National University of Singapore

CS3201/2 Iteration 1

Summary

This is our implementation for Iteration 1.

We have developed a prototype with the given requirements.

SEM 1, AY2017/2018

Submitted By :

Group 11

Akankshita Dash (A0132788U)

Chua Ping Chan (A0126623L)

Lau Wenhao (A0121528M)

Marcus Ng (A0139257X)

Zhang Ying (A0142939W)

Zhuang Lei (A0140055W)

1 Development Plan

Team Member	**PKB**	**PQL**	**Testing**	**Report**	**Integration**
Akankshita Dash		Y		Y	
Chua Ping Chan	Y				Y
Lau Wenhao		Y			Y
Marcus Ng		Y	Y		
Zhang Ying	Y			Y	
Zhuang Lei	Y		Y		

Team Member	**E-mail**	**Phone number**
Akankshita Dash	akankshita.dash@u.nus.edu	84010419
Chua Ping Chan	a0126623@u.nus.edu	90876406
Lau Wenhao	a0121528@u.nus.edu	84991295
Marcus Ng	marcusngwenjian@u.nus.edu	97502493
Zhang Ying	e0006954@u.nus.edu	82289895
Zhuang Lei	e0003940@u.nus.edu	98374236

2 Scope of Prototype Implementation

Our prototype for SPA meets all the requirements for Iteration 1 and is a fully operational mini-SPA.

It is able to parse a source **SIMPLE** program containing multiple statements, assignments and while loops. Our PKB implements tables for **Follows, Follows*, Parent, Parent*, Uses, Modifies and Pattern** to store relation clauses.

We evaluate queries given in the form of:

Declaration(s); Select statement (optional such that and/or pattern).

We are able to validate the query, build a query tree with the clauses, evaluate it and project results.

We also added Integration tests for our prototype (not required for this iteration).

3 SPA Design

3.1 Overview

In the SPA, there are two main components:

1. Program Knowledge Base (PKB)
2. Program Query Language (PQL)

In PKB, we have the following components:

1. SPA Front-end Parser
2. PKB Main
3. PKB Tables
    - Variable Index Table

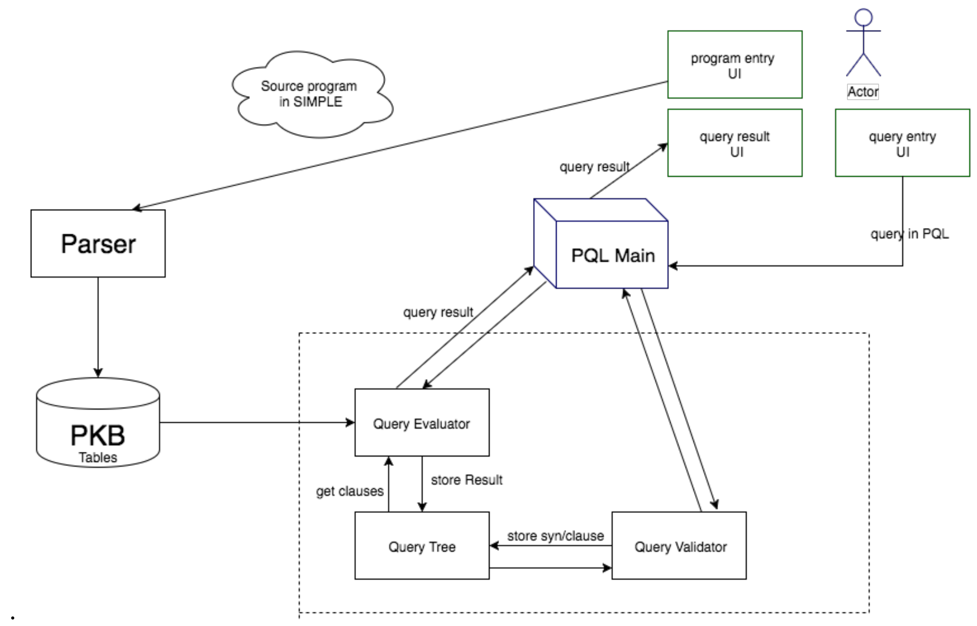

Figure 1: Main components of SPA

- Statement Type Lists
- Parent Table
- Follows Table
- Modifies Table
- Uses Table
- Pattern Table

4. Design Extractor

We have implemented different tables to cater for different design entities called by the query evaluator. This structure can allow easier extensions for the future iterations when more design entities are needed. Design extractor is used to process Parent Table and Follows Table in order to cater to Parent* and Follow* design entities. In PQL, we have the following components:

1. PQL Main

2. Query Pre-Processor

    - Query Validator

    - Query Tree

3. Query Evaluator

We have implemented a PQL Main and Query Tree (for storage), components not specified explicitly in the handbook. PQL Main controls the flow of all the other components in PQL and decides the sequence during query processing. Query Tree acts as storage for validated synonyms and clauses. The Validator and Evaluator interact with Query Tree and store information about the query.

3.2 Design of SPA Components

3.2.1 SPA Front-End(Parser)

The front-end parser is responsible for parsing the SIMPLE source code, validating syntax, and extracting information to populate the Program Knowledge Base. Because parser is only needed for parsing and does not need to store information, it is instantiated only during parsing and subsequently destroyed. A predictive recursive descent parser was implemented, where the SIMPLE source text is tokenized and the prediction and validation of each token done based on the preceding token. Because SIMPLE is a simplified programming language, it is possible to be certain about the just by inspecting the current one. In cases when the expected token is not to be found, it means there is a syntax error in the SIMPLE source code and the program will end.

3.2.2 PKB

Program Knowledge Base(PKB) provides a storage to store program design abstractions such as Uses, Modifies and others. Table-driven approach is used for the design of the PKB structure. Not only it provides a simple yet powerful way to achieve flexibility, but also allow quick accessing of the data by the way of hash maps. We have decided to make different class table files according to the different design entities. This is to make extension to allow additions of new design entities easier and this could minimize modifications of code.

Tables vs AST

Abstract Syntax Tree (AST) is one of the conventional ways to store parsed data into. However, when it comes to accessing the data by the Query Evaluator component, it have to perform traversing of the tree, which could take a lot of runtime during query processing. Also, doing modifications to allow more design entities would be more complicated. Alternatively, hash maps can be used to store data. Having multiple hash maps over AST allows faster searching, addition, deletion of data.

The table below shows the time complexity of query processing when AST and tables are used ('s' represents the number of statements in the SIMPLE programme):

	Using AST	Using Tables
Data searching	O(log s)	O(1)
Retrieving single data	O(log s)	O(1)
Retrieving all the data	O(s log s)	O(s)

Hence, using tables in the form of hash maps is used in this project so that query processing can take as fast as O(1) time.

PKB-Main: PKB-Main controls all the tables in the entire PKB component, it will be used by only SPA front-end parser and query evaluator components in order to reduce dependency. The APIs of the PKB-Main plays a central role in the SPA architecture.

Variable index table

The role of the variable index table is to map all the stored variables into a arbitrary index in order to avoid string comparison when it comes to using strings as a key in some hash maps.

Statement type lists

The role of the statement type lists is to store the statement numbers according to its synonyms. For example, if the statement number is an assignment statement, it is stored in the **assign list**. Similarly, statement number of the while statement is stored in **while list**.

As such, when query evaluator asks for assign or while statements, we are able to filter away the unwanted data in order to retrieve the corresponding statements.

Parent table

It stores statement number as a key, and a list of statement numbers that is the parent or child of that statement number. This table will be accessed when query evaluator calls Parent(s1,s2). Parents tables will be computed in a design extractor to allow Parent*(s1,s2) to be called.

Follows table

It stores the current statement in a hashmap and then the statements that come before it and after it. This ensures that we can get our follows relations in O(1) time complexity. This table will be accessed when query evaluator calls Follows(s1,s2). The follows table will be computed in design extractor to allow Follows*(s1,s2) to be called.

Modifies table

It stores the statement number as a key, and a list of variables such that Modifies(stmt, var) holds. This table will be accessed when query evaluator calls Modifies(stmt, var).

Uses table

It stores the statement number as a key, and a list of variables such that Uses(stmt, var) holds. This table will be accessed when query evaluator calls Uses(stmt, var).

Pattern table

It stores the statement number as a key, and pair containing LHS variable and postfix expression in that statement number, such that pattern a(var,expression) holds. Postfix expressions are computed after the input in the form of infix expressions are parsed by the parser, before being inserted to the pattern table. This table will be accessed when query evaluator calls pattern a(var,expression). Postfix is used instead of infix expression in order to facilitate partial matching of the sub-expression. Below shows the steps of how the matching is processed.

1. Raw input in the form of infix sub-expression are passed to the function called hasPartialMatch(int stmt, var expression)
2. Within the function, the infix sub-expression is converted to postfix.
3. The postfix sub-expression that perform the partial match checking with the stored expression (postfix) that is retrieved from the pattern table.
4. If both expression matches, the function returns true.

This table will be accessed by the query evaluator when it calls pattern a(var, expression).

Design Extractor

The role of the Design Extractor is to compute the complex relations we have from the tables that the parser has populated. It would compute the Parent* relationships from Parent tables. Similarly, it will compute the Follows* relationships from our Follows table.

3.2.3 Query Processor

PQL Main: The SPA front-end passes queries to PQL Main one by one. PQL Main then passes the query to the Query Pre-Processor as well as managing the Query Tree by controlling the flow of the Validator and Evaluator.

Query Pre-Processor: The pre-processor contains the Validator and the Query Tree.

1. Query Validator: Checks the syntactic (characters used) and semantic validity (num-

ber of arguments, type of argument) of the query and passes the arguments to the Query Tree. If the query is invalid, control returns to PQL Main which passes in the next query.

We use the following schema in Query Validator.

(a) Tokenize the query by using **semi-colon ;** as delimiter to split it into declarations and Select query.

(b) Parse the variable declarations and check if the data types and synonyms are syntactically correct, then store them in the Query Tree.

(c) Parse **Select** by checking for overall syntax.

   i. If overall syntax of Select is valid, check the semantic validity of **Select synonym** by matching **synonym** to the synonyms stored in Query Tree.

   ii. If **Select synonym** is valid, then search for clauses; either **such that** clause and/or **pattern** clause.

   iii. If a **such that** clause and/or a **pattern** clause is found, then check the syntactic and semantic validity of the respective clause. Syntax is checked by regex, then the arguments are tokenized. The type of arguments and validity of arguments are verified by cross-checking with synonyms stored in Query Tree for declaration. If the clause is valid, then store clause in Query Tree for evaluator.

2. Query Tree: The Query Tree acts as storage for the Query Processor and is implemented as a **singleton**, as we need only one instance of it when we process a single query. The Validator calls the relevant Query Tree methods to store individual clauses in the Query Tree. It has several data structures to store different types of data.

   Query Tree X Query Validator

(a) To store synonyms for declarations, we use a vector of strings e.g. **vector<string>** for **stmt**, **var**, **assign**, **while**, **prog_line** and **const**.

(b) To store the clauses, we use arrays of strings, e.g. **array<string,2>** for **Select**, vector of array of strings, i.e **vector<array<strings,4>** for **such that** (use vector as there would be multiple clauses in future iterations) and **vector<array<string,6> >** for **pattern**.

Query Evaluator: The Evaluator accesses the information stored in the different arrays (and vector of arrays in case clauses are present) in the Query Tree and evaluates the query. The results are stored in a list of strings, i.e **list<string>**.
We use the following schema in Query Validator.

1. We have a boolean flag that is set to true in the beginning. At any point in the Evaluator, if any of the clauses return a false result, the flag is set to false and evaluation stops.

2. The Evaluator first accesses the **Select synonym** and gets results from PKB in a **list<strings>** object.

3. It checks whether any of the clauses(Follows(*),Parent(*),Modifies, Uses, pattern) in the Query Tree are empty. If not, it evaluates the clause and stores the result in a pair of lists of strings, i.e **pair<list<string>,list<string»**

4. The final result is consolidated by checking for common synonym and adding all the answers that match to the final result.

3.3 Component interactions

3.3.1 Query Processing

The PQL sequence diagram was drawn to understand how each query is processed, validated and evaluated to return the results. This led us to allocate equal manpower to every component as we realized that the coding was long and complex for each one, and we proceeded to code sequentially.

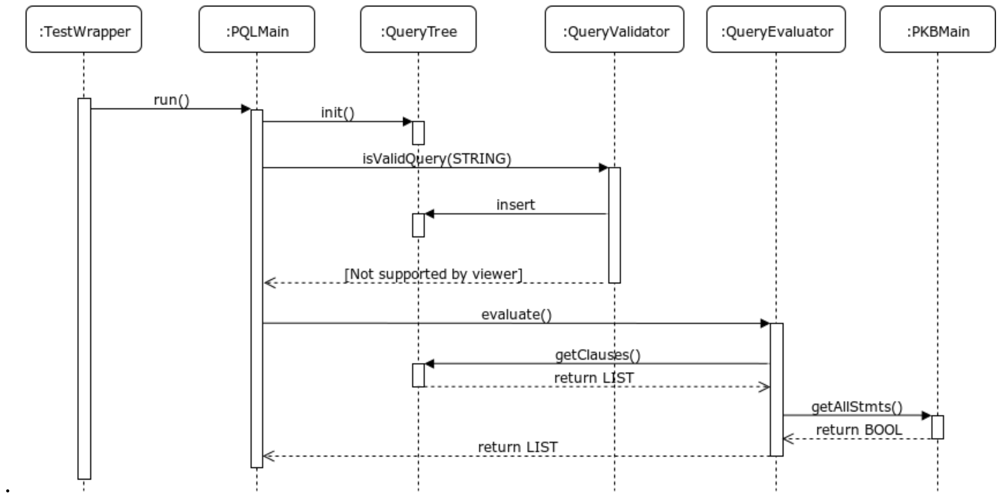

Figure 2: PQL Sequence Diagram

3.3.2 PKB table addition

The diagram below shows the interaction with the Parser. Let's say we have parsed a statement 2 which contains a = b, into statement number = 2, and variable = a upon invoking a function setModTableStmtToVar(2,"a") in the PKBMain. After that, the variable is mapped to varIdx = 5 from variable index table found within the PKB-component. The PKBMain then calls the function addModToAssignList(5,2) and cause the statement number to be appended to the modVarToAssignMap within the ModTableVar class. Assuming that statement number 2 is an assign statement, and has not been appended to the modVarToAssignMap yet, every function calls return true.

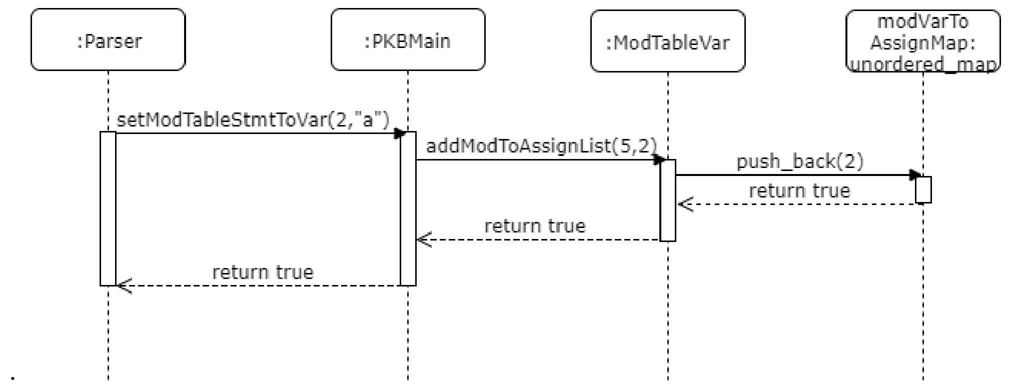

Figure 3: PKB Sequence Diagram

4 Documentation and Coding standards

We follow a combination of Coding Standards from different sources in order to get an optimal set of rules that help us the most in our project.

4.1 Class hierarchy

We store our SPA components in separate directories, divided into Parser, PKB and PQL. Our design abstractions are stored in separate source files (.cpp) and our concrete APIs are defined in the corresponding header files (.h).

4.2 Naming conventions

We follow the naming conventions detailed in this website **Geosoft C++ naming conventions**

1. Names representing types must be in mixed case starting with upper case.
   e.g QueryEvaluator, QueryValidator

2. Variable names must be in mixed case starting with lower case.
   e.g resultSuchThat, resultPattern

3. Names representing methods or functions must be verbs and written in mixed case starting with lower case.
   getAssigns(), insertSelect()

4.3 Formatting

We follow the formatting detailed in this website **Google style C++ formatting**

4.4 Abstract API and Concrete API

We made our Abtract API as similar as we could to concrete APIs in C++ to avoid confusion. At the same time, we tried to keep our Abstract API programming language independent and made it general enough to be implementable. Ex. Abstract API LIST-STMT was implemented with a concrete API of list<string>.

5 Testing

5.1 Plan

We have 3 types of tests - Unit Tests, Integration Tests and System Acceptance Tests. We performed regression testing using the AutoTester whenever new features were added to SPA. This enabled us to detect bugs and resolve them on time.

5.1.1 Unit Tests

We wrote frequent Unit Tests to test each component after it was completed. This is to make debugging more simple when it comes to integration testing and system acceptance testing. Examples of unit testing cases are shown below:

Unit Test for SPA Front-end Parser

Test Purpose: Test of tokenizing of string

Required Test Input: DummyString

Expected Test Results: True

```
        TEST_METHOD(tokenizeStringTest)
        {
            ParserChildForTest parser(dummyPkbMainPtr);
            std::string stringToTokenize = "one two three ; 123 { } a45=";
            std::vector<std::string> expectedTokens = { "one", "two", "three", ";", "123"
                , "{", "}", "a45", "=" };
            std::vector<std::string> actualTokens = parser.tokenizeString(
                stringToTokenize);
            Assert::IsTrue(actualTokens == expectedTokens);
        }

        TEST_METHOD(extractStringUpToSemicolonTest)
        {
            // Set up
            ParserChildForTest parser(dummyPkbMainPtr);
            Assert::IsTrue(createDummySimpleSourceFile_assignmentsOnly());
            Assert::IsTrue(parser.concatenateLines(dummySimpleSourcePath));

            // Testing begins
            for (int i = 0; i < 4; i++) {
```

TestParser.cpp

Unit Test for Query Processor

Test Purpose: Checking syntax of synonym

Required Test Input: Valid/Invalid synonyms

Expected Test Results: True/False

```
        TEST_METHOD(TestSynonymRegex)
        {
            QueryValidator qv;
            string str;

            //Valid Synonym Regex
            str = "a";
            Assert::IsTrue(qv.isValidSynonymTest(str));
            str = "nameWithoutNumAndHexSymbol";
            Assert::IsTrue(qv.isValidSynonymTest(str));
            str = "nameW1thNumb3rW1th0utH3xSymbol";
            Assert::IsTrue(qv.isValidSynonymTest(str));
            str = "nameWith#SymbolWithoutNumbers";
            Assert::IsTrue(qv.isValidSynonymTest(str));
            str = "nameW1thNumb3rAnd#Symbol";
            Assert::IsTrue(qv.isValidSynonymTest(str));
            str = "assign";
            Assert::IsTrue(qv.isValidSynonymTest(str));

            //Invalid Synonym Regex
            str = "0";
            Assert::IsFalse(qv.isValidSynonymTest(str));
            str = "#";
            Assert::IsFalse(qv.isValidSynonymTest(str));
            str = "n@meWithSpeci@lSymbols";
            Assert::IsFalse(qv.isValidSynonymTest(str));
            str = "1invalidEntityStartsWithNumber";
            Assert::IsFalse(qv.isValidSynonymTest(str));
            str = "#invalidEntityStartsWithSymbol";
            Assert::IsFalse(qv.isValidSynonymTest(str));
```

TestQueryValidator.cpp

Unit Test for PKB-Main

Test Purpose: Check PKB Relations

Required Test Input: Modifies Table

Expected Test Results: True

```
        TEST_METHOD(TestModifiesTable) {
            PKBMain PKB;
            Assert::IsTrue(PKB.setModTableStmtToVar(1, "a"));
            Assert::IsTrue(PKB.setModTableStmtToVar(1, "b"));
            Assert::IsTrue(PKB.setModTableStmtToVar(2, "a"));
            Assert::IsTrue(PKB.setModTableStmtToVar(4, "c"));
            Assert::IsTrue(PKB.setModTableStmtToVar(7, "d"));
```

TestPKBMain.cpp

We practice defensive programming using Assertions in the Parser to detect parts of the code that shouldn't be reached.

Test Purpose: Check assertion

Required Test Input: -

Expected Test Results: True

```
void Parser::processAndPopTopFollowStack()
{
    assert(!_stacksOfFollowsStacks.empty() && !_stacksOfFollowsStacks.top().empty());
    stack<int> topFollowsStack = _stacksOfFollowsStacks.top();

    int stmtAfter = topFollowsStack.top();
    topFollowsStack.pop();
    int stmtBefore = 0;
    while (!topFollowsStack.empty()) {
        stmtBefore = topFollowsStack.top();
        (*_pkbMainPtr).setFollowsRel(stmtBefore, stmtAfter);
        stmtAfter = stmtBefore;
        topFollowsStack.pop();
    }
    (*_pkbMainPtr).setFollowsRel(0, stmtAfter);

    // TODO: Free dynamically allocated memory with "delete" keyword.
    _stacksOfFollowsStacks.pop();
}
```

Parser.cpp

5.1.2 Integration Tests

We conducted integration tests to test workability of components with each other, e.g Query Tree tested with Query Validator, and Query Evaluator tested with SPA Front-end parser and PKB component. Like unit testing, we manually populate the PKB tables but for integration testing it is done by other components.

Test Purpose:

Required Test Input:

Expected Test Results: True

```
            /*-------------- Uses Clause --------------*/
            //Valid query
            qtInstance = qtInstance->clear(); qv = QueryValidator();
            str = "Select v such that Uses(1,v)";
            qtInstance->insertVariable("variable", "v");
            Assert::IsTrue(qv.isValidSelectTest(str));

            qtInstance = qtInstance->clear(); qv = QueryValidator();
            str = "Select v such that Uses(a,v)";
            qtInstance->insertVariable("assign", "a");
            qtInstance->insertVariable("variable", "v");
            Assert::IsTrue(qv.isValidSelectTest(str));
```

TestPQLValidatorAndTree.cpp

Test Purpose:

Required Test Input:

Expected Test Results: True

```
        TEST_METHOD(TestEvaluatorFollowsCase1Positive)
        {
            qe = QueryEvaluator();
            qtInstance = qtInstance->clear();
      PKBMain::resetInstance();
      pkbInstance = PKBMain::getInstance();
      pkbInstance->setFollowsRel(0, 1);
      pkbInstance->setFollowsRel(1, 2);
            //qe.setPkb(PKB);
            array<string, 4> arrToEvaluate = { "int", "1", "int", "2" };
            qe.evaluateFollowsTest(arrToEvaluate);
            Assert::IsTrue(qe.getHasResult());
        }
```

TestEvaluatorAndPkb.cpp

5.1.3 System Acceptance Tests

System acceptance tests are done by run using text files containing sample SIMPLE source codes and PQL queries through AutoTester. After that we compare the input and the output of the testing. Below shows some of the sample text files being fed into the AutoTester:

Sample SIMPLE Code

```
procedure ABC {
i = j;
j = 3 + 5;
while a
{
        a = 3;
   b = 2  ;
c = 4;
d = 5;
}
c = c + d;
}
```

Sample PQL Queries

Test Case 1: Check whether Modifies relationship works

```
1 - case 1 Modifies(int, ident)
stmt s;
```

```
Select s such that Modifies(1, "i")

1, 2, 3, 4, 5, 6, 7, 8

5000

2 - case 2 Modifies(int, _)

variable v;

Select s such that Modifies(3, _)

1, 2, 3, 4, 5, 6, 7, 8

5000

3 - case 3 Modifies(int, variable)

variable v;

Select v such that Modifies(8, v)

c

5000

4 - case 4 Modifies(synonym, ident)

assign a;

Select a such that Modifies(a, "c")

6, 8

5000

5 - case 5 Modifies(synonym, _)

assign a;

Select a such that Modifies(a, _)

1, 2, 4, 5, 6, 7, 8

5000

6 - case 6 Modifies(synonym, synonym)

assign a; variable v;

Select a such that Modifies(a, v)

1, 2, 4, 5, 6, 7, 8
```

```
5000

7 - invalid modifies

variable v;

Select v such that Modifies(10, v)

none

5000

8 - invalid modifies

stmt s;

Select s such that Modifies(1, 2)

none

5000

9 - invalid, synonym/var cannot start w number

stmt s;

Select s such that Modifies(4, "3")

none

5000

10 - invalid

stmt stmt;

Select stmt such that Modifies(stmt, "j")

none

5000

11 - invalid synonym

stmt 8;

Select 8 such that Modifies(5, "d")

none

5000
```

Test Case 2: Check whether Parents relationship works

```
1 - Case 1

stmt s;

Select s such that Parent(3, 4)

1, 2, 3, 4, 5, 6, 7, 8

5000

2 - Case 1

while w;

Select w such that Parent(3, 4)

3;

5000

3 - Case 1

assign a;

Select a such that Parent(3, 4)

1, 2, 4, 5, 6, 7, 8

5000

4 - Case 2

stmt s;

Select s such that Parent(3, _)

3

5000

5 - Case 2

while w;

Select w such that Parent(3, _)

3

5000

6 - Case 2

assign a;
```

```
Select a such that Parent(3, _)

1, 2, 4, 5, 6, 7, 8

5000

7 - Case 3

stmt s;

Select s such that Parent(3, s)

4, 5, 6, 7

5000

8 - Case 3

stmt s; assign a;

Select s such that Parent(3, a)

1, 2, 3, 4, 5, 6, 7, 8

5000

9 - Case 3

while w;

Select w such that Parent(3, w)

none

5000

10 - Case 4

stmt s;

Select s such that Parent(_, 5)

1, 2, 3, 4, 5, 6, 7, 8

5000

11 - Case 4

while w;

Select w such that Parent(_, 5)

3
```

```
5000

12 - Case 4

assign a;

Select a such that Parent(_, 5)

1, 2, 4, 5, 6, 7, 8

5000

13 - Case 5

stmt s;

Select s such that Parent(_, _)

1, 2, 3, 4, 5, 6, 7, 8

5000

14 - Case 5

assign a;

Select a such that Parent(_, _)

1, 2, 4, 5, 6, 7, 8

5000

15 - Case 5

while w;

Select w such that Parent(_, _)

3

5000

16 - Case 6

stmt s;

Select s such that Parent(_, s)

4, 5, 6, 7

5000

17 - Case 6
```

```
assign a;

Select a such that Parent(_, a)

4, 5, 6, 7

5000

18 - Case 6

while w;

Select w such that Parent(_, w)

none

5000

19 - Case 7

stmt s;

Select s such that Parent(s, 5)

3

5000

20 - Case 7

while w;

Select w such that Parent(w, 5)

3

5000

21 - Case 8

stmt s;

Select s such that Parent(s, _)

3

5000

22 - Case 8

while w;

Select w such that Parent(w, _)
```

```
3

5000

23 - Case 9

stmt s1, s2;

Select s1 such that Parent(s1, s2)

3

5000

24 - Case 9

stmt s1, s2;

Select s2 such that Parent(s1, s2)

4, 5, 6, 7

5000

25 - Case 9

stmt s; while w;

Select s such that Parent(s, w)

none

5000

26 - Case 9

stmt s; while w;

Select w such that Parent(s, w)

none

5000

27 - Case 9

while w; assign a;

Select w such that Parent(w, a)

3

5000
```

```
28 - Case 9
while w; assign a;
Select a such that Parent(w, a)
4, 5, 6, 7
5000
```

6 Discussion

6.1 Technical Difficulties

Since nobody in our team was familiar with C++, we spent a lot of time familiarizing ourselves with the language; however, it didn't help much when we actually started coding. We were also unfamiliar with Visual Studio 2015, and installing it and running it on our personal devices took an inordinate amount of time, especially as 3 members of our team use MacBooks, so additional resources (e.g purchasing Hard Drive, software for virtualization) were needed.

6.2 Project Management

For the PKB side, we have drafted out a plan to structure the PKB component, and discovering the APIs along the way. We also adjust the APIs accordingly depending on what the query evaluator is requesting for. Also, we discussed the most efficient algorithms and data structures for the PKB before starting with the implementation.
On the PQL side, we never divided the components separately and we all coded collaboratively for sub-components on websites like codeshare.io and pushed the code through one person's account. Although this resulted in everybody being familiar with how each component worked, we also fell behind and had to frequently rush to keep up with personal

deadlines.

6.3 Workload and Time Management

We never anticipated that this project would require so many hours and so much collaboration. Everybody in our team is overloading this semester, so cutting out time for our project despite setting personal deadlines turned out to be a problem. Sometimes we also spent an inordinate amount of time discussing before the actual implementation, which slowed us down.

7 Abstract APIs

7.1 PKB main

PKBMain

Overview: PKBMain class is act as a facade for SPA-Front End Parser and Query evaluator to invoke the method from this class. Some of the methods are included here.

+BOOLEAN isAssign(INTEGER stmtNumber)
Description: Returns true is statement number is in the assignment list. Otherwise, returns false.

+BOOLEAN isWhile(INTEGER stmtNumber)
Description: Returns true is statement number is in the while list. Otherwise, returns false.

+BOOLEAN isPresent (INTEGER stmtNumber)
Description: Returns true is statement number exists. Otherwise, return false.

+BOOLEAN isPresent(STRING variable)

Description: Check if the variable is present in the table

+VARIABLE_LIST getAllVariables(VOID)

Description: Returns a list of strings of all variables used in the program

+CONST_LIST getAllConstants(VOID)

Description: Returns a list of all constants used in the program

+BOOLEAN setModTableStmtToVar(INTEGER stmt, STRING variable)

Description: If statement number is not found in the table, insert it and its respective variables to the table. Otherwise, append the variable to the existing list. Returns true.

+BOOLEAN setUseTableStmtToVar(INTEGER stmt, STRING variable)

Description: If statement number is not found in the table, insert it and its respective variables to the table. Otherwise, append the variable to the existing list. Returns true.

+BOOLEAN setPatternRelation(INTEGER stmt, STRING variable, STRING expression)

Description: If statement number does not exist in key, insert it as a key and the statement number's respective left-hand side variable and right-hand side infix expression which are stored as a pair. Within this method, infix expression is converted to postfix expression before insertion.

+BOOLEAN isUses(INTEGER stmtNumber, STRING variable)

Description: Returns true if Uses(stmtNumber,variable) holds. Returns false otherwise.

+BOOLEAN isMod(INTEGER stmtNumber, STRING variable)

Description: Returns true if Modifies(stmtNumber,variable) holds. Returns false otherwise.

+BOOLEAN isUsingAnything(INTEGER stmtNumber)

Description: Returns true if Uses(stmtNumber,_) holds. Returns false otherwise.

+BOOLEAN isModifyingAnything(INTEGER stmtNumber)

Description: Returns true if Modifies(stmtNumber,_) holds. Returns false otherwise.

+BOOLEAN addToPatternTable(INTEGER stmtNumber, STRING variable, STRING expression)

Description: If statement number does not exist in key, insert it as a key and the statement number's respective left-hand side variable and right-hand side infix expression which are stored as a pair. Within this method, infix expression is converted to postfix expression before insertion.

+BOOLEAN getExpression(INTEGER stmtNumber)

Description: Returns its respective postfix expression.

+PAIRS_OF_STMT_AND_VAR_LIST getLeftVariables(VOID)

Description: Returns a pair of list of statements and the list of its respective variables, such that pattern a(v,_) holds

+PAIRS_OF_STMT_AND_VAR_LIST getLeftVariablesThatMatchWithString(STRING sub-expression)

Description: Returns a pair of list of statements and the list of its respective variables that partially matches with the expression, such that pattern a(v,_expression_) holds.

+STMT_LIST getPartialMatchStmt(STRING sub-expression)

Description: Returns a list of statements whose expression contains the sub-expression, such that pattern a(_,_expression_) holds.

+STMT_LIST getPartialBothMatches(STRING variable, STRING sub-expression)

Description: Returns a list of statements whose both variables and expression both matches with the input variable and sub-expressions respectively, such that pattern a(variable,_expression_) holds.

7.1.1 Variable Table

VarIdxTable

Overview: VarIdxTable is used to store the indexes of every variable that appeared in the program

+VAR_INDEX getIdxFromVar(STRING variable)

Description: Returns an integer index from the variable index table

+VAR_INDEX getIdxFromVar(STRING variable)

Description: Returns an integer index from the variable index table

+VAR_INDEX getIdxFromVar(STRING variable)

Description: Returns an integer index from the variable index table

+BOOLEAN isVarPresent(STRING variable)

Description: Check if the variable is present in the table

+VARIABLE_LIST getAllVariables(VOID)

Description: Returns a list of strings of all variables used in the program

+VARIABLE_LIST getUsesFromStmt(INTEGER stmtNumber)

Description: Returns a list of variables such that Uses(stmtNumber, variable) holds.

+STMT_LIST getStmtThatUsesAnything(STRING type)

Description: Returns a list of statement numbers such that Uses(stmtNumber,_) holds.

+PAIRS_OF_STMT_AND_VAR_LIST getUsesPairs(STRING type)

Description: Returns a pair of list of statement numbers and its corresponding list of variable such that Uses(stmtNumber, variable) holds.

+VARIABLE_LIST getModFromStmt(INTEGER stmtNumber)

Description: Returns a list of variables such that Modifies(stmtNumber, variable) holds.

+STMT_LIST getStmtThatModifiesAnything(STRING type)

Description: Returns a list of statement numbers such that Modifies(stmtNumber,_) holds.

+PAIRS_OF_STMT_AND_VAR_LIST getModPairs(STRING type)

Description: Returns a pair of list of statement numbers and its corresponding list of variable such that Modifies(stmtNumber, variable) holds.

7.1.2 Statement Type Table

StmtTypeTable

+BOOLEAN addToAssignStmtList(INTEGER stmtNumber)

Description: If statement number does not exist in the assignment list, insert it to the list and return true. Otherwise, the function returns false.

+BOOLEAN addToWhileStmtList(INTEGER stmtNumber)

Description: If statement number does not exist in the while list, insert it to the list and return true. Otherwise, the function returns false.

+BOOLEAN isAssignStmt(INTEGER stmtNumber)

Description: Returns true is statement number is in the assignment list. Otherwise, returns false.

+BOOLEAN isWhileStmt(INTEGER stmtNumber)

Description: Returns true is statement number is in the while list. Otherwise, returns false.

+BOOLEAN isPresent (INTEGER stmtNumber)

Description: Returns true is statement number exists. Otherwise, return false.

+STMT_LIST getAssignStmtList();

Description: Returns the list of assignment statement numbers.

+STMT_LIST getWhileStmtList();

Description: Returns the list of while statement numbers.

+STMT_LIST getStmtType(STMT_LIST stmtList, STRING type)

Description: Filters away the unwanted statements and return the list of statements with the desired statement type.

+PAIRS_OF_STMT_AND_VAR_LIST getStmtType(PAIRS_OF_STMT_AND_VAR_LIST stmtList, STRING type)

Description: Filters away the unwanted statements and its respective variables. Return the pairs of list of statements with the desired statement type and list of its respective variables.

7.1.3 Uses Table

UsesTableStmtToVar

Overview: To store the uses relationship between the statement and the variables

+BOOLEAN addUsesStmtToList(INTEGER stmtNumber, STRING variable)

Description: If statement number is not found in the table, insert it and its respective variables to the table. Otherwise, append the variable to the existing list. Returns true.

+BOOLEAN isUses(INTEGER stmtNumber, STRING variable)

Description: Returns true if Uses(stmtNumber,variable) holds. Returns false otherwise.

+BOOLEAN isUsingAnything(INTEGER stmtNumber)

Description: Returns true if Uses(stmtNumber,_) holds. Returns false otherwise.

+VARIABLE_LIST getUsesVariablesFromStmt(INTEGER stmtNumber)

Description: Returns a list of variables such that Uses(stmtNumber, variable) holds.

+STMT_LIST getStmtThatUses(VOID)

Description: Returns a list of statement numbers such that Uses(stmtNumber,_) holds.

+PAIRS_OF_STMT_AND_VAR_LIST getUsesPair(VOID)

Description: Returns a pair of list of statement numbers and its corresponding list of variable such that Uses(stmtNumber, variable) holds.

7.1.4 Modifies Table

ModTableStmtToVar

Overview: To store the modifies relationship between the statement and the variables

+BOOLEAN addModStmtToList(INTEGER stmtNumber, STRING variable)

Description: If statement number is not found in the table, insert it and its respective variables to the table. Otherwise, append the variable to the existing list. Returns true.

+BOOLEAN isMod(INTEGER stmtNumber, STRING variable)

Description: Returns true if Modifies(stmtNumber,variable) holds. Returns false otherwise.

+BOOLEAN isModifyingAnything(INTEGER stmtNumber)

Description: Returns true if Modifies(stmtNumber,_) holds. Returns false otherwise.

+VARIABLE_LIST getModVariablesFromStmt(INTEGER stmtNumber)

Description: Returns a list of variables such that Modifies(stmtNumber, variable) holds.

+STMT_LIST getStmtThatModifies(VOID)

Description: Returns a list of statement numbers such that Modifies(stmtNumber,_) holds.

+PAIRS_OF_STMT_AND_VAR_LIST getModPair(VOID)
Description: Returns a pair of list of statement numbers and its corresponding list of variable such that Modifies(stmtNumber, variable) holds.

7.1.5 Pattern Table

PatternTable

Overview: To store the pattern relationship between the statement (key) and the pair containing left-hand side variable and right-hand side postfix expression

+ BOOLEAN addToPatternTable(INTEGER stmtNumber, STRING variable, STRING expression)
Description: If statement number does not exist in key, insert it as a key and the statement number's respective left-hand side variable and right-hand side infix expression which are stored as a pair. Within this method, infix expression is converted to postfix expression before insertion.

+ BOOLEAN getExpression(INTEGER stmtNumber)
Description: Returns its respective postfix expression.

+ PAIRS_OF_STMT_AND_VAR_LIST getLeftVariables(VOID)
Description: Returns a pair of list of statements and the list of its respective variables, such that pattern a(v,_) holds

+ PAIRS_OF_STMT_AND_VAR_LIST getLeftVariablesThatMatchWith-String(STRING sub-expression)

Description: Returns a pair of list of statements and the list of its respective variables that partially matches with the expression, such that pattern a(v,_expression_) holds.

+ STMT_LIST getPartialMatchStmt(STRING sub-expression)

Description: Returns a list of statements whose expression contains the sub-expression, such that pattern a(_,_expression_) holds.

+ STMT_LIST getPartialBothMatches(STRING variable, STRING sub-expression)

Description: Returns a list of statements whose both variables and expression both matches with the input variable and sub-expressions respectively, such that pattern a(variable,_expression_) holds.

7.1.6 Parents Table

<u>ParentToChildTable</u>

Overview: To store the relationship between the parent (key) and the child (value)

+BOOLEAN addParentChild(INTEGER parentStmt, INTEGER childStmt)

Description: If parentStmt number is not found in the table, insert it and its childStmt to the table. Otherwise, append the variable to the existing list. Returns true.

+LIST_OF_STMTS getChildren(INTEGER parentStmt)

Description: Returns childStmt from parentToChildMap

+BOOLEAN isParent(INTEGER parentStmt)

Description: Returns true if the integer passed into the parameter has child statement. Otherwise, return false.

+BOOLEAN isParentChild(INTEGER parentStmt, INTEGER childStmt)
Description: Returns true if Parent(parentStmt,childStmt) holds. Otherwise, return false.

+LIST_OF_STMTS getChildren(INTEGER parentStmt)
Description: Returns a list of children statements related to parentStmt

+LIST_OF_STMTS getAllParents()
Description: Returns a list of parent statements.

+PAIR_OF_LISTS_OF_STMTS getAllParentsRel()
Description: Returns a pairs of two lists such that Parents(stmt1,stmt2) holds.

ParentStarTable

Overview: To store the parents star relationship between the parent(key) and child(value)

+BOOLEAN addParentChild(INTEGER parentStmt, INTEGER childStmt)
Description: If parentStmt number is not found in the table, insert it and its childStmt to the table. Otherwise, append the variable to the existing list. Returns true.

+LIST_OF_STMTS getChildren(INTEGER parentStmt)
Description: Returns grandchildren statements from parentToChildStarMap

+BOOLEAN isParentStarChild(INTEGER parentStmt, INTEGER childStmt)
Description: Returns true if Parent*(parentStmt,childStmt) holds. Otherwise, return false.

+PAIR_OF_LISTS_OF_STMTS getAllParentStarRel()

Description: Returns a pairs of two lists such that Parents*(stmt1,stmt2) holds.

7.1.7 Follows Table

FollowsTable

Overview: To store the follow relationship between the statement (key) and the pair containing statement numbers that comes before(first) and after(second).

+BOOLEAN addFollows(INTEGER stmt1, INTEGER stmt2, INTEGER stmt3)

Description: Insert relation with stmt2 being the statment before and stmt3 being after.

+INTEGER getStmtBef(INTEGER stmt)

Description: Returns statement that comes before the inputted statement

+INTEGER getStmtAfter(INTEGER stmt)

Description: Returns statement that comes after the inputted statement

+INTEGER hasStmtBefore(INTEGER stmt)

Description: Returns true if there exists a statement that comes before the inputted statement. Otherwise returns false.

+INTEGER hasStmtAfter(INTEGER stmt)

Description: Returns true if there exists a statement that comes after the inputted statement. Otherwise returns false.

+BOOLEAN isFollows(INTEGER stmt1, INTEGER stmt2)

Description: Returns true if Follows(stmt1, stmt2) holds.

+LIST_OF_STMTS getAllBefore()

Description: Returns a list of all statements that is found in the first part of the pair.

+LIST_OF_STMTS getAllAfter()

Description: Returns a list of all statements that is found in the second part of the pair.

+PAIR_OF_LISTS_OF_STMTS getAllFollows()

Description: Returns a pairs of two lists such that Parents(stmt1,stmt2) holds.

<u>FollowStarBeforeTable</u>

Overview: To store the follow star relationship between the statement (key) and the one that comes before it (value)

+LIST_OF_STMTS getBeforeStar(INTEGER stmt)

Description: Returns the list of all statement that comes before the inputted statement number.

+BOOLEAN isBeforeStar(INTEGER stmt)

Description: Returns true if inputted statement comes before the statement found in the key. Otherwise return false.

+LIST_OF_STMTS getAllBeforeStar(INTEGER stmt)

Description: Returns a list of statements that is found in the hashmap values.

FollowStarAfterTable

Overview: To store the follow star relationship between the statement (key) and the one that comes after it (value)

+LIST_OF_STMTS getAfterStar(INTEGER stmt)
Description: Returns the list of all statement that comes after the inputted statement number.

+BOOLEAN isAfterStar(INTEGER stmt)
Description: Returns true if inputted statement comes after the statement found in the key. Otherwise return false.

+LIST_OF_STMTS getAllAfterStar(INTEGER stmt)
Description: Returns a list of statements that is found in the hashmap values.