1. Problem Statement

An *interpreter* is a program that can execute instructions for a programming language. You have been using the `python3` interpreter this quarter to execute your Python programs. In this lab you will be writing an interpreter for a very simple language called **Derp**.

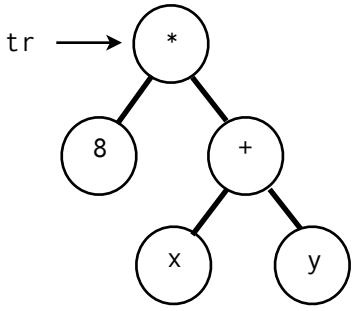
The Derp language processes *prefix* mathematical expressions into a tree structure called a *parse tree*. It works with the mathematical operators `*`, `//`, `+` and `-`. The supported operands are integer literals (e.g. `8`) and variables (e.g. `'x'`). A data structure called a *symbol table* is used by the interpreter to associate a variable with its integer value. When given a prefix expression, it displays the *infix* form of the expression and evaluates the result.

Consider the following example using the prefix expression: `* 8 + x y`

Symbol Table

Variable	Value
'x'	10
'y'	20
'z'	30

Parse Tree



Infix expression: `(8 * (x + y))`

Evaluation: `240`

Let's assume that the parse tree has already been constructed from the prefix expression. The prefix form of the expression can be re-obtained by doing a `preorder` (parent, left, right) traversal of the tree from the root. Since we haven't covered this type of traversal in lecture, here is the high level pseudocode for the traversal.

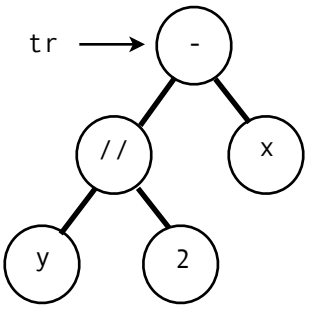
```
function preorder(node):
      if node is empty then return
      else:
            print node's value
            preorder(node's left child)
            preorder(node's right child)
```

Likewise, the infix expression can by constructed by doing an `inorder` (i.e. left, parent, right) traversal of the tree from the root. This is similar to `bstToString` from lecture.

We will leave the details of the symbol table, as well as the exercise of evaluating the parse tree to generate the result of the expression, for the implementation stage.

2. Problem Solving Session (20%)

You will be working in a team of four to six students as determined by your instructor. Each team will work together to complete a set of activities. Please complete the questions in order. Do not read ahead until you are finished with the first two questions.



1. Generate the **prefix expression** for the tree, tr, above.
2. Generate the **infix expression** for the tree, tr, above. Make sure you use parentheses to denote the order of operations

So far we have ignored how the parse tree is built. This is done by reading the individual tokens in the prefix expression and constructing the appropriate node types. Let's assume for this problem that we have the following node classes at our disposal:

Node class	Slot Name	Type	Maker Function
`MultiplyNode`	`left` `right`	`Node` `Node`	`mkMultiplyNode(left, right)`
`DivideNode`	`left` `right`	`Node` `Node`	`mkDivideNode(left, right)`
`AddNode`	`left` `right`	`Node` `Node`	`mkAddNode(left, right)`
`SubtractNode`	`left` `right`	`Node` `Node`	`mkSubtractNode(left, right)`
`LiteralNode`	`val`	`int`	`mkLiteralNode(val)`
`VariableNode`	`name`	`string`	`mkVariableNode(name)`

For example. the following code constructs the parse tree for the expression on the top of this page:

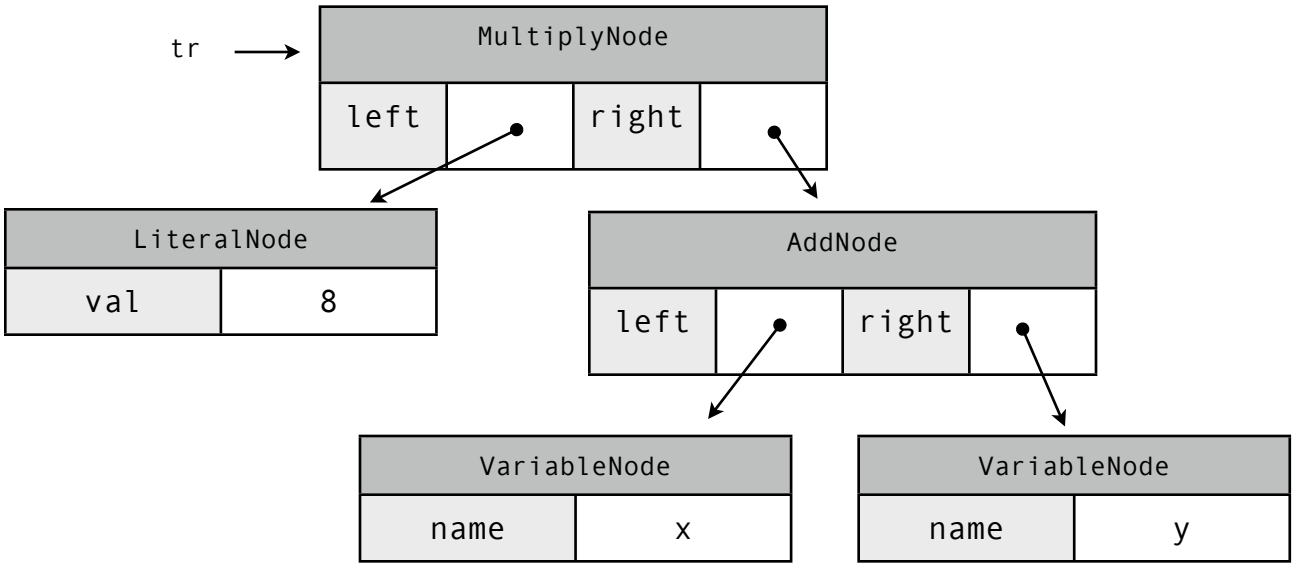
```
>>> tr = mkSubtractNode( \
            mkDivideNode(mkVariableNode('y'), mkLiteralNode(2)), \  # left
            mkVariableNode('x'))  # right
```

Now, let's assume that the prefix expression has already been converted into a list of string tokens, e.g.:

```
>>> prefixExp = "* 8 + x y"
>>> tokens = prefixExp.split()
>>> tokens
['*', '8', '+', 'x', 'y']
```

You will now explore how you can build and return a parse tree using the token list and the node definitions above.

This is a diagram which shows the node structure of the parse tree. The root of the tree, tr, is a `MultiplyNode`. Each node is indicated by its type (one of the 6 possible node classes), and its internal slot values are also shown. We use arrows to indicate the node has a child node. Here is the parse tree for the prefix expression, `* 8 + x y`, which is stored in the token list as `['*', '8', '+', 'x', 'y']`:



3. Given the prefix expression as the token list:

```
['*', '//', '-', 'x', '10', 'y', '+', '4', 'x']
```

   a.) Using the diagramming method above, show the parse tree for the expression.
   b.) Generate the infix expression for the tree.

4. Write the pseudocode for a function, `parse`. It should take the list of tokens for the prefix expression and return the root of the parse tree(as a collection of the node classes). The parse tree for the token list `['*', '8', '+', 'x', 'y']` should produce the image above.

3. Implementation (80%)

3.1. Dictionaries & Symbol Table

Recall that the symbol table is used to associate a variable name with its value (e.g. 'x' = 6). Python has a built-in data structure that is perfectly suited for this problem. It is called a **dictionary**. The details of the underlying implementation of this structure are left for the follow-up course. For now, you only need to know the syntax for using them.

```
>>> symTbl = dict()           # or symTbl = {}, creates an empty dictionary
>>> type(symTbl)
<class 'dict'>
```

An entry in a dictionary is composed of two parts - a unique key (e.g. the variable name 'x'), and its associated value (the integer value 6). Here are some basic dictionary operations:

```
>>> symTbl['x'] = 6           # insert the key 'x', with the value 6
>>> symTbl                    # python displays the dictionary as:
{'x': 6}                      #     {key:value, ... }
>>> len(symTbl)               # get the number of entries in the dictionary
1
>>> symTbl['x']               # get the value associated with the key 'x'
6
>>> symTbl['x'] = 10          # update the associated value of key 'x' to 10
>>> symTbl['foo']             # if a key is not present, this will cause an
KeyError: 'f'                 #     exception
>>> 'foo' in symTbl           # the right way to check if a key is present
False
>>> 'x' in symTbl
True
>>> symTbl['y'] = 4           # insert the key 'y', with the value 4
>>> symTbl['z'] = 15          # insert the key 'z', with the value 15
>>> symTbl                    # the dictionary is not sorted alphabetically
{'y': 4, 'x': 10, 'z': 15}    #     by key!!
```

You can iterate over the entries in a dictionary using a for loop. For example:

```
>>> for varName in symTbl:
...   print("Variable name:", varName, "=>", "Value:", symTbl[varName])
Variable name: y => Value: 4
Variable name: x => Value: 10
Variable name: z => Value: 15
```

For this assignment, the symbol table will be specified in a separate text file. Each line will contain one variable name, followed by a space, followed by its integer value. For example, the file `vars.txt`:

```
x 10
y 20
z 30
```

3.2. Required Functions

You are required to write three functions for this assignment. The functions, their parameters and behavior must **match exactly**.

3.2.1. Parsing The Prefix Expression

You will implement the `parse` function that you pseudocode'd in problem solving. It takes a list of tokens (strings) and constructs and returns the root of the parse tree.

```
>>> tr = parse([‘*’, ‘8’, ‘+’, ‘x’, ‘y’])
>>> isinstance(tr, MultiplyNode)
True
```

Note, you may find this useful! In order to check whether a string is a number or not, you can use the built-in string function, `isdigit()`. For example:

```
>>> str1 = “123”
>>> str1.isdigit()
True
>>> str2 = “x”
>>> str2.isdigit()
False
```

3.2.2. Generating The Infix Expression

You will implement a function called `infix` that takes a parse tree for the expression as an argument. It traverses the tree in inorder fashion and generates/returns a string which represents the infix notation of the expression.

```
>>> infix(tr)
‘(8 * (x + y))’
```

Please note, your solution should include sufficient parenthesis to make the order of operations unambiguous.

3.2.3. Evaluating The Parse Tree

You will implement a function called `evaluate` that takes a parse tree and a symbol table (a python dictionary with entries: key=string, value=integer) as arguments. It traverses the tree in preorder fashion and returns the integer result for the expression.

```
>>> symTbl
{'y': 20, 'x': 10, 'z': 30}
>>> evaluate(tr, symTbl)
240
```

3.3. Program Details

We are providing you with a starter implementation, rename it to `derp.py`:

http://www.cs.rit.edu/~vcss241/Pub/Labs/08/derp_py.txt

It contains all of the class definitions and maker functions from the table on page 2. In addition it has stubbed out the three functions you are required to implement, as well as a skeleton that you must complete for the `main`.

To begin, the interpreter (Derp) will greet the user (Herp) and prompt for the symbol table file.

```
$ python3 derp.py
Hello Herp, welcome to Derp v1.0 :)
Herp, enter symbol table file: vars.txt
```

Next, your program should read the symbol table into a python dictionary, and display the variable names and values (in any order).

```
Derping the symbol table (variable name => integer value)...
x => 10
y => 20
z => 30
```

Finally, your program should enter a loop that runs until the user hits the RETURN key (by itself). It should prompt for a prefix expression, and then produce both the infix expression and the evaluated result.

```
Herp, enter prefix expressions, e.g.: + 10 20 (RETURN to quit)...
derp> + 10 * x y
Derping the infix expression: (10 + (x * y))
Derping the evaluation: 210
derp> // * x 5 + - 10 y z
Derping the infix expression: ((x * 5) // ((10 - y) + z))
Derping the evaluation: 2
derp>
Goodbye Herp :(
```

For this assignment we guarantee the following:

- The symbol table will exist and follow the described format.
- The user will enter valid prefix expressions in the correct format.
- Any variables used in the prefix expressions will exist in the symbol table file.

In order to receive full credit for this assignment you must construct a tree using the node classes we provide. *** **You are not allowed to use Python's** `eval()` **method to directly evaluate expressions stored as infix strings.** ***

4. Grading (100 points)

Problem Solving (20 points): As determined by your instructor from problem solving work.

Implementation (80 points):

Functionality (75 points):

- symbol table dump: 5 points
- `parse` function: 25 points
- `infix` function: 20 points
- `evaluate` function: 25 points

Code Style (5 points)

5. Submission

Submit your program, `derp.py`, to the **myCourses dropbox** for this lab.