TQL

(Text Query Language)

Practical Work II

Subject

Processing Languages

Teacher

Alberto Simões

Student

João Rodrigues, 16928

LESI

Summary

Present report is meant to document an academic work.

The project is intended to implement a query language, to do same, or almost same, functions as SQL. The base of it, are csv files and language processing, and the fictional name for it, is **TQL (Text Query Language)**.

Index

Summary 2
Index 3
Contextualization 5
Topic Picked 5
Approach 5
Lexer -> Analysis of tokens 6
Grammar -> Analysis of grammar 6
Eval -> Evaluates expressions 6
Main -> Program 6
Implementation 7
Lexer 7
Operations 7
Syntax 7
Comparison_ch 8
Literals 8
t_ignore 8
File 8
Var 8
Nr 9
IntNr 9
BE 9
LE 9
DIFFERENT 9
Grammar 9
Prompt 9
Command 10
Query 10
Query_list 10
Load 10
Discard 10
Save 10
Show 11
Select 11

Call 11
Procedure 11
Columns 11
VarColumns 11
Param_lim 12
Param_where 12
Comparisons 12
Booleans 12
Bool_Op 12
CreateSource 13
Join 13
Error 13

Contextualization

As mentioned on summary, this project has academic purposes, having origin on a practical work to be evaluated on the 2nd year subject, **Languages Processing**, from the course LESI – Bachelor on Computer Systems Engineering.

Project consists on applying content taught on the subject's class, where the student should, mainly, be able to create regular expressions that adapt to the situation in hands, and apply them on a lex compiler, which was also applied on the first project, as well as apply grammar rules and evaluation of obtained grammar. Besides that, the program is meant to be developed in Python, to do operations like database languages allow, with a syntax close to SQL.

The work was given from the subject's teacher, Alberto Simões, on the current lective year of 2020/2021, and enunciation is available on this project's git, in Portuguese language.

Topic Picked

It was possible to choose between 2 themes, with the same objective, in terms of evaluation, approaching topics mentioned on the contextualization.

Theme A consists of converting pseudocode into C code and/or execute pseudocode.

Theme B consists on TQL (Text Query Language), which is the theme picked for this project, which was already exposed its objective.

Approach

The process to develop the project, passed through study the needs on each file by order. The files were established based on what was learned on classes, being theme:

- Lexer
- Grammar
- Eval
- Main

Lexer -> Analysis of tokens

A token is a set of characters, just like a string. It's on this file, where is defined that mentioned token, which is part of our Language, and is where we can categorize it / insert on a group which has something in common, like being an operation.

Grammar -> Analysis of grammar

Grammar defines valid expressions of our Language, just like a sentence on a text. It's here where is determined the order tokens can assume, to have a valid expression.

Eval -> Evaluates expressions

Gives a meaning to the expressions and acts according to it. This mean, this file is responsible to connect the functions to the sentence/expression received and apply that right function with the inputted arguments.

Main -> Program

It's here that is defined the order to apply the functionalities of each file. Main, is the program, is the file containing the script to run.

Now, this order is basically the order in which was aborded the project, with only having in mind that, being main, the file to put the script to run, this file is created to test methods from each file.

Also to mention that, although there's an order, this doesn't mean that after implementing 1 file, we can't go back, as wasn't used Waterfall model for software development, but an agile model, since some files would need to change the type of a token, for example.

Each file was studied with the help of GitHub, to have a more organized and easier way to access information and tasks, to do that study.

Implementation

For this chapter, it's going to be used same process as in 'Approach', which is go file – by – file. As in that previous chapter, it was defined the function of each one, the explanation for this chapter, will consist more on the actual components used and how and why are used on the project's context.

Lexer

The objective here, was to establish the terms we could have and distinguish the ones that are part of the actual Language, as syntax, so they're reserved to it, and cannot be used as a variable from the ones that are not, so they are variables, and can assume various values; and also, inside those reserved strings, distinguish which was each, and group them.

Operations

The first token, which seemed more obvious, was operations, which are the type of queries we want to have. For this group, it was included the follow elements, as strings:

- LOAD
- DISCARD
- SAVE
- SHOW
- SELECT
- CREATE
- PROCEDURE
- CALL

Syntax

On this group, is stored the other terms that are reserved, but are not operations. Are part of this group, the following elements:

- AS
- FROM
- WHERE
- DO
- JOIN
- TABLE
- USING
- LIMIT
- END
- AND
- OR

Comparison_ch

This, defines elements on 1 character that are used to compare values. Are part of this group, the following elements:

- >
- <
- =

Literals

Define 1 character elements, used for operations. This group is formed by elements of **Comparison_ch** and the following ones:

- *
- ;
- ,
- (
-)

t_ignore

Tokens that can be used but do not affect expressions. Elements:

- Space -> ‘ ’
- Newline -> \n

File

Describes a string between quotation marks, which only contains, valid values, for a filename. It has 2 expressions: one for files on current directory, and the other for navigating through directories.

Regular Expression : "\.(\/[^<>:\"\/\|?*]+)+" | "[^<>:\"\/\|?*]+"

Var

Defines a variable to be used, being it a column name or a table name. As so, there’s no quotation marks, and characters are restricted to possible ones, used on SQL. There’s 2 expressions, so ‘_’ character is not used more than once, consecutively, at beginning of the string.

Regular Expression: _[a-zA-Z0-9_@$]+ | [a-zA-Z][a-zA-Z0-9_@$]*

Nr

Describes general numbers used, allowing floats and ints.

Regular Expression: [0-9]+(\.[0-9]+)?

IntNr

Describes only integer numbers.

Regular Expression: [0-9]+

BE

Defines Bigger or Equal.

Regular Expression: >=

LE

Defines Less or Equal.

Regular Expression: <=

DIFFERENT

Defines different/ inequality symbol.

Regular Expression: <>

These last 3 tokens, are comparison operators, which are defined individually, since they have more than 1 character.

Grammar

The process used here, was to, first, observe if was possible to group various queries on 1 group, which ended up being unworthy, so there's 1 grammar rule, at least, for each query. Beside that, it was about checking the forms, each query could have and try to adapt it into grammar. Next, will be presented each grammar rule, written on same line, opposed to programming syntax.

Prompt

That's the initial state, for grammar, this is the full input received. For this, we can have a single command, or various.

Grammar Rule: prompt : prompt ';' command | command

Command

This rule was mentioned in the rule before; a command can be a normal query or a procedure. Reason being, for not adding procedure on queries, is the need to have queries on a procedure, and we don't want a procedure inside another procedure.

Grammar Rule: command : query | procedure

Query

A query is a command, a query is an operation on database, and simply defines the possible operations to use. Each query rule, has a 'q_' attached to it, to differentiate the operation from the query's definition.

Grammar Rule: query : q_load | q_discard | q_save | q_show | q_create | q_select | q_call

Query_list

As the name suggests, this defines list of queries. This rule is only used on procedures.

Grammar Rule: query_list : query_list ';' query | query

These, are the top-level grammar rules, so now it's all about queries and procedures. Although each operation (being it query or procedure), has the same functionality as it would in SQL, there will be an explanation on what it does, for someone not familiarized with SQL. Let's start with the queries.

Load

Loads a csv file table into memory as a DataFrame, and stores it on a list, so loaded tables are accessible.

Grammar Rule: q_load : LOAD TABLE var FROM file

Discard

Discards a table from memory, in other words, it deletes it from there. In order to do more interactions on that table, it must be loaded again.
Grammar Rule : p_discard : DISCARD TABLE var

Save

Stores a table in memory, to a csv file. The user may choose the name to give to the file.

Grammar Rule: p_save : SAVE TABLE var AS file

Show
Shows a preview of values on a table.

Grammar Rule: p_show : SHOW TABLE var

Select
Like SHOW, it also previews values of a table, but with the particularity of adding parameters of comparison operations and limiting the number of records shown.

Grammar Rule: p_select : SELECT columns FROM var param_where param_lim

Call
Executes a procedure.

Grammar Rule: p_call : CALL var

Now, for procedure.

Procedure
Creates like a macro, this means, that we can aggroup a lot of queries and give a name to this 'script', to run it later, anytime we want it. In this syntax is not being considered indentation (\n and \t).

Grammar Rule: p_procedure : PROCEDURE var DO query_list ';' END

Some grammar rules simply used tokens, straightforward, while others called other reserved names, which are other grammar rules.

On p_select, we had 3: columns, param_where, param_lim, as the syntax was:
" p_select : SELECT columns FROM var param_where param_lim "

Columns
Determines how to choose columns, which allows to have a '*' character, which in SQL, represents all columns, or explicit columns, being able to be 1 or various.

Grammar Rule: p_columns : '*' | var_columns

VarColumns
Allows to choose between 1 column or various, using recursion.

Grammar Rule: var_columns : var_columns ',' var | var

Param_lim

Limits the maximum number of records to be shown. There's the option of empty, in case of no limit, and if there's a limit, it must be an integer number.

Grammar Rule: param_lim : LIMIT IntNr |

Param_where

Creates comparisons of columns' values. There's the option of empty, in case of no restrictions of this type are expressed.

Grammar Rule: param_where : WHERE COMPARISON |

Comparisons

Establish possible comparisons to be used on WHERE. The presentation of the rule, is in a way to be more readable.

Grammar Rule: COMPARISON : var '>' nr | var '<' nr | var BE nr | var LE nr | B

| var '=' nr | var '=' file | var DIFFERENT nr | var DIFFERENT file

This B, allows to have Boolean comparisons, between these values' comparisons.

Booleans

Makes AND and OR operations between comparisons defined on p_comparisons. This uses COMPARISON, which allows to have multiple Boolean conditions, by using B there.

Grammar Rule: B : COMPARISON Bool_Op COMPARISON | '(' COMPARISON Bool_Op

COMPARISON ')'

Bool_Op

Defines the operations used on Booleans, which are AND and OR. This could also be defined on lexer, if was created a group just for these 2.

Grammar Rule: Bool_Op: AND | OR

Finally, on p_create, it's used one unmentioned grammar rule: create_source.

CreateSource

Responsible for table's source for creating a new table. The source can be from a SELECT or JOIN query.

Grammar Rule: create_source : table_join | q_select

Join

Joins two tables in one, crossing on a given column.

Grammar Rule: table_join : var JOIN USING '(' var ')'

Error

Common rule to lead with invalid grammar, raising an exception with the problem.