Lab03 - Application of Stacks: Expression Evaluator

1 Assigment

Suppose we would like to evaluate the following arithmetic expression: "20+2*3+(2*8+5)*4". To evaluate this expression, we have to realize that * has higher precedence than +, and therefore, compute 2*3 and add it to 20 rather than computing 20+2 and multiplying it with 3. Additionally, we need to put parenthesis around (2*8+5) to specify that the result of this part will be multiplied by 4.

A typical evaluation sequence for the given infix expression would be the following:

(1) Store 20 in accumulator A1

(2) Compute 2*3 and store the result 6 in accumulator A2

(3) Compute A1+A2 and store the result 26 in A1

(4) Compute 2*8 and store the result in A2

(5) Compute 5+A2 and store the result 21 in A2

(6) Compute 4*A2 and store the result 84 in A2

(7) Compute A1+A2 and store the result 110 in A1

(8) Return the result, 110, stored in A1

We can write this sequence of operations as follows: "20 2 3 * + 2 8 * 5 + 4 * +". This notation is known as postfix or reverse polish notation.

Postfix notation unambiguously specifies the sequence of operations that will be performed to evaluate an expression without the need for parenthesis. Therefore, it is much easier to evaluate an expression specified in postfix notation.

It turns out, we can easily convert a well-formed infix expression to postfix notation using a stack. We will only consider the operators +, -, *, /, (,), and follow the usual precedence rules: * and / has the highest precedence, + and - come next, and (has the lowest precedence.

Here is a rough sketch of the algorithm:

(1) When an operand is encountered, output it

(2) When '(' is encountered, push it

(3) When ')' is encountered, pop all symbols off the stack until '(' is encountered

(4) When an operator, +, -, *, /, is encountered, pop symbols off the stack until you encounter a symbol that has lower priority

(5) Push the encountered operator to the stack

We start with an initially empty stack. When a operand is encountered it is immediately placed onto the output. When a left parenthesis, (, is encountered, it is placed onto the stack. If we see a right parenthesis, then we pop the stack, writing the symbols out until we encounter a corresponding left parenthesis, which is popped but not output. When an operator is encountered, then we pop entries from the stack until we find an operator of lower priority. That is, when a * or / is encountered, the stack is popped as long as we see * or /. Similarly, when + or - is encountered, the stack is popped as long as we see *, /, + or -, we stop only if the stack is empty or (is encountered.

Figure 1 shows the steps in converting the infix expression 20+2*3+(2*8+5)*4 to postfix (reverse polish) notation.

(1) The first symbol, 20, is placed on the output. Next, we encounter a '+', which is pushed to the stack. Then we encounter 2 which is also placed on the output.

(2) Next symbol is '*'. The top entry on the operator stack, '+', has lower precedence than '*', so nothing is output and '*' is put on the stack. Next symbol is 3, which is placed on the output.

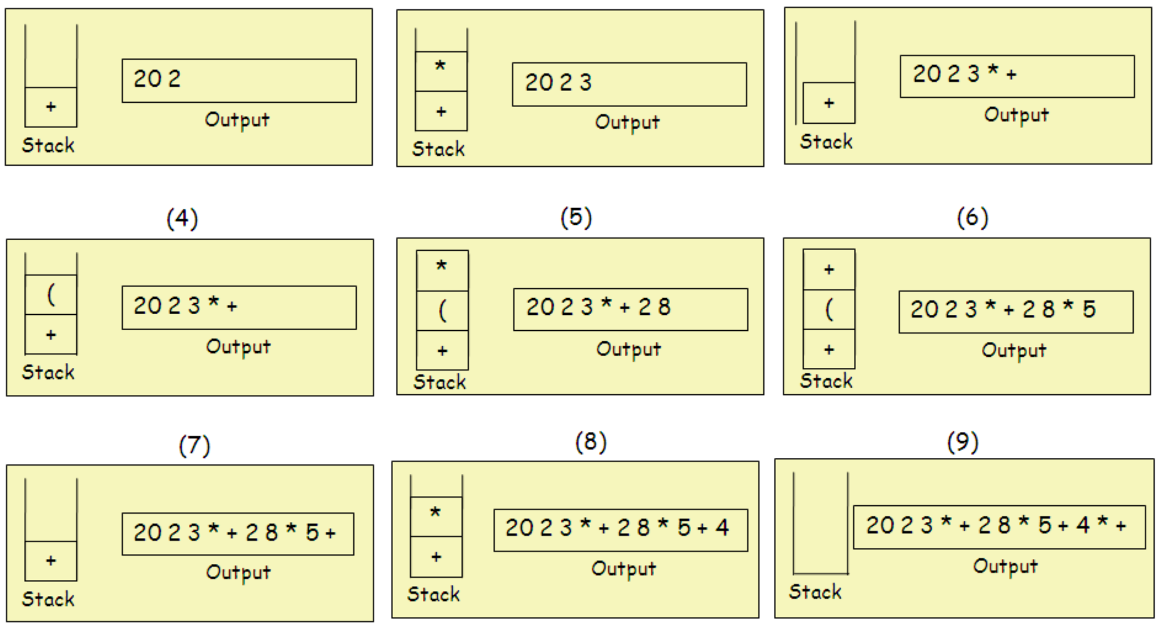

Figure 1: Steps in converting the infix expression 20+2*3+(2*8+5)*4 to postfix (reverse polish) notation.

(3) Next symbol is '+'. Checking the stack we find that we will pop '*' and place it on the output; pop the other '+', which is not of lower but equal priority, and place is on the output. Finally, we push the newly encountered '+' to the stack.

(4) The next symbol is a '(', which, being of lowest precedence, is placed on the stack.

(5) Next symbol is 2, which is placed on the output. Then we see a '*'. The top symbol of the stack is '(', so we place '*' on the stack. Then we see an 8 and place it on the output.

(6) Next symbol is '+'. Checking the stack we find that we will pop '*' and place it on the output. We will stop at '(' and push '+' to the stack. Then 5 is encountered and placed on the output.

(7) Now we encounter the closing parenthesis ')'. At this point we pop symbols off the stack until we see the opening parenthesis, ')'. All symbols except ')' are placed on the output.

(8) Next symbol is '*'. The top entry on the operator stack, '+', has lower precedence than '*', so nothing is output and '*' is put on the stack. Next 4 is encountered and placed on the output.

(9) All symbols on the expression has exhausted. As a final step, we pop all remaining symbols off the stack and place them on the output.

We can also use a stack to evaluate an expression specified in postfix notation. Here is a rough sketch of the algorithm:

(1) When an operand is encountered, push it to the stack

(2) When an operator is encountered, pop 2 operands off the stack, compute the result and push the result back to the stack

(3) When all symbols are exhausted, the result will be the last symbol in the stack

When a number is seen, it is pushed to the stack. When an operator is seen, the operator is applied to the two numbers that are popped off the stack, and the result is pushed onto the stack.

Figure 2 shows the steps in evaluation of the postfix expression: 20 2 3 * + 2 8 * 5 + 4 * +.

(1) We start by pushing numbers 20, 2 and 3 to the stack.

(2) Next we see a '*' operator. We pop 3 and 2 off the stack, compute 3*2, and push the result 6 onto the stack.

(3) Next we see a '+' operator. We pop 6 and 20 off the stack, compute 6+20, and push the result 26 onto the stack.

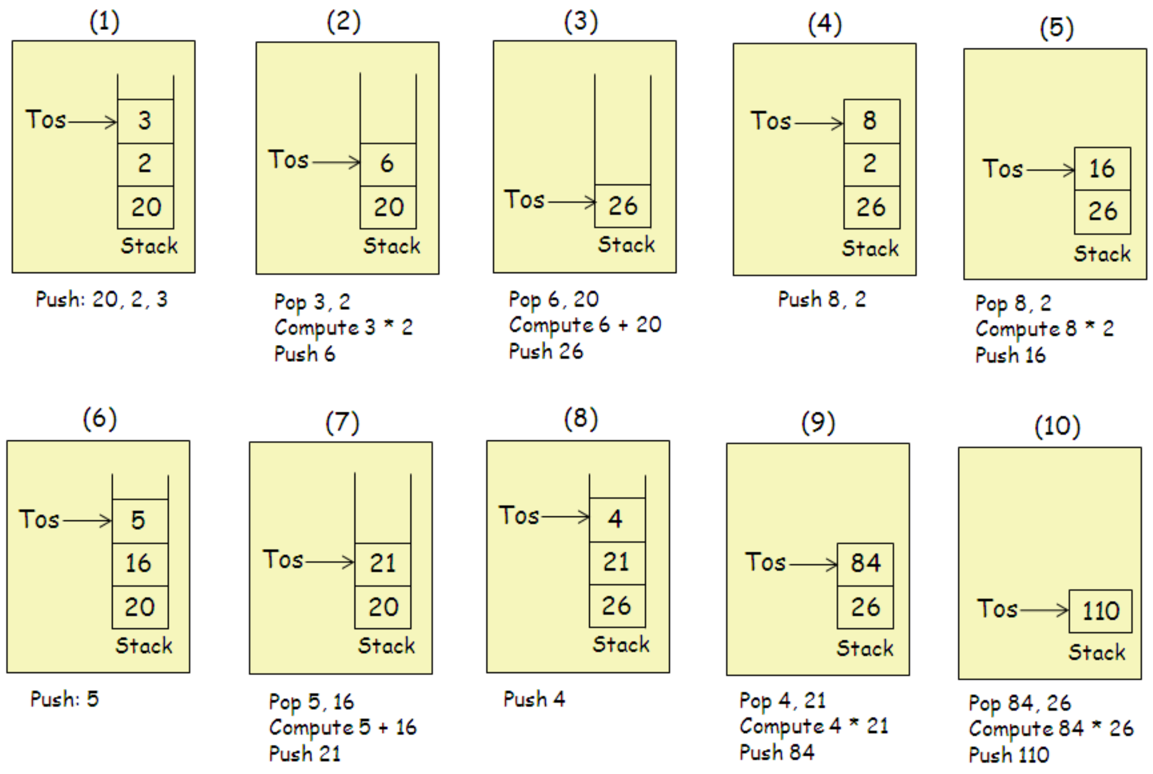

Figure 2: Steps in evaluating the postfix expression 20 2 3 * + 2 8 * 5 + 4 * +

(4) We then push numbers 2 and 8 to the stack.

(5) Next symbol is a '*' operator. We pop 8 and 2 off the stack, compute 8*2, and push the result 16 onto the stack.

(6) Now we encounter number 5 and push it to the stack.

(7) Next symbol is a '+' operator. We pop 5 and 16 off the stack, compute 5+16, and push the result 21 onto the stack.

(8) Now we encounter number 4 and push it to the stack.

(9) Next symbol is a '*' operator. We pop 4 and 21 off the stack, compute 4+21, and push the result 84 onto the stack.

(10) Finally, we see a '+' operator. We pop 84 and 26 off the stack, compute 84+26, and push the result 110 onto the stack. Since the expression is exhausted, we return the final result, 110.

2 Testing

Base code for the project is given on the Web site. We only care that you implement the two functions defined in Expression.cpp. You can use C++ stack class in your project. That is, you do not have to implement the stack class yourself.