School of Computing

CS3203 Software Engineering Project

AY20/21 Semester 2

Iteration 3 Project Report

Team 32

Team Members	Matriculation No.	Email
Chia Wei Hao David	A0183642A	davidchia@u.nus.edu
Teo Jun Hong	A0183984L	e0310779@u.nus.edu
Le Quang Tuan	A0184537W	e0313526@u.nus.edu
Tran Tan Phat	A0170740M	e0196695@u.nus.edu
Zhao Huan	A0211628L	e0496106@u.nus.edu
Bruce Ong	A0182333L	e0309128@u.nus.edu

Consultation Hours: 15:00 – 16:00, Wednesday

Tutor(s): Aaron Ong

Table of Contents

1 Scope

	Programs Parsed	Relationships Answered	Clauses handled	Selected from query
Iteration 1	✔ SIMPLE parser ✔ Design extraction	✔ Follows/Follows* ✔ Parent/Parent* ✔ Uses ✔ Modifies	✔ Select ✔ Such that ✔ Pattern (assign)	✔ Statement ✔ Variable ✔ Procedure ✔ Constant
Iteration 2	✔ multiple procedures	✔ Uses (proc) ✔ Modifies (proc) ✔ Calls/Calls* ✔ Next/Next*	✔ Pattern (container)	✔ Prog_line
Iteration 3		✔ Affects/Affects* ✔ NextBip/NextBip* ✔ AffectsBip	✔ With	✔ Tuple ✔ Boolean ✔ Attribute

As of iteration 3, our Static Program Analyzer (SPA) has met all the advanced requirements, and some of the extended requirements. Specifically, our program is able to:

1.1 Source Processor

- Validate a SIMPLE source code both syntactically and semantically.
    - Identifies the first location that has a syntax error. This helps the user to know where the program goes wrong.
    - Identifies any semantic errors such as: Calling non-existing procedures, cyclic calls of different procedures.
- Extract information about entities:
    - Statements: It identifies all statements and sets their types properly.
    - Variables: It identifies all variables used and modified in the program.
    - Constants: A list of constants is also extracted.
- Extract relationships between entities while parsing:
    - Direct *parent-children* relationships between statements.
    - Direct *follows* relationships between statements inside one statement list.
    - Direct *uses* relationships between statements and variables.
    - Direct *modifies* relationships between statements and variables.
    - All expressions that belong to assignment statements.

- *Control variables* of container statements.irect *next* relationships between statements.
- All the indirect relationships such as affect(*), next(*), calls(*), uses, modifies.
- Extensions such as nextBip, nextBip(*), affectBip, affectBip(*)

1.2 PKB

- Insert and store information extracted by the design extractor:
  - Entities, including all properties about statements, variables and constants.
  - Direct and indirect relationships between entities, including *parent(*)*, *follows(*)*, *uses, modifies*, *calls(*)*, *next(*)*, *affect(*)*, *nextBip(*)* and *affectBip*.
  - *Expressions* of assignment statements, *control variables* of container statements.
- Retrieve results for no-clause or single-clause PQL queries:
  - Select clauses
  - Single such that clauses
  - Single assign pattern clauses
  - Single container pattern clauses
  - Single with clauses

1.3 Query Processor

- Validate and answer queries in Program Query Language (PQL).
- Perform semantic validation and terminate query execution for invalid PQL queries.
- As of Iteration 3, the types of queries that can be evaluated by our SPA correspond to that of the basic and advanced PQL requirements:
  - A valid PQL Query must consist of one **Select** clause, with any number and combination of design-entity relations that might be filtered by:
    - Any number of **such that** clauses
    - Any number of **assign/while/if pattern** clauses
    - Any number of **with** clauses
  - Valid entities to select in the **Select** clause are:
    - Boolean
    - Tuples
    - Synonym
    - Synonym with attribute name
  - The valid relations to a query in a **such that** clause consists of at most one of
    - Follows /*

        - Parent /*
        - Next /*
        - Uses - (Statement/Procedure)
        - Modifies - (Statement/Procedure)
        - Calls/*
        - Affects/*
    - Expressions in **assign-pattern** clauses only contain constants or variable names. Valid assign-pattern clauses are
        - Partial match of an expression
        - Full match of an expression
        - Match any expression (“_”)
    - Valid **while-pattern** or **if-pattern** clauses perform a filter on the variables in the condition of the while/if statement by:
        - A variable synonym
        - A variable literal
        - _ (wildcard all variable) match
    - Valid **with** clauses consist of comparison between any two of the following:
        - Synonym
        - Literal
        - Attribute Ref which is syn.attrName where attrName is allowed to be:
            - value
            - procName
            - varName
            - stmt#
    - Clauses can be separated by the keyword “and” provided that the clauses are of the same clause type (**such that/pattern/with**)

2 Development Plan

This section describes how our team allocated our time and manpower for the project and iteration 1-2, together with our test plan.

Our team has split the project into 3 main components - Source processor, PKB and Query Processor - and assigned team members accordingly to how much work we estimated each component would require. Additionally, we had a dedicated system tester who was initially involved in the planning and design stage of the PKB but has the main responsibility of writing system tests for our SPA.

Source Parser	PKB
● Le Quang Tuan ● Tran Tan Phat	● Zhao Huan ● Jun Hong (Design and planning)
Query Processor	**System Tests**
● David Chia ● Bruce Ong (Team Leader)	● Jun Hong

2.1 Project Plan

Iteration 1

Tasks/ Components	Week 1	Week 2	Week3	Week 4
Environment	Cross platform set up	Set up CI		
Source Processor	Understand SIMPLE grammars and project requirements	Research on parsing algorithms	Design subcomponents and respective APIs	Implement a parser without condition expressions
			Implement tokenizer	

PKB		Design internal data structures	Design concrete PKB APIs used by SIMPLE Parser and Query Evaluator	Set up data structures
				implement insertion and simple retrieval methods
Query Evaluator		Design Query internal representation	Design Query Evaluator APIs used by PQL Parser	Implement QueryEvaluator for selection of single synonym
			Design evaluated results data structure	Implement ResultsProjector
Query Preprocessor		Research on parsing algorithms	Design and implement the Tokenizer (Lexer)	Write Unit Tests for Tokenizer
				Implement the Parser
Testing			Design Unit Testing for all components	Design test cases for system testing. Perform Unit Testing
Documentation			APIs	

Activities/ Components	**Week 5**	**Week 6**	**Recess**
Source Processor	Implement parser for condition expressions	Write more unit test cases	
	Implement Design Extractor		

	Write unit/integration tests	Help in system integration	
PKB	Implement more complex result retrieval methods	Write more unit test cases	
	Implement an extractor for indirect relationships with unit tests	Help in system integration	
Query Evaluator	Implement QueryEvaluator for such that clause	Implement QueryEvaluator for pattern clause	
	Implement natural join and cartesian product algorithms		
Query Preprocessor	Finish implementing Parser with calls to Query Evaluator API	Write more integration tests for Parse	
	Write integration tests for Parser	Add checks for overlooked semantic errors	
Testing	Integration Testing for Query Preprocessor + Query Evaluator	System Testing	
	Integration Testing for PKB + Query Preprocessor + Query Evaluator		
	Integration Testing for PKB + Parser		
Documentation			Work on report

Iteration 2

Activities/ Components	**Week 8**	**Week 9**	**Week 10**
SourceProcessor	Implement Calls, Calls(*)	Implement Uses(), Modifies(), Affect, Affect(*)	
	Semantic checking (cyclic calls, ..)	Implement full-pattern matching	

	Implement Next, Next(*)	Unit/integration tests	
PKB	PKB API overhaul	Implement pattern of container statements	
		Implement Next(*)	
	Implement Calls(*), and Uses / Modifies (p)	Implement full pattern of assign statements	
Query Evaluator	Refactor query evaluation code flow due to PKB API overhaul	Implement Calls(*) and Next(*)	
	Implement container (if/while) pattern		
Query Preprocessor	Implement Calls(*), Next(*), “and”, as well as any number of “pattern” and “such that” clauses in any order	Implement (if/while) pattern and (assign) pattern with complicated expressions	Refactored code: moved semantic checking logic to QueryParserErrorUtility class
	Write unit tests for Tokenizer	Write unit and integration tests for QueryParser	
Testing	Write system test cases	Integration testing for Query Preprocessor + Query Evaluator	System testing
		Integration testing for PKB + PQL	
		Integration testing for PKB + Parser	
Documentation	Refine report draft from iteration 1	Work on project report and extension	

Iteration 3

Activities/ Components	Week 11	Week 12	Week 13
SourceProcessor	Implement NextBip	Write unit tests for DE	
	Implement AffectBip	Write integration tests with PKB and QE	
	Implement NextBip(*)		
PKB	Design API and DS used for with clauses	Implement methods related to with clauses	
	Implement methods for new relationships	Write unit and integration tests for PKB	
Query Evaluator	Implement with clauses and attributes	Implement QE Optimization	
	Implement Select Boolean and Tuple	Write unit and integration tests for QueryEvaluator	
	Implement Affects(*), NextBip(*) and AffectsBip(*)	Stress testing for QueryEvaluator Optimization	
Query Preprocessor	Implement Affects(*), NextBip(*) and AffectsBip(*)	Write unit and integration tests for QueryParser	
	Implement With-clause		
	Implement Select BOOLEAN and Select tuple	Write unit tests for Tokenizer	
Testing	Write system test cases	Integration testing for Query Preprocessor + Query Evaluator	Integration testing for PKB + Parser
		Integration testing for PKB + PQL	System Testing
Documentation		Refine report draft from iteration 2	Work on the final project report

2.2 Iteration 1 Activities

Week	Activity	Tuan	Phat	Zhao Huan	Jun Hong	David	Bruce
1	Cross platform Set Up	✔	✔	✔	✔	✔	✔
	Set up CI			✔			
	Set up Trello					✔	✔
	Decide on coding standard	✔	✔	✔	✔	✔	✔
2	Design PKB internal data structures			✔	✔		
	Design internal representation of Query					✔	
	Write unit tests for query evaluator					✔	
3	Design SIMPLE Token and Tokenizer API	✔	✔				
	Implement SIMPLE Tokenizer and related classes		✔				
	Design concrete PKB API used by QE			✔	✔	✔	
	Design concrete PKB API used by Parser	✔		✔	✔		
	Design Token and Tokenizer API (Query Preprocessor or QPP)						✔
	Implement Query Preprocessor Tokenizer and related classes						✔

4	Implement Parser for without handling conditional expressions	✔					
	Implement PKB internal data structures			✔			
	Implement insertion methods of relationships / types + unit tests			✔			
	Implement getBoolean methods of relationships + unit tests			✔			
	Implement query evaluator for select					✔	
	Implement results projector + unit tests					✔	
	Implement QueryParser for declaration, select, pattern and such that clauses						✔
	Write unit tests for QPP Tokenizer						✔
	Design of SIMPLE Source Code				✔		
5	Implement Parser for conditional clause		✔				
	Design and Implement Design Extractor	✔	✔				
	Implement complex get methods			✔			
	Implement insertion and retrieval of factor assign patterns + unit tests			✔			
	Implement methods to extract indirect relationships + unit tests			✔			
	Implement ResultUtil (Natural join and Cartesian Product) + Unit tests					✔	

	Implement query evaluator for **such that**					✔	
	Implement query evaluator for **pattern**					✔	
	Implement QueryParser for multiple declaration under same entity type (e.g. variable v1, v2, v3) + Integration tests						✔
	Implement QueryParser for storing all the information about clauses in Query object (using query evaluator API)						✔
	Design of PQL Queries				✔		
6	Parser unit testing	✔	✔				
	Integration testing: query evaluator + query preprocessor					✔	✔
	Integration testing: QE + PKB			✔		✔	
	Integration testing: Parser + PKB	✔	✔	✔			
	System testing	✔	✔	✔	✔	✔	✔

2.3 Iteration 2 Activities

Week	Activity	Tuan	Phat	Zhao Huan	Jun Hong	David	Bruce
	Implement Call, Call(*)	✔					
	Implement Next, Next(*)		✔				

8	Extraction of advanced relationship such as uses/modifies of procedures	✔					
	Full expression matching	✔					
	Split the PKB API for data retrieval and add new methods for container pattern			✔		✔	
	Update PKB API for insertion			✔			
	Implement uses/modifies (p), calls(*)			✔			
	Refactor query evaluation code due to split in PKB API for data retrieval					✔	
	Refactor Pattern Clause class to support container (if/while) pattern clauses					✔	
	Implemented evaluation for container pattern clauses					✔	
	Implement Query Parser for uses/modifies (p), calls(*), next(*), ‘and’, as well as multiple ‘such that’ and ‘pattern’ clauses in any order						✔
	Design of SIMPLE Code and PQL queries				✔		
9	Implement Uses(), Modifies()	✔					
	Implement Affect, Affect(*)		✔				
	Unit Testing on Design Extractor	✔	✔				
	Implement full assign pattern	✔		✔			✔

	Implement next(*) relationship			✔			
	Implement container pattern			✔			
	Added Call(*) and Next(*) relationships to relationship types					✔	
	Added Prog_line to entity types					✔	
	Implement Query Parser for pattern-if, pattern-while, also pattern-assign with complicated expression using shared library with SIMPLE Parser						✔
	Refactor code: Stored all tables in EntitiesTable class such as valid entity types in left and right side of relRef						
	Integration testing: query evaluator + query preprocessor					✔	✔
	Integration testing: QE + PKB			✔		✔	
	Integration testing: Parser + PKB	✔	✔	✔			
	Design of PQL Queries				✔		
	Finalize and work on report of extension	✔	✔				
10	System testing	✔	✔	✔	✔	✔	✔

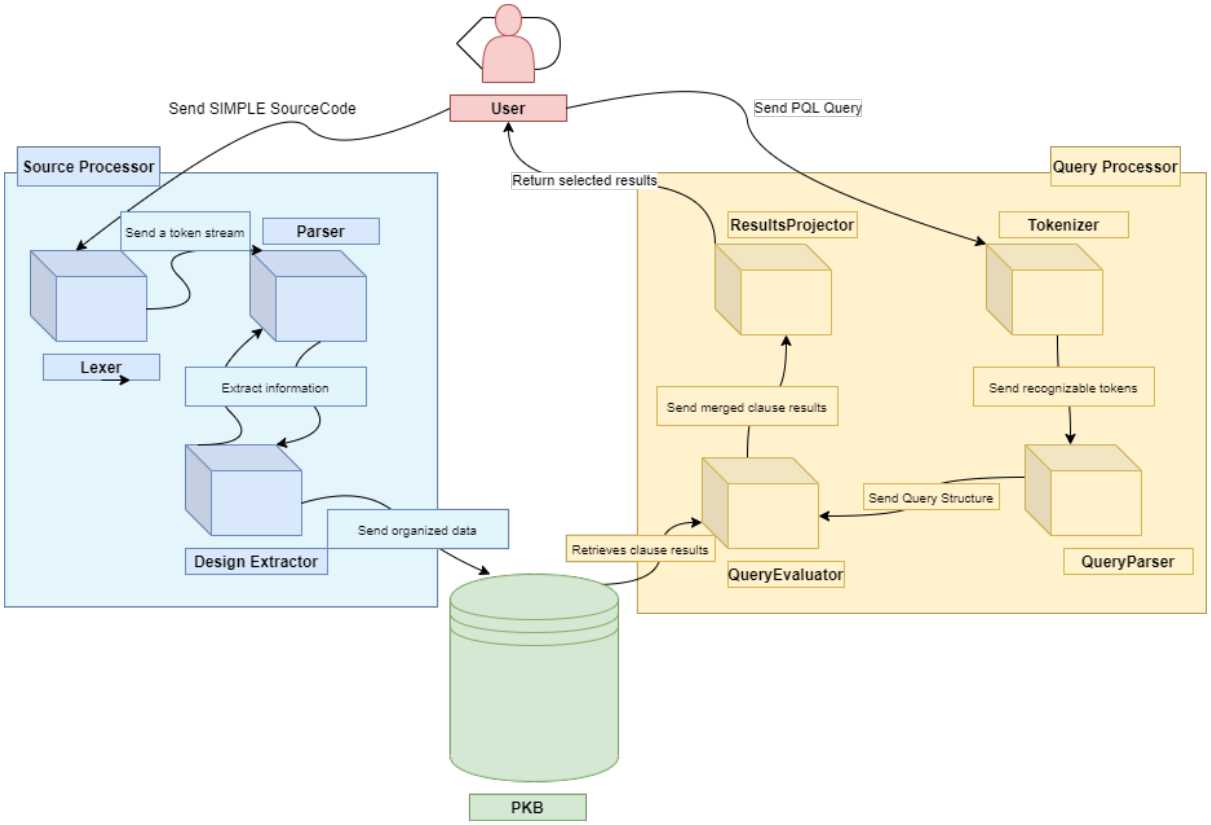
2.3 Iteration 3 Activities

Week	Activity	Tuan	Phat	Zhao Huan	Jun Hong	David	Bruce
11	Implement NextBip, NextBip(*)	✔					
	Implement AffectBip, AffectBip(*)		✔				
	Design PKB API for with clauses			✔		✔	
	Write methods for new relationships			✔			
	Refactor clause classes to inherit from generic parent clause class					✔	
	Refactor Declaration class to handle attributes					✔	
	Implement **with** clause in QueryEvaluation					✔	
	Add new Relationship Types for Affects(*), NextBip(*) and AffectsBip(*)					✔	
	Implement Select BOOLEAN and select tuple					✔	
	Refactor code: Move semantic validation logic into QueryParserErrorUtility						✔
	Add new Relationship clauses for Affects(*), AffectsBip(*), NextBip(*)						✔
	Implement With clause						✔

	Implement Select Tuple and Boolean						✔
	Modified Tokenizer implementation to handle symbols coming after identifiers (e.g. stmt#, prog_line, variable*)						✔
	Design of SIMPLE Code and PQL queries				✔		
12	Unit tests for NextBip	✔					
	Unit tests for AffectBip		✔				
	Integration testing with PKB	✔					
	Implement with clauses			✔			
	Implemented Union-Find algorithm for dividing clauses into groups					✔	
	Implemented sorting of clause and group results					✔	
	Stress testing for QE Optimization					✔	
	Implemented TestWrapper and Query methods to detect Select BOOLEAN correctly						✔
	Integration testing: query evaluator + query preprocessor					✔	✔
	Integration testing: QE + PKB			✔		✔	
	Design of PQL Queries				✔		

13	Integration testing: Parser + PKB	✔		✔			
	System testing	✔	✔	✔	✔	✔	✔
	Work on the final report	✔	✔	✔	✔	✔	✔

3 SPA Design



The Architecture Diagram given above explains the high-level design of our SPA.

The SPA consists of three main components:

- **Source Processor**: First, it receives a SIMPLE source code. It sends to Lexer (Tokenizer) to tokenize the code into a token stream. Then the Parser will gradually consume tokens and extract fundamental information into the Design Extractor. The Design Extractor finally organizes, extracts advanced relationships and sends all the information to the PKB. The semantic errors will also be checked while parsing and it will terminate the program upon any occurings of errors.

- **PKB**: When the PKB has been populated with all necessary information, it organizes the knowledge in a tabular manner. After the initialization, it will wait for retrieval requests from the Query Processor, and send back the information being requested.
- **Query Processor**: Validates and evaluates user inputs queries written in PQL. The Tokenizer first converts the PQL query into recognizable tokens, which are passed on to the QueryParser to parse them and populate the query structure accordingly. With the query structure, the QueryEvaluator evaluates the query by making use of data stored in the PKB. Finally, from the evaluated results by the QueryEvaluator, the ResultsProjector returns the selected results.

3.1 Source Processor

3.1.1 General Structure

The front end of SPA is divided into 3 main components:

1. **Lexer** (tokenizer): The **Lexer** transforms a SIMPLe parser as a string into a more well-defined data structure - a token stream.
2. **Program Parser**: The **Parser** receives a SIMPLE Token stream and extracts the necessary information while parsing the program.
3. **Design Extractor:** The **Design Extractor** receives information from the **Parser** and organizes them in an appropriate structure and then passes them to the **Program Knowledge Base.**

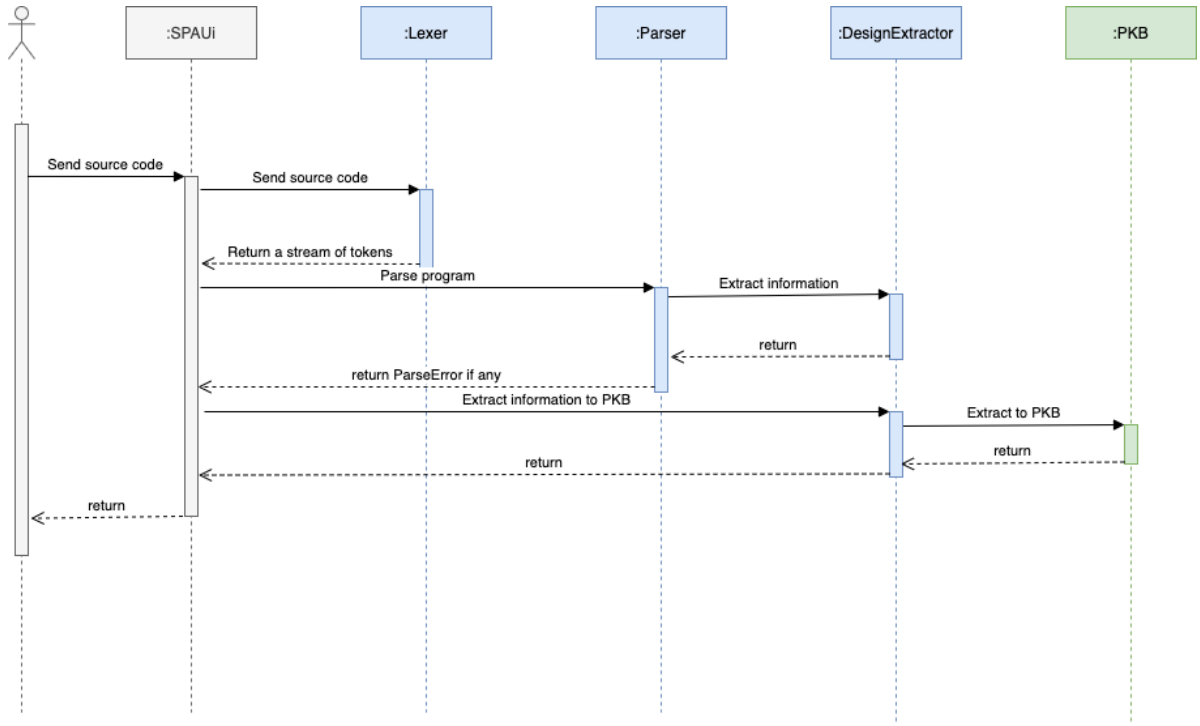


Figure 3.1.1 : Sequence diagram of interaction between components in Source Processor and PKB.

3.1.2 Lexer (tokenizer)

Design

These are the classes used:

- **TokenType:** An enum class, denoting the possible type of a token.
- **SIMPLEHelper:** The helper is responsible for converting a string into a proper token type. In order to do that, the helper also implements some convenient functions with reusability so higher level components can also utilize.
- **SIMPLEToken:** This SIMPLEToken will hold a string, which represents the actual value, and the position of this token in the code, consisting of row position and column position. This position is used when an error occurs, which corresponds to a parse error, so that we can quickly analyze if the input is actually incorrect. Also, the token holds the type of the string it contains, which is acquired via the helper for consistency.
- **SIMPLETokenStream**: This will convert a list of string with no endline character into a consumable stream of token or an empty stream if the tokenize process is unsuccessful. This also implements some method of consuming tokens that the Parser will use.

Implementation

First, we will describe how the helper recognizes which string corresponds to which type of token. The given string must not be empty and must not contain separators such as space and tab.
At the time of creation, the helper will first initialize a mapping from keyword and special characters to its corresponding TokenType.
Then, to convert a string to the TokenType, first the function will consult the table to see if the string is in it. If there is, then we can directly return the mapping from the table. Otherwise, we still have three cases that have to be considered.

1. The token can be a number.
2. The token can be an identifier.

These 2 cases are well defined.

3. Otherwise, we return a default TokenType for later error handling.

Next, we will consider how the TokenStream works. It will repeatedly take a string from the list provided, which corresponds to one line of code, then repeatedly consume character from that string to construct tokens and append it to the stream. If a token is not recognized as valid, the function will empty the stream and indicate that the tokenize process is unsuccessful.

In order to do that, we will maintain a buffer of what we have for our current token. Then for a new character c:

1. If c is a delimiter (space, tab etc) then we will convert the content of the current buffer into a token if it is not empty, then empty it and move on to the next character.
2. If c is a brace (either '(' or ')'), then first we will convert the content of the current buffer into a token if it is not empty, then empty it. Also, we will push a token with the value of c into the stream, since there is no special token which can start with a brace.
3. If c is a delimiter, in this case only ';' is available , then same as case 2.
4. Now we need to check if c can be appended to our buffer or not. If not, then we will convert the content of the current buffer into a token if it is not empty, then empty it and append c to the buffer. Else, we will just append c to the buffer.

To check if we can append a character into the buffer or not, we have these cases:

1. If the buffer is empty, we can always append the character c.
2. Else, we will first check if all the characters in the buffer are alphanumeric, call the boolean result A1. Then we also check if the current character is alphanumeric, call the boolean result A2. Then if A1 is the same as A2, we can append c to the buffer and otherwise we can not.

Example :

1. buffer = "a27", c = ':' -> not appendable
2. buffer = "!", c = '=' -> appendable
3. buffer = "=", c = '!' -> appendable. Notice that we have not checked if the token can not be valid at this stage.

Maintaining the position of the token in the code is trivial and will not be discussed here.

Example

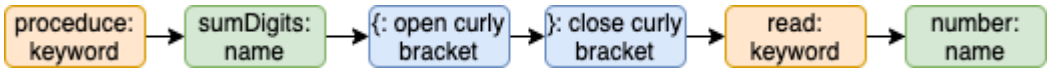
This is the sample code of how to compute sum of digits of a number in SIMPLE

```
procedure sumDigits {
    read number;
    sum = 0;

    while (number > 0) {
        digit = number % 10;
        sum = sum + digit;
    }

    print sum;
}
```

The first six tokens in the stream will be:



3.1.3 Program Parser

Design and Algorithms

The Parser serves a purpose of:

- receiving a stream of tokens from the Lexer.
- identifying a list of important entities such as: statements, variables, …
- extract relationship between entities.

We have observed that the SIMPLE language has a grammar of a **LL(2)** language - the language that requires only looking up the next two tokens to decide the type of parsing objects.

Hence, we use the famous **Recursive descent parser** algorithm. However, for the grammar to be LL(2), we also need to use **Operator precedence parser** for conditional expression.

It simply consists of many recursive functions that can call each other, each receives a stream of token and necessary scoping information, and returns a parse error if there is any. An parse error can be either an empty error (which means the program has been parsed successfully), or can contain the token where things go wrong. Currently, in iteration 1, there are no requirements on semantical errors.

For example, a function can look like this

- ParseError *parseProcedure*(SIMPLETokenStream stream);
- ParseError *parseReadStatement*(SIMPLETokenStream stream, int parentStatementIndex);

We allow ParseError to be combined so that we can return the user what and where the error is. A parsing function also remembers scoping information that helps extract relationships. For example:

- When parsing a statement, the parent' index is passed as a parameter. This allows the Parser to call DesignExtractor's API *addDirectParentRelationship.*
- When parsing an expression, the current statement's index is also passed as a parameter. This again allows the Parser to call *insertDirectUses* APi provided by the DesignExtractor.

There are two states that the Parser are in:

1. Need to consume one particular token of a particular type. For example: when we call *parseProcedure,* we first need to consume a keyword token with value "procedure" and a name token, followed by an open curly bracket token. After that we will call *parseStatementList* to parse a statement list. After the sub-call returns, it means that we have successfully parsed a statement list. The job now is just to consume the closing curly bracket and then return successfully.
2. Need to look up the next two tokens to decide the parsing objects. For example, if we have to parse a list of statements, then we do not know which type of statement to parse next. We can try to look ahead to two tokens. If the first token is of type "name" and the second token is an equal sign, then the next statement is definitely an assignment statement. We can safely call *parseAssignmentStatement* without a need to backtrack like a normal recursive parser.

If any of the time, the expected token needed is not present, the parser will return a ParseError.

During the parsing process, the parser will extract entities and relationships between entities accordingly and then call **Design Extractor**'s APIs to report those. For example:

- If the parser is inside *parseAssignmentStatement*, the parser can indicate the assigner of the statement. Then it will call *insertModifiesRelationship* of Design Extractor to report that there is a **modifies** relationship involved.
- If the parser is inside *parseStatementList*, it can maintain a vector of statements that follow each other, and then call Design Extractor's API accordingly.

Let's look at the above example:

```
procedure sumDigits {
    read number;
    sum = 0;

    while (number > 0) {
        digit = number % 10;
        sum = sum + digit;
    }

    print sum;
}
```

The activity diagram below illustrates how the Parser will consume tokens from the token stream and recursively calls the *parseStatementList.*

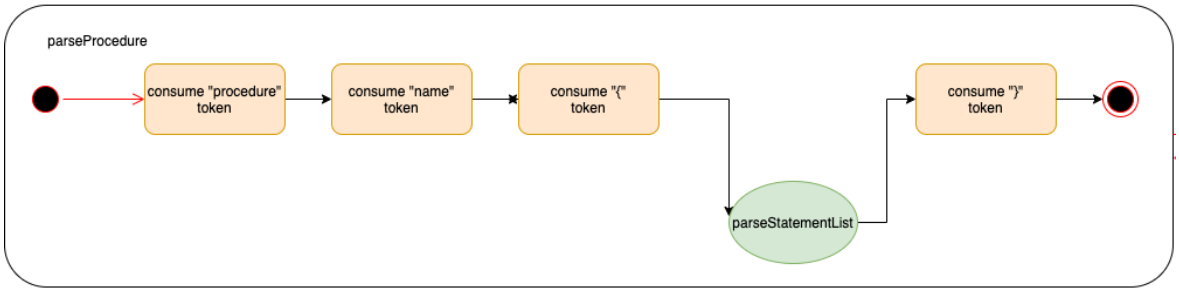


Figure 3.1: Activity diagram of *parseProcedure*

Next, the *parseProcedure* will constantly look up 2 tokens, and recursively call parse a single statement. Before returning, it will call DesignExtractor's API to add follows relationship between statements of the current statement list.

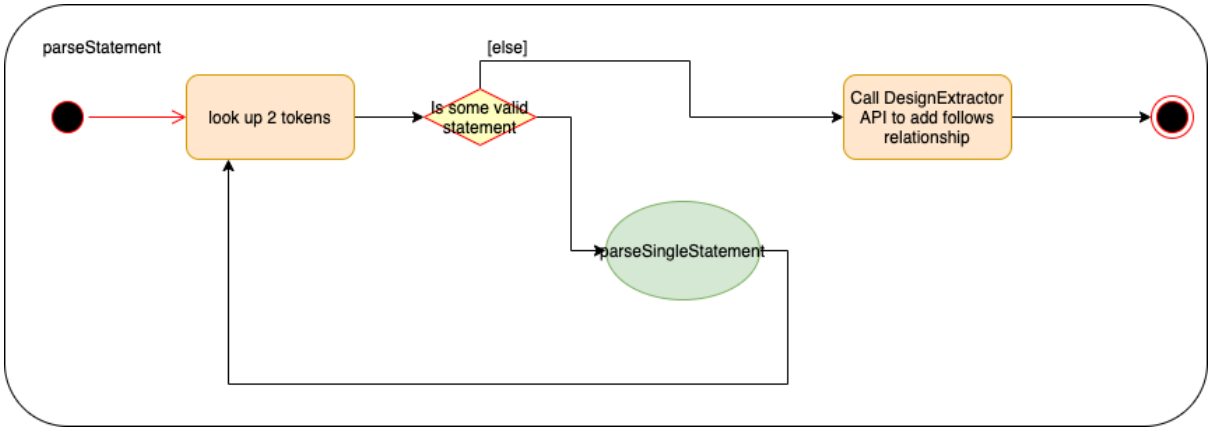


Figure 3.2: Activity diagram of *parseStatementList.*

We can see that in the example above, *parseStatementList* will iteratively call *parseReadStatement, parseAssignmentStatement, parsePrintStatement* before returning its execution. We take the *parseReadStatement* for example. It will consume "read", a name and ";" tokens. Then before returning its execution, it will call DesignExtractor's API to insert a ***direct modifies*** relationship.

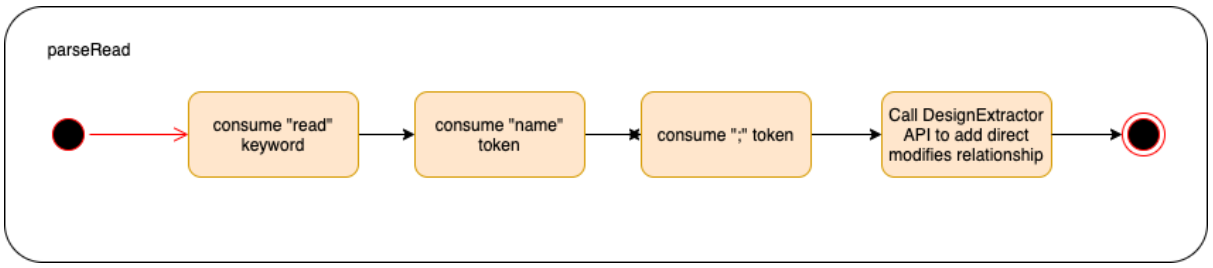


Figure 3.2: Activity diagram of *parseReadStatement.*

If during the process, when we expect a “read” keyword and the head of the token stream is, for example, an opening curly bracket, then we return immediately a ParseError.

Now, let’s take a look at the example below

```
procedure Foo {
1.    if (x == 1) then {
2.         while (y == 2) {
3.             call Bar;
           }
      } else {
5.         if (p == 2) then {
6.              x = y;
           } else {
7.              call Bar;
           }
      }
}

procedure Bar {
8.    x = z + 2;
9.    if (x == 1) then {
10.        y = 1;
      } else {
11.        y = 2;
      }
}
```

The sequence of Design Extractor’s APIs called are listed as follow:

- We firstly call parse the first procedure Foo
    - We recursively call to parse the if statement (1)
        - Call addUses(1, x)
        - We call parse while statement (2)
            - call addParent(1, 2)
            - call addUses (y, 2)
            - call to parse call statement (3)

- We call parse if statement (5)
  - call addUses(5, p)
  - call to parse assign statement (6)
    - add expressions
    - call addParent(6, 5)

We keep doing the same to extract all basic relationships between entities

A ParseError can also be raised if semantic errors are found:

1. A call statement calls a non-existing procedure.
2. There are cyclic calls between procedures.

Checking the 1, semantic error is easy. We maintain a set of procedure names.

Checking 2 is well-known in Algorithms and Data structure. We maintain a queue of procedures that have no incoming calls. We gradually perform topo-sorting by erasing edges from those procedures with no incoming calls and push newly independent procedures into the queue. If after this process, all the procedures have no incoming calls, we finish the check with a successful result. Otherwise, there exists a cycle and we raise a ParseError accordingly.

For example, we consider this SIMPLE source code

```
procedure A {
    call B;
}

procedure B {
    call C;
}

procedure C {
    call B;
}
```

We simulate the running of the check:

1. **A** has no incoming calls, we push A to the queue.
2. We pop A from the queue. A calls B, then we delete the edge from A to B. After this, the degree of B is still 1 because C is calling B.
3. The queue is empty, we terminate the process because B and C's degrees are positive and it means that there is a cyclic call.

As mentioned earlier, we handle parsing of conditional statements separately. First, we define a total order on the operators, represented by a number. Operator with higher precedence will bind more tightly, for example:

```
(3 + x) < 2 * x + 3
```

we can see that '(' and ')' will bind most tightly, then '*, then '+' and finally '<'.
For SIMPLE, all operators have left-to-right associativity, therefore we define the precedence as follows.

Precedence	Operators
1	&&, ||
2	!
3	>, >=, <, <=, ==, !=
4	+, -
5	*, /, %
6	()

Table 3.1.3.2: Table of operand precedence.

We also define a result from an expression to be of three types: number, boolean with no surrounding bracket, boolean with one surrounding bracket. The reason is that SIMPLE grammar rules require some conditions to be wrapped by exactly a pair of brackets. We will discuss more about the difficulty with strict nested rules in the following paragraphs.

The main idea of operator parser is that it will maintain a result stack implicitly (by recursive call) and a current level of precedence. It will associate the current result on the top of the stack with an operator if and only if the precedence of the incoming operator is larger than the current level (hence, binding tighter). Else, it will combine two results on the stack top into one by a look-up table involving operators and results. All operators are binary, except !, which we convert into a binary operator by allowing a placeholder result. With that, we can also check for semantic error here.

Below is a simple example of the algorithm. We denoted the result as {N, B} for number and boolean respectively. The red color highlights the part of the input which has been consumed by the Parser.

1. `x + 3 * a + 2 < 5`
2. `x + 3 * a + 2 < 5.` Result stack [N], current level: 0
3. `x + 3 * a + 2 < 5.` Result stack [N], current level: 4
4. `x + 3 * a + 2 < 5.` Result stack [N, N], current level: 4
5. `x + 3 * a + 2 < 5.` Result stack [N, N], current level: 5
6. `x + 3 * a + 2 < 5.` Result stack [N, N, N], current level: 5
7. `x + 3 * a + 2 < 5.` Result stack [N], current level: 4 (We combine 3 and a to 3 * a since + have lower precedence than 5. Also we combine x and 3 * a to x + 3 * a since + do not have larger precedence to 5).
8. `x + 3 * a + 2 < 5.` Result stack [N, N], current level: 4
9. `x + 3 * a + 2 < 5.` Result stack [N], current level: 2. (We combine x + 3 * a and 2 to x + 3 * a + 2)
10. `x + 3 * a + 2 < 5.` Result stack [N, N], current level: 2
11. `x + 3 * a + 2 < 5.` Result stack [B]. (Final step of combination)

There is one small modification we have to make to fit this algorithm into SIMPLE grammar. Normally, it is permissible to have an expression like this

```
((x + y) < 3) && x < 2
((x + y) < 3) && ((x < 2))
((x + y) < 3) && (x < 2)
```

However, the first two expressions are not valid in SIMPLE. This forces us to consider the level of nested expression in a particular expression.

Therefore, when we do type checking for a conditional expression, we have to define 4 result types : E (empty), N (number), B0 (boolean with 0 nested level), B1 (boolean with 1 nested level) (for demonstration, actual implementation can be different).

We do not have to define more than that since further nested levels of boolean will result in a syntax error. The empty result is used to handle unary operators and brackets for simplification of implementation.

The table below will show how the operators and results can be combined into a new result. A combination that does not appear in the table means that it will result in a semantic error.

ResultLHS	Operator	ResultRHS	ResultCombined
N	+, -, *, /,%	N	N
N	<, <=, >, >=, ==	N	B0

B1	&&, ||	B1	B0
E	()	N	N
E	()	B0	B1
E	!	B1	B0

3.1.4 Design Extractor

Problem and purpose

The problem is that the **Parser** parses a program in two passes without remembering its state. Hence, it's necessary to have a separate component that allows the Parser to gradually enlarge the set of statements and report important data. In later iterations, many more relationships are to be retrieved and organized by it. Also, it supports validating certain semantic errors.
The **Design Extractor** serves the purpose of receiving data from the **Parser,** organizes them in an appropriate data structure, and then passes them to the P**rogram Knowledge Base.**

Design

The **Design Extractor** exposes APIs so that the Parser can call to extract information. Some of them are

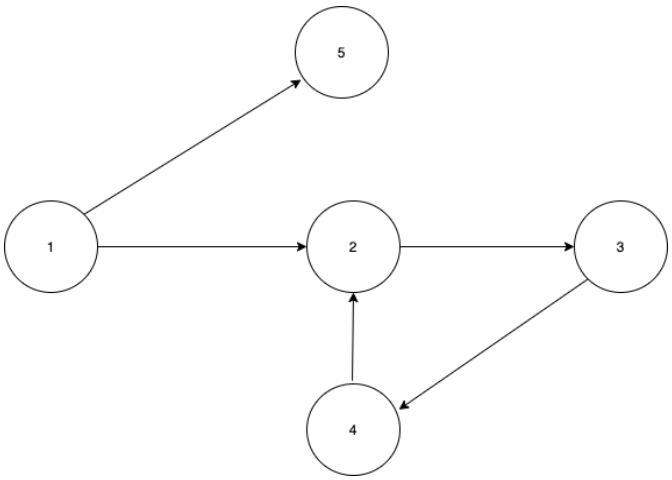
1. **int** startNewStatement(StatementType type): The Parser will report a new statement with a statement type. The Design Extractor will return the index of the new statement.
2. **void** insertDirectParentRelationship(int parent, int child): When the parser parses a new statement, it gets the parent statement's index in the local scope and then reports the Design Extractor about the new relationship.
3. **void** insertDirectFollowRelationship(int before, int after): When the parser parses a statement list, it gets all the statements index and reports the Design Extractor.
4. **void** insertDirectUsesRelationShip(int statementId, string variable): When the parser parses an expression, it gets the using statement's index in the local scope and then reports the Design Extractor about the new ***uses*** relationship.

The **Design Extractor** stores **dynamically allocated vectors** of data:

1. List of all variables used in the program.
2. List of all constants used in the program.
3. List of all statements with corresponding types.
4. List of all parent statements for each statement.
5. List of all following statements for each statement.
6. List of all uses/modifies variables for each statement.
7. List of all expressions.
8. List of all next statements for each statement
9. List of all affected statements for each statement

All of these data are populated during program parsing, except 9, which is calculated after the parse completed (since it requires relationship Next, Modifies and Uses). The list of affected statements is calculated by a BFS starting from each ***assign*** statement via Next relationship. Modifies and Uses is needed to check if the variable that is modified by the starting statement remains unmodified or not.

After that for all the closure transitive relationships (namely, each that end with '*'), we implemented a generic method that receives a data structure storing one direct relationship such as parent, next or follow. The direct relationship can be seen as a directed graph.



For example, we consider this graph of "directed relationship"

We will run Breadth First Search algorithm starting from each vertex {1, 2, 3, 4, 5}, and find all reachable vertices, stored in a list L.

We assign the list of indirect relationships starting as L.

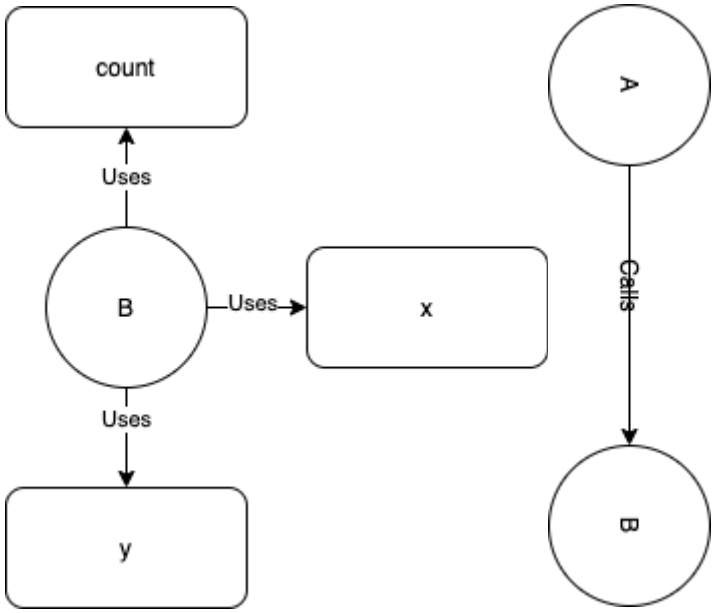
Vertex index	List of direct relationship	List of indirect relationship
1	2, 5	2, 3, 4, 5
3	3, 4	2, 3, 4
3	4	2, 3, 4
4	2	2, 3, 4
5		

We can see that this algorithm handles the Next relationship very well as it handles all the reachable vertices including the starting vertex itself.
The complexity of this algorithm is **O(n(n + m))** where n is the number of entities and m is the number of direct relationships between those entities.

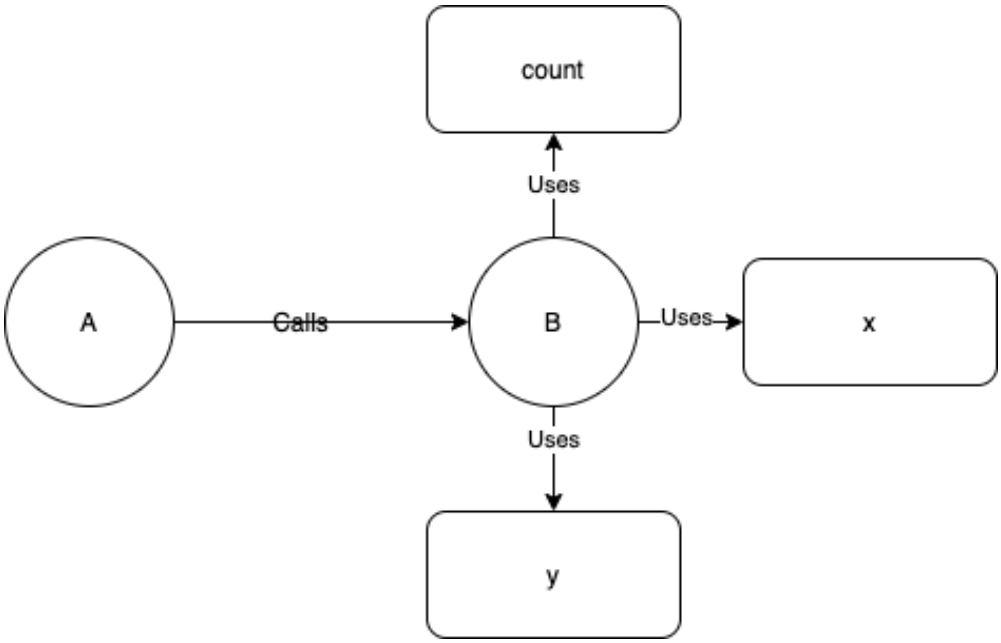
Next, how can we extract advanced relationships such as Uses/Modifies of procedures? We use the same algorithm which is implemented in a generic function.
For example, we have extracted Call(*) as a directed graph: **<string> -> <string>.** We also have extracted directed Uses(procedure) as a graph with edges from procedures and variables. We merge the two graphs and do BFS again to get so called “convolute” relationships:

- Given a graph F with edges from set A to set B, and a graph G from set B to set C, the function produces a graph with edges from A to C where a has an edge to c iff a has a path to b in F and b has a path to c in G. This can be done in O(n(n + m)) as above.

For example, given F and G as follows:



The combined graph should be this



Then, A reaches count, x, and y then A uses count, x, y.

For example, with this code (adding statement number)

```
procedure sumDigits {
1.    read number;
2.    sum = 0;

3.    while (number > 0) {
4.        digit = number % 10;
5.        sum = sum + digit;
      }

6.    print sum;
}
```

After parsing is completed, it will **conceptually** store

Variables	sum, number, digit
Constants	0, 10
Statement types	(1, PRINT), (2, ASSIGN), (3, WHILE), (4, ASSIGN), (5, ASSIGN), (6, PRINT)
Parent	(3, 4), (3, 5)
Follow*	(1, 2), (1, 3), (1, 6), … , (4, 5)

Uses	(4, number), (5, sum), (5, digit).
Modifies	(1, number), (4, digit), (5, sum)
Expression	“0”, “number % 10”, “sum + digit”

It will then call the respective API provided by PKB to insert the information into the PKB.

Variables	**bool insertVariable(string variable)** to insert a variable used
Constants	bool insertConst(string number) to insert a constant used
Statement types	**bool setStatementType(int stmt, EntityType type)** to set a type of a statement, such as: Assign or While
Parent	**bool insertParent(int parentStmt, int childStmt)** to insert a direct Parent relationship between two statements
Follow	**bool insertFollow(int beforeStmt, int afterStmt)** to insert a direct Follow relationship between two statements.
Uses	**bool insertDirectUses(int index, string variable)** to insert a direct Uses relationship between a statement and a variable
Modifies	**bool insertDirectModifies(int index, string variable)** to insert a direct Modifies relationship between a statement and a variable
Expression	**bool insertExpression(int id, Expression expression)**

Then, after building all the tables, the DE will make APIs calls to PKB to extract information:

1. Add all parent relationships:
    a. PKB.insertParent(3, 4)
    b. PKB.insertParent(3, 5)
2. Add all follows relationships:
    a. PKB.insert(1, 2)
    b. PKB.insert(2, 3)

It will sequentially add everything into PKB.

Design Decisions

We choose to implement 2 passes parser as it allows both components to be develop separately

Options available

Approach 1: String manipulations

The idea of string manipulation is to detect the presence of keywords, such as 'procedure' and 'while' and use that to then split the string into different parts, which can then be manipulated further, depending on the expected type of code. One problem we noticed at this approach is that SIMPLE allows an identifier to have keyword as name, that is

```
while = while + 5;
```

is a valid statement. This problem can be solved by looking ahead a few tokens to determine if the word is actually a keyword or an identifier.

Approach 2: Recursive descent parser

Based on the grammar provided, which is of Backus–Naur form (BNF), it is relatively straight-forward to implement a recursive descent parser.

The predictive version of the recursive descent parser is the most appealing option to us, since the runtime will be linear to the length of the input. However, it has an additional constraint that we have to find a small constant k such that by looking ahead k token, we can directly decide which production rule to follow (or in short term, the SIMPLE language must belong to LL(k) for some k). However, as an expression can start with as many round open brackets as we like, k will not exist in this case (since we cannot detect if we are parsing a conditional expression or an arithmetic expression)

We also consider the backtracking version of recursive descent parser, however, in the worst case, it might not even terminate (since the language does not belong to LL(k)). Even if our parser does terminate, it might take exponential time to the input length. Also, after a branch is taken and it is not valid, we will have to restore the state back and only then can take another branch. With limited development time, we think this approach might not be practical as the chance of introducing a bug can be high (due to the large number of functions we have to implement)

Approach 3: Operator precedence climbing (or Pratt parser)

Operator precedence climbing is mainly used to convert expressions written in infix notation to a format that can be computed more efficiently. Therefore, if we want to use it in our parser, then it can only be used to parse conditional and arithmetic expressions.

The main advantage of using operator precedence climbing is that an expression can always be parsed by just looking one token ahead. Also, it is able to handle both left and right associativity (in this project, we only need left associativity). By rewriting the grammar, we were able to assign precedence to all operators.

Evaluation Criteria

The most important criteria for us are **ease of implementation**, that is each component should be relatively simple to implement, even if it will require boilerplate code. Then, we will try to ensure **strong invariants** in each parser function, so that the combination will be relatively bug-free to ensure that we extract and provide correct information to the PKB.

Another important metric that we consider is the **runtime** of our parser, as there is a time limit requirement on parsing time.

Evaluation of Options

	Coverage	**Ease of Implementation**	**Invariant**	**Runtime**
String manipulation	Full	Difficult	Weak	Linear
Recursive descent parser	Full	Difficult	Strong	Exponential
Operator precedence parser	Expression only	Moderate	Strong	Linear

Final Decision and Rationale

Both recursive parser and string manipulation are difficult to implement, therefore based on the fact that the recursive descent parser can maintain a stronger invariant, we choose it as the main algorithm for our Parser.

While implementing it, we realise that if we incorporate operator precedence parser for expression parsing, then we can rewrite the grammar such that we only have to look up 2 tokens to decide which branch to take. Therefore, we are able to take advantage of both algorithms, since it results in

a parser that is linear and still maintains a strong invariant. Also, by this way, we do not have to account for backtracking, which reduces a large number of cases that we have to consider and therefore further simplifies our implementation.

3.2 PKB

Design

The Program Knowledge Base (PKB) serves as a database for the system which is a persistent intermediate component that interacts with both the source parser and the query processor. The PKB is designed to achieve fast retrieval of data, and to reduce the overhead of repetitively calling methods from the PKB interface. Here is a brief summary of the **conceptual** internal structures used in PKB categorized by their functionality.

Entity Tables

- **`map<string, EntityType> types`**
  a map that maps a statement index to its type
- **`map<EntityType, set<string>> entities`**
  a map that maps an entity type to a set of entities

Relationship Tables

- **`map<string, set<string>> relations[]`**
  a map that maps an entity to a set of entities that are of some relationship with it
- **`map<string, set<string>> relationsBy[]`**
  an inverted map of **`relations[]`** above
- **`set<string> relationKeys[]`**
  a set containing all keys of the **`relations[]`** map
- **`set<string> relationByKeys[]`**
  a set containing all keys of the **`relationsBy[]`** map

Assign Pattern Tables

- **`map<string, set<Expression>> expressions`**
  a map that maps an expression to a set of indices of statements that contain it

Container Pattern Tables

- **`map<string, set<string>> contPattern[]`**
  a map that maps a container statement to its control variable

- **map<string, set<string>> contPatternBy[]**
  an inverted map of **contPattern[]** above
- **set<string> contPatternKeys[]**
  a set containing all keys of the **contPattern[]** map

With Clause Tables

- **set<string> procVarName**
  a set of identical names for both some procedure and some variable
- **map<EntityType, set<string>> constStmtNum**
  a map that maps a statement type to a set of statement numbers of the specified type which happen to occur as a constant in the program
- **map<string, set<string>> nameUsed[]**
  a map that maps a call/read/print statement to the name it uses
- **map<string, set<string>> nameUsedBy[]**
  an inverted map of **nameUsed[]** above

To illustrate the purposes of different structures more clearly, consider the content of PKB after being populated by the Source Parser given the following SIMPLE program as shown previously:

```
procedure sumDigits {
    read number;
    sum = 0;
    while (number > 0) {
        digit = number % 10;
        sum = sum + digit;
    }
    print sum;
}
```

types		entities	
Key	**Value**	**Key**	**Values**
1	read	read	1
2	assign	print	6

3	while	while	3
4	assign	assign	2, 4, 5
5	assign	const	0, 10
6	print	var	number, sum, digit
-	-	procedure	sumDigits

relations[follows*]		**relationsBy[follows*]**	
Key	**Value**	**Key**	**Values**
1	2, 3, 6	2	1
2	3, 6	3	1, 2
3	6	6	1, 2, 3
4	5	5	4
relationKeys[follows*]		**relationByKeys[follows*]**	
1, 2, 3, 4		2, 3, 5, 6	

relations[uses]		**relationsBy[uses]**	
Key	**Value**	**Key**	**Values**
3	number, sum, digit	number	3, 4
4	number	sum	3, 5, 6

5	sum, digit	digit	3, 5
6	sum	-	-

relationKeys[uses]	relationByKeys[uses]
3, 4, 5, 6	number, sum, digit

expressions	
number	4
10	4
number%10	4
0	2
sum	5
digit	5
sum+digit	5

contPattern[while]		contPatternBy[while]	
Key	**Value**	**Key**	**Values**
3	number	number	3

contPatternKeys[while]
3

nameUsed[print]		nameUsedBy[print]	
Key	**Value**	**Key**	**Values**
6	sum	sum	6

nameUsed[read]		nameUsedBy[read]	
Key	**Value**	**Key**	**Values**
1	number	number	1

The information stored for some other relationships is of similar structures, hence not listed. As the format returned to the query processor is consistent with that stored in PKB, we can achieve a good retrieval performance most of the time. More rationale about the choice of data structures is described in the next section. Now consider the following queries with retrieval methods:

- **Select s such that follows*(s, 3);**
  `relationsBy[follows*].get(3) - O(1)`
- **Select s1 such that follows*(s1, s2);**
  `relationsBy[follows*] - O(1)`
- **Select v such that uses(_, v);**
  `relationByKeys[uses] - O(1)`
- **Select a pattern a(“digit”, _“10”_);**
  `expressions.get(“10”) intersect with`
  `relationsBy[modifies].get(“digit”) - O(n)`
- **Select w pattern w(“number”, _);**
  `contPattern[while].get(“number”) - O(1)`
- **Select <pn, r> with pn.varName = r.varName;**
  `nameUsed[print] join nameUsedBy[read] on name - O(n)`

The API of PKB is divided for the source parser and the query processor. It provides methods for the design extractor to insert each of the relationships and the query evaluator to retrieve data of different formats. Below lists the abstract structure of data retrieval APIs, where we define a separate method for different clauses and different numbers of synonyms. Thel API for insertion methods can be found in the appendix at the end of this document.

- **LIST OF ENTITY** `getEntities(`**ENTITY TYPE**`)`
  This method is to handle queries with no clause, eg., `Select s; Select c;`
- **Boolean** `getBooleanResultOfRS(`**RELATION TYPE, QUERY INPUT**`)`
  This method is to handle queries with no synonyms in its **such that** clause, eg.,
  `Select s such that uses(_, “x”); Select s such that follows(3, 5);`
- **LIST RES** `getSetResultOfRS(`**RELATION TYPE, QUERY INPUT**`)`
  This method is to handle queries with one synonym in its **such that** clause, eg.,
  `Select p such that uses(p, “x”); Select s such that follows(s, 5);`

- **LIST OF LIST RES** getMapResultsOfRS(**RELATION TYPE, QUERY INPUT**)
  This method is to handle queries with two synonyms in its **such that** clause, eg.,
  `Select s such that uses(s, v); Select a such that calls*(p1, p2);`
- **LIST RES** getSetResultOfAssignPattern(**QUERY INPUT, EXPRESSION**)
  This method is to handle queries with no synonym in its assign **pattern** clause, eg.,
  `Select a such that pattern a(_, "x");`
- **LIST OF LIST RES** getMapResultOfAssignPattern(**QUERY INPUT, EXPRESSION**)
  This method is to handle queries with one synonym in its assign **pattern** clause, eg.,
  `Select a such that pattern a(v, "x");`
- **LIST RES** getSetResultOfContPattern(**STMT TYPE, VAR NAME**)
  This method is to handle queries with no synonym in its container **pattern** clause, eg.,
  `Select a such that pattern w(_, _);`
- **LIST OF LIST RES** getMapResultOfContPattern(**STMT TYPE, VAR NAME**)
  This method is to handle queries with one synonym in its container **pattern** clause, eg.,
  `Select a such that pattern ifs(v, _, _);`

For with clauses, however, things are trickier and we cannot separate the API according to the number of synonyms present. A property for with clauses is that there exist some attributes that essentially make difference as their synonyms when act as an argument, (examples being v.varName and s.stmt#, which behave identically as v or s when compared or selected), whereas some attributes do have an effect (such as c.procName and pn.varName, which reveals extra information). This observation inspires us to divide the API based on whether the two arguments are effective or not. An ineffective attribute is just a synonym during matching, while an effective one isn't.

- **LIST OF LIST RES** getDeclarationsMatchResults(**SYNONYM, SYNONYM**)
  This method is to handle queries with a **with** clause between two declarations, eg.,
  `Select p with p.procName = v.varName;`
- **LIST OF LIST RES** getDeclarationMatchAttributeResults(**SYNONYM, ATTR**)
  This method is to handle queries with a **with** clause between a declaration and an attribute, eg., `Select p with p.procName = c.procName;`
- **LIST OF LIST RES** getAttributesMatchResults(**ATTR, ATTR**)
  This method is to handle queries with a **with** clause between two effective attributes, eg., `Select r with r.varName = c.procName;`
- **LIST OF LIST RES** getAttributeMatchNameResults(**ATTR, NAME**)
  This method is to handle queries with a **with** clause between an attribute and a given name, eg., `Select r with r.varName = "randomVariable";`

For a query with multiple clauses, the query evaluator will call multiple methods of PKB, and combine all the retrieved results. The PKB will not handle the ordering of clauses or intermediate results.

Design Decisions: Choice of Data Structure

When deciding on the internal structure of PKB, three possible implementations were considered. This section will provide the details of our design process.

Approaches

Approach 1: Maps and Sets (current design)

This approach was to store knowledge in a tabular manner. Hash maps, such as the default std::unordered_map in C++ which features an average O(1) retrieval using the key, are preferred for efficiency reasons. Sets (specifically, std::unordered_set in C++), on the other hand, feature distinctness of elements and an average O(1) lookup time, which comes handy in face of frequent duplicate handling and lookup attempts in our project. With auxiliary structures maintained (such as the set of keys), this approach can achieve a good average retrieval time for almost any type of queries as the data stored can be directly used as results.

Approach 2: AST

Relationships such as *parent-child*, *follows*, indirect *uses* and *modifies* seem very natural to be represented as a tree structure. AST is also especially useful for matching subexpressions in pattern clauses, as the rule of matching is defined using AST.

Approach 3: Matrices

Lists, sets and maps are all single-sided data structures, indicating a potentially high duplicate rate of information stored inside the PKB. Matrices offer the flexibility to handle different types of queries using a single structure. For instance, the same time complexity can be achieved for queries like follows(1, s) and follows(s, 2) using a single *follows* matrix.

Evaluation Criteria

Ease of implementation: Approach 2 (AST) is especially difficult to implement and very prone to bugs. Approach 1 and 3 share a similar difficulty to implement.

Time complexity: Let n be the number of statements in the source program. Consider the **average** time to handle different types of queries about the *follows** relationship:

Select s such that...	Maps & Sets	AST	Matrices
follows*(3, 5)	**O(n)**	O(logn)	**O(1)**
follows*(s, 3)	**O(1)**	O(logn)	**O(n)**
follows*(s, _)	**O(1)**	**O(n)**	**O(n)**
follows*(3, s)	**O(1)**	O(logn)	**O(n)**

follows*(_, s)	**O(1)**	**O(n)**	**O(n)**

Other relationships have similar results. We can see that Approach 1 outperforms the others in most cases in terms of average retrieval time.

Final Decision and Rationale

A unanimous conclusion is that speed of retrieval is the main priority for PKB. In other words, we value time over space. Approach 1 is ultimately selected for its outstanding performance in most scenarios. Though it may take up extra space for duplicate information, we believe the tradeoff is worthwhile. In the actual implementation, we use string uniformly to better reuse code (as *uses* and *modifies* relationships require strings for both fields) and eliminate the need to convert results at the frontend. unordered_map<string, set<string>> is adopted for all relationships even though *follows* is a one-to-one relationship and *parent* is one-to-many, so that the map for all relationships can be represented in a single array.

Design Decisions: Data Retrieval API

Considering the possibility of multiple clauses in the future, the public methods provided for the query evaluator should be efficient and stable. The following two approaches were considered.

Approaches

Approach 1: List of Clauses

This approach is to provide a single API for the query evaluator to call in all scenarios. It will take in a list of clauses with their parameters and return the final result.

Approach 2: Single Clause (current design)

Approach 2 aims to provide APIs for different numbers of queries and different numbers of synonyms involved in each query separately, and leave the consolidation work to the query evaluator.

Evaluation Criteria

Ease of Extension: More extensibility is offered by the second approach, with support of selecting tuples of synonyms. Also, Approach 1 may result in a higher coupling between PKB and QE's substructures, as it may require internal knowledge of the query evaluator.

Ease of Implementation: Approach 1 is practically much harder to implement and test. It poses real challenges on how to formulate a structure of returned results to take advantage of the internal structure of PKB, which we failed to work out during ideation.

Time Complexity: Theoretically, the first approach may have a slight advantage, as less transferring of data is needed for optimization, and previous results may be better reused.

Final Decision and Rationale

Though Approach 1 may have lower time complexity, its high difficulty to code and extend makes it not appealing. Approach 2 is eventually adopted, as we wish to avoid major overhauls in the future in a quick-paced development. This decision is an excellent demonstration of the tradeoff between high maintainability and possibly slightly better performance. We vote for the former.

3.3 Query Processor

The Query Processor component can be further broken down into these subcomponents:

- Query Preprocessor
- QueryEvaluator
- ResultsProjector

3.3.1 Query Preprocessor

The Query Preprocessor is responsible for parsing the user's PQL query, storing the extracted clauses and their arguments in a Query object, and finally passing this object to the Query Evaluator.

Design

The Query Preprocessor comprises 2 major components which handle the parsing logic: the **Tokenizer** and **QueryParser**. We have decided to split the process of parsing into 2 parts to allow for better decoupling and adhering to the Single Responsibility Principle.

The **Tokenizer** (or Lexer) will handle the conversion of raw query input string into recognizable tokens (as well-defined by SPA concrete grammar rules) while the **QueryParser** will only have to parse these tokens generated by the Tokenizer (rather than dealing directly with the raw input string), and populate the Query object accordingly.

Implementation

InputStream

InputStream is a component that simply stores the input string and an index to the current character. It exposes 3 methods to the Tokenizer:

- **`char next()`** returns the current character and advances the index
- **`char peek()`** returns the current character
- **`bool eof()`** returns true or false depending on whether the end of the input has been reached

Tokenizer

The Tokenizer will process the characters passed to it by the InputStream and convert it into tokens. These tokens are represented by **Token** objects, which store both the actual value of the token and an enum type **TokenTypes** indicating the type of the token.

A string that does not match any of the recognised token strings (see Appendix for the list of recognised strings and their corresponding TokenTypes) will cause our own custom Syntactic error to be thrown. (Note: We catch all such syntactic errors in the TestWrapper evaluate method and simply return without populating the results list)

The Tokenizer exposes the following method to the QueryParser:

- `std::unique_ptr<Token> readNext()`
  process the next Token and return it

Query Parser

We have chosen to implement the Query Parser using a **recursive descent** approach. A recursive descent parser is a top-down parser built from a set of mutually-recursive procedures where each such procedure implements one of the non-terminals of the grammar. (Non-terminals are symbols that appear on the left-hand side of any grammar production, e.g. select-cl, declaration, design-entity etc)

This is not only an intuitive approach, but it also achieves separation of concerns, where each function in the Query Parser corresponds to a single non-terminal of PQL grammar. For example, we have a **selectClause** function to represent the select clause non-terminal, and a **patternClause** function to represent the pattern clause non-terminal in the grammar rules, and so on.

One extra design consideration we made was to allow keywords (for example “stmt”, “const” design entity tokens or “Follows”, “Parent” relationship name tokens etc) to be able to be treated as Identifier tokens by the Query Parser. This was done to circumvent edge case queries where synonym names happen to be the same as keywords, which is allowed based on SPA requirements.

For example, suppose we have the query "**stmt Select; Select Select**".

1. When parsing the select clause, after consuming the "stmt" design entity token we are expecting to see an identifier token.
2. However, here we encounter the "Select" keyword token.
3. Instead of throwing an exception we check to see if the current token can be treated as an identifier. (that is, it is of either of identifier token type or one of keyword token types)
4. In this case it indeed can, and thus we simply consume the token and no grammatical rule is violated.

The main role of the Query Parser would be to populate a Query object with information representing each of the various clauses and their arguments. Before diving into some examples of how the Query Parser parses sample PQL queries and populates this Query object, let us first look at how we have designed this Query object.

3.3.2 Query Internal Representation

Design

To represent a query, each non-terminal in the PQL grammar is a class, and the aggregation of the objects of these classes is used to represent the grammar rules.
Next page is the high-level structure of the classes used to internally represent a PQL query:

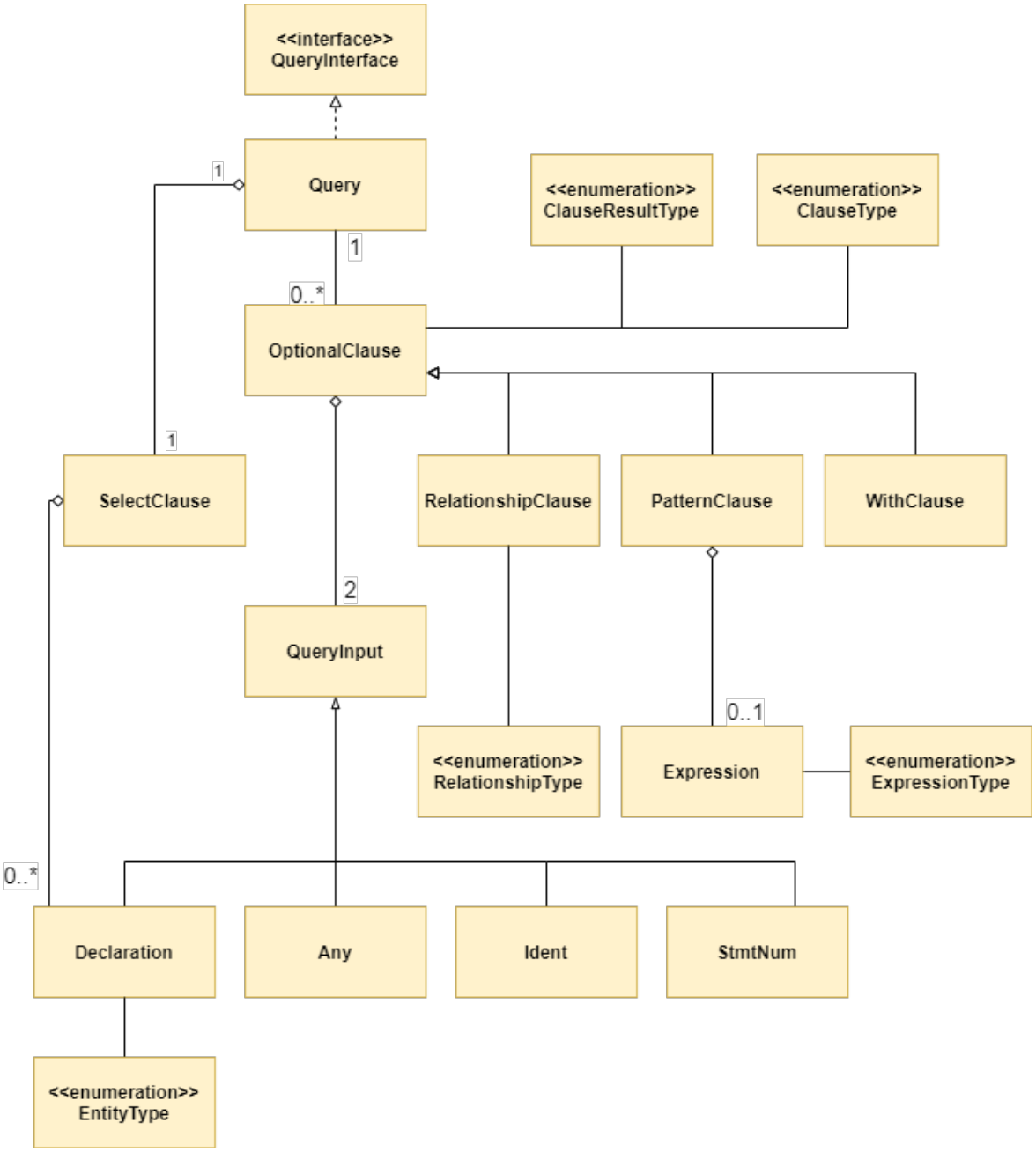


Figure 3.3.1.1: High level class representation of a PQL query

Implementation

`QueryInterface`: The interface that has setter methods for the query preprocessor to populate the query object with, and also getter methods for other classes like `QueryEvaluator` and `ResultsProjector` in the Query Processor component to retrieve query data.

`Query`: The implementation of the internal representation of a PQL query. It holds exactly one `SelectClause` object. The Query class holds a vector of `OptionalClause` objects, where `OptionalClause` is the generic parent class of all optional clauses, **such that** (relationship),

pattern and **with** clauses. Using a vector to store the optional clauses not only allows the query evaluator to easily check for the presence of the optional clause, but also allows for no limit on the number of optional clauses in each query, according to the advanced SPA requirements.

`SelectClause`: Represents the Select clause in the query and holds a vector of `Declaration` objects. Using a vector to store the selected declarations allows the different types of selected entities to be represented. An empty vector represents select BOOLEAN, a vector containing only one `Declaration` object represents select single synonym or select tuple with one synonym, and a vector containing more than one `Declaration` object represents select tuple.

<table>
<tr><td>Example query</td><td>assign a; constant c; procedure proc; read r;
Select <a, c, proc, r.varName></td></tr>
<tr><td>SelectClause object population</td><td>Select Clause
<table>
<tr><td>aDeclaration (vector of Declarations)</td><td>
Declaration
<table>
<tr><td>aEntityType</td><td>Assign</td></tr>
<tr><td>aValue</td><td>“a”</td></tr>
<tr><td>isAttribute</td><td>false</td></tr>
</table>
Declaration
<table>
<tr><td>aEntityType</td><td>Constant</td></tr>
<tr><td>aValue</td><td>“c”</td></tr>
<tr><td>isAttribute</td><td>false</td></tr>
</table>
Declaration
<table>
<tr><td>aEntityType</td><td>Procedure</td></tr>
<tr><td>aValue</td><td>“proc”</td></tr>
<tr><td>isAttribute</td><td>false</td></tr>
</table>
</td></tr>
</table>
</td></tr>
</table>

	Declaration aEntityType: Read aValue: "r" isAttribute: true

Example query	Select BOOLEAN
SelectClause object population	Select Clause aDeclaration (vector of Declarations): (Empty, no Declarations) As can be seen, if we have "Select BOOLEAN" our SelectClause's vector of Declarations will be empty. Otherwise, for a result clause of elem or tuple the SelectClause will hold at least one Declaration.

Figure 3.3.1.2: SelectClause object example population

`OptionalClause`: The generic parent class of all optional clauses of a query: **such that** (relationship), **pattern** and **with** clauses. This class encapsulates all common features of these clauses, like containing two `QueryInput` objects. Such a parent clause class is needed for the optimization process in query evaluation, as the optimization component needs to sort all clauses regardless of clause type. `ClauseType` has the values of RELATIONSHIP, PATTERN and WITH to indicate the type of clause, and `ClauseResultType` has the values of MAP, SET and BOOL to indicate the data structure of the clause's results from the PKB. These enumerations are needed in the query optimization and evaluation process.

`RelationshipClause`: Represents the **such that** clause in the query and holds an enum `RelationshipType`, and two `QueryInput` objects. `RelationshipType` has values of `FOLLOWS`, `FOLLOWS_T`, `PARENT`, `PARENT_T`, `USES`, `MODIFIES`, `CALLS`, `CALLS_T`, `NEXT`, `NEXT_T`, `AFFECTS`, `AFFECTS_T`, `NEXTBIP`, `NEXTBIP_T`, `AFFECTSBIP` and `AFFECTSBIP_T` to represent the entity relationship in the **such that** clause. The two `QueryInput` objects it contains represent the LHS and RHS parameters in the such that clause.

<table>
<tr><td>Example query</td><td>assign a;
Select a with Affects*(a, 6)</td></tr>
<tr><td>RelationshipClause object population</td><td>RelationshipClause
<table>
<tr><td>aRelationshipType</td><td>AffectsT</td></tr>
<tr><td>aLeftInput</td><td>Declaration
<table>
<tr><td>aEntityType</td><td>Assign</td></tr>
<tr><td>aValue</td><td>“a”</td></tr>
<tr><td>isAttribute</td><td>false</td></tr>
</table></td></tr>
<tr><td>aRightInput</td><td>StmtNum
<table>
<tr><td>aValue</td><td>“6”</td></tr>
</table></td></tr>
<tr><td>clauseType</td><td>Relationship</td></tr>
</table></td></tr>
</table>

Figure 3.3.1.3: RelationshipClause object example population

`PatternClause`: Represents the **pattern** clause in the query. `PatternClause` only takes in an `Expression` object if it is representing an **assign pattern** clause. It holds two `QueryInput` objects, where the first `QueryInput` object represents the `Declaration` whose pattern is being matched - an assignment, while or if entity. The second `QueryInput` object represents the LHS parameter in the pattern clause.

`Expression`: Represents the expression to be matched in the pattern clause. This class does not inherit from the `QueryInput` class as the grammar rules are different from what is accepted in the RHS parameter of the **pattern-assign** clause, and therefore warrants it being its own class. `Expression`

also contains an `ExpressionType`, which can assume the values of EXACT, PARTIAL, EMPTY to represent full expression matching, subexpression matching and wildcard expression respectively. `Expression` is also shared with the SIMPLE Parser since the grammar rules of an expression in PQL and SIMPLE are exactly the same.

<table>
<tr><td>Example query</td><td>assign a;
Select a pattern a(“x”, “a + b * c - 2 / z”)</td></tr>
<tr><td>PatternClause object population</td><td>PatternClause
<table>
<tr><td>aLeftInput</td><td>Declaration
<table>
<tr><td>aEntityType</td><td>Assign</td></tr>
<tr><td>aValue</td><td>“a”</td></tr>
<tr><td>isAttribute</td><td>false</td></tr>
</table></td></tr>
<tr><td>aRightInput</td><td>Ident
<table>
<tr><td>aValue</td><td>“x”</td></tr>
</table></td></tr>
<tr><td>aExpression</td><td>Expression
<table>
<tr><td>aValue</td><td>“((a + (b * c)) - (2 / z))”</td></tr>
<tr><td>expressionType</td><td>Exact</td></tr>
</table></td></tr>
<tr><td>clauseType</td><td>Pattern</td></tr>
</table>
Our Expression will add parentheses into the raw expression so as to allow for ease of matching in the PKB.

It is to be noted that we will have an expressionType of Exact since in pattern a(“x”, “a + b * c - 2 / z”) we are trying to do full expression matching.</td></tr>
</table>

Figure 3.3.1.4: PatternClause object example population

`WithClause`: Represents the **with** clause in the query. The two `QueryInput` objects it contains represent the LHS and RHS of an attribute comparison. For example, "with 4 = s.stmt#" would be a `WithClause` object with a left `QueryInput` of type `StmtNum` and a right `QueryInput` of type `Declaration`.

<table>
<tr><td>Example query</td><td>procedure p;
Select p with p.procName = “procA”</td></tr>
<tr><td>WithClause object population</td><td>WithClause
<table>
<tr><td>aLeftInput</td><td>Declaration
<table>
<tr><td>aEntityType</td><td>Procedure</td></tr>
<tr><td>aValue</td><td>“p”</td></tr>
<tr><td>isAttribute</td><td>false</td></tr>
</table></td></tr>
<tr><td>aRightInput</td><td>Ident
<table>
<tr><td>aValue</td><td>“procA”</td></tr>
</table></td></tr>
<tr><td>clauseType</td><td>With</td></tr>
</table>
It is to be noted that we represent attribute references like “p.procName” using the <code>Declaration</code> class, the same as how we represent synonyms. In this case, we represent the synonym “p” in the SelectClause with a Declaration as well.

This is because by default, when we do “Select p”, we will return the procedure names of all matches. This means that p.procName is already implied by its synonym “p” based on its entity type of procedure.

However, it is to be noted that there are 3 exceptions, namely print, read and call. These are the only entity types whose synonyms have more than one</td></tr>
</table>

	valid attribute name. For synonyms of all other entity types, they have only one attribute name, and it can be inferred directly from its entity type. Thus, we **differentiate between a synonym's primary attribute name and secondary attribute name** by means of our `Declaration class` having a `setIsAttribute` method. A synonym's primary attribute would be defined as an attribute name that is implied directly from the synonym itself, while a secondary attribute is not implied directly. For all synonyms of entity types that have more than one valid attribute name (only 3 of them exist: print, read and call), we will call this method that sets a boolean `isAttribute` to true if the attribute name used is its secondary attribute. Otherwise, we will not call the method if the attribute used is the primary attribute (primary attribute being stmt# in the case of these 3 entity types). Adding the ability to differentiate between different attribute names to the existing `Declaration` class **removes the need for us to create a separate class for attribute references.**

<table>
<tr><td>Example query</td><td>procedure p; read r;
Select p with r.varName = p.procName</td></tr>
<tr><td>WithClause object population</td><td>WithClause
<table>
<tr><td>aLeftInput</td><td>Declaration
<table>
<tr><td>aEntityType</td><td>Read</td></tr>
<tr><td>aValue</td><td>“r”</td></tr>
<tr><td>isAttribute</td><td>true</td></tr>
</table></td></tr>
<tr><td>aRightInput</td><td>Declaration
<table>
<tr><td>aEntityType</td><td>Procedure</td></tr>
<tr><td>aValue</td><td>“p”</td></tr>
<tr><td>isAttribute</td><td>false</td></tr>
</table></td></tr>
<tr><td>clauseType</td><td>With</td></tr>
</table>
For this query, we will have a WithClause with left input and right input of both Declaration.

The entity type of “read” is one of the 3 entity types that have two attribute names. Also, the implicit attribute name (primary attribute) of a synonym of type “read” would be “stmt#”. However, in this case we are using its “varName” attribute, or its secondary attribute. Thus, we will call the <code>setIsAttribute</code> method on its <code>Declaration</code> object and set <code>isAttribute</code> to true.

For the entity type of “procedure”, as well as all other entity types with no secondary attributes, we do not need to differentiate between its attribute names since they only have one. Thus, we create a <code>Declaration</code> object but do not call <code>setIsAttribute</code> and <code>isAttribute</code> remains false.</td></tr>
</table>

Figure 3.3.1.5: WithClause object example population

QueryInput: Generic parent class that represents the parameters of the clauses. `Declaration`, `StmtNum`, `Ident` and `Any` (Wildcard) inherits from `QueryInput`. This generic parent class is needed as there would be many combinations of input types for the clauses' parameters if we do not generalise the input type. Hence, this makes the design of our APIs cleaner and resistant to changes due to any addition of new input types.
Even though the grammar rules state that some clause parameters are 'stmtRef' while other parameters are 'entRef', grouping all the input types under one generic class instead of two is acceptable in this case as the query preprocessor is expected to detect any syntactic errors in the clause input types before populating the query. Therefore, there is no need to distinguish between the two groups of input types here.

`Declaration, Any, Ident, StmtNum`: Represents the possible input types for the query clauses as stated in the grammar rules.

`Any` represents the wildcard argument "_", `Ident` represents an identifier type argument, while `StmtNum` represents an integer type argument.

`Declaration` class has an additional enum `EntityType` to represent the declaration's entity type like WHILE, ASSIGN ,etc. Furthermore, it has a `setIsAttribute` method which allows us to set `isAttribute` to true (by default it is false). This method allows us to differentiate between a synonym's primary and secondary attribute names as mentioned above under `WithClause.`

Primary and Secondary Attributes of each Entity Type		
Entity Type	Primary Attribute	Secondary Attribute
Select	stmt#	-
Assign	stmt#	-
If	stmt#	-
While	stmt#	-
Program Line	-	-
Constant	value	-
Variable	varName	-
Procedure	procName	-

Read	stmt#	varName
Print	stmt#	varName
Call	stmt#	procName

Figure 3.3.1.6: Primary and Secondary Attribute names of each Entity Type

Design Decisions: Query Structure

There were two design considerations (including the current design) for the structure of the query. This section will detail the considerations for all alternatives and provide justification for the design chosen based on some evaluation criteria.

Approach 1: AST

This approach would model the PQL query as an abstract syntax tree (AST), a tree representation of the abstract syntactic structure of a PQL query. Each node of the tree would correspond with the non-terminals of the PQL syntax grammar and the edges in the tree would appear for each grammar rule. (E.g a node representing the Select clause would have edges directed to nodes that represent the declaration, the such that and the pattern clauses (if there are any))

Approach 2: Classes (current design)

This approach would model each non-terminal in the PQL grammar as a class, and have aggregation of objects to represent the grammar rules. (E.g the Select clause class has variables to hold the declaration object, and such that/pattern clause objects). The classes will have getter methods to easily retrieve any query data needed from the objects.

Evaluation Criteria

Ease of Access: Since there is a time limit on query evaluation, the query structure must support quick access to query data.

Ease of Extensibility: Keeping in mind further expansion of the PQL grammar in future iteration, the query design structure must be easy to extend and not require major modification to existing design.

Final Decision and Rationale

Approach 2 was chosen as it fulfills the ease of access and ease of extensibility criteria.

Firstly, approach 1 fails in its ease of access as the query evaluator would need to traverse the tree node by node, which makes query evaluation less efficient and inflexible compared to using getter

methods in approach 2. Furthermore, since the structure of a PQL query is not overly complex (no nestings etc), using a tree to organise its structure reaps little benefit over simply representing the query as class objects.

Secondly, approach 1 meets the ease of extensibility criteria due to the use of polymorphism and inheritance among the query classes. Generic parent classes like `QueryInput` makes it so that the addition of new PQL grammar will not require existing classes to be modified and yet these additions can be implemented by adding new classes.

Example of Query Parsing and Query object population

Let us use the following example query to better illustrate the recursive descent approach:

assign a; while w; prog_line n; Select <a, n> such that Parent*(w, a) and Next*(60, n) pattern a(“x”, _) with a.stmt# = n

In the activity diagrams below, we show the main logic of the query parsing. Syntactic errors are thrown whenever we encounter a token that is different from what we want to consume.

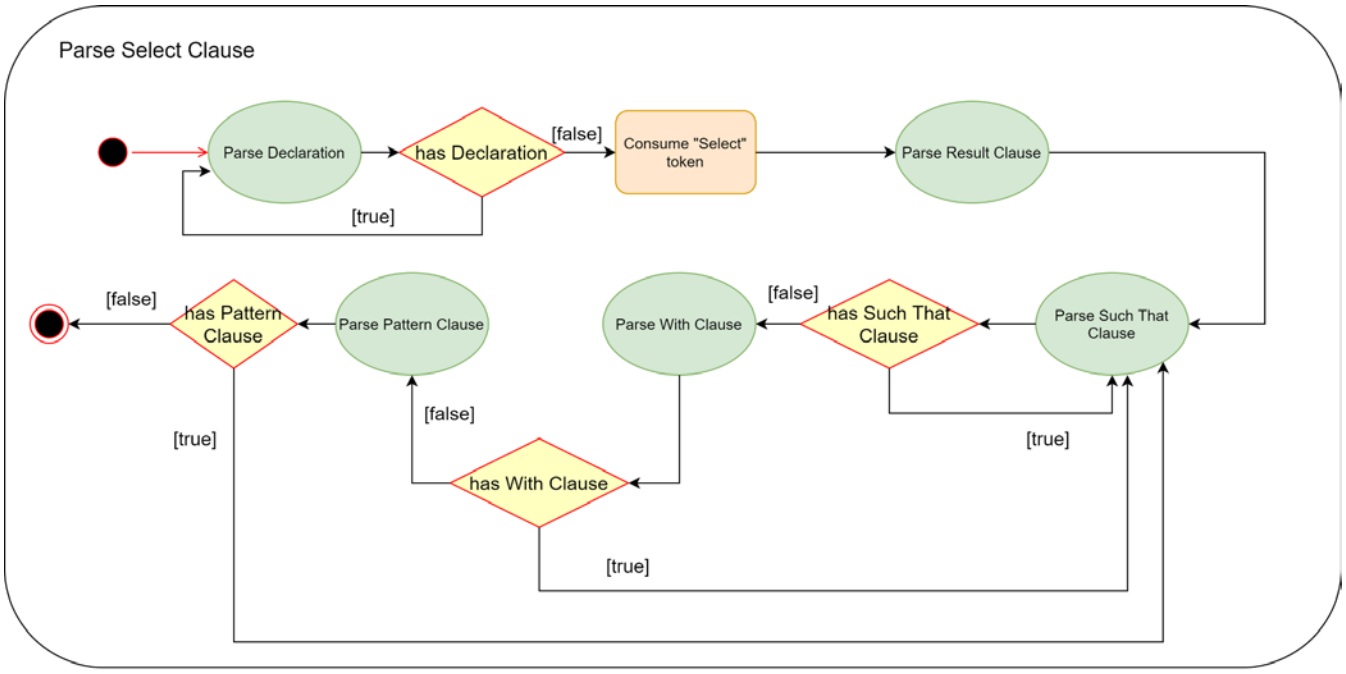


Figure 3.3.1.7: Activity Diagram of Parsing Select Clause

For the recursive descent approach, each function representing a non-terminal will recursively call other functions representing non-terminals if non-terminals exist in their grammar production. In the above diagram, The **select clause function** calls the non-terminal functions for **declaration, such that**

clause, pattern clause as well as **with clause.** Each of these functions that **select clause** calls will return true or false depending on whether the particular clause it represents was found.

Within the select clause, we will first attempt to parse a synonym declaration. This means we will call the declaration function whose activity diagram is shown below. Inside this declaration, we first check if the current token is a design entity token, returning true if it is and false otherwise to indicate whether a declaration was found.

Since we have **“assign a;”,** we can indeed consume the design entity token “assign”, identifier token “a” and semicolon token “;”, returning true. This process repeats for a total of 3 times, one for each of the declarations in **“assign a; while w; prog_line n;”.** We also store each declared synonym and its entity type inside a map in order to facilitate semantic checks.

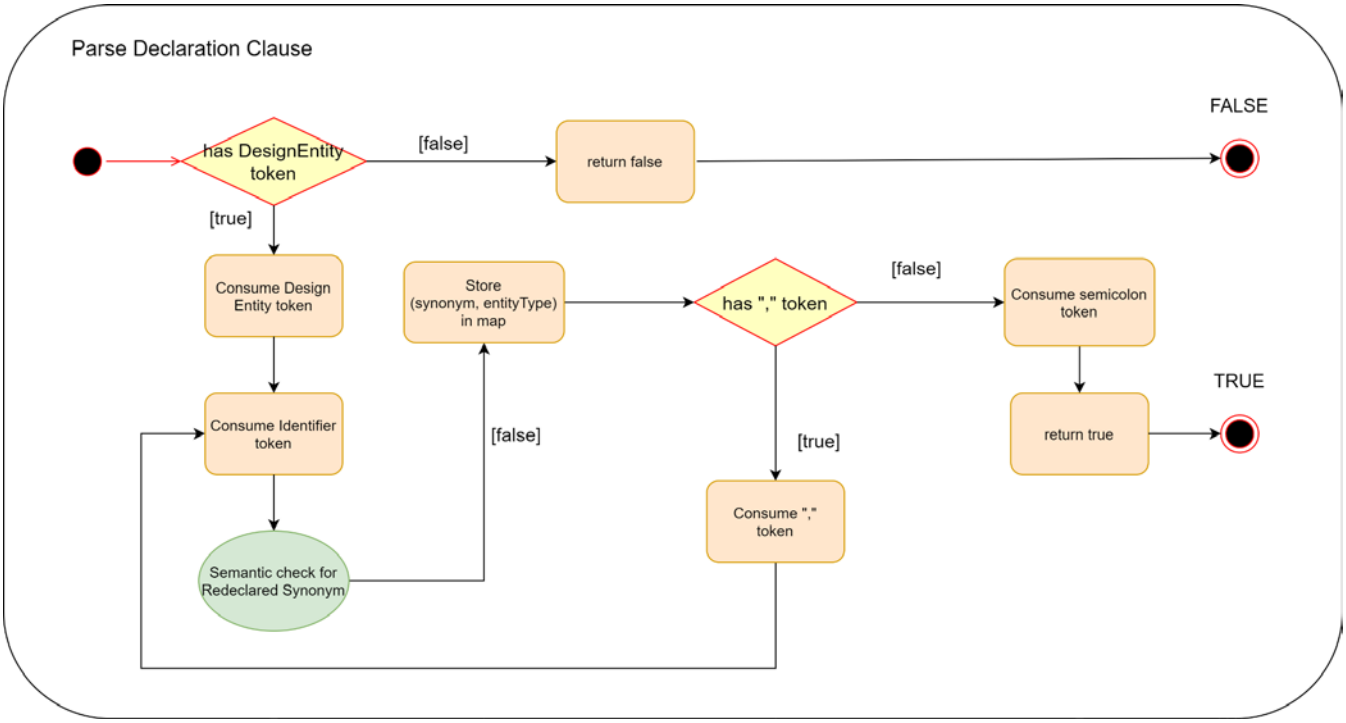


Figure 3.3.1.8: Activity Diagram of Parsing Declaration Clause

When there are no more declarations left, the declaration function will return false and the loop will be exited. The select clause is now able to proceed with parsing other types of clauses.
We now proceed to parse a result clause. The result clause is represented by a SelectClause object containing a vector of Declaration objects. For each of the synonyms within the tuple **“<a, n>”,** we create a Declaration and add it to the Query object’s SelectClause.

This is what the Query object's SelectClause looks like after the result clause has been parsed:

Query

aSelectClause	SelectClause

aDeclarations

Declaration

aEntityType	Assign
isAttribute	false
aValue	"a"

Declaration

aEntityType	ProgLine
isAttribute	false
aValue	"n"

Figure 3.3.1.9: Population of Query after parsing Result Clause

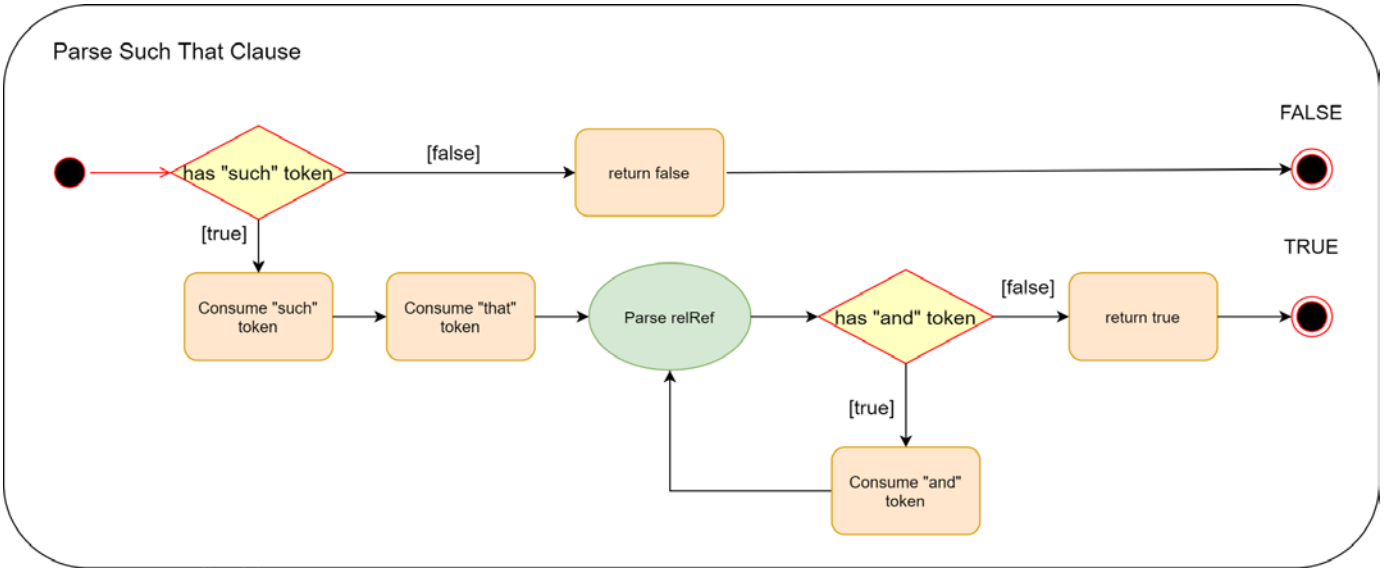


Figure 3.3.1.10: Activity Diagram of Parsing Such That Clause

The **selectClause function** now calls the **suchThatClause.** Since it is able to parse a “such” token from the next portion **“such that Parent*(w, a)”** of our example query, **suchThatClause** in turn calls **relRef. RelRef** looks at the current token type, and sees that it is “Parent”. Thus, it proceeds to parse **Parent**.

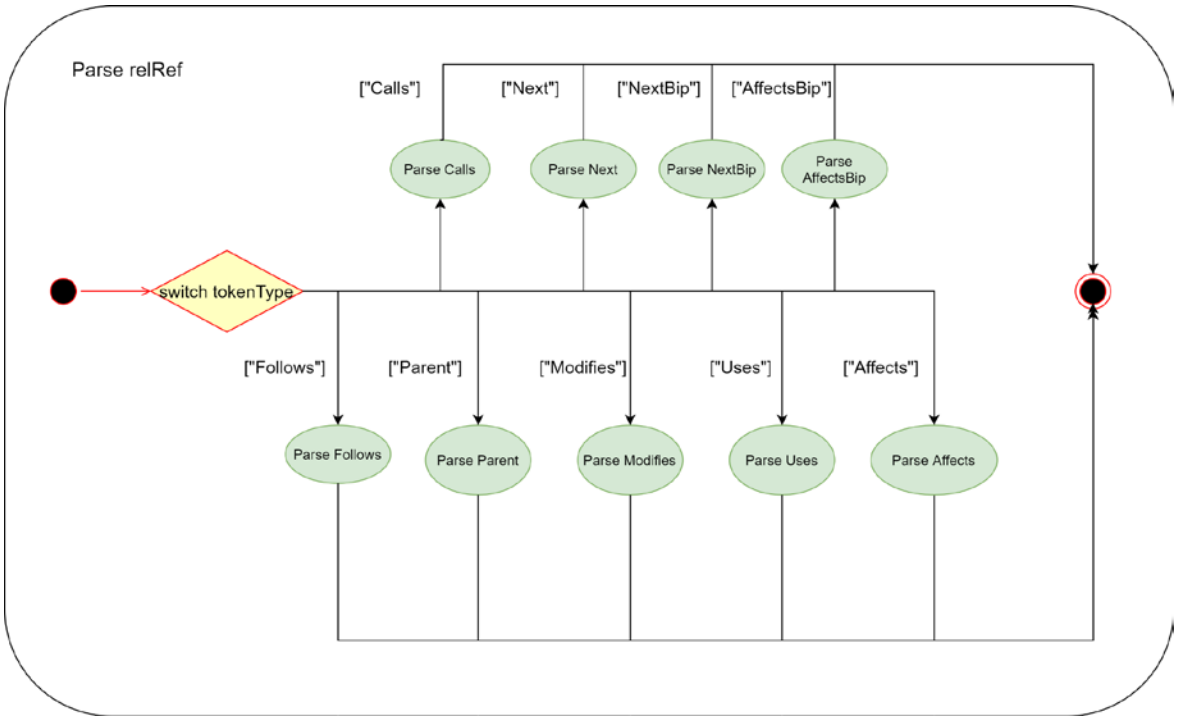


Figure 3.3.1.11: Activity Diagram of Parsing relRef

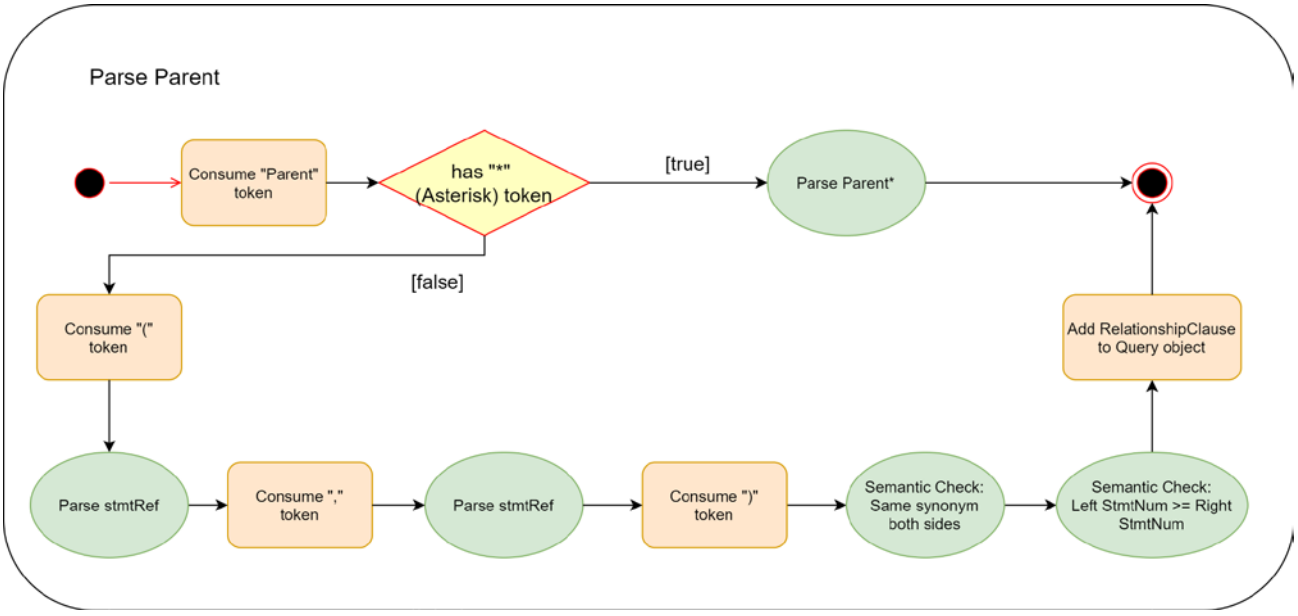


Figure 3.3.1.12: Activity Diagram of Parsing Parent

When parsing **Parent,** we differentiate between Parent and Parent* based on whether an asterisk token was encountered. However, the underlying parsing logic is exactly the same for both these relationships, with the only difference being the RelationshipType of the RelationshipClause object that we will add to the Query. we will call **stmtRef** twice, one for the left-hand side argument and the other for the right-hand side.

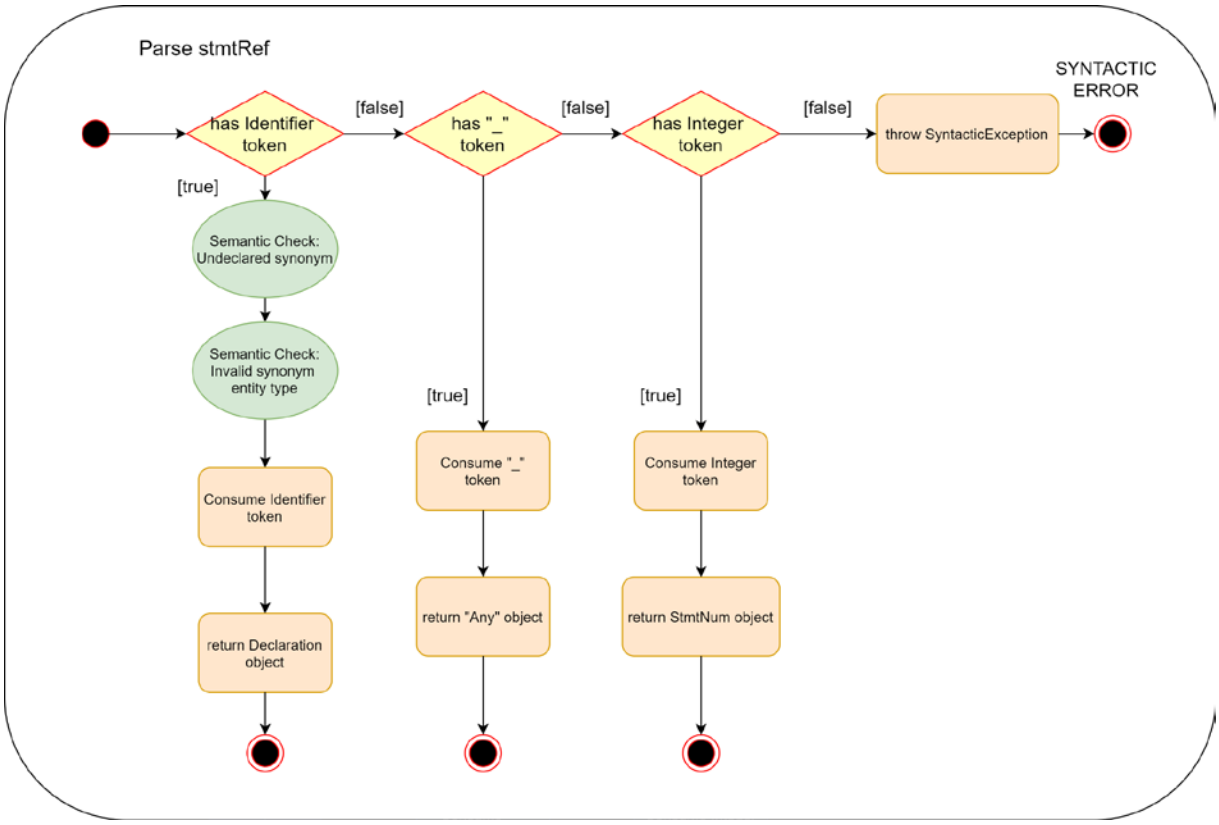


Figure 3.3.1.13: Activity Diagram of Parsing stmtRef

For the left-hand argument, we are able to consume an Identifier token of **"w"** which is a synonym of entity type "while". We first perform semantic checks on this synonym and only if these checks pass will we then create a Declaration object representing this synonym and return it from the stmtRef.

For the right-hand argument, since we are able to consume an Identifier token of **"a"** which is a synonym of entity type "assign", we will also create a Declaration object representing this argument and return it from the stmtRef.

Finally, after consuming all the relevant tokens in **Parent,** we wrap the above Declaration objects in a RelationshipClause object with the Relationship type of **ParentT** (transitive form of Parent

relationship). We then add the RelationshipClause to the Query's vector of OptionalClauses (which is the base class for RelationshipClause, PatternClause and WithClause).

The same process happens for parsing **"Next*(60, n)",** resulting in the following Query object thus far:

Query

aSelectClause	omitted for brevity...

aOptionalClauses	RelationshipClause

aRelationshipType	ParentT
aLeftInput	Declaration

aEntityType	While
isAttribute	false
aValue	"w"

aRightInput	Declaration

aEntityType	Assign
isAttribute	false
aValue	"a"

clauseType	Relationship

continued on the next page...

RelationshipClause

aRelationshipType	NextT
aLeftInput	StmtNum aValue: "60"
aRightInput	Declaration aEntityType: ProgLine isAttribute: false aValue: "n"
clauseType	Relationship

Figure 3.3.1.14: Population of Query object after parsing Relationship clauses

The select clause now moves on to parse the **pattern** clause. We check whether the entity type of the synonym in the pattern clause matches one of assign, if or while types. If not, this represents a syntactic error because pattern clauses are only defined for synonyms of these types. In this case we have **"pattern a("x", _)"** where **"a"** is of type **pattern-assign**, hence we proceed to parse a pattern-assign clause.

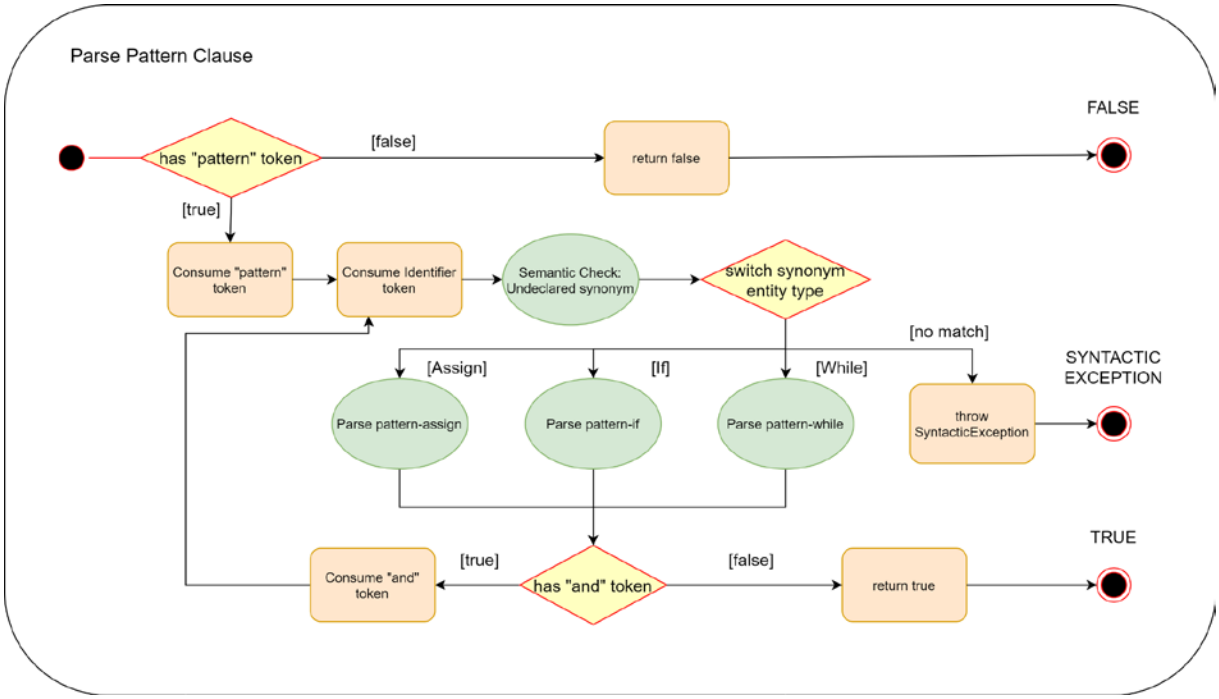


Figure 3.3.1.15: Activity Diagram of parsing Pattern Clause

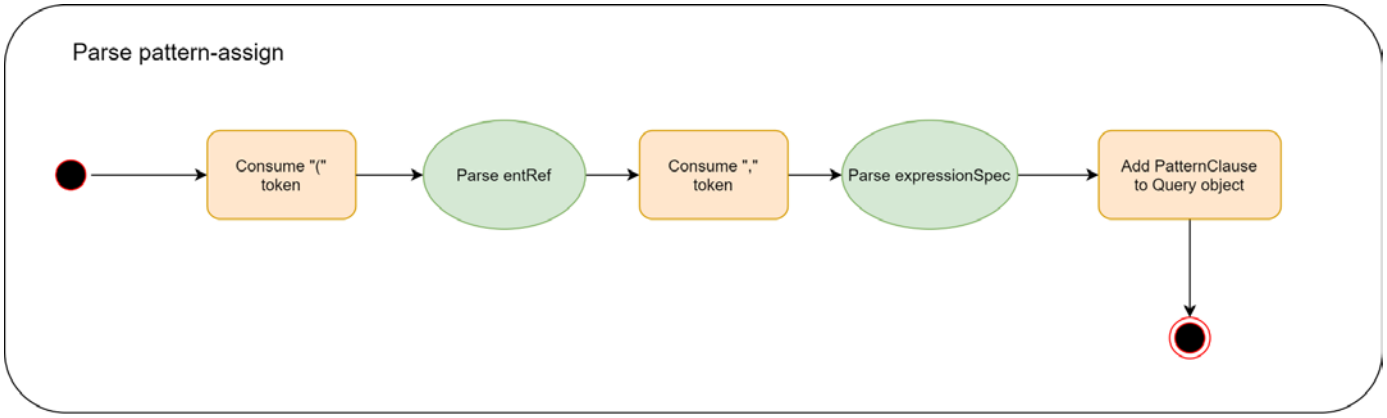


Figure 3.3.1.16: Activity Diagram of parsing Pattern-Assign Clause

The pattern-assign will then proceed to parse an **entRef,** to obtain an `Ident` object containing the value **"x"** representing the LHS argument's identifier. For the RHS argument, an **expressionSpec** will be parsed and an `Expression` object returned. We will then wrap the above objects in a PatternClause object and add this PatternClause to the Query object's vector of OptionalClauses.

The Query now looks like so:

<table>
<tr><td>Query

<table><tr><td>aSelectClause</td><td>omitted for brevity...</td></tr></table>

<table><tr><td>aOptionalClauses</td><td><table><tr><td>RelationshipClause</td></tr></table>
<table><tr><td>RelationshipClause</td></tr></table>
<table><tr><td>PatternClause

<table><tr><td>aLeftInput</td><td>Declaration

<table><tr><td>aEntityType</td><td>Assign</td></tr><tr><td>isAttribute</td><td>false</td></tr><tr><td>aValue</td><td>“a”</td></tr></table></td></tr><tr><td>aRightInput</td><td>Ident

<table><tr><td>aValue</td><td>“x”</td></tr></table></td></tr><tr><td>aExpression</td><td>Expression

<table><tr><td>aValue</td><td>“_”</td></tr><tr><td>type</td><td>ExpressionType::
EMPTY</td></tr></table></td></tr><tr><td>clauseType</td><td>Pattern</td></tr></table></td></tr></table></td></tr></table></td></tr></table></td></tr></table>

Figure 3.3.1.17: Population of Query object after parsing Pattern Clause

Lastly, we proceed to parse the **with clause.**

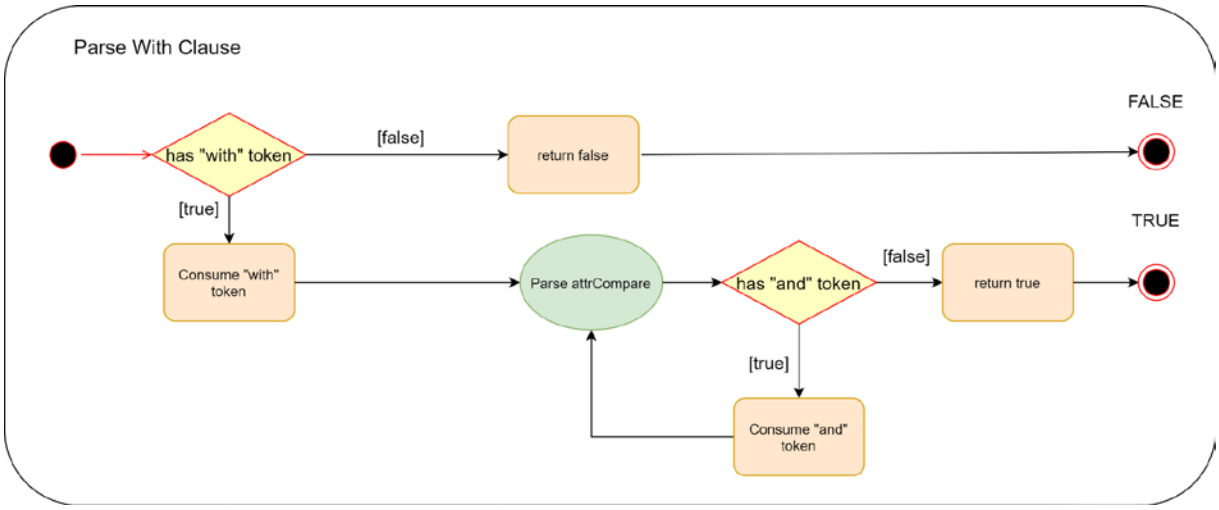


Figure 3.3.1.18: Activity Diagram of parsing With Clause

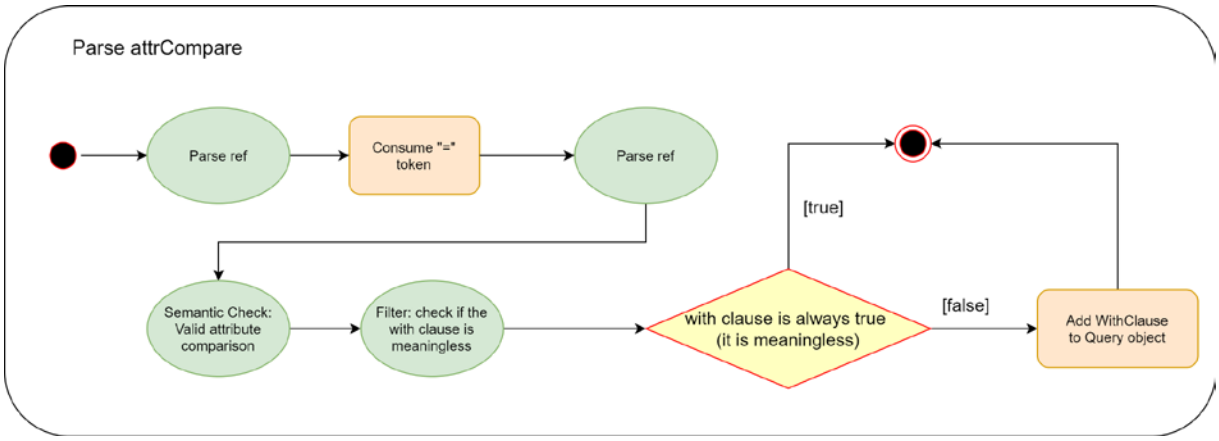


Figure 3.3.1.19: Activity Diagram of parsing attrCompare Clause

We only have a single attribute comparison in **"with a.stmt# = n".** For this clause, We first perform semantic checks such as ensuring that the LHS and RHS of the comparison are of the same data type. (e.g. s.stmt# = 3 is valid but s.stmt# = "procA" is invalid as LHS is an integer while RHS is a string) Next, we also filter out meaningless clauses such as 10 = 10 that are always true and can be removed safely without affecting the result. If the current attrCompare clause is deemed meaningless, we will not add the WithClause to the Query.

However, in this case since the attribute comparison is useful, we will add it to the Query.

At this stage, the parsing is complete. Here is the Query after population with the WithClause:

Query

aSelectClause	omitted for brevity...

aOptionalClauses

RelationshipClause

RelationshipClause

PatternClause

WithClause

aLeftInput	Declaration
	aEntityType: Assign isAttribute: false aValue: “a”
aRightInput	Declaration
	aEntityType: ProgLine isAttribute: false aValue: “n”
clauseType	With

Figure 3.3.1.20: Population of Query object after parsing With Clause

Semantic Validation

We have a QueryParserErrorUtility which will handle the throwing of exceptions: our own custom SyntacticException and SemanticException.

This QueryParserErrorUtility contains the logic of semantic validation so as to enforce separation of concerns to a large degree as code within the QueryParser can be focused on the parsing rather than being cluttered with the logic of validation checks.

However, we have not gone to the extent of performing the semantic validation only after the parsing is complete; in other words, we have chosen to take a one-pass approach. This is because we feel that abstracting the validation logic is already sufficient; whenever a QueryParser function needs certain types of semantic checks it can simply call the corresponding QueryParserErrorUtility API functions to enforce those checks on its grammar.

Let us take an example query and see how the semantic validation is enforced:

```
procedure p; variable v; assign a;
Select p.procName such that Uses(p, v) pattern a(v, _) with a.stmt# = 6
```

Declaration

We store all declared synonyms inside a table mapping the synonym value to its entity type. Below we illustrate the process of semantic validation on declaration clauses.

Semantic validation for Declaration Clause	
procedure p;	Before we add the synonym **"p"** to this synonyms table, we will call `QueryParserErrorUtility::semanticCheckRedeclaredSynonym`. This function will check whether the synonym already exists inside the table. In this case no such synonym exists, so the check passes and we add **("p", Procedure)** to the table.
variable v;	Likewise, before we add the synonym **"v"** to this synonyms table, we will call `QueryParserErrorUtility::semanticCheckRedeclaredSynonym`. Since no such synonym exists in the table, so the check passes and we add **("v", Variable)** to the table.
assign a;	Likewise, before we add the synonym **"a"** to this synonyms table, we will call

	`QueryParserErrorUtility::semanticCheckRedeclaredSynonym`. Since no such synonym exists in the table, so the check passes and we add **("a", Assign)** to the table.

We have now finished all semantic checking for declaration clauses, with the query passing all checks.

Below, we show more examples of valid and invalid declarations:

Example Declaration Clauses	
Redeclaration of Declared Synonym	We only allow synonyms to be redeclared as the same entity type as its previous declaration (which is meaningless but we allow it) **Example:** variable v; variable v; The above is **valid**. We however, do not allow any redeclarations of synonyms as a different entity type from its previous declaration. **Example:** prog_line p; print p; The above is **invalid**. When adding the second "p" to the table, we find that "p" already exists with a different entity type from the current declaration. procedure p; prog_line p; constant c; call c; These declarations are all **invalid** for the same reason.

Result Clause

For the Result clause to pass all semantic checks, all synonyms used within have to be well-defined. If an attribute reference is present, it has to be a valid attribute name for the synonym it is referenced on.

Semantic validation for Result Clause	
Select p.procName	Since we encounter an attribute reference, we will call semantic check functions to validate this attribute reference. We will first call `QueryParserErrorUtility::semanticCheckUndeclaredSynonym`. This function will check whether the synonym of the attribute reference already exists inside the table. In this the synonym **"p"** exists, thus the check passes. Next, we will call `semanticCheckInvalidAttrForSynonym.` This function will check whether the attribute name used in this reference is valid for the synonym. In this case the synonym **"p"** is of entity type Procedure and procName is a valid attribute name for Procedure synonyms, hence the check passes.

Example Invalid Result Clause	
Invalid Synonym -Attribute pair	**Example:** p.varName (where "p" is of type procedure) p.stmt# p.value Procedure synonyms only have procName as a valid attribute name.

Such That Clause

For the Such that clause to pass all semantic checks, all synonyms used within have to be well-defined. Each relationship imposes constraints on the entity types of synonyms that can be passed as its left argument and right argument.

We store this information in tables mapping the type of relationship to a set of allowed entity types, one table for the left argument and another for the right. Also, we also perform semantic checking on the left argument to ensure that the type of argument is valid with respect to the relationship. This is because for Uses and Modifies, "_" is not a valid left argument.

Semantic validation for Such That Clause	
Uses(p, v)	We will first retrieve the set of entity types which Uses can take as left argument. We then call `QueryParserErrorUtility::semanticCheckUndeclaredSynonym`. This function will check whether the synonym of the attribute reference already exists inside the table. In this the synonym **"p"** exists, thus the check passes. Next, we will call `semanticCheckValidSynonymEntityType.` This function will check whether the synonym is of valid entity type with respect to the relationship type. In this case the synonym **"p"** is of entity type Procedure, which is one of the allowed entity types for LHS argument of **Uses**, thus the check passes. For the synonym **"v"**, the above 2 semantic checks are also run and passed successfully.

In the below table, we see all the different entity types for synonyms that are allowed in the left and right side of a relationship reference.

Design Entity type (Left/ Right argument)	Stmt	Assign	If	While	Print	Read	Constant	Variable	Call	Procedure	Program Line
Follows / Follows*	✔/✔	✔/✔	✔/✔	✔/✔	✔/✔	✔/✔	✗/✗	✗/✗	✔/✔	✗/✗	✔/✔
Parent/ Parent*	✔/✔	✗/✗	✔/✔	✔/✔	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✔/✔
Uses	✔/✗	✔/✗	✔/✗	✔/✗	✔/✗	✗/✗	✗/✗	✔/✔	✔/✗	✔/✗	✔/✗
Modifies	✔/✗	✔/✗	✔/✗	✔/✗	✗/✗	✔/✗	✗/✗	✔/✔	✔/✗	✔/✗	✔/✗
Calls/ Calls*	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✔/✔	✗/✗
Next/ Next*/ NextBip/ NextBip*	✔/✔	✔/✔	✔/✔	✔/✔	✔/✔	✔/✔	✗/✗	✗/✗	✔/✔	✗/✗	✔/✔
Affects/ Affects*/ AffectsBip/ AffectsBip*	✔/✔	✔/✔	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✔/✔

When the relRef makes a call to a stmtRef or entRef function, it looks up our tables for the set of allowed entity types and passes it as argument. If a synonym is encountered, it must have an entity type from among the types included in this set. Otherwise, a semantic error will be thrown.

By allowing each relationship to control what are allowed entity types just by changing the values in the allowable entities table, we can easily extend semantic checks to include more relationships for future iterations.

Pattern Clause

For the pattern clause to pass all semantic checks, all synonyms used within have to be well-defined. The synonym of pattern clause can also only be of type assign, if or while. There are also constraints on the synonyms passed as its LHS argument and RHS argument.

Semantic validation for Pattern Clause	
pattern a(v, _)	We first call `QueryParserErrorUtility::semanticCheckUndeclaredSynonym`. This function will check whether the synonym of the attribute reference already exists inside the table. In this the synonym **"a"** exists, thus the check passes. Next, we will call `semanticCheckPatternClauseSynonym.` This function will check whether the synonym is of valid entity type with respect to a pattern clause. In this case the synonym **"a"** is of entity type Assign, which is one of the allowed entity types for the synonym of pattern, thus the check passes. When parsing the **entRef** of the pattern clause, we will call `semanticCheckUndeclaredSynonym.` In this the synonym **"v"** exists, thus the check passes. We also call `semanticCheckValidSynonymEntityType.` This function will check whether the synonym is of valid entity type with respect to the pattern type. In this case the synonym **"v"** is of entity type Variable, which is one of the allowed entity types for LHS argument of **pattern-assign**, thus the check passes.

Example Invalid Pattern Clause	
Invalid Pattern Synonym	**Example:** pattern c(_, _) (where “c” is of type call) pattern v(_, _) (where “v” is of type variable)
Invalid LHS argument	**Example:** pattern a(p, _) (where “a” is of type assign, “p” of type procedure) pattern a(w, _) (where “w” is of type while)

With Clause

For the With clause to pass all semantic checks, all synonyms used within have to be well-defined. If an attribute reference is present, it has to be a valid attribute name for the synonym it is referenced on. Furthermore, the data type of both LHS and RHS must be the same (either integers or string) for a comparison to be valid.

Semantic validation for With Clause	
with a.stmt# = 6	Since we encounter an attribute reference, we will call semantic check functions to validate this attribute reference. We will first call `QueryParserErrorUtility::semanticCheckUndeclaredSynonym`. This function will check whether the synonym of the attribute reference already exists inside the table. In this the synonym **“a”** exists, thus the check passes. Next, we will call `semanticCheckInvalidAttrForSynonym.` This function will check whether the attribute name used in this reference is valid for the synonym.

	In this case the synonym **"a"** is of entity type Assign and stmt# is a valid attribute name for Assign synonyms, hence the check passes. We then proceed to call `semanticCheckInvalidAttrCompare.` This function will check whether the data types of both LHS and RHS of comparison are the same. In this case since **"a.stmt#"** is an integer and **"6"** is also an integer the check passes.

Example Invalid With Clause	
Invalid Attribute name	**Example:** with a.value = 3 (where "a" is of type assign) with a.varName = "var1"
Mismatch of LHS and RHS types	**Example:** with a.stmt# = "three" with a.stmt# = "3" LHS is an integer while RHS is a string.

Design Decision: Storage of Query Clauses

Approach 1: Store such that, with and pattern clauses in separate lists (Design prior to Iteration 3)

Each type of clause RelationshipClause, PatternClause, WithClause would be stored in a separate list. In this case, there would be no need to create an overarching abstract class that would be the parent class of each of these query clauses.

Approach 2: Store such that, with and pattern clauses in a single list (Current Design)

Store all the clauses in a single list by creating an abstract class OptionalClauses.

Evaluation Criteria

There are 2 main criteria to consider when choosing between the options mentioned above. The first criteria would be the ease of implementation; the easier it is to implement, the more appealing it might be. The second criteria is to consider how well the option supports the optimisation of the

query evaluation process. Both criteria are important, but the second criteria outweighs the first because of time being a limiting factor when evaluating queries.

	Design considerations	
	Easy to Implement	Optimisation Support
Store such that, with and pattern clauses in separate lists	✔	
Store the clauses in a single list		✔

Final Decision and Rationale

Approach 2 was eventually chosen for Iteration 3, as reducing running time was one of the primary concerns for our group. The running time for our group was above the average for iteration 1, therefore paving the way for optimisation was a must. Storing all query clauses in a single list would allow us to compare and sort the clauses, and combine the tables in an order that would result in fewer rows of the intermediary tables.

Even though we had to rework our code and many of our test cases to accommodate the new OptionalClause class, it was still the right decision as it would allow our group to circumvent many of the running time issues we were facing.

Design Decision: Filtering of With Clauses

This section details our design decision pertaining to the filtering of meaningless with clauses that are always true such as “with 10 = 10” as well as early termination of the program whenever we encounter a With clause that is always false/always yields 0 results in its table such as “assign.stmt# = call.stmt#”. (Since an assignment statement can never be a call statement) This helps to reduce the running time of our SPA program.

Approach 1: The Query Evaluator will handle the filtering of these With clauses

The Query Evaluator will have to iterate through the list of OptionalClauses. For each OptionalClause that has a clauseType of With, the clause will have to be checked. If it constitutes a comparison that is always true, it will be removed from the list. If it constitutes a comparison that is always false, the

program will terminate early, returning an empty result list or a list with only “FALSE” for **elem/tuple** and **boolean** result clause respectively.

Approach 2: The Query Preprocessor will handle the filtering of With clauses (Current Design)

The Query Preprocessor will perform the above filtering and early termination as it is parsing the With clauses.

Evaluation Criteria

There are 2 main criteria to consider when choosing between the options mentioned above. The first criteria would be the ease of implementation. The second criteria is to consider how well we are able to adhere to the Single Responsibility Principle.

	Design considerations	
	Easy to Implement	Single Responsibility Principle
Query Evaluator handles the logic		
Query Preprocessor handles the logic	✔	✔

Final Decision and Rationale

Approach 2 was eventually chosen as there does not seem to be any benefits in letting the Query Evaluator handle the logic of filtering the With Clauses. It is easy to implement because the WithClause can be chosen to be added to the list of OptionalClauses based on whether it is “meaningless” when it is being parsed and there is no need to iterate over the list of OptionalClauses.

Furthermore, we have decided to classify WithClauses with attribute comparisons that are always false as an instance of SemanticException (e.g. “print.stmt# = assign.stmt# is always false as print cannot be an assignment statement). This is to leverage on our implementation of the TestWrapper to terminate early whenever a SemanticException is encountered and return the appropriate results accordingly (either “FALSE” or empty result list).

We do not want to introduce any new dependencies into our system since QueryEvaluator is not aware of QueryParserErrorUtility class which handles all the semantic and syntactic errors; only QueryParser is aware. Therefore, we have allocated this task to the QueryPreprocessor.

3.3.3 Query Evaluator

Design

The job of the Query Evaluator class is to evaluate a `Query` object using a PKB, and return the results in the form of a `ResultsTable` object:

```
QueryEvaluator(shared_ptr<QueryInterface> query, shared_ptr<PKBInterface> pkb);
shared_ptr<ResultsTable> evaluate();
```

With the help of the `ResultsUtil` and `QueryOptimizer` static classes, the Query Evaluator evaluates a query in several steps:

1. The individual results of each clause are first retrieved from the PKB and cached.
2. The order of the clauses are then sorted according to their results size.
3. Next, the clauses are separated into different groups, where in each group, each clause will contain at least one synonym that has already appeared in any of the previous clauses in the group.
4. For each group, merge the results of its clauses, producing an intermediate `ResultsTable` object.
5. Next, the intermediate `ResultsTable` objects of the groups are sorted by size.
6. Lastly, merge all the intermediate `ResultsTable` of the groups into one final `ResultsTable` object, which will be the final output of the Query Evaluator.

Steps 2, 3 and 5 are optimization steps that are performed using functions from the `QueryOptimizer` static class. The merging process done in steps 4 and 6 is achieved either by natural join or cartesian product using functions from the `ResultsUtil` static class. The final results table that is returned will be passed on to the `ResultsProjector`.

Implementation

This section will run through the query evaluation steps mentioned in the previous section in greater detail, and explain the implementation of the query evaluation using an example query:

```
while w; assign a; variable v, v1; print pn;
Select w such that Parent*(w, a) pattern w(v, _) such that Follows(_, a)
with v1 = pn.varName
```

This section will be broken down into several subsections, each to explain the implementation of a feature of the query evaluator.

Any reference to “example query” will be referring to this query above, and any reference to “step <number>” will be referring to the step number of the query evaluation steps in the previous section.

Retrieval and Format of PKB Results

This subsection focuses on explaining how the query evaluator retrieves the results of each clause and the data structure of PKB results used.

In general, clauses can be categorized by the number of synonyms they contain. As the PKB needs to return values of each synonym in the clause results, the data structure returned by the PKB would have to differ depending on the number of synonyms the clause has. Below is the data structure used for each category, and each format explained.

Number of Synonyms in Clause	Data Structure of Results	Format Explanation	Clause Examples
0	Boolean	No value of synonyms is needed to be retrieved, the query evaluator just needs to know if the clause is satisfied or not.	`Follows(1, 2)` `Parent(5, 10)`

1	Set of string	Values of a single synonym are needed, so storing all values in a set suffices.	`Follows*(s, 9)`
			`Pattern a(_, _)`
			`with v = v`

2	Map of key string mapped to set of string	The keys of the map will be the values of the left synonym in the clause. Each key maps to a set that contains values of the right synonym in the clause that pairs with the key to satisfy the clause.	`Uses(p, v)`
			`Pattern w(v1, _)`
			`with p = v`

Table 3.3.3.1: PKB Results Data Structures

This categorization applies for most clauses except for a special case of **with** clause that contains 2 synonyms, where the synonyms are the same. In this case, despite having 2 synonyms in the clause, as they are not distinct and refer to the same declaration, they would have the same values and it would not make sense to store their values in a map. Hence, the results of such **with** clauses are stored in a set. Example clauses: `with v = v`, `with p = p`, etc.

If any of the PKB results retrieved is empty or false, then the Query Evaluator will end the query evaluation process at this step, and return an empty `ResultsTable` object with the `noResult` boolean flag toggled to true. Details of the `ResultsTable` class is covered in the next subsection.

Therefore, the PKB API used by the Query Evaluator would only return the aforementioned three types of data structures. Refer to the PKB section for detailed explanation on the design of the PKB API, and the full abstract API used by the Query Evaluator can be found in the appendix, section 8. Refer to *Figure 4.2.1.1* in section 4 for the high-level sequence diagram of the interaction between the Query Evaluator and the PKB.

Referring to the example query, we will now go through which abstract API is called to retrieve each clause's results and how these results look. The results are retrieved from the PKB in step 1 of the query evaluation process. As the clauses are initialized as `OptionalClause` objects in the order in which they appear in the query, their result will be retrieved from the PKB in the same order as well.

Example clause	Number of synonyms in clause	Abstract PKB API called
`Parent*(w,a)`	2	`LIST_OF_LIST_RES getMapResultsOfRS(RELATION_TYPE, QUERY_INPUT)`
`Pattern w(v, _)`	2	`LIST_OF_LIST_RES getMapResultOfContainerPattern(STMT_TYPE, VAR_NAME)`
`Follows(_, a)`	1	`LIST_RES getSetResultOfRS(RELATION_TYPE, QUERY_INPUT)`
`with v1 = pn.varName`	2	`LIST_OF_LIST_RES getDeclarationMatchAttributeResults(SYNONYM, ATTR)`

Table 3.3.3.2: Abstract API Called for Example Clauses

`unordered_map<string, unordered_set<string>>`

values of synonym "w" maps to values of synonym "a"

key	value
5	6 | 7 | 8

Figure 3.3.3.1: PKB results of Parent(w, a) clause*

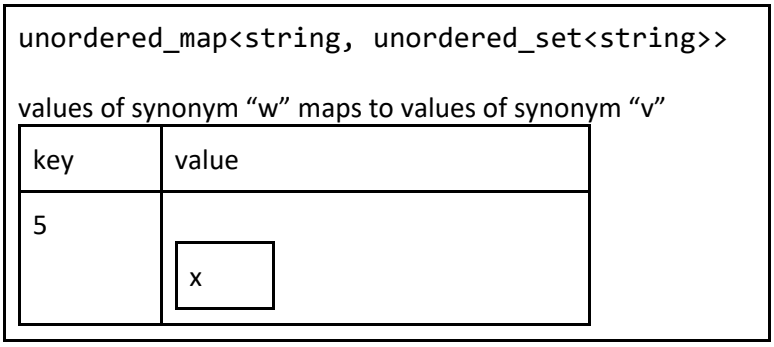


Figure 3.3.3.2: PKB results of Pattern w(v, _) clause

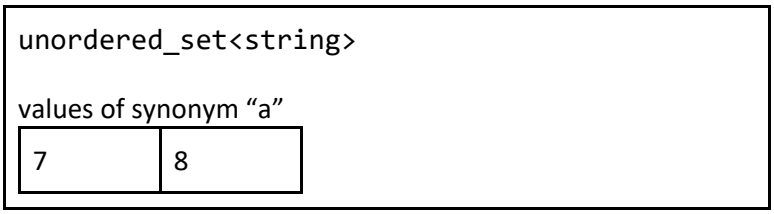


Figure 3.3.3.3: PKB results of Follows(_, a) clause

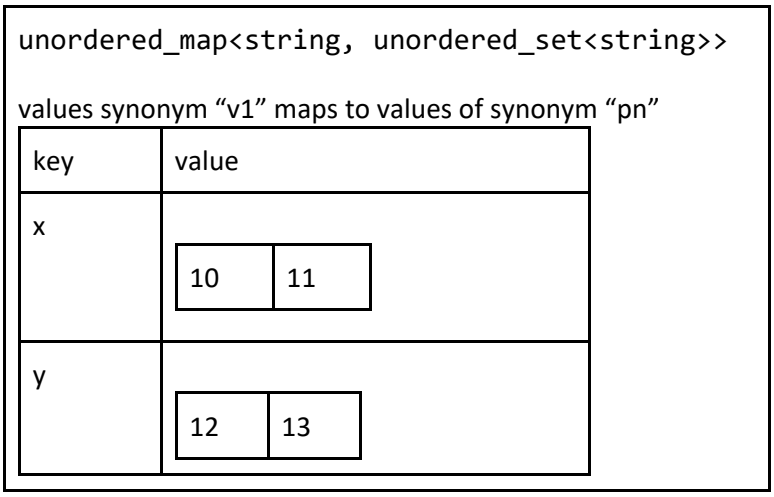


Figure 3.3.3.4: PKB results of with v1 = pn.varName clause

To clearly explain the map result format using this example, in *Figure 3.3.3.4* above the key "x" maps to a set with values "10" and "11". This means that v1 = "x" and pn = "10" is one result, and v1 = "x" and pn = "11" is another result, and so on for the other key-value pairs.

Do note that when retrieving results for attributes, the query evaluator only retrieves the statement number of the synonym of the attribute, and not the attribute value itself. This is done as the merging process operates on the statement numbers of the synonyms. So in *Figure 3.3.3.4* above, the statement numbers of "pn" are stored instead of the variables that are printed.

For **with** clauses where the value of the attribute is being set (eg. with pn.varName = “x”), the query evaluator will have to retrieve the attribute value of the synonym from the PKB using the synonym’s statement number to filter the statement numbers of the synonym that satisfy the clause.

Optimization

This subsection focuses on the rationale behind optimization choices and their implementation.
The optimization process in query evaluation has **two objectives**:

1. Minimize the time taken to perform merging operations (Small intermediate results table)
2. Discover empty results as fast as possible (Short-circuiting evaluation process)

To achieve these objectives, **sorting of clauses and groups by result size** and **separation of clauses into groups of connected synonyms** are implemented.

Sorting clauses and groups by result size

Rationale

In steps 2 and 5, the clause and group results are sorted by non-decreasing size before being merged together in later steps. This is done so that the smaller tables are being merged first, thus the size of the intermediate results table is kept to a minimum at any point in the merging process. Since the time complexity of merging operations is dependent on the size of the tables being merged, the average time taken to perform each merging operation is kept to the minimum if the clauses/groups are sorted by non-decreasing size.

This achieves **objective 1** as since the average time taken to perform each merging operation is smaller when the clauses/groups are sorted by non-decreasing size as compared to the average time of any other order of clauses/groups. Therefore, the overall total time taken by all merging operations performed is kept to the minimum.

Objective 2 is also achieved as since the intermediate results table is kept as small as possible from the start, the faster merging operations are performed first. Therefore, if an empty intermediate result is discovered mid-evaluation, the total evaluation time that has elapsed up until the point of discovering the empty result is kept to the minimum required.

Implementation Details

For sorting clauses, the results of each clause is retrieved from the PKB and cached into each `OptionalClause` object in step 1. `QueryOptimizer` takes in the vector of these `OptionalClause` objects, and uses the builtin function `sort()` from C++ Standard Template Library (STL) to sort the `OptionalClause` objects in the vector by result size.

For sorting group results in step 5, `QueryOptimizer` takes in the vector of `ResultsTable` objects, which represent the intermediate results table of each group after the results of its clauses are merged together. This vector is also sorted using the same `sort()` function.

Referring to the example query and the PKB results of the clauses in *Figure 3.3.3.1* to *Figure 3.3.3.4*, the following will be the input vector of the `OptionalClause` objects at step 2, and their result sizes:

	vector<OptionalClause>			
Clause	`Parent*(w, a)`	`pattern w(v, _)`	`Follows(_, a)`	`with v1 = pn.varName`
Result size (Number of rows)	3	1	2	4

Table 3.3.3.3: Vector of Clauses Before Sorting

After sorting, the output vector at step 2 will have the following order:

	vector<OptionalClause>			
Clause	`pattern w(v, _)`	`Follows(_, a)`	`Parent*(w, a)`	`with v1 = pn.varName`
Result size (Number of rows)	1	2	3	4

Table 3.3.3.4: Vector of Clauses After Sorting

In the subsequent steps, the clauses will be processed in this order.

Separating clauses into groups

Rationale

In step 3, the clauses are divided into groups where in each group, each clause will contain at least one synonym that has already appeared in any of the previous clauses in the group. This is done so that when merging the results of the clauses of a group, only the natural join merging operation is

performed as each clause will always have at least one common synonym with the intermediate results table.
Since cartesian product merging operation is never done and the clauses are sorted by non-decreasing results size whenever possible, the size of the intermediate results table during merging clause results within each group will be kept to the minimum, and will never exceed the maximum clause results size.

This achieves **objective 1**, as all merging operations done to merge clause results within each group will be faster since the time complexity of merging operations is dependent on the size of the tables being merged.

Objective 2 is achieved as well, as merging results of clauses in each group first means that natural join operations are performed first while cartesian product operations are only performed in later steps of the overall evaluation process when merging the intermediate results tables of groups together. This means that empty result tables that are produced from merging will be discovered earlier on in the evaluation process as only natural join operation can produce empty tables.

Implementation Details

In order to divide the clauses into groups with connected synonyms as described before, a modified Union-Find algorithm is used. The algorithm maintains a disjoint-set data structure to keep track of the groupings, as each clause can only be part of one group. Each synonym in the query will be associated with a group. Furthermore, the clauses in each group are stored as a `ClauseList` object, a linked list data structure used so that appending a clause to a group or appending a group to another group can be done in constant time.

For each clause, the algorithm determines which group to assign the clause to by looking up which group its synonyms are associated with. If all its synonyms belong to the same group, then the clause is simply appended to the group's `ClauseList` object. However, if the clause's left and right synonyms belong to different groups, the clause is first appended to the left synonym's group. Next, the algorithm checks if either the right synonym's group is empty or the first clause in the right synonym's group has common synonyms with the current clause. If so, then the current clause is able to connect its left synonym group with its right synonym group. Therefore, the right synonym's group `ClauseList` object is appended to that of the left synonym's group. All synonyms that were associated with the right synonym's group will now be associated with the left synonym's group.

After the algorithm has processed every clause, it will return a 2-dimensional vector, where each inner vector holds `OptionalClause` objects, representing each group. Since the input vector already has the clauses sorted in non-decreasing result size, the order of the clauses in each group will also be in non-decreasing result size order as well wherever possible, only this order is broken when two groups were combined during the algorithm.

The following will be a step-by-step run through of the union-find algorithm using the example query and the sorted vector of clauses in *Table 3.3.3.4* as the input of the algorithm. The current state of the groups during the algorithm is represented by the table below:

Synonyms associated with Group:	w	a	v1	v	pn
Group# (for reference):	Group 1	Group 2	Group 3	Group 4	Group 5

Table 3.3.3.5: Start of Union-Find algorithm, empty group state

Initially, no clause has been grouped yet, so each synonym is associated with its own group. The first clause processed is `Pattern w(v, _)`. The clause's left synonym is "w" and so the clause is first added to group 1. The clause's right synonym is "v" which is associated with group 4. Since group 4 is empty, the algorithm will union group 1 and 4 by associating "v" with group 1 instead:

Synonyms associated with Group:	w, v	a	v1	pn
Group# (for reference):	Group 1	Group 2	Group 3	Group 5
	`Pattern w(v, _)`			

Table 3.3.3.6: Pattern w(v, _) added to group 1, group 5 appended to group 1

Next, we consider clause `Follows(_, a)`. It's only synonym is "a", which is associated with group 2. Thus, `Parent(_, a)` is added to group 2:

Synonyms associated with Group:	w, v	a	v1	pn

Group# (for reference):	Group 1	Group 2	Group 3	Group 5
	`Pattern w(v, _)`	`Follows (_, a)`		

Table 3.3.3.7: Follows(_, a) added to group 2, group 3 appended to group 2

Next, we consider `Parent*(w, a)`. In this case, the left synonym "w" belongs to group 1 while the right synonym "a" belongs to group 2. `Parent*(w, a)` is first added to group 1. Since the first clause of group 2, `Follows(_, a)` has a common synonym with the current clause `Parent*(w, a)`, group 2 is appended to group 1:

Synonyms associated with Group:	w, v, a	v1	pn
Group# (for reference):	Group 1	Group 3	Group 5
	`Pattern w(v, _)`		
	`Parent*(w, a)`		
	`Follows(_, a)`		

Table 3.3.3.8: Parent(w, a) added to group 1, group 2 appended to group 1*

Lastly, we consider `with v1 = pn.varName`. The clause's left synonym is "v1" and so the clause is first added to group 3. The clause's right synonym is "pn" which is associated with group 5. Since group 5 is empty, the algorithm will union group 3 and 5 by associating "pn" with group 3 instead:

Synonyms associated with Group:	w, v, a	v1, pn
Group# (for reference):	Group 1	Group 3
	`Pattern w(v, _)`	`with v1 = pn.varName`
	`Parent*(w, a)`	
	`Follows(_, a)`	

Table 3.3.3.9: Final groups - with v1 = pn.varName added to group 3, group 5 appended to group 3

At this point, there are no more clauses left to process and the algorithm ends. A 2-Dimensional vector with 2 inner vectors will be returned, with the first inner vector containing the `OptionalClause` objects for `Pattern w(v, _), Parent*(w, a)` and `Follows(_, a)` (representing group 1) and the second inner vector containing only the `OptionalClause` object for `with v1 = pn.varName` (representing group 3).

Below is the activity diagram to visualize the steps in processing each clause in the Union-Find algorithm:

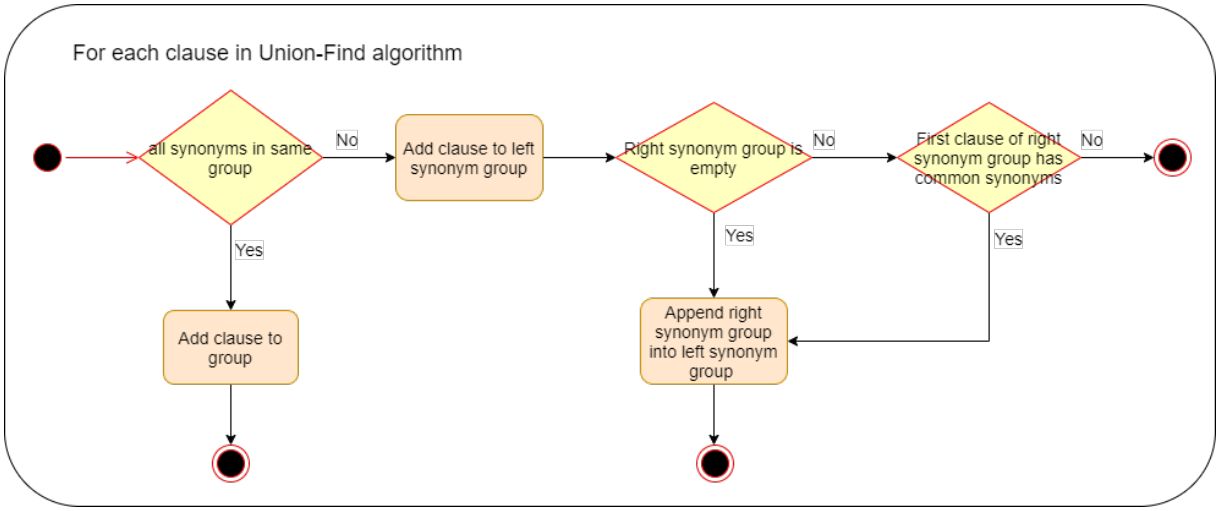


Figure 3.3.3.5: Activity diagram of Union-Find algorithm for each clause

Merging Results

This subsection focuses on the implementation of the operations and data structures used in the merging process of clause and group results in steps 4 and 6 of the query evaluation process.

The `ResultsTable` class is used in the query evaluator to encapsulate information of each result from the PKB, and is meant to represent the results as a table. A `ResultsTable` object maintains a 2-dimension vector that stores the values of the synonyms, with each inner vector representing a row in the table. The object also maintains a map that maps the synonym names to an integer, where this integer is the synonym's index in the row. Lastly, `ResultsTable` has a no`Result` boolean flag which is to be toggled to true if a clause returns empty results.

Now we will take a look at an example of how the results of a clause would look like in a `ResultsTable` object. Refer to the example query and the final groups from *Table 3.3.3.9*.

Let's run through step 4 of the query evaluation process where the results of clauses within each group will be merged first.

At the start of this merging process, an empty `ResultsTable` object is created where results of each clause will be merged into. For group 1, `pattern w(v, _)` is the first clause with its PKB results as shown in *Figure 3.3.3.2* in the previous subsection. Since the intermediate ResultsTable object is empty, the PKB results of `Pattern w(v, _)` will populate the empty `ResultsTable` object as shown:

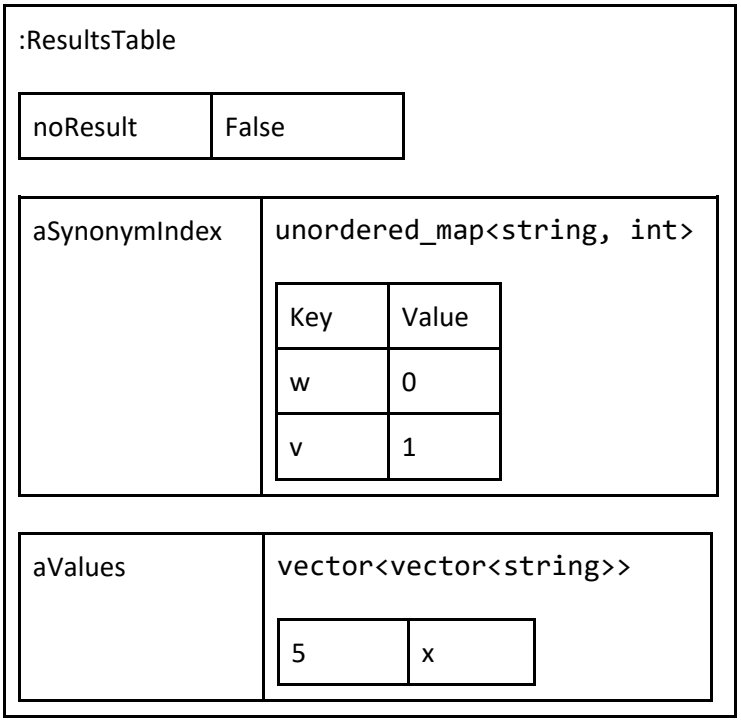
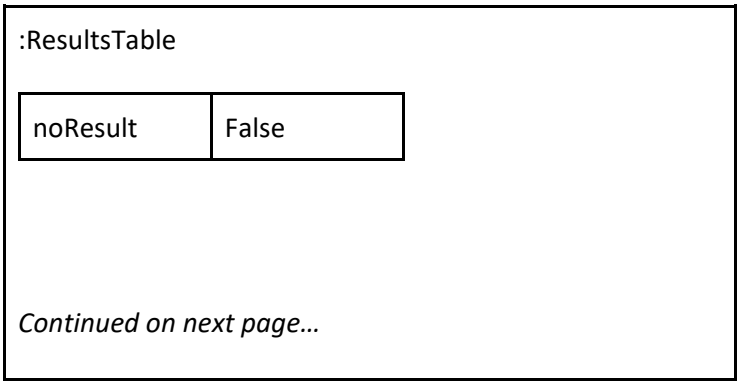


Figure 3.3.3.6: ResultsTable object of Pattern w(v, _) clause results

The same will occur in group 3, where the first clause, `with v1 = pn.varName` will have its PKB results (refer to *Figure 3.3.3.4*) populate an empty `ResultsTable` object as well:



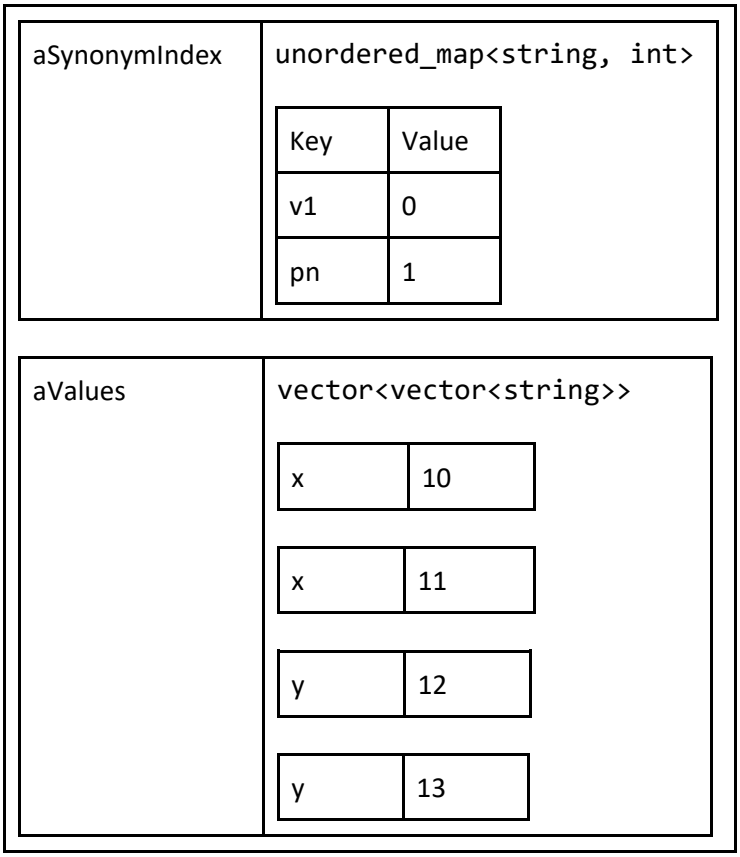


Figure 3.3.3.7: ResultsTable object of with v1 = pn.varName clause results

Since `with v1 = pn.varName` is the only clause in group 3, the merging process is done for this group, and the `ResultsTable` object in *Figure 3.3.3.7* above is returned as the intermediate results table for group 3. However, the merging process is not done yet for group 1, and we will continue the go through this example after explaining the first merging operation, natural join, in the next section.

When merging a clause result into an existing results table, the merging operation performed depends on the case in which the clause falls under:

Case		**Merging Operation Performed**	**Example clauses**
1: Clauses with no synonyms		-	`Follows(1, 2)`
2: Clauses with at least one synonym	**2a:** Clauses with some common synonyms with current ResultsTable	Natural Join	`pattern a(v, _"x"_)` `Uses(p, v)`
	2b: Clauses with no common synonym with current ResultsTable	Cartesian Product	

Table 3.3.3.10: Cases for Different Merging Operations Performed

Case 1: Clauses with no synonyms

For **clauses with no synonyms**, an API provided by the PKB that returns a boolean will be called. If the clause is satisfied, the query evaluator will pass over and continue on to the next clause as there are no values to be added to the `ResultsTable` object. However, if the PKB returns false meaning the clause is not satisfied, then the query evaluator will toggle `noResult` to true in the `ResultsTable` object and return it immediately.

Case 2a: Clauses with some common synonyms with current ResultsTable

For **clauses with at least one synonym**, there are 2 further subcases. When the clause has **some common synonym with current ResultsTable**, merging is done by "natural join" where the resultant `ResultsTable` object of the merge is obtained by adding the uncommon synonyms from the PKB result, and contains only rows where the values of all common synonyms match.

Referring to the merging process in group 1, we will see how natural join is performed on the intermediate `ResultsTable` object in *Figure 3.3.3.6* and the PKB results of the next clause in the group, `Parent*(w, a)`. The PKB results of `Parent*(w, a)` can be found in *Figure 3.3.3.1*.

:ResultsTable

noResult	False

aSynonymIndex	unordered_map<string, int>

Key	Value
w	0
v	1
a	2

aValues	vector<vector<string>>

5	x	6
5	x	7
5	x	8

Figure 3.3.3.8: Intermediate ResultsTable object after merging with Parent(w, a) clause results*

The synonym ‘w’ is common between `Parent*(w, a)` and the current `ResultsTable` object. Thus, only the entries in the current `ResultsTable` object with the same value of ‘w’ as an entry in the PKB result will be kept, with the corresponding value of the uncommon synonym ‘a’ being added to that entry.

In this example, only w = 5 matches, so the corresponding values of a = 6/7/8 will be added to entries where w = 5, creating a new entry for each value of “a”. Furthermore, ‘a’ being mapped to index 2 will be added to the aSynonymIndex map in the resultant `ResultsTable` object to indicate that the 3rd value in every inner vector corresponds to ‘a’.

The natural join operation is performed yet again when merging the last clause, `Follows(_, a)`, in group 1 with the `ResultsTable` object in *Figure 3.3.3.8* above. The PKB results of `Follows(_, a)` can be found in *Figure 3.3.3.3*.

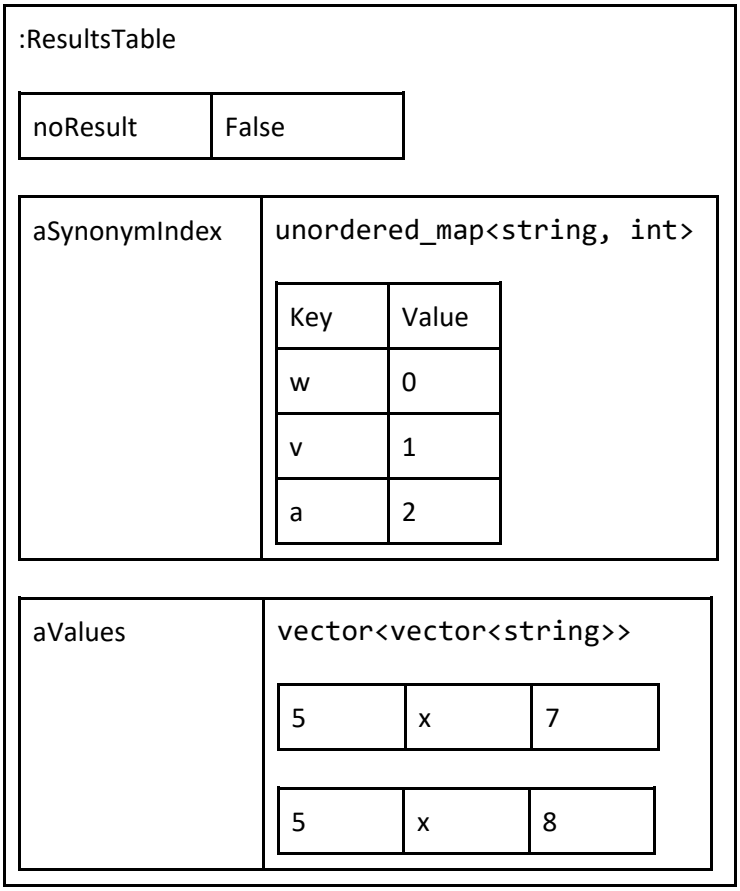


Figure 3.3.3.9: Intermediate ResultsTable object after merging with Follows(_, a) clause results

The synonym ‘a’ is common between `Follows(_, a)` and the current `ResultsTable` object and there is no uncommon synonym. Thus, only the entries in the current `ResultsTable` object with the same value of ‘a’ as an entry in the PKB result will be kept
In this example, only a = 7, a = 8 match, so the resultant results table will only contain entries that have the “a” value match any of these.

Since all clauses in group 1 are merged, the `ResultsTable` object in *Figure 3.3.3.9* above is added to a vector of `ResultTable` objects together with the `ResultsTable` object in *Figure 3.3.3.7* that was for group 3. In step 5, this vector is then sorted by non-decreasing table size, similar to step 2. However, in the example, as there are only 2 `ResultsTable` objects, their order does not affect query evaluation timing.

Case 2b: Clauses with no common synonym with current ResultsTable

When the clause or table has **no common synonyms with current ResultsTable**, the merging done is taking the cartesian product of the two tables. This is done as the query evaluator at this point does not know which synonym will be eventually selected. Furthermore, this allows for tuples to be readily selected as the final output.

Referring back to the example query, we will see how the cartesian product operation is performed on the ResultTable objects of groups 1 and 3. The ResultTable objects of groups 1 and 3 can be found at *Figure 3.3.3.9* and *Figure 3.3.3.7* respectively.

:ResultsTable

noResult	False

aSynonymIndex	unordered_map<string, int>

Key	Value
w	0
v	1
a	2
v1	3
pn	4

aValues	vector<vector<string>>				
	5	x	7	x	10
	5	x	7	x	11
	5	x	7	y	12
	5	x	7	y	13
	5	x	8	x	10
	5	x	8	x	11
	5	x	8	y	12
	5	x	8	y	13

Figure 3.3.3.10: Final ResultsTable object after merging ResultsTable objects of both groups

In this example, there are no common synonyms, thus the resultant `ResultsTable` object contains the cartesian product of both tables of the two `ResultsTable` objects. The synonyms “p” and “v1” are added to the first `ResultsTable` object’s aSynonymIndex map and mapped to values 3 and 4 to indicate that their values are in the 3rd and 4th index of each row in the resultant table.

Merging using natural join or cartesian product is done by a `ResultUtil` class that handles all the logic of the different types of merging. This is done so that the details of how the results are merged are abstracted out of the Query Evaluator class as its sole responsibility should be on evaluating the query, adhering to Single Responsibility Principle (SRP).

Below is the flow of the query evaluator’s evaluation process using an activity diagram:

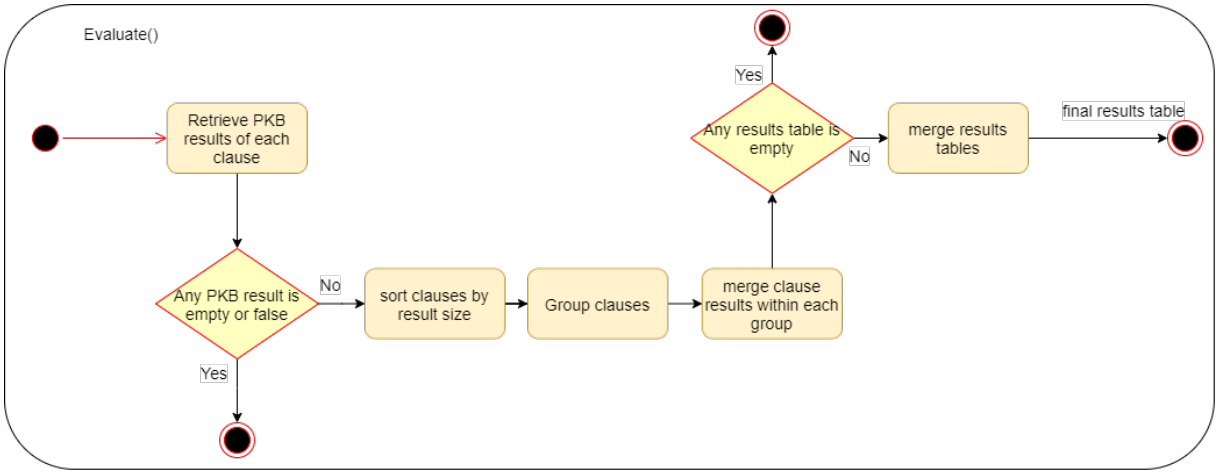


Figure 3.3.3.11: Activity diagram of the query evaluation process

Design Decisions: Query Evaluation Method

When planning how the Query Evaluator would evaluate the queries and deal with the different relationships between the clauses, two approaches were considered (including the current design). This section will detail the considerations for all alternatives and provide justification for the design chosen based on some evaluation criteria.

Approach 1: Modelling results as tables (current design)

This approach was to model the results of each clause as a table, and merging results would be performing natural join or cartesian product operations on the tables depending on the presence of common synonyms.

Approach 2: Modelling query as a constraint satisfaction problem

In this approach, the query will be modelled as a constraint satisfaction problem where the clauses are constraints on the synonym's domain of values. A graph must be formed where the nodes are the synonyms, and the edges/arcs between 2 nodes represent a constraint. To evaluate the query, the AC-3 algorithm (Arc-Consistency algorithm #3) is used on a graph where it examines the arcs between synonyms, and then removes values in the synonyms' domains that do not satisfy the arc/clause.

Evaluation Criteria

Ease of implementation: Approach 1 is less complex in the algorithms used as natural join and cartesian product only requires iterating through the list of results. Implementing the AC-3 algorithm in approach 2 is not a trivial task as it requires research and knowledge of arc-consistency and constraint satisfaction problems.

Time complexity: Let n be the maximum domain of a synonym. The cartesian product and natural join operations in approach 1 runs in $O(n^4)$ time complexity: maximum number of entries in a result table of a clause from the PKB is $y = n^2$, and time complexity of forming the cartesian product of 2 such tables is $O(y^2) = O(n^4)$. On the other hand, AC-3 has a time complexity of $O(en^3)$ where e is the number of constraints/clauses.

Final Decision and Rationale

Approach 1 was eventually chosen, mainly for its ease of implementation. Given the fact that both approaches work as well and are equally valid solutions to the problem of evaluating queries, approach 1 could be delivered with fewer possibilities of bugs and be more thoroughly tested due to the limited development time in iteration 1 and the developer's inexperience in C++. Even though approach 2 may be more efficient theoretically, this may not be the case when the number of clauses becomes very large. Furthermore, approach 2 would require more extensive testing as it requires more classes to be implemented, since a graph has to be modelled in the AC-3 algorithm.

3.3.4 Results Projector

Design

`ResultsProjector` is a static class with the job of returning the final output of the SPA given the results returned from the query evaluator, the **select** clause and the PKB objects. It does this through a single static function `projectResults`:

```
static void projectResults(shared_ptr<ResultsTable> evaluatedResults, shared_ptr<SelectClause>
selectClause, shared_ptr<PKBInterface> PKB, list<string>& results);
```

Implementation

The `SelectClause` object contains a vector of `Declaration` objects that represents the set of synonyms that are being selected. The size of this vector indicates what is being selected. If it has size 0, then the select clause in the query is selecting BOOLEAN. if it has size 1, then either a single synonym/attribute is being selected or a tuple with single synonym/attribute is being selected. These 2 cases are being grouped together as the format of the final results output is the same for both cases. Lastly, if the vector has a size of more than 1. then a tuple with multiple synonyms/attributes are being selected.

How the ResultsProjector returns the final output is based on which case the size of the Declaration vector and the `ResultsTable` object from the query evaluator fall under:

Cases		**Size of `Declaration` vector**
Case 1: Select Boolean		0
Case 2: Select Single Synonym/Attribute or Tuple with single Synonym/Attribute	**Case 2a**: `ResultsTable` object contains selected synonym/attribute	1
	Case 2b: `ResultsTable` object does not contain selected synonym/attribute	
Case 3: Select Tuple with more than one synonym/attribute	**Case 3a**: `ResultsTable` object contains **all** selected synonyms/attributes	More than 1
	Case 3b: `ResultsTable` object does not contain **all** selected synonyms/attributes	
	Case 3c: `ResultsTable` object contains **some, but not all** selected synonyms/attributes	

Table 3.3.4.1: Different Cases for Selecting Results

For every case, `ResultsProjector` first checks if the `noResult` flag in the `ResultsTable` object is set to true. In case 1, if the `noResult` flag is true, then "FALSE" will be added to the final results list and returned. In cases 2 and 3, if the `noResult` flag is true, then the empty final results list is returned immediately.

If the `noResult` flag is false, then the `ResultsProjector` will carry on the process of selecting values to be projected as explained in the following sections:

Case 1: Select Boolean

Since the `noResult` flag in the `ResultsTable` object was already checked, the `ResultsProjector` will simply add "TRUE" to the final results list and return it immediately.

Case 2a: `ResultsTable` object contains selected single synonym/attribute

In this case, the ResultsProjector just needs to extract all distinct values of the selected synonym/attribute from the results table.

If a synonym is selected, the values of the synonym in the results table is added to the final results list. However, if an attribute is selected, then the ResultsProjector will have to retrieve the attribute value from the synonym statement number since the result table only contains the statement numbers of the synonym of attributes. This is done using the abstract PKB API:

```
STRING getNameFromStmtNum(STRING).
```

Using *Figure 3.3.3.10* as the ResultsTable object from the QueryEvaulator, if the select clause is `select a`, the final results list will return with: `7, 8`. If the select clause is `select pn.varName`, the `ResultsProjector` will retrieve the corresponding variables from the PKB using the values of pn in the results table: 10, 11, 12, 13. Thus, the final results list will return with: `x, y`. (For example, `getNameFromStmtNum(10)` returns "x", so pn.varName = "x" when pn = 10)

Case 2b: `ResultsTable` object does not contain selected synonym/attribute

In this case, the `ResultsTable` object does not contain the synonym from the declaration in the **Select** clause, the `ResultsProjector` will call the PKB API below to get all the values of the synonym's entity type:

```
LIST_OF_ENTITY getEntities(ENTITY_TYPE)
```

If the selected declaration is not an attribute, the values from the API above are added to the final results list and returned. If the selected declaration is an attribute, each of the values from the API above will be called as a parameter to the API mentioned in case 2a:

```
STRING getNameFromStmtNum(STRING)
```

The value returned from this API is then added to the final results list and returned.

Using *Figure 3.3.3.10* as the `ResultsTable` object from the QueryEvaulator, if the select clause is `select ifs`, where ifs is an if declaration, the `ResultsProjector` will retrieve all statement numbers of if statements from the PKB and return them in the final results list.

If the select clause is c.procName, where c is a call declaration, then the `ResultsProjector` will retrieve all call statement numbers first. For each of these call statement numbers, the procedure name being called at the statement number will be retrieved from the PKB and added to the final results list.

Case 3a: `ResultsTable` object contains all selected synonyms/attributes

In this case, the `ResultsProjector` can simply iterate through each row in the results table, and concatenate the values of the selected synonyms in the required order. The concatenated values will then be added to the final results list.

Using *Figure 3.3.3.10* as the `ResultsTable` object from the QueryEvaulator, if the select clause is `select <w, a>`, the final results list will contain the values: `5 x, 5 y`.

Case 3b: ResultsTable object does not contains all selected synonyms/attributes

In this case, all the synonyms/attributes in the selected tuple are not in the results table. Therefore, the `ResultsProjector` has to retrieve the values of all selected synonyms/attributes and store them in a new `ResultsTable` object. The results will then be projected from this new `ResultsTable` object in the same manner as in case 3a with the `ResultsTable` object from the `QueryEvaluator`.

Using *Figure 3.3.3.10* as the `ResultsTable` object from the `QueryEvaulator`, if the select clause is `select <ifs, c>`, where ifs and c are synonyms of if and call declarations respectively, then the `ResultsProjector` will retrieve the PKB results of ifs and c:

`unordered_set<string>`

values of synonym “ifs”

15	19

Figure 3.3.4.1: PKB results of ifs declaration

`unordered_set<string>`

values of synonym “c”

3	4

Figure 3.3.4.2: PKB results of c declaration

An empty `ResultsTable` object is first created, and the PKB results above will be merged into it using the merging functions from `ResultUtil` static class. The final `ResultsTable` object will be as shown:

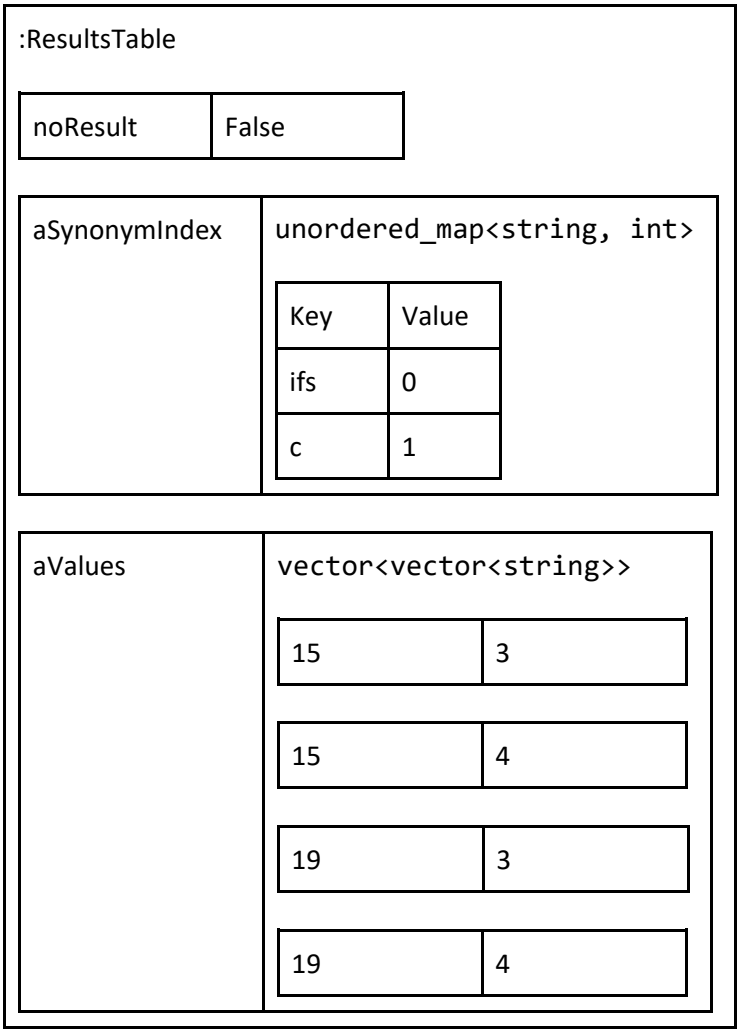


Figure 3.3.4.3: ResultsTable object of selected tuple synonyms

Therefore, the final results will be projected from the `ResultsTable` object above, so the final result list will return with the values: `15 3, 15 4, 19 3, 19 4.`

Case 3c: ResultsTable object contains some, but not all selected synonyms/attributes

In this case, the tuple contains a mix of synonyms/attributes that are in the results table and not in the results table. Therefore, for the selected synonyms not in the results table from the query evaluator, the `ResultsProjector` has to retrieve their values from the PKB and merge them into the `ResultsTable` object from the `QueryEvaluator`. The resultant `ResultsTable` object will then contain all selected synonyms/attributes in the tuple, and so the results will be projected from it into the final results list like in case 3a.

Using *Figure 3.3.3.10* as the `ResultsTable` object from the `QueryEvaulator`, if the select clause is `select <a, rd>`, where "rd" is a synonym for a read declaration and "a" is the synonym in the `ResultsTable` object. Thus, the `ResultsProjector` will retrieve the PKB results of "rd":

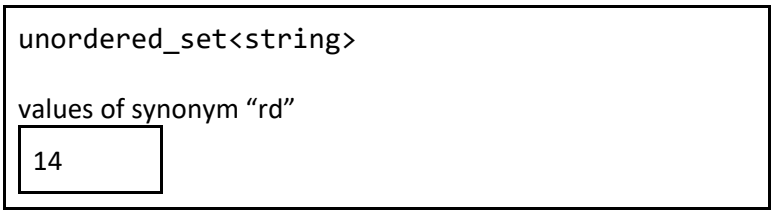
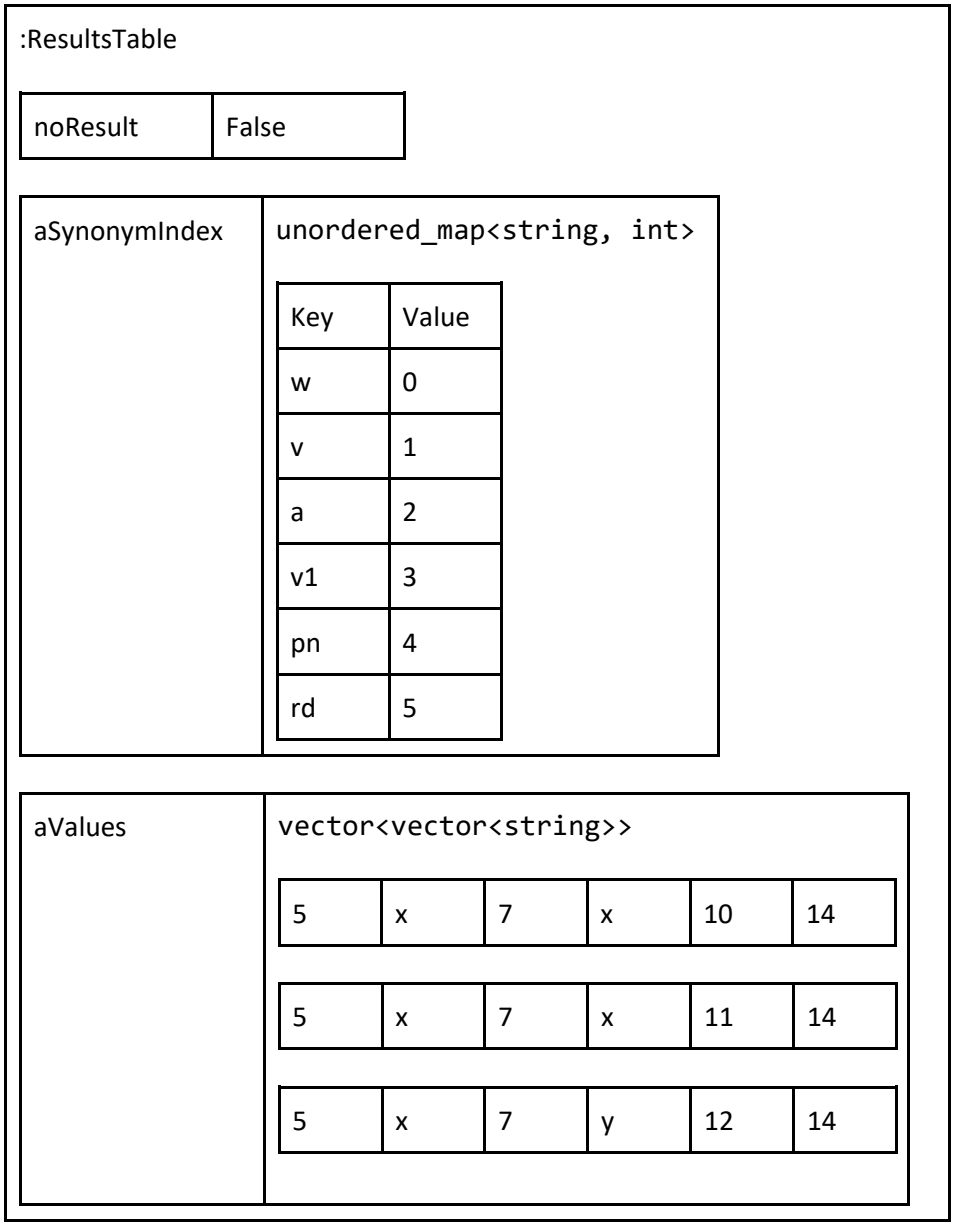


Figure 3.3.4.4: PKB results of rd declaration

Next, this PKB result will be merged into the `ResultsTable` object from the `QueryEvaluator`:



	5	x	7	y	13	14
	5	x	8	x	10	14
	5	x	8	x	11	14
	5	x	8	y	12	14
	5	x	8	y	13	14

Figure 3.3.4.5: Final ResultsTable object after merging with PKB results of "rd"

Therefore, the final results will be projected from the `ResultsTable` object above, so the final results list will be returned with values: `7 14, 8 14`.

Below is the activity diagram of the `ResultsProjector`'s `projectResults` function to illustrate the flow of the results projection process:

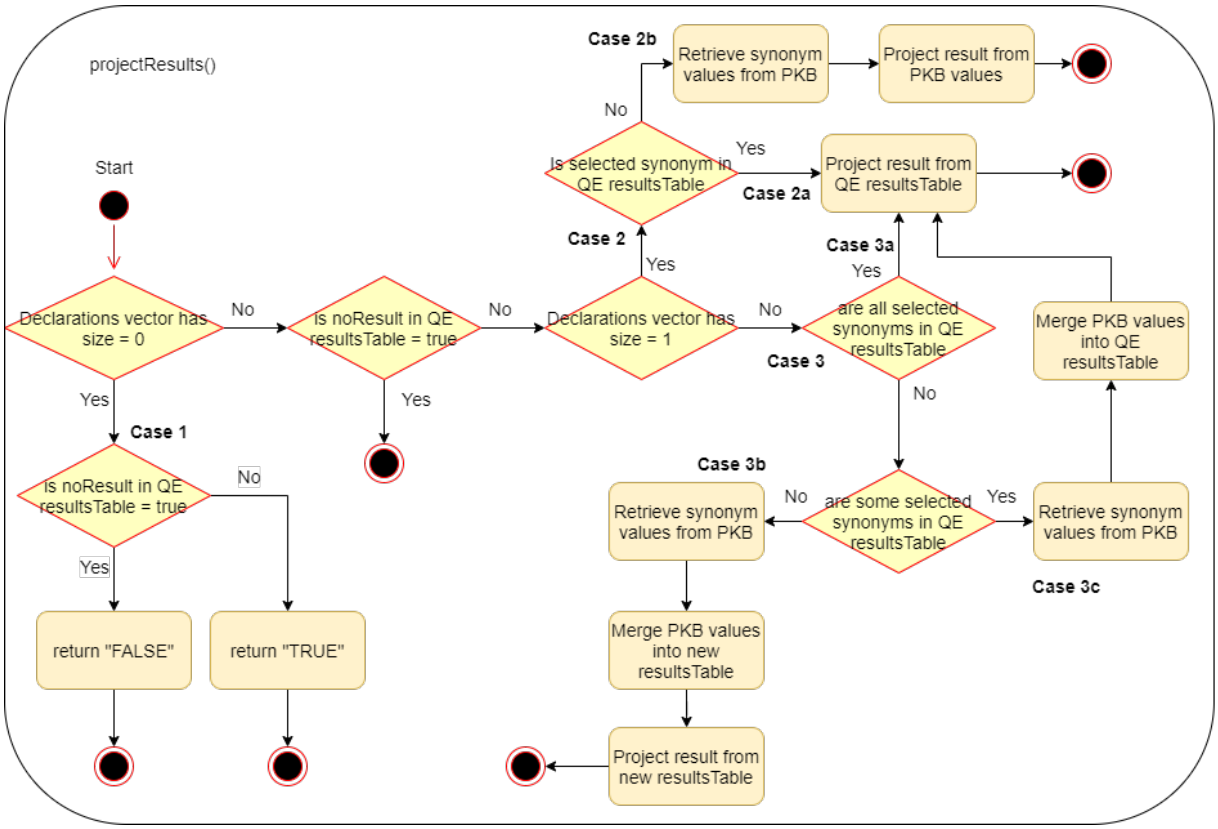


Figure 3.3.4.6: Activity Diagram of `ResultsProjector`'s `projectResults` function

Below is a high-level sequence diagram to illustrate the `ResultsProjector`'s interaction with the PKB:

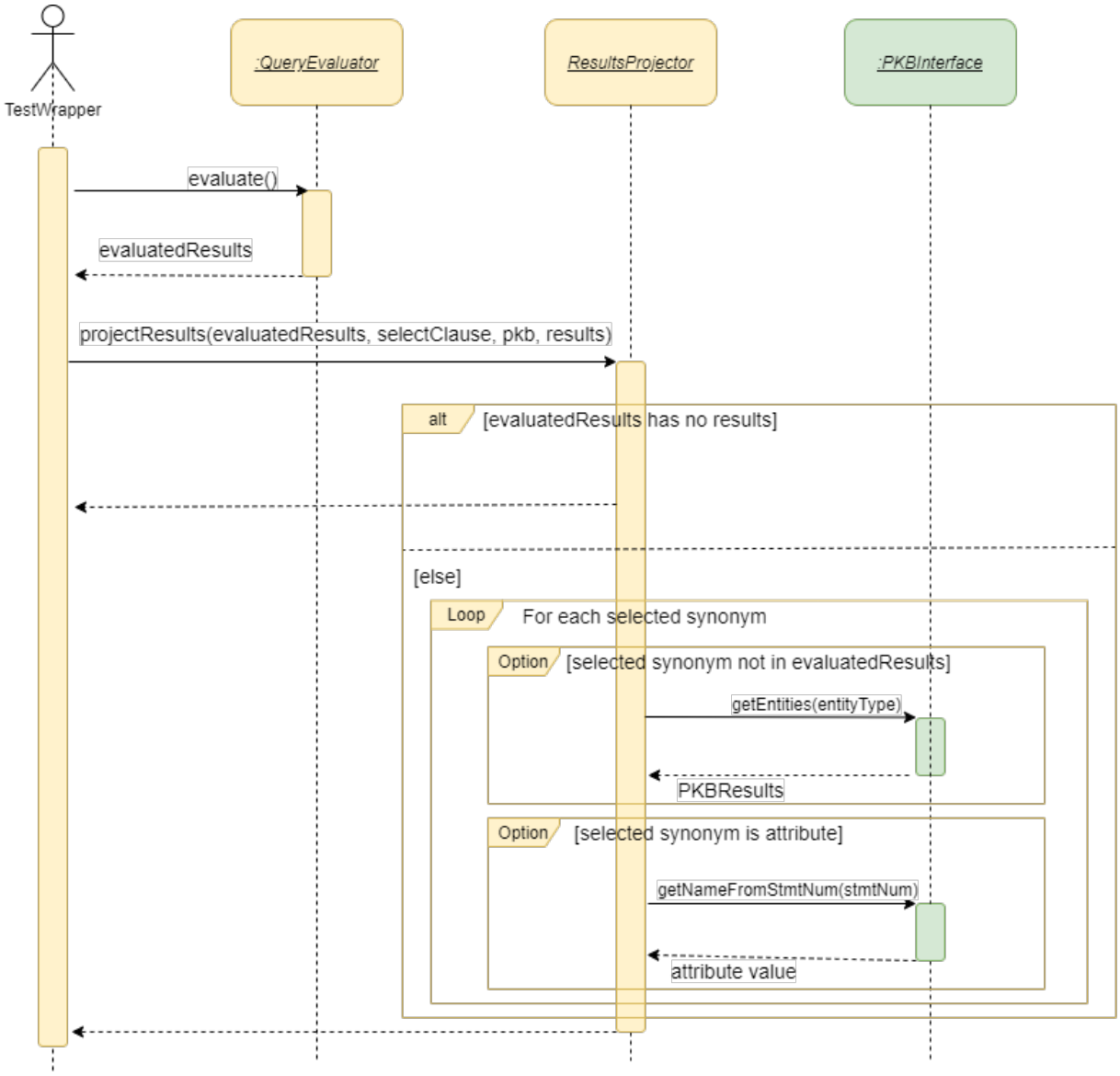


Figure 3.3.4.7: Sequence Diagram of `ResultsProjector` and PKB interaction

Design Decisions

Much consideration was made into deciding if there was a need for a results projector subcomponent within the Query Processor component, and how to structure it. This section will detail the considerations for all alternatives and provide justification for the design chosen based on some evaluation criteria.

Approach 1: Separate Results Projector from Query Evaluator (current design)

This approach split the job of the query evaluation process into two - one is to retrieve and merge results from the PKB, and the other is selecting the values for the final input. In this case, the Results Projector is separate from the Query Evaluator, designed as a static class on its own to handle the second job as mentioned.

Approach 2: Results Projection done in Query Evaluator

In this approach, the Query Evaluator would select the values for the final input immediately after evaluating the results of the clauses. This task would be designed as a function in the Query Evaluator class instead.

Evaluation Criteria

Ease of implementation: Approach 1 involves an extra class, which would mean more code needs to be written, and more testing is required. This may not be worth it as the job of the results projector is not overly complex, and may be easier to implement in a function in approach 2.

Single Responsibility Principle: Each class should not have multiple responsibilities, so that any changes in the SPA requirements would result in minimal changes in the class code.

Final Decision and Rationale

Approach 1 was eventually chosen, mainly for its adherence to the single responsibility principle. By separating the Results Projector and Query Evaluator, each class can focus on a single responsibility which leads to better design and cleaner code. Even though approach 2 may be easier to implement in iteration 1 and 2, this would not be the case in iteration 3, when select tuple/boolean is a requirement. With select tuple/boolean, the code needed to be implemented in selecting values for the final input significantly increases. Thus, segregating this job into its own class results in neater code organisation as well in iteration 3 and avoids code bloat in the Query Evaluation class.

4 Testing

Week No.	Iteration 1 Tasks Allocation			Rationale
Week 3	Design Unit Testing for all components			Design of Unit testing is planned at the same week as the implementation of SPA so that once a certain component is done, unit testing can be commenced immediately
Week 4	Design test cases for system testing. Perform Unit Testing			Implementation of individual components are done, thus, unit testing is executed before integrating different components together. In addition, Designing test cases for System Testing also start so that system testing can be executed once all components are integrated together.
Week 5	Integration Testing for Query Preprocessor and Query Evaluator	Integration Testing for PKB and Query Processor	Integration Testing for PKB and Parser	Integration tests are done simultaneously with Integration of different components.
Week 6	System Testing			System test is executed once all components are integrated together

Week No.	Iteration 2 Tasks Allocation			Rationale
Week 8	Design of SIMPLE source code and PQL Queries			Discussion had been made about which extension is supposed to be done in iteration 2. Thus, Design of SIMPLE source code and PQL queries can start simultaneously with the implementation to save time.
Week 9	Design of PQL Queries	Integration Testing for all components		Continue to discover possible cases while waiting for all components to be integrated
Week 10	System Testing			System test is executed once all components are integrated together

Week No.	Iteration 3 Tasks Allocation			Rationale
Week 11	Design of SIMPLE source code			Design of SIMPLE source code can be done as the remaining group members focus on the remaining requirement of our SPA program
Week 12	Designing of PQL Queries	Unit Testing		Designing of PQL queries can be done before Integration Testing so that System Testing can start immediately once all components are integrated together. Implementation of individual components is done, thus, unit testing is executed before integrating different components together.
Week 13	Integration Testing	System Testing		Integration Testing can start once each individual component is tested and working accurately. System Testing can then start once all components are integrated and working correctly.

4.1 Unit Testing

4.1.1 PKB Examples

The sample test cases of PKB are based on the following SIMPLE program, where relevant relationships are manually inserted into the PKB to ensure separation from other components:

```
procedure main {
    while (1 == 2) {
        m = 1;
        while (x == 0) {
            while (true) {
                call first;}}}
}

procedure first {
    while (y + 1 == t) {
        call second;}
}

procedure second {
    count = 0;
    count = count + z;
    count = count + 10;
}
```

	Test Purpose	Required Test Inputs	Expected Test Results

PKB	Verify that Uses / Modifies of multiple procedures can be correctly retrieved, and hence correctly inserted and stored. This is a whitebox test as we know uses / modifies of procs are handled separately.	`modifies(p, v)`
		
		This is simulated by calling:
		
		`getMapResultOfRS (modifies, PROC, VAR)`
			`(main, {m, count})`
			
			`(first, {count})`
			
			`(second, {count})`

PKB	Verify that calls* relationship can be correctly retrieved, and hence correctly inserted and stored.	`calls*(first, main)`
		
		This is simulated by calling:
		
		`getBooleanResultOfRS (calls*,first,second)`
			`false`

4.1.2 PQL Processor

	Test Purpose	Required Test Inputs	Expected Test Results
Query Evaluator	Verify that natural join merging operation is successfully performed when evaluating query with clauses that have some common synonym	Query Evaluator is initialized with: 1. Query object that represents the syntactically valid query: "`stmt s; assign a; variable v; Select a such that Uses(s, v) pattern a(v, _)`" 2. PKB stub object that is set to return non empty results for each of the clauses in the query above.	ResultsTable object that contains values equivalent to the values initialized in the PKB stub and has rows in which the values of the common synonym v match from the results of both clauses.
Results Projector	Verify that the Results Projector is able to successfully extract out the values of the selected synonym into the final results list	1. ResultsTable object representing a non empty table of values 2. SelectClause object that represents the syntactically valid select clause: "`Select s`"	Final results list that is populated with only the values of the selected synonym s from the ResultsTable object.
Tokenizer	Verify that correct number of tokens are generated, and are of the correct types and order	Tokenizer is initialized with: 1. A query input string "assign a; stmt s; Select a"	Tokens generated: DesignEntity, Identifier, Semicolon, DesignEntity, Identifier, Semicolon, Select, Identifier
Query Parser	Verify that the Query object is populated correctly with information about various clauses	Query Parser is initialized with: 1. Empty Query object 2. A query input string "`prog_line pgl1, pgl2; Select pgl1 such that Next(pgl1, pgl2)`" 3. Tokenizer stub object that is set to return hard-coded tokens corresponding to the input string	Query object is populated correctly: 1. Select clause has synonym of type prog_line and value "pgl1" 2. There is only one relationship clause of relationship type Next added with left and right query inputs both of type Declaration, with the first having value "pgl1" and second "pgl2"

4.2 Integration Testing

4.2.1 Interaction between SPA Components

Interaction between Source Processor and PKB

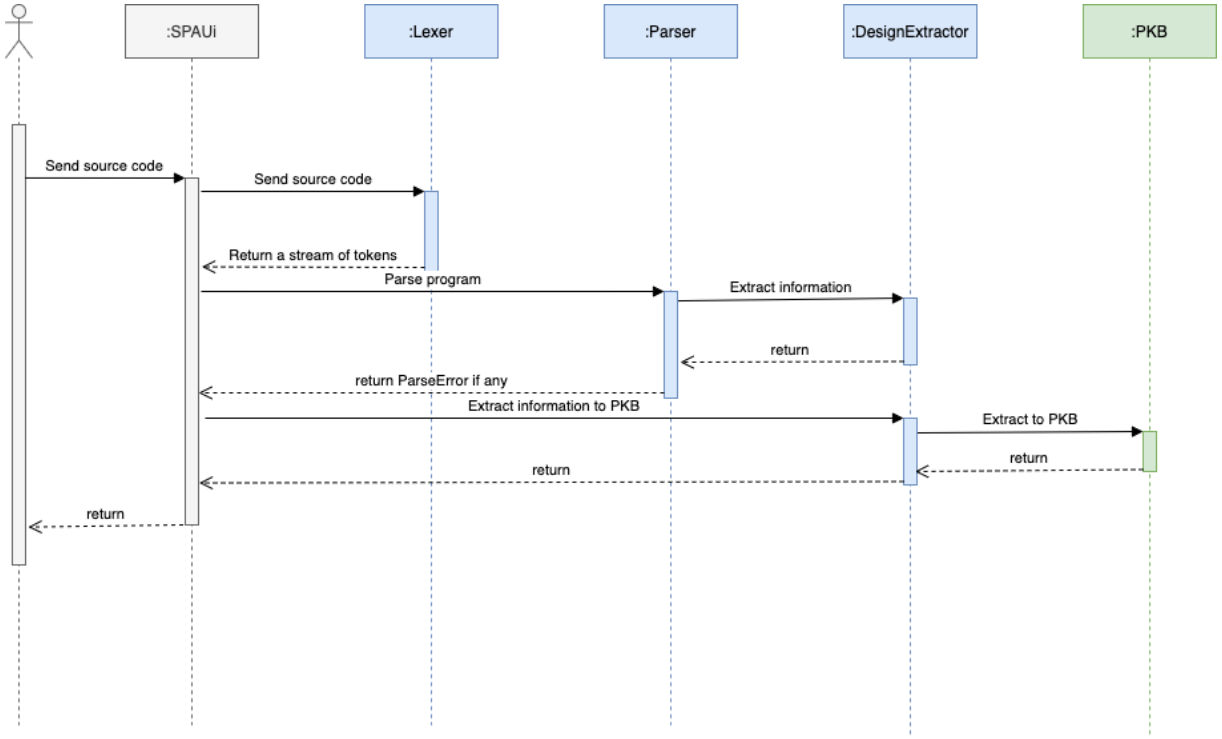


Figure 4.2.1.0: High level interaction between Source Processor

Figure 4.2.1.0 shows the condensed high-level interaction between components of the Source Processor and the PKB. The DE will simply call PKB APIs to sequentially send all the organized data and relationships.

Interaction between Query Processor and PKB

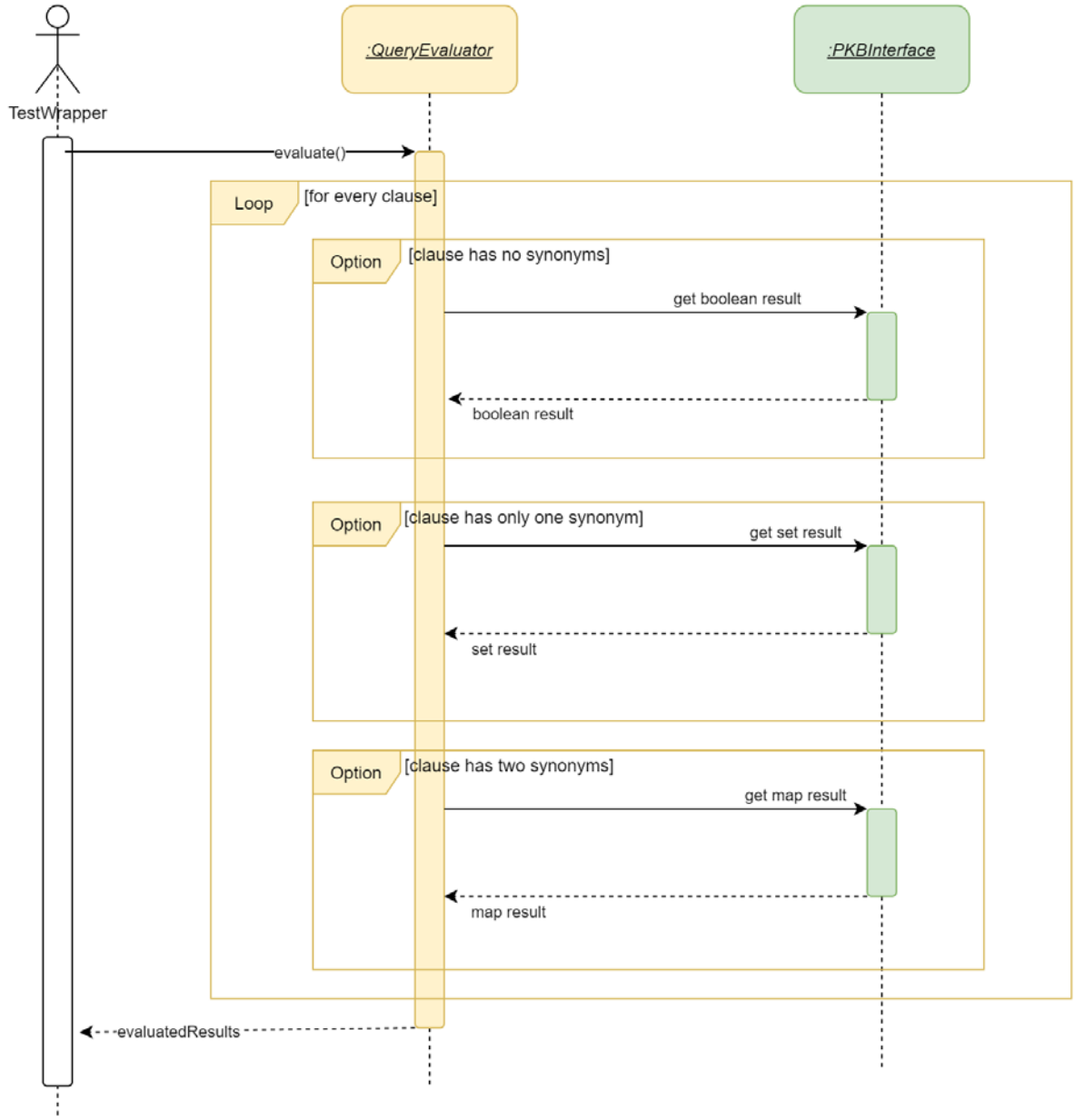


Figure 4.2.1.1: High level interaction between Query Processor and PKB

Figure 4.2.1.1 shows the condensed high-level interaction between the Query Processor and the PKB. There are 3 different types of interactions depending on the number of synonyms in the clause that the Query Evaluator is evaluating. If the clause has no synonyms, the Query Evaluator will retrieve a boolean result, if the clause has one synonym, the Query Evaluator will retrieve a set of result values and if the clause has two synonyms, the Query Evaluator will retrieve a map of result values. Therefore, our PKB-Query Processor integration tests are designed to test these different types of interactions.

4.2.2 Source Parser-PKB Examples

	Test Purpose	Required Test Inputs	Expected Test Results
Test relationship insertion	Verify that the design extractor successfully inserts all extracted relationships into the PKB	The simple program involving multiple procedures used in PKB unit testing (4.1.1)	Retrieved results contains values equivalent to that of manual calculation
Test pattern insertion	Verify that the DE successfully inserts all patterns into the PKB	The simple program involving similar assign and container statements	Retrieved results contains values equivalent to that of manual calculation

4.2.3 PKB-PQL Processor Examples

	Test Purpose	Required Test Inputs	Expected Test Results
Test interaction for clauses with 1 synonym only	Verify that the Query Processor successfully retrieves **set** results from the PKB when the clause has 1 synonym only	Query Evaluator is initialized with: 1. Query object that represents the syntactically valid query: `stmt s; Select s such that Follows*(1, s)` 2. The actual PKB object containing manually inserted information.	ResultsTable object that contains values equivalent to the values initialized in the PKB.
Test interaction for clauses with 2 synonyms	Verify that the Query Processor successfully retrieves **map** results from the PKB when the clause has 1 synonym only	Query Evaluator is initialized with: 1. Query object that represents the syntactically valid query: `variable v; assign a; Select v pattern a(v, _"x+1"_)` 2. The actual PKB object containing manually inserted information.	ResultsTable object that contains values equivalent to the values initialized in the PKB.

4.3 System Testing

4.3.1 SIMPLE Source Code

Before writing a SIMPLE source code, there are 2 main aspects which are taken into consideration. The aspects are the followings:

- The aim of the source code - Each source code is designed to tackle an aspect of SPA requirement, allowing the system test to proceed in a manner so that the tester can zoom in to a certain area for debugging purposes and not look through the whole code.
- The challenges provided by the source code - While each source code serves their own purposes like testing for conditional statements, nested statements etc. Each source code also needs to include complexity to test the accuracy of the parser and design extractor.

For Iteration 1, 4 SIMPLE source codes are designed with different difficulty to tackle each aspect of SPA requirement and test the SPA program in a well-rounded format. With different difficulty, the tester is able to test in a systematic order and not go straight to the hardest test cases. Ordering from the simplest to the hardest, the SIMPLE source codes are as followed:

- simpleTest - a basic SIMPLE source code which only contains 4 assign statements within the procedure. The purpose of simpleTest is to ensure that all basic functions of our SPA program are able to work correctly without taking into consideration the more complex requirement for iteration 1 like loops and conditional statements.

```
procedure simpleTest {
1.  a = 1;
2.  b = a + 1;
3.  c = a + b;
4.  d = c;
}
```

figure 4.3.1.1 source code for simpleTest

- conditionalTest - building upon simpleTest, conditionalTest taking into account of if and while statements. With conditional statements within the procedure, this source code will aim to challenge the Parser and Design Extractor to test if relationships between statements are correctly perceived by the program. Within the conditionalTest, there exist at most a single level of conditional statement, thus, the program had yet to need to worry about Parent* relationship.
- nestedTest - similar to conditionalTest, nestedTest proceeds with testing our SPA program with conditional clauses with the difference being that within nestedTest, there exists different levels of nesting within the procedure. This difference in nesting level proceeds to further test

if our Parser and Design Extractor is able to accurately parse and understand the relations between the statement list. Excluding the nested level of conditional statements, nestedTest aims to test the SPA program in a similar manner as conditionalTest.

```
procedure nestedTest {
   1. a = 1;
   2. b = a + 1;
   3. c = a + b;
   4. while(a > 0) {
   5. a = a - 1;
   6. b = a + c;
   7. if (b > c) then {
   8. d = c;
      } else {
   9. e = b;
      }
      }
   10.a = a - b * c;
   11.if ((e < 10) && (a > e)) then {
   12.e = e + 1;
   13.a = a - e;
      } else {
   14.e = e - 1;
   15.c = c + e;
   16.while ((d <= 20) || (d < (a + b))) {
   17.d = d + a * (c - e);
            }
   18.d = d - 1;
      }
   19.c = c / a + (b - d);
}
```

figure 4.3.1.2 source code for nestedTest

- complexTest - With the first 3 SIMPLE source codes, the SPA program had already been cleared of all the basic requirements stated in Iteration 1, thus, the need arose to test the program with even more difficulty. Within complexTest, there exist multiple levels of nesting aiming to test the capability of our Parser. In addition to the multiple levels of nesting, complexTest also increases the complexity of expression and conditional statements. Though the complexity of expression and conditional statement had yet to affect much due to the requirement of iteration 1, it will be useful in future iteration.

For Iteration 2, 3 SIMPLE source codes are designed with different purposes to tackle the new extensions made in iteration 2. By designing SIMPLE source code with different aims, it allows the tester to zoom into particular extensions and aim to uncover possible edge cases. The SIMPLE source code are as followed:

- patternTest - A SIMPLE source code containing 3 procedures, each only containing assign, while and if statements. For while and if statements, the conditional statements are not required to be complex as conditional pattern checking only focuses on the presence of variables. Thus, the main focus will be the full expression matching of the assign statements. Instead of writing many different expressions, each assign statement consists of the same expression with only the difference being the placement of brackets within the expression. This allows the tester to test if the parser is able to correctly extract the exact subexpression from the procedure.

```
procedure assignment {
    1.  a = a * b - c / d + e - f * g % h;
    2.  b = a * (b - c) / d + (e - f) * g % h;
    3.  c = a * ((b - c) / d + (e - f)) * g % h;
    4.  d = a * (b - c / d) + (e - f * g) % h;
    5.  e = (a * b - c) / (d + e) - f * (g % h);
}
```

```
procedure  ifpattern {
    6.  if ((a >= 0) || (b < a)) then {
    7.  read a;
        } else {
    8.  read b;
        }

    9.  if ((a >= 0) || ((b < a) && (c == d))) then {
    10. read a;
        } else {
    11. read b;
        }
```

```
procudure whilepattern {
  21. while (x > 0) {
  22. print z;
        }

  23. while ((x>0) && (a > b)){
  24. print z;
        }
```

figure 4.3.1.3 snapshot of patternTest

- callTest - Contains multiple procedures, with each procedure only containing call statements or nothing at all. This source code is designed to test if the parser is able to accurately perceive the relations between different procedures. Having certain procedure calling either 1 or more procedure, it form a “network” which allows the tester to fully test the correctness of Call/* relationship
- extendTest - Similar to the source code designed in iteration 1, extendTest aims to test different relationships focussing on modifies, uses and next relationship. By containing multiple procedures, it allows the tester to test if the parser is able to extract accurate information regarding modifies and uses relationships. In addition, with conditional

statements, different CFG are formed to allow the tester to test the correctness of the next relationship.

For Iteration 3, 1 SIMPLE source code is designed for the purpose of tackling the new features made in iteration 3. With a single complex SIMPLE source code, all new extensions will be able to be tested in a rigorous manner. The SIMPLE source code is :

- complexTest - A SIMPLE source code containing multiple procedures with each procedure containing different levels of nesting. In addition, each procedure has a different number of lines with one procedure containing only print statements while other procedures containing complexed expressions. This source code is designed to contain the “highest” level of difficulty to test the SPA program, in order to ensure that the SPA program is working with a minimal number of bugs.
- affectsbipTest - A SIMPLE source code containing multiple procedures. The purpose of this source code is to test the extension AffectsBip and AffectsBip* relationship. As according to the information provided, there is a single call assumption for the extension, AffectsBip/* is unable to be tested under complexTest. Containing multiple procedures with just assign statement, affectsbipTest is designed to have a simplified code to test the accuracy of AffectsBip/*.

4.3.2 PQL queries

With the design of SIMPLE source code done, the only remaining part left before the start of the system test is the design of test cases for PQL queries. In an attempt to encompass as many edge cases as possible, several considerations are made while writing the queries.

- Possible design entities for a certain relationship
    - e.g. For Follows relationship, possible design entities are stmt, assign, while, if, read, print. Any other design entities included in the queries will result in a Semantic error
- Possible combination of design entity
    - Pairing different combinations of design entities like stmt and assign to maximise the coverage of the test cases. e.g. Follows(s, a) and Follows(a, s) may results in different output
- Possible edge cases
    - e.g. Based on SIMPLE source code, the queries may aim to target the certain area within the source code like a nested statement list, to test if PKB contains the correct information
- Possible error
    - Aim to cover all possible syntax and semantic error

Based on the consideration stated, during iteration 1, each of the 4 source code written is accompanied by a list of PQL queries to test if the correct output is provided. The list includes:

- Follows and FollowStar
    - Test direct Follows and indirect Follows* relationship, with different combination of synonym including stmt s; assign a; while w; if ifs;
    - Input are designed in the following manners:
        - Possible location for single synonyms
            - Follows/Follows*(s / a / w / ifs, _)
            - Follows/Follows*(_, s / a / w / ifs)
            - Follows/Follows*(s / a / w / ifs, 3)
            - Follows/Follows*(2, s / a / w / ifs)
        - Possible combination of different synonyms
            - Follows(s / a / w / ifs, s / a / w / ifs)
        - To note, available synonyms are adapted based on the source code the file is attached to. For example, in simpleTest, there will not be the option of w and ifs as there are no conditional statements within the source code
    - Test for incorrect input
        - Syntax error - e.g. stmt s; Select s such that Follows(s, “S”)
        - Semantic error - e.g. variable v; select v such that Follows(v, _)
    - Available in all 4 Simple source code

```
2 - comment
stmt s;
Select s such that Follows*(s, 4)
1, 2, 3
5000
```

figure 4.3.2.1 sample query for FollowStar

- Parent and ParentStar
    - Test direct Parent and indirect Parent* relationship, with different combination of synonym including stmt s; assign a; while w; if ifs;
    - Input are designed in the following manners:
        - Possible location for single synonyms
            - Parent/Parent*(s / w / ifs, _)
            - Parent/Parent*(_, s / a / w / ifs)
            - Parent/Parent*(s / w / ifs, 3)
            - Parent/Parent*(2, s / a / w / ifs)

        - Possible combination of different synonyms
            - Parent(s / w / ifs, s / a / w / ifs)
        - For Parent and ParentStar, assign a is not available on the Left Hand Side(LHS) of the query as there is no assign statements that are parent of any type statement
    - Test for incorrect input
        - Syntax error - stmt s; Select s such that Parent(s, “1”)
        - Semantic Error - assign a; select a such that Parent(a, _)
    - Available in all 4 SIMPLE source code except for simpleTest which only have invalid cases
- Uses
    - Test Uses relationship, with different combination of synonym including stmt s; assign a; variable v; while w; if ifs;
    - Input are designed in the following manners:
        - Possible location for single synonyms
            - Uses(s / a / w / ifs, _)
            - Uses(s / a / w / ifs, “a”)
        - Possible combination of different synonyms
            - Uses(s / a / w / ifs, v)
        - For Uses relationships, the Right Hand Side can only contain either “variable name” or the variable synonym while the LHS can only contain different types of statements like stmt and assign.
    - Test for incorrect input
        - Syntax error - stmt s; Select s such that Uses(s, 3)
        - Semantic error - variable v; Select v such that Uses(_, v)
    - Available in all 4 SIMPLE source code

```
3 - comment
stmt s;
Select s such that Uses(s, "S")
none
5000
```

figure 4.3.2.2 sample query for Uses

- Modifies
    - Test Modifies relationship, with different combination of synonym including stmt s; assign a; variable v; while w; if ifs;

- Input are designed in the following manners:
  - Possible location for single synonyms
    - Modifies(s / a / w / ifs, _)
    - Modifies (s / a / w / ifs, "a")
  - Possible combination of different synonyms
    - Modifies (s / a / w / ifs, v)
  - Similar to Uses relationship, the RHS of Modifies relationship can only contain either "variable name" or the variable synonym while the LHS can only contain different types of statements like stmt and assign.
- Test for incorrect input
  - Syntax error - stmt s; Select s such that Modifies(s, 3)
  - Semantic error - variable v; Select v such that Modifies(_, v)
- Available in all 4 SIMPLE source code

- Pattern
  - Test for pattern clauses, with different combination of synonym including assign a; variable v;
  - Input are designed in the following manners:
    - Possible location for single synonyms
      - pattern a(_, _)
      - pattern a(v, _)
      - pattern a(v, _"x"_)
    - For pattern clauses, there are limited test cases as Iteration 1 only test the basic of pattern matching without full expression matching
  - Test for incorrect input
    - Syntax error - assign a; Select a pattern a(_, "a")
    - Semantic error - assign a; constant c; select a pattern a(c, _)
  - Available in all 4 SIMPLE source code
- 2_Clauses
  - Test for RS + Pattern clauses with different combination of synonym including stmt s; assign a; variable v; while w; if ifs;
  - Input are designed in the following manners:
    - no common synonym
      - Follows(s, _) pattern a(v, _)
      - Follows(s, a1) pattern a(v, _"A"_)

            - etc.
        - 1 common synonym
            - Follows(a, _) pattern a(v, _)
            - etc
    - Available in all 4 SIMPLE source code

In Iteration 2, with the SIMPLE source code designed with different purposes, PQL queries are also designed more towards a particular source code, resulting in some source code only testing a certain relationship. The test cases are as follows:

- Call and CallStar
    - Test for direct Calls and indirect Calls* relationship with different placement of procedure p;
    - Inputs are designed in the following manner:
        - Calls / Calls*(p, _)
        - Calls / Calls*(_, p)
        - Calls / Calls*("procedure", p)
        - Calls / Calls*(p, "procedure")
    - Test for invalid Inputs
        - Syntax error - procedure p; Select p such that Calls(p, 1)
        - Semantic error - while w; Select w such that Calls*(w, _)
    - Available for callTest and extendTest
- Modifies
    - Test Modifies relationship, with different combination of synonym including stmt s; assign a; variable v; while w; if ifs; procedure p;
    - Input are designed in the following manners:
        - Possible location for single synonyms
            - Modifies(s / a / w / ifs / p, _)
            - Modifies (s / a / w / ifs / p, "a")
        - Possible combination of different synonyms
            - Modifies (s / a / w / ifs / p, v)
        - The difference between iteration 1 and 2 is the existence of multiple procedures, with LHS able to contains procedure synonym and variable v in RHS being affected by call statements
    - Test for incorrect input

        - Syntax error - stmt s; Select s such that Modifies(s, 3)
        - Semantic error - variable v; Select v such that Modifies(_, v)
    - Available in extendTest
- Uses
    - Test Uses relationship, with different combination of synonym including stmt s; assign a; variable v; while w; if ifs; procedure p;
    - Input are designed in the following manners:
        - Possible location for single synonyms
            - Uses(s / a / w / ifs / p, _)
            - Uses(s / a / w / ifs / p, “a”)
        - Possible combination of different synonyms
            - Uses(s / a / w / ifs / p, v)
        - Similar to Modifes, the difference between iteration 1 and 2 is the existence of multiple procedures, with LHS able to contains procedure synonym and variable v in RHS being affected by call statements
    - Test for incorrect input
        - Syntax error - stmt s; Select s such that Uses(s, 3)
        - Semantic error - variable v; Select v such that Uses(_, v)
    - Available in extendTest
- Pattern
    - Test for pattern clauses, with different combination of synonym including assign a; variable v; while w; if ifs;
    - Input are designed in the following manners:
        - Possible location for single synonyms
            - pattern a / w(_, _)
            - pattern ifs(_, _, _)
            - pattern a / w(v, _)
            - pattern ifs(v, _, _)
            - pattern a(v, _”x”_)
            - pattern a(v, _”x + 1”_)
            - pattern w(“a”, _)
            - pattern ifs(“b”, _, _)
        - The difference between iteration 1 and 2 is that pattern assign is now able to check for full expression instead of just the presence of a variable. In addition,

pattern if and pattern while is able to check if a variable exist within the conditional statement.

- Test for incorrect input
  - Syntax error - assign a; Select a pattern a(_, “a”)
  - Semantic error - assign a; constant c; select a pattern a(c, _)
- Available in patternTest.

- next and nextStar
  - Test for direct Next and indirect Next* relationship with different combination of synonym including stmt s; assign a; while w; if ifs; prog_line n;
  - Inputs are designed in the following manner:
    - Next / Next *(s / a / w / ifs / n/ _, s / a / w / ifs / n / _)
    - Next / Next *(3, s / a / w / ifs / n)
    - Next / Next *(s / a / w / ifs / n, “3)
  - Test for invalid Inputs
    - Syntax error - prog_line n; Select n such that Next(n, “a”)
    - Semantic error - prog_line n; Select n such that Next(3, 2)
  - Available for extendTest

- multipleRS
  - Test for multiple clauses with different levels of common attributes
  - Inputs are designed in the following manners:
    - 0 common attributes
      - Follows(s, _) and Parent(_, a)
      - Uses(s, v) and Modifies(w, _)
    - 1 common attributes
      - Follows(s, _) and Parent(s, a)
      - Calls(p, _) and Modifies(p, "u")
    - 2 common attributes
      - Uses(a, v) and Modifies(a, v)
    - Different compared to Select
      - Select a such that Uses(s, v) and Parent(s, _)
      - Select p such that Parent(ifs, w) and Uses(ifs, "a")
  - Available in extendTest

In Iteration 3, with the SIMPLE source code designed with a high difficulty level, PQL queries are designed solely towards the particular source code, resulting in test cases only containing several queries. The test cases are as follows:

- affects and affectStar
    - Test for direct Affects and indirect Affects* relationship with different combination of synonym including stmt s; assign a;
    - Inputs are designed in the following manner:
        - Next / Next *(s / a / _, s / a / _)
        - Next / Next *(3, s / a)
        - Next / Next *(s / a , “3)
    - Test for invalid Inputs
        - Syntax error - stmt s; Select s such that Affects(s, “a”)
        - Semantic error - print p; Select p such that Next(p, 2)
    - Available for complexTest
- with
    - Test for with clause with different combination of synonym and attributes including procedure p; call c; variable v; read r; print pr; constant c1; stmt s; while w; if ifs; assign a;
    - Possible attributes are as followed:
        - p.procName
        - c.procName / c.stmt#
        - v.varName
        - r.varName / r.stmt#
        - pr.varName / pr.stmt#
        - c1.value
        - s.stmt#
        - w.stmt#
        - ifs.stmt#
        - a.stmt#
    - inputs are designed in the following manner:
        - with [possible attributes] = [possible attributes]
        - with [possible attributes] = 3 / “a”
    - Test for invalid inputs:
        - Syntax error - stmt s; Select s with s.procName

        - Semantic Error - stmt s; Select s with s.stmt# = “a”
    - Available for ComplexTest
- NextBip and NextBipStar
    - Test for direct NextBip and Indirect NextBip* relation with different combination of synonym including stmt s; assign a; while w; if ifs; call c; print p; read r;
    - Inputs are designed in the following manner:
        - NextBip / NextBip* (s / a / w / ifs / c / p / r / _, s / a / w / ifs / c / p / r / _)
        - NextBip / NextBip* (s / a / w / ifs / c / p / r / _, 2)
        - NextBip / NextBip* (7, s / a / w / ifs / c / p / r / _)
    - Test for invalid inputs:
        - Syntax Error - stmt s; Select s such that NextBip(s, a)
        - Semantic Error - variable v; Select v such that NextBip(v, _)
    - Available for complexTest
- AffectsBip and AffectsBipStar
    - Test for direct AffectsBip and Indirect AffectsBip* relation with different combination of synonym including stmt s; assign a;
    - input are designed in the following manner:
        - AffectsBip / AffectsBip* (s / a, s / a / _)
        - AffectsBip / AffectsBip* (s / a, 3)
        - AffectsBip / AffectsBip* (2, s / a / _)
    - Test for invalid inputs:
        - syntax error - stmt s; Select s such that AffectsBip(s, “2”)
        - Semantic Error - while w; Select w such that AffectsBip(w, _)
    - Available for affectsbipTest

Each grouping of test cases is further split into Valid and Invalid indicating whether the query will provide an output or not.

In order to have a more efficient time testing the program with the test cases, PQL queries are written into autotester format, allowing the tester to have an easier time testing the written queries. The queries are written according to the format provided in the github link.

Based on the xml file, the tester is then able to zoom into the particular test case which generates a wrong output. Based on queries, debugging is commenced aiming to find the source of the bug. Debugging is mainly done through running the same queries again through direct inputs similar to Unit/Integration Testing. Once the component that held the fault is discovered, the bug is fixed by the coder in charge of the component and the same queries are executed again to ensure that there are no additional bugs.

5 Extension: NextBip and AffectBip

5.1 Overview

Our team choose to implement ***NextBip***/***NextBip****, ***AffectBip***/***AffectBip**** with the assumption that each procedure can only be called once as building CFG in the general case is proved to be difficult. Hence, we will not go into detail about the definition of these extensions as it is available on the wifi page. Instead, we will focus on the implementation details of both extensions.

5.1.1 NextBip, NextBip*

For the implementation, NextBip(*) relationship is mostly calculated in the same way as Next relationship, except the one which involves a call statement.

To handle that, for each procedure p, we need to find all the statements which can be the end statement of that procedure, called as E(p). This is already handled in the Next relationship, which we reused. Then we create a dummy branch out node for each procedure, and then connect those statements in E(p) to the dummy node.

Then for each call statement ***stmt*** of procedure p into procedure p', we can get the next statement **stmt'** and add NextBip(x, stmt') for each x in E(p').

NextBip* can then be calculated as all other transitive closure relationships.

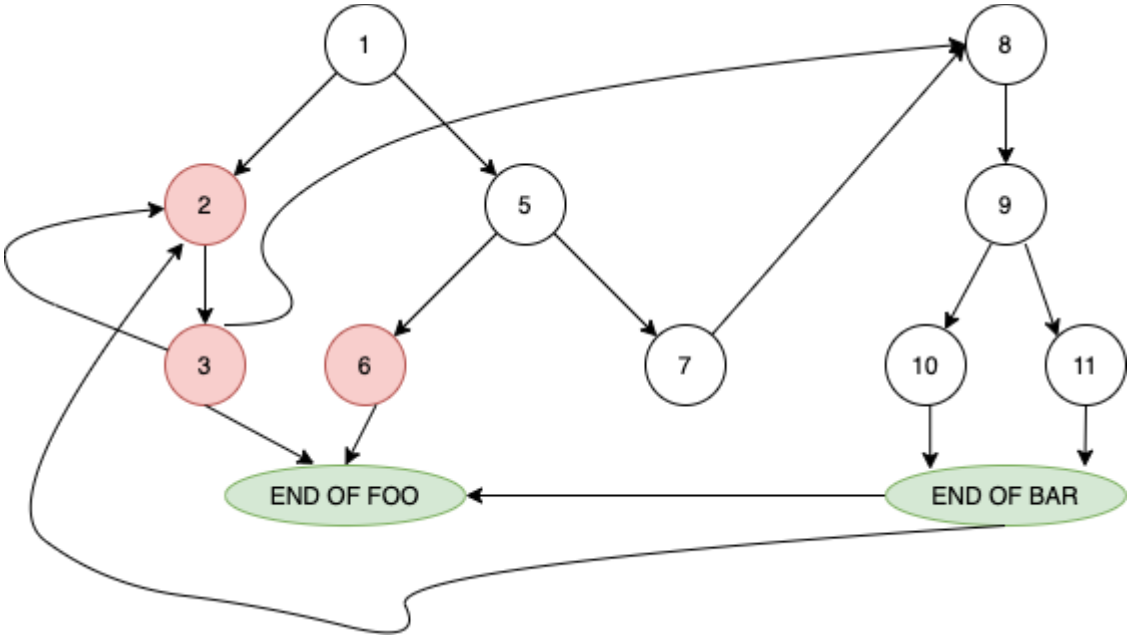
Take an example

```
procedure Foo {
1.    if (x == 1) then {
2.         while (y == 2) {
3.            call Bar;
           }
      } else {
5.         if (p == 2) then {
6.              x = y;
           } else {
7.              call Bar;
           }
      }
}

procedure Bar {
8.    x = z + 2;
9.    if (x == 1) then {
10.        y = 1;
      } else {
11.        y = 2;
      }
}
```

We have E("Foo") = { 2, 6, 7} and E("Bar") = {10, 11}. The figure below represents the resulting CFG BIP:

1. There is an edge from 3 to 8 as statement 3 calls procedure Bar.
2. There is an edge from "END OF BAR" to 2, as after returning the call from statement 3, the control continues at statement 2.



5.1.2 AffectBip, AffectBip*

Given that as the relationship AffectBip can only involve 2 assign statements, we use Breadth-First Search (BFS).

Starting from an assign statement, we can deduce the modified variable V. Then we follow the CFG graph if and only if the target statement does not directly modify the variable. By that, for each visited statement, we can add the new AffectBip relationship between the starting statement of the BFS process to the current statement if the current statement uses variable V.

For examples, we consider

```
procedure Foo {
1.    read x;
2.    count = 0;

3.    while (count > 0) {
4.       y = x % 10;
5.       call Bar;
6.       count = count - 1;
     }

7.   print y;
}

procedure Bar {
8.    x = z + 2;
}
```

Only AffectBip(8, 4) is true.

The resulting CFG for this code is as follow.

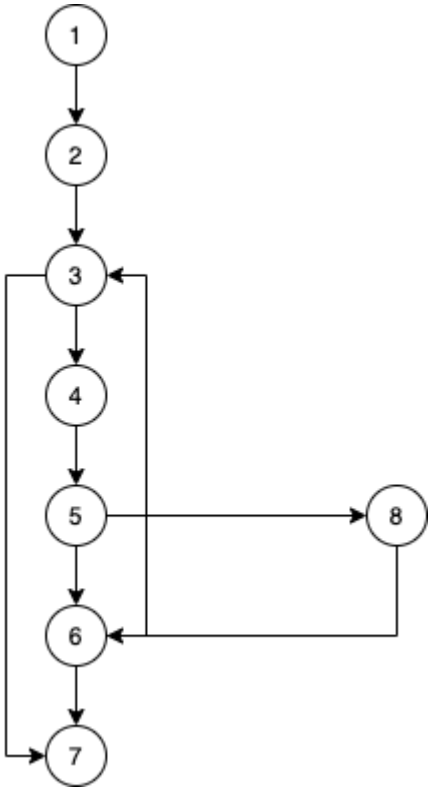


Figure 5.1.2.1 The ***NextBip*** graph where ***a*** has an edge to ***b*** if ***NextBip(a,b).***

Starting from 8, we deduce that statement 8 modifies variable x. Then the path taken by the BFS will be 8 -> 6 -> 3 -> 4. When it reaches 4, as statement 4 is an assignment, the BFS will check if statement 4 uses variable x, which it does, therefore AffectBip(4, 8) is added.

Using the assumption, we can deduce that AffectBip* can be defined as the transitive closure of AffectBip and therefore can be handled as others transitive relationships.

5.2 Challenges

The most difficult challenge we faced was to correctly connect inter-procedural statements for NextBip. At the start, we thought that each statement x that has Next(x, y) = false for all y must be the end statement of some procedures. However, that is incorrect.

Consider this example:

```
procedure Foo {
1.    read x;
2.    count = 0;

3.    while (count > 0) {
4.       z = y;
5.       y = x;
6.       x = u + 2;
7.       count = count - 1;
     }
```

```
}
```

Statement 3 is the only possible end statement of procedure Foo, however Next(3, 4) is true. After some more failed attempts, we realised that we can modify the code that calculates the Next relationship to return the end statement of each procedure. Using that, we only need to make some modification to Next relationship to convert it to NextBip

5.3 Affected Components

5.3.1 Source Processor

This new extension mostly affects Source Processor and particularly the Design Extractor. It needs to

1. Extract ***NextBip*** relationship from basic relationships such as ***Next, Parent, Follow, Call, …***
2. Extract ***AffectBip*** relationships using BFS.

5.3.2 PKB

NextBip(stmt1, stmt2) is of a similar structure as Next(stmt1, stmt2), whereas AffectBip(stmt1, stmt2) closely resembles the Affects relationship. The extension for PKB is simply:

1) add new relationship types: **NextBip**, **NextBip***, **AffectBip, AffectBip***
2) add insertion methods used by the design abstractor. All of these new relationships will be stored in the current relationship tables by increasing the size of arrays, namely relations[], relationsBy, relationKeys[] and relationByKeys[].
    a) `Boolean insertNextBip(STMT NO, STMT NO)`
    b) `Boolean insertNextBipStar(STMT NO, STMT NO)`
    c) `Boolean insertAffectBip(STMT NO, STMT NO)`
    d) `Boolean insertAffectBipStar(STMT NO, STMT NO)`
3) No change for retrieval, as these relationships are treated uniformly in getResultsOfRS().

5.3.3 Query Processor

Query Preprocessor

For the Tokenizer, 4 new token types will have to be added, **NextBip, AffectsBip, NextBip* and AffectsBip*.**

For the Program Query Language, these are the new grammar rules:

1) **NextBip ➔ 'NextBip' '(' stmtRef ',' stmtRef ')'**
2) **NextBipT ➔ 'NextBip*' '(' stmtRef ',' stmtRef ')'**
3) **AffectsBip ➔ 'AffectsBip' '(' stmtRef ',' stmtRef ')'**
4) **AffectsBipT ➔ 'AffectsBip*' '(' stmtRef ',' stmtRef ')'**

The **relRef** function will now have to possibly make calls to 2 new functions mirroring the new non-terminals added, **nextBip** and **AffectsBip.** (Only 2 new functions are created because **NextBip** and **NextBip*** will be handled under the same function, similarly for **AffectsBip** and **AffectsBip***). Besides this change, the hash tables containing allowed entity types of synonyms as left and right-hand arguments to either a stmtRef or entRef will be updated to include **NextBip, NextBip*,** **AffectsBip** and **AffectsBip*.**

Design Entity type (Left/ Right argument)	Stmt	Assign	If	While	Print	Read	Constant	Variable	Call	Procedure	Program Line
AffectsBip / AffectsBip*	✔/✔	✔/✔	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✔/✔
NextBip / NextBip*	✔/✔	✔/✔	✔/✔	✔/✔	✔/✔	✔/✔	✗/✗	✗/✗	✔/✔	✗/✗	✔/✔

The structure of **nextBip** and **affectsBip** closely mirrors that of the other functions representing relationship references and the only change will be to pass the correct set of allowed entity types from the table above as a parameter into a stmtRef call.

When populating the **Query** object, the only change that is required is to add ***NextBip*** and ***AffectsBip*** relationships to the **`RELATION_TYPE`** enum.

Example Query object population	
NextBip / NextBip*	**Example:** prog_line pgl1; prog_line pgl2; Select pgl1 such that Follows(pgl1, pgl2) and NextBip(pgl1, pgl2) For this example, each of the relationships will be represented by a RelationshipClause object. (See Section 3.3.2 for all Query-related classes) A RelationshipClause contains a relationshipType enum, as well as a left and a right QueryInput. The NextBip RelationshipClause object created here will contain a relationshipType of **NextBip,** and hold **Declaration** objects as its left and right QueryInputs with values “pgl1” and “pgl2”. The only change is the relationType enum that has to be added. **Example:** prog_line pgl1; prog_line pgl2; Select pgl1 such that Follows(pgl1, pgl2) and NextBip(1, 2) In this case, the left and right QueryInputs will be **StmtNum** objects which were already used previously (for example in **Next/Next***)
AffectsBip / AffectsBip*	**Example:** assign a1; assign a2; procedure p; variable v; Select a1 such that AffectsBip(a1, a2) and Uses(p, v) and Modifies(a1, v) The AffectsBip RelationshipClause object created here will contain a relationshipType of **AffectsBip,** and hold **Declaration** objects as its left and right QueryInputs with values “a1” and “a2”. Likewise for **AffectsBip,** the only real change is the addition of its corresponding relationship type as an enum since all of the possible input arguments **(StmtNum, Ident, Declaration)** are well-defined for previous relationships.

Query Evaluator/Results Projector

Since ***NextBip(*)*** and ***AffectsBip(*)*** are binary relationships, the following functions in the existing abstract PKB API will be called by the Query Evaluator to retrieve results of **such that** clauses with ***NextBip(*)*** and ***AffectsBip(*)*** relationships:

- `LIST RES getSetResultOfRS(RELATION TYPE, QUERY INPUT)`
- `LIST OF LIST RES  getMapResultsOfRS(RELATION TYPE, QUERY INPUT)`

The query representation classes, query evaluation code and the PKB API used by the Query Evaluator were designed to adhere to the open-close principle, therefore as ***NextBip(*)*** and ***AffectsBip(*)*** are of similar binary input format as the existing relationships, this extension would not result in any changes in the Query Evaluator and Results Projector.

As mentioned in the above Query Preprocessor section, only new relationship type enums will have to be added in order to populate the **Query** object the Query Evaluator receives.

5.4 System Test

5.4.1 SIMPLE source code

For testing our extension, SIMPLE source code designed in iteration 3 are used. With the source code, the tester will be able to test the functionality of the extensions.

The designed SIMPLE source code can be designed in the following way:

- complexTest - A SIMPLE source code containing multiple procedures with each procedure containing different levels of nesting. In addition, each procedure has a different number of lines with one procedure containing only print statements while other procedures containing complexed expressions. This source code is designed to contain the "highest" level of difficulty to test the SPA program, in order to ensure that the SPA program is working with a minimal number of bugs.
- affectsbipTest - A SIMPLE source code containing multiple procedures. The purpose of this source code is to test the extension AffectsBip and AffectsBip* relationship. As according to the information provided, there is a single call assumption for the extension, AffectsBip/* is unable to be tested under complexTest. Containing multiple procedures with just assign statement, affectsbipTest is designed to have a simplified code to test the accuracy of AffectsBip/*.

5.4.2 PQL Queries

For the possible extension, there are only 2 sets of relationships that the tester is required to focus on - ***NextBip/NextBip**** and ***AffectsBip/AffectsBip****. Taking into account that all other relationships are already tested in iteration 2, the test for extension is only required to focus on the new functionality of our program. The test cases are as follows:

- **NextBip** and **NextBipStar**
    - Test for direct **NextBip** and Indirect **NextBip*** relation with different combination of synonym including stmt s; assign a; while w; if ifs; call c; print p; read r;

- Inputs are designed in the following manner:
  - **NextBip / NextBip*** (s / a / w / ifs / c / p / r / _, s / a / w / ifs / c / p / r / _)
  - **NextBip / NextBip*** (s / a / w / ifs / c / p / r / _, 2)
  - **NextBip / NextBip*** (7, s / a / w / ifs / c / p / r / _)
- Test for invalid inputs:
  - Syntax Error - stmt s; Select s such that NextBip(s, a)
  - Semantic Error - variable v; Select v such that NextBip(v, _)
- Available for complexTest

- **AffectsBip** and **AffectsBipStar**
  - Test for direct **AffectsBip** and Indirect **AffectsBip*** relation with different combination of synonym including stmt s; assign a;
  - input are designed in the following manner:
    - **AffectsBip / AffectsBip*** (s / a, s / a / _)
    - **AffectsBip / AffectsBip*** (s / a, 3)
    - **AffectsBip / AffectsBip*** (2, s / a / _)
  - Test for invalid inputs:
    - syntax error - stmt s; Select s such that AffectsBip(s, “2”)
    - Semantic Error - while w; Select w such that AffectsBip(w, _)
  - Available for affectsbipTest

5.5 Conclusion

The ***NextBip/NextBip**** and ***AffectBip/AffectBip**** that we implemented have a limitation, as we assume that each procedure can only called once. Therefore, special test cases need to be created as no other relationships have this assumption.

Based on previous sections, we believe that we have researched the extension requirement sufficiently, have a suitable schedule and extensive testing to ensure that our extension is well-tested.

6 Coding & Documentation Standards

6.1 Coding Standards

In general, we mostly refer to Google coding standard for C++:

https://google.github.io/styleguide/cppguide.html

However, there are some standards that we modify for this project.

6.1.1 General

- Each line should have a maximum of 140 characters

6.1.2 Curly Braces Indentation

- Open curly brace of function declaration is on the same line, not the start of the next line
- There should be a space between a close round bracket and an open curly brace,
- Close curly brace should always be on the last line. The exception is when the statement list is empty, then it can be on the same line as the open brace.

6.1.3 Headers and Include

- There should be 2 groups included in any files: first group for standard libraries, second group for project libraries.
- These 2 groups should be separated by 1 empty line

6.1.4 Naming conventions

We only define conventions for some standard types, the remaining are up to each member judgement to decide

Types	**Convention**
Variables	Camel case (ex: numOfStatement)
Functions	Camel case, must be a verb (ex: lookAheadSingle)
Class names	PascalCase, must be a noun (ex: QueryParser)
Enumerations, Constants, etc	All upper case (ex: ASSIGN)

6.1.5 Comments

Comments generally show the purpose of a public function or implicit assumptions used.

- Single line comments should start with // and a space after that
- Multiple line comments should be between /* and */ pair.

6.2 Documentation Standards

6.2.1 Structure

In general, major section is arrange in this order:

- Overview (if needed)
- Design
- Implementation
- Examples

6.2.2 Diagrams and Table

We use Draw.io platform to draw all of our diagrams. Components are in a consistent color throughout the document.

- **The Source Processor** is in blue.
- **The Program Knowledge Base** is in green.
- **PQL** is in yellow.

6.2.3 Documentation standards for Abstract APIs

We describe the abstract APIs using the following format:

- Parameter and return types are in bold letters.
- Types are described using words with underscores ("_") in between.
- Method names are in Camel case (consistent with our naming convention).
- Method names are the same as those in concrete APIs.

Below lists the mapping between the abstract class and the actual type used:

Abstract Type	Concrete Type	Abstract Type	Concrete Type
VAR_NAME	string	**PROC_NAME**	string
CONST_VALUE	string	**NAME**	string
STMT_NO	int	**STMT_TYPE**	EntityType
ENTITY_TYPE	EntityType	**RELATION_TYPE**	RelationshipType
EXPRESSION	Expression	**QUERY_INPUT**	shared_ptr<QueryInput>
SYNONYM	shared_ptr<Declaration>	**ATTR**	EntityType
LIST_RES	unordered_set<string>	**LIST_OF_LIST_RES**	unordered_map<string, unordered_set<string>>

7 Discussion

7.1 Cross platform

We faced several issues due to cross platform development. To illustrate more, 3 of our team members are using Windows operating systems with Visual Studio setup, one of us is using Linux and one of us is using Mac. MacOS uses the Clang compiler where a few libraries are not supported compared to Visual Studio libraries. Hence, at some point in time, our codes just do not compile.

7.2 Online meetings

The nature of online meetings are just not efficient compared to offline with white board meetings. Sometimes we have ideas but cannot be shared efficiently using words/oral presentations. Poor/interrupted internet connections occur sometimes.

8 Appendix

8.1 PKB Abstract API

Below is the abstract API used by the source parser:

- **Boolean** `insertVariable(`**VAR_NAME**`)`
  Insert a variable by its name. Returns true if the operation is successful.
- **Boolean** `insertConstant(`**CONST_VALUE**`)`
- **Boolean** `insertProcedure(`**PROC_NAME**`)`
- **Boolean** `setStatementType(`**STMT_NO, STMT_TYPE**`)`
  Set a statement as an entity type, where the statement is identified by its index. Returns true if the operation is successful. This method modifies `types` and `entities`.
- **Boolean** `insertParent(`**STMT_NO, STMT_NO**`)`
  Insert a *parent* relationship between two statements, each of which is identified by its index. Other insertion methods are similarly defined. Returns true if the operation is successful.
- **Boolean** `insertParentStar(`**STMT_NO, STMT_NO**`)`
- **Boolean** `insertFollows(`**STMT_NO, STMT_NO**`)`
- **Boolean** `insertFollowsStar(`**STMT_NO, STMT_NO**`)`
- **Boolean** `insertNext(`**STMT_NO, STMT_NO**`)`
- **Boolean** `insertNextStar(`**STMT_NO, STMT_NO**`)`
- **Boolean** `insertAffect(`**STMT_NO, STMT_NO**`)`
- **Boolean** `insertAffectStar(`**STMT_NO, STMT_NO**`)`
- **Boolean** `insertNextBip(`**STMT_NO, STMT_NO**`)`
- **Boolean** `insertNextBipStar(`**STMT_NO, STMT_NO**`)`
- **Boolean** `insertAffectBip(`**STMT_NO, STMT_NO**`)`

- **Boolean** insertAffectBipStar(**STMT_NO, STMT_NO**)
- **Boolean** insertUses(**STMT_NO, VAR_NAME**)

  Insert a list of variables directly or indirectly used by a statement. The statement is given by its index, and the variables are given by their names. New variables are inserted into the PKB if there are any. Returns true if the operation is successful.
- **Boolean** insertModifies(**STMT_NO, VAR_NAME**)
- **Boolean** insertProcUses(**PROC_NAME, VAR_NAME**)

  Insert a list of variables directly or indirectly used by a procedure. New variables are inserted into the PKB if there are any. Returns true if the operation is successful.
- **Boolean** insertProcModifies(**PROC_NAME, VAR_NAME**)
- **Boolean** insertExpression(**STMT_NO, EXPRESSION**)

  Insert an expression contained. An insertion should be made for every sub-expression of the RHS of the original statement. Returns true if the operation is successful.

Below is the abstract API used by the Query Processor:

- **LIST_RES** getEntities(**ENTITY_TYPE**)

  This method is to handle queries with no clause, eg., `Select s; Select c;`
- **Boolean** getBooleanResultOfRS(**RELATION_TYPE, QUERY_INPUT**)

  This method is to handle queries with no synonyms in its **such that** clause, eg.,

  `Select s such that uses(_, "x"); Select s such that follows(3, 5);`
- **LIST_RES** getSetResultOfRS(**RELATION_TYPE, QUERY_INPUT**)

  This method is to handle queries with one synonym in its **such that** clause, eg.,

  `Select p such that uses(p, "x"); Select s such that follows(s, 5);`
- **LIST_OF_LIST_RES** getMapResultsOfRS(**RELATION_TYPE, QUERY_INPUT**)

  This method is to handle queries with two synonyms in its **such that** clause, eg.,

  `Select s such that uses(s, v); Select a such that calls*(p1, p2);`
- **LIST_RES** getSetResultOfAssignPattern(**QUERY_INPUT, EXPRESSION**)

  This method is to handle queries with no synonym in its assign **pattern** clause, eg.,

  `Select a such that pattern a(_, "x");`
- **LIST_OF_LIST_RES** getMapResultOfAssignPattern(**QUERY_INPUT, EXPRESSION**)

  This method is to handle queries with one synonym in its assign **pattern** clause, eg.,

  `Select a such that pattern a(v, "x");`

- **LIST_RES** getSetResultOfContainerPattern(**STMT_TYPE, VAR_NAME**)

  This method is to handle queries with no synonym in its container **pattern** clause, eg.,

  `Select a such that pattern w(_, _);`
- **LIST_OF_LIST_RES** getMapResultOfContainerPattern(**STMT_TYPE, VAR_NAME**)

This method is to handle queries with one synonym in its container **pattern** clause, eg., `Select a such that pattern ifs(v, _, _);`

- **LIST OF LIST RES** getDeclarationsMatchResults(**SYNONYM, SYNONYM**)
  This method is to handle queries with a **with** clause between two declarations (i.e., not effective attributes, such as v.varName and s.stmt#, as described earlier), eg., `Select p with p.procName = v.varName;`
- **LIST OF LIST RES** getDeclarationMatchAttributeResults(**SYNONYM, ATTR**)
  This method is to handle queries with a **with** clause between a declaration and an attribute, eg., `Select p with p.procName = c.procName;`
- **LIST_OF_LIST_RES** getAttributesMatchResults(**ATTR, ATTR**)
  This method is to handle queries with a **with** clause between two effective attributes, eg., `Select r with r.varName = c.procName;`
- **LIST OF LIST RES** getAttributeMatchNameResults(**ATTR, NAME**)
  This method is to handle queries with a **with** clause between an attribute and a given name, eg., `Select r with r.varName = “randomVariable”;`
- **STRING** getNameFromStmtNum(**STRING**)
  This method is to retrieve the attribute value of a synonym, given the synonym’s statement number.

8.2 API Discovery Process

[Good-to-have Appendix]

To help you discover your APIs for PKB and Query Evaluator, start by answering the numbered questions. You can answer ONE question in this appendix to show the thought process in coming up with the APIs. You do not need to answer more than one question in this appendix, but they can still be used in discovering your APIs.

Using subsections, describe how SPA components work with design abstractions (possibly via APIs), following examples in course materials using English.

Feel free to shorten/abbreviate repeated steps. For each interaction, continue description until you feel you are ready to switch to document a given API. Include as many steps as you think it is enough for you to get useful feedback.

Base your answers on the following SIMPLE source program and queries.

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

```
assign a; stmt s; variable v;
```

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)
11	Select a such that Modifies (a, “x”)

Questions:

1. Write a sequence of steps describing how Parser works with ProcTable & VarTable.
   For example,
   a. Insert “main” to ProcTable, return index
   b. Insert “x” to VarTable, return index
2. Write a sequence of steps describing how Parser builds an AST when parsing procedure Main.
3. Describe how Parser works with Follows and Parent
   For example,
   a. Parser must set Follows (1, 2)
4. Describe how Query Processor works with AST and VarTable when evaluating Query #1
   Hint: Query Processor must traverse the AST in depth-first order and find subtrees that match pattern.
   For example, you could start off as follows:
   a. Get root of procedure main, return p
   b. Get first child of p, return p
   c. isMatch(p,”assign”) – no
   d. etc.
5. Describe how Query Processor works with Follows/* when evaluating Query #2, #3, #4, and #5.
6. Describe how Query Processor works with Parent/* when evaluating Query #6, #7, #8, and #9.
7. Describe how Program Parser works with Modifies when parsing statements 1 and 4.
8. Describe how Query Processor works with Modifies when evaluating Query #10, and #11.

8.3 Sample Test Cases

[Compulsory Appendix]

See Section 4

8.4 Any other appendices

Query Preprocessor

The table below shows all strings that are "valid" and the corresponding type of token they are classified under.

String to process	TokenTypes
DIGIT+	Integer
LETTER (LETTER | DIGIT)*	Identifier
stmt | read | print | while | if | assign | variable | constant | procedure	DesignEntity
Modifies	Modifies
Uses	Uses
Parent	Parent
Follows	Follows
Select	Select
such	Such
that	That
Pattern	Pattern
;	Semicolon
_	Underscore
(	LeftParen
)	RightParen
,	Comma
"	DoubleQuote
*	Asterisk