School of Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

COMP2212 Programming Language Concepts
Group Report
MQL - query language for CSV files

Popa *Andrei (ap4u19@soton.ac.uk)*

April 10, 2022

1 Introduction

For this coursework our group chose to implement a query language following the imperative paradigm, allowing the user to have more freedom in crafting specific queries. MQL is designed for programmers who want a more controlled, step-by-step manner of constructing queries. The main reason for choosing this approach over declarative query languages was to provide a much different alternative for querying CSV documents, the main benefit being that the user can design its own, more efficient, algorithm rather than using the already provided, built-in, functionalities of a declarative language.

2 Main Language Features

2.1 Syntax

MQL follows closely the syntax of most common imperative programming languages.

Firstly, the user needs to define and initialise the table data types that are going to be used for writing the query. Consequently, there is the possibility for defining String, Int, Float, Boolean, Row (list of strings), Table(list of lists of strings) variables that could be used for holding element values.

Furthermore, the language provides the option of having if-then-else conditions and looping, which gives a lot of flexibility for designing queries. In addition, we also have integrated the basic arithmetic operations("+,-,*,/") for int and float types as well as comparison for int("==="),string ("====") and boolean("=="). All statements including loops and if statements end in a semi-colon which announces the end of a logical statement.

Scoping

MQL has dynamic scoping as the variables must all be declared globally at the beginning of the program before the logical part to maintain a clean look for the program and also make it easier to understand. Variables can not be declared inside loops and each variable can be declared individually on the same or a separate line, but in both cases they still need to be followed by a semi-colon individually.

```
4   boolean flag1 = true; boolean flag2 = true;
```

Lexical Rules, Grammar and Syntax

Lexical Grammar

MQL's grammar provides a broad range of tokens and is extensive giving as much freedom as possible to the user. MQL also has different built-in helper methods. These methods include checking the type of a variable("isInt()"), for changing the type from int to string("toString()"), for inserting elements in a table or in a row("putElement()") or retrieveing an element("getElement()"), for merging rows("mergeRow()") and for sorting ascending or descending accordingly("ascSort()","descSort()").

see Picture 1

Lexical Syntax

Below you can find a picture of the grammar of MQL. The non-terminal ***Lines*** is the point of enter in the grammar, which can be derived to the general statements of the language (assignments, if expressions, loops, etc.). The non-terminal ***Line*** can be derived to the basic expressions of the language (such as addition, subtraction, division, multiplication, etc.). We chose the option to declare the order of precedence in the *Grammar.y* file instead of layering the BNF Grammar in multiple non-terminals.

see Picture 2

2.2 Type Checking

MQL is a Strong Dynamically Typed programming language because every node's type from the resulted parse tree will be evaluated during the runtime of the program only if that particular node has an effect on the output of the program. For example, the following program will not throw any error as the illegal assignments(line 5 and line 6) are never reached due to the fact that they do not have any effect on the output table C (notice that line 8 is commented).

see Picture 3

But, if we uncomment line 8, then the program will throw a TypeCheckException, because of line 5.

see Picture 4

We have made this design choice because we think that if the statement does not affect the semantics and, ultimately, the output of the overall program, then the query should run perfectly fine.

Additional Features for Type Checking/Switching

MQL gives the programmer the possibility to do type checking/switching during the runtime of any element from the input table, as every element from the CSV file will be treated as a string initially. There are built-in helper functions for this:

- for type checking: .isInt(), .isString(), .isFloat()
- for type switching: .getInt(), .toString(), .getFloat()

The functions for type checking require a string type and they return a boolean. Example:

```
string foo = "5";
string bar = "notAnInt";
boolean result1 = foo.isInt(); -- this will return true
boolean result2 = bar.isInt(); -- this will return false
```

The functions for type switching require a string (in the case of *getInt()* and *getFloat()* and will return an int and a float, respectively; whereas *toString()* will require as input either an int or a float and will return a string. Example:

```
int foo1 = 10; float foo2 = 10.50; string foo3 = "10"; string foo4 = "10.50";
string bar1;   string bar2;        int bar3;           float bar4;
bar1 = foo1.toString();
bar2 = foo2.toString();
bar3 = foo3.getInt();
bar4 = foo4.getFloat();
```

We advise to use these 2 features concurrently when reading data from the CSV file using an if block statement. Example:

```
... declared table A, row aRow, string aEl, int aInt
aRow = A.getRow(0);
aEl = aRow.getElement(0);
if(aEl.isInt())
    aInt = aEl.getInt();
endif;
... rest of logic
```

2.3 Errors/Exceptions of MQL

Below is the Exception Hierarchy of MQL.

see Picture 5

2.4 Informative Error Messages

As seen in the Exception Hierarchy diagram, MQL has numerous errors/exceptions, including:

- Lexical Error: in this case the program will point the respective row and column of the code with the error.
- Parse Error: in this case the program will point the respective row and column of the code with the error.
- Exceptions :
  - VariableExistenceException: This exception will be thrown when either
    - a variable that is not defined is used in an operation.
      Error message: "VariableExistenceException : << *variable_type* >> with name: << *variable_name* >> does not exist"
    - another declaration of a variable with the same name is being made.
      Error message: "VariableExistenceException : << *variable_type* >> variable: << *variable_name* >> already exists"
  - OutputException: This exception will be thrown when the end line of the program is not a printTable() statement.
    Error message: "OutputException : the last line of the program must be printing the output Table."
  - TypeCheckException: This exception will be thrown when the expected type for an operation does not match with the actual type given.
    Error message: "TypeCheckException : Used << *actual_type* >> variable where << *expected_type* >> was expected"
  - TableOutOfBoundsException: This exception will be thrown when the requested index row is either
    - negative
      Error message: "TableOutOfBoundsException : Used negative number when trying to get a row from table."

    - larger than the table size
      Error message: "TableOutOfBoundsException : Used number greater than the table size when trying to get a row from table. Index requested: << *requested_index* >> Table : << *table_rows* >>Table length: << *table_length* >>"
  - RowOutOfBoundsException: This exception will be thrown when the requested index is either
    - negative
      Error message: "RowOutOfBoundsException : Used negative number when trying to get an element from row."
    - larger than the row size
      Error message: "RowOutOfBoundsException : Used number greater than the row size when trying to get an element from row. Index requested : << *requested_index* >> Row : << *row_elements* >> Row length : << *row_length* >>"
  - ArithmethicException: This exception will be thrown when an operation that is not possible for an expected data type is being executed.
    The error messages for this exception are based on the illegal operation that was requested to be executed, including:
    - "ArithmethicException : Tried getting a row variable from a variable that is not a table."
    - "ArithmethicException : Tried getting a string variable from a variable that is not a row"
    - "ArithmethicException : Tried to append to something that is not a Table."
    - "ArithmethicException : Tried to merge something that is not a Row."
    - "ArithmethicException : Tried to put an element into a variable that is not a Row."

3 Execution Model For the Interpreter

3.1 Declare and Initialise

see Picture 6

Every text file containing code in our language will represent a single query. Every such query **must** begin by firstly declaring and possibly initialising the variables that are going to be used:

- variable "A" is being assigned the datatype *table* , which will represent a Haskell list of lists, each containing String elements;
- variable "C" will be of type *table* an it will hold the resulting output;
- *boolean* flags are used as conditions for performing loops;
- *string* variables are being declared and afterwards will be given specific values from the csv table;
- *row* datatype holds a list of Strings
- *int* datatypes are used for performing basic arithmetic operations;

3.2 If-Then-Else

see Picture 7

The language supports functionality for the *if-then-else* conditional statements. The condition can contain multiple conjunction/disjunction operations. In this particular example the *putElement()* function is being used to insert a specific value on the current column of row *cRow* .

3.3 Loop

see Picture 8

Our loop acts a standard Java loop, taking *boolean* values as condition and performing language specific operations as long as the condition maintains the semantic value ***true***. The way our language works towards building a desired output is as follows:

- a ***row*** variable will be used to insert each element into the output table in a sequential order (i.e. first column, then second column, etc) using the *putElement()* method. Please note that, before starting to add elements to a row, an user will need to call the *resetRow()* function which will erase any previous content that variable might have hold;
- after every element from a specific row is put, the *putRow()* method will be invoked on a specific table into which we want that row to be inserted (i.e. the resulting table "C"). Doing these operations for each row will end up populating the output table with the desired output;

3.4 Print

see Picture 9

In order to see the resulting table, an user will need to invoke the *printTable* function on the "C" variable, which will write to Standard Output a CSV table containg elements from the query. There is also the functionality, which allows one to sort in lexicographical or in reverse lexicographical order.

4 APPENDIX

4.1 Picture 1

```
{
module Tokens where
}

%wrapper "posn"
$digit = 0-9
-- digits
$alpha = [a-zA-Z]
-- alphabetic characters

tokens :-
$white+      ;
  "--".*       ;
  int             { tok (\p s -> TokenTypeInt p) }
  string          { tok (\p s -> TokenTypeString p) }
  boolean         { tok (\p s -> TokenTypeBool p)}
  float           { tok (\p s -> TokenTypeFloat p)}
  list            { tok (\p s -> TokenTypeList p)}
  lists           { tok (\p s -> TokenTypeLists p)}
  loop            { tok (\p s -> TokenLoop p)}
  do\:            { tok (\p s -> TokenDo p)}
  if              { tok (\p s -> TokenIf p)}
  endLoop         { tok (\p s -> TokenEndLoop p)}
  endIf           { tok (\p s -> TokenEndIf p)}
  else            { tok (\p s -> TokenElse p)}
  break           { tok (\p s -> TokenBreak p)}
  $digit+\.$digit+ { tok (\p s -> TokenFloatValue p (read s) ) }
  table           { tok (\p s -> TokenTypeTable p)}
  row             { tok (\p s -> TokenTypeRow p)}
  \.tableLength   { tok (\p s -> TokenTableLength p)}
  \.putRow        { tok (\p s -> TokenPutRow p)}
  \.getRow        { tok (\p s -> TokenGetRow p)}
  \.putElement    { tok (\p s -> TokenPutElement p)}
  \.getElement    { tok (\p s -> TokenGetElement p)}
  \.resetRow      { tok (\p s -> TokenResetRow p)}
  \.mergeRow      { tok (\p s -> TokenMergeRow p)}
  \.isInt()       { tok (\p s -> TokenIsInt p)}
  \.isString()    { tok (\p s -> TokenIsString p)}
  \.isFloat()     { tok (\p s -> TokenIsFloat p) }
  \.getFloat()    { tok (\p s -> TokenGetFloat p) }
  \.getInt()      { tok (\p s -> TokenGetInt p) }
  \.toString()    { tok (\p s -> TokenToString p) }
  getTable        { tok (\p s -> TokenGetTable p)}
  printTable      { tok (\p s -> TokenPrintTable p)}
  \"              { tok (\p s -> TokenQuotation p) }
  \.asc           { tok (\p s -> TokenAscSort p) }
  \.desc          { tok ( \p s -> TokenDescSort p ) }

  \>\=            { tok (\p s -> TokenGreaterEquals p)}
  \<\=            { tok (\p s -> TokenLessEquals p)}
  \>              { tok (\p s -> TokenGreater p)}
  \<              { tok (\p s -> TokenLess p)}
  \=\=            { tok (\p s -> TokenEquals p)}
  \=\=\=          { tok (\p s -> TokenIntEquals p)}
  \=\=\=\=        { tok (\p s -> TokenStringEquals p) }
  \!\=            { tok (\p s -> TokenNotEquals p)}
  \!              { tok (\p s -> TokenNot p)}
  or              { tok (\p s -> TokenOR p)}
  and             { tok (\p s -> TokenAND p)}
```

Go back to "Lexical Grammar"

4.2 Picture 2

```
data Lines = If Line Lines | IfElse Line Lines Lines | Loop Line Lines| IntDeclareAssign String Line | IntDeclare String
           | StringDeclareAssign String Line| StringLiteralDeclareAssign String Line| StringDeclare String
           | BoolDeclareAssign String Line | BoolDeclare String | ListDeclareAssign String Line | ListDeclare String
           | TableDeclareAssign String Line | TableDeclare String | RowDeclareAssign String Line | RowDeclare String
           | FloatDeclareAssign String Line | FloatDeclare String | ListsDeclareAssign String Line | ListsDeclare String
           | VarAssign String Line| PrintTable String| PutRow String Line | PutElement String Line | MergeRow String Line
           | ResetRow String | AscSort String | DescSort String | ToString String| LinesStream Lines Lines
           deriving (Show,Eq)

data Line = Int Int | Var String |  TTrue | TFalse | Float Float
          | Negate Line | Plus Line Line | Minus Line Line | Times Line Line | Divide Line Line | Mod Line Line | Div Line Line
          | IntEquals Line Line | StringEquals Line Line | GreaterEquals Line Line| LessEquals Line Line | Greater Line Line | Less Line Line
          | Equals Line Line | NotEquals Line Line | Or Line Line | And Line Line | Not Line | IsInt String | IsString String| IsFloat String
          | TableLength String | GetRow String Line| GetInt String| GetFloat String| GetElement String Line| GetTable String| BreakLoop
          deriving (Show,Eq)
}
```

Go back to "Lexical Syntax"

4.3 Picture 3

```
test.cql
  row foo1;
  int foo2 = 0;

  row bar1 = foo1.getRow(0);
  row bar2 = foo2.getRow(0);

  -- C.putRow(bar1);

  C.asc();
  printTable C;

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
Andrei@DESKTOP-B4TDOBC MINGW64 /d/Myosotis/submission1
$ ./csvql.exe test.cql


Andrei@DESKTOP-B4TDOBC MINGW64 /d/Myosotis/submission1
$
```

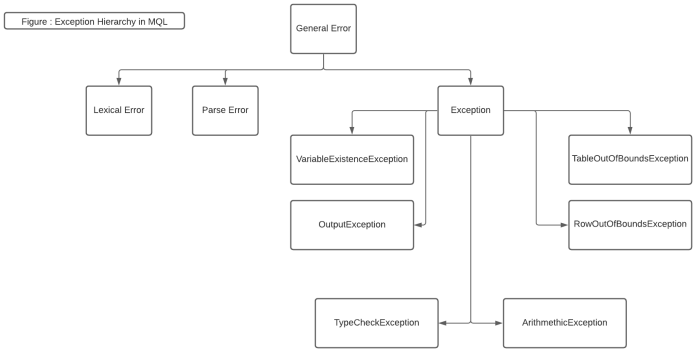
Go back to "Type Checking"

4.4 Picture 4

```
test.cql
  row foo1;
  int foo2 = 0;

  row bar1 = foo1.getRow(0);
  row bar2 = foo2.getRow(0);

  C.putRow(bar1);

  C.asc();
  printTable C;

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Andrei@DESKTOP-B4TDOBC MINGW64 /d/Myosotis/submission1
$ ./csvql.exe test.cql
GENERAL ERROR : TypeCheckException : Used row variable where table was expected
CallStack (from HasCallStack):
  error, called at .\Evaluator.hs:364:39 in main:Evaluator
```

Go back to "Type Checking"

4.5 Picture 5



Go back to "Errors/Exceptions of MQL"

Picture 6

```
2   table A = getTable A;
3   table C;
4   boolean flag1 = true;
5   string a1; string empty;
6   string p = "0";
7   row aRow; row cRow;
8   int i = 0; int aLimitTableLength = A.tableLength - 1;
```

Go back to "Declare and Initialise"

4.6 Picture 7

```
if( p2 ==== empty )
    cRow.putElement(q2);
else
    cRow.putElement(p2);
endIf;
```

Go back to "If-Then-Else"

4.7 Picture 8

```
loop(flag2) do:
    bRow = B.getRow(j);
    cRow.resetRow();
    cRow.mergeRow(aRow);
    cRow.mergeRow(bRow);
    C.putRow(cRow);
    if(bLimitTableLength === j )
        flag2 = false;
    endIf;
    j++;
endLoop;
```

Go back to "Loop"

4.8 Picture 9

```
37    C.asc();
38    printTable C;
```

Go back to "Print"