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

COMP2212 - PipeQL
Language Report

Version 1

Dzhani Daud *dsd1u19@soton.ac.uk*
Velimir Anastasov *vna1u19@soton.ac.uk*
Velina Valcheva *vv1u19@soton.ac.uk*

4^{th} May 2021

1 About PipeQL

PipeQL is a domain-specific declarative query language for working with CSV files. It follows the Pipes and Filters architecture and syntax, and is heavily inspired by Unix's philosophy "Make each program do one thing well". As for the language's functionality, it is based on the basics of Relational Algebra and Set Theory - the language's most simple operations are Union, Difference, Cartesian Product, Projection and Selection.

1.1 Getting Started - Variables and Imports

To start working with a CSV file, first we would need to import it using the `import <fileName>` pipe (we will explain how pipes work shortly). We will assign the value that the import returns using the keyword `csv` :

```
csv A = import "A.csv"; // each statement must end with a ';'
```

Now that we have imported "A.csv", and stored it in a variable called A, we can start using it by applying it to different queries.

1.2 Queries and Pipes

In PipeQL almost everything acts as a pipe - a Pipe is an operation which takes a CSV as input and outputs another CSV. A Query is simply a bunch of Pipes connected to each other. To connect different pipes, we use the '|' symbol between them:

<Query> ::= <Pipe> '|' <Pipe> '|' <Pipe> '|' .. <Pipe>;

We use ';' to mark the end of the query. As we already said, `import <fileName>` is a pipe. It takes a CSV (in this case an empty one), ignores it and returns the contents of the file. Another pipe is `print` - it takes a CSV, prints it to the console and then returns the same CSV. We can combine the print and import pipes into a single query:

```
import "A.csv" | print; // imports "A.csv", then prints its contents
```

In fact, we can even assign the value of this query to a variable:

```
csv A = import "A.csv" | print; // reads "A.csv", prints it and then stores it in A
```

1.3 The Different Pipes

- Import pipe: `import <String>` - Returns a CSV. Expects a string / path to a ".csv" file. Throws an error if it fails to import the desired file.
- Print pipe: `print` - Takes a CSV, prints it, and outputs it to the next pipe.
- Variable Pipe: `Variable Name` - Checks the variable name in the environment. If the variable is pointing to a CSV value, we return it. Otherwise, if the variable is pointing to a procedure (method), then we call the procedure itself by also passsing the input CSV we just received.
- Ascend pipe: `asc` - Orders the input CSV into ascending order.
- Descend pipe: `desc` - Orders the input CSV into descending order.
- Select pipe: `select <Conds>` - Expects a predicate. Takes a CSV and filters it based on the predicate.
- Reform pipe: `reform <Cols>` - Expects a list of colums. Takes a CSV and re-shapes it based on the list of columns.

- Update pipe: `update <Col> <Col>` - Expects two columns. Takes a CSV and updates the first column based on the second one.
- Write pipe: `write <String>` - Expects a string / path to a ".csv" file. Takes a CSV and writes it in the desired location. Passes the CSV to the next pipe.
- Unique pipe: `unique` - Takes a CSV and filters out all duplicate entries.
- Note pipe: `note <String>` - Expects a string. Prints a message to the console.
- Error pipe: `error <String>` - Expects a string. Terminates the program with an error message.
- If control: `if <Conds> '->' <Query>;` - Takes a predicate and applies a query to all entries that satisfy the predicate.

The most interesting pipe is the if control. It takes an input CSV and a predicate. We map each CSV row, that satisfies the predicate to a query. Let's take a look at the following complex piece of code:

```
// Imports "A.csv". Then for each entry:
//     - if the entry's id > 4, we update its first value with "foo" and print
//     - otherwise: we do nothing
// After fully evaluating the if control, we sort the modified CSV in ascending order,
// and then write it to a file "output.csv"
// Finally, we store the the final CSV into a variable called B
csv B = import "A.csv" | if (id > 4) -> update $0 "foo" | print;
                       | asc
                       | write "output.csv";
```

To improve readability, we have placed some of the pipes on separate lines. In a condition block (a predicate), we can compute Math and Boolean expressions as well as use the keywords `arity` and `id` to get the current entry's length and index. Additionally, an entry's value can be accessed by `$valueIndex` , which will throw an error if `arity <= valueIndex`.
Another powerful pipe is the reform, which mimics the Projection operator from Relational Algebra. It takes a list of columns (a column is represented by either a `$valueIndex` or a filler string value):

```
csv A = import "A.csv"; // "A.csv" contains {a,b,c,d,e} with arity = 5
A | reform [$0, "foo", $1, $0] | print; // outputs: a,foo,b,a
A | reform [$0..$3, "bar"] | print; // syntax sugar for reform [$0, $1, $2, $3, "bar"]
A | reform [$0..$arity - 3, $1] | print; // outputs: a,b,c,b
```

1.4 Other Query Operations

There are 5 different operations which involve queries:

- CSV variable declaration: `csv A = <Query>`
- Procedure (function/method) declaration. Stores a reference to a query: `query customPipe = <Query>`
- Cartesian Product: `<Query> x <Query>` - Returns a CSV, the cross product of the 2 queries
- Union: `<Query> ++ <Query>` - Returns a CSV, the union of the 2 queries
- Difference: `<Query> -- <Query>` - Returns a CSV, the difference of the 2 queries

We can also call a query without storing it as a variable or procedure. See appendix A for the full grammar.

1.5 PipeQL Interpreter, Scoping and Lexical Rules

The interpreter evaluates the program one Query at a time. Additionally, a Query is evaluated one Pipe at a time. Like most languages, a variable can be called only after it has already been declared. Additionally, procedure declarations can only call pre-defined variables. The language also supports comments which are declared using "//". Pipes are left associative, meaning that:

`A | asc | print;` is interpreted as `(((A) | asc) | print);`

Note: the example query takes the variable A, sorts it in ascending order and prints it to standard out.

2 Other Features

2.1 Type Checking

PipeQL has a type system which gets called right after the program is parsed. The type system also checks if we are calling any undeclared variables / procedures. In reality, PipeQL's grammar is specific enough to catch most type errors - for example, the grammar would fail to parse `update 0 "foo"` because it would expect a column `e.g. $0` as its first argument, and not the number 0.

2.2 Error Handling

PipeQL supports the following errors:

- Parse Errors - when the program fails to parse, we get notified with a meaningful message including the row and column of where the error occured
- Undeclared Variable / Procedure - again, this is done by the type system right after parsing
- Index Out of Bounds Error - gets raised when the user tries accessing a CSV entry's index that is bigger than the entry's arity
- Invalid Math Expression - for example, division by 0
- Failed Import Error

Additionally, developers can create their own custom error cases using the `error "messsage"` pipe. For example:

```
// If "A.csv" contains an entry, whose arity is not an even number, we throw an error
import "A.csv" | if (arity % 2 != 0) -> error "Error: Arity is not even"; | ...
```

While designing the language, we decided that explicitly specifying the arity of the imports would unnecessarily restrain the the language. After all, CSV files come with varying arities. What's more, the language's operations would still work on any CSV. Of course, developers can artificially restrict operations by adding `if (arity != x)` statements.

2.3 Language Server, Syntax Highlighting and PipeQL Theme

We implemented a language server, added syntax highlighting and even created a custom PipeQL colour theme - all in the form of a VS Code Extension.

The language server is following the Language Server Protocol. It provides us with keyword auto-completion and on-Hover Events. The syntax highlighter identifies all tokens and colours them in appropriate colours, based on the VS Code's selected colour theme. We weren't pleased with how VS Code coloured our language and for this reason, we decided to create our own PipeQL Colour Theme! Additionally, extra features such as Comment-Toggling, Code Block Folding and Brackets Auto-closing are also included.

The source code for the extension is stored in a private GitHub repository (so as not to breach academic integrity). To gain access, please email me at dsd1u19@soton.ac.uk

Appendices

A The Language Grammar

⟨*Program*⟩ ::= [⟨*Expr*⟩]

⟨*Expr*⟩ ::= '`csv`' String '`;`'
| '`csv`' String '`=`' ⟨*Query*⟩ '`;`'
| '`query`' String '`=`' ⟨*Query*⟩ '`;`'
| ⟨*Query*⟩ '`;`'

⟨*Query*⟩ ::= ⟨*Pipe*⟩ | ⟨*Query*⟩ '`|`' ⟨*Query*⟩

⟨*Pipe*⟩ ::= '`import`' String
| '`asc`'
| '`desc`'
| '`reform`' ⟨*Cols*⟩
| '`print`'
| '`select`' ⟨*Conds*⟩
| '`update`' ⟨*Col*⟩ ⟨*Col*⟩
| VarName
| '`write`' String
| '`note`' String
| '`error`' String
| '`unique`' String
| '`if`' ⟨*Conds*⟩ '`->`' ⟨*Query*⟩ '`;`'
| ⟨*Query*⟩ '`x`' ⟨*Query*⟩
| ⟨*Query*⟩ '`++`' ⟨*Query*⟩
| ⟨*Query*⟩ '`--`' ⟨*Query*⟩

⟨*Cols*⟩ ::= [⟨*ColItem*⟩]

⟨*ColItem*⟩ ::= ⟨*Col*⟩ | '`[$`' ⟨*MathExpr*⟩ '`..$`' ⟨*MathExpr*⟩'`]`'

⟨*Col*⟩ ::= '`$`' ⟨*MathExpr*⟩ | String

⟨*Conds*⟩ ::= ⟨*Cond*⟩ | '`!`' ⟨*Conds*⟩ | ⟨*Cond*⟩ '`&&`' ⟨*Cond*⟩ | ⟨*Cond*⟩ '`||`' ⟨*Cond*⟩

⟨*Cond*⟩ ::= ⟨*Col*⟩ ⟨*Operation*⟩ ⟨*Col*⟩ | ⟨*MathExpr*⟩ ⟨*Operation*⟩ ⟨*MathExpr*⟩ | '`true`' | '`false`'

⟨*Operation*⟩ ::= '`==`' | '`!=`' | '`<`' | '`>`' | '`<=`' | '`>=`'

⟨*MathExpr*⟩ ::= '`arity`' | '`id`' | Int | ⟨*MathExpr*⟩ ⟨*MathOperation*⟩ ⟨*MathExpr*⟩

⟨*MathOperation*⟩ ::= '`+`' | '`-`' | '`*`' | '`/`' | '`%`'

B VS Code Extension

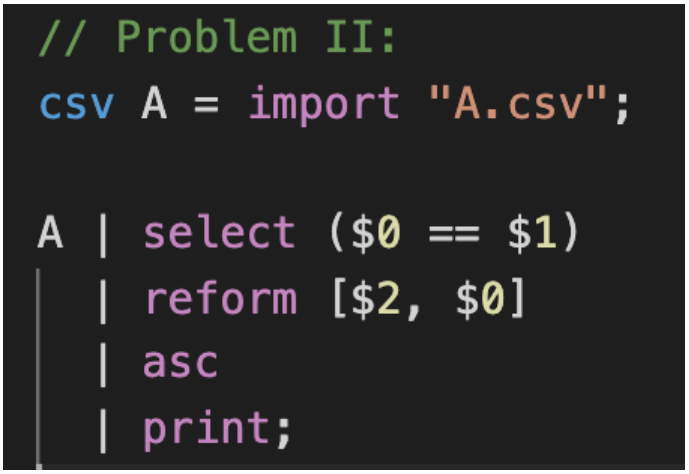

Figure 1: How the language looks using VS Code's Dark+ theme

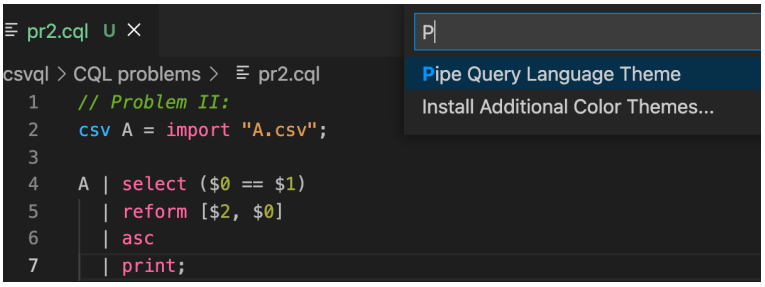

Figure 2: How the language looks with the PipeQL Theme

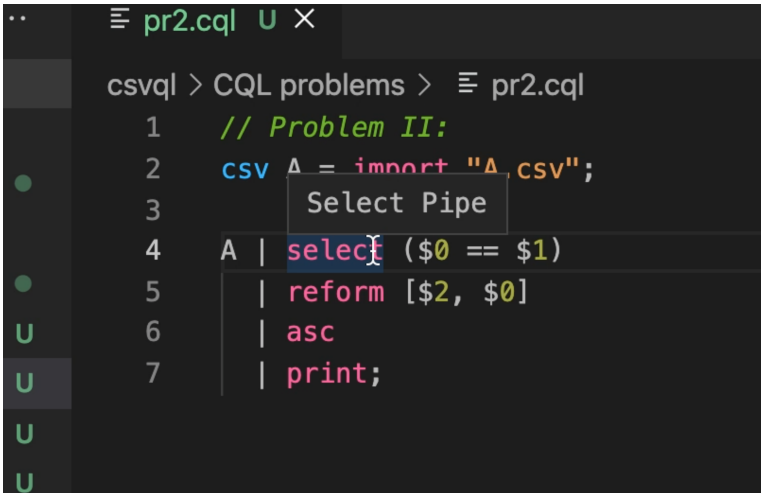

Figure 3: An example of an on-Hover Event

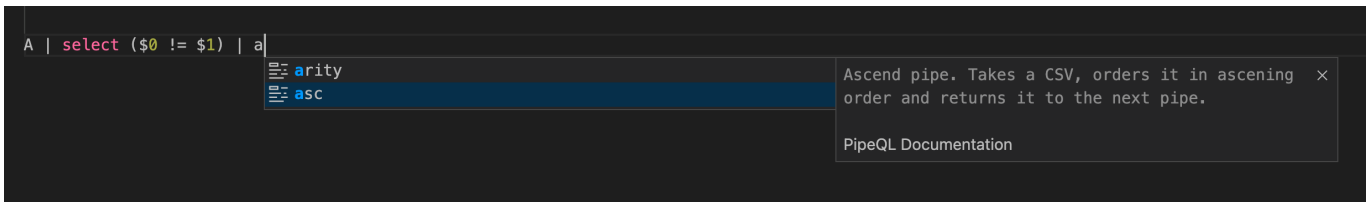

Figure 4: Keyword auto-completion