|  |
| --- |
| CS3201 |
| SPA Project Report |
| Team 02 |

|  |
| --- |
|  |

Table of Contents

**Type chapter title (level 1)1**

Type chapter title (level 2)2

Type chapter title (level 3)3

**Type chapter title (level 1)4**

Type chapter title (level 2)5

Type chapter title (level 3)6

1. Team Members

Adinda Ayu Savitri (A0099214B)

Hisyam Nursaid Indrakesuma(A0098139R)

Ipsita Mohapatra (A0101286N)

2. Development Plan

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Team Member | | Mini-Iteration 1 | | | | | | | |
| Implement CodeParser to parse SIMPLE program with simple assignment statements. Unit Testing. | Create AST while parsing SIMPLE program. | Implement VarTable, Follows in PKB. Unit Testing. | Implement TypeTable, Parent in PKB. Set up PKB as singleton. Unit Testing. | Implement Query Evaluator for Follows, Parent. Unit Testing. | Make Query and Relationship objects. Implement query parser for such-that clause. Unit Testing. | Implement Query Processor to integrate Query Parser and Query Evaluator. | Write system test cases for Mini-Iteration 1. |
| PKB | Kester | ✓ | ✓ |  |  |  |  |  |  |
| Yohanes |  |  |  | ✓ |  |  |  |  |
| Ipsita |  |  | ✓ |  |  |  |  | ✓ |
| PQL | Lacie |  |  |  |  | ✓ |  |  |  |
| Hisyam |  |  |  |  |  |  | ✓ |  |
| Adinda |  |  |  |  |  | ✓ |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Team Member | | Mini-Iteration 2 | | | | | | | |
| Implement complete CodeParser. Integrate with VarTable, TypeTable, Follows, Parent relationships. | Write system test cases for Mini-Iteration 1 and 2. | Fix PKB errors for previous iteration. | Implement Query Evaluator for Follows Star, Parent Star. Unit Testing. | Integration Testing of CodeParser with PKB. | Integration Testing of CodeParser and PKB. | Integrate with AutoTester. Start System testing. | Validate query for such-that clause.  Implement Query Parser for pattern clause. Unit Testing. |
| PKB | Kester | ✓ |  |  |  | ✓ |  |  |  |
| Yohanes |  |  | ✓ |  |  |  |  |  |
| Ipsita |  | ✓ |  |  |  |  | ✓ |  |
| PQL | Lacie |  |  |  | ✓ |  | ✓ |  |  |
| Hisyam |  |  |  |  |  |  | ✓ |  |
| Adinda |  |  |  |  |  |  |  | ✓ |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Team Member | | Mini-Iteration 3 | | | | | |
| Implement Modifies in PKB. Unit Testing. | Implement Uses in PKB. Unit Testing. | Write test cases for system testing of Mini-Iteration 3. Do system testing for Modifies, Uses and pattern and a mixture. | Implement Query Evaluator for Modifies and Uses. Unit Testing. | Implement Query Evaluator for pattern. Unit Testing. | Implement complete Query Parser for complex queries. Unit Testing. |
| PKB | Kester |  |  |  |  |  |  |
| Yohanes |  | ✓ | ✓ |  |  |  |
| Ipsita | ✓ |  | ✓ |  |  |  |
| PQL | Lacie |  |  |  | ✓ |  |  |
| Hisyam |  |  | ✓ |  | ✓ |  |
| Adinda |  |  |  |  |  | ✓ |

3. The Prototype

# 3.1 Architecture

# 3.2 Design Decisions

# 3.3 Interaction (Uml Diagrams)

4. Components

# 4.1 Parser

# 4.2 Pkb

# 4.3 Query Processor

Query processor consists of three parts: query processor (controller), query parser, and query evaluator.

**4.3.1 Query Processor**

**4.3.2 Query Parser**

Query parser has two major functionalities: query validation and query parsing, and they are implemented as functions in the QueryParser class. The controller calls query validator to check if the given query is syntactically correct. If it is, query controller will then parse the query by calling the query parser.

**4.3.2.1 Query Validation**

Query validation is done using predictive parsing method. As an example, consider this valid query

assign a; while w; Select a such that Follows(w, a) pattern a (“x”, \_”x+y”\_)

Query validator will first break down the query into statements, separated by semicolon. The query above will be broken down into three statements:

1. assign a
2. while w
3. Select a such that Follows(w, a) pattern a (“x”, \_”x+y”\_)

The validator will then put these statements into a static vector in the QueryParser class. The validator will iterate through the vector and check the validity of each statement separately, whether it is a declaration statement or a select statement.

Valid declaration statements will be concatenated into a static string, to be used later by the parsing function. Select statement will be broken down into such-that and/or pattern clauses, whose parameters will be checked against the grammar rule.

Consider statement number 3 from the example above. Query validator will break the statement into three parts:

1. Select a
2. such that Follows(w, a)
3. pattern a (“x”, \_”x+y”\_)

The such-that clause from part number 2, will be further broken down into tokens, while validating whether each token matches the expected value from the grammar rule:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **tokens[]** | | | | | | | |
| such | that | Follows | ( | w | , | a | ) |

Similarly, the pattern clause from part number 3 will be processed into these tokens:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **tokens[]** | | | | | | |
| pattern | a | ( | “x” | , | \_”x+y”\_ | ) |

If any of the token does not match the expected value, the validator will instantly terminate and declare the query invalid. In the case where all the clauses are valid, all the tokens from all the statements will be concatenated one after another into one static vector, *selectStatement*. This vector is the one that will be accessed by the parsing function later on. For efficiency, the *selectStatement* vector will only contain relevant tokens from the statement. Therefore, the unnecessary syntactic punctuation will be removed.

From the example above, the value of *selectStatement* will be:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **selectStatement[]** | | | | | | | | | |
| Select | a | such | that | Follows | w | a | pattern | x | \_”x+y”\_ |

**4.3.2.2 Query Parsing**

The parser processes two objects from the earlier validation which are the declaration string and select-statement vector. The declaration string will be converted into a table that maps each synonym with its type.

As an example, the declaration string from the query above, “assign a while w”, will be tokenized again to produce the following synonym table:

|  |  |
| --- | --- |
| **Synonym** | **Type** |
| a | ASSIGN |
| w | WHILE |
| BOOLEAN | BOOLEAN |

Table 1: Synonym table

This enables easy look up when the query evaluator evaluates the query. Note that the BOOLEAN type always exists in the synonym table because user can use “BOOLEAN” in his select statement without any declaration.

The select-statement vector will be processed to construct the following Query object.

|  |
| --- |
| **Query** |
| *string* **selected-synonym**  *vector<Relationship>* **relationships**  *map* **synonym-table** |

The selected synonym, in this example is ‘a’, will be stored inside a string in the Query object. The synonym table that was created earlier will also be included in the Query object. Both the such-that and pattern clauses will be stored as another object, Relationship, as the following.

|  |
| --- |
| **Relationship** |
| *enum* **relationship-type**  *string* **argument-1**  *string* **argument-2** |

Since each query can contain many select clauses, these clauses are stored inside a vector for scalability purposes. From the example above, the select-statement vector will be processed to produce the following.

|  |  |
| --- | --- |
| **Query** | |
| **selected-synonym** | a |
| **relationships** | [rel1, rel2] |
| **synonym-table** | map1 |

|  |  |
| --- | --- |
| **rel1** | |
| **relationship-type** | FOLLOWS |
| **argument-1** | w |
| **argument-2** | a |

|  |  |
| --- | --- |
| **rel2** | |
| **relationship-type** | PATTERN |
| **argument-1** | “x” |
| **argument-2** | \_”x+y”\_ |

|  |  |
| --- | --- |
| **map1** | |
| **Synonym** | **Type** |
| a | ASSIGN |
| w | WHILE |
| BOOLEAN | BOOLEAN |

When the controller calls the parsing function, the function will return a query object. This object will then be passed to query evaluator.

**4.3.3 Query Evaluator**

5. Testing

# 5.1 Testing Plan

|  |  |  |  |
| --- | --- | --- | --- |
| **Tasks** | **Start Date** | **Duration (days)** | **End Date** |
| CodeParser Unit Testing | 20-Mar | 13 | 2-Apr |
| VarTable Unit Testing | 20-Mar | 8 | 28-Mar |
| TypeTable Unit Testing | 20-Mar | 8 | 28-Mar |
| Parent Unit Testing | 20-Mar | 8 | 28-Mar |
| Follows Unit Testing | 20-Mar | 8 | 28-Mar |
| Modifies Unit Testing | 7-Apr | 8 | 15-Apr |
| Uses Unit Testing | 7-Apr | 8 | 15-Apr |
| Query Evaluator Unit Testing | 20-Mar | 26 | 15-Apr |
| Query Parser Unit Testing | 20-Mar | 26 | 15-Apr |
| CodeParser and PKB Integration Testing | 29-Mar | 8 | 6-Apr |
| PKB and Query Processor Integration Testing | 29-Mar | 17 | 15-Apr |
| System Testing of Parent and Follows | 29-Mar | 8 | 6-Apr |
| System Testing of Parent\* and Follows\* | 7-Apr | 8 | 15-Apr |
| System Testing of Modifies, Uses and pattern | 7-Apr | 8 | 15-Apr |
| System Testing of meaningless/complex queries | 14-Apr | 3 | 17-Apr |

We did testing on 3 different levels, namely unit testing (using CPPUnit), integration testing (using CPPUnit) and system testing (using AutoTester). Unit Testing was done while coding the components, while integration testing was done between SIMPLE program parser and PKB and between PKB and Query component.

From the testing experience in this project, we realised the need for timely and consistent unit, integration and system testing. By testing individual components early, we detect bugs earlier in the project’s lifetime, thus, saving us time towards the end of the project. We also did regression testing by reusing our unit tests and system tests. This helped us to quickly identify bugs that could have been introduced while we were trying to solve other bugs.

# 5.2 Unit Testing

Unit Testing was done on every sub-component of the SPA.

For the Front-End, some examples would be the TestNode.cpp, which is used to unit test our ASTNode object, and the TestParser.cpp, which is used to unit test all source code parsing methods.

For the Query Processor, we have the QueryEvaluatorTest.cpp, which is used to unit test all evaluation after Query Pre-Processing, and the QueryParserTest.cpp, which is used to unit test methods involved in parsing the queries into QueryTree objects.

For the PKB, every single implemented relationship(Parent, Follows, Uses, and Modifies) has a UnitTest specific to the relationship.

# 5.3 Integration Testing

# 5.4 System Testing

All system testing was executed by running test cases using the AutoTester. There are five source codes for parsing, and three query sets associated to every source code. Each source code is associated to a certain mini iteration, and each query set is associated to a different part of the iteration.

6. Coding Standards

7. API

## 1 Parser

|  |
| --- |
| **Parser**  *Overview*: Parser is responsible to read the source code, creates AST, and set the tables accordingly |
| **API:** |
| **VOID parseDriver(FILENAME fileName, PKB\_PTR pkb);**  Parameter:   * fileName – filename of the source code. (e.g. “source1.txt”) * pkb – is a pointer to a pkb class, which contains all the tables and data needed to answer queries.   (e.g. VARTABLE,PROCTABLE,FOLLOWS,MODIFIES,PARENT,USES,AST, etc)  Description: parser reads in the source code, tokenizes and detects the structure of the code, then create AST, and set the tables accordingly. |

## 2 Node (Ast)

|  |
| --- |
| **Node**  *Overview*: Node is the node structure of AST nodes. |
| **API:** |
| **NODE\_PTR getChild(INDEX i);**  Description: returns a pointer to a node which is the i th children of the current node. First children is index 0. |
| **NODE\_PTR\_LIST getChild();**  Description: returns a LIST of Node pointers of the current node’s children. |
| **NODE\_PTR getParent ();**  Description: returns a pointer to a node which is the parent of the current node. |
| **TYPE getType ();**  Description: returns the statement type of the current node. |
| **DATA getData ();**  Description: returns the Data of the current node. |
| **INDEX getProgLine();**  Description: returns the program line of the current node. |
| **VOID setChild(NODE\_PTR newChild);**  Description: set the next children of the current node to be the node pointed by newChild. |
| **VOID setData(DATA newData);**  Description: set the DATA of the current node to be newData. |
| **VOID setType(TYPE newType);**  Description: set the TYPE of the current node to be newType. |
| **VOID setParent(NODE\_PTR newParent);**  Description: set the next parent of the current node to be the node pointed by newParent. |
| **VOID setProgLine(INDEX newProgLine);**  Description: set the INDEX progLine of the current node to be newProgLine. |
| **VOID printPreOrderExpressionTree(NODE\_PTR root);**  Description: print the details of every node from root, using pre-order traversal. |
| **VOID stringPreOrderExpressionTree(NODE\_PTR root, STRING\_REF word);**  Description: retrieve the details of every node from root, using pre-order traversal, and store it in word, which is a STRING passed by reference. |

## 3 Vartable

|  |
| --- |
| **VarTable**  *Overview*: VarTable stores all the variables from a source program. |
| **API:** |
| **INDEX insertVar (VARNAME varName);**  Parameters:  varName - name of the variable  Description:  If varName is not in the VarTable, inserts varName into the  VarTable and returns its index. Otherwise, return its index and the table remains unchanged. |
| **VARNAME getVarName (INDEX index);**  Parameters:  index - index of the variable  Description: Returns the name of a variable at VarTable [index]  If ‘index’ is out of range, return error code |
| **INDEX getVarIndex (VARNAME varName);**  Parameters:  varName - name of the variable  Description: If varName is in VarTable, returns its index. Otherwise, returns error code |
| **INDEX getNumVar() ;**    Description: Returns the total number of unique variables stored in the VarTable. |

## 4 Proctable

|  |
| --- |
| **ProcTable**  *Overview*: ProcTable stores all the procedure names from a source program. |
| **API:** |
| **INDEX insertProc (PROCNAME procName);**  Parameters:  procName - name of the procedure to be inserted into ProcTable  Description: If procName is not in the ProcTable, inserts procName into the  ProcTable and returns its index. if procName already exists, return its index and the table remains unchanged. |
| **INDEX getProcIndex (PROCNAME procName);**  Parameters:  procName - procedure name  Description: If procName has a corresponding index in the ProcTable, returns its index. Otherwise, returns error code. |
| **PROCNAME getProcName (INDEX index);**  Parameters:  index - index of the procedure  Description: If the procedure denoted by the index ‘index’ exists in the ProcTable, returns the name of a procedure at ProcTable.  If ‘index’ is out of range, return error code |

## 5 TypeTable

|  |
| --- |
| **TypeTable**  *Overview*: TypeTable stores all the types (assign,calls, while,etc) for each program line, from a source program. |
| **API:** |
| **INDEX insertStmtNumAndType (STMTNUM s, TYPE t);**  Description: Associate statement s, with type t, store it in the typeTable, and returns its index. If the pair already exists, return its index and the table remains unchanged |
| **TYPE getType (STMTNUM s);**  Description: Search statement s in the typeTable and returns the type of statement s. if s is out of range or invalid, return error code. |
| **STMTNUM\_LIST getAllStmts (TYPE t);**  Description: retrieve all statements of type t, store that in a LIST, and return the LIST. |
| **BOOLEAN isType (TYPE t, STMTMNUM s);**  Description: If STMTNUM s, is associated with TYPE t, in the typeTable, returns true. If s is invalid of out of range, returns false. |

## 6 Follows

|  |
| --- |
| **Follows**  *Overview*: Follows is used to keep track of the Follows relationship between two statements (denoted by their statement numbers: STMT#). |
| **API:** |
| **VOID setFollows (STMTNUM s1, STMTNUM s2);**  Description: Set the Follows(s1, s2) in the FollowsTable.  If s1 or s2 are out of range, do nothing. |
| **BOOLEAN isFollows (STMTNUM s1, STMTNUM s2);**  Description: If the Follows(s1, s2) is true, return true. Otherwise, return false.  If s1 or s2 are out of range, return false; |
| **BOOLEAN isFollows (SYNTYPE t1, SYNTYPE t2);**  Description: If Follows(t1, t2) is true, return true, where t1 is the type of statement 1, and t2 is the type of statement 2. Otherwise, return false.  If t1 or t2 is of invalid type, return false; |
| **BOOLEAN isFollowedBy (SYNTYPE t, STMTNUM s2);**  Description: If getFollowedBy (t,s) returns a valid result, this method returns true.  If s2 is out of range or type t is invalid, return false; |
| **STMTNUM getFollows (SYNTYPE t, STMTNUM s);**  Description: Returns x such that Follows(s, x) holds. (and if x is of SYNTYPE t)  If s does not exist or is out of range, return error code. |
| **STMTNUM getFollowedBy (SYNTYPE t, STMTNUM s);**  Description: Returns x such that Follows(x, s) holds. (and if x is of SYNTYPE t)  If s does not exist or is out of range, return error code. |
| **STMTNUM\_LIST getFollows (SYNTYPE t1, SYNTYPE t2);**  Description: Returns all STMTNUM x such that for every x, Follows(x, t2) holds, and every x is of type t1.  If s does not exist or is out of range, return error code. |
| **STMTNUM\_LIST getFollowedBy (SYNTYPE t1, SYNTYPE t2);**  Description: Returns all STMTNUM x such that for every x, Follows(t1, x) holds, and every x is of type t2.  If s does not exist or is out of range, return error code. |

## 7 Follows\*

|  |
| --- |
| **FollowsStar**  *Overview*: FollowsStar is used to keep track of the Follows\* relationship between two statements. |
| **API:** |
| **BOOLEAN isFollowsStar (STMTNUM s1, STMTNUM s2);**  Description: If Follows\*(s1, s2) is true, returns true. Else, returns false.  If s1 or s2 are out of range, return error code |
| **STMTNUM\_LIST getFollowsStar (STMTNUM s);**  Description: Returns a list containing statements x such that Follows\*(s, x) holds.  If s is out of range r does not exist, return a list with -1 as the only element (error code) |
| **STMTNUM\_LIST getFollowedStarBy (STMTNUM s);**    Description: Returns a list containing statements x such that Follows\*(x, s) holds.  If s is out of range r does not exist, return a list with -1 as the only element (special value). |

## 8 Parent

|  |
| --- |
| **Parent**  *Overview*: Parent is used to keep track of the Parent relationship between two statements. |
| **API:** |
| **VOID setParent (STMTNUM s1, STMTNUM s2);**  Description: Set the Parent(s1, s2) in the ParentTable.  If s1 or s2 are out of range, do nothing. |
| **BOOLEAN isParent (STMTNUM s1, STMTNUM s2);**  Description: If Parent(s1, s2) holds, return true. Else, return false.  If s1 or s2 are out of range, return false. |
| **BOOLEAN isParent (SYNTYPE t1, SYNTYPE t2);**  Description: If Parent(t1, t2) holds, return true. Else, return false.  If t1 or t2 is invalid, return false. |
| **BOOLEAN isParent (SYNTYPE t1, STMTNUM s2);**  Description: If Parent(s2, t1) holds, return true (where t1 is the type of the statement). Else, return false.  If t1 is invalid or s2 is out of range, return false. |
| **BOOLEAN isChildren (SYNTYPE t1, STMTNUM s2);**  Description: If Parent(t1, s2) holds, return true (where t1 is the type of the statement). Else, return false.  If t1 is invalid or s2 is out of range, return false. |
| **STMTNUM\_LIST getParent (SYNTYPE t1, SYNTYPE t2, STMTNUM s);**  Description:  Returns ALL STMTNUM x such that for each x, Parent(x, s) holds.  Where s is of type t2, and each x is of type t1.  If no such statement x exists or if s is out of range, return error code. |
| **STMTNUM\_LIST getChildren (SYNTYPE t1, SYNTYPE t2, STMTNUM s);**  Description:  Returns ALL STMTNUM x such that for each x, Parent(s, x) holds.  Where s is of type t2, and each x is of type t1.  If no such statement x exists or if s is out of range, return error code. |
| **STMTNUM\_LIST getParent (SYNTYPE t1, SYNTYPE t2);**  Description:  Returns ALL STMTNUM x such that for each x, Parent(x,t2) holds.  where each x is of type t1.  If no such statement x exists or if s is out of range, return error code. |
| **STMTNUM\_LIST getChildren (SYNTYPE t1, SYNTYPE t2);**  Description:  Returns ALL STMTNUM x such that for each x, Parent(t2,x) holds.  where each x is of type t1.  If no such statement x exists or if s is out of range, return error code. |
| **STMTNUM getParent (SYNTYPE t1, STMTNUM s2);**  Description:  Returns STMTNUM x such that Parent(x, t2) holds.  where each x is of type t1.  If no such statement x exists or if s is out of range, return error code. |
| **STMTNUM\_LIST getChildren (STMTNUM s);**  Description:  Returns ALL STMTNUM x such that Parent(s, x) holds.  If no such statement x exists or if s is out of range, return error code. |

## 9 Parent\*

|  |
| --- |
| **ParentStar**  *Overview*: ParentStar is used to keep track of the Parent\* relationship between two statement numbers. |
| **API:** |
| **BOOLEAN isParentStar (STMTNUM s1, STMTNUM s2);**  Description: If Parent\*(s1, s2) is holds, return true. Else, return false.  If s1 or s2 are out of range, return false. |
| **STMTNUM\_LIST getChildrenStar (STMTNUM s);**  Description: Returns a list containing ALL STMTNUM x such that Parent\*(s, x) holds.  If no such statement x exists or if s is out of range, return error code |
| **STMTNUM\_LIST getParentStar (STMTNUM s);**  Description: Returns a list containing ALL STMTNUM x such that Parent\*(x, s) holds.  If no such statement x exists or if s is out of range, return error code |

## 10 Modifies

|  |
| --- |
| **Modifies**  *Overview*: Modifies is used to keep track of the Modifies relationship. |
| **API:** |
| **VOID setModifies (STMTNUM s, VARNAME varName);**  Description: Set the Modifies relationship between s and varName to be true in the modifiesAssignmentTable.  If either v or s given is out of range, do nothing. |
| **VOID setModifiesProcedures (PROCNAME procName, VARNAME varName);**  Description: Set the Modifies relationship between procName and varName to be true.  If either p or v given is out of range, do nothing. |
| **BOOLEAN isModifies (STMTNUM s, VARNAME varName);**  Description: If the Modifies relationship between s and v in the ModifiesAssignmentTable is true, return true. Otherwise, return false.  If either v or s given is out of range, return false. |
| **BOOLEAN isModifiesProcedures (PROCNAME procName, VARNAME varName)**  Description: If the Modifies relationship between procName and varName in the ModifiesStatementsTable is true, return true. Otherwise, return false.  If either s or vgiven is out of range, return false. |
| **STMTNUM\_LIST getModifies (SYNTYPE t);**  Description: Return the list of all STMTNUM x, of type t, that modifies any variables.  If there is no such statements, return empty LIST. |
| **VARINDEX\_LIST getModifies (STMTNUM s);**  Description: returns all VARINDEX x such that Modifies(s,x) is true.  If s is out of range, or there no such VARINDEX, return empty LIST. |
| **STMTNUM\_LIST getModifies (SYNTYPE t, VARNAME varName);**  Description: Return the list of all STMTNUM x, of type t, such that Modifies(x,varName) is true.  If there is no such statements, return empty LIST. |
| **PROCNAME\_LIST getModifiesProcedures(VARNAME varName);**  Description: Given varName, get all of the Procedure whose Modifies relationship with varName in the ModifiesStatementsTable is true. Return the list of all of the PROCNAME.  If no PROCNAME fulfils the condition, return empty LIST. |
| **VARNAME\_LIST getModifiesProcedureVariable(PROCNAME procName);**  Description: Given procName, get all of the VARNAME whose Modifies relationship with procName in the ModifiesStatementsTable is true. Return the list of all of the VARNAME.  If no VARNAME fulfils the condition, return empty LIST. |

## 11 Uses

|  |
| --- |
| **Uses**  *Overview*: Uses is used to keep track of the Uses relationship |
| **API:** |
| **VOID setUses (STMTNUM s, VARNAME varName);**  Description: Set the Uses relationship between s and varName to be true.  If s or v given is out of range, do nothing. |
| **VOID setUsesProcedures (PROCNAME procName, VARNAME varName);**  Description: Set the Uses relationship between procName and varName to be true.  If either procName or varName given is invalid, do nothing. |
| **BOOLEAN isUses (STMTNUM s, VARNAME varName);**  Description: If the Uses relationship between s and varName in the UsesStatementsTable is true, return true. Otherwise, return false.  If either s or varName is invalid, return false. |
| **BOOLEAN isUsesProcedures (PROCNAME procName, VARNAME varName);**  Description: If the Uses relationship between procName and varName in the UsesStatementsTable is true, return true. Otherwise, return false.  If either procName or varName is invalid, return false. |
| **VARINDEX\_LIST getUses (STMTNUM s);**  Description: obtain all VARNAME x such that Uses(s,x) is true for each x. Return the list of all of the VARINDEX, by converting it using varTable.  If s is out of range, return empty LIST. |
| **STMTNUM\_LIST getUses (VARNAME varName);**  Description: return all STMTNUM x, such that for each x, Uses(x,varName) is true.  If v is invalid, return empty LIST. |
| **VARNAME\_LIST getUsesProceduresVariable(PROCNAME procName);**  Description: Get all of the VARNAME whose Uses relationship with procName in the UsesStatementsTable is true. Return the list of all of the VARNAME.  If procName is invalid, return empty LIST. |
| PROCNAME\_LIST getUsesProcedures(VARNAME varName);  Description: Given varName, get all of the Procedure whose Uses relationship with v in the UsesStatementsTable is true. Return the list of all of the PROCNAME.  If varName is invalid, return empty LIST. |

## 12 CALLS and CALLS\*

|  |
| --- |
| **Calls**  *Overview*: Calls is used to keep track of the Calls relationship between procedures. |
| **API:** |
| **VOID setCalls (PROCNAME procCall, PROCNAME procCalled);**  Description: Set the Calls relationship between procCall and procCalled to be true. If procCall or procCalled does not exists, error (or throw exception). |
| **BOOLEAN isCalls (PROCNAME procCall, PROCNAME procCalled);**  Description: If the Calls relationship between procCall and procCalled is true, return true. Otherwise, return false.  If procCall or procCalled does not exists, return false. |
| **PROCNAME\_LIST getCalls(PROCNAME procCalled);**  Description: Returns all procedures that calls procCalled directly. |
| **PROCNAME\_LIST getCalled(PROCNAME procCalls);**  Description: returns all procedures that are called by procCalls directly. |
| **PROCNAME\_LIST getCallsStar(PROCNAME procCalled);**  Description: returns all procedures that calls procCalled indirectly or directly . |
| **PROCNAME\_LIST getCalledStarBy(PROCNAME procCalls);**  Description: returns all procedures that are called by procCalls indirectly or directly. |

## 13 CFG (NEXT and NEXT\*)

|  |
| --- |
| **CFG**  *Overview*: This API provides the necessary methods to build a control flow graph and to work with the Next relationship. |
| **API:** |
| **GNODE createNode(INDEX progLine);**  Parameters:  progLine - program line in the given SIMPLE program  Description: Creates and returns reference to a GNode corresponding to the progLine given. |
| **GNODE setNextNode(GNODE currNode, GNODE nextNode);**  Parameters:  currNode - current GNODE  nextNode - next GNODE  Description: Link nextNode as the next node of the currNode. Returns the reference of the currNode. |
| **BOOLEAN isNext(INDEX progLine1, INDEX progLine2);**  Description: If the Next(progLine1, progLine2) holds, return true. Else, do nothing. If progLine1 or progLine2 does not exist, returns false |
| **INDEX getNext(INDEX progLine);**  Description: Returns the index of the program line which comes next directly after progLine. If progLine is out of range, returns error code |
| **INDEX getPrevious(INDEX progLine);**  Description: Returns the index of the program line which comes before directly progLine. If progLine is out of range, returns error code |
| **INDEX getNextStar(INDEX progLine);**  Description: Returns the index of the program line which comes next after dircetly or indirectly progLine. If progLine is out of range, returns error code |
| **INDEX getPreviousStar(INDEX progLine);**    Description: Returns the index of the program line which comes before directly or indirectly progLine. If progLine is out of range, returns error code |

## 14 AFFECTS

|  |
| --- |
| **Affects**  *Overview*: Affects is used to keep track of the Affects relationship between two statements. |
| **API:** |
| **BOOLEAN isAffects (STMTNUM s1, STMTNUM s2);**  Description: If Affects(s1, s2) holds, return true. Else, return false.  If s1 or s2 are out of range, return false |
| **STMTNUM getAffects (STMTNUM s);**  Description: Get a STMTNUM from the right side of the AffectsTable where the left side is STMTNUM s and the relationship is true. Return the STMTNUM.  If the STMTNUM does not exist or s is out of range, return error code. |
| **STMTNUM getAffectedBy (STMTNUM s);**  Description: Get a STMTNUM from the left side of the AffectsTable where the right side is STMTNUM s and the relationship is true. Return the STMTNUM.  If the STMTNUM does not exist or s is out of range, return error code. |

## 14 AFFECTS\*

|  |
| --- |
| **AffectsStar**  *Overview*: AffectsStar is used to keep track of the AffectsStar relationship between two STMT# |
| **API:** |
| **BOOLEAN isAffectsStar (STMTNUM s1, STMTNUM s2);**  Description: If Affects\*(s1, s2) holds, return true. Else, return false.  If s1 or s2 are out of range, return false. |
| **STMTNUM\_LIST getAffectsStar (STMTNUM s);**  Description: Get all of the STMTNUM from the right side of the AffectsStarTable where the left side is STMTNUM s and the relationship is true. Return all of the STMTNUM in list.  If the STMTNUM does not exist or s is out of range, return error code. |
| **STMTNUM\_LIST getAffectedStarBy (STMTNUM s);**  Description: Get all of the STMT# from the left side of the AffectsStarTable where the right side is STMT# s and the relationship is true. Return all of the STMT# in a list.  If the STMT# does not exist or s is out of range, return error code. |

8. Discussion

9. Comments