Enhanced **Q**uery **L**anguage
Programming Language Manual

Oliver J. Martin (ojm1g16)
William J. Bradley (wjb1n15)

April 2018

1 Introduction

EQL (pronounced ‘equal’) is a declarative programming language for executing queries on CSV files. Its semantics are derived from those of conjunctive queries, but don’t worry if you are unfamiliar with these. If you have previously used a language such as SQL, many of the concepts are similar. If not, all will be explained in this manual.

2 Queries

Comma separated values (CSV) files are a simple, yet very common, way to store tabulated data. Here is an example file, People.csv, which contains information about four fictional people:

```
Oliver,20,Sevenoaks
Dan,19,Southampton
James,22,Isle of Wight
Billy,21,Maidstone
```

The first field is the person’s name, the second their age, and the third the town in which they live. Using EQL we can perform queries on this data to extract information. An EQL program is simply a list of queries[1]. Here is a short query to list the names of everyone in the file.

```
name {
    People(name, age, town);
}
```

The first part of an EQL query is a comma separated list of the variables you want to return. In this case we just want to get the ‘name’ field, so we use a variable with the same name. Next we must specify which file we are reading, and what variable each field should be bound to. We are reading the ‘People.csv’ file[2], so write the file-name without the ‘.csv’ extension. CSV file names must begin with an upper case letter, and variable names with a lower case letter. In the parentheses we bind variables to each field of the data. The identifiers of these variables should be meaningful and hence will normally relate to the contents of the field. The result of this line is to represent the CSV file as a relation, which is essentially another word for a table.

This query results in the ‘name’ variable binding to all the values of that column of the CSV file, and since there are no other constraints on it (more on this later), the output of this query will simply be all the values from the first column of ‘People.csv’.

It should be noted that the results of EQL queries will be sorted in lexicographical order.

[1]Many programs will only have one query, but it is possible to have multiple queries in the same program.
[2]The CSV file to read must be in the same directory as the EQL interpreter.

3 Equality constraints

Lets assume we have the following CSV file of peoples' first and last names:

```
Billy ,Parker
Thomas,Thomas
Daniel ,Martin
Bradley ,Bradley
```

We want to extract only those people whose first name is the same as their last name. One such program to do this would be:

```
first , second {
    Names(first , second);
    first = second;
}
```

You'll notice that there are two variables on the first line this time. This means the output of the program will have two columns, the first column being the values of the 'first' variable, and the second column being the values of the 'second' variable.

The second line is essentially the same as in the first example, but the third line demonstrates a new constraint on the output. Only rows where the variable bound to the first row and the variable bound to the second row are equal will be included in the output of the program.

4 Joins

Data from several different relations (or possibly the same relation multiple times) can be combined in a single query using joins. Consider the scenario where there are two relations kept by a vet: People.csv (which stores names and phone numbers) and Pets.csv (which stores pet names and owner names).

```
Aaron,12345678909
Beth,11223344556
Charlie,09988776655
```

```
Fluffy ,Aaron
Spot ,Beth
Polly ,Aaron
```

The following program outputs the name of each pet with the phone number of its owner.

```
pet, number {
    People(owner, number);
    Pets(pet, owner);
}
```

Here, the same variable is used for the owner's name field in each relation, so the interpreter will match rows using that column. The same functionality could be achieved using an equality constraint:

```
pet, number {
    People(person, number);
    Pets(pet, owner);
    person = owner;
}
```

The output for both programs is as follows:

```
Fluffy,12345678909
Polly,12345678909
Spot,11223344556
```

5 Existence checks

To check existence is similar to what we've done in previous examples, only without printing the results. We will look at an example of a simple team game. Each player will take two coloured cards from a bag. If the colours match then their entire team will receive a prize. If two peoples colours match their team will get two prizes, and so on. The files below store the results from an example game in the following format: name,colour1,colour2

```
Max,Green,Green
Oscar,Blue,Red
May,Red,Red
```

The following program will output all players who have won a prize, and they will be listed multiple times if they have won more than one prize.

```
name {
    Team(name,_,_); // This is a comment!
                    // The underscore means 'don't care about this column.'
    Team(_,colour1,colour2);
    colour1=colour2;
}
```

This should be read as 'print name where name is the first column from Team.csv, where there exists some row in that file with both colours equal to each other.' Because there are two winners in Team.csv, the output will be:

```
Max
Max
May
May
Oscar
Oscar
```

6 Enhanced relation declarations

Imagine there is a relation recording, for each person, the length of each of their fingers going from left to right.

```
Alice,3,4,4.5,4,2.5,2.4,4.1,4.4,3.9,3
Bob,3.2,4.3,4.9,4,2.8,2.7,3.9,4.6,4.2,3.1
Charlotte,2,3,3.5,3,1.5,1.4,3.1,3.4,2.9,2
```

It is critically important that we extract the length of everyone's right little finger, so we might come up with something like this:

```
name,length {
    Fingers(name,_,_,_,_,_,_,_,_,_,length);
}
```

However, that's verbose, and it's easy to miscount the columns. Fortunately, EQL has a feature that makes such queries more concise and their meaning more apparent:

```
name,length {
    Fingers[0,10](name,length);
}
```

By stating that we're only interested in columns 0 and 10, we cut out much of the bulk while making the program tidier and clearer.

7 Appendix

7.1 Programs

```
x1,x3,x2,x4 {
  A(x1,x2);
  B(x3,x4);
}
```

Listing 7: pr1.cql

```
x1,x2,x3 {
  A(x1,x2);
  B(x2,x3);
}
```

Listing 8: pr2.cql

```
x1,x2 {
  P(x1);
  Q(x2);
  x1=x2;
}
```

Listing 9: pr3.cql

```
x1 {
  R(x1,_);
}
```

Listing 10: pr4.cql

```
x1,x2 {
  A(x1,z);
  B(z,x2);
}
```

Listing 11: pr5.cql

```
x1 {
    R(x1);
    S(z);
}
```

Listing 12: pr6.cql

```
x1, x2 {
    R(x1, z1);
    R(z1, z2);
    R(z2, x2);
}
```

Listing 13: pr7.cql

```
x1, x2, x3, x4, x5 {
    R(x1, x2);
    R(x2, x3);
    R(x3, x4);
    R(x4, x5);
    x1 = x5;
}
```

Listing 14: pr8.cql

```
x1, x2 {
    A(x1, z1);
    B(z1, z2);
    C(z2, x2);
}
```

Listing 15: pr9.cql

```
x1, x2, x3 {
    S(x1,x2,x3);
    S(a,b,_);
    a = b;
    S(_,c,d);
    c = d;
}
```

Listing 16: pr10.cql

7.2 Error Reporting

7.2.1 Parse Error

An error which occurs as a result of parsing the tokens is reported as a parse error. This displays a helpful message informing the user of the location of the error, and information about the characters involved.

Example program:

```
x1,x2 {{
  A(x1,z);
  B(z,x2);
}
```

Listing 17: error1.cql

Error output:

```
myinterpreter.exe: Parse Error - Line 1, Column 8 - Near '{'
CallStack (from HasCallStack):
  error, called at Main.hs:26:26 in main:Main
```

Listing 18: error1 output

7.2.2 Lexical Error

An error which occurs as a result of lexical analysis is reported as a lexical error. This displays a helpful message informing the user of the location of the error.

Example program:

```
x1@ {
  A(x1,z);
  B(z,x2);
}
```

Listing 19: error2.cql

Error output:

```
myinterpreter.exe: lexical error at line 1, column 3
CallStack (from HasCallStack):
  error, called at templates\wrappers.hs:451:61 in main:Lexer
```

Listing 20: error2 output

7.2.3 Unconstrained Variable

An error which occurs as a result of referring to a variable which has not been bound to a column of a relation is flagged as an 'unconstrained variable' error.

Example program:

```
x3 {
  A(x1,z);
  B(z,x2);
}
```

Listing 21: error3.cql

Error output:

```
myinterpreter.exe: Variable unconstrained: x3
CallStack (from HasCallStack):
  error, called at .\Interpreter.hs:89:27 in main:Interpreter
```

Listing 22: error3 output

7.2.4 Missing file

If a relation is reffered to, but the corresponding CSV file is missing, then a missing file error is displayed, identifying the file which could not be found.

```
x1,x2 {
  Z(x1,z);
  B(z,x2);
}
```

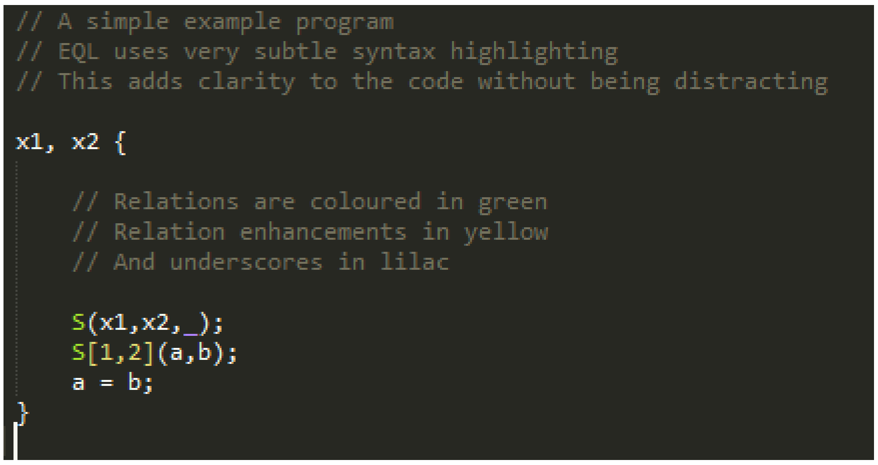
Listing 23: error4.cql

```
myinterpreter.exe: Relation Z does not have matching file: Z.csv
CallStack (from HasCallStack):
  error, called at .\Interpreter.hs:61:30 in main:Interpreter
```

Listing 24: error4 output

7.3 Syntax Highlighting for Sublime Text 3

As part of the development of our language we wrote a custom syntax definition for Sublime Text. This allows EQL programs to be displayed more clearly.



```
// A simple example program
// EQL uses very subtle syntax highlighting
// This adds clarity to the code without being distracting

x1, x2 {

    // Relations are coloured in green
    // Relation enhancements in yellow
    // And underscores in lilac

    S(x1,x2,_);
    S[1,2](a,b);
    a = b;
}
```

Here is the YAML file which can be imported into Sublime Text:

```
%YAML 1.2
---
name: EQL
# See http://www.sublimetext.com/docs/3/syntax.html
file_extensions:
  - eql
  - cql
scope: source.eql

contexts:
  # The prototype context is prepended to all contexts but those setting
  # meta_include_prototype: false.
  prototype:
    - include: comments

  main:
    # The main context is the initial starting point of our syntax.
    # Include other contexts from here (or specify them directly).
    - include: underscore
    - include: enhancement
    - include: relation
    - include: variable

  underscore:
    # Note that blackslashes don't need to be escaped within single quoted
    # strings in YAML. When using single quoted strings, only single quotes
    # need to be escaped: this is done by using two single quotes next to each
    # other.
    - match: '\b(\_)\b'
      scope: constant.numeric.eql

  relation:
    - match: '[A-Z][A-Za-z_''0-9]*'
      scope: entity.name.class.eql

  variable:
    - match: '[a-z][A-Za-z_''0-9]*'
      scope: variable.other.eql

  enhancement:
    # Enhancements are within square brackets
    - match: '\['
      scope: punctuation.definition.string.begin.eql
      push: inside_enhancement

  inside_enhancement:
    - meta_include_prototype: false
    - meta_scope: string.quoted.double.eql
    - match: '\]'
      scope: punctuation.definition.string.end.eql
      pop: true

  comments:
    # Comments begin with a '//' and finish at the end of the line.
    - match: '//'
      scope: punctuation.definition.comment.eql
      push:
        # This is an anonymous context push for brevity.
        - meta_scope: comment.line.double-slash.eql
        - match: $\n?
          pop: true
```

Listing 25: eql.sublime-syntax