Lecture No.06
Infix to Postfix Conversion using Stack

CC-213 Data Structures
Department of Computer Science
University of the Punjab

Slides modified very slightly from the late Dr. Sohail Aslam’s lectures at VU

Use of Stack

- Example of use: prefix, infix, postfix expressions.
- Consider the expression A+B: we think of applying the *operator* “+” to the *operands* A and B.
- “+” is termed a *binary operator*: it takes two operands.
- Writing the sum as A+B is called the *infix* form of the expression.

Prefix, Infix, Postfix

- Two other ways of writing the expression are

+ A B	*prefix*
A B +	*postfix*

- The prefixes "pre" and "post" refer to the position of the operator with respect to the two operands.

Prefix, Infix, Postfix

- Consider the infix expression
  $$A + B * C$$
- We “know” that multiplication is done before addition.
- The expression is interpreted as
  $$A + (B * C)$$
- Multiplication has *precedence* over addition.

Prefix, Infix, Postfix

- Conversion to postfix

  A + (B * C)   infix form

Prefix, Infix, Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication

Prefix, Infix, Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication
A (B C *) +	convert addition

Prefix, Infix, Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication
A (B C *) +	convert addition
A B C * +	postfix form

Prefix, Infix, Postfix

- Conversion to postfix

  (A + B) * C   infix form

Prefix, Infix, Postfix

- Conversion to postfix

(A + B) * C	infix form
(A B +) * C	convert addition

Prefix, Infix, Postfix

- Conversion to postfix

(A + B) * C	infix form
(A B +) * C	convert addition
(A B +) C *	convert multiplication

Prefix, Infix, Postfix

- Conversion to postfix

(A + B) * C	infix form
(A B +) * C	convert addition
(A B +) C *	convert multiplication
A B + C *	postfix form

Precedence of Operators

- The five binary operators are: addition, subtraction, multiplication, division and exponentiation.
- The order of precedence is (highest to lowest)
- Exponentiation ↑
- Multiplication/division *, /
- Addition/subtraction +, -

Precedence of Operators

- For operators of same precedence, the left-to-right rule applies:

  A+B+C means (A+B)+C.

- For exponentiation, the right-to-left rule applies

  $A \uparrow B \uparrow C$ means $A \uparrow (B \uparrow C)$

Infix to Postfix

Infix	Postfix
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A ↑ B * C – D + E/F	A B ↑ C*D – E F/+

Infix to Postfix

Infix	Postfix
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A ↑ B * C – D + E/F	A B ↑ C*D – E F/+

Infix to Postfix

- Note that the postfix form an expression does not require parenthesis.
- Consider ‘4+3*5’ and ‘(4+3)*5’. The parenthesis are not needed in the first but they are necessary in the second.
- The postfix forms are:

4+3*5	435*+
(4+3)*5	43+5*

Evaluating Postfix

- Each operator in a postfix expression refers to the previous two operands.
- Each time we read an operand, we push it on a stack.
- When we reach an operator, we pop the two operands from the top of the stack, apply the operator and push the result back on the stack.

Evaluating Postfix

```
Stack s;
while( not end of input ) {
   e = get next element of input
   if( e is an operand )
    s.push( e );
   else {
    op2 = s.pop();
    op1 = s.pop();
    value = result of applying operator ‘e’ to op1 and op2;
    s.push( value );
   }
}
finalresult = s.pop();
```

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	**op1**	**op2**	**value**	**stack**
6			**6**	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6			6	
2			6,2	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	**op1**	**op2**	**value**	**stack**
6				**6**
2				**6,2**
3				**6,2,3**
+	**2**	**3**	**5**	**6,5**

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6					6
2					6,2
3					6,2,3
+		2	3	5	6,5
-	6	5	1	1	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6					6
2					6,2
3					6,2,3
+		2	3	5	6,5
-	6	5	1	1	
3		6	5	1	1,3

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6					6
2					6,2
3					6,2,3
+		2	3	5	6,5
-	6	5	1	1	
3		6	5	1	1,3
8		6	5	1	1,3,8

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6					6
2					6,2
3					6,2,3
+		2	3	5	6,5
-	6	5	1	1	
3		6	5	1	1,3
8		6	5	1	1,3,8
2		6	5	1	1,3,8,2

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6					6
2					6,2
3					6,2,3
+		2	3	5	6,5
-	6	5	1	1	
3		6	5	1	1,3
8		6	5	1	1,3,8
2		6	5	1	1,3,8,2
/	8	2	4	1,3,4	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6					6
2					6,2
3					6,2,3
+		2	3	5	6,5
-	6	5	1	1	
3		6	5	1	1,3
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

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6					6
2					6,2
3					6,2,3
+		2	3	5	6,5
-	6	5	1	1	
3		6	5	1	1,3
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	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6					6
2					6,2
3					6,2,3
+		2	3	5	6,5
-	6	5	1	1	
3		6	5	1	1,3
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

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6					6
2					6,2
3					6,2,3
+		2	3	5	6,5
-	6	5	1	1	
3		6	5	1	1,3
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	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6	6			
2	6,2			
3	6,2,3			
+	2	3	5	6,5
- 6	5	1	1	
3	6	5	1	1,3
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

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	**op1**	**op2**	**value**	**stack**
6	6			
2	6,2			
3	6,2,3			
+	2	3	5	6,5
- 6	5	1	1	
3	6	5	1	1,3
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

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6	6			
2	6,2			
3	6,2,3			
+	2	3	5	6,5
- 6	5	1	1	
3	6	5	1	1,3
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

Converting Infix to Postfix

- Consider the infix expressions ‘A+B*C’ and ‘ (A+B)*C’.
- The postfix versions are ‘ABC*+’ and ‘AB+C*’.
- The order of operands in postfix is the same as the infix.
- In scanning from left to right, the operand ‘A’ can be inserted into postfix expression.

Converting Infix to Postfix

- The ‘+’ cannot be inserted until its second operand has been scanned and inserted.
- The ‘+’ has to be stored away until its proper position is found.
- When ‘B’ is seen, it is immediately inserted into the postfix expression.
- Can the ‘+’ be inserted now? In the case of ‘A+B*C’ cannot because * has precedence.

Converting Infix to Postfix

- In case of ‘(A+B)*C’, the closing parenthesis indicates that ‘+’ must be performed first.
- Assume the existence of a function ‘prcd(op1,op2)’ where op1 and op2 are two operators.
- Prcd(op1,op2) returns TRUE if op1 has precedence over op2, FASLE otherwise.

Converting Infix to Postfix

- prcd(‘*’,’+’) is TRUE
- prcd(‘+’,’+’) is TRUE
- prcd(‘+’,’*’) is FALSE
- Here is the algorithm that converts infix expression to its postfix form.
- The infix expression is without parenthesis.

Converting Infix to Postfix

```
Stack s;
While( not end of input ) {
    c = next input character;
    if( c is an operand )
        add c to postfix string;
    else {
        while( !s.empty() && prcd(s.top(),c) ){
             op = s.pop();
             add op to the postfix string;
         }
         s.push( c );
    }
    while( !s.empty() ) {
        op = s.pop();
        add op to postfix string;
    }
```

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
	ABC *	+

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
	ABC *	+
	ABC * +	

Converting Infix to Postfix

- Handling parenthesis
- When an open parenthesis ‘(‘ is read, it must be pushed on the stack.
- This can be done by setting prcd(op,‘(‘) to be FALSE.
- Also, prcd(‘(‘,op) == FALSE which ensures that an operator after ‘(‘ is pushed on the stack.

Converting Infix to Postfix

- When a ‘)’ is read, all operators up to the first ‘(‘ must be popped and placed in the postfix string.
- To do this, prcd(op,’)’) == TRUE.
- Both the ‘(‘ and the ‘)’ must be discarded: prcd(‘(‘,’)’) == FALSE.
- Need to change line 11 of the algorithm.

Converting Infix to Postfix

```
if( s.empty()  ||  symb != ‘)’ )
      s.push( c );
else
      s.pop(); // discard the ‘(‘
```

prcd(‘(‘, op) = FALSE	for any operator
prcd(op, ‘(’) = FALSE	for any operator other than ‘(’
prcd(op, ‘)’) = TRUE	for any operator other than ‘(‘
prcd(‘)’, op) = error	for any operator.