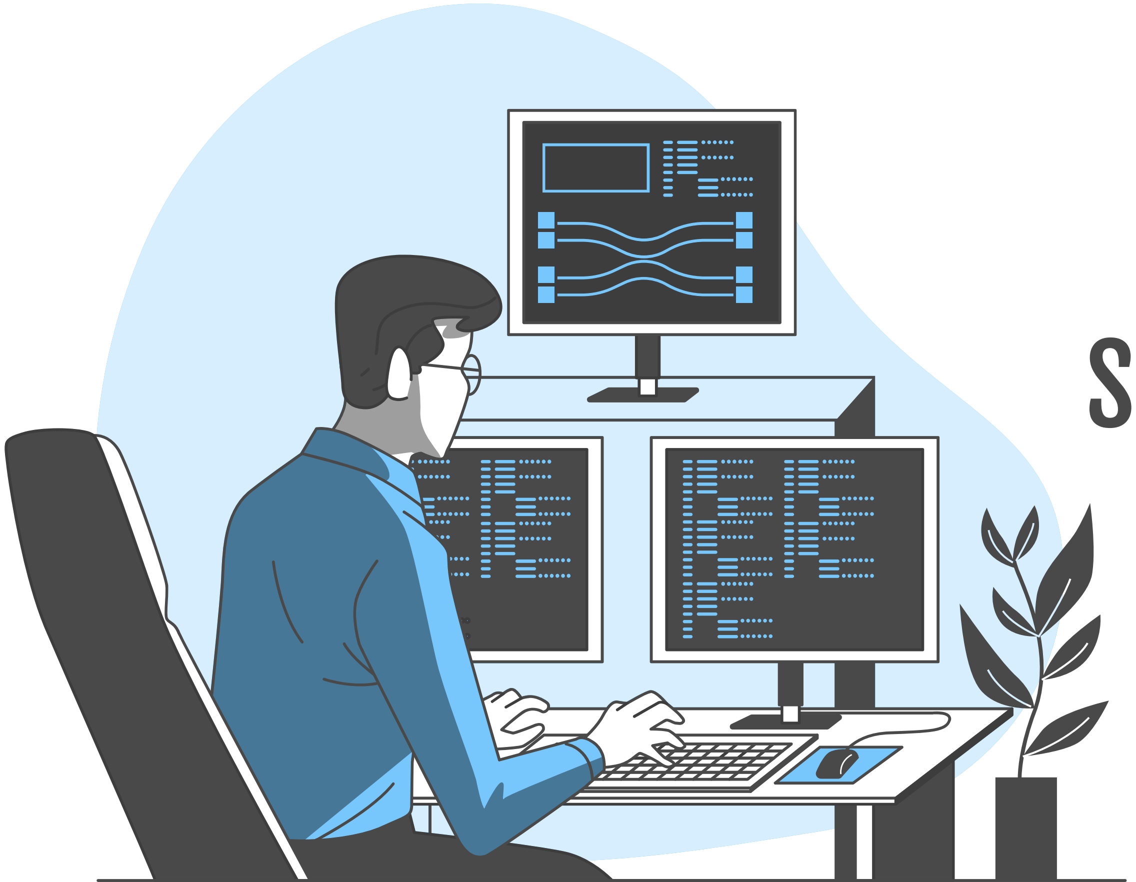
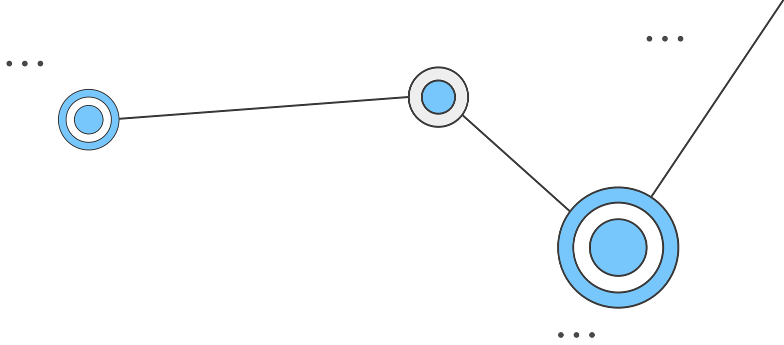
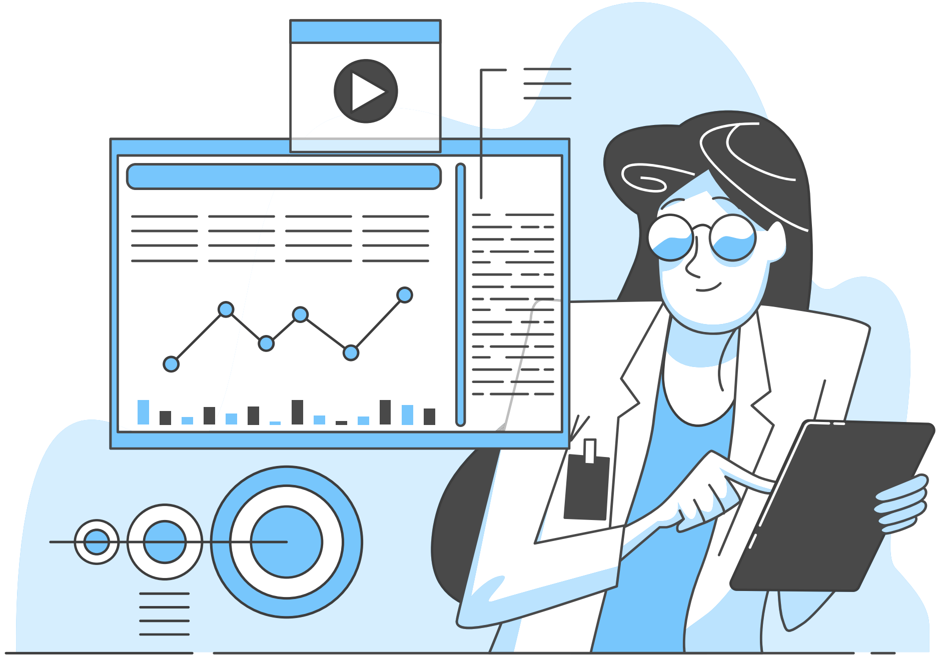




Static Program Analyzer

Team07

Table of Contents



Project Roles

Title	In Charge	Assigned
Team Lead	Project Management	Zong Sien
Tech Lead I	Architecture & Integration	Haliq
Tech Lead II	Test strategy	Pakorn
Editor	Documentation	Dian Hao
Editor	Code Quality	Bo Hao
QA/Tester	Optimisation	Chuan Kai

Component Assignment

Parser

Haliq & Zong Sien

...

PKB

Pakorn & Dian Hao

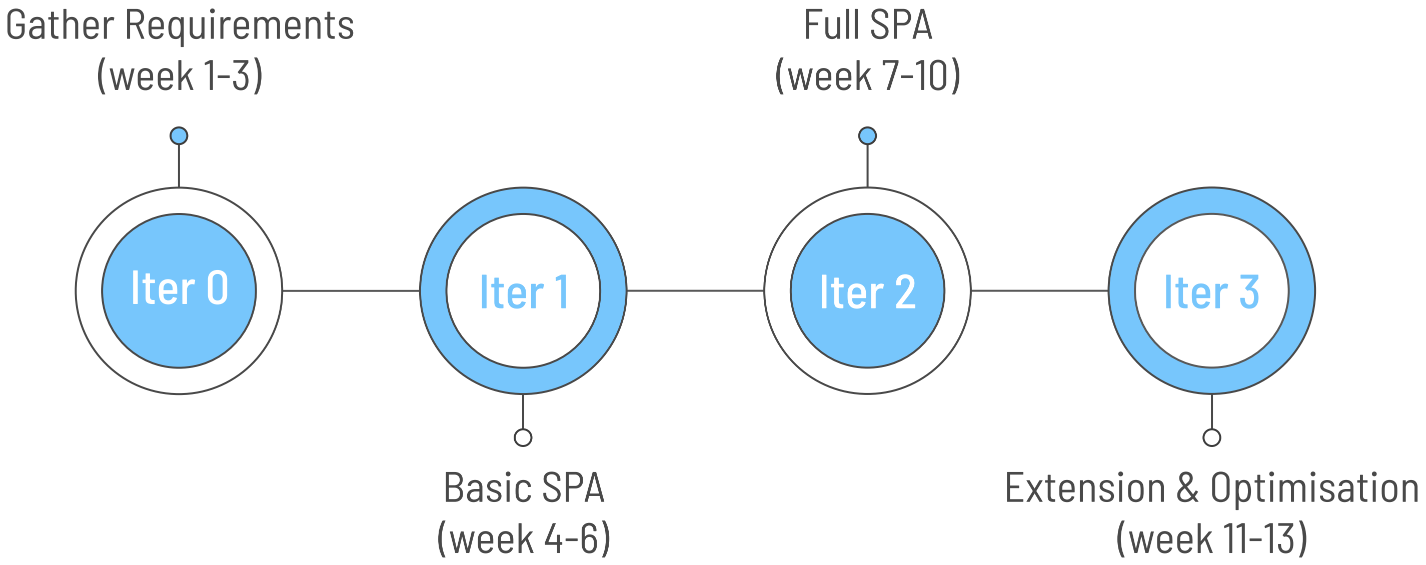
...

Evaluator

Bo Hao & Chuan Kai

...

Iterative Breadth-First (SDLC)


Gather Requirements (week 1-3)
Iter 0
Iter 1
Basic SPA (week 4-6)
Full SPA (week 7-10)
Iter 2
Iter 3
Extension & Optimisation (week 11-13)

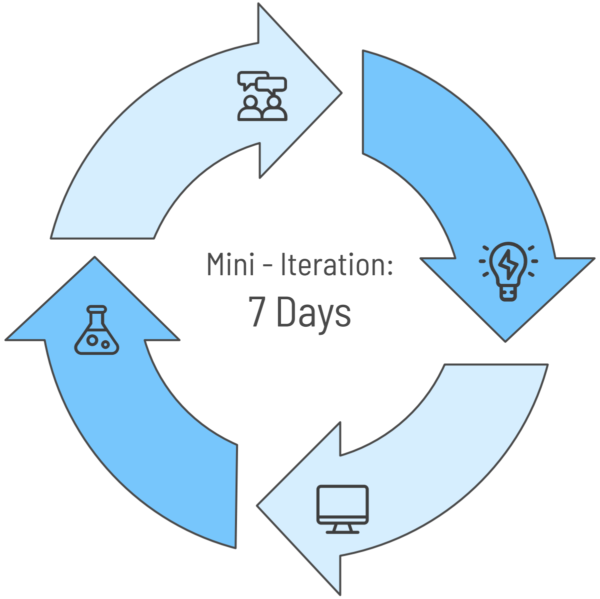
Mini - Iteration

Merge

Code review and merge PRs.

Test

CI unit & integration tests.



Design

Define and design features to implement.

Implement

Development of SPA.

SWE Practices

Coding standards

Naming conventions,
linter (clang-tidy)

Single Responsibility Principle

A class should have only
one reason to change

Open-closed Principle

Software entities should be open for
extension, but closed for modification

Interface Segregation Principle

No client should be forced to
depend on methods it does not use

Features

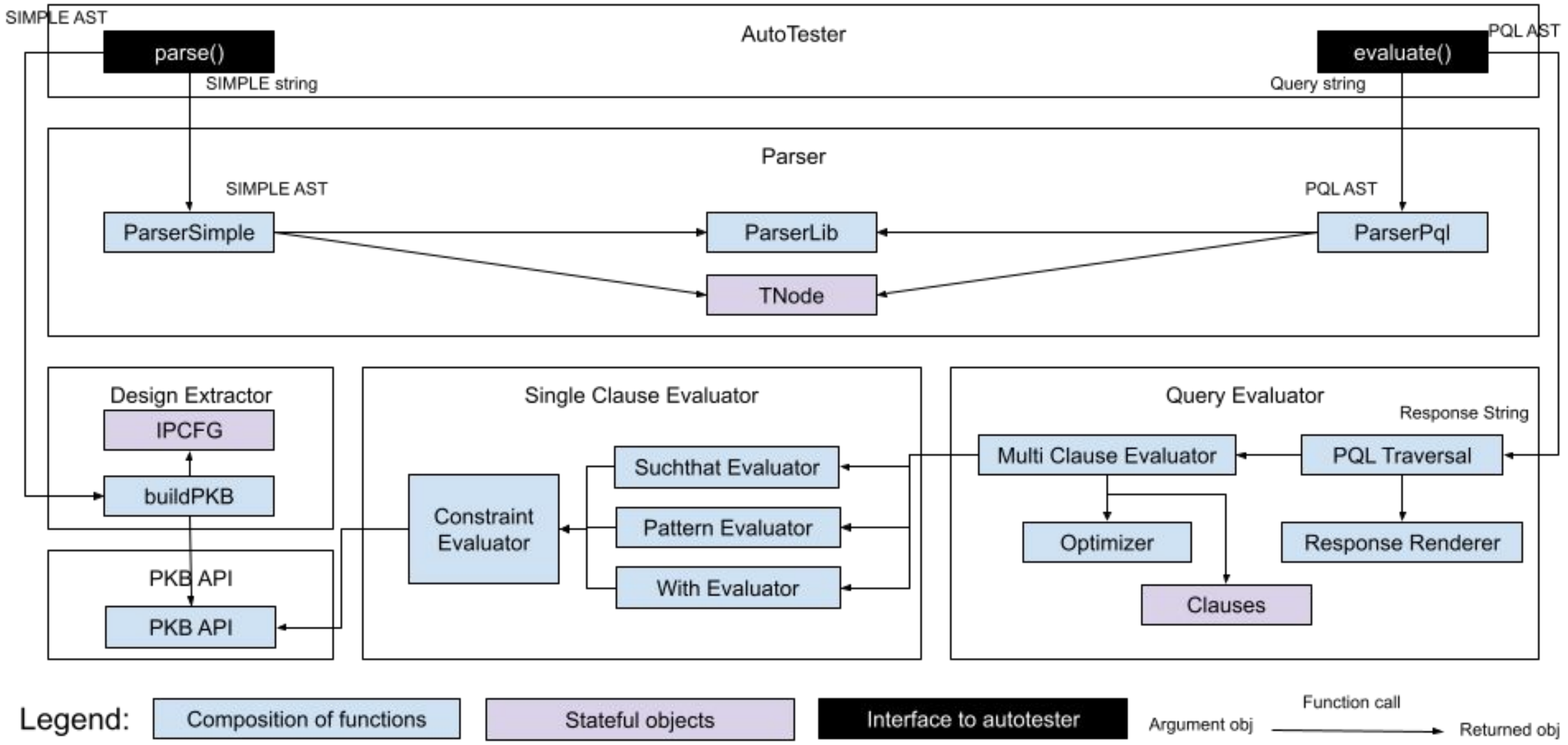
SIMPLE

- Parse support:
  - program, procedure, stmtList, stmt, read, print, while, if, assign, cond_expr, rel_expr, rel_factor, expr, term, factor, var_name, proc_name, const_value, variable
- Validation support:
  - call defined procedures, no duplicate procedure definition, no cyclic procedure calls
- Pretty print parse errors

PQL

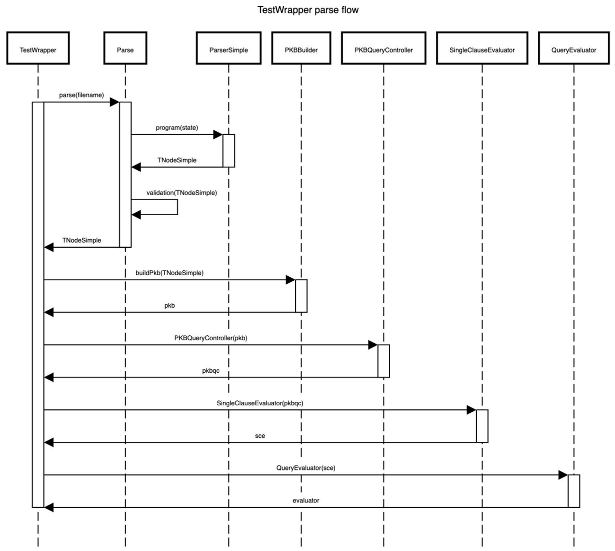
- Design entities:
  - procedure, stmt, assign, read, print, while, if, variable, constant, prog_line
- Result clause:
  - single & multiple synonyms
  - boolean
  - attribute reference
- Constraint clause:
  - Such That: ModifiesP, ModifiesS, UsesP, UsesS, Calls(T), Parent(T), Follows(T), Next(T), Affects(T), NextBip(T), AffectsBip(T)
  - Pattern: syn_assign, syn_if, syn_while
  - With
- Pretty print parse errors

Architecture

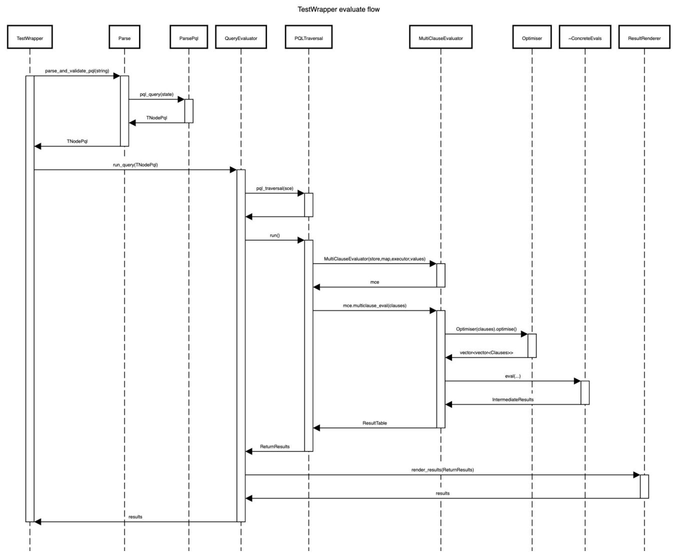


Component Interaction

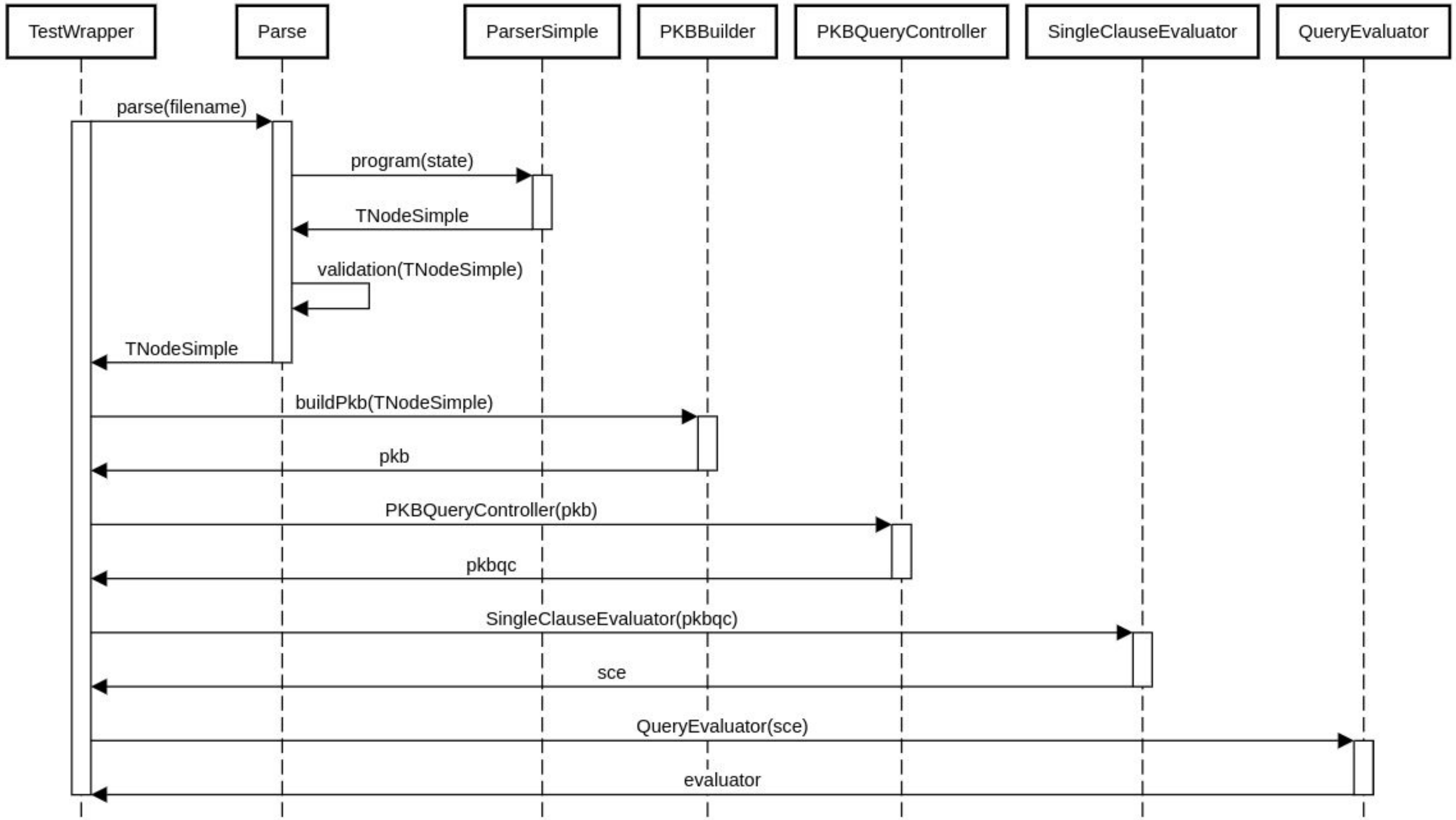
Parse Sequence



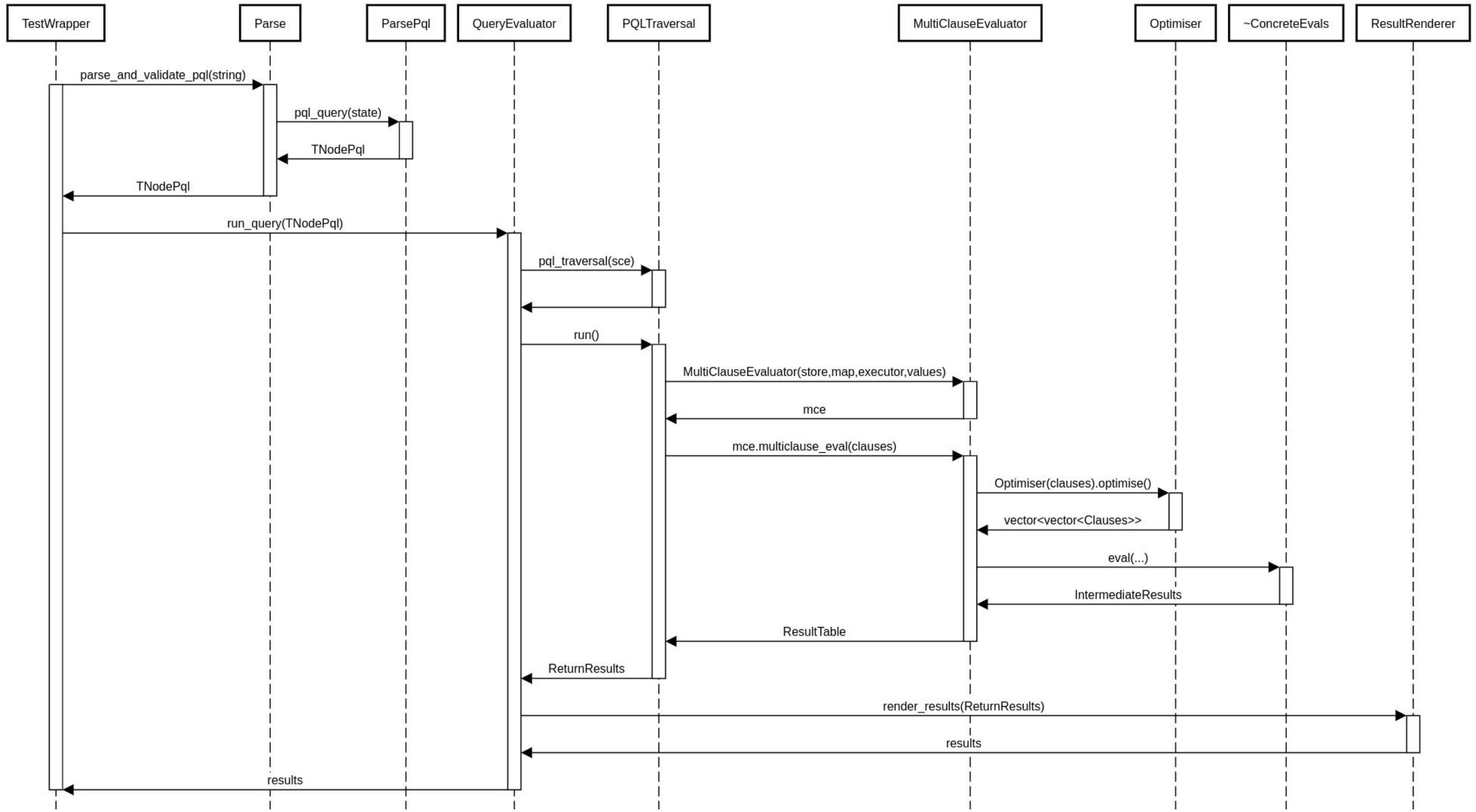
Evaluate Sequence



Parse Sequence Diagram



Evaluate Sequence Diagram



Notable Highlights

Parser Design

Reusable across SIMPLE and PQL. Pretty prints parsing errors

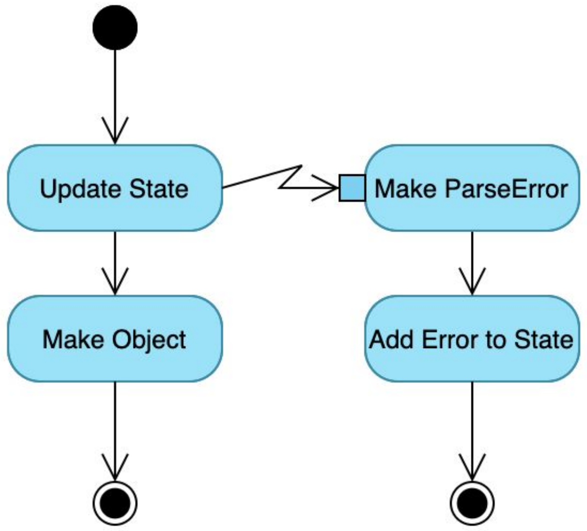
Optimisation

Pruning, precomputation and priority sorting of clauses

Design Extractor Design

Modular, extensible and an elegant recursive solution

Parser Library



Parser combinators take in a state, mutates it and returns an object corresponding to the parse.

The state mutation is typically advancing the index of the currently consumed character of the source code string.

There is a library defining primitive parsers such as matching a single character or a string, or matching characters that satisfy a predicate.

Simple Parser

Composing parsers out of other simpler parsers is how more complex parsers are made

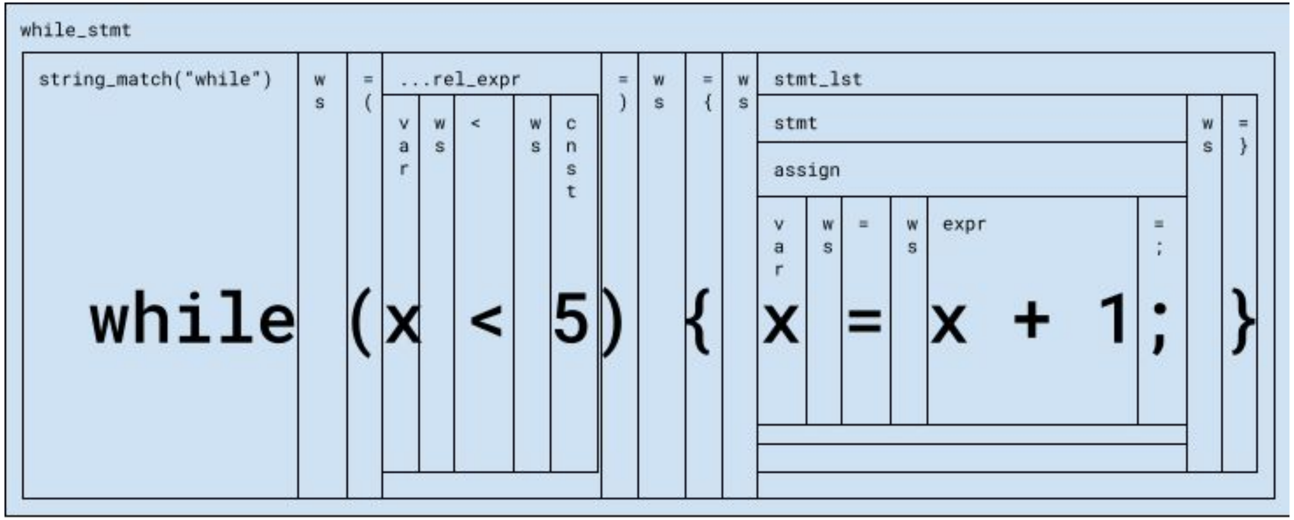
Clever use of try catch to select alternative parsers if one fails

Upon parse failure we also know the stack of parsers that end up failing (pretty printing)

```
Phase 1:
   parsers: ... > while_stmt

   while ( x < 5) { x = x + 1; }
   ^
```

Simple Parser



```
Phase 1:
   parsers: ... > while_stmt

   while ( x < 5) { x = x + 1; }
   ^
```

Design Decision of Parser

	Ease of Development	Maintainability	Readability
Parser Combinators (chosen)	Simple function calls composition performing recursive descent	Functions are modular and are adaptable to syntax changes. Can be used for both SIMPLE and PQL.	Function call order closely corresponds to grammar
Lexer + LALR Parser	Extremely complex but efficient, there are many parser generators based on this design	If the parser is generalized to the point of a parser generator only then will it be maintainable, otherwise logic is tightly coupled	If generalized to the point of a parser generator, the grammar can be fed as is, but typically we can’t easily identify the grammar from the parser code

Optimisation – Constraint Evaluator

Key Features:

- Stateful pruning of candidate values
- Precomputation of non-null queries

Example of non-null query:

- stmt s; Select s such that Follows*(s, 4)

Example code:

```
procedure main {
   read a;      // 1
   print b;     // 2
   a = a + 1;   // 3
   b = a + 4;   // 4
}
```

Design Decision of Constraint Evaluator

	Time complexity (each query)	Time complexity (precomputation)	Space complexity
Precompute	O(N) for non-null queries O(1) for null queries	Additional time required to perform precomputation	Additional space required to store precomputation data
On-the-fly	O(N) for all queries	No additional time needed for precomputation	No additional space needed
Mixed: precompute & on-the-fly (chosen)	O(N) for non-null queries O(1) for most null queries	Additional time required to perform precomputation Precompute only for reasonably bounded queries	Additional space required to store precomputation data

Optimisation – Multi Clause Evaluator

Key Features:

- Clause grouping
- Intra group priority sort
- Inter group size sort
- Clause result size sort

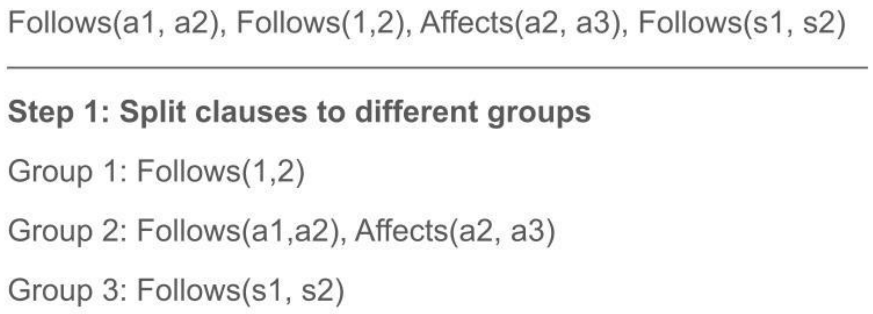
Optimisation Algorithm:

1. Split clauses into different groups.
2. Sort the groups.
    a. Intra-sort within the groups based on priority score.
    b. Inter-sort across groups based on group size.
3. Evaluate each clause.
4. Sort the groups based on size of the clause result.
5. Merge the results.

Optimization – Multi Clause Evaluator

Optimisation Algorithm:

1. Split clauses into different groups.
2. Sort the groups.
    a. Intra-sort within the groups based on priority score.
    b. Inter-sort across groups based on group size (synonyms).
3. Evaluate each clause.
4. Sort the groups based on size of the clause result.
5. Merge the results.



Follows(a1, a2), Follows(1,2), Affects(a2, a3), Follows(s1, s2)

Step 1: Split clauses to different groups

Group 1: Follows(1,2)

Group 2: Follows(a1,a2), Affects(a2, a3)

Group 3: Follows(s1, s2)

NextBip Extractor

Key Features:

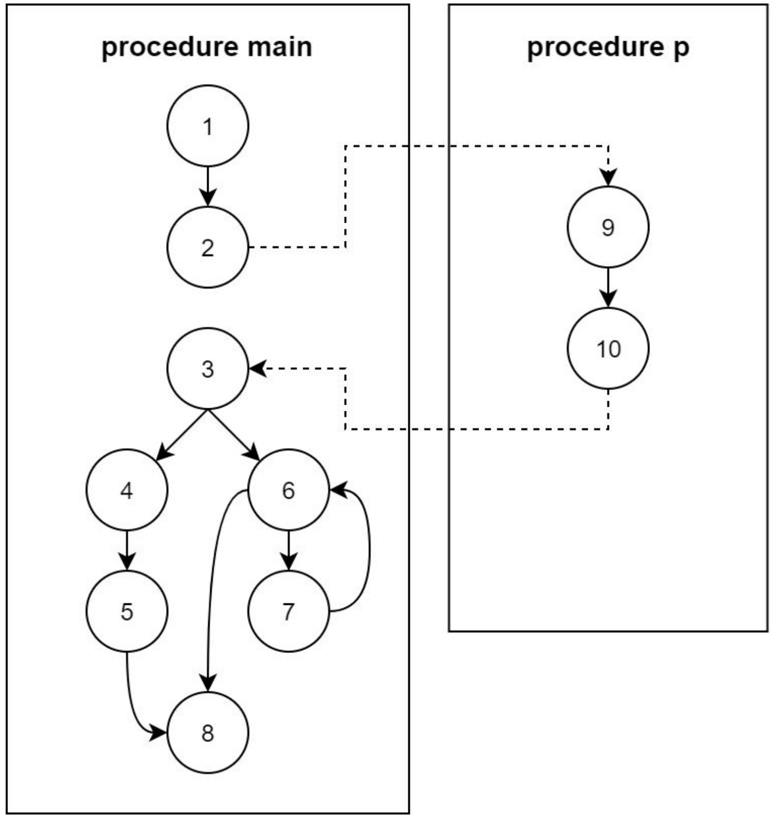
- Implemented as a function that takes in an AST and reference to a PKB Data Structure Object
- Store results by mutating the referred PKB Data Structure Object
- Blocks-based extraction algorithm, with each blocks corresponding to a statement, statement list, or procedure
- Each block contains information on the first and last statements in the block to be executed

NextBip Extractor

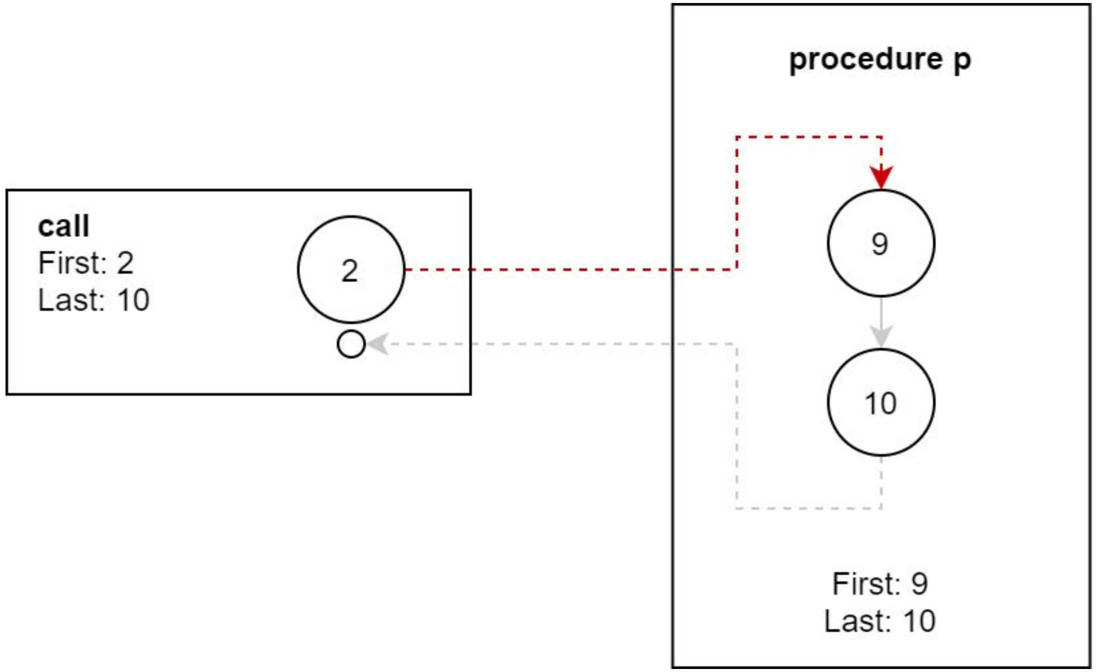
code

```
procedure main {
1.   x = 1;
2.   call p;
3.   if (y > 0) then {
4.       y = y - 1;
5.       x = y
     } else {
6.       while (x < y) {
7.           x = x + 1;
         }
     }
8.   print x;
}

procedure p {
9.   read y;
10.  y = y + 1;
}
```



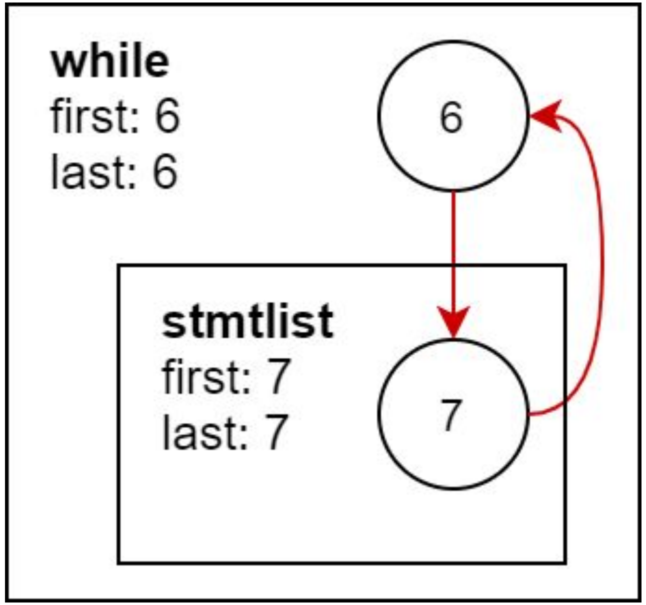
NextBip: Call Block



Relationships Extracted:

(2, 9)

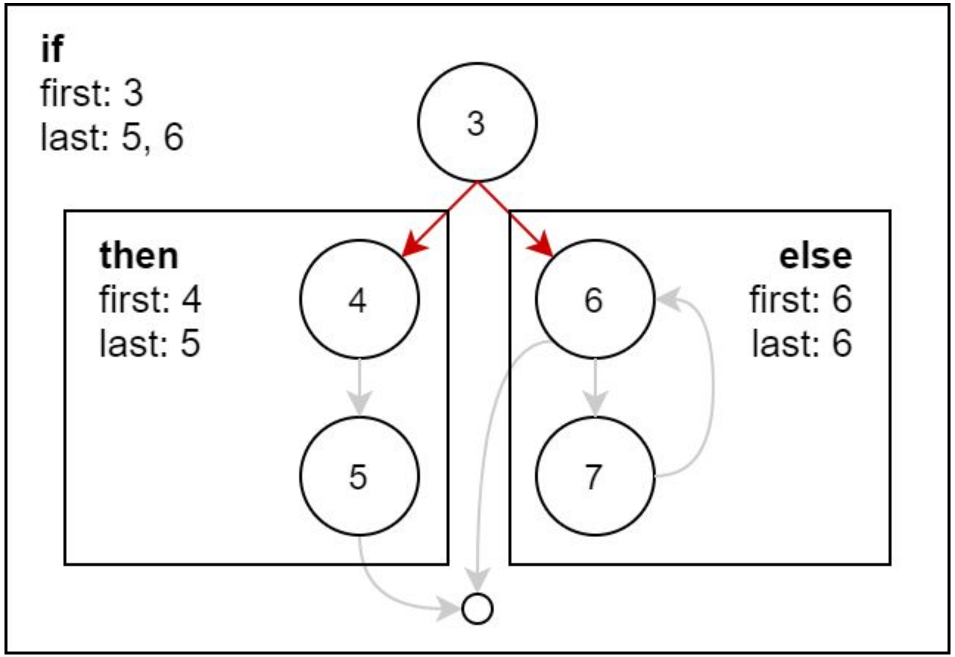
NextBip: While Block



Relationships Extracted:

(6, 7), (7, 6)

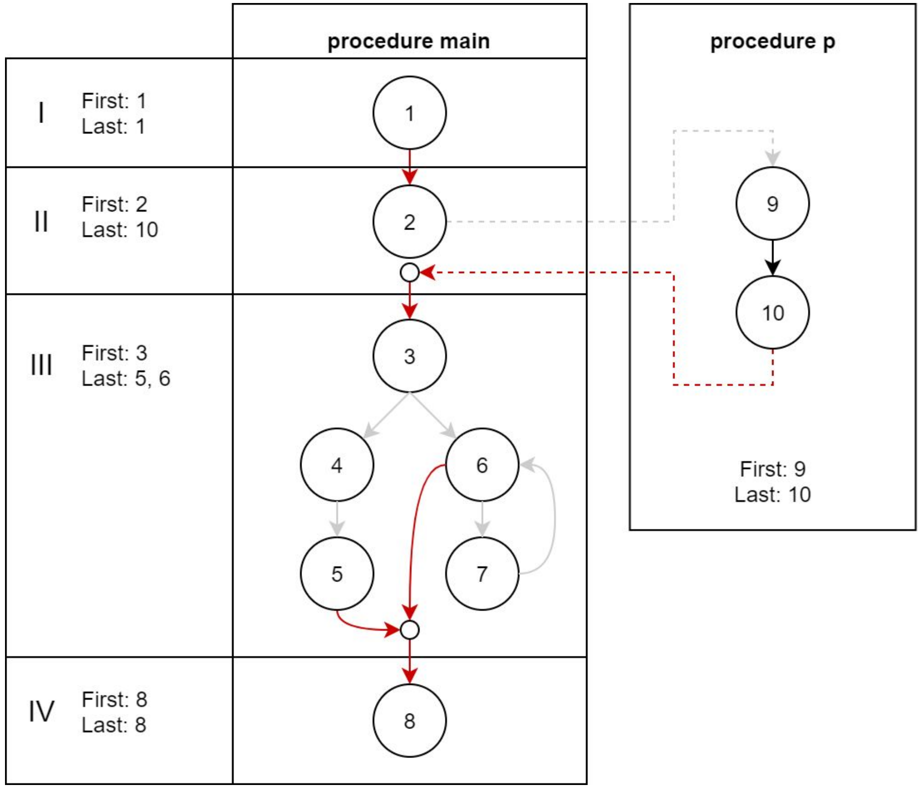
NextBip: If Block



Relationships Extracted:

(3, 4), (3, 6)

NextBip: Statement List Block



Relationships Extracted:

(1, 2), (10, 3), (5, 8), (6, 8)

Design Decision of NextBip Extractor

	Ease of Implementation	Speed
CFG Based Design	Difficult, due to the need to consider proper usage of branch-in/branch-backlinks.	More time consuming as the Interprocedural CFG needs to be generated and traversed.
Blocks Based Design (chosen)	Easy, problem is decomposed into smaller piece of logic for each type of blocks.	Faster as there is no need to generate the CFG.

NextBip Extractor: Storing the Results

- Extracted relationship are directly written into the referred PKB Data Structure Object
- For example, the relationship NextBip(2, 4) is recorded with the instruction

```
pkb.statements[“2”].nextbip.insert(“4”)
```

which adds 4 to the list of statements that is immediately executed after 2 in some interprocedural control flow path, and

```
pkb.statements[“4”].prevbip.insert(“2”)
```

which stores the inverse relationship

Test Strategy

1. Unit Test

- Component wise test for correctness in isolation
- 6 subcomponents

2. Integration Test

- Component interaction test for correctness
- 5 pairs of subcomponent interaction
  - SIMPLE Parser + DE
  - DE + PKB
  - PKB + Constraint Evaluator
  - Constraint Evaluator + QE
  - PQL Parser + QE

3. System Test

- System wide test for correctness
- 5 group types:
  - Single Clause
  - Selection
  - Combination
  - Invalid (SIMPLE, PQL)
  - Stress

Unit Test Example

Test Example: Extract relationships from a mock SIMPLE AST.
Test purpose: Tests the correctness of pattern decomposition into subexpressions.
Required input: PKB.
Expected results: A REQUIRE function is used to ensure correct extraction.

```
REQUIRE(pkb.assignments["10"].subexpr.find("((v)-(1))") != pkb.assignments["10"].subexpr.end());
```

Integration Test Example

Test Example: Extract relationships from the parsed SIMPLE AST.
Test purpose: Tests the correctness of extracted Next relationship.
Required input: Simple source string.
Expected results: A REQUIRE function is used to ensure correct extraction.

```
REQUIRE(pkb.statements["1"].next.find("2") != pkb.statements["1"].next.end());
```

System Test

01 **Single Constraint**
Permutation of constraint arguments

02 **Selection**
Permutation of result clause selection

03 **Combination**
Random mix of multi-constraint clauses

04 **Invalid**
Syntactic & Semantic errors in SIMPLE & PQL

05 **Stress**
Nested SIMPLE code

System Test: Single Constraint

SIMPLE

- Specifically designed to match the topologies required for each test case

```
procedure a {
   …
   while (y > 1) {
      …
       if (y > 3) then {
         y = y - 1;
      } else {
         …
      }
      while (x < 2) {
          …
      }
   }
   …
}
```

PQL

- Permuted list of queries with all possible types of arguments for each constraint
- Example: Integer & Assign Synonym

```
>      assign a;
       Select a.stmt# such that Affects(41,a)
```

- Example: Wildcard & Assign Synonym

```
>      assign a;
       Select a such that Affects(_,a)
```

System Test: Selection

SIMPLE

- Relatively simple SIMPLE code with focus on the query.

```
procedure main {
   e = f;
   if (a == 1) then {
      while (j>a) {
         ...
      }
   } else {
      c = j + 3;
   }
}

procedure second {
   a = b;
   c = d;
}
```

PQL

- Includes all 3 types of result clauses
  - Boolean synonym selection
    ```
    >    read r; print p;
         Select BOOLEAN
    ```
  - Single synonym selection
    ```
    >    assign a;
         Select a.stmt#
    ```
  - Tuple selection
    ```
    >    read r; print p;
         Select <r.varName,p.stmt#>
    ```

System Test: Combination

SIMPLE

- Reasonably complicated SIMPLE source code
- Each source code contains all types of statements, multiple procedures, and multiple levels of nesting

PQL

- Queries are sampled from a generator script that generates all 2 or 4 constraints combination

System Test: Invalid Code/Query

SIMPLE

- Contains source code that are syntactically invalid:

```
if (x < 5) {
    a = a + 1;
} else {
    a = a + b;
}
```

- Contains source code that are semantically invalid:

```
procedure a {
    call b;
}
procedure b {
    call a;
}
```

PQL

- Includes syntactically invalid queries:

```
>    stmt s stmt s;;
     Select s
```

- Includes semantically invalid queries:

```
>    stmt a;
     Select BOOLEAN such that Calls(_, a)

>    variable v;
     Select v such that NextBip*(_,v)
```

System Test: Stress

SIMPLE

- Heavily nested while loops

```
procedure nest {
    while (...) {
        while (...) {
            while (...) {
                ...
            }
        }
    }
}
```

PQL

- Any valid query

```
>     Select BOOLEAN
```

Thanks!

Do you have any questions?