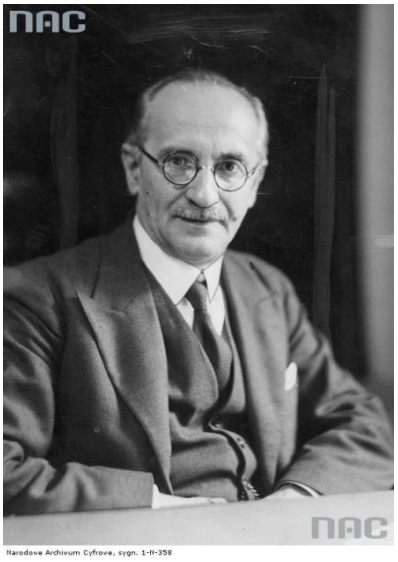


Figure 9.2 Jan Łukasiewicz

Now let us summarize the benefits of the *reverse-Polish* notation from the perspective of computing

- no ambiguity and no brackets are required
- this is the same process used by a computer to perform computations:
- operands must be loaded into registers before operations can be performed on them.
- reverse-Polish can be processed using stacks

Reverse-Polish notation is used with some programming languages, examples include postscript, pdf, and HP calculators. This processing is similar to the thought process required for writing assembly language code, where you cannot perform an operation until you have all of the operands loaded into registers.

9.2.2 Algebraic Expression Operations using Stacks

When the ADT stack is used to solve a problem, the use of the ADT's operations should not depend on its implementation. In this section we discuss the following functions that use stacks namely that conversion of an infix expression to postfix form and evaluation of the postfix expression. Now let us look at each of these functions in detail.

9.2.2.1 Conversion of Infix to Postfix

Here are some facts to be considered when converting from infix expression to postfix expression. Operands always stay in the same order with respect to one another. An operator will move only "to the right" with respect to the operands. All parentheses are removed. In other words when we analyze the conversion process we discover that

- *Operands* are in same order in infix and postfix
- *Operators* occur later in the case of postfix

The basic strategy is to send operands straight to output, output higher precedence operators first before outputting lesser precedence operators. If operators have same precedence, send them to the output in left to right order. The stack is used to hold pending operators that are yet to be given as output.

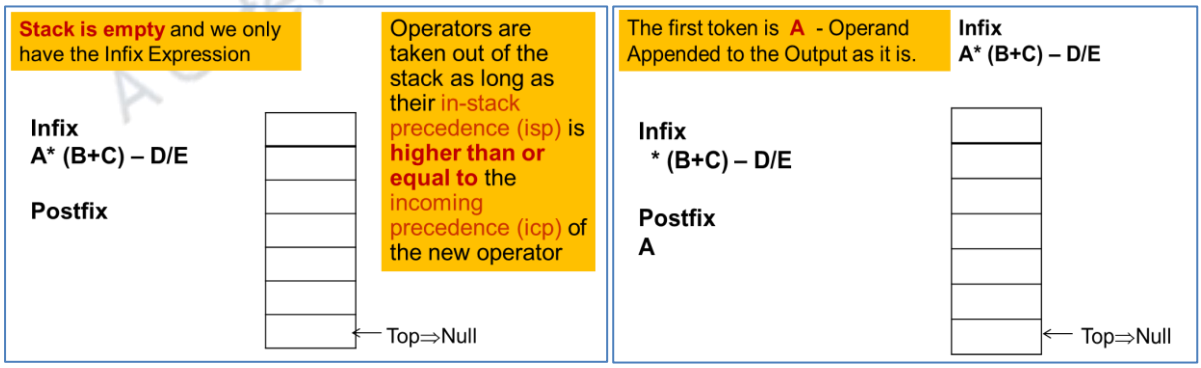
Before we discuss the steps in detail let us give the table (Table 9.1) for the incoming priority (ICP) and in stack priority (ISP) of the different operators

Incoming priority	3	2	2	1	1	4	0	0
Symbol (operator)	^	*	/	+	-	(	)	#
Instack priority	3	2	2	1	1	0	?	0

Table 9.1 Incoming priority and Instack priority

- **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
- **Step2 -** If token is **operand, Append it** in postfix expression
- **Step3 -**If token is a **left parentheses "(", push it** in stack.
- **Step4 -**If token is an operator.
  (a) Pop all the operators which are of higher or equal precedence (In stack priority –ISP) then the incoming token (Incoming priority - ICP) and append them (in the same order) to the output Expression.
  (b) After popping out all such operators, push the new token on stack.
- **Step5 -**If **")"** right parentheses is found,
  (a) **Pop all the operators** from the Stack and append them to Output String, **till** you **encounter the Opening Parenthesis "(".**
  (b) **Pop the left parenthesis** but don't append it to the output string (Postfix notation does not have brackets).
- **Step6 -**
  (a) When all tokens of Infix expression have been scanned. **Pop all the elements from the stack** and **append** them to the Output String.
  (b) The Output string is the **Postfix Notation**.

Figure 9.3 Steps to Convert Infix and Postfix

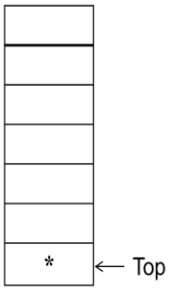


(a) (b)

Next token is * & **Stack is empty** it is **pushed into the Stack**

Infix
(B+C) – D/E

Postfix
A

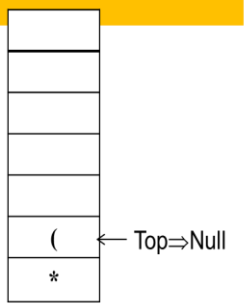


(c)

Next token is (- the incoming precedence (icp) of open-parenthesis **is maximum**.But when another operator is to come on the top of '(' **then its in-stack precedenc e(isp) is least.**

Infix
B+C) – D/E

Postfix
A

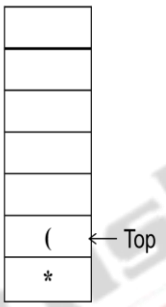


(d)

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

Infix
+C) – D/E

Postfix
AB

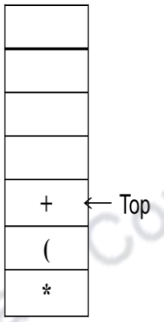


(e)

Next token, **+** is operator,. The in stack precedence **(isp)** of open parenthesis '(' **is the least** - So **+** gets **pushed into the Stack**

Infix
C) – D/E

Postfix
AB

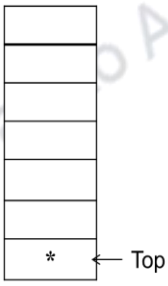


(f)

Next token), means that **pop all the elements from Stack** and **append them to the output** expression till we read an opening parenthesis - Pop out (- but not added to output)

Infix
– D/E

Postfix
ABC+



(g)

Next token, **-**, is an operator. The isp of '*' on the top of Stack –Higher than Minus. So we **pop** '*' and **append it to output** expression. Then **push - in the Stack.**

Infix
D/E

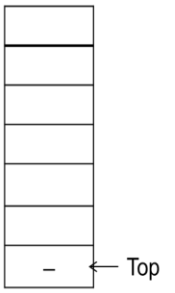
Postfix
ABC+*

(h)

Next, Operand 'D' gets **appended to the output**

Infix
/E

Postfix
ABC+*D

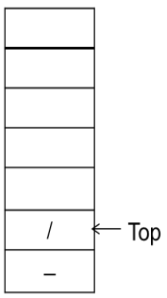


(i)

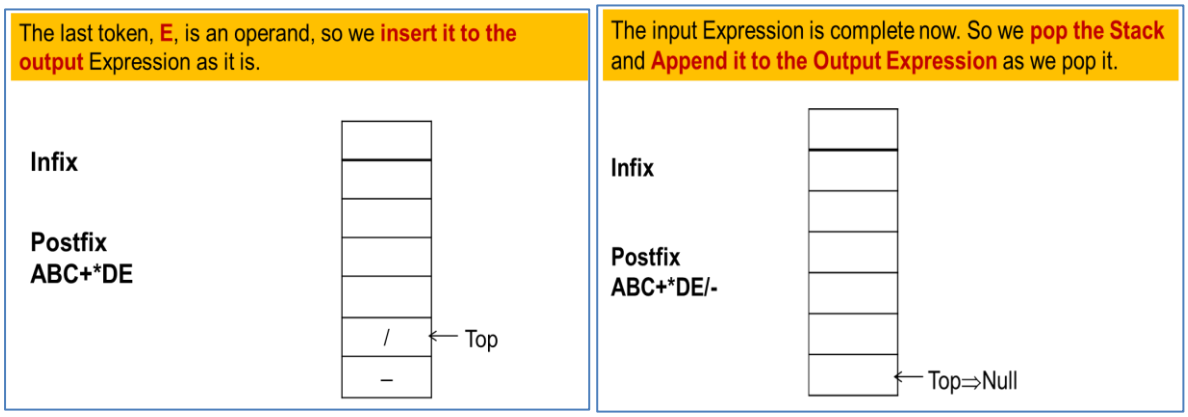
Next, we will insert the **division** operator into the Stack because its precedence (isp) is more than that of minus.

Infix
E

Postfix
ABC+*D



(j)



(k) (l)

Figure 9.4 Simulation of conversion of Infix to Postfix

The steps and simulation of conversion of infix to postfix are shown in Figure 9.3 and Figure 9.4 respectively.

1. Figure 9.4(a) shows the initial state with Stack empty and the infix expression ***A*(B+C) –D/E*** yet to be processed. Then we scan the Infix expression from left to right for tokens.
2. Figure 9.4 (b) shows the first token which is ***A*** given as output (as per step 2 of the algorithm)
3. Figure 9.4 (c) shows the pushing of the operator ***** into the stack according to step 4 (b).
4. Then the open parenthesis ***(*** is pushed onto the stack as per step 3 of the algorithm.
5. Similarly the next token the operand ***B*** is given as output and the operator **+** is pushed onto the stack as shown in Figure 9.4 (e) and 9.4 (f).
6. Next token we see is a ***)*** which means all entries in the stack up to and including open parenthesis is removed from the stack as shown in Figure 9.4 (g).
7. However neither (or) form part of the output. As per step 4 (a) ***** is popped out and – is pushed onto the stack (Figure 9.4 (h)).
8. Now the operand D is output (Figure 9.4(i)) and then the operator / is pushed into the stack (Figure 9.4 (j)).
9. Figure 9.4 (k) shows E being the next output token and
10. Finally operators left in the stack are popped out in the last in first out order (Figure 9.4 (l)).

As you can see an important part of the algorithm is the comparison of the precedence of the operators that are scanned and those that are in the stack. If the operators in the stack have higher precedence than the precedence of the incoming operator then they are outputted in reverse order to which they were pushed into the stack. Then the lower precedence operator is pushed into the stack. The other important part of the algorithm is the handling of parenthesis. When a right parenthesis is encountered, we pop all operators in the stack up to and including the matching left parenthesis. However neither the open or closing parenthesis is given as output. When this step is carried out when we say matching parenthesis we mean that the type of parenthesis should also match.

9.2.2.2 Evaluating Postfix Expression

The next application of stack that we discuss is the evaluation of postfix expression. The basic concept is as follows:

- Whenever an operand is encountered, push it onto the stack
- Whenever an operator is encountered, pop required number of arguments from operand stack and evaluate
- Push result back onto stack

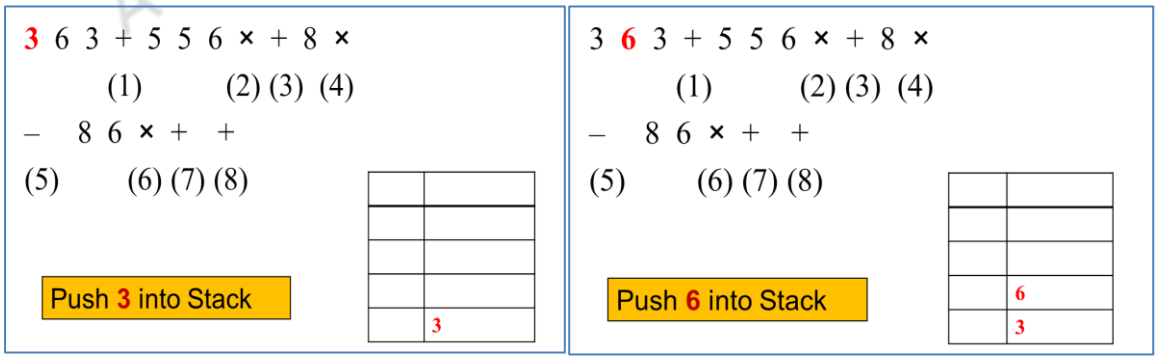
The algorithm considers two cases – namely when the operator is binary or when it is unary where the number of operands popped out of the stack depends on the type of operator. Figure 9.6 shows a running example of the use of stack for evaluating a postfix expression. In the figure we have numbered the operators in order to explain the process. Please note that the stack is a single stack. The first column shown in the Figure 9.6 is the stack. However the second column is used to show the progress of the evaluation and does not form part of the stack.

The detail steps of the algorithm are given in Figure 9.5.

Algorithm for Evaluation

1. Empty the operand stack
2. **while** there are more tokens - Get the next token
3. **if** the first character of the token is an operand
   Push it onto the stack
4. **else if** the token is a binary operator
   a. Pop the right operand off the stack
   b. Pop the left operand off the stack
   c. Evaluate the operation
   d. Push the result onto the stack
5. **else if** the token is an unary operator
   a. Pop the top operand off the stack
   b. Evaluate the operation
   c. Push the result onto the stack
6. Pop the stack and return the result

Figure 9.5 Algorithm for Evaluation of Postfix Expression



(a) **(b)**

3 6 3 + 5 5 6 × + 8 ×
(1) (2) (3) (4)
– 8 6 × + +
(5) (6) (7) (8)

Push 3 into Stack

	3
	6
	3

(c)

3 6 3 + 5 5 6 × + 8 ×
(1) (2) (3) (4)
– 8 6 × + +
(5) (6) (7) (8)

Pop 3 & 6 off the stack do the operation + (1) & push result 9 into Stack

+ (1)	9
	3

(d)

3 6 3 + 5 5 6 × + 8 ×
(1) (2) (3) (4)
– 8 6 × + +
(5) (6) (7) (8)

Push 5 into Stack

	5
+ (1)	9
	3

(e)

3 6 3 9 5 5 6 × + 8 ×
(1) (2) (3) (4)
– 8 6 × + +
(5) (6) (7) (8)

Push 5 into Stack

	5
	5
+ (1)	9
	3

(f)

3 6 3 9 5 5 6 × + 8 ×
(1) (2) (3) (4)
– 8 6 × + +
(5) (6) (7) (8)

Push 6 into Stack

	6
	5
	5
+ (1)	9
	3

(g)

3 6 3 9 5 5 6 × + 8 ×
(1) (2) (3) (4)
– 8 6 × + +
(5) (6) (7) (8)

Pop 6 & 5 off the stack do the operation × (2) & push result 30 into Stack

× (2)	30
	5
+ (1)	9
	3

(h)

3 6 3 9 5 5 6 × + 8 ×
(1) (2) (3) (4)
– 8 6 × + +
(5) (6) (7) (8)

Pop 30 & 5 off the stack do the operation + (3) & push result 35 into Stack

× (2)	
+ (3)	35
+ (1)	9
	3

(i)

3 6 3 9 5 3 6 30 35 8 ×
(1) (2) (3) (4)
– 8 6 × + +
(5) (6) (7) (8)

Push 8 into Stack

× (2)	8
+ (3)	35
+ (1)	9
	3

(j)

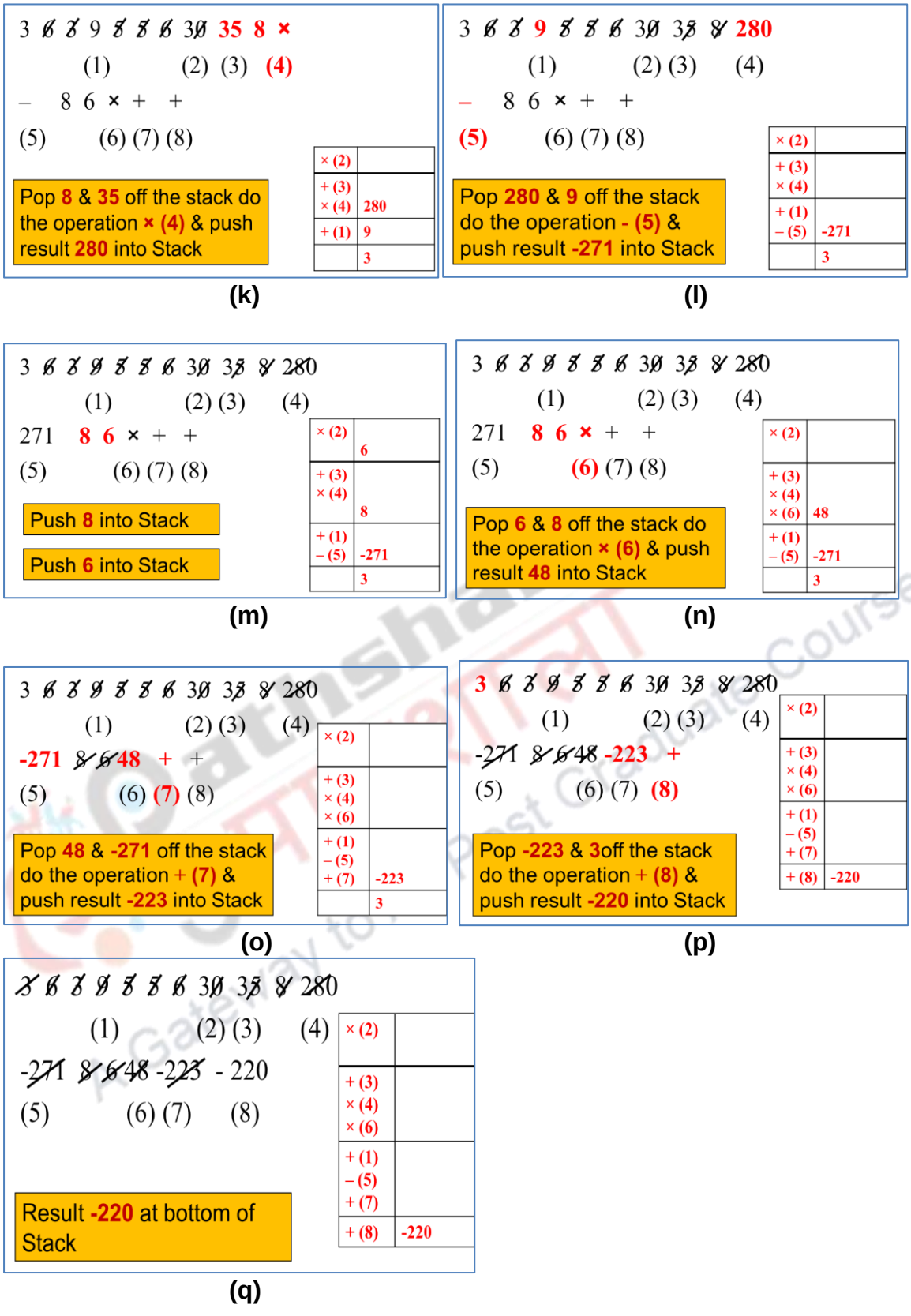


Figure 9.6 A Running Example for Evaluating Postfix Expression

Figure 9.6(a) shows the initially empty stack and the input postfix expression with 8 operators (given below) which we wish to evaluate. The expression will be read one by one, left to right until the end of the expression is reached.

3 6 3 + 5 5 6 × + 8 × – 8 6 × + +