Static Program Analyzer (SPA)
Project Iteration 1 Report

KIANG SIONG BOON (A0245446B)
LEE WEE LUN (A0245304N)

Table of Contents

Project Background

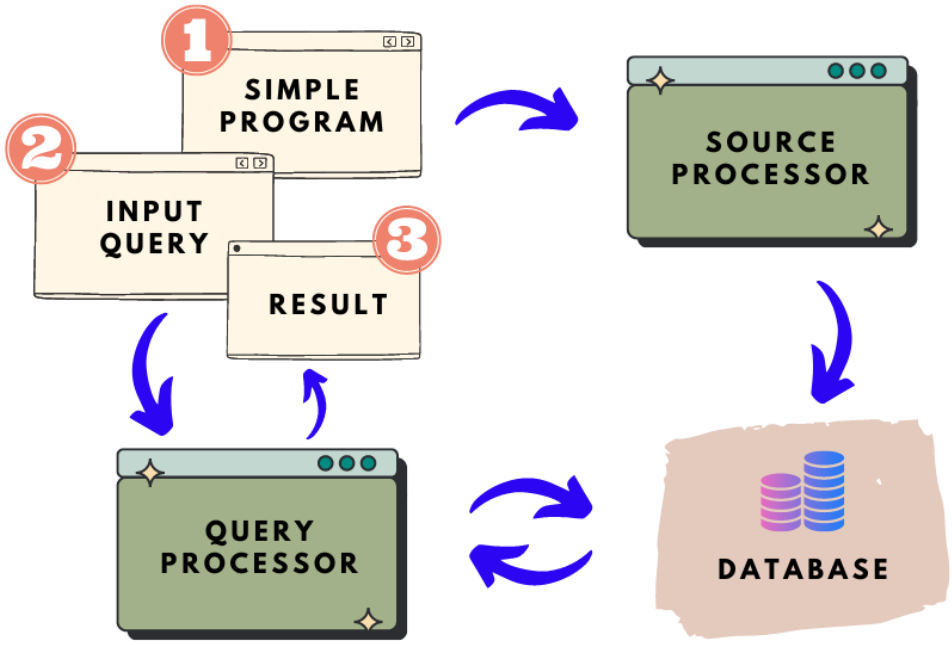
There is a high demand for IT experts in the modern digital era, particularly software developers. These professionals are in charge of developing, testing, and maintaining the software applications utilized by businesses and organizations all around the world.

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

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)

Prototype Database Design

Database Diagram

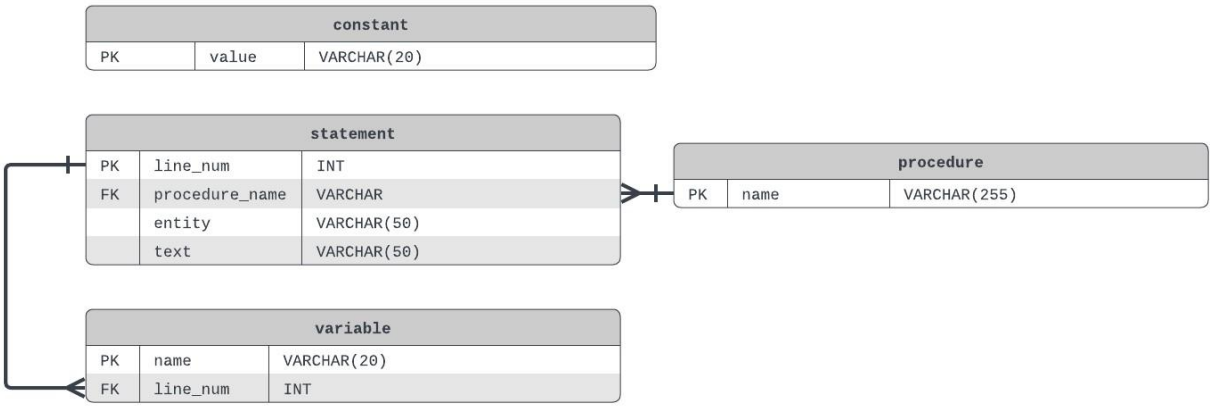


Table Description

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
	line_num	Contains the statement number that contains this variable
constant	value	Contains the constant value

Table Relationship

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

Database Design Rational

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.

Example of PQL to SQL

PQL	SQL
procedure p; select p	SELECT name FROM procedure;
variable v; select v	SELECT name FROM variable;

SPA Implementation

Source Processor

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. If the word is “procedure”
    a. Get the procedure name at index + 1, and insert it into the procedure table.
2. If the word is “while”
    a. Perform a while loop to process the condition portion of this While statement. The terminating condition is when we encounter “}”.
    b. Within the while loop, we use regex match to insert the correct tokens into the Constant table or Variable table.
    c. At the end, we insert this While statement into the Statement table as a “while” statement type.
3. If the word is “if”
    a. Perform a while loop to process the condition portion of this “if” statement. The terminating condition is when we encounter the keyword “then”.
    b. Within the while loop, we use regex match to insert the correct tokens into the Constant table or Variable table.
    c. At the end, we insert this “if” statement to the Statement table as an “if” statement type.
4. If the word is “else”
    a. For Iteration 1’s scope, it does nothing.
5. If the word is “=”
    a. Insert the Right Hand Side variable into the Variable table
    b. Perform a while loop to process the Left Hand Side assignment statement. The terminating condition is “;”.
    c. Within the while loop, we use regex match to insert the correct tokens into the Constant table or Variable table.
    d. At the end, we insert this Assign statement into the Statement table as an “assign” type.
6. If the word is “read”, “print” or “call”
    a. We insert the statement into the Statement table as the respective type.
    b. If the word is “read” or “print”.
        i. We insert the variable into the Variable 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 Constans 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

Query Processor

We created database functions that the Query Processor could call for a particular synonym.

We utilized the given tokenizer to tokenize the PQL query. Then, we extracted the synonym from the token array, and called the corresponding database function. Example as follows

PQL Input	**Extracted Keywords**	**Database Class Function**	**SQL**
constant c; select c;	Synonym = “constant”	getConstant()	SELECT * FROM constant;
stmt s; select s	Synonym = “stmt”	getStmt(stmt)	SELECT * FROM stmt WHERE entity = ‘stmt’
print p; select p	Synonym = “print ”	getStmt(print)	SELECT * FROM stmt WHERE entity = ‘print’

Testing

Approach

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, ifififif, callcallcall
```