8 September, 2014



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

Team Name: Team 4

Team Members Information:

Group PKB

Saloni Kaur A0084053L [a0084053@nus.edu.sg](mailto:a0084053@nus.edu.sg)

M I Azima A0085594N [a0085594@nus.edu.sg](mailto:a0085594@nus.edu.sg)

Group PQL

Saima Mahmood A0084176Y [a0084176@nus.edu.sg](mailto:a0084176@nus.edu.sg)

Nguyen Trong Son A0088441 [A0088441@nus.edu.sg](mailto:A0088441@nus.edu.sg)

Vu Phuc Tho A0090585X [A0090585@nus.edu.sg](mailto:A0090585@nus.edu.sg)

CS3202 Software Engineering Project

Iteration 1 Report

# Iteration Overview

## Scope of SPA Implemented

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

## Achievements & Problems

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

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

# Project Plans

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

## 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 : Whole Project Tasks Breakdown

## Plan for Iteration 1

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

Table : Iteration 1 Work Distribution

# UML Diagrams

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

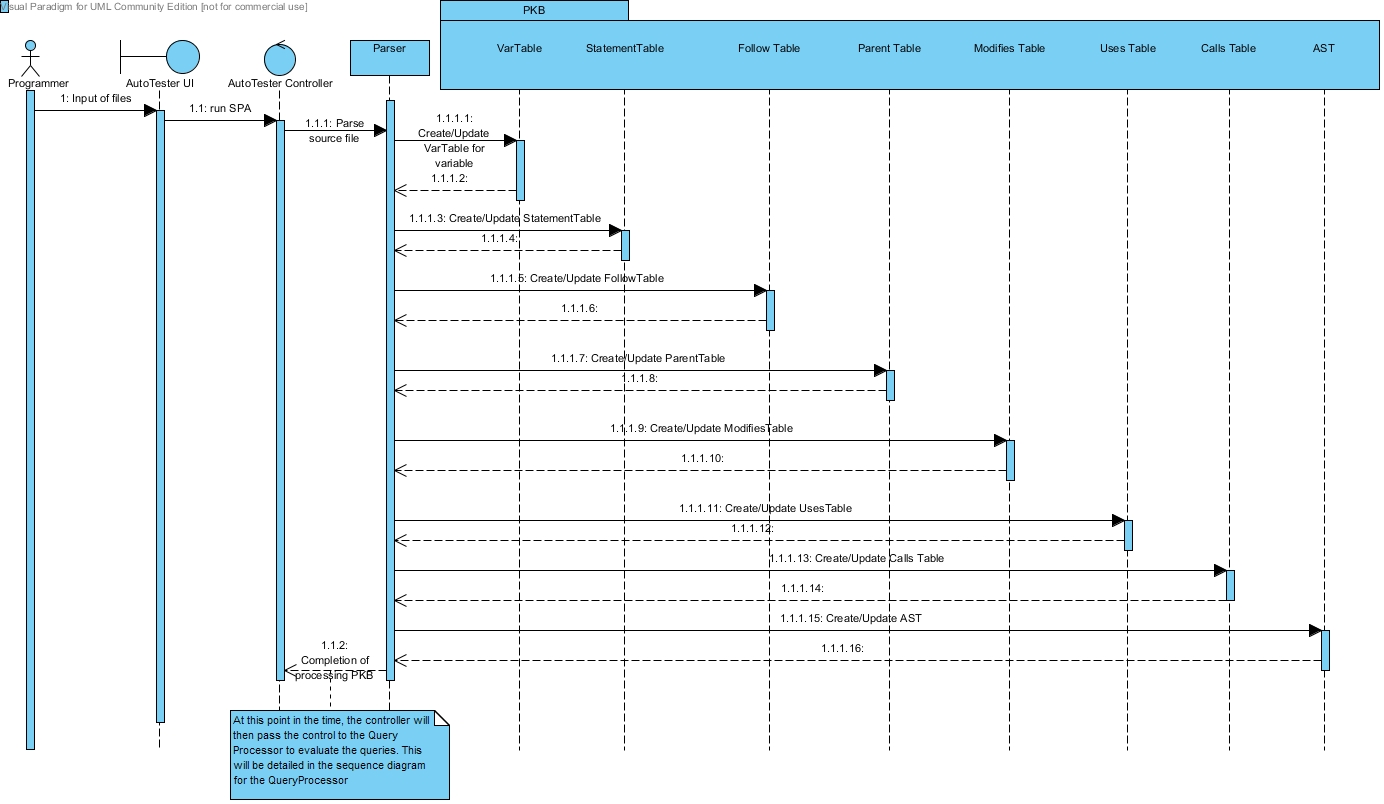


Figure 1: Sequence Diagram for Processing PKB

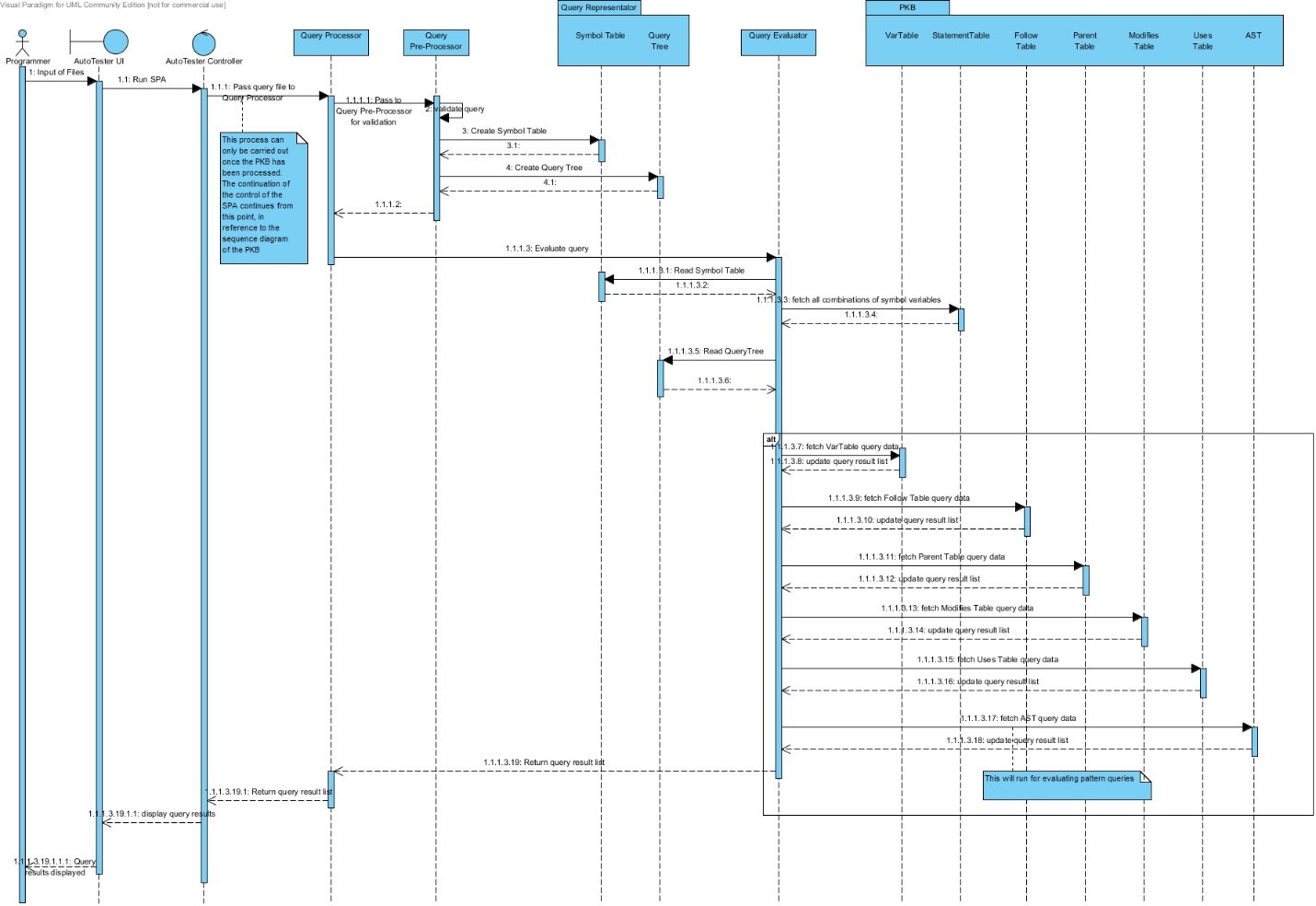


Figure 2: Sequence Diagram for Query Processor Flow

# Design Decisions

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

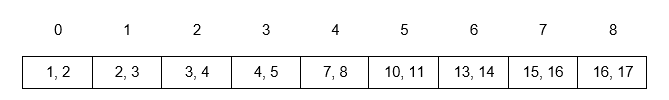


Figure : Previous Representation of Follow Table

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

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

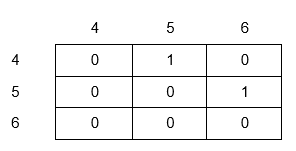


Figure : Updated Representation of Follow Table

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

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

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

# Coding Standards & Experiences

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

## Naming Conventions

### 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.

### 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.

# Query Processing

## Query Validation

Query validating process is handled by QueryPreprocessor (QPP). QPP will read and validate data from query file, and store necessary information for query evaluating in QueryRepresentator.

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

* If finding line containing declaration (e.g. stmt s; ), QPP will send this part to checkDeclaration() method.
* If finding line containing query part (e.g. Select s such that Follows(s, 1)), QPP will send this part to checkQueryPart() method.
* While checkQueryPart() method runs, if QPP finds new clause of query, it will call the corresponding method checkClause() (e.g. checkSuchThatCondtion() ) for this clause. Currently, we provide method for “such that” clause and “pattern” clause, and still work on “with” clause.

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

## Query Evaluation

Query evaluating process is carried out by QueryEvaluator (QE). Using the stored data in QueryRepresentator and PKB, QE will try to find values that satisfy the query’s conditions, and send the list of result values to the UI (i.e. AutoTester).

During evaluating a query, firstly, QE will create a list of values for all parameters declared in that query. After that, it will check all 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, QE will hang up the process and return an empty list for result of the query.

After the process finishes with all condition being satisfied, QE will take result value and save it to the result list.

# Testing

## 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.

## Examples of Test Cases

### Unit Testing

#### PKB

Sample 1

Test Purpose:

Required Test Inputs:

Expected Test Results:

Other Requirements:

Sample 2

Test Purpose:

Required Test Inputs:

Expected Test Results:

Other Requirements:

#### PQL

Sample 1

Test Purpose:

Required Test Inputs:

Expected Test Results:

Other Requirements:

Sample 2

Test Purpose:

Required Test Inputs:

Expected Test Results:

Other Requirements:

### Integration Testing

#### Parser & PKB

Test Purpose:

Required Test Inputs:

Expected Test Results:

Other Requirements:

#### PKB & QP

Test Purpose:

Required Test Inputs:

Expected Test Results:

Other Requirements:

### Validation Testing

Test Purpose:

Required Test Inputs:

Expected Test Results:

Other Requirements:

# Discussion

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

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

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

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

# Appendix A: Abstract PKB API

## 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) |

## 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 |

## 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 |

## Modify API

|  |
| --- |
| Modify |
| Overview:   1. 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. 2. 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). 3. 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. |

## 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. |

## 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) |

## Calls API

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