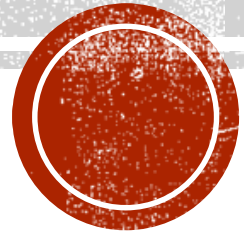
EVALUATION OF EXPRESSIONS



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

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
2. Actions at end of each input

Opening brackets	(2.1) *Push* into stack and then Go to step (1)
Number	(2.2) *Push* into stack and then Go to step (1)
Operator	(2.3) *Push* into stack and then Go to step (1)
Closing brackets	(2.4) *Pop* from character stack
	(2.4.1) if it is opening bracket, then discard it, Go to step (1)
	(2.4.2) *Pop* is used four times
	The first popped element is assigned to op2
	The second popped element is assigned to op
	The third popped element is assigned to op1
	The fourth popped element is the remaining opening bracket, which can be discarded
	Evaluate op1 op op2
	Convert the result into character and *push* into the stack
	Go to step (2.4)
New line character	(2.5) *Pop* from stack and print the answer *STOP*

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

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

EVALUATION OF INFIX OPERATIONS (NOTFULLY PARENTHESIZED)

1. Read an input character
2. Actions that will be performed at the end of each input

Opening parentheses (2.1) *Push* it into character stack and then Go to step 1

Number (2.2) *Push* into integer stack, Go to step 1

Operator (2.3) Do the comparative priority check

(2.3.1) if the character stack's *top* contains an operator with equal or higher priority,

Then *pop* it into op *Pop* a number from integer stack into op2

Pop another number from integer stack into op1

Calculate op1 op op2 and

Push the result into the integer stack

Closing par. (2.4) *Pop* from the character stack

(2.4.1) if it is an opening parentheses, then discard it and Go to step 1

(2.4.2) To op, assign the popped element

Pop a number from integer stack and assign it op2

Pop another number from integer stack and assign it to op1

Calculate *op1 op op2* and push the result into the integer stack

Convert into character and *push* into stack

Go to the step (2.4)

New line character (2.5) Print the result after popping from the stack *STOP*

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

Input Symbol	Character Stack (from bottom to top)	Integer Stack (from bottom to top)	Operation performed
(	(		
*	(*		*Push* as * has higher priority
-	(*		Since '-' has less priority, we do 2 * 5 = 10
	(-	10	We push 10 and then push '-'
1	(-	10 1	
*	(- *	10 1	Push * as it has higher priority
2	(- *	10 1 2	
)	(-	10 2	Perform 1 * 2 = 2 and push it
2	(	2	
5	(*	2 5	
New line		4	Perform 8 / 2 = 4 and push it
		4	Print the output, which is 4
(	/ (	8	
/	/	8	
	(	8	Pop - and 10 - 2 = 8 and push, Pop (
-	/ (-	8 11	
11	/ (	8 11	
9	/ (-	8 11 9	
)	/	8 2	Perform 11 - 9 = 2 and push it

PREFIX

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* in to the integer stack and then go to step (1)
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 integer stack.
	Go to step (2)

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

/ - * 2 5 * 1 2 -11 9

/	/		
-	/-		
*	/ - *		
2	/ - * 2		
5	/ - * 2 5		
*	/ - * 2 5 *		
1	/ - * 2 5 * 1		
2	/ - * 2 5 * 1 2		
-	/ - * 2 5 * 1 2 -		
11	/ - * 2 5 * 1 2 - 11		
9	/ - * 2 5 * 1 2 - 11 9		
\n	/ - * 2 5 * 1 2 - 11	9	
	/ - * 2 5 * 1 2 -	9 11	
	/ - * 2 5 * 1 2	2	11 - 9 = 2
	/ - * 2 5 * 1	2 2	
	/ - * 2 5 *	2 2 1	
	/ - * 2 5	2 2	1 * 2 = 2
	/ - * 2	2 2 5	
	/ - *	2 2 5 2	
	/ -	2 2 10	5 * 2 = 10
	/	2 8	10 - 2 = 8
	Stack is empty	4	8 / 2 = 4
		Stack is empty	Print 4

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.

62/3-42*+

Token	Stack [0]	[1]	[2]	Top
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

ALGORITHM

CONVERT AN INFIX TO POSTFIX

a+b*c

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

a*(b+c)*d

Token	Stack [0]	[1]	[2]	Top	Output
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**