SPA Project Report

Iteration 2

Muhammad Naufal Dusan Urosevic, e0657867@u.nus.edu

Chan Kong Yew, A0227199R e0650692@u.nus.edu

Contents

1. Scope of the prototype implementation

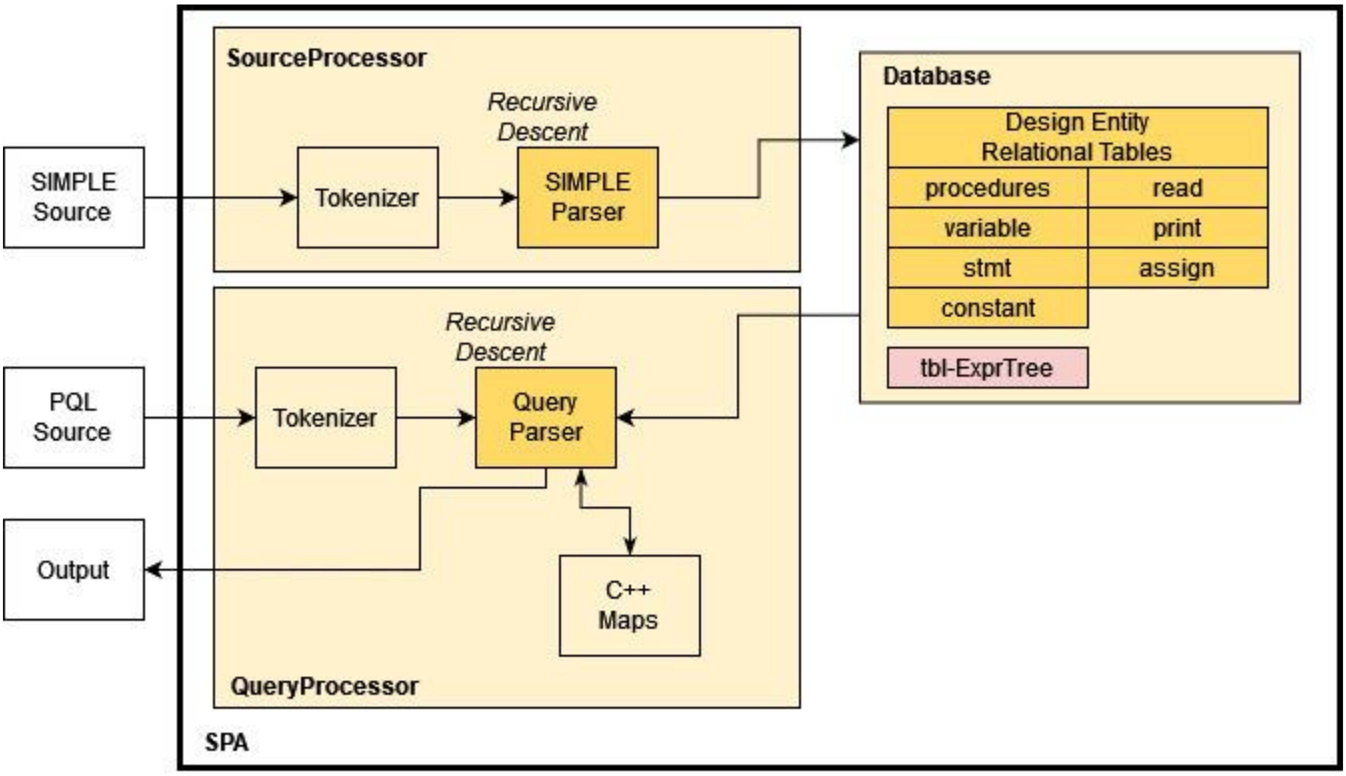
The objectives of Iteration 1 are taken from the project Wiki and the requirements fulfilment matrix is provided in Appendix 1a and 1b.

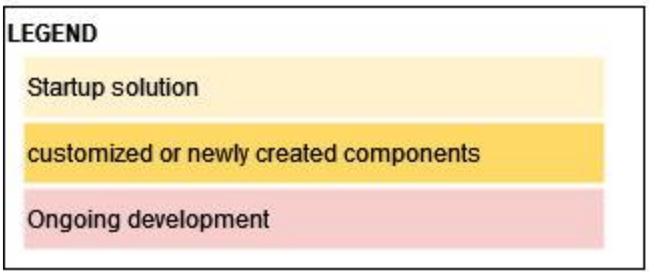
In general, all stated Iteration 1 requirements are met. However, Iteration 2 requirements are partially met , incomplete work are planned for in iteration 3 and constraints are described in section 3) Conclusion. Current design in this report is reflective of such constraints.

2. SPA design

2.1 Overview

The system diagram below shows the SPA design leveraging the Startup Solution provided. 3 major components in place are: SourceProcessor, QueryProcessor and Database and a matching C++ class exists in the codes. Following sections elaborate on the design of each class.

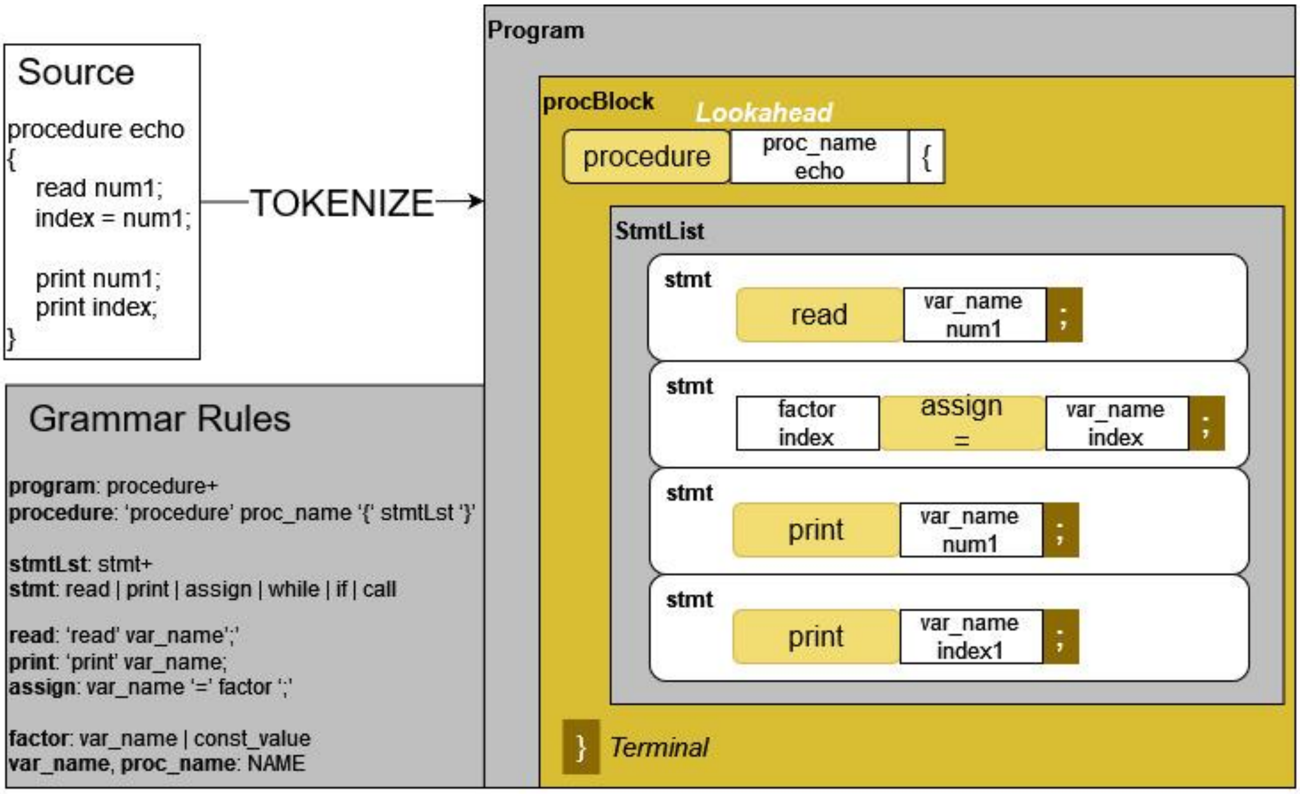




2.2 Design of SPA components

Source Processor

The Source Processor supports the grammar rules and syntax diagram in Appendix 2 through the SimpleParser class, that implements a recursive descent parser(Jack, 2013). Each design entity corresponds to a private method in the SimpleParser class, exposing only a public method simpleparse that correspond to "program". After the database is initialized, the tokenizer places each "word" in the SIMPLE source file into a vector of string. The parser, then, processes this vector as a queue of tokens. By looking one token ahead of the currently processed token, the parser can match and identify a segment of tokens against SIMPLE statement types, namely read, print and assign in Iteration 1. Database class methods are used to insert information like, e.g., proc_name, line number into different tables. These tables, in turn, form the basis of evaluating query results in QueryProcessor.



Parser Pseudocode

Init
Initialize counters and operator map

fetchToken
fetch token from vector and map
to an operator, keyword, NAME or const_value

lookAhead(expected upcoming token)
return sucess if expected upcoming token
matches token retrieved by fetchToken()

simpleparse(Program block)
preload token queue - Call fetchToken()
Call procBlock()
return success if next token is close
curly bracket

procBlock
if token read is keyword "procedure"
followed by NAME
write entity information into **procedure**
table through Database methods
if lookAhead token is open curly
Call statementList()

statementList
Repeat until terminal curly close bracket
lookAhead() to fetch token
if keyword token(read/print)
lookAhead() to match stmt grammar
write entity information into **read** and
variable table through Database methods
if token is a NAME
lookAhead() to match assignment grammar
write entity information into **assign, constant** or
variable table through Database methods
if token is a semicolon(terminal)
advance lineNo counter
write entity information into **statement** table
through Database methods

Handling Expressions

Since the grammar rules for expression(expr) and term appears as the first component in its definition, it has the potential to cause infinite recursion loops. Therefore, left recursion elimination is necessary. Below are the modified rules applied in parsing expressions and included in syntax trees in appendix 2.

Grammar rule for expression:
expr: expr ‘+’ term | expr ‘-’ term | term

After left recursion elimination:

expr: term exprPrime

exprPrime: '+' term exprPrime | '-' term exprPrime | ε

Grammar rule for term:

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

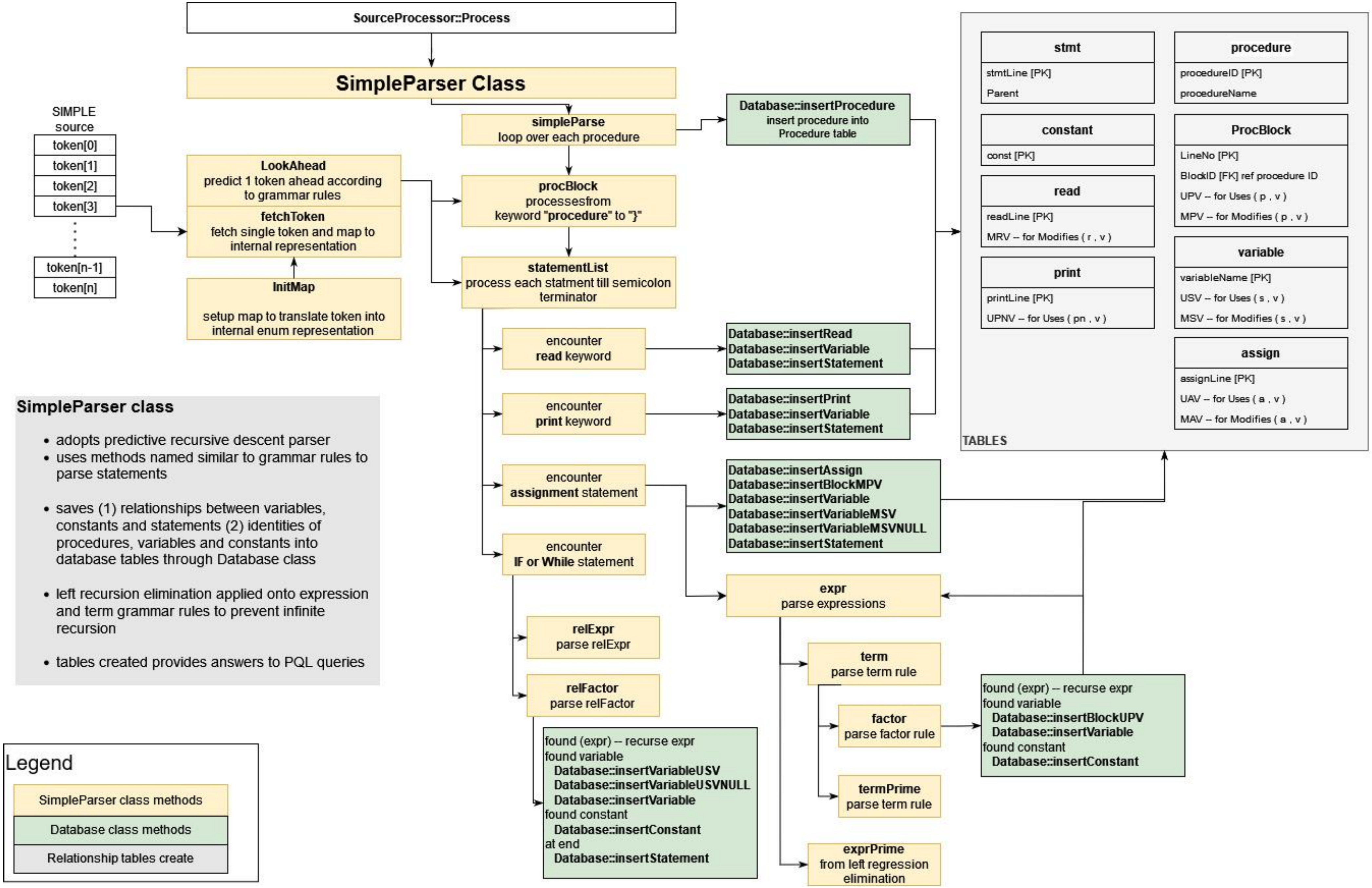
After left recursion elimination:

term: factor termPrime

termPrime: '*' factor termPrime | '/' factor termPrime | '%' factor termPrime | ε

The SimpleParse class responsible for parsing receives corresponding updates to handle expressions, with class methods added that mirrors the names of each grammar rule. The next diagram shows how SimpleParse methods interacts with the Database class and to create tables that are uncovered during the parsing process.

Appendix 3 illustrates the design process of how relational information between SIMPLE statements are determined and populated in newly created database tables. These tables are, subsequently, used to answer PQL queries which undergoes a parsing process described in the QueryParser section.


SourceProcessor::Process
SimpleParser Class
SIMPLE source
token[0]
token[1]
token[2]
token[3]
token[n-1]
token[n]
LookAhead
predict 1 token ahead according to grammar rules
fetchToken
fetch single token and map to internal representation
InitMap
setup map to translate token into internal enum representation
simpleParse
loop over each procedure
procBlock
processesfrom keyword "procedure" to "}"
statementList
process each statment till semicolon terminator
encounter read keyword
encounter print keyword
encounter assignment statement
encounter IF or While statement
relExpr
parse relExpr
relFactor
parse relFactor
found (expr) – recurse expr
found variable
Database::insertVariableUSV
Database::insertVariableUSVNULL
Database::insertVariable
found constant
Database::insertConstant
at end
Database::insertStatement
Database::insertProcedure
insert procedure into Procedure table
Database::insertRead
Database::insertVariable
Database::insertStatement
Database::insertPrint
Database::insertVariable
Database::insertStatement
Database::insertAssign
Database::insertBlockMPV
Database::insertVariable
Database::insertVariableMSV
Database::insertVariableMSVNULL
Database::insertStatement
expr
parse expressions
term
parse term rule
factor
parse factor rule
termPrime
parse term rule
exprPrime
from left regression elimination
found (expr) – recurse expr
found variable
Database::insertBlockUPV
Database::insertVariable
found constant
Database::insertConstant
stmt
stmtLine [PK]
Parent
constant
const [PK]
read
readLine [PK]
MRV – for Modifies (r , v)
print
printLine [PK]
UPNV – for Uses (pn , v)
procedure
procedureID [PK]
procedureName
ProcBlock
LineNo [PK]
BlockID [FK] ref procedure ID
UPV – for Uses (p , v)
MPV – for Modifies (p , v)
variable
variableName [PK]
USV – for Uses (s , v)
MSV – for Modifies (s , v)
assign
assignLine [PK]
UAV – for Uses (a , v)
MAV – for Modifies (a , v)
TABLES
SimpleParser class
adopts predictive recursive descent parser
uses methods named similar to grammar rules to parse statements
saves (1) relationships between variables, constants and statements (2) identities of procedures, variables and constants into database tables through Database class
left recursion elimination applied onto expression and term grammar rules to prevent infinite recursion
tables created provides answers to PQL queries
Legend
SimpleParser class methods
Database class methods
Relationship tables create

Design of Database

The database(DB) is made up of 8 tables intended to depict the relationship between individual components of a SIMPLE program. Appendix 3 covers the design considerations on attributes added to tables in relation to PQL query types. It also lists the corresponding methods developed for QueryParser class to retrieve information from tables to answer queries.

DB Initialization

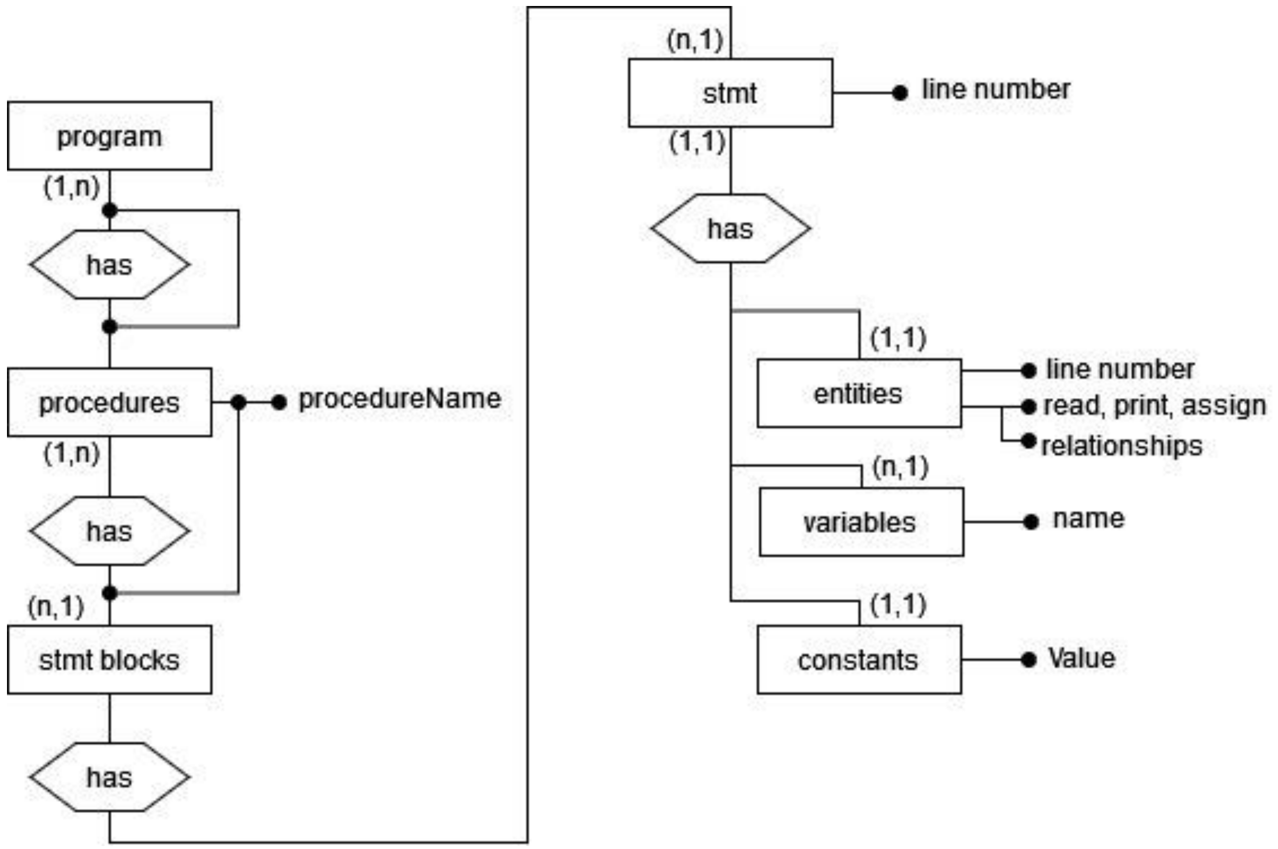
All 7 tables are created new on initialization, dropping any previous existing ones.

DB Insert methods

As the source SIMPLE program is parsed, SQL INSERT INTO statements embedded in different insert methods, e.g. insertStatement, allows relational information to be written into the DB

DB get methods

As queries are evaluated, SQL SELECT statements in different get methods, e.g. getProcedures, retrieves the



information necessary to generate query results.

Query Processor

The QueryParser class runs in a loop to parse the incoming vector of string tokens and returning a result string to QueryProcessor. Its design concept deviates from SimpleParse class where each PQL query is parsed in groups of components, e.g. QueryParser::fetchQToken has taken on composite role in processes the all declaration statements and identifying incoming tokens against grammar.

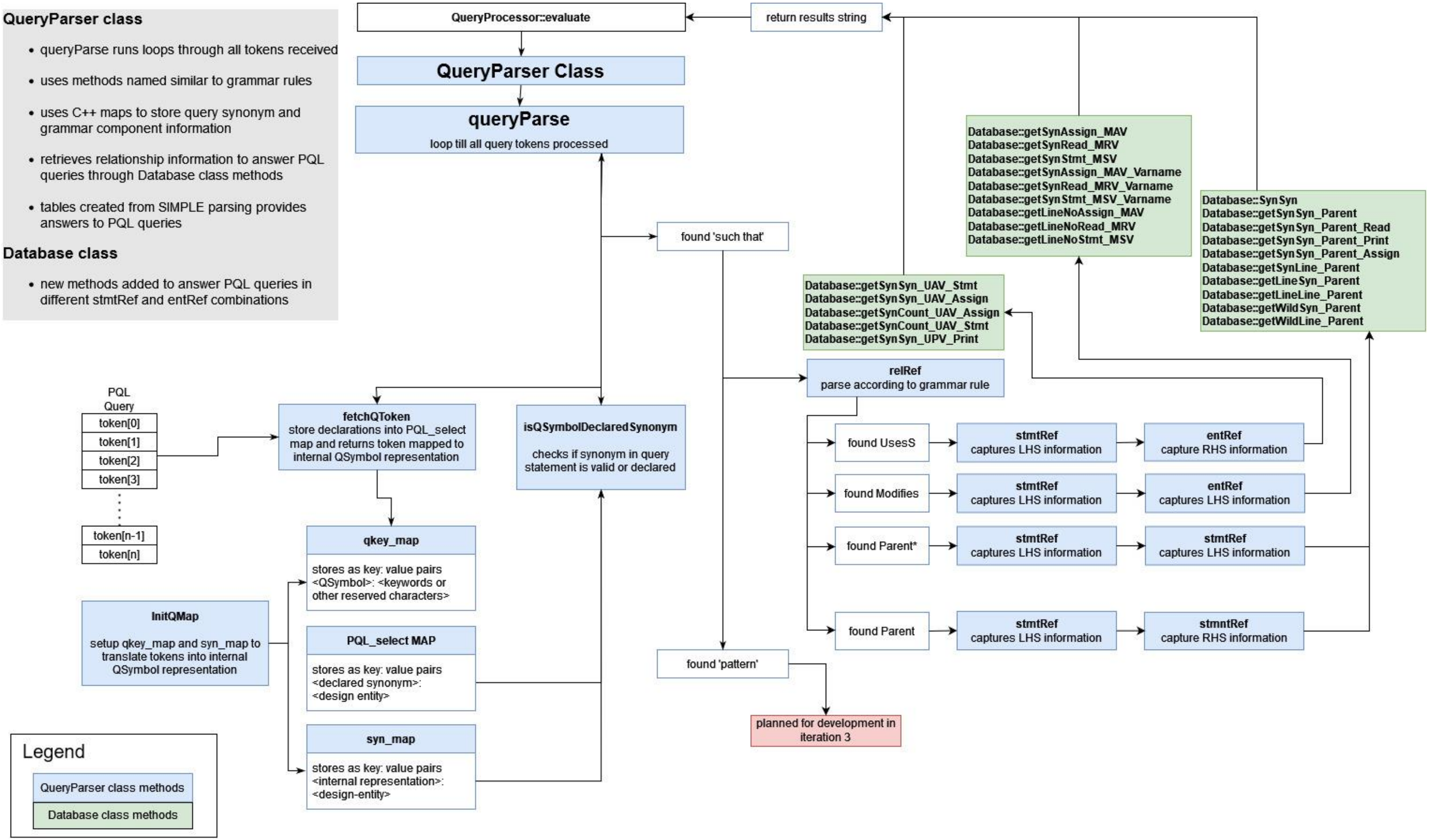
“such that” clauses are processed by labelling stmtRef as ‘LHS’ and entRef as ‘RHS’ in codes produced. In Parent/ParentT queries, RHS are stmtRefs following SIMPLE grammar rule. As LHS and RHS are decomposed and identified as a synonym, wildcard, statement identifying integer or a variable identifier, a matching Database class method is identified to answer the parsed query. To illustrate: *stmt s; Select s such that Parent(s,5)* decomposes to LHS as a synonym, whereas RHS decomposes to a line number identifier. This results in Database::getSynLine_Parent called to answer the query. Method getSynLine_Parent, in turn, queries stmt table to find the parent of statement 5.

QueryParser class

- queryParse runs loops through all tokens received
- uses methods named similar to grammar rules
- uses C++ maps to store query synonym and grammar component information
- retrieves relationship information to answer PQL queries through Database class methods
- tables created from SIMPLE parsing provides answers to PQL queries

Database class

- new methods added to answer PQL queries in different stmtRef and entRef combinations



3. Conclusion

Overall, the delivery timeline for iteration 2 was delayed where PQL queries are only completed till handling 'such that' clauses. The delay can be attributed to (1) a longer-than-expected period spent on understanding PQL and its queries, such that relationships in statement components can be placed as attribute or columns in tables (2) debugging white space variations in statements such that Tokenizer class is able to identify each statement components correctly and (3) deciding on a suitable architecture for QueryParser class.

Outstanding work includes (1) expression tree implementation to process pattern clauses and (2) testing on existing solution with all the permutations of stmtRef and entRef. (3) Handling of next line on the query source file on the tokenizer.

It is planned in iteration 3 schedule to complete these outstanding work, alongside a review of QueryParser and Tokenizer class design to cater for additional iteration 3 requirements.

Reference

Jack. (2013, April 21). *Recursive descent parser example for C*.

https://stackoverflow.com/questions/16127385/recursive-descent-parser-example-for-c

Retrieved January 28, 2022.

Appendix

1a. Requirements breakdown and implementation matching

Requirements	Count	Fulfilment	Matching implementation
Wiki: 1. General requirements for SPA prototype			
This prototype should allow you to enter a source program (written in SIMPLE) and some queries (written in a subset of PQL).	2	2	Based off statup solution
It should parse the source program, build some of the design abstractions in Database, evaluate the queries and display the query results	4	4	**1. parse source** - uses 3 classes: SourceProcessor, Tokenizer, SimpleParser to perform parsing function **2. design abstraction** - Database class provides "insert" and "get" methods to set and retrieve abstraction (relational) information, supporting SIMPLE parsing and PQL query evaluation **3. evaluate and display queries** - uses QueryProcessor, Tokenize and Database classes to answer queries with return vector of strings
Your solution should comply with the SPA architecture described in the course materials	1	1	All implementation based off startup solution provided and built on that framework to fulfill Iteration 1 requirements. Beside a new SimpleParser class providing recursive descent parsing capabilities, the framework of existing classes are intact and only new methods added to them.
organize your code so that source files and directories clearly correspond to the SPA architecture.	1	1	
Each of the design abstractions must be implemented in separate source files (.cpp), and its public interfaces should be defined in the corresponding header file (.h)	2	2	all classes and methods are placed in existing file structure that corresponds to SPA architecture or Visual Studio Project frame.
Integrate Autotester with your program.	1	1	

1b. Requirements breakdown and implementation matching

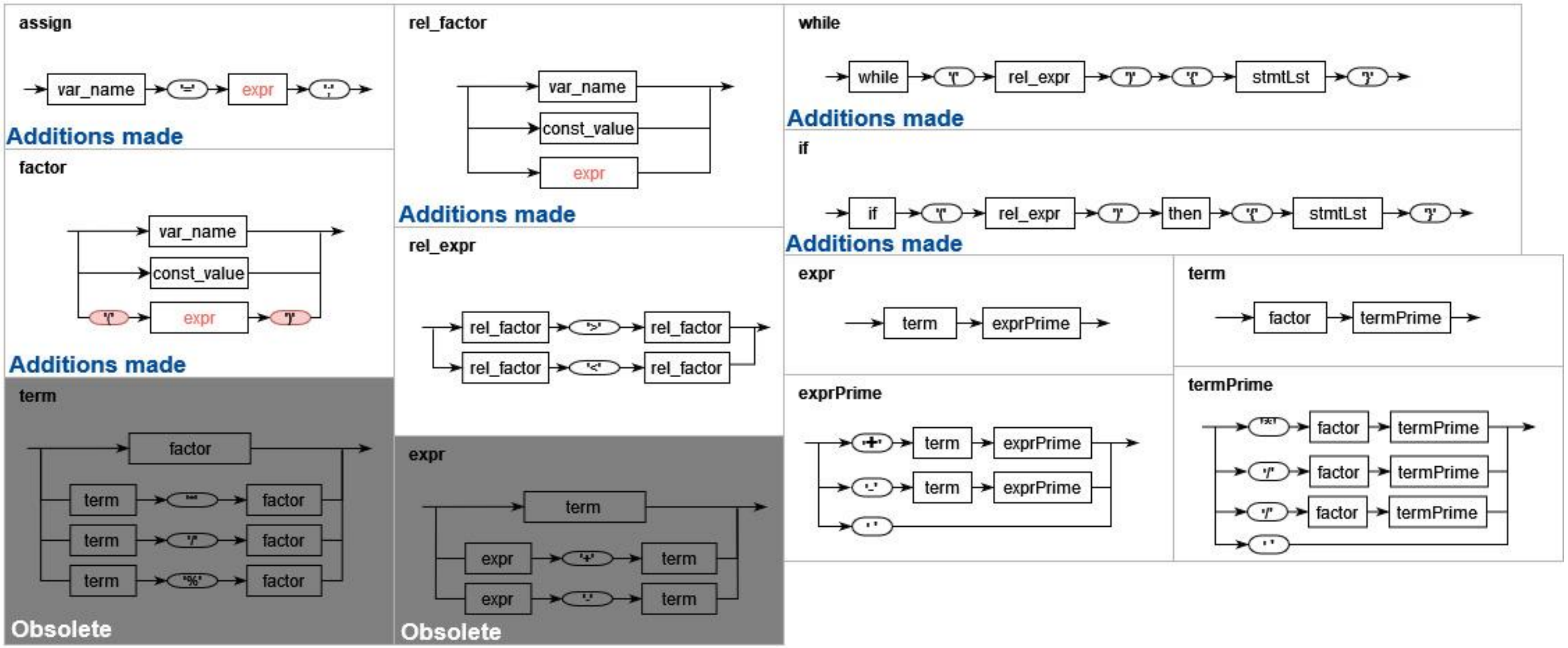
Requirements	Count	Fulfilment	Matching implementation
2. The scope of Iteration 1 (prototype) implementation			
2.1 SIMPLE			
Lexical tokens:			
LETTER : A-Z | a-z -- capital or small letter	1	1	SimpleParser::fetchToken
DIGIT : 0-9	1	1	SimpleParser::fetchToken
NAME : LETTER (LETTER | DIGIT)* -- procedure names and variables are strings of letters, and digits, starting with a letter	1	1	SimpleParser::fetchToken
INTEGER: DIGIT+ -- constants are sequences of digits	1	1	SimpleParser::fetchToken
Grammar rules:			

<table>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td>program: procedure</td><td>1</td><td>1</td><td>SimpleParser::simpleparse</td></tr>
<tr><td>procedure: ‘procedure’ proc_name ‘{‘ stmtLst ‘}’</td><td>1</td><td>1</td><td>SimpleParser::procBlock</td></tr>
<tr><td>stmtLst: stmt+</td><td>1</td><td>1</td><td>SimpleParser::statementList</td></tr>
<tr><td>stmt: read | print | assign | while | if</td><td>1</td><td>1</td><td>SimpleParser::fetchToken,
SimpleParser::statementList</td></tr>
<tr><td>read: ‘read’ var_name;</td><td>1</td><td>1</td><td>SimpleParser::fetchToken,
SimpleParser::statementList</td></tr>
<tr><td>print: ‘print’ var_name;</td><td>1</td><td>1</td><td>SimpleParser::fetchToken,
SimpleParser::statementList</td></tr>
<tr><td>assign: var_name ‘=’ expr ‘;’</td><td>1</td><td>1</td><td>SimpleParser::fetchToken,
SimpleParser::statementList</td></tr>
<tr><td>while: ‘while’ ‘(’ rel_expr ‘)’ ‘{‘ stmtLst ‘}’</td><td>1</td><td>1</td><td>SimpleParser::fetchToken,
SimpleParser::statementList</td></tr>
<tr><td>if: ‘if’ ‘(’ rel_expr ‘)’ ‘then’ ‘{‘ stmtLst ‘}’ ‘else’ ‘{‘ stmtLst ‘}’</td><td>1</td><td>1</td><td>SimpleParser::fetchToken,
SimpleParser::statementList</td></tr>
<tr><td>rel_expr: rel_factor ‘>’ rel_factor | rel_factor ‘<’ rel_factor</td><td>1</td><td>1</td><td>SimpleParser::relExpr, SimpleParser::relFactor</td></tr>
<tr><td>rel_factor: var_name | const_value | expr</td><td>1</td><td>1</td><td>SimpleParser::relFactor</td></tr>
<tr><td>expr: expr ‘+’ term | expr ‘-’ term | term</td><td>1</td><td>1</td><td>SimpleParser::expr, SimpleParser::exprPrime,
SimpleParser::term, SimpleParser::termPrime</td></tr>
<tr><td>term: term ‘*’ factor | term ‘/’ factor | term ‘%’ factor | factor</td><td>1</td><td>1</td><td>SimpleParser::term, SimpleParser::termPrime</td></tr>
<tr><td>factor: var_name | const_value | '(' expr ')'</td><td>1</td><td>1</td><td>SimpleParser::fetchToken</td></tr>
<tr><td>var_name, proc_name: NAME</td><td>1</td><td>1</td><td>SimpleParser::fetchToken</td></tr>
<tr><td>const_value: INTEGER</td><td>1</td><td>1</td><td>SimpleParser::fetchToken</td></tr>
<tr><td>2.2 Database</td><td></td><td></td><td></td></tr>
<tr><td>Program design entities: statement, read, print, while, if, assignment, variable, constant, procedure.</td><td>7</td><td>7</td><td>Database::insertX and Database::getX
where X is the corresponding design entity name: Statement, read, print, assignment, variable, constant, procedure</td></tr>
</table>

<table>
<tr><th>Requirements</th><th>Count</th><th>Fulfilment</th><th>Matching implementation</th></tr>
<tr><td>2.3 PQL</td><td></td><td></td><td></td></tr>
<tr><td>Queries contains only one declaration and one Select clause with a single synonym, at most one such that clause and at most one pattern clause.</td><td>2</td><td>2</td><td></td></tr>
<tr><td>Grammar definition of PQL subset for the prototype:</td><td></td><td></td><td></td></tr>
<tr><td>Lexical tokens:</td><td></td><td></td><td></td></tr>
<tr><td>LETTER: A-Z | a-z -- capital or small letter</td><td>1</td><td>1</td><td>QueryProcessor::evaluate</td></tr>
<tr><td>DIGIT: 0-9</td><td>1</td><td>1</td><td>QueryProcessor::evaluate</td></tr>
<tr><td>IDENT: LETTER (LETTER | DIGIT)*</td><td>1</td><td>1</td><td>QueryProcessor::evaluate</td></tr>
<tr><td>NAME: LETTER (LETTER | DIGIT)*</td><td>1</td><td>1</td><td>QueryProcessor::evaluate</td></tr>
<tr><td>synonym: IDENT</td><td>1</td><td>1</td><td>QueryProcessor::evaluate</td></tr>
<tr><td>Grammar rules:</td><td></td><td></td><td></td></tr>
<tr><td>select-cl: declaration+ ‘Select’ synonym [suchthat-cl | pattern-cl]</td><td>1</td><td>1</td><td>1. QueryParser class added to parse and process multiple synonym declarations and Select clause
2. QueryProcessor::evaluate</td></tr>
<tr><td>declaration: design-entity synonym (‘,’ synonym)* ‘;’</td><td>1</td><td>1</td><td>QueryParser::queryParse</td></tr>
<tr><td>design-entity: ‘stmt’ | ‘read’ | ‘print’ | ‘assign’ | ‘variable’ | ‘constant’ | ‘procedure’</td><td>1</td><td>1</td><td>QueryProcessor::evaluate</td></tr>
</table>

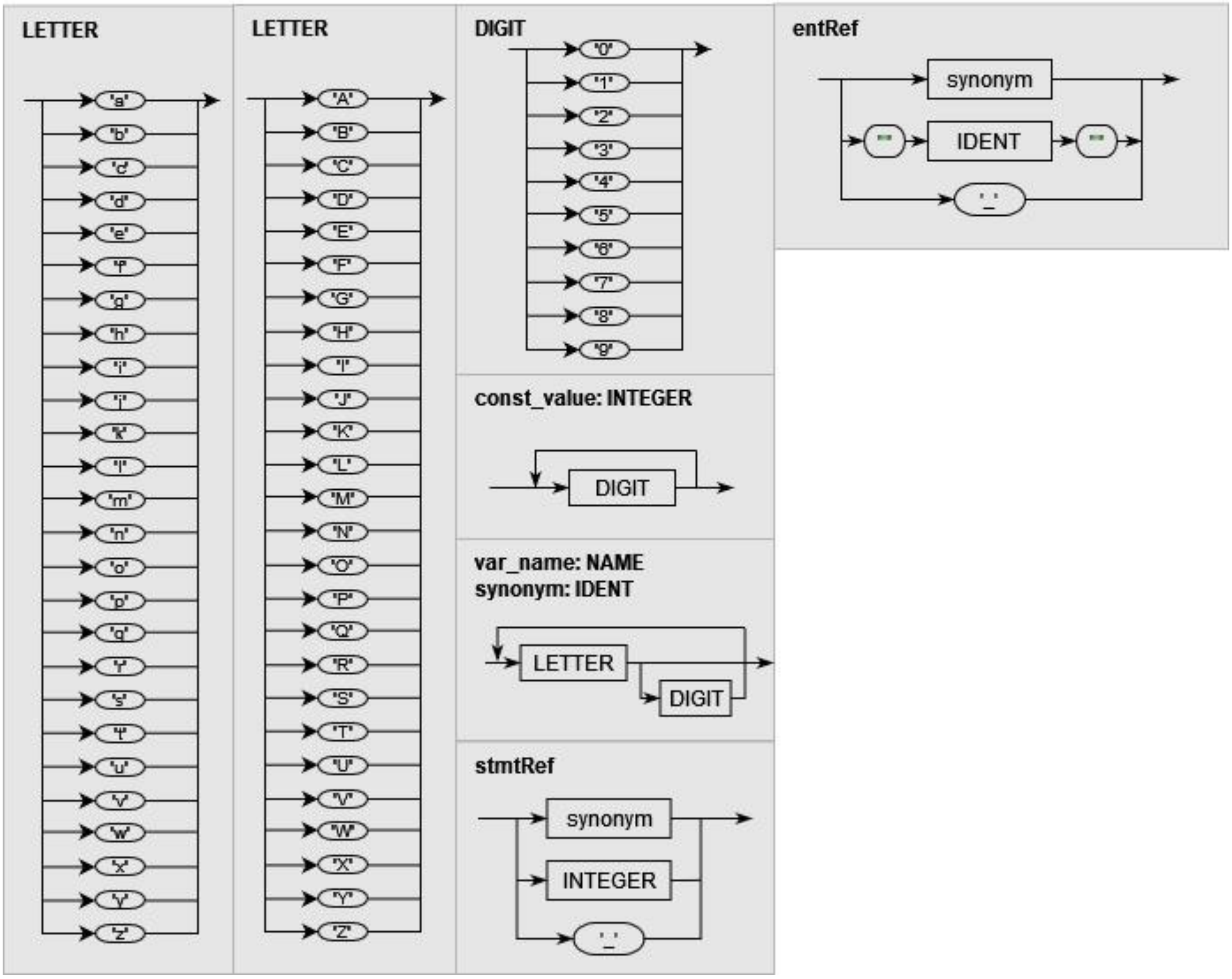
2. Syntax Diagram

program: procedure+ **procedure**: 'procedure' proc_name '{' stmtLst '}' **stmtLst**: stmt+ **stmt**: read | print | assign | while | if **print**: 'print' var_name; **read**: 'read' var_name';' **assign**: var_name '=' expr ';' **while:** 'while' '(' rel_expr ')' '{' stmtLst '}' **if**: 'if' '(' rel_expr ')' 'then' '{' stmtLst '}' 'else' '{' stmtLst '}' **factor**: var_name | const_value | '(' expr ')' **rel_expr:** rel_factor '>' rel_factor | rel_factor '<' rel_factor **rel_factor**: var_name | const_value | expr	**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 **LETTER** : A-Z | a-z -- capital or small letter **DIGIT** : 0-9 **var_name**, **proc_name**: NAME **const_value**: INTEGER **Left recursion elimination** **expr**: expr '+' term | expr '-' term | term **expr: term exprPrime** **exprPrime: '+' term exprPrime | '-' term exprPrime | ε** **term**: term '*' factor | term '/' factor | term '%' factor | factor **term: factor termPrime** **termPrime: '*' factor termPrime | '/' factor termPrime | '%' factor termPrime | ε**



LETTER : A-Z | a-z -- capital or small letter
DIGIT : 0-9
IDENT: LETTER (LETTER | DIGIT)*
NAME : LETTER (LETTER | DIGIT)*
INTEGER: DIGIT+ -- constants are sequences of digits

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



select-cl: declaration+ 'Select' synonym [suchthat-cl | pattern-cl]*

declaration: declaration : design-entity synonym (',' synonym)* ';'
design-entity: 'stmt' | 'read' | 'print' | 'assign' | 'while' | 'if' | 'variable' | 'constant' | 'procedure'

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

suchthat-cl: '**such that**' relRef
relRef: ModifiesS | UsesS | Parent | ParentT |

Parent: 'Parent' '(' **stmtRef** ',' **stmtRef** ')'
ParentT: 'Parent*' '(' **stmtRef** ',' **stmtRef** ')'

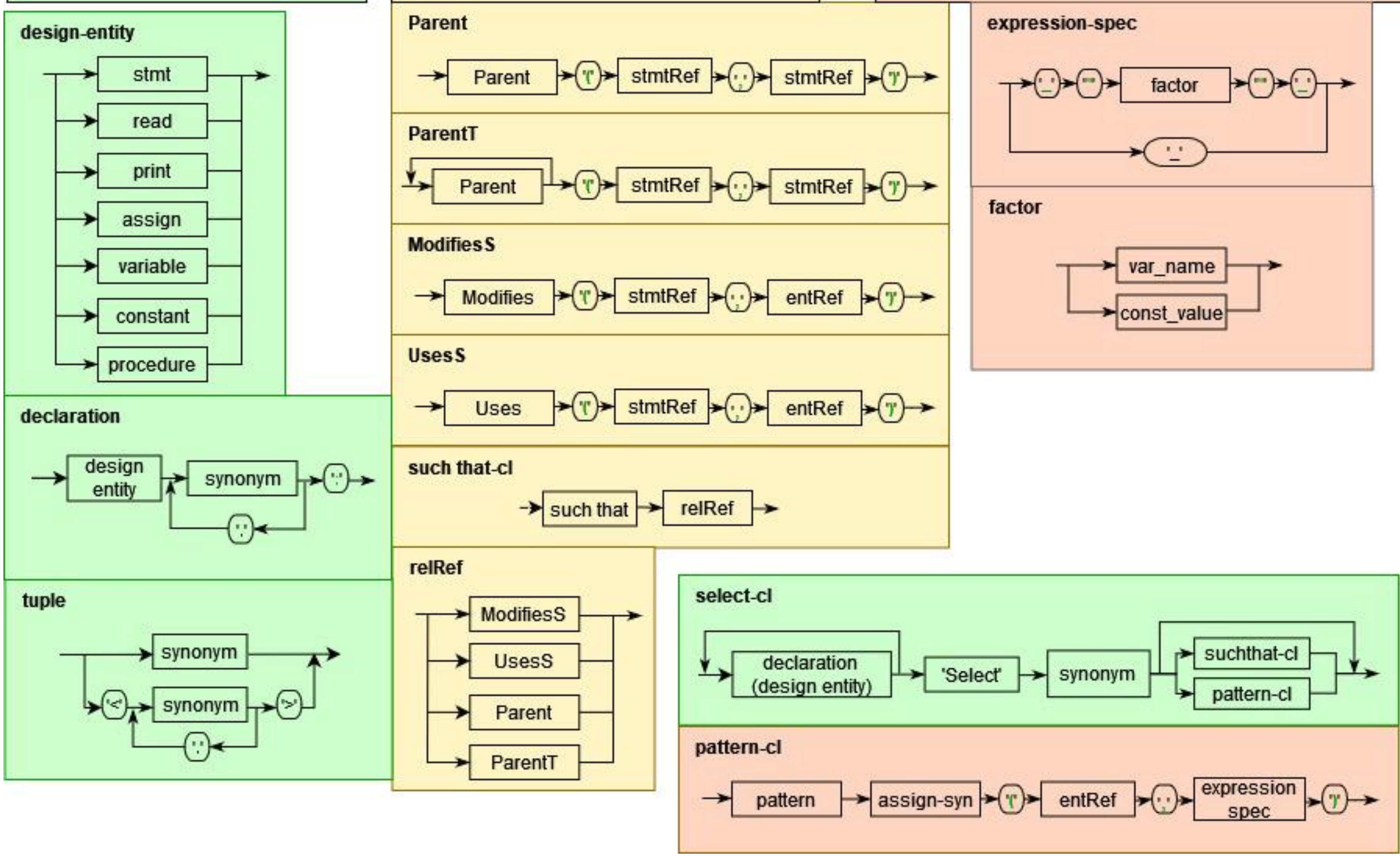
ModifiesS: 'Modifies' '(' **stmtRef** ',' **entRef** ')'
UsesS: 'Uses' '(' **stmtRef** ',' **entRef** ')'

pattern-cl: **'pattern'** syn-assign '('entRef ',' expression-spec ')'
// syn-assign must be declared as synonym of assignment (design entity 'assign').

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

expression-spec: '_' '"' **factor** '"' '_' | '_'

factor: **var_name** | **const_value**
var_name: **NAME**
const_value: **INTEGER**



3. SIMPLE statement relational tables

The sample SIMPLE program provided is used to illustrate the mechanics of SimpleParse class storing statement relational information and replicated below. Text Formatting and line numbering are added on the SIMPLE statements to aid understanding of design rationale.

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

Figure A3.1 Sample SIMPLE program with line and block number

3a. 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*

To answer Parent/Parent* queries, it is necessary to identify statements and statement blocks. As illustrated in Figure A3.1 blocks of SIMPLE statements can be identified and differentiated. The resulting statement relational table is created below.

stmnt	
StmtLine[PK]	Parent
1	0
2	0
3	0
4	0
5	0
6	5
7	5
8	5
9	5
10	0
11	10
12	10
13	10
14	0
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0

The statement line identifier is selected as primary key[PK] and leverages on the property that PKs have database constraints of UNIQUE and NOT NULL. This prevents duplicated entries on line identifier. Identifying blocks allow SimpleParser class to differentiate parents of each statement list in the SIMPLE grammar, thus, resulting in the Parent column.

To illustrate, statements 6-9 has statement 5 as parent. This, in turn, answers the PQL query *stmt s; Select s such that Parent (s , 6)* via Database class method Database::getSynLine_Parent

Other methods developed to handle variations in StmtRef and ss are:

Database::getSynSyn_Parent
Database::getSynSyn_Parent_Read
Database::getSynSyn_Parent_Print
Database::getSynSyn_Parent_Assign
Database::getSynLine_Parent
Database::getLineSyn_Parent
Database::getLineLine_Parent
Database::getWildSyn_Parent
Database::getWildLine_Parent

3b. Uses/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.*

assign

assignLine[PK]	UAV	MAV
1	0	count
2	0	cenX
3	0	cenY
6	count	count
7	cenX	cenX
8	cenY	cenY
11	0	flag
12	cenX, count	cenX
13	cenX, count	cenY
14	cenX, cenY	normSq
16		flag

In the form Uses (a, v), information can be stored in the assign table where assignment statements are recorded. Using the statement identifier as primary key[PK] prevents duplicated entries. Uses (a, v) is abbreviated to form the column UAV where variables appearing on the right-hand side (RHS) of an assignment statement is stored.

The corresponding Database class methods to query tables for Uses(a,v)queries are:

Database::getSynSyn_UAV_Stmt
Database::getSynSyn_UAV_Assign
Database::getSynCount_UAV_Assign
Database::getSynCount_UAV_Stmt

variable

variableName[PK]	USV	MSV
count	10	
cenX		12
cenY		13
x	5	5
y	5	5
flag		10
normSq		

The same rationale can be applied for the forms Uses(pn,v), Uses(s,v) and Uses(p,v) creating columns UPNV in print table, USV in variable table and UPV in ProcBlock table respectively.

print

printLine[PK]	UPNV
15	cenY
21	flag
22	cenX
23	cenY
24	normSq

ProcBlock

LineNo[PK]	UPV	MPV	BlockID[FK] REFERENCES procedures(procedureID)
6	count	count	1
7	cenX, x	cenX	1
8	cenY, y	cenY	1
⋮	⋮	⋮	⋮

3c. Modifies/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.*

assign

assignLine[PK]	UAV	MAV
1	0	count
2	0	cenX
3	0	cenY
6	count	count
7	cenX	cenX
8	cenY	cenY
11	0	flag
12	cenX, count	cenX
13	cenX, count	cenY
14	cenX, cenY	normSq
16		flag

The Modifies (a,v) form requires assignment statements identified along with the involved variable. This can be done by recording parsed information in the assign table, using assignLine column to identify the statement and column MAV to record the variables.

The above rationale can be extended to read table for Modifies(r,v) form.

In the Modifies(s,v) form, the variable table has to be used in conjunction with the stmt-parent table to provide an answer.

Database methods implemented to handle Modifies queries are:

Database::getSynAssign_MAV
Database::getSynAssign_MAV_Varname
Database::getLineNoAssign_MAV
Database::getSynRead_MRV
Database::getSynRead_MRV_Varname
Database::getLineNoRead_MRV
Database::getSynStmt_MSV
Database::getSynStmt_MSV_Varname
Database::getLineNoStmt_MSV

variable

variableName[PK]	USV	MSV
count	10	
cenX		12
cenY		13
x	5	5
y	5	5
flag		10
normSq		

read

readLine[PK]	MRV
19	x
20	y

ProcBlock

LineNo[PK]	UPV	MPV	BlockID[FK] REFERENCES procedures(procedureID)
6	count	count	1
7	cenX, x	cenX	1
8	cenY, y	cenY	1
⋮	⋮	⋮	⋮