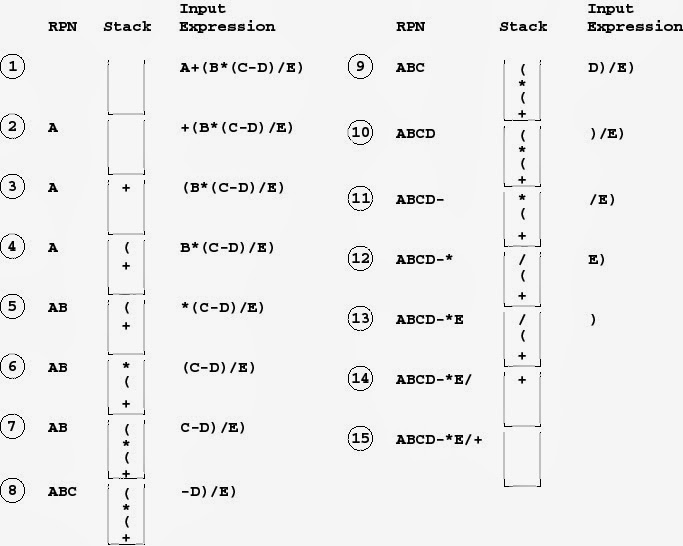
Convert Infix expression to Postfix expression using stack

Below are the steps to implement the above idea:

1. Scan the infix expression **from left to right**.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
   * If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack [or the stack is empty or the stack contains a ‘**(**‘ ], then push it in the stack. [‘**^**‘ operator is right associative and other operators like ‘**+**‘,’**–**‘,’**\***‘ and ‘**/**‘ are left-associative].
     + Check especially for a condition when the operator at the top of the stack and the scanned operator both are ‘**^**‘. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.
     + In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
   * Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
     + After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is a ‘**(**‘, push it to the stack.
5. If the scanned character is a ‘**)**’, pop the stack and output it until a ‘**(**‘ is encountered, and discard both the parenthesis.
6. Repeat steps **2-5** until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.



To convert an infix expression (e.g., A + B \* C) to a postfix expression (e.g., A B C \* +) using a stack, you can follow these rules:

### 1. **Initialize an empty stack and an empty postfix list**

* The stack is used to hold operators and parentheses.
* The postfix list will store the final postfix expression.

### 2. **Read the infix expression from left to right**

### 3. **Handle operands**

* If the character is an **operand** (e.g., a variable or number), directly append it to the postfix list.

### 4. **Handle left parentheses** (

* If the character is a **left parenthesis** (, push it onto the stack.

### 5. **Handle right parentheses** )

* If the character is a **right parenthesis** ), pop operators from the stack and append them to the postfix list until a left parenthesis ( is encountered.
* Discard the left parenthesis (.

### 6. \*Handle operators (+, -, , /, etc.)

* If the character is an **operator**:
  + While the stack is not empty and the **precedence** of the operator at the top of the stack is **greater than or equal** to the precedence of the current operator, pop the stack and append the popped operator to the postfix list.
  + Then, push the current operator onto the stack.

### 7. **Pop all remaining operators**

* After reading the entire infix expression, pop any remaining operators from the stack and append them to the postfix list.

### 8. **Precedence and Associativity**

* **Precedence** determines the order of operations:
  + \*, / have higher precedence than +, -.
  + Operators with higher precedence should be processed first.
* **Associativity** defines how operators of the same precedence are grouped:
  + +, -, \*, / are left-associative, meaning they group from left to right.

Try these examples:

1. x+y\*z/w+u
2. a+b\*(c^d-e)^(f+g\*h)-i

**Infix to prefix using stack:**

To convert an **infix expression** (e.g., A + B \* C) to a **prefix expression** (e.g., + A \* B C), you can use a stack with a modified approach. Here's how to do it:

### General Steps for Infix to Prefix Conversion

1. **Reverse the infix expression.**
   * Read the infix expression from **right to left** instead of left to right.
   * Reverse both operands and operators, swapping parentheses (i.e., ( becomes ) and vice versa).
2. **Replace operands, operators, and parentheses using the postfix conversion rules.**
   * Follow the rules for converting infix to postfix but with the reversed expression.
   * Use a stack to hold operators and apply the postfix conversion technique to the reversed expression.
3. **Reverse the postfix result to get the prefix expression.**

### Detailed Rules

#### 1. **Reverse the infix expression**

* Reverse the infix expression.
* Replace every left parenthesis ( with a right parenthesis ) and vice versa.

#### 2. **Convert the reversed expression to postfix**

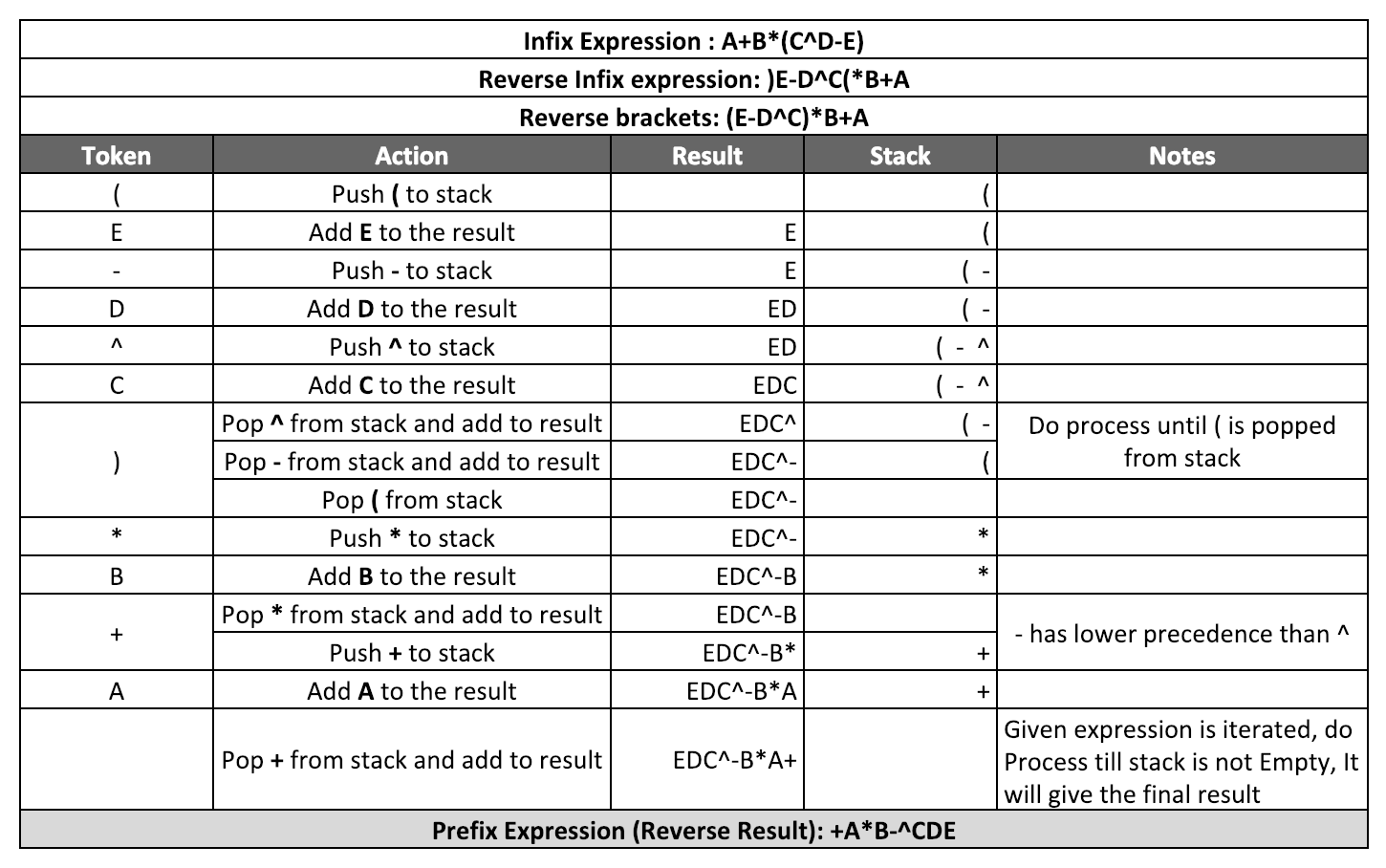
* Use the same rules as for infix to postfix:
  + If the character is an **operand** (e.g., variable or number), append it directly to the result.
  + If the character is an **operator** (e.g., +, -, \*, /):
    - Pop from the stack to the result until an operator with lower precedence is found.
    - Push the current operator onto the stack.
  + If the character is a **right parenthesis** ), push it onto the stack.
  + If the character is a **left parenthesis** (, pop from the stack to the result until a right parenthesis ) is encountered.

#### 3. **Reverse the postfix expression to get the prefix expression**

* Once you've generated the postfix expression from the reversed input, reverse it to get the final prefix expression.

### Operator Precedence and Associativity

* Operators \*, / have higher precedence than +, -.
* All operators have **left-to-right** associativity, so for operators of equal precedence, the one appearing first is popped from the stack.



Try these Examples:

1. x+y\*z/w+u
2. a+b\*(c^d-e)^(f+g\*h)-i

**Evaluation of prefix expression:**

### Prefix Evaluation (Polish Notation)

**Steps:**

1. **Initialize an empty stack.**
2. **Read the prefix expression from right to left.**
3. **For each token:**
   * If the token is an operand (number), push it onto the stack.
   * If the token is an operator (+, -, \*, /):
     + Pop the top two operands from the stack.
     + Apply the operator to these operands in the order: first operand (popped second) and second operand (popped first).
     + Push the result back onto the stack.
4. **After reading all tokens, the result will be the only value left in the stack.**

**Example:** For the prefix expression + 9 \* 2 6, the evaluation steps would be:

1. Read 6 (push onto stack).
2. Read 2 (push onto stack).
3. Read \* (pop 2 and 6, compute 2 \* 6 = 12, push 12).
4. Read 9 (push onto stack).
5. Read + (pop 9 and 12, compute 9 + 12 = 21, push 21).

**Evaluation of postfix expression:**

### Postfix Evaluation (Reverse Polish Notation)

**Steps:**

1. **Initialize an empty stack.**
2. **Read the postfix expression from left to right.**
3. **For each token:**
   * If the token is an operand (number), push it onto the stack.
   * If the token is an operator (+, -, \*, /):
     + Pop the top two operands from the stack.
     + Apply the operator to these operands in the order: first operand (popped first) and second operand (popped second).
     + Push the result back onto the stack.
4. **After reading all tokens, the result will be the only value left in the stack.**

**Example:** For the postfix expression 9 2 6 \* +, the evaluation steps would be:

1. Read 9 (push onto stack).
2. Read 2 (push onto stack).
3. Read 6 (push onto stack).
4. Read \* (pop 2 and 6, compute 2 \* 6 = 12, push 12).
5. Read + (pop 9 and 12, compute 9 + 12 = 21, push 21).