CSVQL: CSV Query Language Documentation

COMP2212: Programming Language Concepts

Charles Powell (cp6g18)

Electronics and Computer Science
University of Southampton

Contents

Introduction

The following report provides documentation on the programming language developed for the COMP2212: Programming Language Concepts coursework. The developed programming language supports the querying of CSV files, and is known as CSVQL.

1 Language Definition

1.1 Syntax

The syntax of CSVQL is heavily influenced by SQL, with many common SQL concepts being implemented in CSVQL according to the same syntax (e.g., `SELECT`, `INSERT`, `JOIN`). Syntactically, a CSVQL program will consist of a mixture of CSVQL keywords and variable user input. A "keyword" refers to the syntax of an operation that can be performed within a CSVQL program, such as a function like `DELETE` or `UNION`. All keywords are known prior to a program being written (i.e., a programmer cannot define their own keywords), and are always written in capitals. Variable user input refers to the parts of the program for which the syntax is provided by the user, such as file or variable names. Whilst there are some general constraints applied to such statements (e.g., a variable name must start with an alpha-character), the final appearance of these statements is otherwise decided by the programmer.

A full understanding of the CSVQL syntax can be acquired by referring to Language Semantics and Complete CSVQL Grammar.

1.2 Grammar

The following list outlines the language rules of CSVQL, providing an insight into it's grammar.A full understanding of the CSVQL grammar can be acquired by referring to Complete CSVQL Grammar. Example CSVQL programs are provided in Example CSVQL Programs, which demonstrates instances of both valid and invalid program structure according to the following rules.

- Each CSVQL program consists of zero or more `LET` statements followed by a single `RETURN` statement.
- The `LET` keyword is used to declare a variable by assigning a table object to a syntax label.
- The `RETURN` keyword is used to output the contents of a table object to the terminal. No execution will take place beyond the `RETURN` statement.
- A **table object** refers to any statement in a CSVQL program that can be evaluated to a table of values. All values within tables are considered as strings. There are three types of table object:
  - A **file** reference (in which the file contains the table of values), e.g., `READ "A.csv"`
  - A **variable** reference (i.e., using a label to represent the assigned table).
  - A **function** applied to a table object.
- Functions are operations that can be used to transform and query table objects. Functions take in table objects as input (and possibly other parameters depending on the type of function) and return table objects as output. Functions can therefore be composed onto each other to the n$^{\text{th}}$ degree. The list of functions included in CSVQL along with their corresponding semantics are provided in Language Semantics.
- Every line of code in CSVQL must be terminated with a semi-colon `;`.
- The execution of a CSVQL program can be configured by adding the `SETUP` keyword to the start of a program, followed by a list of configurations enclosed in braces. If included, the `SETUP` keyword must be the first statement within the program, and should not be succeeded with a semi-colon `;`.

Essentially, any CSVQL program consists of zero or more variable declarations followed by a single `RETURN` statement. Each variable declaration or `RETURN` statement references a table object, which can either be a table read from a file, a variable label, or a function applied to a table object.

1.3 Variables

CSVQL supports the declaration of variables using the `LET` statement. The `LET` statement can be thought of as a function that takes in a string and a value, and assigns the given value to the string. The result of this is a label that can be used for convenience within a program in place of the value that was assigned to it.

The following list provides information on the functionality of variables within CSVQL.

- The only form of data that can be stored within a variable is a table object.
- CSVQL programs contain only a single layer of scope, and so all variables are therefore globally scoped.

- If the variable name provided in a LET statement is an already declared variable within the program, it's value is overwritten based on the newly provided assignment.
- As variables themselves are table objects, one variable can be declared using another variable. In this case, the newly declared variable stores the current state of the provided variable, and not a reference to it (i.e., two independent variables storing the same value at the point of declaration).

2 Execution Model

When executing a program, the interpreter first checks for the occurrence of setup configurations, and gathers them in the case where they are used. Following this, program expressions are evaluated in order of occurrence, with recursion being used for continuation.

Each expression is evaluated in the context of the system state, which comprises of two elements; a mapping of string values to tables, and a system configuration. The mapping of string values to tables represents the variables declared within the system, with this list being updated throughout program execution following LET statements. The system configuration represents the configurations supplied by the programmer within the SETUP clause, and is empty in the case of omission. At any point in execution, the system configuration can be accessed in order to adjust the way in which the execution takes place.

A program has been completely evaluated when the RETURN statement has been evaluated, and following this, the table object referenced by the statement is returned to the main control flow. At this point, the table is passed to the printing system, which makes use of the system configuration to structure the program output. Program output is provided on `stdout`, while program errors are provided on `stderr`. Descriptions of the CSVQL error handling process are given within Error Messages of Additional Features.

Program expressions that succeed the RETURN statement are not evaluated, and an error is thrown in the case where they are present. Similarly, an error is thrown in the case where no RETURN statement is present.

Language Semantics contains a detailed description of the language semantics, providing a valuable insight into the execution model of the CSVQL language.

3 Additional Features

This section of the report describes the additional features that have been implemented into CSVQL for the purpose of programmer convenience. These features are showcased within Demonstration of Additional Features.

3.1 Function Composition

All functions within CSVQL take a table object in as input (as well as possible additional parameters), and return a table object as output, making the type of a function a table object. All functions are therefore able to receive other functions as input, which avoids the need for the programmer to break a program up into individual lines for each function being used. The result of this is that a program written with any number of lines can be expressed as an operationally equivalent program written on a single line. However, whilst the recursive nature of functions can be used for programmer convenience, over use will ultimately lead to less-readable programs, particularly when a large number of functions are used.

3.2 Predicate Composition

Predicates, used in WHERE statements in a variety of program operations can be composed to allow for complex comparisons. Predicates can be composed using the standard Boolean operations of AND and OR. This was done for programmer convenience, as to avoid the need for the programmer to make use of several statements when querying data, with each statement specifying individual predicates.

3.3 Comments

CSVQL supports single-line comments with the use of `--` as the comment initializer. Any text that succeeds `--` will be ignored by the interpreter and not considered as part of the program being executed.

3.4 Parenthesised Expressions

CSVQL supports the use of parenthesis in certain cases to allow for increased program readability. There are two instances in which the use of parenthesis are permitted; with composed functions, and with composed predicates. Note that parenthesis are not required in these cases, and that the operational semantics of a statement with parenthesis is equivalent to one without.

3.4.1 Table Objects

As CSVQL supports function composition to the n th degree, the programmer is permitted to enclose table objects (e.g., functions) within parenthesis to provide separation between the different areas of program logic.

3.4.2 Predicates

As CSVQL supports predicate composition to the n th degree, the programmer is permitted to enclose predicates in parenthesis. This allows for a complex predicate to be broken down into distinguishable sections to increase program readability.

3.5 Program Configurations

CSVQL supports program configurations, which can be used to adjust the manner in which programs are executed. A programmer can provide an optional program configuration using the `SETUP` keyword at the start of their program, followed by a list of configurations enclosed in braces. The execution of the program will then be adjusted according to the provided configuration. If the programmer does not provide a program configuration, the default execution method will be carried out. The supported configurations are described below in terms of the functionality they provide, and the default functionality carried out in the case when they are omitted from the program setup.

3.5.1 Pretty Printing

A programmer can configure their CSVQL program to output the results of a query in a ”pretty” format by including the `PRETTYPRINT` keyword within their program configuration. If this configuration is not provided, the query will be outputted as standard plain text, with no formatting applied.

3.5.2 Input Delimiter

A programmer can configure the delimiter used in the reading of tables from files by including the `INPUTDELIM` keyword followed by their chosen delimiter enclosed in quotation marks `""`. If this configuration is not provided, the CSVQL interpreter will instead use commas `,` as the delimiter (i.e., CSV format).

3.5.3 Output Delimiter

A programmer can configure the delimiter used in the outputting of executed programs by including the `OUTPUTDELIM` keyword followed by their chosen delimiter enclosed in quotation marks `""`. If this configuration is not provided, the CSVQL interpreter will instead use commas `,` as the delimiter (i.e., CSV format). Note that this configuration has no effect when the `PRETTYPRINT` configuration is used.

3.6 Error Messages

The CSVQL interpreter displays error messages to the user as a result of invalid input. The errors thrown by the interpreter are described below in terms of the point in execution in which they are thrown.

3.6.1 Compile Time Errors

The following list details the errors that can be thrown during the compilation of a CSVQL program. When a compilation error is thrown, an error message is displayed to the user that provides a line and column number to specify the part of the program that caused the error.

- A lexing error is thrown if text within the input program is not supported by the CSVQL syntax (i.e., cannot be tokenized).
- A parsing error is thrown if the structure of the input program is not supported by the CSVQL grammar (i.e., cannot be parsed).

3.6.2 Run-time Errors

The following list details the actions that will result in an error being thrown during the run-time of a CSVQL program. When a run-time error is thrown, an informative error message is displayed to the user that provides an explanation as to why the erroneous program statement is invalid.

- Loading a table from a file that does not exist.
- Loading a non-uniform table from a file.
- Referencing a variable that has not yet been instantiated.
- Referencing columns that are out of bounds for the table in question (e.g., `WHERE`).
- Inserting values that do not have the same arity as the table they are being inserted into.
- Performing binary operations on tables that do not have the same arity when this is a requirement of the operation being performed (e.g., `INTERSECTION`, `MERGE`).
- Providing parameters that are out of range for the table in question when performing formatting functions (e.g., providing a value greater than the length of the table when using `LAST`).

A Example CSVQL Programs

Provided in this appendix are examples of both valid and invalid programs written in the CSVQL programming language. For valid programs, the program operation is described (i.e., what the program does). For invalid programs, an explanation is provided as to why the program is not supported.

A.1 Valid Programs

A.1.1 Example 1

The following CSVQL program reads two tables from two files, produces a cross join of the two tables, and returns the result in ascending order. The output of this program will be every row in ”A.csv” paired with every row in ”B.csv”, sorted in ascending order.

```
LET table1 = READ "A.csv";
LET table2 = READ "B.csv";

LET joinedTable = JOIN FULL table1 table2;

RETURN ORDER IN ASC joinedTable;
```

Listing 1: Valid CSVQL Program

A.1.2 Example 2

The following CSVQL program reads a single table from a file, and outputs the third and first column for all rows where the first column has the same value as the second, with the output sorted in ascending order. The table is read from a TSV file, as the LOADFROMTSV setup configuration is used.

```
SETUP{
    LOADFROMTSV
}

LET table1 = READ "A.tsv";

LET selectedRows = SELECT [@2,@0] WHERE [@0 == @1] table1;

RETURN ORDER IN ASC selectedRows;
```

Listing 2: Valid CSVQL Program

A.1.3 Example 3

The following CSVQL program reads a single table from a file, and outputs all rows where the second column is non-null (i.e., not the empty string), with the output sorted in ascending order. The PRETTYPRINT configuration is used, which will result in a formatted output.

```
SETUP{
    PRETTYPRINT
}

LET table = READ "A.csv";

LET selectedRows = SELECT * WHERE [@1 != ""] table;

RETURN ORDER IN ASC selectedRows;
```

Listing 3: Valid CSVQL Program

A.2 Invalid Programs

A.2.1 Example 1

The following CSVQL program is invalid as it includes an expression after the RETURN statement.

```
LET table = READ "A.csv";

LET insertedTable = INSERT VALUES ["1","2","3"];

RETURN ORDER IN ASC insertedTable;

LET selectedTable = SELECT [@1,@2] FROM insertedTable;
```

Listing 4: Invalid CSVQL Program

A.2.2 Example 2

The following CSVQL program is invalid as the first line of code is not terminated with a semi-colon `;`.

```
LET table = READ "A.csv"

RETURN DELETE WHERE [@0 == "this"] FROM table;
```

Listing 5: Invalid CSVQL Program

A.2.3 Example 3

The following CSVQL program is invalid as the SETUP configuration does not come at the start of the program.

```
LET table = READ "A.csv";

SETUP{
    PRETTYPRINT
    LOADFROMTSV
}

RETURN ORDER IN DESC table;
```

Listing 6: Invalid CSVQL Program

B Language Semantics

Provided in this appendix is a detailed explanation of the semantics of the CSVQL programming language. The CSVQL operations have been divided into sections based on the functionality they provide, and descriptions are given to illustrate their semantics.

B.1 Program Structure

CSVQL programs are structured as a series of zero or more LET statements followed by a single RETURN statement. A programmer can also provide an optional configuration at the start of the program by using the SETUP keyword.

B.1.1 LET Statements

The LET keyword is used to declare variables in CSVQL. Variables are only able to store table objects, and once declared, can be used throughout the program in place of the table that was assigned to them. The syntax of the LET statement is as follows.

```
LET <VariableName> = <TableObject>;
```

B.1.2 RETURN Statements

The RETURN keyword is used to terminate a CSVQL program, and output the result of a query. A RETURN statement takes a table object as a parameter and outputs the contents of the table on stdout. The RETURN statement must be the last statement within the program, and only a single table object can be returned. The syntax of the return statement is as follows.

```
RETURN <TableObject>;
```

B.1.3 Program Configurations

Program execution can be configured by supplying configurations within a program. The SETUP keyword can be used at the start of a program, followed by a list of configurations enclosed in braces, in order to adjust the way in which the CSVQL interpreter handles the input. All configurations are optional, with the default execution carried out in the case of their omission. The SETUP statement can only appear at the start of a CSVQL program, and has the following syntax.

```
SETUP {
    <Configuration1> ,
    ... ,
    <ConfigurationN>
}
```

The list of supported configurations is supplied within Program Configurations of Additional Features.

B.2 Table Objects

In CSVQL, a table object is a construct that represents a table of values. A table object can be seen as a structure built on top of an underlying table of values. There are three types of table object; file reference, variable reference and function table. The concept of a table object exists to generalise the various components that can represent a physical table of values.

The CSVQL interpreter considers all table objects to fundamentally be the same. This is because all table objects represent a table of values, but do so by providing some overlaying structure. The result of this is that any operation that expects a table of values as input can instead receive a generalised table object, increasing the language's flexibility as a result.

In CSVQL, every physical table of values is considered to be a table of strings, with no other data-type being supported.

B.2.1 File Reference

A file reference is a form of table object in which the table of values is loaded from a CSV (or possibly TSV) file. Such a table object is defined by the READ statement, and has the following syntax.

```
READ "<pathToFile/filename>";
```

Note that the only way new tables can be loaded into the system is through a file reference.

B.2.2 Variable Reference

A variable reference is a form of table object in which a label is used to represent a physical table of values. After a variable has been declared and assigned some value within a LET statement, the name of this variable can be used anywhere in the remainder of the program to represent the underlying table object that was assigned to it. The functionality of CSVQL variables is detailed within Variables.

B.2.3 Function Tables

A function table is a form of table object that is produced when a function has been applied to another table object (or possibly multiple). All functions within CSVQL take table objects in as input (and possibly additional parameters), and produce table objects as output (i.e., the result of a function being applied to a table object is another table object). A list of functions supported by CSVQL and their corresponding semantics are provided in the following section.

B.3 Functions

A function in CSVQL is an operation that can be applied to a table object in order to query and transform it. Functions take in table objects as input (and possibly other parameters depending on the type of function) and return table objects as output. Functions can therefore be composed onto each other to the n$^{\text{th}}$ degree. CSVQL draws inspiration from popular query languages such as SQL within it's own function library, following similar syntax, grammar and semantics.

B.3.1 WHERE

The WHERE statement cannot be used within CSVQL as an independent function, but considering it as such can help to understand its operation. A WHERE statement is often added to a function call in order to configure the operation of that function. The WHERE statement can be thought of as a function that takes in a list of predicates and a table as input, and returns all rows within the input table that satisfy each predicate within the provided list. The WHERE statement could also take in two tables as input, in which case, it returns all rows from within the two input tables that satisfy the list of predicates.

A predicate represents a comparison between two objects that evaluates to a Boolean value of either True or False. The details of this comparison differ based on the number of tables involved in the underlying operation. Every comparison within a predicate is based on the standard comparison operators of ==, <, >, <=, >= and !=.

In the case of a single table, a predicate is a comparison between a column within the table and either some string value or, another column within the table. Table columns indexed from zero and prefaced with an @ symbol whilst string values are enclosed in " ". Such a comparison is known as a **Column Comparison**, and has the following syntax.

```
-- Column Comparison between column and value
@<ColumnNumber> <ComparisonOperator> <StringValue>

-- Column comparison between column and column
@<ColumnNumber> <ComparisonOperator> @<ColumnNumber>
```

In the case of two tables, a predicate is a comparison between a column within the first table, and a column within the second table, with table columns indexed from zero and prefaced with an @ symbol. The first table involved in the comparison is known as the left table, and the second is known as the right. Such a comparison is known as a **Table Comparison** and has the following syntax.

```
-- Table comparison between the columns of two tables
LEFT.@<ColumnNumber> <ComparisonOperator> RIGHT.@<ColumnNumber>
```

WHERE statements take in a list of either column or table comparisons, and predicates can be combined using the standard Boolean operations of AND, OR and NOT to create complex predicates.

B.3.2 SELECT

The SELECT function can be used to select specific columns from a table, rows that satisfy some predicate, or a combination of both. The SELECT function has the following syntax.

```
-- Selecting all columns and rows from within a table
SELECT *
    FROM <TableObject>;

-- Selecting specific columns from within a table
SELECT [@<Column1>, ..., @<ColumnN>]
    FROM <TableObject>;

-- Selecting all columns of rows that satisfy some predicate
SELECT *
    WHERE [<ColumnComparison1>, ..., <ColumnComparisonN>]
    FROM <TableObject>;

-- Selecting specific columns of rows that satisfy some predicate
SELECT [@<Column1>, ..., @<ColumnN>]
    WHERE [<ColumnComparison1>, ..., <ColumnComparisonN>]
    FROM <TableObject>;
```

B.3.3 INSERT

The INSERT function can be used to insert a column or row of values into a table. When inserting values, the output table will contain a new row containing the values provided. The values must be of the same width as the table they are being inserted into. When inserting a column, the output table will have a null column at the index provided. The column index provided must be less than or equal to the width of the table. The INSERT function has the following syntax.

```
-- Inserting values into a table
INSERT
    VALUES [<Value1>, ..., <ValueN>]
    INTO <TableObject>;

-- Inserting a column into a table
INSERT
    COLUMN <ColumnNumber>
    INTO <TableObject>
```

B.3.4 DELETE

The DELETE function can be used to delete specific columns from a table or delete rows that satisfy some predicate. These actions cannot combined as all rows within the table must have the same width. The DELETE function has the following syntax.

```
-- Deleting an entire table
DELETE <TableObject>;

-- Deleting specific columns from a table
DELETE [@<Column1>, ..., @<ColumnN>]
    FROM <TableObject>;

-- Deleting rows from a table that satisfy some predicate
DELETE
    WHERE [<ColumnComparison1>, ..., <ColumnComparisonN>]
    FROM <TableObject>;
```

B.3.5 UPDATE

The UPDATE function can be used to update columns of an entire table, or update columns of rows that satisfy some predicate. The UPDATE function has the following syntax.

```
-- Updating all rows of a table
UPDATE
    TO [@<Column1> = <StringValue1>, ..., @<ColumnN> = <StringValueN>]
    ON <TableObject>;

-- Updating rows of a table that satisfy some predicate
UPDATE
    TO [@<Column1> = <StringValue1>, ..., @<ColumnN> = <StringValueN>]
    WHERE [<ColumnComparison1>, ..., <ColumnComparisonN>]
    ON <TableObject>;
```

B.3.6 Set Functions

CSVQL supports the use of set operations on table objects with UNION, INTERSECTION and DIFFERENCE functions.

UNION

UNION takes in two tables as input, and produces an output of the first table placed on top of the second (i.e., a single table with all rows from the both input tables). The two input tables must have the same width. The UNION function has the following syntax.

```
UNION <TableObject> AND <TableObject>;
```

INTERSECTION

INTERSECTION takes in two tables as input and produces an output table consisting of rows that are present only in both tables. The two input tables must have the same width. The INTERSECTION function has the following syntax.

```
INTERSECTION <TableObject> AND <TableObject>;
```

DIFFERENCE

DIFFERENCE takes in two tables as input and produces an output table consisting of rows that are present in the first table, but not present in the second table. The two input tables must have the same width. The DIFFERENCE function has the following syntax.

```
DIFFERENCE <TableObject> AND <TableObject>;
```

B.3.7 Joining Functions

CSVQL allows for two tables to be joined together (connected side by side) under various conditions through the use of several joining functions.

JOIN

JOIN performs a standard join between two tables. The JOIN function takes in two tables as input and produces an output table equivalent to the two input tables side by side. In the case where one input table has a greater length than the other, null values are used to create an output table of uniform width. The standard JOIN function has the following syntax.

```
JOIN ON <TableObject> AND <TableObject>;
```

JOIN INNER

JOIN INNER performs an inner join between two tables. The JOIN INNER function takes in two tables and a list of predicates as input, and produces an output table that contains rows from both tables that satisfy the predicates joined together. The JOIN INNER function has the following syntax.

```
JOIN INNER
    WHERE [<TableComparison1>, ..., <TableComparisonN>]
    ON <TableObject> AND <TableObject>;
```

JOIN LEFT

JOIN LEFT performs a left join between two tables. The JOIN LEFT function takes in two tables and a list of predicates as input, and produces an output table that contains rows from both tables that satisfy the predicates joined together, in addition to all rows in the left table. Rows in the left table that have no matches in the right table are joined to a series of null values. The JOIN LEFT function has the following syntax.

```
JOIN LEFT
    WHERE [<TableComparison1>, ..., <TableComparisonN>]
    ON <TableObject> AND <TableObject>;
```

JOIN RIGHT

JOIN RIGHT performs a right join between two tables. The JOIN RIGHT function takes in two tables and a list of predicates as input, and produces an output table that contains rows from both tables that satisfy the predicates joined together, in addition to all rows in the right table. Rows in the right table that have no matches in the left table are joined to a series of null values. The JOIN RIGHT function has the following syntax.

```
JOIN RIGHT
    WHERE [<TableComparison1>, ..., <TableComparisonN>]
    ON <TableObject> AND <TableObject>;
```

JOIN OUTER

JOIN OUTER performs an outer join between two tables. The JOIN OUTER function takes in two tables and a list of predicates as input, and produces an output table that contains rows from both tables that satisfy the predicates joined together, in addition to all rows in the left table and all rows from the right table. Rows from either table that have no matches are joined to a series of null values. The JOIN OUTER function has the following syntax.

```
JOIN OUTER
    WHERE [<TableComparison1>, ..., <TableComparisonN>]
    ON <TableObject> AND <TableObject>;
```

JOIN FULL

JOIN FULL performs a cross join between two tables. The JOIN FULL function takes in two tables as input and produces an output table that contains every row in the left table joined to every row in the right table (i.e., contains every possible pairing of rows). The JOIN FULL function has the following syntax.

```
JOIN FULL ON <TableObject> AND <TableObject>;
```

MERGE

MERGE merges two tables together based on some predicates. The MERGE function takes in two tables, a list of predicates and a list of column numbers and produces an output table that contains rows formed by merging each pair of rows that satisfy the given predicates together. Two matching rows are merged by taking values from the left table if they are non-null, and taking values from the right table otherwise. This condition is ignored for those columns specified in the provided list of column indexes. Note that the two input tables must have the same width. The syntax of the MERGE function is as follows.

```
MERGE
    WHERE [<TableComparison1>, ..., <TableComparisonN>]
    KEEPING [@<Column1>, ..., @<ColumnN>]
    ON <TableObject> AND <TableObject>;
```

B.3.8 Formatting Functions

CSVQL allows for tables to be formatted through the use of several formatting functions. Formatting functions are typically applied to a table object immediately before it's output.

ORDER

ORDER can be used to sort the rows of a table lexicographically. The function takes in a table and direction as input, and produces a sorted table as output. Table columns can also be provided as an additional parameter to specify the precedence of the sort, but are not required. In the case where no column numbers are provided, the function will sort in order of increasing columns (i.e., column 1, columns 2, and so on). The ORDER function has the following syntax.

```
-- Ordering across the whole table
ORDER
    IN <Direction>
    <TableObject>;

-- Ordering according to specific columns
ORDER
    IN <Direction>
    BY [<Column1>, ..., <ColumnN>]
    <TableObject>
```

Note that as the ORDER function sorts based on the lexicographical ordering of entries, tables of numbers may not be sorted in numerical ordering following the use of this function.

LIMIT

LIMIT can be used to limit the number of rows within a given table. The LIMIT function takes in an integer n and a table as input and produces an output table that contains only the first n rows of the input table. The value of n must be greater than zero less than or equal to the length of the table. The LIMIT function has the following syntax.

```
LIMIT <LimitValue> <TableObject>;
```

OFFSET

OFFSET can be used to skip/remove a number of rows from the start of a table. The OFFSET function takes in an integer n and a table as input, and produces an output table equivalent to the input table, but with the first n rows omitted. The value of n must be greater than zero less than the length of the table. The OFFSET function has the following syntax.

```
OFFSET <OffsetValue> <TableObject>;
```

LAST

LAST can be used to gather only the rows at the end of a table. The LAST function takes in an integer n and a table as input, and produces an output table that contains onnly the last n rows of the input table. The value of n must be greater than zero less than or equal to the length of the table. The LAST function has the following syntax.

```
LAST <LastValue> <TableObject>;
```

UNIQUE

UNIQUE can be used to remove repeated rows from a table. The UNIQUE function takes in a table as input, and produces an output table that is equivalent to the input table, but with any duplicate rows omitted. The UNIQUE function has the following syntax.

```
UNIQUE <TableObject>;
```

TRANSPOSE

TRANSPOSE can be used to transpose a table. The TRANSPOSE function takes in a table as input and produces an output table whose columns are the rows from the input table, and whose rows are columns from the input table. The TRANSPOSE function has the following syntax.

```
TRANSPOSE <TableObject>;
```

C Demonstration of Additional Features

Provided here are demonstrations of the additional features present in the CSVQL programming language, as discussed in Additional Features. These demonstrations are presented in the form of example programs, along with example interpreter output in the cases where this is required.

C.1 Function Composition

Provided below are two example programs that are operationally equivalent (i.e., produce the same output when given the same input). One of the programs is defined sequentially, with each function defined on a separate line, where as the other makes use of function composition to produce a more simplisitic program structure.

```
LET a = READ "A.csv";
LET b = READ "B.csv";

LET joinedTable = JOIN a AND b;

LET selectedTable = SELECT [@1, @2] FROM joinedTable;

LET orderedTable = ORDER IN ASC selectedTable;

RETURN orderedTable;
```

Listing 7: CSVQL Program with no function composition

```
LET a = READ "A.csv";
LET b = READ "B.csv";

RETURN ORDER IN ASC SELECT [@1, @2] FROM JOIN a AND b;
```

Listing 8: CSVQL Program with function composition

C.2 Predicate Composition

Provided below are two example programs that are operationally equivalent (i.e., produce the same output when given the same input). The first program queries a given table with multiple predicates using a series of statements, while the second combines predicates together to perform the same query in a single statement.

```
LET a = READ "A.csv";

LET selectedTable1 = SELECT * WHERE [@0 == "foo"] FROM a;

LET selectedTable2 = SELECT * WHERE [@1 == "bar"] FROM selectedTable1;

LET selectedTable3 = SELECT * WHERE [@2 == "sho"] FROM selectedTable2;

RETURN selectedTable3;
```

Listing 9: CSVQL Program with no predicate composition

```
LET a = READ "A.csv";

LET selectedTable = SELECT * WHERE [@0 == "foo" AND @1 == "bar" AND @2 == "sho"] FROM a;

RETURN selectedTable;
```

Listing 10: CSVQL Program with predicate composition

C.3 Comments

Provided below is an example CSVQL program that makes use of comments to improve readability and programmer understanding.

```
-- Gathering the table from the file
LET a = READ "A.csv";

-- Selecting the needed rows based on predicates
LET selectedTable = SELECT * WHERE [@0 == "foo" AND @1 == "bar" AND @2 == "sho"] FROM a;

-- Outputting the contents of the selection
RETURN selectedTable;
```

Listing 11: CSVQL Program with comments

C.4 Parenthesised Expressions

C.4.1 Table Objects

Provided below are two example programs that are operationally equivalent (i.e., produce the same output when given the same input). The second program makes use of parenthesised table objects to increase readability, while the first does not.

```
LET a = READ "A.csv";
LET b = READ "B.csv";

RETURN ORDER IN ASC SELECT [@1, @2] FROM JOIN a AND b;
```

Listing 12: CSVQL Program without parenthesised table objects

```
LET a = READ "A.csv";
LET b = READ "B.csv";

RETURN ORDER IN ASC (SELECT [@1, @2] FROM (JOIN a AND b));
```

Listing 13: CSVQL Program with parenthesised table objects

C.4.2 Predicates

Provided below are two example programs that are operationally equivalent (i.e., produce the same output when given the same input). The second program makes use of parenthesised predicates to increase readability, while the first does not.

```
LET a = READ "A.csv";

LET selectedTable = SELECT * WHERE [@0 == "foo" AND @1 == "bar" OR @2 == "sho"] FROM a;

RETURN selectedTable;
```

Listing 14: CSVQL Program without parenthesised predicates

```
LET a = READ "A.csv";

LET selectedTable = SELECT * WHERE [(@0 == "foo" AND @1 == "bar") OR @2 == "sho"] FROM a;

RETURN selectedTable;
```

Listing 15: CSVQL Program with parenthesised predicates

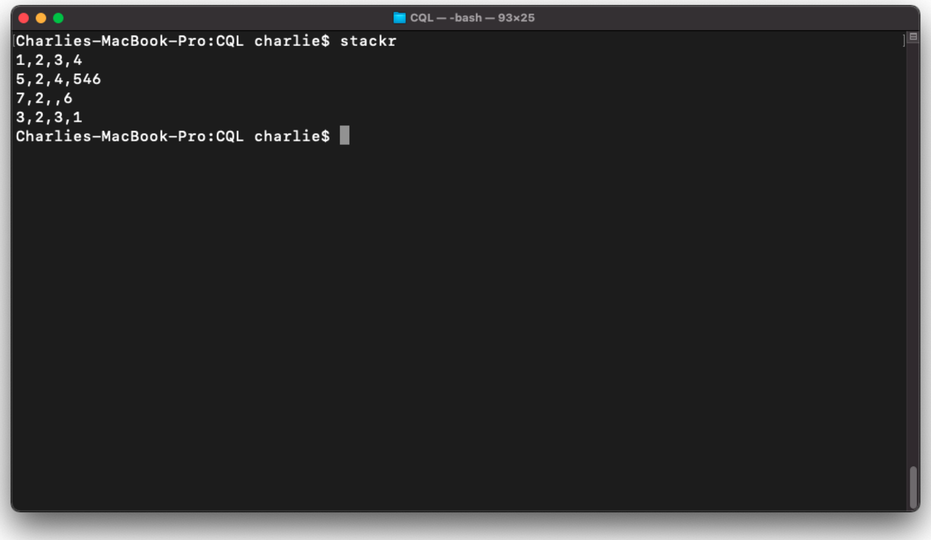
C.5 Program Configurations

C.5.1 Pretty Printing

Provided below are example CSVQL programs and their corresponding outputs. The first program does not make use of pretty printing while the second does.

```
LET a = READ "A.csv";

LET selectedTable = SELECT [@0, @1, @2, @3] WHERE [@1 == "2"] FROM a;

RETURN selectedTable;
```

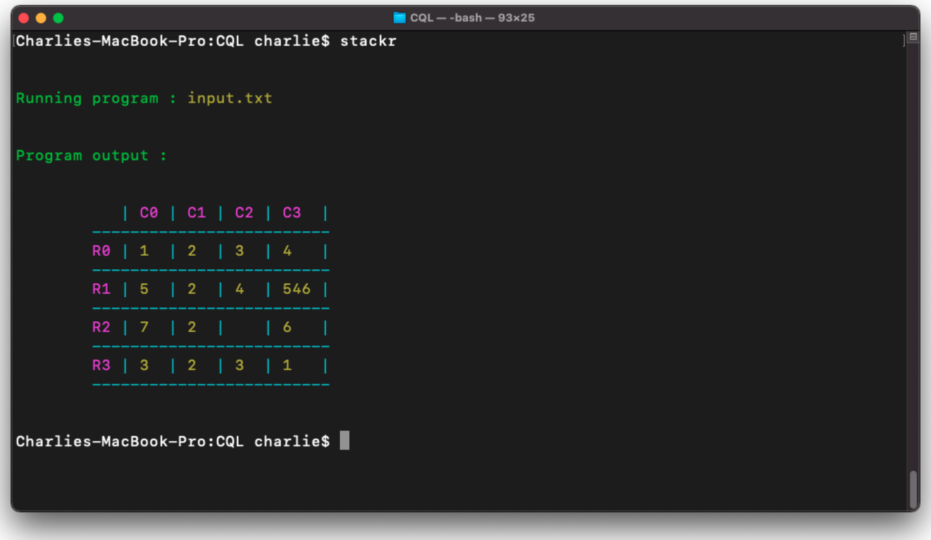
Listing 16: CSVQL Program with no pretty print configuration



```
Charlies-MacBook-Pro:CQL charlie$ stackr
1,2,3,4
5,2,4,546
7,2,,6
3,2,3,1
Charlies-MacBook-Pro:CQL charlie$ 
```

```
SETUP {
    PRETTYPRINT
}

LET a = READ "A.csv";

LET selectedTable = SELECT [@0, @1, @2, @3] WHERE [@1 == "2"] FROM a;

RETURN selectedTable;
```

Listing 17: CSVQL Program with pretty print configuration



```
Charlies-MacBook-Pro:CQL charlie$ stackr


Running program : input.txt


Program output :


         | C0 | C1 | C2 | C3  |
      -------------------------
      R0 | 1  | 2  | 3  | 4   |
      -------------------------
      R1 | 5  | 2  | 4  | 546 |
      -------------------------
      R2 | 7  | 2  |    | 6   |
      -------------------------
      R3 | 3  | 2  | 3  | 1   |
      -------------------------


Charlies-MacBook-Pro:CQL charlie$ 
```

C.5.2 Input/Output Delimiters

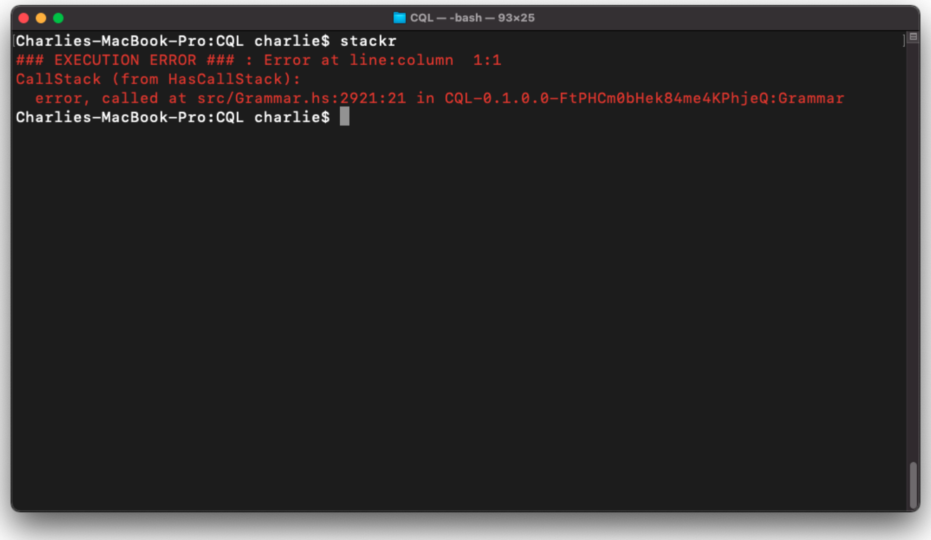
Provided below is an example CSVQL program that makes use of the INPUTDELIM and OUTPUTDELIM keywords to configure the format by which tables are loaded in from files and program results are outputted.

```
SETUP{
    INPUTDELIM="@"  -- Load table from file with "@" as the delimiter - e.g., 1@2@3
    OUTPUTDELIM="-" -- Output results with "-" as the delimiter - e.g., 1-2-3
}

LET a = READ "A.tsv";
LET b = READ "B.tsv";

LET joinedTable = JOIN FULL ON a AND b;

RETURN ORDER IN ASC joinedTable;
```

Listing 18: Example CSVQL Program with loading from TSV configuration

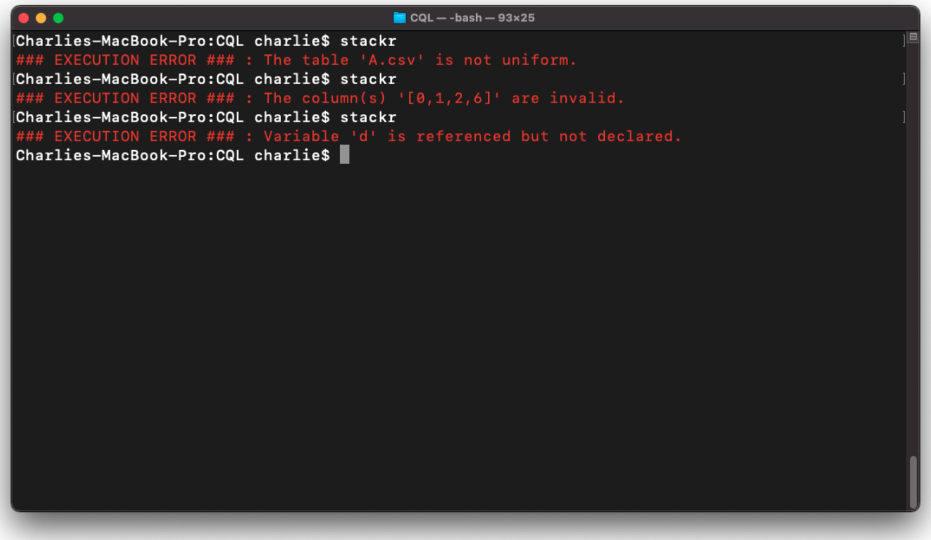
C.6 Error Messages

Provided below are images that showcase the error messages displayed by the CSVQL interpreter in the case of invalid programs. The first shows an error message displayed following a compilation error, while the second shows an error message displayed following a run-time error. Note that error messages displayed by the CSVQL interpreter are shown in red (as opposed to the standard white output) to improve readability for the user.



```
Charlies-MacBook-Pro:CQL charlie$ stackr
### EXECUTION ERROR ### : Error at line:column  1:1
CallStack (from HasCallStack):
  error, called at src/Grammar.hs:2921:21 in CQL-0.1.0.0-FtPHCm0bHek84me4KPhjeQ:Grammar
Charlies-MacBook-Pro:CQL charlie$
```

Figure 1: Example CSVQL compilation error



```
Charlies-MacBook-Pro:CQL charlie$ stackr
### EXECUTION ERROR ### : The table 'A.csv' is not uniform.
Charlies-MacBook-Pro:CQL charlie$ stackr
### EXECUTION ERROR ### : The column(s) '[0,1,2,6]' are invalid.
Charlies-MacBook-Pro:CQL charlie$ stackr
### EXECUTION ERROR ### : Variable 'd' is referenced but not declared.
Charlies-MacBook-Pro:CQL charlie$
```

Figure 2: Example CSVQL run-time errors

D Complete CSVQL Grammar

Expressions

$\langle \mathit{Program} \rangle$ ::= `SETUP` $\langle \mathit{CurlyList(Configuration)} \rangle$ $\langle \mathit{Exp} \rangle$
| $\langle \mathit{Exp} \rangle$

$\langle \mathit{Configuration} \rangle$ ::= `PRETTYPRINT`

$\langle \mathit{Exp} \rangle$ ::= `LET` $\langle \mathit{Var} \rangle$ ‘=’ $\langle \mathit{TableObject} \rangle$ ‘;’ $\langle \mathit{Exp} \rangle$
| `RETURN` $\langle \mathit{TableObject} \rangle$ ‘;’

Tables

```
⟨TableObject⟩      ::= READ ⟨Filename⟩
                    |  ⟨Var⟩
                    |  ⟨FunctionTable⟩
                    |  ‘(’ ⟨TableObject⟩ ‘)’

⟨FunctionTable⟩    ::= ⟨SelectFunction⟩
                    |  ⟨InsertFunction⟩
                    |  ⟨DeleteFunction⟩
                    |  ⟨UpdateFunction⟩
                    |  ⟨SetFunction⟩
                    |  ⟨JoinFunction⟩
                    |  ⟨FormatFunction⟩
```

Functions

```
⟨SelectFunction⟩   ::= SELECT ‘*’ FROM ⟨TableObject⟩
                    |  SELECT ⟨List (ColumnRef)⟩ FROM ⟨TableObject⟩
                    |  SELECT ‘*’ WHERE ⟨List (Predicate (ColumnComparison))⟩ FROM ⟨TableObject⟩
                    |  SELECT ⟨List(ColumnRef)⟩ WHERE ⟨List (Predicate (ColumnComparison))⟩ FROM ⟨TableObject⟩

⟨InsertFunction⟩   ::= INSERT VALUES ⟨List(Str)⟩ INTO ⟨TableObject⟩
                    |  INSERT COLUMN ⟨ColumnRef⟩ INTO ⟨TableObject⟩

⟨DeleteFunction⟩   ::= DELETE ⟨TableObject⟩
                    |  DELETE ⟨List(ColumnRef)⟩ FROM ⟨TableObject⟩
                    |  DELETE WHERE ⟨List (Predicate (ColumnComparison))⟩ FROM ⟨TableObject⟩

⟨UpdateFunction⟩   ::= UPDATE TO ⟨List (Assignment)⟩ ON ⟨TableObject⟩
                    |  UPDATE TO ⟨List (Assignment)⟩ WHERE ⟨List(Predicate (ColumnComparison))⟩ ON ⟨TableObject⟩

⟨SetFunction⟩      ::= UNION ⟨TableObject⟩ AND ⟨TableObject⟩
                    |  INTERSECTION ⟨TableObject⟩ AND ⟨TableObject⟩
                    |  DIFFERENCE ⟨TableObject⟩ AND ⟨TableObject⟩

⟨JoinFunction⟩     ::= JOIN ON ⟨TableObject⟩ AND ⟨TableObject⟩
                    |  JOIN INNER WHERE ⟨List (Predicate (TableComparison))⟩ ON ⟨TableObject⟩ AND ⟨TableObject⟩
                    |  JOIN LEFT WHERE ⟨List (Predicate (TableComparison))⟩ ON ⟨TableObject⟩ AND ⟨TableObject⟩
                    |  JOIN RIGHT WHERE ⟨List (Predicate (TableComparison))⟩ ON ⟨TableObject⟩ AND ⟨TableObject⟩
                    |  JOIN OUTER WHERE ⟨List (Predicate (TableComparison))⟩ ON ⟨TableObject⟩ AND ⟨TableObject⟩
                    |  JOIN FULL ON ⟨TableObject⟩ AND ⟨TableObject⟩
                    |  MERGE WHERE ⟨List (Predicate (TableComparison))⟩ KEEPING ⟨List(ColumnRef)⟩ ON ⟨TableObject⟩ AND
                       ⟨TableObject⟩

⟨FormatFunction⟩   ::= ORDER IN ⟨Direction⟩ ⟨TableObject⟩
                    |  ORDER IN ⟨Direction⟩ By ⟨List (ColumnRef)⟩ ⟨TableObject⟩
                    |  LIMIT ⟨Int⟩ ⟨TableObject⟩
                    |  OFFSET ⟨Int⟩ ⟨TableObject⟩
                    |  LAST ⟨Int⟩ ⟨TableObject⟩
                    |  UNIQUE ⟨TableObject⟩
                    |  TRANSPOSE ⟨TableObject⟩

⟨Direction⟩        ::= ASC
                    |  DESC
```

Predicates

```
⟨Predicate (Comparison) ⟩  ::= NOT ⟨Predicate (Comparison)⟩
                            |  ⟨Predicate (Comparison)⟩ AND ⟨Predicate (Comparison)⟩
                            |  ⟨Predicate (Comparison)⟩ OR ⟨Predicate (Comparison)⟩
                            |  ⟨Comparison⟩
                            |  ‘(’ ⟨Predicate (Comparison)⟩ ‘)’
```

```
⟨Comparison⟩              ::= ⟨ColumnComparison⟩
                           |  ⟨TableComparison⟩

⟨ColumnComparison⟩        ::= ⟨ColumnRef⟩ ⟨ComparisonOperator⟩ ⟨Str⟩
                           |  ⟨ColumnRef⟩ ⟨ComparisonOperator⟩ ⟨ColumnRef⟩
                           |  INDEX ⟨Operator⟩ ⟨Int⟩ ⟨ComparisonOperator⟩ ⟨Int⟩

⟨TableComparison⟩         ::= LEFT ‘.’ ⟨ColumnRef⟩ ⟨ComparisonOperator⟩ RIGHT ‘.’ ⟨ColumnRef⟩

⟨ComparisonOperator⟩      ::= ‘==’
                           |  ‘<’
                           |  ‘>’
                           |  ‘<=’
                           |  ‘>=’
                           |  ‘!=’
```

Object Referencing

```
⟨ColumnRef⟩      ::= ‘@’ ⟨Int⟩

⟨TableColumnRef⟩ ::= ⟨Var⟩ ‘.’ ⟨ColumnRef⟩

⟨Assignment⟩     ::= ⟨ColumnRef⟩ ‘=’ ⟨Str⟩
```

Operators

```
⟨Operator⟩       ::= ‘+’
                  |  ‘-’
                  |  ‘/’
                  |  ‘*’
                  |  ‘%’
```

Values

```
⟨Var⟩            ::= ⟨Alpha⟩ [⟨Alpha⟩ ‘_’ ⟨Digit⟩]*

⟨Str⟩            ::= [⟨Alpha⟩ ⟨White⟩ ‘_’ ‘,’ ⟨Digit⟩]*

⟨Filename⟩       ::= ¨[⟨Alpha⟩ ⟨White⟩ ‘_’ ‘,’ ⟨Digit⟩]* ‘.’ [⟨Alpha⟩ ⟨White⟩ ‘_’ ‘,’ ⟨Digit⟩]*

⟨Int⟩            ::= [⟨Digit⟩]+

⟨Alpha⟩          ::= [a - z]
                  |  [A - Z]

⟨Digit⟩          ::= [0 - 9]

⟨White⟩          ::= ¨ ¨
```

Lists

```
⟨List (a)⟩          ::= ‘[’ ‘]’
                     |  ‘[’ ⟨ListCont (a)⟩ ‘]’

⟨ListCont (a)⟩      ::= ⟨a⟩
                     |  ⟨a⟩ ‘,’ ⟨listCont (a)⟩

⟨CurlyList (a)⟩     ::= ‘{’ ‘}’
                     |  ‘{’ ⟨CurlyListCont (a)⟩ ‘}’

⟨CurlyListCont (a)⟩ ::= ⟨a⟩
                     |  ⟨a⟩ ‘,’ ⟨CurlylistCont (a)⟩
```