

**National University of Singapore**

**School of Computing**

**CS3202: Software Engineering Project II**

**TEAM 05: Flying Cockroach**

Semester 1, AY2014/2015

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

Consultation Day/Hour: Monday 6-6.30pm

Contents

[1. SPA 3](#_Toc402389959)

[1.1. Architecture 3](#_Toc402389960)

[1.2. Interaction 4](#_Toc402389961)

[2. Summary of Achievements 6](#_Toc402389962)

[2.1. Basic SPA 6](#_Toc402389963)

[2.2. Bonus Features 7](#_Toc402389964)

[2.2.1. Flexible CodeParser 7](#_Toc402389965)

[2.2.2. Flexible QueryParser 8](#_Toc402389966)

[2.2.3. Highly Organised Repository 9](#_Toc402389967)

[3. Project Plan 13](#_Toc402389968)

[3.1. Schedule For Whole Project 13](#_Toc402389969)

[3.2. Comments & Problems 13](#_Toc402389970)

[4. Components 14](#_Toc402389971)

[4.1. Code Parser 14](#_Toc402389972)

[4.2. PKB 15](#_Toc402389973)

[4.3. Design Extractor 16](#_Toc402389974)

[4.4. Query Processor 17](#_Toc402389975)

[4.4.1 Query Processor 17](#_Toc402389976)

[4.4.2 Query Parser 17](#_Toc402389977)

[4.4.3 Query Evaluator 18](#_Toc402389978)

[5. Testing 20](#_Toc402389979)

[6. Coding Standards 21](#_Toc402389980)

[7. Project Evaluation 22](#_Toc402389981)

[8. API 23](#_Toc402389982)

# SPA

Static Program Analyser (SPA) is a program to answer queries about an input SIMPLE program. In this report, we will be describing the design and implementation decisions made during the development of the SPA during CS3201 and CS3202.

## Architecture

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



Figure 1

## Interaction

/\* **DINDA**

PLEASE REMOVE THIS COMMENT

Draw UML diagrams that you found useful. For each diagram that you draw, explain how you used it (e.g., in project planning, communication, test planning or in other situations), and comment on the value a diagram added to your project.

\*/

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

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



Figure 2

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



Figure 3

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

# Summary of Achievements

## Basic SPA

/\***IPSITA**\*/

## Bonus Features

We have extended the features of SPA beyond its call of duty. The features will be explained in more detail below:

* + 1. Flexible CodeParser

According to the handbook, the source code that is to be tested against our SPA, is defined to be in standardized format and neatly arranged. They have regular and consistent spacing, indentation, tabs, and endline characters. Please refer to the example below:

procedure Orchid {

while idx {

y = z\*3 + 2\*x;

call Tulip;

idx = idx - 1; }

z = z + y + idx; }

procedure Lily {

while v {

y = x + y; }

x = y + x; }

However, our SPA source code parser is very flexible. It is able to detect erroneous and inconsistent spaces, tabs, endline characters and erroneous close curly brackets. A flexible CodeParser is, thus, one of our bonus features, as it was not required for us to have in the basic SPA. To demonstrate this ability, please refer to the example below (which will have the same extracted information as the code fragment above):

procedure Orchid

{

while idx {

y = z \*3 + 2\* x ;

call Tulip ;

idx = idx -1;

}

z=z+y+idx;

}

procedure Lily{

while v{

y =x + y ;}

x = y + x;}

### Flexible QueryParser

The queries that are to be tested against basic SPA are defined to be in a standard format. They have regular and consistent lowercase/uppercase command, spacing, and characters. Please refer to the sample standard query below:

if ifstat; Select ifstat such that Follows\* (ifstat, 17)

assign a; Select a such that Modifies (a, "idx") and Uses (a, "idx")

assign a; while w; Select a such that Modifies (a, "idx") and Uses (a, "idx") and Follows (15, a) and Parent\* (w, a)

However, our QueryParser is very flexible and is able to detect queries with inconsistent/extra spaces and lowercase/uppercase clauses. As it was not required for us to have such flexibility for CS3202’s basic SPA, we have this advanced feature. To demonstrate this ability, please refer to the example below (which will have the same information/result as the example above):

If ifstat ; Select ifstat such that Follows\* ( ifstat,17)

assign a;Select a such that Modifies(a,"idx") and Uses(a,"idx")

assign a ; while w ; Select a such that Modifies (a, "idx" ) and Uses ( a , "idx") and Follows (15, a) and Parent\* ( w , a )

### Highly Organised Repository

In our shared repository, our code, report files and directories are very well organised. We do this by following standard naming conventions and following a structured hierarchy such that each team member has fast and easy access.



Appropriate milestones and issues are also tracked and reported regularly to achieve any goals or/and objectives.





Every commit related to certain issues or milestones was tagged and linked to the issue for easy tracking and reference in the future. It was also used to monitor the progress of any issues and the SPA system as a whole. Github features such as milestones, issue tracker with assignees, milestones, and issue category (bugs, testing, documentation, enhancement, etc) were highly useful for our project management process.





Overall, our code and API are well-documented and in sync with each other. Every detail of the code is described clearly, showing and explaining all methods, attributes, and inheritance diagram whenever applicable.



Overall, our progress in building this project was conscientiously and thoroughly tracked and documented so that any new addition to the team could understand the project. Also, by posting and tracking issues, we can determine who tackled what.

Our project is currently hosted on GitHub, and the relevant links are as follows:

* Repository : <https://github.com/yulonglong/Static-Code-Analyzer>
* Issue tracker : <https://github.com/yulonglong/Static-Code-Analyzer/issues>
* Milestones : <https://github.com/yulonglong/Static-Code-Analyzer/milestones>
* Commits : <https://github.com/yulonglong/Static-Code-Analyzer/commits/master>

# Project Plan

/\* **DINDA**

PLEASE REMOVE THIS COMMENT

Describe how you organized project work, the actual schedule, etc. Organize your description into the following sub-sections:

**The actual schedule for the project, milestones**

Discuss problems encountered that affected project schedule.

**Any comments on division of work and project discussion meetings**

**\*/**

## 3.1. Schedule For Whole Project

## 3.2. Comments & Problems

# 4. Components

## 4.1. Code Parser

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

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

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

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

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

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

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

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

Example:

Source code (source1.txt):

procedure Mini {

A1 = 29;

a1 = 31;

i = 51; }

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

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

CodeParser’s Unit testing is done by checking the content of each table, whether it has set the tables properly, and by checking the contents of each node in the AST, whether it matches the expected AST.

## 4.2. PKB

/\***YOLIM**

PLEASE REMOVE THIS COMMENT

Follow guidelines in Handbook Section 10.2 to analyze, justify and document detailed design decisions. Pay attention to clarity of the description (check hints in Section 10.2).

If you applied design patterns, document them in this section:

a) Explain the design problem and pattern you applied

b) Document expected benefits and costs of applying a design pattern

c) Document the actual benefits and costs of a design pattern that you experienced in the project after applying it.

\*/

## 4.3. Design Extractor

/\***IPSITA**

PLEASE REMOVE THIS COMMENT

Follow guidelines in Handbook Section 10.2 to analyze, justify and document detailed design decisions. Pay attention to clarity of the description (check hints in Section 10.2).

If you applied design patterns, document them in this section:

a) Explain the design problem and pattern you applied

b) Document expected benefits and costs of applying a design pattern

c) Document the actual benefits and costs of a design pattern that you experienced in the project after applying it.

\*/

## 4.4. Query Processor

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

### 4.4.1 Query Processor

/\***HISYAM**

\*/

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

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

### 4.4.2 Query Parser

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

**Query Validation (Grammar check)**

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

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

1. The code is much neater and simpler
2. It is simpler to define the book grammar rule in regex, hence less prone to mistakes
3. It is faster to detect errors rather than parsing and validating (especially if the error is towards the end of the query)
4. Query parsing becomes simpler if we already know the exact possible format of the string query that needs to be parsed.

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

As an example, consider this valid query

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

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

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

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

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

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

Select statement will be broken down into such-that, with, or pattern clauses, whose parameters will be checked against the grammar rule.

Consider statement number 3 from the example above. Query validator will use regular expression to check the validity of the statement and retrieve the appropriate tokens.

If the regular expression matching fails, (for example, the number of arguments in the clause is not exactly two) the validator will instantly terminate and declare the query invalid. In the case where statement is valid, all the tokens from the select statement will be stored in a vector, *selectStatement*. This vector will be accessed by the parsing function later on. For efficiency, the *selectStatement* vector will only contain relevant tokens from the statement. Therefore, the unnecessary syntactic punctuation will be removed.

From the example above, the value of *selectStatement[] (dynamic array, c++ vector)* will be:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Select | a | such | that | Follows | w | a | pattern | a | x | \_”x+y”\_ |

**Query Parsing (Semantic check)**

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

|  |
| --- |
| **Query** |
| *vector<string>* selected-synonym  *vector<Relationship>* relationship-vector  *map<string, synonym-type>* synonym-table |

Since each query can contain many select clauses, the selected synonyms and clauses are stored inside a vector for scalability purposes. All the synonyms present in the *selected-synonym* vector will be detected and validated once again on whether they have been declared earlier. If it is not declared, the query is invalid and **QueryParser** will indicate and return invalid. The string vector will then be stored in a **Query** class.

The synonym map that was created earlier will also be included in the **Query** object. Both such-that and pattern clauses will be stored as another object, **Relationship**, as the following.

|  |
| --- |
| **Relationship** |
| *enum* relationship-type  *enum* token-type  *string* token1  *string* token2  *TokenType* token1-type  *TokenType* token2-type  *string* pattern-synonym |

**Relationship** arguments will also be semantically checked to determine if they are valid. For example, both arguments in Follows clause have to be a statement (i.e. stmt, while, if, assign, call). If the arguments contain a constant synonym (e.g. constant c; while w; Select w such that Follows (c,w), then **QueryParser** will detect the error and return invalid.

The types of the tokens/arguments are detected by **QueryParser** and store them in the **Relationship** class. For example, if the token is “Second”, then its type will be IDENTIFIER, if the token is 1, then its type will be INTEGER, and if the token is w, then its type will be SYNONYM.

From the valid example above, the select-statement vector will be processed to produce the following.

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

|  |  |
| --- | --- |
| **rel1** | |
| relationship-type | FOLLOWS |
| token1 | w |
| token2 | a |
| token1-type | SYNONYM |
| token2-type | SYNONYM |

|  |  |
| --- | --- |
| **rel2** | |
| relationship-type | PATTERN |
| token1 | “x” |
| token2 | \_”x+y”\_ |
| token1-type | IDENTIFIER |
| token2-type | UNDERSCOREEXPR |
| pattern-synonym | a |

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

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

**With-clause**

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

For example:

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

p.procName = ”Second” will be parsed into two parts, left-hand-side and right-hand-side. Right-hand-side includes “Second” and will be stored in token2/argument-2 in a relationship object. Left-hand-side includes p.procName and will be stored in token1/argument-1 if token is valid. Synonym p will be checked against the map whether it exists. Since the attribute name is procName, it will also be checked on whether it is of type procedure.

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

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

|  |  |
| --- | --- |
| **rel2** | |
| relationship-type | WITH |
| token1 | p |
| token2 | “Second” |
| token1-type | SYNONYM |
| token2-type | IDENTIFIER |

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

However, for special cases when the synonym is of type call. There will be another attribute to represent whether call refers to the procedure name (c.procName) or statement number (stmt#). **QueryEvaluator** will then check the additional attribute before evaluating the clause.

**Multiple Clauses**

For multiple clauses (such that, pattern, with), Query Parser works by keeping the previous clause keyword (e.g. “pattern”, or “with”), and use it to detect the clause type when it encounters “and”.

For example:

assign a; Select a pattern a(\_,\_) and a(“x”,\_”x+y”\_);

Parser will detect the first clause and store in the relationship class accordingly

|  |  |
| --- | --- |
| **rel1** | |
| relationship-type | PATTERN |
| token1 | \_ |
| token2 | \_ |
| token1-type | UNDERSCORE |
| token2-type | UNDERSCORE |
| pattern-synonym | a |

It also keeps track of the last relationship-type. Therefore, when Query Parser reaches the word “and”, it knows that it will be parsing a pattern clause again, translating the “and” keyword into “pattern”. Query Parser will then validate and parse accordingly with respect of the clause type, in this case it is “pattern”.

|  |  |
| --- | --- |
| **rel2** | |
| relationship-type | PATTERN |
| token1 | “x” |
| token2 | \_”x+y”\_ |
| token1-type | IDENTIFIER |
| token2-type | UNDERSCOREEXPR |
| pattern-synonym | a |

Both relationships will then be stored in a **Query** class and passed to **QueryEvaluator**.

### 4.4.3 Query Evaluator

/\***LACIE**

PLEASE REMOVE THIS COMMENT

1. Describe data representation for program queries

2. Describe your strategy for Basic Query Evaluation (BQE)

3. Describe optimizations

4. Discuss detailed design decisions regarding BQE and optimizations

\*/

/\***HISYAM**

PLEASE REMOVE THIS COMMENT

1. Describe data representation for program queries

2. Describe your strategy for Basic Query Evaluation (BQE)

3. Describe optimizations

4. Discuss detailed design decisions regarding BQE and optimizations

\*/

# 5. Testing

/\* **DINDA**

PLEASE REMOVE THIS COMMENT

**Testing: Group-PKB and Group-PQL**

Describe your testing experience (not ex ceeding TWO pages).

\*/

# 6. Coding Standards

/\***IPSITA**

Include experiences

\*/

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

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

**Standards between abstract APIs and concrete APIs**

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

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

# 7. Project Evaluation

/\***YOLIM** \*/

1. How would you improve your SPA if more time was available?

2. What would you done differently if you were to start project again?

3. Comment on the experience gained in this project in respect to: a) working in the team,

b) incremental development,

c) complexity of the SPA problem and program solution,

d) what did work well?

e) what did not work well?

f) what did you learn in this project course?

4. Comment on the tools used for the project a) Were the recommended tools useful?

b) What other tools did you use (if any), and in what ways were they useful?

c) What were the problems you faced when using each tool?

d) In which areas would you like to have had more tool support?

/\***IPSITA** \*/

5. What management lessons have you learned?

6. What advice would you give to the students who will take this course in the future?

7. Suggest how we could improve this project course.

8. Discuss any other experiences.

9. Comment on Handbook

# 8. API

Please view our Doxygen at:

[www.comp.nus.edu.sg/~kester/cs3202](http://www.comp.nus.edu.sg/~kester/cs3202)