CS3203 Iteration 1 Project Report

Team 18 AY19/20-S1

Ang Jing Zhe (A0154853Y)
Bobbie Soedirgo (A0181001A)
Daniel Tan Joon Kai (A0156672X)
Ng Eng Sheng (A0156028H)
Leon Chow Wenhao (A0156052M)
Richard Goh Chang Rui (A0139819N)

Contents

1. Scope of the Prototype Implementation 4

2. Development Plan 5

3. SPA Components Design 10

3.1 SPA Front-end Design 10

3.1.1 List of functions 10

3.1.2 More Details on Statement Type Handling/Parsing 11

3.1.3 SIMPLE Source Code Validation 12

3.1.4 Design Choice Rationale 13

3.2 Program Knowledge Base (PKB) 14

3.2.1 Internal Data Structure 14

3.2.2 PKB Components 16

3.2.3 Design Rationale for Advanced Relationship Tables 17

3.2.4 Relationship table without inverse table 18

3.3 Design Extractor 19

3.3.1 Follows* Relationship 19

3.3.2 Parent* Relationship 20

3.2.3 Design Rationale for Populating the FollowsStarTable and ParentStarTable 21

3.4 Query Validation and Processing 22

3.4.1 List of functions in Query Parser 22

3.4.2 More details on parsing queries 23

3.4.3 Query Validation 24

3.4.4 Design Consideration 25

3.5 Query Evaluator 26

3.5.1 Data Representation of a Query 26

3.5.2 Method of Evaluation 26

3.5.3 Evaluator Example 28

3.5.4 Design Consideration 30

4. SPA Component interactions 31

4.1 Source Code Parser and PKB 31

4.2 Query Evaluator and PKB 31

5. Testing 33

5.1 Test plan 33

5.2.1 Source Parser Unit Test 34

5.2.2 Query Parser Unit Testing 36

5.2.3 System Testing 38

6. Documentation and Coding Standards 40
6.1 Abstract APIs Naming Convention 40
6.2 Enhancing correspondence between abstract APIs and the concrete API counterparts 41
6.3 Coding standards and style adopted 41

7. Abstract API Documentation 42
7.1 PKB Statement and Variable Table Operations 42
7.1.1 varTable 42
7.1.2 stmtTable 42
7.1.3 whileTable 43
7.1.4 ifTable 43
7.1.5 printTable 44
7.1.6 readTable 44
7.1.7 constantTable 45
7.1.8 ProcTable 45
7.2 PKB Relationship Table Operations 46
7.2.1 stmtModifiesByVarTable 46
7.2.2 varModifiesByStmtTable 47
7.2.3 stmtUsesByVarTable 48
7.2.4 varUsesByStmtTable 49
7.2.5 assignStmtTable 49
7.3 Design Extractor Table Operations 50
7.3.1 FollowsTable 50
7.3.2 followsStar Table 51
7.3.3 parentTable 51
7.3.4 parentStarTable 52
7.4 Higher Level Methods / Wrappers 53

8. Discussion 54
8.1 Problems Faced 54
8.2 Lessons Learnt 54

1. Scope of the Prototype Implementation

Team 18 project prototype is able to handle iteration 1 requirements.

The goal of this iteration was to create a working system from scratch in order to handle a subset of the full SPA requirements. The reduced SPA requirements helped us rigorously test our system for the correctness of this iteration's SPA requirements instead of flooding us with an overwhelming number of features to be implemented into the system.

In summary, the prototype system is able to handle:

- **SIMPLE Source Program:**
    - Procedure statement parsing (only 1).
    - Read | print | assign | if | while statement handling.
    - Modifies | Uses | Parent | Follows | Parent* | Follows* relations capturing for the above statement types handled.

- **PQL Queries:**
    - Handle multiple declarations of procedure | statement | assign | read | print | while | if | variable | constant synonyms.
    - Handle 0 | 1 | 2 clauses (no clause, 1 such that clause, 1 pattern clause or 1 such that and 1 pattern clause).

The system was designed to suit the specific requirements for this iteration. For further iterations and additional features required of our system, the design of our system could change accordingly to suit the new requirements.

2. Development Plan

2.1 Project Management Tools

Team 18 will be using GitHub as proposed by the project requirement, for the code repository.

Team 18 is using Asana for its project management tool, Asana gave us the tools that we needed to get organised, planning and structure our priorities with its board layout feature. With the board layout feature, we can easily keep track of each member's weekly progress and see the backlog of each member are having. This visual board layout gives definitive view of what has been done and which are the tasks that the team members are currently working on. The team leader then has a clear indication on how to indicate man power in the following mini iteration week based on each team member workload.

In addition, we also used Google Suite Products (Slides, Docs) for all other project related documentation and presentations. Google Docs allows the team to collaborative work on APIs documentation, it also allows us to keep a single source of truth and shows the most up to date APIs documentation. This is important for the team members who are using those APIs, to ensure that they are using the updated APIs if there are to be any changes. While Google Slides allow the team to collaborative work on the weekly consultation presentation.

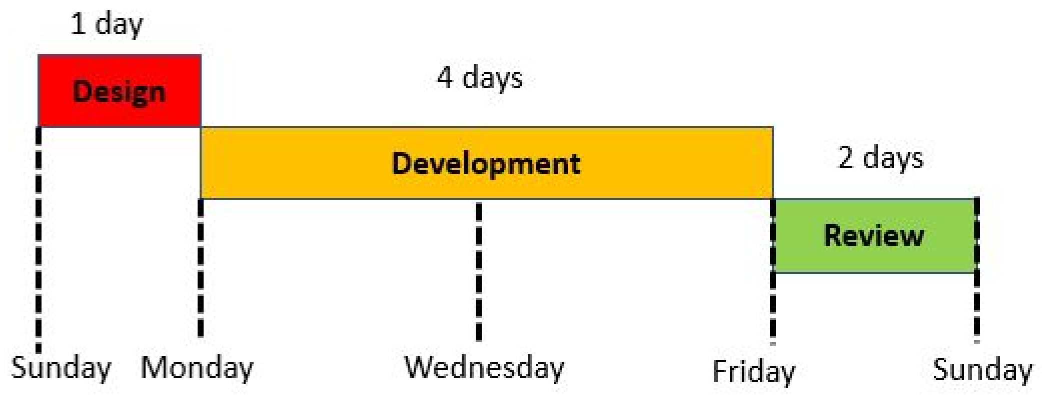
2.2 Software Development LifeCycle (SDLC)

Team 18 will be using iterative breadth-first model. We chose the breath-first model as it allows us to develop by functionality, developing all the components in parallel as well as continuous integration. This helped us achieve our mini iteration goals we set for each week.

2.3 Mini Iteration (Weekly Schedule)

In team 18 development plan, we chose to do weekly mini iterations. In each mini iteration, it is split into 3 sections, design, development and review. This allows us to integrate our codes and ensure the functionality for each component works together. In addition, mini iteration help us to dynamically shift roles and man-power needed per weekly basis.

We structured our weekly schedule which starts on a Sunday and a review on Friday.



Design	1 Day	Plan and design features that needs to be done in the weekly mini iteration. We want the team to design their implementation before coding straight away. A good design can reduce the amount of codes to be written significantly.
Development	5 Days	Coding Phase, which includes Wednesday for project consultation at the halfway mark. This gives enough time for the team to implement halfway to seek feedback from the consultation tutor as well as enough time to make changes for the weekly mini iteration review.
Review	2 Days	Code integration, and code review, ensures all task is completed and done. System testing and integration testing are done to ensure that the new features implemented are working as intended. Regression testing is also done with the previous iterations test cases to ensure current code base does not break previous mini iteration features. These 2 days includes fixing of bugs as well, reported by other team members.

2.4 Iteration 1 Schedule

The duration of each mini iteration is a week. We have a total of 3 mini iterations and a design week at the start. The design phase week lasted from 1st September to 7th September. The design phase week allow each team member to carefully plan out their components and design the API that are needed.

The main objective for each mini iteration is defined in the table as shown below:

Iteration 1			
Component	**Mini Iteration 1 (8 Sep - 14 Sep)**	**Mini Iteration 2 (15 Sep - 21 Sep)**	**Mini Iteration 3 (22 Sep - 26 Sep)**
Query Parser, Evaluator	0 Clause	0|1 Clause	0|1|2 Clause (such that & pattern)
Source Parser,	All the zero clauses	Uses, Modifies Parent, Follows	
Design Extractor			Parents*, Follow*

1. In mini-iteration 1, the objective is to integrate all components and the program is able to evaluate 0 clause queries successfully.
    - Source parser is able to parse the source program and insert 0 clause data (variable name, stmt number) into the PKB
    - PKB able to populate the 0 clause data (stmt, variable, assign, print, read, procedure, while, if)
    - Query Parser is able to extract declaration and matching synonyms

2. In mini-iteration 2, the program is able to evaluate 0 | 1 clause queries successfully.
    - Source parser is able nested source program insert data (Uses,Modifies,Parent, Follows) relationship into the PKB
    - PKB able to populate the single clause data (Uses,Modifies, Follows,Parent) relationship
    - Query Parser is able to extract such that clause and pattern clause
3. In mini-iteration 3, the program is able to evaluate 0 | 1 | 2 clause queries successfully.
    - Source parser is able to do validation of invalid source program
    - PKB able to populate the single clause data (Follows*, Parent*) relationship
    - Design Evaluator able to populate the PKB for (Follows*, Parent*) relationship
    - Query Parser is able to do validation, check for incorrect query syntax.

Mini Iteration 1 (8 Sep - 14 Sep)			
Team Member	**Task**	**Activity**	**Time Taken**
Daniel	Create functions for Source Parser to read source program and insert 0 clause data and Uses and Modifies data to the PKB	• Created functions for Source Parser to read and tag the correct statement number to each line.	8 - 9 Sep
		• Created functions for Source Parser to parse procedure statements, read statements, print statements and assign statements, for assign statements with only 1 variable or constant on the right hand side. Populates PKB varTable, usesTable and modifiesTable accordingly.	9 - 14 Sep
Eng Sheng	Create APIs to insert and store 0 clause data	• Created get and set abstract APIs for 0 clause data (stmt, variable, assign, constant) • Created abstract data types for abstract APIs	8 - 9 Sep
	Create APIs to insert and store Uses and Modifies data	• Created get and set abstract APIs for Uses and Modifies relationship • Created inverse table for uses and modifies for faster retrieval	9 - 12 Sep
	Conduct System Testing	• Integrate all components and solve merge conflicts • Conduct system testing with the 0 clause test cases	13 Sep
Richard	Create APIs to insert and store 0 clause data	• Created get and set abstract APIs for 0 clause data(assign, ifs, while, proc)	9 - 12 Sep
Jing Zhe	Create function that allows the Query to parse queries and 1 such that clause without validation	• Created functions for Query Parser to parse declaration statements of all types and 1 such that claus without validation	8 - 9 Sep
		• Created functions for Query Parser to send the query object to the Evaluator, containing declaration variables, selected variables and the clauses in such that and pattern.	9 - 12 Sep
Bobbie	Setup Git Management	• Setup .git ignore file to ignore debug files and compiled files.	8 Sep
	Create functions for Evaluator handle 0 clause	• Created functions for Evaluator to query 0 clause (stmt, variable, assign, ifs, while, constant, procedure)	8 - 12 Sep
Leon	Create Test Cases for 0 clause system testing	• Created a basic source program • Created 7 basic 0 clause queries	7-8 Sep

Mini Iteration 2 (15 Sep - 21 Sep)			
Team Member	**Task**	**Activity**	**Time Taken**
Daniel	Create functions for Source Parser to read source program with nested if and while, and insert single clause data to the PKB	● Created functions for Source Parser to parse full range of assign statements.	16 - 17 Sep
		● Created functions for Source Parser to parse if and while statements along with their statement lists.	17 - 19 Sep
		● Created functions for Source Parser to populate uses and modifies relation for if and while statements and able to handle nested if and while statements.	19 - 20 Sep
		● Created functions for Source Parser to populate all Parent and Follows relations correctly.	20 - 21 Sep
Eng Sheng	Create APIs to insert and store single clause data	● Created get and set abstract APIs for Follows and Parents relationship ● Created abstract APIs to check if Uses,Modifies,Follow,Parent relationship is true	16 - 18 Sep
	Conduct System Testing	● Integrate all components and solve merge conflicts ● Conduct regression testing with the 0 clause test cases ● Conduct system testing with single clause queries ● Debug single clause test queries expected answer	20 - 21 Sep
	Create Test Cases for single clause system testing	● Created a single nested source program and 57 Modifies queries	20 Sep
Richard	Create APIs to insert and store pattern information for assignments	● Created abstract APIs to check if constant or variable is used in RHS of assignment statement.	20 - 21 Sep
	Create higher level function wrappers	● Created abstract APIs to populate multiple main table and inverse table with wrapper methods	20 - 21 Sep
Jing Zhe	Create function that allows the Query parser to parse pattern clauses without validation	● Created functions for Query parser to identify pattern clause in the query and parse the pattern to evaluator	16 - 18 Sep
Bobbie	Conduct System Testing	● Debug single clause test queries expected answer	20 - 21 Sep
	Create functions for Evaluator to handle single clause	● Created functions for Evaluator to query single clause (Uses, Modifies)	16 - 18 Sep
		● Created functions for Evaluator to query single clause (Parent, Follows)	18 - 20 Sep
		● Created functions for Evaluator to query single clause (Pattern)	20 Sep
Leon	Create Test Cases for single clause system testing	● Created a double nested source program and 30 Follows and 43 Parent queries ● Created a single nested source program and 57 Uses queries ● Created a triple nested source program and 135 single clause queries	16 - 21 Sep

Mini Iteration 3 (22 Sep - 26 Sep)			
Team Member	**Task**	**Activity**	**Time Taken**
Daniel	Create functions for Source Parser to do validation	• Created functions for Source Parser to do some validation such as checking if each line has no more than one ";".	22 - 24 Sep
Eng Sheng	Create APIs to insert and store Follows* and Parents* relationship	• Created get and set abstract APIs for Follows* and Parents* relationship	22 Sep
	Create functions for Design Extractor to populate the PKB with follows* and parent* relationship	• Created functions for Design Extractor to populate the PKB with follows* and parent* relationship	22 Sep
	Conduct System Testing	• Integrate all components and solve merge conflicts • Conduct regression testing with the 0 and single clause test cases • Conduct system testing with single and double clause queries • Debug single clause test queries expected answer	24 - 26 Sep
Richard	Conduct Integration Testing	• Conducted integration testing between PKB and Program Parser	22-24 Sep
Jing Zhe	Create functions for Query parser to do validation to check for invalid syntax	• Created functions for Query Parser to validate invalid declaration types, clause types and variable naming in queries • Created functions for Query Parser to validate variables in clauses that are not declared	22 - 24 Sep
Bobbie	Create functions for Query validation	• Implemented additional query validation	23 - 26 Sep
	Conduct System Testing	• Debug multiple system tests (Follows, nesting, etc.)	24 - 25 Sep
	Create functions for Evaluator to handle pattern clause and double clause	• Created functions for Evaluator to query single clause (Follows*, Parent*)	22 Sep
		• Created functions for Evaluator to query double clause	22 - 23 Sep
Leon	Create Test Cases for single clause and double clause system testing	• Created a 3 source program for test cases submission and a total of 117 mixture of 0, single, double clauses queries • Created a double nested source program and 59 Parent* queries • Created a single nested source program and 63 Follows* queries	22 - 24 Sep
		• Created a single nested source program and 86 pattern queries • Created double nested source program and 260 double clause queries	24 - 26 Sep

3. SPA Components Design

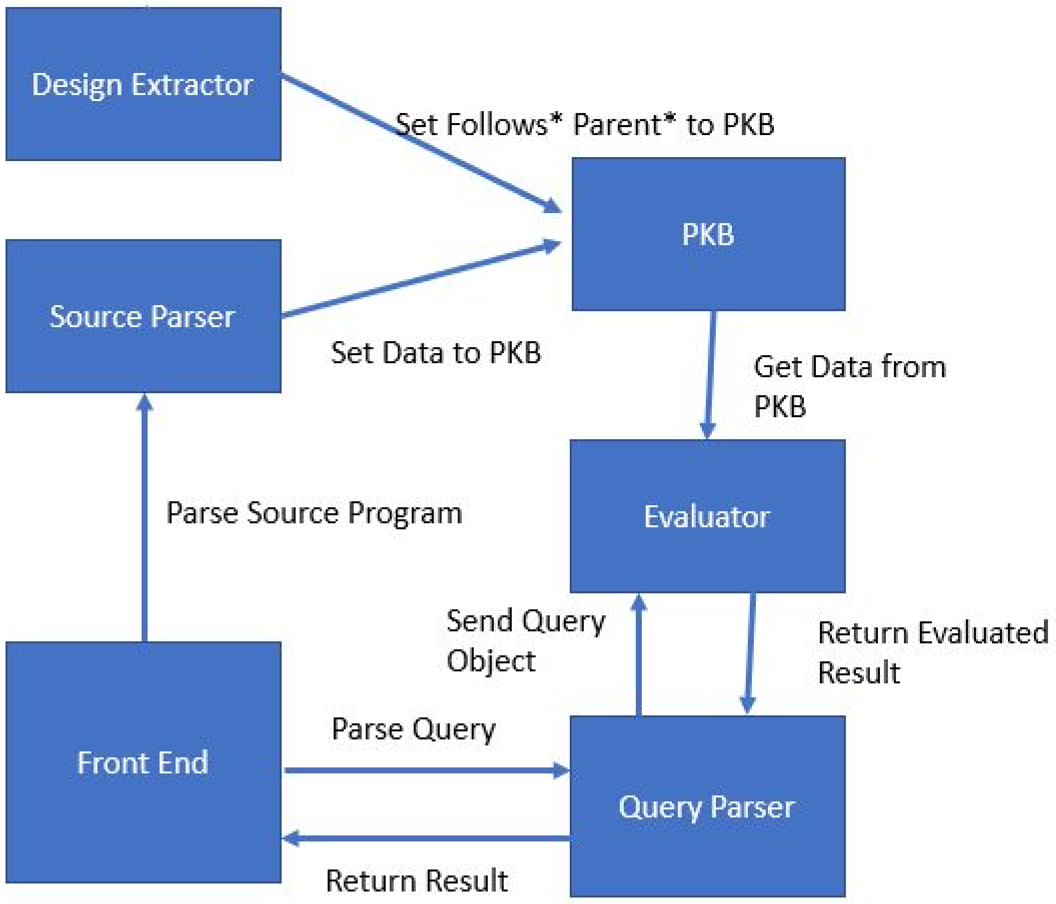
3.0 Overview

The SPA front-end design will parse in a source program text file. This text file will then be read by the Source Parser, the source Parser will do its validation to check for any invalid source program before doing the parsing. The source program will read line by line and evaluate which PKB setter APIs to call to insert data into the PKB.

The Design Extractor will then call the getters APIs from the PKB and then use the result to reinsert it into the PKB for Follows* and Parent* relationship table by using the setters APIs from PKB.

After the PKB has been populated by the Design Extractor, the front-end design will also parse the query into the Query Parser, the Query Parser will read the string do its validation to check for any incorrect query syntax, if the query syntax is invalid it will return empty string back as a result, otherwise it will construct a query object based on the query to the Evaluator.

The Evaluator will extract out information on the query object. Based on the query object, it will evaluate those clauses and calls the getters APIs from the PKB. The PKB will then return the results to the Evaluator. The Evaluator will then evaluate the query result after getting all the results it needed from the PKB. After which, the Evaluator will return the result to the Query Parser and the Query Parser will return the result back to the front end.


Design Extractor
Set Follows* Parent* to PKB
PKB
Source Parser
Set Data to PKB
Get Data from PKB
Evaluator
Parse Source Program
Send Query Object
Return Evaluated Result
Front End
Parse Query
Query Parser
Return Result

3.1 SPA Front-end Design

The front-end parser has an iterative design that parses the SIMPLE source code using a top-down approach. At each iteration, the high level design of the parser is seen as follows:

1. Determine the statement type.
2. Calls PKB API to populate the statement type along with its statement number.
3. Parses the statement according to the statement type.
4. Populates Uses, Modifies, Parents and Follows relations accordingly.

After the parsing has been done, the design extractor is then called to populate the Follows* and Parents* relations into the PKB.

3.1.1 List of Functions

1. **parseProc** - Tokenizes the procedure name.
2. **parseRead** - Tokenizes the read variable.
3. **parsePrint** - Tokenizes the print variable.
4. **parseAssignInit** - Does white space and “;” removal from the assign statement.
5. **parseAssignRHS** - tokenizes the variables and constants on the RHS of an assign statement.
6. **parseIf** - This handles the parsing of the statements inside the then and else statement lists of the If statement. The modifies and uses relations that are populated for the statements in the statement list are stored in an intermediate object, NestedResult, and the object is passed back to be populated for the parent statement also.
7. **parseIfNestedInThen** - This serves the same function as parseIf but is meant to handle the specific case of if statements nested in a then statement list.
8. **parseWhile** - This handles the parsing of the statements inside the while statement lists of the If statement. The modifies and uses relations that are populated for the statements in the statement list are stored in an intermediate object, NestedResult, and the object is passed back to be populated for the parent statement also.
9. **parseWhileNestedInThen** - This serves the same function as parseWhile but is meant to handle the specific case of while statements nested in a then statement list.
10. **parseCondStmt** - Tokenizes the cond_expr in the if/while statement.
11. **parseCondition** - Tokenizes the rel_expr in the cond_expr.

3.1.2 Example of Tokenization

Simple tokenization of statement types that would definitely only have 1 token, such as procedure, read and print statements return a string representing the token. However, for other statement types a vector is used to capture all the tokens that have been tokenized.

Given the example of an assign statement: “a = b + (c - (d / e));”, the tokenizing process is as such:

1. Remove all white spaces and “;” character.
2. Splits into LHS and RHS around the “=” character.
3. LHS (“a”) will be set as a modifies variable.
4. Remove all parentheses from RHS (b+(c-(d/e))).
5. Iterate through finding the next immediate operation (+|-|*|/|%) and tokenizing each item on the LHS of the operation.

Given another example of a cond_expr: “((a > b) || (c != (d + e)))”, the tokenizing process is as such:

1. Remove all white spaces.
2. Checks if it is just 1 rel_expr.
3. For every rel_expr, splits by the conditional expression (==|!=|>=|<=|>|<), and calls the same function used to tokenize the RHS of assign statements to tokenize the LHS and RHS of the rel_expr.

3.1.3 More Details on Statement Type Handling/Parsing

The sequence of how the Parser handles each statement type is as follows, it is notable that before the parser starts to parse each line, it will perform a simple validity check to ensure that each line contains no more than one “;”:

- Procedure:
    1. Tokenizes procedure name.
    2. Populates the procName into the PKB.
    3. No statement number assigned to this line.
- Read/Print:
    1. Populates PKB with respective statement type to the statement number.
    2. Tokenizes read/print variable.
    3. Populates PKB with the read/print variable and their corresponding uses/modifies relation.
    4. Populates PKB with follows relation for the previous statement in the statement list to this statement.
- Assign:
    1. Populates PKB with assign statement type to the statement number.
    2. Tokenizes the assign variables.
    3. Populates PKB with the assign variables and their corresponding uses/modifies relation.
    4. Populates PKB with follows relation for the previous statement in the statement list to this statement.
- If/While:
    1. Populates PKB with respective statement type to the statement number.
    2. Tokenizes the cond_expr in the statement.
    3. Enters a function call to parse statements within the if/while statement list.
    4. For each statement, parse it accordingly (similar to how parsing is described in this section) while storing all the tokenized variables/constants in an intermediate NestedResult object.
    5. For each statement, also populate PKB with Parent relation with the parent statement.
    6. Terminates function call when end condition for if/while loop has been reached.
    7. Populates PKB uses and modifies relation for parent statement accordingly from NestedResult object.
    8. Populates PKB with follows relation for the previous statement in the statement list to this statement.

3.1.4 SIMPLE Source Code Validation

There was not much validation implement for the Source Parser during this mini iteration. Some simple validations were done:

- Checks that each line contains no more than one ";".
- The parser only parses lines upon detection of specific keywords, if none are found, the line is assumed to be a blank line and not parsed.
- When parsing expr, checks if the first char of each tokenized result is a digit, if it is, then the token is assumed to be a constant, if it is not, then the token is assumed to be a variable name.

3.1.5 Design Choice Rationale

This was the first iteration and the parser had to be implemented from scratch. The possible design options of this solution was to either use an iterative top-down approach, or a 2 step approach with the separation of statement lists and then the parsing of the individual statement lists. We came up with 3 criterias of evaluation in order to determine which design option would be best for the source code parser. Our analysis as seen in the table below:

Method	**Criteria 1: Ease of implementation**	**Criteria 2: Expandability**	**Criteria 3: Efficiency**
2 step approach	This method would be comparatively harder to implement and would require more time and effort to design.	This method would be more modular and comparatively easier to expand.	In terms of efficiency, both methods are similar.
Iterative top-down approach	This method would be easier to implement.	This method would be harder to expand upon when more requirements come in such as the call statement type.	In terms of efficiency both methods are similar.

A decision was made to ignore the third criteria after one of our consultations when it was mentioned that the efficiency of the source code parser was not going to be evaluated. Other factors were considered too, such a tight submission schedule of three weeks. Hence, the main criteria in selecting a design option for the parser during this iteration was the ease of implementation of the solution. From the analysis above and the decided upon main criteria, the design option selected was the iterative top-down approach as it was the simplest way to implement the source code parser for this prototype.

3.2 Program Knowledge Base (PKB)

The purpose of the PKB is to encompass design extractions representing important relationships within the source program and store relevant data. It is a key component that interacts primarily with the Program Parser and the Query Evaluator. During the parsing of the source program, **setter methods** of the PKB will be called by the Program Parser to build design abstractions by populating internal data structures. During evaluation of queries, one or more **getter methods** of the PKB will be called by the Query Evaluator to retrieve information necessary to obtain the query result.

3.2.1 Internal Data Structure

Abstract Representation of PKB Components

table1Title
<ABSTRACT_DATA_TYPE_A>
key1
key2

table2Title	
<ABSTRACT_DATA_TYPE_B>	<ABSTRACT_DATA_TYPE_C>
key1	value1
key2	<value2, value3>

In general, PKB components can be represented using either single column or double column table-like data structures. Internally, these components are formed using C++ 's **unordered sets**, and **unordered maps** respectively. For single column tables, the key and value are the same, while for double column tables the keys of the hashmap form the left column and their corresponding value(s) form the right column. This internal data structure is **used for all tables** within the PKB, and only vary in terms of the data type of the key and values in the table. For example, the **constantTable** would internally be a **unordered_map<CONS_NAME, STMT_LIST>** which translates to **unordered_map<string ,unordered_set<int>>** data structure. Whereas the **assignStmtTable** is internally made up of a **unordered_map<STMT_NO, VAR_NAME>** which translates into **unordered_map<int, string>**

Choice of Internal Data Structure

When deciding on the appropriate data structure for the internal implementation of our table-like data structures, we narrowed our choices down to choosing between **hashmaps**, **vectors,** or **lists** offered by C++, each of which have their own pros and cons.

	Unordered Maps/Set	**Vector**	**List**
Internal Implementation	Hashmap	Array	Doubly-linked Linked List
Retrieval Method	Accessed by value	Accessed by index	Accessed by index
Time Complexity (Retrieval)	**O(1)** (with the help of inverse tables)	**O(n)**	**O(n)**
Time Complexity (Insertion)	**O(1)** on average O(n) in worst case.	**O(n)**	**O(1)**
Time Complexity (Update)	**O(1)** on average	**O(n)** O(log n) if sorted	**O(n)**
Time Complexity (Iteration through table)	**O(n)**	**O(n)**	**O(n)**
Time Complexity (Deletion)	Irrelevant in SPA context.		

Eventually, a decision was made to use hashmaps (**unordered_set**, and **unordered_maps**) for the PKB's internal structure due to the factors listed above. Firstly, hashmaps allow for fast retrieval of individual elements based on their value, which is an important factor since the Query Evaluator would likely need to query the PKB based on certain values rather than index. Moreover, the hashmap implementation is faster or equal to a vector or a list for the most operations (insert, retrieve, update).

However, the team is aware of possible inefficiencies of such an internal implementation in terms of query processing efficiency for more complex queries from Iteration 2 onwards. It is likely that the PKB internal implementation and the Query Evaluator's algorithm would undergo significant changes post Iteration 1.

3.2.2 PKB Components

For the following section, the contents of PKB will be a simulation of as though the sample source code below is parsed.

SIMPLE program:

```
_    procedure main {
1.       x = 1
2.       y = x
3.       if (x == y) then {
4.           z = 0;
_        } else {
5.           z = 1;
_        }
6.       while (x > 0) {
7.           read x;
8.           if (y==z) then{
9.                   print x;
             }
             else{
10.                  print y;
.            }
_        }
```

Storing Variables, Constants and Procedure Names

varTable
<VAR_NAME>
x
y
z

constantTable	
<CONS_NAME>	<STMT_LIST>
1	1,5

procedureTable
<PROC_NAME>
main

For storage of variables, constants and procedure names, simple lists are created to facilitate easy retrieval of the entire table at once for simple queries such as **getAllVar()** or **getAllProc()**. In addition, statement lists are attached as values as part of a key-value pair relationship for the constant table to locate constants within the program.

Storing Statement Types

For storage of statement types, the PKB utilises 6 tables to represent the various statement types (less callTable for iteration 1).

stmtTable	
<STMT_NO>	<STMT_TYPE>
1	assign
2	assign
3	if
4	assign
5	assign
6	while
7	read
8	if
9	print
10	print

readTable	
<VARIABLE>	<STMT_LIST>
x	**7**

assignStmtTable	
<STMT_NO>	<VAR_NAME>
1	x
2	y
4	z
5	z

printTable	
<VAR_NAME>	<STMT_LIST>
x	9
x	y

whileTable
<STMT_NO>
6

ifTable
<STMT_NO>
3
8

The **stmtTable** stores basic information about every single statement in the program and tags it to a statement type. The table is typically populated immediately through an API call by the program parser once the parser detects the statement type and line number. The **readTable** stores additional information about read statements, including the variable that is being read (var modified) in the first column, as well as all the statements in which this read statement appears. Similarly, the **printTable** stores additional information about print statements, including the variable that is being printed(var used) in the first column, as well as all the statements in which this print statement appears.

The **assignStmtTable** stores additional information about assign statements, with the statement number on the left column and theon LHS of '=' in an assignment statement (i.e. the variable that is modified). Lastly, the **whileTable** and **ifTable** stores lists of statement numbers in which the while and if statements appear respectively.

<u>Storing Advanced Relationship</u>

The following tables help the PKB to encapsulate relationships between statements and/or variables within the source program, by storing variables and statements in a meaningful order in one table for each relationship.

followsTable	
<STMT_NO>	<STMT_NO>
1	2
2	3
3	6
7	8

followsStarTable	
<STMT_NO>	<STMT_LIST>
1	2,3,6
2	3,6
3	6
7	8

parentTable	
<STMT_NO>	<STMT_LIST>
3	4,5
6	7,8
8	9,10

parentStarTable	
<STMT_NO>	<STMT_LIST>
3	4,5
6	7,8,9,10
8	9,10

The **follows, followsStar, parent, and parentStar** table stores entries that are mappings between a statement number and another set of statement numbers within the program. The relationship of this mapping is represented by the table title.

For the **followsTable** above, the first entry of **<1,2>** can be read as meaning that **Follows(1,2)** is true since they are at the same nesting level and are in the same statement list and **#2** appears in the program text immediately after **#1**. Similarly, in the **followsStarTable** above, the first entry of **<1,<2,3,6>>** can be read as Follows*(1,2), Follows*(1,3) and Follows*(1,6) to be true. By visual inspection, we can confirm that that is true, since Follows(1,2) is true and Follows(2,3) is true and Follows(3,6) is true and hence by transitive closure Follows*(1,6).

A similar interpretation can be made for the **parentTable** and **parentStarTable** for the parent and parent* relationship respectively. The first entry <3,<4,5>> in the **parentTable** translates to parent(3,4), parent(3,5) being true, and so on for the second and third entry.

The second entry <6,<7,8,9,10>> in the **parentStarTable** translates to parent*(6,7), parent*(6,7), parent*(6,8), parent*(6,9), parent*(6,10)being true. Similarly, by visual inspection of the **parentTable** we can verify these relationships through transitive closure. The parent and parent* relationships are populated by the design extractor.

stmtModifiesByVarTable	
<VAR_NAME>	<STMT_LIST>
x	1,7
y	2
z	4,5

stmtUsesByVarTable	
<VAR_NAME>	<STMT_LIST>
x	2,3,6,9
y	3,8,10
z	8

varModifiesByStmtTable	
<STMT_NO>	<VAR_LIST>
1	x
2	y
4	z
5	z
7	x

varUsesByStmtTable	
<STMT_NO>	<VAR_LIST>
2	x
3	x,y
6	x
8	y,z
9	x
10	y

The **stmtModifiesByVarTable** stores entries that are mappings between a variable that is modified and a set of statement numbers within the program in which this modification takes place. The **varModifiesByStmtTable** is in an inverse of the **stmtModifiesByVarTable** with the exact same information to facilitate efficient information retrieval. For example the first entry of the **stmtModifiesByVarTable,** <x,<1,7>>, can be interpreted as Modifies(1,"x") is true and Modifies(7,"x") is true.

The stmtUsesByVarTable stores entries that are mappings between a variable that is used and a set of statement numbers within the program in which this usage takes place. The **varUsesByStmtTable** is in an inverse of the **stmtUsesByVarTable** with the exact same information to facilitate efficient information retrieval. For example the first entry of the **stmtUsesByVarTable,** <x,<2,3,6,9>>, can be interpreted as Uses(2,"x"), Uses(3,"x"), Uses(6,"x"), Uses(9,"x") is true..

3.2.3 Design Rationale for Inverse Relationship Tables

In the PKB, every two column table has an inverse table with the exact same information to facilitate efficient information retrieval. This is an intentional design decision made to prioritise information retrieval speed over space complexity and data redundancy.

	Criteria 1: **Efficiency of Retrieval Process by Query Evaluator** **(Most important)**	**Criteria 2:** **Ease of Implementation**	**Criteria 3:** **Space Complexity and Data Redundancy** **(Least important)**
Without inverse Tables	Slower retrieval speed, due to frequent traversal of entire tables.	Easy to implement with almost half the number of tables.	Minimum data redundancy. Smaller space.
With inverse Tables	Faster retrieval since number of traversals are greatly reduced.	More difficult to implement since the total number of tables is almost doubled. Due to the high amount of duplicated fields, extra effort is required to ensure data consistency across the different tables.	High amount of data redundancy since several fields (e.g. variables) are repeated in two or more tables. Larger space.

In all, since the SPA requirements key evaluation criteria focuses on the speed of the query evaluator, data redundancy is less significant in this context. Moreover, the space complexity can be considered as negligible since these data structures are not expected to be extremely large in size (<5000 rows) in the worst case, and can be comfortably stored in the PC’s memory.

3.2.4 Relationship table without inverse table

In the PKB, some relationship table such as parentTable, parentStarTable, followsTable, followsStarTable and assignStmtTable does not have an inverse table. This is because the evaluator only uses the isRelationship type of API calls to do the evaluation, as such there is no need for the PKB to have an inverse table for those relationship checks.

3.3 Design Extractor

The design extractor's role for this iteration is to populate the Parent* and Follows* relation. The design extractor's implementation is a simple recursive algorithm that finds a statement's children* of its children and followers* of its followers to populate the Parent* and Follows* relation respectively. The information needed are retrieved from the Parent and Follows tables in the PKB, and the PKB API are called when populating Parent* and Follows*.

3.3.1 Follows* Relationship

FollowsTable

followedBy	follows
1	2
2	4
4	5

Follows*Table

followedBy	follows*
1	2,4,5
2	4,5
4	5

The sequence of how the Design Extractor populates the Follows* Table in the PKB as follows:

1. The Design Extractor do an API call to PKB to get all the followedBy statement number from the PKB, and iterate it through in a loop
2. For every followedBy statement number, it does an API call to the PKB to insert the Follows statement number from the FollowsTable corresponding to the followedBy statement. As such, statement number 1 is inserted to the Follows* Table as followedBy and 2 is inserted as Follows*.
3. It then does a recursive call to get followers of its followers.

A recursive call pseudocode to populate Follows* to the PKB

```
recursiveFollow( followedBy, follows) {
        followsFollower = getFollows(follows);
        //followsFollower will be set to -1 if not found
        if (followsFollower does not exist in followTable)
                return;
        else {
                setFollows*Relationship;
                recursiveFollow(followedBy, followsFollower);
        }
}
```

Using the recursive call, with the base case to exit when it fails to find a follower's follower. We can set the Follows* relationship easily, as such it will find statement 4 and 5 to be set as follows* relationship to statement 1.

3.3.2 Parent* Relationship

We do the same sequence of how the Design Extractor populates the Follows* but this time for the Parent* relationship, along with a slight modification to the recursive call as we have a list in the child column.

ParentTable

parent	child
1	2,3,6
3	4,5

Parent*Table

parent*	child
1	2,3,6,4,5
3	4,5

A recursive call pseudocode to populate Parent* to the PKB

```
recursiveParent( parent, child) {
        childList = getChildrenStmtList(child);
        if (childList is an empty list)
                return;
        else {
                for all child in childList {
                        setParent*Relationship;
                        recursiveParent(parent, child);
                }
        }
}
```

Using the recursive call, with the base case to exit when it fails to find a child of a child. This allows us to set the Parent* relationship easily, as such it will find statement 4 and 5 to be set as Parent* relationship to statement 1.

3.2.3 Design Rationale for Populating the FollowsStarTable and ParentStarTable

The Design Extractor is able to populate the followsStarTable and parentStarTable through 2 ways. The first way is to use the existing tables followsTable and parentTable to iterate through a recursive call and populate the followsStarTable and parentStarTable. The other method is to create an AST and traverse through the AST to find the ancestors and descendants to populate the parentStarTable, as well as populating the followStarTable.

	Criteria 1: Ease of implementation	**Criteria 2: Efficiency**
AST Traversal	This method would require AST APIs to traverse through the AST. However, at this point of time, there are no existing AST APIs created. Hence it require the creation of AST APIs to be made and the Parser to populate the AST as well.	In terms of efficiency, AST is able to populate both relationship tables in a single AST traversal, hence it will be faster.
Recursive Call on Existing Tables	This method would be easy to implement as there are existing PKB APIs to get the required data and to set the data into the PKB	With the recursive call, it will take a longer time to fully populate the 2 relationships table. However, as there are no time-constraint being placed on setting the data. This factor is not a concern.

Therefore, as time-constraints with populating the PKB is not a priority of concern. And the PKB has existing APIs to make the recursive call implementation fairly simple and fast to do. We have decided to use the recursive call to populate the parentStarTable and followsStarTable.

3.4 Query Validation and Processing

The query parser has an iterative approach to parsing queries. At each iteration, query parser does these steps:

1. It validates the query by checking whether is it empty and existence of select statement
2. It splits out all declarations through the delimiter ';' and validates the declarations by checking through a list of valid declared types.
3. It splits out the selected variables by splitting the first half of the query string through finding the position of such that/pattern, if any. It will validate the select variables by checking the naming convention.
4. It splits out each such that/pattern clauses from the next half of the query string by repeated finding the positions of the next such that/pattern clauses. It will validate these clauses by checking through a list of valid clause types.
5. It will put declarations, selected variables and such that/pattern clauses (Item 2-4) into a query object and be sent to the evaluator.

More details will be covered in the subsections of the report.

3.4.1 Details of splitting and validation in Query Parser

The main method of splitting queries is to find the position of delimiters and substring the query accordingly. Delimiters can be ; or open/close brackets or whitespace. The delimiter to be used will depend on which section its splitting. An example will be shown in 3.4.2.

Validation checking is done mainly by checking against a predefined valid types by unordered set. The valid types in each set will depend on the section its validating. General rules of validation will be shown in 3.4.3.

Here are a list of functions Query Parser have used to split and validate query strings

Split query strings

1. **findInitialDeclaration -** Splits the declarations out from the query string through the delimiter ';'
2. **splitVariablesInDeclaration -** Make a mapping of variable to its variable type by finding the whitespace
3. **splitSelect -** Splits the selected variable out from the leftover query string by finding the index of the first such that/pattern clause
4. **splitSuchThat -** Splits clauses after such that through finding the positions of open bracket, comma and close bracket in each clause
5. **splitPattern -** Splits clauses in pattern by finding positions of open bracket, comma and close bracket in each clause.

Validate query strings

1. **initialValidation -** Checks whether the query is empty and the existence of 'select'
2. **declarationsValidation -** Checks the naming of variables and whether the declarations are valid types
3. **selectVariablesValidation** - Checks the naming of selected variables and whether the types of selected variables have been declared before
4. **suchThatValidation -** Checks the naming of variables in clauses and whether the clause types are valid
5. **patternValidation -** Checks the naming of variables in pattern and whether the pattern type is 'assign'

3.4.2 Example on parsing queries

An example of a valid query as follows:

```
stmt s; assign a; variable v; Select s such that Uses(a, v) pattern a(v, _)
```

The sequence of how the Query Parser handles a query as follows:

Get the declaration out

1. Through findInitialDeclaration, we will split the query by finding the delimiter ‘;’ and arrived at { “stmt s”, “assign a”,” variable v” }.

2. This will be parsed into splitVariablesInDeclaration to do mapping of variable to its variable type, we would have a map of { {“s”,”stmt”}, {“a”,”assign”},{“v”,”variable”} } by finding the whitespace between the variable and its variable type.

Get the such that clause

3. The leftover query string is now “Select s such that Uses(a,v) pattern a(v, _)”. The Query Parser will find the first index of “such that” and “pattern” and we will parse the substring into splitSelect.

4. In this case, “Select s“ is parsed into splitSelect and the result will be “s” for selectedVariable. The leftover query string is now “such that Uses(a,v) pattern a(v, _)”. It first detects “such that”, and gets the indexes and parse the substring “such that Uses(a,v)” into splitSuchThat.

5. In the function splitSuchThat, it finds the indexes of open bracket, comma and close bracket and makes a pair of {“Uses”, {“a”,”v”} }.

Get the pattern clause

6. The leftover query string is now “ pattern a(v, _)”. It detects that it’s a pattern clause, parse the string into splitPattern. In function splitPattern, it finds the indexes of open bracket, comma and close bracket and makes a pair of { {“a”, {“v”,”_”} }.

7. A query object containing declarations { {“s”,”stmt”}, {“a”,”assign”},{“v”,”variable”} } , selected variable “s”, such that clause {“Uses”, {“a”,”v”} } and pattern clause { {“a”, {“v”,”_”} } will be passed to the evaluator.

3.4.3 Query Validation

The Query Parser upholds some rules for validation checking:

1. All variables names are validated according to SPA grammar requirements given.
2. There should not be any empty variables in declarations and clauses.
3. Declaration types and clause types must be valid and specified according to SPA requirements given.

Variable names validations

Here are some invalid variable names: 1a2, a2*. It will be rejected in declarationsValidations, selectVariablesValidation or suchThatValidation/patternValidation, depending on where's declared. It first checks whether the 1st character is a letter, following which it will check the rest of the string whether it's alphanumerical.

Empty variables in declarations and clauses validation

Some examples of invalid declarations/clauses:{ {" ", "stmt"} }, {"Uses",{ , "v} }. It will be rejected in the respective validation section. In each validation section it will check whether are there empty variables.

Invalid declaration/clause types

Some examples of invalid declaration/clause types:{"c","char"}, {"Make", {"a","v"}}. These will be rejected in the respective validation section. In each validation section it will check these types against the defined valid types given in the requirements and will reject those that are not defined.

3.4.4 Data Representation of a Query

An example of a valid query as follows:

```
stmt s; assign a; variable v; Select s such that Uses(a, v) pattern a(v, _)
```

A query is represented as an object with 4 attributes:

Declarations	Select synonym	Such-that clause(s)	Pattern clause(s)
Unordered_map<string,string>	string	vector<pair string, pair<string,string>>>	vector<pair string, pair<string,string>>>
<table><tr><td>“s”</td><td>“stmt”</td></tr><tr><td>“a”</td><td>“assign”</td></tr><tr><td>“v”</td><td>“variable”</td></tr></table>	s	(“Uses”,(“a”,”v”))	(“a”,(“v”,”_”))

3.4.5 Design Rationale of Query Parser

In iteration 1, the query parser has to be implemented from scratch. The design consideration is the method used to split the string. The 1st method is the current method used, which is to split the string by finding positions of delimiter or token and substring them accordingly. Let's call the 1st method 'Delimiter splitting'. The second method thought of is to split by regular expressions, also commonly known as 'regex'.

Method	**Criteria 1: Ease of implementation**	**Criteria 2: Expandability**	**Criteria 3: Efficiency**
Split by regular expressions (Regex)	This method would be comparatively harder to implement and would require more time and effort to design based on the technical level of the person implementing query parser.	This method would be more capable in handling multiple patterns.	In terms of efficiency, both methods are similar.
Delimiter splitting	This method would be easier to implement based on the technical level of the person implementing query parser.	This method would be harder to expand upon when more complicated requirements come in such as multiple patterns.	In terms of efficiency both methods are similar.

Ultimately, the trade off between the 2 methods will be the time and effort spent for the one implementing query parser vs the capability of the query parser to handle complicated clause types in the future. The decision to choose 'Delimiter Splitting' over 'Split by regular expressions' in iteration 1 is being the requirements in iteration 1 is not that complicated. In the event that the basic/advanced requirements are complicated, there could be an opportunity to use method 2 'Split by regular expressions' or a mix of both methods, i.e. to use Delimiter splitting as default and 'Split by regular expressions' for multiple pattern clauses. For iteration 1, 'Delimiter Splitting' method would be used as it's easier to start with in iteration 1.

3.5 Query Evaluator

3.5.1 Data Representation of a Query

A query is represented as an object with 4 attributes:

- Declarations,
- Select synonym,
- Such-that clause(s),
- Pattern clause(s).

This fully covers a query's requirements under Iteration 1 (up to one such-that clause, up to one pattern clause). The Evaluator retrieves each of these attributes via its corresponding getter after the Query Parser constructs the object.

3.5.2 Method of Evaluation

The Evaluator has 5 important parts:

- The interface (i.e. the public method),
- The generic clause evaluator,
- The specific clause evaluators (one for each of Uses, Modifies, Pattern, etc.),
- The enumerators[1] (for statement reference, variable reference and factor),
- The synonym map (a synonym to an instance of it, e.g. "s" to "1").

The first step of evaluation is through the interface. The Evaluator gets all the attributes in the query. Then it interacts with the PKB to iterate through all possible instances of the Select synonym (e.g. all statement numbers in the case of "stmt s; Select s"), mapping the synonym to an instance and then going to the next step, generic evaluation.

Mapping a synonym to the synonym map is done every time we iterate through instances of a synonym. This will become relevant in Merging Evaluation of Multiple Clauses.

In generic clause evaluation, we get the next clause and proceed to call the appropriate specific clause evaluator for the clause.

In specific clause evaluation, we get the information of the clause passed by the generic evaluator. This information could be a statement reference or a variable reference. For ease of evaluation, these references are substituted by a collection of all possible instances of that reference (e.g. "v" can be substituted by ["x", "y", "z"]). This is done using the enumerators, which we will explain later.

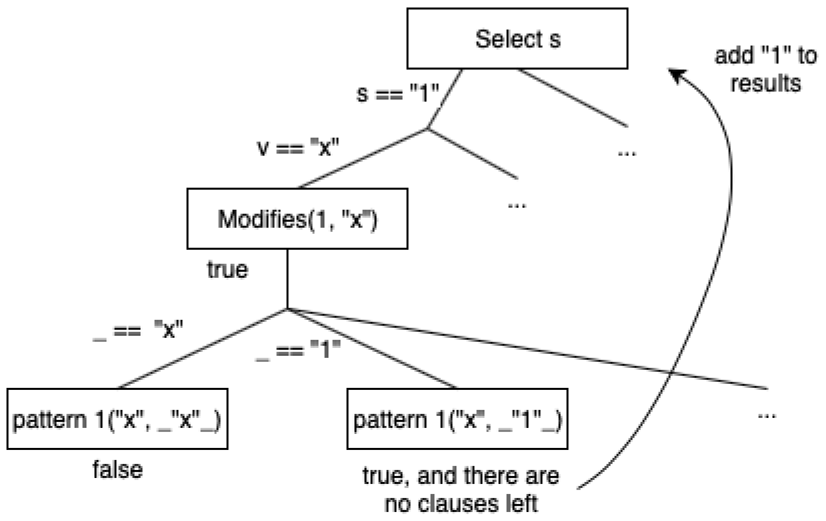
[1] Not to be confused with C++ enums.

For specific clause evaluation, we retrieve the instances of each enumerated collections, and call the appropriate PKB API predicate to get the truth value of the clause given said instances (e.g. in the case of “Uses(s, v)”, call isUsesStmtVar(1, “x”) for s == 1 and v == “x”). If the predicate returns true, we proceed to generic evaluation once again to evaluate the next clause. If the predicate returns false, we try the next set of instances with the PKB call. If no more set of instances can be used, we say “this clause is false given the synonym map”.

Then we trace back to the previous specific evaluation and try another set of instances for the previous clause, or if there are no clauses left, we go to the Select clause and say “this instance of the Select synonym **will not** be included in the results”. On the other hand, if in a generic clause evaluation we can get no more clauses, we say “this instance of the Select synonym **will** be included in the results”.

The enumerators, as mentioned, enumerate a statement/variable/factor reference. One important thing to note is that it first **checks the synonym map** if the reference is a synonym (e.g. “v” will only take on the name “x” if “v” is mapped to “x”).

In summary, the evaluation is done using a depth-first search approach, an example of which is shown in the below diagram.



Each node but the root node is associated with a boolean value. A node represents “true” if at least one of its children is “true” OR if it has no children AND the node represents a true clause.

The root node (the Select node) represents a collection of results, each of which is an instance of the Select synonym for which the corresponding child node is true.

Three important things to note:

- **Evaluation of individual clauses** is done with the specific clause evaluator, which uses the PKB API predicate to get the truth value (e.g. isUsesStmtVar() to test Uses relationship).

- **Merging evaluation of multiple clauses** is achieved because evaluation of a later clause "remembers" the mappings of synonyms used in earlier clauses. That is, when we say "this clause is true" we mean "this clause is true while using a synonym mapping under which the earlier clauses are true". This means once we reach the last clause and it says "true", then all the clauses are "true", and we add the instance of the Select synonym to the list of results.
- **Results of evaluation** is obtained by iterating through possible instances of the Select synonym, and taking only instances for which all clauses are true, under the mapping of the synonym to the instance.

3.5.3 Evaluator Example

We will be using the diagram in 3.5.2 to illustrate a step-by-step example of evaluation.

Say we have the following SIMPLE program:

```
procedure main {
1.  x = 1;
2.  y = 2;
3.  z = 3;
}
```

And the following query:

```
stmt s; variable v; Select s such that Modifies(s, v) pattern a(v, _)
```

We get a query with attributes:

Declarations:	(s:stmt, v:variable)
Select synonym:	s
Such-that clauses:	((Modifies, s, v))
Pattern clauses:	((a, v, _))

Generic evaluation walkthrough

1. Select synonym is s, We call PKB::getAllStmt() to get all statements (since s maps to stmt) and we get ("1", "2", "3")

2. We iterate through all of ("1", "2", "3"). We first try s == "1". We update the synonym map to (s:"1").

Handling the Such That clause

3. We get the clause (Modifies, s, v), a such-that clause, and call the specific clause evaluator for Modifies.

4. We enumerate s to (“1”) and v to (“x”, “y”, “z”). Note that s is enumerated as such because the synonym map has s mapped to “1”.

5. We try s == “1” and v == “x”. We update the synonym map to (s:”1”, v:”x”).

6. We call PKB::isModifiesStmtVar(1, “x”). We get true. We proceed to generic evaluation of the next clause.

Handling the Pattern clause

7. We get the clause (a, v, _), a pattern clause. We call the specific clause evaluator for Pattern.

8. We enumerate a to (“1”, “2”, “3”), v to (“x”) (again, synonym map), and _ to (“x”, “1”, “y”, “2”, “z”, “3”).

9. We try a == “1”, v == “x”, _ == “x”. We call PKB::isModifiesStmtVar(1, “x”) and we get true. But PKB::isConstUsedInAssign(1, “x”) **OR** PKB::isVarUsedInAssign(1, “x”) gives us false (all these PKB calls basically mean “PKB, I want to know if pattern a(v, _) holds under these mappings.”). We try the next triad.

10. a == “1”, v == “x”, _ == “1”. We call PKB::isModifiesStmtVar(1, “x”) and we get true. We try PKB::isConstUsedInAssign(1, “1”) and we also get true. We now proceed to generic evaluation of the next clause.

11. For this generic evaluation, there are no more clauses left. So we say ‘all the clauses hold under s == “1”, so we add “1” to the results.’

12. Do the same for s == “2” and s == “3” and we get the results (“1”, “2”, “3”). And we are done.

3.5.4 Design Rationale of Evaluator

An alternative to this approach of evaluation is to evaluate for one clause, then if there is a second clause, evaluate it and merge at the end. Roughly speaking, comparing the chosen approach vs. the alternative is like comparing depth-first search vs. breadth-first search on a Boolean satisfiability problem[2]. One thing to note is that the decision point was near the end of Iteration 1, and thus we include overhead of rewriting the Evaluator to use the alternative approach as an important consideration. Time & space complexity is put to be least important since Iteration 1 doesn't put too much pressure on time constraints.

	Criteria 1: Overhead of rewriting (Most important)	**Criteria 2: Flexibility for future iterations**	**Criteria 3: Time & space complexity (Least important)**
Alternative approach	A lot of overhead, rewriting pretty much from scratch (or paid up front, if we decide to use the alternative anyway)	May or may not be flexible, difficult to tell without actually implementing (though Prof. Zhao Jin claims it's flexible, so we'll say slightly easier, probably.)	Time complexity is roughly the same. Space complexity is high on storing a big table of merged results.
Chosen approach	No overhead (or carried over to Iteration 2 if we decide to use the alternative anyway)	Allows room for flexibility (though might be tricky since it uses some temporary hacks)	Time complexity is roughly the same. Space complexity is high on replicated collections and function calls.

Overall, we decided to use the current approach as the time and effort is better used somewhere else, and maybe reconsider using the alternative in Iteration 2.

[2] https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

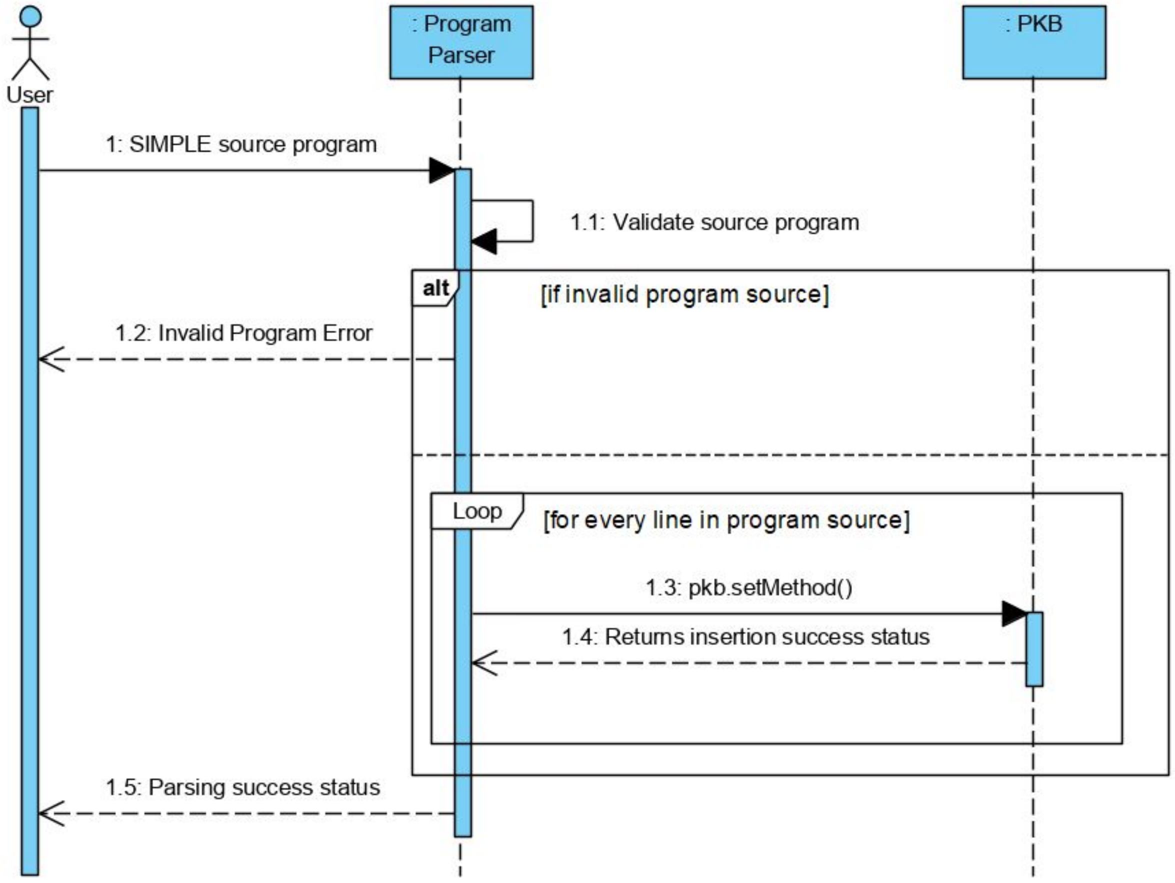
4. SPA Component interactions

Within the SPA, the main component interactions are between (i) the Program Parser and PKB, and (ii) between the PKB and the Query Evaluator.

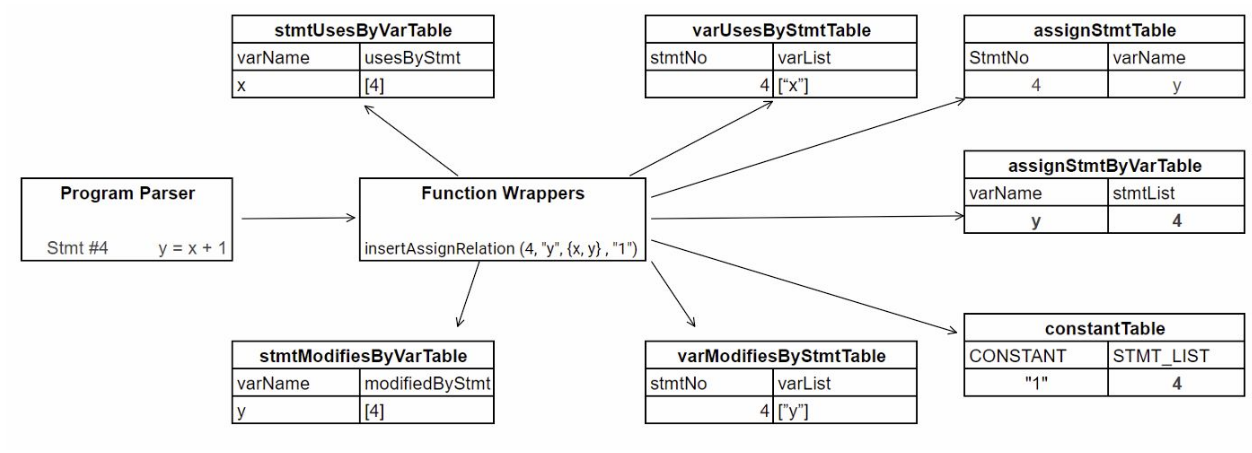
4.1 Source Code Parser and PKB

The source code parser and PKB have a good number of interactions where the Parser would call the API of PKB, specifically the setters APIs in order to insert the parsed information into the PKB. The parser makes these API calls as it parses the source code.

<u>High Level Sequence Diagram Between Source Code Parser and PKB</u>


User
: Program Parser
: PKB
1: SIMPLE source program
1.1: Validate source program
alt
[if invalid program source]
1.2: Invalid Program Error
Loop
[for every line in program source]
1.3: pkb.setMethod()
1.4: Returns insertion success status
1.5: Parsing success status

When parsing the source program the program parser populates the PKB by calling the **setter methods** of the PKB API for each line that is read. This process is repeated until no more lines are available. The figure above shows an example of the program parser calling the insertAssignRelations method when it parses a sample stmt #4: y=x+1.



stmtUsesByVarTable	
varName	usesByStmt
x	[4]

varUsesByStmtTable	
stmtNo	varList
4	[“x”]

assignStmtTable	
StmtNo	varName
4	y

assignStmtByVarTable	
varName	stmtList
y	**4**

stmtModifiesByVarTable	
varName	modifiedByStmt
y	[4]

varModifiesByStmtTable	
stmtNo	varList
4	["y”]

constantTable	
CONSTANT	STMT_LIST
"1"	**4**

4.2 Query Evaluator and PKB

The PKB is also used when the query evaluator needs to retrieve information to evaluate a query. The query evaluator uses mainly **getter methods** of the PKB API.
The Evaluator uses the following calls for evaluation logic:

getAlls: Mostly used when enumerating Select synonyms.

- getAllStmt()
- getAllReadStmt()
- getAllPrintStmt()
- getAllWhileStmt()
- getAllIfStmt()
- getAllAssignStmt()
- getAllVar()
- getAllConstant()
- getAllProc()

Relationship predicates: Used in the last steps of clause evaluation

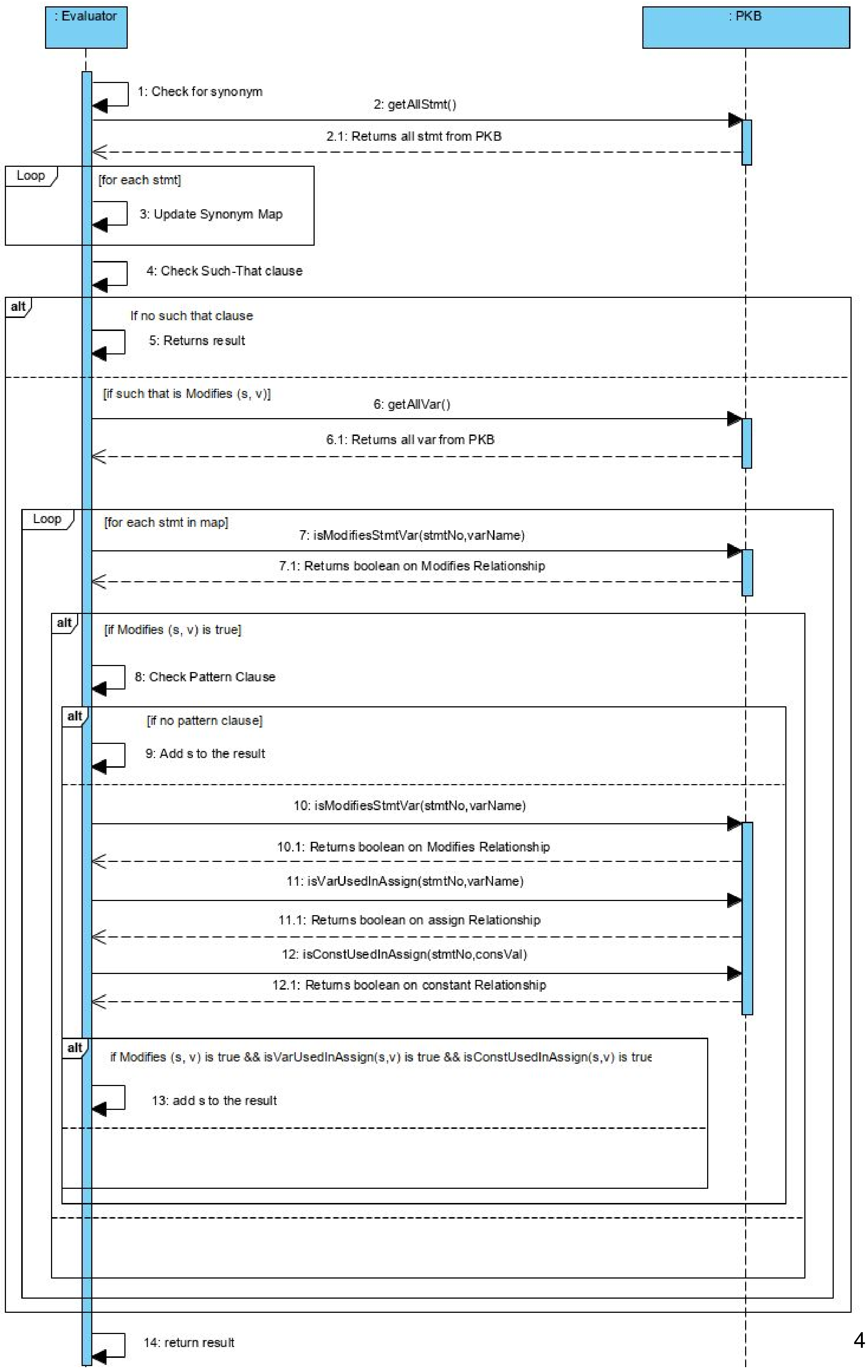
- isUsesStmtVar()
- isModifiesStmtVar()
- isFollowRelationship()
- isFollowStarRelationship()
- isParentRelationship()
- isParentStarRelationship()

Pattern handlers: Used to check if a factor is in the RHS of an assign statement

- isConstUsedInAssign()
- isVarUsedInAssign()

Sequence Diagram Between Query Evaluator and PKB

The sequence diagram below shows a general flow for evaluating a query, which does the above APIs calls to the PKB to retrieve data for evaluation. Shown below is just an example of how a query is being evaluated for that specific case, as there are too many varieties to show all possibilities.



: Evaluator
: PKB
1: Check for synonym
2: getAllStmt()
2.1: Returns all stmt from PKB
Loop
[for each stmt]
3: Update Synonym Map
4: Check Such-That clause
alt
If no such that clause
5: Returns result
[if such that is Modifies (s, v)]
6: getAllVar()
6.1: Returns all var from PKB
Loop
[for each stmt in map]
7: isModifiesStmtVar(stmtNo,varName)
7.1: Returns boolean on Modifies Relationship
alt
[if Modifies (s, v) is true]
8: Check Pattern Clause
alt
[if no pattern clause]
9: Add s to the result
10: isModifiesStmtVar(stmtNo,varName)
10.1: Returns boolean on Modifies Relationship
11: isVarUsedInAssign(stmtNo,varName)
11.1: Returns boolean on assign Relationship
12: isConstUsedInAssign(stmtNo,consVal)
12.1: Returns boolean on constant Relationship
alt
if Modifies (s, v) is true && isVarUsedInAssign(s,v) is true && isConstUsedInAssign(s,v) is true
13: add s to the result
14: return result

5. Testing

In order to ensure that our program could handle the large number of possible source codes and queries, testing needed to be done. Testing of our program during this iteration was done with 3 types of testing: Unit testing, Integration testing and System testing.

5.1 Test plan

Our test plan is an extension of our development plan for each mini iteration. For every mini iteration, we have a development period after we have decided on the deliverables for the mini iteration. During the development period of the mini iteration, one member in our team would be dedicated to create test sources and queries for system testing at the end of the mini iteration, while the rest of the team members in charge of the component development will also be in charge of their own unit testing.

The diagram below illustrates our testing plan:

Role	Design & First 2 days of Development	Second Last day of Development	Last day of Development	Review
Component Developer	Development of Component feature(s)	Unit Testing of individual component feature(s)	Integration of components and integration testing	Respond to bug reports accordingly.
System Test Developer	Creation of System test source and queries	Check that system tests coverage is sufficient.		Run system tests & regression tests, report bugs.

The goal of the test plan for each mini iteration is to have a 100% pass rate of the system test cases by the end of the mini iteration in order to ensure that most if not all requirements are met.

At the end of each mini iteration, the respective mini iteration's system test cases are kept and used to conduct regression tests for subsequent mini iterations.

System test cases are designed based on the features that are set to be worked on for that mini iteration. For the features set, basic cases that the system should be able to solve for said feature are identified and test cases are created for them. After which, attempt to identify as many possible complex cases and create test cases for them. Test case sources and queries are then written in the autotester format and manually ran by the autotester for system testing.

Failed test cases are examined to check if there are any possible query error such as unintentionally missing a declaration or the expected answer has been misevaluated. If there are such errors, changes are made and the test cases are run by the autotester again. Else, if there are no such errors, the failed test cases are communicated to the team through the established communications channel and possible bugs and related components are identified. Team members will be assigned to fix bugs related to the components they are working on. Once all the bugs are verified to be fixed and all test cases have passed, the test cases are then kept to be used as regression tests for the next iteration.

Test Strategy

0 clause - As this is the first set of tests and syntax checking was not planned for the first mini iteration, only functional tests are created for this set. Firstly, possible select results are listed out alongside the inputs corresponding to them. Secondly, a simple test source is then written with at least one of each select results. Lastly, test queries are then written for each select result.

1 clause - For such that and pattern clause, functional, algorithmic and boundary tests are written. First, all such that clauses are listed out (Follows, Follows*, Parent, etc.) and pattern clause. Next, possible valid inputs are identified for the functional tests. After which, possible relations of each clause is identified for algorithmic tests (E.g. A Follows with 2 distant statement numbers that is true due to a if/while). The boundary inputs such as the first, last and non-existant statement numbers are then identified alongside the complex boundary cases (stmt that does not follow any statement due to being the last statement in a if/while container). Test sources are then written for each clause. The test queries for each clause, which consists of functional, algorithmic and boundary test, are then written.

2 clause - For 2 clauses, functional, algorithmic and boundary tests are written. All possible combinations of clauses are listed out. For each combination, valid inputs are identified for the functional tests. Next, possible inputs are identified for each combination and listed out for algorithmic tests as well as complex cases (E.g. Result of such that clause returns a list, result of pattern returns a list but the final merge result consists of only a few that show up in both lists).. The boundary inputs such as the first, last and non-existant statement numbers are then identified. A test source is then written. Test queries for each combination are then written.

5.2 Examples of test cases of different categories

5.2.1 Source Parser Unit Test

The Source parser unit tests consist of multiple tests on the many sub components of the Parser. Two examples have been selected to show the level of complexity of the inputs to tokenize that is being tested in these test cases:

1. **Condition statement test**

   Test Purpose: This was to test a sub component under the Source Parser that was designed to handle tokenization the variables used in the condition statement in an if/while condition. Multiple tests were conducted with different permutations.

   Required Input: A sample input is shown below, the input would encompass the possible permutations of the entire line that is read for an “if” statement. E.g. “if (a < b) then {“.

   Expected test results: An assert is used to ensure the pre-defined expected vector and the actual results are equal.

   An example is shown below in the picture.

```
TEST_METHOD(IfCondStmtTest2)
{
	Parser parser = Parser();
	string input = "if ((x == 1) || (a != b) && (c <= d) || (e >= f) || (g < h) && (i > j)) then {";
	vector<string> expected{ "x", "1", "a", "b", "c", "d", "e", "f", "g", "h", "i", "j" };
	vector<string> actual = parser.parseCondStmt(input);
	Assert::AreEqual(expected == actual, true);
}
```

2. **<u>AssignRHSTest</u>**

Test Purpose: This was to test a sub component that was used to parse the used variables/constants in the RHS of an assign statement. The use of this sub component was then expanded to tokenize the LHS and RHS of a rel_expr as the grammar rules were similar. Multiple tests were created with different permutations.

Required Input: A sample input is shown below, the input would encompass the possible permutations of the RHS of an assign statement. E.g. “(a + b) - (c / (d * (3 % 4)))“.

Expected test results: An assert is used to ensure the pre-defined expected vector and the actual results are equal.

An example is shown below in the picture.

```
TEST_METHOD(AssignRHSTest3)
{
        Parser parser = Parser();
        string input = "(a+(b-((c*d)/(e%2))))";
        vector<string> expected{ "a", "b", "c", "d", "e", "2" };
        vector<string> actual = parser.parseAssignRHS(input);
        Assert::AreEqual(expected == actual, true);
}
```

5.2.2 Query Parser Unit Testing

The query parser unit tests consists of tests for the functions implemented in query parser. Two examples have been selected to show the level of complexity of the inputs to tokenize.

1. **splitVariablesInDeclarations Test**

   Test Purpose: This is to test the function that is used to map variables to its respective types by splitting through whitespace. The output of this function will then be parsed into one of the parameters of the query object.

   Required Input: A sample input is shown below, the input would encompass the various declarations statement with select variable and such that/pattern clauses.

   Expected test results: An assert is used to ensure the pre-defined expected map and the actual results are equal.

   An example is shown below in the picture.

```
TEST_METHOD(splitVariablesInDeclerations)
{
	vector<string> input = { "variable v1, v2", "while w1 ,w2", "Select v such that Follows(v,w)" };
	unordered_map<string, string> actual = QueryParser::splitVariablesInDeclerations(input);
	unordered_map<string, string> expected{ {"v1", "variable"}, {"v2", "variable"}, {"w1", "while"}, {"w2","while"} };
	Assert::AreEqual(actual == expected, true);
}
```

2. declarationsValiation Test

Test Purpose: This is to test the function declarationsValidation that is used to validate the variable naming and validity of declared types in declarations. The output of this result will determine whether the query will be rejected.

Required Input: A sample input is shown below, the input would encompass the possible invalid map of declarations.

Expected test results: An assert is used for each input to ensure the pre-defined expected string and the actual results are equal.

An example is shown below in the picture.

```
TEST_METHOD(declarationsValidation)
{
	unordered_map<string, string> input{ {"1a2", "assign"},{"v", "variable"} };
	string actual = QueryParser::declarationsValidation(input);
	string expected = "Invalid";
	Assert::AreEqual(actual == expected, true);

	unordered_map<string, string> input2{ {"a1_", "assign"},{"v", "variable"} };
	string actual2 = QueryParser::declarationsValidation(input2);
	string expected2 = "Invalid";
	Assert::AreEqual(actual2 == expected2, true);

	unordered_map<string, string> input3{ {"a", "assign"},{"char", "c"} };
	string actual3 = QueryParser::declarationsValidation(input3);
	string expected3 = "Invalid";
	Assert::AreEqual(actual3 == expected3, true);
}
```

5.2.3 System Testing

The following is the test source of the system test cases with statement numbers for reference.

```
s       procedure TestCase3 {
1         x = 1;
2         y = 3;
3         while (y >= x) {
4           read z;
5           if (z < 10) then {
6             z = z + x - y;
7             read i;
8             print i; }
/           else {
9             read k;
10            print k; } }
11        read max;
12        mid = max / 2;
13        read start;
14        while (start < max) {
15          start = start + 1;
16          while (start <= mid) {
17            start = start + 4; }
18          print start; }
19        read end;
e       }
```

1. **System Testing Test Case 1: Follows* and pattern assign - merge true**

   Test Purpose: Test system's ability to select assign statement/s with a specific variable on the LHS that is also followed by another stmt (Not the last stmt in a if/while container or procedure).

   Required Test Input : This tests the whole system. Input required is the test source program as well as query written in the specified format for the auto tester.

   Expected Test Result : The statement number/s of assign statements that have the specified variable on the LHS that are also followed by another stmt.

   An example is shown below in the picture in autotester format.

```
24 - Follows* and pattern assign - merge true
assign a; stmt s;
Select a such that Follows*(a, s) pattern a("start", _)
15
5000
```

2. **System Testing Test Case 2: Follows* and pattern assign - merge true**

Test Purpose: Test system's ability to select all variables that are used in a print statement and also appear in the LHS of an assign statement.

Required Test Input : This tests the whole system. Input required is the test source program as well as query written in the specified format for the auto tester.

Expected Test Result : The name/s of variables that are used in a print statement and also appear in the LHS of an assign statement.

An example is shown below in the picture in autotester format.

```
30 - Uses and pattern assign - merge true
assign a; variable v; print pn;
Select v such that Uses(pn, v) pattern a(v, _)
start
5000
```

6. Documentation and Coding Standards

6.1 Abstract APIs Naming Convention

There are 3 main types of abstract APIs being made with the starting naming convention of "get","set" and "is". Each have a different return type with a clear state of what it does through its API name. We want to keep it simple yet intuitive of what the API does simply through its name.

An API that starts with "get" will retrieve and return data from the PKB, in an abstract type of STMT_NO, LIST, VAR_NAME, these are the "getters" APIs that will be called by the Evaluator and the Design Extractor. The "get" APIs have middle name convention of the data they are returning. For example, getAllVar means that it will return all variable from varTable, and getChildrenStmtList will return a list of children statement numbers from the parentTable.

An API that starts with "set" will set and insert data into the PKB and returns a Boolean, such as true for a successful insertion and false if the insertion fails. These "setters" APIs will be called by the Program Parser and the Design Extractor. For example, setVar means that it will set a variable into the PKB, varTable, while setParent will set a parent relationship into the PKB.

An API that starts with "is" will check if a relationship is true or false and returns the Boolean. These "is" APIs will be called by Evaluator to check for Follows, Follows*, Parent, Parent*, Uses, Modifies. For example, isParentRelationship, will check if the relationship exist in the PKB, parentTable, if it found the relationship does exist, it will return true to that relationship, otherwise it returns false.

	Starting Name	**APIs Names Examples**	**Description**	**Return Type**	**For Which Component**
1	get	getAllVar, getAllStmt, getAllWhileStmt, getChildrenStmtList	Gets data from the PKB	STMT_NO, LIST, VAR_NAME	Evaluator, Design Extractor
2	set	setVar, setStmt, setWhile, setIfs, setConstant, setProc, setParent	Set data into the PKB	Boolean	Program Parser, Design Extractor
3	is	isParentRelationship, isParentStarRelationship, isUsesRelationship, isModifiesRelationship,	Check if a relationship is true from the PKB	Boolean	Evaluator

6.2 Enhancing correspondence between abstract APIs and the concrete API counterparts

Abstract data types are used in the abstract APIs and its concrete APIs counterpart. The C programming language provides a keyword called typedef, which we can use to create an abstract data type, giving string a new name such as STMT_NO, VAR_NAME, PROC_NAME. And even an unordered_set <string> can be defined as STMT_LIST as an abstract type. Therefore, by using the abstract type listed in the abstract APIs in our concrete API counterparts, it ensures that the APIs are called correctly and returns correctly with its abstract data type.

6.3 Coding standards and style adopted

For the team coding standards, we tried adopting some C++ coding standards from Sutter and Alexandrescu's "C++ Coding Standards: 101 Rules, Guidelines, and Best Practices".

For our source file we are using .cpp extension and for our header file we are using .h extension. The source file name and header file name are the same corresponding to each other. For example, there will be a Evaluator.cpp and it will include Evaluator.h. Even for our unit testing source file, we correspond the name such as TestEvaluator.cpp.

We are using namespace std for our source file. This helped us to reduce the need to type std:: over and over in our source file. For example in our Evaluator.cpp we have:

```
#include “Evaluator.h”
using namespace std;
```

However, according to Sutter and Aleaxndrescus’s C++ coding standards, using namespace should not be applied to the header files under section 59 of the book. It mentioned that “Don’t write namespace usings in a header file or before an #include”. It stated that “Namespace usings are for your convenience, not for you to inflict on others: Never write a using declaration or a using directive before an #include directive.” With the reasoning that a header file is a guest in one or more source files, a header file that includes using directives and declarations can unexpectedly change the meaning of code in any other files that include that header. Therefore, for our header file, we are not using any namespace.

Additionally, all of our methods and variables are camel case such as getAllVar(), with the exception of abstract data types, which we use with alls caps and underscores such as STMT_NO and STMT_LIST. All public methods of a class must have comments with what it does and its input and expected output.

7. Abstract API Documentation

7.1 PKB Statement and Variable Table Operations

7.1.1 varTable

varTable *Overview:* varTable stores all variables in the program from the Parser.	
	API
	BOOLEAN setVar(**VAR_NAME** varName) ***Description***: Adds a variable to the varTable.
	VAR_LIST getAllVar() ***Description***: Returns a list of all variables stored in the varTable.

7.1.2 stmtTable

stmtTable Overview: stmtTable stores all statements and their stmt type in the program from the Parser.	
	API
	BOOLEAN setStmt(**STMT_NO** stmtNo, **STMT_TYPE** type) ***Description***: Adds a statement into the stmtTable.
	STMT_LIST getAllStmt () ***Description***: Returns a list of all the statements in the stmtTable.
	STMT_LIST getAllAssignStmt () ***Description***: Returns a list of all assign statements in the stmtTable.

7.1.3 whileTable

whileTable Overview: whileTable stores a list of all while statements in the program from the Parser.	
	API
	BOOLEAN setWhileStmt(**STMT_NO** stmtNo) ***Description***: Adds a while statement stmtNo into the whileTable
	STMT_LIST getAllWhileStmt() ***Description***: Returns a list of all while statements in the whileTable

7.1.4 ifTable

ifTable Overview: ifTable stores a list of all if statements in the program from the Parser.	
	API
	BOOLEAN setIfStmt(**STMT_NO** stmtNo) ***Description***: Adds a if statement stmtNo into the ifTable
	STMT_LIST getAllIfStmt() ***Description***: Returns a list of all if statements in the ifTable

7.1.5 printTable

printTable Overview: printTable stores a list of all print statements in the program from the Parser.	
	API
	BOOLEAN setPrintStmt(**STMT_NO** stmtNo, **VAR_NAME** varName) ***Description***: If varName is not in the printTable, adds a variable varName with its stmtNo to the printTable. Else only add stmtNo to the list of statements at printTable[“varName”].
	STMT_LIST getPrintStmtByVar(**VAR_NAME** varName) ***Description***: Returns a list of statements that prints variable varName. If ‘varName’ is not found in the printTable, returns an empty list.
	STMT_LIST getAllPrintStmt() ***Description***: Returns a list of all print statements in the printTable.

7.1.6 readTable

readTable Overview: readTable stores a list of all print statements in the program from the Parser.	
	API
	BOOLEAN setReadStmt(**STMT_NO** stmtNo, **VAR_NAME** varName) ***Description***: If varName is not in the readTable, adds a variable varName with its stmtNo to the readTable. Else only add stmtNo to the list of statements at readTable[“varName”].
	STMT_LIST getReadStmtByVar(**VAR_NAME** varName) ***Description***: Returns a list of statements that reads variable varName. If ‘varName’ is not found in the readTable, returns an empty list.
	STMT_LIST getAllReadStmt() ***Description***: Returns a list of all read statements in the readTable.

7.1.7 constantTable

constantTable Overview: constantTable stores all constants in the program from the Parser.	
	API
	BOOLEAN setConstant (**CONS_VAL** constantValue, **STMT_NO** stmtNo) ***Description***: If constantName is not in the constantTable, adds a constant value with its stmtNo to the constantTable. Else only add stmtNo to the list of statements at constantTable[“constantValue”].
	CONS_LIST getAllConstant() ***Description***: Returns a list of all the constants stored in the constantTable.
	CONS_LIST getStmtByConst(**CONS_VAL** constantValue) ***Description***: Returns a list of all the statements at constantTable[constantValue]. If ‘constantValue’ is not found in the constantTable, returns empty list.
	BOOLEAN isConstantExist() **Description:** Returns true if a constant exists in the constantTable, else return false.

7.1.8 ProcTable

procedureTable Overview: procedureTable stores a list of all procedures in the program from the Parser.	
	API
	BOOLEAN setProc(**PROC_NAME** procName) ***Description***: Adds a procedure procName into the procedureTable
	PROC_LIST getAllProc() ***Description***: Returns a list of all procedures in the procedureTable

7.2 PKB Relationship Table Operations

7.2.1 stmtModifiesByVarTable

stmtModifiesByVarTable Overview: stmtModifiesByVarTable stores a list of variables and the stmt lines in which they are modified in the program from the Parser.	
	API
	BOOLEAN setModifiesStmtByVar(**STMT_NO** stmtNo, **VAR_NAME** varName) ***Description***: If varName is not in the stmtModifiesByVarTable, adds a variable varName to the table and stmtNo to the list at stmtModifiesByVarTable['varName']. Else only add stmtNo to the list of statements at stmtModifiesByVarTable['varName'].
	STMT_LIST getModfiesStmtByVar(**VAR_NAME** varName) ***Description***: Returns a list of all statements that has been modified at stmtModifiesByVarTable['varName']. If varName is not found in the stmtModifiesByVarTable, returns an empty list.
	STMT_LIST getAllModifiesVar() ***Description***: Returns a list of all variables that are modified.

7.2.2 varModifiesByStmtTable

varModifiesByStmtTable Overview: varModifiesByStmtTable stores a list of statements and the variables that are modified in these statements in the program from the Parser.	
	API
	BOOLEAN setVarModifiesByStmt(**STMT_NO** stmtNo, **VAR_NAME** varName) ***Description***: If stmtNo is not in the varModifiesByStmtTable, adds a variable varName to the table and varName to the list at varModifiesByStmtTable[stmtNo]. Else only add varName to the list of variables at varModifiesByStmtTable[stmtNo].
	STMT_LIST getModifiesVarByStmt(**STMT_NO** stmtNo) ***Description***: Returns a list of all variables that has been modified at varModifiesByStmtTable[stmtNo]. If varName is not found in the varModifiesByStmtTable, returns an empty list.
	STMT_LIST getAllModifiesStmt() ***Description***: Returns a list of all statement number that has been modified
	BOOLEAN isModifiesRelationship(**STMT_NO** stmtNo, **VAR_NAME** varName) ***Description***: Returns true if there is a modifies relationship between statement stmtNo, variable varName, modifies(stmtNo, variable).

7.2.3 stmtUsesByVarTable

stmtUsesByVarTable Overview: stmtUsesByVarTable stores a list of variables and the stmt lines in which they are used in the program from the Parser.	
	API
	BOOLEAN setUsesStmtByVar(**STMT_NO** stmtNo, **VAR_NAME** varName) ***Description***: If varName is not in the stmtUsesByVarTable , adds a variable varName to the table and stmtNo to the list at stmtUsesByVarTable [‘varName’]. Else only add stmtNo to the list of statements at stmtUsesByVarTable [‘varName’].
	STMT_LIST getUsesStmtByVar(**VAR_NAME** varName) ***Description***: Returns a list of all statements that has been uses at stmtUsesByVarTable[‘varName’]. If varName is not found in the stmtUsesByVarTable , returns an empty list.
	STMT_LIST getAllUsesVar() ***Description***: Returns a list of all variables that are used

7.2.4 varUsesByStmtTable

varUsesByStmtTable Overview: varUsesByStmtTable stores a list of statements and the variables that are used in these statements in the program from the Parser.	
	API
	BOOLEAN setUsesVarByStmt(**STMT_NO** stmtNo, **VAR_NAME** varName) ***Description***: If stmtNo is not in the varUsesByStmtTable, adds a variable varName to the table and varName to the list at varUsesByStmtTable[stmtNo]. Else only add varName to the list of variables at varUsesByStmtTable[stmtNo].
	STMT_LIST getUsesVarByStmt(**STMT_NO** stmtNo) ***Description***: Returns a list of all variables that has been modified at varUsesByStmtTable[stmtNo]. If varName is not found in the varUsesByStmtTable, returns an empty list.
	STMT_LIST getAllUsesStmt() ***Description***: Returns a list of statements that uses a variable
	BOOLEAN isUsesRelationship(**STMT_NO** stmtNo, **VAR_NAME** varName) ***Description***: Returns true if there is a uses relationship between statement stmtNo, variable varName, uses(stmtNo, variable).

7.2.5 assignStmtTable

assignStmtTable Overview: assignStmtTable stores the assign relationship in the program from the Parser.	
	API
	BOOLEAN insertAssignRelation (**STMT_NO** stmtNo, **VAR_NAME** varModified, **VAR_LIST** varsUsed, **CONS_LIST** constantsUsed) ***Description***: Adds an assignment statement stmtNo**,** the modified variable varName on the LHS of the "=" operator, a list of zero or more variables used in the RHS of the "=" operator varsUsed and a list of zero or more constants used constantsUsed in the RHS of the "=" operator into the PKB. Returns true if insertion is successful, and false if otherwise. Inserts the relevant entries into **assignStmtTable** respectively.
	BOOLEAN isConstUsedInAssign(**STMT_NO** stmtNo, **CONS_VAL** const) Description: Returns true if constant **const** is used in RHS of assign stmt otherwise false.
	BOOLEAN isVarUsedInAssign(**STMT_NO** stmtNo, **VAR_NAME** var) Description: Returns true if variable **var** is used in RHS of assign stmt otherwise false.

7.3 Design Extractor Table Operations

7.3.1 FollowsTable

followsTable Overview: followsTable stores follow relationship in the program from the Parser.	
	API
	BOOLEAN setFollow(**STMT_NO** s1, **STMT_NO** s2) ***Description***: Adds a followedBy statement number s1 to the followsTablewith s2 being added as follows.
	STMT_NO getFollowStmt(**STMT_NO** followedBy) ***Description***: Returns the follow statement number from followsTable given the followedBy statement number. If the followedBy statement number cannot be found in followsTable, then it returns -1.
	STMT_NO getFollowedByStmt(**STMT_NO** follows) ***Description***: Returns the followedBy statement number from followsTable given the follows statement number. If the follows statement number cannot be found in followsTable, then it returns -1.
	BOOLEAN isFollowRelationship(**STMT_NO** s1,**STMT_NO** s2) ***Description***: Returns true if there is a follows relationship between the two statement s1, s2, follows(s1,s2).

7.3.2 followsStar Table

followStarTable Overview: followsStarTable stores follow* relationship in the program from the Parser.	
	API
	BOOLEAN setFollowStar(**STMT_NO** s1, **STMT_NO** s2) ***Description***: If s1 is not in the followStarTable, adds a followedBy statement number s1 to the followsStarTable with s2 being added into the followStarList. Else only add s2 to the list of statements at followsStarTable [s1].
	STMT_NO getFollowedByStar(**STMT_NO** followedBy) ***Description***: Returns the statement number of the followedByStar from followStarTable given the followsStar statement number. If the followsStar statement number cannot be found in followsStarTable, then it returns -1.
	STMT_LIST getFollowsStarStmtList(**STMT_NO** followedBy) ***Description***: Returns a list of all the followsStar statements at followsStarTable [followedBy]. If parent is not found in the followsStarTable, returns an empty list.
	BOOLEAN isFollowStarRelationship(**STMT_NO** s1,**STMT_NO** s2) ***Description***: Returns true if there is a follows* relationship between the two statement s1, s2, follows*(s1,s2).

7.3.3 parentTable

parentTable Overview: parentTable stores parent relationship in the program from the Parser.	
	API
	BOOLEAN setParent(**STMT_NO** s1, **STMT_NO** s2) ***Description***: If s1 is not in the parentTable as a parent, adds a parent statement number s1 to the parentTable with s2 being added into the child list. Else only add s2 to the list of statements at parentTable[s1].
	STMT_NO getParentStmt(**STMT_NO** child) ***Description***: Returns the parent statement number from parentTable given the child statement number. If the child statement number cannot be found in parentTable, then it returns -1.
	STMT_LIST getChildrenStmtList(**STMT_NO** parent) ***Description***: Returns a list of all the child statements at parentTable[parent]. If parent is not found in the parentTable, returns empty list.
	BOOLEAN isParentRelationship(**STMT_NO** s1,**STMT_NO** s2) ***Description***: Returns true if there is a parent relationship between the two statement s1, s2, parent(s1,s2).

7.3.4 parentStarTable

parentStarTable Overview: parentStarTable stores parent* relationship in the program from the Parser.	
	API
	BOOLEAN setParentStar(**STMT_NO** s1, **STMT_NO** s2) ***Description***: If s1 is not in the parentStarTable, adds a parentStar statement number s1 to the table with s2 being added into the child list. Else only add s2 to the list of statements at parentStarTable[s1].
	STMT_NO getParentStarStmt(**STMT_NO** child) ***Description***: Returns the statement number of the parentStar from parentStarTable given the child statement number. If the child statement number cannot be found in parentStarTable, then it returns -1.
	STMT_LIST getChildrenStarStmtList(**STMT_NO** parent) ***Description***: Returns a list of all the child statements at parentStarTable[parent]. If parent is not found in the parentStarTable, returns an empty list.
	BOOLEAN isParentStarRelationship(**STMT_NO** s1,**STMT_NO** s2) ***Description***: Returns true if there is a parent relationship between the two statement s1, s2, parent*(s1,s2).

7.3.4 assignStmtTable

assignStmtTable Overview: assignStmtTable stores assign relationship in the program from the Parser.	
	API
	BOOLEAN setAssignRelation (**STMT_NO** stmtNo, **VAR_NAME** varModified, **LIST** varsUsed, **LIST** constantsUsed) ***Description***: Adds an assignment statement stmtNo**,** the modified variable varName on the LHS of the “=” operator, a list of zero or more variables used in the RHS of the “=” operator varsUsed and a list of zero or more constants used constantsUsed in the RHS of the “=” operator into the PKB. Returns true if insertion is successful, and false if otherwise.
	BOOLEAN isConstUsedInAssign(**STMT_NO** stmtNo, **CONS_VAL** const) Description: Returns true if constant const is used in RHS of assign stmt.
	BOOLEAN isVarUsedInAssign(**STMT_NO** stmtNo, **VAR_NAME** varName) Description: Returns true if variable varName is used in RHS of assign stmt.

7.4 Higher Level Methods / Wrappers

Higher Level Methods / Wrappers Overview: These are just wrapper APIs which uses the other basic abstract APIs. The insert APIs served as a convenience to do 1 API call to populate both the actual and inverse table for Uses and Modifies relationship.	
	API
	BOOLEAN insertModifiesRelation(**STMT_NO** stmtNo, **VAR_NAME** varName) ***Description***: Adds a statement stmtNo that modifies variable varName to the PKB. Inserts the relevant entries into stmtModifiesByVarTable and varModifiesByStmtTable respectively. Returns true if insertion is successful, and false if otherwise.
	BOOLEAN insertUsesRelation(**STMT_NO** stmtNo, **VAR_NAME** varName) ***Description***: Adds a statement stmtNo that modifies variable varName to the PKB. Inserts the relevant entries into stmtUsesByVarTable and varUsesByStmtTable respectively. Returns true if insertion is successful, and false if otherwise.
	STMT_LIST getAllStmtByType(**STMT_TYPE** stmtType) Description: Returns a list of all statements of type stmtType**.**

8. Discussion

8.1 Problems Faced

The first integration of everyone's component took us longer than expected. This actually causes us to change our mini iteration 1 plan by removing the Uses and Modifies portion for our first mini iteration and simply focus on 0 clause first. The problem arises when we took a huge step ahead, trying to solve single clause problem before solving 0 clause is being integrated. Additionally, more problems arise from the lack of extensive unit testing in some of our components at the start, which causes some of the components to fail when trying to do integration testing and system testing.

Ensuring an extensive unit testing on everyone’s component allows a smoother integration, as once the first integration has been completed, it is easier to spot mistakes and bugs from each team member which allow us to solve it quickly. Development has also increased on our code base after the first integration with regards to the confidence boost from the first successful integration.

8.2 Lessons Learnt

If we were to start the project again, we would integrate the 0 clause first, instead of trying to solve both 0 and single clause for the first integration.

Additionally, after the first integration, for our source code management we decided to split into 2 branches for each of our components, a dev branch and a stable branch. This ensures that the stable branch is working and has been unit tested, ready for merging. This also helps improve integration process, as team members are able to work on other features for the next mini iteration without affecting the current stable branch to be merged.