SCHOOL OF COMPUTING

CS3203
Software Engineering Project

Iteration 3 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 SPA Program Implementation

1.1. Parsing Source Language SIMPLE

Frontend is able to tokenize and parse the source program and create an AST, which will be sent to the PKB for design extraction. 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, Next*, Affects, Affects**. The aforementioned information is stored in the PKB Storage. The stored design abstractions are then searched for and given to the Query Evaluator for evaluation via PKB API calls. During extraction, the Design Extractor will also check for cyclic procedure calls and procedures of the same name.

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 iteration 3 including *Follows, Follows*, Parent, Parent*, Uses, Modifies, Calls, Calls*, Next, Next*, Affects, Affects**. QP is also able to handle with-constraints, attribute names, pattern while- and if- types.

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 of synonyms.

1.5. Extensions

The first extension is the For-loops and Do-While Loops. The second and third extension is the implementation of *NextBip, NextBip*, AffectsBip, AffectsBip**. The front end parser is extended to handle the new For-loops and Do-While Loops syntax in the source program. PKB is also extended to extract the design entities with the For-loops and Do-while Loops. The new design abstractions and relationships are extracted and stored in PKB Storage. Query Processor and evaluator is able to handle the queries with the new design abstraction.

1.6. Constraints

Our SPA Program will not store *Next*, NextBip*, Affects, Affects*, AffectsBip* and *AffectsBip** 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 (QP, System Tests)
Rachel Tan	Testing IC, Developer (System Tests)
Jaime Chow	Documentation IC, Developer (Front-end, System Tests)
Chen Su	Developer (QP)
Chen Anqi	Developer (QP)
Adam Tan	Developer (PKB, QP)

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 12 for every Monday, Tuesday and Saturday

In iteration 3, our efforts were split among the implementations of Affects, optimisations, extensions and generating quality system tests that can test every aspect of our system to its limit. This was different from what we did in iteration 2, where most of our efforts were directed to the Query Processor component, which required the most work at that time.

Some of our roles have changed since the start of the project to cater to the changing demands of the project from start to end. Jefferson was in charge of implementing PKB storage methods and the Query Evaluator portion for relational clauses in iterations 1 and 2. The workload was heavy at the start since there was significant logic to be implemented. Towards the end, the workload reduced as remaining clauses can be extended from existing ones, so he could help out with system testing. Rachel was part of the Front-end team in iteration 1, focusing on the Tokenizer component. Similarly, workload was heavier at the start but lessened later on. Being the Testing IC, she was responsible for

the majority of the system tests implemented from start to end of the project. Jaime was also part of the Front-end team, and as the workload lessened after the groundwork had been laid out, she helped out in writing system tests as well. Additionally, Rachel and Jaime worked together to implement the With clause. Chen Su and Anqi's roles did not change as the QP is an extremely heavy component. Chen Su's focus was in Query Evaluator and Anqi's focus was in Query Preprocessor. In iteration 3, they worked together to implement the optimisations for the system. Lastly, Adam was responsible for the Design Extraction part of the project. He contributed immensely to query evaluation as well, implementing the more challenging clauses, and is the brain behind all the algorithms in the PKB.

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

	Iteration 1				Iteration 2				Iteration 3		
Main Task/Week	**3**	**4**	**5**	**6**	**R**	**7**	**8**	**9**	**10**	**11**	**12**
Front-End Parser Implementation											
PKB Storage Implementation											
Design Extractor Implementation											
PQL Implementation											
Follows/Follows*/Parent/Parent*/Modifies/Uses											
Pattern Clause											
Calls/Calls*/Next/Next* Clauses Implementation											
Affects/Affects* Clauses Implementation											
With Clause Implementation											
Clause Refactorization and Abstraction											
Intermediate Table Implementation											
Rewriting Clause to implement intermediate tables											

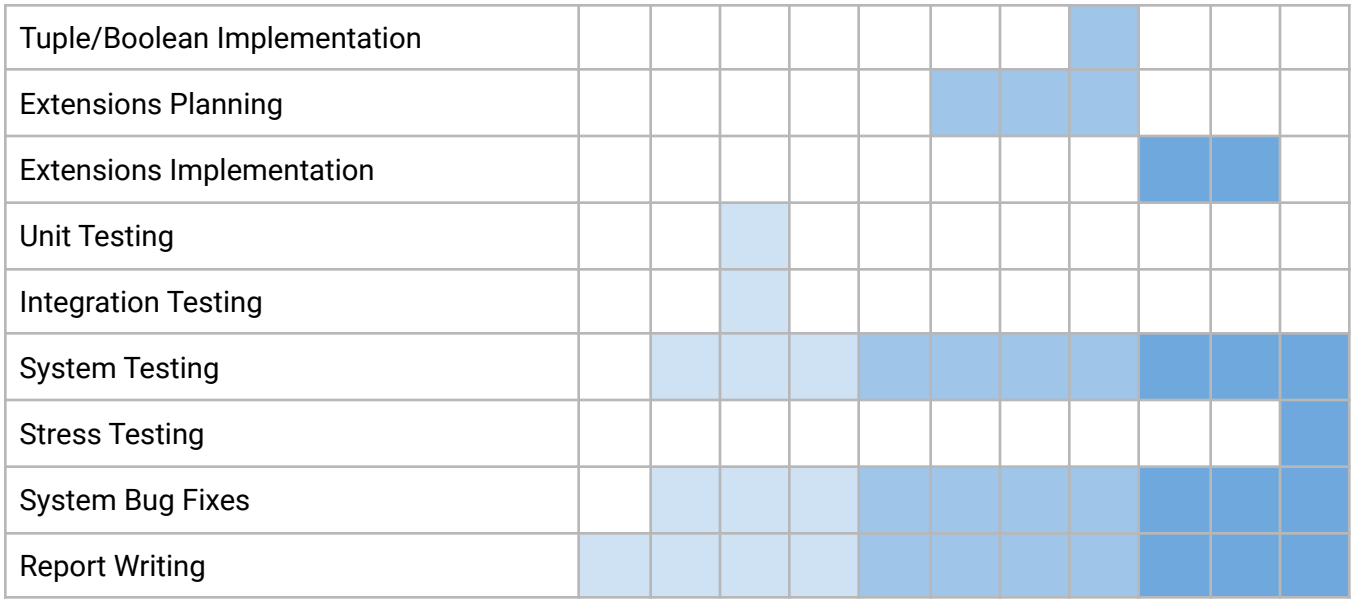


Fig. 2.2: An overview of the milestone of our project main development tasks

2.3. Activity Breakdown and Responsibilities

Activity	Completed By:					
	Jefferson	**Adam**	**Jaime**	**Rachel**	**Su**	**Anqi**
Front-End Parser Implementation Task						
Tokenizer Implementation		*		*		
Parser Implementation		*	*			
AST API		*	*			
Program Knowledge Base Implementation Task						
Design Extractor		*				
PKB Storage API		*				
Query Processor Implementation Task						
Refactor QPP for full PQL grammar						*
Refactor QuerySyntaxChecker						*
Refactor QueryParser						*
Create/Modify REGE X constants for syntax validation						*
Create Intermediate Table APIs					*	*

Implement Intermediate Table		*				
Implement Calls/Calls* Clause		*				
Implement Next/Next* Clause		*				
Implement Affects/Affects*		*				
Implement With Clause			*	*		
Implement If/While Pattern Clause					*	
Implement Tuple/Boolean Select Clause	*				*	
Refactorization of Clauses	*		*		*	*
Rewrite Clauses For Intermediate Table	*				*	
Optimisations					*	*
Extensions Implementation Task						
Extend Parser Implementation			*			
Extend QPP Implementation						*
Extend Design Extractor Implementation		*				
Extend PKB Implementation		*				
Implement For/Do Pattern Clause					*	
Implement NextBip/* and AffectBip/AffectsBip* Clause	*	*				
Unit Testing Task						
Implement SPA Controller Unit Tests				*		
Implement Tokenizer Unit tests				*		
Implement Parser Unit Tests			*			
Implement PKB Storage Unit Tests	*	*				
Implement AST API Unit Tests			*			
Implement Design Extractor Unit Tests		*				
Implement QueryPreProcessor Unit Tests						*
Implement QuerySyntaxChecker Unit Tests						*
Implement 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 • Follows/Follows*/Parent/Parent*/ ModifiesS/UsesS • Assign Pattern clause • Pattern and such that clause			*	*		
System Tests that covers for • Tuple/Boolean Select Clause • With clause • Such that clauses (Next/Next*, Calls/Calls*) • ModifiesP and UsesP Clause • Assign Pattern clause • While/If Pattern Clause				*		
System Tests that covers for • Affects/Affects* • Multicauses	*			*		
Stress Tests				*		
Extension system tests that covers for • NextBip/NextBip*/AffectsBip/AffectsBip* • Do-While Loops, For-Loops	*		*			
Source Focused System Tests • Invalid Source • Complicated Source				*		

Fig. 2.3: An overview of the activity development plan

2.4. Task Breakdown

	Iteration 1				
	Week1&2	**Week 3**	**Week 4**	**Week 5**	**Week 6**
Jefferson Sie	Plan and Design System APIs	Implement PKB storage APIs for Follows, Follows*, Parent, Parent*, Uses, Modifies	Unit Testing Implementation: - PKB storage APIs	QP Query Evaluator implementation for relational clauses: - Follows, Follows*, Parent, Parent*, Uses, Modifies	Integration Testing implementation: - PKB-QP, in particular, evaluate functions of relational clauses Report Writing: - Query evaluator relational clauses
Rachel Tan	Plan and Design System APIs	Implement SPAController, Tokenizer Subcomponent Plan and Design system test	Unit Testing implementation: - Token class - Tokenizer component - SPA Controller Implement system test: - Select and return clause	Implement system test: -Follows/Follows*/Parent/Parent*/ModifiesS/UsesS - Assign Pattern clause	Implement system test: - Pattern and such that clauses Report Writing: - Tokenizer component - Development plan - Test plan and System test
Jaime Chow	Plan and Design System APIs	Implement Parser subcomponent Implement AST API	Unit Testing Implementation: - Parser Class - AST/TNode Class Integration Testing Implementation: - Front End/PKB Integration	Implement System Test: -ModifiesS/UsesS	Report Writing: -Parser component -SPA Architecture overview -Component Overviews -API documentation -Documentation and Coding Standards
Chen Su	Plan and Design System APIs	Implement Query Parser, REGEX validation, Query Evaluator, Query Result Projector Design Clause Class Structure	Implement Query Parser, REGEX validation, Query Evaluator, Query Result Projector Implement Clause, Follows, Pattern Classes	Implement merging algorithm for multiple clauses Implement Unit Testing: - Query Parser - Query Evaluator - Query Result Projector - merging results Implement Integration Testing: - PKB-PQL: eval function in Pattern Clause	Fix merging algorithm for multiple clauses Report Writing: - PQL overview - Query Evaluator - Query Result Projector - merging algorithm for multiple clauses
Chen Anqi	Plan and Design	Implement	Implement	Design merging	QPP and QSC bug

	System APIs	QueryPreProcessor, QuerySyntaxChecker	QueryPreProcessor, QuerySyntaxChecker Implement validate functions in Clause classes	algorithm for multiple clauses evaluation Refactor QPP and Query for merging multiple clauses Implement Unit Testing: - QueryPreProcessor - QuerySyntaxChecker - Clause::validate	fixes Report Writing: - PQL overview - Query Pre Processor - Clause class - Unit testing for QPP and Clause
Adam Tan	Plan and Design System APIs	Implement Design Extractor	Implement Design Extractor	Implement Unit Testing: - Design Extractor	Report Writing: - PKB overview - Clause algorithms

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

	Iteration 2			
	R	**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 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 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	Design Query Result	Refactor PatternClause	Refactor QE and QRP

	development schedule for Iteration 2	Class (the intermediate table to store results) and algorithm of merging results from multiple clauses.	Implement PatternWhile and PatternIf Classes.	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. 2.4-2: An overview of the development task breakdown for iteration 2

	Iteration 3		
	Week 10	**Week 11**	**Week 12**
Jefferson Sie	Fix bugs in Query Evaluator relational clauses found during system tests from iteration 2 Update Integration Tests: - Follows, Follows*, Parent, Parent*, Uses, Modifies, Calls, Calls*	Continued fixing bugs in Query Evaluator relational clauses Implement System Test: - Implement complicated Affects/Affects* clause	Implement System Test: - Implement AffectsBip clause Report Writing: - Write Affects* - Write NextBip, NextBip* - Write For loop (PKB) - Write Do-While loop (PKB) - Update development plan - Update Query Evaluator relational clauses - Update API documentation - Update discussion
Rachel Tan	Implement system test: - Affects/Affects* clause	Implement system test: - Multi causes group c1 - Multi causes group c2 - Multi causes group c3 - Multi causes group c4 - Multi causes group c5 - Multi causes group c6	Implement Stress Test: - Select Tuple stress test - Multiple clauses stress test - 500 lines source and multicauses query stress test Resport Writing:

		View group details in Test Category 9: Multi Clauses	- Update development plan - Update Test plan and System test
Jaime Chow	Extend Components for Extension: - Do-While parsing for Parser - For parsing for Parser - ForNode and DoNode for AST class - Extend AST API Extend Unit Testing: - Parser Component - AST API	Implement System Test: - For/Do-While Extensions Refactor WithClause to accommodate new QP implementation Refactor Next/Next* clause Integration tests to accommodate new QP implementation	Implement System Test: - NextBip/NextBip* Extension Update Front-End/PKB Integration tests: - Affects/Affects* Optimize Query Result class Report Writing: - Write For/Do-While Extensions(Parser) - Write System Testing for Extensions - Update Component interaction
Chen Su	Fix bugs in QE and pattern clauses found from iteration2 system testing Extend Unit Testing: - PatternWhile/PatternIf - Query Evaluator - Query Result Projector	Design Optimization Strategy Implement Query Optimizer Implement Unit Testing: - Query Optimizer Implement For/Do-While Clause Classes	Create QueryResult APIs for sub-table implementation Refactor QE for merging results from sub-table Report Writing: - Optimization - Update QE - Update QueryResult
Chen Anqi	QPP bug fixes for iteration2 system testing Extend Unit Testing: - QueryPreProcessor - QueryParser - QuerySyntaxChecker - Clause::validate	Design Optimization Strategy Implement Query Optimizer Implement QPP extension (For/Do, AffectsBip, AffectsBip*, NextBip, NextBip*) Implement validate functions for iteration 3 extension	Implement unit testing for QPP and Clause for iteration 3 extension Report Writing: - Optimization - Update QPP - Update unit testing - Update extension
Adam Tan	Implement Affects/Affect* Clauses Implement NextBip/NextBip* Clauses	Implement AffectsBip/ AffectsBip* Clauses Implement For/Do-While Design Extraction Implement For/Do-While Pattern Matching	Optimize Affects/Affects*/Next* Optimize QueryResult intermediate table Report Writing: - Write Affects* - Write NextBip, NextBip* - Write For loop (PKB) - Write Do-While loop (PKB)

Fig. 2.4-3: An overview of the development task breakdown for iteration 3

2.5. Software Engineering Principles in Planning Process

2.5.1. Overview

Apart from the iterative nature of the SPA project, our group adapts heavily from the Agile framework[1], adapting to changes swiftly and without hesitation. Since before the project even began, we were prepared to divert resources to different components, especially the Query Processor, any time in the semester as we understood that the workload will be heavy there. In iteration 1, we had 2 members assigned to the Front-end component, 2 members to PKB, and 2 members to QP. However, there was a point in iteration 2 where all 6 team members contributed to the development of the Query Processor.

2.5.2. Iteration 3

For iteration 3 in particular, we allocated work based on experience in the component. What was to be implemented in iteration 3 were Affects, Affects* clauses, optimisations, extensions, system tests for all these components as well as stress tests for our system. Adam is our algorithm expert and has been responsible for the implementation of design extraction algorithms, thus he was tasked to implement the clauses. Chen Su and Anqi have been responsible for the QP component from the start, and are the most familiar with how clauses are evaluated, thus they were tasked with optimisation. Rachel, Jaime and Jefferson were allocated the remaining work, with most of the focus in writing complex and elaborate system tests to ensure the correctness of our system.

The emphasis our team puts on taking advantage of each other's strengths ensures that tasks are completed with the highest level of efficiency with the least time taken. This flexibility and agility to adapt to changes throughout the semester has allowed Team 21 to consistently perform well.

[1] https://www.agilealliance.org/

3. SPA Architecture

3.1. SPA Architecture Overview

3.1.1. Architecture Component

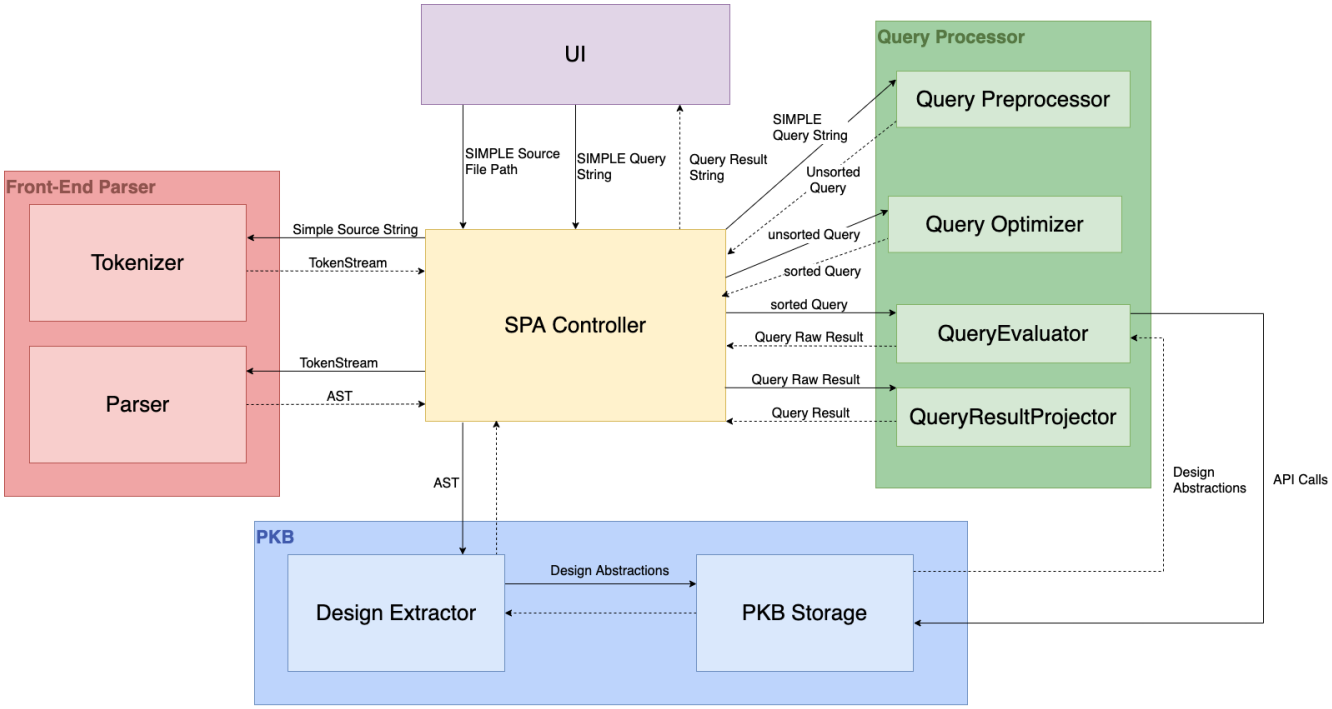


Fig. 3.1-1: Overview of SPA Architecture Component

Our SPA consists of four main components and their respective subcomponents:

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.
    a. **Tokenizer -** Converts the Simple Source program into a token stream.
    b. **Parser -** Generates the AST based on a token stream 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.
    a. Design Extractor - Extracts the design abstractions from the AST input and stores them in the PKB Storage.
    b. PKB Storage - Stores the design extractions in internal data structures and retrieves the appropriate information depending on the API calls.
4. **Query Processor** - Processes and evaluates a SIMPLE query input, and outputs the formatted query result.

a. **QueryPreProcessor**, which handles the pre-processing of query string and packaging into Query struct, and performs semantic checks on the query.
b. **QueryOptimizer**, which handles the sorting and grouping of the clauses to make the evaluation of the clauses more efficient.
c. **QueryEvaluator**, which takes in a Query object and evaluates the raw result.
d. **QueryResultProjector**, which formats the raw results and returns the result to I/O.

3.1.2. Optimizations

For iteration 3, we have implemented features in three SPA Components as a means of optimizing the query evaluation of the SPA Program:

1. **Query Preprocessor -** The Query Preprocessor has been updated to check for and remove duplicate clauses. Implementation is elaborated in section 3.5.2.4.
2. **Query Optimizer -** The Query Optimizer sub-component has been implemented to group and reorder clauses into and within groupings. This sorting is based on their similar characteristics, impact on the number of results and ease of evaluation. Implementation is elaborated in section 3.5.3.
3. **PKB -** The PKB has been updated to, for each query, cache the results of evaluated relationship clauses that are computed on the fly. Affected clauses include Next*,Affects,Affects*, NextBip*,AffectsBip and AffectsBip*. Implementation is elaborated in section 3.4.3.3.

3.2. Component Interaction Overview

The interaction between components are as follows:

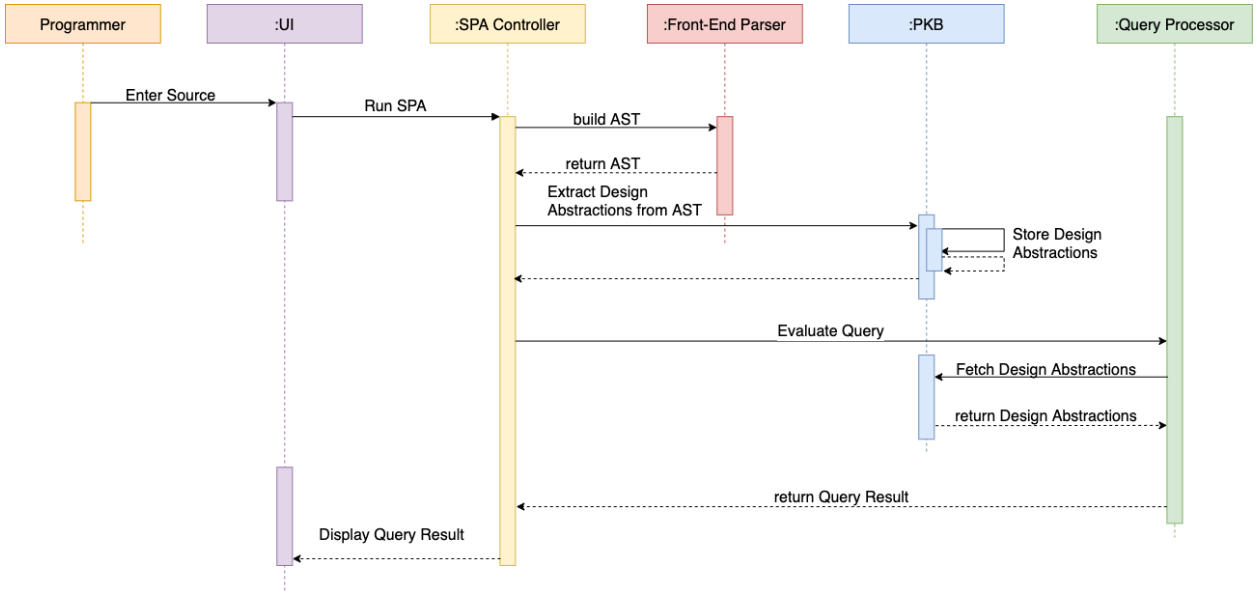


Fig. 3.2-1: High Level UML Sequence Diagram of SPA Program

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.

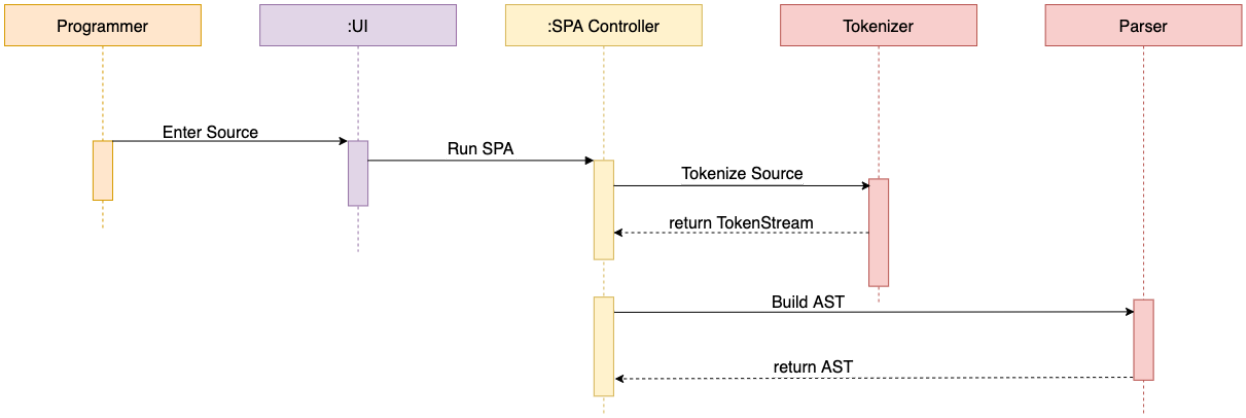


Fig. 3.2-2 High Level UML Sequence Diagram of Front-End Parser

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.

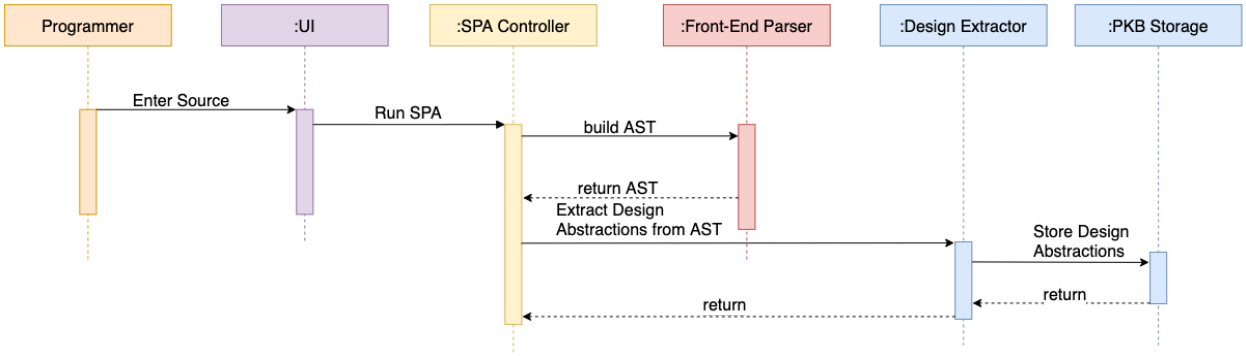


Fig. 3.2-3 High Level UML Sequence Diagram of PKB

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 an unsorted Query object. The SPA Controller then passes the unsorted Query object to the Query Optimizer, which sorts the clauses within the Query object and returns a sorted Query object. The SPAController then passes the sorted Query object to the Query Evaluator which 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.

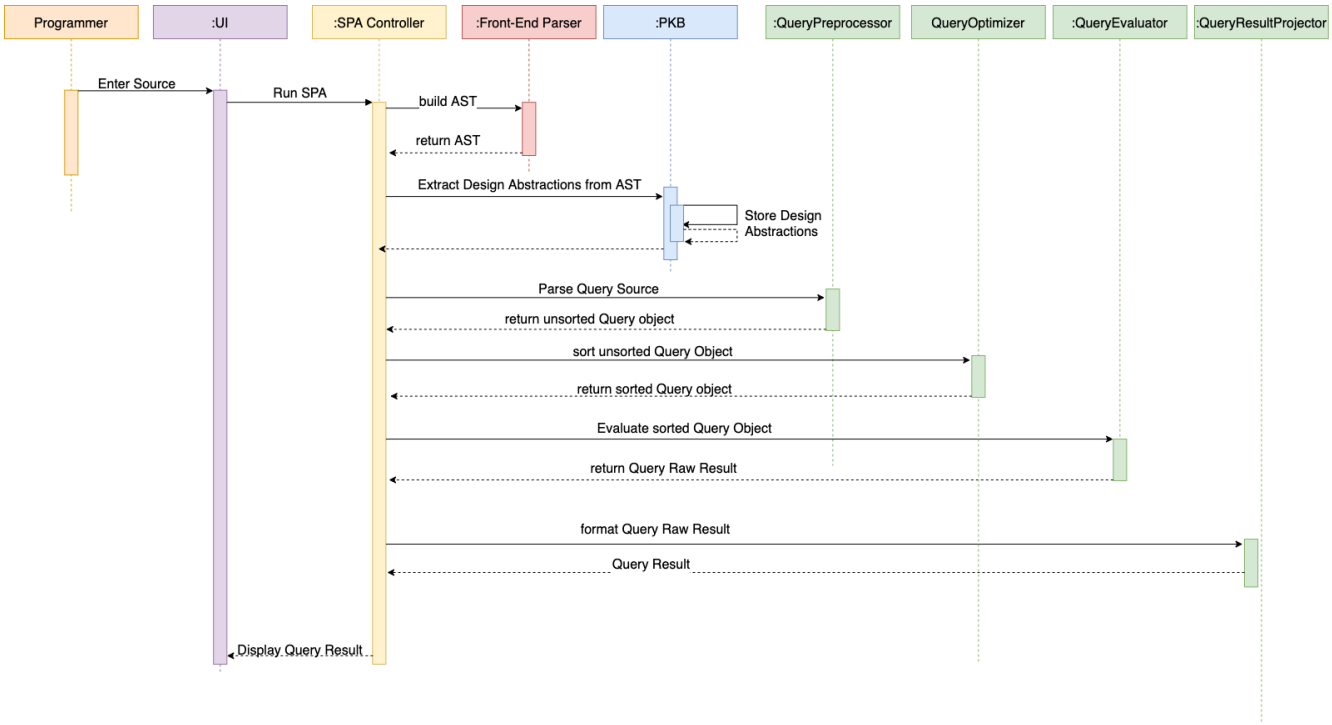


Fig. 3.2-4 High Level UML Sequence Diagram of Query Processor

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.

The responsibility of checking cyclic procedure calls and procedures with the same names has been shifted to the Design Extractor in the PKB, as the errors can be detected during design extraction.

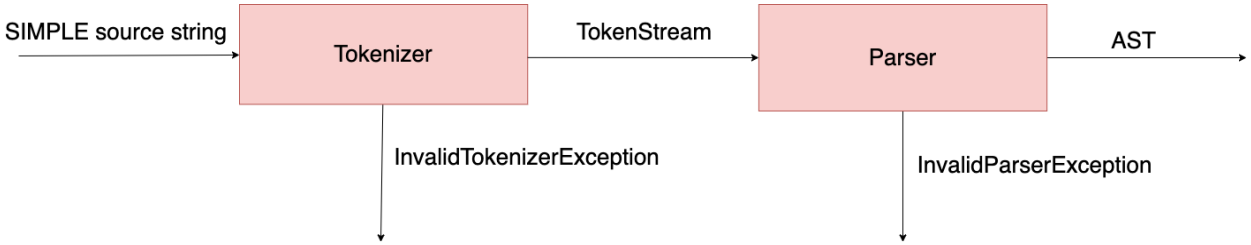


Fig. 3.3.1-1: Production Pipeline of Front End Parser Architecture

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.

Tokenizing Logic

The Tokenizer will start iterating through the source program, it will get the Current and next character called currChar and nextChar respectively. It will then form the current and next characters into a string called tempToken. Finally, it will append any characters or tempToken that does not satisfy as a token value called currToken

Case 1: It start by checking if `tempToken` is an **expression** token value

- 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`
- 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`
- Update the next `currChar` and `nextChar`

Case 2: `currChar` is in **Operator** or **Separator** token

- Send `currChar` to Token class for grammar syntax rule validation and create Token object of Operator or Separator type and finally, appended to `TokenStream`
- Update the next `currChar` and `nextChar`

Case 3: `nextChar` is in **Operator** or **Separator** token

- Send `currToken` to Token class for grammar syntax rule validation and create Token object of Operator or Separator type and finally, appended to `TokenStream`
- Update the next `currChar` and `nextChar`

Case 4: `currChar` is a space and currToken has characters, this means `currToken` is a **complete token value.**

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`

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**

- Update the next `currChar` and `nextChar`

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**

- `currChar` is sent to Token class for grammar syntax rule validation and to create a Token object and finally, appended to `TokenStream`

Once the Tokenizer has finished iterating and tokenizing the source program, an end Token object is created and appended to the TokenStream. The example below illustrates how the Tokenizer return the corresponding TokenStream.

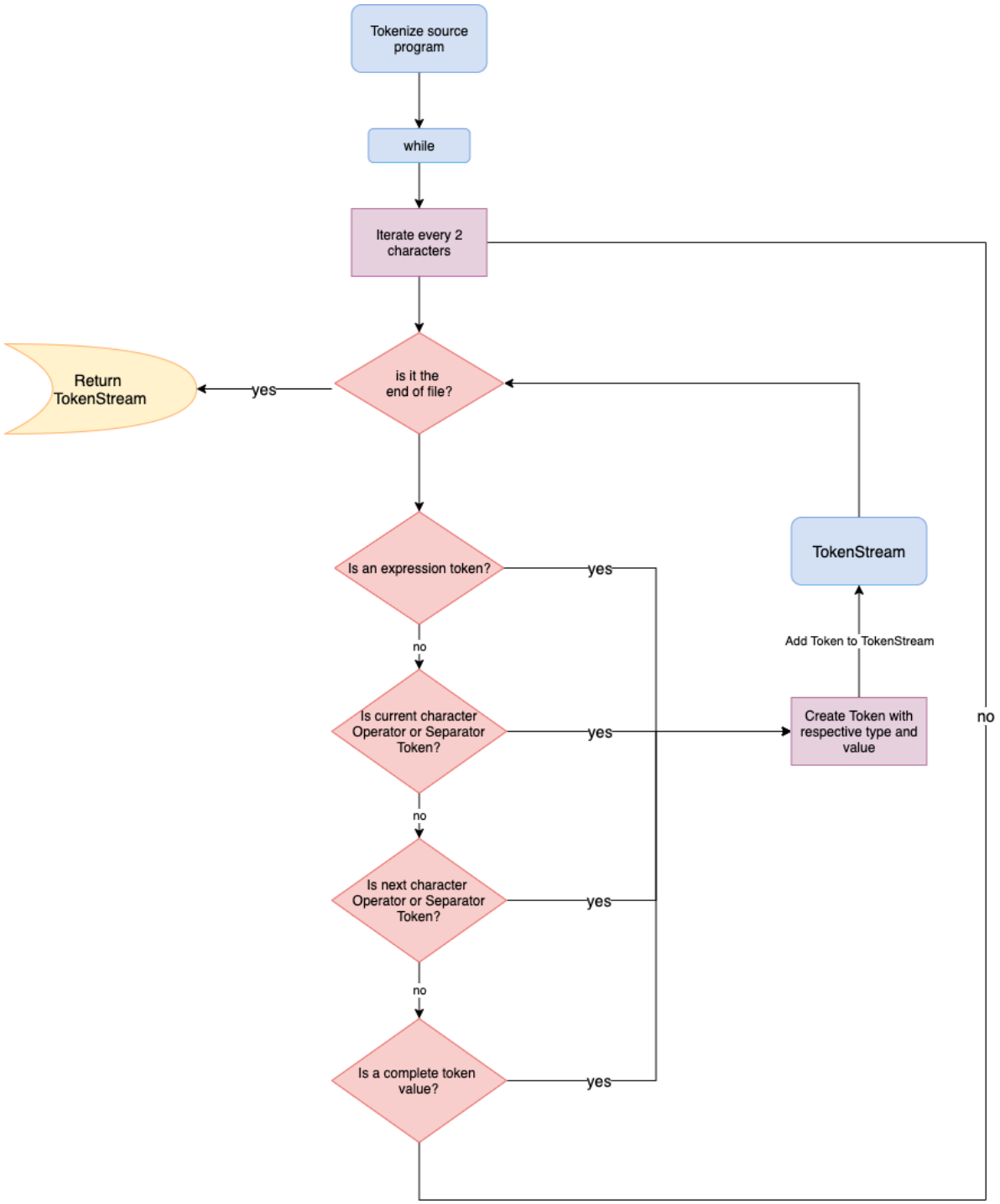


Fig. 3.3.2.3-1: High-level Flowchart of Tokenization Sequence

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.

Creation of Token object Logic

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

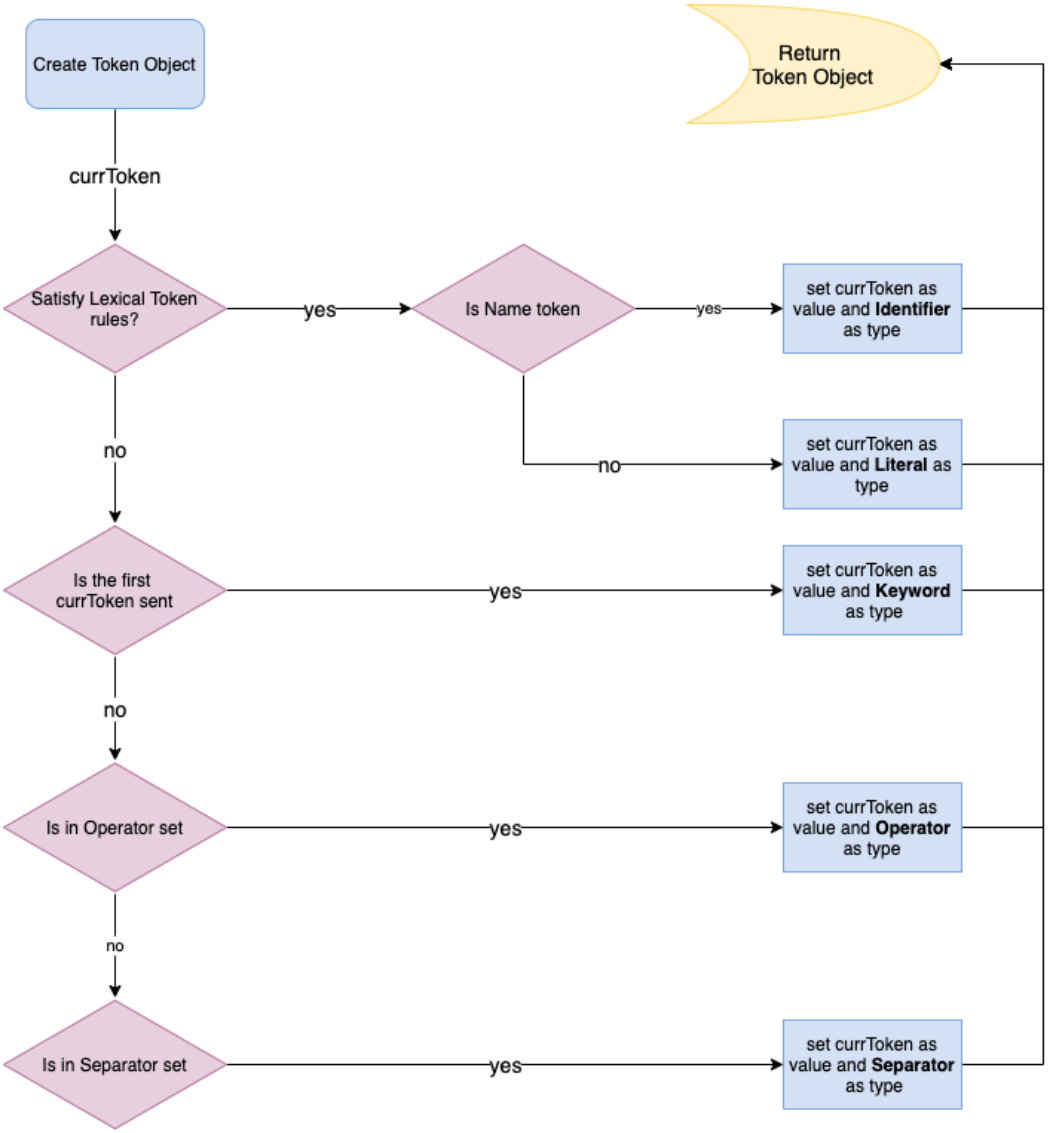


Fig. 3.3.2.3-4: High-level Flowchart of Creation of Token object

Example

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.

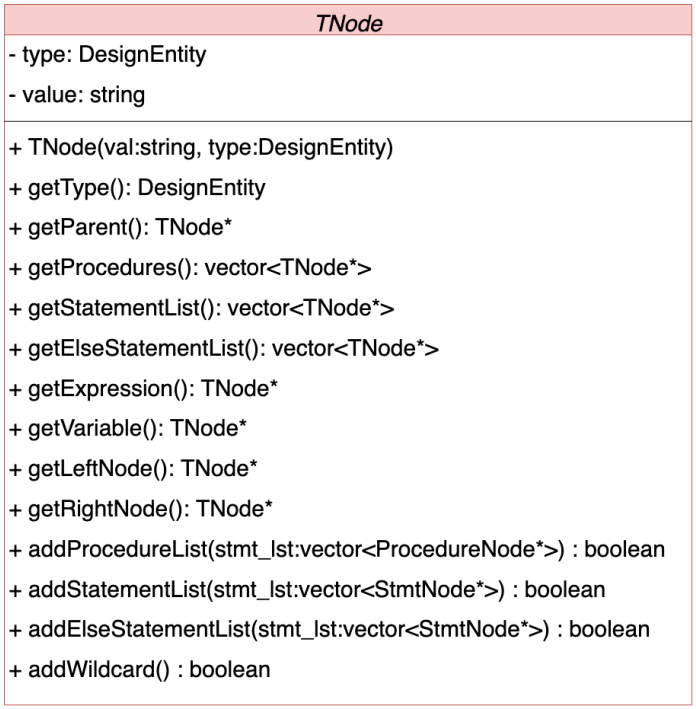


Fig. 3.3.3.2-1: Class Diagram of the abstract TNode Object

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

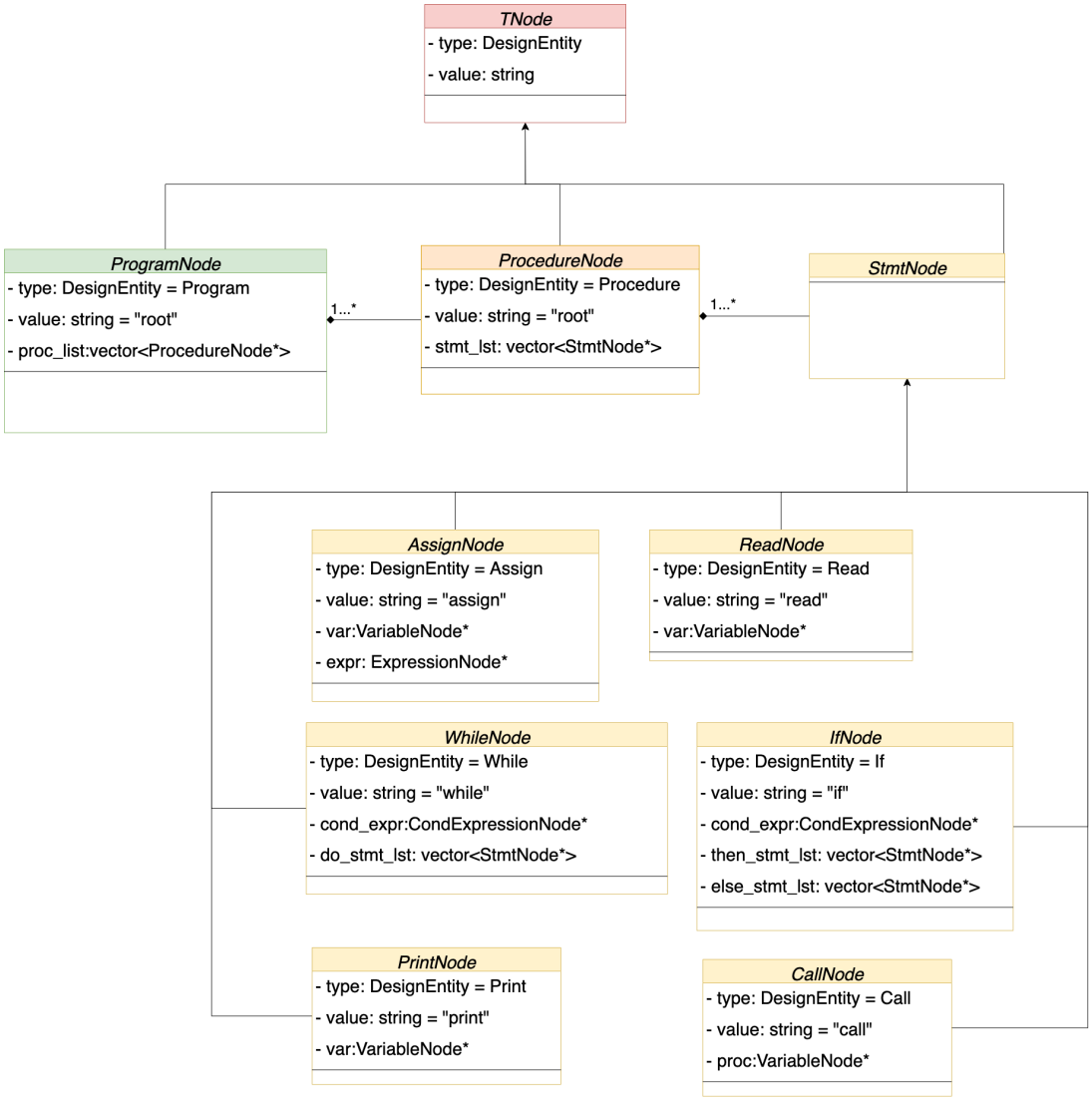


Fig. 3.3.3.2-2: Nodes objects generated by general Program Parsing

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

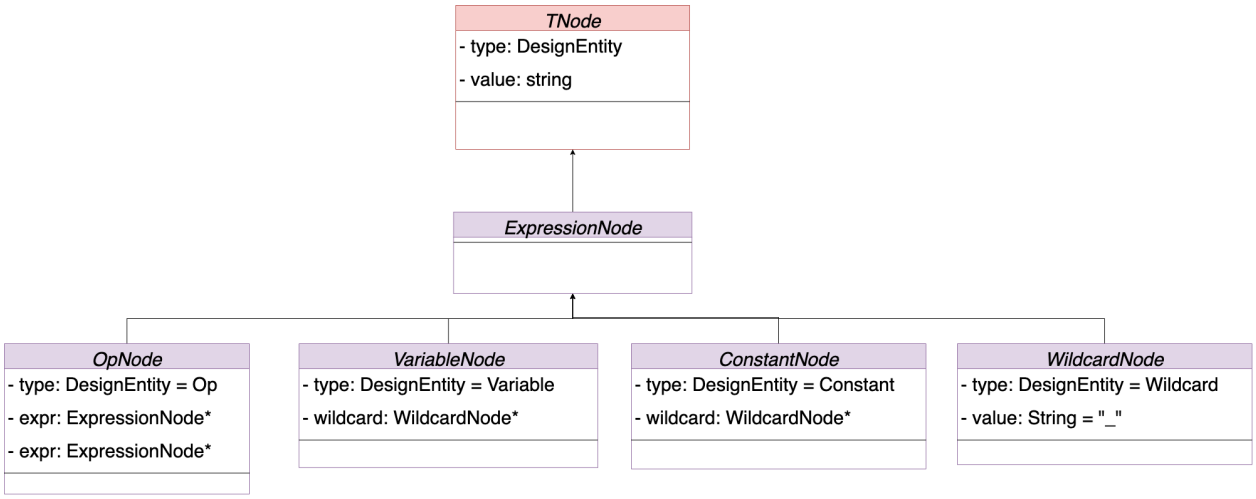


Fig. 3.3.3.2-3: Node objects generated by Expression Parsing

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

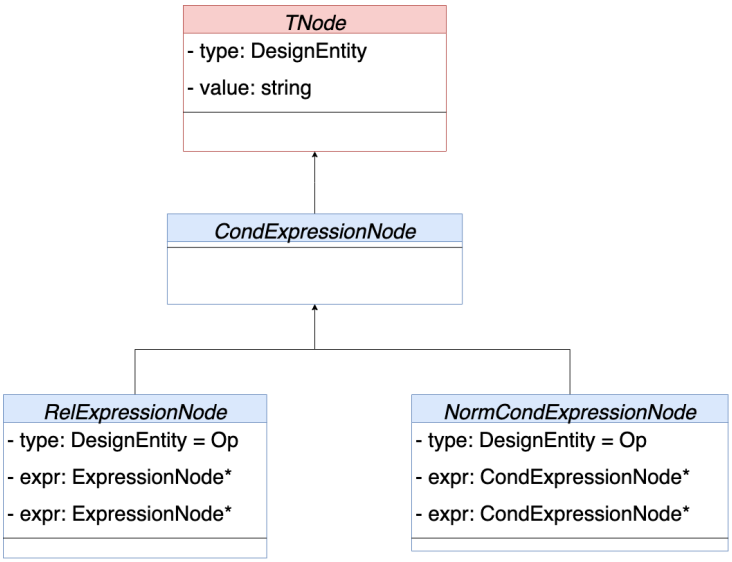


Figure 3.3.3.2-4: Node objects generated by Conditional Expression Parsing

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.

Example

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 of Programs

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 parsing of multiple procedures 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.

(4) Parsing Programs with Multiple Procedures

For the parsing of multiple procedures, we utilize a token of type “EndOfFile”, which is appended to the end of every Token stream. When a procedure is finished parsing, the parser checks for a next token type of “EndOfFile”. If this condition is fulfilled, the program assumes that there are no further tokens and ends the parsing process. Otherwise, the parser assumes that there are more procedures to be parsed and calls the method to parse a procedure again.

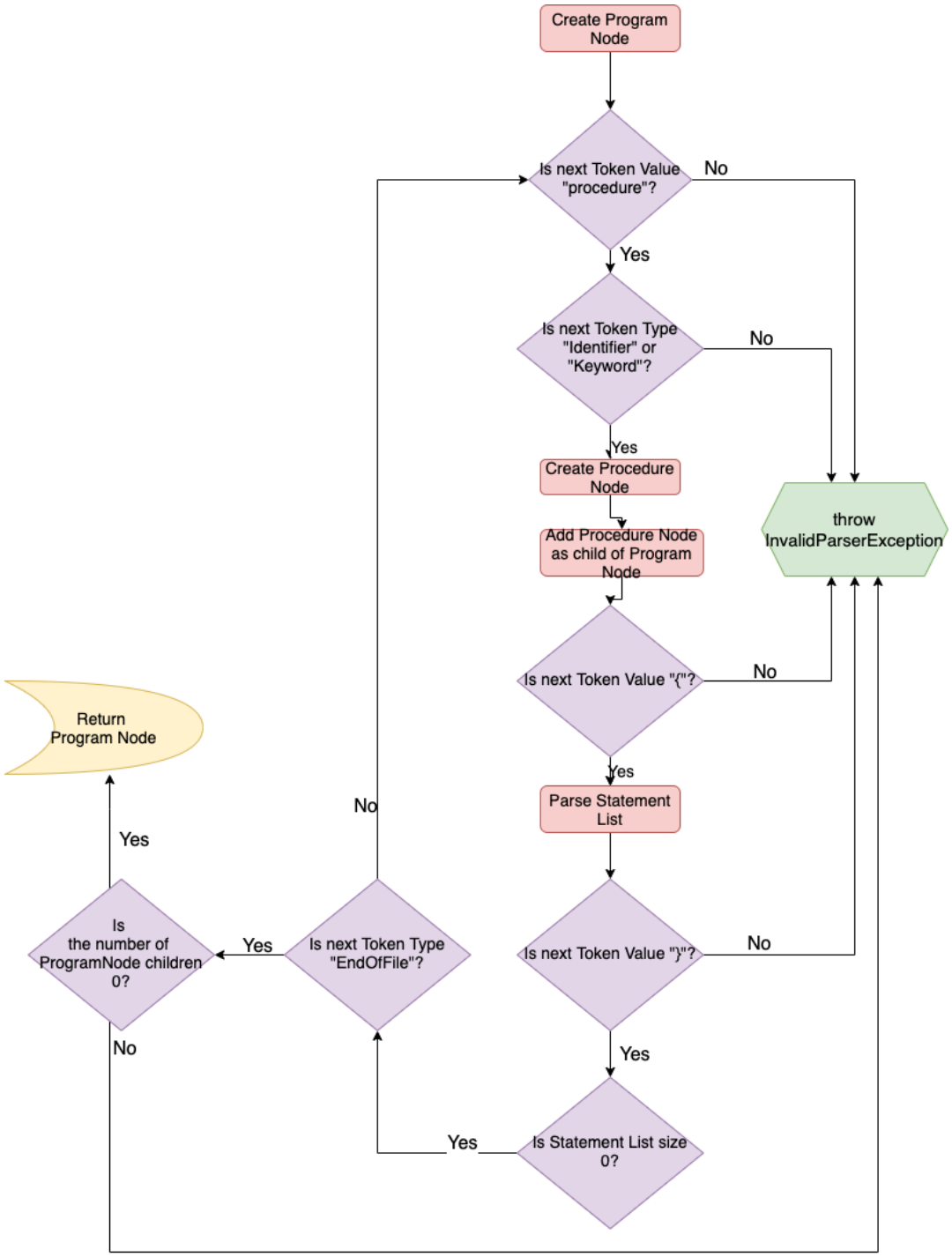


Fig. 3.3.3.3-3: High-level Flowchart of parsing mulitple programs

Example

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

```
procedure one {
```

```
      read x;
}

procedure two{
      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	-	Token Type “EndOfFile” not detected, Parser continues parsing. Parser makes call to parse procedure non-terminal
‘procedure’	“procedure”	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.
‘}’	‘}’	Token value matches ‘}’. Parser returns vector<StmtNode> object denoted as stmt_lst, containing stmt_1 As stmt_lst has a size of 1, an error is not thrown. Parser returns ProcedureNode of value “one” and child node stmt_lst, denoted as one_node, and adds it as a child of the ProgramNode.
procedure	-	Token Type “EndOfFile” not detected, Parser continues parsing.
‘procedure’	“procedure”	Token value matches ‘procedure’.
proc_name	“two”	Token Type is “Literal” or “Keyword”
‘{‘	“{“	Token value matches ‘{‘.

stmtLst		Parser makes call to parse stmtLst non-terminal
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_2. As stmt_lst has a size of 1, an error is not thrown. Parser returns ProcedureNode of value “one” and child node stmt_lst, denoted as two_node, and adds it as a child of the ProgramNode.
-	“EOF”	“Token Type “EndOfFile” detected, Parser ends parsing. Parser returns ProgramNode with child one_node and two_node. As ProgramNode has 2 children, 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

(5) 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:

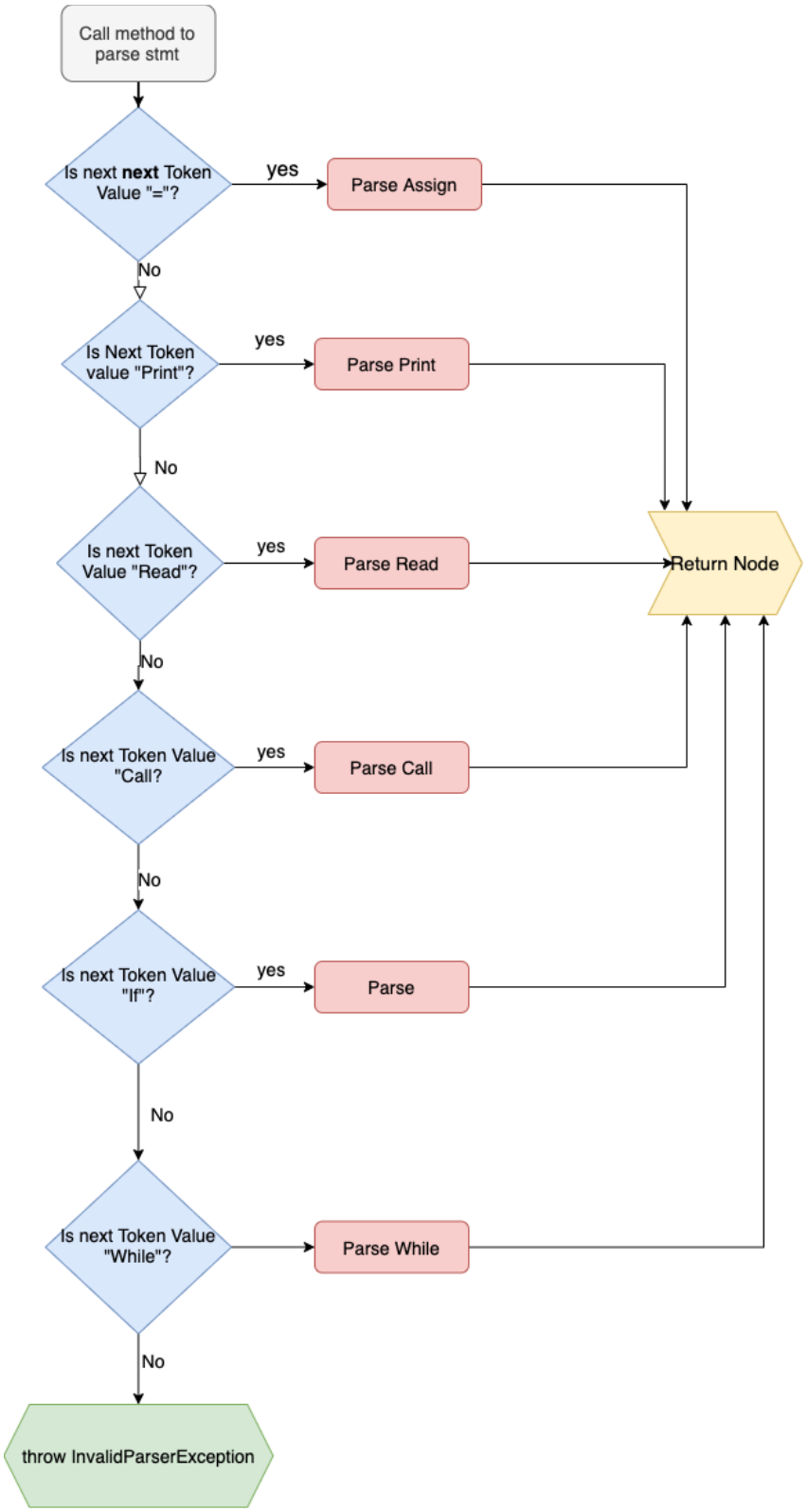


Figure 3.3.3.3-6: Flow Chart for parsing stmt non-terminal

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:

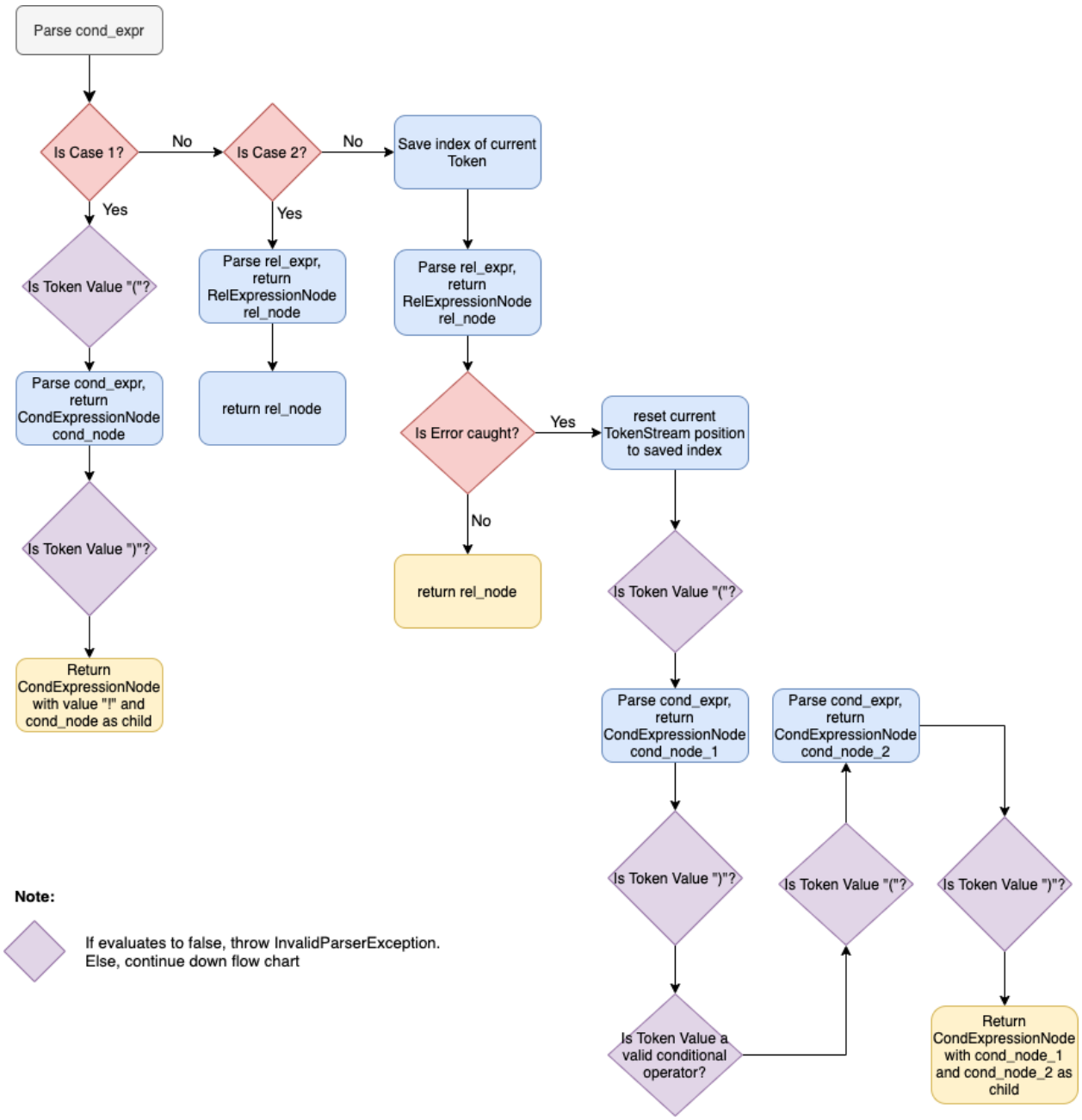


Fig. 3.3.3.3-8: Flowchart of parsing cond_expr non-terminal

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.

(6) 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.

Example

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 with the same names. These will be explained later in this section.

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

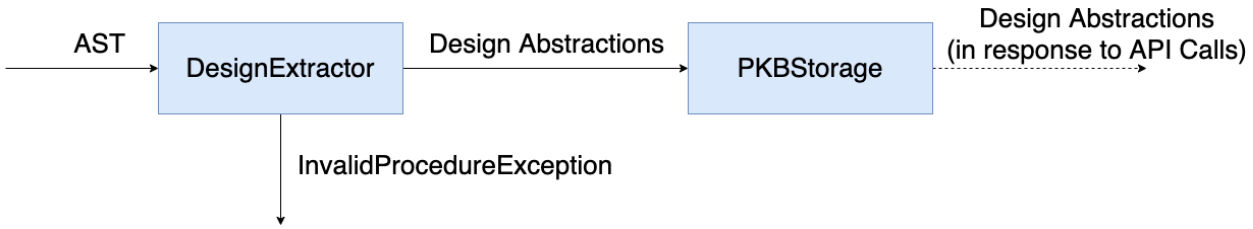


Fig. 3.4.1-1: Overview on PKB Architecture

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.

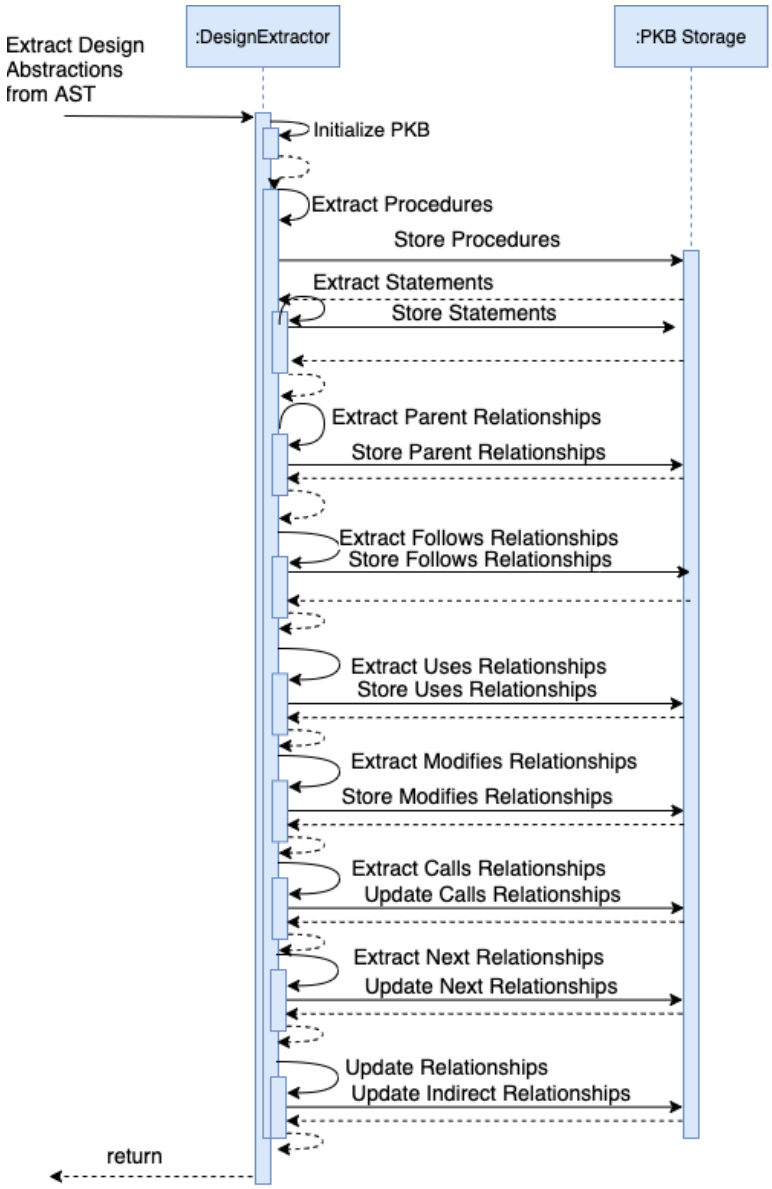


Fig. 3.4.1-2: Sequence Diagram for initializing PKB

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+1;
      }
}
```

Fig. 3.4.2.2-1: Sample Source

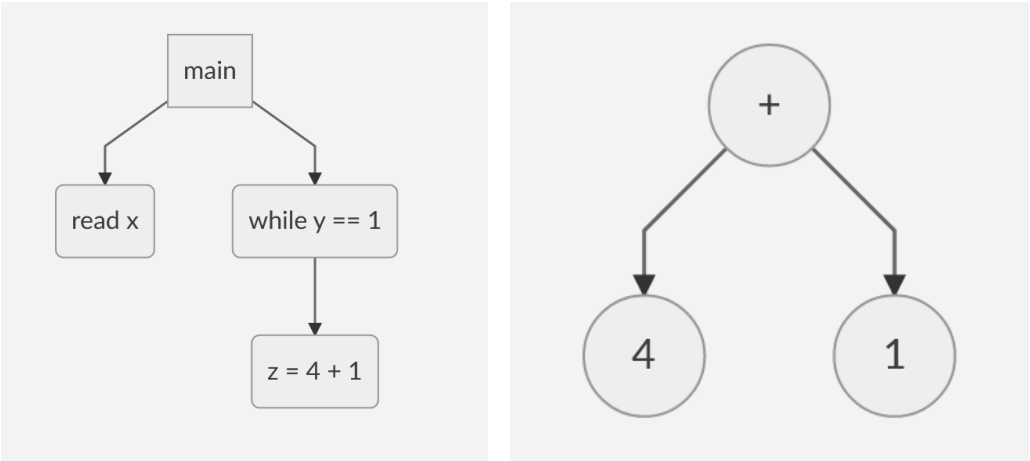


Fig. 3.4.2.2-2: Simplified AST generated from Fig. 3.4.2.2-1

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}.

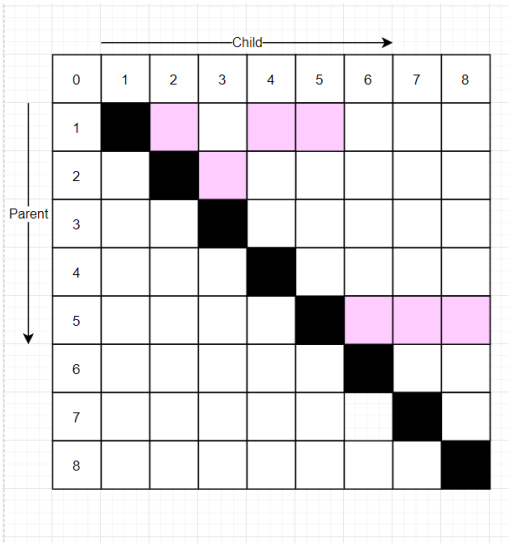


Fig. 3.4.2.2-6: Pink squares for parent relationship

As we perform an in order traversal of the AST, each statement under the parent statement is given an index and 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 Parent and inverse relationships are stored in unordered maps where the statement numbers act as they key for indexing the maps

Example: Extracting Follows Relationship

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 the 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.

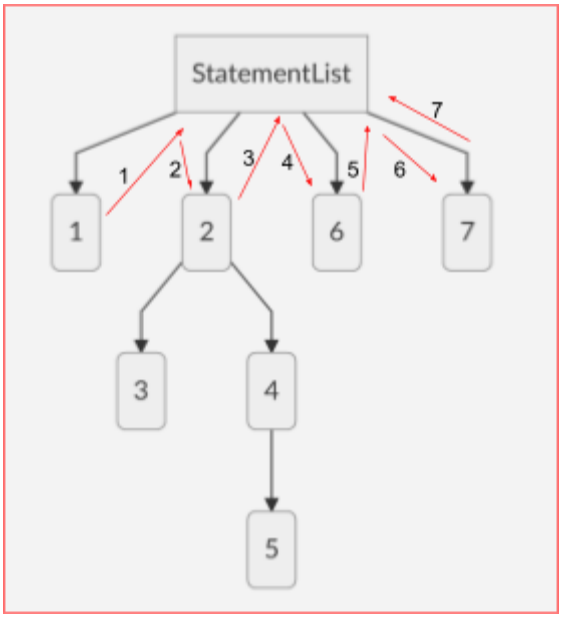


Fig. 3.4.2.2-7: Order of traversal when extracting Follows Relationships

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}

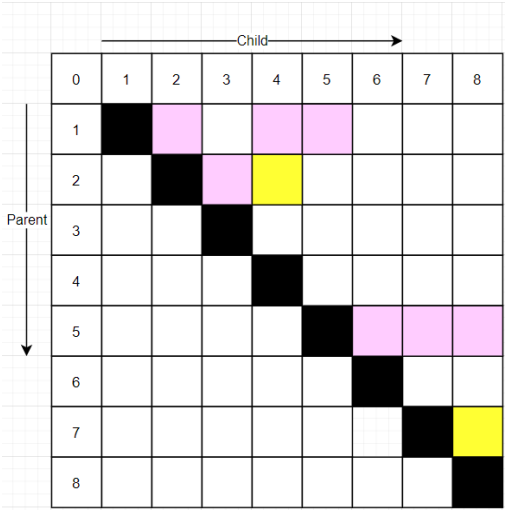


Fig. 3.4.2.2-8: Yellow squares for follows relationship

Example: Extracting Uses 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.

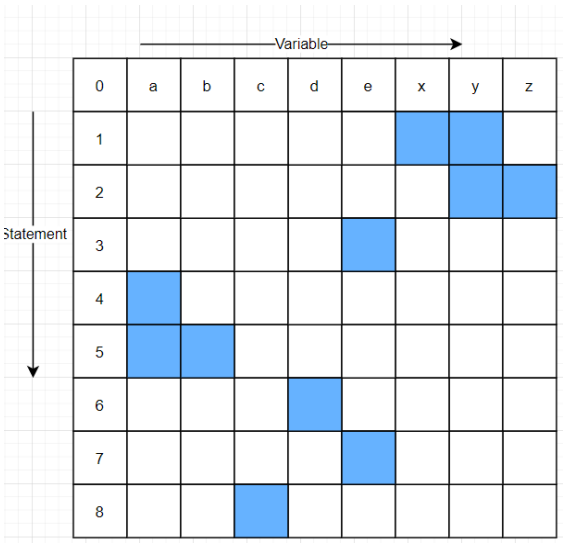


Fig. 3.4.2.2-9: Blue squares for primary uses relationship

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.

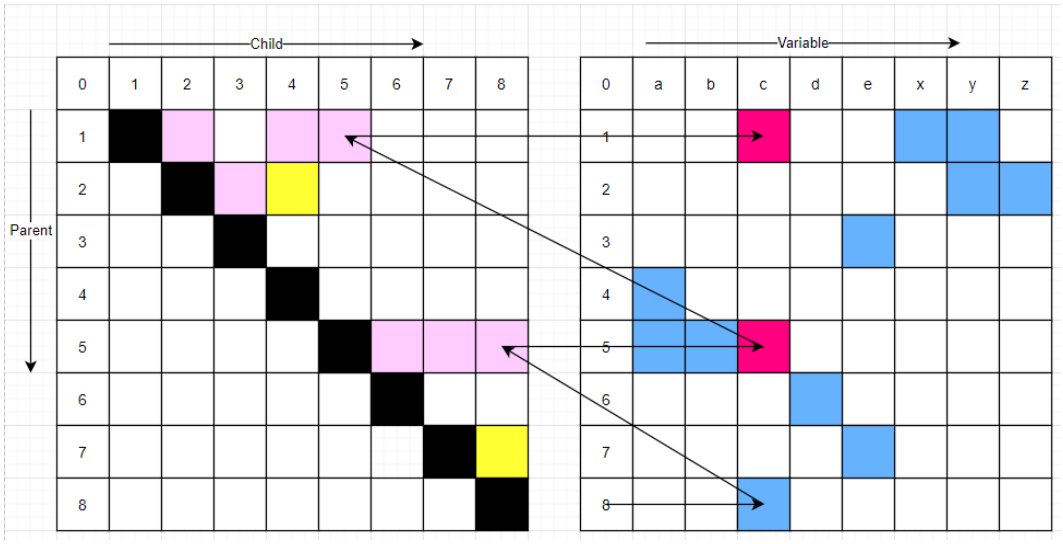


Fig. 3.4.2.2-10: Red squares for secondary uses relationship, arrows indicate the order of updates

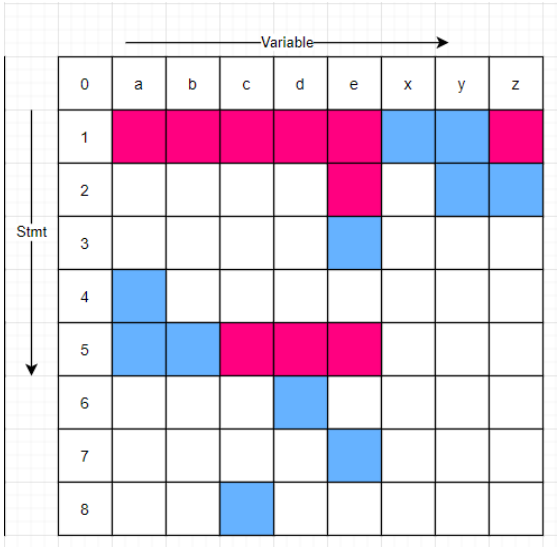


Fig. 3.4.2.2-11: Final uses table

Example: Extracting Follows* Relationship

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

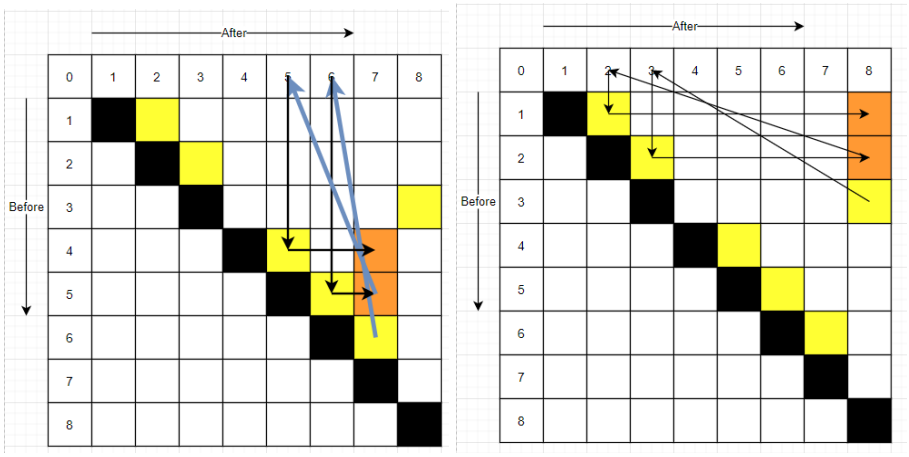


*Fig. 3.4.2.2-13: Updating of follows star relationship - Yellow squares for Follow, Orange squares for Follow**

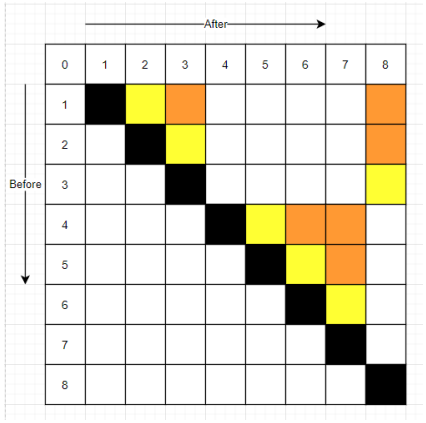


Fig. 3.4.2.2-14: Completed follows star table

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	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

uses_list modifies_list	

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*

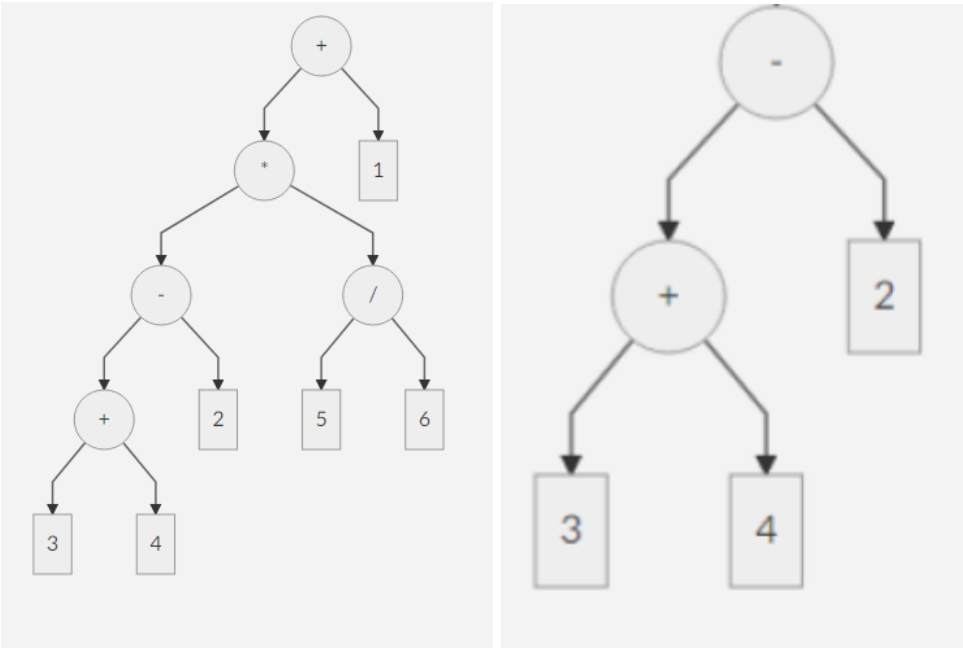


*Fig. 3.4.3.3-2: The AST for the equation : (((3 + 4) - 2) * (5 / 6)) + 1 and (3 + 4 - 2) * (5 / 6) + 1 and subtree _"(3 + 4) - 2"_*

To search for the subtree we start from the root node, we compare the "+" 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*. For specific examples i.e Next*(int, int) , Next*(a, int) ...

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

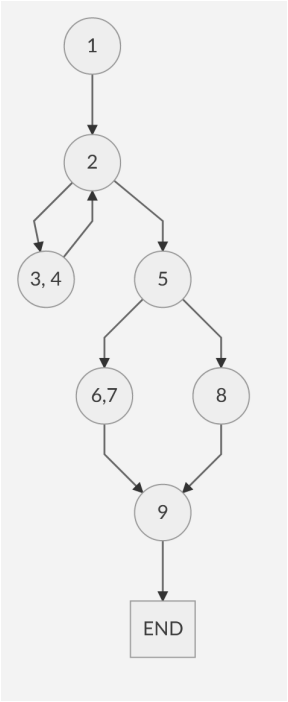


Fig. 3.4.3.3-4: Resulting CFG

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. As an optimization strategy, since if Next(a1, a2) is true, then all the Next* of a2 will also be the Next* of a1. So if the Next* of a2 is already evaluated, then we do not need to reevaluate from a2 onwards and we can just retrieve it from the cache. To maximize the occurrences of accessing the caches, for all Next(a1, a2) queries we evaluate the possible a1 from the largest to smallest statement numbers. This ensures that most proceeding evaluations of Next(a1) will access the cache the moment it reaches the already evaluated statements.

Category	Case	Algorithm
1	Next*(1, 2)	Run the algorithm in category 2 on statement 1 and check if statement 2 is in the result
2	Next*(1, s)	Run a DFS on the CFG generated by the Next relationships and add all reachable nodes into the answer. Cache the result for future DFS.
3	Next*(s, 1)	Run a DFS using the CFG generated by the Prev relationship and add all reachable nodes into the answer. Cache the result for future DFS
4	Next*(s1, s2) Next*(s, s)	Carry out the algorithm in category 2 on all statements starting from the largest statement

		number to smallest. If DFS reaches a node that has already been cached, search stops there and the cached values is added to the answer and the DFS terminates at that node without searching the following nodes. Merge all evaluations into a single set and return the answer. Since the largest numbers are evaluated first the hit rate of the cache greatly increases, increasing the number or early terminations and speeding up the evaluation

Fig. 3.4.3.3-5: Extracting Next relationships*

(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 and 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 the PKB 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

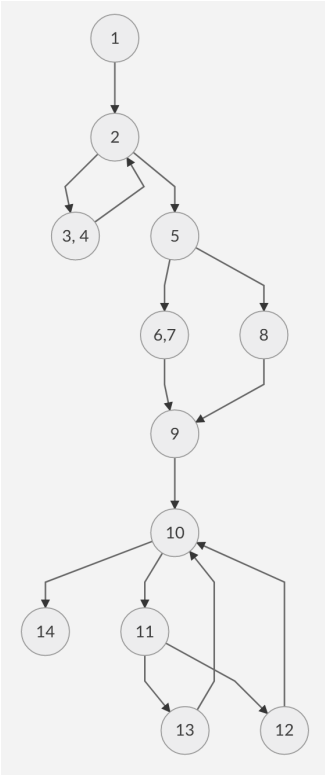


Fig. 3.4.3.3-6: CFG of SIMPLE source

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	I	a	answer.insert(9) continue()
10			continue()
11			continue()
12	i	I	continue()
14	b	a	answer.insert(14) continue()
13	a		stopDFS()

Fig. 3.4.3.3-7: Affects Extraction Process

Category	Case	Algorithm
1	Affects(1, 2)	Run the algorithm in category 2 on statement 1 and check if statement 2 is in the result
2	Affects(1, s)	Run a DFS on the CFG generated by the Next relationships on statement 1, if the node uses the variable modified by 1 then add it into the answer, if the node modifies the variable modified by 1 then terminate the DFS at the node. Cache the result for future DFS.
3	Affects(s, 1)	Run a DFS on the CFG generated by the Prev relationships on statement 1, if the node modifies the variable used by 1 then add it into the answer and terminate the DFS at the node. Cache the result for future DFS.

4	Affects(s1, s2) Affects(s, s)	Carry out the algorithm in category 2 on all assign statements starting from the largest statement number to smallest. Merge all evaluations into a single set and return the answer.

Fig. 3.4.3.3-8: Affects Extraction Process

The algorithm for Affects* builds on top of Affects, but with additional implementations and checks. In Affects, the DFS algorithm stops when an encountered node modifies the modified variable. In Affects*, when this case is encountered, a new DFS algorithm is spawned with that node as the starting point. This allows for the transitive aspect of Affects* to be accounted for.

Example: Affects*

Using Fig. 3.4.3.3-5 as an example, statement 1 affects statement 4, and the DFS algorithm stops at statement 4 as statement 4 modifies the modified variable in statement 1. Therefore, statement 1 does not affect statement 6. In Affects*, a new Affects algorithm is spawned at statement 4, capturing the relationship that statement 4 affects statement 6. As a result, the relationship that statement 1 Affects* statement 6 is captured. The stop condition for the algorithm is only when a modified variable is modified but not through an assign statement, such as a read statement.

Another implementation in Affects* is the caching of evaluated relationships as an optimisation to the algorithm. This significantly reduces computation time in complex cases where there are many affects statements that cascade. Whenever a new DFS algorithm is spawned in the case where an assign statement modifies a modified variable, the cache will be checked to see if this relationship had been pre-computed earlier. If it has, there would not be a need to run the algorithm, and the relationships can be directly obtained from the cache. An implementation detail as a result of the cache is to sort the affects statements in inverse order, from largest to smallest, in order to maximise usage of the cache. This is to maximise the number of affects relationships stored in the cache, so that fewer algorithms will be spawned.

Category	Case	Algorithm
1	Affects*(1, 2)	Run the algorithm in category 2 on statement 1 and check if statement 2 is in the result
2	Affects*(1, s)	Run a DFS on the CFG generated by the Next relationships on statement 1, if the node uses the variable modified by 1 then add it into the answer, if the node modifies the variable modified by 1 then start another instance of the algorithm at the node. Cache all intermediate evaluations and the final evaluation.
3	Affects*(s, 1)	Run a DFS on the CFG generated by the Prev relationships on statement 1, if the node modifies

		the variable used by 1 then add it into the answer andstart another instance of the algorithm at the node. Cache all intermediate evaluations and the final evaluation.
4	Affects*(s1, s2) Affects*(s, s)	Carry out the algorithm in category 2 on all assign statements starting from the largest statement number to smallest. Since all intermediate evaluations and final evaluations are cached, each time we need to check for a following affects* relationship we do not need to traverse the graph again. i.e Each assign statement is only evaluated once and if the assign statement is affected by another assign statement the affects* relationship is extracted at the same time.

Fig. 3.4.3.3-8: Affects Extraction Process*

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 correct ezly, 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 is no 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 is no 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 **seven** 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. **QueryOptimizer**, which handles the sorting and grouping of the clauses to make the evaluation of the clauses more efficient.
5. **QueryEvaluator**, which takes in a Query object and evaluates the raw result.
6. **QueryResult**, which contains the table that stores the intermediate results from the evaluations of individual clauses.
7. **QueryResultProjector**, which formats the raw results and returns the result to I/O.

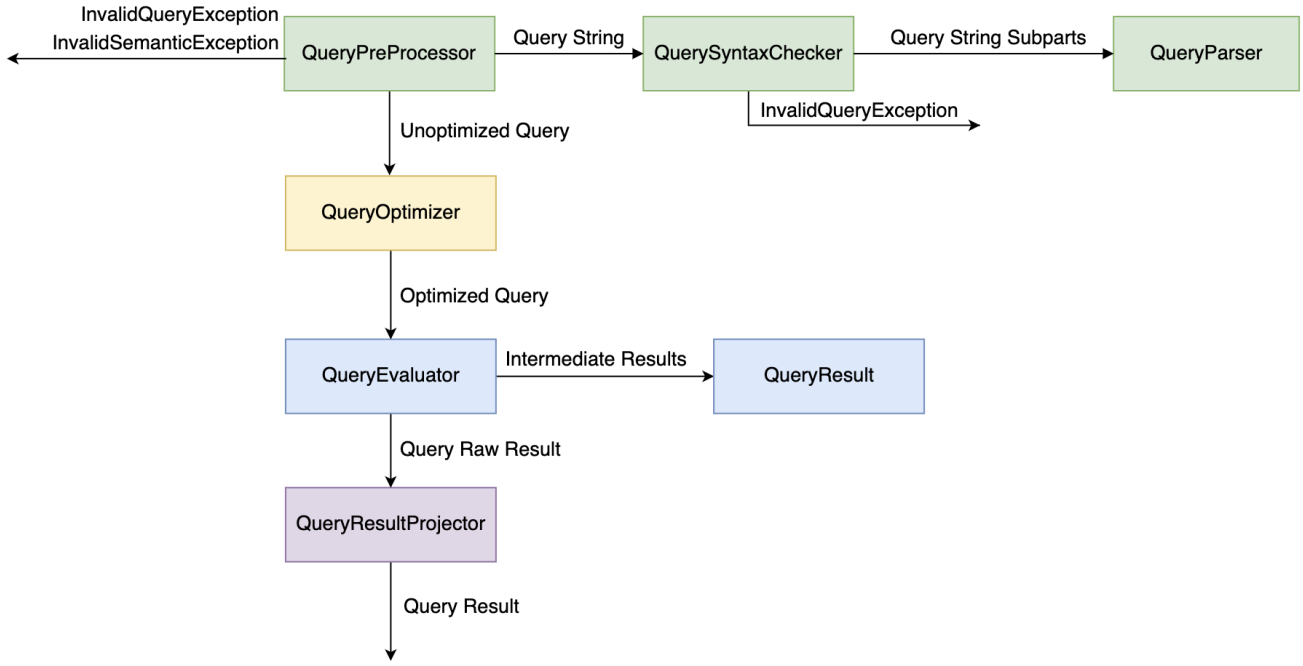


Fig. 3.5.1-1: Production Pipeline for Query Processor

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:

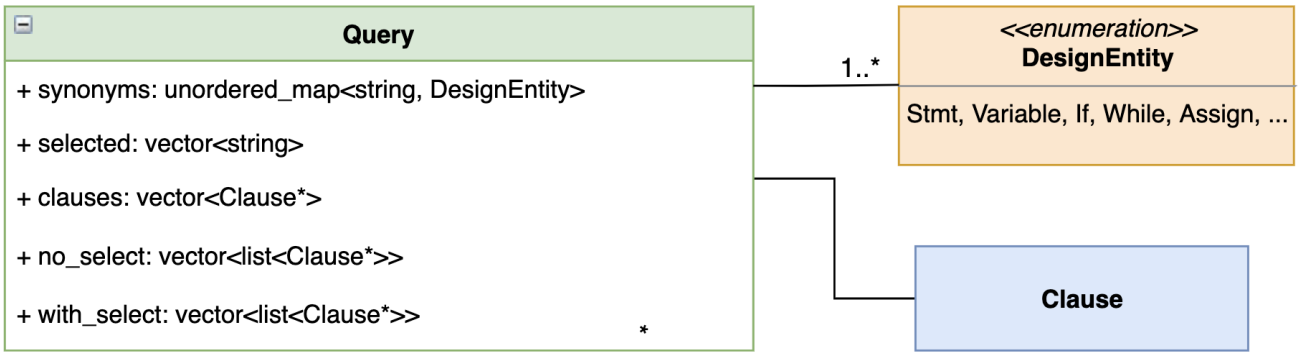


Fig. 3.5.2.2-1: Class diagram for a complete `Query` object

As shown in the class diagram above, a complete `Query` object has five fields:

1. **`synonyms`**: stores mappings of synonyms and their corresponding design entities that are declared in the declaration part of query
2. **`selected`**: 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
4. **`no_select`**: clause groups that does not contain selected synonyms, grouped according to optimization rules (this component is constructed by `QueryOptimizer`)
5. **`with_select`**: clause groups that contains selected synonyms, grouped according to optimization rules (this component is constructed by `QueryOptimizer`)

We will focus on the creation of the first three components in `QueryPreProcessor`. After the `Query` object is half-completed, it will be passed to `QueryOptimizer` which will add the last two components according to the **`clauses`** vector prepared by `QueryPreProcessor`.

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 constructed by `QueryPreProcessor` will have the following structure:

query:Query
synonyms = { { "a" : Assign } , { "w" : While } , { "v" : Variable } } selected = ["a" , "v"] clauses = [UsesClause("w","v") , PatternAssign("a","v","_") , WithClause("a.stmt#", "3")] with_select = {} no_select = {}

Fig. 3.5.2.2-3: `Query` object structure for sample query, constructed by `QueryPreProcessor`

(2) Clause

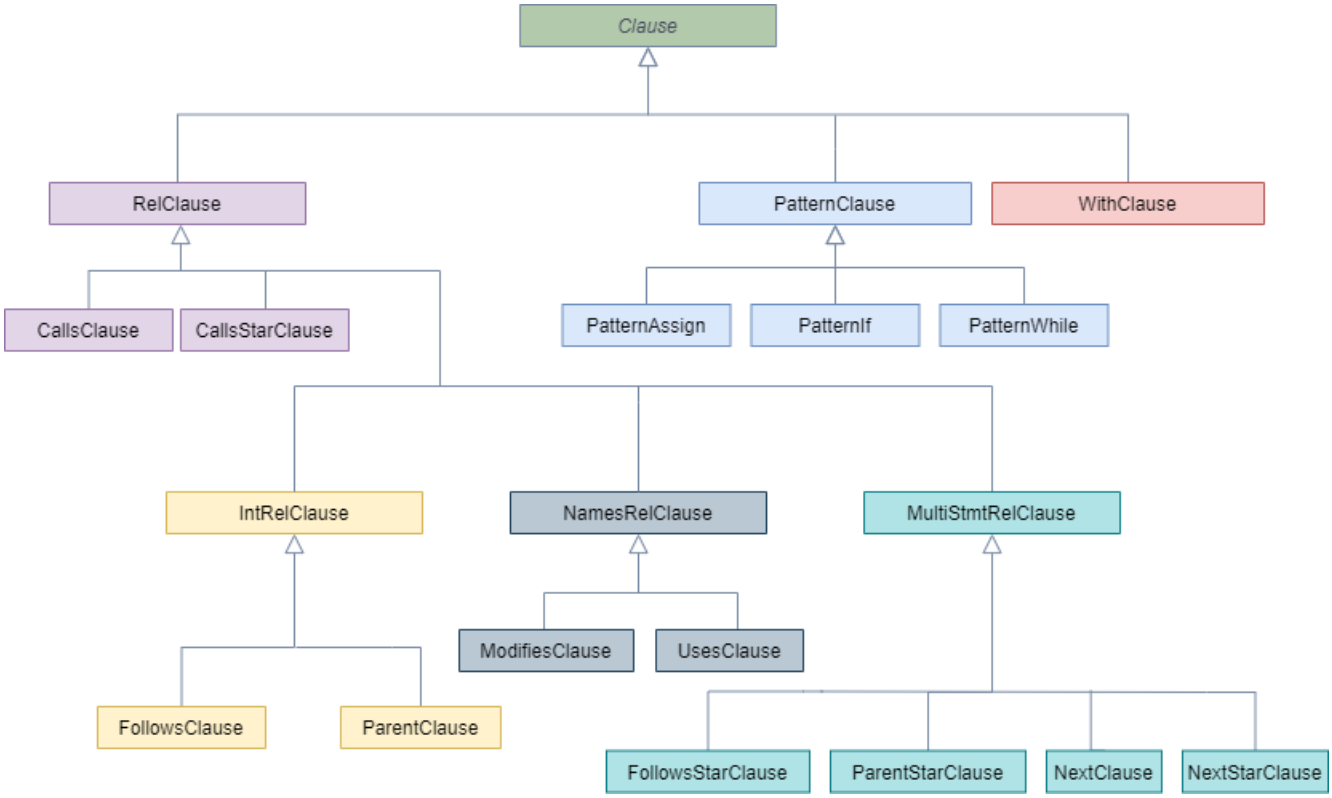


Fig 3.5.2.2-4: Clause Class Diagram

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

Overview

In the process of packaging the `Query` object, `QueryPreprocessor` performs various syntactic and semantic checks on the input query string.

(1) Syntactic Validation

`QueryPreprocessor` calls `QuerySyntaxChecker` to perform the following syntactic checks:

1. The overall query structure: declarations (optional), 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)

Validation Rules

Here is a comprehensive list of syntactic validation rules, checked by respective regexes. 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, no semicolons, etc.	REGEX_QUERY	“stmt s; blablabla such that” “stmt s Select s”	"Invalid query structure."
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(dsaf3erewt)”	"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."

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). 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:

```
assign a; variable v;
Select <a,v> such that Uses(a, “x”) and pattern a(v, _)
```

Fig. 3.5.2.3-2: Sample query

While performing syntactic checks, `QueryPreprocessor` detects ”and” between clause groups of different types (Uses and pattern). An `InvalidQueryException` would be thrown with the following message:

```
InvalidQueryException: Invalid query structure.
```

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) Semantic Validation

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

Next, 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 "_".

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

Invalid Query Type	Checked By	Example	Exception Message
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 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."`

Specifically, for the last type of invalid query, i.e. invalid parameter types, here is a comprehensive list of semantic validation rules implemented in `Clause::validate`:

Clause	First Parameter	Second Parameter
Follows/Follows* Next/Next*	stmt (broad sense) synonym, int, wildcard	stmt (broad sense) synonym, int, wildcard
Parent/Parent*	stmt, while, if, prog_line synonym, int, wildcard	stmt (broad sense) synonym, int, wildcard
Modifies	"IDENT", stmt (except print) synonym, proc synonym, int	variable synonym, "IDENT"
Uses	"IDENT", stmt (except read) synonym, proc synonym, int	variable synonym, "IDENT"
Calls/Calls*	"IDENT", proc synonym	"IDENT", proc synonym
Affects/Affects*	stmt, prog_line, assign synonym, int, wildcard	stmt, prog_line, assign synonym, int, wildcard
PatternAssign	"IDENT", variable synonym	"expressio"n / _"expression"_ / wildcard

PatternWhile	“IDENT”, variable synonym	wildcard
PatternIf	“IDENT”, variable synonym	wildcard
With	valid synonym attribute pair (as specified in wiki), prog_line synonym, Int, “IDENT” *The type must match with the second param	valid synonym attribute pair (as specified in wiki), prog_line synonym, Int, “IDENT” *The type must match with the first param

*stmt (broad sense): stmt, assign, print, read, if, while, call, prog_line

Below is the code snippet of the `validate` function to check whether the parameters of the Affects clause is semantically valid. According to the table above, both Affects parameters can be either integer, or wildcard, or synonyms of assign, stmt, or prog_line. Such `validate` function is written in every `Clause` Class to perform the semantic checking for each type of clause.

```
void AffectsClause::validate(unordered_map<string, DesignEntity> map) {
   vector<string> params = { left, right };
   for (string p : params) {
       bool isAssign = isSynonym(p) && map[p] == DesignEntity::Assign;
       bool isStmt = isSynonym(p) && map[p] == DesignEntity::Stmt;
       bool isProgLine = isSynonym(p) && map[p] == DesignEntity::ProgLine;
       if (!(isAssign || isStmt || isProgLine || isInteger(p) || isWildcard(p))) {
           throw InvalidSemanticException(
                   "Invalid param type for Affects clause.");
       }
   }
}
```

An `InvalidSemanticException` is thrown when the input query string 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 checks, `QueryPreprocessor` detects duplicate synonyms “p” in the declaration. An `InvalidSemanticException` would be thrown with the following message:

```
InvalidSemanticException: Duplicate synonyms in query.
```

Fig. 3.5.2.3-3: Sample Exception

(3) 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

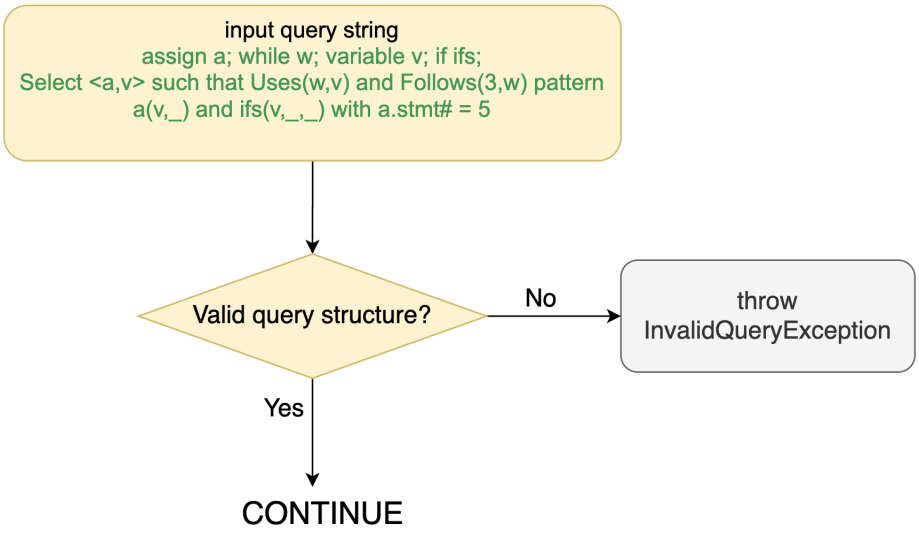


Figure 3.5.2.3-8: Overall query structure validation

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:

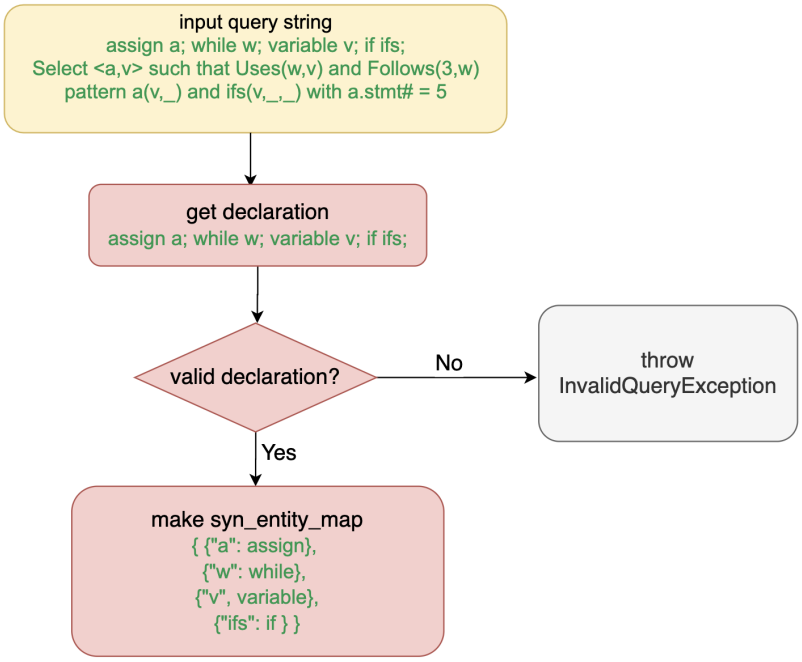


Figure 3.5.2.3-9: Extract declarations and create syn_entity_map

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:

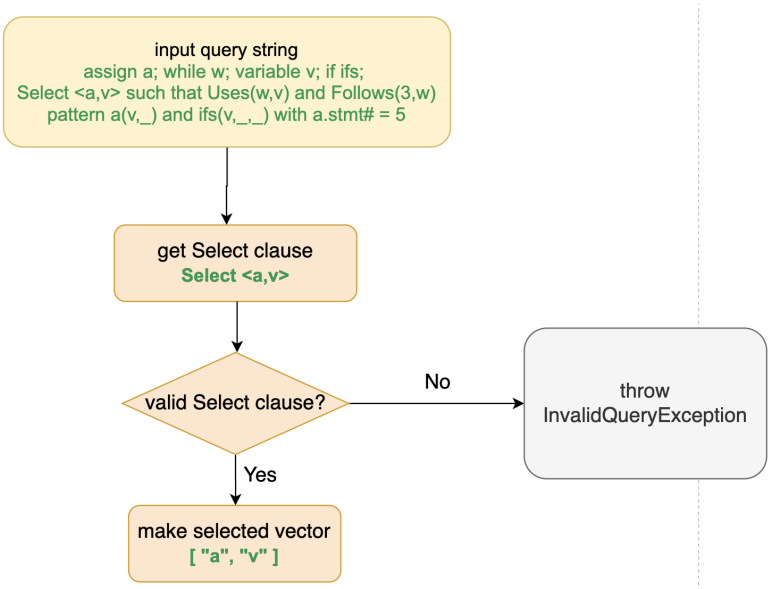


Figure 3.5.2.3-10: Extract selected tuple/BOOLEAN and make `selected` vector

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:
(For simplicity, only the extraction process of relational clauses is fully shown. The processes to extract pattern clauses and with clauses are the same.)

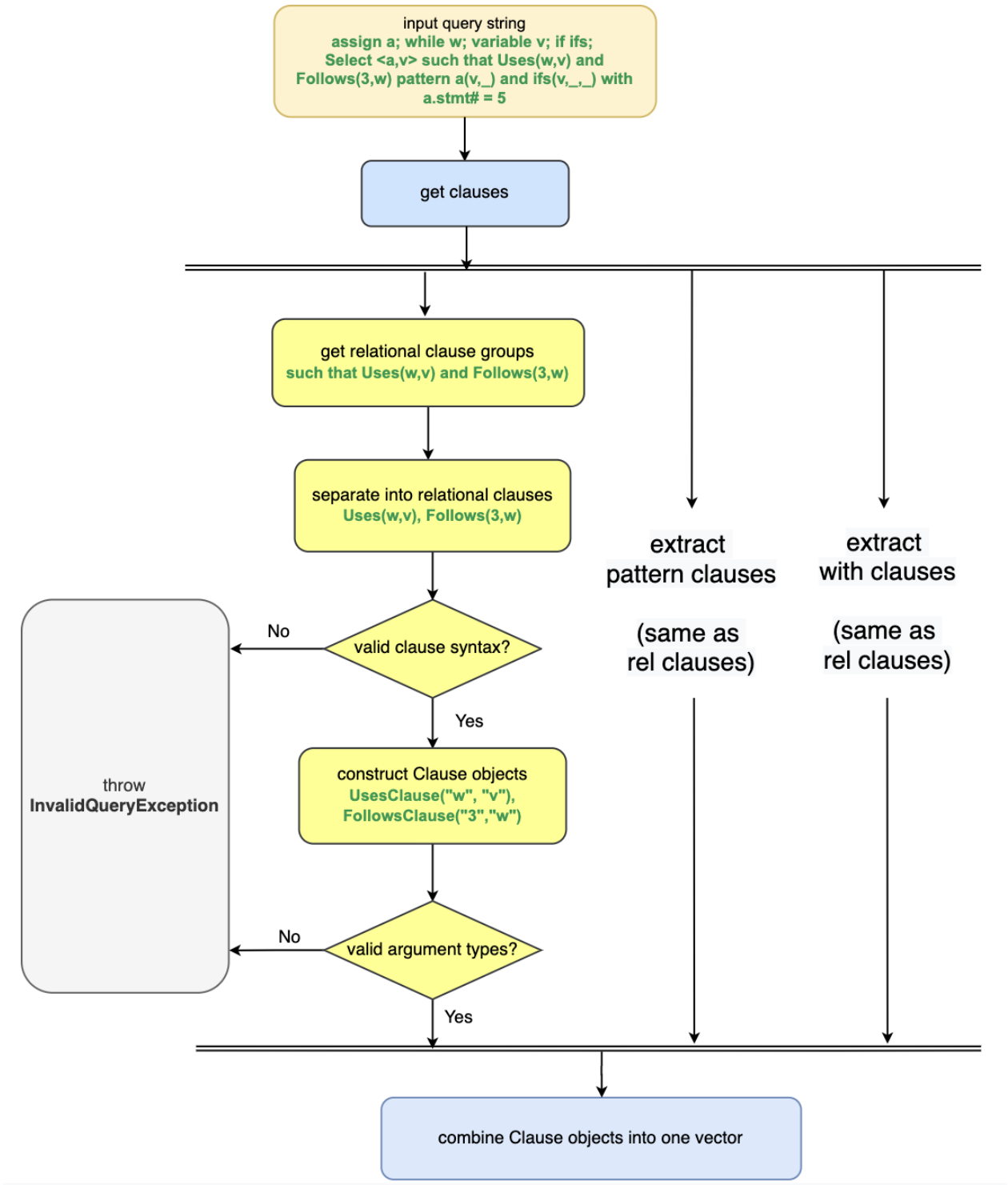


Figure 3.5.2.3-11: Extract clauses and make `clauses` vector

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 (half-completed) `Query` object, which will then be passed to `QueryOptimizer` for further optimization.

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:Query

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

Figure 3.5.2.3-12: `Query` object structure for sample query, created by `QueryPreProcessor`

3.5.2.4. Query Optimization Done in QueryPreProcessor

There are two optimization steps done:

1. **Remove duplicate RelClauses with identical synonyms:**
   When adding a new `RelClause` into the **`clauses`** vector, `QueryPreProcessor` checks if there is an existing `RelClause` in the vector with identical synonyms, and adds the new clause only if there is no existing identical clause.

2. **Remove WithClauses with identical LHS and RHS:**
   When adding a new `WithClause` into the **`clauses`** vector, `QueryPreProcessor` checks whether the LHS and RHS of the `WithClause` are identical, and adds the clause only if the two sides are not the same.

3.5.3. Query Optimizer

3.5.3.1. Overview of Query Optimizer

`QueryOptimizer` is the component in PQL that handles the optimization of clauses. It takes in a `Query` object created by `QueryPreProcessor`, looks into the list of clauses, and tries to find out the best way to arrange them into subgroups and re-order the clauses. The goal is to make the subsequent clause evaluation in `QueryEvaluator` more efficient in terms of both time and space complexity, than the original ordering of clauses.

3.5.3.2. Data Structure for Query Optimizer

`QueryOptimizer` class has the following attributes:

- **`Query* query`:** pointer to the `Query` object constructed by `QueryPreProcessor`
- **`set <string> selected_synonym`:** a set of selected synonyms in the query
- **`list<Clause*> no_syn`:** a list of clauses with no synonyms
- **`list<Clause*> with_one_syn`:** a list of clauses with only 1 synonym
- **`list<Clause*> with_two_syns`:** a list of clauses with 2 synonyms
- **`unordered_map<int, list<Clause*>> groups`:** a utility map that hashes clause groups to an integer index, so that we can link a clause group to its respective set of synonyms (stored in the following map) through the index. Used for dividing subgroups and forming the chain of connected synonyms.
- **`unordered_map<int, set<string>> syn_in_groups`:** a utility map that hashes a set of synonyms to an integer index same as that of its corresponding clause group (stored in the above map). Used for dividing subgroups and forming the chain of connected synonyms.
- **`vector<list<Clause*>> with_select`**: a vector that stores clause groups containing selected synonyms. Will be added into the `Query` object as the **`with_select`** component after the whole optimization process.
- **`vector<list<Clause*>> no_select`**: a vector that stores clause groups that do not contain any selected synonyms. Will be added into the `Query` object as the **`no_select`** component after the whole optimization process.

`QueryOptimizer` also provides an API **`optimizeQuery()`** to be called by `SPAController` after the `Query` object is constructed by `QueryPreProcessor` and before it is passed to `QueryEvaluator` for evaluation. This function will put the optimized clause groups into the **`no_select`** and **`with_select`** components in the `Query` object accordingly.

3.5.3.3. Implementation of Query Optimizer

General Steps

1. **Sort clauses according to clause type**
2. **Divide clauses based on number of synonyms**
3. **Form subgroups with a chain of connected synonyms**
4. **Sort subgroups based on selected synonyms**

Now, we will illustrate each of the above steps in detail with the following sample query (10 clauses):

```
assign a, a1; variable v; while w, w1; procedure p; print pr; read r;
Select <w,w1> such that Affects(a,a1) and Parent(w,a) with w.stmt#=10
such that Follows*(pr, r) and Follows(3,4) and Next(6,7) and Uses(p,
"x") and Calls ("main", p) pattern a(v,_) pattern w1("x", _)
```

Step 1: Sort clauses according to clause type

Rationale:
Certain types of clauses are easy to evaluate and return fewer results on average, for example, with, Follows, Calls, etc. Therefore, we want to evaluate them first so that the intermediate table size can be kept small. For certain types of clause that return more results on average, or the results are not stored in PKB and have to be computed dynamically, for example, Next*, Affects, Affects*, we want to evaluate them as late as possible, and only when necessary. Hence, we will push them to the end such that we may not have to evaluate them at all in the case of early termination.

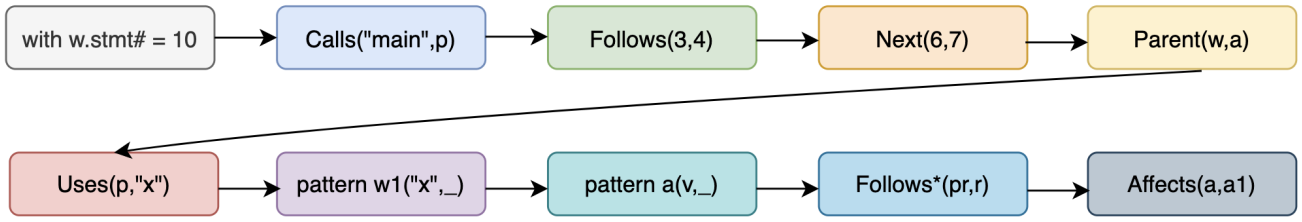
Implementation:
We rank the clauses by their types according to the following table, in order of descending priority:

Type	With	Calls	Follows	Next	Parent	Modifies	Uses	PatternWhile
Rank	1	2	3	4	5	6	7	8

Type	PatternIf	PatternAssign	Calls*	Parent*	Follows*	Next*	Affects	Affects*
Rank	9	10	11	12	13	14	15	16

At this step, we will sort the list of unordered clauses given by the `Query` object according to the rank.

Result:
Now, we have rearranged the original clauses in the following order:



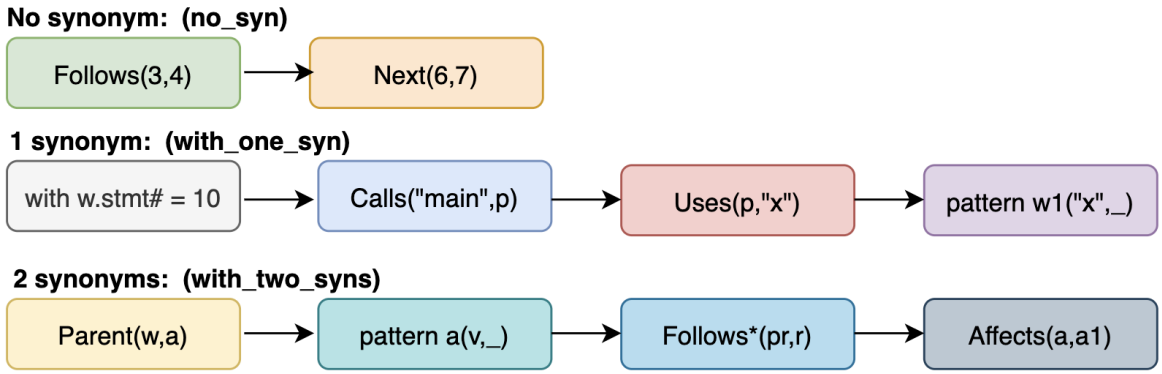
Step 2: Divide clauses based on number of synonyms

Rationale:
Generally, the more synonyms in a clause, the more results will be returned by this clause, and therefore, more results to be added into the intermediate result table. Therefore, we would like to prioritize the

evaluation of clauses with fewer synonyms, especially those without any synonyms. Since it just evaluates to `TRUE` or `FALSE`, it does not increase the size of the intermediate result table. It also has a higher chance of early termination since the condition is more restrictive.

Implementation:
We would use the API provided by the `Clause` class - **`getNumOfSyn()`** to obtain the number of synonyms contained in each clause, and sort the clauses into **`no_syn`**, **`with_one_syn`**, **`with_two_syns`** lists accordingly.

Result:
After this step, we obtain the 3 lists of clauses based on their number of synonyms. Within each list, clauses' relative ranking (from Step 2) is preserved:



Step 3: Form subgroups with a chain of connected synonyms

Rationale:
The cross-product procedure will increase the size of the table exponentially, and therefore, we would like to try our best to avoid it. Grouping the clauses with common synonyms allows us to avoid performing cross-product since whenever a clause is evaluated, either the table is empty, or at least one of the synonyms is already inside the table, and thus we can perform filtering instead.

Another motivation is that if we can ensure there are no common synonyms across groups, we can create separate result tables for each subgroup, as the results from other groups will not have an impact on the evaluation of this group. This allows us to further avoid cross-product across groups.

Implementation:

Clause added	Bridging Synonym	Synonym set
`with w.stmt#=10`	`null (first clause)`	`w`
`Parent(w,a)`	`w`	`w,a`
`pattern a(v,_)`	`a`	`w,a,v`
`Affects(a,a1)`	`a`	`w,a,v,a1`

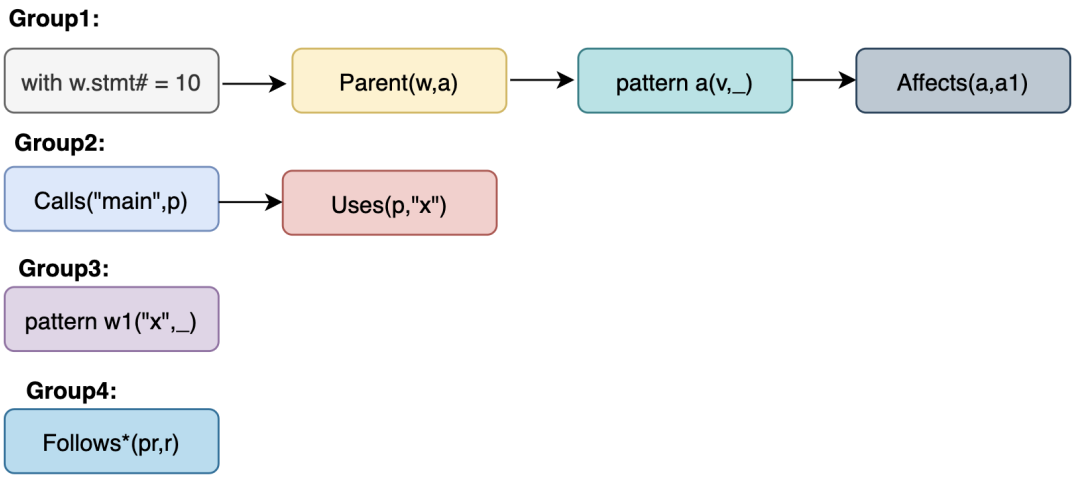
As shown in the chart above, we pick the first clause from **`with_one_syn`**, in this case it will be *with "w.stmt#=10"*. We then look at the synonym it contains, which is "w", and we push it into a set and try to look for other clauses containing "w". The clause we find is *"Parent(w,a)"*. We push it into the clause list, and push both synonyms "w" and "a" into the set. Now, all clauses containing "w" have been found, but since there is another synonym "a" newly added to the set, we need to continue to look for all clauses containing "a". The clauses we find are *"pattern a(v,_)", "Affects(a,a1)"*. We push them into the lists, and push the other synonyms "v" and "a1" into the set. However, there are no other clauses containing "v" or "a1", we will stop here for this subgroup and continue to form the next subgroup.

Clause added	Bridging Synonym	Synonym set
`Calls("main",p)`	`null (first clause)`	`p`
`Uses(p,"x")`	`p`	`p`

The chart above shows another linked clause group, bridged by the common synonym "p".

If **`with_one_syn`** is empty, we will pick the first clause for a subgroup from **`with_two_syns.`** This is because we always prefer a subgroup to start with a clause that contains fewer synonyms since they are more restrictive and likely to return fewer results to keep the table size small.

Result:
After this step, we will store the subgroups of clauses containing synonyms into **`groups`** (map) and the set of synonyms of each subgroup into **`syn_in_group`** (map):



Step 4: Sort subgroups based on select clause

Rationale:
For subgroups that do not contain any synonyms in the Select Clause, their results will not affect the evaluation of the Select Clause. And because they do not have common synonyms with other subgroups, the results will not be used for the evaluation of other subgroups either. In other words, once we finish evaluating subgroups that do not contain any synonyms in the Select Clause, the result can be reduced to a

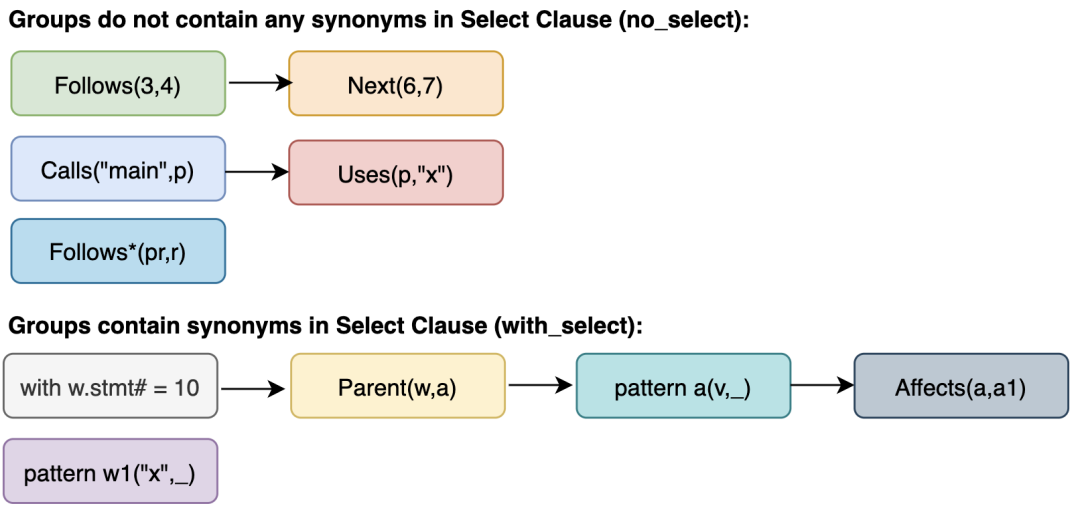
single value - `TRUE` or `FALSE`, just like the clauses without synonyms. Therefore, we would like to prioritize this kind of subgroups for early termination, and to avoid unnecessary increase in the intermediate result table size.

Implementation:

We just need to check whether any of the synonyms in the Select Clause is inside the **`syn_in_group`** set of this subgroup as well. If yes, we push this subgroup into **`with_select`** list. Otherwise, we push this subgroup into **`no_select`** list. **`no_syn`** list will always be pushed into **`no_select`** first.

Result:

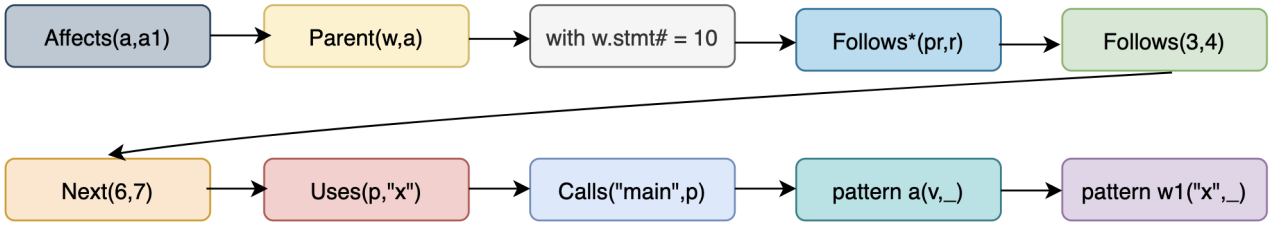
We will have the 2 vectors of subgroups of clauses based on whether the subgroup contains any synonym in the selected clause:



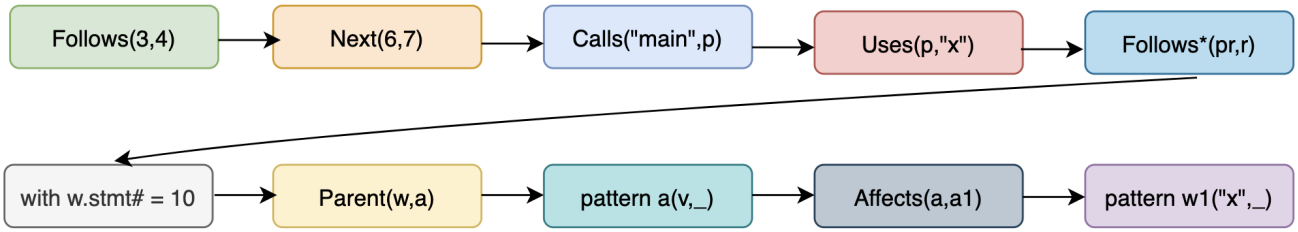
Lastly, we will put **`no_select`** and **`with_select`** into the `Query` object, such that `QueryEvaluator` can access the two lists through the `Query` object.

Conclusion

Original order of the 10 clauses in the sample query:



Optimized order of the 10 clauses in the sample query:



We can see that originally, we need to evaluate Affects(a,a1) clause first, whose results are not pre-stored in `PKB` and is likely to return many results. Therefore, the size of the intermediate result table can increase significantly at the beginning. However, in the optimized order, Affects(a,a1) becomes the second last clause to be evaluated. And since the result for "a" is already inside the table at the time when we evaluate it, we can perform filtering instead of pure cross-product.

Also, since the clauses are not grouped in connected synonyms, we need to store all intermediate results in a single giant table. If we perform cross-product several times, the size of the table can expand exponentially. However, in the optimized order, we have the flexibility to have smaller subtables for each subgroup, since there are no common synonyms across subgroups. In this way, we can keep the overall time size small and avoid unnecessary cross-product processes across different subgroups. We only need to perform the cross-product when we evaluate the Select Clause at the end.

Lastly, the optimized order also has a high chance of early termination since the more restrictive clauses are evaluated first. Thus, we can improve the efficiency by avoiding evaluation of the more complex clauses.

3.5.3.4. Difference from Suggested Implementation

The implementation suggested in lecture notes first divides clauses into subgroups based on linked synonyms, before ranking them according to the clause types. Our implementation, on the other hand, reverses these two steps, i.e. we reorder clauses based on their types, and then sort them into subgroups with linked synonyms. This is to make sure that within each subgroup, every pair of consecutive clauses have at least one common synonym. If we rank clauses within subgroups instead, this will interfere with the order and potentially break the link of synonyms, thus introducing unlinked consecutive clauses which will require cross-product.

For example, given a sample query with these 3 clauses in their original order:

```
Parent(s,pn) -> with a.stmt#=9 -> Affects(s,a)
```

If we sort them into subgroups first, all three will belong to the same subgroup as they have pairwise linked synonyms:

```
Parent(s,pn) -> Affects(s,a) -> with a.stmt#=9
```

If we further rank them according to clause types:

```
with a.stmt#=9 -> Parent(s,pn) -> Affects(s,a)
```

Notice that now the pairwise link of common synonyms is broken, and the first two clauses do not have any synonyms in common. This will lead to cross product between the evaluation of consecutive clauses.

3.5.4. Query Evaluator (QE)

3.5.4.1. 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 intermediate 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.

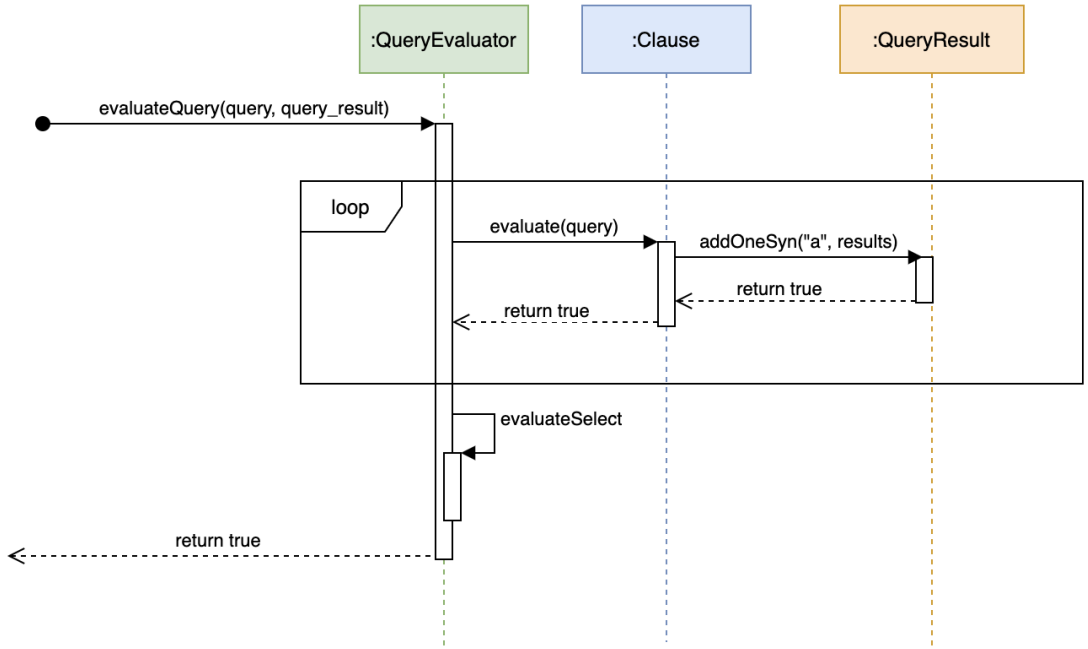


Fig 3.5.3-1: QE Sequence Diagram

3.5.4.2. Data Structures for Query Evaluator

(1) Query Result Class - The Intermediate 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 intermediate 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. 3.5.4.2: 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.
- **`bool filterAndAdd(string existing_syn, string syn_to_add, Vector<pair<string, string>> &results)`**: to filter the rows based on the existing synonym and add the value for the other synonym accordingly.

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*.

3.5.4.3. 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 3, we have added the implementations for Affects and Affects* as part of relational clauses. Most of the relational clauses are similar in terms of structure and principle except for Next*, Affects and Affects*, which are evaluated on-the-fly. Even so, the only main difference is in storage, which does not affect the core of our explanation for evaluation. We will use the Calls clause as an example for how we evaluate most of the relational clauses. This clause accounts for all variations of the Calls clauses, and interacts with the PKB to obtain results regarding Calls relationships. The evaluate function in this class accounts for the all possible combinations that are valid for the Calls clause. This is illustrated in the figure below:

LHS	RHS	Example	PKB APIs called
Name	Name	Calls("One","Two")	isCalls
Name	Wildcard	Calls("One",_)	getCalls
Name	Synonym	Calls("One",p)	getCalls
Wildcard	Name	Calls(_,"Two")	getCalledBy
Wildcard	Synonym	Calls(_,p)	getAllMatchedEntity, getCalledBy
Wildcard	Wildcard	Calls(_,_)	getCallsListSize
Synonym	Name	Calls(cp,9)	getCalledBy
Synonym	Wildcard	Calls(cp,_)	getAllMatchedEntity, getCalls
Synonym	Synonym	Calls(p1,p2)	getAllMatchedEntity, isCalls

Fig. 3.5.4.3-1: Combinations of Parameters in Calls Clause, where p is the synonym for procedures and cp is the synonym for call statements

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 Calls clause when given a query.

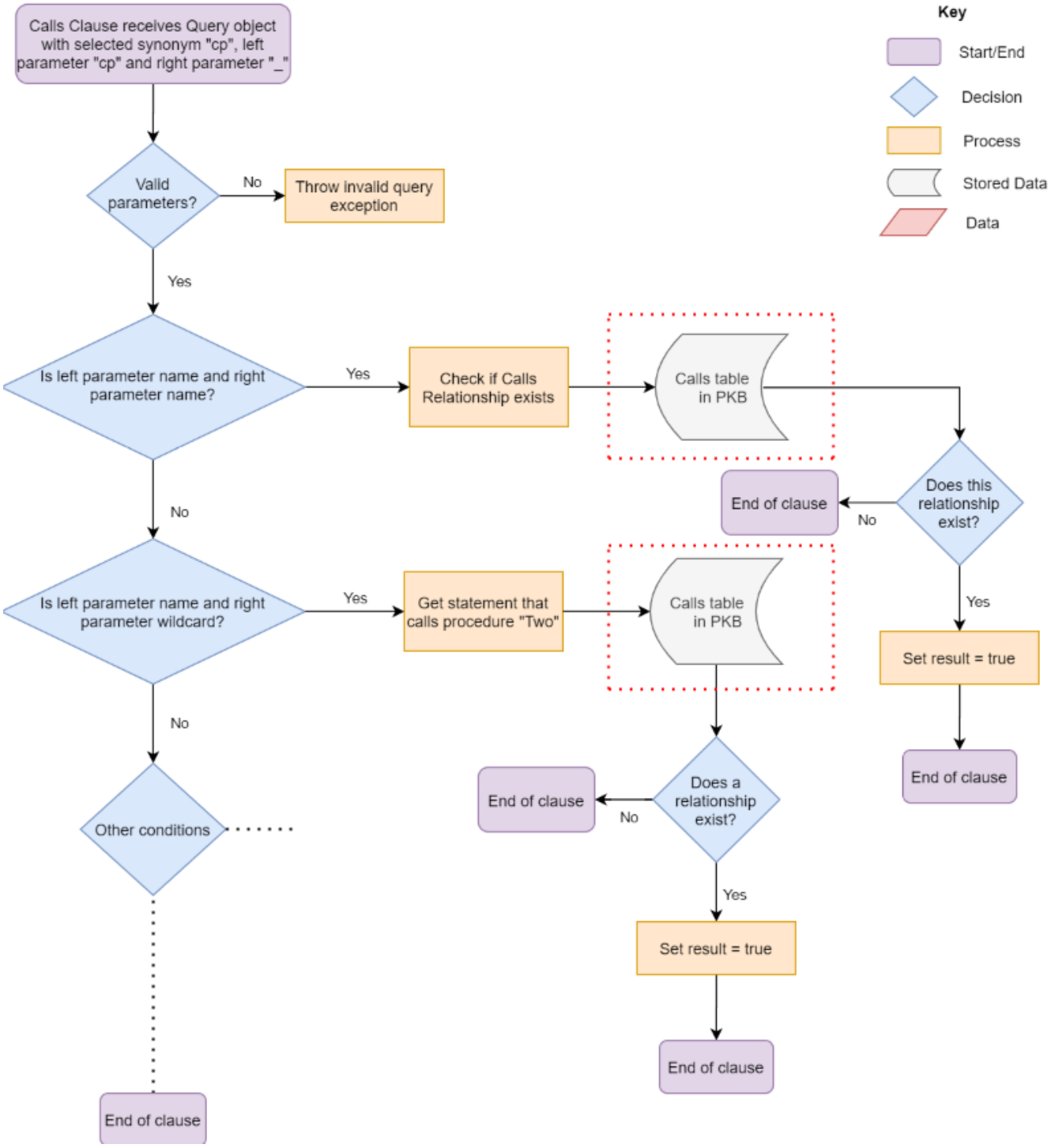


Fig. 3.5.4.3-2: Calls evaluate() Activity Diagram

The only difference Figure XXXX (above) has with clauses that are computed on-the-fly (Next*, Affects, Affects*) is that during the API call to the PKB to obtain the relevant relationships, indicated

by the components of the diagram that are enclosed by the red-dotted boxes, the results are not directly retrieved from a pre-stored table. Instead, the results are computed when the call is made.

Example 1
In this example, we will illustrate the evaluation of the Calls clause with the following query:

"procedure p; assign a;
Select a such that Calls("One",p)"

The left parameter is a procedure name and the right parameter is a synonym, in this case, the synonym refers to call statements. Therefore, we need to evaluate whether there is (1) a procedure with name "One", and (2) the procedure "One" has a statement that makes a call to another procedure.

First, a call, `getCalls("One")`, is made to the PKB. This call checks if there is any Calls relationship between "One" and at least one other procedure. If there exists values (procedures) in the Calls table, those values will be returned as a list. If this value does not exist, the function will return an empty list. In this case, assume that the value "Two" is returned, indicating that procedure "One" calls procedure "Two". The clause is therefore satisfied.

Example 2
In this example, we will illustrate the evaluation of the Calls clause with the following query:

"procedure p;
Select p such that Calls(_,p)"

The left parameter is a wildcard and the right parameter is a synonym, in this case, the synonym refers to procedures. Therefore, we need to evaluate all the procedures that are called by other procedures.

First, a call, `getAllMatchedEntity(procedure type)`, is made to the PKB. This call retrieves the names of all the procedures in the source code. We then need to check for each of these procedures, whether they are called by at least one other procedure, thus the call `getCalledBy(procedure name)`. If there exists values (procedures) in the Calls table that calls that procedure, those values will be returned as a list. If this value does not exist, the function will return an empty list. All the procedures that return a list of size greater 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.

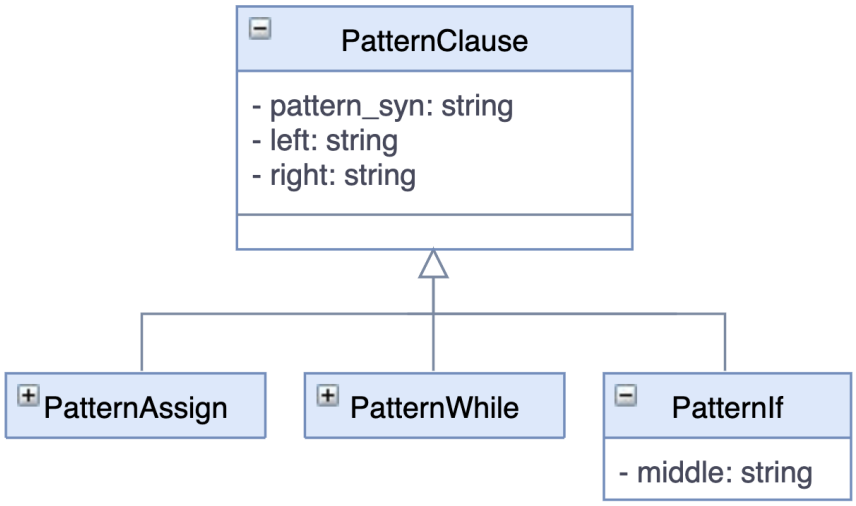


Fig. 3.5.4.3-3: PatternClause class diagram

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 below 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. 3.5.4.3-4: 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. 3.5.4.3-5: Valid attribute in With Clause

Notice that certain attributes are mapped to multiple Attribute types based on the synonyms it can be attached to. For instance, the procName attribute has been split into Attribute types ProcProcName and CallProcName which represent the cases “procedure.procName”, and “call.procName” respectively.

Many Attribute pairings share the same evaluation logic but only differ in PKB API calls. These categories of evaluation logic have been abstracted into internal methods that take in these API call methods as arguments 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 a `WithClause` evaluation case.

Category	Attribute Pairing	PKB API Calls to Pass into Method
Value type NAME: Statement Synonym attribute and non-statement Synonym attribute	VarVarName, PrintVarName ProcProcName, PrintVarName	`getAllStatementsThatPrint( VARIABLE_NAME v)`

		`getVariablePrintedBy(STMT_NO s)`
	ProcProcName, CallProcName VarVarName, CallProcName	`getAllStatementsThatCall(VARIABLE_NAME v)` `getProcedureCalledBy(STMT_NO s)`
	VarVarName, ReadVarName ProcProcName, ReadVarName	`getAllStatementsThatRead(VARIABLE_NAME v)` `getVariableReadBy(STMT_NO s)`

Fig. 3.5.4.3-6: Subset of WithClause evaluation categories

Consider the 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.

If the above conditions are fulfilled, the Query Evaluator will call the internal method handling the catagory’s logic, and pass in the appropriate PKB API methods depending on the Attribute types.

(4) Merging results in Query Result table

Remember that Query Result is used to maintain a table of all intermediate results and provides 5 operations for edit the table contents - namely **`"addOneSyn()"`**, **`"addTwoSyn()"`**, **`"filterOneSyn()"`**, **`"filterTwoSyn()"`**, **`"filterAndAdd()"`**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)"*

To just illustrate the idea, we will assume that optimization is not performed and the query will not be reordered here. We will evaluate the clause from left to right.

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`.** The sequence diagram shows how the API is called.

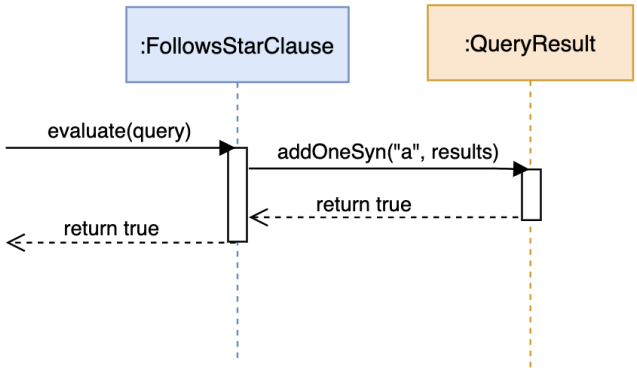


Fig. 3.5.4.3-7: Sequence Diagram of adding a column of values

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

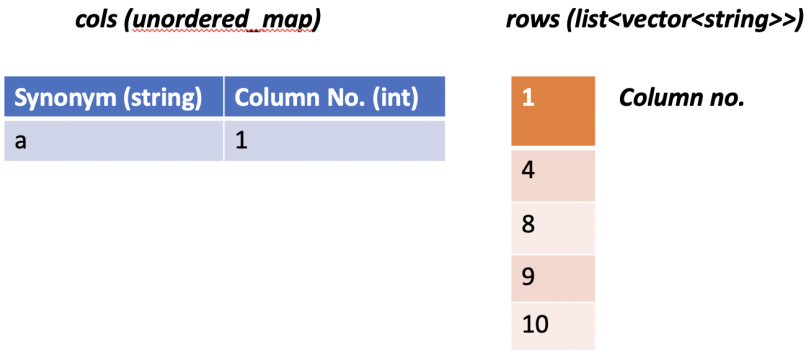


Fig. 3.5.4.3-8: Query Result content after evaluating Follows (3,a)*

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`**. 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):

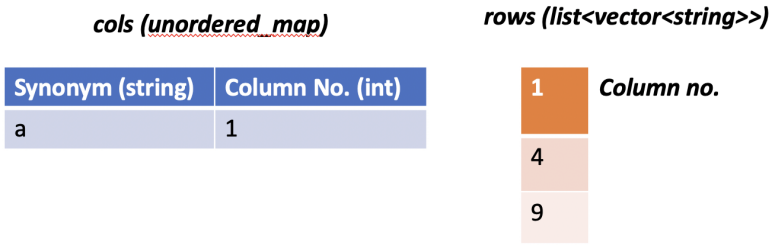


Fig. 3.5.4.3-9: Query Result content after evaluating Modifies(a,"x")

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`**. 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. 3.5.4.3-10: 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.`** 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. 3.5.4.3-11: 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 will call **`filterAndAdd`**. In this API, it will perform filtering on "w". After that, the new synonym, "s", will be added to the table for each remaining "w" value. 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. 3.5.4.3-12: Query Result content after evaluating Uses(w,v)

Creating Sub-table for Each Subgroup of Clauses

Since we divide the clauses into several subgroups in Query Optimizer, there will be no common synonyms across these subgroups. This allows us to create separate tables for each subgroup, since the results for the synonym in a subgroup will not affect the results of another subgroup. We can also clear the table content for subgroups which do not contain any synonym in the Select Clause, since the results of these subgroups will not be referred later. Hence, we are able to keep several small tables instead of a giant table, so that unnecessary cross-product can be avoided. Cross-product will be performed when we are evaluating the Select Clause. We will retrieve particular columns for the synonyms to be selected from each table, remove duplicate values, and perform cross product across different tables.

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.

3.5.5. 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.

3.5.5.1. 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.

3.5.5.2. 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 (since the synonyms are connected, we will a table for this query):

"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. 3.5.5.1: 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.

3.5.5.3. 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.

3.5.5.4. 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.

Additional notes
In the previous section, we mentioned that we have separate tables for different subgroups. The evaluation of the Select Clause follows the same logic above, with one more step of retrieving the selected columns from each table and performing cross-product across columns from different tables.

3.5.6. Design Considerations

3.5.6.1. 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.

3.5.6.2. Design Considerations for Query Evaluator

(1) 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.

(2) 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 several sub tables for each group of clauses

Query Result class maintains a list of tables to store intermediate results. Each table stores the result for a particular subgroup of clauses. For groups that do not contain any synonyms in the Select Clause, the table can be cleared after evaluation. For groups that contain synonyms in the Select Clause, the table will be kept. When evaluating Select clause, we retrieve columns of results according to the synonyms. If the synonyms are from different tables, we will perform cross-product on each row.

Pros: Results that will not be used later can be ignored, such that unnecessary cross-product can be avoided. The overall table size is smaller, resulting in better space complexity. Cross-product is pushed to the end, resulting in better time complexity.

Cons: Retrieving results might be the slowest step since a lot of cross-product might be done here. Refactoring needs to be done since the implementation of Query Result class now is different from iteration 2 version. But only Query Result and Query Evaluator classes need to be changed, and the individual clause classes will not be affected.

Alternative 1: Maintain a single table to keep the result (Iteration 2 implementation)

Query Result class maintains a table of intermediate results. At the end of evaluation of a clause, it adds results of new synonym(s) or filters the existing results in this table by calling APIs provided by the Query Result class.

Pros: Implementation is simpler. Faster when retrieving results for Select Clause since we only need to retrieve the relevant columns according to the synonyms selected and remove the duplicate values.

Cons: The size of the table can become large when we have many synonyms. Unnecessary cross product might be performed when adding new synonyms into the table.

Justification for the chosen decision

As the number of synonyms and clauses can be large in iteration 3, a single table might be too large when there are a lot of synonyms. Another important reason is that we have implemented Query Optimizer in iteration 3 that helps us to group the clauses based on connected synonyms, we can ensure that the results of a subgroup will not affect the result from another subgroup. Therefore, we can implement the sub-tables to avoid unnecessary cross product procedures.

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

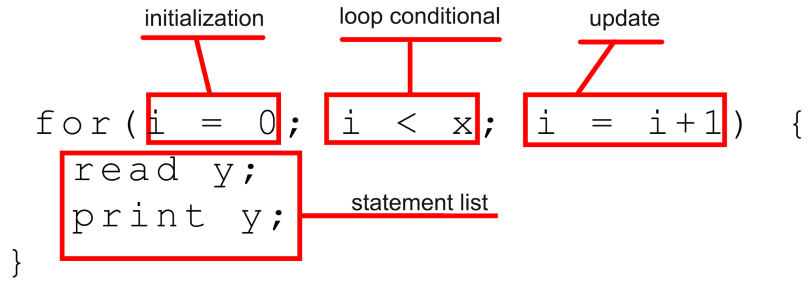


Fig. 4.1.1-1: For Loop Definition

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

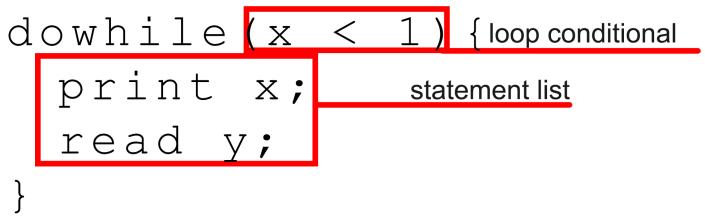


Fig. 4.1.1-2: Do-While Loop Definition

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: 'dowhile' '(' cond_expr ')' '{' stmtLst '}'**

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.

Example

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. Extending the AST Class

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

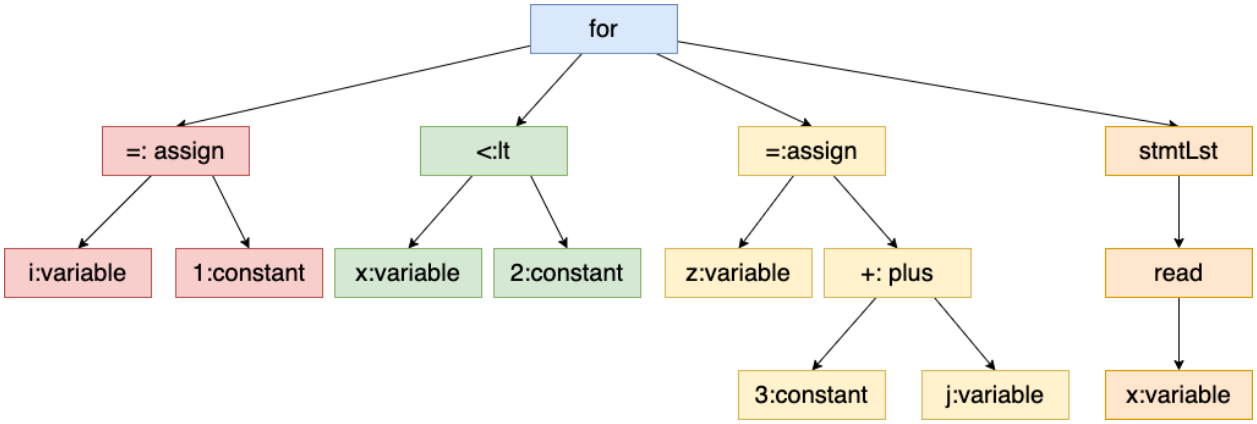


Fig. 4.1.2-5: Resulting AST subtree

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

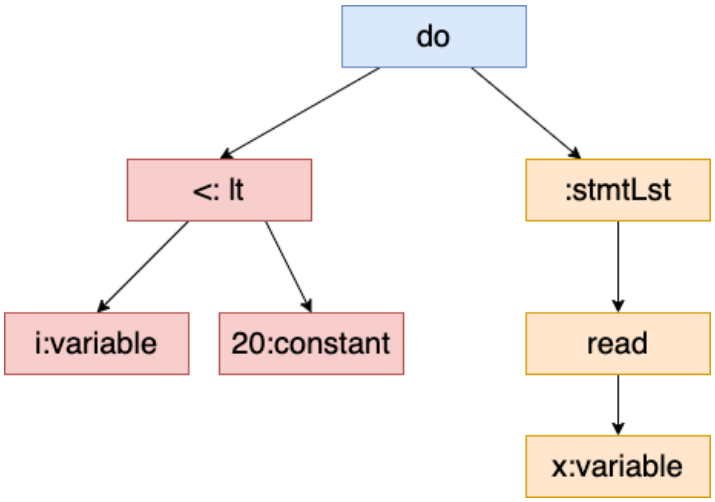


Fig. 4.1.2.2-3: Resulting AST subtree

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' | **'for'** | **'do'** pattern : assign | while | if | **for** | **do** **for : syn-for '(' entRef ',' '_' ')'** //syn-for must be of type 'for' **do : syn-do '(' entRef ',' '_' ')'** //syn-do must be of type 'do'

Fig. 4.1.2.3-1: Extension of PQL

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 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

Do-Loop Relationships

The relationships rules for Do-While Loop are identical to the 'While' relationships. It should be noted when calculating Next/Next* and Affects/Affect* that the statements in the do-while statement list occur before the do-while statement in the CFG.

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

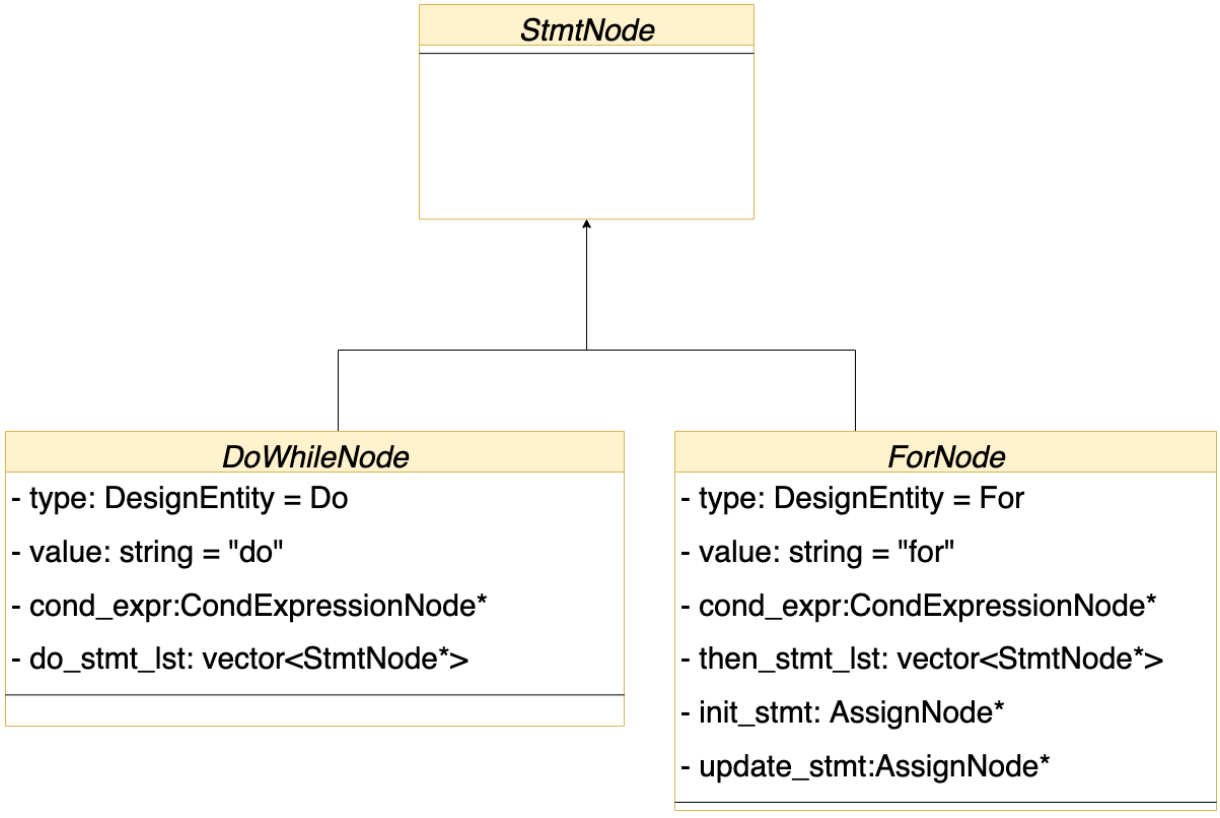
(1) Update AST Data Structure and API

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 has been extended to ensure the AST data structure can properly represent the new AST extensions:



- **ForNode:** Represents the For node of the AST. Stores two `AssignNodes` representing the Initialization and Update statement, a list of `StmtNodes`, and a `CondExpressionNode` representing the loop conditional. Mapped to the 'For' Design Entity
- **DoNode:** Represents the Do node for the AST. 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 has been 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.

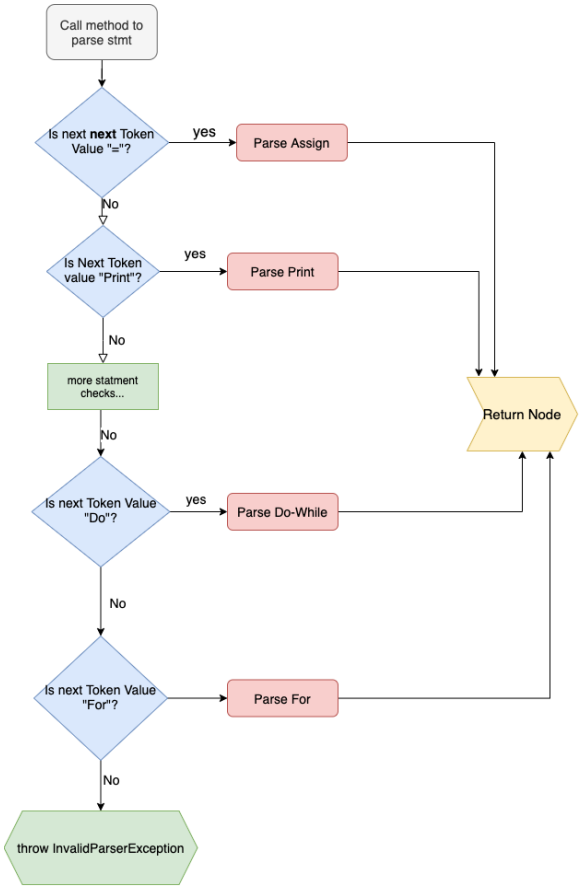


Fig. 4.1.3.1-4: Updated Statement Parsing Flow Chart

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.

For Loop

While the For loop has the same CFG structure as a While loop, some changes have to be made. There are three statements in the declaration of a For loop - the initialisation statement of the control variable, the condition statement and the modification of the control variable. Of these statements, two of which are assign statements - initialisation and modification. Therefore, these two statements are added to the list of assign statements.

The challenge here is that our original implementation was such that every statement in the source program only has one statement, but in this case, there are three - the for loop and the two assign statements. We thus had to change our implementation to prevent the increment of statement numbers after each addition, so that we can add these two assign statements with the same statement number as the For loop statement.

Example: For Loop

```
for(a = 1; a < 5; a = a + 1) { //1
    // stmtLst
}
```

Fig. 4.1.3.2-1: *Example of For loop declaration*

Using Fig. 4.1.3.2-1 as an example, we add statement 1 to the list of for statements. `'a = 1'` and `'a = a + 1'` are assign statements, and they are added to the list of assign statements with statement numbers of 1 as well.

Do-While Loop

The implementation of Do-While loop was more challenging as it goes against how statements are normally added and related. Our final implementation of Do-While has an underlying structure of a While loop, with various differences. Firstly, the statement prior to the Do-While loop has the first statement of the Do-While loop as the next statement. This is because Do-While loops compute the contents of the loop before checking the condition. Secondly, we have to remove the next relationship between the statement prior to the Do-While loop and the Do-While loop statement, as it had been previously populated with the loop being based on a While loop.

Example: Do-While Loop

```
b = 12; //1
dowhile(a < 5) { //2
    c = 7; //3
}
```

Fig. 4.1.3.2-2: Example of Do-While loop declaration

Using Fig. 4.1.3.2-2 as an example, we first implement it as a While loop. This will result in statement 1 having 2 as its next statement, statement 2 having statement 1 as its previous statement, statement 2 having statement 3 as its next statement, and statement 3 having statement 2 as its previous statement. We now make the appropriate changes to convert this to a Do-While implementation. Firstly, we add a Next relationship between statements 1 and 3, as a Do-While loop entails that statement 3 is the next statement of statement 1. As per our usual implementation of inverse relationships, we also add statement 1 as the previous statement of statement 3. Secondly, we remove the Next relationship between statements 1 and 2 from the list of next statements, as well as their relationship from the list of previous statements. Lastly, the next relationship between statements 2 and 3 remains as the program can proceed from statement 2 to 3 if the condition is satisfied.

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 has been 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 PatternFor, that inherit from PatternClause.

4.1.4. Testing

Testing of this extension is elaborated in Section 6.4.15 Test Category 15: For Loop and Do-While Loop Extension.

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.

4.2.2. Extension Specification for NextBip/NextBip*

4.2.2.1. 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.

4.2.2.2. 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 NextBip/NextBip*

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.

NextBip
We accomplished the extraction by taking care of the situation of call statements. Whenever we hit a procedure call, we take the call statement number and point it to the first statement of the called procedure. We then navigate down to the end of the procedure, and point the last statement of the called procedure to the next statement of the caller procedure, after the call statement. Concurrently, we ensure that there is no next relationship between the call statement and the next statement in the same procedure.

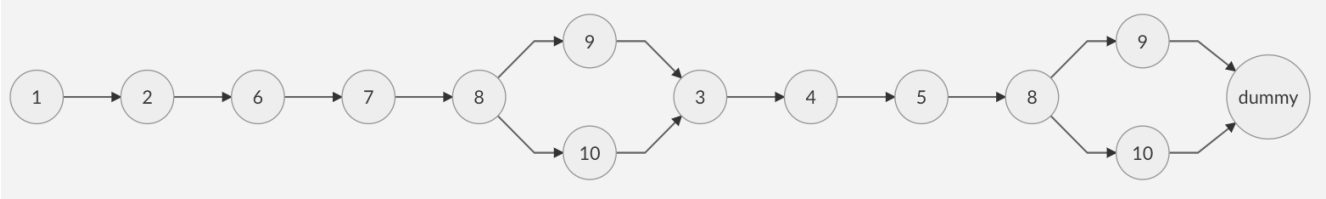


*Fig. 4.2.3.1-1: CFG of NextBip/NextBip**

Example: NextBip

Fig. 4.2.4.1-1 is the NextBip CFG of Fig 4.2.1. Procedure main calls Mary in statement 2, which points the next statement to the first statement of Mary, statement 6. The last statement of Mary happens to be a procedure call to John, thus statement 7 points to statement 8. The last statements of John, 9 and 10, point back to the statement next of statement 7, which happens to be the statement after the call for Mary, which is statement 3.

NextBip*

The implementation of NextBip*, however, is significantly more complicated. This is because we have to capture the cascading characteristic of the relationship, which has to be done with care. Not handling it properly may result in a called procedure not knowing which caller procedure to return to, which causes the system to fail. We dealt with this by implementing a stack of procedure call statements.

Example: NextBip*

Using Fig. 4.2.3.1-1 as an example, traversing down the main procedure, we encounter statement 2 which is a procedure call, so we push 2 into the stack. We then proceed into Mary and traverse it until another procedure call is encountered, statement 7. We then push statement 7 into the stack as well and proceed into John. After traversing through John, we find that no procedure calls were encountered, and we want to return to the caller procedure to continue traversal. We then pop the top element of the stack, which is 7, and return to the statement after statement 7 in the caller procedure, Mary. This happens to be the end of Mary, so we pop the top element of the stack again, which is 2, and return to the statement after statement 2 in the caller procedure, main.

Seeing that the number of NextBip* relationships stored can be potentially very large, we implemented caching to optimise the evaluation of NextBip*. This significantly reduces computation time as it ensures that every Next relationship is only computed once. An implementation detail as a result of the cache is to sort the next statements in inverse order, from largest to smallest, in order to maximise usage of the cache. This is to maximise the number of next relationships stored in the cache, so that the number of computations is minimised. The accessing of the cache is locked behind 2 logical checks, the first being that the cache has to exist and the second being that the stack has to be empty. The stack is required to be empty because the control flow of the cached relationship is based on the empty stack evaluation, if there are values in the stack the control flow would be different from the cached values and would lead to an incorrect result. The alternative would be to cache all possible evaluations from all possible stacks, however that would require too much information to be stored, more information than if we were to precalculate all the NextBip* relationships, and hence was not used as part of the solution.

4.2.3.2. PQL

(1) QPP

Regexes for Query Syntax Validation

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 `NextStarBIPClause`. 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 stmt = isSynonym(p) && isStmt(map[p]);
        if (!(stmt || isInteger(p) || isWildcard(p))) {
            throw InvalidSemanticException("Invalid param type for
NextBip clause.");
        }
```

```
        if (isSynonym(left) && isSynonym(right) && left == right) {
            throw InvalidSemanticException("Two parameters of NextBip
clause are the same synonym.");
        }
    }
    //valid left & right argument types: stmt, 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) Query Evaluator

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

4.2.4. Testing

Testing of this extension is elaborated in Section 6.4.13 Test Category 13: NextBip/NextBip* Extension

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.

4.3.2. Extension Specification for AffectsBip/AffectsBip*

4.3.2.1. 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*.

4.3.2.2. 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** subsituting the *Next*(statement) calls with a *NextBip*(statement) call. This however introduces a limitation to the relationship which is that if a call statement is the last statement of the procedure the search algorithm is unable to find the statement following it.

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 `AffectsStarBIPClause`. 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 isAssign = isSynonym(p) && map[p] == DesignEntity::Assign;
        bool isStmt = isSynonym(p) && map[p] == DesignEntity::Stmt;
        bool isProgLine = isSynonym(p) && map[p] ==
DesignEntity::ProgLine;
        if (!(isAssign || isStmt || isProgLine || isInteger(p) ||
isWildcard(p))) {
            throw InvalidSemanticException("Invalid param type for
AffectsBip clause.");
        }
    }
    //valid left & right argument types: stmt, prog_line, assign, 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) Query Evaluator

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

4.3.4. Testing

Testing of this extension is elaborated in Section 6.4.14 Test Category 14: AffectsBip/AffectsBip* Extension

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[2] 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"
Enum	Initialized with upper camel casing: ("EnumName")

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

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 APIs

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.

Example

LIST_OF_STMT_NO getAffects(**STMT_NO** s)
Description: Returns all the statements that s Affects

Fig. 5.3: Example of Abstract API

For example, the getAffects method shown in Fig. 5.3 takes in an abstract argument **STMT_NO**, which consists of **NO**. When converted to concrete API, this translates to a statement with type integer. getAffects has an abstract return value **LIST_OF_STMT_NO**, which consists of **LIST** and **NO**. When converted to concrete API, this translates to a vector of integers. Therefore, in concrete API, getAffects takes in a statement of integer type and returns a vector of integer of statements of integer type.

In our experience, abstract types help to simplify the planning process significantly, and allows us to foresee potential shortcomings with respect to variable types, prompting us to change certain variable

types in advance. This ultimately saves us a lot of unnecessary work done later in the project when the ripple effect would be significantly greater.

6. Testing

6.1. Test Plan

6.1.1. Test Plan Activity Breakdown

Week	Testing Activities	Components
10	Unit Testing	- Extension additions for Tokenizer, Parser, and AST classes: Do-While and For loop parsing - PKB Design Extractor + Extracting relationship of Affect, Affects*
	Integration Testing	- Front end parser and PKB - PKB and PQL
	Regression Testing	- Run all unit testing and integration test - Run all system test from iteration 1 and 2
	System Testing	- Simple Affects, Affects* source and query test - Simple multi-clauses source and query test
	Bug Fixes	
11	Unit Testing	- Extensions additions for PKB: NextBip/NextBip*/For Pattern Matching/Do Pattern Matching -Extension Additions for QP: For Pattern and Do Pattern Clause/NextBip and NextBip* Clause
	Integration Testing	- QP/PKB interaction: For Pattern and Do Pattern Evaluation/NextBip and NextBip* Evaluation
	System Testing	- NextBip/NextBip* Source and query test - For and Do while loop source and query test
	Stress Testing	- Stress test Affects, Affects* source and query - Stress test multiclauses
	Bug Fixes	
	Regression Testing	- Run all system test from iteration 1 and 2 - Run existing iteration 3 system test
12	Unit Testing	- Extensions additions for PKB: AffectsBip/AffectsBip* -Extension Additions for QP: AffectsBip and AffectsBip* Clause
	Integration Testing	- QP/PKB interaction: AffectsBip and AffectsBip* Evaluation
	System Testing	- AffectsBip/AffectsBip* source and Query test

	Stress Testing	- Stress test multiclauses
	Regression Testing	- Run all unit and integration testing - Run all system test from iteration 1,2 and 3

Fig. 6.1: An overview of Test plan

6.1.2. Test Plan Task Allocation

	Iteration 3		
	Week 10	**Week 11**	**Week 12**
Jefferson Sie	Update Integration test - all the relational clauses - PKB/PQL Integration Testing for Calls/Calls* clauses	Create Stress test for Affects, Affects* Create source and query for extension - Affect, Affects* bip (extension)	Create source and query for extension tests - Affect, Affects* bip (extension)
Rachel Tan	Create system test - Simple Affects, Affects* - Simple multi-clauses	Create source and query for system test - Multi clauses test categories (Test Category 8: Multi Clauses)	Create stress test - multi clauses Test Category 9: Stress Testing
Jaime Chow	Update unit tests affected by extensions: - Tokenizer - Parser - AST classes Update integration test - PKB/Front-End Parser Integration Testing for Next/Next* clauses	Create source and query for extension test - do while loop - for loop	Create source and query for extension tests: - nextbip/nextbip* - Affect, Affects* bip
Chen Su	Update Unit test - Query evaluator - Query optimizer Update Integration test - Pattern clause		
Chen Anqi	Update Unit test - Query optimizer - Query preprocessor, - Query parser - Query syntax checker		
Adam Tan	Update Unit test - PKB algorithm to handle		

	/Affect/Affects* clauses - query result Update integration test - PKB/Front-End Parser Integration Testing and PKB/PQL for Affects/Affects* clauses		
All	Regression test & Bug fixes		

6.1.3. 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*. In week 10, *Affects, Affects** design abstractions were added. More Unit testing for Parser, AST API, Design Extractor, PKB Storage, QP, QPP, QE and QRP was done from week 10 onwards 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 (Affects/Affects*) were completed in week 10, 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*, Affects, Affects*, 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 3, in week 11 and 12, 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 ensure our system can handle multiple *such that*, *with* and *pattern* clauses, with any number of *and* operators. In the final 2 weeks, we also ensure our system can handle *Affects,Affects**. Once the correctness of our system to handle for all clauses were well tested, we proceeded with stress testing (Test Category 9: Stress Testing).

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.

Running the System Tests:
1. In the same terminal
Navigate to Team21/Tests21 to navigate to all the system tests with the following command:
- cd <PATH_TO_PROJECT_DIRECTORY>/Team21/Tests21

Navigate to Team21/Tests21 to navigate to the extensions system tests with the following command:
- cd <PATH_TO_PROJECT_DIRECTORY>/Team21/Tests21/test_extensions

2. Run the respective bash script file for system test on iteration 1, 2 3, or extension by executing the following command on terminal

To run all system test for iteration 1,2,3 in one:
- cd <PATH_TO_PROJECT_DIRECTORY>/Team21/Tests21
- bash run_all_tests.sh

To run iteration 1 system test only:
- cd <PATH_TO_PROJECT_DIRECTORY>/Team21/Tests21/Iteration_1
- bash run_all_iter1_test.sh

To run iteration 2 system test only:
- cd <PATH_TO_PROJECT_DIRECTORY>/Team21/Tests21/Iteration_2
- bash run_all_iter2_test.sh

To run iteration 3 system test only:

- cd <PATH_TO_PROJECT_DIRECTORY>/Team21/Tests21/Iteration_3
- bash run_all_iter3_test.sh

To run iteration 3 extension only:

- bash run_all_iter3_ex_tests.sh

3. Check the query result xml files with the following:

To check the result xml files for Iteration 1:

- cd <PATH_TO_PROJECT_DIRECTORY>/Team21/Tests21/Iteration_1/out
- The result xml files produced are:
    1.out_no_clause.xml
    2.out_Follows_Parent.xml
    3.out_pattern_ModifiesS_UsesS.xml
    4.out_pattern_suchthat_clauses.xml
    5.out_complex_condexpr_nested.xml
    6.out_uncommon_invalid.xml
    7.out_bonus_3_features.xml

To check the result xml files for Iteration 2:

- cd <PATH_TO_PROJECT_DIRECTORY>/Team21/Tests21/Iteration_2/out/valid
- The result xml files produced are:
    1.out_valid_Select_clause.xml
    2.out_valid_With_clause.xml
    3.out_simple_Calls.xml
    4a.out_valid_Calls.xml
    4b.out_valid_Next.xml
    4c.out_valid_ModifiesP_UsesP.xml
    5.out_valid_pattern_assign.xml
    6.out_valid_pattern_full_spec.xml
    7.out_multiclauses1.xml

- cd <PATH_TO_PROJECT_DIRECTORY>/Team21/Tests21/Iteration_2/out/invalid
- The result xml files produced are:
    1.out_invalid_Select_clause.xml
    2.out_invalid_With_clause.xml
    4a.out_invalid_Calls.xml
    4b.out_invalid_Next.xml
    4c.out_invalid_ModifiesP_UsesP.xml
    5.out_invalid_pattern_assign.xml
    6.out_invalid_pattern_full_spec.xml

To check the result xml files for Iteration 3:

- cd <PATH_TO_PROJECT_DIRECTORY>/Team21/Tests21/Iteration_3/out/valid
- The result xml files produced are:
    1.out_valid_simple_affects.xml
    2.out_valid_simple_affectsStar.xml
    3a.out_valid_complex_affects.xml
    3b.out_valid_complex_affects_star.xml
    4.out_multiclauses1.xml
    5.out_valid_multiclause_c1.xml
    6.out_valid_multiclause_c2.xml
    7.out_valid_multiclause_c3.xml
    8.out_valid_multiclause_c4.xml
    9.out_multiclause_stress_test.xml
    10.out_nested_stress_test.xml

- cd <PATH_TO_PROJECT_DIRECTORY>/Team21/Tests21/Iteration_3/out/invalid
- The result xml files produced are:
    1.out_invalid_simple_affects.xml
    2.out_invalid_simple_affectsStar.xml
    3a.out_invalid_complex_affects.xml
    3b.out_invalid_complex_affects_star.xml
    4.out_invalid_multiclause.xml

To check the result xml files for Iteration 3 extensions:

- cd <PATH_TO_PROJECT_DIRECTORY>/Team21/Tests21/test_extensions/out/valid
- The result xml files produced are:
    1.out_do_for.xml
    2.nextbip_nextstarbip.xml
    3.affectsbip.xml

(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_iter3_tests.sh` located in `tests > Iteration_3 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.

We also made use of python tocorrect the test queries sequence number, commenting each statement with a statement number, permutating and generating the answers in Tuple format.

(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.

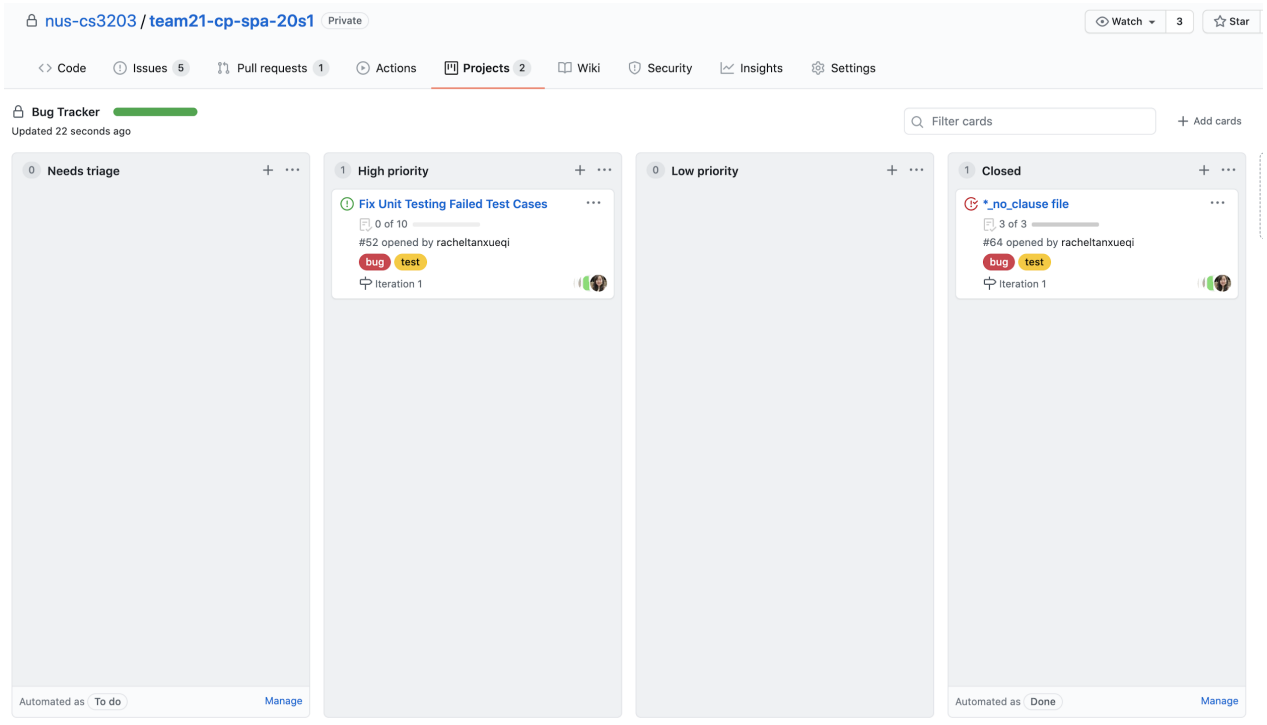


Fig. 6.1.1: Bug Tracking Tool

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.

For iteration 3, we updated the Tokenizer test cases to include the testing of do-while and for 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(){}"`	`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 3, we included tests for proper parsing of SIMPLE programs with do-while and for 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

Two Sample Unit Test Cases

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`	`procedure main{`	`PKB pkb = PKB() <PKB is manually`	isParentStar(1, 3)

of Parent* Relationships	while (1 == 1) { while (2 == 2) { x = 2; }}}	constructed by inserting values> updateParent() updateParentStar() REQUIRE(isParentStar(1, 3) == true)	should return 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; stmts; 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"); PatternAssign pttn = PatternAssign("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((PatternAssign *) 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"};` `list<string> res;`	`formatResults(raw, res)` `res == {"1", "2", "3"}`	list<string> res should contain all the results in raw results

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. Test Category 1: Correctness of Select/result clause without with, such that or pattern clause

In iteration 3, the queries should be able to select synonyms and its attribute names of the design entities and return the correct results in tuple or BOOLEAN. To ensure the result clause is producing the correct format of result according to the SPA requirement, we first start by testing the select clause with the result clause without with, such that or pattern clause.

Firstly, the correctness of the format of result is tested by selecting the following without with, such that or pattern clause.

Select	Should return
BOOLEAN	TRUE/FALSE
Statement (stmt/read/print/call/while/if/assign) Program line	Statement number
Variable	Name
Procedure	Name
Constant	Constant value
Tuple(<... , … >)	Tuple values

Fig. 6.4.1-1: Format of result

Secondly, to further ensure the correctness of the results returned by our system, the queries are created to fall into 4 main categories:

1. valid with result
2. valid but has no result
3. semantically invalid
4. syntactically invalid

The select queries in the Fig. 6.4.1-1 falls into the valid with result and valid but has no result categories. Selecting the correct attribute name of the synonym statement, variable procedure or constant is also considered a valid query.

For syntactically invalid cases, select BOOLEAN and any other return clause should return an empty string. Such syntactically invalid queries happen when there is no or incorrect declaration of design entities and selecting anything other than the options in Fig. 6.4.1-1. For example, selecting an integer. For semantically invalid cases, select BOOLEAN should return FALSE and other return clause should return empty string. Such semantically invalid queries happen when selecting an incorrect attribute name of the synonym statement, variable, procedure or constant. For example, select v.value where v is variable synonym is semantically invalid.

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

6.4.2. Test Category 2: With clause

Identifying what needs to be tested

In this test category, the purpose is to test the correctness of the With clause, *with-cl*. The select clause consists of only *With* clause and adding variation return result format. With clause should be able to deal with attrnames for all design entities except proc_line and both the args it takes in must be of the same type. Hence, the following describes how the test cases are created.

Source Design Considerations

The source program is specially designed to cater to *with-cl*. This includes having a few multiple procedures and call design entities. Having repeated variable names or constants in different statements allows for *with-cl* to allow for statements to share the same design abstraction.

Queries Design Considerations

Generate every possible *ref* combinations to form the *with-cl* query.

LHS ref	RHS ref
“” IDENT “”	“” IDENT “”
“” IDENT “”	INTEGER
“” IDENT “”	attrRef
“” IDENT “”	synonym
INTEGER	INTEGER

INTEGER	“” IDENT “”
INTEGER	attrRef
INTEGER	synonym
attrRef	attrRef
attrRef	“” IDENT “”
attrRef	INTEGER
attrRef	synonym
synonym	synonym
synonym	“” IDENT “”
synonym	INTEGER
synonym	attrRef

Fig. 6.4.2-1: Combination with ref in with-cl

They are then further categorised into **valid**, **syntactically invalid** or **semantically invalid** *with-cl* queries.

Valid *with-cl* query. The LHS and RHS refs must be of the same type. For example, both strings or both integers.

```
prog_line n; stmt s; variable v;

Select BOOLEAN with n = 1
Select BOOLEAN with s.stmt# = 1
Select BOOLEAN with v.varName = “VARIABLENAME”
```

Both n and s.stmt# on the LHS ref have the same value type of an integer which is equal to the RHS ref which is an integer type. Similarly, v.varName on the LHS ref has the a value type of string which is equal to the RHS ref “VARIABLENAME” which is a string type.

Invalid *with-cl* query. For a ref of synonym type, a valid query will have a synonym of type ‘prog_line’ while any other synonym types would be classified as a semantically invalid query. For example,

```
variable v;

Select BOOLEAN with v = “VARIABLENAME”
```

The above query is **semantically invalid** as v violates allowable synonym type for ref. On the contrary, for ref of attrRef type, 'prog_line' do not have an attrRef and using that results in an semantically invalid query while other synonym's attrRef type would be a valid query.

A **syntactically invalid** query design consideration includes attrRef of incorrect type. For instance, assuming that synonyms have been declared:

```
constant c;

Select BOOLEAN with c.stmt# = "20"
```

Having c.stmt# violates the grammar rule for attribute name of the synonym type. Hence, it is a syntactically invalid query. Therefore, these considerations for a valid or invalid *with-cl* queries are carefully designed and added to test the correctness of *with-cl*.

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` `5000`	`3`

Fig. 6.4.2-2: Sample test cases

6.4.3. Test Category 3: Such that clause for relationship between Statements/program lines

Identifying what needs to be tested

The purpose is to test the design abstractions for relationship between statements. This includes clauses such as *Follows, Follows*, Parent, Parents*, Next, Next**. There is only a relationship between statements if they are in the same procedures. For Next, Next*, testing if the relationships are correctly abstracted in a while loop is important as statements before itself can be executed due to the looping. Hence, the following describes how the test cases are created.

Source Design Considerations

The source program is specially designed to cater to such that clause, *suchthat-cl,* that has a relationship between statements or program lines. Firstly, the source program should contain statements of different design entities which are on the same nesting level to cater to *Follows* and *Follows** design abstraction. Secondly, having a few nested loops with statement lists to cater to *Parent* and *Parents** design abstraction. Lastly, between every statement, there is a design abstraction established the *Next* and *Next** relationship.

The design considerations include having different CFG in each procedure in a source program. Each CFG will have each possible combination of while, if and other statements types so that the execution path differs.

Types		Test Intention
`if(a==b) then {` `  ...` `} else {` `  ...` `}`		- No statements Follows/Follows* if statement - Has Parent/Parents* relationship - Has statement executable after the if statement for Next/Next* relationship
`While (a==b) { // 1` `... // 2` `}`		- No statements Follows/Follows* while statement - Has Parent/Parents* relationship - Has statement executable after the if statement for Next/Next* relationship
`if(a==b) then {` `  while(c < d) {` `  ...` `  }` `} else {` `  ...` `}` `}`	`While (c<d) {` `  if(a==b)then{` `...` `} else {` `...` `}` `}`	- No statements Follows/Follows* container statement - Has Parent/Parents* relationship - Has statement executable after the container statement for Next/Next* relationship
`if(a==b) then {` `  Left = right + 1;` `  while(c < d) {` `  ...` `  }` `} else {` `  ...`	`while(c < d) {` `  Left = right + 1;` `  if(a==b)then{` `  ...` `  }else {` `  ...` `  }`	- No statements Follows/Follows* container statement - Has Parent/Parents* relationship - Has statement executable after the container statement for Next/Next* relationship - Container positioned as the last statement of parent container

`}`	`}`	
`if(a==b) then { ... } else { while(c < d) { ... } Left = right + 1; } Left = right + 1;`	`while(c < d) { if(a==b)then{ ... }else { ... } Left = right + 1; } Left = right + 1;`	- Has statements Follows/Follows* container statement - Has Parent/Parents* relationship - Has statement executable after the container statement for Next/Next* relationship - Container positioned as the first statement of parent container
`if(a==b) then { ... } else { Left = right + 1; while(c < d) { ... } Left = right + 1; } Left = right + 1;`	`while(c < d) { Left = right + 1; if(a==b)then{ ... }else { ... } Left = right + 1; } Left = right + 1;`	- Has statements Follows/Follows* container statement - Has Parent/Parents* relationship - Has statement executable after the container statement for Next/Next* relationship - Container positioned as the middle statement of parent container

Fig. 6.4.3-1: Variation of the Positioning of container statements and CFG path

Queries Design Considerations

Generate every possible *args* combinations to form the *suchthat-cl* query. Since Follows,Follows*,Parent,Parent*,Next,Next* established a relationship between statement or program lines, the following can be generated and classified as a valid query design.

LHS args	RHS args
synonym	synonym
synonym	wildcard
synonym	INTEGER
wildcard	synonym
wildcard	wildcard
wildcard	INTEGER
INTEGER	synonym
INTEGER	wildcard
INTEGER	INTEGER

Fig. 6.4.3-1: Combination with stmtRef/lineRef args in suchthat-cl

Valid *suchthat-cl* query. The above Fig. 6.4.3-1 shows arguments that form syntactically valid suchthat-cl queries along with having a variation of Select result clause in BOOLEAN or tuple. The valid queries are designed to return with some result while some do not return any result.

Semantically invalid suchthat-cl query. While both args could take in synonym or INTEGER, when these args are the exact same value, it will be classified as semantically invalid as there is no relationship between its own statement for Follows, Follows*, Parent, Parents* and Next clauses.

Syntactically invalid suchthat-cl query. When the args does not satisfy the Fig.6.4.3-1 shown above, it will be classified as syntactically invalid and empty return should be returned. For example, passing in an expression statement into the RHS args is invalid as args does not take in an expression. Passing negative INTEGER is also syntactically invalid as there are no lines that are negative statement numbers.

```
stmt s;

Select BOOLEAN with Next*(1, ”a*b”)
Select s with Next*(-1, s)
```

Two Sample System Test Cases

Source Code Snippet	Required Test inputs	Expected Test Results
procedure Main { ... while(g!=f) { // 1 a = b + c; // 2 g = f + 2; // 3 } ... }	8 - Invalid Next: cannot execute itself after Select BOOLEAN such that Next(2,2) FALSE 5000	FALSE
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. 6.4.3-2: Sample test cases

6.4.4. Test Category 4: Such that clause for relationship between Statement/Procedure and Variable

Identifying what needs to be tested

The purpose is to test the design abstractions for relationship between statements, procedure and variables. This includes clauses such as Modifies and Uses for all statements including procedure calls, as well as procedures (*ModifiesS, UsesS, ModifiesP, UsesP*). Hence, the following describes how the test cases are created.

Source Design Considerations

- Source program should have assignments, while loops, if-then-else statements, call statements, print, read
- To cater to the Modifies and Uses procedures and procedure calls relationship, there should be multiple procedures and call statements.
- read statements to cater to the Modifies design abstraction
- print statements to cater to the Uses design abstraction

Queries Design Considerations

Similar to the previous test category, every possible *args* combinations are formed to satisfy the suchthat-cl query for *ModifiesS, UsesS, ModifiesP, UsesP.*

Semantically Invalid *suchthat-cl* query. Create query that leads to ambiguity as it is a semantically invalid query case. For example, the args for *Modifies* and *Uses* can be of type wildcard. However, it is unclear if the wildcard refers to a statement or procedure. The RHS args of Modifies and Uses relationship only allows for name, wildcard or synonym of variable type. Hence, any other types used in the RHS args will be semantically invalid. For example,

```
variable v; assign a; stmt s; call cp; procedure p;
Select v such that Modifies(_,v)
Select v such that Uses(_,v)
Select a such that Modifies(s,a)
Select a such that Modifies(cp, p)
```

Syntactically invalid *suchthat-cl* query. Passing in an args that violates the *entRef* rules for RHS args of Modifies and Uses. For example, passing in an INTEGER as the RHS args. This will be classified as a syntactically invalid query and return an empty string regardless of result return type.

Two Sample System Test Cases

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

Procedure Medium2 { ... }	5000	
	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. 6.4.4-1: Sample test cases

6.4.5. Test Category 5: Such that clause for Relationship between Procedures

Identifying what needs to be tested

The purpose is to test the design abstractions for relationship between Procedures which are *Calls* and *Calls** abstraction. Hence, the following describes how the test cases are created.

Source Design Considerations

As a start, the source program is designed with multiple procedures including direct and indirect calls statements between different procedures. To ensure the PKB extract the calls relationship correctly, more source programs with complicated call graphs were tested.

Valid source program. Design a source where there are no cyclic calls between procedures regardless of how complicated the call graphs are. PKB should extract and store the relationship with no exception thrown.

Invalid source program. Design a source where there is a cyclic call between procedures. The system, specifically the PKB, is expected to throw an exception and exit the program.

Queries Design Considerations

Similar to previous test category, every possible *args* combinations are formed to satisfy the suchthat-cl query for *Calls,Calls**.

Semantically Invalid *suchthat-cl* query. Any synonym, wildcard or IDENT *args* for *Calls* and *Calls** but not a procedure type is semantically invalid. When both args are of procedure type but it is directly or indirectly calling itself is semantically invalid too. For example,

```
procedure p,q; assign a;
Select BOOLEAN such that Calls(p,a)
Select BOOLEAN such that Calls*(q,q)
```

Two Sample System Test Cases

Source Code Snippet	Required Test inputs	Expected Test Results
procedure p1 {	1 - Valid Calls*: syn on	p1 p2,p1 p3,p2 p3,

call p2; call p3; } procedure p2 { call p3; } procedure p3 { … }	both args procedure p,q; Select <p,q> such that Calls*(p,q) p1 p2,p1 p3,p2 p3, 5000	
procedure p1 { call p2; } procedure p2 { call p1; }		Invalid calls Exception: Recursive procedure calls in SIMPLE source

Fig. 6.4.5-1: Sample test case

6.4.6. Test Category 6: Such that clause for Relationship between Assignments

Identifying what needs to be tested

The purpose is to test the design abstractions for *Affects and Affects**. For any procedure calls, if a variable is being modified in a different procedure call along its control path, there is a modified relationship. Testing if the relationships are correctly abstracted in a while loop is important as statements before itself can be executed due to the looping and *Affects/Affects** relationship could hold. Hence, such cases need to be tested and the following describes how the test cases are created.

Source Design Considerations

There are a few design consideration to add to the source program as it test the relationship between assignment statements for Affects and Affects*. For example, having assignment statements as well as different types of non-assignment statements including read, print and call.

```
procedure p1 {
a=3; //1
b= 0; // 2
read a; //3
print b; //4
a = b + c; //5
d = a; //6
}

Affects(1,5) does not hold as statement 3 does modifies variable a
Affects(5,6) holds
```

To further test the correctness of *Affects, Affects** algorithm and ensuring it extracts the design abstractions, we add complexity to the source code by varying the positioning of statements with Affects, Affects* relationship within the nested loops. There are also call statements positioned within and outside of the nested loops or containers. Some procedures that are being called modify the variables from that procedure and causing *Affects, Affects** relationship to not hold.

Queries Design Considerations

Similar to previous test category, every possible *args* combinations are formed to satisfy the suchthat-cl query for *Affects, Affects**. For synonym used in *args,* only synonym of assignment type is considered as a valid query. Other synonym type such as while, read, print and etc would be semantically invalid.

Two Sample System Test Cases

Source Code Snippet	Required Test inputs	Expected Test Results
`procedure p1 {` `a=3; //1` `b= 0; // 2` `call p2; //3` `a = b + c; //4`	`1 - Valid Affects*: INTEGER on both args, no result` `Select BOOLEAN such that Affects*(1,5)`	`FALSE`

d = a; //5 } procedure p2 { read a; //6 print b; //7 }	FALSE 5000	
	2 - Valid Affects*: syn on lhs and integer on rhs prog_line n; Select BOOLEAN such that Affects*(n,5) 2,4 5000	2,4

Fig.6.4.6-1: Sample test case

6.4.7. Test Category 7: Pattern clause

Identifying what needs to be tested

The purpose is to test the design abstractions for pattern assign, while and if types. For pattern-assign, it handles full expression partial and exact matching. All variables used in the cond_expr of while and if statements should be correctly identified. Hence, such cases need to be tested and the following describes how the test cases are created.

Source Design Considerations

Types of assignment statements in the source to test pattern-assign partial and exact matching of the expression.

Types	Sample source snippet	Rationale
Expression that contains variables, constants and operators	`procedure Main {` `...` `a = b + c * 10;` `...` `}`	Satisfy pattern assign syntax grammar rule
Expression with variations in brackets to establish priority	`procedure Main {` `...` `a = b + c * 10; // 1` `a = (b + c) * 10; // 2` `...` `}`	Having brackets in statement 2 result in `b+c` forming a sub tree while statement 1 do not have `b+c` as a sub tree
Multiple assignment statements that creates different AST tree	`procedure Main {` `...` `a = b + c * 10; // 1` `a1 = b1 * 2 - c1; //2` `...` `}`	
Same expressions in assignment statement but do not have generate same same tree	`procedure Main {` `...` `a = b-c*d/e; // 1` `a = d/e-x; //2` `...` `}`	Both statements have `d/e` expression. It is a partial match for statement 2 subtree but not statement 1.
Same expressions in assignment statement, have same sub tree	`procedure Main {` `...` `a = b-c*d/e; // 1` `a = x-c*d; //2` `...` `}`	Both statements would have a subtree partial matching of `c*d` expression. Test for assignments with common expression results.
Positioning of assignment statements in while loops, if-then-else containers		

Fig.6.4.7-1: Source design consideration for pattern assign type clause

Having While loops and if-then-else statements with conditional expressions that contain variables to establish pattern while and if-else-then design abstractions.

Types	Sample source snippet	Rationale
Have variables and constants in the *cond_expr* of the While loops and if container	`procedure Main {` `while (a == 2) {` `...}` `if (b == 3) then {..}` `else {..}` `}`	Satisfy pattern while and if syntax grammar rule Test for control variables in the containers
Complicated *cond_expr*	`Procedure Main {` `while` `(((a<s)&&((d>f)||((55/aa` `)<=66)))  &&` `((((!(true==(a+2))) &&` `(0==0)) &&` `(11>=(q*(22+44)/w%e)))) {` `...` `}}`	Adds complexity to make sure it stores the container's control variables are correctly
Have nested while loops and if-else-then containers		

Fig.6.4.7-2: Source design consideration for pattern while and if type clause

Queries Design Considerations

Similar to the previous test category, every possible *args* combinations are formed to satisfy the *pattern-cl*. For a valid pattern assign, while and if type query test design, we want to

- Test for partial matching and exact matching to the assignments statements designed in the source program.
- Test for the control variables in the *cond_expr* of while and if containers
- Add complexity to test the validation correctness by adding spaces in the query, between the pattern *args,* between the expressions of pattern assign type

Invalid Pattern assign type query. Passing in an expression for partial matching or exact matching but do not have matching brackets will be considered as a **syntactically invalid** pattern assign type query. Hence, we want to test if the query processor validates the *args* passed correctly. When the LHS args is a constant synonym, it is **semantically invalid** as LHS *args* can only be a synonym of variable type.

Invalid Pattern while and if type query. For pattern while and if type, it should have two and three *args* respectively. The second *args* in while type and, second and third args in if type should only be of a wildcard type. If these are violated, they are considered an **syntactically invalid** *pattern-cl*. Similarly, the first args for both types can take in synonyms but only of a variable type or it will be considered as a **semantically invalid** *pattern-cl.*

Two Sample System Test Cases

Source Code Snippet	Required Test inputs	Expected Test Results

<table>
<tr>
<td rowspan="2">procedure Match {
   a = b + c; // 1
   call Match2; // 2
...
}

procedure Match2 {
   while (d == e) { // 3
     if (f == d) then {//4
     ...
     }
   }
...
}</td>
<td>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</td>
<td>a</td>
</tr>
<tr>
<td>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</td>
<td>3 d, 3 e</td>
</tr>
</table>

Fig. 6.4.7-3: Sample test cases

6.4.8. Test Category 8: Multi Clauses

Identifying what needs to be tested

The purpose is to test if the queries are 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*, Affects, Affects**. Hence, the following describes how the test cases are created.

Valid Multi Clauses Queries

To test the correctness of our intermediate results, the workflow to generate queries with multi clauses is as such

C1. Clauses without synonyms

```
stmt s;
Select BOOLEAN such that Follows(1,2) and Parent(5,6) // C1. all constants
```

C2. Same group of clauses with disconnected synonyms

a. Non-selected disconnected synonyms
b. Selected disconnected synonyms

```
variable v1,v2,v3,v4; stmt s1,s2,s3,s4,s5; while w; if ifs; assign a1,a2; prog_line n;
```

```
// C2a.
Select s5 such that Follows(s1,s2) and Modifies(s3,„v1)  and Uses(s4,v2) // such that clauses
Select BOOLEAN with s1.stmt# = 1 and v1.varName = “a” // with clauses
Select n pattern a1(v1,_) and pattern a2(v2,_) and w(v3,_) and ifs(v4,_,_) // pattern clauses
```

```
// C2b.
Select s2 such that Follows(s1,s2) and Modifies(s3,„v1)  and Uses(s4,v2) // C2b. for such that clauses
Select <s1,a> with s1.stmt# = 1 and v1.varName = “a” // C2b. for with clauses
Select a1 pattern a1(v1,_) and a2(v2,_) and w(v3,_) and ifs(v4,_,_) //C2b. for pattern clauses
```

C3. Same group of clauses with connected synonyms

a. Start with 1 and gradually increase the number of connected synonyms
b. Non-selected connected synonyms
c. Selected connected synonyms

```
procedure p1,p2; variable v1; assign a1; while w,w2; stmt s1;
```

```
// C3a&b.
Select BOOLEAN such that Calls(“main”,p1) and Calls(p1,p2) // 1 connected syn
Select a1 such that Calls(“main”,p1) and Calls(p1,p2) and Modifies(p2,v1)  // 2 connected syn
Select p.procName pattern a1(v1,_”a”_) and a1(v1, _”a+b”_) and w(v1,_) // 2 connected syn
Select a1 such that Calls(“main”,p1) and Modifies(p1,v1) and Uses (w,v1) and Parents(w,w2) // 3 connected
syn
```

```
// C3a&c.
Select p1 such that Calls("main",p1) and Calls(p1,p2) and Modifies(p2,v1)  // 2 connected syn
Select <a1, w, v1> pattern a1(v1,_"a"_) and a1(v1, _"a+b"_) and w(v1,_) // 2 connected syn
Select <p1, v1, w> such that Calls("main",p1) and Modifies(p1,v1) and Uses (w,v1) and Parents(w,w2) // 3 connected syn
```

C4. Clauses across different groups (such that, with, pattern) in sorted order (Section 3.5.3.3)

a. Selected/non-selected disconnected synonyms
   i. Most restrictive clauses to place at the start of query
b. Selected/non-selected connected synonyms
   i. Place those that will be computed in the previous clause at the start of the query
   ii. Start with 1 and gradually increase the number of connected synonyms

```
stmt s1,s2,s3; procedure p1,p2; if ifs;
```

```
// C4a.i.
Select s1 with s1.stmt# = 3 such that Follows(1,2) and Parents(3,s2) // with clause is the most restrictive placed at the start, followed by Follows and Parents

Select BOOLEAN with s1.stmt# = 3 such that Calls(p1,p2) and Affects*(s2,s3) // Affects* at the last position of the query as it is the least restrictive compared to the rest
```

```
// C4b.i & ii.
Select s1 such that Follows(s1,s2) and Parents(s2, ifs) and Calls(p1,p2) // s2 will be computed in Follows and used in Parents

Select <s1, s2, ifs, v1> such that Follows(s1,s2) and Parents(s2, ifs) and Uses(ifs, v1) pattern ifs(v1,_,_) with s1.stmt# = 10 // increase the number of connected synonyms, i.e. 4
```

C5. Evaluate Select

a. Select part of intermediate results such as one of the connected synonyms to obtain final results
b. Vary results format in BOOLEAN or tuple format

Invalid Multi Clauses Queries

To further test the correctness of our system, syntactically invalid multi clause queries are tested. The system is expected to return an empty string upon any selected type. Syntactically, *and* connects clauses of the same type that do not contain the keyword (such that, with or pattern) and should not be used to connect or introduce clauses of different types. The following queries are syntactically incorrect:

```
Select a such that Parent* (w, a) and Modifies (a, "x") and such that Next* (1, a)
Select a such that Parent* (w, a) and pattern a ("x", _) such that Next* (1, a)
Select a such that Parent* (w, a) pattern a ("x", _) and Next* (1, a)
```

6.4.9. Test Category 9: Stress Testing

Once we have thoroughly tested our SPA system on its correctness of the individual clauses as well as multi clauses, we want to further test the efficiency performance of our system. To do so, we perform a stress test on our system.

Source Design Consideration

- 500 lines of statement lines
- Have multiple procedure calls with different control flow graph in each procedure (Section 6.4.3)
- Have nested loops (Section 6.4.11)
- Vary the positions of design entity statements (Section 6.4.11)
- Finding the maximum limit our parser can handle
    - Still able to handle a 1000 lines of statement lines in source program
    - Able to handle 100 nested containers

Query Design Consideration

The next step after testing on our multi clauses queries correctness is to do a stress test to ensure that the optimisation in our system is able to meet the time constraints. We tested up till 66 clauses on each of the following category mentioned below. The select clause includes selecting syn, tuple size of less than 250000 or BOOLEAN and it is still working well. The workflow to generate the queries are

C6. Clauses across different groups in unsorted/random order

a. Enumerate the number of diff type clauses in each query, starting with 2 clauses in 1 query.
b. Zero connected synonyms to test on cross product and our system performance
c. Enumerate the number of synonyms which will be computed in a previous clause in ascending order

Stmt s1,s2,s3,s4; assign a1,a2,a3,a4; while w1,w2; variable v,v1,v2; if ifs; procedure p1;
//C6a. Select s1 such that Affects*(s1,s2) such that Follows*(s2,3) and Next*(s3,s2) with p1.procName = “Main” such that Modifies(p1,v1) pattern a1(v1,_) // 6 clauses across different groups in random order Select BOOLEAN such that Next*(25, w2) such that Modifies(ifs, v) such that Uses(p, v) pattern a(v, _) pattern w1(v, _) such that Parent*(w1, w2) pattern ifs("bA", _, _) // 7 clauses across different groups in random order
//C6b. Select p2 such that Calls*(“P1”, p1) and Calls*(“P2”,p2) and Affects*(s1,s2) and Affects*(s3,s4) pattern a1(v1,_) and a2(_,_) and a3 (v2,_) //C1b. test the cross product performance
//C6c. Select <s1, s2, s3, s4,s5>such that Next*(s1,s2) and Next*(s2,s3) and Next*(s3,s4) and Next*(s4,s5)

```
Select s2 such that Affects*(s1,s2) and Affects*(s2,s3) and Affects*(s3,s4) and Affects*(s4,a1) and Affects*(a1,a2) pattern a2(_,_)
```

C7. Same query that returns the same results but in different orders

```
// queries below are sorted in different order but all return the same results
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”, _)
```

C8. Order the clauses according to the level of restriction in descending order

a. Placing Affects* at the start of the query to test if our optimization properly sort and place Affects* to the last positions in the group for intermediate steps

```
stmt s1,s2,s3,s4; assign a1,a2,a3,a4;
Select <s1,s2,s3,s4,a1,a2,a3,a4> such that Affects*(s1,s2) and Affects*(s2,s3) and Affects*(s3,s4) and Affects*(s4,a1) and Next*(a1,a2) and Next(a3,a4) with a4.stmt# = 3
```

C9. Testing the cross product performance

a. The maximum number of tuple size select can handle on a non-selected disconnected synonym
b. Returning tuple size of 2 with a table of size 500x500 is our system’s limit
c. Returning tuple size of at least 2 but less than a total size of 250000 will not time out

6.4.10. Test Category 10: General Invalid Source and Query Design

The purpose is to ensure there is a large test coverage for both valid but invalid test cases. For **invalid sources**, we want to make our SPA system to throw exceptions with informative description and exit the program. For **semantically invalid queries**, we want our SPA system to return an empty result or FALSE for a return clause of BOOLEAN type. For **syntactically invalid queries**, we want our SPA system to return an empty result regardless of the return clause type. Hence we design the source to contains the following:

Invalid Source Design Consideration

- Invalid conditional expression
- Missing separator
- Source program without procedure structure

- Tokens do not meet the grammar or lexical token rules
- Invalid assign statement
- Cyclic procedure call (Example in Fig. 6.4.5-1)

Two Sample System Test Cases

Invalid sample source snippet	Expected assert
`procedure main {` `while(i && j) {` `...` `}` `}`	`Invalid While Statement: expected relative expression comparator, got '&&'.`
`x = x + 1;`	`Invalid Procedure: expected 'procedure', got 'x'.`

Fig. 6.4.10-1: Example of invalid source

Invalid Query Design Consideration

In addition to the individual with, such that and pattern clauses that will form an invalid queries if it violates its individual clause's rules, the following shows more possible design consideration to make a query invalid as it violates the SIMPLE grammar syntax rule in general.

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

6.4.11. Test Category 11: General Valid and Complicated Source Design

To further ensure the correctness of the system, we have designed the source programs and queries at a more complex level. The purpose is to add difficulty to the design abstraction of with, such that and pattern clauses. Test category 1 - 7 have sources that are designed at a rather easy complexity to keep the test focused on ensuring PKB and PQL correctness. With a more complicated source and query 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.

Nesting Levels for container statements

- 2 x 2 possible combinations between if and while nesting (Section 6.4.3)
- Deeper level of nesting for if and while statements each
- More levels of nesting between if and while

Complicated conditional expression in container statements

Source Code Snippet	Required Test Input	Expected Test Results
`while (((a<s)&&((d>f)||((55/aa )<=66)))  && ((((!(true==(a+2))) && (0==0)) && (11>=(q*(22+44)/w%e)))) {` `  ...` `}`	`1 - Valid Uses variable v; while w; Select v such that Uses(w,v) a,s,d,f,aa,true,q,w,e 5000`	`a,s,d,f,aa,true,q,w,e`
`if (a && b) then {` `...` `} else {` `...` `}`		`Invalid If Statement: expected relative expression comparator, got '&&'.`

Fig. 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. For example, “while = while + 1;” means that “while” is the variable name in an assignment statement of the source program. 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.

6.4.12. Test Category 12: Valid and Confusing Query Design

The valid and confusing 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 in Select clauses such as between clauses and arguments to ensure no flaw in our system's regex matching.

Valid and Confusing Query Design	Example
Declaring synonyms with the same name as the design entities	stmt Select; select Select
Declaring long synonym names	variable longName123longName123longName123longName123; Select longName123longName123longName123longName123
Tricky spaces	assign a; stmt s; Select s such that Calls*(_,_) pattern w(_, _)

Fig. 6.4.12-1: Examples of valid but complicated queries

6.4.13. Test Category 13: NextBip/NextBip* Extension

Identifying what needs to be tested

The purpose of this test is to ensure the correctness of the *NextBip* and *NextBip** extension features.

Source Design Considerations

- Source program contains all statement types.
- Source program contains multiple procedures and calls to those procedures. The call statements are strategically positioned at important points of the program, such as the beginning, middle and end of procedures, loops or if statements.
- Source program contains multiple nested calls.

Query Design Considerations

The query design structure follows similarly to those used to test *Next/Next** clauses by iterating through every possible pair of synonyms on the left and right-hand-side of the *NextBip* and *NextBip** Clause, as detailed in Section 6.4.3. However, we also craft queries to specifically test if the CFG constructed by the SPA program correctly takes called procedures into consideration. Examples of such queries include:

- *NextBip/NextBip** queries where the left-hand side represents a call statement, such as a synonym of design entity 'call' or a call statement number. The system will need to consider the statements of the called procedure.
- *NextBip/NextBip** queries where the right-hand side represents a statement inside some called procedure denoted as *called_proc*. The system will need to consider the statements of procedures that call *called_proc.*

Source Code Snippet	Required Test Input	Expected Test Results

procedure black{ **...** **if(x==4) then { //23** **call purple; //24** **call violet; //25** **} else {** **call lavender; //26** **}** **} //end of procedure black** **...** **procedure violet{** **z=z; //41** **}** **...**	55 - Valid NextBip. LHS int RHS syn. Statement in calls procedure return to current procedure. call cl; Select BOOLEAN such that NextBip(41,cl) TRUE 5000	TRUE We see that in the context of NextBip, the next statement 41 after statement 41 is statement 25, which is a call statement. Thus, the clause should evaluate to TRUE.
procedure grey{ **...** **call orange; //3** **...** **}** **procedure orange{** **...** **if(s==3) then { //9** **...** **}** **...** **} //end of procedure orange** **procedure black {** **...** **while(x==4) { //31** **...** **if(z==2) then { //34** **call indigo; //35** **} else {** **call mauve; //36** **}** **...** **}** **...** **} //end of procedure black** **procedure rainbow{** **call black; //51** **call grey; //52** **}**	92 - Valid NextBip*. LHS int RHS syn. Iterate through stmt# if ifs; Select ifs such that NextBip*(36,ifs) 34,9 5000	34,9 we see that statement 36 is a statement that calls procedure mauve. Trivially, statement 36 has a next relationship with if statement 36 which is in the same while statement. Furthermore, as NextBip*(51,52), NextBip*(52,3), and NextBip*(3,9) holds, then NextBip*(36,9) should hold as well.

Fig. 6.4.13-1: Sample test cases for testing NextBip/NextBip Cases*

6.4.14. Test Category 14: AffectsBip/AffectsBip* Extension

Identifying what needs to be tested

The purpose of this test is to ensure the correctness of the *AffectsBip* and *AffectsBip** extension features.

Source Design Considerations

- Source program contains all statement types.

- Source program contains multiple procedures and calls to those procedures. The call statements are strategically positioned at important points of the program, such as the beginning, middle and end of procedures, loops or if statements.
- Source program contains multiple nested calls.

Query Design Considerations

The query design structure follows similarly to those used to test *Affects/Affects** clauses by iterating through every possible pair of synonyms on the left and right-hand-side of the *AffectsBip* and *AffectsBip** Clause, as detailed in Section 6.4.3. However, we also craft queries to specifically test if the CFG constructed by the SPA program correctly takes called procedures into consideration. Examples of such queries include:

- *AffectsBip/AffectsBip** queries where the left-hand side represents an assign statement that modifies a statement used in a called procedure. The system will need to consider the statements of the called procedure.
- *AffectsBip/AffectsBip** queries where the right-hand side represents an assign statement inside some called procedure that uses a variable modified in the caller procedures. The system will need to consider the assign statements of procedures that call the called procedure.

Source Code Snippet	Required Test Input	Expected Test Results
procedure black{ **...** **if(x==4) then { //23** **a = 342; //24** **call violet; //25** **} else {** **call lavender; //26** **}** **} //end of procedure black** **...** **procedure violet{** **z=a; //41** **}** **...**	55 - Valid AffectsBip. LHS int RHS syn. Statement in calls procedure return to current procedure. assign al; Select BOOLEAN such that AffectsBip(24,al) TRUE 5000	TRUE We see that in the context of AffectsBip, statement 24 modifies a. a is not modified and is used by statement 41 along its CFG path. Thus, the clause should evaluate to TRUE.
procedure grey{ **b = a; //2** **call orange; //3** **...** **}** **procedure orange{** **c = b; //8** **if(s==3) then { //9** **...** **}** **...** **} //end of procedure orange** **procedure rainbow{** **a = 123; //51** **call grey; //52** **}**	92 - Valid AffectsBip*. LHS int RHS syn. Iterate through stmt# stmt s; Select ifs such that AffectsBip*(51,s) 2,8 5000	2,8 We see that procedure rainbow contains line 51 which modifies the value a. Procedure rainbow then calls grey, where statement 2 uses variable a and variable a has not been modified along the CFG path between these 2 statements, thus statement 51

		affects* statement 2 across procedures. Procedure grey then calls orange, where statement 8 uses variable b and variable b has not been modified along the CFG path between these 2 statements, thus statement 51 affects* statement 8 across procedures. Thus, the clause evaluates to return statements 2 and 8.

Fig. 6.4.14-1: Sample test cases for testing AffectsBip/AfectsBip Cases*

6.4.15. Test Category 15: For Loop and Do-While Loop Extension

Identifying what needs to be tested

The purpose of this test is to test the correctness of clause evaluation on a source program containing for and do-while statements. We want to ensure the parser is able to parse the for and do-while structure correctly. Refer to the example below for the structure created by the team. The design extractor should extract the relationship should have no difference just like the existing while loop for the SPA requirement.

Source Design Considerations

- Source program contains all base statement types.
- Source program contains both singular or nested for loops and do-while loops. These loops can be nested within if, while, for or do statements.

Query Design Considerations

The query design structure follows similarly to those used to test the base relationship clauses by iterating through every possible pair of synonyms on the left and right-hand-side of each clause, such as detailed in Section 6.4.4. However, the clauses are tailored to test the extended features by having the clause arguments focused around the ‘for’ or ‘do’ design entities, or statements found within these entities. Examples of cases covered by the query design include:

- Clauses where the arguments represent Design Entities ‘for’ and ‘do’, or statements within for and do-while statements.
- *Next/Next** and *Affects/Affects** Clauses that test whether the statements inside a do-while statement occur before the do-while statements in the CFG generated by the SPA program.
- Uses and Modifies clauses that test whether the for-loop adopts the Uses and Modifies relationships of its initialization and update assign statements.

Source Code Snippet	Required Test Input	Expected Test Results
... **pidgey = 3; //31** **dowhile(fearow % spearow <= skarmory) {** **//32** **pidgey = 5 + pidgeot + pidgey; //33** **}** **pidgey = pidgey + 2; //34** ...	133 - Valid Affects, LHS int and RHS int, No Answer, test that do loop stmtLst executes first Select BOOLEAN such that Affects(31,34) FALSE 5000 134 - Valid	FALSE. By definition of a do-while loop, statement 33 is executed before statement 32. Thus, Affects(31,34) is false as variable 'pidgey' is modified by statement 33.
... **ivysaur = bulbasaur + 16; //1** **dowhile(charmander != everstone) { //2** **charmander = charmander + 1; //3** **}** ...	111 - Valid Next, LHS int and RHS do, No Answer do d; Select d such that Next(1,d) 5000	None Thus, Next(1,d) is false as by definition of a do-while loop, statement 2 is executed before statement 2.
... **for(torchic = combusken; combusken <** **blaziken; mudkip = 55) { //41** ... **}** ...	9 - Valid Modifies. LHS Syn and RHS Name. Select for. Modified in For Conditional Only. for f; Select f such that Modifies(f,"torchic") and Modifies(f,"mudkip") 41 5000	41 For statement 41 Modifies "torchic" in the initialization statement and modifies 'mudkip' in the update statement.

Fig. 6.4.14-1: Sample test cases for testing the correctness of clauses for a source program with for and do-while statements

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. However, in iteration 3, we actually start to see the benefits of the cross-platform solution. The system builds a lot faster on Mac OS than on Windows, and the system tests also run a lot faster on Mac. For example, tests that take 4 minutes to run on Windows can be run on Mac within a minute. This ultimately led to a lot of time saved as we did our system tests on Mac.

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 iterations 2 and 3, we did not use the GitHub issue tracker at all. This was partly because all the issues from iteration 1 have been resolved and the later iterations involved fewer tasks, so we did not see a need to use the issue tracker. We found ourselves turning to our Telegram group chat and frequent meetings to keep each other updated on our respective progress.

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.

In iteration 3, we realised that one reason why we have several late-night sessions was because we start our meetings late. Thus, we readjusted our meeting sessions such that when a longer session is expected, we would meet both in the afternoon and night. This has proven to be really effective as we have not had a late night since. If we were to start the project again, we would take this into account from week 1 of the semester so that no late nights are needed at all.

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. Also, as mentioned in the previous section, one coding session can be divided into several shorter ones where a long session is expected.

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. Affects, Affects*

Affects, Affects*   Overview: API related to the Affects and Affects* clauses
BOOLEAN isAffects(**STMT_NO** s1, **STMT_NO** s2)   ***Description:*** Returns true if s1 Affects s2
LIST_OF_STMT_NO getAffects(**STMT_NO** s)   ***Description:*** Returns all the statements that s Affects
LIST_OF_STMT_NO getAffected(**STMT_NO s**)   ***Description:*** Returns all the statements s is Affected by
BOOLEAN isAffectsStar(**STMT_NO** s1, **STMT_NO** s2)   ***Description:*** Returns true if s1 Affects* s2
LIST_OF_STMT_NO getNextStar(**STMT_NO s**)   ***Description:*** Returns all the statements that s Affects*
LIST_OF_STMT_NO getPreviousStar(**STMT_NO s**)   ***Description:*** Returns all the statements s is Affected* by

8.9. 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.

8.10. 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_STMT_NOS getMatchedWhileVar(**INT** s) ***Description:*** Returns all variable used as a control variables in this while statement
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_STMT_NOS getMatchedIfVar(**INT** s) ***Description:*** Returns all variable used as a control variables in this if statement
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

8.11. Extensions: Do/For Pattern

ForPattern, DoPattern Overview: API related to For and Do Pattern
LIST_OF_STMT_NOS getMatchedDo(**STRING** lhs) ***Description:*** Returns all do-while statement numbers that uses lhs as control variable
LIST_OF_STMT_NOS getMatchedDoVar(**INT** s) ***Description:*** Returns all variable used as a control variables in this do-while statement
LIST_OF_PAIRS_OF_STMT_NOS_AND_VARIABLE_NAMES getMatchedDoPair(**STRING** LHS) ***Description:*** Returns all do-while statement-variable pairs that match the pattern
LIST_OF_STMT_NOS getMatchedFor(**STRING** lhs) ***Description:*** Returns all for loop statement numbers that uses lhs as control variable
LIST_OF_STMT_NOS getMatchedForVar(**INT** s) ***Description:*** Returns all variable used as a control variables in this for loop statement
LIST_OF_PAIRS_OF_STMT_NOS_AND_VARIABLE_NAMES getMatchedForPair(**STRING** LHS) ***Description:*** Returns all for loop statement-variable pairs that match the pattern

8.12. Extensions: NextBip/NextBip*

NextBip, NextBip* Overview: API related to NextBip and NextBip* Clauses
BOOLEAN isNextBIP(**STMT_NO** s1, **STMT_NO** s2, **LIST_OF_RETURN_POINTER** pointers) ***Description:*** Returns true if s2 is the NextBip statement of s1
LIST_OF_STMT_NO getNextBIP(**STMT_NO** s, **LIST_OF_RETURN_POINTER** pointers) ***Description:*** Returns all the NextBip statements of s
LIST_OF_STMT_NO getPrevBIP(**STMT_NO s**, **LIST_OF_STMT_NO** stack) ***Description:*** Returns all the statements s is the NextBip of
BOOLEAN isNextStarBIP(**STMT_NO** s1, **STMT_NO** s2) ***Description:*** Returns true if s2 is the NextBip* statement of s1
LIST_OF_STMT_NO getNextStarBIP(**STMT_NO s**) ***Description:*** Returns all the NextBip* statements of s

LIST_OF_STMT_NO getPrevStarBIP(**STMT_NO s**) ***Description:*** Returns all the statements s is the NextBip* of

8.13. Extensions: AffectsBip/AffectsBip*

AffectsBip/AffectsBip* Overview: API related to AffectsBip and AffectsBip* Clauses
BOOLEAN isAffectsBIP(**STMT_NO** s1, **STMT_NO** s2) ***Description:*** Returns true if s1 AffectsBip s2
LIST_OF_STMT_NO getAffectsBIP(**STMT_NO** s) ***Description:*** Returns all the statements that s AffectsBip
LIST_OF_STMT_NO getAffectedBIP(**STMT_NO s**) ***Description:*** Returns all the statements s is AffectedBip by
BOOLEAN isAffectsStarBIP(**STMT_NO** s1, **STMT_NO** s2) ***Description:*** Returns true if s1 AffectsBip* s2
LIST_OF_STMT_NO getAffectsStarBIP(**STMT_NO** s) ***Description:*** Returns all the statements that s AffectsBip*
LIST_OF_STMT_NO getAffectedStarBIP(**STMT_NO s**) ***Description:*** Returns all the statements s is AffectedBip* by