SPA Project Report Iteration 1

Done by Team 19 :

Muhammad Naufal Dusan Urosevic,A0227505J, 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.

2. SPA design

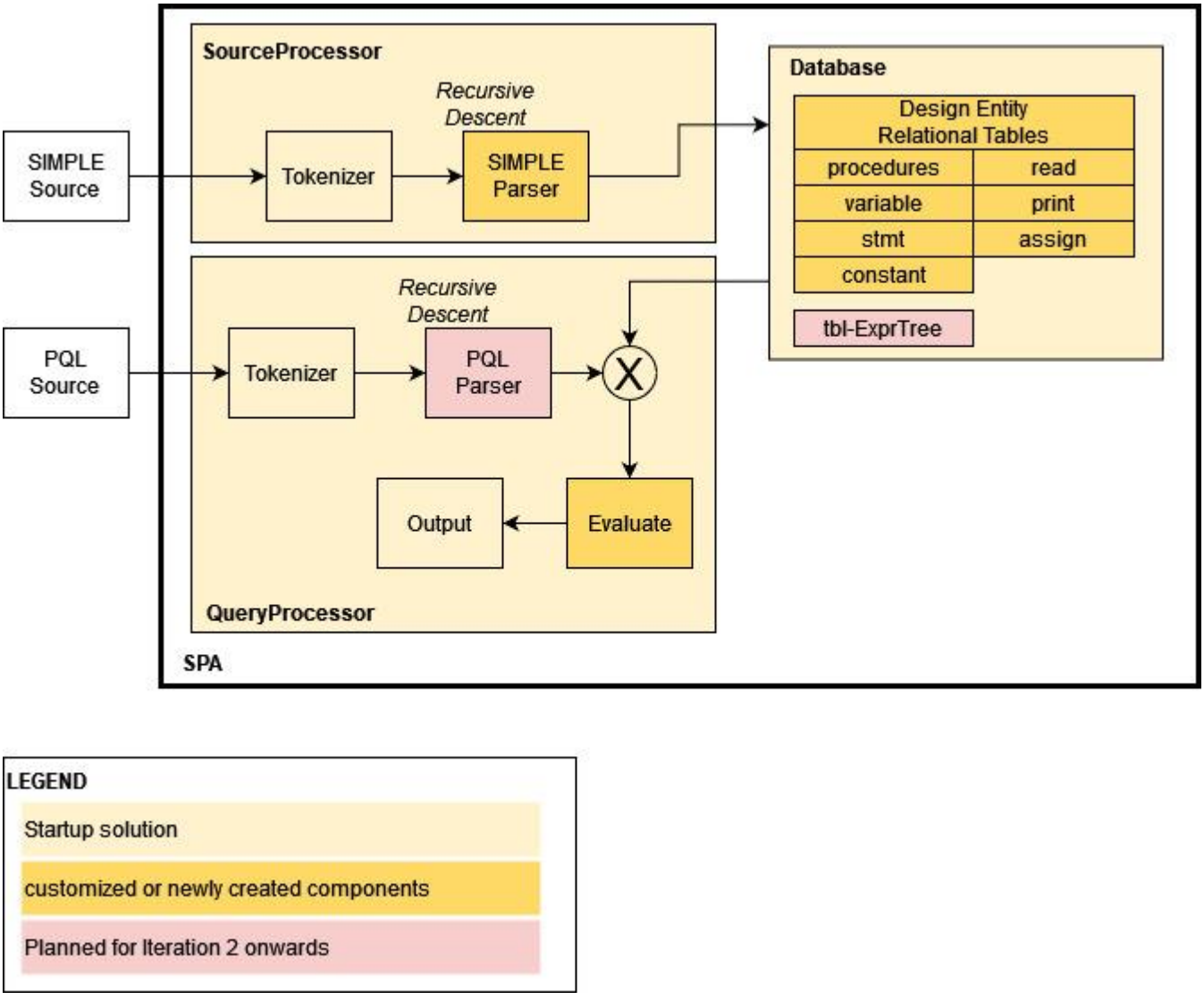
2.1 Overview

The system diagram below shows the SPA design leveraging off the Startup Solution provided. 3 major components in place are: SourceProcessor, QueryProcessor and Database. To handle Iteration 1, the 3 C++ classes are customized according to the next section.

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.

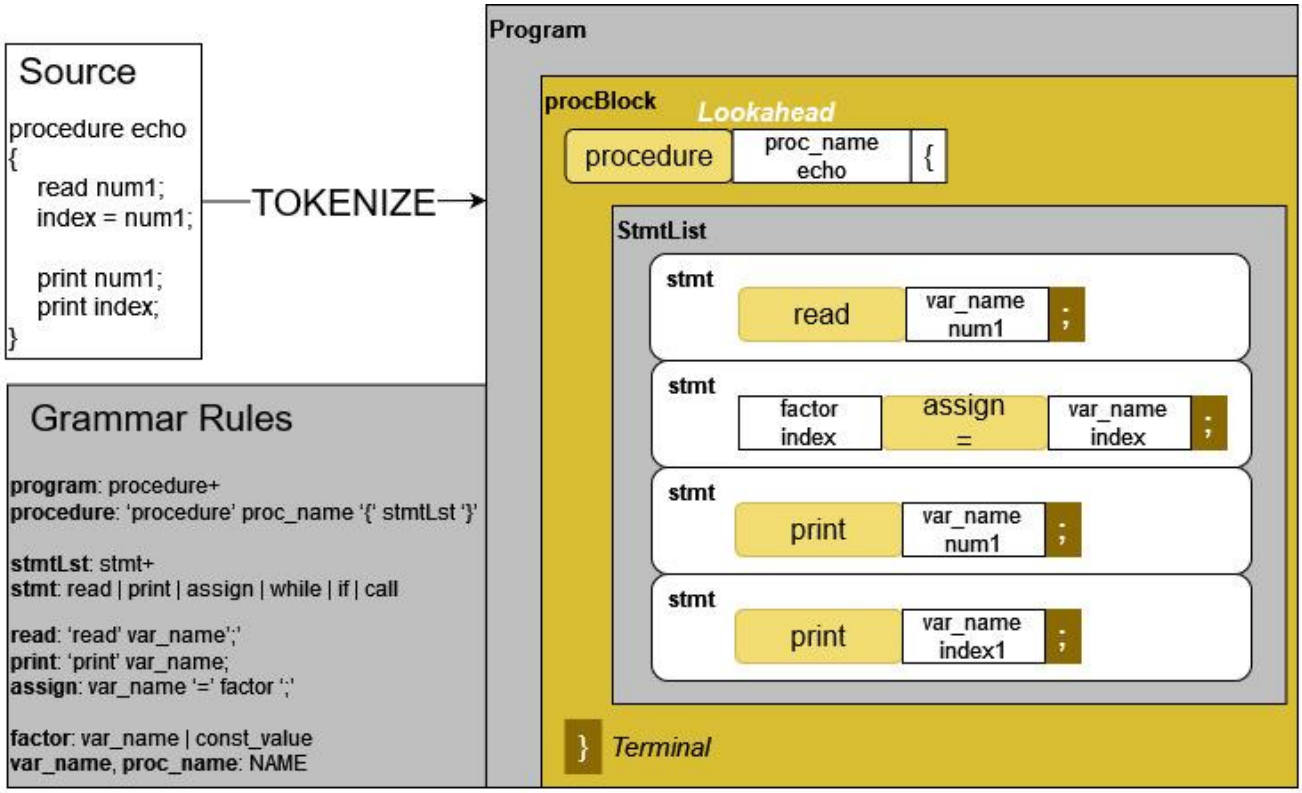


Design of Database

The database(DB) is made up of 7 tables intended to depict the relationship between individual components of a SIMPLE program. The SELECT clause, when used exclusively, returns the identifying names or existence of procedures, variables, constants and statements(assignment, print, read and line location on program). Therefore, the 7 tables are able to answer these queries.

DB Initialization

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



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

DB Insert methods

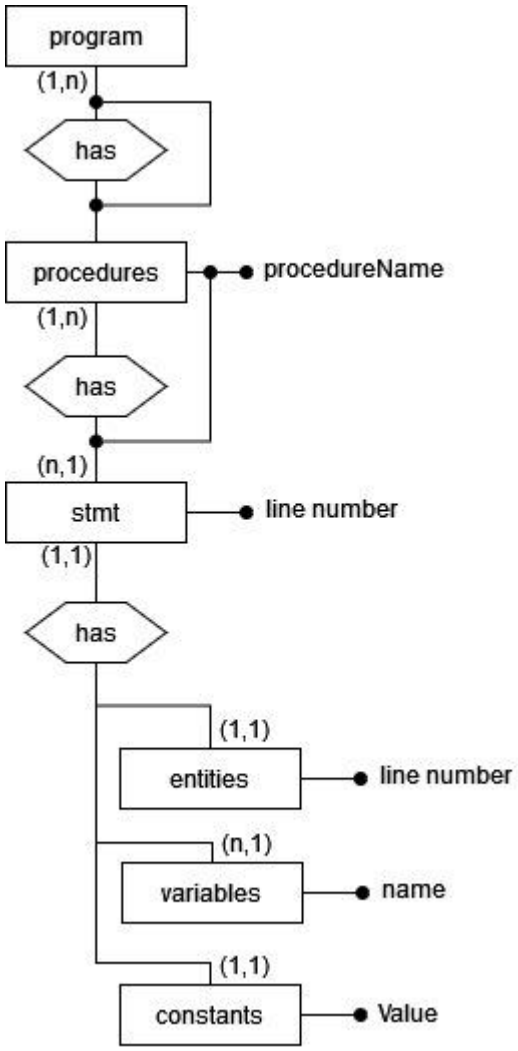
As the source SIMPLE program is parsed, SQL INSERT INTO statements embedded in different insert methods, e.g. insertStatement, allows relational information to be written into the DB

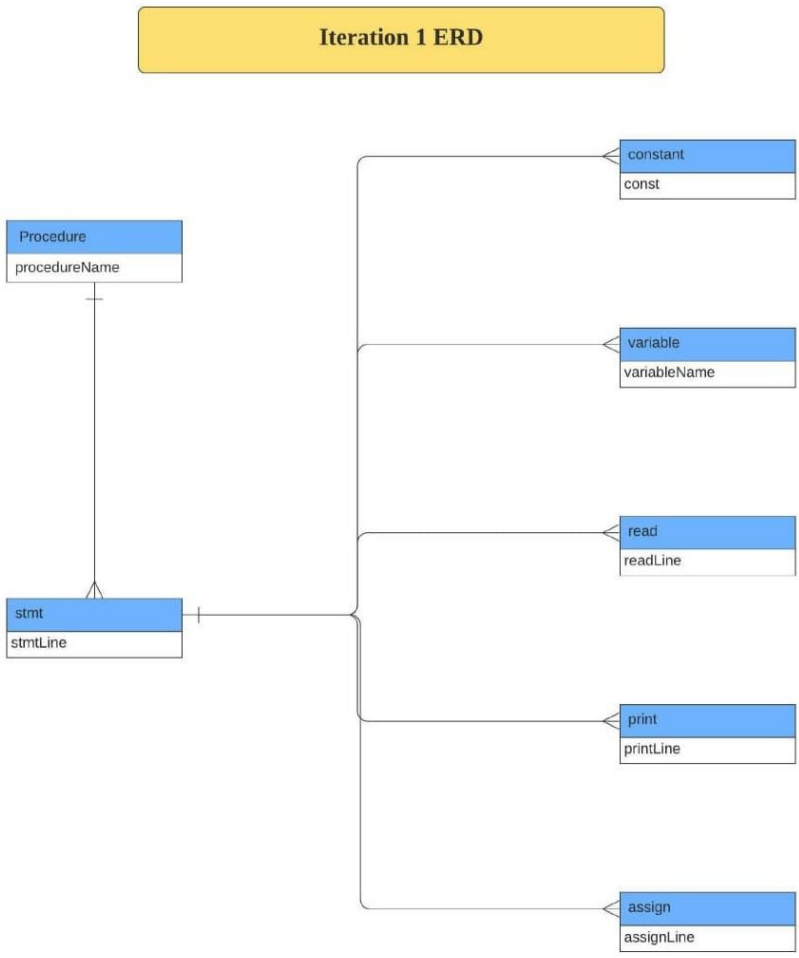
DB get methods

As queries are evaluated, SQL SELECT statements in different get methods, e.g. getProcedures, retrieves the information necessary to generate query results.

Query Processor

Riding on the assumption that syntactic errors do not exists in Iteration 1, synonym declarations can be safely presumed to occur before a SELECT clause with only a single synonym in that SELECT clause. The sample code can correctly identify the token positions for processing with “if” constructs acting like a switch-case statement to evaluate queries. Hence, the parser development is postponed to Iteration 2.





3. Testing

Sample SIMPLE program and Query files with the Startup Solution was the main source of test cases. The files can cover Iteration 1 requirements and, therefore, suffice for purpose.

Additional consideration was given to testing duplicated NAME entities and this lead to the DB table designs to select procedure or variable names as the primary key(PK) for the respective tables. Since PKs holds inherently both characteristics unique and not null in DB, duplicate entries are naturally rejected by the system.

4. Conclusion

Iteration 1 solution was done in accordance to requirements and based off the Startup Solution. Designs for the 3 major components: SourceProcessor, Database and QueryProcessor were also made with a view for expansion to meet Iteration 2 and 3 demands. Thus, Iteration 1 can be successfully closed.

The team is has already begun design work and feasibility studies for Iteration 2.

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	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 '=' factor ';'	1	1	SimpleParser::fetchToken, SimpleParser::statementList
factor: var_name | const_value	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, 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
2.3 PQL			
Queries contains only one declaration and one Select 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	1	1	1. no Query Parser since Iteration 1 is based on (a) entry syntax is always valid and (b) only one SELECT clause and one definition 2. QueryProcessor::evaluate
declaration: design-entity synonym ';'	1	1	QueryProcessor::evaluate
design-entity: 'stmt' | 'read' | 'print' | 'assign' | 'variable' | 'constant' | 'procedure'	1	1	QueryProcessor::evaluate

2. Syntax Diagram

program: procedure+ **procedure**: 'procedure' proc_name '{' stmtLst '}' **stmtLst**: stmt+ **stmt**: read | print | assign **read**: 'read' var_name';' **print**: 'print' var_name; **assign**: var_name '=' factor ';' **factor**: var_name | const_value	**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