# CS417 Programming Assignment #9: FAQ

1. Q: How do I convert an infix expression into a tree?

   A: First convert it to postfix, and convert that into a tree.

---

2. Q: OK, Ok, so how do I convert a **postfix** expression into a tree?

   A: Use an algorithm similar to postfix evaluation:

```
make an empty stack
for each token in postfix expression,
    if token is an operand,
        create a Number node, or a Variable node
        push that node
    else (it's an operator)
        pop right operand node
        pop left operand node
        create an Operator node, with the two operand nodes
          as its children
        push the operator node
```

   Upon completion, the tree's root will be at the top of the stack. Pop it and return it.

---

3. Q: How do I evaluate an expression tree?

   A: Evaluate its root node. Everything will follow from that. To do this, each of your nodes should have a `get_value` method.

---

4. Q: How does the `get_value(symbol_table)` method work, for a `Number` node?

   A: Return the number.

---

5. Q: How does the `get_value(symbol_table)` method work, for a `Variable` node?

   A: Look up the name's value in the symbol table, and return that, unless the `Variable` node is a *left* operand (see the next question).

---

6. Q: How does the `get_value(symbol_table)` method work, for an `Operator` node?

   A: If the operator is `+`, `-`, `*`, or `/`, then:
   - call the get_value() method for both the left and right children.
   - add, subtract, multiply or divide these values.
   - return the result.

   However, if the operator is `=`, then:
   - call the `get_value()` method of the **right** child only.
   - get the name in the left child (it should be a `Variable` node).
   - set the variable's value in the symbol table.
   - return the right value.

---

7. Q: What should I do with the `parser417.py` module?

   A: The module includes these useful functions:
   - `tokenize` : break up a string into tokens
   - `lexer` : identify tokens, get a list of lexemes.
   - `to_postfix` : given an infix list of lexemes, return a postfix list of lexemes.
   - `eval_postfix` : evaluate a postfix list of lexemes
   - `main` : test the above functions (basically a solution to the previous assignment).

---

8. Q: What should I do with the `stack.py` module?

   A: The module implements a `Stack` class. It has `push`, `pop`, `top`, `len`, and `empty` methods.

---

9. Q: What should I do with the `expression_node.py` module?

   A: Add several classes to it. Each of these should be a sub-class of `Expression_Node`. You will need a class `Operator_Node`, a class `Number_Node`, and a class `Variable_Node`.

   Each of these classes will have its own `__str__` and `get_value` methods.

---

10. Q: How do I print a tree sideways?

    A: This will be covered in a later lab. Basically, run inorder traversal (but traverse backwards: do right, node, left, instead of left, node, right).

    As you do this, pass a depth variable to each recursive call. Use the depth to indent the output, when you visit the node.

---

11. Q: I'm stumped. How do I get started?

    A: Proceed as follows:

    Edit `expression_node.py`. It has one class in it: `Expression_Node`.

    Add three more classes: `Variable_Node`, `Number_Node`, and `Operator_Node`.

    Each of these classes will need these three methods:
    - `__init__`
    - `__str__`
    - `get_value`

    The methods are different, in each class. See the bottom of the assignment handout, which tells you what the `__str__` and `get_value` methods should do.

    What about the `__init__` methods? They will do different things, depending on what data is stored in each kind of node. For example, the `Number_Node` class should have this `__init__` method:

```
class Number_Node(Expression_Node):
    def __init__(self, value):
        self.number = value
```

    Now, in the `calculator2.py` module, you should implement `to_tree(postfix)`. See this FAQ for the pseudocode. There's a loop that goes through all the lexemes.

    For example, if the lexeme is a variable, then you need to push a `Variable_Node`:

```
if lexeme[1] == 'variable':
    name = lexeme[0]
    node = Variable_Node(name)
    s.push( node )
```

---

12. Q: I still need some hints. How do I evaluate an `Operator_Node`'s children?

    A: This should get you started:

```
class Operator_Node(Expression_Node):
    def __init__(self, op, l, r):
        ... here, create self.left, self.right, and
self.operator

    def get_value(self, symbol_table):
        if self.operator == '+':
```

```
                l_value = self.left.get_value(symbol_table)
                r_value = self.right.get_value(symbol_table)
                return l_value + r_value
            elif ...
```

---

13. Q: How do I create an `Operator_Node`?

    A: This will happen in `to_tree`:

```
    def to_tree(postfix_lexemes):
        ... make a stack ...
        for token, lex_type in postfix_lexemes:
            ...
            elif lex_type == 'operator':
                right = s.pop()
                left = s.pop()
                node = Operator_Node(token, left, right)
                s.push(node)
```