I.J. Intelligent Systems and Applications, **2021, 3, 43-65**
Published Online June 2021 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijisa.2021.03.05

Detection and Classification of Cross-language Code Clone Types by Filtering the Nodes of ANTLR-generated Parse Tree

Sanjay B. Ankali
Research Scholar at VTU-RRC & Faculty, Dept. of CSE, KLE College of Engineering & Technology, Chikodi, India,591201
E-mail: sanjayankali123@gmail.com

Latha Parthiban
Department of Computer Science, Pondicherry University, Community College, Lawspet, India-605008
E-mail: lathaparthiban@yahoo.com

Received: 28 January 2021; Revised: 15 February 2021; Accepted: 15 March 2021; Published: 08 June 2021

Abstract: A complete and accurate cross-language clone detection tool can support software forking process that reuses the more reliable algorithms of legacy systems from one language code base to other. Cross-language clone detection also helps in building code recommendation system. This paper proposes a new technique to detect and classify cross-language clones of C and C++ programs by filtering the nodes of ANTLR-generated parse tree using a common grammar file, CPP14.g4. Parsing the input files using CPP14.g4 provides all the lexical and semantic information of input source code. Selective filtering of nodes performs serialization of two parse trees. Vector representation using term frequency inverse document frequency (TF-IDF) of the resultant tree is given as an input to cosine similarity to classify the clone types. Filtered parse tree of C and C++ increases the precision from 51% to 61%, and matching based on renaming the input/output expressions provides average precision of 91.97% and 95.37% for small scale and large scale repositories respectively. The proposed cross-language clone detection exhibits the highest precision of 95.37% in finding all types of clones (1, 2, 3 and 4) for 16,032 semantically similar clone pairs of C and CPP codes.

Index Terms: Cross-language clones, ANTLR parse tree, TF-IDF, cosine similarity, software forking.

1. Introduction

Creating functionally similar programs with or without any syntactical change is called software cloning or code cloning [1]. Accurate cross language clone detector can assist software forking process by locating functionally equivalent codes across multiple language code bases. Based on the editing taxonomy, most of the researchers find four basic types of clones [2]. These clone types are grouped into syntactic and semantic classes:

- **Syntactic**: *Type 1* is also called as exact clones; *Type 2* is also called as renamed clones; and *Type 3* is also called near-miss clones. Syntactic clones are based on different editing taxonomies.
- **Semantic**: *Type 4* includes semantically similar codes that are different in syntax.

In the extensive and milestone literature work of Roy (2007) [3], Dhavleesh Rattan et al. (2013) [4], Ain (2019) [5], and Walker (2020) [6], we find that the last three decades have introduced more than 250 same language clone detection tools. We have witnessed great scalable same language clone detection tools such as *VUDDY* [7], scalable *SourcererCC* [8], and *Siamese* [9], these tools scale well on large repositories, with excellent precision and recall.

However, modern software development approach leverages several different languages for product development. An open-source software project frequently has between 2–5 different languages in product development [10]. Same language clone detection using the scalable tools mentioned above does not apply to the modern programming approach and context for cross-language clone detection. Complete and accurate cross-language clone detection can have significant applications such as bug detection in ported code, refactoring to maintain the quality [13], detecting code smells to understand design [11] and protection of the security of products and can significantly introduce automation in software development process [12]. The Internet contains several million lines of code available

nowadays that make the development process much simpler [14]. For the same reason, developers tend not to start coding from scratch [15]. We find evidence of this reusing of code in a survey conducted by Saini et al. (2015) [16] involving 72 developers; 96% of the developers preferred searching for codes and reusing them in any coding task. The main drawback of using online source code is maintainability and bug propagation [17] or tends to preach software license [18, 19]. Text search engines such as Google cannot provide codes based on the content of source code; hence, there is a need for a software clone extraction system that will use the code as a query and fetch the entire code from the repository by matching the complete code [9]. Existing cross language clone detection such as CroLSim [21] exhibits lowest precision of 64% and the tool FETT [31] do not find clone among C and CPP code. In this paper, we have proposed a accurate and reliable technique to detect and classify the clones of C and CPP with the highest precision of 95.37% in finding all types of clones (1, 2, 3 and 4) for 16,032 semantically similar clone pairs of C and CPP codes to act as proper method recommendation system to assist in software forking process that converts legacy system from procedural language code base to object oriented code base and vice versa. The proposed method can significantly help the software development team which is involved in forking the systems from C-code base to C++ code base and vice versa.

Research questions

1. Assuming there are two repositories of C and C++, when a C code is given as a query, is it possible to extract the functionally equivalent C++ code or vice-versa with clone types classified into 1, 2, 3, and 4 accurately?
2. For a software porting team that intends to convert a large C-based legacy system into a CPP-based system containing N separate C code files, is it possible to fetch the corresponding functionally similar codes of CPP from the repository of C/CPP and vice versa with all the four clone types classified?

This research paper answers both the research questions by proposing a novel approach to detect and classify cross-language code clone types by filtering the nodes of ANTLR-generated parse tree.

The contributions of this research paper are listed below:

1. The proposed work makes use of the capability of ANTLR parser generator to generate the parse trees for both C and C++ programs using the common grammar CPP14.g4 to get near almost parse trees.
2. Selective filtering of generated parse tree using treeListener class of ANTLR helps to perform serialization and eliminates the need of pre-processing step employed by previous clone detection techniques.
3. Vector representation of the pruned parse tree is fed as input to cosine similarity to detect and classify all four types of cross-clones of C and C++.

2. Background and Related work

Basis of Software Clones

Software development process never starts with writing the code from scratch; the process saves the development cost and time by using existing open-source code fragments without breaching the version histories. Device drivers or OS such as Free BSD or Net BSD contain a 15% duplicate code [20]. Software cloning increases the chances of bug propagation, which in turn reduces the software quality and maintenance cost.

In this section, we have briefly defined the clone types and presented the C and CPP examples to justify our understanding of cross-language clone types based on many of the great studies on clone detection done in the past [2].

2.1. Background

Type 1 (Exact clones): Syntactical and semantically similar codes with a change in white space and comments [1].

Type 2 (Renamed clones): Syntactically similar codes with multiple entities edits such as renaming identifier, function, or class name [1].

Type 3 (Near-miss clones): Type 2 clones with addition and deletion of lines create Type 3 clones [1].

Type 4 (Semantic clones): Functionally similar code with a change in syntax.

Definition of cross-language clones

In light of the definition of clones, we can say that two or more codes that produce the same output but implemented in different languages with four clone type variations are called cross-language clones. We have presented few small case studies in this section.

```
1. main()
2. {
3. int x=100,y=200,z;
4. z = x+y ;
5. printf("sum of two numbers=%d", z);
6. }
```

C-code-1

```
1. main()
2. {
3. int x=100,y=200,z;
4. z=x+y;
5. cout<<"sum of two numbers is ="<<z;
6. }
```

CPP-code-1

Except for the structure of the language, both the codes above behave in a similar manner. They declare the same two variables, x and y, and the sum is stored in a variable z, then the result is displayed on the screen. These are examples of Type1 cross clones.

```
1. main()
2. {
3. int a=10,b=20,c;
4. c=a+b;
5. printf("sum of two numbers=%d",c);
6. }
```

C-code-2

```
1. #include<iostream>
2. using namespace std;
3. main()
4. {
5. int a=10,b=20,c;
6. c=a+b;
7. cout<<"sum  of  two numbers is ="<<c;
8. }
```

CPP-code-2

From the above code snippet, it can be said that C-code-2 and CPP-code-2 are Type 1 clones, as the variable names remain same;on the other hand, C-code-1 is a Type2 clone of C-code-2 and CPP-code-2 and vice versa.

Another example is the following looping statement:

```
int fact(int x)
{
for(i=1;i<=n;i++)
fact=fact*i;
printf("factorial of number is=%d",fact);
}
```

C-Code-3

```
int fact(int n)
{
if(n==0)
return 1;
else
return n*fact(n-1);
cout<<fact;
}
```

CPP-Code-3

C-code-3 and CPP-code-3 try to find factorial of integer numbers by using iteration and recursion respectively; hence, these are called Type 4 clones. This work will significantly contribute to the software porting process by identifying all the functionally similar code fragments of C and CPP.

2.2. Related work

There is limited work done in finding cross-language clones. In this section, we have presented some of the significant work related to the subject of this study.

1. A tool called CroLSim [21] is proposed to detect cross-language software similarity for open source software categorization by representing “DreP” files as continuous bag of words to find the correlation between API and Library methods used by each programming language using the KNN algorithm. The results show only 28% precision for searching functionally similar code in a repository which is the major limitation of this work.
2. A semantic cross-clone detection tool named SLACC [22] executes cross-language clone detection based on the input/output behaviour of code. The approach is used in detecting clones across statically typed language, Java, and dynamically typed language, Python. The proposed approach has three limitations ***A***. finds only semantic similarities; ***B***. does not classify cross-language clone types (Type 1,2, 3, and 4); ***C.*** does not support long and complex types of python; ***D***. dead code elimination
3. A recent study on cross-language source code by (Karnalim, 2020) [23] does not handle identifier renaming; and since it works by converting the original code to intermediate code, the proposed approach is computationally complex.
4. AST-based cross-language clone detection was proposed by Perez (2019) [24]. The approach is a semi-supervised machine learning model which is capable of detecting cross-language clones by employing a token level vector generation algorithm and tree-based skip-gram algorithm. This approach does not support more granular clone type classification (type 1, 2, 3, and 4)

5. A tool named CLCDSA is proposed by Nafi (2019) [25], which use action filters to filter out non-probable clones and make the model more scalable. This method has the limitation with respect to more granular clone classifications.
6. A study by Bui (2017) [26] proposed a technique based on bilateral neural networks (BiNN's) to find similarity and differences in structures based on the language AST using BTBCNN. We find two Issues: ***A***. large programs may slow down the training process; ***B***. no results found to prove the type of classification.
7. A binary instruction-based technique was proposed by Hu (2017) [27] to detect the semantically similar functions. This method is limited to detects only semantically similar functions and is vulnerable to code obfuscation.
8. An integrated tool called LICCA [28] is based on the application of the modified longest common subsequence algorithm on enriched concrete syntax tree (eCST) of source code. But this method is limited to semantic clone detection.
9. Cheng (2017) [29] proposed the first-ever technique to process cross-language clone detection for Java and C# languages without using intermediate code that does not share common libraries. The method has following issues.

    - The experiment done on 10 projects shows that the correlation between attributes and *diffs* is weak.
    - Results are encouraging only if mapping clones in *diffs* of change histories to the latest revision.
    - Only cross-language clones of Java and C# are found.

10. Clone detection among the .NET language family was proposed by Al-Omari (2012) [30]. Using SimHash (SimCad), Longest Common Subsequence (NICAD), and Levenshtein Distance, on the Common Intermediate Language of a .NET framework to detect clone pairs. The method has following issues.

Issues:

- The matching algorithm is limited to the information present in boxes.
- Platform dependent.
- No proper results to prove efficiency.

11. The work of Nichols (2019) [31] is the extension of the previous work by Cheng (2017) that finds syntactic similarity using structural and nominal similarity. The method works on the function instead of *diffs* of VCS and currently handles C++, Java, and JavaScript. The method has following limitations.

    - Cannot scale well to get more function mapping.
    - Needs pre-processing work based on the domain knowledge to identify the input to be given.
    - Works only for object-oriented programming languages.

Overview of similarity measures

According to Naumann (2013) [32], there exist many NLP techniques to find the similarity measures of the documents. Table 1 presents various similarity measures that classify the documents based on the data.

Table 1. Similarity measures for document comparison

Sl.No	Name of algorithm	Concept used
1	Smith-Waterman	Edit Based
2	Levenshtein distance	
3	Jaro	
4	Hamming	
5	Jaro	
6	Smith-Waterman-Gotoh	
7	Damerau-Levenstein	
8	Jaro-wrinkler	Token-Based
9	Cosine similarity	
10	Jaccard	
11	Dice	
12	Word/N-gram	
13	Monge-Elkan	Hybrid
14	Soft –TFIDF	

These algorithms work on the string matching or token matching; moreover, because of the fact that these similarity measures do not take into consideration the position of tokens/words in the document, they do not produce

proper results of similarity when applied to the input codes directly. As evidence, we have presented the results of document similarity for the code snippets explained above using Levenshtein distance in Table 2 and similarity using TF-IDF in Table 3. We got the same matching results when applied using other algorithms mentioned in Table 1.

Table 2. matching similarity using Levenshtein distance

Sl. No	Input code-1	Input code-2	Matching percentage
1	C-code-1	C-code-1	100
2	C-code-1	C-code-2	90.41
3	C-code-1	CPP-code-1	27.17
4	C-code-3	CPP-code-3	32.52
5	C-code-2	CPP-code-2	28.26

Levenshtein distance finds the number of edits required to convert one document into another. From Table 2, we find that the similarity between C-code-1 and itself is 100% because both the codes contain same tokens. The results also show that the matching similarity between C-code-1 and C-code-2 is 90.41%, which shows that Levenshtein distance can be used to find Type 1 and Type 2 clones of same language, whereas 27.17% similarity between C-code-1 and CPP-code-1 and 32.52% similarity between C-code-3 and CPP-code-3 proves that Levenshtein distance cannot contribute in finding the Type 3 and Type 4 clones of the same language and any of the four types of clone detection of cross-language. The reason for this type of behavior of the algorithm is that the Levenshtein distance only tries to find the number of edits required to get one document from other and does not take into consideration syntactic information such as the position of each token in the code.

Table 3. Matching similarity using TF-IDF

Sl. No	Input code-1	Input code-2	Matching Similarity (0-1)
1	C-code-1	C-code-1	1.0
2	C-code-1	C-code-2	0.30
3	C-code-1	CPP-code-1	0.14
4	C-code-3	CPP-code-3	0.13
5	C-code-2	CPP-code-2	0.11

From Table 3 we can observe that only for the same code, the matching results are 100%. Even though C-code-1 and C-code-2 differ only in the variable declaration, we get a similarity matching of only 30.2%. The reason for poor matching similarity is that TF-IDF works only on the basis of finding out the number of tokens that are relevant to each document and do not take into account the position of the token in the document, which is the most relevant information needed for detection of cross-language clone or same language clone. This problem can be solved by making them work on a parse tree instead of directly applying matching to the input source codes.

Introduction to ANTLR (Another Tool for Language Recognition)

ANTLR is mainly used to build the lexical analyzers, using which it can easily read and parse documents or binary files for application-specific analysis. Industry uses ANTLR to build the language compilers and frameworks [33]. As of now, there are four versions, with the latest release being ANTLR-4.8-complete, which is available on the ANTLR site. To prove the significance of ANTLR capability, it is worth mentioning that every day; ANTLR parses two billion queries posted on twitter. ANTLR is used by Hadoop's data warehouse and analysis systems. NetBeans IDE, oracle, and lex Machina use ANTLR for various data extraction and analysis purposes.

Finding software clones is not as simple as performing document comparison; we need to match two different codes by considering all the syntactic and semantic issues. The capability of ANTLR as a parser generator can be used to parse the source codes in C, CPP, Java, and other languages by using the respective language grammar made available on Github by ANTLR [34]. We can create our own application by extracting the information from the parse tree through listener interfaces and callback.

Even though ANTLR is written in java, it generates lexer and parser that perfomr respectively lexical and semantical analysis to build the parse tree from the input files. In this research work, we have generated the parse tree by using formal language description called a grammar, along with lexer and parser ANTLR-generated files such as grammar tokens, lexer tokens, BaseListner, Listener, parse tree visitor, and parse tree walker which can be used to process the parse tree as per our needs [33].

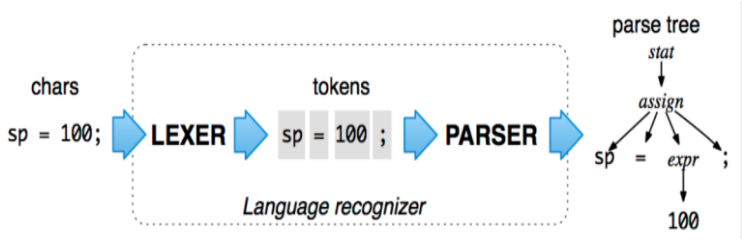

Fig.1. Char stream to produce the parse tree

ANTLR parser generator creates lexer and parser based on the language grammar that later parses the input file based on the grammar. Parse trees are extremely effective data structures that sit between language recognizers and interpreters. They contain all of the input and complete knowledge of grouping the tokens into sentences. The parser generates it automatically for easy understanding [34].

For example, we can write a grammar and include it as file.g4 to parse the simple arithmetic expression like 100+2*34 as follows:

```
grammar EX;
start: (EX NEWLINE)* ;
EX:  EX '*' EX | EX '/' EX  | EX '+' EX | EX '-' EX  | INT
| (EX);
NEWLINE : [\r\n]+ ;
IN     : [0-9]+ ;
```

After installing the latest ANTLR 4.8-v4 version and Java (JDK) classpath set on the system, we followed ANTLR tutorial steps available on the ANTLR site to generate a parse tree for the arithmetic expression 100+2*34.

a. Execute the ANTLR command as "antlr file.g4"; at command prompt, we get various files such as file.tokens, fileBaseListener.java, fileLexer.java, fileListener.java, fileParser.java, file.interp.
b. Compile all the generated java files to get the class file using the java compiler.
c. Run the java org.antlr.v4.gui.TestRig for the input file to obtain the parse tree.

All the three steps have to be performed manually at a command prompt or in memory compilations, which can be done by using the automated APIs of "inmemantlr-tool" which has 14 releases so far and is available on Github (Thome).

Once we perform Step 2 and get the class files, using tree Listener class, we can implement our own application to process the parse tree by creating methods such as getFirstChild(),getLastChild(), deChild(), getSubtree(), replaceSubtree() to access the basic ANTLR tree class.

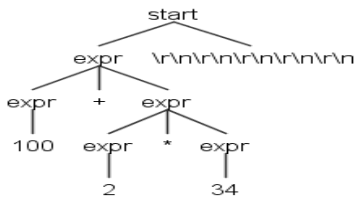

Fig.2. Parse Tree for expression 100+2*34

3. Proposed Cross-Language Clone Detection Approach

In this section, we have presented the architecture of clone detection for C and CPP languages in three phases:

3.1 Repository building and parse tree generation.
3.2 Extracting the nodes of the parse tree.
3.3 Finding the parse tree similarity and displaying clone types.

3.1. Repository building and parse tree generation.

Figure 3 presents the architecture of parse tree generation and node filtering. It is the automated process of running the ANTLR tool through the java application that performs all the manual work explained in Section 2 to build a parse tree. We have used "inmemantlr-tool-1.6" of APIs available at maven central [35] to generate the parse tree in the dot file. Since we intended to build a clone detection model to include only two languages, we used CPP14.g4 that parses

both C and CPP files. The grammar is available on the ANTLR site.

The process begins by providing grammar file CPP14.g4 that contains rules for parsing the C and CPP files. Most of the grammar files for parsing different languages source codes, such as Algol, Basic, Python, and Javascript, are available for use.

ANTLR tool generates various token and java files such as Grammar.tokens, Lexer.tokens, Lexer.java, Parser.java, Listner.java, and BaseListner.java. All the java files are then compiled to get class files. Based on the generation of java class, to provide a glance of the parse tree and corresponding pruned tree, we have shown the pictorial representations. We have submitted the separate file containing JPEG of all the tree structures of our data sets. Figure 5 and Figure 6 show the generated parse tree for input file C-code-1 and CPP-code-1, respectively. Even though both C and CPP codes can be parsed using grammar file CPP14.g4, we find different parse trees for both the codes. Figure 5 and Figure 6 presents the parse tree containing 210 nodes for the C-code-1 and 188 nodes for CPP-code-1, respectively. This change in the parse tree is seen because of the rules for generating the rightmost descending subtrees. We also observe that the initial 158 nodes are generated using same rule for both C and CPP codes that makes same the parse tree up to 158 nodes. Drastic change in the rules for node "shiftexpression " of C code generates lengthy recursive sequence for rightmost subtree derivation that makes the biggest change between C and CPP code.

Figures 11 and 12 show the deviation in deriving the rules for "shiftexpression" for C-code-1 and CPP- code-1, respectively. Grammar CPP14.g4 is freely available [34]. We have presented the rule for node "shiftexpression" for C and CPP code in the following section.

For the C-code-1 nodes, 158 to 168 are generated through the following rules [34]:

shiftexpression : additiveexpression ;
additiveexpression: multiplicativeexpression ;
multiplicativeexpression: pmexpression ;
pmexpression: castexpression;

castexpression: unaryexpression;
unaryexpression: postfixexpression;

postfixexpression:primaryexpression;
primaryexpression:idexpression;
idexpression: unqualifiedid;

Nodes 164 to 181 are generated through the following rules:
postfixexpression: expressionlist ;
expressionlist: Initilizerlist ;

initializerlist: Initializerclause;
initializerclause:assignmentexpression;

assignmentexpression: conditionalexpression ;
conditionalexpression : logicalexpression ;

logicalexpression: inclusiveexpression;
inclusiveexpression: exclusiveexpression;

exclusiveexpression:andexpression;
andexpression: equalityexpression;

equalityexpression: relationalexpression;
relationalexpression: shiftexpression ;

3.2. Extracting the nodes of parse tree

The rule for "shifexpression" will be recursively used to generate nodes 182 to 190, with last rule being *primaryexpression: literal;*

We can streamline the rightmost subtree nodes of C and CPP by eliminating these long recursive rule expressions of C by filtering the nodes that are common between C and CPP code.

We present the rule that makes serialization of the rightmost subtree of C and CPP code.

statement:expression ;
expression:primaryexpression ;

primaryexpression: unqualifiedid|literal ;

Considerations made for filtering the parse tree.

Since we are interested in finding the functional similarity between the C and CPP codes, we applied selective extracting to the ANTLR-generated tree, wherein we excluded all the rules that represent the header of the code. We started filtering the parse tree from the child nodes of rule "statementseq" that corresponds to the function body that contains all the computation statements. Nodes corresponding to the statements of declaration, reading, computation, and writing were only included in the final filtered parse tree.

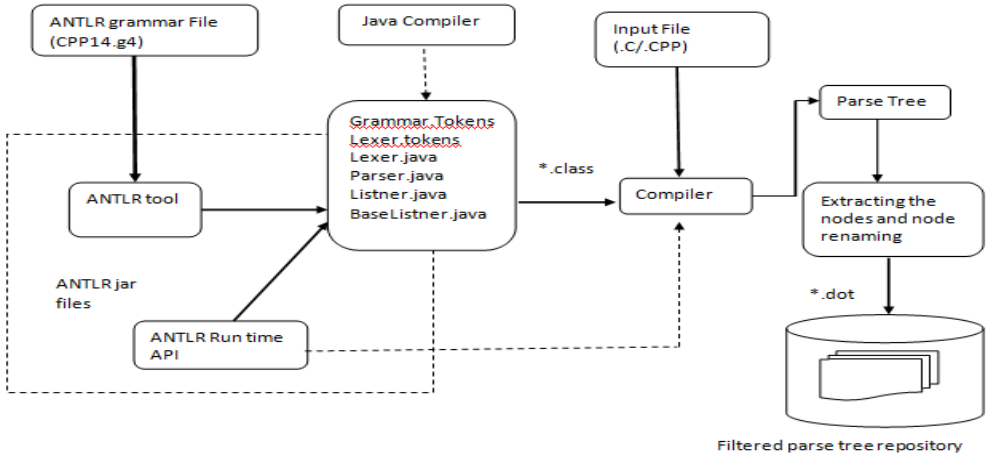

Fig.3. Repository building and parse tree generation

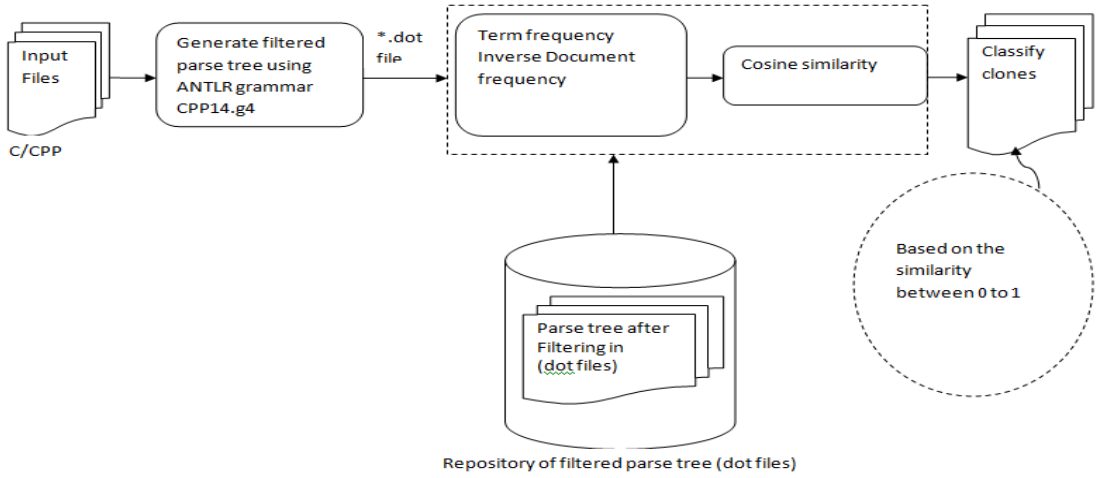

Fig.4. Matching the filtered parse tree similarity using TF-IDF and cosine similarity.

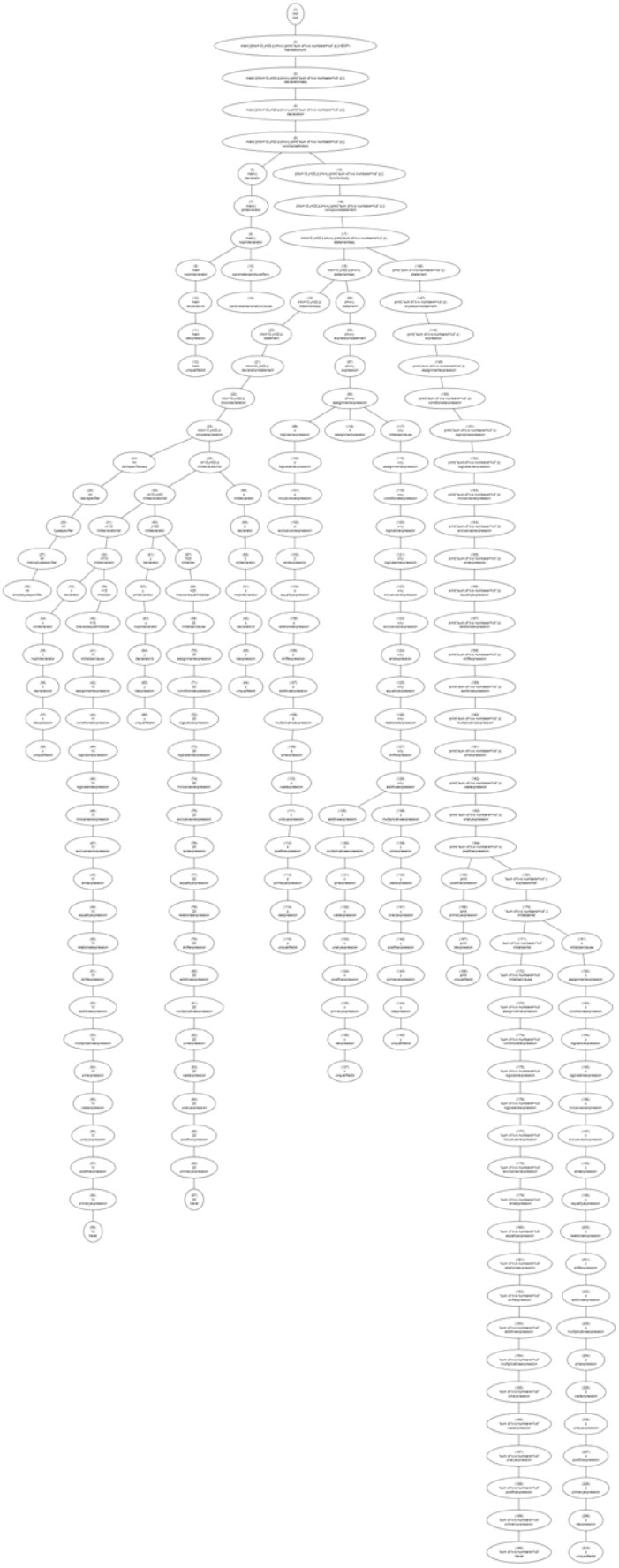

Fig.5. Parse tree for C-code-1

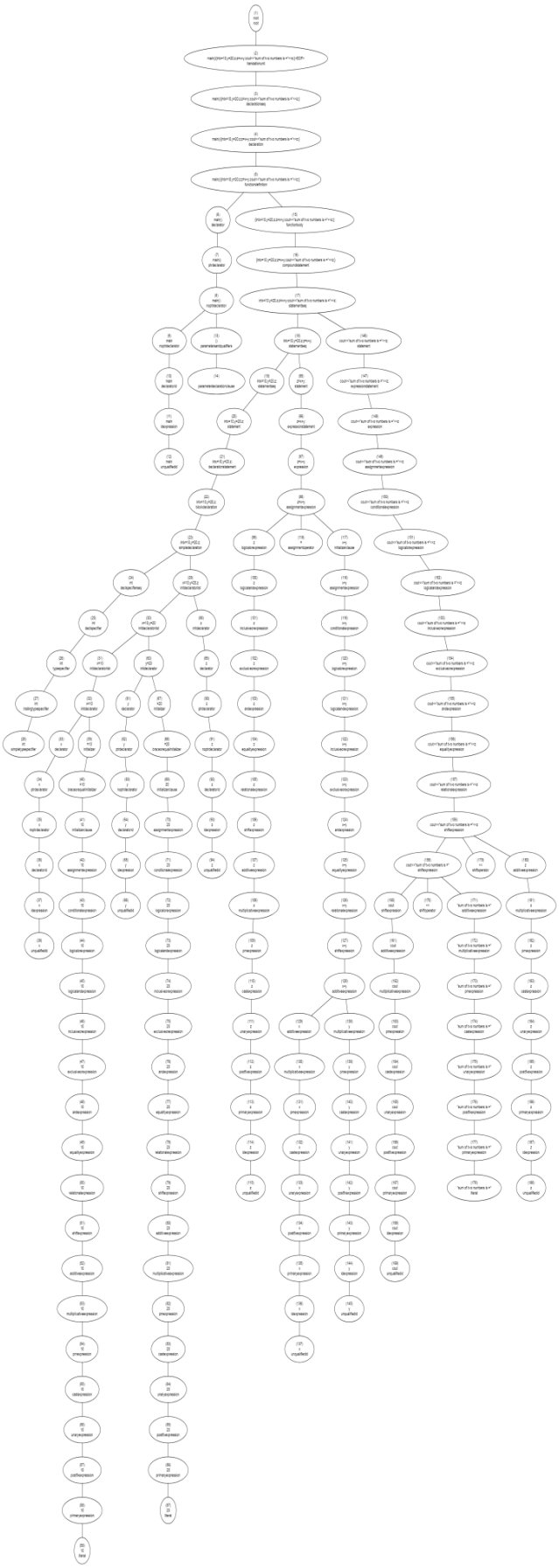

Fig.6. Parse tree for CPP-code-1

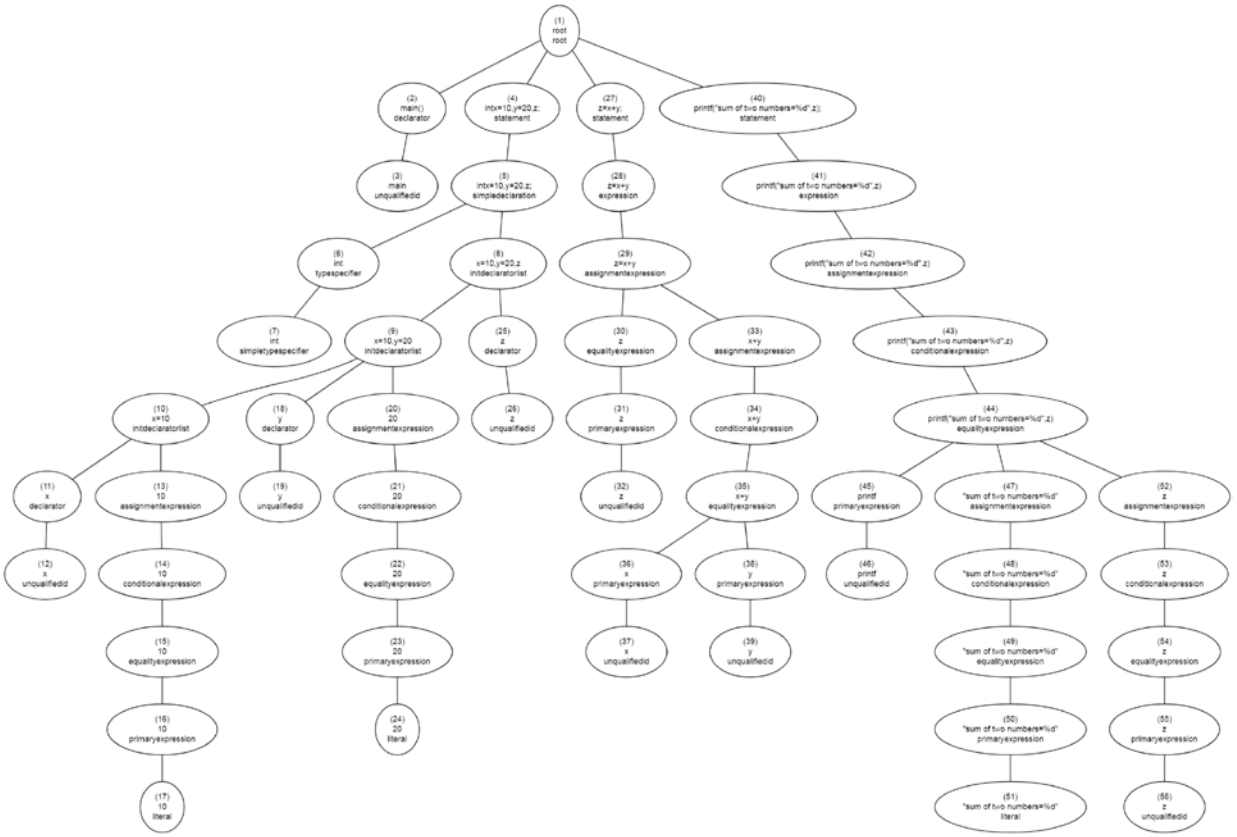

Fig.7. Filtered parse tree for C-code-1

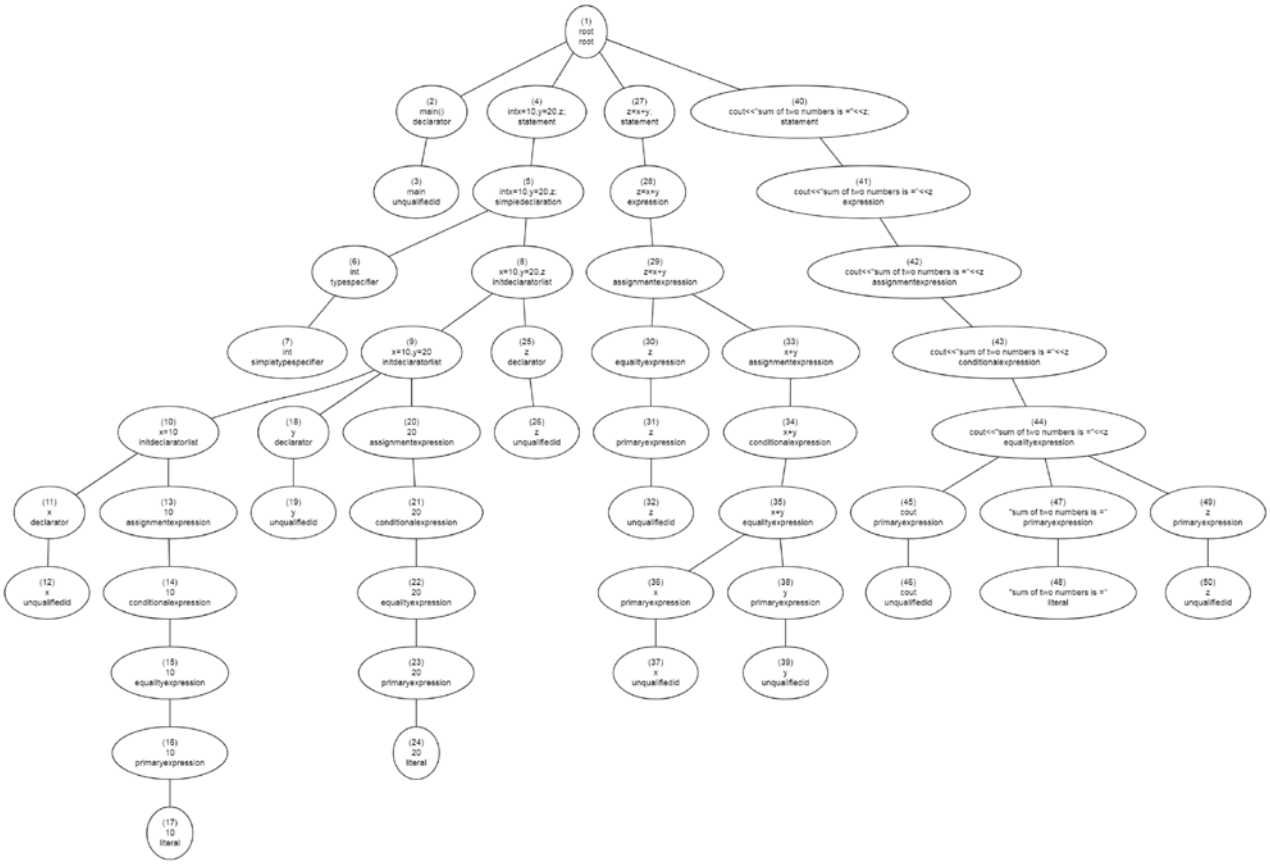

Fig.8. Filtered parse tree for CPP-code-1

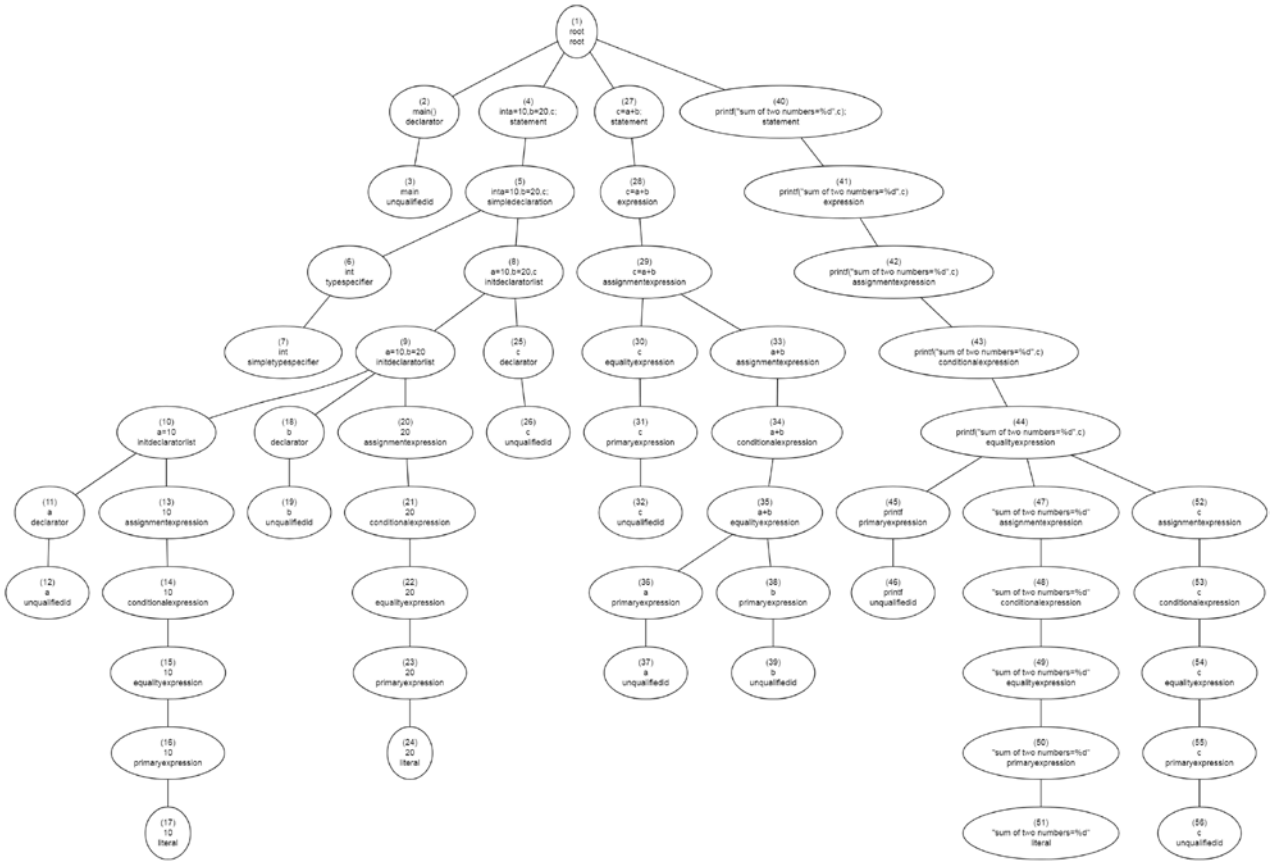

Fig.9. Filtered parse tree for C-code-2

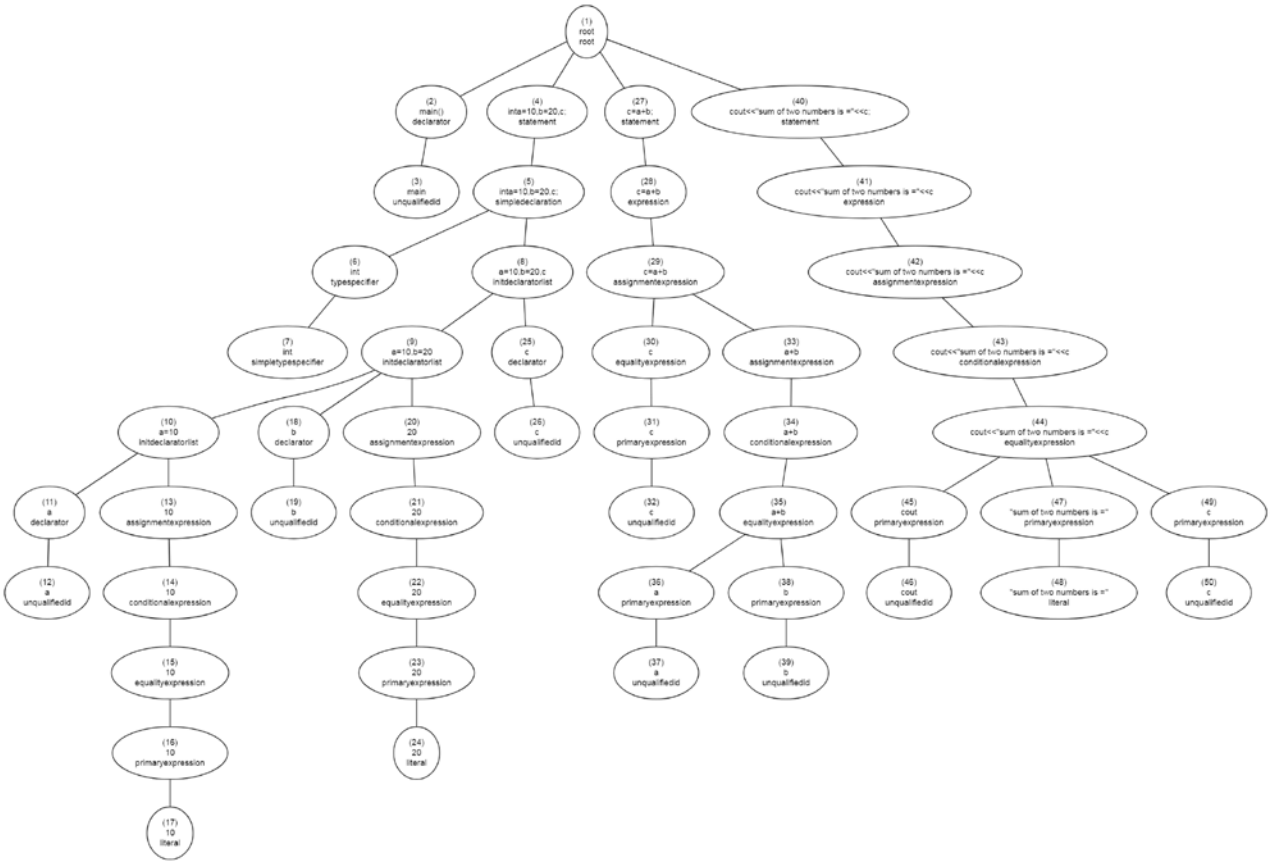

Fig.10. Filtered parse tree for CPP-code-2

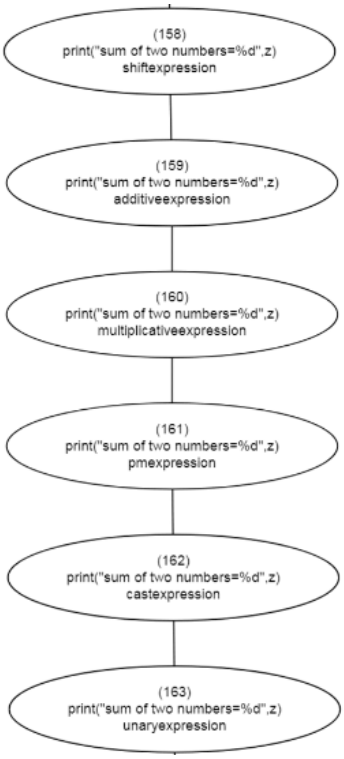

Fig.11. Childs of node shiftexpression for C code

Fig.12. Childs of node shiftexpression for CPP code

The following rules are included to make the serialization of C and CPP code. These node names are passed as an argument to DefaultTreeListener class to get filtered parse tree.

Declaration:
Simpledeclaration:typespecifier;

typespecifier:simpletypespecifier;

Simpledeclaration:initdeclaratorlist;
initdeclaratorlist:declaratory ;
declarator:unqualifiedid ;
initdeclaratorlist:primaryexpression ;
primaryexpression:literal ;

Reading/Writing statements
Statement:expression;
Expression:primaryexpression;
Primaryexpression:unqualifiedid|literal

Computation statements:
statement:statement|expression;
expression:primaryexpression;
primaryexpression:unqualifiedid|literal

C and CPP files are parsed using common grammar file CPP14.g4 to generate the parse tree. This process can be done by writing the java application to read the grammar files and input file and then calling the treeListner class generated in the previous stage of ANTLR processing. This application can be made to work on the generated parse tree to filter and extract all the nodes of our interest.

The advantage of using filtered parse tree is that, it reduces the size of the parse tree by omitting the headers and pre-processing lines. We have applied filtering on the code snippets, C-code-1, CPP-code-1, and C-code-2, and obtained filtered parse tree as shown in figure 7, figure 8, figure 9, and figure 10 respectively. Node information corresponding to line number 3 to 5, line number 4 to 7 for CPP-code-2, line number 3 to 8 for C-code-3, and line number 5 to 13 for CPP-code-3.

Note that there is a lot of difference between the actual parse tree generated by ANTLR, which is huge, even for small programs (the size of the parse tree generated in dot file for above C-code-1 code is 14KB). Figure 3 and Figure 4 show the parse tree for code C-code-1 and CPP-code-1, respectively. The corresponding filtered parse tree for code C-code-1 and CPP-code-1 have been presented in Figure 5 and Figure 6, respectively, which show the filtered parse tree for code C-code-1 and CPP-code-1 as compared to the original parse tree that has 210 nodes for C-code-1 and 188 nodes for CPP-code-1.We have filtered only the similar nodes that hold key information about the declaration, reading, computation, and writing statements of the program. We have presented all the images of original parse tree and filtered parse tree separately for evaluation.

3.3. Finding the parse tree similarity and displaying clone types

Filtered parse tree generated in the previous phase can be stored in dot file using the APIs of inmemantlr-tool-1.6 [35] that can be used for comparison using any of the natural language comparison technique such as Levenstein distance, Edit distance, Hamming distance, and Longest Common Subsequence. By looking at the fact that parse tree generates lot of information through repetitive rules, we have considered Term Frequency Inverse Document Frequency with cosine similarity to find the similarity between two different parse tree representations in dot file. The reason for using TF-IDF and cosine similarity is that this method will match all the tokens, including node names in the parse tree. Figure 4 shows the sequence for matching the filtered parse trees.

The Cosine Similarity function [36] is a widely used function to compute the similarity between two given term vectors, that is, the ratio of inner product (v1•v2) to the product of vector length. The similarity between the two vectors, v_1 and v_2 is given by

$$\cos(v_1, v_2) = (v_1 * v_2) \Big/ \|v_1\| \|v_2\| \tag{1}$$

where $\|v_1\|$ and $\|v_2\|$ *are Euclidean norms of vector V.*

From Table 1 we can easily find the cosine similarity between two documents by passing the vector

$$v_1 = TF * IDF - A \text{ and } v_2 = TF * IDF - B \tag{2}$$

Based on the cosine similarity scores of 135 C codes and 99 CPP codes, the threshold for clone type classification is set as follows:

```
if(val>=0.99 && val<=1.0)
print ("Type1");
```

```
if(val>=0.88 && val<0.99)
print (" Type2");
if(val>=0.78 && val<0.88)
print (" Type4");
if(val>=0.63 && val<0.78)
print (" Type3");
else
print ("Not a clone");
```

The following statements find cosine similarity between vectors v1 and v2 similar to [35].

```
for i= 1 to length (TF*IDF-A)
   {
   dotProduct+ = TF-IDF-A[i]* TF-IDF-B[i]
   magnitude1+=Math.pow(TF-IDF-A[i],2)
   magnitude2+=Math.pow(TF-IDF-B[i],2)
   }
   magnitude1=√magnitude1
   magnitude2=√magnitude2
```

Finally, we get a similarity between Doc A and Doc B as

$$\cos ine(TF-IDF-A, TF-IDF-B) = dot\,Product / (magnitude1 * magnitude2) \quad (3)$$

Frequency Inverse Document frequency (TF-IDF) [36] is a hash map-like data structure that finds frequencies of term occurrence in a document with the relative location. It is an SVM-based natural language processing technique that is used for document processing and comparison. Term frequency gives a count of each token in a document, and inverse document frequency gives the uniqueness of each token in a document. The advantage of using this approach is that it gives the relative position of each and every token in a document.

Equation (4) shows the use of TF-IDF in our work. TF-IDF is used to find how relevant a term is in a document [37]. TF (term frequency) measures how frequently a term occurs in a document, and IDF (inverse document frequency) gives log (number of documents/ number of documents with the term in it) [39].

$$tf - idf(t, D) = tf(t, D) * idf(t, D) \quad (4)$$

Where

$$tf(t, D) = freq\, t \in D \quad (5)$$

$$idf(t, D) = 1 + \log\left(N \Big/ \left| \{ d \in D : t \in d \} \right| \right) \quad (6)$$

N is a number of documents.

To illustrate the working of TF and IDF, consider two documents, A and B. Exclude the special characters and find seven readable characters in each document. Table 4 shows the results obtained for TF and IDF for Doc A and Doc B.

```
int a, b, c ;
c = a + b ;
```

```
int a, b, z ;
z = a + b ;
```

Table 4. TF-IDF values for code A and B

Token	Doc A	Doc B	TF-A	TF-B	IDF-A	IDF-B	TF.IDF-A	TF.IDF-B
Int	1	1	1/7=0.142	1/7=0.142	1+Log(2/2)=1	1+Log(2/2)=1	0.1428	0.1428
A	2	2	2/7=0.285	2/7=0.285	1+Log(2/2)=1	1+Log(2/2)=1	0.2857	0.2857
B	2	2	2/7=0.285	2/7=0.285	1+Log(2/2)=1	1+Log(2/2)=1	0.2857	0.2857
C	2	0	2/7=0.285	0/7=0	1+Log(2/1)=1.69	1+log(2/1)=1.69	0.4837	0
Z	0	2	2/7=0.285	0/7=0	1+Log(2/1)=1.69	1+log(2/1)=1.69	0	0.4837

Significance of using TF-IDF is that Term frequency identifies the words having a high occurrence in the documents, that is, we find words having more relevance in the document. On the other hand, Inverse Document Frequency finds the uniqueness of words by eliminating the words appearing many times in a document collection**.** Too general words should not be given high weightage (eg. "or," "not," "is," and "the"). Hence, words found rarely in a document collection but frequently in a particular document get high weightage. In other words, combining TF and IDF can assign high weightage to discriminative words in a document.

4. Experimental Setup and Dataset Creation

The experiment is conducted on the Windows 7 operating system with Intel core 2 duos CPU with 2 GHZ speed and 2GB RAM on three sets of data samples. Since there exist no standard data sets for cross language clone detection, we plan 3 set of case studies as shown below.

1. **Dataset-1:** To understand the parsing ability of ANTLR, and clone classification accuracy of cosine similarity, we initially validated our approach on 135 small programs of C with 99 C++ programs with average line count of 5. We recorded time taken to parse the various inputs files. The system took 128 minutes to parse and compare 135 C files with 99 CPP files, making a total of 13,365 exhaustive comparisons which is shown in figure 14 and 15. Table 6 and 7 presents sample case studies with node extraction and node filtering respectively.
2. **Dataset-2:** Encouraging results on our sample data set shown in the table 6 and 7 motivates us to evaluate our approach on the large scale data set. We have collected functionally similar C/C++ codes from sanfoundry.com which contains 1000 algorithm based semantically similar codes of C and C++. We have edited the codes according to clone type definitions to get total of 4234 true clone pairs.
3. **Dataset-3:** Next we perform systematic GitHub-web scrapping on 73,075 active repositories of C and 86,505 active repositories of C++ to collect 3246 semantically similar C/C++ codes based on simple problem statements with average line count of 21. Manual inspection of GitHub codes derives 12786 true clone pairs. Both the datasets together presents 16,032 true clone pairs. The group of 8 computer science students manually inspected all 16,032 clone pair to find number of different clone types as show in Table 5.

Table 5. Number of clone types for dataset 2 and 3.

Clone Type	T1	T2	T3	T4
No of clone pairs	5468	3742	2978	3844

5. Results and Analysis

In order to understand the effectiveness of our proposed work, in this section, we have first presented the similarity matching on a complete parse tree without any filtering. Then we have compared the similarity matching with that of the filtered parse tree and filtering with node renaming to prove the accuracy of clone detection.

5.1. Dataset-1

Table 6 shows the similarity matching on ANTLR-generated parse tree. The result shown in the table behaves excellently well for the same language code clone detection i.e similarity matching of addc with addc is 1.0 and add1cpp with add1cpp is 1.0 where both the examples are clone type 1. However, for Row 2, it gives a similarity matching of 0.61 for two functionally similar codes of C and CPP; the reason behind this is a long recursive rule for node name “shiftexpression ” for generating most descendent subtree for C code, as explained in Section 3.2.

We can significantly improve the matching of similarity between two codes of all the false-positive results shown in yellow color by applying the filtering to the nodes of ANTLR-generated parse tree, as explained in section 3.2. Table 7 shows the results of similarity matching by applying the filtering to the nodes of parse tree, where the similarity matching for the yellow-colored rows is increased significantly.

However, the clone classification precision did not improve. Since the functionality of both the codes of Row 2 is same, both the parse trees resemble the same. Except for the reading and writing functions, both the codes of yellow-colored rows are similar. Since they are functionally similar, we obtain similarity to be 1. This can be easily obtained by replacing the node names of “scanf” and “cin” to “read” and “printf” and “cout” to “write”. Table 9 provides accurate results by matching the similarity after node renaming.

Analysis: We have analyzed the results of the parse tree similarity using TFID-Cosine similarity in terms of precision and recall.

Precision tries to find out what proportion of positive instances were correct [36].

For instance, we know that C-code-1 and C-code-2 are Type 2. If the matching result is Type 2, then it indicates true positive results. But if we get the result of matching to be any type other than Type 2, then we call it false positive.

With this basic knowledge, we present precision as follows:

Precision=True Positive/ True Positive + False Positive

Recall tries to find out how many positives are identified correctly.

Recall= True Positive / True Positive + False Negative

Selected case studies before filtering, after filtering, and after node renaming have been presented in Tables 6, 7, and 8, respectively.

Table 6. Similarity matching and clone classification without filtering

Sl. No	Source file	Destination file	Similarity matching	Result obtained	Expected Result
1	add1c	add1c	1.0	Type1	Type1
2	add1c	add1cpp	0.61	Type3	Type1
3	add1c	addc	0.91	Type2	Type2
4	add1c	addcpp	0.55	Type3	Type2
5	add1c	factdowhilec	0.22	Not a clone	Not a clone
6	add1c	factdowhilecpp	0.20	Not a clone	Not a clone
7	add1c	factforc	0.21	Not a clone	Not a clone
8	add1c	factforcpp	0.14	Not a clone	Not a clone
9	add1c	factwhilec	0.22	Not a clone	Not a clone
10	add1c	factwhilecpp	0.18	Not a clone	Not a clone
11	add1c	helloc	0.09	Not a clone	Not a clone
12	add1cpp	add1cpp	1.0	Type1	Type1
13	add1cpp	addc	0.57	Not a clone	Type2
14	add1cpp	addcpp	0.90	Type2	Type2
15	factdowhilec	factdowhilec	1.0	Type 1	Type1
16	factdowhilec	factdowhilecpp	0.63	Type 3	Type1
17	factdowhilec	factforc	0.85	Type4	Type4
18	factdowhilec	factforcpp	0.66	Type3	Type4
19	factdowhilec	factwhilec	0.93	Type2	Type4
20	factdowhilecpp	factwhilec	0.63	Type3	Type4
21	factdowhilecpp	factforc	0.61	Type3	Type4
22	factdowhilecpp	factforcpp	0.76	Type3	Type4
23	factdowhilecpp	factwhilec	0.63	Type3	Type4
24	factdowhilecpp	factwhilecpp	0.81	Type4	Type4
25	factwhilec	factwhilecpp	0.61	Type 3	Type1
26	factforc	factforcpp	0.64	Type 3	Type1
27	factdowhilec	factwhilec	0.93	Type2	Type2
28	factwhilecpp	factforc	0.54	Not a clone	Type4
29	factwhilecpp	helloc	0.09	Not a clone	Not a clone
30	factdowhilecpp	hellocpp	0.06	Not a clone	Not a clone
31	helloc	hellocpp	0.37	Not a clone	Type1

The three tables present the results of similarity matching without node extraction, with node extraction, and node renaming. Few yellow-colored rows indicate false positive values recorded cosine similarity matching.

A. Table 6 shows similarity matching without node filtering. Precision and recall for the similarity matching can be calculated as follows.

True Positive=16, False Positive= 15, False Negative=3
Hence the Precision= 16/16+15=0.51 which 51%
Recall= 16/16+3=0.63 which 84%

The accuracy of clone detection and classification is exactly 53% which is very low, but the method can be used to match and classify type 1 and type 2 clones of same and for little instance cross-language too but do not produce Encouraging results for type 3 and type 4 of the same language and all the types of clones between C and CPP.

B. Table 7 presents similarity values after node filtering that significantly improves the percentage of similarity on the two functionally similar codes indicated in green color. From the table 6, we find precision =19/19+12 which is

61% and recall =19/19+1 which is 95% we can find that there is increase in accuracy compared to similarity measure without filtering. The good thing about node filtering is, it improves the similarity matching for two functionally equivalent codes.

Table 7. Similarity matching and clone classification with filtering.

Sl. No	Source file	Destination file	Percentage of matching	Result obtained	Expected Result
1	add1c	add1c	1.0	Type1	Type1
2	add1c	add1cpp	0.82	Type4	Type1
3	add1c	addc	0.94	Type2	Type2
4	add1c	addcpp	0.64	Type3	Type2
5	add1c	factdowhilec	0.39	Not a clone	Not a clone
6	add1c	factdowhilecpp	0.39	Not a clone	Not a clone
7	add1c	factforc	0.37	Not a clone	Not a clone
8	add1c	factforcpp	0.31	Not a clone	Not a clone
9	add1c	factwhilec	0.39	Not a clone	Not a clone
10	add1c	factwhilecpp	0.37	Not a clone	Not a clone
11	add1c	helloc	0.27	Not a clone	Not a clone
12	add1cpp	add1cpp	1.0	Type1	Type1
13	add1cpp	addc	0.64	Type3	Type1
14	add1cpp	addcpp	0.94	Type2	Type2
15	factdowhilec	factdowhilec	1.0	Type4	Type2
16	factdowhilec	factdowhilecpp	0.69	Type 3	Type1
17	factdowhilec	factforc	0.87	Type 4	Type4
18	factdowhilec	factforcpp	0.60	Type3	Type4
19	factdowhilec	factwhilec	0.86	Type4	Type4
20	factdowhilecpp	factwhilec	0.70	Type3	Type4
21	factdowhilecpp	factforc	0.64	Type3	Type4
22	factdowhilecpp	factforcpp	0.64	Type3	Type4
23	factdowhilecpp	factwhilec	0.89	Type4	Type4
24	factdowhilecpp	factwhilecpp	0.85	Type4	Type4
25	factwhilec	factwhilecpp	0.67	Type 3	Type1
26	factforc	factforcpp	0.86	Type 4	Type1
27	factdowhilec	factwhilec	0.86	Type4	Type4
28	factwhilecpp	factforc	0.82	Type 4	Type4
29	factwhilecpp	helloc	0.07	Not a clone	Not a clone
30	factdowhilecpp	hellocpp	0.11	Not a clone	Not a clone
31	helloc	hellocpp	0.45	Not a clone	Type1

C. Finally, we found the accuracy of the proposed cross-language clone detection in terms of precision and recall on the values generated in Table 9 after the node renaming; it contains True Positive=28, False Positive= 3, False Negative=0 Hence, we get precision= 28/27+4=90.0 and the recall value 28/28+0=1.0, which is 100%, except for the code of ‘for’ and ‘do’ while that are of type 3 we find that the proposed method works well with an accuracy of 100%. Accuracy of Type 4 clone detections can be increased to 100% by including the node name “iterativestatement,” which is common among C and CPP.

Table 8. similarity matching and clone classification with node renaming

Sl.No	Source file	Destination file	Percentage of matching	Result obtained	Expected Result
1	add1c	add1c	1.0	Type1	Type1
2	add1c	add1cpp	0.91	Type2	Type2
3	add1c	addc	0.95	Type2	Type2
4	add1c	addcpp	0.92	Type2	Type2
5	add1c	factdowhilec	0.32	Not a clone	Not a clone
6	add1c	factdowhilecpp	0.32	Not a clone	Not a clone
7	add1c	factforc	0.37	Not a clone	Not a clone
8	add1c	factforcpp	0.29	Not a clone	Not a clone
9	add1c	factwhilec	0.32	Not a clone	Not a clone
10	add1c	factwhilecpp	0.33	Not a clone	Not a clone
11	add1c	helloc	0.24	Not a clone	Not a clone
12	add1cpp	add1cpp	1.0	Type1	Type1
13	add1cpp	addc	0.90	Type2	Type2
14	add1cpp	addcpp	0.93	Type2	Type2
15	factdowhilec	factdowhilec	1.0	Type1	Type1
16	factdowhilec	factdowhilecpp	1.0	Type 3	Type1
17	factdowhilec	factforc	0.73	Type 3	Type4
18	factdowhilec	factforcpp	0.71	Type3	Type4
19	factdowhilec	factwhilec	0.86	Type4	Type4
20	factdowhilecpp	factwhilec	0.82	Type4	Type4
21	factdowhilecpp	factforc	0.78	Type3	Type4
22	factdowhilecpp	factforcpp	0.82	Type4	Type4
23	factdowhilecpp	factwhilec	0.89	Type4	Type4
24	factdowhilecpp	factwhilecpp	0.87	Type4	Type4
25	factwhilec	factwhilecpp	0.81	Type 4	Type1
26	factforc	factforcpp	1.0	Type 1	Type1
27	factdowhilec	factwhilec	0.87	Type4	Type4
28	factwhilecpp	factforc	0.72	Type 3	Type4
29	factwhilecpp	helloc	0.09	Not a clone	Not a clone
30	factdowhilecpp	hellocpp	0.13	Not a clone	Not a clone
31	helloc	hellocpp	1.0	Type1	Type1

We apply the same technique on 135 C codes and 99 CPP codes by generating the pruned parse tree. Precision and recall value for manual inspection is presented in the table 9.

Table 9. Precision and recall for cosine similarity.

Clone types	Precision	Recall
Type1	(77/77) 100	(77/77) 100
Type2	(63/63) 100	(63/63) 100
Type3	(46/49) 91.83	(46/49) 91.83
Type4	(52/58) 89.65	(52/58) 89.65

5.2. Results on dataset 2 and 3

We apply exhaustive comparison on the dataset 1 and 2 to detect the clone types shown in table 5. each of the clone types to find the ability of our method in detecting these 4 clone types. The precision and recall of the proposed method is presented in table 10

Table 10. Precision and recall on known clone pairs of datasets 2 and 3.

Clone Type	Detected clones	Precision and recall
1	5466	99.9
2	3704	98.98
3	2604	87.44
4	3202	83.29

Proposed method performs excellent on the known clone pairs of collected datasets and we are the first to provide cross language clone detection with type's classified. The table 12 presents the comparative study of the recently proposed tree based cross language clone detections based on ANTLR. Since the code for FETT is freely available, we run FETT on our dataset to get the average precision of 24%. The tools CroLSim prepared dataset containing to obtain average precision of 65%. Detection of cross language clones of C and C++ is out of the scope of CroLSim.

6. Comparative Study

We have made a direct comparison with Nichols'(2019) [31] work that is a recent contribution to the literature on cross-language clone detection. Table 11 shows the comparison between two approaches using general parameters like time, algorithm used, languages it works with and clone type classification. The major limitation of FETT is that the work do not consider parse tree node information for similarity comparison hence it cannot be applied to find clones between procedural language and object oriented languages and it cannot detect all 4 types of cross clones.

Table 11. comparison of proposed work with FETT

Sl.No	Parameter	Proposed Architecture	FETT
1	Clone detection and classification Time	1.47 sec	4.77 seconds
2	Algorithm used	Cosine similarity-TFIDF considers parse tree node information for clone detection	Tree edit distance does not consider parse tree node information for clone detection
3	Languages used	Procedure oriented C and object oriented CPP	Only object-oriented programming languages such as CPP, Java, and Java script
4	Clone classification	Classifies clones as Type 1, Type 2, Type 3 and Type4	Finds the similarity between two ASTs; does not classify clone types

Table 12. Similarity comparison for chosen inputs using FETT

Sl. No	Source	Destination	Similarity (0-1)
1	add1c	add1c	1.0
2	add1c	add1cpp	0.08
3	add1c	addc	0.30
4	add1c	addcpp	0.39
5	add1c	factdowhilec	0.55
6	add1c	factdowhilecpp	0.60
7	add1c	factforc	0.65
8	add1c	factforcpp	0.62
9	add1c	factwhilec	0.55
10	add1c	factwhilecpp	0.62
11	add1c	helloc	0.53
12	add1cpp	add1cpp	1.0
13	add1cpp	addc	0.08
14	add1cpp	addcpp	0.39
15	factdowhilec	factdowhilec	1.0
16	factdowhilec	factdowhilecpp	0.61
17	factdowhilec	factforc	0.81
18	factdowhilec	factforcpp	0.45
19	factdowhilec	factwhilec	0.89
20	factdowhilecpp	factwhilec	0.52
21	factdowhilecpp	factforc	0.46
22	factdowhilecpp	factforcpp	0.71
23	factdowhilecpp	factwhilec	0.71
24	factdowhilecpp	factwhilecpp	0.52
25	factwhilec	factwhilecpp	0.61
26	factforc	factforcpp	0.59
27	factdowhilec	factwhilec	0.89
28	factwhilecpp	factforc	0.47
29	factwhilecpp	helloc	No results
30	factdowhilecpp	hellocpp	No results
31	helloc	hellocpp	0.05

Since the code for FETT [31] is available Table 12 shows the similarity recorded for all the input sample codes shown in Table 7 using FETT that are useful in validating the functionally similar codes of C and CPP. The input code sample 'factdowhilec' and 'factdowhilecpp'are functionally the same; thus, we assumed the similarity matching for these samples to be 1. The same thing applies to two functionally similar codes of C and CPP, 'factforc' and 'factforcpp,' 'factwhilec and factdowhilec' respectively. The results obtained for the sample data inputs have been shown in the table 12.

Leaving the row 29 and 30 from the sample data shown in table 12 we get true positives=06, false positives=23, false negative=02 hence precision= 06/06+19= 2.4 % which proves that FETT is not suitable for the cross-language clones of C and CPP. Table 10 shows the cross-language accuracy for each of the clone types. These values were obtained from the manual validation of clone types. The same dataset was applied to the FETT to obtain the precision and recall values that prove that our system outperforms the FETT in finding the cross clones of C and CPP. Table 13 shows the accuracy of our work against that of FETT and CroLSim.

Table 13. Performance comparison of tree based techniques

Clone detection tools	Average Precision (%)	Dataset
CroLSim	65	Java, C# and Python
FETT	2.4	C++, Java and JavaScript
Our proposed method	91.97	C and C++

Accuracy results presented in table 13 indicate that the proposed method for cross-language clone detection presented in this paper is the first attempt made to match the similarity of procedural and object-oriented programs and significantly outperforms the recent cross language clone detection tools in finding functionally equivalent codes between C and C++ and vice versa.

7. Conclusion

This paper presents the methodology for detection and classification of cross-language code clones of C and C++ by generating the ANTLR parse tree for the input codes using CPP14.g4 grammar. Node filtering was applied on parse tree to eliminate the program header and unnecessary lengthy recursive rule of rightmost subtree node "shiftexpression" of C program, which is found to be the main difference between C and CPP code. TFIDF-Cosine similarity is applied to the filtered parse tree to get matching similarity and clone classification. For our case studies, the proposed technique has improved clone detection and classification precision from 51% to 61% with node filtering. Proposed method outperforms other tree based cross clone detection tools in terms of 95.37% and 91.97% precision for small scale and large scale datasets respectively. For the case studies containing six codes of each C and CPP, a total of 66 comparisons can be performed in less than 1 sec. The proposed cross-language clone detection out performs the existing techniques by exhibiting the highest precision of 95.37% in finding all types of clones (1, 2, 3 and 4) for 16,032 semantically similar clone pairs of C and CPP codes from sanfoundry.com. As ANTLR grammars are made freely available, the proposed methodology can be advanced to include any other programming language to build robust and more reliable method recommendation system to assist software forking process.

References

[1] Wang, W. L. (2020). Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. *27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 261-271). IEEE.

[2] Chanchal K. Roy, J. R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming, 470-495.

[3] Chanchal Kumar Roy, J. R. (2007). A survey on software clone detection research. *Queen's School of Computing TR* , 64-68.

[4] Dhavleesh Rattan, R. B. (2013). Software clone detection: A systematic review. *Information and Software Technology* , 1165-1199.

[5] Ain, Q. U. (2019). A systematic review on code clone detection. IEEE access, 86121-86144.

[6] Walker, A. a. (2020). Open-Source Tools and Benchmarks for Code-Clone Detection: Past, Present, and Future Trends. *SIGAPP Appl. Comput. Rev.*, 28–39.

[7] Kim, S. S. (2017). VUDDY: a scalable approach for vulnerable code clone discovery. *In Security and Privacy (SP), 2017 IEEE Symposium* (pp. 595-614). San Jose, CA, USA: IEEE.

[8] Sajnani, H. V. (2016). SourcererCC: scaling code clone detection to big-code. *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference* (pp. 1157-1168). Austin Texas: IEEE/ACM.

[9] Ragkhitwetsagul, C. K. (2019). Siamese: scalable and incremental code clone search via multiple code representations. *Empir Software Eng.* , 2236–2284.

[10] Philip Mayer, M. K. (2017). On multi-language soft-ware development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research andDevelopment.*

[11] Henryk Krawczyk, Dawid Zima, "Automation in Software Source Code Development", International Journal of Information

Technology and Computer Science, Vol.8, No.12, pp.1-9, 2016.
[12] Zeba Khanam,S.A.M Rizvi,"Aspectual Analysis of Legacy Systems: Code Smells and Transformations in C", IJMECS, vol.5, no.11, pp.57-63, 2013.
[13] Godfrey, C. K. (2006). clones considered harmful. *Reverse Engineering (WCRE'06)* (pp. 19-28). Benevento, Italy: IEEE.
[14] Sadowski C, S. K. (2015). How developers search for code: a case study. *ESEC/FSE* (pp. 191–201). New York, NY, USA: Association for Computing Machinery.
[15] Acar Y, B. M. (2016). You get where you're looking for: the impact of information sources on code security. *SP*, 289–305.
[16] Saini, V. a. (2018). Oreo: Detection of Clones in the Twilight Zone. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 354–365). New York, NY, USA: Association for Computing Machinery.
[17] Abdalkareem, R. a. (2017). On Code Reuse from StackOverflow. *Inf. Softw. Technol.*, 148–158.
[18] An L, M. O. (2017). Stack Overflow: a code laundering platform? *SANER*, 283–293.
[19] Baltes S, D. S. (2018). Usage and attribution of Stack Overflow code snippets in GitHub projects. *EmpirSoftw Eng*, 1–37.
[20] Mimoun, M. A. (2015). Clone detection using time series and dynamic time warping techniques. *Third World Conference on Complex Systems (WCCS)* (pp. 1-6). Marrakech: IEEE.
[21] Nafi, Kawser & Roy, Banani & Roy, Chanchal & Schneider, Kevin. (2019). A Universal Cross Language Software Similarity Detector for Open Source Software Categorization. Journal of Systems and Software. 162. 110491. 10.1016/j.jss.2019.110491.
[22] George Mathew, C. P. (2020). SLACC: Simion-based Language Agnostic Code Clones, arXiv:2002.03039 [cs.SE]. *Accepted at ICSE 2020 technical track*, (p. 11).
[23] Karnalim, O. (2020). TF-IDF Inspired Detection for Cross-Language Source Code Plagiarism and Collusion. *https://doi.org/10.7494/csci.2020.21.1.3389* .
[24] Perez, D. a. (2019). Cross-Language Clone Detection by Learning over Abstract Syntax Trees. *16th International Conference on Mining Software Repositories* (pp. 518–528). Montreal, Quebec, Canada: IEEE Press.
[25] K. W. Nafi, T. S. (2019). CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation. *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (pp. 1026-1037). San Diego, CA, USA: IEEE.
[26] Nghi D. Q. Bui, L. J. (2017). *Cross-Language Learning for Program Classification using Bilateral Tree-Based Convolutional Neural Networks*. arXiv:1710.06159.
[27] Hu, Y. a. (2017). Binary Code Clone Detection across Architectures and Compiling Configurations. *Proceedings of the 25th International Conference on Program Comprehension* (pp. 88-98). Buenos Aires, Argentina: IEEE.
[28] T. Vislavski, G. R. (2018). LICCA: A tool for cross-language clone detection. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 512-516). Campobasso: IEEE.
[29] Cheng, X. P. (2017). CLCMiner: Detecting Cross-Language Clones without Intermediates. *IEICE Transactions on Information and Systems*, 273-284.
[30] Al-Omari, F. K. (2012). Detecting Clones Across Microsoft .NET Programming Languages. *19th Working Conference on Reverse Engineering* (pp. 405-414). IEEE.
[31] Nichols, L. (2017). *https://sites.cs.ucsb.edu/~benh/research/downloads.html*. Retrieved April 10, 2020, from https://sites.cs.ucsb.edu: https://sites.cs.ucsb.edu/~benh/research/downloads.html
[32] Naumann, F. (2013). *SImilarity Measures.*
[33] Parr, T. (2014). *ANTLR.* Retrieved April 10, 2020, from https://www.antlr.org/: https://www.antlr.org/
[34] Parr, T. (2014). *https://github.com/antlr/grammars-v4*. Retrieved April 10, 2020, from https://github.com/: https://github.com/antlr/grammars-v4.
[35] Thome, J. (n.d.). *https://github.com/julianthome/inmemantlr*. Retrieved April 10, 2020, from https://github.com: https://github.com/julianthome/inmemantlr.
[36] Christopher D. Manning, P. R. (2008). *Introduction to Information Retrieval*. Cambridge: Cambridge University Press.
[37] Juan Zheng, W. X. (2019). Examining sequential patterns of self- and socially shared regulation of STEM learning in a CSCL environment. *Computers & Education*, 34-48.
[38] Goel, E. B. (2014). LectureKhoj: Automatic tagging and semantic segmentation of online lecture videos. *Seventh International Conference on Contemporary Computing (IC3)*. (pp. 7-43). Noida, 2014: IEEE.
[39] Elhadad, M. K. (2018). A Novel Approach for Ontology-Based Feature Vector Generation for Web Text Document Classification. *Int. J. Softw. Innov.* 1-10.

Authors' Profiles

Sanjay Ankali is a research scholar at VTU-RRC, Belagavi-590018 and working as Assistant Professor in the Department of CSE at KLECET, Chikodi, India-591201. His research interest is in the field of Software Engineering, Software clone detection and code plagiarism detection.

Dr. Latha Parthiban is working as Assistant Professor in department of Computer Science at, Pondicherry University- Community College, India-605008. She has received Bachelors of Engineering in Electronics from Madras University in the year 1994. M. E from Anna University in the year 2008 and Ph D from Pondicherry University in the year 2010. Her research interest includes Software Engineering, Big Data Analytics, and Computer Networking.

How to cite this paper: Sanjay B. Ankali, Latha Parthiban, "Detection and Classification of Cross-language Code Clone Types by Filtering the Nodes of ANTLR-generated Parse Tree", International Journal of Intelligent Systems and Applications(IJISA), Vol.13, No.3, pp.43-65, 2021. DOI: 10.5815/ijisa.2021.03.05