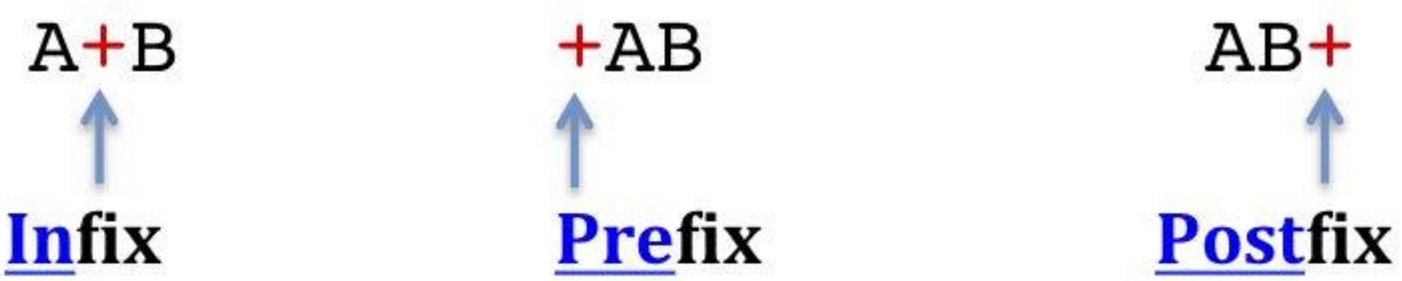
Data Structure

Lec 03 Stack Applications

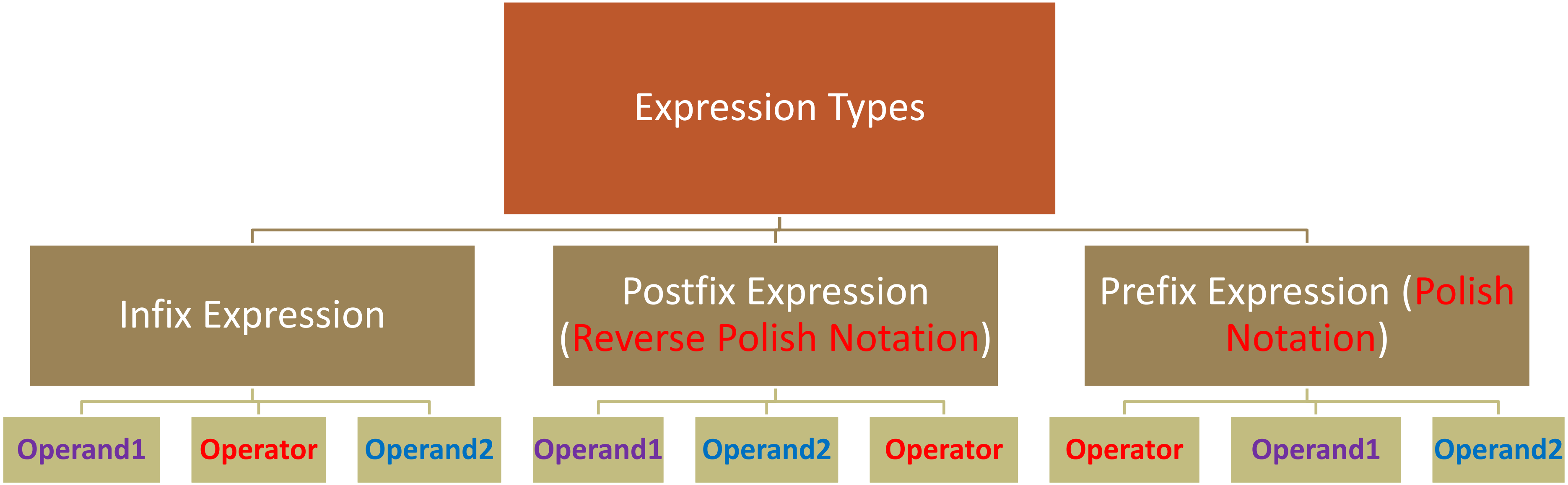


Expressions

What is an Expression?

- In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task.
- These set of symbols (+,-,*,/,%,^,<,>,=) makes an expression.
- An expression is a collection of operators and operands that represents a specific value.
- An operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.
- Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types



Expression Types

Infix Expression

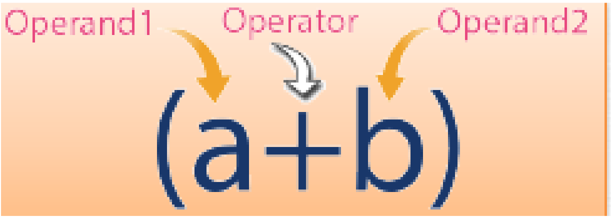
Postfix Expression (Reverse Polish Notation)

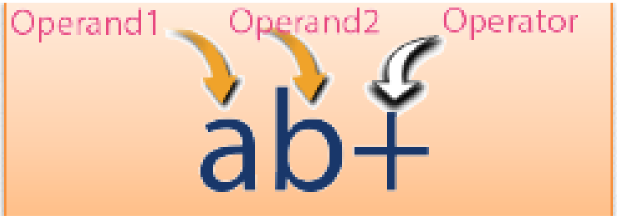
Prefix Expression (Polish Notation)

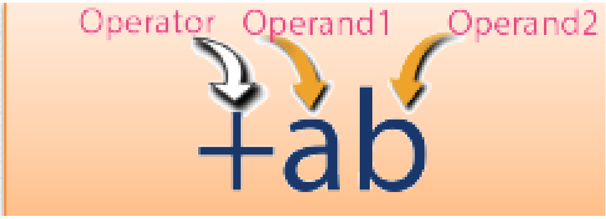
Operand1 | Operator | Operand2

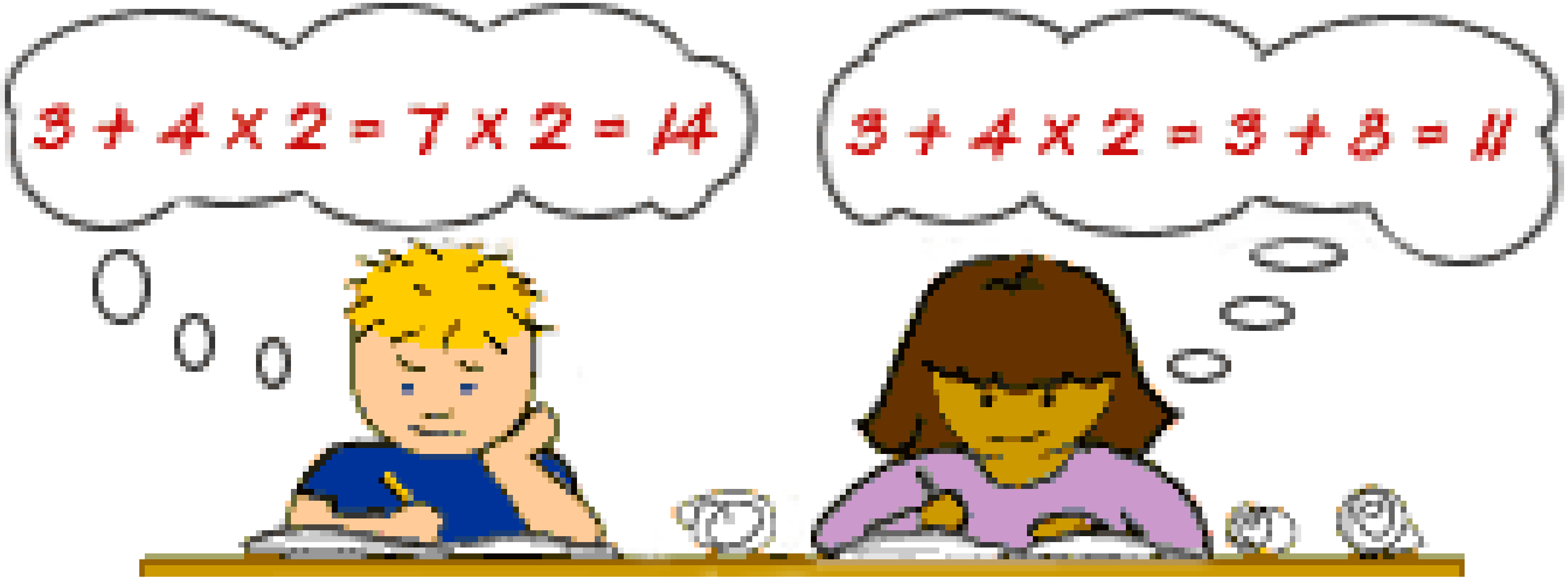
Operand1 | Operand2 | Operator

Operator | Operand1 | Operand2









Expression Evaluation

Expression Evaluation

- Expression evaluations is one of the major application that illustrates the different types of stacks.
- Consider the sum of A and B

A+B

Where A and B are called operands and "+" is called operator

- This representation is called Infix
- There are two alternate notations for expressing the sum of A and B:

+ A B Prefix

A B + Postfix

Expression Evaluation

- We have five binary operations: addition, subtraction, multiplication, division, and exponentiation.
- The first four are available in C++ and are denoted by the usual operators +, -, *, and /
- The fifth exponentiation is represented by the operator ^

A^B   A is raised to the power B

3^2=9

- In order to get the value of any expression (infix, postfix, prefix). You have to know the priority or precedence of each operator in the expression.
- Then the highest Priority will be executed first then the less priority and the less and …..

Expression Evaluation Priority

The order of precedence is:

	Symbol	Comment
Parentheses	**[ ] , { }, ( )**	**highest priority**
Exponentiation, + - sign	**^, - (unary negation)**	
Multiplication / division	***, /, %**	**Left to right**
Addition / subtraction	**+,-**	**Left to right**

Expression Evaluation

- The expression A+B*C is evaluated as A+(B*C) because multiplication takes precedence over addition.

- To rewrite A+B*C in postfix:

A+(B*C)	parentheses for emphasis
A+(BC*)	convert the multiplication
A(BC*)+	convert the addition
ABC*+	postfix form

- To rewrite (A+B)*C in postfix:

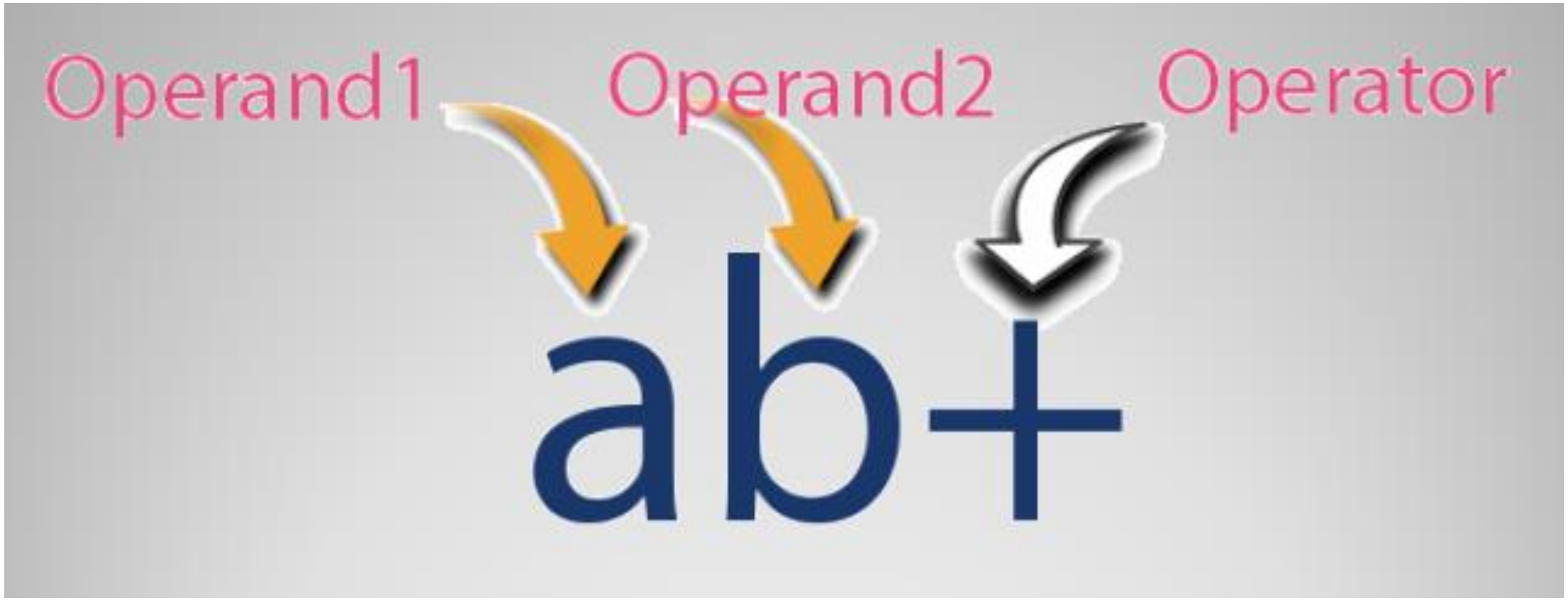
(A+B)*C	infix form
(AB+)*C	convert the addition
(AB+)C*	convert the multiplication
AB+C*	postfix form

Expression Evaluation

Infix	Postfix
A+B	AB+
A+B-C	AB+C-
(A+B)*(C-D)	AB+CD-*
A^B*C-D+E/F/(G+H)	AB^C*D-EF/GH+/+
((A+B)*C-(D-E))^(F+G)	AB+C*DE--FG+^
A-B/(C*D^E)	ABCDE^*/-

Expression Evaluation

Infix	Prefix
A+B	+AB
A+B-C	-+ABC
(A+B)*(C-D)	*+AB-CD
A^B*C-D+E/F/(G+H)	+-*^ABCD//EF+GH
((A+B)*C-(D-E))^(F+G)	^-*+ABC-DE+FG
A-B/(C*D^E)	-A/B*C^DE



Postfix Evaluation

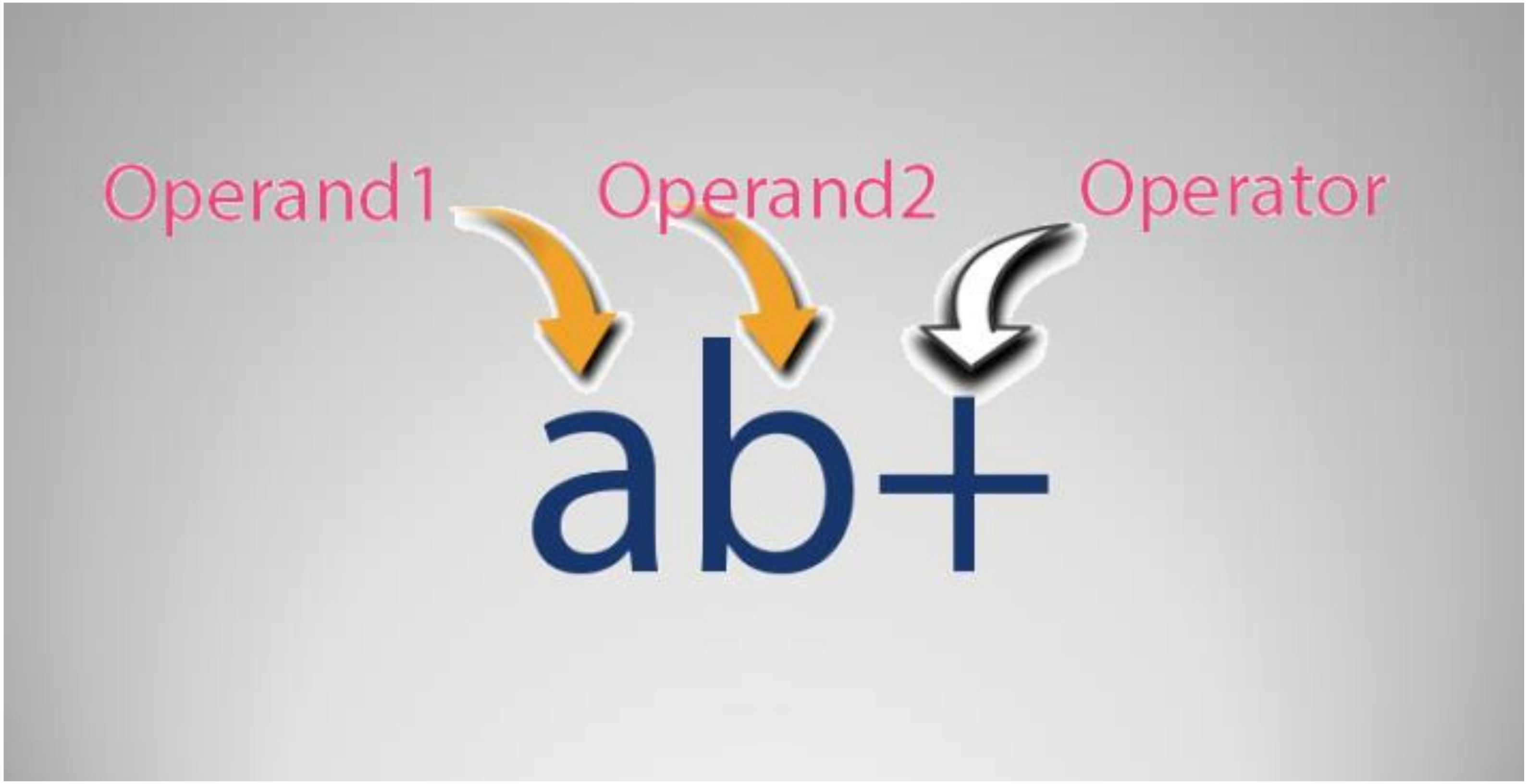
Postfix Evaluation

To evaluate a postfix expression using **Stack** data structure we can use the following steps....

1. Read all the symbols one by one from left to right in the given Postfix Expression.
2. If the reading symbol is operand, then push it onto the **Stack**.
3. If the reading symbol is operator (+,-,*,/etc.,), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back onto the **Stack**.
4. Finally! Perform a pop operation and display the popped value as final result.

Evaluating a postfix expression

```
opndstk = the empty stack;
/* scan the input string reading one element at a time into symb*/
while (not end of input) {
    symb = next input character;
    if (symb is an operand)
            opndstk.push (symb);
    else // symb is an operator
    {
      opnd2 = opndstk.pop();
      opnd1 = opndstk.pop();
      value  =  result of applying symb to opnd1 and opnd2;
      opndstk.push( value);
    }
}//end while
return opndstk.pop();
```



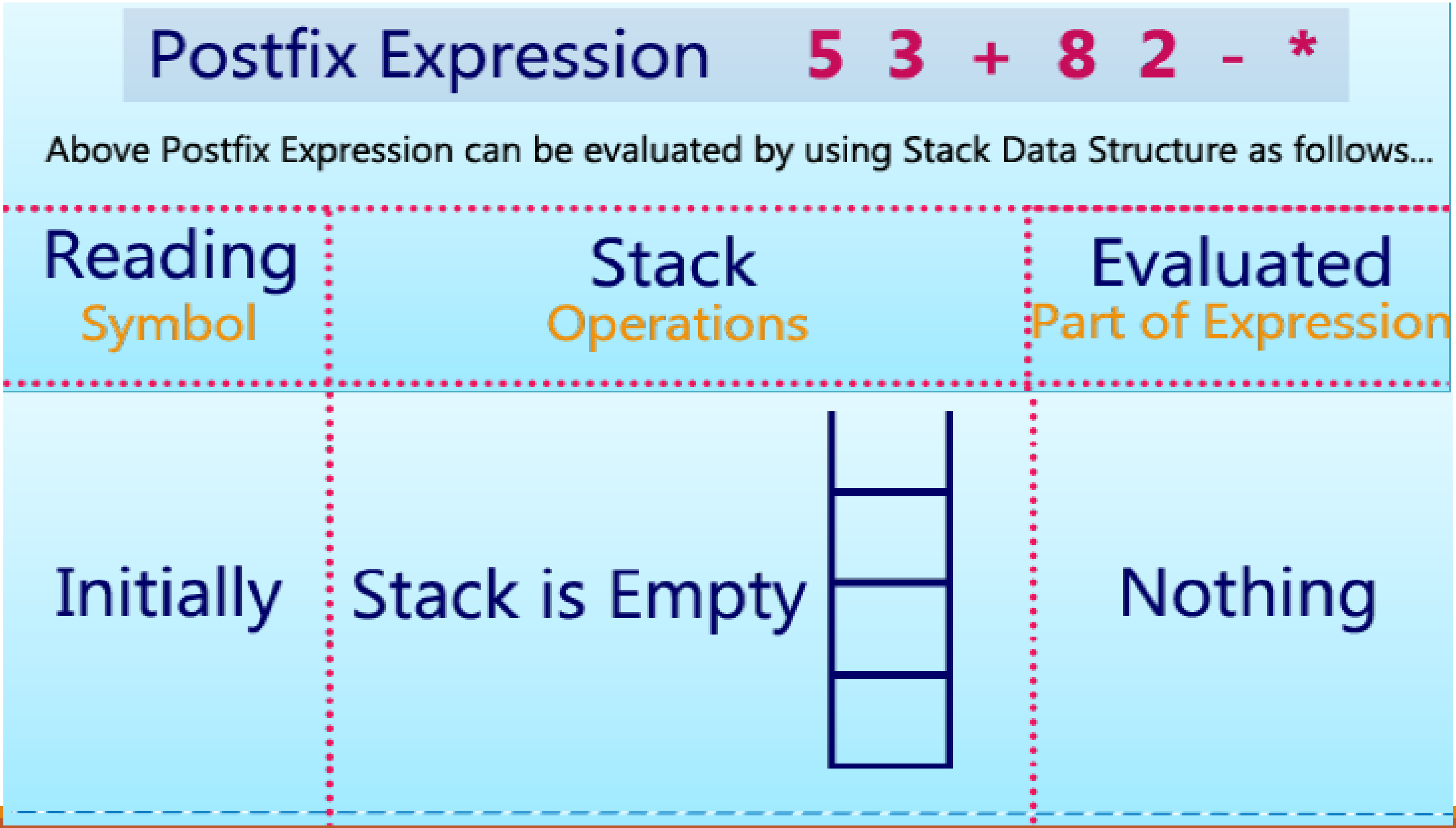
Postfix Evaluation Examples

Evaluating a postfix expression

6 2 3 + -3 8 2 / + * 2 ^ 3 +

symb	opnd1 (2nd pop)	opnd2 (1st pop)	value	opndstk
push(6)				6
push(2)				6, 2
push(3)				6, 2,3
+	pop() ≡ 2	pop() ≡ 3	push(2+3) ≡ 5	6, 5
-	pop() ≡ 6	pop() ≡ 5	push(6-5) ≡ 1	1
push(3)				1, 3
push(8)				1, 3, 8
push(2)				1, 3, 8, 2
/	pop() ≡ 8	pop() ≡ 2	push(8/2) ≡ 4	1, 3, 4
+	pop() ≡ 3	pop() ≡ 4	push(3+4) ≡ 7	1, 7
*	pop() ≡ 1	pop() ≡ 7	push(1*7) ≡ 7	7
push(2)				7, 2
^	pop() ≡ 7	pop() ≡ 2	push(7^2) ≡ 49	49
push(3)				49, 3
+	pop() ≡ 49	pop() ≡ 3	push(49+3) ≡52	52

Postfix Evaluation Example

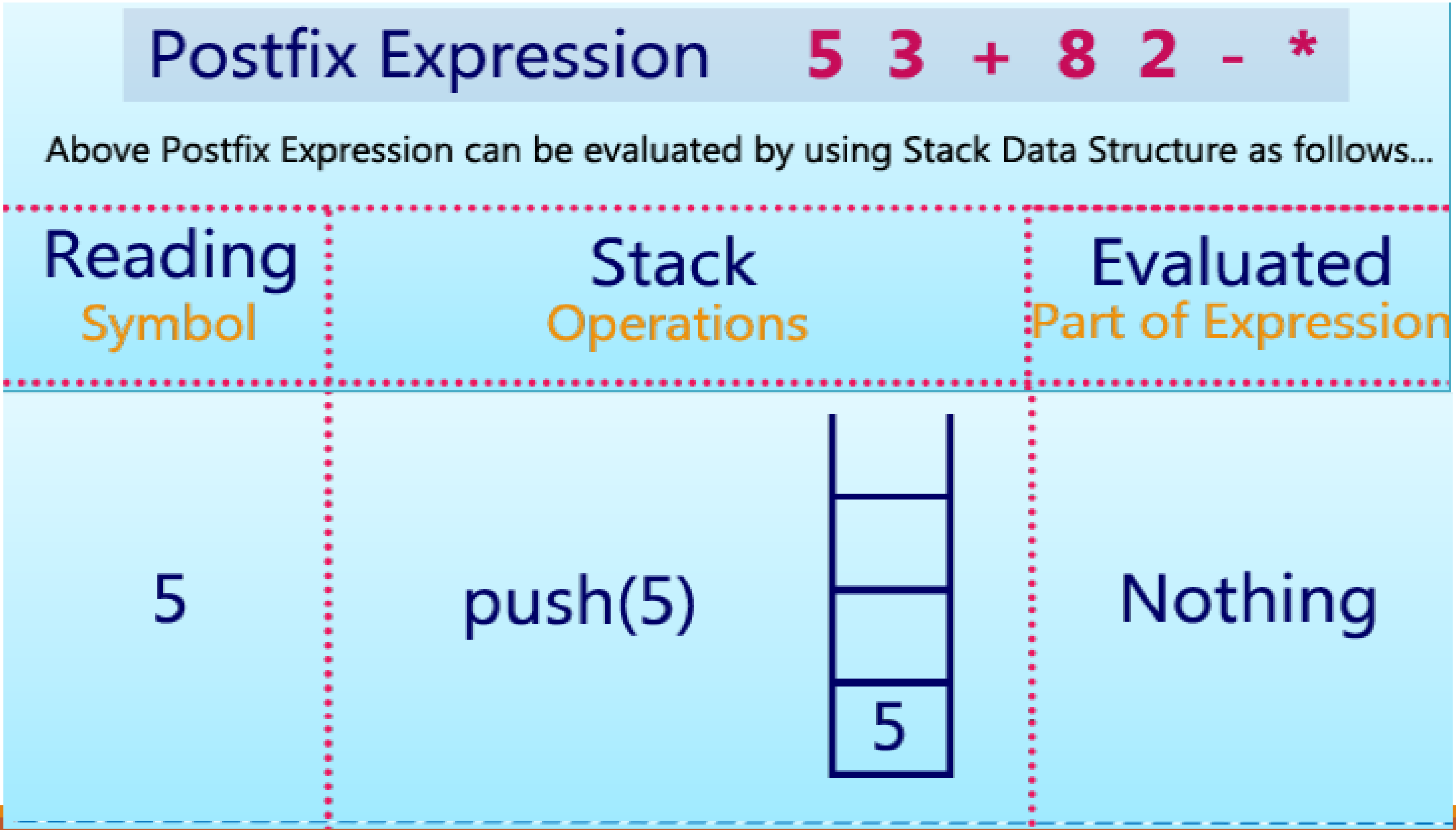


Postfix Expression **5 3 + 8 2 - ***

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty	Nothing

Postfix Evaluation Example

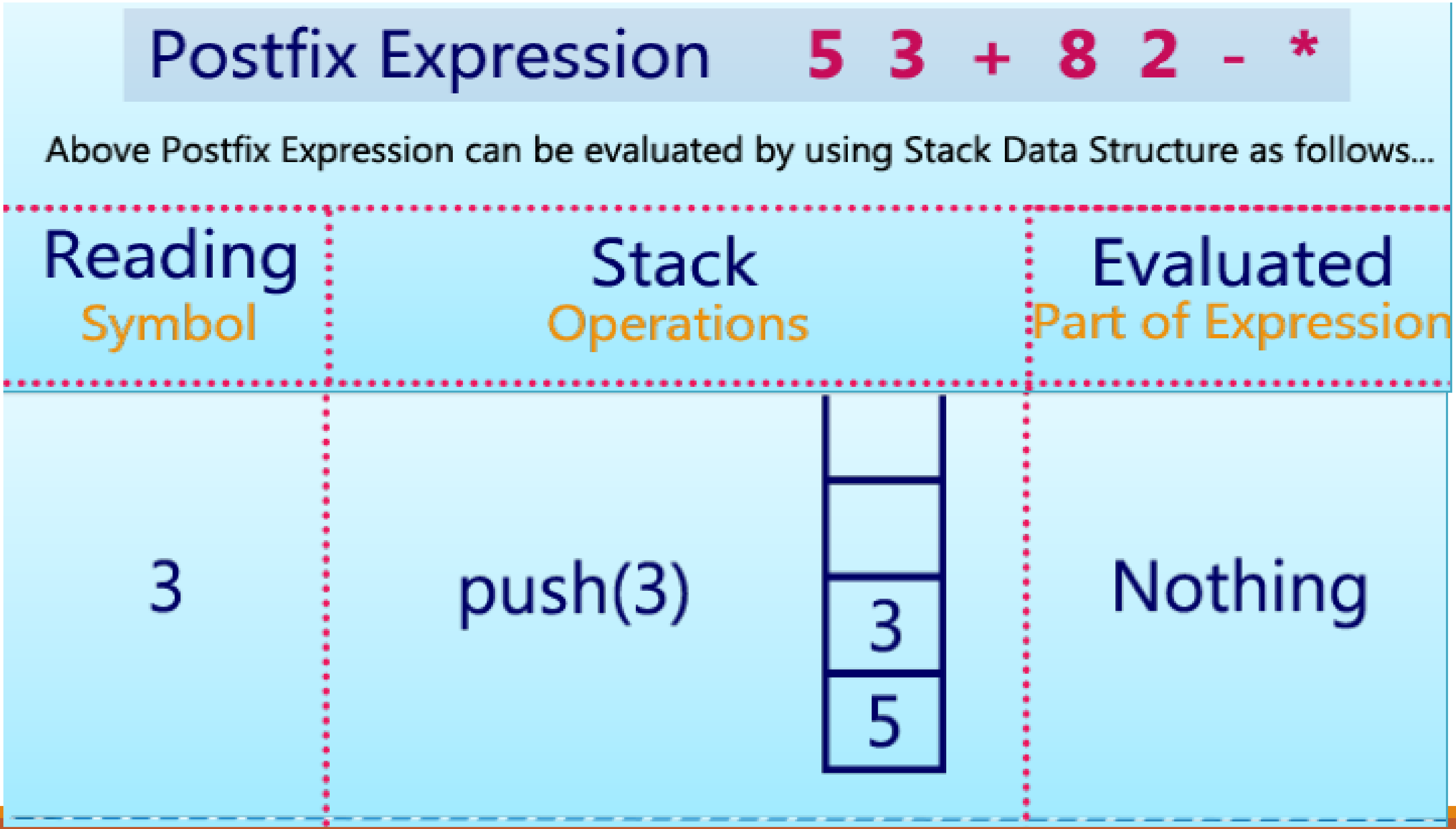


Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
5	push(5)	Nothing

Postfix Evaluation Example

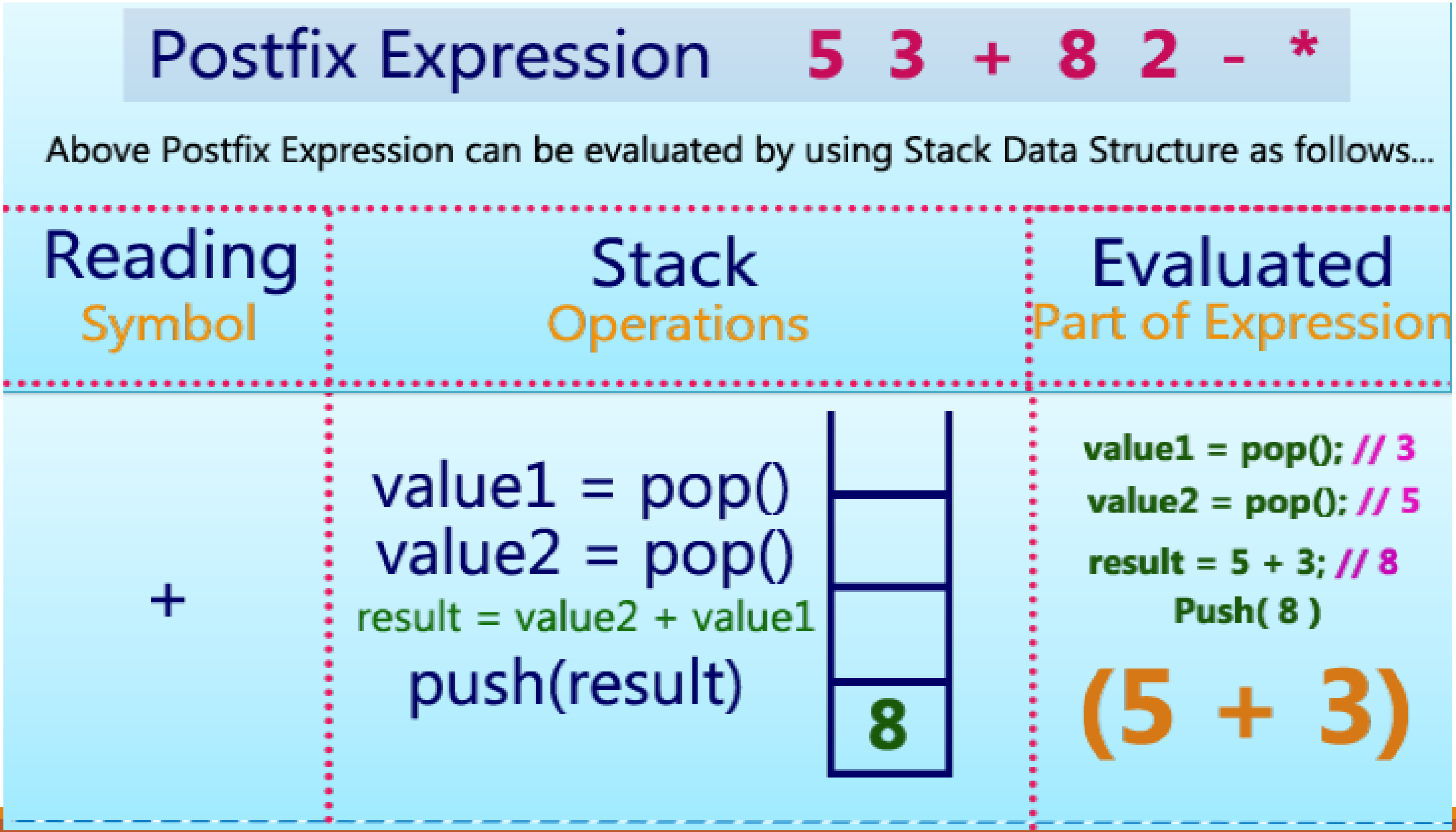


Postfix Expression **5 3 + 8 2 - ***

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
3	push(3)	Nothing

Postfix Evaluation Example



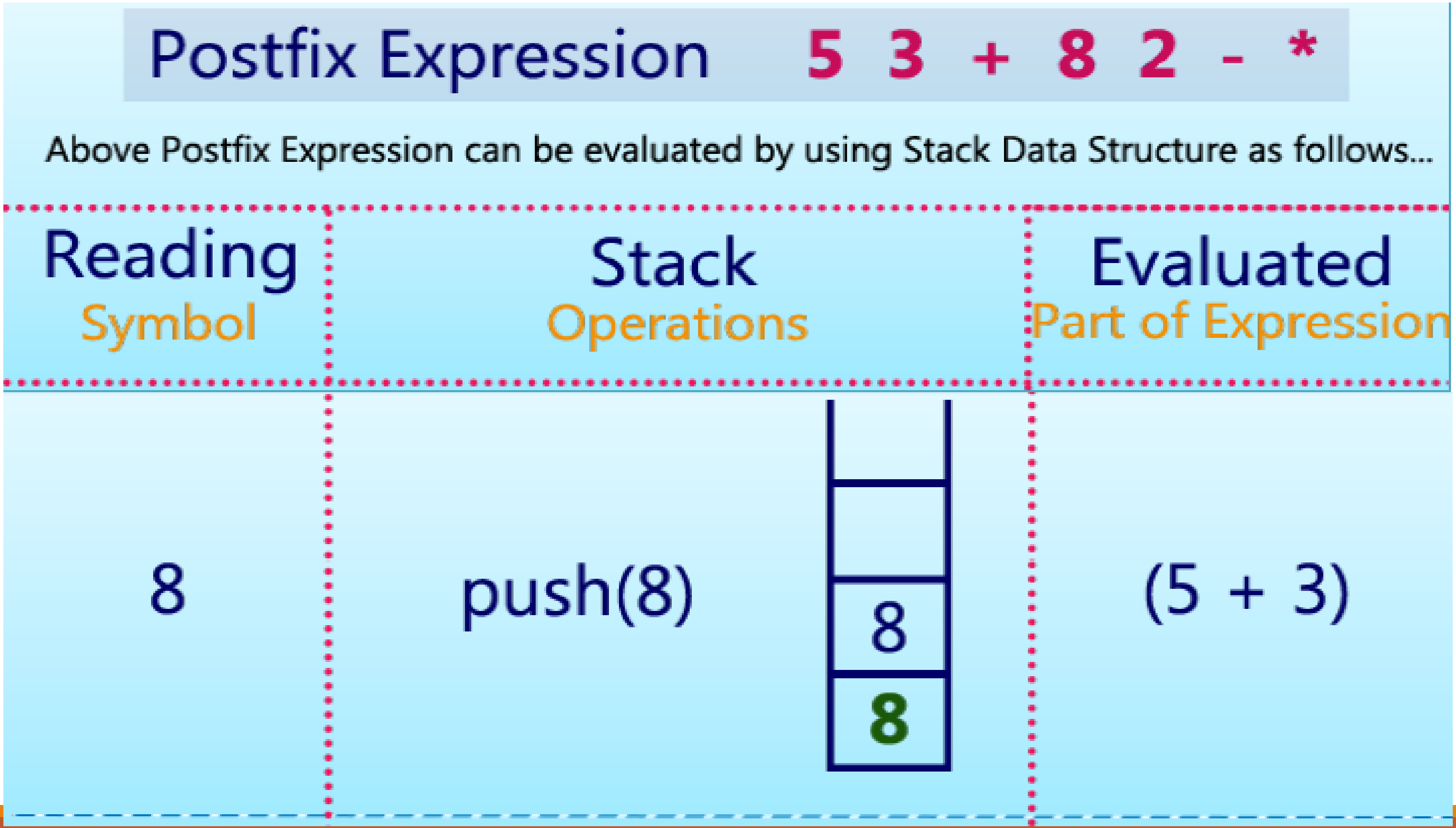
Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations		Evaluated Part of Expression

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
		8	value1 = pop(); // 3
			value2 = pop(); // 5
			result = 5 + 3; // 8
			Push(8)
			(5 + 3)

Postfix Evaluation Example

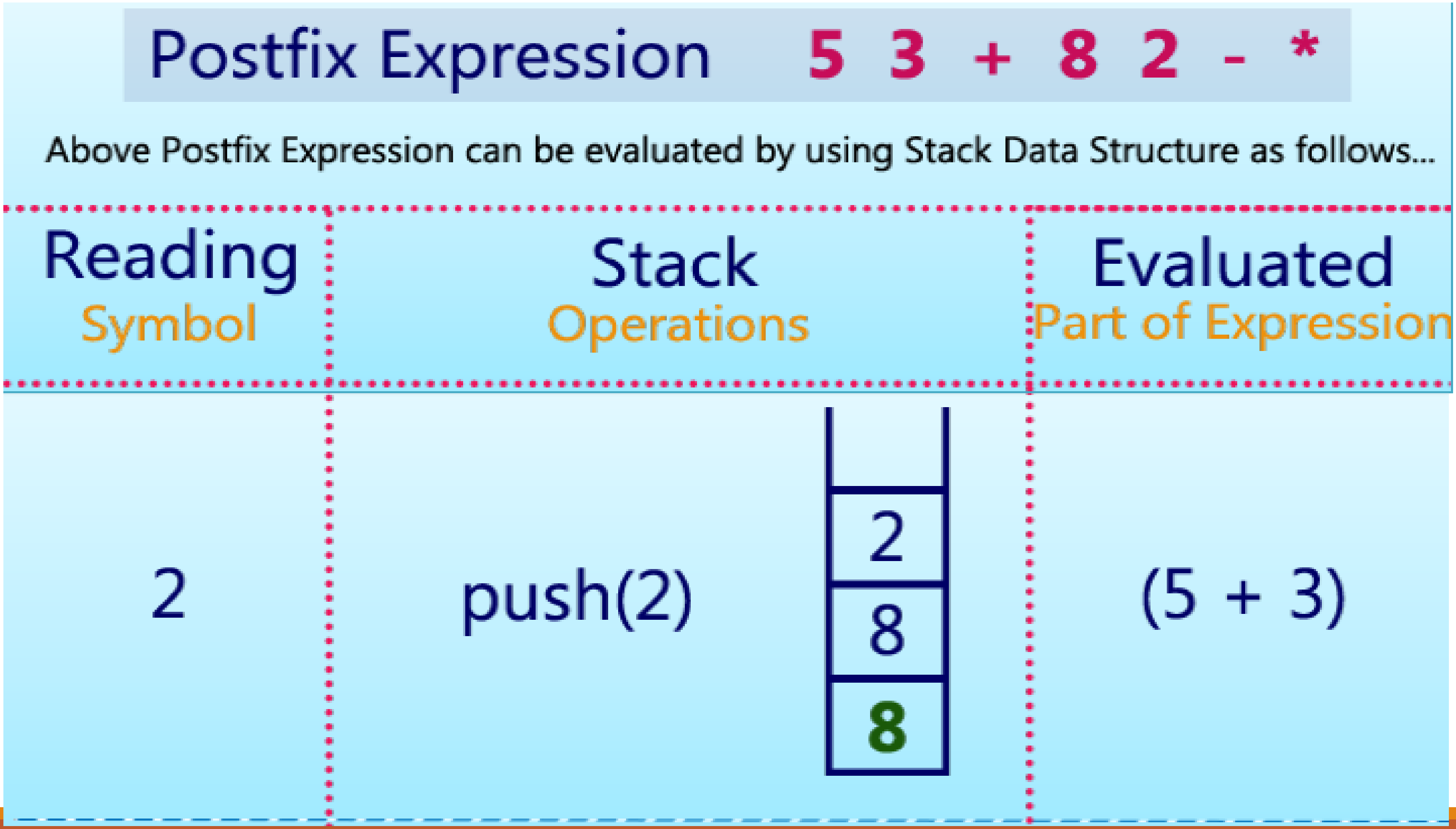


Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
8	push(8)	(5 + 3)

Postfix Evaluation Example

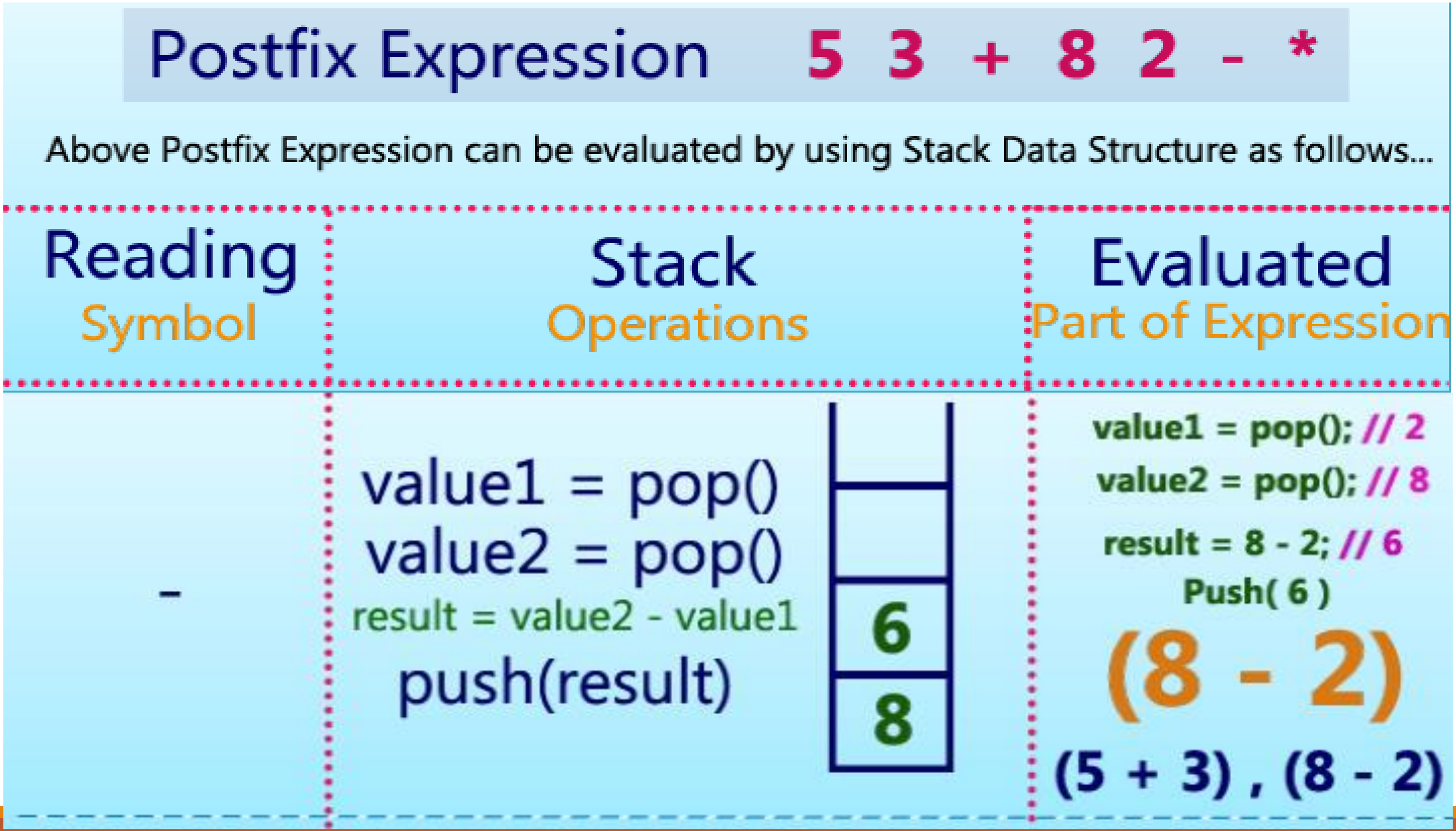


Postfix Expression **5 3 + 8 2 - ***

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations		Evaluated Part of Expression
2	push(2)	2 8 8	(5 + 3)

Postfix Evaluation Example



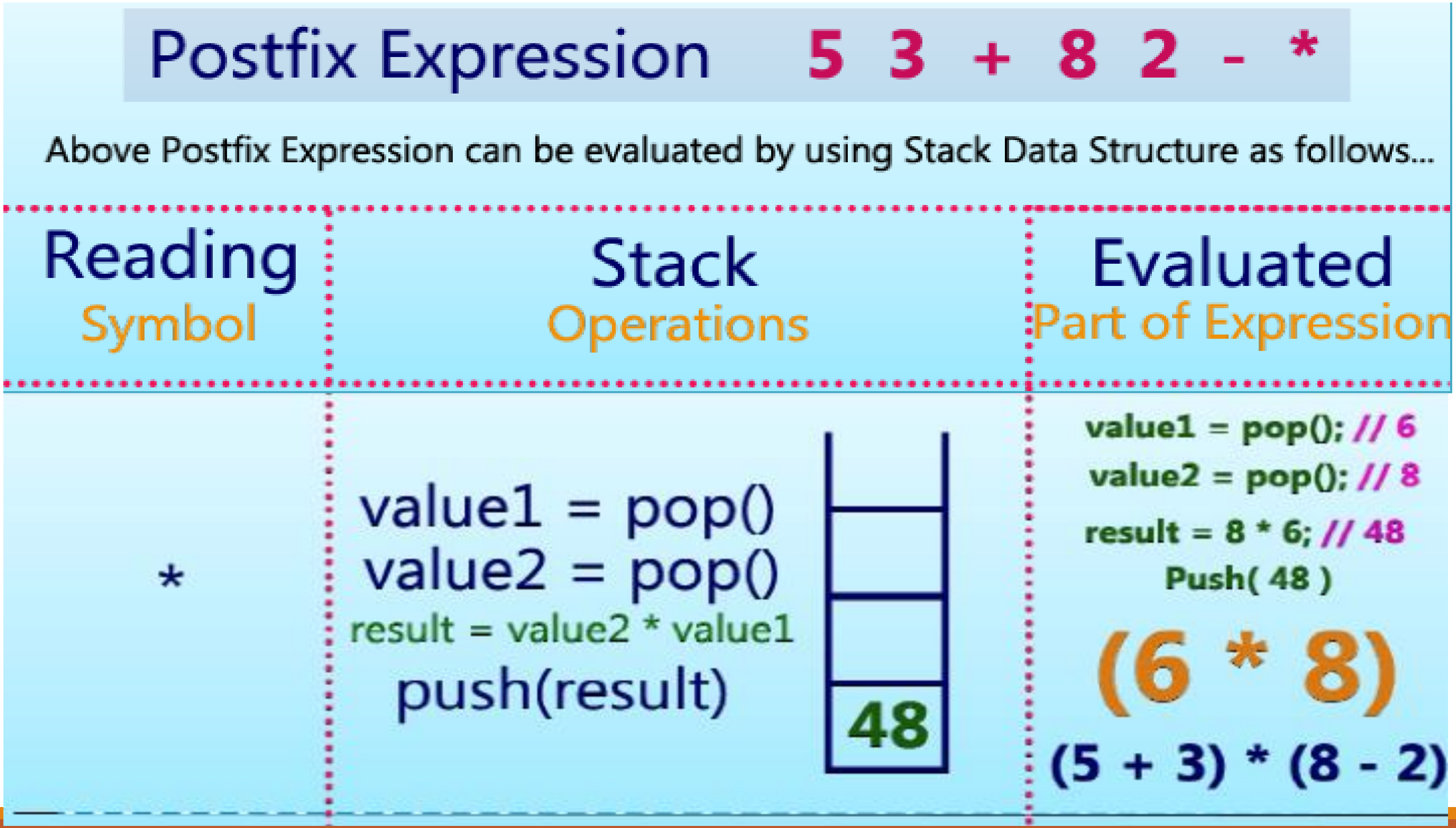
Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations		Evaluated Part of Expression

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
		6
		8
			value1 = pop(); // 2
			value2 = pop(); // 8
			result = 8 - 2; // 6
			Push(6)
			(8 - 2)
			(5 + 3) , (8 - 2)

Postfix Evaluation Example



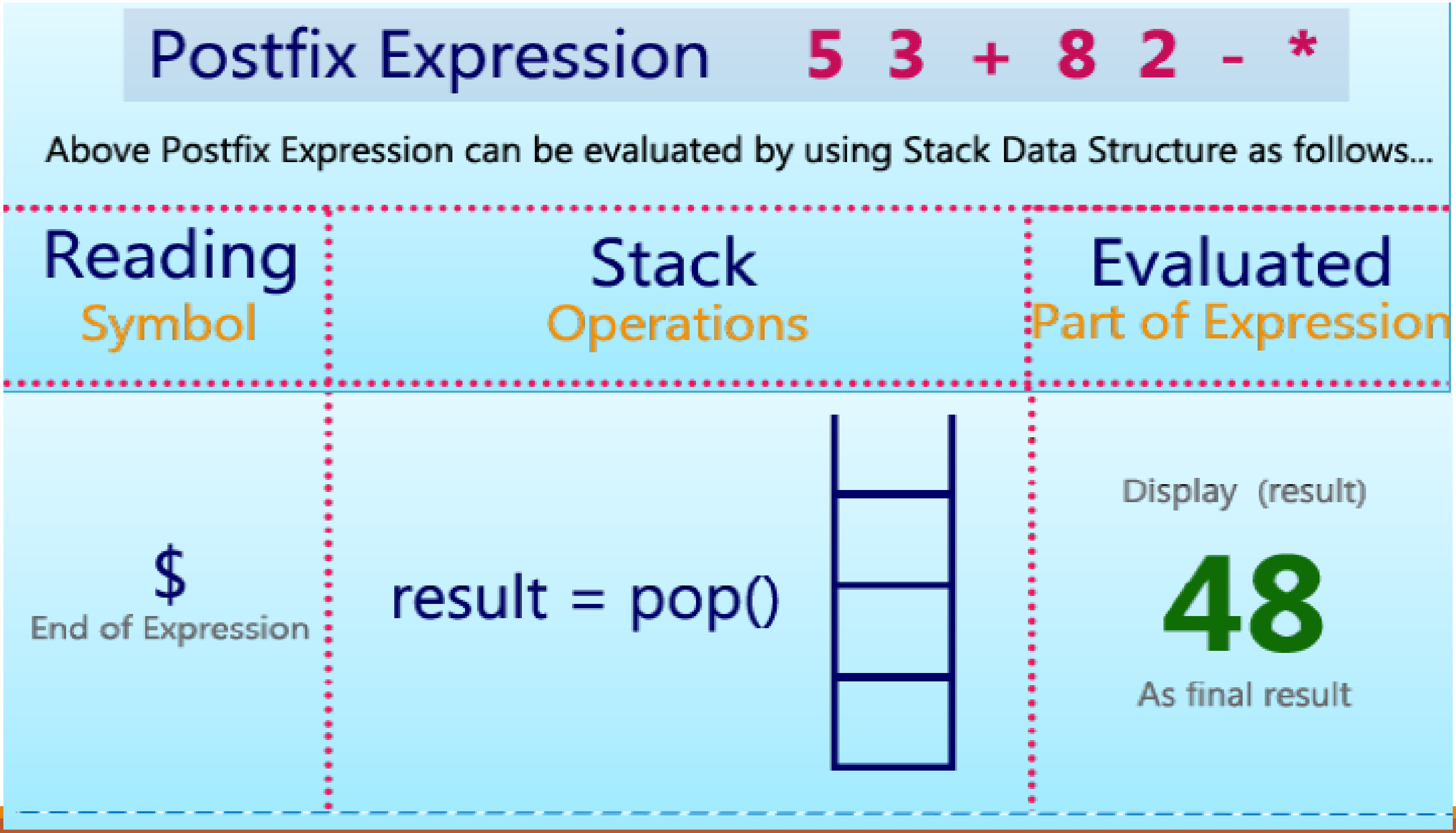
Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result)
	48
		value1 = pop(); // 6
		value2 = pop(); // 8
		result = 8 * 6; // 48
		Push(48)
		(6 * 8)
		(5 + 3) * (8 - 2)

Postfix Evaluation Example

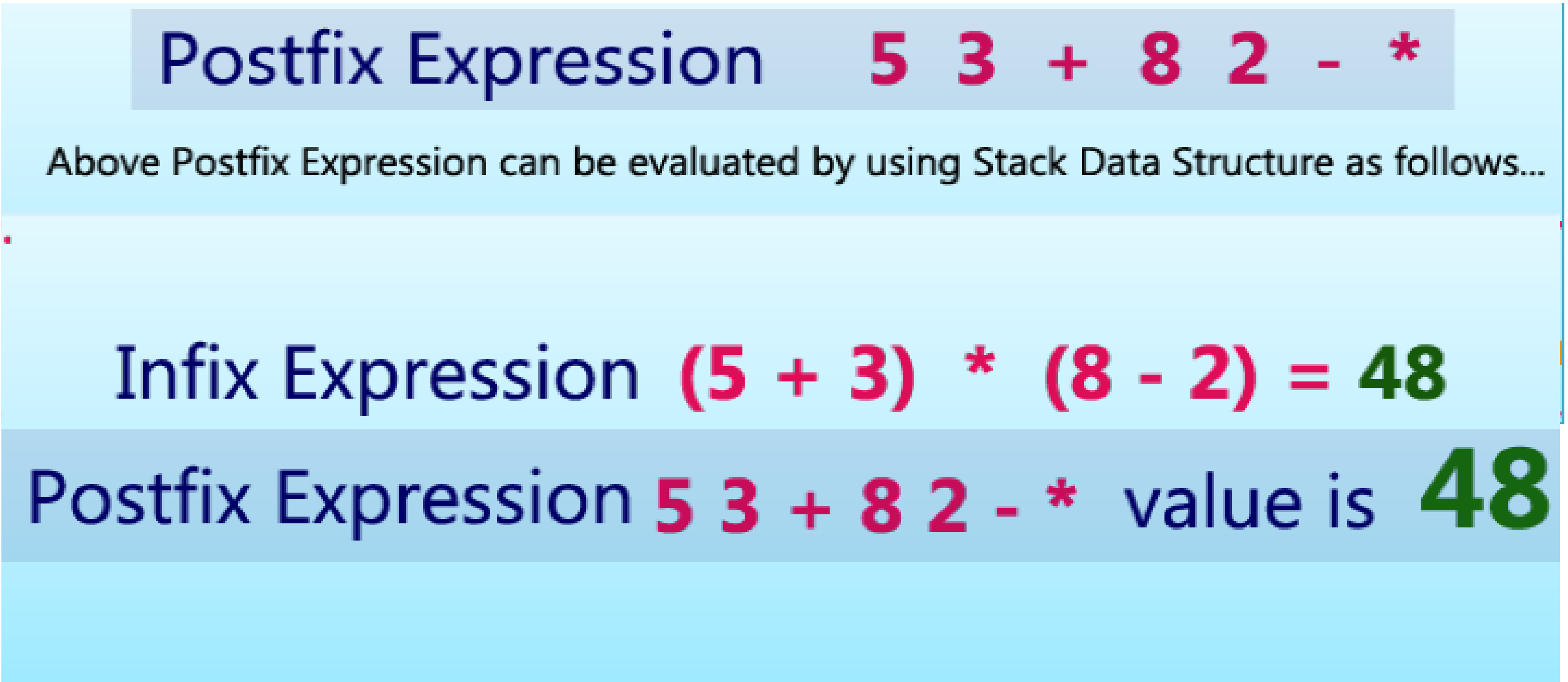


Postfix Expression **5 3 + 8 2 - ***

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
$ End of Expression	result = pop()	Display (result) **48** As final result

Postfix Evaluation Example

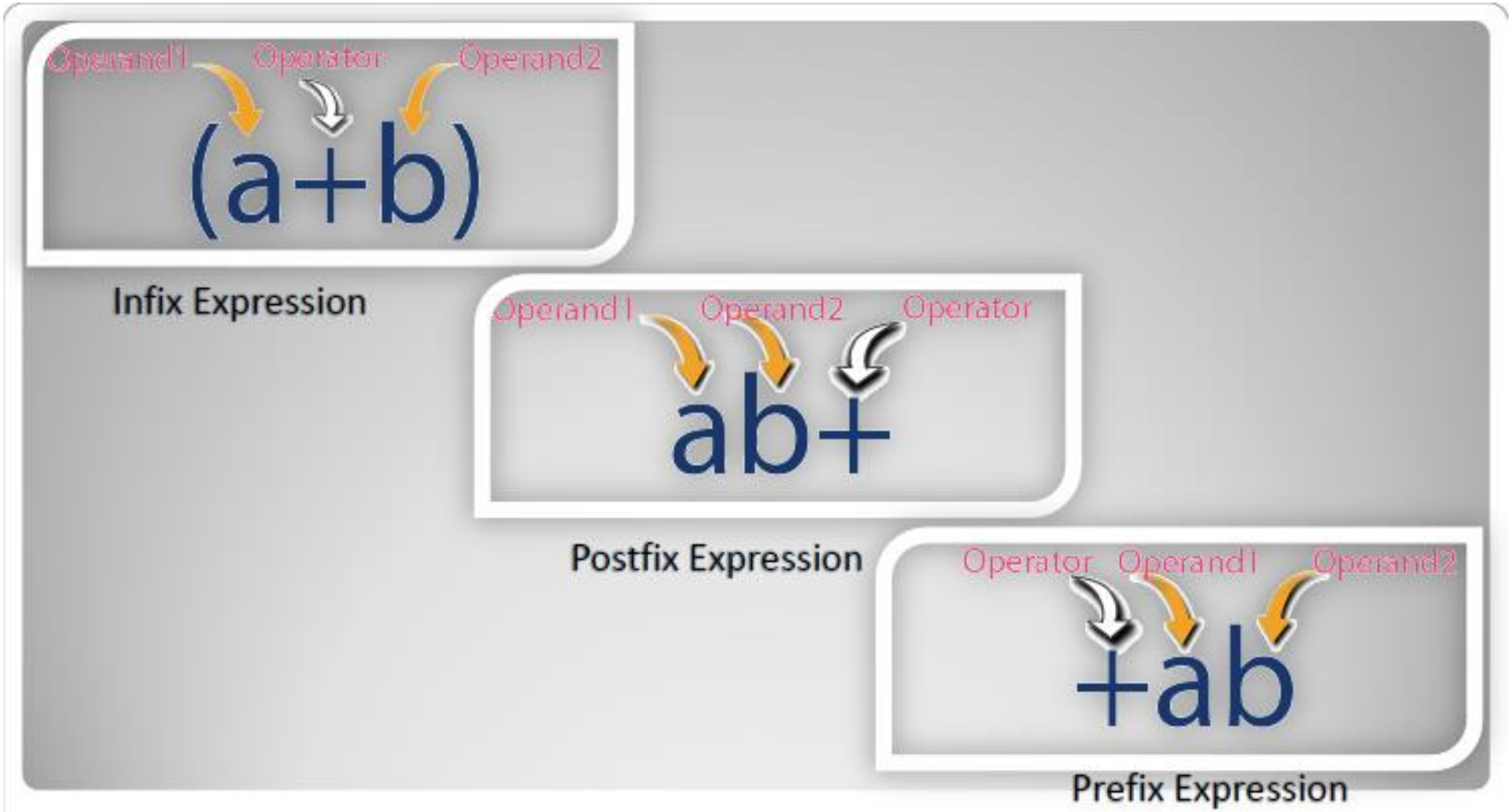


Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Infix Expression (5 + 3) * (8 - 2) = 48

Postfix Expression 5 3 + 8 2 - * value is 48



Expression Conversion

Expression Conversion

- Any expression can be represented using three types of expressions (Infix, Postfix and Prefix).

- We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

- To convert any Infix expression into Postfix or Prefix expression we can use the following procedure:

  1. Find all the operators in the given Infix Expression.
  2. Find the order of operators evaluated according to their Operator precedence.
  3. Convert each operator into required type of expression (Postfix or Prefix) in the same order.

Expression Conversion Example

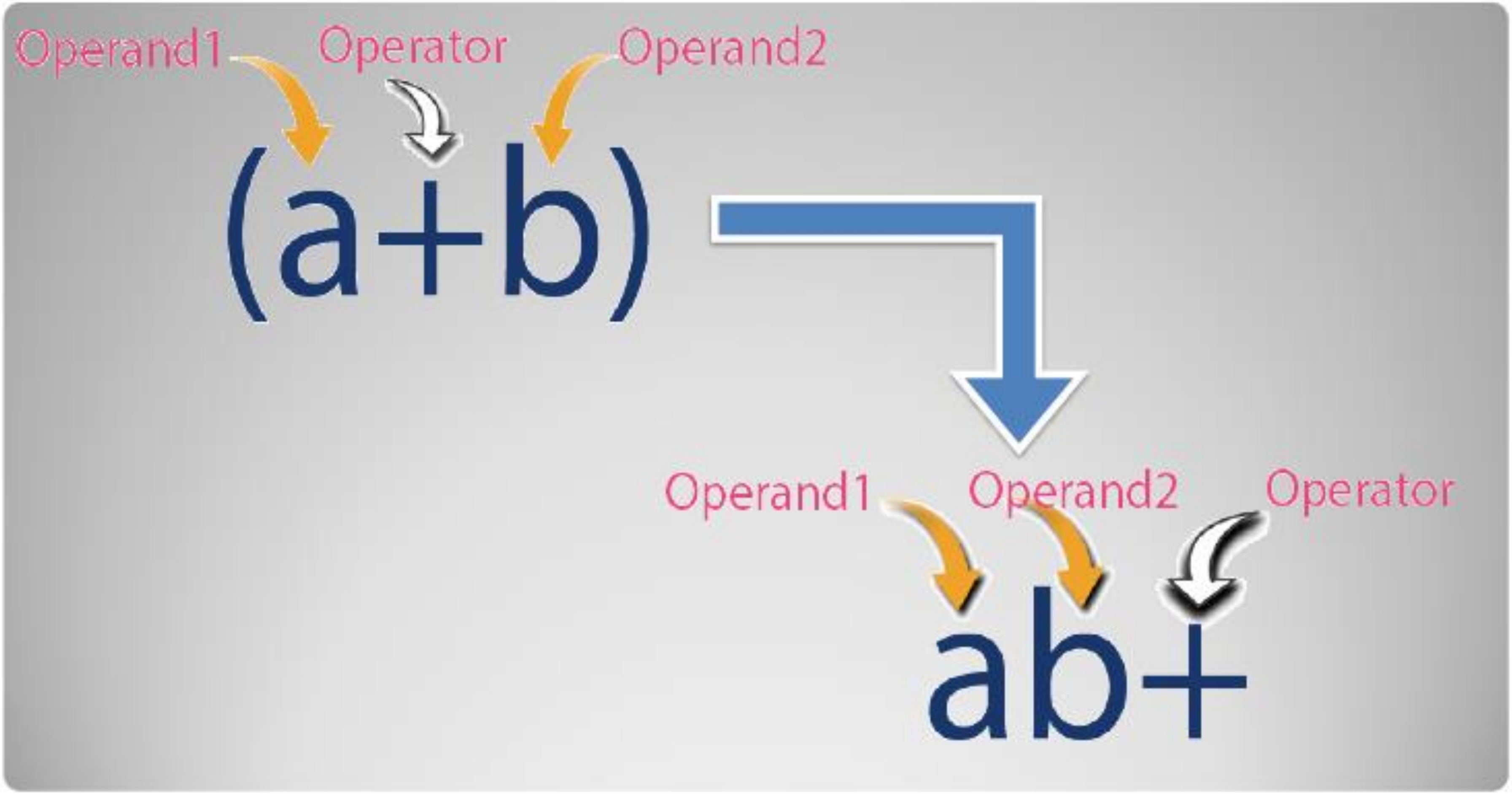
Consider the following Infix Expression to be converted into Postfix Expression.

D = A + B * C

- **Step1:**The Operators in the given Infix Expression: **=,+,***
- **Step2:**The Order of Operators according to their preference: ***,+,=**
- **Step3:**Now, convert the first operator ***-----D=A+BC***
- **Step4:**Convert the next operator **+-----D=ABC*+**
- **Step5:**Convert the next operator **=-----DABC*+=**

Finally, given Infix Expression is converted into Postfix Expression as follows...

D A B C * + =



Infix To Postfix

Infix to postfix Conversion

The rules to be remembered during infix to postfix conversion are:

- 1.Parenthesize the expression starting from left to light.
- 2.During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized.
- 3.The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
- 4.Once the expression is converted to postfix form, remove the parenthesis.

Infix to postfix Conversion

Example: Evaluate the postfix expression of the infix exp.

(A + B) * C / D + E ^ A / B

Solution:

[(AB+)*C/D]+[(EA^)/B]

[(AB+)*C/D]+[(EA^)B/]

[(AB+)C*D/]+[(EA^)B/]

(AB+)C*D/(EA^)B/+

AB+C*D/EA^B/+ “PostfixForm”

Infix to postfix Algorithm

Input: infix_str

Output: postfix_str

Algorithm:

1.Push “(” onto stack, and add “)” to the end of infix_str.

2.Scan infix_str from left to right and repeat Steps 3 to 6 for each element of infix_str until the stack is empty.

3.If an operand is encountered, add it to postfix_str.

4.If a left parenthesis is encountered, push it onto stack.

5.If an operator ⊗ is encountered, then:

(a)Repeatedly pop from stack and add to postfix_str each operator (on the top of stack), which has the same precedence as, or higher precedence than ⊗.

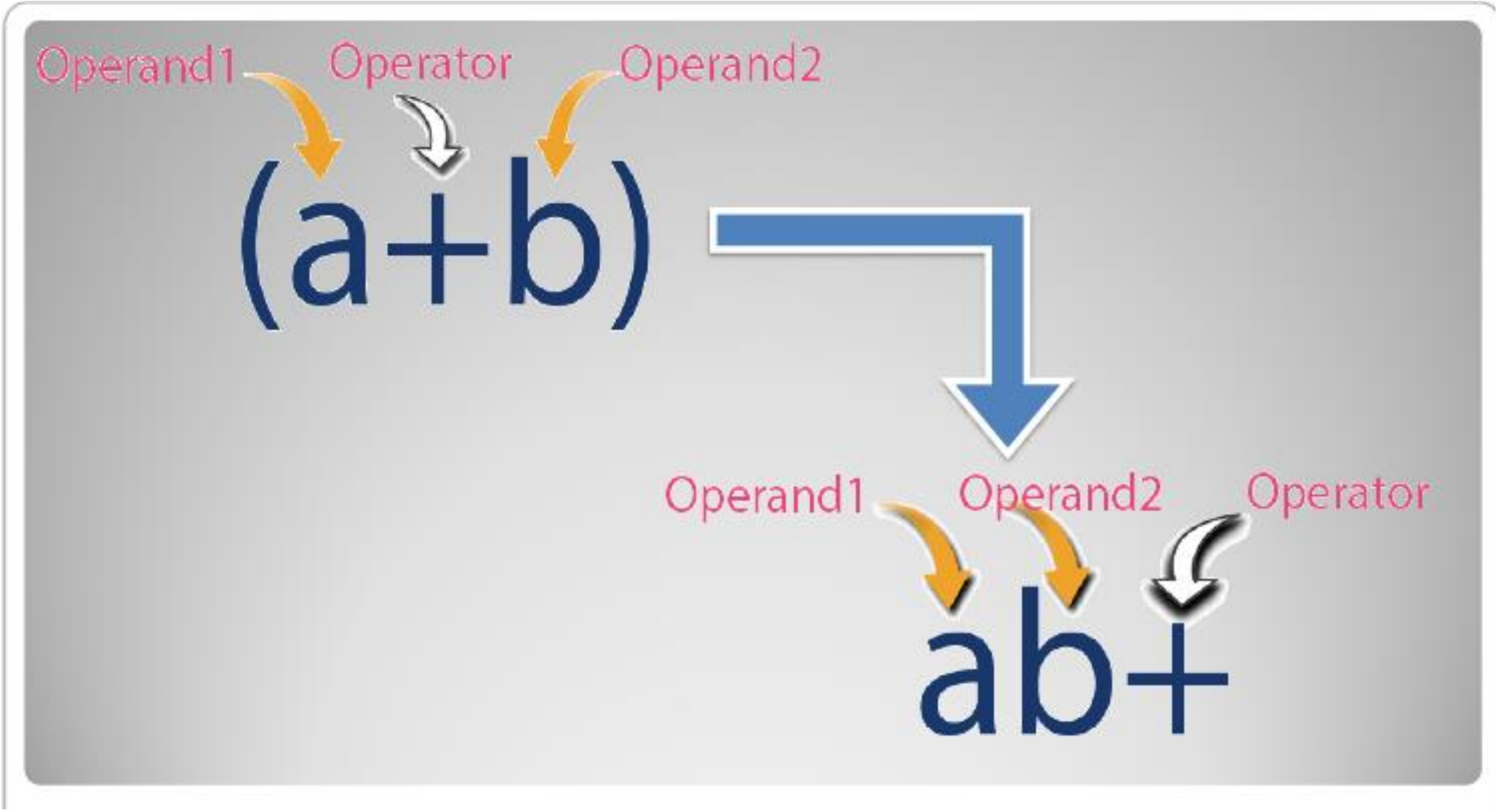
(b)Add operator ⊗ to stack.

6.If a right parenthesis) is encountered, then:

(a)Repeatedly pop from stack and add to postfix_str on the top of stack until a left parenthesis (is encountered.

(b)Remove the left parenthesis.

7.Exit.



Infix To Postfix Examples

Infix to Postfix Example

A + B * C)

symb	Postfix string	opstk
		(
A	A	(
+	A	(+
B	A B	(+
*	AB	(+ *
C	ABC	(+ *
)	ABC*+	(

Infix to Postfix Example

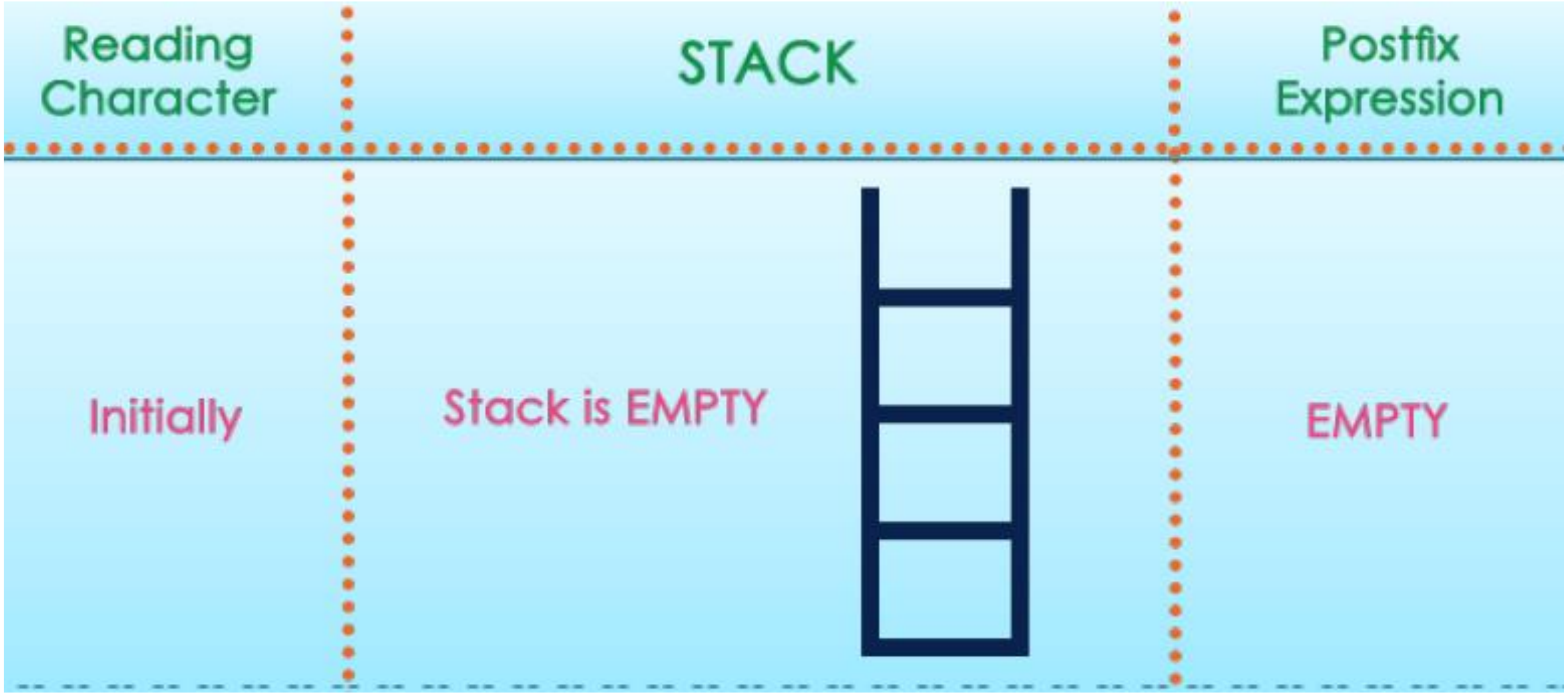
(A + B) * C)

symb	Postfix string	opstk
(		((
A	A	((
+	A	((+
B	AB	((+
)	AB+	(
*	AB+	(*
C	AB+C	(*
)	AB+C*	(

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)

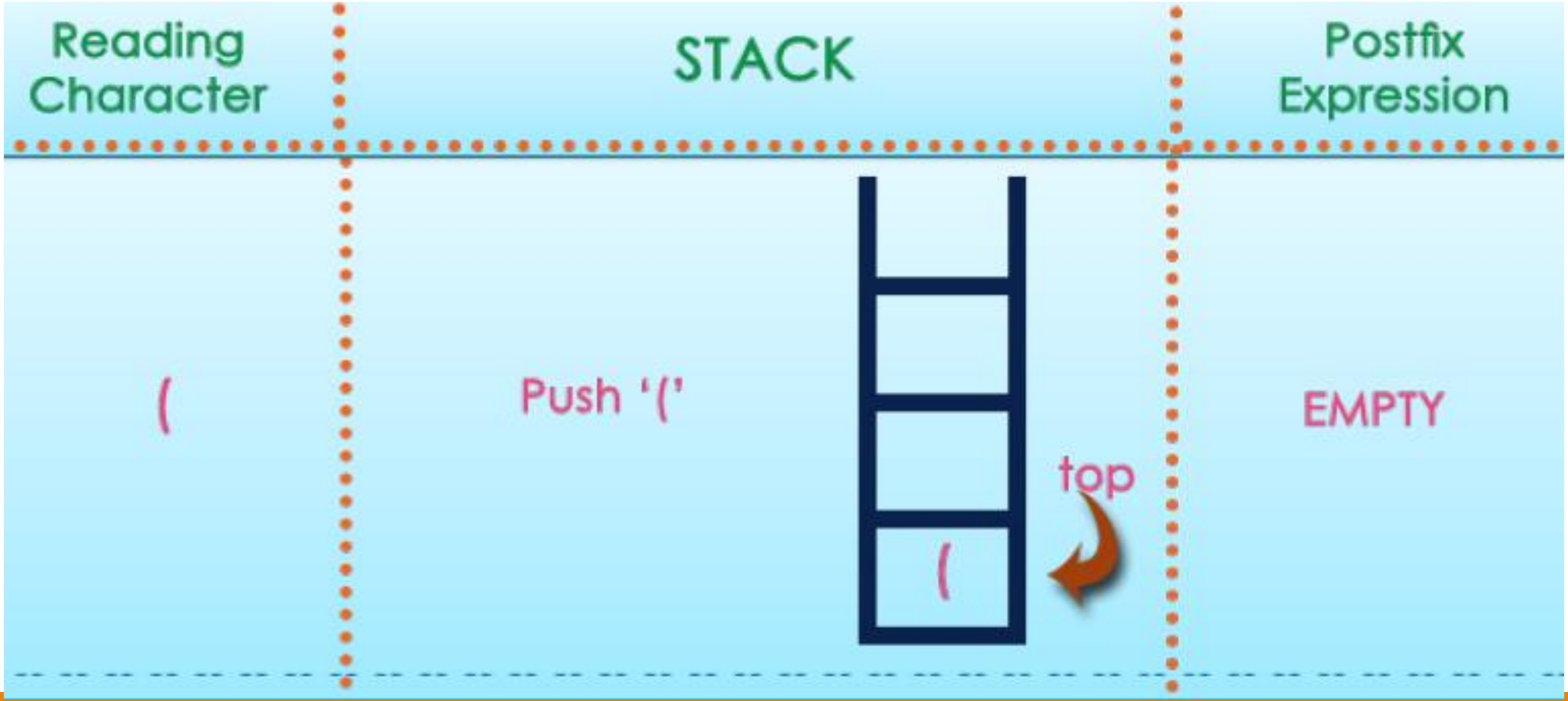


Reading Character	STACK	Postfix Expression
Initially	Stack is EMPTY	EMPTY

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)

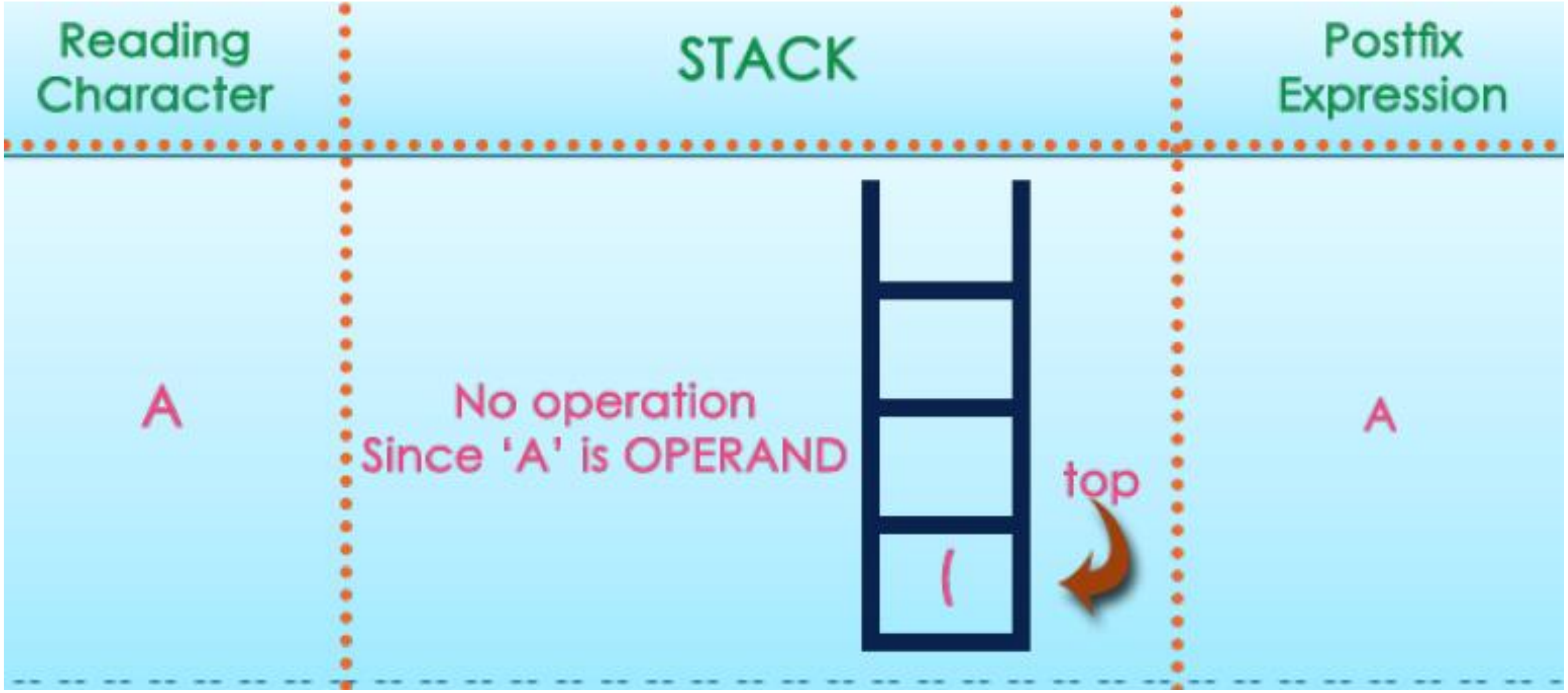


Reading Character	STACK	Postfix Expression
(	Push '('	EMPTY

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)

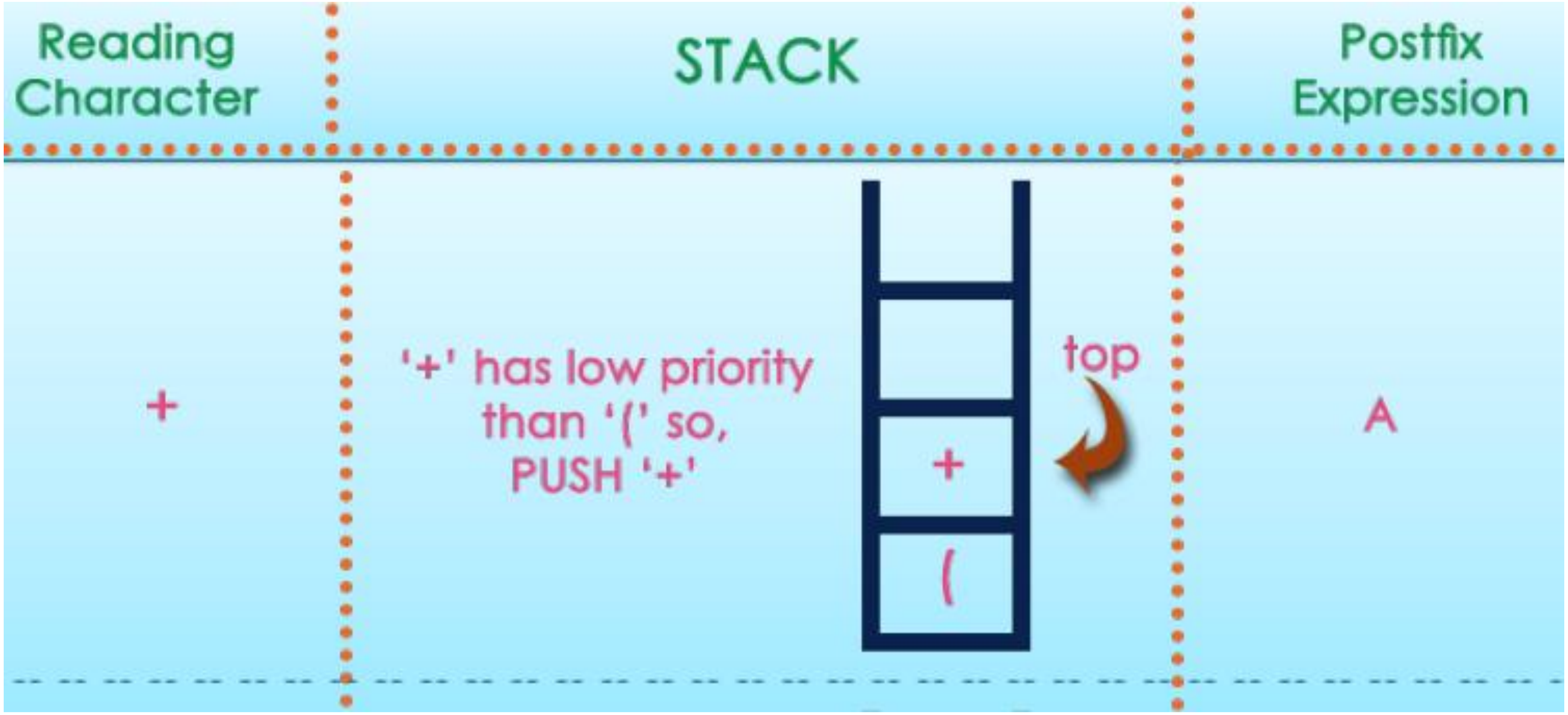


Reading Character	STACK	Postfix Expression
A	No operation Since 'A' is OPERAND	A

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)

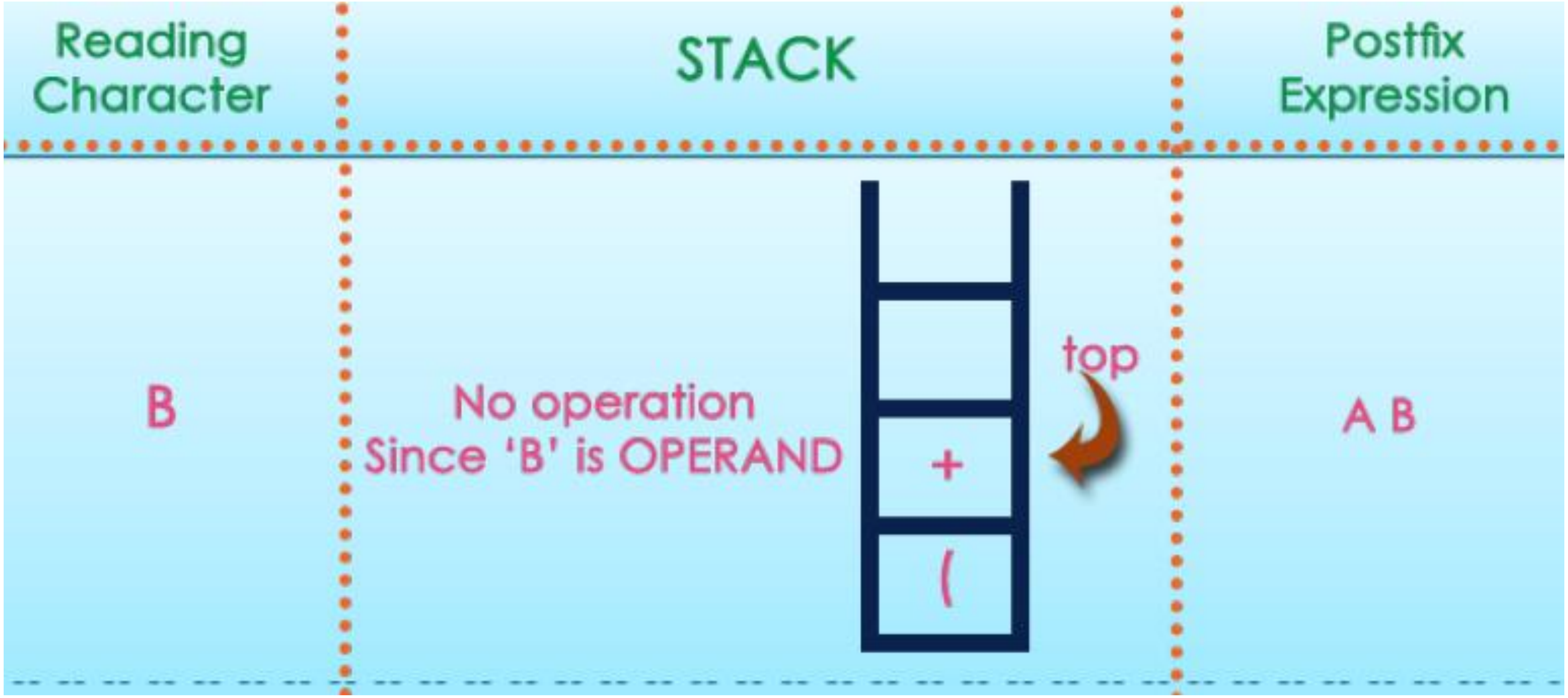


Reading Character	STACK	Postfix Expression
+	'+' has low priority than '(' so, PUSH '+'	A

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)

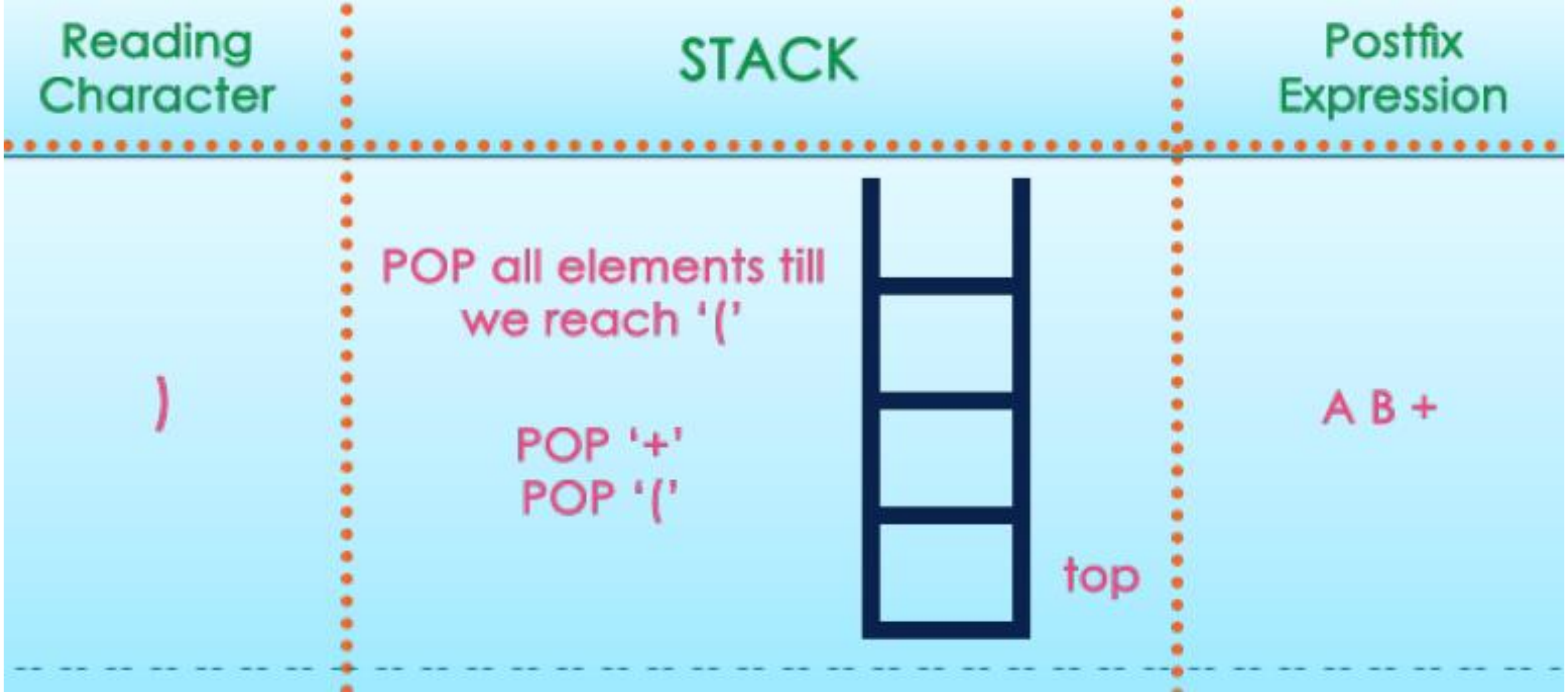


Reading Character	STACK	Postfix Expression
B	No operation Since 'B' is OPERAND	A B

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)

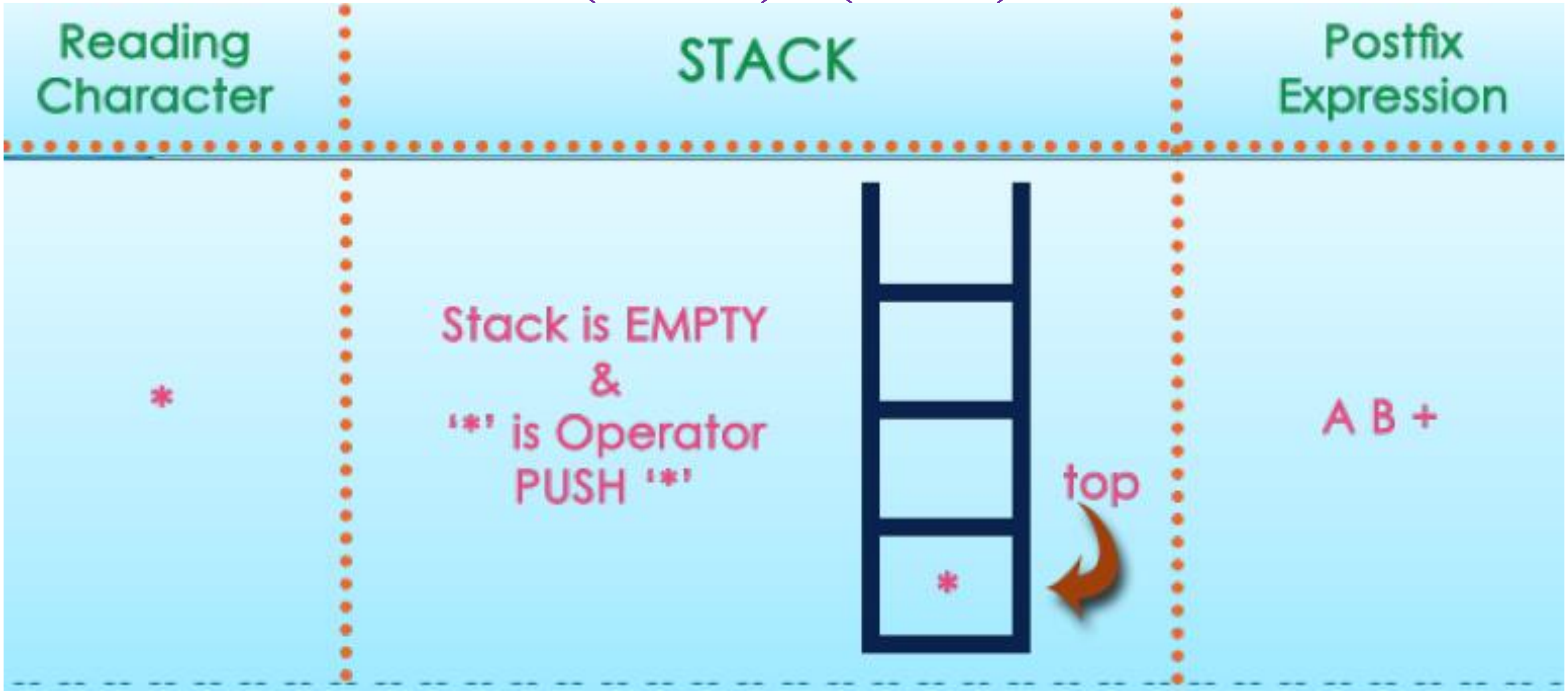


Reading Character	STACK	Postfix Expression
)	POP all elements till we reach '(' POP '+' POP '(' top	A B +

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)

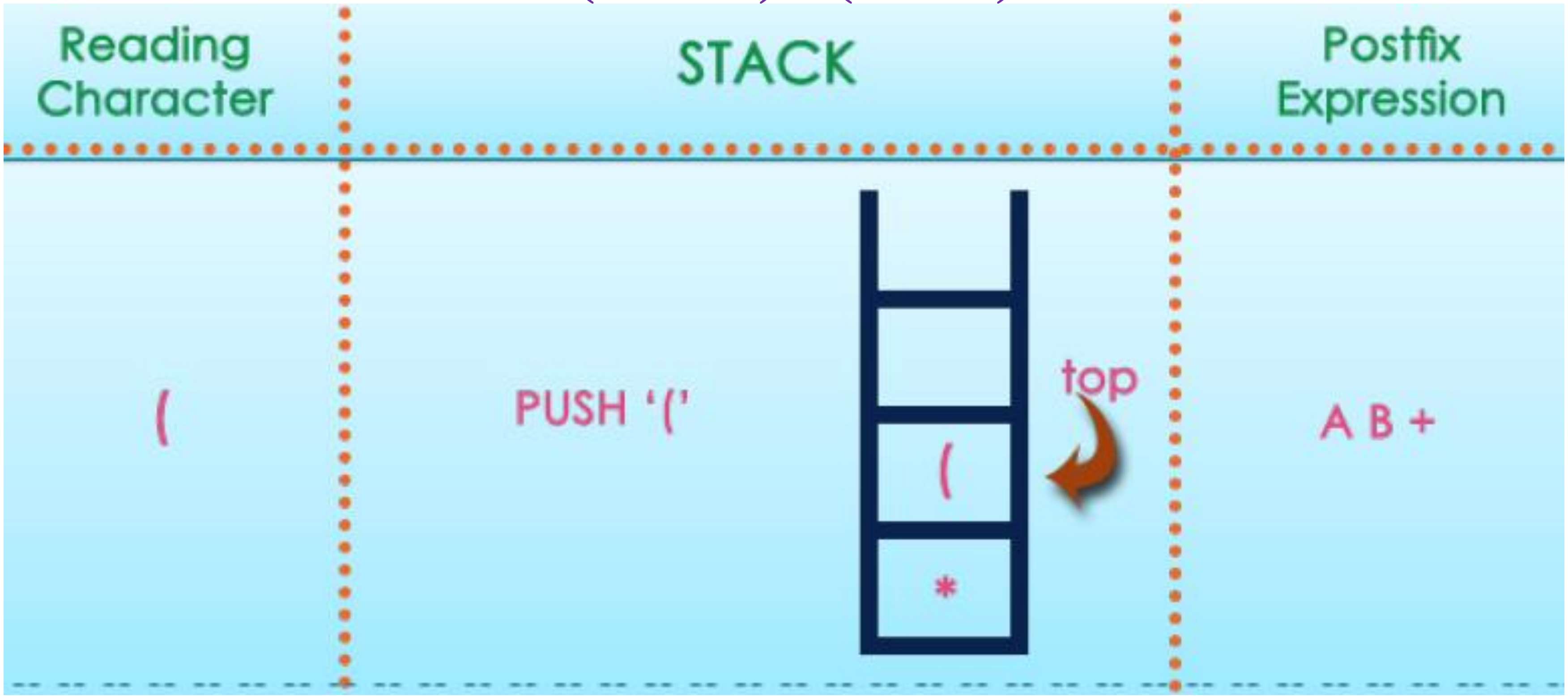


Reading Character	STACK	Postfix Expression
*	Stack is EMPTY & '*' is Operator PUSH '*'	A B +

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)

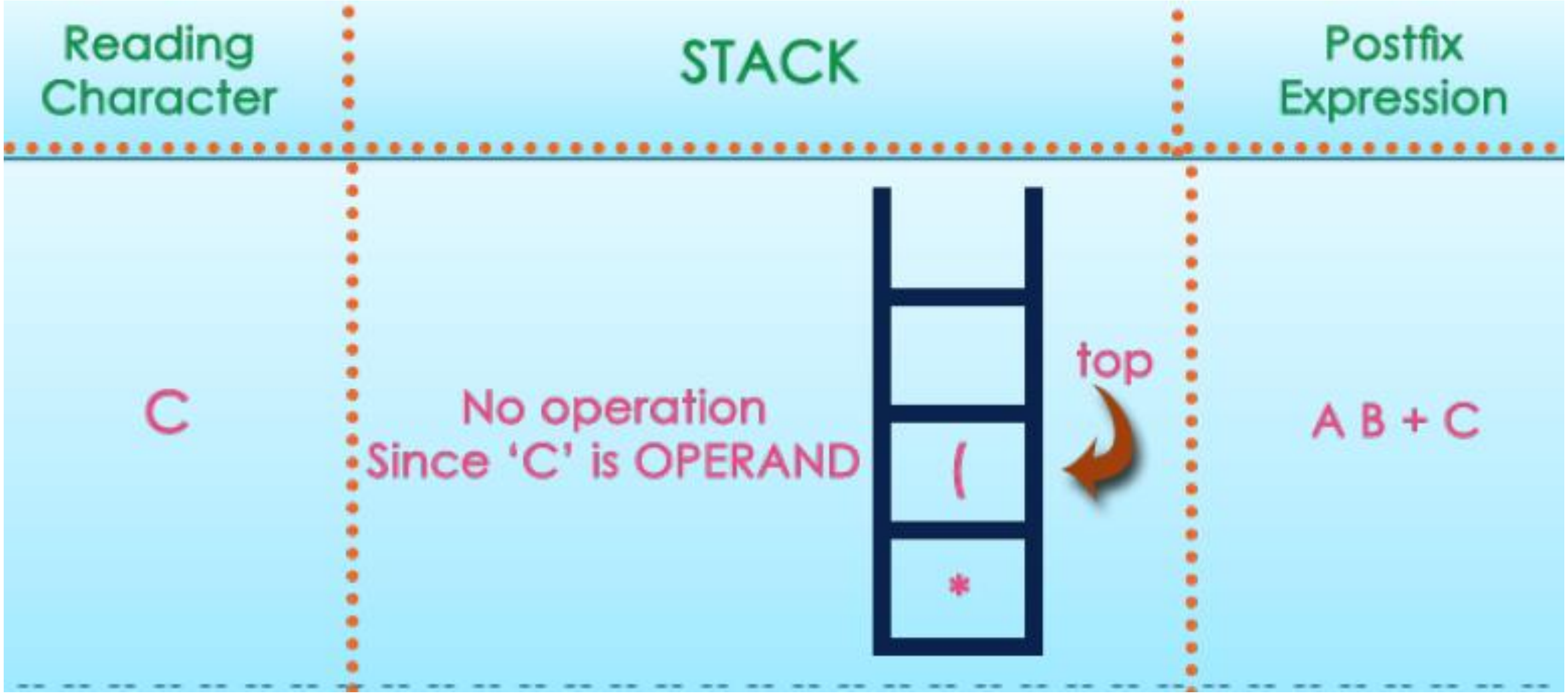


Reading Character	STACK	Postfix Expression
(	PUSH '('	A B +

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)

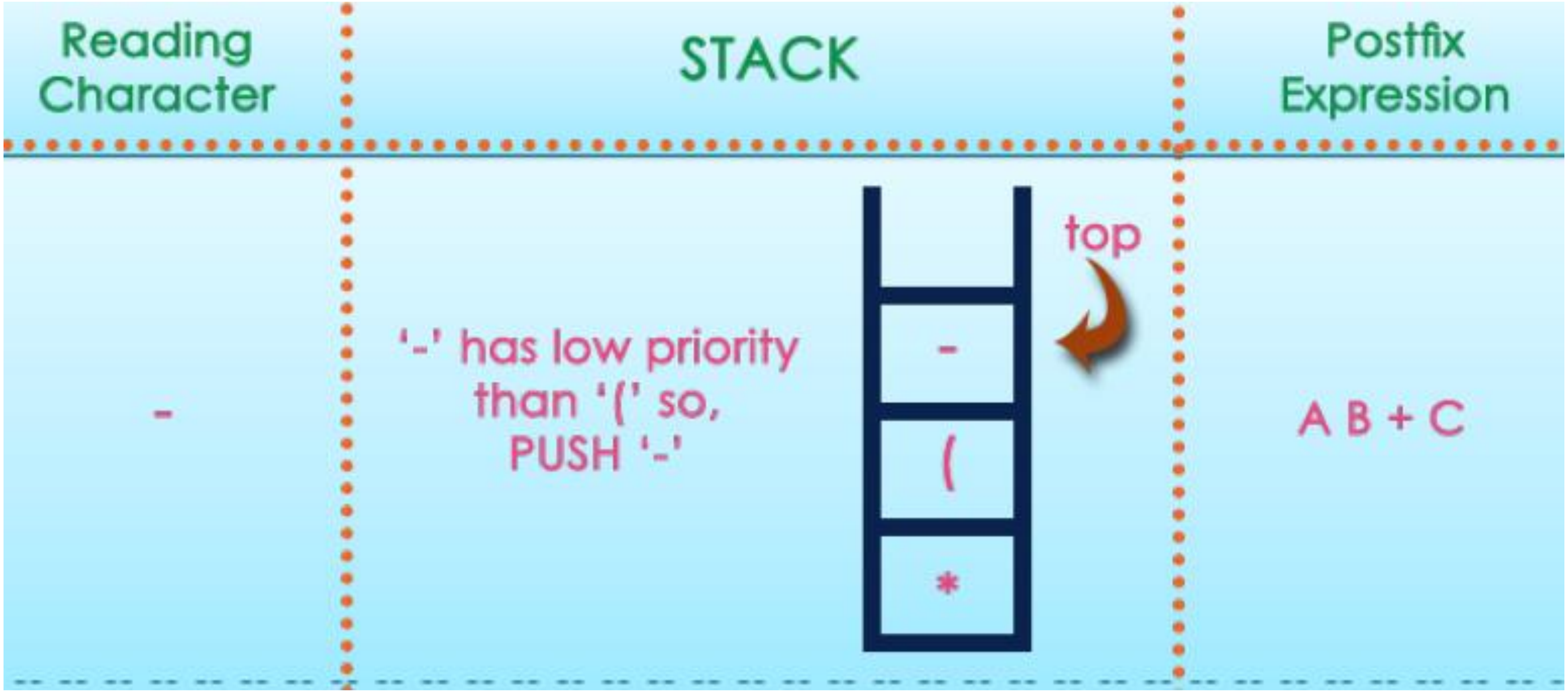


Reading Character	STACK	Postfix Expression
C	No operation Since ‘C’ is OPERAND	A B + C

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)

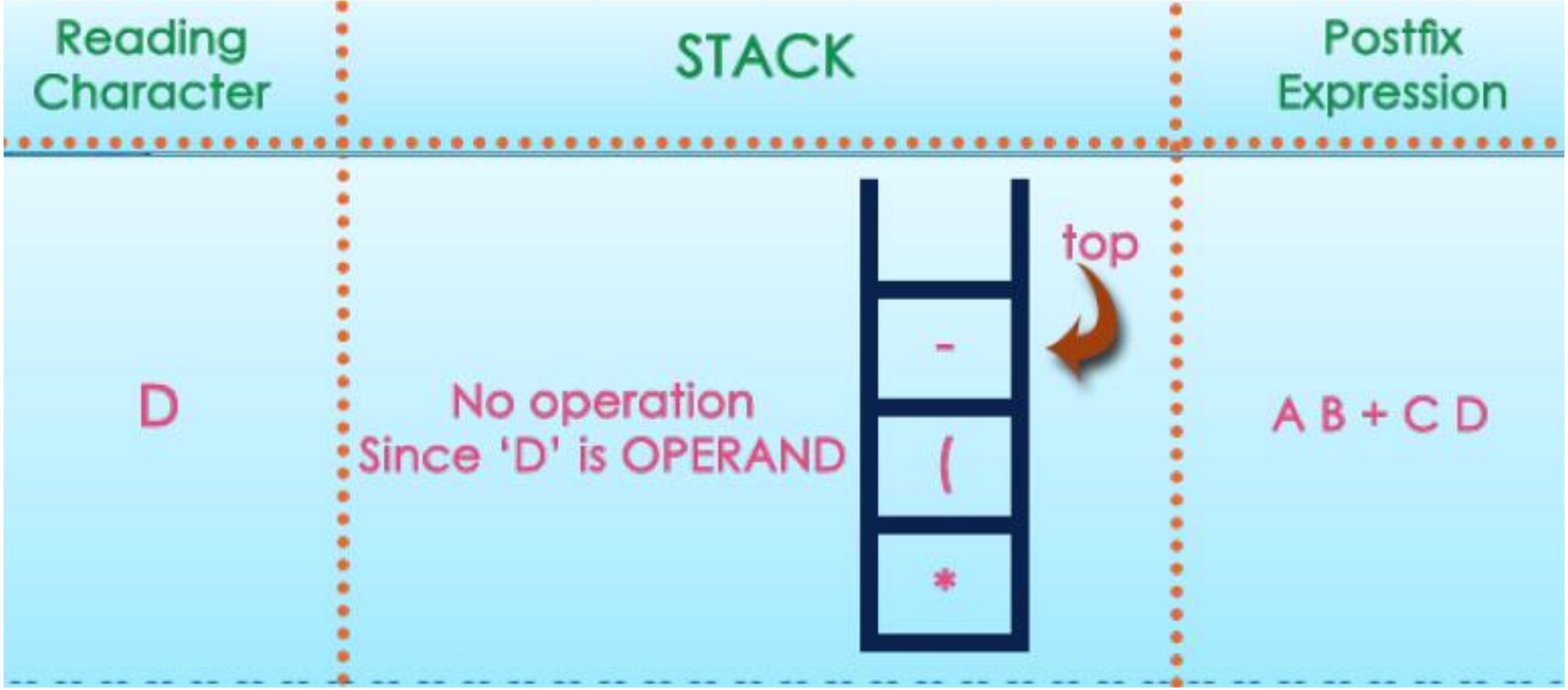


Reading Character	STACK	Postfix Expression
-	'-' has low priority than '(' so, PUSH '-'	A B + C

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)

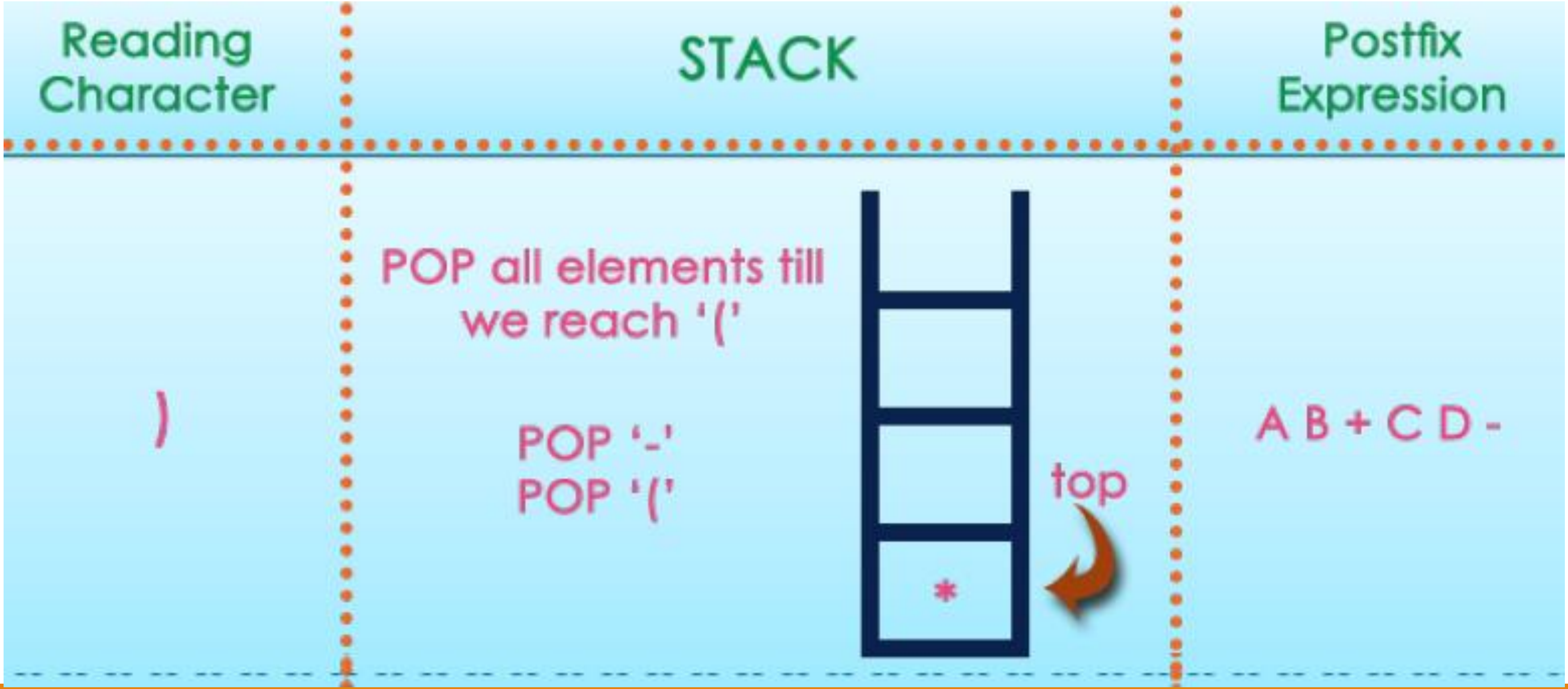


Reading Character	STACK	Postfix Expression
D	No operation Since 'D' is OPERAND	A B + C D

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)

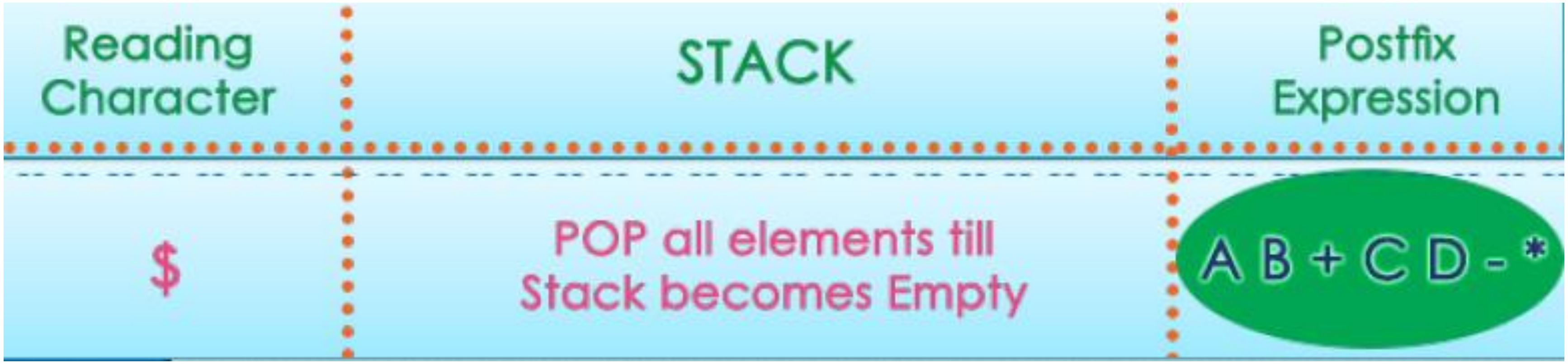


Reading Character	STACK	Postfix Expression
)	POP all elements till we reach ‘(' POP ‘-' POP ‘('	A B + C D -

Infix to Postfix Example

Consider the following Infix Expression...

(A + B) * (C -D)



Reading Character	STACK	Postfix Expression
$	POP all elements till Stack becomes Empty	A B + C D - *