Lecture

NOTE: FOR FURTHER DETAILS AND MORE COMPREHENSIVE STUDY, PLEASE SEE RECOMMENDED BOOKS OR INTERNET.

Stack

A ***Stack*** is a collection of objects inserted and removed according to the Last In First Out (***LIFO***) principle.

Applications of Stack:

The simplest application of a stack is to reverse a word. You push a given word to stack ***letter by letter*** and then pop letters from the stack. Another application is an ***"undo"*** mechanism in text editors; this operation is accomplished by keeping all text changes in a stack. Stack is used by processor to keep the data. In this course we will look at the widely used and explained examples of stack applications. The applications of stack that we will be discussing are:

1. ***Expression Notation Conversion***
2. ***Expression Evaluation***
3. ***Solving Recursion***

First we will learn what an expression notation is.

Expression Notation & Evaluation:

Expression Notation is the representation of mathematical expressions in different forms. There are three types of notations:

1. ***Infix***
2. ***Prefix***
3. ***Postfix***

In computer science the expression which is not in infix form can be solved by the computer more easily and quickly because there are no parentheses involved.

1. Infix Notation (Normal Notation):

Normally the expressions are represented in the form of infix notation. The notation is all about the location of the operator in an expression. When the operator is in between the operands then it is called infix notation. E.g. ***a+b*** or ***4*5*** where **+** and ***** are in between the operands ***a*** and ***b***, ***4*** and ***5*** respectively.

2. Prefix Notation (Polish Notation):

"Pre" is an English word which means ***"before"***, for example pre-requisite which means required before hand.

In ***Prefix*** notation the operator is placed before the operands. E.g. ***+ab*** or ****45*** where **+** and ***** are placed before the operands ***a*** and ***b***, ***4*** and ***5*** respectively. Mathematically, ***+ab*** and ****45*** are the equivalent of ***a+b*** and ***4*5*** respectively.

3. Postfix Notation (Reverse Polish Notation):

"Post" is an English word which means ***"after"***, for example post-mortem which means after mortem or after death.

In ***Postfix*** notation the operator is placed after the operands. E.g. ***ab+*** or ***45**** where **+** and ***** are placed after the operands ***a*** and ***b***, ***4*** and ***5*** respectively. Mathematically, ***ab+*** and ***45**** are the equivalent of ***a+b*** and ***4*5*** respectively.

Examples of Conversion

Let's consider an example below for our conversion to different notations from infix notation.

The expression A+B-C*D/E+F will be solved by computer in these format based on precedence (((A+B)-((C*D)/E))+F). Parenthesizing an expression is a very difficult process. What if we change the notation of expression A+B-C*D/E+F ourselves then it won't require parenthesizing process.

Prefix:

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

A+B-**[*CD]**/**E**+F	next we need solve	A+B-**[*CD]/E**+F
A+B-**[/*CDE]**+F	next we need solve	**A+B**-[*CD]/E+F
[+AB]-**[/*CDE]**+F	next we need solve	**[+AB]-[/*CDE]**+F
[-+AB/*CDE]+F	next we need solve	**[-+AB/*CDE]+F**
[+-+AB/*CDEF]	Prefix Expression	**+-+AB/*CDEF**

Postfix:

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

A+B-**[CD*]**/**E**+F	next we need solve	A+B-**[CD*]/E**+F
A+B-**[CD*E/]**+F	next we need solve	**A+B**-[CD*E/]+F
[AB+]-**[CD*E/]**+F	next we need solve	**[AB+]-[CD*E/]**+F
[AB+CD*E/-]+F	next we need solve	**[AB+CD*E/-]+F**
[AB+CD*E/-F+]	Prefix Expression	**AB+CD*E/-F+**

Converting Infix to Postfix Notation using Stack

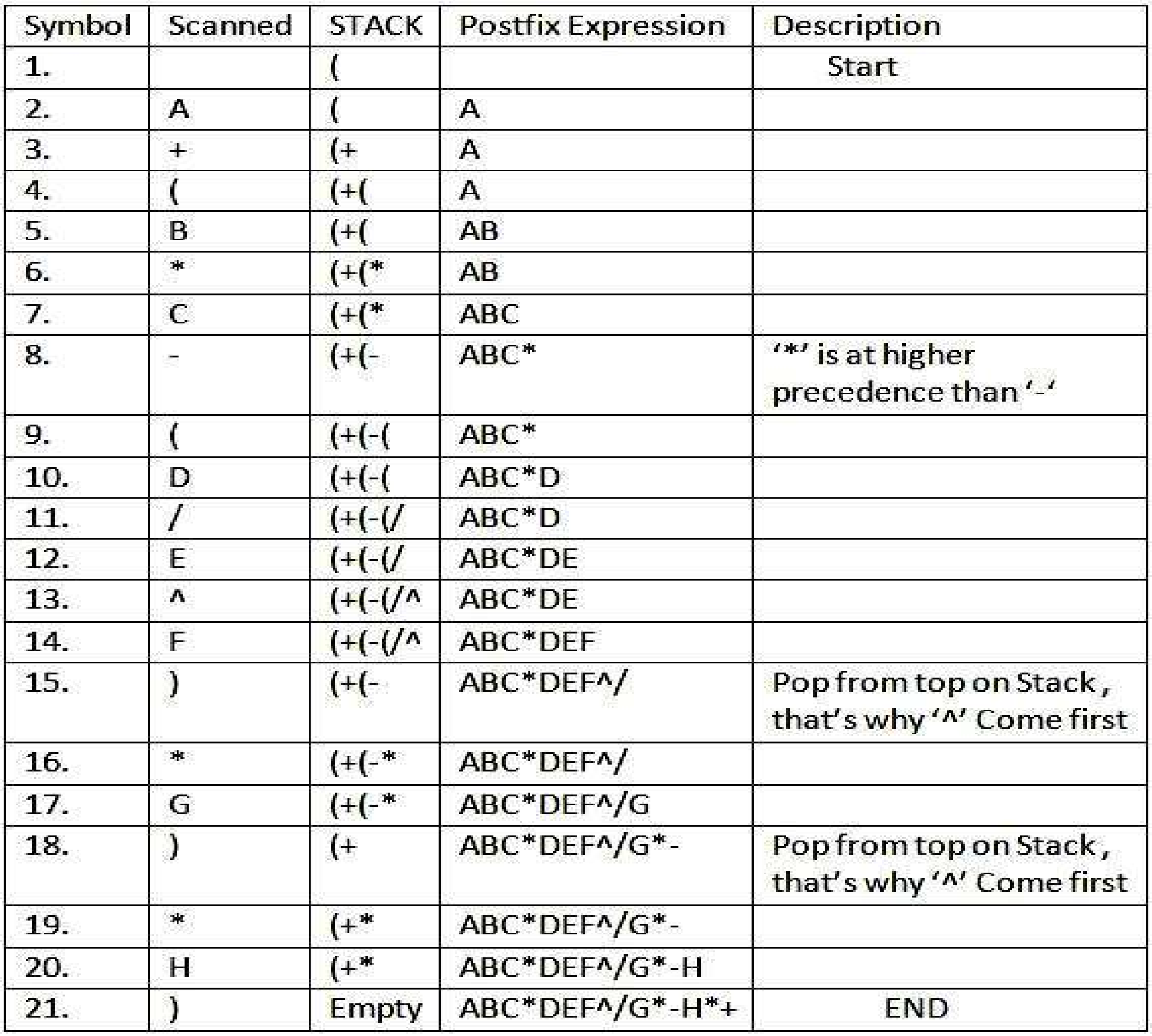
The following algorithm is followed when converting infix expression to postfix expression using stack.

Algorithm:

1. Start
2. Scan ***expression*** from left to right ***character by character.***

3. Encountered character is evaluated for one of the following ***four cases***:
   a. If ***character or digit*** is encountered then add it to your ***solved expression***.
   b. If ***'('*** is encountered then ***push*** it onto STACK.
   c. If ***')'*** is encountered then ***repeatedly pop*** from STACK and add it the ***solved Expression*** until STACK is empty or encountered ***'('***. Discard both the ***'('*** and ***')'***.
   d. If ***an operator*** is encountered then ***repeatedly pop*** from STACK each operator which has the ***same or higher precedence*** than operator encountered.
4. Repeat ***2*** and ***3*** until reach end of ***expression***.
5. ***Repeatedly pop*** from stack and add to the ***solved expression*** until STACK is empty.
6. End.

Let's take an example to better understand the algorithm. A+(B*C-(D/E^F)*G)*H, where ^ is an exponential operator.



Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Resultant Postfix Expression: **ABC*DEF^/G*-H*+**

Code for Infix to Postfix Notation Conversion:

The following code is the Method for converting infix expression to postfix expression.

```
public static String i2p(String exp)
{
    MyStack.nSize = exp.length();
    MyStack st = new MyStack();

    String cExp = "";
    for (int i = 0; i < exp.length(); i++)
    {
        char c = exp.charAt(i);

        if (Character.isLetterOrDigit(c))
            cExp = cExp + c;

        else if (c == '(')
            st.push(c);

        else if (c == ')')
        {
            while (!st.isEmpty() && st.peek() != '(')
                cExp += st.pop();

            if (st.peek() == '(')
                st.pop();
        }
        else
        {
            while (!st.isEmpty() && prec(c) < prec(st.peek()))

                cExp += st.pop();

            st.push(c);
        }
    }
    while (!st.isEmpty())
        cExp += st.pop();
    return cExp;
}

private static int prec(char op)
{
    if (op == '^')
        return 3;

    else if (op == '*' || op == '/')
        return 2;

    else if (op == '+' || op == '-')
        return 1;

    else
        return 0;
}
```

Converting Infix to Prefix Notation using Stack

The following algorithm is followed when converting infix expression to prefix expression using stack.

Algorithm:

1. Start
2. ***Reverse*** the infix expression, i.e. ***A+B*C*** will become C*B+A. Note while reversing each ***'('*** will become ***')'*** and each ***')'*** becomes ***'('***.
3. Apply ***postfix conversion*** on the *reversed expression*.
4. ***Reverse*** the ***solved expression*** in **3** again to get the ***infix expression***.
5. End