SCHOOL OF COMPUTING

CS3203
Software Engineering Project

Iteration 1 Report

Team Number: 21

Full Name	Email Address	Phone Number
RACHEL TAN XUE QI	E0191632@U.NUS.EDU	98439419
JEFFERSON SIE	E0175320@U.NUS.EDU	90235523
CHEN ANQI	E0324117@U.NUS.EDU	90174780
JAIME CHOW WEN JUAN	E0191616@U.NUS.EDU	92989425
CHEN SU	E0323703@U.NUS.EDU	83759946
TAN WEI ADAM	E0273854@U.NUS.EDU	84359239

Table of Contents

1. Scope of the Prototype Implementation

The features required Project Iteration 1 have been implemented.

For bonus features, we have implemented the following features that are not required for Iteration 1 but are part of the Basic SPA Requirements under Section 4.1 of CS3203 wiki. More information on the bonus features implementation and testing details can be found in the following report under Section 3.4.4.

- The position of the “pattern” and “such that” clause can be swapped in the PQL query.
- UsesP and ModifiesP have been implemented.
- Full pattern matching for var_name and const_value have been implemented.

2. Development Plan

2.1. Milestone of Main Tasks GANTT Chart

Main Task/Week	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Analyse and Design System *(All members)*							
Plan and Design System APIs *(All members)*							
Plan System Acceptance Test *(Rachel)*							
SPA Controller Implementation Task *(Rachel)*							
Front-End Parser Implementation Task *(Rachel, Jaime)*							
PKB Implementation Task *(Adam, Jeff)*							
PQL Implementation Task *(Anqi, Chen Su, Jeff)*							
Unit Testing Implementation Task *(All Members)*							
Integration Test Implementation Task *(All Members)*							
System Test Implementation Task *(Rachel)*							
Run System Test with Autotester *(All Members)*							
System Bug Fixes *(All Members)*							

Fig. 2.1: An overview of the milestone of our project main development tasks

2.2. Main Task and Activities Plan

Activity	**Completed By:**					
	Jefferson	Adam	Jaime	Rachel	Su	Anqi
Implement SPA Controller				*	*	
Front-End Parser Implementation Task						
Implement Tokenizer				*		
Implement Parser			*			
Implement AST API			*			
Program Knowledge Base Implementation Task						
Implement Design Extractor		*				
Implement PKB Storage API	*	*				
Query Processor Implementation Task						
Implement Query Preprocessor						*
Implement Query Syntax Checker						*
Implement Query Parser					*	
Implement Query Evaluator	*				*	
Implement Query Result Projector					*	
Unit Testing Task						
Implement SPA Controller Unit Tests				*		*
Implement Tokenizer Unit Tests				*		
Implement Parser Unit Tests			*			
Implement AST API Unit Tests			*			
Implement Design Extractor Unit Tests		*				
Implement PKB Storage Unit Tests	*	*				
Implement Query Preprocessor Unit Tests					*	*
Implement Query Evaluator Unit Tests					*	*

Implement Query Result Projector Unit Tests					*	
Integration Testing Task						
Implement Front End Parser/PKB Integration Tests			*			
Implement PKB/PQL Integration Tests	*				*	*
System Testing Task						
System Tests that covers for ● No clause ● Such that clauses (Follows, Follows*, Parent, Parents*, Modifies, Uses) ● Pattern clause ● Such that and Pattern Clauses			*	*		
Source Focused System Tests ● Invalid Source ● Complicated Source				*		

Fig. 2.2: An overview of the activity development plan

2.3. Project Roles & Meeting Plans

Team members	**Roles**
Jefferson Sie	Team Leader, Developer (PKB & PQL)
Rachel Tan	Testing IC, Developer (Front-End)
Jaime Chow	Documentation IC, Developer (Front-End)
Chen Su	Developer (PQL)
Chen Anqi	Developer (PQL)
Adam Tan	Developer (PKB)

Fig. 2.3-1: The roles of each team member

Meeting Plans / Days	**Mon**	**Tues**	**Wed**	**Thurs**	**Fri**	**Sat**	**Sun**
Progress Check							
Group Coding							
Go through bugs together							

Fig. 2.3-2: Fixed Meeting Schedule from Weeks 3 to 7 for every Monday, Tuesday and Saturday

3. SPA Architecture

3.1. Overview

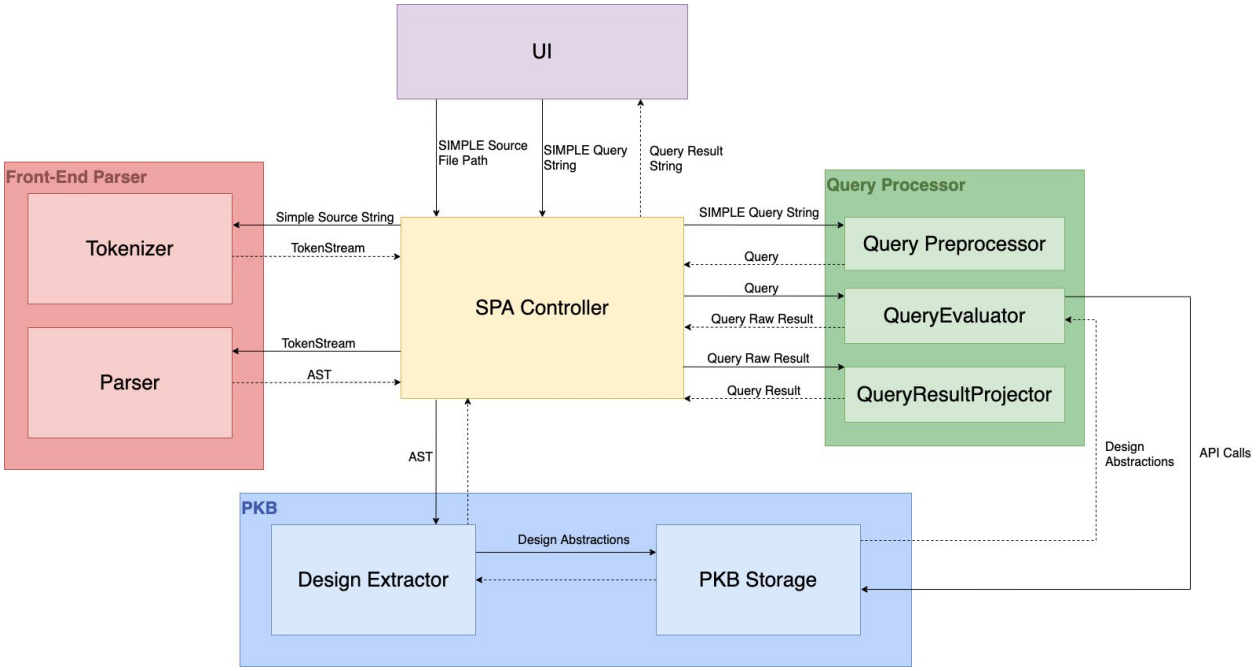

Fig. 3.1-1: Overview of SPA Architecture Component

Our SPA consists of four main components:

1. **SPA Controller** - Acts as the middleman for communication between components. As it is only responsible for trivial tasks such as reading the SIMPLE source file contents into a string and calling component API, we chose to omit its explanation in this report.
2. **Front-End Parser** - Generates an abstract syntax tree (AST) tree based on a SIMPLE source program input.
3. **PKB** - Traverses the AST to extract design abstractions and store them into internal data structures. Fetches said design abstractions from the internal data structures via API calls from the Query Processor.
4. **Query Processor** - Processes and evaluates a SIMPLE query input, and outputs the formatted query result.

The interaction between components are as follows:

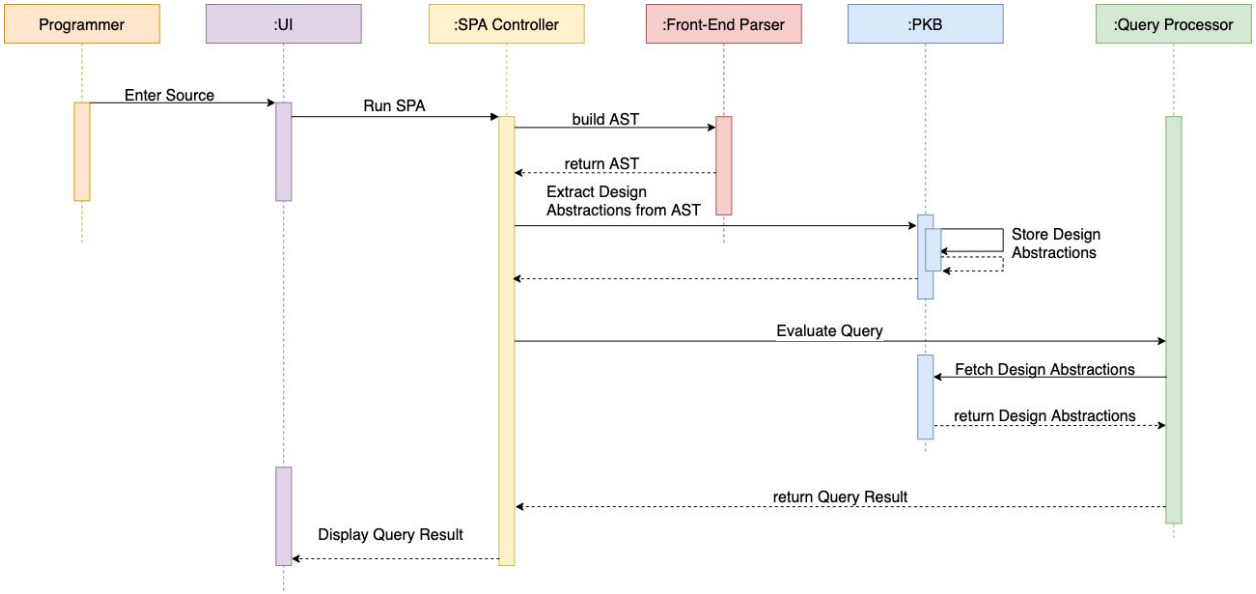

Fig. 3.1-2: High Level UML Sequence Diagram of SPA Program

Design Abstraction Storage Process

When the SPA Controller receives the file path string from the UI, it reads the file content into a string. The source string is passed to the Tokenizer, which returns a stream of tokens. The Token Stream is then passed to the Parser, which generates the AST. The AST is then passed back to the SPA Controller.

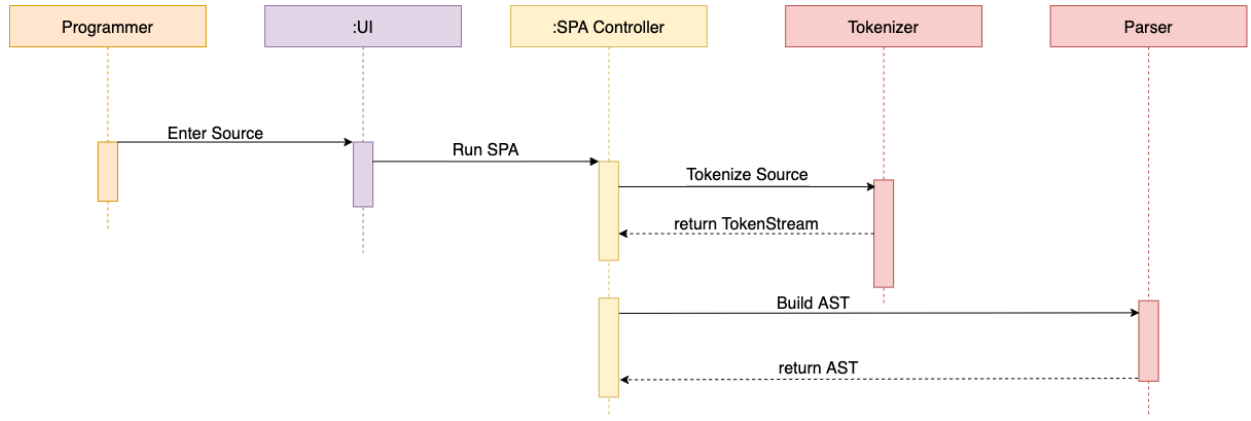

Fig. 3.1-3 High Level UML Sequence Diagram of Front-End Parser

The SPA Controller will then pass the AST to a PKB object. The PKB traverses the AST to extract design abstractions and stores them in internal data structures within the PKB via internal API calls.

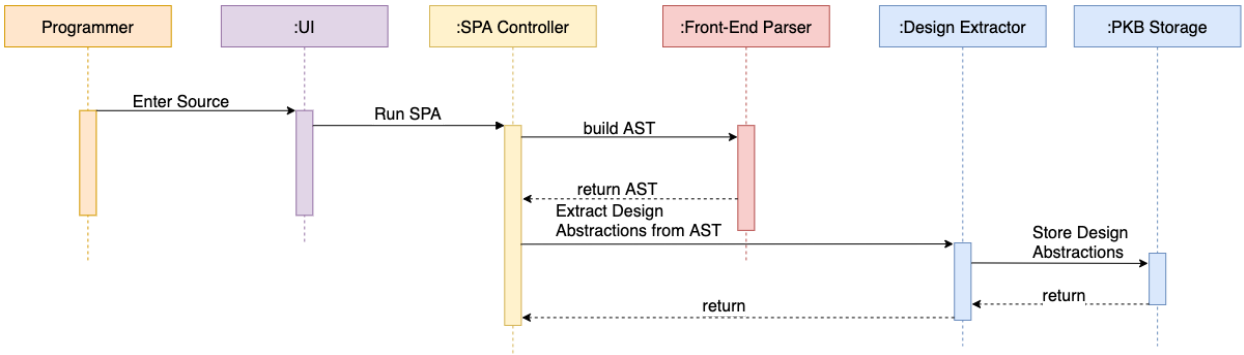

Fig. 3.1-4 High Level UML Sequence Diagram of PKB

Query Processing Process

The SPA Controller receives the query string and a reference to a result list that should be populated with the formatted query evaluation result. The query string is passed to the Query Preprocessor for parsing, which returns a Query object. The SPA Controller then passes the Query object to the Query Evaluator that returns an unformatted list of values that have been selected as answers to the Query. The SPA Controller then passes the raw result list and the result list reference to the Query Result Projector. The Query Result Projector will then populate the result list with the correctly formatted results.

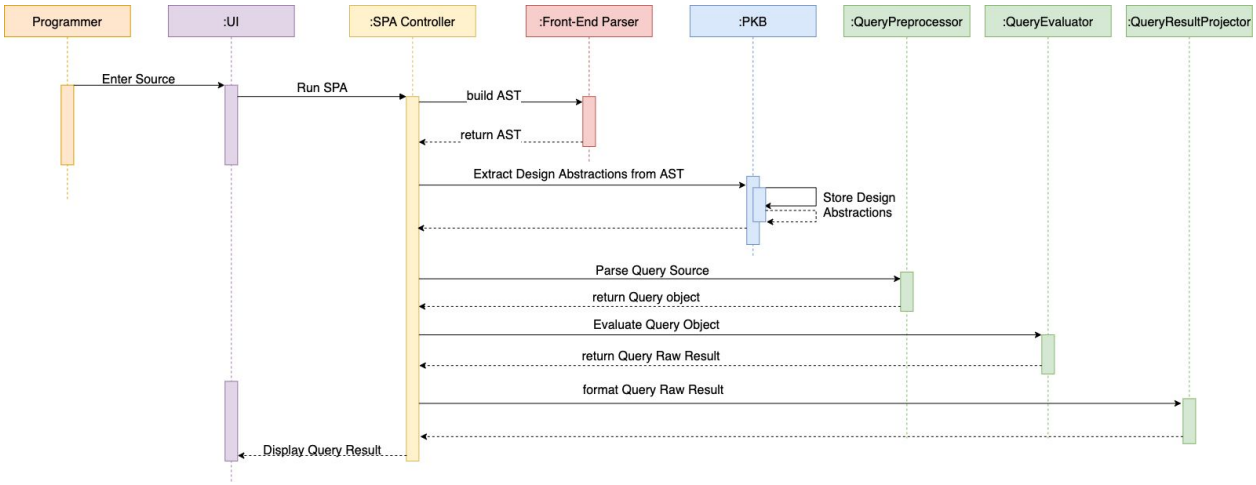

Fig. 3.1-5 High Level UML Sequence Diagram of Query Processor

3.2. Front-End Parser

Overview

The Front-End Parser is responsible for the generation of the AST. It takes in a SIMPLE source program text as input, and outputs an AST. For abstraction purposes, the Front-End Parser has been decomposed into 2 subcomponents - the **Tokenizer** and the **Parser**.

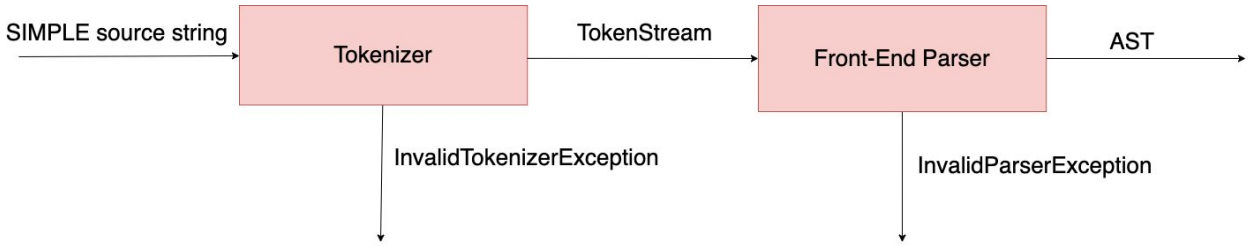

Fig. 3.2-1: Production Pipeline of Front End Parser Architecture

Design Considerations: AST Implementation

We have chosen to add AST generation to our SPA Program process. The implemented AST structure coincides completely with what was taught during lectures and during design considerations, we avoid making alteration to the structure. An AST data structure is implemented as a tree graph populated by node objects that inherit from the given TNode class.

As seen in Fig. 3.2-2, each node class contains pointers to its corresponding child nodes. For instance, an `AssignNode` would point to a `VariableNode` and an `ExpressionNode`. Each node class is also assigned a Design Entity enum type that allows the Design Extractor to determine the type of node it is looking at without knowing AST implementation such as class names. There is an assumption that the developer implementing the Design Extractor is knowledgeable of the AST structure taught in lectures and will use that knowledge to call the correct AST API methods. For instance, they should know that only a node of Design Entity type “If” can call `getElseStatementList()`. Thus, it is imperative that the structure of the SPA program AST structure does not deviate from standard. The mappings between Design Entity types and allowed API calls are documented in Section 7.

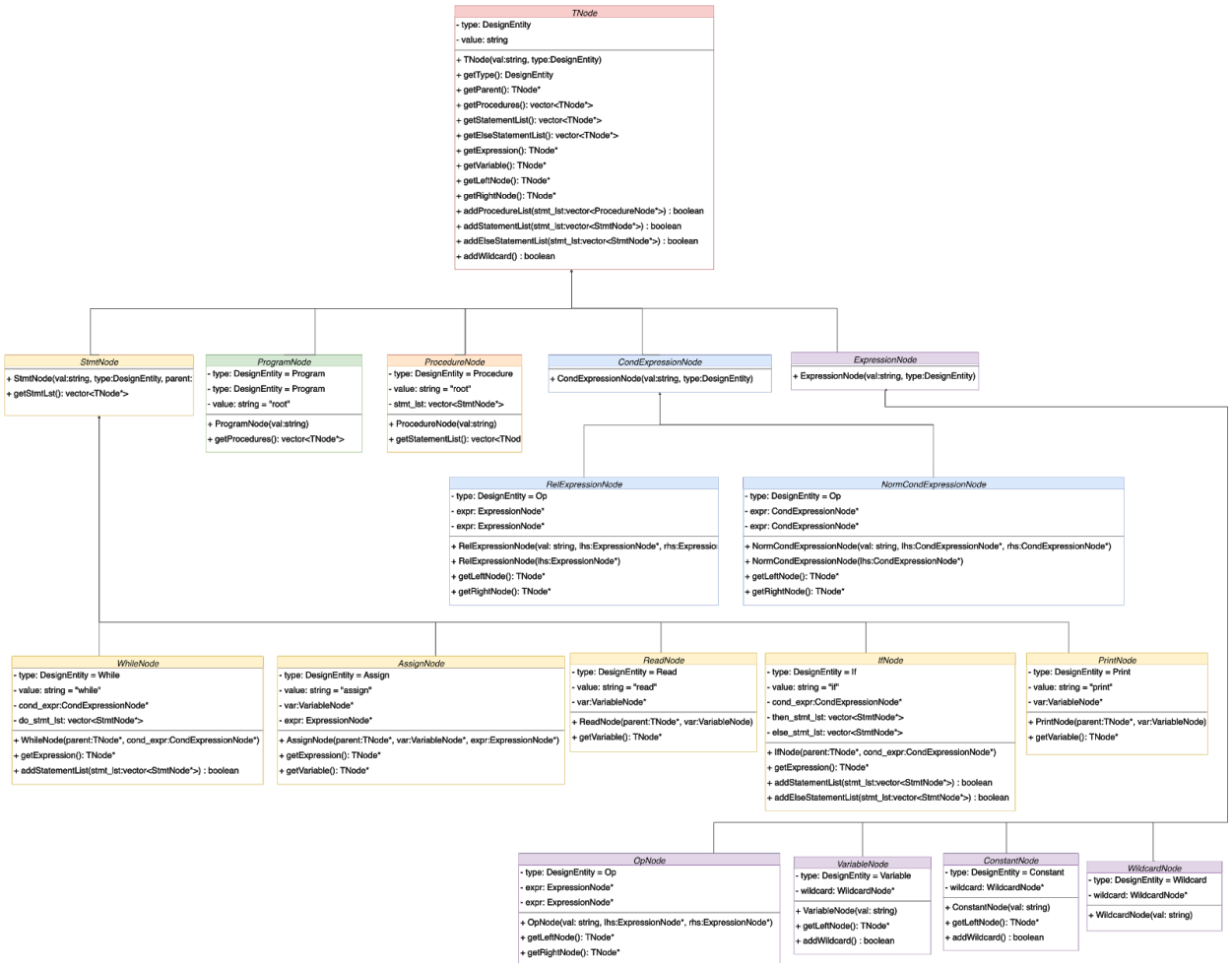

Fig. 3.2-2: AST Class Diagram. Composition relationships omitted to avoid cluttering.

When deciding whether to implement as AST, we considered the following:

(1) Chosen Decision: Implement AST

Pros: AST generation is considered part of the tried-and-tested parsing and static code analysis process. The AST also captures the structure of the SIMPLE source while omitting unnecessary syntactic details such as semi-colons and parentheses, allowing for a more focused design extraction process.

Cons: Development resources would need to be spent implementing the AST data structure and conversion of the SIMPLE source to AST. Time taken and memory required to process the SIMPLE source program will be increased.

(2) Alternative: Do not implement AST

Pros: There is an argument to be made that the AST is an unnecessary addition to the SPA program. The SIMPLE concrete syntax grammar is arguably simple enough that structural validity checks and data abstraction extraction can be performed directly from the source string

or Token Stream. This alternative is simpler and faster to implement, as we do not need to spend effort designing and coding the AST data structure.

Cons: This decision requires us to deviate from the standard parsing process. There is a likely possibility we will make oversights regarding the logic and design when implementing a less standard parsing process.

Justification for choosing AST implementation:
Due to our unfamiliarity with Parser implementation, we do not wish to reinvent the wheel by attempting to implement a non-standard approach to parsing.

The drawbacks to implementing the AST are also negligible. As discussed, the Design Extractor has been moved to the PKB. Thus, the team implementing the Front-End Parser can dedicate all their development efforts into developing AST generation. Furthermore, there are many resources online on AST implementation for reference due to its wide usage in standard parsing processes.

As there is also no time constraints for the processing of the SIMPLE source in Iteration 1, any impacts to time and memory can be ignored as well.

Design Considerations: Move Design Extractor to PKB

The Front-End Parser should live up to its namesake and parse only. This means the goal of the Parser is to determine if the source program input is structurally and syntactically valid. When choosing the core responsibilities of the Parser, we prioritised high cohesion; the output or behaviour of the parser should be strongly tied to checking structural validity.

(1) Chosen Decision: Move Design Extractor to PKB
We decided that the Front-End Parser should only be responsible for parsing a Simple Source input and generating an AST only. The Design Extractor sub-component is instead moved to the PKB component.

Pros: The output of the Front-End Parser is strongly related to the Front-end Parser's defined goal as successful AST generation implies structural validity. AST generation is also a widely applicable functionality that can improve reusability of the Front-End Parser.

Cons: Both the Front-End Parser and PKB will be coupled to the AST API for AST generation and design abstraction extraction respectively. However, this is not our priority concern for this design consideration.

(2) Alternative: Design Extractor is part of Front-End Parser
The alternative core would be to keep the Design Extractor as a Front-End Parser component, allowing the AST to become an internal data structure of the Front-End Parser.

Pros: Reduces coupling between the PKB and AST as the responsibility of traversing the AST is now handed to the Front-End Parser.

Cons: Lower cohesion as the extraction of Design Abstractions is outside the job scope of checking structural validity. Furthermore, compared to the chosen decision, the Front-End Parser now has less reusability as its functionality is strongly tied to the PKB.

Justification for moving Design Extractor to the PKB:
The pros of moving the design extractor align more with our design priorities of enforcing high cohesion and better separation of concerns. The applicable functionality of the Front-End Parser component is also a huge benefit; for instance the Front-End Parser can be reused to generate a SIMPLE arithmetic expression AST for pattern query evaluation.

3.2.1. Subcomponent 1: Tokenizer

Overview

The Tokenizer is responsible for tokenizing or splitting the input source program into individual tokens and parsing it to Parser in as a vector of tokens. SPA Controller read the source program into string format and parses it to Tokenizer. The Tokenizer will tokenize the source program into its token as value, assigning the token with a token type and finally returning as a vector of tokens for the Parser.

Design Considerations: Token Object

In the Tokenizer subcomponent, when the token has been split, we want to categorise these tokens with an identified meaning so that it can be used in semantic analysis in Parser. These tokens and their types are identified based on the requirement stated by SIMPLE syntax grammar rules and lexical token rules as seen in Fig. 3.2.1-1.

Token Type	Sample Token Values
Identifier	Any var_name and proc_name that satisfy NAME rules
Keyword	read, print, while, if, assign
Separator	(,), {, }, ; ,
Operator	+, -, *, /, %, =, !, <, >, ||, &&, <=, >=, ==, !=
Literal	Any constant value that satisfy INTEGER rules
EndOfFile	EndOfFile token will be created and added to the end of token streams created

Fig. 3.2.1-1: Examples of Token values

Design Considerations: Token Stream Data Structure Choice

During tokenization, we want to split and assign each token with its type and pass the whole token stream for parser to further process. There are a few design considerations for the implementation of token stream as this will determine how the parser will be implemented to traverse through the token stream. The main criteria for generating the suitable token stream while the parser processes them is to ensure that there is flexibility in accessing the token stream. The ability to insert and delete tokens from the token stream is not the main concern as there should be no changes to the source program that has been tokenized into a token stream. Time complexity and memory storage are also low concerns in our design considerations.

(1) Chosen Decision: Vector of Token Class Objects

Following the principles of Object Oriented programming, any information that is about the token such as the token type and token values, is created in a Token class. This will tokenize the source program and create each token as a Token with its value and types.

Pros: Provides the ability for random access for Parser where necessary.

Cons: Any insertion or deletion of elements will be less efficient as compared to the use of Linked List. In vector insertion and deletion at start or middle will make all elements to shift by one. If there is insufficient contiguous memory in vector at the time of insertion, then a new contiguous memory will be allocated and all elements will be copied there.

(2) Alternative 1: Vector of Tuples or Pairs
Similar to the chosen design decision, the comparison was made between creating a Token class for information related to the tokens or creating a Tuples or Pairs where it will contain token values and token types in string type.

Pros: Using the Pair or Tuples is easy to implement and it was quickly taken into consideration.

Cons: With more iterations to come, the ease for extendibility is considered and Pair is limited to have two values. There might be more information about Token that we want to capture and parse to Parser in the later iterations. Tuples do not have limitations to the number of values. If there is more information to capture about Token then using OOP and creating a Token class would be easier for Tokenizer and Parser to interact.

(3) Alternative 2: Linked List of Token Class Node
Pros: Creating the Linked List is insertion and Deletion in List is very efficient as compared to vector as described in (1) Chosen Decision because it inserts an element in list at start, end or middle, internally just a couple of pointers are swapped.

Cons: No random access allowed. A head node will most likely be returned to indicate the start of the Linked List of Token Class and parsered to Parser. Parser will then iterate through from the start, and can only traverse back and forward one at a time. This has lower flexibility for Parser to access desired elements when necessary.

Justification for choosing Vector of Token Class Objects:
We finalised our decision with a vector of Token Class because providing the flexibility to access where necessary when Parser is iterating through while building AST is our main priority. Parser can easily call the private APIs to get each element's token type and values in the Vector of Token. In addition, the main disadvantage about using a vector is negligible as the Parser will not insert or delete any of the token stream once Tokenizer has created the token stream for Parser to process.

Exceptions

A Tokenizer exception is thrown when the source program is an empty file or contains spaces only and when the token does not satisfy the Identifier lexical rule. A message will be printed to the output channel indicating the exception type and the received value. This will inform the user when it is unable to pass the tokenization stage. For example, given the following source program:

```
procedure main {
      1_var = 2;
}
```

Fig. 3.2.1-2: Sample source

The Tokenizer detects an invalid variable name “1_var”. A `InvalidTokenizerException` would be thrown with the following message:

```
Invalid Lexical token Exception! Received Invalid Lexical Token:
1_var
```

Fig. 3.2.1-3: Sample Exception

3.2.2. Subcomponent 2: Parser

Overview

The Parser is responsible for the building of the AST. It is instantiated and initialized by the SPA Controller with a vector of tokens generated by the Tokenizer. The Parser looks at the Token string value or Token Type to determine the type of node in the AST tree it should be attempting to create and whether the SIMPLE concrete syntax grammar is adhered to.

Design Considerations: Parser Method

We have implemented a single pass **Top-Down Recursive Descent Parser**. This form of Parser was chosen as the SIMPLE grammar, barring arithmetic expressions, is LL(k) and thus suitable for this method of parsing. As an example, the Parser will parse to following program as follows:

```
procedure main {
      read x;
}
```

Fig. 3.2.2-1: Sample source

With reference to the node objects depicted in Fig. 3.2-2, the Parser will evaluate tokens in sequential order as follows:

Current Token Value	**Current Token Type**	**Parser Behaviour**	**Parser Expectation**
-	-	A root ProgramNode is created.	Next token value must be “procedure”
“procedure”	Keyword	A ProcedureNode is created an attached as a child node to the ProgramNode	Next Token Type must be “Identifier” or “Keyword”. We accept “Keyword” type tokens to accommodate the fact words defined by the SIMPLE CSG can still be used as procedure names.
“main”	Identifier	The value of the ProcedureNode is assigned the current token value.	Next Token value must be “{“
“{“	Separator	No action taken by the Parser.	Check if the token value after the next Token has a value of “=”. Otherwise, the next token value must be “read”, “while”,”print” or ”if”.
“read”	Keyword	Parser acknowledges that it is parsing a Read statement due to the “read” token value. A ReadNode is created and attached as a child	Next Token Value must be “Identifier” or “Keyword”. We accept “Keyword” type tokens to

		node to the ProcedureNode.	accommodate the fact words defined by the SIMPLE CSG can still be used as variable names.
“x”	Keyword	A VariableNode is created and attached as a child node to the ReadNode. The value of the VariableNode is assigned the current token value.	Next Token value must be “;”.
“;”	Identifier	No action taken by the Parser.	Check if the next Token value is “}”. Otherwise, check if the token value after the next Token has a value of “=”. Otherwise, the next token value must be “read”, “while”,”print” or ”if”.
“}”	Separator	Parser acknowledges that there are no more child nodes to add to the ProcedureNode.	Check if the next Token Type is “EOF”. Otherwise, the next Token must have a value of “procedure”.
“EOF”	EndOfFile	Parser acknowledges that there are no more Tokens that need to be parsed. The parsing process ends and the ProgramNode is returned.	-

Fig. 3.2.2-2: Sequential process of parsing the Sample Source in Fig. 3.2.2-1

If the Parser expectations are not met at any point during the parsing, an exception is thrown.

Design Considerations: Expression Parsing

SIMPLE arithmetic expressions are not able to be parsed via standard recursive descent parsing due to left recursion in the SIMPLE grammar.

When deciding on a solution for expression parsing, we considered the following:

(1) Chosen Decision: Pratt Parser

We have decided to implement the Pratt Parser, which is a one-pass Top-Down Recursive Operator Precedence Parser. The Pratt Parser handles operator precedence by assigning binding power (BP) to operators based on their precedence.

The BP assigned to arithmetic operators are as follows:

- ‘+’ and ‘-’ have a BP of 1
- ‘*’, ‘/’ and ‘%’ have a BP of 2

As an example of Pratt Parser parsing, given the following assign statement:

```
x = 4*x+3;
```

Fig. 3.2.2-3: Example of expression

With reference to the node objects depicted in Fig. 3.2-2, the Parser will evaluate tokens in sequential order as follows:

Recursive Level (by end of Parser Behaviour)	**Current Token Value**	**Current Token Type**	**Current Token BP**	**BP**	**Parser Behaviour**	**Parser Expectation**
0	“=”	Operator	-	0	The Parser makes a method call to create an ExpressionNode with a set BP of 0.	The next Token Type must be “Identifier”, “Keyword” or “Literal”
0	“4”	Literal	-	0	A VariableNode `var_4` is created. the value of the `var_4` is assigned the current token value.	Check if the next Token Value is “+”,”-”,”/”,”*” or ”%”. If true, check if the Current Token BP is greater than the set BP.
1	“*”	Operator	2	2	A OpNode `op_times` is created. the value of the `op_times` is assigned the current token value. `var_4` is attached to `op_times` as the left child node. As the Token BP is greater than the current BP, the Parser makes a recursive call to create an Expression Node with a set BP equal to Current Token BP.	The next Token Type must be “Identifier”, “Keyword” or “Literal”
1	“x”	Identifier	-	2	A VariableNode `var_x` is created. the value of the `var_x` is assigned the current token value.	Check if the next Token Value is “+”,”-”,”/”,”*” or ”%”. If true, check if the Current Token BP is greater than the set BP.
0	“+”	Operator	1	1	The token BP is less than the current BP, the Parser breaks out of a recursion level and returns `var_x`. `var_x` is attached to `op_times` as the right child node. A OpNode `op_plus` is created. The value of the `op_plus` is assigned the current token value. `op_times` is attached to `op_plus` as the left child node.	The next Token Type must be “Identifier”, “Keyword” or “Literal”

					As the Token BP is greater than the current BP, the Parser makes a recursive call to create an Expression Node with a set BP equal to Current Token BP.	
1	“3”	Identifier	-	1	A VariableNode `var_3` is created. The value of the `var_3` is assigned the current token value.	Check if the next Token Value is “+”,”-”,”/”,”*” or ”%”. If true, check if the Current Token BP is greater than the set BP.
0	“;”	Separator	-	0	The Parser acknowledges that the expression has ended, breaks out of the current recursion and returns `var_3`. `var_3` is attached to `op_plus` as the right child node. `op_plus` is returned.	-

Fig. 3.2.2-4: Sequential process of parsing the Sample Source in Fig. 3.2.2-3

Pros: Implementation is simple and well-documented online. Furthermore, its characteristics, namely that it is top-down, recursive and one-pass, coincide with our existing parser characteristics. Maintenance of any additional data structures is not required for implementation.

Cons: The base Pratt Parser algorithm coupled with our AST implementation is not able to handle expressions with parentheses. That particular case must be handled separately.

(2) Alternative 1: Shunting Yard Algorithm

The Shunting Yard Algorithm may be used to implement two-pass Bottom-Up Operator Precedence parsing. It uses stacks to manage the expression operators and operands when parsing infix expressions.

Pros: Implementation is simple and well-documented online.

Cons: Increased memory cost as it requires the maintenance of a stack for parsing.

(3) Alternative 2: Eliminate Left Recursion from SIMPLE CSG

We considered altering the SIMPLE Concrete Syntax Grammar to eliminate left recursion from the grammar.

Before: expr: expr ‘+’ term | expr ‘-’ term | term term: term ‘*’ factor | term ‘/’ factor | term ‘%’ factor | factor
After: expr: term expr’ expr’: ‘-’ term expr’ | ’+’ term expr’ | null term: factor term’ term’: ‘*’ term’ | ‘/’ term’ | ‘&’ term’ | null

Fig. 3.2.2-5: SIMPLE CSG alterations to remove left recursion

Pros: It is trivial to change and implement the SIMPLE concrete syntax grammar.

Cons: Requires alteration to the AST structure to accommodate the grammar changes.

Justification for choosing Pratt Parser:
We chose to implement a second Parser to handle expression parsing separately over altering the SIMPLE grammar to avoid the complications that would arise when altering the AST structure. The principle of Separation of Concerns would be violated as the PKB would now have to be made aware of changes for correct AST traversal, as opposed to relying on general knowledge about the standard AST structure. A second Parser would allow for arithmetic expression parsing whilst maintaining the current AST structure.

As the Pratt Parser is a one-pass parser that does not require the maintenance of any data structures to implement, we considered it superior and more efficient to the Shunting Yard Algorithm method.

Exceptions

A parser exception is thrown when the source input is structurally or syntactically invalid. A message will be printed to the output channel indicating the expected value and the received value.

For example, given the following source program:

```
procedure main {
      1 = 2;
}
```

Fig. 3.2.2-6: Sample source

The Parser will detect “1” as an invalid name for a variable. An `InvalidParserException` would be thrown with the following message:

```
Invalid Assign Statement: expected variable value, got ‘1’.
```

Fig. 3.2.2-7: Sample Parser Exception

3.3. PKB

Overview

The PKB is responsible for the extraction and storage of design abstractions. It takes in an AST as input, traverses the AST to extract design abstractions, and stores them in internal data structures. For abstraction purposes, the PKB has been decomposed into 2 subcomponents - the **Design Extractor** and the **PKB Storage**.

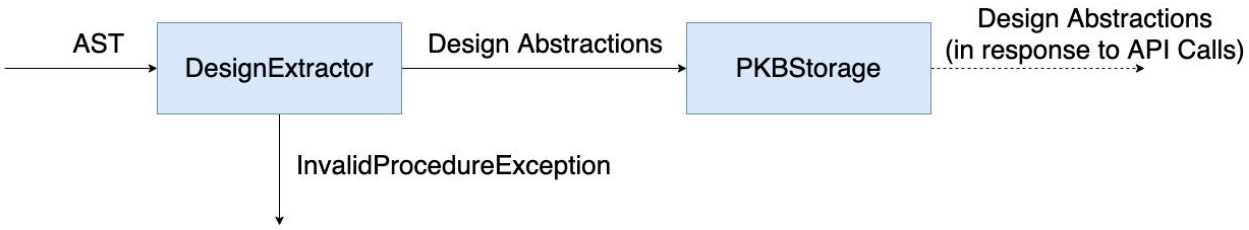

Fig. 3.3-1: Overview on PKB Architecture

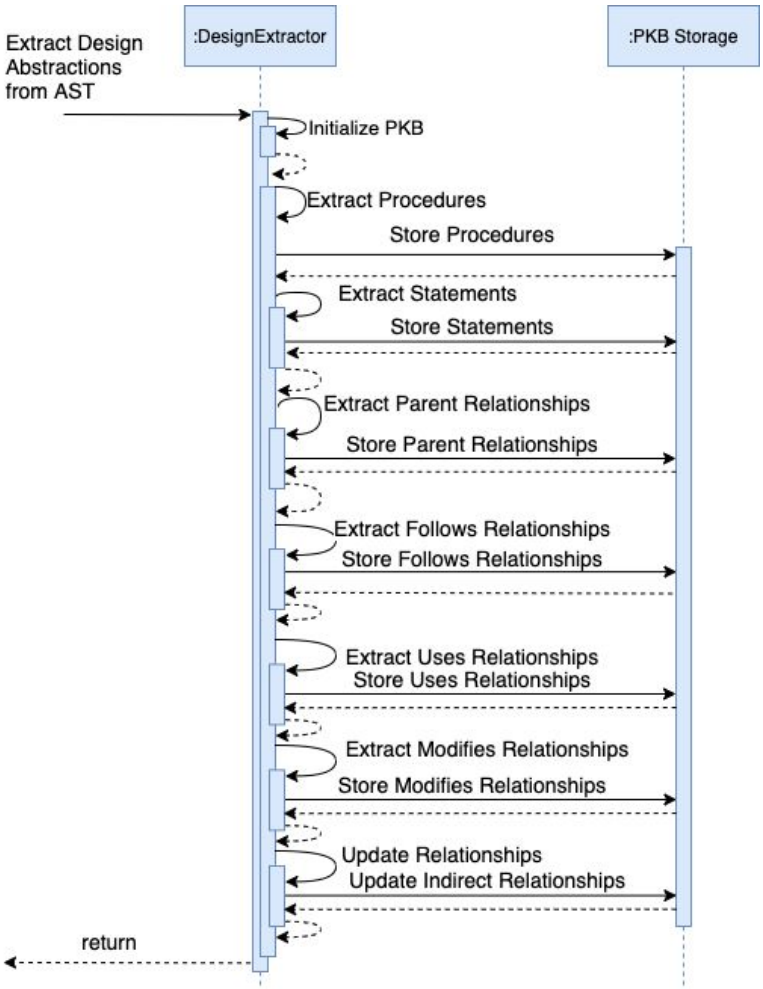

Fig. 3.3-2: Sequence Diagram for initializing PKB

3.3.1. Subcomponent 1: Design Extractor

Overview

The Design Extractor is responsible for the extraction of design abstractions. It traverses the AST passed to it by the SPA Controller to extract relationships. It then stores these relationships in the PKB's internal data structures.

Design Consideration: Number of Passes for Design Extraction

The number of passes of the AST affects the data population performance of the SPA. However, since the requirements of the project do not impose much restrictions for this performance, we can prioritise other areas such as the cleanliness of our code which allows for easier debugging.

(1) Chosen Decision: Multi-pass Design Extraction

The Design Extractor will make two passes to extract all design abstractions from the AST. It first performs a simple left-first traversal of the AST to extract the basic design entities and relationships from the AST. On the second pass, the Design Extractor updates the relevant secondary relationship traits of each design entity.

Pass	Design Entities Extracted	Relationships Extracted
1	Variables Constants Procedures If Statements Assign Statements Read Statements While Statements Print Statements	Follows Parent Modifies (Direct) Uses (Direct)
2		Follows* Parent* Uses (Indirect) Modifies (Indirect)

Fig. 3.3.1-1: Design abstractions extracted per pass

As an example, assume the Design Extractor receives the AST for the following program:

```
procedure main {
    if (x == y) then { //1
        while (y == z) { //2
            z = e; //3
        }
        e = a; //4
    } else {
        if (a == b) then { //5
```

```
            b = d; //6
        } else {
            c = e; //7
            d = c; //8
        }
    }
}
```
Fig. 3.3.1-2: Sample source for subsequent figures

To showcase the first pass extraction process, we will look at the extraction of the Parent, Follows and direct Uses relationships. As illustrated in the figure below, for the extraction of Parent relationships, each statement is stored using an insert function that takes in the parent statement index and stores it in the ParentRelationship table with the index {parent, child}.

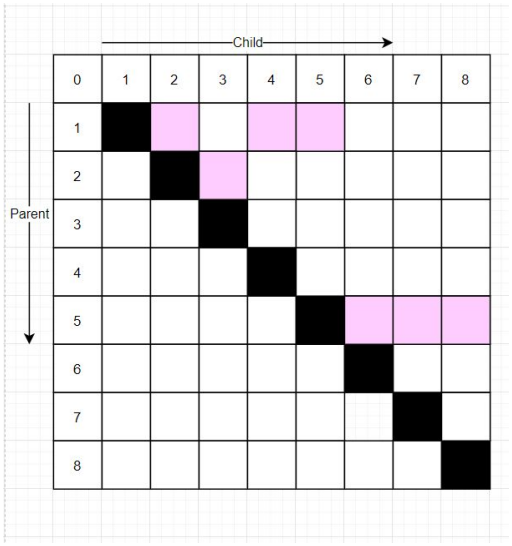

Fig. 3.3.1-3: Pink squares for parent relationship

As illustrated in the figure below, for the extraction of Follows relationships, each statement is stored using an insert function that takes in the previous statement's index in the statement list and the current statement's index and stores it in the FollowsRelationship table with the index {previousStatementIndex, currentStatementIndex}

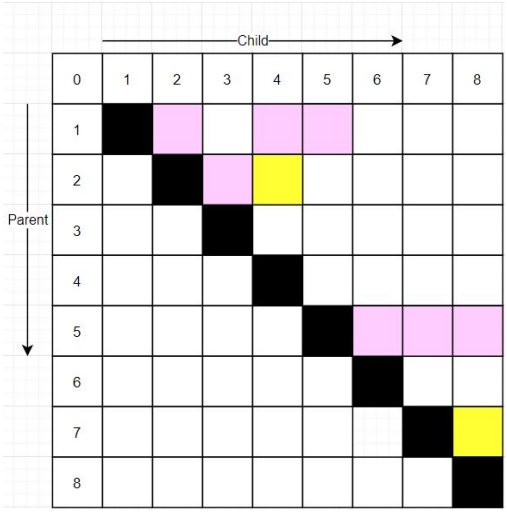

Fig. 3.3.1-4: Yellow squares for follows relationship

As illustrated in the figure below, for the extraction of Uses relationships, each statement is updated with its immediate uses relationship and each pair is stored in a table to prepare for the updating of secondary uses relationships

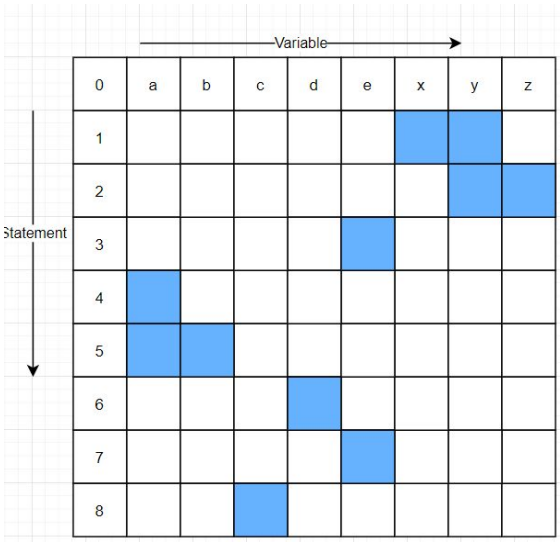

Fig. 3.3.1-5: Blue squares for primary uses relationship

Fig. 3.3.1-6 below explains the second pass. We updated the list from the bottom up since the structure of the program meant that for the secondary relationships, prior statements relied on primary relationships of the latter but not the other way around. This means that we did not require recursive updating for each statement; we just need to update each statement once more. The time required for second pass extraction is O(n^2). Starting from statement 8, we know from the first pass that statement 8 uses variable c. We also know that 8 is the child of 5. Therefore, we add the indirect relationship of Uses(5, c). Now that we know statement 5 uses c, we can also add Uses(1, c) since statement 1 is the parent of statement 5.

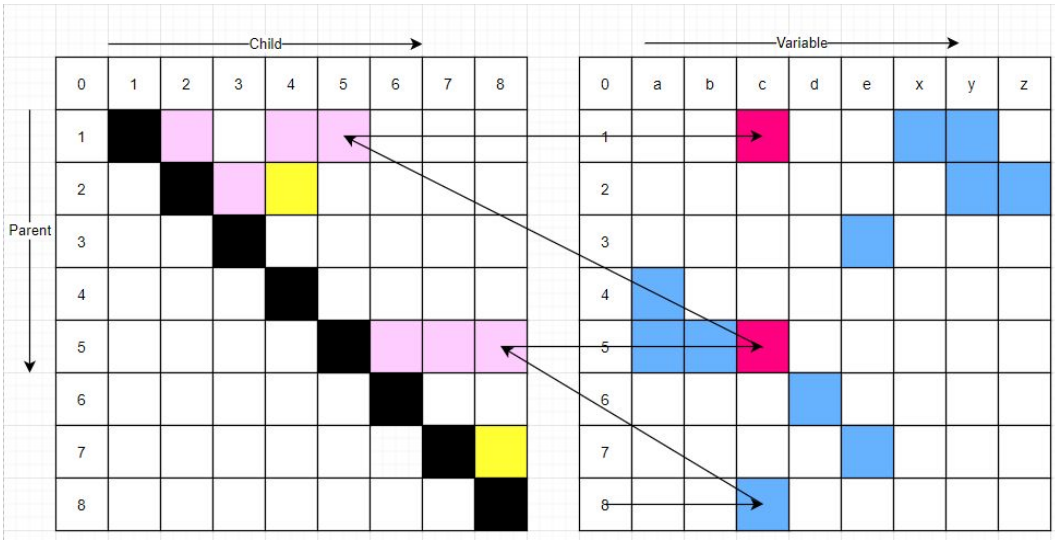

Fig. 3.3.1-6: Red squares for secondary uses relationship, arrows indicate the order of updates

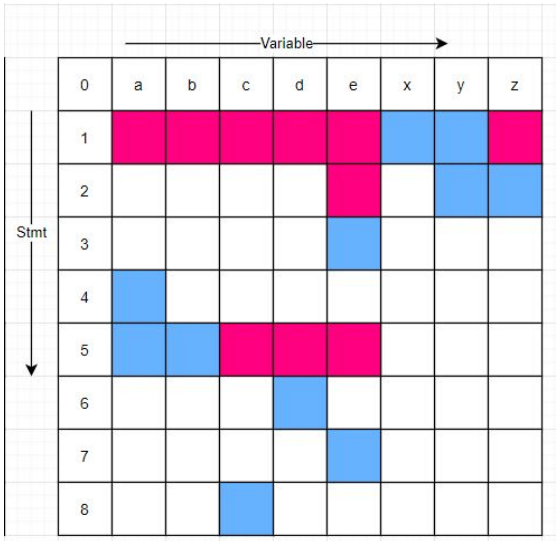

Fig. 3.3.1-7: Final uses table

For the extractions of Follows* relationships, we look at each statement list, starting from the end. Where Sn refers to the current statement number, we extract as follows: Sn follows* Sn-1, Sn-2 follows* (Sn-1 AND follows* of Sn-1). By induction S1 follows* (S2 AND follows* of S2) which we would already be updated since we began updating from Sn.

```
procedure main {
    a = a; //1
    b = b; //2
    if (e == e) then { //3
        f = f; //4
        g = g; //5
        h = h; //6
    } else {
        d = d; //7
    }
    z = z; //8
}
```

Fig. 3.3.1-8: Sample source for subsequent figures

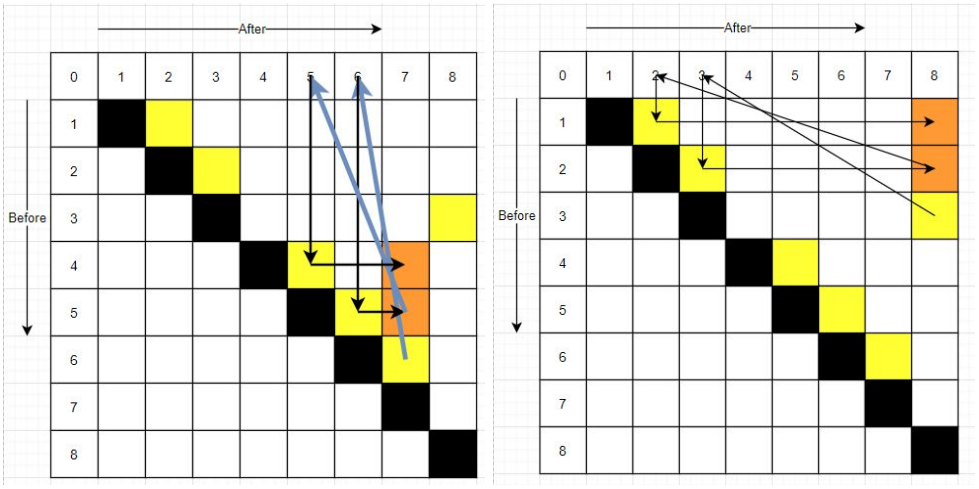

Fig. 3.3.1-9: Updating of follows star relationship - Yellow squares for Follow, Orange squares for Follow*

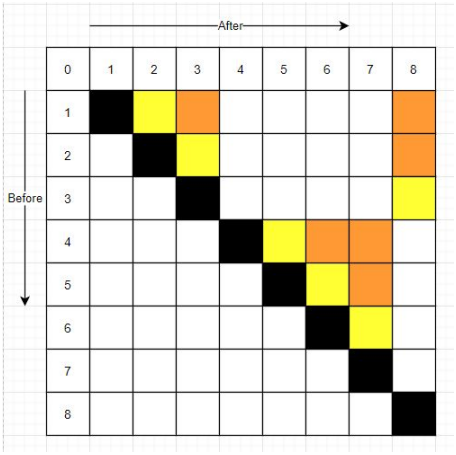

Fig. 3.3.1-10: Completed follows star table

Pros: Reduces overhead for extracting second order relationships, reduces runtime variables required to ensure that the relationships can be accurately extracted. Separating the types of designs extracted to individual passes lowers the coupling between the functions and allows for easier debugging.

Cons: Multiple passes means that the expected runtime would be predicted to be twice as long as a one pass design.

(2) Alternative: Single-pass Design Extraction
Pros: Single pass allows for relationships to be extracted simultaneously as the extractor traverses the AST. The reduced overhead along with only requiring one pass means that the runtime is expected to be lower than a multi-pass design.

Cons: Extraction functions would be coupled together and if there were bugs present or if there was a need to add additional relationships, we would require extremely careful changes to ensure that the new code does not break any of the old functionalities.

Justification for choosing Multi-pass Design Extraction:
Since the main aim was not efficiency and speed but rather to ensure that the program could be initialized correctly, the pros of the single-pass design did not provide an edge over the multi-pass design. Having cleaner code with easily separable components allowing for swift debugging gives us an edge when trying to meet deadlines.

Exceptions

A PKB exception is thrown when there exists two procedures with the same name within the SIMPLE source. A message will be printed to the output channel informing the user of the issue. For example, given the following source program:

```
procedure main {
      var = 2;
}

procedure main {
      var = 2;
}
```

Fig. 3.3.1-11: Sample source

The Design Extractor detects two procedures of the name “main”. An `InvalidProcedureException` would be thrown with the following message:

```
Repeated procedure name: main
```

Fig. 3.3.1-12: Sample Exception

3.3.2. Subcomponent 2: Storage

Overview

The storage subcomponent stores information encompassing all the relationships of the SPA. It is critical that the storage method optimises the search and retrieval of information regarding these relationships when requested so that each query is processed in the least time possible.

Design Consideration: Internal Data Structure

The internal data structure implemented is what determines the efficiency of search and retrieval. Having an efficient data structure will ensure that our answering of queries will meet the project requirements. We narrowed down our choices to two possible solutions, the first being unordered maps and second being AST. However, this choice only exists for relational relationships. We decided to stick with AST implementation for pattern queries as we could easily match the requested pattern with subtrees of the AST.

(1) Chosen Decision: Unordered Maps

```
unordered_map<int, vector<int>> parent_list;
//if i is parent of j
parent_list.insert(i, j)
```

Fig. 3.3.2-1: Example of inserting a parent relationship

Parent (statement numbers)	Children (statement number(s))
1	2, 3, 4
5	6
6	7, 8

Fig. 3.3.2-2: Example of a populated parent map

The underlying implementation of an unordered map is a hash table, where the search time for a particular key is amortised O(1). Fig. 3.3.2-1 shows an example of how we made use of unordered maps to store information from the Parent relationship, and Fig. 3.3.2-2 shows an example of an unordered map for the Parent relationship that has been populated.

Pros: The search time is a lot faster than AST, especially when the data stored is large.

Cons: Additional space constraint as additional data structures are implemented.

(2) Alternative: AST

Retrieving information from the AST requires sequential access from the root node, traversing the entire tree to find the relevant statement, and then checking if the relationship holds for the statement.

Pros: Fewer data structures to implement as AST has already been created.

Cons: Slower search time for a key.

Justification for choosing Unordered Maps

When comparing unordered maps with AST, the AST implementation takes O(n) time to search for a key while the unordered_map implementation takes O(1) time.

3.4. Query Processor

Overview

The Query Processor component is divided into four main subcomponents:

1. **QueryPreProcessor**, which handles the pre-processing of query string and packaging into Query struct, and performs semantic checks on the query.
2. **QuerySyntaxChecker**, which performs syntactic checks on the query structure and each subpart. It is called by QueryPreProcessor.
3. **QueryParser**, which performs lower-level tasks like tokenizing string by delimiter, removing spaces, regex matching etc., and is called by QuerySyntaxChecker.
4. **QueryEvaluator**, which takes in a Query object and evaluates the raw result.
5. **QueryResultProjector**, which formats the raw results and returns the result to I/O.

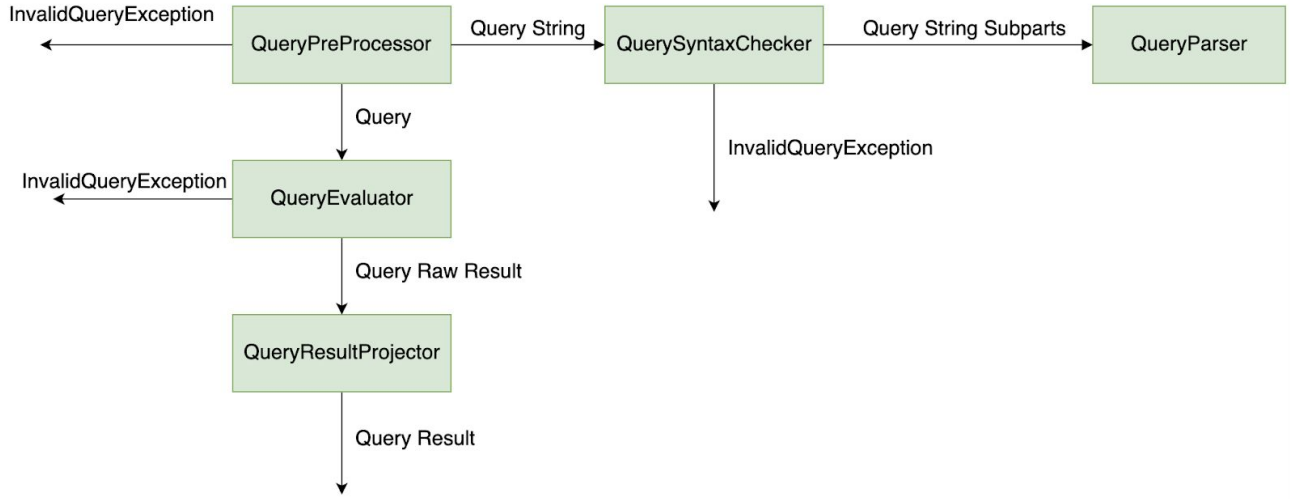

Fig. 3.4: Production Pipeline for Query Processor

3.4.1. Subcomponent 1: Query Preprocessor

Query Preprocessor (QPP) packages an input query string into a Query object. The class diagram of Query is shown below:

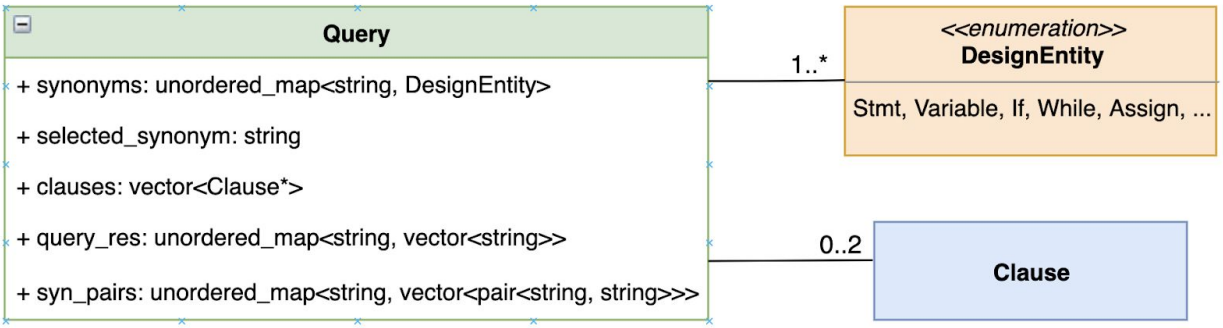

Fig. 3.4.1-1: Class diagram for Query

As shown in the Class diagram above, Query has five fields:

1. **synonyms**: stores mappings of synonyms and their corresponding design entities that are declared in the declaration part of query
2. **selected_synonym**: the synonym that is to be selected and returned in results
3. **clauses**: relational clause and pattern clause that appear in the query. There can only be 3 sizes of clauses: 0 (no clause), 1 (one relational clause or one pattern clause), or 2 (one relational clause and one pattern clause).
4. **query_res**: stores mappings of synonyms and their corresponding results.
    a. The synonyms stored in query_res include: i) selected synonym; ii) any common synonyms that appear in both relational and pattern clauses.
    b. The result for each synonym is initialized as an empty vector in QPP, and will be filled along the evaluation process.
5. **syn_pairs**: stores mappings of synonym pairs to their corresponding result pairs.
    a. synonym pairs: a pair of synonyms that appear together within a relational or pattern clause. For example, in *Uses (s, v)*, *s* and *v* are a synonym pair; in *pattern a(v, _)*, *a* and *v* are a synonym pair. For ease of implementation, the pair is represented using a string separated by ",", for example, *"a,v".*
    b. result pairs: the corresponding result pairs for each synonym pair. For example, the result pairs for "*s,v*" could be *<1, "x">, <2, "y">, <3, "z">*.

Apart from packaging Query, QPP also performs syntactic and semantic checks on the query. It calls QuerySyntaxChecker to perform the following types of syntactic checks:

1. The overall query structure: having at least one declaration, followed by a semicolon and keyword "Select"
2. The detailed syntax of subparts: declaration, select, relational clause, pattern clause

3. No irrelevant characters between subparts

QPP performs the following semantic checks by itself:

1. Check that the synonyms used in the query have been declared
2. Check that there are no duplicate synonyms

Besides, QPP calls `Clause::validate` to perform additional semantic checks that are specific to each type of Clause: check that the left and right parameters are of the correct type. For example, the left parameter of Uses cannot be a variable or wildcard.

A sample query along with an Activity Diagram of QPP, including input query string validation and `Query` object construction, is illustrated below:

Sample query: "*assign a; while w; variable v; Select a such that Uses(w,v) pattern a(v,_)"*

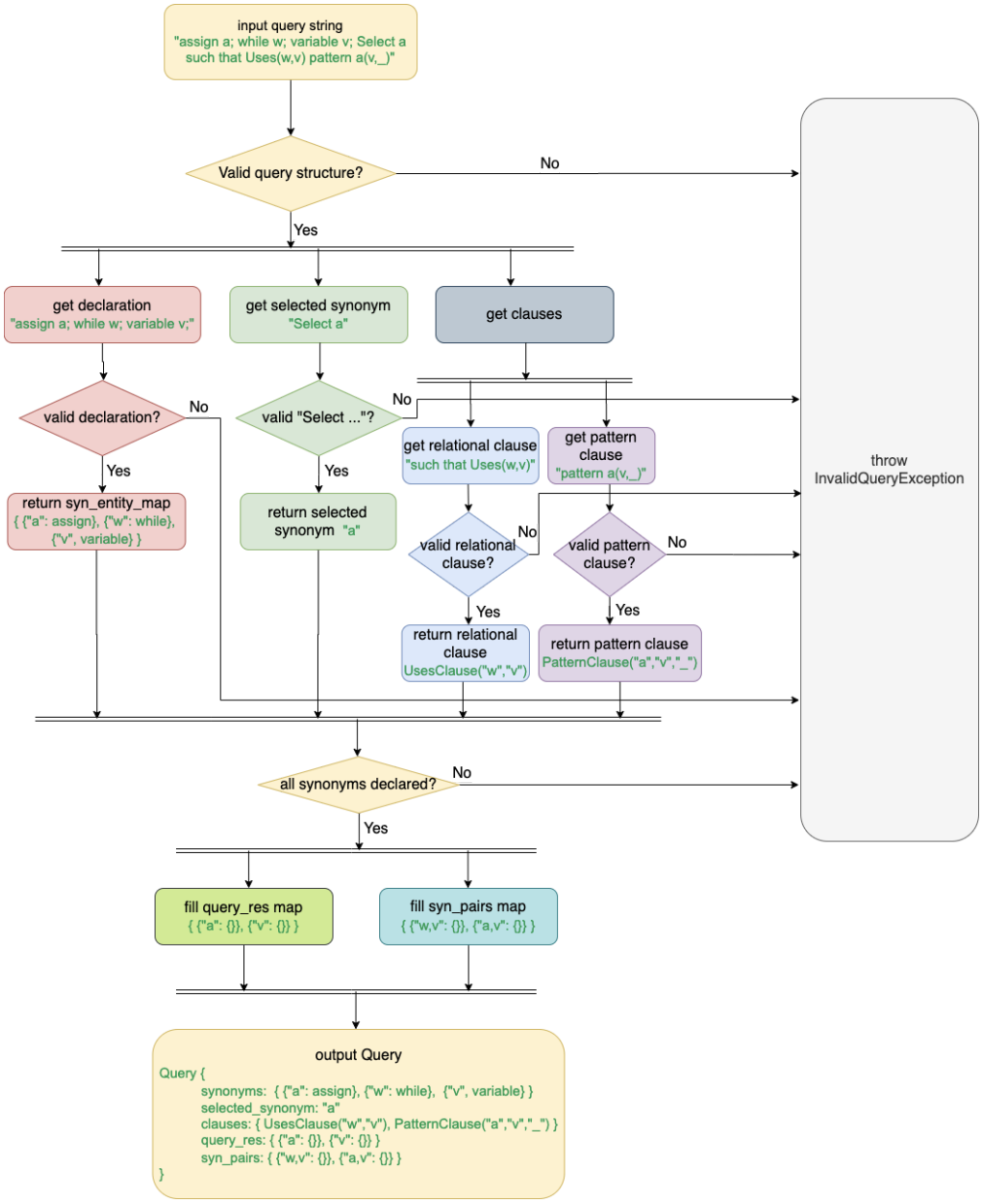

Fig. 3.4.1-2: The process of query string validation & `Query` object construction in QPP

Design Consideration: Query Syntax Validation

Query syntax validation is a check done in QPP. It checks for syntactically wrong queries, and throws `InvalidQueryException` with a detailed message about the error. There are two alternatives of implementing query syntax validation in QPP.

(1) Chosen Decision: Regex matching

Regex matching is to use a regex representing valid syntax to match the input string, and raise an exception if no matching substring is found.

Here are some examples of regexes we have written for declaration, select, and Follows clause:

```
const string REGEX_DECL =
R"((\s*(stmt|read|print|call|while|if|assign|variable|constant|proced
ure)\s+[A-Za-z][A-Za-z0-9]*\s*(\s*,\s*[A-Za-z][A-Za-z0-9]*)*\s*;\s*)+
)";

const string REGEX_SELECT = R"(Select\s+[A-Za-z][A-Za-z0-9]*)";

const string REGEX_FOLLOWS =
R"(\s*Follows\s*\(\s*(\s*[A-Za-z][A-Za-z0-9]*|_|[0-9]+)\s*,\s*([A-Za-
z][A-Za-z0-9]*|_|[0-9]+)\s*\))";
```

Fig. 3.4.1-3: Example of inserting a parent relationship

Pros: Regex matching is cleaner, and easier to debug. If problems are encountered, we only need to modify the regex and there is no need to go into the code. It is also more suitable for extension. For instance, to accommodate a new clause, we will only need to add a new regex for it.

Cons: It is difficult to come up with a comprehensive regex that covers all possible valid syntaxes. With many optional subparts included, the regex could be very long.

(2) Alternative: Tokenizer

Another way of implementing syntax check is to use a tokenizer to parse the string. Similar to a buffered reader, it could store tokens already parsed until it encounters a key breakpoint (such as semicolon or “Select”), after which it processes all the existing tokens and stores them in a suitable data structure. If it encounters invalid characters while parsing, it raises an exception immediately.

Pros: Tokenizer is faster in detecting an invalid query, because it is able to raise an exception the moment it encounters an error, instead of waiting for the regex checker to finish searching the whole string. It also performs better in corner cases that are unable to be covered by regex.

Cons: Tokenizer is harder to debug than Regex matching as it involves large chunks of code. When a problem occurs, we need to look closer into the code, and some logical mistakes are hard to find. It is also more difficult to extend, because to accommodate a new clause, potentially many parts of the code will need to be refactored.

Justification for choosing Regex Matching

Regex matching is chosen for three main reasons. Firstly, the PQL grammar provided on wiki is very concrete and largely follows regex notations. Instead of coming up with regexes from scratch (if we are given abstract requirements described in English), we only need to analyse, compare and integrate various grammar rules, so it saves a lot of effort.

Secondly, as the SPA prototype's correctness is of uttermost concern, a lot of debugging is involved in Iteration 1. Regexes, once written, will save a huge amount of time in debugging later. Given the tight schedule for prototype implementation, regex matching is a more suitable choice.

Thirdly, regex matching follows many good software engineering principles. It supports better abstraction, because regexes are declared as constants, and to use different types of regexes does not require the caller function to be altered. It also demonstrates Open Close Principle, because when a new grammar rule needs to be included, it is easy to extend the current functionality by just modifying or adding regex constants, without touching the internal code logic. Considering that there will be more new grammar rules coming in future iterations of the project, we find extendability crucial to our implementation, and have thus chosen regex matching as the preferred method.

Exceptions

An `InvalidQueryException` is thrown when the input query string does not follow PQL grammar (syntactically wrong), or has semantic errors. A message will be printed to the output channel indicating the exception type. This will inform the user when it is unable to continue evaluating the query. For example, given the following query:

```
print p; procedure p;
Select p such that Uses(p, “x”)
```

Fig. 3.4.1-4: Sample query

The Tokenizer detects duplicate synonyms “p” in the declaration.
An `InvalidQueryException` would be thrown with the following message:

```
InvalidQueryException: Duplicate synonyms in query.
```

Fig. 3.4.1-5: Sample Exception

The different situations where an `InvalidQueryException` would be thrown, and the corresponding exception messages are documented below:

Invalid Query Type	Example	Exception Message
Invalid query structure, e.g. no “Select” keyword, no semicolons, etc.	“stmt s; blablabla such that” “stmt s Select s”	"Invalid Query Structure"
The selected synonym has not been declared	“stmt s; Select v”	"Selected synonym not declared."
A synonym used in clauses has not been declared	“stmt s: Select s such that Follows (s, p)”	"Synonym used in clauses not declared."
Duplicate synonyms appear in the query	“print p; procedure p; Select p”	"Duplicate synonyms in query."
The relational clause is invalid	“assign a; variable v; Select a suxx that blablabla”	"Invalid rel clause."
The relational clause has wrong syntax (i.e. not following grammar rules of Follows/Follows*/Parent/Parent* /Modifies/Uses)	“assign a; variable v; Select a such that xcfdsf(odasfa)”	"Rel clause syntax is wrong."
The pattern clause has wrong syntax	“assign a; variable v; Select a pattern sdfasdff(dsaf3erewt)”	"Pattern clause syntax is wrong."
The parameter types are invalid for a specific clause <XXXClause>	“assign a; variable v; Select a such that Follows(a,v)”	"Invalid param type for Follows clause."

Fig. 3.4.1-6: Different Types of Invalid Queries and Exception Messages

3.4.2. Subcomponent 2: Query Evaluator

Overview

After successfully getting a Query object from Query Preprocessor, the SPA Controller calls the Query Evaluator (QE) and passes in the Query object for evaluation. The QE looks into the list of clauses in the query and calls evaluate() function provided by each Clause object and passes in the pointer of the query object. The involved clauses then analyse the given parameters and obtain the relevant results from the relationships stored in the PKB. At the end of the evaluate() function, each clause puts the results of selected synonym and common synonyms into query_res map and updates the valid pair answers in syn_pairs map.

The following sequence diagram shows the overall logic in QE.

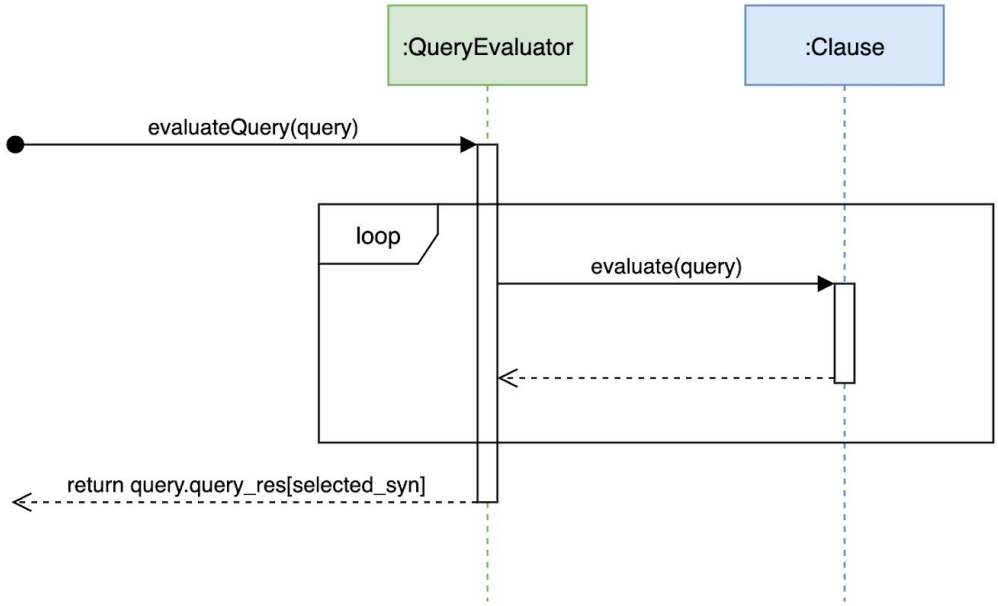

Fig 3.4.2-1: QE Sequence Diagram

Design Considerations: Evaluating Different Clauses

No two clauses evaluate queries the same way, since they have to obtain their respective relationships from different tables. However, there exist similarities across certain clauses. This brings up the possibility of the categorisation of clauses to certain types. We therefore evaluate two solutions based on the principles of Object Oriented Programming to determine whether it would be more beneficial to have standalone clauses, or clauses that are grouped into categories.

(1) Chosen Decision: Separation into Relational and Pattern Clauses

We separated the clauses into two main categories - Relational and Pattern clauses. This was based on the understanding that relational clauses, encompassing Follows, Follows*, Parent,

Parent*, Uses and Modifies, are similar in their methods of evaluation. Furthermore, they are all part of the ‘such that’ clause of a query.

Pros: Such a categorisation would allow for methods specific to categories to be used by clauses belonging to that category. There is a better Separation of Concerns (SoC) between relational and pattern clauses and this leads to higher cohesion within each category. This has allowed our two team members to easily work on separate categories without having to modify areas from the other clause type

Cons: If there are no methods common to relational clauses, categorising them may not be effective.

(2) Alternative: Standalone Clauses
The alternative solution is to have all the clauses as standalone clauses.

Pros: This solution is easier to implement at the beginning as there are fewer things to think about.

Cons: Implementing standalone clauses results in excessive separation of concerns that may lead to unnecessary work done. It may potentially be more tedious to deal with ripple effects and there is a wasted opportunity for better Separation of Concerns to be executed.

Justification for choosing Separation into Relational and Pattern Clauses
If we were to choose to code standalone clauses, several issues may arise in the future. For example, future developers may get the initial impression that all the clauses are very different, delegating clauses to different developers to work on. Similar clauses may then be programmed very differently when similar methods can be shared. If a change that affects the clauses were to be made, the ripple effects would be far greater and many different changes need to be made since the clauses are programmed differently. Therefore, separation into relational and pattern clauses is the far superior option.

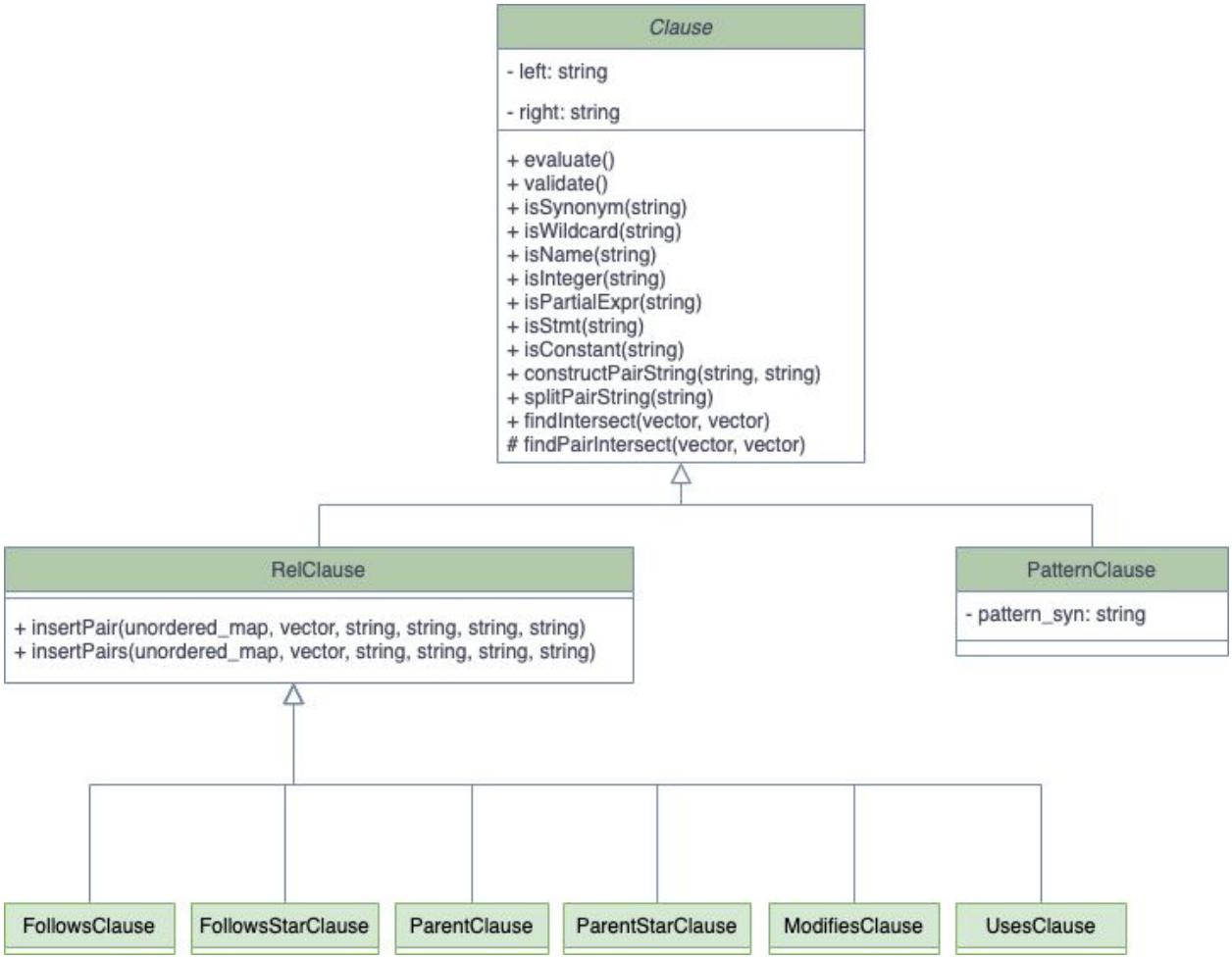

Fig 3.4.2-2: Clause Class Diagram

Clause

This is the parent class that the Relational Clauses and Pattern Clause classes inherit from. Each clause has at least 2 attributes: left and right parameters. Both parameters are stored as strings in a clause object. It contains utility methods that are common to all clauses, such as the regular expression checks to determine the Design Entity type of a received parameter.

Relational Clause

This class inherits from Clause, and it is the class that all the relational clauses - Follows, Follows*, Parent, Parent*, Uses and Modifies, inherit from. It also contains methods that are common among relational clauses.

Follows Clause

For explanation purposes, we will use the Follows clause as an example. All the other clauses are similar in terms of structure. This clause accounts for all the Follows clauses, and interacts

with the PKB to obtain results regarding Follows relationships. The evaluate function in this class accounts for the all possible combinations that are valid for the Follows clause. This is illustrated in the figure below:

LHS	RHS	Example
Integer	Integer	Follows(3,4)
Integer	Wildcard	Follows(3,_)
Integer	Synonym	Follows(3,s)
Wildcard	Integer	Follows(_,4)
Wildcard	Synonym	Follows(_,a)
Wildcard	Wildcard	Follows(_,_)
Synonym	Integer	Follows(w,9)
Synonym	Wildcard	Follows(s,_)
Synonym	Synonym	Follows(a,a1)

Fig. 3.4.2-3: Combinations of Parameters in Follows Clause

Each case is dealt with accordingly by making the necessary calls to the PKB to obtain the relevant relationships. The similar idea is repeated for all relational clauses, with Uses and Modifies clauses having different combinations of cases from the above table.

The following flowchart provides an example of the logic behind the execution of the Follows clause when given a query.

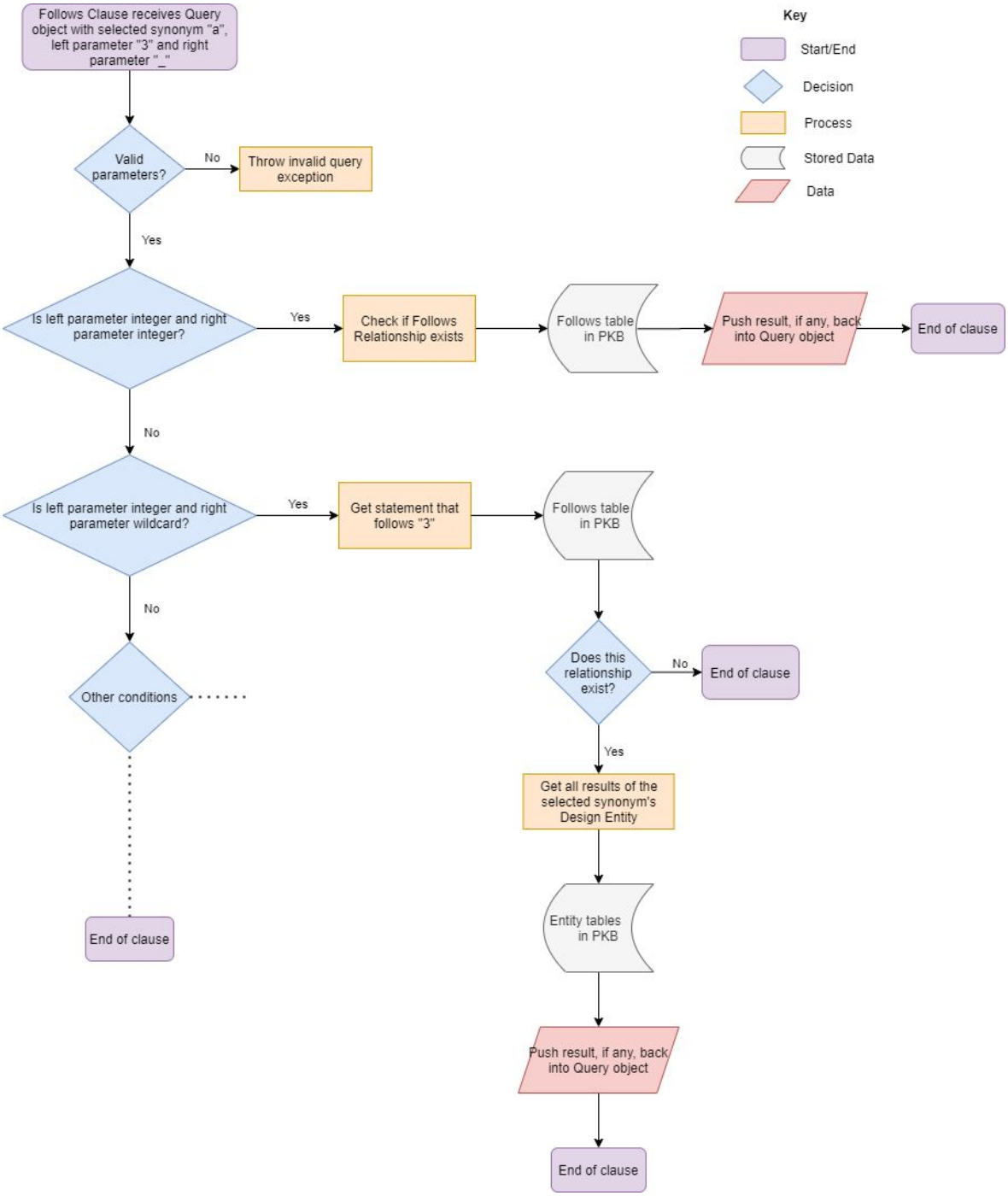

Fig 3.4.2-4: Follows evaluate() Activity Diagram

Pattern Clause

This class inherits from Clause and has another attribute: pattern_syn, which is the assign synonym in a pattern clause. Similar to RelClause, Pattern clause evaluates based on the type of parameters, checks the synonym to be selected and calls the PKB APIs accordingly. The

activity diagram shows how PatternClause evaluates the clause "*pattern a (v, _"x"_)*" when the pattern clause can be evaluated independently.

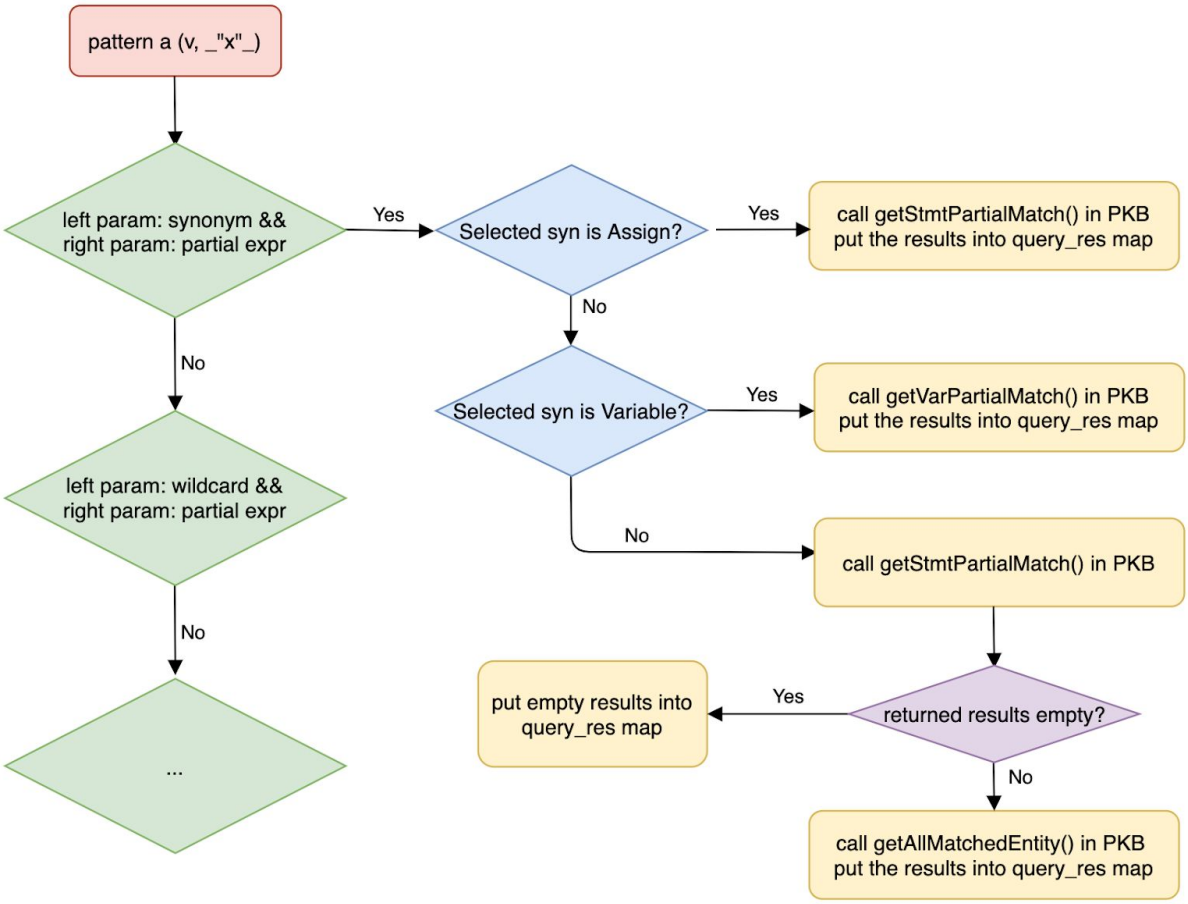

Fig 3.4.2-5: Pattern evaluate() Activity Diagram - single clause

In the future when there is pattern matching for while and if, we plan to make the PatternClause a base clause and have three subclasses - PatternAssign, PatternWhile, and PatternIf to inherit from it.

Design Considerations: Merging results from multiple clauses

In the previous section, we discussed how RelClause and PatternClause help to evaluate the clauses independently, i.e. without knowing the results from other clauses. In the case when there is no clause or only 1 clause, the QE will just call evaluate() functions and the clause will update the query_res map. So there is no need to merge the results for these two cases.

However, when there are two clauses, there are common synonyms between 2 clauses. For example, *"Select a such that Follows*(w, a) pattern a (v, _)"*. Therefore, we should only output the results which satisfy both clauses.

(1) Chosen Decision: Let each clause have the knowledge about the results from the other clause

Query results are stored in the query object and each clause evaluates based on the current valid results and updates the results after evaluation.

For queries with 2 clauses, RelClause will always be evaluated before PatternClause. This sequence is ensured by QPP at the time when it constructs a Query object, it always inserts RelClause into the Clauses vector before PatternClause. Since the RelClause always evaluates first, it is in charge of inserting the results of selected synonym and common synonyms into query_res map, and updating the valid answer pairs in syn_pairs map *(query_res and syn_pairs are two attributes inside the Query object)*.

On the other hand, PatternClause evaluates independently if there is the only clause in this query, or there are no common synonyms between 2 clauses. If there are common synonyms, PatternClause will get the results that satisfy the pattern clause first. Then, it examines the current valid results in query_res and current valid answer pairs in syn_pair, and discards the results that do not satisfy the pattern clause by finding the intersection between the vectors stored in the maps and the vector of results of the pattern clause. After updating the two maps, the QE can retrieve the result of the selected synonym from the query_res map.

The merging method is similar for 1 common synonym and 2 common synonyms. The example below shows how the sample query is being processed in QE and corresponding Clause classes. The sample query is the same as the query processed by QPP in Fig 3.4.1-2. This sample query is one of the most complex query that will appear for iteration 1.

```
procedure main {
    while (a > 1) { //1
        b = b; //2
        c = d; //3
    }
}
```

Fig 3.4.2-6: Sample SIMPLE Source Code

Sample query: *"assign a; while w; variable v; Select a such that Uses(w,v) pattern a(v,_)"*

At the beginning, the query_res and syn_pairs maps look like the following. The query_res stores the selected synonym “a”, and the common synonym “v”. The syn_pairs stores two pairs of synonyms, “w, v” and “a,v”.

query_res {	syn_pairs {
{ “a” : { } },	{ “w,v” : { } },
{ “v” : { } }	{ “a,v” : { } }
}	}

After UsesClause evaluates and updates the two maps, they look like the following:

query_res {
{ “a” : { “2” , “3”} },
{ “v” : { “a”, “b”, “d” } }
}
	syn_pairs {
	{ “w,v” : {<1,“a”>, <1,“b”>, <1,“d”> } },
	{ “a,v” : { } }
	}

When PatternClause receives the query, it follows the below activity diagram to update the maps as shown below.

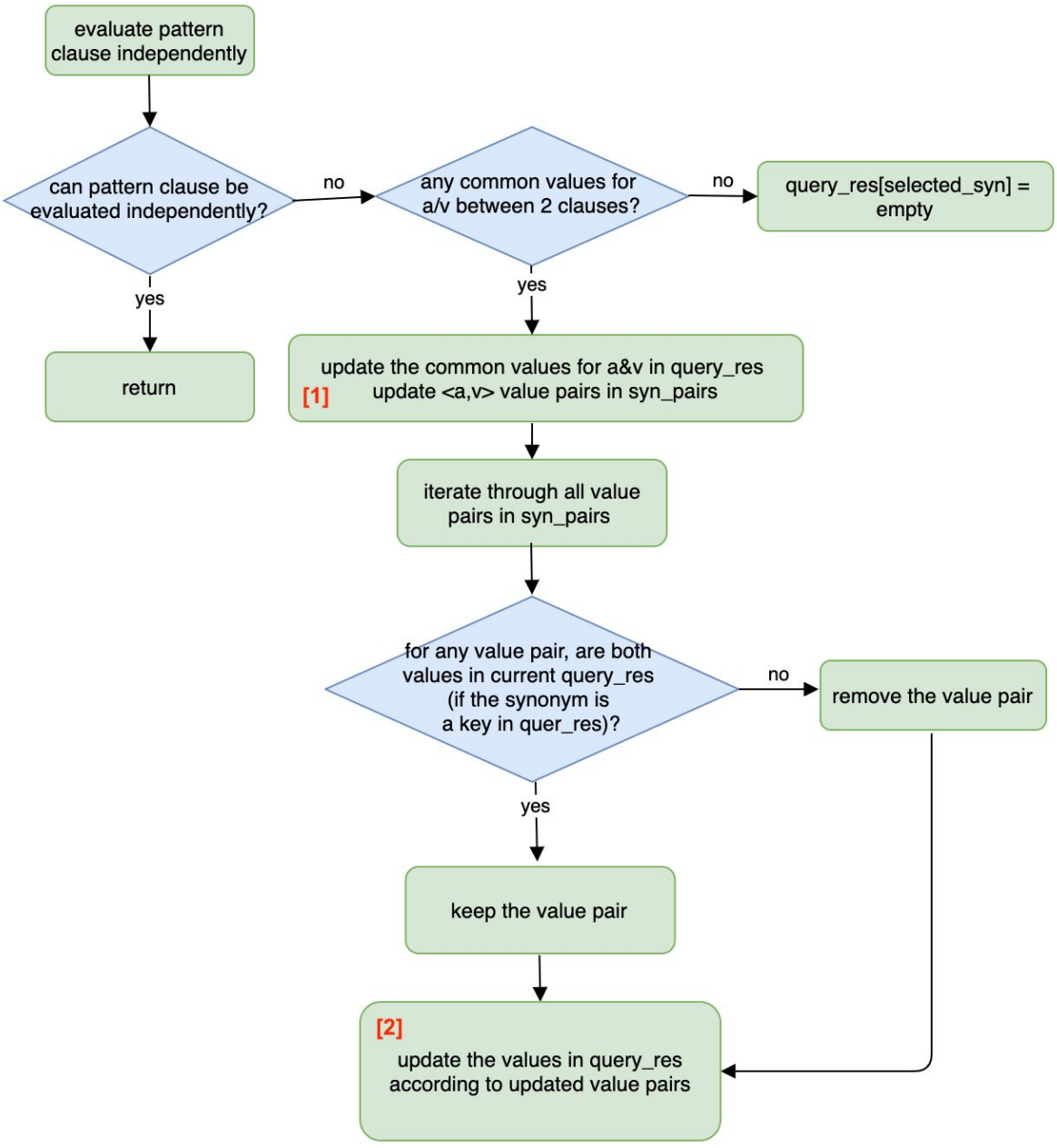

Fig 3.4.2-7: Pattern evaluate() Activity Diagram - 2 clauses

After performing the step at [1] in the Fig.3.4.2-7 above, the maps have the following contents:

query_res {
{ “a” : { “2” } },
{ “v” : { “b” } }
}
	syn_pairs {
	{ “w,v” : { <1,“a”>, <1,“b”>, <1,“d”> } },
	{ “a,v” : { <2,“b”>, <3,“c”> } }
	}

After performing the step at [2] in the Fig.3.4.2-7 above, the maps have the following contents:

query_res {
{ “a” : { “2” } },
{ “v” : { “b” } }
}
	syn_pairs {
	{ “w,v” : { <1,“b”> } },
	{ “a,v” : { <2,“b”>, <3,“c”> } }
	}

Lastly, QE obtains the final result by retrieving the list of values in query_res with selected synonym name as the key.

Pros:
The implementation has better time complexity. The searching can be faster since RelClause is only in charge of inserting to the maps and PatternClause is only in charge of updating the maps. In terms of space complexity, we are updating the original vectors in query_res and syn_pair maps instead of adding new vectors into the maps. It is also easier to implement at iteration 1 stage, and fulfills the requirements for at most 1 relational clause and 1 pattern clause.

Cons:
Worse extendability - May not be extendable when there are more multiple clauses in a single query.

(2) Alternative: QE merge the results after each clause evaluates independently
Each clause evaluates independently, and inserts the results of its own clause to query_res and syn_pair maps. QE waits for all clauses to finish evaluation and does the merging outside the Clause classes.

Pros:
It has better separation of concerns since each clause is only in charge of evaluating its own clause without the need of having knowledge of others. It also has better extendability. When adding new types of clauses into SPA, the merging algorithm in QE does not need to change.

Cons:
This approach is harder to implement. Logic can be much more complex when merging from

several different types of clauses. It might also result in worse space complexity and time complexity, since each clause evaluates independently, it needs more space to store the temporary results and there might be time wasted in evaluating clauses that need not to be evaluated if the clause has knowledge about the results from other clauses.

Justification for choosing to Merge Results from Multiple Clauses

In the scope of iteration 1, we think the former implementation fulfills our requirement and is easier to implement. Due to the time constraint, we chose to go with the less complex implementation with high accuracy. However, we are likely to switch to the alternative implementation in the future iterations since it seems to have better performance when the number of clauses increases and the merging algorithm becomes more complex.

3.4.3. Subcomponent 3: Query Result Projector

In iteration 1, there is not much work to be done in QRP. Since the Query Evaluator returns a vector of strings of the correct results, QRP merely helps to copy the results from the vector to the list pointer given by SPA Controller.

3.4.4. Bonus Features in Query Processor

(1) Swapping of "such that" and "pattern" clauses

In iteration 1 requirement, the select clause looks like the following:

select-cl: declaration* '**Select**' synonym [suchthat-cl] [pattern-cl]

We extended the scope to allow for [pattern-cl] to come before [suchthat-cl]. If the rest of the syntax is correct, the query is also considered valid and will be evaluated accordingly.

(2) UsesP and ModifiesP

In Iteration 1 requirements, Modifies and Uses clauses can only evaluate stmtRef.

ModifiesS: 'Modifies' '(' stmtRef ',' entRef ')' UsesS: 'Uses' '(' stmtRef ',' entRef ')'

We extended the scope to allow Modifies and Uses clauses to be able to evaluate the relationship between procedure and variable. For example, the following queries are also considered valid and will be evaluated accordingly.

procedure p; Select p such that Uses(p, "x")
variable v; Select v such that Modifies("Sun", v) // where Sun is a procedure name

(3) Full Pattern Match of var_name and const_value

In Iteration 1 requirements, the second parameter in the pattern clause can only be "_" or _"factor"_. That means the SPA can only evaluate _ or partial expressions on the RHS.

pattern-cl: **'pattern'** syn-assign '('entRef ',' expression-spec ')' expression-spec: '_' '"' factor '"' '_' | '_'

We extended the scope to allow full pattern match of factor on the RHS to be evaluated as well. That means, our grammar rule is as following:

pattern-cl: **'pattern'** syn-assign '('entRef ',' expression-spec ')' expression-spec: '_' '"' factor '"' '_' | '_' | "factor"

Hence, the following query is considered as valid and will be evaluated accordingly:

assign a; Select a pattern a (_, "x")
This query returns all assign statements that have only "x" on the RHS.

assign a; Select v pattern a (_, "3")
This query returns all assign statements that have only "3" on the RHS.

3.5. Component Interactions

Sequence Diagrams

Sequence diagrams were used during the project planning stage to identify interactions between components and discover sub-components. The high level SPA Sequence diagram, depicted in Fig 3.1-3, allowed us to API calls between different components. Subsequently, high-level sequence diagrams were created for the main components as well. It allowed us to better understand the flow of the program, and what should be the parameters or returned results between different components. These diagrams were also useful during the testing phase, as they gave us a good overview of crucial points of interaction between components that would need to be thoroughly tested.

Class Diagrams

Class diagrams were used in the component design planning stage to assist with data structure design. Examples of a class diagram is the AST class diagram, depicted in Fig. 3.2-2, and the Clauses class diagram, depicted in Fig. 3.4.2-2.

The class diagram served as a useful and easy-to-read visual representation of the classes required for implementation. This was especially useful for complex structures such as the AST, which comprises 17 different classes and various inheritance levels. It could be shared to other team members, who were able to provide feedback to streamline the design quality.

The class diagram was also useful during the development stage. It gave the developer a useful overview of the classes that needed to be implemented, and the requirements for each class. This allowed for a smoother implementation process with fewer mistakes made.

Architecture Component Diagrams

Component diagrams such as the one seen in Fig. 3.1-1 were extremely helpful during the infancy stages of our project, way before we started coding. A more bare version was drawn during our very first project meeting as a high level illustration to ensure that every team member understands the interactions between the various components.

Subsequently, we further developed the bare version of the diagram, making changes such as the migration of the Design Extractor to the PKB from Front-end, and eventually culminated in

the final overview diagram that is Fig 3.1-1. This diagram was presented to our TA during our first consultation and formed the bedrock of our project that our TA would occasionally refer back to during subsequent consultation sessions. This shows that component diagrams are extremely useful when it comes to presenting solutions to non-developers of the project as it provides an excellent summary of the project's main components and interactions between them.

Activity Diagram

Activity diagrams such as Fig 3.4.2-5 helps in the development stage. Since the logic flows in the PQL, especially QPP and QE, are complicated, when developers of these components communicate with each other, the diagram could illustrate the flow of checking and evaluation better than text or verbal.

4. Documentation and Coding Standards

4.1. Naming Conventions

Our naming conventions for APIs are as follows:

API Naming Convention	
Getter methods	Prefixed with ‘get’: “getStatementList()”
Methods checking for conditionals	Prefixed with ‘is’: “isModifies()”
Other Methods	Written in upper camel casing: “tokenizeSource()”
Abstract Classes	Written with uppercase separated by underscores: “STMT_LIST”
Arguments	Written in lower casing separated by underscores: “stmt_lst”

Fig. 4.1: An overview of API Naming Conventions

4.2. Coding Standards

Our coding standards are as follows:

Naming Convention	
Variables	Initialized with lower casing separated by underscores: “variable_name”
Methods	Initialized with camel casing: “methodName()”
Class	Initialized with upper camel casing : “ClassName”
Constant	Initialized with upper casing separated by underscore : “CONSTANT_NAME”
Enum	Initialized with upper camel casing: (“EnumName”)
Format Standardization	
Declarations	The .ccp file should only declare an include to its corresponding header file. All declarations of includes and namespaces should only be in header files.
Declaration Order	The sequence of declaration types should be as follows: - Include header files (include “header.h”) - Include std libraries - Namespace declaration
Code Cleanup	On CLion, command+option+shift L should be used to clean up code.
Coding Conventions	
auto	Use type inference when declaring object whenever possible for easier readability

pointers	Use pointers and smart pointers whenever possible to reduce the chance of memory leaks

Fig. 4.2: An overview of coding standards

4.3. Abstract to Concrete API

We have enhanced correspondence between Abstract and Concrete API as follows:

- Any Abstract API classes whose name contains the word “LIST” will be implemented as a vector.
- Any Abstract API classes whose name contains the word “NAME” will be implemented as a string
- Any Abstract API classes whose name contains the word “NO” will be implemented as an integer
- Whenever possible, abstract API and concrete API class names should be the same.

5. Testing

5.1. Test Plan

Our project is a test driven development. When an activity such as a subcomponent implementation has been completed, we run the test. If the test passes, development continues as well as adding more tests for better test coverage or running the next testing strategy. If the test fails, bug fixes will be done and the running of tests cycle continues.

Week	Testing Activities	Rationale
3	Unit testing	Work on components and test incrementally. Easier to fix bugs coming from individual component
4	Unit testing Integration testing	Continuous development of individual components which require adding more unit test cases. Most components should be ready for integration and integration testing is done to ensure interaction between components are working well.
5	Integration testing	All components are integrated by week 5. Integration testing ensures all components are interacting well with each other.
6	System Testing	Validates the fully integrated components. Evaluates the whole system's behaviour by giving a source and query file and analysing the displayed results.
7	Regression Testing System Testing	Any failed system test cases indicate logic flaws, do not meet functional requirements and will be fixed. System testing is done again to ensure the whole system meets functional requirements. Regression Testing on the unit and integration testing is done as there were changes made to the logic and should be tested to ensure each component works and has a stable interaction between components.

Fig. 5.1: An overview of Testing Activities and the rationale

5.1.1. Test Plan in Details

(1) Unit Testing

In weeks 3 and 4, unit testing was done along with the development of individual components to ensure correct logic implementation in the individual component. Specifically, there are unit tests for the components such as SPA Controller, Tokenizer, Parser, AST API, Design Extractor, PKB Storage, Query Preprocessor, Query Evaluator and Query Result Projector. For this unit testing stage, each team member incharge of their own components creates their own

`.cpp` test file under `unit_testing > src` directory to test specifically on the functionality of the component. The unit test cases cover both valid and invalid test cases. When bugs are detected, the respective team member will have to make improvements to their implementation logic.

(2) Integration Testing

In weeks 4 and 5, when most of the components have gone through unit testing and are relatively stable, integration testing is carried out to ensure different components can interact with each other properly. Specifically, the integration testing details for Parser and PKB integration testing as well as PKB and PQL integration testing can be found under Section 5.3.1 and 5.3.2.

(3) System Testing

In week 4, system test cases have been carefully planned and designed to cover the SPA requirements. This is especially important because it determines if our SPA system is properly integrated and does not have any logic flaws or bugs that will break our system. The system testing is done using Autotester which takes in the source and query inputs and generates a xml file where we can view the query results.

Queries Design

For each of the no clause, such that clause, pattern clause as well as both such that and pattern clauses, all the combinations of arguments are written down. Within each argument, all the possible combinations of input are written down as well. They are then classified into valid and invalid cases. Valid cases are queries that will return an answer. Invalid cases include those queries that cannot be satisfied due to no result found, semantically invalid and syntactically invalid. For example, when designing the Follows query, the grammar rules states that

```
Follows: (stmtRef , stmtRef)
stmtRef: syn / _ / INTEGER
```

Similarly, as illustrated in Fig. 3.4.2-3, the Follows could have different argument combinations generated for testing of the Follows queries.

In each argument, there can be `syn` of read, print, while, if, assign, variable, constant, procedure, stmt used and permuted. To be more specific, any combinations that use variable or constant as the `syn` can be classified as invalid cases as variable and constant do not satisfy `stmtRef`. The rest can be classified as valid cases if such statements exist in the respective source code. In addition, `INTEGER` does not only need to satisfy the clause but can be intentionally designed to cover invalid cases. For example, an out of range INTEGER can test if the our SPA system handles such cases and correctly return an expected empty result.

Hence, the queries are designed with the following in mind

- Different possible arguments combinations
- Different input combinations within each arguments
- Classify them into valid or invalid test cases
- Focuses on completely invalid queries causes spaces, spelling, syntax mistakes

Source Design

The sources are designed to cater specifically to the clauses as well as designing the source in a more complicated way to capture the edge cases. For instance, there are a few considerations when designing the source code in order to capture the complicated cases of the query relations:

- Complex conditional expression
- Position of the loops
- Nested loops
- Variable names that are similar to the name of design entities
- Simple Source caters to Select, such that clause, pattern clause as well as both clauses
- An invalid source that does not satisfy Grammar Rules (even though there is not query to evaluate, it is important that our system handles invalid source correctly)

With those in mind, the source codes are designed with an intention in mind and they are documented in the figure below.

File	Testing Intention
no_clause_source.txt no_clause_queries.txt	Refer to Section 5.4.1, Section 5.4.5
Follows_Parent_source.txt Follows_Parent_queries.txt	Refer to Section 5.4.2, Section 5.4.5
pattern_ModifiesS_UsesS_source.txt pattern_ModifiesS_UsesS_queries.txt	Refer to Section 5.4.3, Section 5.4.5
pattern_suchthat_source.txt queries_pattern_suchthat.txt	Refer to Section 5.4.4, Section 5.4.5
complex_condexpr_nested_source.txt complex_condexpr_nested_queries.txt	Refer to Section 5.4.6, Section 5.4.7, Section 5.4.5,
uncommon_invalid_source.txt uncommon_invalid_queries.txt	Refer to Section 5.4.5
invalid_test_source.txt	Refer to Section 5.4.5
bonus_3_features_souce.txt bonus_3_features_queries.txt	Refer to Section 5.4.8

Fig. 5.1.1-1: Overview of System Tests

From week 4 onwards, when the system has been fully integrated, system testing is conducted on the complete integrated system to evaluate the system's compliance with the project's functionality requirements using Autotester. For instance, with the well designed system test

design, we will run the autotester command on the terminal by passing in a source file, its respective queries file and write to a xml file for displaying the results.

(4) Scripts

In order to improve efficiency of calling the desired commands for the set of source, queries and output files that we have, a bash script was created in CLion. It consists of a series of commands that navigate to the correct folders for Autotester executor as well as the system test files that will be called. By clicking the `Run File` for `run_all_iter1_test.sh` located in `tests > Iteration_1 folder`, we can easily run 1 command to trigger multiple set of commands at once without having to run the commands multiple times. We are also able to control which set of commands to focus on run first by commenting out the line of codes.

(5) Bug Tracking

When there is any bug found, our team will raise an issue on github and add this into a Bug Tracker project which has columns of Needs Triage, High priority, Low Priority and Closed. This is to ensure that all bugs are accounted for. The team will also be aware of who is currently carrying out the bug fixes and to ensure that there is not double work.

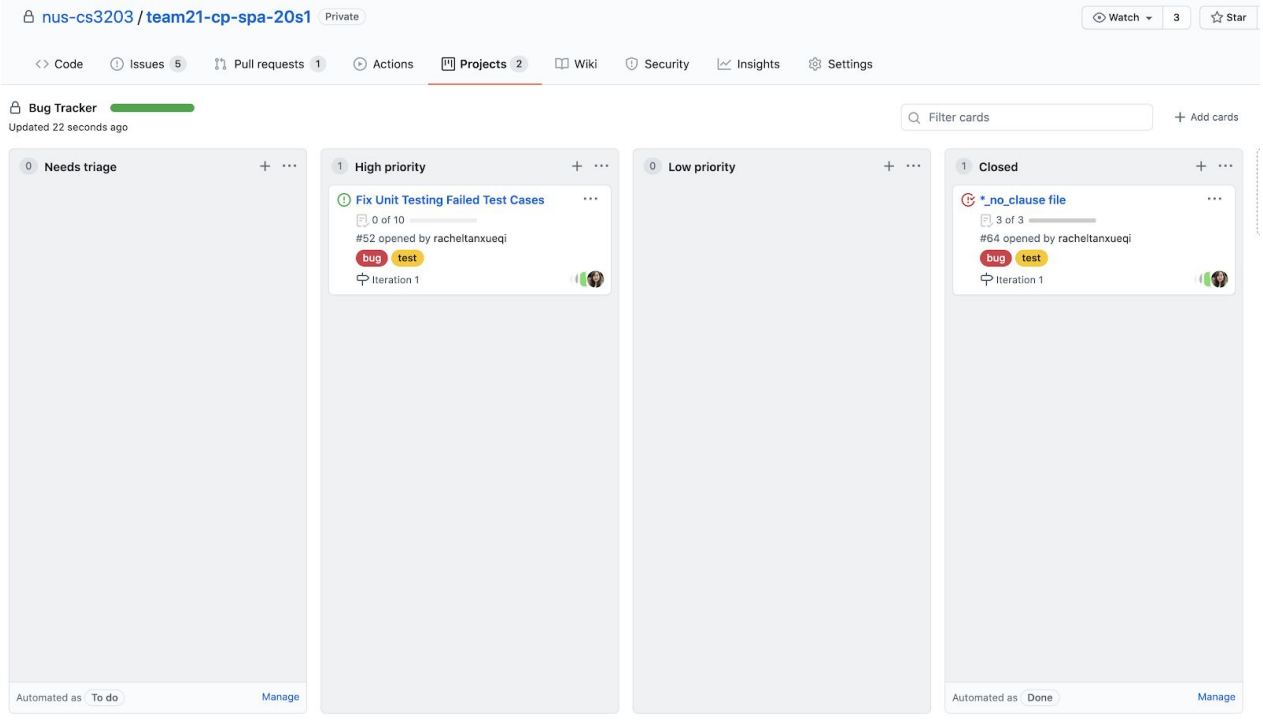

Fig. 5.1.1-2: Bug Tracking Tool

5.2. Unit Testing

5.2.1. Unit Testing: Front-End Parser

For the Front-End Parser unit testing, we test 3 aspects of the Front-End Parser: Token object creation, Tokenizer correctness, and Parser correctness.

Test Category 1: Token Object Creations

Overview

To test Token object creation, we create a set of token values and verify that the Token object created has been assigned the correct value and token type. All possible mappings between Token Type and Token value will be tested. For Token Types with an infinite number of mappings, such as “Identifier”, we will select a subset of token values that best effectively maximize test coverage. The following figure is an example of the test cases for creating Tokens of type “Identifier”.

Case	Input

Valid Test Cases	`Variable`
	`VARIABLE`
	`Var1`
	`VAR2`

Invalid Syntax: Starts with DIGIT	`1var`
Invalid Syntax: Contains invalid symbol	`var_1`

Fig. 5.2.1-1: Test cases for Token Object Creations

Two Sample Unit Test Cases

Purpose	Input	Assertion	Expected Test Results

`Valid Test Case`	`string value=”variable”`	`TOKEN token = Token::createToken(value)`
		
		`TOKEN expected = <manually constructed TOKEN object>`
		
		`REQUIRE(token == expected)`
			We manually construct the `TOKENSTREAM` object to be passed into the parser. We then traverse the tree and determine if the value of a node in a certain position is correct.

`Invalid Test Case: invalid variable name`	`string value=”1var”`	`REQUIRE(Token::createToken(value), “Invalid Lexical Token Exception! Received Invalid Lexical Token: 1var”)`	We use `REQUIRE_THROWS_WITH` to validate that an invalid token value throws an error, and compare the expected and outputted message to see if the correct message is produced.

Fig. 5.2.1-2: Sample Unit test cases for Token object creation

Test Category 2: Tokenizer

Overview

To test the Tokenizer, we pass in valid and invalid SIMPLE source strings and if the correct set of tokens are generated for a particular source.

Two Sample Unit Test Cases

Purpose	Input	Assertion	Expected Test Results
Valid Test Case	string stmt = “if()else{}”	TOKENSTREAM result = tokenizer->tokenizeSource(stmt); TOKENSTREAM expected = <manually constructed set of tokens> REQUIRE(result == expected)	We manually construct the TOKENSTREAM object containing the tokens we expect to see being output by the Tokenizer . We then compare this expected TOKENSTREAM with the actual TOKENSTREAM output result of the tokenization.
Invalid Test Case: Empty SIMPLE source	string stmt = “”	REQUIRE_THROWS_WITH(tokenizer->tokenizeSource(stmt), “Invalid Tokenizer Exception! Nothing to tokenize: expected: Non-empty or non-space source file, got: ‘’.”)	We use REQUIRE_THROWS_WITH to validate that an invalid source input throws an error, and compare the expected and outputted message to see if the correct message is produced.

Fig. 5.2.1-3: Sample Unit test cases for Tokenizer

Test Category 3: Parser

Overview

To test the Parser, we manually create sets of valid and invalid TokenStream objects to pass into the Parser. A set of test cases are constructed to test the parsing for all SIMPLE design entities required for iteration 1, as well as expression and conditional expressions.

Invalid test cases are created based on the possible types of structural violations of the SIMPLE CSG. The following figure is an example of the test cases for Assign statement parsing. Expression parsing is tested in another set of cases.

Case	Input

Valid	procedure main {
	
	x = 1+x;
	
	}

Invalid Variable	procedure main {
	
	1 = 1+x;
	
	}

Invalid Expression	procedure main {
	
	x = 1++;
	
	}

Missing semicolon	procedure main {
	
	x = 1+x
	}

Fig. 5.2.1-4: Example test cases for parsing Assign Statements

Two Sample Unit Test Cases

Purpose	**Input**	**Assertion**	**Expected Test Results**

Valid Test Case: Assign Statement	procedure main {
	
	x = 1+x;
	
	}
		TOKENSTREAM token_stream = <manually constructed token stream input>
		
		AST ast = parser->buildAST(token_stream);
		
		for each node in ast {
		string expected = <expected node value>
		REQUIRE(node->getValue() == expected)
		}
			We manually construct the TOKENSTREAM object to be passed into the parser. We then traverse the tree and determine if the value of a node in a certain position is correct.

Invalid Test Case: Invalid variable name	procedure main{
	
	1=1+x;
	
	}
		TOKENSTREAM token_stream = <manually constructed token stream input>
		
		REQUIRE_THROWS_WITH(parser->buildAST(token_stream), “Invalid Assign Statement: expected variable, got ‘1’.”)
			We use REQUIRE_THROWS_WITH to validate that an invalid TOKENSTREAM input throws an error, and compare the expected and outputted message to see if the correct message is produced.

Fig. 5.2.1-5: Sample Unit test cases for Parser

5.2.2. Unit Testing: PKB

Test Category 1: Retrieval and Storage

Overview

Each “gets” function is individually tested to ensure that they retrieve them from the correct internal table. Each storage function is tested to ensure that repeated inputs of the same value do not store the values multiple times into the same table since each value is to be unique.

Case	Input

Valid Test Cases	`insertVariable(variable)`
	`insertParent(parent, child)`
	`insertFollows(prior, latter)`
	`insertConstant(constant)`
	`insertProcedure(procedure_name)`
	`insertStatement(statement)`

Repetition	`1.insertVariable(“v”)`
	`2.insertVariable(“v”)`
	`3.getsVariable(“v”) == “v” in line 1`

Fig. 5.2.2-1: CRUD test cases for PKB

Test Category 2: Relationship Extraction

Overview

Each relationship is tested to ensure that they are correctly extracted. Follows/Parent/Uses/Modifies are each individually tested after a dummy PKB is set up to separate testing the relationship extraction from the gets/insert functions. Parent/Follows relationships are tested before the second pass is inserted to ensure that the information extracted from Parent/Follows does cause a failed test case in the functions relying on them.

Two Sample Unit Test Cases

Purpose	Input	Assertion	Expected Test Results

`Test Extraction of Follows Relationships`	`procedure main`
	`{`
	
	`x = 1 + x;`
	`x = 2 + x;`
	
	`}`
		`PKB pkb = PKB() <PKB is manually constructed by inserting values>`
		
		`updateFollows()`
		
		`REQUIRE(isFollows(1, 2) == true)`
			isFollows(1, 2) should return true

`Test Extraction of Parent Relationships`	`procedure main{`
	
	`while (1 == 1)`
	`{`
	`x = 2;`
	`}`
	`}`
		`PKB pkb = PKB() <PKB is manually constructed by inserting values>`
		
		`updateParent()`
		
		`REQUIRE(isParent(1, 2) ==`
			isParent(1, 2) should return true

		true)	

Test Extraction of Parent* Relationships	procedure main{
	
	while (1 == 1)
	{
	while (2 == 2)
	{
	x = 2;
	}
	}
	}
		PKB pkb = PKB() <PKB is manually constructed by inserting values>
		
		updateParent()
		updateParentStar()
		
		REQUIRE(isParentStar(1, 3) == true)
			isParentStar(1, 3) should return true

Fig. 5.2.2-2: Relationship test cases

5.2.3. Unit Testing: Query Processor

For the PQL unit testing, we individually test the 4 subcomponents of the Query Processor.

(1) Query Preprocessor

Test Category 1: Query Subparts Extraction

Overview

To test the various query extraction functions like `getDeclaration`, `getSelectedSynonym`, `getRelClause` and `getPatternClause`, a query string is passed in and the expected output is manually constructed and compared with the output of the function.

Two Sample Unit Test Cases

Test Purpose	Required Test Inputs	Expected Test Results and Assertion	Explanation
To test that QPP::getDeclaration outputs the correct result for a query with valid declaration	string query = "stmt s; variable v; while w; Select s";	unordered_map<string, DesignEntity> expected = { {"s",DesignEntity::Stmt}, {"v",DesignEntity::Variable},{"w",DesignEntity::While}}; unordered_map<string, DesignEntity> map = QueryPreProcessor::getInstance()->getDeclaration(query); REQUIRE(map == expected)	We pass in a sample query, and manually construct the expected output (a map of synonyms and their corresponding design entities). We then compare the result of calling QPP::getDeclaration with the expected output, and require the two maps to be equal.
To test that QPP::getRelClause throws an exception when the input query has an invalid RelClause (negative	string query = "stmt s; Select s such that Follows(-2,1)";	REQUIRE_THROWS(QueryPreProcessor::getInstance()->getRelClause(query));	We use REQUIRE_THROWS to check that an invalid RelClause (with negative stmt numbers) causes an exception to be thrown.

stmt number)			

Fig. 5.2.3-1: Sample Unit Tests for Query Subparts Extraction

Test Category 2: Construction of the Query object

Overview

To test the function `QPP::constructQuery`, a query string is passed in, and the five components of the expected output Query object (syn_entity_map, selected_synonym, clauses, query_res, syn_pairs) are manually constructed and compared with the components of the output of the function.

Two Sample Unit Test Cases

Test Purpose	Required Test Inputs	Expected Test Results and Assertion	Explanation
To test that QPP::constructQuery outputs the correct result for a syntactically and semantically valid query	string query = "while w; assign a; stmt s; Select a pattern a(_, _) such that Follows*(w,a)";	unordered_map<string,DesignEntity> syn_entity_map = { {"w",DesignEntity::While}, {"a",DesignEntity::Assign}, {"s",DesignEntity::Stmt}}; FollowsStarClause rel = FollowsStarClause("w", "a"); PatternClause pttn = PatternClause("a", "_", "_"); string selected_syn = "a"; unordered_map<string,vector<string >> query_res = {{"a", {}}}; unordered_map<string, vector<pair<string, string>>> syn_pairs = {"w,a", {}}}; Query q = QueryPreProcessor::getInstance()-> constructQuery(query); REQUIRE(syn_entity_map == q.synonyms); REQUIRE(q.clauses.size() == 2); REQUIRE(rel.equals((FollowsStarClause *) q.clauses[0])); REQUIRE(pttn.equals((PatternClause *) q.clauses[1])); REQUIRE(selected_syn == q.selected_synonym); REQUIRE(q.query_res == query_res);	We pass in a sample query, and manually construct the expected components of the Query object (syn_entity_map, selected_syn, clauses, query_res and syn_pairs). We then compare the result of calling QPP::constructQuery with the expected output, and require every component to be equal.

		`REQUIRE(q.syn_pairs == syn_pairs);`	
`To test that QPP::constructQuery throws an exception when the input query is invalid (repeated synonyms)`	`string query = "assign a; while a; variable v; Select v";`	`REQUIRE_THROWS(QueryPreProcessor::getInstance() -> constructQuery(query));`	We use `REQUIRE_THROWS` to check that an invalid query (with repeated synonyms in the declaration) causes an exception to be thrown.

Fig. 5.2.3-2: Sample Unit Tests for Query Construction

(2) Query Evaluator

All tests for QE are done in integration testing between PQL-PKB.

(3) Query Result Projector

Overview

To test whether QRP is able to copy all strings in raw_results returned by QE to the list pointer passed in by SPA Controller.

Input	Assertion	Expected Test Results

`vector<string> raw = {“1”, “2”, “3”};`
`list<string> res;`
	`formatResults(raw, res)`
	
	`res == {“1”, “2”, “3”}`
		list<string> res should contain all the results in raw results

Fig. 5.2.3-3: Sample Unit Test for Query Result Formatter

(4) Clauses

Overview

To test the `Clause::validate` function in respective clauses, a specific clause with parameters (of valid or invalid types) is constructed. A synonym-entity map is also constructed, and passed into the `validate` function of the clause. If the parameters are valid, no exception should be thrown. If at least one parameter is invalid, an exception is expected to be thrown.

Two Sample Unit Test Cases

Test Purpose	Required Test Inputs	Expected Test Results and Assertion	Explanation
`To test that Clause::validate does not throw any`	`FollowsClause clause = FollowsClause("s", "r");`	`REQUIRE_NOTHROW(clause.validate(map));`	We construct a `FollowsClause` with 2 synonym parameters “s” and “r”, and a synonym-entity map for

exception if all parameters are valid	unordered_map<string,DesignEntity> map = {{"s",DesignEntity::Stmt}, {"r",DesignEntity::Read}};		reference. From the map, since both s and r refer to statements (s is stmt and r is read), they are valid parameter types for Follows. There should be no exception thrown by `FollowsClause::validate`.
To test that Clause::validate throws an exception if at least one parameter is invalid	ModifiesClause clause = ModifiesClause("p", "7"); unordered_map<string,DesignEntity> map = {{"p",DesignEntity::Print}};	REQUIRE_THROWS(clause.validate(map));	We construct a `ModifiesClause` with 2 parameters “p” (synonym) and “7” (stmt number), and a synonym-entity map for reference. From the map, “p” refers to print. But “7” is a stmt number which cannot be the second parameter of Modifies (only variables can be used). Hence, “7” is an invalid parameter type for Modifies. An exception is expected to be thrown by `ModifiesClause::validate`.

Fig. 5.2.3-4: Sample Unit Tests for `Clause::validate`

5.3. Integration Testing

5.3.1. Integration Testing: Front-end - PKB

Integration testing between Front-End Parser and PKB tests a subsection of the SPA Program flow from the passing of a SIMPLE source string into the Tokenizer to the extraction and storing of design abstractions by the PKB. We look for any unexpected breakdown in communication as indicated by a thrown error, and check the accuracy of the information stored in the PKB.

One fairly simple SIMPLE source program containing all SIMPLE design entities and relationships is used for all tests. This is because we are only looking for communication breakdowns during integration testing. Edge case testing with more complicated sources is better suited for System testing.

Prior to each test, the environment will be initialized as follows:

- Instantiate a Parser, Tokenizer and PKB object.
- Store a Test Input as a string. The string is passed into the Tokenizer to generate a TokenStream output.
- Pass the TokenStream into the Parser to generate an AST output.
- Pass the AST into the PKB for Design Abstraction Extraction.

Two Sample Integration Test Cases

Purpose	**Source Code Snippet**	**Assertion**	**Expected Test Results**

Testing Procedure Relationships	procedure Example
	{
	...
	}
	
	procedure p {
	…
	}
	
	procedure q {
	…
	}
		vector<string> expected = {“Example”, “p”,”q”}
		
		REQUIRE(expected == pkb->getAllProcedure( ))
			In this test case, we check if the Design Extractor has correctly extracted all Procedures from the AST.
			
			We manually construct a vector of strings containing the expected names of the procedures in the SIMPLE source test input. We then use REQUIRE to compare this expected vector to the actual vector received when making a PKB API call to retrieve all Procedures.
			
			We also implicitly check if the Design Extractor is able to extract design abstractions without error.

Testing Follows Relationship	procedure p {
	...
	i = i - 1;
	//16
	d = x + 1;
	//17
		REQUIRE(pkb->isFollows(16, 17) == true);
		
		REQUIRE(pkb->isFollows(17, 18) == true);
			In this test case, we check if the Design Extractor has correctly extracted the Follows Relationships from the AST.

	z = x + m;
	//18
	...
	}
		REQUIRE(pkb->isFollows(16, 18) == false);	We make a PKB API call that returns a BOOLEAN to test if the PKB is able to accurately detect valid and invalid Follows relationships.

Fig. 5.3.1: Sample Integration test cases for Front-End Parser to PKB interaction

5.3.2. Integration Testing: PKB - Query Processor

Integration testing between PKB and Query Processor tests for any breakdown in communication between the two components. This is accomplished by testing the 'evaluate' method in the QE of the Query Processor as that is where the communication takes place, with QE requesting the relevant relationships from the information stored in the PKB.

The evaluate methods of every clause are tested with various sections catering to different conditions of the methods. In addition, each section tests for every potential scenario within that section, ensuring that the tests cover every call made to the PKB.

Two Sample Integration Test Cases

Test Setup	**Assertion**	**Expected Test Results**

query.selected_synonym = "s";
query.clauses = clauses;
query.query_res = {
{"s", {}},
{"p", {}} };
	FollowsClause("3", "4").evaluate(&query);
	
	vector<string> answer1 = { "1", "2", "3","4" };
	
	REQUIRE(query.query_res["s"] == answer1);
		In this test case, we are checking for the (integer, integer) condition in Follows clause.
		
		We first initialise the Query object with the involved synonyms containing empty results, simulating the Query object that is passed to the QE and filtered to the Follows clause. We then call the evaluate method which will populate the same Query object with the relevant Follows relationship results obtained from the PKB. Lastly, we check the results string in the Query object to confirm that it has been populated with the correct results.

Same as above	FollowsClause("1", "4").evaluate(&query); vector<string> answer2 = { }; REQUIRE(query.query_res["s"] == answer2);	Under the same section of this test case, we also check the other possible alternative pathways that the program can take, such as when the requested Follows relationship does not exist in the PKB. In such cases, we expect no result to be populated.

Fig. 5.3.2: Sample Integration Tests Between PKB and Query Processor

5.4. System Testing

5.4.1. Test Category 1: Simple Testing of Queries with No clause

In this test category, the purpose is to simply test if the queries are able to Select each of the synonyms of the design entities properly and return the correct results. Firstly, the source code is designed to contain all possible design entities; 'stmt' | 'read' | 'print' | 'while' | 'if' | 'assign' | 'variable' | 'constant' | 'procedure'. Secondly, the queries are generated with all the possible synonyms which are declared and met the Grammar definition of PQL. Finally, the test expects the whole system to return test results that pass both the valid and invalid cases.

Two Sample System Test Cases

Source Code Snippet	Required Test inputs	Expected Test Results

procedure one {
read a; // 1
read b; // 2
...
if (a==c) then { // 7
...
read z; // 11
...
}
	2 - Valid: Query return all read test
	read r;
	Select r
	1,2,11
	5000
		1,2,11

	7 - Valid: Query return all variable test variable v; Select v a,b,c,d,e,f,g,h,i,z,j 5000	a,b,c,d,e,f,g,h,i,z,j

Fig. 5.4.1: Sample test cases for test category 1

5.4.2. Test Category 2: Simple Testing of Queries with Follows, Follows*, Parent, Parents* clauses

In this test category, the purpose is to simply test if the queries Follows, Follows*, Parent and Parent* clauses return the correct results. The source is designed to cater to the clauses to produce valid results. It is designed with a low level complexity to both the source and queries as described below.

(1) Source Design Considerations

- To satisfy Follows and Follows*, the source contains few different type of design entity statements that are on the same nesting level
- To satisfy Parent and Parents*, the source has a few nested loops with if and while design entities.

(2) Queries Design Considerations

- Generated with all the possible arguments combinations to test the relationship of the clause (Fig. 3.4.2-3)
- Each argument has different variations of inputs to test the relationship of the clause
- Add variation to the Select
- Ensure statements in queries do not Follows/Follow*/Parent/Parent* follow itself
- Valid queries categorised when they satisfy the clause
- Invalid queries are categorised when they do satisfy the clauses due to no result, violate clause rules syntactically or semantically (Section 5.4.5)

Two Sample System Test Cases

Source Code Snippet	Required Test inputs	Expected Test Results

procedure Main {
read a; // 1
b = 20; // 2
print c; // 3
d = 10; // 4
read e; // 5
if (f==1) then { // 6
while(g!=f) { // 7
a = b + c; // 8
g = f + 2; // 9
}
} else {
...
}
	8 - Valid Follows: Select syn p, syn r and if in both args
	read r; if ifs;
	Select r such that Follows(r,ifs)
	5
	5000
		5

procedure Main {
...
if (a==a) then { // 14
read b; // 15
} else {
h = h - g; // 16
}
...
}
	128 - Valid Parent*: Select syn s, Concrete stmt number on LHS and syn on LHS
	stmt s;
	Select s such that Parent*(14,s)
	15,16
	5000
		15,16

Fig. 5.4.2: Sample test cases for test category 2

5.4.3. Test Category 3: Simple Testing of Queries with Modifies, Uses, pattern clauses

In this test category, the purpose is to simply test if the queries with Modifies, Uses and pattern clauses return the correct results. The source is designed to cater to the clauses to produce valid results. It is designed with a medium level complexity for both the source and queries as described below.

(1) Source Design Considerations

- Have at least 2 or more levels of nesting variations between if and while loops (Section 5.4.6)

- Each container statement is carefully positioned to have different kinds of statements either before it, after it and no statements that come before or after it (Section 5.4.6).
- Add valid but similar variable name as design entities
- To satisfy all the above 3 clauses, assignment statements are included with different term and brackets variation
- To satisfy Modifies, read statements are included
- To satisfy Uses, print statements are included

(2) Queries Design Considerations

- Add valid but queries synonym name are the same as design entities name
- Generated with all the possible arguments combinations to test the relationship of the clause (Fig. 3.4.2-3)
- Each argument has different variations of inputs to test the relationship of the clause
- Add variation to the Select
- Valid queries categorised when they satisfy the clause
- Invalid queries are categorised when they do satisfy the clauses due to no result, violate clause rules syntactically or semantically or are ambiguous (Section 5.4.5)

Two Sample System Test Cases

Source Code Snippet	Required Test inputs	Expected Test Results

procedure Medium {
read while; // 1
a = a+b-c * d +e/ f;
// 2
sum = sum * sum + x;
// 3
...
if = (while + (d +
sum) / f) - 20; //18
...
}
	17 - Valid Pattern Assign:
	Select syn v, syn v on LHS,
	Partial Pattern Match on RHS
	assign a; variable v;
	Select v pattern a(v,_"sum"_)
	sum, if
	5000
		sum, if

procedure Medium {
read while; // 1
...
read VaR2; //13
...
}
	71 - Valid Modifies: Select r,
	syn r on LHS and INDENT on RHS
	read r; print pn; while w; if
	ifs; assign a; variable v;
	procedure p; stmt s;
	Select r such that Modifies(r,
	"VaR2")
	13
	5000
		13

Fig. 5.4.3: Sample test case for test category 3

5.4.4. Test Category 4: Testing of Queries with one pattern clause and one such that clause

In this test category, the purpose is to test if the queries with one pattern clause and one such that clause return the correct results. The source is designed to cater to clauses to produce

valid results. There are queries that must share similar synonyms to ensure the relationship is established. It is designed with a medium level complexity to both the source and queries as described below.

(1) Source Design Considerations

- Contains all of the design entity statements that satisfy all types of such that and pattern clauses
- Few level of nested containers to satisfy Parent, Parent* and Uses
- Assignment statements are included with different term and brackets variation to add slight complexity
- Variables in the conditions of container statement are planned with the intention to satisfy Uses clause

(2) Queries Design Considerations

- Generate all the possible combination between pattern and all the other such that clauses (Modifies, Uses, Follows, Follows*, Parent, Parent*)
- Generated with all the possible arguments combinations to test the relationship of the clause (Section 5.4.6)
- Each argument has different variations of inputs to test the relationship of the clause
- Add variation to the Select
- Such that and pattern clauses share similar syn
- Valid queries categorised when they satisfy the clause
- Invalid queries are categorised when they do satisfy the clauses due to no result, violate clause rules syntactically or semantically or are ambiguous (Section 5.4.5)
- Structure variation of pattern and such that clause to validate QPP logic further

For example, the query all produce the same result:
Assign a; statement s;

- Select a pattern a(_,_) such that Uses(s,a)
- Select a such that Uses(s,a) pattern a(_,_)
- Select a pattern a(v,_) such that Uses(s,a)
- Select a such that Uses(s,a) pattern a(v,_) and many more

5.4.5. Test Category 5: Invalid Source and Query Test Cases

In this test category, the purpose is to ensure there is a large test coverage for both valid but invalid test cases. So far, Section 5.4.1 to 5.4.4 has described some of the system testing on valid cases. For **invalid sources**, our SPA system is expected to throw exceptions with informative description and exit the program. For **invalid queries**, our SPA system is expected to return an empty result and pass the test cases as invalid.

(1) Invalid Source Design Consideration

- Inconsistent conditional expression
- Missing separator
- No procedure used
- Tokens do not meet the grammar or lexical token rules
- Invalid assign statement (Example in Fig. 5.2.1-5)

Invalid sample source snippet	Expected assert

`procedure InvalidSource {`
` if (a == b)  {`
`  r = 1;`
` } else {`
` r = 1;`
` }`
`}`
	`Invalid If Statement: expected 'then', got '{'.`

Fig. 5.4.5: Sample test case for test category 5

(2) Invalid Query Design Consideration

- Queries that are syntactically incorrect and has invalid grammar rules
    - Double declaration of the same name
    - Extra separator, comma, missing syntax in select-cl, pattern-cl or suchthat-cl
    - Weird spaces
    - Declaration of synonym names that do not meet grammar rules
- Queries that are semantically Incorrect
    - Have '_' as the first argument for Modifies and Uses
    - The argument of a relationship is not one of the allowable types

The following are some examples of the invalid query mentioned above which written for invalid system testing:

- assign sameName; procedure sameName;
- stmt s;
  Select s such that Follows(_,_)
- assign a_1;
- variable v;
  Select v such that Modifies(_,v)
- variable v;
  Select v such that Uses(_,v)

5.4.6. Test Category 6: Valid and Complicated Source Design

In this test category, we have designed a source that is at a more complex level. The purpose of testing with a complex source is to add difficulty to the abstraction of such that clauses and pattern clauses relationships. For test category 1 to 6, the sources are designed at a rather easy complexity to keep the test focussed on ensuring PQL correctness. With a more complicated

source design, it places focus on identifying if we can correctly implement our components that meet the system requirement as well as capturing any implementation mistakes and logic flaws that could potentially cause unwanted system behaviour.

(1) Positioning of container statements

Positioning Types Source Code Snippet	Test Coverage

`if(a==b) then {`
`  while(c < d) {`
`  ...`
`  }`
`} else {`
`  ...`
`}`
`}`
	● Satisfies Parent, Parent*, Modifies, Uses, pattern
	● Does not satisfy Follows/Follows* statement

`if(a==b) then {`
`  Left = right + 1;`
`  while(c < d) {`
`  ...`
`  }`
`} else {`
`  ...`
`}`
	● Satisfies Parent, Parent*, Modifies, Uses, pattern
	● An assignment statement that Follows/Follows* container statement

`if(a==b) then {`
`  while(c < d) {`
`  ...`
`  }`
`  Left = right + 1;`
`} else {`
`  ...`
`}`
	● Satisfies Parent, Parent*, Modifies, Uses, pattern
	● A container statement that Follows/Follows* an assignment statement

`if(a==b) then {`
`  Left = right + 1;`
`  while(c < d) {`
`  ...`
`  }`
`  Left = right + 1;`
`} else {`
`  ...`
`}`
	● Satisfies Parent, Parent*, Modifies, Uses, pattern
	● A container statement that Follows/Follows* an assignment statement
	● An assignment statement that Follows/Follows* container statement

Fig. 5.4.6-1: Variation of the Positioning of container statements

(2) Nesting Types for container statements

The variation for nesting types includes

- Deeper levels of nesting for if statements
- Deeper levels of nesting for while statements
- More levels of nesting between while and if
- 2 x 2 possible combinations between if and while nesting

(3) Complicated conditional expression in container statements

Source Code Snippet	Required Test Input	Expected Test Results

while (((a<s)&&((d>f)||((55/aa)<=66))) && ((((!(true==(a+2))) && (0==0)) && (11>=(q*(22+44)/w%e)))) {
  ...
}
	1 - Valid Uses variable v; while w;
	Select v such that Uses(w,v)
	a,s,d,f,aa,true,q,w,e
	5000
		a,s,d,f,aa,true,q,w,e

if (a && b) then {
...
} else {
...
}
		Invalid If Statement: expected relative expression comparator, got '&&'.

Fig. 5.4.6-2: Sample test cases with complicated conditional expression

(4) Syntactically correct but confusing variable names and constants

Design entity names can also be used as the procedure and variable names. This ensures that tokenizer probably tokenizes, gives the correct token types and the parser is able to build AST correctly. The names also are not restricted to any sizes. Similarly for constants, it can be a long integer. Hence, it helps to ensure that our implementation logic does not downcast the long integer which could possibly cause an error to be thrown.

5.4.7. Test Category 7: Valid and Confusing Query Design

In this test category, we add a complexity to the query test cases. The purpose is to ensure our SPA system returns valid results regardless of the complicated query designed as well as our Query Processor meets the functional requirement.

The valid but complicated query design includes

- Declaring synonyms with the same name as the design entities to ensure no flaw in validation rule logic
- Declaring long synonym names to handle correct string handling
- Tricky spaces between clauses and arguments to ensure no flaw in PQL grammar rules

The following are some examples of the complicated query mentioned above but system test does return valid results:

- stmt Select;
  select Select
- variable longName123longName123longName123longName123longName123longName123;
  Select longName123longName123longName123longName123longName123longName123
- stmt s;
  Select s such that Follows(_,_)
- assign a; stmt s;
  Select s such that Follows(_,_) pattern a(_, _)

5.4.8. Test Category 8: Bonus Features

In this test category, we design the test for the bonus features which we have implemented. The bonus features include queries with ModifiesP, UsesP and add flexibility to the pattern and such that structure.

<table>
<tr><th>Source Code Snippet</th><th>Required Test Input</th><th>Expected Test Results</th></tr>
<tr><td rowspan="2">procedure bonus {
   exact = 0; // 1
   partial = 0 / (1 +a   ) * b +c;
// 2
   read readread; // 3
   if ((!(d==e   )) && ((f+g*2) !=
h)) then { // 3
       while (i >= true) { // 4
           exactLeft = exactRight; //
5

           if (if==else) then { // 6
               exactLeft1 = else; //
7
               left = left + middle -
right; // 8
           } else {
               else = else + 10; // 9
           }
       }
       print abc; // 10
   } else {
       f = e + 12; // 11
       g = 12; // 12
   }
}</td><td>1 - Valid ModifiesP
stmt s; assign a;
variable v; constant c;
while w; if ifs;
procedure p;
Select v such that
Modifies(p,v)
exact, partial,
exactLeft,exactLeft1,le
ft,else,readread,f,g
5000</td><td>exact, partial,
exactLeft,exactLeft1
,left,else,readread,
f,g</td></tr>
<tr><td>3 - Valid UsesP
stmt s; assign a;
variable v; constant c;
while w; if ifs;
procedure p;
Select s such that
Uses(p,"i")
1,2,3,4,5,6,7,8,9,10,11
,12,13
5000</td><td>1,2,3,4,5,6,7,8,9,10
,11,12,13</td></tr>
</table>

Fig. 5.4.8: Sample test cases with complicated conditional expression

6. Discussion

6.1. Takeaways (What works fine for you?)

There are several main takeaways for our Team 21. Firstly, our team started off the semester by meeting up to get to know each other better. This helped us understand each other's working styles, strengths, weaknesses and interests. This allowed us to have a better idea of how the team should be allocated to the various components and utilise each member's skill sets to the maximum. The teamwork in Iteration 1 turned out really well as we could easily complete our own tasks and were flexible to assist other components when needed.

Secondly, the project management skills and tools were effective and helpful to our team. Specifically, the project management tools such as Google Drive, GitHub issue tracker, Telegram and Draw.io helped us to communicate tasks and complete them effectively. Before working on the system implementation, we ensured we have the groundwork laid out completely, following the Design Decision Principles learnt from CS3203 Lecture.

Last but not the least, the team sets aside time on fixed days every week as described in Section 2.3 and Fig. 2.3-2 to meet and share out progress, issue faced, or clarifications about the communication between components of the SPA system.

6.2. Issues Encountered (What was a problem?)

The main issue that our team encountered was the inconsistencies with the cross-platform solution which results in Windows and Mac systems having different output results when running certain tests. For example, the Windows platform is able to run the test on comparing vectors of strings and the orders of the elements in the vectors do not matter. However, on the Mac platform, the orders of the elements in the vectors matter. Hence we can use REQUIRE for this aspect of the test. Since we were forewarned about the potential of these issues, many of them were resolved quite quickly.

Another issue was not being disciplined with project management tools. Although we laid out the coding standards to adhere to before we started coding, it was easy for old habits to kick in, resulting in us having to refactor the code to follow the coding standards a few days before submission. Furthermore, most of us were not consistent with the GitHub issue tracker. As more sub-tasks came up, most of us would directly code the solution without opening an issue on GitHub.

Lastly, it was a pity that we are unable to meet and have group coding sessions in person.

6.3. What we would do differently if we were to start the project again

The lack of time to enhance our project iteration 1 with more bonuses and making it more extensible in preparation for iteration 2 and 3 was a pity for our team. If we were to start the project again, we hope to start planning the system and coding in Week 2.. If we started early, this would also mean that we have more time to debug our system. Currently, our team had long nights in week 6. Frequent lack of sleep could eventually impact our quality of work, coding, testing phase, and most importantly, our health.

Our team would also hope to read Iteration 1 requirements more carefully before we start coding, in particular, the PQL grammar. In the beginning, our failure to do so led to logic flaws. For instance, the regex expression was missing a grammar rule. Hence, unnecessary time was spent debugging and fixing them.

Lastly, our team would try to estimate the workload for each component more accurately. Our underestimation of the PQL workload resulted in us not being able to strictly adhere to the development plans and had to add buffers to our development plans.

6.4. What management lessons have you learned?

Our team has learned to have better time management skills. We should space out coding sessions instead of coding through the night to meet our internal deadlines. This helps to ensure quality in our code and system tests.

We also learned to have better communication between team members, and be responsible for our assigned tasks so that we do not bottleneck the team. Interpersonal skills also proved helpful, lifting the spirits during times when the team was visibly discouraged. For example, when a team member needs help, we would either answer each other’s queries on Telegram or hold a video meeting to discuss and work towards solving the issues together.

Lastly, we learned that having frequent reminders from respective ICs to keep team members in check of their progress helps to ensure that tasks are not left to the last minute. Team ICs can get a sense of team member’s progress and their own welfare. The team ICs could adjust the timeline accordingly to suit team member’s needs.

7. API Documentation

7.1. Tokenizer

Tokenizer Overview: Tokenizer tokenizes a SIMPLE source and outputs a token stream
TOKENIZER* getInstance() ***Description:*** Return the instance of Tokenizer
TOKEN_STREAM tokenizerSource(**PROGRAM** program) ***Description:*** Generate a stream of tokens based on the program input and return it

7.2. Parser

Parser Overview: Parser takes in a tokenized source and generates an AST.
PARSER* getInstance() ***Description:*** Return the instance of Parser
AST buildAST(**TOKEN_STREAM** token_stream) ***Description:*** If token_stream follows the production rules of a SIMPLE source program, return an AST. Else, throw an error.
AST buildExpressionAST(**TOKEN_STREAM** token_stream) ***Description:*** If token_stream follows the production rules of a SIMPLE arithmetic expression, return an AST. Else, throw an error.

7.3. AST

AST Overview: AST API allows for traversal of an AST object.
DESIGN_ENTITY getType() ***Description:*** Return the Design Entity type of the node object.
STRING getValue() ***Description:*** Return the value of the node object.
LIST_OF_TNODE getProcedures()

Description: If the node object has a design entity type of ‘Program’, return a list of procedure nodes associated with the node object. Else, throw an error.
TNODE* getExpression() ***Description:*** If the node object has a design entity type of ‘If’, ‘While’, or ‘Assign’, return the expression node associated with the node object. Else, throw an error.
LIST_OF_TNODE getStatementList() ***Description:*** If the node object has a design entity type of ‘If’, ‘While’, or ‘Procedure’, return the first list of statement nodes associated with the node object. Else, throw an error.
LIST_OF_TNODE getElseStatementList() ***Description:*** If the node object has a design entity type of ‘If’, return the second list of statement nodes associated with the node object. Else, throw an error.
TNODE* getVariable() ***Description:*** If the node object has a design entity type of ‘Read’,’Print’ or ‘If’, return the variable node associated with the node object. Else, throw an error.
TNODE* getParent() ***Description:*** If the node has a design entity type of ‘Read’,’Print’ or ‘If’, ‘Assign’ or ‘While’, return the parent node associated with the node object. Else, throw an error.
TNODE* getLeftNode() ***Description:*** If the node has a design entity type of ‘Op’,’Constant’ or ‘Variable’, return the root node of the left subtree associated with the node object. Else, throw an error.
TNODE* getRightNode() ***Description:*** If the node has a design entity type of ‘Op’, return the root node of the right subtree associated with the node object. Else, throw an error.

7.4. AST Builder

AST Builder *Overview:* ASTBuilder provides methods for building an AST
TNODE ProgramNode() ***Description:*** Instantiates a node representing a program.
TNODE ProcedureNode(**STRING** val) ***Requires:*** val is a valid procedure name. ***Description:*** Instantiates a node representing a procedure.
TNODE IfNode(**TNODE*** cond_expr)

Requires: cond_expr represents a conditional expression. ***Description:*** Instantiates a node representing an if statement.
TNODE WhileNode(**TNODE*** cond_expr) ***Requires:*** cond_expr represents a conditional expression. ***Description:*** Instantiates a node representing an while statement.
TNODE ReadNode(**TNODE*** var) ***Requires:*** var represents a variable. ***Description:*** Instantiates a node representing a read statement.
TNODE PrintNode(**TNODE*** var) ***Requires:*** var represents a variable. ***Description:*** Instantiates a node representing a print statement.
TNODE AssignNode(**TNODE*** var, **TNODE*** expr) ***Requires:*** var and expr represents a variable and expression respectively. ***Description:*** Instantiates a node representing an assign statement
TNODE OpNode(**STRING** val, **TNODE*** lhs, **TNODE*** rhs) ***Requires:*** lhs and rhs represent an expression. val must have a value of +,-,/,* or %. ***Description:*** Instantiates a node representing a operator
TNODE VariableNode(**STRING** val) ***Requires:*** val must be a valid variable name. ***Description:*** Instantiates a node representing a variable
TNODE ConstantNode(**STRING** val) ***Requires:*** val must be a valid constant value. ***Description:*** Instantiates a node representing a constant
TNODE NormCondExpressionNode(**STRING** val, **TNODE*** lhs, **TNODE*** rhs) ***Requires:*** lhs and rhs represent an expression. val must be of value && or ||. ***Description:*** Instantiates a node representing a conditional expression
TNODE NormCondExpressionNode(**TNODE*** lhs) ***Requires:*** lhs represents an expression. ***Description:*** Instantiates a node representing a ‘not’ conditional expression
TNODE RelExpressionNode(**STRING** val, **TNODE*** lhs, **TNODE*** rhs) ***Requires:*** lhs and rhs represent an expression. val must be of value <,>,==,!=,<= or >=. ***Description:*** Instantiates a node representing a relational expression
VOID addProcedureList(**LIST_OF_PROCEDURES** proc_lst) ***Description:*** If the node has a design entity type of ‘Program’, assign the nodes in proc_lst as child nodes. Else, throw an error.
VOID addStatementList(**LIST_OF_STMT** stmt_lst)

Description: If the node has a design entity type of ‘If’, ’While’ or ‘Procedure’, assign the nodes in stmt_lst as child nodes. Else, throw an error.
VOID addElseStatementList(**LIST_OF_STMT** stmt_lst) ***Description:*** If the node has a design entity type of ‘If’, assign the nodes in stmt_lst as child nodes. Else, throw an error.
VOID addWildcard() ***Description:*** If the node has a design entity type of ‘Variable’ or ‘Constant’, assign a node of type ‘Wildcard’ as a child node. Else, throw an error.

7.5. PKB (Including VarTable, ProcTable, etc.)

PKB Overview: PKB extracts, stores and fetches design abstractions from the AST
PKB getInstance() ***Description:*** Return an instance of PKB
BOOL initAST(**TNODE*** ast) ***Description:*** Pass the AST to the API for design abstraction extraction.
LIST_OF_STMT_NOS getAllWhile() ***Description:*** Returns all statement numbers that represent *while* in a vector
LIST_OF_STMT_NOS getAllIf() ***Description:*** Returns all statement numbers that represent *if* in a vector
LIST_OF_STMT_NOS getAllRead() ***Description:*** Returns all statement numbers that represent *read* in a vector
LIST_OF_STMT_NOS getAllPrint() ***Description:*** Returns all statement numbers that represent *print* in a vector
LIST_OF_STMT_NOS getAllCall() **Description:** Returns all statement numbers that represent *call* in a vector
LIST_OF_STMT_NOS getAllAssign() ***Description:*** Returns all statement numbers that represent *assign* in a vector
LIST_OF_STMT_NOS getAllStmt() ***Description:*** Returns all statement numbers in a vector

DESIGN_ENTITY getStmtType(**STMT_NO** s) ***Description:*** Returns the most specific Design Entity value associated with statement s.
LIST_OF_VARIABLE_NAMES getAllVar() ***Description:*** Returns all variable names
LIST_OF_CONSTANTS getAllConstant() ***Description:*** Returns all constants
LIST_OF_PROCEDURE_NAMES getAllProcedure() ***Description****:* Returns all procedure names
LIST_OF_STRING getAllMatchedEntity(**DESIGN_ENTITY** type) ***Description:*** Returns a string of names or statement numbers associated to the given Design Entity
LIST_OF_STRING convertIntToString(**LIST_OF_INT** list) ***Description:*** Takes in a list of integers, converts all the integer values to string, and returns the list of strings

7.6. Follows, Follows*

Follows, Follows* Overview: API related to the Follows and Follows* clauses
BOOLEAN isFollows(**STMT_NO** s1, **STMT_NO** s2) ***Description:*** Returns true if Follows(s1, s2) holds
STMT_NO getFollows(**STMT_NO** s) ***Description:*** Returns the statement number that follows s. Else, returns <= 0
STMT_NO getFollowedBy(**STMT_NO** s) ***Description:*** Returns the statement number that is followed by s. Else, returns <= 0
BOOLEAN isFollowsStar(**STMT_NO** s1, **STMT_NO** s2) ***Description:*** Returns true if Follows*(s1, s2) holds
LIST_OF_STMT_NOS getFollowsStar(**STMT_NO** s) ***Description:*** Returns all the statement numbers that follows* s
LIST_OF_STMT_NOS getFollowedByStar(**STMT_NO** s) ***Description:*** Returns all the statement numbers that is followed* by s

7.7. Parent, Parent*

Parent, Parent* Overview: API related to the Parent and Parent* clauses
BOOLEAN isParent(**STMT_NO** s1, **STMT_NO** s2) ***Description:*** Returns true if Parent(s1, s2) holds
STMT_NO getParent(**STMT_NO** s) ***Description:*** Returns the statement number that is the parent of s. Else, returns < 0
LIST_OF_STMT_NOS getChildren(**STMT_NO** s) ***Description:*** Returns all the statement numbers that s is the parent of
BOOLEAN isParentStar(**STMT_NO** s1, **STMT_NO** s2) ***Description:*** Returns true if Parent(s1, s2) holds
LIST_OF_STMT_NOS getParentStar(**STMT_NO** s) ***Description:*** Returns the statement numbers that are the parent* of s
LIST_OF_STMT_NOS getChildrenStar(**STMT_NO** s) ***Description:*** Returns all the statement numbers that s is the parent* of

7.8. Modifies for assignment statements

Modifies Overview: API related to the Modifies clause
BOOLEAN isModifies(**STMT_NO** s, **VARIABLE_NAME** v) ***Description:*** Returns true if s modifies v
BOOLEAN isModifies(**PROCEDURE_NAME** p, **VARIABLE_NAME** v) ***Description:*** Returns true if p modifies v
LIST OF VARIABLE_NAMES getAllVariablesModifiedBy(**STMT_NO** s) ***Description:*** Returns a list of variables modified by s
LIST OF VARIABLE_NAMES getAllVariablesModifiedBy(**PROCEDURE_NAME** p) ***Description:*** Returns a list of variables modified by p

7.9. Uses for assignment statements

Uses Overview: API related to the Uses clause
BOOLEAN isUses(**STMT_NO** s, **VARIABLE_NAME** v) ***Description:*** Returns true if s uses v
BOOLEAN isUses(**PROCEDURE_NAME** p, **VARIABLE_NAME** v) ***Description:*** Returns true if p uses v
LIST OF VARIABLE_NAMES getAllVariablesUsedBy(**STMT_NO** s) ***Description:*** Returns a list of variables used by s
LIST OF VARIABLE_NAMES getAllVariablesUsedBy(**PROCEDURE_NAME** p) ***Description:*** Returns a list of variables used by p

7.10. Pattern

Pattern Overview: API related to the Pattern clause
LIST_OF_STMT_NOS getStmtFullMatch(**STRING** lhs, **STRING** rhs) ***Description:*** Returns all statement numbers that match lhs on the left of the equal sign and rhs on the right of the equal sign
LIST_OF_VARIABLE_NAMES getVarFullMatch(**STRING** rhs) ***Description:*** Returns all variables that match rhs
LIST_OF_STMT_NOS getStmtPartialMatch(**STRING** lhs, **STRING** rhs) ***Description:*** Returns all statement numbers that match lhs on the left of the equal sign and partially matches rhs on the right of the equal sign
LIST_OF_VARIABLE_NAMES getVarPartialMatch(**STRING** rhs) ***Description:*** Returns all variables that partially match
LIST_OF_PAIRS_OF_STMT_NOS getPairFullMatch(**STRING** RHS) ***Description:*** Returns all statement-variable pairs that fully match the pattern
LIST_OF_PAIRS_OF_STMT_NOS getPairPartialMatch(**STRING** RHS) ***Description:*** Returns all statement-variable pairs that partially match the pattern

7.11. Query Preprocessor

Query Preprocessor Overview: Query Preprocessor generates a Query object from a query string
QUERY_PREPROCESSOR* getInstance() ***Description:*** Returns an instance of Query Preprocessor
QUERY constructQuery(**STRING** query_string) ***Description:*** Returns a QUERY object that stores the relevant information of the input query

7.12. Query Evaluator

Query Evaluator Overview: Query Evaluator evaluates a Query object and returns the unformatted raw results
QUERY_EVALUATOR* getInstance() ***Description:*** Returns an instance of Query Evaluator
VOID evaluateQuery (**QUERY** query) ***Description:*** Update from a given QUERY object

7.13. Query Result Projector

Query Result Projector **Overview:** Query Result Projector formats the raw results and stores them into a result list
QUERY_RESULT_PROJECTOR* getInstance() ***Description:*** Returns an instance of Query Result Projector
VOID formatResults (**STRING_LIST** raw_results, **STRING_LIST&** result) ***Description:*** Insert all results in raw_results list into result list.