Lecture Notes – Expression Trees / Parse Trees (etext section 7.6)

An expression is a sequence of tokens that follow prescribed rules.
An expression tree (or a parse tree) can be used to real world construction of sentences or math expressions.

Here is a parse tree representing a sentence: (from etext section 7.6)

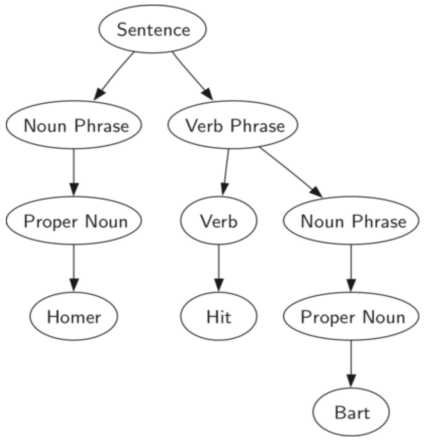
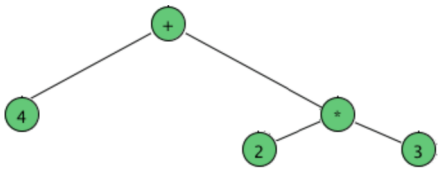


Figure 1: A Parse Tree for a Simple Sentence

(from cs.wcupa.edu/rkline)
A computer language is basically a context-free language. The symbols are tokens without any particular semantic meaning, namely, all numbers are the same, and even all literal (strings, numbers, etc) are regarded equally. All variables are regarded equally, etc. So the point is that we have a finite symbol set.
The first step of a compiler is to create a parse tree of the program, and the second phase is to assign meaning, or semantics to the entities in the tree. In reality, you create an *abstract syntax tree* of the program. For example, considering the parse tree for 4 + 2 * 3 above, an abstract syntax tree for this expression would look like this:



Let's consider an expression tree representing a math expression using binary operators (operators using 2 operands). The expression tree is a binary tree with the following properties:

1. each leaf is an operand
2. the root and internal nodes are operators
3. subtrees are subexpressions, with the root being an operator

Example from etext section 7.6:
We start with a fully parenthesized expression, such as: ((7+3)∗(5−2))

1. If the current token is a `'('`, add a new node as the left child of the current node, and descend to the left child.
2. If the current token is in the list `['+','-','/','*']`, set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.
3. If the current token is a number, set the root value of the current node to the number and return to the parent.
4. If the current token is a `')'`, go to the parent of the current node.

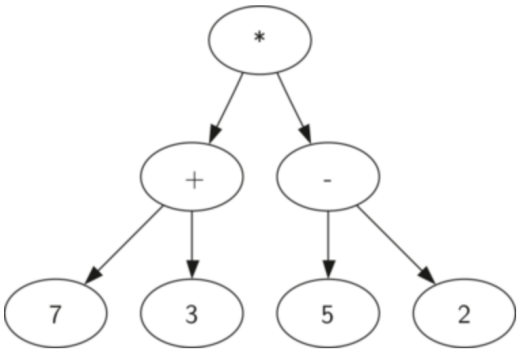
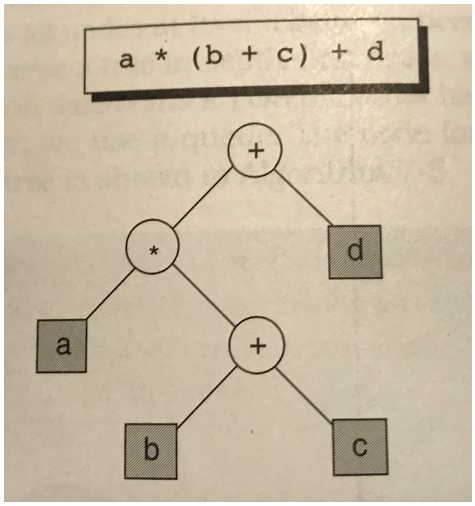
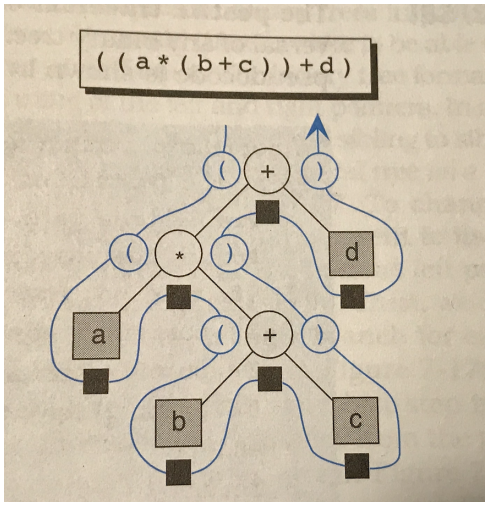


Figure 2: Parse Tree for ((7 + 3) ∗ (5 − 2))

Consider another example: (from Gilberg & Forouzan)





The 1st figure shows the math expression and the corresponding expression tree.

The 2nd figure shows the fully parenthesized expression and how the expression tree is generated following the steps described in page 1.

From the expression tree, we can generate the **infix notation** by:

```
def infix (tree):
        if tree != None:
                print(tree.getRootVal())
        else:
                print ( '(' )
                infix ( tree.getLeftChild() )
                print ( tree.getRootVal() )
                infix ( tree.getRightChild() )
                print ( ')' )
```

[The glitch was due to a typo in the code. I am leaving the above incorrect infix code and providing the corrected infix method below.]

```
def infix (tree):
        if tree != None:
                if not( tree.getRootVal() in ['+', '-', '*', '/']):  #token is an operand
                        print (tree.getRootVal())
                else:
                        print ( '(' )
                        infix ( tree.getLeftChild() )
                        print ( tree.getRootVal() )
```

```
            infix ( tree.getRightChild() )
            print ( ')' )
```

To derive the **prefix notation**:

```
def prefix (tree):
    If tree != None:
        print ( tree.getRootVal() )
        prefix (tree.getLeftChild() )
        prefix (tree.getRightChild() )
```

prefix expression: + * a + b c d

To derive the **postfix notation**:

```
def postfix (tree):
    if tree != None:
        postfix ( tree.getLeftChild() )
        postfix ( tree.getRightChild() )
        print ( tree.getRootVal() )
```

postfix expression: a b c + * d +

Note that parentheses are only needed for infix expression.