Computational Problem Solving CSCI-603
PreTee Interpreter Lab 8

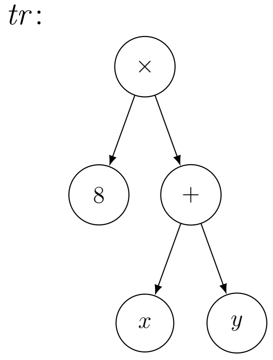
1 Introduction

An *interpreter* is a program that executes instructions for a programming language. The `python3` interpreter executes Python programs. This assignment involves writing an interpreter for a very simple language called **PreTee**.

The interpreter accepts *prefix* mathematical expressions as input, where a prefix expression is one in which the mathematical operation is written at the beginning rather than in the middle. Inside the interpreter is a *parser*, whose job is to convert prefix expressions, represented as a string, into a collection of tree structures known as *parse trees*. The PreTee interpreter can evaluate mathematical expressions using the operators `+`, `-`, `*`, and `//`. The supported operands are positive 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, PreTee displays the *infix* form of the expression and evaluates the result.

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

Variable	Value
`‘‘x’’`	10
`‘‘y’’`	20
`‘‘z’’`	30



Infix expression: `‘‘(8 * (x + y))’’`

Evaluation: `240`

Let's assume that the parse tree has already been constructed from the prefix expression. The infix form of the expression can be obtained by doing an `inorder` (left, parent, right) traversal of the tree from the root and constructing a string. Recall the private `__inorder` helper function from lecture that `__str__` called when getting a string representation of a general purpose tree.

```
def __inorder(self, node):
    if not node:
        return ' '
    else:
        return self.__inorder(node.left) + \
               str(node.val) + \
               self.__inorder(node.right)
```

The result of the expression can be found by evaluating the root node in `preorder` fashion (parent, left, right).

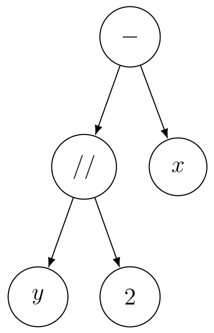
The implementation stage will cover details of the symbol table and evaluation of the parses tree to generate the result of the expression.

2 Problem Solving

Work in a team of three to four students as determined by the instructor. Each team will work together to complete a set of activities. Please complete the questions in order. Do not read ahead until you have finished the first two questions.

1. Consider the following tree.

   tr:



   Write the prefix expression for the tree, *tr*, above. Recall in a prefix traversal, the order is parent, left then right.

2. Write the infix expression for the tree, *tr*, from problem 1. Use parentheses to specify the order of operations.

3. The parse tree can be built by reading the individual tokens in the prefix expression and constructing the appropriate node types. Given this prefix expression as the token list:

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

   (a) Draw the parse tree for the expression using the diagramming method above.
   (b) Write the infix expression for the parse tree. Make sure you include parentheses to preserve the operator precedence.
   (c) What would be the result of evaluating the parse tree? Assume the symbol table on the first page is in effect.

We will be representing the various types of nodes in the tree as Python classes.

Node class	Slot name(s)	Type(s)	Constructor
LiteralNode	val	int	__init__(self, val)
VariableNode	id	str	__init__(self, id)
	left	object (a Node class)	
MathNode	right	object (a Node class)	__init__(self, left, right, token)
	token	str (“+”, “-”, “*”, or “//”)	

For example, the code below constructs a parse tree for the expression tree in the first question:

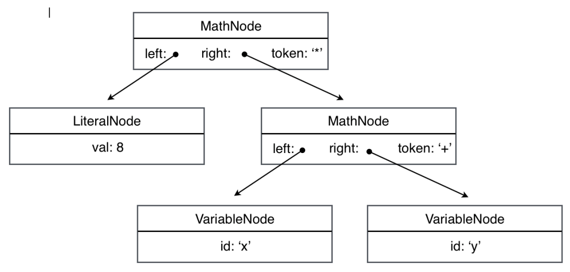
```
>>> tr = MathNode( \
              MathNode( \
                    VariableNode('y'), \
                    LiteralNode(2), \
                    '\\'), \
              VariableNode('x'), \
              '-')
```

Assume the prefix expression has been converted into a list of string tokens:

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

Now let’s build and return a parse tree using the token list and node classes. The diagram below shows the node structure of the parse tree. For this problem, we will call that tree *tr*. The root of the tree *tr* is a MathNode. Each node is indicated by its type (one of the 3 possible node classes), and its internal slot values are also shown. Arrows indicate that a node has a child node. Here is a picture of the parse tree for the prefix expression, `‘‘* 8 + x y’’`, which is stored in the token list as `[‘‘*’’, ‘‘8’’, ‘‘+’’, ‘‘x’’, ‘‘y’’]`:

tr:



4. Write a function `parse`. It takes a list of tokens for a prefix expression and returns a parse tree. The parse tree for the token list `[‘‘*’’, ‘‘8’’, ‘‘+’’, ‘‘x’’, ‘‘y’’]` should produce the tree corresponding to the image above.

   Note that a prefix expression has one of the following forms:

   - A number that is non-negative in value (0 or greater).
   - A variable.
   - $+ \, e_1 \, e_2$, where e_1 and e_2 are prefix expressions.
   - $- \, e_1 \, e_2$, where e_1 and e_2 are prefix expressions.
   - $* \, e_1 \, e_2$, where e_1 and e_2 are prefix expressions.
   - $// \, e_1 \, e_2$, where e_1 and e_2 are prefix expressions.

   The string function `isdigit` is useful for testing whether or not a string looks like a number. The string function `isidentifier` is useful for testing whether or not a string looks like a variable.

```
>>> str1 = '123'
>>> str1.isdigit()
True
>>> str2 = '3x'
>>> str2.isdigit()
False
>>> str3 = 'x'
>>> str3.isidentifier()
True
>>> str4 = '2x'
>>> str4.isidentifier()
False
```