Infix to prefix (Efficient Approach)

I/p: x + y * z

o/p: + x * y z

I/p : (x + y) * z

o/p : * + x y z

I/p : a ^ b ^ c

o/p : ^ a ^ b c

Infix to Prefix (Efficient Approach)

I/p: x + y * z

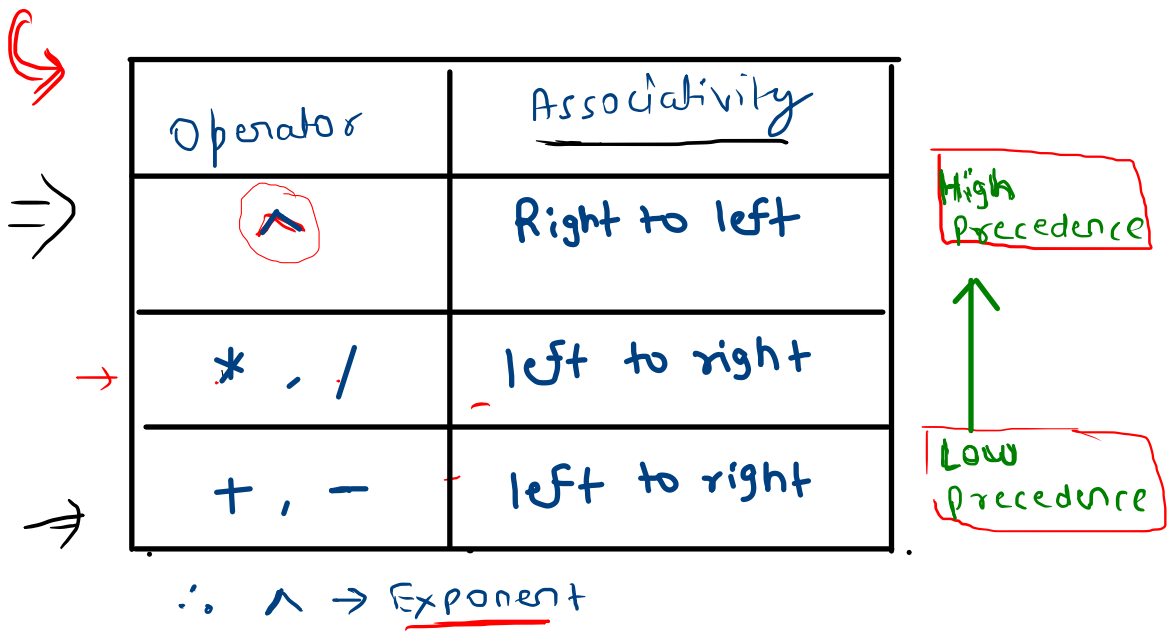
o/p: + x * y z

I/p : (x + y) * z

o/p : * + x y z

I/p : a ^ b ^ c

o/p : ^ a ^ b



∴ ^ → Exponent

NAIVE APPROACH

Infix to Prefix (Efficient Approach)

I/p: x + y * z ⇒ (x + (y * z)) ⇒ x + (*yz) ⇒ +x*yz

O/p: +x*yz

I/p: (x+y) * z ⇒ (+xy) * z ⇒ *+xyz

O/p: *+xyz

I/p: a^(b^c) ⇒ a^(b^c) ⇒ a^(^bc) ⇒ ^a^bc

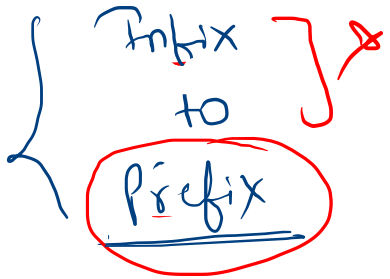
O/p: ^a^bc

Algorithm

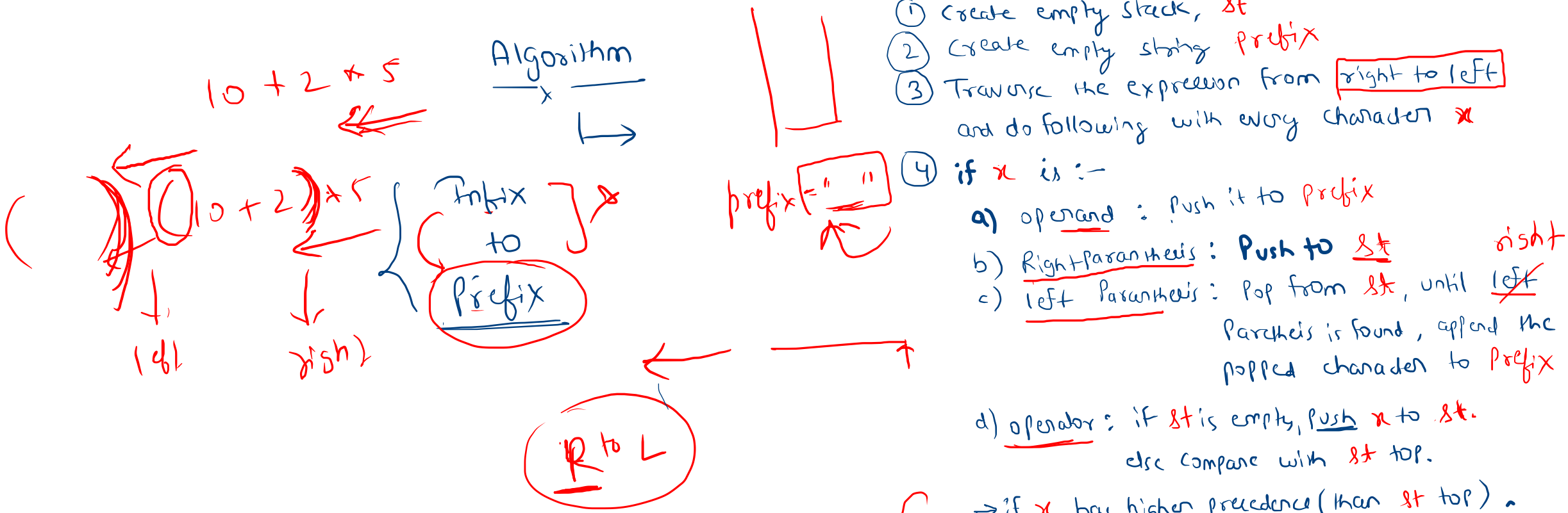
Infix to Postfix

1. Create empty stack, st
2. Traverse the expression from left to right and do following with every character x
3. if x is :-
   a) operand : print it
   b) left paranthesis : **Push to** st
   c) Right Paranthesis : Pop from st, until left Parthesis is found, output the popped character
   d) operator : if st is empty, Push x to st.
      else compare with st top.
   → if x has higher precedence (than st top). **Push** to st.
   → Lower precedence, pop st top, and output until a higher precedence operator is found. **then** push x to st.
   → **Equal** precedence, use associativity.
4. pop and output Everything from st.

Algorithm



1. Create empty stack, st
2. Create empty string prefix
3. Traverse the expression from right to left and do following with every character x
4. if x is :-
   a) operand : Push it to prefix
   b) RightParanthesis : Push to st
   c) LeftParanthesis : Pop from st, until left Paranthesis is found, append the popped character to Prefix
   d) operator : if st is empty, Push x to st.
      else compare with st top.
   → if x has higher precedence (than st top).
   Push to st.
   → Lower precedence, pop st top, and append to prefix until a higher precedence operator is found.
   then push x to st.
   → Equal precedence, use associativity.
5. pop and append Everything from st. to Prefix
6. Return Reverse of Prefix



1. Create empty stack, st
2. Create empty string prefix
3. Traverse the expression from right to left and do following with every character x
4. if x is :-
   a) operand : Push it to prefix
   b) RightParanthesis : Push to st
   c) left Paranthesis : Pop from st, until left ~~left~~ right Paranthesis is found, append the popped character to Prefix
   d) operator : if st is empty, Push x to st.
      else compare with st top.
      → if x has higher precedence (than st top). Push to st.
      → Lower precedence, pop st top, and append to prefix until a higher precedence operator is found. then push x to st.
      → Equal precedence, use associativity.
5. pop and append Everything from st. to prefix
6. Return Reverse of Prefix

Ex → I/p: a + b * d ← + a * b d

Input Symbol	Stack	Prefix (Reverse)
d		d
*	*	d
b	*	db
+	+	db*
a	+	db*a
pop all		db*at





→ (+ a * b d)

1. Create empty stack, st
2. Create empty string prefix
3. Traverse the expression from right to left and do following with every character x
4. if x is :-
   a) operand : Push it to prefix
   b) RightParanthesis : Push to st
   c) left Paranthesis : Pop from st, until right Paranthesis is found, append the popped character to prefix
   d) operator : if st is empty, push x to st.
      else compare with st top.
   → if x has higher precedence (than st top). Push to st.
   → Lower precedence, pop st top, and append to prefix until a higher precedence operator is found. then push x to st.
   → Equal precedence, use associativity.
5. pop and append Everything from st. to prefix
6. Return Reverse of prefix

Ex: (a + b) * d

o/p:

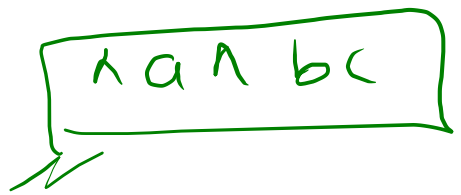
Input Symbol	Stack	Prefix (Reverse)
d		d
*	*	d
)	) *	d
b	) *	db
+	) + *	db
a	) + *	dba
(	*	dba+
pop all		dba+*
reverse		(*+abd)

1. Create empty stack, st
2. Create empty string prefix
3. Traverse the expression from right to left and do following with every character x
4. if x is :-
   a) operand : Push it to prefix
   b) RightParanthesis : Push to st
   c) left Paranthesis : Pop from st, until right Paranthesis is found, append the popped character to Prefix
   d) operator : if st is empty, push x to st. else compare with st top.
   → if x has higher precedence (than st top). Push to st.
   → Lower precedence, pop st top, and append to prefix until a higher precedence operator is found. then push x to st.
   → Equal precedence, use associativity.
5. pop and append Everything from st. to Prefix
6. Return Reverse of Prefix

Ex: $a \wedge_1 b \wedge_2 c$ ← $\wedge_1$ $\wedge_2$ R to L

o/p:

Input Symbol	Stack	Prefix (Reverse)
c		c
$\wedge_2$	$\wedge_2$	c
b	$\wedge_2$	cb
$\wedge_1$	$\wedge_1$	$cb\wedge_2$
a	$\wedge_1$	$cb\wedge_2 a$
		$cb\wedge a\wedge$



1. Create empty stack, st
2. Create empty string prefix
3. Traverse the expression from right to left and do following with every character x
4. if x is :-
   a) operand : Push it to prefix
   b) Right Parantheis : Push to st
   c) left Paranthesis : Pop from st, until ~~left~~ right Parantheis is found, append the popped character to prefix
   d) operator : if st is empty, push x to st.
      else compare with st top.
   → if x has higher precedence (than st top). Push to st.
   → Lower precedence, pop st top, and append to prefix until a higher precedence operator is found. then push x to st.
   → Equal precedence, use associativity.
5. pop and append Everything from st. to prefix
6. Return Reverse of prefix

Ex: x + y / z - w * u ← L to R

o/p:

Input Symbol	Stack	Prefix (Reverse)
u	+	u
*	+ *	u
w	+ *	u w
-	- *	u w *
z	-	u w * z
/	- /	u w * z
y	- /	u w * z y
+	+ -	u w * z y /
x	+ -	u w * z y / x
		u w * z y / x + -

sol

rev ← - + x / y z * w u

1) Create empty stack, st
2) Create empty string prefix
3) Traverse the expression from right to left and do following with every character x
4) if x is :-
   a) operand : Push it to prefix
   b) Right Paranthesis : Push to st
   c) left Paranthesis : Pop from st, until right Paranthesis is found, append the popped character to Prefix
   d) operator : if st is empty, Push x to st.
      else compare with st top.
   → if x has higher precedence (than st top). Push to st.
   → Lower precedence, pop st top, and append to prefix until a higher precedence operator is found. then push x to st.
   → Equal precedence, use associativity.
5) pop and append Everything from st. to Prefix
6) Return Reverse of Prefix