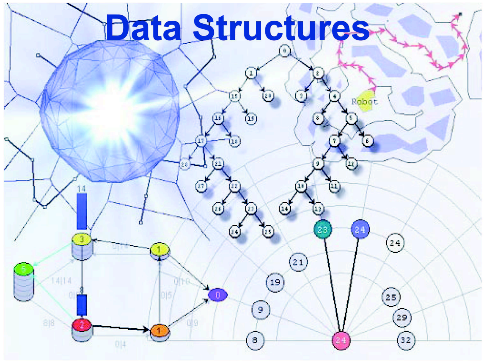
BBM 201
DATA STRUCTURES

Lecture 6:

EVALUATION of EXPRESSIONS



2019-2020 Fall

Evaluation of Expressions

- Compilers use stacks for the arithmetic and logical expressions.
- **Example:** x=a/b-c+d*e-a*c
- If a=4, b=c=2, d=e=3   what is x?
  - ((4/2)-2)+(3*3)-(4*2), ('/' and '*' have a priority)
- There may be also parenthesis, such as:
  - a/(b-c)+d*(e-a)*c
  - **How does the compiler solve this problem?**

Infix, prefix, postfix

- Normally, we use 'infix' notation for the arithmetic expressions:
  - Infix notation: a+b
- However, there is also 'prefix' and 'postfix' notation:
  - Prefix notation: +ab
  - Postfix notation: ab+

- Infix : 2+3*4
- Postfix: 234*+
- Prefix: +2*34

Prefix

$$+ \; 2 \; * \; 3 \; 5 =$$

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

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

$$* \; + \; 2 \; 3 \; 5 =$$

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

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

Postfix

$2\ 3\ 5\ *\ +\ =$

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

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

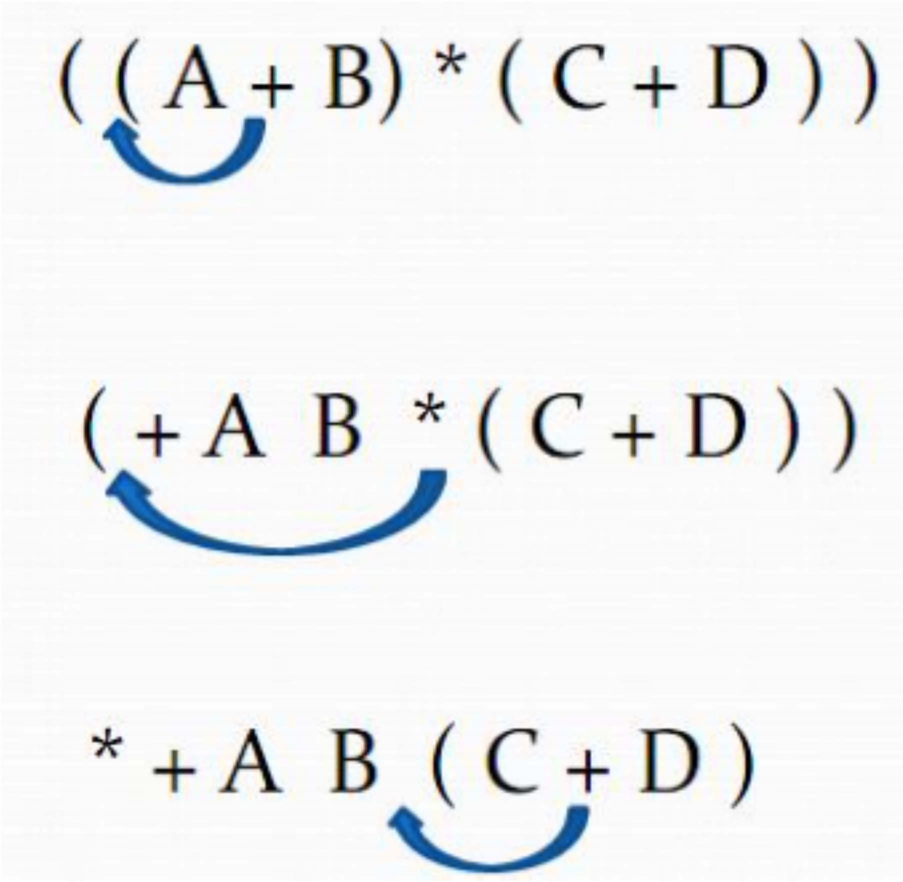
$2\ 3\ +\ 5\ *\ =$

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

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

How to convert infix to prefix?

Move each operator to the left of the operands:



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

(+ A B * (C + D))

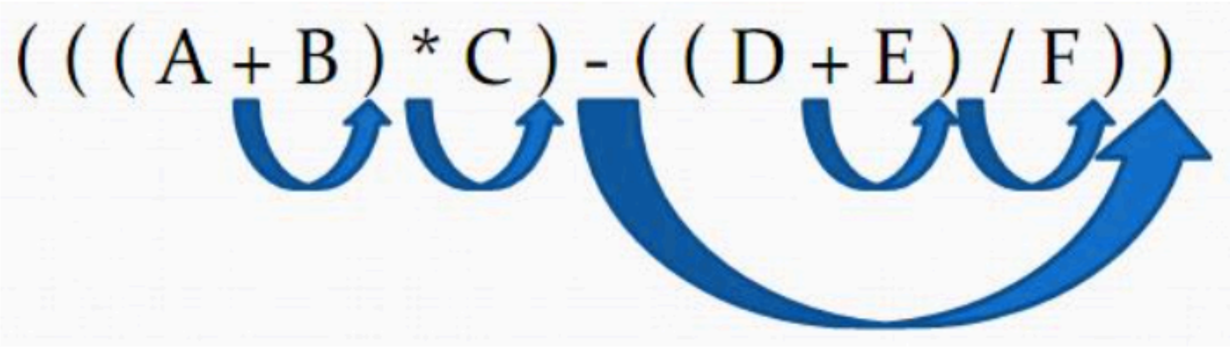
* + A B (C + D)

* + A B + C D

Operand order does not change!

How to convert infix to postfix?

Move each operator to the right of the operands:



- ((AB+* C) - ((D + E) / F))
- (AB+C* - ((D + E) / F))
- AB+C* ((D + E) / F)-
- AB+C* (DE+ / F)-
- A B + C * D E + F / -

Operand order still does not change!

Example - 1

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

- Postfix: ???
- Prefix: ???

Example – 1 (solution)

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

- Postfix: ab+c*de/-
- Prefix: -*+abc/de

Example - 2

- Infix: a/b-c+d*e-a*c

- Postfix: ???
- Prefix: ???

Example – 2 (solution)

- Infix: a/b-c+d*e-a*c

- Postfix: ab/c-de*+ac*-
- Prefix:-+-/abc*de*ac

Example – 3

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

- Postfix: ???
- Prefix: ???

Example – 3 (solution)

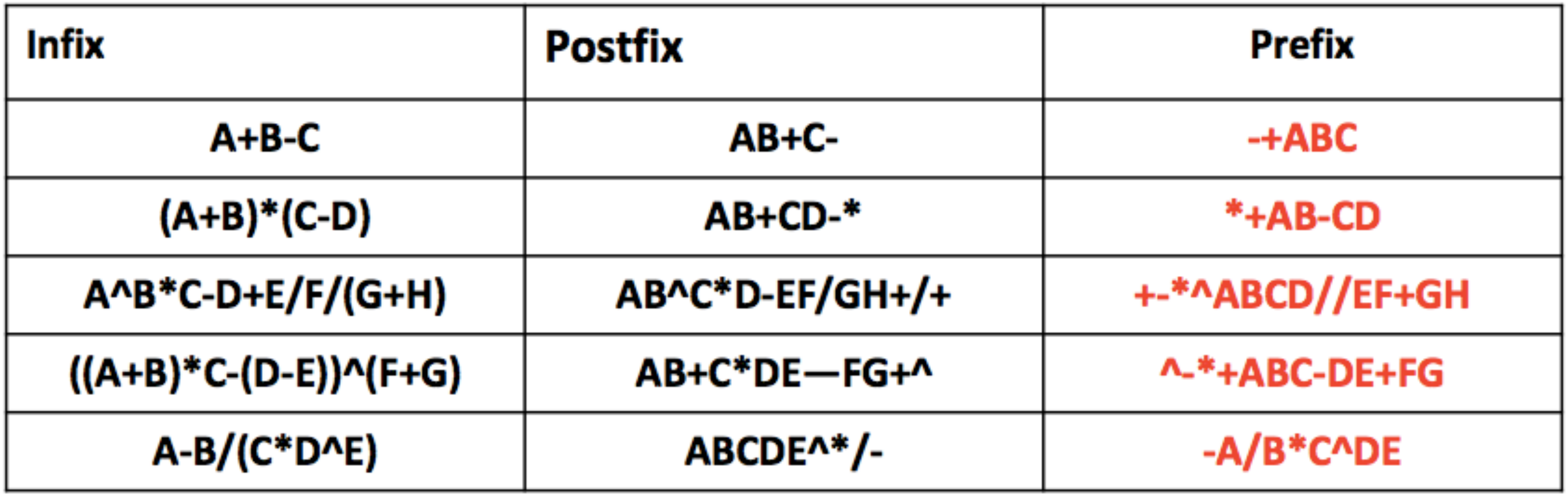
- Infix: (a/(b-c+d))*(e-a)*c

- Postfix: abc-d+/ea-*c*
- Prefix: **/a+-bcd-eac

Expressions

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

Infix, prefix, postfix



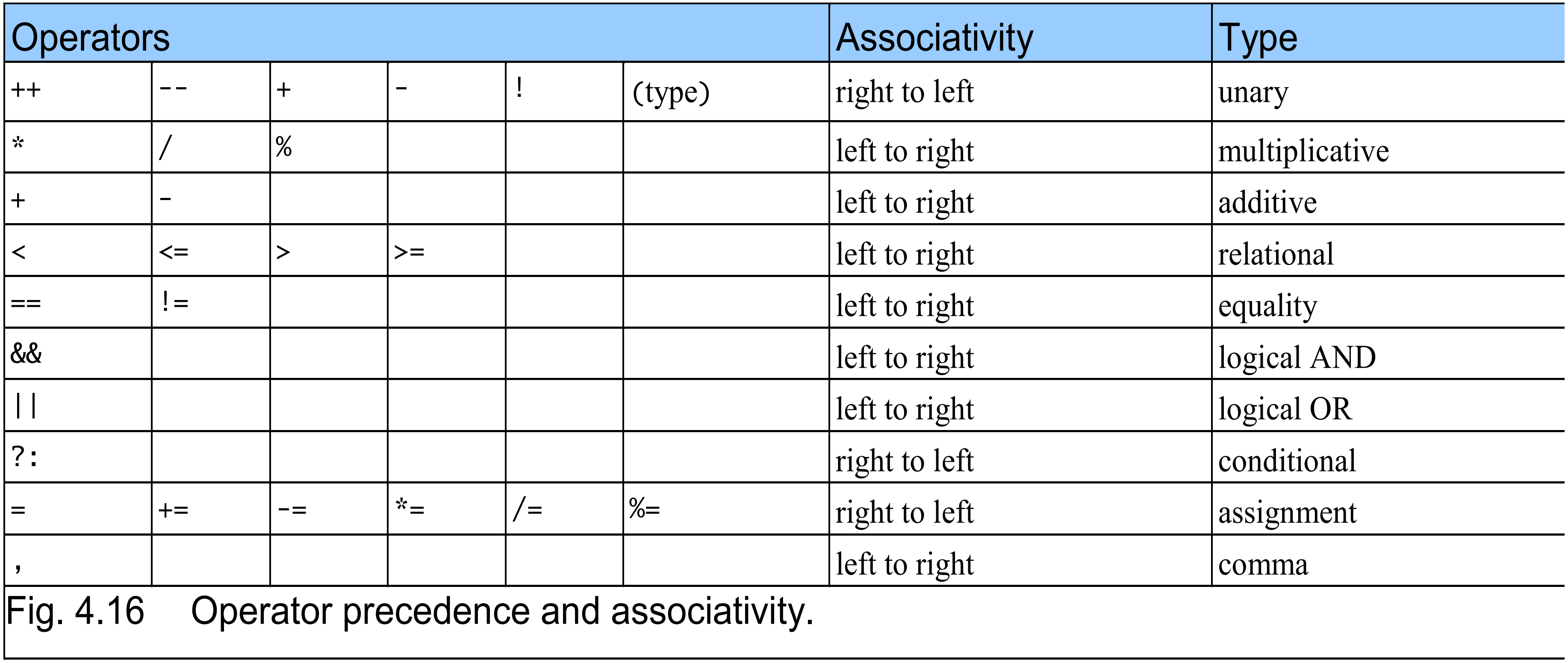
Infix	Postfix	Prefix
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD
A^B*C-D+E/F/(G+H)	AB^C*D-EF/GH+/+	+-*^ABCD//EF+GH
((A+B)*C-(D-E))^(F+G)	AB+C*DE—FG+^	^-*+ABC-DE+FG
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Why postfix?

- For the infix expressions we have two problems:
  - Parenthesis
  - Operation precedence

  - Example: ((4/2)-2)+(3*3)-(4*2)   (infix)
  - 42/2-33*+42*-   (postfix)

Operator PRECEDENCE



Operators						Associativity	Type
++	--	+	-	!	(type)	right to left	unary
*	/	%				left to right	multiplicative
+	-					left to right	additive
<	<=	>	>=			left to right	relational
==	!=					left to right	equality
&&						left to right	logical AND
||						left to right	logical OR
?:						right to left	conditional
=	+=	-=	*=	/=	%=	right to left	assignment
,						left to right	comma

Fig. 4.16 Operator precedence and associativity.

Parentheses are used to override precedence.

EVALUATION OF INFIX OPERATIONS

(fully Parenthesized)

```
1.  Read one input character -> c
2.  Action to follow based on the type of c
      Opening bracket       (2.1)  Push c into stack and then Go to step (1)
      Number                (2.2)  Push c into stack and then Go to step (1)
      Operator              (2.3)  Push c into stack and then Go to step (1)
      Closing bracket       (2.4)  Pop a character from stack -> b
                                   (2.4.1) if b is opening bracket
                                                Discard it,
                                                then Go to step (1)
                                   (2.4.2) else
                                                Pop op, then a, then p from stack
                                                p is the opening bracket, discard it
                                                Evaluate e = a op b
                                                Convert e to character
                                                Push e into the stack
                                                then Go to step (1)
      New line character    (2.5)  Pop e from stack and print e
                                   STOP
```

(((2 * 5) - (1 * 2)) / (11 - 9))

Input Symbol	*Stack (from bottom to top)*	*Operation*
(	(	Push Input
(	((	Push Input
(	(((	Push Input
2	(((2	Push Input
*	(((2 *	Push Input
5	(((2 * 5	Push Input
)	((10	**Pop 5 , Pop * , Pop 2 , Pop (, Do 2 * 5 = 10 , Push 10**
-	((10 -	Push Input
(	((10 - (	Push Input
1	((10 - (1	Push Input
*	((10 - (1 *	Push Input
2	((10 - (1 * 2	Push Input
)	((10 - 2	**Pop 2 , Pop * , Pop 1 , Pop (, Do 1 * 2 = 2 , Push 2**
)	(8	**Pop 2 , Pop - , Pop 10 , Pop (, Do 10 - 2 = 8 , Push 8**
/	(8 /	Push Input
(	(8 / (	Push Input
11	(8 / (11	Push Input
-	(8 / (11 -	Push Input
9	(8 / (11 - 9	Push Input
)	(8 / 2	**Pop 9 , Pop - , Pop 11 , Pop (, Do 11 - 9 = 2 , Push 2**
)	4	**Pop 2 , Pop / , Pop 8 , Pop (, Do 8 / 2 = 4 , Push 4**
New line	Empty	**Pop & Print 4**

EVALUATION OF INFIX OPERATIONS

(**Not** fully Parenthesized)

```
(1) Read an input character -> c
(2) Actions to follow based on the type of c
      Opening parenthesis   (2.1) Push c into character stack and Go to step (1)
      Number                (2.2) Push c into integer stack and Go to step (1)
      Operator              (2.3) Let d be top of the character stack
                                  (2.3.1)  If d is an operator of equal or higher priority,
                                                 Then process
                                                 (2.3.1.1) If the character stack is empty, Go to step (2.3.2)
                                                 (2.3.1.2) Else, Go to step (2.3)
                                  (2.3.2)  Else, Push c into the character stack and Go to step (1)
      Closing paranthesis   (2.4) Let d be top of the character stack
                                  (2.4.1)  If d=='(' then Pop from character stack and Go to step (1)
                                  (2.4.2)  Else, process
                                           Go to the step (2.4)
      New line character    (2.5)     If the character stack is not empty
                                      (2.5.1) Then process and Go to step (2.5)
                            (2.5.2)   Else, pop e from the integer stack, print e and STOP
```

```
process:   (1) Pop from character stack to op
           (2) Pop from integer stack to op2
           (3) Pop from integer stack to op1
           (4) Calculate op1 op op2 and Push the result into the integer stack
```

(2*5-1*2)/(11-9)

Input Symbol	Operation performed	Character Stack after Operation (left: bottom)	Integer Stack after Operation (left: bottom)
(	Push Input	(	
2	Push Input	(	2
*	Push Input	(*	
5	Push Input	(*	2 5
-	**since '-' < '*', we Process: 2 * 5 = 10 and Push the result**	(	10
	then Push '-'	(-	10
1	Push Input	(-	10 1
*	**Push * since * has higher priority than -**	(- *	10 1
2	Push Input	(- *	10 1 2
)	**Process: 1 * 2 = 2 and Push the result**	(-	10 2
	Process: 10 - 2 = 8 and Push the result	(	8
	Pop (		8
/	Push Input	/	8
(	Push Input	/ (	8
11	Push Input	/ (	8 11
-	Push Input	/ (-	8 11
9	Push Input	/ (-	8 11 9
)	**Process 11 - 9 = 2 and Push the result**	/	8 2
New line	**Process 8 / 2 = 4 and Push the result**		4
	Pop 4, Print the result		

Evaluation of a prefix operation

Input: / - * 2 5 * 1 2 - 11 9

Output: 4

Data structure requirement: a character stack and an integer stack

1. Read one character input at a time and keep pushing it into the character stack until the new line character is reached

2. Perform *pop* from the character stack. If the stack is empty, go to step (3)

Number (2.1) *Push* into the integer stack and then go to step (2)

Operator (2.2) Assign the operator to op

Pop a number from integer stack and assign it to op1

Pop another number from integer stack and assign it to op2

Calculate op1 op op2 and push the output into the int. stack.

Go to step (2)

3. *Pop* the result from the integer stack and display the result

/ - * 2 5 * 1 2 -11 9

Input	Operation	Character Stack (after)	Integer Stack (after)
/	Push to Char. Stack	/	
-	Push to Char. Stack	/-	
*	Push to Char. Stack	/ - *	
2	Push to Char. Stack	/ - * 2	
5	Push to Char. Stack	/ - * 2 5	
*	Push to Char. Stack	/ - * 2 5 *	
1	Push to Char. Stack	/ - * 2 5 * 1	
2	Push to Char. Stack	/ - * 2 5 * 1 2	
-	Push to Char. Stack	/ - * 2 5 * 1 2 -	
11	Push to Char. Stack	/ - * 2 5 * 1 2 - 11	
9	Push to Char. Stack	/ - * 2 5 * 1 2 - 11 9	
\n	Pop 9, Push 9 to Int. Stack	/ - * 2 5 * 1 2 - 11	9
	Pop 11, Push 11 to Int. Stack	/ - * 2 5 * 1 2 -	9 11
	Pop -, then 11 and 9, Do 11 - 9 = 2, Push 2 to Int. Stack	/ - * 2 5 * 1 2	2
	Pop 2, Push 2 to Int. Stack	/ - * 2 5 * 1	2 2
	Pop 1, Push 1 to Int. Stack	/ - * 2 5 *	2 2 1
	Pop *, then 1 and 2, Do 1 * 2 = 2, Push 2 to Int. Stack	/ - * 2 5	2 2
	Pop 5, Push 5 to Int. Stack	/ - * 2	2 2 5
	Pop 2, Push 2 to Int. Stack	/ - *	2 2 5 2
	Pop *, then 2 and 5, Do 2 * 5 = 10, Push 10 to Int. Stack	/ -	2 2 10
	Pop -, then 10 and 2, Do 10 - 2 = 8, Push 8 to Int. Stack	/	2 8
	Pop /, then 8 and 2, Do 8 / 2 = 4, Push 4 to Int. Stack	Stack is empty	4
	Print 4		Stack is empty

POSTFIX

Compilers typically use a parenthesis-free notation (postfix expression).

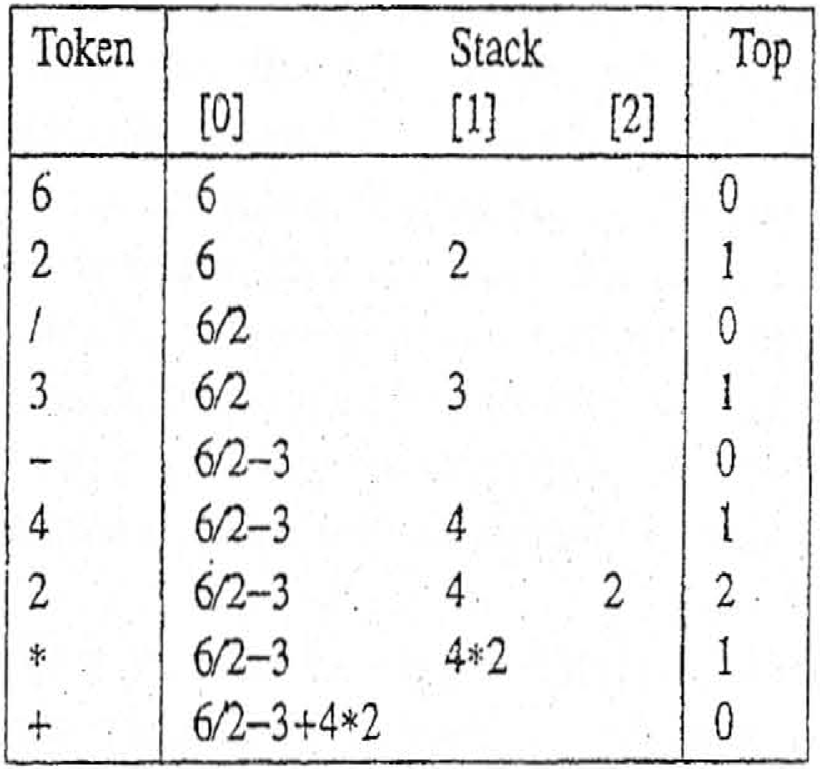
The expression is evaluated from the left to right using a stack:

- when encountering an operand: push it
- when encountering an operator: pop two operands, evaluate the result and push it.

Evaluation of a postfix expression

Token	Stack			Top
	[0]	[1]	[2]	
4	4			0
2	4	2		1
/	4/2			0
2	4/2	2		1
-	(4/2)-2			0
3	(4/2)-2	3		1
3	((4/2)-2)	3	3	2
*	((4/2)-2)	3*3		1
+	((4/2)-2)+(3*3)			0
4	((4/2)-2)+(3*3)	4		1
2	((4/2)-2)+(3*3)	4	2	2
*	((4/2)-2)+(3*3)	4*2		1
-	((4/2)-2)+(3*3)-(4*2)			0

6 2 / 3 – 4 2 * +



Token		Stack		Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
–	6/2–3			0
4	6/2–3	4		1
2	6/2–3	4	2	2
*	6/2–3	4*2		1
+	6/2–3+4*2			0

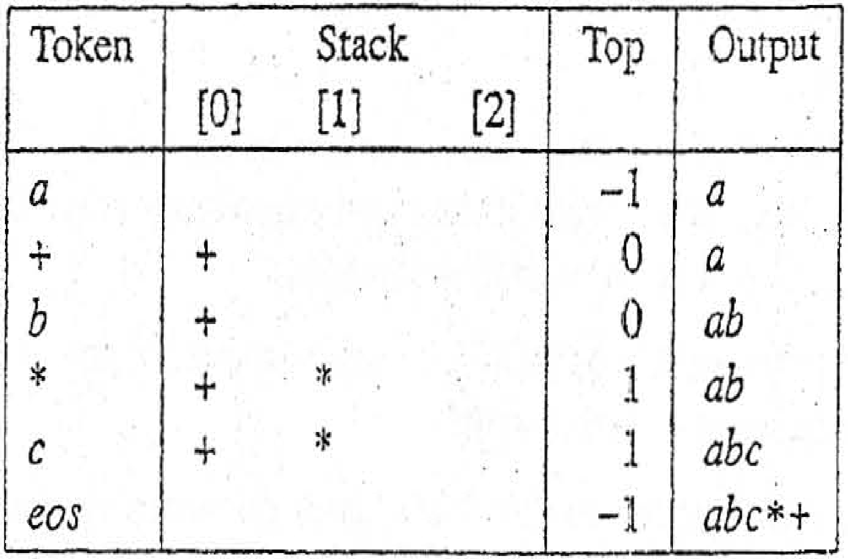
How to evaluate a postfix expression?

```
float eval(char* exp){
   float op1, op2;
   int i = 0;

   for (i = 0; exp[i]; i++) {  // Scan characters from left to right
      if (isdigit(exp[i]))     //  Number
         push(exp[i] – '0');   //   Push it to the stack
      else                     //  Operand
      {
         int val1 = pop();     //   Pop 2 numbers
         int val2 = pop();
         switch (exp[i])       //   Evaluate and push
         {
            case '+': push(val2 + val1); break;
            case '-': push(val2 - val1); break;
            case '*': push(val2 * val1); break;
            case '/': push(val2 / val1); break;
         }
      }
   return pop();
}
```

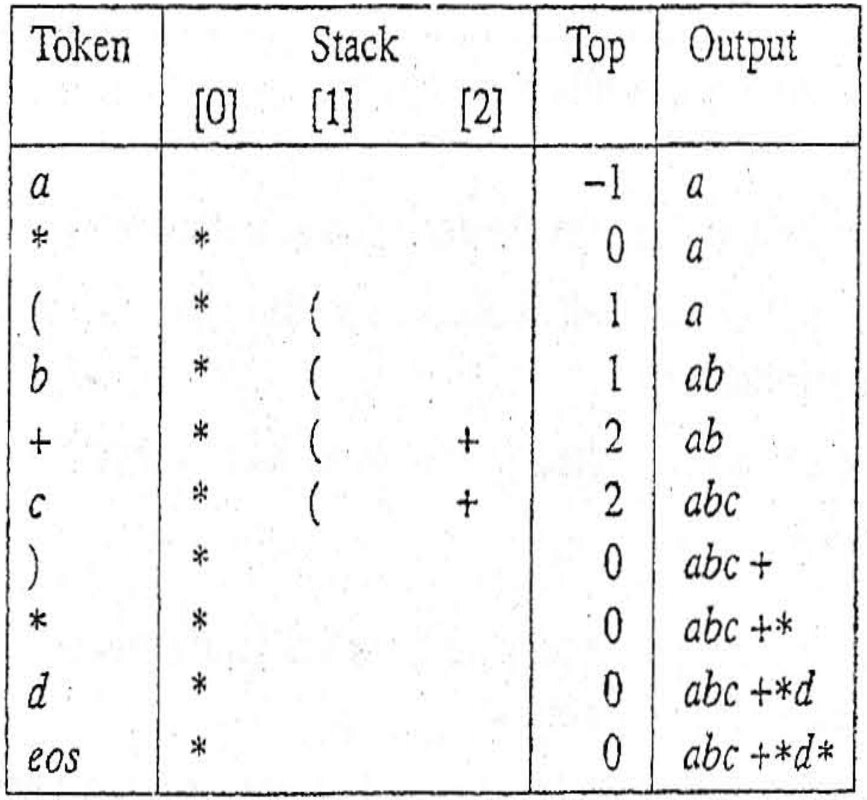
CONVERT an INFIX to POSTFIX

a+b*c



Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

a*(b+c)*d



Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc +
*	*			0	abc +*
d	*			0	abc +*d
eos	*			0	abc +*d*

How to convert infix to postfix?

```
// to check if the input character
// is an operator or a '('
int isOperator(char input) {
   char* operators = "+-*^%/(";
   for (int i = 0; i < 7; i++)
      if (operators[i] == input)
         return 1;
   return 0;
}

// to check if the input character is an operand
int isOperand(char input) {
   return !isOperator(input) && input != ')';
}

// function to return precedence value
// if operator is present in stack
int inPrec(char input) {
   switch (input) {
   case '+': case '-':
      return 2;
   case '*': case '%': case '/':
      return 4;
   case '^':
      return 5;
   case '(':
      return 0;
   }
}
```

```
// function to return precedence value
// if operator is present outside stack.
int outPrec(char input)
{
   switch (input) {
   case '+': case '-':
      return 1;
   case '*': case '%': case '/':
      return 3;
   case '^':
      return 6;
   case '(':
      return 100;
   }
}
```

How to convert infix to postfix?

```
void inToPost(char* input) {
   // while not EOS, iterate
   int i = 0;
   while (input[i] != '\0') {

      // if input is operand, then print
      if (isOperand(input[i]))
         printf("%c", input[i]);

      // If input is operator, then push
      else if (isOperator(input[i])) {
         if (isEmpty(s) ||
             outPrec(input[i]) > inPrec(top(s)))
            push(s, input[i]);
         else {
            while (!isEmpty(s) &&
                   outPrec(input[i]) <
                      inPrec(top(s))) {
               printf("%c", pop(s));
            }
            push(s, input[i]);
         }
      }
```

…continues on the right

```
      // condition for opening bracket
      else if (input[i] == ')') {
         while (top(s) != '(') {
            printf("%c", pop(s));

            // if opening bracket not present
            if (isEmpty(s)) {
               printf("Wrong input\n");
               exit(1);
            }
         }

         // pop the opening bracket.
         pop(s);
      }
      i++;
   }

   // pop the remaining operators
   while (!isEmpty(s)) {
      if (top(s) == '(') {
         printf("\n Wrong input\n");
         exit(1);
      }
      printf("%c", pop(s));
   }
} // end of inToPost
```

Exercise to do at home:

1. Write the code that converts infix to prefix.
2. Write the code that evaluates a prefix expression.