Stack

Notations

- The way to write arithmetic expression is known as a **notation**.
- An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression.

These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

Infix Notation

- An infix notation is a notation in which an expression is written in a usual or normal format.
- It is a notation in which the operators lie between the operands.
- The examples of infix notation are A+B, A*B, A/B, etc.
- As we can see in the above examples, all the operators exist between the operands, so they are infix notations.
- Therefore, the syntax of infix notation can be written as:
- **<operand> <operator> <operand>**

Parsing Infix expressions

- In order to parse any expression, we need to take care of two things, i.e., **Operator precedence** and **Associativity**.
- Operator precedence means the precedence of any operator over another operator.

For example:

- A + B * C → A + (B * C)
- As the multiplication operator has a higher precedence over the addition operator so B * C expression will be evaluated first.
- The result of the multiplication of B * C is added to the A.

Precedence order

Operators	Symbols
Parenthesis	{ }, ( ), [ ]
Exponential notation	^
Multiplication and Division	*, /
Addition and Subtraction	+, -

Associativity through an example.

- 1 + 2*3 + 30/5
- * and / have the same precedence, so we will apply the **associativity rule.**
- * and / operators have the left to right associativity, so we will scan from the leftmost operator.
- The operator that comes first will be evaluated first.
- The operator * appears before the / operator, and multiplication would be done first.
- 1+ (2*3) + (30/5)
- 1+6+6 = 13

Infix Notation

- We write expression in **infix** notation,
- e.g. $\underline{a - b + c}$ where operators are used **in**-between operands.
- It is easy for us humans to read, write, and speak in infix notation.
- An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

- It is also known as **polish notation**.
- In prefix notation, an operator comes before the operands.

The syntax of prefix notation is given below:

- **<operator> <operand> <operand>**
- **For example,** if the infix expression is 5+1, then the prefix expression corresponding to this infix expression is +51.
- **If the infix expression is:**

a * b + c

(1) *ab+c

(2) +*abc

Prefix Notation

- Prefix notation is also known as **Polish Notation**.
- In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands.
- For example, <u>**+ab**</u>. This is equivalent to its infix notation <u>**a + b**</u>.

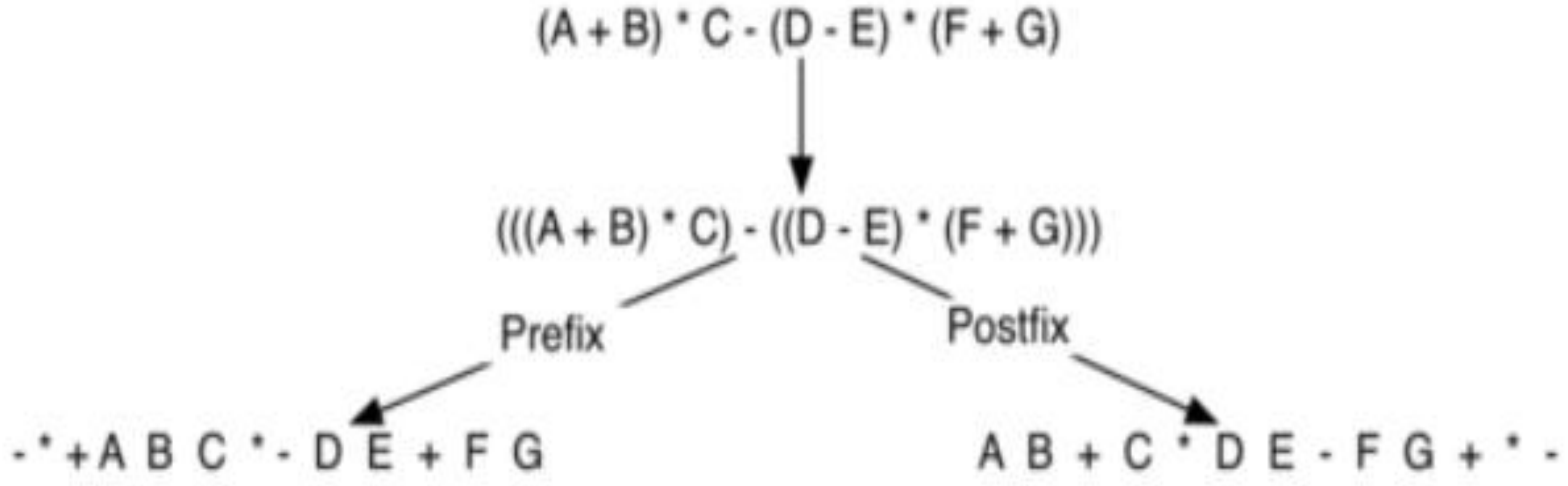
Postfix Notation

- This notation style is known as **Reversed Polish Notation**.
- In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands.
- For example, **ab+.** This is equivalent to its infix notation **a + b.**

Infix to Prefix and Postfix Notation

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Infix to Prefix and Postfix Notation


(A + B) * C - (D - E) * (F + G)
(((A + B) * C) - ((D - E) * (F + G)))
Prefix
Postfix
- * + A B C * - D E + F G
A B + C * D E - F G + * -

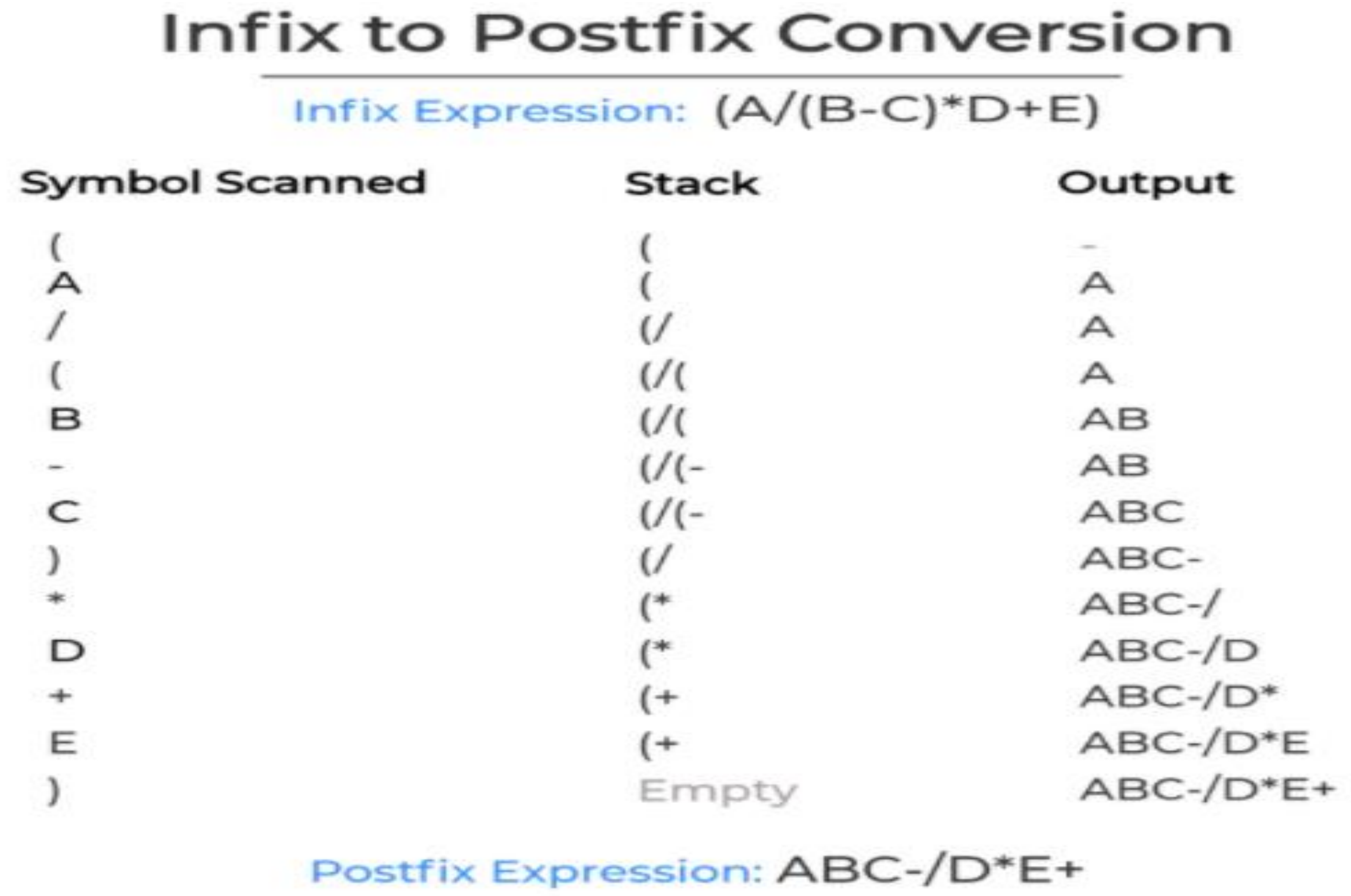
Infix to Prefix and Postfix Notation: Examples

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D		
(A + B) * (C + D)		
A * B + C * D		
A + B + C + D		

Infix to Prefix and Postfix Notation: Examples

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

Infix to Postfix using Stack



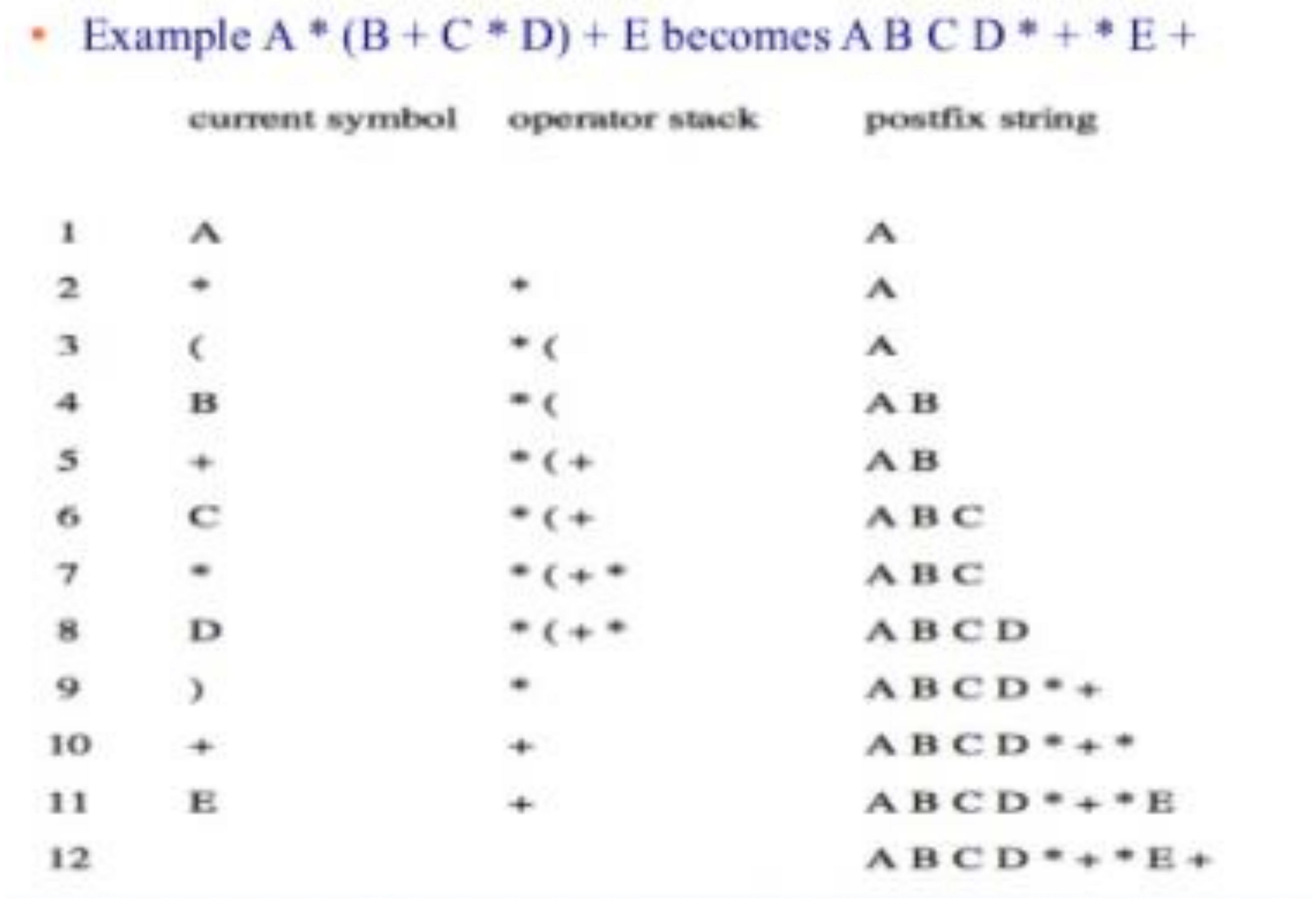
Infix to Postfix Conversion

Infix Expression: (A/(B-C)*D+E)

Symbol Scanned	Stack	Output
(	(	-
A	(	A
/	(/	A
(	(/(	A
B	(/(	AB
-	(/(-	AB
C	(/(-	ABC
)	(/	ABC-
*	(*	ABC-/
D	(*	ABC-/D
+	(+	ABC-/D*
E	(+	ABC-/D*E
)	Empty	ABC-/D*E+

Postfix Expression: ABC-/D*E+

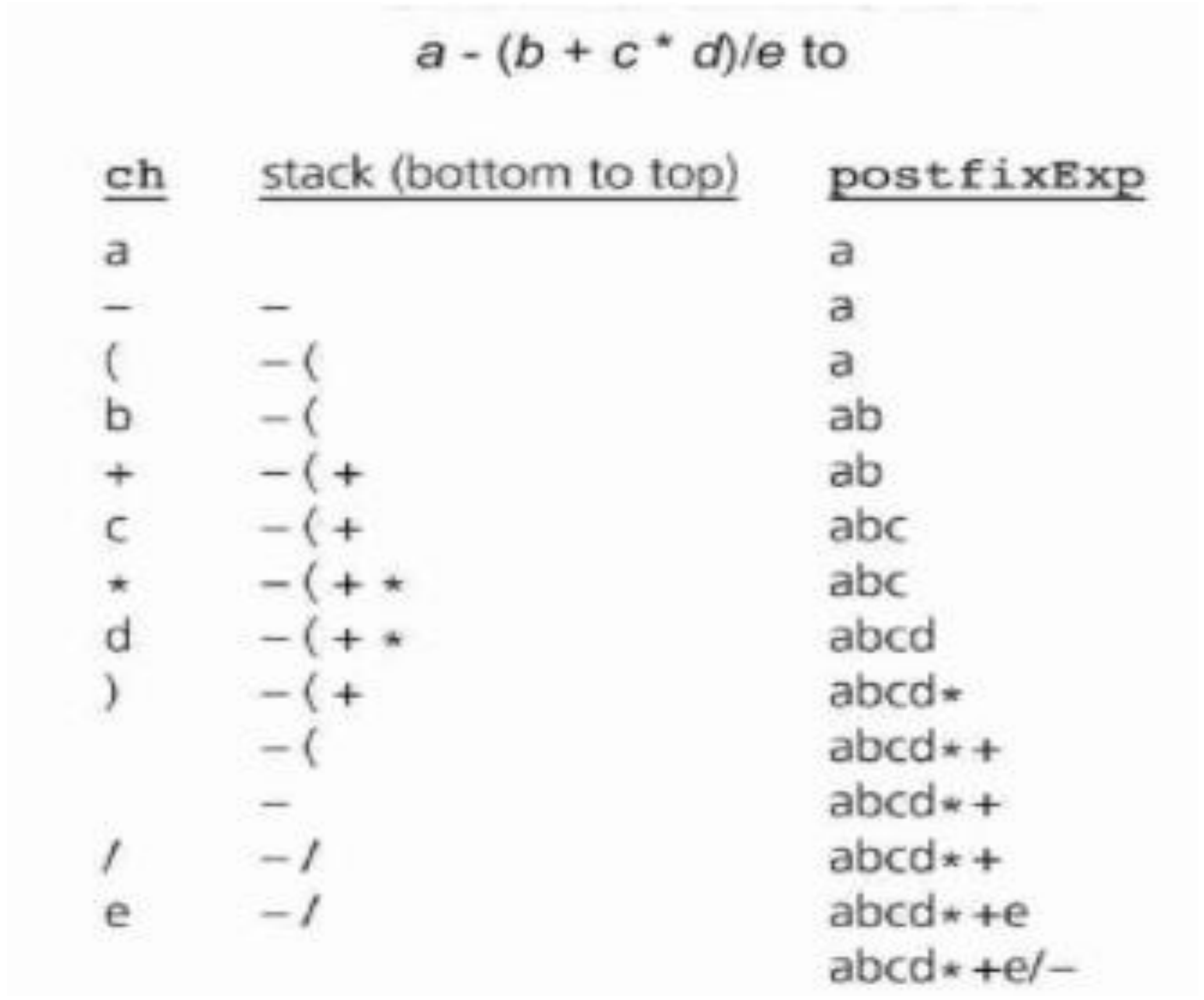
Infix to Postfix using Stack



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

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Infix to Postfix using Stack

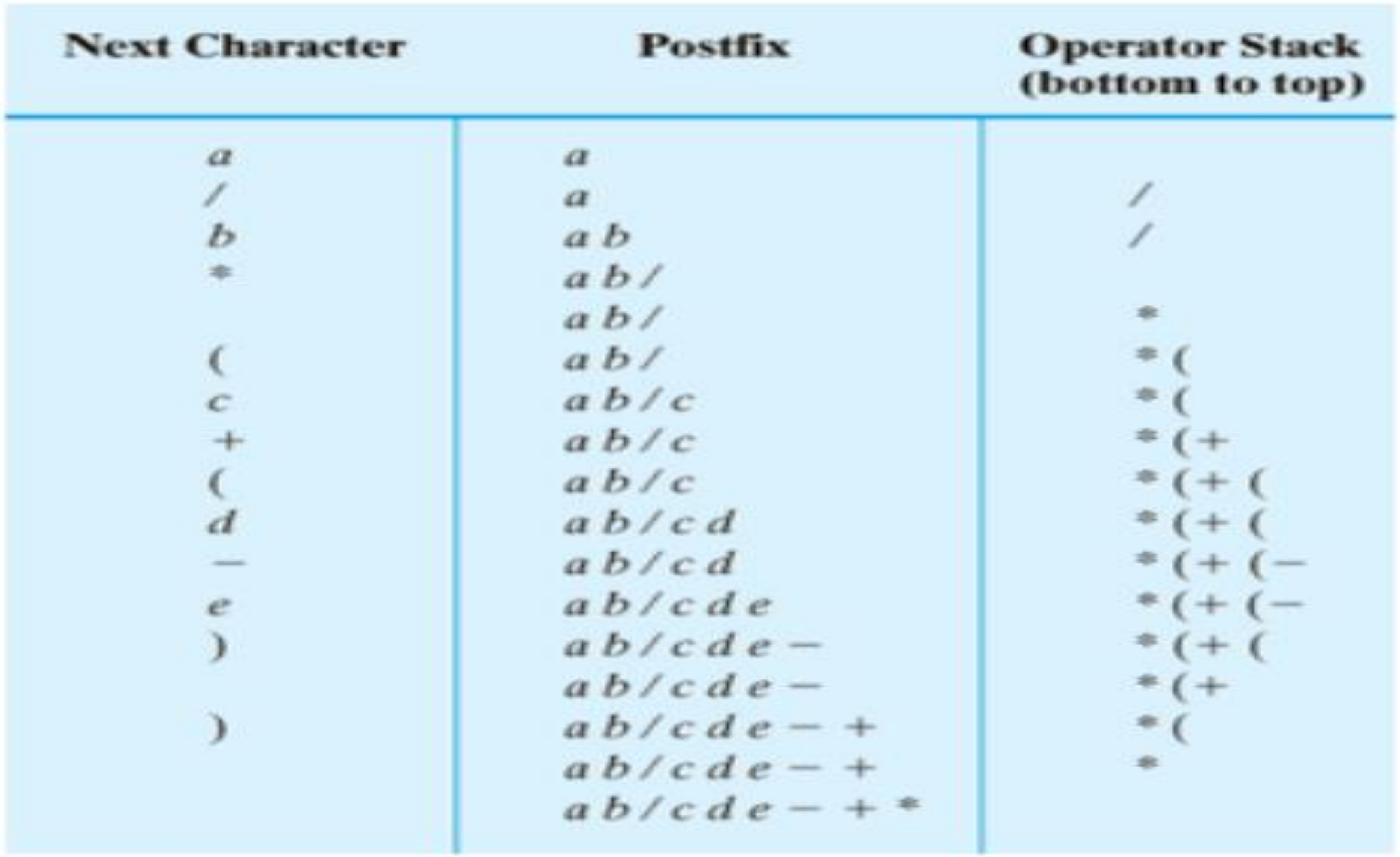


a - (*b* + *c* * *d*)/*e* to

ch	stack (bottom to top)	postfixExp
a		a
–	–	a
(	– (	a
b	– (	ab
+	– (+	ab
c	– (+	abc
*	– (+ *	abc
d	– (+ *	abcd
)	– (+	abcd*
	– (	abcd*+
	–	abcd*+
/	– /	abcd*+
e	– /	abcd*+e
		abcd*+e/–

Infix to Postfix using Stack

Expression: a/b*(c+(d-e))



Next Character	Postfix	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (
c	*a b* / *c*	* (
+	*a b* / *c*	* (+
(	*a b* / *c*	* (+ (
d	*a b* / *c d*	* (+ (
−	*a b* / *c d*	* (+ (−
e	*a b* / *c d e*	* (+ (−
)	*a b* / *c d e* −	* (+ (
	a b / *c d e* −	* (+
)	*a b* / *c d e* − +	* (
	a b / *c d e* − +	*
	a b / *c d e* − + *	

Infix to Postfix using Stack

Examples:

1) A*(B+C)/D-G
2) (A+B^C)*D+E^5
3) (A+B)*D+E/(F+A*D)+C
4) (A+B*C/D-E+F/G/(H+I))

Infix to Postfix using Stack

Examples:

1) A*(B+C)/D-G - ABC+*D/G-
2) (A+B^C)*D+E^5 - ABC^+D*E5^+
3) (A+B)*D+E/(F+A*D)+C - AB+D*EFAD*+/+C+
4) (A+B*C/D-E+F/G/(H+I)) - ABC*D/+E-FG/HI+/+

Steps for infix to prefix conversion using stack

1. Search the Infix string from **right to left**.

2. Initialize a vacant stack.

3. If the scanned character is an operand add it to the Prefix string.

4. When the scanned character is an operator and if the stack is empty push the character to stack.

5. If a scanned character is an Operator and the stack is not empty, compare the precedence of the character with element on top of the stack.

6. When top Stack has higher precedence over the scanned character pop the stack else push the scanned character to stack. Repeat this procedure till the stack is not empty and top Stack has precedence over the character.

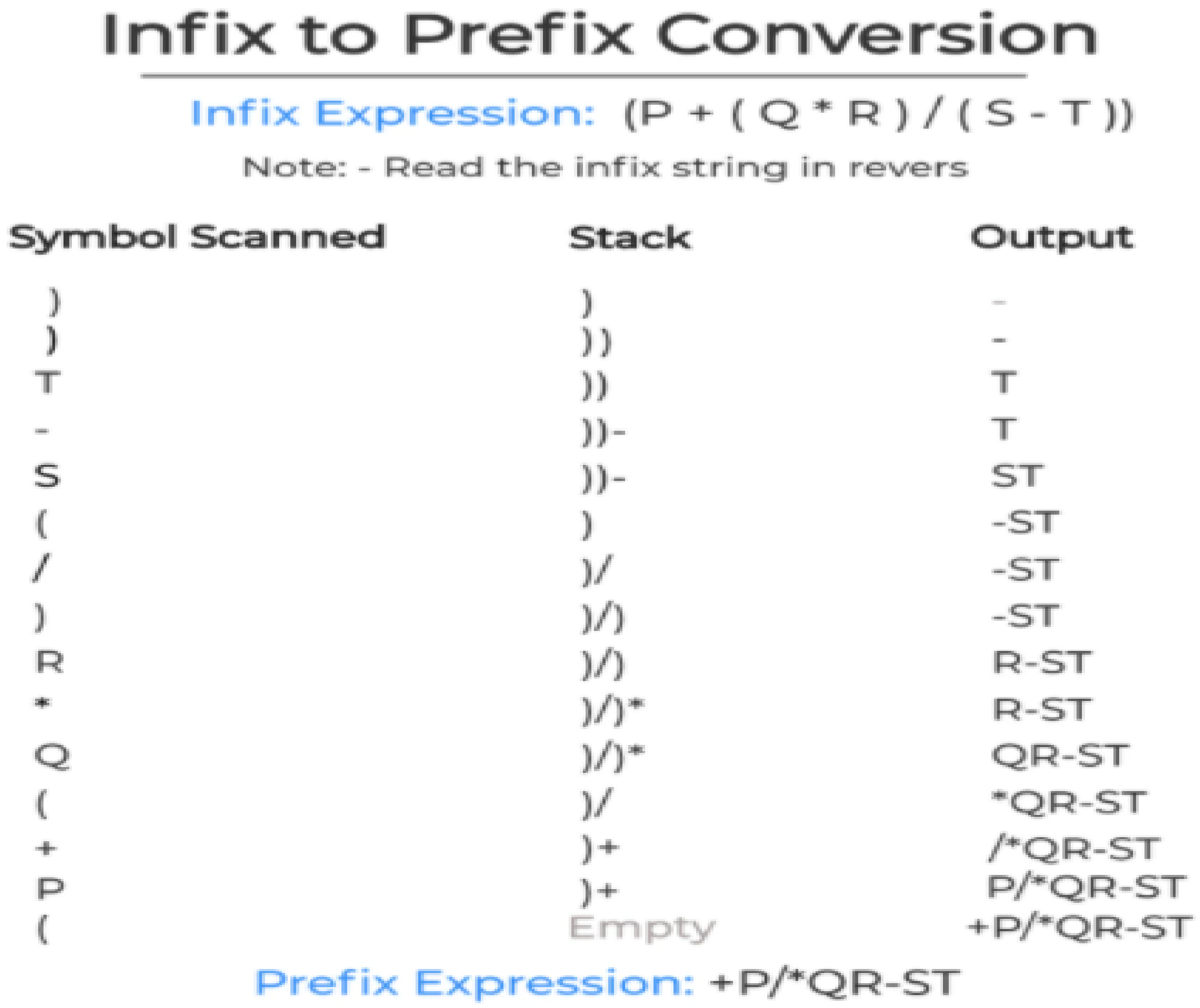
7. Repeat the 4 and 5 steps till all characters are scanned.

8. After all characters are scanned, we have to add any character that the stack may have to the Prefix string.

9. When stack is not empty add top Stack to Prefix string and Pop the stack.

10. Repeat the process as long as stack is not vacant.

Infix to Prefix using Stack : Method 1



Infix to Prefix Conversion

Infix Expression: (P + (Q * R) / (S - T))

Note: - Read the infix string in revers

Symbol Scanned	Stack	Output
)	)	-
)	))	-
T	))	T
-	))-	T
S	))-	ST
(	)	-ST
/	)/	-ST
)	)/)	-ST
R	)/)	R-ST
*	)/)*	R-ST
Q	)/)*	QR-ST
(	)/	*QR-ST
+	)+	/*QR-ST
P	)+	P/*QR-ST
(	Empty	+P/*QR-ST

Prefix Expression: +P/*QR-ST

Infix to Prefix using Stack : Method 2

Infix: (P + (Q * R) / (S – T))

Reverse the String :))T – S(/)R * Q(+ P((Convert (to) and) to ( )

Infix String : ((T – S) / (R * Q) + P)

Symbol	Stack	Prefix Expression
(	(	-
(	((	-
T	((	T
-	((-	T
S	((-	TS
)	(	TS-
/	(/	TS-
(	(/(	TS-
R	(/(	TS-R
*	(/(*	TS-R
Q	(/(*	TS-RQ

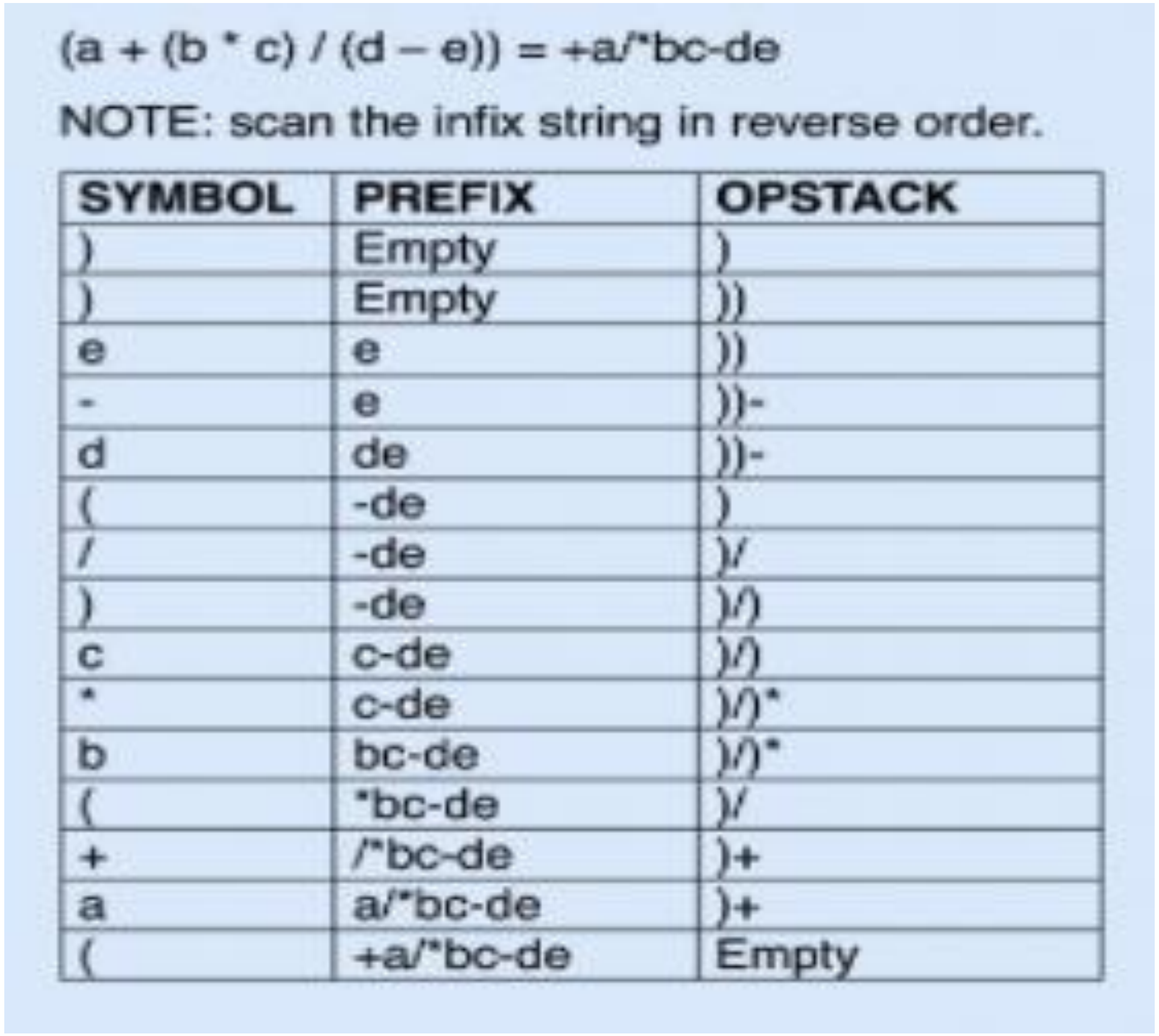
Infix to Prefix using Stack : Method 2

Infix: (P + (Q * R) / (S – T))
Reverse the String : ((T – S) / (R * Q) + P)

Symbol	Stack	Prefix Expression
)	(/	TS-RQ*
+	(+	TS-RQ*/
P	(+	TS-RQ*/P
)		TS-RQ*/P+

Answer: +P/*QR-ST

Infix to Prefix using Stack



(a + (b * c) / (d – e)) = +a/*bc-de

NOTE: scan the infix string in reverse order.

SYMBOL	PREFIX	OPSTACK
)	Empty	)
)	Empty	))
e	e	))
-	e	))-
d	de	))-
(	-de	)
/	-de	)/
)	-de	)/)
c	c-de	)/)
*	c-de	)/)*
b	bc-de	)/)*
(	*bc-de	)/
+	/*bc-de	)+
a	a/*bc-de	)+
(	+a/*bc-de	Empty

Infix to Prefix using Stack

Examples:

1) (A-(B/C))*((D*E)-F)
2) (a+b^c^d)*(e+f/d)
3) (A+B^C)*D+E^5

Infix to Prefix using Stack

Examples:

1) (A-(B/C))*((D*E)-F) - *-A/BC-*DEF
2) (a+b^c^d)*(e+f/d) - *+a^b^cd+e/fd
3) (A+B^C)*D+E^5 - +*+A^BCD^E5

Balanced Parenthesis

- The parenthesis is represented by the brackets shown below:
- () or {} or []

Where, (or { or [→ Opening bracket

) or } or] → Closing bracket

- These parentheses are used to represent the mathematical representation.
- **Balanced parenthesis** : When the opening parenthesis is equal to the closing parenthesis, then it is a balanced parenthesis.

Balanced Parenthesis

Algorithm to check balanced parenthesis:

The variable is used to determine the balance factor. Let's consider the variable 'x'. The algorithm to check the balanced parenthesis is given below:

Step 1: Set x equal to 0.

Step 2: Scan the expression from left to right.

For each opening bracket "(", increment x by 1.

For each closing bracket ")", decrement x by 1.

This step will continue scanning until x<0.

Step 3: If x is equal to 0, then

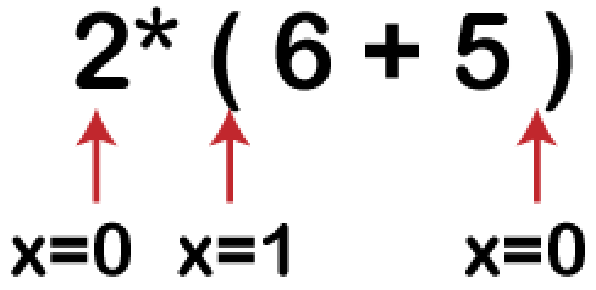
"Expression is balanced."

Else

"Expression is unbalanced."

Balanced Parenthesis

Suppose expression is 2 * (6 + 5)



Solution: First, the x variable is initialized by 0. The scanning starts from the variable '2', when it encounters '(' then the 'x' variable gets incremented by 1 and when the x reaches to the last symbol of the expression, i.e., ')' then the 'x' variable gets decremented by 1 and it's final value becomes 0. The above algorithm that if x is equal to 0 means the expression is balanced; therefore, the above expression is a balanced expression.

Balanced Parenthesis

- **Example 1: (2+5) * 4**
- **Example 2: 2 * ((4/2) + 5)**
- **Example 3: 2 * ((10/2) + 7**

Balanced Parenthesis

C program to check the balanced parenthesis.

```
#include<stdio.h>
int main()
{
    char expression[50];
    int x=0, i=0;
    printf("\nEnter an expression");
    scanf("%s", expression);
    while(expression[i]!= '\0')
    {
 if(expression[i]=='(' || expression[i]=='{')
        {
           x++;
        }
        else if(expression[i]==')' || expression[i]=='}')
        {
           x--;
           if(x<0)
           break;
        }
    i++;
}
if(x==0)
    {
        printf("Expression is balanced");
    }

    else
    {
        printf("Expression is unbalanced");
    }
    return 0;
}
```