CS417 Programming Assignment 9

- Due on Friday, December 11th

NO LATE SUBMISSIONS WILL BE ACCEPTED!

Getting Started

Create a folder for your assignment, and download these three files to get started:

- `calculator2.py`
- `stack.py`
- `expression_node.py`
- `parser417.py` (has several useful functions)

Calculator

This assignment extends the calculator from the previous assignment, but uses expression trees.

The calculator should perform a read-evaluate-print loop, until end-of-file is reached, but rather differently from the previous assignment:

- get an infix expression from the input
- convert the expression to postfix
- **convert the postfix into a parse tree** (this is different)
- **print the tree recursively** (this is also different)
- **evaluate the tree** (also different)
- print the result

Implement this version of the calculator in a module called `calculator2.py`. Since this version uses trees, you should implement classes for the tree nodes (see below). The module `calculator2.py` should implement these functions:

- `to_tree(postfix_expression)`: this function takes a list of *lexemes*, in postfix order, and builds an expression tree. A *lexeme* is a `(token, token_type)` pair. The function should return *the root of the tree*. Example:

```
infix = "a = 1 +b * 4"
tokens = tokenize(infix)
# now, tokens == ['a', '=', '1', '+', 'b', '*', '4']
```

```
lexemes = lexer(tokens)
# lexemes == [('a', 'variable'), ('=', 'operator'), (1, 'number')
...]
postfix = to_postfix(lexemes)
# now, postfix == ['a', 1.0, 'b', 4.0, '*', '+', '=']
root = to_tree(postfix)
```

Here, `root` is the root of this tree:

```
      =
     / \
    a   +
       / \
      1   *
         / \
        b   4
```

- `print_tree(node)`: this function is passed the root of an expression tree, and prints it out "sideways", indenting each node to show its depth. For the tree above, the function should print the following result. Notice that successive printed rows visit the tree *right-to-left*, so you will have to modify the usual in-order traversal to get this result:

```
            4
        *
            b
    +
            1
=
    a
```

- `eval_tree(node, symbol_table)`: this function is passed the root of an expression tree, and returns a `float` (the tree's value). **This function may change the symbol table**. Example: if `node` is the root of the tree just shown, and the variable `b` has value `3`, then

```
eval_node(node, symbol_table)
```

should return 13, and also set the value of `a` to `13`, in the symbol table. Evaluation of this node should proceed recursively, as the following trace illustrates:

```
eval(= node) -> sets a to 13, and returns 13
  eval(a leaf) -> not evaluated (a is a name)
    eval(+ node) -> returns 1+12=13
      eval(1 leaf) -> returns 1
      eval(* node) -> returns 3*4=12
        eval(b leaf) -> looks up b in the table, and returns 3
```

```
        eval(4 leaf) -> returns 4
```

Tree Nodes

The expression tree is a binary tree, and all its nodes have two children (even leaf nodes, whose children are both `None`). However, different kinds of nodes have different behavior.

In this situation, it is very convenient to use class inheritance. I am providing a base class `Expression_Node`, but you should extend it with two subclasses: `Operator_Node` and `Operand_Node`.

The class `Operand_Node` will itself have two subclasses: `Variable_Node` and `Number_Node`.

The `Operand_Node` class should have `left` and `right` fields.

In addition, *all* classes should have these two methods:

- `__str__()`, which returns a printable version of the node.
- `get_value(symbol_table)`, which evaluates the nodes (in the case of an `Operator_Node`, it does so by first evaluating one or both of its two children).

Of course, each kind of node will have its own `__init__` method, and its own `__str__` method, and its `get_value` method will be different. Here is a summary:

Class	**`__str__`**	**`get_value`**
`Variable_Node`	variable's name	variable's value in symbol table
Number_Node	`str(number)`	number
Operator_Node	operator	value of left subtree, operated with value of right subtree, unless operator is `=`

Error Handling

This assignment will be tested with expressions that may contain errors. Such expressions should **not** crash your program. You must handle and report such errors. I am providing a sample session from my solution, showing error detection.

Turning in your work

When you are done, go to mycourses.unh.edu, and find CS417, and find the assignment. Click the "Submit" button and upload `calculator2.py`, `expression_node.py`, and **any other files** that you modified, or created, to run the program.