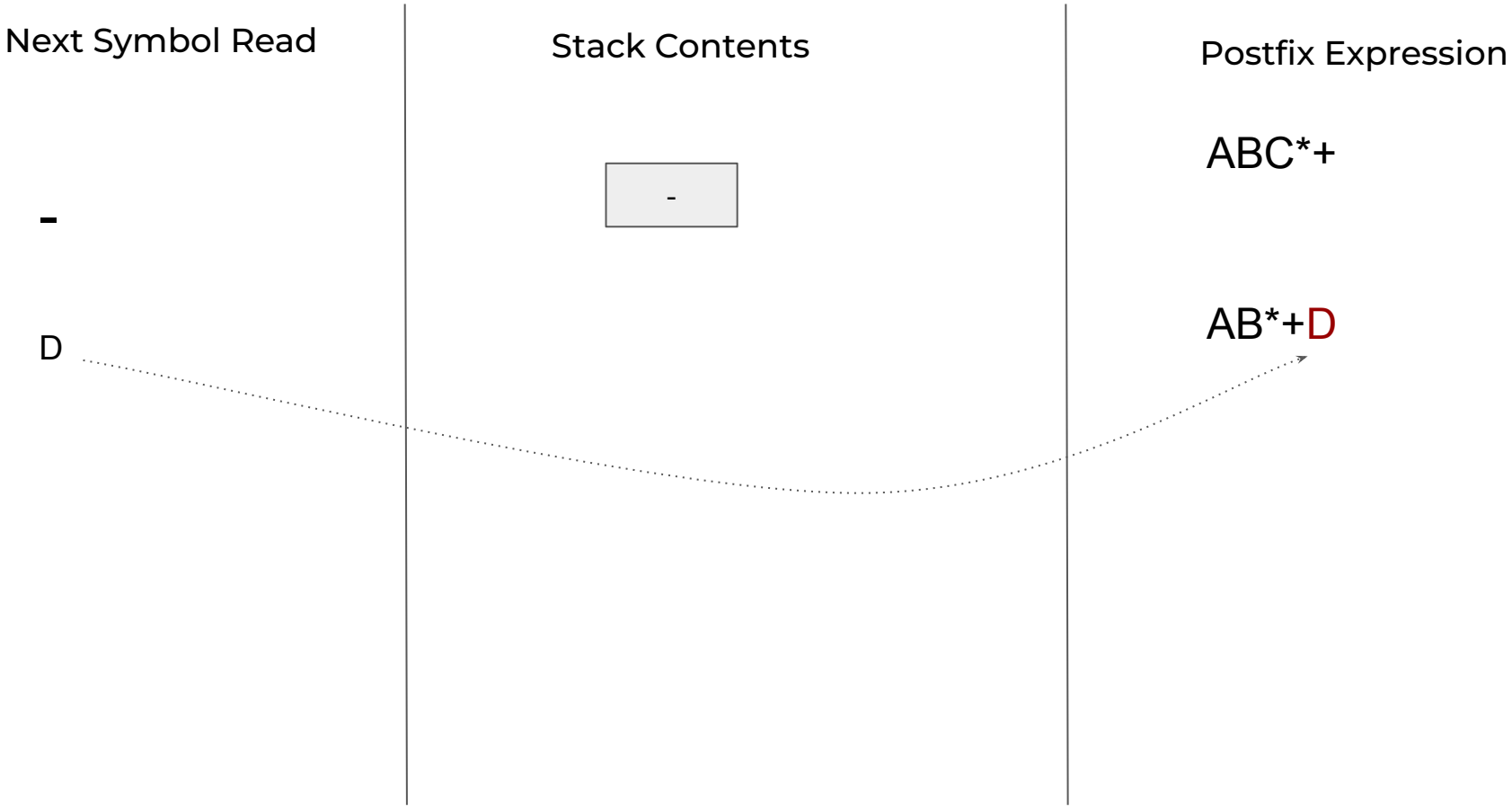
Postfix Expression

-

-

ABC*+

A + B*C - D/F

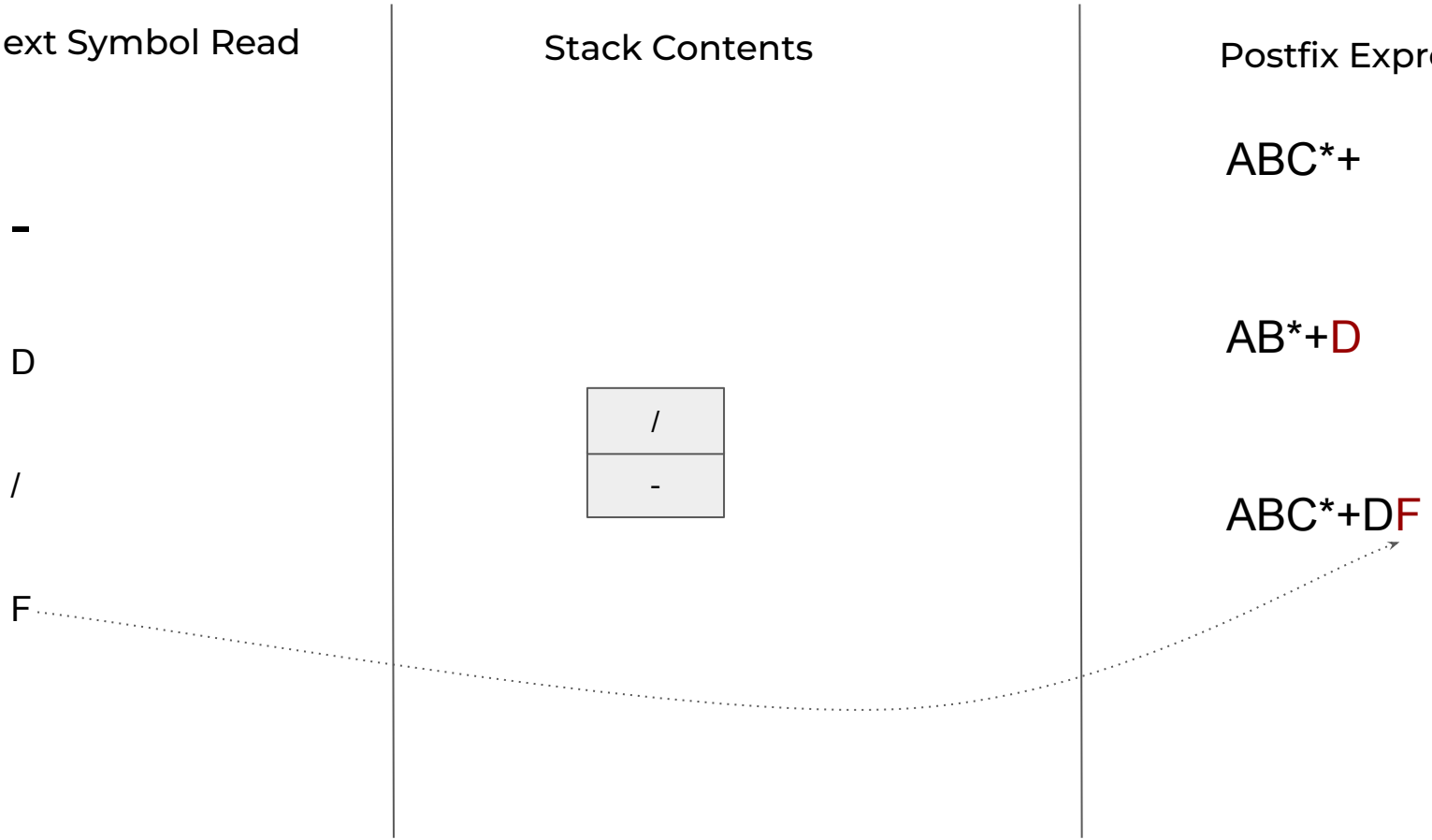

Next Symbol Read
Stack Contents
Postfix Expression
-
D
-
ABC*+
AB*+D

A + B*C - D/F

Next Symbol Read	Stack Contents	Postfix Expression
-		ABC*+
D		AB*+D
/	/ -	

A + B*C - D/F

Next Symbol Read	Stack Contents	Postfix Expression
-		ABC*+
D		AB*+D
/	/ -	
F		ABC*+DF



A + B*C - D/F

Next Symbol Read	Stack Contents	Postfix Expression
-		ABC*+
D		AB*+D
/	/ -	ABC*+DF
F		ACB*+DF/-

When all symbols are scanned, simply "pop" the contents of the stack to the output

Parentheses

(a+b)*c = ab+c*

The algorithm

Step 1: Scan the next symbol, terminate if NULL (i.e., no more symbol)
Step 2: If symbol is an *operand*, write straight to *postfix* expression
Step 3: If symbol is an operator:

.....**3.1** If the precedence of the scanned operator is *greater* than the precedence of the operator in the stack (or the stack is empty), **push** it.

If the symbol is a "(", simply push it

.....**3.2** Else, **Pop all the operators** from the stack which are greater than or equal to in precedence than that of the scanned operator. **After doing that Push** the scanned operator to the stack.

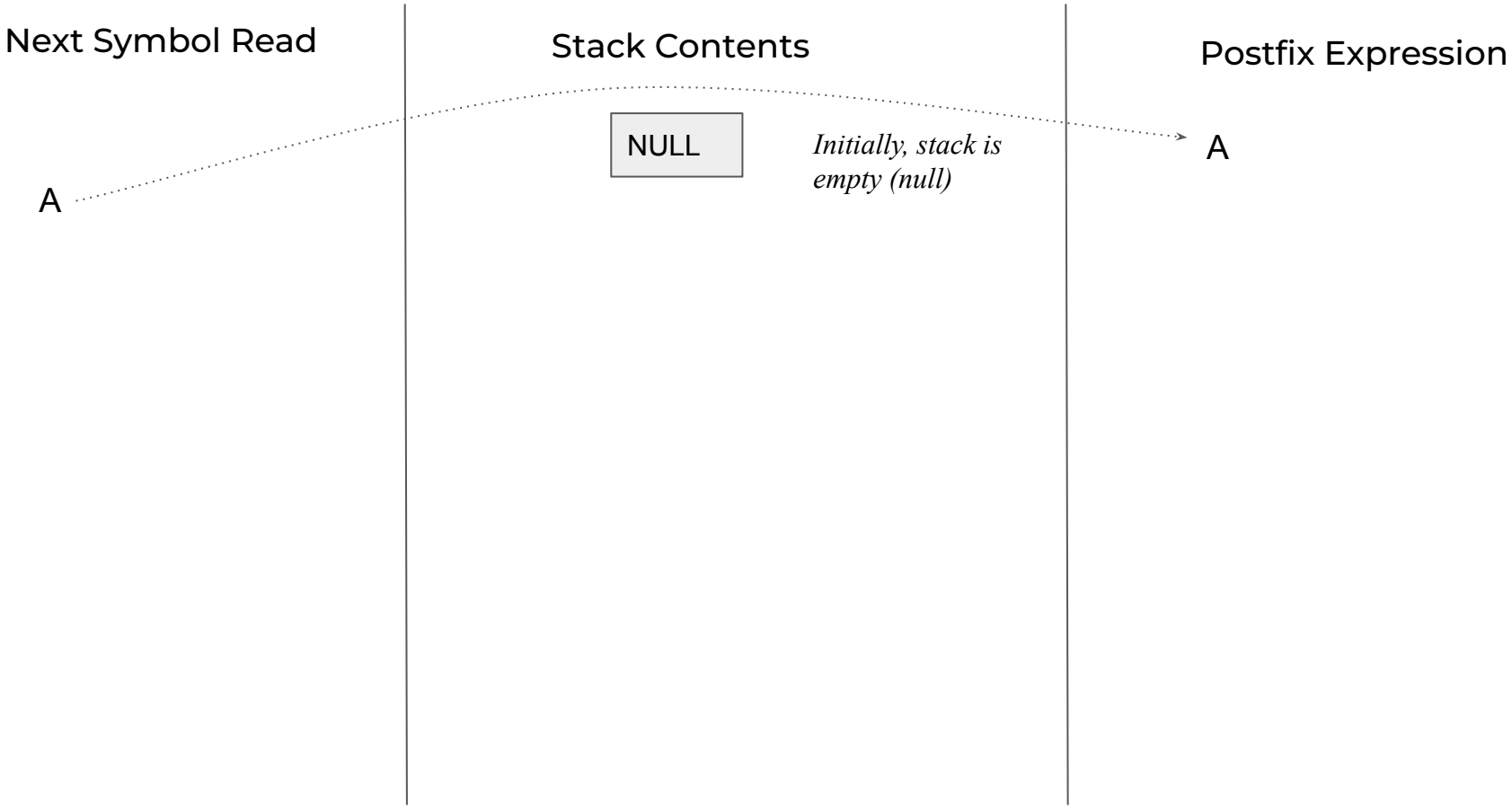
If the symbol is a ")", pop all operators until you encounter "(". DO NOT write parentheses to postfix

Step 4: Repeat 1-3 for all symbols. When all symbols are scanned, pop the contents of the Stack onto the postfix.

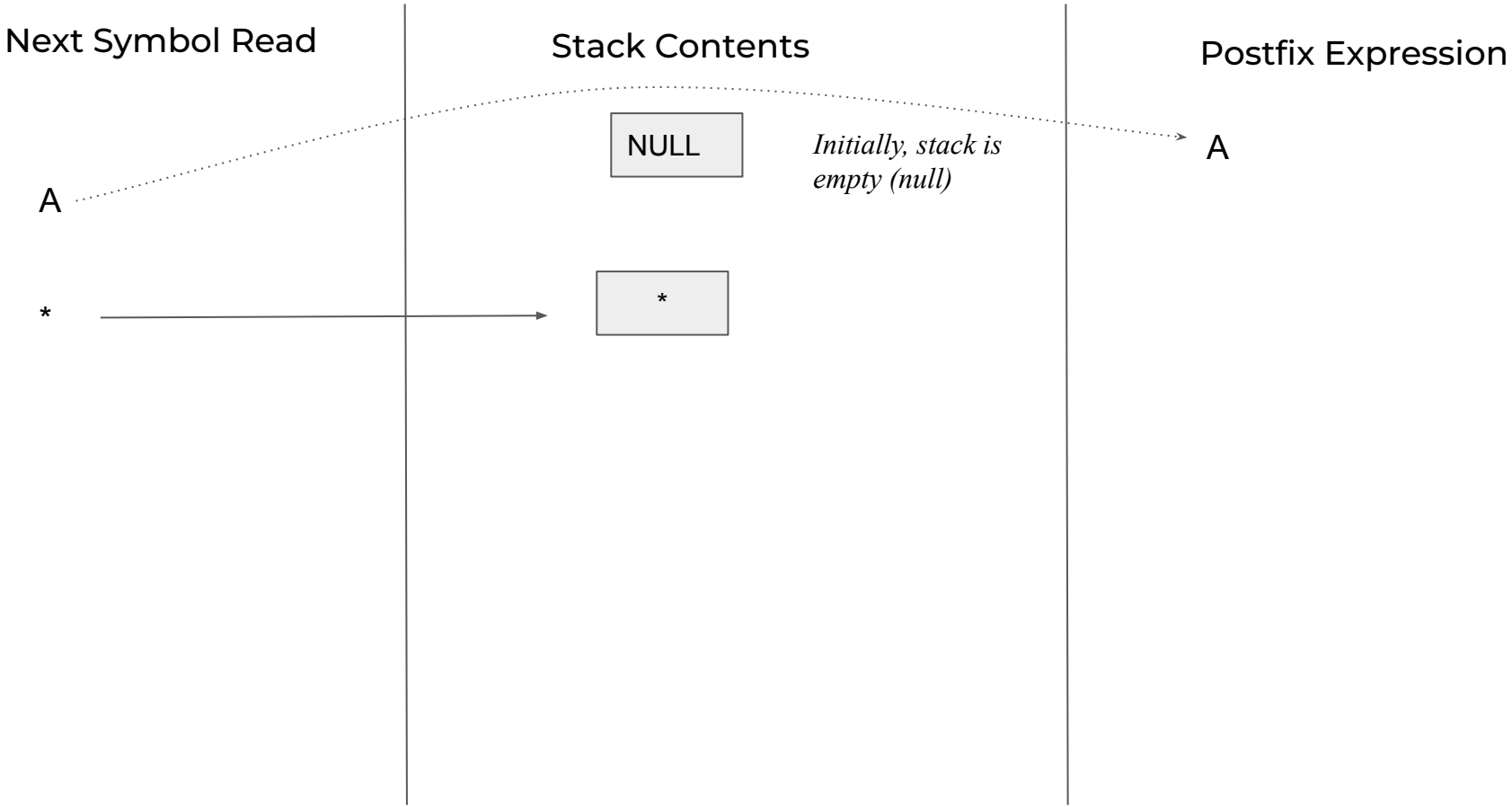
Alright, let's try this one now

A * (B + C) / D

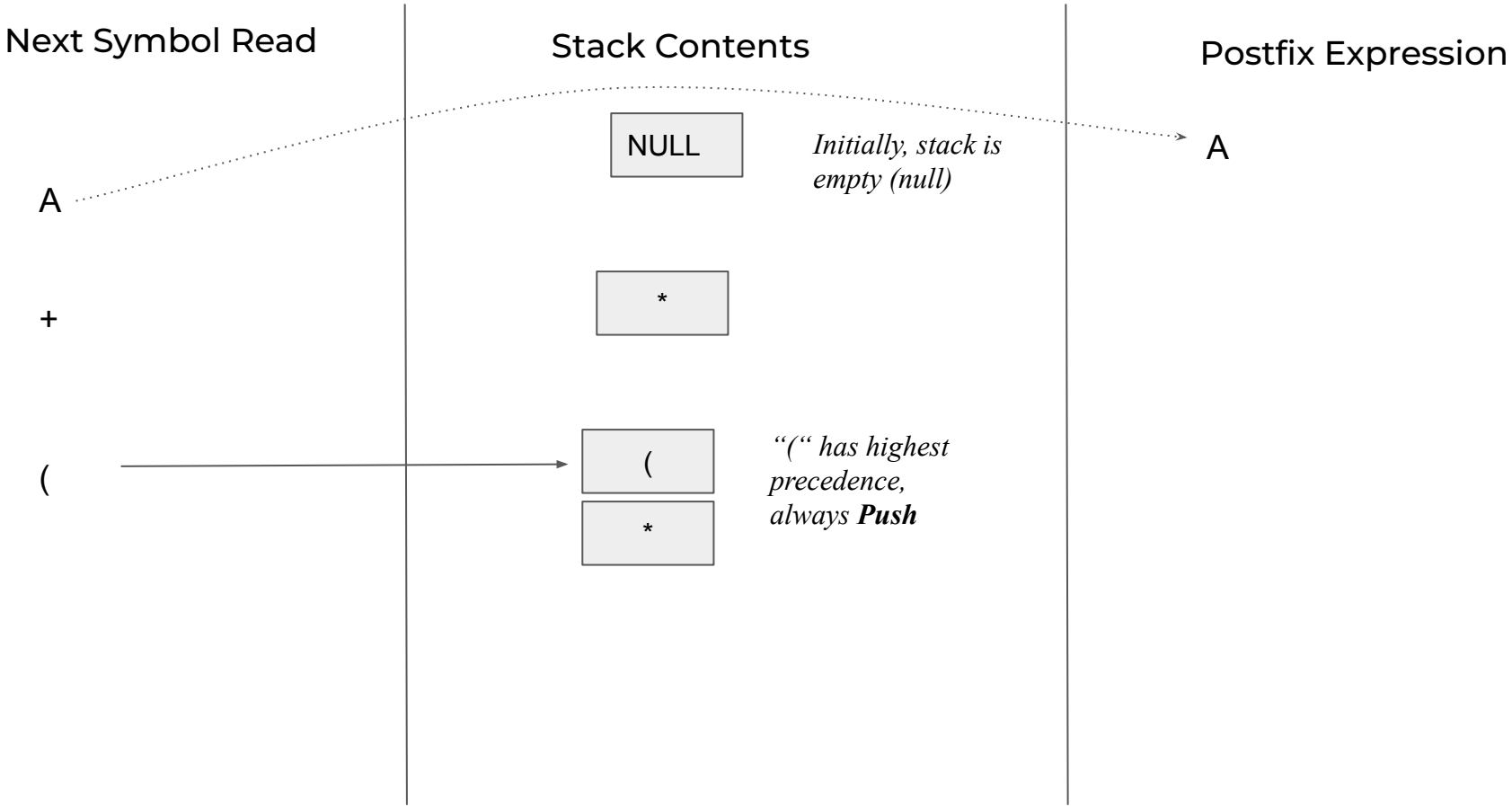
A * (B + C) / D


Next Symbol Read
Stack Contents
Postfix Expression
A
NULL
Initially, stack is empty (null)
A

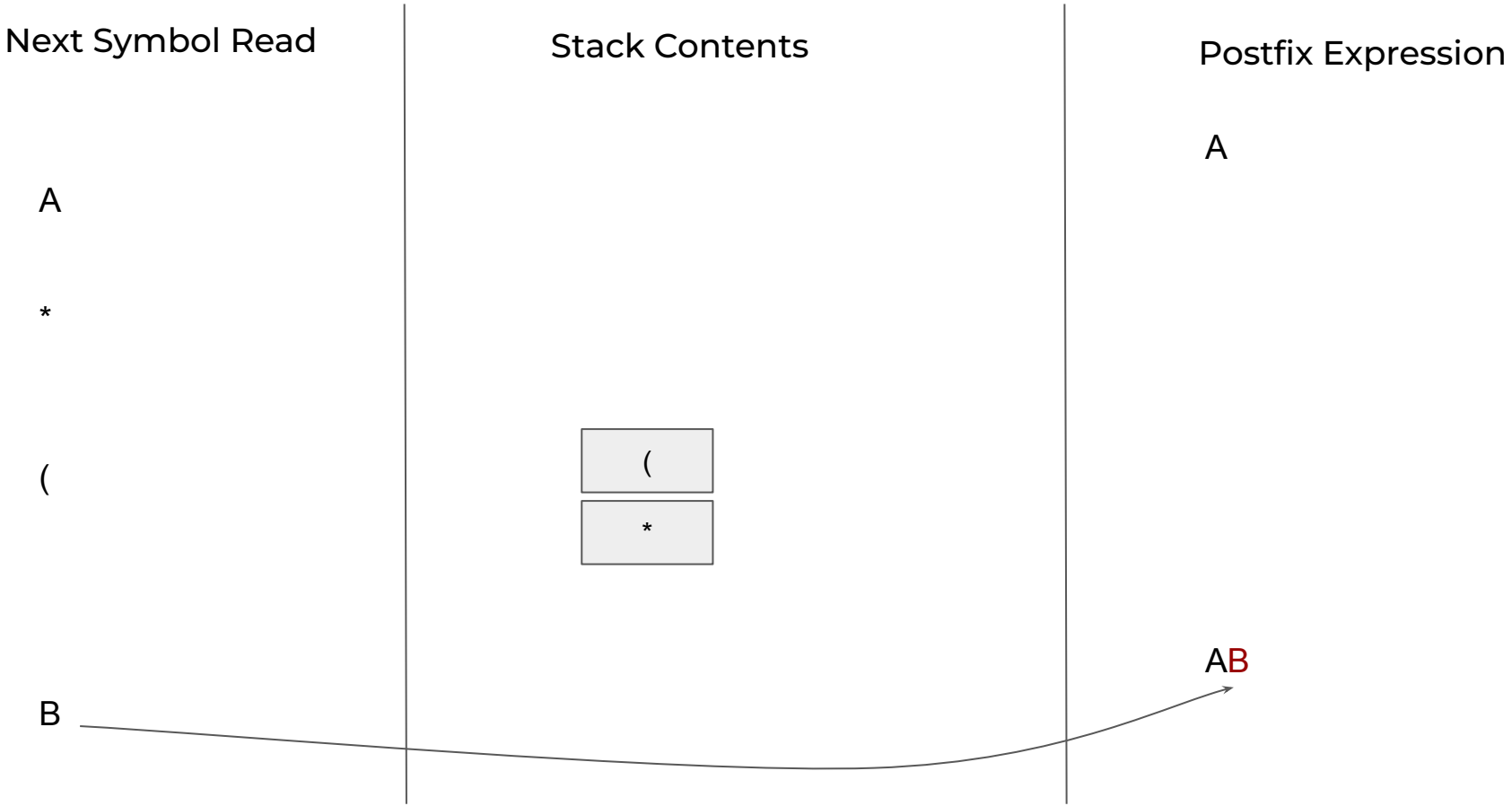
A * (B + C) / D


Next Symbol Read
Stack Contents
Postfix Expression
NULL
Initially, stack is empty (null)
A
A
*
*

A * (B + C) / D


Next Symbol Read
Stack Contents
Postfix Expression
NULL
Initially, stack is empty (null)
A
A
+
*
(
(
*
"(" has highest precedence, always **Push**

A * (B + C) / D

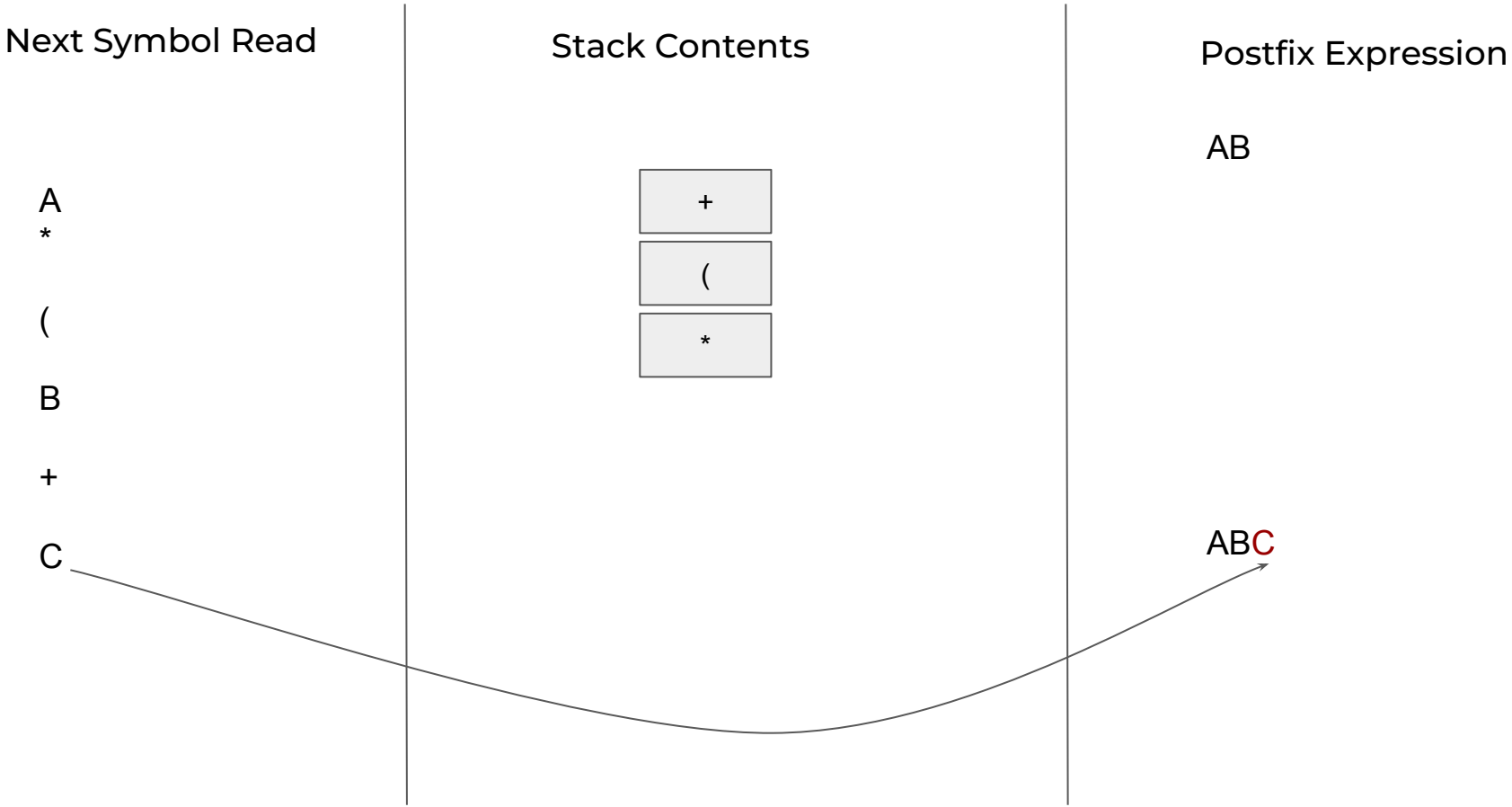


A * (B + C) / D

Next Symbol Read	Stack Contents	Postfix Expression
A		A
*		
(		
B		
+	+ (*	AB

"(" at the top of Stack almost creates a brand new scope -- notice that "+" is pushed.

A * (B + C) / D


Next Symbol Read
Stack Contents
Postfix Expression
A
*
(
B
+
C
+
(
*
AB
ABC

A * (B + C) / D

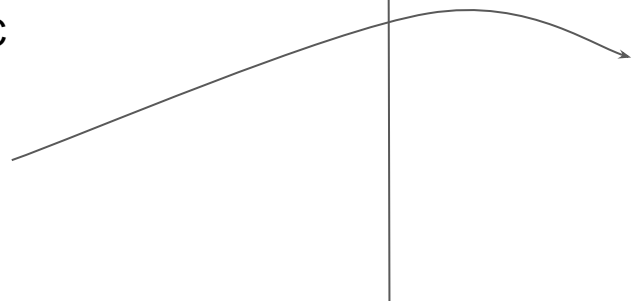
Next Symbol Read

Stack Contents

Postfix Expression

A
*
(
B
+
C

)

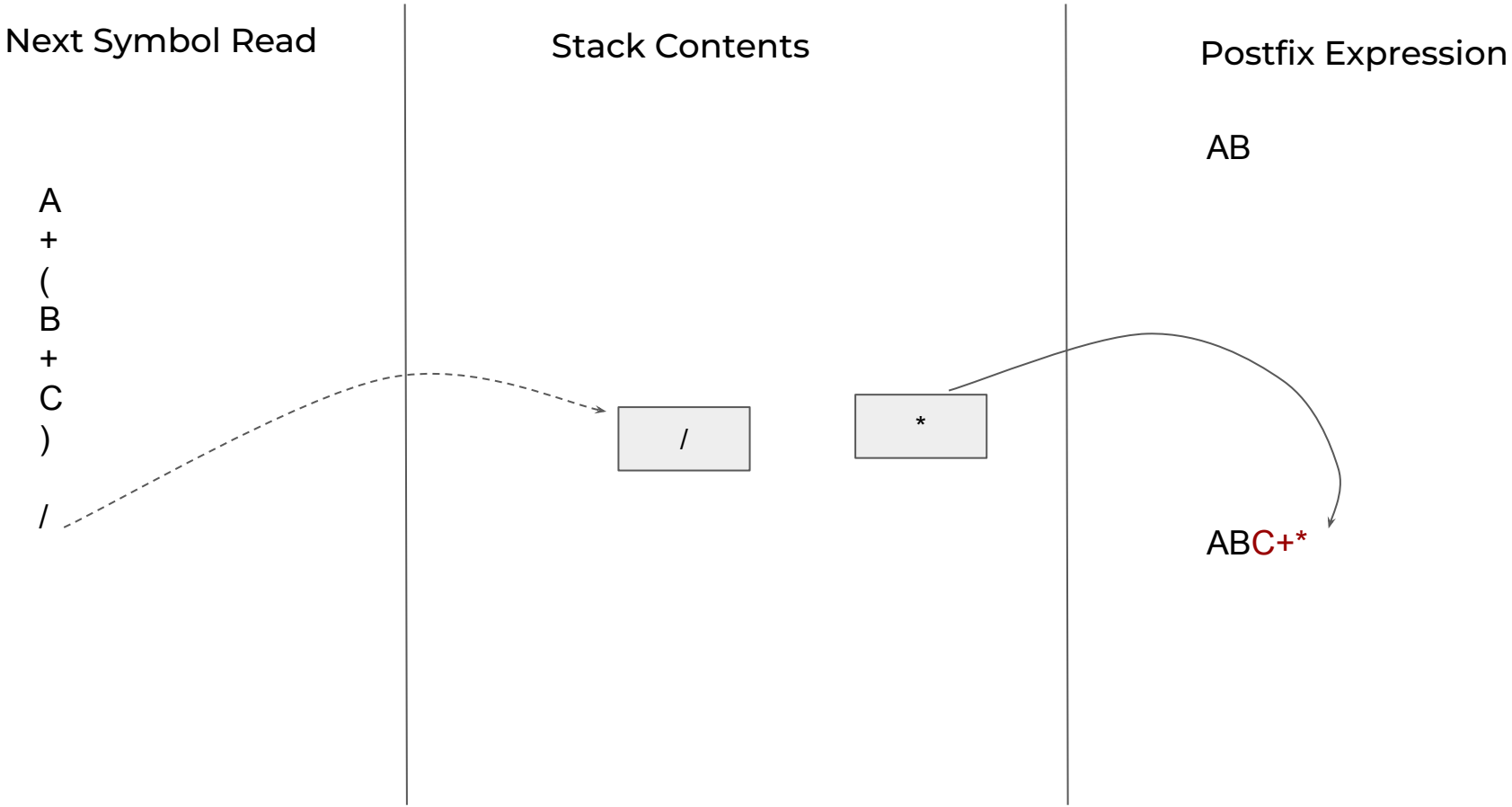


")" Pop everything until you encounter "(" -- and do not write parentheses to postfix

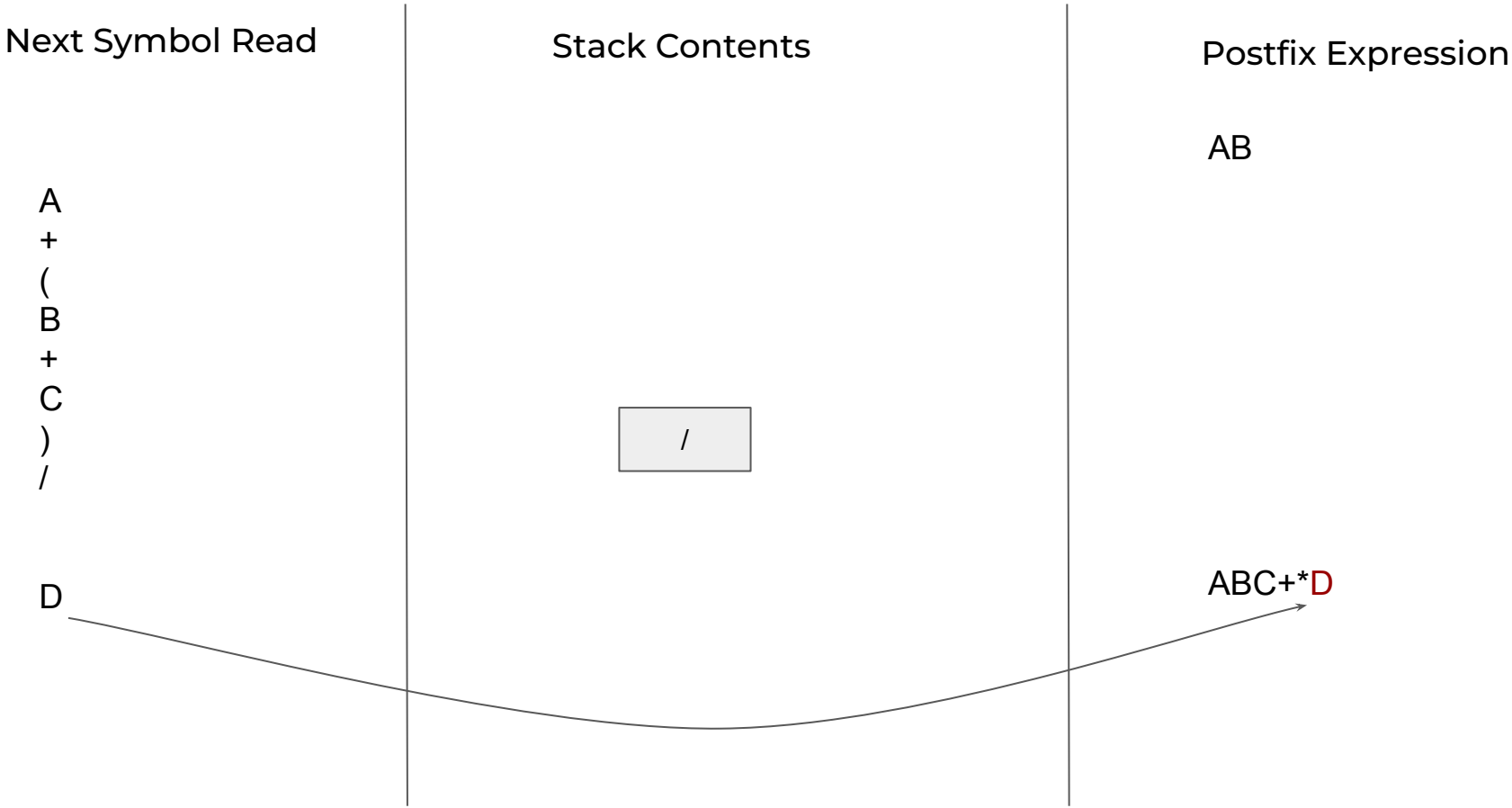
AB

ABC

A * (B + C) / D


Next Symbol Read
Stack Contents
Postfix Expression
AB
A
+
(
B
+
C
)
/
/
*
ABC+*

A * (B + C) / D


Next Symbol Read
Stack Contents
Postfix Expression
AB
A
+
(
B
+
C
)
/
/
D
ABC+*D

A * (B + C) / D

Next Symbol Read	Stack Contents	Postfix Expression
A		AB
+		
(		
B		
+		
C		
)	*	
/		
D		ABC+*D/

When all symbols are scanned, simply "pop" the contents of the stack to the output

Hope this was fun!