CS3203 - Software Engineering Project

Iteration 1 Report

Team 24

Name	Matriculation No.	Email Address	Contact No.
Ang Wei Neng	A0164178X	weineng.a@gmail.com	9114 4933
Jeffery Kwoh Ji Hui	A0173005U	jefferykwoh@gmail.com	+1 415 937 6369
Tay Hui Chun	A0170109N	tayhc@u.nus.edu	91824098
Vignesh Shankar	A0167354Y	vigneshshankar@gmail.com	83383292
Yang Sihan	A0177161E	sihanyang@u.nus.edu	98967856
Yang Yang	A0168815U	yang.yang@u.nus.edu	92972030

1. Scope of the Prototype Implementation

This prototype meets the full specifications for iterations one. In particular, it is able to accept a SIMPLE source program and respond to simple queries on the source program.

The SIMPLE source program must have one procedure and can include read, print, call, while, if and assignment statements.

Queries must have a single synonym for its return value, at most one such-that clause and one pattern clause. Supported relations for such-that clauses are Follows, Follows*, Parent, Parent*, Uses and Modifies. Pattern clauses must reference assignment statements.

2. Development Plan

X = main contributor(s)

x = made contributions

Component	Lexer	Parser	Design Extractor	PKB	Query Evaluator	Query Pre-processor	Systems Testing
Wei Neng	x	X	X	X	x		
Vignesh	X	X	X	X			
Jeffery	x			x		X	x
Hui Chun	x					x	X
Yang Yang					x		
Sihan					X		X

Component	Documentation	Tooling	Planning
Wei Neng	x	X	
Vignesh	x		
Jeffery	x	X	X
Hui Chun	X		x
Yang Yang	X		x
Sihan			

The primary objective of each sprint are as follows:

- Sprint 1 (21 August - 4 September): Planning and setting up team workflows
    - By doing necessary planning, setting up tools and establishing expectations, we will lay the groundwork for smooth operation of the team in the coming weeks
    - Minimal actual implementation at this stage as that would likely result in code that will have to be discarded or changed later
    - PKB team can begin work on Lexer (simple predefined role of turning string into tokens) and code scaffolding that will remain flexible in the face of possible design decision changes
- Sprint 2 (5 September - 18 September): Implement basic functionalities
    - Once APIs and objects such as the Query struct and AST are well established, implementation can begin with the expectation that changes can largely be limited to each component
    - Epics for SFE, PKB and QP are broken down into more manageable issues to be dealt with one at a time
    - Establish code scaffolding for QP
    - Begin implementation of initial functionalities
- Sprint 3 (19 September - 24 September): Implement all remaining functionalities. Complete systems testing and documentation
    - Within the framework established in Sprint 2, we can quickly implement the remaining features necessary to meet the iteration requirements for iteration 1
    - Systems testing to check for bugs as iteration draws to a close

A more detailed breakdown of tasks can be found below.

Sprint 1 (21 August - 4 September)							
Task	Activity	Wei Neng	Vignesh	Jeffery	Hui Chun	Yang Yang	Sihan
Project Setting up Rationale: To configure the remote repository with the basic settings required for each team member, as well as setting some ground rules for project deliverables.	Sprint planning			✓	✓	✓	
	Set up C++ linter	✓					
	Set up master branch protection	✓		✓			
	Add pre-commit script for clang-format	✓					
	Set up respective CODEOWNERS for repository			✓			
	Establish project report skeleton and style				✓		
	Create README.md for Team Practices in creating PRs in repo			✓			
Designing of Abstract APIs Rationale: To discuss with the relevant team members after role allocation on the design considerations to be made when designing the abstract APIs.	Propose PKB API for QP	✓	✓	✓			
	Review PKB API for QP	✓	✓	✓		✓	
	Address API comments in Notion document	✓	✓	✓	✓	✓	✓
	Propose QE-QPP internal APIs				✓	✓	✓
	Propose Query struct design				✓	✓	✓

Planning of Tasks Rationale: Before we start implementation, it is good to have epics being broken down into granular tasks.	Plan Iteration 1 task breakdown for Autotester testing			✓			
	Plan Parser implementation work into ticket sized issues for visibility	✓	✓	✓			
	Design and Plan tasks for Query Preprocessor				✓		
	Design and Plan tasks for Query Evaluator					✓	
Basic Implementation of Skeleton Code for SFE and Parser Rationale: SFE and PKB implementation can be started first as it is non-blocking.	Add a Lexer	✓	✓				
	Add Parser Scaffold	✓	✓				
	Add PKB Scaffold	✓	✓				
	Parse SIMPLE statements in Parser	✓	✓				

Sprint 2 (5 September - 18 September)							
Task	Activity	Wei Neng	Vignesh	Jeffery	Hui Chun	Yang Yang	Sihan
Planning of Tasks and Tooling Rationale: Additional planning for PKB implementation work and tooling requirements.	Plan PKB implementation work into ticket sized issues for visibility	✓	✓	✓			
	Plan QP implementation work into ticket sized issues for visibility					✓	
	Set up CI for repo	✓		✓			

Implementation Work for Parser Rationale: To meet the requirements set by the SFE team outlined in the project board	Extract “log” function in Parser into a separate module		✓				
	Lexer changed to not have reserved keywords		✓				
	Add conditional statements (while, if/else) in parser	✓					
	Parse statements properly	✓					
	Parse assign properly	✓					
	Parse expressions properly	✓					
Implementation Work for PKB Rationale: To meet the requirements set by the PKB team outlined in the project board	Get statement numbers for the SIMPLE prog AST		✓				
Implementation Work for QPP Rationale: To meet the requirements set by the QP team outlined in the project board	Use Lexer to tokenize queries properly		✓		✓		
	Parse queries with no such that and no pattern clauses			✓			
Implementation Work for QE Rationale: To meet the requirements set by the QP team outlined in the project board	Add Query struct header file					✓	✓
	Implement support for Parent/Follows in Query Evaluator					✓	✓
	Implement support for Uses in Query Evaluator						✓
	Implement support for Modifies in Query Evaluator						✓

	Implement support for Pattern in Query Evaluator	✓					✓
Testing Rationale: To create test cases for PKB and QP, and let team members review them on Notion.	Add test cases for PKB	✓	✓				
	Add test cases for QP						✓
	Review PKB test cases	✓	✓				
	Python Autotester to run Autotester and raise failed test cases			✓			
Documentation Rationale: To update the project report on the abstract APIs and the architecture in the different components	Write SPA Design Overview				✓	✓	
	Transfer issues from GitHub project board to Development Plan				✓	✓	
	Design Parser + PKB + Design Extractor architecture				✓		
	Update PKB APIs for Follows and Parent to match Notion document		✓				
	Update PKB APIs for Uses and Modifies to match Notion document		✓				

Sprint 3 (19 September - 24 September)							
Task	Activity	Wei Neng	Vignesh	Jeffery	Hui Chun	Yang Yang	Sihan
Implementation Work for PKB & DesignExtractor Rationale: To implement the public facing APIs for the QE such as Follows/Follows*, Parent/Parent*, Uses and Modifies. This task involves both the PKB and the DesignExtractor because the PKB is dependent on the DesignExtractor to extract information from the AST.	Implement Follow* APIs	✓					
	Implement Parent* APIs	✓					
	Make PKB virtual	✓					✓
	Implement Uses		✓				
	Expose Uses API from PKB Implementation		✓				
	Implement Pattern API	✓					
	Extract Modifies in DesignExtractor		✓				
Implementation Work for QPP Rationale: To implement support for validating and parsing queries based on the SIMPLE grammar rules.	QP parsing process to be whitespace aware			✓			
	Implement equality in Query Struct			✓			
	Update lexer to be whitespace aware			✓			
	Parse relation(stmtRef, stmtRef) (Parent(*) / Follows (*))			✓			
	Implement parsing for Uses and Modifies relations			✓			
	Implement parsing for patterns			✓			

Implementation Work for QE Rationale: To implement support for evaluating queries that involve design abstractions such as Follows/Follows*, Parent/Parent*, Uses and Modifies.	Complete Query Evaluator Implementation						✓
	Add Unit Tests for Follows/Follows* for Query Evaluator						✓
	Evaluation of Parent/Parent* clause in Query Evaluator						✓
Documentation Rationale: To document details of every SPA component and what is required for the project report for Iteration 1.	Write description for SPA components				✓	✓	
	Write design considerations for SPA components				✓	✓	
	Update documentation on Testing section				✓	✓	
	Document Abstract APIs for Follow/Follows*, Parent/Parent*, Uses and Modifies				✓	✓	
System Tests Rationale: To run and test the System Tests that our team have drafted in the earlier Sprints.	Update Autotester CI to use System Tests			✓	✓		
	Update Autotester CI to expect parsing failure from malformed SIMPLE source			✓			
	Write System Tests				✓		✓

3. SPA Design

3.1 Overview

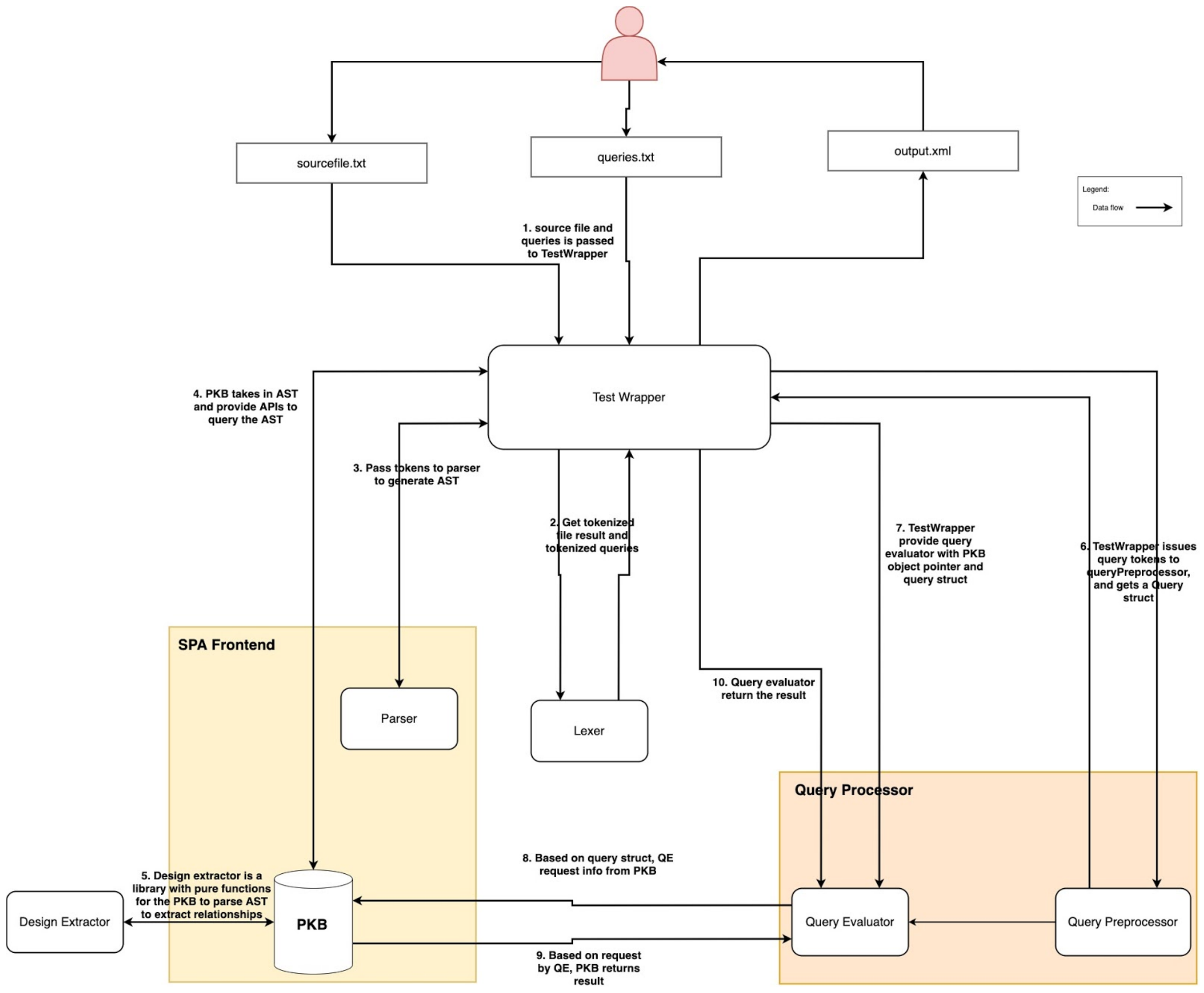

sourcefile.txt
queries.txt
output.xml
Legend:
Data flow
1. source file and queries is passed to TestWrapper
Test Wrapper
4. PKB takes in AST and provide APIs to query the AST
3. Pass tokens to parser to generate AST
2. Get tokenized file result and tokenized queries
7. TestWrapper provide query evaluator with PKB object pointer and query struct
6. TestWrapper issues query tokens to queryPreprocessor, and gets a Query struct
SPA Frontend
Parser
Lexer
10. Query evaluator return the result
Query Processor
8. Based on query struct, QE request info from PKB
5. Design extractor is a library with pure functions for the PKB to parse AST to extract relationships
Design Extractor
PKB
Query Evaluator
Query Preprocessor
9. Based on request by QE, PKB returns result

Figure 1. Overall Architectural Diagram-Arrows in this diagrams are data flow arrows, not dependency arrows

3.1.1 SPA Frontend

The SPA Frontend (SFE) mainly consists of four sub-components - Lexer, Parser, PKB and Design Extractor.

3.1.1.1 Lexer

The Lexer is responsible for analyzing the source program written in SIMPLE and converting it into tokens. The lexer is used by the Parser to convert the tokens into an Abstract Syntax Tree (AST).

3.1.1.2 Parser

The parser is responsible for parsing a stream of tokens generated by the lexer, and creating an AST. The parser also checks the syntactic validity of the SIMPLE program, where any invalid syntax will result in the parser throwing an exception.

3.1.1.3 Design Extractor

To allow for easy testing, we built our design extractor as pure function libraries which only depend on its inputs, and is referentially transparent. This allows the team to write unit tests to validate its correctness. The results are then used by the PKB to draw information from the parser's AST, such as the validity of the SIMPLE program and design abstractions.

3.1.1.4 PKB

The job of the PKB is to provide a set of APIs for use by the Query Processing team. The PKB component uses the design extractor as a library for parsing the AST, to generate any relevant data structure. The PKB only stores necessary information on the AST for fast resolution of results required by its exposed APIs to the Query Evaluator. More information on what the APIs are can be found in section 7.

3.1.2 Query Processor

The Query Processor (QP) mainly consists of two sub-components - Query Preprocessor and Query Evaluator.

3.1.2.1 Query Preprocessor

The Query Preprocessor (QPP) is responsible for transforming a query string into a Query struct for the use of the Query Evaluator. As tokenizing the input string is the first step in this process, the SFE’s Lexer sub-component can be reused for this purpose. The Lexer analyzes the query string and converts it into tokens which will be used by the QPP. The QPP performs syntactic and semantic validation and stores the relevant clauses into a Query struct to be used by the Query Evaluator.

3.1.2.2 Query Evaluator

The Query Evaluator (QE) evaluates the given Query struct by getting the required information from the PKB through API calls. Finally, the QE formats the results and returns it back to the user.

3.2 Design of SPA Components

3.2.1 Lexer

The Lexer is a shared component (much like a utility class due to similar responsibilities) by the Parser and the Query Preprocessor. The Lexer's role is to tokenize the SIMPLE source code / queries line by line based on a set of syntax rules (i.e. a *LBRACE* token represents a left brace) and return a list of tokens. Similarly, the Lexer's role for the Query Preprocessor includes tokenizing the query string into a list of tokens.

The lexer is called by the TestWrapper (lexer::tokenize(std::stringstream s))) to generate the list of tokens.

Example:

Program:

```
procedure Example {
  x = 1 + 1;
  y = 5;
}
```

Tokens:

```
PROCEDURE NAME LBRACE NAME EQ CONSTANT PLUS CONSTANT SEMICOLON NAME
EQ CONSTANT SEMICOLON RBRACE
```

Design Decisions & Justifications

Design Problem: Whether to use the same set of rules for tokenizing the SIMPLE source code and PQL query

Option 1: Fully reuse component with same set of rules for both SFE and QP
- **Pros**: Single Responsibility Principle is enforced
- **Cons:** Using the same set of rules restricts our coverage for the Lexer's test cases with regards to tokenizing PQL queries (i.e. Follows * is not a valid design relationship but has the same token form as a valid relationship in query)

Option 2: Modify some rules to meet use case requirements
- **Pros**: Coverage for Test Cases increases
- **Cons:** Increases redundancy as there are now two functions with the same responsibility, which violates the Single Responsibility Principle. Increased documentation is also necessary to ensure clarity

Option 3: Implement lexing for SFA and QP independently

- **Pros:** Ensures full independence and reduces coupling between major components
- **Cons:** Large amount of redundancy in code

Criteria: Single Responsibility Principle, Coverage for Test Cases, Ease of implementation

Final Design Decision: Option 2 is chosen as a good compromise between fully reusing the component and fully independent implementation. Compared to independent implementation, the additional coupling and documentation is deemed acceptable for the reduction in repeated work. Compared to fully reusing the component, behaviour can be modified to better meet requirements of use cases and allow more extensive testing.

3.2.2 Parser

The Parser is responsible for converting the list of tokens returned by the Lexer into an internal representation that stores the design entities / abstractions of the SIMPLE source code. For example, the Parser parses design entities statement types such as read, print, assign, call, while and if. It is implemented using Recursive Descent Parsing[1], which is a top-down approach to parse the SIMPLE program starting from the first non-terminal (program). Every function implementation corresponds to each of the concrete syntax grammar rules for SIMPLE. For example, for parsing while statement types, the parseWhile function would contain a parseCondition function to implement the grammar rules when defining a condition expression in the while statement. The result from parsing the SIMPLE source code is the construction of an Abstract Syntax Tree (AST), which is used by the PKB.

We deal with syntax error by using a try-catch mechanism. For example, in parseProcedure, we would expect the tokens "PROCEDURE, NAME, LBRACE, ...". In the event a procedure token is not preceded by a name, an error is thrown, indicating that parsing has failed. However, there are cases where we do not know what the next token should be. An example would be parseStatement. A statement could be a while-loop, if-then-else, or an assignment statement. In this case, we peek at future tokens to see which statement entity to parse.

The parser is called by the TestWrapper to generate an AST.

[1] https://en.wikipedia.org/wiki/Recursive_descent_parser

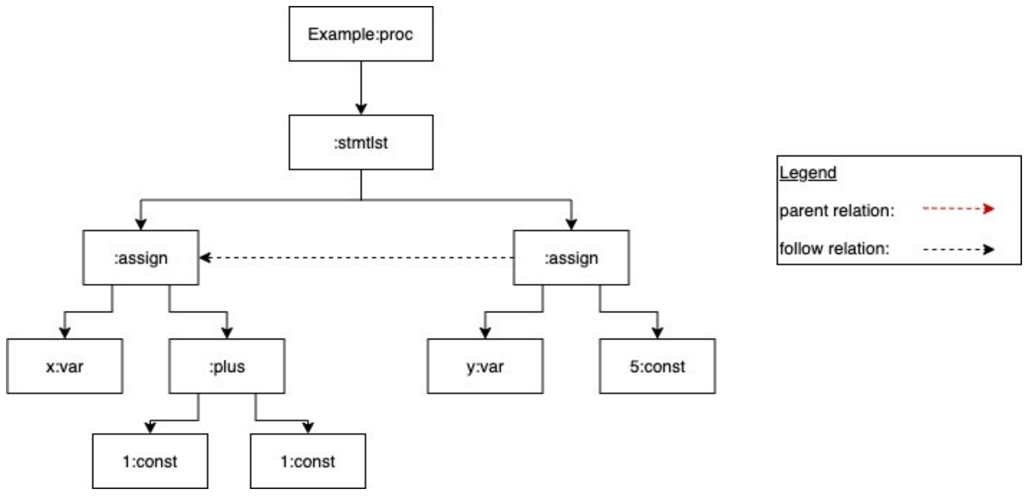


Figure 2. AST Generated by Parser

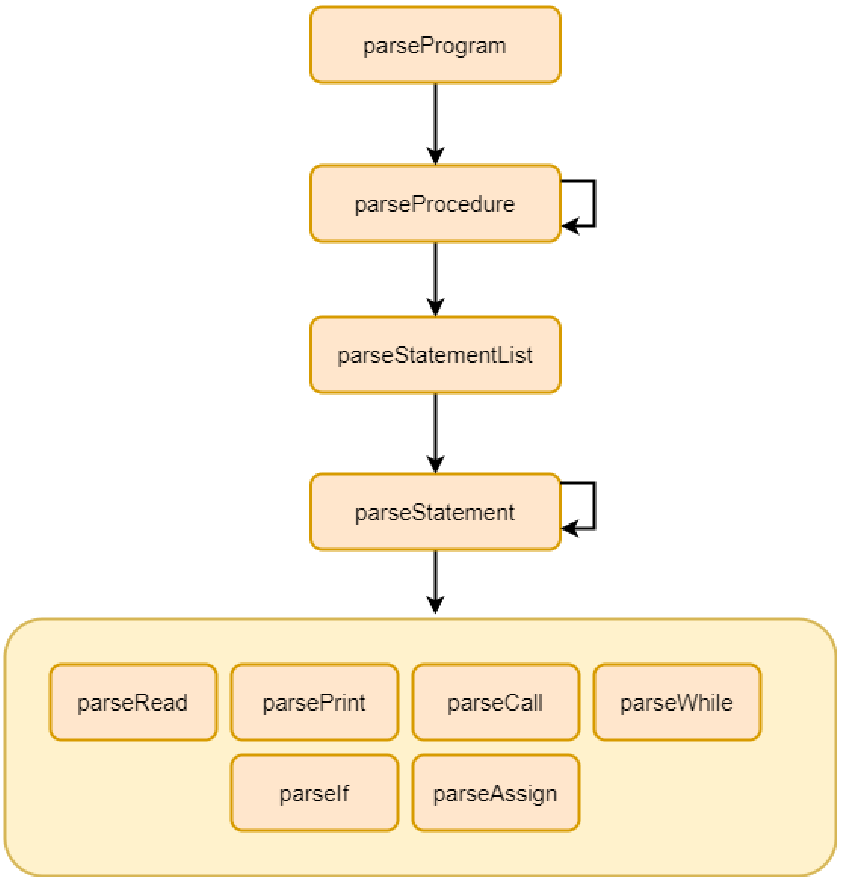


Figure 3. Architectural Diagram representing Parser functions

Design Decisions & Justifications

Design Problem: Data structure for populating the design entities / abstractions

Option 1: Abstract Syntax Tree (AST)

- **Pros**: Natural and complete representation of SIMPLE program. All relations within the program will be fully and accurately preserved. Better extensibility to support future relationship types. Allows for strong separation of concerns.
- **Cons:** Looking up relations using a tree requires tree traversal. This may be computationally expensive. As a tree, node objects and traversal methods will have to be implemented, adding to implementation time and difficulty.

Option 2: Lookup Tables

- **Pros:** Fast lookup, with O(n) access for standard arrays and tables and O(1) access for hash tables.
- **Cons:** Implementing logic to accurately and fully preserve all relationships in tables on parsing is non-trivial. We would also need to think of how the relationships are generated on-the-fly. (We need to determine how setUses and setModifies would work)

Criteria: Completeness and Correctness, Performance at Retrieval, Ease of Implementation, Ease of Increment.

Final Design Decision: Option 1 is chosen.

- In our implementation, an AST is first constructed by the parser to ensure that we have a complete and accurate internal representation of the SIMPLE source program. The DesignExtractor then populates lookup tables, implemented using hashmaps for O(1) access time, using the AST for common relations to improve query performance. Having an AST allows us to test that our internal representation was generated correctly, and with more context. While the construction of both AST and hash tables takes additional time, correctness is prioritized here.
- For option 2, any new retrieval information would mean more tables would need to be created, and thus require significant changes to the PKB. With an AST, there is no need to reparse the source program, and we simply need to add rules to query the AST to retrieve the new information required, making the AST the more extensible option.
- We also wanted our model to have strong separation of concerns. With the lookup table, we are forced to couple some query implementation together. For example, to get constants used in a particular statement, we would need to depend on setUses's result to generate the constants. With an AST, we simply look for the node representing the statement, and traverse down the node to look for any constant.
- While it would take a longer time to implement a parser for an AST, it is simpler since we would only need to specify the grammar rules, and allow the recursive decent parser to generate the AST for us.

3.2.3 PKB

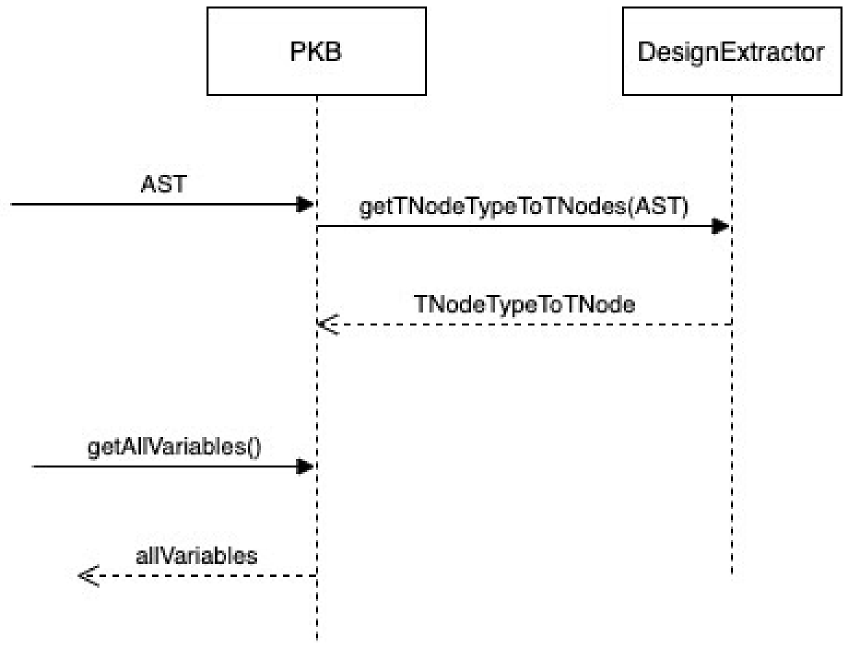
The PKB is a class with states based on the AST supplied to it.Based on the AST, it will pull results such as variables in the program, names of procedures, statement numbers of different statements etc.

Working in tandem with the Design Extractor, the PKB stores the processed data for fast retrieval when requested by the Query Evaluator. In essence, the PKB stores lookup tables generated from the AST, and returns the results to the Query Evaluator.

Unlike the traditional way of PKB taking a source file and parsing it, we generate an AST using the lexer (for tokenizing) and the parser (for parsing). This allows significant separation of concerns as the PKB is now only required to store relations and respond to relationship queries.

Example:

The following image is a sequence diagram indicating how to query the PKB. We first set up the PKB by generating a TNodeTypeToTNode mapping from the AST. To generate a list of all variables used in the PKB, we query the *TNodeTypeToTNode* mapping for *TNodes* with type *Variables.* This allows us to retrieve all the variables in the AST. In implementation, to boost performance, we perform more precomputation before returning the result.



Design Decisions & Justifications

Design Problem: PKB Structure
The PKB had to be designed in a way that allowed the Query Processing team to easily mock the public APIs.

Option 1: Both actual PKB implementation and mock PKB are derived classes of an abstract PKB class.

- **Details:** Set the base PKB class as the public facing API class for Query processor team. When the actual PKB class is needed, PkbImplementation object is used. When a mock PKB class is needed, PkbMock object is used instead.
- **Pros**: This allows development of both mock PKB and PKBImplementation independently, where we only need to add virtual methods to the abstract PKB class for both PKB to implement.
- **Cons:** There is now 3 classes for us to maintain; PKB, PkbImplementation, PkbMock

Option 2: PkbMock directly inherits from PkbImplementation

- **Pros:** Less code needed, as the mock only overrides methods that it implements.
- **Cons:** Hard to reason correctness. In the event PkbMock is not implemented, the PkbImplementation method might be called instead, which could be unexpected.

Criteria: Correctness.

Final Design Decision: The team decided on Option 1 as we believe that correctness is of utmost importance, even at the expense of additional workload. With Option 2, when tests are calling an unimplemented method of PkbMock, the program would crash.

3.2.4 Design Extractor

The design extractor is mainly responsible for parsing the AST and returns the processed information to the PKB.

Performs the following functions:

- Check for SIMPLE program correctness
  - Three conditions that are required of the AST is checked here:
    1. Two procedures with the same name is considered an error.
    2. Call to a non-existing procedure produces an error
    3. Recursive and cyclic calls are not allowed. For example, procedure A calls procedure B, procedure B calls C, and C calls A should not be accepted in a correct SIMPLE code.

- Generate design entity information
    - From the AST, generate all read, call, print statements
    - From the AST, generate all constants, variables and procedure names
- Generate design entity relationships
    - Follow:
        - From the AST, we look for all statement-list nodes. From each statement list, we set the Follow relation such that each child of a statement-list follow the previous child.
        - For the transitive relationship, we simply traverse the Follow relation.
    - Parent
        - From the AST, we look for all while-loop and if-then-else nodes. There, we set all children of its corresponding statement list as children, and we set all of its children's parents as the while/if-then-else loop.
        - For the transitive relationship, we simply traverse the Parent relation.
    - Uses
        - From the AST, we generate a topological order of the procedures, where a procedure can only appear after all the procedures that it calls.
        - We then process each procedure in the topological order one by one, recursively indexing the Uses information of the statements in the procedure
    - Modify
        - We generate the same topological order as in Uses.
        - We then process each procedure in the topological order one by one, recursively indexing the Modifies information of the statements in the procedure

In particular, the DesignExtractor extracts relevant design details and populates the lookup tables that the PKB references to respond to API requests from the QueryEvaluator. DFS is used for all traversals. DFS covers the entire tree and in an order conducive for evaluating Parents and Parents* relationships. A single traversal method keeps code familiar and clear. Tables are implemented using Hashmaps for quick access.

Design Decisions & Justifications

Design Problem: Consideration of whether design extractor is necessary

Option 1: Implement the design extractor logic into PKB

- **Details:** The main purpose of the design extractor is to abstract methods that the PKB uses. These methods can thus be reasonably be implemented as part of the PKB.
- **Pros**: One less component to manage, especially when the logic of the component could be subsumed into the PKB.

- **Cons:** The PKB would be extremely bloated, making development harder, as developers would need to comb through thousands of lines of code to understand the codebase.

Option 2: Separate design extractor component
- **Pros:**
    1) Separation of concerns: The main job of the PKB now is simply to serve data generated by the design extractor. It does not need to parse the AST.
    2) Testability: If there was no design extractor, we would only be able to test the public facing API of the PKB. This makes testing harder, as we would need to comb through a larger section of code if a test case fails.
    3) Maintainability: Having two different files serving two different roles (PKB serves data, design extractor parses AST), it is much easier for other developers to hop on and contribute.
- **Cons:** One more component to manage.

Criteria: Testability, Maintainability, Separation of Concerns

Final Design Decision: Option 2 was chosen. Managing an extra set of files is a small price to pay for the advantages listed above. Overall, this decision has resulted in significant savings in development effort.

3.2.5 Query Preprocessor

The Query Preprocessor accepts the tokenized query string from the Lexer and constructs and populates a Query struct (details below). Query validation is performed concurrently and checked against the PQL grammar rules from Basic SPA Requirements.

The Query Preprocessor is implemented as a recursive descent parser. Each nonterminal in the grammar is encoded as a parse function in the Query Preprocessor. In other words, `querypreprocessor::parseTokens` will call each possible non terminal rule in a top down manner. Whenever non-terminals are found, based on the context provided to the calling function--if any--the Query object is updated with new found information.

Whenever parsing for a non-terminal fails, the parser backtracks and iterates through all possible non-terminal rules at that point of parsing. Whenever the parser is deterministically unable to match any more rules, it will throw an error which will be caught in the `querypreprocessor::parseTokens` entrypoint. Then, an empty Query is returned to semantically represent an invalid PQL query.

Thus, as long as the tokens provided fail to parse, we can be guaranteed that the provided tokens do not adhere to the PQL grammar. By structuring our Query Preprocessor as a recursive descent parser, we are guaranteed that as long as the provided tokens parse, there are no syntactic errors.

Upon successful parsing, a Query object with all the information required to evaluate the PQL query is passed to the QueryEvaluator.

Design Decisions & Justifications

Design Problem: Data structure for representing a query

Option 1: Query Tree

- **Pros**: Natural and complete representation of Query, similar to AST for SIMPLE program. All relations within the query will be fully preserved. Easy to implement as nodes for the AST can be added and removed immediately upon encountering and backtracking from each non-terminal parse function.
- **Cons:** Accessing fields requires tree traversal. This adversely affects both ease of access to fields and thus performance if the query is large with many clauses. Nodes and traversal methods will need to be implemented. Difficult to rearrange order of clauses after tree construction.

Option 2: Query Struct

- **Details:** Query Struct with a map to store declarations, values to return, and containers for tuples representing such-that and pattern clauses (one tuple for each clause)
- **Pros:** Completely flattens query structure. All fields can be easily accessed. Clauses can be easily reordered in order to facilitate performance optimization. Standard C++ libraries can be used for implementation.
- **Cons:** Complicates the implementation of the Query Preprocessor as each function that parses a non-terminal need to capture additional context from the calling function. E.g. when parsing `assign a, b` with the rule: declaration : design-entity synonym (‘,’ synonym)* ‘;’. The function parsing declaration needs to extract the design-entity type from the design-entity non-terminal and pass it to the synonym non-terminal parse function. This is in order to have all the information to update the mapping of `a` to an `assign` design-entity.

Criteria: Completeness and correctness, Ease of Access, Extensibility to iterations 2 and 3, Ease of Implementation, Performance

Final Design Decision: Option 2 is chosen.

Both options are complete and correct.

The primary advantage of the Query Tree, that of a natural, accurate and full representation, is not significant compared here to AST for SIMPLE source program, which may have more complex relations that are more difficult to fully and accurately preserve.

On the other hand, disadvantages relative to a flat Query Struct are numerous as expressed above, with performance considerations being especially relevant due to time limitations for query evaluation in later iterations, a constraint not present for SIMPLE program parsing.

Due to speed and ease of query optimization being key considerations in iteration 2 and 3, Option 2 is chosen despite being harder to implement.

3.2.6 Query Evaluator

The central logic of the Query Evaluator is as follows:

1. Initialize an initial list of result candidates based on return value (synonym for iteration 1) indicated in select clause
2. For each clause, make a call or set of API calls (for clauses that include multiple wildcards) to the PKB for the list of entities that satisfy the given clause
3. For each such-that and pattern clause, constrain the list of result candidates to satisfy that clause
    a. For each such-that and pattern clause, a list of entities that satisfy is fetched from the PKB
    b. If the return value appears in a clause, the list of result candidates is replaced by its intersection with the list of entities satisfying that clause
    c. If a clause does not contain the return value, the clause acts as a boolean check. If the returned entity list from the PKB is empty, an empty result is projected. Otherwise, no change is made to the result candidate list

By handling each clause individually, we are able to have intuitive PKB APIs that handle possible clauses efficiently. While performance issues may emerge when number of clauses is large, this current implementation is sufficient for Iteration 1. This implementation is subject to change for future iterations.

In our current implementation, a PKB instance is injected as a dependency into the QueryEvaluator instance. This increases the testability of the QueryEvaluator since a mock PKB can then be injected for unit tests.

The Query is represented as a C++ struct with an unordered map for declarations, a string for the return value, and vectors of tuples for such-that and pattern clauses. The details can be found in 3.2.5.

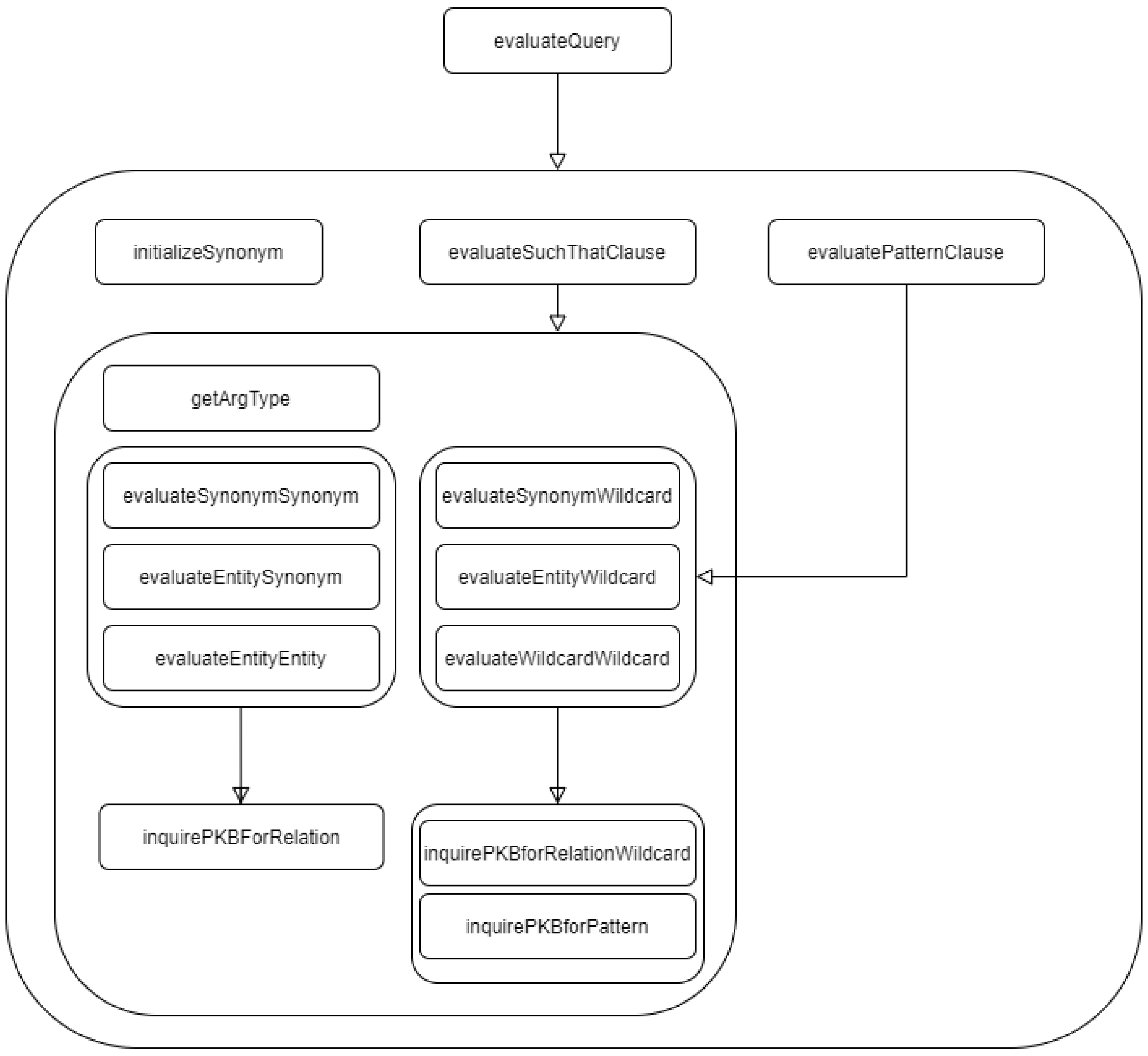


Figure 4. Architectural Diagram Detailing QE Functions Calls

Design Decisions & Justifications

Design Problem: How should the initial candidate list be constrained

Option 1: Evaluate each against each clause to see if it satisfies that clause

- **Pros:** Performs better for large numbers of clauses. At most O(nm) operations on the part of the QueryEvaluator where m is the number of candidates and n is the number of clauses.
- **Cons:** Large number of API calls need to be made to the PKB. May increase complexity in terms of numbers and types that the PKB needs to offer.

Option 2: Find intersection for results satisfying each constraint

- **Pros:** Few (O(n)) API calls need to be made to the PKB. PKB APIs only need to return lists instead of both lists and booleans.
- **Cons:** May face performance issues with a large number of clauses. Requires $O(n^2m)$ comparisons.

Criteria: Performance, Complexity of APIs, Frequence of API calls

Final Design Decision: Option 2. Given that a query has at most one such-that and one pattern clause in iteration 1, we have chosen the option that limits component interactions. For future iterations, Option 1 may be favored if performance becomes a greater issue.

3.3 Component Interactions

3.3.1 Architectural Diagram

An overall architectural diagram (Figure 1, included in section 3.1) was used to represent the different components and subcomponents of the SPA. Allows for division of responsibilities between group members. Helps identify APIs and dependencies between components.

3.3.2 PKB Sequence Diagram

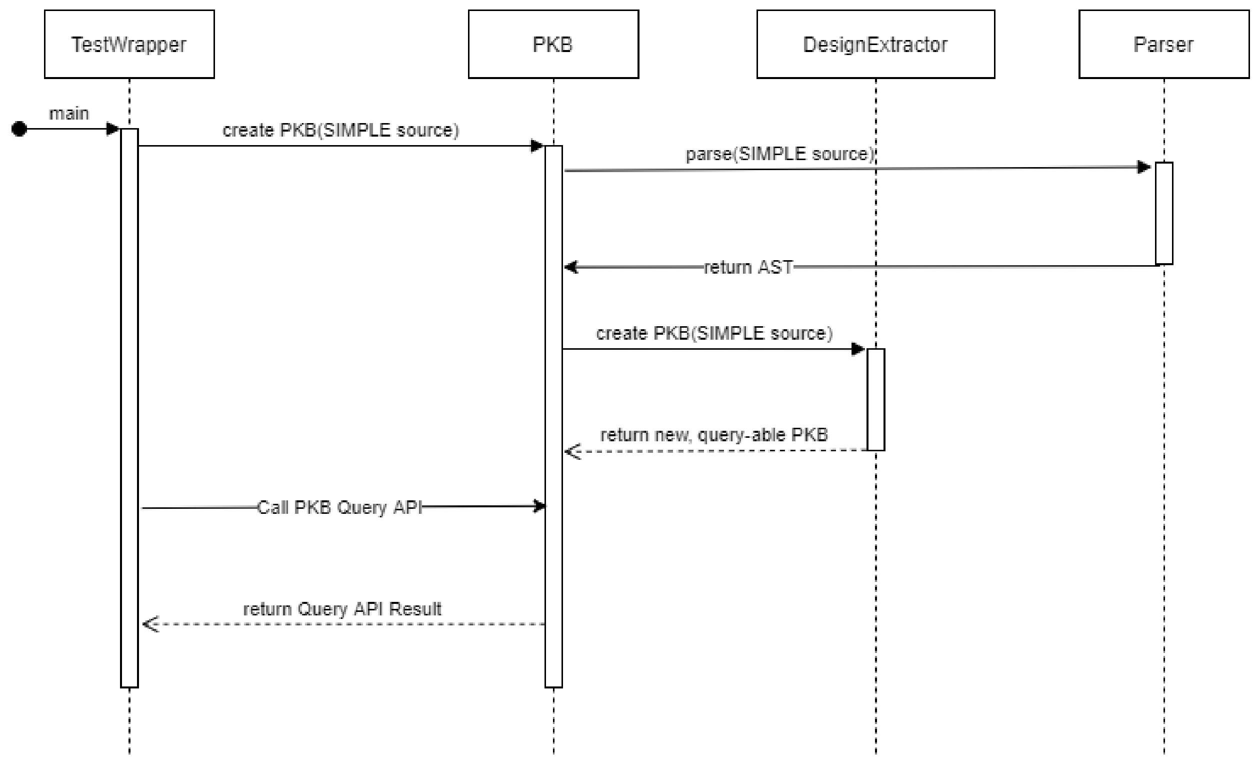


Figure 5. Sequence Diagram for PKB

This diagram helps elucidate the APIs of the SFE and PKB components. Due to the high amount of coupling involved, a clear representation significantly improves code quality and coherence.

3.3.3 QueryEvaluator Sequence Diagram

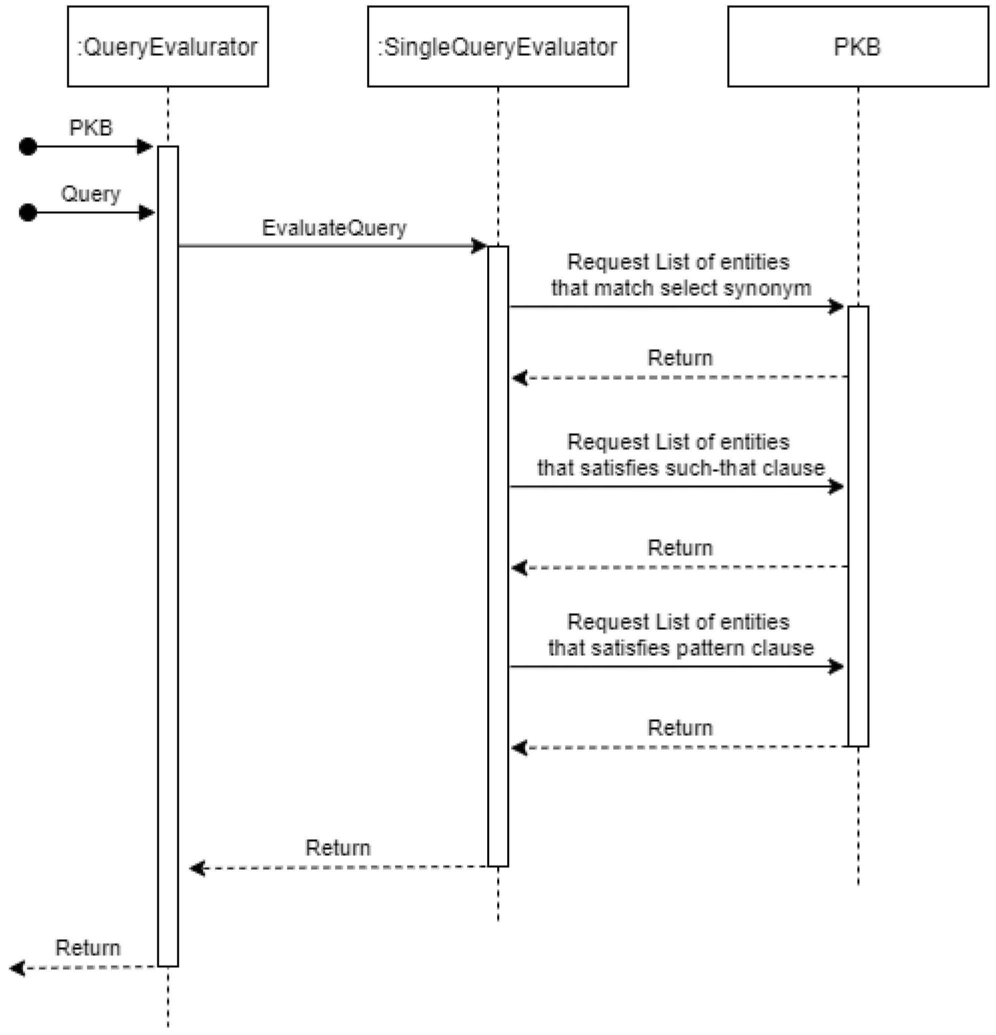

:QueryEvalurator
:SingleQueryEvaluator
PKB
PKB
Query
EvaluateQuery
Request List of entities that match select synonym
Return
Request List of entities that satisfies such-that clause
Return
Request List of entities that satisfies pattern clause
Return
Return
Return

Figure 6. Sequence Diagram for QueryEvaluator

This diagram highlights the interactions between the QueryEvaluator and the PKB for a valid query with both a such-that clause and a pattern clause. The PKB needs to be able to return a list of entities matching a certain constraint for each clause. This is the dominant interaction between the PKB and the SingleQueryEvaluator instance created by the QueryEvaluator to handle a received query. The PKB API is thus primarily a collection of possible constraint formats with a length of a single clause. Some additional utility APIs are also provided. For details see section 7.

4. Documentation and Coding Standards

4.1 Naming Conventions for Abstract APIs

For documenting abstract APIs, our team uses **typedefs** to define the different data types. The **typedefs** are named using capitalized casing with underscores in between words. The following code snippet is taken from our PKB header file, which is the file exposing the abstract APIs to the QE.

```
typedef std::string PROCEDURE;
typedef std::vector<std::string> PROCEDURE_LIST;
typedef std::string VARIABLE;
typedef std::vector<std::string> VARIABLE_LIST;
typedef int STATEMENT_NUMBER;
typedef std::vector<STATEMENT_NUMBER> STATEMENT_NUMBER_LIST;
```

Function names, as according to the LLVM Coding Standards, are verb phrases and named using camel casing (i.e. **getAllStatements()**). Since the PKB class is defined in the PKB header file, functions are also tagged with the keyword **virtual**.

The general structure of documenting every abstract API resembles that of what was taught in class.

Name *Overview:* Rationale & responsibility of the module	
API	
	RETURN_TYPE functionName(**PARAM_TYPE** paramName...); ***Description:*** - describe what the operation does - describe both normal and abnormal behavior

4.2 Coding Standards

For Coding Standards, our team adopts the LLVM Coding Standards[2]. A subsection of rules are included below:

- #includes are introduced in this order:
  - Main Module Header
  - Local/Private Headers
  - LLVM project/subproject headers (clang/..., lldb/..., llvm/..., etc)
  - System #includes
- Header files should be self-contained (compile on their own) and end in .h.
- Source code is fitted to 80 width

- **Type names** (including classes, structs, enums, typedefs, etc) should be nouns and start with an upper-case letter (e.g. TextFileReader).
- **Variable names** should be nouns (as they represent state). The name should be camel case, and start with an uppercase letter (e.g. Leader or Boats).
- **Function names** should be verb phrases (as they represent actions), and command-like functions should be imperative. The name should be camel case, and start with a lowercase letter (e.g. openFile() or isFoo()).
- **Enum declarations** (e.g. enum Foo {...}) are types, so they should follow the naming conventions for types. A common use for enums is as a discriminator for a union, or an indicator of a subclass. When an enum is used for something like this, it should have a Kind suffix (e.g. ValueKind).
- **Enumerators** (e.g. enum { Foo, Bar }) and **public member variables** should start with an upper-case letter, just like types. Unless the enumerators are defined in their own small namespace or inside a class, enumerators should have a prefix corresponding to the enum declaration name. For example, enum ValueKind { ... }; may contain enumerators like VK_Argument, VK_BasicBlock, etc. Enumerators that are just convenience constants are exempt from the requirement for a prefix. For instance:

4.3 Correspondence between Abstract APIs & Concrete APIs

Correspondence between Abstract and Concrete APIs is assured by including abstract APIs in Header files, which are then concretized in corresponding CPP files.

[2] https://llvm.org/docs/CodingStandards.html

5. Testing

5.1 Background

Our team follows a Continuous Integration (CI) software development philosophy scaled to a 20 hour work week instead of a typical 40 hour work week.

In other words, we check in small logical units of code to the remote repository frequently. In the context of our GitHub repository, this means we expect to merge small pull requests (<= 200 LoC)[3] frequently. Each pull request comes from each individual contributor's own branches.

5.2 Automated Testing

As we work in a team on a relatively large project, it would be an inefficient use of everyone's time to have all team members review every pull request in order to keep the team members updated and aware of buggy behaviour due to incomplete implementation.

We want to reduce unnecessary overheads in code reviews and the cognitive load in committing details about incomplete implementation to memory. Automated testing, once established, allows assurance of code quality and correctness while allowing changes to be merged in quickly.

To do this, we set up a CI workflow[4] where tests are run automatically on the pull request whenever we attempt to merge into the master branch. Our master branch is protected, and will only allow pull requests to be merged when all tests pass.

This means that every pull request should come with their own set of tests when they introduce new public methods or functions. The individual contributors have a responsibility for making sure that they do not check in buggy code.

[3] https://github.com/nus-cs3203/team24-cp-spa-20s1/blob/master/README.md
[4] https://github.com/nus-cs3203/team24-cp-spa-20s1/actions

5.3 Choice of Test Types (Unit, Integration, System)

In general, our team prefers testing using system tests. However in the initial stages, no System Tests are written as our SPA is not end to end testable. In fact the only kind of tests we can write are unit tests.

Effectively, we will pick system tests, integration tests, and units tests in this order depending on what components are ready to be tested at the time of being written. We believe that system tests should eventually be written in order to provide proper coverage of all interactions.

If systems tests and unit tests are both provided, we will skip integration testing as systems tests and unit tests would allow us to diagnose our relatively small codebase. We conscientiously tradeoff coverage for development velocity.

There are integration tests for the QueryPreprocessor since it's dependent on the lexer, which has been fully developed before the QueryPreprocessor is written. In this scenario, we skip unit testing and write integration directly for these two components. Additionally as the lexer has unit tests, we have an easier time diagnosing bugs in the test cases between these two components.

5.3.1 Impact of Bug Discovery

> ***Nobody actually creates perfect code the first time around, except me. But there's only one of me. - Linus Torvalds***

Understandably, we are not Linus Torvalds, so we do not write perfect code on the first try. When we discover non-trivial bugs in the master branch, this means that there were specified behaviours that our initial test cases could not account for.

When we have fixed the bugs, we should update existing unit or system tests to account for those specified behaviours. This expedites diagnosing the same bug and mitigates against the same bug resurfacing again in the future.

5.3.2 AutoTester Integration into CI

To integrate the Autotester into our CI workflow, a Python script is written to execute the Autotester and run the System Test cases. It then examines the XML output files for failed test cases. The Python script also handles expected failures in parsing SIMPLE source by examining the exit code returned from the Autotester. This way, we consistently perform regression tests with every available test we have.

5.4 Test Plan

Type of tests	Week 3 (Planning)	Week 4	Week 5	Week 6
Unit	-	To be implemented concurrently with new features	To be implemented concurrently with new features	To be implemented concurrently with new features
Integration	-	-	Integration tests for Lexer + QPP to be added	Integration tests for Lexer + QPP to be added
Regression	-	Embed Unit tests into the Autotester CI script	Embed Unit and Integration tests into the Autotester CI script	Embed Unit, Integration and System tests into the Autotester CI script
System	-	Prepare system test cases	Review system tests with SPA component owners Perform system testing for any end-to-end case that are have been implemented.	Perform extensive system testing involving all features

Unit Testing	Unit tests are written for each public API of each SPA component. Every possible control flow is tested where possible (i.e. parsing an if statement involves a series of function calls to parse the conditions and expressions within the condition itself). Unit tests are included in the same pull request as code implementing support for an API where possible. Usually the member implementing that feature would handle the unit tests for it.
Integration Testing	For Integration Tests, the following interactions can be tested: **Test the construction of the AST** Lexer → Parser **Test the population of design entities/abstractions** Parser → DesignExtractor **Test the storing of information in the PKB** DesignExtractor → PKB **Test the evaluation of queries** PKB → QueryEvaluator **Test the parsing of query to a Query Struct** QueryEvaluator ← QueryPreprocessor **Test the tokenizing of the query** QueryPreprocessor ← Lexer However, as a team, we have decided against prioritizing integration testing. The reason for this decision is backed by our earlier philosophy that pull requests which involve the calling of public methods or functions should come with an extensive set of unit test cases to cover all possible scenarios. This allows us to have a better coverage of the individual SPA components. We also felt that writing the System Tests is sufficient enough to test the various interactions. The exception to this rule are the interactions between QueryPreprocessor and the Lexer. QueryPreprocessor tests are written using the actual Lexer. We have high confidence in the Lexer's correctness due to extensive unit tests. By implementing

	QueryPreprocessor tests as integration tests, we save the effort of mocking another token source that would be little different from the actual lexer and we gain a better sense of how these components perform together. In the case of failed tests, it would also be easy to identify the source of the problem as the components throw different errors.
System Testing	A System Test should consist of a SIMPLE program and a list of queries to be evaluated. The SIMPLE program needs to be comprehensive enough that it covers almost all concrete syntax grammar rules of SIMPLE, meaning that it can consist of statements which range from simple to arbitrarily complicated statements (nested loops, nested expressions, etc). The queries to be evaluated have to also be comprehensive enough to cover all possible PQL grammar rules. Usually the testing in-charge would be involved in drafting out the System Tests, but every member would also be involved in one way or another in testing the AutoTester against the System Tests. For our project, we have written a python script to run the AutoTester against our System Tests. System testing is performed once end-to-end support for a given relation type (Follows, Follows*, Parent, Parent*, Uses, Modifies, Pattern) is implemented.
Acceptance Testing	The client for our project submission is the Professor, who decides whether or not our code has passed the Acceptance Tests. Since it is not possible to liaise with the Professor as he is the one grading our project, the way we ran our Acceptance Tests is that we have a checklist of what is required for Iteration 1 and we check each requirement off our list once we have successfully completed it.
Regression Testing	CI workflows are established in GitHub to ensure that for every pull request to merge the contributor's work into the master branch, the contributor's code is executed automatically against the test cases that we have created earlier. If the build passes, their code is merged into the master branch. This form of regression testing prevents buggy code from being merged in.

User Testing	As there is no clearly defined 'user' that uses the AutoTester, we as a team assume the roles of the 'user' to test the AutoTester. We do this through running the System Tests and evaluating if our code has met the requirements for Iteration 1.
Stress Testing	No stress testing is currently implemented for iteration 1. Future iterations will include systems and unit tests for queries with a large number of clauses

5.5 Examples of test cases of different categories

5.5.1 SPA Frontend Unit Test Case #1

Test Purpose: To check if operator precedence is respected when parsing expressions.

Component Tested: Parser

Test Inputs: Tokenized form of "procedure p{while (x + y * z <= 1){y = y + 1;}}"

Expected Test Result: The * operator node is a child of the + operator node on the created AST.

Found in TestParser.cpp

```
TEST_CASE("Test expr precedence: + before *") {
    Parser parser =
    testhelpers::GenerateParserFromTokens("procedure p{while (x + y * z <= 1){y = y + 1;}}");
    TNode result = parser.parse();


    // We expect our node to look like:
    //          +
    //         /  \
    //        x    *
    //            /  \
    //           y    z

    TNode y(Variable, 1);
    y.name = "y";
    TNode z(Variable, 1);
    z.name = "z";
    TNode lCondLHSMultiply(Multiply);
    lCondLHSMultiply.addChild(y);
    lCondLHSMultiply.addChild(z);

    TNode x(Variable, 1);
    x.name = "x";
    TNode lCondLHSPlus(Plus);
    lCondLHSPlus.addChild(x);
    lCondLHSPlus.addChild(lCondLHSMultiply);
    TNode lCondRHSConst(Constant, 1);
    lCondRHSConst.constant = "1";

    TNode condNode(LesserThanOrEqual, 1);
    condNode.addChild(lCondLHSPlus);
    condNode.addChild(lCondRHSConst);

    REQUIRE(result == testhelpers::generateProgramNodeFromCondition(condNode));
}
```

5.5.2 SPA Frontend Unit Test Case #2

Test Purpose: To check if the getTNodeToStatementNumber function of the DesignExtractor assigns the nodes on the AST to their correct statement numbers. Focus of the test is on nested statements which will be assigned to different layers on the AST.

Component Tested: Design Extractor

Test Inputs: Sample AST

Expected Test Result: The return values of getTNodeToStatementNumber are match against a predetermined list of correct statement numbers

Found in TestDesignExtractor.cpp

```
TEST_CASE("Test getTNodeToStatementNumber maps TNodes to their correct statement numbers") {
    Parser parser = testhelpers::GenerateParserFromTokens(STRUCTURED_STATEMENT);
    TNode ast(parser.parse());
    auto tNodeToStatementNumber = extractor::getTNodeToStatementNumber(ast);
    auto statementNumberToTNode = extractor::getStatementNumberToTNode(tNodeToStatementNumber);

    TNode* statementList = &ast.children[0].children[0];
    REQUIRE(statementList->type == TNodeType::StatementList);

    REQUIRE(tNodeToStatementNumber.size() == 6);

    // While statement
    TNode& whileNodeInAst = statementList->children[0];
    REQUIRE(whileNodeInAst.type == TNodeType::While);
    REQUIRE(statementNumberToTNode[1] == &whileNodeInAst);
    TNode& whileNodeStatements = whileNodeInAst.children[1];
    REQUIRE(whileNodeStatements.type == TNodeType::StatementList);
    REQUIRE(statementNumberToTNode[2] == &whileNodeStatements.children[0]);

    // If-Then-Else statement
    TNode& ifNodeInAst = statementList->children[1];
    REQUIRE(ifNodeInAst.type == TNodeType::IfElse);
    REQUIRE(statementNumberToTNode[3] == &ifNodeInAst);
    // If -> Then
    TNode& ifNodeThenStatements = ifNodeInAst.children[1];
    REQUIRE(ifNodeThenStatements.type == TNodeType::StatementList);
    REQUIRE(statementNumberToTNode[4] == &ifNodeThenStatements.children[0]);
    // If -> Else
    TNode& ifNodeElseStatements = ifNodeInAst.children[2];
    REQUIRE(ifNodeElseStatements.type == TNodeType::StatementList);
    REQUIRE(statementNumberToTNode[5] == &ifNodeElseStatements.children[0]);

    REQUIRE(statementNumberToTNode[6]->type == TNodeType::Assign);
    REQUIRE(statementNumberToTNode[6] == &statementList->children[2]);
}
```

5.5.3 Query Processor Unit Test Case #1

Test Purpose: A test case to see if the correct Entity type is mapped to each string by the entityTypeFromString(std::string) function.

Component Tested: Query

Test Inputs: Strings representing design entity types

Expected Test Result: The return value of the evaluateQuery function is matched against a predetermined set of correct answers

Found in TestQPbackend

```
TEST_CASE("Test entityTypeFromString mappings") {
    REQUIRE(EntityType::STMT == entityTypeFromString("stmt"));
    REQUIRE(EntityType::PRINT == entityTypeFromString("print"));
    REQUIRE(EntityType::WHILE == entityTypeFromString("while"));
    REQUIRE(EntityType::ASSIGN == entityTypeFromString("assign"));
    REQUIRE(EntityType::CONSTANT == entityTypeFromString("constant"));
    REQUIRE(EntityType::READ == entityTypeFromString("read"));
    REQUIRE(EntityType::CALL == entityTypeFromString("call"));
    REQUIRE(EntityType::IF == entityTypeFromString("if"));
    REQUIRE(EntityType::VARIABLE == entityTypeFromString("variable"));
    REQUIRE(EntityType::PROCEDURE == entityTypeFromString("procedure"));
}
```

5.5.4 Query Processor Unit Test Case #2

Test Purpose: A test case to see if the QueryEvaluator is able to correctly handle a clause of the form Modifies(entity, synonym). Other combinations of relations and argument types are also included in other test cases in the file.

Component Tested: QueryEvaluator

Test Inputs: Modifies clauses with different types of entities and synonyms as its arguments

Expected Test Result: The return value of the evaluateQuery function is matched against a predetermined set of correct answers

Found in TestQueryEvaluator.cpp

```
TEST_CASE("Test evaluation of Modifies between entity and synonym") {
    PKBMock pkb(2);
    queryevaluator::QueryEvaluator qe(&pkb);

    Query queryModsStmt = { { { "s", STMT } }, { "s" }, { { MODIFIES, "s", "\"n\"" } }, {} };
    REQUIRE(checkIfVectorOfStringMatch(qe.evaluateQuery(queryModsStmt), { "2", "4", "5" }));
    Query queryModsVar = { { { "v", VARIABLE } }, { "v" }, { { MODIFIES, "2", "v" } }, {} };
    REQUIRE(checkIfVectorOfStringMatch(qe.evaluateQuery(queryModsVar), { "n", "random", "y" }));

    Query queryModpProc = { { { "p", PROCEDURE } }, { "p" }, { { MODIFIES, "p", "\"y\"" } }, {} };
    REQUIRE(checkIfVectorOfStringMatch(qe.evaluateQuery(queryModpProc), { "foo", "bar" }));
    Query queryModpVar = { { { "v", VARIABLE } }, { "v" }, { { MODIFIES, "\"bar\"", "v" } }, {} };
    REQUIRE(checkIfVectorOfStringMatch(qe.evaluateQuery(queryModpVar), { "a", "n", "random", "y" }));
}
```

5.5.5 Query Processor Integration Test Case #1

Test Purpose: Test if Lexer + Query Preprocessor creates the correct Query struct from a single string input for a query with one such-that and one pattern clause

Component Tested: Lexer + Query Preprocessor

Test Inputs: Query string

Expected Test Result: Correctly constructed Query struct

Found in TestLexerQueryPreprocessorParsing.cpp

```
TEST_CASE("Test such-that and pattern clause") {
    std::stringstream queryString = std::stringstream(
    "assign a; Select a such that Follows*(a,a) pattern a (_, _\"x+s+Follows*38\"_)    ");
    qpbackend::Query expectedQuery = qpbackend::Query({ { "a", qpbackend::EntityType::ASSIGN } }, { "a" },
                                                      { { qpbackend::RelationType::FOLLOWST, "a", "a" } },
                                                      { { "a", "_", "_\"x+s+Follows*38\"_" } });

    std::vector<lexer::Token> lexerTokens = backend::lexer::tokenizeWithWhitespace(queryString);
    qpbackend::Query actualQuery = querypreprocessor::parseTokens(lexerTokens);

    REQUIRE(expectedQuery == actualQuery);
}
```

5.5.6 Systems Test #1 (TEST-NESTED-COND)

Test Purpose: To test the parsing of nested while and if conditional statements, whitespace and braces concatenation, as well as variable names. Since there are many while and if statements for this System Test, we also test for the ability to parse different categories of conditional expressions. For formulating the type of queries to test, we would be testing the Follows/Follows* and Parent/Parent* relationships more comprehensively than Uses/Modifies (which is tested comprehensively in Systems Test #2).

Test Inputs: <name_of_source_file>.txt <name_of_queries_file>.txt <name_of_output_file>.xml

Expected Test Result: When running the Autotester with the required test inputs, our program should pass all the test cases.

This is the SIMPLE program source code in TEST-NESTED-COND_source.txt

```
procedure main {
  H2O = 0;
  naMeWiThDiGiTS85 = 3;
  call Dipsy; call PO;}

procedure Dipsy {
    dummy = dummy + 1;
    x = 5;
    if (1 != 1) then {while (x != 0) {
            if (dummy > 0) then {
                x = x + dummy;
                call lala;
                print x;
                dummy = dummy + x;
            } else {
                x = x - 1;
                print x;
            }
        }
        read a;
    } else {
        print dummy;
    }
}

procedure lala {
    read special; read mango; read tree;
    print special;
    print
    mango;
    print                   tree;
}

procedure PO {
    count = 11;
    while ((!(count - dummy <= 5)) && (count != 0)) {
        if (count >= 6) then {
            while (!((naMeWiThDiGiTS85 > 0) || (count < 0))) {
                naMeWiThDiGiTS85 = naMeWiThDiGiTS85 - 1;
                print naMeWiThDiGiTS85;
            }
            count = count - 1;
        } else {
            H2O = H2O + 1;
            print naMeWiThDiGiTS85;
            while (H2O != 0) {
                a = a + 1;
                print a;
                H2O = H2O - 1;
                print H2O;
            }}}}
```

These are some of the queries that will be tested against TEST-NESTED-COND_source.txt

```
1 - (Invalid Query) Parent(s, s)
stmt s;
Select s such that Parent(s, s)
none
5000
2 - (Valid Query) Select statements that are children of #3
stmt s;
Select s such that Parent(3, s)
none
5000
3 - (Valid Query) Select statements that are parent of #3
stmt s;
Select s such that Parent(s, 3)
none
5000
4 - (Valid Query) Select statements that are children of #7
stmt s;
Select s such that Parent(7, s)
8, 16, 17
5000
5 - (Valid Query) Select statements that are children of #9
stmt s;
Select s such that Parent(9, s)
10, 11, 12, 13, 14, 15
5000
6 - (Valid Query) Select statements in which they are a parent of some other statement
stmt s;
Select s such that Parent(s, _)
7, 8, 9, 25, 26, 27, 33
5000
7 - (Valid Query) Select statements such that Parent(_, _) is true
```

```
stmt s;
Select s such that Parent(_, _)
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37
5000
8 - (Invalid Query) Invalid argument type in clause
variable v;
Select v such that Parent("s", v)
none
5000
.
.
.
.
32 - follows* for wildcard
procedure p;
Select p such that Follows* (_, _)
main, Dipsy, lala, PO
5000
33 - follows with pattern
assign a; while w;
Select a such that Follows(w, a) pattern a(_, _"1"_)
30
5000
34 - follows* with pattern
assign a; if ifs; variable v;
Select v such that Follows*(a, ifs) pattern a(v, _"1"_)
dummy
5000
35 - follows* with pattern
assign a; call cl; variable v;
Select v such that Follows*(a, cl) pattern a(v, _)
H2O, naMeWiThDiGiTS85, x
5000
36 - follows of wildcard
print pt; call cl; assign a
Select pt such that Follows(cl, a)
none
5000
.
.
.
.
47 - (Invalid Query) Such that with pattern
assign a; while w;
Select w such that pattern a(_,_) Parent*(w, a)
none
5000
48 - (Valid Query) Select statements such that it use some variable and it has a pattern
```

```
assign a; variable v;
Select a such that Uses(a, v) pattern a(_, _"1"_)
5, 14, 28, 30, 31, 34, 36
5000
49 - (Valid Query) Select statements such that it modify some variable and it has a pattern
assign a; variable v;
Select a such that Modifies(a, v) pattern a("x", _)
6, 10, 14
5000
50 - (Valid Query) Select statements such that it use some variable
stmt s; variable v;
Select s such that Uses(s, _)
3, 4, 5, 8, 9, 10, 11, 12, 13, 14, 15, 17, 21, 22, 23, 25, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37
5000
51 - (Valid Query) Select statements such that some assignment statement modifies a variable
v and v is on LHS.
assign a; variable v; stmt s;
Select s such that Modifies(a, v) pattern a(v, _)
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37
5000
52 - (Invalid Query) Cannot have synonym on RHS argument of pattern
assign a; variable v1, v2; stmt s;
Select s such that Modifies(a, v1) pattern a(_, v2)
none
5000
53 - (Valid Query) Select assign statements with some pattern
assign a1, a2, a3;
Select a2 pattern a1("H2O", _)
1, 2, 5, 6, 10, 13, 14, 24, 28, 30, 31, 34, 36
5000
54 - (Valid Query) Select assign statements with some pattern
assign a;
Select a pattern a(_, _)
1, 2, 5, 6, 10, 13, 14, 24, 28, 30, 31, 34, 36
5000
```

5.5.7 Systems Test #2 (TEST-OPERATORS-N-PATTERNS)

Test Purpose: To test our program on its ability to recognise patterns and parse the SIMPLE program properly when assignment statements have multiple different operators and order of precedence. For this purpose, the SIMPLE source code that we have come up with is of a relatively simpler structure but containing more assignment statements. The queries to be tested are more toward Uses, Modifies and patterns.

Test Inputs: <name_of_source_file>.txt <name_of_queries_file>.txt <name_of_output_file>.xml

Expected Test Result: When running the Autotester with the required test inputs, our program should pass all the test cases.

```
procedure main {
  coco = 5;
  call Second;
}
procedure First {
    dummy = dummy + 1;
    N = 17;
    F = N % dummy + (coco - dummy) * (1);
    print F;
}

procedure Second {
    count = 10;
    call First;
    while ((x > 0) && (count != 0)) {
        read k;
        count = count - 1;
        a = (a - 1) * 5 / (2 * F);
        y = (F / 2 + (a + b) * coco) * 1;
        x = y * y - b * y * (y + b);
        if (count % 2 == 0) then {
            b = b + 1 / 2;
            d = c + 1;
        } else {
            e = f + 2;
            print a;
        }
        print x;
    }
}
```

These are some of the queries that will be tested against TEST-OPERATORS-N-PATTERNS_source.txt

```
.
.
11 - (Valid Query) Select statements which use the variable count
stmt s;
Select s such that Uses(s, "count")
2, 9, 11, 15
5000
12 - (Valid Query) Select variable names that are used in statement #12
variable v;
Select v such that Uses(12, v)
F, a
5000
13 - (Valid Query) Select statements which use the variable dummy
stmt s;
Select s such that Uses(s, "dummy")
2, 3, 5, 8
5000
14 - (Valid Query) Select procedures which use the variable dummy
procedure p;
Select p such that Uses(p, "dummy")
main, Second, First
5000
15 - (Valid Query) Select while statements that use the variable count
while w;
Select w such that Uses(w, "count")
9
5000
16 - (Valid Query) Select variables that are used by the cond statement at #9
variable v;
Select v such that Uses(9, v)
x, count, a, F, b, coco, y, c, f
5000
17 - (Valid Query) Select if statements that use the variable count
if ifs;
Select ifs such that Uses(ifs, "count")
15
5000
18 - (Valid Query) Select call statements that use the variable F
call c;
Select c such that Uses(c, "F")
2, 8
5000
19 - (Valid Query) Select print statements that use the variable F
print pn;
Select pn such that Uses(pn, "F")
6
5000
20 - (Valid Query) Select variables that are used by any print statements
print pn; variable v;
Select v such that Uses(pn, v)
F, a, x
5000
21 - (Valid Query) Select assignment statements that use the variable b
assign a;
Select a such that Uses(a, "b")
13, 14, 16
```

```
5000
22 - (Valid Query) Select assignment statements that use some variable
assign a; variable v;
Select a such that Uses(a, v)
3, 5, 11, 12, 13, 14, 16, 17, 18
5000
23 - (Valid Query) Select variables which are used by some assignment statements
assign a; variable v;
Select v such that Uses(a, v)
dummy, N, coco, count, a, F, b, y, c, f
5000
24 - (Valid Query) Select statements s such that Uses(main, coco) is true
stmt s;
Select s such that Uses("main", "coco")
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
5000
25 - (Valid Query) Select statements such that Uses(s, _) is true
stmt s;
Select s such that Uses(s, _)
2, 3, 4, 5, 6, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
5000
26 - (Valid Query) Select procedures such that Uses(p, _) is true
procedure p;
Select p such that Uses(p, _)
main, First, Second
5000
27 - (Invalid Query) Select statements such that Uses(_, _) is true
stmt s;
Select s such that Uses(_, _)
none
5000
.
.
.
38 - (Valid Query) Select variables which are modified by statement #12
variable v;
Select v such that Modifies(12, v)
a
5000
39 - (Invalid Query) Select variables which are modified by a print statement
variable v; print pn;
Select v such that Modifies(pn, v)
none
5000
40 - (Invalid Query) Inaccurate argument type
variable v; stmt s;
Select s such that Modifies(v, s)
none
5000
41 - (Invalid Query) Inaccurate argument type
assign a; stmt s;
Select s such that Modifies(s, a)
none
5000
42 - (Valid Query) Select procedures which modify variable x
procedure p;
Select p such that Modifies(p, "x")
main, Second
5000
43 - (Valid Query) Select variables that are modified by an if statement
```

```
if ifs; variable v;
Select v such that Modifies(ifs, v)
b, d, e
5000
44 - (Valid Query) Select variables that are modified by a while statement
while w; variable v;
Select v such that Modifies(w, v)
k, count, a, y, x, b, d, e
5000
45 - (Valid Query) Select variables that are modified by a while statement
while w; variable v;
Select v such that Modifies(w, v)
k, count, a, y, x, b, d, e
5000
46 - (Valid Query) Select variables that are modified by a read statement
read r; variable v;
Select v such that Modifies(r, v)
k
5000
47 - (Valid Query) Select variables that are modified by statement #10
variable v;
Select v such that Modifies(10, v)
k
5000
48 - (Invalid Query) Integer in second argument of Modifies
stmt s;
Select s such that Modifies(s, 10)
none
5000
49 - (Invalid Query) Modifies(_, v) wildcard in first argument
variable v;
Select v such that Modifies(_, 10)
none
5000
50 - (Invalid Query) Modifies(_, _) wildcards
stmt s;
Select s such that Modifies(_, _)
none
5000
51 - (Valid Query) ModifiesP('main', _) entRef in first argument
stmt s;
Select s such that Modifies("main", _)
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
5000
52 - (Invalid Query) ModifiesS('unknownProcedure', _) entRef in first argument
stmt s;
Select s such that Modifies("unknownProcedure", _)
none
5000
53 - (Valid Query) Select variables by some call statement
call c; variable v;
Select v such that Modifies(c, v)
count, x, k, y, F, N, dummy, b, d, e, a
5000
.
.
63 - subexpression match pattern
assign a;
Select a pattern a(_, _"y * (y + b)"_)
none
```

```
5000
64 - subexpression match pattern
assign a;
Select a pattern a(_, _"b * y * (y + b)"_)
14
5000
65 - subexpression match pattern
assign a;
Select a pattern a(_, _"y * y - b"_)
none
5000
66 - subexpression match pattern
assign newa;
Select newa pattern newa(_, _"a+b"_)
13
5000
67 - subexpression match pattern
assign a;
Select a pattern a(_, _"(a+b)"_)
13
5000
68 - subexpression match pattern
assign a;
Select a pattern a("b", _"1"_)
16
5000
69 - subexpression match pattern
assign a; variable v;
Select a pattern a(v, _"1"_)
3, 5, 11, 12, 13, 16, 17
5000
70 - wildcard match pattern
assign a; variable v;
Select v pattern a(v, _)
coco, dummy, N, F, count, a, y, x, b, d, e
5000
71 - wildcard match pattern
assign a;
Select a pattern a("count", _)
7, 11
5000
.
.
.
78 - pattern and Uses
assign a; variable v;
Select a pattern a(v, _) such that Uses(a, v)
3, 11, 12, 16
5000
79 - pattern and Uses
assign a; variable v;
Select v such that Uses("Second", v) pattern a(v, _"1"_)
count, a, y, b, dummy, F
5000
80 - pattern and Uses
assign a; stmt s; variable v;
Select v such that Uses(s, v) pattern a(v, _"2"_)
a, y, b
5000
81 - pattern and Uses
```

```
assign a; stmt s; variable v;
Select s such that Uses(s, v) pattern a(v, _"2"_)
2, 9, 12, 13, 14, 15, 16, 19
5000
82 - pattern and Modifies
assign a; variable v;
Select a such that Modifies(a, v) pattern a(v, _)
1, 3, 4, 5, 7, 11, 12, 13, 14, 16, 17, 18
5000
83 - pattern and Modifies
assign a; variable v; stmt s;
Select s such that Modifies(s, v) pattern a(v, _"1"_)
2, 3, 5, 7, 8, 9, 11, 12, 13, 15, 16, 17
5000
84 - pattern and Modifies
assign a; variable v;
Select v pattern a(v, _"1"_) such that Modifies("First", v)
dummy, F
5000
```

6. Discussion

Through the course of Iteration 1, we have drawn the following lessons:

1. The setting up of effective communications channels that double for record keeping is helpful in managing a project with many moving parts.
2. Breaking up of iteration 1 into projects and issues on Github has made the project more manageable. By matching these projects and issues to our development plan, we are able to keep track of progress and adjust plans accordingly
3. There were numerous instances of components blocking one another. A willingness to inform the rest of the team and quick turnaround times on communication help keep things running smoothly. Expectations on communication established in the initial weeks has helped the team communicate more smoothly in subsequent weeks.
4. The impact of early testing that involves the testing of different components. It is important to continuously spend effort in testing so that as new features are being implemented, they don't break existing features.
5. Having teamwork with one another helps build rapport for the entire team. Without teamwork, we would not be able to finish on time. Being willing to ask for help when we are met with difficulties during implementation requires great courage, but it improves communication.
6. It is important to set out the requirements and get all questions regarding requirements addressed as soon as possible. For example, there were some grammar rules that we only got to understand mid-week. If we had clarified earlier, we would have been able to finish building the feature, rather than building something then having to make changes to accommodate what is required for the actual submissions.

7. Documentation of abstract APIs

The abstract APIs listed below are APIs exposed by the PKB component to the QE component. As our current implementation of parsing the SIMPLE program is through an AST, the abstract APIs listed are also part of the AST APIs.

7.1 Statement, Variable, Procedure, Constants List

Statement List *Overview:* Retrieves all statements in the SIMPLE program.	
API	
	STATEMENT_NUMBER_LIST getAllStatements(); ***Description:*** Returns the list of STATEMENT_NUMBERs if list is not empty. Otherwise, return an empty list. A STATEMENT_NUMBER corresponds to the statement number of statements in the SIMPLE program.

Variable List *Overview:* Retrieves all variables used in the SIMPLE program.	
API	
	VARIABLE_NAME_LIST getAllVariables(); ***Description:*** Returns the list of VARIABLE_NAMEs if list is not empty. Otherwise, return an empty list. A VARIABLE_NAME corresponds to the name of a variable used in the SIMPLE program.

Procedure List *Overview:* Retrieves all procedures defined in the SIMPLE program.	
API	
	PROCEDURE_NAME_LIST getAllProcedures(); ***Description:*** Returns the list of PROCEDURE_NAMEs if list is not empty. Otherwise, return an empty list. A PROCEDURE_NAME corresponds to the name of a procedure defined in the SIMPLE program.

Constants List *Overview:* Retrieves all constants defined in the SIMPLE program.	
API	
	PROCEDURE_NAME_LIST getAllConstants(); ***Description:*** Returns the list of CONSTANTS if list is not empty. Otherwise, return an empty list.

7.2 Statement Types

Statement types *Overview:* Checks if a statement is of a certain statement type	
API	
	BOOLEAN isRead(**STATEMENT_NUMBER** s); ***Description:*** Checks if statement s is a Read statement
	BOOLEAN isPrint(**STATEMENT_NUMBER** s); ***Description:*** Checks if statement s is a Print statement
	BOOLEAN isCall(**STATEMENT_NUMBER** s); ***Description:*** Checks if statement s is a Call statement
	BOOLEAN isWhile(**STATEMENT_NUMBER** s); ***Description:*** Checks if statement s is a While statement
	BOOLEAN isIfElse(**STATEMENT_NUMBER** s); ***Description:*** Checks if statement s is an If-Else statement
	BOOLEAN isAssign(**STATEMENT_NUMBER** s); ***Description:*** Checks if statement s is an Assignment statement

7.3 Follows, Follows*

Follows, Follows* *Overview:* Retrieves all the statements that appear after the statement at statementNumber, in the same "level" of the AST, in order.	
API	
	STATEMENT_NUMBER_LIST getDirectFollow(**STATEMENT_NUMBER** s); ***Description:*** To get the first statement before s. first_statement_before_statement_9 = getDirectFollow(9)[0], if it exists.
	STATEMENT_NUMBER_LIST getDirectFollowedBy(**STATEMENT_NUMBER** s); ***Description***: To get the first statement number after s. first_statement_after_statement_9 = getStatementsFollowedBy(9)[0], if it exists.
	STATEMENT_NUMBER_LIST getStatementsFollowedBy(**STATEMENT_NUMBER** s); ***Description:*** Returns a list of STATEMENT_NUMBERs which represents statements that follow after a given STATEMENT_NUMBER s. If there are no statements appearing after s, return an empty list.
	STATEMENT_NUMBER_LIST getAllStatementsThatAreFollowed(); ***Description:*** Returns a list of STATEMENT_NUMBERs which represents statements that are followed by some other statement. If there are no statements, return an empty list.
	STATEMENT_NUMBER_LIST getAllStatementsThatFollows(**STATEMENT_NUMBER** s); ***Description:*** Returns a list of STATEMENT_NUMBERs which represents statements that appear before a given STATEMENT_NUMBER s. If there are no statements appearing before s, return an empty list.
	STATEMENT_NUMBER_LIST getAllStatementsThatFollows(); ***Description:*** Returns a list of STATEMENT_NUMBERs which represents statements that follow some statement. If there are no statements, return an empty list.

7.4 Parent, Parent*

Parent, Parent* *Overview:* Retrieves all the statements that have the statement at statementNumber as a direct or indirect descendant. A list of statements is returned such that for statement s, Parent*(s, <statement at statementNumber>) holds true.	
API	
	STATEMENT_NUMBER_LIST getParent(**STATEMENT_NUMBER** statementNumber); **Description**: To get the direct parent of statementNumber. parent_of_9 = getParent(9)[0], if it exists.
	STATEMENT_NUMBER_LIST getChildren(**STATEMENT_NUMBER** statementNumber); **Description**: To get the direct children of statementNumber. children_of_9 = getChildren(9).
	STATEMENT_NUMBER_LIST getAncestors(**STATEMENT_NUMBER** statementNumber); ***Description***: Returns a list of STATEMENT_NUMBERs which represents statements that are ancestors of a given statementNumber. If there are no ancestors, return an empty list.
	STATEMENT_NUMBER_LIST getStatementsThatHaveAncestors(); ***Description:*** Returns a list of STATEMENT_NUMBERs which represents all statements that have ancestors. If there are no statements, return an empty list.
	STATEMENT_NUMBER_LIST getDescendants(**STATEMENT_NUMBER** statementNumber, bool isDirect); ***Description:*** Returns a list of STATEMENT_NUMBERs which represents statements that are descendants of the statement from the given STATEMENT_NUMBER statementNumber. If there are no descendants, return an empty list.
	STATEMENT_NUMBER_LIST getStatementsThatHaveDescendants(); ***Description:*** Returns a list of STATEMENT_NUMBERs which represents statements that have descendants. If there are no statements, return an empty list.

7.5 Uses

Uses *Overview:* Retrieves a list of statements or procedures that uses some variable or retrieves a list of variables that are used by some procedure or statement.	
API	
	STATEMENT_NUMBER_LIST getStatementsThatUse(**VARIABLE_NAME** v); ***Description:*** Returns a list of STATEMENT_NUMBERs which represent statements that use some variable v. If there are no statements that use that variable, return an empty list.
	STATEMENT_NUMBER_LIST getStatementsThatUseSomeVariable(); ***Description:*** Returns a list of STATEMENT_NUMBERs which represent statements that use some variable. If there are no statements, return an empty list.
	PROCEDURE_NAME_LIST getProceduresThatUse(**VARIABLE_NAME** v); ***Description:*** Returns a list of PROCEDURE_NAMEs which represent procedures that use some variable v. If there are no procedures that use that variable, return an empty list.
	PROCEDURE_NAME_LIST getProceduresThatUseSomeVariable(); ***Description:*** Returns a list of PROCEDURE_NAMEs which represent procedures that use some variable. If there are no procedures, return an empty list.
	VARIABLE_NAME_LIST getVariablesUsedIn(**PROCEDURE_NAME** p); ***Description:*** Returns a list of VARIABLE_NAMEs which represent variables that are used in a procedure p. If there are no variables used in that procedure, return an empty list.
	VARIABLE_NAME_LIST getVariablesUsedBySomeProcedure(); ***Description:*** Returns a list of VARIABLE_NAMEs which represent variables that are used by some procedure. If there are no variables, return an empty list.
	VARIABLE_NAME_LIST getVariablesUsedIn(**STATEMENT_NUMBER** s); ***Description:*** Returns a list of VARIABLE_NAMEs which represent variables that are used by a statement s. If there are no variables used by that statement, return an empty list.
	VARIABLE_NAME_LIST getVariablesUsedBySomeStatement(); ***Description:*** Returns a list of VARIABLE_NAMEs which represent variables that are used by some statement. If there are no variables, return an empty list.

7.6 Modifies

Modifies *Overview:* Retrieves a list of statements or procedures that modifies some variable or retrieves a list of variables that are modified by some procedure or statement.	
API	
	STATEMENT_NUMBER_LIST getStatementsThatModify(**VARIABLE_NAME** v); ***Description:*** Returns a list of STATEMENT_NUMBERs which represent statements that modify some variable v. If there are no statements that modify that variable, return an empty list.
	STATEMENT_NUMBER_LIST getStatementsThatModifySomeVariable(); ***Description:*** Returns a list of STATEMENT_NUMBERs which represent statements that modify some variable. If there are no statements, return an empty list.
	PROCEDURE_NAME_LIST getProceduresThatModify(**VARIABLE_NAME** v); ***Description:*** Returns a list of PROCEDURE_NAMEs which represent procedures that modify some variable v. If there are no procedures that modify that variable, return an empty list.
	PROCEDURE_NAME_LIST getProceduresThatModifySomeVariable(); ***Description:*** Returns a list of PROCEDURE_NAMEs which represent procedures that modify some variable. If there are no procedures, return an empty list.
	VARIABLE_NAME_LIST getVariablesModifiedBy(**PROCEDURE_NAME** p); ***Description:*** Returns a list of VARIABLE_NAMEs which represent variables that are modified by a procedure p. If there are no variables modified by that procedure, return an empty list.
	VARIABLE_NAME_LIST getVariablesModifiedBySomeProcedure(); ***Description:*** Returns a list of VARIABLE_NAMEs which represent variables that are modified by some procedure. If there are no variables, return an empty list.
	VARIABLE_NAME_LIST getVariablesModifiedBy(**STATEMENT_NUMBER** s); ***Description:*** Returns a list of VARIABLE_NAMEs which represent variables that are modified by a statement s. If there are no variables modified by that statement, return an empty list.
	VARIABLE_NAME_LIST getVariablesModifiedBySomeStatement(); ***Description:*** Returns a list of VARIABLE_NAMEs which represent variables that are modified by some statement. If there are no variables, return an empty list.

7.7 Pattern

Pattern
Overview:
Get all statements that match the input pattern.
Example:

```
pattern a(_, "_1+1_") -> getAllAssignmentStatementsThatMatch("", "1+1", true);
pattern a("x", "_") -> getAllAssignmentStatementsThatMatch("x", "", true);
pattern a("x", "1+1") -> getAllAssignmentStatementsThatMatch("x", "1+1", false)
```

API

STATEMENT_NUMBER_LIST getAllAssignmentStatementsThatMatch(**STRING** assignee, **STRING** pattern, **BOOLEAN** isSubExpr);
Description: Returns STATEMENT_NUMBER_LIST which represent assignment statements that match a given pattern. If an invalid (e.g. "1++++") pattern is provided, an empty STATEMENT_NUMBER_LIST will be returned.

~ End of report ~