**Abstract**

The project consists of two phases. The first phase implements a **Context Free Grammar Generator.** The generator doesn’t develop Context Free Grammars(CFG) in a conventional way, where the user frames grammar rules from his understanding of the language. Instead, the grammar rules are deduced from the example strings written in the language of the grammar. The second phase involves implementing a **matcher module.** The module checks if the two strings written in the same grammar result in same code execution sequence. As an extension to the matcher module, a partial matcher module has also been proposed, which calculates the extent to which two trees match.

**Introduction**

The concepts of grammars for programming languages and parsers can be used for domains outside Computer Science. For example, two balance sheets can be compared,even if their field are jumbled, if we have a grammar defined for balance sheets and we compare their parse trees. But professionals from non Computer Science domain don’t have the knowledge to write grammars and parsers. The projects attempts to solve the issue by making both Context Free Grammar Generator and Matcher. They can be used by the user without requiring the knowledge of computer science fundamentals.

**Content**

**1. Writing CFGs for sample cases**

Before beginning with the first phase of the project, CFGs were written for some sample cases.

This was done to get a better understanding of CFGs, which was crucial to implement the CFG generator. These four sample cases have been listed below:

**Case 1 : Big O notation**

Notation -> O(F)

F-> 1| F\*nReal | F\* NaturalF | F\*((T)!)F | F\*log(F)F

T -> Natural | T\*n | T+T

Real -> Natural | Irrational

Irrational -> n1/Natural

Natural-> 1|2|3|4|5|6|7|8|9|NaturalWhole

Whole -> 0|1|2|3|4|5|6|7|8|9|WholeWhole

**Case 2: Graphs**

(assuming vertices are represented by whole numbers)

G->({V},{E})

V->W MoreVertices | 𝞊

MoreVertices -> ,W MoreVertices | 𝞊

E -> (W,W) MoreEdges | 𝞊

MoreEdges -> ,(W,W) MoreEdges | 𝞊

W -> 0|1|2|3|4|5|6|7|8|9|0|WW

**Case 3 : Arithmetic Expressions**

A ->Num | (A) |A/A |A\*A | A+A| A-A

Num -> Real| Complex

Real -> W.W |Irrational |W.W **-** | -Real

Complex -> Real + 𝛊Real

W -> 0|1|2|3|4|5|6|7|8|9|WW

**Case 4: Recurrence Relation**

(General form : f(n) = an-1 f(n-1) + an-2f(n-2)+.....+a0)

F(n) -> Polynomial | F(n) +Polynomial\*F(n-H)

H-> 1|2|3|4|5|6|7|8….. n-1

Polynomial -> Real | Polynomial\*n | Polynomial + Polynomial

Real-> W.W | W.W**-**|Irrational| -Real

W-> 0|1|2|3|4|5|6|7|8|9|WW

**2. Making a generalised parser**

Since the further stages of the project required making parser generators for so many grammars, it wouldn’t have been feasible to write a separate parser for each case. So the intention was to arrive at a generalised parser, which with the input of a particular grammar would give parser for the grammar.**Yacc,** along with **Lex** proved to be a useful tool in the quest.

**Yacc :** Yacc is short for *Yet Another Compiler - Compiler* . It is based in C and it accepts LALR(1) grammars. It is a quite flexible tool and the user can use it for various purposes viz. checking the validity of a string in a given grammar, printing the parse tree/ abstract syntax tree.

It could also be extended to doing semantic analysis of a given program. All the grammar rules are written in the yacc file and certain *actions*(execution of code snippets written for that rule)are performed once a rule is detected.

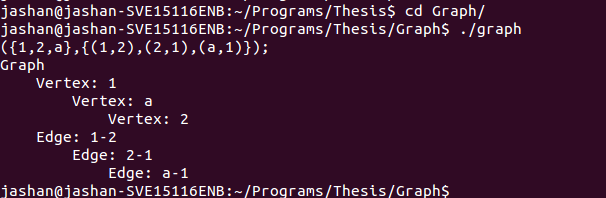
But yacc doesn’t take the code as it’s input. Rather it needs a token of strings (defined in the grammar) as input. The task of tokenizing the code into a stream of tokens is performed by lex.

**Lex:** Lex is a lexical analyzer. It takes in the code and tries to match it with any of the tokens mentioned in the file and returns it to yacc, in case a match is found.

To find out a generalised parse tree generator, they were implemented for 2-3 sample cases.

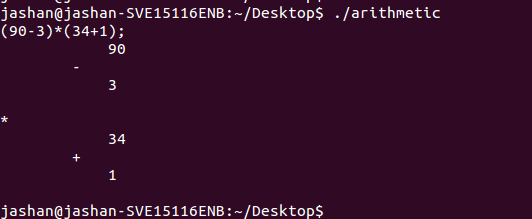
Case 1: Graph of the format (V,W)

(*Refer :* ***graph.l*** *and* ***graph.y*** *)*



Case 2: Arithmetic Expressions

(*Refer :* ***arith.l*** *and* ***arith.y*** *)*



Although there wasn’t much variation across the grammars, it wasn’t possible to find a generalised solution for the following reasons:

1. **Different data types :** The yacc file requires declaration of the data types used in the grammar. SInce the list of data types drastically varies across grammars and it is not possible to compile an exhaustive list of all the data types (user defined data types included), it wasn’t possible to generalise them.
2. **Varying structure of trees :**  Declaration of a common structure for tree node is also not possible, primarily because the arity of it changes for different grammars. Eg. A grammar for arithmetic expressions using binary operators needs binary tree, whereas the root for grammar for graphs containing (V,E,W) would be a trinary node.

Nonetheless, a template for parser generator was arrived upon and the user needs to input the following attributes:

1. **For Lex:**

1. A list of all the tokens with token names

2. In case the token is an identifier, the data type and the method to extract the value of

Identifier

(*Refer :* ***template.l*** *for the template of lex files)*

**2. For Yacc :** 1. Different types of nodes that can be there in the tree

2. Definition of these tree nodes

3.List of all the data types

4. List of tokens

5. Datatype of all the non terminals

6. The grammar

Optional : Name of the starting non terminal

(*Refer :* ***template.y*** *for the template of yacc files)*

**3. Context Free Grammar Generator**

The basic algorithm behind this phase is to do take 5 input strings of the language and find the common patterns in those strings to deduce grammar rules from them. It should be noted that the strings should be as varied from each other as possible so as to cover all the possible rules in the grammar.

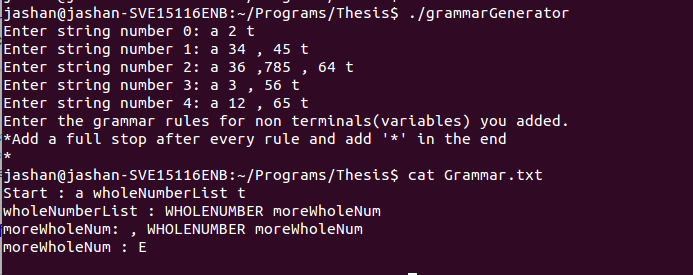
A step by step implementation of the algorithm used is as follows:

(*Refer :* ***grammarGenerator.cpp*** *for the code of the module)*

1. Input five strings from the language the grammar describes
2. Tokenize the five strings, taking *space* as the delimiter  
   (NOTE: The module assumes that all the tokens are space separated)
3. Find common tokens across all the five strings and store them in an array
4. Now, the task is to find out the grammar rules for what comes in between these constants. The module tries to match them with certain patterns like  
   a) A number (whole / decimal)  
   b) A list of numbers separated by a certain separator
5. If no such pattern is found, the user is asked to input the pattern/ rule for what comes between those constants. The user can input one of these patterns and the module will write the rules itself :  
   a) Union of two things  
   b) Concatenation of two things  
   c) A list of something
6. If the input is not from one of the above three patterns, then the user is asked to fully define the rules for what comes in between the constants

The following two examples clarify the working of the module.

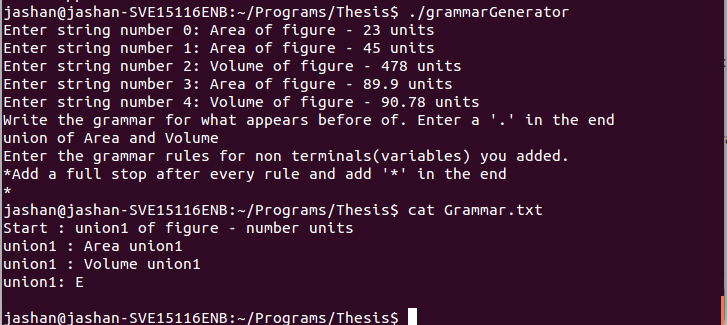
Case 1:



In this case, the generator in the first step identifies that the string the language starts with ‘a’ and ends with ‘t’. Then it tries to find what pattern appears with them. It recognises it to be a whole number list of variable length separated by ‘,’ . So it writes the rules for a whole number list.

Case 2:

In this case, the generator identifies that everything is constant except for the first token and the second last one. While it is able to identify the latter as a number, it fails to do so for the former. So it asks the user to define it further. The user inputs that it a union of ‘Area’ and ‘Volume’ token. After this, it writes the rules for union of two entities.

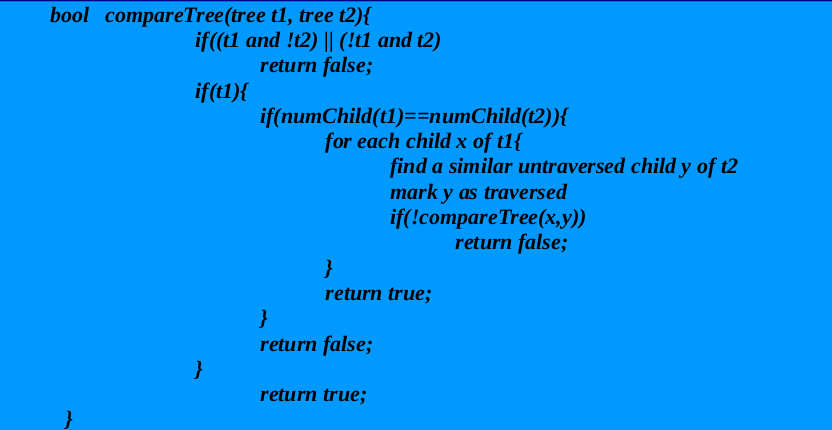


**Limitations:**  The grammar generator has it limitations as follows:

1. Small Subset of Grammars : It works only for small subset of grammars with very few rules. Since it takes only 5 input strings, it is not practically possible to deduce patterns from this small number of strings for large grammars. Moreover, it is able to write grammar rules by itself only for few patterns and it is the task for the user to write rules for whatever doesn’t come under the ambits of these patterns.
2. Space Separated Tokens : Since the strings have been tokenised by keeping space as delimiter, the tokens have to written separated by space. Whereas we see that it is generally not necessary for tokens in programming languages to be separated by a delimiter.

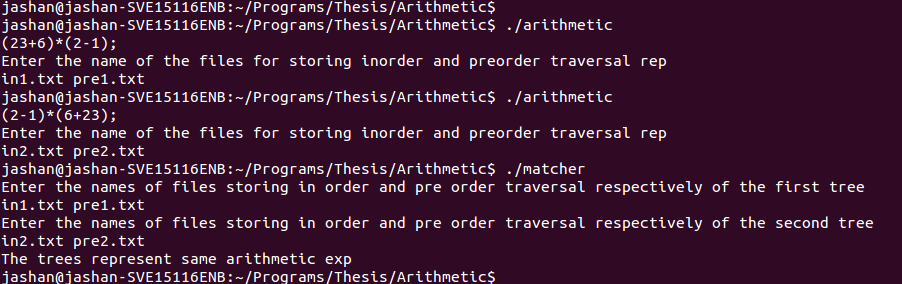
**4. Matcher**

This phase of the project matches two strings of a given language to find out if they produce same/ similar parse trees. The simplest solution is to check if their nodes store the same data and if all the child nodes of one tree match their counterparts in the other. But this rudimentary of trees omits some cases where the trees may be matching. In some cases we need to expand the rule to check for the following conditions:

1. **Commutativity** : As we know that (5+3) and (3+5) are same arithmetic expressions. But if we compare their parse trees as suggested above, they seem to be different. So to handle the cases in which commutativity is allowed, all the child nodes from one trees are mapped to the matching child nodes in the second tree. If one-one matching is possible, the trees match. It is further explained by the following pseudo code:   
     
   

The same has been implemented for arithmetic expressions and in the following example, it outputs **(23+6)\*(2-1)** and **(2-1)\*(6+23)** to be matching.

(*Refer :* ***matcher.cpp*** *for the code)*



b) **Variable Mapping:**  Let’s take the following two examples:

i) int i;

for(i=0;i<5;i++)

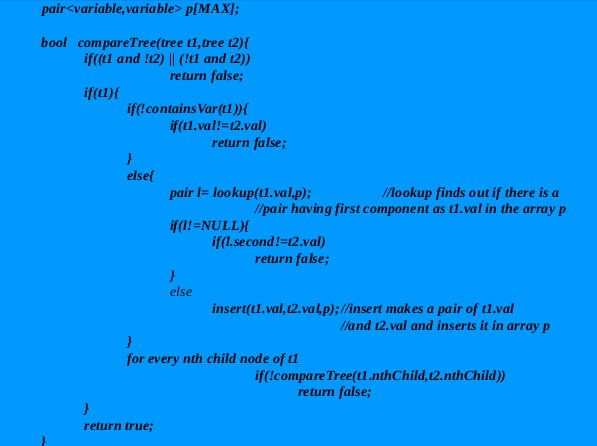
printf(“Hello World\n”);

ii) int j;

for(j=0;j<5;j++)

printf(“Hello World\n”);

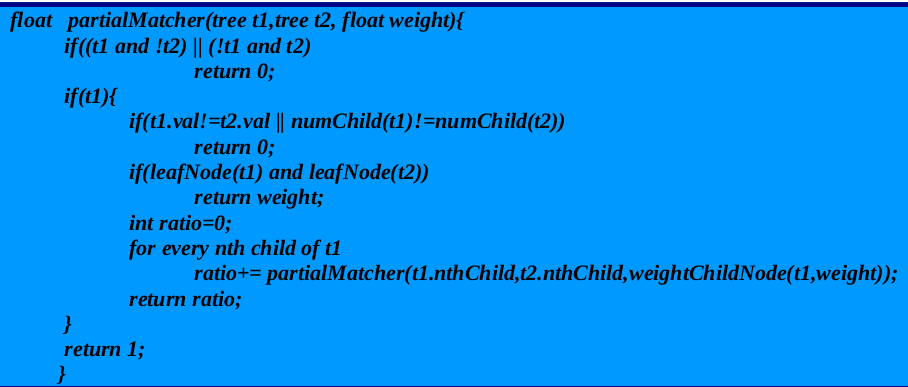
Plain matching of the parse trees of two will yield that these two code segments are different since they use two different variables. So, in this case the variables appearing on similar nodes for the first time are stored as pairs. In their further occurences, they are verified to be occuring together. The pseudo code for the same is as follows:



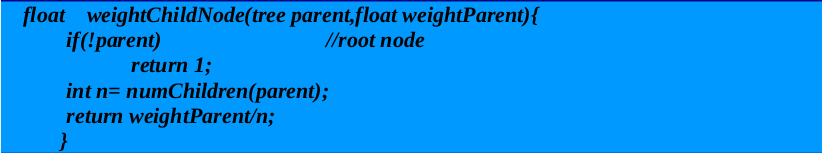
c) **Case Neutrality:**  Languages such as SQL are case neutral. In such cases, the method for comparing the data stored at a node is verified to take into account case neutrality

**5. Partial Matcher**

This part of the project compares two parse trees to find out what percentage of the two trees match. The algorithm it uses assigns weight to each sub node of a tree. If a particular sub tree matches with it’s counterpart, it’s weight is added to the final matching percentage.



Calculating weight of a node: For the sake of simplicity, it assigns weight equally to all the child nodes. In case of the root node, it is assigned weight 1. However, the user at his discretion can specify an algorithm by which weights are assigned non uniformly. The pseudo code for assigning weights equally is :



**6. Conclusion**

The project could implement a Context Free Grammar Generator which works by extracting

patterns from the strings written in the language. Also, it develops an approach to write matcher function for two parse trees. We have seen how this approach could be modified to handle cases of commutativity, case neutrality etc. In the final leg of the project, a method to calculate the percentage of match of two parse tree has been suggested.

**7. Future Scope**

1. The grammar generator module can be extended to identify more patterns so that it deals with a larger subset of grammars
2. An algorithm could be designed, for the partial matcher part to assign weights to sub trees based on their relative importance compared to other sub trees Eg. The statements doing assignment operation can be more important than the print statements. So the sub tree for assignment operation should be assigned more weight than the one with print statement. On the similar lines, a relative order and magnitude of importance can be developed for a language.

**8. References**

1. Context Free Grammars : <https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html>
2. Yacc Introduction :<https://en.wikipedia.org/wiki/Yacc>
3. Lex Introduction : <https://en.wikipedia.org/wiki/Lex_(software)>
4. Yacc Tutorial : <http://dinosaur.compilertools.net/yacc/>
5. Compilation of Yacc and Lex : <http://www.cs.bilkent.edu.tr/~guvenir/courses/CS315/lex-yacc/linux.html>
6. Yacc Video Tutorial : <https://www.youtube.com/watch?v=yTXCPGAD3SQ>

(All the code files for the project are uploaded at :)