Logo, company name

Description automatically generated  
  
SPA Project Report   
Iteration 2

Muhammad Naufal Dusan Urosevic, [e0657867@u.nus.edu](mailto:e0657867@u.nus.edu)

Chan Kong Yew, A0227199R [e0650692@u.nus.edu](mailto:e0650692@u.nus.edu)

Contents

[1. Scope of the prototype implementation 3](#_Toc875540400)

[2. SPA design 3](#_Toc1176721134)

[2.1 Overview 3](#_Toc1978752934)

[2.2 Design of SPA components 4](#_Toc1599532658)

[Handling Expressions 5](#_Toc1426479515)

[Design of Database 7](#_Toc1844235222)

[Query Processor 8](#_Toc1898807322)

[3. Conclusion 10](#_Toc153151644)

[Reference 10](#_Toc1417432127)

[Appendix 11](#_Toc373809714)

[1a. Requirements breakdown and implementation matching 11](#_Toc551228226)

[1b. Requirements breakdown and implementation matching 11](#_Toc1059240800)

[2. Syntax Diagram 13](#_Toc1245634757)

[3. SIMPLE statement relational tables 16](#_Toc1506673055)

[3a. Parent/Parent\* 17](#_Toc58640483)

[3b. Uses/UseS 18](#_Toc864724966)

[3c. Modifies/Modifies\* 19](#_Toc1211289935)

# 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.

Diagram

Description automatically generated

## 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.

Graphical user interface

Description automatically generated

### **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.

Diagram, table

Description automatically generated

### **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.

Diagram

Description automatically generated

### **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.

Diagram

Description automatically generated

# 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:** |  |  |  |
| program: procedure | 1 | 1 | SimpleParser::simpleparse |
| procedure: ‘procedure’ proc\_name ‘{‘ stmtLst ‘}’ | 1 | 1 | SimpleParser::procBlock |
| stmtLst: stmt+ | 1 | 1 | SimpleParser::statementList |
| stmt: read | print | assign | while | if | 1 | 1 | SimpleParser::fetchToken, SimpleParser::statementList |
| read: ‘read’ var\_name; | 1 | 1 | SimpleParser::fetchToken, SimpleParser::statementList |
| print: ‘print’ var\_name; | 1 | 1 | SimpleParser::fetchToken, SimpleParser::statementList |
| assign: var\_name ‘=’ expr ‘;’ | 1 | 1 | SimpleParser::fetchToken, SimpleParser::statementList |
| while: ‘while’ ‘(’ rel\_expr ‘)’ ‘{‘ stmtLst ‘}’ | 1 | 1 | SimpleParser::fetchToken, SimpleParser::statementList |
| if: ‘if’ ‘(’ rel\_expr ‘)’ ‘then’ ‘{‘ stmtLst ‘}’ ‘else’ ‘{‘ stmtLst ‘}’ | 1 | 1 | SimpleParser::fetchToken, SimpleParser::statementList |
| rel\_expr: rel\_factor ‘>’ rel\_factor | rel\_factor ‘<’ rel\_factor | 1 | 1 | SimpleParser::relExpr, SimpleParser::relFactor |
| rel\_factor: var\_name | const\_value | expr | 1 | 1 | SimpleParser::relFactor |
| expr: expr ‘+’ term | expr ‘-’ term | term | 1 | 1 | SimpleParser::expr, SimpleParser::exprPrime, SimpleParser::term, SimpleParser::termPrime |
| term: term ‘\*’ factor | term ‘/’ factor | term ‘%’ factor | factor | 1 | 1 | SimpleParser::term, SimpleParser::termPrime |
| factor: var\_name | const\_value | '(' expr ')' | 1 | 1 | SimpleParser::fetchToken |
| var\_name, proc\_name: NAME | 1 | 1 | SimpleParser::fetchToken |
| const\_value: INTEGER | 1 | 1 | SimpleParser::fetchToken |
| **2.2 Database** |  |  |  |
| Program design entities: statement, read, print, while, if, assignment, variable, constant, procedure. | 7 | 7 | Database::insertX and Database::getX where X is the corresponding design entity name: Statement, read, print, assignment, variable, constant, procedure |

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

2. Syntax Diagram

Graphical user interface

Description automatically generated with medium confidence

Table

Description automatically generated

Diagram

Description automatically generated with low confidence

## 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.



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.



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



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*

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.







## 3c. Modifies/Modifies\*





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



