SCHOOL OF COMPUTING

CS3203
Software Engineering Project

Iteration 2 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

1.1. Parsing Source Language SIMPLE

Frontend is able to tokenize and parse AST to PKB. The parser is able to check for valid lexical rules as well as satisfying the SIMPLE grammar rule. It is able to parse complete source language SIMPLE, according to the SIMPLE grammar rules as listed in the Full SPA Requirements including the parsing of multiple procedures.

1.2. Program Knowledge Base (PKB)

PKB receives the AST from parser and the Design Extractor Sub-component extracts design entities such as procedures, statements, variables, constants and the following design abstractions: *Follows, Follows*, Parent, Parent*, Uses, Modifies, Calls, Calls*, Next*. The aforementioned information is stored in the PKB Storages. The stored design abstractions are then searched for and given to the Query Evaluator for evaluation via PKB API calls.

1.3. Query Processor (QP)

QP is able to handle multiple constraints within a clause and multiple same-type clauses within a query. QP is also able to handle a single synonym, tuple or BOOLEAN in the Select clause. QP handles multiple **such that**, **with**, and **pattern** clauses with any number of **and** operators. QP handles the new relationships in interaction 2 including *Follows, Follows*, Parent, Parent*, Uses, Modifies, Calls, Calls*, Next, Next**. QP is also able to handle with-constraints, attribute names, pattern while- and if- type.

1.4. Query Evaluator (QE)

QE is able to handle queries containing multiple clauses and evaluate queries with multiple conditions. The multiple clauses can have multiple **such that**, **with**, and **pattern** clauses with any number of **and** operators. The multiple clauses can be a combination of *Follows, Follows*, Parent, Parent*, Uses, Modifies, Calls, Calls*, Next, Next*, with or pattern assign, pattern while, pattern if*. QE is able to return results of synonym, BOOLEAN or Tuples.

1.5. Extensions

For extensions, we plan to extend the SIMPLE CSG to include For-loops and Do-While Loops. We also plan to implement NextBip/NextBip* and AffectsBip/AffectsBip*.

1.6. Constraints

Our SPA Program will not store Next* relationships; it will instead be computed on the fly by the PKB during Query Evaluation.

2. Development Plan

2.1. Project Roles & Meeting Plans

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

Fig. 2.1-1: 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. 2.1-2: Fixed Meeting Schedule from Weeks 3 to 9 for every Monday, Tuesday and Saturday

Based on our experience with Iteration 1 as well as our scope for Iteration 2, we concluded that most of the development efforts would likely be spent on Query Processor implementation. Furthermore, we predicted that the Iteration 2 implementation requirements for Front-End Parser and PKB were simple to code. Thus, we decided to assign one team member to work on the implementation of Front-End Parser and PKB, so that the rest of the team members may focus entirely on completing Query Processor implementation.

For our meeting plans, we have decided to meet at the beginning of each week discuss our assigned tasks and plans for that week. A meeting is also scheduled after our meeting with the Teaching Assistant on Tuesday for a group coding session, as well as to discuss any issues brought up during the meeting. Finally, we schedule a meeting at the end of the week to discuss any component bugs discovered through integration or system testing, so that the team member responsible for implementing the component is made aware and may work on bug fixes in the coming week.

2.2. Milestone of Main Tasks GANTT Chart

Main Task/Week	**R**	**7**	**8**	**9**
Update Front-End Parser Implementation	🟦			
Update PKB Storage Implementation	🟦			
Update Design Extractor Implementation	🟦			
Calls/Calls* Clause Implementation	🟦			
Next/Next* Clause Implementation	🟦			
With Clause Implementation		🟦		
Clause Refactorization and Abstraction		🟦	🟦	
Intermediate Table Implementation		🟦	🟦	
Rewriting Clause to implement intermediate tables			🟦	🟦
Tuple/Boolean Implementation				🟦
Extensions Planning		🟦	🟦	🟦
System Test Implementation Task		🟦	🟦	🟦
Run System Test with Autotester				🟦
System Bug Fixes				🟦
Iteration 2 Report Writing	🟦	🟦	🟦	🟦

Fig. 2.2: An overview of the milestone of our project main development tasks

2.3. Activity Breakdown

Activity	**Completed By:**					
	Jefferson	**Adam**	**Jaime**	**Rachel**	**Su**	**Anqi**
Front-End Parser Implementation Task						
Update Tokenizer Implementation		*				
Update Parser Implementation		*				
Update AST API		*				
Program Knowledge Base Implementation Task						

Update Design Extractor		*				
Update PKB Storage API		*				
Query Processor Implementation Task						
Refactor QPP for full PQL grammar						*
Refactor QuerySyntaxChecker						*
Refactor QueryParser						*
Create/Modify REGEX constants for syntax validation						*
Create Intermediate Table APIs					*	*
Implement Intermediate Table		*				
Implement Calls/Calls* Clause		*				
Implement Next/Next* Clause		*				
Implement With Clause			*	*		
Implement If/While Pattern Clause					*	
Implement Tuple/Boolean Select Clause	*				*	
Refactorization of Clauses	*				*	
Rewrite Clauses For Intermediate Table	*				*	
Unit Testing Task						
Update Parser Unit Tests			*			
Update PKB Storage Unit Tests		*				
Update AST API Unit Tests			*			
Update Design Extractor Unit Tests		*				
Update QueryPreProcessor Unit Tests						*
Update QuerySyntaxChecker Unit Tests						*
Update QueryParser Unit Tests						*
Implement If/While Pattern Clause Unit Tests					*	
Implement With Clause Unit Tests			*			
Implement Next/Next* Unit Tests	*	*				
Implement Calls/Calls* Unit Tests		*				

Implement Intermediate Table Unit Tests		*				
Integration Testing Task						
Update Front End Parser/PKB Integration Tests			*			
Update PKB/PQL Integration Tests	*				*	
System Testing Task						
System Tests that covers for ● Select and return clause ● Such that clauses (Next/Next*, Calls/Calls*,) ● ModifiesP and UsesP Clause ● While/If Pattern Clause ● Tuple/Boolean Select Clause				*		
Source Focused System Tests ● Invalid Source ● Complicated Source				*		

Fig. 2.3: An overview of the activity development plan

2.4. Task Breakdown

	Iteration 2			
	Recess Week	**Week 7**	**Week 8**	**Week 9**
Jefferson Sie	Plan development schedule for Iteration 2	Refactor relational clauses further into IntRelClause, NamesRelClause and MultiStmtRelClause to reduce repeated code	Rewrite evaluate functions for relational clauses in Query Evaluator to account for BOOL Report writing: - Query Evaluator - Optimisations (removed for Iteration 2)	Rewrite evaluate functions for relational clauses in Query Evaluator to adopt for intermediate table implementation Report writing: - Query Evaluator - Discussion - API
Rachel Tan	Plan development schedule for Iteration 2	Add missing system test cases that did not capture iteration 1 system bugs Implement with clause logic Plan system test cases for iteration 2 system and extensions	Implement system test cases for iteration 2 system: - Select clauses - with clause - pattern while, if types - pattern assign full specification Report writing: - Update Tokenizer portion - Update Test plan	Implement system test cases for iteration 2 system: - Multiple clauses Report writing: - Update Scope - Update Development Plan - Update Test Plan - Update System test

Jaime Chow	Plan Documentation Schedule For Iteration 2	Implement With Clause Validation and Evaluation logic Update Unit tests for AST and Parser Overhaul report structure for SPA Component section according to feedback given	Implement Integration testing for With Clause Report Writing: - Update Parser Section - Update Front-End Parser Section	Refactor With Clause Report Writing: - Write SIMPLE CSG Extension Section - Write With Clause section
Chen Su	Plan development schedule for Iteration 2	Design Query Result Class (the intermediary table to store results) and algorithm of merging results from multiple clauses.	Refactor PatternClause Implement PatternWhile and PatternIf Classes.	Refactor QE and QRP Implement Select Clause Integration testing between QE and PKB Report writing: - Update Query Evaluator Section - Write NextBip/ AffectsBip extension
Chen Anqi	Plan development schedule for Iteration 2	Refactor regexes, QueryPreProcessor, QuerySyntaxChecker & QueryParser for clause groups, tuple / BOOLEAN selection, Call/Calls*/Next/Next*/ PatternIf/PatternWhile/ PatternAssign Create QueryResult class and APIs	Implement unit testing for newly implemented functions in QueryPreProcessor, QuerySyntaxChecker & QueryParser	Improve regexes for syntax validation Update unit tests Report writing: - Query Processor section - Update PQL unit testing section - Write PQL part for NextBip/ AffectsBip extension
Adam Tan	Plan development schedule for Iteration 2	Add Calls/Calls*, Next/Next* Update Frontend and PKB logic	Implement Query Result API Implement Integration Testing for Calls/Calls* and Next/Next*	Add API for PatternWhile and PatternIf Report Writing: - PKB

Fig. 2.4-1: An overview of the development task breakdown

3. SPA Architecture

3.1. Main Architecture Components

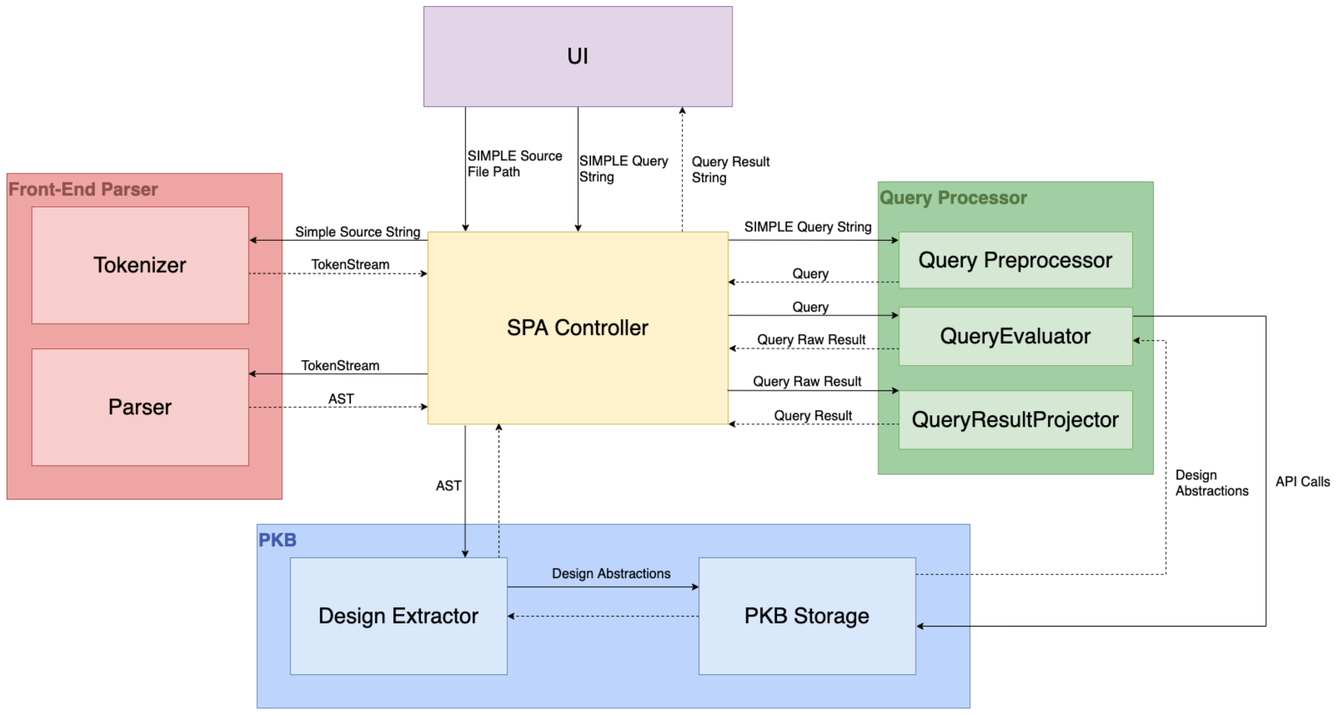


Fig. 3.1-1: Overview of SPA Architecture Component

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.

3.2. Component Interaction Overview

The interaction between components are as follows:

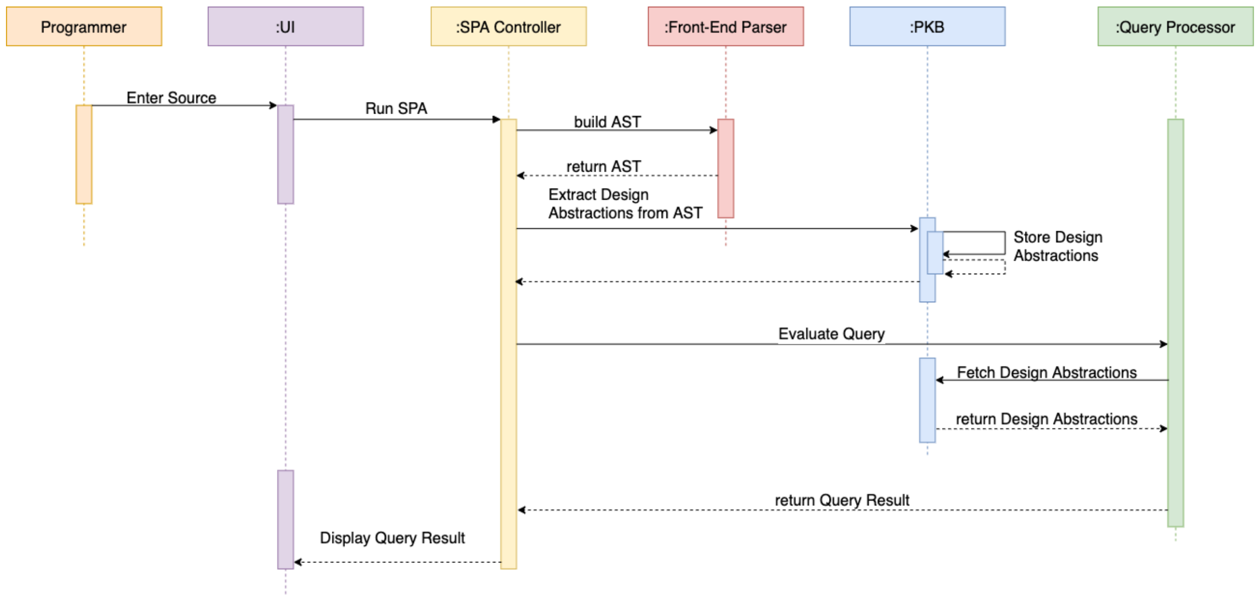


Fig. 3.2-1: High Level UML Sequence Diagram of SPA Program

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.

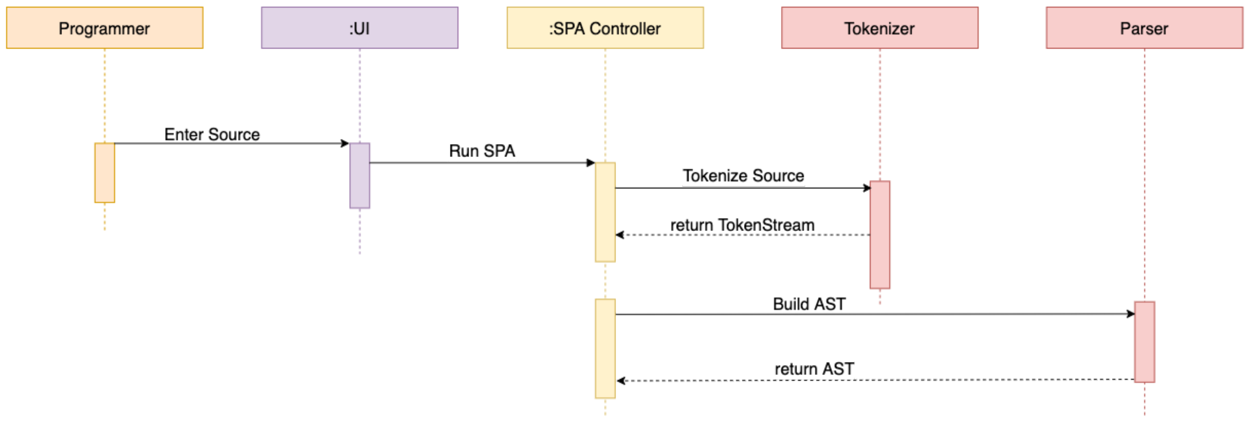


Fig. 3.2-2 High Level UML Sequence Diagram of Front-End Parser

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.

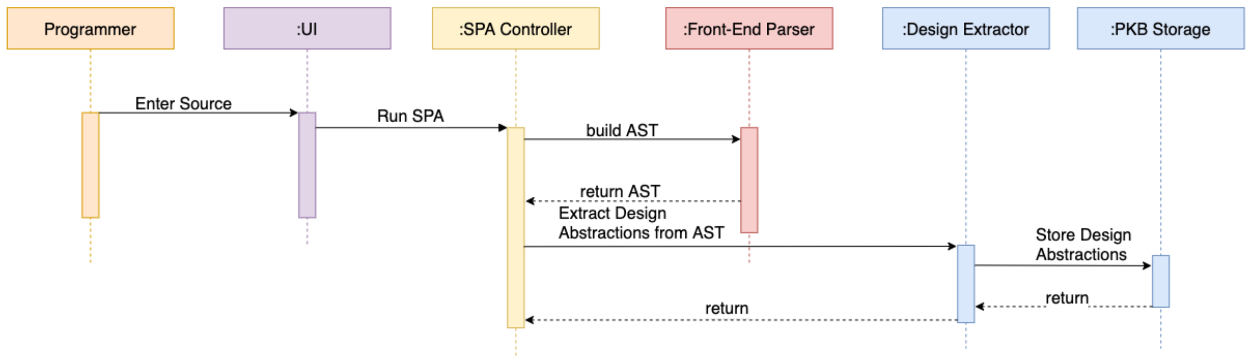


Fig. 3.2-3 High Level UML Sequence Diagram of PKB

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.

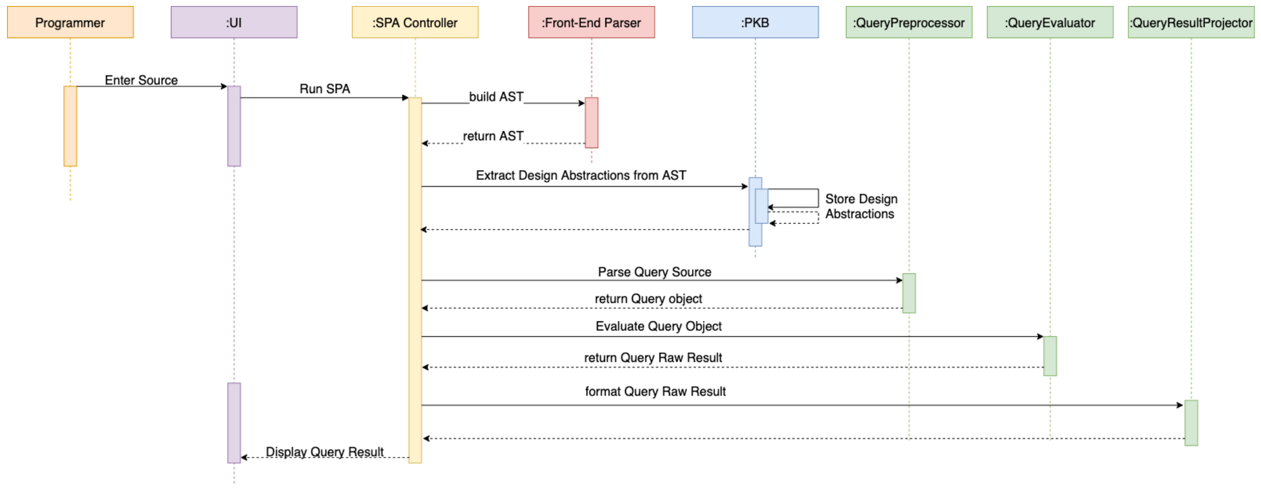


Fig. 3.2-4 High Level UML Sequence Diagram of Query Processor

3.3. Front-End Parser

3.3.1. Overview of Front-End Parser

The Front-End Parser takes in a SIMPLE source program string as input, and outputs an AST. The AST would be traversed by the PKB to extract design relationships. The purpose of the Front-End Parser is to determine if the source program input is structurally and syntactically valid.

To accomplish this task, the Front-End Parser has been abstracted into 2 subcomponents - the Tokenizer and the Parser. Our design differs from the recommended SPA architecture setup in that the Design Extractor sub-component has been moved to the PKB component.

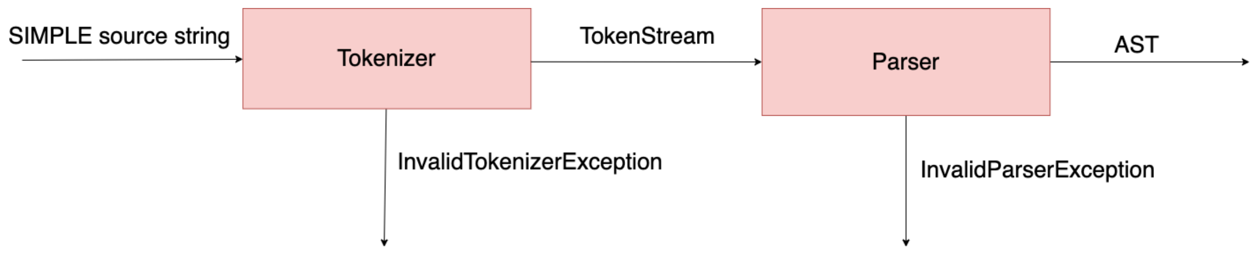


Fig. 3.3.1-1: Production Pipeline of Front End Parser Architecture

The process to convert a SIMPLE source program to an AST is as follows:

1. The SPA Controller passes the SIMPLE source string to the Tokenizer. The Tokenizer takes in the SIMPLE source string and converts it into a “Token Stream”.
2. The Token Stream is passed to the SPA Controller.
3. The SPA Controller passes the Token Stream to the Parser. The Parser takes in the Token Stream and converts it into a corresponding AST.

3.3.2. Tokenizer

3.3.2.1. Overview of Tokenizer

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.

3.3.2.2. Data Structures for Tokenizer

(1) Token

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. 3.3.2.2-1.

Token Type	Sample Token Values
Identifier	Any var_name and proc_name that satisfy NAME rules
Keyword	read, print, while, if, assign. call
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. 3.3.2.2-1: Examples of Token values

(2) Token Stream

The Token Stream data structure is implemented as a **vector of Token objects**. While the source program is being tokenized which the implementation details has been found in Section 3.3.2.3 below, each tokenized token generated will be sent to Token class for the creation of Token object. After a Token object has been created, it will be appended to the resulting token stream vector list.

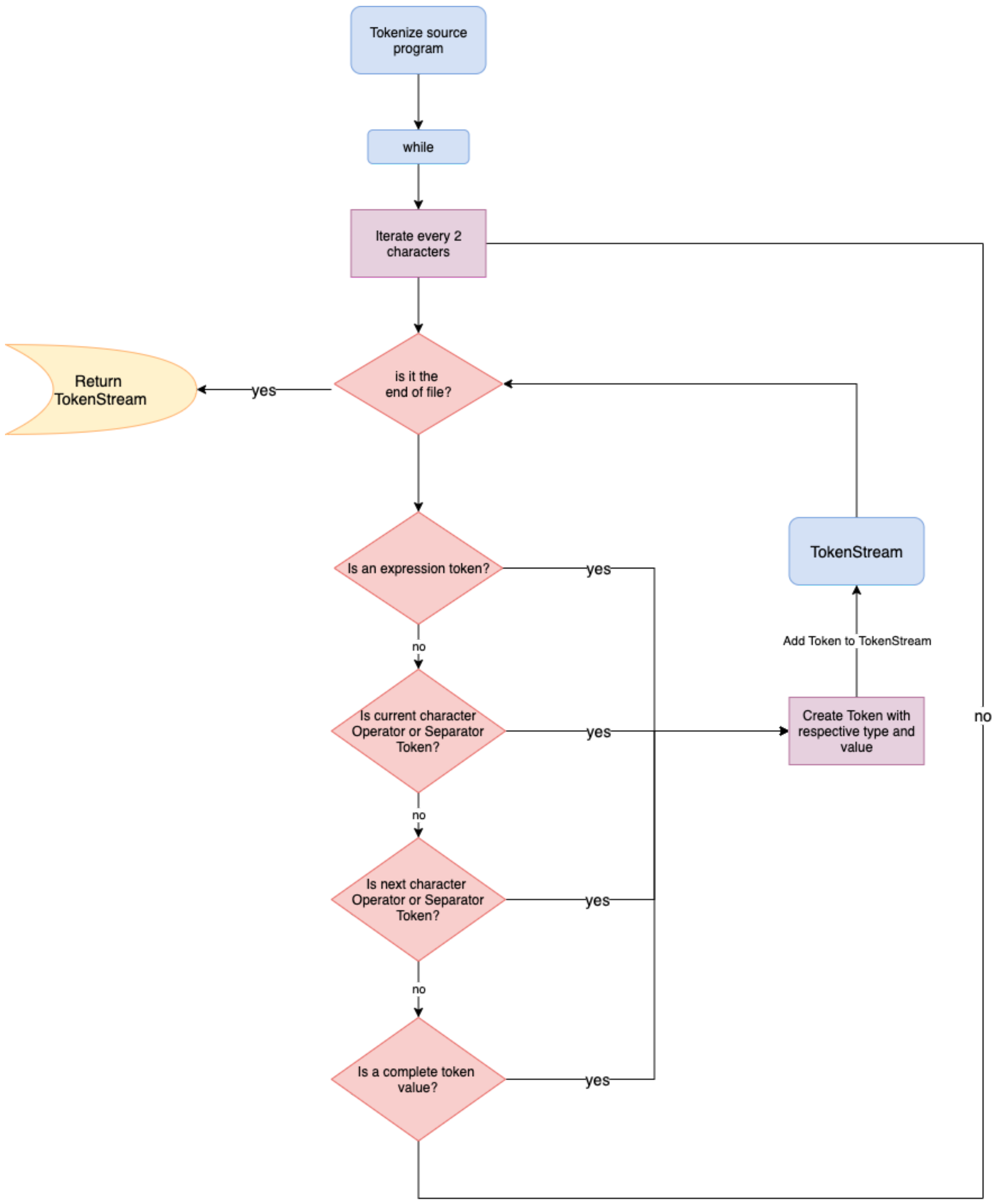
3.3.2.3. Implementation for Tokenizer

(1) Tokenization Sequence

The task of determining the sequence of tokenizing the source program is accomplished by doing a logic check while iterating through every 2 characters in the source program.

The tokenizing logic is as follows:

1. The Tokenizer will start iterating through the source program, it will get the following:
    a. Current and next character called `currChar` and `nextChar` respectively
    b. Form the current and next characters into a string called `tempToken`
    c. Append any characters or `tempToken` that does not satisfy as a token value called `currToken`
2. Case 1: It start by checking if `tempToken` is an **expression** token value
    a. If the `currToken` has characters, this means that it is has a token value and it will be send to Token class for grammar syntax rule validation and create Token object and finally, appended to `TokenStream`
    b. If the `currToken` has no character, the `tempToken` value is instead used to send to Token class for grammar syntax rule validation and to create a Token object and finally, appended to `TokenStream`
    c. Update the next `currChar` and `nextChar`
3. Case 2: `currChar` is in **Operator** or **Separator** token
    a. Send `currChar` to Token class for grammar syntax rule validation and create Token object of Operator or Separator type and finally, appended to `TokenStream`
    b. Update the next `currChar` and `nextChar`
4. Case 3: `nextChar` is in **Operator** or **Separator** token
    a. Send `currToken` to Token class for grammar syntax rule validation and create Token object of Operator or Separator type and finally, appended to `TokenStream`
    b. Update the next `currChar` and `nextChar`
5. Case 4: `currChar` is a space and currToken has characters
    a. `currToken` is a **complete token value**
6. Case 5: If `currChar` is not a space, this means `currToken` is **not a complete token value** and should append the `currChar` to the `currToken` and update the next `currChar` and `nextChar`
7. Case 6: If all the steps 2-6 is not satisfied, this means that the `currChar` and `nextChar` are spaces, do not need to tokenize and can be **skipped**
    a. Update the next `currChar` and `nextChar`
8. Case 7: Once the Tokenizer reached the **end of file**, it will exit the iterating loop and will check if `currChar` contain a **final character**
    a. `currChar` is sent to Token class for grammar syntax rule validation and to create a Token object and finally, appended to `TokenStream`
9. An **end** Token object is created and appended to the `TokenStream`
10. The Tokenizer will return the corresponding `TokenStream`



Tokenize source program
while
Iterate every 2 characters
is it the end of file?
yes
Return TokenStream
Is an expression token?
yes
no
Is current character Operator or Separator Token?
yes
no
Is next character Operator or Separator Token?
yes
no
Is a complete token value?
yes
TokenStream
Add Token to TokenStream
Create Token with respective type and value
no

Fig. 3.3.2.3-1: High-level Flowchart of Tokenization Sequence

Example

As an example, the Tokenizer will tokenize the source program as follows

```
read x;
```

Fig. 3.3.2.3-2: Sample source

Steps	currChar	nextChar	Logic Check	temptoken	currToken	TokenStream
1	r	e	Case 6	re	r	
2	e	a	Case 5	ea	re	
3	a	d	Case 5	ad	rea	
4	d		Case 5	d	read	
5		x	Case 4			[{type:Keyword, value:read}]
6	x	;	Case 3		x	[{type:Keyword, value:read}, {type:Identifier,value:x}]
7	;		Case 2			[{type:Keyword, value:read}, {type:Identifier,value:x}, {type:Separator, value: ;}]
8			Create end token			[{type:Keyword, value:"read"}, {type:Identifier, value:"x"}, {type:Separator, value: ";"}, {type:EndOfFile, value:"EOF"}]

Fig. 3.3.2.3-3: High-level Flowchart of Tokenization Sequence

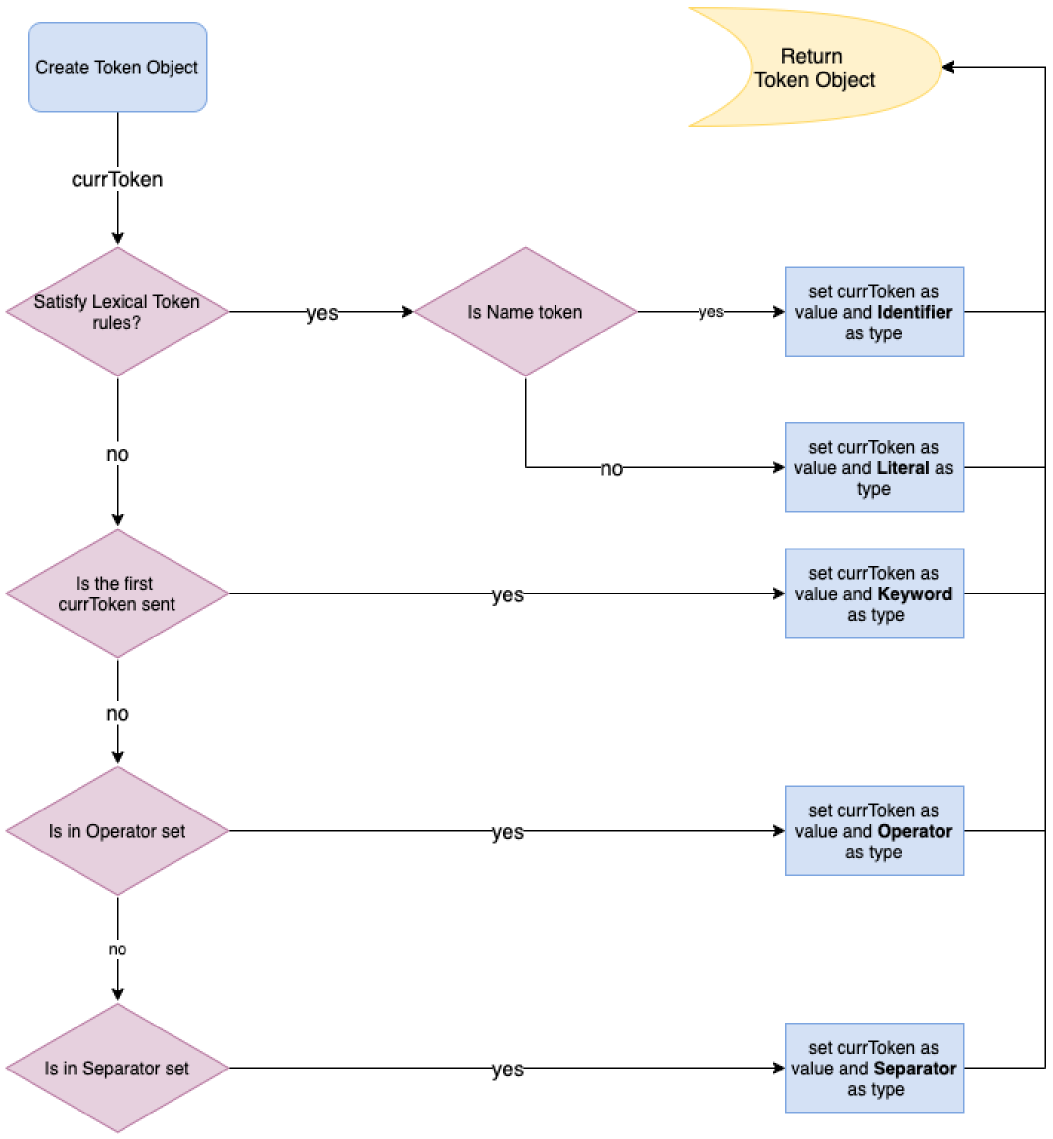
(2) Creation of Token object

The task of creating a Token object is accomplished by setting the token type and token value of the `currToken` that was sent from Tokenizer to Token class.

The creation of Token object logic is as follows:

1. `currToken` first checks if it is a valid lexical token
    a. If `currToken` is a NAME, it should only contain LETTER and DIGIT with first character being a LETTER
    b. If `currToken` is an INTEGER, it should only contain DIGITS from 0 to 9
2. If `currToken` is a valid lexical token and an INTEGER, then it is set to token type of Literal
3. If `currToken` is a valid lexical token and a NAME, then it is set to token type of Identifier
4. If `currToken` is not a valid lexical token and is in the `TOKEN_OPERATOR_SET` which contains Operator Token Type as shown in Table 3.3.2.2-1, then it is set to token type of Operator
5. If `currToken` is not a valid lexical token and is in the `SEPARATOR_SEPARATOR_SET` which contains Separator Token Type as shown in Table 3.3.2.2-1, then it is set to token type of Separator
6. If `currToken` is the first token sent, not a valid lexical token, then it is set to token type of Keyword

7. If `currToken` does not satisfy step 2 to 6, it does not satisfy SIMPLE procedure rules and an exception will be thrown
8. An End Token object is created at the end of the creation of all tokenized token object



Create Token Object
Return Token Object
currToken
Satisfy Lexical Token rules?
yes
Is Name token
yes
set currToken as value and **Identifier** as type
no
no
set currToken as value and **Literal** as type
Is the first currToken sent
yes
set currToken as value and **Keyword** as type
no
Is in Operator set
yes
set currToken as value and **Operator** as type
no
Is in Separator set
yes
set currToken as value and **Separator** as type

Fig. 3.3.2.3-4: High-level Flowchart of Creation of Token object

<u>Example</u>

As an example, the Token class will create token object after tokenizing the source program as follows

```
procedure eg {
 read x;
}
```

Fig. 3.3.2.3-5: Sample source

As explained in the previous section on tokenizing sequence, the above sample source will be tokenized and each token will have a Token object with value and type.

currToken	Resulting Token value	Resulting Token Type
procedure	procedure	Keyword
eg	eg	Identifier
{	{	Separator
read	read	Keyword
x	x	Identifier
;	;	Separator
}	}	Separator

Fig. 3.3.2.3-6: Token Objects created by tokenizing Fig. 3.3.2.3-5: Sample source

3.3.2.4. Tokenizer 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. 3.3.2.4-1: 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. 3.3.2.4-2: Sample Exception

3.3.3. Parser

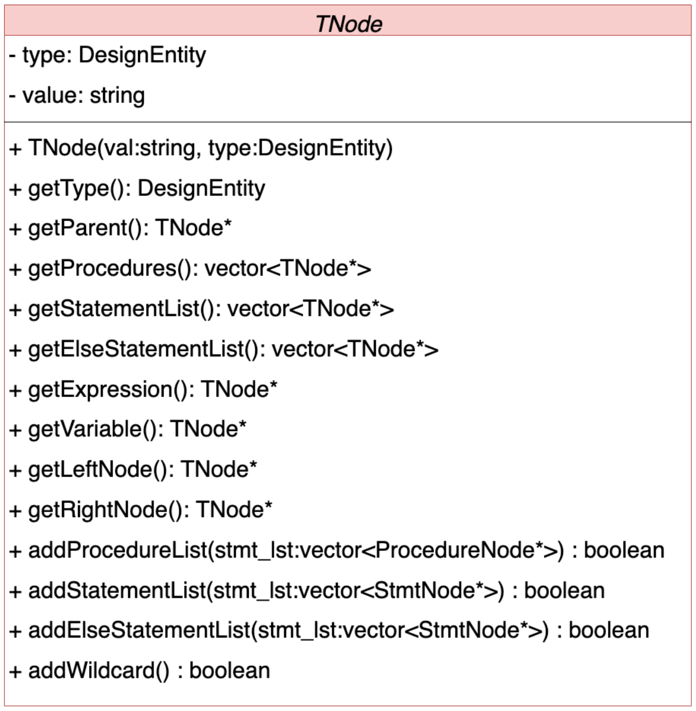
3.3.3.1. Overview of Parser

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.

3.3.3.2. Data Structures for Parser

(1) AST

The implemented AST structure coincides completely with what was taught during lectures and during design considerations, we avoid making alterations to the structure. An AST data structure is implemented as a **tree graph** populated by the various Node class objects that represent different SIMPLE entities. The Node objects all inherit from the given TNode object.



TNode
- type: DesignEntity
- value: string
+ TNode(val:string, type:DesignEntity)
+ getType(): DesignEntity
+ getParent(): TNode*
+ getProcedures(): vector<TNode*>
+ getStatementList(): vector<TNode*>
+ getElseStatementList(): vector<TNode*>
+ getExpression(): TNode*
+ getVariable(): TNode*
+ getLeftNode(): TNode*
+ getRightNode(): TNode*
+ addProcedureList(stmt_lst:vector<ProcedureNode*>) : boolean
+ addStatementList(stmt_lst:vector<StmtNode*>) : boolean
+ addElseStatementList(stmt_lst:vector<StmtNode*>) : boolean
+ addWildcard() : boolean

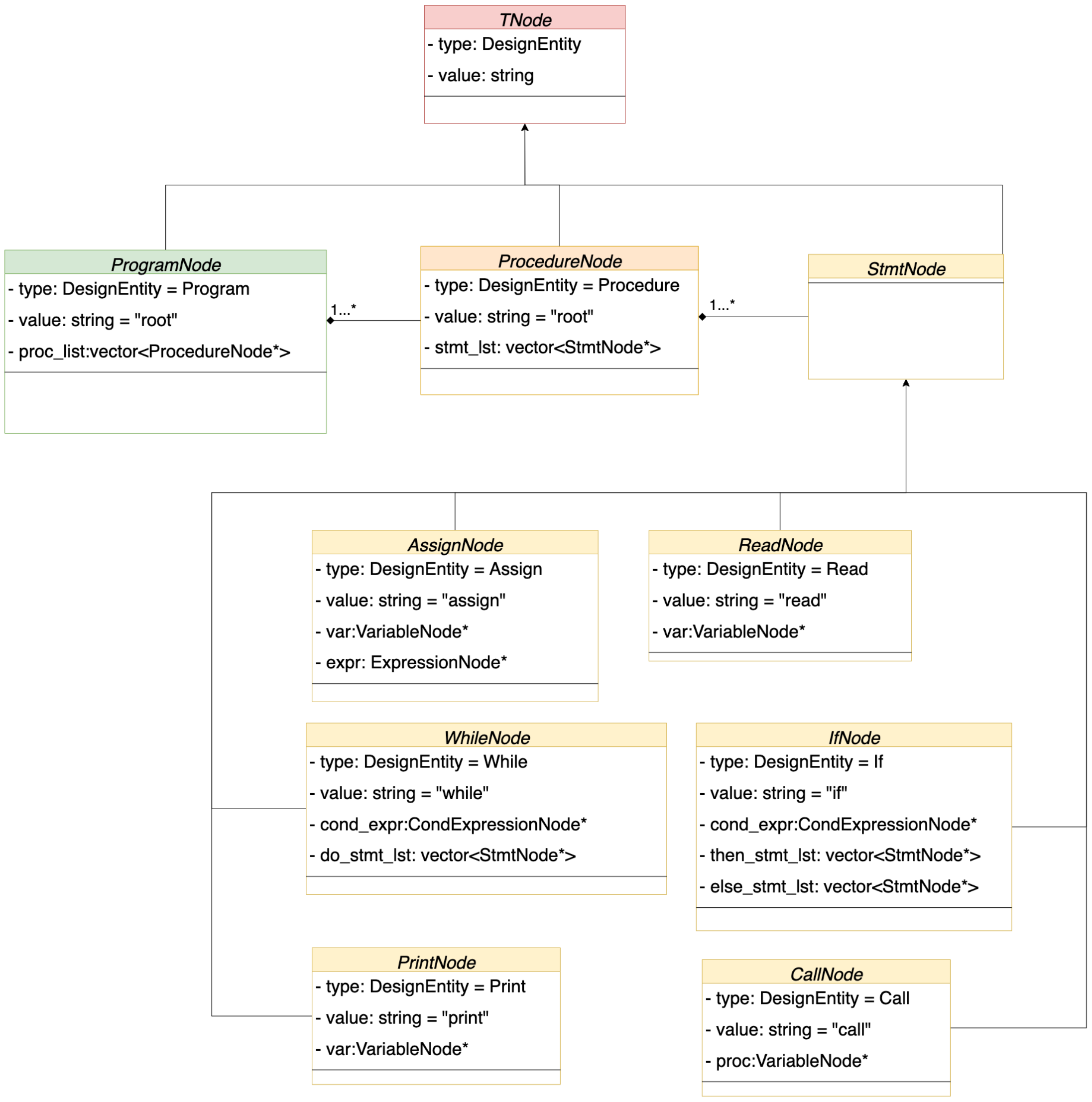
Fig. 3.3.3.2-1: Class Diagram of the abstract TNode Object

Each TNode object has two variables: a string **value** and a DesignEntity enum **type.** The Design Entity enum type 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 node classes can be classified into three categories:

General Nodes



TNode
- type: DesignEntity
- value: string
ProgramNode
- type: DesignEntity = Program
- value: string = "root"
- proc_list:vector<ProcedureNode*>
1...*
ProcedureNode
- type: DesignEntity = Procedure
- value: string = "root"
- stmt_lst: vector<StmtNode*>
1...*
StmtNode
AssignNode
- type: DesignEntity = Assign
- value: string = "assign"
- var:VariableNode*
- expr: ExpressionNode*
ReadNode
- type: DesignEntity = Read
- value: string = "read"
- var:VariableNode*
WhileNode
- type: DesignEntity = While
- value: string = "while"
- cond_expr:CondExpressionNode*
- do_stmt_lst: vector<StmtNode*>
IfNode
- type: DesignEntity = If
- value: string = "if"
- cond_expr:CondExpressionNode*
- then_stmt_lst: vector<StmtNode*>
- else_stmt_lst: vector<StmtNode*>
PrintNode
- type: DesignEntity = Print
- value: string = "print"
- var:VariableNode*
CallNode
- type: DesignEntity = Call
- value: string = "call"
- proc:VariableNode*

Fig. 3.3.3.2-2: Nodes objects generated by general Program Parsing

General Nodes are nodes that represent the broad general structure of an AST.

- **ProgramNode:** Represents the root node of the AST. Stores a list of `ProcedureNodes` as children.
- **ProcedureNode:** Represents a procedure of the SIMPLE program. Stores a list of `StmtNodes` as children.
- **StmtNode:** Represents the different statement types of a SIMPLE program. The `StmtNode` is further abstracted into the different statement types. Each `StmtNode` type is able to store the required child node types. For instance, a `ReadNode` will be able to store a child `VariableNode`.

Expression Nodes

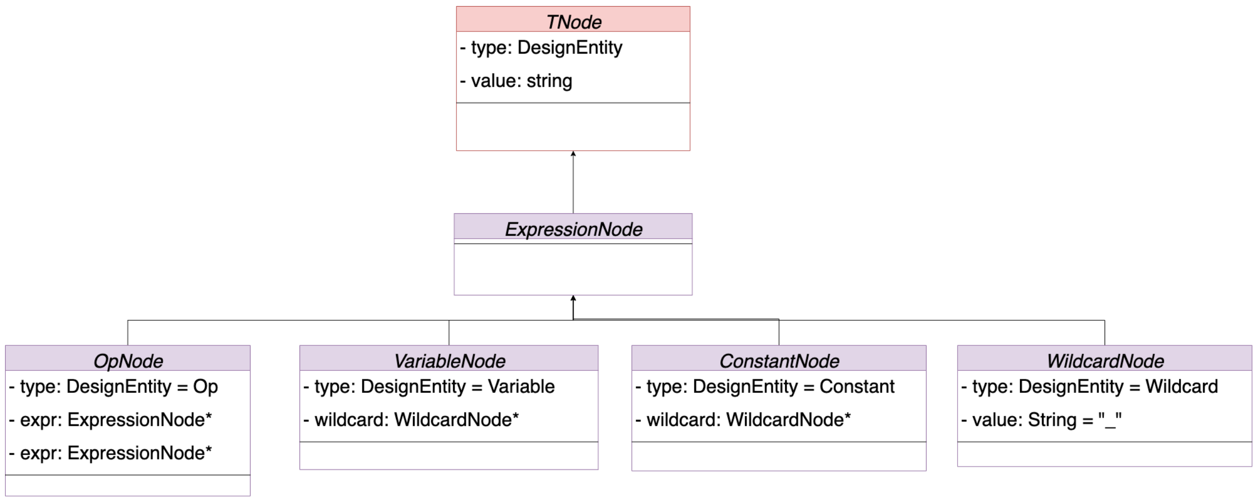

TNode
- type: DesignEntity
- value: string
ExpressionNode
OpNode
- type: DesignEntity = Op
- expr: ExpressionNode*
- expr: ExpressionNode*
VariableNode
- type: DesignEntity = Variable
- wildcard: WildcardNode*
ConstantNode
- type: DesignEntity = Constant
- wildcard: WildcardNode*
WildcardNode
- type: DesignEntity = Wildcard
- value: String = "_"

Fig. 3.3.3.2-3: Node objects generated by Expression Parsing

Expression Nodes are nodes that represent SIMPLE expressions:

- **OpNode:** Represents an arithmetic operation in a SIMPLE program. Stores two Expression nodes as children.
- **VariableNode:** Represents a variable of the SIMPLE program. Able to store a `WildcardNode` as a child.
- **ConstantNode:** Represents a constant of a SIMPLE program. Able to store a `WildcardNode` as a child.
- **WildcardNode:** A node used to indicate if a variable or constant appears at the beginning and end of an expression. For instance, in the expression `"a+x+1"`, the node objects representing "a" and "1" will have a `WildcardNode` attached. The `WildcardNode` is used to assist in expression pattern matching.

Conditional Expression Node

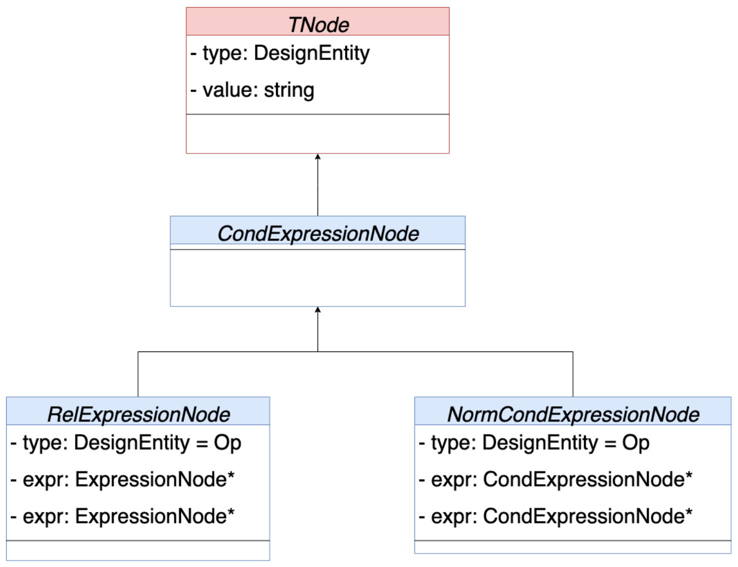


Figure 3.3.3.2-4: Node objects generated by Conditional Expression Parsing

The Conditional Expression Nodes are nodes that represent Conditional Expressions.

- **RelExpressionNode:** Represents a relational expression node. Able to store two child expression nodes representing the left and right expression from the relational operator.
- **NormCondExpressionNode:** Represents a conditional expression node. Able to store two child conditional expression nodes representing the left and right expression from the conditional operator.

Each Node class corresponds to a SIMPLE grammar non-terminal. The mappings are as follows:

Non-terminal being parsed	Object returned after parsing
program	ProgramNode
procedure	ProcedureNode
stmtLst	Vector<StmtNode>
stmt	StmtNode
read	ReadNode
print	PrintNode
call	CallNode
while	WhileNode
if	IfNode
assign	AssignNode

cond_expr	NormCondExpressionNode
rel_expr	RelExpressionNode
expr	ExpressionNode
variable	VariableNode
constant	ConstantNode

Fig. 3.3.3.2-5: Mapping between Node objects and SIMPLE grammar non-terminals

3.3.3.3. Implementation for Parser

(1) Parser Types

The Parser consists of two parser types:

- A single pass **Top-Down Recursive Descent Parser**. This parser is used to parse non-expression SIMPLE Language entities.
- A Pratt Parser, a one-pass **Top-Down Recursive Operator Precedence Parser**. This parsing type is used to parse SIMPLE expressions.

The Top-Down Recursive Descent Parser was chosen for general parsing as the SIMPLE grammar, barring arithmetic expressions, is LL(k) and thus suitable for this method of parsing.

However, SIMPLE arithmetic expressions are not able to be parsed via standard recursive descent parsing due to left recursion in the SIMPLE grammar. Thus, we chose to implement the Pratt Parser, which is able to handle infix expression parsing, to parse expression separately.

(2) Parser Validation

The Parser checks the structural validity of the SIMPLE source program by mapping the Token Stream sequence to SIMPLE grammar rules. The logic for parsing the various grammar rules is abstracted into separate internal methods. For instance, there is a method dedicated to parsing program non-terminals, procedure non-terminals, and so on.

Within each internal method, the Parser traverses the SIMPLE appropriate grammar from left to right. It checks if there is a mapping between a given symbol in a grammar rule for the non-terminal being parsed and the current Token being looked at in the Token Stream:

- If the symbol is a keyword, the Token value should exactly match the keyword value.
- If the symbol is a non-terminal, the Parser calls the method to validate the token-to-grammar rule mapping for that non-terminal symbol.

If the mapping is successful, the Parser will then check the mapping between the next symbol in the grammar rule and the next Token in the Token Stream.

The parsing of the non-terminal is completed when all symbols have been successfully mapped. A node object corresponding to the non-terminal, as shown in Figure 3.3.3.2-5 is returned. For instance, a successful call to parse a procedure non-terminal will return a `ProcedureNode`. If mapping fails, the SIMPLE program is considered structurally invalid and an InvalidParserException is thrown.

<u>Example</u>

As an example, assume the Parser has called the internal method to parse the following while statement:

```
while (x == 1) {
```

```
read x;
print y;
}
```

Fig. 3.3.3.3-1: Sample Source

The token-to-grammar rule mapping logic of the internal method will be as follows:

Grammar Rule	Token Value/s Parsed	Parser Expectations
'while'	'while'	Token value matches 'while'.
'('	'('	Token value matches '('.
cond_expr	"x" "==" "1"	Parser makes a call to parse cond_expr non-terminal. CondExpressionNode cond_node is returned.
')'	')'	Token value matches ')'.
'{'	'{'	Token value matches '{'.
stmtLst	"read" "x" ";" "print" "y" ";"	Parser makes a call to parse stmtLst non-terminal. Vector<StmtNode> stmt_lst is returned;
'}'	"}"	Token value matches '}'.
-	-	End of while grammar rules. All checks have passed, a WhileNode with cond_node and stmt_lst as child nodes is returned.

Fig. 3.3.3.3-2: Parsing of Fig. 3.3.3.3-1

(3) Parsing Sequence

The task of determining the sequence of SIMPLE grammar syntax to parse is accomplished by simply following the natural order of a SIMPLE Program. Calling the method to parse a program non-terminal at the beginning of every parsing sequence will trigger a series of recursive calls to parse non-terminals within the grammar rules.

The program parsing logic is as follows:

1. The Parser will continuously call the method to parse procedure non-terminal, which will return the corresponding `ProcedureNode`.

2. Prior to each call, the Parser checks if the next token type is “EndOfFile”. This Token is appended to the end of every Token Stream and indicates that there are no more tokens to parse.
3. If this check is met, a `ProgramNode` is created containing all instantiated `ProcedureNodes` as children. As a program non-terminal must contain at least one procedure, an error is thrown if the `ProgramNode` has no children. Otherwise, the `ProgramNode` is returned.

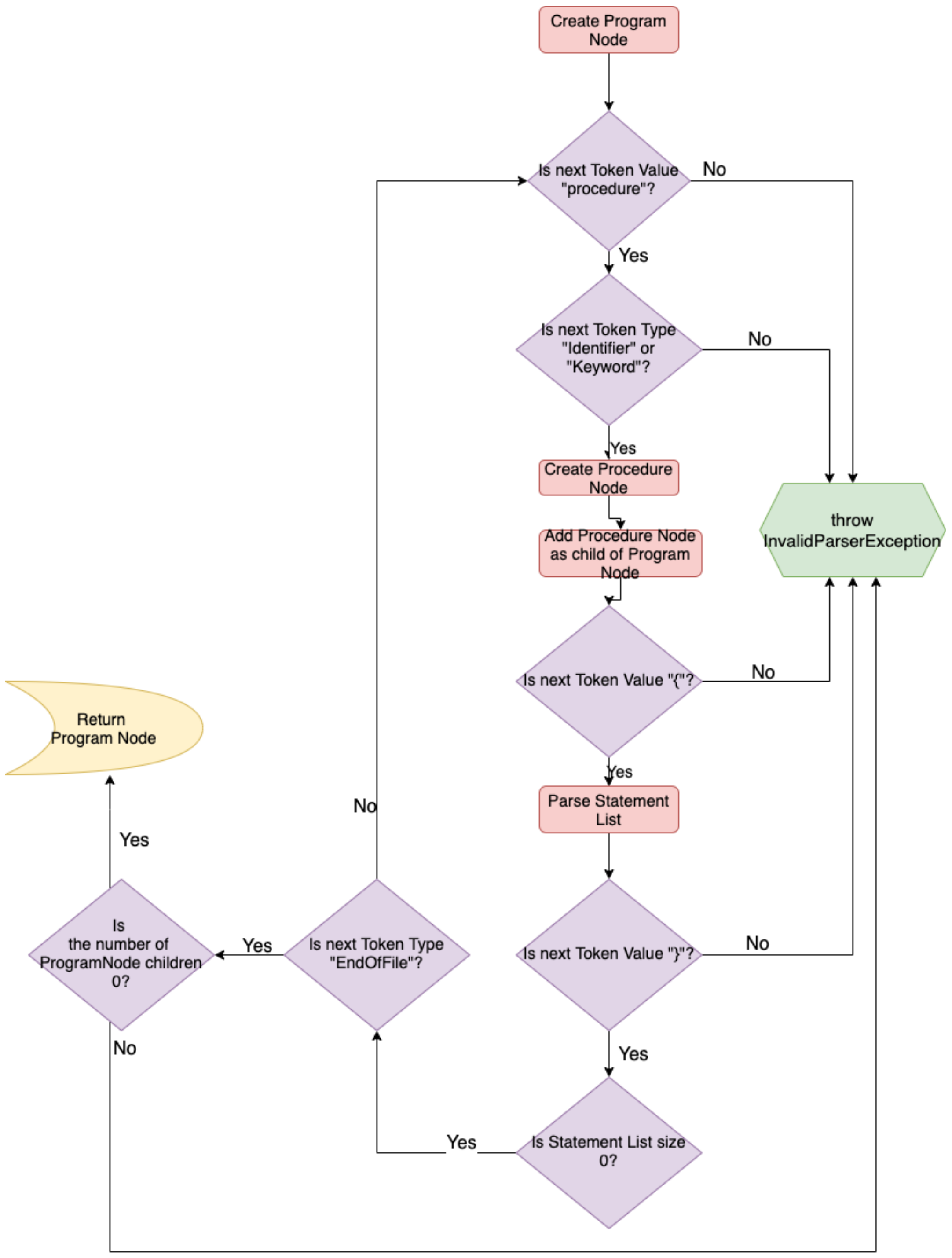

Create Program Node
Is next Token Value "procedure"?
No
Yes
Is next Token Type "Identifier" or "Keyword"?
No
Yes
Create Procedure Node
Add Procedure Node as child of Program Node
throw InvalidParserException
Is next Token Value "{"?
No
Yes
Parse Statement List
Return Program Node
No
Yes
Is the number of ProgramNode children 0?
Yes
Is next Token Type "EndOfFile"?
Is next Token Value "}"?
No
Yes
No
Yes
Is Statement List size 0?

Fig. 3.3.3.3-3: High-level Flowchart of parsing programs

Example
As an example, the Parser will parse to following program as follows:

```
procedure one {
     read x;
     print x;
}
```
Fig. 3.3.3.3-4: Sample source

With reference to the node objects depicted in Section 3.3.3.2, the Parser will evaluate tokens in sequential order as follows:

Grammar Rule	Token Value/s Checked	Parser Expectation
program	-	Parser makes call to parse program non-terminal
procedure	-	Parser makes call to parse procedure non-terminal
‘procedure’	“procedure”	Token Type “EndOfFile” not detected, Parser continues parsing. Token value matches ‘procedure’.
proc_name	“one”	Token Type is “Literal” or “Keyword”
‘{‘	“{‘	Token value matches ‘{’.
stmtLst		Parser makes call to parse stmtLst non-terminal
stmt	“read” “x” “;”	Parser makes call to parse stmt non-terminal Parser returns StmtNode stmt_1.
stmt	“print” “x” “;”	Parser makes call to parse stmt non-terminal Parser returns StmtNode stmt_2.
‘}’	‘}’	Token value matches ‘}’. Parser returns vector<StmtNode> object denoted as stmt_lst, containing stmt_1 and stmt_2. As stmt_lst has a size of 2, an error is not thrown. Parser returns ProcedureNode of value “one” and child node stmt_lst, denoted as one_node.
-	“EOF”	“Token Type “EndOfFile” detected, Parser ends parsing.

		Parser returns ProgramNode with child one_node. As ProgramNode has 1 child, an error is not thrown.

Fig. 3.3.3.3-5: Sequential process of parsing the Sample Source in Fig. 3.3.3.3-4

(4) Parsing Ambiguous Grammar

We define non-terminals to have ambiguous grammar rules if the rule contains the “|” meta symbol, which indicates that the non-terminal can be defined by more than one ruleset. Examples of such non-terminals include stmt and cond_expr.

The parsing logic for unambiguous grammar rules is straightforward as we only need to consider one case, making it trivial to sequentially map token to grammar rule from left-to-right. However, this may not be possible for ambiguous non-terminals, especially if the different rulesets differ wildly from each other in terms of number and types of symbols.

Example: Statements

The stmt non-terminal maps to 6 other non-terminals: if, while, call, assign, print and read. We must create checks to determine which statement type we are pasing. This is done by checking the value of the next sequence of Tokens prior to parsing the actual statement. For instance, to detect the assign non-terminal, the Parser checks if the second token in the Token Stream relative to the current position has a value of “=”. To detect other statement types, the Parser checks if the Token value matches the first keyword symbol of the statement type’s grammar rule. Once the statement type has been determined, the appropriate internal method will be called for parsing. As an example, if the current Token value is “print”, the Parser assumes it is parsing a print statement, and will call the method to parse the print non-terminal. The following flowchart depicts the sequence for statement parsing and validation:

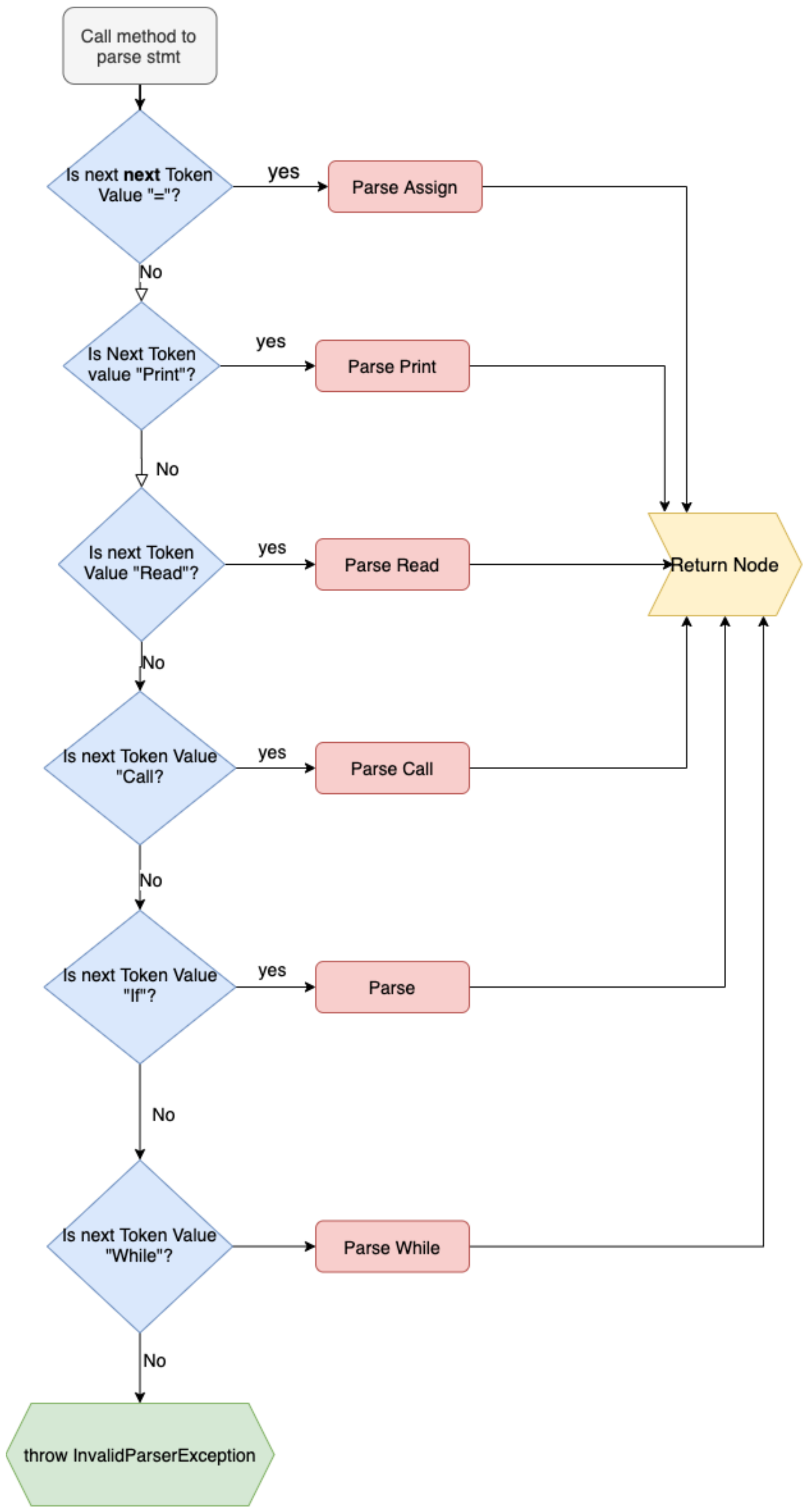

Call method to parse stmt
Is next **next** Token Value "="?
yes
Parse Assign
No
Is Next Token value "Print"?
yes
Parse Print
No
Is next Token Value "Read"?
yes
Parse Read
Return Node
No
Is next Token Value "Call?
yes
Parse Call
No
Is next Token Value "If"?
yes
Parse
No
Is next Token Value "While"?
Parse While
No
throw InvalidParserException

Figure 3.3.3.3-6: Flow Chart for parsing stmt non-terminal

The check for Assign statements must occur before checking for other statement types. This is to handle cases where the variable on the left side of the Assign statement is a Keyword, such as:

```
procedure main {
if = 1;
}
```
Fig. 3.3.3.3-7: Sample Source

Assume the check for assign occurs after the check for if. Since there is a valid mapping between Token value “if” and grammar rule keyword “if”, the Parser would incorrectly attempt to parse an if statement. To avoid this issue, the Parser must always check for Assign statements first.

Example: Conditional Expression

When parsing the cond_expr non-terminal, we must take note of three cases:

- Case 1 : The Conditional Expression has a conditional operator of “!”.
- Case 2: The Conditional Expression only consists of one relational expression.
- Case 3: The Conditional Expressions contain a left-side relational expression that begins with an open parenthesis “(“.
- Case 4: The Conditional Expression has a conditional operator that is not “!”.

The following flowchart shows the validation for each case:

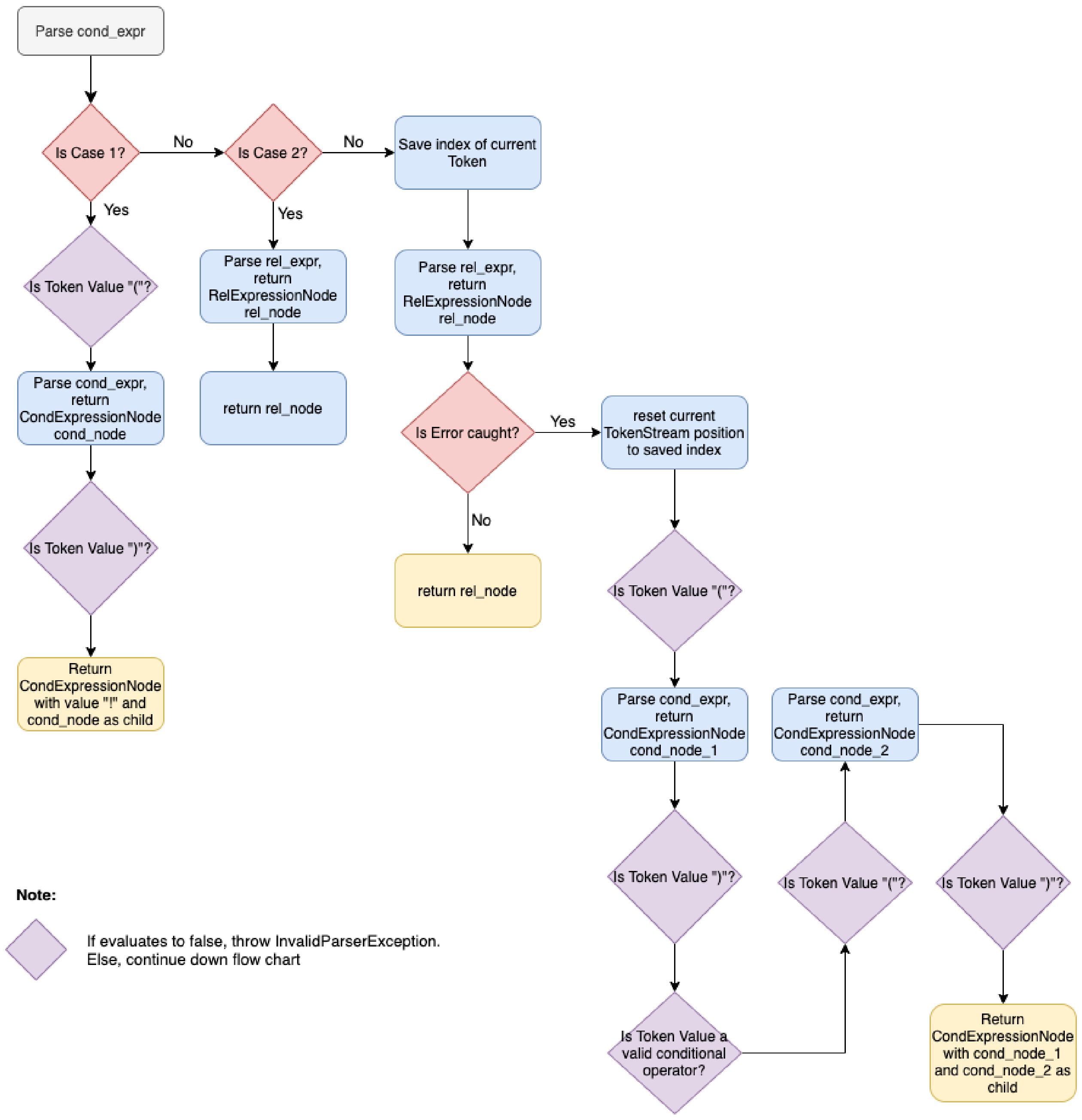

Parse cond_expr
Is Case 1?
No
Is Case 2?
No
Save index of current Token
Yes
Is Token Value "("?
Yes
Parse rel_expr, return RelExpressionNode rel_node
Parse rel_expr, return RelExpressionNode rel_node
Parse cond_expr, return CondExpressionNode cond_node
return rel_node
Is Error caught?
Yes
reset current TokenStream position to saved index
Is Token Value ")"?
No
return rel_node
Is Token Value "("?
Return CondExpressionNode with value "!" and cond_node as child
Parse cond_expr, return CondExpressionNode cond_node_1
Parse cond_expr, return CondExpressionNode cond_node_2
Note:
If evaluates to false, throw InvalidParserException. Else, continue down flow chart
Is Token Value ")"?
Is Token Value "("?
Is Token Value ")"?
Is Token Value a valid conditional operator?
Return CondExpressionNode with cond_node_1 and cond_node_2 as child

Fig. 3.3.3.3-8: Flowchart of parsing cond_expr non-terminal

Case 3 is not a direct rule set of cond_expr grammar. At first glance at the cond_expr grammar rules, it seems we are able to omit Case 3 and parse in the following manner:

1. If the Token value matches "!", assume Case 1 instance .
2. If the Token value does not match "(", assume Case 2 instance .
3. If the above conditions fail, assume Case 4 instance.

However, consider the following conditional expression:

```
procedure main {
if( (x+1)+2 < 5 ) then {..} else {...}
}
```

Figure 3.3.3.3-9: Sample Source

The Parser cannot tell that the first open parenthesis of the conditional expression belongs to the relational expression, and will incorrectly assume that this is a Case 4 instance instead of Case 2. Thus, we must include Case 3 to accommodate this edge case.

To check for Case 3, we first save the index of the current Token in the Token Stream to allow for backtracking. We let the Parser assume that the first "(" is part of a rel_expr, and parses accordingly. If an error is caught, the Parser will reset the Token Stream position to the saved index, and proceed to parse Case 4.

(5) Parsing Arithmetic Expressions with Pratt Parser

The Pratt Parser handles infix expression parsing and operator precedence through recursive calling and assigning binding power (BP) to operators based on their precedence. The binding power of an operator determines how "strongly" it is bound to its surrounding operands. The BP assigned to arithmetic operators are as follows:

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

For explanation purposes, we will refer to each recursive call as `Pratt_[x]`, where x is the recursive level of the call, starting from 0. Each `Pratt_[x]` is initialized with an initial BP value, which we will denote as "Method BP". For `Pratt_[0]`, the Method BP is initialized to 0.

For any given `Pratt_[x]`, the algorithm is as follows:

1. Let the AST being built by `Pratt_[x]` be `ast`. Depending on the characteristics of the next token, the root node of `ast` will be initialized differently:
    a. **If the next token value is "(" ,** `Pratt_[x+1]` is called and the return value is set as the root node. The Parser will then expect a next Token Value of ")" after the recursive call is complete.
    b. **Else,** the Parser expects a token type of "Identifier", "Keyword" or "Literal". In this case, the generated `VariableNode` or `ConstantNode` is set as the root node.
2. The Parser enters a **loop** whose conditions are (1) the next token value must be a valid arithmetic operator, and (2) Op BP > Method BP where "Op BP" denotes the BP of the operator. If these conditions are fulfilled, the following steps are performed:
    a. The Parser generates a corresponding `OpNode`, denoted as `op_node`.
    b. `Pratt_[x+1]` is recursively called and initialized with Op BP as the Method BP. The return value is denoted as `right_ast`.
    c. Assign `ast` as the left subtree of `op_node` and `right_ast` as the right subtree of `op_node`. Logically, `op_node` becomes the root node of `ast`.
    d. Repeat the loop.
3. When the Parser breaks out of the loop, `Pratt_[x]` returns `ast`.

If the Parser's expectations are not met at any point, an error will be thrown.

<u>Example</u>

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

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

Figure 3.3.3.3-10: Sample Source

The Parser will evaluate tokens in sequential order as follows:

Current Token Value	**Current Token Type**	**Op BP**	**Method BP**	**Parser Behaviour**	**Parser Expectation**
"="	Operator	-	0	The Parser calls Pratt_[0]	The next Token Type must be "Identifier", "Keyword" or "Literal"
"4"	Literal	-	0	A VariableNode `var_4` is created with a value of "4".	Check if the next Token Value is "+","-","/","*" or "%". If true, check if the OP Bp is greater than the Method BP.
"*"	Operator	2	0	A OpNode `op_times` is created with a value of "*" As the Op BP is greater than the Method BP, the Parser makes a recursive call Pratt_[1]	The next Token Type must be "Identifier", "Keyword" or "Literal"
"x"	Identifier	-	2	A VariableNode `var_x` is created with a value of "x".	Check if the next Token Value is "+","-","/","*" or "%". If true, check if the current Token BP is greater than the Expression BP.

“+”	Operator	1	1	The Op BP <= Method BP, the Parser breaks out of a recursion level and returns `var_x`. `var_4` is attached to `op_times` as the left child node. `var_x` is attached to `op_times` as the right child node. A OpNode `op_plus` is created with a value of “+”. As the Op BP is greater than the Method BP, the Parser makes a recursive call Pratt_[1] .	The next Token Type must be “Identifier”, “Keyword” or “Literal”
“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 Expression BP.
“;”	Separator	-	0	The Parser acknowledges that the expression has ended, breaks out of the current recursion and returns `var_3`. `op_times` is attached to `op_plus` as the left child node. `var_3` is attached to `op_plus` as the right child node. `op_plus` is returned.	-

Fig. 3.3.3.3-11: Sequential process of parsing the Sample Source in Fig. 3.3.3.3-10

3.3.3.4. Parser 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. 3.3.3.4-1: 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. 3.3.3.4-2: Sample Parser Exception

3.3.4. Design Considerations for Front End

3.3.4.1. Considerations for General Design

(1) Move Design Extractor to PKB

Criteria
The decision of whether to move the Design Extractor to PKB is influenced by the decided core responsibilities of the Parser. When choosing the goal of the Parser, we prioritised high cohesion. As mentioned previously, the goal of the Parser is to check structural validity, thus the output or behaviour of the parser should be strongly tied to this purpose.

Chosen Decision: Move Design Extractor to PKB
Our current design sees that the Design Extractor is moved to the PKB, and setting the end-goal of the Front-End Parser to be generating an AST.

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.

Alternative: Design Extractor is part of Front-End Parser
The alternative core responsibility 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
We believe that the Parser should live up to its namesake and parse only. 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.

(2) Implementing the AST

Criteria
Our criteria for whether to implement the AST depended on the ease of implementation, whether sufficient development resources were available for implementation, and whether the inclusion of an AST would help to achieve the Parser's goal of checking structural validity.

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.

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 as it may lead to unexpected implementation difficulties..

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, easing the potential difficulty of implementation.

As there is also no time constraints for the processing of the SIMPLE source in Iteration 2, any impacts to time and memory can be ignored as well.

3.3.4.2. Design Considerations for Tokenizer: Token Stream Data Structure Choice

Criteria

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.

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 while tokenization is done in Tokenizer class.

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.

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.

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.

3.3.4.3. Design Considerations for Parser: Handling Left Recursion of SIMPLE Expression

There is left recursion in SIMPLE arithmetic expressions, which prevents it from being parsed by the chosen Top Down Recursive Descent Parser. To alleviate this issue, we have chosen to implement a second Pratt parser that is capable of parsing infix expression as is.

Criteria

When choosing how to best handle the left recursion in Simple arithmetic expressions, we prioritize maintaining the original structure of the AST to avoid traversal complications. We also look at ease of implementation and availability of implementation resources.

Chosen Decision: Pratt Parser

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.

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.

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. 3.3.4.3-1: 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.

3.4. PKB

3.4.1. Overview of PKB

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.

The PKB is also responsible for checking for SIMPLE Program errors not covered by the SIMPLE CSG, namely cyclic procedure calls or procedure and variables with the same names.

For abstraction purposes, the PKB has been decomposed into 2 subcomponents - the **Design Extractor** and the **PKB Storage**.

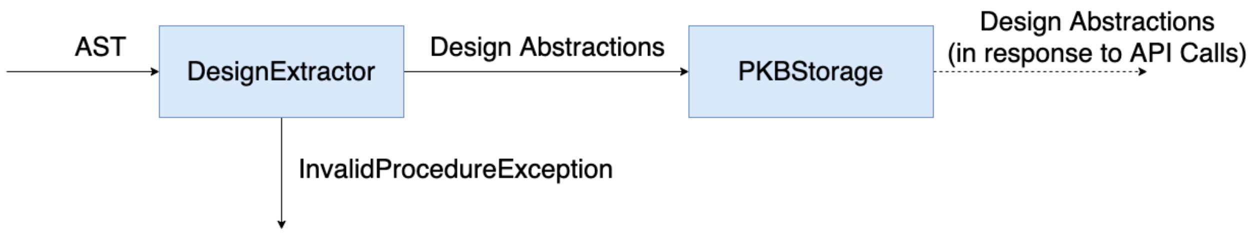

AST
DesignExtractor
Design Abstractions
PKBStorage
Design Abstractions (in response to API Calls)
InvalidProcedureException

Fig. 3.4.1-1: Overview on PKB Architecture

The process to extract and store Design Abstractions are as follows:

1. The SPA Controller passes the AST data structure to the Design Extractor.
2. The Design Extractor takes in the AST and traverses it to extract design abstractions.
3. The Design Extractor calls the PKB Storage methods to store the design abstractions in internal data structures.

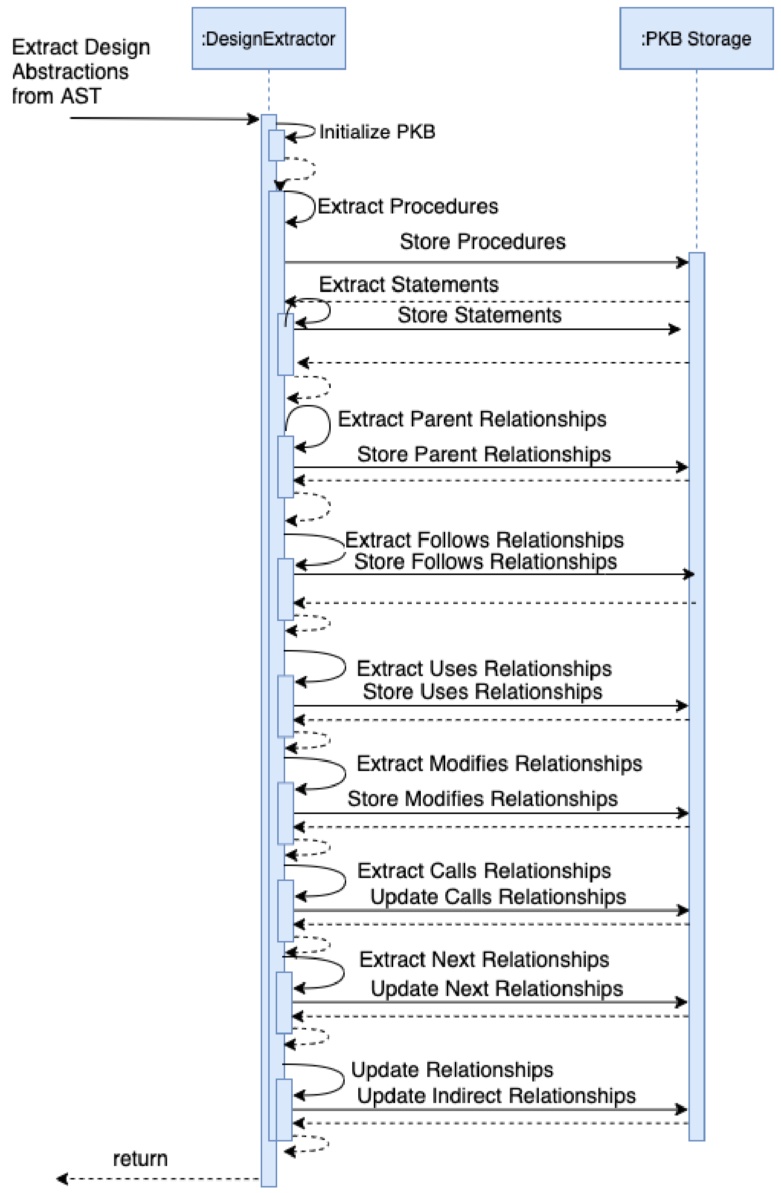

Extract Design Abstractions from AST
:DesignExtractor
:PKB Storage
Initialize PKB
Extract Procedures
Store Procedures
Extract Statements
Store Statements
Extract Parent Relationships
Store Parent Relationships
Extract Follows Relationships
Store Follows Relationships
Extract Uses Relationships
Store Uses Relationships
Extract Modifies Relationships
Store Modifies Relationships
Extract Calls Relationships
Update Calls Relationships
Extract Next Relationships
Update Next Relationships
Update Relationships
Update Indirect Relationships
return

Fig. 3.4.1-2: Sequence Diagram for initializing PKB

3.4.2. Design Extractor

3.4.2.1. Overview of Design Extractor

The Design Extractor is responsible for the extraction of design abstractions. It receives an AST data structure from the SPA Controller and traverses the AST to extract relationships. It then stores these relationships in the PKB's internal data structures.

3.4.2.2. Implementation for Design Extractor

(1) AST Traversal

The Design Extractor traverses the AST through the use of the AST API. Given a TNode object, the design extractor may attempt to call the following methods:

AST Overview: The 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_TREE_NODE 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.
TREE_NODE* 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_TREE_NODE 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_TREE_NODE 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.
TREE_NODE* getVariable() ***Description:*** If the node object has a design entity type of ‘Read’,’Print’, ‘Call’ or ‘If’, return the variable node associated with the node object. Else, throw an error.
TREE_NODE* 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.
TREE_NODE* 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.
TREE_NODE* 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.

Fig. 3.4.2-1: AST API

The API allows the Design Extractor to traverse the AST data structure without knowledge of AST data structure implementation such as class names and class variables. This is done by mapping the API calls to DesignEntity enum types rather than node classes, such that only nodes of certains types may call certain APIs. For instance, a node of type 'Program' may call `getProcedures()`, but may not call `getStatementList()`. In the case where a node of an incorrect type attempts to call an illegal method, an InvalidNodeException will be thrown. The error was implemented for debugging purposes to detect flaws in the Design Extractor logic during development.

This implementation method was chosen as the DesignEntity enum types are highly related to the SIMPLE design entities and non-terminals, which are fixed. Thus, the type names are unambiguous and unlikely to change during development. Furthermore, the Design Extractor checks Node object type using switch/case statements, which is trivial to modify in the unlikely event of a renaming, addition or removal of an enum type.

Example

To demonstrate the AST API usage, we will look at how the API calls are used to traverse the following source program to get to the constant node "4":

```
procedure main{
      read x;
      while(y == 1) {
            z = 4+i;
      }
}
```

Fig. 3.4.2.2-1: Sample Source

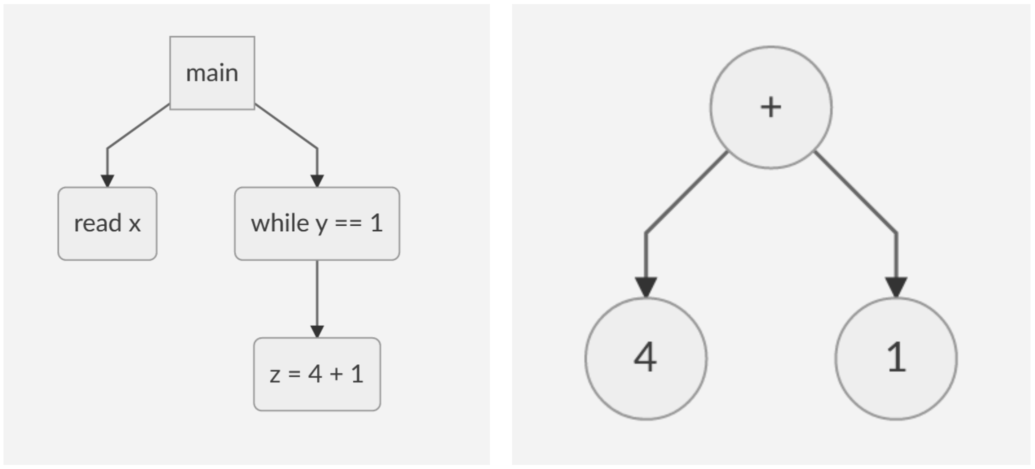


Fig. 3.4.2.2-2: Simplified AST generated from Fig. 3.4.2.2-1

API Calls	Current Node Location
getProcedures()[0]	main
getStatementList()[1]	while (y == 1)
getStatementList()[0]	z = 4 + 1
getExpression()	+ Node
get LeftNode()	"4" Node

Fig. 3.4.2.2-3: API Calls to traverse AST

(2) Number of Passes

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	Statement Number Variables Constants Procedures If Statements Assign Statements Read Statements While Statements Print Statements Call statements	Follows Parent Modifies (Direct) Uses (Direct) Calls (Direct) Next
2		Follows* Parent* Calls* Uses (Indirect) Modifies (Indirect)

Fig. 3.4.2.2-4: Design abstractions extracted per pass

(3) Extracting Relationships

We traverse the AST using in order traversal, and at each step of the traversal the parent node is set as the parent of children nodes. The Parent and the inverse relationships are stored in an unordered map with the statement number as the key for indexing the values.

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. 3.4.2.2-5 Sample source for subsequent figures

Example: Extracting Parent Relationships

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}.

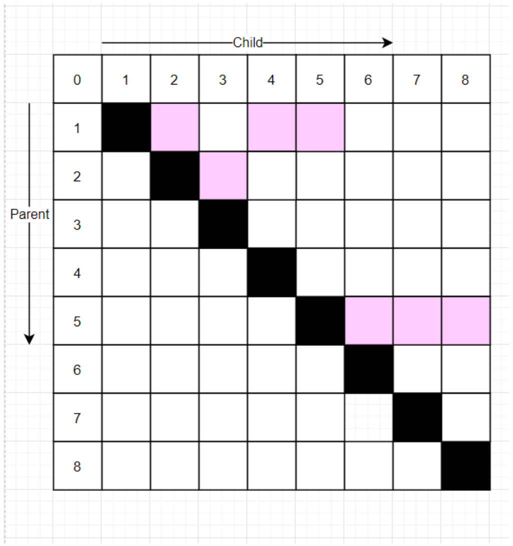


Fig. 3.4.2.2-6: Pink squares for parent relationship

Example: Extracting Follows Relationship

As we perform an in order traversal of the AST, each statement under the parent statement is given an index and we after all statements are assigned their statement numbers, we loop through the list of statements and assign each statement with a relationship with their preceding and proceeding statements. If either does not exist, we do not set any relationships. The Follows and inverse relationships are stored in unordered maps where the statement numbers act as they key for indexing the maps

<u>Example: Extracting Follows Relationship</u>

In the figure below we show a statement list containing 4 statements, we start assigning the follows relationship by taking the first statement, statement 1, we traverse back into the statement list and get the second statement, statement 2. We then set the Follows(1, 2) relationship, and move on to then next value. We traverse back to the statement list and obtain the third statement, statement 6. We set the Follows(2, 6) relationship and move on to the next value. We traverse back to the statement list and obtain the fourth statement, statement 7, and we assign the Follows(6, 7) relationship, we move back to the statement list and try to obtain the fifth statement, which does not exist and the process ends there. Until the next statement list is encountered

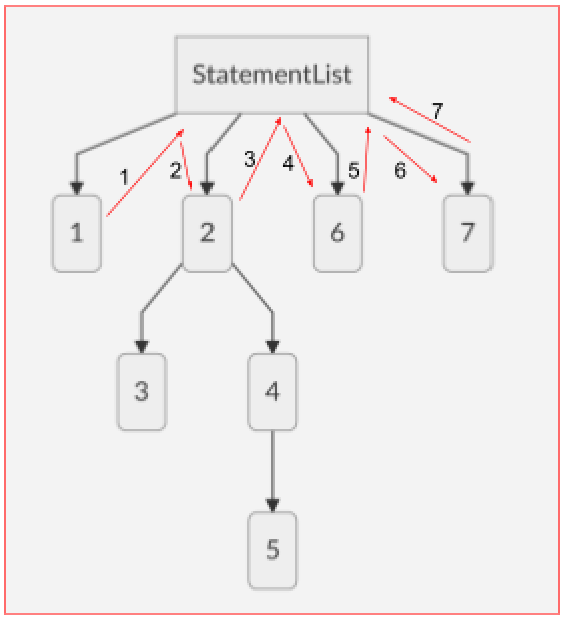


Fig. 3.4.2.2-7: Order of traversal when extracting Follows Relationships

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}

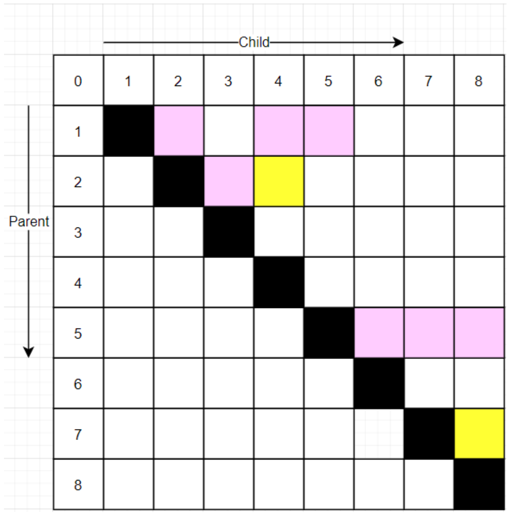


Fig. 3.4.2.2-8: Yellow squares for follows relationship

Example: Extracting 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

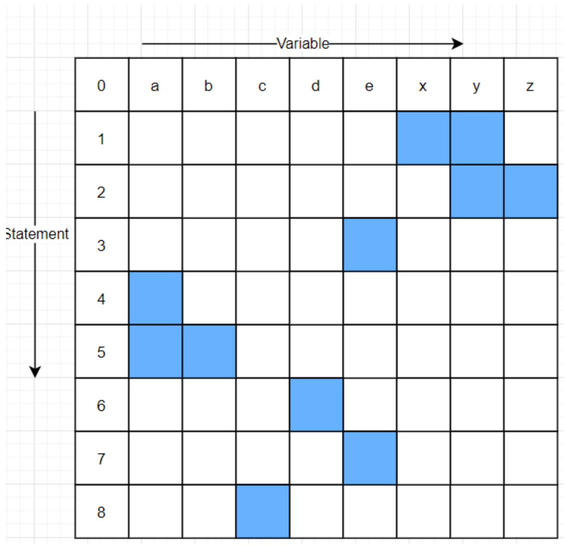


Fig. 3.4.2.2-9: Blue squares for primary uses relationship

Fig. 3.4.2.2-10 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. At the end of the second pass, we will have Fig. 3.4.2.2-11.

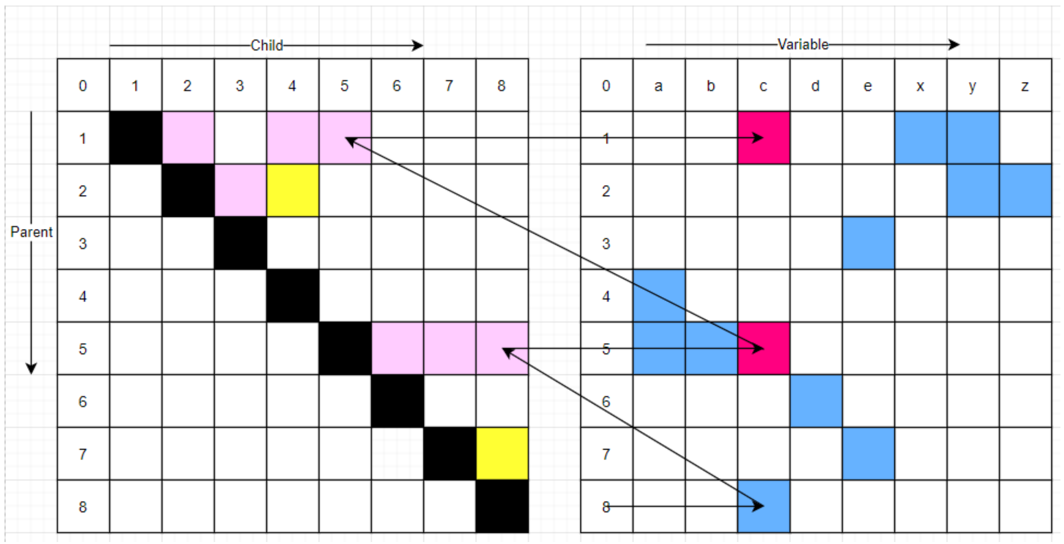
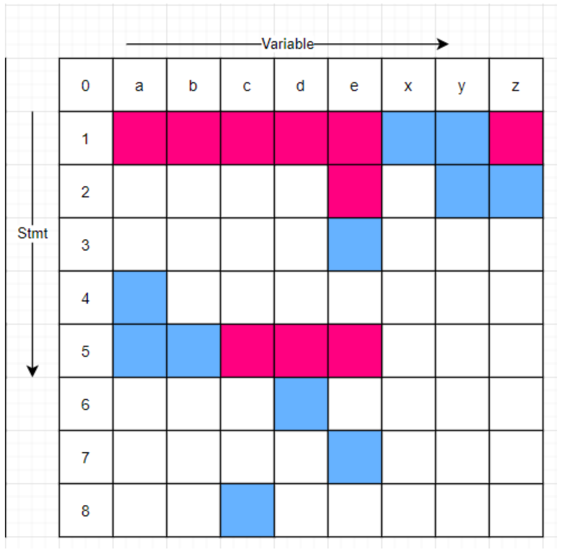


Fig. 3.4.2.2-10: Red squares for secondary uses relationship, arrows indicate the order of updates


Variable
Stmt

0	a	b	c	d	e	x	y	z
1	🟥	🟥	🟥	🟥	🟥	🟦	🟦	🟥
2					🟥		🟦	🟦
3					🟦			
4	🟦							
5	🟦	🟦	🟥	🟥	🟥			
6				🟦				
7					🟦			
8			🟦					

Fig. 3.4.2.2-11: Final uses table

<u>Example: Extracting Follows Relationship</u>
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. 3.4.2.2-12: Sample source for subsequent figures

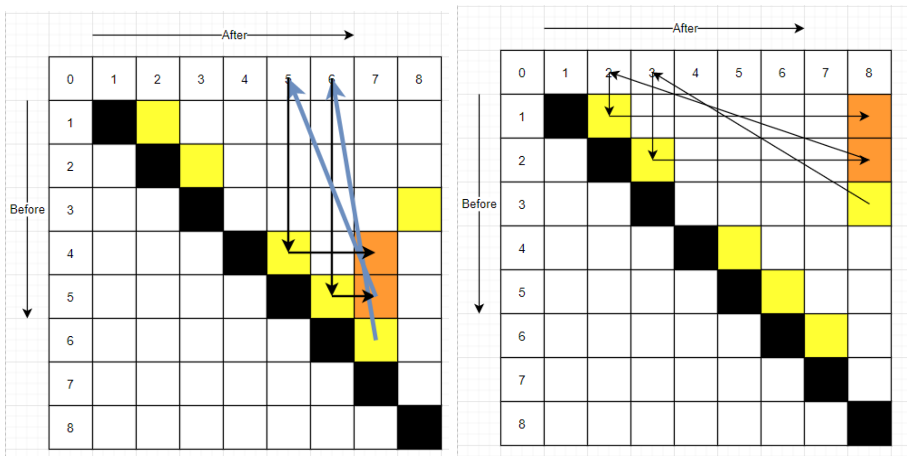
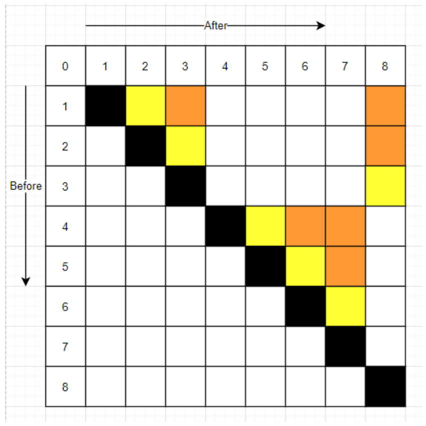


*Fig. 3.4.2.2-13: Updating of follows star relationship - Yellow squares for Follow, Orange squares for Follow**


After
Before

0	1	2	3	4	5	6	7	8
1	⬛	🟨	🟧					🟧
2		⬛	🟨					🟧
3			⬛					🟨
4				⬛	🟨	🟧	🟧	
5					⬛	🟨	🟧	
6						⬛	🟨	
7							⬛	
8								⬛

Fig. 3.4.2.2-14: Completed follows star table

3.4.2.3. PKB 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. 3.4.2.3-1: 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. 3.4.2.3-2: Sample source

3.4.3. PKB Storage

3.4.3.1. Overview of PKB Storage

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.

3.4.3.2. Data Structures for PKB Storage

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. 3.4.3-1 shows an example of an unordered map for the Next relationship that has been populated.

Previous(statement numbers)	Next(statement number(s))
1	2, 3, 4
5	6
6	7, 8, 15

Fig. 3.4.3.2-1: Example of a populated parent map

The following table shows the different types of tables and their purpose:

Stored Tables	Rationale
statement_master_list variable_master_list	References back to the node objects of each statement and variable so that we can access each without having to traverse the AST to look for the individual nodes
while _list if_list assign_list call_list print_list read_list	List of indexes so that we can quickly identify and/or loop through a single statement type to perform our extraction/evaluation algorithms
parent _list parent _star_list follows_list follows_star_list calls_list uses_list modifies_list	First order relationships that functions as a cache of evaluating different relationships, all stored relationships are small enough to be stored in the form of an unordered map allowing for quick retrieval of relationships without having to traverse the AST

Fig. 3.4.3.2-2: Subset of tables used for PKB Storage

3.4.3.3. Implementation for PKB Storage

(1) Inserting Values

For any relationship rel, and statements x, y in statements in SIMPLE source, if rel(x, y) we call the API insert function for the list or table insert(x, y) and we call the API insert function for the inverse list or table insert(y, x).

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

Fig. 3.4.3.3-1: Example of inserting a parent relationship

(2) Retrieving Values

For any relationship rel, and statements x, y in statements in SIMPLE source, to retrieve all y values that satisfies the relationship rel(x, y) where x is the chosen value. We look through the list or unordered map using the statement number of x as the index and the element paired with the corresponding index will be the indices of all possible y values that satisfies the relationship.

(3) Pattern Matching

To identify a statement contains a match for a pattern we first retrieve the expression from the corresponding statement. We then use the parser to construct an AST from our pattern string. Once we have both ASTs to compare we start by checking if the root node of the trees are of equal values. If they are we then compare the both left subtrees to see if they fully match using the same algorithm called recursively. If both left subtrees are equal we then compare both right subtrees to see if they fully match using the same algorithm called recursively. If all 3 comparisons evaluate to true, we then conclude that the pattern fully matches the expression in the statement. To identify a partial match, we simply also perform a full match at each subtree of the expression AST we get from the statement. If any subtree returns a full match with our pattern we conclude that there is a partial pattern match.

Example: Pattern Matching
For this example, consider the following AST for the equation:

*(((3 + 4) - 2) * (5 / 6)) + 1 and (3 + 4 - 2) * (5 / 6) + 1*

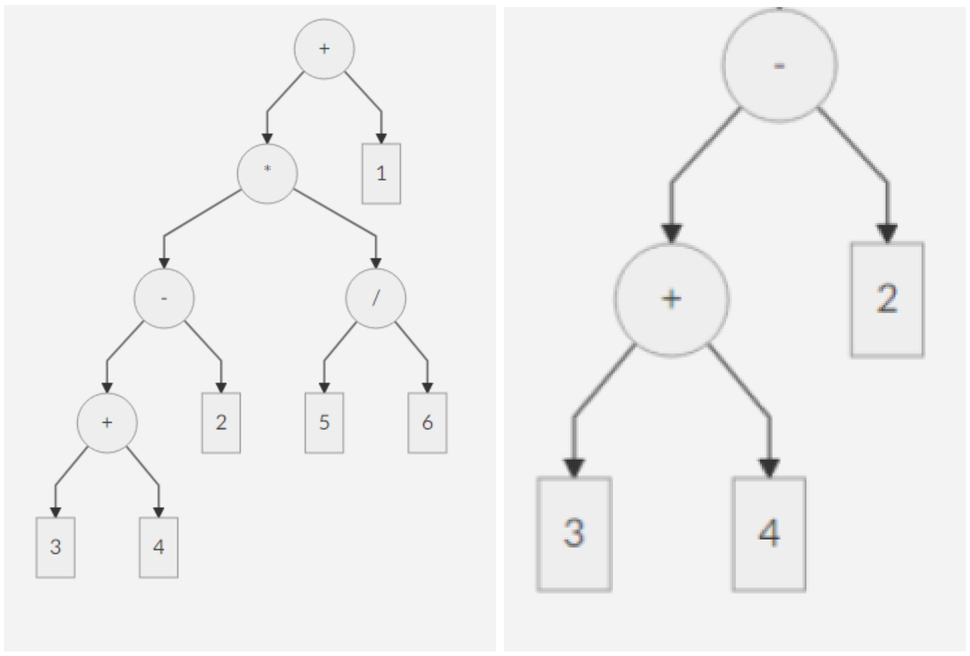


*Fig. 3.4.3.3-2: The AST for the equation : (((3 + 4) - 2) * (5 / 6)) + 1 and (3 + 4 - 2) * (5 / 6) + 1 and subtree _"(3 + 4) - 2"_*

To search for the subtree we start from the root node, we compare the the "+" node with our "-" node in the subtree and see that it doesn't match so we move on to compare the children. We look at the right node, we compare the "1" node with our subtree's root and find that they are not equal. We move on to the left child. We compare the "*" node and our "-" node and they are not equal, we compare the right child of "*" and find that "/" and "-" are not equal, we move on to compare the children of "/" both "5" and "6" are not equal to "-", we start comparing the left child of "*" and we find that both the child and our subtree match at the root, we now compare the right child of both and find that they are both "2", we compare the right child and find that they are both "+" nodes so we move on to compare the left and right child of both "=" nodes which are equal. So we conclude that there exists a subtree *_"(3 + 4) - 2"_* in our target equation.

(4) Next and Next*

Next statements are obtained during the parsing of the AST when initializing the PKB. Each proceeding statement that is inserted to the PKB passes its statement number to all proceeding statements that satisfies the Next relationship. The Next and Previous statement pairs are then stored in an unordered map. The unordered map of Next and Previous statements makes up a CFG of the simple program which is then used to compute Next*/Affects/Affects*.

Example: Next and Next*
For this example, we will extract the Next relationships for the following SIMPLE source:

```
procedure main {
    A = a; // 1
    while (b == b) { //2
        C = c; //3
        D = d; //4
    }
    if ( e == e) then { //5
        F = f; //6
        G = g; //7
    } else {
        H = h; //8
    }
    I = i; //9
}
```
Fig. 3.4.3.3-3: Sample Source

The order of extraction and storing of Next relationships is as follows:

Statement Number Being Read	**Statement Number for Next Relationship**	**API called**
1	2	next_list.insert (1, 2)
2	3, 5	next_list.insert (2, 3) next_list.insert (2, 5)
3	4	next_list.insert (3, 4)
4	2	next_list.insert (4, 2)
5	6, 8	next_list.insert (5, 8) next_list.insert (5, 8)
6	7	next_list.insert (6, 7)
7	9	next_list.insert (7, 9)
8	9	next_list.insert (7, 9)
9		

Fig. 3.4.3.3-3: Next Relationships extracted

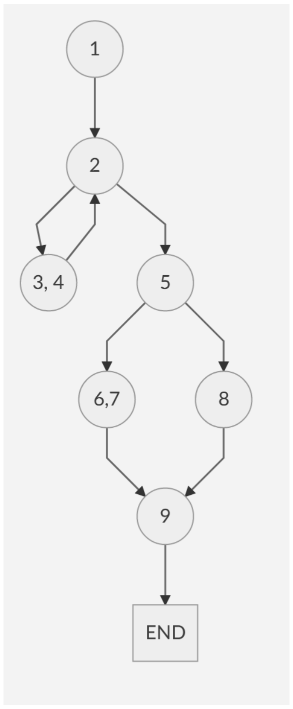


Fig. 3.4.3.3-4: Resulting CFG

Each statement is a node in the CFG, each insert call inserts a directed vertex from one node to another. A Next* relationship is then calculated using a DFS graph traversal algorithm and all points reachable from the targeted node is added into the result for the Next* relationship by the PKB, if a path from statement x to statement y exists, then the Next* relationship exists.

(5) Calls and Calls*

At each call statement, the PKB adds the procedure that contains the statements into the Calls table using the function insertCalls(procedure, statement). The relationship is stored in an unordered map which can then be used to evaluate Calls*. The PKB evaluates Calls* by treating the Calls relationship as a directed graph where the edges are the procedures and the nodes are the relationships, we then use an adapted Bellman-Ford Edge relaxation algorithm to update the directed graph. We have each edge treated as a negative value, and at each relaxation step the total distance is equal to the negative the length of connected nodes to each edge. Each edge is then appended with any new procedure encountered. We carry out the relaxation step for each procedure.

We follow it up with another relaxation step, if any value is updated we know that there are recursive calls, so we throw an error. Otherwise all procedures' Calls relationships are updated.

(6) Affects

Affects is calculated using the CFG generated by processing Next. Starting at an assign statement, the PKB starts a DFS of the CFG. At the start the modified variable is obtained using the getModifiedVar function from the PKB. At each next node, if the node uses the modified variable and is an assign statement, this is checked using the isUsesVar function from the PKB. If the node uses the modified variable, it is added into the result vector. The PKB then checks if the node modifies the modified variable, this is checked by thePKB using the isModifiedVar function from the PKB. If the node modifies the modified variable the DFS is terminated, otherwise the DFS continues from the next nodes in the CFG. When the DFS is complete, the result will be all the statements affected by the original assign statement.

Example: Affects

For this example, we will extract the Affects relationships for the following SIMPLE source:

```
procedure main {
    a = a; // 1
    while (b == b) { //2
        b = a; //3
        a = a; //4
    }
    if ( e == e) then { //5
        a = a; //6
        G = g; //7
    } else {
        b = a; //8
    }
    I = a; //9
    while (x == y) { //10
        if (a == b) { //11
            i = I; //12
        } else {
            read a; //13
        }
    }
    b = a; //14
}
```

Fig. 3.4.3.3-5: Sample Source

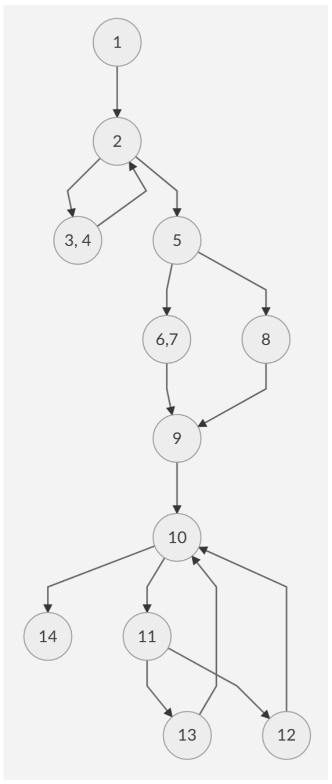


Fig. 3.4.3.3-6: CFG of SIMPLE source

Starting from Statement 1 to find all Affects(1, a) relationships the following is the traversal and API calls:

Statement Number Being Read	Value Modified	Value Used	API called
1	a	a	startDFS()
2			continue()
3	b	a	answer.insert(3) continue()
4	a	a	stopDFS()
5			continue()
6	a	a	answer.insert(6) stopDFS()
8	b	a	answer.insert(8) continue()
9	l	a	answer.insert(9) continue()
10			continue()

11			continue()
12	i	l	continue()

14	b	a	answer.insert(14)
			continue()

13	a		stopDFS()

3.4.4. Design Considerations

3.4.4.1. Design Considerations for Design Extractor: Number of Passes for Design Extraction

Criteria
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.

Chosen Decision: Multi-pass Design Extraction
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.

Alternative 1: 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.

3.4.4.2. Design Considerations for PKB Storage

(1) Internal Data Structure

Criteria

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.

Chosen Decision: Unordered Maps

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.

Alternative 1: 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.

(2) Pattern Storage

Criteria

Patterns necessarily have to be compared in the form of an AST, to facilitate the comparison of patterns there are two possible storage methods.

Chosen Decision: Store as AST

```
getStmtFullMatch(LHS, RHS)
```

Fig. 3.4.4.2-1: Function call for getting match

The method of storage is to simply retain the constructed AST and store pointers to each individual statement that could be a candidate for a match. These pointers are stored in unordered maps as explained in the previous section.

Pros: The time taken for matching will be faster since there isn't a need to reconstruct the AST of the pattern before matching.

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

Alternative 1: Store as Strings
Storing the patterns as strings, we then do a simple regex before deciding if we need to construct a tree to do the comparison.

Pros: Regex will be faster than comparing sub trees, and no additional computation will be required for patterns that definitely don't match. Full matches will also be faster since there isn't a need to use an AST to identify matches in that case.

Cons: The need to construct an AST means that the processing time will be longer.

Justification for choosing AST
When comparing the two situations, the AST implementation provides a more consistent time regardless of the input source code and pattern query. If we were to have the program meet the timing restrictions, slower edge cases for string storage are more likely to cause a timeout.

3.5. Query Processor

3.5.1. Overview of Query Processor

The Query Processor component is divided into **six** major 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. **QueryResult**, which contains the table that stores the intermediary results from the evaluations of individual clauses.
6. **QueryResultProjector**, which formats the raw results and returns the result to I/O.

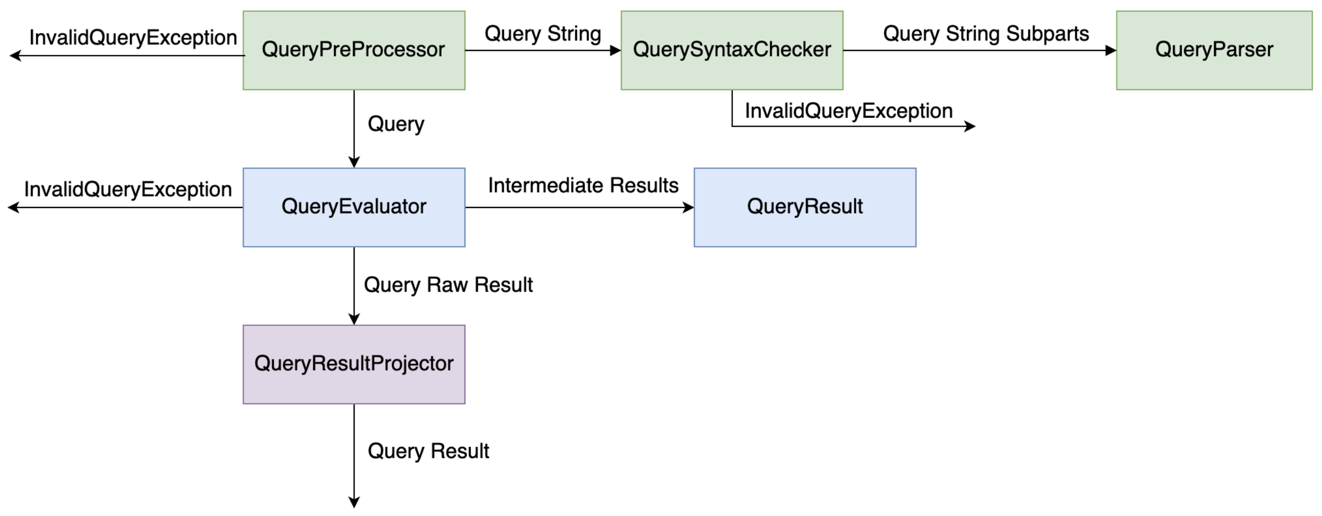


Fig. 3.5.1-1: Production Pipeline for Query Processor

3.5.2. Query Preprocessor (QPP)

3.5.2.1. Overview of Query Preprocessor

`QueryPreprocessor` packages an input query string into a `Query` object, which stores relevant information about synonyms, synonym types, clauses and their respective arguments that will be used in evaluation of the query later.

3.5.2.2. Data Structures for Query Preprocessor

(1) Query

When information about the query is extracted by `QueryPreprocessor`, it is stored as a `Query` object. The class diagram for the `Query` class is as follows:

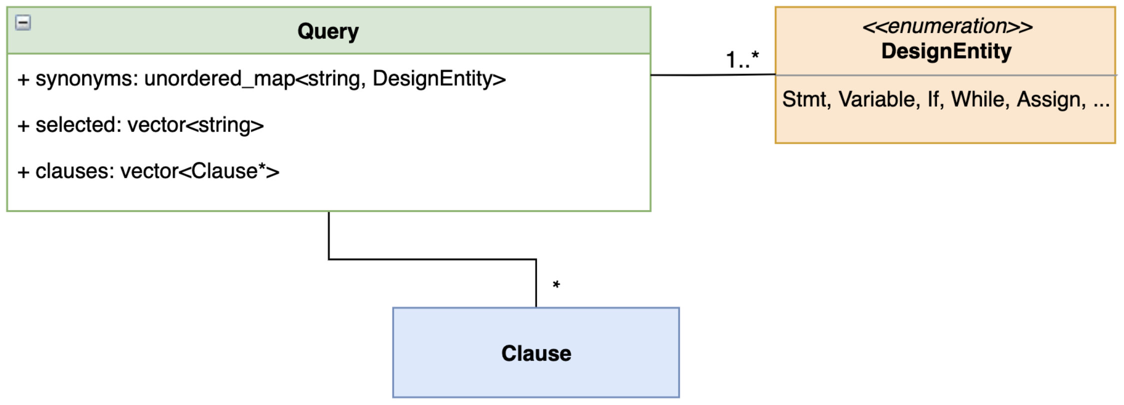

Query
+ synonyms: unordered_map<string, DesignEntity>
+ selected: vector<string>
+ clauses: vector<Clause*>
1..*
<<enumeration>>
DesignEntity
Stmt, Variable, If, While, Assign, ...
*
Clause

Fig. 3.5.2.2-1: Class diagram for `Query`

As shown in the class diagram above, `Query` has three fields:

1. **`synonyms`**: stores mappings of synonyms and their corresponding design entities that are declared in the declaration part of query
2. **`selected`**: the tuple or `BOOLEAN` that is to be selected and returned in results
3. **`clauses`**: relational clauses, pattern clauses and/or with clauses that appear in the query. The number of clauses ranges from 0 to any number.

Example

Given a sample query:

```
assign a; while w; variable v;
Select <a,v> such that Uses(w,v) pattern a(v,_) with a.stmt# = 3
```

Fig. 3.5.2.2-2: Sample query to illustrate `Query` object structure

The `Query` object will have the following structure:

```
Query {
    synonyms: {
        {“a”, DesignEntity::Assign},
        {“w”, DesignEntity::While},
        {“v”, DesignEntity::Variable}
    }
    selected: [“a”, “v”]
    clauses: [
        UsesClause(“w”, “v”),
        PatternAssign(“a”, “v”, “_”),
        WithClause(“a.stmt#”, “3”)
    ]
}
Note: {} stands for map and syn_entity pairs in map, [] stands for vector
```

Fig. 3.5.2.2-3: `Query` object structure for sample query

(2) Clause

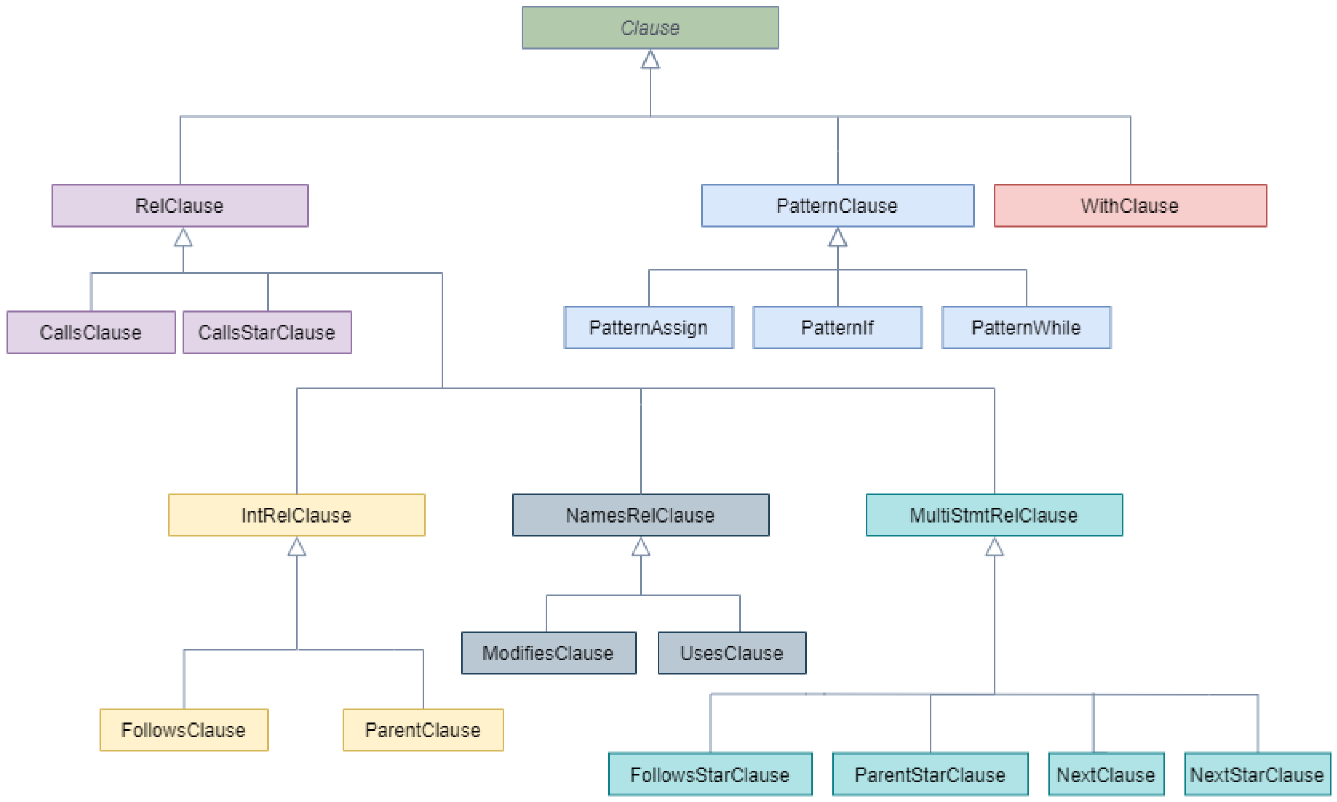


Fig 3.5.2.2-4: Clause Class Diagram

Clause

This is the parent class that the `RelClause` (Relational Clauses), `PatternClause` and `WithClause` 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 `validate`, a function that performs semantic checks to determine the `DesignEntity` type of a received parameter.

Relational Clause

This class inherits from `Clause`, and it contains methods that are common among relational clauses. With the same design consideration separating relational and pattern clauses, we see that the relational clauses can be further abstracted based on their similarities. We have thus created 3 clauses that inherit from `RelClause`, where various clauses in turn inherit from based on their suitability to a clause. This significantly reduced the amount of redundant code written as we saw in Iteration 1 that some clauses had almost the same logic of evaluation. Therefore, `Parent` and `Follows` clauses inherit from `IntRelClause` (Integer Relational Clause), `Uses` and `Modifies` clauses inherit from `NamesRelClause` (Names Relational Clause), `Follows*`, `Parent*`, `Next` and `Next*` inherit from `MultiStmtRelClause` (Multi-Statement Relational Clause), and `Calls` and `Calls*` inherit directly from `RelClause`. This can be seen from Fig 3.5.2.2-4.

Pattern Clause

This class inherits from `Clause` and has another attribute: `pattern_syn`, which is the synonym before parenthesis. `PatternClause` has three children classes: `PatternAssign`, `PatternWhile`, and `PatternIf` to handle different types of pattern matching.

With Clause

This class inherits from `Clause` and stores two other variables, `l_attribute` and `r_attribute`, which contains the `Attribute` enum type associated with the `left` and `right` parameters respectively. The `Attribute` variables are extracted from the respective parameters and assigned during validation.

3.5.2.3. Implementation for Query Preprocessor

(1) Syntactic & Semantic Validation with Regex Matching

Overview

In the process of packaging the `Query` object, `QueryPreprocessor` performs various syntactic and semantic checks on the input query string. First, it calls `QuerySyntaxChecker` to perform the following 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 clause, relational clause groups, pattern clause groups, with clause groups
3. No irrelevant characters between subparts (such as “and” between different types of clause groups)

Next, after constructing the `synonyms` map, `QueryPreprocessor` performs the following semantic checks by itself:

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

Finally, after constructing `clauses` vector, `QueryPreprocessor` calls `Clause::validate` on each `Clause` in the vector to perform additional semantic checks that are specific to each type of `Clause`. Specifically, `validate` checks that the arguments are of the correct types. For example, the left argument of `UsesClause` cannot be a variable or “_”.

Validation Rules

Here is a comprehensive list of validation rules, checked by respective regexes / `QueryPreprocessor`’s semantic check / `Clause::validate` semantic check. The different situations where an `InvalidQueryException` would be thrown, and the corresponding exception messages are documented below:

Invalid Query Type	Checked By	Example	Exception Message
Invalid query structure, e.g. no “Select” keyword,	REGEX_QUERY	`“stmt s; blablabla such`	`"Invalid Query Structure"`

no semicolons, etc.		`that"`
		`"stmt s Select s"`	

The selected synonym has not been declared	QPP semantic check (check against syn_entity_map)	`"stmt s; Select v"`	`"Selected synonym not declared."`
A synonym used in clauses has not been declared	QPP semantic check (check against syn_entity_map)	`"stmt s: Select s such that Follows (s, p)"`	`"Synonym used in clauses not declared."`
Duplicate synonyms appear in the query	QPP semantic check (check against syn_entity_map)	`"print p; procedure p; Select p"`	`"Duplicate synonyms in query."`
The relational clause is invalid	REGEX_REL_CLAUSE	`"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/ Next/Next*/Calls/Calls*)	REGEX_FOLLOWS, REGEX_FOLLOWS_STAR, REGEX_PARENT, REGEX_PARENT_STAR, REGEX_MODIFIES_S, REGEX_MODIFIES_P, REGEX_USES_S, REGEX_USES_P, REGEX_NEXT, REGEX_NEXT_STAR, REGEX_CALLS, REGEX_CALLS_STAR	`"assign a; variable v; Select a such that xcfdsf(odasfa)"`	`"Rel clause syntax is wrong."`
The pattern clause has wrong syntax	REGEX_PATTERN, REGEX_PATTERN_ASSIGN, REGEX_PATTERN_WHILE, REGEX_PATTERN_IF	`"assign a; variable v; Select a pattern sdfasdff(dsaf3ere wt)"`	`"Pattern clause syntax is wrong."`
The with clause has wrong syntax	REGEX_WITH	`"assign a; variable v; Select a with dsfasdf"`	`"With clause syntax is wrong."`
The parameter types are invalid for a specific clause	Clause::validate	`"assign a; variable v; Select a such that Follows(a,v)"`	`"Invalid param type for Follows clause."`

Fig. 3.5.2.3-1: Different Types of Invalid Queries, Validators and Exception Messages

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. 3.5.2.3-2: Sample query

While performing semantic check, `QueryPreprocessor` detects duplicate synonyms “p” in the declaration. An `InvalidQueryException` would be thrown with the following message:

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

Fig. 3.5.2.3-3: Sample Exception

Implementation

Regex matching is used for query syntax validation. Specifically, `QuerySyntaxChecker` uses a regex representing valid syntax to match the input string, and raises an exception if no matching substring is found.

Example

Here are some examples of regexes we have written for declaration and Follows clause. The detailed mapping of PQL grammar rules to regex is demonstrated in the declaration regex `REGEX_DECL`.

```
const string REGEX_DECL =
R"((\s*(stmt|read|print|call|while|if|assign|variable|constant|procedur
e)\s+[A-Za-z][A-Za-z0-9]*\s*(\s*,\s*[A-Za-z][A-Za-z0-9]*)*\s*;\s*)+)";
//corresponding PQL grammar rules:
declaration : design-entity synonym ( ‘,’ synonym )* ‘;’
design-entity : ‘stmt’ | ‘read’ | ‘print’ | ‘call’ | ‘while’ | ‘if’ |
‘assign’ | ‘variable’ | ‘constant’ | ‘procedure’
synonym : IDENT
IDENT : LETTER ( LETTER | DIGIT )*

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. 3.5.2.3-4: Example of regexes used for syntax validation

We will now show how syntax validation is performed, using `REGEX_FOLLOWS` as an example:

```
Follows(1,2) //valid Follows clause, matches REGEX_FOLLOWS

Follows(1,”v”) //invalid Follows clause, the parts highlighted do not
match:
R"(\s*Follows\s*\(\s*(\s*[A-Za-z][A-Za-z0-9]*|_|[0-9]+)\s*,\s*([A-Za-z]
[A-Za-z0-9]*|_|[0-9]+)\s*\))"
```

```
//This part requires the second argument to be either a synonym, a
wildcard(_), or an integer, so quotation marks should not appear here.
```
Fig. 3.5.2.3-5: Example of how `REGEX_FOLLOWS` validates Follows clause

If the query fails to match any predefined regex, it is considered syntactically invalid. `QuerySyntaxChecker` will throw an `InvalidQueryException` with a detailed message about the error.

(2) Preprocessing Sequence

Overview
In this section, we will elaborate on the entire process from taking in a raw query string to the successful creation of the `Query` object. There are five major steps in this process:

1. Overall query structure validation;
2. Get declarations and create `synonyms` map;
3. Extract selected tuple / `BOOLEAN` from Select clause;
4. Get relational / pattern / with clause groups, separate them into individual clauses, and create `clauses` vector;
5. Construct the `Query` object from the three components obtained in steps 2-4.

Throughout the process, we adopt a top-down approach of syntax validation. First, we check the overall query structure. Then, after getting each subpart, we do separate checks on more detailed syntaxes. This will be explained in more details in the "Implementation" section below.

Implementation
We will now elaborate each step of query preprocessing in more details:

Step 1: Overall Query Structure Validation
Upon taking in a query string, check the overall query structure using a regex, and raise an exception immediately upon failing the check.

```
const string REGEX_QUERY =
R"((\s*(stmt|read|print|call|while|if|assign|variable|constant|procedur
e)\s+[A-Za-z][A-Za-z0-9]*\s*(\s*,\s*[A-Za-z][A-Za-z0-9]*)*\s*;\s*)+\s*S
elect\s+(BOOLEAN|([A-Za-z][A-Za-z0-9]*|[A-Za-z][A-Za-z0-9]*.(procName|v
arName|value|stmt#))|<\s*([A-Za-z][A-Za-z0-9]*|[A-Za-z][A-Za-z0-9]*.(pr
ocName|varName|value|stmt#))(\s*,\s*([A-Za-z][A-Za-z0-9]*|[A-Za-z][A-Za
-z0-9]*.(procName|varName|value|stmt#)))*>)\s*.*)";

Declarations are highlighted in green,
Select clause is highlighted in yellow,
Everything after Select clause is highlighted in pink.
```
Fig. 3.5.2.3-6: Regex to check overall query structure

In this stage, we only check declarations and the Select clause, as these two are compulsory components of a query. As this is only a high-level check, we do not enforce syntax validation on anything that follows the Select clause, thus we simply use "`.*`" to match it.

Example with Activity Diagram

Sample (valid) query:

```
assign a; while w; variable v; if ifs;
Select <a,v> such that Uses(w,v) and Follows(3,w) pattern a(v,_) and
ifs(v,_,_) with a.stmt# = 5
```

Figure 3.5.2.3-7: Sample Query used to illustrate Query Preprocessing Procedure

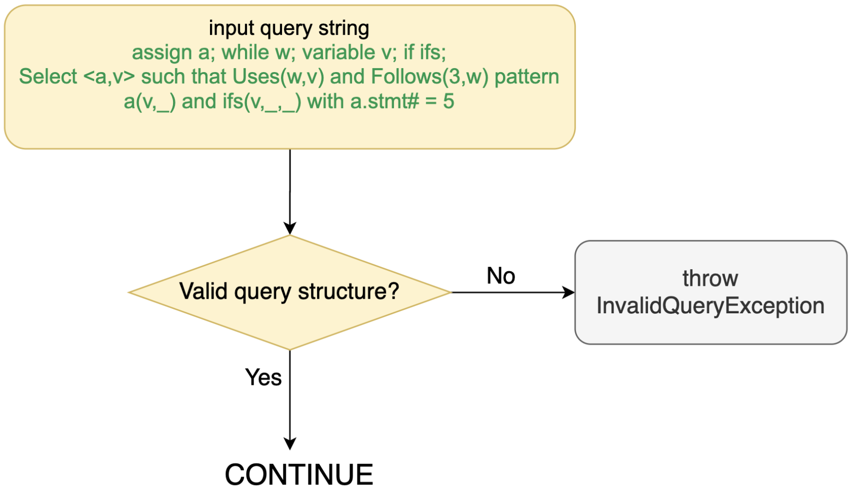

input query string
assign a; while w; variable v; if ifs;
Select <a,v> such that Uses(w,v) and Follows(3,w) pattern a(v,_) and ifs(v,_,_) with a.stmt# = 5
Valid query structure?
No
throw InvalidQueryException
Yes
CONTINUE

Figure 3.5.2.3-8: Overall query structure validation

Step 2: Get Declarations and Create `synonyms` Map

Once done checking the overall query structure, `QueryPreProcessor` continues to validate the syntax of declarations. If the check passes, it will proceed to parse the declarations and make the `synonyms` map, which stores synonyms and their corresponding types.

Example with Activity Diagram

Using the sample query *"assign a; while w; variable v; if ifs; Select <a,v> such that Uses(w,v) and Follows(3,w) pattern a(v,_) and ifs(v,_,_) with a.stmt# = 5"* from the previous section:

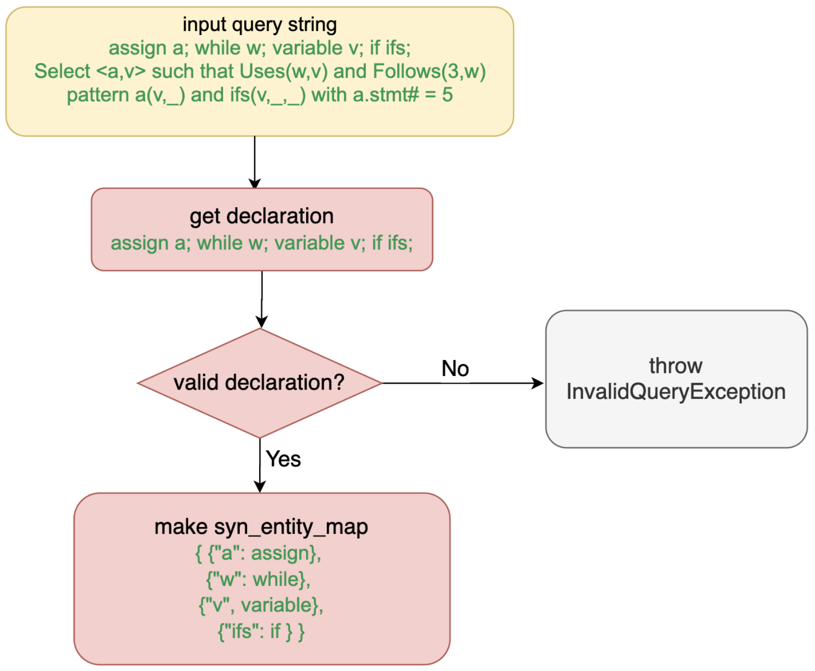

input query string
assign a; while w; variable v; if ifs;
Select <a,v> such that Uses(w,v) and Follows(3,w)
pattern a(v,_) and ifs(v,_,_) with a.stmt# = 5
get declaration
assign a; while w; variable v; if ifs;
valid declaration?
No
throw InvalidQueryException
Yes
make syn_entity_map
{ {"a": assign},
{"w": while},
{"v", variable},
{"ifs": if } }

Figure 3.5.2.3-9: Extract declarations and create syn_entity_map

Step 3: Extract Selected Tuple/BOOLEAN from Select clause

Once the `synonyms` map is created from parsing declarations, `QueryPreProcessor` proceeds to validate the syntax of the Select clause. If the check passes, it continues to parse selected synonyms/BOOLEAN, and store them in a vector.

Example with Activity Diagram

Using the sample query *"assign a; while w; variable v; if ifs; Select <a,v> such that Uses(w,v) and Follows(3,w) pattern a(v, _) and ifs(v, _ , _) with a.stmt# = 5"* from the previous section:

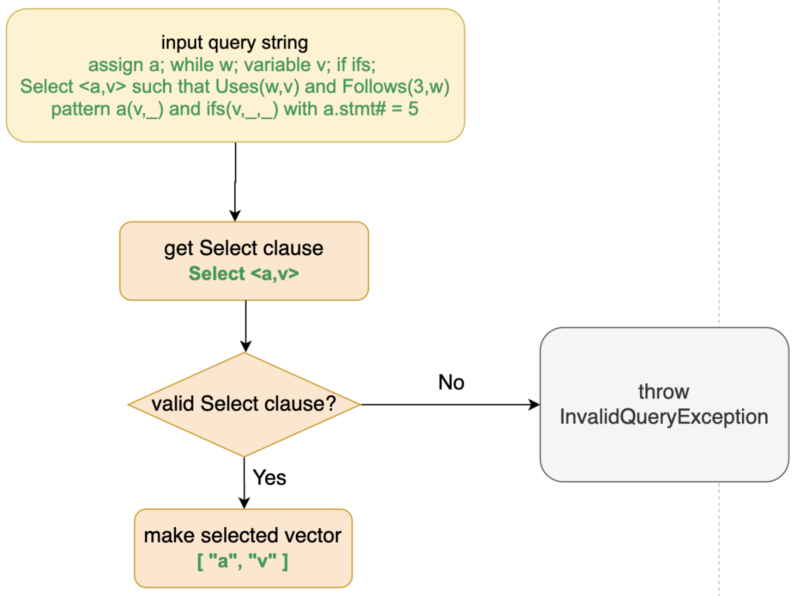

input query string
assign a; while w; variable v; if ifs;
Select <a,v> such that Uses(w,v) and Follows(3,w)
pattern a(v,_) and ifs(v,_,_) with a.stmt# = 5
get Select clause
Select <a,v>
valid Select clause?
No
throw InvalidQueryException
Yes
make selected vector
["a", "v"]

Figure 3.5.2.3-10: Extract selected tuple/BOOLEAN and make `selected` *vector*

Step 4: Get Clause Groups and Separate into Individual Clauses

Once the `selected` vector is created from parsing the Select clause, `QueryPreProcessor` proceeds to extract different types of clause groups from the remaining query, using regex matching. A clause group is multiple clauses of the same type (relational, pattern or with), connected with "and". The clause groups will then be split by "and" to get individual clauses. `QueryPreProcessor` then performs syntax validation on each individual clause using regex matching. After this, `QueryPreProcessor` constructs `Clause` objects from clause strings, and calls `Clause::validate` method of each individual clause to perform semantic check on argument types. Once all checks clear, `QueryPreProcessor` will create a vector of valid clauses to be packed into `Query` object.

Example with Activity Diagram

Using the sample query *"assign a; while w; variable v; if ifs; Select <a,v> such that Uses(w,v) and Follows(3,w) pattern a(v, _) and ifs(v, _ , _) with a.stmt# = 5"* from the previous section:

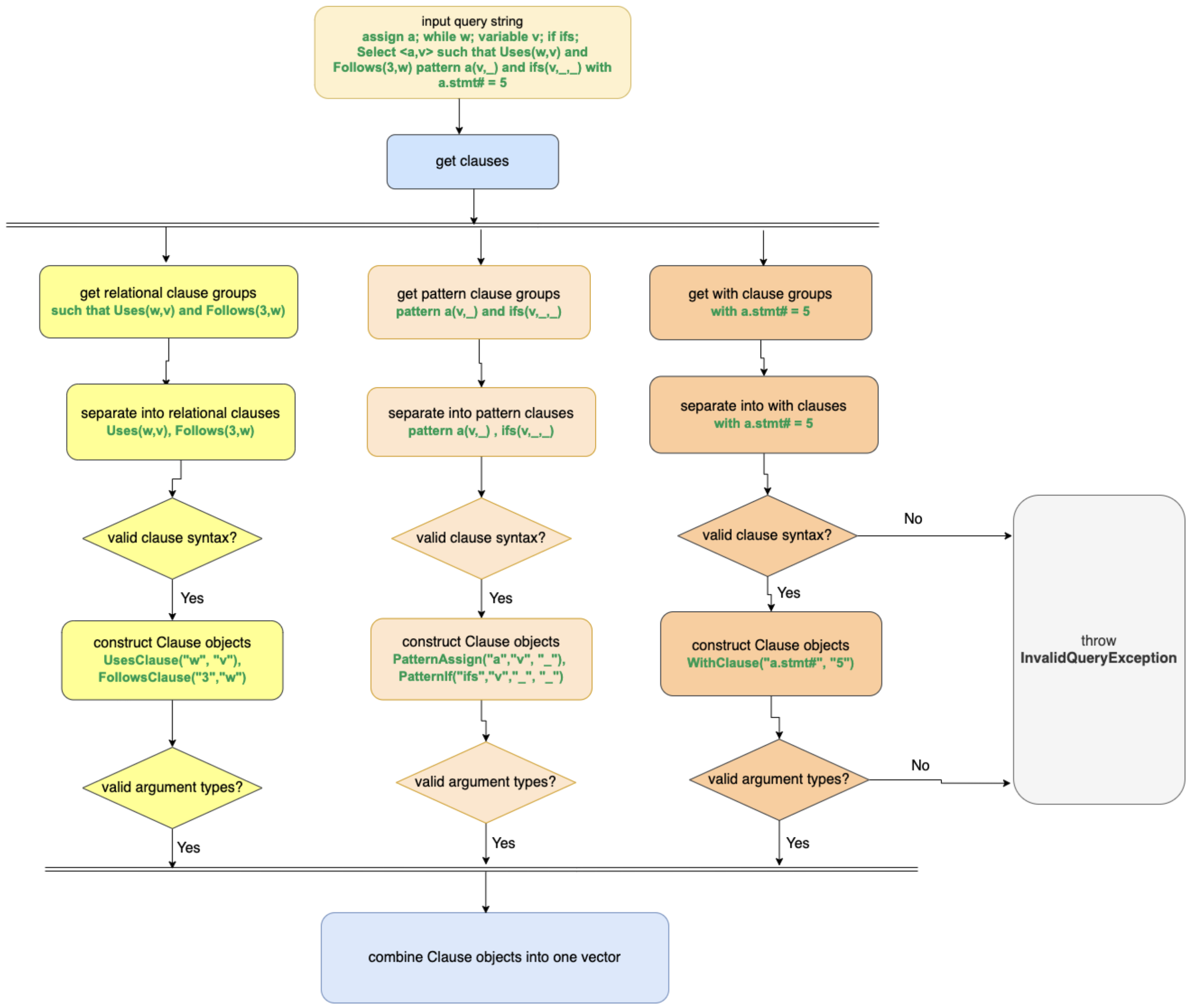

input query string
assign a; while w; variable v; if ifs; Select <a,v> such that Uses(w,v) and Follows(3,w) pattern a(v,_) and ifs(v,_,_) with a.stmt# = 5
get clauses
get relational clause groups
such that Uses(w,v) and Follows(3,w)
get pattern clause groups
pattern a(v,_) and ifs(v,_,_)
get with clause groups
with a.stmt# = 5
separate into relational clauses
Uses(w,v), Follows(3,w)
separate into pattern clauses
pattern a(v,_) , ifs(v,_,_)
separate into with clauses
with a.stmt# = 5
valid clause syntax?
Yes
No
construct Clause objects
UsesClause("w", "v"), FollowsClause("3","w")
PatternAssign("a","v", "_"), PatternIf("ifs","v","_", "_")
WithClause("a.stmt#", "5")
valid argument types?
throw InvalidQueryException
combine Clause objects into one vector

Figure 3.5.2.3-11: Extract clauses and make `clauses` vector

Step 5: Construct Query object from components

After the `synonyms` map, the `selected` vector, and the `clauses` vector have all been successfully created, these three components are combined into a `Query` object, which will then be passed to `QueryEvaluator` for further evaluation.

Shown below is the concrete structure of the `Query` object created from the sample query *"assign a; while w; variable v; if ifs; Select <a,v> such that Uses(w,v) and Follows(3,w) pattern a(v, _) and ifs(v, _ , _) with a.stmt# = 5"*:

```
Query {
    synonyms: {
        {"a", DesignEntity::Assign},
        {"w", DesignEntity::While},
        {"v", DesignEntity::Variable},
        {"ifs", DesignEntity::If}
    }
    selected: ["a", "v"]
    clauses: [
        UsesClause("w", "v"), FollowsClause("3", "w"),
        PatternAssign("a", "v", "_"), PatternIf("ifs", "v", "_", "_"),
        WithClause("a.stmt#", "3")
    ]
}

Note: {} stands for map and syn_entity pairs in map, [] stands for vector
```

Figure 3.5.2.3-12: Final `Query` object structure for sample query

3.5.3. Query Evaluator (QE)

3.5.3.1. Overview of Query Evaluator

There are several **sub-components** in QE:

- **`Query Evaluator`**: acts like a controller for the evaluation of all the relational/pattern/with clauses in the clause list in the query object. It also evaluates the Select clause after the evaluation of all relational/pattern/with clauses.
- **`Query Result:`** maintains a table of intermediary results from clauses that has been evaluated. It provides APIs for Clause classes to add and/or filter results in the table.
- **`Clause`** classes: relational/pattern/with clauses provides an evaluate() function to be called by Query Evaluator. It evaluates the result of itself and adds and/or filters the results in the Query Result table accordingly.

The **flow of query evaluation** is as following:

After successfully getting a `Query` object from Query Preprocessor, the SPA Controller creates a new `Query Result` object with an empty table. SPA Controller then calls the Query Evaluator (QE) and passes in the `Query` object and `Query Result` 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 and `Query Result` object. The clauses call relevant APIs provided by PKB and Query Result to retrieve the results and put them into the table accordingly. The details are discussed in [Section 3.5.3.3 part 4 - *Merging Results in Query Result Table*].

The following sequence diagram shows the overall logic in QE.

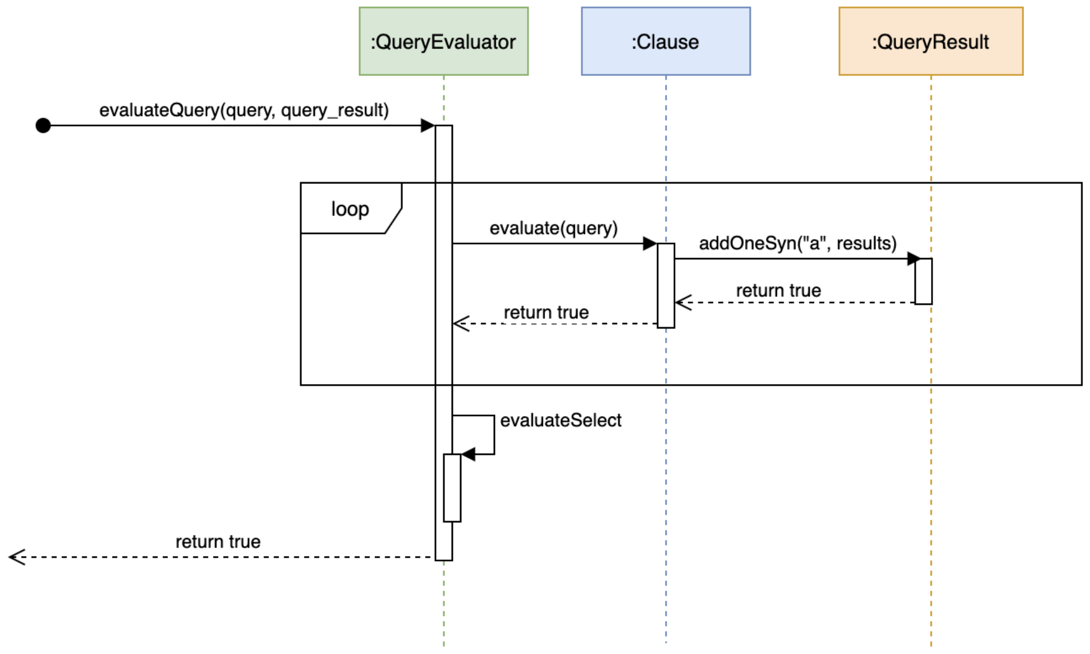


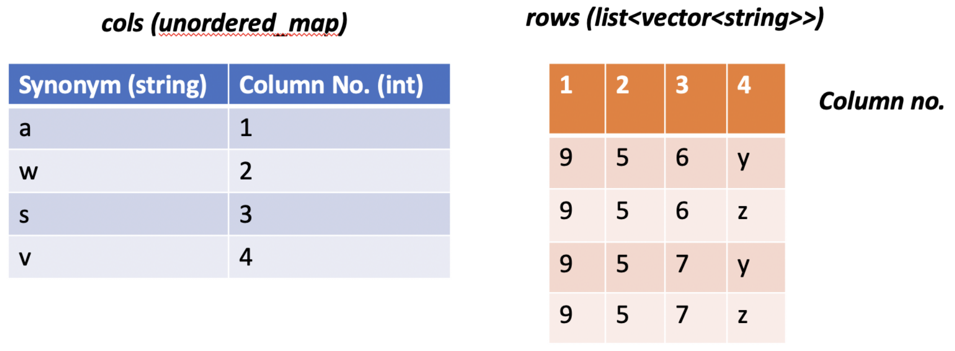
Fig 3.5.3-1: QE Sequence Diagram

3.5.3.2. Data Structures for Query Evaluator

(1) Query Result Class - The Intermediary Table

`Query Result` maintains the following attributes:

- **`unordered_map<string, int>cols`**: the map in which the key is the synonym name and the value is the column number representing that synonym. **`list<vector<string>>rows`**: the table of intermediary results. Each element in the list is a vector of results.



cols (unordered_map)

Synonym (string)	Column No. (int)
a	1
w	2
s	3
v	4

rows (list<vector<string>>)

1	2	3	4
9	5	6	y
9	5	6	z
9	5	7	y
9	5	7	z

Column no.

Fig. 3.5.3.2: Data structure of Query Table

`Query Result` provides the following APIs that are called by Clause object to modify the table content:

- **`bool addOneSyn(string syn, vector<string> &result)`**: to add a new column (new synonym) of results into the table by performing a cross-product with the existing rows
- **`bool addTwoSyn(string syn_1, string syn_2. Vector<pair<string, string>> &results)`**: to add two new column (two new synonym) of pairs of results into the table by performing a cross-product with the existing rows
- **`bool filterOneSyn(string syn, function<bool (string)> func)`**: given a synonym which is already in the table, delete the rows in which the value of that synonym does not satisfy the current clause using the function passed in.
- **`bool filerTwoSyn(string syn_1, string syn_2, function<bool (string, string)> func)`**: given a pair of synonyms which are already in the table, delete the rows in which the values of that synonym pairs do not satisfy the current clause using the function passed in.

The usage of the above APIs will be illustrated with an example in Section 3.5.3.3 part 4 - *Merging Results in Query Result Table.*

3.5.3.3. Implementation for Query Evaluator

This section discusses the implementation of each subcomponent in Query Evaluator in detail. Section (1) - (3) explains how the evaluate() function in `Clause` classes evaluates the clause independently, i.e. assuming itself is the only clause in the query, by checking the type of parameters and calling relevant PKB APIs.
Section (4) explains how `Clause` objects call relevant `Query Result` APIs and modifies the table content.

(1) Evaluation of Relational Clauses

In Iteration 2, we have added the implementations for Calls, Calls*, Next and Next* as part of relational clauses. Additionally, UsesP and ModifiesP have already been implemented in Iteration 1 as part of our bonus features. For explanation purposes, we will use the Follows clause as an example. All the other clauses are similar in terms of structure and principle. 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	PKB APIs called
Integer	Integer	Follows(3,4)	isFollows
Integer	Wildcard	Follows(3,_)	getFollows
Integer	Synonym	Follows(3,a)	getFollows, getStmtType
Wildcard	Integer	Follows(_,4)	getFollowedBy
Wildcard	Synonym	Follows(_,a)	getAllMatchedEntity, getFollowedBy
Wildcard	Wildcard	Follows(_,_)	getFollowsListSize
Synonym	Integer	Follows(w,9)	getFollowedBy, getStmtType
Synonym	Wildcard	Follows(s,_)	getAllMatchedEntity, getFollows
Synonym	Synonym	Follows(a,a1)	getAllMatchedEntity, isFollows

Fig. 3.5.3.3-1: 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 other clauses having different combinations of cases from the above table. Refer to the API Documentation in ______ for more details about the API functions.

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

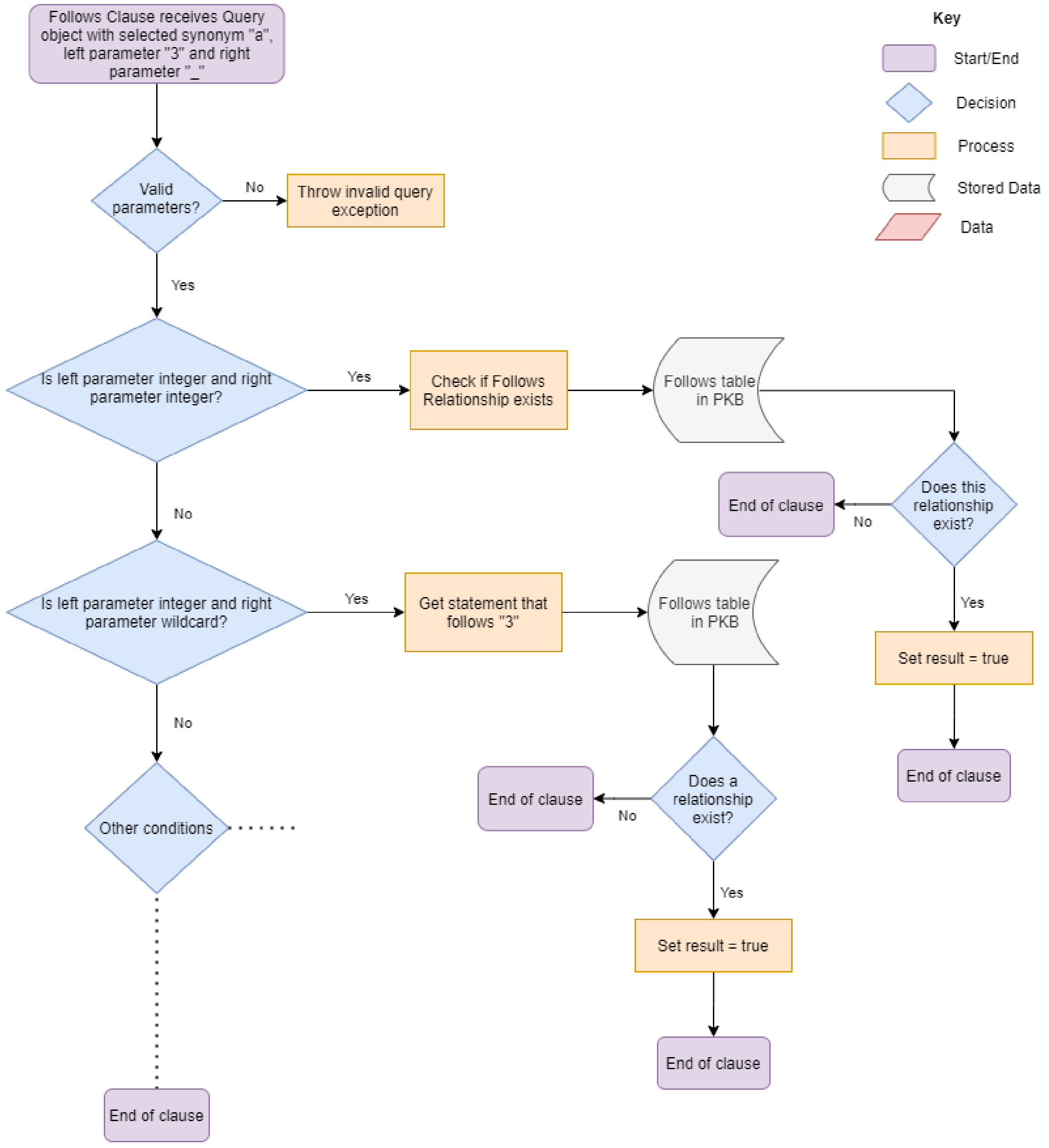

Follows Clause receives Query object with selected synonym "a", left parameter "3" and right parameter "_"
Valid parameters?
No
Throw invalid query exception
Yes
Is left parameter integer and right parameter integer?
Yes
Check if Follows Relationship exists
Follows table in PKB
Does this relationship exist?
No
End of clause
Yes
Set result = true
End of clause
No
Is left parameter integer and right parameter wildcard?
Yes
Get statement that follows "3"
Follows table in PKB
Does a relationship exist?
No
End of clause
Yes
Set result = true
End of clause
No
Other conditions
End of clause
Key
Start/End
Decision
Process
Stored Data
Data

Fig. 3.5.3.3-2: Follows evaluate() Activity Diagram

Example 1

In this example, we will illustrate the evaluation of the Follows clause with the following query:

"while w; assign a;
Select a such that Follows(3,w)"

The left parameter is an integer and the right parameter is a synonym, in this case, the synonym refers to while statements. Therefore, we need to evaluate whether there is (1) a statement that follows 3, and (2) the statement that follows 3 is a while statement.

First, a call, getFollows(3), is made to the PKB. This call checks if there is any Follows relationship between 3 and a statement that follows 3. If there exists a value (statement) in the Follows table that follows statement 3, that value will be returned. If this value does not exist, the function will return 0. In this case, assume that the value 4 is returned, indicating that statement 4 follows statement 3. The evaluate function then calls getStmtType(4) to obtain the statement type of 4. If statement 4 is a while statement, the clause is satisfied.

<u>**Example 2**</u>
In this example, we will illustrate the evaluation of the Follows clause with the following query:

"assign a;
Select a such that Follows(_,a)"

The left parameter is a wildcard and the right parameter is a synonym, in this case, the synonym refers to assign statements. Therefore, we need to evaluate all the assign statements that follow another statement.

First, a call, `getAllMatchedEntity(assign type)`, is made to the PKB. This call retrieves the statement numbers of all the assign statements. We then need to check for each of these statements, whether they follow another statement, thus the call getFollowedBy(assign statement number). If there exists a value (statement) in the Follows table that is followed by that statement number, the value will be returned. If this value does not exist, the function will return 0. All the assign statement numbers that return a value other than 0 will be added to the temporary result list.

(2) Evaluation of Pattern Matching

This section discusses how `Pattern Clauses` evaluate independently by calling PKB APIs.

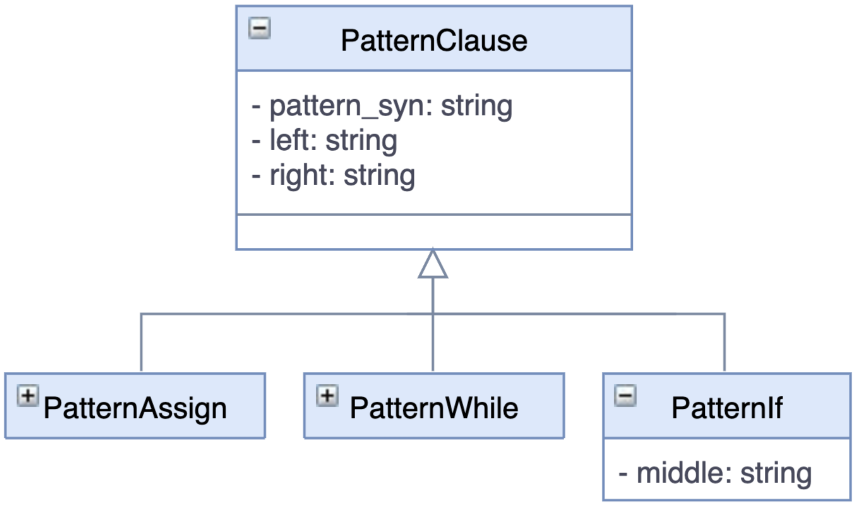


Fig. 3.5.3.3-3: PatternClause class diagram

Unlike Relational clauses, pattern clauses always have at least 1 synonym. For example:

"pattern a ("x", _)", *"pattern w ("x", _)"*, and *"pattern ifs ("x", _ , _)"*.

Therefore, we store the synonym before the parenthesis in Pattern clauses as `pattern_syn` attribute. The arguments in the parenthesis are stored as `left` and `right` attributes. (PatternIf has one more dummy attribute `middle` that stores the other wildcard).

Subclass 1: PatternAssign

The table belows shows two cases in which the number of synonyms in the clause is different. To obtain the results for the synonym / synonym pair, different PKB APIs will be called.

Case	Example	PKB API called	Results obtained
(1) The clause contains only 1 synonym, i.e. the `pattern_syn`	*pattern a (_ , _)* *pattern a ("x", _)* *pattern a ("x", "x+y")* *pattern a (_, _"x+y"_)*	`getMatchedStmt( string left, string right)`	`vector<int>` where the integers are stmt#
(2) The clause contains 2 synonyms, i.e. `pattern_syn` + `left` store synonyms	*pattern a (v , _)* *pattern a (v, _)* *pattern a (v, "x+y")* *pattern a (v, _"x+y"_)*	`getMatchedAssignPair (string right)`	`vector<pair <string, string>>` where the value pair represents <assign_stmt#, var_name>

Fig. 3.5.3.3-4: PKB APIs called in different scenario in PatternAssign class

When there is expression to be matched, for example, *pattern a (_, _"x+y"_) and pattern a ("x", "x+y")*, the whole expression will be passed as a string into PKB, PKB will build the AST tree for the expression using the same approach as Front-End Parser builds the AST tree, and perform the comparison based on the tree built. The details have been discussed in PKB Section 3.3.3.3 Part 3 - Pattern Matching. In this way, the logic of expression matching is hidden from Query Evaluator.

Subclass 2: PatternWhile

The evaluation of `PatternWhile` follows the same logic as `PatternAssign`. In fact, the evaluation process is simpler since there is no expression to be matched. We simply change the PKB API called. In Case 1 (only 1 synonym), we will call `getMatchedWhile(string left)` which returns a vector of integer with corresponding while stmt#. In Case 2 (2 synonyms), we will call `getMatchedWhilePair()` which returns <while_stmt#, var_name>.

Subclass 3: PatternIf

The evaluation of `PatternIf` follows the same logic as `PatternWhile`. In Case 1 (only 1 synonym), we will call `getMatchedIf(string left)` which returns a vector of integer with corresponding ifs stmt#. In Case 2 (2 synonyms), we will call `getMatchedIfPair()` which returns <ifs_stmt#, var_name>.

(3) Evaluation of With Clauses

`WithClause` evaluation begins with determining the Attribute enum type of the left and right parameters using Regex. The two extracted Attribute enums will form an Attribute pairing. The following shows the different possible Attribute types:

```
Value, StmtNo, CallProcName, ProcProcName, VarVarName, ReadVarName,
PrintVarName, Integer, Synonym, Ident
```

Fig. 3.5.3.3-5: Valid attribute in With Clause

Notice that certain attributes are mapped to multiple Attribute types based on the synonym it is attached to. For instance, the procName attribute has been split into Attribute type ProcProcName and CallProcName which represents the cases "procedure.procName", and "call.procName" respectively.

The different possible Attribute pairings have been categorized based on whether they share the same evaluation logic. These categories of evaluation logic have been abstracted into internal methods to avoid code redundancy. For each evaluation, the `WithClause` object checks the Attribute pairing category and makes a call to the appropriate internal method for evaluation.

The following table shows a subset of the `WithClause` evaluation cases.

Category	Attribute Pairing	PKB API Call

Value type NAME:
Statement Synonym and non-statement Synonym
	VarVarName, PrintVarName	`getAllMatchedEntity(DESIGN_ENTITY type)`
		`getVariablePrintedBy(STMT_NO s)`

	ProcProcName, CallProcName	`getAllMatchedEntity(DESIGN_ENTITY type)`
		`getProcedureCalledBy(STMT_NO s)`

	VarVarName, ReadVarName	`getAllMatchedEntity(DESIGN_ENTITY type)`
		`getVariableReadBy(STMT_NO s)`

Left parameter Attribute matches right parameter attribute	ProcProcName, ProcProcName	`getAllMatchedEntitiy(DESIGN_ENTITY type)`

Fig. 3.5.3.3-6: Subset of WithClause evaluation categories

Consider the first category named "Value type NAME: Statement Synonym and non-statement Synonym". This category consists of Attribute pairings that satisfy the following conditions:

- Both Attributes have a type value of NAME

- One Attribute is attached to a synonym that maps to a statement Design Entity, such as read, print or call.
- The other Attribute is attached to a synonym that maps to a non-statement Design Entity, such as a Procedure or Variable.

All attribute pairings within this category can be evaluated as such: First, call the PKB API method `getAllMatchEntity` to retrieve a list of statement numbers corresponding to the statement Design Entity.

For each statement number `n` in the list, we call the second listed API method using `n` as the method argument, which returns some string `s`. For instance, if the attribute pairing is <VarVarName, PrintVarName>, method `getVariablePrintedBy(n)` will be called and return the name of the variable used by `n`. Each pair of <n,s> is considered to satisfy the clause.

(4) Merging results in Query Result table

Remember that Query Result is used to maintain a table of all intermediary results and provides 4 APIs, namely **`"addOneSyn()"`**, **`"addTwoSyn()"`**, **`"filterOneSyn()"`**, **`"filterTwoSyn()"`**, for `Clause` classes to modify the results in the table. This section explains how these APIs are called under different circumstances and how the table content changes accordingly with the following query example:

"assign a; variable v; while w; statement s;
Select <w,v> such that Follows(3,a) and Modifies (a,"x") and Parent*(w,s) and Next*(w,a) and Uses(w,v)"*

The SPA Controller first creates a new `Query Result` object with an empty table, and passes it together with the `Query` object to Query Evaluator. The QE looks into the list of clauses in the `Query` object and calls `evaluate()` function provided by each `Clause` object and passes in the pointer of the `Query` object and `Query Result` object.

Assuming the clauses are evaluated from left to right, the first clause to be evaluated is *"Follows*(3,a)"*. Since there is only 1 synonym in this clause, and now the `Query Result` table is empty, we need to evaluate the clause independently and obtain a list of results for "a", and then add the list of results into the table by calling **`addOneSyn("a", results).`** The sequence diagram shows how the API is called.

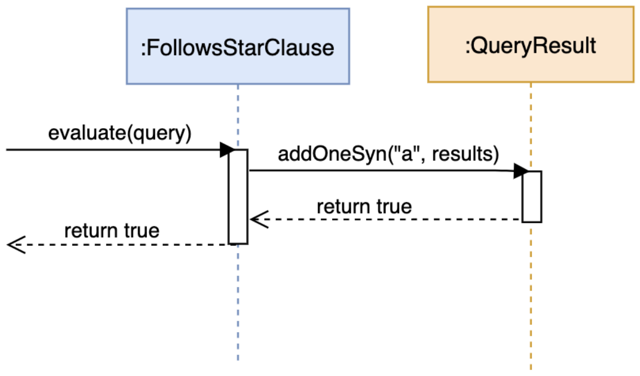


Fig. 3.5.3.3-7: Sequence Diagram of adding a column of values

After the above API calls, the `Query Result` looks like the following if valid results of a are <4, 8, 9. 10>:

cols (unordered_map)

Synonym (string)	Column No. (int)
a	1

rows (list<vector<string>>)

1
4
8
9
10

Column no.

Fig. 3.5.3.3-8: Query Result content after evaluating Follows (3,a)*

The second clause to be evaluated is "*Modifies (a,"x")".* This clause contains only 1 synonym and the synonym is already in the table. Hence, instead of adding new results, the existing results for "a" are re-evaluated. Those values that satisfy "*Modifies (a,"x")"* will be kept, otherwise will be deleted from the table. To achieve the above aim, the Modifies clause calls **`filterOneSyn("a", function)`**. The function parameter here is an anonymous function that utilises PKB API to check whether an existing value of "a" satisfies the current Modifies clause. After the above operation, the `Query Result` looks like the following if among the four values only a = 4 and 9 satisfy this clause (i.e. a = 8 and 10 are removed from the table):

cols (unordered_map)

Synonym (string)	Column No. (int)
a	1

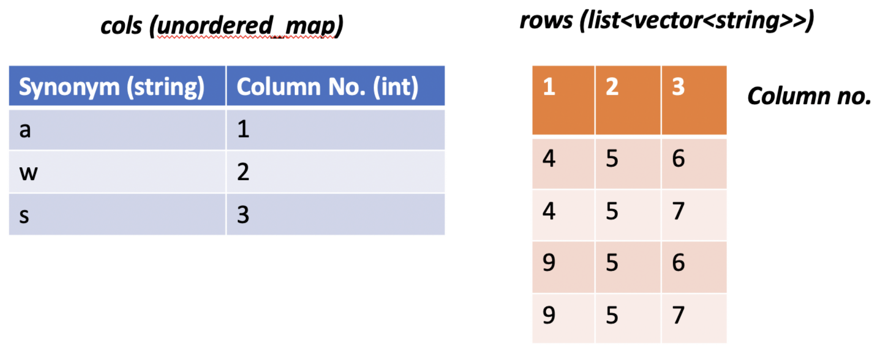
rows (list<vector<string>>)

1
4
9

Column no.

Fig. 3.5.3.3-9: Query Result content after evaluating Modifies(a,"x")

The third clause is *Parent*(w,s).* There are 2 synonyms in this clause and none of them are in the table at this point. Hence, we need to add all pairs of values for (w,s) into the table by calling **`addTwoSyn(“w”, “s”, resultPairs)`**. Two new columns are added and we perform a cross product computation between the existing rows and new value pairs. For example, if all <w,s> value pairs that satisfy this clause are <5,6> and <5,7>, we need to obtain all combinations of the existing rows and the new new pairs. The table then looks like below:



cols (unordered_map)

Synonym (string)	Column No. (int)
a	1
w	2
s	3

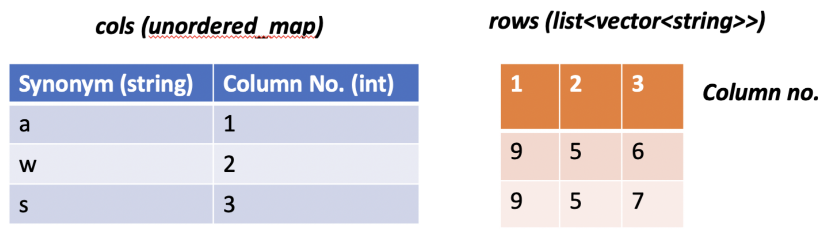
rows (list<vector<string>>)

1	2	3
4	5	6
4	5	7
9	5	6
9	5	7

Column no.

Fig. 3.5.3.3-10: Query Result content after evaluating Parent(w,s)*

The fourth clause is *Next*(w,a)*. We see that there are 2 synonyms in this clause and both of them are already in the table. Hence, we need to check whether the existing value pairs for (w,a) satisfy the Parent* relationship. This process is done by calling **`filterTwoSyn(“w”, “a”, function).`** Again, the function parameter is an anonymous function that utilises PKB API to check whether an existing value pair of (w, a) satisfies the current Parent* clause. For example, if w = 5, a = 4 does not satisfy this clause, we remove all the rows containing these value pairs for w and a. The table then looks like below:



cols (unordered_map)

Synonym (string)	Column No. (int)
a	1
w	2
s	3

rows (list<vector<string>>)

1	2	3
9	5	6
9	5	7

Column no.

Fig. 3.5.3.3-11: Query Result content after evaluating Next (w,a)*

Lastly, *Uses(w,v)* is evaluated. There are 2 synonyms in this clause, but only one of the synonyms, “w”, is in the table. In this case, we still call **`addTwoSyn(“w”, “s”, resultPairs)`**. In this API, it will check whether one of the synonyms is already in the table. If so, filtering is performed first on the existing values for this synonym. After that, the new synonym is added to the table according to the remaining values of the existing synonym. If there are 2 while loops in the code at line 5 and 11, and each of them Uses some variables. The value pairs we obtained by evaluating the clause independently are <5, x>, <5, y>, <11, m>. Since only w = 5 is in the table, we will only add v = x and v = y into the table and perform cross product on each row with w = 5. The table then looks like this:

cols (unordered_map)

Synonym (string)	Column No. (int)
a	1
w	2
s	3
v	4

rows (list<vector<string>>)

1	2	3	4
9	5	6	y
9	5	6	z
9	5	7	y
9	5	7	z

Column no.

Fig. 3.5.3.3-12: Query Result content after evaluating Uses(w,v)

Additional notes

All the APIs return a boolean value to the caller. It returns `FALSE` when the table becomes empty at the end of any one of the operations mentioned above, i.e. there are no values that can satisfy all the clauses so far. It returns `TRUE` as long as the table is not empty. This is used for early termination of the evaluation process, which will be further explained in the next section - Evaluation of Select Clause.

3.5.4. Query Result Projector (QRP)

QRP tries to retrieve the final results from the Query Result table according to the type of entities selected in Select Clause. After that, it formats the results as a list and assigns the list to the list pointer given by SPA Controller. The details of retrieving results according to Select Clause is discussed below.

After the evaluation of all relational/pattern/with clauses and the results are merged in the Query Result object, we are able to evaluate the Select clause based on the table content. For each case below, we will talk about the operations when there is no early termination as well as when there is early termination in the middle of the evaluation.

3.5.4.1. Case 1: Select BOOLEAN

If there is no early termination during the clauses evaluation, the Select clause returns `TRUE` if the `Query Result` table is not empty at the end of evaluation, otherwise, returns `FALSE`.

Early termination of evaluation can occur when any one of the clauses returns `false` from `evaluate()` function. This can happen when the results from PKB are empty, or `Query Result` table becomes empty after evaluating this clause. This means that no valid values that can satisfy all the clauses evaluated so far. Hence, the Select clause can return `FALSE` immediately without evaluating the clauses behind.

3.5.4.2. Case 2: Select Single Synonym

Assuming the selected synonym appears in at least one of the clauses, it must be inside the `Query Result` table at the end of evaluation. In this case, we can retrieve the results by getting all the values in the column that corresponds to this synonym.

For example, using the same query in the previous section:

"assign a; variable v; while w; statement s;
Select ***v*** *such that Follows*(3,a) and Modifies (a,"x") and Parent*(w,s) and Next*(w,a) and Uses(w,v)"*
And the final Query Result looks like this:

cols (unordered_map)

Synonym (string)	Column No. (int)
a	1
w	2
s	3
v	4

rows (list<vector<string>>)

1	2	3	4
9	5	6	y
9	5	6	z
9	5	7	y
9	5	7	z

Column no.

Fig. 3.5.4.1: Table content sample after evaluation of all clauses

The last column which represents “v” will be retrieved and duplicate values will be removed. The final result is: *y, z*.

Similar to select boolean, early termination of evaluation can occur when any of the clauses obtained an empty list from PKB or `Query Result` table becomes empty after evaluating this clause. In this case, the Select clause returns an empty list as the final result.

3.5.4.3. Case 3: Select Tuple

Similar to Select a single synonym, assuming all synonyms in the tuple appear in at least one of the clauses, all the synonyms must be inside the `Query Result` table at the end of evaluation. Hence, we can retrieve the values row by row and select the corresponding columns of values.

For example, if we use the same query above but modify the select clause:
“assign a; variable v; while w; statement s;
*Select **<a, v>** such that Follows*(3,a) and Modifies (a,”x”) and Parent*(w,s) and Next*(w,a) and Uses(w,v)”*

We go through every row and only retrieve the values at the 1st and 4th columns. Duplicate values will be removed, and thus the final result is: *9 y, 9 z*.

Again, early termination can occur under the same circumstances above, and an empty list will be returned.

3.5.4.4. Case 4: At least 1 synonym that does not appear in any of the clauses

The previous two cases assume that all the synonyms in the Select clause appear in at least one of the clauses, i.e., they must be in the `Query Result` table at the end of evaluation. However, when there is at least 1 synonym in the Select clause that does not appear in any of the clauses, we need to perform one more step - get all results matched for that synonym and do a cross product on the current table. This is equivalent to **`addOneSyn`** into the table. After that, we can retrieve the values just like Case 2 or 3.

If the `Query Result` table is empty during or at the end of evaluation, or result obtained from PKB for that synonym is empty, we shall return an empty list immediately.

3.5.5. Design Considerations

3.5.5.1. Design Considerations for Query Preprocessor: Query Syntax Validation

Criteria
Ease of implementation given existing specification; Ease of debugging; Ease of extension.

Chosen Decision: Regex matching
Our chosen decision involves using Regex matching to check the validity and type of the query statement.

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.

Alternative 1: 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 correctness of query evaluation is of our utmost concern, a lot of debugging is involved in developing SPA. Regexes, once written, will save a huge amount of time in debugging later. Given the tight schedule for 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.

3.5.5.2. Design Considerations for Query Evaluator

(1) Storing Clauses

Criteria
The storing method should have high extendability to best support the optimization of clauses in iteration 3.

Chosen Decision: Storing all clauses in one single list
Pros: Offers greater flexibility for optimization based on types of clause, types of parameters. Only a single Query Result table is needed to store intermediary results.

Cons: More complex optimization algorithm might be needed to sort the single list of clauses.

Alternative 1: Storing clauses into three lists based on type (relational / pattern / with clause)
Pros: able to separate the optimization algorithm into different types of clause and only sort the clause within the same type of , and thus make the algorithm simpler to implement.

Cons: Multiple tables might be needed to store results from different clauses. The sorting is restricted within the same type of clause, so the performance can be slowed down in some cases.

Justification for the chosen decision
We wish to offer greater flexibility for optimization algorithm and prioritize the time performance over the simplicity of implementation.

(2) Level of Abstraction of Clause Classes

Criteria
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 definitely want some level of abstraction, but there are varying levels of abstraction that we can achieve. We thus evaluate two solutions based on the principles of Object Oriented Programming to determine whether it would be more beneficial to have clauses divided into their main categories, or further divided into more specific categories.

Chosen Decision: Further separation into more specific clauses
We first separated the clauses into three main categories - Relational, Pattern and With clauses. We then further separated Relational clause into Integer Relational clause, Names Relational clause and Multi Statement Relational clause. We also separated Pattern clause into Pattern Assign clause, Pattern If clause and Pattern While clause.

Pros: From Iteration 1, we learned that certain clauses have more similarities with each other than other clauses, resulting in significant amounts of repeated code. By further abstracting these similar clauses, we are able to remove repeated redundant code to achieve a better level of Separation of Concerns (SoC) thus higher cohesion within each type of clause.

Cons: Some logic may be done differently in, for example, Integer Relational clause. For such cases, the respective logics have to be written in Follows and Parent clauses.

Alternative 1: Separation into main clauses (Iteration 1 implementation)
This alternative involves the separation of the clauses into three main categories - Relational, Pattern and With clauses. This was based on the understanding that relational clauses, encompassing Follows, Follows*, Parent, Parent*, Uses, Modifies, Call, Call*, Next and Next* are similar in nature. 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 some SoC between relational, pattern and with clauses.

Cons: There is a wasted opportunity for more abstraction to be carried out, resulting in a lot of repeated and redundant code, and this is bad Software Engineering practice.

Justification for choosing Further separation into more specific clauses
If we were to choose to code only the main clauses, several issues may arise in the future. For example, 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 each clause has their own logic. However, if we were to abstract clauses with similar logic to a common clause, fewer changes need to be made overall as changes only need to be applied to the common clause. This also increases the extendability of the code in the future. Therefore, separation into more specific clauses is the far superior option.

(3) Data structure used when merging results from multiple clauses

Criteria
The data structure should provide good time complexity when we merge results from different clauses. The space complexity for storing the temporary results should not be too large. It should also be extendable when the number of synonyms and clauses increases.

Chosen Decision: Maintain the intermediary results inside Query Result table
We created a Query Result class that maintains a table of intermediary results. At the end of evaluation of a clause, it adds results of new synonym(s) or filters the existing results in the table by calling APIs provided by the Query Result class.

Pros: Better extendability, time complexity and space complexity. Hide the details of modifying table contents from the clause classes by providing relevant APIs for different types of operation.

Cons: Implementation can be complex

Alternative 1: Maintain a list of values for each synonym. For clauses with pairs of synonyms, maintain a list of value pairs (Iteration 1 implementation)

In iteration 1, since there are at most 2 clauses, there are at most 2 common synonyms and 2 synonym pairs from each clause. Therefore, the size of both lists will not be too large. We only need to update the values in both lists at most once when we evaluate the second clause, if there is any. Such data structure is enough for

Pros: Easier to implement at iteration 1 stage. Satisfy the requirements when there are at most 2 clauses.

Cons: Poor extendability, time complexity, and space complexity when the number of synonyms and clauses increases.

Justification for the chosen decision
As the number of synonyms and clauses can be large in iteration 2 and 3, we need a more organized data structure with good extendability, time complexity and space complexity to support frequent merging of results.

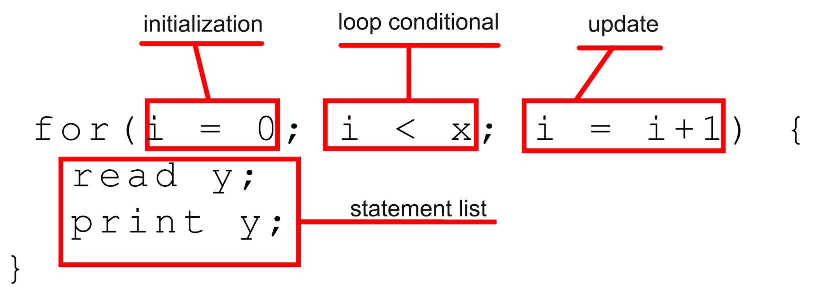
4. Extensions

4.1. Extension 1: For Loop and Do-While Loop

4.1.1. Definition

For this extension, we intend to extend the SIMPLE Concrete Syntax Grammar (CSG) to allow for parsing and analysis of more standard programming concepts. The scope of the extension is as follows:

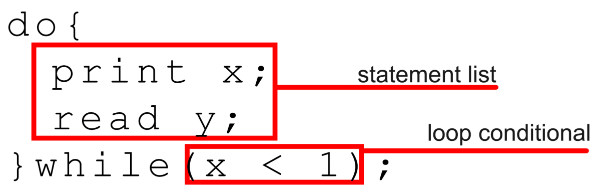
1. The SIMPLE CSG will be extended to include the 'for loop', defined as follows



```
for(i = 0; i < x; i = i+1) {
  read y;
  print y;
}
```

Fig. 4.1.1-1: For Loop Definition

2. The SIMPLE CSG will be extended to include the 'do-while loop', defined as follows:



```
do{
  print x;
  read y;
}while(x < 1);
```

Fig. 4.1.1-2: Do-While Loop Definition

3. The Front-End Parser is able to parse these new additions to the SIMPLE language and represent their structure in the AST data structure

4. The Design Extractor is able to identify, extract and store information about the new SIMPLE language extensions

5. The Query Processor is able to detect queries regarding the new extensions, retrieve the relevant information from the PKB during evaluation, and display the results.

4.1.2. Extension Specifications

4.1.2.1. SIMPLE CSG Extension

New non-terminals have been added to the SIMPLE CSG to represent the for-loop and do-while loop. The 'for' non-terminal is mapped to the for-loop, while the 'do' non-terminal is mapped to the do-while loop. The grammar rules have been updated as such:

Before stmt: read | print | call | while | if | assign

After
stmt: read | print | call | while | if | assign | for | do
 	
for : 'for' '(assign ';' cond_expr ' ; ' assign)' '{' stmtLst '}'
 	
do: 'do' '{' stmtLst '}' 'while' '(' cond_expr ')' ';'

Fig. 4.1.2.1-1: Extension to SIMPLE CSG

The relationship between source code and grammar rule is as follows:

For-Loop:
- The initialization statement has been mapped to an assign non-terminal
- The update statement has been mapped to an assign non terminal
- The loop conditional has been mapped to a cond_expr non-terminal
- The statement list has been mapped to a stmtLst non-terminal
- The remaining characters in the source code are represented as keywords.

Do-While Loop:
- The loop conditional has been mapped to the cond_expr non-terminal
- The statement list has been mapped to a stmtLst non-terminal
- The remaining characters in the source code are represented as keywords.

<u>Example</u>

The following is a valid declaration of the 'for' non-terminal in a SIMPLE Program:

```
for (i = 4; x < 20; x = x + 1) {
     read x;
     print y;
}
```

Fig. 4.1.2.1-2: Valid SIMPLE For Loop

The following is a valid declaration of the 'do' non-terminal in a SIMPLE Program:

```
do ((x + 4 < y) && (j == k) ) {
     read x;
     print y;
}
```

Fig. 4.1.2.1-3: Valid SIMPLE Do-While Loop

4.1.2.2. AST Extension

The standard AST structure must also be updated such that it can visually represent the structure for a do-while and for loop.

For Loop

The 'for' subtree will have 4 direct child subtrees to represent:

- the loop conditional expression
- the loop initialization
- the update statement
- the statement list to be executed upon each loop iteration.

To give an example, the AST sub-tree for the following SIMPLE for loop will be as follows:

```
for (i = 1; x < 2; z = 3+j) {
      read x;
}
```

Fig. 4.1.2.2-1: Sample Source

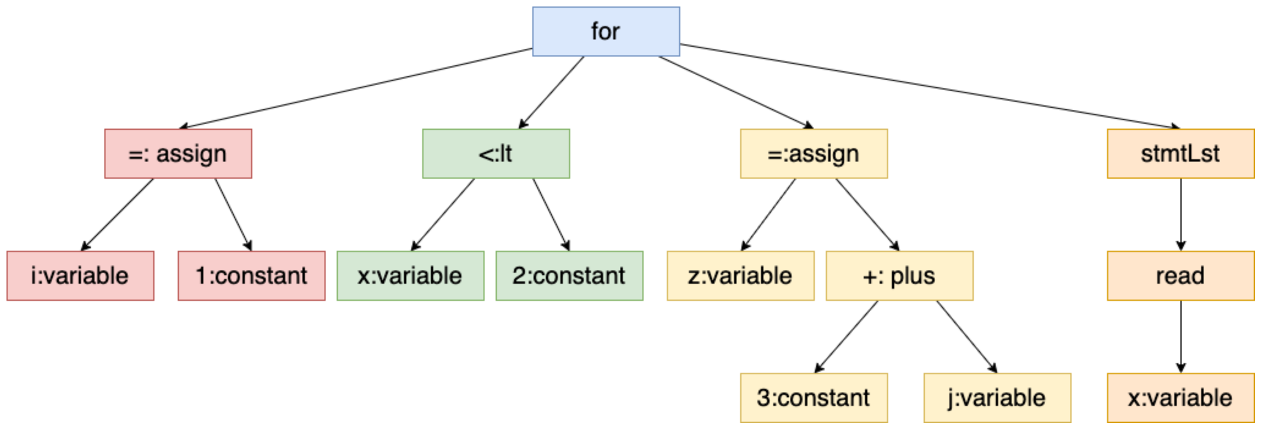


Fig. 4.1.2-5: Resulting AST subtree

Do-While Loop

The 'do' subtree will have 2 direct child subtrees to represent:

- The loop conditional expression
- The statement list to be executed during a loop iteration.

To provide an example, the AST sub-tree for the following SIMPLE for loop will be as follows:

```
do {
read x;
} while(i < 20);
```

Fig. 4.1.2.2-2: Sample Source

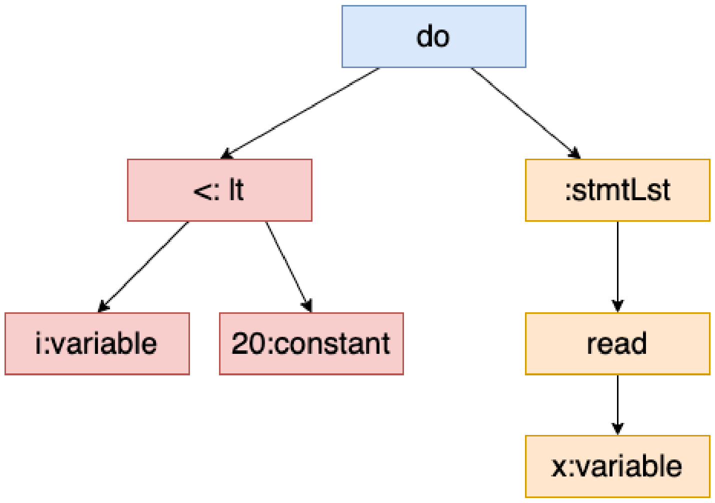


Fig. 4.1.2.2-3: Resulting AST subtree

4.1.2.3. PQL Extension

The PQL must be updated to allow the declaration of synonyms representing Do-While loop and for loop, as well as allow pattern matching of do-while and for loops.

Before: design-entity : 'stmt' | 'read' | 'print' | 'call' | 'while' | 'if' | 'assign' | 'variable' | 'constant' | 'procedure' pattern : assign | while | if

After:
design-entity : 'stmt' | 'read' | 'print' | 'call' | 'while' | 'if' | 'assign' | 'variable' | 'constant' | 'procedure' | ***'do'*** | ***'for'***
 	
pattern : assign | while | if | **for** | **do**
 	
for : syn-do '(' entRef ',' '_' ',' '_' ')'
//syn-while must be of type 'for'
 	
for : syn-do '(' entRef ',' '_' ')'
//syn-while must be of type 'dor'

Fig. 4.1.2.3-1: Extension of PQL

<u>**Example**</u>

To provide an example, the following are valid PQL queries related to do-while loops and for loops:

```
do d; for f;
Select d such that Follows(d,f) pattern d("x",_) and pattern f("x",_,_)
```

Fig. 4.1.2.3-2: Valid queries with PQL extension

4.1.2.4. Relationship Extensions

<u>**Do-Loop Relationships**</u>

The relationships rules for Do-While Loop are identical to the 'While' relationships.

While Relationships

The for loop Modifies relationships are defined as follows:

- Modifies(s,v) holds if there exists a statement s1 in the For Loop statement list such that Modifies(s1,v) holds.
- Modifies(s,v) holds if given the initialization assign statement a1 or update assign statement a2, Modifies(a1,v) or Modifies(a2,v) holds.

The For loop Uses relationships are defined as follows:

- Uses(s,v) holds if there exists a statement s1 in the For Loop statement list such that Ues(s1,v) holds.
- Uses(s,v) holds if given the initialization assign statement a1 or update assign statement a2, Uses(a1,v) or Uses(a2,v) holds.

4.1.2.5. Attribute Extensions

We have extended the attributes, with reference to Figure x.x to include the following:

- for.stmt#: Returns INTEGER
- do.stmt#: Returns INTEGER

Example

To provide an example, the following are valid PQL queries related to do-while loops and for loops:

```
do d; for f;
Select d with d.stmt# = f.stmt#
```

Fig. 4.1.2.5-1: Valid queries with PQL extension

4.1.3. Implementation for For Loop and Do-While Loop

4.1.3.1. Parser

(1) Update AST Data Structure and API

The data structures related to the Parser must be updated. Firstly the DesignEntity enum must be extended as follows:

Before: If, While, Assign, Read, Print, Stmt, Procedure, Program, Constant, Variable, Op, Call, Undefined, Wildcard
After: If, While, Assign, Read, Print, Stmt, Procedure, Program, Constant, Variable, Op, Call, Undefined, Wildcard, ***For, Do***

*Fig. 4.1.3.1-10: Valid queries with PQL extension. New Addition in **bold-italic***

With reference to Section 3.3.3.2 ,the StmtNode class must be extended to ensure the AST data structure can properly represent the new AST extensions:

- **ForNode:** Represents the For Loop statement. Stores the control variable declaration as an AssignNode, the loop conditional expression as CondExpressionNode, the variable update statement as another Assign Node, and the loop statement list as a vector<StmtNode>. Mapped to the 'for' DesignEntity enum.
- **DoNode:** Represents the Do Loop statement. Stores the loop conditional expression as CondExpressionNode and the loop statement list as a vector<StmtNode>. Mapped to the 'do' DesignEntity enum.

The AST API must also be updated to accommodate traversal of the nodes of design entity type 'For' and 'Do'.

TREE_NODE* getExpression() ***Description:*** If the node object has a design entity type of 'If', 'While', 'Assign', **'For' or 'Do',** return the expression node associated with the node object. Else, throw an error.
TREE_NODE* getLeftNode() ***Description:*** If the node has a design entity type of 'Op','Constant', 'Variable', or **'For',** return the root node of the left subtree associated with the node object. Else, throw an error.
TREE_NODE* getRightNode() ***Description:*** If the node has a design entity type of 'Op' or **'For'**, return the root node of the right subtree associated with the node object. Else, throw an error.

Fig. 4.1.3.1-2: Updated AST API. Modifications to the Description have been bolded.

Calling `GetExpression()` on a node of type 'For' or 'While' will return the child `CondExpressionNode`, while calling `GetLeftNode()` and `GetRightNode()` on a node of type 'For' will return the AssignNode representing the initialization statement and the update statement respectively. This will allow the Design Extractor to access all child nodes of the `ForNode` and `DoNode` for design abstraction extraction.

(2) Update Tokenizer Subcomponent

The Token Types must be updated to include the detection of keywords 'do' and 'for'.

Token Type	Sample Token Values
Identifier	Any var_name and proc_name that satisfy NAME rules
Keyword	read, print, while, if, assign. call, ***do, for***
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. 4.1.3.1-3: Examples of Token values. Additions are in ***bold-italic***.

(3) Update Parser Subcomponent

As the syntax grammar for the stmt non-terminal has been extended, we must update the logic of the internal method that parses it to for and do non-terminals. Internal methods to parse the **for** and **do** non-terminals must be implemented as well.

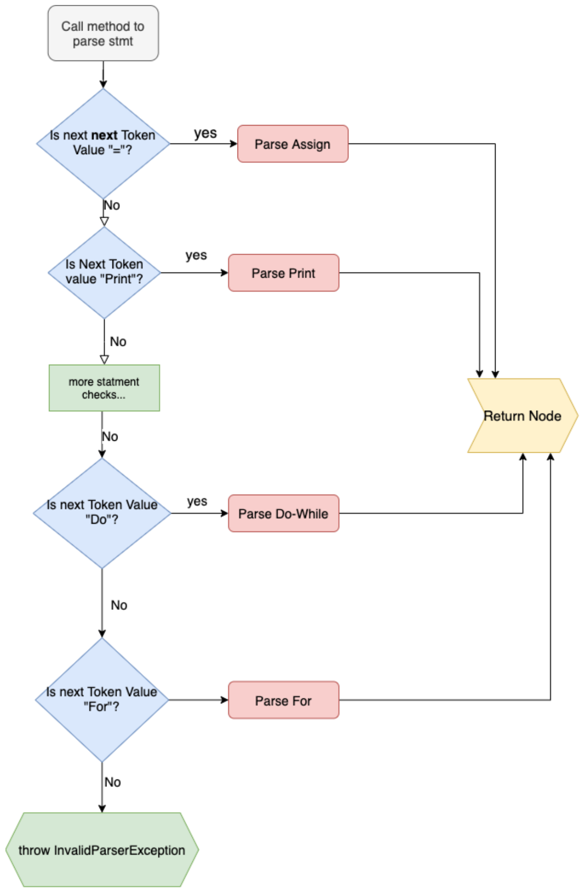

Call method to parse stmt
Is next **next** Token Value "="?
yes
Parse Assign
No
Is Next Token value "Print"?
yes
Parse Print
No
more statment checks...
No
Return Node
Is next Token Value "Do"?
yes
Parse Do-While
No
Is next Token Value "For"?
Parse For
No
throw InvalidParserException

Fig. 4.1.3.1-4: Updated Statement Parsing Flow Chart

4.1.3.2. PKB

(1) Update Storage Tables

Two new internal tables must be added to store the Do-While statement numbers and For Loop statement numbers.

(2) Update Design Extractor

The Next relationship extraction must be changed for Do-While loops to reflect the change in CFG and the For-Loop will adopt the same CFG structure as a while loop. The remainder of the relationships will be dynamically resolved since they rely on the CFG and AST generated.

4.1.3.3. Query Processor

(1) Update Query Preprocessor

The REGEX in the Query Preprocessor must be updated to accommodate for the PQL extension. The Regex rule to check query structure correctness and synonym declaration, for instance, will be updated as such:

```
const string REGEX_QUERY =
R"((\s*(stmt|read|print|call|while|if|assign|variable|constant|procedur
e|prog_line|for|do)\s+[A-Za-z][A-Za-z0-9]*\s*(\s*,\s*[A-Za-z][A-Za-z0-9]
*)*\s*;\s*)+\s*Select\s+(BOOLEAN|([A-Za-z][A-Za-z0-9]*|[A-Za-z][A-Za-z0
-9]*.(procName|varName|value|stmt#))|<\s*([A-Za-z][A-Za-z0-9]*|[A-Za-z]
[A-Za-z0-9]*.procName

const string REGEX_DECL =
(\s*(stmt|read|print|call|while|if|assign|variable|constant|procedure|p
rog_line|for|do)\s+[A-Za-z][A-Za-z0-9]*\s*(\s*,\s*[A-Za-z][A-Za-z0-9]*)*
\s*;\s*)+
```
Fig. 4.1.3.3-1: Regex updates. Additions in bold

(2) Update Clause Classes

The Clause class diagram must be updated to allow for the instantiation of Clause objects that handle the evaluation and validation of Pattern matching queries for Do-While loops and For loops. The update involves the implementation of two Classes, PatternDo and PatternIf, that inherit from PatternClause.

4.1.4. Implementation Challenges

The scope of this implementation is complex in that it requires functionality extension to all components in the SPA Program. As a result, implementation of this extension will likely span the entire duration of iteration 3 to ensure enough time is allocated for implementation, integration and thorough testing. To ensure that we are still able to implement the rest of iteration 3 requirements, certain team members will be assigned to work on tasks with trivial implementation and short deadlines. Once they have implemented their assigned tasks, they will be able to work on iteration 3 requirements. This plan is further detailed in Section 4.1.5.

4.1.5. Implementation Schedule

Main Task/Week	**9**	**10**	**11**	**12**
Update Tokenizer (Rachel)	🟦			
Update AST API (Jaime)	🟦			
Update Parser (Jaime)	🟦			
Update Query Preprocessor (Anqi)	🟦			
Update Query Evaluator (Chen Su)	🟦	🟦		
Update PKB Storage (Jefferson)	🟦			
Update Design Extractor (Adam)		🟦	🟦	

Fig. 4.1.5: Implementation Schedule

The workload for week 9 has been divided up between team members, and each subtask can be implemented independently from other components. As the Design Extractor is coupled with the AST API, its implementation will begin after Week 9.Task that are trivial to complete are given a one week deadline, while tasks that require more complex updates to logic are given a 2 week deadline.

4.1.6. Test Plan for For Loop and Do-While Loop

Main Task/Week	**9**	**10**	**11**	**12**
Unit Testing of Tokenizer (Rachel)	🟦			
Unit Testing of Parser (Jaime)	🟦			
Unit Testing of Query PreProcessor (Anqi)		🟦		
Unit Testing of Query Evaluator (Chen Su)	🟦			
Unit Testing of PKB Storage (Jefferson)	🟦			
Unit Testing of Design Extractor (Adam)		🟦		
Integration Testing of Parser/PKB (Jaime)		🟦	🟦	
Integration Testing of PKB/QP (Chensu)		🟦	🟦	
System Testing (Rachel)			🟦	🟦

Fig. 4.1.6: Test Plan

Unit testing is expected to be implemented as the component is being updated. Integration testing and system testing may be written in conjunction with the unit tests and integration tests respectively, though a second week has been allocated for the running of tests and subsequent bug-fixing.

4.2. Extension 2: NextBip/NextBip*

4.2.1. Definition

NextBip/NextBip* are extension of normal Next/Next* relationship. It is defined between program lines not only in the same procedure but also different procedures. It stands for “branch into procedures”. It provides a more accurate picture of the flow of execution of a program.

For example, if we have a SIMPLE source program with multiple procedures as the following:

```
procedure main {
1.      x = 5;
2.      call Mary;
3.      y = x + 6;
4.      z = y * 2;
5.      call John;
}

procedure Mary {
6.      y = x * 3;
7.      call John;
}

procedure John {
8.      if (i > 0) then {
9.              x = x + z;
     } else {
10.             y = x * y;
     }
}
```

Fig. 4.2.1: Sample code with multiple procedures

If we evaluate Next*(1, a), the results will be all the assignments that can be executed after line 1 in the CFG of the *“main”* procedure, which is *“3, 4”*. However, if we evaluate NextBip*(1,a), the results will be all assignments that can be executed after line 1 in the CFG of the entire program, including the lines in *“Mary”* and *“John”*. Thus, the result will be *“3,4,6,9,10”*.

4.2.2. Extension Specification for For Loop and Do-While Loop

4.2.2.1. SIMPLE Source Program

There is no new syntax allowed for SIMPLE source program for this extension.

4.2.2.2. PKB / Design Extractor

Besides the existing CFG extracted and stored for each individual procedure, additional CFG for inter-procedural NextBip relationships. NextBip* relationship is calculated dynamically.

4.2.2.3. PQL

PQL grammar extension (only relevant rules listed; changes are highlighted in bold):
relRef : ModifiesP | ModifiesS | UsesP | UsesS | Calls | CallsT | Parent | ParentT | Follows | FollowsT | Next | NextT | Affects | AffectsT | **NextBip | NextBipT**
NextBip : 'NextBip' '(' lineRef ',' lineRef ')'
NextBipT : 'NextBip*' '(' lineRef ',' lineRef ')'

4.2.3. Implementation for For Loop and Do-While Loop

4.2.3.1. PKB / Design Extractor

Procedure calls can call procedures which are not stored yet when traversing the AST to extract the relationships, which might result in a desynchronization when extracting from the AST. We hence have to formulate a new method of extracting the inter procedure Next relationship from the already constructed CFG. A potential solution is to store the start and stop statements of each procedure, the following NextBip relationship would then only need to be updated at each calls statement.

4.2.3.2. PQL

(1) QPP

<u>**Regexes for Query Syntax Validation**</u>
New regexes need to be written for NextBip and NextBip*. Besides, current regex for relational clause needs to be modified to accommodate NextBip and NextBip*. Details are shown below:

```
const string REGEX_REL_CLAUSE = R"(such
that\s+(Follows|Follows\*|Parent|Parent\*|Modifies|Uses|Calls|Calls\*|N
ext|Next\*|NextBip|NextBip\*)\s*\(\s*([A-Za-z][A-Za-z0-9]*|_|\"\s*[A-Za-
z][A-Za-z0-9]*\s*\"|[0-9]+)*\s*,\s*([A-Za-z][A-Za-z0-9]*|_|\"\s*[A-Za-z
][A-Za-z0-9]*\s*\"|[0-9]+)*\s*\)(\s+and\s+(Follows|Follows\*|Parent|Par
ent\*|Modifies|Uses|Calls|Calls\*|Next|Next\*|NextBip|NextBip\*)\s*\(\s*
([A-Za-z][A-Za-z0-9]*|_|\"\s*[A-Za-z][A-Za-z0-9]*\s*\"|[0-9]+)*\s*,\s*(
[A-Za-z][A-Za-z0-9]*|_|\"\s*[A-Za-z][A-Za-z0-9]*\s*\"|[0-9]+)*\s*\))*)"
;

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

```
const string REGEX_NEXT_BIP_STAR =
R"(\s*NextBip\*\s*\(\s*([A-Za-z][A-Za-z0-9]*|_|[0-9]+)\s*,\s*([A-Za-z][A-
Za-z0-9]*|_|[0-9]+)\s*\))";
```
*Fig. 4.2.3.2-1: Updated PQL Concrete Grammar with NextBip/NextBip**

New `Clause` classes & `validate` methods

Two new clauses need to be implemented: `NextBipClause` and `NextBipStarClause`. Corresponding `validate` functions inside these classes need to be implemented for semantic validation of argument types. As both clauses take in identical argument types `(lineRef, lineRef)`, we will use `NextBipClause::validate` as an example:

```
void NextBipClause::validate(unordered_map<string, DesignEntity> map) {
    vector<string> params = {left, right};
    for (string p : params) {
        bool isLineSyn = isSynonym(p) && isLine(map[p]);
        if (!(isLineSyn || isInteger(p) || isWildcard(p))) {
            throw InvalidQueryException("Invalid param type for NextBip
clause.");
        }
    }
    //valid left & right argument types: line syn, int, wildcard
}
```
*Fig. 4.2.3.2-2: Semantic Validation of NextBip/NextBip**

In the code snippet above, `validate` enforces that both arguments of `NextBipClause` need to be one of the following types: (i) a synonym that represents a line type (stmt, prog_line, etc.); (ii) an integer that represents a statement number; (iii) a wildcard ("_").

(2) QE

New Classes of NextBipClause and NextBipStarClause are created. The evaluation process shall be similar to NextClause and NextStarClause. It calls relevant PKB APIs to obtain the results.

4.2.4. Implementation schedule

Main Task/Week	9	10	11	12
Update Design Extractor (Adam)	🟦			
Update PKB Storage (Jefferson)	🟦			
Implement PKB API called by QE (Adam)		🟦		
Update QPP (Anqi)	🟦			

Implement NextBip/NextBips* Clause Classes (Chen Su)		▨		

*Fig. 4.2.4: Implementation Schedule for NextBip/NextBip**

4.2.5. Test Plan

Main Task/Week	9	10	11	12
Unit Testing of Query PreProcessor (Anqi)	▨			
Unit Testing of PKB Storage (Jefferson)	▨			
Unit Testing of Design Extractor (Adam)	▨			
Integration Testing of PKB/QP (Chen Su)		▨	▨	
System Testing (Rachel)			▨	▨

*Fig. 4.2.5: Testing Schedule for NextBip/NextBip**

4.3. Extension 3: AffectsBip/AffectsBip*

4.3.1. Definition

Similar to NextBip/NextBip*, AffectsBip/AffectsBip* extend the normal Affects/Affects* relationship. It is defined between assignments not only in the same procedure but also different procedures. It provides a more accurate picture of how the modification of variables affects other variables somewhere else in the program.

We will use the same code fragment as Fig. 4.2.1 for illustration:

If we evaluate Affects*(1, a), the results will be all the assignments that can be executed after line 1 in the CFG of the *"main"* procedure, which is *"3, 4".*

However, if we evaluate AffectsBip*(1,a), the results will be all assignments that can be executed after line 1 in the CFG of the entire program, including the lines in *"Mary"* and *"John"*. Thus, the result will be *"3,4,6,9,10"*.

4.3.2. Extension Specification for AffectsBip/AffectsBip*

4.3.2.1. SIMPLE Source Program

There is no new syntax allowed for SIMPLE source program for this extension.

4.3.2.2. PKB / Design Extractor

Explanation: Storage of AffectsBip

The computation of AffectsBip will be identical to Affects with the caveat that when a calls statement is encountered, instead of skipping the statement, we check if there are affects relationships in the function called. No AffectsBip will be stored in tables, and all computation will be done dynamically.

The evaluation of AffectsBip* will be evaluated identically to AffectsBip* except that each Affects check in the algorithm is replaced with AffectsBip.

4.3.2.3. PQL

PQL grammar extension (only relevant rules listed; changes are highlighted in bold):
relRef : ModifiesP | ModifiesS | UsesP | UsesS | Calls | CallsT | Parent | ParentT | Follows | FollowsT | Next | NextT | Affects | AffectsT | **AffectsBip | AffectsBipT**
AffectsBip : 'AffectsBip' '(' stmtRef ',' stmtRef ')'
AffectsBipT : 'AffectsBip*' '(' stmtRef ',' stmtRef ')'

4.3.3. Implementation for AffectsBip/AffectsBip*

4.3.3.1. PKB / Design Extractor

Implementation is to build upon Affects/Affects* with similar modification to NextBip/NextBip*.

4.3.3.2. PQL

(1) QPP

Regexes for Query Syntax Validation
New regexes need to be written for AffectsBip and AffectsBip*. Besides, current regex for relational clause needs to be modified to accommodate AffectsBip and AffectsBip*. The changes are similar to what is proposed in the section about NextBip & NextBip* above.

New `Clause` classes & `validate` methods
Two new clauses need to be implemented: `AffectsBipClause` and `AffectsBipStarClause`. Corresponding `validate` functions inside these classes need to be implemented for semantic validation of argument types. As both clauses take in identical argument types `(stmtRef, stmtRef)`, we will use `AffectsBipClause::validate` as an example:

```
void AffectsBipClause::validate(unordered_map<string, DesignEntity>
map) {
    vector<string> params = {left, right};
    for (string p : params) {
        bool isStmtSyn = isSynonym(p) && isStmt(map[p]);
        if (!(isStmtSyn || isInteger(p) || isWildcard(p))) {
            throw InvalidQueryException("Invalid param type for
AffectsBip clause.");
        }
    }
    //valid left & right argument types: stmt syn, int, wildcard
}
```

*Fig. 4.2.3.2-2: Semantic Validation of AffectsBip/AffectsBip**

In the code snippet above, `validate` enforces that both arguments of `AffectsBipClause` need to be one of the following types: (i) a synonym that represents a statement type (stmt, assign, etc.); (ii) an integer that represents a statement number; (iii) a wildcard ("_").

(2) QE

New Classes of AffectsBipClause and AffectstBipStarClause are created. The evaluation process shall be similar to AffectsClause and AffectsStarClause. It calls relevant PKB APIs to obtain the results.

4.3.4. Implementation Schedule

The implementation of AffectsBip/AffectsBip* should start after Affects/Affects* are implemented and well tested in Iteration 3.

Main Task/Week	9	10	11	12
Update Design Extractor (Adam)		🟦		
Update PKB Storage (Jefferson)		🟦		
Implement PKB API called by QE (Adam)			🟦	
Update QPP (Anqi)		🟦		
Implement AffectsBip/AffectsBips* Clause Classes (Chen Su)			🟦	

*Fig.4.3.4: Implementation Schedule for AffectsBip/AffectsBip**

4.3.5. Test Plan for AffectsBip/AffectsBip*

Main Task/Week	9	10	11	12
Unit Testing of Query PreProcessor (Anqi)		🟦		
Unit Testing of PKB Storage (Jefferson)		🟦		
Unit Testing of Design Extractor (Adam)		🟦		
Integration Testing of PKB/QP (Chen Su)			🟦	🟦
System Testing (Rachel)				🟦

*Fig.4.3.5: Testing Schedule for AffectsBip/AffectsBip**

5. Documentation and Coding Standards

5.1. 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

5.2. Coding Standards

Having a coding standard helps to give a uniform appearance to the codes written among all the 6 different team members. It improves readability, maintainability of the codes and reduces complexity. This will help in code reuse and to detect error easily moving into Iterations 2 and 3 implementation.

The coding standard we have come up with was primarily decided within ourselves at the beginning of Iteration 1 after some online searches. We loosely follow the C++ naming convention by an organisation known as Chaste[1] to the extent that we felt was appropriate. Our primary focus was to ensure that the naming convention was easily adaptable by all team members and would not cause any confusion, particularly between variable and method names. We agreed that we should not follow parts of the convention such as the method argument names and class attribute names as it would make the code unnecessarily confusing to keep track of, so it was in such cases where we deviated from Chaste’s naming convention.

The coding standards are carefully discussed and planned by our team members 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”

[1] https://chaste.cs.ox.ac.uk/trac/raw-attachment/wiki/CodingStandardsStrategy/codingStandards.pdf

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. 5.2: An overview of coding standards

5.3. 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.

6. Testing

6.1. Test Plan

Week	Testing Activities	Components
3	Unit Testing	SPA Controller
4-5	Unit Testing	Front End subcomponent: Tokenizer Token Class Front End subcomponent: Parser Parser: AST API PKB: Design Extractor + Extracting relationships of Follows, Follows*, Parent, Parents*, pattern assign clauses With Clause PKB: PKB Storage PQL: Query Preprocessor + Test validate function Query Evaluator Query Result Projector Query Result
5-6	Integration Testing	Front End Parser and PKB PKB and PQL
	System Testing	Front End, PKB and PQL - No clause but Select synonym - Such that clauses (Follows, Follows*, Parent, Parents*, Modifies, Uses) - Pattern assign clause (not full specification)
	Regression Testing	
	Bug Fixes	
Recess Week	System Testing	Front End, PKB and PQL - Such that and Pattern assign Clauses - Invalid source program - Complicated source program
	Bug Fixes	
	Regression Testing	
7	Unit Testing	QPP Subcomponent: Query Parser - Fix of white space bugs from iteration 1 feedback
		Parser : Call Statement Parsing PKB Storage: Storing New relationships
	System testing	QPP Subcomponent: Query Parser - Fix of white space bugs from iteration 1 feedback

8	Unit Testing	PKB Design Extractor: Extracting new relationships With Clause Next Clause Next* Clause Calls Clause Calls* Clause Pattern while, if Clause
	Integration Testing	PQL and PKB Front-End and PKB
	System Testing	Front End, PKB and PQL - Select attributes, Tuple and Boolean - with clause - pattern while, if type clauses - pattern assign exact and partial expression matching - Multiple clauses
	Bug Fixes	
9	System Testing	Front End, PKB and PQL - Multiple clauses
	Bug Fixes	

Fig. 6.1: An overview of Test plan

6.1.1. Test Plan in Details

(1) Unit Testing

Unit testing is done after each development of the individual components to ensure correct logic implementation in the individual component.

In week 3 to 5, the development of SPA Controller, Tokenizer, Parser, AST API, Design Extractor, PKB Storage, Query Preprocessor, Query Evaluator and Query Result Projector are completed. Unit Testing on those components were done alongside with the completion. For example, with a manually created AST, PKB is able to use the AST, stores all the design entities such as procedures, statements, variables, constants and all the design abstractions that are allowed to be preprocessed including *Follows, Follows*, Parent, Parent*, Uses, Modifies*. In week 7, additional features are added to the development of the system with new design abstractions that are allowed to be preprocessed such as *Calls, Calls*, Next, Next*, pattern assign full specification, pattern while and if type*. More Unit testing for Parser, AST API, Design Extractor, PKB Storage, QP, QPP, QE and QRP was done in week 7 and 8 to accommodate the new design entities and design abstractions.

To run the unit testing, go to the files in `unit_testing > src` directory and right click and select `Run`. When bugs are detected, the respective team member will have to make improvements to their implementation logic.

(2) Integration Testing

Integration testing is done when most of the components have gone through unit testing and are relatively stable. It is carried out to ensure different components can interact with each other properly. Parser-PKB and PKB-PQL testing are done.

Once the unit testing of the newly added design abstractions were completed in week 8, integration testing of Parser-PKB and PKB-PQL was done. For example, we need to ensure Parser is able to correctly build the AST, parse the AST to PKB, PKB receives the AST from Parser, stores the design entities and design abstraction including *Calls, Calls*, Next, Next*, pattern assign full specification, pattern while and if type*. More details on integration testing can be found under Section 6.3.1 and 6.3.2.

To run the integration testing, go to the files in `integration_testing > src` directory and right click and select `Run`. When bugs are detected, the respective team member will have to make improvements to their implementation logic.

(3) System Testing

Once unit testing and integration testing are done and relatively stable, system testing is carried out to ensure our SPA system is properly integrated and does not have any logic flaws or bugs that will break our system. To ensure the system works and meet the specification requirement for iteration 2, in week 8 and 9, system testing was carried out. In week 8, we ensure our system can handle multiple tuple or BOOLEAN in the select clause and new relationships: *Calls, Calls*, Next, Next*, pattern assign full specification, pattern while and if type. In week 9,* we also ensure our system can handle multiple *such that*, *with* and *pattern* clauses, with any number of *and* operators.

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. To run all the system test cases at once for iteration 2, go to the files in `tests > Iteration_2` directory and `Open` and right click `Run` the script file `run_all_iter2_tests.sh`.

(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_iter2_tests.sh` located in `tests > Iteration_2 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.

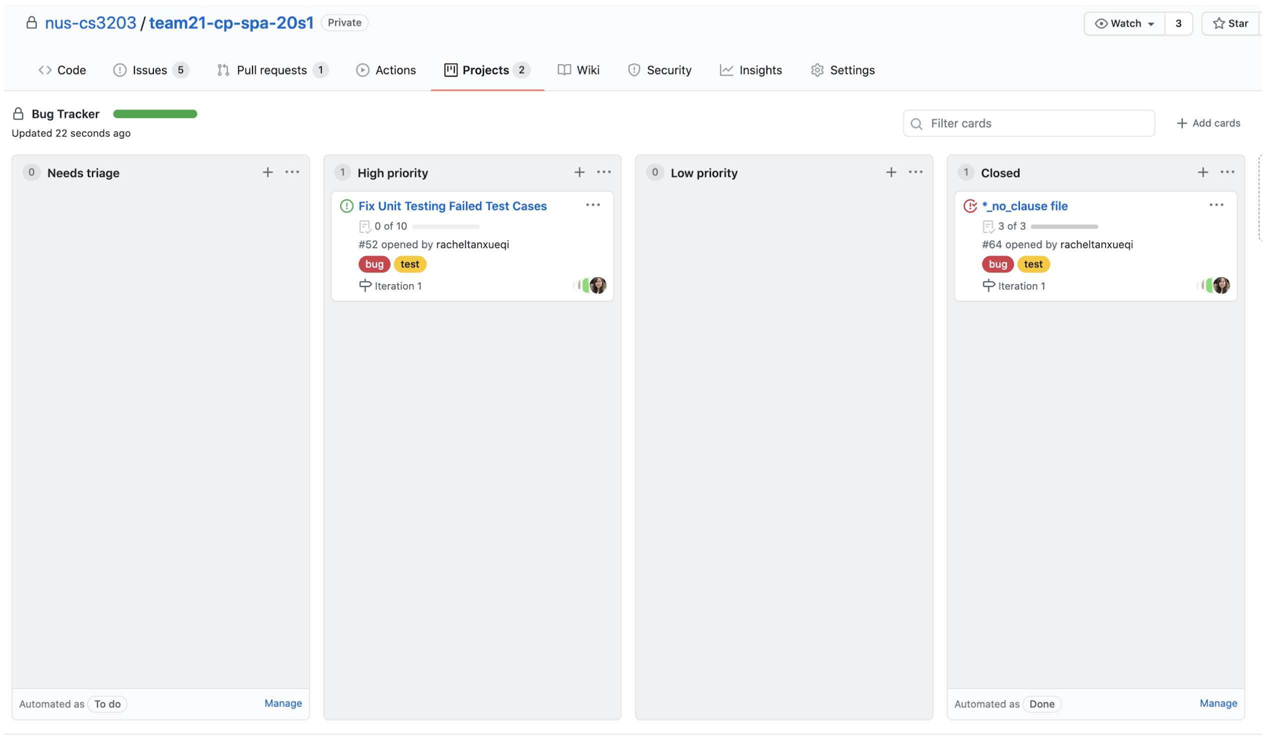


Fig. 6.1.1: Bug Tracking Tool

6.2. Unit Testing

6.2.1. 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.

6.2.1.1. 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”.

For iteration 2, we updated the Tokenizer test cases to include the testing of Call statement tokenization.

Case	Input

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

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

Fig. 6.2.1-1: Test cases for Token Object Creations

Two Sample Unit Test Cases

Purpose	Input	Assertion	Expected Test Results

`Valid Variable 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. 6.2.1-2: Sample Unit test cases for Token object creation

6.2.1.2. 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. 6.2.1.2: Sample Unit test cases for Tokenizer

6.2.1.3. Test Category 3: Parser

Overview

To test the Parser, we manually create sets of valid and invalid TokenStream objects to pass into the Parser. For iteration 2, the Parser test cases have been updated to include tests for proper parsing of SIMPLE programs with multiple procedures and call statements.

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 {`
	
	 `call x;`
	
	 `}`

Invalid Variable	procedure main {
	
	call 1;
	
	}

Missing semicolon	procedure main {
	
	call x
	
	}

Fig. 6.2.1-3: Example test cases for parsing Assign Statements

<u>Two Sample Unit Test Cases</u>

Purpose	Input	Assertion	Expected Test Results

Valid Test Case: Call Statement	procedure main {
	
	call 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{
	
	call 1;
	
	}
		TOKENSTREAM token_stream = <manually constructed token stream input>
		
		REQUIRE_THROWS_WITH(parser->buildAST(token_stream), “Invalid Call 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. 6.2.1-4: Sample Unit test cases for Parser

6.2.2. Unit Testing: PKB

6.2.2.1. 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. 6.2.2-1: CRUD test cases for PKB

6.2.2.2. 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) == true)`
			isParent(1, 2) should return true

`Test Extraction of Parent*`	`procedure main{`	`PKB pkb = PKB() <PKB is manually constructed by inserting values>`	isParentStar(1, 3) should return true

Relationships	`while (1 == 1) {`
	`while (2 == 2) {`
	`    x = 2;`
	`}}}`
		`updateParent()`
		`updateParentStar()`
		
		`REQUIRE(isParentStar(1, 3) == true)`	

Fig. 6.2.2: Relationship test cases

6.2.3. Unit Testing: Query Processor

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

6.2.3.1. Query Preprocessor

(1) 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 stmt number)	`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.

Fig. 6.2.3.1-1: Sample Unit Tests for Query Subparts Extraction

(2) Test Category 2: Construction of the Query object

Overview

To test the function `QPP::constructQuery`, a query string is passed in, and the three components of the expected output `Query` object (syn_entity_map, `selected` vector, `clauses` vector) 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", "_", "_");`
		
		`vector<string> selected = {"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 == q.selected);`
			We pass in a sample query, and manually construct the expected components of the Query object (syn_entity_map, selected_syn, clauses). We then compare the result of calling `QPP::constructQuery` with the expected output, and require every component to be equal.

`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. 6.2.3.1-2: Sample Unit Tests for Query Construction

6.2.3.2. Query Evaluator

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

6.2.3.3. 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”};`	`formatResults(raw, res)`	list<string> res should contain all the results in raw results

`list<string> res;`	`res == {“1”, “2”, “3”}`	

Fig. 6.2.3.3: Sample Unit Test for Query Result Formatter

6.2.3.4. 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 exception if all parameters are valid	`FollowsClause clause = FollowsClause("s", "r");`
	
	 `unordered_map<string,DesignEntity> map = {{"s",DesignEntity::Stmt}, {"r",DesignEntity::Read}};`
		`REQUIRE_NOTHROW(clause.validate(map));`	We construct a `FollowsClause` with 2 synonym parameters “s” and “r”, and a synonym-entity map for 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. 6.2.3.4: Sample Unit Tests for `Clause::validate`

6.3. Integration Testing

6.3.1. 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 Multiple 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 Calls Relationship`	`procedure p {`
	`...`
	`calls q;`
	`...`
	`}`
		`REQUIRE(pkb->isCalls(“p”, “q”) == true);`
		
		`REQUIRE(pkb->isCalls(“p”, “Example”) == false);`
			In this test case, we check if the Design Extractor has correctly extracted the Calls Relationships from the AST.
			
			We make a PKB API call that returns a BOOLEAN to test if the PKB is able to accurately detect valid and invalid Calls relationships.

Fig. 6.3.1: Sample Integration test cases for Front-End Parser to PKB interaction

6.3.2. 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

Purpose	Test Setup	Assertion	Expected Test Results

`Testing if CallsClause is able to retrieve correct results for Calls relationship through API calls and add the results into the Query Result Table`	`query.selected_synonym = "p"; query.clauses = clauses; query_res = new QueryResult();`	`CallsClause("3", p).evaluate(&query, &query_res);`
		
		 `vector<string> answer = { "Mary" };`
		
		 `REQUIRE(query_res.getCol("p") == answer);`
			We first initialise the Query object, simulating the Query object that is passed to the QE.
			
			CallsClause then evaluates the clause by calling PKB APIs and adds the results into query_res.
			
			Lastly, we check the results in the table by retrieving corresponding columns of values.

`Testing if PatternWhile is able to retrieve correct results for While pattern matching through API calls and add the results into the Query Result Table`	`query.selected_synonym = "w"; query.clauses = clauses; query_res = new QueryResult();`	`PatternWhile("w", "x", "_").evaluate(&query, &query_res);`
		
		 `vector<string> answer = {"3", "4" };`
		
		 `REQUIRE(query_res.getCol("w") == answer);`
			Similar to the above

Fig. 6.3.2: Sample Integration Tests Between PKB and Query Processor

6.4. System Testing

6.4.1. Queries Design Consideration

For each of the no clause, such that, with, pattern and multiple 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 Calls query, the grammar rules states that

```
Calls: (entRef , entRef)
entRef: syn / '_' / '"'IDENT'"'
```

The Calls could have different argument combinations generated for testing of the Follows queries.

LHS	RHS
Synonym	Same Synonym
Synonym	Different Synonym
Synonym	Wildcard
Synonym	IDENT
Wildcard	Wildcard
Wildcard	Synonym
Wildcard	IDENT
IDENT	IDENT
IDENT	Synonym
IDENT	Wildcard

Fig. 6.4.1: All possible combination permutations of arguments for Calls

In each argument, there can be `syn` of read, print, while, if, assign, variable, constant, procedure, stmt, prog_line used and permuted. To be more specific, any combinations that use non-procedure as the `syn` can be classified as invalid cases as anything other than procedure `syn` do not satisfy `entRef`. 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 caused by tab, spaces, spelling, syntax mistakes (Section 6.4.10)
- Shared arguments between different relationship in multiple clauses and select return result
- Different Tuple arguments combinations, synonym and attributes variations for return result (Section 6.4.4)

6.4.2. Source Program 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, with 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)
- Procedures with many call design entity but do not have cyclic links between procedures call

6.4.3. Categorizing Source Program and Query Test files

The source program and queries are designed with a testing intention and more details can be found in their respective section linked in the Fig.6.4.3 below. In addition to categorizing the source program and Query files based on testing intention, the files are also splitted up to help the team pinpoint bugs and issues in a more easier manner. For example, as seen in File No. 12 in Fig.6.4.3, to carry out system test on *Calls, Calls*, Next, Next*, ModifiesP* and *UsesP* clauses, one source was designed to cater to these clauses. The queries were splitted up to 3 different files and they share the same source. Instead of combining them and having to look through over 1000 test cases, focusing on each query file and returning its respective xml file with 1 type of clause allow for a clearer workflow.

No.	File	Testing Intention
Iteration 1		
1	no_clause_source.txt no_clause_queries.txt	Refer to Section 6.4.4, Section 6.4.10
2	Follows_Parent_source.txt Follows_Parent_queries.txt	Refer to Section 6.4.5, Section 6.4.10
3	pattern_Modifies_Uses_source.txt pattern_Modifies_Uses_queries.txt	Refer to Section 6.4.6, Section 6.4.8
4	pattern_suchthat_source.txt queries_pattern_suchthat.txt	Refer to Section 6.4.10

5	complex_condexpr_nested_source.txt complex_condexpr_nested_queries.txt	Refer to, Section 6.4.11, Section 6.4.12, Section 6.4.13
6	uncommon_invalid_source.txt uncommon_invalid_queries.txt	Refer to Section 6.4.11
7	invalid_test_source.txt	Refer to Section 6.4.11
8	bonus_3_features_souce.txt bonus_3_features_queries.txt	Refer to Section 6.4.6
	Iteration 2	
9	Select_source.txt Select_queries.txt	Refer to Section 6.4.4,
10	With_clause_source.txt With_clause_queries.txt	Refer to Section 6.4.9
11	pattern_full_spec_source.txt pattern_full_spec_queries.txt	Refer to Section 6.4.8, Section 6.4.11
12	Calls_Next_ModifiesP_UsesP_souce.txt Calls_queries.txt Next_queries.txt ModifiesP_UsesP_source.txt	Refer to Section 6.4.7, Section 6.4.6
13	multiple_clauses_source.txt multiple_clauses_queries.txt	Refer to Section 6.4.10,

Fig. 6.4.3: Overview of System Tests

6.4.4. Test Category 1: Simple Testing of Queries with Select and Result clause

In this test category, the purpose is to simply test if the queries are able to Select each of the synonyms and its attribute names of the design entities properly and return the correct results in BOOLEAN or Tuple. The test expects the whole system to return test results that pass both the valid and invalid cases.

Source Design Considerations

- Contain all possible design entities: ‘read’ | ‘print’ | ‘call’ | ‘while’ | ‘if’ | ‘assign’ | ‘procedure’ | var_name | const_value
- Include multiple procedures
- No cyclic calls of procedures

Queries Design Considerations

- Different result clause tuple element combinations where element can be a combination of synonym or synonym attribute names
- Synonyms need to be declared for return results of synonyms and should meet the Grammar definition of PQL.

- 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 6.4.11)

Result Clause Combinations

Result Clause Type	Valid Sample
synonyms	Procedure, read, print, call, while, if, assign, variable, constants, prog_line, stmt
Attribute names	procedure-syn.procName, stmt-syn.stmt#, variable-syn.varName, constants-syn.value, read-syn.varName, read-syn.stmt#, print-syn.varName, print-syn.stmt#, call-syn.procName, call-syn.stmt#, while-syn.stmt#, if-syn.stmt#, assign-syn.stmt#
Tuple	elem | '<' elem (',' elem)* '>' where elem: attributes or synonyms

Fig. 6.4.4-1: Result clause combinations

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.stmt#,r.varName>`
	`1 a,2 b,11 z`
	`5000`
		`1 a,2 b,11 z`

`procedure a {`
`   call b;`
`}`
`procedure b {`
`   call c;`
`}`
`procedure c {`
`   ...`
`}`
	`7 - Valid: Query return`
	`results of procedure`
	`attributes`
	`procedure p`
	`Select p.procName`
	`a,b,c`
	`5000`
		`a,b,c`

Fig. 6.4.4-2: Sample test cases for test category 1

6.4.5. 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.

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.

Queries Design Considerations

- Generated with all the possible arguments combinations to test the relationship of the clause
- 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 6.4.11)

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. 6.4.5: Sample test cases for test category 2

6.4.6. Test Category 3: Simple Testing of Queries with Modifies and Uses clauses

In this test category, the purpose is to test if the queries with Modifies and Uses meet the requirement for iteration 2 which is that all statements should include assignment, print, container, procedure as well as procedure call for Modifies and Uses. Hence, 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.

Source Design Considerations

- Include multiple procedures and *call* design entity
- Have at least 2 or more levels of nesting variations between if and while loops (Section 6.4.12)
- 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 6.4.12,).
- 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

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
- Each argument has different variations of inputs to test the relationship of the clause
- Add variation to the Select, different result clause tuple element combinations where element can be a combination of synonym or synonym attribute names
- 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 6.4.11)

Two Sample System Test Cases

Source Code Snippet	Required Test inputs	Expected Test Results
`procedure Medium {` `a = b + c; // 1` `call Medium2; // 2` `...` `}` `Procedure Medium2 {` `...` `}`	`1 - Valid ModifiesP, syn on both args` `procedure p;variable v;` `Select p that Modifies(p,v)` `Medium, Medium2` `5000`	`Medium, Medium2`
	`3 - Valid UsesP, syn on both args` `procedure p;variable v;` `Select <p.procName, v.varName> that Uses(p,v)` `Medium b, Medium c` `5000`	`Medium b, Medium c`

Fig. 6.4.6: Sample test cases for test category 3

6.4.7. Test Category 4: Simple Testing of Calls, Calls*, Next, Next* clauses

In this test category, the purpose is to test if the queries with Calls, Calls*, Next, Next* return the correct results. It is designed with a high level of complexity to both the source and queries as described below.

Source Design Considerations

- Include several procedures and *call* design entity
- Different variation of procedure call links but no cyclic call links
- 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 6.4.12) to cater to query design with Next and Next*
- Different kind of statements positioned with an intention to execute after one and other or not to cater to query design with Next and Next* (Section 6.4.12)

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
- Each argument has different variations of inputs to test the relationship of the clause
- Add variation to the Select, different result clause tuple element combinations where element can be a combination of synonym or synonym attribute names
- 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 6.4.11)

6.4.8. Test Category 5: Simple Testing of pattern clauses

In this test category, the purpose is to simply test if the queries with pattern clauses return the correct results. In iteration 2, the pattern clauses have s types *assign, while* and *if*. For pattern-assign, it now handles full expression partial and exact matching. It is designed with a medium level complexity for both the source and queries as described below.

Source Design Considerations

- Include multiple procedures and *call* design entity
- Have at least 2 or more levels of nesting variations between if and while loops (Section 6.4.12)
- 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 6.4.12).
- Add valid but similar variable name as design entities
- Assignment statements are included with different term, complicated expression, brackets variation to test pattern-assign partial and exact matching of the expression

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
- Each argument has different variations of inputs to test the relationship of the clause
- Add variation to the Select, different result clause tuple element combinations where element can be a combination of synonym or synonym attribute names
- Shared arguments between pattern and select clause
- 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 6.4.11)

Two Sample System Test Cases

Source Code Snippet	Required Test inputs	Expected Test Results

`procedure Match {`
`   a = b + c; // 1`
`   call Match2; // 2`
`...`
`}`
 	
`procedure Match2 {`
`   while (d == e) { // 3`
`     if ( f == d ) then {//4`
`     ...`
`     }`
`   }`
`...`
`}`
	`17 - Valid Pattern Assign: Select syn v, syn v on LHS, Exact Expression Pattern Match on RHS`
	`assign a; variable v;`
	`Select v pattern a(v,”b+c”)`
	`a`
	`5000`
		`a`

	`17 - Valid Pattern While: Select syn v, syn v on LHS, Exact Expression Pattern Match on RHS`
	`assign a; variable v; while w;`
	`Select <w,v> pattern w(v,_)`
	`3 d, 3 e`
	`5000`
		`3 d, 3 e`

Fig. 6.4.8: Sample test case for test category 5

6.4.9. Test Category 6: Simple Testing of With clauses

In this test category, the purpose is to test if the queries with With clause are able to return results properly.

Source Design Considerations

- Include multiple procedures and *call* design entity
- Include repeated variable names or constants in different statements
-

Queries Design Considerations

- Generated with all the possible arguments combinations of LHS and RHS of with clause: IDENT, INTEGER, attrRef, synonym
- Add variation to the Select, different result clause tuple element combinations where element can be a combination of synonym or synonym attribute names
- Shared arguments between pattern and with clause
- Valid queries categorised when they satisfy the clause
- Include queries with different combinations of with clause
-

Two Sample System Test Cases

Source Code Snippet	Required Test inputs	Expected Test Results

`procedure p1 {`
`   p1 = b + c; // 1`
`   call p1; // 2`
`   ...`
`}`
 	
`procedure p1 {`
`  print a1; // 3`
`}`
	`10 - Valid With, attrRef = attrRef`
	`procedure p; variable v;`
	`Select v.varName with p.procName = v.varName`
	`p1`
	`5000`
		`p1`

	`16 - Valid With, attrRef = INTEGER`
	`print pr;`
	`Select pr.stmt# with pr.stmt# = 3`
	`3`
		`3`

	5000	

Fig. 6.4.9: Sample test case for test category 6

6.4.10. Test Category 7: Testing of Queries with Multiple clauses

In this test category, the purpose is to test if the queries is able to involve multiple **such that, with,** and **pattern** clauses, with any number of **and** operators. In addition, the queries is now able to include all relationships including Follows, Follows*, Parent, Parent*, Modifies, Uses (for all statements including procedure calls, as well as procedures), Calls, Calls*,Next, Next*. This is a high level complexity to both the source and queries as we now want to accommodate all the possible considerations mentioned in Test Category 1 to 5.

Source Design Considerations

- Contains all of the design entity statements that satisfy all types of such that, with pattern clauses
- Take into consideration all the above mentioned design from Test Category 1 to 5

Queries Design Considerations

- Generate query with any number of such that, with and pattern clauses and there is a default **and** operators between any two consecutive clauses of the same types
    - All the possible combination between all the clauses and relationships
    - Generate with all the possible arguments combinations to test the relationship of the clause
- Each argument has different variations of arguments to test the relationship of the clause
    - Have shared synonyms as the arguments between multiple clauses and relationships
- Add variation to the Select, different result clause tuple element combinations where element can be a combination of synonym or synonym attribute names
- 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 6.4.12)
- Structure variation of multiple clause but return the same results to validate QPP logic further

For example, the query all produce the same result:

assign a; while w;

- Select a such that Modifies (a, “x”) and Parent* (w, a) and Next* (1, a)
- Select a such that Parent* (w, a) and Modifies (a, “x”) such that Next* (1, a)
- Select a such that Next* (1, a) and Parent* (w, a) and Modifies (a, “x”)
- Select a pattern a (“x”, _) such that Parent* (w, a) such that Next* (1, a)
- Select a such that Parent* (w, a) and Next* (1, a) pattern a (“x”, _)
- Select a such that Next* (1, a) and Parent* (w, a) pattern a (“x”, _)

Two Sample System Test Cases

Source Code Snippet	Required Test inputs	Expected Test Results

`procedure Match {`
`print var1; // 1`
`call Match2; // 2`
`...`
`}`
 	
`procedure Match2 {`
`while (d == e) { // 3`
`if ( f == d ) then {//4`
`a = b + var1; // 5`
`}`
`}`
`d = e - var1; // 6`
`}`
	`17 - Valid Multiple clauses:`
	`assign a; while w;`
	`Select <w,a> such that Modifies (a, "a") and Parent* (w, a) and Next* (3, a)`
	`3 6`
	`5000`
		`3 6`

	`18 - Valid Multiple clauses:`
	`assign a; while w; procedure p;`
	`Select <a,v> such that Uses(p,v) and Calls (p, "Match2") pattern a(_,v)`
	`5 var1,6 var1`
	`5000`
		`5 var1,6 var1`

Fig. 6.4.9: Sample test case for test category 6

6.4.11. Test Category 8: 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 6.4.4 to 6.4.10 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.

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)
- Cyclic procedure call

Invalid sample source snippet	Expected assert

`procedure InvalidSource {`
`call a;`
`}`
`procedure b {`
`call c;`
`}`
`procedure c {`
`call InvalidSource;`
`}`
	`Invalid calls Exception: Recursive procedure calls in SIMPLE source`

Fig. 6.4.10-1: Example of invalid source

Invalid Query Design Consideration

Invalid Query Design Explanation	Example
Syntactically incorrect and has invalid grammar rules	
Has **and** connecting or introducing clauses of the different types	assign a; while w; Select a such that Parent*(w,a,) and Modifies(a,"x") **and such that** Next*(1,a)
Double declaration of the same name	call sameName; procedure sameName;

Extra separator, comma, missing syntax in select-cl, pattern-cl, with-cl or suchthat-cl	stmt s;
	Select s;

Weird spaces or tabs that violate the grammar syntax	stmt s; Select s such that Calls(_,_)
Declaration of synonym names that do not meet grammar rules	call cp_1;
Invalid synonym's attribute names	stmt s; Select p.procName
Semantically Incorrect	
Have '_' as the first argument for Modifies and Uses	variable v; Select v such that Modifies(_,v) Select v such that Uses(_,v)
The argument of a relationship is not one of the allowable types	while w; variable v,v1; if ifs; Select w pattern w(v, v1) Select ifs pattern ifs(v, _ , v1)

Fig. 6.4.10-2: Example of invalid query

6.4.12. Test Category 9: 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.

Positioning of container statements

Positioning Types Source Code Snippet	Test Coverage

```
if(a==b) then {
  ...
} else {
  ...
```
	• For Next, Next*, there is a statement executable after the last statement in else container.

`}`
`...`	

`While (a==b) { // 2`
`... // 2`
`}`
`... // 3`
	• For Next, Next*, the next executable statement after the while statement could be statement 2 or 3

`if(a==b) then {`
`  while(c < d) {`
`   ...`
`  }`
`} else {`
`  ...`
`}`
`}`
	• Satisfies Parent, Parent*, Modifies, Uses, pattern
	• Does not satisfy Follows/Follows* statement
	• For Next, Next*, there is no statement executable after the last statement in if and else container.

`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
	• For Next, Next*, there is no statement executable after the last statement in if and else container.

`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
	• For Next, Next*, there is no statement executable after the last statement in if and else container.

`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
	• For Next, Next*, there is no statement executable after the last statement in if and else container.

Fig. 6.4.11-1: Variation of the Positioning of container statements

<u>Nesting Types for container statements</u>

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

<u>Complicated conditional expression in container statements</u>

Source Code Snippet	Required Test Input	Expected Test Results
`while (((a<s)&&((d>f)||((55/aa`	`1 - Valid Uses`	`a,s,d,f,aa,true,q,w,`

`)<=66)))    && (((!(true==(a+2))) && (0==0)) && (11>=(q*(22+44)/w%e)))) { ... }`	`variable v; while w; Select v such that Uses(w,v)` `a,s,d,f,aa,true,q,w,e 5000`	`e`
`if (a && b) then { ... } else { ... }`		`Invalid If Statement: expected relative expression comparator, got '&&'.`

Fig. 6.4.11-2: Sample test cases with complicated conditional expression

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.

6.4.13. Test Category 10: 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

Valid but Complicated Query Design

Valid but Complicated Query Design Explanation	Example
Declaring synonyms with the same name as the design entities to ensure no flaw in validation rule logic	stmt Select; select Select
Declaring long synonym names to handle correct string handling	variable longName123longName123longName123longName123; Select longName123longName123longName123longName123
Tricky spaces between clauses and arguments to ensure no flaw in PQL grammar rules	assign a; stmt s; Select s such that Calls*(_,_) pattern w(_, _)

Fig. 6.4.12: Examples of valid but complicated queries

7. Discussion

7.1. 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 Iterations 1 and 2 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 2.1 and Fig. 2.1-2 to meet and share about our progress, issues faced, or clarifications about the communication between components of the SPA system.

7.2. 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 the submission of Iteration 1. 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. In Iteration 2, we did not use the GitHub issue tracker at all. This was partly because all the issues from Iteration 1 have been resolved and Iteration 2 involved fewer tasks, so we did not see a need to use the issue tracker.

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

7.3. 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 Iterations 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.

In Iteration 2, we struggled a little with time management as midterms and deadlines from other modules started piling on. This resulted in some delay in tasks completion as certain tasks depended on other tasks. If we were to start the project again, we would have the foresight to better prepare for this issue and either delegate tasks differently based on our schedules or ensure that certain tasks are completed first.

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.

7.4. 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.

-

8. API Documentation

8.1. 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
BOOLEAN 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

8.2. 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

8.3. 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

8.4. 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

8.5. 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

8.6. Calls, Calls*

Calls, Calls*   Overview: API related to the Calls and Calls* clauses
BOOLEAN isCalls(**PROCEDURE_NAME** p1, **PROCEDURE_NAME** p2)   ***Description:*** Returns true if p1 Calls p2
LIST_OF_PROCEDURE_NAMES getCalls(**PROCEDURE_NAME** p)   ***Description:*** Returns the procedures that p Calls
LIST_OF_PROCEDURE_NAMES getCalledBy(**PROCEDURE_NAME** p)   ***Description:*** Returns all the procedures that Calls p
BOOLEAN isCallsStar(**PROCEDURE_NAME** p1, **PROCEDURE_NAME** p2)   ***Description:*** Returns true if p1 Calls* p2
LIST_OF_PROCEDURE_NAMES getCallsStar(**PROCEDURE_NAME** p)   ***Description:*** Returns the procedures that p Calls*
LIST_OF_PROCEDURE_NAMES getCalledByStar(**PROCEDURE_NAME** p)   ***Description:*** Returns all the procedures that Calls* p

8.7. Next, Next*

Next, Next*   Overview: API related to the Next and Next* clauses
BOOLEAN isNextStatement(**STMT_NO** s1, **STMT_NO** s2)   ***Description:*** Returns true if s2 is the Next statement of s1
LIST_OF_STMT_NO getNextStatements(**STMT_NO** s)   ***Description:*** Returns all the Next statements of s
LIST_OF_STMT_NO getPreviousStatements(**STMT_NO s**)   ***Description:*** Returns all the statements s is the Next of
BOOLEAN isNextStar(**STMT_NO** s1, **STMT_NO** s2)   ***Description:*** Returns true if s2 is the Next* statement of s1
LIST_OF_STMT_NO getNextStar(**STMT_NO s**)   ***Description:*** Returns all the Next* statements of s
LIST_OF_STMT_NO getPreviousStar(**STMT_NO s**)

Description: Returns all the statements s is the Next* of

8.8. With

With Overview: API related to the With clause
LIST_OF_STMT_NOS getAllStatementsThatRead(**VARIABLE_NAME** v) ***Description:*** Returns all read statement numbers that modifies variable v.
LIST_OF_STMT_NOS getAllStatementsThatPrint(**VARIABLE_NAME** v) ***Description:*** Returns all print statement numbers that use variable v.
LIST_OF_STMT_NOS getAllStatementsThatCall(**PROCEDURE_NAME** p) ***Description:*** Returns all call statement numbers that call procedure p.
PROCEDURE_NAME getProcedureCalledBy(**STMT_NO** s) ***Description:*** Returns the procedure called by s.
VARIABLE_NAME getVariableReadBy(**STMT_NO** s) ***Description:*** Returns the variable read by s.
VARIABLE_NAME getVariablePrintedBy(**STMT_NO** s) ***Description:*** Returns the variable printed by s.
BOOL isVariable(**STRING** s) ***Description:*** Returns true if s is a variable name.
BOOL isProcedure(**STRING** s) ***Description:*** Returns true if s is a procedure name.

8.9. Pattern

Pattern Overview: API related to the Pattern clause
LIST_OF_STMT_NOS getMatchedStmt(**STRING** lhs, **STRING** rhs) ***Description:*** Returns all assign statement numbers that match lhs on the left of the equal sign and rhs on the right of the equal sign
LIST_OF_PAIRS_OF_STMT_NOS_AND_VARIABLE_NAMES getMatchedAssignPair(**STRING** RHS) ***Description:*** Returns all assign statement-variable pairs that match the pattern
BOOLEAN isMatch **(INT** s**, STRING** lhs**, STRING** rhs**)**

Description: Returns true if assign statement number s matches lhs on the left of the equal sign and rhs on the right of the equal sign
LIST_OF_STMT_NOS getMatchedWhile(**STRING** lhs) ***Description:*** Returns all while statement numbers that uses lhs as control variable
LIST_OF_PAIRS_OF_STMT_NOS_AND_VARIABLE_NAMES getMatchedWhilePair(**STRING** LHS) ***Description:*** Returns all while statement-variable pairs that match the pattern
LIST_OF_STMT_NOS getMatchedIf(**STRING** lhs, **STRING** lhs) ***Description:*** Returns all if statement numbers that uses lhs as control variable
LIST_OF_PAIRS_OF_STMT_NOS_AND_VARIABLE_NAMES getMatchedIfPair(**STRING** LHS) ***Description:*** Returns all if statement-variable pairs that match the pattern
BOOLEAN isControlVar **(INT** s**, STRING** expr**)** ***Description:*** Returns true if statement number s if while/if and uses expr as a control variable