Lorax Language Reference Manual

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

Columbia University

October 28, 2013

"I am the Lorax. I speak for the trees." – Dr. Seuss 1971

Table of Contents

Introduction

This manual describes the Lorax programming language. The Lorax language provides a syntax that enables the easy creation and manipulation of the tree abstract data type. Trees are a native data type of the language. Each tree encloses a value of a Lorax primitive type. Tree's branching factor and value data type are strongly typed. Language operators allow you to insert trees, traverse their structure, access their node contents, and compare data items within tree nodes. The programmer can create and manipulate these trees while the Lorax language handles memory management and tree structural consistency under the hood.

Lexical Conventions

Comments

In-line comments are preceded by `//`. Block comments are delimited by `/*` and `*/`. Block comments can be written on a single line or can span multiple lines. Nesting is not be allowed.

Identifiers

An identifier is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper and lower case letters are different. Identifiers may not have a length greater than 31 characters.

Keywords

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

```
int             root            char
float           mod             degree
string          print           while
return          if              tree
for             else            bool
break           true            null
continue        false
```

Constants

A constant is a literal numeric or character value, such as 5 or 'm'. All constants are of a particular data type.

Integer Constants

An integer constant is a sequence of digits, starting with a non-zero digit. All integer constants are assumed to be decimal (base 10). Decimals values may use digits from 0 to 9.

```
459
0
8
```

```
12
```

Character Constants

A character constant is a single ASCII character enclosed within a single quotation marks, such as ‘Q’. Some characters, such as the single quotation mark character itself cannot be represented using only one character. To represent such characters there are several “escape sequences” that you can use:

Sequence	Definition
\n	New line.
\’	Single quote.
\”	Double quote.
\\	Backslash

Floating Point Constants

A floating point constant is a value that represents a fractional (floating point) number. It consists of a sequence of digits which represents the integer (or “whole”) part of the number, a decimal point, and a sequence of digits which represents the fractional part. Either the integer part or the fractional part may be omitted, but not both. The decimal point may not be omitted. Here are some examples:

```
float a, b, c, d;
a = 4.7;
b = 4.;
c = .7;
d = 0.7;
```

String Constants

A string constant is a sequence of zero or more ASCII characters, or escape sequences enclosed within double quotation marks. A string constant is of type “array of characters”. Strings are stored as a 1 dimensional tree of characters. For more on the structure of the string object see *Tree Types* section below. Here are some example of string constants:

```
// this is a single string constant
“tutti frutti ice cream”
// this one uses two escape sequences
“\”hello, world!\””
/* to insert a newline character into a string, so that when the
 * string is printed it will on two different lines you can use
 * the newline escape sequence ‘\n’
 */
print(“Hello\nGoodbye”);
```

Boolean Constants

There are only two boolean constants, `true` and `false`. They must be typed in all lowercase letters. An example of declaring a boolean from a constant:

```
bool success;
success = true;
```

Tree Constant

A tree constant is expressed as a sequence of values of a consistent primitive data type (choice of char, int, float, bool). A tree constant begins with the first value representing the root node value, followed by square brackets containing the root's children separated by commas. Trees maintain a single data type in all of the tree node values. Lorax strictly enforces the type it first recognizes in the root if no value has been explicitly declared.

```
/* a is a tree of depth 3, degree 2, of integer data type value of the
 * root node is int 1, and its children are of value 2 and 3
 * respectively. The child with value of 2 has no children.
 * The child of value 3 has two children of value 4 and 5 respectively.
 * The nodes of value 4 and 5 have no children

 */
1[2, 3[4, 5]]

/* b is a tree of depth 1, degree 1, of integer data type.
 * This tree has no children. Data type is inferred by root node value
 * in constant and branching factor is inferred.
 */
6[];
```

Data Types

Lorax promotes the use of tree structures as much as possible. There are four basic types that escape this norm.

Primitive Data Types

Integers

Integers (`int`) are represented in 16-bit 2's complement notation. The default value of an integer variable is 0.

Floating Point Numbers

Single precision floating point (`float`) quantities have a magnitude in the range of approximately 10^(+ or - 38) or 0; their precision is 24 bits or about seven decimal digits.

Booleans

Booleans can be either `true` or `false`. The default value of a boolean variable is `false`.

Characters

A character, or char, is any single ASCII character. The default value for a char is `0`.

Tree Types

As stated previously, Lorax encourages the use of tree structures as much as possible. Trees may contain any primitive data type as their tree node value. A string in Lorax is a tree of char data type node values that has its own definition syntax as we shall see but can be expressed as a tree constant as well.

Declaring Trees

You declare a tree using the `tree` keyword, followed by whitespace, followed by less than symbol, followed by a primitive data type in Lorax representing the node values of this tree, followed by the greater than symbol, followed by the identifier name being declared, followed by open parentheses, expression resulting in integer representing the branching factor, and finally closed parentheses. A tree can have a maximum degree of 64. Here is an example that declares a tree that has a degree (a.k.a branching factor) of 4 of `int` type:

```
tree <int>e(4);
```

Initializing Trees

You can initialize the elements in a tree by listing the initialized values, separated by commas, in a set of square braces. When declaring a tree without defining it, you must specify the type and branching factor in the declaration. Here is an example declaration sans definition:

```
tree <int>a(2);
a = 1[2, 3[4, 5]];
```

Here is an example declaration and definition:

```
tree a = 1[2, 3[4, 5]];
```

Accessing Tree Children

You can access the child of a tree by specifying the tree name, followed by the percent symbol, followed by the child index. The child index beings with zero. Here is an example statement of accessing the 4th index (5th child) of tree `a`:

```
a%4;
```

null is No-Child Indicator

`null` is a keyword without an explicit value. It cannot be assigned to any data type or tree in Lorax. It is used only in order to answer the question: does a tree exist? In the below example we test if this tree has a child:

```
tree <int>a(1) = 42;
bool b;
b = (a%0 == null); // b is true
```

Accessing Tree Node Values

You can access the node value of a tree by specifying the tree name, followed by the percent symbol, followed by the @ symbol. This can be combined with child accessing facility presented above. Here is an example statement of accessing the 4th index (5th child) of tree `a` and setting the value stored in that child to integer 5:

```
a%4@ = 5;
```

Strings

Strings are no different than a tree, but in Lorax a special keyword and syntax is provided to make the use of strings easier. Strings are combinations of characters that are delimited by double quotes. Strings are initialized as a tree structure branching factor of one terminated by the end of the tree having no child (`null`). Each tree node encapsulates a single character and has a single child for the next letter in the string. Below is an example of declaring and defining a string using convenient syntax:

```
string simple;
simple = "Hello";

// the above may also be represented this way
tree complicated;
complicated = 'H'['e'['l'['l'['o']]]];
```

Expressions and Operations

An expression consists of at least one operand and zero or more operators. Operands are typed objects such as constants, variables, and function calls that return values. Here are some examples:

```
47
2 + 2
cosine(3.14159)
```

Parentheses group subexpressions. Innermost expressions are evaluated first:

```
(2 * ( ( 3 + 10 ) - ( 2 * 6 ) ) )
```

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are type and value of the right operand.

Assignment Operators

Assignment operators store primitive values in variables, copy the reference of a tree to a tree variable, or assign a value to a tree nodes value. The Lorax assignment operator is =. It's a binary operator and is right-associative. When assignment is taken, the value of the expression on the right is assigned to the left value, and the new value of the left value is returned, which allows chaining of assignments. Assignment can take some of these example forms:

```
// where a is declared as int a;
a = 4;
// where b is declared as tree b; and c is a previously declared
// and defined tree
b = c;
// where d is previously declared as a tree containing int value types
d%0@ = 5;
```

In this section we describe the built-in operators for Lorax, and define what constitutes and expression in our language. Operators are listed in order of precedence.

Arithmetic Operators / Tree Operators

Lorax provides operators for standard arithmetic operations: addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`), along with modular division (`mod`) and negation (`-`) . Usage of these operators is straightforward when using primitive types. Arithmetic operations are not valid among boolean types. Here are some examples using arithmetic operators with primitives:

```
x = 5 + 3; // where x is of type int
y = 10.23 + 37.332; // where y is of type float
z = 'a' + 'c'; // where z of type char
```

You use the modulus operator `mod` to obtain the remainder produced by dividing its two operands.

```
x = 5 mod 3;
```

You use the negation operator on a `float` or `int` type.

```
x = -4;
```

Trees may only use the addition operator. Like all arithmetic operators the binary operation must contain the same type on either side. When the addition operator is used the tree on the right hand side of the `+` symbol will be inserted at the lowest available child index of the parent tree on the left hand side. If the tree on the left hand side contains a full set of children and even grandchildren the right hand side tree will be inserted in the nearest depth down. This rule allows for the easy concatenation of two trees representing strings. Performing an operation like this requires that the left hand side and right hand side are of the same data type. The left hand side tree must have a tree-degree greater than or equal to that of the tree on the right hand side. If the right hand tree has a tree-degree less than that of the left hand side tree the right hand side tree's degree will be increased to equal the left hand side tree degree. Examples of this operation below:

```
tree <int>a(2);
a = 1[2, 3[4, 5]]; // tree of degree 2, depth 3, int data type
```

```
tree <int>b(2);
/* Take the first child of tree a and insert as its first child
 * (which is it's first available child index) a tree whose
 * root value is 6, with no children, degree 2, int data type
 */
b = a%0 + <int>6[](2);
```

Trees may also use the unary postfix operator `--`. When this operator is written to the right of a tree expression the tree is "popped" from the tree. This "popped" portion is "orphaned" and cannot be retrieved after this operation if there is no tree reference variable to the child popped. The return value of this operation is the former parent of the orphan tree.

```
tree a;
a = 1[2, 3[4, 5]]; // tree of degree 2, depth 3, int data type
tree b;
/* after the below right hand side expression is complete tree a will
 * reference to a tree resembling 1[2, 3]. b will reference the tree
 * resembling 3[](2)
 */
b = a%1--;
```

Comparison Operators

You use the comparison operators to determine how two operand relate to each other: are they equal to each other, is one larger than the other, is one smaller than the other, and so on. When you use any of the comparison operators, the result is either `true` or `false`. Comparison operators are all binary operators and are left-associative. The operators require that operands me of the same type. Comparison operators may be used with all of the Lorax defined types. In the case of comparing trees the the definition of this comparison is indicated below:

Operator	Primitive Types Definition	Tree Type Definition
>	Greater than.	LHS # of nodes > RHS # of nodes
>=	Greater than or equals.	LHS # of nodes >= RHS # of nodes
==	Equal to.	LHS tree structure and data is equal to RHS tree structure and data Can also be used to compare to `null`
!=	Not equal to.	LHS tree structure and data is not equal to RHS tree structure and data
<=	Less than or equals.	LHS # of nodes <= RHS # of nodes
<	Less than or equals.	LHS # of nodes <= RHS # of nodes

Logical Operators

Logical operators test the truth value of a pair of operands. The following logical operators `&&` (logical and) and `||` (logical or) are binary operators and left associative. They take two operands of type boolean, and return a boolean value. The `||` operator supports short-circuit

evaluation. `!` is a unary operator and appears on the left side of the operand. The type of the operand must be of type boolean and return type is also a boolean value.

Operator Precedence

The following is a list of expressions, presented in order of highest precedence first. Sometimes two or more operators have equal precedence; all those operators are applied from left to right.

```
()
function calls % @
* / mod
+ - --
> < >= <=
== !=
!
&&
||
=
,
```

Statements

Except as indicated, statements are executed in sequence.

Expression Statement

Most statements are expression statements, which have the form:

expression `;`

Compound Statement

So that several statements can be used where one is expected, the compond statement is provided:

compond-statement:
`{` *statement-list* `}`

statement-list:
statement
statement, statement-list

Conditional Statement

The two forms of the conditional statement are:

`if (` *expression* `)` *statement*
`if (` *expression* `)` *statement* `else` *statement*

In both cases the expression is evaluated and if it is `true` the first substatement is executed. In the second case the second substatement is executed if the expression is `false`. As usual the “else” ambiguity is resolved by connecting an `else` with the last encountered elseless `if`.

While Statement

The `while` statement has the form:

```
while ( expression ) statement
```

The substatement is executed repeatedly so long as the value of the expression remains `true`. The test takes place before each execution of the statement.

For Statement

The `for` statement has the form:

```
for ( expression_1 ; expression_2 ; expression_3 ) statement
```

This statement is equivalent to:

```
expression_1
while ( expression_2 ) {
        statement
        expression ;
}
```

Break Statement

The statement:

```
break;
```

causes termination of the smallest enclosing `while` or `for` statement; control passes to the statement following the terminated statement.

Continue Statement

The statement:

```
continue;
```

causes control to pass to the loop-continuation portion of the smallest enclosing `while` or `for` statement; that is to the end of the loop.

Return Statement

A function returns to its caller by means of the `return` statement, which has one of the forms:

```
return;
return ( expression );
```

In the first case no value is returned. In the second case, the value of the expression is returned to the caller of the function. If required the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

Functions

Function Definition

The Lorax language supports user defined functions. Every function declaration must be followed immediately by the definition of that function. Every function declaration must begin by specifying the return type of the function. The return type is followed by an identifier and comma-separated formal parameters enclosed within parentheses. A function may have any number of parameters, and all parameters are passed by value. The implementation details of the function follow immediately within braces. Every function must have a single return statement that returns a value consistent with its return type. A function is called using its identifier followed by its parameters in parentheses separated by commas. If there are no required parameters, the function is called using its identifier followed by empty parentheses. Lorax does not support function overloading. Here is an example of a user-defined function in Lorax:

```
int square(int x) {
    return x * x;
}

int main() {
    int x = 4;
    int s = square(x);
    return 0;
}
```

main Function

In Lorax there is an entry function where the program starts. There must be one main function and should be defined like this:

```
int main() {
      statement-list
}
```

Built-in Functions

print Function

The print function provided accepts one argument of any of the Lorax data types. Presenting `print` with any of the primitive types will print the type in its most natural form. Presenting print with a tree argument will print the tree in a kind of debug format unless the data type for the tree is of 1-degree `char` type in which case it will print a string. Examples below:

```
print("hello, world"); // will print hello, world
print(3); // will print 3
print(3.14); // will print 3.14
print('a'); // will print a
tree <int>t(2);
t = 1[2, 3];
print(t); // will print 1[2, 3]
```

root Function

The root function takes a single tree argument. The return value of the function is the parent of argument. Example below:

```
tree <int>grandFather(2);
grandFather = 1[2, 3[4, 5]];
tree grandChild <int>(2);
grandChild = (t%2)%0; // referencing the child with value 4
tree middleChild <int>(2);
// middle refers to node with value 3
middleChild = root(grandChild);
```

degree Function

The degree function takes a single tree argument. The return value of the function is `int` type. The function returns the defined or inferred degree of the tree. Example below:

```
print(degree(3[4, 5])); // prints 2
```

Scope

Lorax is closed and statically scoped. Local primitive types are passed to their functions by value. Tree identifiers hold a reference to their tree structure and the tree reference may be passed from function to function. Tree objects are allocated at run time and deallocated when there is no tree identifier is in scope and referencing them.

Sample Programs

Depth First Search

```
bool dfs(tree <int>t, int val) {
      if (t == null) { return false; }
      if (t@) { return true; }

      int child;
      bool match;
```

```
      for (child = 0; child < degree(t); child = child + 1) {
            if (t%child != null) {
                  match = match || (t%child@ == val) || dfs(t%child, val);
                  if (match) { break; }
            }
      }

      return match;
}

int main() {
      tree t = 1[2, 3[4, 5]];
      if (dfs(t, 3)) {
            print("found it\n");
      } else {
            print("it\'s not there\n");
      }
}
```

Hello World

```
int main() {
      string s = "Hello, ";
      tree t = `,'[` `['w'[`o'[`r'[`l'[`d']]]]]];
      print(s + t + "\n");
}
```

Euclid's GCD

```
/*
 * As GCD does not involve trees, this
 * algorithm is almost identical to C.
 */
int gcd(int x, int y){
    int check;
    while (x != y){
        if (x < y){
            check = y-x;
            if (check > x)
                x = check;
            else
                y = check;
        } else {
            check = x-y;
            if (check > y)
                y = check;
            else
                x = check;
        }
    }
```

```
    return x;
}
```