Prefix, Postfix, Infix Notation

Infix Notation

- To add A, B, we write
  A+B
- To multiply A, B, we write
  A*B
- The operators ('+' and '*') go in between the operands ('A' and 'B')
- This is "*Infix*" notation.

Prefix Notation

- Instead of saying "A plus B", we could say "add A,B " and write

  + A B

- "Multiply A,B" would be written

  * A B

- This is *Prefix* notation.

Postfix Notation

- Another alternative is to put the operators after the operands as in

  A B +

and

  A B *

- This is *Postfix* notation.

Pre A In B Post

- The terms infix, prefix, and postfix tell us whether the operators go between, before, or after the operands.

Parentheses

- Evaluate 2+3*5.
- + First:

  (2+3)*5 = 5*5 = 25
- * First:

  2+(3*5) = 2+15 = 17
- Infix notation requires Parentheses.

What about Prefix Notation?

- + 2 * 3 5 =

  $= + 2\ \underline{*\ 3\ 5}$

  $= \underline{+\ 2\ 15} = 17$

- * + 2 3 5 =

  $= *\ \underline{+\ 2\ 3}\ 5$

  $= \underline{*\ 5\ 5} = 25$

- No parentheses needed!

Postfix Notation

- 2 3 5 * + =

  $= 2\ \underline{3\ 5\ *}\ +$

  $= \underline{2\ 15\ +} = 17$

- 2 3 + 5 * =

  $= \underline{2\ 3\ +}\ 5\ *$

  $= \underline{5\ 5\ *} = 25$

- No parentheses needed here either!

Conclusion:

- Infix is the only notation that requires parentheses in order to change the order in which the operations are done.

Fully Parenthesized Expression

- A FPE has exactly one set of Parentheses enclosing each operator and its operands.
- Which is fully parenthesized?

(A + B) * C

★ ((A + B) * C)

((A + B) * (C))

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

((A + B) * (C + D))

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

$$\begin{tikzcd} (& + A \quad B & * \arrow[ll, bend left] & (C + D)) \end{tikzcd}$$

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

$$\begin{tikzcd}[column sep=tiny] * & + & A & B & (& C & + \arrow[ll, bend left] & D &) \end{tikzcd}$$

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

* + A B + C D

Order of operands does not change!

Infix to Postfix

$$\begin{tikzcd}[column sep=tiny]
(& (& (& A & + \arrow[rr, bend right] & B &) & * \arrow[rr, bend right] & C &) & - \arrow[rrrrrrrrrr, bend right] & (& (& D & + \arrow[rr, bend right] & E &) & / \arrow[rr, bend right] & F &) &)
\end{tikzcd}$$

A B + C * D E + F / -

- Operand order does not change!
- Operators are in order of evaluation!

Computer Algorithm FPE Infix To Postfix

- Assumptions:

1. Space delimited list of tokens represents a FPE infix expression
2. Operands are single characters.
3. Operators +,-,*,/

FPE Infix To Postfix

- Initialize a Stack for operators, output list
- Split the input into a list of tokens.
- for each token (left to right):

```
    if it is operand:  append to output
    if it is '(': push onto Stack
    if it is ')': pop & append till '('
```

FPE Infix to Postfix

(((A + B) * (C - E)) / (F + G))

▲

- stack: <empty>
- output: []

FPE Infix to Postfix

((A + B) * (C - E)) / (F + G))

▲

- stack: (
- output: []

FPE Infix to Postfix

((A + B) * (C - E)) / (F + G))

▲

- stack: ((
- output: []

FPE Infix to Postfix

A + B) * (C - E)) / (F + G))

▲

- stack: (((
- output: []

FPE Infix to Postfix

+ B) * (C - E)) / (F + G))

▲

- stack: (((
- output: [A]

FPE Infix to Postfix

B) * (C - E)) / (F + G))

▲

- stack: (((+
- output: [A]

FPE Infix to Postfix

) * (C - E)) / (F + G))

- stack: (((+
- output: [A B]

FPE Infix to Postfix

* (C - E)) / (F + G))

▲

- stack: ((
- output: [A B +]

FPE Infix to Postfix

(C - E)) / (F + G))

▲

- stack: ((*
- output: [A B +]

FPE Infix to Postfix

C - E)) / (F + G))

▲

- stack: ((* (
- output: [A B +]

FPE Infix to Postfix

- E)) / (F + G))

▲

- stack: ((* (
- output: [A B + C]

FPE Infix to Postfix

E)) / (F + G))

▲

- stack: ((* (-
- output: [A B + C]

FPE Infix to Postfix

)) / (F + G))

▲

- stack: ((* (-
- output: [A B + C E]

FPE Infix to Postfix

) / (F + G))

▲

- stack: ((*
- output: [A B + C E -]

FPE Infix to Postfix

/ (F + G))

△

- stack: (
- output: [A B + C E - *]

FPE Infix to Postfix

(F + G))

- stack: (/
- output: [A B + C E - *]

FPE Infix to Postfix

F + G))

▲

- stack: (/ (
- output: [A B + C E - *]

FPE Infix to Postfix

+ G))

▲

- stack: (/ (
- output: [A B + C E - * F]

FPE Infix to Postfix

G))

- stack: (/ (+
- output: [A B + C E - * F]

FPE Infix to Postfix

))

▲

- stack: (/ (+
- output: [A B + C E - * F G]

FPE Infix to Postfix

)

▲

- stack: (/
- output: [A B + C E - * F G +]

FPE Infix to Postfix

- stack: <empty>
- output: [A B + C E - * F G + /]

Problem with FPE

- Too many parentheses.
- Establish precedence rules:

  My Dear Aunt Sally
- We can alter the previous program to use the precedence rules.

Infix to Postfix

- Initialize a Stack for operators, output list
- Split the input into a list of tokens.
- for each token (left to right):

```
if it is operand:  append to output
if it is '(': push onto Stack
if it is ')': pop & append till '('
if it in '+-*/':
    while peek has precedence ≥ it:
            pop & append
    push onto Stack
```

pop and append the rest of the Stack.

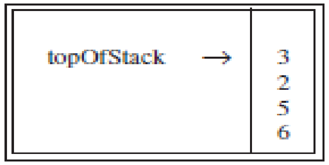
POST FIX EXPRESSION FROM IN TEXT BOOK

two numbers (symbols) that are popped from the stack, and the result is pushed onto the stack. For instance, the postfix expression

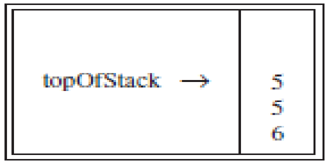
$$6\ 5\ 2\ 3 + 8 * + 3 + *$$

is evaluated as follows:

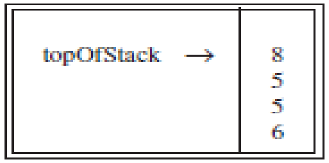
The first four symbols are placed on the stack. The resulting stack is



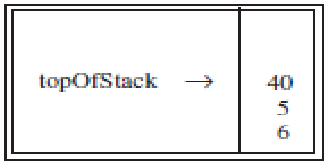
Next, a '+' is read, so 3 and 2 are popped from the stack, and their sum, 5, is pushed.



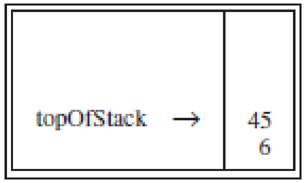
Next, 8 is pushed.



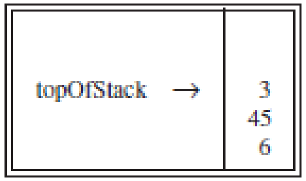
Now a '*' is seen, so 8 and 5 are popped, and $5 * 8 = 40$ is pushed.



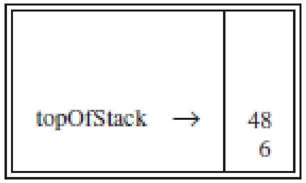
Next, a '+' is seen, so 40 and 5 are popped, and $5 + 40 = 45$ is pushed.



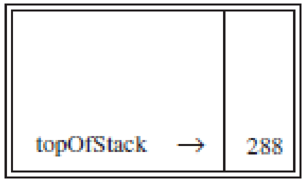
Now, 3 is pushed.



Next, '+' pops 3 and 45 and pushes $45 + 3 = 48$.

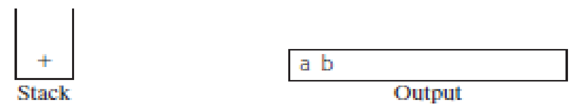


Finally, a '*' is seen and 48 and 6 are popped; the result, $6 * 48 = 288$, is pushed.

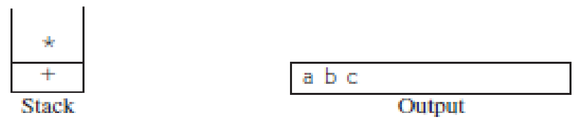


INFIX TO POST FIX EXAMPLE IN TEXT BOOK

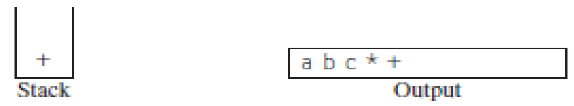
Then + is read and pushed onto the stack. Next b is read and passed through to the output. The state of affairs at this juncture is as follows:



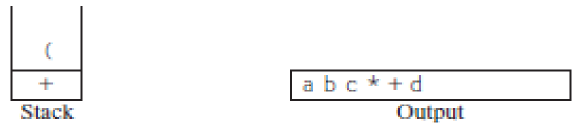
Next, a * is read. The top entry on the operator stack has lower precedence than *, so nothing is output and * is put on the stack. Next, c is read and output. Thus far, we have



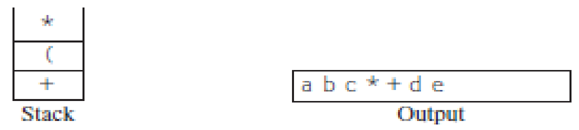
The next symbol is a +. Checking the stack, we find that we will pop a * and place it on the output; pop the other +, which is not of *lower* but equal priority, on the stack; and then push the +.



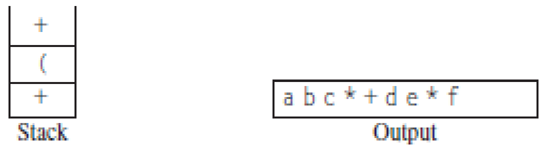
The next symbol read is a (. Being of highest precedence, this is placed on the stack. Then d is read and output.



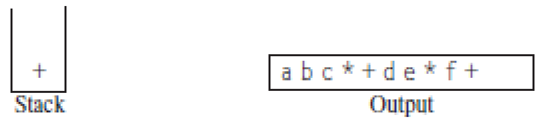
We continue by reading a *. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.



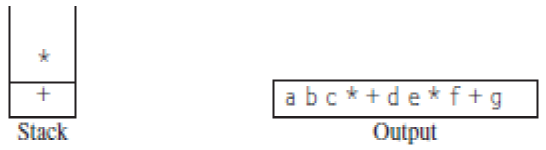
The next symbol read is a +. We pop and output * and then push +. Then we read and output f.



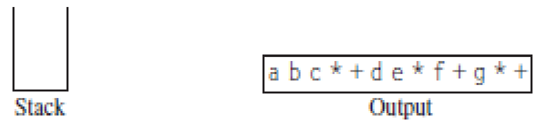
Now we read a), so the stack is emptied back to the (. We output a +.



We read a * next; it is pushed onto the stack. Then g is read and output.



The input is now empty, so we pop and output symbols from the stack until it is empty.



As before, this conversion requires only $O(N)$ time and works in one pass through the input. We can add subtraction and division to this repertoire by assigning subtraction and addition equal priority and multiplication and division equal priority. A subtle point is that the expression a - b - c will be converted to a b - c - and not a b c - -. Our algorithm does the right thing, because these operators associate from left to right. This is not necessarily the case in general, since exponentiation associates right to left: $2^{2^3} = 2^8 = 256$, not $4^3 = 64$. We leave as an exercise the problem of adding exponentiation to the repertoire of operators.