



# **Static Program Analyzer (SPA)**

## **Project Iteration 3 Report**

KIANG SIONG BOON (A0245446B)

LEE WEE LUN (A0245304N)

# Table of Contents

<b>Project Background</b>	<b>3</b>
<b>Static Program Analyzer (SPA)</b>	<b>3</b>
<b>Prototype Implementation</b>	<b>4</b>
Iteration 1 implementation	4
Iteration 2 implementation	4
Iteration 3 Implementation	5
<b>Database Design</b>	<b>7</b>
Database Diagram (Iteration 1, 2, 3)	7
Table Description (Iteration 1)	8
Table Description (Iteration 2)	8
Table Description (Iteration 3)	9
Table Relationship (Iteration 1)	11
Table Relationship (Iteration 2 & Iteration 3)	11
Database Design Rational (Iteration 1)	11
Database Design Rational (Iteration 2)	12
Database Design Rational (Iteration 3)	13
<b>SPA Implementation</b>	<b>15</b>
Source Processor (Iteration 1, 2)	15
Source Processor (Iteration 3)	20
Query Processor (Iteration 2)	20
Design	20
Example	21
Query Processor (Iteration 3)	22
Design	22
Example	23
<b>Testing</b>	<b>26</b>
Approach (Iteration 1)	26
Examples	27
Approach (Iteration 2)	28
Examples	28
Approach (Iteration 3)	29
“Next” and “Next*”	29
“Calls” and “Calls*”	31
“Modifies” and “Uses”	31
Tuple < >	32
Order of Clause	33

# Project Background

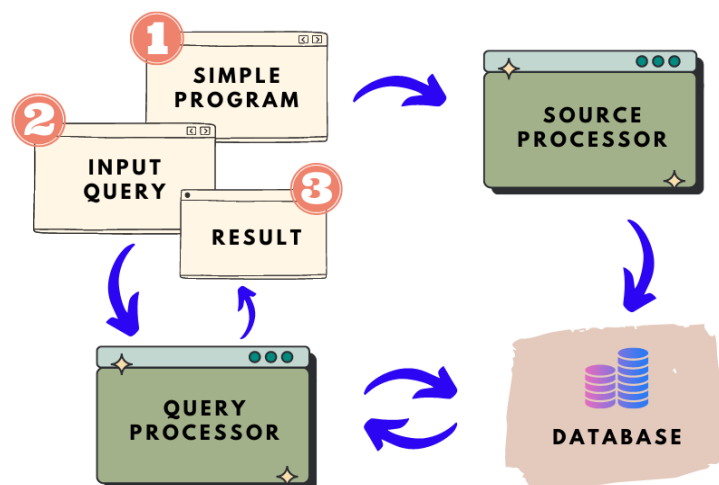
There is a high demand for IT experts in the modern digital era, particularly software developers. These professionals are in charge of developing, testing, and maintaining the software applications utilized by businesses and organizations all around the world.

One of the key tasks that software developers often have to deal with is understanding legacy code. This refers to code that has been written in the past and is still in use, but may not be well-documented or easy to understand.

To address this problem, there is a need for a tool that can help to break down large chunks of code into smaller, more manageable pieces of information. The goal of this tool is to improve the software maintenance process and make it more efficient, saving companies time and resources.

## Static Program Analyzer (SPA)

For the SPA to respond to program queries, a source program must first be analyzed, with relevant program design entities extracted and stored in a database. The user is then given the ability to utilize Program Query Language (PQL) to inquire about the program using the SPA . These PQL queries are processed by the SPA and the results are displayed to the user based on the data held in the database.



# Prototype Implementation

## Iteration 1 implementation

All of these prototype features have been implemented in iteration 1 of the project. Additionally, we have fulfilled all of the prototype iteration 1's requirements.

### 1. Simplified Programming Language (SIMPLE)

- a. Single Procedure
- b. Statement
- c. Read / Assign / Print / Call
- d. Variable

### 2. Database (SQL)

- a. Procedure
- b. Variable
- c. Statement
- d. Constant

### 3. Program Query Language (PQL)

- a. Declaration (single)
- b. Select Cause (single)

## Iteration 2 implementation

These are new prototype features that have been implemented on top of iteration 1 implementation

### 1. Simplified Programming Language (SIMPLE)

- a. While statement

- b. If statement
- c. Expression
- d. Parent/Next/Use/Modify/Call/Pattern

## **2. Database (SQL)**

- a. Modify
- b. Use
- c. Next
- d. Parent
- e. Call
- f. Pattern

## **3. Program Query Language (PQL)**

- a. Declaration (Multiple Synonyms)
- b. Such that (Single Cause)
  - i. Parent
  - ii. Modifies
  - iii. Uses
- c. Pattern (Single Cause)
  - i. ("variable",\_)
  - ii. ("variable","\_expression\_")
  - iii. (\_, "\_expression\_")
  - iv. (\_,\_)

## **Iteration 3 Implementation**

We completed all the required PQL for this iteration, and redesigned the query processor.

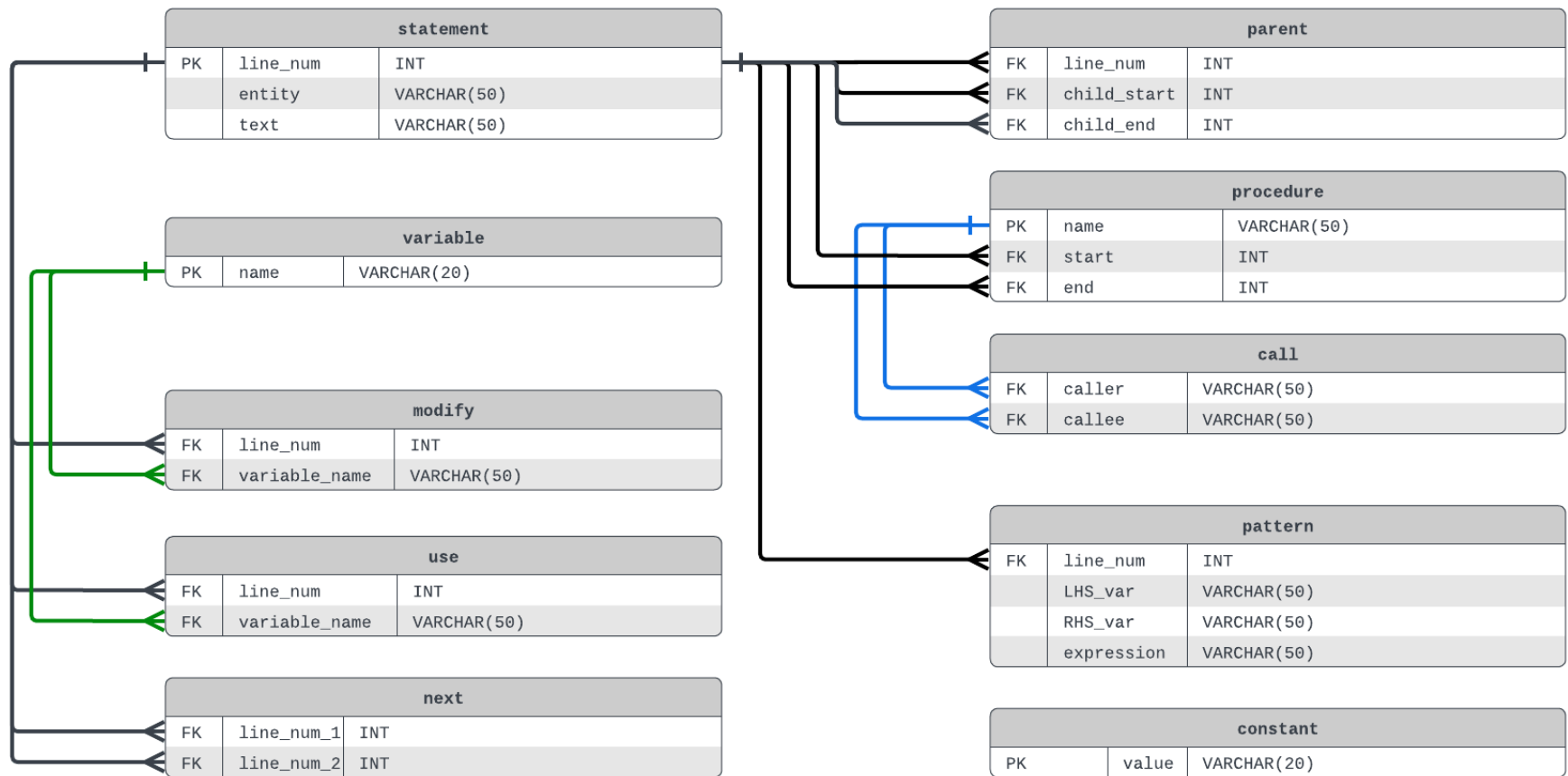
### **1. Program Query Language (PQL)**

- a. Such that (Single)
  - i. Call, Calls\*

- ii. Next, Next\*
  - iii. Modifies (procedure / “\_” / “IDENT” , variable / “\_” / “IDENT”)
  - iv. Uses (procedure / “\_” / “IDENT” , variable / “\_” / “IDENT”)
- b. Such that (Multiple)
- c. Pattern (Multiple)
- d. Meaningless Query
- e. Tuple Query
- f. Order of clauses

# Database Design

## Database Diagram (Iteration 1, 2, 3)



## Table Description (Iteration 1)

Table	Column	Description
procedure	name	Stores the procedure name
statement	line_num	Stores the statement number
	procedure_name	Stores the procedure name that contains this statement
	entity	Stores the different types of statements such as print, read, assign, call, etc.
	text	Contains the content for the the statement to be used for pattern query
variable	name	Contains the variable name
constant	value	Contains the constant value

## Table Description (Iteration 2)

Table	Column	Description
parent	line_num	Stores the statement number of the container
	child_start	Stores the statement number of the first line in the container
	child_end	Stores the statement number of the last line in the container



modify	line_num	Stores the statement number where modifies(s,v) is true
	variable_name	Stores the variable name where modifies(s,v) is true
use	line_num	Stores the statement number where uses(s,v) is true
	variable_name	Contains the variable name where uses(s,v) is true
call	line_num	Stores the statement number
	procedure_name	Stores the procedure name
	variable_name	Call variable in this procedure
	direct_call	Direct call within the procedure or indirect call via other procedure
pattern	line_num	Stores the statement number
	LHS_var	Store left hand side variable of assign
	RHS_var	Store right hand side variable of assign
	expression	Store postfix of RHS variable
next	line_num_1	Current line number of Control Flow Graph
	line_num_2	Next line number of Control Flow Graph

### Table Description (Iteration 3)

Table	Column	Description
-------	--------	-------------

parent	line_num	Stores the statement number of the container
	child_start	Stores the statement number of the first line in the container
	child_end	Stores the statement number of the last line in the container
modify	line_num	Stores the statement number where modifies(s,v) is true
	variable_name	Stores the variable name where modifies(s,v) is true
use	line_num	Stores the statement number where uses(s,v) is true
	variable_name	Contains the variable name where uses(s,v) is true
call	caller	Procedure name that who make the call
	callee	Procedure name that who receive the call
pattern	line_num	Stores the statement number
	LHS_var	Store left hand side variable of assign
	RHS_var	Store right hand side variable of assign
	expression	Store postfix of RHS variable
next	from	Current line number of Control Flow Graph
	to	Next line number of Control Flow Graph

## Table Relationship (Iteration 1)

Design Entity	Relationship	Design Entity
procedure	One to One	statement
variable	One to Many	statement
constant	-	-

## Table Relationship (Iteration 2 & Iteration 3)

Design Abstraction	Relationship	Design Entity
modify	Many to One	statement
modify	Many to One	variable
use	Many to One	statement
use	Many to One	variable
parent	Many to One	statement
next	Many to One	statement
call	Many to One	procedure
pattern	Many to One	statement

## Database Design Rational (Iteration 1)

The database is designed around the PQL. We analyzed the PQL, and determined that we're able to answer the Iteration 1 PQL queries using the database design shown above.

We created the Statement table for stmt, read, print, assign, while, if, call query. We store them in the same table as all of them return a Statement number. We identify the different entities via the entity field, and fulfill the pattern query via the text field.

### Example of PQL to SQL

PQL	SQL
print p; select p	SELECT line_num FROM statement WHERE entity = 'print'
stmt s; select s	SELECT line_num FROM statement;

We created the Variable, Procedure, and Constant table for variable, procedure, and constant query.

For the Constant table, we determined that there's no Relationship query that uses constant. Hence, we don't need to link it with the other tables.

For the Variable, and Procedure table, we determined that there are Relationship query that uses Variable, and Procedure. Hence, we separated them to simplify the SQL query, and linked them together via the Statement table using Foreign Keys.

## Database Design Rational (Iteration 2)

For the design abstractions of "Uses" and "Modifies", we decided to store all statements and variables that satisfy the indirect and direct relationships to the SQL table. By doing so, we aim to simplify the PQL to SQL conversion by eliminating the use of recursive query

For the design abstraction of "Parent", we store the container statement number, and the child start and end statement number. This design allows us to fulfill the Parent and

Parent\* PQL as we are able to determine whether a given Parent statement number is the direct or indirect Parent via SQL

For the pattern clause, we store the statement number, variables and expression of the “assign” to the SQL table. The expression stores the post-fix expression. This allows us to easily find the pattern of the expression.

### PQL to SQL Example

PQL	SQL
<pre>assign t; while w; Select t such that Parent(w, t)</pre>	<pre>select 1 from statement s where s.line_num = x and ((select p.line_num from parent p join statement s2 on p.line_num = s2.line_num where s2.entity = 'while' and s.line_num between p.child_start and p.child_end order by p.line_num desc limit 1) = (select p.line_num from parent p where s.line_num between p.child_start and p.child_end order by p.line_num desc limit 1))</pre>
<pre>assign a; variable v; Select a such that Uses(a, v)</pre>	<pre>select 1 from use u where u.line_num in (select s.line_num from statement s where entity = 'assign');</pre>

### Database Design Rational (Iteration 3)

For “Uses” and “Modifies”, we stored all statements and variables that satisfy the indirect and direct relationships to the SQL table. We want to simplify the PQL to SQL conversion by eliminating the use of recursive query.

For “Next” and “Next\*”, we build the Control Flow Graph and store the CFG nodes as “from” and “to” column. Then, we use the Next table to handle the “Next” relationship, and use recursive query to handle “Next\*”.

For “Calls” and “Calls\*”, we need to accommodate multiple levels of nested calls in “Calls\*”, it became clear that the old call table from iteration 2 was insufficient, as it could only store the first level of nested calls. To address this issue, we undertook a redesign of the table responsible for storing both the callee and caller which belong to direct call only. This redesign still enabled us to identify both direct and indirect calls, which was achieved through the implementation of recursive queries within the query processor.

### PQL to SQL Example

PQL	SQL
Uses(5, v), where 5 is a call statement	Select variable_name as v from (select s.line_num, m.variable_name from statement s join modify m on m.line_num between (select start from procedure where name = s.text) and (select end from procedure where name = s.text) where s.entity = 'call')
Next*(ifs, w)	<pre> select from as ifs, to as w from ( select * from (with recursive nextT as (select from_line, to_line from next where from_line in (select line_num from statement where entity = 'ifs') union select n.from_line, n.to_line from next n join nextT nT on n.from_line = nT.to_line) select * from nextT)  intersect  select * from (with recursive nextT as (select </pre>

	<pre> from_line, to_line from next where to_line in (select line_num from statement where entity = 'while') union select n.from_line, n.to_line from next n join nextT nt on n.to_line = nt.from_line) select * from nextT) ) </pre>
<pre> Calls*(p, q) </pre>	<pre> select distinct p,q from ((SELECT p0.name AS p,p1.name AS q FROM procedure p0,procedure p1)  natural join  (select distinct caller as p, callee as q from (with recursive callT as ( select caller, callee from call union select ct.caller, c.callee from callT cT join call c on c.caller = cT.callee) select * from callT))) </pre>

## SPA Implementation

### Source Processor (Iteration 1, 2)

We utilized the given tokenizer to tokenize the SIMPLE code. Then, we iterated the tokens array, and used the SIMPLE grammar rules to process the code in the following order, and logic.

1. We created the following variables to store the information that we need.
  - a. `vector<Procedure*> procedure;`
    - i. It stores all the Procedures that were parsed.
    - ii. The SourceProcessor will insert the statements that satisfy the `Uses(s,v)`, `Modifies(s,v)`, and `Calls(p,q)` to the `_uses`, `_modifies`, `_calls` vector respectively.
    - iii. When the parse completes, we'll iterate through each Procedure, performs some processing on the `_uses`, `_modifies`, `_calls` to insert the statements and variables that satisfy the indirect and

direct Uses(s,v), Modifies(s,v) and Calls(p,q) relationship into their respective table.

- iv. When the parse completes, we iterate through each Procedure, and insert the Procedure names into the “procedure” table.

b. `stack<Container*> parentStack;`

- i. It stores all the Containers (if/else/while/procedure) that were parsed. We treat the “else” keyword as Containers also.
- ii. The purpose is to help us to create a tree representation of the code for constructing the Control Flow Graph, and the Parent(s,v) relationship
- iii. The SourceProcessor will create a new Container object when it encounters `if`, `else`, `while`, and `procedure` keywords, and push it onto the stack.
- iv. The SourceProcessor will pop the Container object from the stack when it encounters a closing curly brackets “}”.
- v. The SourceProcessor will create a Statement object, and insert it into `_statements` vector when it encounters a statement.
- vi. The SourceProcessor will create a Container object, and insert it into `_childContainers` vector when it encounters a Container.

c. `vector<Statement*> callStatements;`

- i. It stores all the `call` Statement that were parsed
- ii. The purpose is to assist us in populating the indirect and direct Uses(s,v) and Modify(s,v).

d. `int stmtNumSubtract = 0;`

- i. It stores the extra incremented statement number due to the `else` keyword.
- ii. Our method to construct the Control Flow Graph requires us to treat `else` as a separate Container instead of a statement under an `if` Container. Hence, we treat it as a statement, and assign a statement number to it. Thus, we need to account for the extra



incremented statement number so that we can get the correct statement number as defined in the project

- e. `int stmtNum = 0;`
  - i. It stores the current statement number
- f. `int nestedLevel = 0;`
  - i. It stores the current nested level for each statement.
  - ii. The purpose is to assist us in constructing the Control Flow Graph
- 2. We iterate through the tokens, and parse it as follows
- 3. If the word is “procedure”
  - a. Get the procedure name at index + 1, and insert it into the procedure table.
  - a. Create a Procedure object, populate it with the correct data, and push it onto the `procedure vector`, and `parentStack`
  - b. Increment the `nestedLevel`
- 4. If the word is “while”
  - a. Increment the `stmtNum` and `nestedLevel`
  - b. Create a Container object, fill it with the correct data, and push it onto the `parentStack`.
  - c. Create a new Statement object, fill it with the correct data, and insert it into the created Container object `_statements` vector attribute.
  - d. Perform a while loop to process the condition portion. The terminating condition is when we encounter “}”.
  - e. Within the while loop, we use regex match to insert the correct tokens into the Constant or Variable table.
  - f. At the end, we insert this While statement into the Statement table as a “while” statement type.
- 5. If the word is “if”
  - a. Increment the `stmtNum` and `nestedLevel`
  - b. Create a Container object, populate it with the correct data, and push it onto the `parentStack`.

- c. Create a new Statement object, fill it with the correct data, and insert it into the created Container object `_statements` vector attribute.
  - d. Perform a while loop to process the condition portion. The terminating condition is when we encounter the keyword “then”.
  - e. Within the while loop, we use regex match to insert the correct tokens into the Constant or Variable table
  - f. At the end, we insert this “if” statement to the Statement table as an “if” statement type.
6. If the word is “else”
  - a. Increment the `stmtNum`, `nestedLevel`, and `stmtNumSubtract`
  - b. Create a Container object, populate it with the correct data, and push it onto the `parentStack`.
  - c. Create a new Statement object, fill it with the correct data, and insert it into the created Container object `_statements` vector attribute.
7. If the word is “=”
  - a. Create a Statement object, fill it with the correct data, and insert it into the Container object `_statements` vector attribute via `parentStack.top()`
  - b. Insert the Left Hand Side variable into the Variable, and Modifies table
  - c. Perform a while loop to process the Right Hand Side assign statement. The terminating condition is “;”.
  - d. Within the while loop, we use regex match to insert the correct tokens into the Constant or Variable table
  - e. The right hand side values are converted to postfix expression and inserted into the pattern table.
  - f. At the end, we insert this statement into the Statement table as an “assign” type.
8. If the word is “read”, “print” or “call”
  - a. Create a Statement object, fill it with the correct data, and insert it into the Container object `_statements` vector attribute via `parentStack.top()`

- b. We insert the statement into the Statement table as the respective type.
  - c. If the word is “read” or “print”.
    - i. We insert the variable into the Variable table
    - ii. We insert the Statement object into Uses or Modifies table depending on whether the word is “print” or “read”
  - d. If the word is “call”
    - i. We insert the Statement object into the `callStatements` vector
9. When all the tokens has been parsed, we iterate through the `procedure` vector to do the following
- a. Do additional processing to insert all direct and indirect Uses(s,v) and Modifies(s,v) statements and variables into the Uses and Modifies table.
  - b. Construct Control Flow Graph, and insert each Control Flow Graph node into the Next table.
  - c. Get all the Containers, and insert them into the Parent table.

The regex pattern for Variable is as follows.

```
^(?! (procedure|while|if|then|else|call|read|print)+$) [A-Za-z] [A-Za-z0-9]*
```

It does the following

1. If the word matches “procedure”, “while”, “if”, “then”, “else”, “call”, “read”, “print”, skip.
2. If (1) isn’t skipped, match only if the word starts with an alphabet, followed by 0 or more alphanumeric. It follows the SIMPLE grammar for Variables

The regex pattern for Constants is as follows. It matches a series of digits

```
^[0-9]+$
```

Using the above logic, we’re able to parse the SIMPLE grammar, extract, and insert the correct data into the various tables

## Source Processor (Iteration 3)

The following is a requirement for Iteration 3, and had been implemented in Iteration 2 under point 9

- Next
- Next\*
- Indirect Modifies(s, v)
- Indirect Uses(s, v)

We implemented “Calls” and “Calls\*” by doing the following steps for each call statement

1. Insert the caller and callee to a list.
2. Insert direct call to the database by getting the caller from the procedure stack, and callee from the token.

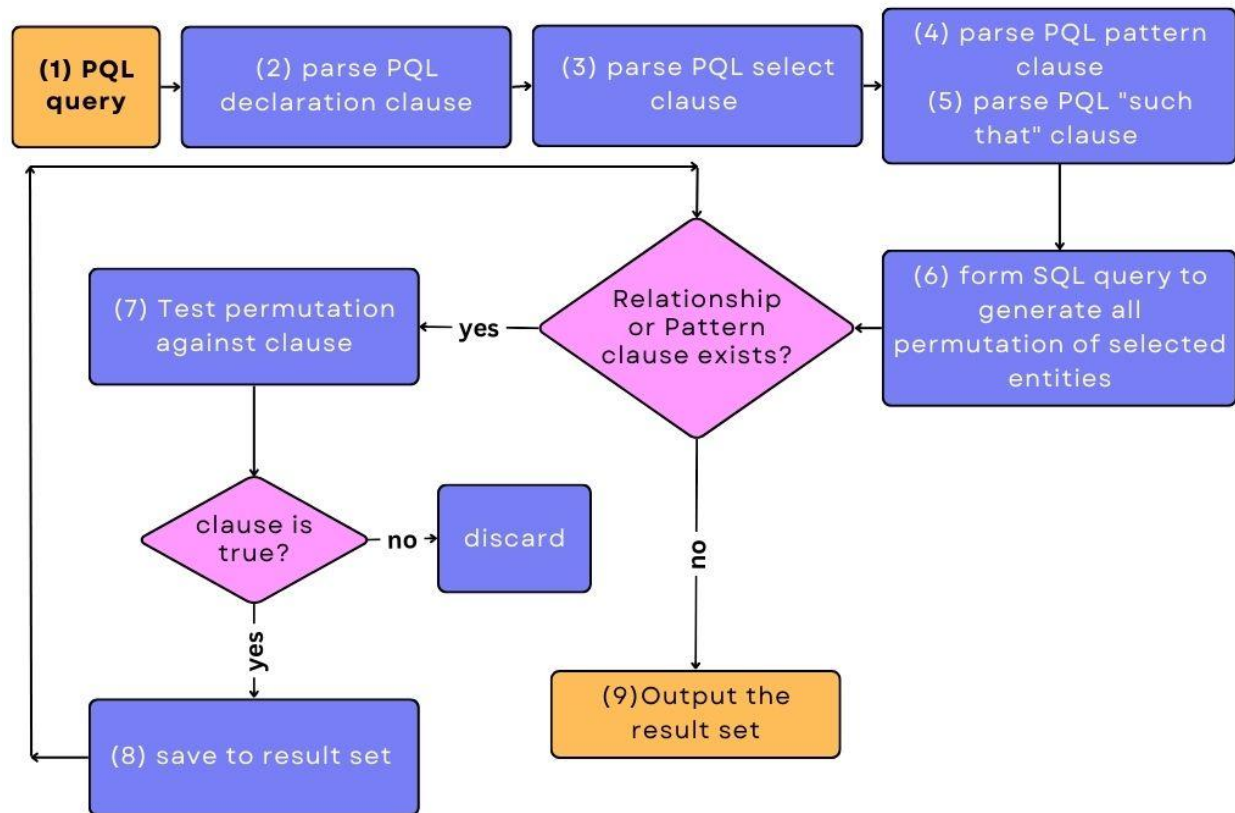
## Query Processor (Iteration 2)

### Design

We designed the query processor such that it's able to

- Return the correct output for different entity synonyms
- Handle any number of “such that” and “pattern” clause

The diagram shows an illustration of the query processor flowchart below.



## Example

We illustrate the flowchart using the following PQL query

```
while w; variable v;
Select <w,v> such that Parent(w,v)
```

1. In step (2), we process the declared entity and its synonym. This step is necessary for step (3) as the entity type determines the returned output. For example, "while", "if", "call", etc return statement numbers, and "variable", "procedure" return names.
2. In step (3), we process the selected synonym, and construct the SQL statement to obtain the selected synonyms' permutations.
  - a. In this example, the SQL statement would be

```
select  s.line_num,  v.name  from  statement  s  where  
s.entity = "while" join variable v
```

3. In step (4) and (5), we process the “such that” and “pattern” clauses to store the relationship inputs, and add them to a stack.
4. In step (6), we execute the SQL statement constructed in step (2) to get the full data set.
5. In step (7), we test each permutation in the data set against all the “such that” and “pattern” clauses. We created database functions to cater for all relationships and entities. The database functions would form and execute the SQL query based on the input. Then, the function returns true if there's a result, and false if there's none. Using the returned boolean, we would determine whether the “such that” and “pattern” clause is true for the given permutation
  - a. In this example, we'll call the Parent function, and pass the “while” statement number, and variable names as input.
6. If there's no more clauses to test, we return the output.

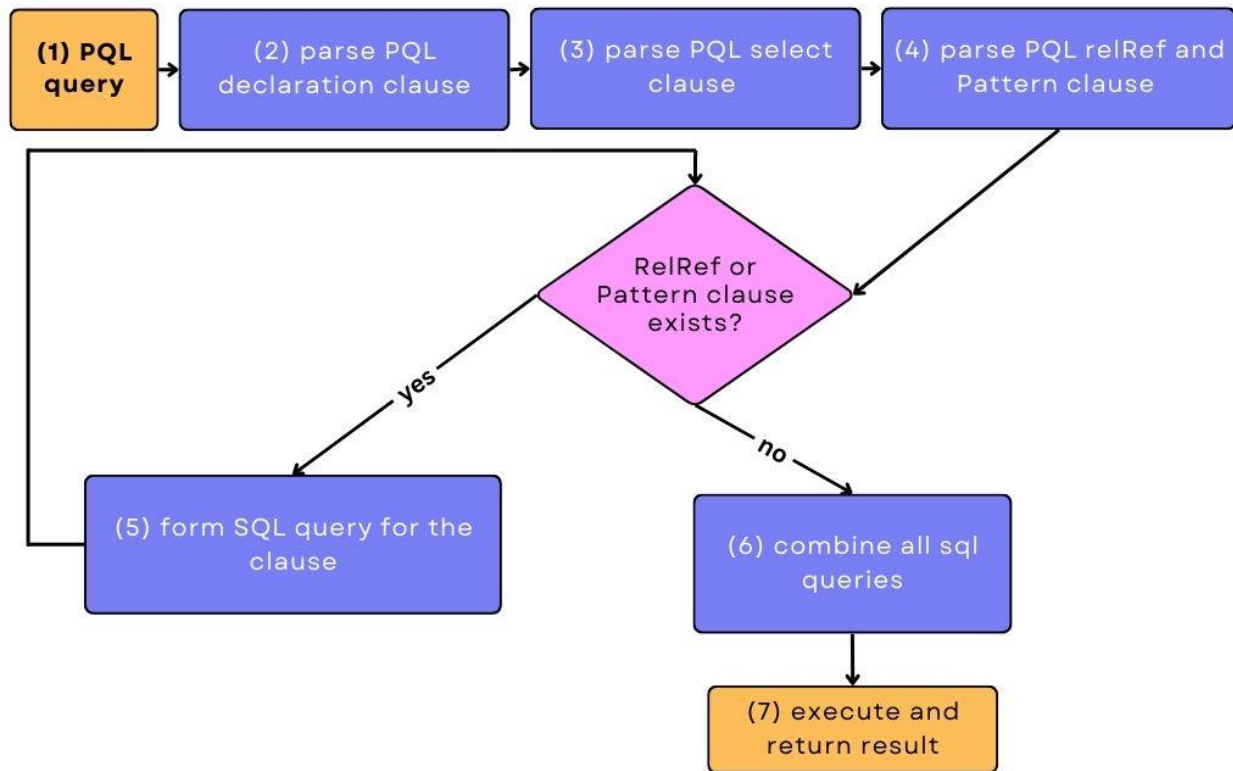
## Query Processor (Iteration 3)

### Design

We redesigned the query processor such that it's able to

- Return the correct output for different entity synonyms.
- Handle any number of “such that” and “pattern” clauses.
- Handle a large number of tuples.

We changed the handling of the constrain clauses (“such that” and “pattern”) for iteration 3. The diagram shows an illustration of the query processor flowchart below



## Example

We illustrate the flowchart using the following PQL query

```

stmt s; read re; while w; if i; variable v;
Select <s,w,i,v> such that Next*(re,w) such that Parent(w,i)
such that Parent(i,s) such that Uses(s,v)
  
```

1. Step (1) to (4) processes the PQL to extract relevant information for us to create the correct SQL statements for “select”, “uses”, and “pattern”. The process is the same as Iteration 2.
2. In step (5), we build the SQL query for the constrain clauses. There are two parts to the SQL query - Column SQL and Result SQL. Column SQL selects the column as synonym, and Result SQL queries the table. Then, the two parts are joined together to form the final PQL to SQL query. Example is given below for “Uses(s,v)”. The same logic applies to all the other clauses

Clause	Part	SQL
Uses(s,v)	Column SQL	Select line_num as s, variable_name as v
	Result SQL	Select p.line_num, u.variable_name from parent p join use u on u.line_num between p.line_num and p.child_end union select line_num, variable_name from use
	Final SQL	Select line_num as s, variable_name as v from (select p.line_num, u.variable_name from parent p join use u on u.line_num between p.line_num and p.child_end union select line_num, variable_name from use)

3. In step (6), we have already constructed the SQL queries for all the constrain clauses. We join them together into a single SQL query and utilized the SQL “NATURAL JOIN” keyword to achieve the following
- If two query results have common columns, join both on common columns.
  - If two query results do not have common columns, perform cross join.

Example is given below for the PQL to SQL translation

PQL	SQL
Select <s,w,i,v>	select distinct s,w,i,v from (
stmt s; read re; while w; if i; variable v;	( SELECT s0.line_num AS s,s1.line_num AS w,s2.line_num AS i,v0.name AS v FROM statement s0,statement



	<pre>s1,statement s2,variable v0 WHERE s1.entity='while' AND s2.entity='if' )</pre>
	natural join
such that Uses(s,v)	<pre>( select distinct line_num as s, variable_name as v from (select p.line_num, u.variable_name from parent p join use u on u.line_num between p.line_num and p.child_end union select line_num, variable_name from use) )</pre>
	natural join
such that Parent(i,s)	<pre>( select distinct parent as i, child as s from (select p.line_num as parent, s.line_num as child from parent p join statement s on s.line_num between p.child_start and p.child_end where p.line_num in (select line_num from statement where entity = 'if') and p.line_num = (select p.line_num from parent p where s.line_num between p.child_start and p.child_end order by p.line_num desc limit 1)) )</pre>
	natural join
such that Next*(re,w)	<pre>( select distinct parent as w, child as i from (select p.line_num as parent, s.line_num as child from parent p join statement s on s.line_num between p.child_start and p.child_end where s.entity = 'if' and p.line_num in (select line_num from statement where entity = 'while') and p.line_num = (select p.line_num from parent p where s.line_num between p.child_start and p.child_end order by p.line_num desc limit 1)) ) natural join ( select distinct from_line as re, to_line as w from (select * from (with recursive nextT as (select from_line, to_line from</pre>

	<pre> next where from_line in (select line_num from statement where entity = 'read') union select n.from_line, n.to_line from next n join nextT nT on n.from_line = nT.to_line) select * from nextT) intersect select * from (with recursive nextT as (select from_line, to_line from next where to_line in (select line_num from statement where entity = 'while') union select n.from_line, n.to_line from next n join nextT nt on n.to_line = nt.from_line) select * from nextT)) ) </pre>
--	---

4. The SQL result will be outputted.

## Testing

### Approach (Iteration 1)

Our approach is as follows

1. We tested the base case by having a single procedure, un-nested condition statements, and all the various entities such as call, read, print, etc.
2. We increased complexity by
  - Adding multiple nested conditions.
  - Adding equations to the If and While condition, and Assignment statement.
3. We test the pattern matching rule by
  - Adding variation of Variable names from the SIMPLE keywords.
4. We produce the final test cases by combining (2), and (3)

The Source Processor is correct if it passes all the above. Examples as follows

## Examples

### Nested Conditions

```
if(...) then{
    while(...) {
        if(...) then{
            while(...) {
            }
        }
        else{
            while(...) {
            }
        }
    }
}
else{
    while(...) {
    }
    if(...) then{
    }
    else{
        while(...) {
        }
    }
}
```

### Complex Conditions and Equations

```
If == ((WHile4) + ((CaLLg)) * ((1224 - AssiGn)/(pRINT * 2212))
```

```
pRINT = ((rEAD) * 2) - ((PRinT + 1) * (WHile / PRinT) / PRINT /
(ReAd)
```

### Variation of Variable names from SIMPLE keywords

```
Read, Print, Assign, While, If, Call, rEAD, pRINT, aSSIGN, wHILE,
```

```
If, cALL, ReaD, PriNt, AssiGn, While, iF, CalL, READ, PRINT,  
ASSIGN, WHILE, IF, CALL, ififif, callcallcall
```

## Approach (Iteration 2)

We identified that the input to the “such that” and “pattern” clauses can be classified into generic or specific input.

The input is generic if

- It’s a declared synonym and the synonym is not present in the select clause.

The input is specific if

- It’s a declared synonym and the synonym is present in the select clause.
- It’s not a declared synonym

Hence, there are four cases to test

1. (Generic , Generic)
2. (Generic , Specific)
3. (Specific, Generic)
4. (Specific, Specific)

## Examples

PQL query for (Generic, Generic) inputs

```
variable v; procedure p; statement s;  
Select s such that Uses(p, v)
```

PQL query for (Generic, Specific) inputs

```
variable v; procedure p  
Select v such that Uses(p, v)
```

```
procedure p;  
Select p such that Uses(p, "myVar")
```

PQL query for (Specific, Generic) inputs

```
variable v; procedure p  
Select p such that Uses(p, v)  
  
variable v;  
Select p such that Uses(p, "myVar")
```

PQL query for (Specific, Specific) inputs

```
variable s;  
Select s such that Uses("main", "myVar")
```

## Approach (Iteration 3)

### “Next” and “Next\*”

We created a container pattern matrix to test the correctness of our CFG builder code with different container patterns as the CFG varies depending on the container type and nest level.

		If-Else	While
No nested	No head and tail statement	yes	yes
	With head statement	yes	yes
	With tail statement	yes	yes
	With head and tail statements	yes	yes

<b>Nested If-Else</b>	<b>No head and tail statement</b>	yes	yes
	<b>With head statement</b>	yes	yes
	<b>With tail statement</b>	yes	yes
	<b>With head and tail statements</b>	yes	yes
<b>Nested While</b>	<b>No head and tail statement</b>	yes	yes
	<b>With head statement</b>	yes	yes
	<b>With tail statement</b>	yes	yes
	<b>With head and tail statements</b>	yes	yes

Then, we checked the database result to ensure correctness. Then, we created the following “Next” and “Next\*” PQL test cases to test the SQL correctness of our QueryProcessor for all the possible inputs. We grouped “stmt” and “\_” together as both use the same SQL query.

1. (stmt / “\_” , stmt / “\_”)
2. (stmt / “\_” , read/print/assign/while/if/call)
3. (stmt / “\_” , Digit)
4. (stmtRef without stmt , stmt / “\_”)
5. (stmtRef without stmt , stmtRef without stmt)
6. (stmtRef without stmt , Digit)
7. (Digit , stmt / “\_”)
8. (Digit , stmtRef without stmt)
9. (Digit , Digit)

## Examples

```

assign a; while w; stmt s;
select s such that Next*(w, a)

assign a; while w; stmt s;
select s such that Next(w, a)

while w; stmt s;
select s such that Next(w, _)

print pr; stmt s;
select pr such that Next(pr,s)

while w;
select w such that Next(10,11)

```

## “Calls” and “Calls\*”

We utilized the existing source codes to test the correctness of our SourceProcessor. Then, we created the following test cases to test the SQL correctness of our QueryProcessor for all the possible inputs.

1. (procedure / “\_”, procedure / “\_”)
2. (procedure / “\_”, “IDENT”)
3. (“IDENT”, procedure / “\_”)
4. (“IDENT”, “IDENT”)

## Examples

```

procedure p,q;
select such that Calls(p,q)

procedure p;
select such that Calls(p,_)

procedure p;
select p such that Calls(_, "proc1")

```

## “Modifies” and “Uses”

For “Modifies for (all) statements”, “Uses for (all) statements”, “Modifies for procedures”, “Uses for procedures”, we utilized the existing source codes to test the correctness of our SourceProcessor in extracting the indirect Modifies and Uses

statements. Then, we created the following test cases to test the SQL correctness of our QueryProcessor.

1. (Digit, "IDENT") where digit is a call statement
2. (Digit , variable / "\_" ) where digit is a call statement
3. (procedure , "IDENT")
4. (procedure , variable / "\_" )
5. ("IDENT" , "IDENT")
6. ("IDENT" , variable / "\_" )

### Examples

```
assign a;
select a such that Modifies(5, "var1")

assign a;
select a such that Modifies(5, _)

assign a; variable v;
select a such that Modifies(5, v)
```

### Tuple < >

We created the following test cases to test the correctness of our QueryProcessor in handling tuples.

### Examples

```
procedure p, q;
select <p,q> such that Calls(p,q)

procedure p, q;
Select <p,q> such that Calls*(p,q)

print pr; while w; if ifs;
select <w,ifs,pr> such that Parent(w,ifs) such that
Next(ifs,pr)

if f1; while w2, w3;
Select <f1, w2, w3> such that Parent* (f1, w2) such that
Parent* (w2, w3)
```



## Order of Clause

We tested order of clause by swapping the clauses and checking if it returns the correct result.

### Examples

```
assign a; variable v;  
Select a pattern a (v,_) pattern a (_,_"cenX"_)  
  
assign a; variable v;  
Select a pattern a (_,_"cenX"_) pattern a (v,_)
```