COMP2212 Report

Piers Jonckers (pj2g19), Harry Nelson (hjn2g19), Sofia Kisiala (zmk1g19)

May 2021

Main Language Features

CSVQL is a CSV reading and manipulating language, which can import, create and edit CSV files or new tables. To aid with the CSV manipulation, the language has support for basic integer, boolean, and string types, as well as lists and custom extensions of the list type to represent rows and tables. Basic functionality with these types has been implemented (addition and comparisons between integers, boolean comparisons and negating), and there is support for variables for later use and indexing to navigate lists without looping over them.
There are also *for* loops to iterate over lists/tables/rows, and *if*/*if-else* statements to aid with the construction of new tables. The language also features several built in utility functions:

- *table.add(row)*/*table.add(x, y, z)* to add new rows to the table
- *length(x)* to get the length of a specific object
- *empty(x)* for a boolean indicating if an object is empty
- *table(x)* to create a new table of arity x
- *import(x)* to import the file x as a CSV

Finally, there is a table in the variable named "out", which is automatically printed to *stdout* upon completion of the program if it has been populated. However, other tables can be shown at specific times using the *show()* function.
Executing the interpreter without any arguments will open a line by line interpreter which shows the parsed structure and returned value of any statements executed, with any variables declared/edited persisting between entries.

Syntax

Scoping

Much like with typical programming languages such as Java and Python, created variables declared within *for*/*if*/*if-else* blocks will not be available outside of the bracketed scope of that block, however variables that existed prior that are updated will keep any changes made while inside the block. Any variable declared outside of a loop will be usable in any subsequent statements. Variable names can be overwritten, and in cases where this happens, the most recently declared version of the variable will be the one used upon calling the variable. None of this is atypical of common programming languages, and will thus not be difficult to learn to work with for new users, in line with our goal of being accessible to programmers.

Lexical Rules

An emphasis was placed on the English-language readability of CSVQL, with *for* loops and *if-else* statements presenting similarly to other programming languages which utilise these such as Python or Java. All functions are lowercase and describe what they return. We also make use of standard punctuation for marking out blocks to be executed in *for*/*if*/*if-else* statements ({curly brackets like so}), alongside standard punctuation for separating statements (;) so that the structure of a program is easily understandable. The intended result is for skills in other common programming languages to be transferable

to CSVQL. Newlines between statements and indents inside *for/if/if-else* blocks are not required; they are, however, strongly encouraged for the purposes of readability.
Following the grammar rules in appendix A, a statement can either be an assignment, a for expression, an if expression, a function, an operation, or just a base value like an int or a variable. The return value of the expressions in an if/else statement must match, and the type of the second variable in a for loop expression must be of type iterable, such as a list, a table, or a row. There is no way to manually write out a single table or row, however you can create tables with import() or table(x), access any rows in those tables via indexing them (t[0] = row). Some types, such as tables, have functions like add() assigned to them so they can be written as t.add(x), instead of add(t, x), for readability and so it is obvious that the whole contents of the brackets will be added to the row. Functions can behave differently depending on the input, again with add as an example, there can be any number (equal to the arity of the table) of arguments of (almost) any type, and they will all be converted to a string, encapsulated into a single row and stored in the table, however if the only argument is a single object already of type Row then it can recognise that and add the row without encapsulation.

Additional Features

Syntactic Sugar

- Can combine rows using + as well as using it for its typical function of integer addition.
- No need to create a table and print it each time, as the output table prints to stdout, and the arity is automatically set to however wide the first row is.
- Some functions can take different types and react differently to them (e.g. *table.add(x, y, z)* creates a row containing x,y,z, or if x is a row then *table.add(x)* will just add the row to the table without wrapping it in another row).
- Can compare any data type with ==.
- Or/and can be used to avoid needing nested if/else statements.

Type Checking

We have implemented a thorough type checker, *CSVQLTypes.hs*, which cascades through the AST of each expression and recursively validates the types for correctness right down to each base case. An intuitive *getType* function allows us to take the expected return type of anything, and *typesValid* is the main function which travels through the AST and ensures that any expression which has a specific required type follows the expected pattern. Informative errors tell the user what kind of statement failed the validity check, as well as the part of it that did: for example, whether it was the conditional expression, consequent, alternate, or a combination of these in an IfElse statement.
The checker works in harmony with any user-declared variables through the use of a type map which lists any non-default names and their associated type. Should the lookup fail, an informative error message is displayed to the user with the name of the variable for which the lookup failed.

Error Messages

Any parsing errors found by alex/happy will be delivered with a specific line and column number. Once in the type checking stage, the program specifies what kind of statement has thrown an error and where in said statement it has an invalid type. As mentioned before, if there are any variables involved then it specifies the name of the variable which has caused an error as well, in order to assist the user with locating the error.
During runtime, if any errors occur (e.g. index out of bounds, wrong variable passed to function) then the program will halt execution and an error message with some details will be printed to stderr as well. As the type checker and evaluator are executed line by line, only the first error it comes across will be reported, as it cannot continue evaluating past that point. This facilitates step-by-step debugging, and means that no excess errors are thrown as a consequence of run-on issues stemming from an earlier mistake.

Commenting

Users can make comments within their code by putting "#" before a line. This line will then be disregarded by the compiler.

Execution Model

Upon executing a .cql file with the CSVQL interpreter, the interpreter first loads the in-built functions of the language and creates the "out" table that can be contributed to. It then tokenises and parses the .cql file to create a tree representing the program.
Next, it passes the program through a type checker to make sure no statements are being used incorrectly (including tracking variable/function return types). If any types are invalid upon resolution, the user is informed of the problem found by means of printing it to stderr. The parse tree evaluation is then skipped, and execution terminates here.
If the program passes the type checker, it starts evaluating the parse tree statement by statement. Statements can add to and use a variable environment that is passed on between each one. Each statement is separated by a ";" in the grammar. Before moving onto the next one, the statement is evaluated to its fullest, so in the example 2 in the appendix, x will be saved to the variable environment as "$x = 5$" as opposed to "$x = y + z$". If there are any errors in a specific statement, the program will output an error message with details about what went wrong to stderr and will terminate the program instantly. Every statement has an evaluated value, although the majority will be *null*.
If every statement is evaluated successfully, the contents of the *out* table (if any) will be printed to stdout and execution will finish. The final result of the evaluation (what type it reduces down to) is ignored as long as it is valid.

A Grammar Rules

```
program     ::=    <expr>; <program> | <expr>;

expr        ::=    <assignexpr> | <forexpr> | <ifexpr> | <function> |
                   <operation> | <baseexpr>

assignexpr  ::=    <var> = <expr>

forexpr     ::=    for <var> in <var> { <expr> }

ifexpr      ::=    if <expr> { <expr> } |
                   if <expr> { <expr> } else { <expr> }

function    ::=    <var> . <var> ( <expr> ) | <var> ( <expr> )

operation   ::=    <expr> == <expr> | <expr> + <expr> | not <expr> |
                   <expr> and <expr> | <expr> or <expr> |
                   <expr> [ <expr> ]

baseexpr    ::=    <int> | true | false | <string> | null |
                   <var> | <list>

int         ::=    [0-9]+

string      ::=    "any characters, the regex was quite long"

var         ::=    [A_Za_z] [A_Z a_z 0-9 _]*

list        ::=    [ <expr> ]
```

B Execution Example

```
y = 3;
z = 2;
x = y + z;
```