**HW 5 Report**

**Kyle J Cooper**

**Examples:**

Below are two examples of my code, figure 1 and figure 2. Figure 1’s input is the same as that in the homework a\*(b+c), and Figure 2, is a much longer expression, just to show it works. The output shows multiple traversal types, prefix(preorder), postfix(postorder), and infix(inorder).

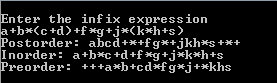
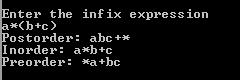


Figure 1 Figure 2

**Building the tree:**

For this program there are actually two main algorithms at work when building the expression tree.

Algorithm 1

The first algorithm takes the infix string given by the user, and converts it to a postfix expression. The reason for this is twofold. One, the conversion from infix to prefix takes care of the parentheses present in the infix expression. Two, the method I used to build the tree only works with a postfix expression and not an infix expression. This algorithm is inside my main function, and I comment in my code where it starts and stops. This algorithm is just a codification of the following rules.

“1. Print operands as they arrive.

2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.

3. If the incoming symbol is a left parenthesis, push it on the stack.

4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.

5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.

6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.

7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.

8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)” [Source](http://csis.pace.edu/~wolf/CS122/infix-postfix.htm)

Algorithm 2

This algorithm takes the postfix expression created in the first algorithm and turns it into a tree. It is important to note, that at this point the postfix expression does not exist as a postfix string like the user input infix string did. Instead the postfix expression is stored in an array of nodes, with each nodes data being the corresponding operand or operator that it would have in a postfix string, all nodes have been initialized so that they have left and right children equal to NULL.

Example:

The Infix string, a\*b, is ab\* in Postfix. So the postfix array would look something like

Array = [{data=a, left=NULL,right=NULL}, {data=b, left=NULL,right=NULL},{data=\*, left=NULL,right=NULL}]

To turn this array into a tree, we start at the first element in the array, and work our way down until we reach the first operator, ‘\*’ in the expression above, if we consider the current element to be our ith element, the i-1, element will be the right child of our current element/node, and the i-2 element will be the left child. Then the i-2 element is made equal to element i, and everything in the array is shifted toward the i-2 element by 2 elements. Another important thing about this algorithm is that after an operator is given children it is not treated as an operator by the algorithm or the rest of the time.

Example: a\*(b+c) is abc+\* in postfix

Start: abc+\*, search until first operator which is +.

a+l\*, +l = {data=x,left=b,right=c}, b and c are made children of + and + is ignored as an operator now.

\*l , a and +l are made children of \*.

By now all elements in the array but the first element are NULL, the first element now holds a pointer to the root of the tree which now looks like,

\*

/ \

A +

/ \

B C

**Tree Traversal:**

Traversing the tree is easy, and is based on the reclusive algorithms learned in class. For the preorder traversal we pass the root, we then output our current node, then recall the preorder function for the left subtree, then the right subtree, returning if the current node is NULL. This will reclusively print out all the data in the nodes. To switch to inorder or preorder we just have to which the order in which we visit nodes and print out the node data.

Example: for the above tree made from a\*(b+c) preorder traversal works like this.

Root is \*, \* is printed.

Output: \*

Call preorder(\*->left)

Root is a, a is printed.

Output: \*a

Left and right child of a are NULL so return.

Root is \* again, call preorder(\*->right)

Root is +, print +

Output: \*a+

Call preorder(+->left)

Root is b, b is printed.

Left and right child of b are NULL so return.

Root is + again, call preorder(+->right)

Root is c, print c

Output: \*a+bc

Left and right child of c are NULL so return.

Nothing left in + so return

Nothing left in \* so return/end

Final Output: \*a+bc

**Comments:**

The main problem I had starting out was figuring out how to handle the parentheses. I ended up using the stack method to convert from infix to postfix before making the tree, mostly because I had done something similar in another class, and it was the easiest way I could figure out to take care of the parentheses. From there I came up with the algorithm to build the expression tree from the postfix expression.