GEBZE TECHNICAL UNIVERSİTY

CSE 222- HW4 QUESTION 1

1) A + ((B – C * D) / E) + F – G / H

Prefix)

Step 1 : I took the reverse of explanation. And

H / G – F + (E / (D * C – B)) + A

Step 2 : Like doing postfix , I started first character of explanation.

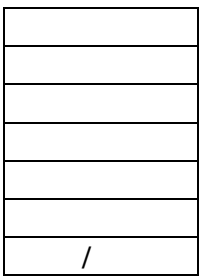
H / G – F + (E / (D * C – B)) + A

Stack Output : **H**

Step 3 : Operators goes to the stack

H / G – F + (E / (D * C – B)) + A

Stack Output : **H**



Step 4 : G goes to output

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG**

Step 5 : - priority less than / . So / goes to output and – push into the stack

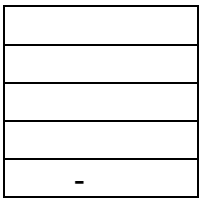
H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/**

Step 6 : F goes to the output

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/F**



Step 7 : + goes to the stack and – pop from stack

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/F-**

+

Step 8 : (goes to the stack

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/F-**

(
+

Step 9 : E goes to the output

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/F-E**

(
+

Step 10 : / goes to the stack

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/F-E**

/
(
+

Step 11 : (goes to the stack

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/F-E**

(
/
(
+

Step 12 : D goes to the output

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/F-ED**

(
/
(
+

Step 13 : * goes to the stack

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/F-ED**

*
(
/
(
+

Step 14 : C goes to the output

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/F-EDC**

*
(
/
(
+

Step 15 : - priority less than * . So * goes to output and – push into the stack

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/F-EDC***

-
(
/
(
+

Step 16 : B goes to the output

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/F-EDC*B**

-
(
/
(
+

Step 17 : We saw the close parentheses so we pop the operator until we saw open parentheses. Therefore we push - to the output and pop from stack. (also will pop from stack.

H / G – F + (E / (D * C – B)) + A

Stack Output : **HG/F-EDC*B-**

/
(
+

Step 18 : We saw another close parentheses so we pop the operator until we saw open parentheses. Therefore we push / to the output and pop from stack. (also will pop from stack.

H / G – F + (E / (D * C) – B)) + A

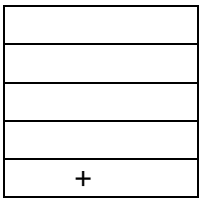
Stack Output : **HG/F-EDC*B-/**

+

Step 19 : + goes to the stack and other + pop from stack

H / G – F + (E / (D * C) – B)) + A

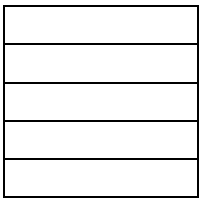
Stack Output : **HG/F-EDC*B-/+**



Step 20 : A goes to the output and we don't have more elements so we pop operators from stack and push into the output

H / G – F + (E / (D * C) – B)) + A

Stack Output : **HG/F-EDC*B-/+A+**



Step 21 : We are trying to prefix of that expressions. So we reverse the output and we will find prefix.

Prefix : **+ A + / - B * C D E – F / G H**

Evaluate : Expression

Infix : A + ((B – C * D) / E) + F – G / H

Prefix : A + / - B * C D E – F / G H

A : 2

B : 34

C : 7

D : 4

E : 6

F : 9

G : 12

H : 4

Infix : 2 + ((34 – 7 * 4) / 6) + 9 – 12/ 4 = 9

This is my prefix to Infix : (A+(((B-(C*D))/E)+(F-(G/H)))) =

2+(((34-(7*4))/6)+(9-(12/4)))) = 9 And now let's check postfix expression below.

1.b) A + ((B – C * D) / E) + F – G / H

Postfix

Step 1 : I started first character of explanation. A goes to the output

A + ((B – C * D) / E) + F – G / H

Stack Output : **A**

Step 2 : + goes to the stack

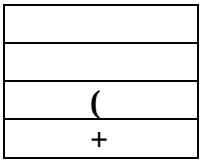
A + ((B – C * D) / E) + F – G / H

Stack Output : **A**

Step 3 : (goes to the stack

A + ((B – C * D) / E) + F – G / H

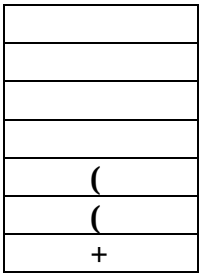
Stack Output : **A**



Step 4 : (goes to the stack

A + ((B – C * D) / E) + F – G / H

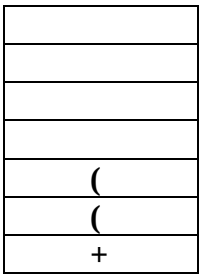
Stack Output : **A**



Step 5 : B goes to the output

A + ((B – C * D) / E) + F – G / H

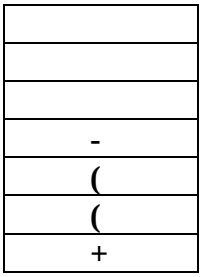
Stack Output : **AB**



Step 6 : - goes to the stack

A + ((B – C * D) / E) + F – G / H

Stack Output : **AB**



Step 7 : C goes to the output

A + ((B – C * D) / E) + F – G / H

Stack Output : **ABC**

-
(
(
+

Step 8 : *'s priority is greater than -. So we won't pop from stack and * push into the stack

A + ((B – C * D) / E) + F – G / H

Stack Output : **ABC**

*
-
(
(
+

Step 9 : D goes to the output

A + ((B – C * D) / E) + F – G / H

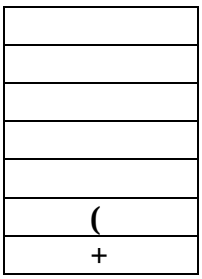
Stack Output : **ABCD**

*
-
(
(
+

Step 10 : We saw the close parentheses so we pop the operator until we saw open parentheses. Therefore we push * and - to the output and pop from stack. (also will pop from stack.

A + ((B – C * D) / E) + F – G / H

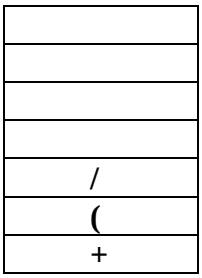
Stack Output : **ABCD*-**



Step 11 : / goes to the stack

A + ((B – C * D) / E) + F – G / H

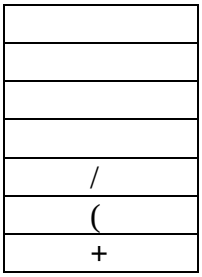
Stack Output : **ABCD*-**



Step 12 : E goes to the output

A + ((B – C * D) / E) + F – G / H

Stack Output : **ABCD*- E**



Step 13 : We saw the close parentheses so we pop the operator until we saw open parentheses. Therefore we push / to the output and pop from stack. (also will pop from stack.

A + ((B – C * D) / E) + F – G / H

Stack Output : **ABCD*- E/**

+

Step 14 : +'s priority is equal +. So we pop from stack + and other + goes to the stack

A + ((B – C * D) / E) + F – G / H

Stack Output : **ABCD*- E/+**

+

Step 15 : F goes to the output

A + ((B – C * D) / E) + F – G / H

Stack Output : **ABCD*- E/+F**

+

Step 16 : -'s priority is equal +. So we pop from stack + and - goes to the stack

A + ((B – C * D) / E) + F – G / H

Stack Output : **ABCD*- E/+F+**

Step 17 : G goes to the output

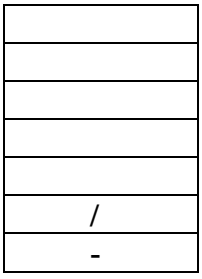
A + ((B – C * D) / E) + F – G / H

Stack Output : **ABCD*- E/+F+G**

Step 18 : /'s priority is greater than – so we won't pop from stack and we push / to the stack

A + ((B – C * D) / E) + F – G / H

Stack Output : **ABCD*- E/+F+G**



Step 19 : H goes to the output and we don't have more characters so we pop from stack these operators.First we pop last in / and then pop - .

A + ((B – C * D) / E) + F – G / H

Stack

Output : **ABCD*- E/+F+GH/-**

AND WE HAVE POSTFIX EXPRESSION : **A B C D * - E / + F + G H / -**

Evaluate : Expression

Infix : A + ((B – C * D) / E) + F – G / H

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

A : 2

B : 34

C : 7

D : 4

E : 6

F : 9

G : 12

H : 4

Infix : 2 + ((34 – 7 * 4) / 6) + 9 – 12/ 4 = 9

This is my postfix to Infix : (((A+((B-(C*D))/E))+F)-(G/H))=

(((2+((34-(7*4))/6))+9)-(12/4)) = 9

Postfix and prefix are equal so these expressions are true.

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

Prefix) Firstly I'd like to say that I gave priority * before / and + before –

Step 1 : I took the reverse of explanation. And

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

Step 2 : Like doing postfix , I started first character of explanation. (goes to the stack

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

Stack Output :

Step 3 : E goes to the output

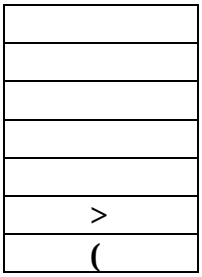
(E > C) || (((D < C) || (C > B)) ! && A) !

Stack Output : **E**

Step 4 : > goes to the stack

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

Stack Output : **E**



Step 5 : C goes to the output

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

Stack Output : **EC**

Step 6 : We saw the close parentheses so we pop the operator until we saw open parentheses. Therefore we push > to the output and pop from stack. (also will pop from stack.

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

Stack Output : **EC>**

Step 7 : || goes to the stack

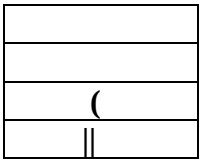
(E > C) || (((D < C) || (C > B)) ! && A) !

Stack Output : **EC>**

Step 8 : (goes to the stack

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

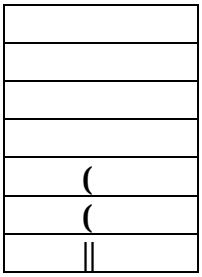
Stack Output : **EC>**



Step 9 : (goes to the stack

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

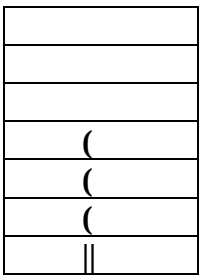
Stack Output : **EC>**



Step 10 : (goes to the stack

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

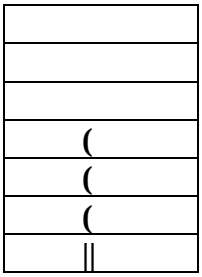
Stack Output : **EC>**



Step 11: D goes to the output

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

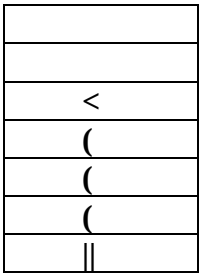
Stack Output : **EC>D**



Step 12: < goes to the stack

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

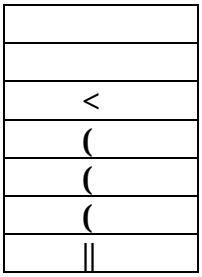
Stack Output : **EC>D**



Step 13: C goes to the output

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

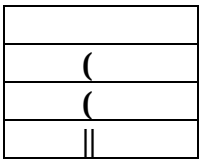
Stack Output : **EC>DC**



Step 14: We saw close parentheses so we pop the operator until we saw open parentheses. Therefore we push < to the output and pop from stack. (also will pop from stack.

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

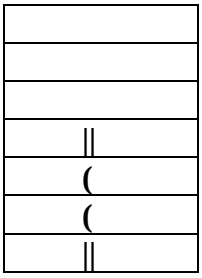
Stack Output : **EC>DC<**



Step 15: || goes to the stack

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

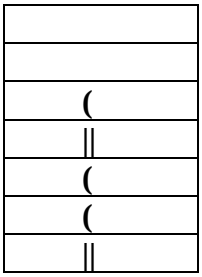
Stack Output : **EC>DC<**



Step 16: (goes to the stack

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

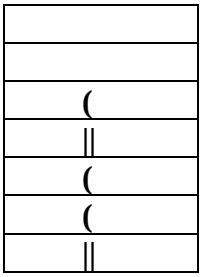
Stack Output : **EC>DC<**



Step 17: C goes to the output

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

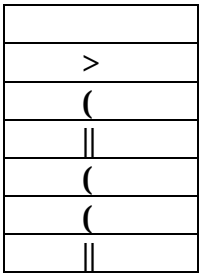
Stack Output : **EC>DC<C**



Step 18: > goes to the stack

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

Stack Output : **EC>DC<C**



Step 19: B goes to the output

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

Stack Output : **EC>DC<CB**

>
(
(
(

Step 20: We saw another close parentheses so we pop the operator until we saw open parentheses. Therefore we push > to the output and pop from stack. (also will pop from stack.

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

Stack Output : **EC>DC<CB>**

(
(

Step 21: We saw another close parentheses so we pop the operator until we saw open parentheses. Therefore we push || to the output and pop from stack. (also will pop from stack.

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

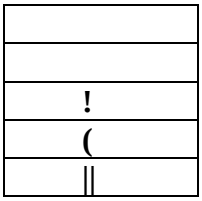
Stack Output : **EC>DC<CB>||**

(

Step 22: ! goes to the stack

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

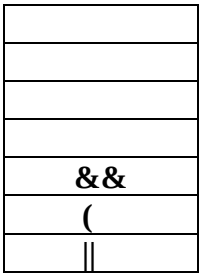
Stack Output : **EC>DC<CB>||**



Step 23: &&'s priority is less than !'s priority. So we pop ! from stack and push into the output. And then we push && into the stack

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

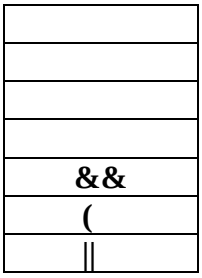
Stack Output : **EC>DC<CB>||!**



Step 24: A goes to the output

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

Stack Output : **EC>DC<CB>||!A**



Step 25: We saw another close parentheses so we pop the operator until we saw open parentheses. Hence we push && to the output and pop from stack. (also will pop from stack.

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

Stack Output : **EC>DC<CB>||!A&&**

Step 26: !'s priority is greater than || . So we won't pop from stack and we push ! into the stack.

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

Stack Output : **EC>DC<CB>||!A&&**

Step 27: And finally, we come end of the expression. So we orderly pop from stack and push into the output.

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

Stack Output : **EC>DC<CB>||!A&&!||**

Step 28 : We are trying to prefix of that expressions. So we reverse the output and we will find prefix.

Prefix : || ! && A ! || < B C > C D < C E

Evaluate : Expression

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

Prefix : || ! && A ! || < B C > C D < C E

A : 5

B : 7

C : 8

D : 4

E : 2

Infix : ! (5 && !((7 < 8) || (8>4))) || (8<2) = 1 || 0 = 1 (True)

7 < 8 True

8 > 4 True

8<2 False

!(True || True) = False

!(5 && False)= True

True || False = True

This is my prefix to Infix : (!(! ((C > D) || (B < C)) && A) || (C < E)

!(!((8 > 4) || (7 < 8)) && 5) || (8 < 2) = 1 || 0 = 1 (True)

8>4 : True

7< 8 : True

8<2 : False

!(True || True) = False

(False && 5) = False

!(False) || False = True || False = True

And now let's check postfix expression below.

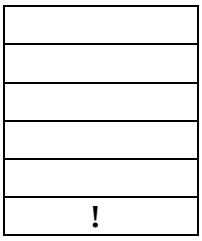
2.b) **! (A && ! ((B < C) || (C > D))) || (C < E)**

Postfix)

Step 1 : I started first character of expression. ! goes to the stack.

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

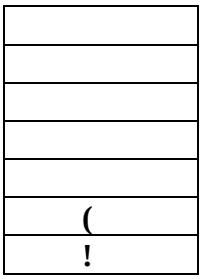
Stack Output :



Step 2: (goes to the stack.

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

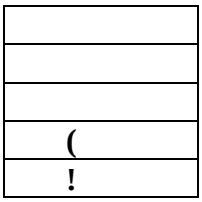
Stack Output :



Step 3: A goes to the output.

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

Stack Output : **A**



Step 4: && goes to the stack.

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

Stack Output : **A**

&&
(
!

Step 5: !'priority is greater than &&. So we won't pop anything else push ! into the stack

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

Stack Output : **A**

!
&&
(
!

Step 6: (goes to the stack

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

Stack Output : **A**

(
!
&&
(
!

Step 7: (goes to the stack

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

Stack Output : **A**

(
(
!
&&
(
!

Step 8: B goes to the output.

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

Stack Output : **AB**

(
(
!
&&
(
!

Step 9: < goes to the stack.

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

Stack Output : **AB**

<
(
(
!
&&
(
!

Step 10: C goes to the output.

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

Stack Output : **ABC**

<
(
(
!
&&
(
!

Step 11: We saw close parentheses so we pop the operator until we saw open parentheses. Therefore we push < to the output and pop from stack. (also will pop from stack.

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

Stack Output : **ABC<**

(
!
&&
(
!

Step 12: || goes to the stack.

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

Stack Output : **ABC<**

(
!
&&
(
!

Step 13: (goes to the stack.

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

Stack

(
||
(
!
&&
(
!

Output : **ABC<**

Step 14: C goes to the output.

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

Stack

(
||
(
!
&&
(
!

Output : **ABC<C**

Step 15: > goes to the stack.

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

Stack

>
(
||
(
!
&&
(
!

Output : **ABC<C**

Step 16: D goes to the output.

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

Stack Output : **ABC<CD**

>
(
(
!
&&
(
!

Step 17: We saw close parentheses so we pop the operator until we saw open parentheses. Therefore we push > to the output and pop from stack. (also will pop from stack.

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

Stack Output : **ABC<CD>**

(
!
&&
(
!

Step 18: We saw close parentheses so we pop the operator until we saw open parentheses. Therefore we push || to the output and pop from stack. (also will pop from stack.

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

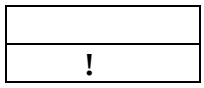
Stack Output : **ABC<CD>||**

!
&&
(
!

Step 19: We saw close parentheses so we pop the operator until we saw open parentheses. Hence we push ! and && to the output and pop from stack. (also will pop from stack.

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

Stack

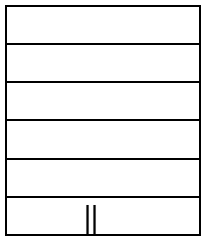


Output : **ABC<CD>||!&&**

Step 20: ||'s priority is less than ! . So ! pop from stack and || push to the stack

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

Stack

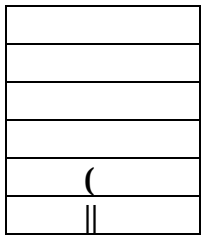


Output : **ABC<CD>||!&&!**

Step 21: (goes to the stack

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

Stack

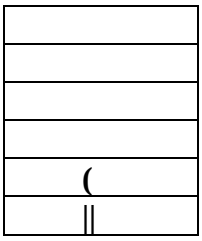


Output : **ABC<CD>||!&&!**

Step 22: C goes to the output

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

Stack

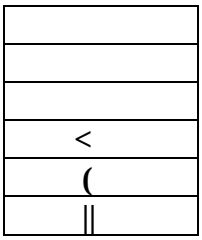


Output : **ABC<CD>||!&&!C**

Step 23: < goes to the stack

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

Stack

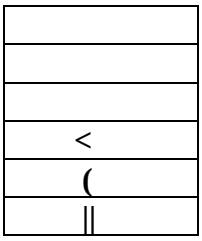


Output : **ABC<CD>||!&&!C**

Step 24: E goes to the output

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

Stack

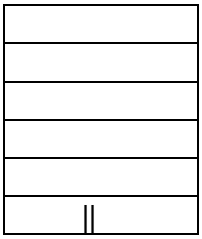


Output : **ABC<CD>||!&&!CE**

Step 25: We saw close parentheses so we pop the operator until we saw open parentheses. Therefore we push < to the output and pop from stack. (also will pop from stack.

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

Stack



Output : **ABC<CD>||!&&!CE<**

Step 26: And finally we don't have more characters . So || goes to the output

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

Stack

Output : **ABC<CD>||!&&!CE<||**

AND WE HAVE POSTFIX EXPRESSION : A B C < C D > || ! && ! C E < ||

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

! (5 && !((7 < 8) || (8>4))) || (8<2) = 1 || 0 = 1 (True)

This is my postfix to infix expression : (! (A && ! ((B < C) || (C > D))) || (C < E))

(! (5 && ! ((7 < 8) || (8 > 4))) || (8< 2)) = 1 || 0 = 1 (True)

So this expression is true

Evaluate : Expression

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

Postfix : A B C < C D > || ! && ! C E < ||

A : 5

B : 7

C : 8

D : 4

E : 2

Infix : ! (5 && !((7 < 8) || (8>4))) || (8<2) = 1 || 0 = 1 (True)

(7 < 8) True

(8 > 4) True

(8<2) False

!(True || True) = False

!(5 && False)= True

True || False = True

This is my postfix to Infix : (! (A && ! ((B < C) || (C > D))) || (C < E))

(! (5 && ! ((7 < 8) || (8 > 4))) || (8< 2)) = 1 || 0 = 1 (True)

8>4 : True

7< 8 : True

8<2 : False

!(True || True) = False

(5 && False) = False

!(False) || False = True || False = True

These prefix, postfix and infix are equal. So they are true expressions.