# Tree++: Proposal

Allison Costa, Laura Matos, Jacob Penn, and Laura Smerling

arc2211, lm3081, jp3666, les2206

*Programming Languages and Translators*
*Department of Computer Science, Columbia University*

**Abstract:**

We propose a language whose underlying format utilizes the data structure of a tree. In our language, trees must be utilized for more complex algorithms. Whereas other languages, such as Java and C, allow for multiple data structures, our language restricts the user to trees, because they can be used effectively to represent most all other data structures (such as lists, graphs, queues, stacks, etc.) and, with our implementation, provide a quick and intuitive way of manipulating data. We do this by optimizing the way trees perform search and sorting algorithms.

**Motivation:**

The motivation for our language arose from our confidence in the tree data structure to effectively solve complex and interesting algorithms and yet our frustration with the current set up of trees in Java and C where search and sorting algorithms force the user to utilize multiple loops and traverse through the tree in a fairly inefficient manner. We plan to both allow the user to decide a great deal about the layout and balancing of the tree; however, we also want to provide general structural information about child, parent, and level of node, which will prove useful when implementing search and sorting algorithms. These features make our language ideal for performing a number of programs including and beyond those typically allocated to trees in object-based programming languages. Like in most languages, our trees can be used for a number of sorting and search algorithms, such as Breadth First Search or Binary Search; however, our tree structure can also be used for applications such as managing temporary results, which in other languages is typically allocated to the stack data structure. Overall, our implementation increases the efficiency and intuitiveness of typical tree-based algorithms and can be effectively used to program other algorithms, which are usually allocated to different data structures.

**Syntax:**

*References:*

1)*"The C Programming Language"* by Brian W. Kernighan, Dennism Ritchie, (2nd edition)

2) *The Lorax Programming Language Doug Bienstock*, Chris D'Angelo, Zhaarn Maheswaran, Tim Paine, and Kira Whitehouse,
http://www.cs.columbia.edu/~sedwards/classes/2013/w4115-fall/proposals/Lorax.pdf

**1: Tokens-**

Similar to C there are six classes of tokens: identifiers, keywords, constants, string literals, operators and other separators. All tabs, newlines and comments are considered "white space" and will be ignored. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

| | |
|---|---|
| *child.parent* | Pointer to parent |
| *.data* | Get data for node |
| ; | Line terminator |
| | No white space detector |

**2: Comments-**

The characters (* introduces a comment and *) ends a comment. The comments can be nested like they are in OCaml.

**3: Identifiers-**

An identifier can contain a series of upper and lowercase letters and numbers. All identifiers are case sensitive. If declaring a tree the identifier must begin with a capital letter. All other identifiers must begin with a lower-case letter. Proper naming convention is camelCase. All identifiers within the same function must have different naming conventions.

**4: Keywords-**

The following identifiers are reserved for use as keywords, and may not be used otherwise:

int, double, float, string, tuple
tree, node, data
print, empty, null

If, else, for, switch, case, return, break, range

**5: Constants- ( C syntax for all types)**

Types: Integer-constant, floating-constant, and character-constant

**6: Operators-**

| | |
|---|---|
| +, - | The additive operators are grouped left to right |
| /, *, % | The multiplicative operators are grouped from left to right |
| <,>, <=, >= | Relational operators group left to right and are used for comparison |
| ==, != | The equality operators follow the same rules as the relational operators |
| &&, \|\| | Boolean operators |
| ^^ | Swap child and parent |
| << | Shift children left |
| >> | Shift children right |
| *root* : *n* | Find the nth node in the tree via breadth first search (bfs) traversal (head of the tree = 0, first child in tree = 1, if doesn't have n nodes throws error) |
| *node*[n] (*node*[n][n]) | Finding the nth child (child's child) of node (first child = 0) |

Shift and swap operators explained:

ex) has the ability to shift and add node in tree through <<+, >>+, <+>

ex) has the ability to shift and delete node in tree through <<-, >>-, <->

ex) has the ability to shift and swap node and child in tree through <<^^, >>^^

**7: Type Specifiers-**

Char
String
Tuple
Int
Float
Double
Node

- Primitive data type and data
- Info on parent
- Info on children

**8: Declarations-**

**8.1: Node declaration syntax**

- ex) appleLeaf = [parent , data] creates a node with specific parent and data
- ex) appleLeaf = [parent] creates a node with void data

- ex) appleLeaf = [data] creates a node without a parent but specific data
- Ex) appleLeaf = [] creates a node without a parent and void data

**8.2: Functions**

- ex)AlphaBetaPrunning(x){
  statement;
  return x;
  }
- Must have a main function that should contain the desired final output

**9: Statements-**

**Selection Statements -**

if ( *expression*){ *statement*}
if ( *expression* ){ *statement* }else{ *statement*}
switch (*expression*){ case: *statement*...}

**Iteration statements - similar to python**

| | |
|---|---|
| for x in range(0, 3){<br>*statement*<br>} | Iterates through the nodes based on the bfs numbers in the tree |
| for x in AppleTree{<br>*statement*<br>} | Iterates through all of the nodes in the entire tree |
| for x in dfs() range(0,3){<br>*statement*<br>} | One can write their own iterative function if they desire with their desired order and iterate through this function through the for loop |
| for x in dfs() AppleTree{<br>*Statement*<br>} | One can write their own iterative function if they desire to iterate through all the nodes in the entire tree |

**10: Standard Library Functions-**

tree *name* - creates node 0 with data null, parent null, children null

*name*{head[child1, child2[grandchild1a, granchild2a...], child3[grandchild1b, grandchild2b...]...]} - reads in the data, all strings should be replaced with data or identifiers

root(*node*)
print(*value*)
empty(*node*)
height(*node*)
depth(*node*)
level(*node*)

edge(*node*)
path(*node*)

**Sample Code:**

```
(* a functions that prints out the tree's nodes through breadth first traversal *)
bfs(tree){
	int h = height tree;
	for x in tree{
		print(x.data);
	}
}

(* makes a binary tree *)
binaryTree(node, int)
{
	(* make sure the depth from the target node to root of the tree is equal to the given int arg *)
	if depth(node) == int {return 0;}
	else {

		node[0] = leaf[];
		node[1] = leaf[];

		binaryTree(node[0],int);
		binaryTree(node[1],int);
	}

}

main(){

	(* a tree with string children *)
	tree SampleTree1 = {'a'['a1', 'a2'['a21', 'a22'], 'a3'['a31', 'a32']]};

	(* an empty tree *)
	tree SampleTree2 = {};

	(* a node with a parent and int data *)
	leaf1 = [SampleTreeData, 20];

	(* an empty node *)
	leaf2 = [];
```

```
        (* a tree with int children *)
        tree SampleTree3 = {4[5, 6[leaf1, 7]]};

        (* replacing the second child in SampleTree3 with the leaf1 node*)
        tree SampleTree3 : 2 = leaf1;

        (* replacing the third child of SampleTree3 with a node of value 8 *)
        SampleTree3 : 3 = [8];

        bfs(SampleTree1);
        binaryTree([], 5);

}
```