CS3203 SIMPLE-SPA Project

Iteration 1 Final Report

Team 08	Lau Kar Rui
	Kevin Chin
	Ken Lim
	Ow Zhen Wei
	Tan Mingrui
	Park Se Hyun

Table of Contents

Development Plan

Below is the development plan for Iteration 1. We divided the team into 3 smaller teams: Frontend, PKB, and PQL. The main tasks for Iteration 1 was to implement each part of the programme. The activities of each task are shown below. Each activity includes unit tests and integration tests, whenever applicable.

	Iteration 1					
Members	Data structure and parsing logic for PQL	Parser for Front-end	Shunting yard algorithm for expression parsing	Query Evaluator for PQL	PKB Basic Storage	PKB Relationship Storage
Kar Rui		*				
Ken	*					
Kevin				*		
Mingrui					*	*
Se Hyun					*	*
Zhen Wei		*	*			

	Iteration 1					
Members	Tokenizer for Front-end	Validator for Front-end	PKB Patterns	Evaluating such that clauses	Evaluating pattern clauses	Adding system tests
Kar Rui	*	*				*
Ken					*	*
Kevin				*		*
Mingrui			*			*
Se Hyun						*
Zhen Wei		*				*

Scope of the Prototype Implementation

All features expected to be implemented in Iteration 1 has been implemented. There are also three bonus features:

1. ModifiesP has been implemented in this iteration,
2. the program can process assignment patterns with more than 1 synonym, e.g. pattern a ("x", _"x + 1"_)
3. the parser allows for comments in code. Comments are delimited by the `\\` token and everything after the token until a new line (\n) will be ignored by the parser. You can see this in action in the unit tests (specifically front end unit test #2) near the end of this document.

SPA design

Overview

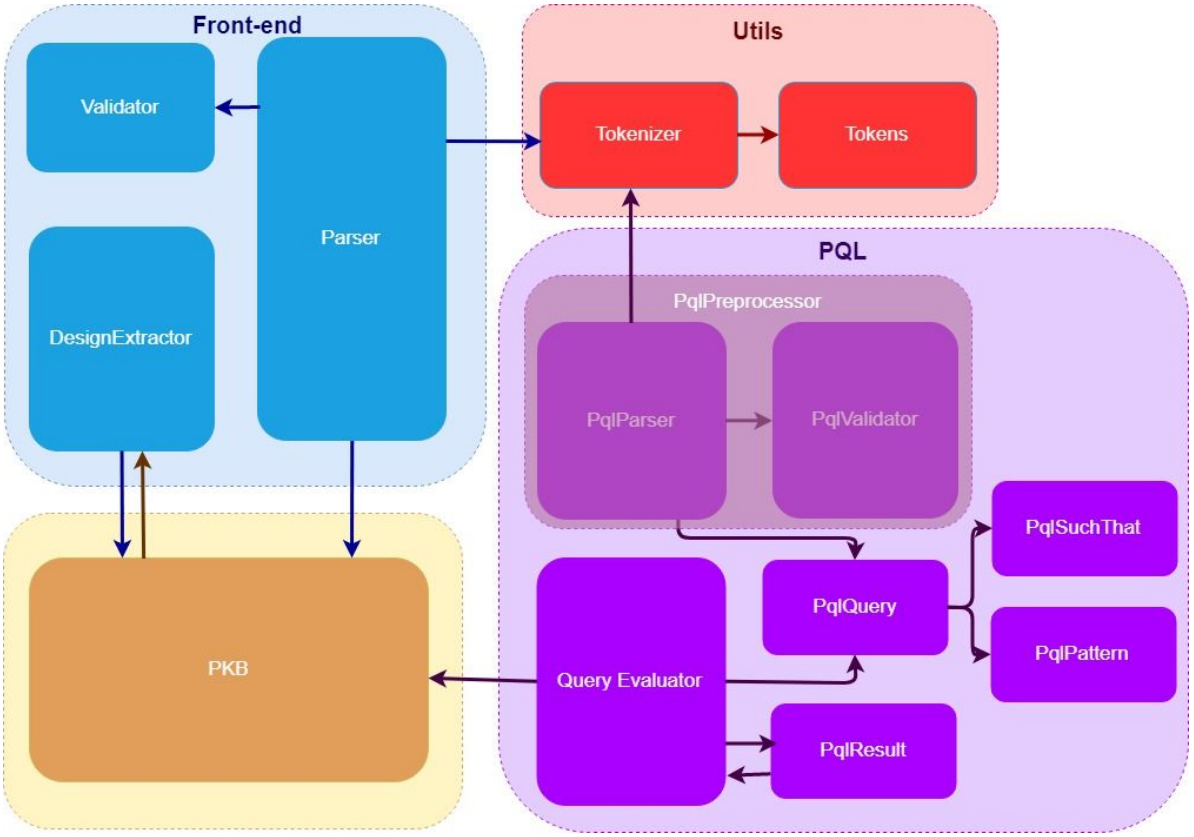
Design of SPA components

The SPA program is divided into various components: Utils, Front-end, PQL and PKB. The responsibility of each components is as such:

> **Front-end:** *Responsible for reading and processing the SIMPLE source code. Processing includes validation and populating the* **PKB.**
>
> **PKB:** *Stores the basic entities and relationships, from the SIMPLE source code processed, using various data structures (i.e Tables and Lists).*
>
> **PQL:** *Responsible for reading and processing the PQL query. Processing includes validation and evaluation of the input query.*
>
> **Utils:** *Contains all general helper classes used by other components.*

Architecture Diagram

The following architecture diagram illustrates the organization and relationship of our SPA program:



- The `Parser` and `DesignExtractor` components of **Front-end** will mainly interact with **PKB** to update the information about the source program.
- The `PqlPreprocesor` component of **PQL** will take in the PQL input from user, parse and validate, and store it in the internal data structure, `PqlQuery`
- The `QueryEvaluator` component of **PQL** will mainly interact with **PKB** for evaluating queries.

SPA front-end Design

The first step of the program is to read in a text file from the GUI program that the user provides. However, the text file has no meaning to the program (and may not even be a valid input) until some sense can be made of the file's contents. This requires a parser to **preprocess** the input, converting the input into data the parser can use, then **extract** data from the tokens and **store** such data into data structures that the PKB holds.

As alluded to above, the source file may not even be a valid input, and thus a **validator** is needed to provide syntactic analysis and stop the processing of the inputs if any syntax errors are found in the source file (by checking if the input matches SIMPLE's Concrete Syntax Grammar). The parser is also expected to **parse** the syntactic structure of the source program, so as to correctly populate design extractions into the PKB.

Thus, the parser is expected to fulfil the responsibilities of preprocessing, perform validation, parse and storing the parsed data into the PKB.

Design Decisions and Justifications

Design Decision: What should be the component design of Parser

Options

There are a few options to fulfil such responsibilities. We came up with a few alternatives, such as:

1. Have a single `Parser` class that performs all the responsibilities going through the source file once, where it simultaneously preprocesses, validates, parses, and stores the data, stopping if the validation fails.
2. Have four distinct classes `Tokenizer`, `Validator`, `Parser`, and `Populator`, each with their own responsibilities, following the Separation of Concerns design principle.
3. Have three main classes `Tokenizer`, `Validator`, and `Parser`, with the parser performing some of the populating roles too, since the parser is the final step before populating, and we can reduce coupling of yet another module. The more challenging design relations (like Parent*, Next*) is left to a smaller class `DesignExtractor`.

Criteria

Before deciding which option we were going to use, we first rated them according to the criterias below, in the order of importance from most important to least:

1. Modularity (Is there high coupling and low cohesion?)
2. Ease of implementation (Can the work be split up easily, and whether tests are able to be written easily?)
3. Performance (Time taken to complete parsing)

Performance was ranked last due to the requirements specifying that parsing is not time sensitive and has no penalty for being slow, and thus is not a real consideration.

Comparison

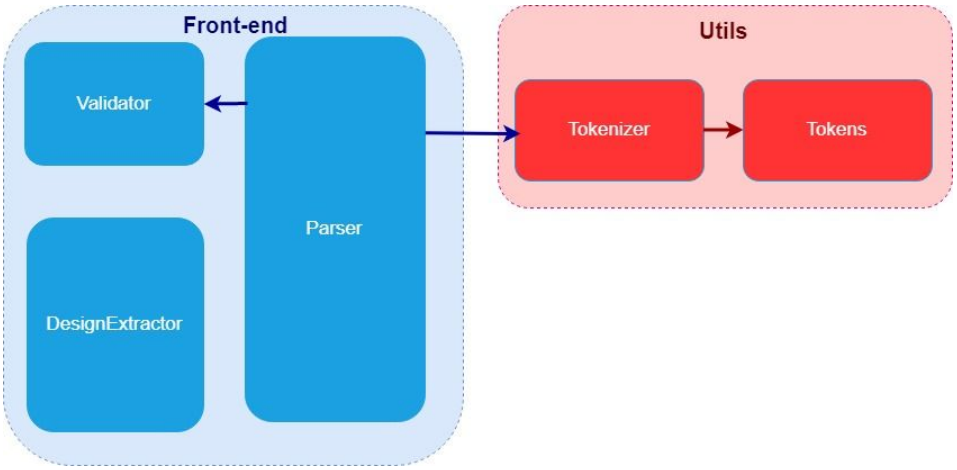
Design Option	Option 1	Option 2	Option 3
Ease of Implementation	- Requires implementing many data structures to store intermediate data. - Workload not easily shared amongst multiple people. - Workload might be too high for a single person.	- Easy to implement since every class has a narrow responsibility. - Might be overkill since only two people are working on front end.	- Easier than Option 2 in implementation since Design Extractor is a narrower class than Populator.
Modularity	- Not modular at all, a	- The most modular	- Lower coupling, and

	single Parser class has multiple responsibilities, and it clearly violates the Separation of Concerns principle.	option, as all classes have distinct responsibilities. - High cohesion, but higher coupling compared to Option 3	slightly lower cohesion as Parser is doing some of the populating too.
Performance (speed of parsing)	- Since everything is done by Parser, it only takes 1 pass of the source input and will be the fastest in parsing	- This design requires multiple passes of the source input to tokenize, validate and parse. - Compared to option 3, there is the additional overhead of passing the parsed data into the Populator class.	- Slightly faster compared to Option 3. - The `DesignExtractor` class will also not need to do redundant extraction of data from the PKB as the Parser has already populated most of the design relations.

Decision

Based on the comparisons, our team decided to go ahead with **Option 3**, which proved to be useful to modularize our components as the `Tokenizer` class was able to be shared with the `PQLParser` class to process their queries, as seen in our component diagram.

From the UML Diagram of our Front End components following our design decision below, it can be seen that Single Responsibility Principle is closely followed, with each component having an obvious role to perform.



Tokenizer (aka Preprocessor)

The tokenizer exists to convert the (string) content of the input file into a vector of Token structs based on various tokenizer functions such as `TokenizeDigits`, `TokenizeParenthesis`, etc for the Validator to validate and for the Parser to parse.

Design Decisions and Justifications

Since we have decided to modularize the Parser into distinct classes with their own responsibilities, there must exist a form of "tokens" for the parser to read instead of just reading the string and checking what the next value is.

Vectors are also chosen over other list data types to store Tokens because we need the tokens to be ordered and vectors are faster in insertion, deletion, and retrieval than other ordered list data structures if the elements are smaller than 32 bytes[1].

Design Decision: Data structure of tokens

Options

1. Simple delimiting of text with regular expressions (regex) and storing it as a vector of strings.
2. Delimiting of text with regex and store the type of string it is (e.g. token types of `Assign`, `OpenBrace`, `Conditional`)

Criteria

Ease of use, extensibility, and ease of implementation were the main criterias used.

Decision

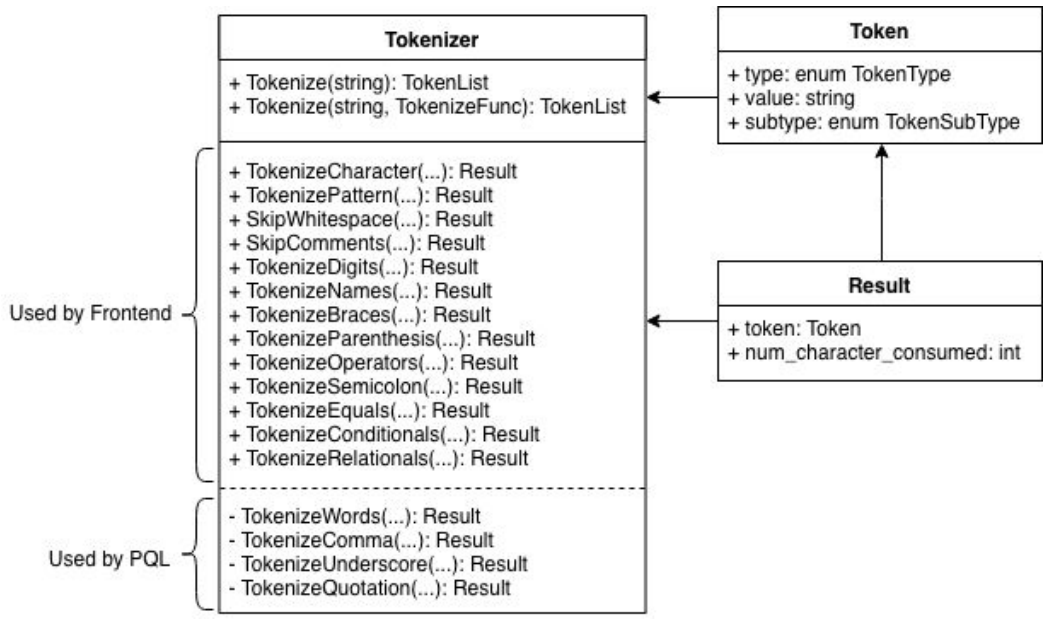
Option 2 was chosen, even if it is more work to implement. As the tokens generated will be used heavily by the `Parser` and `Validator` (and later the `PqlParser`) to validate and extract data, having the type of the token along with the value will be indispensable as it reduces the need to check multiple conditions to see if the value fulfils the type (e.g. checking whether the value matches `!=`, `>`, `<`, etc for an expected conditional token).

Extensibility and ease of use is also way higher in the second option, where anyone can create a new tokenizer function to tokenize their desired characters or patterns. If the first option was chosen, it would also be more work downstream when type checking is needed to ascertain the actions to undertake, be it parsing or validating.

Implementation

Great care has been taken for Tokenizer to be easily extensible, and leaves the possibility for Open-Closed principle to be applied (even though it is not necessary for this project). The UML diagram of the class can be seen below:

[1] https://baptiste-wicht.com/posts/2012/11/cpp-benchmark-vector-vs-list.html



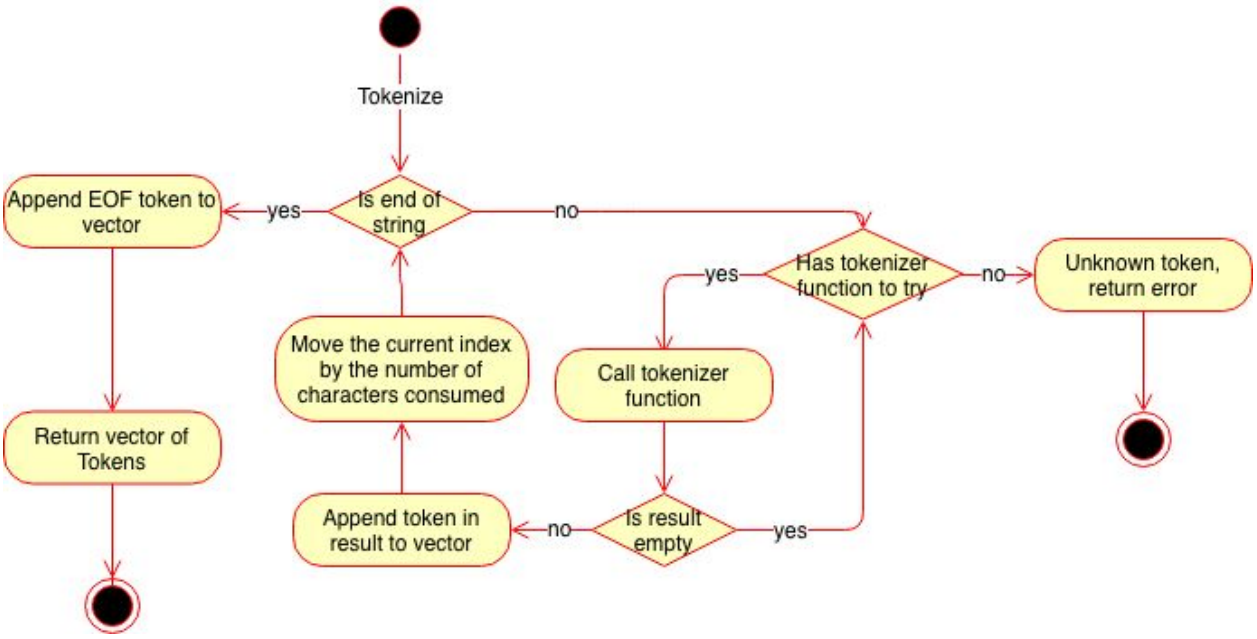
The main backbone of the Tokenizer class are the `Tokenizer::TokenizeCharacter` and `Tokenizer::TokenizePattern` functions, which all the other functions use by supplying either a character value or a specific regex respectively in order to obtain a Result object containing a Token which will be added into the current TokenList.

As you can imagine, extending the tokenizer to tokenize other needed inputs such as quotations are as simple as creating a new function using any of the above two functions. In fact, extended functions such as `TokenizeComma` is what the `PkbParser` uses to tokenize their own `PkbQuery` objects.

A step by step of how the Tokenizer works:

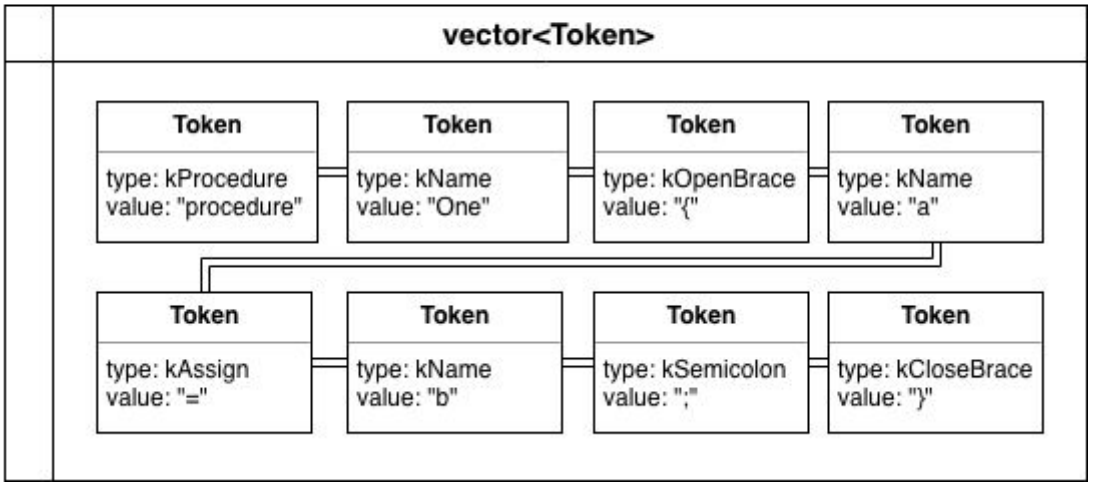
1. Tokenizer loops through array of tokenizer functions supplied to tokenize input
2. If non empty result is found (meaning something got tokenized), loop will break and the token result will be appended into a vector containing all the tokenized inputs.
3. Pointer to current character of the input will be increased by number of characters consumed tokenizing the previous token.
4. The tokenizer loops restarts, and steps 1 - 3 repeats until all the input has been consumed.
5. When the input has been processed, a final `EOF` (End of File) token will be appended to the vector and the vector will be returned by the `Tokenize` function. The `EOF` token will be useful for the Parser and Validator to know when the token stream is about to end.

The action diagram below better illustrates this process:



The figure below shows what the `Tokenizer::Tokenize` function returns when it tokenizes a simple program.

```
procedure One {
    a = b;
}
```



After the vector of `Token`s has been returned, it is then passed into the `Validator` class to perform validation using the `Validator::ValidateProgram` function.

Validator

The validator checks for syntax errors according to SIMPLE's Concrete Syntax Grammar (CSG), and the program returns an error message and stops immediately if any errors are found, preventing the Parser from doing unnecessary work.

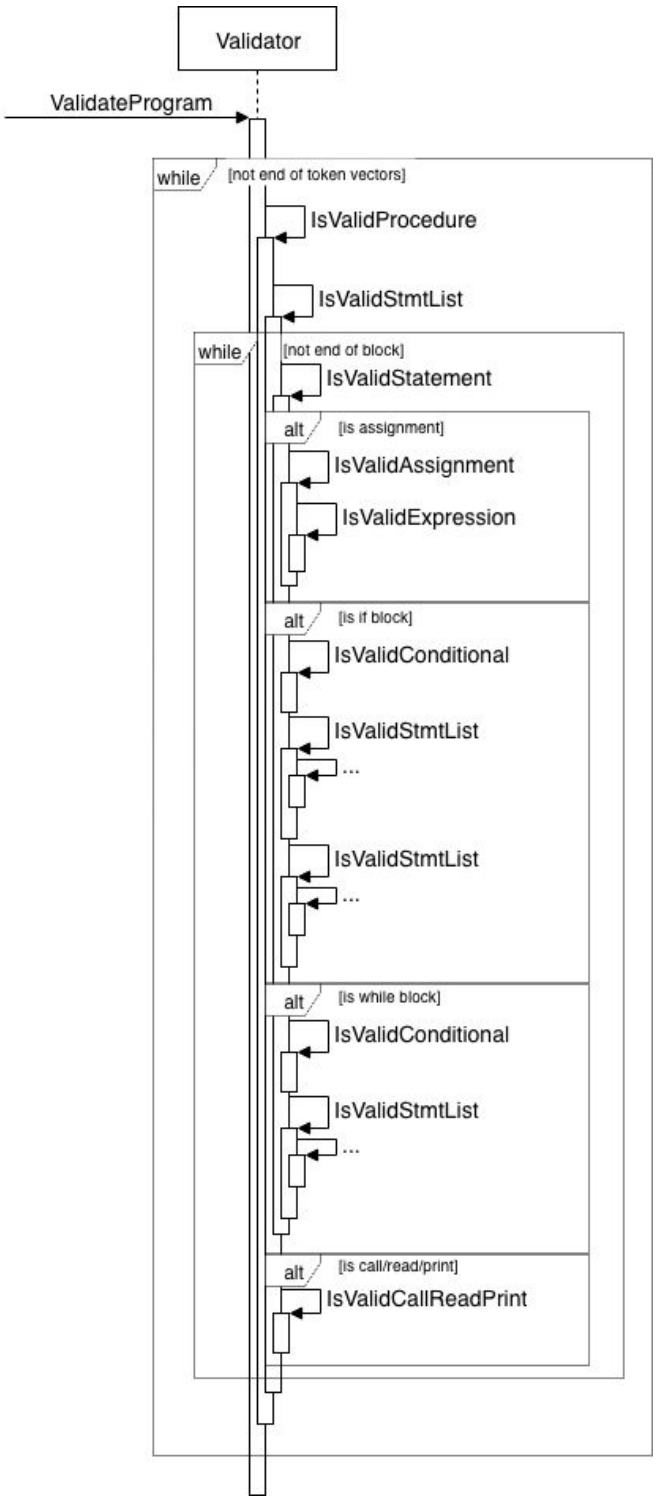
Design Decisions and Justifications

Recursive descent validation was chosen as it is much simpler to implement compared to other alternatives like LL[2] or LR validation, and is sufficient for the relatively simple (pun intended) syntax of SIMPLE.

Implementation

SIMPLE's Concrete Syntax Grammar (CSG) is followed closely to create a sort of sequence for the function to follow. For instance, `Validator::ValidateProgram` will call `Validator::IsValidProcedure` in a do...while loop to ensure program's CSG of `program: procedure+` is followed. A sequence diagram of `Validator` is displayed below to show how a program is validated.

[2] https://en.wikipedia.org/wiki/LL_parser

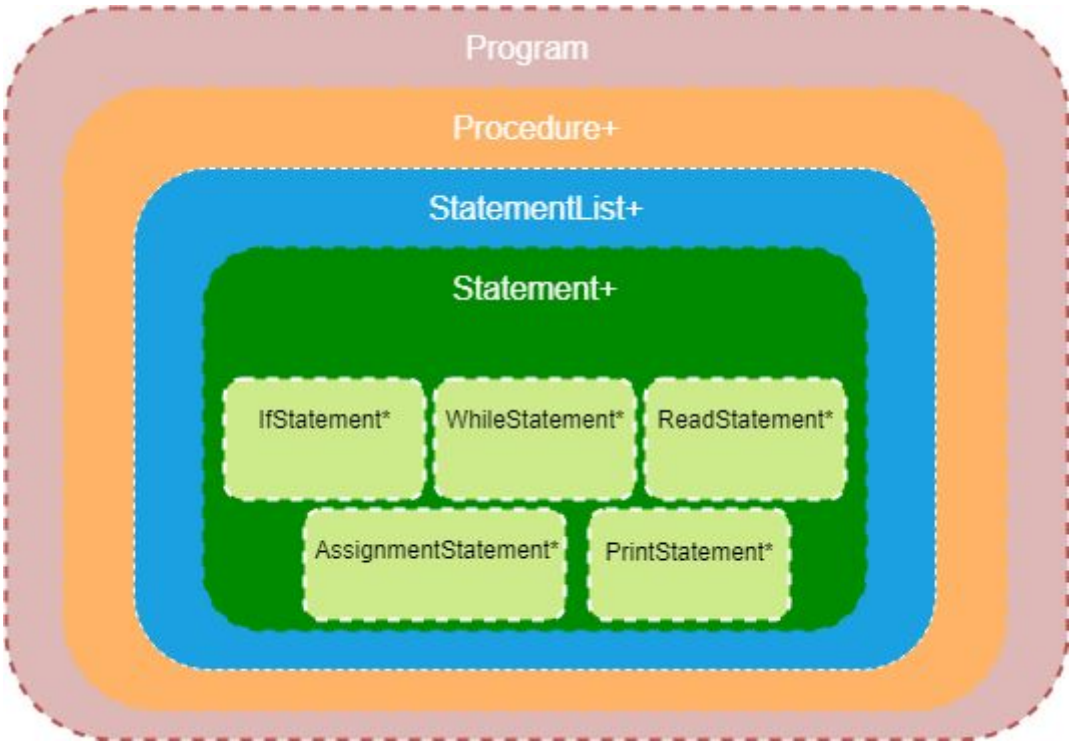


All of the validate functions return false if the current token in the token vector **does not** match the expected type or value, and will propagate upwards, causing `ValidateProgram` to return false to `Parser`, which will return an error message to the GUI and not proceed with parsing.

If the `Validator` successfully validates the program, the `Parser` will continue with the parsing.

Parser

The `Parser` is responsible for making sense out of the SIMPLE program that has been tokenized and validated. It iterates through the vector of `Token` structs in blocks, based on the block types. These blocks are processed as "procedure", "assignment", ‘if", "while", "read" and "print". Similar to the `Validator`, the `Parser` performs a recursive descent to parse the tokenized program. Information acquired during the parse is used for populating the PKB.



Design Decisions and Justifications

Design Decision 1: Parser population of design extractions into the PKB

Options

1. Parser to create an abstract syntax tree (a.k.a AST, commonly seen in compilers) and insert the AST into the PKB, allowing other components to extract information out of the AST.
2. Parser to insert design relations directly into the PKB while parsing without using an AST.

Criteria

The main criteria to consider when implementing the data structure is mainly the ease of implementation (Can the work be split up easily, and whether tests are able to be written easily?).

Comparison

Design Option	Option 1	Option 2
Ease of Implementation	- Hard to implement as trees are a fairly complex structure. Bugs may be hard to spot - Less reliant on PKB APIs as AST will be passed into PKB/Design Extractor for processing	- Easy to implement as the parser is able to determine all the design entities at time of parsing - Requires more calling of PKB APIs to update design relations

Decision

Based on the comparison between the designs, we have decided that the extra work that comes along with implementing additional complex data structure is not worth the benefits provided, and thus **Option 2** is chosen.

Design Decision 2: What relations should Parser handle?

Options

1. Parser handles all design relations
2. Parser handles relations that it can process while parsing, DesignExtractor handles relations that requires extracting of data from the PKB.

Criteria

The criterias to consider are:

1. Ease of implementation (Can the work be split up easily, and whether tests are able to be written easily?).
2. Modularity
3. Performance (Total time taken to update PKB completely)

Comparison

Design Option	Option 1	Option 2
Ease of Implementation	- Requires additional data structures to keep track of transitive relations such as ParentT - Complexity of Parser will be increased.	- Easier to implement as we do not need additional data structures to keep track of relations - Higher cohesion between classes also means tests are easier to be written
Modularity	- No modularity, as everything is performed by the Parser	- Slightly more modular, as the responsibility of populating complex relations is left to a new

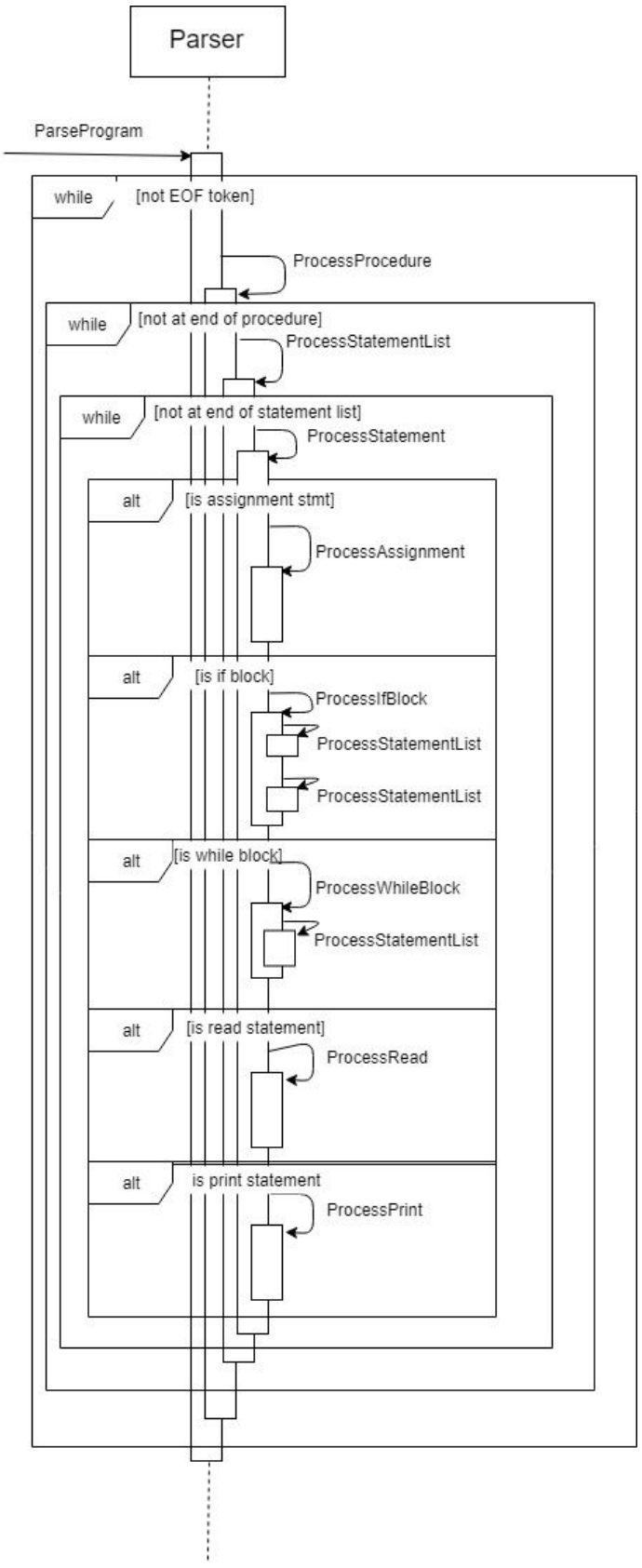
		DesignExtractor component - Coupling between the Parser and PKB is reduced
Performance (Time taken to update PKB)	- Depending on implementation, it might take the same time as option 2, as the parser will need to recursively go up the nesting or call stacks to update the relations	- Parser will notify PKB of the parse completion, and the DE will be called to extract the ParentT relation. - This might take longer (or shorter) than option 1 depending on implementation.

Decision

Design option 2 is a better option as we have to consider the size of the Parser for future iterations. There is still the consideration of **Next**, **Affects** and **Calls** relation to be added. Packing many responsibilities into a single Parser component could lead to testing difficulties and potential bugs. Therefore we decided that **Parent*** should be handled by `DesignExtractor` , as it does not create a significant disadvantage and also reduces the responsibility of the `Parser`.

Implementation

Similar to the `Validator`, the `Parser` adopts the recursive descent when `ParseProgram` is called. The following sequence diagram illustrates how the `Parser` recursively process the different procedures and statements:



Various functions are called at the right points during parsing in order to populate the PKB, as elaborated upon in the next few paragraphs:

General Behaviour

1. All modified/used variables or statement numbers are stored in `ParseData` at each method. Information is used by various methods in Parser to populate PKB of the relevant data.
2. The return value for all the methods below is `ParseData` .
3. For ProcessAssignmentStatement, ProcessIfStatement, ProcessWhileStatement, ProcessReadStatement and ProcessReadStatement methods, the VarTable and ConstantTable are populated accordingly

ProcessProcedure:

- `ProcessProcedure` calls `ProcessStatementList.`
- Populates PKB with **Modifies** and **Uses** relation for the current Procedure.

ProcessStatementList:

- At each iteration of `ProcessStatement` in this method, it consolidates all the `ParseData` belonging in this set of `StatementList` . Using the `ParseData`, `Parser` populates `PKB` with the **Follows** and **Follows*** relations.

ProcessStatement

- A statement is further determined to be either an `IfStatement`, `WhileStatement`, `ReadStatement` or `PrintStatement` using the `subtype` variable of the token.
- This method calls the relevant methods to process different statement types.

ProcessAssignmentStatement

- This method handle expressions, by calling the helper class `ExpressionHelper` to transform tokens in the *infix form* to the *postfix form.*
- These tokens in the *postfix form* are inserted into PKB to be used for pattern matching.
- Populates PKB with the **Modifies** relation of the current `AssignmentStatement`.

ProcessIfStatement

- `ProcessIfStatement` *further calls* `ProcessStatementList` *twice to parse the 'then' block and 'else' sub blocks.*
- Consolidate `ParseData` from both `ProcessStatementList`.
- Populate PKB with the **Parent, Modifies** and **Uses** relation of the current `IfStatement`.

ProcessWhileStatement

- `ProcessWhileStatement` *calls* `ProcessStatementList` once to process all statements in the while block.
- Consolidate `ParseData` from `ProcessStatementList`.
- Populate PKB with the **Parent, Modifies and Uses** relation of the current `WhileStatement`.

ProcessReadStatement

- Populate PKB with the **Modifies** relation of the current `ReadStatement`.

ProcessPrintStatement

- Populate PKB with the **Uses** relation of the current `PrintStatement`.

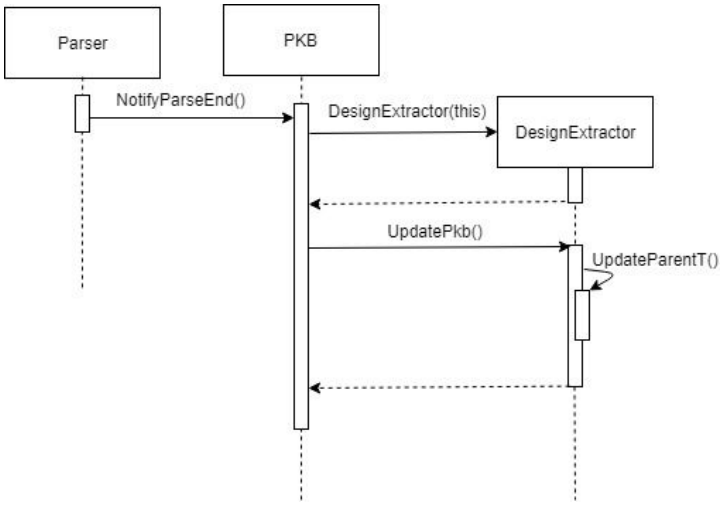
Design Extractor

As discussed above, the design extractor is responsible for populating design relations into the PKB that the Parser is unable to do in a single pass without adding complexity that was deemed unnecessary for the team. These design relations are notably Parent* relations for this iteration, and **Next*** relations for future iterations.

Implementation

Since we only have to implement a subset of design relations this iteration, the Design Extractor currently only populates the **Parent*** relations. It does so by recursively searching for all the children of the given statement, and then calling the `InsertParentT` method from the PKB to store the Parent* relationships.

A sequence diagram of what the `DesignExtractor` can do in iteration 1 is shown below:



As talked about above, future iterations will see the `DesignExtractor` take on more responsibilities, such as checking for cycle calls, populating the Next and Next* relations into the PKB, and so on.

PKB Design

Purpose and Principles of PKB

As explained in the SPA requirements, the purpose of PKB is to store design abstractions that are passed by the Frontend and provide those design abstractions with the correct criteria to the PQL, so that queries can be answered. Hence, our main principle in designing the PKB and its structure is **efficiency in storage and retrieval**. This section will explain how the PKB was designed to follow this principle.

Implementation

The PKB consists of two different types of tables for storing data, namely:

- Basic entities storage
- Relationships storage

The Basic Entities Storage stores the basic entities of a SIMPLE program, such as the Procedure, Statements, Variables, and Constants. For Iteration 1, there are a total of 6 structures: `ProcList`, `StmtTypeList`, `StmtListTable`, `StmtTable`, `VarList`, `ConstList`.

The Relationship Storage stores all relationships between the basic elements. For Iteration 1, there are four classes for this: `ParentTable`, `FollowsTable`, `UsesTable` and `ModifiesTable`.

Figure PKB-1 below illustrates the structure of the PKB:

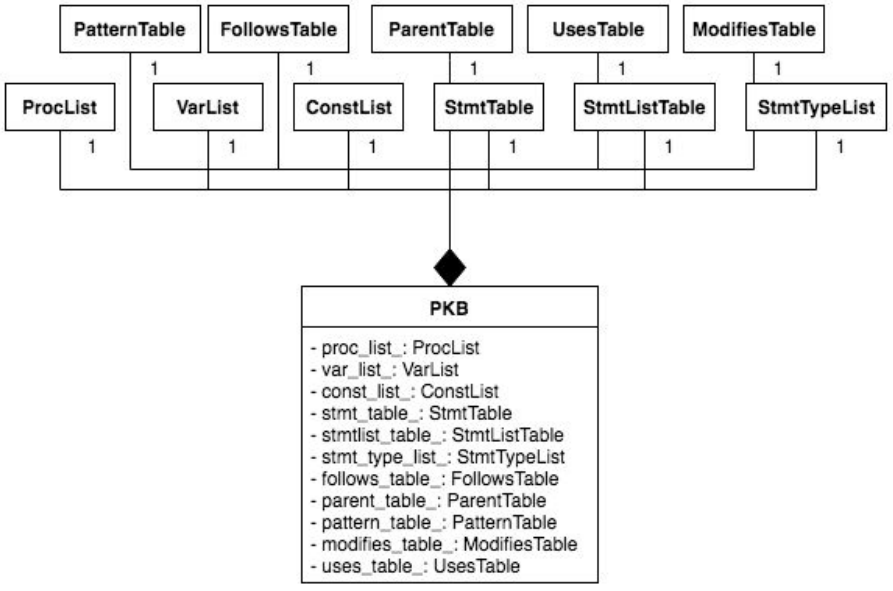


Figure PKB-1. Brief Class Diagram of PKB
The member variables and methods for the associated classes are omitted. PKB class contains public methods which are also omitted on this diagram for simplicity.

Below is a sample class illustration of one relationship storage, the ParentTable. This figure is merely given to explain how the PKB is structured. For the full abstract API, refer to **Documentation of Abstract APIs**.

Figure PKB-2: Sample Relationship Storage Structure

Storage Operations

How PKB Stores Basic Entities

This section explains how the PKB stores ‘basic entities’ that are passed from the Parser. Basic entities refer to raw elements that make up a SIMPLE programme, including:

- Statements
- Variables
- Procedure(s)
- Constants

These entities are stored in their respective storage classes, as follows:

Statements	StatementTable: Stores all statements and their statement list index and type StatementListTable: Stores the list of statements for every statement list index StatementTypeList: Stores the list of statements for every statement type (e.g., While)
Variables	VarList
Procedures	ProcList

Constants	ConstList

These basic entity storage classes allow the PQL to fetch a list of all entities in the programme, such as `GetAllVar()` and `GetAllConst()`.

How PKB Stores Design Abstractions

This section explains how the PKB stores 'design abstractions' that are passed from the Parser. Design abstractions refer to relationships that are identified from the SIMPLE programme, including but not limited to:

- Follows/* relationships
- Parent/* relationships
- UsesS/P relationships*
- ModifiesS/P relationships*

The PKB provides very basic operations that allow the Parser to insert such relationships. This is to follow the principle that the PKB should not be involved in any logic or computations. Hence, the PKB allows the Parser to insert relationships between two entities, such as:

```
void InsertModifiesP(ProcName modifying_proc, VarName modified_var)
```

which exposes the ModifiesTable to the Parser.

This method has benefits and shortcomings. The benefit of this method is that the PKB is free from all logical computations. The PKB does not need to identify any relationships from statements. However, one small shortcoming is that insertion to all relationship storage classes (such as ModifiesTable and ParentTable) are directly exposed to the Parser.

However, this poses little problems, as the Parser does not need to do any extra processing, as the parameters are already known to the Parser at the time of populating the PKB.

Fetch Operations

Returning Design Abstractions to PQL

This section explains how the PKB returns information necessary for the PQL to evaluate queries.

The PKB provides public methods that can be invoked by the PQL to fetch a list of entities, single entity, or a boolean value. Assume the following examples of some Uses relationships that the PQL might encounter. The table shows what methods the PQL can call from the PKB to obtain the entities it needs to resolve a query.

statement s; variable v; procedure p;			
Uses Relationship	**What questions might the PQL need to answer?**	**What methods can be used?**	**What do(es) the method(s) return?**
Uses(s, v)	What statements use any variable? What variables are used by any statement?	GetAllUsesPairS()	List of <statement number, variable> pairs
Uses(s, "y")	What statement s uses the variable "y"?	GetUsingStmt("y")	List of statement numbers
Uses(s,_)	What statements use any variable?	GetAllUsingStmt()	List of statement numbers
Uses(p, v)	What procedures use any variable? What variables are used by any procedure?	GetAllUsesPairP()	List of <procedure name, variable> pairs
Uses(p, "z")	What procedure "p" uses the variable "z"?	GetUsingProc("z")	List of procedure names
Uses(p, _)	What procedures use any variable?	GetAllUsingProc()	List of procedure names
Uses(1,v)	What variable "v" is used by statement number 1?	GetUsedVarS(1)	List of variables
Uses(1, "a")	Is variable "a" used by statement number 1?	IsUsesS(1, "a")	Boolean (True / False)
Uses(1, _)	Does statement number 1 use any variable?	GetUsedVarS(1)	List of variables - check whether it is empty
Uses("ABC", v)	What variable "v" is used by the procedure "ABC"?	GetUsedVarP("ABC")	List of variables
Uses("ABC", "z")	Is variable "z" used by procedure "ABC"?	IsUsesP("ABC", "z")	Boolean (True / False)
Uses("ABC", _)	Does procedure ABC use any variable?	GetUsedVarP("ABC")	List of variables - check whether it is empty

For a full list of these methods, refer to **Documentation of Abstract APIs**.

Further details on how the populated information is used will be explained in the PQL section.

Design Decisions and Justifications

Design Decision 1: PKB Structure

Problem

The PKB consists of several storage modules that cater to each relationship that is identified from the SIMPLE programme.

Options

1. Instantiate each map / list for each relationship storage in PKB class.
2. Instantiate each map / list for each relationship storage in their respective classes. Then, the PKB can simply instantiate each class. For instance, there will be a ParentTable class which instantiates the ParentMap, ChildMap et cetera. The PKB class will create an instance of ParentTable.

Criteria

1. When the PKB has more relationships to store, how easy is it to extend the PKB?

Decision

We chose **Option 2**. Option 2 allows the PKB to adhere to the **Open-Closed Principle**, since an addition of a new relationship storage is simply an instantiation of a new class without having to add multiple member variables and methods to the PKB.

Design Decision 2: Storing Relationships

Problem

PQL will require various forms of data from the PKB. The operation for each query should be as efficient as possible. This applies to all four Relationship Storage tables that store Parent, Follows, Uses, and Modifies relationships.

Options

1. **Single Storage:** Store a single storage, eg. a follows map that stores <follower, list of followees> pairs
2. **Joint Single Storage:** Store a joint single storage, e.g., a single follows table that has pairs of <follower, list of followees> and <followee, list of followers pairs>
3. **Inverse Double Storage:** Store an inverse of relationship storages, e.g., a follows map that stores <follower, list of followees> and a followed by map that stores <followee, list of follower pairs>

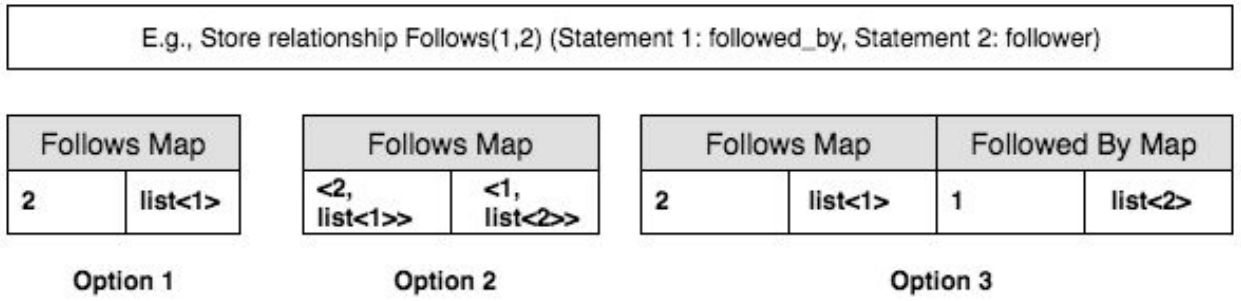


Figure PKB-3. Graphic depiction of three storage options

Criteria

1. Efficiency and complexity in retrieval: How fast is it to retrieve information from the relationship storage for the PKB?
2. Ease of testing: How easy is it to conduct modular testing, i.e., to thoroughly test every part of the data structure? It is important to take this criteria into account, since the relationship storage is central to the PKB and it is crucial that it is tested thoroughly.
3. Ease of implementation ($\propto 1/time\ taken\ to\ implement$)

Comparison & Decision

After comparison, we decided on **Option 3: Inverse Double Storage**. This is because the three options do not differ significantly in terms of ease of testing or implementation. And Option 3 has the highest efficiency in retrieving information. The PQL requires a variety of combinations of relationships from the PKB, hence it is best to store an inverse copy of every relationship map that is stored.

Design Decision 3: Dual Storage or Single Storage?

Problem

PQL requires several formats of information from the PKB, one of which is when it needs a list of entities. To illustrate this example more clearly, let us examine the following two relationships:

statement s, s1; variable v; procedure p;

RELATIONSHIP 1 Parent (s, s1)
RELATIONSHIP 2 Modifies (p , v)

When a query contains **Relationship 1**, the PQL may require "a list of statements that are parent to *some* other statement." When a query contains **Relationship 2**, the PQL may require "a list of procedures that modify *some* variable v."

As such, PQL may require **a list of entities that have a relationship with *some* other entity**, regardless of what that 'other entity' is.

Options

1. **Single Storage:** Store a single storage for all relationships, as decided in Design Decision 2.

2. **Dual Storage:** Store two copies of storage for all relationships; one in list form and one as decided in Design Decision 2.

Since the two options are rather abstract, see Figure PKB-4 for a graphic depiction of the options:

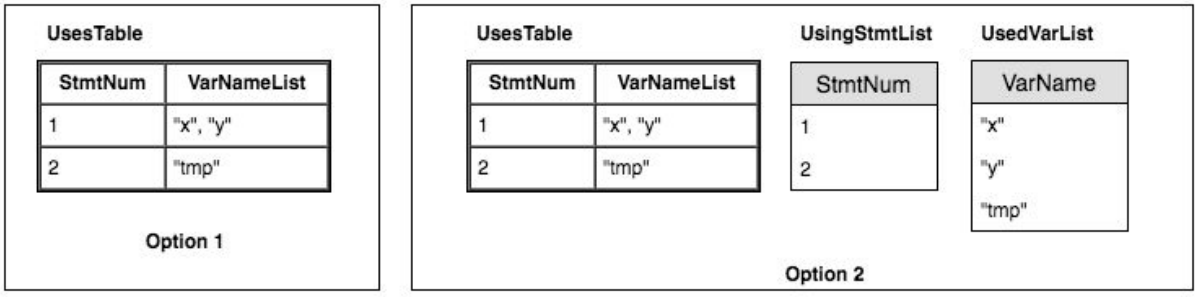


Figure PKB-4. Graphic depiction of Option 1 and Option 2

As seen from Figure PKB-4, while Option 1 keeps a copy of one relationship table, Option 2 maintains two extra lists containing one type of entity. In Option 2, if the PQL wants a '*list of statement numbers that use some variable'*, the PKB can simply return the UsingStmtList.

Criteria

1. **Low complexity in retrieval:** How easy is it to retrieve a list of entities who are parents / follows / uses / modifies another entity, and vice versa?
2. **Ease of implementation:** How much time does it take to implement the data structure?

We considered Criteria 2 as one of our criteria, since, at the time of the implementation, it was crucial that the PKB finished its classes as fast as possible. Adhering to the principle of PKB, we prioritize Criteria 1.

Comparison

Design Option	Option 1	Option 2
Low complexity in retrieval	- Slow for retrieving certain entities. For example, it needs to traverse the UsesTable to generate a list of variables that are used in some statement	- Fast retrieval for all operations
Ease of Implementation	- Simple	- More data structures to implement and manage

Decision

We chose **Option 2: Store both list and table forms**, because the lower complexity obviously outweighs the slight difference in ease of implementation.

Design Decision 4: Double Storage - Vector + Unordered_set

Problem

Carrying on from Design Decision 3, another decision to make is how to configure the double storage.

For the storage of a list of entities, PQL usually needs the PKB to:

- Return the whole list (as a vector). E.g. `GetAllVar()`.
- Check whether a certain value is inside the list. E.g. `IsParent(5)`.

PKB needs to ensure fast retrieval for both situations.

Options

1. Store the list as a vector
2. Store the list as an unordered_set
3. Store two copies of the list, vector + unordered_set

Criteria

1. Speed of returning the whole list as a vector
2. Speed of finding an element in the list
3. Speed of insertion

Comparison

Option 1: Vector guarantees that the returning the whole list is O(1) since no conversion is needed. But finding an element is O(n).
Option 2: Unordered Set gives average O(1) lookup time since unordered_set is implemented as a hash table. But returning the whole list requires inserting all elements in the unordered_set to a vector, which is O(n).
Option 3: Vector and Unordered Set can use unordered_set for O(1) lookup and vector for O(1) return of the whole list. But insertion time is the sum of Option 1 and 2.

Decision

We chose **Option 3** because it has the best performance for both situations described in the problem. Although it has the slowest insertion time, time taken for insertion does not contribute to the query evaluation time, hence it is not a priority here.

Design Decision 5: Which Data Structure(s) should PKB use?

Problem

All the previous design decisions come to the issue of, which data structure is best suit for PKB's needs.

In the PKB, many relationships have to be stored in *1:many ("1-to-many")* form. For instance, a statement might have multiple children, and a procedure may use multiple variables. This introduces a need for a storage with two levels. Hence, one crucial design decision for the PKB is to decide on a suitable data structure that allows to store multiple relationships.

Options

1. `unordered_map` with <key, value> pairs of `<string, vector<string>>`
2. List within a list, `<list<list<string>>`
3. Vector within a vector, `<vector<vector<string>>`
4. `map` with <key, value> pairs of `<string, list<string>>`

Criteria

- **Low complexity of retrieval:** How fast is it to retrieve an entity with a fixed index or key? Note that the complexity of storage is not an issue here, as it does not contribute to query evaluation time.

Comparison

- Comparing an unordered map with a map, the former has a faster retrieval and storage complexity as it is implemented as a hashmap. The latter is implemented as a tree, thus introducing a greater complexity in both insertion and search.
- Comparing an unordered map with a list, the latter is implemented as a Linked List in C++. Hence, the complexity for finding the key alone is O(n). Finding and inserting a value would be $O(n^2)$ which is a complexity we cannot afford for such a frequent operation.
- A vector is generally a better choice in C++, since the list data structure is a Linked List while a vector is an indexed contiguous list, allowing for quicker retrieval if the size of the element is small (> 32 bytes, as discussed in the front end section). Hence, use a vector within the unordered_map for optimized speed.

Decision

Based on the above comparison, we decided to use **Option 1** - unordered maps with values as vectors of strings (`unordered_map<string, vector<string>>`).

Complications & Mitigation

The element retrieval for unordered_map (using the [ ] operator) inserts an empty key to the map if there is no value associated with the key.

Hence, if there is no <key, value> pair in the map, there will be an empty <stmt_num> key inserted into the unordered_map. This can produce inaccurate results for functions such as `HasFollowsRelationships()` (checking whether the `FollowsMap` is empty). To mitigate this, we used the `find()` method to check for existing keys.

Design Decision 6: Pattern Storage

Problem

PKB needs to store assign patterns in a way that ensures fast retrieval of assign statements (and the corresponding modified variable) that match a certain pattern (partially or exactly).

Options

1. Store the **right-hand-side expression** (as a TokenList) for every assign statement. Check through all the expressions when PKB needs to find assign statements that contain a certain (exact or sub) expression.
2. Generate **all possible sub-expressions** for every assign statement. Store them using an unordered_map with sub-expression (as a string in postfix format) as keys and statement numbers as values. Store exact-expressions in a similar manner.

Criteria

1. Speed of retrieving assign statements that contain a certain (exact or sub) expression.
2. Ease of extending to while/if pattern.

Comparison

It is obvious that Option 1 takes O(n) time to retrieve the answer for a pattern clause whereas Option 2 only takes O(1) time. Ease of extending to while/if pattern is similar for both options because both require a separate table for while/if.

Decision

We chose Option 2. More specifically, we created the following four unordered maps in PatternTable class. The diagram illustrates what they store after processing these two assign statements inserted by the Parser:

1. `x = (3 + a) / b;`
2. `y = 3;`

StmtVarMap	
StmtNum = Key	VarName = Value
1	"x"
2	"y"

VarStmtMap	
VarName = Key	StmtNum = Value
"x"	1
"y"	2

ExactExprMap	
Expr = Key	StmtNum = Value
"3 a + b /"	1
"3"	2

SubExprMap	
Expr = Key	StmtNum = Value
"3"	1, 2
"a"	1
"3 a +"	1
"b"	1
"3 a + b /"	1

Figure PKB-5: Pattern storage

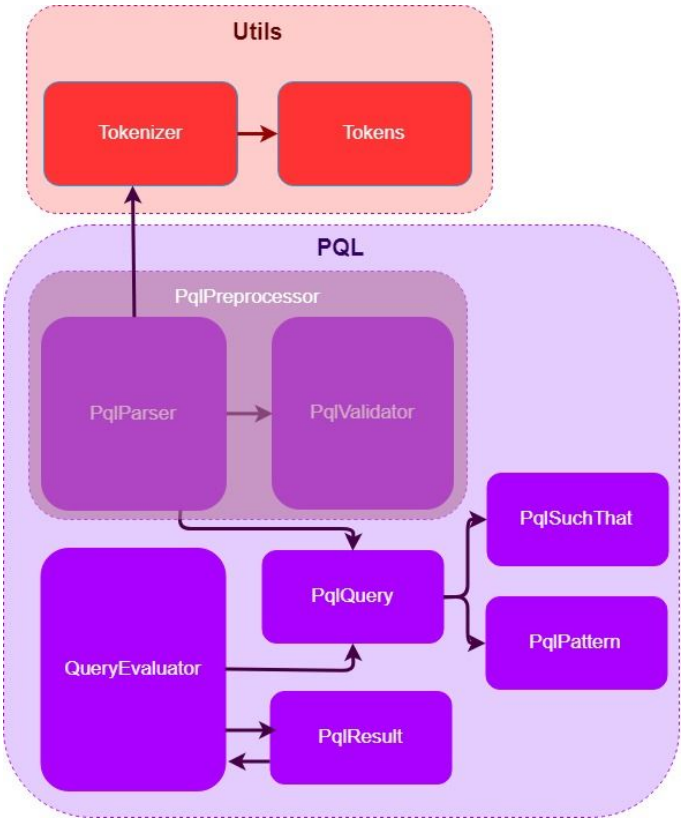
StmtVarMap: Fast retrieval of the left hand side variable of an assign statement. Used when generating all pairs of <a, v> that contain a certain expression.
VarStmtMap: Fast retrieval of all assign statements that contain the given variable as the left hand side variable.

ExactExprMap: Fast retrieval of all assign statements that contain the given expression on the right hand side.
SubExprMap: Fast retrieval of all assign statements that contain the given expression as a sub-expression on the right hand side.

These four unordered maps ensure that PKB can give results regarding any pattern query clause in an efficient way.

PQL Design

To recap, the UML Diagram of our PQL components can be seen below:



The **PQL** component is responsible for receiving user query input, process it, and output the result. The `PqlPreprocessor` sub-component deals with the the parsing and validation of the PQL input. It utilizes the Tokenizer to tokenize the input into tokens (such as name, digits, symbols) for parsing. During parsing, relevant information of the query is stored in the internal data structure `PqlQuery`, which contains a list of `PqlSuchthat` and `PqlPattern` instances. The populated `PqlQuery` object is then passed into the `QueryEvaluator` to evaluate against the **PKB**, in which results are stored in `PqlResult` ready to be displayed.

Design Decisions and Justifications

Design Decision 1: Data Structure of PqlQuery object:

Problem

The query preprocessor will need a way to store the query obtained after the `PqlParser` has finished parsing the user query.

Options

1. Store the query in a query tree
2. Store the query as a collection of lists in one class

Criteria

The main criteria to consider when implementing the PqlQuery object is mainly the ease of implementation (Can the work be split up easily and whether tests are able to be written easily), and whether it allows ordering of clauses so optimizations can be performed.

Comparison

Design Option	Option 1	Option 2
Ease of implementation	- Harder to implement due to tree structure complexity. - Less work in PqlEvaluator as the clauses and constraints are already stored in the tree	- Easy implementation for PqlQuery, as the clauses can just be added into a list of clauses. - Might require additional logic in PqlEvaluator to extract clauses and constraints
Allows sorting of clauses for optimization	- Additional complexity has to be introduced to sort the clauses in a tree form	- List objects can easily be sorted

Decision

Based on the comparison between the designs, we chose to implement **Option 2**, as we decided that the extra work that comes along with implementing complex tree data structure is not worth the benefits provided. Option 2 also allows easier sorting of the clauses compared to option 1.

Design Decision 2: Data Structure of clauses

Problem

The query preprocessor will need a way to store the clauses of the query obtained after the `PqlParser` has finished parsing the user query.

Options

1. Store different clauses as an object by itself
2. Store all clauses as the same object

Criteria

The main criteria to consider is the ease of implementation (Can the work be split up easily and whether tests are able to be written easily)

Comparison

Design Option	Option 1	Option 2
Ease of implementation	- Very easy to implement and test	- Require more work to implement a single object to store many

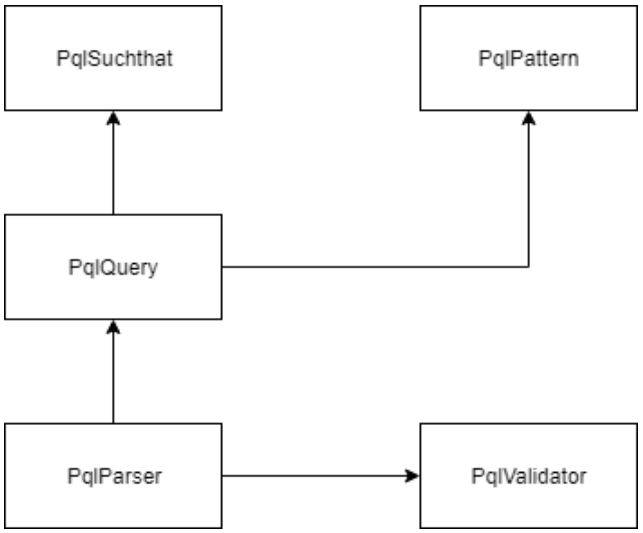
		different clauses

Decision

Based on the comparison between the designs, we have decided that we should work on the clauses as individual objects for now as we need to have the system up and running and handle other more important functions and components first. Therefore, we think that **option 1** is the better option for **this** iteration. However, we recognize that this is not ideal and might see a refactoring of this component in future iterations.

Query Preprocessor

The query preprocessor consist of the data structures to store the query, as well as parsing and validation algorithms. The following class diagram below depicts the relationship between the classes that form the preprocessor:



PqlQuery

This is the core class that stores all the relevant information for a query. It contains the declarations made, the selection clause, such that clauses, and pattern clauses. The validation rules of each of the clauses is also stored here (more information below). An instance of this class can be passed into the parser to populate the data, and into the evaluator to query the PKB.

PqlSuchthat

This class stores information of a such that clause, such as the type (Follows, Parent, Modifies, Uses, etc.) and the first and second parameters.

PqlPattern

Similar to `PqlSuchthat`, this class stores the information of a pattern clause, such as the type (assign, while, or if), and the first parameter. If it is an assign type, the expression in the second parameter will be stored as a post-fix token list (more information in `PqlParser`). For

while type, as specified by the grammar list, the second parameter is always "_" and hence is not necessary to store. Similarly, for if type, the second and third parameters are always "_" and thus will not be stored either.

PqlValidator

This is a helper class that helps to validate if an input is IDENT, integer, or an expression. For validation of IDENT and integer, regex is used.

PqlParser

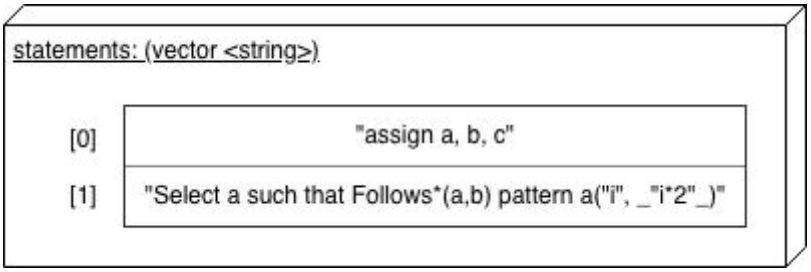
Implementation

Start of parsing

Let us take the following example query to explain how parsing is done.

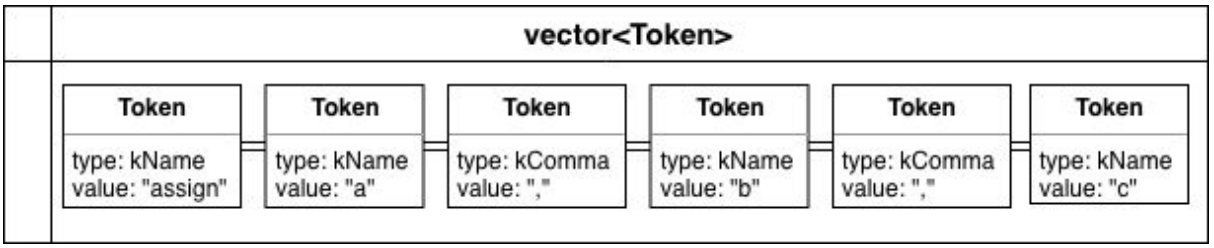
```
assign a, b, c; Select a such that Follows*(a, b) pattern a(“i”,_”i*2”_)
```

The entire query will first be split by *semicolons*, hence splitting up the declaration statement(s) and the select statement:



Parsing a statement

Each statement is then tokenized into individuals tokens of NAME, DIGITS, and other symbols (comma, parenthesis, etc.) using the `Tokenizer` class shown at the beginning of this document. For example, `statements[0]` will be tokenized to:



The list of tokens is then parsed using recursive descent. As stated before in the front end section, recursive descent is much simpler to implement compared to its alternatives.

Parsing a declaration

assign a, b, c; Select a such that Follows*(a, b) pattern a(“i”,_”i*2”_)

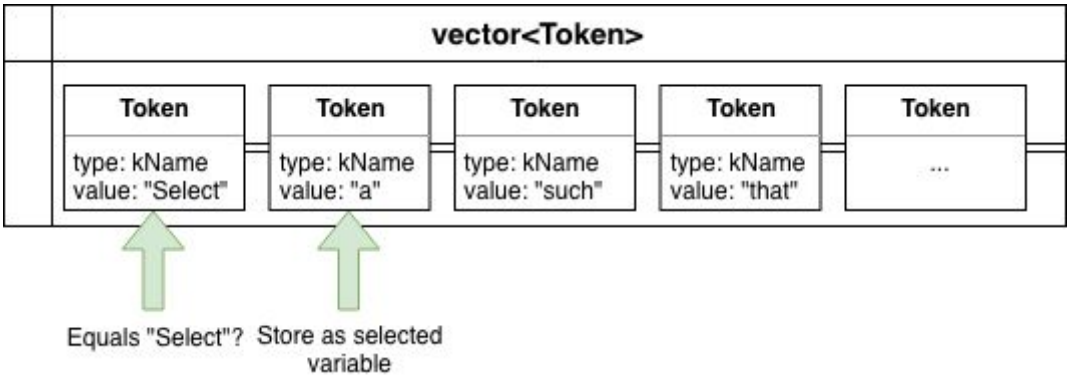
The first token will be validated with a list of possible design-entity types, such as assign, variable, prog_line etc. Subsequent tokens, the synonyms, will be associated with the design-entity type and be stored in a map that maps the synonym to the type. In this example, the map will look like this:

Declarations	
key	**value**
"a"	assign
"b"	assign
"c"	assign

Parsing the select clause

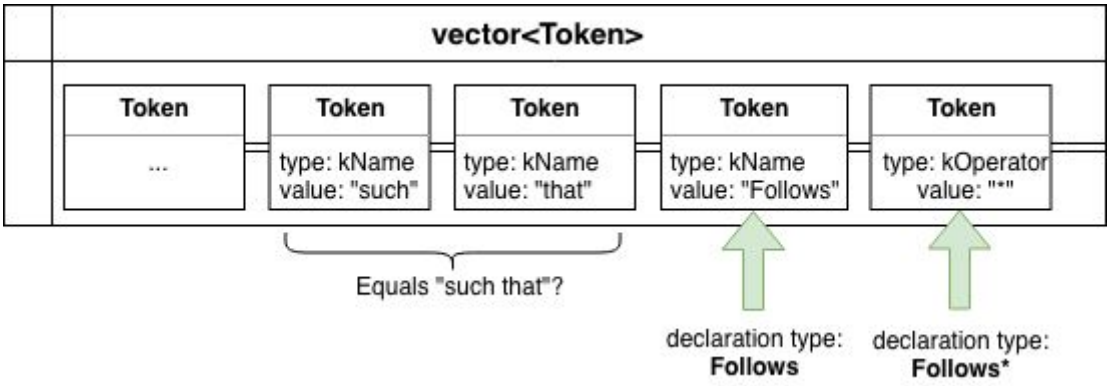
assign a, b, c; Select a such that Follows*(a, b) pattern a(“i”,_”i*2”_)

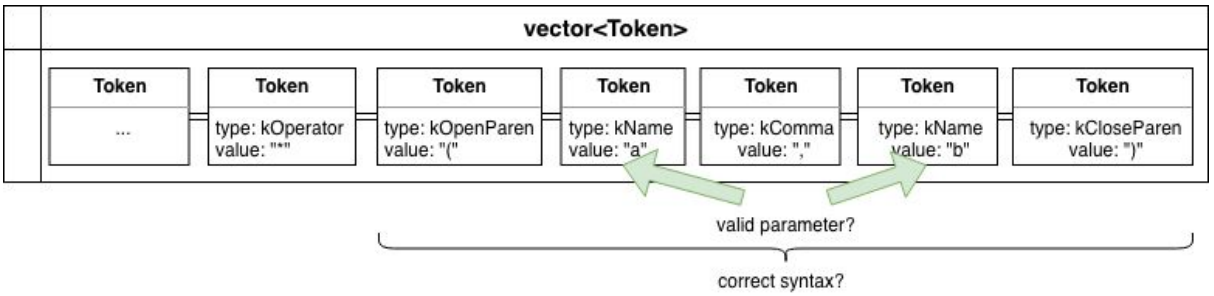
Firstly, the first token is validated to ensure that it is "Select". The following token is then stored as the selected variable. Next, the algorithm looks out for such that, pattern, or with clauses and parses them accordingly.



Parsing a "such that" clause

assign a, b, c; Select a such that Follows*(a, b) pattern a(“i”,_”i*2”_)





Firstly, ‘such that’ is validated to ensure that it is a such that clause. Then, the type is validated with a list of such that types (relationships), such as Follows, Parent*, Uses, etc. Next, the parameters are validated with a predetermined table of possible parameters based on each relationship type, using table driven parsing as follows:

Type	**First parameter**	**Second parameter**
Follows	synonym (stmt, read, print, call, while, if, assign, prog_line), integer, "_"	synonym (stmt, read, print, call, while, if, assign, prog_line), integer, "_"
FollowsT (Follows*)	synonym (stmt, read, print, call, while, if, assign, prog_line), integer, "_"	synonym (stmt, read, print, call, while, if, assign, prog_line), integer, "_"
Parent	synonym (stmt, while, if, prog_line), integer, "_"	synonym (stmt, read, print, call, while, if, assign, prog_line), integer, "_"
ParentT (Parent*)	synonym (stmt, while, if, prog_line), integer, "_"	synonym (stmt, read, print, call, while, if, assign, prog_line), integer, "_"
UsesS	synonym (stmt, print, call, while, if, assign, prog_line), integer	synonym (variable), ident, "_"
UsesP	synonym (procedure), ident	synonym (variable), ident, "_"

ModifiesS	synonym (stmt, read, call, while, if, assign, prog_line), integer	synonym (variable), ident, "_"
ModifiesP	synonym (procedure), ident	synonym (variable), ident, "_"

Once all validation has passed, a `PqlSuchthat` object is created and stored in the `PqlQuery` object.

Parsing a "pattern" clause

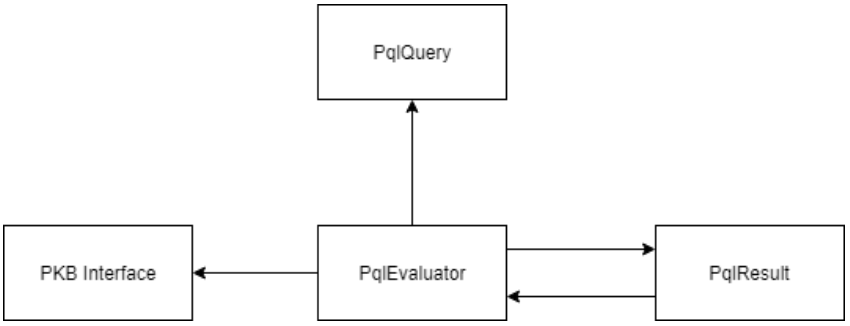
```
assign a, b, c; Select a such that Follows*(a, b) pattern a(“i”,_”i*2”_)
```

Similar to parsing a "such that" clause, the first token is first validated to ensure it is a pattern clause. Then the next token is checked with the declaration map (initialized previously) to validate if it has been declared previously, and check its type (assign, while, or if). The parameters are then validated against a predetermined validation table as follows:

Pattern type	First parameter	Second parameter	Third parameter
assign	Synonym (variable), "_", ident	expression	
while	Synonym (variable, constant), "_", ident	"_"	
if	Synonym (variable, constant), "_", ident	"_"	"_"

Once all validation has passed, a `PqlPattern` object is created and stored in the `PqlQuery` object.

Query Evaluator



PqlEvaluator

The `PqlEvaluator` is designed to take in the `PqlQuery` object and retrieve results from the PKB based on the data extracted from `PqlQuery`. Then, the evaluator will either store the

result in a `PqlResult` class or immediately return it to the user, depending on whether there were any clauses like a "such that" clause or a "pattern" clause.

Design Decisions and Justifications

Storing results of multiple clauses:

Problem

The query evaluator will need a way to store the results obtained from multiple clauses in order to return a final result that is constrained by the clauses.

Options

1) Store the results in the `PqlQuery` object
2) Store the result in the `PqlEvaluator`
3) Store the result in a new object called `PqlResult`

Criteria

Modularity of code base. It is important to modularise the code as it improves readability of code and the code base will be easier to maintain.

Comparison

Design Option	Option 1	Option 2	Option 3
Modularity	- Storing the result in the query violates the Separation of Concerns principle	- Storing the result in evaluator also violates the Separation of Concerns principle	- This option distinguishes concerns and adds modularity

Decision

Based on the criteria, we have decided to store the results of the clauses in a new object PqlResult. This will ensure that the code adheres to the principle of Separation of Concerns which improves code modularity and readability.

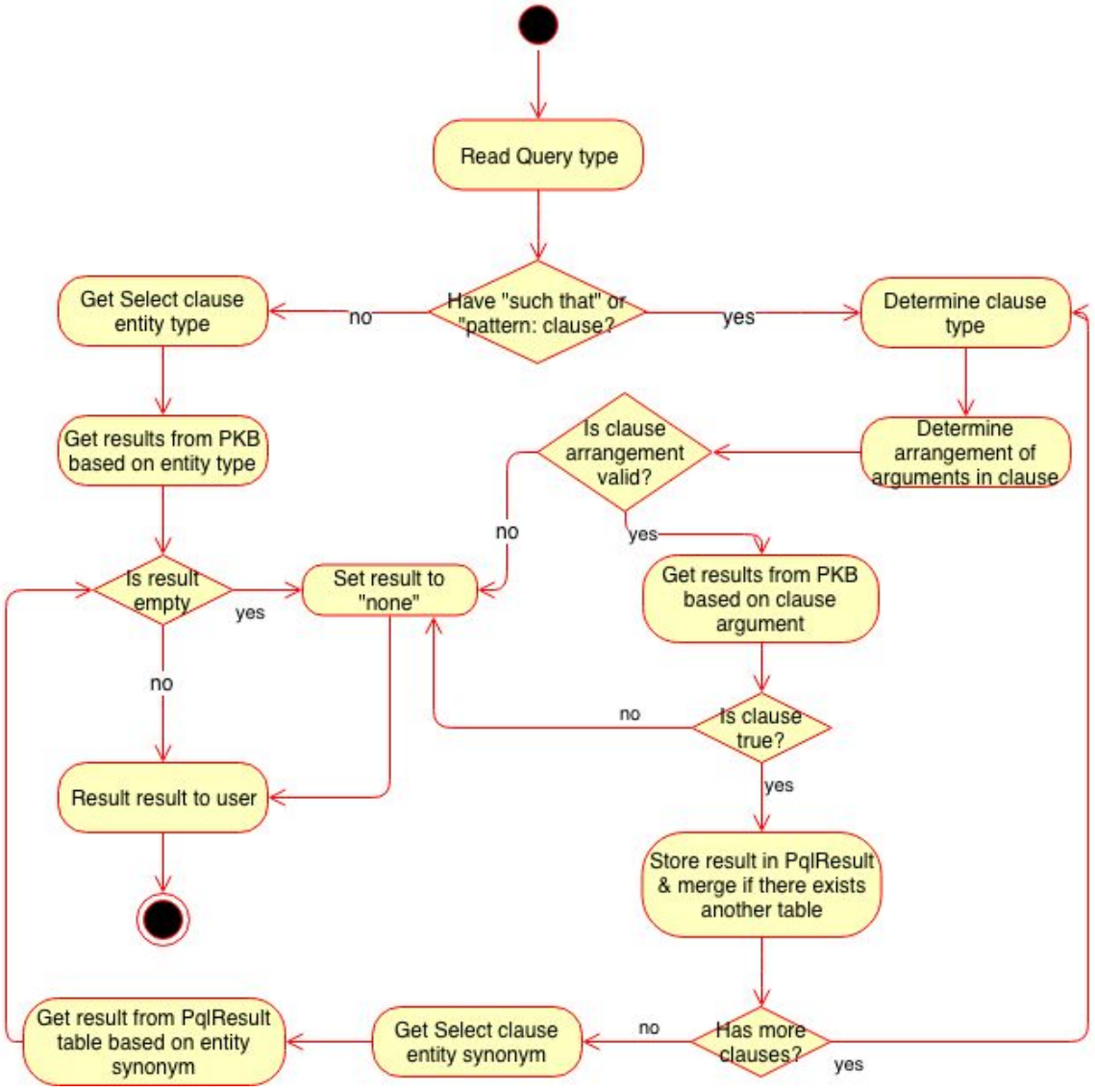
Implementation

A general flow of how the evaluator works is as follows:

1) Evaluator will first check if there are any "such that" or "pattern" clause in the given `PqlQuery`.
2) If none of the above two clauses exist, the Select clause will be evaluated and returned
3) Else, the arrangement of arguments in the clause will be identified. (e.g for the clause "such that Follows**(1,2)**", the evaluator will determine that the parameters are of the type `(INTEGER, INTEGER)`)

4) Evaluator will retrieve the relevant results from the PKB, and access if the clause is true or false (if no results are retrieved, the clause is false)
5) If the clause(s) are true, the Select clause will be evaluated and result returned.

Below is the action diagram which summarises what the evaluator do:



PqlResult

The PqlResult is designed to take in the list of results obtained from the PKB by the evaluator and then storing it in a "result table" for later use.

Implementation

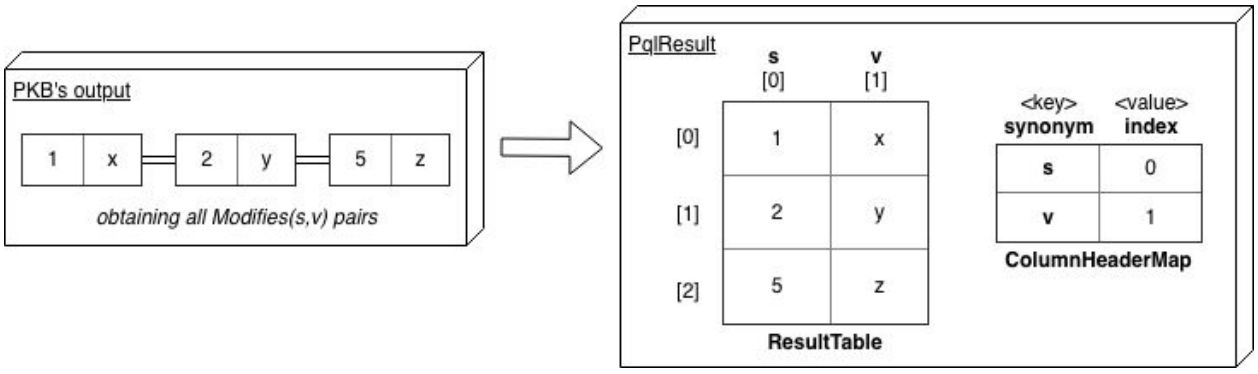
Data Structure

The data structure of this table is in the form of a '2D" vector of vectors containing strings (`vector<vector<string>>`).

The 2D vector data structure was chosen because it maintains its sequence of insertion and retrieval of elements we want is O(1) using indices. There will also be an `unordered_map` to keep track of the column value with reference to each synonym. The unordered map data

structure was chosen due to its O(1) retrieval and insertion, which will be useful to quickly obtain the index of the column that corresponds to the desired synonym (for comparison and merging). Furthermore, an unordered map can also be used as the order of the column header does not matter for our desired operations.

For example, after evaluating a "such that Modifies(s, v)" clause and retrieving a list of result pair from PKB "(1, x), (2, y), (5, z)", it will be stored in `PqlResult` as such:



The outer vector (`vector<vector<string>>`) element will store the entire row while the inner vector (`vector<vector<string>>`) element will store the column values as string. In this example there will be three outer vector elements (Rows) and two inner vector elements (Columns) for every row.

From the above diagram, you will see the unordered_map `ColumnHeaderMap` with its use explained previously to keep track of which column index refers to which synonym. The **key** is the synonym and the **value** is the column index of the synonym in the `ResultTable` data structure.

Storing result from multiple clauses

The `PqlResult` class is designed to store result from all the clauses, but will only store a single `ResultTable`. If there are more than 1 clause and a second clause is evaluated, `PqlResult` will merge the current `ResultTable` and the new `ResultTable` together by calling a helper function `PqlResult::MergeResults`, and discard the old tables as they are unneeded after merging. Since each clause is evaluated individually and each clause returns a result table with maximum of **two** columns, we only need to think about 3 cases of merging:

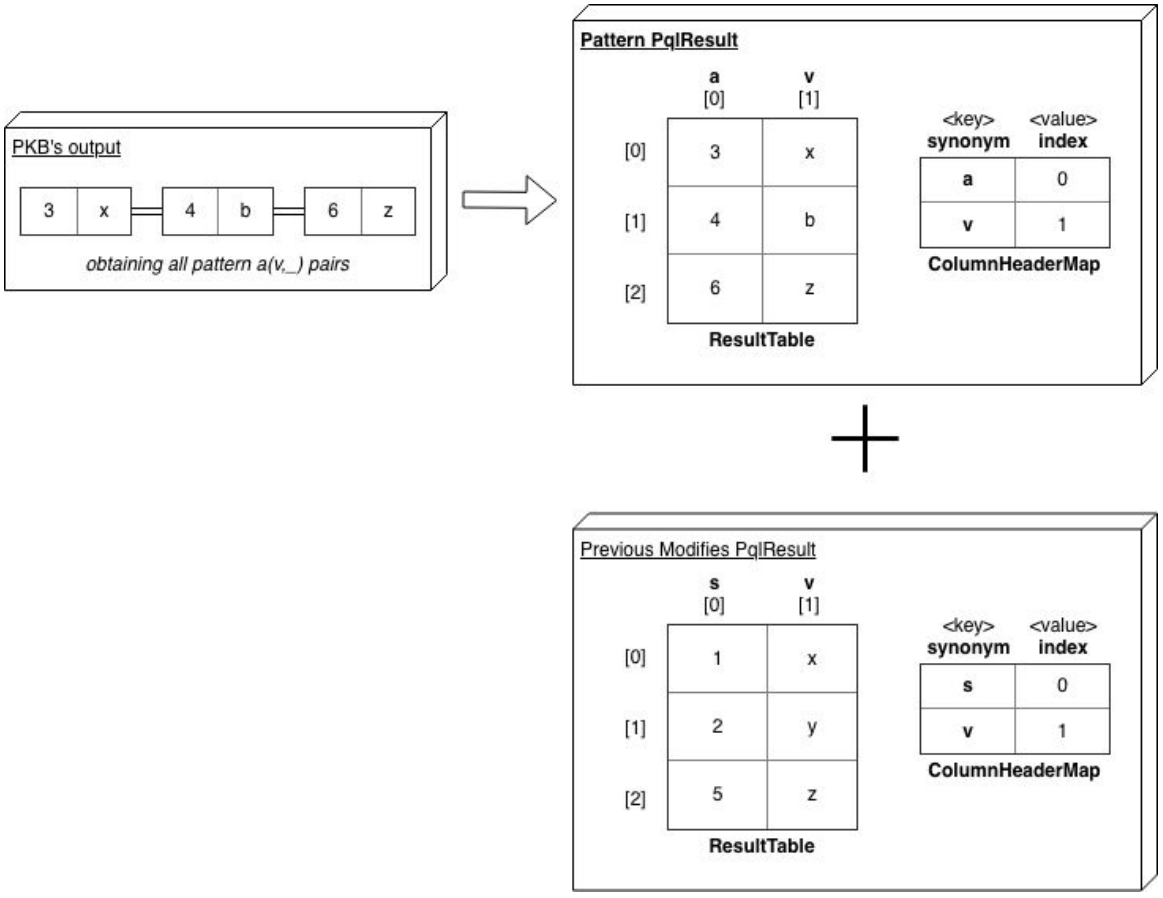
1) A single synonym match between the two `ResultTable`s,
2) No synonym match
3) Both synonym match

To summarise, below are the methods used for the 3 cases:

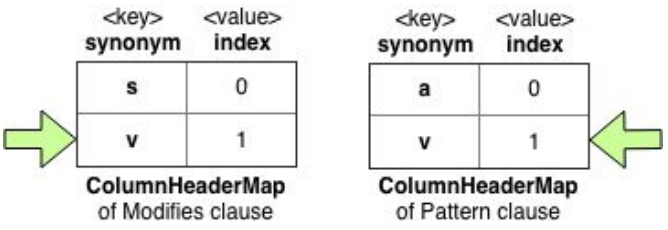
1) One synonym match - Get intersections of the tables (Similar to SQL INNER JOIN)
2) No synonym match - Cross product of the tables (Similar to SQL JOIN)
3) Two synonym match - Get intersections of the tables (Similar to SQL INNER JOIN)

Case 1: Single synonym match

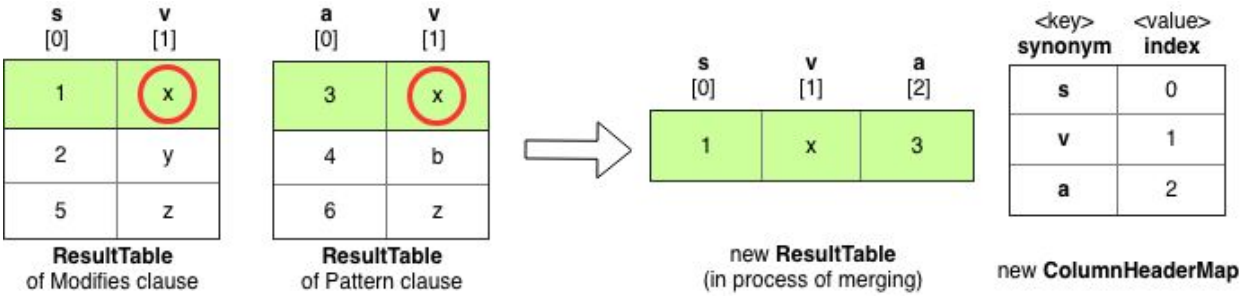
Following the previous example, say there is another clause "pattern a(v,_)", where another list of result pairs "(3, x) (4, b) (6, z)" is retrieved and directed into the `PqlResult` class, we will now have two `ResultTable`s as shown below:



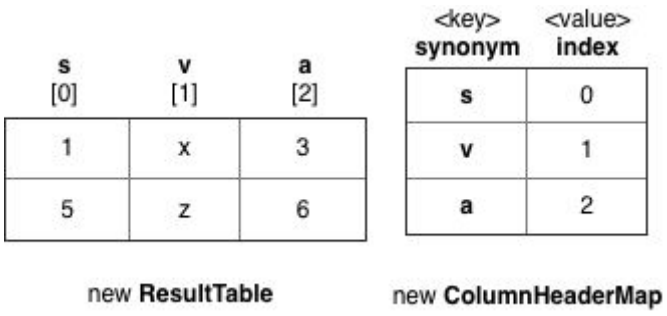
To merge the second `ResultTable` to the first, the class will first check if the same SYNONYM is in the first table by using the hashmap. In this example, the `ResultTable` of the previous Modifies clause has column "v" at index 1 and the `ResultTable` of the newly evaluated Pattern clause also has column "v" at index 1.



Next, the function will compare both table's column 1 value for every row to determine if they have the same value. In this case, the first row of table one match the first row of table two, so it is merged together like so:

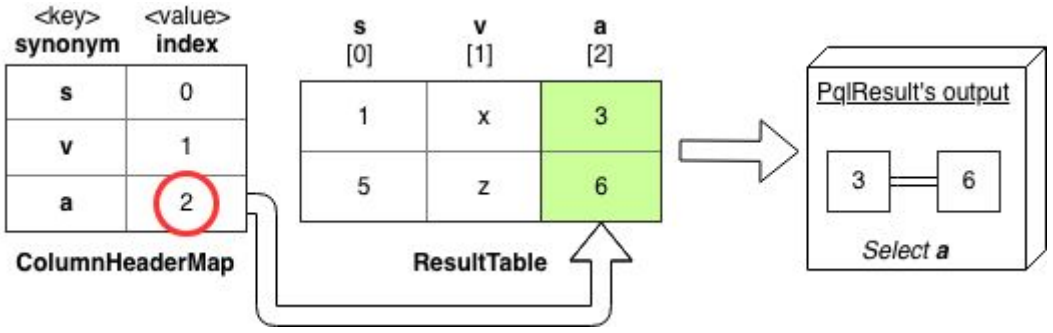


Notice that the column has been increased as we have merged the value from column "a" as well. After comparing every row, the end result will be as follows:



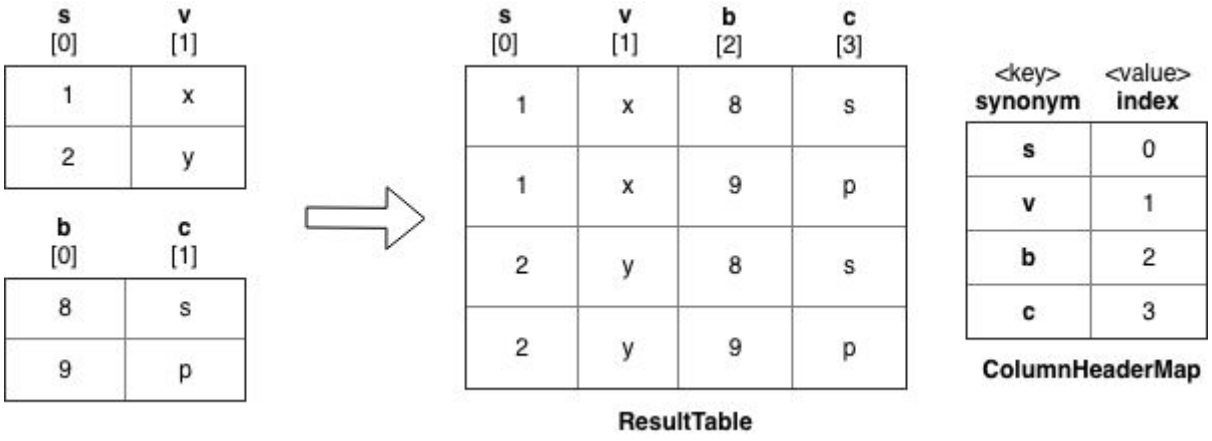
After merging, the old `ResultTable` are discarded as they are not relevant anymore.

Finally, the evaluator is ready to evaluate the Select clause. It will use the SYNONYM of the Select clause as the key to look up the `ColumnHeaderMap`. Say the Select clause is "Select a", the evaluator will use SYNONYM "a" as the key, and get a value "*2*" back from the `ColumnHeaderMap`, which refers to column index 2 of the `ResultTable`. All values of column 2 are then extracted into a list<string>, which will then be returned to the GUI (which will display the answer: "3, 6"), as shown below. An empty list<string> will be returned to the GUI if the table is empty.



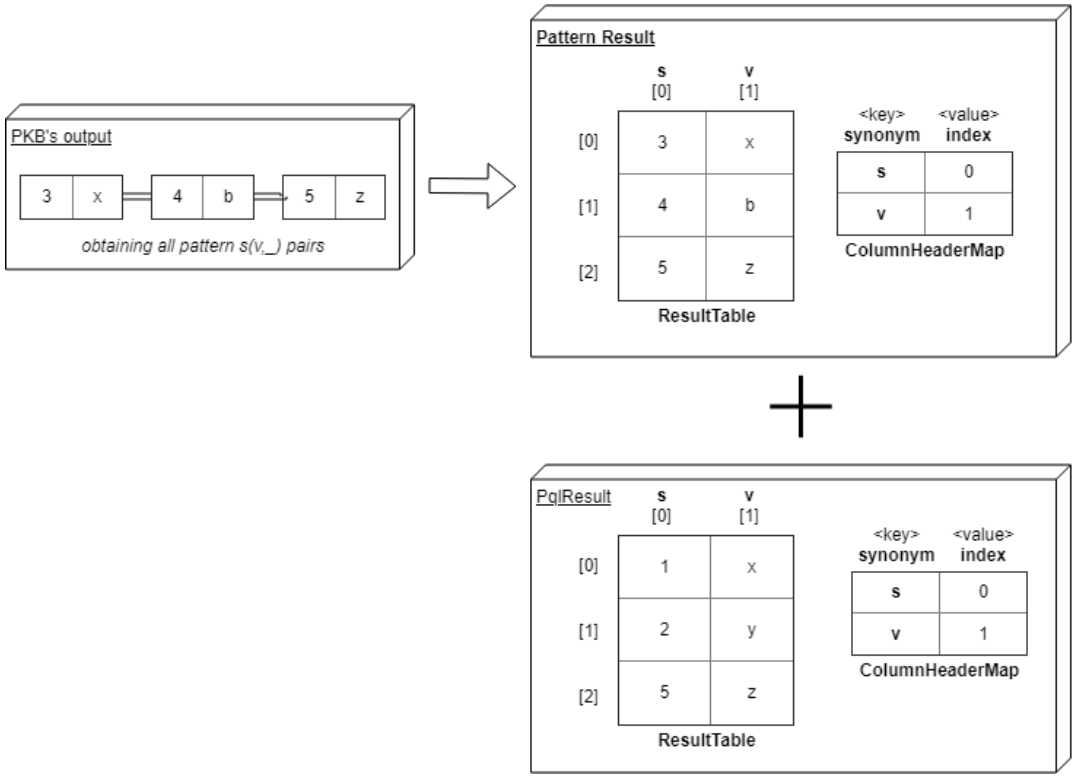
Case 2: No synonym match

In the event that the second table does not have matching SYNONYM with the first table, the two tables will be merged using **cross-product**. Example:



Case 3: Both synonym match

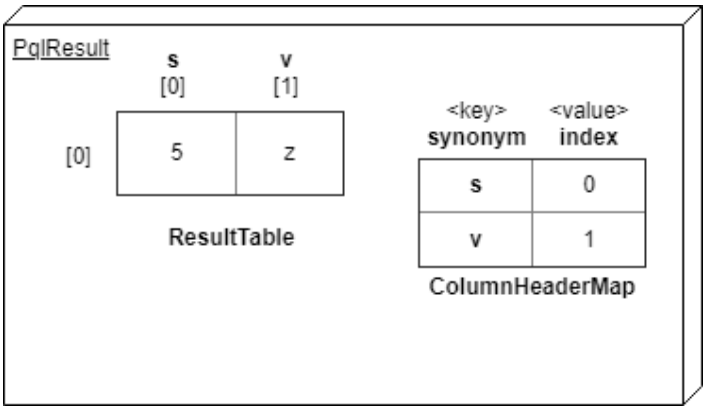
There will be cases where the second table have **TWO** matching SYNONYM with the first table. Example:
"pattern s(v,_)", return a new list of result pairs "(3, x) (4, b) (5, z)"



In this case, the function will first check if the first SYNONYM of the new pattern result is found in `ColumnHeaderMap` and retrieve its index it it exists. It will do the same for its second SYNONYM.

The function will then proceed to **compare** the two SYNONYM between the two table using the retrieved index, row by row. If **both** SYNONYM matches with their counterparts, a new `ResultTable` will be generated and **matching** row will be added into the new `ResultTable`.

After every row has been compared, the old `ResultTable` is discarded and **replaced** by the new one. In the case of the above example, the new `ResultTable` should only have one row ("5","z") after merging as shown below:



In the event where there are **ZERO** matching elements for the two tables, the new `ResultTable` will be empty, and will return immediately, without evaluating any more clauses for optimization. (This means that the constraints for this new clause does not match the other constraints) The unordered_map will also be cleared as there are no more columns.

Documentation and Coding Standards

Coding Standards

We adopted the C++ style guide by Google[3]. We did not introduce any deviation from the style guide, and as such perusing the Google guide is enough to gain a general understanding of how our code looks.

Documentation of Abstract APIs

Naming Conventions and API Documentation Standards

The team took great care in following a standard naming convention for our documentation. We used terms that are used in the SPA Requirements document so that our reader can understand our documentation more easily. For example, we use terms like **SYNONYM** and **DESIGN-ENTITY** when describing the elements of a PQL query such as "assign a; Select a". Readers can immediately understand that the **DESIGN-ENTITY** refers to "assign" and that **SYNONYM** refers to "a". This helps the documentation to be read semantically naturally.

As suggested in the lecture notes, we followed the following structure for documenting abstract APIs:

***Overview**: An overview of the module*

API
RETURN-TYPE MethodName (**PARAM-TYPE** paramName) ***Description:*** What does the method return? Are there any special cases (such as null or empty list)?

To ensure that the abstract API is language agnostic - that is, the API is not limited by the actual implementation language - we adopted some naming conventions to write the abstract API.

Instead of primitive types such as String or Integer, we used types specific to the SPA, such as **LIST_OF_STMT_NUMS** or **LIST_OF_VAR_NAMES**. This is a better alternative than using primitive types, such as LIST_OF_STRINGS. This practice helps other developers in the team to understand what each of the method does. See below for a method copied from the API for **ModifiesTable,** which adopts this standard:

API
LIST_OF_STMT_NUM_AND_VAR_PAIRS GetAllModifiesPair() ***Description:*** Returns a list of all <modifying stmt, modified var> pairs. (e.g. Select s1 such that modifies(s2, v))

[3] https://google.github.io/styleguide/cppguide.html

Use of Typedefs

Such standards in API documentation translated nicely into our practice of using *typedefs* in our code. In the header files, we defined our own type aliases so that:

1. The code is semantically easier to understand. Knowing that a method returns a 'list of variable names' is more helpful than knowing that it returns an arbitrary 'list of strings'.
2. When types of entities change, it is easy to make changes across the entire code base. For instance, if we decide to use 'strings' instead of 'integers' for Statement Numbers, every instance of the type alias 'StmtNum' can be changed very easily by changing the typedef of `StmtNum` itself. It would be a pain to do so, if we were not using typedefs.
3. Reduces reader fatigue, improving code readability. Imagine if you saw return types of `unordered_map<string, vector<string>>` all around the code, it would be very hard to make sense of the data.

A few examples of typedefs (`using` keyword in C++11) found in our code and in this document:

```
using TokenList = vector<Token>
using StmtNum = int
using ResultTable = vector<vector<string>>
using ModifiesMap = unordered_map<string, vector<string>>
```

Testing

Test plan

The team first planned and design a set of queries (test suite) for testing the system. There are many different arguments in the query clauses so the team has divided the query set by the different clauses (e.g Follows, Modifies, Uses, Parent, etc) For each query, we will be testing the Syntax, Validity and the final Result output to the user.

We enforced frequent unit testing after each individual component has been implemented. This is to ensure that bugs found in integration testing is scoped to bugs related with components integration.

For unit and integration tests, we employed Visual Studio's code coverage feature to highlight to us any part of our code that was not covered by tests, and aimed to write test cases that would cover those statements. What we did for system testing will be elaborated upon in the next few sections.

Syntax

We have used both positive and negative test cases to test the syntax of the queries to ensure that our query parser is working properly.

Validity

We have also used both positive and negative test cases to test the validity of each arguments in the query's clause. (e.g For Uses, the right argument must be an UNDERSCORE/IDENT/SYNONYM of variable design-entity type). Negative tests should result in an empty result.

Forming test cases

Equivalence partition was used to decide what test cases to write, and we wrote test cases that were designed to cover each partition at least once. For example, the equivalence partitions of a **Follows** clause may be as such:

Equivalence partition	Valid inputs	Invalid inputs
First parameter	- INTEGER, - synonym (design-entity: *'stmt', 'if', 'print', 'read', 'while', 'assign*', 'prog_line') - Underscores "_"	- synonym (design-entity: *'constant', 'procedure', 'variable'*) - "IDENT" - Symbols like !@#%^ - Empty field
Second parameter	- INTEGER,	- synonym (design-entity: *'constant', 'procedure',*

	- synonym (design-entity: *'stmt', 'if', 'print', 'read', 'while', 'assign*', 'prog_line') - Underscores "_"	*'variable'*) - "IDENT" - Symbols like !@#%^ - Empty field

We made sure our test cases included at least queries with VALID-VALID, VALID-INVALID, and INVALID-VALID parameters to test for bugs.

Result

Finally, the team has to look through the sample source code in order to come up with proper queries that the system has answers to in order to test whether the evaluator is working as intended. The team has written many combinations of clauses and arguments when each function of clauses has been developed so that we can ensure that the function is working without bugs. When new functions are written, regression testing is also done for older functions. The team has written the test cases into our own sample-queries text document so that it is integrated into SPA and Autotester can test it automatically throughout the project.

At the time of writing, we have more than **180** AutoTester test cases (*Sample-Queries.txt*)!

Unit Testing

Front End

As the token stream is assumed to be syntactically correct, as it has already been validated, the Parser has no real unit tests (but lots of integration tests). As such, the sample test cases for the front end will be focused on the Validator.

Sample Test Case #1

Test Purpose

To test the validation of a procedure with incorrect assignment statement (in stmt #3), consisting of a valid post-fixed expression but invalid assignment expression according to SIMPLE's CSG.

Required Test Inputs

SIMPLE source:

```
procedure one {
    a = b;                      \\ 1
    if (a > b) then {           \\ 2
        e = f c d * -;          \\ 3
        asd = ddd;              \\ 4
    } else {
        k = 3;                  \\ 5
        print print;            \\ 6
```

```
    }
}
```

Expected Test Results

Expected result = FALSE
Validator receives a tokenized program, detects the error in the assignment statement at statement 3, and returns false.
The return value is asserted to be FALSE.

Sample Test Case #2

Test Purpose

To test the validation of a procedure with varying amounts of whitespace or lack of whitespace. It should still pass validation as whitespace should not matter.

Required Test Inputs

SIMPLE source:

```
procedure                        two {
            a=                       b             ;   \\ 1
     if (a>       b ) then {                           \\ 2
            e = f;          asd=ddd;                  \\ 3 & 4
                         } else {
    k = 3;                                            \\ 5
                  print print
;                                                     \\ 6
            }}
```

Expected Test Results

Expected result = TRUE
Validator receives a tokenized program, which consists a perfectly valid (albeit unreadable) syntax. Validator should be able to validate it properly and realize it is a valid program and returns true.
The return value is asserted to be TRUE.

Query Processor

Sample Test Case #1

Test Purpose

To test parsing of such that Uses(Stmt) clause

Required Test Inputs

"assign a; variable v; Select a such that Uses(a,v)"

Expected Test Results

Declarations: a of assign design-entity type, v of variable design-entity type
Select clause: a of assign design-entity type
Suchthat type: UsesS (for statements)
Arguments: First argument - a of assign design-entity type
Second argument - v of variable design-entity type

Sample Test Case #2

Test Purpose

To test parsing of pattern(Assign) clause

Required Test Inputs

"variable v; assign a; Select v pattern a(v,\"oSCar\")"

Expected Test Results

Declarations: a of assign design-entity type, v of variable design-entity type
Select clause: v of variable design-entity type
Pattern type: a of assign design-entity type
Arguments: First argument - v of variable design-entity type
Second argument - expression without underscore

Assertion of Equality - Comparing the result with expected value to see if they are equal.
Example: Is the design-entity type of Select clause equal to the expected design-entity type
Assertion of True/False - Comparing if the outcome of the result is true
Example: Is the result table empty after merging two conflicting constraints

System (Validation) Testing

Sample Test Case #1

Test Purpose

Test the evaluation of such that Modifies and pattern assign clause. This is a test case with a constrained synonym "a". The result should contain all variables that are modified by an assignment statement and on the right of the assignment statement the expression must be "1 * beta % tmp".

Required Test Inputs

Source file

prototype_sample_SIMPLE_source.txt as uploaded on IVLE.

AutoTester input

1- MODIFIES[assign][variable] pattern-expression [assign]
assign a; variable v;

Select v such that Modifies (a, v) pattern a (_, "1 * beta % tmp")
oSCar
5000

Expected Test Results

oSCar

Sample Test Case #2

Test Purpose

Test the return value when evaluation clauses are true with the selected synonym not being constrained by any of the clauses. The result should be all statements if the clauses are true. The clauses are also fairly complex.

Required Test Inputs

Source file

prototype_sample_SIMPLE_source.txt as uploaded on IVLE.

AutoTester input

2 - PARENT*[while][assign] pattern-single-expression [assign]
while w; assign a; stmt s;
Select s such that Parent*(w, a) pattern a(_,_"Romeo"_)
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
5000

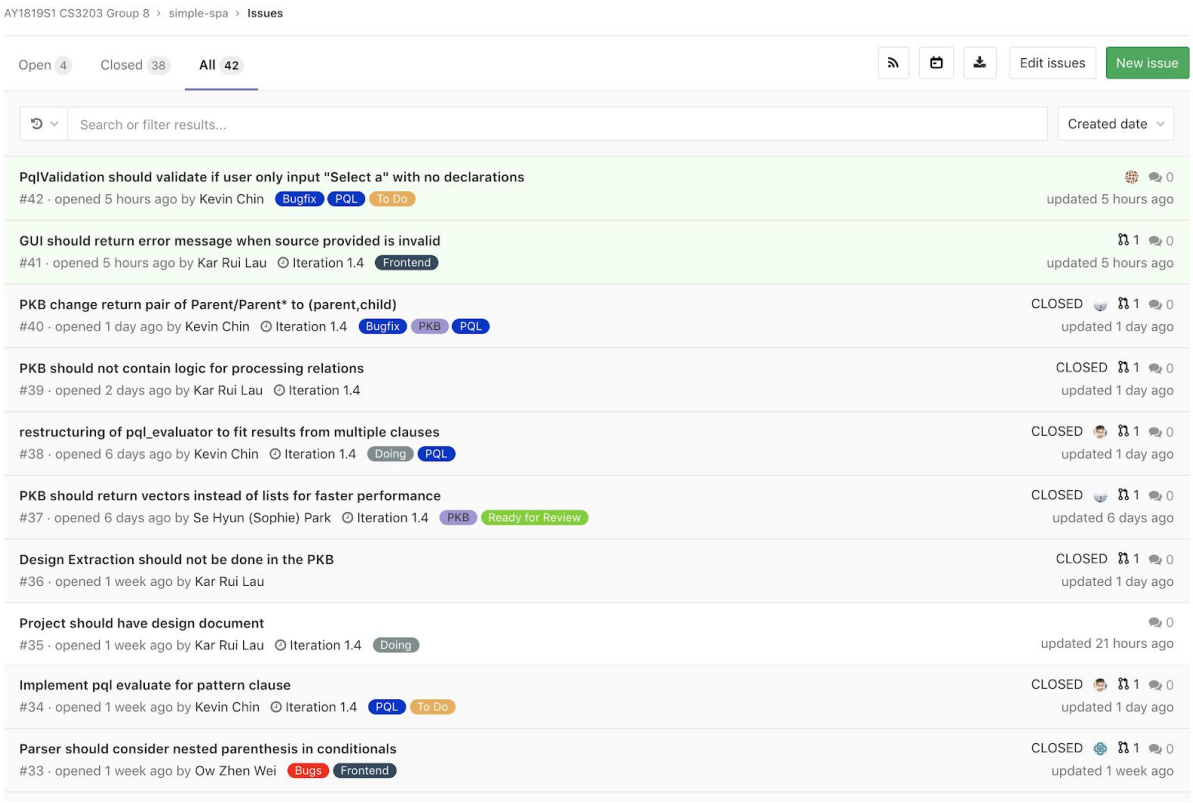
Expected Test Results

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Discussion

Managing Tasks & Issues

We used the Issues board on GitLab to create issues and assign them to one another as tasks. At the beginning of each mini iteration, each team member would log the tasks that they would like to achieve during the mini iteration as an issue. Then, they would assign it to themselves and raise a pull request after a few days. This helped the entire team to have a clear overview of what everyone is doing for the iteration. Hence, it became easy for team members to ask for help or extra code reviews from other members who may not have so much tasks for that week. A screenshot of our issue board can be seen below:



Kar Rui's General Thoughts

Zhen Wei and I were mainly responsible for the front end portion of the code base. We sat down and drew crude diagrams and thought heavily about extensibility and readability even before we started coding. I prioritized readability and flow of the program while Zhen Wei prioritized having working code regardless of the runtime complexity.

I would say we learnt from each other and worked well together, as I would sometimes not start work until I can visualize the program's flow. By having Zhen Wei go ahead with a "brute force" algorithm, I am able to refactor the program piece by piece as there was already basic functionality, while Zhen Wei would be able to appreciate the importance of design principles as he worked in the future mini iterations in extending the functionality of the program. To this point, I would say this arrangement worked well for me, albeit slightly more work in implementation as we had to heavily refactor parts sometimes.

What would I do differently?

I don't think I would do anything differently if I had to start the project again, except write the code we have currently in the codebase from the start (since we already know the implementation). I think our code base is pretty enjoyable to trace through and efficient, as the group heavily followed design principles such as Separation of Concerns and have efficiency in mind when they are implementing their parts of the program.

Zhen Wei's General Thoughts

Kar Rui and I worked closely together to design the Front-end component. While it was quite challenging to decide upon a method for parsing and validation, brainstorming ideas and critiquing them together has helped us in picking a suitable design option.

Throughout the implementation process, I am glad that we did frequent testing as it has helped us to uncover both major and minor bugs. These bugs would have been fatal if it was not discovered early.

After the first iteration of CS3203, it has also reinforced the importance of good communication. Although there was a little confusion between the Front-end and PKB side, it was resolved smoothly as we explained our ideas and thoughts clearly to one another.

Se Hyun (Sophie)’s General Thoughts

For iteration 1, I worked on the PKB with Mingrui. It was helpful to discuss the design of the PKB in-depth before going into the implementation. Mingrui and I communicated closely with the other team members to figure out what they need from the PKB. Then, we wrote up a comprehensive abstract API that would meet most of their needs. Towards the end of the iteration, there were some methods that we missed out, but these were easily added to the programme. I learned that it is helpful to discuss the design decisions for a software and document it prior to implementation, as this can ensure that everybody is on the same page on how the system is created.

Due to the nature of the PKB, there were some implementation changes every mini-iteration. For instance, the data structure or the return type would change because it’s a better decision for the whole team. But even when there were such changes, it was helpful that we had frequent code reviews and adhered to coding standards.

Mingrui’s General Thoughts

Sophie and I worked on the PKB component for Iteration 1. At the start of each mini iteration, we drafted an updated PKB abstract API to show Front-end and PQL teams and make sure we were going to implement all the operations they need. Although frequent changes needed to be made sometimes, overall the implementation was not a big challenge since PKB is the simpler and more straight-forward part compared to Front-end and PQL.

In retrospect, I realise communication between different teams is very important. Even though each member works on a specific component, it is still a good idea to discuss with everyone about what you are going to implement and how you are going to do it. Because other members might have a better idea regarding the implementation or spot a mistake that

you overlooked. It can also avoid doing repetitive work when two components have similar tasks. It is also crucial to spend some time on planning and deciding on the design decisions before actual implementation. At the start, we simply assumed that PKB is going to populate all the relationships and Parser just needs to insert statements to PKB. But later on we realised it would be a better design if Parser and Design Extractor do the population of relationships so we changed the implementation. Doing more thorough planning before implementation would minimise changes needed at the later stage of the project.

Ken's General Thoughts

Together with Kevin, we were responsible for the PQL component. I was mainly engaged in doing the parsing and validation of the query, while Kevin did the evaluation. We had 3 mini-iterations in total, starting off with the basic assign declaration and select, then proceeding to such that clause and pattern clause. Kevin and I worked close to discuss about the internal data structure needed to store the query information. We initially started small with just one class to store the query, but later on expanded to 2 more classes for modularity. There was a bit back and forth in deciding the details of the data structure as we talk to the PKB team to understand how information can be retrieved. Based on the API provided by the PKB, the PQL component has to make adjustments accordingly.

As the project group is big with many components to work on, communication is key to success. Due to our individual busy schedules, we interacted mostly through Telegram which works out pretty well as all of us are committed to complete our individual components. I engage mostly with Kevin as the parsing, validation, and evaluation have to work hand in hand together. The PQL and PKB teams also have our private chat to discuss on matters regarding integration and ensuring that the PKB can provide the necessary functions for query evaluation.

As I am in charged of implementing the query internal data structures, I make it a point to code early in the iterations so that I can get it reviewed by Kevin, and providing him time to familiarize with the implementation before coding the evaluation part. As such, there is no procrastination on this project and any on fire bugs regarding the parser or data structure needs to rectify as soon as possible for smooth integration and component testing

In hindsight, I think our group has pretty good rapport and if something can be different, I would hope that the team can have more common free time together so that we can engage even closely and update everybody with deeper details of the project.

Kevin's General Thoughts

As the person working on the pql evaluator, I realise that communication is very important as I have to use the API and object provided by other component and functions. If there was a lot of miscommunication, it will result in having the wrong result output in the query evaluation, which is bad. Therefore, we have taken great care in our communication both within the component itself and with the PKB team as well.

Sometimes, I have to start coding early, without having the other functions ready. Therefore, I have to code based on my own assumption of what the function will return to my function, without a way to really check if it is correct or not. (Apart from Unit test, of course) This is

definitely a unique experience to be coding without having all the parts needed, like a special form of Test Driven Development.

Documentation of Abstract APIs

Below is a list of abstract APIs for the basic entities and design abstractions that were drafted and consolidated over a few weeks.

VarList

Overview: VarList stores the variables passed into PKB by the Parser.

API
LIST_OF_VARNAMES GetAllVar() ***Description:*** If VarList is not empty, returns a list of all varNames in the VarList. Else returns empty list.

ConstList

Overview: ConstList stores the constants passed into PKB by the Parser.

API
LIST_OF_CONSTANTS GetAllConstValue() ***Description:*** Return a list of all constants in the ConstList.

*there are no public insertion functions for VarList and ConstList because PKB inserts all modified or used variables and constants whenever a statement is inserted

ProcList

Overview: ProcList stores the procedure names passed into PKB by the Parser.

API
VOID InsertProcName(**STRING** procName) ***Description:*** If procName is not already in the ProcList, inserts procName into the ProcList.
LIST_OF_PROCNAMES GetAllProcName() ***Description:*** Returns a list of all procNames in the ProcList.

StatementTable

Overview: StmtTable (Statement Table) stores the statement number, statement type, and statement list index for all statements.

API

BOOLEAN InsertStmt(**STMT_NUM** stmt_num, **STMT_TYPE** stmt_type, **STMT_LIST_INDEX** stmtlist_index) ***Description:*** Inserts a statement into the StmtTable. Returns true if the statement is not already in StmtTable and is inserted, false otherwise.
STMT_LIST_INDEX GetStmtListIndex(**STMT_NUM** stmt_num) ***Description:*** Returns the statement list index of the given statement.
STMT_TYPE GetStmtType(**STMT_NUM** stmt_num) ***Description:*** Returns the statement type of the given statement.

Follows, Follows*

Overview: FollowsTable stores all the Follows and Follows* (= FollowsT) relationships.

API
BOOLEAN IsFollows(**STMT_NUM** s1, **STMT_NUM** s2) ***Description:*** Returns TRUE if Follows(s1, s2) is TRUE, FALSE otherwise.
BOOLEAN IsFollowsT(**STMT_NUM** s1, **STMT_NUM** s2) ***Description:*** Returns TRUE if Follows*(s1, s2) is TRUE, FALSE otherwise.
LIST_OF_STMT_NUMS GetFollows(**STMT_NUM** s) ***Description:*** Returns a list of all n's that satisfy Follows(s, n).
LIST_OF_STMT_NUMS GetFollowsT(**STMT_NUM** s) ***Description:*** Returns a list of all n's that satisfy Follows*(s, n).
LIST_OF_STMT_NUMS GetAllFollows() ***Description:*** Returns a list of all n's that satisfy Follows(_, n)/Follows*(_, n).
LIST_OF_STMT_NUMS GetFollowedBy(**STMT_NUM** s) ***Description:*** Returns a list of all n's that satisfy Follows(n, s).
LIST_OF_STMT_NUMS GetFollowedByT(**STMT_NUM** s) ***Description:*** Returns a list of all n's that satisfy Follows*(n, s).
LIST_OF_STMT_NUMS GetAllFollowed() ***Description:*** Returns a list of all n's that satisfy Follows(n, _)/Follows*(n, _).
BOOLEAN HasFollowsRelationship() ***Description:*** Returns TRUE if Follows(_, _) is TRUE, FALSE otherwise.
LIST_OF_STMT_NUM_PAIRS GetAllFollowsPair() ***Description:*** Returns a list of all pairs of <s1, s2> that satisfy Follows(s1, s2).
LIST_OF_STMT_NUM_PAIRS GetAllFollowsTPair() ***Description:*** Returns a list of all pairs of <s1, s2> that satisfy Follows*(s1, s2).

Parent, Parent*

Overview: ParentTable stores all the Parent and Parent* (= ParentT) relationships.

API
BOOLEAN IsParent(**STMT_NUM** s1, **STMT_NUM** s2) ***Description:*** Returns TRUE if Parent(s1, s2) is TRUE, FALSE otherwise.
BOOLEAN IsParentT(**STMT_NUM** s1, **STMT_NUM** s2) ***Description:*** Returns TRUE if Parent*(s1, s2) is TRUE, FALSE otherwise.
LIST_OF_STMT_NUMS GetParent(**STMT_NUM** s) ***Description:*** Returns a list of all n's that satisfy Parent(n, s).
LIST_OF_STMT_NUMS GetParentT(**STMT_NUM** s) ***Description:*** Returns a list of all n's that satisfy Parent*(n, s).
LIST_OF_STMT_NUMS GetAllParent() ***Description:*** Returns a list of all n's that satisfy Parent(n, _)/Parent*(n, _).
LIST_OF_STMT_NUMS GetChild(**STMT_NUM** s) ***Description:*** Returns a list of all n's that satisfy Parent(s, n).
LIST_OF_STMT_NUMS GetChildT(**STMT_NUM** s) ***Description:*** Returns a list of all n's that satisfy Parent*(s, n).
LIST_OF_STMT_NUMS GetAllChild() ***Description:*** Returns a list of all n's that satisfy Parent(_, n).
BOOLEAN HasParentRelationship() ***Description:*** Returns TRUE if Parent(_, _) is TRUE, FALSE otherwise.
LIST_OF_STMT_NUM_PAIRS GetAllParentPair() ***Description:*** Returns a list of all pairs of <s1, s2> that satisfy Parent(s1, s2).
LIST_OF_STMT_NUM_PAIRS GetAllParentTPair() ***Description:*** Returns a list of all pairs of <s1, s2> that satisfy Parent*(s1, s2).

ModifiesP, ModifiesS

Overview: ModifiesTable stores all the Modifies relationships.

API
BOOLEAN IsModifiedByS(**VAR_NAME** v, **STMT_NUM** s) ***Description:*** Returns TRUE if Modifies(s, v) is TRUE, FALSE otherwise.
BOOLEAN IsModifiedByP(**VAR_NAME** v, **PROC_NAME** p) ***Description:*** Returns TRUE if Modifies(p, v) is TRUE, FALSE otherwise.
LIST_OF_VAR_NAME GetModifiedVarS(**STMT_NUM** s) ***Description:*** Returns a list of all v's that satisfy Modifies(s, v).
LIST_OF_VAR_NAME GetModifiedVarP(**PROC_NAME** p) ***Description:*** Returns a list of all v's that satisfy Modifies(p, v).
LIST_OF_STMT_NUM GetModifyingStmt(**VAR_NAME** v) ***Description:*** Returns a list of all s's that satisfy Modifies(s, v).

LIST_OF_PROC_NAME GetModifyingProc(**VAR_NAME** variable) ***Description:*** Returns a list of all p's that satisfy Modifies(p, v).
LIST_OF_STMT_NUM GetAllModifyingStmt() ***Description:*** Returns a list of all s's that satisfy Modifies(s, _).
LIST_OF_PROC_NAME GetAllModifyingProc() ***Description:*** Returns a list of all p's that satisfy Modifies(p, _).
LIST_OF_STMT_NUM_AND_VAR_PAIRS GetAllModifiesPairS() ***Description:*** Returns a list of all pairs of <s, v> that satisfy Modifies(s, v).
LIST_OF_PROC_NAME_AND_VAR_PAIRS GetAllModifiesPairP() ***Description:*** Returns a list of all pairs of <p, v> that satisfy Modifies(p, v).

UsesP, UsesS

Overview: UsesTable stores all the Uses relationships.

API
BOOLEAN IsUsedByS(**VAR_NAME** v, **STMT_NUM** s) ***Description:*** Returns TRUE if Uses(s, v) is TRUE, FALSE otherwise.
BOOLEAN IsUsedByP(**VAR_NAME** v, **PROC_NAME** p) ***Description:*** Returns TRUE if Uses(p, v) is TRUE, FALSE otherwise.
LIST_OF_VAR_NAME GetUsedVarS(**STMT_NUM** s) ***Description:*** Returns a list of all v's that satisfy Uses(s, v).
LIST_OF_VAR_NAME GetUsedVarP(**PROC_NAME** p) ***Description:*** Returns a list of all v's that satisfy Uses(p, v).
LIST_OF_STMT_NUM GetUsingStmt(**VAR_NAME** v) ***Description:*** Returns a list of all s's that satisfy Uses(s, v).
LIST_OF_PROC_NAME GetUsingProc(**VAR_NAME** v) ***Description:*** Returns a list of all p's that satisfy Uses(p, v).
LIST_OF_STMT_NUM GetAllUsingStmt() ***Description:*** Returns a list of all s's that satisfy Uses(s, _).
LIST_OF_PROC_NAME GetAllUsingProc() ***Description:*** Returns a list of all p's that satisfy Uses(p, _).
LIST_OF_STMT_NUM_AND_VAR_PAIRS GetAllUsesPairS() ***Description:*** Returns a list of all pairs of <s, v> that satisfy Uses(s, v).
LIST_OF_PROC_NAME_AND_VAR_PAIRS GetAllUsesPairP() ***Description:*** Returns a list of all pairs of <p, v> that satisfy Uses(p, v).

PatternTable

Overview: PatternTable stores all the patterns (Only assign patterns for Iteration 1).

API
LIST_OF_STMT_NUM GetAssignWithPattern(**VAR_NAME** v, **EXPRESSION** sub_expr) ***Description:*** Returns a list of a's that satisfy pattern a(v, _sub_expr_).
LIST_OF_STMT_NUM GetAssignWithExactPattern(**VAR_NAME** v, **EXPRESSION** exact_expr) ***Description:*** Returns a list of a's that satisfy pattern a(v, exact_expr).
LIST_OF_STMT_NUM_VAR_PAIR GetAllAssignPatternPair(**EXPRESSION** sub_expr) ***Description:*** Returns a list of all pairs of <a, v> that satisfy pattern a(v, _sub_expr_).
LIST_OF_STMT_NUM_VAR_PAIR GetAllAssignExactPatternPair(**EXPRESSION** exact_expr) ***Description:*** Returns a list of all pairs of <a, v> that satisfy pattern a(v, exact_expr).

**Some methods that are not required in Iteration 1 are omitted.*