Motivation

Some companies spend ~80% of software budgets on software maintenance. During software maintenance, programmers spend ~50% of the time understanding a program. Therefore, methods and tools that can ease program understanding can cut development costs substantially.

During program maintenance, programmers often try to locate code relevant to the maintenance task at 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 the sub-expression `x * y + z`!
- Which statements affect the value of `x` at statement #120?
- Which statements can be affected if I modify statement #20?

A programmer may need to examine a large amount of code to answer those questions. Doing this by hand may be time-consuming and error-prone.

Static Program Analyzer (SPA)

A **Static Program Analyzer (SPA)** is an interactive tool that automatically answers queries about programs. We will design and implement a SPA for the SIMPLE programming language.

The SPA can be used by users (programmers) in the following way:

1. John, a programmer, is given a task to fix a crashing error in a program.
2. He 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. He starts using the SPA to help him find program statements that cause the crash by entering queries. The SPA evaluates the queries and displays the results.
4. He analyzes query results and examines related sections of the program to locate the source of the error.
5. He finds program statement(s) responsible for an error and modifies the program to fix the error. He can ask the SPA with more queries to examine the possible unwanted ripple effects of the changes.

From the users' point of view, static analysis requires three actions:

1. Enter the source program
2. Enter queries
3. View query results

How It Works

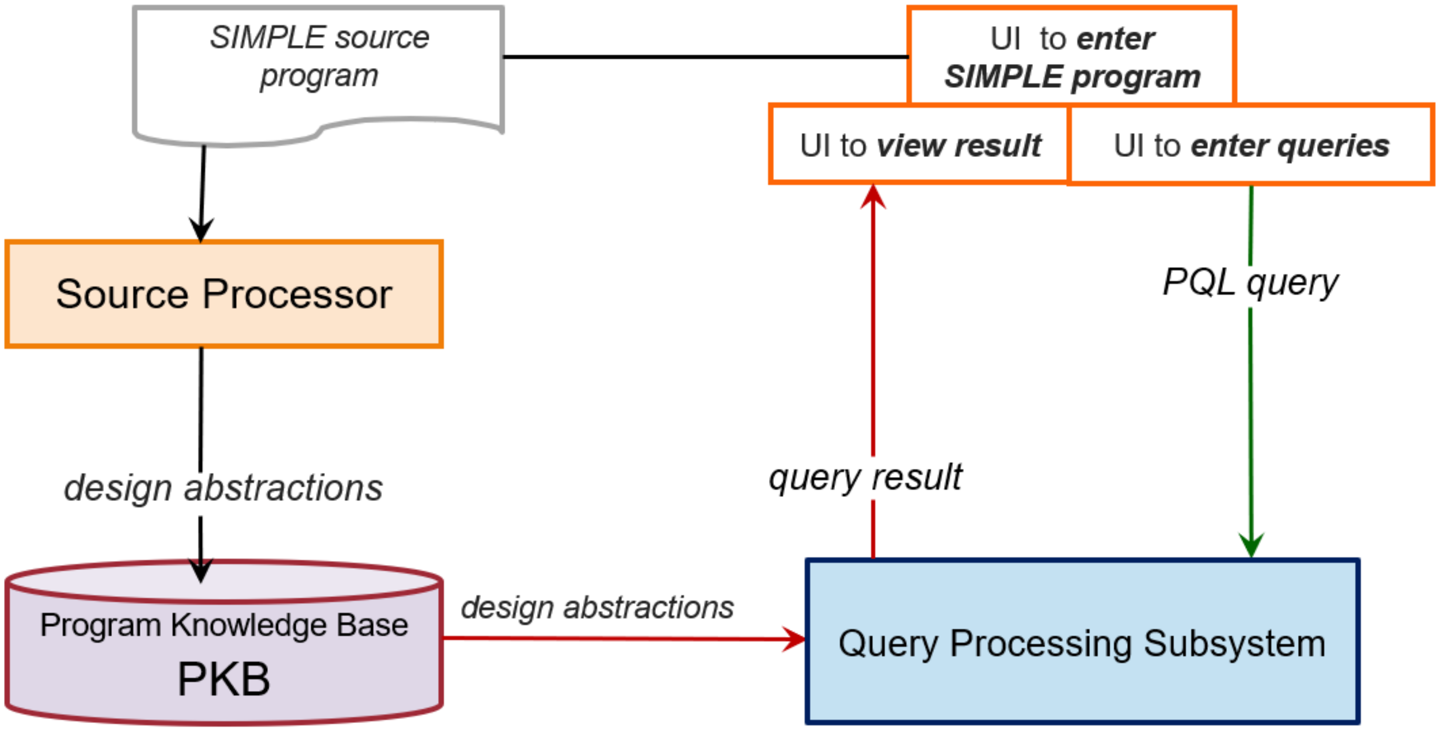
SPA works by:

1. Analyzing a source program and extract relevant program design entities, program design abstractions, Abstract Syntax Tree (AST), and program Control Flow Graph (CFG)
2. Storing the information in a PKB
3. Providing the user with the means to ask queries written in a formal Program Query Language (PQL)
4. Processing the PQL queries based on the information found in the PKB
5. Returning the results to the user

Figure 1 shows the SPA's main components:

- The User Interface (UI) allows users to enter a source program written in SIMPLE.
- The **Source Processor** parses the source program. It extracts information and stores the information in the **PKB**.
- The UI also allows users to input PQL queries.
- The **Query Processing Subsystem** validates and evaluates the queries by making use of the information stored in the PKB.
- The UI then display the query results to the user.

Figure 1: SPA's Main Components



[Generated by MarkBind 4.0.2]

SIMPLE Programming Language

The static analysis is done on a programming language called SIMPLE. It is designed for the purpose of experimenting with SPA techniques and allows students to complete the project in one semester. It is not a language for solving real programming problems. However, it contains all the basic constructs of a programming language for writing meaningful programs.

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

Code 1: Compute average of three integer numbers

```
procedure computeAverage {
    read num1;
    read num2;
    read num3;

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

    print ave;
}
```

Code 2: Print numbers in ascending order

```
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

```
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

```
    procedure main {
01      flag = 0;
02      call computeCentroid;
03      call printResults;
    }
    procedure readPoint {
04      read x;
05      read y;
    }
    procedure printResults {
06      print flag;
07      print cenX;
08      print cenY;
09      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;
22          cenY = cenY / count;
```

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

General Information

The following are some general information:

- **Program**: Made up of one or more procedures
- **Procedure**: Non-empty list of statements with no parameters, nesting nor recursion
- **Variable**: Unique names, global scope, integer type, and does not have declarations
- **Statement**: Assignments, while loops, if-then-else statements, call statements, print statements or read statements
- **Condition**: Boolean expressions containing operators
- **Operator**: `+, -, *, /, %, <, >`, etc.
- No arrays, no pointers

Program statements are of the following types:

- Call statements
  - Invokes a procedure by its name.
  - E.g. `call readPoint;`
- Assignment statement
  - Assigns the results of the expression on the right to the variable on the left.
  - E.g. `x = a + 2 * b;`
- While statement
  - Executes statements in the body until the condition becomes false.
  - E.g. `while (i > 0) { ... }`
- If statement
  - Executes statements in the `then` branch if condition is true; otherwise executes statements in the `else` branch.
  - The `else` branch is mandatory.
  - E.g. `if (i > 0) then { ... } else { ... }`
- Read statement
  - Reads the value from standard input (console) and assigns it to the variable.
  - It is similar to `scanf` and `cin` in C/C++, and does not read the value from the variable.
  - E.g. `read x;`
- Print output
  - Outputs the value of the variable to standard output (console).
  - E.g. `print x;`

Concrete Syntax Grammar (CSG)

A source program written in SIMPLE is syntactically valid 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 (keywords) and non-terminals.

- Terminals (keywords) 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.

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
NZDIGIT: 1-9                           - non-zero digit
NAME: LETTER (LETTER | DIGIT)*    - procedure names and variables are strings of
                                    letters, and digits, starting with a letter
INTEGER : 0 | NZDIGIT ( DIGIT )*       - Constants are sequence of digits with no leading zero
```

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
```

Note:

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. Whitespaces (including multiple spaces, 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 character streams:
   - `x+y`
   - `x + y`
   - `x +y`
3. Procedure names, variable names and terminals can all be the same.
   - E.g. `read read; read = read + 1;`
4. Logical expressions may only appear as the conditions for while / if statements and they are fully bracketed.
5. There are no boolean values (`true` / `false`).
6. 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 ')'

Other Rules

A syntactically valid source program written in SIMPLE is semantically invalid if it violates rules that cannot be captured by the CSG.

The following are rules that cannot be captured by CSG:

1. A program cannot have two procedures with the same name.
2. A procedure cannot call a non-existing procedure.
3. Recursive and cyclic calls are not allowed.
    - E.g. Procedure A calls procedure B, procedure B calls procedure C, and procedure C calls procedure A.
    - E.g. Procedure A calls procedure A.

Statement Numbers

The statements in a SIMPLE program are indexed for easy referencing and are not part of the syntax. 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.

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.

An example of how statements are numbered is shown in Code 4.

Abstract Syntax Grammar (ASG)

A CSG defines the language and its vocabulary. However, you may need to use abstractions to define a language. In that case, the ASG 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 can be multiple CSGs for the same ASG.

Meta symbols:

```
a+        - repetition 1 or more times of a
a | b     - a or b
```

Lexical tokens:

```
LETTER: A-Z | a-z               - capital or small letter
DIGIT: 0-9
NZDIGIT: 1-9
NAME: LETTER (LETTER | DIGIT)*
INTEGER: 0 | NZDIGIT ( 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 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
```

[Generated by MarkBind 4.0.2]

Abstract Syntax Tree (AST)

An 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 two branches.

There are core requirements when representing information in the AST:

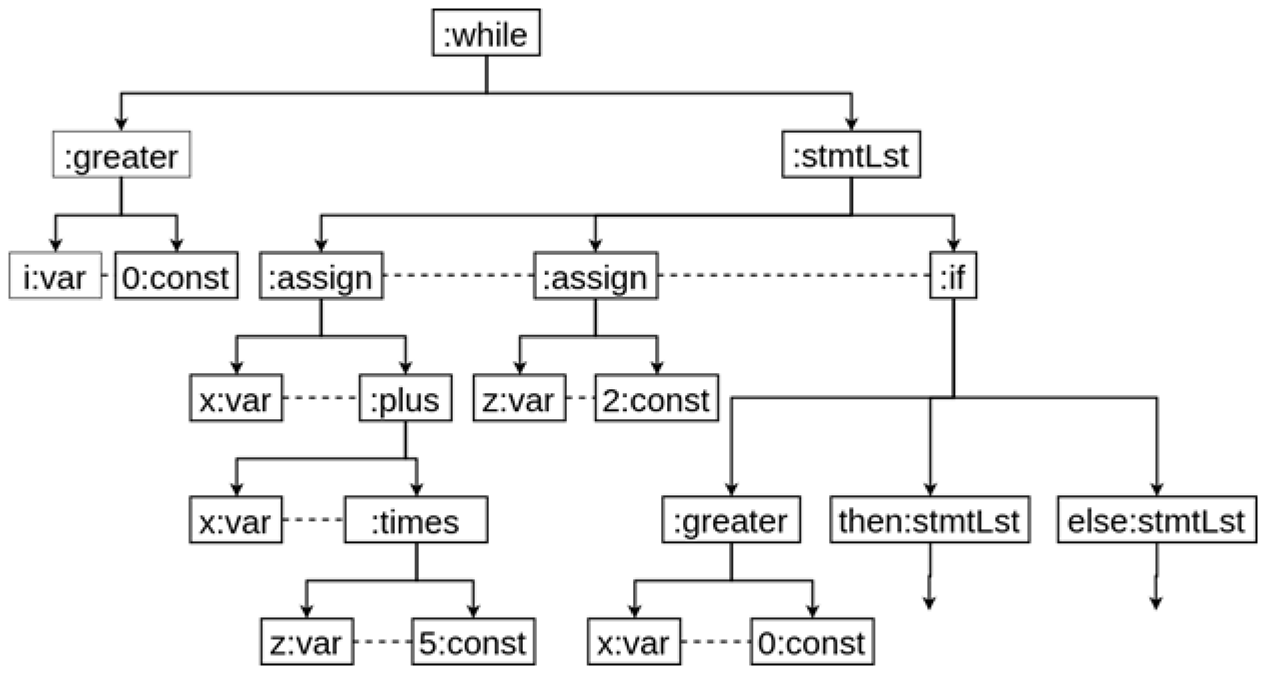
- The order of executable statements must be explicitly represented and well defined.
- Left and right components of binary operations must be stored and correctly identified.
- Identifiers and their assigned values must be stored for assignment statements.

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. 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



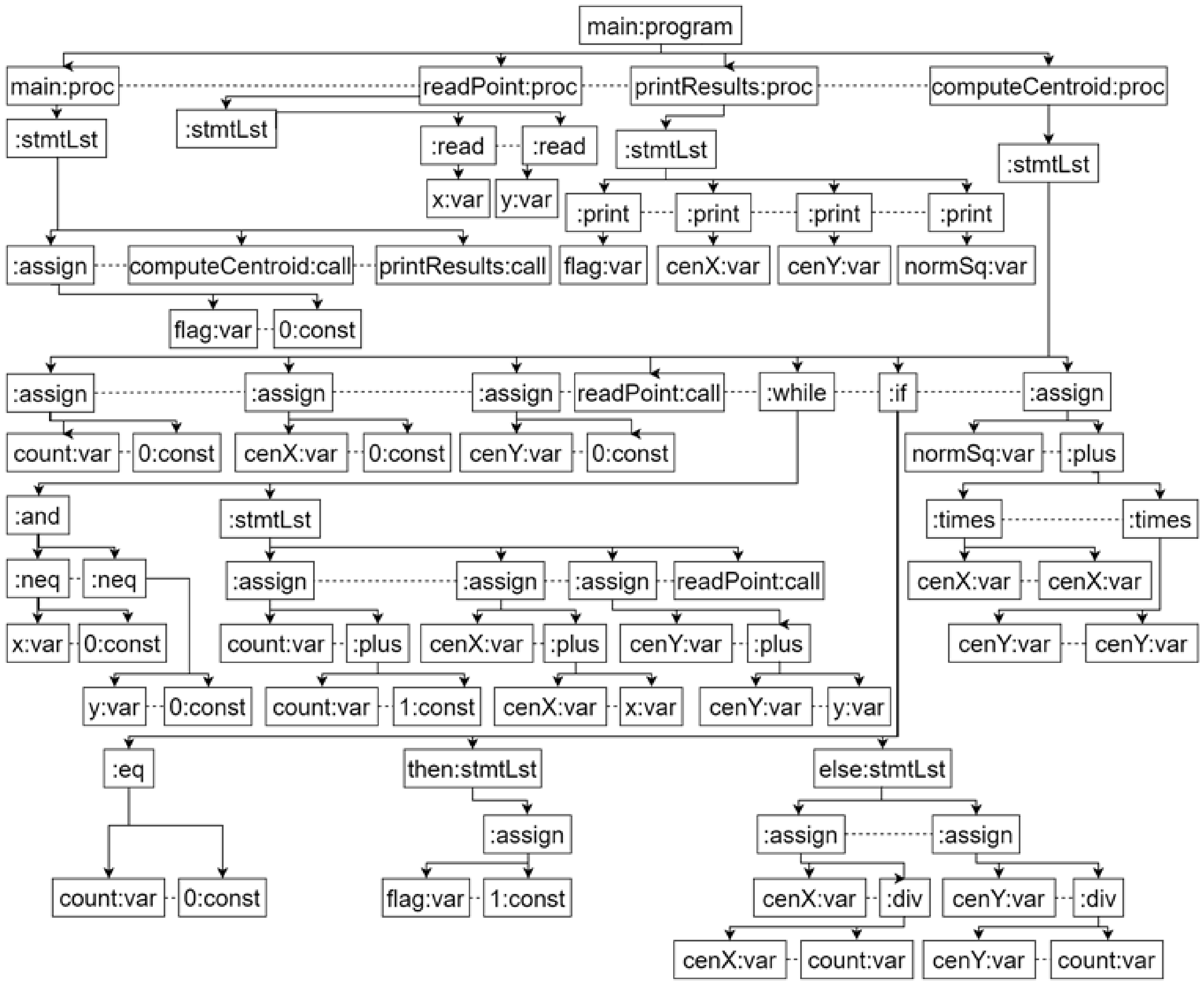
Formally, AST nodes are labeled with values (procedure names, variable names, constants etc.) and syntactic types. These are placed before after `:` respectively. For example, node `readPoint:call` represents a call to the procedure `readPoint`, and node `cenX:variable` represents a variable `cenX`.

Figure 3 shows a full AST for program Main in Code 4 example.

Code 4: Program with multiple procedures

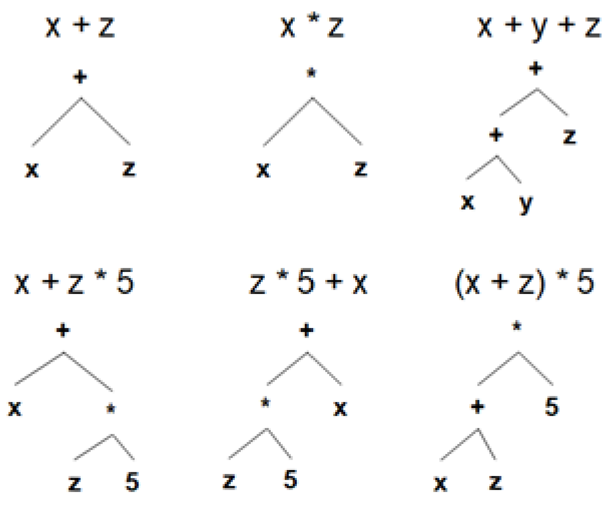
```
    procedure main {
01      flag = 0;
02      call computeCentroid;
03      call printResults;
    }
    procedure readPoint {
04      read x;
05      read y;
    }
    procedure printResults {
06      print flag;
07      print cenX;
08      print cenY;
09      print normSq;
    }
    procedure 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;
22          cenY = cenY / count;
        }
23      normSq = cenX * cenX + cenY * cenY;
    }
```

Figure 3: Full AST for Code 4



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 (see AST for `x+y+z` below).

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, 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.

[Generated by MarkBind 4.0.2]

Design Entities

The design entities characterize and help refer to different abstraction from the program. A list of design entities for SIMPLE programming language are as follows:

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

[Generated by MarkBind 4.0.2]

Design Abstractions

Program design abstractions are relationships among program design entities.

In this section, we will use Code 5 to explain the design abstractions. For simplicity, procedures `main`, `readPoint` and `printResults` are excluded from statement numbering.

Code 5

```
      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 {
01        count = 0;
02        cenX = 0;
03        cenY = 0;
04        call readPoint;
05        while ((x != 0) && (y != 0)) {
06            count = count + 1;
07            cenX = cenX + x;
08            cenY = cenY + y;
09            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;
      }
```

Basic Design Abstractions

The table shows a summary of the design abstractions discussed in this section.

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

Follows / Follows*

Definition:

```
For any statements s1 and s2:

Follows(s1, s2) holds if they are 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`.

Examples:

In procedure `computeCentroid` (Code 5), 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 (do not hold):

- `Follows (5, 6)`
- `Follows (9, 10)`

- `Follows (11, 12)`
- `Follows* (12, 14)`

Notes:

- For `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.

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

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`.

Examples:

In procedure `computeCentroid` (Code 5), the following relationships hold (are true):

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

The following relationships are false (do not hold):

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

Notes:

- As in the case of `Follows`, statement number `5` refers to the whole while-statement, and statement number `10` refers to the whole if-statement. The first statement should be a container statement, while the second statement should be nested inside the first statement.)

Uses

Definition:

```
For any variable v,
        assignment a,
        print stmt pn,
        container stmt (if or while) s,
        procedure call c,
        procedure p:

Uses(a, v) holds if v appears on the right hand side of a.

Uses(pn, v) holds if variable v appears in pn.

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.

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.

Uses(c, v) is defined in the same way as Uses(p, v).
```

Examples:

In the program (Code 5), the following relationships hold (are true):

- `Uses (7, "x")`
- `Uses (10, "count")`
- `Uses (10, "cenX")`
- `Uses ("main", "cenX")`
- `Uses ("main", "flag")`
- `Uses ("computeCentroid", "x")`

The following relationships are false (do not hold):

- `Uses (3, "count")`
- `Uses (10, "flag")`
- `Uses (9, "y")`

Notes:

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).

- E.g. `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`.

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`.

- E.g. Statement 10 contains statement 12 that uses `cenX`; hence 10 uses `cenX`.

Modifies

Definition:

```
For any variable v,
        assignment a,
        read stmt re,
        container stmt (if or while) s,
        procedure call c,
        procedure p:

Modifies(a, v) holds if variable v appears on the left hand side of a.

Modifies(re, v) holds if variable v appears in re.

Modifies(s, v) holds if there is a statement s1 in the container
               such that Modifies(s1, v) holds.

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.

Modifies(c, v) is defined in the same way as Modifies(p, v).
```

Examples:

In the program (Code 5), the following relationships hold (are true):

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

The following relationships are false (do not hold):

- `Modifies (5, "flag")`
- `Modifies ("printResults", "normSq")`

Notes:

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).

- E.g. `y` is modified in the call statement `call computeCentroid;` from `main` because procedure `computeCentroid` calls procedure `readPoint` that modifies `y` in a read statement.

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).

- E.g. Statement 5 contains statement 9 that modifies `x` (in procedure `readPoint` read statement); hence 5 modifies `x`.

[Generated by MarkBind 4.0.2]

Program Query Language (PQL)

PQL queries are expressed in terms of program design models (entities and abstractions).

PQL queries references the following:

- Design entities
  - E.g. procedure, variable, assign, etc.
- Attributes
  - E.g. `procedure.procName` or `variable.varName`
- Relationships
  - E.g. `Modifies (procedure, variable)`
- Syntactic patterns
  - E.g. `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 (e.g. procedure named `main`, statement number `35`, or variable named `x`).

Basic PQL

Basic queries contain:

- **Declaration of synonyms** to be used in the query
  - E.g. `procedure p; variable v;` where `p` is a procedure entity and `v` is a variable entity
- **Select clause** that specifies a single return value for query result
- At most one **such-that clause** that constrains the results in terms of relationships
- At most one **pattern clause** that constrains results in terms of syntactic patterns

There is an implicit `and` operator between clauses. Query results must make sure that they satisfy all clauses.

Basic PQL Grammar

A query written in PQL is syntactically valid if it follows all the defined language rules.

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
```

Lexical tokens:

```
LETTER: A-Z | a-z                     - capital or small letter
DIGIT: 0-9
NZDIGIT: 1-9                          - non-zero digit
IDENT : LETTER ( LETTER | DIGIT )*
NAME : LETTER ( LETTER | DIGIT )*
INTEGER : 0 | NZDIGIT ( DIGIT )*      - no leading zero

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 ')'

pattern-cl : 'pattern' syn-assign '(' entRef ',' expression-spec ')'
expression-spec :  '"' expr'"' | '_' '"' expr '"' '_' | '_'

expr: expr '+' term | expr '-' term | term
term: term '*' factor | term '/' factor | term '%' factor | factor
factor: var_name | const_value | '(' expr ')'

syn-assign : IDENT

var_name: NAME
const_value : INTEGER
```

Notes:

1. PQL is case-sensitive. The grammar shows the accepted casing for the keywords of the language. Due to case-sensitivity, synonyms "abc" and "Abc" are two different synonyms.
2. Whitespaces (including multiple spaces, tabs, or no spaces) can be used freely in PQL. For example, tokenizer should recognize three tokens `x`, `+` and `y` in any of the following character streams:
    - `x+y`
    - `x + y`
    - `x +y`
3. Synonym names and terminals can all be the same.
    - E.g. `assign pattern; variable Select, assign; Select Select pattern pattern(assign, _)`

Other Rules

A syntactically valid query is semantically invalid if it violates rules that cannot be captured by the language rules.

The following are rules that are not captured by the grammar:

1. A synonym name can only be declared once.
2. All the synonyms used in clauses must be declared exactly once.
3. `syn-assign` must be declared as a synonym of an assignment (design entity `assign`).
4. The first argument for `Modifies` and `Uses` cannot be `_`, as it is unclear whether `_` refers to a statement or procedure.
5. Synonyms of design entities can appear as relationship arguments, and should match the design entity defined for the relationship.
    - E.g. For `Parent(arg1, arg2)`, if `arg1` is a synonym, then `arg1` must be a statement synonym, or a subtype of a statement synonym (read, print, assign, if, while, call).
    - E.g. For `Modifies(arg1, arg2)`, if `arg2` is a synonym, then `arg2` must be a variable synonym.
6. Similarly, synonyms of design entities can appear as arguments in pattern clauses, and should match the design entity defined for pattern clauses.
    - E.g. For clause `pattern a (arg1, _)`, if `arg1` is a synonym, then `arg1` must be a variable synonym.

[Generated by MarkBind 4.0.2]

Relationships in Such-that Clauses

There are relationships (design abstractions) to be considered.

Within each relationship used in such-that clauses, there are a few arguments that are allowed in each relationship.

- Synonyms of program design entities
- Wildcard `_`
- Character strings / Integers

Synonyms of design entities can appear as relationship arguments, and should match the design entity defined for the relationship.

- E.g. For `Parent(arg1, arg2)`, if `arg1` is a synonym, then `arg1` must be a statement synonym, or a subtype of a statement synonym (read, print, assign, if, while, call).
- E.g. For `Modifies(arg1, arg2)`, if `arg2` is a synonym, then `arg2` must be a variable synonym.

Assign, read, print, if, while, call synonyms can appear in place of statement synonyms, and vice-versa, and the query will still be semantically valid. However, there may not be results for some relationships. Let us use `Parent` as illustration:

- E.g. For `Parent(w, _)`, where `w` is a while synonym: Semantically valid, and may have answers.
- E.g. For `Parent(a, _)`, where `a` is a assign synonym: Semantically valid, but will not have any answers (since assignment is not a container statement).
- E.g. For `Parent(proc, _)`, where `proc` is a procedure synonym: Semantically invalid because procedure synonym cannot be substituted as statement synonym.

In addition, wildcard `_` can appear as relationship arguments.

- The only exception would be for `Modifies(arg1, arg2)` and `Uses(arg1, arg2)`, where `arg1` cannot be `_`, as it leads to ambiguity whether `_` refers to a statement or procedure.

A character string in quotes (e.g., `"xyz"`) can appear as relationship arguments if an instance of the design entity can be identified by that string.

- E.g. For `Modifies(arg1, arg2)`, if `arg1` is `"xyz"`, then it will be interpreted as a procedure.
- E.g. Similarly, for `Modifies(arg1, arg2)`, if `arg2` is `"xyz"`, then it will be interpreted as a variable.

An integer (e.g. `23`) can also appear as relationship arguments if an instance of the design entity can be identified by the integer.

- E.g. For `Modifies(arg1, arg2)`, if `arg1` is `23`, then it will be interpreted as a statement.

Do refer to example queries with one such-that clause.

[Generated by MarkBind 4.0.2]

Assign Pattern Clauses

The assign pattern clause comprises of an assign synonym, followed by 2 arguments.

The following are allowed as the first argument:

- Variable synonyms
- Wildcard `_`
- Character strings

The following are allowed as the second argument:

- Wildcard `_`
- Expression for exact match (e.g. `"x*y"`)
- Expression for partial match (e.g. `_"x*y"_`)

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

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

The evaluation of the following queries would result in:

`assign a;`

PQL query	Returns	Explanation
`Select a pattern a ( _ , "v + x * y + z * t")`	1	Exact pattern match
`Select a pattern a ( _ , "v")`	none	stmt #1 contains other expression terms other than v
`Select a pattern a ( _ , _"v"_)`	1	stmt #1 contains `v` as part of the expression on the RHS
`Select a pattern a ( _ , _"x*y"_)`	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"_)`	1	stmt #1 contains `v+x*y` as a sub-expression on the RHS

PQL query	Returns	Explanation
`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"_)`	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.

Do refer to example queries with one pattern clause.

[Generated by MarkBind 4.0.2]

Example Queries

In this section, we will use Code 5 to answer some queries.

For simplicity, procedures `main`, `readPoint` and `printResults` are excluded from statement numbering.

Code 5

```
     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 {
01       count = 0;
02       cenX = 0;
03       cenY = 0;
04       call readPoint;
05       while ((x != 0) && (y != 0)) {
06           count = count + 1;
07           cenX = cenX + x;
08           cenY = cenY + y;
09           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;
     }
```

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 5, the results of evaluating the query are: `main, readPoint, printResults, computeCentroid`

The order of the names of the procedures does not matter. Hence, another correct result is: `printResults, main, readPoint, computeCentroid`

Q2. What are the variables in the program?

```
variable v;
Select v
```

The query returns all variable names: `flag, count, cenX, cenY, x, y, normSq`

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: `7, 8, 9`

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

```
variable v;
Select v such that Modifies (6, v)
```

Answer: `count`

Q5. Which variables are used in assignment 14?

```
variable v;
Select v such that Uses (14, v)
```

Answer: `cenX, cenY`

Q6. Which procedures modify variable "x"?

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

Answer: `main, computeCentroid, readPoint`

Q7. Find assignments within a loop.

```
assign a; while w;
Select a such that Parent* (w, a)
```

Answer: `6, 7, 8`

Q8. Which is the parent of statement #7?

```
stmt s;
Select s such that Parent (s, 7)
```

Answer: `5`

Queries with one pattern clause

With reference to Code 5, 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: `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: `14`

Same query can be written using a different synonym name:

```
assign newa;
Select newa pattern newa ( "normSq" , _"cenX * cenX"_)
```

Answer: `14`

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: `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: `6, 7, 8, 12, 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;
Select a pattern a ("x", _) such that Uses (a, "x")
```

or

```
assign a;
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

```
assign a; while w;
Select a pattern a ("count", _) such that Parent* (w, a)
```

Answer: `6`

[Generated by MarkBind 4.0.2]

SPA Behaviour

It is crucial to take note that SPA should **terminate**, and not answer any queries if SPA detects an invalid source program (both syntactically and semantically).

Otherwise, queries should be answered according to the required format of results.

Format of Results

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

Select	Should return
Statement (stmt/read/print/call/while/if/assign)	In SPA implementation, return an array of statement numbers in string format (no need to use "").
Variable	In SPA implementation, return an array of variable names in string format (no need to use "").
Procedure	In SPA implementation, return an array of procedure names in string format (no need to use "").
Constant	In SPA implementation, return an array of constant values in string format (no need to use "").

Array should not be populated with any string elements if there are no answers to the query (not even `none`).

Results should not contain duplicates.

For **syntactically invalid queries**, populate the list of results with one value `SyntaxError`. On paper, state `SyntaxError` as your answer, together with the reason why.

Examples of syntactically invalid queries are as follows:

```
variable v;
Select v;
```

```
variable v;
select v
```

For **semantically invalid queries**, populate the list of results with one value `SemanticError`. On paper, state `SemanticError` as your answer, together with the reason why.

Examples of semantically invalid queries are as follows:

```
variable v;
Select v such that Uses(_, v)
```

```
stmt s;
Select s pattern s(_, _)
```

Note that a semantically invalid query must be syntactically valid.

[Generated by MarkBind 4.0.2]

Q: When do we need a whitespace between tokens, and when do we not need a whitespace?

A: Between two alphanumeric tokens, there must be at least one whitespace in between the tokens. E.g. `read x1` is valid as two tokens (perhaps part of a read statement with a keyword `read` and a variable name), while `readx1` should be treated as one token (such as a variable name).

Between one alphanumeric token and one non-alphanumeric token, or between two non-alphanumeric tokens, there need not be a whitespace in between the tokens. E.g. Both `x1;` and `x1 ;` can be read as two tokens `x1` and `;`. Both `;}` and `; }` can be read as two tokens `;` and `}`.

Q: Does every semicolon needs to be followed by a newline in SIMPLE? For example, is this valid?

```
procedure A {
    x = 1; y = 3; z = 5;
}
```

A: No, every semicolon do not need to be followed by a newline in SIMPLE. In this case, the program is valid. In fact, statement 1 refers to "x = 1;", statement 2 refers to "y = 3;" and so on. The statement number is incremented every time a new statement is encountered.

Q: Does every semicolon needs to be followed by a newline in PQL? For example, is this valid?

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

A: No, semicolons do not need to be followed by a newline in PQL queries. However, bear in mind that due to AutoTester limitations, queries (including the declaration itself), can only span 2 lines. This means the query can only look like:

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

or

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

Q: Can a statement or query have multiple whitespace spread over multiple lines? For example, is this valid?

```
procedure A {
    x =
          1      +
  y
;
}
```

```
assign a;
Select a pattern a(_, "1                +   y")
```

A: Yes, it is valid. You should ensure that your parser is able to parse this by recognizing all types of whitespace. In this case, statement 1 refers to "x = 1 + y;", and the query will be valid and returns the answer 1.

Q: According to SIMPLE language rules, procedures have no nesting. Does "no nesting" mean that procedures cannot be defined within other procedures, or does it mean something else?

A: Yes. For example, this is an invalid program.

```
procedure A {
    procedure B {
        read num 1; } }
```

Q: Is it grammatically correct to print a procedure? Can we use keywords for procedure names and variable names?

```
procedure procedure {
    print procedure; }
```

A: Concrete Syntax grammar (CSG) for SIMPLE program states that the print statement has the syntax of "print var_name". In this case, "print procedure" is valid and it will be parsed as a variable name instead of a procedure name. An additional note is that "procedure names can be the same as variable names" as stated in the grammar rules as well. Note that keywords can also be used as variable names and procedure names as well.

Q: Is it grammatically correct to use keywords for synonym names? For example:

```
assign pattern; variable Select;
Select Select pattern pattern(Select, _)
```

A: Yes, it is valid and your parser should not fail to parse this query.

Q: What should I do if the system faces an invalid SIMPLE source code?

A: Terminate your program without evaluating any queries. You may also choose to display helpful messages in the CLI for your ease of debugging.

Q: For Parent*(s1, s2), may I clarify the precise meaning for "nested"? For example, can I say stmt 2 is directly nested inside ProcB, and indirectly nested inside ProcA?

```
procedure ProcA {
1.   call ProcB; }
procedure ProcB {
2.   read a;
3.   print a; }
```

A: No. In full SPA terminology, the word nested refers to the fact that s1 is a container statement and s2 appears directly (or indirectly) in the statement list of s1.

Q: What is the maximum size / length of digits / characters for INTEGER and NAME are?

A: You can make reasonable assumptions about them.

Q: What is the maximum level of nesting in SIMPLE?

A: You can make reasonable assumptions about them.

Q: Can we use an SQL database to answer queries?

A: No.

[Generated by MarkBind 4.0.2]

Control Flow Graph (CFG)

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, which contains statement 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 6 is shown in Figure 5.

For simplicity, procedures `First` and `Third` are excluded from statement numbering.

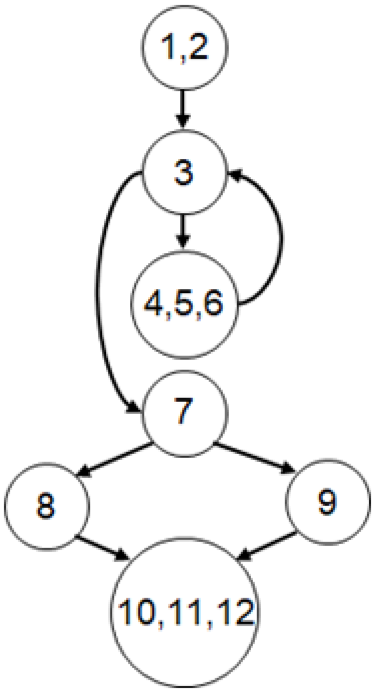
Code 6

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

        procedure Second {
01          x = 0;
02          i = 5;
03          while (i!=0) {
04              x = x + 2*y;
05              call Third;
06              i = i - 1; }
07          if (x==1) then {
08              x = x+1; }
            else {
09              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 6



Do note that:

1. There is one CFG per procedure.
2. The while statement (head) should be in a separate node from other statements in the loop (body).
3. Dummy nodes can be used (but not excessively) to show that
    - if statements in the CFG have a diamond shape
    - while loops have a loop shape
    - procedure ends

[Generated by MarkBind 4.0.2]

Design Abstractions

In this section, we will use Code 6 to explain more design abstractions.

For simplicity, procedures `First` and `Third` are excluded from statement numbering.

Code 6

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

        procedure Second {
01          x = 0;
02          i = 5;
03          while (i!=0) {
04              x = x + 2*y;
05              call Third;
06              i = i - 1; }
07          if (x==1) then {
08              x = x+1; }
            else {
09              z = 1; }
10          z = z + x + i;
11          y = z + 2;
12          x = x * y + z; }

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

Advanced Design Abstractions

The table shows a summary of the design abstractions discussed in this section.

Design Abstraction

Design Abstraction	Relationship between
Calls / Calls*	Procedures

Design Abstraction	Relationship between
Next / Next*	Statements
Affects / Affects*	Assignments

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
```

`Calls*` is the transitive closure of `Calls`.

Examples:

In Code 6, the following relationships hold (are true):

- `Calls ("First", "Second")`
- `Calls ("Second", "Third")`
- `Calls* ("First", "Second")`
- `Calls* ("First", "Third")`

In contrast, the following relationships are false (do not hold):

- `Calls ("First", "Third")`
- `Calls ("Second", "First")`
- `Calls* ("Second", "First")`

Next / Next*

Definition:

```
For two statements s1 and s2 in the same procedure:

Next(s1, s2) holds if s2 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.
```

`Next*` is the transitive closure of `Next`.

Examples:

In Code 6, 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)`

Notes:

- The `Next/*` relationship is determined by the control flow, while `Follows/*` is determined by the structural location of a statement.
- When discussing `Next/*` for container statements, the statement number refers to the evaluation of the conditional expression, rather than the whole body when discussing `Follows/*`.
  - E.g. In Code 6, statement 3 refers to `while (i!=0)` and statement 7 refers to `if (x==1)`.

Affects

Definition:

```
For two statements s1 and s2 in the same procedure:
```

```
Affects(s1, s2) holds if:
    - s1 and s2 are assignment statements,
    - s1 modifies a variable v which is used in s2 and
    - there is a control flow path from s1 to s2 on such that v is not modified
      in any assignment, read, or procedure call statement on that path
```

Examples:

In Code 6, 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)`

In particular, `Affects (1, 12)` holds because:

- Variable `x` is modified in statement `1`,
- Variable `x` is used in statement `12`, and
- There is a path in the CFG (`1 > 2 > 3 > 7 > 9 > 10 > 11 > 12`) on which variable `x` is not modified by any assignment, read, or procedure call.

The following relationships are false (do not hold):

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

In particular, `Affects (9, 12)` does not hold because:

- Variable `z` is modified by assignment `10` that is on any control flow path between assignments `9` and `12`.

Notes:

- The `Affects` relationship models data flows in a program.

- The `Affects` relationship 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`.

More examples:

Code 7

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

In Code 7, `Affects (1, 5)` is true because variable `x` is modified in statement `1` and used in statement `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.

Code 8

```
procedure p {
1.     x = a;
2.     call q;
3.     v = x; }
```

In Code 8, if `Modifies ("q", "x")` holds, then `Affects (1, 3)` does not hold.

If `Modifies ("q", "x")` does not hold, then `Affects (1, 3)` holds.

Code 9

```
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 9,

- `Affects (1, 5)` and `Affects (2, 5)` do not hold as both variable `x` and `y` are modified in procedure `q`.
  - Even though modification of variable `y` in procedure `q` is only conditional, we will still use the definition of `Modifies` for procedures. That is, `Modifies(4, "q")` holds.
- `Affects (3, 10)` do not hold because assignments `3` and `10` are in different procedures.

Code 10

```
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; } }
```

In Code 10, `Affects (1, 3)` is false because variable `x` is modified in statement `1` and used in statement `3`, but the only paths from 1 to 3 (1 > 2 > 3) contains a call statement `2` that modifies `x` according to the definition of `Modifies` for procedure calls.

Code 11

```
procedure p {
1.     x = a;
2.     read x;
3.     v = x; }
```

In Code 11, `Affects (1, 3)` does not hold because read statement `2` modifies variable `x` on the paths from statement `1` to `3`.

[Generated by MarkBind 4.0.2]

Advanced PQL

Queries now contain:

- **Declaration of synonyms** to be used in the query
  - E.g. `procedure p; variable v;` where `p` is a procedure entity and `v` is a variable entity
- **Select clause** that specifies either:
  - a single return value
  - **[NEW]** multiple return values (tuples)
  - **[NEW]** BOOLEAN value
- Any number of **such-that clauses** that constrains the results in terms of relationships
- Any number of **pattern clauses** that constrains results in terms of syntactic patterns
- **[NEW]** Any number of **with clauses** that constrains results in terms of attribute values
- **[NEW]** `not` operator that negates the results for a single clause

There is an implicit `and` operator between clauses. Query results must make sure that they satisfy all clauses.

Advanced PQL Grammar

A query written in PQL is syntactically valid if it follows all the defined language rules.

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
```

Lexical tokens:

```
LETTER: A-Z | a-z                     - capital or small letter
DIGIT: 0-9
NZDIGIT: 1-9                          - non-zero digit
IDENT : LETTER ( LETTER | DIGIT )*
NAME : LETTER ( LETTER | DIGIT )*
INTEGER : 0 | NZDIGIT ( DIGIT )*      - no leading zero [Updated: 27th Feb]

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

elem : synonym | attrRef
```

```
attrName : 'procName'| 'varName' | 'value' | 'stmt#'
design-entity : 'stmt' | 'read' | 'print' | 'call' | 'while' | 'if' | 'assign' |
                'variable' | 'constant' | 'procedure'
```

Grammar rules:

```
select-cl : declaration* 'Select' result-cl ( suchthat-cl | pattern-cl | with-cl)*
declaration : design-entity synonym (',' synonym)* ';'

result-cl : tuple | 'BOOLEAN'
tuple : elem | '<' elem ( ',' elem )* '>'


suchthat-cl : 'such' 'that' relCond
pattern-cl : 'pattern' patternCond
with-cl : 'with' attrCond

relCond : [ 'not' ] relRef ( 'and' [ 'not' ] relRef )*
relRef: Follows | FollowsT | Parent | ParentT | UsesS | UsesP | ModifiesS |
        ModifiesP | Calls | CallsT | Next | NextT | Affects

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 ')'

Calls : 'Calls' '(' entRef ',' entRef ')'
CallsT : 'Calls*' '(' entRef ',' entRef ')'

Next : 'Next' '(' stmtRef ',' stmtRef ')'
NextT : 'Next*' '(' stmtRef ',' stmtRef ')'

Affects : 'Affects' '(' stmtRef ',' stmtRef ')'


patternCond : [ 'not' ] pattern ( 'and' [ 'not' ] pattern )*
pattern : assign | while | if

assign: syn-assign '(' entRef ',' expression-spec ')'
```

```
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

while : syn-while '(' entRef ',' '_' ')'
if : syn-if '(' entRef ',' '_' ',' '_' ')'

syn-assign : IDENT
syn-while : IDENT
syn-if : IDENT

attrCond : [ 'not' ] attrCompare ( 'and' [ 'not' ] attrCompare )*
attrCompare : ref '=' ref
ref : '"' IDENT '"' | INTEGER | attrRef
attrRef : synonym '.' attrName
```

Notes:

1. PQL is case-sensitive. The grammar shows the accepted casing for the keywords of the language. Due to case-sensitivity, synonyms "abc" and "Abc" are two different synonyms.
2. Whitespaces (including multiple spaces, tabs, or no spaces) can be used freely in PQL. For example, tokenizer should recognize three tokens `x`, `+` and `y` in any of the following character streams:
   - `x+y`
   - `x + y`
   - `x +y`
3. Synonym names and terminals can all be the same.
   - E.g. `assign pattern; variable Select, assign; Select Select pattern pattern(assign, _)`

Other Rules

A syntactically valid query is semantically invalid if it violates rules that cannot be captured by the language rules.

The following are rules that are not captured by the grammar:

1. Rules stated in Basic SPA requirements still holds.
2. `syn-while` must be declared as a synonym of an while-statement (design entity `while`).
3. `syn-if` must be declared as a synonym of an if-statement (design entity `if`).

4. For `attrCompare`, the two `ref` comparison must be of the same type (both `NAME`, or both `INTEGER`).
5. For `attrRef`, the `attrName` must be of acceptable attribute of `synonym` as stated in With Clause Discussion.

In addition, if `BOOLEAN` is declared as a synonym in a PQL query, this declaration takes *precedence*, i.e. we can no longer get a boolean query result for that PQL query.

- E.g. `stmt BOOLEAN; Select BOOLEAN` gives the list of statements in the SIMPLE source.

[Generated by MarkBind 4.0.2]

Select Clause

The Select clause can now specify the following results to be returned:

1. Single return values (discussed in Basic SPA requirements).
2. Multiple return values
3. BOOLEAN value

For multiple return values, the query reports any results for which there exists a combination of synonym instances satisfying all the conditions specified in such-that, pattern and with clauses.

For BOOLEAN, the query reports `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;

`FALSE` otherwise.

Queries should adhere to format of results introduced in Basic SPA requirements and Advanced SPA requirements.

[Generated by MarkBind 4.0.2]

Such-that Clauses

There are new design abstractions to be considered as discussed in previous sections.

[Generated by MarkBind 4.0.2]

Pattern Clauses

There are now three types of pattern clauses:

1. Assign pattern clauses (discussed in Basic SPA requirements)
2. While pattern clauses
3. If pattern clauses

In while and if patterns, we are looking at matching control variables. A control variable is a variable that is used in the conditional expression of a container statement.

While pattern and if pattern has 2 arguments and 3 arguments within the parenthesis respectively.

- Similar to assign pattern, the first argument within the parenthesis can only be:
  - variable synonym
  - wildcard `_`
  - variable name in quotes
- The second (and third) argument can only be a wildcard `_`.

Example:

Code 13

```
procedure p {
1.  if (1 == 2) then {
2.      while (3 == 4) {
3.          x = 1; } }
    else {
4.      if (x == 1) then {
5.          while (y == x) {
6.              z = 2; } }
        else {
7.          y = 3; } } } }
```

The following queries will yield these answers:

`if ifs; while w;`

- `Select ifs pattern ifs(_,_,_)` will return `4`, as only stmt `4` is a if statement uses variable(s) in conditional expression
- `Select w pattern w(_,_)` will return `5` as only stmt 5 is a while statement uses variable(s) in conditional expression
- `Select <ifs, v> pattern ifs(v,_,_)` will return `4 x`

- `Select <w, v> pattern w(v,_)` will return `5 x, 5 y`

[Generated by MarkBind 4.0.2]

With Clauses

This is a new type of clause that compares attribute values.

```
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
```

[Generated by MarkBind 4.0.2]

Multi-clause Queries

Note that a query can now contain any number of such-that, pattern and with clauses.

- Clauses can be swapped without changing the meaning of the query.
- `and` can be used to connect clauses of the same type.

The following queries are the same: `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) such that Parent* (w, a) and Modifies (a, "x")`

The following queries are the same: `assign a; while w;`

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

However, the following queries are syntactically incorrect. `and` should not be used to connect or introduce clauses of different types. E.g.: `assign a; while w;`

- `Select a such that Parent* (w, a) and Modifies (a, "x") and such that Next* (1, a)`
  - There is usage of `and` and `such that` together before `Next* (1, a)`.
- `Select a such that Parent* (w, a) and pattern a ("x", _) such that Next* (1, a)`
  - There is usage of `and` and `pattern` together before `a ("x", _)`.
- `Select a such that Parent* (w, a) pattern a ("x", _) and Next* (1, a)`
  - There is usage of `and` to connect a pattern clause `a ("x", _)` and such-that clause `Next* (1, a)`.

[Generated by MarkBind 4.0.2]

Example Queries

In this section, we will use Code 6 to answer some queries.

For simplicity, procedures `First` and `Third` are excluded from statement numbering.

Note that for each scenario, there are multiple ways to write a query.

Code 6

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

        procedure Second {
01          x = 0;
02          i = 5;
03          while (i!=0) {
04              x = x + 2*y;
05              call Third;
06              i = i - 1; }
07          if (x==1) then {
08              x = x+1; }
            else {
09              z = 1; }
10          z = z + x + i;
11          y = z + 2;
12          x = x * y + z; }

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

Meaningful Queries

Q1. What are the procedures in the program call another procedure?

```
procedure p, q;
Select p such that Calls (p, _)
```

or

```
procedure p, q;
Select p.procName such that Calls (p, q)
```

Answer: `First, Second`

Note that we can select attributes in the select clause.

Wildcard `_` can be used as a placeholder for an unconstrained design entity (procedure in this case). It can only be used when the context uniquely implies the type of the argument denoted by `_`. In this case, we infer from the program design model that `_` stands for the design entity "procedure".

Q2: Which procedures directly call procedure "Third" and modify the variable "i"?

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

or

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

Answer: `Second`

Q3: Which procedures call procedure "Third" directly or indirectly?

```
procedure p;
Select p such that Calls* (p, "Third")
```

Answer: `First, Second`

Q4: 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: `Third`

Q5: Is there an execution path from statement 2 to statement 9?

```
Select BOOLEAN such that Next* (2, 9)
```

Answer: `TRUE`

Q6: 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`

Q7: 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

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

Answer: `4`

Q8: Which statements can be executed between statement 5 and statement 12?

```
stmt s;
Select s such that Next* (5, s) and Next* (s, 12)
```

Answer: `3, 4, 5, 6, 7, 8, 9, 10, 11`

Q9: Which assignments directly or indirectly affect value computed at assignment 10?

```
assign a;
Select a such that Affects* (a, 10)
```

Answer: `1, 2, 4, 6, 8, 9`

Q10. Which are the pair of assignments that affect each other?

```
assign a1, a2;
Select <a1, a2> such that Affects (a1, a2)
```

or

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

Answer omitted.

Q11. 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`

Query Formation

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

Q12. Find all statements whose statement number is equal to some constant.

```
stmt s; constant c;
Select s with s.stmt# = c.value
```

Q13. Find procedures whose name is the same as the name of some variable.

```
procedure p; variable v;
Select p with p.procName = v.varName
```

Q14. Find statements that is followed by statement 10.

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

Q15. 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)
```

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

Q16. 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 statement 60.

```
assign a; while w; stmt s;
Select a such that Parent* (w, a) and Next* (60, s) pattern a("x", _) with a.stmt# = s.stmt#
```

Q17. Find all while statements with "x" as a control variable.

```
while w;
Select w pattern w ("x", _)
```

Q18. Find assignment statements where variable x appears on the left hand side.

```
assign a;
Select a pattern a ("x", _)
```

Q19. Find assignments with expression x*y+z on the right hand side.

```
assign a;
Select a pattern a (_, "x * y + z")
```

Q20. 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"_)
```

Q21. Find all assignments to variable "x" such that value of "x" is subsequently re-assigned 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)
```

Meaningless Queries

SPA should be able to evaluate any syntactically and semantically valid query, even if some of these queries do not have meaning or make little sense.

Q22. Is the value 12 equals to 12?

```
Select BOOLEAN with 12 = 12
```

Answer: `TRUE`

Q23. Is statement 12 an assignment?

```
assign a;
Select BOOLEAN with a.stmt# = 12
```

Q24. If statement 12 is an assignment statement, find all the assignment statements.

```
assign a, a1;
Select a1 with a.stmt# = 12
```

or

```
assign a, a1;
Select a1 with 12 = a.stmt#
```

Note that it is possible to swap the values to be compared around.

Q25. If procedure "Second" calls procedure "Third", find all the combinations of an assignment-while statement pairs.

```
assign a; while w;
Select <a, w> such that Calls ("Second", "Third")
```

Q26. If there exist an assignment that modifies the variable "y", find all the combinations of two-procedures pair.

```
procedure p, q; assign a;
Select <p, q> such that Modifies (a, "y")
```

Q27. Is there a procedure that calls some other procedure in the program?

```
Select BOOLEAN such that Calls (_,_)
```

[Generated by MarkBind 4.0.2]

Format of Results

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

Select	Should return
BOOLEAN	In SPA implementation, return an array of a single string TRUE or FALSE. On paper, write either TRUE or FALSE.
Tuple (e.g. <a, p, s>)	In SPA implementation, arrange tuple elements separated by space in a string, and return an array of strings. E.g. Return a list of strings where the first element is 3 First 10, second element is 5 Second 4 etc.

Otherwise, format of results previously discussed in Basic SPA requirements still holds.