Expression Conversion and Evaluation

An Application of Stack

Prof. Siddharth Shah

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 can be defined as a collection of operators and operands that represents a specific value.
  - In above definition, **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.

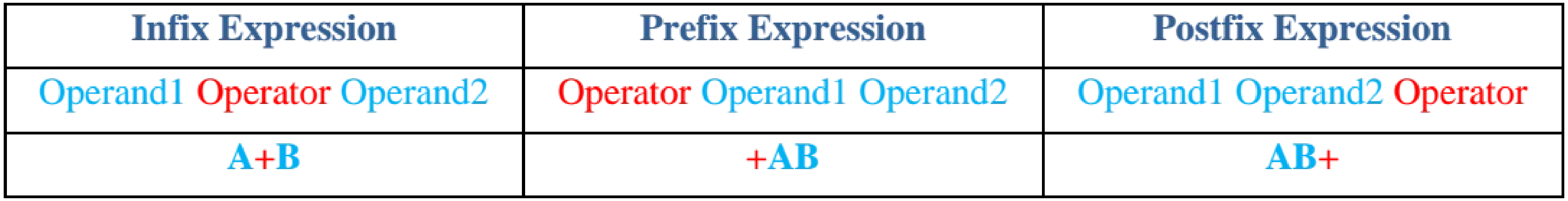
Expression Types

- Based on the operator position, expressions are divided into THREE types. They are as follows…

    1. Infix Expression

    2. Prefix Expression (Polish Notation)

    3. Postfix Expression (Reverse Polish Notation)



Infix Expression	Prefix Expression	Postfix Expression
Operand1 Operator Operand2	Operator Operand1 Operand2	Operand1 Operand2 Operator
A+B	**+AB**	**AB+**

- Any expression can be represented using the above three different types of expressions.

- Any expression can be converted from one form to another form like Infix to Postfix, Infix to Prefix, Prefix to Postfix and vice versa.

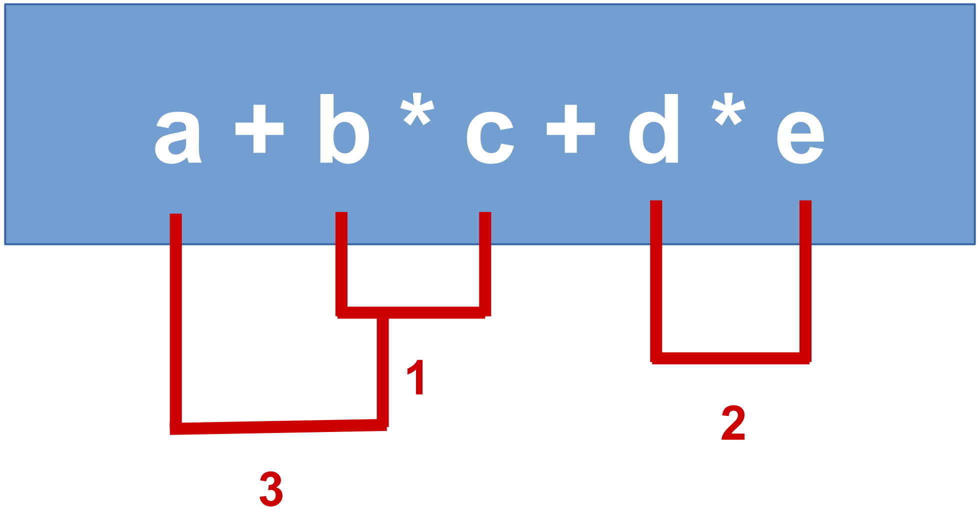
Evaluating Infix Expression

$$a + \underbrace{b * c}_{1} + d * e$$

Evaluating Infix Expression

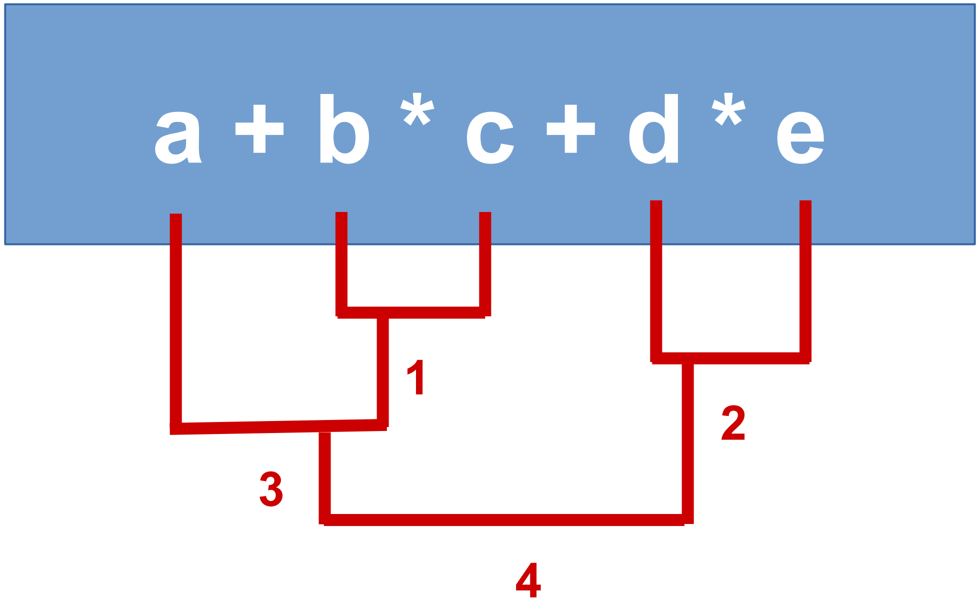
$$a + \underbrace{b * c}_{1} + \underbrace{d * e}_{2}$$

Evaluating Infix Expression



a + b * c + d * e

Evaluating Infix Expression

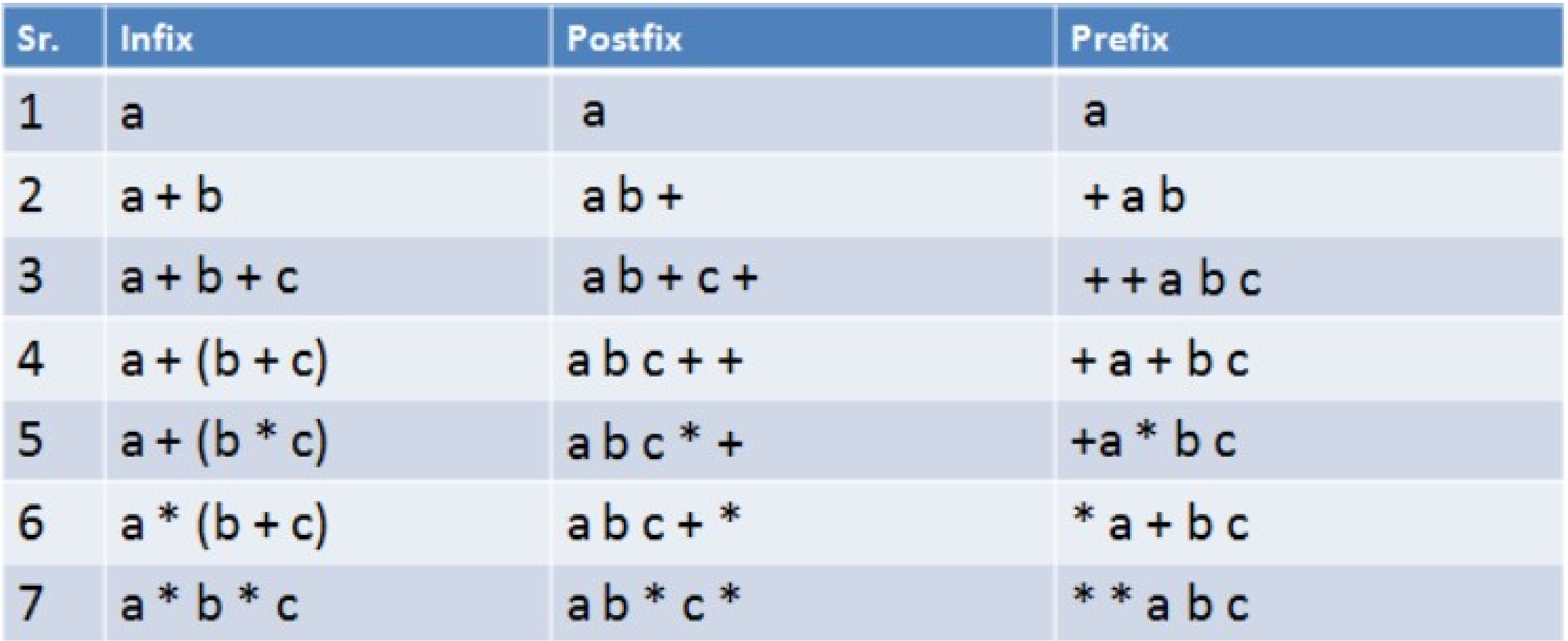


a + b * c + d * e

Evaluating Infix Expression

- A repeated scanning from left to right is needed as operators appear inside the operands.
- This repetition can be avoided if the infix expression is first converted to parenthesis free prefix or postfix (suffix) expression.
- In both prefix and postfix equivalents of an infix expression, the variables are in the same relative position.
- The expressions in postfix or prefix form are parenthesis free and operators are rearranged according to rules of precedence for operators

Some Examples



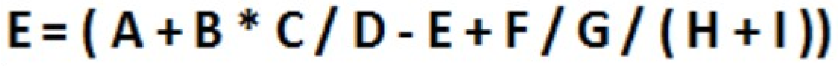
Sr.	Infix	Postfix	Prefix
1	a	a	a
2	a + b	a b +	+ a b
3	a + b + c	a b + c +	+ + a b c
4	a + (b + c)	a b c + +	+ a + b c
5	a + (b * c)	a b c * +	+a * b c
6	a * (b + c)	a b c + *	* a + b c
7	a * b * c	a b * c *	* * a b c

Postfix Expression

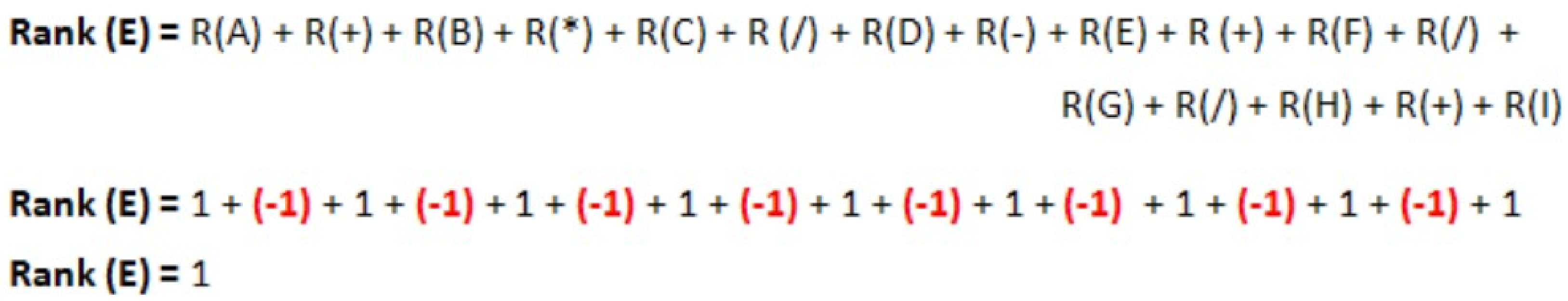
- $S = \{S_1, S_2, \ldots, S_q\}$ = set of symbols, typically variable names and literals
- $O = \{O_1, O_2, \ldots, O_m\}$ = set of operators for constructing expressions using elements of S
- Degree of an operator is the number of operands which that operator has. For example, **binary operator** which has two operands has degree 2.
- A postfix expression is defined by the following:
  1. A single symbol S_i is an expression.
  2. If $X_1, X_2, \ldots, X_n$ are expressions and O_i is of degree n, then $X_1\ X_2 \ldots X_n\ O_i$ is an expression.
  3. The only valid expressions are those obtained by steps 1 and 2.

Valid Expression

- To determine whether an expression is valid, a rank is associated with each expression, which is defined as follows:
  1. The rank of a symbol S_i is “one”.
  2. The rank of an operator O_j is 1-n, where n is the degree of O_j .
  3. The rank of an arbitrary sequence of symbols and operators is the sum of the ranks of the individual symbols and operators.
- **Example:**



E = (A + B * C / D - E + F / G / (H + I))



Rank (E) = R(A) + R(+) + R(B) + R(*) + R(C) + R (/) + R(D) + R(-) + R(E) + R (+) + R(F) + R(/) + R(G) + R(/) + R(H) + R(+) + R(I)

Rank (E) = 1 + (-1) + 1 + (-1) + 1 + (-1) + 1 + (-1) + 1 + (-1) + 1 + (-1) + 1 + (-1) + 1 + (-1) + 1

Rank (E) = 1

- Any expression is valid if the rank of that expression is 1.
- If z = x O y is a string, then x is a head of z. Finally x is a proper head if y is not empty.

Infix to Postfix Conversion

Algorithm: UNPARENTHESIZED_SUFFIX

- Given an input string **INFIX** representing an infix expression whose single character symbols have precedence values and ranks as given in following table.
- A vector **S** representing a stack, and a string function **NEXTCHAR** which, when invoked, returns the next character of the input string.
- This algorithm converts the string **INFIX** to its reverse Polish string equivalent, **POLISH.**
- **RANK** contains the value of each head of the reverse Polish string.**NEXT** contains the symbol being examined and **TEMP** is a temporary variable which contains the unstacked element.
- It is assumed that the given input string is padded on the right with the special symbol '#'

Symbol	Input precedence function F	Rank function R
+, -	1	-1
*, /	2	-1
a, b, c, ….	3	1
#	0	-

Infix to Postfix Conversion

```
1. [Initialize stack]
       TOP $\leftarrow$ 1
       S[TOP] $\leftarrow$ ‘#‘
2. [Initialize output string and rank count ]
       POLISH $\leftarrow$ ‘ ’
       RANK $\leftarrow$ 0
3. [Get first input symbol]
       NEXT $\leftarrow$ NEXTCHAR (INFIX)
4. [Translate the infix expression ]
       Repeat thru step 6 while NEXT $\neq$ ‘#‘
5. [Remove symbols with greater or equal
                          precedence from stack]
       Repeat while F(NEXT) $\leq$ F(S[TOP])
           TEMP $\leftarrow$ POP (S, TOP)
           POLISH $\leftarrow$ POLISH O TEMP
           RANK $\leftarrow$ RANK + R(TEMP)
           IF RANK <1 Then
               write (‘INVALID’)
               EXIT
6. [Push current symbol onto stack and
                          obtain next input symbol]
       Call PUSH (S,TOP, NEXT)
       NEXT $\leftarrow$ NEXTCHAR (INFIX)
7. [Remove remaining elements from stack]
       Repeat while S[TOP] $\neq$ ‘#’
           TEMP $\leftarrow$ POP (S, TOP)
           POLISH $\leftarrow$ POLISH O TEMP
           RANK $\leftarrow$ RANK + R(TEMP)
           IF RANK <1 Then
               write (‘INVALID’)
               EXIT
8. [Is the expression valid]
       If RANK = 1 Then
           write (‘VALID ‘)
       Else
           write (‘INVALID ’)
```

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	**Content of Stack (rightmost symbol is top of stack)**	**Reverse-Polish Expression**	**Rank**

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	Content of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		0

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	Content of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		0
A	# A		0

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	Content of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		0
A	# A		0
+	# +	A	1

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	Content of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		0
A	# A		0
+	# +	A	1
B	# + B	A	1

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	**Content of Stack (rightmost symbol is top of stack)**	**Reverse-Polish Expression**	**Rank**
	#		0
A	# A		0
+	# +	A	1
B	# + B	A	1
*	# + *	A B	2

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	Content of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		0
A	# A		0
+	# +	A	1
B	# + B	A	1
*	# + *	A B	2
C	# + * C	A B	2

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	Content of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		0
A	# A		0
+	# +	A	1
B	# + B	A	1
*	# + *	A B	2
C	# + * C	A B	2
-	# -	A B C * +	1

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	Content of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		0
A	# A		0
+	# +	A	1
B	# + B	A	1
*	# + *	A B	2
C	# + * C	A B	2
-	# -	A B C * +	1
D	# - D	A B C * +	1

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	Content of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		0
A	# A		0
+	# +	A	1
B	# + B	A	1
*	# + *	A B	2
C	# + * C	A B	2
-	# -	A B C * +	1
D	# - D	A B C * +	1
/	# - /	A B C * + D	2

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	Content of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		0
A	# A		0
+	# +	A	1
B	# + B	A	1
*	# + *	A B	2
C	# + * C	A B	2
-	# -	A B C * +	1
D	# - D	A B C * +	1
/	# - /	A B C * + D	2
E	# - / E	A B C * + D	2

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	Content of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		0
A	# A		0
+	# +	A	1
B	# + B	A	1
*	# + *	A B	2
C	# + * C	A B	2
-	# -	A B C * +	1
D	# - D	A B C * +	1
/	# - /	A B C * + D	2
E	# - / E	A B C * + D	2
*	# - *	A B C * + D E /	2

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	Content of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		0
A	# A		0
+	# +	A	1
B	# + B	A	1
*	# + *	A B	2
C	# + * C	A B	2
-	# -	A B C * +	1
D	# - D	A B C * +	1
/	# - /	A B C * + D	2
E	# - / E	A B C * + D	2
*	# - *	A B C * + D E /	2
H	# - * H	A B C * + D E /	2

Example: Infix to Postfix Conversion

Convert infix string **A + B * C – D / E * H** to Postfix

Character Scanned	Content of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		0
A	# A		0
+	# +	A	1
B	# + B	A	1
*	# + *	A B	2
C	# + * C	A B	2
-	# -	A B C * +	1
D	# - D	A B C * +	1
/	# - /	A B C * + D	2
E	# - / E	A B C * + D	2
*	# - *	A B C * + D E /	2
H	# - * H	A B C * + D E /	2
#	#	A B C * + D E / H * -	1

Infix to Postfix Conversion

Algorithm: REVPOL

- Given an input string INFIX containing an infix expression which has been padded on the right with ‘)’ and whose symbols have precedence value given by below table.
- A vector S used as a stack and a NEXTCHAR which when invoked returns the next character of its argument.
- This algorithm converts INFIX into reverse polish and places the result in the string POLISH.
- The integer variable TOP denotes the top of the stack. Algorithm PUSH and POP are used for stack manipulation.
- The integer variable RANK accumulates the rank of expression. Finally the string variable TEMP is used for temporary storage purpose.

Symbol	Input precedence function F	Stack precedence function G	Rank function R
+, -	1	2	-1
*, /	3	4	-1
^	6	5	-1
Variables	7	8	1
(	9	0	-
)	0	-	-

Infix to Postfix Conversion - REVPOL

```
1. [Initialize stack]
       TOP ← 1
       S[TOP] ← ‘(’
2. [Initialize output string and rank count ]
       POLISH ← ‘ ‘
       RANK ← 0
3. [Get first input symbol]
       NEXT ← NEXTCHAR (INFIX)
4. [Translate the infix expression ]
       Repeat thru step 7 while NEXT ≠ ‘ ’
5. [Remove symbols with greater
                       precedence from stack]
       IF      TOP < 1        Then
               write (‘INVALID’)
               EXIT
       Repeat while G (S[TOP]) > F(NEXT)
               TEMP ← POP (S, TOP)
                 POLISH ← POLISH O TEMP
                 RANK ← RANK + R(TEMP)
                 IF      RANK <1       Then
                         write (‘INVALID’)
                         EXIT
6. [Are there matching parentheses]
       IF      G(S[TOP]) ≠ F(NEXT)       Then
               call PUSH (S,TOP, NEXT)
       Else
               call POP (S,TOP)
7. [Get next symbol]
       NEXT ← NEXTCHAR(INFIX)
8. [Is the expression valid]
       If      TOP ≠ 0 or RANK ≠ 1       Then
               write (‘INVALID ‘)
       Else
               write (‘VALID ’)
```

Example 1: Infix to Postfix Conversion - REVPOL

Convert infix string **(A + B) * C ^ D / (E - F) * G** to Postfix

- Append ')' at the end of the INFIX string
- So now INFIX string: (A + B) * C ^ D / (E - F) * G)

Input Symbol	Content of stack	Reverse polish	Rank
	(		0
(	((		0
A	((A		0
+	((+	A	1
B	((+ B	A	1
)	(	A B +	1
*	(*	A B +	1
C	(* C	A B +	1
^	(* ^	A B + C	2
D	(* ^ D	A B + C	2
/	(/	A B + C D ^ *	1
(	(/ (	A B + C D ^ *	1
E	(/ (E	A B + C D ^ *	1
-	(/ (-	A B + C D ^ * E	2
F	(/ (- F	A B + C D ^ * E	2
)	(/	A B + C D ^ * E F -	2
*	(*	A B + C D ^ * E F - /	1
G	(* G	A B + C D ^ * E F - /	1
)		A B + C D ^ * E F - / G *	1

Postfix expression is: **A B + C D ^ * E F - / G ***

Example 2: Infix to Postfix Conversion - REVPOL

Convert infix string **(A + B) * C + D / (B + A * C) + D** to Postfix

- Append ‘)’ at the end of the INFIX string
- So now INFIX string: **(A + B) * C + D / (B + A * C) + D)**

Example 2: Infix to Postfix Conversion - REVPOL

Convert infix string **(A + B) * C + D / (B + A * C) + D** to Postfix

- Append ')' at the end of the INFIX string
- So now INFIX string: **(A + B) * C + D / (B + A * C) + D)**

Input Symbol	Content of stack	Reverse polish	Rank
	(		0
(	((		0
A	((A		0
+	((+	A	1
B	((+ B	A	1
)	(	A B +	1
*	(*	A B +	1
C	(* C	A B +	1
+	(+	A B + C *	1
D	(+ D	A B + C *	1
/	(+ /	A B + C * D	2
(	(+ / (	A B + C * D	2
B	(+ / (B	A B + C * D	2
+	(+ / (+	A B + C * D B	3
A	(+ / (+ A	A B + C * D B	3
*	(+ / (+ *	A B + C * D B A	4
C	(+ / (+ * C	A B + C * D B A	4
)	(+ /	A B + C * D B A C * +	3
+	(+	A B + C * D B A C * + / +	1
D	(+ D	A B + C * D B A C * + / +	1
)		A B + C * D B A C * + / + D +	1

Postfix expression is: **A B + C * D B A C * + / + D +**

Infix to Prefix Conversion

1. Reverse infix expression
2. Convert ‘(’ to ‘)’ and ‘)’ to ‘(’ and append extra ‘)’ at last
3. Now convert this string to postfix using REVPOL algorithm
4. Reverse the postfix expression

Example: Infix to Prefix Conversion

Convert the string **A - B / ((C * D) ^ E)** into Prefix Expression

1. Reverse infix expression:

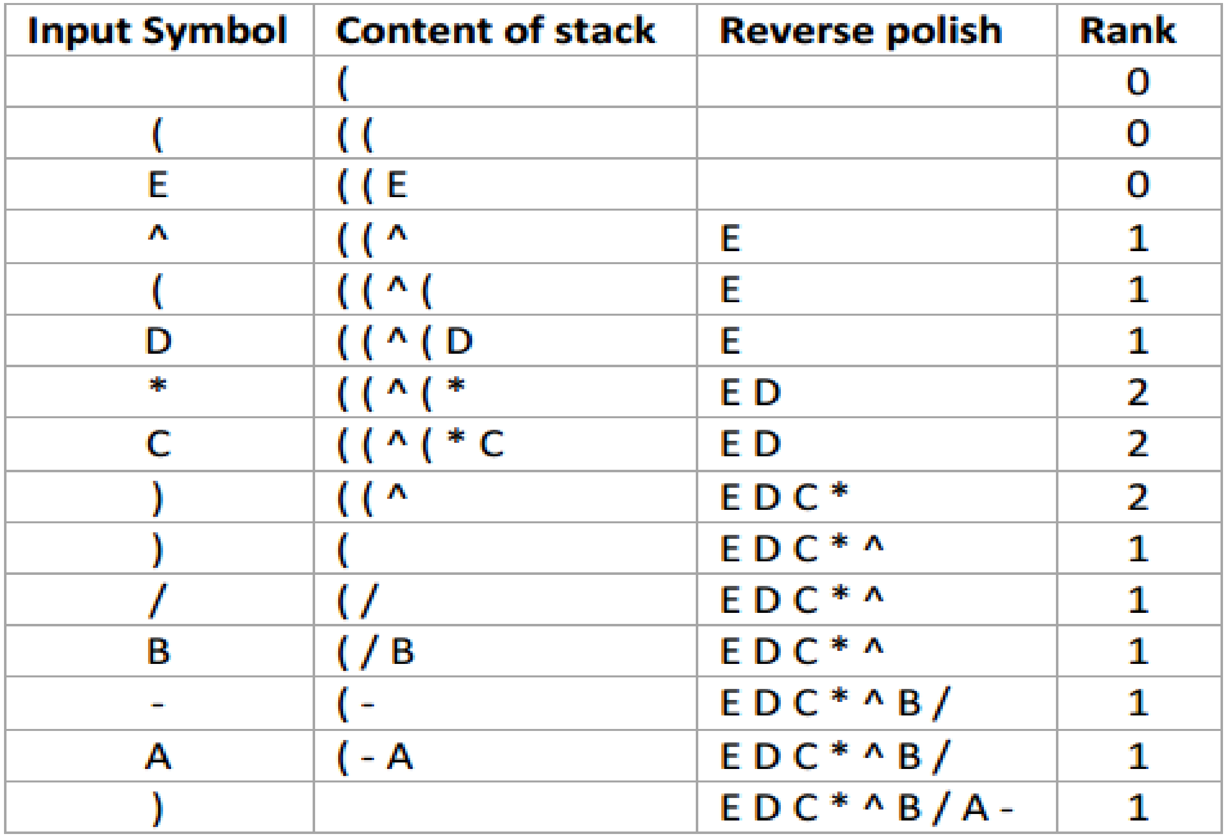
) E ^) D * C ((/ B – A

2. Convert ‘(’ to ‘)’ and ‘)’ to ‘(’ and append extra ‘)’ at last

   (E ^ (D * C)) / B – A)

3. Now convert this string to postfix using **REVPOL**

Example: Infix to Prefix Conversion (Cont..)



Input Symbol	Content of stack	Reverse polish	Rank
	(		0
(	((		0
E	((E		0
^	((^	E	1
(	((^ (	E	1
D	((^ (D	E	1
*	((^ (*	E D	2
C	((^ (* C	E D	2
)	((^	E D C *	2
)	(	E D C * ^	1
/	(/	E D C * ^	1
B	(/ B	E D C * ^	1
-	(-	E D C * ^ B /	1
A	(- A	E D C * ^ B /	1
)		E D C * ^ B / A -	1

4. Reverse the POSTFIX expression

- A / B ^ * C D E

Postfix Expression Evaluation

Algorithm: EVALUATE_POSTFIX

- Given an input string POSTFIX representing postfix expression, this algorithm is going to evaluate postfix expression and put the result into variable VALUE.

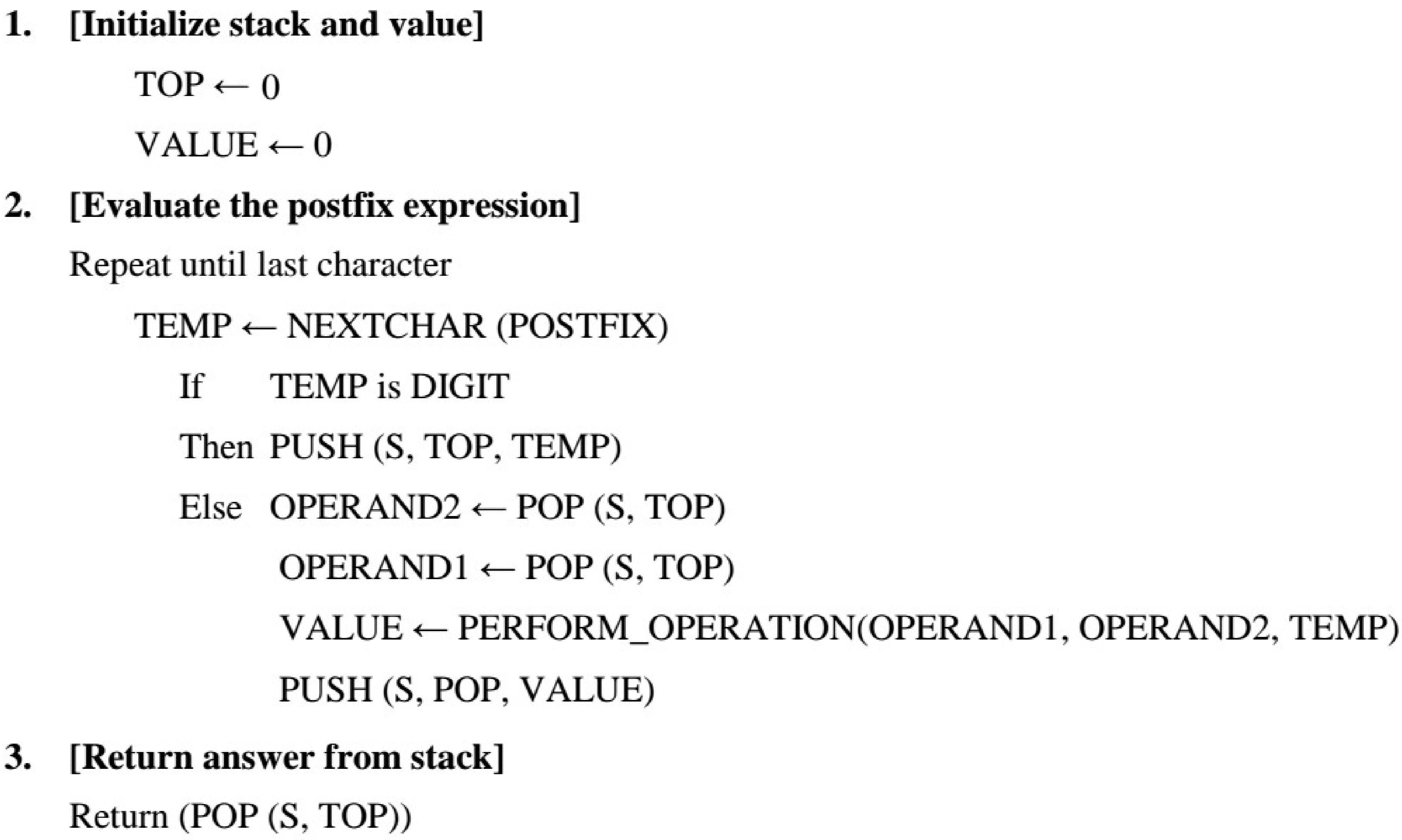
- A vector S is used as a stack PUSH and POP are the function used for manipulation of stack.

- Operand2 and operand1 are temporary variables; TEMP is used as temporary variable to store the next character from the input string given by NEXTCHAR function which when invoked returns the next character.

- PERFORM_OPERATION is a function which performs required operation on OPERAND1 and OPERAND2.

Postfix Expression Evaluation

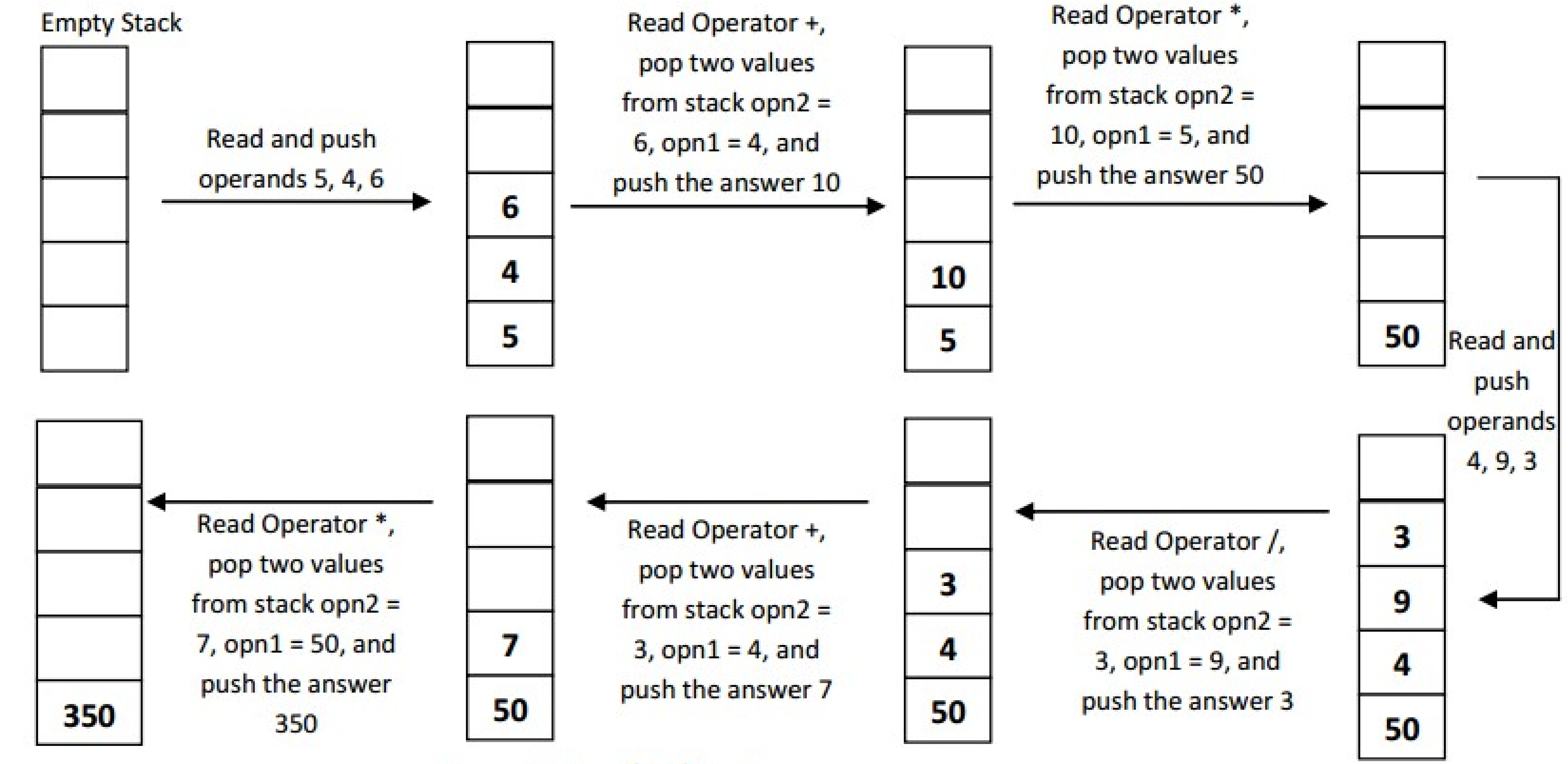
Algorithm: EVALUATE_POSTFIX



```
1.  [Initialize stack and value]
        TOP ← 0
        VALUE ← 0
2.  [Evaluate the postfix expression]
    Repeat until last character
        TEMP ← NEXTCHAR (POSTFIX)
            If     TEMP is DIGIT
            Then   PUSH (S, TOP, TEMP)
            Else   OPERAND2 ← POP (S, TOP)
                   OPERAND1 ← POP (S, TOP)
                   VALUE ← PERFORM_OPERATION(OPERAND1, OPERAND2, TEMP)
                   PUSH (S, POP, VALUE)
3.  [Return answer from stack]
    Return (POP (S, TOP))
```

Example: Postfix Expression Evaluation

*Evaluate (i): 5 4 6 + * 4 9 3 / + **



Empty Stack

Read and push operands 5, 4, 6

Read Operator +, pop two values from stack opn2 = 6, opn1 = 4, and push the answer 10

Read Operator *, pop two values from stack opn2 = 10, opn1 = 5, and push the answer 50

Read and push operands 4, 9, 3

Read Operator /, pop two values from stack opn2 = 3, opn1 = 9, and push the answer 3

Read Operator +, pop two values from stack opn2 = 3, opn1 = 4, and push the answer 7

Read Operator *, pop two values from stack opn2 = 7, opn1 = 50, and push the answer 350

Poped value **350** is the answer