Full (Basic + Advanced) SPA requirements

1. Motivation

Some companies spend as much as 80% of software budgets on software maintenance. During software maintenance, programmers spend almost 50% of the time trying to understand a program. Therefore, methods and tools that can ease program understanding have a potential to substantially cut development costs.

During program maintenance, programmers often try to locate code relevant to the maintenance task in hand. Here are examples of questions programmers might ask to locate code of interest:

- I need to find code that implements salary computation rules!
- Where is variable ‘x’ modified? Where is it used?
- I need to find all statements with sub-expression x*y+z
- Which statements affect value of ‘x’ at statement #120?
- Which statements can be affected if I modify statement #20?

To answer the above questions, a programmer may need to examine huge amount of code. Doing this by hand may be time consuming and error prone.

1.1 What is an SPA and how is it used?

A **Static Program Analyzer (SPA for short)** is an interactive tool that automatically answers queries about programs. In this project, we design and implement an SPA for a simple source language.

The following scenario describes how an SPA is used by users:

1. John, a programmer, is given a task to fix an error in a program.
2. John feeds the program into SPA for automated analysis. The SPA parses a program into the internal representation stored in a Program Knowledge Base (PKB).

3. Now, John can start using SPA to help him find program statements that cause the crash. John repeatedly enters queries to the SPA. The SPA evaluates queries and displays results. John analyses query results and examines related sections of the program trying to locate the source of the error.
4. John finds program statement(s) responsible for an error. Now he is ready to modify the program to fix the error. Before that, John can ask the SPA more queries to examine a possible unwanted ripple effect of changes he intends to do.

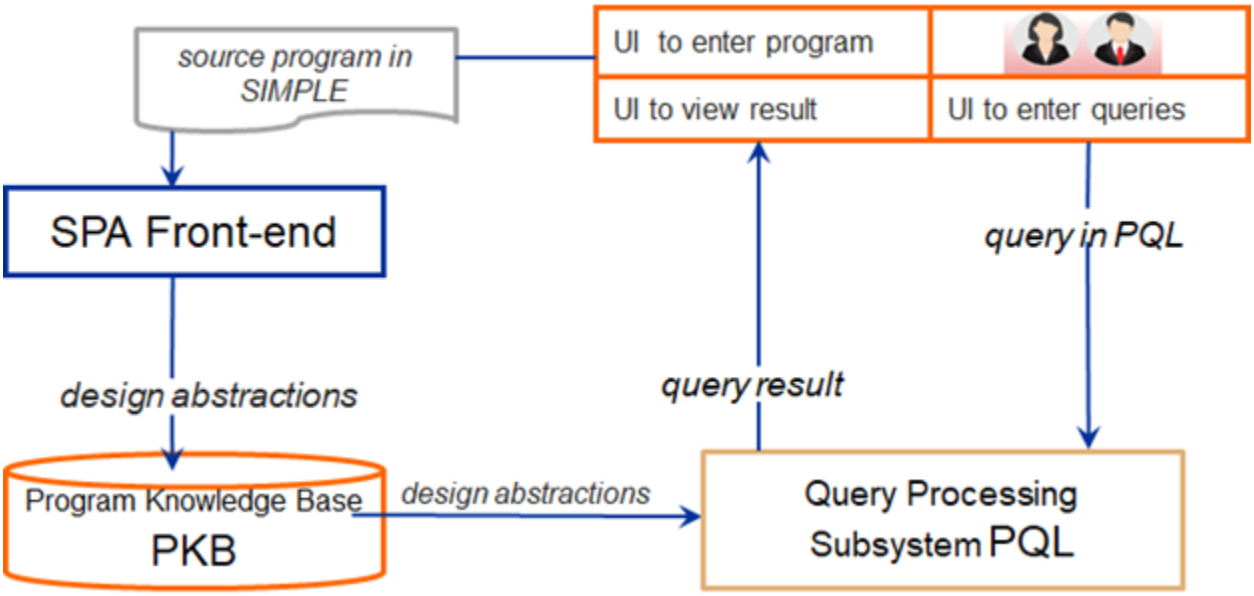
From the users' point of view, there are three actions that need to take place during static analysis: enter source program, enter query, and view query results.

1.2 How does a SPA work?

To answer program queries, the SPA must first analyze a source program and extract relevant program design entities, abstractions, Abstract Syntax Tree (AST), program Control Flow Graph (CFG) that are stored in a Program Knowledge Base (PKB). Secondly, the SPA must provide the user with means to ask questions about programs. These questions are written in a formal Program Query Language (PQL). SPA processes the PQL queries based on the information found in the PKB and returns the results to the user.

Figure 1 shows the main components of SPA. The users use the User Interface (UI) to enter a source program written in a simplified programming language called SIMPLE. SPA FrontEnd parses the source program according to the syntax grammar, extracts information, and stores it in PKB. These design abstractions include relationships (such as Calls, Modifies, Uses, etc) defined for program design entities (such as procedure, statement, variable). Furthermore, the user inputs queries written in PQL using the SPA's user interface. The queries are validated and evaluated by a Query Processor (QP). To answer queries, Query Processor makes use of the program design abstractions stored in the PKB in table form. Finally, the query results are displayed by the UI to the user.

Figure 1: Components of a Static Program Analyzer



2. Source Language SIMPLE

The static analysis is done on a simplified programming language called SIMPLE. SIMPLE is designed for the purpose of experimenting with SPA techniques, not as a language for solving real programming problems. It is designed for the purpose of this project to allow students to complete the project in one semester. However, SIMPLE contains all the basic constructs of a programming language for writing meaningful programs. The name of a SIMPLE program is given by the name of the first procedure in the program text.

Several sample programs that can be written in SIMPLE are shown below:

Code 1: Compute average of three integer numbers in SIMPLE.

```
procedure computeAverage {

   read num1;
   read num2;
   read num3;

   sum = num1 + num2 + num3;
   ave = sum / 3;
```

```
    print ave;
}
```

Code 2: Print numbers in ascending order in SIMPLE.

```
procedure printAscending {
    read num1;
    read num2;
    noSwap = 0;

    if (num1 > num2) then {
      temp = num1;
      num1 = num2;
      num2 = temp;
    } else {
      noSwap = 1;
    }

    print num1;
    print num2;
    print noSwap;
}
```

Code 3: Compute the sum of the digits of an integer in SIMPLE.

```
procedure sumDigits {
    read number;
    sum = 0;

    while (number > 0) {
        digit = number % 10;
        sum = sum + digit;
        number = number / 10;
    }

    print sum;
}
```

Code 4: Program with multiple procedures in SIMPLE.

```
    procedure main {
1     flag = 0;
2     call computeCentroid;
3     call printResults;
    }
    procedure readPoint {
4       read x;
5       read y;
    }
    procedure printResults {
6       print flag;
7       print cenX;
8       print cenY;
9       print normSq;
    }
    procedure computeCentroid {
10      count = 0;
11      cenX = 0;
12      cenY = 0;
13      call readPoint;
14      while ((x != 0) && (y != 0)) {
15          count = count + 1;
16          cenX = cenX + x;
17          cenY = cenY + y;
18          call readPoint;
        }
19      if (count == 0) then {
20          flag = 1;
        } else {
21          cenX = cenX / count;
            cenY = cenY / count;
        }
```

```
22      normSq = cenX * cenX + cenY * cenY;
    }
23
```

2.1. SIMPLE Language Rules

The following general rules apply for SIMPLE:

- A program consists of **one or more procedures.** Program execution starts by calling the first procedure.
- **Procedures**: non-empty list of statements, no parameters, no nesting, no recursion
- **Variables**: unique names, global scope, integer type, no declarations
- **Statements**: assignments, while loops, if-then-else statements, call statements, print, read
- **Conditions**: Boolean expressions containing operators
- **Operators**: +, -, *, /, %, <, >, etc.
- No arrays, no pointers

Program statements are of the following types:

- Procedure call, e.g., call readPoint;
  - Call procedure by name
- Assignment, e.g., x = 2; x = a + 2 * b;
  - Left hand side of the assignment is assigned with the expression on the right hand side.
- While statement, e.g., while (i > 0) { … }
  - Execute statements in the while body until the condition becomes false
- If statement, e.g., if (i > 0) then { … } else { … }
  - If condition is true, execute “then” branch; otherwise execute “else” branch. The else-branch is mandatory.
- Read input, e.g., read x;
  - Note that this is NOT to read the value from the variable.
  - Read the value for a variable from standard input (console). It is similar to scanf and cin in C/C++.
- Print output, e.g., print x;
  - Print statement outputs the value of a variable to standard output.

2.2. Statement Numbers

The statements in a SIMPLE program are indexed for easy referencing. The procedure definition does not receive an index. Furthermore, empty lines, “else” keywords on a line, curly brackets on a line do not receive an index. The first statement in the program located in the first procedure of the program is statement number 1 (stmt#1). The numbering continues from one procedure to the next. An example of how statements are numbered is shown in Code 4 snippet. For simplicity, the statement number is the same with the program line number, and vice-versa.

2.3. Syntax Grammar of SIMPLE

Once a SIMPLE program is entered through the user interface, the first step of the static program analysis is to parse the program. A program in SIMPLE is correct if it follows all the defined language rules. The rules are defined as a concrete syntax grammar (CSG) . CSG contains rules (or grammar productions) that are written using terminals and non-terminals. Keywords (terminals) are between apostrophes (e.g., ‘procedure’, ‘)’, ‘+’, ‘while’, ‘if’, etc.). Non-terminals are in lower-casing (e.g. var_name, term, expr, stmt_list, etc).

For example, the grammar for SIMPLE starts with a non-terminal “program”. A program contains one or more procedures. Each procedure is defined using the terminal keyword ‘procedure’, followed by a procedure name (non-terminal proc_name), followed by the keyword ‘{‘, followed by a statement list (non-terminal stmtLst), followed by ‘}’. Non-terminal stmtLst contains one or more statements (stmt), and stmt can be a read, print, call, while, if or assign statement.

Concrete Syntax Grammar (CSG) for SIMPLE

Meta symbols:

a* - repetition 0 or more times of a

a+ - repetition 1 or more times of a

a | b - a or b

brackets (and) are used for grouping

Lexical tokens:

LETTER: A-Z | a-z -- capital or small letter

DIGIT: 0-9

NAME: LETTER (LETTER | DIGIT)* -- procedure names and variables are strings of letters, and digits, starting with a letter

INTEGER: DIGIT+ -- constants are sequences of digits

Grammar rules:

program: procedure+

procedure: ‘procedure’ proc_name ‘{‘ stmtLst ‘}’

stmtLst: stmt+

stmt: read | print | call | while | if | assign

read: ‘read’ var_name‘;’

print: ‘print’ var_name‘;’

call: ‘call’ proc_name ‘;’

while: ‘while’ ‘(’ cond_expr ‘)’ ‘{‘ stmtLst ‘}’

if: ‘if’ ‘(’ cond_expr ‘)’ ‘then’ ‘{‘ stmtLst ‘}’ ‘else’ ‘{‘ stmtLst ‘}’

assign: var_name ‘=’ expr ‘;’

cond_expr: rel_expr | ‘!’ ‘(’ cond_expr ‘)’ | ‘(’ cond_expr ‘)’ ‘&&’ ‘(’ cond_expr ‘)’ | ‘(’ cond_expr ‘)’ ‘||’ ‘(’ cond_expr ‘)’

rel_expr: rel_factor ‘>’ rel_factor | rel_factor ‘>=’ rel_factor | rel_factor ‘<’ rel_factor | rel_factor ‘<=’ rel_factor | rel_factor ‘==’ rel_factor | rel_factor ‘!=’ rel_factor

rel_factor: var_name | const_value | expr

expr: expr ‘+’ term | expr ‘-’ term | term

term: term ‘*’ factor | term ‘/’ factor | term ‘%’ factor | factor

factor: var_name | const_value | ‘(’ expr ‘)’

var_name, proc_name: NAME

const_value: INTEGER

Additional rules that cannot be captured by the concrete syntax grammar follow:

1. Two procedures with the same name is considered an error.
2. Call to a non-existing procedure produces an error.
3. Recursive and cyclic calls are not allowed. For example, procedure A calls procedure B, procedure B calls C, and C calls A should not be accepted in a correct SIMPLE code.

Notes

1. SIMPLE is case-sensitive. The grammar shows the accepted casing for the keywords of the language. Due to case-sensitivity, variables “abc” and “Abc” are two different variables.
2. Spaces (including multiple spaces and tabs, or no spaces) can be used freely in SIMPLE. For example, tokenizer should recognize three tokens ‘x’, ‘+’ and ‘y” in any of the following three character streams:

x+y, x + y, x +y

1. The statements are indexed. To make things simple, the program line number is the same as the statement number.
2. Procedure names can be the same as variable names.
3. Logical expressions may only appear as the conditions for while/if statements and they are fully bracketed.
4. Constants are sequences of digits. If more than one digit, the first digit cannot be 0
5. Print statements only print a single variable (followed by a new line).
6. Read statements read in an integer for a single variable.
7. There are no Boolean values (true / false) since it is not meaningful. Example: while (true) {…} is always an infinite loop since there is no break or return. To get the same behaviour, flag variables can be set as 1 or 0 with comparison.
8. Expressions are left-associative due to the following grammar rules:

   expr: expr ‘+’ term | expr ‘-’ term | term

   term: term ‘*’ factor | term ‘/’ factor | term ‘%’ factor | factor

   factor: var_name | const_value | ‘(’ expr ‘)’

A concrete syntax grammar defines the language and its vocabulary. However, sometimes you might need to use abstractions to define a language. In that case, the abstract syntax grammar defines the relationships among different abstractions in a language, without defining the exact vocabulary. For

example, an expression (non-terminal expr) in the ASG of SIMPLE may be a plus, minus, times, div, or mod expression or a reference (non-terminal ref). There might be multiple concrete syntax grammars for the same abstract syntax grammar.

Abstract Syntax Grammar (ASG) for SIMPLE

Meta symbols:

a+ means a list of 1 or more a's;

'|' means or

Lexical tokens:

LETTER: A-Z | a-z -- capital or small letter

DIGIT: 0-9

NAME: LETTER (LETTER | DIGIT)*

INTEGER: DIGIT+

Grammar rules:

program: procedure+

procedure: stmtLst

stmtLst: stmt+

stmt: read | print | call | while | if | assign

read, print: variable

while: cond_expr stmtLst

if: cond_expr stmtLst stmtLst

assign: variable expr

cond_expr: rel_expr | not | and | or

not: cond_expr

and, or: cond_expr cond_expr

rel_expr: gt | gte | lt | lte | eq | neq

gt, gte, lt, lte, eq, neq: rel_factor rel_factor

rel_factor: variable | constant | expr

expr: plus | minus | times | div | mod | ref

plus, minus, times, div, mod: expr expr

ref: variable | constant

Attributes and attribute value types:

procedure.procName, call.procName, variable.varName, read.varName, print.varName: NAME

constant.value: INTEGER

stmt.stmt#, read.stmt#, print.stmt#, call.stmt#, while.stmt#, if.stmt#, assign.stmt#: INTEGER

3. Types of Information Stored in PKB

As mentioned in Section 1.2, once SPA parses the program it populates the information in the Program Knowledge Base (PKB). The information in the PKB is structured using tables, control flow graph, and/or abstract syntax trees. The tables store program design abstractions in the form of relationships (e.g., Follows, Modifies) between program design entities (e.g., statements, variables). The program design abstractions and entities are used to write queries in Program Query Language (PQL).

3.1 Abstract Syntax Tree (AST) for SIMPLE programs

In general, an abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression is denoted by means of a single node with three branches.

Core requirements when representing information is the AST include the following:

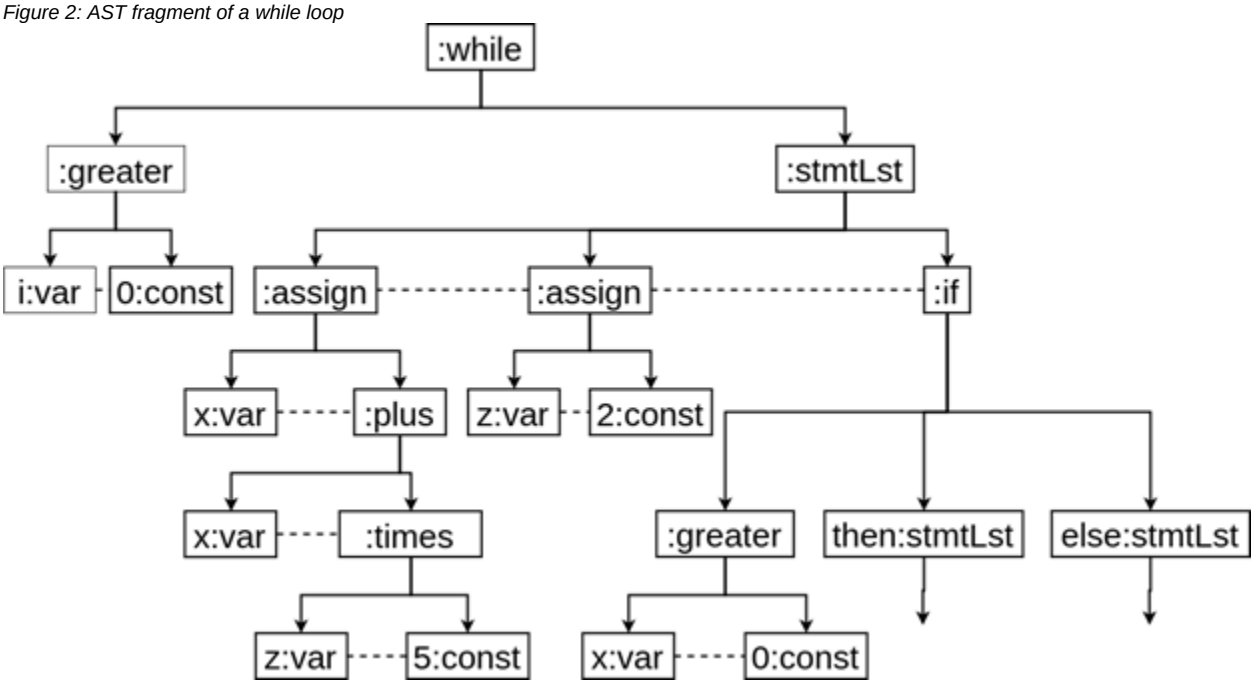
- The order of executable statements must be explicitly represented and well defined.
- Left and right components of binary operations must be stored and correctly identified.
- Identifiers and their assigned values must be stored for assignment statements.

AST is a schematic representation in the format of a tree of the SIMPLE program. The nodes typically correspond with the non-terminals of the syntax grammar (procedure, assign, if, while, plus, etc). The directed edges appear for each grammar rule (production).

Figure 2 shows a simplified representation of an AST for the while statement in the following code fragment:

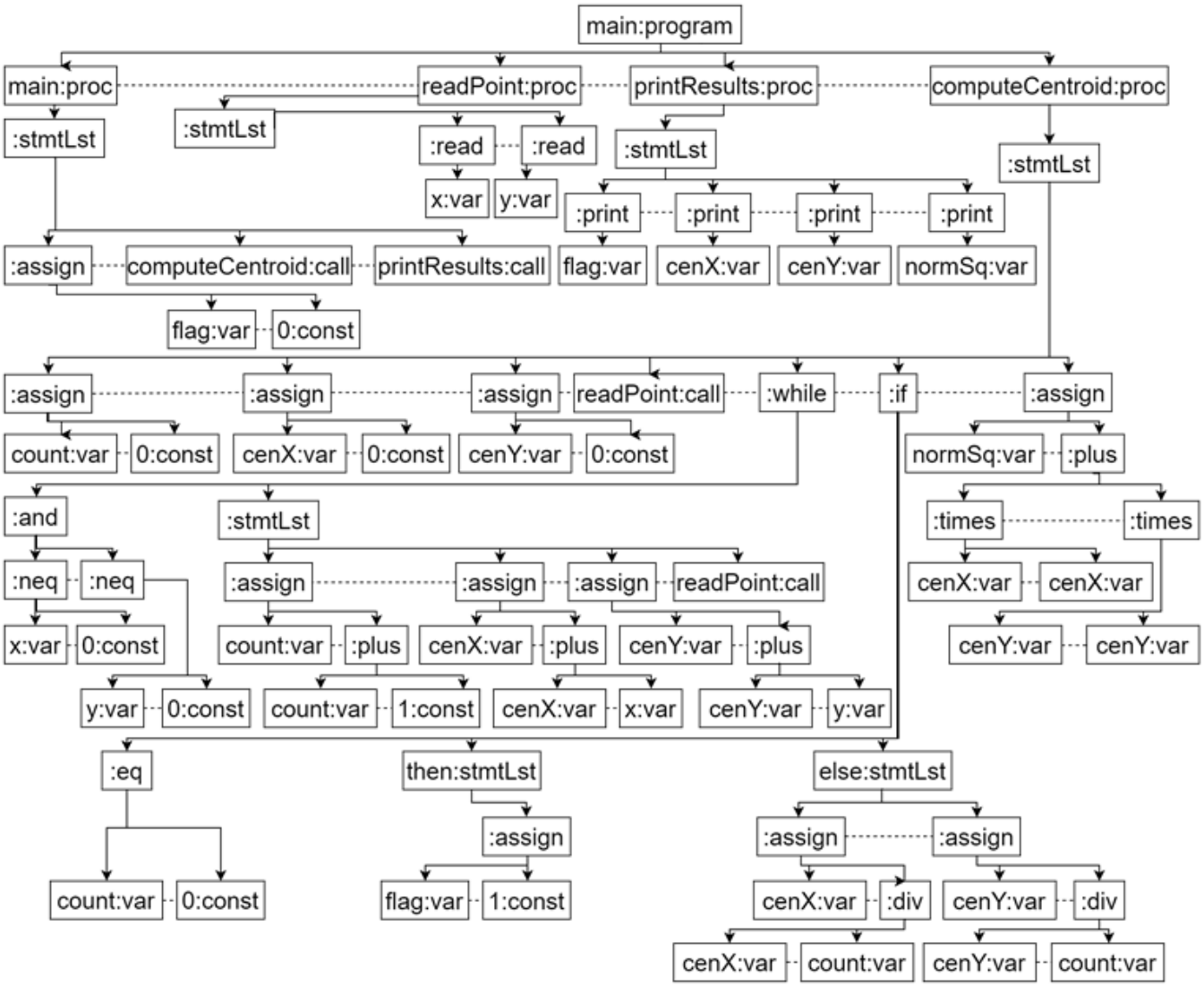
```
while (i > 0) {
   x = x + z * 5;
   z  = 2;
   if (x > 0) then {…} else {…}
}
```

Figure 2: AST fragment of a while loop



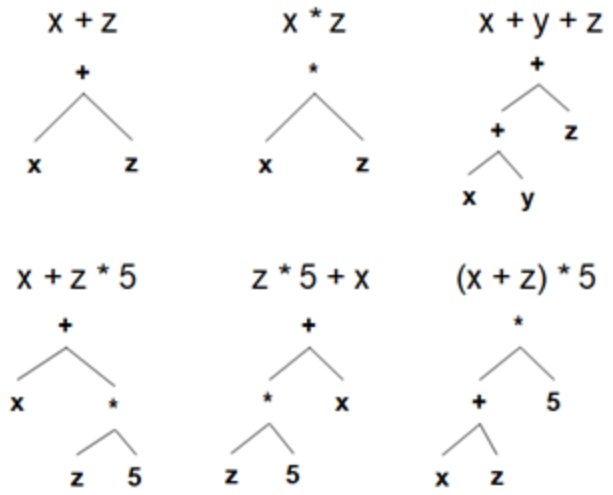
Formally, AST nodes are labeled with values (procedure/variable names, constants - placed before ':') and syntactic types (placed after ':'). For example, node readPoint:call represents a call to the procedure readPoint, and cenX:variable a node representing variable cenX. Figure 3 shows a full AST for program Main in Code 4 example.

Figure 3: Full AST for program Main in SIMPLE Code 4 example.



The graphical representation is flexible, as long as nodes of the tree are correct and the links in the tree match the SIMPLE program. For example, you may simplify the representation of a :plus node to a simple “+” sign in AST that correctly connects with the other elements of the assignment. Figure 4 shows a few ASTs for expressions containing plus and times operations in simplified representation. Note again that expressions are left-associative (AST for “x+y+z”)

Figure 4: Example ASTs for expressions



Observe that the AST is built strictly according to abstract syntax grammar rules for SIMPLE. Each non-leaf node corresponds to a syntactic type on the left-hand-side of some grammar rule, and its children nodes correspond to the syntactic types on the right-hand-side of that grammar rule. For any unambiguously defined language (such as SIMPLE), each well-formed program corresponds to exactly one AST. A node is a child of another node if it appears directly below it in AST. Child relationship is shown as a solid edge in Figures 2, 3 and 4.

3.2 Control Flow Graph

A Control Flow Graph (CFG) is a compact and intuitive representation of all control flow paths in a program. Nodes in the CFG are called basic blocks. A basic block contains program line numbers that are known to be executed one by one, in sequence. Basic blocks are formed of maximal program region with a single entry and single exit point or maximal code fragments executed without control transfer. That is, a decision point in 'while' or 'if' always marks the end of a basic block. We only need to explicitly show control flows among basic blocks using directed edges. Directed edges in CFG show the possibility that program execution proceeds from the end of one region directly to the beginning of another.

An example of CFG for Code 5 is shown in Figure 5:

Code 5: SIMPLE program used to explain advanced relationships and advanced PQL queries. For simplicity, Procedures First and Third are excluded from statement numbering.

```
        procedure First {
        read x;
        read z;
        call Second; }

        procedure Second {
1.          x = 0;
2.          i = 5;
3.          while (i!=0) {
4.              x = x + 2*y;
5.              call Third;
6.              i = i - 1; }
7.          if (x==1) then {
8.              x = x+1; }
            else {
9.              z = 1; }
10.         z = z + x + i;
11.         y = z + 2;
12.         x = x * y + z; }

        procedure Third {
            z = 5;
            v = z;
            print v; }
```

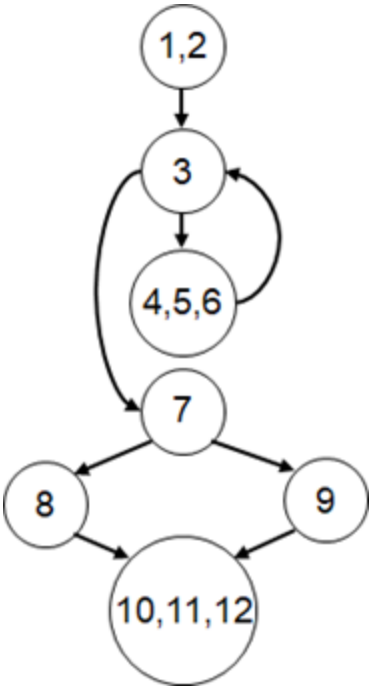


Figure 5: CFG for procedure Second in Code 5

Note that there is one CFG per procedure. Furthermore, the while statement (head) should be in a separate node from other statements in the loop (body). You may use dummy nodes (but not excessively) to ensure that

- if statements in the CFG have a diamond shape
- While loops have a loop shape
- Show procedure exit node

Following is is CFG for a procedure in Code 6

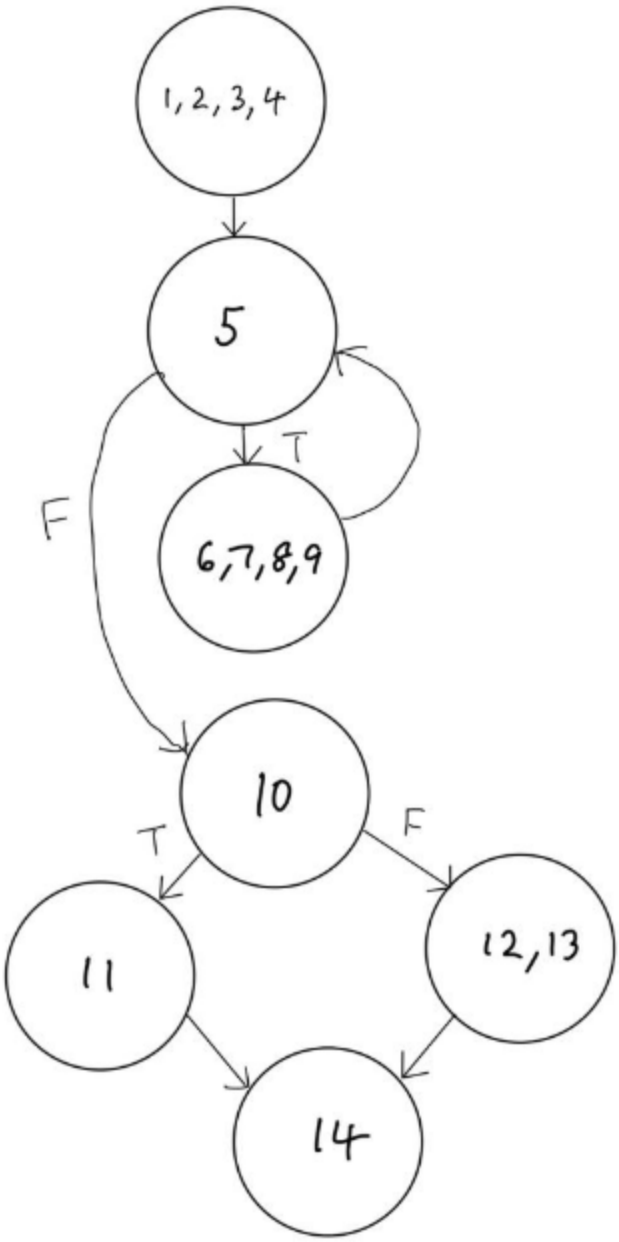


Figure 6: CFG for procedure computeCentroid in Code 6

3.3 Program Design Entities

The information stored in PKB about the SIMPLE program includes program design entities. The design entities characterize and help us refer to different abstraction from the program. A list of design entities for SIMPLE follows:

- program
- procedure
- stmt
- stmtLst
- read (statement)
- print (statement)
- assign (statement)
- call (statement)
- while (statement)
- if (statement)
- variable
- constant
- prog_line

3.4 Program Design Abstractions

Program design abstractions are relationships among program design entities. These relationships are divided between basic and advanced relationships.

Code 6: SIMPLE program used to explain basic relationships and basic PQL queries. For simplicity, procedures main, readPoint and printResults are excluded from statement numbering.

```
      procedure main {
        flag = 0;
        call computeCentroid;
        call printResults;
      }
      procedure readPoint {
          read x;
          read y;
      }
      procedure printResults {
          print flag;
          print cenX;
          print cenY;
          print normSq;
      }
      procedure computeCentroid {
1         count = 0;
2         cenX = 0;
3         cenY = 0;
4         call readPoint;
5         while ((x != 0) && (y != 0)) {
6             count = count + 1;
7             cenX = cenX + x;
8             cenY = cenY + y;
9             call readPoint;
          }
10        if (count == 0) then {
11            flag = 1;
          } else {
12            cenX = cenX / count;
13            cenY = cenY / count;
          }
14        normSq = cenX * cenX + cenY * cenY;
      }
```

3.4.1 Basic Relationships

A list of basic relationships follows:

Design Abstraction	Relationship between
Follows/ Follows*	Statements
Parent/ Parent*	Statements
Uses	Statement/Procedure and Variable
Modifies	Statement/Procedure and Variable

a. Follows/Follows*

Definition:

For any statements s1 and s2:

- Follows (s1, s2) holds if they at the same nesting level, in the same statement list (stmtLst), and s2 appears in the program text immediately after s1
- Follows* (s1, s2) holds if
    - Follows (s1, s2) or
    - Follows (s1, s) and Follows* (s, s2) for some statement s

Follows* is the transitive closure of Follows.

With reference to Code 6, statements are referenced via their statement numbers, e.g, Follows (4, 5). Statement number ‘5’ refers to the whole while statement, including lines 5-9, rather than to the program line “while ((x != 0) && (y != 0)) {“ only. Similarly, statement number ‘10’ refers to the whole if statement including lines 10-13.

For example, in procedure computeCentroid (Code 6) the following relationships hold (are true):

- Follows (1, 2)
- Follows (4, 5)
- Follows (5, 10)
- Follows* (3,10)
- Follows* (1, 14)

In contrast, the following relationships are false:

- Follows (5, 6)
- Follows (9, 10)
- Follows (11, 12)
- Follows* (12, 14)

(Hint: It is useful to check which statement list directly contains the statement mentioned in a relationship.)

b. Parent/Parent*

Definition:

For any statements s1 and s2:

- Parent (s1, s2) holds if s2 is directly nested in s1
- Parent* (s1, s2) holds if
    - Parent (s1, s2) or
    - Parent (s1, s) and Parent* (s, s2) for some statement s

Parent* is the transitive closure of Parent.

As in the case of Follows, number '5' refers to the whole while statement and number '10' refers to the whole if statement for Code 6.

For example, in procedure computeCentroid (Code 6) the following relationships hold:

- Parent (5, 7)
- Parent (5, 8)
- Parent (10, 11)
- Parent (10, 13)
- Parent* (5, 7)
- Parent* (10, 13)

The following relationships are false:

- Parent (2, 3)
- Parent (4, 7)
- Parent (9, 5)
- Parent* (10, 14)

(Hint: The first statement should be a container statement, while the second statement should be nested inside the first statement.)

c. Uses

For Design Entities	Definition
Assignment a Variable v	Uses (a, v) holds if variable v appears on the right hand side of a.
Print statement pn Variable v	Uses (pn, v) holds if variable v appears in pn.
Container statement s (if or while) Variable v	Uses (s, v) holds if v appears in the condition of s, or there is a statement s1 in the container such that Uses(s1, v) holds
Procedure p Variable v	Uses (p, v) holds if there is a statement s in p or in a procedure called (directly or indirectly) from p such that Uses (s, v) holds.
Procedure call c ("call p ") Variable v	Uses (c, v) is defined in the same way as Uses (p, v).

With reference to Code 6, statements are referenced via their statement numbers, variables via their variable names, and procedures via their procedure names.

For example, in program main (Code 6) the following relationships hold (are true):

- Uses (7, “x”)
- Uses (10, “count”)
- Uses (10, “cenX”)
- Uses (“main”, “cenX”)
- Uses (“main”, “flag”)
- Uses (“computeCentroid”, “x”)

If a number refers to statement s that is a procedure call, then Uses (s, v) holds for any variable v used in the called procedure (or in any procedure called directly or indirectly from that procedure). For example, “flag” is used in the print statement of procedure printResults; hence the call statement to printResults in procedure main uses flag; therefore, “main” uses “flag”.

Also, if a number refers to a container statement s (while or if statement), then Uses (s, v) holds for any variable used by any statement in the container s (including call statements), or used in the condition of s.

For example, statement 10 contains statement 12 that uses “cenX”; hence 10 uses “cenX”. In contrast, the following relationships are false (do not hold):

- Uses (3, “count”)
- Uses (10, “flag”)
- Uses (9, “y”)

d. Modifies

For Design entities	Description
Assignment a Variable v	Modifies (a, v) holds if variable v appears on the left hand side of a.
Read statement r Variable v	Modifies (r, v) holds if variable v appears in r.
Container statement s (“if” or “while”) Variable v	Modifies (s, v) holds if there is a statement s1 in the container such that Modifies (s1, v) holds.
Procedure p, Variable v	Modifies (p, v) holds if there is a statement s in p or in a procedure called (directly or indirectly) from p such that Modifies (s, v) holds.
Procedure call c (“call p”) Variable v	Modifies (c, v) is defined in the same way as Modifies (p, v).

With reference to Code 6, statements are referenced via their statement numbers, variables via their variable names, and procedures via their procedure names.

For example, in program main (Code 6) the following relationships hold (are true):

- Modifies (1, “count”)
- Modifies (7, “cenX”)
- Modifies (9, “x”)
- Modifies (10, “flag”)
- Modifies (5, “x”)
- Modifies (“main”, “y”)

If a number refers to statement s that is a procedure call, then Modifies (s, v) holds for any variable v modified in the called procedure (or in any procedure called directly or indirectly from that procedure). For example, “y” is modified in the call statement “call computeCentroid” from “main” because procedure “computeCentroid” calls procedure “readPoint” that modifies “y” in a read statement.

Also, if a number refers to a container statement s (while or if statement), then Modifies (s, v) holds for any variable modified by any statement in the container s (including call statements). For example, statement 5 contains statement 9 that modifies “x” (in procedure readPoint read statement); hence 5 modifies “x”.

The following relationships are false (do not hold):

- Modifies (5, “flag”)
- Modifies (“printResults”, “normSq”)

3.4.2 Advanced Relationships

A list of advanced relationships follows:

Design Abstraction	Relationship between
Calls/Calls*	Procedures
Next/Next*	Program lines
Affects/Affects*	Assignments

e. Calls/Calls*

Definition:

For any procedures p and q:

- Calls (p, q) holds if procedure p directly calls q
- Calls* (p, q) holds if procedure p directly or indirectly calls q , that is:
    - Calls (p, q) or
    - Calls (p, p1) and Calls* (p1, q) for some procedure p1

For example, in Code 5 the following relationships hold (are true):

- Calls (“First”, “Second”)
- Calls (“Second”, “Third”)
- Calls* (“First”, “Second”)
- Calls* (“First”, “Third”)

The following relationships are false (do not hold):

- Calls (“First”, “Third”)
- Calls (“Second”, “First”)
- Calls* (“Second”, “First”)

f. Next/Next*

Definition:

Relationship Next defines the control flow:

For two program lines n1 and n2 in the same procedure:

- Next (n1, n2) holds if n2 can be executed immediately after n1 in some execution sequence
- Next* (n1, n2) holds if n2 can be executed after n1 in some execution sequence

Note that we define control flow (Next) only in the scope of a procedure, not across procedures. Program lines belonging to two different procedures cannot be related by means of relationship Next.

In case of assignments or procedure calls, a line number also corresponds to a statement (see Statement numbers). But for container statements, line numbers refer to the “header” of the container rather than the whole statement as it was the case before. For example in Code 5, line number 3 refers to “w hile (i!=0) {“ and line number 7 refers to “if (x==1) then {“.

For example, in Code 5 the following relationships hold (are true):

- Next (2, 3)
- Next (3, 4)
- Next (3, 7)
- Next (5, 6)
- Next (7, 9)
- Next (8, 10)
- Next* (1, 2)
- Next* (1, 3)
- Next* (2, 5)

- Next* (4, 3)
- Next* (5, 5)
- Next* (5, 8)
- Next* (5, 12)

The following relationships are false (do not hold):

- Next (6, 4)
- Next (7, 10)
- Next (8, 9)
- Next* (8, 9)
- Next* (5, 2)

g. Affects/Affects*

Definition:

Affects (a1, a2) holds if:

- a1 and a2 are in the same procedure
- a1 modifies a variable v which is used in a2
- There is a control flow path from a1 to a2 on such that v is not modified (as defined by Modifies relationship) in any assignment, read, or procedure call statement on that path

Relationship Affects models data flows in a program. Relationship Affects is defined only among assignment statements and involves a variable. Informally, the relationship Affects (a1, a2) holds, if the value of v as computed at a1 may be used at a2.

For example, in Code 5 the following relationships hold (are true):

- Affects (2, 6)
- Affects (4, 8)
- Affects (4, 10)
- Affects (6, 6)
- Affects(1,4)
- Affects(1,8)
- Affects(1,10)
- Affects(1,12)
- Affects(2,10)
- Affects(9,10)

Affects (1,12) holds because variable x is modifies in 1 and used in 12 and there is a path in the CFG (1->2->3->7->9->10->11->12) on which variable x is not modified (according to the definition of Modifies by any assignment, read, or procedure call).

The following relationships are false (do not hold) in Code 5:

- Affects (9, 11)
- Affects (9,12)
- Affects (2, 3)
- Affects (9, 6)

Affects (9,12) does not hold as the value of z is modified by assignment 10 that is on any control flow path between assignments 9 and 12.

Suppose you have the following codes:

Code 7: Sample code for Affects with procedure calls

```
1.  x=a;
2.  call p;
3.  v=x;
```

Code 8: Sample code for Affects with procedure call and if statement

```
procedure p {

1.  x = 1;
2.  y = 2;
3.  z = y;
4.  call q;
5.  z = x + y + z;}

procedure q {
```

```
6.  x = 5;
7.  t = 4;
8.  if (z>0) then {
9.      t = x + 1;}
    else {
10.     y = z + x;}
11. x = t + 1; }
```

In Code 7, if procedure p modifies variable x, that is Modifies (p, “x”) holds, then assignment Affects (1, 3) DOES NOT HOLD, as procedure call ‘kills’ the value of x as assigned in statement 1. If procedure p does not modify variable x; then Affects (1,3) HOLDS.

In Code 8, Affects (1,5) and Affects (2,5) do not hold as both x and y are modified in procedure q. Even though modification of variable y in procedure q is only conditional, we take it as if variable y was always modified when procedure q is called (which is in sync with our definition of Modifies for procedures). We make this assumption to simplify analysis for SPA. Affects (3,10) do not hold because assignments 3 and 10 are in different procedures.

Next, we will explain two examples for Affects with container statement and procedure call:

Container statement	**Procedure call**

procedure alpha {
1. x = 1;
2. if (i!=2) {
3. x = a + 1;}
else {
4. a = b; }
5. a = x; }
	procedure alpha {
	1. x = 1;
	2. call beta;
	3. a = x; }
	
	procedure beta {
	4. if (i!=2) {
	5. x = a + 1;}
	else {
	6. a = b; }}

Affects (1,5) is true because x is modified in 1 and use din 5, and there is a path in the CFG (1->2->4->5) on which x is not modified in any assignment, read, or procedure call.	Affects (1,3) is false because x is modified in 1 and used in 3, but the only paths from 1 to 3 (1->2->3) contains procedure 2 that modifies x according to the definition of Modifies for procedure calls.

Suppose you have the following code:

Code 9: Sample code for Affects with read statement

```
1. x=a;
2. read x;
3. v=x;
```

In Code 9, Affects (1, 3) does not hold because 2 modifies x on the paths from 1 to 3.

Definition:

Affects* (a1, a2) holds if:

- Affects (a1, a2) or
- Affects (a1, a) and Affects* (a, a2) for some assignment statement a

Consider the following program fragment as an illustration of the basic principle:

```
1. x=a;
2. v=x;
```

3. z=v;

Modification of x in statement 1 affects variable v in statement 2 and modification of v in statement 2 affects use of variable v in statement 3. Hence we have: Affects (1,2), Affects (2,3) and Affects*(1,3).

For example, in Code 5 the following relationships hold (are true):

- Affects*(1,4)
- Affects*(1,10)
- Affects*(1,11)
- Affects*(1,12)

4. Query Language (PQL)

PQL queries are expressed in terms of program design models (entities and abstractions). In general, queries reference design entities (such as procedure, variable, assign, etc.), attributes (such procedure.procName or variable.varName), relationships (such as Calls (procedure, procedure)) and syntactic patterns (such as assign (variable, expr)). Evaluation of a query yields a list of program elements that match a query. Program elements are specific instances of design entities, for example, procedure named “main”, statement number 35, or variable named “x”.

4.1 Basic Query Language

Basic queries in PQL contain:

- Declaration of synonyms to be used in the query
  - Example: procedure p; variable v; (p: entity procedure, v: entity variable)
- Select clause specifies query result
  - Single return values
  - One such that clause constrains the results in terms of relationships
  - One pattern clause constrain results in terms of code patterns
- Query results must check (make true) all clauses

Grammar of basic PQL

Meta symbols:

a* - repetition 0 or more times of a

a+ - repetition 1 or more times of a

[a] - repetition 0 or one occurrence of ‘a’

a | b - a or b

brackets (and) are used for grouping

Lexical tokens:

LETTER: A-Z | a-z -- capital or small letter

DIGIT: 0-9

IDENT : LETTER (LETTER | DIGIT)*

NAME : LETTER (LETTER | DIGIT)*

INTEGER : DIGIT+

synonym : IDENT

stmtRef : synonym | ‘_’ | INTEGER

entRef : synonym | ‘_’ | “” IDENT “”

Grammar Rules:

select-cl : declaration* ‘Select’ synonym [suchthat-cl] [pattern-cl]

declaration : design-entity synonym (‘,’ synonym)* ‘;’

design-entity : ‘stmt’ | ‘read’ | ‘print’ | ‘call’ | ‘while’ | ‘if’ | ‘assign’ | ‘variable’ | ‘constant’ | ‘procedure’

suchthat-cl : ‘such that’ relRef

relRef : Follows | FollowsT | Parent | ParentT | UsesS | UsesP | ModifiesS | ModifiesP

Follows : ‘Follows’ ‘(’ stmtRef ‘,’ stmtRef ‘)’

FollowsT : ‘Follows*’ ‘(’ stmtRef ‘,’ stmtRef ‘)’

Parent : ‘Parent’ ‘(’ stmtRef ‘,’ stmtRef ‘)’

ParentT : ‘Parent*’ ‘(’ stmtRef ‘,’ stmtRef ‘)’

UsesS : ‘Uses’ ‘(’ stmtRef ‘,’ entRef ‘)’

UsesP : ‘Uses’ ‘(’ entRef ‘,’ entRef ‘)’

ModifiesS : ‘Modifies’ ‘(’ stmtRef ‘,’ entRef ‘)’

ModifiesP : ‘Modifies’ ‘(’ entRef ‘,’ entRef ‘)’

Note: It is semantically invalid to have ‘_’ as the first argument for Modifies and Uses because it is unclear whether ‘_’ stands for a statement or a procedure.

pattern-cl : ‘pattern’ syn-assign ‘(‘ entRef ‘,’ expression-spec ’)’

// syn-assign must be declared as synonym of assignment (design entity ‘assign’).

expression-spec : “”‘ expr’”” | ‘_’ “”’ expr ‘””’ ‘_’ | ‘_’

expr: expr ‘+’ term | expr ‘-’ term | term

term: term ‘*’ factor | term ‘/’ factor | term ‘%’ factor | factor

factor: var_name | const_value | ‘(’ expr ‘)’

var_name: NAME

const_value : INTEGER

With respect to Code 6, a few examples of simple questions that can be translated into PQL queries follow:

a. Queries with no such that and pattern clause:

Q1. What are the procedures in the program?

procedure p;

Select p

The declaration “procedure p” refers to entities of type procedures found in the SIMPLE program that is analyzed. This query returns as a result all the procedures in the program. The results are displayed as a list of procedure names.

With respect to Code 6, the results of evaluating the query are:

-- answer: *procedures “main”, “readPoint”, “printResults”, “computeCentroid”*

In your SPA implementation, the format of the result should be a list of strings without quotes (“) and additional words such as “variable” :

-- answer: *main, readPoint, printResults, computeCentroid*

The order of the names of the procedures does not matter. Hence, another correct result is:

-- answer: *printResults, main, readPoint, computeCentroid*

Q2. What are the variables in the program?

variable v;

Select v

The query returns all variable names:

-- answer: *variables “flag”, “count”, “cenX”, “cenY”, “x”, “y”, “normSq”*

In your SPA implementation, the format of the result should be a list of strings without quotes (“) and additional words such as “variable” :

-- answer: *flag, count, cenX, cenY, x, y, normSq*

b. Queries with one such that clause:

Q3. Which statements follow assignment 6 directly or indirectly?

stmt s;

Select s such that Follows (6, s)*

-- answer: *statements #7, #8, and #9*

In your SPA implementation, the format of the result should be a list of statement numbers, without # and words such as “statements” or “stmts”:

-- answer: *7, 8, 9*

Q4. Which variables have their values modified in statement 6?

variable v;

Select v such that Modifies (6, v)

-- answer: variable “count”

Q5. Which variables are used in assignment 14?

variable v;

Select v such that Uses (14, v)

-- answer: *variables “cenX” and “cenY”*

Q6. Which procedures modify variable “x”?

variable v; procedure p;

Select p such that Modifies (p, “x”)

-- answer: *procedures “main”, “computeCentroid” and “readPoint”*

Q7. Find assignments within a loop.

assign a; while w;

Select a such that Parent (w, a)*

-- answer: *statements #6, #7 and #8*

Q8. Which is the parent of statement #7?

stmt s;

Select s such that Parent (s, 7)

-- answer: *statement #5*

c. Queries with one pattern clause:

You may want to find all places in you code that match a code pattern. In such cases you can use pattern clause in a PQL query. The pattern clause for expressions comprises of a synonym name of type variable, the pattern for the left hand side (LHS) of the expression, and the pattern for the right hand side (RHS) of the expression. LHS can be an exact match (a variable name between double quotes), a synonym of type variable, or anything/unrestricted (represented using “_”). RHS can be an exact match (expression between double quotes), a partial match (subexpression between double quotes surrounded by “_”), or anything/unrestricted (represented using “_”).

Assuming your SIMPLE source code contains only one procedure with only one assignment (stmt# 1):

1. x = v + x * y + z * t

Which of the following patterns match this assignment statement?

assign a;

Pattern PQL query	Returns	Explanation

*Select a pattern a (_ , “v + x * y + z * t”)*	*stmt #1*	Exact pattern match
Select a pattern a (_ , “v”)	*none*	stmt #1 contains other expression terms other than v
Select a pattern a (_ , _“v”_)	*stmt #1*	stmt #1 contains v as part of the expression on the RHS
*Select a pattern a (_ , _“x*y”_)*	*stmt #1*	stmt #1 contains x*y as a term on the RHS
Select a pattern a (_ , _“v+x”_)	*none*	stmt #1 does not contain v+x as a sub-expression on the RHS
*Select a pattern a (_ , _“v+x*y”_)*	*stmt #1*	stmt #1 contains v+x*y as a sub-expression on the RHS
*Select a pattern a (_ , _“y+z*t”_)*	*none*	stmt #1 does not contain y+z*t as a sub-expression on the RHS
*Select a pattern a (_, _“x * y + z * t”_)*	*none*	stmt #1 does not contain x*y+z*t as a sub-expression on the RHS
*Select a pattern a (_ , _“v + x * y + z * t”_)*	*stmt #1*	stmt #1 contains v+x*y+z*t as a sub-expression on the RHS

All sub-expressions in pattern matching an expression are sub-trees in the AST.

As such, all subexpressions for stmt# 1 are:

- x, y, z, t, v
- v, x * y, z * t,
- v +x * y,
- v + x * y + z * t

Another way to identify a sub-expression is to use brackets, and every bracket is a sub-expression:

1. x = (((v) + ((x) * (y))) + ((z) * (t)))

Do not forget that expression in SIMPLE are left-associative (by CSG).

With reference to Code 6, the following questions can be translated into PQL queries:

Q9. Find assignments that contain expression count + 1 on the right hand side

assign a;

Select a pattern a (_ , “count + 1”)

-- answer: *statement #6*

Q10. Find assignments that contain sub-expression cenX * cenX on the right hand side and normSq on the left hand side

assign a;

*Select a pattern a (“normSq” , _“cenX * cenX”_)*

-- answer: *statement #14*

Same query can be written using a different variable name:

assign newa;

*Select newa pattern newa (“normSq” , _“cenX * cenX”_)*

-- answer: *statement #14*

d. Queries with one pattern clause and one such that clause:

Q11: Find while loops with assignment to variable “count”

assign a; while w;

Select w such that Parent (w, a) pattern a (“count”, _)*

-- answer: *statement #5*

Q12. Find assignments that use and modify the same variable

assign a; variable v;

Select a such that Uses (a, v) pattern a (v, _)

-- answer: *assignments #6, #7, #8, #12, and #13*

Note that many questions can be correctly translated in multiple ways into PQL. Moreover, results returned by a query must satisfy all conditions (clauses) of the query at the same time. Changing the order of conditions (clauses) in a query does not change the query result; but changing the order of conditions may affect query evaluation time. For example:

Q13: Find assignments that use and modify the variable "x"

assign a; while w;

Select a pattern a ("x", _) such that Uses (a, "x")

or

Select a such that Uses (a, "x") pattern a ("x", _)

-- answer: *none*

Q14: Find assignments that are nested directly or indirectly in a while loop and modify the variable "count"

assign a; while w;

Select a such that Parent (w, a) pattern a ("count", _)*

or

Select a pattern a ("count", _) such that Parent (w, a)*

-- answer: *statement #6*

The format of the result returned by PQL queries is summarized below:

Select	**Should return**
Statement (stmt / read / print / call / while / if / assign)	Statement number
Variable	Name (no need to use "")
Procedure	Name (no need to use "")
Constant	Constant value
Empty result (no entities matching the query)	On paper, keyword "none" In your SPA implementation, do not populate the list of results with any values (not even keyword "none")

4.2 Advanced Query Language

Basic queries in PQL contain:

- Declaration of synonyms to be used in the query
    - Example: procedure p; variable v; (p: entity procedure, v: entity variable)
- Select clause specifies query result
    - single or multiple return values (tuples) or BOOLEAN
    - such that clauses constrain the results in terms of relationships
    - pattern clauses constrain results in terms of code patterns
    - with clauses constrain the results in terms of attribute values
- Query results must check (make true) all clauses

All clauses are optional. Clauses "such that", "with" and "pattern" may occur many times in the same query. There is an implicit and operator between clauses – that means a query result must satisfy the conditions specified in all the clauses.

A query reports any result for which there exists a combination of synonym instances satisfying all the conditions specified in such that, with and pattern clauses. The existential quantifier is always implicit in PQL queries.

The result of Select BOOLEAN is TRUE if there are no constraints in the query or there exists a combination of synonym instances satisfying all the conditions specified in the query; otherwise, FALSE.

We introduce new queries by examples, referring to program design abstractions and entities explained before.

Full grammar of PQL

Meta symbols:

a* - repetition 0 or more times of a

a+ - repetition 1 or more times of a

a | b - a or b

brackets (and) are used for grouping

Lexical tokens:

LETTER: A-Z | a-z -- capital or small letter

DIGIT: 0-9

IDENT : LETTER (LETTER | DIGIT)*

NAME : LETTER (LETTER | DIGIT)*

INTEGER : DIGIT+

synonym : IDENT

stmtRef : synonym | '_' | INTEGER

entRef : synonym | '_' | '"' IDENT '"'

lineRef : synonym | '_' | INTEGER

elem : synonym | attrRef

attrName : 'procName'| 'varName' | 'value' | 'stmt#'

design-entity : 'stmt' | 'read' | 'print' | 'call' | 'while' | 'if' | 'assign' | 'variable' | 'constant' |

'prog_line' | 'procedure'

Grammar Rules:

select-cl : declaration* 'Select' result-cl (suchthat-cl | with-cl | pattern-cl)*

declaration : design-entity synonym (',' synonym)* ';'

result-cl : tuple | 'BOOLEAN'

tuple: elem | '<' elem (',' elem)* '>'

with-cl : 'with' attrCond

suchthat-cl : 'such that' relCond

pattern-cl : 'pattern' patternCond

attrCond : attrCompare ('and' attrCompare)*

attrCompare : ref '=' ref

// the two refs must be of the same type (i.e., both strings or both integers)

ref : '"' IDENT '"' | INTEGER | attrRef | synonym

// synonym must be of type 'prog_line'

attrRef : synonym '.' attrName

relCond : relRef ('and' relRef)*

relRef : ModifiesP | ModifiesS | UsesP | UsesS | Calls | CallsT |

    Parent | ParentT | Follows | FollowsT | Next | NextT | Affects | AffectsT

ModifiesP : 'Modifies' '(' entRef ',' entRef ')'

ModifiesS : 'Modifies' '(' stmtRef ',' entRef ')'

UsesP : ‘Uses’ ‘(’ entRef ‘,’ entRef ‘)’

UsesS : ‘Uses’ ‘(’ stmtRef ‘,’ entRef ‘)’

// The first argument for Modifies and Uses cannot be ‘_’ because it is unclear whether the ‘_’ stands for a statement or a procedure.

Calls : ‘Calls’ ‘(’ entRef ‘,’ entRef ‘)’

CallsT : ‘Calls*’ ‘(’ entRef ‘,’ entRef ‘)’

Parent : ‘Parent’ ‘(’ stmtRef ‘,’ stmtRef ‘)’

ParentT : ‘Parent*’ ‘(’ stmtRef ‘,’ stmtRef ‘)’

Follows : ‘Follows’ ‘(’ stmtRef ‘,’ stmtRef ‘)’

FollowsT : ‘Follows*’ ‘(’ stmtRef ‘,’ stmtRef ‘)’

Next : ‘Next’ ‘(’ lineRef ‘,’ lineRef ‘)’

NextT : ‘Next*’ ‘(’ lineRef ‘,’ lineRef ‘)’

Affects : ‘Affects’ ‘(’ stmtRef ‘,’ stmtRef ‘)’

AffectsT : ‘Affects*’ ‘(’ stmtRef ‘,’ stmtRef ‘)’

patternCond : pattern (‘and’ pattern)*

pattern : assign | while | if

assign : syn-assign ‘(‘ entRef ‘,’ expression-spec ‘)’

// syn-assign must be of type ‘assign’.

expression-spec : “”‘ expr’”” | ‘_’ “”’ expr ‘””’ ‘_’ | ‘_’

expr: expr ‘+’ term | expr ‘-’ term | term

term: term ‘*’ factor | term ‘/’ factor | term ‘%’ factor | factor

factor: var_name | const_value | ‘(’ expr ‘)’

var_name: NAME

const_value : INTEGER

if : syn-if ‘(’ entRef ‘,’ ‘_’ ‘,’ ‘_’ ‘)’

// syn-if must be of type ‘if’

while : syn-while ‘(’ entRef ‘,’ ‘_’ ‘)’

// syn-while must be of type ‘while’

Summary of Program Design Models

For your reference, here is a summary of program design entities, attributes and relationships defined in program design models. When writing program queries, we can refer ONLY to entities, attributes and relationships listed below.

Program design entities:

program, procedure

stmt, stmtLst, assign, call, while, if, read, print

plus, minus, times, div, mod

variable, constant

prog_line

Attributes and attribute value types:

procedure.procName, call.procName, variable.varName, read.varName, print.varName : NAME

constant.value : INTEGER

stmt.stmt#, read.stmt#, print.stmt#, call.stmt#, while.stmt#, if.stmt#, assign.stmt#: INTEGER

Query examples

With respect to Code 5, a few examples of questions that can be translated into PQL queries (using the basic and advanced relationships) follow:

Q15: What are the procedures in the program call another procedure?

procedure p, q;

Select p such that Calls (p, _) Or Select p.procName such that Calls (p, q)

--- answer: procedures First, Second

Note: Dot '.' notation means a reference to the attribute value of an entity (procedure name in this case).

Underscore '_' is a placeholder for an unconstrained design entity (procedure in this case). Symbol '_' can be only used when the context uniquely implies the type of the argument denoted by '_'. Here, we infer from the program design model that '_' stands for the design entity "procedure".

Q16: Which procedures directly call procedure Third and modify i?

procedure p, q;

Select p such that Calls (p, q) with q.procName = "Third" such that Modifies (p, "i")

Note: The with clause constrains that the procedure name of q is "Third".

Short form to avoid with clause:

procedure p;

Select p such that Calls (p, "Third") and Modifies (p, "i")

--- answer: procedure Second

Q17: Which procedures call procedure "Third" directly or indirectly?

procedure p;

Select p such that Calls (p, "Third")*

--- answer: procedures First and Second

Q18: Which procedures are called from "Second" in a while loop?

procedure p; call c; while w;

Select p such that Calls ("Second", p) and Parent (w, c) with c.procName = p. procName

--- answer: procedure Third

Q19: Is there an execution path from statement #2 to statement #9?

Select BOOLEAN such that Next (2, 9)*

--- answer: TRUE

Q20: Is there an execution path from statement #2 to statement #9 that passes through statement #8?

Select BOOLEAN such that Next (2, 8) and Next* (8, 9)*

--- answer: FALSE

Q21: Find assignments to variable "x" located in a loop, that can be reached (in terms of control flow) from statement #1.

assign a; while w;

Select a pattern a ("x", _) such that Parent (w, a) and Next* (1, a)*

or

Select a such that Modifies (a, "x") and Parent (w, a) and Next* (1, a)*

-- answer: statement #4

Q22: Which program lines can be executed between line #5 and line #12?

prog_line n;

Select n such that Next (5, n) and Next* (n, 12)*

--- answer: statements #3-11

Q23: Which assignments directly or indirectly affect value computed at assignment #10?

assign a;

Select a such that Affects (a, 10)*

--- answer: statements #9, #1, #4, #8, #2, #6

Q24. Which are the pair of assignments that affect each other?

assign a1, a2;

Select <a1, a2> such that Affects (a1, a2)

or

Select <a1.stmt#, a2> such that Affects (a1, a2)

--- answer: …

Q25. Which assignments directly or indirectly affect the value computed at assignment #10?

assign a;

Select a such that Affects (a, 10)*

--- answer: 1, 2, 4, 6, 8, 9

Q26. Find all pairs of procedures p and q such that p calls q.

procedure p, q;

Select <p,q > such that Calls (p, q)

--- answer: <First, Second>, <Second, Third>

The following queries do not refer to a SIMPLE program source code. They are meant to help you understand how to use PQL.

Q27. Find all statements whose statement number is equal to some constant.

stmt s; constant c;

Select s with s.stmt# = c.value

Q28. Find procedures whose name is the same as the name of some variable.

stmt s; procedure p; var v;

Select p with p.procName = v.varName

Q29. Find statements that follow 10.

prog_line n; stmt s;

Select s.stmt# such that Follows* (s, n) with n=10

or

Select s.stmt# such that Follows* (s, s1) with s1.stmt#=n and n=10

Q30. Find three while loops nested one in another.

while w1, w2, w3;

Select <w1, w2, w3> such that Parent* (w1, w2) and Parent* (w2, w3)

Explanation: notice that the query returns all the instances of three nested while loops in a program, not just the first one that is found.

Q31. Find all assignments with variable “x” at the left-hand side located in some while loop, and that can be reached (in terms of control flow) from program line 60

assign a; while w; prog_line n;

Select a such that Parent (w, a) and Next* (60, n) pattern a(“x”, _) with a.stmt# = n*

Explanation: in a(“x”, _), we refer to variable in the left hand side of the assignment via its name. Patterns are specified using relational notation so they look the same as conditions in such that clause. Think about a node in the AST as a relationship among its children. So assignment is written as assign (variable, expr) and while loop is written as while(variable, _). Patterns are specified in pattern clause. Conditions that follow pattern specification can further constrain patterns to be matched.

Q32. The two queries below yield the same result for SIMPLE programs. Notice also that this might not be the case in other languages.

assign a;

Select a pattern a (“x”, _)

Select a such that Modifies (a, “x”)

Q33. Find all while statements with “x” as a control variable

while w;

Select w pattern w (“x”, _)

Explanation: We use a place holder underscore ‘_’ as statements in the while loop body are not constrained in the query (they are irrelevant to the query).

Q34. Find assignment statements where variable x appears on the left hand side.

assign a;

Select a pattern a (“x”, _)

Q35. Find assignments with expression x*y+z on the right hand side

assign a;

*Select a pattern a (_, “x*y+z”)*

Explanation: To find matching assignments, you draw AST for the expression x*y+z. Any assignment whose right hand side expression matches exactly AST for the expression x*y+z, matches the above pattern. For example, a = x*y+z matches the above pattern, but a = x*y+z + v does not match the pattern.

Q36. Find assignments in which x*y+z is a sub-expression of the expression on the right hand side.

assign a;

*Select a pattern a (_, _“x*y+z”_)*

Explanation: underscores on both sides of the x*y+z indicate that x*y+z may be part of a larger expression. Again, pattern matching is done on AST. For example, a = x*y+z + v matches the above pattern, but a = w+ x*y+z + v does not match the pattern.

Q37. Find all assignments to variable “x” such that value of “x” is subsequently reassigned recursively in an assignment statement that is nested inside two loops.

assign a1, a2; while w1, w2;

Select a2 pattern a1 (“x”, _) and a2 (“x”,_”x”_) such that Affects (a1, a2) and Parent (w2, a2) and Parent* (w1, w2)*

Allowable arguments of relationships in program queries

Relationship arguments are program design entities such as procedure, statement (stmt), while, etc. Naturally, synonyms of suitable design entities can appear as relationship arguments in queries. An argument in a relationship can be as follows:

1. A synonym of a valid design entity. Validation is done according to the definition of a given relationship. For example, Calls (p,q) requires p and q to be synonyms of type ‘procedure’
2. A placeholder ‘_’ (that is free, unconstraint argument) provided it does not lead to ambiguity. Therefore, Modifies (_, “x”) and Uses (_, “x”) are not allowed, as here it is not clear if the ‘_’ refers to a statement or procedure.
3. A character string in quotes, e.g., “xyz” can be used as an argument whenever an instance of the design entity can be identified by that string. When relationship argument is procedure or variable then string is interpreted as its name (should be NAME in quotes).
4. An integer can be used as an argument whenever an instance of the design entity can be identified by integer. In relationship Next and Next*, integer arguments mean program line numbers, and in Modifies, Uses, Parent, Parent*, Follows, Follows*, Affects, Affects* an integer argument means statement number.
5. Synonym of prog_line can be used for arguments of type stmt (and vice-versa). Then the prog_line is interpreted as a statement number.
6. Synonyms of stmt, assign, call, read, print, while and if can appear in place of program lines (and vice-versa). Statement numbers are then interpreted as program lines.
7. Synonym of type assign, call, read, print, while and if can appear in place of type stmt (and vice-versa).

Return arguments

The format of the result returned by PQL queries is summarized below:

Select	Should return

BOOLEAN	TRUE/FALSE
Statement (stmt / read / print / call / while / if / assign) Program line (prog_line)	Statement number
Variable	Name (no need to use “”)
Procedure	Name (no need to use “”)
Constant	Constant value
Tuple (<..., ...>)	Tuple values (e.g. 2 x, 3 z, …)

The returning of tuple results for a query should be done as follows:

- On paper/tutorial/tests:
  - Elements of the tuple separated by space
  - Different tuples separate by comma
  - Example: Select <a, p, s> … returns

3 First 10, 5 Second 4, …

- In AutoTester ():
  - Elements of the tuple separated by space in a string
  - Each tuple string is an element in the list of strings returned by evaluate (TestWrapper)
  - Example: Select <a, p, s> … returns a list of strings

First element: “3 First 10”, second element: “5 Second 4”

If there are no results for a query at all, the returning of results for this query should be done as follows:

- On paper/tutorial/tests:
  - Write “none”
- In AutoTester ():
  - Do not populate the list of results with any values (not even keyword “none”)

Invalid Queries

Query Parser must be able determine whether a given query is valid or invalid.

A query can be invalid syntactically (if it does not follow PQL syntax) and / or semantically (e.g., if the argument of a relationship is not one of the allowable types).

If a query is syntactically invalid, the returning of results for this query should be done as follows:

- On paper/tutorial/tests:
  - Write “Invalid” and explain why
- In AutoTester ():
  - Return empty result

If a query is syntactically valid but semantically invalid, the returning of results for this query should be done as follows:

- On paper/tutorial/tests:
  - Write “Invalid” and explain why
- In AutoTester ():
  - If it is a SELECT BOOLEAN query, return FALSE; otherwise return empty result.
  - Note: It should be possible to determine whether the query is a SELECT BOOLEAN query since it is **syntactically valid**.

For the system testing in the project evaluation, you may assume that the test cases used are all **syntactically** valid. If you are making additional assumptions about the system output for semantically invalid queries, please mention them clearly in your report.

“Meaningless” Queries

Query Processor must be prepared to evaluate any query that is valid (both syntactically and semantically), even if some of these queries may have little use and make little sense.

Here are some example of such queries:

procedure p, q; assign a, a1; while w; variable v; prog_line n1, n2;

(the above declarations apply to all the queries below)

- Select BOOLEAN with 12 = 12

Answer: The result is TRUE since 12 is equal to 12.

- Select BOOLEAN with a.stmt# = 12

Answer: if statement at program line number 12 happens to be an assignment statement, then this the result is TRUE; otherwise, the result is FALSE.

- Select a1 with a.stmt# = 12

Answer: if statement at program line number 12 happens to be an assignment statement, then all the assignment statements in the program are selected; otherwise, the result is nil.

Please note that the same query can be also written in the following way:

Select a1 with 12 = a.stmt#

- Select <a, w> such that Calls (“Second”, “Third”)

Answer: if procedure Second happens to call procedure Third in the program, then a set of pairs including all the combinations of assignment and while statements in the program are selected; otherwise, the result is nil.

- Select <p, q> such that Modifies (a, “y”)

Answer: This query means exactly: Select all the pairs <p, q> such that there exists assignment a which modifies “y”. Though, p and q are not related to assignment a – query is correct. If there is an assignment statement modifying “y”, the result is a set of pairs including all the combinations of procedures in a program.

- Select BOOLEAN such that Calls (_,_)

Answer: If there is a procedure that calls some other procedure in the program, the result is TRUE; otherwise, the result is FALSE.

Format for PQL queries

A complete grammar of PQL is shown above. The following is a general format of PQL queries:

select-cl : declaration* Select result-cl (with-cl | suchthat-cl | pattern-cl)*

The result-cl clause specifies a program view to be produced (i.e., tuples or a BOOLEAN value). In the with-cl clause, we constrain attribute values (e.g., procedure.procName=“Parse”). The suchthat-cl clause, specifies conditions in terms of relationship participation. The patterncl describes code patterns to be searched for.

Notice that a PQL query can contain any number of such that, with and pattern clauses and there is a default “and” between any two consecutive clauses. We can swap clauses without changing the meaning of the query. All the clauses may appear more than one time in a query, in any order:

assign a; while w;

- Select a such that Modifies (a, “x”) and Parent* (w, a) and Next* (1, a)
- Select a such that Parent* (w, a) and Modifies (a, “x”) such that Next* (1, a)
- Select a such that Next* (1, a) and Parent* (w, a) and Modifies (a, “x”)
- Select a pattern a (“x”, _) such that Parent* (w, a) such that Next* (1, a)
- Select a such that Parent* (w, a) and Next* (1, a) pattern a (“x”, _)
- Select a such that Next* (1, a) and Parent* (w, a) pattern a (“x”, _)

However, the following queries are syntactically incorrect. Syntactically, **and** connects clauses of the same type that do not contain the keyword (such that, with or pattern) and should not be used to connect or introduce clauses of different types (check grammar for full syntax):

- Select a such that Parent* (w, a) and Modifies (a, “x”) **and such that** Next* (1, a)
- Select a such that Parent* (w, a) **and pattern** a (“x”, _) such that Next* (1, a)
- Select a such that Parent* (w, a) pattern a (“x”, _) **and** Next* (1, a)