Software Engineering 2 (C++)

CSY2006

[Expression Evaluation]

Application of Stacks - Evaluating Postfix Expressions

$$(5+9)*2+6*5$$

- An ordinary arithmetic expression like the above is called infix-expression -- binary operators appear in between their operands.

- The order of operations evaluation is determined by the precedence rules and parentheses.

- When an evaluation order is desired that is different from that provided by the precedence, parentheses are used to override precedence rules.

Application of Stacks - Evaluating Postfix Expressions (Cont'd)

- Expressions can also be represented using postfix notation - where an operator comes after its two operands or prefix notation – where an operator comes before its two operands.

- The advantage of postfix and prefix notations is that the order of operation evaluation is unique without the need for precedence rules or parentheses.

Infix Notation	Postfix (Reverse Polish) Notation	Prefix (Polish) Notation
16 / 2	16 2 /	/ 16 2
(2 + 14)* 5	2 14 + 5 *	* + 2 14 5
2 + 14 * 5	2 14 5 * +	+ * 2 14 5
(6 – 2) * (5 + 4)	6 2 - 5 4 + *	* - 6 2 + 5 4

Infix to Postfix conversion (manual)

- An Infix to Postfix manual conversion algorithm is:
  1. Completely parenthesize the infix expression according to order of priority you want.
  2. Move each operator to its corresponding **right** parenthesis.
  3. Remove all parentheses.

- Examples:

  3 + 4 * 5 → (3 + (4 * 5)) → 3 4 5 * +

a / b ^ c – d * e – a * c ^ 3 ^ 4

((a / (b ^ c)) – ((d * e) – (a * (c ^ (3 ^ 4)))))

a b c ^ / d e * a c 3 4 ^ ^ * - -

Infix to Prefix conversion (manual)

- An Infix to Prefix manual conversion algorithm is:
  1. Completely parenthesize the infix expression according to order of priority you want.
  2. Move each operator to its corresponding **left** parenthesis.
  3. Remove all parentheses.

- Examples:

3 + 4 * 5 → (3 + (4 * 5)) → + 3 * 4 5

a / b ^ c – d * e – a * c ^ 3 ^ 4

((a / (b ^ c)) – ((d * e) – (a * (c ^ (3 ^ 4)))))

- / a ^ b c - * d e * a ^ c ^ 3 4

Application of Stacks - Evaluating Postfix Expression (Cont'd)

- The following algorithm uses a stack to evaluate a postfix expressions.

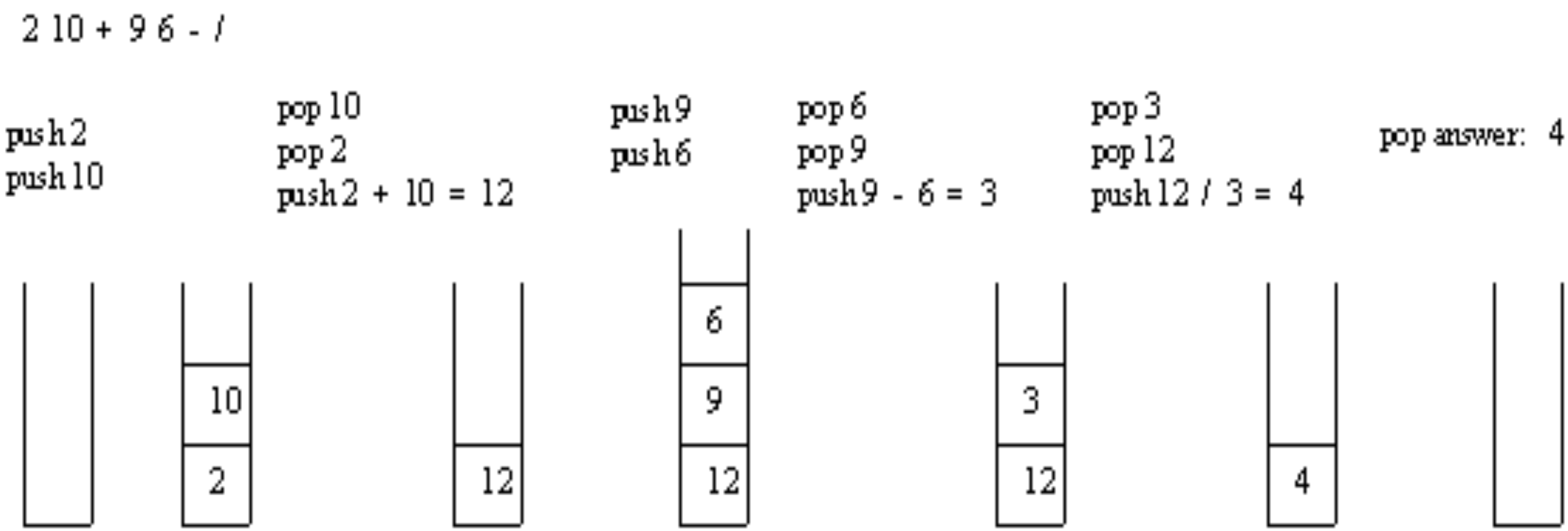
```
Start with an empty stack
for (each item in the expression) {
  if (the item is an operand)
       Push the operand onto the stack
  else if (the item is an operator  operatorX){
       Pop operand1 from the stack
       Pop operand2 from the stack
        result = operand2 operatorX operand1
       Push the result onto the stack
  }
}
```

Pop the only operand from the stack: this is the result of the evaluation

Application of Stacks - Evaluating Postfix Expression (Cont’d)

- Example: Consider the postfix expression, **2 10 + 9 6 - /**, which is **(2 + 10) / (9 - 6)** in infix, the result of which is 12 / 3 = 4.

- The following is a trace of the postfix evaluation algorithm for the postfix expression:



Application of Stacks – Infix to Postfix Conversion

```
postfixString = “”;
while(infixString has tokens){
    Get next token x;
    if(x is operand)
       Append x to postfixString;
   else if(x is ‘(’   )
       stack.push(x);
   else if(x is ‘)’  ){
        y = stack.pop();
        while(y is not ‘(’ ){
              Append y to postfixString;
              y = stack.pop();
        }
        discard both ‘( ‘ and ‘)’ ;
   }
   else if(x is operator){
             while(stack is not empty){
                  y = stack.getTop();        // top value is not removed from stack
                  if(y is ‘( ‘   ) break;
                  if(y has lower precedence than x) break;
                  if(y is right associative with equal precedence to x) break;
                  y = stack.pop();
                  Append y to postfixString;
             }
            push x;
  }
```

Application of Stacks – Infix to Postfix Conversion (cont' d)

```
while(stack is not empty){
        y = stack.pop( );
        Append y to postfixString;
}
```

step1:

Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7

Output : 1

step2:

Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7

-

Output : 1

step3:

Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7

-

Output : 1 2

step4:

Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7

^

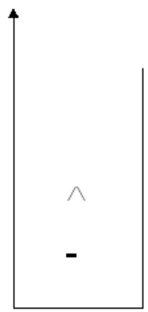
-

Output : 1 2

Application of Stacks – Infix to Postfix Conversion (cont’d)

step5:

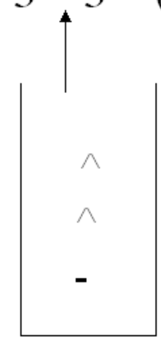
Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7



Output : 1 2 3

step6:

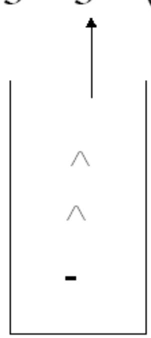
Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7



Output : 1 2 3

step7:

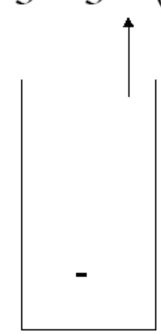
Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7



Output : 1 2 3 3

step8:

Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7

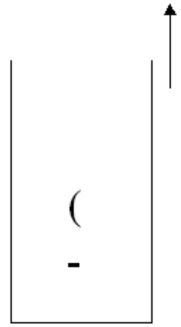


Output : 1 2 3 3 ^ ^ -

Application of Stacks – Infix to Postfix Conversion (cont’d)

step9:

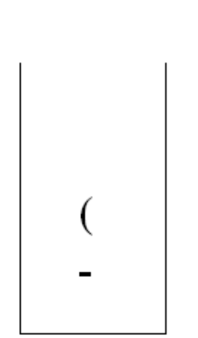
Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7



Output : 1 2 3 3 ^ ^ -

step10:

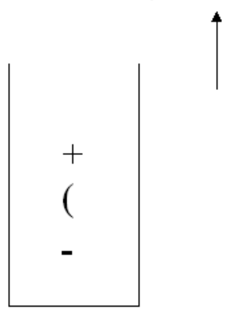
Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7



Output : 1 2 3 3 ^ ^ - 4

step11:

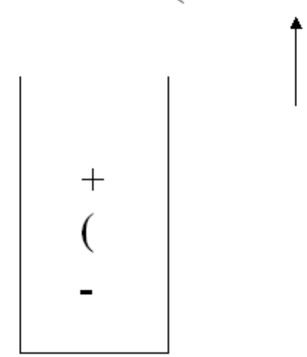
Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7



Output : 1 2 3 3 ^ ^ - 4

step12:

Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7



Output : 1 2 3 3 ^ ^ - 4 5

Application of Stacks – Infix to Postfix Conversion (cont' d)

step13:

Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7

*
+
(
-

Output : 1 2 3 3 ^ ^ - 4 5

step14:

Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7

*
+
(
-

Output : 1 2 3 3 ^ ^ - 4 5 6

step15:

Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7

-

Output : 1 2 3 3 ^ ^ - 4 5 6 * +

step16:

Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7

*
-

Output : 1 2 3 3 ^ ^ - 4 5 6 * +

Application of Stacks – Infix to Postfix Conversion (cont’d)

step17:

Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7

*
\-

Output : 1 2 3 3 ^ ^ - 4 5 6 * + 7

step18:

Input : 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) * 7

Output : 1 2 3 3 ^ ^ - 4 5 6 * + 7 * -

Application of Stacks – Infix to Prefix Conversion

An infix to prefix conversion algorithm:

1. Reverse the infix string
2. Perform the infix to postfix algorithm on the reversed string
3. Reverse the output postfix string

Example: (A + B) * (B – C)

reverse

(C – B) * (B + A)

Infix to postfix algorithm

C B - B A + *

reverse

* + A B - B C

NOTES

Two common techniques used to programmatically convert expressions from one form into another (infix/prefix/postfix).

1) USING STACKS
2) USING Expression TREES