Converting a binary tree expression to infix notation using the BAIT algorithm

Emmanuel Attard Cassar
e-mail: emmanuel.attard-cassar@um.edu.mt

Abstract: When we evaluate an expression (be it mathematical, Boolean, or other) we have to deal with the precedence and associativity rules on operators (this is because the notation we use is infix). Following these rules an expression can be represented by means of a tree hierarchy and vice-versa an expression tree can be transformed into a linear infix expression.

To derive the linear infix expression from an expression-tree the algorithm given in textbooks consists of the use of the inorder traversal (IT) enriched with placing brackets that themselves impose the operator order expressed implicitly in the tree. The author devised the BA (Brackets Algorithm) which consists of adding brackets inside the terminal nodes of the tree (call this tree TB – Tree with Brackets). If IT is applied on TB (without any need of augmenting the code of IT) then the equivalent linear infix expression will be delivered.

Proof is given in the paper of the correctness of the BA. The BAIT algorithm consists of the application of the BA and the IT algorithms in succession.

Keywords: BAIT algorithm, linear infix expression, tree hierarchy, expression tree, inorder traversal, algorithms

Background to the Problem

As part of the Matsec syllabus in Data Structures (Computing A level) students are taught traversal methods on binary trees.[1] One of the applications of binary trees is binary tree

1 Matsec: https://www.um.edu.mt/__data/assets/pdf_file/0010/258877/AM07.pdf: pg. 17 [26/3/2016]

expressions.[2] A Binary Expression Tree (BET) can be converted into an equivalent infix expression, postfix expression, and prefix expression.[3] While the postfix and prefix expressions are directly obtained by means of the postorder and preorder traversals respectively, when applying the inorder traversal, brackets at the right places should be inserted to guarantee that the order of operations is kept as indicated in the tree.[4]

The example in figure 1 shows a BET.

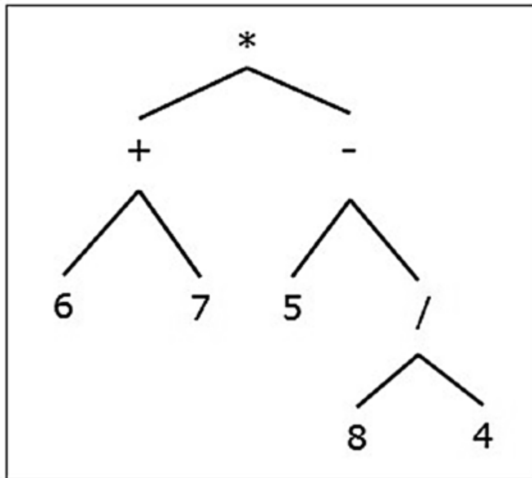

Figure 1

The postorder traversal will produce the correct postfix expression i.e. 6 7 + 5 8 4 / - *. The preorder traversal will produce the correct prefix expression i.e. * + 6 7 – 5 / 8 4. However, the inorder traversal produces 6 + 7 * 5 – 8 / 4. This is not truthful of the expression in the tree as this expression is implicitly indicating (by following the BIDMAS rule) that multiplication is performed before the addition while it is clear from the tree that multiplication should be performed after the addition.

The algorithm explained in a number of textbooks makes use of a modified inorder traversal enriched with placing brackets to abide by

2 Wikipedia: https://en.wikipedia.org/wiki/Binary_expression_tree: [26/3/2016]
3 Ibid.
4 Ibid.

the priority indicated in the tree.[5] The algorithm presented here, which is being called the BAIT (Brackets Algorithm and Inorder Traversal) algorithm, is divided into two parts. In the first part brackets are added to a BET and in the second part the inorder traversal is applied to the modified BET to produce the correct expression.

Definitions

A *right* terminal node is one which is placed to the right of its parent operator. Similarly, a *left* terminal node is one which is placed to the left of its parent operator.

An *uphill line* is a connection starting from a terminal node (a value) that visits parent-nodes in the same direction until no other parent node exists in that direction.

The *order* of an uphill line is the number of operators found on the uphill line. Figure 2 shows all the uphill lines, from left to right, of the BET in figure 1. They have orders 2, 1, 1, 1, and 3 respectively.

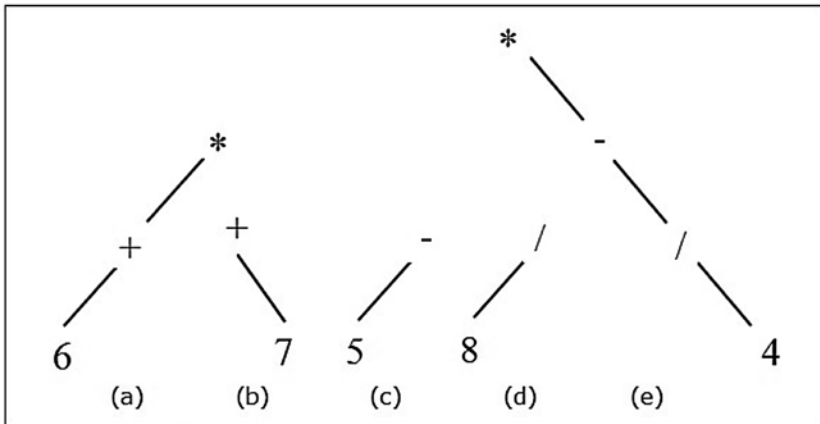

Figure 2

Assumption

I am assuming that operators are either binary or unary.

5 Ibid.

The Brackets Algorithm (BA)

The BA will add brackets to all values (terminal nodes). Left nodes will be supplemented with open-brackets to the left of the value. Right nodes will be supplemented with closed-brackets to the right of the value. Figure 3 shows the BA expressed in pseudocode.

```
begin
    for all operators (non-terminal nodes) do
    begin
        if operator is unary
            then add an empty node to make the operator binary
    end

    for all values (terminal nodes) do
    begin
        n = order of the uphill line from the node
        if (terminal node is ‘left’)
            then add n open-brackets to the left of the value
            else add n closed-brackets to the right of the value
    end
end
```

Figure 3

Conversion

To convert correctly a BET to infix notation perform the following two algorithms in sequence:

1. BA on BET (call the resulting tree BET-BA)
2. Inorder traversal on BET-BA

The resulting infix expression will be a truthful interpretation of the BET. I am calling the sequence of the two algorithms the BAIT algorithm.

Proof

The following abbreviations will be used.

- op: operator e.g. +, *
- lv: value (terminal) on the left of its parent node
- vr: value (terminal) on the right of its parent node
- expt: a tree expression e.g. the tree shown in figure 1 (note that expt can also be a subtree)
- exps: an expression written as a string e.g. ((6+7)*(5-(8/4)))

Consider the tree in figure 4. The infix notation that corresponds with such a BET is (lv op exps). Note that during the translation from BET to infix notation lv has acquired an open-bracket on its left.

Now consider the tree in figure 5. This can be translated in an infix notation in two stages starting from the bottom of the tree then moving upwards. These stages are shown in figure 6. Finally the resulting expression would be ((lv op1 exps1) op2 exps2). Note that lv has acquired two open-brackets on its left.

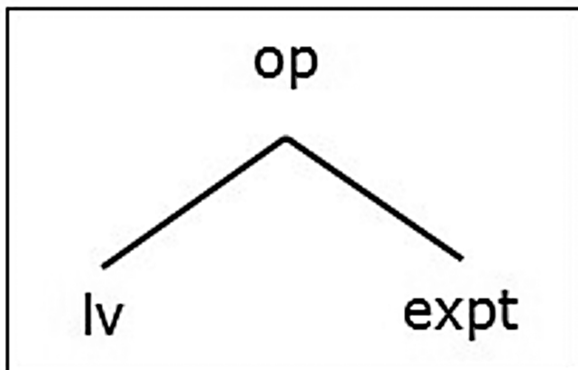

Figure 4

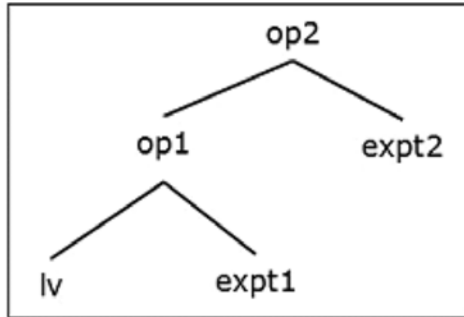

Figure 5

op2 op1 expt2 lv expt1	op2 (lv op1 exps1) expt2	((lv op1 exps1) op2 exps2)
	Translation of lowest level	Translation of final level

Figure 6

Let us now consider the tree in figure 7. The equivalent infix notation is (exps2 op2 (lv op1 exps)). This time lv has not acquired another open-bracket to its left.

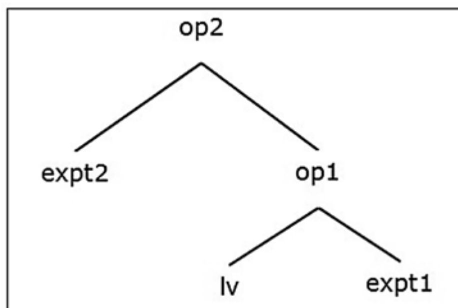

Figure 7

We can therefore conclude the following: 'A left value acquires an open-bracket to its left each time, during the bottom-up transformation it is carried up one level to the right.'

This can be stated as follows: 'After transforming an expression tree to a string expression, a left value will have as many open-brackets to its left as the order of its uphill line.' A similar result can easily be obtained for a right-value.

Substituting a Value with an Expression

Suppose we have a BET-BA and we want to substitute one of its values with another BET-BA. How would we do this? Consider the two BET-BAs found in figure 8.

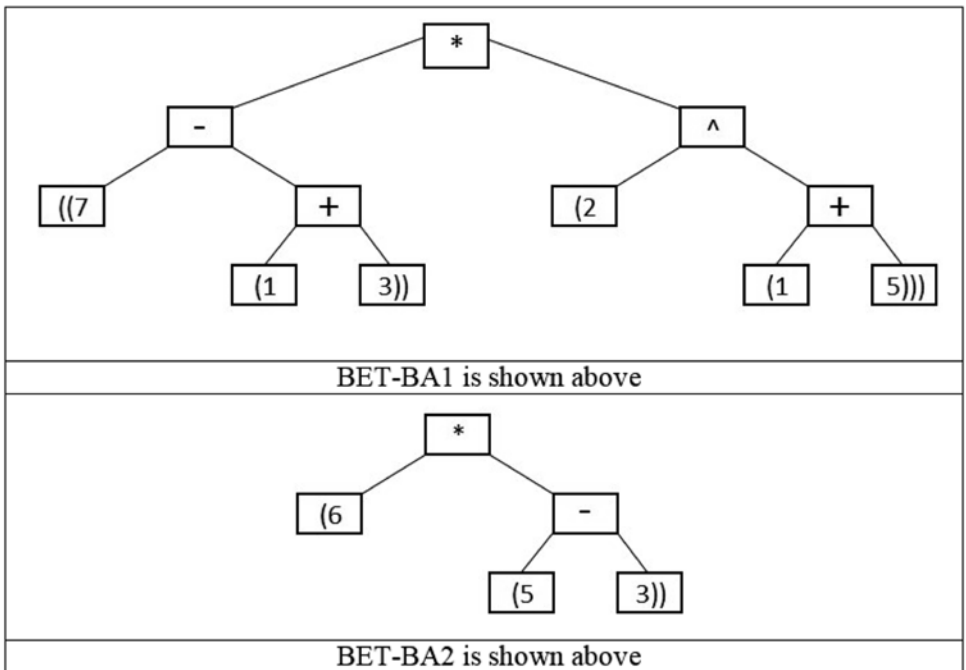

Figure 8

Let us assume that we would like to substitute a value from BET-BA1 with BET-BA2. In this case there is no need to remove all the brackets and re-work the brackets from the start. All one has to do is follow the algorithm found in figure 9. It is called BET-SUBST.

```
begin
   remove the value but not the brackets associated with it
   if the value is on a left node
      then  remove the brackets from the node and add them to the
            leftmost node of BET-BA2
      else  remove the brackets from the node and add them to the
            rightmost node of BET-BA2
end
```

Figure 9

If BET-BA2 of figure 8 substitutes the node with the value 3 in BET-BA1, then the result will be the one shown in figure 10.

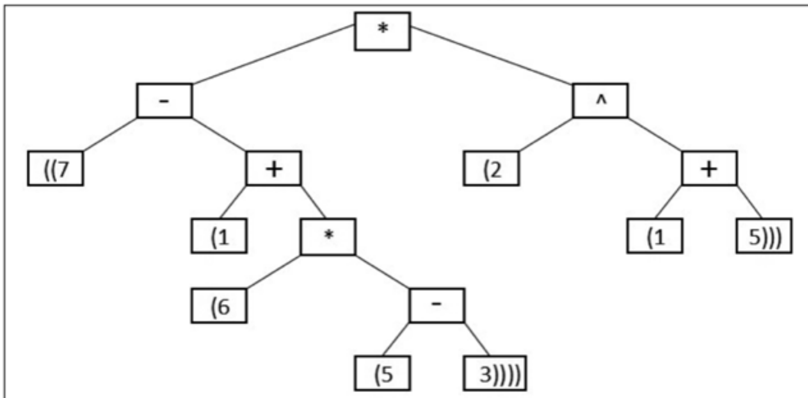

Figure 10

The validity of the algorithm can be shown in this way: Suppose that a particular BET-BA called BET-BAi contains a terminal node with the left value Val that will be substituted by BET-BAj. Refer to figure 11.

Let us say that the infix notation of BET-BAj is exprj. Then the new BET can be looked at as shown in figure 12.

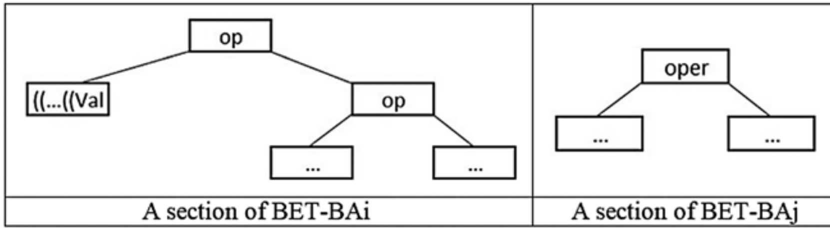

Figure 11

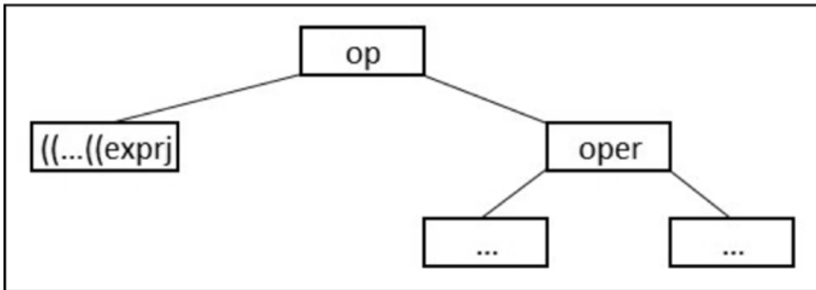

Figure 12

Now exprj must have a leftmost value and obviously the brackets that were previously associated with Val will now be attached to this leftmost value. A similar argument can be made for a value on a right node.

Substituting an Expression with a Value

This is the reverse process of the previous operation and its solution is expressed by means of the algorithm in figure 13.

```
begin
    eliminate the subtree
    in its place put the desired value
    use the BA to work out the brackets of this value
end
```

Figure 13

This can be proved by the following reasoning: When a subtree is eliminated the uphill lines of all the other terminals are not changed so the brackets associated with them remain the same. So the only the brackets that need to be calculated are the ones of the new value.

Joining Two Binary Expression Trees

Suppose two trees need to be joined. Then there is no need to re-calculate all the brackets of the terminal nodes. The simple algorithm shown in figure 14 will suffice.

```
begin
    join the trees
    add '(' to the leftmost node
    add ')' to the rightmost node
end
```

Figure 14

An example showing two BETs being joined is found in figure 15.

The justification for the algorithm shown in figure 14 is the following: If two BETs are joined with the operation op and their respective infix notations are inexp1 and inexp2 then the resulting infix notation of the adjoined BET is (inexp1 op inexp2). So an open-bracket is added at the

start of inexp1 and a closed-bracket is added at the end of inexp2. This means that an open-bracket is added to the leftmost value of inexp1 and a closed-bracket is added to the rightmost value of inexp2.

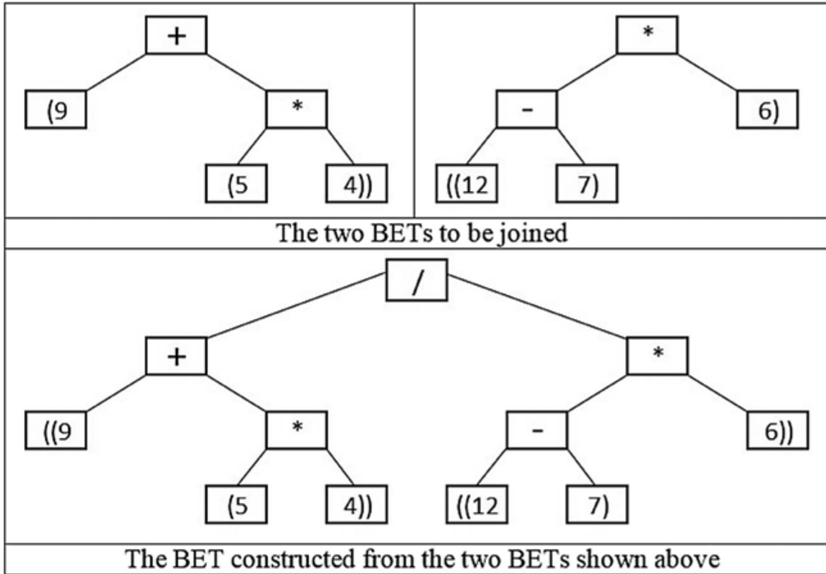

Figure 15

A Second Notation

After working on the above notation (uphill line, order, etc.) further tinkering with the formation of expression trees led to yet another notation for expressing the BAIT algorithm.

In this new notation the position of a node is represented as a sequence of Ls and Rs (e.g. LRRLL). L and R stand respectively for Left and Right. The sequence LRRLL represents the path taken starting from the root node (see figure 16). The root node is represented by the letter T.

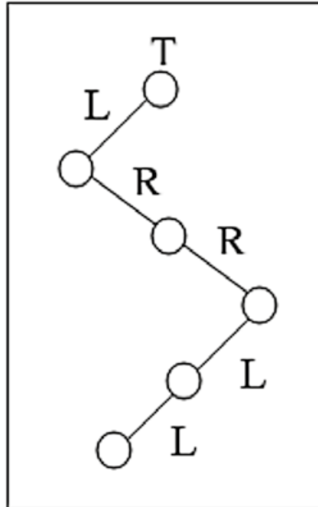

Figure 16

The ‘order’ of a terminal node can now be calculated by taking the last letter of the position of a node and then counting backwards as long as the letter is the same as the last letter. If it is not, then the counting stops. As examples the nodes in positions LRRLL, RRRLLRRR, and LRL would have respectively order of 2, 3, and 1.

Now the Bracket Algorithm can be expressed as shown in figure 17.

```
begin
    for all terminal nodes do
    begin
        n = order of node
        if last letter is L
            then add n open-brackets to the left of the value
            else add n closed-brackets to the right of the value
    end
end
```

Figure 17