Tree++ LRM

Allison Costa (arc2211)
Laura Matos (lm3081)
Jacob Penn (jp3666)
Laura Smerling (les2206)

October 15th, 2018

Language Reference Manual

How to Read This Manual To facilitate the explanation of various language features, both prose and code are used throughout the manual.

All prose is written using this font, while
(all code is written using this font. *)*

0 Introduction

Tree algorithms are relevant to AI, CS theory and data manipulation. Tree++ is a language that uses the underlying data structure of a tree in order so simplify complex tree based algorithms. 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.

1 Lexical Conventions

If a sequence of characters can be split into tokens ambiguously, the one with the longest first token will be adopted.

Whitespace

Whitespace consists of spaces, tabs, and newline characters. Whitespace characters separate tokens, but are otherwise ignored by the compiler.

Comments

Comments begin with the character combination *(* and end with *).* Any text in between the beginning and end of a comment is ignored by the compiler. Comments can be nested and multilin**e.**

Identifiers

An identifier is any sequence of letters, numbers, or the underscore character _. Identifiers are case sensitive and must not begin with a numerical digit.

2 Literals

Boolean Literals

A boolean literal can take on the value true or false.

Integer Literals

An integer literal is a sequence of digits, representing a number in base 10.

Floating-point Literals

A floating-point literal consists of two sequences of digits separated by ., a decimal point. The sequence to the left of the decimal is the integral component, while the sequence to the right is the fractional component. The integral component must not be empty but the fractional component may.

String Literals

Any sequence of characters surrounded by double quotes (excluding the " double quotation character) constitues a string literal. Certain characters may also be expressed with an escape sequence:

\n newline character
\" double quote
\' single quote
\t tab

Tuple Literal

The notation (expr1 ,...,exprn) represents a list literal with the condition that all expressions in the list are of the same type. For example:

*(2/1, 7*60, 8-2, 4+5)*

Node Literal

A node can be expressed as [expr1, expr2] where expr1 corresponds to the parent of the node and expr2 corresponds to the data. There are no declarations without assignment.

```
node<expr1> newNode = [expr2, expr3];
```

Keywords

The following is a list of reserved keywords and may not be used otherwise:

int char void range true
false bool if else
float for while parent
func return empty node
print NULL data break

3 Data Types

Tree++ uses a variety of standard data types from C, with an addition of type that is mathematically expressive, such as node. Functions are first-class objects in Tree++, so a function type (func) is also present. The operators that can be used with each datatype are explained in the next section on expressions and operators.

int

We support a 32-bit integer data type, int. It might be declared as:

```
int foo;
int bar = 5;
```

float

The float type is a floating-point data type used to store real values. It might be declared as:
float foo;
float bar = 5.5;

bool

The bool type is a boolean type taking on two values, true and false. It might be declared as:
bool bar = false;

string

The string type is a type used to store and manipulate strings of ASCII characters. It might be declared as:
string foo;
string bar = "Hello World!";

Tuple

A tuple is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are immutable. In order to initialize a tuple you must first declare a type.
tuple<type> tupleName = (expr1, expr2 ...)

To create a tuple with a single element, you have to include the final comma:
tuple<int> tupleName = (int1,)

Most list operators also work on tuples. The bracket operator indexes an element:
tuple<int> tupleName = (1,2,3,4,5,6)
*print(tupleName[0]); (*1*)*

Comparing tuples

The comparison operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big)
(0, 1, 2) < (0, 3, 4)
(*true*)
(0, 1, 20000) < (0, 3, 4)
(*true*)
The sort function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on. This feature lends itself to a pattern called DSU. (Decorate, Sort, Undecorate)

It is required to enclose tuples in parentheses to help us quickly identify tuples when we look at code. Because tuple is the name of the constructor, you should avoid using it as a variable name.

len(a) Returns the length of a (this is a constant-time operation, because list length is stored and updated internally
Note that tuple concatenation does not mutate the tuple arguments, but creates an entirely new tuple whose elements are the copies of the originals.

Note that list concatenation does not mutate the list arguments, but creates an entirely new list whose elements are the copies of the originals.

void
The void data type represents an empty type, behaving just as it does in C.

NULL
NULL represents a null node. It can be returned from one of the built-in node functions, which are mentioned below.

func
Functions in Tree++ are first-class objects: they can be passed as arguments and returned from other functions. Therefore, Tree++ supplies a function type, func. Refer to subsection 9 for more in-depth information on functions.

node
The node data type is a generic types that can be used in the following manner:
 Node<type> foo;
where foo can only hold values of type T. A node consists of parent, a list of children and a data value.

4 Manipulation of Nodes

Built-ins

We provide built-in operations in order to allow for easy manipulation of the node type:
Node<type> newNode = [parent, data] Declares a new node with parent and data
newNode.parent Returns the parent node
newNode.data Returns the data stored in the node
newNode:n Returns the nth node of the tree according to BFS ordering
The head of the tree is 0, the first child is 1 ...n If there are no nodes in position n the language will throw an error
newNode:n(newNode:i) Returns the grandchild in position i according to bfs where the head is reset to position n. This allows index access of sub-trees.

5 Node Internals

5.1 Mutability and References

Nodes are mutable data types, so the members of an instance of each type (node.parent, node.data, node:n) can be updated. This allows for nodes, edges, and graphs to be easily manipulated after construction.

Node variables are treated as data values. Therefore, one variable must refer to the same node. Assignments, for instance, will act as tree copying and data swapping:
node<string> newNode = [newNode:3, “hello”];
node<string> secondNode = newNode;
newNode = [newNode: 3, “test”];

*print(newNode:3.data); (*test*)*

```
print(secondNode:3.data); (*hello*)
```

All behavior must be clearly defines as the results will change for every assignment.

5.2 Indices

Integer indices are used to identify location of nodes with in trees based on the head of the node. The ordering of the nodes is BFS where the head node is 0 and the first child is 1. If a new child is added in the center of the tree the indices will shift. Nodes are not added based on index they are added by referencing parents and shifting children. Consider the example below:

```
print(newNode:2.data); (*c*)

  a
 /  \
b    c
char in = 'd';
newNode.parent >> + in;
print(newNode:2.data); (*b*)
    a
  / / \
d  b  c
```

5.3 Invalid Nodes

When a node is added to a tree with a parent that does not correspond to a node in the tree, an error will occur. For example:

```
  a
 /  \
b    c

node<char> newNode = [newNode:3, 'e'];
//error this is an invalid parent argument, check your index
```

Hence, it is recommended to make sure that a tree contains all necessary parent nodes before adding a new child node.

5.4 Adjacency Linked Lists

Adjacency link lists are internally updated within a tree. Each time an child is added or removed, the adjacency linked lists of the corresponding children will be updated accordingly.

5.5 Removing Parent Nodes from a tree

When a parent node is removed from a tree, all children involving that node will also be removed from the tree.

6 Expressions and Operators

We describe expression in order of precedence below, starting with those with the highest level of precedence. Any expression can be disambiguated by parentheses. Consider the following example:
*10 * 3 + 4 ;(* 34 */)*
*10 * (3 + 4); (* 70 *)*
Also note that each unary and binary operator is only allowed for the data types which have it defined (definitions of various operators can be found in the previous subsection).

7 Primary expressions

Identifiers:
Identifiers are named variables including function, and they have a strict type assigned at declaration. Literals The literals are as specified in the Lexical Convention subsection.

8 Operators

Node Operators:
Node operators are very important for shifting the information from one direction to another. A node operator used multiple times groups left to right. Combining different node operators in one statement must have the parent node first and the preceding children second. The node operators are left associative when they are stacked. Complete the operation to the left first and then continue to the preceding operations. Expr1 is the parent expression and Expr2 is the child.

- ^^
  Deep swap node and node (can be parent and child or child and child)
  - o If you deep swap node and parent then the child becomes the new parent node and the parent becomes a child of the original child. All children of the original parent remain attached to the parent, which is now the new child of the original child.
  - o This is present to write percolating logic easily
  - o expr1 ^^
    - The first child is swapped with the parent
- <<
  - o Shift children left
  - o If you want to specify the exact number of children to shift left you can add an index in [] at the end of the operators used. The index is counted from left to right starting at 0.
  - o expr1<<
    - If the children are shifted without an addition or subtraction than the first child becomes the last child and the last child becomes the second to last child.
- >>
  - o Shift children right
  - o If you want to specify the exact number of children to shift right you can add an index in [] at the end of the operators used. The index is counted from right to left starting at 0.
  - o expr1>>
    - If the children are shifted without an addition or subtraction than the first child becomes the second child and the last child becomes the first child.

Shift and swap operators explained:

```
  a
/   \
b    c
```

expr1 is the parent, expr2 is the input:

expr1 <<+expr2 shift children left and add node in tree last

char new_data = ‘d’;

root(newNode)<<+ new_data;

```
   a
 / /  \
b c    d
```

expr1 >>+ expr2 shift children right and add node in tree first

char new_data2 = ‘e’;

root(newNode)>>+ new_data2;

```
    a
  / \  \  \
e   b c   d
```

expr1 <<- shift left and delete last node in tree

root(newNode)<<-;

```
   a
  / \  \
e   b  c
```

expr2 >>- shift right and delete first node in tree

root(newNode)>>-;

```
    a
   /  \
 b    c
```

expr1<<^^ shift left and swap parent node and child in tree

root(newNode)<<^^;

```
      c
     /
    a
   /
 b
```

expr1 >>^^ shift right and swap parent node and child in tree

*root(newNode)>>^^*

```
c
 \
 a
 /
b
```

- If there is no node to the left to swap, the only remaining child will swap with the parent

```
    a
   / \
 c    b
```

- If there are no children then the program will throw an error when one tries to shift

expr1 is the parent, expr2 is the input, expr3 is the counter to specify exact location of transition:
expr1 <<+ [expr3] expr2 shift children left and add node in tree last
expr1 >>+ [expr3] expr2 shift children right and add node in tree first
expr1 <<- [expr3] shift left and delete last node in tree
expr2 >>-[expr3] shift right and delete first node in tree
expr1<<^^ [expr3] shift left and swap parent node and child in tree
expr1 >>^^ [expr3] shift right and swap parent node and child in tree
If expr3 is not present in the tree then the program will throw an error

Negation Operators:
For an expression, expr, which is of type int or float, (-expr) returns the negated value. On the other hand if expr is of type bool, (!expr) returns the negated value.

Assignment Operators

Variables are assigned using the = operator. The left hand side must be an identifier while the right hand side must be a value or another identifier. The LHS and RHS must have the same type, as conversions or promotions are not supported. The variable assignment operator groups right-to-left.

In the format below, l_val is any identifier or tuple element (e.g. lst[0]) and r_val is any expression. The following assignment operators are right associative.

Assigns r_val to l_val then the entire expression takes on the new value of l_val:
(l_val = r_val)
Computes l_val * r_val and assigns it to l_val then the entire expression takes on the new value of l_val:
*(l_val *= r_val)*
Computes l_val / r_val and assigns it to l_val then the entire expression takes on the new value of l_val:
(l_val /= r_val)
Computes l_val % r_val and assigns it to l_val then the entire expression takes on the new value of l_val:
(l_val %= r_val)
Computes l_val + r_val and assigns it to l_val then the entire expression takes on the new value of l_val:
(l_val += r_val)
Computes l_val - r_val and assigns it to l_val then the entire expression takes on the new value of l_val:
(l_val -= r_val)

Note that empty tuple literals ([]) must have a declared tuple type. Hence, the following is valid:
tuple<int> lst = ();
node<string> newNode = [];
while the following is now:
tuple<> lst = ();
node<> newNode = [];

Multiplicative Operators:
The multiplicative operations are left associative and the return value matches the type of the operands. The operands must match type. The * and / operations exist for int and float types, while the * operation only exists for int.
Computes expr1 * expr2 and the entire expression takes on the resulting value:

*(expr1 * expr2)*
Computes expr1 / expr2 and the entire expression takes on the resulting value:
(expr1 / expr2)
Computes expr1 % expr2 and the entire expression takes on the resulting value:
(expr1 \% expr2)

Additive Operators :
Additive operators are left associative and the return value matches the type of the operands. The operands must match type. The + operation exists for int, float, string (concatenation) and tuple types while the - operation only exists for int and float types.
(expr1 + expr2)
Computes expr1 + expr2 and the entire expression takes on the resulting value.
(expr1 - expr2)
Computes expr1 - expr2 and the entire expression takes on the resulting value.

Relational Operators:
The relational operators are left associative and evaluate to a boolean. These operations exist for int and float types.
Compares expr1 to expr2 and if expr1 is strictly less than expr2 it returns true; otherwise false:
(expr1 < expr2)
Compares expr1 to expr2 and if expr1 is strictly greater than expr2 it returns true; otherwise false:
(expr1 > expr2)
Compares expr1 to expr2 and if expr1 is less than or equal to expr2 it returns true; otherwise false:
(expr1 <= expr2)
Compares expr1 to expr2 and if expr1 is greater than or equal to expr2 it returns true; otherwise false:
(expr1 >= expr2)

Equality Operators:

The equality operators are left associative and evaluate to a boolean. These operators exist with any type, and can also be used with NULL to test whether a variable is null. For strings, == is structural (lexicographical) equality. For nodes and tuples, == is physical equality.
Compares expr1 and expr2 and returns true if the two expressions have the same value:
(expr1 == expr2)
Compares expr1 and expr2 and returns true if the two expressions do not have the same value:
(expr1 != expr2)

Logical Operators

Logical operators on booleans are left associative and evaluate to a boolean. Returns false if expr1 and expr2 are both false; otherwise returns true:
(expr1 && expr2)
Returns true if expr1 and expr2 are both true; otherwise returns false:
(expr1 || expr2)

Function Expression

In Tree++, functions are treated as a kind of expression. We allow for inplace anonymous functions as follows:
func name<return_type>(arg1, arg2,...){

```
        statement1;
        statement2;...
}
```

The arguments, arg, are of the form type identifier indicating the type and local name of the function's arguments. The statements, statement1, statement2... are specified in the next subsection and ends in a return statement returning a value of the specified return type. For more information, refer to the section dedicated to functions.

Function Calls

A call to a function is an expression that takes on the same value as the return value of the function called with the given arguments. For example:

```
func add<int>(int a,int b) {
 return a+b;
};
 print(add(2, 4)); (* 6 *)
```

9 Statements

9.1 Expression statements

A statement is defined as any expression followed by a semi-colon. Below are several examples:

```
31;
1 == (3-2);
 x++;
y = 10;
```

The first two statements are evaluated but are generally not useful since they have no side effect. On the other hand, the last two will alter the variables x and y respectively.

9.2 Control Flow

9.3 Conditionals

If-else statements use the following formats:

```
if (condition) {statements}
if (condition) {statements} else {statements}
```

Each condition is evaluated to either true or false, and the corresponding set of statements is executed accordingly. The braces are optional if there is only one statement; a sequence of statements must be enclosed within corresponding braces.

9.4 Loops

C-style while loops and for loops are provided, such as the following:

```
while (condition) {statements}
 for (initialization; condition; update) {statements}
```

They can either be followed by a single statement to be looped, or by a sequence of statements enclosed within brackets. Tree iteration over nodes and children is also allowed, using "for each" loops, which take the following format:

for value in range(aug1, aug2){statement} - Iterates through the nodes based on the bfs numbers in the tree

for value in newNode{statement} - Iterates through all of the nodes in the entire tree according to BFS

for x in functionOrder() range(aug1, aug2){statement } - 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 functionOrder() newNode{statement } - One can write their own iterative function if they desire to iterate through all the nodes in the entire tree
The braces are optional if there is only one statement; a sequence of statements must be enclosed within corresponding braces.

9.5 Block Statement

A block statement is a list of zero or more statements, enclosed within corresponding braces.

```
{
int x;
 x = 5;
 }
```

As in C, blocks are often used as the body in conditional statements or loops and can be nested inside a bigger block. See subsection 8 for more information on rules on scope related to block statements.

```
{
        int x = 5;
{
int x = 2;
x == 5; (* this evaluates to false *)
 }
x == 5; (* this evaluates to true *)
}
```

9.6 Return statement

Return statements take the form of:
return expression;
The return expression can be any single expression or omitted. The type of the expression must match the return type of its enclosing function.

10 Functions

10.1 Function definition

The general form for function definition is:
 func function-name<return_type>(arguments) {statements, return-type return};
where return-type can be of any data type discussed above. If no value is returned from the function, then one can use the return type void. Arguments are separated by comma, with each having its own type and identifier. The number of arguments can be zero or more. An example of a function definition is:
func add<int>(int a, int b)

```
{
return a + b;
};
```

Note that return-type and type of arguments need to be given.

10.2 Calling functions

A function can be called using its name and any necessary parameters. The number and type of parameters must match with the function definition. As with function declaration, function parameters are separated by a comma. Continuing with the example above, we can call add as follows:

```
add(3, 6); (* this will return 9 *)
```

11 Program Structure and Scope

A Tree++ program must exist within a single source file. As with scripting languages, Tree++ programs start execution from the beginning of the file.

11.1 Scope

Tree++ follows scoping rules similar to those in C. Variables declared in the outer body of the program (i.e., not within any enclosing set of braces) are considered global and can be accessed throughout the entire program, but not outside the file in which they reside. Local variables can only be accessed within the block in which they were declared, and will shadow any existing variables with the same names in the outer scope. Formal variables of functions are only visible within their corresponding functions.
A declaration is not visible to statements that precede it. For example:

```
int x = y; (* error: y is not yet declared at this point *)
int y = 6 ;
```

13 Built-in Functions

We include the following built-in functions:

13.1 Printing

The print function prints the formal argument (either a string, int, or float) to standard output.

```
print("hello"); (* prints "hello" followed by a newline *)
print(5); (* prints 5 followed by a newline *)
print(3.5); (* prints 3.5 followed by a newline *)
```

13.2 Length

The length function returns the integer length of a tuple.

```
len ((1,2,3)); (* 3 *)
```

13.3 Node functions

The built-in node functions are explained in more detail under the subsection of the manual on the graph datatype. They are listed below:

```
empty(node)
height(node)
depth(node)
root(node)
```

The built in tuple functions are explained more in the tuple subsection of the manual. They are listed below:

tuple.count()
tuple.index()
len(tuple)
print(tuple)