CHAPTER 4
ABSTRACT SYNTAX TREE MATCHING

4.1 INTRODUCTION

Structure-based plagiarism detection techniques for text (or source code) extract structure information from the files using different features of the text (or programming language) unlike LSI which considers any document as a bag of words irrespective of the type of document. A study by Verco and Wise (1996) shows that structure-metric-based methods tend to outperform attribute-counting-based information retrieval or similarity detection methods. One such method for code similarity detection is abstract syntax tree matching.

4.1.1 Abstract Syntax Trees

Abstract syntax tree is an intermediate tree representation of the source code. An AST is the output of the syntax analysis phase of a compiler. It represents the abstract syntactic structure of the program. Each node in the AST represents a construct in the source code. A terminal node in AST is either an identifier or a constant.

ASTs are often used to generate intermediate code by applying some transformations. The simplest way to generate intermediate code using ASTs is to traverse the AST and generate a node sequence which may be used as the intermediate code.

A parser generator is required to produce ASTs. The trees generated by parser generators are called parse trees and are usually huge in size. Parse trees are huge because most of the parsers create nodes for all the non-terminals in the grammar. A large number of nodes in them carry no structure information. A simple method to reduce the size of a parse tree and produce an AST is to retain all the terminal nodes and only those non-terminal nodes which have more than one child. These trees can also be reduced in size by making suitable modifications in the parser definition for a

specific language to remove redundant nodes which do not add any extra information to the program structure. The most common nodes which are eliminated include nodes that represent punctuation marks such as semi-colons and commas. The reduced tree will contain only those nodes which carry useful structural information and hence the name ***abstract syntax tree***.

Figure 4.1 shows the parse tree and its corresponding AST obtained without modifying the grammar for the C++ assignment statement a = a+2.

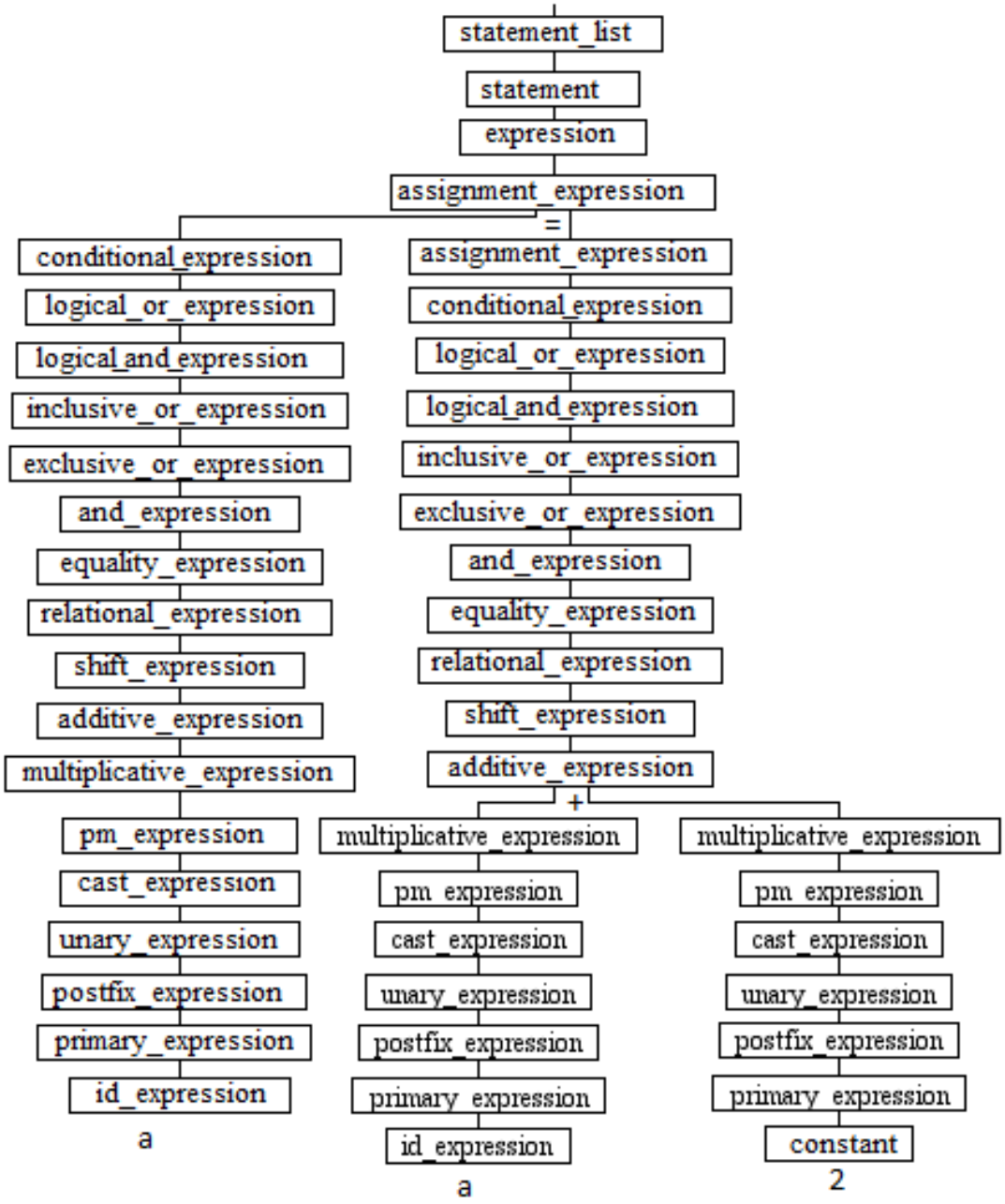
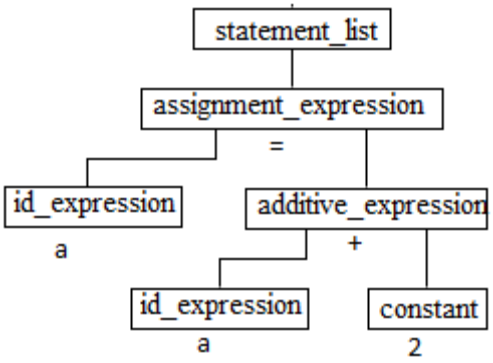


Figure 4.1. (a)



(b)

Figure 4.1. Parse Tree and AST for a = a+2 a) Parse Tree b) AST

AST contains much fewer nodes as compared to its parse tree. However, the information content in both the trees remains the same.

4.1.2 AST Matching

For AST matching, each source code file is parsed and its AST is generated. Once the ASTs are generated, comparison of ASTs can be done in different ways.

- One simple way is to compare the ASTs node by node. However, this method is not very efficient since such an algorithm halts whenever it encounters two nodes with different labels. There is hardly any meaning in this approach if root nodes of the ASTs have immediate children with different labels.

- Another method is to partition the ASTs into subtrees and compare each subtree in one AST with each subtree in another AST. However, this requires huge amount of time as well as space.

- A better approach employing hash functions, which is still computationally expensive, is used by the authors in (Baxter et al., 1998), as already discussed in section 2.1.

- Ligaarden (2007) proposes an AST based approach to detect plagiarism in Java source code. AST is generated for each of the source code files and a preorder traversal is done through the ASTs to be compared as done in (Koschke et al., 2006) to generate node sequences. Top Down Unordered Maximum Common Subtree Isomorphism (TDUMCSI) algorithm (Valiente,

2000; 2002) along with sequence matching algorithms - NW algorithm and LCS algorithm, are then used to compare the node sequences and find matches.

4.2 DIFFERENT TYPES OF PLAGIARISM

Plagiarism in source code can be done in many different ways. These include slight modifications in the code which do not affect the syntax or structure of the program as well as complex modifications which greatly affect the structure of the program. Based on the literature survey on source code plagiarism detection, the possible plagiarisms have been identified and classified into different types (eg; as in (Ligaarden, 2007)). These are briefly discussed in this section.

There are mainly two types of changes that can be made to a source code fragment.

- Lexical changes – These are changes which do not affect the syntax, semantics or structure of a program or code fragment. These changes do not modify the parse trees.

- Syntactic and structural changes – These are changes made to the syntax, and structure of the program. These changes greatly affect and modify the parse trees.

4.2.1 Classification of Common Plagiarism Strategies

Some of the common plagiarism strategies are changing identifier names, changing data types, changing the order of independent code, changing order of operands in expressions, replacing an expression with an equivalent expression, replacing one loop statement with another (eg; *for* with *while*, *while* with *for*), replacing on selection statement with another (eg; *if-else-if* with *switch*, *switch* with *if-else-if*), replacing function calls with function bodies, and group of statements with function calls.

The most common plagiarism strategies are classified into different types as:

Type 0

Type 0 plagiarism is directly copying the code from another source without any modifications.

–No changes in original and duplicated code.

Type 1

Type 1 plagiarism involves only lexical changes which do not affect the structure of the program.

–Adding or removing comments, blank spaces and blank lines

–Changing variable names and method names

–Changing constant values

–Changing data types

Type 2

Type 2 plagiarism involves syntactic and structural changes that slightly affect the syntax or structure of the program.

–Changing the order of operands in expressions

–Adding redundancy into the original code

–Adding extra display statements to change the overall appearance of the output.

Type 3

Type 3 plagiarism involves syntactic and structural changes that greatly affect the program structure.

–Replacing expressions with equivalent expressions

–Changing the order of independent code such as local variable declarations, class and method declarations and definitions

–Replacing *if-else* with *switch*, *switch* with *if-else*, *for* with *while* or *do-while*, *while* or *do-while* with *for*

–Replacing function call with function body or using a function for a set of statements.

4.2.2 ASTs for Most Common Plagiarism Strategies

In this research, Ligaarden's approach (Ligaarden, 2007) is modified and extended to detect plagiarisms in C, C++ and Java source code files. The ASTs obtained without making any modifications in C++ parser definition for original and plagiarized code fragments corresponding to some of the common plagiarism strategies are being discussed in this section. The need to modify the grammars has been identified from these ASTs.

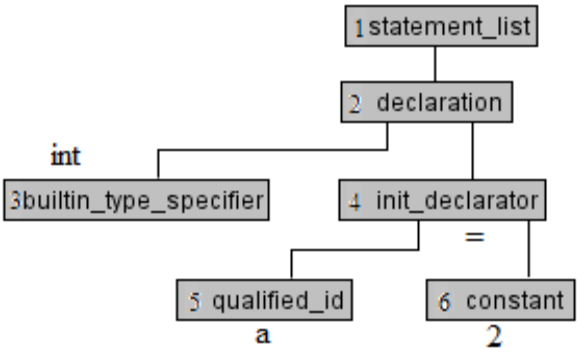
The figures show the ASTs and corresponding code. Separate nodes are not maintained for braces and semicolons. The shaded nodes in the ASTs are common to ASTs of both original and plagiarized code fragments. The numbers inside the shaded nodes in an AST indicate to which nodes in the other AST are they aligned to. Dark gray nodes are matched but with a shift in position.

Type 1 plagiarism does not modify the ASTs. Both the original and plagiarized code fragments will generate identical ASTs. Plagiarism strategies 1 and 2 are examples of Type 1 plagiarism.

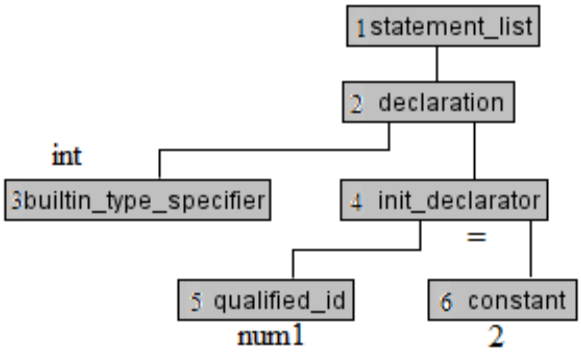
Plagiarism Strategy 1: Changing variable names

Fragment 1	**Fragment 2**
int a = 2;	int num1 = 2;

Figure 4.2 shows the ASTs for plagiarism strategy 1. It can be noted from the figure that both fragments 1 and 2 yield same AST.



(a)



(b)

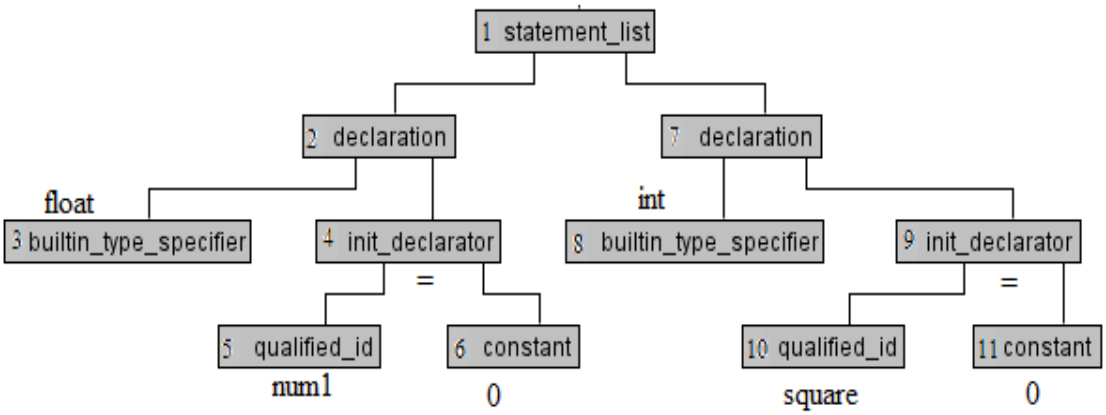
Figure 4.2. ASTs for Plagiarism Strategy 1 – Changing Variable Names a) AST for Fragment 1 b) AST for Fragment 2

Plagiarism Strategy 2: Changing data types

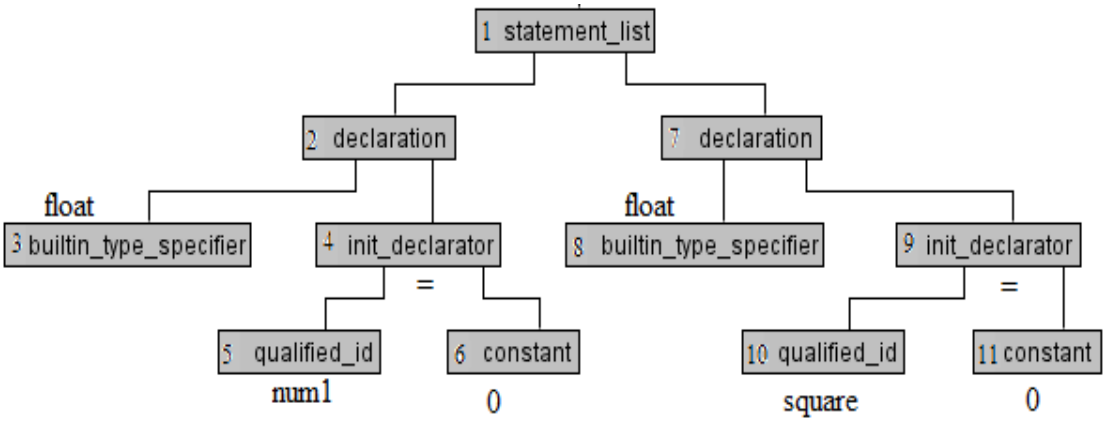
Fragment 1	**Fragment 2**

float num1 = 2;
 	
int square = 0;
	float num1 = 2;
	
	float square = 0;

Figure 4.3 shows the ASTs for plagiarism strategy 2. It can be noted from the figure that there is no type distinction in the original grammar. Therefore, both fragments 1 and 2 yield same AST. This helps to effectively identify type 1 plagiarism.



(a)



(b)

Figure 4.3. ASTs for Plagiarism Strategy 2 – Changing Data Types a) AST for Fragment 1 b) AST for Fragment 2

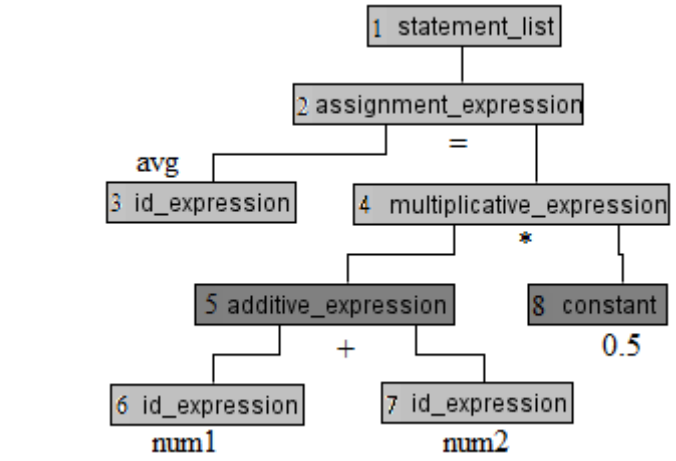
Type 2 plagiarism slightly modifies the ASTs. ASTs of original and plagiarized code fragments will differ by shift in positions of one or more subtrees. Plagiarism strategies 3 and 4 are examples of Type 2 plagiarism.

Plagiarism Strategy 3: Changing the order of operands in expressions

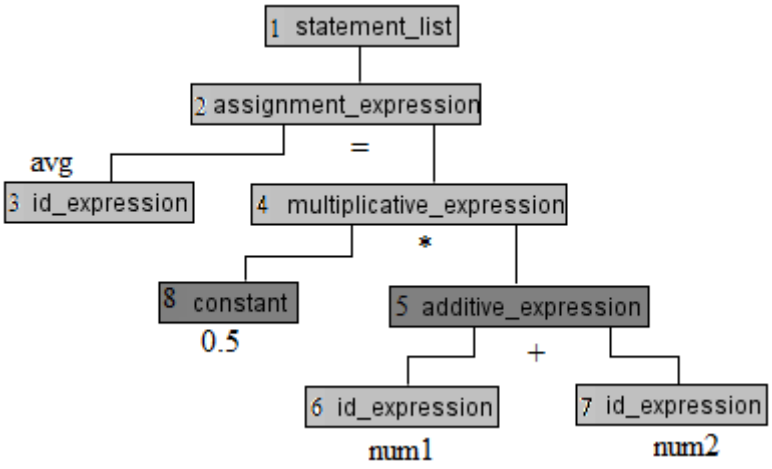
Fragment 1	**Fragment 2**
avg = (num1 + num2)*0.5;	avg = 0.5*(num1 + num2);

Both fragments 1 and 2 do the same operation and involve a multiplicative expression which adds *num1* and *num2* first and then multiplies it with 0.5. However, the order of operands in both the fragments is interchanged. In fragment 1, an additive expression is multiplied with a constant. In fragment 2, a constant is multiplied with an additive expression.

Figure 4.4 shows the ASTs for plagiarism strategy 3.



(a)



(b)

Figure 4.4. ASTs for Plagiarism Strategy 3 – Changing Order of Operands in Expressions a) AST for Fragment 1 b) AST for Fragment 2

It can be noted that the ASTs of the two fragments differ in positions of the subtrees of node 4 labeled multiplicative expression. Left subtree of node 4 in AST of

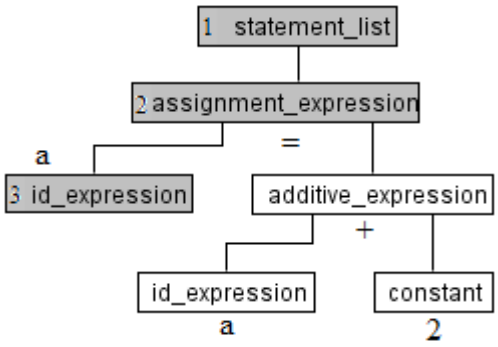
fragment 1 is aligned with right subtree of node 4 in AST of fragment 2. Right subtree of node 4 in AST of fragment 1 is aligned with left subtree of node 4 in AST of fragment 2. Nodes shaded in dark gray are matched but with a shift in position.

Type 3 plagiarism modifies the ASTs significantly except for some cases. This is clear from the ASTs obtained for type 3 plagiarisms discussed here. Plagiarism strategies 4, 5, 6, and 7 are examples of Type 3 plagiarism.

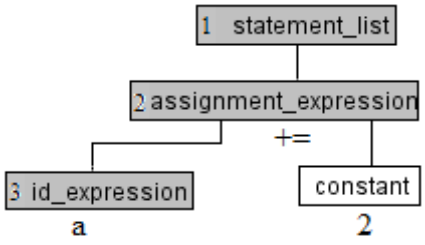
Plagiarism Strategy 4: Replacing expressions with equivalent expressions

Fragment 1	**Fragment 2**
a = a + 2;	a += 2;

Figure 4.5 shows the ASTs for plagiarism strategy 4.



(a)



(b)

Figure 4.5. ASTs for Plagiarism Strategy 4 – Replacing Expressions with Equivalent Expressions a) AST for Fragment 1 b) AST for Fragment 2

Both fragments 1 and 2 do the same operation and produce same result – increments the value of *a* by 2 and stores the result in *a*. Hence, they are equivalent. Fragment 1 uses normal assignment operator. Node 2 in AST of fragment 1 has therefore id_expression as its left subtree and an additive expression as its right subtree. Fragment 2 uses compound assignment operator to perform addition. Therefore, node 2 in AST of fragment 2 has id_expression as its left subtree but a constant as the right subtree.

Plagiarism Strategy 5: Changing the order of independent code (such as local variable declarations)

Fragment 1	**Fragment 2**
Statement 1: int num1=2, num2=3; *Statement 2:* int sum = 0; *Statement 3:* sum = num1 + num2;	*Statement 1:* int sum = 0; *Statement 2:* int num1=2, num2=3; *Statement 3:* sum = num1 + num2;

Fragment 1 has two variabe declaration statements. Fragment 2 has the same declaration statements with their order interchanged. Changing the order of these statements does not affect the subsequent code.

Figure 4.6 shows the ASTs for plagiarism strategy 5. The figure shows that the subtrees rooted at node 2 and node 11 in the ASTs of fragments 1 and 2 have swapped their positions corresponding to the change in position of the statements in the code fragments 1 and 2. The nodes 2 and 11 are shaded in dark gray in both the ASTs to indicate that they are matched by swapping positions.

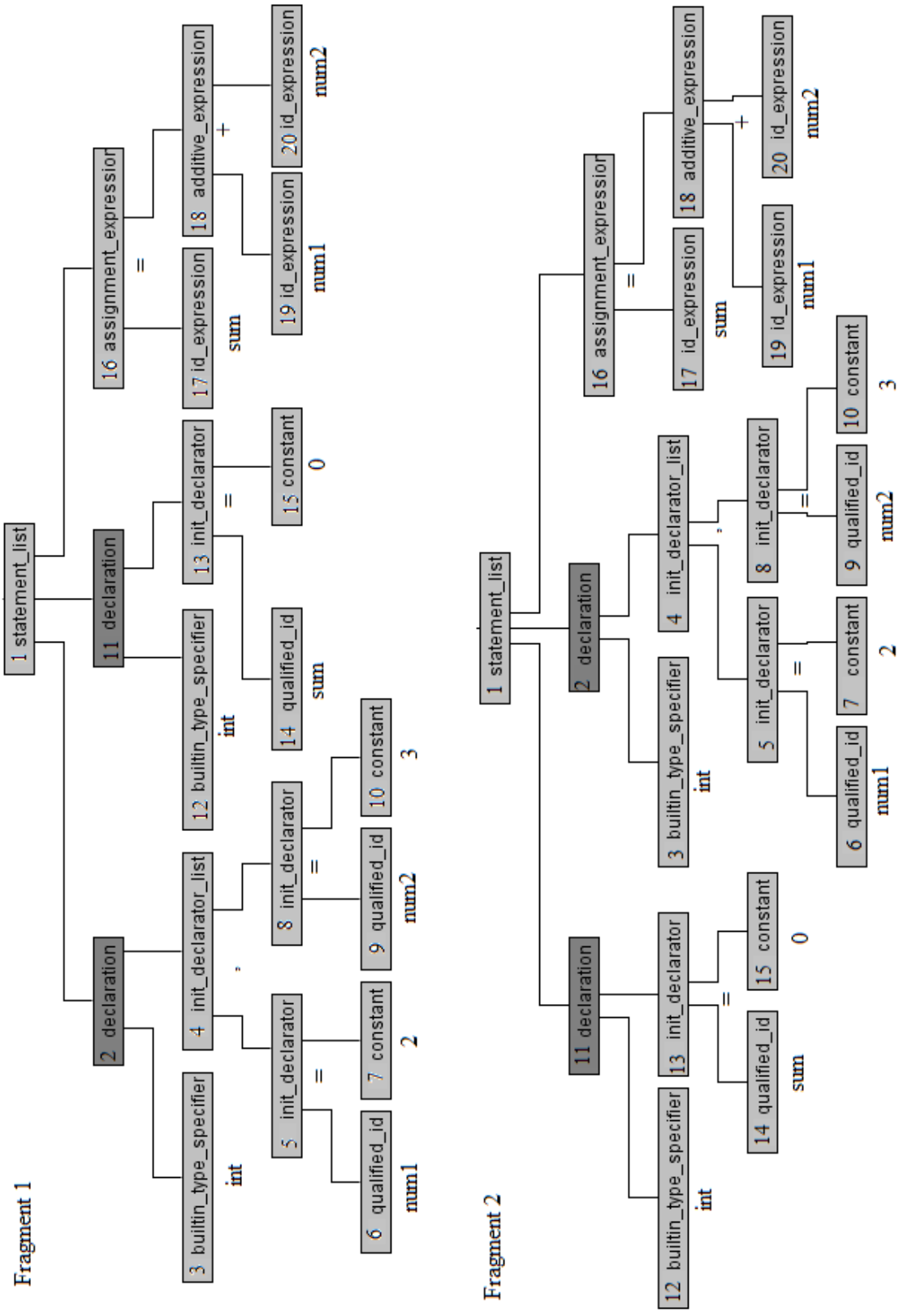


Figure 4.6. ASTs for Plagiarism Strategy 5 – Changing Order of Independent Code - ASTs for Fragment 1 and Fragment 2

Plagiarism Strategy 6: Replacing one selection statement with another: *if-else* with *switch* or *switch* with *if-else*

In C++, a compound statement (block) is a sequence of statements enclosed within braces. An *if*-statement either executes a single statement or a compound statement if the test condition is satisfied and it either executes a single statement or a compound statement if the test condition is not satisfied.

Consider three code fragments:

1. *if* without block
2. *if* with block
3. *if* is replaced with a switch statement

Fragment 1 (*if* without block)	**Fragment 2 (*if* with block)**	**Fragment 3 (*switch* statement)**

if (ch==1)
cout<<"one";
else if (ch==2)
cout<<"two";
else
cout<<"wrong choice!";
	if (ch==1)
	{ cout<<"one";}
	else if (ch==2)
	{ cout<<"two";}
	else
	{ cout<<"wrong choice!";}
		switch(ch)
		{
		case 1: cout<<"one";
		break;
		case 2: cout<<"two";
		break;
		default: cout<<"wrong choice!";
		}

The AST obtained for *if* with a single statement differs from that obtained for *if* with a compound statement. This difference can be noticed from the ASTs given in figure 4.7.a. Figure 4.7.b shows the ASTs (along with the corresponding code) for fragments 1 and 3. The subtrees corresponding to the body of *if* without compound statement and those corresponding to the case-blocks in *switch* statement do not show any match. Hence, the grammar has to be modified so as to allow a comparison between the corresponding subtrees.

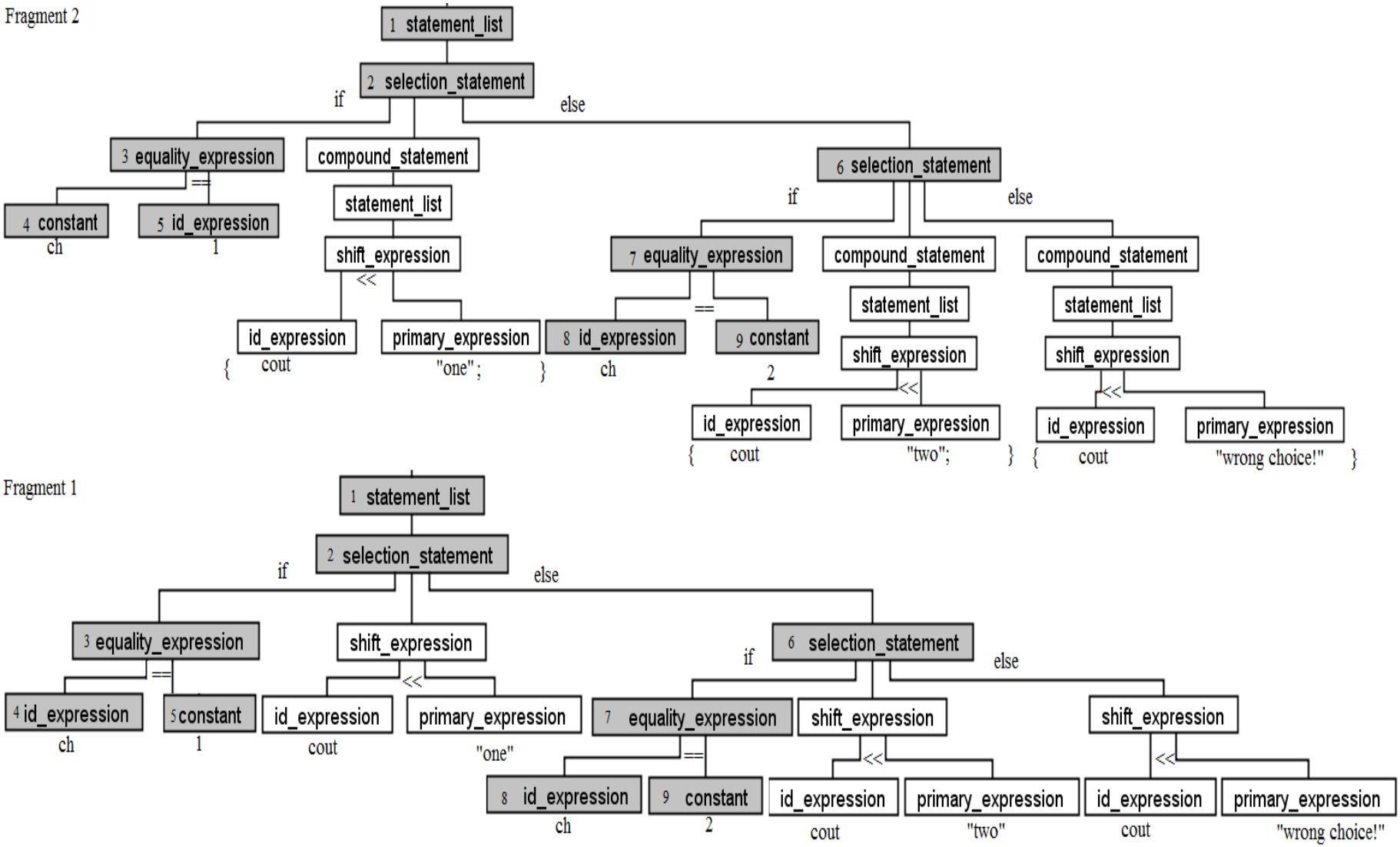

Fragment 2
1 statement_list
2 selection_statement
if
else
3 equality_expression
==
4 constant
ch
5 id_expression
1
compound_statement
statement_list
shift_expression
<<
id_expression
cout
primary_expression
"one";
6 selection_statement
if
else
7 equality_expression
==
8 id_expression
ch
9 constant
2
compound_statement
statement_list
shift_expression
<<
id_expression
cout
primary_expression
"two";
compound_statement
statement_list
shift_expression
<<
id_expression
cout
primary_expression
"wrong choice!"
Fragment 1
1 statement_list
2 selection_statement
if
else
3 equality_expression
==
4 id_expression
ch
5 constant
1
shift_expression
<<
id_expression
cout
primary_expression
"one"
6 selection_statement
if
else
7 equality_expression
==
8 id_expression
ch
9 constant
2
shift_expression
<<
id_expression
cout
primary_expression
"two"
shift_expression
<<
id_expression
cout
primary_expression
"wrong choice!"

Figure 4.7.(a)

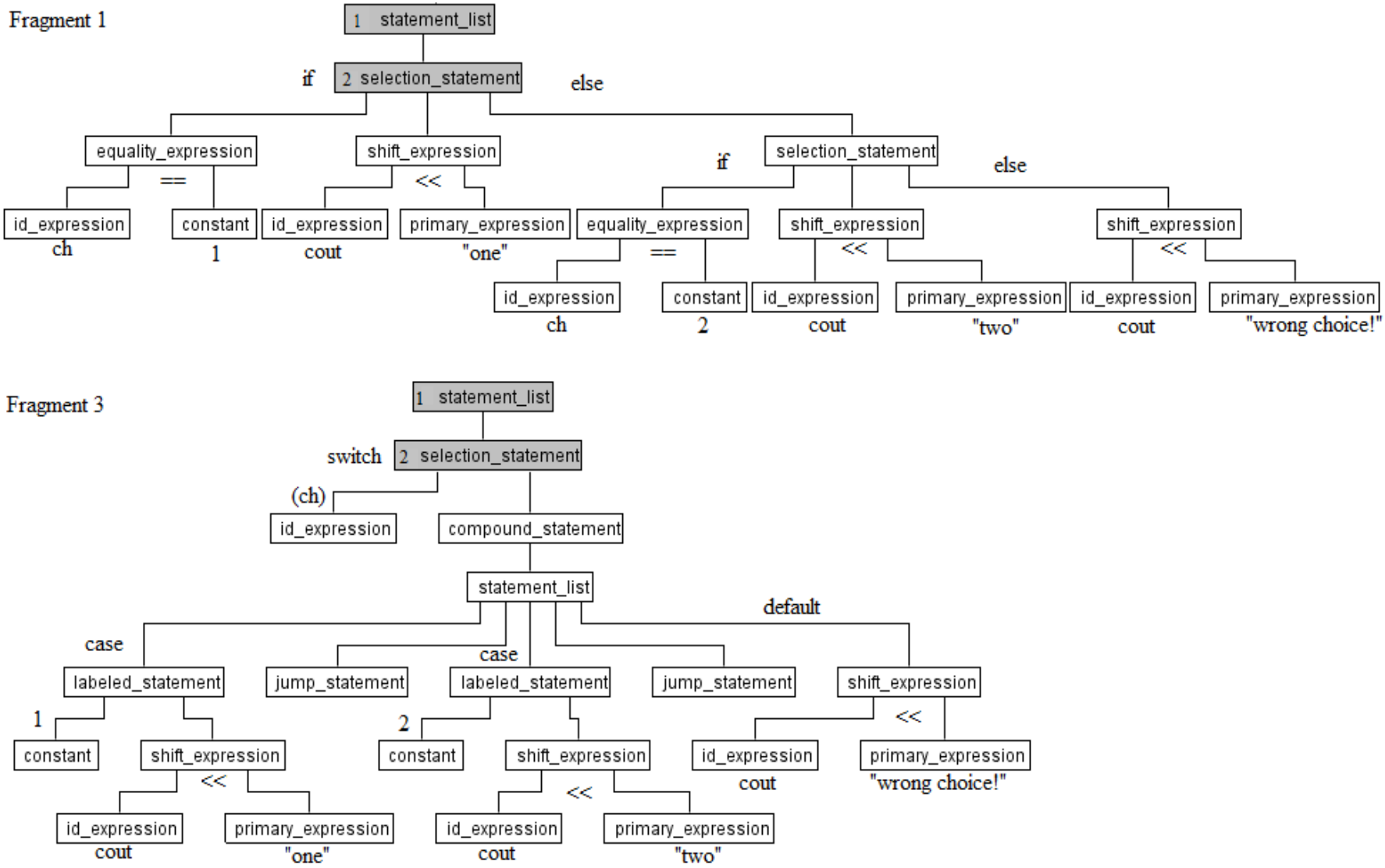

Fragment 1
1 statement_list
if
2 selection_statement
else
equality_expression
==
shift_expression
<<
if
selection_statement
else
id_expression
ch
constant
1
id_expression
cout
primary_expression
"one"
equality_expression
==
shift_expression
<<
shift_expression
<<
id_expression
ch
constant
2
id_expression
cout
primary_expression
"two"
id_expression
cout
primary_expression
"wrong choice!"
Fragment 3
1 statement_list
switch
2 selection_statement
(ch)
id_expression
compound_statement
statement_list
default
case
case
labeled_statement
jump_statement
labeled_statement
jump_statement
shift_expression
1
2
<<
constant
shift_expression
constant
shift_expression
id_expression
cout
primary_expression
"wrong choice!"
<<
<<
id_expression
cout
primary_expression
"one"
id_expression
cout
primary_expression
"two"

Figure 4.7. (b)

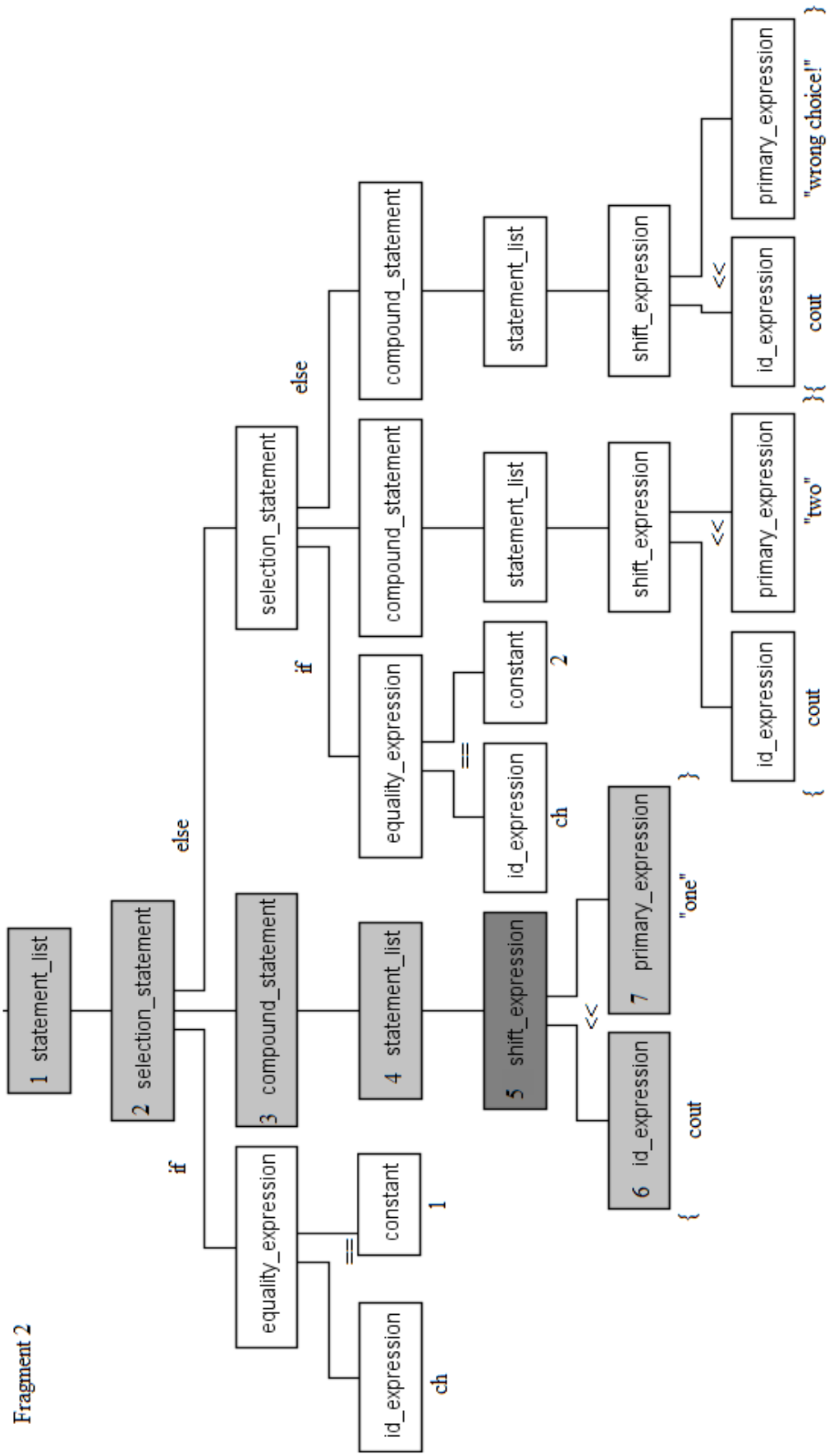
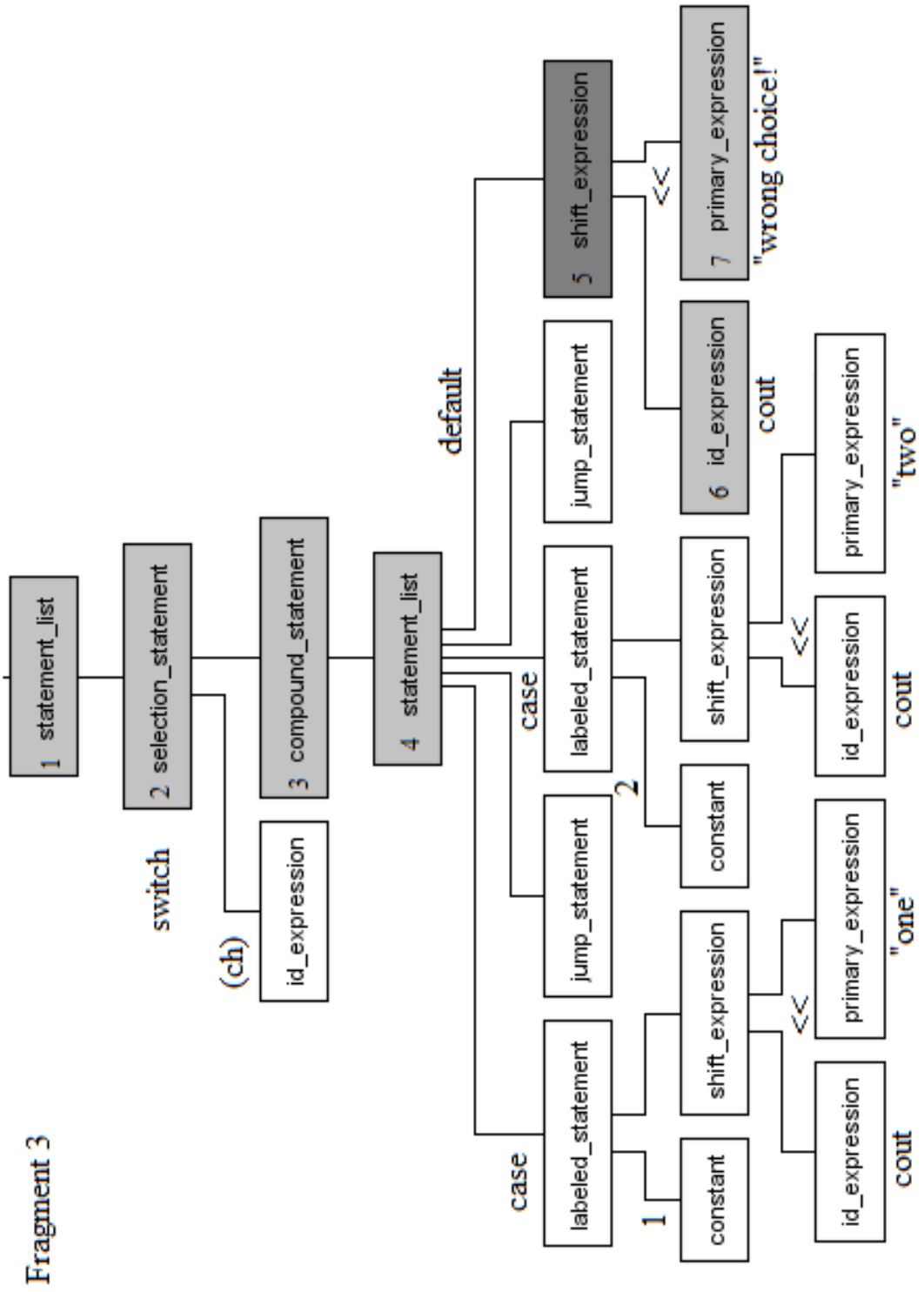

Fragment 2
1 statement_list
2 selection_statement
if
equality_expression
id_expression
ch
==
constant
1
else
3 compound_statement
4 statement_list
5 shift_expression
<<
6 id_expression
cout
7 primary_expression
"one"
selection_statement
if
equality_expression
id_expression
ch
==
constant
2
compound_statement
statement_list
shift_expression
id_expression
cout
<<
primary_expression
"two"
else
compound_statement
statement_list
shift_expression
id_expression
cout
<<
primary_expression
"wrong choice!"

Figure 4.7. (c)



(d)

Figure 4.7. ASTs for Plagiarism Strategy 6 – Replacing One Selection Statement with Another a) ASTs for Fragment 1 and Fragment 2 b) ASTs for Fragment 1 and Fragment 3 c) AST for Fragment 2 Aligned with AST for Fragment 3 d) AST for Fragment 3 Aligned with AST for Fragment 2

Figure 4.7.c shows the AST for *if* with compound statement aligned with the AST for *switch* statement shown in figure 4.7.d with the nodes common to both shaded in gray. A node shaded in dark gray indicates that the node is matched with a shift in position of the subtree.

In C++, the body of *switch* statement is always a compound statement. Moreover, the *case*-blocks in the *switch* statement are identified as labeled_statements unlike the *default*-block. Therefore, from figures 4.7.c and 4.7.d, it can be noted that the body of first *if*- which is a compound statement {cout<<"one"; } is matched with the *default*-block of *switch*.

If the *if* statement is modified to always use a compound statement, and if a compound statement is used for each *case*-block in the *switch* statement (which otherwise do not use a compound statement), this problem can be solved.

Plagiarism Strategy 7: Replacing one loop statement with another*: for* with *while* or *do-while*, *while* or *do-while* with *for*

Similar to the *if* statement, the loop statements *for*, *while*, or *do-while* statements may also either execute a single statement or a compound statement. Consider ASTs for the four code fragments:

1. *for* without block
2. *for* with block
3. *while* without block
4. *while* with block

Fragment 1 (*for* without block)	**Fragment 2 (*for* with block)**	**Fragment 3 (*while* without block)**	**Fragment 4 (*while* with block)**

for (i = 0; i<5; i++)
cout<<i;
	for (i = 0; i<5; i++)
	{
	cout<<i;
	}
		i = -1;
		while(++i<5)
		cout<<i;
			i = 0;
			while(i<5) {
			cout<<i;
			i++; }

Figure 4.8 shows the ASTs for plagiarism strategy 7.

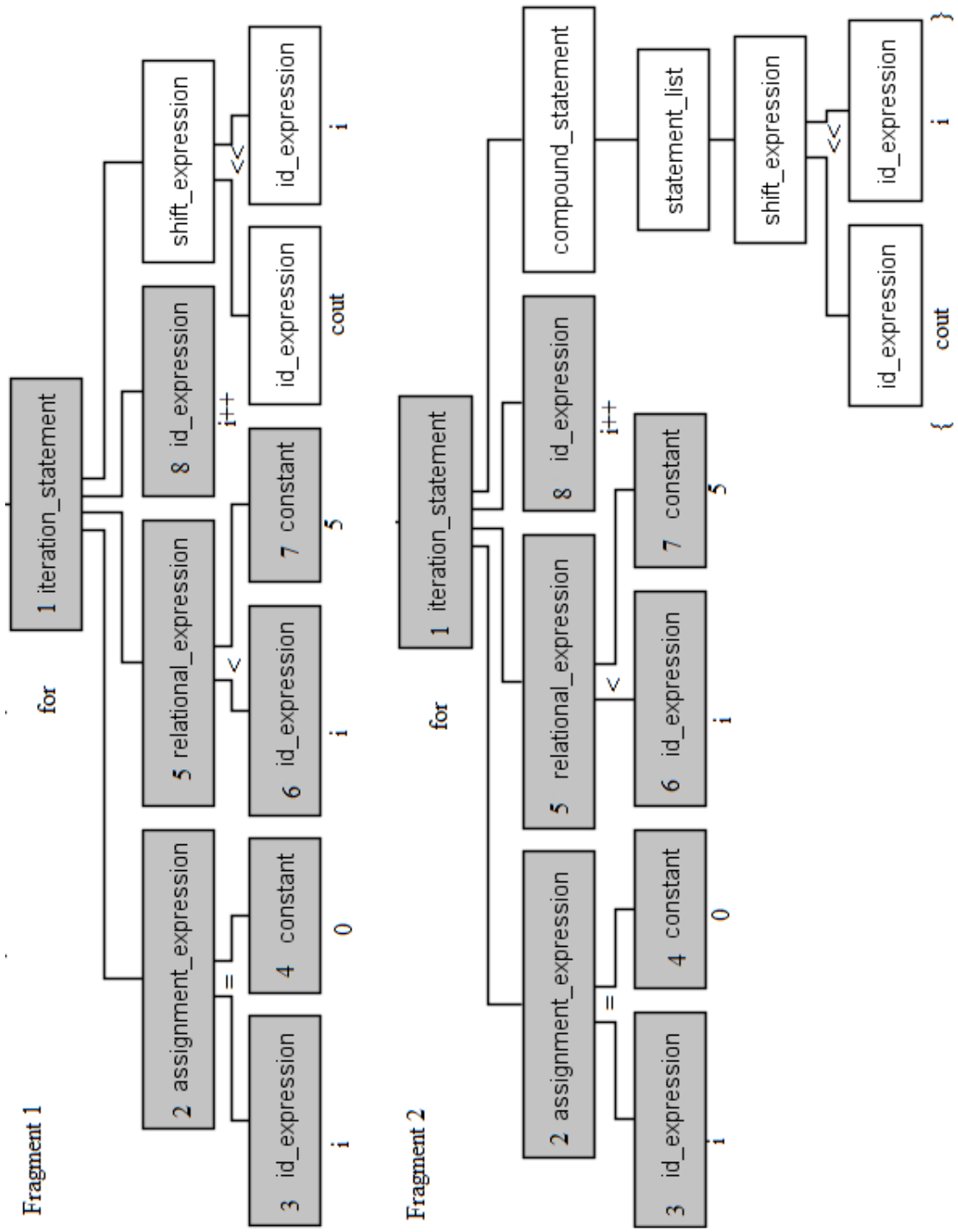


Figure 4.8. (a)

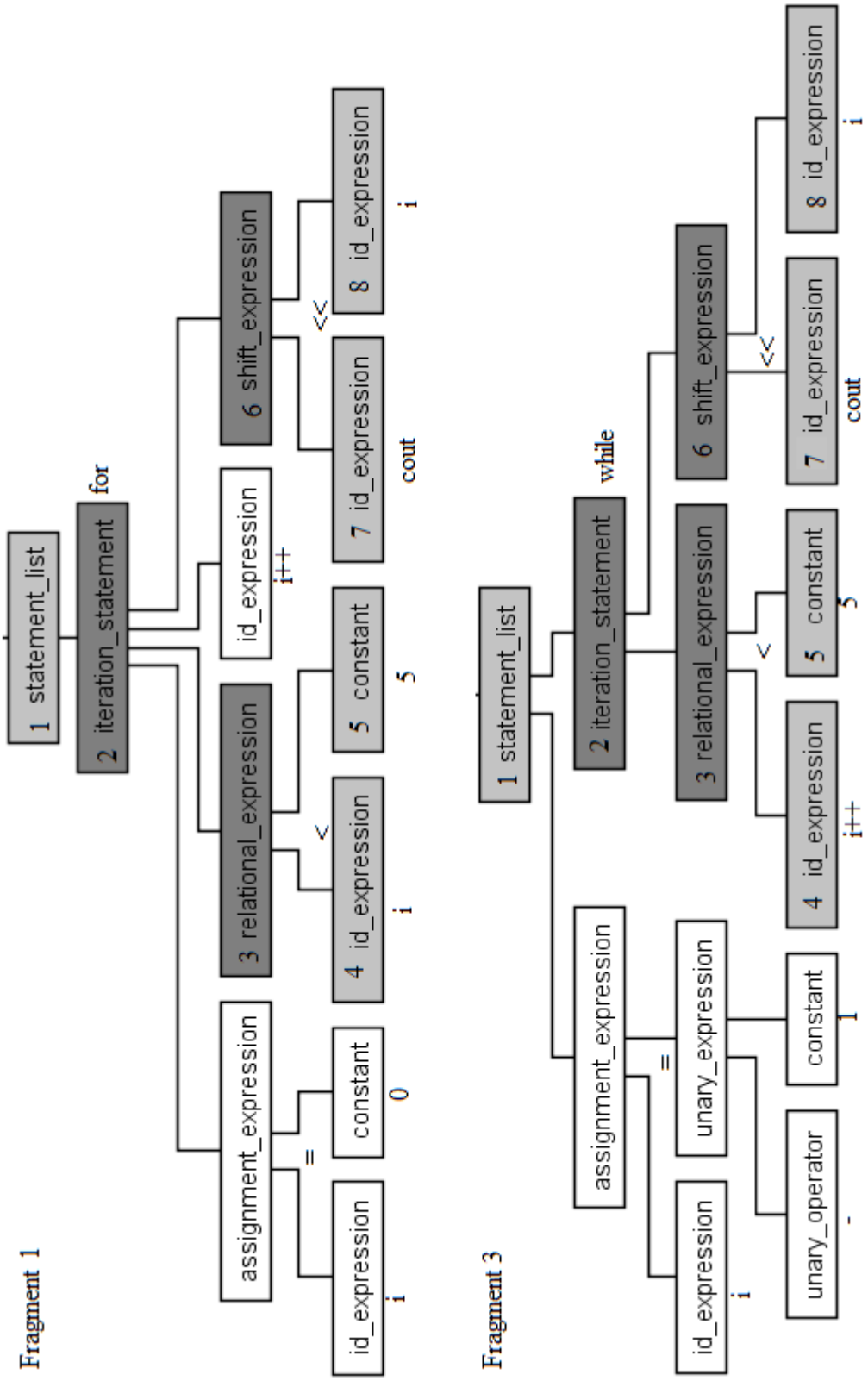

Fragment 1
1 statement_list
2 iteration_statement
for
assignment_expression
=
id_expression
i
constant
0
3 relational_expression
<
4 id_expression
i
5 constant
5
id_expression
i++
6 shift_expression
<<
7 id_expression
cout
8 id_expression
i
Fragment 3
1 statement_list
2 iteration_statement
while
assignment_expression
=
id_expression
i
unary_expression
unary_operator
-
constant
1
3 relational_expression
<
4 id_expression
i++
5 constant
5
6 shift_expression
<<
7 id_expression
cout
8 id_expression
i

Figure 4.8. (b)

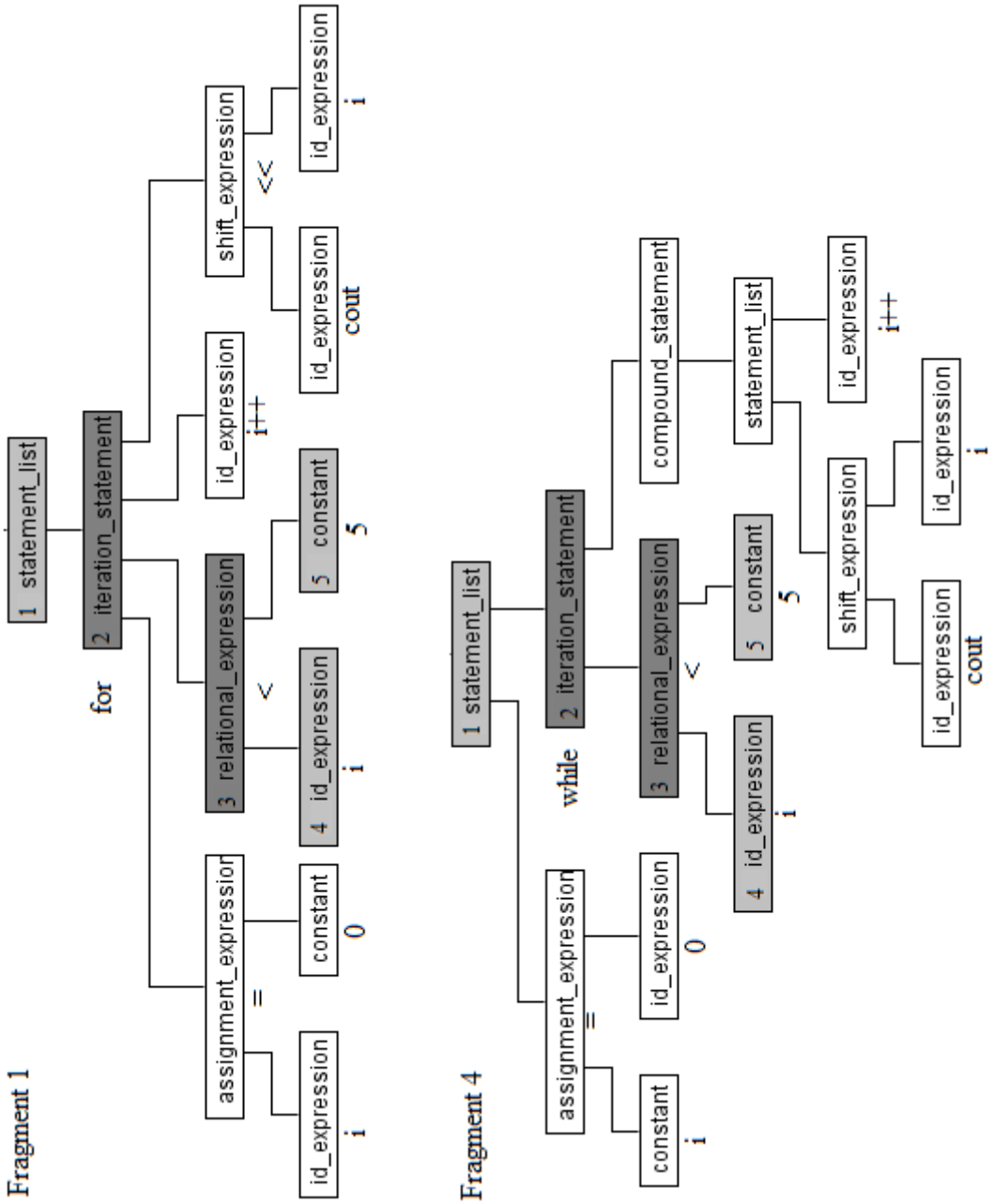

Fragment 1
1 statement_list
2 iteration_statement
for
assignment_expression
=
id_expression
i
constant
0
3 relational_expression
<
4 id_expression
i
5 constant
5
id_expression
i++
shift_expression
<<
id_expression
cout
id_expression
i
Fragment 4
1 statement_list
2 iteration_statement
while
assignment_expression
=
constant
i
id_expression
0
3 relational_expression
<
4 id_expression
i
5 constant
5
compound_statement
statement_list
shift_expression
id_expression
cout
id_expression
i
id_expression
i++

Figure 4.8. (c)

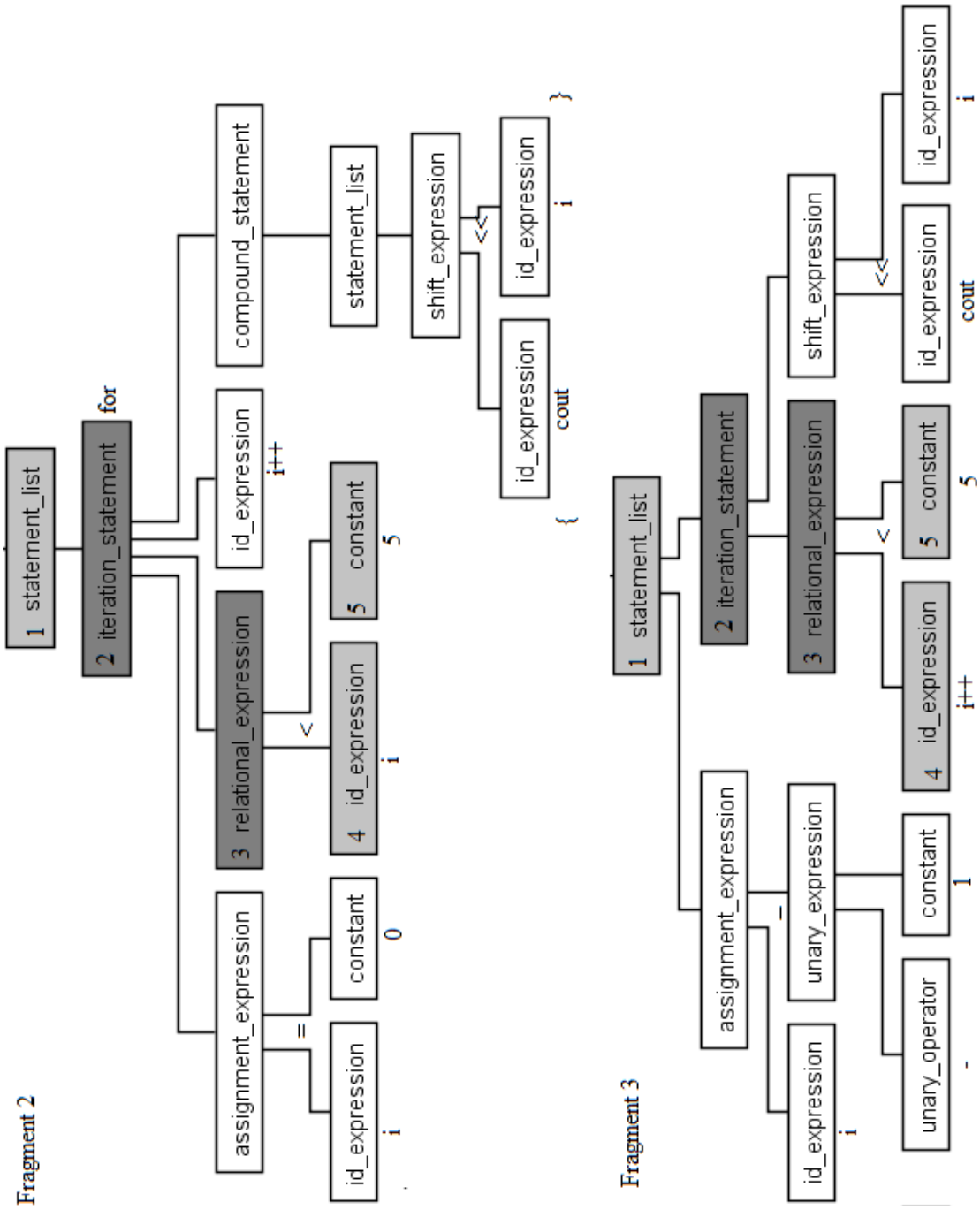
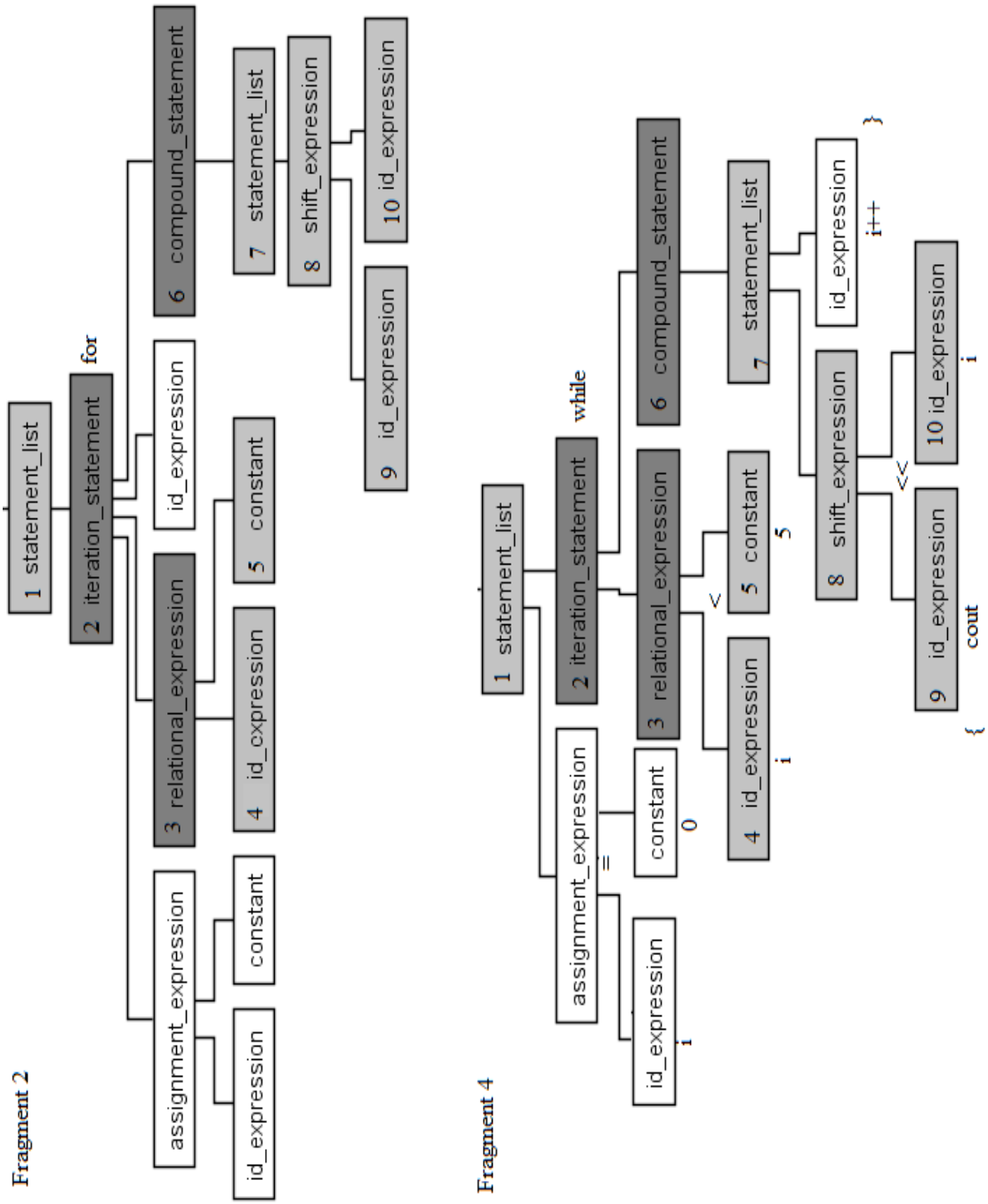

Fragment 2
1 statement_list
2 iteration_statement
for
3 relational_expression
assignment_expression
=
id_expression
i
constant
0
<
4 id_expression
i
5 constant
5
id_expression
i++
compound_statement
{
statement_list
shift_expression
id_expression
cout
<<
id_expression
i
}
Fragment 3
1 statement_list
2 iteration_statement
assignment_expression
id_expression
i
=
unary_expression
unary_operator
-
constant
1
3 relational_expression
4 id_expression
i++
<
5 constant
5
shift_expression
id_expression
cout
<<
id_expression
i

Figure 4.8. (d)



(e)

Figure 4.8. ASTs for Plagiarism Strategy 7 – Replacing One Loop Statement With Another a) ASTs for Fragment1 – *for* without block and Fragment2 – *for* with block b) ASTs for Fragment1 – *for* without block and Fragment 3 – *while* without block c) ASTs for Fragment1 – *for* without block and Fragment 4 - *while* with block d) ASTs for Fragment2 – *for* with block and Fragment 3 – *while* without block e) ASTs for Fragment2 – *for* with block and Fragment 4 – *while* with block

Figure 4.8.a shows the ASTs for fragments 1 and 2. It can be noted that the subtrees corresponding to the body of *for* without compound statement and that corresponding to *for* with compound statement do not show any match.

Figure 4.8.b shows the ASTs for fragments 1 and 3. It can be noted that the subtrees corresponding to the body of *for* without compound statement and that corresponding to *while* without compound statement are matched.

Figure 4.8.c shows the ASTs for fragments 1 and 4. The subtrees corresponding to the body of *for* without compound statement and that corresponding to the *while* with compound statement do not show any match.

Figure 4.8.d shows the ASTs for fragments 2 and 3. Here too, the subtrees corresponding to the body of *for* with compound statement and that corresponding to the *while* without compound statement do not show any match.

Figure 4.8.e shows the ASTs for fragments 2 and 4. It can be noted that the subtrees corresponding to the body of *for* with compound statement and that corresponding to *while* with compound statement are matched.

Hence, the grammar has to be modified so as to allow a comparison between the subtrees corresponding to the body of iteration statements with and without block. If the *for*, *while* and *do-while* statements are modified to always use a compound statement, the corresponding subtrees can be matched.

4.3 MODIFICATIONS TO GRAMMAR AND AST

4.3.1 Modifications to C Grammar and AST

4.3.1.1 C Grammar

```
Statement()      :    ( LOOKAHEAD(2) LabeledStatement()
                      | ExpressionStatement() | CompoundStatement()
                      | SelectionStatement() | IterationStatement()
                      | JumpStatement() )
```

```
LabeledStatement()    :    (<IDENTIFIER> ":" Statement()
                           |  <CASE> ConstantExpression() ":" Statement()
                           |  <DFLT> ":" Statement())
CompoundStatement() :      "{" [ LOOKAHEAD(DeclarationList())
                           DeclarationList() ] [ StatementList() ] "}"
StatementList()       :    (Statement())+
SelectionStatement()  :    <IF> "(" Expression() ")" Statement()
                           [ LOOKAHEAD(2) <ELSE> Statement() ]
                           | <SWITCH> "(" Expression() ")" Statement() )
IterationStatement()  :    <WHILE> "(" Expression() ")" Statement()
                           | <DO> Statement() <WHILE> "(" Expression() ")" ";"
                           | <FOR> "(" [ Expression() ] ";" [ Expression() ] ";"
                           [ Expression() ] ")" Statement() )
```

4.3.1.2 Modifying C Grammar and AST

The C grammar is modified as discussed in section 4.2.2 so as to incorporate the changes required in order to allow comparison between subtrees of different iteration statements or between subtrees of different selection statements.

A new rule StatementBlock is added and the rules for IterationStatement and SelectionStatement are redefined using StatementBlock. The AST generated with this modified grammar will always have StatementBlock as root of the subtree which corresponds to the body of iteration or selection statement. The rule for LabeledStatement is also redefined.

The subtree for a StatementBlock either corresponds to a single statement or a compound statement. If the iteration statement or selection statement has a single

statement as its body then the node label StatementBlock in the AST generated is changed to CompoundStatement. If the iteration statement or selection statement has a compound statement as its body then the node StatementBlock is removed.

The rule for *switch* statement is changed so that subtree rooted at CaseBlock corresponding to each *case* block with one or more statements is separated from the other. The root of each of these subtrees corresponding to each of the *case* blocks in *switch* statement is changed from CaseBlock to CompoundStatement to allow subtree matching. In the modified AST, the iteration statements *for*, *while* and *do-while* or the selection statement *if-else* will always have CompoundStatement as root of its subtree which corresponds to the body of iteration or *if-else* statement and the *case* blocks in *switch* will also be rooted at CompoundStatement. This modification allows the subtrees of iteration or selection statement with and without block to be matched.

4.3.1.3 Modified C Grammar

```
StatementBlock()      :     Statement()

Statement()           :     ( LOOKAHEAD(2) LabeledStatement()

                            | ExpressionStatement() | CompoundStatement()

                            | SelectionStatement() | IterationStatement()

                            | JumpStatement() )

LabeledStatement()    :     <IDENTIFIER> ":" Statement()

CompoundStatement() :       "{" [ LOOKAHEAD(DeclarationList())

                            DeclarationList() ] [ (Statement())+ ] "}"

SelectionStatement()  :     ( <IF> "(" Expression() ")" StatementBlock()

                            [ LOOKAHEAD(2) <ELSE> StatementBlock() ]

                            | <SWITCH> "(" Expression() ")" "{" ( CaseLabel()

                            CaseBlock() )* "}")
```

```
CaseLabel()          :    (<CASE> ConstantExpression() ":" | <DFLT> ":" )

CaseBlock()          :    (Statement())*

IterationStatement() :    ( <WHILE> "(" Expression() ")" StatementBlock()

                          | <DO> StatementBlock() <WHILE> "(" Expression()

                          ")" ";" | <FOR> "(" [ Expression() ] ";" [ Expression() ]

                          ";" [ Expression() ] ")" StatementBlock() )
```

4.3.1.4 Example of ASTs Generated using Modified C Grammar

To understand the effect of modifying the grammar rules and modifying the AST, consider, for example, ASTs of the two code fragments:

Fragment 1 (***for* without block**)	**Fragment 2** (***for* with block**)

for (i = 0; i<5; i++)
printf("%d", i);
	for (i = 0; i<5; i++)
	{
	printf("%d", i);
	}

Figure 4.9.a shows the ASTs obtained using original grammar for the two code fragments where the ASTs do not match due to the difference in labels of the root of the subtrees corresponding to loop bodies with and without block.

Figure 4.9.b shows the ASTs obtained using modified C grammar for the two code fragments. It also shows the ASTs obtained after modifying them to allow subtree matching. It can be noted from figure 4.9.b that changing the label StatementBlock to CompoundStatement in the AST for *for* without block and removing the node labeled StatementBlock from the AST for *for* with block, both obtained using modified C grammar, yields the same AST as given in figure 4.9.c.

for without block
IterationStatement
AssignmentExpression
PrimaryExpression
AssignmentOperator
Constant
RelationalExpression
PrimaryExpression
Constant
PostfixExpression
PrimaryExpression
ArgumentExpressionList
Constant
PrimaryExpression
for with block
IterationStatement
AssignmentExpression
PrimaryExpression
AssignmentOperator
Constant
RelationalExpression
PrimaryExpression
Constant
CompoundStatement
PostfixExpression
PrimaryExpression
ArgumentExpressionList
Constant
PrimaryExpression

Figure 4.9. (a)

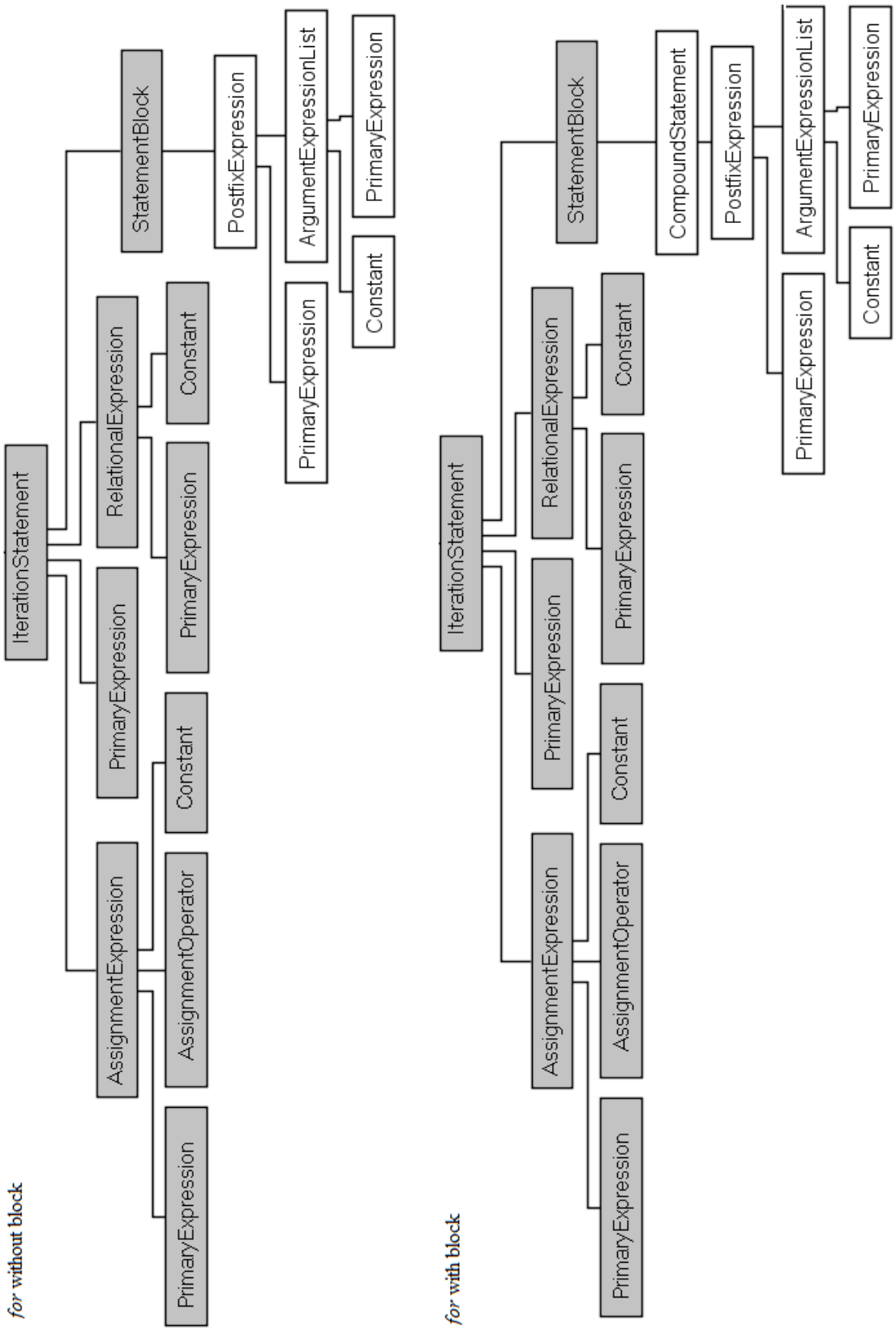
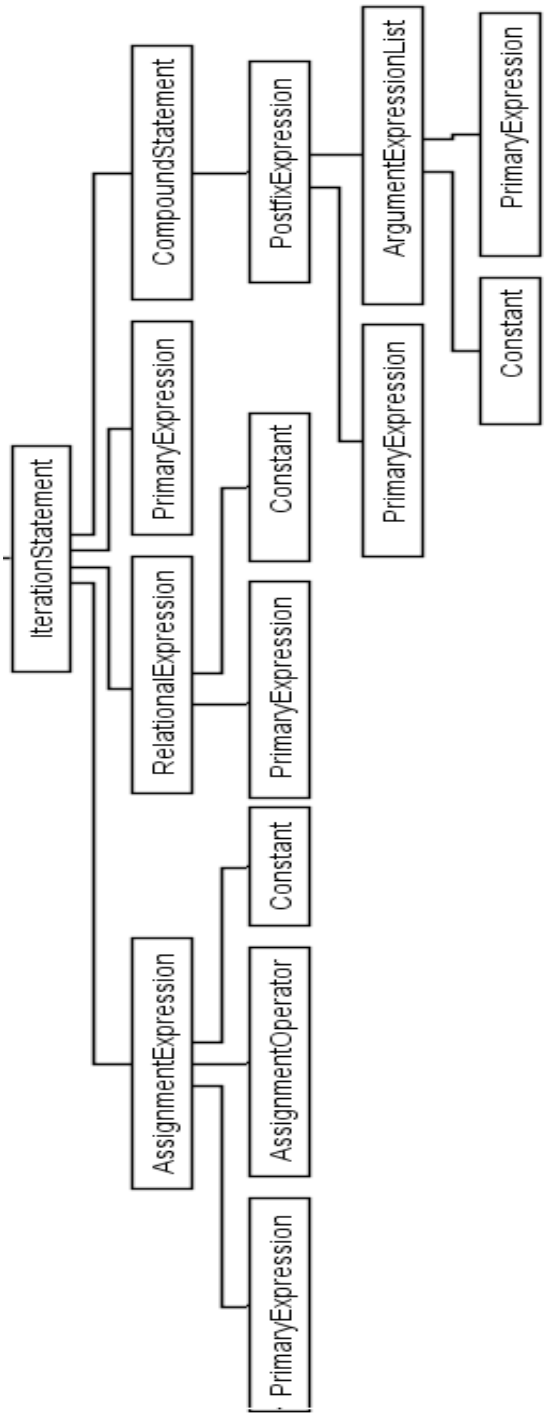

for without block
IterationStatement
AssignmentExpression
PrimaryExpression
AssignmentOperator
PrimaryExpression
RelationalExpression
PrimaryExpression
Constant
Constant
StatementBlock
PostfixExpression
PrimaryExpression
ArgumentExpressionList
Constant
PrimaryExpression
for with block
IterationStatement
AssignmentExpression
PrimaryExpression
AssignmentOperator
PrimaryExpression
RelationalExpression
PrimaryExpression
Constant
Constant
StatementBlock
CompoundStatement
PostfixExpression
PrimaryExpression
ArgumentExpressionList
Constant
PrimaryExpression

Figure 4.9. (b)



(c)

Figure 4.9. ASTs Generated using Original and Modified C Grammar a) ASTs Generated using Original Grammar for Fragments 1 and 2 b) ASTs Generated using Modified Grammar for Fragments 1 and 2 b) AST for *for* without Block with Label StatementBlock Changed to CompoundStatement or for *for* with Block with Node Labeled StatementBlock Removed

4.3.2 Modifications to C++ Grammar and AST

4.3.2.1 C++ Grammar

```
statement_list()        :   (LOOKAHEAD(statement())  statement())+

statement()             :   LOOKAHEAD(declaration())  declaration()
                            | LOOKAHEAD(expression()";") expression() ;"
                            | compound_statement() | selection_statement()
                            | jump_statement() | ";" | try_block() | throw_statement()
                            | LOOKAHEAD(2)  labeled_statement()
                            | iteration_statement()

labeled_statement()     :   <ID> ":" statement()
                            | "case" constant_expression() ":" statement()
                            | "default"  ":" statement()

compound_statement() :      "{" (statement_list())?  "}"

selection_statement()   :    "if" "(" expression() ")" statement()
                            (LOOKAHEAD(2) "else" statement())?
                            | "switch" "(" expression() ")" statement()

iteration_statement()   :    "while" "(" expression() ")" statement()
                            | "do" statement() "while" "(" expression() ")" ";"
                            | "for" "("(LOOKAHEAD(3) declaration() |
                            expression() ";" | ";") ( expression() )? ";"
                            ( expression() )? ")" statement()
```

4.3.2.2 Modifying C++ Grammar and AST

The modification done to C++ grammar is similar to that done to C grammar. A new rule statement_block is added and the rules for iteration_statement and selection_statement are redefined using statement_block. The AST generated with this modified grammar will always have statement_block as root of the subtree which corresponds to the body of iteration or selection statement. The rule for labeled_statement is also redefined.

The subtree for a statement_block either corresponds to a single statement or a compound statement. If the iteration statement or selection statement has a single statement as its body then the node label statement_block in the AST generated is changed to compound_statement. If the iteration statement or selection statement has a compound statement as its body then the node statement_block is removed.

The rule for *switch* statement is changed so that subtree rooted at case_block corresponding to each *case* block with one or more statements is separated from the other. The root of each of these subtrees corresponding to each of the *case* blocks in *switch* statement is changed from case_block to compound_statement to allow subtree matching. In the modified AST, the iteration statements *for*, *while* and *do-while* or the selection statement *if-else* will always have compound_statement as root of its subtree which corresponds to the body of iteration or *if-else* statement and the *case* blocks in *switch* will also be rooted at compound_statement. This modification allows the subtrees of iteration or selection statement with and without block to be matched.

4.3.2.3 Modified C++ Grammar

```
statement_block()   :   statement()

statement()         :   LOOKAHEAD( declaration()) declaration()

                        | LOOKAHEAD( expression() ";" ) expression() ";"

                        | compound_statement() | iteration_statement()

                        | LOOKAHEAD(2) labeled_statement()

                        | selection_statement() | jump_statement() | ";"
```

```
                                | try_block() | throw_statement()

labeled_statement()     :       <ID> ":" statement()

compound_statement() :          "{"(statement())* "}"

iteration_statement()   :       "while" "(" expression() ")" statement_block()

                                | "do" statement_block() "while" "(" expression() ")" ";"

                                | "for" "("(LOOKAHEAD(3) declaration() |

                                expression() ";" | ";")(expression())? ";"

                                (expression())? ")" statement_block()

selection_statement() :         "if" "(" expression() ")" statement_block()

                                (LOOKAHEAD(2) "else" statement_block())?

                                | "switch" "(" expression() ")" "{" ( case_label()

                                case_block() )*  "}"

case_label()            :       "case" constant_expression() ":" | "default" ":"

case_block()            :       ( statement() )*
```

4.3.2.4 Example of ASTs Generated using Modified C++ Grammar

Consider the two code fragments:

Fragment 1 (*if* without block)	**Fragment 2 (*if* with block)**

if (ch==1)
cout<<"one";
else
cout<<"wrong choice!";
	if (ch==1)
	{ cout<<"one";}
	else
	{ cout<<"wrong choice!";}

Figure 4.10 shows the ASTs generated using original and modified C++ grammar for the two given code fragments.

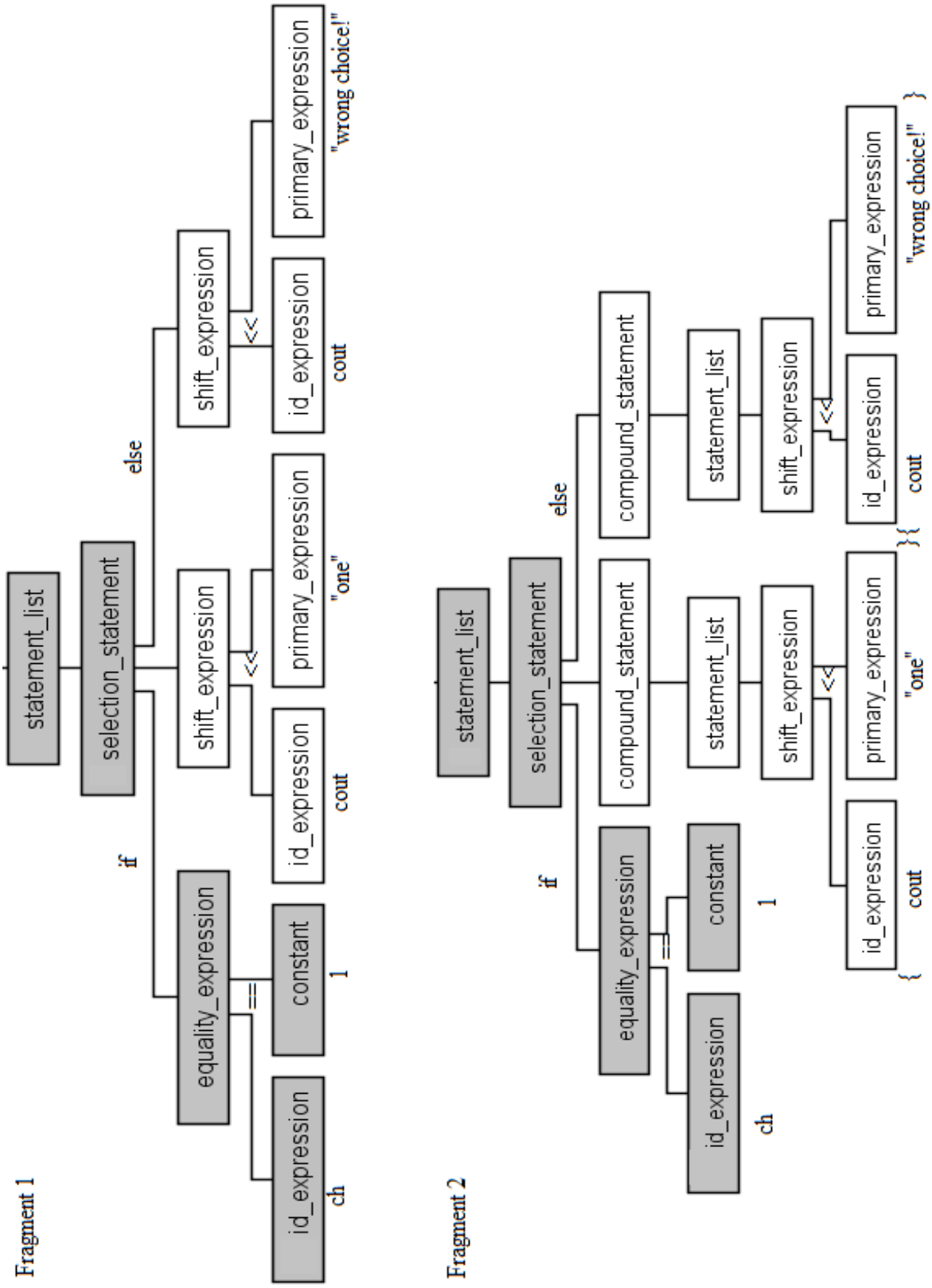


Figure 4.10. (a)

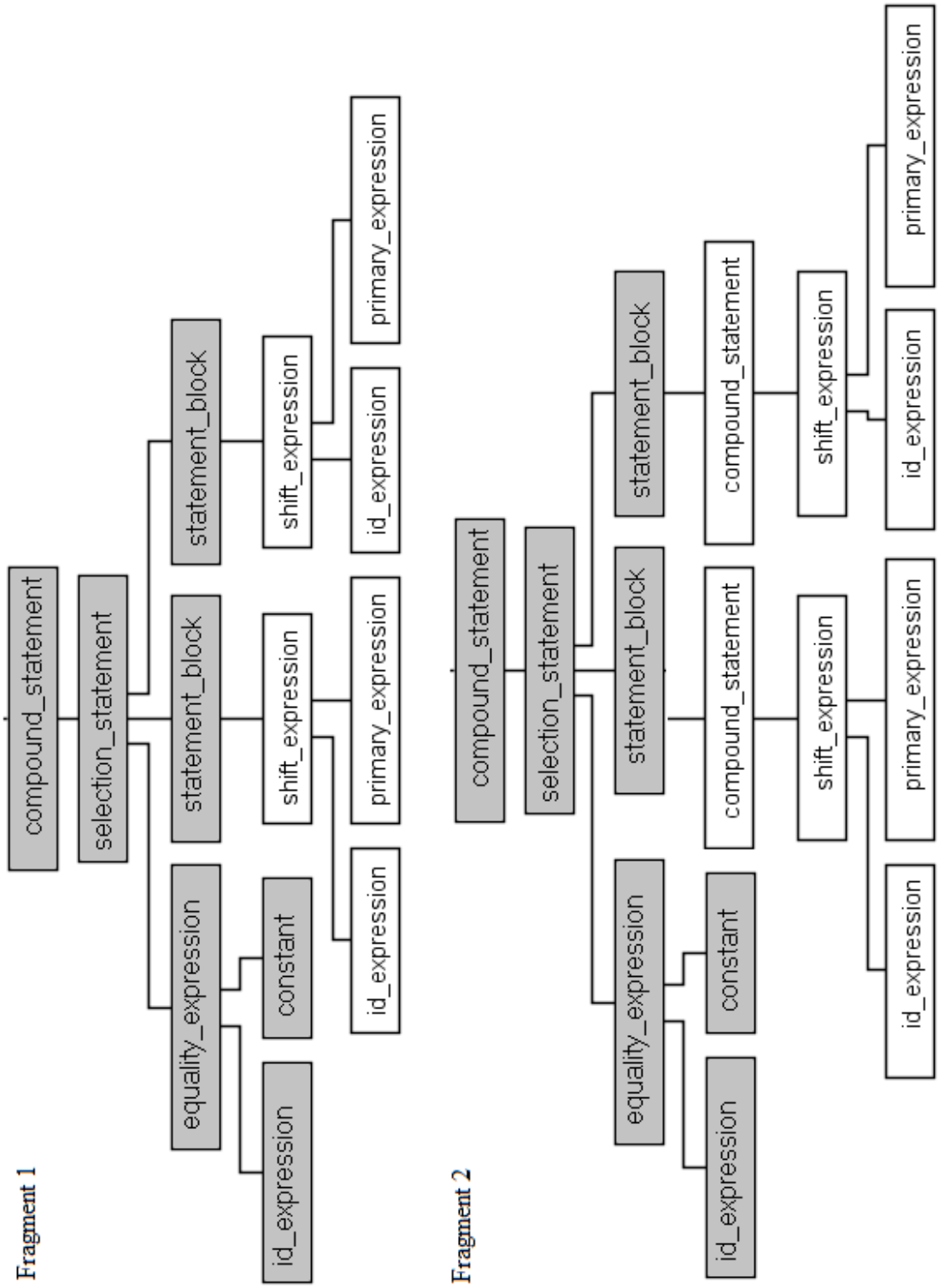
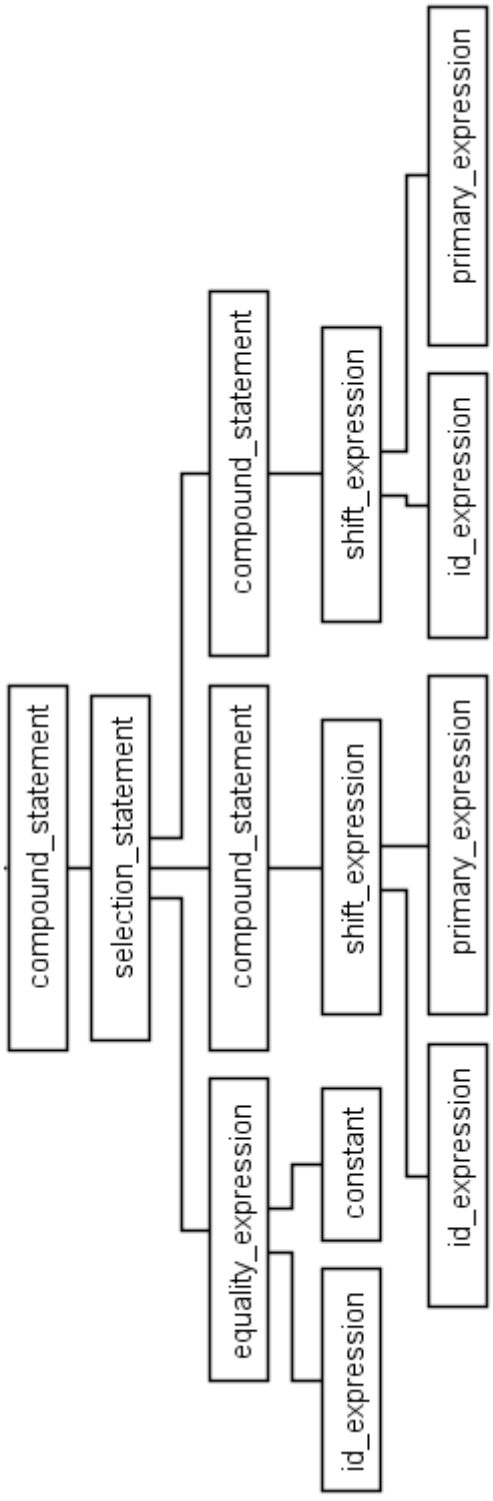

Fragment 1
compound_statement
selection_statement
equality_expression
id_expression
constant
statement_block
shift_expression
id_expression
primary_expression
shift_expression
id_expression
primary_expression
Fragment 2
compound_statement
selection_statement
equality_expression
id_expression
constant
statement_block
compound_statement
shift_expression
id_expression
primary_expression
compound_statement
shift_expression
id_expression
primary_expression

Figure 4.10. (b)



(c)

Figure 4.10. ASTs Generated using Original and Modified C++ Grammar a) ASTs Generated using Original Grammar for Fragments 1 and 2 b) ASTs Generated using Modified C++ Grammar for Fragments 1 and 2 c) AST for *if* without Block with Label StatementBlock Changed to CompoundStatement or for *if* with Block with Node Labeled StatementBlock Removed

Figure 4.10.a shows the ASTs obtained using original grammar for *if* with and without block aligned with each other. It can be noted that the subtrees corresponding to the body of *if* in the two ASTs do not match.

Figure 4.10.b shows the ASTs obtained using modified C++ grammar for *if* without block and *if* with block. The subtrees corresponding to body of *if* in both the ASTs are now rooted at node labeled statement_block. It can be noted from figure 4.10.b that changing the label statement_block to compound_statement in the AST for *if* without block and removing the node labeled statement_block from the AST for *if* with block yields the same AST as given in figure 4.10.c.

4.3.3 Modifications to Java Grammar and AST

4.3.3.1 Java Grammar

```
Statement()          :    LOOKAHEAD(2)  LabeledStatement()
                          | AssertStatement() | Block() | EmptyStatement()
                          | StatementExpression() ";" | SwitchStatement()
                          | IfStatement() | WhileStatement() | DoStatement()
                          | ForStatement() | BreakStatement()
                          | ContinueStatement() | ReturnStatement()
                          | ThrowStatement() | SynchronizedStatement()
                          | TryStatement()
SwitchStatement()    :    "switch" "(" Expression() ")" "{" ( SwitchLabel()
                          SwitchLabelBlock() )* "}"
SwitchLabel()        :    "case" Expression() ":" | "default" ":"
SwitchLabelBlock()   :    ( BlockStatement() )*
IfStatement()        :    "if" "(" Expression() ")" Statement()
```

```
                        [LOOKAHEAD(1) "else" Statement() ]

WhileStatement()    :   "while" "(" Expression() ")" Statement()

DoStatement()       :   "do" Statement() "while" "(" Expression() ")" ";"

ForStatement()      :   "for" "(" (LOOKAHEAD(Type() <IDENTIFIER> ":")

                        Type() <IDENTIFIER> ":" Expression() | [ ForInit() ]

                        ";" [ Expression() ] ";" [ ForUpdate() ] ) ")" Statement()
```

4.3.3.2 Modifying Java Grammar and AST

Ligaarden (2007) makes a distinction between the different types and between the literals of different types on modifying the grammar. Making a type distinction and literal distinction will only help to discriminate the files rather than finding their similarity. It is therefore necessary to retain the original grammar rules for primitive types and literals to identify type 1 plagiarism effectively.

In the original Java1.5 grammar, there are separate rules for the selection statements *if* and *switch*. In case of different rules for the selection statements, the comparison stops at nodes labeled IfStatement and SwitchStatement since the labels do not match. Similarly, there are separate rules for *for*, *while*, and *do-while*. The comparison stops at nodes labeled ForStatement, WhileStatement, and DoStatement since the labels do not match. Hence, the rules are modified so that the separate rules for *if* and *switch* are combined to form a new rule SelectionStatement and the separate rules for *for*, *while*, and *do-while* are combined to form a new rule IterationStatement.

The modification done to C and C++ grammar is also done to Java grammar. A new rule StatementBlock is added and the rules for IterationStatement and SelectionStatement are redefined using StatementBlock. The AST generated with this modified grammar will always have StatementBlock as root of the subtree which corresponds to the body of iteration or selection statement.

The subtree for a StatementBlock either corresponds to a single statement or a compound statement. If the iteration statement or selection statement has a single statement as its body then the node label StatementBlock in the AST generated is

changed to CompoundStatement. If the iteration statement or selection statement has a compound statement as its body then the node StatementBlock is removed.

The rule for *switch* statement is changed so that subtree rooted at CaseBlock corresponding to each *case* block with one or more statements is separated from the other. The root of each of these subtrees corresponding to each of the *case* blocks in *switch* statement is changed from CaseBlock to CompoundStatement to allow subtree matching. In the modified AST, the iteration statements *for*, *while* and *do-while* or the selection statement *if-else* will always have CompoundStatement as root of its subtree which corresponds to the body of iteration or *if-else* statement and the *case* blocks in *switch* will also be rooted at CompoundStatement. This modification allows the subtrees of iteration or selection statement with and without block to be matched.

4.3.3.3 Modified Java Grammar

```
StatementBlock()      :    Statement()

Statement()           :    LOOKAHEAD(2)  LabeledStatement()
                           | AssertStatement() | Block() | EmptyStatement()
                           | StatementExpression() ";" | IterationStatement()
                           | SelectionStatement() | BreakStatement()
                           | ContinueStatement() | ReturnStatement()
                           | ThrowStatement() | SynchronizedStatement()
                           | TryStatement()

SelectionStatement()  :    "if" "(" Expression() ")" StatementBlock()
                           [LOOKAHEAD(1) "else" StatementBlock() ]
                           | "switch" "(" Expression() ")" "{" (CaseLabel()
                           CaseBlock() )* "}"

CaseLabel()           :    "case" Expression() ":" |  "default" ":"
```

```
CaseBlock()          :    ( BlockStatement() )*

IterationStatement()  :   "while" "(" Expression() ")" StatementBlock()
                          | "do" StatemenBlock() "while" "(" Expression() ")" ";"
                          | "for" "(" (LOOKAHEAD(Type() <IDENTIFIER> ":")
                          Type() <IDENTIFIER> ":" Expression() | [ ForInit() ]
                          ";" [ Expression() ] ";" [ ForUpdate() ] ) ")"
                          StatementBlock()
```

4.3.3.4 Example of ASTs Generated using Modified Java Grammar

Fragment 1 **(*if* with block)**	**Fragment 2** **(*switch* statement)**

if (ch==1)
{
System.out.println("one");
}
else
{
System.out.println("wrong choice!");
}
	switch(ch)
	{
	case 1:
	System.out.println("one");
	break;
	default:
	System.out.println("wrong choice!");
	}

Figure 4.11.a shows the ASTs generated using original grammar for the two code fragments where the ASTs do not match due to the difference in labels of the root of the subtrees corresponding to *if* and *switch* statements.

Figure 4.11.b shows the ASTs obtained using modified C grammar for the two code fragments. Figure 4.11.c shows the ASTs obtained after modifying them to allow subtree matching.

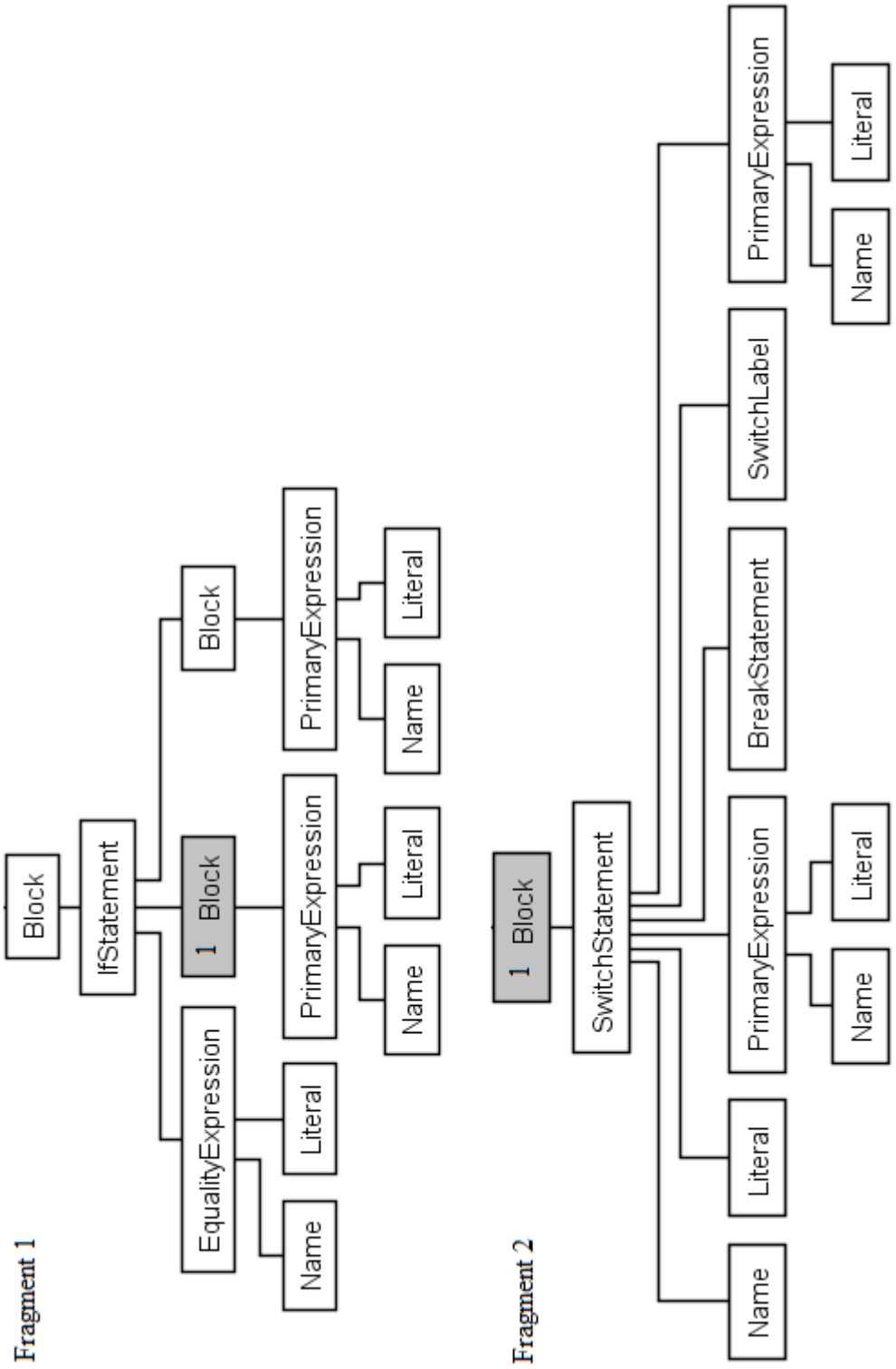

Fragment 1
Block
IfStatement
EqualityExpression
Name
Literal
1 Block
PrimaryExpression
Name
Literal
Block
PrimaryExpression
Name
Literal
Fragment 2
1 Block
SwitchStatement
Name
Literal
PrimaryExpression
Name
Literal
BreakStatement
SwitchLabel
PrimaryExpression
Name
Literal

Figure 4.11. (a)

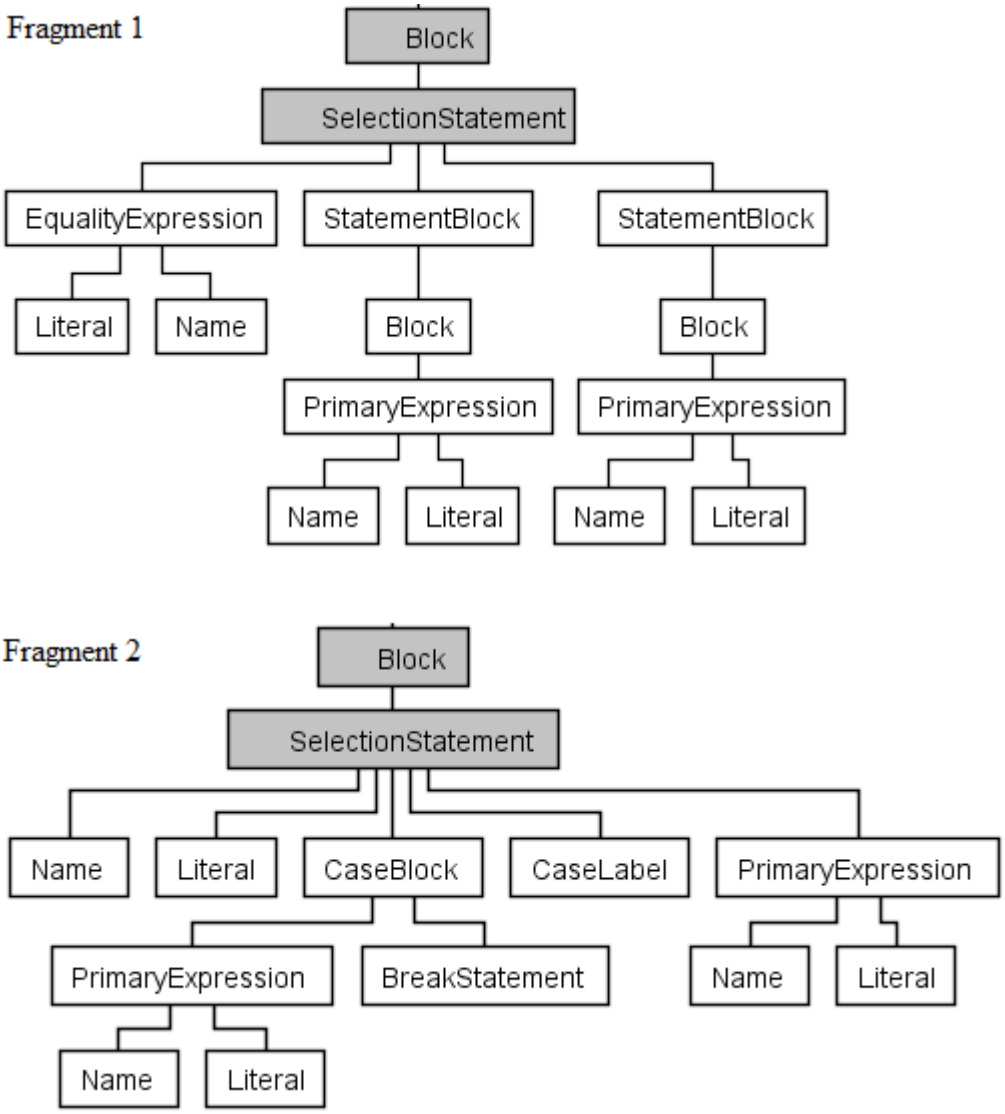
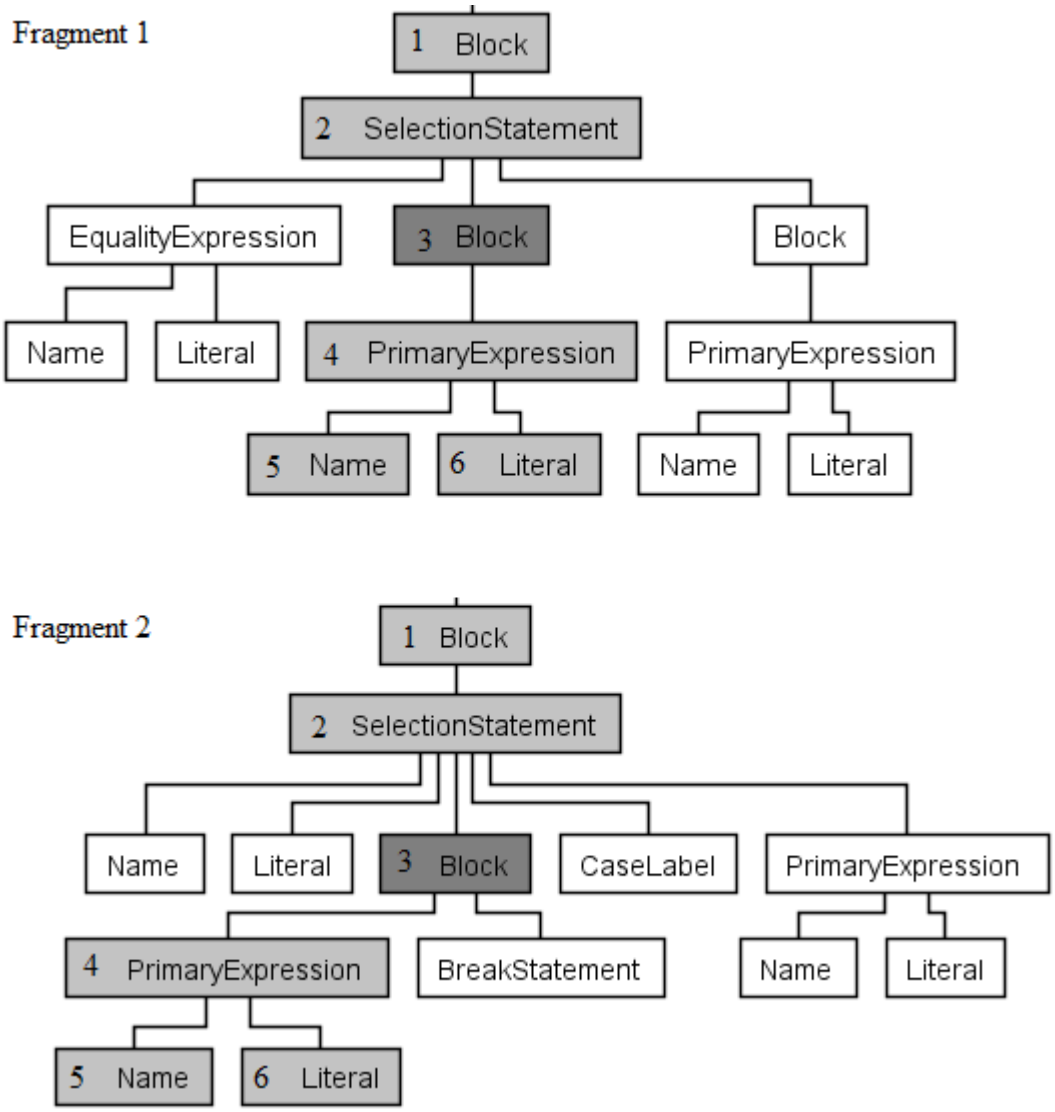

Fragment 1
Block
SelectionStatement
EqualityExpression
StatementBlock
StatementBlock
Literal
Name
Block
Block
PrimaryExpression
PrimaryExpression
Name
Literal
Name
Literal
Fragment 2
Block
SelectionStatement
Name
Literal
CaseBlock
CaseLabel
PrimaryExpression
PrimaryExpression
BreakStatement
Name
Literal
Name
Literal

Figure 4.11. (b)



(c)

Figure 4.11. ASTs Generated using Original and Modified Java Grammar a) ASTs Generated using Original Java Grammar for Fragment1 – *if* with Block and Fragment 2 – *switch* statement b) ASTs Generated using Modified Java Grammar for Fragment 1 and Fragment 2 c) Modified ASTs for Fragment 1 and Fragment 2.

4.4 AST ALGORITHM

For a set of source code files to be checked for detecting plagiarisms, the first step is to generate their ASTs. ASTs of the source code files are then traversed in preorder and the resulting node sequences are compared using sequence matching algorithms. A similarity score is also computed based on the matches detected.

Steps involved in AST matching are:

1. Generate ASTs using modified grammar for each of the source code files to be checked for plagiarism.
2. Do a pre-order traversal of each AST.
3. Store the resulting node sequences.
4. Detect similarities in the stored node sequences by applying:
    a) TDUMCSI Algorithm
    b) NW Algorithm
    c) LCS Algorithm
5. Compute similarity scores for each pair of files using the detected matches.
6. Mark files with a high degree of similarity as plagiarized.

4.4.1 Top-Down Unordered Maximum Common Subtree Isomorphism Algorithm

Tree Isomorphism

Trees can be either ordered or unordered and labelled or unlabelled. Two trees are isomorphic if one of them can be obtained from the other through a series of swaps of left and right children of any number of nodes at any of the levels. Two unlabelled trees are isomorphic if they have the same tree structure. Two labelled trees are isomorphic if they have the same tree structure and the corresponding nodes in the two trees have the same labels. Different tree isomorphisms (Valiente, 2002) are being discussed in this section.

Ordered Tree Isomorphism

Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be two ordered trees and $M \subseteq V_1 \times V_2$ be a bijective mapping between the node sets. M is an ordered tree isomorphism of T_1 to T_2 if the following conditions hold:

- (root[T_1], root[T_2]) ∈ M
- (first[v], first[w]) ∈ M for all non-leaves $v \in V_1$ and $w \in V_2$ with (v,w) ∈ M
- (next[v], next[w]) ∈ M for all non-last children $v \in V_1$ and $w \in V_2$ with (v,w) ∈ M

Unordered Tree Isomorphism

Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be two unordered trees and $M \subseteq V_1 \times V_2$ be a bijective mapping between the node sets. M is an unordered tree isomorphism of T_1 to T_2 if the following conditions hold:

- (root[T_1], root[T_2]) ∈ M
- (parent[v], parent[w]) ∈ M for all non-roots $v \in V_1$ and $w \in V_2$ with (v,w) ∈ M

Maximum Common Subtree

A maximum common subtree of two ordered or unordered trees is the largest subtree common to both the trees. Common subtree of the two trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ is defined as a structure (X_1, X_2, M), where $X_1 = (W_1, S_1)$ is a subtree of T_1, $X_2 = (W_2, S_2)$ is a subtree of T_2, and $M \subseteq W_1 \times W_2$ is a tree isomorphism of X_1 to X_2. A common subtree (X_1, X_2, M) of T_1 to T_2 is maximum if there is no subtree (X_1', X_2', M') of T_1 to T_2 with size[X_1] < size[X_1'].

Top-Down Ordered Maximum Common Subtree Isomorphism

A top-down common subtree of an ordered tree $T_1 = (V_1, E_1)$ to another ordered tree $T_2 = (V_2, E_2)$ is a structure (X_1, X_2, M), where $X_1 = (W_1, S_1)$ is a top-down ordered subtree of T_1, $X_2 = (W_2, S_2)$ is a top-down ordered subtree of T_2, and $M \subseteq W_1 \times W_2$ is an ordered tree isomorphism of X_1 to X_2. A top-down common subtree (X_1, X_2, M) of T_1 to T_2 is maximal if there is no top-down common subtree (X_1', X_2', M') of T_1 to T_2, where X_1 is a proper top-down subtree of X_1' and X_2 is a proper top-down subtree of X_2', with size[X_1] < size[X_1'].

In top-down ordered maximum common subtree isomorphism, the comparison is between two trees with identical structures and nodes with corresponding labels.

For a node v to be a part of maximum common subtree, the parent node, say w, and all the left siblings of v should be part of maximum common subtree. A drawback of this algorithm is that it cannot detect changes in the order of independent code and operands.

Top-Down Unordered Maximum Common Subtree Isomorphism

When changes in the order of independent code are to be detected, an unordered approach is preferred. This method does matching by rotating the different declarations in the second tree so that they are in same order as the declarations in first tree. It also finds the best unordered matching between subtrees of different statements and local variable declarations and unordered matching of operands in expressions.

Top-down maximum common subtree isomorphism of an unordered tree $T_1 = (V_1, E_1)$ into another unordered tree $T_2 = (V_2, E_2)$ is constructed from the maximum common subtree isomorphisms of each of the subtrees rooted at the children of node v in T_1 into each of the subtrees rooted at the children of node w in T_2.

Start at the root nodes of T_1 and T_2. For each node $v \in V_1$ and $w \in V_2$, the maximum common subtree isomorphism between the subtree rooted at v and the subtree rooted at w is obtained as long as the labels correspond. If the labels do not correspond, then the maximum common subtree has size zero. If v or w is a leaf node, the maximum common subtree has size 1 if $T_1[v] = T_2[w]$. Otherwise, maximum weight bipartite matching algorithm is used to find the size of maximum common subtree of the two tree structures of non-zero size.

A bipartite graph is a graph where the vertices can be partitioned into two disjoint vertex sets. Let p be the number of children of node v in T_1 and let q be the number of children of node w in T_2. Let the children of the two nodes be represented as $v_1, \ldots, v_p$ and $w_1, \ldots, w_q$. A bipartite graph $G = (\{v_1, \ldots, v_p\}, \{w_1, \ldots, w_q\}, E)$ with $p + q$ vertices is then built with $(v_i, w_j) \in E$ if and only if the size of a maximum common subtree of the subtree rooted at node vi of T_1 and the subtree rooted at wj of T_2 is nonzero, and with edge $(v_i, w_j) \in E$ weighted by that nonzero size.

The maximum common subtree of *v* and *w* then has size 1 (since *v* and *w* can be mapped against each other) plus the weight of the maximum weight bipartite matching in *G*. An example of top-down unordered maximum common subtree isomorphism is provided in (Ligaarden, 2007).

Normalized distance function $d_{tdumcsi}(p,q)$ and normalized similarity score $sim_{tdumcsi}(p,q)$ of two programs *p* and *q* computed based on top-down maximum common subtree isomorphism are defined as

$$d_{tdumcsi}(p,q) = \frac{|V_1| + |V_2| - 2|M|}{|V_1| + |V_2|} \tag{4.1}$$

$$sim_{tdumcsi}(p,q) = \frac{2|M|}{|V_1| + |V_2|} \tag{4.2}$$

where $|V_1|$ and $|V_2|$ are the number of nodes in the ASTs of *p* and *q* respectively and $|M|$ is the number of nodes in the maximum common subtree.

Unordered approach cannot find good alignment between statements and local variable declarations of two blocks. A solution is to use the NW algorithm. Moreover, TDUMCSI algorithm gives a low similarity for trees of different loop statements and different selection statements. A solution to this is to use LCS algorithm for the subtrees.

4.4.2 Needleman-Wunsch Algorithm

Needleman-Wunsch algorithm is a dynamic programming based algorithm usually used in bioinformatics for sequence alignment. It is a technique of arranging the sequences in such a way as to identify the regions of similarity between the sequences. It is an algorithm to find the global alignment between two sequences. That is, it finds the best alignment over the entire length of two sequences and is most suitable when the two sequences are of similar length. Scores for the aligned characters are obtained using a similarity matrix. To penalize the gaps in the alignment, it also uses a linear gap penalty.

4.4.2.1 NW Score Calculation

Given two sequences *X* and *Y* of lengths *x* and *y* respectively, a score matrix is created with *x+1* rows and *y+1* columns. The first row and first column is used to represent gap. Remaining *x* rows represent the *x* characters in *X* and remaining *y* columns represent the *y* characters in *Y*. An NW score of prefixes X_i and Y_j of the two sequences *X* and *Y*, denoted by *c(i,j)*, is given by

$$c(i,j)=\begin{cases} 0 & if\ i=0\ and\ j=0 \\ g.i & if\ i\neq 0\ and\ j=0 \\ g.j & if\ i=0\ and\ j\neq 0 \\ \max(c(i-1,j-1)+S(i,j), & \\ c(i,j-1)+g, & otherwise \\ c(i-1,j)+g) & \end{cases} \tag{4.3}$$

where *S(i,j)* denotes the alignment score between two characters *i* and *j* obtained from the substitution matrix *S* and *g* is the linear gap penalty.

4.4.2.2 Example

Consider the two sequences to be aligned: *X* = TCG (*x*=3, length of sequence 1) and *Y* = ATC (*y*=3, length of sequence 2).

Scoring scheme

- Match Score = +1
- Mismatch Score = -1
- Gap penalty = -1

Substitution Matrix S

$$S(i,j)=\begin{cases} 1, \text{ if the characters in both the sequences match} \\ \\ \text{-1, if the characters in both the sequences do not match} \end{cases}$$

A unit matrix scheme can also be followed which gives a score of 1 for a match and 0 for a mismatch.

	A	C	G	T
A	1	-1	-1	-1
C	-1	1	-1	-1
G	-1	-1	1	-1
T	-1	-1	-1	1

Three steps involved are:

1. Initialization
2. Scoring
3. Trace back (Alignment)

Initialization:

Create a score matrix with $x+1$ rows and $y+1$ columns. Set $c(i,j) = g.i$, for $i \neq 0$ and $j = 0$, and $c(i,j) = g.j$, for $i = 0$ and $j \neq 0$.

		T	C	G
	0	-1	-2	-3
A	-1			
T	-2			
C	-3			

That is, the 1st row and the 1st column of the score matrix are filled as multiples of gap penalty.

Scoring:

The score of any cell $c(i,j)$ depends only on three adjacent cell values. The score of any cell $c(i, j)$ is the maximum of:

scorediag = *c (i-1, j-1) + S(i, j)*

scoreup = *c(i-1, j) + g*

scoreleft = *c(i, j-1) + g*

where *S(i,j)* is the substitution score for letters *i* and *j*, and *g* is the gap penalty.

c(i-1,j-1)	c(i-1,j)
c(i,j-1)	c(i,j)

An arrow is used to indicate from which of the three neighbouring cells (diagonal, up, and left) the maximum score is obtained.

***Example**:* The calculation for the cell c(2, 2):

c(2,2) = max (scorediag, scoreup, scoreleft)

=max(c(i-1,j-1)+S(i,j),c(i-1,j)+g,c(i,j-1)+g)

= max (c(1,1)+S(2,2), c(1,2)+g, c(2,1)+g)

= max (0+(-1), -1+(-1), -1+(-1))

= max (-1, -2, -2)

= -1

		T	C	G
	0	-1	-2	-3
A	-1	-1		
T	-2			
C	-3			

c(2,3) = max (scorediag, scoreup, scoreleft)

= max (c(1,2) + S(2,3), c(1,3)+g, c(2,2)+g

= max (-1+(-1)), -2+(-1), -1+(-1))

= max (-2, -3, -2)

= -2

		T	C	G
	0	-1	-2	-3
A	-1	-1	-2	
T	-2			
C	-3			

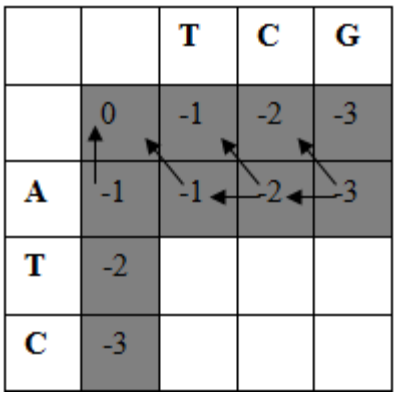
c(2,4) = max(scorediag, scoreup, scoreleft)

=max(c(1,3)+S(2,4),c(1,4)+g, c(2,3)+g)

=max (-2+(-1)), -3+(-1), -2+(-1))

= max (-3, -4, -3)

= -3



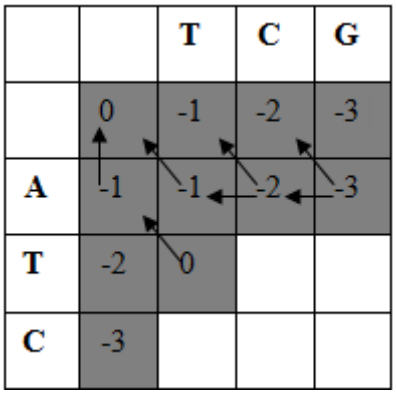
c(3,2) = max (scorediag, scoreup, scoreleft)

=max (c(2,1)+S(3,2), c(2, 2)+g, c(3,1)+g)

= max (-1+1, -1+(-1), -2+(-1))

= max (0, -2, -3)

= 0



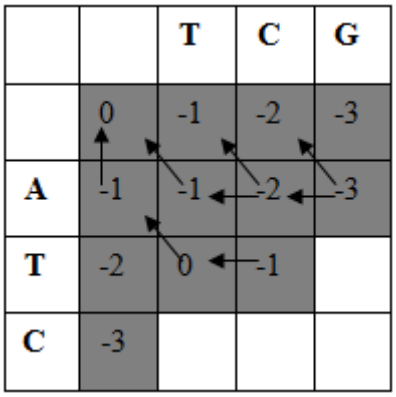
c(3,3) = max (scorediag, scoreup, scoreleft)

=max (c(2,2)+S(3,3), c(2,3)+g, c(3,2)+g)

= max (-1+(-1), -2+(-1), 0+(-1))

= max (-2, -3, -1)

= -1



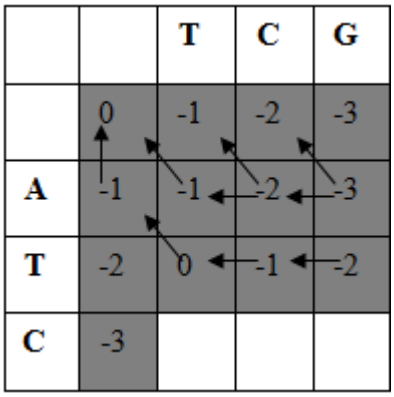
c(3,4) = max (scorediag, scoreup, scoreleft)

=max (c(2,3)+S(3,4), c(2,4)+g, c(3,3)+g)

= max (-2+(-1), -3+(-1), -1+(-1))

= max (-3, -4, -2)

= -2

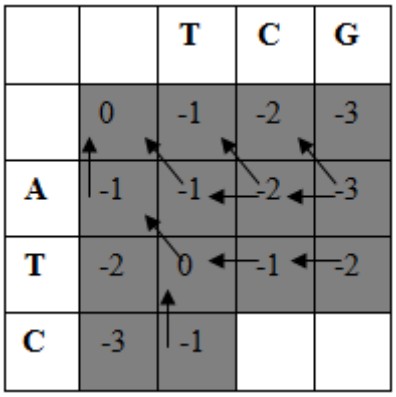


c(4,2) = max (scorediag, scoreup, scoreleft)

= max (-2+(-1), 0+(-1), -3+(-1))

= max (-3,-1,-4)

= -1

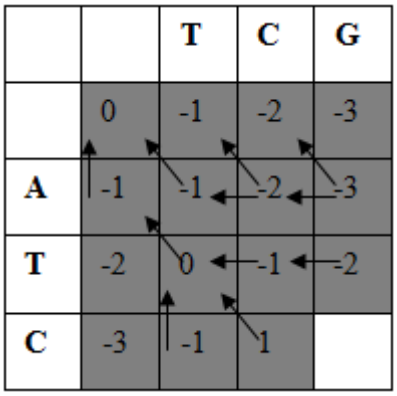


c(4,3) = max (scorediag, scoreup, scoreleft)

= max (0+1, -1+(-1), -1+(-1))

= max (1,-2,-2)

= 1



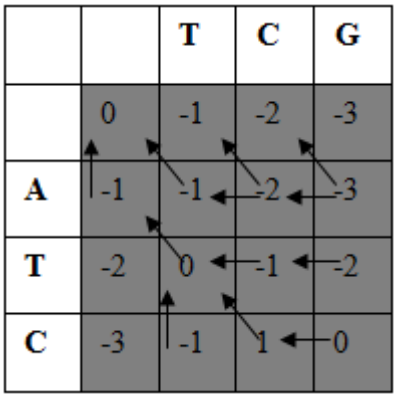
Final Scoring Matrix

c(4,4) = max (scorediag, scoreup, scoreleft)

= max (-1+(-1), -2+(-1), 1+(-1))

= max (-2,-3,0)

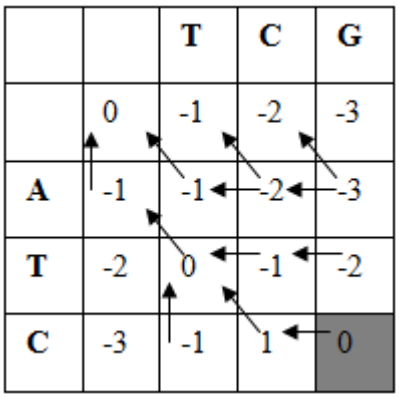
= 0



Trace Back:

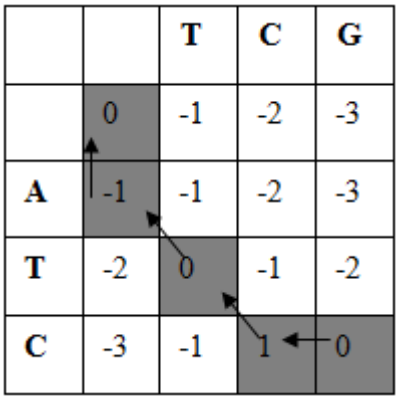
- The trace back step determines the actual alignment(s) that result in the maximum score.
- There can be multiple maximal alignments. Trace back starts from the last cell (bottom right corner) in the matrix and gives alignment in reverse order.

- There are three possible moves: diagonal (towards the top-left corner of the matrix), up, or left. Trace back takes the current cell and looks to the neighbor cells that could be direct predecessors. This means it checks :
    - the neighbor to the left, (a gap is introduced in the left sequence),
    - the diagonal neighbor (match), and
    - the neighbor above it (a gap is introduced in top sequence).
- The algorithm for trace back chooses one of the possible predecessors as the next cell in the sequence. In the example, starting from the last cell, the only possible predecessor is the left neighbor.



- Trace back is completed when the first cell (top left corner) is reached.

Final Trace Back



The alignment which gives the maximum score corresponding to the one with maximum possible match or the best alignment is:

Seq. 1: _ **T C** G

Seq. 2: A **T C** _

Normalized distance function $d_{NW}(p,q)$ and normalized similarity score $sim_{NW}(p,q)$ of two programs p and q computed based on NW algorithm are defined as

$$d_{NW}(p,q) = \frac{|V_1| + |V_2| - 2NW(p,q)}{|V_1| + |V_2|} \tag{4.4}$$

$$sim_{NW}(p,q) = \frac{2NW(p,q)}{|V_1| + |V_2|} \tag{4.5}$$

where $|V_1|$ and $|V_2|$ are the number of nodes in the ASTs of p and q respectively.

With respect to the source code plagiarism detection application domain, NW algorithm is used to find good alignment between statements and local variable declarations of two blocks (Ligaarden, 2007). The rows and columns of the similarity or substitution matrix represent the statements and local variable declarations of the two blocks to be matched. Scores in the substitution matrix are the sizes of the different maximum common subtree isomorphisms of each of the subtrees rooted at the children of one node into each of the subtrees rooted at the children of other node.

For detecting matches between two blocks of code, say *A* and *B*, a score matrix is created with dimensions $(m+1) \times (n+1)$, where m and n are the total number of statements in blocks *A* and *B* respectively. Each row of the score matrix, say *C*, represents statements and variable declarations of *A* and each column represents statements and variable declarations of *B*. NW score is calculated and each cell $c(i,j)$ in the matrix will be an optimal solution of the alignment of the first i statements and local variable declarations in *A* with the first j statements and local variable declarations in *B*.

4.4.3 Longest Common Subsequence Algorithm

LCS algorithm is also used for global sequence alignment. LCS finds the best possible alignment by comparing the sequences from the beginning till the end. It is usually used when the sequences to be compared are of similar length. The LCS

algorithm finds the longest common subsequence of two sequences. It uses dynamic programming technique where a complex problem is broken down into simpler sub-problems and it is solved by combining the solutions of the sub-problems. Every sub-problem is solved just once and the result is stored for later use.

4.4.3.1 LCS Length Calculation

Given two sequences *X* and *Y* of lengths *x* and *y* respectively, a score matrix is created with *x+1* rows and *y+1* columns. The first row and first column is used to represent gap. Remaining *x* rows represent the *x* characters in *X* and remaining *y* columns represent the *y* characters in *Y*. Length of LCS of prefixes X_i and Y_j of sequences *X* and *Y*, denoted by *c[i,j]*, is given by

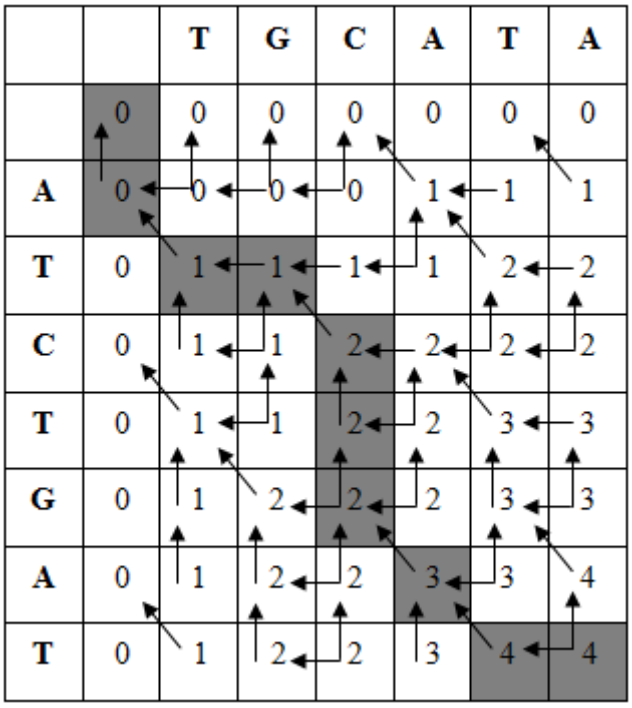
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1]+1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases} \quad (4.6)$$

4.4.3.2 Example

Consider the two sequences: *X*= (ATCTGAT) and *Y* = (TGCATA). The best alignment between *X* and *Y* can be obtained by constructing the LCS score matrix as given in equation 4.6. Trace back is performed to determine the alignment which gives the maximum score. The score matrix obtained is:

		T	G	C	A	T	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
T	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
T	0	1	1	2	2	3	3
G	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
T	0	1	2	2	3	4	4

There may be multiple maximal alignments as with NW algorithm. Two possible maximal alignments of *X* and *Y* obtained after trace back are:

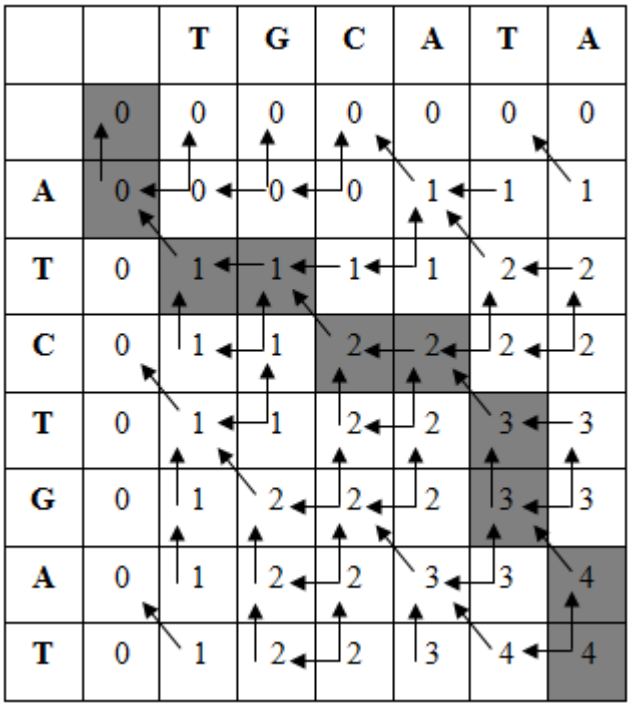


		T	G	C	A	T	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
T	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
T	0	1	1	2	2	3	3
G	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
T	0	1	2	2	3	4	4

_ **T** G **C** _ _ **A** **T** A

A **T** _ **C** T G **A** **T** _

LCS for *X* and *Y* is obtained as TCAT.



		T	G	C	A	T	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
T	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
T	0	1	1	2	2	3	3
G	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
T	0	1	2	2	3	4	4

_ **T** G **C** A **T** _ **A** _

A **T** _ **C** _ **T** G **A** T

LCS for *X* and *Y* is obtained as TCTA.

Normalized distance function $d_{LCS}(p,q)$ and normalized similarity score $sim_{LCS}(p,q)$ of programs *p* and *q* computed based on LCS algorithm are defined as

$$d_{LCS}(p,q) = \frac{|V_1| + |V_2| - 2LCS(p,q)}{|V_1| + |V_2|} \tag{4.7}$$

$$sim_{LCS}(p,q) = \frac{2LCS(p,q)}{|V_1| + |V_2|} \tag{4.8}$$

where $|V_1|$ and $|V_2|$ are the number of nodes in the ASTs of *p* and *q* respectively.

With respect to the source code plagiarism detection application domain, LCS is used to find similarity between method bodies (Ligaarden, 2007). Consider two ASTs T_1 and T_2. Let W_1 and W_2 consist of nodes in the method bodies to be compared of T_1 and T_2 respectively. A preorder traversal is done on the two subtrees that represent the method bodies to generate two ordered sequences of nodes. For all nodes $v_i \in W_1$ and $w_j \in W_2$, *i* and *j* denote in which order the nodes are visited during the traversal.

To find an alignment with maximum score, first a score matrix *C* is created with dimensions $(m+1) \times (n+1)$. First row and first column represent the gap character. Each row represents each of the nodes from the preorder traversal of W_1. Each column represents each of the nodes from the preorder traversal of W_2. LCS score is calculated and each cell *c[i,j]* in the matrix will be an optimal solution of the alignment of the first *i* nodes in preorder traversal of W_1 with the first *j* nodes in the preorder traversal of W_2.

Top down unordered maximum common subtree isomorphism is used for the tree as a whole. Since it does an unordered matching, it can find good match between independent structures but it fails to find good match between the statements and local

variable declarations of two blocks and also between the trees of different loops and different selection statements. This is solved by using NW and LCS algorithms.

4.5 IMPLEMENTATION AND TESTING

The AST generation for C, C++ and Java languages uses open source scanner and parser generator Java Compiler Compiler (JavaCC) and tree builder JJTree. JJTree is a preprocessor for JavaCC that inserts parse tree building actions at various places in the JavaCC source. C, C++ and Java grammars are modified as given in section 4.3. JavaCC and JJTree are used to construct the ASTs for source code files written in C, C++ and Java.

4.5.1 Testing

In the initial stage, parse trees were used to assess similarity between source code files. Parse trees generated using original grammars for C, C++ and Java were then reduced in size by eliminating unnecessary nodes to form ASTs. ASTs were later generated using modified grammars for C, C++ and Java.

Consider the program HelloWorld.c

```
#include<stdio.h>
main()
{
printf("Hello World");
}
```

Figure 4.12 shows the parse tree and AST generated for the program HelloWorld.c. The parse tree generated, without any modification in C grammar, for the simple C program consisted of 41 levels and 44 nodes (including 3 leaf nodes). Its AST, generated using modified C grammar, has only 6 levels and 8 nodes.

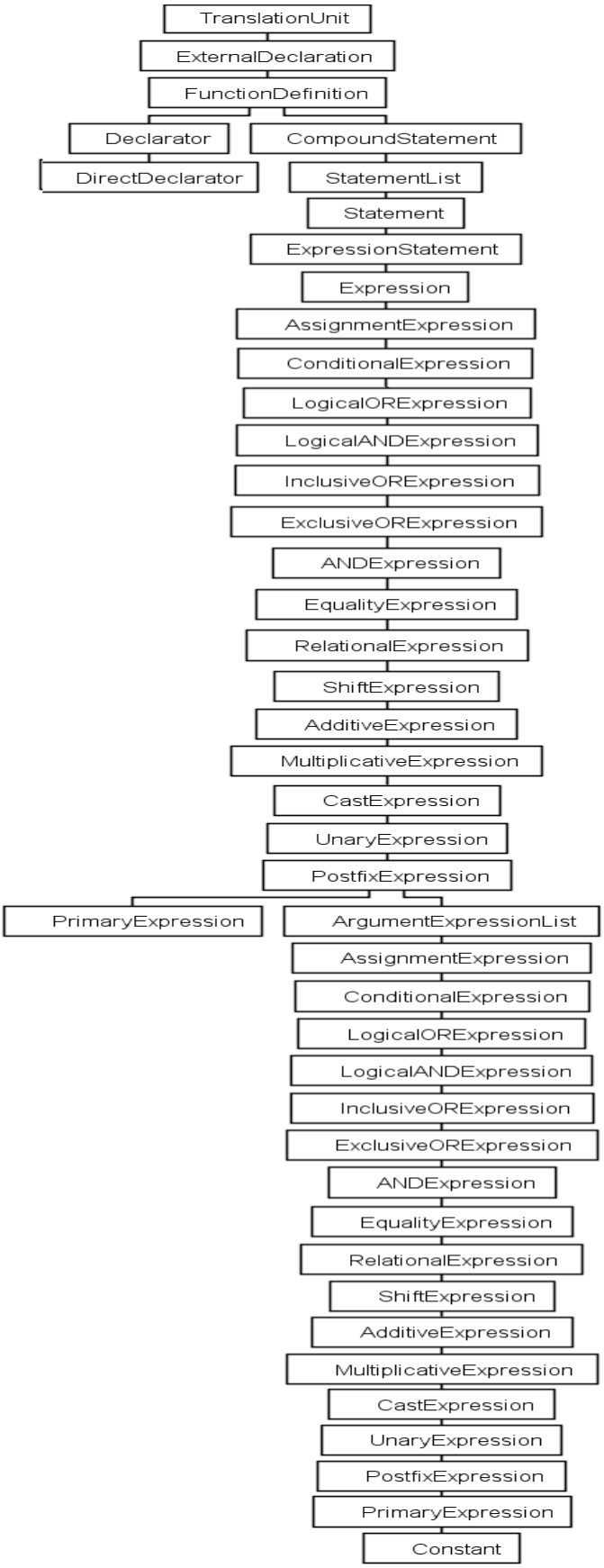
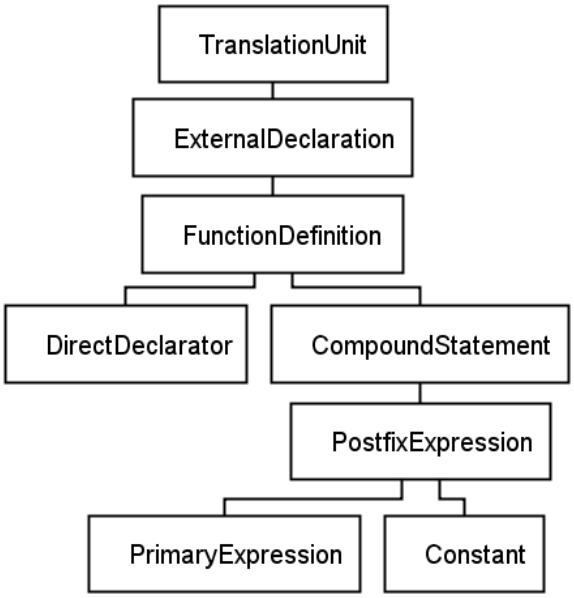

TranslationUnit
ExternalDeclaration
FunctionDefinition
Declarator
CompoundStatement
DirectDeclarator
StatementList
Statement
ExpressionStatement
Expression
AssignmentExpression
ConditionalExpression
LogicalORExpression
LogicalANDExpression
InclusiveORExpression
ExclusiveORExpression
ANDExpression
EqualityExpression
RelationalExpression
ShiftExpression
AdditiveExpression
MultiplicativeExpression
CastExpression
UnaryExpression
PostfixExpression
PrimaryExpression
ArgumentExpressionList
AssignmentExpression
ConditionalExpression
LogicalORExpression
LogicalANDExpression
InclusiveORExpression
ExclusiveORExpression
ANDExpression
EqualityExpression
RelationalExpression
ShiftExpression
AdditiveExpression
MultiplicativeExpression
CastExpression
UnaryExpression
PostfixExpression
PrimaryExpression
Constant

Figure 4.12. (a)



(b)

Figure 4.12. Parse Tree and AST for HelloWorld.c a) Parse Tree for Helloworld.c (Original C Grammar) b) AST for Helloworld.c (Modified C Grammar)

Consider the program HelloWorld.cpp.

```
#include <iostream.h>
void main()
{
cout << "Hello World!";
}
```

Figure 4.13 shows the parse tree and AST generated for HelloWorld.cpp.

The parse tree generated, without any modification in C++ grammar, for the simple C++ program consisted of 26 levels and 38 nodes (including 4 leaf nodes). Its AST, generated using modified C++ grammar, has only 7 levels and 10 nodes.

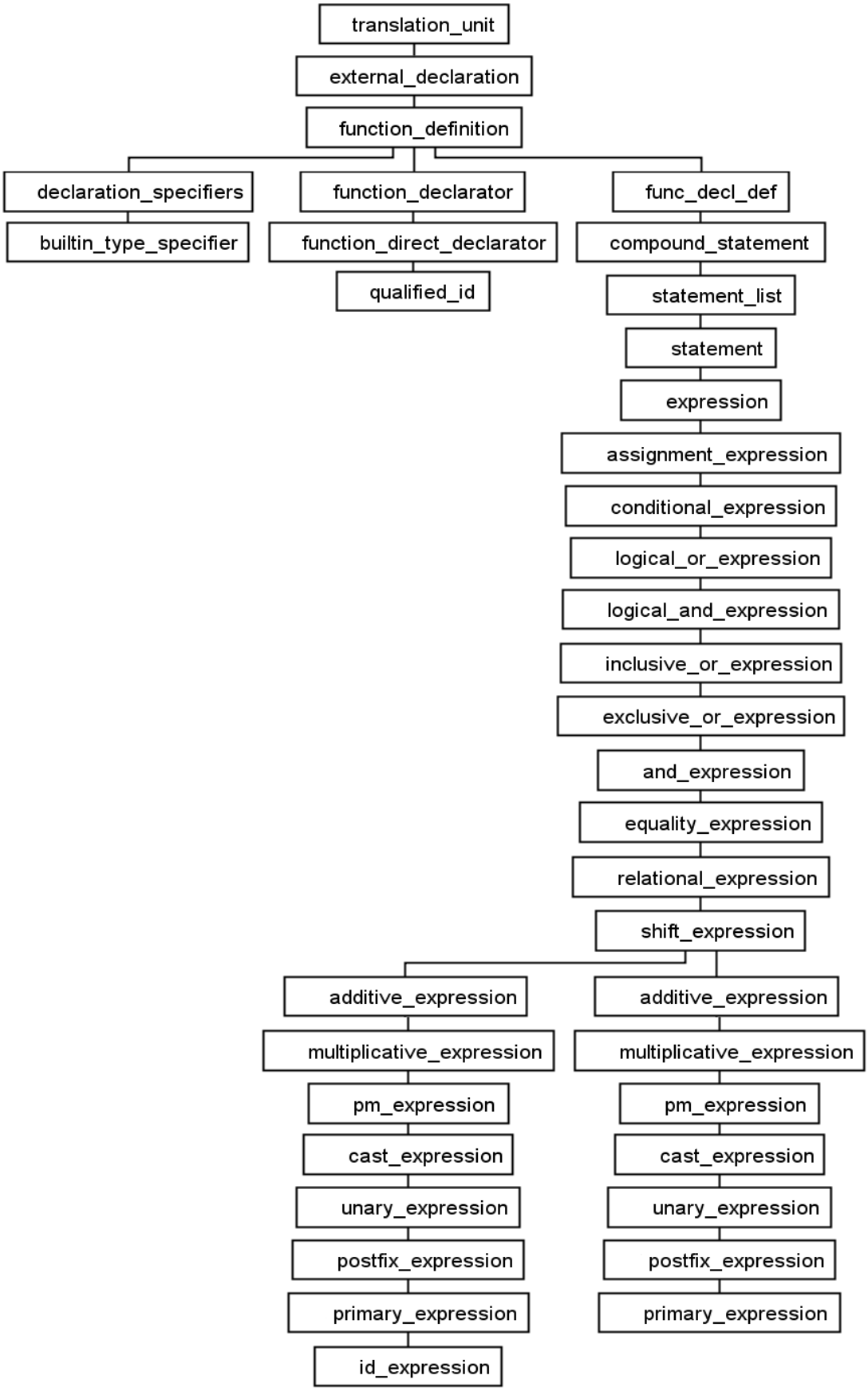
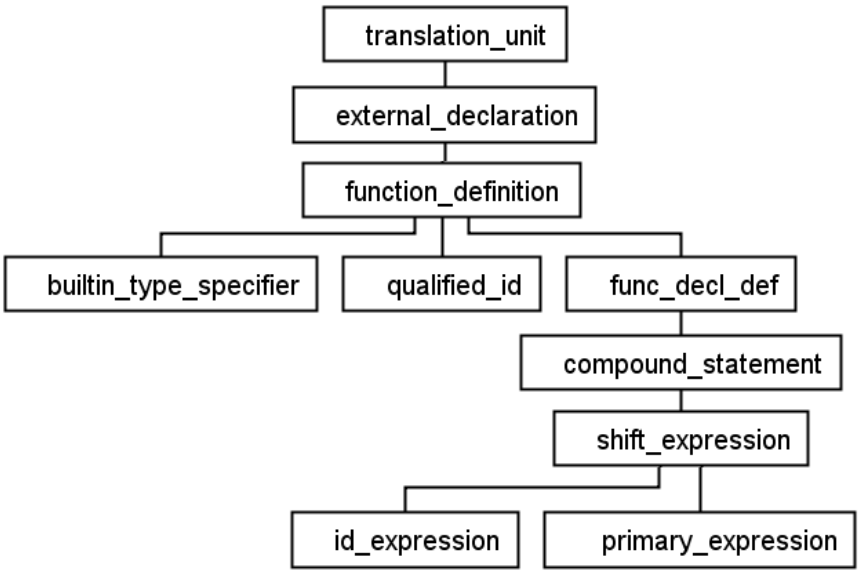

translation_unit
external_declaration
function_definition
declaration_specifiers
builtin_type_specifier
function_declarator
function_direct_declarator
qualified_id
func_decl_def
compound_statement
statement_list
statement
expression
assignment_expression
conditional_expression
logical_or_expression
logical_and_expression
inclusive_or_expression
exclusive_or_expression
and_expression
equality_expression
relational_expression
shift_expression
additive_expression
multiplicative_expression
pm_expression
cast_expression
unary_expression
postfix_expression
primary_expression
id_expression
additive_expression
multiplicative_expression
pm_expression
cast_expression
unary_expression
postfix_expression
primary_expression

Figure 4.13. (a)



(b)

Figure 4.13. Parse Tree and AST for HelloWorld.cpp a) Parse Tree for Helloworld.cpp (Original C++ Grammar) b) AST for Helloworld.cpp (Modified C++ Grammar)

Consider the Java program HelloWorld.java.

```
public class HelloWorld {
public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Figure 4.14 shows the parse tree and AST generated for HelloWorld.java.

The parse tree generated, without any modification in Java grammar, for the simple Java program consisted of 33 levels and 45 nodes (including 7 leaf nodes). Its AST, generated using modified Java grammar, has only 9 levels and 17 nodes.

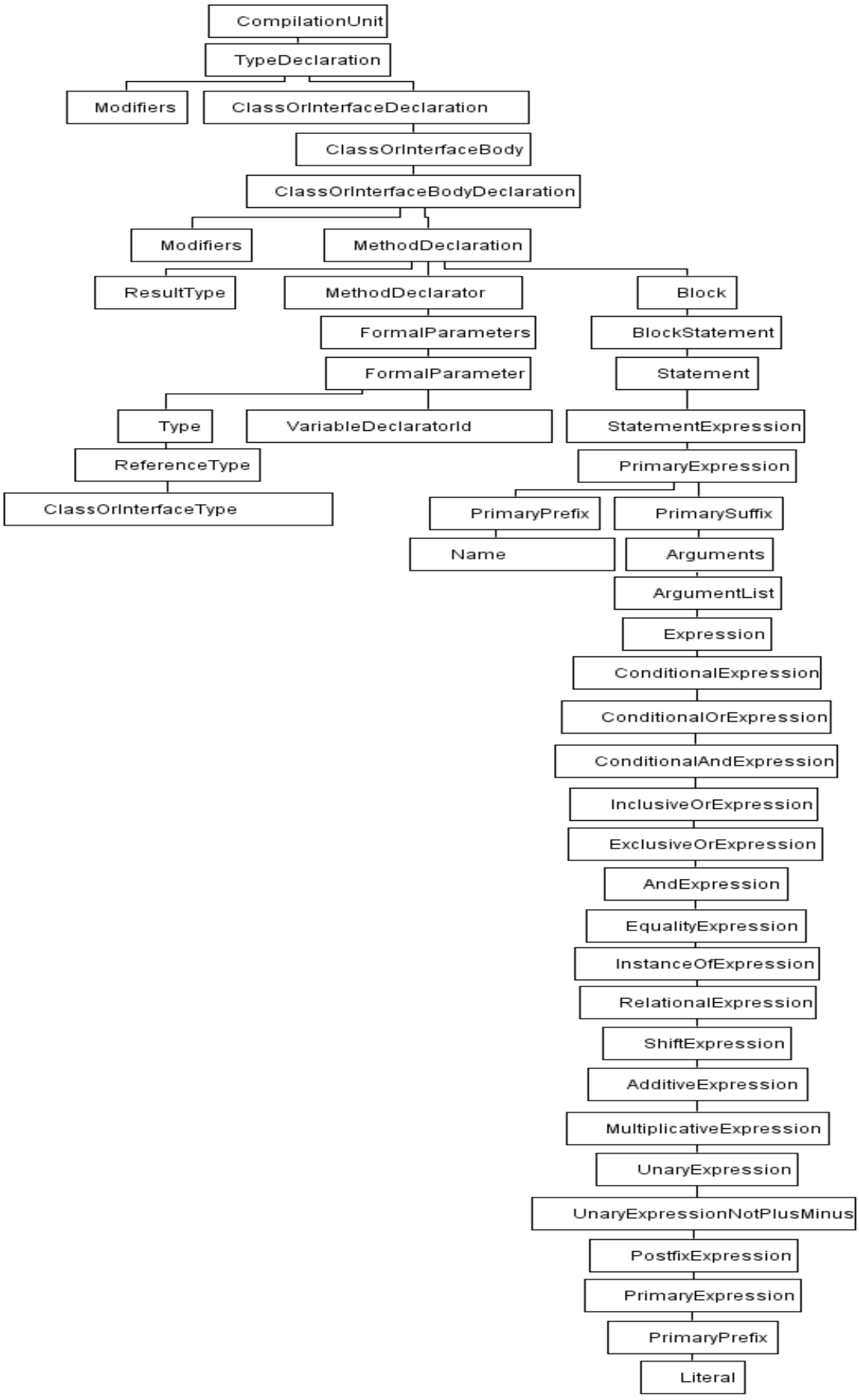
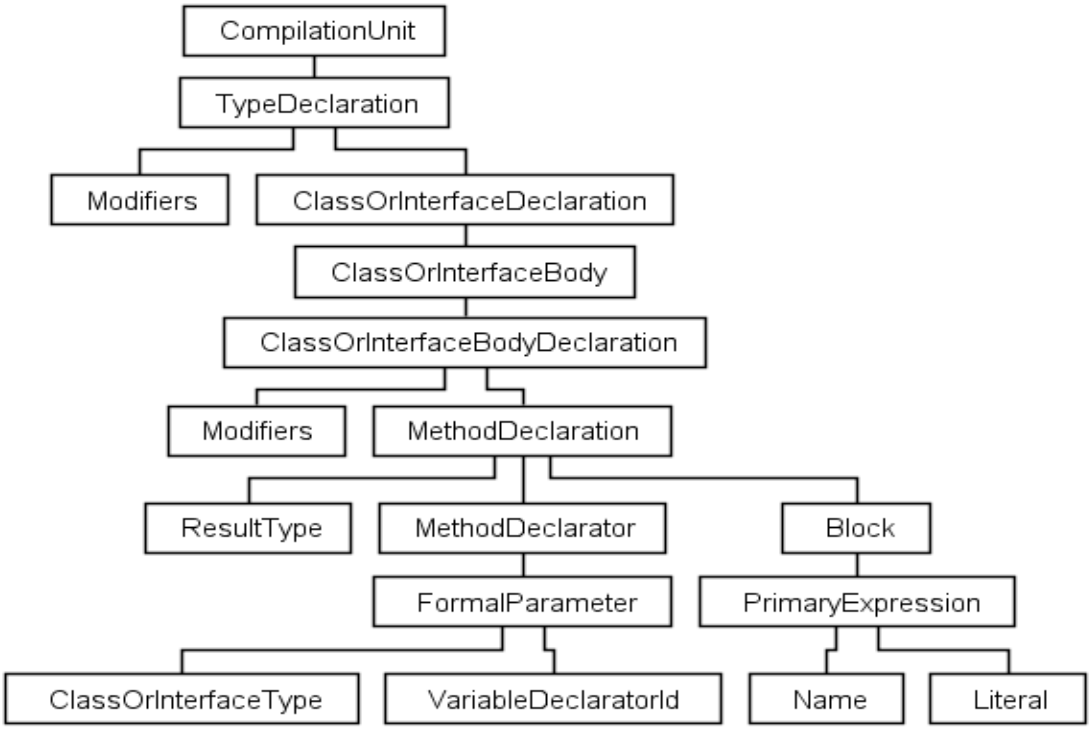

CompilationUnit
TypeDeclaration
Modifiers
ClassOrInterfaceDeclaration
ClassOrInterfaceBody
ClassOrInterfaceBodyDeclaration
Modifiers
MethodDeclaration
ResultType
MethodDeclarator
Block
FormalParameters
BlockStatement
FormalParameter
Statement
Type
VariableDeclaratorId
StatementExpression
ReferenceType
PrimaryExpression
ClassOrInterfaceType
PrimaryPrefix
PrimarySuffix
Name
Arguments
ArgumentList
Expression
ConditionalExpression
ConditionalOrExpression
ConditionalAndExpression
InclusiveOrExpression
ExclusiveOrExpression
AndExpression
EqualityExpression
InstanceOfExpression
RelationalExpression
ShiftExpression
AdditiveExpression
MultiplicativeExpression
UnaryExpression
UnaryExpressionNotPlusMinus
PostfixExpression
PrimaryExpression
PrimaryPrefix
Literal

Figure 4.14 (a)



(b)

Figure 4.14. Parse Tree and AST for Helloworld.java a) Parse Tree for Helloworld.java (Original Java Grammar) b) AST for Helloworld.java (Modified Java Grammar)

Table 4.1 shows the number of nodes and height of the parse trees generated and their corresponding ASTs constructed for the HelloWorld programs written in C, C++ and Java.

Table 4.1. Number of Nodes and Height of Parse Tree and Corresponding AST

ProgrammingLanguage	**Number of nodes, Height of Parse Tree**	**Number of nodes, Height of AST**
C	44, 40	8,5
CPP	38, 25	10, 6
Java	47, 33	17, 8

It can be noted that the size of AST is considerably small compared to the parse tree. Hence, there is a significant reduction in the computational cost involved in AST based comparison in plagiarism detection as compared to parse tree based comparison.

The similarity scores obtained on applying LCS and NW algorithms on ASTs generated and modified using original and modified C, C++ and Java grammars for the common plagiarism strategies listed in section 4.2.2 are given in Tables 4.2, 4.3, and 4.4.

Table 4.2. Similarity Scores Obtained on Applying LCS and NW Algorithms on ASTs Generated and Modified using Original and Modified C Grammar for the Common Plagiarism Strategies

Plagiarism Strategy	C		Modified C	
	NW	**LCS**	**NW**	**LCS**
Changing identifiers	100.0	100.0	100.0	100.0
Changing data types	100.0	100.0	100.0	100.0
Changing the order of operands in expressions	100.0	100.0	100.0	100.0
Changing the order of independent code	100.0	100.0	100.0	100.0
Replacing an expression with an equivalent expression	83.33	83.33	81.82	81.82
Replacing one loop statement with another: a) *for* without block – *for* with block	72.73	72.73	100.0	100.0
b) *while* without block – *while* with block	62.22	62.22	88.37	88.37
c) *for* without block – *while* without block	74.42	74.42	74.42	74.42
d) *for* without block – *while* with block	50.0	50.0	76.19	76.19
e) *for* with block – *while* without block	48.89	48.89	74.42	74.42

f) *for* with block – *while* with block	78.26	78.26	76.19	76.19
g) *do-while* – *for* without block	50.0	50.0	76.19	76.19
h) *do-while* – *for* with block	78.26	78.26	76.19	76.19
i) *do-while* – *while* without block	62.22	62.22	88.37	88.37
j) *do-while* – *while* with block	100.0	100.0	100.0	100.0
Replacing one selection statement with another: a) *if* without block – *if* with block	57.89	57.89	100.0	100.0
b) *if* without block – *switch*	43.24	43.24	81.08	81.08
c) *if* with block – *switch*	63.41	63.41	81.08	81.08
Replacing a statement block with a function call	56.41	56.41	56.0	56.0

Table 4.3. Similarity Scores Obtained on Applying LCS and NW Algorithms on ASTs Generated and Modified using Original and Modified C++ Grammar for the Common Plagiarism Strategies

Plagiarism Strategy	**C++**		**Modified C++**	
	NW	**LCS**	**NW**	**LCS**
Changing identifiers	100.0	100.0	100.0	100.0
Changing data types	100.0	100.0	100.0	100.0
Changing the order of operands in expressions	100.0	100.0	100.0	100.0
Changing the order of independent code	81.48	100.0	80.77	100.0
Replacing an expression with an equivalent expression	83.33	83.33	81.82	81.82
Replacing one loop statement with another: a) *for*	80.0	80.0	100.0	100.0

without block – *for* with block				
b) *while* without block – *while* with block	68.29	68.29	87.18	87.18
c) *for* without block – *while* without block	76.92	76.92	76.92	76.92
d) *for* without block – *while* with block	60.0	60.0	78.95	78.95
e) *for* with block – *while* without block	58.54	58.54	76.92	76.92
f) *for* with block – *while* with block	80.95	80.95	78.95	78.95
g) *do-while* – *for* without block	60.0	60.0	78.95	78.95
h) *do-while* – *for* with block	80.95	80.95	78.95	78.95
i) *do-while* – *while* without block	68.29	68.29	87.18	87.18
j) *do-while* – *while* with block	100.0	100.0	100.0	100.0
Replacing one selection statement with another: a) *if* without block – *if* with block	60.0	60.0	100.0	100.0
b) *if* without block – *switch*	46.15	46.15	82.05	82.05
c) *if* with block – *switch*	65.12	65.12	82.05	82.05
Replacing a statement block with a function call	48.78	48.78	48.10	48.10

Table 4.4. Similarity Scores Obtained on Applying LCS and NW Algorithms on ASTs Generated and Modified using Original and Modified Java Grammar for the Common Plagiarism Strategies

Plagiarism Strategy	**Java**		**Modified Java**	
	NW	**LCS**	**NW**	**LCS**
Changing identifiers	100.0	100.0	100.0	100.0
Changing data types	100.0	100.0	100.0	100.0

Changing the order of operands in expressions	100.0	95.45	100.0	100.0
Changing the order of independent code	90.91	75.76	90.91	100.0
Replacing an expression with an equivalent expression	89.47	89.47	89.47	89.47
Replacing one loop statement with another: a) *for* without block – *for* with block	86.79	86.79	100.0	100.0
b) *while* without block – *while* with block	84.62	84.62	98.11	98.11
c) *for* without block – *while* without block	54.90	54.90	83.02	83.02
d) *for* without block – *while* with block	52.83	52.83	81.48	81.48
e) *for* with block – *while* without block	53.85	53.85	83.02	83.02
f) *for* with block – *while* with block	51.85	66.67	81.48	81.48
g) *do-while* – *for* without block	52.83	52.83	81.48	81.48
h) *do-while*– *for* with block	51.85	66.67	81.48	81.48
i) *do-while* – *while* without block	67.92	83.02	98.11	98.11
j) *do-while* – *while* with block	66.67	85.19	100.0	100.0
Replacing one selection statement with another: a) *if* without block – *if* with block	63.16	52.63	100.0	100.0
b) *if* without block – *switch*	50.0	50.0	85.19	85.19
c) *if* with block – *switch*	54.90	54.90	85.19	85.19
Replacing a statement block with a function call	60.98	60.98	60.98	60.98

These scores are obtained with gap penalty $g = 0$.

4.5.2 System Evaluation and Performance

The AST approach is tested for a student program database with source code files of varying functionality and size. When tested, it is observed that unlike LSI, increase in the number of files (which tends to increase the variation in functionality and file size) does not affect the precision and recall of AST matching. It is found to be very effective in detecting files which are plagiarized. However, running time of AST algorithm increases with increase in number of files.

Figure 4.15 shows the plots of running times of AST and LSI algorithms against number of files under comparison when applied on C source code files in database Sahrdaya (see Appendix B for database statistics).

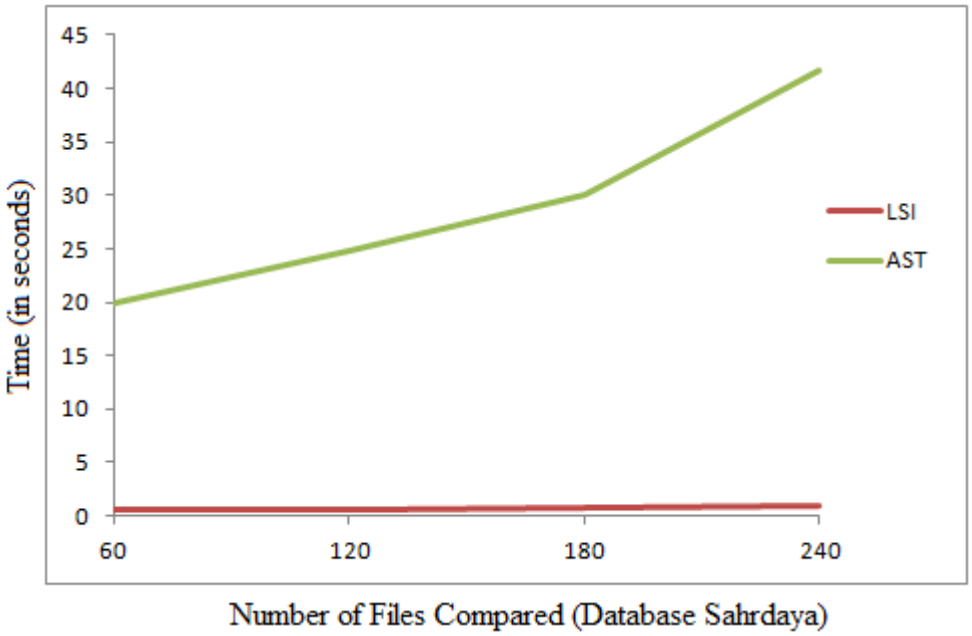


Figure 4.15. Performance Plots of LSI and AST Algorithms

There is a great reduction in the size of the tree obtained after suitable modifications in the grammar. This reduction in size is considerable when compared to the size of parse tree, but otherwise it still poses a problem. Even for the simplest of the programs with less than 5 statements and not more than a few words, the size of ASTs is more as observed in section 4.5.1. That is, as compared to the size of the program, the size of AST is very large. Hence, a system based on AST matching alone requires more runtime and is computationally expensive.

4.6 CONCLUSIONS

The sole aim of this research is to design a strategy to develop a tool which can effectively identify source code files which are highly similar thereby discouraging the students from copying or editing someone else's work. Though computationally expensive, results of AST matching are found to be highly reliable. AST based approach proves to be very efficient in terms of similarity detection, but for a huge program database the runtime is found to be very high. An initial screening of files would help to reduce the number of files given as input to AST-based algorithm and would also reduce the file-file comparisons involved. A two-phase architecture is proposed as a solution. This is discussed in detail in the next chapter.