CS3203 Software Engineering Project

AY20/21 Semester 2

Iteration 3 Project Report

Team 19

Team Members	Matriculation No.	Email
Ang Lu Shing	A0187364U	e0322948@u.nus.edu
Cheyanne Sim	A0191171L	e0335668@u.nus.edu
Lee Li Ying Irene	A0187732W	e0323316@u.nus.edu
Ng Zi Xin	A0189083U	e0324667@u.nus.edu
Ong Yin Ming Jonas	A0155237E	e0031374@u.nus.edu
Voong Yu Xuan	A0185284W	e0318575@u.nus.edu

Consultation Hours: Wednesday 11am - 12nn

Tutor(s): Leong Zhan Hao, Nicholas

Table of Contents

1 Scope

At the end of Iteration 3, the features implemented for our SPA are as stated in the table below. We have also implemented query optimization to evaluate queries efficiently.

Iterations	SIMPLE	Design abstractions stored	Clauses	Design entities
1	● Basic SIMPLE syntax with no calls and single procedure	● Follows/Follows* ● Parent/Parent* ● Modifies ● Uses	● Such that ● Assign pattern	● Statement ● Procedure ● Variable ● Constant ● Read ● Print ● Assign ● If ● While
2	● Basic SIMPLE syntax with calls and multiple-procedures	● Calls/Calls* ● Next/Next*	● With ● While Pattern ● If Pattern	● Call ● Program line ● Tuple ● BOOLEAN ● Attributes
3		● Affects/Affects* ● NextBip/NextBip* ● AffectsBip/AffectsBip*		

Our SPA is able to do the following:

Source Processor (SP):

1. Validate SIMPLE source code according to full grammar rules
2. Perform checks of cyclical calls
3. Form the abstract syntax tree (AST) to extract design entities
4. Extract all design abstractions in Design Extractor in Full SPA requirements

Program Knowledge Base (PKB):

1. Store and retrieve all design entities as per Full SPA requirements
2. Store and retrieve all design abstractions as per Full SPA requirements

Query Processor (QP):

1. Validate syntax and semantics of queries
2. Evaluate queries which must consist of single Select clause and any number of 'and' combinations of the following types of clauses:
   a. Such that clauses
   b. Assign, while, if pattern clauses

      c.   With clauses

3. Single Select clause can be one of the following:

      a.   Synonym of design entities

      b.   BOOLEAN

      c.   Tuple

      d.   Attributes

4. Optimize query evaluation

2 Development Plan

This section describes the development plan we undertook during Iterations 1, 2 and 3.

We adopted the iterative software development process model to develop our SPA incrementally through repeated development cycles. We are able to constantly review our process, gather feedback and improve for the next cycle.

Also, we used the breadth-first iterative approach where we worked on all the components in parallel and focused on completing some of the functionalities at a time. We adopted this because we wanted to be able to perform integration and system testing at each iteration and to meet the requirements of the project. Furthermore, we can also find bugs and problems related to our design early and make the relevant changes early. Each week, we aim to deliver the features described in the plan. Every week, we implement parts of each component such that the entire SPA is iteratively built up and tested.

2.1 Project Plan

Weeks 2 - 3	
After delegating the work, we have to research and brainstorm on the design of each component. We also have to set up the development environment and various tools.	
All	Set up development environment and tools
Lu Shing, Jonas	Design Program Parser APIs for basic SPA functionality
Yu Xuan	Design DE APIs for basic SPA functionality
Yu Xuan, Cheyanne	Design PKB APIs for basic SPA functionality
Irene, Zi Xin	Design QP and QE APIs for basic SPA functionality
All	Set up testing frameworks, scripts etc., and begin writing system tests

Weeks 4 - 6

We start implementing the components and writing unit and integration tests for those that we have implemented as we implement them. We aim to complete all these by week 6 so that we are ready for the Iteration 1 submission.

All	Finish the report
Lu Shing, Jonas	Implement the Parser for basic SPA functionality
Yu Xuan	Implement extraction for all design entities in basic SPA
Cheyanne	Implement storage for all design entities and design abstractions in basic SPA
Irene, Zi Xin	Implement QP and QE for basic SPA functionality
All	Add unit tests and system tests for functionalities implemented, and integration tests when components are completed

Weeks 7 - 8

After fixing bugs identified in Iteration 1, we plan our Iteration 2 activities and start designing and implementing advanced SPA features.

All	Come up with a development plan for Iteration 2 and 3
All	Improve report based on Iteration 1 feedback and update the Scope, Development Plan, Testing, Coding & Documentation Standards, API Design sections in the report
Lu Shing, Jonas	Fix bugs found in Iteration 1
Lu Shing, Jonas, Yu Xuan	Design and implement the parsing, validation and processing of call statements & multiple procedures
Yu Xuan, Cheyanne	Design and implement the population of full Modifies/Uses relationships, Call/Call* and container pattern information
Yu Xuan, Cheyanne	Design the population of Next/Next* and Affects/Affects* relationships
Irene, Zi Xin	Implement QP and QE for advanced SPA functionality

Weeks 9 - 10

We aim to continue implementing Advanced SPA features and finish the report for Iteration 2. Components that have finished will start to focus heavily on system testing.

All	Complete report for Iteration 2 submission
Lu Shing, Jonas, Yu Xuan	Help out with test cases and add test cases that are found missing in the demo.
Yu Xuan, Cheyanne	Implement the population of Next/Next* and Affects/Affects* relationships and with-clause information
Irene, Zi Xin	Design and implement the evaluation of queries with Affects/Affects* relationships, with-clauses and synonym attributes
Irene, Zi Xin	Evaluation of queries with optimization

Weeks 11 - 12

We aim to work on query optimization and the implementation for the extension, with unit and integration tests for the features implemented. We will also continue to write system tests for our SPA system and update the report for Iteration 3 submission.

All	Update report for Iteration 3
Lu Shing, Jonas	Write system tests to ensure correctness of Advanced SPA and extension features
Yu Xuan, Cheyanne	Population of NextBip / NextBip* / AffectsBip / AffectsBip* relationship extension
Irene, Zi Xin	Parsing and evaluation of queries, focusing on NextBip / NextBip* / AffectsBip / AffectsBip* relationship

Week 13

We aim to complete system testing and the report for Iteration 3, as well as prepare for the final presentation.

All	Complete report for Iteration 3 submission
All	Complete system testing
All	Conduct presentation rehearsal after code + report submission

2.2 Iteration 3 Plan

Week 2
We were in the Initiation and Definition phase where we made plans on how we can develop SPA. We also set up the development environment and research to come out with a development plan for Iteration 1.

All	Set up tools
All	Decide on coding standards
Lu Shing	Research on Lexer algorithm
Jonas	Research on basic parsing algorithm
Yu Xuan, Cheyanne, Irene, Zi Xin	Brainstorm on how to store information for their respective components
All	Set up testing frameworks, scripts etc.

Week 3
We were in the Planning or Design phase where we design our APIs and tests to be implemented for the iteration. After so, we can start on the Implementation phase and begin building our SPA.

All	Decide on API documentation standards and correspondence between abstract API types with C++ classes
All	Fill up Scope, Development Plan, Coding & Documentation Standards, API Design sections in the report
All	Begin implementation of some of APIs designed
Lu Shing	Design SP internal APIs (Tokens and TokenType enums)
Jonas	Design AST APIs
Yu Xuan	Design DE APIs
Cheyanne	Design PKB APIs
Irene, Zi Xin	Design QP APIs
All	Come up with test cases according to the features to be implemented in the plan. Test cases generation should be ahead of the schedule by one week (e.g. System test cases for features aimed to be completed in Week 4 must already be generated by end of Week 3)

Week 4	
We build some parts of the different components.	
All	Fill up Testing section in the report
Lu Shing	Implement Token classes, syntactic checks for Lexical Tokens, full Lexer functionality
Jonas	Implement the Parser to parse read, print, assign Implement basic AST structure
Yu Xuan	Implement extraction of design entities
Cheyanne	Implement storage for design entities and design abstraction. Conduct unit tests for each design entity and design abstraction after implementing them.
Irene	Implement parsing of pattern clauses and evaluation of Parent/Parent* and Modifies
Zi Xin	Implement query utilities, parsing of such that clauses and evaluation of queries Follows/Follows* and Uses
All	Implement unit tests for functionalities implemented

Week 5	
We aim to complete the coding so that we can start on our documentation and allow system testing of SPA as a whole.	
All	Begin SPA Design section in the report
Lu Shing	Implement Parser to parse procedure and program Implement AST traversal for Parser to call Design Extractor to extract Design Entities
Jonas	Implement conditional expressions syntax check and parsing of while and if statements
Yu Xuan	Implement extraction of design abstractions
Cheyanne	Finish up implementing storage for design entities and design abstraction and performing unit tests. Clean up codes written in PKB folder.
Irene	Implement evaluation of queries with pattern clauses
Zi Xin	Implement evaluation of queries with multiple clauses
All	Come up with system testing test cases, integration tests and more unit tests

Week 6	
We aim to complete the coding so that we can start on our documentation and allow system testing of SPA as a whole.	
All	Finish up Iteration 1 report
All	Clean up code and complete testing

Week 7	
We aim to fix the bugs from Iteration 1, plan our Iteration 2 activities and start implementing Advanced SPA features.	
All	Come up with a development plan for Iteration 2 and 3
Lu Shing	Brainstorm how to fix bug found in Iteration 1, regarding pattern matching issue Fix issue where SPA does not terminate when encountering a syntax error in SIMPLE source code
Jonas	Implemented the fix for bug found in Iteration 1 based on recommendations from brainstorm on how best to fix the bug
Yu Xuan, Cheyanne	Population of full assign and container pattern information
Zi Xin	Parsing of queries with multiple clauses and 'and' keyword, selection of BOOLEAN and tuples, other additional synonyms such as prog_line, call etc., new design abstractions, new pattern clauses
Irene	Evaluation of queries with multiple clauses, focusing on selection of BOOLEAN and tuples, Call/Call* relationships and table merging correctness

Week 8	
We aim to continue implementing Advanced SPA features, with unit and integration tests for all implemented features.	
All	Improve report based on Iteration 1 feedback and update the Scope, Development Plan, Testing, Coding & Documentation Standards, API Design sections in the report
Lu Shing	Add functionality to Parser to allow for multiple procedures in a program. Add semantics checks to detect when error when facing duplicate procedure names
Jonas	Add semantics checks to detect for cyclical procedure calls Finish AST traversal to call DE methods to call statements and integration test
Yu Xuan	Add DE functionality to handle while and if pattern information
Yu Xuan, Cheyanne	Extraction and population of Call/Call*, and full Modifies/Uses relationship information
Zi Xin	Parsing of queries with with-clauses and attributes
Irene	Evaluation of queries with multiple clauses, focusing on Next/Next* relationships, if and while pattern and table merging correctness

Week 9	
We aim to continue implementing Advanced SPA features and finish the report for Iteration 2.	
All	Complete report for Iteration 2 submission
Yu Xuan, Cheyanne	Population of Next/Next* relationship and with-clause information
Irene	Evaluation of queries with multiple clauses, focusing on with-clauses, and table merging correctness
Zi Xin	Evaluation of queries with multiple clauses, focusing on Affects/Affects* relationship, synonym attributes and table merging correctness

Week 10	
We aim to continue implementing Advanced SPA features. Testing will have an emphasis on writing system tests for the new features.	
All	Make sure the system is well-tested by adding new test cases that are found missing in the demo
Lu Shing, Jonas, Yu Xuan	Write system tests to ensure correctness of Advanced SPA features
Yu Xuan, Cheyanne	Population of Affects/Affects* relationship
Irene	Evaluation of queries with optimization (order groups)
Zi Xin	Evaluation of queries with optimization (group clauses, order clauses, project only relevant columns from intermediate tables)

Week 11	
We aim to begin on the implementation for the extension, with unit and integration tests for the features implemented. We will also continue to write system tests for our SPA system.	
All	Update report for Iteration 3
Lu Shing, Jonas	Write system tests to ensure correctness of Advanced SPA features
Yu Xuan	Clean up code
Yu Xuan	Add DE functionality to support NextBip/NextBip* relationship extension
Cheyanne, Zi Xin	Population of NextBip/NextBip* relationship extension
Irene	Evaluation of queries with optimization (rewrite clauses with information in with clauses)
Zi Xin	Parsing and evaluation of queries, focusing on NextBip/NextBip* relationship

Week 12	
We aim to complete the implementation for the extension, with unit and integration tests for the features implemented. We will also continue to write system tests for our SPA system and update the report for Iteration 3 submission.	
All	Complete Iteration 3 Report Draft
Lu Shing, Jonas, Yu Xuan	Write system tests to ensure correctness of Advanced SPA and extension features
Cheyanne, Zi Xin	Population of AffectsBip/AffectsBip* relationship extension
Irene	Parsing and evaluation of queries, focusing on AffectsBip/AffectsBip* relationship

Week 13	
We aim to complete system testing and the report for Iteration 3, as well as prepare for the final presentation.	
All	Complete report for Iteration 3 submission
All	Complete system testing
All	Conduct presentation rehearsal after code + report submission

3 SPA Design
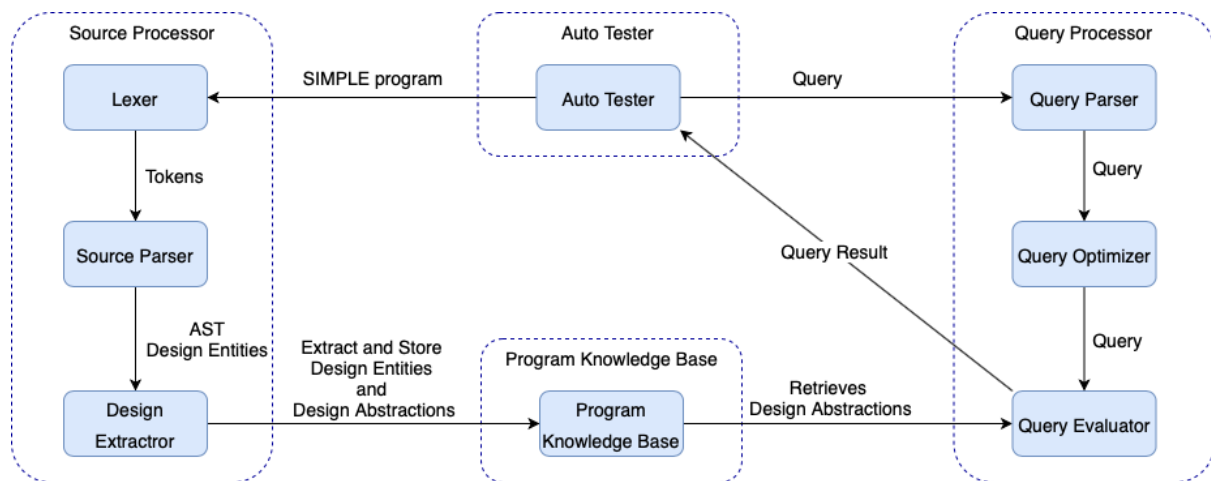
In this section, we give an overview of our main SPA components and the way they interact. The above is a high level architecture diagram illustrating how the components work with each other. Later each subsection would include UML diagrams to explain how each subcomponent works in greater detail as well as the design patterns used and trade offs.

From the above architectural diagram we have chosen for Source Processor to be split into the Parser and the Design Extractor (DE). The Parser would then focus on syntax checking of SIMPLE source code and pass the AST and design entities to the Design Extractor. The DE would in turn extract the required design abstractions and entities and store it in the PKB. These design entities and abstractions would be queried by the QP when required to resolve user FrontEnd PQL queries.

Design Considerations

Our architecture deviates from the standard SPA architecture at the point of communication between SP and PKB. The main consideration was about which SP components should communicate with the PKB. The three approaches we considered are described below.

	Approach 1 Both Parser and DE communicate with PKB	**Approach 2** Only DE communicates with PKB	**Approach 3** Only Parser communicates with PKB (no DE)
Parser	Extract and store design entities	Create AST, parses design entities and passes them to DE	Do everything in Parser
Design Extractor	Extract and store design abstractions	Extracts design abstractions and stores both the entities & abstractions	-

Each approach was evaluated according to the criteria below.

Criteria	Approach 1	Approach 2	Approach 3
Complexity of Implementation	Parser is simpler to write, but DE may become more complicated	Each component has less functionality and is simpler (compared to Approach 3)	Doing everything in parser results in high cohesion
Performance	Slower due to increased overhead of communicating with other components (Requires DE to query the PKB)	Slower due to increased overhead of communicating with other components	May perform better due to reduced overhead caused by communicating with other components like DE
Maintainability	May have to update both Parser and DE if PKB API changes	Only DE needs to be changed if PKB API changes.	Hard to maintain as everything is done in the parser, problems may be harder to identify
Extendibility to future iterations	Extensible if new entities or abstractions are added	Extensible if new abstractions are added	Depends on how Parser is implemented

Eventually, we chose approach 2 where only the DE communicates with the PKB. This is because it is less coupled, as the Parser doesn't need to query the PKB. It follows the separation of concerns principle by distinguishing between parsing functionality and relational extraction functionality, and this also makes it easier to maintain.

3.1 Source Processor

The Source Processor (SP) is the first component of SPA and is responsible for the parsing of the SIMPLE source code and storing of design entities and design abstractions into the PKB.

Design

The Source Processor is made up of:

- Lexer (Section 3.1.2)
  - Token classes (Section 3.1.1)
- Parser (Sections 3.1.3 to 3.1.7)
  - Cond_expr checking Utilities (Section 3.1.4)
  - Node classes of the AST (Section 3.1.5)
  - Semantics Checks (Section 3.1.7)
- Design Extractor (Section 3.1.8)

An overview of how these sub components of the Source Processor works on a high level are described as follows. A more in depth explanation would be covered in their respective annotated sections. For example Lexer would be described in Section 3.1.2.

How It Works
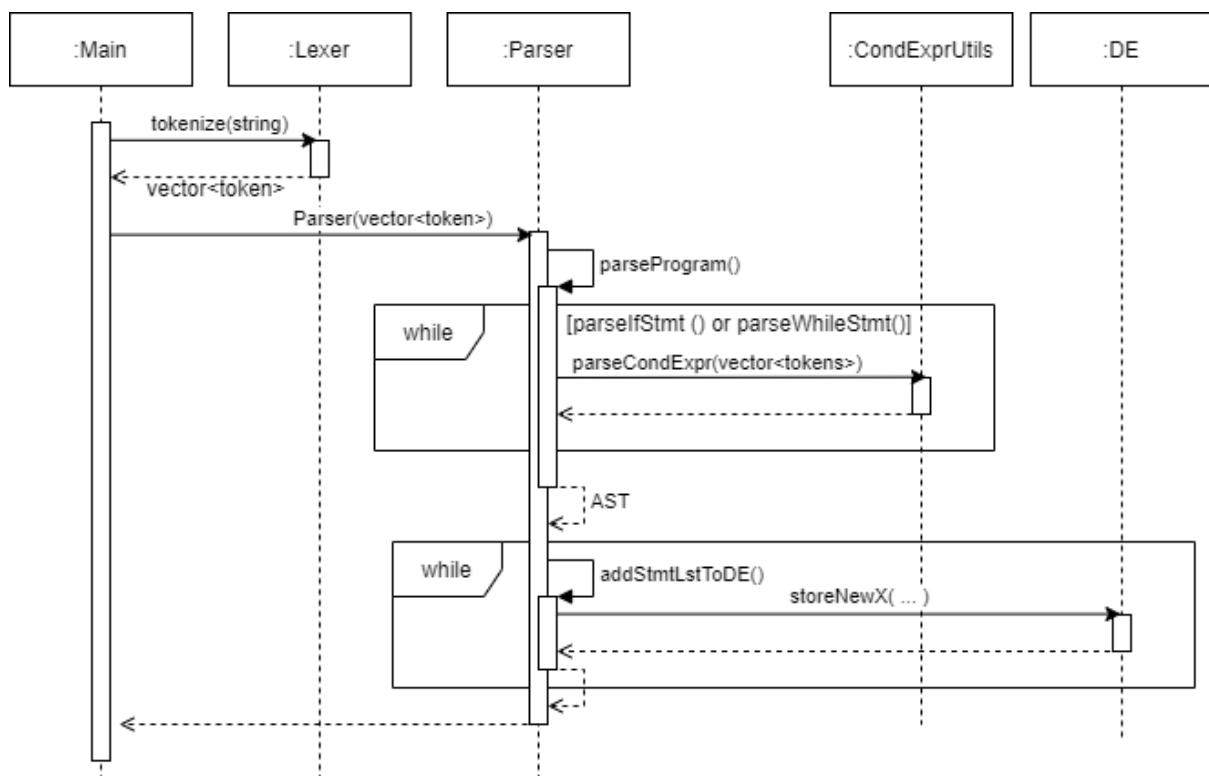
The Lexer (Section 3.1.2) constructs a vector of Token classes (Section 3.1.1) to replace strings as a representation of the SIMPLE source code.

The vector of Tokens are used as an input by Parser (Sections 3.1.2 to 3.1.7) to first do syntax checks on SIMPLE source code. The source code would be rejected if it does not follow the grammar rule of SIMPLE. Secondly Parser also generates the Abstract Syntax Tree (AST) representation of the SIMPLE source code (Section 3.1.6).

The Source Processor uses Node and Token classes to represent the Abstract Syntax Tree (AST). The AST is then traversed for check for semantics (Section 3.1.7). If semantics errors are found, the source code would be rejected and the program terminated.

The Parser will then traverse the AST and call Design Extractor (Section 3.1.8) methods to then extract and store the design entities and abstractions in the PKB (Section 3.2).

Design Considerations

Speed is not a critical factor when dealing with the SIMPLE source code. As such most of the design decisions would focus more on reducing complexity and making it more maintainable and extensible to future iterations.

When designing the components for SP to populate PKB, we were also faced with three choices. The table below shows the considerations made to arrive at our decision.

	Approach 1	Approach 2	Approach 3
Description	Parser extract and store design entities DE extract and store design abstractions	Parser create AST, extract design entities and pass to DE DE extracts design abstractions and store both entities and abstractions	Parser extract and store both design entities and abstractions Absence of DE
Complexity of Implementation	Parser will be simpler to write, but DE will be more complex	Compared to approach 3, Parser will have lesser functionality to implement and simpler Compared to approach 1, DE will also be less complex	Doing everything in Parser may result in high cohesion
Performance	May perform slower due to the need for communication with other components such as DE querying PKB	May perform slower due to the need for communication between the Parser and DE	May perform better due to lesser overhead that can be caused by interaction between Parser and DE
Maintainability	May have to update both Parser and DE if PKB API changes	Only DE needs to be changed if PKB API changes	Hard to maintain as everything is done in the parser, problems may be harder to identify
Extendibility	Extensible if new entities or abstractions are added	Extensible if new abstractions are added	Depends on how Parser is implemented

Approach 2 is chosen as there is less coupling since only one component (DE) needs to interact with PKB. By separating the parsing functionality and relational functionality, it is easier to maintain and allow for easier bug detection and testing. Approach 2 also allows for easier and less complex implementation and is extensible which is important since there may be other requirements in future iterations that the project has to accommodate to. Although performance is not prioritised as mentioned above, we still think that it is important to not have too much performance overhead, thus choosing Approach 2 which has fewer communication between the components.

3.1.1 Parser: Token and TokenTypes

Token are objects which are used to represent source code keywords, operators and lexical tokens. Using a vector of Tokens over a string to represent SIMPLE source code removes the need for the rest of the Source Processor to have to deal with input strings and whitespace of the source code.

Design

Token
+ type: TokenType
+ literal: string

Tokens are simple object classes that contain only a `TokenType` enum and a string `literal`.

The TokenType enum is what the token represents, for example IF, OR, PLUS, PROCEDURE, PRINT, ASSIGN, SEMICOLON, among other types

The string literal is used to distinguish between the same TokenType identifiers CONST, NAME. For example two Tokens with the same NAME TokenType, could refer to the same var_name with literal "x" or to different var_names, with differing literals "x" or "y".

Conclusion

Tokens are simple objects that encapsulate the smallest unit of SIMPLE source code.

3.1.2 Parser: Lexer

SIMPLE source code is represented by a `string` in a file. The Lexer takes an input string and returns a vector of Token objects (Section 3.1.1) representing the SIMPLE source code for use by other sub-components of the parser as the source code.

Design

The Lexer iterates through a string of source code character by character. Based on each `char` type it decides if it is a valid SIMPLE keyword, operator, name, constant and encapsulates it as a Token with the correct TokenType ignoring whitespaces.

Design Considerations

Design decision: have Parser work with strings, or tokenize it and work with tokens.

The choice to use tokenize the source code was based on what has worked well in the past. It is common in most Parsers to tokenize source code before processing it.

	Approach 1: Use Lexer to Tokenize	Approach 2: No Lexer, do not tokenize
Description	Lexer tokenizes SIMPLE source code and the rest of Parser works on the Tokens	Parser needs to read the source code as strings and decide what the strings mean
Complexity of Implementation	Separation of Concerns, String checking responsibilities abstracted away from the rest of parsing logic reducing complexity	No separation of Concerns, Parsing logic required to check strings in addition to parsing the source code increasing its responsibilities
Ease of extension	Easier, reduces coupling allows adding additional tokens to represent new features	Harder, increased coupling caused by parsing logic and string checking being grouped together

Conclusion

The Lexer tokenizes the source code, achieving separation of concern and reducing complexity. The rest of Parser need not deal with the bytes of strings of the source code or lexical checks and can focus on the parsing logic.

3.1.3 Parser: Validating Syntax of SIMPLE with two pointers

One responsibility of the Parser is to ensure the SIMPLE source code conforms to the syntax of the language represented by the grammar.

Design

Parser uses a two pointer approach iterating over the vector of Tokens representing SIMPLE source code. In the following example runs of parseStmt() method, the two pointers are the "current" token in red and the "peek" token in blue.

Based on the tokens it found at these pointers, the Parser method calls other methods to parse the relevant statements or expressions. These methods check the validity of the syntax by asserting a certain order of tokens when it encounters them.

Examples:

parseStmtLst() calls parseStmt() and breaks down into one of several cases depicted below.

1a. current TokenType is NAME, peek TokenType is ASSIGN. parseStmt() method calls parseAssignStmt(). The algorithm parseAssignStmt is covered in Section 3.1.5.

x	=	y	+	1	;

1b. parseAssignStmt() is covered in Section 3.1.5 and deals with the orange region. When it returns, parseStmt() checks the "current" token ensuring it ends with a semicolon.

x	=	y	+	1	;

2a. current TokenType is CALL, peek TokenType is NAME. parseStmt() method calls parseCallStmt().

call	cthulhu	;			

2b. parseCallStmt() ensures "current" TokenType is CALL (above), then it moves the token and ensures the next "current" TokenType is a NAME (below). Call statement grammar is validated and parseCallStmt() returns. If the "current" TokenType is not a NAME, there is a syntax error and the program terminates. This is similar for parseRead() and parsePrint().

call	cthulhu	;			

2c. parseStmt() then moves the "current" token pointer, and checks if the SIMPLE statement ends with a semicolon (below). If it does, it fulfills the "call: 'call' proc_name ';'" grammar for call statements.

call	cthulhu	;				

3a. "current" TokenType is IF, "peek" TokenType is LPAREN. parseStmt() calls parseIfStmt()

if	(	x	<	b	)	then	{	...	}	else	{	...	}

3b. parseIfStmt() asserts that "current" TokenType is IF and "peek" TokenType is LPAREN. It calls parseCondExpr() to conducts syntax check for cond_expr (orange region) explained in Section 3.1.4.

if	(	x	<	b	)	then	{	...	}	else	{	...	}

3c. parseCondExpr() returns. parseIfStmt then asserts that the "current" token is now THEN. parseIfStmt() would then call parseStmtLst() to check the syntax for the orange region.

if	(	x	<	b	)	then	{	...	}	else	{	...	}

3d. upon return, parseIfStmt() then validates the "current" TokenType is ELSE, then calls parseStmtLst() to check the orange region. This process is similar for parseWhile().

if	(	x	<	b	)	then	{	...	}	else	{	...	}

If any of the TokenType asserts fail, then the SIMPLE grammar syntax is not followed by the given source code and the program terminates.

Conclusion

Syntax validation is done as Parser parses the vector of tokens. Each statement or expression has its own methods which assert the requisite TokenType. A missing or unexpected TokenType would mean the SIMPLE source code contains a syntax error and the program terminated.

3.1.4 Parser: Syntax Check of Cond_expr with Utils

One complication faced was syntax checking of the `cond_expr`. Parenthesis "(" and ")" are a compulsory part of the syntax. For example AND operator: '(' cond_expr ')' '&&' '(' cond_expr ')'. This differs from `expr` where the parenthesis are an optional way of grouping expressions to establish precedence. This makes the parsing algorithm used to parse `expr` to not work here and a manual syntax check is used instead.

Design

Because `cond_expr` would not be queried by QP, the AST is not built for the `cond_expr`. The syntax checks were abstracted into the CondExpr Utils namespace of functions. This allows a separate collection of functions specifically for checking syntax of `cond_expr`.

Parser collects all the tokens belonging to the `cond_expr` into a vector while building up a collection of var_names and constants found in the `cond_expr` to pass to DE later. Parser then calls the CondExpr Utils function with the vector of tokens belonging to the cond_expr.

Utils functions provide syntax checking by naively going over the entire `cond_expr` multiple times. In the first pass, it resolves all `rel_expr`, replacing every instance with a placeholder token `BOOL`. The second pass resolves `!`, `&&` and `||` operators while ensuring only `BOOL`, cond_expr operators and correct nesting of parentheses are used.

Example for `while ((a + b < 7) && (4= 5)) { }`

1. First pass: scan for rel_expr

(	(	a	+	b	<	7	)	&&	(	4	! =	5	)	)

2. First pass: check LHS and RHS of rel_expr is a valid expr

(	(	a	+	b	<	7	)	&&	(	4	! =	5	)	)

3. First pass: Replace contents within parenthesis with 'BOOL' placeholder, continue scan

(	(	BOOL	)	&&	(	4	! =	5	)	)

4. First pass: resolve next rel_expr, replacing with `BOOL` placeholder

(	(	BOOL	)	&&	(	4	! =	5	)	)

5. Second pass: scan for `cond_expr` (green)

(	(	BOOL	)	&&	(	BOOL	)	)

6. Second pass: cond_expr parenthesis checks (red)

(	(	BOOL	)	&&	(	BOOL	)	)

7. Second pass: check LHS (blue) and RHS (purple)

(	(	BOOL	)	&&	(	BOOL	)	)

8. then resolve with `BOOL` placeholder

(	BOOL	)

In each of the two passes, the algorithm takes a middle out approach. Starting from the rel_expr operator (eg <, ==, >=) or && and || operators, it proceeds outwards in both directions until it finds the parenthesis pair that encloses that cond_expr or rel_expr.

After ensuring the syntax within the enclosing parentheses are valid, each pass will then replace everything within and including the parenthesis with a placeholder token `BOOL`.

Design Considerations

This was chosen over the alternative of repurposing the existing parsing logic for `expr`. While another parsing algorithm from the one used for parsing `expr` could be possibly found, there was not sufficient time to find, understand and implement it.

Conclusion

Due to issues while attempting to parse cond_expr, a dedicated syntax checker is employed. Because the structure of the AST is not required for cond_expr, information required by the Design Extraction such as variable names and constants are obtained here and placed into a placeholder terminal node and attached into the AST.

3.1.5 Parser: Parsing Algorithm for Expr

There are several ways of parsing source code. We have decided to use the Pratt Parsing algorithm, a variant of recursive descent parsing.

Design

The Pratt Parsing algorithm (Pratt, 1973) is a variant of the Operator Precedence Parsing which is itself a variant of recursive descent. It sidesteps the issue commonly found in left-recursive grammars such as the "expr: expr '+' term | expr '-' term | term" or "term: term '*' factor | term '/' factor | term '%' factor | factor". This issue tends to cause traditional Recursive Descent Parsers to go into an infinite loop which is not desirable.

When parsing left-recursive grammar such as the expression in SIMPLE, the Pratt Parser uses loops as well as recursion to prevent the need to rewrite the grammar or use more advanced Left-Recursion Parsing techniques (Kladov, 2020).

Operator precedence levels are used to allow the parser to determine which operator takes priority. This removes the issue of parsing left-recursive grammar by allowing parsing to be done in a loop. Parsing expressions at this stage does not follow the grammar in its left-recursive form but is still able to accurately represent the left recursive nature of SIMPLE expressions when forming the AST. It also allows maintaining a more natural expression of operator precedence in the code as compared to a purely type based system.

Design Considerations

	Approach 1: Pratt Parsing	Approach 2: Traditional Recursive Descent
Description	Use operator precedence levels and a while loop to parse expressions	Use type based system and recursively attempt to parse expressions
Complexity of Implementation	Low	Either work arounds are needed to avoid being stuck endlessly for certain expressions or harder to implement advanced algorithms are required
Performance	(Not an issue for SP)	(Not an issue for SP)
Memory Utilization	(Not an issue for SP)	(Not an issue for SP)
Ease of extension	Operator precedence levels can be easily swapped around to allow expressions to change	Harder to change type system established

While the use of Pratt Parsing causes deviation from a strict one to one between SIMPLE grammar and AST node classes, the ease of implementation made it an obvious choice.

Conclusion

The left-recursive nature of the SIMPLE grammar can potentially cause issues with traditional recursive descent parsing. The choice of parsing algorithm makes this a non-issue.

3.1.6 Parser: Abstract Syntax Tree and Node Objects

In the process of performing syntax checks, it is possible for the Parser to construct a data structure representing the SIMPLE program. This is the Abstract Syntax Tree (AST).

Tree traversals on the AST are then used for the following purposes:

1. Semantics checks (Section 3.1.7)
2. Calling Design Extractor to extract Design Entities and Abstractions

Design

The AST is a tree constructed out of Node objects. The Nodes are split among different derived classes to allow the type system of C++ to convey the kind of child node each node is allowed to have. This makes the syntax of the SIMPLE program evident in the various Node derived classes.

The tree structure of the AST also allows for encoding of information that can be used later in the SP. These can be explicitly like Statement Number or implicitly like the Parent design abstraction between statements. It also makes the left-recursive nature of the expressions in SIMPLE explicit whereas it is not evident in the string representation.

Below is a sample AST object diagram.
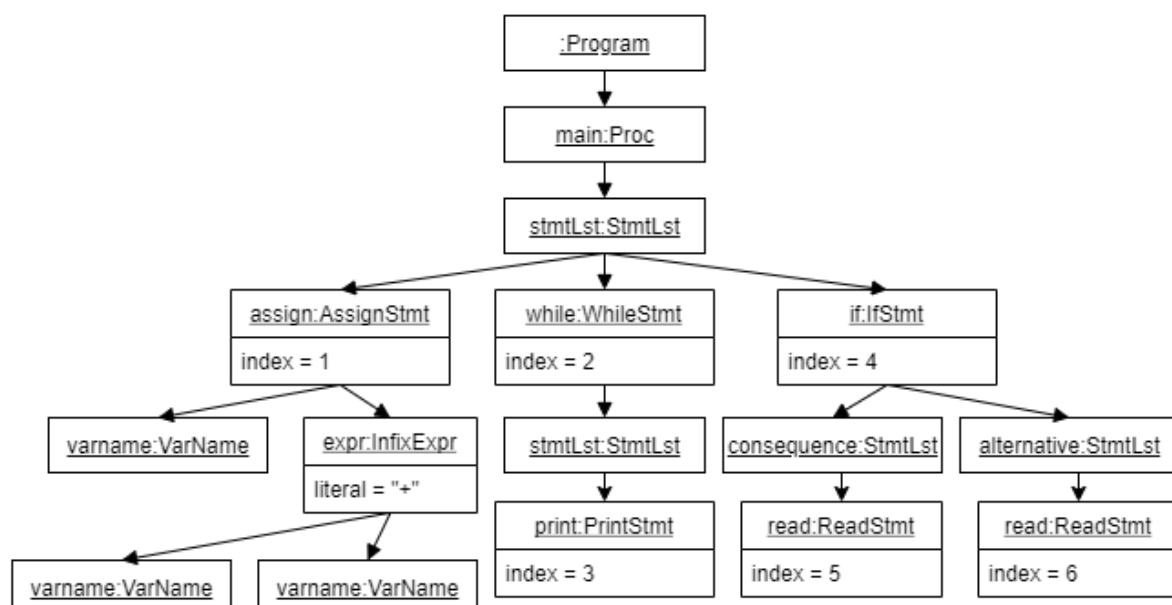
Each object is a different type of Node such as Program, Proc, and StmtLst. Stmt objects such as AssignStmt, WhileStmt have an index to represent their Statement Number. Expressions are used to represent the right-hand side of an assign statement and can be split further into other classes such as InfixExpr and VarName. Expressions contain literal to show the operators used.

When checking the syntax of the source code, the construction of the AST ensures only a valid syntax would allow for the proper AST node object to be constructed. For example, an AssignStmt class representing the `assign` SIMPLE statement contains exactly one VarName object and one Expr object. These represent a `var_name` and an `expr` defined by the SIMPLE grammar respectively.

Design Considerations

	Approach 1: Build AST	Approach 2: Do not build AST
Description	Construct an AST while parsing source code, traverse and store into PKB	Do not construct AST, just store the information required into the PKB
Complexity of Implementation	Requires usage of additional Node classes. AST would be created as a product of syntax checking. Not much additional code required.	No additional classes required. Syntax checking and design abstraction extraction would be coupled
Performance	(Not an issue for SP)	(Not an issue for SP)
Memory Utilization	(Not an issue for SP)	(Not an issue for SP)
Ease of extension	Easier to traverse AST after construction allowing multiple passes if required for future extensions	Harder to implement multiple passes if required in the future. Future extensions would have to be done in one pass.

It should be emphasized that this section governs only the decision to build an AST or not. Despite building the AST, information is stored in the PKB differently. As such performance is not an issue in this consideration as the AST is used only within the SP as a representation of the SIMPLE source program. Should the AST be used for queries, the performance of such a usage will be revisited again in another section.

Conclusion

Building the AST allows Parser to conduct multiple passes of the source code with minimal additional cost. Parser would not need to do syntax checks, semantic checks and calling design extractor all in one pass. These tasks can be split into their own sub-components, each performing its own pass of the source code reducing overall complexity.

3.1.7 Syntax Check

After Parser checks for syntax ([Section 3.1.3](#) to [Section 3.1.5](#)), Parser performs semantics checks before calling the Design Extractor.

Design

The following semantics checks are performed:

1. Duplicate Procedure Names
2. Calls to undefined procedures
3. Cyclical Procedure calls.

Duplicate procedure names are checked by building a hashset of existing procedure names. By checking the Program Node of the AST, if a procedure name already exists in the hashset, a duplicate name is found and the program terminates.

Calls to undefined procedures are checked by building a hashset of procedures defined. Then ensure each "Call" statement is made to a proc_name that exists in the hashset.

Checks for cyclical procedure calls are done by running a depth first search in the AST while maintaining an explicit stack for cycle detection. Visiting each procedure would add it to the stack, and any call to a procedure made already in that stack would be a semantic error resulting in program termination.

Conclusion

Semantics check algorithms outlined in this section detect semantic errors before design extraction begins.

3.1.8 Design Extractor

The DE is the sole component in the SP that communicates with the PKB. In our design, it is responsible for extracting program design entities and abstractions before storing that information into the PKB.

Design

Fundamentally, the DE is called by the Parser for every single statement in the source program through the DE's public API (e.g. storeNewAssignment is a function).

The DE is responsible for extracting all program design abstractions except for the starred variants of each, as seen in the table below. Any remaining relationships are handled by the PKB.

	DE	PKB
Relationships Handled	Follows Parent Uses Modifies Calls Next	Follows* Parent* Calls* Next* Affects Affects*

This is because we consider extraction and computation separate responsibilities. Extraction involves usage of the Parser's output whereas computation only involves operations on existing data added by the DE. For example, the starred variants of the relationships are computed by calculating the transitive closure of the non-starred relationships.

To extract relationships, the DE needs to track state across different statements. The DE makes use of the singleton pattern, and uses static attributes in the class to store the local state variables. These variables can be split into 2 general categories, as seen in the table below. The first is the 'working set', which tracks information about the current locale: for instance, its parent, the statements in its statement list, and so on. The current locale can refer to a procedure's statement list or the statement list of a container statement.

Working Set		Stack Variables	
Name	Type	Name	Type
currentProcedureID	ID	stmtLstsStack	vector<vector<StmtNum>>
currentStmtLst	vector<StmtNum>	usesStack	vector<set<ID>>
currentParent	StmtNum	modifiesStack	vector<set<ID>>
currentModifiedVarsLst	set<ID>	parentStack	vector<StmtNum>
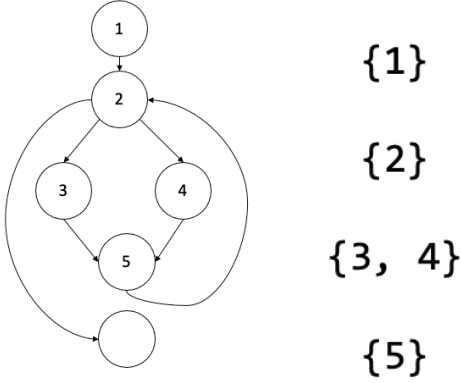
currentUsedVarsLst	set<ID>	nextStack	vector<vector<set<ProgLine> >>
currentNext	vector<set<ProgLine >>		

The remaining local state variables as shown in the right of the table are stacks that store information preceding the current locale. For example, the stmtLstsStack will store all statement lists prior to the current locale. These are used to handle the relationships of container statements like if and while statements which have their own statement lists and additional relationships to extract.

Handling Relationships

The table below explains how the DE processes each relationship it is responsible for and how those relationships are populated into the PKB.

Relationship	
Follows	Data Structures: **currentStmtLst** and **stmtLstsStack** currentStmtLst keeps a list of the statement numbers in a particular statement list. At the end of a statement list, use PKB's **storesFollows**(stmt1, stmt2) to store the Follows relationship between pairs of consecutive statement numbers. If a new statement list is encountered, currentStmtLst can be pushed onto the stmtLstsStack. The new statement list's statement number then populates currentStmtLst. At the end of it, the Follows relationships are stored as described above and the stmtLstsStack is popped, restoring the original currentStmtLst.
Parent	Data Structures: **currentParent** and **parentStack** Initially set to NULL at the start of a procedure to indicate 'no parent', currentParent stores the statement that is the parent of the statement list. If a container statement is encountered, currentParent is updated to be the statement's statement number. Thereafter, at the end of a statement list: for every statement in the currentStmtLst, PKB's **storesParent**(currentParent, childStatement) is added. When there are nested container statements, the currentParent will be stored in the parentStack and the nested container statement's statement number becomes the new currentParent. After the nested container statement is processed, the stack is popped to restore the currentParent.
Uses	Data Structures: **currentUsedVarsLst** and **usesStack** When variables appear in the RHS of assignment statements, appear in the conditional statements of if/while statements, or used in print statements, they are stored into the PKB using its **storeStmtUses**(stmt#, variableID) call. Additionally, they are stored into the currentUsedVarsLst. At the end of a procedure, all variable IDs in the currentUsedVarsLst are stored into the PKB using its **storeProcUses**(procedureID, variableID) call. Also, at the end of a container statement's statement list, PKB's **storeStmtUses**(parentStmt#, variableID) is called on the variable IDs in currentUsedVarsLst in order to add Uses relationships between parent and child.
Modifies	Data Structures: **currentModifiedVarsLst** and **modifiesStack** When variables appear in the LHS of assignment statements, or modified in read statements, they are stored into the PKB using its **storeStmtModifies**(stmt#, variableID) call. Additionally, they are stored into the currentModifiedVarsLst.

	At the end of a procedure, all variable IDs in the currentModifiedVarsLst are stored into the PKB using its **storeProcModifies**(procedureID, variableID) call. Also, at the end of a container statement's statement list, PKB's **storeStmtModifies**(parentStmt#, variableID) is called on the variable IDs in currentModifiedVarsLst in order to add Modifies relationships between parent and child.
Next	Data Structures: **currentNext** and **nextStack** currentNext is a vector of sets containing ProgLines. It can be understood as a CFG, where the ProgLines in each set have a Next(n1, n2) relationship with the ProgLines in the set succeeding it. This is illustrated in the CFG below: Next(1, 2) holds and thus the set {1} is succeeded by the set {2}. Then, Next(2, 3) and Next(2,4) hold. Thus the set {3, 4} is preceded by {2}. At the end of a statement list, PKB's **storeNext**(progLine1, progLine2) is used to store the Next relationship between the ProgLines in the pairs of sets within currentNext as described above. **How currentNext is populated** For print, read and call statements, the stmt# is simply added to currentNext. *While Statements* The while statement's statement number (startStmt#) is added to currentNext before currentNext is saved to nextStack. Then, an empty currentNext is instantiated to process the new statement list. The startStmt# is also added to currentNext--this allows Next to be added between the while loop and the statements it contains. At the end of the while statement, the startStmt# is appended to currentNext to complete the path of the while loop. This marks the end of the statement list, upon which the information in currentNext will be stored as described above. *If Statements* The if statement's statement number (startStmt#) is added to currentNext before currentNext is saved to nextStack. Then, an empty currentNext is

	instantiated to process the new statement list. The startStmt# is also added to currentNext--this allows Next to be added between the if statement and the statements it contains. After the if-then portion of the if statement, the DE will have to process the else portion. As this signals the end of the if-then statement list, the information in currentNext is stored into PKB as described above. Then, it stores the lastStmt# of the if-then statement list into nextStack. At the end of the else portion, currentNext is restored by popping nextStack. The lastStmt# of the else statement list is appended to the last set in currentNext. This means that the lastStmt# for both branches of the if statement are stored in the same set within currentNext (e.g. set {3, 4} in the pictorial example above), allowing Next relationships to be added for both accordingly. After this, the information in currentNext will be stored as described above.
Calls	When a call statement is encountered, the DE uses PKB's **storeCalls**(stmtNum, callerProcedureID, calledProcedureID) to store the Calls relationship. Additional Modifies/Uses relationships due to Call statements are added by the PKB after all statements have been parsed.

As an example, we can look at the DE's storeNewRead API call. When the Parser encounters a read statement, it calls DE's storeNewRead and provides it the statement number and the name of the variable used, as well as the statement's AST. The purpose of this function is to extract all the program design entities and abstractions from that read statement. The sequence diagram below shows how the DE stores this information into the PKB. As a read statement modifies the variable it reads, the Modifies relationship is also stored into the PKB.
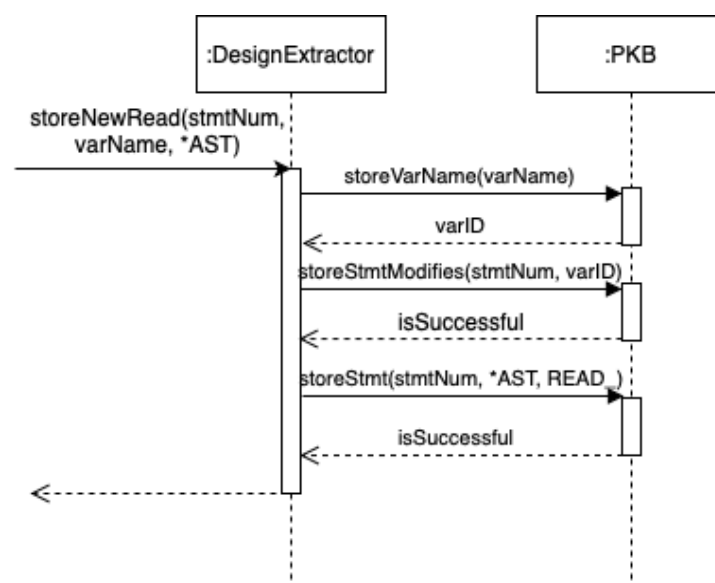
The read statement may have Follows, Parent and Next relationships, and both Modifies and Uses for procedures and container statements. To extract these relationships, storeNewRead will add the given statement number to currentStmtLst and currentNext, and add the given variable's ID to the list of currently modified variables (currentModifiedVarsLst). Based on these input data, the DE processes all these relationships at the end of a statement list instead of individually, for all statements in the list. Therefore, at the end of a statement list, a series of operations are carried out to store the program design entities and relationships that are currently in the working set. The activity diagram below illustrates what the DE stores into the PKB. Overall, any non-container statement is handled in a similar way as the storeNewRead example.
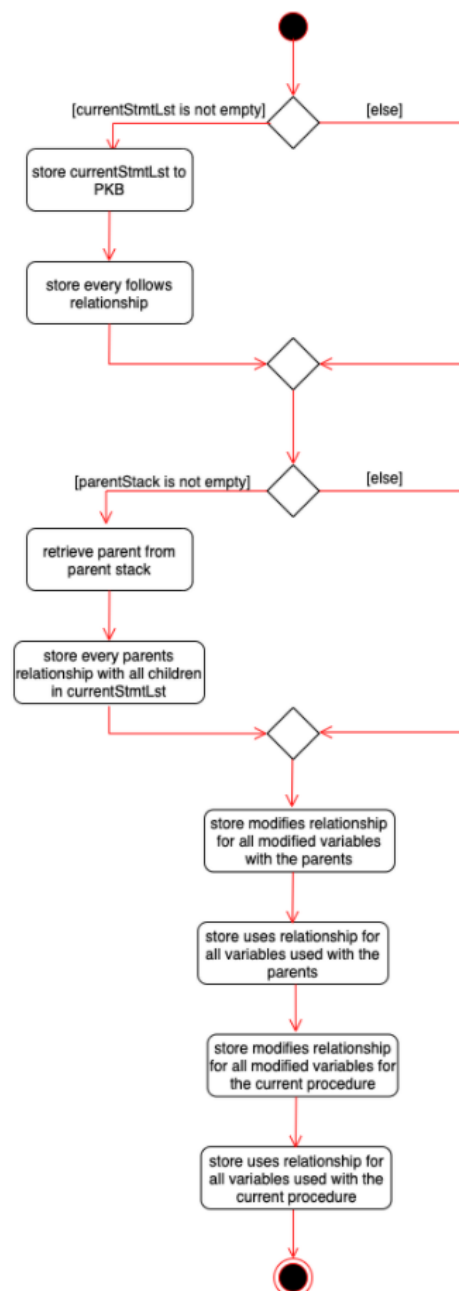
For container statements, the idea is similar, only with the additional idea of using stack variables. When a new container statement is encountered, for example a while loop, the DE will process it similarly to the storeNewRead example above. It stores the program design entities, Uses relationships for the variables in the conditional statement and so on. However, all statements after this (until the termination of the while loop) would be in the while loop's statement list. Thus the DE will store the current working set into the stack variables and initialise a new, empty working set for subsequent statements to use. At the end of the while loop, the DE stores all relationships in the working set and retrieves the previous working set from the stack variables. As we use a stack, this design naturally extends to handle nested container statements.

Design Considerations

The main consideration around the DE's design was whether explicit stacks or implicit stacks should be used. The table below shows the points considered when we made our decision.

	Explicit Stack	Implicit Stack
Description	Actual stacks are stored as variables in the DE.	The call stack is used by using recursion to traverse the AST.
Maintainability	Easier to debug.	Harder to debug, but more readable and elegant.
Performance	Could be faster than implicit stack because there is no need to create or tear down call stack frames.	Could be faster if code is written to take advantage of tail call optimization.
Likelihood of overflow	Stack overflow is less likely.	Stack overflow is possible. There is a limit to the depth of the recursion, which could pose problems if the source program is long and complicated

Additionally, the singleton design pattern was chosen because the SPA is only required to parse one SIMPLE source program at a time, and does not need to keep state from past source programs. Thus, only one instance of the DE should exist at the same time. Testing problems due to the singleton design pattern is not a major issue in this case, because C++ does not inherently have garbage collection and state variables can be cleared at will.

Conclusion

The DE is responsible for extracting program design abstractions as well as storing both the abstractions and entities into the PKB. It uses local variables to keep state between statements in a procedure, which is used to extract relationships.

3.2 PKB

The main data structures used by the PKB to store information are combinations of unordered maps, unordered sets, vectors and pairs. We take into consideration the performance and behaviour of each data structure as well as the needs of the QP.

Design and Design Considerations

Name	Data Structure Used	Reasons
varNameIDMap procNameIDMap constMap	**Chosen:** unordered_map<STRING, INT>	- Need a mapping from the variable/procedure name to it's ID, and constant string to its integer value. - To have a fast look-up for the ID/integer value given the string.
	Alternative Considered: Just have vector<STRING> (just keeping varNames and procNames below)	- Need to use the find(), which is O(N). - V.s. searching for elements in unordered_map has an average constant-time complexity.
varNames procNames	**Chosen:** vector<STRING>	- The "reverse maps" of varNameIDMap and procNameIDMap. - Vectors can be used here because we set the ID of variables and procedures based on the insertion order - The index of a variable/procedure in the vector corresponds to the ID of the variable/procedure.
	Alternative Considered: unordered_map<INT, STRING>	- Vectors perform slightly better than unordered_maps in random access, especially when comparing the worst-case, due to caching.
consts stmtNums assignStmtNums readStmtNums printStmtNums callStmtNums whileStmtNums	**Chosen:** vector<INT>	- Not worried about having duplicates in the data structure - Duplicates are handled separately. - Orders won't matter - Simply insert new elements to the back

ifStmtNums stmtLst	**Alternative Considered:** list<INT>	- Inserting elements at the back is always faster in vectors - No need any random insertion or removal - Finding elements is also faster in vectors
assignExprMap	**Chosen:** unordered_map<INT, pair<STRING, STRING>>	- Need to map the statement number to the pair - need a fast look-up to find the pair, given the StmtNum.
	Alternative Considered: unordered_map<INT, tuple<STRING, STRING>>, unordered_map<INT, vector<STRING>>	- For each assign statement, we want to store exactly two pieces of information – the LHS of the assign statement and the RHS of the assign statement. → use pairs
	Alternative Considered: vector<pair<STRING, STRING>>	- Did not use because the insertion order does not correspond to the StmtNum of the procedure (not all statements are assign statements).
procStmtMap	**Chosen:** unordered_map<INT, pair<INT, INT>>	- Need to map the procedure to the first and last statement of the procedure - Need a fast look-up to find the pair, given the procedure ID.
	Alternative Considered: unordered_map<INT, tuple<INT, INT>>, unordered_map<INT, vector<INT>>	- Need to store the first statement and the last statement for each procedure (exactly two values) → use pairs
	Alternative Considered: vector<pair<INT, INT>>	- Did not use it because the insertion order does not correspond to the ID of the procedure.
callsRawInfoTable	**Chosen:** unordered_map<INT, pair<INT, INT>>	- Need to map the statement number to the caller and callee - Need a fast look-up to find the pair, given the statement number.
	Alternative Considered: unordered_map<INT, tuple<INT,	- Need to store the caller and the callee for each call

	INT>>, unordered_map<INT, vector<INT>>	statement (exactly two values) → use pairs
	Alternative Considered: vector<pair<INT, INT>>	- Did not use it because the insertion order does not correspond to the statement number.
stmtASTMap	**Chosen:** unordered_map<INT, ast::Stmt*>	- Need a mapping from a StmtNum to its corresponding AST node - need a fast retrieval of the AST node given the StmtNum.
	Alternative Considered: vector<ast::Stmt*>	- Not used because it is not guaranteed that the order we insert corresponds to the StmtNum that we are inserting
reverseParentMap followsMap reverseFollowsMap readVariablesMap printVariablesMap callsStmtCalleeMap	**Chosen:** unordered_map<INT, INT>	- Need a mapping from a StmtNum to its corresponding StmtNum/ID (depending on the map). - Need a fast retrieval of the StmtNum/ID given the key StmtNum.
	Alternative Considered: vector<INT>	- Not used because it is not guaranteed that the order we insert corresponds to the StmtNum that we are inserting
parentMap parentStarMap reverseParentStarMap followsStarMap reverseFollowsStarMap stmtModifiesMap stmtUsesMap reverseStmtModifiesMap reverseStmtUsesMap procModifiesMap reverseProcModifiesMap procUsesMap reverseProcUsesMap reverseReadVariablesMap reversePrintVariablesMap ifPatternsMap whilePatternsMap reverseIfPatternsMap reverseWhilePatternsMap reverseCallsStmtCalleeMap	**Chosen:** unordered_map<INT, unordered_set<INT>>	- In these maps, the value has to be an unordered_set, because for each key, there are multiple statements/IDs that we want to store.
	Alternative Considered: unordered_map<INT, vector<INT>>	- Not used because we want to make sure that we do not have duplicates

callsMap reverseCallsMap callsStarMap reverseCallsStarMap nextMap reverseNextMap nextStarMap reverseNextStarMap affectsMap reverseAffectsMap affectsStarMap reverseAffectsStarMap		
callsStmtMap	**Chosen:** unordered_map<INT, unordered_map<INT, unordered_set<INT>>>	- Need a mapping from the caller to the callee to the statement numbers of such calls. - Need a fast look-up of where Calls(p,q) occurs given p and q.

Below is a sample SIMPLE program which will be used to illustrate what is stored in each data structure.

```
procedure p {
1.     a = b + 1;
2.     read c;
3.     while (d == e) {
4.          if (f > 5) {
5.               print g;
             } else {
6.               call q;} }
7.          b = a + 1;
procedure q {
8.     f = a;
}
```

Map	What is stored	Map	What is stored
varNameIDMap	{"a":1, "b":2, "c":3, "d":4, "e":5, "f":6, "g":7}	followsMap	{1:2, 2:3}
varNames	["a", "b", "c", "d", "e", "f", "g"]	reverseFollowsMap	{2:1, 3:2}
procNameIDMap	{"p":1, "q":2}	followsStarMap	{1:{2,3}, 2:{3}}
procNames	["p", "q"]	reverseFollowsStarMap	{2:{1}, 3:{1,2}}
procStmtMap	{1: <1, 7>, 2: <8,8>}	stmtModifiesMap	{1:{1}, 2:{3}, 3:{2,6}, 4:{6}, 6:{6}, 7:{2}, 8:{6}}
constMap	{"1":1, "5":5}	reverseStmtModifiesMap	{1:{1}, 2:{3,7}, 3:{2}, 6:{3,4,6,8}}
const	[1, 5]	procModifiesMap	{1:{1,2,3,6}, 2:{6}}
stmtNums	[1, 2, 3, 4, 5, 6, 7, 8]	reverseProcModifiesMap	{1:{1}, 2:{1}, 3:{1}, 6:{1,2}}
stmtASTMap	{1:*ast, 2:*ast, 3:*ast, 4:*ast, 5:*ast, 6:*ast, 7:*ast, 8:*ast}	stmtUsesMap	{1:{2}, 3:{1,4,5,6,7,8}, 4:{1,6,7}, 5:{7}, 6:{1}, 7:{1}, 8:{1}}
assignStmtNums	[1, 7, 8]	reverseStmtUsesMap	{1:{3,4,6,7,8}, 2:{1}, 4:{3}, 5:{3}, 6:{3,4}, 7:{3,4,5}}
assignExprMap	{1:<"a", "b + 1">, 7:<"b", "a + 1">, 8:<"f", "a">	procUsesMap	{1:{1,2,4,5,6,7}, 2:{1}}
readStmtNums	[2]	reverseProcUsesMap	{1:{1,2}, 2:{1}, 4:{1}, 5:{1}, 6:{1}, 7:{1}}
readVariablesMap	{2:3}	callsRawInfoTable	{6, <1,2>}
reverseReadVariablesMap	{3:{2}}	callsMap	{1:{2}}
printStmtNums	[5]	reverseCallsMap	{2:{1}}
printVariablesMap	{5:7}	callsStarMap	{1:{2}}
reversePrintVariablesMap	{7:{5}}	reverseCallsStarMap	{2:{1}}

callStmtNums	[6]	callsStmtCalleeMap	{6:2}
whileStmtNums	[3]	reverseCallsStmtCalleeMap	{2:{6}}
whilePatternsMap	{3:{4, 5}}	callsStmtMap	{1:{2:{6}}}
reverseWhilePatternsMap	{4:{3}, 5:{3}}	nextMap	{1:{2}, 2:{3}, 3:{4}, 4:{5,6}, 5:{7}, 6:{7}, 7:{3}}
ifStmtNums	[4]	reverseNextMap	{2:{1}, 3:{2,7}, 4:{3}, 5:{4}, 6:{4}, 7:{5,6}}
ifPatternsMap	{4:{6}}	nextStarMap	{1:{2,3,4,5,6,7}, 2:{3,4,5,6,7}, 3:{3,4,5,6,7}, 4:{3,4,5,6,7}, 5:{3,4,5,6,7}, 6:{3,4,5,6,7}, 7:{3,4,5,6,7}}
reverseIfPatternsMap	{6:{4}}	reverseNextStarMap	{2:{1}, 3:{1,2,3,4,5,6,7}, 4:{1,2,3,4,5,6,7}, 5:{1,2,3,4,5,6,7}, 6:{1,2,3,4,5,6,7}, 7:{1,2,3,4,5,6,7}}
stmtLst	[1, 4, 5, 6, 8]	affectsMap	{1:{7}}
parentMap	{3:{4, 7}, 4:{5, 6}}	reverseAffectsMap	{7:{1}}
reverseParentMap	{4:3, 5:4, 6:4, 7:3}	affectsStarMap	{1:{7}}
parentStarMap	{3:{4, 5, 6,7}, 4:{5, 6}}	reverseAffectsStarMap	{7:{1}}
reverseParentStarMap	{4:{3}, 5:{3, 4}, 6:{3,4}, 7:{3}}		

Conclusion

Unordered maps is our most used data structure. It is used when the ordering is not important. It has an average of O(1) for insertions, searches and deletions and is chosen over vectors to speed up information access in the PKB. The PKB also stores the reverse maps. For instance we have a followsMap where the key is the s1 and value is s2 such that Follows(s1, s2), and we also have the reverseFollowsMap where the key is s2 and value is s1 such that Follows(s1, s2). This is to take advantage of the average O(1) searches for queries for either a known s1 or a known s2.

We also use unordered sets when ordering is not important and when we want to ensure that there are no duplicate values. On average, all operations on unordered sets take O(1) times. For instance, in the parentMap, we used an unordered map where the key is the parent, and the value is a set of children. This is because a parent can have many children, but the children should all be unique.

Vectors are used when the insertion order matters. For instance, we have used vectors to store the variable names, such that the index of the variable corresponds to its variable ID.

Within some unordered maps, we have a Pair data structure as the value. For instance, in the procStmtMap, we store the procedure ID as the key and a pair of <startStmt, endStmt> as value (startStmt is the first statement in the procedure, endStmt is the last statement in the procedure). For each procedure we want to store exactly these two pieces of information and hence, Pair was used.

3.3 Query Processor

The Query Processor is the last component of SPA and is responsible for the parsing of queries and evaluating queries by calling the PKB's API.

Design

The Query Processor consists of the Query Preprocessor (Parser + Validator), the Query Optimizer and the Query Evaluator. The Query Preprocessor will take in queries, do validation checks and create a Query Object to be passed on to the Query Optimizer. The Query Optimizer performs optimization and returns the optimized Query Object to be passed on to the Query Evaluator. The Query Evaluator will then call the relevant helper evaluators to handle the queries.

Design Considerations

Performance is a critical factor when handling queries. As such, most of the heavy computational work is handled by the Source Processor and the PKB. The Query Processor is only responsible for calling functions in the API which the PKB exposes.

3.3.1 Query Preprocessor

The Query Preprocessor takes in a raw query in the form of a string and validates it before creating a Query Object.

Design

The following activity diagram illustrates how the Query Preprocessor parses and validates a query.
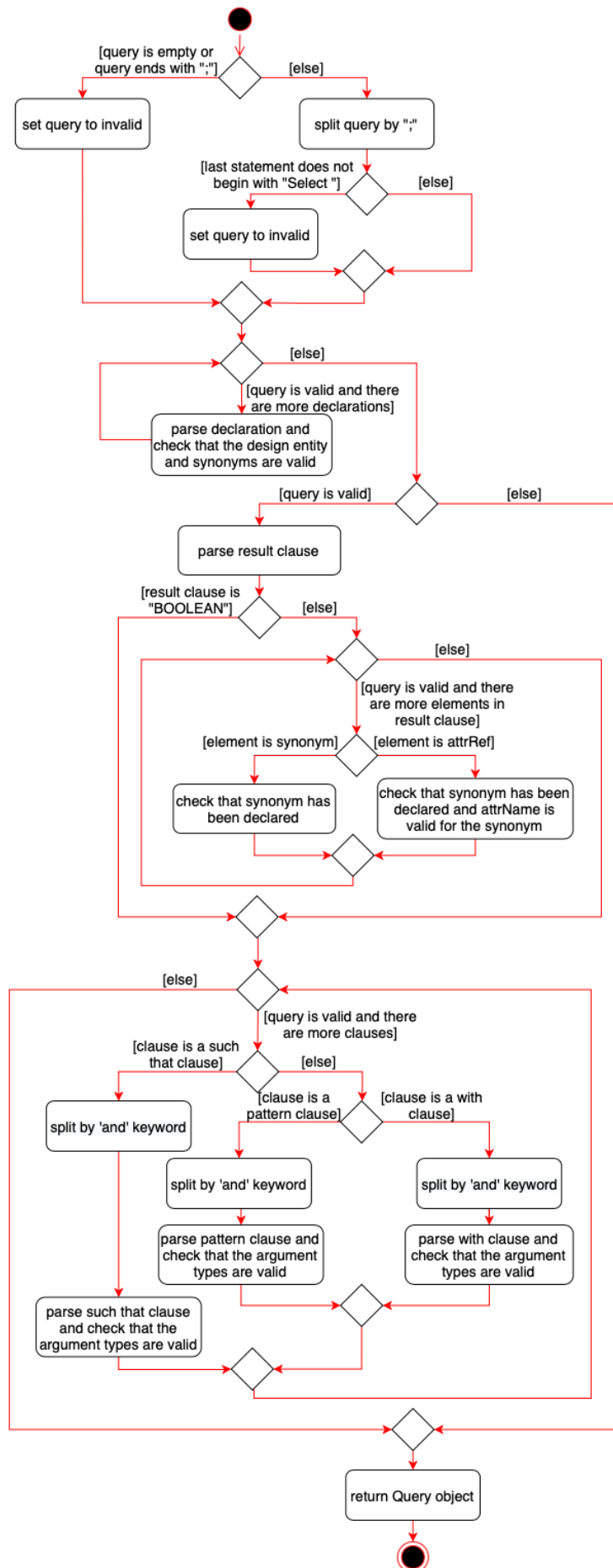
The Query Preprocessor splits the query by semicolons into a vector of statements and checks if the last statement begins with the keyword "Select" and does not end with a semicolon.

For each statement excluding the last, the Query Preprocessor parses and validates the declarations. It will check if the design entities i.e., 'stmt' | 'read' | 'print' | 'call' | 'while' | 'if' | 'assign' | 'variable' | 'constant' | 'procedure' | 'prog_line', and the name of the synonym is valid before checking if the synonym has been declared previously. If a synonym has been declared before, the query will be considered invalid.

The Query Preprocessor then parses and validates the select statement by checking if the syntax for BOOLEAN or the tuple to select is valid. For each element in the tuple, the Query Preprocessor checks if it is a synonym or an attrRef. For synonyms, it checks if the synonym has been declared. For attrRef, it splits the attrRef into synonym and attrName, checks if the synonym has been declared and checks if the attrName is valid for the synonym type.

To parse multiple clauses, the Query Preprocessor looks for the " such that ", " pattern " and " with " keywords. Between each of these keywords, the Query Preprocessor looks for the " and " keyword and parses each of the clauses. It will also check if the arguments in the clause are valid as specified by SPA requirements by parsing the arguments and identifying the argument type. Listed below is how we classify arguments into their argument types.

Argument in clause	Example	Argument type
Synonym	a	Design entity of the declared synonym
Statement number	3	Integer
Underscore	_	Underscore
Variable name in quotes	"x"	Name
Expression in quotes	"x + y"	Expression
Expression in quotes, with underscore	_ "x + y" _	Expression with underscore

To validate clauses in the query, we adopt a table-driven approach. We have validArgType maps for such that, pattern and with clauses. The validSuchThatArgType map maps each design abstraction to a vector of sets and the validPatternArgType map maps the keywords 'assign', 'while' and 'if' to a vector of sets. In both maps, the nth set in the vector includes all valid argument types for the nth

argument in the clause. For validation of attrRef, we have an attrMap, which maps synonym types to a set of valid attrNames. The following table shows key-value pairs for the validArgType maps we use for query validation.

Map	Key	Value
validSuchThatArgType	Follows / Follows*	{ { STMT_, READ_, PRINT_, ASSIGN_, CALL_, WHILE_, IF_, PROGLINE_, INTEGER_, UNDERSCORE_ }, { STMT_, READ_, PRINT_, ASSIGN_, CALL_, WHILE_, IF_, PROGLINE_, INTEGER_, UNDERSCORE_ } }
	Parent / Parent*	{ { STMT_, WHILE_, IF_, PROGLINE_, INTEGER_, UNDERSCORE_ }, { STMT_, READ_, PRINT_, ASSIGN_, CALL_, WHILE_, IF_, PROGLINE_, INTEGER_, UNDERSCORE_ } }
	Uses	{ { STMT_, PRINT_, PROCEDURE_, ASSIGN_, CALL_, WHILE_, IF_, PROGLINE_, INTEGER_, NAME_ }, { VARIABLE_, NAME_, UNDERSCORE_ } }
	Modifies	{ { STMT_, READ_, PROCEDURE_, ASSIGN_, CALL_, WHILE_, IF_, PROGLINE_, INTEGER_, NAME_ }, { VARIABLE_, NAME_, UNDERSCORE_ } }
	Calls / Calls*	{ { PROCEDURE_, NAME_, UNDERSCORE_ }, { PROCEDURE_, NAME_, UNDERSCORE_ } }
	Next / Next*	{ { STMT_, READ_, PRINT_, ASSIGN_, CALL_, WHILE_, IF_, PROGLINE_, INTEGER_, UNDERSCORE_ }, { STMT_, READ_, PRINT_, ASSIGN_, CALL_, WHILE_, IF_, PROGLINE_, INTEGER_, UNDERSCORE_ } }
	Affects / Affects*	{ { STMT_, ASSIGN_, PROGLINE_, INTEGER_, UNDERSCORE_ }, { STMT_, ASSIGN_, PROGLINE_, INTEGER_, UNDERSCORE_ } }
validPatternArgType	ASSIGN_	{ { VARIABLE_, NAME_, UNDERSCORE_ }, { UNDERSCORE_, NAME_, EXPRESSION_, EXPRESSIONWITHUNDERSCORE_ } }
	WHILE_	{ { VARIABLE_, NAME_, UNDERSCORE_ }, { UNDERSCORE_ } }
	IF_	{ { VARIABLE_, NAME_, UNDERSCORE_ }, { UNDERSCORE_ }, { UNDERSCORE_ } }
attrMap	PROCEDURE_	{ "procName" }
	CALL_	{ "procName", "stmt#" }

	VARIABLE_	{ "varName" }
	READ_ / PRINT_	{ "varName", "stmt#" }
	CONSTANT	{ "value" }
	STMT_ / WHILE_ / IF_ / ASSIGN_	{ "stmt#" }

For example, consider the query `assign a; Select a pattern a (_, "x + y")`. Since this is a pattern clause, the Query Preprocessor checks if the argument type of the synonym `a` is one of 'assign', 'while' or 'if'. In this case, the argument type of `a` will be classified as 'assign'. It then looks up the value 'assign' in the validPatternArgTypeMap. The key 'assign' is mapped to `{ { VARIABLE_, NAME_, UNDERSCORE_ }, { UNDERSCORE_, NAME_, EXPRESSION_, EXPRESSIONWITHUNDERSCORE_ } }`. The Query Preprocessor then checks that the argument types of the arguments in the clause is one of the valid argument types in the set. In this case, the argument type of `_` will be classified as 'underscore' and the argument type of `"x + y"` will be classified as 'expression', which are both valid.

Besides checking the argument types, the Query Preprocessor also checks for semantically invalid arguments. For example, if an underscore is the first argument for a Modifies or Uses clause, the query will be considered invalid. The validated queries will then be stored in a Clause Object, which consists of a relationship, i.e., Follows, Follows*, Parent, Parent*, Uses, Modifies, Calls, Calls*, Next, Next*, Affects, Affects*, empty string (for with clause) or a synonym (for pattern clause), and a vector of strings which are the arguments in the clause.

After all validation checks are done, the Query Preprocessor creates a Query Object, which consists of a few data structures. First, an unordered map of declarations where the key-value pairs are the synonyms and its design entity. Second, a string of the synonym queried. Third, a vector consisting of vectors of clauses, where each clause is represented by a Clause Object. Lastly, two boolean values to indicate if the query is syntactically valid and semantically valid.

The object diagram for a valid Query Object is shown below.

assign a; variable v;

Select a such that Uses (a, v) pattern a (v, _) with a.stmt# = 2

q:Query
declarations = {["a": "assign"], ["v": "variable"]}
toSelect = {"a"}
isSyntacticallyValid = true
isSemanticallyValid = true

:Clause
rel = "Uses"
args = {"a", "v"}
synonyms = {"a", "v"}
numOfKnown = 0

:Clause
rel = ""
args = {"a.stmt#", "2"}
synonyms = {"a"}
numOfKnown = 1

:Clause
rel = "a"
args = {"v", "_"}
synonyms = {"v"}
numOfKnown = 0
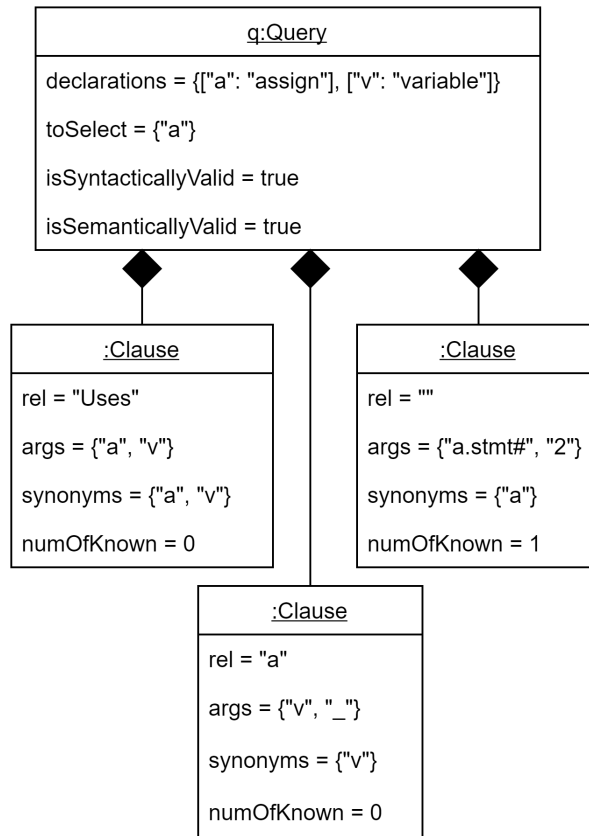
The object diagram for a syntactically invalid Query Object is shown below.

assign a;

Select a such that Follo (a, 2)

q:Query
declarations = {["a", "assign"])
toSelect = {"a"}
isSyntacticallyValid = false
isSemanticallyValid = true
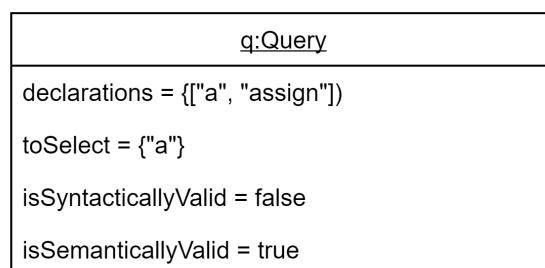
Note that if the query is syntactically invalid, the boolean value isSemanticallyValid is irrelevant. As isSyntacticallyValid is checked first by the Query Evaluator, the value in isSemanticallyValid will not have an effect on query evaluation.

Design Considerations

When designing the structure of a Query Object, we were faced with two choices. Below are the design considerations that we have made.

	Query Tree	Vector of clauses
Description	Each node on the tree represents a clause. Evaluate clauses starting from leaves up to the node. Restructure the tree for optimization.	Clauses are stored in a vector. Iterate through the vector to evaluate the clauses. Sort the clauses in the vector for optimization.
Complexity of Implementation	More complex as it requires Node classes and functions to access the tree and requires parsing and building of a tree before query evaluation begins.	Less complex, does not require building a tree. Putting objects in a vector is easier as we can utilise functions specified in the C++ vector API. Implementation can be better understood by the team.
Performance	Slower as tree traversal does not guarantee constant time for look up and might take up to $O(n)$ time for accessing a clause.	Accessing clauses in the vector using index takes $O(1)$ time and iterating through the vector takes $O(n)$ time, which is faster than tree traversal.
Memory Utilization	More memory required as tree nodes are stored in random places in memory. Each tree node contains a pointer to its child node. The storage of these pointers might result in overhead.	Less memory required as vectors use contiguous storage locations for their elements and there is no need for pointers.
Ease of extension	Harder to extend for future iterations and may require complicated algorithms for information retrieval. May need to restructure the tree for query optimization.	Easier to extend as we can utilise methods in the API for vectors. Easier for optimization as we can sort the clauses in the vector.

A vector of clauses was chosen to represent the queries as it is more memory and performance efficient, with less overhead. We prioritised performance for storing queries due to the speed requirement for query evaluations. A vector of clauses is also easier to implement and extend, while ease of extension is important as there may be other requirements in future iterations and the project has to be extended to accommodate more features.

In addition, we were faced with two choices for the validation of argument types for relationships. The following table shows the alternatives we have considered.

	Switch-case approach	Table-driven approach
Description	Have a case label for each relationship and the block of statements for each case label to validate each type of relationship.	An unordered map that maps each relationship to a vector of sets where the nth set contains all valid argument types of the nth arguments in the clause.
Complexity of Implementation	Although more intuitive, it may be more complex and less neat since different relationships have to be validated differently depending on the type of valid arguments.	Slightly easier to implement since validation for the different relationships are similar but compared with different valid arguments retrieved from the map.
Performance	Performance is likely to be similar.	Performance is likely to be similar.
Memory Utilization	Less memory required since no data structure is involved.	May require more memory to store the unordered maps.
Ease of extension	Harder to extend since we have to add more case values and their corresponding statements to support more relationships.	Easier to extend as we can only need to update the unordered map when more relationships need to be supported.

The table-driven approach was chosen since it is relatively simpler to implement and extend. This is important since the Query Preprocessor has to support the validation of many different types of relationships in the clauses and the project can be more easily extended to support more design abstractions. Memory utilisation is less of a concern and the tradeoff is justified since a table-driven approach is also likely to be neater and thus easier to debug.

Also, when designing the storage of clauses, we were faced with two choices. Below are the design considerations that we have made.

	Store such that, pattern and with clauses in separate vectors	Store such that, pattern and with clauses in a single vector
Description	One vector for all such that clauses, one vector for all pattern clauses and one vector for all with clauses	One vector for all different types of clauses
Complexity of Implementation	Relatively easier to implement since different types of clauses are stored separately	Slightly more complex since different clause type may have different number and types of arguments and we need a data structure that is able to support all three types of clauses
Performance	May be slower since it is unable to support optimization as well	Performance may be better since it is better able to support query optimization
Memory Utilization	Memory utilization is likely to be similar	Memory utilization is likely to be similar
Ease of extension	Less able to support query optimization since we can only compare and sort clauses of the same type	Better support for query optimization since we can compare and sort clauses regardless of their type

We have chosen to store all clauses in a single vector as it can better support query optimization. This is more important than ease of implementation since queries in iteration 2 and 3 are more complex with potentially more clauses and optimization is important to improve performance of query evaluation.

Conclusion

In summary, the Query Preprocessor takes in query strings and parses and validates the declarations, the synonym to select and the clauses. It then creates a Query Object that will be passed to the Query Evaluator for evaluation.

3.3.2 Query Optimizer

The Query Optimizer takes in the Query object, performs optimization and returns the optimized Query object for evaluation. Optimization is done by rewriting clauses using information in with clauses, dividing clauses into multiple groups, sorting groups for evaluation and sorting clauses inside each group.

Design

For each with clause with one known argument and one synonym, the Query Optimizer rewrites all other clauses that contain the synonym by replacing the synonym with the known value. For example, consider the with clause `a.stmt# = 2`, where a is of type assign. The synonym 'a' will be replaced with 2 in all other clauses. Note that this is only done for the synonym prog_line and attributes procedure.procName, variable.varName, constant.value, stmt.stmt#, read.stmt#, print.stmt#, call.stmt#, while.stmt#, if.stmt#, assign.stmt#. Attributes call.procName, read.varName, print.varName are ignored since there may be multiple values that satisfy the clause.

The Query Optimizer divides the clauses in the query into multiple groups by putting clauses without synonyms in one group and clauses with connected synonyms in the same group. An adjacency list is first created where there is an edge between synonyms in the same clause. The depth-first search algorithm is then used to group connected synonyms together and each group is assigned a group number. The Query Optimizer then groups the clauses by assigning clauses to their respective groups based on the group number of the synonyms in the clause.

To sort groups for evaluation, the Query Optimizer places the group without synonyms at the start and prioritizes groups that do not return results in Select. This increases efficiency as evaluation can be terminated immediately if clauses in these groups do not satisfy the conditions in the clause. Groups that will likely result in larger intermediate result tables will also not be computed. Note that after the evaluation of each group, the intermediate result table is projected to remove unnecessary columns (synonyms that are not in the result clause) before performing the join.

For sorting of clauses within each group, the Query Optimizer prioritizes clauses that are more restrictive so that there will be fewer results after these clauses are evaluated and the intermediate results table can be kept small. It first sorts the clauses by the number of synonyms in the clause that has already been computed and is in the result table, then by the number of known arguments in the clause, and lastly by the size of the relevant tables in the PKB. The size of the relevant tables in the PKB is determined as follows:

- Such that clause: size of corresponding table for the relationship
- Pattern clause: number of assign/while/if statements
- With clause:
    - Both arguments are unknown: choose the maximum between the number of procedures, variables, constants and statements
    - At least one argument is known: 0 (prioritized and evaluated first)

Design Considerations

We considered two choices for the implementation for the sorting of clauses within each group. Below are the design considerations that we have made.

	Static approach	Dynamic approach
Description	A priority value is assigned for each type of relationship and clauses are sorted based on the priority value assigned to the relationship in the clause	Clauses are sorted based on the size of the corresponding tables in the PKB
Complexity of implementation	More complex since we need to decide which type of relationship to prioritize	Slightly simpler since we simply query the size of the corresponding tables
Performance	Performance might not be as good as there may be exceptions where certain relationships which are usually small may result in large intermediate tables, and vice versa	Performance might be better as it can provide a better estimation of the size of the intermediate results table
Memory Utilization	Memory utilization is likely to be similar	Memory utilization is likely to be similar
Ease of extension	May be more difficult to extend since many priority values need to be assigned when many relationships need to be supported	May be easier to extend when many relationships need to be supported since we just have to query the size of the corresponding tables

We decided to adopt the second approach as we think that getting a more accurate estimation of the size of the intermediate result table returned by the clause and prioritising those clauses that result in smaller intermediate result tables can lead to faster evaluation. Besides, it is easier to implement and extend.

Conclusion

The Query Optimizer improves the efficiency of query evaluation by trying to keep the intermediate result table small.

3.3.3 Query Evaluator

The Query Evaluator receives the Query object and returns a list of strings containing the results.

Design

The Query Evaluator consists of the QueryEvaluator class, helper evaluators classes and the QueryUtility class. The QueryEvaluator class is the class in which Query objects from the Query Preprocessor are passed to. It is also the class which calls helper evaluators and joins results tables. Helper evaluators are evaluators to aid in the evaluation of specific clauses. The QueryUtility class is a class with methods that are commonly used by classes in the Query Processor. For exact details on the methods in the QueryUtility class, refer to Appendix 8.3.3.

Below is a list of helper evaluators.

Helper evaluator	Purpose
AffectsEvaluator	Evaluates Affects clauses
AffectsTEvaluator	Evaluates Affects* clauses
CallsEvaluator	Evaluates Calls clauses
CallsTEvaluator	Evaluates Calls* clauses
FollowsEvaluator	Evaluates Follows clauses
FollowsTEvaluator	Evaluates Follows* clauses
ModifiesEvaluator	Evaluates Modifies clauses
NextEvaluator	Evaluates Next clauses
NextTEvaluator	Evaluates Next* clauses
ParentEvaluator	Evaluates Parent clauses
ParentTEvaluator	Evaluates Parent* clauses
PatternEvaluator	Evaluates pattern clauses
UsesEvaluator	Evaluates Uses clauses
WithEvaluator	Evaluates With clauses

For each clause in the Query object, the Query Evaluator evaluates the clause by calling the relevant helper evaluators and returning an unordered map which maps synonyms in the clause to a vector of possible values for that synonym.

The helper evaluators check the lexical tokens in the arguments supplied to the clause and identify the types of arguments supplied by calling methods from the QueryUtility class. It then calls functions from the different objects located in the PKB to get design abstractions. Finally, it will scan the results obtained and remove all irrelevant results based on the synonym and its design entity.

For a clause with multiple synonyms as its arguments, an example of how the unordered map for the results obtained would look like is as shown below. Consider the query `assign a; while w; Select w such that Parent*(w, a)`. If the pairs <5, 6>, <5, 7>, <5, 8>, <5, 9> satisfy the clause where the first element in the pair is w and the second element is a, then the results are as shown in the table below.

w	a
5	6
5	7
5	8
5	9

The unordered map returned by the helper evaluator will map "w" to the vector { 5, 5, 5, 5 } and "a'' to the vector { 6, 7, 8, 9 }. Each column in the table is represented as an entry (key-value pair) in the map. When the table has more than one column, the vectors in the unordered map will have the same size. In this case, each vector in the unordered map has size 4.

After a clause is evaluated, it will return a result table back to the Query Evaluator. The Query Evaluator then performs a natural join (based on the common synonym) of the results obtained from the helper evaluators with the global result table using a hash join. If there are no common synonyms, a cross product is performed.

For example, consider the evaluation of the query `assign a; variable v; Select a such that Follows*(6, a) pattern a(v, _"cenX"_)`. Suppose the results of the Follows* and pattern condition are as shown in the tables on the left. In this example, the common synonym is "a". The join gives the result table on the right.

a
7
8

⋈

a	v
7	1
12	1
14	7

→

a	v
7	1
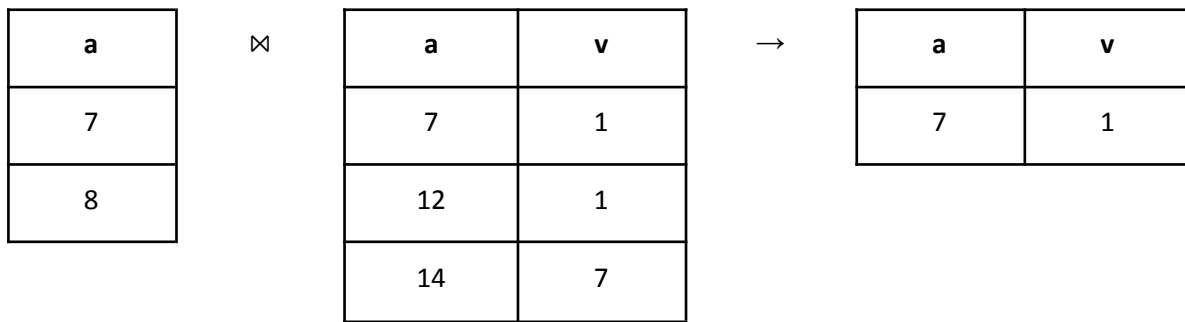
Since there are at most 2 synonyms in each clause, we split the merging of result tables into four cases:

- no common synonyms
- one common synonym and table size of clause result is one
- one common synonym and table size of clause result is two
- two common synonyms

The table merging algorithm in the case of one common synonym with a table size of one is as shown below. We first build a hash table with the common synonym. Then, we scan each row in the results table to search for matches. If a match is found, we insert that row into the newResults table.

```
} else if (commonSynonyms.size() == 1) {
     string commonSynonym = commonSynonyms.at(0);
     if (table.size() == 1) {
        unordered_set<string> set;
        for (int i = 0; i < tableNumRows; i++) {
           int val = table[commonSynonym].at(i); // value of commonSynonym in
table at row i
           string s = to_string(val);
           set.insert(s);
        }
        for (int i = 0; i < resultsNumRows; i++) {
           int val = results[commonSynonym].at(i); // value of commonSynonym
in results at row i
           string s = to_string(val);
           if (set.find(s) != set.end()) {
              // insert row i in results into newResults
              for (pair<string, vector<int>> col : results) {
                 newResults[col.first].push_back(col.second.at(i));
              }
           }
        }
     }
```

Finally, the Query Evaluator evaluates the synonym to select by retrieving the vector in the results table that maps to the synonym and removing duplicates. If the synonym is not present in the table and all clauses can be satisfied, the Query Evaluator returns all possible values for the design entity of the synonym.

For ambiguous query syntaxes not explicitly defined in PQL, we have decided to go with the following definitions.

Query	Reason for ambiguity	Result and explanation
stmt BOOLEAN; Select BOOLEAN	BOOLEAN has been declared as a synonym of type stmt, according to PQL this query can return either a boolean value or a list of statement numbers when we select it.	Since the query is ambiguous, we will treat it as semantically invalid. Since the query is a Select BOOLEAN query and is semantically invalid, we will return FALSE.
assign a; stmt BOOLEAN; Select <BOOLEAN, a> such that Affects (a, 11)	BOOLEAN should not be in a tuple so the query could be semantically invalid. However, BOOLEAN has been declared as a stmt so PQL can choose to treat it as a synonym.	Since BOOLEAN appears in a tuple and has been declared as a synonym of type statement, we will treat the query as syntactically and semantically valid. We will return pairs of statement numbers and assignment statement numbers where the assignment statement affects statement 11.
assign a; Select <BOOLEAN, a> such that Affects (a, 11)	BOOLEAN is a keyword and its use in a tuple is not well-defined in the PQL grammar rules.	Since BOOLEAN should not be used in a tuple and no synonym has been declared with the name BOOLEAN, the query is semantically invalid. We will return an empty result.
Select BOOLEAN with -5 = -5	As the argument types for the with clause does not follow PQL syntax and is also not one of the allowable types, it is unclear if the error should be treated as a syntactic or semantic error.	Since -5 is not a valid INTEGER according to the grammar rules, we treat the query as syntactically invalid.
Select BOOLEAN with "" = ""		Since an empty string is not a valid IDENT according to the grammar rules, we treat the query as syntactically invalid.
Select BOOLEAN with "covid19" = "covid 19"		Since IDENT should not contain spaces, the second argument is syntactically invalid and the query is treated as syntactically invalid.
variable v; constant c; Select BOOLEAN with v.value = c.value	A synonym of type variable should not have attribute value. However, this is not explicitly stated in the PQL grammar rules, hence it is unclear if the error should be treated as a syntactic or semantic error.	We have chosen to treat the query as semantically invalid.

stmt s; variable v; Select s with s.stmt# = v.varName	According to PQL grammar: attrCompare : ref '=' ref // the two refs must be of the same type (i.e., both strings or both integers). It is unclear whether the part in // is part of the grammar rules and should be treated as syntactic or semantic error.	We have chosen to treat the part in // as a semantic error. Since the refs are of different types, the query is treated as semantically invalid.
assign a; constant c; Select BOOLEAN with a = c.value	According to PQL grammar: ref : ""' IDENT ""'	INTEGER	attrRef	synonym // synonym must be of type 'prog_line' It is unclear whether the part in // is part of the grammar rules and should be treated as syntactic or semantic error.	We have chosen to treat the part in // as a semantic error. Since the synonym a is not of type prog_line, the query is treated as semantically invalid.
stmt s; Select s . stmt#	According to PQL grammar: attrRef : synonym '.' attrName. It is unclear whether there can be whitespaces before and after the '.'.	We have chosen to treat the query as valid.

Design Considerations

When designing the structure for the result table, we were faced with two choices. The table below shows the considerations we have made to arrive at the decision.

	Unordered Map	Vector of Pairs
Description	Each entry in the map maps the synonym (key) to a vector of possible values (value) for that synonym.	Each pair in the vector has the synonym as the first element and a vector of possible values for that synonym as the second element.
Complexity of Implementation	Relatively simple since we can utilise functions specified in the C++ unordered map API.	Relatively simple since we can utilise functions specified in the C++ vector API.
Performance	Better performance since maps have an average O(1) retrieval.	May have to iterate through the entire vector to retrieve the vector of values corresponding to a particular synonym.
Memory Utilization	Greater overhead as more memory is required for pointer manipulation between nodes.	Less memory required as vectors use contiguous storage locations for their elements.
Ease of extension	Relatively easy to extend as we can utilise methods in the API for unordered maps.	Relatively easy to extend as we can utilise methods in the API for vectors.

We chose to use unordered maps over vectors for storing our results as they have an average of O(1) retrieval time. This was the most important factor to us after taking into account future iterations of SPA. The length of the queries for future iterations of SPA might be longer, with more clauses in a 'Select' statement. As such, there might be a need to repeatedly retrieve results from the result table when joining the result table with the result table returned each clause. Memory utilization is less of a concern as it is not a requirement in the project.

Conclusion

The Query Evaluator consists of many subcomponents, where there is a main Query Evaluator class to call all other helper evaluators depending on the query type. All results returned from the helper evaluators will be stored in an unordered map for easy information retrieval.

3.3.4 Evaluation of Such That Clauses

The Query Evaluator will pass arguments to the respective helper evaluators for the evaluation of such that clauses.

Design

All helper evaluators for such that clauses follow the same general design and evaluation logic. We shall consider the Follows Evaluator here. When arguments are passed to the Follows Evaluator by the Query Evaluator, it will first find the argument types in the clause by calling `getArgType()` from the Query Utility. The argument types for the clause will enable it to decide on how to proceed with the evaluation. For the query `assign a; Select a such that Follows*(a, 3)`, it will classify `a` as an 'assign' type and `3` as an integer type.

After classifying the arguments, it will get a list of statements that comes before statement 3 in the statement list by calling `getFolloweeStar("3")` from the Follows object in the PKB. It will then check and remove statement numbers that are not assignment statements by calling `intersectSingleSynonym()` from the Query Utility on the statement list returned by `getFolloweeStar("3")` and all assignment statements returned by `selectAll("assign")`. The relevant statement numbers will then be stored into the unordered map, `tempResults`. In the map, `a` will be mapped to a vector of integers representing the statement numbers that satisfy the clause.

Below is a sequence diagram on how the FollowsEvaluator gets all assignment statements before statement 3 in the statement list.

Design Considerations

We considered two choices for the implementation of such that clauses. Below are the design considerations that we have made.

	General evaluator class	Helper evaluator classes
Description	The general evaluator class will handle all relationship queries	A main evaluator class will call different helper evaluator classes to handle queries for each relationship
Complexity of Implementation	May be slightly more complex since the class has to evaluate all types of clause	May be slightly simpler since each evaluator class is only responsible for the evaluation of one type of clause
Performance	May be slightly faster since less functions calls are likely to be made	May be slightly slower since there will be more function calls
Memory Utilization	Memory utilization is likely to be similar	Memory utilization is likely to be similar
Ease of extension	Slightly more difficult to extend as each additional relationship that can be queried will result in many changes to the general evaluator class	Easier to debug and implement additional features for each relationship as the classes all handle a different relationship

We decided to adopt the second approach as we believe that by the Single Responsibility Principle, it will be much easier to debug and implement additional features for each relationship if we separate it into different classes.

Conclusion

Such that clauses are evaluated by different evaluators based on their relationship, and these evaluators generally follow the same design and way of evaluation.

3.3.5 Evaluation of Pattern Clauses

The Query Evaluator will pass arguments to the Pattern Evaluator for the evaluation of pattern clauses.

Design

The design and evaluation logic for if and while pattern clauses is similar to such that clauses, albeit simpler as both clauses take in fewer arguments since the second and third arguments are of type 'underscore'. As such, we shall only be considering how assign pattern clauses are evaluated here. When arguments are passed to the Pattern Evaluator, it will first classify the arguments in the clause, then call the relevant helper functions to evaluate the clause. For example, when the Pattern Evaluator encounters the assign pattern clause `a(v, _"x"_)`, it will classify the first argument as a variable, and the second argument as an expression with underscore. The helper function `evaluateVarExpressionWithUnderscore()` will be called.

Below is an activity diagram of how the Query Evaluator interacts with the PKB for the evaluation of assign pattern clauses.
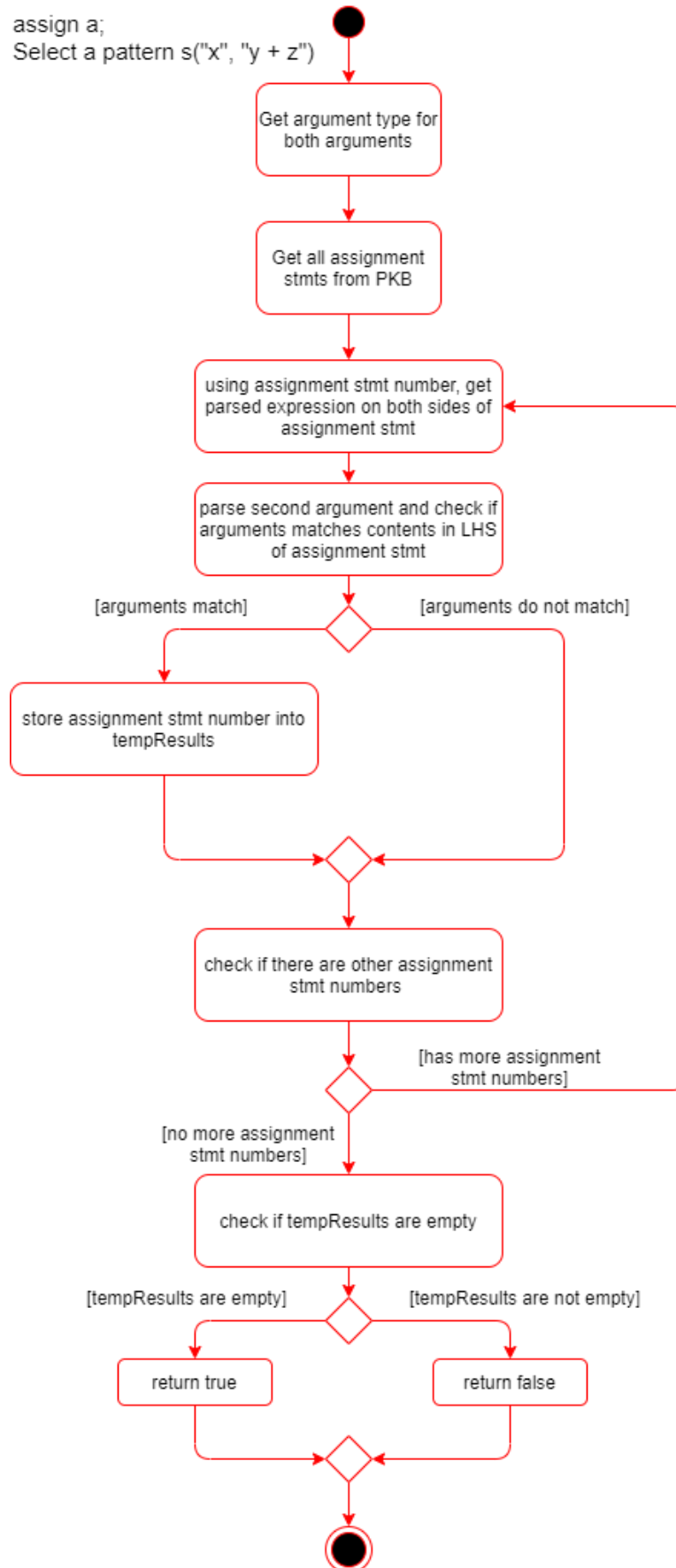
The activity diagram starts after the Query Evaluator has called `evaluate()` in the Pattern Evaluator. The Pattern Evaluator calls `getAllAssignStmtNums()` from the Statement Table in the PKB to get a vector of all assignment statement numbers. For each assignment statement number, it then calls `getAssignExpr()` from the statement table in the PKB to get the parsed contents of the statement and uses it to compare with the arguments supplied to the pattern clause, storing all relevant statement numbers into an unordered map, `tempResults`.

In this example, the arguments in the clause contain expressions. Hence the Pattern Evaluator will use functions from the Source Processor to add brackets to the expression such that it is left-associative. For instance, the expression "`x + y + z`" will be converted to "`((x + y) + z)`". The Pattern Evaluator will then be able to compare this expression to other expressions stored in the PKB. The expressions in the PKB would have been stored with added brackets. To check if the right-hand side of an assignment statement contains a sub-expression, the Pattern Evaluator will convert the sub-expression to its left-associative version, and check if it is a substring of the right-hand side of the assignment statement.

Design Considerations

We considered two choices for the implementation of pattern matching. Below are the design considerations that we have made.

	AST traversal	String matching
Description	For each expression in the LHS of an assign-pattern query, traverse all the ASTs stored in the PKB	Add brackets to all assignment statements such that it is left-associative and store the strings in the PKB Add brackets to the expression in the LHS of an assign-pattern query Compare the expressions by matching strings
Complexity of Implementation	Slightly more complex since an algorithm for tree comparison is required	Relatively easier to implement since only string comparison is required
Performance	Depending on the algorithm used, may be slower than tree comparison	Likely to be more efficient since string comparison takes O(n) time
Memory Utilization	More memory may be required as pointers to the tree nodes need to be stored, resulting in greater overhead	Memory utilisation is likely to be lower since pointers need not be stored
Ease of extension	No significant difference in ease of extension	No significant difference in ease of extension

We decided on string matching as it is comparatively easier to implement and debug, is more efficient and utilises less memory compared to AST traversal.

Conclusion

The Pattern Evaluator evaluates assign pattern clauses by converting it to its left-associative expression with brackets and comparing it with other left-associative expressions with brackets derived from the PKB.

3.3.6 Evaluation of With Clauses

The Query Evaluator will pass arguments to the With Evaluator for the evaluation of with clauses.

Design

The With Evaluator first finds the argument types and attribute names of arguments in the clause by calling `getArgumentMap()`, which returns a map of the argument type, the attribute name (if it exists) and the synonym/integer/name in the argument. This enables the evaluator to decide on how to proceed with the evaluation. If both arguments are known (integer/name), the With Evaluator checks if their values are equal.

If the attribute values for both arguments are names (varName/procName/name), the With Evaluator compares the names of the two arguments. If one of the arguments is known, the With Evaluator calls relevant PKB methods using the known name argument to get all values for the unknown argument. The following table shows the PKB methods called in the evaluation of the attrRef which are names and with one argument known.

attrRef	PKB methods
procedure.procName	PKB::procTable->getProcID(name);
call.procName	PKB::procTable->getProcID(name) PKB::calls->getStmtNumThatCallsCallee(procId)
variable.varName	PKB::varTable->getVarID(name)
read.varName	PKB::varTable->getVarID(name) PKB::stmtTable->getStmtNumsOfReadWithVar(varId)
print.varName	PKB::varTable->getVarID(name) PKB::stmtTable->getStmtNumsOfPrintWithVar(varId)

If both arguments are unknown, i.e. they are attrRef and have the structure synonym '.' attrName, the With Evaluator retrieves all possible values for each synonym and gets the name for each possible synonym value. The attribute values in both arguments are then compared and added to the results if their values are equal. The following table shows the PKB methods called in the evaluation of the attrRef which are names and with both arguments unknown.

attrRef	PKB methods
procedure.procName	PKB::procTable->getProcName(num)
call.procName	PKB::calls->getCalleeInStmt(num) PKB::procTable->getProcName(procCalled)
variable.varName	PKB::varTable->getVarName(num)
read.varName	PKB::stmtTable->getReadVariableOfStmt(num) PKB::varTable->getVarName(varRead)
print.varName	PKB::stmtTable->getPrintVariableOfStmt(num) PKB::varTable->getVarName(varPrint)

If the attribute values for both arguments are integer (stmt#/value/integer), the With Evaluator gets all values for the synonym or integer. In this case, the attrName is ignored since the value of the attrRef (synonym '.' attrName) and the value of the synonym without the attrName are the same. The With Evaluator then performs an intersection with the two sets of values and checks if the intersection is non-empty. The synonyms will then be mapped to the values in the intersection and stored into the unordered map, `tempResults.`

For example, consider the evaluation of the with clause `p.procName = "computeCentriod"` as shown in the sequence diagram below. Suppose `p` has been declared with type 'procedure'. The With Evaluator first gets the argument types and attribute names of the arguments. In this case, the first argument p.procName will have argument 'p', argument type 'procedure' and attribute name 'procName', and the second argument `"computeCentroid"` has argument 'computeCentroid', argument type 'name' and no attribute name. Since the first argument is an attrRef and the second argument is a name, the With Evaluator then calls `getProcID("computeCentroid")` from the procTable object in the PKB to get the procedure ID of the procedure with name `"computeCentroid"`. Suppose that the procedure ID is 0. The With Evaluator maps the synonym `"p"` to a vector containing 0, stores the results into the unordered map, `tempResults`, and returns true to indicate that the clause can be satisfied.

```
p.procName = "computeCentroid".
```
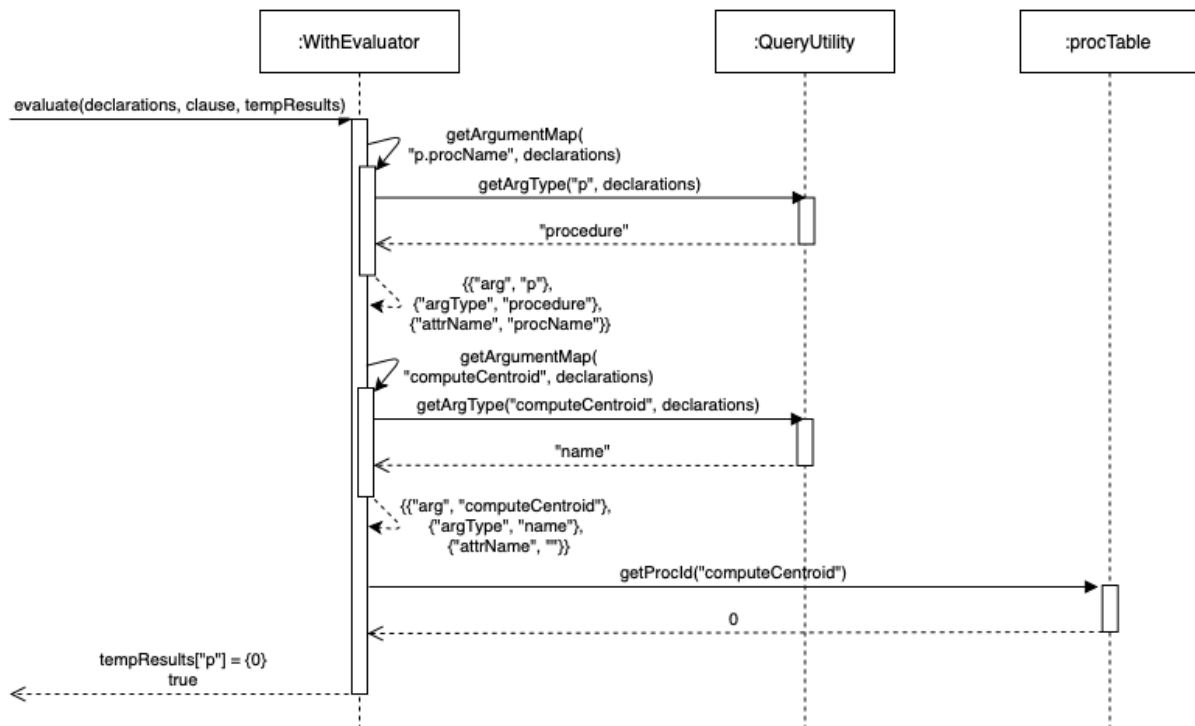

Design Considerations

We considered two choices in the evaluation of with clauses in the case where one of the arguments is known. Below are the design considerations that we have made.

	Get all values for unknown synonym	Query PKB
Description	Retrieve all values for the unknown synonym and for each synonym, get the attribute value for the attribute name. Compare the attribute value with the known argument.	Call API methods from the PKB to get the synonyms with attribute values equal to the known argument
Complexity of Implementation	Complexity of implementation is likely to be similar	Complexity of implementation is likely to be similar
Performance	Will be slower as it takes $O(n)$ time to iterate and compare through all values stored in the PKB for a particular synonym	Will be faster as PKB stores synonyms and their attribute values in a hash map so retrieval time is $O(1)$
Memory Utilization	Less memory required since no additional data structure is needed	More memory may be required to store the maps in the PKB
Ease of extension	No significant difference in ease of extension	No significant difference in ease of extension

We decided to query the PKB as performance may be better since the information can be precomputed and the PKB can store them in maps. Thus, it may not be necessary to iterate through all possible values for the synonym.

Conclusion

The With Evaluator evaluates with clauses by getting the argument types and attribute names (if any) of the arguments in the clause and calling relevant methods from the PKB.

4 Testing

This section describes the testing plan we undertook during Iteration 1. We had decided on writing unit tests, integration tests and system tests to test our program.

Weeks 1 - 3

We were still in the midst of implementing the various components and hence could only write unit tests for our own components. The unit tests are written using Catch2 and GitHub Actions is used to automate the testing process by running the executable created, while GitHub Issues is used to keep track of bugs we detected while testing the program.

Weeks 4 - 5

We had each finished our own components and thus begun integration testing. We focused on the integration between SP and DE, DE and PKB, QP and PKB for integration tests as these are the components with the most component communications. All integrations tests are written using Catch2, with GitHub actions acting as an automated tester. We also wrote a few simple system tests to check if all components can integrate together smoothly.

Week 6

We started on more complex system tests, writing different source codes to parse and different combinations of queries to test the code.

Weeks 7 - 8

During this time period we worked on designing and implementing many Iteration 2 features, namely: the parsing, validation and processing of call statements & multiple procedures; the population and querying of the C/C* and N/N* relationships and container pattern information; the parsing of queries containing multiple clauses, the 'and' keyword, BOOLEAN and tuples, the new prog_line, call synonyms, new pattern clauses, attributes and with-clauses; table merging correctness. For all these features, comprehensive unit and integration tests were written. Due to the workload, we decided to leave system tests for later.

Weeks 9 - 10

Our focus was mainly on completing the population and querying of A/A*. Time was also spent on finishing up features from the previous week, namely N/N*, with-clauses, synonym attributes and table merging correctness. We wrote basic and complex system tests for the new features we plan to demo for Iteration 2, as well as tests for cyclical procedural calls.

4.1 Unit Testing

We mainly adopted specification based (black-box) test design technique for our unit test cases. This is because the SPA requirements are well-defined and we find that black-box testing is the most useful. There are a few test cases that use the white-box approach to test the inner implementations of certain methods.

4.1.1 Unit Test Cases for SP

Testing for SP focuses on syntax checking and parsing.

Positive test cases would use the generated AST as a proxy to check if parsing is done correctly. Correctness is checked by ensuring contents of the AST matches what is expected, e.g. number of child AST nodes, type of nodes, value of nodes, structure of AST etc.

Positive test cases checks:
- Check expressions to see if it has the correct left-associativity defined by CSG (Concrete Syntax Grammar)
- Check expressions for correct operator precedence, e.g. brackets take higher precedence, then *, /, %, before +, -
- Keywords used as var_names: eg. `print call;`, `read = x + 1;` etc.
- Keywords used as proc_names: eg. `procedure procedure { }`
- Parses nested statements in container statements
    - check for correct number of child Nodes in If/While node
    - check for correct Stmt nodes
    - container statements numbered before nested statements
- Correct syntax used for program | program | stmt

Negative test cases check if Parser would reject SIMPLE source code with syntax errors. Correctness is determined by a negative test case causing a sp::ParserException to be thrown which occurs when there are syntax errors or when methods return a false.

The following are some of the invalid inputs used:

- Invalid expression syntax: e.g. `x = 5 +* a;`

- Improper parenthesis pairings in expressions

- Container Statement with invalid cond_expr: e.g. `while ((a > b) && (c != d) || (g <= h) ) {}`

- Syntax error eg `if ( a > b ) { … } else { … }` missing 'then'

- Missing semicolon after assign | call | print | read statements

- Invalid lexical tokens eg `&`, `098`, `1name` etc

4.1.2 Unit Test Cases for DE

Unit testing for the DE is done by calling various methods from its public API, which is meant to simulate the various statements in a SIMPLE program. Tests are written for both valid and invalid relationships and entities, with an emphasis on valid test cases. The PKB is directly used to check correctness, because the parts involved are not logic-heavy. However, dummy ASTs are created and passed to the DE during all API calls to avoid using the Parser's functionality.

Below is an example of a DE unit test case. The test queries the PKB after DE's processing to check that all program design entities and abstractions have been correctly extracted and stored.

```
ReadStmt *readStmt =  new ast::ReadStmt(1, new
sp::Token(sp::Token::TokenType::READ, "r"), varName);

TEST_CASE("storeNewRead Test") {
    DesignExtractor::signalReset();

    DesignExtractor::storeNewRead(1, "procedure", readStmt);

    // Check varName
    ID varID = PKB::varTable->getVarID("procedure");
    REQUIRE(varID == 0);     // varTable ID starts indexing at 0
    REQUIRE(PKB::varTable->getVarName(varID) == "procedure");

    // Check Modifies
    REQUIRE(PKB::modifies->getStmtsModifies(varID) == unordered_set<StmtNum>{ 1
}); // StmtNums start from 1
}
```

4.1.3 Unit Test Cases for PKB

There are positive/negative test cases, for instance, when trying to retrieve the variable ID of a variable name, we will test cases when the name exists in VarTable and when the name doesn't exist in the VarTable, in which case an error will be thrown.

In some test cases, we have used dependency injection, such as those in TestStmtTable.cpp. StmtTable depends on statement node classes from the parser. To isolate the system under test, we have created a StmtNodeStub. An example of a test case in TestStmtTable is shown below.

```
class StmtNodeStub : public ast::Stmt {
public:
   StmtNodeStub(int index): ast::Stmt(new sp::Token(), index){};
};

TEST_CASE("storeStmt Test") {
   StmtTable* stmtTable = new StmtTable();
   ast::Stmt* stmtNodeStub1 = new StmtNodeStub(1);
   ast::Stmt* stmtNodeStub2 = new StmtNodeStub(2);
   REQUIRE(stmtTable->storeStmt(1, stmtNodeStub1, "assign"));
   // Check that stmtNum is also stored in AssignStmtNums
   vector<StmtNum> const &assignStmtNums = stmtTable->getAllAssignStmtNums();
         REQUIRE(find(assignStmtNums.begin(),   assignStmtNums.end(),   1)   !=
assignStmtNums.end());

   REQUIRE(stmtTable->storeStmt(2, stmtNodeStub2, "read"));
   // Check that stmtNum is also stored in WhileStmtNums
   vector<StmtNum> const &readStmtNums = stmtTable->getAllReadStmtNums();
         REQUIRE(find(readStmtNums.begin(),   readStmtNums.end(),   2)   !=
readStmtNums.end());

   // Check that all StmtNums has both stmt 1 and stmt2
   vector<StmtNum> const &stmtNums = stmtTable->getAllStmtNums();
      REQUIRE_THAT(stmtNums,  Catch::Matchers::UnorderedEquals(vector<StmtNum>{1,
2}));

   // stmtNum 1 already exists
   REQUIRE_FALSE(stmtTable->storeStmt(1, stmtNodeStub1, "assign"));
}
```

White-box testing is used here, as we have designed this test case with the implementation in mind. For instance, when storing an assign statement, we check that it is stored into the vector 'AssignStmtNums' as well as 'StmtNums'. We also check that we are not able to store a statement that has already been stored before.

We also have algorithmic tests for PKB. For the methods populateParentStar() and populateFollowsStar(), populateNextStar() and populateAffectsAndAffectsStar(), algorithmic tests are used to check that we have computed and stored the Parent*, Follows*, Next*, Affects and Affects* relationships correctly. For each of the unit tests, we will come up with a simple source program with certain desired characteristics. Below is the test case for populateParentStar(). When we set up the

parent test, the source program we have in mind have nested while loops, so that we can test for the Parent* relationship.

```
Parent* setUpParentTest() {
   Parent* parent = new Parent();
   parent->storeParent(1,2); // 1. while (...) {
   parent->storeParent(2,3); // 2.    while (...) {
   parent->storeParent(3,4); // 3.       while (...) {
   parent->storeParent(2,5); // 4.          b = 2; }
   parent->storeParent(1,6); // 5.       c = 3; }
   parent->storeParent(1,7); // 6.    d = 5;
   return parent;            // 7.    e = 6; }
}

TEST_CASE("populateParentStar Test") {
   Parent* parent = setUpParentTest();
   parent->populateParentStar();
   REQUIRE(parent->getChildrenStar(1) == unordered_set<StmtNum>({2,3,4,5,6,7}));
   REQUIRE(parent->getChildrenStar(2) == unordered_set<StmtNum>({3,4,5}));
   REQUIRE(parent->getChildrenStar(3) == unordered_set<StmtNum>({4}));
   REQUIRE(parent->getChildrenStar(7).empty());
   REQUIRE(parent->getParentStar(1).empty());
   REQUIRE(parent->getParentStar(2) == unordered_set<StmtNum>({1}));
   REQUIRE(parent->getParentStar(3) == unordered_set<StmtNum>({1,2}));
   REQUIRE(parent->getParentStar(4) == unordered_set<StmtNum>({1,2,3}));
   REQUIRE(parent->getParentStar(5) == unordered_set<StmtNum>({1,2}));
   REQUIRE(parent->getParentStar(6) == unordered_set<StmtNum>({1}));
   REQUIRE(parent->getParentStar(7) == unordered_set<StmtNum>({1}));
}
```

The populateParentStar test is the last test case of our file TestParent.cpp. We have already performed the tests for storeParent(), getChildrenStar() and getParentStar(). Thus, in the populateParentStar() test, we used a helper method setUpParentTest() to populate valid Parent relationships. Then, we run populateParentStar() and use the getters to check that we have populated the Parent* relationships correctly. We also perform boundary tests and check that the last statement has no children and the first statement has no parent, thus returning an empty set.

4.1.4 Unit Test Cases for QP

For the Query Preprocessor, tests are written for both valid and invalid queries whereas for the Query Evaluator, tests were written for queries with empty and non-empty results. Testing of different combinations of arguments in clauses is done in the individual Evaluator for the different design abstractions.

For the Query Preprocessor with valid inputs, we tested for:
-   query with select BOOLEAN
-   query with select single element
-   query with select tuple

- query with attrRef in select

- query with no such that and no pattern clause

- query with comma in the declaration

- query with one such that clause

- query with one pattern clause

- query with one with clause

- query with multiple clauses

- query with 'and' keyword

For the Query Preprocessor with invalid inputs, we tested for:

- empty query

- query with select ending with semicolon

- query with missing select

- query with multiple select

- query with declarations after select

- query with invalid design entity in declaration

- query with invalid synonym in declaration

- query with synonym not declared

- query with invalid attrRef

- query with syntactically or semantically invalid such that clause

- query with syntactically or semantically invalid pattern clause

- query with syntactically or semantically invalid with clause

For the Query Evaluator, we tested for:

- syntactically or semantically invalid query

- query with no clauses

- query with select BOOLEAN

- query with select single element

- query with select tuple

- query with attrRef in select

- query with one such that clause and the synonym to select is in the clause

- query with one such that clause and the synonym to select is not in the clause

- query with one pattern clause

- query with one with clause

- query with multiple clauses

One of the test cases for the Query Preprocessor is to test for the processing of queries with invalid such that clause.

```
TEST_CASE("process invalid such that clause") {
   QueryPreprocessor qp = QueryPreprocessor();
   string query = "assign a; stmt s; \nSelect a such that Uses (a, s)";
   Query actual = qp.process(query);
   unordered_map<string, string> declarations;
   declarations["a"] = "assign";
   declarations["s"] = "stmt";
   Query expected = Query(declarations, "a", {}, false);
   REQUIRE(actual == expected);

    query = "assign a; while w; \nSelect w such that Parent* (w, a) and pattern a
(\"count\", _)";
   actual = qp.process(query);
   unordered_map<string, string> declarations1;
   declarations1["a"] = "assign";
   declarations1["w"] = "while";
   expected = Query(declarations1, "w", {}, false);
   REQUIRE(actual == expected);
}
```

In the first test case, the query to process is

```
assign a; stmt s;
Select a such that Uses (a, s)
```

There is a semantic error in the query since the second argument of Uses should only be a variable and not a statement. Since the declarations are valid, the declarations will be successfully stored in the unordered map in the query object. The synonym to select has also been declared and will be stored in the second attribute of the Query object. The 'such that' clause is invalid and will not be stored in the Query object. When the Query Preprocessor encounters the invalid 'such that' clause, it will also set the isValid boolean in the Query object to false.

In the second test case, the query to process is

```
assign a; while w;
Select w such that Parent* (w, a) and pattern a (\"count\", _)
```

The query is invalid as there is an additional "and" between the such that and pattern clauses. Similar to the first test case, since both the declarations and the synonym to select are valid, they will be stored in the Query object. The 'such that' clause is invalid and will not be stored in the Query object and the isValid boolean in the Query object will be set to false.

One of the test cases for the Query Evaluator is to test the evaluation of queries with one such that clause and one pattern clause. To test the evaluation of the query, we have to first populate the PKB with design entities and design abstractions, as shown in the setupQe() function, which can be found in Appendix 8.6.

```
TEST_CASE("process valid query with such that and pattern clause") {
    QueryPreprocessor qp = QueryPreprocessor();
     string query = "assign a; while w;\nSelect a pattern a (\"x\", _) such that
Uses (a, \"x\")";
    Query actual = qp.process(query);
    Clause c1 = Clause("Uses", { "a", "\"x\"" });
    Clause c2 = Clause("a", { "\"x\"", "_" });
    unordered_map<string, string> declarations;
    declarations["a"] = "assign";
    declarations["w"] = "while";
    Query expected = Query(declarations, "a", { c2, c1 }, true);
    REQUIRE(actual == expected);

     query = "assign a; while w;\nSelect a such that Uses (a, \"x\") pattern a
(\"x\", _) ";
    actual = qp.process(query);
    c1 = Clause("Uses", { "a", "\"x\"" });
    c2 = Clause("a", { "\"x\"", "_" });
    unordered_map<string, string> declarations1;
    declarations1["a"] = "assign";
    declarations1["w"] = "while";
    expected = Query(declarations1, "a", { c1, c2 }, true);
    REQUIRE(actual == expected);
}
```

In the first test case, the query to evaluate is

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

In the second test case, the query to evaluate is

```
assign a; while w;
Select w such that Parent* (w, a) pattern a("count", _)
```

The evaluate function takes in the Query object that represents the query and returns a list of strings containing the results of the synonym to select with all clauses satisfied. Since order does not matter, we convert the list into a set and compare with the expected results.

4.2 Integration Testing

4.2.1 Integration test cases for Parser and DE

The test cases involve the Parser parsing SIMPLE source code. A snippet of an integration test can be found below, showing what the Parser takes in as input. Then in the parse() method, the Parser creates and traverses the resulting AST, calling the DE at every statement.

```
    string input = "procedure star {x = 1 + 2 * a; while (v > 1) {x = 2; if
(v==1)then{c = a;}else{b = 2;}} print p; read s;} ";
    std::vector<sp::Token> actual_tok;
    std::vector<sp::Token*> tok_ptrs;
    ParserUtils::StringToTokenPtrs(input, actual_tok, tok_ptrs);
    auto l = new LexerStub(tok_ptrs);
    auto p = Parser(l);

    REQUIRE(p.parse()); // no errors
```

Then, the PKB is checked to see if the correct program design abstractions and entities were stored. We make use of the PKB in this test as the parts of the PKB involved is not logic-heavy, and stubbing the many tables in the PKB would be tedious. Both valid and invalid inputs were tested, with the negative inputs covering a range of syntactic and semantic errors.

4.2.2 Integration test cases for DE and PKB

Since the DE assumes that all inputs are valid, the test cases involve processing correct inputs and checking if the program design abstractions and entities were extracted and stored into the PKB properly. The focus was on positive test cases to ensure that all relevant information was extracted and stored correctly, but negative test cases were also included.

4.2.3 Integration test cases for PKB and QP

We mainly focused on positive test cases so that the Query Processor can correctly retrieve information stored in PKB. We then compare the results acquired with our expected results to ensure that the information retrieved from PKB is correct. As in the unit tests for PKB and QP, we have created a StmtNodeStub to populate PKB's StmtTable.

Below is an example of a test case created for a 'Uses' clause with a synonym and a variable name in quotes, where `setupQp()` is a function to populate the tables in PKB.

```
    setupQp();

    QueryPreprocessor qp = QueryPreprocessor();
    QueryEvaluator qe = QueryEvaluator();

    string query7 = "assign stmt; Select stmt such that Uses (stmt, \"count\")";
    Query q7 = qp.process(query7);
    list<string> list7 = qe.evaluate(q7);
    unordered_set<string> actual7(begin(list7), end(list7));
    unordered_set<string> expected7 = { "6", "12", "13" };
    REQUIRE(actual7.size() == list7.size());
    REQUIRE(actual7 == expected7);
```

Below is a sequence diagram of how the Query Evaluator interacts with the PKB to evaluate the query above.
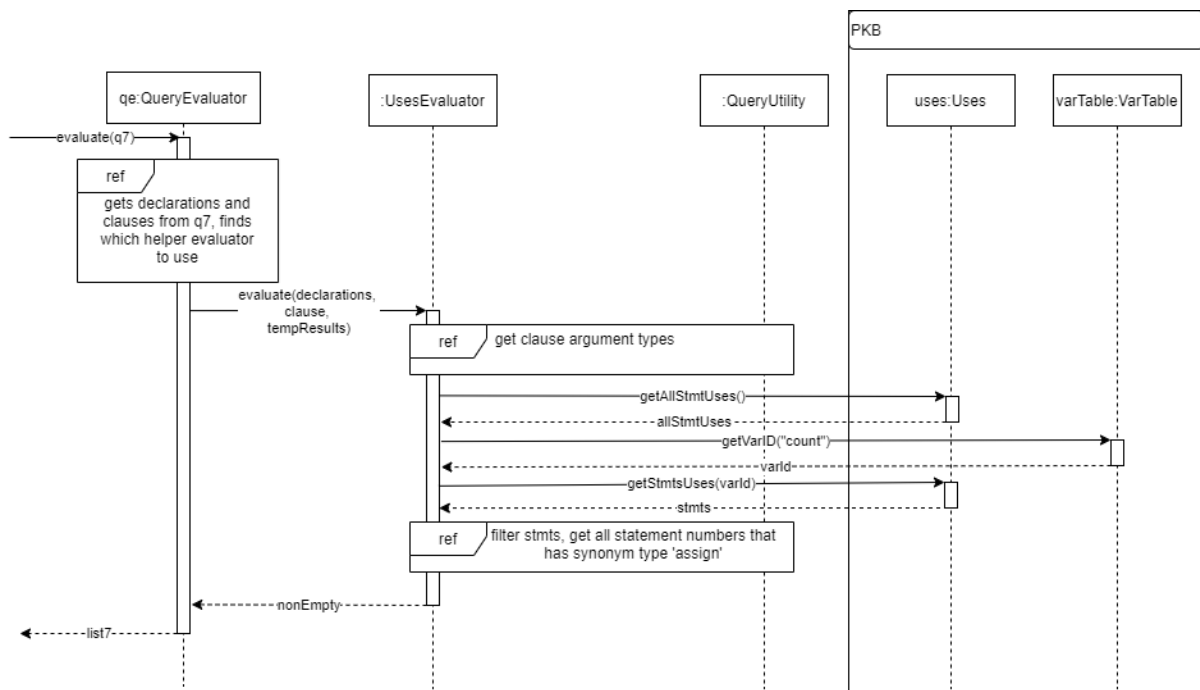
4.3 System Testing

System tests are written to test the correctness of the SPA. We test for valid and invalid queries and valid and invalid SIMPLE source programs.

4.3.1 SIMPLE source programs

We have 3 variations of SIMPLE source program: basic, complex and invalid. Below is a summary of their characteristics. Some of these source programs can be found in Appendix 8.5.

Type of SIMPLE program	Characteristics
Simple	- Non-nested if/while statements - Simple expressions - Relational expression for conditional expression
Complex	- Multiple nesting levels - Different combinations of nesting - Using design entities as variable names and procedure names - Complex conditional expressions using relational expressions and conditional operators such as !, && and || - Complex assignment expressions
Invalid	Cyclical call

All the source programs contain at least one of each design entity and a few procedures, variables and constants. Queries are written for these programs and for valid source programs, we ensure that almost non-trivial number of rows are returned for the tables of each clause for queries with multiple clauses. Thus, we tried to include as many combinations of relationships there are between the different design entities as possible in the program. For example, considering Next* relationship, we tried to have multiple statements executed immediately after one another. We would also try to have different combination of nesting and different level of nesting to form different structures of CFG for Next*. We then tried to have Next* relationships between the different types of design entities.

We designed one basic SIMPLE source program to test if our system is able to deal with the basic grammar rules of SIMPLE. If we fail to parse, we are able to quickly identify the errors and rectify it. The complex SIMPLE sources are then used to test if our system is able to parse and validate a more complicated source code. We also have a source that is made up of mostly keywords such as "procedure", "read" and etc. to test if our parser is able to distinguish between keywords and variable or procedure names.

Lastly, the invalid SIMPLE source is used to test if our system is able to correctly handle programs that do not follow grammar rules. We also ensure that errors such as cyclic calls are identified. Errors identified will be printed as messages if possible and the SPA will terminate.

4.3.2 Queries

We wrote both valid and invalid queries. The following table gives a summary of what we tested.

Query type	Characteristics
Invalid query	- Missing declarations - Invalid arguments for each different type of clause
Query with no clause (no join)	Different combinations of synonyms
Query with single clause (no join)	0, 1 or 2 synonyms in clause - Select single synonym (used or unused in clause) - Different combinations of types of synonyms - Select tuples (2 and more synonyms) - Select BOOLEAN - Select attributes - Wildcards - Whitespaces - Long synonym names - Unused synonym - Return no result, single result, multiple result
Query with multiple clauses (multiple joins)	0, 1 or 2 synonyms in each clause 0, 1 or 2 common synonyms between each clause - Select single synonym - Select tuples (2 and more synonyms) - Select BOOLEAN - Select attributes - Wildcards

For valid queries, we mainly test the ability to:

1. Retrieve correct information from PKB
2. Join the tables in multiple clauses
3. Project the correct results in Select

Queries with no join are mainly to test point (1). This ensures that the SPA can correctly evaluate the different kinds of design abstraction and pattern individually. We can then put less focus on the type of clauses used for queries with multiple clauses. Nevertheless, we still ensure that the clauses used are as diverse as possible.

For example, we design queries that return no result, single results and multiple results to ensure that the SPA is able to evaluate queries returning different types of results correctly. We also design queries with different permutations of the arguments in the clause. For example, for Follows clauses, we consider the different permutations as shown below.

_, _	_, INTEGER	_, synonym
INTEGER, _	INTEGER, INTEGER	INTEGER, synonym
synonym, _	synonym, INTEGER	synonym, synonym

Queries with multiple joins test for point (2) and (3). We will also test if the SPA is able to handle the different combinations of groups of clauses, where groups are formed by clauses with common, connected synonyms.

For invalid queries, we test for the ability to:

1. Detect syntactic errors
2. Detect semantic errors

Queries with syntax errors are not supposed to return any results. However, queries with semantic errors will be checked further to see if it returns FALSE for BOOLEAN result. Otherwise, it is supposed to return nothing for other types of Select clauses.

Some examples are shown below.

Query	Purpose
1 - No clause call call c; Select c 3, 12, 20, 24, 52, 68, 84, 85, 94, 98 5000	This is a no clause query. We test to see if all calls are properly stored and retrieved from PKB.
17 - while next prog_line n; Select n such that Next*(3, n) 3,4,5,6,7,8,9,10,11,12 5000 38 - Next*(a, _) assign a; Select a such that Next*(a, _) 2, 4, 5, 7, 8, 16, 17, 18, 22, 25 5000 42 - Get all assign after call call c; assign a; Select a such that Next*(c, a) 16, 17, 18, 21 5000	We test different combinations of arguments. This is also done for other relationships and patterns. We mainly do these combinations in single clause queries.

| ```
3 - boolean (clause return table)
stmt s;
Select BOOLEAN such that Follows(s, 42)
TRUE
5000
16 - exact, synonym
procedure p; read r; assign a; print
pr; constant c; variable v;
Select r such that Calls("simple", p)
1, 13, 34, 61, 67, 76, 89, 97
5000
9
call c; assign a; if ifs;
Select <c, a, ifs> such that Next*(a,
c) and Next*(ifs, a)
11 8 6, 12 8 6, 11 7 6, 12 7 6
5000
``` | We test for different kinds of results such as tuples, BOOLEAN, synonyms that do not overlap with synonyms in clauses. |
| ```
12 - multiple groups (2 groups)
assign a; variable v; stmt s; read r;
while w;
Select <a, r> pattern a(v, _"red"_)
with a.stmt# = s.stmt# such that
Modifies(s, v) such that Next*(r, r)
and Parent(w, r)
25 13,25 89,33 13,33 89,41 13,41 89,50
13,50 89,57 13,57 89,65 13,65 89,70
13,70 89
5000
``` | We test for queries with multiple different groups. |
| ```
10 - if pattern, invalid
if ifs;
Select ifs pattern ifs("green", _,
"red")
none
5000
1 - wildcard, wildcard

Select BOOLEAN such that Uses(_, _)
FALSE
5000
``` | We test for syntactic and semantic errors in queries. |

4.4 Stress Testing

System tests are written to test the efficiency of the SPA. Firstly, we wanted to ensure that the parsing of SIMPLE source program and populating of PKB is efficient. The source has the following characteristics:

- More than 500 lines of code

- Many variables used but mostly reused variables

- Many layers of nesting of while and if-else statements

- Many chains of calls

- As many combinations of relations between different design entities as possible, ensuring non-empty and non-trivial results for most design abstractions

Secondly, we wanted to ensure the efficiency of evaluation of queries. We checked the time taken measured by AutoTester with the optimizer turned on and ensured that none of the queries would timeout. Some queries are also written by taking into consideration the characteristics of the QueryOptimizer mentioned in Section 3.3.2. Thus, we can observe if the optimizer is able to make evaluation faster. However, checking the results of some queries proved to be difficult and thus, we are unable to fully check the correctness of all the queries. The queries have the following characteristics:

- Characteristics mentioned in Section 4.3.1

- Multiple clauses with unsorted clauses in a group

- Multiple clauses with multiple groups but clauses are not placed together with other clauses in the same group

- Multiple clauses with multiple groups random ordering of groups

- Multiple clauses that can be optimized by rewriting clauses using information in with clauses

Below are some of the results of AutoTester when testing queries with multiple clauses.

Query	Query Type	Timing without optimization/ms	Timing with optimization/ms

`while w; assign a; if ifs; variable v;`
`Select w pattern ifs(v, _, _)`
`such that Parent*(w, a) with`
`v.varName = "green" such that`
`Next*(a, ifs)`
	1 group with unsorted clauses	2532	46

`if ifs; variable v, v1, v2;`
`read r; call c; constant`
`constant; print print; assign`
`a;`
`Select <ifs, constant, v2>`
`pattern ifs(v, _ , _ ) such`
`that Uses(c, v1) with`
`print.varName = v2.varName`
`such that Next(ifs, r) with`
`c.stmt# = constant.value such`
`that Uses(a, v2)`
	3 groups but clauses are not put together with the other members in the same group	40570	423

`prog_line n; stmt s,s1,s2;`
`assign a,a1,a2; read r; print`
`pn; variable v,v1,v2;`
`Select <r, v> such that`
`Follows*(pn, r) with`
`v.varName = pn.varName such`
`that Affects*(n, a) with`
`s.stmt# = n such that`
`Follows*(460, 520) and`
`Next(484, 481)`
	Unsorted 3 groups	233	14

`stmt s,s1; assign a;`
`prog_line n;`
`Select <s,n,s1,a> such that`
`Affects*(s,n) and`
`Affects*(n,s1) with s.stmt# =`
`n and s.stmt# = 48 such that`
`Affects(50,a)`
	Can be optimized by rewriting clauses	73	27

5 Extensions to SPA

The implementation of the extensions are mainly done within the PKB and the QP. The algorithms to compute NextBip, NextBip*, AffectsBip and AffectsBip* are computed in the PKB. The PKB also provide the getter methods for the QP to retrieve the stored information. In the QP, we have created additional helper evaluators in the QP to handle evaluation of NextBip, NextBip*, AffectsBip and AffectsBip*.

Below is a list of the helper evaluators created in this extension.

Helper evaluator	Purpose
AffectsBipEvaluator	Evaluates AffectsBip clauses
AffectsBipTEvaluator	Evaluates AffectsBip* clauses
NextBipEvaluator	Evaluates NextBip clauses
NextBipTEvaluator	Evaluates NextBip* clauses

The extension to the query processor to parse these clauses is similar to that of all other such that clauses, except that it calls different getter methods from the PKB. These getter methods can be found within the PKB API in Appendix 8.2.2. Thus, for the rest of this section, we will mainly be discussing the implementation of the extensions within the PKB. NextBip and NextBip* are implemented in the NextBip.cpp file, while AffectsBip and AffectsBip* are implemented in the Affects.cpp file. To run the extensions, set the booleans runNextBip and runAffectsBip to be true in NextBip.h and AffectsBip.h respectively. Note that to run AffectsBip and AffectsBip* extensions, both booleans have to be turned on (that is, we need to compute NextBip* relationship first).

5.1 Data Structures Used for Extensions in PKB

Similar to the storage of all the other design abstractions, we have nextBipMap, reverseNextBipMap, nextBipStarMap, reverseNextBipStarMap, affectsBipMap, reverseAffectsBipMap, affectsBipStarMap and reverseAffectsBipStarmap to store the NextBip, NextBip*, AffectsBip, AffectsBip* relationships.

In addition to the above maps, we have cfgBipMap. This map is used to keep track of branches between two program lines that have the NextBip relationship. The key of the map will be the pair of program lines, and the value will be the type of branch. For instance, if we are branching in from program line 3 to program line 6 in another procedure, we will store {<3,6>:3} in the cfgBipMap. Suppose the last statement of this callee procedure that we just branched into is 10, and we are

returning to program line 4 of the caller procedure. Then, we will store {<10,4>: -3} into the cfgBipMap. If there are no branches taken between two program lines that has NextBip relationship, we will store the branch type value as 0. For instance, if program line 7 runs right after program line 6, and they are in the same procedure, then we will store {<6,7>: 0} in the cfgBipMap.

We also have nextWithDummyMap and nextBipWithDummyMap. The nextWithDummyMap is akin to the nextMap. But for every program line that has no next program line that can run right after, we will give it a unique dummy node with a negative program line. The nextBipWithDummyMap is just the nextBipMap before we remove the dummy nodes.

Finally, we have nextBipWithBranchStackMap, nextBipStarWithBranchStackMap, nextBipStarWithBranchStackNoDummyMap and affectsBipWithBranchStackMap. In these maps, in addition to storing the statement numbers involved in the relationship, we also store the branches taken by a path to reach the program line. Thus, the keys of these maps will be strings, and the values of the map will be a set of strings. Each string will be a concatenation of the statement number with the branches taken, separated by spaces. To give a concrete example, an entry in the nextBipWithBranchStackMap for the code example below will be {"4 1": ["5 1 4"]}, representing that node 4 (that has been reached after taken a branch-in 1) has NextBip relationship with node 5, which is reached via branch-ins 1 and 4.

5.2 Implementation of NextBip in PKB

The computation of NextBip relationship is implemented within PKB's method `populateNextBip()`. Within this method, we will be populating three maps, namely, the nextBipMap, cfgBipMap, and the nextBipWithDummyMap. We will iterate through all the program lines, and check that if it is a call statement. If it is a call statement, we will get the procedure range, that is, the first and the last statement of the procedure.

We will store the NextBip relationship between the call statement and the first statement number in the callee procedure. We are branching into the callee procedure, thus we keep track of the branch-ins in the cfgBipMap as well.

We also need to branch back from the last statement number of the callee procedure to the caller. There are a few cases to consider. Firstly, if the last statement of the callee procedure is an if-statement, then there are actually two statements to branch back from – the last statement in the if-statement list and the last statement in the else-statement list. Secondly, if the last statement of the callee procedure is a call statement, then we will be branching back from the dummy node after the call statement. For all the other cases, we can just branch back from the last statement of the callee procedure. Hence, we will store the NextBip relationship and keep track of the branch-backs in the cfgBipMap according to the above-mentioned cases.

If the program line that we are looking at is not a call statement, we can handle it like how we handle the Next relationship, since there is no branching in and branching back. Thus, we will store the relationship according to what we already have in the nextWithDummyMap.

Now, the nextBipMap contains dummy nodes. Some branch-backs involve dummy nodes, for instance, in the case of nested calls as seen in the diagram below.

To deal with these branch-backs, referring to the diagram above, when we encounter a dummy node (e.g. -3), we will keep getting the NextBip of this dummy node, until we see a non-dummy node (e.g. 2). We will then update the NextBip relationship between two non-dummy nodes (e.g. NextBip(5,2). After the NextBip relationship has been updated, we will clean up by removing the dummy node keys in the nextBipMap (e.g. -1, -2, -3).

Finally, in the method `populateReverseNextBip()` we will populate the reverseNextBipMap by iterating through the nextBipMap and store the reverse relationship.

5.3 Implementation of NextBip* in PKB

Before computing the NextBip* relationship, we will first populate nextBipWithBranchStackMap. This is because merely computing the transitive closure from the NextBip relationship is insufficient. For instance, if we have branched into procedure b from procedure a, we need to take the branch back to procedure a and not to another procedure. Thus, we need additional information about the path taken to reach each node.

To populate the nextBipWithBranchStackMap, we will do a depth-first-search (dfs) from the first statement of each procedure that has no callers. We will skip the procedures with callers because by doing dfs, we will branch-in and reach it at some point anyway. During the dfs from a particular procedure, we will first set the key of the map to be the concatenation of n1 with the branches taken to the program line n1 so far (we call it the branchStack). Next, we will get the NextBip of n1 (n2's) (including the dummy nodes). For each n2, we will look up the cfgBipMap to see what kind of branch will get n1 to n2. If it is a branch-in, we will concatenate the branch to the branchStack to form s2, and store (s1, s2) into the nextBipWithBranchStackMap. If it is a branch-back, we will look at what's on top of the branchStack. If the last branch taken is the branch-in corresponding to this branch-back, we will take the branch back by popping the branch-in from the branchStack to form s2 and store (s1,s2) into nextBipWithBranchStackMap. The last case will be if we are not branching-in or branching-back. Then we won't need to do any concatenation to the branch stack, and just store (s1,s2) into nextBipWithBranchStackMap. Like any other dfs algorithm, we will then proceed to perform dfs on the next neighbour if the next neighbour has not been visited before. Finally, before moving on to the next n2, we will restore the branchStack that we have previously (the branchStack of n1).

When we have the nextBipWithBranchStackMap, we can simply perform computation of transitive closure of this map to obtain the NextBip* relationship and populate the nextBipStarMap and nextBipStarWithBranchStackMap. We chose to do this using breadth-first search. At the same time, we populate another map, the nextBipStarWithBranchStackNoDummyMap, by removing the dummy nodes from the branchStack. This map will be used to compute the AffectsBip relationship later on.

Finally, in the method `populateReverseNextBipStar()` we will populate the reverseNextBipStarMap by iterating through the nextBipStarMap and store the reverse relationship.

5.4 Implementation of AffectsBip in PKB

The population of AffectsBip relationship is implemented within the method `populateAffectsBip()`. In this method, we will be populating affectsBipMap and the affectsBipWithBranchStackMap. For every program line that are assignment statements, a1, and has NextBip, we first get all the NextBip that are also assignment statements, a2s. Because the branch taken by every path to reach each node is important, instead of retrieving the NextBip's from the nextBipMap, we will retrieve from nextBipWithBranchStackMap. For each a2, we check that a2 uses some variable v that is modified by a1. We then check that v is not modified by any assignment statement along the path using dfs. If all these checks are passed, we will store the AffectsBip relationship between a1 and a2. We will also store the relationship into the affectsBipWithBranchStackMap.

5.5 Implementation of AffectsBip* in PKB

The computation of the AffectsBip* relationship is implemented within the method `populateAffectsBipStar()`. This is done by computing the transitive closure of the AffectsBip relationship in the affectsBipWithBranchStackMap using the dfs algorithm. We can populate the relationship simply by computing the transitive closure since we are using the map with branchStack and the same program line called from different program lines in the SIMPLE program can be uniquely identified.

5.6 Schedule for Implementation

The following table shows the schedule for implementation of the extensions.

Week	Implementation Schedule
Week 11	Brainstorm data structures needed and algorithms needed to compute NextBip and NextBip* relationship. Populate the NextBip and NextBip* relationships in the PKB. Parse and evaluate queries, focusing on NextBip and NextBip* relationships. Conduct unit testing for NextBip and NextBip*.
Week 12	Brainstorm data structures needed and algorithms needed to compute AffectsBip and AffectsBip* relationship. Populate the AffectsBip and AffectsBip* relationships in the PKB. Parse and evaluate queries, focusing on AffectsBip and AffectsBip* relationships. Conduct unit testing for AffectsBip and AffectsBip*.
Week 13	Conduct integration testing of PKB and QP for the extension. Conduct system testing and stress testing.

5.7 Test Plan

In each unit test for the extensions, we come up with a small SIMPLE source code. We design each source code with certain characteristics.

To test NextBip and NextBip*, some characteristics that we want in our SIMPLE program are:
- Calling procedure with the last statement being another call statement
- Calling procedure with the last statement being a (nested) if-statement
- Calling procedure within a (nested) while loop
- Calling the same procedure multiple times within a procedure
- Calling different procedures within a procedure
- Calling procedure at the last statement
- Calling procedure at the last statement of the if statement list
- Calling procedure at the last statement of the else statement list

To test AffectsBip and AffectsBip*, in addition to the characteristics that we want for our NextBip and NextBip* test, some characteristics that we want in our SIMPLE program are:

1) Affects relationships in reverse order in a procedure (e.g. statement 2 affects 1, 3 affects 2, etc) and there are no Next relationships between statement 2 to 1, statement 3 and 2 (only NextBip relationship).

2) Calling procedure described in (1) multiple times, and check the number of AffectsBip relationship is captured.

3) All procedures use and modifies common variables, so that we can test for AffectsBip and AffectsBip* across different procedures.

Integration testing between PKB and QP is similar to that of the other design abstractions. We mainly focused on positive test cases, checking that the NextBip, NextBip*, AffectsBip, AffectsBip* relationships are correctly computed in PKB and that the Query Processor correctly retrieves information stored in PKB using the getter methods provided by the PKB.

Similarly, system testing of the extensions follow the strategy we described in 4.3 System Testing.

6 Coding & Documentation Standards

This section documents the coding standards used in the project, and documentation standards for API design.

6.1 Coding Standards

We mainly follow the coding standards documented in https://google.github.io/styleguide/cppguide.html. However, for indentation, we use tabs (4 spaces) instead of the stated 2 spaces. For method and variable names, we use camel case instead of Pascal case and we use Pascal case for filenames. We also indent contents in namespaces.

6.2 Documentation Standards

We generally follow the same naming conventions in abstract API and our C++ program. This promotes consistency in the code and improves clarity in case of ambiguity. It also avoids naming conflicts and improves understanding within the team.

Typedefs provide a level of abstraction away from the actual types being used, enabling us to focus more on the concept of what a variable should mean. Symbolic names for types make it easier to write clean code and modify the code. We also use the following typedefs to map abstract API type names to C++ types.

```
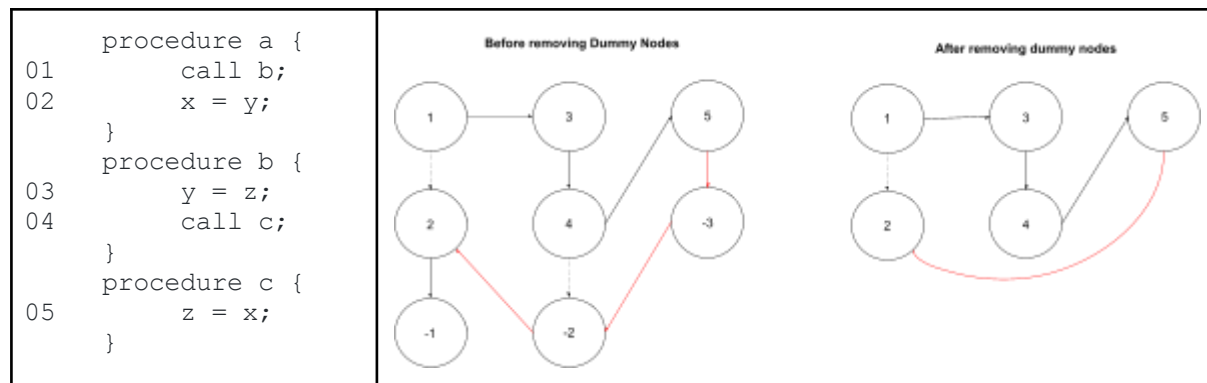typedef int StmtNum;
typedef int ProgLine;
typedef int VarID;
typedef int ProcID;
typedef int CONST;
typedef std::string STRING;
```

The abstract API provides a layer of abstraction and hides implementation details so that we can focus on the more important aspects of our component. It also provides a clearer direction on how to implement the project and improves communication within the team as we have a common understanding.

7 Discussion

One of the problems encountered was our initial adoption of the code-first approach. Initially we were too eager and jumped straight into coding, without having the big picture in mind. As a result, we ended up getting stuck frequently which hindered our progress. However, after the lecture on API, we realised that we should design the API first. Hence, we started on planning our API carefully, discussing and finding out the needs of each sub-teams before we started coding. We have learnt that designing the API first can make sure that everyone is on the same page, and save our time during implementation. This is something we can take away and apply to the future projects that we embark on.

Another problem faced was communication. This is made worse by the fact that our team does not meet up physically at all. Hence, it took quite a while to break the ice and start communicating our needs. As a result, there is a severe lack of communication within the team. Often, we would communicate within each sub-team, and when needed we would communicate with a member that we know in another sub-team to find out their needs and progress. This is highly inefficient, and sometimes, some team members are not aware of major changes. However, we gradually improved in this area and the members communicated more frequently, and things got better in subsequent iterations.

8 Appendix

8.1 SP Abstract API

8.1.1 Parser API

Parser API
Overview: Parser methods used to convert SIMPLE source code strings into AST.

	API
	PARSER Parser(**STRING** src_code); **Description**: Returns a Parser object to parse the given SIMPLE source code provided as argument.
	NODE* parseProgram(); **Description:** Returns root AST node representing entire Program if SIMPLE source code is a valid program. Syntax errors would cause a ParserException to be thrown.
	EXPR parseExpr(); **Description:** Returns root AST node of an expression if SIMPLE source code is a valid expression. Syntax errors would cause a ParseExprException to be thrown.
	BOOLEAN parse(); **Description:** Parses SIMPLE source code as a program, populates PKB via DE. Returns true if successful, false if syntax error or failure caused by ParserException methods it calls.

8.1.2 ParserException API

ParserException API
Overview: Exceptions indicating syntax errors that are thrown by Parser methods.

	API
	STRING ParserException::what(); **Description**: Returns a string describing exception. ParserException subclasses std::exception and can be caught
	STRING ParseExprException::what(); **Description**: Returns a string describing exception. ParseExprException subclasses ParserException and can be caught

8.1.3 Node API

Node API
Overview: A Node of an AST, returned from Parser::parseProgram() Most of the AST methods are mainly used within SP, DE and are not intended for use outside the component.

	API
	TOKEN getToken(); **Description**: Returns token to identify type of Node

8.1.3 Expr API

Expr API
Overview: An Expr Node of an AST, returned from Parser::parseExpr(), there is no need to construct it, it is a subclass of Node.

	API
	STRING toString(); **Description**: Returns string representation of the entire expression in explicitly bracketed infix left-associative order. This is intended for use outside of SP.

8.1.5 DE API

Design Extractor
Overview: Stores information into the PKB

	API
	VOID storeNewProcedure(**STRING** procedureName); **Description**: This method is called when processing a new procedure.
	VOID exitProcedure(); **Description:** This method is called when exiting a procedure.
	VOID storeNewAssignment(**StmtNum** stmtNum, **STRING** variableName, **AssignStmt*** AST); <pre-condition>: all given information is correct and valid. **Description**: Calls storeVariableName first to store the variableName and retrieve the variable ID. Note that variableName is the name of the variable on the LHS of the assignment. Stores <stmtNum, **PAIR**<variable ID, *AST>> into Assignment Map.Stores Modifies and Uses relationships as well.
	BOOLEAN storeNewCall(**StmtNum** stmtNum, **STRING** callerName,**STRING** procedureName, **CallStmt*** AST); **Description**: Stores <stmtNum, *AST> into StmtAST Map by calling storeStatement. Finds or adds the entry of the callerName in the NestedCallsMap and append procedureName to the calledProceduresList of the entry. Returns true (success) if successful; otherwise, returns false (if this is a recursive call).
	VOID storeNewWhile(**StmtNum** startStmtNum, **vector<STRING>** condVars, **LIST_OF_VARIABLES** condConsts, **WhileStmt*** AST); **Description**: Stores <startStmtNum, *AST> into StmtAST Map by calling storeStatement. Stores startStmtNum into WhileStmtNums. Store each var in condVars into the PKB, and update Uses relationships for them. Note: condVars is the list of all variables used in the conditional
	VOID exitWhile(); **Description:** This method is called when exiting a while statement.
	VOID storeNewIf(**StmtNum** startStmtNum, **LIST_OF_VARIABLES** condVars, **LIST_OF_VARIABLES** condConsts, **IfStmt*** AST);

	Description: Stores <stmtNum, *****AST> into StmtAST Map by calling storeStatement. Stores <startStmtNum, endStmtLstNum, *****AST> into IfMap. Stores the condVarName into PKB if it doesn't already exist. Store each var in condVars into the PKB, and update Uses relationships for them.
	VOID storeNewElse(); **Description:** Processes the else section of if-else statement
	VOID endIfElse(); **Description:** This method is called when exiting an if-else statement.
	VOID storeNewRead(**StmtNum** stmtNum, **STRING** variableName, **ReadStmt*** AST); **Description:** calls storeVariableName first to store the variableName and retrieve the variable ID. Calls storeStatement(stmtNum, *AST). Adds a new entry to the ModifiesTable.
	VOID storeNewPrint(**StmtNum** stmtNum, **STRING** variableName, **PrintStmt*** AST); **Description:** calls getVariableID(variableName) first to convert variableName into the variableID. Calls storeStatement(stmtNum, *AST). Adds a new entry to the UsesTable.
	BOOLEAN signalEnd(); **Description:** Informs DE that end of file (EOF) has been reached. This will initiate various operations in PKB. Returns true if successful, false otherwise.
	VOID signalReset(); **Description:** Informs DE that an error was countered, causing DE (local state variables) and PKB to be reset.

8.2 PKB Abstract API

8.2.1 Design Entities

VarTable API Overview: VarTable stores variable information into the PKB.	
	API
	VarID storeVarName(**STRING** variableName); **Description**: stores <variableName, variableID> into the varNameIDMap if it does not exist. Stores <variableName> in varNames vector if it does not exist. Returns the ID of the variable name.
	BOOLEAN hasVar(**STRING** varName); **Description:** Returns true if the variable has already been added before; otherwise, returns false.
	VarID getVarID(**STRING** varName); **Description:** returns the ID corresponding to the varName, if it exists. Returns -1 otherwise.
	STRING getVarName(**VarID** varID); **Description:** returns the varName corresponding to the ID, if it exists. If 'varID' is out of range, throw an exception.
	LIST_OF_VARIABLES CONST &getAllVarNames() **CONST**;

	Description: returns a const reference to the vector of all variable names stored.
	LIST_OF_VARIABLES convertVarIDsToNames(**LIST_OF_VARIABLE_IDS** varIDs); **Description:** returns a vector of variable names corresponding to the vector of variable IDs given.
	LIST_OF_VARIABLE_IDS getAllVarIDs(); **Description:** returns a vector of all variable IDs.
	INT getSize(); **Description:** Returns the number of variables in the table.

ProcTable API
Overview: ProcTable stores procedure information into the PKB.

	API
	ProcID storeProcName(**STRING** procedureName); **Description**: Stores <procedureName, procedureID> into the ProcName map if it does not exist. Stores <procedureID, procedureName> into the ProcID map if it does not exist. Returns the ID of the procedure name.
	BOOLEAN hasProc(**STRING** varName); **Description:** Returns true if the procedure has already been added before; otherwise, returns false.
	BOOLEAN storeProcStmt(**ProcID** procID, **StmtNum** startStmt, **StmtNum** endStmt); **Description:** stores the procedure that spans from startStmt to endStmt into the ProcStmtMap. Returns true if insertion took place, returns false otherwise.
	ProcID getProcID(**STRING** procedureName); **Description:** returns the ID corresponding to the procedureName, if it exists. Returns -1 otherwise.
	STRING getProcName(**ProcID** procID); **Description:** returns the procedureName corresponding to the ID, if it exists. If 'procID' is out of range, throw an exception.
	PAIR<StmtNum, StmtNum> getProcRange(**ProcID** procID); **Description:** returns the procedure's start statement and end statement. Throws an exception otherwise.
	LIST_OF_PROCEDURES CONST &getAllProcNames() **CONST**; **Description:** returns a const reference to the vector of all procedure names stored.
	LIST_OF_PROCEDURES convertProcIDsToNames(**LIST_OF_PROCEDURE_IDS** procIDs); **Description:** returns a vector of procedure names corresponding to the vector of procedure IDs given.
	LIST_OF_PROCEDURE_IDS getAllProcIDs(); **Description:** returns a vector of all procedure IDs.
	INT getSize();

	Description: Returns the number of procedures in the table.

ConstTable API
Overview: ConstTable stores information about constants and the statements using them.

	API
	CONST getConstValue(**STRING** const); **Description:** Returns the value of the specified constant. If const does not exist in ConstMap, return -1.
	INT getSize(); **Description:** Returns the number of variables in the table.
	LIST_OF_CONSTANTS CONST &getAllConst() **CONST;** **Description:** Returns a const reference to the vector of all constants.
	BOOLEAN hasConst(**STRING** const); **Description**: Returns TRUE if const exists in the table; false otherwise.
	INT getSize(); **Description:** Returns the number of constants in the table.
	BOOLEAN storeConst(**STRING** constantName); **Description:** stores the <constantName, constantID> into the constMap if it does not exist. Stores <constantName> into the consts vector if it does not exist. Returns true if the information is successfully added. Returns false otherwise.

StmtTable API
Overview: StmtTable stores statements design entities in the PKB.

	API
	BOOLEAN storeStmt(**StmtNum** stmtNum, **NODE*** AST, **STRING** type); **Description**: stores <stmtNum, *AST> into StmtAST Map. Also store the stmtNum into respective vectors depending on the type (assign, read, print, etc). Returns true if the information is added successfully, returns false otherwise.
	BOOLEAN storeAssignExpr(**StmtNum** stmtNum, **STRING** varName, **STRING** expr); **Description:** stores <stmtNum, pair<lhs, rhs>> of an assignment statement to the assignExprMap. Returns true if the information is added successfully, returns false otherwise.
	BOOLEAN storeIfPattern(**StmtNum** stmtNum, **VarID** controlVarID); **Description:** Stores <stmtNum, controlVarID> into ifPatternsMap, and <controlVarID, stmtNum> into the reverseIfPatternsMap. Returns true if the information is added successfully, returns false otherwise.
	BOOLEAN storeWhilePattern(**StmtNum** stmtNum, **VarID** controlVarID); **Description:** Stores <stmtNum, controlVarID> into whilePatternsMap, and <controlVarID, stmtNum> into the reverseWhilePatternsMap. Returns true if the information is added successfully, returns false otherwise.

BOOLEAN storeReadVariableForStmt(**StmtNum** stmtNum, **VarID** readVarID); **Description:** Stores <stmtNum, varID> into readVariablesMap. Stores <varID, stmtNum> into the reverseReadVariablesMap. Returns true if the information is stored successfully, returns false otherwise.
BOOLEAN storePrintVariableForStmt(**StmtNum** stmtNum, **VarID** readVarID); **Description:** Stores <stmtNum, varID> into printVariablesMap. Stores <varID, stmtNum> into the reversePrintVariablesMap. Returns true if the information is stored successfully, returns false otherwise.
BOOLEAN isIfStmtWithControlVar(**StmtNum** stmtNum, **VarID** controlVarID); **Description:** Returns true if statement at stmtNum is an If statement and it has controlVarID as its control variable.
BOOLEAN isIfStmtWithControlVar(**StmtNum** stmtNum, **VarID** controlVarID); **Description:** Returns true if statement at stmtNum is a While statement and it has controlVarID as its control variable.
VarID getReadVariableOfStmt(**StmtNum** stmtNum); **Description:** Returns the read variable given the stmtNum. Returns -1 if the stmtNum is not a read statement.
VarID getPrintVariableOfStmt(**StmtNum** stmtNum); **Description:** Returns the print variable given the stmtNum. Returns -1 if the stmtNum is not a print statement.
SET_OF_STMT_NOS CONST &getIfStmtsWithControlVar(**VarID** controlVarID) **CONST**; **Description:** Returns a const reference to the set of statement numbers given the control variable. Returns an empty set if it is not a control variable of an If statement.
SET_OF_STMT_NOS CONST &getWhileStmtsWithControlVar(**VarID** controlVarID) **CONST**; **Description:** Returns a const reference to the set of statement numbers given the control variable. Returns an empty set if it is not a control variable of a While statement.
SET_OF_VARIABLE_IDS CONST &getControlVarsOfIfStmt(**StmtNum** stmtNum) **CONST**; **Description:** Returns a const reference to the set of control variables given the statement number. Returns an empty set if the statement number given is not an If statement or does not use a control variable.
SET_OF_VARIABLE_IDS CONST &getControlVarOfWhileStmt(**StmtNum** stmtNum) **CONST**; **Description:** Returns a const reference to the set of control variables given the statement number. Returns an empty set if the statement number given is not a While statement or does not use a control variable.
SET_OF_STMT_NOS CONST &getStmtNumsOfReadWithVar(**VarID** readVarID) **CONST**; **Description:** Returns a const reference to the set of read statement numbers given the read variable. Returns an empty set if the variable given is not a variable involved in any read statements.
SET_OF_STMT_NOS CONST &getStmtNumsOfPrintWithVar(**VarID** printVarID) **CONST**; **Description:** Returns a const reference to the set of print statement numbers given the print variable. Returns an empty set if the variable given is not a variable involved in any print statements.

	LIST_OF_STMT_NOS CONST &getAllAssignStmtNums() **CONST**; **Description:** Returns a const reference to the vector of all the StmtNums in the AssignStmtMap.
	LIST_OF_STMT_NOS CONST &getAllReadStmtNums() **CONST**; **Description:** Returns a const reference to the vector of all the StmtNums in the ReadStmtMap.
	LIST_OF_STMT_NOS CONST &getAllPrintStmtNums() **CONST**; **Description:** Returns a const reference to the vector of all the StmtNums in the PrintStmtMap.
	LIST_OF_STMT_NOS CONST &getAllCallStmtNums() **CONST**; **Description:** Returns a const reference to the vector of all the StmtNums in the CallStmtMap.
	LIST_OF_STMT_NOS CONST &getAllWhileStmtNums() **CONST**; **Description:** Returns a const reference to the vector of all the StmtNums in the WhileStmtMap.
	LIST_OF_STMT_NOS CONST &getAllIfStmtNums() **CONST**; **Description:** Returns a const reference to the vector of all the StmtNums in the IfStmtMap.
	LIST_OF_STMT_NOS CONST &getAllStmtNums() **CONST**; **Description:** Returns a const reference to the vector of all the StmtNums stored.
	PAIR<LIST_OF_STMT_NOS, LIST_OF_STMT_NOS > getAllIfPatterns(); **Description:** Returns a pair of vectors in the ifPatternsMap. The first vector is a vector of statement numbers. The second vector is a vector of controlVarIDs. For e.g., if (1,2), (3,4), (5,6) is in ifPatternsMap, then it will return <[1,3,5],[2,4,6]>.
	PAIR<LIST_OF_STMT_NOS, LIST_OF_VARIABLE_IDS> getAllWhilePatterns(); **Description:** Returns a pair of vectors in the whilePatternsMap. The first vector is a vector of statement numbers. The second vector is a vector of controlVarIDs. For e.g., if (1,2), (3,4), (5,6) is in whilePatternsMap, then it will return <[1,3,5],[2,4,6]>.
	BOOLEAN hasStmt(**StmtNum** stmtNum); **Description:** Returns true if the stmtNum has already been added before; otherwise, returns false.
	NODE* getStmtNode(**StmtNum** stmtNum); **Description:** Returns the pointer to the part of the AST corresponding to the stmtNum as stored in StatementASTMap. Throws error if stmtNum is not found.
	PAIR<STRING, STRING> getAssignExpr(**StmtNum** stmtNum); **Description:** Returns the PAIR<lhs, rhs> of the assignment statement given the stmtNum. Throws error if the information is not stored in the assignExprMap.
	INT getSize(); **Description:** Returns the number of statements in the table.
	INT getIfPatternsSize(); **Description:** Returns the no. of entries in ifPatternsMap. E.g. if the ifPatternsMap has [1: {2,3}], then the size is 2. Because there are pairs (1,2) and (1,3).
	INT getWhilePatternsSize(); **Description:** Returns the number of entries in whilePatternsMap. E.g. if the whilePatternsMap has [1: {2,3}], then the size is 2. Because there are pairs (1,2) and (1,3).

	StmtLstTable API Overview: StmtLstTable stores the first statement number of all stmtLsts
	API
	BOOLEAN storeStmtLst(**StmtNum** firstStmtNum); **Description**: stores <firstStmtNum> into StmtLsts. Returns true if the information is added successfully, returns false otherwise.
	INT getSize(); **Description:** Returns the number of stmtLsts stored.
	LIST_OF_STMT_NOS CONST &getAllStmtLsts() **CONST**; **Description:** Returns a const reference to the vector of the first statement numbers of all the stmtLsts stored.
	BOOLEAN hasStmtLst(**StmtNum** firstStmtNum); **Description:** Returns true if the firstStmtNum has already been added before (stmtLst is stored); otherwise, returns false.

8.2.2 Design Abstractions

	Follows API Overview: Follows store Follows and Follows* relationship information in the PKB.
	API
	VOID populateFollowsStar(); **Description:** Compute all Follows* relationships from the FollowsMap and populate the FollowsStarMap and ReverseFollowsStarMap.
	BOOLEAN storeFollows(**StmtNum** followedStmtNum, **StmtNum** followerStmtNum); **Description:** Stores <s1, s2> in the followsMap. Stores <s2, s1> in the reverseFollowsMap. Returns true if the information is added successfully, returns false otherwise.
	BOOLEAN hasFollower(**StmtNum** s1); **Description:** Returns true if there is a statement s2 such that Follows(s1, s2) holds, returns false otherwise.
	BOOLEAN hasFollowee(**StmtNum** s2); **Description:** Returns true if there is a statement s1 such that Follows(s1, s2) holds, returns false otherwise.
	BOOLEAN isFollows(**StmtNum** s1, **StmtNum** s2); **Description:** Returns true if Follows(s1, s2) holds and the information is stored in the PKB, returns false otherwise.
	BOOLEAN isFollowsStar(**StmtNum** s1, **StmtNum** s2); **Description:** Returns true if Follows*(s1, s2) holds and the information is stored in the PKB, returns false otherwise.
	StmtNum getFollower(**StmtNum** s1);

	Description: Returns s2, the statement that follows s1. Returns -1 if there is no such statement.
	StmtNum getFollowee(**StmtNum** s2); **Description:** Returns s1, the statement that is followed by s2. Returns -1 if there is no such statement.
	SET_OF_STMT_NOS CONST &getFolloweeStar(**StmtNum** s2) **const**; **Description:** Returns a const reference to the set of s1, the statements that Follows*(s1, s2).
	SET_OF_STMT_NOS CONST &getFollowerStar(**StmtNum** s1) **const**; **Description:** Returns a const reference to the set of s2, the statements that Follows*(s1, s2).
	PAIR<LIST_OF_STMT_NOS, LIST_OF_STMT_NOS> getAllFollows(); **Description:** Returns a pair of vectors in the followsMap. First vector is a vector of s1's. Second is a vector of s2's. For e.g., if (1,2), (3,4), (5,6) is in the followsMap, then it will return <[1,3,5],[2,4,6]>
	PAIR<LIST_OF_STMT_NOS, LIST_OF_STMT_NOS> getAllFollowsStar(); **Description:** Returns a pair of vectors in the followsStarMap. First vector is a vector of s1's. Second is a vector of s2's. For e.g., if (1,2), (3,4), (5,6) is in the followsStarMap, then it will return <[1,3,5],[2,4,6]>
	INT getFollowsSize(); **Description:** Returns the no. of entries in FollowsMap
	INT getFollowsStarSize(); **Description:** Returns the no. of pairs of Follows* relationship. E.g. if the followsStarMap has [1: {2,3}], then the size is 2. Because there are pairs (1,2) and (1,3).

Parent API Overview: Parent store Parent and Parent* relationship information in the PKB.
	API
---	---
	VOID populateParentStar(); **Description:** Compute all Parent* relationships from the ParentMap and populate the ParentStarMap and ReverseParentStarMap.
	BOOLEAN storeParent(**StmtNum** parentStmtNum, **StmtNum** childStmtNum); **Description:** add <parentStmtNum, childStmtNum> to the ParentMap. Add <childStmtNum, parentStmtNum> to the ReverseParentMap. Returns true if the information is added successfully or if the information already exists in PKB, returns false otherwise.
	BOOLEAN hasChild(**StmtNum** s1); **Description:** Returns true if there is a statement s2 such that Parent(s1, s2) holds, returns false otherwise.
	BOOLEAN hasParent(**StmtNum** s2); **Description:** Returns true if there is a statement s1 such that Parent(s1, s2) holds, returns false otherwise.
	BOOLEAN isParent(**StmtNum** s1, **StmtNum** s2);

	Description: Returns true if Parent(s1, s2) holds and the information is stored in the PKB, returns false otherwise.
	BOOLEAN isParentStar(**StmtNum** s1, **StmtNum** s2); **Description:** Returns true if Parent*(s1, s2) holds and the information is stored in the PKB, returns false otherwise.
	SET_OF_STMT_NOS CONST &getChildren(**StmtNum** s1) **CONST**; **Description:** Returns a const reference to the set of s2, the statement that is the child of s1. Returns an empty set if there are no children.
	StmtNum getParent(**StmtNum** s2); **Description:** Returns s1, the statement that is the parent of s2. Returns -1 if there is no such statement.
	SET_OF_STMT_NOS CONST &getParentStar(**StmtNum** s2) **const**; **Description:** Returns a const reference to the set of s1, the statements that Parent*(s1, s2).
	SET_OF_STMT_NOS CONST &getChildrenStar(**StmtNum** s1) **const**; **Description:** Returns a const reference to the set of s2, the statements that Parent*(s1, s2).
	PAIR<LIST_OF_STMT_NOS, LIST_OF_STMT_NOS> getAllParent(); **Description:** Returns a pair of vectors in the parentMap. First vector is a vector of s1's. Second is a vector of s2's. For e.g., if (1,2), (3,4), (5,6) is in parentMap, then it will return <[1,3,5],[2,4,6]>
	PAIR<LIST_OF_STMT_NOS, LIST_OF_STMT_NOS> getAllParentStar(); **Description:** Returns a pair of vectors in the parentStarMap. First vector is a vector of s1's. Second is a vector of s2's. For e.g., if (1,2), (3,4), (5,6) is in parentStarMap, then it will return <[1,3,5],[2,4,6]>
	INT getParentSize(); **Description:** Returns the no. of pairs of Parent relationship. E.g. if the parentMap has [1: {2,3}], then the size is 2. Because there are pairs (1,2) and (1,3).
	INT getParentStarSize(); **Description:** Returns the no. of pairs of Parent* relationship. E.g. if the parentStarMap has [1: {2,3}], then the size is 2. Because there are pairs (1,2) and (1,3).

Uses API Overview: Uses store Uses relationship information in the PKB.	
	API
	BOOLEAN storeStmtUses(**StmtNum** stmtNum, **VarID** variableID); **Description:** Stores <stmtNum, variableID> to the stmtUsesMap. Returns true if the information is added successfully, returns false otherwise.
	BOOLEAN storeProcUses(**ProcID** procedureID, **VarID** variableID**);** **Description:** Stores <procedureID, variableID> to the procUsesMap. Returns true if the information is added successfully, returns false otherwise.
	BOOLEAN stmtUsesVar(**StmtNum** stmtNum, **VarID** variableID);

	Description: Returns true if Uses(stmtNum, variable) holds and the information is stored in the PKB, returns false otherwise.
	BOOLEAN procUsesVar(**ProcID** procID, **VarID** variableID); **Description:** Returns true if Uses(proc, variable) holds and the information is stored in the PKB, returns false otherwise.
	SET_OF_VARIABLE_IDS CONST &getVarsUsedByStmt(**StmtNum** stmtNum) **CONST**; **Description:** returns a const reference to a set of variables used by the stmtNum.
	SET_OF_VARIABLE_IDS CONST &getVarsUsedByProc(**ProcID** procID) **CONST**; **Description:** returns a const reference to a set of variables used by the procID.
	SET_OF_STMT_NOS CONST &getStmtsUses(**VarID** varID) **CONST**; **Description:** returns a const reference to a set of statements that uses the varID.
	SET_OF_PROCEDURE_IDS CONST &getProcsUses(**VarID** varID) **CONST**; **Description:** returns the const reference to a set of procedureIDs that uses varID.
	PAIR<LIST_OF_STMT_NOS, LIST_OF_VARIABLE_IDS> getAllStmtUses(); **Description:** Returns a pair of vectors in the stmtUsesMap. First vector is a vector of statements. Second is a vector of varIDs. For e.g., if (1,2), (3,4), (5,6) is in stmtUsesMap, then it will return <[1,3,5],[2,4,6]>
	PAIR<LIST_OF_PROCEDURE_IDS, LIST_OF_VARIABLE_IDS> getAllProcUses(); **Description:** Returns a pair of vectors in the procUsesMap. First vector is a vector of procIDs. Second is a vector of varIDs. For e.g., if (1,2), (3,4), (5,6) is in procUsesMap, then it will return <[1,3,5],[2,4,6]>
	INT getStmtSize(); **Description:** Returns the number of Uses relationship in stmtUsesMap. If stmtUsesMap has [1: {2,3}], then the size is 2.
	INT getProcSize(); **Description:** Returns the number of Uses relationship in procUsesMap. If procUsesMap has [1: {2,3}], then the size is 2.

Modifies API Overview: Modifies store Modifies relationship information in the PKB.	
	API
	BOOLEAN storeStmtModifies(**StmtNum** stmtNum, **VarID** variableID); **Description:** Stores <stmtNum, variableID> to the StmtModifiesMap. Returns true if the information is added successfully, returns false otherwise.
	BOOLEAN storeProcModifies(**ProcID** procedureID, **VarID** variableID**)**; **Description:** Stores <procedureID, variableID> to the procModifiesMap. Returns true if the information is added successfully, returns false otherwise.
	BOOLEAN stmtModifiesVar(**StmtNum** stmtNum, **VarID** variableID); **Description:** Returns true if Modifies(stmtNum, variable) holds and the information is stored in

	the PKB, returns false otherwise.
	BOOLEAN procModifiesVar(**ProcID** procID, **VarID** variableID); **Description:** Returns true if Modifies(proc, variable) holds and the information is stored in the PKB, returns false otherwise.
	SET_OF_VARIABLE_IDS CONST &getVarsModifiedByStmt(**StmtNum** stmtNum) **CONST**; **Description:** returns a const reference to a set of variables modified by the stmtNum.
	SET_OF_VARIABLE_IDS CONST &getVarsModifiedByProc(**ProcID** procID) **CONST**; **Description:** returns a const reference to a set of variables modified by the procID.
	SET_OF_STMT_NOS CONST &getStmtsModifies(**VarID** varID) **CONST**; **Description:** returns a const reference to a set of statements that modifies the varID.
	SET_OF_PROCEDURE_IDS CONST &getProcsModifies(**VarID** varID) **CONST**; **Description:** returns a const reference to a set of procedureIDs that modifies varID.
	PAIR<LIST_OF_STMT_NOS, LIST_OF_VARIABLE_IDS> getAllStmtModifies(); **Description:** Returns a pair of vectors in the stmtModifiesMap. First vector is a vector of statements. Second is a vector of varIDs. For e.g., if (1,2), (3,4), (5,6) is in stmtModifiesMap, then it will return <[1,3,5],[2,4,6]>
	PAIR<LIST_OF_PROCEDURE_IDS, LIST_OF_VARIABLE_IDS> getAllProcModifies(); **Description:** Returns a pair of vectors in the procModifiesMap. First vector is a vector of procIDs. Second is a vector of varIDs. For e.g., if (1,2), (3,4), (5,6) is in procModifiesMap, then it will return <[1,3,5],[2,4,6]>
	INT getStmtSize(); **Description:** Returns the number of Modifies relationship in stmtModifiesMap. If stmtUsesMap has [1: {2,3}], then the size is 2.
	INT getProcSize(); **Description:** Returns the number of Modifies relationship in procModifiesMap. If procModifiesMap has [1: {2,3}], then the size is 2.

Calls API Overview: Calls store Calls and Calls* relationship information in the PKB.	
	API
	BOOLEAN storeCalls(**StmtNum** stmtNum, **ProcID** p, **ProcID** q); **Description:** Stores a calls relationship into the callsRawInfoTable. Returns true if the information is successfully added to the PKB.
	BOOLEAN hasCyclicalCalls(); **Description:** Returns true if the program has cyclical calls.
	BOOLEAN isCalls(**ProcID** p, **ProcID** q); **Description:** Returns true if Calls(p, q) holds and the information is stored in the PKB, where p, q are the ProcID of procedures. Returns false otherwise.

	BOOLEAN isCallsStar(**ProcID** p, **ProcID** q); **Description:** Returns true if Calls*(p, q) holds and the information is stored in the PKB, where p, q are the ProcID of procedures. Returns false otherwise.
	BOOLEAN hasCalls(**StmtNum** stmtNum); **Description:** Returns true if the calls statement at stmtNum is stored in the callsRawInfoTable. Returns false otherwise.
	BOOLEAN processCalls(); **Description:** Processes the callsRawInfoTable to populate the rest of the tables in this class, and updates Modifies and Uses relationship. This method is called after all calls have been stored in the callsRawInfoTable. Returns true if the processing is successful and there are no cyclical calls detected. Returns false otherwise.
	ProcID getCalleeInStmt(**StmtNum** stmtNum); **Description:** Returns the callee of the call statement that occurs at stmtNum. Returns -1 if the statement is not a call statement.
	SET_OF_STMT_NOS CONST &getStmtNumThatCallsCallee(**ProcID** calleeID); **Description:** Returns a const reference to the set of call statements that calls the callee. Returns an empty set if there is no such callee.
	SET_OF_PROCEDURE_IDS CONST &getCallers(**ProcID** q) **CONST**; **Description:** returns a const reference to the set of p's such that Calls(p,q).
	SET_OF_PROCEDURE_IDS CONST &getCallees(**ProcID** p) **CONST**; **Description:** returns a const reference to the set of q's such that Calls(p,q).
	SET_OF_PROCEDURE_IDS CONST &getCallersStar(**ProcID** q) **CONST**; **Description:** returns a const reference to the set of p's such that Calls*(p,q).
	SET_OF_PROCEDURE_IDS CONST &getCallees(**ProcID** q) **CONST**; **Description:** returns a const reference to the set of q's such that Calls*(p,q).
	SET_OF_STMT_NOS CONST &getStmtsOfCalls(**ProcID** p, **ProcID** q); **Description:** returns a const reference to the set of stmts where Calls(p,q) occur.
	PAIR<LIST_OF_PROCEDURE_IDS, LIST_OF_PROCEDURE_IDS> getAllCalls(); **Description:** Returns a pair of vectors in the callsMap. First vector is a vector of p's. Second is a vector of q's. For e.g., if (1,2), (3,4), (5,6) is in callsMap, then it will return <[1,3,5],[2,4,6]>
	PAIR<LIST_OF_PROCEDURE_IDS, LIST_OF_PROCEDURE_IDS> getAllCallsStar(); **Description:** Returns a pair of vectors in the callsStarMap. First vector is a vector of p's. Second is a vector of q's. For e.g., if (1,2), (3,4), (5,6) is in callsStarMap, then it will return <[1,3,5],[2,4,6]>
	INT getCallsSize(); **Description:** Returns the no. of pairs of Calls relationship. E.g. if the callsMap has [1: {2,3}], then the size is 2, because there are pairs (1,2) and (1,3).
	INT getCallsStarSize(); **Description:** Returns the no. of pairs of Calls* relationship. E.g. if the callsStarMap has [1: {2,3}],

	then the size is 2, because there are pairs (1,2) and (1,3).

Next API Overview: Next stores Next and Next* relationship information in the PKB.	
	API
	VOID populateNextStar(); **Description:** Compute all Next* relationships from the nextMap and populate the nextStarMap and reverseNextStarMap.
	BOOLEAN storeNext(**ProgLine** n1, **ProgLine** n2); **Description:** Stores a Next relationship into the nextMap. Returns true if the information is successfully added to the PKB. Returns false otherwise.
	BOOLEAN isNext(**ProgLine** n1, **ProgLine** n2); **Description:** Returns true if Next(n1, n2) holds and the information is stored in the PKB, returns false otherwise.
	BOOLEAN isNextStar(**ProgLine** n1, **ProgLine** n2); **Description:** Returns true if Next*(n1, n2) holds and the information is stored in the PKB, returns false otherwise.
	SET_OF_PROGLINES CONST &getNext(**ProgLine** n1) **CONST**; **Description:** returns a const reference to the set of n2's such that Next(n1, n2).
	SET_OF_PROGLINES CONST &getPrevious(**ProgLine** n2) **CONST**; **Description:** returns a const reference to the set of n1's such that Next(n1, n2).
	SET_OF_PROGLINES CONST &getNextStar(**ProgLine** n1) **CONST**; **Description:** returns a const reference to the set of n2's such that Next*(n1, n2).
	SET_OF_PROGLINES CONST &getPreviousStar(**ProgLine** n2) **CONST**; **Description:** returns a const reference to the set of n1's such that Next*(n1, n2).
	PAIR<LIST_OF_PROGLINES , LIST_OF_PROGLINES> getAllNext(); **Description:** Returns a pair of vectors in the nextMap. First vector is a vector of n1's. Second is a vector of n2's. For e.g., if (1,2), (3,4), (5,6) is in nextMap, then it will return <[1,3,5],[2,4,6]>
	PAIR<LIST_OF_PROGLINES, LIST_OF_PROGLINES> getAllNextStar(); **Description:** Returns a pair of vectors in the nextStarMap. First vector is a vector of n1's. Second is a vector of n2's. For e.g., if (1,2), (3,4), (5,6) is in nextStarMap, then it will return <[1,3,5],[2,4,6]>
	INT getNextSize(); **Description:** Returns the no. of pairs of Next relationship. E.g. if the nextMap has [1: {2,3}], then the size is 2, because there are pairs (1,2) and (1,3).
	INT getNextStarSize(); **Description:** Returns the no. of pairs of Next* relationship. E.g. if the nextStarMap has [1: {2,3}], then the size is 2, because there are pairs (1,2) and (1,3).

	Affects API Overview: Affects stores Affects and Affects* relationship information in the PKB.
	API
	VOID populateAffectsAndAffects*(); **Description:** Compute the Affects and Affects* relationship after Uses and Modifies relationship have been completely populated
	BOOLEAN isAffects(**StmtNum** a1, **StmtNum** a2); **Description:** Returns true if Affects(a1, a2) holds and the information is stored in the PKB, returns false otherwise.
	BOOLEAN isAffectsStar(**StmtNum** a1, **StmtNum** a2); **Description:** Returns true if Affects*(a1, a2) holds and the information is stored in the PKB, returns false otherwise.
	SET_OF_STMT_NOS CONST &getAffects(**StmtNum** a1) **CONST**; **Description:** returns a const reference to the set of a2's such that Affects(a1, a2).
	SET_OF_STMT_NOS CONST &getAffected(**StmtNum** a2) **CONST**; **Description:** returns a const reference to the set of a1's such that Affects(a1, a2).
	SET_OF_STMT_NOS CONST &getAffectsStar(**StmtNum** a1) **CONST**; **Description:** returns aa const reference to the set of a2's such that Affects*(a1, a2).
	SET_OF_STMT_NOS CONST &getAffectedStar(**StmtNum** a2) **CONST**; **Description:** returns a const reference to the set of a1's such that Affects*(a1, a2).
	PAIR<LIST_OF_STMT_NOS, LIST_OF_STMT_NOS> getAllAffects(); **Description:** Returns a pair of vectors in the affectsMap. First vector is a vector of a1's. Second is a vector of a2's. For e.g., if (1,2), (3,4), (5,6) is in affectsMap, then it will return <[1,3,5],[2,4,6]>
	PAIR<LIST_OF_STMT_NOS, LIST_OF_STMT_NOS> getAllAffectsStar(); **Description:** Returns a pair of vectors in the affectsStarMap. First vector is a vector of a1's. Second is a vector of a2's. For e.g., if (1,2), (3,4), (5,6) is in affectsStarMap, then it will return <[1,3,5],[2,4,6]>
	INT getAffectsSize(); **Description:** Returns the number of Affects relationship in affectsMap. If affectsMap has [1: {2,3}], then the size is 2.
	INT getAffectsStarSize(); **Description:** Returns the number of Affects* relationship in affectsStarMap. If affectsStarMap has [1: {2,3}], then the size is 2.

	NextBip API Overview: Next stores NextBip and NextBip* relationship information in the PKB.
	API

	VOID setRunNextBip(**BOOLEAN** runNextBip); **Description:** To switch on (set runNextBip to true) and off population of NextBip and NextBip* relationship.
	BOOLEAN getRunNextBip(); **Description:** Returns true if we are populating NextBip and NextBip* relationship. Returns false otherwise.
	VOID populateNextBipAndNextBipStar(); **Description:** Compute all NextBip and NextBip* relationships.
	BOOLEAN isNextBip(**ProgLine** n1, **ProgLine** n2); **Description:** Returns true if NextBip(n1, n2) holds and the information is stored in the PKB, returns false otherwise.
	BOOLEAN isNextBipStar(**ProgLine** n1, **ProgLine** n2); **Description:** Returns true if NextBip*(n1, n2) holds and the information is stored in the PKB, returns false otherwise.
	BOOLEAN isNextBipStarWithBranchStack(**STRING** s1, **STRING** s2); **Description:** The string si is the concatenation of the ProgLine ni and the branch stack (branch in and branch back taken) of the path. Returns true if the ProgLine with the corresponding branch stack in s1 and the ProgLine with the corresponding branch stack in s2 has the NextBip relationship, and the information is stored in the PKB. Returns false otherwise.
	SET_OF_PROGLINES CONST &getNextBip(**ProgLine** n1) **CONST**; **Description:** returns the const reference to a set of n2's such that NextBip(n1, n2).
	SET_OF_PROGLINES CONST &getPreviousBip(**ProgLine** n2) **CONST**; **Description:** returns the const reference to a set of n1's such that NextBip(n1, n2).
	SET_OF_PROGLINES CONST &getNextBipStar(**ProgLine** n1) **CONST**; **Description:** returns the const reference to a set of n2's such that NextBip*(n1, n2).
	SET_OF_PROGLINES CONST &getPreviousBipStar(**ProgLine** n2) **CONST**; **Description:** returns the const reference to a set of n1's such that NextBip*(n1, n2).
	SET_OF_PROGLINES CONST &getNextBipWithBranchStack(**ProgLine** n1) **CONST**; **Description:** The string si is the concatenation of the ProgLine ni and the branch stack (branch in and branch back taken) of the path. Returns the const reference to a set of s2's such that NextBip(n1, n2).
	PAIR<LIST_OF_PROGLINES, LIST_OF_PROGLINES> getAllNextBip(); **Description:** Returns a pair of vectors in the nextBipMap. First vector is a vector of n1's. Second is a vector of n2's. For e.g., if (1,2), (3,4), (5,6) is in nextBipMap, then it will return <[1,3,5],[2,4,6]>
	PAIR<LIST_OF_PROGLINES, LIST_OF_PROGLINES> getAllNextBipStar(); **Description:** Returns a pair of vectors in the nextBipStarMap. First vector is a vector of n1's. Second is a vector of n2's. For e.g., if (1,2), (3,4), (5,6) is in nextBipStarMap, then it will return <[1,3,5],[2,4,6]>
	INT getNextBipSize(); **Description:** Returns the no. of pairs of NextBip relationship. E.g. if the nextBipMap has [1: {2,3}],

	then the size is 2, because there are pairs (1,2) and (1,3).
	INT getNextBipStarSize(); **Description:** Returns the no. of pairs of NextBip* relationship. E.g. if the nextBipStarMap has [1: {2,3}], then the size is 2, because there are pairs (1,2) and (1,3).
	MAP<STRING, SET_OF_STRINGS> getNextBipStarWithBranchStackNoDummyMap(); **Description:** Returns the nextBipStarWithBranchStackNoDummyMap.

AffectsBip API Overview: Next stores AffectsBip and AffectsBip* relationship information in the PKB.	
	API
	VOID setRunAffectsBip(**BOOLEAN** runAffectsBip); **Description:** To switch on (set runAffectsBip to true) and off population of AffectsBip and AffectsBip* relationship.
	BOOLEAN getRunAffectsBip(); **Description:** Returns true if we are populating AffectsBip and AffectsBip* relationship. Returns false otherwise.
	VOID populateNextBipAndNextBipStar(); **Description:** Compute all AffectsBip and AffectsBip* relationships.
	BOOLEAN isAffectsBip(**StmtNum** a1, **StmtNum** a2); **Description:** Returns true if AffectsBip(a1, a2) holds and the information is stored in the PKB, returns false otherwise.
	BOOLEAN isAffectsBipStar(**StmtNum** a1, **StmtNum** a2); **Description:** Returns true if AffectsBip*(a1, a2) holds and the information is stored in the PKB, returns false otherwise.
	SET_OF_STMT_NOS CONST &getAffectsBip(**StmtNum** a1) **CONST**; **Description:** returns the const reference to a set of a2's such that AffectsBip(a1, a2).
	SET_OF_STMT_NOS CONST &getAffectedBip(**StmtNum** a2) **CONST**; **Description:** returns the const reference to a set of a1's such that AffectsBip(a1, a2).
	SET_OF_STMT_NOS CONST &getAffectsBipStar(**StmtNum** a1) **CONST**; **Description:** returns the const reference to a set of a2's such that AffectsBip*(a1, a2).
	SET_OF_STMT_NOS CONST &getAffectedBipStar(**StmtNum** a2) **CONST**; **Description:** returns the const reference to a set of a1's such that AffectsBip*(a1, a2).
	PAIR<LIST_OF_STMT_NOS, LIST_OF_STMT_NOS> getAllAffectsBip(); **Description:** Returns a pair of vectors in the affectsBipMap. First vector is a vector of a1's. Second is a vector of a2's. For e.g., if (1,2), (3,4), (5,6) is in affectsBipMap, then it will return <[1,3,5],[2,4,6]>.
	PAIR<LIST_OF_STMT_NOS, LIST_OF_STMT_NOS> getAllAffectsBipStar();

	Description: Returns a pair of vectors in the affectsBipStarMap. First vector is a vector of a1's. Second is a vector of a2's. For e.g., if (1,2), (3,4), (5,6) is in affectsBipStarMap, then it will return <[1,3,5],[2,4,6]>.
	INT getAffectsBipSize(); **Description:** Returns the no. of pairs of AffectsBip relationship. E.g. if the affectsBipMap has [1: {2,3}], then the size is 2.
	INT getAffectsBipStarSize(); **Description:** Returns the no. of pairs of AffectsBip* relationship. E.g. if the affectsBipStarMap has [1: {2,3}], then the size is 2.

8.3 QP Abstract API

8.3.1 Clause Object

Clause API Overview: Clause Object stores information about a clause in the query.	
	API
	STRING getRel(): **Description**: For 'such that' clause, returns the relationship queried. For pattern clause, returns the assignment synonym queried. For with clause, returns an empty string.
	LIST_OF_STRINGS getArgs(): **Description**: Returns the arguments in the clause.
	SET_OF_STRINGS getSynonyms(): **Description**: Returns all synonyms used in the clause.
	INT getNumOfKnown(): **Description**: Returns the total number of known values in the clause.
	VOID replaceSynonym(**STRING** synonym**, STRING** replacement): **Description**: Replaces the synonym used in the arguments of the clause with the replacement string.

8.3.2 Query Object

Query API Overview: Query Object stores information about a query.	
	API
	MAP<STRING, STRING> getDeclarations(): **Description**: Returns all declarations of synonyms used in the query. Where the synonym of the query is the key and its design entity is the value.
	LIST_OF_STRINGS getToSelect(): **Description**: Returns the synonym in the Select clause.

	LIST<LIST_OF_CLAUSES> getClauses(): **Description**: Returns a vector containing vectors of clauses where the vectors of clauses are clauses that are grouped.
	LIST<SET_OF_STRINGS> getSynonyms(): **Description**: Returns a vector containing unordered sets of all synonyms used in the groups of clauses in the query.
	BOOLEAN getIsSyntacticallyValid(): **Description**: Returns true if the query is syntactically valid, false otherwise.
	BOOLEAN getIsSemanticallyValid(): **Description**: Returns true if the query is semantically valid, false otherwise.
	VOID setClauses(**LIST<LIST_OF_CLAUSES>** clauses): **Description**: Sets the clauses in the Query object.
	VOID setClausesAtIdx(**LIST_OF_CLAUSES** clauses, **INT** idx): **Description**: Sets a vector of clauses at the specified index in the group of clauses stored in the Query object.
	VOID setSynonyms(**LIST<SET_OF_STRINGS>** synonyms): **Description**: Sets the synonyms in the Query object.

8.3.3 Query Utility

QueryUtility API Overview: Query Utility provides general methods used by classes in Query Processor.	
	API
	BOOLEAN checkNameWithQuotes(**STRING** s): **Description**: Checks if a name in quotes conforms to naming standards, returns true if it is valid and false otherwise.
	BOOLEAN checkExpression(**STRING** s): **Description**: Checks if an expression is valid as defined in PQL grammar, returns true if it is valid and false otherwise.
	BOOLEAN checkExpressionWithUnderscore(**STRING** s): **Description**: Checks if an expression with underscores in its front and back is valid as defined in PQL grammar, returns true if it is valid and false otherwise.
	BOOLEAN checkSynonymDeclared(**STRING** synonym, **MAP<STRING, STRING>** declarations): **Description**: Checks if a synonym has been declared in a query before, returns true if it is has and false otherwise.
	BOOLEAN isOperator(**TOKENTYPE** tokenType): **Description**: Checks if a token is an operator (i.e. plus, minus, times, divide, mod), returns true if it is and false otherwise.
	STRING getArgType(**STRING** synonym, **MAP<STRING, STRING>** declarations): **Description**: Takes in a synonym and gets its type as declared in the query. Returns its type if it

	has been declared, returns an empty string otherwise.
	LIST_OF_INTS selectAll(**STRING** synonymType): **Description**: Takes in a synonym type, returns all possible statement numbers for 'stmt' | 'read' | 'print' | 'call' | 'while' | 'if' | 'assign' synonym types and all possible IDs for ''variable' | 'constant' | 'procedure' synonym types. Returns an empty vector if synonym type does not exist.
	BOOLEAN intersectSingleSynonym(**LIST_OF_INTS** allResults, **LIST_OF_INTS** allCorrectType, **LIST_OF_INTS** &results): **Description**: Stores the intersection of allResults and allCorrectType in results, removes duplicates. Returns true if the results vector is non-empty, false otherwise.
	BOOLEAN intersectSingleSynonym(**SET_OF_INTS** allResults, **LIST_OF_INTS** allCorrectType, **LIST_OF_INTS** &results): **Description**: Stores the intersection of allResults and allCorrectType in results, removes duplicates. Returns true if the results vector is non-empty, false otherwise. This is an overloaded function, where allResults can be an unordered set instead of a vector.
	BOOLEAN intersectDoubleSynonym(**PAIR<LIST_OF_INTS, LIST_OF_INTS>** allResults, **PAIR<LIST_OF_INTS, LIST_OF_INTS>** allCorrectType, **PAIR<LIST_OF_INTS, LIST_OF_INTS>&** results): **Description**: Stores pairs of entries of allResults such that the first entry exists in allCorrectType.first and the second entry exists in allCorrectType.second in results. Returns true if the results vector is non-empty, false otherwise.
	VOID project(**SET_OF_STRINGS** toProject, **MAP<STRING, LIST_OF_INTS>&** results): **Description**: Removes duplicates from results, filters results to project only synonyms listed in toProject.
	INT getSize(**CLAUSE** clause, **MAP<STRING, STRING>** declarations): **Description**: Returns the size of the table stored in the PKB for the relationship specified in the clause.

8.3.4 Query Preprocessor

QueryPreprocessor API Overview: Query Preprocessor does validation checks on the input query string and stores it into a Query Object.	
	API
	Query process(**STRING** query): **Description**: Calls private methods in the class to validate and splits the input query string into its subcomponents, creates a Query Object and returns it.

8.3.5 Query Optimizer

QueryOptimizer API Overview: Query Optimizer optimizes Query object for evaluation by rewriting clauses using information in with clauses, dividing clauses into multiple groups, sorting groups for evaluation and sorting clauses inside each group.

	API
	VOID setIsRewriteClauses(**BOOLEAN** isRewriteClauses): **Description**: Enables/Disables the rewriting of clauses if the isRewriteClauses boolean is set to true/false respectively.
	VOID setIsGroup(**BOOLEAN** isGroup): **Description**: Enables/Disables the grouping of clauses if the isGroup boolean is set to true/false respectively.
	VOID setIsOrderGroups(**BOOLEAN** isOrderGroups): **Description**: Enables/Disables the ordering of group of clauses if the isOrderGroups boolean is set to true/false respectively.
	VOID setIsOrderClauses(**BOOLEAN** isOrderClauses): **Description**: Enables/Disables the ordering of clauses within each group if the isOrderClauses boolean is set to true/false respectively.
	QUERY optimize(**QUERY** query): **Description**: Calls private methods in the class to optimize the clauses in the Query Object and returns the optimized Query Object.

8.3.6 Query Evaluator

QueryEvaluator API Overview: Query Evaluator evaluates a Query Object.	
	API
	LIST_OF_STRINGS evaluate(**QUERY** query): **Description**: Calls private methods in the class and methods in helper classes to evaluate a query. Returns a list of results. Returns an empty list if the query is invalid.

8.4 API Discovery Process

procedure main {	assign a; stmt s; variable v;

```
procedure main {
1.   read x;
2.   read y;
3.   while (y != 0) {
4.       x = x / y;
5.       read y; }
6.   print x; }
```

1	Select a pattern a("x", _"y"_)
2	Select s such that Follows (1, s)
3	Select s such that Follows (s, 3)
4	Select s such that Follows* (1, s)
5	Select s such that Follows* (s, 3)
6	Select s such that Parent (3, s)
7	Select s such that Parent (s, 5)
8	Select s such that Parent* (3, s)
9	Select s such that Parent* (s, 5)
10	Select v such that Modifies (1, v)

	1 1	`Select a such that Modifies (a, "x")`

8.4.1 How the Parser works with ProcTable, VarTable, ConstTable and StmtTable

- At the start of the procedure, the parser will call `storeProcName("main")`.

- At statement 1, the parser will call `storeVarName("x")` and retrieve the varID. It will also also call `storeStmt(1, ASTNode, "read")`.

- At statement 2, the parser will call `storeVarName("y")` and retrieve the varID. It will also also call `storeStmt(2, ASTNode, "read")`.

- At statement 3, since "y" is used in the condition of the while-loop, the parser will call `storeVarName("y")`. Since the variable "y" already exists in the varTable, the variable will not be stored again, but the varID of "y" stored previously will be returned. Furthermore, since 0 is used in the condition of the while-loop, the parser will also call `storeConst("0")`. The parser will also call `storeStmt(3, ASTNode, "while")`.

- At statement 4, since "x" and "y" appear in the assignment statement, the parser will call `storeVarName("x")` and `storeVarName("y")` . Since both variables already exist in the varTable, the variables will not be stored again, but the varIDs stored previously will be returned. The parser will also call `storeStmt(4, ASTNode, "assign")`.

- At statement 5, the parser will call `storeVarName("y")`. Since the variable "y" already exists in the varTable, the variable will not be stored again, but the varID of "y" stored previously will be returned. It will also also call `storeStmt(5, ASTNode, "read")`.

- At statement 6, the parser will call `storeVarName("x")`. Since the variable "x" already exists in the varTable, the variable will not be stored again, but the varID of "x" stored previously will be returned. The parser will also call `storeStmt(6, ASTNode, "print")`.

8.4.2 How the Parser works with Follows and Parents

- Upon reaching the closing braces for the while-loop, the parser will call `storeFollows(s1, s2)` for every statement within the statement list, i.e., statement 4 to statement 5. It also calls `storeParent(3, child)` for every child within the statement list.

- Upon reaching the closing braces for the procedure, the parser will call `storeFollows(s1, s2)` for every statement within the statement list, i.e., statement 1 to statement 6.

8.5 Sample Test Cases

basicFullSimple_source.txt

```
procedure Honeysuckle {
    print teacup0;
    lesion = ((23 + 1) / bluebell / 5);

    while (melody <= maison) {
       x = y - 1;
       lesion = teacup0;
     }

    if (gossamer == lesion) then {
        x = lesion * 2;
    } else {
        x = (lesion * 2) - 1;
        print x;
    }

    print lesion;
    call something;
    call shikashi;
}

procedure something {
    read teacup0;
    print something;

    while (5 == 5) {
        x = y - 1;
        lesion = teacup0;
        lesion = ((23 + 1) / bluebell / 5);
        print lesion;
        call shikashi;
     }

    melody = coeur - cerveau;
}

procedure shikashi {
    z = 2 + 8;

    if (boro != 1) then {
        read ardent;
        z = 258080 / 56;
        guru = guru;
    } else {
        read pardon;
    }
}
```

complexFullSimple_source.txt

```
procedure f00d {
      read rice;
      noodle = chicken + noodle;
```

```
        call obj3ct;
        chickenRice = chicken + rice * chi11i;
        duck = 2;
        print chickenRice;
        if ((duck > chicken)    &&(0/(2/4)*6+noodle == 10%ramen)) then {
                ramen = noodle + sprING000Ni0n;
                onion = 0 + 4 + duck + 0 + 4;
                while (onion != sprING000Ni0n) {
                        sprING000Ni0n = spring + 3*0;
                        call colors;
                        read sprING000Ni0n;
                        noodle = sprING000Ni0n;
                }
                sprING000Ni0n = sprING000Ni0n * sprING000Ni0n / spring;
        } else {
                spring = ((((2)))) + (4 * 6 - ((2 + 2)/0))    % 3;
                if (rice != noodle) then {
                        print chi11i;
                        duck = chickenRice + ramen;
                } else {
                        call obj3ct;
                        rice = chi11i * 4;
                }
        }
        chickenRice = duck + 4;
        print duck;
        call simple;
}

procedure colors {
        purple = red + blue;
        red1 = 2;
        red2 = 3;
        red = purple;
        if ((red1) == red2) then {
                spring = green;
                while ((!((!(green > yellow)) || (!(blue2 == blue2)))) && ((234 +
(((456 * 789)))) <= (blue1 + green))) {
                        if ((234 + 234 + ((234 - (789 - 789)))) != 456) then {
                                red = red * red;
                        } else {
                                read purple;
                        }
                        red = blue;
                }
                purple = purple;
                while ((red1 > blue3)&&((red2 < blue2) || (!(purple + spring !=
green1)))) {
                        blue = blue - blue;
                        blue = blue;
                }
        } else {
                purple = purple;
        }
        yellow = green + red * blue1;
        black = 7 + blue;
        white = green2 - green2;
        while (yellow > 3) {
                if (green < 5) then {
                        if (!(purple == 9)) then {
                                while ((blue3 == blue2) || (red != ramen)) {
                                        blue2 = blue2 % red2;
                                        print purple;
                                        red2 = (((884-197)+(gold *
169))-((black-black)+3993   *black) + (((black)/(red))+(((((75105820  %9749)/
(3%21))*(red*(gold+black)))/gold)))%((red*red-yellow  %blue)+(blue*  93238-
926535897+926535897));
```

```
                            }
                            purple = purple;
                    } else {
                            call simple;
                            if (red != red1) then {
                                    grey = black / yellow;
                                    while (7 > 234) {
                                            while (456 * 2 == 789) {
                                                    blue = purple - red;
                                                    while ((!(((green) - 2)) >=
((456)*3) - 2)) && (((green1 < 234) || (green2 == green)) ||(!(3*4*(5 - 6) <=
((red1) * (((red2))))))) {
                                                            if (3 * 4 == 12) then{
                                                                    green = yellow;
                                                                    read green1;
                                                            } else {
                                                                    while (green1 == 1) {
                                                                            print rice;
                                                                            rice = 500;
                                                                    }
                                                            }
                                                    }
                                            }
                                            blue = ((yellow
+orange-314/15)*((blue)+((926535897+orange))))*((red*((red))-(yellow%
blue))+((blue*((93238)))-    926535897     +
926535897))-((46/26)+((green)-26433)*((gold + 8)+((3)-silver))%(((3 *gold)*
(3279502))%((blue)*(red/88))));
                                    }
                            } else {
                                    print black;
                                    read white;
                            }
                    }
            } else {

                    call simple;
                    black = black % white /grey;
                    grey = purple + red * blue / (blue1 - blue2-blue3)%(red1
*red2);
            }
        }
      progline = progline;
      if ((procedure > statement) || (((((progline) == (statement))) && (print >
read))) then {
              then = assign;
              print = call - black;
              progline = 5555;
      } else {
              read read;
              print print;
              progline = statement * 100;
      }
      assign = progline;
}

procedure obj3ct {
      sci333ors = 290918238;
      Kn17e = 424809 + w9113t;
      if (w9113t > 1) then {
              Kn17e = 1241242134;
              call colors;
      } else {
              call t0pUp;
      }
      sci333ors = Kn17e;
```

```
}

procedure simple {
      while (while != if) {
            print print;
            read read;
            call = assign + procedure;
      }
      yellow = 0;
      if (!(then - else == 3)) then {
            if = then + else;
      } else {
            call procedure;
            Kn17e = else;
      }
}

procedure t0pUp {
      money = money + earn - used;
      read Kn17e;
      call simple;
}

procedure procedure {
      print else;
}
```

CyclicalSimple_source.txt

```
procedure Test1 {
  read a;
  if (b==1) then {
    z = x / 1;
    call Test2a;
  } else {
    call Test2b;
  }

  a = a + 4;


  print x;
  print y;
}

procedure Test2a {
  call Test3;
}

procedure Test2b {
  call Test3;
}

procedure Test3 {
  while (x >= z) {
    call Test4;
  }
  call Test5;
}

procedure Test4 {
  call Test5;
}
```

```
procedure Test5 {
  x = x + 1;
  call Test2b;
}
```

8.6 Code of setupQe()

```
class StmtNodeStub : public ast::Stmt {
public:
   StmtNodeStub(int index): ast::Stmt(new sp::Token(), index){};
};

void setupQe() {
   PKB::varTable = new VarTable();
   PKB::varTable->storeVarName("count"); // 0
   PKB::varTable->storeVarName("cenX"); // 1
   PKB::varTable->storeVarName("cenY"); // 2
   PKB::varTable->storeVarName("x"); // 3
   PKB::varTable->storeVarName("y"); // 4
   PKB::varTable->storeVarName("flag"); // 5
   PKB::varTable->storeVarName("normSq"); // 6

   PKB::procTable = new ProcTable();
   PKB::procTable->storeProcName("computeCentroid"); // 0
   PKB::procTable->storeProcName("readPoint"); // 1
   PKB::procTable->storeProcName("randomProcName"); // 2

   PKB::constTable = new ConstTable();
   PKB::constTable->storeConst("0");
   PKB::constTable->storeConst("1");

   PKB::stmtTable = new StmtTable();
   ast::Stmt* stmtNodeStub = new StmtNodeStub(0);
   PKB::stmtTable->storeStmt(1, stmtNodeStub, ASSIGN_);
   PKB::stmtTable->storeStmt(2, stmtNodeStub, ASSIGN_);
   PKB::stmtTable->storeStmt(3, stmtNodeStub, ASSIGN_);
   PKB::stmtTable->storeStmt(4, stmtNodeStub, CALL_);
   PKB::stmtTable->storeStmt(5, stmtNodeStub, WHILE_);
   PKB::stmtTable->storeStmt(6, stmtNodeStub, ASSIGN_);
   PKB::stmtTable->storeStmt(7, stmtNodeStub, ASSIGN_);
   PKB::stmtTable->storeStmt(8, stmtNodeStub, ASSIGN_);
   PKB::stmtTable->storeStmt(9, stmtNodeStub, CALL_);
   PKB::stmtTable->storeStmt(10, stmtNodeStub, IF_);
   PKB::stmtTable->storeStmt(11, stmtNodeStub, ASSIGN_);
   PKB::stmtTable->storeStmt(12, stmtNodeStub, ASSIGN_);
   PKB::stmtTable->storeStmt(13, stmtNodeStub, ASSIGN_);
   PKB::stmtTable->storeStmt(14, stmtNodeStub, ASSIGN_);
   PKB::stmtTable->storeStmt(15, stmtNodeStub, PRINT_);
   PKB::stmtTable->storeStmt(16, stmtNodeStub, PRINT_);

   PKB::stmtTable->storeAssignExpr(1, "count", "(0)");
   PKB::stmtTable->storeAssignExpr(2, "cenX", "(0)");
   PKB::stmtTable->storeAssignExpr(3, "cenY", "(0)");
   PKB::stmtTable->storeAssignExpr(6, "count", "((count) + (1))");
   PKB::stmtTable->storeAssignExpr(7, "cenX", "((cenX) + (x))");
   PKB::stmtTable->storeAssignExpr(8, "cenY", "((cenY) + (y))");
   PKB::stmtTable->storeAssignExpr(11, "flag", "(1)");
   PKB::stmtTable->storeAssignExpr(12, "cenX", "((cenX) / (count))");
   PKB::stmtTable->storeAssignExpr(13, "cenY", "((cenY) / (count))");
    PKB::stmtTable->storeAssignExpr(14, "normSq", "(((cenX) * (cenX)) + ((cenY) *
(cenY)))");

   PKB::stmtTable->storePrintVariableForStmt(15, 1);
   PKB::stmtTable->storePrintVariableForStmt(16, 2);
```

```
PKB::stmtTable->storeIfPattern(10, 0);
PKB::stmtTable->storeIfPattern(10, 3);
PKB::stmtTable->storeIfPattern(10, 4);
PKB::stmtTable->storeIfPattern(10, 2);
PKB::stmtTable->storeWhilePattern(5, 5);
PKB::stmtTable->storeWhilePattern(5, 3);
PKB::stmtTable->storeWhilePattern(5, 4);

PKB::follows = new Follows();
PKB::follows->storeFollows(1, 2);
PKB::follows->storeFollows(2, 3);
PKB::follows->storeFollows(3, 4);
PKB::follows->storeFollows(4, 5);
PKB::follows->storeFollows(5, 10);
PKB::follows->storeFollows(6, 7);
PKB::follows->storeFollows(7, 8);
PKB::follows->storeFollows(8, 9);
PKB::follows->storeFollows(10, 14);
PKB::follows->storeFollows(12, 13);
PKB::follows->storeFollows(14, 15);
PKB::follows->storeFollows(15, 16);
PKB::follows->populateFollowsStar();

PKB::parent = new Parent();
PKB::parent->storeParent(5, 6);
PKB::parent->storeParent(5, 7);
PKB::parent->storeParent(5, 8);
PKB::parent->storeParent(5, 9);
PKB::parent->storeParent(10, 11);
PKB::parent->storeParent(10, 12);
PKB::parent->storeParent(10, 13);
PKB::parent->populateParentStar();

PKB::uses = new Uses();
PKB::uses->storeStmtUses(5, 3);
PKB::uses->storeStmtUses(5, 4);
PKB::uses->storeStmtUses(5, 0);
PKB::uses->storeStmtUses(5, 1);
PKB::uses->storeStmtUses(5, 2);
PKB::uses->storeStmtUses(6, 0);
PKB::uses->storeStmtUses(7, 1);
PKB::uses->storeStmtUses(7, 3);
PKB::uses->storeStmtUses(8, 2);
PKB::uses->storeStmtUses(8, 4);
PKB::uses->storeStmtUses(10, 0);
PKB::uses->storeStmtUses(10, 1);
PKB::uses->storeStmtUses(10, 2);
PKB::uses->storeStmtUses(12, 1);
PKB::uses->storeStmtUses(12, 0);
PKB::uses->storeStmtUses(13, 2);
PKB::uses->storeStmtUses(13, 0);
PKB::uses->storeStmtUses(14, 1);
PKB::uses->storeStmtUses(14, 2);
PKB::uses->storeStmtUses(15, 1);
PKB::uses->storeStmtUses(16, 2);
PKB::uses->storeProcUses(0, 0);
PKB::uses->storeProcUses(0, 1);
PKB::uses->storeProcUses(0, 2);
PKB::uses->storeProcUses(0, 3);
PKB::uses->storeProcUses(0, 4);

PKB::modifies = new Modifies();
PKB::modifies->storeStmtModifies(1, 0);
PKB::modifies->storeStmtModifies(2, 1);
PKB::modifies->storeStmtModifies(3, 2);
PKB::modifies->storeStmtModifies(4, 3);
PKB::modifies->storeStmtModifies(4, 4);
```

```
    PKB::modifies->storeStmtModifies(5, 0);
    PKB::modifies->storeStmtModifies(5, 1);
    PKB::modifies->storeStmtModifies(5, 2);
    PKB::modifies->storeStmtModifies(5, 3);
    PKB::modifies->storeStmtModifies(5, 4);
    PKB::modifies->storeStmtModifies(6, 0);
    PKB::modifies->storeStmtModifies(7, 1);
    PKB::modifies->storeStmtModifies(8, 2);
    PKB::modifies->storeStmtModifies(9, 3);
    PKB::modifies->storeStmtModifies(9, 4);
    PKB::modifies->storeStmtModifies(10, 5);
    PKB::modifies->storeStmtModifies(10, 1);
    PKB::modifies->storeStmtModifies(10, 2);
    PKB::modifies->storeStmtModifies(11, 5);
    PKB::modifies->storeStmtModifies(12, 1);
    PKB::modifies->storeStmtModifies(13, 2);
    PKB::modifies->storeStmtModifies(14, 6);
    PKB::modifies->storeProcModifies(0, 0);
    PKB::modifies->storeProcModifies(0, 1);
    PKB::modifies->storeProcModifies(0, 2);
    PKB::modifies->storeProcModifies(0, 3);
    PKB::modifies->storeProcModifies(0, 4);
    PKB::modifies->storeProcModifies(0, 5);
    PKB::modifies->storeProcModifies(0, 6);

    PKB::next = new Next();
    PKB::next->storeNext(1, 2);
    PKB::next->storeNext(2, 3);
    PKB::next->storeNext(3, 4);
    PKB::next->storeNext(4, 5);
    PKB::next->storeNext(5, 6);
    PKB::next->storeNext(6, 7);
    PKB::next->storeNext(7, 8);
    PKB::next->storeNext(8, 9);
    PKB::next->storeNext(9, 5);
    PKB::next->storeNext(5, 10);
    PKB::next->storeNext(10, 11);
    PKB::next->storeNext(10, 12);
    PKB::next->storeNext(12, 13);
    PKB::next->storeNext(13, 14);
    PKB::next->storeNext(11, 14);
    PKB::next->storeNext(14, 15);
    PKB::next->storeNext(15, 16);
    PKB::next->populateNextStar();

    PKB::calls = new Calls();
    PKB::calls->storeCalls(4, 0, 1);
    PKB::calls->storeCalls(9, 1, 2);
    PKB::calls->processCalls();

    PKB::affects = new Affects();
    PKB::affects->populateAffectsAndAffectsStar();
}
```