CSE222-HW4-Question1
Report

Nevzat Seferoglu

171044024

i) Given infix expression ;

A + ((B – C * D) / E) + F – G / H

Firstly , we need an algorithm that can be appropriate for stacking process. After algorithm constructed well , stack usage for converting infix to postfix can be demonstrated with table step by step.

Algorithm that I need for **infix to postfix conversion**;

- There are two different kind of elements in my expression. Those are operand and operation , according to algorithm , all infix character will take step by step from the structure and according to type of character , there will be a stack which can keep the operations on it with correct order**. Operands can be directly appended to newly created postfix expression when encountered.**

  1) If stack is empty or contains a left parenthesis on the top , push the incoming operator to the stack.
  2) If incoming symbol is '(' , push it onto stack.
  3) If incoming symbol is ')' , pop the stack and append the operator to postfix structure until left parenthesis is encountered.
  4) If incoming symbol has higher precedence than the top of the stack , push it on the stack.
  5) If incoming symbol has lower precedence than the top of the stack , pop and append to postfix. Then 1- consider the incoming operator against the new top of the stack.
  6) If incoming operator has equal precedence with the top of the stack , use associativity rule.
  7) At the end of the expression , pop and append all remain operators of the stack.
    - Note : Associativity is crucial factor ;
      - Left to Right , pop the operator from the stack and append to postfix , then push the incoming operator.
      - Right to Left, then push the incoming operator.

Next Token	Action	Effect on operator Stack	Effect on postfix
A	Appended to postfix		A
+	Pushed to stack	+	A
(	Pushed to stack	+ (	A
(	Pushed to stack	+ ((	A
B	Appended to postfix	+ ((	AB
-	Pushed to stack.	+ ((-	AB
C	Appended to postfix	+ ((-	ABC
*	Pushed to stack	+ ((- *	ABC
D	Appended to postfix	+ ((- *	ABCD
)	3th rule	+ (	ABCD*-
/	Pushed to stack	+ (/	ABCD*-
E	Appended to postfix	+ (/	ABCD*-E
)	3th rule	+	ABCD*-E/
+	6th rule	+	ABCD*-E/+
F	Appended to postfix	+	ABCD*-E/+F
-	6th rule	-	ABCD*-E/+F+
G	Appended to postfix	-	ABCD*-E/+F+G
/	Pushed to stack	- /	ABCD*-E/+F+G
H	Appended to postfix	- /	ABCD*-E/+F+GH
End of the tokens	7th rule	-	ABCD*-E/+F+GH/
End of the tokens	7th rule		ABCD*-E/+F+GH/-

New postfix expression : **A B C D * - E / + F + G H / -**

Algorithm that I need for **evaluation of postfix expression**;

- For each character in postfix expression , do
  If operand is encountered , push it onto stack
  else if operator is encountered , pop top 2 elements from the stack
  A > Top element
  B > Next to top element

  result = B operator (current operator) A
  push result onto stack
  return element of the stack top

For make evaluation process clear , handling with reel number will be more beneficial. Therefore, I gave a number for each unknown operands of postfix expression.

Postfix Expression ;
ABCD*-E/+F+GH/- , if A = 5 , B = 4 , C = 2 , D = 1 , E = 2 , F = 3 , G = 7 , H = 1
expression will be ;
5421*-2/+3+71/-

Next Token	Action	Effect on Operand Stack	Effect on result
5	Pushed onto stack	5	
4	Pushed onto stack	5 4	
2	Pushed onto stack	5 4 2	
1	Pushed onto stack	5 4 2 1	
*	Popped first two elements and make operation on it	5 4 2	2
-	Popped first two elements and make operation on it	5 2	2
2	Pushed onto stack	5 2 2	2
/	Popped first two elements and make operation on it	5 1	1
+	Popped first two elements and make operation on it	6	6
3	Pushed onto stack	6 3	6
+	Popped first two elements and make operation on it	9	9
7	Pushed onto stack	9 7	9
1	Pushed onto stack	9 7 1	9
/	Popped first two elements and make operation on it	9 7	7
-	Popped first two elements and make operation on it	2	**Result = 2**

Given infix expression ;

A + ((B – C * D) / E) + F – G / H

Firstly , we need an algorithm that can be appropriate for stacking process. After algorithm constructed well , stack usage for converting **infix to prefix** can be demonstrated with table step by step.

Algorithm that I need for **infix to prefix conversion**;

- There are two crucial tricky for converting infix to prefix expression ;
  - Postfix conversion can be used in this process . Therefore, fist we take the input infix expression as an **inverse** of given infix expression. **Then infix expression to postfix expression can be used. After the applied postfix operation , for getting the actual prefix expression , inversion must be applied again.**
  - In postfix process expression , there was a step which is number 6 , there will be a changing on this step. Previous step , according to associativity two different step is applied . However, in this version , if incoming operation has same precedence with top element of the stack , it will be pushed to the stack.

Inverse of current infix expression : **H / G – F +) E /) D * C – B ((+ A**

Next Token	Action	Effect on operator Stack	Effect on prefix
H	Appended to prefix		H
/	Pushed to stack.	/	H
G	Appended to prefix	/	HG
-	5th rule of (infix to postfix)	-	HG/
F	Appended to prefix	-	HG/F
+	New 6th rule	- +	HG/F
)	Pushed to stack	- +)	HG/F
E	Appended to prefix	- +)	HG/FE
/	Pushed to stack	- +) /	HG/FE
)	Pushed to stack	- +) /)	HG/FE
D	Appended to prefix	- +) /)	HG/FED
*	Pushed to stack	- +) /) *	HG/FED
C	Appended to prefix	- +) /) *	HG/FEDC
-	5th rule of (infix to postfix)	- +) /) -	HG/FEDC*
B	Appended to prefix	- +) /) -	HG/FEDC*B
(	3th rule of (infix to postfix)	- +) /	HG/FEDC*B-
(	3th rule of (infix to postfix)	- +	HG/FEDC*B-/
+	New 6th rule	- + +	HG/FEDC*B-/
A	Appended to prefix	- + +	HG/FEDC*B-/A
End of tokens	7th rule of (infix to postfix)	- +	HG/FEDC*B-/A+
End of tokens	7th rule of (infix to postfix)	-	HG/FEDC*B-/A++
End of tokens	7th rule of (infix to postfix)		HG/FEDC*B-/A++-

New prefix expression : **- + + A / - B * C D E F / G H**

Algorithm that I need for **evaluation of prefix expression**;

- For each character in postfix expression , do
  If operand is encountered , push it onto stack
  else if operator is encountered , pop top 2 elements from the stack
  A > Top element
  B > Next to top element

  result = A operator (current operator) B **(Changing Here)**
  push result onto stack
  return element of the stack top

For make evaluation process clear , handling with reel number will be more beneficial. Therefore, I gave a number for each unknown operands of postfix expression.

Tokens have taken from right to left.

Prefix Expression ;
-++A/-B*CDEF/GH, if A = 5 , B = 4 , C = 2 , D = 1 , E = 2 , F = 3 , G = 7 , H = 1
expression will be ;
-++5/-4*2123/71

Next Token	Action	Effect on Operand Stack	Effect on result
1	Pushed onto stack	1	
7	Pushed onto stack	1 7	
/	Popped first two elements and make operation on it	7	7
3	Pushed onto stack	7 3	7
2	Pushed onto stack	7 3 2	7
1	Pushed onto stack	7 3 2 1	7
2	Pushed onto stack	7 3 2 1 2	7
*	Popped first two elements and make operation on it	7 3 2 2	2
4	Pushed onto stack	7 3 2 2 4	2
-	Popped first two elements and make operation on it	7 3 2 2	2
/	Popped first two elements and make operation on it	7 3 1	1
5	Pushed onto stack	7 3 1 5	1
+	Popped first two elements and make operation on it	7 3 6	6
+	Popped first two elements and make operation on it	7 9	9
-	Popped first two elements and make operation on it	2	**Result = 2 , which is same as postfix evaluation result.**

ii) Given infix expression ;

! (A && ! ((B < C) || (C > D))) || (C < E)

Firstly , we need an algorithm that can be appropriate for stacking process. After algorithm constructed well , stack usage for converting infix to postfix can be demonstrated with table step by step.

Algorithm that I need for **infix to postfix conversion**;

- There are two different kind of elements in my expression. Those are operand and operation , according to algorithm , all infix character will take step by step from the structure and according to type of character , there will be a stack which can keep the operations on it with correct order**. Operands can be directly appended to newly created postfix expression when encountered.**

  1) If stack is empty or contains a left parenthesis on the top , push the incoming operator to the stack.
  2) If incoming symbol is '(' , push it onto stack.
  3) If incoming symbol is ')' , pop the stack and append the operator to postfix structure until left parenthesis is encountered.
  4) If incoming symbol has higher precedence than the top of the stack , push it on the stack.
  5) If incoming symbol has lower precedence than the top of the stack , pop and append to postfix. Then 1-consider the incoming operator against the new top of the stack.
  6) If incoming operator has equal precedence with the top of the stack , use associativity rule.
  7) At the end of the expression , pop and append all remain operators of the stack.
    - Note : Associativity is crucial factor ;
      - Left to Right , pop the operator from the stack and append to postfix , then push the incoming operator.
      - Right to Left, then push the incoming operator.

Next Token	Action	Effect on operator Stack	Effect on postfix
!	Pushed onto stack	!	
(	Pushed onto stack	! (	
A	Appended to postfix	! (	A
&&	Pushed onto stack	! (&&	A
!	Pushed onto stack	! (&& !	A
(	Pushed onto stack	! (&& ! (	A
(	Pushed onto stack	! (&& ! ((	A
B	Appended to postfix	! (&& ! ((	A B
<	Pushed onto stack	! (&& ! ((<	A B
C	Appended to postfix	! (&& ! ((<	A B C
)	3th rule	! (&& ! (	A B C <
||	Pushed onto stack	! (&& ! (||	A B C <
(	Pushed onto stack	! (&& ! (|| (	A B C <
C	Appended to postfix	! (&& ! (|| (	A B C < C
>	Pushed onto stack	! (&& ! (|| (>	A B C < C
D	Appended to postfix	! (&& ! (|| (>	A B C < C D
)	3th rule	! (&& ! (||	A B C < C D >
)	3th rule	! (&& !	A B C < C D > ||
)	3th rule	!	A B C < C D > || ! &&
||	5th rule	||	A B C < C D > || ! && !
(	Pushed onto stack	|| (	A B C < C D > || ! && !
C	Appended to postfix	|| (	A B C < C D > || ! && ! C
<	Pushed onto stack	|| (<	A B C < C D > || ! && ! C
E	Appended to postfix	|| (<	A B C < C D > || ! && ! C E
)	3th rule	||	A B C < C D > || ! && ! C E <
End of the tokens	7th rule		A B C < C D > || ! && ! C E < ||

New postfix form : **A B C < C D > || ! && ! C E < ||**

Algorithm that I need for **evaluation of postfix expression**;

- For each character in postfix expression , do
  If operand is encountered , push it onto stack
  else if operator is encountered , pop top 2 elements from the stack
  A > Top element
  B > Next to top element

  result = B operator (current operator) A
  push result onto stack
  return element of the stack top

For make evaluation process clear , handling with reel number will be more beneficial. Therefore, I gave a number for each unknown operands of postfix expression.

Postfix Expression ;

ABC<CD>||!&&!CE<||, if **A = 1 , B = 3 , C = 4 , D = 2 , E = 1**

expression will be ;

134<42>||!&&!41<|| ,

Next Token	Action	Effect on Operand Stack	Effect on result
1	Pushed onto stack	1	
3	Pushed onto stack	1 3	
4	Pushed onto stack	1 3 4	
<	Popped first two elements and make operation on it	1 0	0
4	Pushed onto stack	1 0 4	0
2	Pushed onto stack	1 0 4 2	0
>	Popped first two elements and make operation on it	1 0 1	1
||	Popped first two elements and make operation on it	1 1	1
!	Popped first element and make operation on it	1 0	0
&&	Popped first two elements and make operation on it	0	0
!	Popped first element and make operation on it	1	1
4	Pushed onto stack	1 4	1
1	Pushed onto stack	1 4 1	1
<	Popped first two elements and make operation on it	1 0	0
||	Popped first two elements and make operation on it	1	**Result = 1**

Given infix expression ;

! (A && ! ((B < C) || (C > D))) || (C < E)

Firstly , we need an algorithm that can be appropriate for stacking process. After algorithm constructed well , stack usage for converting **infix to prefix** can be demonstrated with table step by step.

Algorithm that I need for **infix to prefix conversion**;

- There are two crucial tricky for converting infix to prefix expression ;
  - Postfix conversion can be used in this process . Therefore, fist we take the input infix expression as an **inverse** of given infix expression. **Then infix expression to postfix expression can be used. After the applied postfix operation , for getting the actual prefix expression , inversion must be applied again.**
  - In postfix process expression , there was a step which is number 6 , there will be a changing on this step. Previous step , according to associativity two different step is applied . However, in this version , if incoming operation has same precedence with top element of the stack , it will be pushed to the stack.

Inverse of current infix expression : **) E < C (||))) D > C (||) C < B ((! && A (!**

Next Token	Action	Effect on operator Stack	Effect on prefix
)	Pushed onto stack	)	
E	Appended to prefix	)	E
<	Pushed onto stack	) <	E
C	Appended to prefix	) <	EC
(	3th rule of (infix to postfix)		EC<
||	Pushed onto stack	||	EC<
)	Pushed onto stack	||)	EC<
)	Pushed onto stack	||)))	EC<
)	Pushed onto stack	||)))	EC<
D	Appended to prefix	||)))	EC<D
>	Pushed onto stack	||))) >	EC<D
C	Appended to prefix	||))) >	EC<DC
(	3th rule of (infix to postfix)	||))	EC<DC>
||	Pushed onto stack	||)) ||	EC<DC>
)	Pushed onto stack	||)) ||)	EC<DC>
C	Appended to prefix	||)) ||)	EC<DC>C
<	Pushed onto stack	||)) ||) <	EC<DC>C
B	Appended to prefix	||)) ||) <	EC<DC>CB
(	3th rule of (infix to postfix)	||)) ||	EC<DC>CB<
(	3th rule of (infix to postfix)	||)	EC<DC>CB<||
!	Pushed onto stack	||) !	EC<DC>CB<||
&&	5th rule of (infix to postfix)	||) &&	EC<DC>CB<||!
A	Appended to prefix	||) &&	EC<DC>CB<||!A
(	3th rule of (infix to postfix)	||	EC<DC>CB<||!A&&
!	Pushed onto stack	|| !	EC<DC>CB<||!A&&
End of tokens	7th rule of (infix to postfix)	||	EC<DC>CB<||!A&&!
End of tokens	7th rule of (infix to postfix)		EC<DC>CB<||!A&&!||

New prefix expression : **|| ! && A ! || < B C > C D < C E**

Algorithm that I need for **evaluation of prefix expression**;

- For each character in postfix expression , do
  If operand is encountered , push it onto stack
  else if operator is encountered , pop top 2 elements from the stack
  A > Top element
  B > Next to top element

  result = A operator (current operator) B **(Changing Here)**
  push result onto stack
  return element of the stack top

For make evaluation process clear , handling with reel number will be more beneficial. Therefore, I gave a number for each unknown operands of postfix expression.

Tokens have taken from right to left.

Prefix Expression ;

||!&&A!||<BC>CD<CE, if **A = 1 , B = 3 , C = 4 , D = 2 , E = 1**

expression will be ;

||!&&1!||<34>42<41

Next Token	Action	Effect on Operand Stack	Effect on result
1	Pushed onto stack	1	
4	Pushed onto stack	1 4	
<	Popped first two elements and make operation on it	0	0
2	Pushed onto stack	0 2	0
4	Pushed onto stack	0 2 4	0
>	Popped first two elements and make operation on it	0 1	1
4	Pushed onto stack	0 1 4	1
3	Pushed onto stack	0 1 4 3	1
<	Popped first two elements and make operation on it	0 1 0	0
||	Popped first two elements and make operation on it	0 1	1
!	Popped first element and make operation on it	0 0	0
1	Pushed onto stack	0 0 1	0
&&	Popped first two elements and make operation on it	0 0	0
!	Popped first element and make operation on it	0 1	1
||	Popped first two elements and make operation on it	1	**Result = 1 , which is same as postfix evaluation result.**