

**National University of Singapore**

**School of Computing**

**CS3202: Software Engineering Project II**

**TEAM 05: Flying Cockroach**

Semester 1, AY2014/2015

|  |  |  |  |
| --- | --- | --- | --- |
| **Matriculation Number** | **HP Number** | **Student Name** | **Email** |
| **Group-PQL:** | | | |
| A0099214B | 8518 2707 | Adinda Ayu Savitri | savitri.adinda@gmail.com |
| A0098139R | 9082 0864 | Hisyam Nursaid Indrakesuma | indrakesuma.hisyam@gmail.com |
| A0103494J | 9620 7018 | Lacie Fan Yuxin | lacie.jolene.fan@gmail.com |
| **Group-PKB:** | | | |
| A0101286N | 9833 2474 | Ipsita Mohapatra | ipsita@nus.edu.sg |
| A0080415N | 9148 6248 | Steven Kester Yuwono | a0080415@nus.edu.sg |
| A0099768Y | 9178 6540 | Yohanes Lim | yohaneslim93@gmail.com |

Consultation Day/Hour: Monday 6-6.30pm

Contents

[1. SPA 4](#_Toc399251046)

[1.1. Architecture 4](#_Toc399251047)

[1.2. Interaction 5](#_Toc399251048)

[1.3. Development Plan 7](#_Toc399251049)

[1.3.1. For Whole Project 7](#_Toc399251050)

[1.3.2. For Iteration 1 8](#_Toc399251051)

[2. Components 9](#_Toc399251052)

[2.1. Code Parser 9](#_Toc399251053)

[2.2. PKB 11](#_Toc399251054)

[2.3. Design Extractor 14](#_Toc399251055)

[2.3.1. Extracting Relationships 14](#_Toc399251056)

[2.3.2. Building CFG 15](#_Toc399251057)

[2.4. Query Processor 17](#_Toc399251058)

[2.4.1 Query Processor 17](#_Toc399251059)

[2.4.2 Query Parser 17](#_Toc399251060)

[2.4.3 Design and Implementation of Query Evaluator 21](#_Toc399251061)

[3. Testing 25](#_Toc399251062)

[3.1. Testing Plan For Iteration 1 26](#_Toc399251063)

[3.2. Unit Testing 27](#_Toc399251064)

[3.2. Integration Testing 29](#_Toc399251065)

[3.3. System Testing 32](#_Toc399251066)

[4. Coding Standards 36](#_Toc399251067)

[5. API 37](#_Toc399251068)

[1 Parser 37](#_Toc399251069)

[2 Node (Ast) 37](#_Toc399251070)

[3 Vartable 39](#_Toc399251071)

[4 Proctable 39](#_Toc399251072)

[5 TypeTable 40](#_Toc399251073)

[6 Follows and Follows\* 40](#_Toc399251074)

[7 Parent and Parent\* 42](#_Toc399251075)

[8 Modifies 43](#_Toc399251076)

[9 Uses 45](#_Toc399251077)

[10 CALLS and CALLS\* 46](#_Toc399251078)

[11 Design Extractor 47](#_Toc399251079)

[12 Affects 48](#_Toc399251080)

[13 Affects\* 48](#_Toc399251081)

[14 QueryParser 49](#_Toc399251082)

# 1. SPA

Static Program Analyser (SPA) is a program to answer queries about an input SIMPLE program. In this report, we will be describing the design and implementation decisions made during the development of the SPA prototype in CS3201 as well as the changes we have made to the prototype.

## Architecture

The architecture for the prototype consists of 3 main components: the Code Parser, the PKB and the Query Processor. Both the Code Parser and the Query Processor are dependent on PKB but not dependent on each other. Code Parser parses the code and stores design abstractions in each of the 8 tables in the PKB. After Query Parser has parsed the query, the Query Evaluator consults the PKB API to answer queries.



Figure 1

## 1.2. Interaction

CodeParser works by evaluating each line of the given source code. It creates AST Node, set the pointers accordingly; set the tables and the appropriate databases in PKB.

The attributes in PKB (the tables) will then be used by Query evaluator to answer queries. Testing for CodeParser is done by checking the content of each table, whether it has set the values properly, and check the content of each node in the AST, whether it matches the correct AST.



Figure 2

Figure 3 shows the sequence diagram of query evaluation process. This diagram was useful in demarcating the responsibilities of each PQL group member. For example, QueryEvaluator directly assumes that the Query it receives is valid and syntactically correct. Therefore it is the responsibility of QueryParser to validate each query before passing it to the evaluator.



Figure 3

This diagram also helps to keep track of the dependencies between components. This is especially useful during debugging process of integration testing. When QueryProcessor fails to return the correct result, the team knows that the errors could come from at least three places, i.e. QueryParser, QueryEvaluator, and PKB.

## 1.3. Development Plan

### 1.3.1. For Whole Project

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Iteration 1** | | | | | **Iteration 2** | | | |
| Team member | **Implement Calls in PKB** | **Implement Modifies and Uses in PKB** | **Extend Query Parser to support with clause** | **Extend Query Evaluator to support with clause** | **Write system test cases for the enhancement** | **Implement Next in PKB** | **Extend Query Parser to support multiple clauses** | **Extend Query Evaluator to support multiple clauses** | **Write system test cases for the enhancement** |
| Adinda |  |  |  |  |  |  |  |  |  |
| Lacie |  |  |  | \*Pending |  |  |  |  |  |
| Hisyam |  |  |  |  |  |  |  |  |  |
| Steven |  |  |  |  |  |  |  |  |  |
| Ipsita | \*Achieved |  |  |  |  |  |  |  |  |
| Yohanes |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Iteration 3** | | | |
| Team member | **Implement Affects in PKB** | **Extend Query Parser to support Affects and tuple results** | **Extend Query Evaluator to support Affects and tuple results** | **Write system test cases for the enhancement** |
| Adinda |  |  |  |  |
| Lacie |  |  |  |  |
| Hisyam |  |  |  |  |
| Steven |  |  |  |  |
| Ipsita |  |  |  |  |
| Yohanes |  |  |  |  |

### 1.3.2. For Iteration 1

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Iteration 1** | | | | | | | | | |
| Team member | **Optimize internal data structure for PKB** | **Implement Information hiding in PKB** | **Implement Modifies (proc), Uses (proc), and Calls relations in PKB** | **Implement Design Extractor functions for extracting relationships** | **Extend Query Parser to support with clause** | **Come up with design alternative for optimization of Query Evaluator** | **Redesign internal structure of Query Evaluator** | **Extend Query Evaluator to support with clause** | **Enhance system testing mechanism** | **Write system test cases for iteration 1** |
| Adinda |  |  |  |  |  |  |  |  |  |  |
| Lacie |  |  |  |  |  |  |  |  |  |  |
| Hisyam |  |  |  |  |  |  |  |  |  |  |
| Steven |  |  |  |  |  |  |  |  |  |  |
| Ipsita |  |  |  |  |  |  |  |  |  |  |
| Yohanes |  |  |  |  |  |  |  |  |  |  |

# 2. Components

## 2.1. Code Parser

Code Parser’s main functions are to read in the source code, build the AST, and set the tables (VarTable, ProcTable, TypeTable, Follows, Parent, Modifies, Uses) in PKB according to the input source code.

To build the AST, Code Parser depends on the implementation of node, which is the node structure being used to build AST. A node can have many children, therefore, we decided to use NODE\_PTR\_LIST for dynamic storage of the children pointers, rather than array with fixed size.

Code Parser keeps track of every relevant parent of each node being built, by storing pointers to their parents. For example when there is a while statement, a pointer to the while statement will be stored and then link all the children to their parent node.

Code Parser works by tokenizing the source code as string line by line, and then detecting the tokens for each line, and generating the types, program lines, and setting the tables accordingly.

For an assignment statement, Code Parser will convert the expression from infix to postfix, and then create the expression tree.

Code Parser does its validation by keeping track of the curly brackets (i.e. “{“ and “}” ). It keeps track of the number currently present open curly bracket, “{“. When Code Parser encounters an open curly bracket, it will push it to a stack. When it encounters a closed curly bracket, it will pop from the stack.

When Code Parser reaches the end of the source code, it will return invalid if the stack is not empty, or if Code Parser is trying to pop from an empty stack. It means there is a mismatch in the number of curly brackets.

When the stack is empty, Code Parser will accept a line which defines a procedure. If it encounters any other statement while the stack is empty, it will return invalid.

Example:

Source code (source1.txt):

procedure Mini {

A1 = 29;

a1 = 31;

i = 51; }

The following describes how the Code Parser parses the above SIMPLE program:

1. Code Parser starts reading the source code at line 1, it will check whether the stack is empty. If the stack is empty, it will be expecting a procedure declaration.
2. It then parses procedure Mini, creates an ASTNode, sets it as root, and pushes the curly bracket “{“ into the stack. Insert “Mini” into the ProcTable.
3. At line 2, Code Parser tokenizes and checks the type of statement. Since it starts with a variable, it detects the statement as an assignment statement.
4. Code Parser will check if there exists a semicolon at the end of the line because it is compulsory to have a semi colon at the end of an assignment statement. If it exists, it will create a node containing “=”, and link “A1” as the first child. The expression on the right hand side will be converted into a postfix expression, and then build the expression tree.
5. Code Parser will link the root of the expression tree as the second child of “=”, and then link the “=” to its parent, which in this case, is “procedure Mini”.
6. Code Parser will also set the VarTable, Modifies, Uses, Follows, and Parent accordingly, in this case, it will set Modifies (line 1, and variable A1), and insert A1 into the VarTable.
7. At line 3, it detects that it is an assignment statement, and repeat step 4 to 6.
8. At line 4, it detects that it is an assignment statement, and repeat step 4 to 6. In addition, Code Parser detects a closed curly bracket. Therefore, Code Parser will pop the stack.
9. End of source code is reached. It will now check whether the stack is empty.
10. Since the stack is empty, and there is no violation of the rules stated earlier. Code Parser has built the AST successfully and stored the design abstractions in the relevant tables.

## 2.2. PKB

PKB is implemented using the singleton pattern. One instance of PKB will be initialised during the construction phase of the UI (which is AutoTester). Afterwards, we will only pass the PKB pointer to other components which need to alter the PKB or call the PKB’s methods. This is to ensure that other components are always editing or accessing the same PKB object. Using the same rationale, all the sub-components of PKB (VarTable, ProcTable, ConstTable, Follows, Parent, Modifies and Uses) are singleton classes and only their pointers are passed around.

To make things clearer, some new definitions we used are as follows:

* Typedef int
  + STMTNUM: for statement number
  + VARINDEX: for variable index
  + CONSTVALUE: for constant value
* Typedef string
  + VARNAME: for variable name
  + PROCNAME: for procedure name
* Typedef Enum SynType {ASSIGN, IF, WHILE, STMT, BOOLEAN, CALL, VARIABLE, CONSTANT, PROGLINE, INVALID} TYPE
  + to discern the type of each statement number
* Typedef pair<PROCINDEX,STMTNUM> CALLSPAIR;
  + To store the index of the procedure being called and the statement number where the call is invoked

The data structures used in the PKB for the tables and the relationships are vectors and unordered\_map as listed down below.

|  |  |
| --- | --- |
| **PKB – Design Abstractions** | |
| **Tables** | **Data Structures** |
| ProcTable | vector<PROCNAME> |
| ConstTable | vector<CONSTVALUE> |
| TypeTable | vector<TYPE> |
| **Relationships** | **Data Structures** |
| Follows | unordered\_map<STMTNUM,STMTNUM> |
| Parent | unordered\_map<STMTNUM,vector<STMTNUM>> & unordered\_map<STMTNUM,STMTNUM> |
| Uses | unordered\_map<STMTNUM,vector<VARINDEX>> |
| Modifies | unordered\_map<STMTNUM,vector<VARINDEX>> |
| Calls | unordered\_map<PROCINDEX, vector<CALLSPAIR>> |

For the data structures used in PKB, the main indicator that we are looking for is the speed. This includes the speed of insertion and searching. Listed below is the table comparison between vector, ordered map and unordered map.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector of Size N** | **Ordered Map of Size N** | **Unordered Map of Size N** |
| **Insert** | O(1) or O(N) if need resizing | O(log N) | O(1) (Average case) |
| **Search** | O(1) | O(log N) | O(1) (Average case) |

For the tables (and the inside of the relationship data structrures), we are using vector as we want to keep the ordering of the index. To keep the ordering of the index, the choice boils down to vector and ordered map. From the comparison above, we can see that insertion (if no resizing) and search for vector is faster than ordered map. We can initialise the vector to be very big in the beginning to avoid frequent resizing to make up for the slower time when resizing is needed.

For the relationships, we are using unordered map as it is the fastest out of the three. The main drawback is that it will consume more memory but we prioritize speed over memory consumed.

The PKB mainly interacts with the Code Parser and the Query Evaluator.

**Interaction with Code Parser**

After PKB is initialised, we will pass the PKB pointer to Code Parser to fill in the tables and relationships into the PKB.

For example, when the Code Parser calls insertVar(“x”), the following is done:

1. Check whether the given variable, “x”, exists in the table by iterating through the element in the table one by one.
2. If yes, we will simply just return the index.
3. Otherwise, insert the element at the back of the table and return the index (table size - 1) of the variable.

For example, when the Code Parser calls setUses(12,”x”)the following is done:

1. Get the variable index of “x” from VarTable.
2. If the variable index is -1, it means that there is no such variable and thus the method will terminate.
3. If the variable index is more than -1, it means that the variable exists, therefore insert into the uses table at key 12, the value of variable index “x”.

**Interaction with Query Evaluator**

After interacting with Code Parser, PKB pointer will then be passed to Query Processor so that Query Evaluator can call the public API provided by PKB. Query Processor will need to get the tables or relationships that it needs first and only then it can call the corresponding API that it needs.

For example, if we want to call getChildren(WHILE, CALL), Query Processor needs to get parent from PKB and then calls parent->getChildren(WHILE, CALL). When it is called, it will result a list of STMTNUM x such that for each x, Parent(CALL, x) holds and x is a WHILE statement. If there exists no such statement x, an error code is returned. The steps are as follows:

1. Get parent pointer from PKB using getParent()
2. Calls the method getChildren(WHILE,CALL) from parent
3. Iterate the children table inside parent from beginning to end. The index of the vector, i, will indicate the statement number of the children.
4. Get j, the value of the vector at the specified index which is the statement number of the parent
5. If j is -1, continue with the next index from step 1.
6. Use isType(WHILE, j) to check the type of j from the TypeTable to see whether it is of type WHILE or not.
7. If not, continue with the next index from step 1.
8. If yes, ise isType(CALL, i) to check the type of I from the TypeTable to see whether it is of type CALL or not.
9. If not, continue with the next index from step 1.
10. If yes, push i into the vector of answer and continue with the next index from step 1.
11. After iterating through the whole children map, return the vector of answer.
12. If the vector of answer is empty, return the vector with -1 as the only element.

## 2.3. Design Extractor

The main role of the design extractor is to extract relationships about the SIMPLE program that could not be extracted in the one-time parsing done by the Code Parser. This includes:

* + Extracting information about Modifies and Uses for procedures and for program lines that are calls statements
  + Building the Control Flow Graph (CFG) from the AST and subsequently storing it in the PKB and storing the Next relationship in the PKB

### 2.3.1. Extracting Relationships

The following shows the steps required to extract relationships like Modifies and Uses for multiple procedures in a single SIMPLE program:

1. Obtain the Calls Table from the PKB. An example of the calls Table is shown below where procedure with index 0 calls procedure with index 1 from program line 3 and procedure with index 4 calls procedure with index 3 from program line 21.

0 🡪 (1, 3) (4, 5)  
1 🡪 (2, 8) (3, 10)  
2 🡪  
3 🡪  
4 🡪 (3, 21)

This translates into a Calls Tree as follows where the nodes represent procedure indices and the edges represent the program line at which there is a calls statement.



1. Run Depth First Search (DFS) on the Calls table in order to obtain a topological sort order of the procedure indices in a queue (below) which starts on the left. Each Queue contains a procedure index and a vector of program lines in which the procedure is called either directly or indirectly.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(2, [3, 8]) (3, [3, 10]) (1, [3]) (3, [5, 21]) (4, [5]) (0, [ ])  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Starting from the head of the queue, find all the variables that are modified and all the variables that are used in the procedure. For each of the program lines in the Queue Item, set these program lines (p) to modify and use the respective variables. For each of the program lines (p), if they are contained in another container statement (c), then set these program lines (c) to modify and use the respective variables too.

### 2.3.2. Building CFG

Given the AST, the following shows the pseudo code to build the CFG and store the Next relationship in the PKB.

Maintain a currCFGNode pointer.

**BuildCFG()**

Create CFG Root with progLine = 0  
Iteratively traverse each of the type = procedure nodes in AST

Update the currASTNode to stmtLst node under the procedure node  
Update the currCFGNode to point to the rootCFGNode  
CreateCFGForStmtLst( currASTNode )

**CreateCFGForStmtLst( ASTNode )** [where ASTNode points to the :stmtLst node]

Iteratively traverse each of the children AST nodes of the currASTNode

If type==ASSIGN

CreateCFGForAssign( currASTNode.progLine )

Else if type == CALL

CreateCFGForCall( currASTNode.progLine )

Else if type == WHILE

CreateCFGForWhile( currASTNode.getChild() )

Else if type == CALL

CreateCFGForIf( currASTNode.getChild() )

**CreateCFGForAssign( progLine )**

If currCFGNode is a dummy node

dummyNode.progLine = progLine  
 Set Next relationship in PKB

Else CreateNewNodeAndAttachToCFG()

**CreateCFGForCall( progLine )** [omitted because it is SIMILAR TO CreateCFGForAssign]

**CreateCFGForWhile( ASTNode )** [where ASTNode points to the :stmtLst node]

If currCFGNode is a dummy node

dummyNode.progLine = progLine  
 Set Next relationship in PKB

Else CreateNewNodeAndAttachToCFG()  
Save the toNode to be the currCFGNode  
createCFGForStmtLst( ASTnode.getChild() )  
Find the fromNode in the CFG using DFS.   
Create an arrow in the CFG from fromNode to toNode.  
Set Next relationship in PKB  
Update currCFGNode to toNode

**CreateCFGForIf( ASTNode )** [where ASTNode points to the :stmtLst node]

If currCFGNode is a dummy node

dummyNode.progLine = progLine  
 Set Next relationship in PKB

Else CreateNewNodeAndAttachToCFG()  
For then:stmtLst, createCFGForStmtLst()  
Store pointer to currCFGNode in a vector called leafNodes  
For else:stmtLst, createCFGForStmtLst()  
Store pointer to currCFGNode in a vector called leafNodes

Create a dummy node with progLine = -1  
For each of the CFGNodes in leafNodes, make it the parent of the dummyNode  
Update the currCFGNode to point to the dummyNode

**CreateNewNodeAndAttachToCFG()**

Create new node and attach to existing CFG  
 Set Next relationship in PKB

## 2.4. Query Processor

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

### 2.4.1 Query Processor

Query Processor is a façade class for the whole component. The following shows the steps it takes:

1. Query Processor calls QueryParser to create a Query object from the given query string.
2. Query Processor then passes the Query object to the QueryEvaluator.
3. Query Evaluator will compute all necessary relations and return the results in the form of a list of integers.
4. Query Processor transforms the result into the correct display format and returns the answer to the user.

### 2.4.2 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.

**Query Validation**

**Query validation** is done using regular expression method using the grammar rules written in the handbook.

We have considered to validate while parsing the query (tokens), but decided to use regular expression to validate the query first then parse, because of the following reasons:

1. The code is much neater and simpler
2. It is faster to detect errors rather than parsing and validating (especially if the error is towards the end of the query)
3. Query parsing becomes simpler if we already know the exact possible format of the string query that needs to be parsed.

All types of queries have been defined in a static string, following the grammar rules in the textbook. The strings are then used to validate the queries using regular expression, where the definitions earlier are used. It is very convenient because the grammar rules in the book is close to a regex grammar rules.

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 use regular expression to check the validity of the statement and retrieve the tokens. Valid declaration statements will be converted into a map with the synonym as the key and its type as the value. This map is called the synonym map and will be used later by the parsing function. In this example, the synonym-map will look as follows:

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

This map 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.

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 use regular expression to check the validity of the statement and retrieve the appropriate tokens.

If the regular expression matching fails, (for example, the number of arguments in the clause is not exactly two) the validator will instantly terminate and declare the query invalid. In the case where statement is valid, all the tokens from the select statement will be stored in a vector, *selectStatement*. This vector 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 | a | x | \_”x+y”\_ |

**Query Parsing**

The parser processes the *selectStatement* vector from the earlier. The *selectStatement* vector will be processed to construct a Query object with the following structure.

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

The selected synonym, in this example is ‘a’, will be stored inside a string in the Query object. The synonym map 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  *string* pattern-synonym |

Since each query can contain many select clauses, these clauses are stored inside a vector for scalability purposes. All the synonyms present in the *selectStatement* vector will be detected and validated once again on whether they have been declared earlier. If it is not declared, the query is invalid and **QueryParser** will indicate and return invalid. 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”\_ |
| pattern-synonym | a |

|  |  |
| --- | --- |
| 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.

For with-clause, Query parser with detect the conditions whether they are valid.

For example:

procedure p,q; Select q such that Calls (p,q) with p.procName=”Second”

p.procName = ”Second” will be parsed into two parts, left-hand-side and right-hand-side.

Right-hand-side includes “Second” and will be stored in token/argument-2 in a relationship object.

Left-hand-side includes p.procName and will be stored in token/argument-1 if token is valid. Synonym p will be checked against the map whether it exists. Since the attribute name is procName, it will also be checked on whether it is of type procedure.

Left-hand-side and right-hand-side will be checked on whether they are of the same type, either character strings or integers. If they are of different type, return invalid.

If it passes both validations, then the query is valid and stored in the relationship below:

|  |  |
| --- | --- |
| rel2 | |
| relationship-type | WITH |
| argument-1 | p |
| argument-2 | “Second” |

procName is not stored because it is known that a synonym of type procedure can only have procName as its attribute name.

### 2.4.3 Design and Implementation of Query Evaluator

**Current Implementation for Basic Query Evaluation (BQE)**

The Query Evaluator has one main public function: evaluateQuery which accepts a Query object passed into it by the Query Processor. In evaluateQuery, we decompose the Query object to retrieve the clauses stored in the Relationship objects. We then created a private function for each type of clause that the BQE is required to evaluate:

Query Evaluator

evaluateQuery

evaluateFollows

evaluateFollows\*

evaluateParent

evaluateParent\*

evaluateModifies

evaluateUses

evaluateCalls

evaluateCalls\*

evaluateWith

evaluatePattern

**Data Representation for BQE**

For the Query Evaluator, it only returns a set of answers that is requested by the Query object. The set of answers is found by intersecting the results returned by each respective private function. To represent the set of answers, we considered a list of possiblilities:

* Vector
* Array
* List

**Optimization analysis**

After careful consideration, we have decided that vector is the most desirable data representation as it has better memory management than arrays and an equally rich interface as list.

**Implementing Calls, Calls\*, With and Pattern**

Similar to the rest of the functions, we created evaluateCalls, evaluateCalls\* and evaluateWith for each of the three new relationships that we introduced in this iteration.

1. evaluateCalls

BQE interacts closely with PKB and calls its function based on the selected token in the Query object. For example, if the query is: **procedure p; Select p such that Calls(p, “Third)**, BQE will call the PKB function getCalls. If the query is: **procedure p; Select p such that Calls(“First”, p)**, then BQE will call the PKB function getCalled.

1. evaluateCalls\*

BQE calculates Calls\* by calling the PKB functions repeatedly. For example: **procedure p; Select p such that Calls\*(“Third”,p)**. BQE will first call the PKB function getCalled to get the immediate procedures that are directly called by “Third”. Then from the answers returned, BQE will call getCalled again but pass in all the previous procedures as parameters instead of “Third”.

1. evaluateWith

As a value is associated with each with clause, the value will be passed into the token of which the value corresponds in the rest of the Relationship objects. For example: **variable v; assign a; Select a such that Modifies(a, v) with v.varName=”x”**. BQE will then substitute v in the Modifies clause with x to yearn Modififies(a,”x”) and then evaluate it using evaluateModifies.

1. Pattern

To evaluate Patterns, we do DFS (Depth First Search) starting from the ASTRoot. The DFS itself is implemented using a Stack and continues until the Stack is empty.

We decided to do tree traversal to find patterns instead of storing them as it would be easily extendable when searching for new types of patterns. For example, a pattern which matches if statements can be found the same way we find assignment statements.

Before running the DFS, we first create a new tree that matches the pattern. This tree is used to see if a given line matches the needed pattern.

When we run the DFS, we first search for nodes that match the type of pattern. When a node is found with the appropriate type, we check to see if it matches our pattern tree. If it does, we add it to our answer, else we continue the search.

**Analysis of the current BQE design decisions**

1. Ease of Changing/Flexibility

The current implementation of BQE lacks in flexibility as many changes will need to be made to the algorithm with future extensions to the query formats. For example, if a query is made with multiple clauses, the current BQE will not be able to return an accurate set of answers with its current intersection algorithm:

**assign a; variable v1; variable v2; Select a such that Modifies(a,v1) and Modifies(a,v2)**

From the above query, BQE will return all assign statements that modifies one variable instead of two distinct ones as it is not able to identify v1 and v2 as two different assets.

1. Reusability

As the functions are created according to the type of relationships that can be queried, BQE has a high reusability. If there is a new relationship to be defined, a new separated function can also be created under the evaluateQuery function for that clause. This is useful when implementing relationships such as Next and Affect in the next few relationships

1. Memory Utilization

As we are using vectors, memory utilization is kept to a minimum as compared to arrays. The amount of memory needed is proportional to the size of answers returned and the Query object.

1. Performance

The running time of BQE is O(nx) where n is the number of relationship clauses in the Query object and x is the running time taken by the PKB to find the solutions.

**Design Solution Alternative**

To solve flexibility issues, we have come up with another algorithm that works better with multiple clauses and tuples. Suppose we have the same query as above:

**assign a; variable v1; variable v2; Select a such that Modifies(a,v1) and Modifies(a,v2)**

To solve this query, first, we will create tuples of all combinations of the existing declared variables, which in this case is a,v1,and v2. Suppose there are only 2 assignment statements and 2 variables(“x” and ”y”) in the source code.

|  |  |  |
| --- | --- | --- |
| a | v1 | v2 |
| 1 | x | y |
| 1 | y | x |
| 2 | x | y |
| 2 | y | x |

To solve each relationship individually, we iterate through the relationship vector given to us in the Query object. The first relationship to be evaluated will be Modifies(a,v1), where we substitute each tuple into the relationship. If the tuple is verified to be true for this relationship, we will keep it in the table. If not, it will be eliminated. The resulting table will then be substituted to the next relationship. After all relationships are evaluated, the resulting table will be the set of accurate answers.

**Analysis of the design solution alternative**

1. Ease of Changing/Flexibility

This solution is much more flexible than the current version due to its compatibility with the query format. Suppose new relationships are introduced into the SPA system. Little changes will need to be done to the algorithm and accurate results can still be retrieved

1. Reusability

Reusability is neither improved nor compromised with this alternative

1. Performance and Memory utilization

The running time of the alternative is much slower due to the construction of the tuples and hence is extremely compromised. More memory is also needed to store the tuples in addition to the set of answers evaluated.

# 3. Testing

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.

## 

## 3.1. Testing Plan For Iteration 1

Iteration 1 will adopt Test-Driven Development, which restructures the way that testing is carried out from the last iteration. Previously, the testing cycle started with unit testing, followed by integration testing and system testing. However, we realized that a lot of cases were missed at the last iteration, warranting a more thorough testing mechanism. Now that we have one team member who is tasked mainly at testing, we can ensure a proper test-driven development with more exhaustive test scenarios.

The diagram below illustrates the testing cycle of iteration 1. It started with the writing of system test cases at the beginning of iteration 1, which served as an executable specification of the system. The team then started to implement each functionality with the test cases in mind.

Failed

Passed

Passed

Failed

Each team member is responsible for his/her own unit testing. And once he/she has passed the unit testing, he/she can submit his/her codes for integration testing. Once integration testing is passed, the system will be tested again with system testing. All of these tests were carried out in an agile way, which makes for a flexible testing timeline and life cycle.

## 3.2. 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. in the source code, whether they contains the expected values.



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.

## 

## Integration Testing

Integration Testing was split into two parts, Parser-PKB and PKB-Query Processor.



For Parser-PKB testing, a sample source is parsed and assertions are made to see the correctness of said parsing.



For PKB-Query Processor testing, queries are parsed by the QueryParser and then evaluated in the QueryProcessor. The answers provided by the QueryProcessor are asserted to check for correctness.



## System Testing

System testing is carried out at the beginning and end of each testing life cycle, as already explained in chapter 3.1. In iteration 1, system testing is restructured to allow easier detection of error, and support more exhaustive test scenarios. This is achieved by implementing isolation of test scenarios and mutual exclusivity in the writing of the test cases.

Previously, test cases for system testing is grouped according to the iteration. This means that in each test file, there was a mix of several functions, e.g. Follows and Parent\*, etc.. When the system failed the test cases in that file, each team member has to rigorously scan the output file to identify the defect function.

Now, test cases are grouped according to functional requirement in isolation, unless an interaction is needed. The test cases are also divided into three difficulty levels: basic, intermediate, and advance.

Basic test cases act as prototype testing for each new functionality that is implemented. This means that each test file is only targetting one area of the implementation. This enables easier isolation of errors. Intermediate test cases test the interactions of different functionalities, e.g. such that clause and with clause, etc.. Advance test cases are the more complex version of the intermediate cases, integrating invalid queries and other components that may cause errors.

System test cases are also accompanied by a documentation that enables team members to keep track of issues for the next testing life cycle. The table below illustrates such testing documentation, and also details the test structure for iteration 1.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Iteration** | **Level** | **Availability** | **Function** | **Source File** | **Test File** | **Status** | **Notes** |
| 1 | Basic | Available | With clause | Source1A | QBasic1A with clause | Pending |  |
| 1 | Basic | Available | Calls | Source1B | QBasic1B calls | Pending |  |
| 1 | Basic | Available | Calls\* | Source1B | QBasic1C calls star | Pending |  |
| 1 | Basic | Available | Modifies proc | Source1D | QBasic1D modifies proc | Pending |  |
| 1 | Basic | Available | Uses proc | Source1D | QBasic1E uses proc | Pending |  |
| 1 | Basic | Available | Invalid queries | <any source> | QBasic1F invalid queries | Pending |  |

To ensure that the test cases are collectively exhaustive, scenarios that are covered by the test cases are also documented systematically. The table on the next few pages will detail the scenarios used for iteration 1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Iteration** | **Level** | **Function** | **Availability** | **Requirement 1** | **Requirement 2** | **Status** |
| 1 | Basic | With clause | Available | proc.procName | select procedure | pending |
|  |  |  | Available | proc.procName | select boolean | pending |
|  |  |  | Available | var.varName | select variable | pending |
|  |  |  | Available | var.varName | select boolean | pending |
|  |  |  | Available | cons.value | select constant | pending |
|  |  |  | Available | cons.value | select boolean | pending |
|  |  |  | Available | stmt.stmt# | select statement | pending |
|  |  |  | Available | stmt.stmt# | select boolean | pending |
|  |  |  | Available | assign.stmt# | select assign | pending |
|  |  |  | Available | assign.stmt# | select boolean | pending |
|  |  |  | Available | while.stmt# | select while | pending |
|  |  |  | Available | while.stmt# | select boolean | pending |
|  |  |  | Available | if.stmt# | select if | pending |
|  |  |  | Available | if.stmt# | select boolean | pending |
|  |  |  | Available | prog\_line.prog\_line# | select prog\_line | pending |
|  |  |  | Available | prog\_line.prog\_line# | select boolean | pending |
|  |  |  | Available | procName = varName | select proc | pending |
|  |  |  | Available | procName = varName | select variable | pending |
|  |  |  | Available | procName = varName | select boolean | pending |
|  |  |  | Available | stmt# = value | select statement | pending |
|  |  |  | Available | stmt# = value | select constant | pending |
|  |  |  | Available | stmt# = value | select while | pending |
|  |  |  | Available | stmt# = value | select if | pending |
|  |  |  | Available | stmt# = value | select while | pending |
|  |  |  | Available | stmt# = value | select boolean | pending |
|  |  |  |  |  |  |  |
|  |  |  |  | *param1, param2* |  |  |
| 1 | Basic | Calls | Available | \_, \_ | select assign | pending |
|  |  |  | Available | \_, \_ | select boolean | pending |
|  |  |  | Available | "procName1", "procName2" | select while | pending |
|  |  |  | Available | "procName1", "procName2" | select boolean | pending |
|  |  |  | Available | syn1, \_ | select proc1 | pending |
|  |  |  | Available | \_, syn2 | select proc2 | pending |
|  |  |  | Available | syn1, syn2 | select proc1 | pending |
|  |  |  | Available | syn1, syn2 | select proc2 | pending |
|  |  |  | Available | "procName1", syn2 | select proc2 | pending |
|  |  |  | Available | syn1, "procName2" | select proc1 | pending |
|  |  |  | Available | syn, syn | select proc | pending |
|  |  |  |  |  |  |  |
| 1 | Basic | Calls\* | Available | \_, \_ | select assign | pending |
|  |  |  | Available | \_, \_ | select boolean | pending |
|  |  |  | Available | "procName1", "procName2" | select while | pending |
|  |  |  | Available | "procName1", "procName2" | select boolean | pending |
|  |  |  | Available | syn1, \_ | select proc1 | pending |
|  |  |  | Available | \_, syn2 | select proc2 | pending |
|  |  |  | Available | syn1, syn2 | select proc1 | pending |
|  |  |  | Available | syn1, syn2 | select proc2 | pending |
|  |  |  | Available | "procName1", syn2 | select proc2 | pending |
|  |  |  | Available | syn1, "procName2" | select proc1 | pending |
|  |  |  | Available | syn, syn | select proc | pending |
|  |  |  |  |  |  |  |
| 1 | Basic | Modifies proc | Available | procSyn, varSyn | select proc | pending |
|  |  |  | Available | procSyn, varSyn | select variable | pending |
|  |  |  | Available | procSyn, varSyn | select boolean | pending |
|  |  |  | Available | procSyn, \_ | select proc | pending |
|  |  |  | Available | procSyn, \_ | select boolean | pending |
|  |  |  | Available | procSyn, "varName" | select proc | pending |
|  |  |  | Available | procSyn, "varName" | select boolean | pending |
|  |  |  | Available | procSyn, "varName" | select variable | pending |
|  |  |  | Available | "procName", "varName" | select boolean | pending |
|  |  |  | Available | "procName", "varName" | select proc | pending |
|  |  |  | Available | "procName", "varName" | select variable | pending |
|  |  |  | Available | "procName", \_ | select boolean | pending |
|  |  |  | Available | "procName", \_ | select proc | pending |
|  |  |  | Available | "procName", \_ | select variable | pending |
|  |  |  | Available | "procName", varSyn | select variable | pending |
|  |  |  | Available | "procName", varSyn | select proc | pending |
|  |  |  | Available | "procName", varSyn | select boolean | pending |
|  |  |  |  |  |  |  |
| 1 | Basic | Uses proc | Available | procSyn, varSyn | select proc | pending |
|  |  |  | Available | procSyn, varSyn | select variable | pending |
|  |  |  | Available | procSyn, varSyn | select boolean | pending |
|  |  |  | Available | procSyn, \_ | select proc | pending |
|  |  |  | Available | procSyn, \_ | select boolean | pending |
|  |  |  | Available | procSyn, "varName" | select proc | pending |
|  |  |  | Available | procSyn, "varName" | select boolean | pending |
|  |  |  | Available | procSyn, "varName" | select variable | pending |
|  |  |  | Available | "procName", "varName" | select boolean | pending |
|  |  |  | Available | "procName", "varName" | select proc | pending |
|  |  |  | Available | "procName", "varName" | select variable | pending |
|  |  |  | Available | "procName", \_ | select boolean | pending |
|  |  |  | Available | "procName", \_ | select proc | pending |
|  |  |  | Available | "procName", \_ | select variable | pending |
|  |  |  | Available | "procName", varSyn | select variable | pending |
|  |  |  | Available | "procName", varSyn | select proc | pending |
|  |  |  | Available | "procName", varSyn | select boolean | pending |
|  |  |  |  |  |  |  |
| 1 | Basic | Invalid Queries | Available | Modifies | Type mismatch | pending |
|  |  |  | Available | Uses | Type mismatch | pending |
|  |  |  | Available | With clause | Type mismatch | pending |
|  |  |  | Available | Calls | Type mismatch | pending |
|  |  |  | Available | Modifies | Ambiguous query | pending |
|  |  |  | Available | Uses | Ambiguous query | pending |

# 4. Coding Standards

Our team members adopted similar coding standards which are adjusted appropriately and respectively according to the design specifications of various components. Some of the coding standards that the components possess are listed below:

1. Indentation and whitespace
   1. a. Indication of code segments
2. Comments to enhance understanding and communication
3. Descriptive variable declarations
   1. Always start with lower case
   2. Use CamelCase
   3. Use only letters and numbers
4. Informative function naming conventions
   1. All getters start with “get”
   2. All setters start with “set”
   3. All functions that start with “is” returns a Boolean value
5. Keep it simple and effective
   1. Avoid complex code fragments
6. Refactoring

**Standards between abstract APIs and concrete APIs**

The correspondence between the abstract and concrete APIs was enhanced by doing the following:

1. The abstract APIs provides the interface for the concrete APIs
2. Making abstract APIs as comprehensive as possible a. Offering an Extensive description of the abstract APIs b. Specifying the complete parameters needed for the function

# 5. API

The following tables show the full API for the various components in our SPA. Unimplemented methods/functions are in yellow text while those implemented are in black.

## 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 (Constructor):** |
| Node();  Description: a constructor to create a blank node. |
| Node(DATA newData, TYPE newType);  Description: a constructor to create a node with newData and newType as its attributes. |
| Node(DATA newData, TYPE newType, STMTNUM progLine);  Description: a constructor to create a node with newData, newType, and progLine as its attributes. |
| API (Method): |
| 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. |
| STMTNUM 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(STMTNUM newProgLine);  Description: set the STMTNUM 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:** |
| VARINDEX insertVar (VARNAME varName);  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 (VARINDEX index);  Description: Returns the name of a variable at VarTable [index]  If ‘index’ is out of range, return error code |
| VARINDEX getVarIndex (VARNAME varName);  Description: If varName is in VarTable, returns its index. Otherwise, returns error code |
| INTEGER 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:** |
| PROCINDEX insertProc (PROCNAME procName);  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. |
| PROCINDEX getProcIndex (PROCNAME procName);  Description: If procName has a corresponding index in the ProcTable, returns its index. Otherwise, returns error code. |
| PROCNAME getProcName (PROCINDEX index);  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 |
| INTEGER getNumProcedures ();  Description: Returns the total number of unique procedures stored in the procTable. |

## 5 TypeTable

|  |
| --- |
| **TypeTable**  *Overview*: TypeTable stores all the types (assign,calls, while,etc) for each program line, from a source program. |
| **API:** |
| TYPEINDEX 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, STMTNUM 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 and 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 (TYPE t1, TYPE 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 (TYPE 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 (TYPE t, STMTNUM s);  Description: Returns x such that Follows(s, x) holds. (and if x is of TYPE t)  If s does not exist or is out of range, return error code. |
| STMTNUM getFollowedBy (TYPE t, STMTNUM s);  Description: Returns x such that Follows(x, s) holds. (and if x is of TYPE t)  If s does not exist or is out of range, return error code. |
| STMTNUM\_LIST getFollows (TYPE t1, TYPE 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 (TYPE t1, TYPE 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. |
| 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 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 error code |

## 7 Parent and Parent\*

|  |
| --- |
| **Parent**  *Overview*: Parent is used to keep track of the Parent relationship between two statements.  ParentStar methods are currently being evaluated by QueryEvaluator, but we are planning to move and include the methods here, (Parent). |
| **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 (TYPE t1, TYPE t2);  Description: If Parent(t1, t2) holds, return true. Else, return false.  If t1 or t2 is invalid, return false. |
| BOOLEAN isParent (TYPE 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 (TYPE 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 (TYPE t1, TYPE 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 (TYPE t1, TYPE 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 (TYPE t1, TYPE 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 (TYPE t1, TYPE 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 (TYPE 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. |
| 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 |

## 8 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 modifiesTable.  If either varName or s given is out of range, do nothing. |
| VOID setModifies (STMTNUM s, VARINDEX\_LIST varList);  Description: Set the Modifies relationship between s and variable index inside varList to be true in the modifiesTable.  If either variable indexes inside the varList or s given is out of range, do nothing. |
| VOID setModifiesProc (PROCINDEX procIndex, PROCINDEX\_LIST procList);  Description: Set the Modifies relationship between procIndex and the procedure index inside the proclist to be true.  If any of the procedure index in procList or procIndex 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 (TYPE 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 (TYPE 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. |

## 9 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 varName given is out of range, do nothing. |
| VOID setUses (STMTNUM s, VARINDEX\_LIST varList);  Description: Set the Uses relationship between s and variable index inside varList to be true in the usesTable.  If either variable indexes inside the varList or s given is out of range, do nothing. |
| VOID setUsesProcedures (PROCINDEX procIndex, PROCINDEX\_LIST procList);  Description: Set the Uses relationship between procIndex and the procedure indexes inside the procList to be true.  If any of the procedure indexes in procList or procIndex given is out of range, 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. |

## 10 CALLS and CALLS\*

|  |
| --- |
| **Calls**  *Overview*: Calls is used to keep track of the Calls relationship between procedures. |
| **API:** |
| VOID setCalls (PROCNAME procCall, PROCNAME procCalled, STMTNUM s);  Description: Set the Calls relationship between procCall and procCalled to be true. The statement number s where the calls is invoked will be stored as well.  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. |
| PROCINDEX\_LIST getCalls();  Description: Returns all procedures that calls other procedures directly.  If no procedure if found, return an empty vector. |
| PROCINDEX\_LIST getCalls(PROCNAME procCalled);  Description: Returns all procedures that calls procCalled directly.  If no procedure if found, return an empty vector. |
| **PROCINDEX\_LIST getCalled();**  Description: returns all procedures that are called by other procedures directly.  If no procedure if found, return an empty vector. |
| PROCINDEX\_LIST getCalled(PROCNAME procCalls);  Description: returns all procedures that are called by procCalls directly.  If no procedure if found, return an empty vector. |
| PROCINDEX\_LIST getCallsStar(PROCNAME procCalled);  Description: returns all procedures that calls procCalled indirectly or directly . |
| PROCINDEX\_LIST getCalledStarBy(PROCNAME procCalls);  Description: returns all procedures that are called by procCalls indirectly or directly. |

## 11 Design Extractor

|  |
| --- |
| **Design Extractor**  *Overview*: The following API is responsible for extracting more relationships from the AST and the Calls Relationship before storing them in the PKB. |
| **API:** |
| VOID extractorDriver(PKB);  Parameters:  PKB: contains all the data structures created and stored after parsing the SIMPLE code.  Description: Extracts relationships using Calls table and also constructs the CFG. |
| VOID extractRelationships(AST, Calls, ProcTable, Modifies, Uses, Parent);  Parameters:  AST, Calls, ProcTable, Modifies, Uses, Parent – pointers to the respective data structures in the PKB  Description: Extracts relationships such as modifies and uses for procedures and for program lines that call procedures. Stores these relationships in the Modifes and Uses in the PKB as necessary. |
| NODE buildCFG(AST);  Parameters:  AST: pointer to the AST stored in PKB  Description: Given the AST built by Parser, the Design Extractor will construct the CFG and return a pointer to the root Node of the CFG. |
|  |

## 12 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. |

## 13 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. |

## 14 QueryParser

|  |
| --- |
| **QueryParser**  *Overview*: Parser is responsible to read the query code, tokenize the appropriate sentences, and build a Query class structure to be used by QueryEvaluator |
| **API:** |
| **Query queryParse(STRING queryStr, BOOLEAN\_PTR valid);**  Parameter:   * queryStr – the query being passed into the program.   (e.g. “assign a; while w; Select a such that Follows(w, a) pattern a (“x”, \_”x+y”\_)”)   * valid – is a Boolean pointer to indicate whether the query string is valid or invalid   Description: queryParser reads in the query string, tokenizes and detects the structure of the query, create and store the relationship clauses, create and return a Query class structure. |