COMP2212 Programming Language Concepts: CSVQL Manual - Group 26

Matthew Webb - mw5g23
Daya Gill - dg6g23
Shichen Zhang - sz3e22

May 13, 2025

1 Complete CSVQL Grammar

Grammar

⟨Grammar⟩ ::= `output` = ⟨Expression⟩`;`
| ⟨Assignment⟩ `output` = ⟨Expression⟩`;`

Assignment

⟨Assignment⟩ ::= ⟨Type⟩ `var` = ⟨Expression⟩`;`
| ⟨Assignment⟩ ⟨Type⟩ `var` = ⟨Expression⟩`;`

Types

⟨Type⟩ ::= `Int`
| `Bool`
| `String`
| `Grid`

Grids

⟨Arr⟩ ::= `[`⟨Row⟩`]`
| `[`⟨Row⟩`],`⟨Arr⟩

⟨Row⟩ ::= ⟨Expression⟩
| ⟨Expression⟩`,`⟨Row⟩

Expressions

⟨Expression⟩ ::= `(`⟨Expression⟩`)`
| ⟨Var⟩
| ⟨String⟩
| ⟨Int⟩
| `true`
| `false`
| `[`⟨Arr⟩`]`
| `[[ ]]`
| `read` ⟨Expression⟩
| `addRows` ⟨Expression⟩ ⟨Expression⟩
| `addCols` ⟨Expression⟩ ⟨Expression⟩
| `insertRows` ⟨Expression⟩ ⟨Expression⟩ ⟨Expression⟩
| `insertCols` ⟨Expression⟩ ⟨Expression⟩ ⟨Expression⟩
| `dropRow` ⟨Expression⟩ ⟨Expression⟩
| `dropCol` ⟨Expression⟩ ⟨Expression⟩
| `reverseRow` ⟨Expression⟩ ⟨Expression⟩
| `reverseCol` ⟨Expression⟩ ⟨Expression⟩
| `containsStr` ⟨Expression⟩ ⟨Expression⟩
| `containsCell` ⟨Expression⟩ ⟨Expression⟩
| `containsRow` ⟨Expression⟩ ⟨Expression⟩
| `containsCol` ⟨Expression⟩ ⟨Expression⟩
| `containsChar` ⟨Expression⟩ ⟨Expression⟩

| `getStr` ⟨Expression⟩ ⟨Expression⟩ ⟨Expression⟩
| `getCell` ⟨Expression⟩ ⟨Expression⟩ ⟨Expression⟩
| `getRow` ⟨Expression⟩ ⟨Expression⟩
| `getCol` ⟨Expression⟩ ⟨Expression⟩
| `getChar` ⟨Expression⟩ ⟨Expression⟩

| `numRows` ⟨Expression⟩
| `numCols` ⟨Expression⟩
| `numChars` ⟨Expression⟩

| ⟨Expression⟩ `++` ⟨Expression⟩
| `intToStr` ⟨Expression⟩

| ⟨Expression⟩ `&&` ⟨Expression⟩
| ⟨Expression⟩ `||` ⟨Expression⟩

| ⟨Expression⟩ `==` ⟨Expression⟩
| ⟨Expression⟩ `!=` ⟨Expression⟩
| ⟨Expression⟩ `<` ⟨Expression⟩
| ⟨Expression⟩ `>` ⟨Expression⟩
| ⟨Expression⟩ `<=` ⟨Expression⟩
| ⟨Expression⟩ `>=` ⟨Expression⟩

| ⟨Expression⟩ `+` ⟨Expression⟩
| ⟨Expression⟩ `-` ⟨Expression⟩
| ⟨Expression⟩ `*` ⟨Expression⟩
| ⟨Expression⟩ `/` ⟨Expression⟩
| ⟨Expression⟩ `%` ⟨Expression⟩
| `-` ⟨Expression⟩

| `repeatRow(`⟨Type⟩ ⟨Var⟩ `=` ⟨Expression⟩`,`⟨Expression⟩`)` ⟨Expression⟩
| `repeatCol(`⟨Type⟩ ⟨Var⟩ `=` ⟨Expression⟩`,`⟨Expression⟩`)` ⟨Expression⟩
| `repeatWhile(`⟨Type⟩ ⟨Var⟩ `=` ⟨Expression⟩`,`⟨Expression⟩`)` ⟨Expression⟩

| `if (`⟨Expression⟩`) then (`⟨Expression⟩`) else (`⟨Expression⟩`)`

Values

⟨Var⟩ ::= ⟨Alpha⟩`[`⟨Alpha⟩ | ⟨Digit⟩`]`*
⟨String⟩ ::= `"[`⟨Alpha⟩ | ⟨Digit⟩`]`*`"`
⟨Int⟩ ::= `[`⟨Digit⟩`]`$^+$
⟨Alpha⟩ ::= `[`⟨a-z⟩ | ⟨A-Z⟩`]`
⟨Digit⟩ ::= `[`⟨0-9⟩`]`

2 CSVQL Grammar And Variables

CSVQL is a domain specific programming language for querying and modifying CSV data. We chose to design an imperative language, largely because our group was most familiar with this style. We also felt that a designing a sequential language (Where all programs are expressed as a clear sequence of instructions) would eventually make it easier to conceptualize and reason how programs would execute.

2.1 Grammar

A CSVQL programs is composed of a sequence of variable assignments, each delimited by a semicolon (;). Assignments are processed sequentially, in the order in which they appear in the program.

The final assignment in any CSVQL program should be to variable `output`. After execution completes, the value assigned to output is automatically printed to stdout. If no value is assigned to `output`, or if `output` is assigned earlier than the final line, the program will fail to compile with a syntax error.

Each expression in CSVQL is composed of one or more functions. All functions accept one or more parameters as input, and produce a new value as output.

2.2 Variables And Assignments

To improve clarity and avoid redundant code, CSVQL allows values to be bound to variables. Each line of CSVSQL must begin with a variable assignment. All variables must be declared with:

- A type (Explicitly stated).
- A name (Which must begin with a character and cannot contain any reserved words).
- A value (Which can itself be an expression).

Once declared, the variable name can be used as a placeholder for the assigned value for the remainder of the program.

```
1 Int a = 20;
2 Int b = 10;
3
4 Bool c = a > b;
```

2.2.1 Typing Rules

CSVQL is both statically and strongly typed, each variable is explicitly assigned a type at the time of its declaration, and this cannot change thereafter. Each variable can be assigned one of four types:

- Int (Floating point values are not supported by CSVQL as the goal of numerals is generally to identify the rows and columns of a grid, rather than arithmetic.)
- Bool
- String (Strings must be delimited by quotation marks to differentiate them from variables, by extension this means that variable names cannot contain quotation marks.)
- Grid

To simplify the type checking process, CSVQL does not support null values. This means that that every expression must evaluate to one of the four supported types. In the case that the type produced by an expression does not match the type of its assigned variable, a type error will be thrown at compile time.

2.2.2 Scope And Mutability

All explicitly assigned variables are both global in scope and final - that is, once a value has been assigned to a variable, it can be accessed from any part of the program, but can never be reassigned. Within each Loop construct however, it is possible to assign a single iteration variable. These variables are scoped locally to the iteration block and can be reassigned on each pass of the loop. The lifetime of an iteration variable is limited to the duration of the iteration itself, and each iteration block must still produce a single value upon completion.

2.2.3 Variable Overwriting

If a variable name is declared multiple times within a CSVQL program, the most recent assignment overwrites the previous value assigned to the variable. Additionally, iteration variables temporarily overwrite global variables (Or variables from any broader scope) with the same name during their scope.

2.2.4 The Output Assignment

The final assignment in any CSVQL program must be to variable *output*. While it should not be explicitly assigned, the type of this variable should always be of type Grid, and attempting to assign a different type will result in a type error at compile time.

```
Grid a = read "A.csv";
output = a;
```

2.3 Type Grid And Grid Definitions

In CSVQL, an value of type Grid represents a structure that that can be evaluated to a valid CSV file. For a Grid to be considered valid in CSVQL, all of its rows must have the same number of elements. In other words, the Grid must be rectangular. The CSVQL executor verifies this requirement at runtime and if a grid is found to be non-rectangular, execution will terminate with an error.

There are two methods for defining a grid value in CSVQL:

2.3.1 File Reference

The primary method of creating a grid is by reading it from an existing CSV file. Currently, CSV files can only be read from the same directory in which the program is stored. If the specified file is not present, or if the file contents are not valid CSV, a runtime error will be thrown.

```
Grid a = read "A.csv";
```

2.3.2 Explicit Declaration

Alternatively, grids can be also be explicitly declared using square bracket notation (`[]`). For added flexibility, grids defined in this way can take an expression as input, provided it is of type Int or String.[1]

Several explicit grid definitions, along with their equivalent CSV counterparts, are shown below:

```
Grid a = [["a","b","c"]];
-- a,b,c

Grid b = [["a"],["b"],["c"]];
-- a
-- b
-- c

Grid c = [[1+2,10/2,5-3],["d","e","f"]];
-- 3,5,2
-- d,e,f
```

1. Although allowing grids to be defined this way exposes the programmer to their internal representation, most programmers should already be familiar writing arrays in this syntax, and we felt there should be a way to construct grids manually when needed, without having to read from a file.

3 CSVQL Built In Functions

CSVQL provides a number of built in functions, which can be used to transform and analyze data within programs.

- All functions in CSVQL are statically typed. Each built-in function has explicitly defined input and output types and these are enforced by the type checker at compile time. If the actual arguments passed to a function do not match the expected input types, a type error will be raised during compilation and the program will not execute.
- In cases where a function has preconditions for execution (For instance an index must be in range of a grid), these are checked at runtime. If any conditions are not met, a runtime error will be thrown, and execution will stop.
- Where an index is required as function input, indexing begins at 0 as this practice is likely familiar to most programmers.

3.1 Grid Functions

3.1.1 Add

The `addRows` and `addCols` functions combine two grids either vertically (`addRows`) or horizontally (`addCols`). To perform `addRows`, the number of columns in both grids must be equal, and to perform `addCols`, the number of rows in both grids must be equal.

```
addRows <Grid> <Grid> → <Grid>
addCols <Grid> <Grid> → <Grid>

Grid a =  [["a","b","c"]];
-- a,b,c

Grid b = addRows a [["d","e","f"]];
-- a,b,c
-- d,e,f

Grid c = addRows a ["g","h"];
-- Error: Incompatible Grids.

Grid d = addCols a [["w","x"],["y,","z"]];
-- a,b,c,w,x
-- d,e,f,y,z
```

3.1.2 Insert

The `insertRows` and `insertCols` functions take two grids and an index, and vertically (`insertRows`) or horizontally (`insertRows`) insert the second grid into the first at the specified index. To perform `insertRows`, the number of columns in both grids must be equal, and to perform `insertCols`, the number of rows in both grids must be equal. In both cases, the index must be in range of the first grid.

```
insertRows <Grid> <Grid> <Int> → <Grid>
insertCols <Grid> <Grid> <Int> → <Grid>

Grid a = [["a","b","c"],["d","e","f"]];
-- a,b,c
-- d,e,f

Grid b = insertRows a ["g","h","i"] 1;
-- a,b,c
-- g,h,i
-- d,e,f

Grid c = insertCols a [["w","x"],["y,","z"]] 2;
-- a,b,w,x,c
-- d,e,y,z,f

Grid d = insertCols a ["g","h","i"] 4;
-- Error: Index 4 is out of bounds for Grid a.
```

3.1.3 Drop

The `dropRow` and `dropCol` functions take a grid and an index, and removes the row (`dropRow`) or column (`dropCol`) located at the specified index from the grid. To perform either function, the index must be in range of the grid.

```
dropRow <Grid> <Int> → <Grid>
dropCol <Grid> <Int> → <Grid>

Grid a = [["a","b","c"],["d","e","f"]];
-- a,b,c
-- d,e,f

Grid b = dropRow 0 a;
-- d,e,f

Grid c = dropCol 1 a;
-- a,c
-- d,f
```

3.1.4 Reverse

The `reverseRow` and `reverseCol` functions take a grid and an index, and reverse the row (`reverseRow`) or column (`reverseCol`) located at the specified index. To perform either function, the index must be in range of the grid.

```
reverseRow <Grid> <Int> → <Grid>

Grid a = [["a","b","c"],["d","e","f"]];
-- a,b,c
-- d,e,f

Grid b = reverseRow 0 a;
-- c,b,a
-- d,e,f

Grid c = reverseCol 1 a;
-- a,e,c
-- d,b,f
```

3.1.5 Get

The `getRow` and `getCol` functions take a grid and an index, and return the row (`getRow`) or column (`getCol`) located at the specified index. The `getCell` function takes a grid and two indices, and returns the cell (Type Grid) located at the specified row and column. To perform any `get` function, all indices must be within bounds.

```
getRow <Grid> <Int> → <Grid>
getCol <Grid> <Int> → <Grid>
getCell <Grid> <Int> <Int> → <Grid>

Grid a = [["a","b","c"],["d","e","f"]];
-- a,b,c
-- d,e,f

Grid b = getRow a 0;
-- a,b,c

Grid c = getCol a 2;
-- c
-- f

Grid b = getCell a 1 2;
-- f

Grid c = getCell a 2 4;
-- Error: Index 2,4 is out of bounds for grid a.
```

3.2 Integer Functions

3.2.1 Arithmetic

Five arithmetic operators are available: Addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and modulo (`%`). As CSVQL does not support floating point values, division is always integer division, discarding any remainder.

```
<Int> + <Int> → <Int>
<Int> - <Int> → <Int>
<Int> * <Int> → <Int>
<Int> / <Int> → <Int>
<Int> % <Int> → <Int>
```

3.2.2 Num

The `numRows` and `numCols` functions take a grid as input, and return the number of rows (`numRows`) or columns (`numCols`) contained in the grid.

```
numRows <Grid> → <Int>
numCols <Grid> → <Int>

Grid a = [["a","b","c"],["d","e","f"]];
-- a,b,c
-- d,e,f

Int b = numRows a;
-- 2

Int c = numCols a;
-- 3
```

3.3 Boolean Functions

3.3.1 Boolean Algebra

CSVQL supports boolean algebra operations AND (`&&`) and OR (`||`) on boolean values.

```
<Bool> && <Bool> → <Bool>
<Bool> || <Bool> → <Bool>
```

3.3.2 Comparisons

CSVQL supports equality (`!=`) and inequality (`==`) comparisons for values of all types. Additionally, values of type Int can be compared using relational operators (`<`, `>`, `<=`, `>=`).

```
<Int> == <Int> → <Bool>
<Bool> == <Bool> → <Bool>
<String> == <String> → <Bool>
<Grid> == <Grid> → <Bool>

<Int> != <Int> → <Bool>
<Bool> != <Bool> → <Bool>
<String> != <String> → <Bool>
<Grid> != <Grid> → <Bool>

<Int> < <Int> → <Bool>
<Int> > <Int> → <Bool>
<Int> <= <Int> → <Bool>
<Int> >= <Int> → <Bool>
```

3.3.3 Contains

The `containsCell`, `containsRow`, and `containsCol` functions all determine if a smaller grid is contiguously contained within a larger grid. When performing `containsCell`, the smaller grid must have only one column and one row. When performing either `containsRow`, or `containsCol`, the number of columns (`containsRow`) or rows (`containsCol`) in both grids must be equal.

```
containsCell <Grid> <Grid> → <Bool>
containsRow <Grid> <Grid> → <Bool>
containsCol <Grid> <Grid> → <Bool>

Grid a = [["a","b","c"],["d","e","f"]];
-- a,b,c
-- d,e,f

Bool b = containsCell ["b"] a;
-- true

Bool c = containsRow ["a","b","c"] a;
-- true

Bool d = containsRow ["a","b","d"] a;
-- false

Bool e = containsCol ["a","d"] a;
-- true

Bool f = containsCol ["a","b","c"] a;
-- Error: Incompatible grids.
```

The `containsStr` function works similarly to `containsCell`, taking a grid and a String, and returning true if the String is contained in the grid. The `containsChar` function takes two Strings, and returns true if the second is contained contiguously in the first.

```
containsStr <Grid> <String> → <Bool>
containsChar <String> <String> → <Bool>

Grid a = [["a","b","c"],["d","e","f"]];
-- a,b,c
-- d,e,f

Bool b = containsStr "b" a;
-- true

Bool c = containsStr "bb" a;
-- false

Bool d = containsChar "hello" "ell";
-- true
```

3.4 String Functions

3.4.1 Conversion

The *intToStr* takes an Int, and returns a value of type String representing the integer, meaning operations that require values of type String can also be applied to values of type Int.

```
intToStr <Int> → <String>

Int a = 20;

String b = intToStr a;
-- "20"
```

3.4.2 Concatenation

String concatenation can be performed using the (++) operator, which joins two strings together into a new string.

```
<String> ++ <String> → <String>

String a = "Hello"
String b = "World"

String c = a ++ b;
-- "HelloWorld"
```

3.4.3 Get

The `getStr` function works similarly to `getCell`, taking a grid and two indices, and returns the string located at the specified row and column of the grid. To perform `getStr`, both indices must be within bounds.

```
getStr <Grid> <Int> <Int> → <String>

Grid a = [["a","b","c"],["d","e","f"]];
-- a,b,c
-- d,e,f

Grid b = getCell a  1 2;
-- Gets the cell (So type Grid) at index 1,2.
-- f

Grid c = getCell a 2 4;
-- -- Error: Index 2,4 is out of bounds for grid a.
```

The `getChar` function returns a single character from a string at a specified index. The index must be within the range of the string.

```
getChar <String> <Int> → <String>

String a = "Hello"

String b = getChar 1 a;
-- "e"

String c = getChar 5 a;
-- Error: Index 5 is out of bounds for string a.
```

3.5 Selection

CSVQL supports conditional selection through `if-then-else` expressions. These take a boolean condition and two values of the same type (As the type of the expression must be known at compile time). If the condition evaluates to true, the first value is returned; otherwise, the second is returned. Unlike all other functions in CSVQL, to simplify parsing and improve readability, parentheses are mandatory in selection statements.

```
if <Boolean> then <Int> else <Int> → <Int>
if <Boolean> then <Bool> else <Bool> → <Bool>
if <Boolean> then <String> else <String> → <String>
if <Boolean> then <Grid> else <Grid> → <Grid>

String a = if (1 < 2) then ("t") else ("f");
-- t

Grid b = [[1,2,3]];
-- 1,2,3

Grid c = if (numRows b > 1) then (b) else (addRows b b);
-- 1,2,3
-- 1,2,3
```

3.6 Iteration

3.6.1 Repeat For

The constructs `repeatRows` and `repeatCols` allow grids to be built by repeatedly evaluating an expression across a range of integer values. During each iteration, the current index is assigned to an iteration variable, which can then be used within the expression. Each `repeatRows` and `repeatCols` statement must receive:

- An iteration variable assigned to a start value of type Int.
- An end value of type Int.
- An expression or type Grid.

Execution then proceeds as follows:

1. Initialize the result of the repeat statement to an empty grid (`[[]]`).
2. Assign the start value to the iteration variable.
3. Check whether the start value is greater than or equal to the end value.
4. If true, terminate and return the current result.
5. If false:
   - Evaluate the expression using the current value of the iteration variable.
   - Append the evaluation to the current result using `addRows` (for `repeatRows`) or `addCols` (for `repeatCols`).
   - Increment the iteration variable by 1 and repeat from step 2.

```
repeatRows (<TypeInt> <Var> = <Int>, <Int>) <Grid> → <Grid>
repeatCols (<TypeInt> <Int> = <Int>, <Int>) <Grid> → <Grid>

-- Use a repeatCols statement to construct a grid from 0 to 9.
Grid b = repeatCols (Int var = 0, 10) [[var]];
-- 0,1,2,3,4,5,6,7,8,9

-- Use a repeatRows statement to add an empty cell to the end of each row.
Grid c = [[1],[2],[3]]

Grid d = repeatRows (Int var = 0, numRows c) addCols (getRow c var) [[""]];
-- 1,
-- 2,
-- 3,
```

3.6.2 Repeat While

The `repeatWhile` construct repeatedly evaluates an expression as long as a condition evaluates to true. A `repeatWhile` statement must receive:

- An iteration variable assigned to an initial value.
- A boolean condition.
- An expression of the same type of the iteration variable.

The result of a `repeatWhile` expression is the final value of the iteration variable when the condition evaluates to false.

Execution then proceeds as follows:

1. Evaluate the initial value and assign it to the iteration variable.
2. Evaluate the condition, using the current value of the iteration variable.
3. If the condition is false, terminate and return the current value of the iteration variable.
4. If the condition is true, evaluate the expression, assign the result to the iteration variable, and repeat from step 2.

```
repeatWhile (<TypeInt> <Var> = <Int>, <Bool>) <Int> → <Int>
repeatWhile (<TypeBool> <Var> = <Bool>, <Bool>) <Bool> → <Bool>
repeatWhile (<TypeString> <Var> = <String>, <Bool>) <String> → <String>
repeatWhile (<TypeGrid> <Var> = <Grid>, <Bool>) <Grid> → <Grid>

Grid a = [["","","",1,2,3]]

-- Use a while statement to trim leading empty entries from a row.
Grid b = repeatWhile (Grid var = a, getStr var 0 0 == "") dropCol var 0;
-- 1,2,3
```

4 Programmer Convenience And Additional Features

4.1 The Empty Grid

It is important to understand the structurally empty grid `[[]]`. While it's CSV counterpart is as expected (An empty CSV file), it behaves very differently to other values of type Grid when used in expressions.

In functions where where compatible grids are required (For instance `addRows`), the empty grid is not constrained by that requirement, and will automatically resize to become compatible with the other operand. At first, this seems unimportant:

```
Grid a = [["a","b","c"]];
-- a,b,c

Grid b = addRows a [[]];
-- a,b,c

-- Adding the empty Grid acheives nothing.
```

The empty grid becomes useful however, is when using a *repeat* statement to build a Grid. In this case, the empty grid can be used to skip iterations, regardless of the size of the Grid. For example

```
-- Assume an arbitrary grid of length n.
Grid a = read "A.csv";

-- Remove all rows with odd indexes from the Grid a.
Grid c = repeatRows (row, 0, numRows a)
if (row % 2 == 0)
then (getRow a row)
else ([[]])

-- Regardless of n, the above expression will always evaluate correctly.
```

4.2 Function Composition

All functions in CSVQL are able to take other functions as input. This allows for function composition. The result of this is that any multi line CSVQL program such as:

```
Grid a = repeatCol (col, 0, 5) intToStr col;

output = addRows a [["x", "y", "z"]];
```

Can also be expressed as a semantically equivalent single line program:

```
output = addRows (repeatCol (col, 0, 5) intToStr col) [["x", "y", "z"]];
```

Whilst this allows programmers to write more elegant and concise code, over use of function composition will naturally lead to less readable programs.

4.3 Parentheses

Where it improves legibility, all CSVQL functions can optionally be enclosed in parentheses without altering the semantics of the function.

```
Grid a = repeatCol (col, 0, 5) intToStr col;

Grid b = repeatCol (col, 0, 5) (intToStr col);

-- Both a and b evaluate to the same result.
```

4.4 Comments

CSVQL supports Haskell-style comments using the "`--`" symbol, any text following this symbol on a line will be ignored by the interpreter. Inline commenting is supported.

```
-- This is a full-line comment.

Int a = 1; -- This is an inline comment.
```

4.5 Syntax Highlighting

To make programs easier to read and debug, syntax highlighting for CSVQL is available for Notepad++.

The configuration file is attatched in the first submission, and screenshots of highlighted CSVQL programs are shown below:

```
-- t1.cql

Grid a = read "a.csv";
Grid b = read "b.csv";

Grid c = repeatRows (Int row = 0, numRows a)
(repeatRows (Int row2 = 0, numRows b) addCols (getRow a row) (getRow b row2));

output = [[]];
```

```
-- t4.cql

Grid a = read "A.csv";

Grid b = repeatRows (Int row = 0, numRows a) [["foo"]];

Grid c = addCols a b;

Grid d = addCols c a;

output = d;
```

```
-- t5.cql

Grid p = read "P.csv";

Grid q = read "Q.csv";

Grid a = repeatRows(Int row = 0, numRows p) repeatRows(Int row2 = 0, numRows q)
if ((getStr p row 0) == (getStr q row2 0))
then (addCols (getCell p row 0) (repeatCols(Int col = 1, 4) if ((getStr p row col) == "")
then (getCell q row2 col) else (getCell p row col)))
else ([[]]);

output = a;
```

5 Execution Model

CSVQL programs follow a two-phase execution model: compile time and runtime. This model simplifies the execution process by ensuring that all parse and type errors are caught before execution begins.

5.1 Compile Time

Before execution, CSVQL programs are first analyzed by a compilation pipeline consisting of three components: A lexer, parser, and static type checker.

1. The lexer accepts the source code as input, removes whitespace and comments, and produces a list of tokens.
2. The parser receives the list of tokens from the lexer, and constructs an abstract syntax tree (AST).
3. The type checker then traverses the AST checks that the program is well typed by building a full type environment. The type checker verifies:
   - Every function receives inputs of the correct type.
   - Every variable is assigned a value matching its declared type.

   This process guarantees that no type errors can be thrown at runtime.

5.2 Runtime

Once the program has passed type checking, the result of parsing is passed to the executor. At runtime:

- Each assignment is evaluated in order. For each assignment, the pair (variableName, value) is added to a variable environment.
- Once the pair (output, outputValue) has been added to the environment, execution has completed and outputValue is printed to `stdout` (After being converted to CSV format).

5.3 Compile Time Errors

- Lexical error at line <Row>, column <Column>.
- Parse error at <Row> <Column>.
- Couldn't find assignment for <Variable> - Indicates that a variable is referenced before it has been declared.
- Couldn't match expected type <Type> with actual type <Type> in assignment - <Variable>
- Couldn't match expected type <Type> with actual type <Type> in function <Function>

5.4 Runtime Errors

- File <File> does not exist. - The program attempted to read a file that is not present in the current directory.
- Index <Index> out of range for grid size <Size> in function <Function>
- Incompatible grid sizes in function <Function>
- Invalid grid construction - Occurs when a grid is non-rectangular.[2]
- Division by zero in function <Function>.

2. While grid validity should intuitively be enforced at compile time, early versions of CSVQL did not check for grid compatibility within functions, with validation instead being deferred until after function execution (Requiring this error to be thrown at runtime). While this eventually changed, grid validity checks continued to be performed at runtime.

5.5 Lazy Evaluation

In order to avoid unnecessary computation, CSVQL exploits lazy evaluation (Inherited from its Haskell implementation). Assignments are only evaluated when they contribute to the value of `output`. If this is not the case, the values assigned to such assignments are not type checked or evaluated. By extension, no type or runtime errors can be caused by these assignments:

```
Int a = "Hello"; -- Would normally cause a type error.
Grid b = [["a","b"],["c"]]; -- Would normally cause a runtime error.

output = [["x"]];

-- Despite the errors that should be caused by the first two assignments, the above program is still
    considered well typed and executes without error, because a and b are never used.
```

6 Example CSVQL Programs

6.1 Valid CSVQL Programs

```
Grid a = [[1,2,3],[4,5,6],[7,8,9],[10,11,12]];
Grid b = [[4,5,6],[13,14,15]];

-- Select all rows from a that are contained in b
output = repeatRows (Int row = 0, numRows a)
if (containsRow b (getRow a row)) then (getRow a row) else ([[]]);
-- 4,5,6
```

```
--  Given a grid of arity 2
Grid a = read "A.csv"

-- Output all rows where the first column is non-null
output = repeatRows (Int row = 0, numRows a)
if (getStr a row 0 == "") then (getRow a row) else ([[]]);
```

```
-- Given a grid of arbitrary arity
Grid a = read "A.csv"

-- For each row, output row "a" row
output = repeatRows (Int row = 0, numRows a)
addCols (addCols ((getRow a row) "a") (getRow a row))
```

6.2 Invalid CSVQL Programs

```
-- The following program is invalid as it does not contain an output assignment
Grid a = read "A.csv";

Grid b = addRows a a;
```

```
-- The following program is invalid as the addCols function throws a type error
Grid a = read "A.csv";

Int i = 2 * 100;

output = addCols i a;
```

```
-- The following program is invalid as the tirst line does not end with a semicolon
Grid a = read "A.csv"

output = if (numRows a >= 1) then (dropRow a 0) else (a);
```

7 Influences And Inspiration

The English-style function names and overall syntax of CSVQL are primarily inspired by Python-style pseudocode (https://www.python.org/). Several smaller syntactic elements also draw influence from Java (https://www.java.com/) and Haskell (https://www.haskell.org/). In particular, the static and strong type system used in CSVQL is heavily inspired by Java.