Notations

Prepared by: Afaq Mansoor Khan

BSSE III- Group A

Session 2017-21

IMSciences, Peshawar.

Last Lecture Summary

- Introduction to Stack Data Structure
- Stack Operations
- Analysis of Stack Operations
- Applications of Stack Data Structure in Computer Science

Objectives Overview

- Notations
- Prefix, Infix and Postfix Notations
- Conversion of one type expression to another
- Evaluation of Prefix and Postfix Notations

Notation

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 Notation
- Postfix Notation

These notations are named as how they use operator in expression. The terms infix, prefix, and postfix tell us whether the operators go between, before, or after the operands.

Infix Notation

- We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Parentheses

- Evaluate 2+3*5.
- + First:
  - (2+3)*5 = 5*5 = 25
- * First:
  - 2+(3*5) = 2+15 = 17
- Infix notation requires Parentheses.

Prefix Notation

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

Parentheses

- Evaluate + 2 * 3 5
- + 2 * 3 5 =
  - = + 2 * 3 5
  - = + 2 15 = 17
- * + 2 3 5 =
  - = * + 2 3 5
  - = * 5 5 = 25

- No parentheses needed!

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**.

Parentheses

- Evaluate 2 3 5 * +
- 2 3 5 * + =
  - = 2 3 5 * +
  - = 2 15 + = 17
- 2 3 + 5 * =
  - = 2 3 + 5 *
  - = 5 5 * = 25

- No parentheses needed here either!

Fully Parenthesized Expression

- A FPE has exactly one set of Parentheses enclosing each operator and its operands.

- Which one is fully parenthesized?
- (A + B) * C
- ((A + B) * C)
- ((A + B) * (C))

Conversion from Infix to Postfix

Conversion from Infix to Postfix

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

- Postfix: A B + C * D E + F / -

- Operand order does not change!
- Operators are in order of evaluation!

Infix to Postfix - Algorithm

- Initialize a Stack for operators, output list.
- Split the input into a list of tokens.
- for each token (left to right):
    - if it is operand: append to output
    - if it is '(': push onto Stack
    - if it is ')': pop & append till '('

FPE Infix to Postfix

(((A + B) * (C - E)) / (F + G))

▲

- stack: <empty>
- output: []

FPE Infix to Postfix

((A + B) * (C - E)) / (F + G))

▲

- stack: (
- output: []

FPE Infix to Postfix

(A + B) * (C - E)) / (F + G))

▲

- stack: ((
- output: []

FPE Infix to Postfix

A + B) * (C - E)) / (F + G))

▲

- stack: (((
- output: []

FPE Infix to Postfix

+ B) * (C - E)) / (F + G))

▲

- stack: (((
- output: [A]

FPE Infix to Postfix

B) * (C - E)) / (F + G))

▲

- stack: (((+
- output: [A]

FPE Infix to Postfix

) * (C - E)) / (F + G))

▲

- stack: (((+
- output: [A B]

FPE Infix to Postfix

* (C - E)) / (F + G))

▲

- stack: ((
- output: [A B +]

FPE Infix to Postfix

(C - E)) / (F + G))

▲

- stack: ((*
- output: [A B +]

FPE Infix to Postfix

C - E)) / (F + G))

▲

- stack: ((* (
- output: [A B +]

FPE Infix to Postfix

- E)) / (F + G))

▲

- stack: ((* (
- output: [A B + C]

FPE Infix to Postfix

E)) / (F + G))

▲

- stack: ((* (-
- output: [A B + C]

FPE Infix to Postfix

)) / (F + G))

▲

- stack: ((* (-
- output: [A B + C E]

FPE Infix to Postfix

) / (F + G))

▲

- stack: ((*
- output: [A B + C E -]

FPE Infix to Postfix

/ (F + G))

▲

- stack: (
- output: [A B + C E - *]

FPE Infix to Postfix

(F + G))

▲

- stack: (/
- output: [A B + C E - *]

FPE Infix to Postfix

F + G))

▲

- stack: (/ (
- output: [A B + C E - *]

FPE Infix to Postfix

+ G))

▲

- stack: (/ (
- output: [A B + C E - * F]

FPE Infix to Postfix

G))

▲

- stack: (/ (+
- output: [A B + C E - * F]

FPE Infix to Postfix

))

▲

- stack: (/ (+
- output: [A B + C E - * F G]

FPE Infix to Postfix

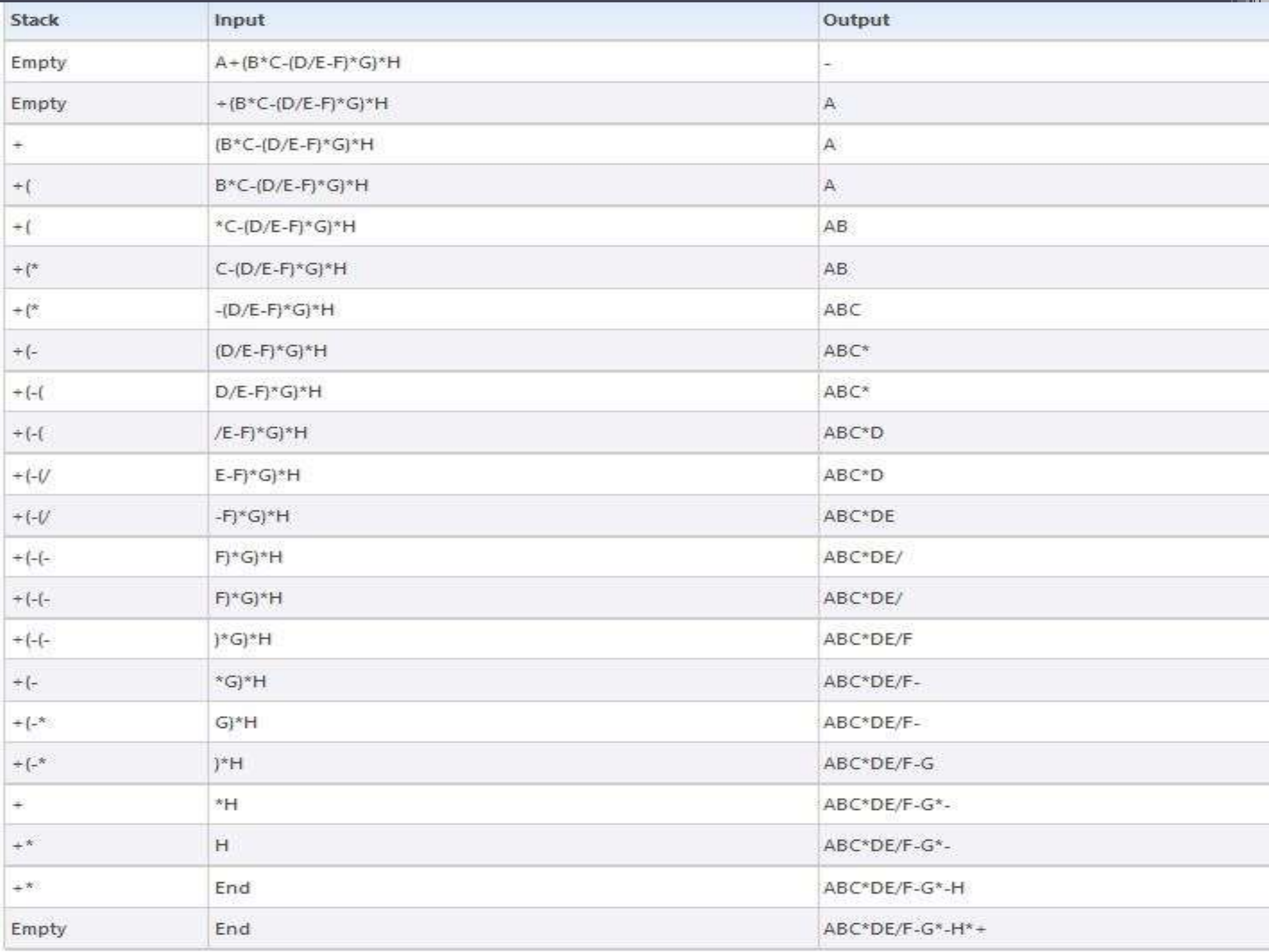
)

▲

- stack: (/
- output: [A B + C E - * F G +]

FPE Infix to Postfix

- stack: <empty>
- output: [A B + C E - * F G + /]



Stack	Input	Output
Empty	A+(B*C-(D/E-F)*G)*H	-
Empty	+(B*C-(D/E-F)*G)*H	A
+	(B*C-(D/E-F)*G)*H	A
+(	B*C-(D/E-F)*G)*H	A
+(	*C-(D/E-F)*G)*H	AB
+(*	C-(D/E-F)*G)*H	AB
+(*	-(D/E-F)*G)*H	ABC
+(-	(D/E-F)*G)*H	ABC*
+(-(	D/E-F)*G)*H	ABC*
+(-(	/E-F)*G)*H	ABC*D
+(-(/	E-F)*G)*H	ABC*D
+(-(/	-F)*G)*H	ABC*DE
+(-(-	F)*G)*H	ABC*DE/
+(-(-	F)*G)*H	ABC*DE/
+(-(-	)*G)*H	ABC*DE/F
+(-	*G)*H	ABC*DE/F-
+(-*	G)*H	ABC*DE/F-
+(-*	)*H	ABC*DE/F-G
+	*H	ABC*DE/F-G*-
+*	H	ABC*DE/F-G*-
+*	End	ABC*DE/F-G*-H
Empty	End	ABC*DE/F-G*-H*+

Infix to Prefix Conversion

Infix to Prefix - Algorithm

1. Reverse the infix expression i.e A+B*C will become C*B+A. Note while reversing each ‘(‘ will become ‘)’ and each ‘)’ becomes ‘(‘.
2. Obtain the postfix expression of the modified expression i.e CB*A+.
3. Reverse the postfix expression. Hence in our example prefix is +A*BC.

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

((A + B) * (C + D))

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

(+ A B * (C + D))

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

* + A B (C + D)

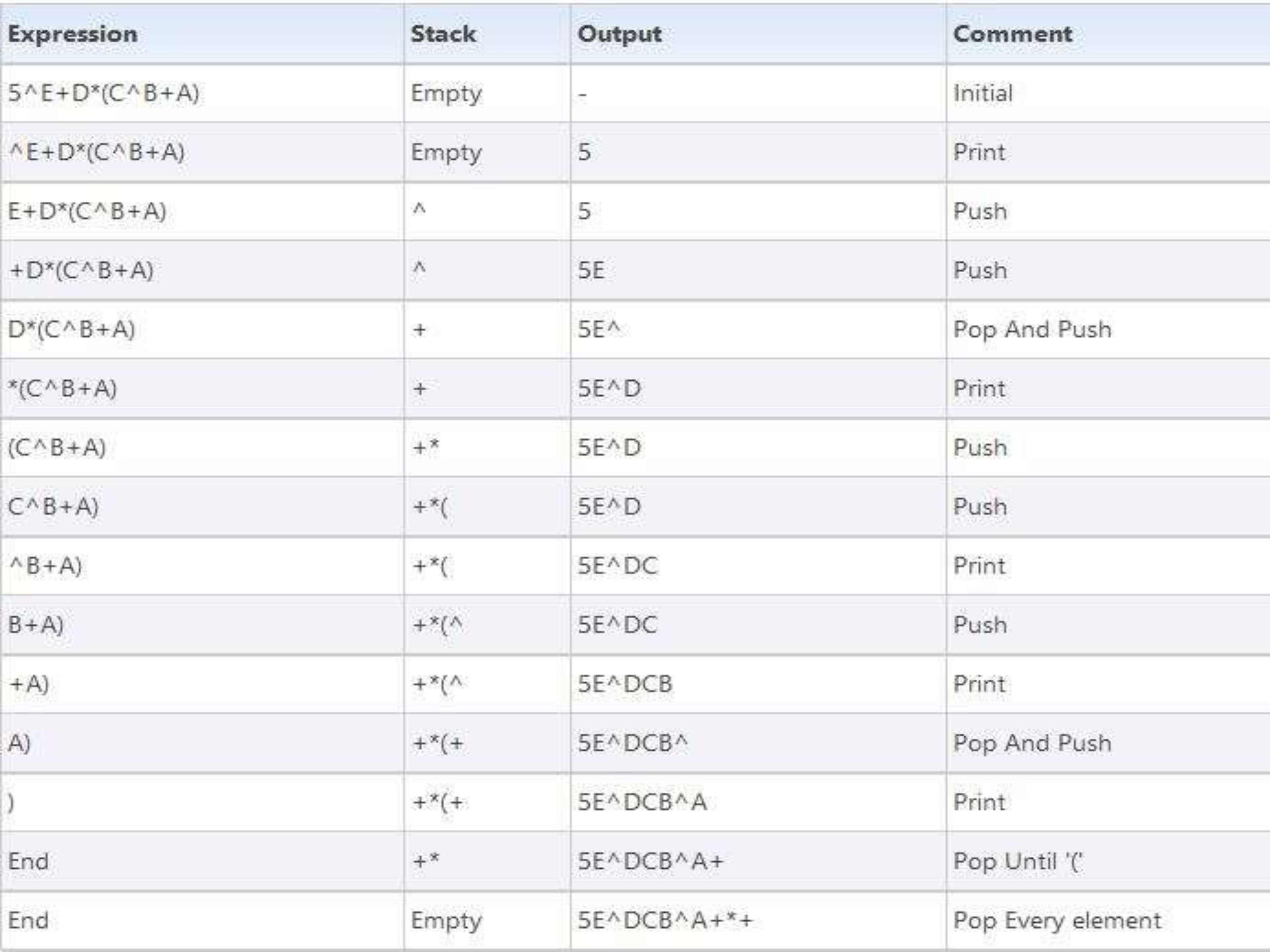
←

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

* + A B + C D

Order of operands does not change!



Expression	Stack	Output	Comment
5^E+D*(C^B+A)	Empty	-	Initial
^E+D*(C^B+A)	Empty	5	Print
E+D*(C^B+A)	^	5	Push
+D*(C^B+A)	^	5E	Push
D*(C^B+A)	+	5E^	Pop And Push
*(C^B+A)	+	5E^D	Print
(C^B+A)	+*	5E^D	Push
C^B+A)	+*(	5E^D	Push
^B+A)	+*(	5E^DC	Print
B+A)	+*(^	5E^DC	Push
+A)	+*(^	5E^DCB	Print
A)	+*(+	5E^DCB^	Pop And Push
)	+*(+	5E^DCB^A	Print
End	+*	5E^DCB^A+	Pop Until '('
End	Empty	5E^DCB^A+*+	Pop Every element

Postfix to Infix Conversion

Algorithm

- While there are input symbol left
  - Read the next symbol from input.
  - If the symbol is an operand
    - Push it onto the stack.
- Otherwise,
  - the symbol is an operator.
- If there are fewer than 2 values on the stack
  - Show Error /* input not sufficient values in the expression */
- Else
  - Pop the top 2 values from the stack.
  - Put the operator, with the values as arguments and form a string.
  - Encapsulate the resulted string with parenthesis.
  - Push the resulted string back to stack.
- If there is only one value in the stack
  - That value in the stack is the desired infix string.
- If there are more values in the stack
  - Show Error /* The user input has too many values */

Prefix to Infix Conversion

Algorithm

- The reversed input string is completely pushed into a stack.
    - prefixToInfix(stack)
- 2.IF stack is not empty
- a. Temp -->pop the stack
- b. IF temp is a operator
    - Write a opening parenthesis to output
    - prefixToInfix(stack)
    - Write temp to output
    - prefixToInfix(stack)
    - Write a closing parenthesis to output
- c. ELSE IF temp is a space -->prefixToInfix(stack)
- d. ELSE
    - Write temp to output
    - IF stack.top NOT EQUAL to space -->prefixToInfix(stack)

Prefix to Postfix Conversion

Algorithm

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack

  Create a string by concatenating the two operands and the operator after them.

  string = operand1 + operand2 + operator

  And push the resultant string back to Stack
- Repeat the above steps until end of Prefix expression.

Postfix to Prefix Conversion

Algorithm

- Read the Postfix expression from left to right
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack

  Create a string by concatenating the two operands and the operator before them.

  string = operator + operand2 + operand1

  And push the resultant string back to Stack
- Repeat the above steps until end of Prefix expression.

Postfix Evaluation

Postfix Evaluation Algorithm

- Step 1 – scan the expression from left to right
- Step 2 – if it is an operand push it to stack
- Step 3 – if it is an operator pull operand from stack and perform operation
- Step 4 – store the output of step 3, back to stack
- Step 5 – scan the expression until all operands are consumed
- Step 6 – pop the stack and perform operation

Prefix Evaluation

Prefix Evaluation Algorithm

1) Put a pointer P at the end of the end
2) If character at P is an operand push it to Stack
3) If the character at P is an operator pop two elements from the Stack. Operate on these elements according to the operator, and push the result back to the Stack
4) Decrement P by 1 and go to Step 2 as long as there are characters left to be scanned in the expression.
5) The Result is stored at the top of the Stack, return it
6) End

Summary

- Notations
- Prefix, Infix and Postfix Notations
- Conversion of one type expression to another
- Evaluation of Prefix and Postfix Notations

References

- http://scanftree.com/Data_Structure/
- https://www.geeksforgeeks.org/convert-infix-prefix-notation/
- https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.htm