ESO207: Data Structures and Algorithms

Programming Assignment *2* *Due: 14 September 2018 Midnight*

This programming assignment is to help you practice with the *stack* data structure. Irrespective of the language you are using, this assignment requires you to code up the stack data structure. Details about the problems and its possible algorithm are given after the basic problem description.

Problem 1. Write a program that takes an arithmetic expression given in *infix* (the usual) notation and prints it in *postfix* notation. The program should return an error if the input is illegal. The first part of this program allows only binary operators in the input infix expression whereas the second part of the program also allows the unary minus operand. In each case your program should correctly identify and return an error if the input infix expression is malformed. The precise syntax for the output postfix expression will be provided.

1. The input expression is made up of non-negative integers as the operands and standard *binary* operators. The binary operators correspond to the characters $\{+, -, *, /, \%, (,)\}$ and are defined in the standard way. (For e.g., $15/8 = 1$ and $15\%8 = 7$, etc.). For precedence, see below.
2. Additionally to the binary operators, the input expression may also contain the unary $-$ operator.

Problem 2. In this problem you are asked to evaluate a postfix expression and return the (integer) result.

Notes

Tokenizer

It might help to write a tokenizer module that parses the input. The tokenizer splits the input into tokens, where each token is either a number or an operator or a parenthesis. Define a token structure that has a *TkType* field and a *TkVal* field. The *TkType* field takes one of three values, *Operator*, *Operand* and *Parenthesis*– these values may be defined as global constants in the program. If the type of the token is *Operator*, then, the *TkVal* field can store the *opcode* (operator code) for that operator. For example, one may define the opcode of + as 1, opcode of − as 2, etc., as global constants in the program. If the type of the token is *Operand* then the *TkVal* field is the integer value of that operand. If the type of the token is *Parenthesis* then the *TkVal* field is either LEFT or RIGHT, which are pre-defined global constants. Note that it is the tokenizer that should correctly distinguish between the two operators, binary − and unary −.

Once the tokenizer is written, one can then focus on the problem at hand, which is to evaluate the expression. If operators have opcodes, contiguously numbered from say 1 to k, then one can define the precedence relation by keeping an array *Precedence* such that *Precedence*[*opcode*] gives the priority of the operator whose code is *opcode*. Similarly, we can keep an the *Associativity*

array such that *Associativity*[*opcode*] gives the associativity values LR or RL (for left-to-right or right-to-left respectively).

Infix to Postfix routine

We will now describe the classical way to transform infix to postfix. First, we give a precedence number to each operator. For now, assume that all operators are binary. The operator + and − are given the lowest precedence of 1, followed by ∗ whose precedence is 2, and then / and %, with precedence 3. Left and right parentheses are treated specially, and not as operators.

Precedence	Operator(s)
3	/, %
2	*
1	+, -

We will use the concept of *associativity*, that is, for operators at the *same* precedence, a left-to-right associativity means that the expression a op_1 b op_2 c is evaluated as (a op_1 b) op_2 c, that is, from left to right. If the associativity is right-to-left, then the parenthesization will be reversed, that is, the evaluation will be a op_1 (b op_2 c). (For e.g., the exponentiation operator $**$ associates from right to left). In the simple set of operators, we will assume left-to-right associativity for all operators that are at the same precedence level.

Decompose your input into tokens by writing a tokenizer (see below). Call *get_next_token* to get the next token from the input, which returns an operator or an operand or a parenthesis. Upon encountering end of input, the tokenizer returns NULL.

Keep a stack in which we will place only the operators. Initialize the stack to empty. Repeat the following procedure to get the next token process it until the NULL token is found.

1. If the token is an operand, output it.
2. If the token is an operator, do the following, in sequence. If at any time, the stack becomes empty or the top-of-stack is '(', push the current operator on top of stack and go to the next token.
   (a) While the precedence of the top-of-stack operator is more than the precedence of the current operator, pop the stack and print the operator.
   (b) If the precedence of the top-of-stack operator is the same as that of the current operator and the operators at this level of precedence associate left-to-right, then,
      i. pop the stack and print the operator.
      ii. push the current operator on the stack. Go to the next token.
   (c) Otherwise, if the precedence of the top-of-stack operator is the same as that of the current operator and the operators at this level of precedence associate right-to-left, then,
      i. push the operator on top of stack.

(d) Otherwise, if the precedence of the top-of-stack operator is lower than that of the current operator, then push the current operator on the stack.

3. If the token is '(', push it on stack.

4. If the token is ')', pop the stack repeatedly printing the operators until the first matching '(' is found. Pop this matching '('.

5. When the end of the input is reached, repeatedly pop the stack and print the operand until the stack is empty.

You may have noticed that it is assumed that all operators at a certain level of precedence either all associate left-to-right or all associate right-to-left.

Let us take an example

$$(10 + 5) - 4\% \ 15 \ / \ 7$$

Input	Stack	Output	Comments
$(10 + 5) - 4 \ \% \ 15/7$			Stack empty, No output
$10 + 5) - 4 \ \% \ 15/7$	(		Push '(' onto stack
$+5) - 4 \ \% \ 15/7$	(	10	Output 10
$5) - 4 \ \% \ 15/7$	(+	10	Push '+'
$) - 4 \ \% \ 15/7$	(+	10 5	Output 5
$-4 \ \% \ 15/7$		10 5+	')' seen, pop stack and print till matching '('
$4 \ \% \ 15/7$	-	10 5+	push '-'
$\% \ 15/7$	-	$10 \ 5 + 4$	output 4
$15/7$	$- \ \%$	$10 \ 5 + 4$	'%' has higher precedence than '-'
$/7$	$- \ \%$	$10 \ 5 + 4 \ 15$	output 15
7	$- \ /$	$10 \ 5 + 4 \ 15 \ \%$	'/' has same precedence as top-of-stack '%'
			'%' is popped and '/' is pushed
	$- \ /$	$10 \ 5 + 4 \ 15 \ \% \ 7$	output 7
		$10 \ 5 + 4 \ 15 \ \% \ 7 \ / \ -$	pop stack and print till empty

Evaluating Postfix expression

Evaluating a postfix expression is very direct. Here is an example. (all operators are binary)

$$10 \ 5 \ + \ 4 \ 15 \ 7 \ \%/-$$

Keep a stack. When you encounter an operand, push it onto stack. When you encounter an operator, pop the right number of its operands from the top of the stack, apply the operator and push the result back onto the stack. Finally, the result will be the only value on the stack.

Let us run this procedure on the above example. Stack grows to the right.

Current Input	Stack	Comment
10 5 + 4 15 7 %/−		stack empty
5 + 4 15 7 %/−	10	push 10
+ 4 15 7 %/−	10 5	push 5
4 15 7 %/−	15	pop 5, pop 10, Add $5 + 10 = 15$, push 15
15 7 %/−	15 4	push 4
7 %/−	15 4 15	push 15
%/−	15 4 15 7	push 7
/−	15 4 1	pop 7, pop 15, compute $15 \% 7 = 1$, push 1
−	15 4	pop 1 , pop 4, compute $4 / 1 = 4$, push 4
	11	pop 4, pop 15, compute $15 - 4 = 11$, push 11

The answer 11 is now on the top of the stack. Note the advantage of postfix evaluation, is that it does not need to consider about precedence or associativity.