TIC2003 Software Development Project
AY23/24 Semester 2
Project Report: Iteration 2
Team 8

Team Member	Student ID	Contact No.	Email Address
HO WEI DIAN	A0265562Y	88927943	e1062697@u.nus.edu
CHIN WEN HUI	A0265582W	85002211	e1062717@u.nus.edu

Table of Contents

1. Scope of the Prototype Implementation

In the iteration 1 phase, the prototype successfully fulfils the basic requirements outlined for Iteration 1. It effectively processes SIMPLE programs, handling single procedures with print, read, and assignment statements involving integers and variables. The database design comprehensively encompasses entities such as statements, reads, prints, assignments, variables, constants, and procedures, ensuring adequate support for program analysis. Additionally, it integrates Page Query Language (PQL) support by enabling synonym declaration and facilitating queries with a single Select clause, streamlining interaction for program analysis through PQL queries.

SIMPLE	• Single procedure • Print / read statements • Assignment statement with single integers / variables
Database	• Design entities: statement, read, print, assignment, variable, constant, procedure
PQL	• One declaration of a single synonym • One Select clause • A single synonym in the Select clause

In the iteration 2 phase, the prototype introduces enhancements primarily focused on refining the PQL functionality. It includes lexical token definitions and grammar rules for SIMPLE, incorporating statements, procedures, and expressions. The database design now encompasses essential entities and abstractions like statements, reads, prints, assignments, variables, constants, and procedures. Furthermore, PQL capabilities are expanded, allowing more complex queries with declarations, a Select clause with a single synonym, and optional such that and pattern clauses. These clauses support relationships like Parent and Modifies, as well as partial pattern matching for assignments based on variables or constants. The work is divided into Phase A, targeting basic SIMPLE features and initial PQL functionalities, and Phase B, focusing on advanced PQL features like Parent* and Modifies, alongside refined pattern matching capabilities.

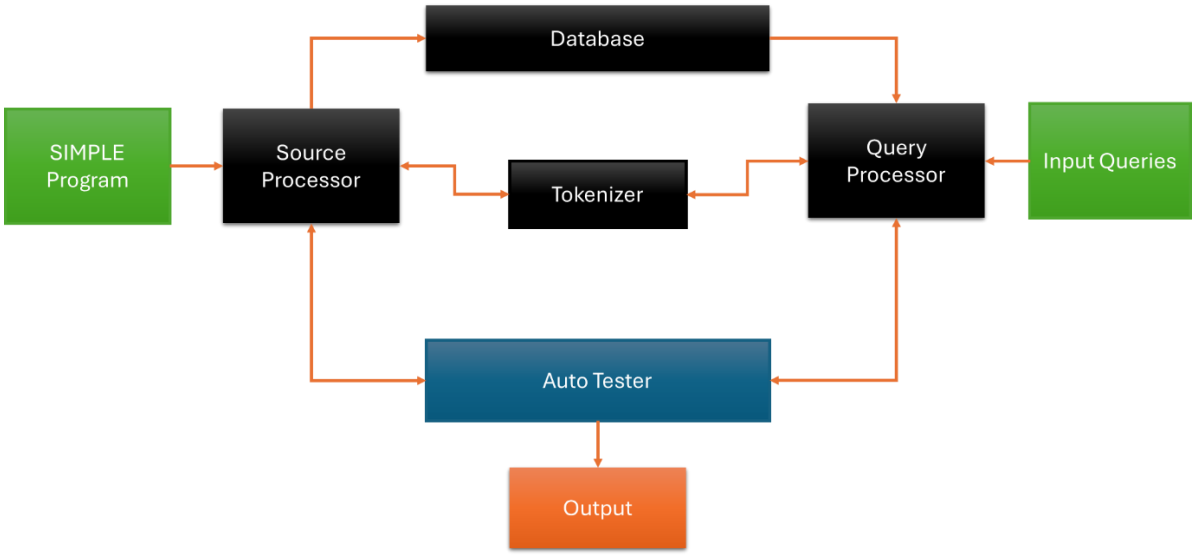
SIMPLE	• +If and while statement • +Assignment statements with expressions
Database	• Design entities: +if, while • Design abstractions: +Parent / Parent* / Modifies for statements (excluding call statements) / Uses for statements (excluding call statements)

PQL	• +Multiple declaration of (possibly multiple) synonyms
	• + One such that clause or one pattern clause
	  ▪ such that clause: +any of the design abstractions in the Database
	  ▪ pattern clause: +partial pattern match with a single variable / constant.

2. SPA Design

2.1 Overview

The diagram provides an overview of each component:



To enable the SPA to effectively handle program queries, the Source Processor initiates the analysis of the source program, which in this prototype is based on SIMPLE source code. The Source Processor employs a tokenizer to dissect the source code, extracting essential tokens. These tokens are then interpreted to identify key program elements like procedures, variables, constants, and statements, among others. For each identified entity, the Source Processor triggers the corresponding insertion method in the Database to populate the relevant data tables.

Implemented using SQLite, the Database comprises various tables to store program entities such as procedures, variables, and statements. Each insertion method appends data to its designated table. Furthermore, the Database furnishes methods to query specific design entities.

Users can leverage the Program Query Language (PQL) to interrogate the program via the SPA. The Query Processor dissects the query string and determines its type (e.g., procedure, statement, variable). Subsequently, it invokes the appropriate Database methods to retrieve pertinent data based on the query type. The retrieved data undergoes post-processing before being presented as query results to the user. These PQL queries are processed by the Source Processor, and the resultant data is presented to the user based on the information stored in the Database.

2.2 Database Design

2.2.1 Database Design Overview

-- Procedure Table

CREATE TABLE procedure (procedureName VARCHAR(255) PRIMARY KEY);

-- Statement Table

CREATE TABLE statement (stmtLine VARCHAR(255) PRIMARY KEY, stmtEntity VARCHAR(255), stmtLHS VARCHAR (255), stmtRHS VARCHAR (255), stmtPostfix VARCHAR (255), parent VARCHAR(255));

-- Variable Table

CREATE TABLE variable (varName VARCHAR(255), varLine VARCHAR(255) REFERENCES statement(stmtLine), varSides VARCHAR(255), PRIMARY KEY (varName, varLine));

-- Constant Table

CREATE TABLE constant (constValue VARCHAR(255) PRIMARY KEY);

Table	Column	Description
procedure	name	Stores the procedure name
statement	stmtLine	Stores the statement number
	stmtEntity	Stores the different types of statements {print, read, assign, if, while}
	stmtLHS	Stores the left-hand sides of the content of an assign statement OR the condition for a “while” or “if” statement
	stmtRHS	Stores the right-hand sides of the content of an assign statement OR NULL for other cases
	stmtPostfix	Stores the postfix of the contents of stmtRHS for the pattern analysis
	parent	Store the parent’s stmtLine

variable	varName	Contains the variable name
	varLine	Contains the statement number that contains this variable
constant	constValue	Contains the constant value

2.2.2 Interaction between SIMPLE and SQL Database

```
-- Insert Procedure

INSERT INTO procedure ('procedureName') VALUES ('procedureNameValue');

-- Insert Statement

INSERT INTO statement ('stmtLine','stmtEntity','stmtLHS','stmtRHS','stmtPostfix')
VALUES ('stmtLineValue', 'stmtEntityValue', 'lhsValue', 'rhsValue', 'postfixRhsValue');

-- Insert Constant

INSERT INTO constant ('constValue') VALUES ('constValue');

-- Insert Variable

INSERT INTO variable ('varName','varLine', 'varSides') VALUES ('varNameValue',
'varLineValue', 'varSidesValue');

-- Update Parent

UPDATE statement SET parent = 'parentValue' WHERE stmtLine = 'stmtLineValue';
```

2.2.3 Interaction between PQL and SQL Database

```
-- Get All Procedures
SELECT * FROM procedure;

-- Get All Constants
SELECT DISTINCT constValue FROM constant;

-- Get All Variables
SELECT DISTINCT varName FROM variable;

-- Get All Statements
```

```
SELECT stmtLine FROM statement;

-- Get Specific Statements (read / print / if / while / assign)
SELECT stmtLine FROM statement WHERE stmtEntity = 'type';

-- Get Parents based on the child’s stmtEntity
SELECT parent FROM statement WHERE stmtEntity = 'childType';

-- Get Parents based on the child’s stmtEntity AND parent’s stmtEntity
SELECT parent FROM statement WHERE stmtEntity = 'childType' AND parent IN
(SELECT stmtLine FROM statement WHERE stmtEntity = 'parentType');

-- Get Parent based on the child’s stmtLine
SELECT parent FROM statement WHERE stmtLine = 'childLine';

-- Get Parent based on the child’s stmtLine AND parent’s stmtEntity
SELECT parent FROM statement WHERE stmtLine = 'childLine' AND parent IN
(SELECT stmtLine FROM statement WHERE stmtEntity = 'parentType');

-- Get Children based on the child’s stmtEntity AND parent’s stmtEntity
SELECT stmtLine FROM statement WHERE stmtEntity = 'childType' AND parent IN
(SELECT stmtLine FROM statement WHERE stmtEntity = 'parentType');

-- Get Children based on the parent’s stmtLine
SELECT stmtLine FROM statement WHERE parent = 'parentLine';

-- Get Children based on the parent’s stmtLine AND child’s stmtEntity
-- SELECT stmtLine FROM statement WHERE stmtEntity = 'childType' AND parent IN
(SELECT stmtLine FROM statement WHERE parent = 'parentLine');

-- Get Pattern
SELECT stmtLine FROM statement WHERE stmtEntity = 'assign';
SELECT stmtLine FROM statement WHERE stmtEntity = 'assign' AND stmtLHS =
‘lhs’;
SELECT stmtPostfix FROM statement WHERE stmtLine = ‘stmtLine’;

-- Get Modifies stmt based on stmtEntity(Assign, Read)
SELECT stmtLine FROM statement WHERE stmtEntity = 'stmtEntity';

-- Get Modifies stmt based on stmtEntity (Container)
```

```
WITH RECURSIVE xx AS (SELECT stmtLine AS s, stmtEntity as st FROM statement
WHERE stmtEntity =  'stmtEntity' UNION ALL SELECT p.stmtLine, p.stmtEntity
FROM statement p JOIN xx ON p.parent = xx.s )SELECT s FROM xx ;

WITH RECURSIVE xx AS (SELECT parent AS p,stmtEntity as st, stmtLine sl FROM
statement WHERE stmtEntity in ('read','assign') UNION ALL SELECT
p.parent,p.stmtEntity, p.stmtLine FROM statement p JOIN xx ON p.stmtLine = xx.p)
SELECT sl FROM xx where st='stmtEntity';

-- Get Modifies variable based on stmtEntity(Assign, Read)
select DISTINCT LHS from statement where stmtEntity = 'stmtEntity';

-- Get Modifies variable based on stmtEntity(Container)
WITH RECURSIVE xx AS (SELECT stmtLine AS s,stmtEntity as st,lhs var FROM
statement WHERE stmtEntity = ' stmtEntity ' UNION ALL SELECT p.stmtLine,
p.stmtEntity, p.LHS FROM statement p JOIN xx ON p.parent = xx.s) SELECT var
FROM xx where st in ('read','assign');

-- Get Modifies variable based on stmtLine
SELECT DISTINCT LHS FROM statement WHERE stmtLine = 'stmtLine' AND
stmtEntity in ('read', 'assign');

-- Get Modifies stmt based on stmtEntity and variable
SELECT DISTINCT stmtLine FROM statement WHERE stmtEntity = 'stmtEntity' AND
LHS = 'variable';

-- Get Modifies stmt based on stmtEntity(Container) and variable
WITH RECURSIVE xx AS ( SELECT parent AS p,stmtEntity as st, stmtLine sl, lhs var
FROM statement WHERE var='variable' UNION ALL SELECT p.parent,p.stmtEntity,
p.stmtLine, p.lhs FROM statement p JOIN xx ON p.stmtLine = xx.p) SELECT sl
FROM xx where st= 'stmtEntity' ;
```

2.3 Source Processor

Source Processor (Iteration 1 and 2)

The Source Processor accepts the source program and invokes the Tokenizer class to break down the source code into vector<string> tokens. A simple while loop is able to iterate through the tokens, identify them and correctly store them into the database accordingly. The function isstringstream is used to convert integers into strings as well as to convert strings to integers as constant, print, statement and assignment requires us to store the line number in the procedure into the database.
Let's break down the key functionalities and understand how the parsing and database population occur:

1. Tokenization and Parsing:
   - Tokenization breaks down a character sequence into meaningful tokens like keywords, identifiers, and literals.
   - The `Tokenizer` class tokenizes the input program, generating a sequence of tokens.
   - Parsed tokens are sent to the `parse` method of the `SourceProcessor` class for analysis and processing.

2. Parsing Functions:
   - The `parse` method of `SourceProcessor` is the entry point for parsing tokens.
   - It delegates parsing tasks to various functions based on the program's structure.
   - Each function handles a specific aspect of syntax or semantics.

3. Procedure Parsing (parseProcedure):
   - Parses procedure definitions, identifying the "procedure" keyword and the procedure name.
   - Ensures proper syntax and inserts parsed procedure names into the database.

4. Assignment Parsing (parseAssignment):
   - Parses assignment statements, extracting the left-hand side (LHS) and right-hand side (RHS) expressions.
   - Converts the RHS to postfix notation and inserts the assignment information into the database.

5. Condition Parsing (parseCondition):
   - Parses conditional expressions within "if" and "while" statements.
   - Extracts the condition expression and ensures correct syntax.
   - Inserts parsed conditions into the database as part of respective statements.

6. Statement Parsing (parseStatement):
   - Parses various statement types like read, print, if, and while statements, along with assignments.
   - Iterates through tokens, identifies statement types, and further parses based on type.
   - Maintains parent-child relationships in the database to reflect program structure accurately.

7. Variable and Constant Parsing (parseVariable, parseConstant):
   - Handles parsing of variable names and constant values encountered in the program.
   - Ensures syntax correctness and inserts parsed information into the database.

8. Parent Tracking (traceParent, parseParent):
   - Tracks and updates parent-child relationships between statements.
   - Maintains a stack (`parentLevel`) to track the current parent statement during parsing.
   - Updates parent statements with child references in the database to represent hierarchical structure accurately.

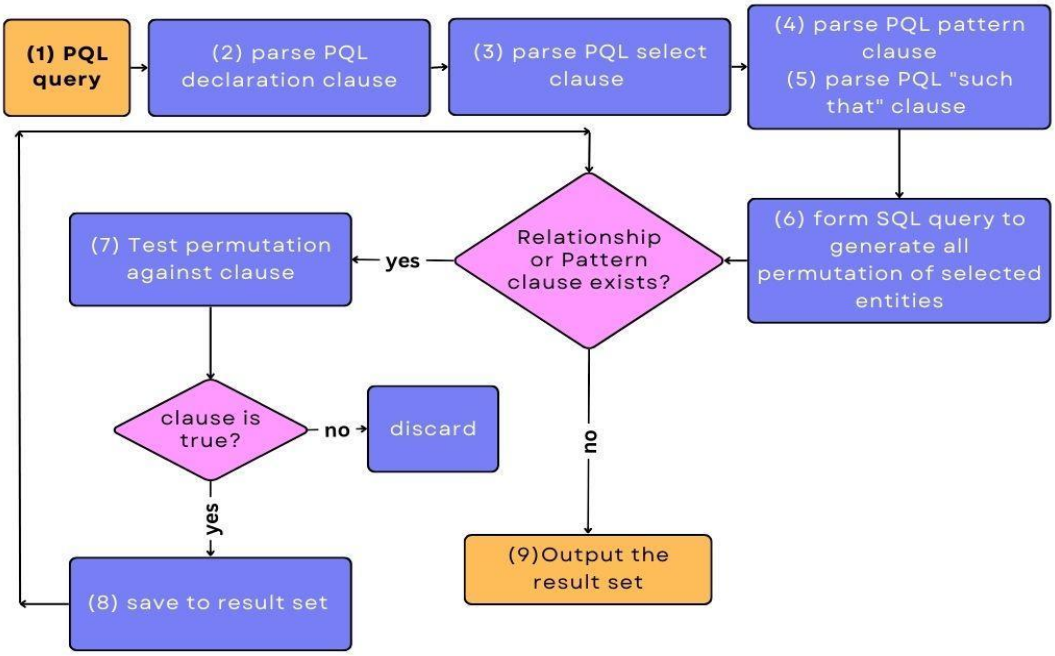
These parser functions systematically analyze input source code, extract essential elements, and store them in a database for further analysis and processing.

2.4 Query Processor

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.



2.4.1 The flowchart of Query Processor

1. Initialization:
The class has a constructor and a destructor, but they are empty, implying that there is no special initialization or cleanup logic needed.

2. Tokenization:

- The `evaluate` method starts by tokenizing the input query string using an instance of the `Tokenizer` class.
- Tokens are extracted from the query string, which includes keywords, identifiers, and other relevant elements.

3. Query Parsing:

- The code iterates over the tokens to identify the type of query being processed and extract relevant information from it.

- It distinguishes between different types of clauses in the query, such as "such that" and "pattern" clauses.
- Information about synonyms, conditions, and patterns is extracted and stored for further processing.

4. Database Interaction:
- The code interacts with a database to retrieve relevant data based on the parsed query.
- Database queries are constructed based on the information extracted from the query string.
- Queries include retrieving procedures, variables, constants, or statements based on specified conditions.

5. Results Processing:
- Retrieved data from the database is stored in a vector called `databaseResults`.
- Depending on the query type and conditions, specific database retrieval methods are called.
- Results are post-processed to fill in the output vector for further analysis or presentation.

6. Output:
- The final step involves copying the database results into the output vector, which can then be used by other components of the system.
- This vector likely contains the results of the query evaluation, which could be displayed to the user or used for further analysis.

```
void QueryProcessor::evaluate(string query, vector<string>& output) {
```

`evaluate` is a method of the `QueryProcessor` class, which takes a query string (`query`) and a reference to a vector of strings (`output`) as input parameters.

```
Tokenizer tk;
vector<string> tokens;
tk.tokenize(query, tokens);
```

- Creates an instance of the `Tokenizer` class named `tk`.
- Declares a vector of strings named `tokens`.
- Calls the `tokenize` method of the `Tokenizer` class to tokenize the `query` string and store the tokens in the `tokens` vector.

```
string selectIdx = "";
vector<vector<string>> suchThatIdx;
```

```
vector<vector<string>> patternIdx;
unordered_map<string, string> entityMap;
vector<string> databaseResults;
```

- Declares variables to store information extracted from the query and the database results:
  - `selectIdx`: Stores the selected synonym (e.g., variable, procedure).
  - `suchThatIdx`: Stores information related to "such that" clauses in the query.
  - `patternIdx`: Stores information related to "pattern" clauses in the query.
  - `entityMap`: Maps synonyms to their types (e.g., variable -> variable).
  - `databaseResults`: Stores the results obtained from the database queries.

```
for (auto i = 0; i < tokens.size(); i++) {
transform(tokens[i].begin(), tokens[i].end(), tokens[i].begin(), [](unsigned char c) { return
std::tolower(c); });
string first = tokens[i];
```

- Iterates through each token in the `tokens` vector.
- Converts each token to lowercase using the `transform` function.
- Stores the lowercase token in the `first` variable.

```
if (selectIdx.empty()) {
	if (regex_match(first, regex(SYNONYM_PATTERN))) {
		entityMap[tokens[i + 1]] = first;
	}
	else if (first == "select") {
		selectIdx = tokens[i + 1];
	}
	continue;
}
```

- Checks if `selectIdx` is empty.
- If empty:
  - Checks if the token matches the `SYNONYM_PATTERN` using regular expressions.
  - If matched, stores the synonym type in the `entityMap`.
  - If the token is "select," stores the next token (synonym) in `selectIdx`.

```
if (first == "such" && tokens[i + 1] == "that") {
	vector<string> declaration;
	string designAbs = tokens[i + 2];
	if (tokens[i + 3] == "*") {
		designAbs += "*";
		i += 1;
```

```
        }
        declaration.push_back(designAbs);
        declaration.push_back(tokens[i + 4]);
        declaration.push_back(tokens[i + 6]);
        suchThatIdx.push_back(declaration);
}
```

- Checks if the current token and the next token form a "such that" clause.
- If yes, extracts information about the clause and stores it in the `suchThatIdx` vector.

```
                else if (first == "pattern") {
                        vector<string> declaration;
                        declaration.push_back(tokens[i + 1]);
                        string lhs = "";
                        string rhs = "";
                        bool comma = 0;
                        for (i = i + 3; tokens[i] != ")"; i++) {
                                if (tokens[i] == ",") {
                                        comma = 1;
                                }
                                else if (comma) {
                                        rhs += tokens[i];
                                }
                                else if (tokens[i] != "\"") {
                                        lhs += tokens[i];
                                }
                        }
                        declaration.push_back(lhs);
                        declaration.push_back(rhs);
                        patternIdx.push_back(declaration);
                }
```

- Checks if the current token is "pattern."
- If yes, extracts information about the pattern clause and stores it in the `patternIdx` vector.

```
        string selectIdxType = entityMap[selectIdx];

        if (suchThatIdx.empty() && patternIdx.empty()) {
                // Handle basic queries without conditions
                // Retrieve data from the database based on the type of the selected synonym.
        }
        else {
```

```
            // Handle queries with conditions (such that or pattern clauses)
        }
```

- Retrieves the type of the selected synonym (`selectIdx`) from the `entityMap`.
- Checks if there are no "such that" or "pattern" clauses in the query.
- If yes, handles basic queries without conditions.
- If not, handles queries with conditions.

```
        else { //Select Clause
        if (!patternIdx.empty()) {
                unordered_map<string, int> count;
                int patternLen = patternIdx.size();
                ////cout << "pattern!" << endl;
                for (int k = 0; k < patternLen; k++) {
                        vector<string>tempResult;
                        string lhs = patternIdx[k][1];
                        string rhs = patternIdx[k][2];
                        bool leftEmpty = 0;
                        bool rightEmpty = 0;
                        string newRHS;
                        bool quote = 0;
                        // pattern a ("g",_"a"_)
                  // pattern a (_, "a"_)
                        if (rhs == "_") {
                                leftEmpty = 1;
                                rightEmpty = 1;
                        }
                        else {
                                for (int a = 0; a < rhs.size(); a++) {
                                        if (rhs[a] == '_') {
                                                if (a == 0)
                                                        leftEmpty = 1;
                                                else if (a == rhs.size() - 1)
                                                        rightEmpty = 1;
                                                continue;
                                        }

                                        if (quote && rhs[a] != "") {
                                                newRHS += rhs[a];
                                        }
                                        quote ^= (rhs[a] == "");
                                }
```

```
                    }
```

Check the clauses inputs for LHS and RHS and return true if Empty.

```
        for (string databaseResult : databaseResults) {
                output.push_back(databaseResult);
        }
```

- Copies the database results to the `output` vector for further processing or display.

This function processes a query string, extracts information about synonyms, conditions, and patterns, interacts with a database to retrieve relevant data, and prepares the results for output.

3. Testing

The test cases for the Static Program Analyzer (SPA) are designed to rigorously evaluate its ability to handle SIMPLE programs and execute queries written in the Page Query Language (PQL).

Iteration 1 Approach

For the SIMPLE program test case, a program named "alpha" is provided, comprising a single procedure. This procedure contains basic programming elements such as assignment, print, and read statements, each involving single integers or variables. By analyzing this program, the SPA's capability to parse and interpret fundamental programming constructs is assessed. It ensures that the SPA can accurately identify and process variables, constants, assignments, prints, reads, and statements within a straightforward program structure. For the PQL test cases, a series of queries are crafted to assess different aspects of the SPA's query processing capabilities.

Source Program

```
procedure alpha {
        num1 = 6999;
        num2 =    88;
        x=num3;
        read x;
        num3 = 10;
        print num2;
        x=num1;
        print x;
```

```
        print num;
        print y  ;
        read num1;

}
```

Queries

```
1 - Procedure
procedure p;
Select p;
alpha
5000
2 - Variable
variable v;
Select v
num, num1, num2, num3, x, y
5000
3 - Constant
constant c;
Select c
6999, 88, 10
5000
4 - Assignment
assign a;
Select a
1, 3, 5, 7
5000
5 - Print
print pr;
Select pr
6, 8, 9, 10
5000
6 - Read
read rd;
Select rd
4, 11
5000
7 - Statement
stmt s;
Select s
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
5000
```

Iteration 2 Approach

Test cases were structured to cover all aspects of the expanded functionality.

Based on the requirement, three set of SIMPLE programs (alpha, beta, gamma), each containing if and while statements along with assignment statements involving expressions were created. These programs were designed to encompass various scenarios of control flow, including nested if-else constructs and complex arithmetic expressions. The test cases help to validate the correct construction and utilization of design abstractions in the database, such as Parent, Parent*, Modifies, and Pattern. By populating the database with data from these programs, we verified that each design abstraction was accurately identified and stored.

PQL queries were then formulated to leverage the newly added design abstractions, encompassing relationships like Parent, Parent*, Modifies, and Uses. These queries cover different combinations of clauses, such as such that and pattern, ensuring comprehensive testing of our prototype's query evaluation capabilities.

Source Program - a

```
procedure alpha {
        x = y;
        a = b + 1;
        print k;
        while (x < a) {
                print a;
                while (a < 1) {
                        a = d + c * b + 1;
                        print c;
                        read b;
                }
                if (y > a) then{
                        read y;
                        print k;
```

```
                } else {
                        print a;
                }
        }
        print s;
        s = s + y;
        read y;
}
```

Queries – a

```
1 - Select all read statement that Modifies something
variable v; read r;
Select r such that Modifies(r, v)
3, 5, 8, 12, 13, 14
5000
2 - Select while statement that Modifies certain variable
while w; variable v;
Select w such that Modifies(w, "y")
11
5000
3 - Select the varible modified in a specific statement
variable v1, v;
Select v1 such that Modifies(9, v)
b
5000
4 - Select all child assignment statements
variable v1; assign a;
Select a such that Parent*(_, a)
7
5000
5 - Select all statements with parent
stmt s1,s;
Select s1 such that Parent(s, s1)
6,7,8,9,10,11,12,13
5000
6 - Select all read statements within a while loop
read r; while w;
Select r such that Parent* (w, r)
9,11
5000
```

```
7 - Select all assignment statements with s+y on RHS
assign a;
Select a pattern a(_, _"s + y"_)
15
5000
8 -Select all assignment statements with a on LHS and b on RHS
assign a1;
Select a1 pattern a("a", _"b"_)
2
5000
9 -Select all assignment statements with b+1 on RHS
assign a1;
Select a1 pattern a(_, _"b + 1"_)
2
5000
```

Source Program - b

```
procedure beta {
        read x;
        read y;
        read h;
        print n;
        while (y < q) {
                print y;
                if (y > 1) then {
                        print g;
                        read f;
                } else {
                        print k;
                }
        }
        read z;
        x = r + 2;
        print a;
        print b;
```

```
        if (x > 1) then {
                read d;
        } else {
                read s;
        }
        read m;
}
```

Queries - b

```
1 - Gets all statements
stmt s;
Select s
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18
5000
2 - Gets all read statements
read r;
Select r
1, 2, 3, 9, 11, 16, 17, 18
5000
3 - Gets all print statements
print p;
Select p
4, 6, 8, 10, 13, 14
5000
4 - Gets all while statements
while w;
Select w
5
5000
5 - Gets all procedures
procedure p;
Select p
q
5000
6 - Gets all variables
variable v;
Select v
x, y, h, n, g, f, k, q, z, a, b, d, s, m, r
5000
7 - Select all read statement that Modifies something
```

```
variable v; read r;
Select r such that Modifies(r, _)
1, 2, 3, 9, 11, 16, 17, 18
5000
8 - Select read statements that Modifies a certain variable
variable v1, v; read r;
Select r such that Modifies(r, "y")
2
5000
9 - Select all while statements that Modifies something
while w; read r; variable v;
Select w such that Modifies(w, v)
5
5000
10 - Select all children of while statements
while w; stmt s;
Select s such that Parent(5, s)
6, 7
5000
11 - Select all children and grandchildren of while statements
while w; stmt s;
Select s such that Parent*(5, s)
6, 7, 8, 9, 10
5000
12 - Select all variables being Modified by a Read
variable v; read r;
Select v such that Modifies(r, v)
x, y, h, f, z, d, s, m
5000
13 - Select all assignment statements that has r on RHS
assign a;
Select a pattern a(_, _"r"_)
12
5000
14 - Select all assignment statements that has x on LHS
assign a;
Select a pattern a("x", _)
12
5000
```

Source Program - c

```
procedure gamma {
        while (r < s) {
```

```
                x = y;
                i = i;
                i = 5;
                while (a > 65) {
                        read a;
                        print b;
                        z = 5 + i * x;
                        if (count < 2) then {
                                i = a;
                                a = 1;
                        } else {
                                y = 3;
                                i = i;
                                i = 5;
                        }
                }
        }
}
```

Queries - c

```
1 - Select Parent for line 6
parent t;
select t such that Parent(t,6)
5
5000
2 - select child assignments for parent if
assign a; if i;
select a such that parent(i,a)
10,11,12,13,14
5000
3 - select all child assignments for parent while
assign a; while w;
select a such that parent(w,a)
2,3,4,8
```

```
5000
4 - select parentT of line 5
parentT t;
select t such that parent*(t,5)
1
5000
5 - select all assignment statements that modifies something
assign a; variable v;
select v such that modifies(a,v)
2,3,4,6,8,10,11,12,13,14
5000
6 - select all read variables modified in statements
read r; variable v;
select v such that modifies(r,v)
6
5000
7 - select variable modified in line 10
variable v;
select v such that modifies(10,v)
i
5000
8 - select all variables modified in if statements
if i; variable v;
select v such that modifies(i,v)
i,a,y
5000
9 - Select pattern with i*x on LHS
assign a;
Select a pattern a(_,_"i*x"_)
8
5000
10 - select pattern with i on LHS
assign a;
Select a pattern a("i",_)
3,4,10,13,14
5000
11 - select pattern with i on LHS and 5 on RHS
assign a;
Select a pattern a("i",_"5"_)
4,14
5000
```