Logo, company name

Description automatically generated  
  
SPA Project Report   
Iteration 3

Team 19

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. Overview and Objective 3](#_Toc99913398)

[2. SPA design 3](#_Toc99913399)

[2.1 Overview 3](#_Toc99913400)

[2.2 Design of SPA components 3](#_Toc99913401)

[**Source Processor** 4](#_Toc99913402)

[**Handling Expressions** 4](#_Toc99913403)

[**Design of Database** 7](#_Toc99913404)

[**Query Processor** 8](#_Toc99913405)

[**Handling Pattern Searches** 8](#_Toc99913406)

[3. Conclusion 10](#_Toc99913407)

[Reference 10](#_Toc99913408)

[Appendix 11](#_Toc99913409)

[1a. Requirements breakdown and implementation matching 11](#_Toc99913410)

[1b. Requirements breakdown and implementation matching 12](#_Toc99913411)

[2. Syntax Diagram 14](#_Toc99913412)

[3. SIMPLE statement relational tables 17](#_Toc99913413)

[3a. Parent/Parent\* 18](#_Toc99913414)

[3b. Uses/UseS 19](#_Toc99913415)

[3c. Modifies/Modifies\* 20](#_Toc99913416)

[Appendix 4 Test results using Iteration 2 test cases **Error! Bookmark not defined.**](#_Toc99913417)

# 1. Overview and Objective

This report describes the design and results of the SPA project implementation over 3 iterations. Project schedule follows the guidelines given in Wiki.

# 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 method in the SimpleParser class with the simpleparse method corresponding to “program” in SIMPLE. 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 matches and identify a segment of tokens against SIMPLE statement types, e.g., read, print and assign. 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 next diagram shows how SimpleParse methods interact with the Database class to create tables to hold relationships within and between statements found 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

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 uses the queryParse method to parse the incoming query, recognizing PQL tokens as it appears from the query token string. Methods in QueryParser class corresponds to PQL grammar component names, for e.g., such that clause is processed by class method suchThat() and design entity Modifies is handled by method modifies().

Each design entity method labels and stores stmtRef and entRef as a ‘RHS’ or ‘LHS’ string according to its position defined by PQL grammar rules. For example, Uses() method stores ‘stmtRef’ as LHS string and entRef as RHS string. Uses() then further breaks down LHS and RHS string, calling methods from Database class to form SQL queries that retrieves information off tables created from SimpleParser. 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 using stmt table to find the parent of statement 5.

### **Handling Pattern Searches**

The pattern() method handles searches within assignment statement expressions. Following pattern grammar rule, entRef is labelled and stored as LHS whereas expression-spec is labelled and stored as RHS. Expressions in the pattern table, stored from parsing the SIMPLE source, are first converted from infix to postfix before any matching performed through isSubtree() method.

Graphical user interface, table

Description automatically generated

# 3. Conclusion

The maturity of this SPA project is guaged using pass rates on iteration 2 test cases. The project schedule slipped from iteration 2 and did not recover sufficiently to meet all the iteration 3 requirements. With the release of test cases from Iteration 2, time in Iteration 3 were spent on (1) redesign of QueryParser class with a better understanding of PQL from iteration 2 results (2) completion of pattern matching feature due from iteration 2 and (3) debugging and passing test cases from iteration 2, ensuring SPA exits gracefully on different error conditions.

In summary, while not all features are delivered, the remedies from iteration 2 review have improved the stability and architecture of the SPA software which provides a better foundation to add on iteration 3 features, if given time.

# 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, text, application, email

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

## 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



