PLT Project Proposal: The Tree Manipulating Language

Jiabin Hu (jh3240)
Akash Sharma (as4122)
Shuai Sun (ss4088)
Yan Zou (yz2437)
(Dated: September 28, 2011)

I. MOTIVATION

Tree is one of the most fundamental data structures not only in computer science, but also in real life. The applications built on it range from data storing and searching to coding and routing algorithms. However, in most modern programming languages, representing a tree requires pointers or references, which often leads to bugs that are hard to catch. Furthermore, codes on tree manipulation are usually difficult to read, since they hardly reflect the abstracted operation. Therefore, we plan to design a new language, the Tree Manipulating Language (TML), specifically for manipulations on trees. The goal of the language is to provide more efficient and user-friendly programming methods to implement operations on trees.

II. INTRODUCTION

In TML, we introduce a new type named type ***Tree***. As our language is specifically designed for tree programs, incurring a type ***Tree*** will make it easier to program. Basic operations to program on a tree are provided in our language, such as tree construction, adding tree node, referring to father, referring to root data, etc. Programmers could both use the provided operation or define new functions to manipulate trees.

The highlight of our language is that, everything except primitive types is regarded as a tree, just like everything in Java is an object. Noted that every child of a tree is the root of its sub-tree. In TML, we regard all nodes in a tree as sub trees which are of the same type as the original one. When applying operations on a tree, we recursively apply the operations on sub trees and the root. The recursive feature of trees is the reason for this language feature. For operations on a single node, reference to the node is available by referring to its sub tree.

In TML, users can define new types of tree inherited from the basic type ***Tree***. The degree and storage field can also be user-defined. Users could use this feature to build their own trees and even queues, stacks, lists, etc.

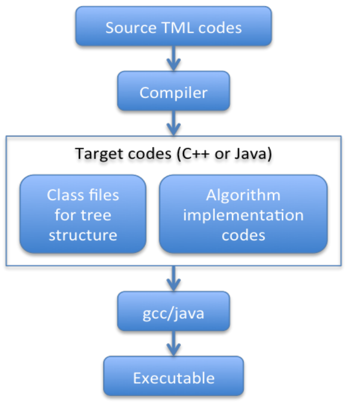
Source TML codes
Compiler
Target codes (C++ or Java)
Class files for tree structure
Algorithm implementation codes
gcc/java
Executable

FIG. 1: TML Compiling Process

TML compiles the source codes and translate them into C++ or Java source code, which is then compiled by gcc or java into executable files. The C++ or Java codes in the middle of this process can also be used by programmers in other C++ or Java programs. Figure 1 shows the compiling process of TML.

III. SAMPLE CODE

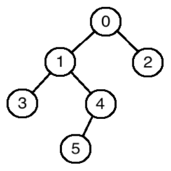

FIG. 2: Tree constrution in sample code

```
// Type definition of MyTree_t
// with Degree 2 and Storage of int.
Type MyTree_t <2, int>;

// Define six trees with initial values for root nodes.
MyTree_t a(0), b(1), c(2), d(3), e(4), f(5);
// Build a tree with six sub trees.
a -> ( b -> (d, e -> (f, _)), c);

// An example of inorder traversal.
forEach child in a by inorder do
{
    print(child);
}
```

IV. SYNTAX DRAFT

A.Basic

Every statement should end with semicolon. Comments can be written as follows:

```
//This is a single-line comment
/*
This is a multi-line comment
*/
```

B.Types

1. int
   Type of integers.
2. float
   Type of floating numbers.
3. char
   Type of single characters.
4. string
   Type of character sequences.
5. Tree
   A tree structure containing a node and connections to its subtrees. Before using this type, a type definition should be in the following format to indicate the degrees, the name index of subtrees and the type of value of each nodes:

   ```
   Type MyTree\_t \textless 2[left, right], int\textgreater
   ```

   This means MyTree_t is a type of tree whose degree is at most 2 with the first subtree called left and second subtree called right. Also, each node of this tree contains an integer.

   [left, right] part is optional. By default, the first subtree could be simply referred by number 0, and the second by number 1, and so on.

The keyword ***Tree*** can be used directly as a type, which means no restrictions on degree and type of node values.

C.Expressions

- Basic Operators:

expr1 + expr2	Add two numbers, or concatenate two strings
expr1 - expr2	Subtraction
expr1 * expr2	Multiplication
expr1 / expr2	Division. If the value of expr1 and expr2 are both integers, the result is the integral part of the result.
expr1 % expr2	The remainder part of the division
<, <=, ==, <>, >=, >	Comparisons, returns 0 if false, 1 if true.
var = expr	Assignments

- Boolean Operators:

expr1 and expr2	Logic and
expr1 or expr2	Logic or
not expr	Logic not

- Tree Operators:

Tree[name]	Get the subtree by its name index. The name index of subtrees is specified at tree type definition.
Tree[integer]	Get the subtree by its number, which is counted from the left. The number of the first subtree is 0.
Tree ->()	Assign all the subtrees. This assignment can be nested.
@Tree	Get the value of the root node
ˆTree	Get the parent of the subtree

D.Branches

There is only one kind of control statement-if:

```
if expr then
{
   //Statements when expr is 1
}
else
{
   //Statements when expr is 0
}
```

E.Loops

There are three kinds of loops - for, while and forEach:

```
for var = start_num to end_num by step do
{
   //Loop body
}
```

```
while expr do
{
   //Loop when expr is 1
}
```

```
forEach var in tree/string by function_iter do
{
    //each element in a tree or a string produces an iteration.
    //The elements will be iterated according to what is specified in function_iter
}
```

F.Functions

```
return_type function_name (argument lists)
{
    //Statements
}
```

Each item in argument lists contains the type of the argument and its name, separated by commas.