29 September, 2014

CS3202 Software Engineering Project

ITERATION 2 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	A0088441	A0088441@nus.edu.sg
Vu Phuc Tho	A0090585X	A0090585@nus.edu.sg

1. Iteration Overview

1.1. Scope of SPA Implemented

In this second iteration we have followed the suggested implementation of the SPA according to the assignment document. We have extended the functionality of the PKB to include the Next relationship. The QP has been extended to deal with multiple clauses. More specifically the Pattern clause has been further implemented to deal with if/else statements and also to deal with more complex expressions.

1.2. Achievements & Problems

The main achievement for this iteration would have to be the fact that we managed to complete the requirements that had been left off from iteration 1 and implement all of the functionality required for this iteration. We also spent much more time testing the SPA considering many more test cases.

From iteration one we picked up on some problems of our system. The biggest problem that we picked up on, with the help of our tutor, was with the implementation of the class TNODE. This also affected our Parser. As such we had to spend a considerable amount of time dealing with this problem. This, alongside other issues, will be further elaborated on 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 TNODE	Next for PKB	Working on QP	Fixing issues with Parser	Report
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.

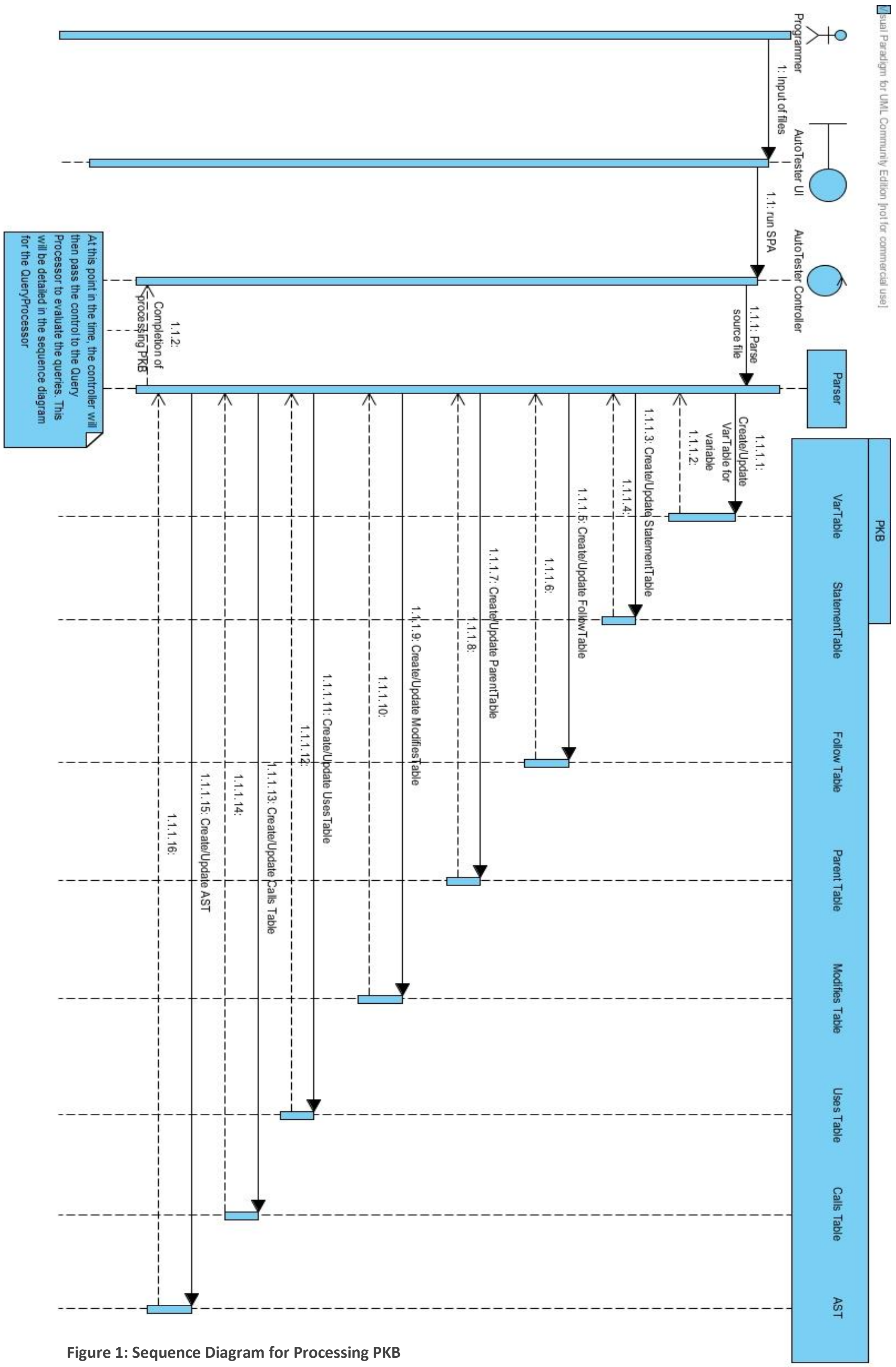

Visual Paradigm for UML Community Edition [not for commercial use]
Programmer
AutoTester UI
AutoTester Controller
Parser
PKB
VarTable
StatementTable
Follow Table
Parent Table
Modifies Table
Uses Table
Calls Table
AST
1: Input of files
1.1: run SPA
1.1.1: Parse source file
1.1.1.1: Create/Update VarTable for variable
1.1.1.2:
1.1.1.3: Create/Update StatementTable
1.1.1.4:
1.1.1.5: Create/Update FollowTable
1.1.1.6:
1.1.1.7: Create/Update ParentTable
1.1.1.8:
1.1.1.9: Create/Update ModifiesTable
1.1.1.10:
1.1.1.11: Create/Update UsesTable
1.1.1.12:
1.1.1.13: Create/Update Calls Table
1.1.1.14:
1.1.1.15: Create/Update AST
1.1.1.16:
1.1.2: Completion of processing PKB
At this point in the time, the controller will then pass the control to the Query Processor to evaluate the queries. This will be detailed in the sequence diagram for the QueryProcessor

Figure 1: Sequence Diagram for Processing PKB

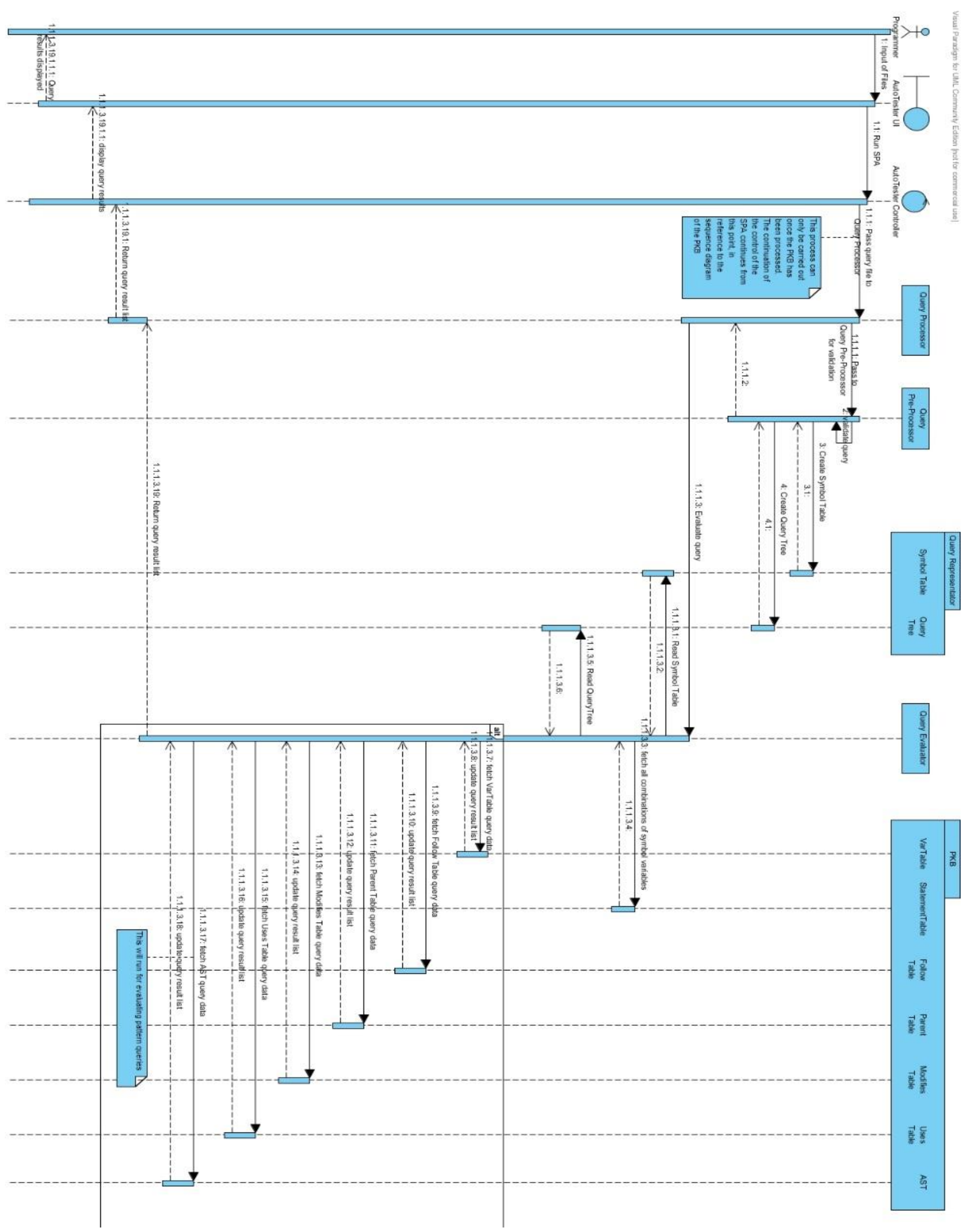
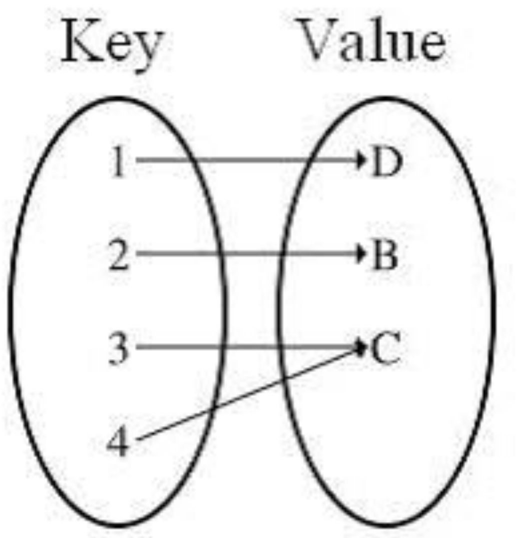

Visual Paradigm for UML Community Edition [not for commercial use]
Programmer
AutoTester UI
AutoTester Controller
Query Processor
Query Pre-Processor
Query Representator
Symbol Table
Query Tree
Query Evaluator
PKB
VarTable
StatementTable
Follow Table
Parent Table
Modifies Table
Uses Table
AST
1: Input of Files
1.1: Run SPA
1.1.1: Pass query file to Query Processor
This process can only be carried out once the PKB has been processed. The continuation of the control of the SPA continues from this point, in reference to the sequence diagram of the PKB
1.1.1.1: Pass to Query Pre-Processor for validation
2: validate query
1.1.1.2:
3: Create Symbol Table
3.1:
4: Create Query Tree
4.1:
1.1.1.3: Evaluate query
1.1.1.3.1: Read Symbol Table
1.1.1.3.2:
1.1.1.3.3: fetch all combinations of symbol variables
1.1.1.3.4:
1.1.1.3.5: Read QueryTree
1.1.1.3.6:
alt
1.1.1.3.7: fetch VarTable query data
1.1.1.3.8: update query result list
1.1.1.3.9: fetch Follow Table query data
1.1.1.3.10: update query result list
1.1.1.3.11: fetch Parent Table query data
1.1.1.3.12: update query result list
1.1.1.3.13: fetch Modifies Table query data
1.1.1.3.14: update query result list
1.1.1.3.15: fetch Uses Table query data
1.1.1.3.16: update query result list
1.1.1.3.17: fetch AST query data
1.1.1.3.18: update query result list
This will run for evaluating pattern queries
1.1.1.3.19: Return query result list
1.1.1.3.19.1: Return query result list
1.1.1.3.19.1.1: display query results
1.1.1.3.19.1.1.1: Query results displayed

Figure 2: Sequence Diagram for Query Processor Flow

4. Design Decisions

For the design of the next relationship, we decided not to use 2D vectors for its implementation. Even though we have used that for the other relationships in the PKB, the data structure that we decided on using is a Map. To implement it for the Next relationship, we have used 2 Maps as shown below.



One map is to keep track of the mapping to a statement that is directly next to a key. While the other map is used to keep track of the mapping to a statement that is directly behind a key.

The benefit seen here of using a map is that it reduces the amount of data storage needed, which is a problem with using a table. It also reduces the searching time to O(1). As such the query evaluation time for evaluating Next(x,y) is always O(1) for any x,y.

For the implementation of Next*, we have evaluated this relationship by continuously calling next from the PKB. So it's an extension of the Next relationship.

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:
    1. Methods to insert new records to the database should have the form `insertXXX()`.
    2. Methods with return value type BOOLEAN should have the form: `isXXX()` e.g. `isExist()`, `isMatchVar()`.
    3. Methods with return types of other values should have the form `getXXX()` e.g. `getVarName()`
    4. Methods that return the number of records inside a table/ list should have the form `getSize()`.
    5. Methods that change the values or status of an object should have the form: `setXXX()`
    6. 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

In this section we will go through how queries are evaluated. For the discussion below we will be referring to the sample query, shown below, to illustrate the process.

stmt s1, s2, s3;

Select s1 such that Parent*(s1, s2) pattern s3(_, _"x+y*2"_)

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).

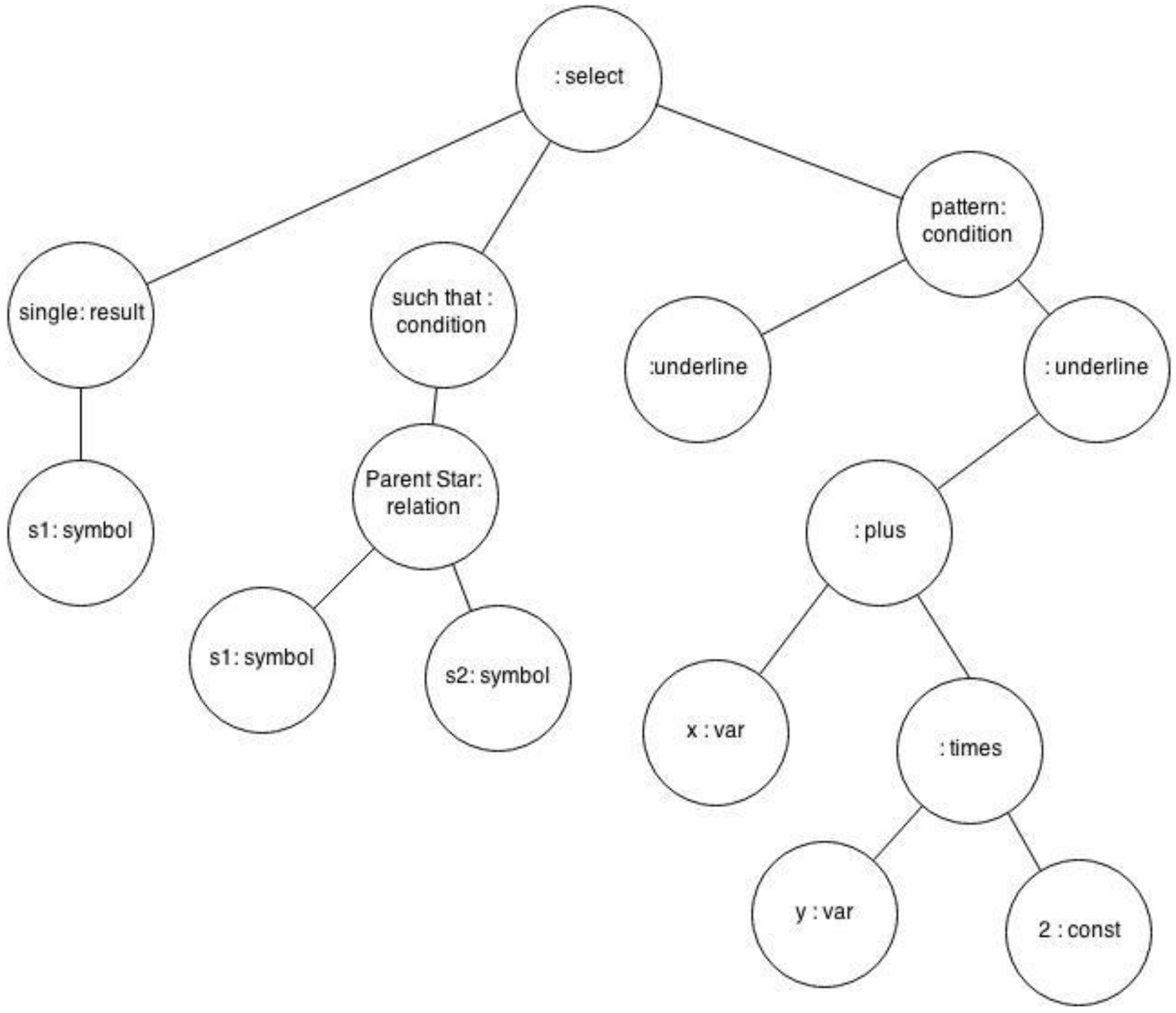
During validation, the QPP checks each query line by line, following these rules:

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

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

- While the `preprocessQueryPart()` method runs, if the QPP finds a new clause of query, it will call the corresponding method `preprocessClause()` (e.g. `preprocessSuchThatCondtion()`) 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. The QR will contain the query tree. An example of what the query tree, based on the sample query, will look like is shown below.


: select
single: result
s1: symbol
such that : condition
Parent Star: relation
s1: symbol
s2: symbol
pattern: condition
:underline
: underline
: plus
x : var
: times
y : var
2 : const

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.

The sample query shown below will be used as an example to explain the query evaluation process.

6.2.1. Manage the temporary results

Before evaluating a query, the QE creates a list of values of all declared symbols. The size of this list is the number of symbols declared in the query, and all symbols are initialized with a dummy value (-1).

Using the example shown above, this list will look something like the table shown below.

S1	S2	S3
-1	-1	-1

During processing a query's condition, if a new value is found, the QE will try to update the list of values. The QE will replace the dummy value with the found value, and use the updated list to continue the evaluation. The list continues to be updated until there is no more condition left or a condition which cannot be satisfied appears.

Since we have considered all of the symbols in the query while creating this list, we do not need to consider expanding this list during the evaluation. Once we have finalized this list, the result node from the query tree will be read and the result to be returned will be determined and returned.

6.2.2. Optimize the evaluation process

Currently, the QPP saves query's conditions in the same order as they appear in the query, and the QE also processes the conditions from left to right. This method, while being correct, is not an optimized one since with suitable order of evaluating the conditions, the QE can work more efficiently.

There are some alternative methods to process the query's conditions:

- Choose condition based on type: a simple solution is instead of evaluating from left to right, the QE will try to find the type of condition (such that, pattern or with) and decide itself which condition to process first.
- Rank the conditions: after preprocessing the query, the QPP can help in ranking the query's conditions based on several criteria: number of symbols, frequency of using symbol, type of conditions, etc. The ranking will be used by QE later for choosing condition for evaluation.

The one that we have chosen to implement to optimize our query evaluation process is the first method mentioned above.

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

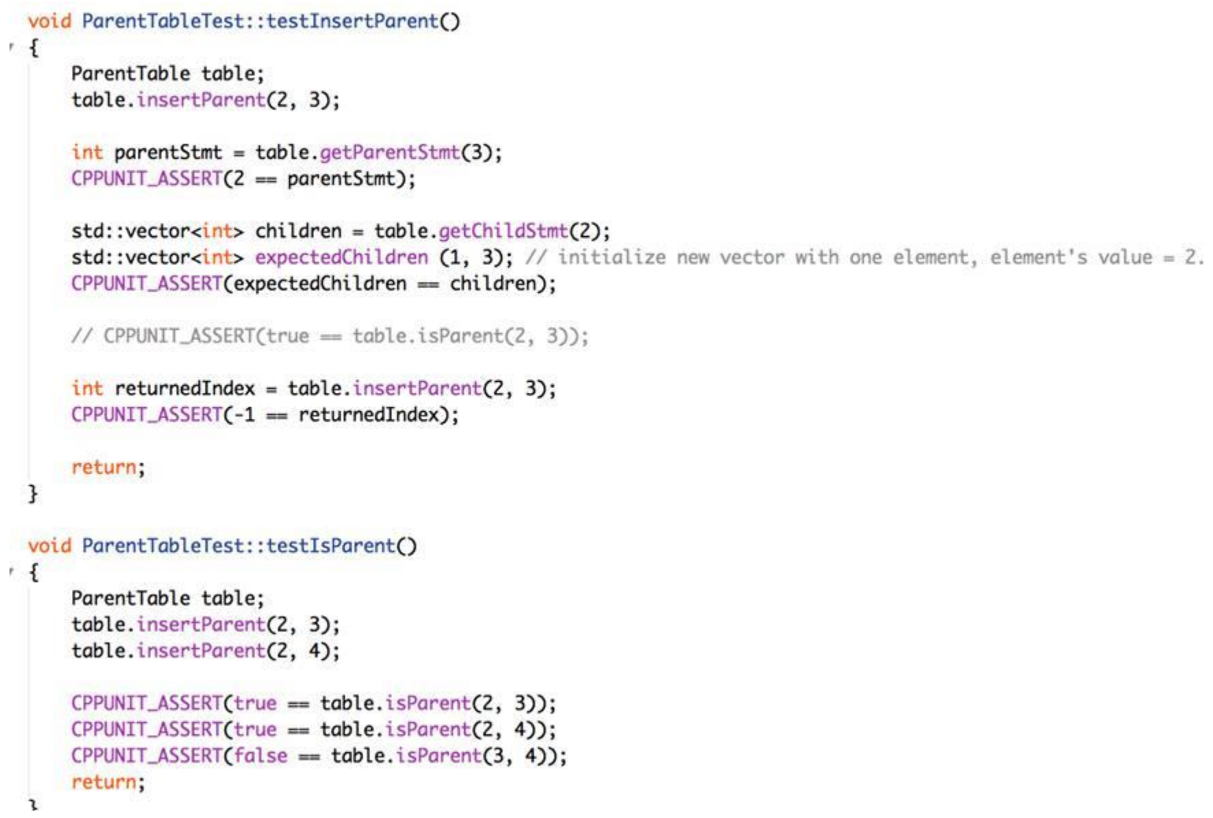
7.2.1. Unit Testing

7.2.1.1. PKB

<u>Sample 1</u>

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 3: Parent Table Unit Test

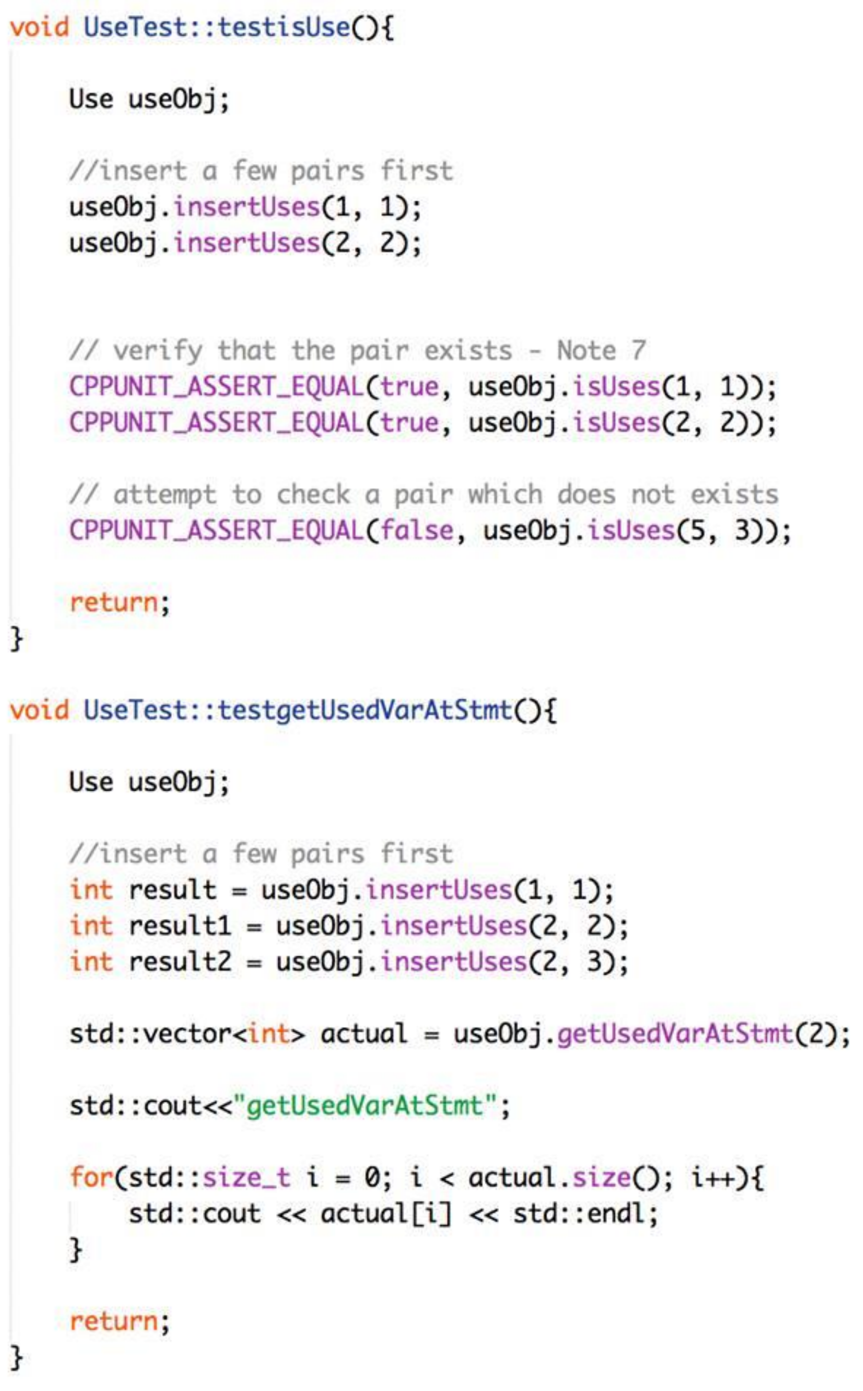
Expected Test Results: “OK” This uses the CPP Unit assert, as seen in figure 5 above.

Other Requirements: None

<u>Sample 2</u>

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 4: Uses Table Unit Test

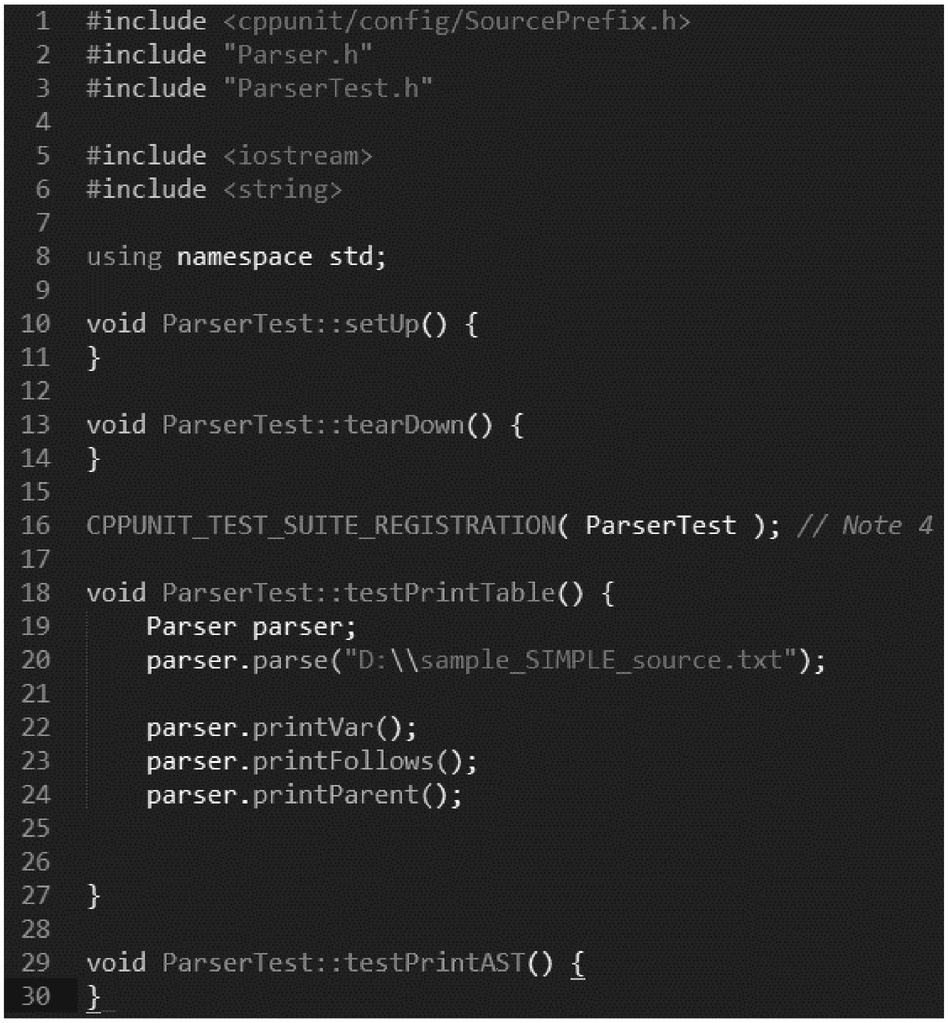
Expected Test Results: “OK” This uses the CPP Unit assert, as seen in figure 6 above.

Other Requirements: None

7.2.1.2. Parser

Test Purpose: To test the Parser component of the SPA

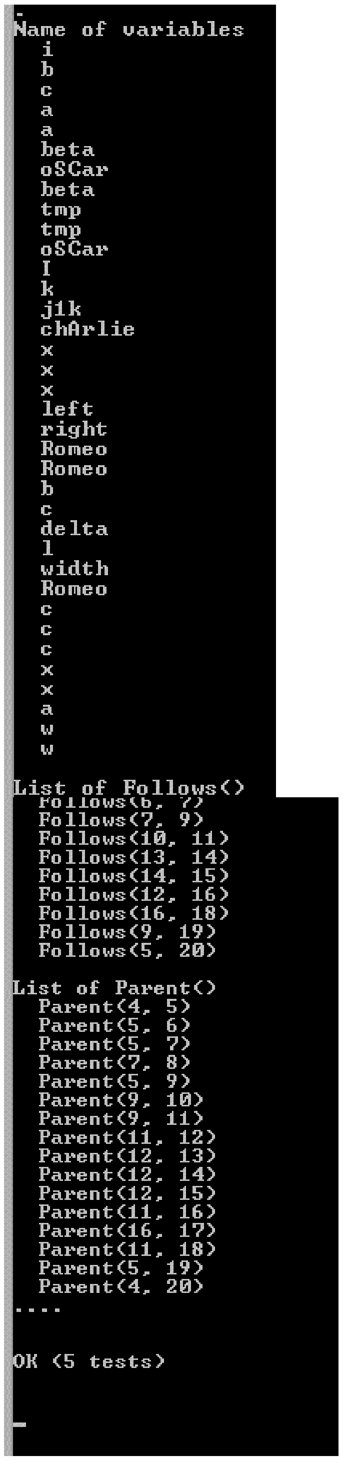
Required Test Inputs: Parser and CPP Unit test case for Parser. An example is shown in figure 7 below.



```
#include <cppunit/config/SourcePrefix.h>
#include "Parser.h"
#include "ParserTest.h"

#include <iostream>
#include <string>

using namespace std;

void ParserTest::setUp() {
}

void ParserTest::tearDown() {
}

CPPUNIT_TEST_SUITE_REGISTRATION( ParserTest ); // Note 4

void ParserTest::testPrintTable() {
    Parser parser;
    parser.parse("D:\\sample_SIMPLE_source.txt");

    parser.printVar();
    parser.printFollows();
    parser.printParent();


}

void ParserTest::testPrintAST() {
}
```

Figure 5: 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.



```
-
Name of variables
  i
  b
  c
  a
  a
  beta
  oSCar
  beta
  tmp
  tmp
  oSCar
  I
  k
  j1k
  chArlie
  x
  x
  x
  left
  right
  Romeo
  Romeo
  b
  c
  delta
  l
  width
  Romeo
  c
  c
  c
  x
  x
  a
  w
  w

List of Follows()
  Follows(6, 7)
  Follows(7, 9)
  Follows(10, 11)
  Follows(13, 14)
  Follows(14, 15)
  Follows(12, 16)
  Follows(16, 18)
  Follows(9, 19)
  Follows(5, 20)

List of Parent()
  Parent(4, 5)
  Parent(5, 6)
  Parent(5, 7)
  Parent(7, 8)
  Parent(5, 9)
  Parent(9, 10)
  Parent(9, 11)
  Parent(11, 12)
  Parent(12, 13)
  Parent(12, 14)
  Parent(12, 15)
  Parent(11, 16)
  Parent(16, 17)
  Parent(11, 18)
  Parent(5, 19)
  Parent(4, 20)
....


OK (5 tests)


_
```

Figure 6: Test Results for Unit Testing

Other Requirements: None

7.2.2. Integration Testing

7.2.2.1. Parser & PKB

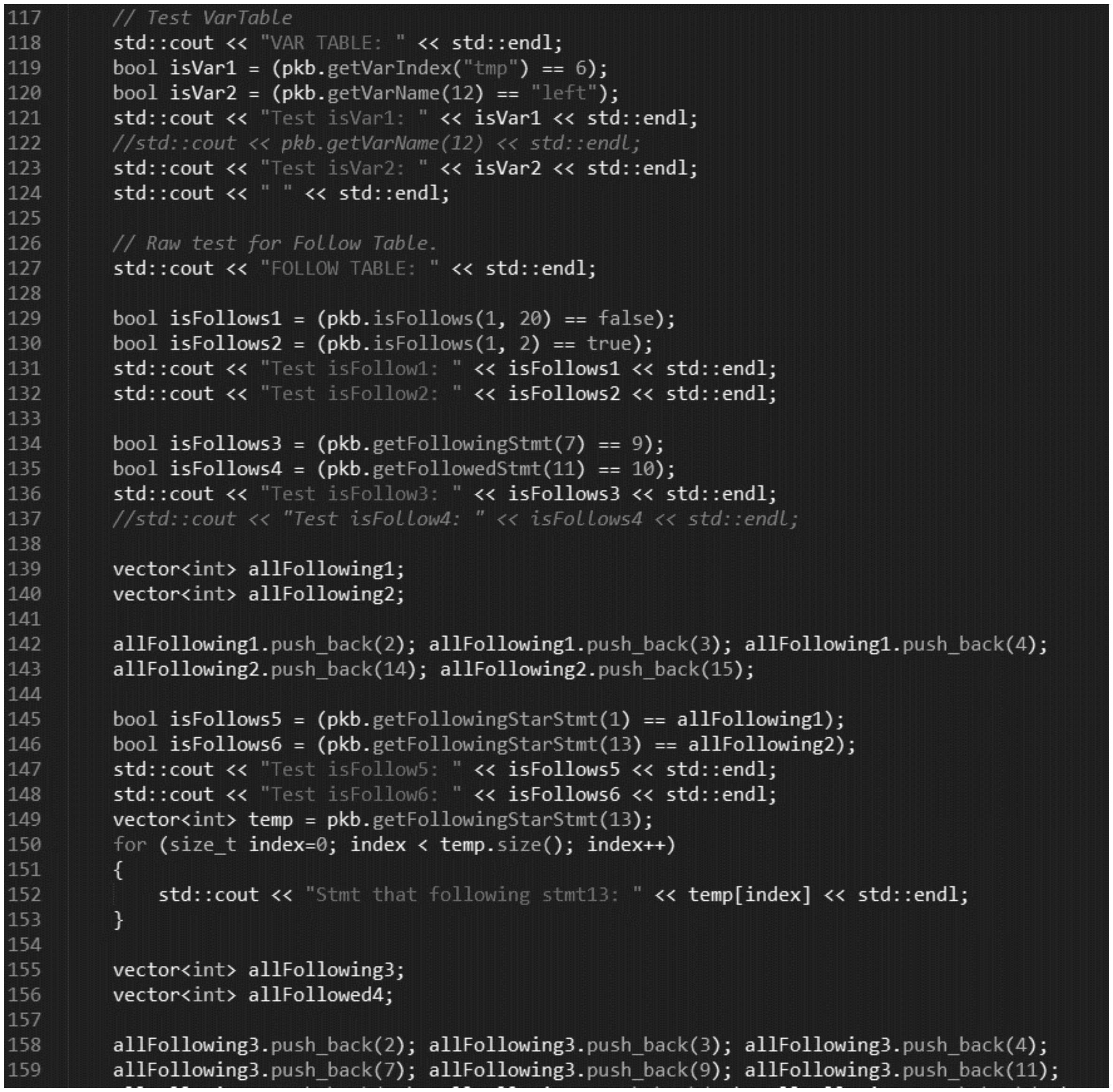
Figures 5- 7 show the implementation of the integration testing done using CPP Unit.

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

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

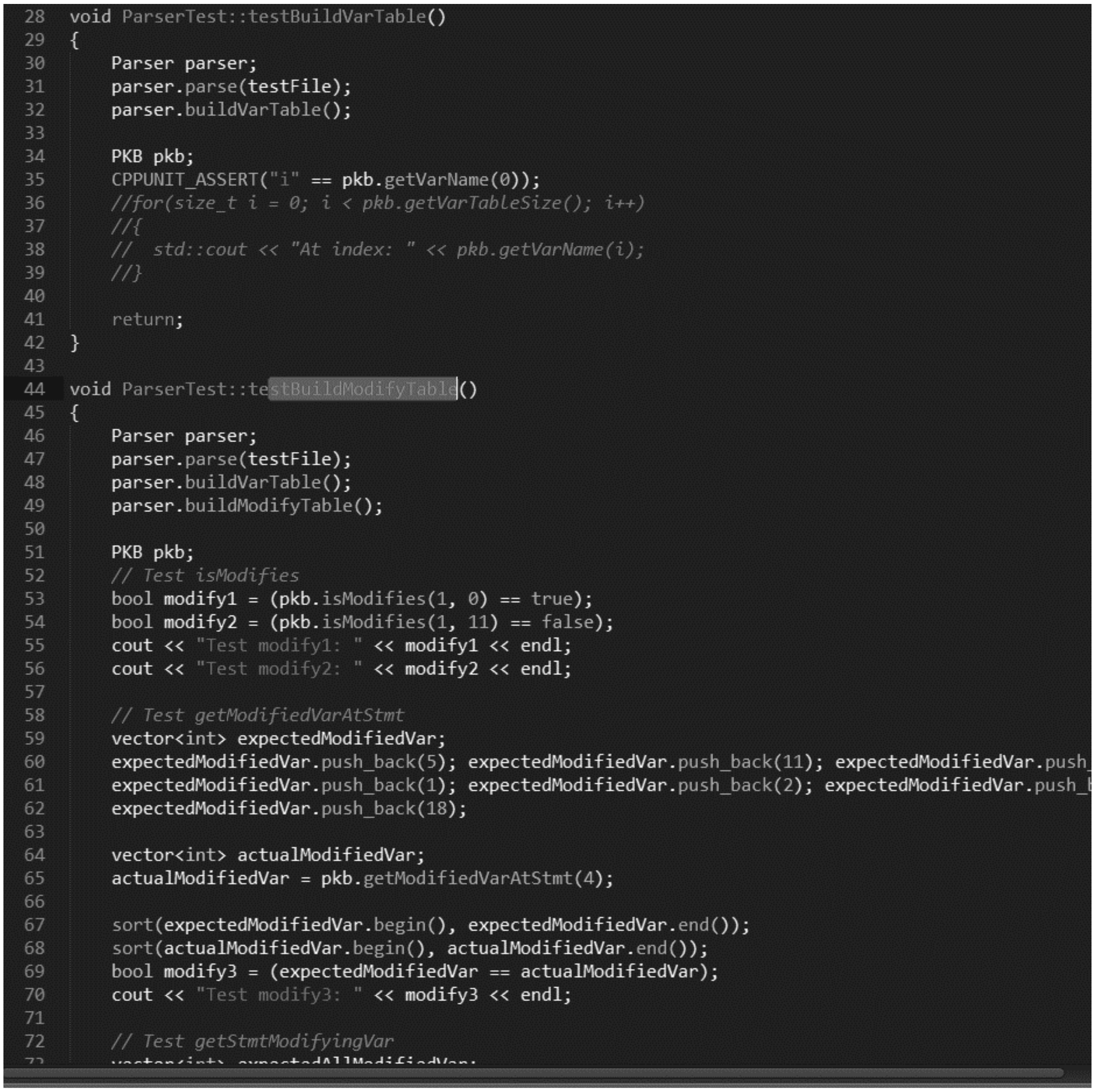
Expected Test Results: Passed all

Other Requirements: None



```
    // Test VarTable
    std::cout << "VAR TABLE: " << std::endl;
    bool isVar1 = (pkb.getVarIndex("tmp") == 6);
    bool isVar2 = (pkb.getVarName(12) == "left");
    std::cout << "Test isVar1: " << isVar1 << std::endl;
    //std::cout << pkb.getVarName(12) << std::endl;
    std::cout << "Test isVar2: " << isVar2 << std::endl;
    std::cout << " " << std::endl;

    // Raw test for Follow Table.
    std::cout << "FOLLOW TABLE: " << std::endl;

    bool isFollows1 = (pkb.isFollows(1, 20) == false);
    bool isFollows2 = (pkb.isFollows(1, 2) == true);
    std::cout << "Test isFollow1: " << isFollows1 << std::endl;
    std::cout << "Test isFollow2: " << isFollows2 << std::endl;

    bool isFollows3 = (pkb.getFollowingStmt(7) == 9);
    bool isFollows4 = (pkb.getFollowedStmt(11) == 10);
    std::cout << "Test isFollow3: " << isFollows3 << std::endl;
    //std::cout << "Test isFollow4: " << isFollows4 << std::endl;

    vector<int> allFollowing1;
    vector<int> allFollowing2;

    allFollowing1.push_back(2); allFollowing1.push_back(3); allFollowing1.push_back(4);
    allFollowing2.push_back(14); allFollowing2.push_back(15);

    bool isFollows5 = (pkb.getFollowingStarStmt(1) == allFollowing1);
    bool isFollows6 = (pkb.getFollowingStarStmt(13) == allFollowing2);
    std::cout << "Test isFollow5: " << isFollows5 << std::endl;
    std::cout << "Test isFollow6: " << isFollows6 << std::endl;
    vector<int> temp = pkb.getFollowingStarStmt(13);
    for (size_t index=0; index < temp.size(); index++)
    {
        std::cout << "Stmt that following stmt13: " << temp[index] << std::endl;
    }

    vector<int> allFollowing3;
    vector<int> allFollowed4;

    allFollowing3.push_back(2); allFollowing3.push_back(3); allFollowing3.push_back(4);
    allFollowing3.push_back(7); allFollowing3.push_back(9); allFollowing3.push_back(11);
```

Figure 7: CPP Unit Integration Testing for Parser & PKB



```
void ParserTest::testBuildVarTable()
{
    Parser parser;
    parser.parse(testFile);
    parser.buildVarTable();

    PKB pkb;
    CPPUNIT_ASSERT("i" == pkb.getVarName(0));
    //for(size_t i = 0; i < pkb.getVarTableSize(); i++)
    //{
    //  std::cout << "At index: " << pkb.getVarName(i);
    //}

    return;
}

void ParserTest::testBuildModifyTable()
{
    Parser parser;
    parser.parse(testFile);
    parser.buildVarTable();
    parser.buildModifyTable();

    PKB pkb;
    // Test isModifies
    bool modify1 = (pkb.isModifies(1, 0) == true);
    bool modify2 = (pkb.isModifies(1, 11) == false);
    cout << "Test modify1: " << modify1 << endl;
    cout << "Test modify2: " << modify2 << endl;

    // Test getModifiedVarAtStmt
    vector<int> expectedModifiedVar;
    expectedModifiedVar.push_back(5); expectedModifiedVar.push_back(11); expectedModifiedVar.push_
    expectedModifiedVar.push_back(1); expectedModifiedVar.push_back(2); expectedModifiedVar.push_
    expectedModifiedVar.push_back(18);

    vector<int> actualModifiedVar;
    actualModifiedVar = pkb.getModifiedVarAtStmt(4);

    sort(expectedModifiedVar.begin(), expectedModifiedVar.end());
    sort(actualModifiedVar.begin(), actualModifiedVar.end());
    bool modify3 = (expectedModifiedVar == actualModifiedVar);
    cout << "Test modify3: " << modify3 << endl;

    // Test getStmtModifyingVar
    vector<int> expectedAllModifiedVar;
```

Figure 8: CPP Unit Integration Testing for Parser & PKB

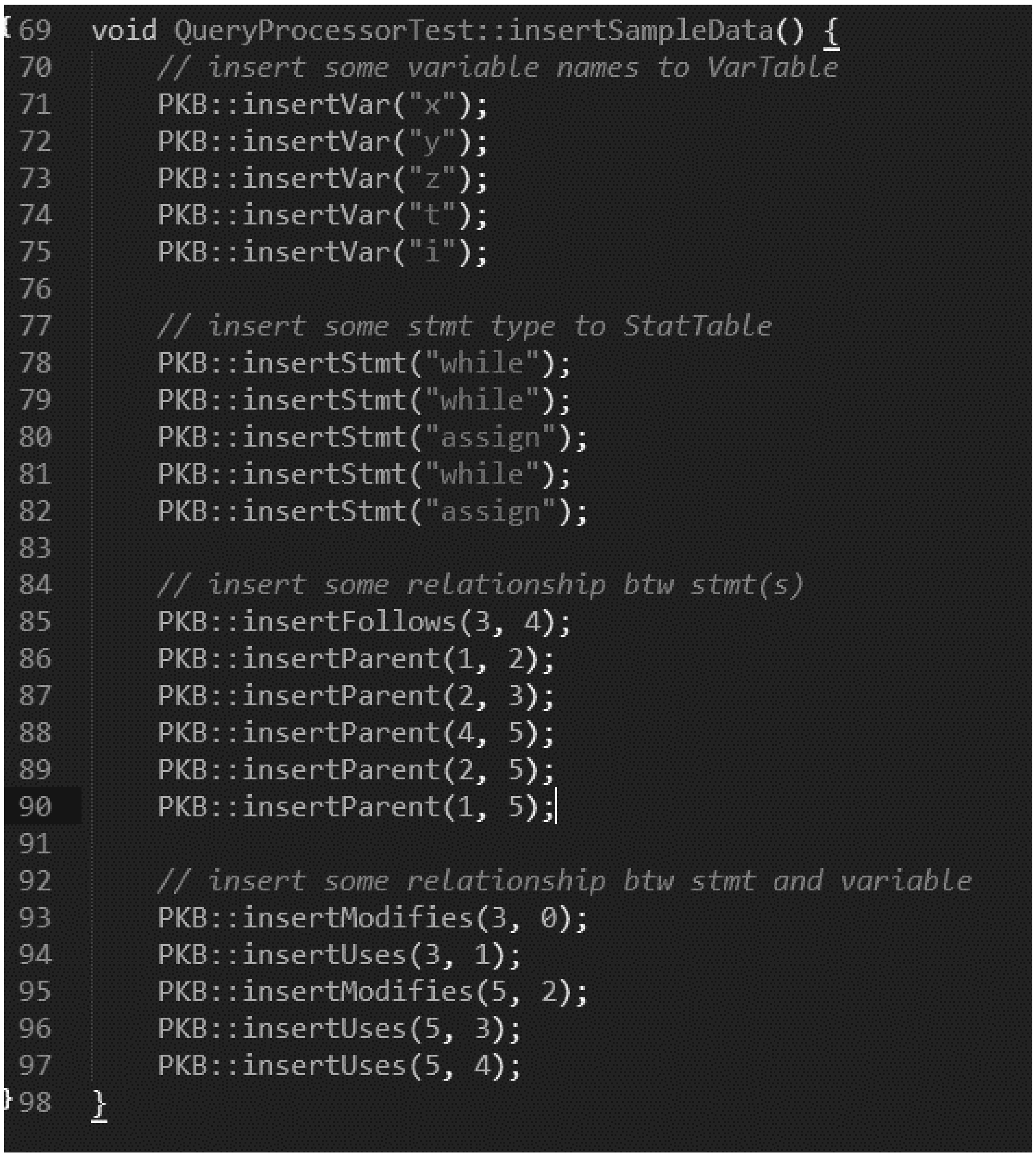
7.2.2.2. PKB & QP

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



```
void QueryProcessorTest::insertSampleData() {
    // insert some variable names to VarTable
    PKB::insertVar("x");
    PKB::insertVar("y");
    PKB::insertVar("z");
    PKB::insertVar("t");
    PKB::insertVar("i");

    // insert some stmt type to StatTable
    PKB::insertStmt("while");
    PKB::insertStmt("while");
    PKB::insertStmt("assign");
    PKB::insertStmt("while");
    PKB::insertStmt("assign");

    // insert some relationship btw stmt(s)
    PKB::insertFollows(3, 4);
    PKB::insertParent(1, 2);
    PKB::insertParent(2, 3);
    PKB::insertParent(4, 5);
    PKB::insertParent(2, 5);
    PKB::insertParent(1, 5);

    // insert some relationship btw stmt and variable
    PKB::insertModifies(3, 0);
    PKB::insertUses(3, 1);
    PKB::insertModifies(5, 2);
    PKB::insertUses(5, 3);
    PKB::insertUses(5, 4);
}
```

Figure 9: CPP Unit Integration Testing for PKB & QP

7.2.3. Validation Testing

<u>Sample 1</u>

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 10: 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 11: 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 12: 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 13: Expected Test Results

Other Requirements: None

8. Discussion

In this iteration, the biggest challenge was to complete the requirements that we were unable to implement for iteration 1 and work on extending the functionality for iteration 2 within the time-frame allocated for iteration 2. We managed to do that while recognizing certain issues with the system.

The biggest problem that we found out was with the implementation of TNODE. This class is used to build various trees used for data storage. Previously we had implemented the TNODE without the use of pointers. So every time we would pass the data it would create a new copy instead of passing the original data form one function to another. This would slow down the SPA process, since each time we would add a new node, the existing tree would be copied another time to attach to the new node. This is a waste of processing time and storage. This slowed our system down considerably.

The next thing that we did to overcome the aforementioned problem was that we decided to use pointers to fix this prob. In every method we assigned the pointer to a local object within the method. So when the method is returned the local object is cleaned. So this pointer would point to nothing. This was another thing that we had to work on.

Finally we managed to fix this problem as well. Instead of creating a new TNODE and assigning a pointer to its address, we are now simply creating a pointer of type TNODE.

Old way:

```
TNode node();

TNode * pointer = &node;
```

New way:

```
TNode * pointer = new TNode();
```

This way the data will not be lost when a particular function is returned.

Secondly we had to spend time working on debugging the Parser as it was failing certain test cases. This issues was dealt with in a timely manner.

Finally we also managed to fix the problem that we had with running the Autotester and outputting the correct results.

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.**

BOOLEAN *isModifiesProc* **(PROC proc, INDEX varIndexOfx)**

Description:

If there is no record of relation Modifies() of procedure "proc" and variable "x", return FALSE.

Otherwise return TRUE.

INDEX *insertModifiesProc* **(PROC proc, INDEX varIndexOfx)**

Description:

If the relation Modifes(proc, "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> *getModifiedVarAtProc* **(PROC proc)**

Description:

If proc > 0 just return an array of all index values recorded in table whose variable are modified by procedure "proc".

Otherwise, return NULL.

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

Description:

If variable name "x" is recorded in VarTable, return an array of all procedure 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<PROCEDURE> *getCalledProc* (int proc1) **Description:** **Returns vector of called procedures by proc1.**
VECTOR<PROCEDURE> *getCallingProc*(int proc1) **Description:** **Returns vector of calling procedures of proc1**
BOOLEAN *isCallStar* (int proc1, int proc2) **Description:** **If there is no record of relation Calls() of procedure “proc1” and “proc2” return FALSE.** **Otherwise return TRUE.**
VECTOR<PROCEDURE> *getCallingStarProc*(int proc1)

Description: **Returns vector of calling procedures of proc1 with the star relationship**
VECTOR<PROCEDURE> ***getCalledStarProc*** **(int proc1)** **Description:** **Returns vector of called procedures by proc1 with star relationship.**

8.8. Next API

Next
Overview: **Next tables is to keep the pairs of line numbers next to each other.**
Public Interface:
BOOLEAN ***isNext*** **(int n1, int n2)** **Description:** **Returns true if n2 is next to n1. Otherwise false.**

INDEX *insertNext* (int n1, int n2) **Description:** **If the relation Next(n1, n2) is not in Next Table, insert it into the table and return the sizeof the table.** **Otherwise: return -1 (special value) and the table remains unchanged.**
VECTOR<STMT_NUM> *getNextStmts*(int s1) **Description:** **Returns vector of stmt numbers next to s1.**
VECTOR<STMT_NUM> *getPreviousStmts*(int n1) **Description:** **Returns vector of stmt numbers previous of n1.**
BOOLEAN *isNextStar* (int n1, int n2) **Description:** **If there is no record of relation Next() of line numbers n1 and n2 return FALSE.** **Otherwise return TRUE.**
VECTOR<STMT_NUM> *getNextStarStmts*(int n1) **Description:**

Returns vector of next to n1 with the star relationship
VECTOR< STMT_NUM > *getPreviouStarStmts* (int n1) **Description:** **Returns vector of previous to n1 with the star relationship**

8.9. PKB API

PKB
Overview: The PKB contains all the components required for the storage of data. Such as the tables and AST.
Public Interface:
static BOOLEAN isFollows(STMT_NUM s1, STMT_NUM s2) **Description:** **Method to return if statement s1 is followed by statement s2. Return true if relationship holds, otherwise return false.**
static BOOLEAN isFollowsStar(STMT_NUM s1, STMT_NUM s2)

Description: **Method to check Follows*(s1, s2) holds. Return true if relationship holds, otherwise return false.**
static BOOLEAN insertFollows(STMT_NUM s1, STMT_NUM s2) **Description:** **Method to insert a pair of following statement numbers in FollowTable. Return true if successful, otherwise return false.**
static STMT_NUM getFollowingStmt(STMT_NUM s1) **Description:** **Method to get the following statement to statement number s1.** **Return the statement number if found, otherwise return -1.**
static STMT_NUM getFollowedStmt(STMT_NUM s1) **Description:** **Method to get statement which is followed by statement s1.** **Return the statement number if found, otherwise return -1.**
LIST<STMT_NUM> getFollowingStarStmt(STMT_NUM s1) **Description:** **Method to get the list of following statements to statement number s1 with relationship Follows*.** **Otherwise, return NULL.**
LIST<STMT_NUM> getFollowedStarStmt(STMT_NUM s1) **Description:** **Method to get the list of statements which are followed star by statement s1.**

Otherwise, return NULL.
LIST<STMT_NUM> getAllFollowingStmt() **Description:** **Method to get statement which is followed star by statement s1.** **Otherwise, return NULL.**
LIST<STMT_NUM> getAllFollowedStmt() **Description:** **Method to get statement which is followed star by statement s1.** **Otherwise, return NULL.**
INTEGER getFollowTableSize() **Description:** **Returns the number of records in FollowTable.**
BOOLEAN isModifies(STMT_NUM s1, INDEX varIndex) **Description:** **Method to check if modifies relationship exists. Return true if exists, otherwise return false.**
BOOLEAN insertModifies(STMT_NUM s1, INDEX varIndex) **Description:** **Method to insert a pair of statement number and variable holding the Modify relationship in ModifyTable.** **Return true if successful, otherwise return false.**

LIST<INDEX> getModifiedVarAtStmt(STMT_NUM s1) **Description:** **Method to get the variables modified in statement s1.** **Otherwise, return NULL.**
LIST<STMT_NUM> getStmtModifyingVar(INDEX varIndex) **Description:** **Method to get the list of statements that modify var.** **Otherwise, return NULL.**
LIST<STMT_NUM> getAllModifyingStmt() **Description:** **Method to get all statements modifying some variable.** **Otherwise, return NULL.**
LIST<INDEX> getAllModifiedVar() **Description:** **Method to get all variables being modified.** **Otherwise, return NULL.**
INTEGER getModifyTableSize() **Description:** **Returns the number of records in ModifyTable.**
INDEX insertVar(STRING name) **Description:** **Inserts variable in VarTable and returns its index.**

Otherwise, return -1.

INDEX getVarIndex(STRING name)

Description:

Returns the index of the variable.

Otherwise, return -1.

STRING getVarName(INDEX index)

Description:

Returns the variable name given its index.

Otherwise, return NULL.

INTEGER getVarTableSize()

Description:

Returns the number of records in VarTable.

BOOLEAN isParent(STMT_NUM s1, STMT_NUM s2)

Description:

Method to return if statement s1 is parent of statement s2. Return true if relationship holds, otherwise return false.

BOOLEAN isParentStar(STMT_NUM s1, STMT_NUM s2)

Description:

Method to return if statement s1 is parent star of statement s2. Return true if relationship holds, otherwise return false.
BOOLEAN insertParent(STMT_NUM s1, STMT_NUM s2) **Description:** **Method to insert a pair of parent and child statement numbers in ParentTable.** **Return true if successful, otherwise return false.**
STMT_NUM getParentStmt(STMT_NUM childStmt) **Description:** **Return the statement number that is parent of child statement if found in ParentTable.** **Otherwise return -1.**
LIST<STMT_NUM> getChildStmt(STMT_NUM parentStmt) **Description:** **Return an array of child statement numbers of parent statement if found in ParentTable.** **Otherwise return NULL.**
LIST<STMT_NUM> getParentStarStmt(STMT_NUM childStmt) **Description:** **Return an array of statement numbers that are parent star of child statement if found in ParentTable.** **Otherwise return NULL.**
LIST<STMT_NUM> getChildStarStmt(STMT_NUM parentStmt) **Description:**

Return an array of child statement numbers of parent star statement if found in ParentTable. **Otherwise return NULL.**
LIST<STMT_NUM> getAllParentStmt() **Description:** **Return an array of all statements that are parent of some child statement found in ParentTable.** **Otherwise return NULL.**
LIST<STMT_NUM> getAllChildStmt() **Description:** **Return an array of all statements that are child of some parent statement found in ParentTable.** **Otherwise return NULL.**
INTEGER getParentTableSize() **Description:** **Returns the number of records in ParentTable.**
BOOLEAN insertStmt(STRING name) **Description:** **Inserts statement in StatTable.** **Return true if successful, otherwise return false.**
LIST<STMT_NUM> getStmtIndex(STRING name) **Description:** **Return index of statement having name in StatTable.** **Otherwise, return NULL.**

STRING getStmtName(INDEX index);

Description:

Return name of statement having index in StatTable.
Otherwise, return NULL.

INTEGER getStatTableSize()

Description:

Returns the number of records in StatTable.

BOOLEAN isUses(STMT_NUM s1, INDEX varIndex)

Description:

Method to check if uses relationship exists.

Return true if exist, otherwise return false.

BOOLEAN insertUses(STMT_NUM s1, INDEX varIndex)

Description:

Method to insert a pair of following statement number and variable.

Return true if successful, otherwise return false.

LIST<INDEX> getUsedVarAtStmt(STMT_NUM s1)

Description:

Method to get the variables used in statement s1.

Otherwise, return NULL.

LIST<STMT_NUM> getStmtUsingVar(INDEX varIndex) **Description:** **Method to get the list of statements using variable.** **Otherwise, return NULL.**
LIST<STMT_NUM> getAllUsingStmt() **Description:** **Method to get the list of all statements using some variable.** **Otherwise, return NULL.**
LIST<INDEX> getAllUsedVar() **Description:** **Method to get all variables being used in some statement.** **Otherwise, return NULL.**
INTEGER getSize() **Description:** **Method to get the number of records in UsesTable.**

8.10. AST API

AST

Overview: The AST stores the source file in a tree representation.
Public Interface:
TNode findNodeOfStmt(INDEX index) **Description:** **Returns the node with statment number "index" in AST. Otherwise, return null.**
BOOLEAN hasSubTree(Tree tree) **Description:** **Checks if AST has the subtree “tree”. Returns true if found, otherwise return false.**

8.11. Tree API

Tree
Overview: The Tree is used as the underlying structure for the implementation of the AST and Query Tree
Public Interface:
TNode createNode(STRING type, STRING value) **Description:** **Creates a Tnode with type and value and returns the node.**

TNode getRoot()

Description:

Returns the root node of the tree.

INTEGER getNumChildren(TNode node)

Description:

Returns the number of child nodes of the node

TNode getChildAtIndex(TNode node, INDEX index)

Description:

Returns the child node of node at the “index” position.

void setRoot(TNode node)

Description:

Sets the node as the root of the tree.

INDEX setChild(TNode parent, TNode child)

Description:

Sets the node as the child node of parent node and returns the index position of the child node.