Polish Notation

Algebraic Expression

- An algebraic expression is a legal combination of operands and the operators.
- Operand is the quantity (unit of data) on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4,0,9,1 etc.
- Operator is a symbol which signifies a mathematical or logical operation between the operands.
- Example of familiar operators include +,-,*, /, ^ An example of expression as x+y*z.

Infix, Postfix and Prefix Expressions

INFIX: the expressions in which operands surround the operator, e.g. x+y, 6*3 etc this way of writing the Expressions is called infix notation.

POSTFIX: Postfix notation are also Known as Reverse Polish Notation (RPN). They are different from the infix and prefix notations in the sense that in the postfix notation, operator comes after the operands, e.g. xy+, xyz+* etc.

PREFIX: Prefix notation also Known as Polish notation. In the prefix notation, operator comes before the operands, e.g. +xy, *+xyz etc.

Operator Priorities

How do you figure out the operands of an operator?

- a + b * c
- a * b + c / d

This is done by assigning operator priorities.

- priority(*) = priority(/) > priority(+) = priority(-)

When an operand lies between two operators, the operand associates with the operator that has higher priority.

Tie Breaker

When an operand lies between two operators that have the same priority, the operand associates with the operator on the left.

- a + b - c
- a * b / c / d

Delimiters

Sub expression within delimiters is treated as a single operand, independent from the remainder of the expression.

- (a + b) * (c – d) / (e – f)

WHY??

Why to use PREFIX and POSTFIX notations when we have simple INFIX notation?

INFIX notations are not as simple as they seem specially while evaluating them. To evaluate an infix expression we need to consider Operators' Priority and Associative property

- E.g. expression 3+5*4 evaluate to 32 i.e. (3+5)*4 or to 23 i.e. 3+(5*4).

To solve this problem Precedence or Priority of the operators were defined. Operator precedence governs evaluation order. An operator with higher precedence is applied before an operator with lower precedence.

Infix Expression is Hard To Parse

- Need operator priorities, tie breaker, and delimiters.
- This makes computer evaluation more difficult than is necessary.
- Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters.
- Both prefix and postfix notations have an advantage over infix that while evaluating an expression in prefix or postfix form we need not consider the Priority and Associative property (order of brackets).
- • E.g. x/y*z becomes */xyz in prefix and xy/z* in postfix. Both prefix and postfix notations make Expression Evaluation a lot easier.
- So it is easier to evaluate expressions that are in these forms.

Examples of infix to prefix and post fix

Infix	Postfix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Example: postfix expressions

- Postfix notation is another way of writing arithmetic expressions.
- In postfix notation, the operator is written after the two operands.
    - infix: 2+5 postfix: 2 5 +
- Expressions are evaluated from left to right.
- Precedence rules and parentheses are never needed!!

Suppose that we would like to rewrite A+B*C in postfix

- Applying the rules of precedence, we obtained

A+B*C

A+(B*C) Parentheses for emphasis

A+(BC*) Convert the multiplication,Let D=BC*

A+D Convert the addition

A(D)+

ABC*+ Postfix Form

Postfix Examples

Infix	Postfix	Evaluation
2 - 3 * 4 + 5	2 3 4 * - 5 +	-5
(2 - 3) * (4 + 5)	2 3 - 4 5 + *	-9
2- (3 * 4 +5)	2 3 4 * 5 + -	-15

Why ? No brackets necessary !

Algorithm for Infix to Postfix

Examine the next element in the input.

- If it is operand, output it.
- If it is opening parenthesis, push it on stack.
- If it is an operator, then
  i) If stack is empty, push operator on stack.
  ii) If the top of stack is opening parenthesis, push operator on stack

iii) If it has higher priority than the top of stack, push operator on stack.

iv) Else pop the operator from the stack and output it, repeat step 4

- If it is a closing parenthesis, pop operators from stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
- If there is more input go to step 1
- If there is no more input, pop the remaining operators to output

Step 1: Add “)” to the end of the infix expression

Step 2: Push “(“ on to the stack.

Step 3: Repeat until each character in the infix notation is scanned.

(a) if a “(“ is encountered , push it on the stack.

(b) if an operand (weather a digit or character ,is encountered . Add it to the postfix expression.

(c) If a “)” is encountered then

i. Repeatedly pop from stack and add it to the postfix expression until a “(“ is encountered.

ii. Discard the “(“ . That is , remove the “(“ stack and do not add it to the postfix expression.

(d) If an operator 0 is encountered , then

i, Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than 0

ii. Push the operator 0 to the stack.

[End If]

Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty.

Step 5 : Exit.

Suppose we want to convert 2*3/(2-1)+5*3 into Postfix form,

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

Example

- (5 + 6) * 9 +10

will be

- 5 6 + 9 * 10 +

Convert the following infix expression into postfix expression

(1) (A – B) * (C + D)

(2) (A + B) / (C + D) - (D * E)

Convert the following infix expression into prefix expression

(1) (A – B) * C

(2) (A – B) * (C + D)

(3) (A + B) / (C + D) - (D * E)

Convert the following infix expression into postfix expression

(a) (A – (B / C + (D % E * F) / G) * H)

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
-	(-	A
(	(-(	A
B	(-(	AB
/	(-(/	AB
C	(-(/	ABC
+	(-(+	ABC/
(	(-(+(	ABC/
D	(-(+(	ABC/D
%	(-(+(%	ABC/D
E	(-(+(%	ABC/DE
*	(-(+(%*	ABC/DE
F	(-(+(%*	ABC/DEF
)	(-(+	ABC/DEF*%
/	(-(+/	ABC/DEF*%
G	(-(+/	ABC/DEF*%G
)	(-	ABC/DEF*%G/+
*	(-*	ABC/DEF*%G/+
H	(-*	ABC/DEF*%G/+H
)		ABC/DEF*%G/+*-

The steps in converting the infix expression

a / b * (c + (d - e))

to postfix form

Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (
c	*a b* / *c*	* (
+	*a b* / *c*	* (+
(	*a b* / *c*	* (+ (
d	*a b* / *c d*	* (+ (
–	*a b* / *c d*	* (+ (–
e	*a b* / *c d e*	* (+ (–
)	*a b* / *c d e* –	* (+ (
	a b / *c d e* –	* (+
)	*a b* / *c d e* – +	* (
	a b / *c d e* – +	*
	a b / *c d e* – + *	

Evaluation a postfix expression

The ease of evaluation acts as the driving force for computers to translate an infix notation into a postfix notation. That is, given an algebric expression written in infix notation, the computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression

Both these tasks converting the infix notation into postfix notation and evaluating the postfix expression make extensive use of Stacks as the primary tool.

Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack. Howevr , is an operator is encountered, then the top values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack.

Evaluation a postfix expression

- Each operator in a postfix string refers to the previous two operands in the string.
- Suppose that each time we read an operand we push it into a stack. When we reach an operator, its operands will then be top two elements on the stack
- We can then pop these two elements, perform the indicated operation on them, and push the result on the stack.
- So that it will be available for use as an operand of the next operator.

Evaluating Postfix Notation

- Use a stack to evaluate an expression in postfix notation.
- The postfix expression to be evaluated is scanned from left to right.
- Variables or constants are pushed onto the stack.
- When an operator is encountered, the indicated action is performed using the top elements of the stack, and the result replaces the operands on the stack.

Algorithm to Evaluate a Postfix Expression

Step 1: Add a “)” at the end of the postfix expression

Step 2: Scan every character of the postfix expression and repeat. Steps 3 and 4 until “)” is encountered.

Step 3. If an operand is encountered ,push it on the stack, If an operator O is encountered then

(a) Pop the top two elements from the stack, as A and B as A and B.

(b) Evaluate B O A, where A is the topmost element and B is the element below A.

(c) Push the result of evaluation on the stack.

(d) [End of If]

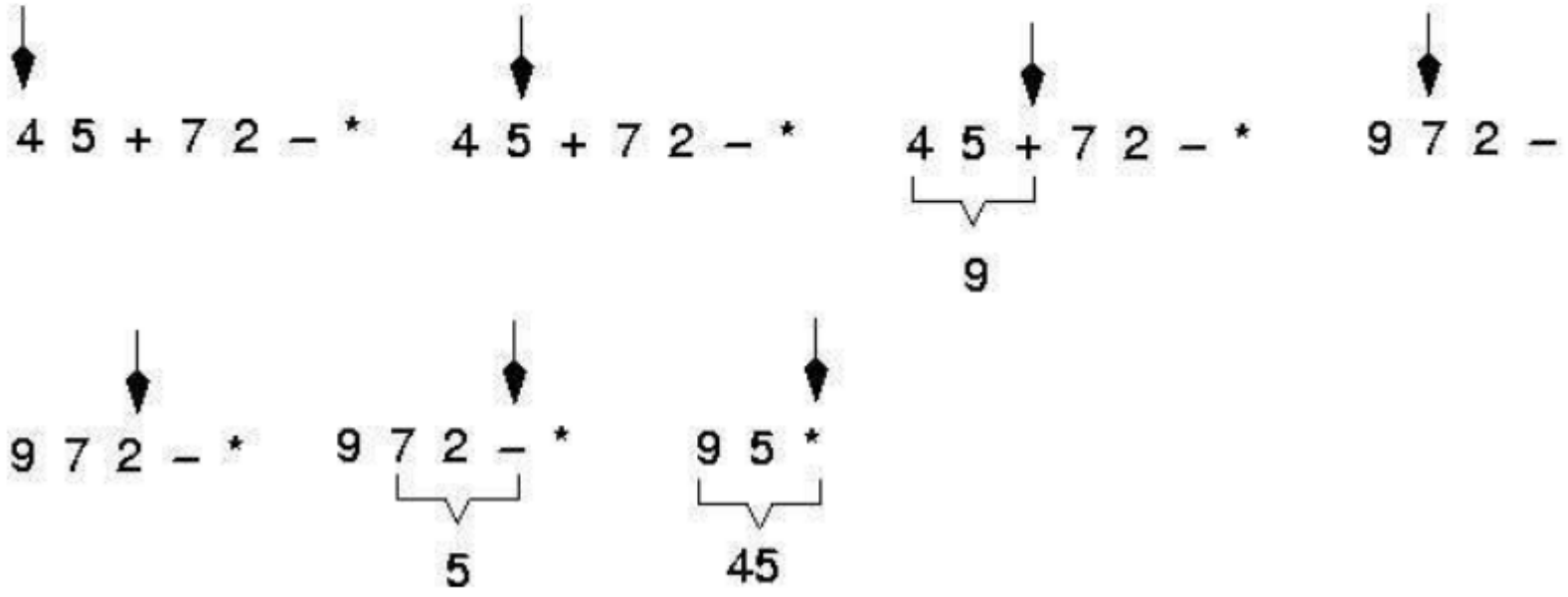
Step 4: Set Result equal to the top most element of the stack.

Step 5: Exit

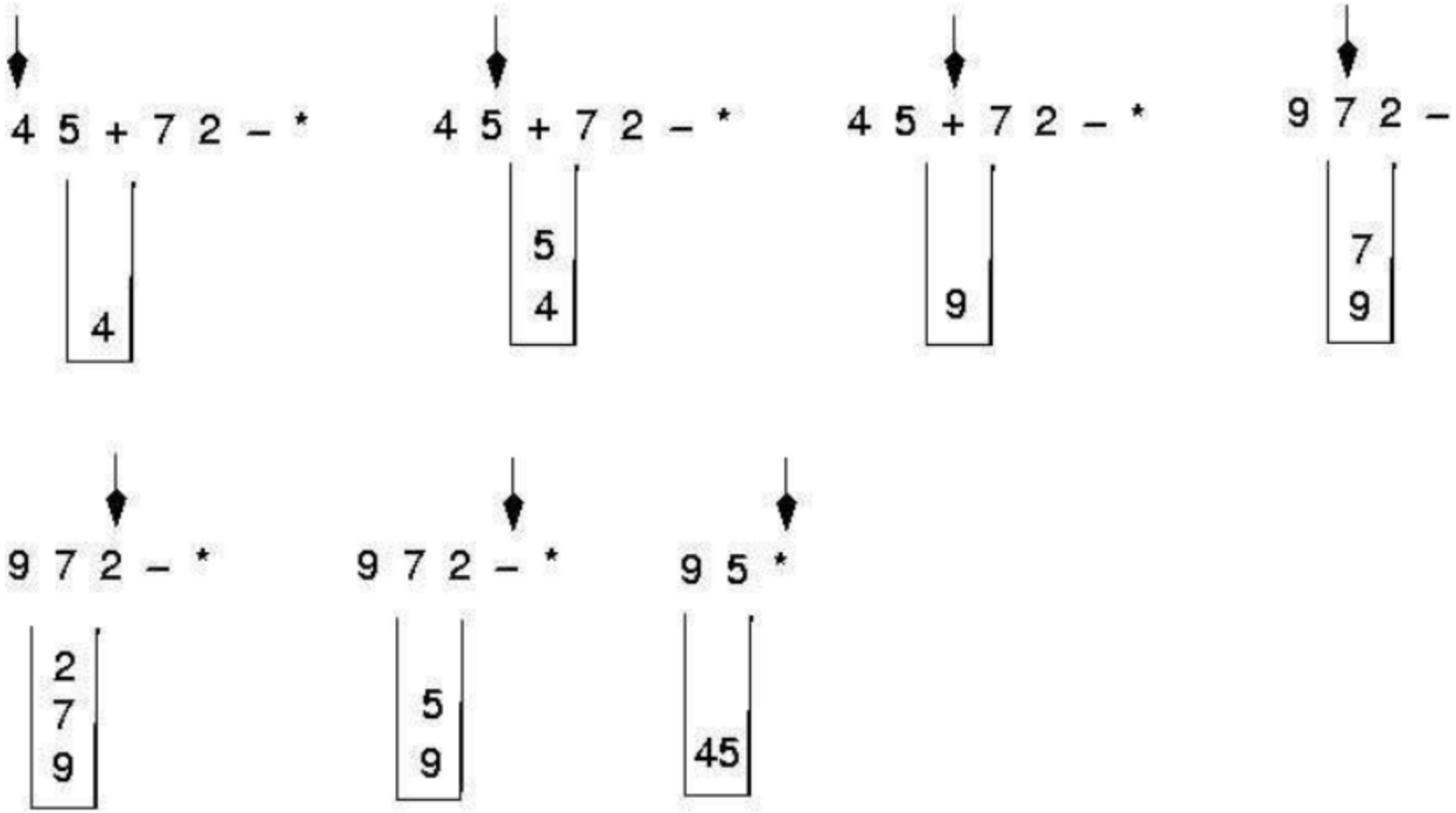
Evaluating a postfix expression

- Initialise an empty stack
- While token remain in the input stream
  - Read next token
  - If token is a number, push it into the stack
  - Else, if token is an operator, pop top two tokens off the stack, apply the operator, and push the answer back into the stack
- Pop the answer off the stack.

Example: Postfix Expressions



Postfix expressions: Algorithm using stacks (cont.)



Algorithm for evaluating a postfix expression (Cond.)

```
WHILE more input items exist
{
        If symb is an operand
                then push (opndstk,symb)
        else //symbol is an operator
        {
                Opnd1=pop(opndstk);
                Opnd2=pop(opndnstk);
                Value = result of applying symb to opnd1 &
opnd2                                   Push(opndstk,value); }
//End of else
} // end while
Result = pop (opndstk);
```

Question : Evaluate the following expression in postfix : 623+-382/+*2^3+

Final answer is

- 49
- 51
- 52
- 7
- None of these

Evaluate- 623+-382/+*2^3+

Symbol	opnd1	opnd2	value	opndstk
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3

Evaluate- 623+-382/+*2^3+

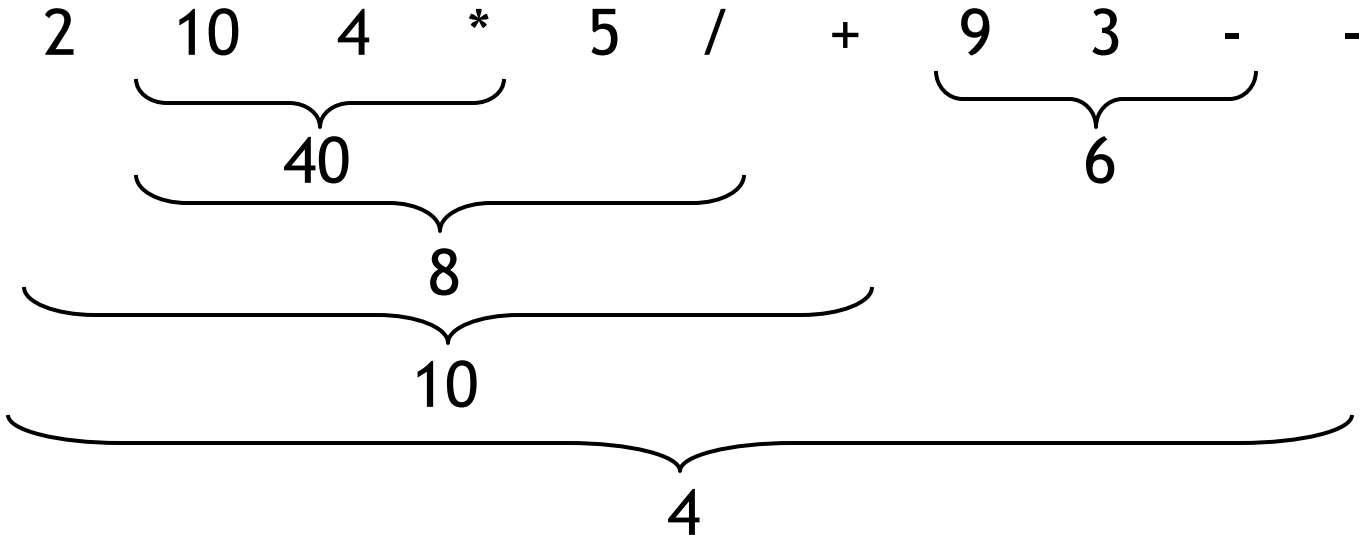
Symbol	opnd1	opnd2	value	opndstk
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
^	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

About postfix notation

- Postfix, or Reverse Polish Notation (RPN) is an alternative to the way we usually write arithmetic expressions (which is called infix, or algebraic notation
  - "Postfix" refers to the fact that the operator is *at the end*
  - "Infix" refers to the fact that the operator is *in between*
  - For example, 2 2 + postfix is the same as 2 + 2 infix
  - There is also a seldom-used prefix notation, similar to postfix
- Advantages of postfix:
  - You don't need rules of precedence
  - You don't need rules for right and left associativity
  - You don't need parentheses to override the above rules
- Advantages of infix:
  - You grew up with it
  - It's easier to see visually what is done first

How to evaluate postfix

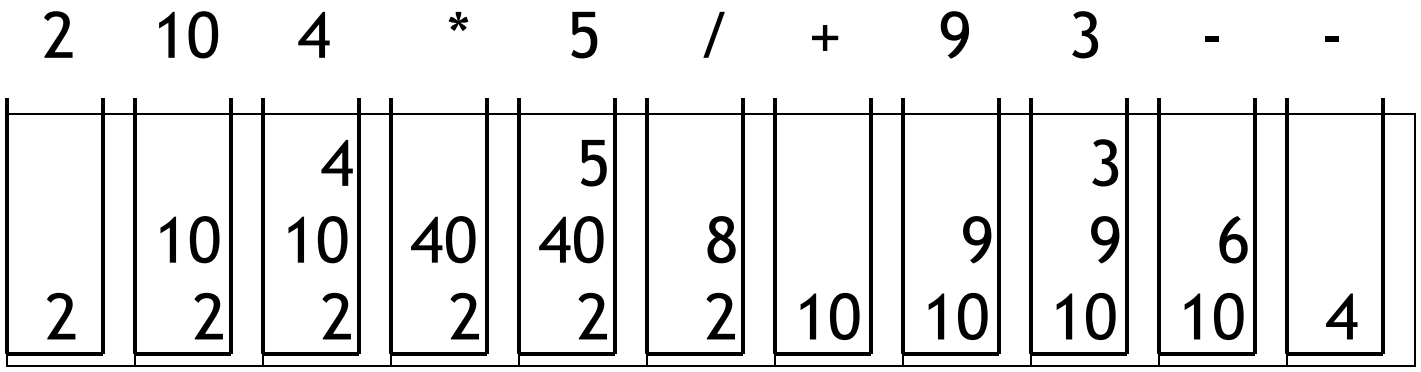
- Going from left to right, if you see an operator, apply it to the previous two operands (numbers)
- Example:



- Equivalent infix: 2 + 10 * 4 / 5 - (9 - 3)

Computer evaluation of postfix

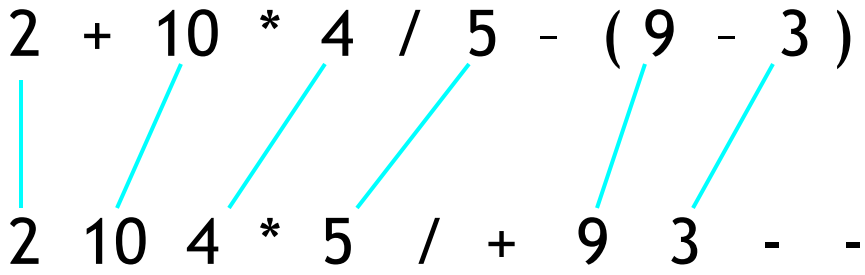
- Going left to right,
  - If you see a number, put it on the stack
  - If you see an operator, pop two numbers from the stack, do the operation, and put the result back on the stack
    - (The top number on the stack is the operand on the right)



- The result is the only thing on the stack when done
  - If there's more than one thing, there was an error in the expression

A helpful observation

- When you convert between infix and postfix, the oper*ands* (numbers) are always in the same order



- The oper*ators* are probably in a different order
- We're going to talk about how to convert manually between the two systems
- Your textbook has more information on how to do the conversion by computer

From infix to postfix

- Figure out, using the infix rules, which operation to perform next
- Write the new operand or operands in their correct places
- Write the operator at the end
- Postfix does *not* use parentheses, but we'll put them in temporarily to help show the way things are grouped
- Example: 2 + 10 * 4 / 5 - (9 - 3)
  - The multiply is done first: 10 4 *
  - Next, the divide: (10 4 *) 5 /
  - The addition: 2 (10 4 * 5 /) +
  - The rightmost subtraction: (2 10 4 * 5 / +) 9 3 -
  - The leftmost subtraction: (2 10 4 * 5 / +) (9 3 -) -
  - The final result: 2 10 4 * 5 / + 9 3 - -

From postfix to infix

- Why would you want to?
- OK--working from left to right, for each operator, move it between the two preceding operands, and put parenthesis around the whole group
- Example:

  2 10 4 * 5 / + 9 3 - -
  2 (10 * 4) 5 / + 9 3 - -
  2 ((10 * 4) / 5) + 9 3 - -
  (2 + ((10 * 4) / 5) 9 3 - -
  (2 + ((10 * 4) / 5) (9 - 3) -
  ((2 + ((10 * 4) / 5) - (9 - 3))

  – Now you can remove unnecessary parentheses (if you want to)