The *Lorax* Programming Language

Doug Bienstock, Chris D'Angelo, Zhaarn Maheswaran,
Tim Paine, and Kira Whitehouse
dmb2168, cd2665, zsm2103, tkp2108, kbw2116

Programming Translators and Languages,
Department of Computer Science,
Columbia University, New York, NY 10027, USA

Abstract

Here we propose a language based upon the tree data structure. Whereas a language like Java relies on an inheritance structure deviating from the type Object, we propose a paradigm wherein the tree, which can be used effectively to represent lists, graphs, queues, stacks, etc, becomes the central object around which the language is structured. Much like Lisp and lists, our language seeks to naturalize the tree structure to allow for fast and simple tree based algorithms, while maintaining the flexibility to code other data structures within tree based constructs.

1 Motivation

The motivation for our language arose out of frustration with implementing trees and tree based algorithms in other high level programming languages. While languages like C and Java allow you to implement trees, they are not a native aspect of the language. As such, you must define and manage the node structure yourself, which can be a tedious and complicated affair, especially for young programmers, or those who are inexperienced with tree data structures. Trees are often simply a construct of the language, rather than something fundamental to its structure, like the object of an object oriented language, or the function of a functional language. Our language attempts to remedy these problems by handling a lot of the business logic of implementing trees and their algorithms under the hood, and allowing the programmer to use trees in a more native and intuitive way. This simplicity combined with added features of our language allow new and unconventional applications beyond those easily managed by typical object-centric programming languages. An intuitive application is to easily build and maintain a min-heap or a binary search tree of strings. A more complex situation could be to create a tree structure used to manage an organizational hierarchy for a company. A programmer could implement the tree structure in this language to manage and perform operations on an organization?s employees easily. Virtually any tree-based algorithm should now be much easier and more natural to program, and in fewer lines.

2 Syntax

2.1 Data Types

Type	Description	Example
int	an integer, typically reflecting a machine's natural integer size	1972
double	double-precision floating point	3.14
string	capable of holding multiple characters in the local character set	"I love trees"
tuple	a grouping of data encapsulated in a single tree node	⟨5, "five"⟩
tree	a collection of data with a parent-child relationship	1[2, 3[4, 5]]

A tree can be initialized a variety of ways. It can be instantiated with null elements, such that children are not defined with any inherent data. Trees can also be defined to hold tuples as their data types. Trees have branching factors, i.e. a maximum number of children. When trees of different branching factors are added together, the larger branching factor is inherited, so that both trees share the same branching factor. A tree can only hold one type of data: thus, whether the children hold data of type double or string or tuples of doubles and strings, all nodes of the tree must be this same type. On the other hand, tuples may contain data of multiple types. This type ruling will be enforced on compile time.

2.2 Arithmetic, Relational, and Logical Operators

⟩ ⟩= ⟨ ⟨= == !=	Compares values within tuples and trees. All comparisons return a tree with no children and a sole integer value, 1 for true, 0 for false.
@	Accesses the data element of a tree node. This operation is used to access the int, double, and string data types stored within trees. For instance, given a tree x whose 0th child is a tuple ⟨3, "six", 9.0⟩. We can access the string "six" with the operation x @ 0 % 0.
+ -	These operations induce compile time errors if not used correctly with types. If both types involved are doubles or integers, this simply produces the mathematical addition or subtraction of the two numbers. If both types involved are strings, the + operator simply concatenates the two strings, one following the other. If both types involved are trees, the + operator operation constructs a new tree consisting of both. The two original trees are glued together into an anonymous tree, which can be assigned to a variable. Note: the subtraction - operation cannot be used with strings or trees; this will induce a compile time error.

2.3 Tokens

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

root(*tree*)	Returns the root of a tree, without any children, but does not dereference its value. The original tree passed in as an argument is unchanged.
children(*tree*)	Returns an ordered set of the children of the input tree. The set can be operated on but cannot be assigned using the = operator. This design choice prevents users from manipulating trees.
print(*value*)	Prints to standard output the value provided as an argument. Values supported are doubles, ints, and strings.

3 Control Flow

3.1 Statements and Blocks

;	The semicolon is a statement terminator.
*/*Hi I am a comment. */*	These are multi-line comments. They are not nestable.
//	These are single line comments.

3.2 If-Else and Loops

if(*expression*)
statement
else
statement
	The if-else statement is used to express decisions. Any expression evaluated will not compile unless it can evaluate to a valid boolean.

for(*expression*)	For loops contain an assignment, a boolean conditional expression, and an incrementor.

4 Miscellaneous Standard Library Functions

height(*tree*)	Returns a tree with no children and data representing the height of the input tree.
empty(*tree*)	Returns whether or not a tree is empty. Note this is the same as checking whether the tree is null, but may not return the same value as height if applied to a tree of height 0. This returns a tree with no children and data representing a boolean 0 or 1. This is helpful when applied to trees who have been declared with branching values, but not defined with any internal data.
type(*tree*)	Returns a tree with no children containing solely a string representation of the type of data stored in the parameter tree's nodes. For instance, a possible return value might be "int" or "int, string, double, int" .
branchingfactor(*tree*)	Returns a tree with no children and data representing the branching factor of the input tree.

5 Program Structure

Below we provide a short sample program to demonstrate the syntax and flow of our language. In short, there is a *main* function and a user defined *depthFirstSearch* function. The interpreter will only run the *main* function. Thus, if we did not call *depthFirstSearch* from within main, that code would never be stepped into during the course of program execution. We provide sparse comments within the program, and point out some important subtleties beneath the complete source code.

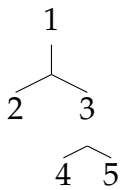
```
1  /*a recursive function moving depth first through a tree printing its data*/
   depthFirstSearch(tree) {
3      if (tree == null)
           return null;
5
       /*print the first value (the data) within the tuple*/
7      print(tree @ 0);

9      /*use second value of tuple as key-value flag to mark node as visited*/
       tree @ 1 = 1;
11
       /*iterate through the tree recursively*/
13     for (child = 0; tree % child != null; child += 1) {
       /*check if we have visited child before making recursive call*/
15         if ((tree % child ) @ 1 == 0) {
               depthFirstSearch(tree % child);
17         }
       }
19 }
```

```
1  /*A program must have a main function, as this will be the only code run by
       the interpreter. Any command line arguments will be passed in in a tree
       data structure.*/
   main (command_line_arguments) {
3
     /*a is a multi-level tree with integer children*/
5    a = 1[2, 3[4, 5]];

7    /*b is a tree with no children, whose data is integer 6*/
     b = 6[];
9
     /*c is a tree with no data, no children, and a declared max degree of 4*/
11   c = [](4);

13   /*d is a tree whose children are tuples of type <int, string, double>*/
     d = <101, "hello", 3.14>[<102, "goodbye", 2.618>];
15
     /*access the 0th data element within tree d, returning integer 101*/
17   d @ 0;
     /*access the 0th data element within tree a, returning integer 1*/
19   a @ 0;

21    /*access the 0th child of tree a, returning a tree whose data item is 2*/
     a % 0;
23   /*access the 0th data element in the 0th child of a, returning integer 2*/
     a % 0 @ 0;
25
     tree_to_print = <1, 0>[<2, 0>, <3, 0>[<4, 0>, <5, 0>]];
27   /*this function call will print 1 2 3 4 5*/
     depthFirstSearch(tree_to_print);
29 }
```

A couple important notes to make about the program above, before we dive more explicitly into the code. In the creation of all of these trees the compiler will be sure to keep the integrity of the tree data structure. Some rules that it will obey:

1. A tree can only have at most one parent
2. A tree cannot be parent to his own parent
3. A tree cannot be parent to his brother

Note, as a result of these rules, that our language does not support internal loops, or any form of graphs. In the beginning lines of the *main* function above, we defined trees *a* and *b*. The definition *a = 1[2, 3[4, 5]];* is displayed below in a visual sense:



Each leaf node, i.e. the nodes without any children, points to null. The compiler will conclude that this tree contains data type int and degree 2. Initializing a tree with improper combinations of data types would result in a compile time error. The definition of tree *b = 6[];* is a little more difficult to explain with diagrams. This tree has no children, but has the integer 6 in its data field. There are a couple of other ways in which we could have defined this tree. For instance, we could have defined b as $b = \langle 6 \rangle[];$. Here we use the explicit tuple bracket operators.

Importantly, every data item in a tree is actually a tuple. Tuples are very much related to trees. They contain collections of items. However, unlike trees, tuples can contain data of different types. Trees cannot mix data types. It would have also been possible to define b simply as $b = 6$. This reveals the inherent tree-ness of our language. Even primitive data types are initialized and stored in tree data structures.

Here we will look at the two more obscure operators in the program above, the @ operator and the % operator. The % operator allows us to access the internals of a tree's structure; that is, its children. Because children are kept in an ordered listing we can refer to them with indexing. Note that the % operator returns a pointer to a single child, not an array or list of multiple children. We use the % operator to traverse through a tree in the *depthFirstSearch* function. The @ operator allows us to access data within a tree node. We can combine the % and @ operators to access the data within a child tree.

With the functionality of these operators in mind, let's take one last look at the main loop within the *depthFirstSearch* function.

```
1     for (child = 0; tree % child != null; child += 1) {
          /*check if we have visited this child*/
3         if ((tree % child ) @ 1 == 0) {
              /*extract the integer data from the child tree and call DFS
                  recursively*/
5             depthFirstSearch(tree % child);
          }
7     }
```

Above, *child = 0* is defining a tree whose data is a tuple containing a single integer 0. This initialization syntax is creating a childless tree, and is syntactic sugar in some regards. It is the shortened form of declaring *child = 0[]*. The condition and afterthought of the for loop, *tree % child != null* and *child += 1* respectively, are also syntactic sugar. In both of these expressions *child* is short for *child @ 0*. This shortened form makes it easy for the programmer to iterate through all the children of a given tree.