CS3203 Software Engineering Project

Team 22

Project Iteration 3 Report

Full Name	Matric Number	Email Address	Phone number
LIM JIA HAO	A0168482U	e0176915@u.nus.edu	90622468
LOW ZHANG XIAN	A0185078U	e0316053@u.nus.edu	93864342
NGEOW XIU QI	A0168352B	e0176785@u.nus.edu	91517831
OLIVIA YU WANZHI	A0173421N	e0202991@u.nus.edu	81631368
TAN MEI YEN	A0169984A	e0191532@u.nus.edu	94318367
TEOH ZHIHUI	A017321W	e0200891@u.nus.edu	81236150

1. Achievements in this iteration

The system has been developed to meet the project requirements. Our system has two main functions:

1. To parse and validate, and extract and store information regarding a SIMPLE source program
2. To answer queries, formatted in the PQL query language, regarding the SIMPLE source program

Our SPA system comprises 3 main components. The **Front-End**, which consists of the **Parser** and the **Design Extractor**, the **PKB**, and the **PQL**, which consists of the **Query Preprocessor** and the **Query Evaluator.** More details on these subcomponents can be found in Section 3.

The SPA system takes in a SIMPLE source program, which contains multiple procedures (Iteration 2 and 3), as opposed to just a single procedure previously (Iteration 1). A procedure consists of statements of different types including: `assign`, `read`, `print`, `call`, `if` and `while`. The system extracts and stores relationships among program design entities including: `Follows/Follow*`, `Parent/Parent*`, `Uses` and `Modifies` (Iteration 1), `Calls/Calls*`, `Next/Next*`, `Uses` and `Modifies` across multiple procedures (Iteration 2), `Affects/ Affects*` (Iteration 3), as well as `NextBip/NextBip*` and `AffectsBip/AffectsBip*` (Extension).

The PKB would then store the above mentioned design entities and relationships among program design entities pulled out by the Design Extractor to be used for query processing.

Using the information about the different design entities and relationships between program design entities, the SPA system is then able to answer queries formatted in Program Query Language (PQL), and return the answer to the query. The QueryPreprocessor is able to evaluate and parse queries with a *select* clause, and one (Iteration 1) or more (Iteration 2) of the following clauses: `such that`, `pattern`, (Iteration 1) and `with` (Iteration 2). `'such that' relRef ('and' relRef)*`, `'pattern' pattern ('and' pattern)*`, `'with' attrCompare ('and' attrCompare)*` clauses will be parsed into separate clauses (Iteration 2) by the QueryPreprocessor. It is also able to handle design entities such as `stmt`, `prog_line`, `read`, `print`, `call`, `while`, `if` and `assign`. Additionally, it is capable of supporting all relationships up to Iteration 3.

The QueryEvaluator will then evaluate the parsed query, and is capable of supporting `Follows`, `Parent`, `Follows*`, `Parent*`, `Uses` and `Modifies` relationships for `such that` clauses (Iteration 1), as well as the newly-added `Calls`, `Next`, `Calls*`, `Next*`, `Uses` and `Modifies` across multiple procedures (Iteration 2) and `Affects`, `Affects*` as well as `NextBip`, `NextBip*`, `AffectsBip` and `AffectsBip*` (Iteration 3). For `pattern` clauses, the Query Evaluator can support `assign` (Iteration 1), `if` and `while` patterns (Iteration 2). It is also able to handle `prog_line` (Iteration 3).

The Query Evaluator can then return an answer that is either of the form of synonyms (Iteration 1), attributes, tuples or booleans (Iteration 2). To support queries that select an attribute, or having a `with` clause, the Query Preprocessor and Query Evaluator have been developed to handle the following attributes for synonyms: `procName`, `varName`, `value`, `stmt#` (Iteration 2). The Query Evaluator now has an optimization feature to efficiently evaluate queries (Iteration 3).

2. Project Plans

This system was developed as a team iteratively and incrementally. The breadth-first iteration was broken down into mini-iterations of a week each to achieve a working system at the end of the iteration. For the tasks and activities, we initially evenly divided the 6 members, each working on a major component of the SPA program, into 3 different groups with members working on interrelated components grouped together. The general overview of the team are as follows:

- Front-end: Olivia, Zhang Xian
- PKB: Jia Hao, Xiu Qi
- PQL: Mei Yen, Zhi Hui

2.1 Project Iteration Plans

The overall workflow for the team for Iterations 1, 2 and 3 are as follows:

2.1.1 Iteration 1

Team Members	Iteration 1					
	Parser	Design Extractor	PKB	Query Preprocessor	Query Evaluator	System Testing
Olivia	X					X
Zhang Xian		X				X
Jia Hao			X			X
Xiu Qi			X			X
Mei Yen				X		X
ZhiHui					X	

Table 2.1.1.1 Responsibilities for Iteration 1

During the iteration 1 phase, our team was split into 3 teams of 2, each team covering each major component. As the PKB team was done with their component earlier, the PKB team members shifted to assisting with other components such as PQL and creating system tests. The table below shows the detailed breakdown of tasks during this iteration. For further details of each task, please refer to Table 2.1.2.3.

<table>
<tr><th>Team Members</th><th colspan="17">Iteration 1</th></tr>
<tr><th></th><th colspan="2">Week 2</th><th colspan="2">Week 3</th><th colspan="2">Week 4</th><th colspan="2">Week 5</th><th colspan="2">Week 6</th><th colspan="7">Recess</th></tr>
<tr><td>Olivia</td><td>1.1</td><td>1.2</td><td>1.3</td><td>2.3</td><td colspan="2">2.4</td><td colspan="2">2.5</td><td colspan="2">3.2</td><td>3.6</td><td>2.11</td><td>4.1</td><td>4.2</td><td>3.13</td><td>3.14</td><td>3.1</td></tr>
<tr><td>Zhang Xian</td><td>1.1</td><td>1.2</td><td colspan="2">1.3</td><td colspan="2">2.6</td><td colspan="4">2.7</td><td>3.3</td><td>3.13</td><td>3.14</td><td>3.9</td><td>3.1</td><td>4.1</td><td>4.2</td></tr>
<tr><td>Jia Hao</td><td>1.1</td><td>1.2</td><td>1.3</td><td colspan="2">2.2</td><td colspan="2">2.9</td><td>3.1</td><td>2.9</td><td>2.15</td><td>3.6</td><td>3.9</td><td>3.1</td><td>3.12</td><td>3.14</td><td>4.1</td><td>4.2</td></tr>
</table>

Xiu Qi	1.1	1.2	1.2	1.3	2.9			3.1	4.1	4.2	3.12							
Mei Yen	1.1	1.2	1.3	2.14	2.8			3.4	4.1	4.2	3.11	3.12	3.13			3.14		
ZhiHui	1.1	1.2	1.3	2.14	2.16	2.1	2.16				3.5	3.11	3.7	3.13	3.14	3.1	4.1	4.2

Table 2.1.1.2 Activities for Iteration 1

2.1.2 Iteration 2

Team Members	Iteration 2				
	Extension of Parser/DE for new relationships	**Extension of PKB for new relationships**	**Extension of Query Preprocessing for new clauses**	**Extension of Query Evaluator for new clauses**	**System Testing**
Olivia	X				X
Zhang Xian	X				
Jia Hao		X			
Xiu Qi					X
Mei Yen			X		X
ZhiHui				X	

Table 2.1.2.1 Responsibilities for Iteration 2

Team Members	Iteration 2															
	Week 7				Week 8					Week 9						
Olivia	2.2	2.9	2.21	2.12	2.13		3.20	3.21	2.25	4.4			4.5		3.14	3.15
Zhang Xian	3.18		2.21		2.22		3.3	3.21	3.22	4.4			4.5		3.14	3.15
Jia Hao	3.2				2.9		3.1			2.26		2.27	4.4	4.5	3.14	3.15
Xiu Qi	4.6	3.22				3.13				3.1			4.4	4.5	3.14	3.15
Mei Yen	3.18				2.18	2.23	3.17	2.23	3.17	2.23	3.17	3.11	4.4	4.5	3.14	3.15
ZhiHui	3.18				2.19	2.24	3.17	4.4	2.24	2.24	3.17	3.11	3.22	4.5	3.14	3.15

Table 2.1.2.2 Activities for Iteration 2

We continued with this arrangement for Iteration 2, but having learnt from our mistakes of the development of components being too isolated in Iteration 1, we were more flexible with the allocation of manpower this iteration: members freed up (such as those working on the PKB and the Front-End Parser) were reallocated to other components where more manpower was needed. For example, Xiu Qi, who was initially working on the PKB in Iteration 1, worked on System Testing (an aspect of development we neglected previously) in Iteration 2 and 3. Another example would be Olivia, who was working on the Front-End Parser in both Iteration 1 and the start of Iteration 2, but was later moved to help Design Extractor and Query Preprocessor. For a further breakdown of the tasks in Table 2.1.2.2, please refer to Table 2.1.2.3 below.

Task Details

Planning Phase	1.1	Familiarization with project SPA requirements
	1.2	Plan APIs for SPA
	1.3	Plan design components
Development	2.1	Develop basic PKB Data Structures
	2.2	Implement PKB APIs
	2.3	Develop and implement Parser structure and logic
	2.4	Develop Parser tokenization and validation
	2.5	Develop Parser to PKB Population
	2.6	Develop Design Extractor (Follows*, Parent*)
	2.7	Develop Design Extractor(Uses, Modifies)
	2.8	Develop Query Parser for Basic SPA
	2.9	Create PKB Tables
	2.10	Implement PQL2PKB interface
	2.11	Shunting Yard Implementation (For Front-End Parser and PQL Evaluator)
	2.12	Develop Design Extractor (Calls, Calls*)
	2.13	Develop Design Extractor (Next)
	2.14	Develop PQL Components
	2.15	Create PKB Object and implement methods for other components
	2.16	Develop Query Evaluator for Basic SPA
	2.17	Integrate Query Parser with Query Evaluator
	2.18	Extend Query Parser (Calls, Calls, Next, Next*)
	2.19	Extend Query Evaluator (Calls, Calls, Next, Next*)
	2.20	Extend Parser for Multiple procedures, recursive and calls to non-existing procedures
	2.21	Extend Design Extractor (Uses, Modifies) Multiple procedures
	2.22	Develop Design Extractor (Next*)
	2.23	Develop Query Parser for Advanced SPA
	2.24	Develop Query Evaluator for Advanced SPA
	2.25	Assist PQL Expression Parsing
	2.26	Extend API for PQL2PKB Interface
	2.27	Assist Query Evaluator "with" clause

Testing	3.1	Unit Test PKB
	3.2	Unit Testing Parser
	3.3	Unit Testing Design Extractor
	3.4	Unit Test Query Parser (Basic SPA)
	3.5	Unit Test Query Evaluator (Basic SPA)
	3.6	Integration Test for Parser - PKB
	3.7	Integration Test for Query- QueryEvaluator - PKB
	3.8	Integration Test Parser (PKB Population)
	3.9	Integration Test Design Extractor - PKB
	3.10	Integration Test PKB - PQL
	3.11	Integration Testing for Query Parser and Query Evaluator
	3.12	Develop System Testing sources and queries (Easy, Medium, Hard)
	3.13	Debugging for Iteration 1
	3.14	System Testing
	3.15	Debugging for Iteration 2
	3.16	Unit Testing Query Parser (Advanced SPA)
	3.17	Unit Testing Query Evaluator (Advanced SPA)
	3.18	Debugging for Iteration 1 Resubmission
	3.19	Optimise tables
	3.20	Unit Testing Front End
	3.21	Integration Test Front End
	3.22	Develop System Testing source and queries for specific clauses
Documentation	4.1	Documentation of development plan
	4.2	Documentation of design and tests
	4.3	Report proposal for SPA extension
	4.4	Updated documentation of existing implementations
	4.5	Update Documentation for Iteration 2
	4.6	Read up on PQL

Table 2.1.2.3 Task Details

2.1.3 Iteration 3

Team Members	Iteration 3						
	NextBIP*, AffectsBIP*	NextBIP, AffectsBIP	Extension of PKB for new relationships	Extension of Query Preprocessing for new relationships	Extension of Query Evaluator for new relationships	Optimization	System Testing
Olivia	X		X				
Zhang Xian		X	X				
Jia Hao			X			X	X
Xiu Qi							X
Mei Yen				X		X	X
ZhiHui					X	X	

The activities plan for Iteration 3 is as follows:

<table>
<tr><th>Team Members</th><th colspan="10">Iteration 3</th></tr>
<tr><th></th><th colspan="2">Week 10</th><th colspan="3">Week 11</th><th colspan="5">Week 12</th></tr>
<tr><td>Olivia</td><td rowspan="2">FE1</td><td>FE3</td><td colspan="3" rowspan="2">FE4</td><td colspan="2" rowspan="2">FE5</td><td rowspan="2">ST9</td><td rowspan="2">ST10</td><td rowspan="2">Doc1</td></tr>
<tr><td>Zhang Xian</td><td>FE2</td></tr>
<tr><td>Jia Hao</td><td>PKB1</td><td>PQL1</td><td>PKB2</td><td colspan="2">PQL6</td><td>PKB2</td><td colspan="2">ST9</td><td>ST10</td><td>Doc1</td></tr>
<tr><td>Xiu Qi</td><td colspan="2">ST1</td><td>ST2</td><td colspan="2">ST3</td><td>ST3</td><td>ST4</td><td>ST6</td><td>ST9</td><td>Doc1</td></tr>
<tr><td>Mei Yen</td><td>PQL2</td><td>PQL1</td><td>PQL3</td><td>PQL4</td><td>PQL5</td><td colspan="3">ST7</td><td colspan="2">Doc1</td></tr>
<tr><td>Zhi Hui</td><td>PQL7</td><td>PQL1</td><td colspan="3">PQL8</td><td>PQL9</td><td>PQL10</td><td>ST9</td><td>ST10</td><td>Doc1</td></tr>
</table>

The corresponding details of each task are in the table below:

Front End	FE1	Planning of Extensions (AffectsBip, NextBip, NextBip*, AffectsBip*)
	FE2	Implemented Affects
	FE3	Implemented Affects*
	FE4	Implemented NextBip and NextBip*
	FE5	Implemented AffectsBip and AffectsBip*
PKB	PKB1	PKB Refactoring
	PKB2	Added tables for Extensions(AffectsBip, NextBip, NextBip*, AffectsBip*)
PQL	PQL1	Planning of optimization for PQL queries

	PQL2	Extended Query Parser to handle Affects, Affects* and Extensions (AffectsBip, NextBip NextBip*)
	PQL3	Integration testing of Query Evaluator and Query Parser
	PQL4	Implemented Optimizer class
	PQL5	Implemented static scoring for Optimizer
	PQL6	Implemented dynamic scoring for Optimizer
	PQL7	Extend Query Evaluator to handle Affects, Affects*
	PQL8	Extend Query Evaluator to handle Extensions (AffectsBip, NextBip, NextBip*)
	PQL9	Implement Optimizer Grouping algorithm
	PQL10	Select Tuple refactoring
Documentation	Doc1	Documentation for respective components
System Test	ST1	Created Affects and Affects* source
	ST2	Created Affects and Affects* queries
	ST3	Created Stress test source
	ST4	Created Stress test queries
	ST5	Created Iteration 3 Easy/Med/Hard source
	ST6	Created Iteration 3 Easy/Med/Hard queries
	ST7	Created Extensions Complex/Extreme source and queries
	ST8	Touch up on Iteration 2 queries
	ST9	System test evaluation
	ST10	Debugging for System Testing

In Iteration 3, the team mostly followed the same team formation as before, with the exception of Jia Hao who was mostly tasked to assist with the optimization of the Query Evaluator with Zhi Hui and then later, together with Mei Yen and Xiu Qi, to assist with the creation of system test cases and evaluating queries. Olivia and Zhang Xian continued working on the extensions for this iteration.

3. SPA Design

3.1 Overview

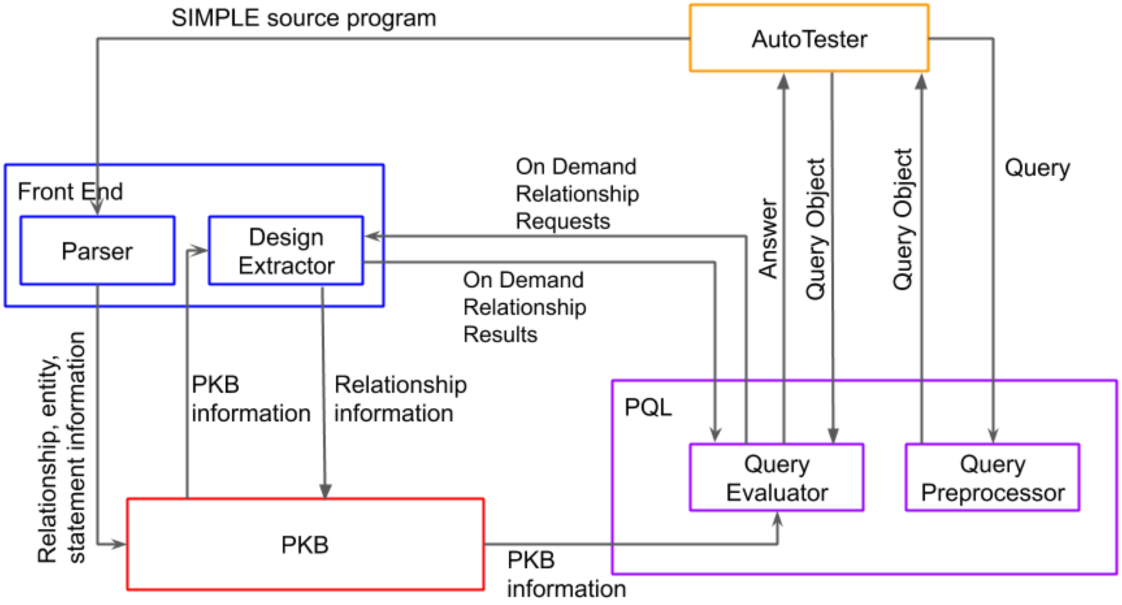


Figure 3.1.1. Design Overview of SPA Program

The SPA system has been split into three major components: **Front-End**, **PKB**, and **PQL**. The **Front-End** and **PQL** components have been further split into two smaller components. **Front-End** encompasses the **Parser** and **Design Extractor** components, and **PQL** encompasses the **Query Preprocessor** and **Query Evaluator** components.

The **Parser** takes in a SIMPLE source program. The component will then tokenize and validate the program according to the SIMPLE language rules, extract design entities such as `statement`, `variable`, `procedure`, `constant` and simple relationships between program design entities, before finally storing them via API calls to the **PKB**.

The following relationships are extracted by the **Parser**:

- `Parent`
- `Follows`
- `Calls/Calls*`

The **Design Extractor** will perform extraction for more complex relationships between design entities by accessing relevant entries in the **PKB**, evaluating the relationships and storing them. On top of this, the **Design Extractor** will perform extraction for `Next*/Affects/Affects*` relationships on demand from the **Query Evaluator**, with the extracted relationships being stored in a cache in the **Design Extractor**.
The following relationships are extracted by the **Design Extractor**:

- `Parent*`

- `Follows*`
- `Uses (across multiple procedures)`
- `Modifies (across multiple procedures)`
- `Next`
- `Next* (on demand)`
- `Affects/Affects* (on demand)`
- `NextBip/NextBip*`
- `AffectsBip/AffectsBip*`

The **PKB** stores the relevant information extracted from the program via the **Parser** and **Design Extractor** components in the form of Tables and Maps. These data structures are chosen to store the information over an AST for the sake of efficiency in accessing the information.

The **Query Preprocessor** takes in a query string. The component will then parse the string and validate it to ensure the query is valid, whether it is syntactically or semantically valid.

The following relevant information are extracted from the query to be stored in a `Query` object:

- Declared synonyms
- Selected synonyms and their attributes
- `BOOLEAN`
- Type of clauses
- Design abstractions
- Design entities
- References
- Reference attribute types

The **Query Evaluator** takes in a `Query` object from the query preprocessor and evaluates the correct answer, making calls to the **PKB** for the necessary information required. The Query Evaluator is capable of handling:

- Relationships
  - `Follows/Follows*`
  - `Parent/Parent*`
  - `Uses`
  - `Modifies`
  - `Calls/Calls*`
  - `Next/Next*`
  - `Affects/Affects*`
  - `AffectsBip/AffectsBip*/NextBip/NextBip*`
- Clauses
  - `such that`
  - `with`
  - `pattern`
    - `assign`
    - `if`
    - `while`

It can return the answer in either of the form of synonyms, attributes, tuple or booleans. The answer will then be formatted and returned to the Autotester.

3.2 Design of SPA Components

3.2.1 Front-End Parser Design

The **Front-End Parser** is responsible for:

1. Parsing and validating the entered SIMPLE program to ensure that it is syntactically correct, i.e. follows all the defined language rules
2. Populating the **PKB** with design entities through the **PKB** API
3. Extracting simple relationships between program design entities and storing them in the **PKB** through the **PKB** API

We implemented a top-down, recursive descent **Front-End Parser**. The parsing and validation of the entered SIMPLE program as well as the populating of the **PKB** with design entities takes place concurrently. Finally, the extraction of simple relationships between program design entities will be done.

Should an error be encountered, the **Front-End Parser** will print an error message indicating the nature of the error and the current line. Execution will be terminated immediately.

Sample SIMPLE Source Program: `FragFood`

The following SIMPLE source program, `FragFood`, will be used to illustrate the workings of the parser throughout the parsing, validation and population of PKB stages.

```
    procedure FragFood {
1     HungrY=1;
2     f00d = 0 ;
3     read          appetitet0d4y;
4     print restaurants   ;
5     while ((availability==0)||(availability<=cravings))
      {
6      print availability
       ;
7      money = (( salary - taxes)/ 10) + (allowance * 2) % 1;
      }
    }
```

SIMPLE Source Program 3.2.1.1 `FragFood`

Tokenizing

The parser reads from the input file stream directly, retrieving token by token. Permitted tokens are as follows:

Token Type	Examples
Punctuation	>, <, >=, <=, ==, !=, +, -, /, *, %, &&, ||, !, (,)
Alphabetic	a, alpha
Numeric	1, 500, 1000000
Alphanumeric	a1, a1alpha
End-Of-File Indicator	EOF

Table 3.2.1.2 Permitted Front-End Parser Tokens

White-space characters are ignored up till the first non-white-space character. If an end-of-file is encountered, the string “EOF” is returned. Otherwise, depending on the character type, certain actions are taken:

1. If the character is a valid punctuation character, return the character
2. If the character is alphanumeric, characters continue to be retrieved and appended until the first non-alphanumeric character is encountered

Erroneous tokens include invalid punctuation (“#”, “$”, “\”, etc.), control characters and tokens violating SIMPLE grammar rules (“2twohundred”, an alphanumeric sequence starting with a digit).

Illustration of Tokenization

With reference to `FragFood`, the following tokens will be extracted from each line:

Line Number	Tokens (in order)
Procedure Line	procedure FragFood {
1	HungrY = 1 ;
2	F00d = 0 ;
3	read appetitet0d4y ;
4	print restaurants ;

<table>
<tr><td>5</td><td>while
(
(
availability
=
=
0
)
|
|
(
availability
<=
cravings
)
)
{</td></tr>
<tr><td>6</td><td>print
availability
;</td></tr>
<tr><td>7</td><td>money
=
(
(
salary
-
taxes
)
/
10
)
+
(
allowance
*
2
)
%
1
;
}
}
EOF</td></tr>
</table>

We can observe that such a tokenization method allows the parser to handle inconsistent white-space characters, such as in:

- Line 3: `read          appetitet0d4y;`
- Line 4: `print restaurants   ;`
- Line 6: `print availability\n;`

At the same time, the parser is able to correctly parse expressions where white-space characters are minimal to none, such as in Line 1: `HungrY=1;`

Parsing and Validation

Parsing of Program (`parseProgram`)

The parser repeatedly calls `parseProcedure` to parse and validate the procedures in the SIMPLE program, until an error or EOF (indicated by the `parseProcedure` return type) is encountered.

Parsing of Procedure (`parseProcedure`)

The parser validates that the procedure keyword “procedure” is found and followed by a valid procedure name (alphanumeric sequence starting with a letter). Then, it calls `parseStatementList` to parse and validate its statement list.

Parsing of Statement List (`parseStatementList`)

The parser validates that the statement list begins with an opening brace. Then, until a closing brace is encountered, the parser repeatedly retrieves the first token of each statement and calls the corresponding function to parse the statement. For example, for a print statement, `parsePrintStatement` would be called. Thereafter, it validates that the statement list ends with a closing brace. The statement list is also checked to ensure it is not empty.

Parsing of Read, Print, Call Statement (`parseReadStatement`, `parsePrintStatement`, `parseCallStatement`)

The parser validates that the statement begins with the appropriate keyword (“read”, “print” or “call”) and is followed by a valid variable name or procedure name before finally ending with a semicolon.

Depending on the statement type, the appropriate table is populated.

- For a Read statement, the line number and the variable being read is stored in PKB `ReadTable`.
- For a Print statement, the line number and the variable being printed is stored in PKB `PrintTable`.
- For a call statement, the line number and the procedure being called is stored in PKB `CallTable`. Additionally, the parser validates that no **recursive call** is made by checking that the program being called is not the same as the current program being parsed.

Parsing of Assign Statement (`parseAssignStatement`)

The parser validates that the statement begins with a valid variable name and is followed by an equals sign “=”. Then, it calls `parseExpression` to parse and validate the expression on the right hand side of the equality sign.

Parsing of Expression (`parseExpression`)

Until a semicolon is encountered, the parser repeatedly retrieves the next token. Several flags are kept to validate the token currently being retrieved. The table belows shows the conditions that would make a token invalid:

Token	Previous: Variable/ constant	Previous: Operator	Previous: “(“	Previous: “)”	Initial Expression	# of “(“
;		✕	✕		✕	✕ if 0
(	✕			✕		
)		✕	✕		✕	✕ if <= 0
Operator (+, -, *, /, %)		✕	✕		✕	
Valid variable name/ constant	✕			✕		
EOF, other characters	✕					

Table 3.2.1.3 Invalid Expression Conditions

Illustration of `parseExpression`

With reference to `FragFood`, the following table shows the steps taken by the parser in parsing the expression after the equals sign below:
Line 7: `money = (( salary - taxes)/ 10) + (allowance * 2) % 1;`

Token	Previous: Variable/ constant	Previous: Operator	Previous: “(“	Previous: “)”	Initial Expression	# of “(“
(	No	No	No	No	Yes	0
(	No	No	Yes	No	No	1
salary	No	No	Yes	No	No	2
-	Yes	No	No	No	No	2
taxes	No	Yes	No	No	No	2
)	Yes	No	No	No	No	2
/	Yes	No	No	No	No	1
10	No	Yes	No	No	No	1

)	Yes	No	No	No	No	1
+	No	No	No	Yes	No	0
(	No	Yes	No	No	No	0
allowance	No	No	Yes	No	No	1
*	Yes	No	No	No	No	1
2	No	Yes	No	No	No	1
)	Yes	No	No	No	No	1
%	No	No	No	Yes	No	0
1	No	Yes	No	No	No	0
;	Yes	No	No	No	No	0

The following table shows the steps taken by the parser in parsing the expression after the equals sign below, if the expression is invalid:
Line 7: `money = (( salary - taxes/) 10) + (allowance * 2) % 1;`

Token	Previous: Variable/ constant	Previous: Operator	Previous: “(“	Previous: “)”	Initial Expression	# of “(“
(	No	No	No	No	Yes	0
(	No	No	Yes	No	No	1
salary	No	No	Yes	No	No	2
-	Yes	No	No	No	No	2
taxes	No	Yes	No	No	No	2
/	Yes	No	No	No	No	2
)	No	Yes	No	No	No	2

Since the token “)” immediately follows an operator, it is syntactically invalid and an exception will be thrown before program execution terminates.

Parsing of If Statement (`parseIfStatement`)

Firstly, the parser validates that the statement begins with the keyword "`if`". Next, it calls `parseConditionalExpression` to parse and validate the conditional expression. Then, it validates that the conditional expression is followed by the keyword "`then`". Thereafter, it calls `parseStatementList` to parse and validate the statement list. Following that, it validates that the statement list is followed by the keyword "`else`". Finally, it calls `parseStatementList` to parse and validate the second statement list.

Parsing of While Statement (`parseWhileStatement`)

The parser validates that the statement begins with the keyword "`while`". Then, it calls `parseConditionalExpression` to parse and validate the conditional expression. Finally, it calls `parseStatementList` to parse and validate the statement list.

Parsing of Conditional Expression (`parseConditionalExpression`)
The parser validates that the conditional expression begins with an opening parenthesis. Then, it calls the recursive function `parseBracketedExpression`.

Parsing of Bracketed Expression (`parseBracketedExpression`)
Until a matching closing parenthesis is encountered, the parser repeatedly retrieves the next token. If an opening parenthesis is encountered, the parser calls the function to parse the following expression in brackets. Several flags are kept to validate the token currently being retrieved. The table belows shows the conditions that would make a token invalid:

<table>
<tr><th>Token</th><th>Previous: Variable</th><th>Previous: Operator</th><th>Previous: “(“</th><th>Previous: “)”</th></tr>
<tr><td>(</td><td>✕</td><td colspan="2">✕ if expected conditional expression and non-conditional expression was parsed
✕ if expected relational factor and relational expression or conditional expression was parsed</td><td>✕</td></tr>
<tr><td>)</td><td>✕ if # of relational operators > 1 or a conditional expression is still expected</td><td>✕</td><td>✕</td><td>✕ if # of relational operators > 1 or a conditional expression is still expected</td></tr>
<tr><td>Relational Operator (<=, <, >, >=, !=, ==)</td><td></td><td>✕</td><td>✕</td><td>✕ if previous bracketed expression was not an expression</td></tr>
<tr><td>Conditional Operator (&&, ||, !)</td><td colspan="4">✕ if previous: “)“ and previous bracketed expression was an expression
✕ if previous not “(“ and token is “!”
✕ if previous not “)“ and token is “&&” or “||”</td></tr>
<tr><td>Operator (+, -, *, /, %)</td><td></td><td>✕</td><td>✕</td><td>✕ if previous bracketed expression was not an expression</td></tr>
<tr><td>Valid variable name</td><td>✕</td><td></td><td></td><td>✕</td></tr>
<tr><td>EOF, other characters</td><td colspan="4">✕</td></tr>
</table>

Table 3.2.1.4 Invalid Conditional Expression Conditions

Illustration of `parseConditionalExpression` and `parseBracketedExpression`

With reference to `FragFood`, the following table shows the steps taken by the parser in parsing the conditional expression in parentheses below:
Line 5: `while ((availability==0)||(availability<=cravings)) {`

Token	Previous: Variable/constant	Previous: Operator	Previous: “(“	Previous: “)”	Expected	# of Relational Operators	Has Conditional Operator	Previous Bracketed Expression	parseBracketedExpression Recursion Depth
(	No	No	No	No	-	0	No	-	1
(	No	No	Yes	No	-	0	No	-	1
availability	No	No	Yes	No	-	0	No	-	2
==	Yes	No	No	No	-	0	No	-	2
0	No	Yes	No	No	Rel Factor	1	No	-	2
)	Yes	No	No	No	Rel Factor	1	No	-	2
||	No	No	No	Yes	-	0	No	Rel Expr	1
(	No	Yes	No	No	Cond Expr	1	Yes	-	1
availability	No	No	Yes	No	-	0	No	-	2
<=	Yes	No	No	No	-	0	No	-	2
cravings	No	Yes	No	No	-	1	No	-	2
)	Yes	No	No	No	-	1	No	-	2
)	No	No	No	Yes	Cond Expr	0	Yes	CondExpr	1

The following table shows the steps taken by the parser in parsing the conditional expression in parentheses below, if the expression is invalid:
Line 5: `while ((availability==0)||(availability<=cravings)) {`

Token	Previous: Variable/constant	Previous: Operator	Previous: “(“	Previous: “)”	Expected	# of Relational Operators	Has Conditional Operator	Previous Bracketed Expression	parseBracketedExpression Recursion Depth
(	No	No	No	No	-	0	No	-	1
(	No	No	Yes	No	-	0	No	-	1
availability	No	No	Yes	No	-	0	No	-	2

+	Yes	No	No	No	-	0	No	-	2
0	No	Yes	No	No	-	0	No	-	2
)	Yes	No	No	No	-	0	No	-	2
||	No	No	No	**Yes**	-	0	No	**Expr**	1

Since the token “||” immediately follows a closing parenthesis and the previous parenthesis-bounded expression was an expression, it is syntactically invalid and an exception will be thrown before program execution terminates.

Extraction of `Calls` and `Calls*` Relationships

The following SIMPLE source program will be used to illustrate the extraction of `Calls` and `Calls*` relationships outlined in this section.

```
    procedure ABC {
1     call BCD;
    }

    procedure BCD {
2     call CDE;
    }

    procedure CDE {
3     call DEF;
    }

    procedure DEF {
4     call FGH;
    }

    procedure FGH {
5     a = b;
    }
```

Extraction of `Calls` Relationship (`extractCalls`)

After the program has been parsed, the parser populates the PKB `CallsTable` (storing Calls relationships between procedures) by iteratively processing the populated PKB `CallTable` (storing statement line numbers and the procedure names they called). The parser also checks if there have been any **calls to non-existing procedures** by performing a check in each iteration if the procedure being called exists in the PKB `ProcedureTable`.

Illustration of `extractCalls`

For the SIMPLE source program above, the `CallTable`, `ProcedureTable` and `ProcedureStatementListTable` would be populated during parsing with the following information:

PKB Call Table	
LineNumber	**Called ProcedureName**
1	“BCD”
2	“CDE”
3	“DEF”
4	“FGH”

PKB Procedure Table	
ProcedureIndex	**ProcedureName**
0	“ABC”
1	“BCD”
2	“CDE”
3	“DEF”
4	“FGH”

PKB ProcedureStatementList Table	
ProcedureIndex	**StatementList**
0	1
1	2
2	3
3	4
4	5

Table 3.2.1.5 PKB `CallTable`, `ProcedureTable` and `ProcedureStatementListTable`

Then, by retrieving the index of the procedure the line number belongs to, and retrieving the index of the procedure being called, the following `Calls` relationship would be extracted: <0, 1>, <1, 2>, <2, 3>, <3, 4>. The `CallsTable` would be populated as below:

PKB Calls Table	
Calling ProcedureIndex	**Called ProcedureIndex**
0	1
1	2
2	3
3	4

Table 3.2.1.6 PKB `CallsTable`

For example, for Line 1, we check the `CallsTable` and get the procedureName being called: "BCD". Then, we check the `ProcedureStatementListTable` and get the procedureIndex Line 1 belongs to: 0. Lastly, we check the `ProcedureTable` and get the procedureIndex of "BCD": 1. Hence, the `Calls` relationship extracted would be <0, 1>, which would be stored in the `CallsTable`.

Extraction of Calls* Relationship (extractCallsStar)
After the PKB CallsTable has been populated, the parser populates the PKB CallsStarTable by calling the recursive function recursiveExtractCallsStar to extract Calls* relationship using Depth-First Search (DFS) for each procedure in the program.

Recursive Extraction of Calls* Relationship (recursiveExtractCallsStar)
Calls* relationship is recursively extracted with the help of a boolean array visited that keeps track of the procedures that have had their Calls* relationship extracted, as well as a stack recursionVisited that keeps track of the procedures being recursively processed.

The algorithm for extraction of Calls* relationship for a procedure is as follows:

1. Let the starting procedure be p.
2. If p is already marked as being recursively processed, there is a cyclic call and extraction is terminated. Else, mark p as being recursively processed.
3. Get the set of procedures called by p.
4. For each procedure q in the set, if q has not had their Calls* relationship extracted, call recursiveExtractCallsStar on q.
5. Insert Calls*(p, q).
6. Get the set of procedures transitively called by q, previously extracted by recursiveExtractCallsStar in Step 4.
7. For each procedure r in the set, check if r is the same procedure as p. If yes, there is a cyclic call and extraction is terminated. Else, insert Calls*(p, r).
8. Finally, mark p as having had its Calls* relationship extracted, and unmark p as being recursively processed.

The parser previously checked for **recursive calls** in parseCallStatement by checking that the program being called is not the same as the current program being parsed. The parser also checked for **calls to non-existing procedures** in extractCalls by checking if procedures being called exist in the PKB ProcedureTable.

This recursive function checks for **cyclic calls** in Step 2 and Step 7. Through the above-mentioned functions, the parser ensures that the source program is syntactically valid in terms of its procedure calls.

Population of PKB

The following table outlines the PKB tables that are populated by the parser, the relevant values inserted and the method at which it is populated.

Table	Value	Depth
`ProcedureTable`	Procedure Name	`parseProcedure`
`ParentTable`	Child Line #, Parent Line #	`parseStatementList`
`StatementTable`	Line Number, Statement Type	`parseStatementList`
`FollowTable`	Follower Line #, Line #	`parseStatementList`
`StatementListTable`	Starting Line #, Ending Line #	`parseStatementList`
`ProcedureStatementListTable`	Starting Line #, Ending Line #	`parseStatementList`
`ReadTable`	Line #, Variable Name	`parseReadStatement`
`PrintTable`	Line #, Variable Name	`parsePrintStatement`
`CallTable`	Line #, Procedure Name	`parseCallStatement`
`VariableTable`	Variable Name	`parseReadStatement`, `parsePrintStatement`, `parseAssignStatement`, `parseExpression`, `parseBracketedExpression`
`AssignTable`	Variable Name (LHS), Shunted* Expression (RHS)	`parseAssignStatement` (LHS), `parseExpression` (RHS)
`ConstantTable`	Line #, Constants	`parseExpression`, `parseBracketedExpression`
`IfTable`	Line #, Token	`parseConditionalStatement`, `parseBracketedExpression`
`WhileTable`	Line #, Token	`parseConditionalStatement`, `parseBracketedExpression`
`CallsTable`	Calling Procedure #, Called Procedure #	`extractCalls`
`CallsStarTable`	Calling Procedure #, Called Procedure #	`extrractCallsStar`

Table 3.2.1.7 Parser-PKB Population

Illustration of Parsing, Validation and Population of PKB

With reference to FragFood, the following table shows the steps taken by the parser in parsing and validating the program as well as populating the PKB:

Token	Validation	Line Number	Parent Stack	Statement List	Population of PKB
procedure	Is keyword "procedure"				
FragFood	Is valid procedure name				Insert "FragFood" into ProcedureTable
{	Is opening brace "{"				
HunGrY	Is valid variable name	1		1 Procedure: 1	Insert (1, "assign") into StatementTable Insert "HunGry" into VariableTable Insert "HunGry" into AssignTable (LHS)
=	Is equals sign "="	1		1 Procedure: 1	
1	Is valid expression	1		1 Procedure: 1	
;	Is semicolon ";"	1		1 Procedure: 1	Insert { 1, { 1 } } into ConstantTable Insert "1" into AssignTable (RHS)
f00d	Is valid variable name	2		1, 2 Procedure: 1, 2	Insert (2, "assign") into StatementTable Insert "f00d" into VariableTable Insert "f00d" into AssignTable (LHS)
=	Is equals sign "="	2		1, 2 Procedure: 1, 2	
0	Is valid expression	2		1, 2 Procedure: 1, 2	
;	Is semicolon ";"	2		1, 2 Procedure: 1, 2	Insert (2, { 0 }) into ConstantTable Insert "0" into AssignTable (RHS)
read	Is keyword "read"	3		1, 2, 3 Procedure: 1, 2, 3	Insert (3, "read") into StatementTable
appetitet0d4y	Is valid variable name	3		1, 2, 3 Procedure: 1, 2, 3	Insert (3, "appetitet0d4y") into ReadTable
;	Is semicolon ";"	3		1, 2, 3 Procedure: 1, 2, 3	
print	Is keyword "print"	4		1, 2, 3, 4	Insert (4, "print") into StatementTable

				Procedure: 1, 2, 3, 4	
restaurants	Is valid variable name	4		1, 2, 3, 4 Procedure: 1, 2, 3, 4	Insert (4, “restaurants”) into `PrintTable`
;	Is semicolon “;”	4		1, 2, 3, 4 Procedure: 1, 2, 3, 4	
while	Is keyword “while”	5			
(	Is valid conditional expression	5	5	1, 2, 3, 4, 5 Procedure: 1, 2, 3, 4, 5	Insert (5, “(“) into `WhileTable`
(					Insert (5, “(“) into `WhileTable`
availability					Insert “availability” into `VariableTable` Insert (5, “availability“) into `WhileTable`
==					Insert (5, “==“) into `WhileTable`
0					Insert (5, { 0 }) into `ConstantTable` Insert (5, “0“) into `WhileTable`
)					Insert (5, “)”) into `WhileTable`
||					Insert (5, “||“) into `WhileTable`
(					Insert (5, “(“) into `WhileTable`
availability					Insert “availability” into `VariableTable` Insert (5, “availability“) into `WhileTable`
<=					Insert (5, “<=“) into `WhileTable`
cravings					Insert “cravings” into `VariableTable` Insert (5, “cravings“) into `WhileTable`
)					Insert (5, “)”) into `WhileTable`
)					Insert (5, “)”) into `WhileTable`
{	Is opening brace “{“	5	5	1, 2, 3, 4, 5 Procedure: 1, 2, 3, 4, 5	
print	Is keyword “print”	6	5	6 Procedure: 1, 2, 3, 4, 5	Insert (6, “print”) into `StatementTable` Insert (6, 5) into `ParentTable`
availability	Is valid variable name	6	5	6 Procedure: 1, 2, 3, 4, 5	Insert “availability” into `VariableTable` Insert (6, “restaurants”) into `PrintTable`
;	Is semicolon “;”	6	5	6	

				Procedure: 1, 2, 3, 4, 5	
money	Is valid variable name	7	5	6, 7 Procedure: 1, 2, 3, 4, 5	Insert (7, “assign”) into `StatementTable` Insert (7, 5) into `ParentTable` Insert “money” into `VariableTable` Insert “money” into `AssignTable` (LHS)
=	Is equals sign “=”	7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
(	Is valid expression	7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
(		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
salary		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	Insert “salary” into `SalaryTable`
-		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
taxes		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	Insert “taxes” into `VariableTable`
)		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
/		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
10		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
)		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
+		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
(		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
allowance		7	5	6, 7 Procedure: 1, 2, 3, 4,	Insert “allowance” into `VariableTable`

				5	
*		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
2		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
)		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
%		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
1		7	5	6, 7 Procedure: 1, 2, 3, 4, 5	
;	Is semicolon “;”	7	5	6, 7 Procedure: 1, 2, 3, 4, 5	Insert (7, { 10, 2, 1 }) into `ConstantTable` Insert “salary taxes - 10 / allowance 2 * % 1 +” into `AssignTable` (RHS)
}	Is closing brace “}”		5	6, 7 Procedure: 1, 2, 3, 4, 5	Insert (6, 7) into `FollowsTable` Insert (6, 7) into `StatementListTable`
}	Is closing brace “}”			Procedure: 1, 2, 3, 4, 5	Insert (1, 2), (2, 3), (3, 4), (4, 5) into `FollowsTable` Insert (1, 5) into `ProcedureStatementListTable`

Design Considerations

Implementation of Parser

We considered two options for the implementation of the Parser in parsing and validating the source program as well as populating the PKB with the appropriate design entities.

They are outlined in the table below:

Criteria for Evaluation	Option 1: One-Pass	**Option 2: Three-Pass** (Parsing first, validation second, population of PKB third)
Separation of Concerns	Does not follow: Parsing, validation and population of PKB are done all-at-once; testing can only commence after everything is implemented	Follows: The parser is split into 3 different sub-components to work on the 3 passes; testing can commence as long as one sub-component has been implemented, and it is easier to debug
Redundancy	Less: Similar operations are less likely to be repeated as everything is done in one pass	More: Some similar operations must be performed for all 3 sub-components, which means there will be code redundancy and overhead
Implementation Difficulty	Harder to implement: Parsing, validation and population of PKB may occur within the same method for multiple methods throughout the program	Easier to implement
Time Complexity	Faster: Only 1 pass is required	Slower: 3 passes are required
Development Deadlines	Faster: There is no need to cleanly segment the 3 sub-components for development, and there is no need to test the sub-components individually	Slower

Table 3.2.1.8 Parser Design Options and Criteria for Evaluation

Evaluation of Implementation of Parser

We decided to choose **Option 1: One-Pass**.

This is because parsing and validation can be performed at the same time in a recursive, descent parser, leading to less redundancy and overhead compared to Option 2. Although Option 1 is harder to implement than Option 2, the difference in implementation difficulty is not large as simple design extractions can be easily extracted while parsing. Additionally, for Option 2, more code and test cases need to be written and more planning needs to be done, which makes Option 2 tedious to implement. Given the tight deadline, the one-pass method is preferable for us to deliver a working system on time even though we are giving up the greater Separation of Concerns offered by Option 2.

Implementation of Calls/Calls* Extraction

We also had the option of implementing the extraction of Calls/Calls* relationships in the Parser or in the Design Extractor.

Criteria for Evaluation	Option 1: Extraction of Calls/Calls* in Parser	Option 2: Extraction of Calls/Calls* in Design Extractor
Separation of Concerns	Does not follow: Parser is only responsible for validation and population of PKB with design entities, and the extraction of relationships between program design entities is separate from its concerns	Follows: Extraction of Calls/Calls* relationships is done in the Design Extractor, which aligns with its primary responsibility of extraction of relationships between program design entities
Redundancy	Less: Checking for calls to non-existing procedures and cyclic calls can be done at the same time as extraction of Calls/Calls*	More: Similar operations must be performed for checking for calls to non-existing procedures and cyclic calls and extraction of Calls/Calls*, which means there will be code redundancy and overhead

Table 3.2.1.9 Calls/Calls Design Options and Criteria for Evaluation*

Evaluation of Implementation of Calls/Calls* Extraction

We decided to choose **Option 1: Extraction of Calls/Calls* in Parser**.

Although Option 2 offers greater Separation of Concerns, Option 1 allows the system to adhere to the principle that parsing should fail if the SIMPLE source program violates syntax grammar while also minimising redundancy. For Option 2, the extraction of Calls/Calls* needs to be re-implemented in the Design Extractor despite the fact that the operations performed in checking for cyclic calls are very similar to the operations performed in extracting Calls* relationships. Additionally, unit testing and integration testing must also be done for the extraction of Calls/Calls* in the Design Extractor. Choosing Option 2 would lead to more code redundancy and increased tediousness of implementation, making Option 1 the preferred choice for us even though we are giving up the greater Separation of Concerns offered by Option 2.

3.2.2 Front-End Design Extractor Design

The **Design Extractor** is responsible for:

1. Extracting `Follows*` and `Parent*` relationships by accessing and evaluating `Follow` and `Parent` relationships determined by the Parser from the PKB. Generated `Follows*` and `Parent*` relationships are then populated into the **PKB**.
2. Extracting `Uses` and `Modifies` relationships from assign, read, print, if and while statements determined by the **Parser** from the **PKB**. Generated `Uses` and `Modifies` (across multiple procedures) relationships are then populated into the **PKB.** Note that nested child statements have also been considered and handled here.
3. Extracting `Next` relationships by accessing and evaluating all relationships extracted previously. Generated `Next` relationships are then populated into the **PKB.**
4. On demand extraction of `Next* and Affects/Affects*` relationships for **Query Evaluator.**
5. Extracting `NextBip/NextBip*` and `AffectsBip/AffectsBip*` relationships by accessing and evaluating all relationships extracted previously. Generated `NextBip/NextBip*` and `AffectsBip/AffectsBip*` relationships are then populated into the **PKB.** Please refer to Section 4 for details on `NextBip/NextBip*` and `AffectsBip/AffectsBip*` extraction.

After the **Parser** has extracted and stored basic design relationships into the PKB during parsing, the **Design Extractor** communicates with the **PKB** to evaluate and extract more complex design relationships. Examples of relationships extracted can be found above.

Generating Follows* Relationships

The Follows* relationship is generated by making use of the Follows relationship stored in the PKB. For the purpose of explaining this algorithm, we define two terms "follower" and "leader". For instance, in a Follows*(s1, s2) relationship, where s1 and s2 are both statements, s1 is a leader and s2 is a follower.

To extract Follows* relationship, the algorithm follows this flow:

1. Extract all followers from PKB
2. For each of the followers, perform the next step.
3. If the follower has a leader, then keep finding out if the leader has another leader, adding them into a set "leaders" each time this is TRUE. This repeats until a leader has no more leader, then, the relationship Follows* is set for the follower to all of the elements in the set "leader".

Simply put, this algorithm iterates through all statements that a particular statement can have some form of indirect Follows relationship with to extract all Follows* relationship.

Illustration of Extraction of Follow* Relationships

Assume PKB indicates that Follows(1, 2), Follows(2, 3) and Follows(3, 4) is TRUE. Firstly, we get all followers from PKB - { 2, 3, 4 }. For each follower, we keep finding their leader to add to "leaders". For instance, **during the iteration of 4**, we find that the "leaders" set will contain { 3, 2, 1 }. This means that Follows*(1, 4), Follows*(2, 4), Follows*(3, 4) are TRUE and will be populated into the PKB.

Generating Parent* Relationships

This algorithm is similar to that of the Follows* extraction. Instead of "follower" and "leader", we define "parent" and "child". In a Parent(s1, s2) relationship, where s1 and s2 are both statements, s1 is a parent and s2 is a child.

This algorithm iterates through all statements that a particular statement can form an indirect Parent relationship with to extract the Parent* relationship, with a similar flow to that of the extraction of Follows*.

Illustration of Extraction of Parent* Relationships

Assume PKB indicates that Parent(1,2), Parent(2,3), Parent(3,4) is TRUE. Firstly, we get all children from the PKB - { 2, 3, 4 }. For each child, we keep finding their parents to add to a set "parents". For instance, **during the iteration of 4**, we find that the "parents" set contain {3,2,1). This leads to the population of Follows*(1,4), Follows*(2,4) and Follows*(3,4) in the PKB.

Generating Uses and Modifies Relationships

The Design Extractor makes use of design entities determined by the Parser to extract the `Uses` and `Modifies` Relationships. This algorithm starts off by extracting all procedures from the PKB. Then, the method `extractUsesAndModifiesForProcedure` is called on each procedure to extract all the `Uses` and `Modifies` Relationships that the procedure contains.

The majority of the extraction lies in the method `extractUsesAndModifiesForProcedure.`
All statements in the procedure are iterated through with the following order depending on its design entity:

1. Assign/Read/Print/Calls
2. If/While

Depending on the design entity, the `Uses` and `Modifies` relationships are first extracted in relation to the statement, then extracted in relation to the procedure that it belongs in, before being stored in the PKB.

The following SIMPLE source program will be used to explain how the above is achieved for each design entity.

```
    procedure One {
1      if (a != b) then {
2         c = d;
3         call Two;
       } else {
4           read g;
5           print h;
       }
    }
    procedure Two {
6      d = e;
7       f = g;
    }
```

Assign Statement

Since the Parser has already conveniently stored the LHS and RHS variables of an assign statement in the PKB as a Pair of <LHS, RHS>, we access these variables to extract the `Uses` and `Modifies` relationship.

We populate the PKB with a `Modifies` relationship between the assign statement and the variable on the LHS, and a `Uses` relationship between the assign statement and the variable on the RHS. Finally, the `Uses` and `Modifies` relationship between the procedure and all used and modified variables are also populated into the PKB.

Illustration of Extraction of `Uses/Modifies` Relationships for Assign Statement

For the SIMPLE source program above, the `AssignTable` would be populated during parsing with the following information:

PKB Assign Table	
LineNumber	**<VariableLHS, VariableRHS>**
2	<“c”, “d”>

Table 3.2.2.1 PKB `AssignTable`

We extract `Modifies`(2, “c”) and `Uses`(3, “d”), `Modifies`(“One”, “c”), `Uses`(“One”, “d”) and populate this into the PKB.

Print and Read Statement

Variables that are associated with a print or read statement can be extracted from the PKB, allowing the `Uses` and `Modifies` relationship to be extracted from there.

We populate the PKB with a `Modifies` relationship between the read statement and the variable associated to it. A Uses relationship is also populated between the print statement and the variable associated to it.

Illustration of Extraction of `Uses/Modifies` Relationships for Print and Read Statement

For the SIMPLE source program above, the `ReadTable` and `PrintTable` would be populated during parsing with the following information:

PKB Read Table	
LineNumber	**Read Variable**
4	“g”

PKB Print Table	
LineNumber	**Print Variable**
5	“h”

Table 3.2.2.2 PKB `ReadTable` and `PrintTable`

We extract `Modifies`(4, “g”) and `Uses`(5, “h”), `Modifies`(“One”, “g”), `Uses`(“One”, “h”) and populate this into the PKB.

Call Statement

The algorithm first checks if there exists a `Uses` or `Modifies` relationship stored in the PKB for this statement. This implies that the `Uses` and `Modifies` relationship have already been extracted for the procedure associated with the call statement.

If there isn't any, then `extractUsesAndModifiesForProcedure` will be called with the procedure name associated with the call statement as an argument. This recursive method ensures that the procedure in the call statement will always have all the statements updated with its `Uses` and `Modifies` information.

Illustration of Extraction of `Uses/Modifies` Relationships for Call Statement

For example, in line 3 of the sample program, `extractUsesAndModifiesForProcedure(“Two”)` is called to extract all Uses and Modifies relationships in procedure “Two”. After which, we the PKB stores the following information:

PKB ProcedureUses Table	
ProcedureIndex	**Uses Variables**
1	“e”, “g”

PKB ProcedureModifies Table	
ProcedureIndex	**Modifies Variables**
1	“d”, “f”

Table 3.2.2.3 PKB `ProcedureUsesTable` and `ProcedureModifiesTable`

With this information, we can determine that the line 3 has the following relationships:
`Uses`(3, “e”), `Uses`(3, “g”), `Modifies`(3, “d”), `Modifies`(3, “f”).
On top of that `Uses`(“One”, “e”), `Uses`(“One”, g), `Modifies`(“One”,”d”), `Modifies`(“One”, “f”) is also extracted and populated into the PKB.

It is crucial to note that this extraction is possible because all `Uses` and `Modifies` relationships extracted in a procedure are always populated in the PKB in the `procedureUses` and `procedureModifies` Table.

If and While Statements

The conditions of the if and while statements are first evaluated to determine what variables are used by the conditions.

After that, statements nested within the If or While statement are obtained using by **finding all statements with a `Child*` relationship** to the If or While statement. We then find the variables used or modified by these statements and populate a used or modified relationship respectively into the PKB with relation to the If or While statement. This is possible because the `Uses` and `Modifies` relationship were already populated previously while analysing the assign, read and print statements.

Illustration of Extraction of `Uses/Modifies` Relationships for If and While Statements

For example, the PKB stores the following relevant information after parsing the sample program:

PKB If Table	
LineNumber	**Uses Variables**
1	“a” , “b”

PKB While Table	
LineNumber	**Modifies Variables**
4	“e”, “f”

PKB Uses Table	
LineNumber	**Uses Variables**
2	“d”
3	“e”, “g”,
5	“h”

PKB Modifies Table	
LineNumber	**Modifies Variables**
2	“c”
3	“d”, “f”
4	“g”

Table 3.2.2.4 PKB `IfTable`, `WhileTable`, `UsesTable` and `ModifiesTable`

Firstly, we determine that `Uses`(1, “a”) and `Uses`(1,”b”) since these are the variables used in the condition statement of the If statement in line 1.
Next, we can determine that `Uses`(1,”d”), `Uses`(1,”e”), `Uses`(1,”g”), `Uses`(1,”h”), `Modifies`(1,”c”), `Modifies`(1,”d”), `Modifies`(1,”f”), `Modifies`(1,”g”) to be TRUE.
These information are populated into PKB, along with the `Uses` and `Modifies` relationship between the procedure and all the variables used and modified within it, as mentioned above. This will be left out for the sake of brevity.

Generating `Next` Relationships

This section explains how `Next` relationships can be extracted from various tables in the PKB to populate the `NextTable` in the PKB. By mainly using `Follows` and `Parent` relationships, we are able to extract the `Next` relationship by analysing each statement iteratively. We also have two helper functions: `findFirstChild` and `findFirstElseChild`.

- `findFirstChild` retrieves information from the PKB `StatementListTable` to determine the first child of an If/While statement
- `findFirstElseChild` retrieves information from the PKB `StatementListTable` and `ParentTable` to determine the first child of an If statement

The following SIMPLE source program will be used to illustrate the extraction of `Next` relationship outlined in this section.

```
    procedure ABC {
1     if (a == b) then {
2       a = c;
3       while (a != d) {
4         b = d;
5         d = f;
        }
      } else {
6       e = h;
      }
7     e = d;
    }
```

Firstly, all statements are extracted from the PKB `StatementTable` to be iterated. These statements can be broken down and analysed in three key groups:

Group 1: If statements
For an if statement, the first statement in its then clause and the first statement in its else clause are two possible `Next` statements.

Group 2: While statements
For a while statement, the first statement in its statement list is one of the two possible `Next` statements.

Another possible `Next` statement, if any, is determined based on the following algorithm:

1. Let the while statement be s1.
2. If there is a statement s2 such that `Follows`(s1, s2), then s2 is the other possible `Next` statement and the algorithm terminates.
3. Otherwise, if s1 has no direct parent statement, then there is no other possible `Next` statement and the algorithm terminates.
4. Otherwise, if the direct parent of s1 is a while statement, then that while statement is the other possible `Next` statement, and the algorithm terminates.
5. Otherwise, replace s1 with its direct parent and repeat the algorithm from step 2 until the algorithm terminates.

Group 3: Assign, Call, Read, Print statements

For an assign, call, read or print statement, the one possible `Next` statement, if any, is determined based on the same algorithm as that of the while statement.

Illustration of `extractNext`

With reference to the SIMPLE source program, the PKB stores the following relevant information after parsing:

PKB Follows Table	
Follower LineNumber	**LineNumber**
7	1
3	2
5	4

PKB Parent Table	
Child LineNumber	**Parent LineNumber**
2, 3, 6	1
4, 5	3

PKB Statement Table	
LineNumber	**StatementType**
1	“if”
2	“assign”
3	“while”
4	“assign”
5	“assign”
6	“assign”
7	“assign”

PKB StatementList Table	
First LineNumber	**StatementList**
2	2, 3
4	4, 5
6	6

Table 3.2.2.5 PKB `FollowsTable`, `ParentTable`, `StatementTable` and `StatementListTable`

The following table shows the steps taken by the Design Extractor in extracting `Next` relationships:

Line Number	StatementType	Extraction
1	“if”	- `findFirstChild` returns 2 - `findFirstElseChild` returns 6

		- `Next`(1, 2) and `Next`(1, 6) are added into the PKB `NextTable`
2	“assign”	- `Follows`(3, 2) is in the PKB `FollowsTable` - `Next`(2, 3) is added into the PKB `NextTable`
3	“while”	- `findFirstChild` returns 4 - `Next`(3, 4) is added into the PKB `NextTable` - Line 3 has no follower - Line 3 has a direct parent statement, Line 1 - `Follows`(7, 1) is in the PKB `FollowsTable` - `Next`(3, 7) is added into the PKB `NextTable`
4	“assign”	- `Follows`(5, 4) is in the PKB `FollowsTable` - `Next`(4, 5) is added into the PKB `NextTable`
5	“assign”	- Line 5 has no follower - Line 5 has a direct parent statement, Line 3, and Line 5 is a while statement - `Next`(5, 3) is added into the PKB `NextTable`
6	“assign”	- Line 6 has no follower - Line 6 has a direct parent statement, Line 1 - `Follows`(7, 1) is in the PKB `FollowsTable` - `Next`(6, 7) is added into the PKB `NextTable`
7	“assign”	- Line 7 has no follower - Line 7 has no direct parent statement - No `Next` relationship is added into the PKB `NextTable`

Generating `Next*` Relationships On Demand

This section explains 3 key methods that are used on demand by the Query Evaluator to assist them in the evaluation of queries relating to `Next*` relationships.
The Design Extractor provides a cache NextStar Cache that stores results that have been evaluated in the same query, allowing the Query Evaluator to clear the cache after each query evaluation.

To determine `Next*` relationships, 3 different methods `getNextStar`, `getPreviousStar` and `isNextStar` are provided by the Design Extractor.

Finding out all statements s2 `such that` `Next*(s1, s2)` is TRUE: `getNextStar(s1)`

After checking that the cache does not contain this result, this algorithm will rely on Breadth First Search (BFS) to iterate through the Next Table in the PKB to continuously find statements with a Next relationship to each other.

Illustration of getNextStar()

Given that the `NextTable` currently contains the following information:

PKB Next Table	
LineNumber	**NextLine Number**
1	2
2	3,4
3	1,5
4	1,5
5	

Table 3.2.2.6 PKB `NextTable`

Assuming that the Query Evaluator needs to know what statements s2 satisfies `Next*`(1,s2), the algorithm first stores 1 into a queue, and marks 1 as visited. It calls `getNext` on every element e in the queue and adds the relationship `Next*`(1, e) into PKB, followed by calling `getNext` on the element and adding the results to the queue.

This is a rough flow of extracting this relationship for the current example:

1. After calling `getNext`(1) which returns {2}, the queue now contains {2}, the NextStar Cache stores the relationship Next*(1,2).
2. `getNext`(2) is now called which returns {3,4}, the queue now contains {3,4}, and the NextStar Cache stores the relationship `Next*`(1,3) and `Next*`(1,4).
3. `getNext`(3) is now called which returns {1,5}, the queue now contains {4,5}, noting that 1 is not added in the queue as it was already visited in Step 1. The NextStar Cache now stores the relationship and `Next*`(1,5).
4. `getNext`(4) is now called which returns {1,5}, the queue now contains {5}, noting that 5 is not added in the queue as it was already visited in Step 3.

5. `getNext`(5) is now called, returning no results. The queue is now empty and the extraction for `getNextStar`(1) is completed.

In the end, the NextStar Cache indicates that `Next*`(1,2), `Next*`(1,3), `Next*`(1,4) and `Next*`(1,5) is TRUE. The results returned for the method call `getNextStar`(1) will be {2,3,4,5}.

An important thing to note that we could have stored intermediate results obtained such as `Next*`(2,3) into the cache However, we deliberately chose not to as the next query accessing the NextStar Cache might mistakenly identify `getNextStar`(2) as being evaluated fully previously, thus returning an incomplete result.

Finding out all statements s1 `such that` Next*(s1, s2) is TRUE: `getPreviousStar`(s2)

This algorithm is similar to `getNextStar`, utilising BFS to determine `Next*` relationships. The key difference here is that instead of `getNext`(s1), which returns all statements s2 `such that` `Next`(s1,s2) is TRUE, `getPrevious`(s2) is used instead, returning all statements s1 `such that` `Next`(s1,s2) is TRUE.

Illustration of getPreviousStar()

Given that the Previous Table currently contains the following information:

PKB Previous Table	
LineNumber	**PreviousLine Number**
1	3,4
2	1
3	2
4	2
5	3,4

Table 3.2.2.7 PKB `PreviousTable`

Assuming that the Query Evaluator needs to know what statements s1 satisfies `Next*`(s1,1), the algorithm first stores 1 into a queue, and marks 1 as visited. It calls `getPrevious` on every element e in the queue and adds the relationship `Next*`(e, 1) into PKB, followed by calling `getPrevious` on the element and adding the results to the queue.

This is a rough flow of extracting this relationship for the current example:

1. After calling `getPrevious(1)` which returns {3,4}, the queue now contains {3,4}, the NextStar Cache stores the relationship `Next*`(3,1) and `Next*`(4,1)
2. `getPrevious(3)` is now called which returns {2}, the queue now contains {4,2}, and the NextStar Cache stores the relationship `Next*`(2,1).
3. `getPrevious(4)` is now called which returns {2}, the queue now contains {2}, noting that 2 is not added in the queue as it was already visited in Step 2.
4. `getPrevious(2)` is now called which returns {1}. Since 1 was already visited at the start, the queue is now empty and the extraction for getPreviousStar(1) is completed.

In the end, the NextStar Cache indicates that `Next*`(2,1), `Next*`(3,1) and `Next*`(4,1) is TRUE. The results returned for the method call `getPreviousStar(2)` will be {2,3,4}.

Finding out if `Next*(s1, s2)` is TRUE for statements s1 and s2: `isNextStar(s1,s2)`

This algorithm can be split into different cases to optimally determine this result by utilising the NextStar cache and the Next Table.

Case 1: `Next`(s1,s2) is TRUE
We first check if `Next`(s1,s2) is TRUE. If this is the case, then `Next*`(s1,s2) must be TRUE and we return TRUE immediately.

Case 2: NextStar Cache contains `getNextStar(s1)`
We then check if `getNextStar`(s1) has already been previously computed and stored in the NextStar cache. If so, we simply return whether the result of `getNextStar(s1)` contains s2.

Case 3: NextStar Cache contains `getPreviousStar(s2)`
Similarly, we can check if `getPreviousStar(s2)` has already been previously computed and stored in the NextStar cache. If so, we simply return whether the result of `getPreviousStar(s2)`contains s1.

Case 4: All of the above is FALSE
Otherwise, we will have to call `getNextStar(s1)` to first determine all the statements `x such that Next*(s1,x) is TRUE`, before checking if `s2` exists in `x`. This helps in storing the cache for future queries relating to `Next*`(s1).

By utilizing the NextStar Cache depending on different cases, we can efficiently determine whether `Next*`(s1,s2) is TRUE.

Generating `Affects` Relationships On Demand

This section explains the 3 main methods used for the on demand extraction of Affects relationships. The results here are used by the Query Evaluator directly with no precomputation.

The Design Extractor provides a cache Affects Cache that stores the results that have been evaluated previously in the same query, and also allows the Query Evaluator to clear the cache after each query evaluation.

The following SIMPLE source program will be used to explain certain parts of the extraction of `Affects` relationship outlined in this section.

```
     procedure One {
1       a = b;
2       c = d;
3       if (d == b) then {
4          a = f;
        } else {
5          g = h;
        }
6       h = a + c;
     }
```

Finding out all statements s2 `such that` `Affects`(s1, s2) is TRUE: `getAffects(s1)`

Firstly, the cache is checked to see if the Affects relationship for s1 have been computed in the same query before. If so, the results are simply returned from the cache. Aside from this, we also check if s1 is an assignment statement. If it isn't, then it returns an empty result since only assignment statements can have Affects relationships.

For every assignment statement, we run a Breadth-First Search(BFS) in a similar fashion as that used in the population of Next* relationships. For the sake of brevity, we omit the detailed explanation for this and explain the differences instead.

Before the start of each BFS, we first "blacklist" statements from being visited during the BFS. All statements(apart from if and while statements) that modify the same variable modified by s1 are blacklisted. Note that this does not prevent a visit during BFS, but rather prevents a valid flow passing through it during BFS.

Next, we perform BFS starting from s1, visiting statements that have a Next relationship with the current statement. Whenever a statement is being visited, we check if this statement is an assignment statement that uses the variable modified by s1. If so, then we populate an Affects relationship between this statement and s1.

Illustration of getAffects(1)

Using the code example provided above, we discover how the algorithm runs when running getAffects(1). Firstly, statements 1 and 4 are blacklisted. Note that although statement 3 modifies variable a, we do not blacklist it since it does not directly modifies a.
Next, a BFS is initiated from s1, and the following are some possible and noteworthy flows:

1 -> 2 -> 3 -> 4

The next statements to be visited from 3 can be 4 or 5. However, since 4 is blacklisted, we visit it to check if there is an Affects relationship, but do not consider any execution flows next of 4 to visit next in our BFS.

1 -> 2 -> 3 -> 5 -> 6

Here, we see that statement 6 uses variable a which is modified by statement 1. Hence, we add an Affects relationship between 1 and 6 into the cache.

Finally, the result {6} is returned for an on demand call by the Query Evaluator for getAffects(1).
If the Query Evaluator calls getAffects(1) again in the same query, we can immediately return the same result without any computation.

Finding out all statements s1 `such that` `Affects`(s1, s2) is TRUE: `getAffectedBy(s2)`

The initial checks are the same as that for `getAffects` - checking the cache contains the same results computed before and whether s2 is an assignment statement.

This algorithm is similar to that used in `getAffects(s1)`. Instead of applying BFS in the forward direction, we apply in the backward direction by using `getPrevious` instead of `getNext`. Also, we apply BFS for every variable that is used in s2, tracing possible paths to find s1 that modifies it directly.

Before the start of each BFS, we check if there are any statements that modify this variable directly(omitting if and while statements). If there are such statements, we blacklist them to prevent the backward BFS from visiting them.

Next, the backward BFS is performed starting from s1, visiting statements that have a Previous(opposite of Next) with the current statement. Whenever an assignment statement that uses the variable is visited, an AffectedBy relationship is added between this statement and s2.

Illustration of getAffectedBy(6)

Using the same code example provided above, we explain further how this algorithm works when executing getAffectedBy(6). Since statement 6 uses 2 variables a and c, there will be 2 BFS done.
For BFS of variable a, we first blacklist statements 1 and 4. Next a BFS is initiated from statement 6, with the following being some noteworthy flows:

6 -> 4

Since statement 4 modifies variable a which is used in statement 6, an AffectedBy relationship is populated between 6 and 4 in the cache. However, note that the possible statements previous of 4 in an execution flow will not be added to the BFS to be visited next.

6 -> 5 -> 3 -> 2 -> 1

Since statement 1 modifies variable a used in statement 6, an AffectedBy relationship is populated between 6 and 1 in the cache.
For BFS of variable c, we first blacklist statements 2. Next a BFS is initiated from statement 6, with the only noteworthy flow:

6 -> 5 -> 3 -> 2

Since statement 2 modifies variable c used in statement 6, an AffectedBy relationship is populated between 6 and 2 in the cache.

Finally the result {1,2,4} is returned for an on demand call by the Query Evaluator for getAffectedBy(6).
If the Query Evaluator calls getAffectedBy(6) again in the same query, we can immediately return the same result without any computation.

Finding out if `Affects`(s1, s2) is TRUE for statements s1 and s2: `isAffects(s1,s2)`

This algorithm can be split into different cases to optimally determine this result by utilising the Affects cache and the Affects table

Case 1: s1 or s2 are not assign statements

We first check if s1 and s2 are assign statements. If either of them are not, then we simply return false since only assign statements can have Affects relationship.

Case 2: s2 does not use the variable modified by s1

We check if the variable modified by s1 is within the set of variables used by s2. If this is not the case, then we return false.

Case 3: Affects Cache contains `getAffects(s1)`

Similarly, we can check if `getAffects(s1)` has already been previously computed and stored in the Affects cache. If so, we simply return whether the result of `getAffects(s1)` contains s2.

Case 4: Affects Cache contains `getAffectedBy(s2)`

Similarly, we can check if `etAffectedBy(s2)` has already been previously computed and stored in the Affects cache. If so, we simply return whether the result of `getAffectedBy(s2)` contains s1.

Case 4: All of the above is FALSE

Otherwise, we will have to call `getAffects(s1)` to first determine all the statements `x such that Affects(s1,x) is TRUE`, before checking if `s2` exists in `x`. This helps in storing the cache for future queries relating to get`Affects`(s1).

By utilizing the Affects Cache depending on different cases, we can efficiently determine whether `Affects`(s1,s2) is TRUE.

Generating Affects* Relationships On Demand

This section explains 3 key methods that are used on demand by the Query Evaluator to assist them in the evaluation of queries relating to Affects* relationships.
The Design Extractor provides a cache AffectsStar Cache that stores results that have been evaluated in the same query, allowing the Query Evaluator to clear the cache after each query evaluation.

To determine Affects* relationships, 3 different methods getAffectsStar, getAffectedByStar and isAffectsStar are provided by the Design Extractor.

The algorithm is similar to that of the Next* extraction, with a few minor differences. Hence, examples have been omitted for brevity.

Finding out all statements s2 such that Affects*(s1, s2) is TRUE: getAffectsStar(s1)
After checking that s1 is an assign statement and that the cache does not contain this result, this algorithm will rely on Breadth First Search (BFS) to iterate through the Affects Table in the PKB to continuously find statements with an Affects relationship to each other.

Finding out all statements s1 such that Affects*(s1, s2) is TRUE: getAffectedByStar(s2)
This algorithm is similar to getAffectsStar, utilising BFS to determine Affects* relationships after checking that s2 is an assign statement and that the cache does not contain this result. The key difference here is that instead of getAffects(s1), which returns all statements s2 such that Affects(s1, s2) is TRUE, getAffectedBy(s2) is used instead, returning all statements s1 such that Affects(s1, s2) is TRUE.

Finding out if Affects*(s1, s2) is TRUE for statements s1 and s2: isAffectsStar(s1,s2)
This algorithm can be split into different cases to optimally determine this result by utilising the AffectsStar cache and the Affects Table.

Case 1: s1 or s2 are not assign statements
We first check if s1 and s2 are assign statements. If either of them are not, then we simply return false since only assign statements can have Affects* relationship.

Case 2: AffectsStar Cache contains getAffectsStar(s1)
Similarly, we can check if getAffectsStar(s1) has already been previously computed and stored in the AffectsStar cache. If so, we simply return whether the result of getAffectsStar(s1) contains s2.

Case 3: AffectsStar Cache contains getAffectedByStar(s2)
Similarly, we can check if getAffectedByStar(s2) has already been previously computed and stored in the AffectsStar cache. If so, we simply return whether the result of getAffectedByStar(s2) contains s1.

Case 4: All of the above is FALSE
Otherwise, we will have to call getAffectsStar(s1) to first determine all the statements x such that AffectsStar(s1,x) is TRUE, before checking if s2 exists in x. This helps in storing the cache for future queries relating to getAffectsStar(s1).

By utilizing the AffectsStar Cache depending on different cases, we can efficiently determine whether AffectsStar(s1,s2) is TRUE

Design Considerations

Implementation of Next*/Affects* Cache

We considered two options for the Next* and Affects caches to handle and store the temporary data for the on demand Next* relationship extraction for every query.

They are outlined in the table below:

Criteria for Evaluation	Option 1: NextStar and Affects Cache is implemented inside PKB	Option 2: NextStar and Affects Cache is implemented inside Design Extractor.
Single Responsibility Principle	Does not follow. The role of the PKB is to store persistent design information data for access by various components. If the cache were to be stored in the PKB, PKB would have to be modified to store non-persistent information just that is only referenced by the Design Extractor. This means the PKB will now have an additional unnecessary responsibility to assist on demand extraction. Furthermore, the role of the Design Extractor is to extract advanced relationships, not to be an intermediary to PQL accessing the cache for previously computed results.	Follows. Option 2 follows the Single Responsibility principle. The Design Extractor handles the temporary storage of this information in order to assist it in extracting advanced design relationships. This allows the responsibility of assisting the extraction of on demand relationships to fall on Design Extractor only.
Redundancy	Less	More. Existing solutions inside the PKB cannot be reused to implement this

Table 3.2.2.8 Next/Affects* Cache Design Options and Criteria for Evaluation*

Evaluation of Implementation of Next*/Affects* Cache

We decided to choose **Option 2: NextStar Cache is implemented inside Design Extractor.**
This is because the Single Responsibility Principle here takes more priority over redundancy.
The PKB should only deal with persistent data storage for other components in the system to access, and it should not have to deal with the additional responsibility of storing temporary data just to assist one of Design Extractor's functions. Furthermore, option 2 ensures that the Design Extractor does not have an added responsibility of being an intermediary for PQL to access the cache in the PKB if option 1 was chosen.

Although Option 1 would allow us to have more code reusability and less redundancy, we assessed that the code redundancy caused will be minimal if we went with Option 2 since it is unlikely that the cache implemented in the Design Extractor will be reused for similar solutions. This is especially since the cache should be cleared at the end of every query and used specifically only for each specific design relationship.

3.2.3 PKB Design

The PKB was designed to store all relationships and necessary information to facilitate faster parsing and evaluation of the input SIMPLE source program and queries. Hashtables are used for every table due to their extremely fast insertion and lookup times.

Our PKB is solely used to store information and does not do any processing of data and thus, the data needs to be processed beforehand before storing it in.

In general, the keys and values of each Relationship table are stored in a way that:

1. If the table name is a noun, a key's *tableType* is its associated value.
    a. In the Parent table, Key:3 Value:2 means that 3's parent is 2.
    b. In the Next table, Key:2 Value:3 means that 2's next line is 3.
2. If the table name is a verb, a key *tableTypes* its associated value.
    a. In the Follows table, Key:3 Value:2 means that 3 follows 2.
    b. In the Calls table, Key:2 Value:3 means that procedure 2 calls procedure 3.

The PKB contains the following tables with the following data type mappings:

- Integer:String
    - *Variable, Procedure, Statement, Read, Print, Call*
- Integer:Integer
    - *Follows*, *Parent*
- Integer:Set_of_Integers
    - *FollowsStar*, *ParentStar*, *Constant, StatementList, ProcedureStatementList, Next, Calls, CallsStar*
- Integer:Set_of_Strings
    - *Uses*, *Modifies, ProcedureUses, ProcedureModifies, While, If*
- Integer:Pair_of_<String, String>
    - *assign*

Integer:String tables

The Integer:String tables are designed as such because they either use an Index:Variable, Index:Procedure or LineNumber:Variable key value pair.

Variable, Procedure Table

The *Variable* table and the *Procedure* table are used to store variable names and procedure names that are found in the SIMPLE program. It has a mapping of an index to the variable name and procedure name for the respective tables. Technically, the *Variable* and *Procedure* tables also have private reverse tables used only to quickly return the index of an already inserted variable/procedure.

The tables automatically assign an index to the names while storing and do not store any duplicate values. The indexes start from 0.

It checks for any repeated values and if found, it returns the index that was already assigned to it for the *Variable* table, while ignoring the insertion for *Procedure* table.

When faced with the following SIMPLE program :

```
    procedure Moondrop {
1        blessing = 2;
2        s = 8;
    }
    procedure Sundrop {
3        curse = 2;
4        w = 8;
    }
```

The *Variable* and *Procedure* tables would contain the following mappings as shown by tables 1 and 2 below:

Index	Variable name
0	“blessing”
1	“s”
2	“curse”
3	“w”

Table 3.2.3.1 Variable table

Variable name	Index
“blessing”	0
“s”	1
“curse”	2
“w”	3

Table 3.2.3.2 Private Reverse Variable table

Index	Procedure name
0	“Moondrop”
1	“Sundrop”

Table 3.2.3.3 Procedure table

Procedure name	Index
“Moondrop”	0
“Sundrop”	1

Table 3.2.3.4 Private Reverse Procedure table

During the course of iteration 1, we had a request from the front-end development team that they needed another table to store a mapping of the procedure name to the start and end lines of each procedure in a procedureName:Pair_of_startLine_and_endLine mapping. However, the PKB team then realised that we did not have to create another class for such a table and could just use another instance of the existing statementList table to store all statements that were within each procedure.

Statement Table

The *Statement* table is used to store the mapping between the line and the statement type at the corresponding line. The front-end is required to input the line and the appropriate statement type into the table.

Given the following SIMPLE program:

```
    procedure main {
1       flag = 0;
2       call computeCentroid;
    }
```

The Statement table would contain the following mapping:

Line Number	StatementType
1	“assign”
2	“call”

Table 3.2.3.5 Statement table

Read, Print Table

The *Read* and *Print* tables are used to store the variables that have been read and printed respectively. It has a mapping of the line numbers at which the respective statements are being called to the variables that are being read and printed respectively.

Given the following SIMPLE program:

```
    procedure Legacy {
1      read x;
2      print y;
    }
```

The *Read* and *Print* Tables would contain the following mappings as shown in tables 4 and 5 below:

Line Number	VariableName
1	“x”

Table 3.2.3.6 Read table

Line Number	VariableName
2	“y”

Table 3.2.3.7 Print table

Call Table

The *Call* table is used to store procedure names that are being called in the SIMPLE program. It has a mapping of the line number at which the procedure is being called and the name of the procedure called. Given the following SIMPLE program:

```
    procedure Google {
1      call Lens;
2      call Glass;
    }
```

The Call table would contain the following mappings:

Line Number	ProcedureName
1	“Lens”
2	“Glass”

Table 3.2.3.8 Call table

Integer:Integer and Integer:Set_of_Integers tables

The Integer:Integer tables are designed as such because they use a LineNumber:LineNumber key value pair.
The Integer:Set_of_Integers tables are designed as such because they use a LineNumber:Set_of_LineNumbers or ProcedureIndex:Set_of_ProcedureIndexes key value pair.

Follows, FollowsStar

`Follows` is an object that contains both the `Follows` and *FollowedBy* tables.
Similarly, the `Follows`*Star* is an object that contains both the `Follows`*Star* and *FollowedByStar* tables.

The mappings for each of the tables are as follows:

1) `Follows` table - LineNumber:LeaderLine
2) *FollowedBy* table - LineNumber:FollowerLine
3) `Follows`*Star* table - LineNumber:Set_of_<LeaderLines>
4) *FollowedByStar* table - LineNumber:Set_of_<FollowerLines>

Given the following SIMPLE program:

```
     procedure Moondrop {
1        blessing = 2;
2        s = blessing + 5;
3        if(blessing == 2) then {
4            blessing = blessing + 1;
5            s = 2;
         } else {
6            print s;
         }
7        solis = s + blessing;
     }
```

The 4 tables listed above would contain the following mappings after Front End is done with parsing and design extraction.

Line Number	LeaderLine
2	1
3	2
5	4
7	3

Table 3.2.3.9 `Follows` *Table*

Line Number	FollowerLine
1	2
2	3
4	5
3	7

Table 3.2.3.10 FollowedBy Table

Line Number	LeaderLines
2	<1>
3	<1, 2>
5	<4>
7	<1, 2, 3>

Table 3.2.3.11 FollowsStar Table

Line Number	FollowerLines
1	<2, 3, 7>
2	<3, 7>
3	<7>
4	<5>

Table 3.2.3.12 FollowedByStar Table

Parent, ParentStar

Parent is an object that contains the Parent and *Child* tables.
Similarly, the Parent*Star* is also an object that contains both the Parent*Star* and *ChildStar* tables.

The mappings for each of the tables are as follows:

1) Parent table - LineNumber:ParentLine
2) *Child* table - LineNumber:Set_of_<ChildLines>
3) Parent*Star* table - LineNumber:Set_of_<ParentLines>
4) *ChildStar* table - LineNumber:Set_of_<ChildLines>

Given the following SIMPLE program:

```
     procedure Moondrop {
1        blessing = 2;
2        s = blessing + 5;
3        if(blessing == 2) then {
4            blessing = blessing + 1;
5            while(blessing == 2) {
6                blessing = blessing - 1;
             }
         } else {
7            print s;
         }
8        solis = s + blessing;
     }
```

The 4 tables listed above would contain the following mappings:

Line Number	ParentLine
4	3
5	3
6	5

7	3

Table 3.2.3.13 Parent Table

Line Number	ChildLines
3	<4, 5, 7>
5	<6>

Table 3.2.3.14 Child Table

Line Number	ParentLines
4	<3>
5	<3>
6	<3, 5>
7	<3>

Table 3.2.3.15 ParentStar Table

Line Number	ChildLines
3	<4, 5, 6, 7>
5	<6>

Table 3.2.3.16 ChildStar Table

Constant Table

The *Constant* table stores a mapping between the line number and the value(s) that constitute towards the constant itself. For the value, we used an unordered Set_of_integers to show the numbers that constitute toward the final value of the constant.

Given the following SIMPLE program:

```
    procedure Moondrop {
1        blessing = 2;
2        s = 3 + 5;
    }
```

The Constant table would contain the following mappings:

Line Number	ConstantValues
1	<2>
2	<3, 5>

Table 3.2.3.17 Constant Table

StatementList, ProcedureStatementList Table

The *StatementList* table stores a mapping between the line number at which the statement lists start from and the statement lines that are associated to it. This can also be used to store the index of a procedure and the statement lines within a procedure as a ProcedureStatementList table.

Given the following SIMPLE program:

```
     procedure Moondrop {
1        blessing = 2;
2        s = 3+5;
3        var1 = 4;
     }
```

The StatementList table would contain the following mappings:

Line Number	StatementLines
1	<1, 2, 3>

Table 3.2.3.18 StatementList Table

Procedure Index	StatementLines
0	<1, 2, 3>

Table 3.2.3.19 ProcedureStatementList Table

Next

Next is an object that contains both the *Next* and *Previous* tables.

The mappings for each of the tables are as follows:

1) *Next* table - LineNumber:Set_of_<NextLines>
2) *Previous* table - LineNumber:Set_of_<PreviousLines>

Given the following SIMPLE program:

```
     procedure Moondrop {
1        blessing = 2;
2        if(blessing == 2) then {
3            solis = s + blessing;
         } else {
4            print s;
         }
5        s = blessing + 5;
     }
```

The 4 tables listed above would contain the following mappings after Front End is done with parsing and design extraction.

Line Number	Next Line
1	<2>
2	<3, 4>
3	<5>
4	<5>

Table 3.2.3.20 `Next` *Table*

Line Number	Previous Line
2	<1>
3	<2>
4	<2>
5	<3, 4>

Table 3.2.3.21 Previous Table

Calls, CallsStar

`Calls` is an object that contains both the `Calls` and *CalledBy* tables.
Similarly, the *CallsStar* is an object that contains both the *CallsStar* and *CalledByStar* tables.

The mappings for each of the tables are as Calls:

1) `Calls` table - IndexNumber:CalledIndex
2) *CalledBy* table - IndexNumber:CalledByIndex
3) *CallsStar* table - IndexNumber:Set_of_<CalledIndexes>
4) *CalledByStar* table - IndexNumber:Set_of_<CalledByIndexes>

Given the following SIMPLE program:

```
     procedure Moondrop {
1        call Sundrop;
2        call Planetdrop;
     }
     procedure Sundrop {
3        call Stardrop;
     }
     procedure Stardrop {
4        x = x + 1;
     }
     procedure Planetdrop {
```

```
5        call Sundrop;
     }
```

In the Procedure table:

0 : Moondrop
1 : Sundrop
2 : Stardrop

The 4 tables listed above would contain the following mappings after Front End is done with parsing and design extraction.

Index Number	CalledIndex
0	<1, 3>
1	<2>
3	<1>

Table 3.2.3.22 `Calls` *Table*

CalledByIndex	Index Number
1	<0, 3>
2	<1>
3	<0>

Table 3.2.3.23 CalledBy Table

Index Number	CalledIndex
0	<1, 2, 3>
1	<2>
3	<1, 2>

Table 3.2.3.24 CallsStar Table

CalledByIndex	Index Number
1	<0, 3>
2	<0, 1, 3>
3	<0>

Table 3.2.3.25 CalledByStar Table

Integer:Set_of_Strings tables

The Integer:Set_of_Strings tables are designed as such because they use a LineNumber:Set_of_Variables or ProcedureIndex:Set_of_Variables key value pair.

Uses, Modifies, ProcedureUses, ProcedureModifies

`Uses` is an object that contains both the `Uses` table and the *UsedBy* table. Similarly, the `Modifies` is also an object that contains both the `Modifies` table and the *ModifiedBy* table. This can also be used to store the index of a procedure and the variables used/modified by a Procedure as *Procedure*`Uses`*/ProcedureModifies* table.

The mappings for each of the tables are as follows:

1) `Uses` Table - LineNumber:Set_of_<UsedNames>
2) *UsedBy* Table - Name:Set_of_<UsedByLines>
3) `Modifies` Table - LineNumber:Set_of_<ModifiedNames>
4) *ModifiedBy* Table - Name:Set_of_<ModifiedByLines>
5) *Procedure*`Uses`*Table* - ProcedureIndex:Set_of_<UsedNames>
6) *ProcedureUsedBy* Table - Name:Set_of_<UsedByIndexes>
7) *ProcedureModifies* Table - ProcedureIndex:Set_of_<ModifiedNames>
8) *ProcedureModifiedBy* Table - Name:Set_of_<ModifiedByIndexes>

GIven the following SIMPLE program:

```
     procedure Moondrop {
1        blessing = 2;
2        s = blessing + 5;
3        if(blessing == 2) then {
4            blessing = blessing + 1;
5            s = 2;
         } else {
6            print s;
         }
7        solis = s + blessing;
     }
```

The 4 tables listed above would contain the following mappings:

Line Number	UsedNames
2	<"blessing">
3	<"blessing", "s">
4	<"blessing">
6	<"s">
7	<"s", "blessing">

Table 3.2.3.26 `Uses` *Table*

ProcedureIndex	UsedNames
0	<”blessing”, “s”>

Table 3.2.3.27 Procedure`Uses` *Table*

Variable Names	LineNumber
“blessing”	<2, 3, 4, 7>
“s”	<3, 6, 7>

Table 3.2.3.28 UsedBy Table

Variable Names	ProcedureIndex
“blessing”	0
“s”	0

Table 3.2.3.29 ProcedureUsedBy Table

Line Number	ModifiedNames
1	<”blessing”>
2	<”s”>
3	<”blessing”, “s”>
4	<”blessing”>
5	<“s”>
7	<”solis”>

Table 3.2.3.30 `Modifies` *Table*

ProcedureIndex	UsedNames
0	<”blessing”, “s”, “solis”>

Table 3.2.3.31 ProcedureModifies Table

Line Number	ModifiedByNames
<“blessing”>	<1, 3, 4>
<“s”>	<2, 3, 5>
<“solis”>	<7>

Table 3.2.3.32 ModifiedBy Table

Line Number	ProcedureIndex
<“blessing”>	0
<“s”>	0
<“solis”>	0

Table 3.2.3.33 ProcedureModifiedBy Table

While, If Table

The `While` and `If` tables are used to store the line numbers at which the `While` and `If` statements are located within a SIMPLE program. It contains a mapping of the line number to the conditions within the while and if statements within the program.

Given the following SIMPLE program:

```
    procedure Keyboard {
1       x = 3;
2       y = 2;
3       while(x > 0) {
4           if(y == x)then{
5               call Switches;
            } else {
6               call Spaces;
            }
7           x = x - 1;
         }
    }
```

The *While* and *If* tables would contain the following mappings as shown:

Line Number	Condition
3	<“x”>

Table 3.2.3.34 While table

Line Number	Condition
4	<“y”, “x”>

Table 3.2.3.35 If table

The tables were designed this way as we would be able to easily find out at which line the `While` and `If` statements were being called and the variables used in the conditions

.

Integer:Pair_of_Integer_and_String tables

Assign Table

The `assign` table stores a mapping between the line number at which the assignment statement can be found and a pair of what is on the left and right of the assignment statement. Before storing it, the front-end would need to perform the Shunting Yard Algorithm on the right hand side of the statement.

Given the following SIMPLE program:

```
     procedure Moondrop {
1        s = 3 + 5;
     }
```

The `assign` table would contain the following mappings:

Line Number	AssignStatement
1	<”s”, “3 5 +”>

Table 3.2.3.36. assign Table

The table is designed as such to easily identify where the assignment statements are located and to identify which variables have been involved in the assignment statement. The shunted version of the expression is being stored to also help the PQL segment in solving `pattern` clauses.

The team also did consider having a table called “`assign`*NonShunted*” which stores the assign statements as is, in a line number:assignStatement mapping. For instance, the example in Table 22 would look like this for the `assign`*NonShunted* table :

Line Number	AssignStatement
1	”s = 3 + 5”

Table 3.2.3.37 assignNonShunted Table Example

However, we realised that this was redundant and any information needed from the assign statements can be easily retrieved from the `assign` table instead. Hence, we decided to scrap this table and retain only the `assign` table.

Design Considerations

These are the options considered for the query preprocessing of which are outlined in the table below:

Criteria for Evaluation	Option 1: Abstract Syntax Tree	Option 2: Hash Tables (Ordered Map)	Option 3: Hash Tables (Unordered Map)
Separations of Concerns	Storing relationships into a tree structure, where a node *verbs* its parent node or is a noun of its child node. Example: A `Follows` node `Follows` its parent node, a `Parent` node is a `Parent` of it's child node. However, tables may still be required for Variables or Procedures.	Storing relationships into hash tables, where a node *verbs* its parent node or is a noun of its child node. Example: A `Follows` key `Follows` its value, a `Parent` key is a `Parent` of its value. Consistent usage of tables for all PKB structures.	Storing relationships into hash tables, where a node *verbs* its parent node or is a noun of its child node. Example: A `Follows` key `Follows` its value, a `Parent` key is a `Parent` of its value. Consistent usage of tables for all PKB structures.
Implementation Difficulty	Simple at first, as once a node is designed, it can be replicated and connected piece by piece. However for more complex relationships, we may get a network instead of a tree, which can be harder to debug.	Simple throughout using built in C++ Hash Tables.	Simple throughout using built in C++ Hash Tables.
Performance (Speed)	Insert Average:O(log n) Worst: O(log n) Search Average: O(log n) Worst: O(log n)	Insert Average:O(log n) Worst: O(log n) Search Average: O(log n) Worst: O(log n)	Insert Average:O(1) Worst: O(n) Search Average: O(1) Worst: O(n)

Table 3.2.3.38 Comparison of implementation options for PKB

Evaluation

We decided to go with **Option 3**, by using Hash Tables implemented by Unordered Maps. We chose Hash Tables over Trees because of consistency, ease of implementation and speed. An AST would also get very complex with more complex source programs.

Both Tree and Ordered Map are O(log n) for Insert and Search as the Ordered Map has an underlying tree structure, but in practice building and using a tree is slower due to various underlying mechanisms such as memory access.

While the Unordered Map is slower at its worst compared to the Ordered Map, it is much faster on average. As such, most if not all of our usage should fall into the average runtime.

Criteria for Evaluation	Option 1: “One way” Tables	Option 2: “Two way” Tables
Separations of Concerns	Storing relationships “One way”. For example, storing Modifies relationships into a Modifies table.	Storing relationships “Two way”. For example, storing Modifies relationships into a Modifies table, and in reverse as ModifiedBy relationships into a ModifiedBy table.
Implementation Difficulty	Simple, no additional work required.	Roughly twice the work required compared to “One way” tables.
Performance (Speed)	Insert **Once** per relationship Search **Once** if in the direction of “One way”. For example, finding what a given line modifies. **TableSize** if opposite of the direction of “One way”. For example, finding what a given line is modified by.	Insert **Twice** per relationship Search **Once** if in the direction of “One way”. For example, finding what a given line modifies. **Once** if opposite of the direction of “One way”. For example, finding what a given line is modified by.

Table 3.2.3.39 Comparison of additional implementation options for PKB

As relationships are two-way, we implemented **Option 2** “Two way” tables, which “reverse tables” that store reverse relationships to reduce computation required by the QueryEvaluator. While the tradeoff is that roughly twice as much time was needed for insertion, a loss of speed during Source Parsing for an increase of speed during Query Evaluation was deemed as satisfactory.

Examples of reverse tables are *FollowedBy, FollowedByStar, Child, ChildStar, UsedBy, ModifiedBy*. A full listing of these tables can be found in the Abstract API section.

3.2.4 Query Preprocessor

The **Query Preprocessor** is responsible for parsing and validating queries and adding the relevant details of the query that the query evaluator later needs into a `Query` object. The **Query Preprocessor** parses the entire statement, checks for syntactic errors and semantic errors and throws an exception respectively with an error message when encountered an error.

When the query string is parsed into the program, it will be tokenized, with the symbol '`;`' as the delimiter and split into multiple segments. The ones before '`;`' are treated as declarations while the last segment is treated as the main query to be tokenized. A detailed example can be found under the **Query Preprocessor Example** section below.

For each declaration, the **Query Preprocessor** will obtain the synonym and its design entity, and further divide the segments by the symbol '`,`', which will then provide a list of several synonyms. It will then check if it is a valid design entity. If it is valid, it will then insert the synonym and its design entity type and populate the synonym table. The Recursive Descent Parser is used to tokenize the main query. The parser will specify a token depending on the part of the main body that is being parsed. After identifying `such that`, `pattern` or `with` clauses, the **Query Preprocessor** will then call the respective parsing functions.

The **Query Preprocessor** will proceed to parse and validate the arguments and check for semantic errors once it is done processing the query to check for syntax errors.

Query Validation

Validation of the query is based on whether it is syntactically or semantically correct according to the CSG for Program Query Language (PQL).

Syntactic Correctness

For the declarations, it is checked against a list of all possible design entities. If it is a valid design entity type, it will be inserted into the synonyms table which stores the declared synonym and the design entity type.

Design Entities

```
[stmt, read, print, call, while, if, assign, variable, constant, procedure, prog_line]
```

For the selected synonyms with attributes, the attributes are checked against a list of possible attributes

Attributes

```
[procName, varName, value, stmt#]
```

The query preprocessor will check if the clause is a valid clause type before adding into the list of clauses.

Clauses

```
[such that, pattern, with]
```

For `such that` clause, it is checked against the list of all possible relationship types.

Relationships
`[Follows, Follows*, Parent, Parent*, Uses, Modifies, Calls, Calls*, Next, Next*, Affects, Affects*]`

Pattern Expression
The right hand argument of the assign-type pattern clause is validated as an expression by using the expression validation from the front-end parser.

Semantic Correctness

For the different types of clauses, the query preprocessor uses different methods to check for semantic correctness.

Semantic Correctness for `such that` Clause

To validate the left and right references of the `such that` clause, switch cases are used and the tables below are the possible inputs for the left and right references.

Semantic validation of each reference is done by checking whether the reference type or the given synonym type for the particular clause is valid.

Table 3.2.4.1 shows the valid and invalid relationships between the design abstraction types and the design entities type for Left / Right hand side of the arguments for the `such that` clause. Also in *Table 3.2.4.2,* it also checks the Left / Right hand side of the arguments if it is an INTEGER, “_” OR “IDENT”.

	stmt	read	print	call	while	if	assign	variable	constant	procedure	prog_line
Follows	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✕/✕	✕/✕	✕/✕	✓/✓
Follows*	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✕/✕	✕/✕	✕/✕	✓/✓
Parent	✓/✓	✕/✓	✕/✓	✕/✓	✓/✓	✓/✓	✕/✓	✕/✕	✕/✕	✕/✕	✓/✓
Parent*	✓/✓	✕/✓	✕/✓	✕/✓	✓/✓	✓/✓	✕/✓	✕/✕	✕/✕	✕/✕	✓/✓
Uses	✓/✕	✕/✕	✓/✕	✓/✕	✓/✕	✓/✕	✓/✕	✕/✓	✕/✕	✓/✕	✓/✕
Modifies	✓/✕	✓/✕	✓/✕	✓/✕	✓/✕	✓/✕	✓/✕	✕/✓	✕/✕	✓/✕	✓/✕
Calls	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✓/✓	✕/✕
Calls*	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✓/✓	✕/✕
Next	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✕/✕	✕/✕	✕/✕	✓/✓
Next*	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✕/✕	✕/✕	✕/✕	✓/✓
Affects	✓/✓	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✓/✓	✕/✕	✕/✕	✕/✕	✓/✓

Affects*	✓/✓	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✓/✓	✕/✕	✕/✕	✕/✕	✓/✓

Table 3.2.4.1 Left / Right hand side Design Entities relationship validation

	INTEGER	“_”	“IDENT”
Follows	✓/✓	✓/✓	✕/✕
Follows*	✓/✓	✓/✓	✕/✕
Parent	✓/✓	✓/✓	✕/✕
Parent*	✓/✓	✓/✓	✕/✕
Uses	✓/✕	✕/✓	✓/✓
Modifies	✓/✕	✕/✓	✓/✓
Calls	✕/✕	✓/✓	✓/✓
Calls*	✕/✕	✓/✓	✓/✓
Next	✓/✓	✓/✓	✕/✕
Next*	✓/✓	✓/✓	✕/✕
Affects	✓/✓	✓/✓	✕/✕
Affects*	✓/✓	✓/✓	✕/✕

Table 3.2.4.2 Left / Right hand side arguments relationship validation

Example

```
variable v; procedure p; Select p such that Modifies(_, "x")
```

The query preprocessor will check the design entity type first, and with the switch cases, check the LHS and RHS argument. In this case, “_” is not syntactically correct and an syntax error exception will be thrown, and the query will be flagged as an invalid query.

Semantic Correctness for With Clause

Each attribute reference is checked against a list of allowed design entity type and attribute combinations. This will check if the synonym is allowed to have the attribute type.

Allowed attributes

```
[procedure.procName, call.procName, variable.varName, read.varName, print.varName,
constant.value, stmt.stmt#, read.stmt#, print.stmt#, call.stmt#, while.stmt#,
if.stmt#, assign.stmt#]
```

Each reference would be either a name or a number, and both the left hand and right hand value must be of the same type. If the left hand side is a name, then the right hand side must also be a name.

Example

```
with procedure.procName = “val”
```

Semantic Correctness for Pattern Clause

The pattern synonym only applies for assign, while or if design entity type, and the first argument can either be a synonym, “_” or “IDENT”. For the pattern clause, it will check if the left hand side of the argument is either a synonym, “_” or ““ IDENT ““ according to the lexical tokens, and the right hand side of the argument according to the grammar rules.

Once the query is successfully parsed, the result will be added to a Query object, which the query object consists of a synonym table, a list of clauses, a list of selected synonyms and a boolean of whether the query is valid or invalid, and if the query is syntactically valid and semantically valid. The table below shows what is contained inside the Query Object.

Query Object
Synonyms Table
Clauses
Selected Synonyms
isValid
isSelectBoolean
isValidSyntax
isValidSemantic

Query Object

Query Object consists of:

1. A synonyms table that has the declared synonym as the key and design entity as the value `[stmt, read, print, call, while, if, assign, variable, constant, procedure, prog_line]`
2. A list of clauses

Clause		
ClauseType	`such that`, `pattern`, `with`	
Design Abstraction Type	`Follows, Follows*, Modifies, Parent, Parent*, Next, Next*, Calls, Calls*, Uses, Affects, Affects*`	
Reference[1]	ReferenceAttribute Type	`procName, varName, value, stmtNum, none`
	ReferenceType	`synonym, integer, wildcard, quotation, attribute`

	DesignEntityType	`stmt, read, print, call, while, if, assign, variable, constant, procedure, prog_line`
	Synonym string	
	Reference string	
PatternAbstractionType	`assign, if, while`	
Expression	ExpressionType	`partial match, no match, exact match`
	Expression string	

Structure of Clause

1: The clause has two References, for left and right hand side argument respectively.

3. A list of selected synonyms which is of structure **Element**

Element	
Element Type	`synonym, attribute`
ReferenceAttributeType	`Follows, Follows*, Modifies, Parent, Parent*, Next, Next*, Calls, Calls*, Uses, Affects, Affects*`
Synonym string	`procName, varName, value, stmtNum, none`

Structure of Element

4. Is valid flag
5. Is select BOOLEAN flag
6. Is valid syntax flag
7. Is valid semantic flag

Query Preprocessor Example

An example will be shown to demonstrate what happens or how a query is being processed by the query preprocessor.

```
assign a; variable v, v1; stmt s;
Select v such that Modifies(s, v1) pattern a(_, “10”) with v1.varName = v.varName
```

Step 1: Parse the declarations found in the query

Declarations:

Segment 1: `assign a;`
Segment 2: `variable v, v1;`
Segment 3: `stmt s;`

Declared synonyms: `<“a”, ASSIGN>, <“v”, VARIABLE>, <“v1”, VARIABLE>, <“s”, STMT>`

```
Select v such that Modifies(s, v1) pattern a(_, “10”) with v1.varName = v.varName
```

The query preprocessor will split the query by ‘`;`’ and those segments that end with ‘`;`’ are treated as declarations, then it will proceed to parse the declaration. Any other synonym names of the same design entity type will be separated by a comma. The last one will be treated as the main query to parse. It will first parse the first line, “`assign a`”, and it will check if “`assign`” is a design entity type by checking against the design entity type and assign the design entity type to the synonym accordingly. If it is valid, it will insert a <synonymString:DesignEntityType> mapping into the synonym table with a mapping of `<“a”, ASSIGN>`. If not, it will throw an syntax exception that it is not a valid design entity type.

Note: For `Select BOOLEAN` cases, it is not required to have declarations declared at the start.

Step 2: Parse Select and add selected synonyms into a list of selected synonyms

```
Select v such that Modifies(s, v1) pattern a(_, “10”) with v1.varName = v.varName
```

Selected synonym string: [v]

After parsing the declarations, it will move onto the Parse Select, which will first check if it starts with “Select”, then retrieve the selected synonym by the recursive descent parser. It will find “v” and move onto the Parse Selected Synonyms, where it will check if the synonym has attributes and assign the element type and reference type accordingly. This synonym is then inserted into a list of selected synonyms, where element type and reference type will be assigned to be synonyms.

Step 3: Parse “`such that`” Clause

`such that` Clause: `such that Modifies(s, v1)`
Arguments: [s, v1]

It will then retrieve the next token to check what clause it is and call the respective clause, in this case, it will call the `ParseSuchThatClause`. The query preprocessor will then retrieve the design abstraction string, the Left Hand Side (LHS) of the argument and the Right Hand Side (RHS) of the argument. From there, it will assign the respective design abstraction type to the design abstraction string. For the LHS and RHS arguments, it will go through `GetReference()` where it will take in the argument, check if it is a synonym, integer, quotation, attribute or wildcard and assign the reference type and design entity type accordingly. Once all of this information is retrieved, it will initialize a “`such that`” clause with all the details of the clause loaded inside and add into a list of clauses, and the query preprocessor will then validate the design abstraction reference relationships.

Step 4: Parse “`pattern`” Clause

Pattern Clause: `pattern a(_, “10”)`

Pattern-synonym: a
Reference string: _
Expression string: “10”

It will proceed to `ParseClauses`, parse and extract the clause type and call the respective clause, which is the `ParsePatternClause`. Similarly to the previous step, it will parse, and extract the pattern synonym, to check if it is of assign/if or while type. In this case, it proceeds to `parsePatternAssignClause`, and parse the LHS and RHS arguments, expression, and initialize the “`pattern`” clause and assign the types accordingly. From there, it will validate the pattern clause references to check if the relationship is valid or not. If it is valid, it will assign the respective expression type.

Step 5: Parse “`with`” Clause

With Clause: `with v1.varName = v.varName`
Arguments: [v1.varName, v.varName]

It will proceed to `ParseClauses`, parse and extract the clause type and call the respective clause, which is the `ParseWithClause`. It will parse LHS and RHS references and verify that the references are separated by a ‘=’. For the LHS and RHS references, it will go through `parseWithRef` where it will take in the reference and check if the reference has attributes and assign the attribute type, design entity type and reference type accordingly. It will then proceed to `validateWithReferences()` where it will take in the clause, validate the LHS and RHS references by checking if both are numbers or names.

At the end of the parsing, the tables below show how the query object and the clauses look like.

Query Object		
Synonyms Table		<”a”, ASSIGN> <”v”, VARIABLE> <”v1”, VARIABLE> <”s”, STMT>
Clauses		such that Modifies(s, v1) pattern a(_, “10”) with v1.varName = v.varName
Selected Synonyms	elementType elementAttribType synonymString	synonym none v
isValid		TRUE
isSelectBoolean		FALSE
isValidSyntax		TRUE
isValidSemantic		TRUE

Query Object after parsing and validation

such that Clause	
ClauseType	such that
DesignAbstractionType	Modifies
Reference ReferenceAttributeType ReferenceType DesignEntityType	s none synonym stmt
Reference ReferenceAttributeType ReferenceType DesignEntityType	v1 none synonym stmt

such that *Clause*

Pattern Clause	
ClauseType	pattern
DesignAbstractionType	none
Reference ReferenceAttributeType ReferenceType DesignEntityType	a none synonym assign
Reference ReferenceAttributeType ReferenceType DesignEntityType	_ none wildcard wildcard
PatternAbstractionType	assign
Expression ExpressionType	“10” exact match

pattern *Clause*

With Clause	
ClauseType	with
DesignAbstractionType	none
Reference	v1.varName

ReferenceAttributeType
ReferenceType
DesignEntityType
	`varName`
	`attribute`
	`variable`

Reference
ReferenceAttributeType
ReferenceType
DesignEntityType
	`v.varName`
	`varName`
	`attribute`
	`variable`

`with` *Clause*

Design Considerations

These are the options considered for the query preprocessing of which are outlined in the table below:

Criteria for Evaluation	Option 1: Only Tokenization and store in tree nodes	Option 2: Recursive descent parser and store in Query Object
Separations of Concerns	Parsing and validating at the same time, and having one tree structure to store all the information into the leaf nodes	Parse first, then validate the relationships or types using switch cases and populate relevant information into the `Query` Object
Implementation Difficulty	Simple at first, but as more features are added, it might get complex and a lot of changes have to be made. It is not flexible	Challenging at first but once the recursive descent parser is implemented, it is easier to add on new features for later iterations.
Performance	Slowest	Fastest

Comparison of implementation options for Query preprocessor

Evaluation

We decided to go with **Option 2**, by using Recursive Descent Parser to tokenize and validate the relationships by using tables as there are upcoming relationships that need to be handled, and we do not want to introduce too many changes in the code. We stored the information in a way that is easily retrievable, and that only relevant information is stored and would help in the further iterations for optimization, the validation is done by using switch cases.

3.2.5 Query Evaluator

The Query Evaluator is responsible for evaluating the queries, which have already been parsed and stored in a `Query` object by Query Preprocessor.

Example Query Object:

Query Object		
Synonyms Table		<”a”, ASSIGN> <”v”, VARIABLE> <”s”, STMT>
Clauses		such that Follows(s, 3) pattern a(v, _) with s.stmt# = 2
Selected Synonyms	elementType elementAttribType synonymString	synonym none s
isValid		TRUE
isSelectBoolean		FALSE
isValidSyntax		TRUE
isValidSemantic		TRUE

There are 4 main types of evaluation algorithms that are involved in the Evaluation of Queries. These 4 algorithms pertain to the select, `such that`, pattern, and with clause types respectively. Besides these four algorithms, there is an overarching evaluation flow which facilitates the evaluation of multiple clauses and allows the evaluation of separate clauses to be terminated early should any condition for early termination be met.

Data Structure for intermediate synonym results

Instead of using a Table to store the intermediate answers, our group decided to use a system of maps with reverse mapping. This system allows for each synonym's possible values, to be matched to the possible values of other synonyms. By using this system, we are able to remove the issue of performing cross-products during evaluation, as the relevant synonym pairs can be inserted or removed in O(1) time. However, the cross-product step is not removed entirely, but instead just delayed to the step where the QueryEvaluator evaluates the select clause.

Given the following SIMPLE program:

```
    procedure Keyboard {
1       a = b;
2       d = e;
3       f = g;
    }
```

As well as the query:
`stmt s1,s2; Select s1 such that Follows(s1,s2).`

In the evaluation of the clause Follows(s1,s2), for the synonym pair <s1,s2>, <1,2> and <2,3> are valid value pairs. They will be stored in a map as follows.

Map 1 Key (first synonym)	Map 2 Key (first synonym values)	Map 3 Key (second synonym)	Set (second synonym values)
s1	1	s2	2
s1	2	s2	3

Table 3.2.5.1 Mapping of Follows clause

The reverse is also stored:

Map 1 Key (first synonym)	Map 2 Key (first synonym values)	Map 3 Key (second synonym)	Set (second synonym values)
s2	2	s1	1
s2	3	s1	2

Table 3.2.5.2 Reverse mapping of Follows clause

Using this data structure, it is quick to access the possible values of s1, the possible values of s2, as well as the synonyms that are linked to each synonym. Eg. When s1 is evaluated with s2, given a value of s1(eg. 1), the synonyms that are linked with the value, (s2), as well as the values (2) can be easily accessed and removed.

Overarching evaluation flow

The program flow for the solveQuery method involves several steps that allow the Query Evaluator to quickly return a result if the query is invalid, either semantically or syntactically, as well as if partial evaluation is sufficiently conclusive to return a result before evaluation is fully completed. These behaviors are detailed below.

Handling of invalid Queries

Queries have a boolean in the Query object, isValidSyntax, which is set to be TRUE if the Query is valid syntactically. As such, once a Query object that is passed in has this boolean flag set to FALSE, an empty list of strings is returned as the answer. This is because the query is assumed to be unable to be parsed.

Queries have another boolean in the query object, isValidSemantic, which is set to be TRUE if the Query is valid semantically. Similarly, when a Query object that is passed in has this boolean flag set to FALSE, the overarching evaluation flow detects this and returns one of two outputs. If the Query is a select attribute/synonym/tuple query, an empty string is returned as the answer. If the query is a select boolean query, “FALSE” is returned.

Handling of Multiple Clauses

`such that` clauses, `pattern` clauses, and `with` clauses with “`and`” are parsed as separate clauses in the Query Preprocessor. All of these clauses are added to the clause list in the `query` object. The overarching evaluation flow processes these queries one by one, with each clause evaluation returning a TRUE or FALSE output to indicate if the clause has been successfully evaluated with at least one satisfactory case. This boolean return for each clause is used in the detection for possible early termination, explained in the next section.

Detection of cases of early termination

A boolean variable, cumulativeClauseReturn, is used to keep track of clause evaluation returns, where the TRUE/FALSE outputs of the evaluation for `such that`, `pattern`, and `with` clauses are evaluated together with the cumulativeClauseReturn using the following relation:

cumulativeClauseReturn = cumulativeClauseReturn AND clauseReturn (`such that`, pattern, or with)

This allows the overarching evaluation flow to detect when it is possible to end the evaluation early (Whenever a clause returns FALSE). If the evaluation is cut short, an empty string is returned as the answer if a tuple answer is required, and a “FALSE” is returned, if the required answer is a boolean instead.

Evaluation algorithm for `such that` clause

Example Such That Clause: `such that Follows(s, 3)`

`such that` Clause	
ClauseType	`such that`
DesignAbstractionType	`Follows`

Reference
ReferenceAttributeType
ReferenceType
DesignEntityType
	`s`
	`none`
	`synonym`
	`stmt`

Reference
ReferenceAttributeType
ReferenceType
DesignEntityType
	`3`
	`none`
	`integer`
	`stmt`

The first step to evaluation of a `such that` clause, is to determine the reference type of the 2 references and decide the evaluation flow based upon these 2 reference types. There are four key referenceTypes that will influence the way the clause is evaluated: integer, quotation, wildcard, and synonym. With regards to how they impact the evaluation flow, integers and quotations influence the evaluation similarly. Synonyms will also impact the evaluation flow differently, based upon whether they were previously evaluated.

Based upon the clause in the table, `such that Follows(s, 3)`, "Follows" is the DesignAbstractionType, s is reference 1, and is of type synonym, and 3 is reference 2, and is of type integer. The Query Evaluator will then decide upon an evaluation flow based on the combination of reference types for reference 1 and reference 2. The decision making follows that of the activity diagram as shown below.

Tracing the activity diagram, we follow the steps:

1. Evaluate ref1
    a. Is Synonym
        i. Is unevaluated synonym
2. Evaluate ref2
    a. Is Integer

This leads to the following evaluation flow:

1. For each Value of ref 2, call getDesignAbstractionValues(ref2)
2. Store the results under ref1 in the intermediate answer table.

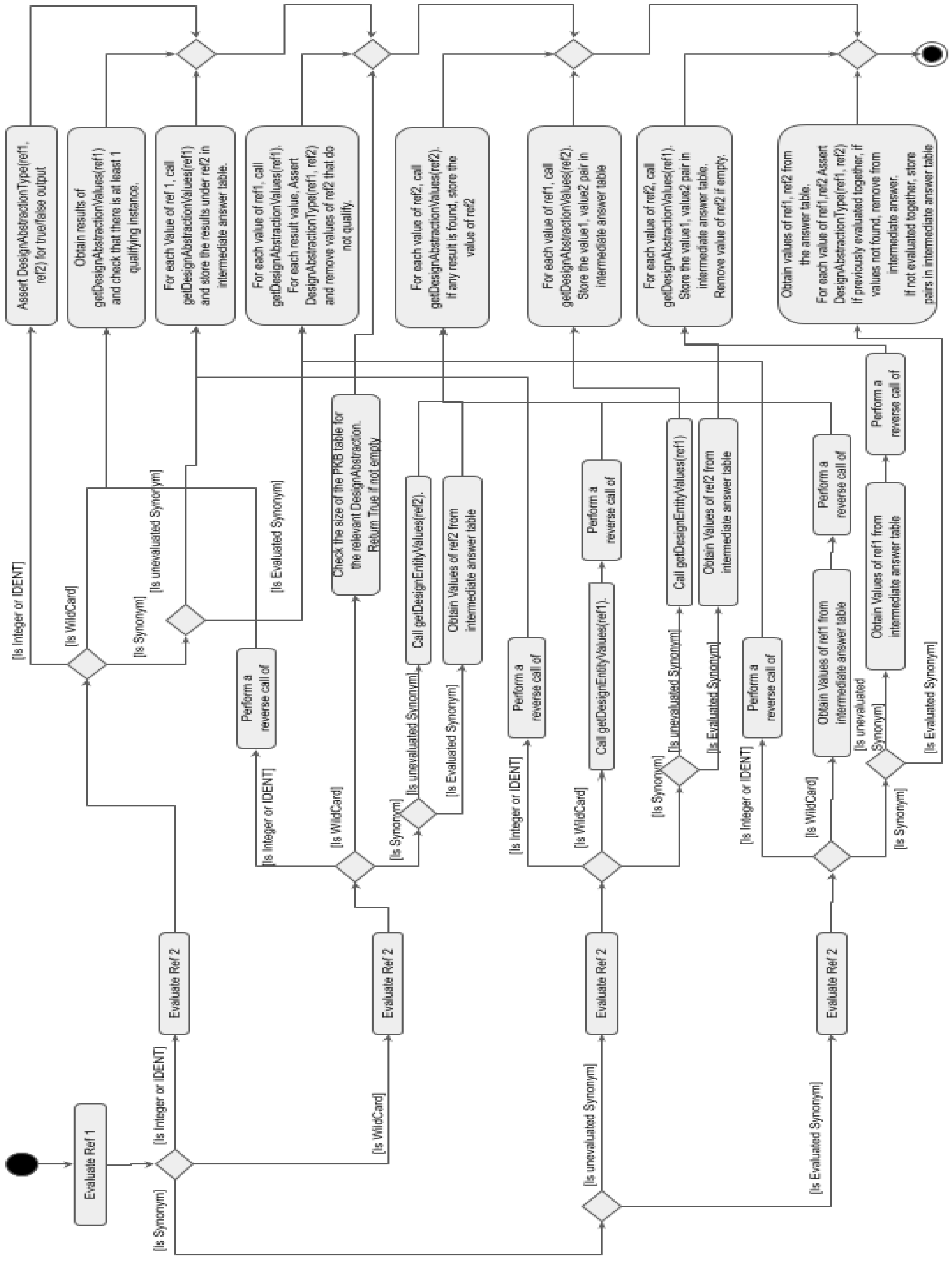

Evaluate Ref 1
[Is Integer or IDENT]
[Is WildCard]
[Is Synonym]
[Is unevaluated Synonym]
[Is Evaluated Synonym]
Evaluate Ref 2
Evaluate Ref 2
Evaluate Ref 2
Evaluate Ref 2
[Is Integer or IDENT]
[Is WildCard]
[Is Synonym]
[Is unevaluated Synonym]
[Is Evaluated Synonym]
Perform a reverse call of
Check the size of the PKB table for the relevant DesignAbstraction. Return True if not empty
Call getDesignEntityValues(ref2).
Obtain Values of ref2 from intermediate answer table
Perform a reverse call of
Call getDesignEntityValues(ref1).
Perform a reverse call of
Call getDesignEntityValues(ref1)
Obtain Values of ref2 from intermediate answer table
Perform a reverse call of
Obtain Values of ref1 from intermediate answer table
Perform a reverse call of
Obtain Values of ref1 from intermediate answer table
Perform a reverse call of
Assert DesignAbstractionType(ref1, ref2) for true/false output
Obtain results of getDesignAbstractionValues(ref1) and check that there is at least 1 qualifying instance.
For each Value of ref 1, call getDesignAbstractionValues(ref1) and store the results under ref2 in intermediate answer table.
For each value of ref1, call getDesignAbstractionValues(ref1). For each result value, Assert DesignAbstractionType(ref1, ref2) and remove values of ref2 that do not qualify.
For each value of ref2, call getDesignAbstractionValues(ref2). If any result is found, store the value of ref2
For each value of ref1, call getDesignAbstractionValues(ref2). Store the value1, value2 pair in intermediate answer table
For each value of ref2, call getDesignAbstractionValues(ref1). Store the value1, value2 pair in intermediate answer table. Remove value of ref2 if empty.
Obtain values of ref1, ref2 from the answer table. For each value of ref1,ref2 Assert DesignAbstractionType(ref1, ref2) If previously evaluated together, if values not found, remove from intermediate answer. If not evaluated together, store pairs in intermediate answer table

Figure 3.2.5.1, Activity Diagram for `such that` clause evaluation

Evaluation algorithm for pattern clause

Pattern if/while

Example Pattern While Clause: `pattern w(“var1”, _)`

Pattern Clause	
ClauseType	`pattern`
DesignAbstractionType	`none`

Reference
ReferenceAttributeType
ReferenceType
DesignEntityType
	`w`
	`none`
	`while`
	`synonym`

Reference
ReferenceAttributeType
ReferenceType
DesignEntityType
	`var1`
	`none`
	`integer`
	`stmt`

PatternAbstractionType	`assign`

Expression
ExpressionType
	`_`
	`no match`

To evaluate a pattern clause, we must first determine the reference type of the 2 references and decide the evaluation flow based upon the reference types, similar to what we do for `such that` clauses. However, for the evaluation of pattern clauses, there is the additional need to determine the PatternAbstractionType, which could be assign, while, and if. If and While patternAbstractionTypes are evaluated similarly to `such that` clauses, where reference 1 and reference 2 reference types are sufficient to determine the evaluation order. Furthermore, if and while pattern clauses may not have a wildcard, or a Quotation (Specified IDENT) as the first reference.

To help illustrate the navigation of the table below, for the clause `pattern w(“var1”, _)`, we will check Reference 1, which is a synonym, and Reference 2, which is an Integer/Quotation. The Query Evaluator will then decide upon an evaluation flow based on the combination of reference types for reference 1 and reference 2. The decision making follows that of the activity diagram as shown below.

Tracing the activity diagram, we follow the steps:

1. Evaluate ref1
    a. Is unevaluated synonym
2. Evaluate ref2
    a. Is Integer

This leads to the following evaluation flow:

1. Obtain results of getDesignEntityValues(ref2)
2. Stores the results of the above step in the intermediate answer table

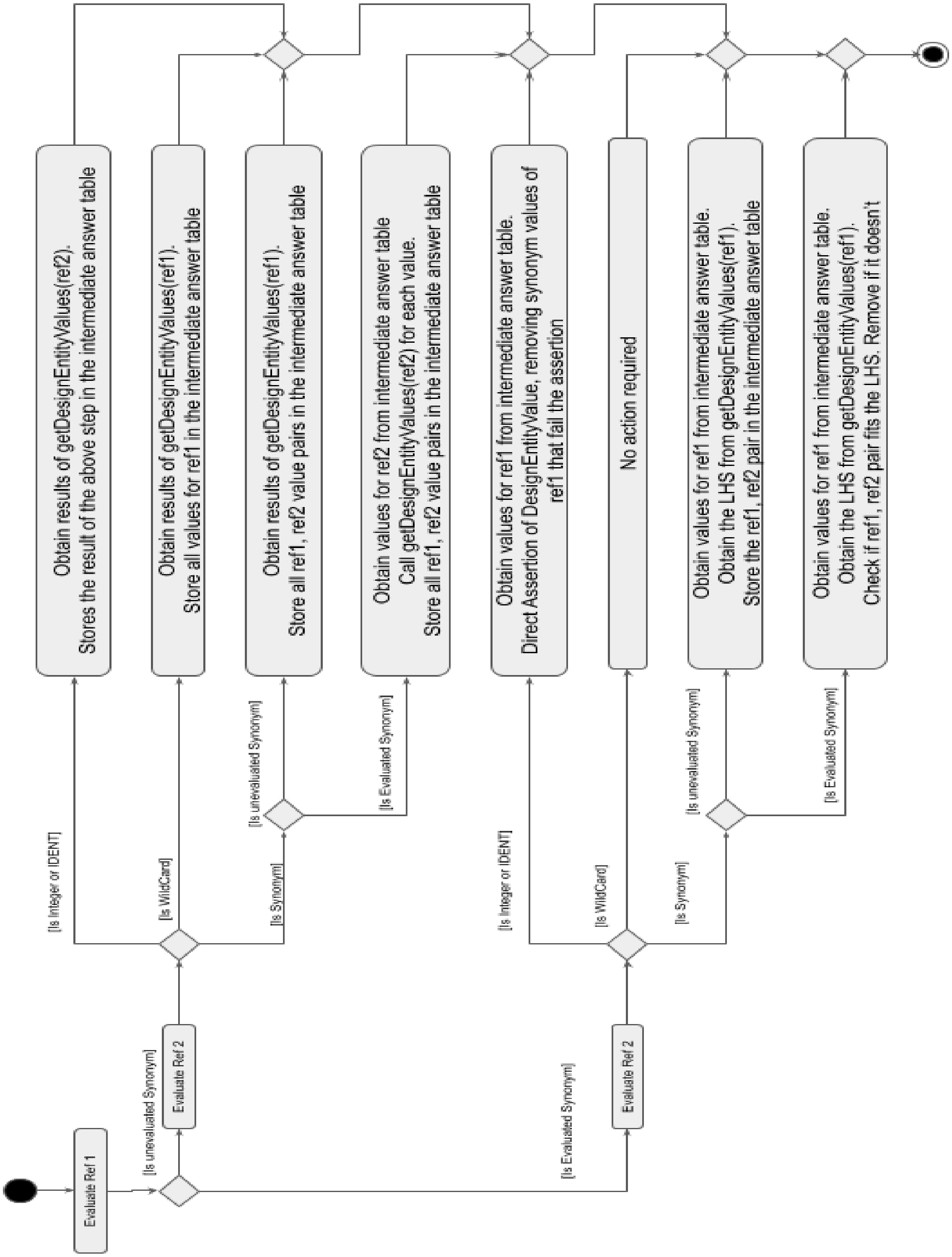

Evaluate Ref 1
[Is unevaluated Synonym]
Evaluate Ref 2
[Is Evaluated Synonym]
Evaluate Ref 2
[Is Integer or IDENT]
[Is WildCard]
[Is Synonym]
[Is unevaluated Synonym]
[Is Evaluated Synonym]
[Is Integer or IDENT]
[Is WildCard]
[Is Synonym]
[Is unevaluated Synonym]
[Is Evaluated Synonym]
Obtain results of getDesignEntityValues(ref2).
Stores the result of the above step in the intermediate answer table
Obtain results of getDesignEntityValues(ref1).
Store all values for ref1 in the intermediate answer table
Obtain results of getDesignEntityValues(ref1).
Store all ref1, ref2 value pairs in the intermediate answer table
Obtain values for ref2 from intermediate answer table
Call getDesignEntityValues(ref2) for each value.
Store all ref1, ref2 value pairs in the intermediate answer table
Obtain values for ref1 from intermediate answer table.
Direct Assertion of DesignEntityValue, removing synonym values of
ref1 that fail the assertion
No action required
Obtain values for ref1 from intermediate answer table.
Obtain the LHS from getDesignEntityValues(ref1).
Store the ref1, ref2 pair in the intermediate answer table
Obtain values for ref1 from intermediate answer table.
Obtain the LHS from getDesignEntityValues(ref1).
Check if ref1, ref2 pair fits the LHS. Remove if it doesn't

Figure 3.2.5.2, Activity Diagram for `pattern if/while` *clause evaluation*

Pattern Assign

Example Pattern Assign Clause: `pattern a(v, “_”)`

Pattern Clause	
ClauseType	`pattern`
DesignAbstractionType	`none`

Reference
ReferenceAttributeType
ReferenceType
DesignEntityType
	`a`
	`none`
	`assign`
	`synonym`

Reference
ReferenceAttributeType
ReferenceType
DesignEntityType
	`v`
	`none`
	`synonym`
	`variable`

PatternAbstractionType	`assign`

Expression
ExpressionType
	`_`
	`no match`

For pattern abstraction type `assign`, there is one additional variable, the ExpressionType, that determines which expression matching algorithm is called. There are 3 possible matching algorithms, no match, partial match, and exact match, specified by *‘_’*, *‘_’expr’_’*, and *expr* respectively. It is also not possible for assign pattern statements to have wildcards or quotations (Specified IDENT) as the first reference.

To help illustrate the navigation of the table below, for the clause `pattern a(v, _)`, we will check Reference 1, which is a synonym, and Reference 2, which is a synonym.There is no match required. The Query Evaluator will then decide upon an evaluation flow based on the combination of reference types for reference 1 and reference 2. The decision making follows that of the activity diagram as shown below.

Tracing the activity diagram, we follow the steps:

1. Evaluate ref1
    a. Is unevaluated synonym
2. Evaluate ref2
    a. Is unevaluated synonym

This leads to the following evaluation flow:

1. Search the PKB for all assign statements
    a. No match required
2. Store all qualifying statements in the intermediate answer table.

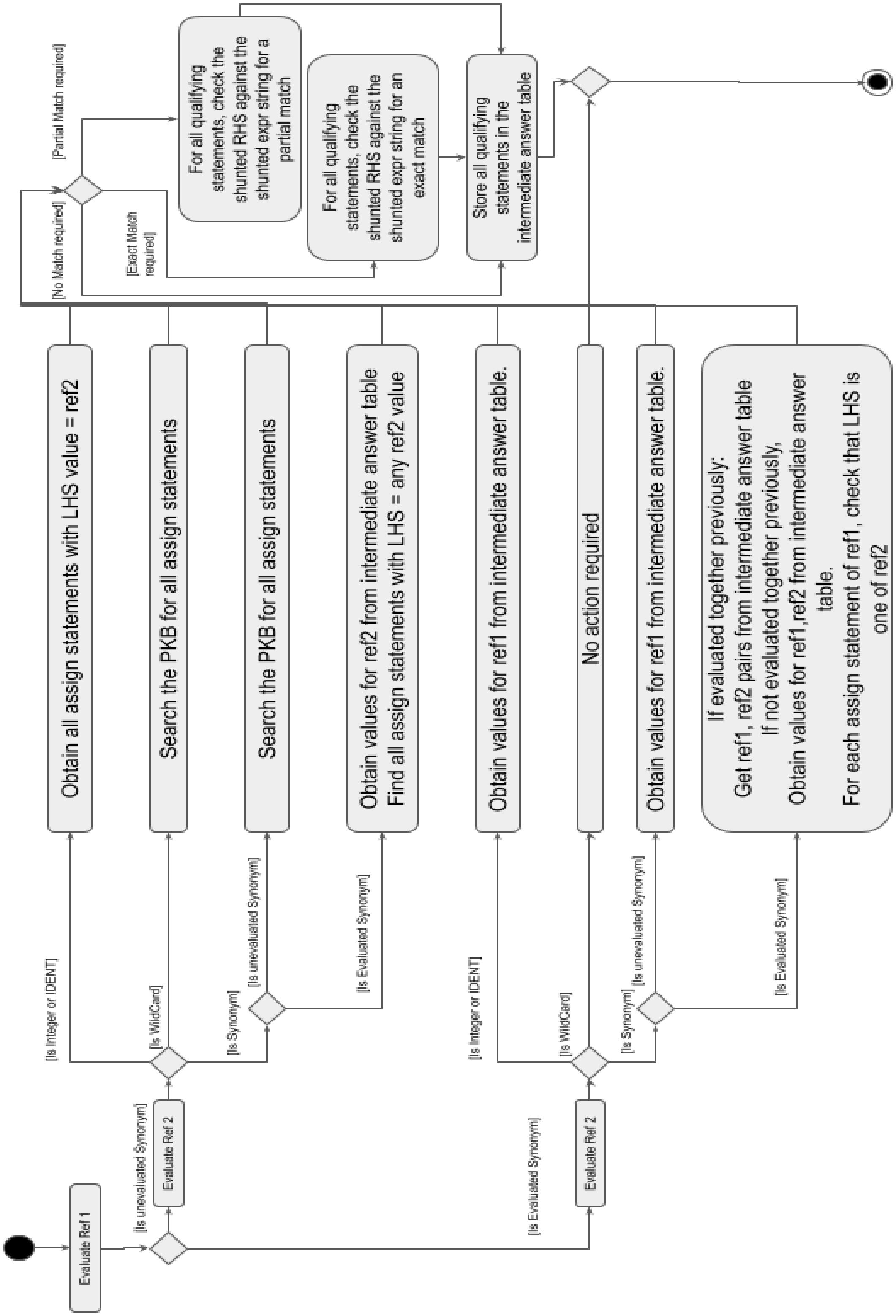

Evaluate Ref 1
[Is unevaluated Synonym]
[Is Evaluated Synonym]
Evaluate Ref 2
Evaluate Ref 2
[Is Integer or IDENT]
[Is WildCard]
[Is Synonym]
[Is unevaluated Synonym]
[Is Evaluated Synonym]
Obtain all assign statements with LHS value = ref2
Search the PKB for all assign statements
Search the PKB for all assign statements
Obtain values for ref2 from intermediate answer table
Find all assign statements with LHS = any ref2 value
[Is Integer or IDENT]
[Is WildCard]
[Is Synonym]
[Is unevaluated Synonym]
[Is Evaluated Synonym]
Obtain values for ref1 from intermediate answer table.
No action required
Obtain values for ref1 from intermediate answer table.
If evaluated together previously;
Get ref1, ref2 pairs from intermediate answer table
If not evaluated together previously,
Obtain values for ref1,ref2 from intermediate answer table.
For each assign statement of ref1, check that LHS is one of ref2
[No Match required]
[Partial Match required]
[Exact Match required]
For all qualifying statements, check the shunted RHS against the shunted expr string for a partial match
For all qualifying statements, check the shunted RHS against the shunted expr string for an exact match
Store all qualifying statements in the intermediate answer table

Figure 3.2.5.3, Activity Diagram for pattern assign clause evaluation

Evaluation algorithm for with clause

Example With Clause: `with v1.varName = v.varName`

With Clause	
ClauseType	`with`
DesignAbstractionType	`none`

Reference
ReferenceAttributeType
ReferenceType
DesignEntityType
	`v1.varName`
	`varName`
	`attribute`
	`variable`

Reference
ReferenceAttributeType
ReferenceType
DesignEntityType
	`v.varName`
	`varName`
	`attribute`
	`variable`

To evaluate a `with` clause, it is necessary to first determine the reference types for both references. The evaluation flow is dependent on the combination of reference types, similar to the evaluation algorithm for `such that` clauses.

In the example given above, the reference type for reference 1 is an attribute, and the reference type for reference 2 is an integer. The Query Evaluator will then decide upon an evaluation flow based on the combination of reference types for reference 1 and reference 2. The decision making follows that of the activity diagram as shown below:

Tracing the activity diagram, we follow the steps:

1. Evaluate ref1
    a. Is Attribute (synonym)
2. Evaluate ref2
    a. Is Integer

This leads to the following evaluation flow:

1. The PKB is searched for all possible values of the synonym in ref 1.
2. If an attribute comparison is required, the attribute value of the possible synonym values are looked-up in the pkb
3. For each attribute-synonym pair, an evaluation against the ref2 is performed.
4. All qualifying attribute-synonym pair has the synonym value added to the intermediate answer table.

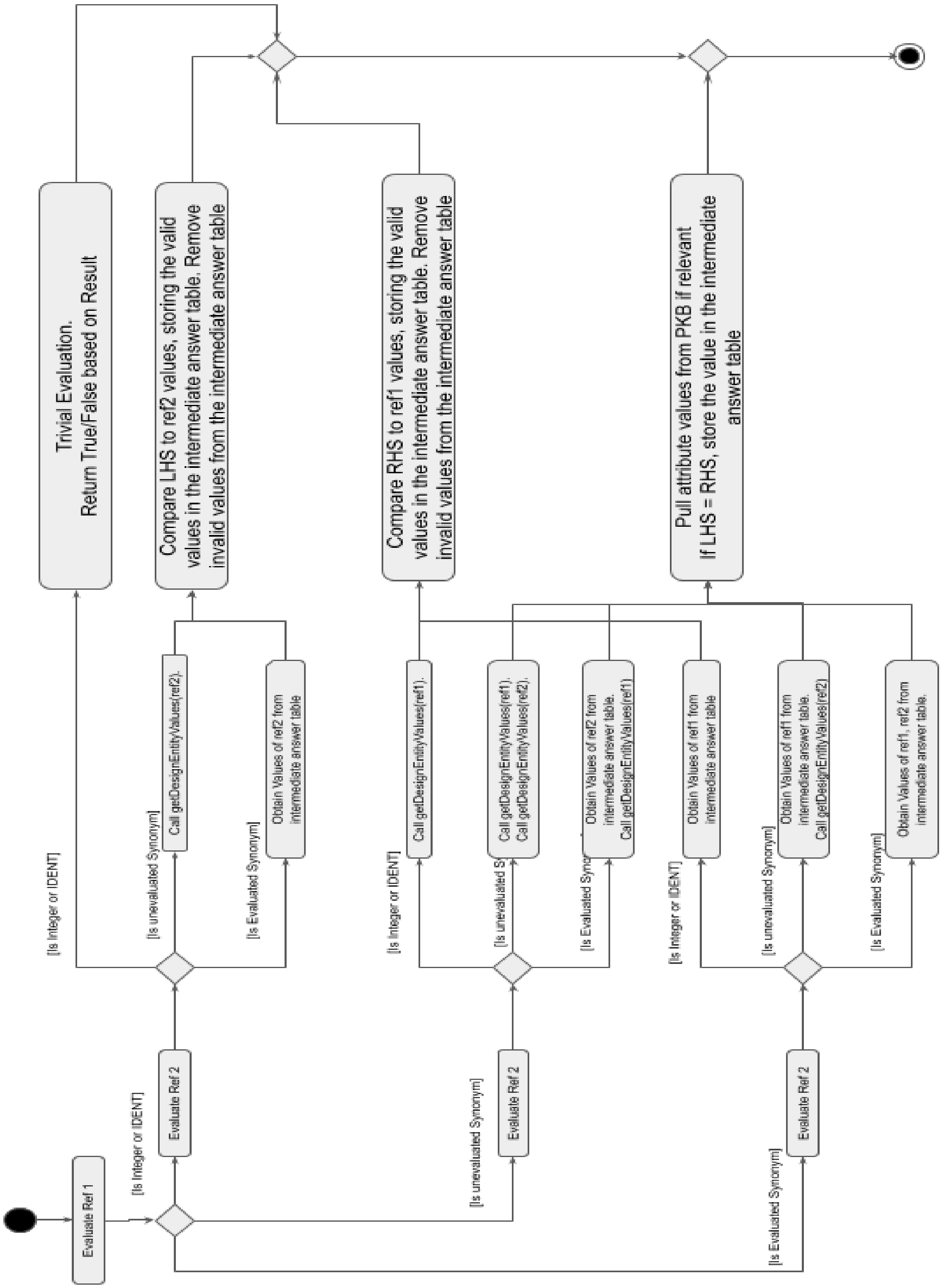

Evaluate Ref 1
[Is Integer or IDENT]
[Is unevaluated Synonym]
[Is Evaluated Synonym]
Evaluate Ref 2
Evaluate Ref 2
Evaluate Ref 2
[Is Integer or IDENT]
[Is unevaluated Synonym]
[Is Evaluated Synonym]
Call getDesignEntityValues(ref2).
Obtain Values of ref2 from intermediate answer table
[Is Integer or IDENT]
[Is unevaluated Sy
[Is Evaluated Syno
Call getDesignEntityValues(ref1).
Call getDesignEntityValues(ref1). Call getDesignEntityValues(ref2).
Obtain Values of ref2 from intermediate answer table. Call getDesignEntityValues(ref1)
[Is Integer or IDENT]
[Is unevaluated Synonym]
[Is Evaluated Synonym]
Obtain Values of ref1 from intermediate answer table
Obtain Values of ref1 from intermediate answer table. Call getDesignEntityValues(ref2)
Obtain Values of ref1, ref2 from intermediate answer table.
Trivial Evaluation. Return True/False based on Result
Compare LHS to ref2 values, storing the valid values in the intermediate answer table. Remove invalid values from the intermediate answer table
Compare RHS to ref1 values, storing the valid values in the intermediate answer table. Remove invalid values from the intermediate answer table
Pull attribute values from PKB if relevant If LHS = RHS, store the value in the intermediate answer table

Figure 3.2.5.4, Activity Diagram for `with` clause evaluation (A larger diagram can be found in Appendix E)

Evaluation algorithm for select clause

Select boolean

The default return for a Select Boolean clause is {“TRUE”}. This enables the trivial query:

```
Select BOOLEAN
```

To return a {“TRUE”} when evaluated.

For more complex queries, the select boolean clause will return the cumulativeClauseReturn, a variable that is computed in the overarching evaluation flow, in a list.

Select elem

This is the trivial case for select tuple, where the tuple has size 1.
Elem can be either a synonym, synonym.attrName

In the case of synonyms, all values of the synonym in the intermediate answer table will be compiled and returned. If the synonym has not been evaluated before, values of the synonym’s design entity are obtained using the getDesignEntityValue function, and the values are compiled and returned. The output format is a list of strings, with each string being a possible value for the synonym.

In the case of synonym.attrName, the evaluation is similar to the case of synonyms, with the additional step of obtaining the value of the relevant attrName from the PKB tables. The output format is a list of strings, with each string being a possible value for the synonym.attrName.

Select Tuple

The evaluation of select tuple, is a multiple step process, which involves the steps for selecting elem.
Firstly, the elements of the tuple are checked to determine if they have been evaluated together. For elements that have been evaluated together, the possible permutations of values for each elem that has satisfied the query can be recursively traced from the intermediate answer table.

For elements that have not been evaluated together with other elements or groups of elements, there is a need to permute the result, as well as the result for other elements or groups that have not been evaluated together.

Example:

```
stmt a, b, c; Select <a, b, c> ...
```

In the query above, synonyms a and b have been evaluated together in a group, but not synonym c.
There is therefore a need to permute through the results of <a, b>, and the results of c. If there are n1 results from the synonym group <a, b>, and n2 results from the synonym c, there will be n1*n2 total results for <a, b, c>
The return type is a list of strings, where each string is one result, with the values of items separated by a space. Eg. {“2 3 4”, “10 12 18”, …, “32 1 23”}

Formatting of Results

The result expected by the AutoTester from the QueryEvaluator, is a list of strings for select synonym|attribute|tuple queries, or a list containing either a “TRUE” or “FALSE” string for select boolean queries. The formatting step is performed at the end evaluation algorithm for the select clause.

QueryPKBInterface

The QueryPKBInterface is a subcomponent of the Query Evaluator. The Query Evaluator algorithms for solving the different clauses: select, `such that`, pattern, and with, all make calls to the PKB indirectly through the QueryPKBInterface.

The Query Evaluator mostly deals with strings - string variable values, string comparisons between possible synonym values, etc. This is due to the implementation style chosen, where similar evaluation flows are abstracted out despite the evaluation flows potentially evaluating multiple Design Abstractions. This gives rise to the possibility of evaluating Modifies(s,v) and Follows(s1,s2), using the same segment of code. However, synonym v for the Modifies Design Abstraction is of type string, whereas synonym s2 for the Follows Design Abstraction is of type integer. Therefore, there is a need for standardization, and strings were chosen due to the more expressive nature.

The QueryPKBInterface helps to resolve the Query Evaluator's dependence on strings, and the PKB's interface methods taking in a variety of different inputs - string, integers, lists of strings. The QueryPKBInterface helps to format the output of PKB calls as well, which include complex return types, such as maps and sets of strings.

The usage of a component that is dedicated to performing PKB calls also allows the decoupling of the Query Evaluator from the PKB's concrete API. If there is any change in the PKB, only the method in the QueryPKBInterface has to be altered, and the Query Evaluator can still use the same evaluation flows for the clauses, as mentioned in the sections above.

QueryDEInterface

The QueryDEInterface is a subcomponent of the Query Evaluator. The Query Evaluator algorithm makes calls to the Design Extractor through the QueryDEInterface, particularly for evaluating the Next*, Affects, and Affects* design abstractions.

Similar to the function of the QueryPKBInterface, the QueryDEInterface modifies the output from the Design Extractor into the format (strings) that is requested by the Evaluator. The QueryDEInterface serves the same function of decoupling the Query Evaluator from the DesignExtractor's concrete API.

Optimizer

The optimizer is signalled to function by a boolean flag, which can be set in the Test Wrapper. There are two key methods used for optimisation, namely, the grouping of clauses by the number of synonyms involved in the clause, and the sorting of clauses using a dynamic scoring system.

Grouping Clauses by number of synonyms

The first strategy utilized, is to group the clauses based on the synonyms in the clauses. Clauses without any synonyms are grouped together in a default group. Clauses with synonyms that are evaluated together will be grouped in the same group.

Example

```
stmt s1,s2,s3; assign a1; if ifs; variable v;
Select v such that Parent(6, 10) and Follows(s1,s2) and Follows(1, s1) and
Follows(s2,s3) pattern a1(v, _) and ifs(v,_,_) with s3.stmt# = 3
```

The above query will be grouped as follows

Default Group	Group 1	Group 2
Such that Parent(6,10)	Such that Follows(s1,s2), such that Follows(1,s1), such that Follows(s2,s3), with s3.stmt# = 3	Pattern a1(v, _), pattern ifs(v, _,_)

Table 3.2.5.2 Grouping Example

By grouping the clauses, we seek to minimise occurrences where the Query Evaluator has to compare “unevaluated” synonyms, where the synonyms have to be initialised with full values from the PKB. By evaluating all the clauses with common synonyms first, we are able to quickly reduce the possible values of each synonym, and potentially end the evaluation process early, should a synonym run out of possible values, and result in the clause evaluating to false.

Sorting Clauses using a static, and dynamic scoring system.

After grouping the clauses into different groups, the Optimizer will then assign a static score based on the clause type and dynamic scoring based on the area(sum of the size of all entries) of the relevant table. This will allow the Query Evaluator to evaluate the simpler clauses first. In cases where the static scoring might be equal due to the clause types, dynamic scoring will be used instead. Dynamic scoring is first based on the number of synonyms in a clause. If that is also equal, the area of the relevant tables are used instead. The area is a sum of the sizes of all value sets in a table. The area of the tables are precomputed before query evaluation, except for non-PKB-storable tables such as Affects. The following table shows the example of the scoring for each group and X is used as a placeholder for the dynamic area scoring.

Group 0	Group 1	Group 2
Such that Parent(6,10) [2, 0, X]	Such that Follows(s1,s2) [1, 2, X] such that Follows(1,s1) [1, 1, X] such that Follows(s2,s3)	Pattern a1(v, _) [3, 1, X] pattern ifs(v, _,_) [2, 1, X]

	[1, 2, X] with s3.stmt# = 3 [0, 1, X]	

Table 3.2.5.3 Scoring of clauses [<static>, <dynamic synonym>, <dynamic area>]

Design Considerations

One key design consideration is the type of data structure used to store the intermediate answers. In the planning phase, two ideas for storing intermediate answers were considered: A table, and a system of hashmaps with reverse storage.

Criteria for Evaluation	**Option 1: Storing intermediate answers in a table**	**Option 2: Storing intermediate answers in a hashmap system**
Implementation Difficulty	Implementation is less complicated and less challenging	Implementation is more challenging.
Insertion performance	Values can be copied from the array (from PKB) into table in O(n).	Values can be inserted into the hash map system at O(1) time while iterating through the array at O(n) time.
Deletion performance	The entire table has to be iterated through, row-wise, in order to find values to remove. This will be performed in O(n) time	Values in the hashmap system can be removed at O(1) time
Combination performance	When evaluating 2 sets of clauses with synonyms that are not evaluated together, a cross product operation has to be performed. This is typically performed in O(n^3)	When synonyms that are not evaluated together are stored, the performance is the same as a typical insertion, and can be performed in O(1) time.
Extraction performance	Extraction can be performed in O(n) time by iterating through the table row-wise and selecting rows that are valid.	For Select element Extraction is relatively quick at O(n) time. However, for select tuple, if elements that are not related are selected, permutation needs to be performed, and the operation will take O(n^3) time.

We have decided to go with option 2. Combination of synonyms that are not evaluated together can not be entirely avoided without the sorting of clauses or other optimisation steps. However, it is more beneficial to only have to perform the combination step once at the end of the evaluation of the query. By performing combinations at the end, there is also the possibility that the synonyms evaluation will be more connected after every single clause has been evaluated. Conversely, if we went with option 1, a bad order of clauses could result in having to perform the combination step multiple times, slowing performance.

3.2.6 Shunting Yard Component

To parse expressions, we first process them using the Shunting Yard algorithm, where it converts expressions from infix into postfix. This component is being used by both the front end and the PQL where they need to parse expressions before storing or checking against the tables in PKB respectively. In the Front-End Parser, **`parseExpression`** shunts the expression for storage in the PKB assignTable.

The tweak that we made to the standard Shunting Yard algorithm was that we needed to handle variable names instead of just constants and single character variable names. This was due to the fact that our expressions had variable names that consisted of more than 1 character and could contain a mix of alphabets and numbers.

To illustrate this, given an example of "`var1+var2`", `var1` and `var2` are variables that have variable names of more than 1 character and consist of numbers. We would then tokenize this expression first:

1) Keep reading in tokens and if it is not an operator, add it to a string called *token*. In this case it will read characters 'v', 'a', 'r', '1' until it moves on to step 2.

2) When it encounters an operator, it adds *token* to a LIST_OF strings and adds the operator as a string to the LIST_OF strings too.

3) When it reaches the end of the expression, returns the LIST_OF strings as the result.

After this tokenization is done on the expression, we can now run the Shunting Yard algorithm but on a LIST_OF strings instead of just a string. This allows us to run the Shunting Yard algorithm on not only variables with variable names of 1 character, but also adapt it to our use case with variable names larger than 1 character.

Example

A procedure contains the following assignment statement:

```
var = var1 * var2 + (var3 - var4 / var5) / var6;
```

The Front-End Parser would process the expression on the RHS of the statement in two steps:

1. The expression would first be decomposed into the following tokens: var1, *, var2, +, var3, -, var4, /, var5, /, var6
2. Then, the Shunting Yard algorithm would be run on the list of tokens, producing the final shunted expression "var1 var2 * var3 var4 var5 / - var6 / +" to be stored in the PKB assignTable

3.3 Component Interactions

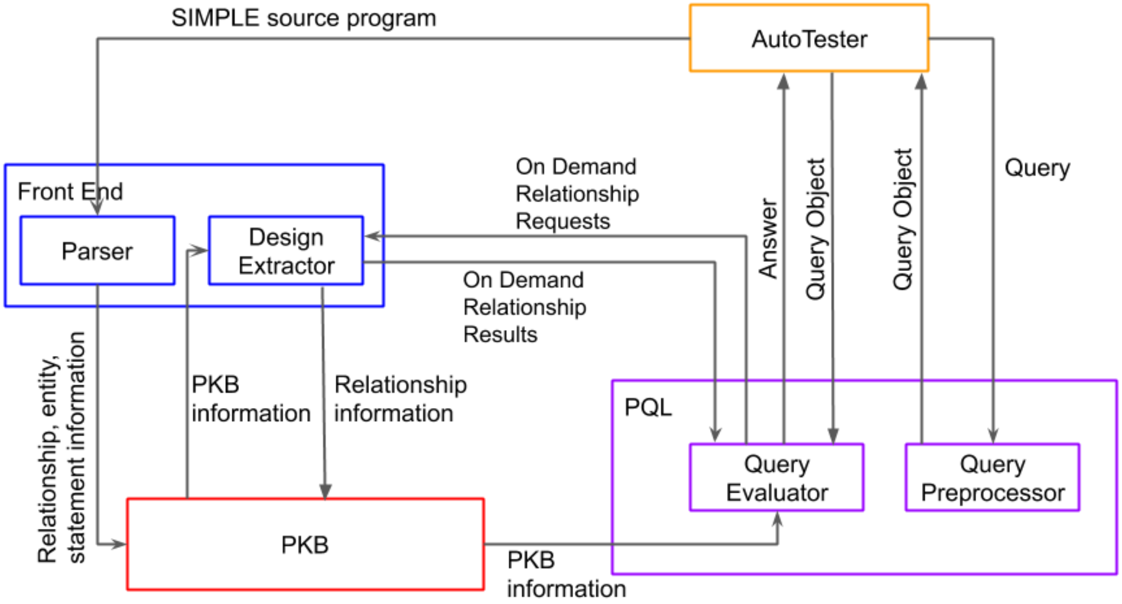


Figure 3.3.1 Component Interactions

Our team has developed this figure shown in *Figure 3.3.1* to help us with identifying the flow and interactions between the different components of the program. During the project planning phase, once we have split the team members into the different components (Front End, PKB, PQL) to work on, the team members in each component would mostly interact with each other as their segments are closely interlinked. For instance, the query evaluator would require the query preprocessor and parser would input the information in the PKB to be used by the design extractor.

Components in each segment (PKB, PQL and Front End) would communicate among themselves until they are done with their unit tests. Once the unit tests are done, integration tests have to be done as such:

1) Front End
   a) Parser → PKB
   b) Design Extractor → PKB

2) PQL
   a) Query Preprocessor → Query Evaluator
   b) Query Evaluator → PKB

During the integration testing phase, there would be communication between the different components as shown in the list above.

Having this structure in our project has not only allowed us to better understand which team members are in charge of which component but just in case if we are facing any issues or if we do need any updates on certain components, we can usually ask the other teammate that is working within the same segment.

4. Extensions to SPA

We decided to implement the recommended extensions for Iteration 3 which are the design abstractions `NextBip`, `NextBip*`, `AffectsBip` and `AffectsBip*.`
A detailed documentation of these design abstractions will be explained in the following sections.

4.1 Extension Definitions

This section explains the definitions of the extensions that we implemented.

`NextBip`
For any program lines n_1 and n_2, relationship `NextBip`(n_1, n_2) holds if n_2 can be executed immediately after n_1 in some execution sequence, including branching into and back from procedures.

`NextBip*`
For any program lines n_1 and n_2, relationship `NextBip*`(n_1, n_2) holds if n_2 can be executed after n_1 in some execution sequence, including branching into and back from procedures.

`AffectsBip`
For any assignments a_1 and a_2, relationship `AffectsBip`(a_1, a_2) holds if

- Assignment statement a_1 modifies value of variable v and a_2 uses the value of variable v
- There is a control flow path from a_1 to a_2 i.e., `NextBip*`(a_1, a_2) – such that v is not modified in any assignment statement on that path

`AffectsBip*`
For any assignments (a_1, a_2), relationship `AffectsBip*`(a_1, a_2) holds if

- There is a chain of a_i ($1 \le i \le n$) such that `AffectsBip`(a_i, a_{i+1}) and there is a control flow path in `CFGBip` on which `NextBip*`(a_i, a_{i+1}) for ($1 \le i \le n-1$)

4.2 Implementations Details

This section demonstrates and explains the implementation of the extensions and the involvement of different components.

4.2.1 Front-End Involvement

4.2.1.1 Creating `CFGBip` and Extracting `NextBip` relationships

To be able to extract all the relationships for the extensions, we first construct a program CFG, `CFGBip`, to represent possible execution paths in the source program. The CFG is stored in the form of an adjacency list, with a mapping of nodes to the nodes adjacent to them.

This is achieved by extracting each procedure individually with the help of

- A nodeIndex to keep track of the current node being processed
- A callStack to keep track of current calls to other procedures

To better explain the creation process, we will use the following sample SIMPLE source program:

```
    procedure One {
1      a = b;
2      call Two;
3      c = d;
    }
    procedure Two {
4      if(x == y) then {
5         e = f;
       } else {
6         g = h;
       }
    }
```

Creating `CFGBip` across all procedures

For each procedure, the following steps are taken in the function `constructProcedureCFG`:

1. First, we set up a custom queue to store nodes to be visited next. This queue stores the line number to be visited next, and a set of nodes directly before this line number. We also define the following sets:
    - **previouslySkippedWhileLines** - stores while statement lines that have been visited once
    - **previouslyVisitedWhileLines** - stores while statement lines that have been visited twice
    - **terminalNodes -** stores all nodes with statement lines without a `Next` relationship with another statement, or nodes with while statement lines with only one Next relationship
2. Then, we check if the callStack is empty. If it is empty, we push the first statement of the procedure with an empty set into the BFS queue. If it is not empty, we the first statement of the procedure with the top element of the callStack into the BFS queue. The purpose construction of each element in the BFS is explained later.
3. Then, we iterate through all statements in this procedure using BFS as follows

Breadth-First Search for each procedure to generate CFGBip

For each element in the BFS, we do the following steps. Note that the element should contain a set of previous nodes, referred to as **prevNodes**, and the current line number, referred to as **currLine.**

<table>
<tr><th colspan="3">Initial Check: Check if previouslyVisitedWhileLines contains the currLine</th></tr>
<tr><td colspan="3">If it does, it means that the current line number, which is a while statement, has already been visited twice. We populate an edge between all prevNodes and the nodes that represent this currLine in the CFGBip

<b>We also move on to the next element in the BFS queue, skipping all the steps below.</b></td></tr>
<tr><td colspan="3">Create a new node, <b>currentNode,</b> which stores the current line number.
Next, we add an edge between all prevNodes and currentNode.

<b>We then handle the currLine depending on the below cases:</b></td></tr>
<tr><td>Case 1: currLine is call statement</td><td colspan="2">We do a recursive call to <b>constructProcedureCFG</b> with the argument being the procedure called in the call statement. The currentNode is also pushed into the callStack. This returns the terminal nodes of the procedure.
We then check if there is a statement with a next relationship(nextLine) to the call statement. If so, we insert all the terminal nodes, and the nextLine into the BFS. If there isn’t we simply return the terminal nodes.</td></tr>
<tr><td rowspan="3">Case 2: currLine is a while statement</td><td>Case 2a:
currLine has been skipped previously (previouslySkippedWhileLines contains currLine)</td><td>We add this currLine into previouslyVisitedWhileLines.</td></tr>
<tr><td>Case 2b:
currLine has not been skipped previously</td><td>We add this currLine into previouslySkippedWhileLines</td></tr>
<tr><td>Case 2c:
currLine has only one line that can be executed next(nextLine)</td><td>currLine is a terminal while statement, we currentNode into a list of terminal nodes terminalNodes</td></tr>
<tr><td rowspan="2">Case 3: currLine is not a call or while statement</td><td>Case 3a:
currLine doesn’t have another statement s1 with a Next(currLine, s1) relationship.</td><td>We push the currentNode into a list of terminal nodes terminalNodes</td></tr>
<tr><td>Case 3b:
currLine has statements s1 with a Next(currLine, s1) relationship</td><td>We add the currentNode and nextline into the queue.</td></tr>
</table>

After a CFGBip has been constructed, the NextBip relationships for the entire source program can easily be extracted by adding a NextBip relationship between statement lines attached to any two nodes whenever there is an edge between two nodes.

Illustration of Constructing a CFGBip and extracting NextBip

Using the code example above, we illustrate how a CFGBip is constructed and how the NextBip relationships can be extracted.

constructProcedureCFG(0) - constructing procedure CFG for procedure One

Current BFS element	Case	Nodes Created (Node Index, Statement No.)	Description
({}, 1}	3b	(1, 1)	{ {1} , 2 } is added into the queue
{ {1} , 2 }	1	(1,1), (2, 2)	Edge between nodes 1 and 2 is created. constructProcedureCFG(1) is called

constructProcedureCFG(1) - constructing procedure CFG for procedure Two

Current BFS element	Case	Nodes (Node Index, Statement No.)	Description
({2}, 4 }	3b	(1,1), (2,2), (3,4)	Edge between nodes 2 and 3 is created. { {3}, 5} and { {3}, 6} is added into the queue
{ {3}, 5 }	3a	(1,1), (2,2), (3,4), (4,5)	Edge between 3 and 4 is created. The currentNode, 4 is pushed into terminalNodes. terminalNodes now contain 4
{ {3}, 6 }	3a	(1,1), (2,2), (3,4), (4,5), {5,6}	Edge between 3 and 5 is created. The currentNode, 5 is pushed into terminalNodes. terminalNodes now contain 4 and 5.

constructProcedureCFG(0) - constructing procedure CFG for procedure One

Current BFS element	Case	Nodes Created (Node Index, Statement No.)	Description
{ {1} , 2 }	1	(1,1), (2, 2)	Edge between nodes 1 and 2 is created. constructProcedureCFG(1) is called
Refer to table constructProcedureCFG(1) - exiting recursive call to constructProcedureCFG(1) and returning {4,5} back to constructProcedureCFG(0)			
{ {1} , 2 }	1	(1,1), (2,2), (3,4), (4,5), {5,6}	Since constructProcedureCFG(1) returns {4,5}, and statement 2 has a nextLine 3, we push the following into the queue: {{4,5}, 3}
{{4,5}, 3}	3a	(1,1), (2,2), (3,4), (4,5), {5,6}, {6.3}	Edge between nodes 4 and 6 is created. Edge between nodes 5 and 6 is created.

Now that a CFGBip has been constructed, we simply iterate through all of the nodes in the CFGBip, adding a NextBip relationship the statement lines that between any two nodes that have an edge.

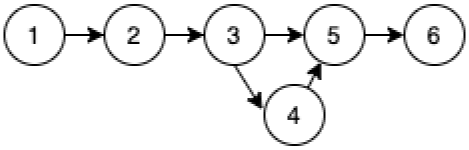
For example, since there is an edge between 4 and 6 in the CFG, and node 4 is attached to statement line 5 while node 6 is attached to statement line 3, a NextStar relationship will be populated between statement line 6 and 3.

4.2.1.2 Generating `NextBip*` Relationships

Using the CFGBip constructed previously, we can easily extract the NextBip* relationships for all the nodes in the CFGBip by applying Breadth-First Search(BFS) on every node. Since BFS has already been explained numerous times in the Design Extractor documentation, we simply explain the extraction here with a simple example:

Illustration of Extraction of NextBip* relationships from the CFGBip constructed

Given that we have constructed the following CFGBip earlier



Mapping of Nodes to Statement Lines	
1	1
2	2
3	4
4	5
5	6
6	3

During the iteration of 1, we use BFS to find all the nodes that it is connected to, adding the following NextBip* relationships: (1,2), (1,3), (1,4), (1,5), (1,6).

Note that a statement line may have more than one node since a statement line can be visited by another path in the CFG.

4.2.1.3 Creating CFGAffectsBip and Generating `AffectsBip` Relationships

The AffectsBip relationship for all the statement lines in source program can be extracted by using the CFGBip constructed previously to create an a special CFG - CFGAffectsBip that represents a graph with nodes that only have an edge connected to each other if they have an AffectsBip relationship.

This is accomplished by iterating through all the nodes in the CFGBip to extract the AffectsBip relationship for that particular statement. This extraction is almost identical to that of the algorithm used to extract the Affects relationship, with the key difference being that that algorithm uses the getNext to continuously iterate through the procedure statement by statement, while this algorithm uses the adjacency list to continuously iterate through the CFGBip node by node.

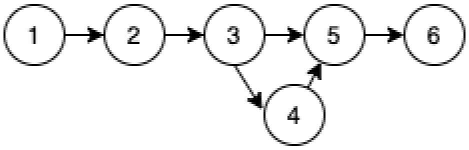
To keep this explanation brief, we will only discuss an example to showcase the difference between the two, and how the CFGAffectsBip is constructed with a similar algorithm

```
    procedure One {
1      a = b;
2      call Two;
3      c = b;
    }
    procedure Two {
4      if(x == y) then {
5         b = a;
       } else {
6         a = h;
       }
    }
```

Example: Construction of CFGAffectsBip and Extracting AffectsBip

Using the code example above, we illustrate how a CFGAffectsBip can be constructed and how all AffectsBip relationship can be extracted from it.

The following represents the CFGBip constructed and the mapping of nodes ot statement lines



Mapping of Nodes to Statement Lines	
1	1
2	2
3	4
4	5
5	6
6	3

Starting from node 1(statement 1), we check that it is an assignment statements, and we blacklist node 1(statement 1) and node 5(statement 6) since they modifies a. Note that although statement 4 modifies a , we do not blacklist it since it does not directly modify a, similar to the algorithm for Affects relationship.

Next, a BFS is initiated in CFGBip for node 1, and the following are some noteworthy flows;

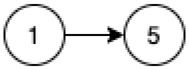
1 -> 2 -> 3 -> 4

Here, we see that node 4 (statement 5) uses variable a which is modified by node 1. Hence, we constructed an edge between node 1 and node 5.

1 -> 2 -> 3- > 5

The next nodes from statement 3 can either be 4 or 5. However, since 5 is blacklisted, we visit it to check if there is an Affects relationship, but do not consider any execution flows next of 5 to visit next in our BFS.

Finally, after the BFS is completed on Node 1, we have the following CFGAffectsBip constructed so far:



We can then simply populate the AffectsBip relationship of (1,5) to store in the PKB.

4.2.1.4 Generating `AffectsBip`* Relationships

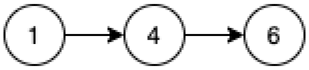
Using the CFGAffectsBip constructed previously, we can then extract the AffectsBip* relationship by applying Breadth-First Search on every node in the CFGAffectsBip. This algorithm is similar to that of the NextBip* extraction. We illustrate more using the example below.

Using the same code example from AffectsBip,

```
    procedure One {
1      a = b;
2      call Two;
3      c = b;
    }
    procedure Two {
4      if(x == y) then {
5         b = a;
       } else {
6         a = h;
       }
    }
```

Example: Extracting AffectsBip* relationships from Affects relationships

The CFGAffectsBip constructed will look like the following:



In order to extract the AffectsBip* relationship, we simply visit every node in the CFGAffectsBip and use BFS to iterate through all the nodes that it is connected to and add an AffectsBip* relationship between the statements associated with the nodes. In this case, during the BFS for node 1, we populate the relationships AffectsBip*(1, 5) and AffectsBip*(1,3) since node 4 is associated with statement 5 and node 6 is associated with statement 3.

4.2.2 PKB Involvement

NextBip and NextBipStar

NextBip is an object that contains both the *NextBip* and *PreviousBip* tables. Similarly, the *NextBipStar* is an object that contains both the *NextBipStar* and *PreviousBipStar* tables.

The mappings for each of the tables are as follows:

1) *NextBip* table - LineNumber:Set_of_<NextBipLines>
2) *PreviousBip* table - LineNumber:Set_of_<PreviousBipLines>
3) *NextBipStar* table - LineNumber:Set_of_<NextBipStarLines>
4) *PreviousBipStar* table - LineNumber:Set_of_<PreviousBipStarLines>

Given the following SIMPLE program:

```
     procedure Moondrop {
1        blessing = 2;
2        if(blessing == 2) then {
3            solis = s + blessing;
         } else {
4            call Sundrop;
         }
5        s = blessing + 5;
     }
     procedure Sundrop {
6        blessing = 2;
7        s = blessing + 5;
     }
```

The 4 tables listed above would contain the following mappings after Front End is done with parsing and design extraction.

Line Number	NextBip Line
1	<2>
2	<3, 4>
3	<5>
4	<5, 6>
6	<7>

Table 4.2.2.1 NextBip Table

Line Number	PreviousBip Line
2	<1>

3	<2>
4	<2>
5	<3, 4>
6	<4>
7	<6>

Table 4.2.2.2 PreviousBip Table

Line Number	NextBipStar Line
1	<2, 3, 4, 5, 6, 7>
2	<3, 4, 5, 6, 7>
3	<5>
4	<5, 6, 7>
6	<7>

Table 4.2.2.3 NextBipStar Table

Line Number	PreviousBipStar Line
2	<1>
3	<1, 2>
4	<1, 2>
5	<1, 2, 3, 4>
6	<1, 2, 4>
7	<1, 2, 4, 6>

Table 4.2.2.4 PreviousBipStar Table

AffectsBip and AffectsBipStar

AffectsBip is an object that contains both the *AffectsBip* and *AffectedByBip* tables. Similarly, the *AffectsBipStar* is an object that contains both the *AffectsBipStar* and *AffectedByBipStar* tables.

The mappings for each of the tables are as follows:

1) *AffectsBip* table - LineNumber:Set_of_<AffectsBipLines>
2) *AffectedByBip* table - LineNumber:Set_of_<AffectedByBipLines>
3) *AffectsBipStar* table - LineNumber:Set_of_<AffectsBipStarLines>
4) *AffectedByBipStar* table - LineNumber:Set_of_<AffectedByBipStarLines>

Given the following SIMPLE program:

```
     procedure Moondrop {
1        blessing = 2;
2        if(blessing == 2) then {
3            solis = s + blessing;
4            read solis;
5            solis = solis + 1;
         } else {
6            s = 1;
7            call Sundrop;
         }
8        s = blessing + solis;
     }
     procedure Sundrop {
9        blessing = blessing + 2;
10       s = s + 5;
11       blessing = s + blessing;
     }
```

The 4 tables listed above would contain the following mappings after Front End is done with parsing and design extraction.

Line Number	AffectsBip Line
1	<3, 8, 9>
5	<8>
6	<10>
9	<11>
10	<11>

Table 4.2.2.5 AffectsBip Table

Line Number	AffectedByBip Line
3	<1>
8	<1, 5>
9	<1>
10	<6>
11	<9, 10>

Table 4.2.2.6 AffectedByBip Table

Line Number	AffectsBipStar Line
1	<3, 8, 9, 11>
5	<8>
6	<10>
9	<11>
10	<11>

Table 4.2.2.7 AffectsBipStar Table

Line Number	AffectedByBipStar Line
3	<1>
8	<1, 5>
9	<1>
10	<6>
11	<1, 9, 10>

Table 4.2.2.8 AffectedByBipStar Table

AffectsBipNode and AffectedByBipNode

AffectsBipNode is an object that contains both the *AffectsBipNode* and *AffectedByBipNode* tables.

The mappings for each of the tables are as follows:

1) *AffectsBipNode* table - LineNumber:Set_of_<AffectsBipNodeLines>
2) *AffectedByBipNode* table - LineNumber:Set_of_<AffectedByBipNodeLines>

4.2.3 PQL Involvement

4.2.3.1 Query Preprocessor

The Query preprocessor will have to validate the new relationship type according to the new grammar rules. There are not many changes in the parsing and validation and the following *Tables 4.2.3.1.1* and *4.2.3.1.2* show the left and right hand side validation for the extended relationships.

	stmt	read	print	call	while	if	assign	variable	constant	procedure	prog_line
NextBip	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✕/✕	✕/✕	✕/✕	✓/✓
NextBip*	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✕/✕	✕/✕	✕/✕	✓/✓
AffectsBip	✓/✓	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✓/✓	✕/✕	✕/✕	✕/✕	✓/✓
AffectsBip*	✓/✓	✕/✕	✕/✕	✕/✕	✕/✕	✕/✕	✓/✓	✕/✕	✕/✕	✕/✕	✓/✓

Table 4.2.3.1.1 Left / Right Hand Side Design Entities extension relationship validation

	INTEGER	“_”	“IDENT”
NextBip	✓/✓	✓/✓	✕/✕
NextBip*	✓/✓	✓/✓	✕/✕
AffectsBip	✓/✓	✓/✓	✕/✕
AffectsBip*	✓/✓	✓/✓	✕/✕

Table 4.2.3.1.2 Left /Right Hand Side arguments extension relationship validation

4.2.3.2 Query Evaluator

For the Query Evaluator, the following extensions are required in order to deal with the new relationships:

In the Query Object, the DesignAbstractions `NextBip`, `NextBip*`, `AffectsBip`, and `AffectsBip*` are added. In the QueryPKBInterface, the relevant methods for interfacing with the PKB for information are added. These include:

1. getNextBip(), getNextBipStar(), getAffectsBip(), getAffectsBipStar().
2. isNextBip(), isNextBipStar(), isAffectsBip(), isAffectsBipStar().

The overarching evaluation flow and major evaluation algorithms for each clause type are not affected significantly, as they are constructed to be flexible in adaptation. Once the dynamic calling method getDesignAbstractionValues() and assertDesignAbstraction() are expanded to include the new relationships, and the QueryPKBInterface is updated accordingly to be able to obtain the information, the QueryEvaluator is capable of evaluating queries involving the new relations.

4.3 Implementation Schedule

In order to meet the deadline for the project, the team has adhered to the following schedule to be able to implement the extensions and fulfill the requirements of iteration 3:

Timeline	Implementations	Members Involved
18 Oct -23 Oct	Planning of extension (`AffectsBip`, `NextBip`, `AffectsBip*`, `NextBip*`)	Olivia, Zhang Xian
	Planning of optimization for PQL queries	Mei Yen, Zhihui, Jia Hao
23 Oct - 25 Oct	Implemented `Affects`, `Affects*` along with unit testing	Olivia, Zhang Xian, Mei Yen, Zhihui
	Integration testing for `Affects`, `Affects*`	
	Regression testing	
24 Oct - 27 Oct	Extending PKB to handle extensions	Olivia, Zhang Xian
27 Oct - 1 Nov	Implemented `NextBip`, `NextBip*` along with unit testing	Olivia, Zhang Xian
	Integration testing for `NextBip`, `NextBip*`	
	Regression testing	
	Implemented `NextBip, NextBip*`, `AffectsBip`, `AffectsBip*` for Query Preprocessor and Query Evaluator	Zhihui, Mei Yen
1 Nov	System testing for system excluding extensions	Olivia, Zhang Xian, Jia Hao, Mei Yen, Zhihui, Xiu Qi
2 Nov - 5 Nov	Implemented `AffectsBip, AffectsBip*` along with unit testing	Olivia, Zhang Xian
	Integration testing for `AffectsBip`, `AffectsBip*`	
	Regression testing	
6 Nov	System and stress testing for overall system with extensions	Olivia, Zhang Xian, Jia Hao, Mei Yen, Zhihui, Xiu Qi
7 Nov - 8 Nov	Optimization of extensions	Olivia, Zhang Xian

Table 4.3.1 Implementation Schedule

Following the above schedule, we managed to implement all the extensions and requirements of iteration 3. This schedule shows that we are able to develop and integrate the extensions and requirements into the system incrementally and iteratively by the deadline.

4.4 Test Plan

Testing	Timeline	Details
Unit Testing	23 Oct - 25 Oct	Unit Testing for `Affects`, `Affects*`
Integration Testing		Integration Testing for `Affects`, `Affects*`
Regression testing		Regression Testing for `Affects`, `Affects*`
Unit Testing	27 Oct - 1 Nov	Unit Testing for `NextBip`, `NextBip*`
Integration Testing		Integration Testing for `NextBip`, `NextBip*`
Regression testing		Regression Testing for `NextBip`, `NextBip*`
System Testing	1 Nov	System Testing for system without extensions
Unit Test	2 Nov - 5 Nov	Unit Testing for `AffectsBip`, `AffectsBip*`
Integration Testing		Integration Testing for `AffectsBip`, `AffectsBip*`
Regression Testing		Regression Testing for `AffectsBip`, `AffectsBip*`
Stress Testing	6 November	Stress Test for system with extensions
System Testing		System Test for system

Table 4.4.1 Test Plan

Rationale

After each new feature has been implemented, unit testing is first conducted to ensure that the feature is bug-free on its own, this means that unit testing will be conducted the most frequently while features are added incrementally into the system. After each unit testing has been conducted for individual components, integration testing between every 2 components is then conducted to ensure that components are well integrated in the system. Then, system testing is conducted to ensure that all components in the system deliver all features as promised bug-free.

On top of the above, regression testing is conducted to ensure that new features do not cause any regressions to the system after it has been implemented into it.

Finally, stress testing is conducted to make sure that the system remains functional under heavy workload.

Responsibilities

All unit testing and integration testing will be conducted by the person in charge of each component after implementing it. Separately, Xiu Qi will be in charge of the writing system test, individual clause test for `Affects` and `Affects*`, and stress test cases. Additionally, Mei Yen will be in charge of the system test for the extensions and Jia Hao will be assisting with the evaluation of the system tests.

4.5 Extension System Testing

To ensure that the newly added Extensions (`NextBip, NextBip*, AffectsBip and AffectsBip*`) are working as intended, the two source programs have multiple call statements (`complexBip_source`) and nested container statements (`extremeBip_source`) to test. Single clauses are tested first followed by multiple clauses and stress testing for optimisation. This was done to ensure that the extensions are well-tested. The extension test consists of two SIMPLE source text file inputs which have multiple calls statements and multiple nesting container statements, which will test the new design abstractions for the extensions. The queries are varied and consists of a mix of synonym, fixed integer value or even a wildcard.

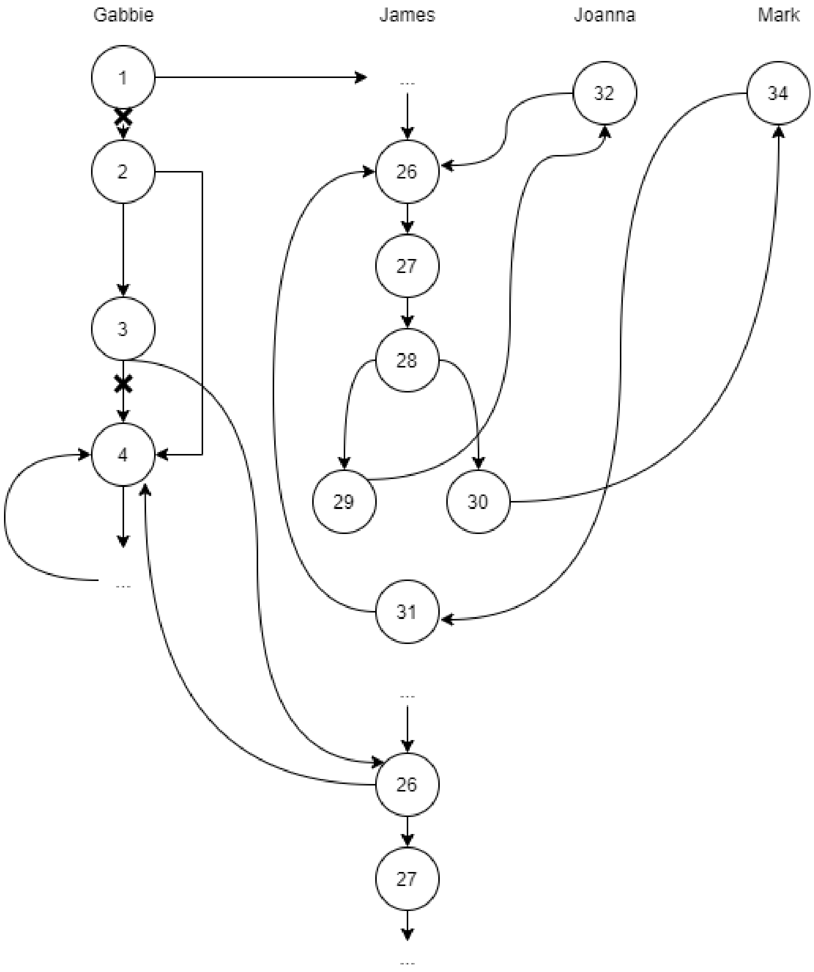


Figure 4.5.1: CFG snippet for complexBip_source SIMPLE program

An example of a extension testing query in Autotester format is as follows:

Extension System Test (Testing Specific Clause/Relationship)

Test Purpose: Tests for the extraction of `NextBip`, and `AffectsBip`. Query 1 tests the edge case where the last statement of the ‘James’ procedure is a while statement while Query 19 tests the case to see if the variable has been modified in another procedure.

Required Test Inputs:
The test input required would be the complex bip source SIMPLE program and the list of queries associated with that test case. The query input has to be in the Autotester format as such:

```
1 - NextBip while multiple calls
stmt s;
Select s such that NextBip(s, 26)
32,34,31
5000
```

Expected Test Results: 32, 34, 31

```
19 - AffectsBip
stmt s;
Select s such that AffectsBip(34, s)
34,31,27
5000
```

Expected Test Results: 34, 31, 27

It has to match the output given in the query.

An example of a stress testing query in Autotester format is as follows:

Extension System Test (Stress testing)

Test Purpose: Test on 4 clauses with no common synonyms, stress test caching

Required Test Inputs:
The test input required would be the complex bip source SIMPLE program and the list of queries associated with that test case. The query input has to be in the Autotester format as such:

```
46 - NextBip* no common stress testing
Prog_line p, p1, p2, p3, p4, p5; stmt s, s1;
Select s1 such that NextBip*(s, s1) and NextBip*(p, p1) and NextBip*(p2, p3) and
NextBip*(p4, p5)
...
5000
```

Expected Test Results: Answer omitted for brevity
It has to match the output given in the query.

4.6 Extension Implementation Challenges

Some of the challenges we faced during implementation of the extension include:

- Existing logic for extracting Next/* and Affects/* could not be adapted to work for the extension
- The nature of the extension meant that NextBip* and AffectsBip* were not trivial transitive closures of NextBip and AffectsBip; in the end, we gathered that a CFG must be constructed to extract Next* and AffectsBip* accurately
- The choice of data structures and algorithm is not trivial: As there are many paths within a more complex CFG, there will be many nodes that need to be traversed, and since we are doing BFS over almost the entire set of nodes, reducing time complexity was a challenge
- As a result of challenge faced in previous point, testing the program on our test cases was a problem as it took a long time initially to run the test cases and hence discover the flaws in our implementation

For PKB, we did not face notable implementation challenges as the tables to be implemented are similar to existing tables, e.g. AffectsBipStarTable and AffectsStarTable. The same applies for PQL as extending the grammar for the NextBip/* and AffectsBip/* design is similar to what has been done over the past few iterations.

5. Coding standards and API development experiences

For coding standards, our team has decided upon using the default Visual Studio formatting style as given in Microsoft Visual Studio which follows Microsoft's style guide.

5.1 Naming

1) The C++ files should have the format of .cpp while header files have the format of .h.
2) We aim to use camel case when naming variables and functions within our code. In general, variables and functions names are separated by an underscore if a word is preceded by an acronym or abbreviation, for the sake of clarity. E.g. `getLongestRHS_AssignExtracted()`.
3) Function names are verbs or verb phrases. E.g. `getAllKeysAssign(), printSynonymTable()`

5.2 Formatting

Most of the formatting is following Microsoft's style guide which can be found here: https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/code-style-rule-options?view=vs-2019

Our team has also adopted using 4 spaces for indentation only.

5.3 Abstract PKB API

One way that we improved the correspondence between the abstract and concrete APIs is to make our concrete function names and arguments exactly the same as their counterparts in the abstract API.

When writing the abstract API for PKB, we noticed that many tables share similarities. Another way that we improved the correspondence between the abstract and concrete APIs is to first split the abstract API by types of keys and values. Using this, we then developed several abstract classes that can be inherited by other classes to be built upon. For instance, for the PKB segment, we created a *PKB_Map_Integer_String* class and various tables such as the *Variable* table and *Read* table inherited from it.

Benefits and problems of abstract PKB API

One benefit of working with an abstract PKB API in our program is that we only have to create a main table for each category such as *PKB_Map_Integer_String* and have other tables that are of that same mapping inherit directly from it. That way, we only have to develop the logic for the common methods once and the rest of the tables can just inherit from it. If there are any changes that need to be made, we can directly pinpoint it to one location.

However, the disadvantage of doing so is that it results in a lot of header files to be created and that we would have to include the correct header files into the program. This has sometimes resulted in certain issues especially when integrating our codes together.

6. Testing

Different tests were done throughout the phase of the project. When each individual segment is developed, they would have to undergo unit testing first to ensure that it passes. Each team member that is developing their component would also create the unit tests for their respective components.

After passing all unit tests, the component would then be integrated with the other components and integration testing would then be done. Regression testing is also done to ensure that the changes made to the code does not affect what was previously developed.

After ensuring that it passes all integration testing, we would then proceed onto system testing, where it encompasses all aspects of the program.

There are 2 rounds of system testing:

Testing	Timeline	Details	Members Involved
Unit Testing	23 Oct - 25 Oct	Unit Testing for `Affects`, `Affects*`	Olivia, Zhang Xian, Mei Yen, Zhihui
Integration Testing		Integration Testing for `Affects`, `Affects*`	
Regression testing		Regression Testing for `Affects`, `Affects*`	
Unit Testing	27 Oct - 1 Nov	Unit Testing for `NextBip`, `NextBip*`	Olivia, Zhang Xian, Mei Yen, Zhihui
Integration Testing		Integration Testing for `NextBip`, `NextBip*`	
Regression testing		Regression Testing for `NextBip`, `NextBip*`	
System Testing	1 Nov	System Testing for system without extensions	Olivia, Zhang Xian, Mei Yen, Zhihui, Xiu Qi, Jia Hao
Unit Testing	2 Nov - 5 Nov	Unit Testing for `AffectsBip`, `AffectsBip*`	Olivia, Zhang Xian, Mei Yen, Zhihui
Integration Testing		Integration Testing for `AffectsBip`, `AffectsBip*`	
Regression Testing		Regression Testing for `AffectsBip`, `AffectsBip*`	
Regression Testing	2 Nov - 5 Nov	Regression testing for optimization	Mei Yen, Zhihui, Jia Hao
Stress Testing	6 - 7 Nov	Stress Test for system with extensions	Olivia, Zhang Xian, Mei Yen, Zhihui, Jia Hao

System Testing		System Test for system with extensions	

6.1 Unit Testing

Each component and its subcomponents have their own set of test cases written to ensure that the individual functions are implemented properly and working as intended. Unit testing is done using the in-built Visual Studio Testing Framework. Testing was done during and after implementation of the component.

6.1.1 PKB Unit Testing

PKB Unit Test Sample 1 (Valid and Invalid isParentStar(INT, SET) for ParentStar table)

Test Purpose: Checks if isParentStar(INT, SET_INT) returns True and False appropriately. This test assumes that insertParentStar() is valid, which should have been tested in other unit tests before this one. It was chosen to demonstrate how the ParentStar table is able to automatically fill in the ChildStar table when insertion is done for the ParentStar table.

Required Test Inputs:

```
PKB_Map_ParentStar table;
table.insertParentStar(3, { 5, 6 });
table.insertParentStar(3, { 4 });
table.insertParentStar(4, { 5 });
table.insertParentStar(99999, { 4 });
```

Current table state:

ParentStar table

LineNumber	ParentLines
3	{ 4, 5, 6 }
4	{ 5 }
99999	{ 4 }

ChildStar table

LineNumber	ChildLines
4	{ 3, 99999 }
5	{ 3, 4 }
6	{ 3 }

Expected Test Results: isParentStar(INT, SET_INT) returns True and False appropriately.

```
Assert::IsTrue(table.isParentStar(3, { 4, 5, 6 }));
```
Checks if isParentStar can correctly verify a full set.

```
Assert::IsTrue(table.isParentStar(3, { 4, 5 }));
```
Checks if isParentStar can correctly verify a subset.

```
Assert::IsTrue(table.isParentStar(3, { 4 }));
```
Checks if isParentStar can correctly verify a set with a single element.

```
Assert::IsFalse(table.isParentStar(3, { 7 }));
```
Checks if isParentStar can correctly return FALSE for an existing key with non-existing value.

```
Assert::IsFalse(table.isParentStar(9, { 4 }));
```
Checks if isParentStar can correctly return FALSE for a non-existing key with an existing value.

```
Assert::IsTrue(table.isChildStar(5, { 3, 4 }));
```
Checks if isChildStar can correctly verify a full set.

```
Assert::IsTrue(table.isChildStar(6, { 3 }));
```
Checks if isChildStar can correctly verify a subset.

```
Assert::IsTrue(table.isChildStar(4, { 3,99999 }));
```
Checks if isChildStar can correctly verify a set with a single element.

```
Assert::IsFalse(table.isChildStar(4, { 7 }));
```
Checks if isChildStar can correctly return FALSE for an existing key with non-existing value.

```
Assert::IsFalse(table.isChildStar(9, { 7 }));
```
Checks if isChildStar can correctly return FALSE for a non-existing key with an existing value.

It asserts that the table can appropriately determine if an input key exists, and that the input value is indeed in the values of the existing key.

PKB Unit Test Sample 2 (Invalid Assign RHS insertion)

Test Purpose: Checks if a LHS exists at the same lineNumber when trying to insert a RHS. Chosen to demonstrate how the table looks like when storing expressions.

Required Test Inputs:
```
bool errorThrown = FALSE;
try {
	PKB::getInstance().insertAssignLHS(3, "x");
	PKB::getInstance().insertAssignRHS(4, "y");
}
catch (std::string error) {
	Logger::WriteMessage(error.c_str());
	Assert::IsTrue(error == "No entry exists at lineNumber 4!");
	errorThrown = TRUE;
}
Assert::IsTrue(errorThrown);
```

Current table state:

Assign table

LineNumber	AssignStmt
3	<”x”, “”>

Expected Test Results: An error should be thrown and caught, and the error thrown is the appropriate error.

```
Assert::IsTrue(error == "No entry exists at lineNumber 4!");
```

Checks if the error thrown is the appropriate type.

```
Assert::IsTrue(errorThrown);
```

Checks if an error is thrown.

It asserts that the table can appropriately determine if a LHS already exists for an assign statement before trying to insert a RHS to the same statement.

6.1.2 PQL Unit Testing

PQL Preprocessor Unit Test Sample 1 (Valid Query)

Test Purpose: Checks if the QueryParser processes the query correctly and that the query is syntactically and semantically correct and returns TRUE. Chosen to demonstrate how the preprocessor parses `BOOLEAN` queries.

Required Test Inputs: `Select BOOLEAN such that Next(32, _)`

Expected Test Results: Query is valid, `IsSelectBoolean` flag is set to TRUE, and the expected values match to the actual values.

```
Assert::IsTrue(result.getIsValid() &&
result.getClauses().at(0).designAbstractionType ==
Query::DesignAbstractionType::NEXT && result.getClauses().at(0).leftRef.refString==
"32" && result.getClauses().at(0).rightRef.refString == "_" &&
result.getIsSelectBoolean());

Assert::IsFalse(!result.getIsValid() && !result.getIsSelectBoolean());
```

It asserts that the query is valid and to ensure the expected values are equal to the actual values found in the clauses, and that the select boolean flag is set to TRUE.

PQL Evaluator Unit Test Sample 1 (formatAnswer)

Test Purpose: Checks if the QueryEvaluator is able to format the return string required by Autotester, by retrieving the relevant results from the answerTable. Chosen to demonstrate how the

QueryEvaluator retrieves the result from the table.

Required Test Inputs: `std::LIST_OF<std::string> selectedSyn = {“a”};`
`QueryEvaluator::GetInstance().answerTable;`

The Test case adds a few values in the answerTable:

```
std::unordered_map <std::string,std::unordered_set <std::string>> dummy;
QueryEvaluator::GetInstance().answerTable["a"]["1"] = dummy;
QueryEvaluator::GetInstance().answerTable["a"]["2"] = dummy;
```

Expected Test Results: Syntax Error Exception thrown as query is invalid.

```
it = std::find(answer.begin(), answer.end(), "1");
Assert::IsTrue(it != answer.end());

it = std::find(answer.begin(), answer.end(), "2");
Assert::IsTrue(it != answer.end());

it = std::find(answer.begin(), answer.end(), "5");
Assert::IsTrue(it != answer.end());
```

It asserts that the query is not valid.

6.2 Integration Testing

Integration testing is also done using the in-built Visual Studio Testing Framework. Testing was done during and after implementation of the component in step with the progress of the corresponding linked component.

6.2.1 Design Extractor - PKB Integration Testing

We perform integration testing between the Design Extractor and the PKB to ensure that the correct design relationships are being extracted and PKB is populated with the correct relationships. Since integration testing has also been conducted for the Parser and the PKB, we can assume that the PKB contains all the correct information from parsing the source program.

One sample integration test case is to test the extraction of Follows* relationship.
Given that the PKB contains the following information:

PKB Follows Table	
LineNumber	**<VariableLHS, VariableRHS>**
1	2
2	3
3	4

Table 6.2.1.1 PKB Follows Table

The sequence diagram that showcases the communication between this extraction is as follows:

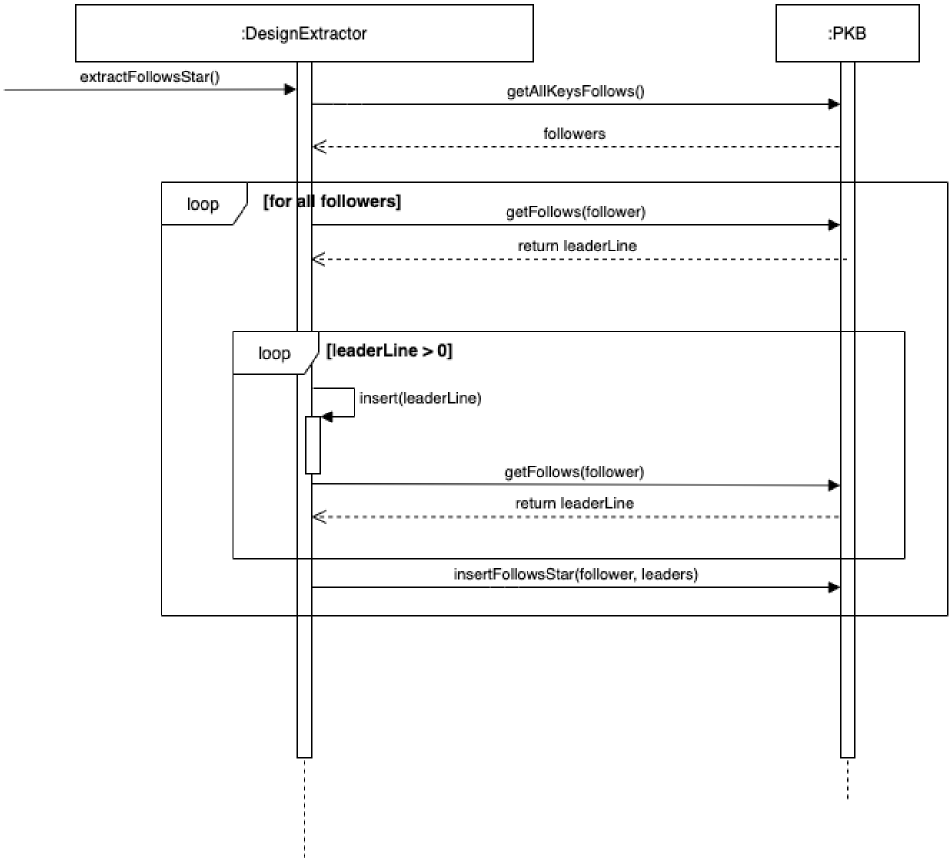


Figure 6.2.1.1 Design Extractor - PKB interaction

After extracting the `Follows*` relationship and populating it into the PKB, the PKB should now store the correct `Follows*` relationship. We test this by using assertions to check for a few statements. The following are the test cases used.

```
Assert::AreEqual(TRUE,            PKB::getInstance().getFollowsStar(4)           ==
std::unordered_set<int>{ 1, 2, 3 });
```

```
Assert::AreEqual(TRUE,            PKB::getInstance().getFollowsStar(3)           ==
std::unordered_set<int>{ 1, 2 });
```

Other extensive test cases with varying complexity were also used to test the integration between PKB and Design Extractor.

6.2.2 PQL Evaluator - PKB Testing

We perform Integration Testing between PQL QueryEvaluator and PKB to ensure that the calls to PKB from the QueryEvaluator are called with the right arguments, and to ensure that the return type from the PKB is properly handled in the QueryEvaluator.

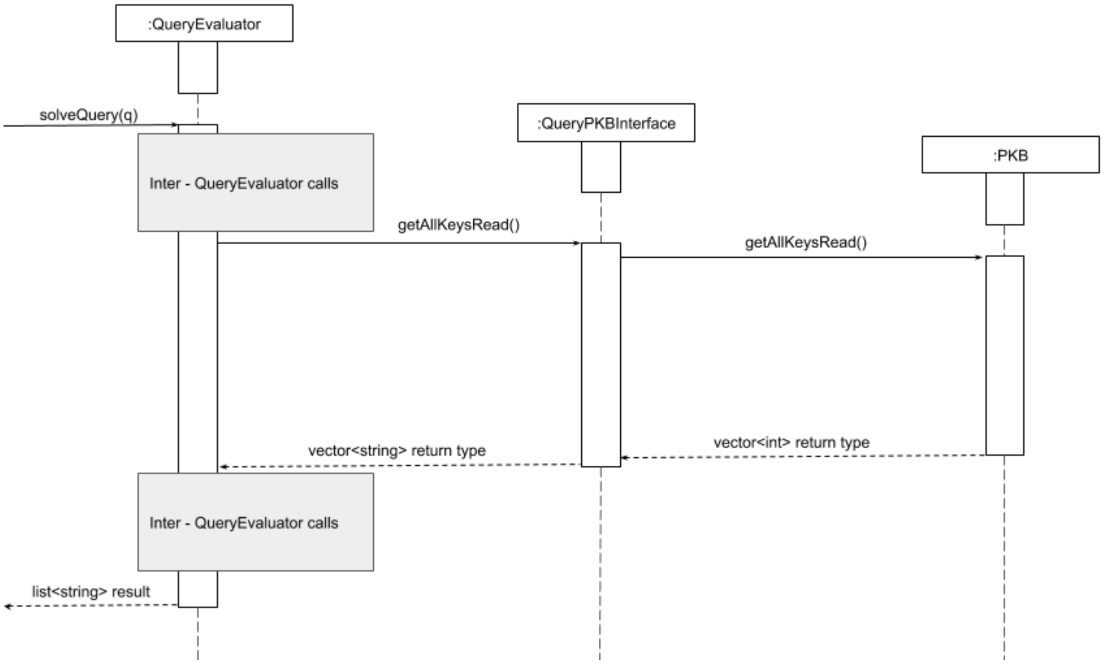


Figure 6.2.2.1 PQL Evaluator - PKB interaction

The below example depicts an example situation where the Query Preprocessor and the Query Evaluator evaluates the following query:

Integration Testing (Testing Valid BOOLEAN Query)

Test Purpose: Checks if the QueryEvaluator is calling getDesignEntityValues correctly in the evaluation order.

Required Test Inputs:
The PKB is required to be initialized with a target dummy value.

```
PKB::getInstance().insertRead(2, “var1”);
```

A query object must be created to query this information.

```
Query q = Query();
q.setIsValid();
Query::Element e
e.elementType = Query::ElementType::SYNONYM
e.synonymString = “rd”
std::vector<Query::Element> selectedSyns= { e }
q.addSelectedSynonyms(selectedSyns)

std::list<std::string> result = QueryEvaluator::getInstance().solveQuery(q)
```

Expected Test Results: The Expected value in the result list is {“2”}

```
it = std::find(answer.begin(), answer.end(), "2");
Assert::IsTrue(it != answer.end());
Assert::IsFalse(it = answer.end());

Assert::IsTrue(result.size() == 1);
```

It asserts that the expected answer is equal to the actual answer output from the solveQuery method.

6.2.3 PQL Preprocessor - PQL Evaluator Testing

We perform Integration Testing between PQL Preprocessor and PQL Evaluator to ensure that the relevant information needed is extracted from the query correctly in the PQL Preprocessor and inserted into the Query object, where the PQL Evaluator will get the information and evaluate accordingly to output the correct answer.

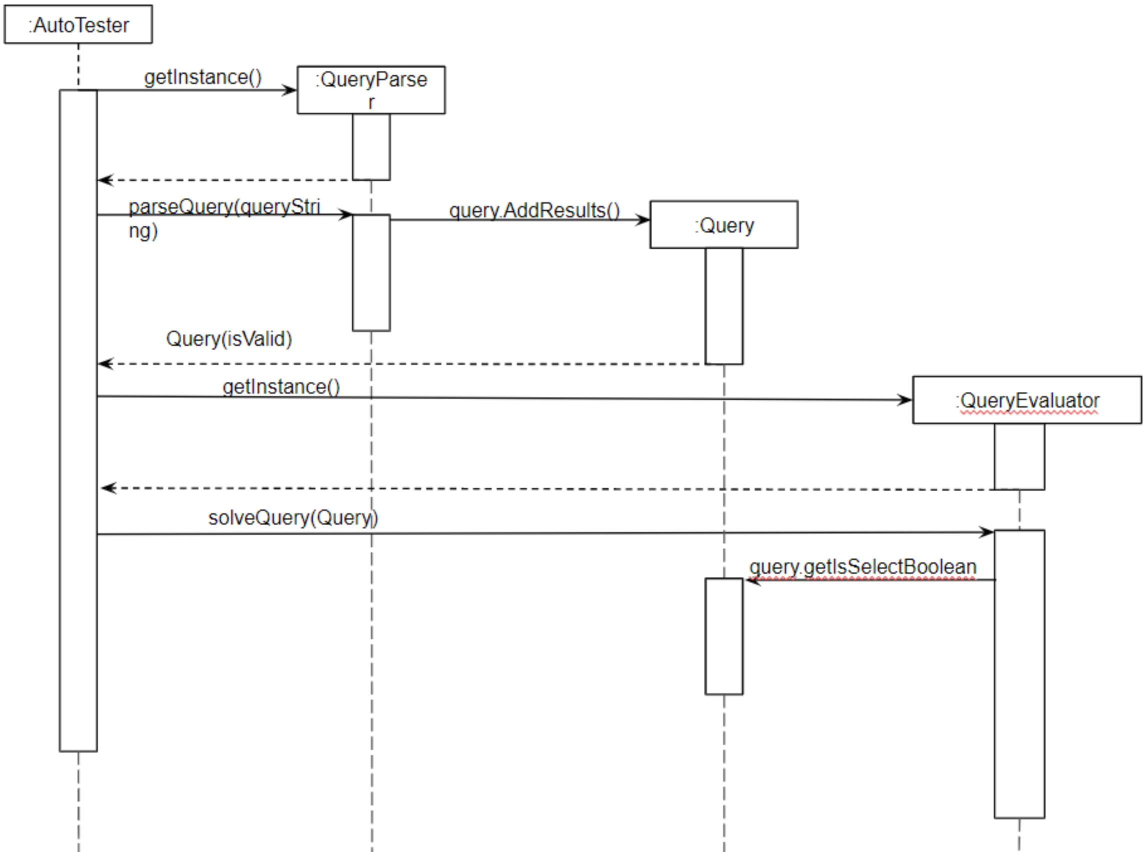


Figure 6.2.3.1 PQL Preprocessor - PQL interaction

The below example depicts an example situation where the Query Preprocessor and the Query Evaluator evaluates the following query:

Integration Testing (Testing Valid BOOLEAN Query)

Test Purpose: Checks if the Query Preprocessor processes the query correctly and that the query is syntactically and semantically correct and returns TRUE and if the Query Evaluator returns the correct answer.

Required Test Inputs: `stmt s; Select BOOLEAN such that Follows(s, s)`

Expected Test Results: Query is valid, `IsSelectBoolean` flag is set to TRUE, and the expected values match to the actual values, the answer returns FALSE.

```
Assert::IsTrue(query.getIsValid() &&
query.getClauses().at(0).designAbstractionType                          ==
Query::DesignAbstractionType::FOLLOWS && query.getClauses().at(0).leftRef.refString
== "s" && query.getClauses().at(0).rightRef.refString == "s" &&
query.getIsSelectBoolean());
```

It asserts that the query is valid and to ensure the expected values are equal to the actual values found in the clauses, and that the select boolean flag is set to TRUE on the PQL preprocessor side.

```
it = std::find(answer.begin(), answer.end(), "TRUE");
Assert::IsFalse(it != answer.end());

it = std::find(answer.begin(), answer.end(), "FALSE");
Assert::IsTrue(it != answer.end());
```

It asserts that the expected answer is equal to the actual answer in the PQL Evaluator side.

6.3 System Testing

For system testing, we are testing it more rigorously by creating system tests for each particular item such as clauses and relationships. For instance, a system test is created for `pattern` and `with` clauses, and on top of that, we would have system test cases to test for `Next` and `Calls`.

On top of that, we would also have Easy, Medium and Hard test cases for iteration 3 that covers all aspects of iteration 3, with each difficulty having an increasing level of complexity in the SIMPLE program such as having more nesting and having more complicated expressions and queries.

For the purposes of this report, the two test case examples would be:

1) Testing for a specific clause/relationship
2) Medium test case

6.3.1 Testing for Specific Clause/Relationship

In such test cases, we would test for all possible inputs and combinations to ensure that our program has accounted for such inputs and that that particular component works as intended. Such individual tests are mostly to test if our program is able to handle the right types of synonyms and to check for the correctness for that particular clause/relationship.

It would also contain some invalid cases whenever possible to make sure that it handles such cases too. For testing specific clauses or relationships, we would generate SIMPLE source programs that are able to return non "none" answers to allow us to check for correctness of the query evaluation.

In this section, the `Affects` and `Affects*` system test would be used as an example. The SIMPLE program for this system test is attached in **Appendix A**. They were being combined together in a file as it was easier to test both types of queries using one source program. This is just a particular example for such system tests. It would contain other queries with different orderings of synonyms/variables and different synonym types to ensure that the program works as intended with such inputs.

System Test (Testing Specific Clause/Relationship)

Test Purpose: Test if the program is able to identify integers as statement numbers and `prog_line` synonym as one of its inputs. It would also ensure that the program can correctly identify the statements that statement number 16 affects directly or indirectly.

Required Test Inputs:
The first input required would be the program given in **Appendix A** as a text file. The next input required would be the list of queries required as a text file. For this case, we would be focusing on the query specified below:

```
41 - Affects*, First stmt#, second synonym (Valid)
prog_line n;
Select n such that Affects*(16, n)
39, 50, 51
5000
```

Expected Test Results: `39, 50, 51`

It has to match the output given in the query.

6.3.2 Medium Test Case

In such test cases, we would be testing the system as a whole, as if we were submitting our program to the teaching team for evaluation. This test case also makes sure that it allows for more advanced query types such as being able to handle multiple clauses, on top of being able to deal with individual clauses too.

For multiple clauses, we would try to ensure that we have different permutations of the clauses to ensure that the ordering does not affect the outcome of the query evaluation. For instance, we would replace `such that` with `and`, while including pattern in different clauses to ensure that it works. An example of such is as follows:

Original query	Different ordering

`Select s1 such that Modifies(p, v) and Calls(p, q) and Next*(s1, a) with p.procName = "K" with "G" = q.procName`	`Select s1 such that Calls(p, q) and Modifies(p, v) and Next*(s1, a) with p.procName = "K" with "G" = q.procName`
	`Select s1 such that Next*(s1, a) with p.procName = "K" with "G" = q.procName such that Calls(p, q) and Modifies(p, v)`

The reason for doing so is to ensure that our program is able to identify that though there are permutations to it, the results are the same. Our team has several queries with different permutations but it has been left out in the report for brevity.

Additionally, we have `with` clauses like “`with a.stmt# = 12`” and we would flip the order of the left-hand side and right-hand side to get “`with 12 = a.stmt#`”. This is to be more thorough with our testing, ensuring that the program accepts such inputs too.

For this example, we would be using the Medium test case and the SIMPLE program can be found in Appendix B. This is just a particular example for such system tests. It would contain other queries with different orderings of synonyms/variables and different synonym types to ensure that the program works as intended with such inputs.

System Test (Medium Test Case)

Test Purpose: To select the variable that is being used in one procedure and modified in another procedure. It also tests if the system is able to handle the use of synonyms in the queries and whether it can retrieve out the right variables and procedures to be used for such queries.

Required Test Inputs:
The test input required would be the SIMPLE program in **Appendix B** and the list of queries associated with that test case. The query input has to be in the Autotester format as such:

```
49 - Multiple clauses (Uses + Modifies, common variable, procedure)
procedure p1, p2; variable v;
Select v such that Uses(p1, v) such that Modifies(p2, v)
reddit
5000
```

Expected Test Results: `reddit`

It has to match the output given in the query.

6.4 Stress Testing

Stress testing is done to test the limits of the system and to ensure that it is able to handle complex queries and SIMPLE programs. It allows us to identify if the optimization strategy that we implemented (grouping and scoring) improves the time required to answer complex queries. With these stress testing queries, we can ensure that optimization works when such complex queries are being evaluated within 5 seconds and have improved the timings compared to with optimization turned off.

The SIMPLE program created for stress testing consists of:

- 21 procedures of varying lengths and nesting
- Maximum level of nesting: 9
- Procedures can call other procedures
- Procedures have up to 19 variables

An example of the stress testing queries in Autotester format are as follows:

Stress Test (Example query)

Test Purpose: To demonstrate that there is early termination used for evaluating queries

Required Test Inputs:
The test input required would be the stress test SIMPLE program and the list of queries associated with that test case. The query input has to be in the Autotester format as such:

```
1 - Test early termination
assign a, a1, a2; stmt s, s1, s2, s3; if ifs; while w; prog_line n; procedure p, q; variable v;
Select <a, s> such that Next*(a1, s1) and Calls*(p, q) and Parent*(w, ifs) and Affects(s2, s3) and Uses(a2, v) and Follows*(_, s) pattern a1(v, _) such that Next(367, 368)
none
5000
```

Expected Test Results: none

It has to match the output given in the query.

Stress Test (Example query)

Test Purpose: To demonstrate that there is grouping of clauses done to optimize evaluation of query.

Required Test Inputs:
The test input required would be the stress test SIMPLE program and the list of queries associated with that test case. The query input has to be in the Autotester format as such:

```
3 - Grouping of clauses (Valid)
assign a1, a2; stmt s1; while w; if ifs; prog_line n; read r;
```

```
Select <a1, a2> such that Next(w, a1) and Next*(a1, a2) and Affects(a1, a2) and
Next*(a1, ifs) and Affects*(a1, a2) and Follows*(ifs, s1) such that Parent*(w, s1)
and Next*(n, r)
(Answer omitted for brevity)
5000
```

Expected Test Results: Answer omitted for brevity

It has to match the output given in the query.

7. Discussion

In iteration 3, our team had to rush for the most part as we struggled to come up with additional test cases for this iteration. As Xiu Qi was initially the sole person working on testing, we have engaged other team members such as Mei Yen and Jia Hao who have been freed up, to assist with the system testing. Creating the source program and queries for the stress testing was also very challenging as we had to ensure that we account for edge cases and having multiple start and end points for our program, while ensuring that our queries are robust enough to test for certain hidden cases.

The front-end also had a hard time doing up the extensions and especially AffectsBIP*, as it was really challenging and they had to juggle other modules at the same time. In the end, the front-end team managed to pull through and implement all the extensions and the full SPA requirements.

We retained the “buddy system” implemented in iteration 2 to update each other on our progress, and would sound off early should we face any difficulties with our own components. We have found this system to be rather useful in keeping track of each other’s progress on top of constantly updating on our progress through Telegram.

As compared to iterations 1 and 2, our team has definitely improved on communication which has helped us to identify problems earlier so that we can rectify them. The team would constantly update each other on our progress and on any issues and this has helped to clear up any misunderstandings or clear up any doubts.

All in all, we felt that the team has improved on its teamwork and communication over the course of the module and that this can be translated over to future software development projects or even in our future jobs.

8. Abstract API

8.1 Variable, Procedure

As Variable and Procedure tables use the same class, their abstract API is the same. As such, the word “Variable” can be swapped for “Procedure” for the following API.

Variable Table
The Variable Table stores all Variables of the program.

VARIABLE_INDEX insertVariable(**VARIABLE**);

Description:
If the VARIABLE is not already in the table, inserts VARIABLE and returns the VARIABLE_INDEX of the entry.
Else, returns the VARIABLE_INDEX of the VARIABLE.

VARIABLE getVariable(**VARIABLE_INDEX**);

Description:
If VARIABLE_INDEX exists as a key in the Variable table, returns the VARIABLE associated with it.
Else, returns an empty result.

VARIABLE_INDEX getKeyVariable(**VARIABLE**);

Description:
If the given VARIABLE exists as a value in the Variable table, returns its VARIABLE_INDEX key.
Else, return -1.

SIZE_OF_TABLE getSizeVariable();

Description:
Returns the number of entries in the Variable table.

BOOLEAN hasKeyVariable(**VARIABLE_INDEX**);

Description:
Returns TRUE **if** the VARIABLE_INDEX exists in the Variable table.
Else, returns FALSE.

LIST_OF_KEYS getAllKeysVariable();

Description:
Returns a List of all keys in the Variable table.

LIST_OF_VALUES getAllValuesVariable();

Description:
Returns a List of all values in the Variable table.

	VOID clearAll(); **Description:** Clears the table.

8.2 Constant

Constant Class The Constant Class stores all Constants of the program.	
	VOID insertConstant(**LINE**, **SET_OF_CONSTANTS**); **Description:** **If** the LINE is not already in the Constant table, inserts LINE:SET_OF_CONSTANTS. **Else**, appends to existing SET_OF_CONSTANTS.
	SET_OF_CONSTANTS getConstant(**LINE**); **Description:** **If** LINE exists as a key in the Constant table, returns the SET_OF_CONSTANTS. **Else**, returns an empty result.
	BOOLEAN isConstant(**LINE**, **SET_OF_CONSTANTS**); **Description:** **If** LINE is an existing key with value SET_OF_CONSTANTS, returns TRUE. **Else**, returns FALSE.
	BOOLEAN isConstant(**LINE**, **CONSTANT**); **Description:** **If** LINE is an existing key, and CONSTANT is in the value, returns TRUE. **Else**, returns FALSE.
	SIZE_OF_TABLE getSizeConstant(); **Description:** Returns the number of entries in the Constant table.
	BOOLEAN hasKeyConstant(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the Constant table. **Else**, returns FALSE.
	LIST_OF_KEYS getAllKeysConstant();

	Description: Returns a List of all keys in the Constant table.
	LIST_OF_VALUES getAllValuesConstant(); **Description:** Returns a List of all values in the Constant table.
	VOID clearAll(); **Description:** Clears both tables.

8.3 Statement, Read, Call, Print

As Statement, Read, Call and Print tables use the same class, their abstract API is the same. As such, the word “Statement” can be swapped for “Read”, “Call” or “Print” for the following API.

Statement Table The Statement Table stores all Statements of the program.	
	VOID insertStatement(**LINE**, **STATEMENT_TYPE**); **Description:** **If** the LINE is not already in the Statement table, inserts LINE:STATEMENT_TYPE.
	STATEMENT_TYPE getStatement(**LINE**); **Description:** **If** LINE exists as a key in the Statement table, returns the STATEMENT_TYPE associated with it. **Else**, returns an empty result.
	BOOLEAN isStatement(**LINE, STATEMENT_TYPE**); **Description:** Returns True **if** LINE is an existing key with value STATEMENT_TYPE. **Else**, returns FALSE.
	LIST_OF_KEYS getKeysStatement(**STATEMENT_TYPE**); **Description:** Returns a list of all LINES that has value STATEMENT_TYPE, **if** STATEMENT_TYPE exists. **Else**, returns an empty list.
	SIZE_OF_TABLE getSizeStatement();

	Description: Returns the number of entries in the Statement table.
	BOOLEAN hasKeyStatement(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the Statement table. **Else**, returns FALSE.
	LIST_OF_KEYS getAllKeysStatement(); **Description:** Returns a List of all keys in the Statement table.
	LIST_OF_VALUES getAllValuesStatement(); **Description:** Returns a List of all values in the Statement table.
	VOID clearAll(); **Description:** Clears the table.

8.4 StatementList, ProcedureStatementList

As StatementList and ProcedureStatementList use the same class, their abstract API is the same. As such, the word “StatementList” can be swapped for “ProcedureStatementList” for the following API.

StatementList Class The StatementList Class stores all StatementLists of the program.	
	VOID insertStatementList(**LINE**, **SET_OF_STATEMENT_LINE_NUMBERS**); **Description:** **If** the LINE is not already in the StatementList table, inserts LINE:SET_OF_STATEMENT_LINE_NUMBERS. **Else**, appends to existing SET_OF_STATEMENT_LINE_NUMBERS.
	SET_OF_STATEMENT_LINE_NUMBERS getStatementList(**LINE**); **Description:** **If** LINE exists as a key in the StatementList table, returns the SET_OF_STATEMENT_LINE_NUMBERS. **Else**, returns an empty result.
	LINE getStatementListKey(**STATEMENT_LINE_NUMBER**);

	Description: **If** the given STATEMENT_LINE_NUMBER exists as a value in the StatementList table, returns its LINE key. **Else**, return -1.
	SIZE_OF_TABLE getSizeStatementList(); **Description:** Returns the number of entries in the StatementList table.
	BOOLEAN hasKeyStatementList(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the StatementList table. **Else**, returns FALSE.
	LIST_OF_KEYS getAllKeysStatementList(); **Description:** Returns a List of all keys in the StatementList table.
	LIST_OF_VALUES getAllValuesStatementList(); **Description:** Returns a List of all values in the StatementList table.
	VOID clearAll(); **Description:** Clears both tables.

8.4 Follows

Follows Class The Follows Class stores all Follows and FollowedBy relationships of the program.	
	VOID insertFollows(**LINE**, **LEADER_LINE**); **Description:** **If** the LINE is not already in the Follows table, inserts LINE:LEADER_LINE. **Else**, returns an error. Also calls ***insertFollowedBy(LEADER_LINE, LINE).***
	LEADER_LINE getFollows(**LINE**);

	Description: **If** LINE exists as a key in the Follows table, returns the LEADER_LINE associated with it. **Else**, return -1.
	BOOLEAN isFollows(**LINE, LEADER_LINE**); **Description:** **If** LINE is an existing key with value LEADER_LINE in the Follows table, returns TRUE. **Else**, returns FALSE.
	SIZE_OF_TABLE getSizeFollows(); **Description:** Returns the number of entries in the Follows table.
	BOOLEAN hasKeyFollows(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the Follows table. **Else**, returns FALSE.
	LIST_OF_KEYS getAllKeysFollows(); **Description:** Returns a List of all keys in the Follows table.
	LIST_OF_VALUES getAllValuesFollows(); **Description:** Returns a List of all values in the Follows table.
	VOID insertFollowedBy(**LINE**, **FOLLOWER_LINE**); **Description:** Inserts LINE:FOLLOWER_LINE **if** the LINE is not already in the FollowedBy table. **Else**, returns error. Only called by ***insertFollows()*** method and is a private method.
	FOLLOWER_LINE getFollowedBy(**LINE**); **Description:** **If** LINE exists as a key in the FollowedBy table, returns the FOLLOWER_LINE associated with it. **Else**, returns -1.
	BOOLEAN isFollowedBy(**LINE**, **FOLLOWER_LINE**); **Description:** **If** LINE is an existing key with value FOLLOWER_LINE in the FollowedBy table, returns TRUE. **Else**, returns FALSE.

	SIZE_OF_TABLE getSizeFollowedBy(); **Description:** Returns the number of entries in the FollowedBy table.
	BOOLEAN hasKeyFollowedBy(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the FollowedBy table. **Else**, returns FALSE.
	LIST_OF_KEYS getAllKeysFollowedBy(); **Description:** Returns a List of all keys in the FollowedBy table.
	LIST_OF_VALUES getAllValuesFollowedBy(); **Description:** Returns a List of all values in the FollowedBy table.
	VOID clearAll(); **Description:** Clears both tables.

8.5 Parent

Parent The Parent stores both the Parent and Child relationships of the program.	
	VOID insertParent(**LINE**, **PARENT_LINE**); **Description:** **If** the LINE is not already in the Parent table, inserts LINE:PARENT_LINE. **Else**, returns error. Also calls ***insertChild(PARENT_LINE, LINE)***.
	PARENT_LINE getParent(**LINE**)**;** **Description:** **If** LINE exists as a key in the Parent table, returns the PARENT_LINE associated with it. **Else**, return -1.
	BOOLEAN isParent(**LINE**, **PARENT_LINE**);

	Description: **If** LINE is an existing key with value PARENT_LINE, returns True. **Else**, returns FALSE.
	SIZE_OF_TABLE getSizeParent(); **Description:** Returns the number of entries in the Parent table.
	BOOLEAN hasKeyParent(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the Parent table. **Else**, returns FALSE.
	LIST_OF_KEYS getAllKeysParent(); **Description:** Returns a List of all keys in the Parent table.
	LIST_OF_VALUES getAllValuesParent(); **Description:** Returns a List of all values in the Parent table.
	VOID insertChild(**LINE**, **CHILD_LINE)**; **Description:** **If** the LINE is not already in the Child table, inserts LINE:SET_OF_CHILD_LINES. **Else**, appends to existing SET_OF_CHILD_LINES. Only called by ***insert*`Parent`*()*** method and is a private method.
	SET_OF_CHILD_LINES getChild(**LINE**); **Description:** **If** LINE exists as a key in the Child table, returns the SET_OF_CHILD_LINES associated with it. **Else**, returns -1.
	BOOLEAN isChild(**LINE**, **SET_OF_CHILD_LINES**); **Description:** **If** LINE is an existing key with value SET_OF_CHILD_LINES, returns TRUE. **Else**, returns FALSE.
	BOOLEAN isChild(**LINE**, **CHILD_LINE**); **Description:** **If** LINE is an existing key and CHILD_LINE is in the value, returns TRUE. **Else**, returns FALSE.

	SIZE_OF_TABLE getSizeChild(); **Description:** Returns the number of entries in the Child table.
	BOOLEAN hasKeyChild(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the Child table. **Else**, returns FALSE.
	LIST_OF_KEYS getAllKeysChild(); **Description:** Returns a List of all keys in the Child table.
	LIST_OF_VALUES getAllValuesChild(); **Description:** Returns a List of all values in the Child table.
	VOID clearAll(); **Description:** Clears both tables.

8.6 Follows*, Parent*, Calls, Calls*

As FollowsStar, ParentStar, Calls, CallsStar tables use the same class, their abstract API is the same. As such, the word “FollowsStar” can be swapped for “ParentStar”, “Calls” or “CallsStar” for the following API.

FollowsStar Class The FollowsStar Class stores all FollowsStar and FollowedByStar relationships of the program.	
	VOID insertFollowsStar(**LINE**, **SET_OF_LEADER_LINES**); **Description:** **If** the LINE is not already in the FollowsStar table, inserts LINE:SET_OF_LEADER_LINES. **Else**, appends to existing SET_OF_LEADER_LINES. Also for each LEADER_LINE in SET_OF_LEADER_LINES, calls ***insertFollowedByStar(LEADER_LINE, LINE).***
	SET_OF_LEADER_LINES getFollowsStar(**LINE**); **Description:** **If** LINE exists as a key in the FollowsStar table, returns the SET_OF_LEADER_LINES. **Else**, returns an empty result.

	BOOLEAN isFollowsStar(**LINE**, **SET_OF_LEADER_LINES**); **Description:** **If** LINE is an existing key with value SET_OF_LEADER_LINES, returns TRUE. **Else**, returns FALSE.
	BOOLEAN isFollowsStar(**LINE**, **LEADER_LINE**); **Description:** **If** LINE is an existing key, and LEADER_LINE is in the value, returns TRUE. **Else**, returns FALSE.
	SIZE_OF_TABLE getSizeFollowsStar(); **Description:** Returns the number of entries in the FollowsStar table.
	AREA_OF_TABLE getAreaFollowsStar(); **Description:** Returns the sum of sizes of each value in the FollowsStar table.
	BOOLEAN hasKeyFollowsStar(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the FollowsStar table. **Else**, returns FALSE.
	LIST_OF_KEYS getAllKeysFollowsStar(); **Description:** Returns a List of all keys in the FollowsStar table.
	LIST_OF_VALUES getAllValuesFollowsStar(); **Description:** Returns a List of all values in the FollowsStar table.
	VOID insertFollowedByStar(**LINE**, **SET_OF_FOLLOWER_LINES**); **Description:** **If** the LINE is not already in the FollowedByStar table, inserts LINE:SET_OF_FOLLOWER_LINES. **Else**, appends to existing SET_OF_FOLLOWER_LINES. Only called by ***insertFollowsStar()*** method and is a private method.
	SET_OF_FOLLOWER_LINES getFollowedByStar(**LINE**); **Description:** **If** LINE exists as a key in the FollowedByStar table, returns the SET_OF_FOLLOWER_LINES.

	Else, returns an empty result.
	BOOLEAN isFollowedByStar(**LINE**, **SET_OF_FOLLOWER_LINES**); **Description:** **If** LINE is an existing key with value SET_OF_FOLLOWER_LINES, returns True. **Else**, returns FALSE.
	BOOLEAN isFollowedByStar(**LINE**, **FOLLOWER_LINE**); **Description:** **If** LINE is an existing key and FOLLOWER_LINE is in the value, returns True. **Else**, returns FALSE.
	SIZE_OF_TABLE getSizeFollowedByStar(); **Description:** Returns the number of entries in the FollowedByStar table.
	AREA_OF_TABLE getAreaFollowedByStar(); **Description:** Returns the sum of sizes of each value in the FollowedByStar table.
	BOOLEAN hasKeyFollowedByStar(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the FollowedByStar table. **Else**, returns FALSE.
	LIST_OF_KEYS getAllKeysFollowedByStar(); **Description:** Returns a List of all keys in the FollowedByStar table.
	LIST_OF_VALUES getAllValuesFollowedByStar(); **Description:** Returns a List of all values in the FollowedByStar table.
	VOID clearAll(); **Description:** Clears both tables.

8.7 Next, Next*, NextBip, NextBip* Affects, Affects*, AffectsBip

As Next, NextStar, NextBip, NextBipStar, Affects, AffectsStar and AffectsBip tables use the same class, their abstract API is the same. As such, the word “Statement” can be swapped for “Next”, “NextStar”, “NextBip”, “NextBipStar”, “Affects”, “AffectsStar“ or “AffectsBip” for the following API.

Next Class The Next Class stores all Next and Previous relationships of the program.	
	VOID insertNext(**LINE**, **SET_OF_NEXT_LINES**); **Description:** **If** the LINE is not already in the Next table, inserts LINE:SET_OF_NEXT_LINES. **Else**, appends to existing SET_OF_NEXT_LINES. Also for each NEXT_LINE in SET_OF_NEXT_LINES, calls ***insertPrevious(NEXT_LINE, LINE).***
	SET_OF_NEXT_LINES getNext(**LINE**); **Description:** **If** LINE exists as a key in the Next table, returns the SET_OF_NEXT_LINES. **Else**, returns an empty result.
	BOOLEAN isNext(**LINE**, **SET_OF_NEXT_LINES**); **Description:** **If** LINE is an existing key with value SET_OF_NEXT_LINES, returns TRUE. **Else**, returns FALSE.
	BOOLEAN isNext(**LINE**, **NEXT_LINE**); **Description:** **If** LINE is an existing key, and NEXT_LINE is in the value, returns TRUE. **Else**, returns FALSE.
	SIZE_OF_TABLE getSizeNext(); **Description:** Returns the number of entries in the Next table.
	AREA_OF_TABLE getAreaNext(); **Description:** Returns the sum of sizes of each value in the Next table.
	BOOLEAN hasKeyNext(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the Next table. **Else**, returns FALSE.

	LIST_OF_KEYS getAllKeysNext(); **Description:** Returns a List of all keys in the Next table.
	LIST_OF_VALUES getAllValuesNext(); **Description:** Returns a List of all values in the Next table.
	VOID insertPrevious(**LINE**, **SET_OF_PREVIOUS_LINES**); **Description:** **If** the LINE is not already in the Previous table, inserts LINE:SET_OF_PREVIOUS_LINES. **Else**, appends to existing SET_OF_PREVIOUS_LINES. Only called by ***insertNext()*** method and is a private method.
	SET_OF_PREVIOUS_LINES getPrevious(**LINE**); **Description:** **If** LINE exists as a key in the Previous table, returns the SET_OF_PREVIOUS_LINES. **Else**, returns an empty result.
	BOOLEAN isPrevious(**LINE**, **SET_OF_PREVIOUS_LINES**); **Description:** **If** LINE is an existing key with value SET_OF_PREVIOUS_LINES, returns True. **Else**, returns FALSE.
	BOOLEAN isPrevious(**LINE**, **PREVIOUS_LINE**); **Description:** **If** LINE is an existing key and PREVIOUS_LINE is in the value, returns True. **Else**, returns FALSE.
	SIZE_OF_TABLE getSizePrevious(); **Description:** Returns the number of entries in the Previous table.
	AREA_OF_TABLE getAreaPrevious(); **Description:** Returns the sum of sizes of each value in the Previous table.
	BOOLEAN hasKeyPrevious(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the Previous table.

	Else, returns FALSE.
	LIST_OF_KEYS getAllKeysPrevious(); **Description:** Returns a List of all keys in the Previous table.
	LIST_OF_VALUES getAllValuesPrevious(); **Description:** Returns a List of all values in the Previous table.
	VOID clearAll(); **Description:** Clears both tables.

8.8 Uses, Modifies, ProcedureUses, ProcedureModifies

As Uses, Modifies, ProcedureUses, ProcedureModifies tables use the same class, their abstract API is the same. As such, the word “Uses” can be swapped for “Modifies”, “ProcedureUses” or “ProcedureModifies” for the following API.

Uses class The Uses class stores all Uses and UsedBy relationships of the program.	
	VOID insertUses(**LINE**, **SET_OF_USED_VARIABLES**); **Description:** **If** the LINE is not already in the Uses table, inserts LINE:SET_OF_USED_VARIABLES. **Else,** appends to existing SET_OF_USED_VARIABLES. Also for each USED_VARIABLE in SET_OF_USED_VARIABLES, calls ***insertUsedBy(USED_VARIABLE, LINE)***.
	SET_OF_USED_VARIABLES getUses(**LINE**); **Description:** **If** LINE exists as a key in the Uses table, returns the SET_OF_USED_VARIABLES associated with it. **Else**, returns an empty result.
	BOOLEAN isUses(**LINE**, **SET_OF_USED_VARIABLES**); **Description:** **If** LINE is an existing key with value SET_OF_USED_VARIABLES, returns TRUE. **Else,** returns FALSE.

	BOOLEAN isUses(**LINE**, **USED_VARIABLE**); **Description:** **If** LINE is an existing key, and USED_VARIABLE is in the value, returns TRUE. **Else,** returns FALSE.
	SIZE_OF_TABLE getSizeUses(); **Description:** Returns the number of entries in the Uses table.
	AREA_OF_TABLE getAreaUses(); **Description:** Returns the sum of sizes of each value in the Uses table.
	BOOLEAN hasKeyUses(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the Uses table. **Else**, returns FALSE.
	LIST_OF_KEYS getAllKeysUses(); **Description:** Returns a List of all keys in the Uses table.
	LIST_OF_VALUES getAllValuesUses(); **Description:** Returns a List of all values in the Uses table.
	VOID InsertUsedBy(**VARIABLE**, **SET_OF_USEDBY_LINES);** **Description:** **If** the VARIABLE is not already in the UsedBy table, inserts VARIABLE:SET_OF_USEDBY_LINES. **Else**, appends to existing SET_OF_USEDBY_LINES. Only called by ***insertUses()*** method and is a private method
	SET_OF_USEDBY_LINES getUsedBy(**VARIABLE**); **Description:** **If** VARIABLE exists as a key in the UsedBy table, returns the list of SET_OF_USEDBY_LINES associated with it. **Else**, returns an empty result.
	BOOLEAN isUsedBy(**VARIABLE**, **SET_OF_USEDBY_LINES**);

	Description: **If** VARIABLE is an existing key with value SET_OF_USEDBY_LINES returns TRUE. **Else**, returns FALSE.
	BOOLEAN isUsedBy(**VARIABLE**, **USEDBY_LINE**); **Description:** **If** VARIABLE is an existing key, and USEDBY_LINE is in the value, returns TRUE. **Else,** returns FALSE.
	SIZE_OF_TABLE getSizeUsedBy(); **Description:** Returns the number of entries in the UsedBy table.
	AREA_OF_TABLE getAreaUsedBy(); **Description:** Returns the sum of sizes of each value in the UsedBy table.
	BOOLEAN hasKeyUsedBy(**USEDBY_LINE**); **Description:** Returns TRUE **if** the USEDBY_LINE exists in the UsedBy table. **Else**, returns FALSE.
	LIST_OF_KEYS getAllKeysUsedBy(); **Description:** Returns a List of all keys in the UsedBy table.
	LIST_OF_VALUES getAllValuesUsedBy(); **Description:** Returns a List of all values in the UsedBy table.
	VOID clearAll(); **Description:** Clears both tables.

8.9 While, If

As While and If use the same class, their abstract API is the same. As such, the word “While” can be swapped for “If” for the following API.

While class The While class stores all While conditions of the program.	
	VOID insertWhile(**LINE**, **SET_OF_CONDITIONS**); **Description:** **If** the LINE is not already in the While table, inserts LINE:SET_OF_CONDITIONS. **Else,** appends to existing SET_OF_CONDITIONS.
	SET_OF_CONDITIONS getWhile(**LINE**); **Description:** **If** LINE exists as a key in the While table, returns the SET_OF_CONDITIONS associated with it. **Else**, returns an empty result.
	BOOLEAN isWhile(**LINE**, **SET_OF_CONDITIONS**); **Description:** **If** LINE is an existing key with value SET_OF_CONDITIONS, returns TRUE. **Else,** returns FALSE.
	BOOLEAN isWhile(**LINE, CONDITION**); **Description:** **If** LINE is an existing key, and CONDITION is in the value SET_OF_CONDITIONS, returns TRUE. **Else,** returns FALSE.
	SIZE_OF_TABLE getSizeWhile(); **Description:** Returns the number of entries in the While table.
	AREA_OF_TABLE getAreaWhile(); **Description:** Returns the sum of sizes of each value in the While table.
	BOOLEAN hasKeyWhile(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the While table. **Else**, returns FALSE.
	LIST_OF_KEYS getAllKeysWhile();

	Description: Returns a List of all keys in the While table.
	LIST_OF_VALUES getAllValuesWhile(); **Description:** Returns a List of all values in the While table.
	VOID clearAll(); **Description:** Clears both tables.

8.10 Assign

Assign Table The Assign Table stores all Assign statements of the program.	
	VOID insertAssignLHS(**LINE**, **VARIABLE_LHS**); **Description:** **If** the LINE is not already in the Assign table, inserts VARIABLE_LHS at LINE.
	VOID insertAssignRHS(**LINE**, **SHUNTED_RHS**); **Description:** **If** the LINE is not already in the Assign table, inserts SHUNTED_RHS at LINE.
	VARIABLE_LHS and SHUNTED_RHS getAssign(**LINE**); **Description:** **If** exists as a key in the Assign table, returns the VARIABLE_LHS and SHUNTED_RHS associated with it. **Else**, returns an empty result.
	SIZE_OF_TABLE getSizeAssign(); **Description:** Returns the number of entries in the Assign table.
	BOOLEAN hasKeyAssign(**LINE**); **Description:** Returns TRUE **if** the LINE exists in the Assign table. **Else**, returns FALSE.

	LIST_OF_KEYS getAllKeysAssign(); **Description:** Returns a List of all keys in the Assign table.
	LIST_OF_VALUES getAllValuesAssign(); **Description:** Returns a List of all values in the Assign table.
	VOID clearAll(); **Description:** Clears the table.

Appendix A - Affects/Affects* Test 2 SIMPLE program

```
      procedure one {
 1          read a;
 2          read b;
 3          read c;
 4          read d;
 5          read e;
 6          read f;
 7          read g;
      }

      procedure two {
 8          a = 1;
 9          b = a;
10          c = b * 3 + 2 * a;
11          d = c / 2 * 4 + a;
12          e = g/c;
13          a = c * 3;
14          f = f/f;
15          g = a * b + c - d / e * f;
16          h = 20 * 5;

17          if (e == f) then {
18                print e;
19                f = e;
20                call one;
21                print e;
            } else {
22                c = b;
23                print b;
24                read b;
25                b = b + c + e;
26                print e;
27                print b;
            }

28          while (c <= c) {
29                d = d + e;
30                g = b + a;

31                while (g >= b) {
32                      a = b;
33                      b = d;
34                      while (a >= 20) {
35                            e = h;
36                            b = c;
37                            call one;
                        }
38                      g = a + b + c + e;
                  }
39      }
40      a = d + e / f * g - h;
41      if (a < 2) then {
            call three;
42      } else {
43          b = g;
44          while (e > 100) {
45              if (g < 10) then {
46                  while (h < 69) {
47                      print g;
                        print b;
48                  }
                    read b;
49              } else {
                    print b;
                }
            }
        }
50
51      c = a + b - c / d % e - f * g + h;
        h = h;
    }

52  procedure three {
53      a = a;
54      b = b;
55      c = c;
56      d = d;
57      e = e;
58      f = f;
        g = g;
    }
```

Appendix B - Medium System Test SIMPLE Program

```
    procedure K {
1       Z = sE + Z + sE % 9;
2       read  Z;
3       while ( ( ( yU != Z) && ( 32 != 32)) && ( ( yU > 9) && ( 32 != 25))) {
4           G = G * G;
5           print  yU;
6           while ( (!( 9 == 25 + 59)) || (!( Z + G != 91 - G))) {
7               if (!( ( Bf <= 91) && ( 32 >= 32))) then {
8                   while (!( 72 <= yU)) {
9                       call  G;
                    }
```

```
                } else {
10                  sE = 25 + 59 + yU * 72;
                }
11              print  Bf;
            }
12          print  Z;
13          call  G;
        }
14      if ( Bf - 25 <= 9 + 59) then {
15          read  sE;
16          print  Bf;
        } else {
17          if ( 72 <= 9 - 72) then {
18              G = 9 * yU;
            } else {
19              call  C;
            }
        }
    }

    procedure G {
20      if (!( ( Bf != 32) || ( yU >= 32))) then {
21          yU = 67 * sE;
22          print  G;
23          if ( ( ( 9 < G - 9) || ( 9 > 7)) || (!( Bf < 25))) then {
24              read  G;
25              sE = 9 % yU;
26              print  sE;
27              if ( ( 9 != 9) && ( ( 25 == 32) || ( 32 > 25 + 59))) then {
28                  call  C;
                } else {
29                  while ( 9 != Bf) {
30                      print  Bf;
31                      G = 91 + 9 - 32 * G;
                    }
32                  call  C;
                }
33              if ( ( ( G != 32) && ( 67 > Z)) || ( 25 >= 25 + 67)) then {
34                  print  sE;
                } else {
35                  G = 25 % 9;
                }
36              Bf = 59 * Z;
            } else {
37              G = 32 + 72 - 59 * sE;
            }
        } else {
38          read  G;
        }
    }

    procedure C {
39      if ( Bf - G >= Bf) then {
40          if ( ( 59 <= 59) && (!( 7 - 9 >= sE))) then {
```

```
            read  Z;
            while (!(!( 59 == 9 - yU))) {
                Z = sE + 91 - 9;
                G = Bf - 7;
            }
            Z = Bf + yU * Bf;
        } else {
            print  yU;
            while (!( ( 25 - 9 <= 59) || ( G != 25 + 91))) {
                while (!(!( 67 >= G - G))) {
                    if (!( ( Bf == 59) && ( Bf + 25 == 91))) then {
                        if (!( ( 67 != sE) || ( 67 + 72 <= yU))) then {
                            while ( (!( G > G)) && ( 72 >= yU + 59)) {
                                print  Z;
                                read  Bf;
                            }
                            Bf = 72 * G;
                        } else {
                            G = 67 - 9 - 25 * yU;
                        }
                        G = 9 * yU;
                    } else {
                        if (!(!( 32 <= yU - yU))) then {
                            G = 9 * yU;
                        } else {
                            while ( 7 + 25 > 9) {
                                if ( G >= G - Bf) then {
                                    while ( 32 < 91) {
                                        sE = 67 + Bf + yU;
                                    }
                                } else {
                                    Bf = G;
                                }
                            }
                        }
                    }
                }
            }
        }
    } else {
        Bf = 67 + 59 % 9;
    }
}
```