American University of Sharjah
School of Engineering
Computer Science & Engineering Dpt.
P. O. Box 26666
Sharjah, UAE

Instructor: Dr. Michel Pasquier
Office: ESB 2057
Phone: 971-6-515-2883
E-mail: mpasquier@aus.edu
Semester: Summer 2022

CMP 321 PROGRAMMING LANGUAGES

PARSE TREE PROJECT

Due date/time: Saturday 2nd July 2022, at 11 pm

Instructions: Submit via the assignment box on *iLearn,* before the deadline, your *project deliverables* as elaborated below, without this document. Make sure to include in all your files the *name and ID of all team members.* No printed/hard copy is required. Follow any further instructions as may be posted on *iLearn.* Note that due to the belated deadline, late submissions will *not* be accepted.

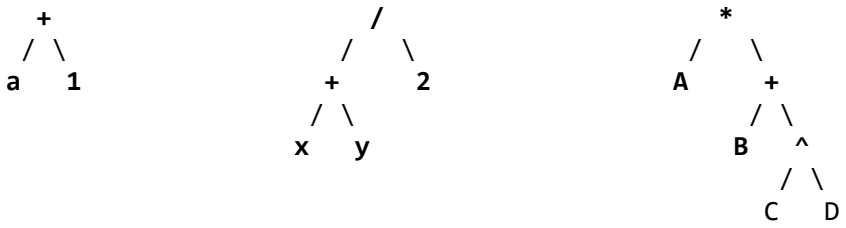
For this assignment, you must submit *one* Python *script* containing your complete *source code*, one text file that is the *program output* as described hereafter, and another file that summarizes each *team member's role and contribution.* You must bundle all files in a *single* Zip/Rar *archive* named after all team members e.g., RAhmed-GBarlas-MPasquier.zip, that you will upload to iLearn. One should be able to extract your program from the archive and run it "as is" to reproduce the output. These are *requirements;* non-compliant submissions will be penalized.

Note that you are to complete this assignment *as a team,* from *two to four* students. Furthermore, each team must *work independently* and hand in their own original answers. You are *not* allowed to discuss or *share* any solution or to *copy* from others or from any sourced material. Plagiarism and cheating will be severely penalized, starting with a zero grade for the assignment. Recall you are bound by the AUS Academic Integrity Code which you signed when joining AUS. Lastly, note that any team member may be called upon later, as deemed neessary, to explain any part of the project code or design, also their contribution, etc.

Overview: In this project, you are to *implement* in Python *and make use of* a simplified *Parse Tree* class that stores the programming logic of a given piece of code, regardless of the programming language used. This parse tree is the intermediate data structure used for translating from one language to another e.g., from Python to Lisp or from Postscript to C, etc. and is also an essential part of the compilation process. For simplicity's sake, the scope of this assignment is limited to the case of binary trees for simple algebraic expressions, as well as ternary operators later.

Binary Parse Tree: Consider a simple parsing application where valid "programs" consist of *algebraic expressions* using binary operators over integer variables or constants. In this scenario, the five possible binary operators are: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `^` (exponent). Variable names consist of a single letter, either uppercase or lowercase, while constants are integers `0..9`.

Furthermore, we consider two generic representations of an algebraic expression i.e., the *parse tree* and the *list* representation. For example, the expression `a+1` is defined by the parse tree below left, and the equivalent list is: `['+', 'a', '1']`.



Similarly, the expression `(x+y)/2` is defined by the parse tree above center, and the equivalent list is: `['/', ['+', 'x', 'y'], '2']`. To take another example, the expression `(A * (B + (C ^ D)))` is the parse tree above right, and the equivalent list is: `['*', 'A', ['+', 'B', ['^', 'C', 'D']]]`. Importantly, notice how the list elements can be obtained via a *pre-order traversal* of the parse tree, and conversely.

Since for now all operators are binary, the parse tree is simply a *binary tree* where the external nodes (leaves) denote a variable or a constant and the internal nodes represent an operator, where left and right children are the operands. Then, the list representation similarly stores operators and their operands, in the same order. The operands may be nested lists, equivalent to the subtrees in the parse tree.

(1) Implement a `BinaryParseTree` class in Python by expanding the code given in the Appendix with the following functions:

- `fromList()`: builds the parse tree out of a given list representation.
- `toList()`: returns the list representation equivalent to the parse tree.
- `prettyPrint()`: prints the parse tree as a binary tree, as shown above.

Note that, since the tree is printed line by line, you will need to perform a level-order traversal. Also, horizontal spacing depends on the height of each node. To help you, the `height` and `nodesByLevel` functions are provided. The latter is a straight BFS traversal, which you will need to amend to suit your purpose.

(2) Write a `test1()` function that will take as argument any number of list representations, create a parse tree object for each, and pretty print it. Run your test function on the five test cases that are the three above examples plus the following two expressions: `(a+b)*(a-b)+(a+b)^2` and `2*x^3*y^3-x^2-y^2`

All output should be included in your submission. As an example, the one-liner below should create then print the parse tree shown above center:

```
BinaryParseTree().fromList(list_rep2).prettyPrint()
```

Your test function should also verify that input and output list representations do match. For example, the following should be true, for any expression list:

```
BinaryParseTree().fromList(list_rep).toList() == list_rep
```

Syntax Translation: Depending on a programming language's syntax, three notations are possible for algebraic expressions: *infix, prefix,* and *postfix.* All of C, C++, Java, and Python use infix syntax, while Lisp and Scheme are famously based on prefix notation, and languages such as Forth and PostScript have a postfix syntax.

For example, the average of two variables is written using infix notation as `"(x+y)/2"`. Parentheses may be needed to denote calculation order, like in this case, or not; spaces are optional. The same average calculation in prefix notation is `"(/ (+ x y) 2)"`. In this format, parentheses and spaces are both mandatory. Finally, in postfix notation, the expression becomes `"x y + 2 /"`. Here, no parentheses are needed; spaces are mandatory.

Note that, for simplicity's sake, in this assignment you can assume that every expression is enclosed in parenthese. Likewise, you can assume as before that variables and constants consist of a single character and a single digit, respectively.

Regardless of syntax, the above three expressions compute the same result hence have the same parse tree(!) and the same list representation. The above average calculation is thus represented as: `['/', ['+', 'a', 'b'], '2']`.

More examples of valid expressions are shown below, all with parentheses:

Infix	Postfix	Prefix
`( a + 1 )`	`(a 1 +)`	`(+ a 1)`
`((a - b) / 3)`	`((a b -) 3 / )`	`(/ (- a b) 3)`
`(a * (b + (c ^ d)))`	`(a (b (c d ^) +) *)`	`(* a (+ b (^ c d)))`

(3) Add to your `BinaryParseTree` class the following functions. Make sure each function will raise an exception if the input string contains an invalid expression.

- `fromPrefix()`: builds the parse tree out of a given prefix string expression.
- `toPrefix()`: returns the prefix string expression equivalent to the parse tree.
- `fromInfix()`: builds the parse tree out of a given infix string expression.
- `toInfix()`: returns the infix string expression equivalent to the parse tree.
- `fromPostfix()`: builds the parse tree out of a given postfix string expression.
- `toPostfix()`: returns the postfix string expression equivalent to the parse tree.

Note that the three functions for creating a parse tree are very similar, and likewise the three functions for returning an equivalent string expression, of course.

- Add to your class the necessary function/s so that passing a parse tree object to the `list` constructor will yield the list representation equivalent to the parse tree. So for instance if `t=BinaryParseTree().fromPostfix('(n 4 ^)')` then the expression `list(t)` should yield: `['^', 'n', '4']`

(4) Write a `test2()` function similar to `test1()` earlier, that will now print for each given parse tree the infix, prefix, and postfix equivalent string expressions. Run your test function on the same five examples as before.

All output should be included in your submission. As an example, the code below should return the postfix string expression for the average calculation:

```
print(BinaryParseTree().fromInfix(infix_str).toPostfix())
```

That is, the output should be: `'((x y +) 2 /)'`

Your test function should also verify that all input and output string representations do match. For example, the following should be true, for any prefix string:
`BinaryParseTree().fromPrefix(prefix_str).toPrefix() == prefix_str`

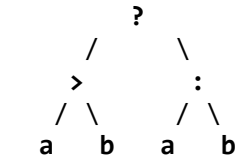
Syntax Extension: The Python *ternary operator* is used to select one of two expressions based on a Boolean condition. The syntax is: `expr1 if bool_expr else exp2` where `expr1` is selected if `bool_expr` evaluates to true, otherwise `exp2` is selected. A ternary operator can be represented as a binary parse tree and its equivalent list representation, as follows: `['?' bool_expr [':' expr1 expr2]]`

Since in this assignment we are limited to 1-character operators and operands, you can assume the only comparison operators are: **>** (larger than) and **<** (smaller than).

Some examples of ternary operators and their representations are given below:

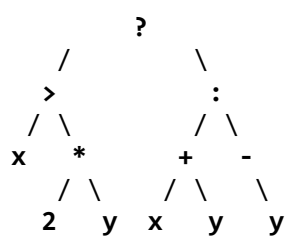
```
a if a>b else b   # max

['?' [> a b]] [':' a b]
```




```
x + y if x>2*y else -y

['?' [> x [* 2 y]] [':' [+ x y] [- y]]]
```



(5) Create a class called `XBinaryParseTree` that extends your `BinaryParseTree` class to accommodate ternary operators. Amend any function as necessary (and no other) so that it can now handle string expressions and list representations that include the Python ternary operator. (Make sure to use proper OOP practices.)

(6) Write a `test3()` function that will take as argument a Python expression i.e., and infix string expression (as per the examples above), create an extended parse tree object, pretty print it, then print and return the equivalent list representation. Run your test function on the three test cases that are the two above examples plus the following expression: `(a-b)^2 if a>b else (a+b)^2`

All results and outcome should be included in the one output file that is part of your submission, as always.

Appendix: Below is the parse tree implementation you need complete, test, and extend.

```
class BinaryParseTree:
    class Node:
        def __init__(self, data, left=None, right=None):
            self.data, self.left, self.right = data, left, right
        def __str__(self): return self.data
    def __init__(self, root=None): self.root = root
    def fromList(self, list_repr): pass  # to be implemented
    def toList(self): pass               # (part I)
    def prettyPrint(self): pass          #
    def fromPrefix(self, expr=''): pass  # to be implemented
    def fromInfix(self, expr=''): pass   # (part II)
    def fromPostfix(self, expr=''): pass #
    def toPrefix(self): pass             #
    def toInfix(self): pass              #
    def toPostfix(self): pass            #
    #def ???                             # for list(tree)
    def nodesByLevel(self):
        if self.root == None: return []
        node = self.root
        bfsQueue, allNodes = [node], []
        while bfsQueue:
            node = bfsQueue.pop()
            allNodes.append(node.data)
            if node.left:  bfsQueue.insert(0, node.left)
            if node.right: bfsQueue.insert(0, node.right)
        return allNodes
    def height(self):
        def __h(node):
            if node == None: return -1
            else: return 1 + max(__h(node.left), __h(node.right))
        return __h(self.root)

class XBinaryParseTree(BinaryParseTree): # to be implemented
    pass                                 # (part III)
```