8 September, 2014

National University of Singapore

CS3202 Software Engineering Project

ITERATION 1 REPORT

Team Number: 4 Consultation Day/Hour: Tuesday, 1pm

Team Name: Team 4

Team Members Information:

Group PKB

Saloni Kaur	A0084053L	a0084053@nus.edu.sg
M I Azima	A0085594N	a0085594@nus.edu.sg

Group PQL

Saima Mahmood	A0084176Y	a0084176@nus.edu.sg
Nguyen Trong Son	A0088441B	A0088441@nus.edu.sg
Vu Phuc Tho	A0090585X	A0090585@nus.edu.sg

1. Iteration Overview

1.1. Scope of SPA Implemented

In this first iteration we have followed the suggested implementation of the SPA according to the assignment document to a major extent. The parser has been implemented for the full SIMPLE as described in the Handbook. The PKB component has been implemented to reflect the Calls, Follows, Parent, Modifies and Uses relationships, via the table and AST representation.

1.2. Achievements & Problems

The main achievement for this iteration would have to be the fact that we revamped the structure of the tables in the PKB. This will be further elaborated on in section 4.

The main problem that we faced was with the Query Processor itself. We had some trouble in correctly implementing the revisions from the previous iteration. This problem will be further discussed in section 8.

2. Project Plans

The way that we have decided to break down the project into tasks is shown in tables 1 and 2 below.

2.1. Plan for Whole Project

The breakdown of tasks reflected in table 1 is tentative and will be subjected to change as we move along the various iterations. The subsequent changes, if any, shall be reflected in the reports of the corresponding iteration.

	Iteration 1				**Iteration 2**			**Iteration 3**		
Team Member	PKB	Parser	Query Processer	Report	PKB	Query Processer	Report	Affects Relationship	Tuple Results	Report
Azima	*			*	*		*			*
Saima			*	*		*	*			*

Saloni	*			*	*		*			*
Sean		*		*		*	*			*
Tho			*			*	*			*

Table 1: Whole Project Tasks Breakdown

2.2. Plan for Iteration 1

Team Member	Testing	Writing Test Cases	Revamp of PKB Tables	Refractoring QP	Working on QP	Extending Parser Funcationality
Azima	*	*	*			
Saima	*				*	
Saloni	*		*			
Sean	*					*
Tho	*			*	*	

Table 2: Iteration 1 Work Distribution

3. UML Diagrams

The UML sequence diagrams presented in this section display how the SPA program flow works between the Parser, PKB and QP. These diagrams allowed us to visualize the various component interaction of the SPA and thus aided in the project planning.

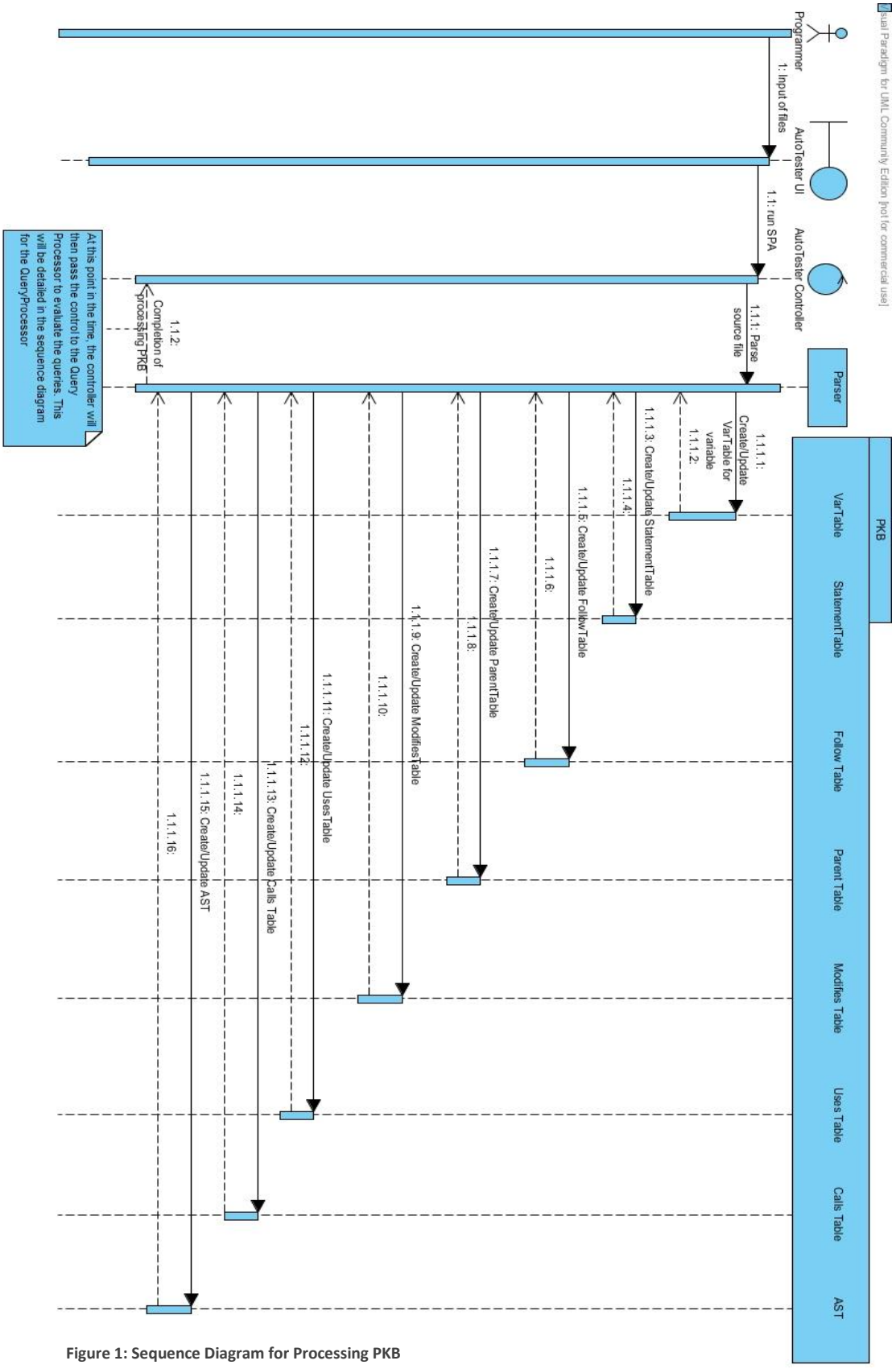

Figure 1: Sequence Diagram for Processing PKB

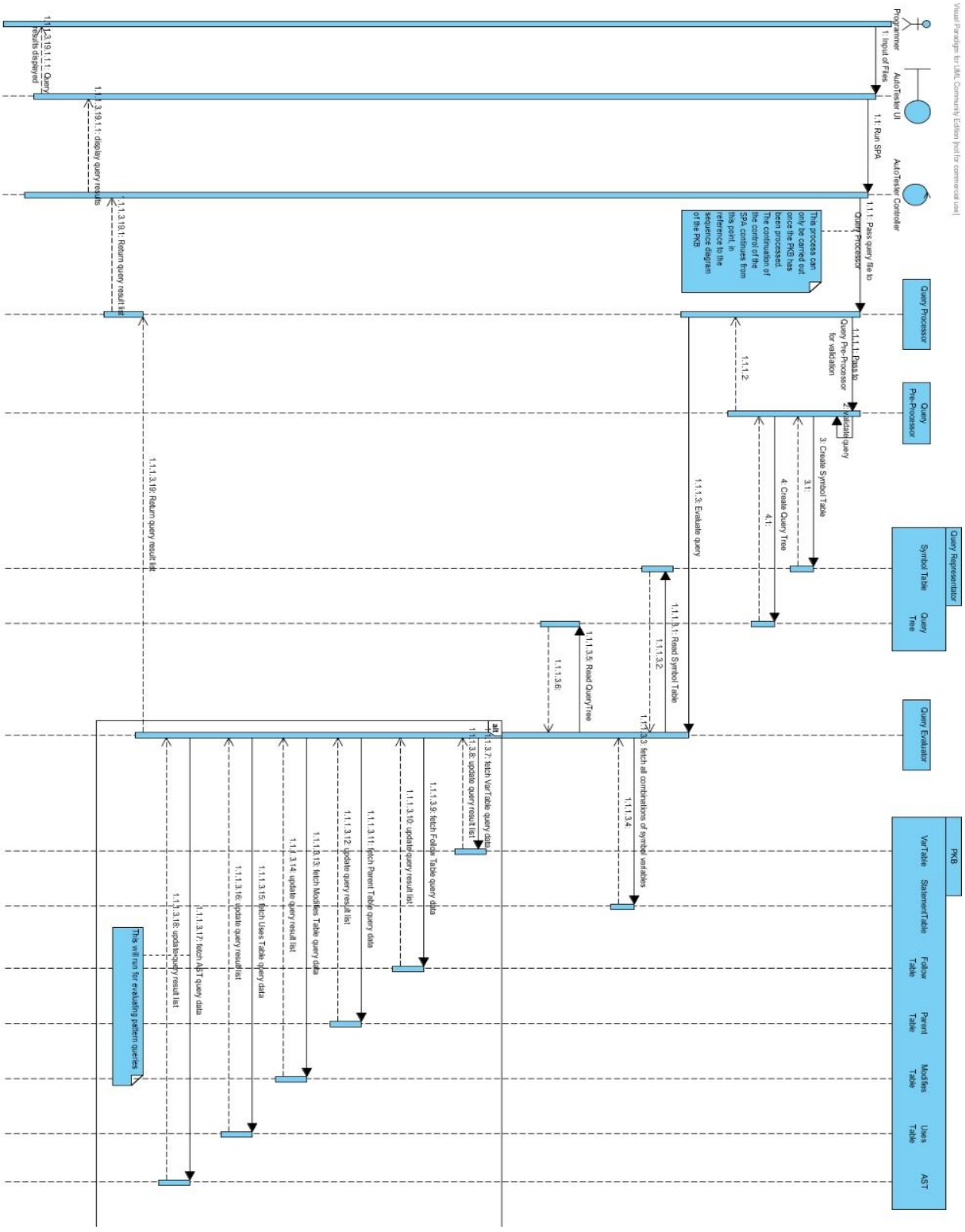

Figure 2: Sequence Diagram for Query Processor Flow

4. Design Decisions

In Iteration 1 our team has revamped the internal structure of the PKB to allow access to the various tables, in the PKB, with a smaller time complexity. Our previous approach was using a vector of integer pairs to store the relationships such as *Follows, Modifies, Uses* and *Parent*. An example of how the Follow table was previously represented is shown in Fig. 1.

0	1	2	3	4	5	6	7	8
1, 2	2, 3	3, 4	4, 5	7, 8	10, 11	13, 14	15, 16	16, 17

Figure 3: Previous Representation of Follow Table

The vector holds elements which are pairs of statement numbers that obey the Follow relationship. For example, in index 3, statement number 4 is followed by statement number 5. This way, methods such as `getFollowingStmt(4)` would return the second element of each element which stores a first element of 4. In this example the time complexity of retrieving the element would been `O(n)`.

In our revised version of the PKB, we have decided to change the internal structure of all the relationship storage to 2D vectors storing Boolean values. This way the time complexity is now `O(1)` to retrieve an element from the table. Fig. 2 shows the representation of the *Follow* table after the change of the data structure.

	4	5	6
4	0	1	0
5	0	0	1
6	0	0	0

Figure 4: Updated Representation of Follow Table

The vertical and horizontal indexes refer to statement numbers. For example, in this case `getFollowingStmt(4)` returns the statement numbers which have a `TRUE (1)` in the horizontal row of index 4. It is statement number 6 in this case.

As a result of the abstract class of PKB, any internal changes made to the PKB do not affect the other components such as the Query Evaluator or the Parser, even though they interact with the PKB. This is a reflection of good separation of concerns and object oriented design of our program.

In addition to the changes made to the PKB, we have updated how we deal with returning vectors. To further improve the system, we have also ensured that we only return a reference to a vector, instead of returning a vector of integers itself, as we did in the previous version of the SPA. As the method receiving the vector of integers does not modify the vector, it is more efficient to return a reference. As such, the speed of the program is also improved.

5. Coding Standards & Experiences

In terms of the coding standards, our group has decided to adopt the following naming conventions described in this section.

5.1. Naming Conventions

5.1.1. General Rules

- Do not use underscore, hyphen or any other non-alphabet characters.
- Any name should has all the first letters of internal words capitalized, e.g. `getProcName()`
- Avoid using abbreviations. Some words are acceptable in short forms, including: *Var*, *Proc*, *Stmt*, *AST*. Other words such as *Children*, *Number* should be fully spelled out.

5.1.2. Specific Rules

- API Name:

  - API names should be nouns, in mixed case with the first letter of each internal word capitalized.

- Method:

  - Method names should be in the form of a verb. With method names containing more than one word, use mixed case with the first letter of each internal word capitalized.

  - Name of some specific methods:
    i. Methods to insert new records to the database should have the form `insertXXX()`.
    ii. Methods with return value type BOOLEAN should have the form: `isXXX()` e.g. `isExist()`, `isMatchVar()`.
    iii. Methods with return types of other values should have the form `getXXX()` e.g. `getVarName()`

iv. Methods that return the number of records inside a table/ list should have the form `getSize()`.
v. Methods that change the values or status of an object should have the form: `setXXX()`
vi. Methods that return values from star queries, such as Calls* and Next*, should have the form `getXXXStar()`.

In all of the above examples, the "XXX" is used in place of the specific name that the method will adopt.

To keep the abstract and concrete PKB API in sync, we created a variable table, statement table and procedure tables. These tables are vectors mapping variable names to indexes of the vectors. So that a API method like `BOOLEAN isModifies(STMT_NUM s1, INDEX varIndex)` understands that `INDEX` is the mapped value of a certain variable name, where `INDEX` is just an integer in C++ type.

6. Query Processing

6.1. Query Validation

The query validating process is handled by the QueryPreprocessor (QPP). The QPP will read and validate data from the query file, and store all necessary information for query evaluation in the Query Representator (QR).

During validation, the QPP checks each query line by line, following these rules:

- On finding a line containing a declaration (e.g. stmt s;), QPP will send this part to the `checkDeclaration()` method.

- On finding a line containing a query part (e.g. Select s such that Follows(s, 1)), QPP will send this part to `checkQueryPart()` method.

- While the `checkQueryPart()` method runs, if the QPP finds a new clause of query, it will call the corresponding method `checkClause()` (e.g. `checkSuchThatCondtion()`) for this clause. Currently, we have provided the methods for "*such that*" clause and "*pattern*" clause.

After validating the query, if no error is found, the QPP will save validated data in the QR.

6.2. Query Evaluation

The query evaluation process is carried out by the QE. Using the stored data from the QR and PKB, the QE will try to find values that satisfy the query's conditions, and send the list of result values to the AutoTester.

During the evaluation of a query, firstly, the QE will create a list of values for all parameters declared in that query. After that, it will check all of the query's conditions one by one, attempting to update the list with values of parameters that satisfy the conditions. If it meets any condition that no value can satisfy, the QE will hang up the process and return an empty list as the result of the query.

After the process finishes, with all conditions being satisfied, the QE will take the result value and save it to the result list. This is the list that will be output to the user.

7. Testing

7.1. Test Plan

In this iteration we continuously tested each and every component after changes were made. We made the decision that unit testing on each component would be done by the person implementing the component. Integration testing amongst components would be carried out by the people responsible for the specific components. Finally validation testing will be carried out by everyone at various points of progress during development.

In some aspects of the testing we have used assertions. Examples of where they were used will be shown in the figures shown below in the following sub-sections.

7.2. Examples of Test Cases

7.2.1. Unit Testing

7.2.1.1. PKB

Sample 1

Test Purpose: To test the Parent Table

Required Test Inputs: Parent table and CPP Unit test case for Parent Table. An example is shown in figure 5 below.

```
void ParentTableTest::testInsertParent()
{
    ParentTable table;
    table.insertParent(2, 3);

    int parentStmt = table.getParentStmt(3);
    CPPUNIT_ASSERT(2 == parentStmt);

    std::vector<int> children = table.getChildStmt(2);
    std::vector<int> expectedChildren (1, 3); // initialize new vector with one element, element's value = 2.
    CPPUNIT_ASSERT(expectedChildren == children);

    // CPPUNIT_ASSERT(true == table.isParent(2, 3));

    int returnedIndex = table.insertParent(2, 3);
    CPPUNIT_ASSERT(-1 == returnedIndex);

    return;
}

void ParentTableTest::testIsParent()
{
    ParentTable table;
    table.insertParent(2, 3);
    table.insertParent(2, 4);

    CPPUNIT_ASSERT(true == table.isParent(2, 3));
    CPPUNIT_ASSERT(true == table.isParent(2, 4));
    CPPUNIT_ASSERT(false == table.isParent(3, 4));
    return;
}
```

Figure 5: Parent Table Unit Test

Expected Test Results: “OK” This uses the CPP Unit assert, as seen in figure 5 above.

Other Requirements: None

Sample 2

Test Purpose: Uses Table

Required Test Inputs: Use Table and and CPP Unit test case for Uses Table. An example is shown in figure 6 below.

```
void UseTest::testisUse(){

    Use useObj;

    //insert a few pairs first
    useObj.insertUses(1, 1);
    useObj.insertUses(2, 2);


    // verify that the pair exists - Note 7
    CPPUNIT_ASSERT_EQUAL(true, useObj.isUses(1, 1));
    CPPUNIT_ASSERT_EQUAL(true, useObj.isUses(2, 2));

    // attempt to check a pair which does not exists
    CPPUNIT_ASSERT_EQUAL(false, useObj.isUses(5, 3));

    return;
}

void UseTest::testgetUsedVarAtStmt(){

    Use useObj;

    //insert a few pairs first
    int result = useObj.insertUses(1, 1);
    int result1 = useObj.insertUses(2, 2);
    int result2 = useObj.insertUses(2, 3);

    std::vector<int> actual = useObj.getUsedVarAtStmt(2);

    std::cout<<"getUsedVarAtStmt";

    for(std::size_t i = 0; i < actual.size(); i++){
        std::cout << actual[i] << std::endl;
    }

    return;
}
```

Figure 6: Uses Table Unit Test

Expected Test Results: “OK” This uses the CPP Unit assert, as seen in figure 6 above.

Other Requirements: None

7.2.1.2. Parser

Test Purpose: To test the Parser component of the SPA

Required Test Inputs: Parser and CPP Unit test case for Parser. An example is shown in figure 7 below.

```
1  #include <cppunit/config/SourcePrefix.h>
2  #include "Parser.h"
3  #include "ParserTest.h"
4
5  #include <iostream>
6  #include <string>
7
8  using namespace std;
9
10 void ParserTest::setUp() {
11 }
12
13 void ParserTest::tearDown() {
14 }
15
16 CPPUNIT_TEST_SUITE_REGISTRATION( ParserTest ); // Note 4
17
18 void ParserTest::testPrintTable() {
19     Parser parser;
20     parser.parse("D:\\sample_SIMPLE_source.txt");
21
22     parser.printVar();
23     parser.printFollows();
24     parser.printParent();
25
26
27 }
28
29 void ParserTest::testPrintAST() {
30 }
```

Figure 7: Excerpt f Parser Test File

Expected Test Results: As seen in figure 8. This uses the CPP Unit assert, as seen in figure 7 above.

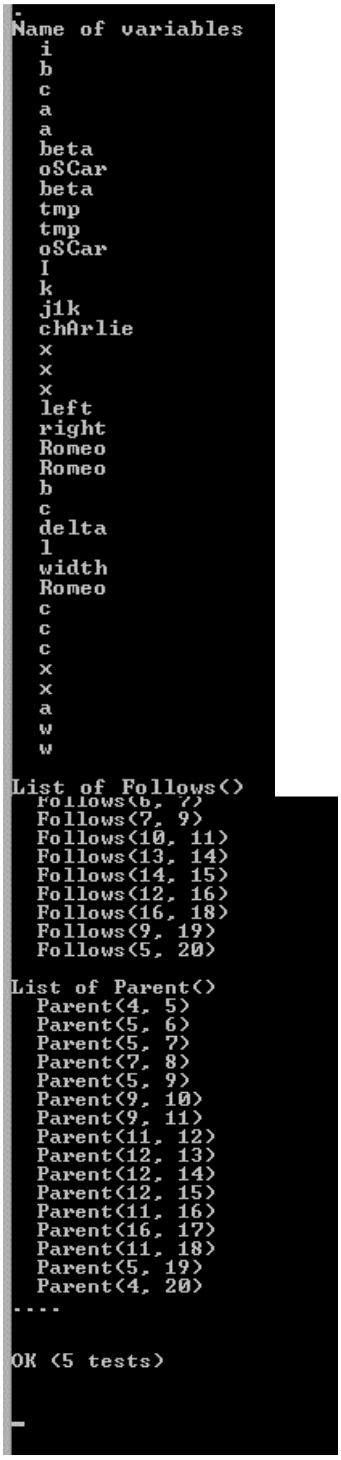

Figure 8: Test Results for Unit Testing

Other Requirements: None

7.2.2. Integration Testing

7.2.2.1. Parser & PKB

Figures 5- 7 show the implementation of the integration testing done using CPP Unit.

Test Purpose: To test the correctness of the interaction between the *Parser* and *PKB.*

Required Test Inputs: *Parser* and *PKB*.

Expected Test Results: Passed all

Other Requirements: None

```
117     // Test VarTable
118     std::cout << "VAR TABLE: " << std::endl;
119     bool isVar1 = (pkb.getVarIndex("tmp") == 6);
120     bool isVar2 = (pkb.getVarName(12) == "left");
121     std::cout << "Test isVar1: " << isVar1 << std::endl;
122     //std::cout << pkb.getVarName(12) << std::endl;
123     std::cout << "Test isVar2: " << isVar2 << std::endl;
124     std::cout << " " << std::endl;
125
126     // Raw test for Follow Table.
127     std::cout << "FOLLOW TABLE: " << std::endl;
128
129     bool isFollows1 = (pkb.isFollows(1, 20) == false);
130     bool isFollows2 = (pkb.isFollows(1, 2) == true);
131     std::cout << "Test isFollow1: " << isFollows1 << std::endl;
132     std::cout << "Test isFollow2: " << isFollows2 << std::endl;
133
134     bool isFollows3 = (pkb.getFollowingStmt(7) == 9);
135     bool isFollows4 = (pkb.getFollowedStmt(11) == 10);
136     std::cout << "Test isFollow3: " << isFollows3 << std::endl;
137     //std::cout << "Test isFollow4: " << isFollows4 << std::endl;
138
139     vector<int> allFollowing1;
140     vector<int> allFollowing2;
141
142     allFollowing1.push_back(2); allFollowing1.push_back(3); allFollowing1.push_back(4);
143     allFollowing2.push_back(14); allFollowing2.push_back(15);
144
145     bool isFollows5 = (pkb.getFollowingStarStmt(1) == allFollowing1);
146     bool isFollows6 = (pkb.getFollowingStarStmt(13) == allFollowing2);
147     std::cout << "Test isFollow5: " << isFollows5 << std::endl;
148     std::cout << "Test isFollow6: " << isFollows6 << std::endl;
149     vector<int> temp = pkb.getFollowingStarStmt(13);
150     for (size_t index=0; index < temp.size(); index++)
151     {
152         std::cout << "Stmt that following stmt13: " << temp[index] << std::endl;
153     }
154
155     vector<int> allFollowing3;
156     vector<int> allFollowed4;
157
158     allFollowing3.push_back(2); allFollowing3.push_back(3); allFollowing3.push_back(4);
159     allFollowing3.push_back(7); allFollowing3.push_back(9); allFollowing3.push_back(11);
```

Figure 9: CPP Unit Integration Testing for Parser & PKB

```
28  void ParserTest::testBuildVarTable()
29  {
30      Parser parser;
31      parser.parse(testFile);
32      parser.buildVarTable();
33
34      PKB pkb;
35      CPPUNIT_ASSERT("i" == pkb.getVarName(0));
36      //for(size_t i = 0; i < pkb.getVarTableSize(); i++)
37      //{
38      //  std::cout << "At index: " << pkb.getVarName(i);
39      //}
40
41      return;
42  }
43
44  void ParserTest::testBuildModifyTable()
45  {
46      Parser parser;
47      parser.parse(testFile);
48      parser.buildVarTable();
49      parser.buildModifyTable();
50
51      PKB pkb;
52      // Test isModifies
53      bool modify1 = (pkb.isModifies(1, 0) == true);
54      bool modify2 = (pkb.isModifies(1, 11) == false);
55      cout << "Test modify1: " << modify1 << endl;
56      cout << "Test modify2: " << modify2 << endl;
57
58      // Test getModifiedVarAtStmt
59      vector<int> expectedModifiedVar;
60      expectedModifiedVar.push_back(5); expectedModifiedVar.push_back(11); expectedModifiedVar.push_
61      expectedModifiedVar.push_back(1); expectedModifiedVar.push_back(2); expectedModifiedVar.push_b
62      expectedModifiedVar.push_back(18);
63
64      vector<int> actualModifiedVar;
65      actualModifiedVar = pkb.getModifiedVarAtStmt(4);
66
67      sort(expectedModifiedVar.begin(), expectedModifiedVar.end());
68      sort(actualModifiedVar.begin(), actualModifiedVar.end());
69      bool modify3 = (expectedModifiedVar == actualModifiedVar);
70      cout << "Test modify3: " << modify3 << endl;
71
72      // Test getStmtModifyingVar
```

Figure 10: CPP Unit Integration Testing for Parser & PKB

7.2.2.2. PKB & QP

Test Purpose: To test the correctness of the interaction between the *PKB* and *QP.*

Required Test Inputs: PKB and QP

Expected Test Results: Passed all.

Other Requirements: None

```
69  void QueryProcessorTest::insertSampleData() {
70      // insert some variable names to VarTable
71      PKB::insertVar("x");
72      PKB::insertVar("y");
73      PKB::insertVar("z");
74      PKB::insertVar("t");
75      PKB::insertVar("i");
76
77      // insert some stmt type to StatTable
78      PKB::insertStmt("while");
79      PKB::insertStmt("while");
80      PKB::insertStmt("assign");
81      PKB::insertStmt("while");
82      PKB::insertStmt("assign");
83
84      // insert some relationship btw stmt(s)
85      PKB::insertFollows(3, 4);
86      PKB::insertParent(1, 2);
87      PKB::insertParent(2, 3);
88      PKB::insertParent(4, 5);
89      PKB::insertParent(2, 5);
90      PKB::insertParent(1, 5);
91
92      // insert some relationship btw stmt and variable
93      PKB::insertModifies(3, 0);
94      PKB::insertUses(3, 1);
95      PKB::insertModifies(5, 2);
96      PKB::insertUses(5, 3);
97      PKB::insertUses(5, 4);
98  }
```

Figure 11: CPP Unit Integration Testing for PKB & QP

7.2.3. Validation Testing

Sample 1

Test Purpose: To test the program's ability to parse multiple nested while loops and a variety of variable names.

Required Test Inputs: The whole system. Source file and query file.

```
procedure proc234 {
        rose = 3 + r;                                                   \\1
        while rose                                                      \\2
        {
                o = 6;                                                  \\3
                s = o + e;                                              \\4
                while s {                                               \\5
                        e = 2 + o + s + e;                              \\6
                }
                e = 8;                                                  \\7
                lily = r + e + rose;                                    \\8
                while r {                                               \\9
                        while o {                                       \\10
                                while s {                               \\11
                                        tulip = rose + lily + 4;        \\12
                                }
                        }
                        flower = tulip + lily;                          \\13
                        flower = flower + rose;                         \\14
                }
                flower = 3;                                             \\15
                r = flower + 123;                                       \\16
                o = r + o + s + e;                                      \\17
        }
        x = 3;                                                          \\18
        carnation = yellow + lily;                                      \\19
        yellow = 8 + carnation + 6 + yellow;                            \\20
        while o {                                                       \\21
                l = l + i + l + y;                                      \\22
                lavender = lily + 8 + a + e;                            \\23
                e = lavender + 2 + l;                                   \\24
        }
        r = r + e + 2 + e + d;                                          \\25
        while e {                                                       \\26
                w = 2;                                                  \\27
        }

}
```

Figure 12: Sample Source File

Expected Test Results: As shown in figure 9 below.

Queries and Expected Output
35
cat: All Category
1 - Follows, ImplStmtLine ::
stmt s1;
Select s1 such that Follows(s1, 4)
3
5000
2 - Follows, ImplStmtLine ::
stmt s2;
Select s2 such that Follows(7, s2)
8
5000
3 - Follows, ImplStmtLine ::
stmt s;
Select s such that Follows(12, 13)
5000
4 - Follows Star, ImplStmtLine ::
stmt s2;
Select s2 such that Follows*(3, s2)
4, 5, 7, 8, 9, 15, 16, 17
5000
5 - Follows Star, ImplStmtLine ::
stmt s1;
Select s1 such that Follows*(s1, 25)
1, 2, 18, 19, 20, 21
5000
6 - Follows Star, ImplStmtLine ::
constant c;
Select c such that Follows*(19, 21)
3, 6, 2, 4, 123, 8
5000
7 - Parent, ImplStmtLine ::
while w;
Select w such that Parent(w, 19)
5000
8 - Parent, ImplStmtLine ::
stmt s;
Select s such that Parent(21, s)
22, 23, 24
5000
9 - Parent, ImplStmtLine ::
stmt s;
Select s such that Parent(25, s)
5000
10 - Parent Star, ImplStmtLine ::
while w;
Select w such that Parent*(w, 12)

11, 10, 9, 2
5000
11 - Parent Star, ImplStmtLine ::
stmt s;
Select s such that Parent*(2, s)
3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17
5000
12 - Parent Star, ImplStmtLine ::
while w;
Select w such that Parent*(5, 6)
2, 5, 9, 10, 11, 21, 26
5000
13 - Modifies, ImplStmtLine ::
assign a;
Select a such that Modifies(a, "flower")
13, 14, 15
5000
14 - Modifies, ImplStmtLine ::
variable var;
Select var such that Modifies(24, var)
e
5000
15 - Modifies, ImplStmtLine ::
stmt s;
Select s such that Modifies(s, "e")
2, 5, 6, 7, 21, 24
5000
16 - Uses, ImplStmtLine ::
variable v;
Select v such that Uses(17, v)
r, o, s, e
5000
17 - Uses, ImplStmtLine ::
stmt s;
Select s such that Uses(s, "e")
2, 4, 5, 6, 8, 17, 21, 23, 25, 26
5000
18 - Uses, ImplStmtLine ::
assign a;
Select a such that Uses(16, "flower")
1, 3, 4, 6, 7, 8, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 27
5000
19 - Parent, ImplStmtLine ::
while w1; while w2;
Select w1 such that Parent(w1, w2)
2, 9, 10
5000
20 - Parent, ImplStmtLine ::

```
stmt s;
Select s such that Parent(s, 1)
5000
21 - Parent Star, ImplStmtLine ::
while w;
Select w such that Parent*(9, w)
10, 11
5000
22 - Pattern, Typed ::
assign a; variable v;
Select a pattern a(v, _"o+s"_)
5000
23 - Pattern, Typed ::
assign a;
Select a pattern a(_, _"r + e"_)
8, 25
5000
24 - Pattern, Typed ::
assign a;
Select a pattern a(_, _" yellow + lily"_)
19
5000
25 - Pattern, Modifies ::
assign a;
Select a such that Modifies(a, "flower") pattern a(_, _" tulip + lily"_)
13
5000
26 - Pattern, Uses ::
assign a;
Select a such that Uses(a, "a") pattern a(_, _"lily + 8"_)
23
5000
27 - Modifies, Pattern ::
assign a; while w; variable v;
Select a such that Modifies(w, v) pattern a(v, _"r + e"_)
8, 25
5000
28 - Typed, Pattern, Uses ::
assign a; variable v;
Select v pattern a(v, _) such that Uses(a, v)
e, flower, o, yellow, l, r
5000
29 - Typed, Pattern, Modifies ::
assign a; while w; variable v;
Select w pattern a(v, _"rose + lily"_) such that Modifies(w, v)
2, 9, 10, 11
```

```
5000
30 - Typed, Pattern, Modifies ::
assign a; variable v;
Select a pattern a(v, _) such that Uses(a, "lily")
12, 13, 19, 23
5000
31 - Typed, Pattern, Parent ::
assign a; while w;
Select a pattern a(_, _"flower"_) such that Parent(w, a)
14, 16
5000
32 - Typed, Pattern, Follow ::
assign a; while w;
Select a pattern a(_, _"8 + carnation"_) such that Follows(a, w)
20
5000
33 - Typed, Pattern, FollowStar ::
assign a;
Select a pattern a(_, _"e"_) such that Follows*(3, a)
4, 8, 17
5000
34 - Typed, Pattern, ParentStar ::
assign a;
Select a pattern a(_, _"rose+lily"_) such that Parent*(2, a)
12
5000
```

Figure 13: Expected Test Results

Other Requirements: None

Sample 2

Test Purpose: To test the program's ability to parse multiple nested while loops, irregular spacing between statements and variables and factors. It also includes a large number of variable names.

Required Test Inputs: Whole system. Source file and query file.

```
procedure spacersoul{
i = 5; \\1
while i{ \\2
j = j + k; \\3
while h{ \\4
while s \\5
{
f = 34; \\6
x = x + h + t + m + l; \\7
}
while f{ \\8
m = 3; \\9
while da{ \\10
plain = boring; \\11
while boring{ \12
a = a + 42; \\13
while q \\14
{
q = q + m + h + k+ 7; \\15
while e{ \\16
summer = summer + holiday + internship + sg; \\17
while internship{ \\18
buggy = small; \\19
buggy = big; \\20
while big \\21
{
headache = headache + 1000 + a + b + c + d + e + f + g + h;
\\22
meow = meow + meow + pat + pat +32; \\23
while term \\24
{
41
busy = busy + hah + deadline + 5; \\25
}
}
}
}
}
}
}
}
a = 2; \\26
}
}
}
```

Figure 14: Sample Source File

Expected Test Results: As shown in in figure 11 below.

Queries and Expected Output
18
cat: All Category
1 - Follows, ImplStmtLine ::
stmt s1;
Select s1 such that Follows(s1, 12)
11
5000
2 - Follows, ImplStmtLine ::
stmt s2;
Select s2 such that Follows(11, s2)
12
5000
3 - Follows, ImplStmtLine ::
stmt s;
Select s such that Follows(12, 13)
5000
4 - Follows Star, ImplStmtLine ::
stmt s2;
Select s2 such that Follows*(5, s2)
8, 26

5000

5 - Follows Star, ImplStmtLine ::
stmt s1;
Select s1 such that Follows*(s1, 24)
22, 23
5000
6 - Follows Star, ImplStmtLine ::
constant c;
Select c such that Follow*(19, 21)
2, 3, 5, 7, 32, 34, 42, 1000
5000
7 - Parent, ImplStmtLine ::
while w;
Select w such that Parent(w, 17)
16
5000
8 - Parent, ImplStmtLine ::
stmt s;
Select s such that Parent(18, s)
19, 20, 21
5000
9 - Parent, ImplStmtLine ::
stmt s;
Select such that Parent(15, s)
5000
10 - Parent Star, ImplStmtLine ::
while w;
Select w such that Parent*(w, 24)

```
2, 4, 8, 10, 12, 14, 16, 18, 21
5000
11 - Parent Star, ImplStmtLine ::
stmt s;
Select s such that Parent*(16, s)
17, 18, 19, 20, 21, 22, 23, 24, 25
5000
12 - Parent Star, ImplStmtLine ::
while w;
Select w such that Parent*(16, 17)
2, 4, 8, 10, 12, 14, 16, 18, 21, 24
5000
13 - Modifies, ImplStmtLine ::
assign a;
Select a such that Modifies(a, "buggy")
19, 20
5000
14 - Modifies, ImplStmtLine ::
variable var;
Select var such that Modifies(24, var)

5000

15 - Modifies, ImplStmtLine ::
stmt s;
Select s such that Modifies(s, "buggy")
2, 4, 8, 10, 12, 14, 16, 18, 19, 20
5000
16 - Uses, ImplStmtLine ::
variable v;
Select v such that Uses(22, v)
headache, a, b, c, d, e, f, g, h
5000
17 - Uses, ImplStmtLine ::
stmt s;
Select s such that Uses(s, "a")
2, 4, 8, 10, 12, 13, 14, 16, 18, 21, 22
5000
18 - Uses, ImplStmtLine ::
assign a;
Select a such that Uses(15, "m")
1, 3, 6, 7, 9, 11, 13, 15, 17, 19, 20, 22, 23, 25, 26

5000
```

Figure 15: Expected Test Results

Other Requirements: None

8. Discussion

AS briefly mentioned in the beginning of the report, we faced some debugging challenges in the QP, more specifically the QE. As a continuation from the previous revision iteration, we were getting failed test cases during our system testing. Once we fixed this, we got a feedback during our consultation that it would be best to refractor the code of the QE to make it more user friendly such that extending it would be smoother. As such we refractored the code and this affected the project schedule for the QP. Due to this we were unable to implement the *with* clause, *Modifies* and *Uses* for procedure calls and extend the *pattern* functionality to accommodate more complex expressions.

As this was the beginning iteration for this semester, we took some time setting up a new repository and integrating Autotester with our SPA once again.

We also gained a new team member this time around, while an old member left, so we had reorganize ourselves and think of redistributing some tasks.

Looking ahead for iteration 2, we definitely have to focus on getting up to speed with extending QP and improving on it.

Appendix A: Abstract PKB API

8.1. VarTable API

VarTable
Overview: VarTable is to keep all the variables appearing in the program
Public Interface:
INDEX ***insertVar*** (STRING VarName) **Description:** If “varName” is not in the VarTable, insert it into the VarTable and return its index value. Otherwise, return -1 (special value) and the table remains unchanged.
STRING ***getVarName*** (INDEX ind)

Description: If there is record in VarTable having index value “ind”, return its variable name. If “ind” is out of range: Throws: InvalidReferenceException
INDEX ***getVarIndex*** (STRING varName) **Description:** If there is record in VarTable having name “varName”, return its index value. Otherwise, return -1 (special value)

8.2. FollowTable API

Follow
Overview: Follow is to keep the relationship Follows of any pair of statements appearing in the program into a table.
Public Interface:
BOOLEAN ***isFollows*** (STMT_NUM s1, STMT_NUM s2) **Description:** If the relation Follows(s1, s2) is recorded in Follow Table, return true. Otherwise return false.
INDEX ***insertFollows*** (STMT_NUM s1, STMT_NUM s2) **Description:**

If the relation Follows(s1, s2) is not in Follow Table, insert it into the table and return its index value. Otherwise: return -1 (special value) and the table remains unchanged.
LIST<STMT_NUM> ***getFollowingStmt*** (STMT_NUM s1) **Description:** If s1 > 0, return an array of all statement numbers recorded in table that follow statement “s1” (Follows(s1, s)). Otherwise, return NULL
LIST<STMT_NUM> ***getFollowedStmt*** (STMT_NUM s1) **Description:** If s1 > 0, return an array of all statement numbers recorded in table that are followed by statement “s1” (Follows(s, s1)). Otherwise, return NULL
LIST<STMT_NUM> ***getFollowedStarStmt*** (STMT_NUM s1) **Description:** If s1 > 0, return an array of all statement numbers “s” recorded in table that Follows*(s, s1) exists. Otherwise, return NULL

8.3. ParentTable API

Parent
Overview: ParentTable is to keep the relationship Parent of any pair of statements appearing in the program into a table.
Public Interface:

BOOLEAN ***isParent*** (STMT_NUM s1, STMT_NUM s2)

Description:

If the relation Parent(s1, s2) is recorded in Parent Table, return true.

Otherwise return false.

INDEX ***insertParent*** (STMT_NUM s1, STMT_NUM s2)

Description:

If the relation Parent(s1, s2) is not in Parent Table, insert it into the table and return its index value.

Otherwise: return -1 (special value) and the table remains unchanged.

STMT_NUM ***getParentStmt*** (STMT_NUM s1)

Description:

If s1 > 0, return statement number “s” recorded in table that is direct parent of statement “s1” (Parent(s, s1)).

Otherwise, return NULL

LIST<STMT_NUM> ***getChildStmt*** (STMT_NUM s1)

Description:

If s1 > 0, return all statement numbers recorded in table that are direct children of statement “s1” (Parent(s1, s)).

Otherwise, return NULL

LIST<STMT_NUM> ***getParentStarStmt*** (STMT_NUM s1)

Description:

If s1 > 0, return all statement numbers recorded in table that are parent (direct or indirect) of statement “s1” (Parent*(s, s1))

Otherwise, return NULL

LIST<STMT_NUM> ***getChildStarStmt*** (STMT_NUM s1)

Description:

If s1 > 0, return all statement numbers recorded in table that are direct or indirect children of statement “s1” (Parent*(s1, s)).

Otherwise, return NULL

8.4. Modify API

Modify

Overview:

a. Modify for assignment statements is to keep the relationship Modifies(a, x) of statement a and variable x appearing in the program into a table. The table keeps Modifies(a, x) by recording the statement number “a” and index value of variable “x” in the VarTable.
b. Modify for statements is to keep the relationship Modifies(“if”, x) or Modifies(“while”, x) of containers “if” or “while” and variable x appearing in the program into a table. The modifies table keeps Modifies relationship by recording the container statements number a and index value of variable “x” in the VarTable. We can check if a container includes a statement by checking the Parent* relationship of that statement number. This table for Modifies is the same table used in point a).
c. Modify for procedures just checks if the the statement is contained in the procedure by checking against the AST and then using the Modify table.This table for Modifies is the same table used in point a).

Public Interface:

BOOLEAN ***isModifies*** (STMT_NUM s, INDEX varIndexOfx)

Description:

If there is no record of relation Modifies() of statement “s” and variable “x”, return FALSE.

Otherwise return TRUE.

INDEX ***insertModifies*** (STMT_NUM s, INDEX varIndexOfx) **Description:** If the relation Modifes(s, “x”), is not in Modify Table, insert it into the table and return its index value. Otherwise: return -1 (special value) and the table remains unchanged.
LIST<INDEX> ***getModifiedVarAtStmt*** (STMT_NUM s) **Description:** If s > 0 just return an array of all index values recorded in table whose variable are modified by statement “s”. Otherwise, return NULL.
LIST<STMT_NUM> ***getStmtModifyingVar*** (INDEX varIndexOfx) **Description:** If variable name “x” is recorded in VarTable, return an array of all statement numbers recorded in table that modify variable having index value “ind” in VarTable. Otherwise, return NULL.

8.5. Uses API

Uses
Overview: UsesTable is to keep the relationship Uses() of any pair of statements appearing in the program into a table.
Public Interface:

BOOLEAN ***isUses*** (int s, INDEX varIndexOfx)

Description:

If there is no record of relation Uses() of statement “s” and index of variable “x”, return FALSE.

Otherwise return TRUE.

INDEX ***insertUses*** (STMT_NUM s, INDEX varIndexOfx)

Description:

If the relation Uses(s, “x”), is not in Uses Table, insert it into the table and return its index value.

Otherwise: return -1 (special value) and the table remains unchanged.

LIST<INDEX> ***getUsedVarAtStmt*** (STMT_NUM s)

Description:

If s > 0 just return an array of all index values recorded in table whose variable are used by statement “s”.

Otherwise, return NULL.

LIST<STMT_NUM> ***getStmtUsingVar*** (INDEX varIndexOfx)

Description:

If variable name “x” is recorded in VarTable, return an array of all statement numbers recorded in table that use variable having index value varIndexOfx” in VarTable.

Otherwise, return NULL.

8.6. Statement Table API

StatTable

Overview: StatTable is to keep all the variables appearing in the program

Public Interface:
INDEX ***insertStmt*** (STRING name) **Description:** If “varName” is not in the VarTable, insert it into the VarTable and return its index value. Otherwise, return -1 (special value) and the table remains unchanged.
STRING ***getStmtName*** (INDEX ind) **Description:** If there is record in StatTable having index value “ind”, return its statement name. If “ind” is out of range: Returns “variable not found” message.
LIST<INDEX> ***getStmtIndex*** (string stmtName) **Description:** If there is record in StatTable having name “stmtName”, return its index value. Otherwise, return -1 (special value)

8.7. Calls API

Calls
Overview: Calls tables is to keep the pairs of procedures being called or calling.

Public Interface:
BOOLEAN ***isCalls*** (int proc1, int proc2) **Description:** If there is no record of relation Calls() of procedure “proc1” and “proc2” return FALSE. Otherwise return TRUE.
INDEX ***insertCalls*** (int proc1, int proc2) **Description:** If the relation Calls(proc1, proc2), is not in Calls Table, insert it into the table and return its index value. Otherwise: return -1 (special value) and the table remains unchanged.
VECTOR<INDEX> ***getCalledProc*** (int proc1) **Description:** Returns vector of called procedures by proc1.
VECTOR<STMT_NUM> ***getCallingProc***(int proc1) **Description:** Returns vector of calling procedures of proc1