Team 08: SPA Documentation

Static Program Analysis (SPA) Tool
CS3201/2

Members of Team 08:

1.	Daren Tan	A0111855J	90103129	a0111855@u.nus.edu
2.	Cheong Yan Qin	A0101728M	97676626	cyanqin@u.nus.edu
3.	Duan Xuzhou	A0108453J	93848055	a0108453@u.nus.edu
4.	Su Gi Chandran	A0101642J	92240731	a0101642@u.nus.edu
5.	Tricia Tan	A0116733J	96499892	tricia.tzy@u.nus.edu
6.	Lim Wei Cheng	A0101871N	91151504	wclim91@u.nus.edu

Table of Contents

1. Development Plan

Iteration 1 (In order of sequence of events)	**Team Member**					
	Su Gi	Daren	Tricia	Yan Qin	Wei Cheng	Leon
Build SIMPLE Parser	*					
Build AST					*	
Build ProcTable and VarTable					*	
Build relationships and storage in PKB				*		
Write Query Test Cases			*			
Build Query Pre-processor		*				
Build Query Evaluator						*
Integration	*	*	*	*	*	*
Integrate SPA and AutoTester			*			
Write Unit Test Cases	*	*		*	*	*
Write System Acceptance Test Cases			*			
Documentation	*	*	*	*	*	*

2. Scope of prototype implementation

Parser

The Parser is able to tokenize a given SIMPLE program and parses it to check its syntax. When the Parser encounters a syntax error, it will print error messages and stop parsing; queries cannot be performed thereafter. Once the Parser verifies that the program is syntactically correct, it returns the tokenized program to the main controller, which uses the tokenized program to construct an AST.

PKB

The PKB provides API for building an AST, VarTable and ProcTable. In addition to calculating the relations, it provides API for querying the PKB about the relations Modifies, Uses, Calls, Parent, Parent*, and Follows.

Query

The Query is split into a preprocessor and evaluator. The preprocessor is able to validate the query string input and passes a vector of custom objects, QueryObject, to the evaluator for logic processing. The query evaluator will get information such as Modifies, Uses, Parent/*, Follows/* relationships from PKB. For error handling, the evaluator will return an empty list<string>.

Bonus Features

The Parser implements parsing of If statements.

Comments on design decisions

Currently, we have set various assumptions for our prototype.
Firstly, we take the inputs of the program to be case insensitive. We also assume that there is at least one space between each word e.g. ‘Selectasuchthat…’ will not work. Architecture design decisions is specified in 3.1.

3. SPA design

3.1 Overview

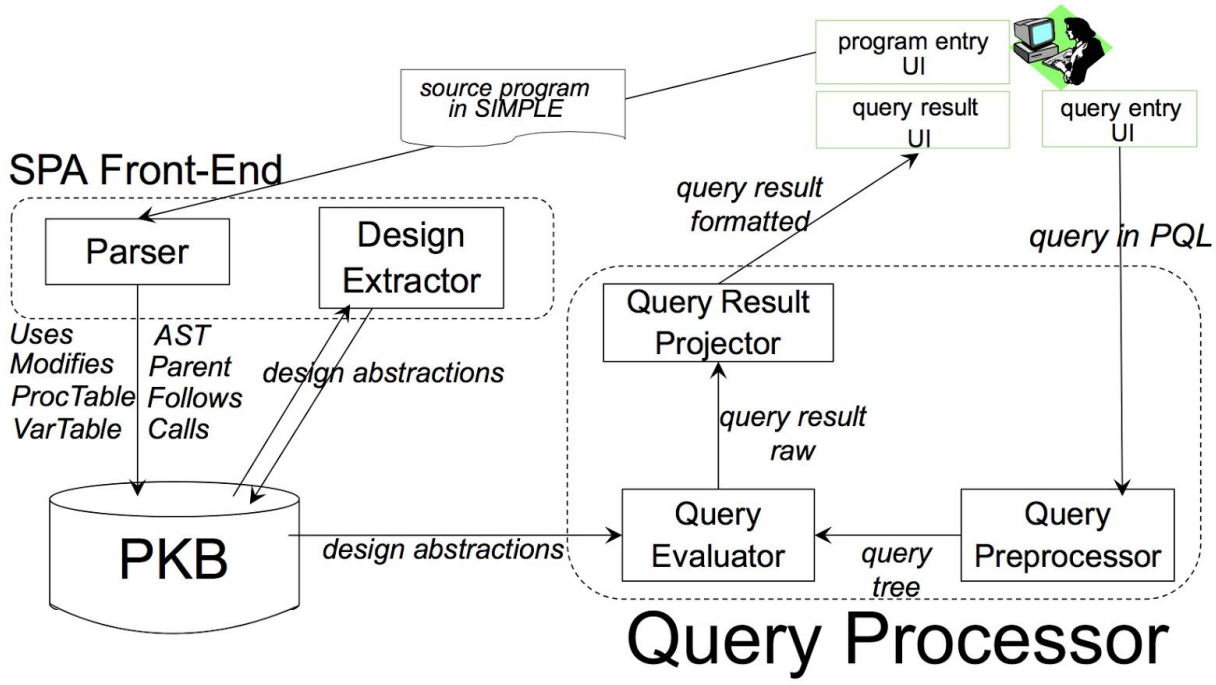

Figure 3.1.1 Processing Flow of SPA

Our overall architecture is similar to the one in the handbook except SPA Front-End has only Parser. We made a design decision not to include a design extractor. Currently, Query Processor only consist of two sub-components, query preprocessor and evaluator. Basis for UI has yet to be implemented.

3.2 Design of SPA components

a. SPA front-end design

The SIMPLE program is read in and tokenized in the parser, and then checked for its syntax. The tokenizing, parsing and grammar rules are implemented in three separate classes (parser.cpp, SIMPLEParser.cpp, SIMPLERules.cpp).

The AST and tables are built in the main function.

b. PKB design

A double linked list is used to represent the AST, while a vector is used to represent the variable table and procedure table, and this accommodates the flexible size of SIMPLE program.

For Modifies, Uses and Calls, an adjacency list is used to store the relationships. It is implemented using an unordered map which represents all the relationships in entirety. This is for quick, random access by mapping keys to their relevant vectors. A vector is then used to represent the relationship a variable or a statement has. For example, for the relation Modifies, there exist two unordered maps - one keyed by statements, and the other keyed by variables. Each key then maps to the respective vector containing variables and statements respectively. This also applies to Uses and Calls relationships.

PKB is used when Parser calls to build AST and tables. PKB simultaneously stores the design abstractions. When Query evaluator needs to know if a relationship, variable or procedure exists in the program it will access the PKB to get relevant information.

c. Query Processing

Query Pre-Processor will first process the declared synonyms into an entity table. It then tokenizes the latter of the select clause into individual tokens based on various delimiters like white spaces, coma, brackets, quotations. Going through the tokens, validation will involve checking that the result clause has been initialized and exist in the entity table, followed by the necessary such that and pattern clauses that have the relationship and attribute conditions.

The conditions are referenced to a relationship table and semantic checks to ensures that all arguments conforms to the grammar and lexical rules before the query is valid and passed to the evaluator.

Query Objects that contains three arguments each are stored in a vector for the evaluator to do the logical evaluation.

In the Query Processor, the query is first validated through the preprocessor before evaluation. After the validation step, the query processor will attempt to get results from the PKB for each Query Objects which represents a querying clause. This is done through the logic component that will match accordingly to the case scenarios/permutations defined to determine the necessary functions to call from the PKB. Currently the AutoTester serves as the UI to represent and display the query result to the user in the form of a .xml file.

3.3 Component Interactions

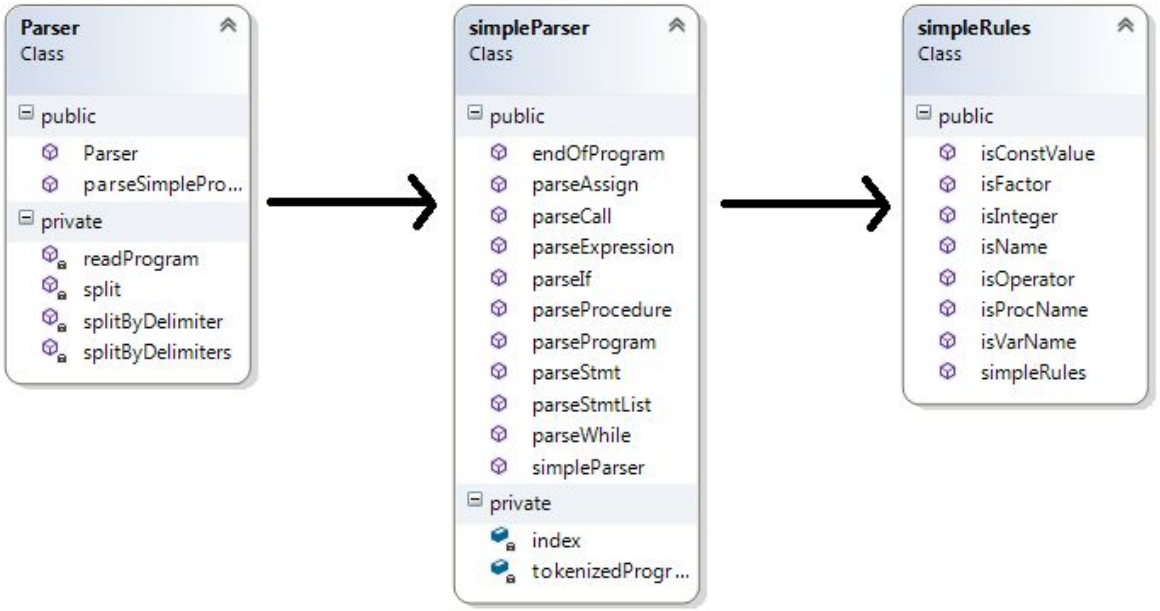

Figure 3.3.1. Parser class diagram

The Parser class creates a SimpleParser object to parse a SIMPLE program. The SimpleParser in turn creates a SimpleRules object to check grammar rules while parsing the program. This UML diagram tells other team members that they only need to use the main Parser class and its public method parseSimpleProgram, because the other two classes are used by Parser internally. SimpleParser's methods are made public for ease of unit testing, and SimpleRules' methods are made public so they can be used by SimpleParser.

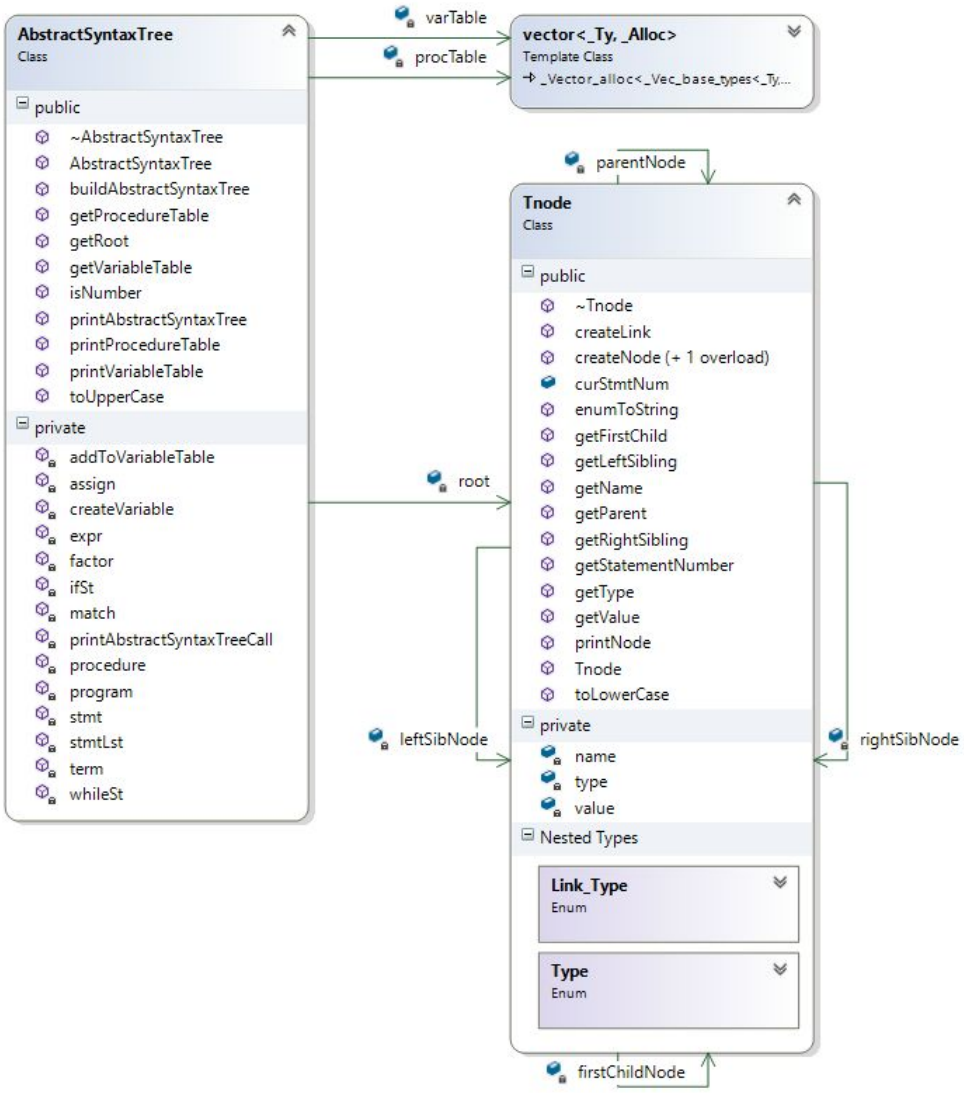

Figure 3.3.2. AST class diagram

The AST is built when the method buildAbstractSyntaxTree() is called by the main method, which passes the tokenized input from the Parser to the AST constructor. The main method then uses the AST object to create a double linked list of Tnode objects as well as a variable table and a procedure table, which can be accessed by the following methods: getRoot(), getVariableTable(), getProcedureTable()

The AST is then passed into the PKB through the PKB's constructor. The PKB then stores a pointer to the AST within itself, and calls another method to calculate relations using the AST which all happens internally during the PKB's construction. The PKB can now be passed into the QueryEvaluator object which also receives it as a parameter during construction.

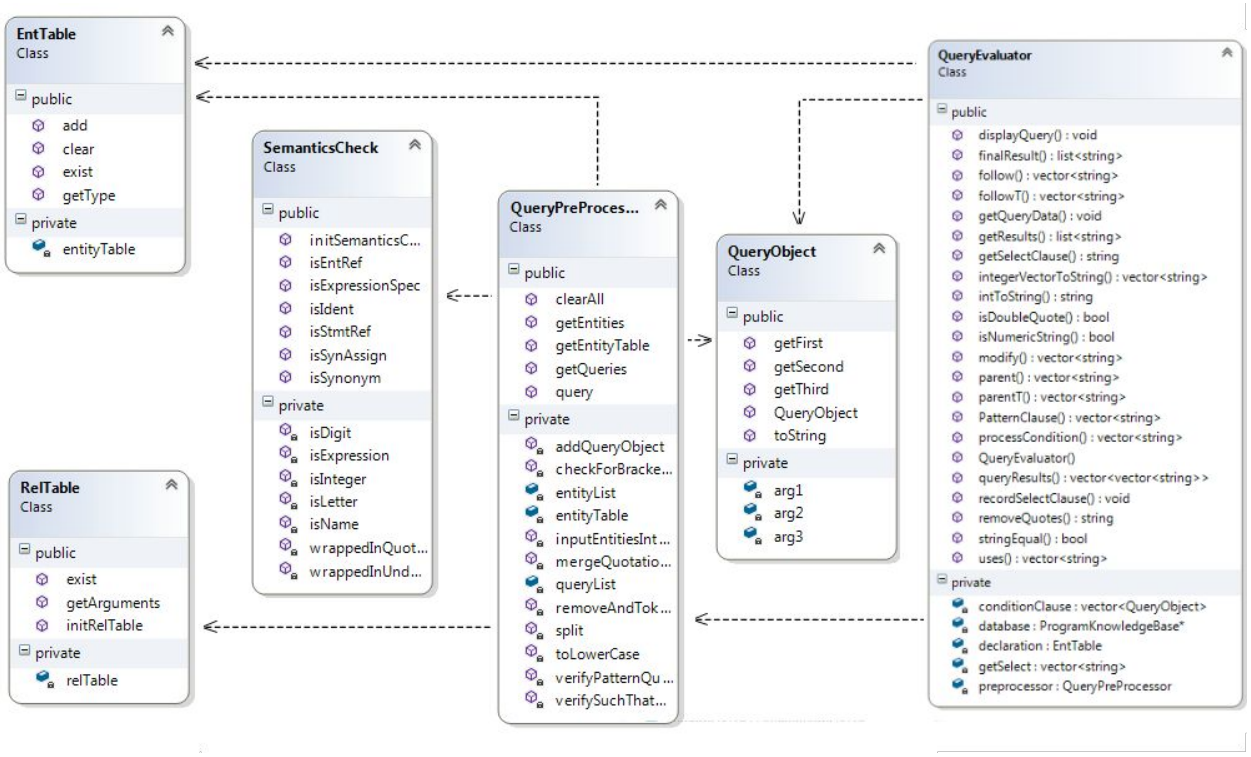

Figure 3.3.4. Query Processor class diagram

The query string is validated based on the method query(). By relying on semantics check and the relationship table, the query is validated, creating an entity table, vector of entities and vector of query objects for the query evaluator to do the logic processing.

By understanding what each sub component requires during the integration, we are able define the class objects for storing our clauses along with their arguments and also passing a list of entity that can be verified based on the entity table.

4. Documentation and coding standards

Naming Conventions

Variable names use camelCase, that is they must start with lowercase and capital letter for each following word
e.g. temp, delim, argVector

Method names must be verbs and use camelCase, that is they start with lowercase and capital letter for each following word
e.g. buildAbstractSyntaxTree(), getQueryData(), getArguments()

Loop variables by default should be set as i while j,k etc are used for nested loops
e.g. for(size_t i =0; i < argVector.size(); i++) { for(size_t j=0; j<i; j++ {...} }

Avoid using abbreviations for method names
e.g. addQueryObject(); //Instead of addQueryObj();

File Conventions

C++ header files should have extension .h and source files should have extension .cpp
e.g. QueryObject.h, QueryObject.cpp

All definitions should be placed in source files.

e.g. in the header file:
class ProgramKnowledgeBase{
public:
bool isContainer(Tnode*); //Instead of bool
isContainer(Tnode*) {return true;}

Include statements should be grouped, sorted with empty line between group of similar statements and only be located at the top of a header file.

e.g.

```
#include “AbstractSyntaxTree.h”
#include “Tnode.h”

#include <string>
#include <iostream>
```

Layout

Class declaration should be in the following form.

e.g.

```
class ProgramKnowledgeBase{
public:
            ...
protected:
            ...
private:
            ...
};
```

Comments

All comments should be written in English and explains what the code does instead of how it is being done.

e.g. `//Helps to group quotation expressions into one token e.g. "x + y + z" -> "x+y+z"`

5. Testing

Testing is divided into 3 parts - System (Validation) Testing, Unit Testing and Integration Testing.

For System Testing, the test cases are designed in order of the complexity of Simple programs. They are also intended for the prototype. Meaningless and invalid queries are kept minimal. For example, “Select a, b, c, d, e, f, g, h, i such that …” is not within the test scope. We assume users will select a reasonable number of variables. The testing focus for the prototype is to assess for plausible queries.

We use automated testing (AutoTester) to facilitate this process. The given AutoTester library is always up to date with each sub-iteration of developing SPA prototype. It also saves a lot of time when doing regression testing. Going through all the test cases again with every new bug fix will uncover if the system has regressed or any components that initially works fail.

Unit Testing (see UnitTest VS project) examines the functionality of each part (mentioned in the 1. Development Plan). One example is the printing of the AST in command line to check if the generated tree is correct.

The Integration Testing is done gradually as follows:

1. Parser and AST
2. Parser, AST and PKB
3. Query Preprocessor and Query Evaluator
4. Query Preprocessor, Query Evaluator and PKB
5. All components (main.cpp)

There are a few important learning points on testing. Define test scope early and more importantly, clearly. Implication of not doing that is repeated refining of test cases. Next, always rely on automated testing tools; it saves everybody’s time.

5.1 Test plan

The first simple program (Simple01-Source.txt), or Index program consists of assign and call statements only. In other words, there are no container statements in the program. This is to test the basic requirement of SPA.

The implication is, queries for parent and parent* will return none. Follows and Follows* queries are straightforward queries.

The second source code (Simple02-Source.txt), or First program is adapted from the Handbook.

Lastly, the third program (Simple03-Source.txt) or kitkat program contains many container statements for the sake of complexity.

5.2 Test Case Examples

a. Unit Testing

1. SPA Front-end

Parser example 1

```
TEST_METHOD(testGoodAssign) {
	vector<string> stmt;
	stmt.push_back("x");
	stmt.push_back("=");
	stmt.push_back("y");
	stmt.push_back("+");
	stmt.push_back("1");
	stmt.push_back(";");

	simpleParser *p = new simpleParser(stmt);
	Assert::AreEqual(true, (*p).parseAssign());
}
```

Test Purpose: Ensure that the parser correctly parses a valid assignment statement.

Required Test Inputs: The Parser has to be initialized, and a vector of a tokenized assignment statement has to be provided to it.

Expected Test Results: true (indicates it is syntactically correct)

Parser example 2

```
TEST_METHOD(testEmptyProcedure) {
                vector<string> proc;
                proc.push_back("procedure");
                proc.push_back("Empty");
                proc.push_back("{");
                proc.push_back("}");

                simpleParser *p = new simpleParser(proc);
                Assert::AreEqual(false,
(*p).parseProcedure());
          }
```

Test Purpose: Ensure that the parser correctly parses an invalid (empty) procedure.

Required Test Inputs: The Parser has to be initialized, and a vector of a tokenized procedure has to be provided to it.

Expected Test Results: false (indicates it is syntactically incorrect)

AST example 1

```
TEST_METHOD(TestNodeCreation) {
	Tnode *testNode;
	testNode = Tnode::createNode(5);

	Assert::AreEqual((string)"", testNode->getName());
	Assert::IsNull(testNode->getParent());
	Assert::IsNull(testNode->getFirstChild());
	Assert::IsNull(testNode->getRightSibling());
	Assert::IsNull(testNode->getLeftSibling());
	Assert::AreEqual((int)Tnode::CONSTANT,
(int)testNode->getType());
	Assert::AreEqual(5, testNode->getValue());
}
```

Test Purpose:	Ensure node creation is correct, in this case, a node that is a constant
Required Test Inputs:	A static method is called to create the node with the required parameters
Expected Test Results:	true

AST example 2

```
TEST_METHOD(TestLinkCreation){
	Tnode *T1;
	T1 = Tnode::createNode(Tnode::PROCEDURE, "myProc");

	Tnode *T2;
	T2 = Tnode::createNode(Tnode::STMTLST, "");

	bool link1;
	link1 = Tnode::createLink(Tnode::PARENT, *T1, *T2);

	Assert::AreEqual(true, link1);
```

```
    Assert::AreEqual((int)T2, (int)T1->getFirstChild());
    Assert::AreEqual((int)T1, (int)T2->getParent());
}
```

Test Purpose:	Ensure links created between nodes are correct, in this case, a parent-child linkage
Required Test Inputs:	2 nodes are created, before calling a static method to form the link
Expected Test Results:	true

2. Query Processor

Query Preprocessor example
`Assert::IsTrue(qpp.query("assign a; Select a such that Modifies (a, \"y\") and Pattern a (\"m\", _)"));`

Test Purpose:	Ensure that the query string is grammatically correct and the arguments are valid.
Required Test Inputs:	The QueryPreProcessor has to be initialized before it is able to call the query method that takes in a string input.
Expected Test Results:	true

Query PreProcessor/Evaluator example
Assert::AreEqual(et.getType("s"), (string("stmt")));

Test Purpose: Enables both the query preprocess and evaluator to check that entities are declared.

Required Test Inputs: A query string will have to be be queried by the query preprocessor for the generation of the entity table.

Expected Test Results: true

b. System (Validation) Testing

Example 1 in AutoTester format
1 - Combine Uses and pattern with same assignment stmt assign a; variable v; Select a such that Uses(a, "x") and pattern a(v, _) 7, 10, 11, 13, 15 5000

Test Purpose: To test if combined queries work for all components.

Required Test Inputs: Simple02-Source.txt is needed

Expected Test Results: Statements #7, #10, #11, #13 and #15

Example 2 in AutoTester format
2 - Test if subexpressions in pattern are recognized assign a; Select a such that pattern a(_, _“y”_) 5 5000

Test Purpose:	This tests both AST and Query processing. Query Parser should identify the position of underscore and evaluator should be able to find the relevant subtree from the AST.
Required Test Inputs:	Simple02-Source.txt is needed
Expected Test Results:	Statement #5

6. Discussion

Parser

Tokenizing was straightforward because I simply used the same function to split the program by all non-alphanumeric characters. Writing non-recursive grammar rules, such as the definition of a VarName, was straightforward, but writing recursive grammar rules, such as the definition of an Expression, took a while because I had to check that the preceding and following tokens were correct. Integration with the AST component was straightforward because we only had a small API to agree on (that the tokenized program would be in the form of a string vector) and it was agreed on before I started writing the parser, so I wrote the parser with that in mind.

If I started the project again, I would have written unit tests much earlier, to aid me in thinking of potential pitfalls when writing the parser, as well as its overall structure.

AST/Tables

Building the AST was initially confusing for me as it has been a while since I played with pointers. But once I got familiar with the pointers, everything became much simpler and the tips given in the handbook makes figuring the logic much easier. Thus the AST code is completed rather quickly.

Integration wasn't an issue as the API was simple. The only problem I faced was testing whether the AST generated is correct. It seems there is not much I can do except to print out the AST and check the whole structure. The tables were added after our 3rd consultation, and this task was more or less trivial since I have the AST code running properly.

To be honest, if I were to start the project again, I will probably stick to most of what I have done since I felt that the coding for my part went rather smoothly, but I would add a function to flatten the expression parts of the AST to aid in my debugging and unit testing. On a whole, I think we could have finished our respective code and unit testing early, so that we can do integration testing as early as possible.

PKB

Building the relations in the PKB was tedious as I had to wrap my head around the AST and figure out how to calculate the relations. There wasn't a straightforward answer. To get through this, I inefficiently spent time writing functions iteratively - as I coded, I would discover better ways to do it. This involved writing many helper functions to access and navigate the AST more easily. In retrospect, I should have focused on the core functions to calculate the relations and asked for more navigation functions to be written into the AST instead of doing it by myself.

Testing was also an issue for me. As I'd spent too much time on auxiliary functionality, I had little time left to spend on other more important factors such as testing my code. This was particularly the case for unit testing as the PKB depended on other components such as the AST which depended on the Parser. This led to ineffective debugging through the AutoTester as there was no stack trace.

The two problems together made for a terrible combination as this led me to having 700 lines of untested code. As a team, we should have set tighter intermediary deadlines and have better communication.

Query Preprocessor

Parsing the query was time consuming but manageable. The more problematic part was handling corner cases like tracking failed declarations, making sure that brackets, commas, quotation marks exists and their respective variables and underscores are tokenized properly.

Discussing the required API between preprocessor and evaluator early allowed me to understand the exact objects that the evaluator will require which saved a lot of time before implementing.

If given a choice, I will write more corner cases at the beginning so that some functions might be structured differently and start integration early.

Query Evaluator

The Query Evaluator is a crucial part of the program since it has to interact with both the preprocessor as well as the PKB. Currently, initializing the preprocessor as an instance is fine and is able to validate the correctness of the PQL grammar rules. The problem lies with discussing and planning the required APIs with the PKB at the early stages and clarifying what the parameters the PKB needs as well as how to access it. The method to initialise the PKB was also complex which requires me to create a constructor to store a pointer to it which is unconventional compared to the preprocessor.

If I would start the project again, I would do more follow up and clarification with the PKB and make sure I could perform simple operations like accessing data and initializing it at the start, through unit testing on my side.

As a team, we should meet up face to face weekly to feedback on the progress and to help out in the severe problematic areas. Inefficient mode of communication and time management issues such as loose deadlines and deliverables is an issue.

Testing

It has been easy to use AutoTester with the given “AutomaticProjectTesting_Aug2015_VS2015” folder and the task of designing test cases is overall manageable. However, I found it difficult to design queries progressively from fundamental queries such as “variable v; Select v” to complex queries. More often than not, I find myself writing intuitive queries first - queries that users will most likely ask. Consequently, it is difficult to stay within the scope of the prototype’s grammar definition. Many times, the test cases went out of the margins of the prototype’s grammar definitions and I end up redoing the test scenarios.

I would spend more time in planning my three test cases and read instructions carefully before writing test cases.

As a team, we should have integrated SPA and AutoTester earlier.