COMP2212 Coursework Report

Authors: Wasif Haque (mwh1g19), Timothy Stevens (ts2g19), Viktor A. Rozenko Voitenko (vr4n18)

Our language is called Coma. It is a functional, expression-based, non-Turing-complete programming language.

Language features

Function

The language's main priority is transforming CSV tables i.e. take a few input files and output a result. We've decided that the functional model of execution is the best fit. We are not worried about side-effects or infinite loops in this language and it is clearly expressed via its execution model and syntax.

REPL

Coma is expression-based: every Coma program is a big expression that can be gradually evaluated to produce a single result, similar to that of the Toy Language. This allowed us to develop a comprehensive, multi-line REPL where you can input an expression such as:

```
let b := 1 < 2 & 1 > 2 in if b “hello” “world”
```

and get an instant response. See the **Appendix** for some other examples.

Non-Turing-Complete

Coma is intentionally not Turing-complete. We didn't want to deal with the halting problem in a language that will only ever work with non-infinite CSV tables. Therefore, we did not provide any facilities for infinite loops or recursion.

Error messages

We haven't given Coma its own type checker. It is a small scripting DSL not intended to be used within huge industrial environments. To counter-balance that, we utilise the *fail fast* principle. If you made a mistake in your Coma program, it will fail with a comprehensive error (in most cases) or a simple Haskell error. We therefore intend for the user to use the REPL to retrace their steps and see what went wrong.

Coma is strongly dynamically typed. For example, expressions like:

```
if (1 < 2) 42 “magic”
```

are permitted. The `if` statement will return `42`, but if you change its condition to `(1 > 2)`, the expression evaluates to `“magic”`.

We have also implemented basic lexing and parsing error checking through using the AlexPosn data type in our lexer and parser. The user is informed about what line(s) in their program has a lexical/parse error.

Minimalistic

Coma does not complicate its users with any syntactic sugar. We only have `let` blocks, a few operators, lambda functions, function calls, lists, and bracketed expressions. We use `each` as our own alternative to Haskell's `map`. Most of the core functionality is provided by our standard library stored in the `Core.hs` file.

Standard library functions

```
read   :: String -> CsvTable

join   :: Table -> Table -> Table

get    :: List Integer -> Row -> Row

select :: List Integer -> Table -> Table

value  :: Integer -> Row -> Any

merge  :: Row -> Row -> Row

given  :: (Row -> Boolean) -> Table -> Table

each   :: (Row -> Row) -> Table -> Table

csv    :: Table -> String

if     :: Boolean -> Any -> Any -> Any

not    :: Boolean -> Boolean
```

The following aliases help understand the above functions better:

```
type alias CsvTable := List (List String)

type alias Table    := List (List Any)

type alias Row      := List Any
```

The above types are not checked at compile time and are placed here for reference only.

Syntax

The syntax for our language is as follows:
(NB: n *is any integer value and* x *is any string value)*

```
 Let ::= let x := Expr in Let | Expr

Expr ::= Expr & Bool | Expr '|' Bool | Bool

Bool ::= Bool = Call | Bool != Call | Bool < Call |

         Bool <= Call | Bool > Call | Bool >= Call |
                Bool ++ Call | Call
        Call ::= Call Literal | Literal
     Literal ::= ( Expr ) | [ List ] | \ x -> Let | n | x
        List ::= empty | Literal List
```

Much of the syntax is self-explanatory and similar to that of the Toy language.

The scope for values instantiated by `let` blocks is all of the code that follows the `let` block. You will notice that in many of our solutions, we have utilised `let` blocks to itself include a lot of the functions that help solve our problems.

`Expr` deals with AND (&) and OR (|) statements. `Bool` mostly consists of comparison operations. These were generally not required for any of the problems but we added them for some extra flexibility in our language. The append operator (`++`) however was useful for our solution to problem 5.

`Call` allows for function application. `Literal` allows for bracketed expressions, lambda expressions (`\`), and also lists (`[1 "hello"]`) which can either be empty or can contain `Literals`.

A `List` would model a row in the CSV file, hence we could use it for example with the append operation to add to the output. CSV tables are represented as `Lists` of `Lists`.

The language also allows for any commenting (using `--`) and white spaces to represent new lines.

Interpreter Execution Model

As mentioned earlier, Coma programs operate over expressions. To evaluate an expression, Coma relies on `Env` - an environment in which identifiers are mapped to their values. Naturally, this is updated as the interpreter reads the program.

The main evaluator function is

```
execWithEnv :: Env -> Coma -> IO Coma
```

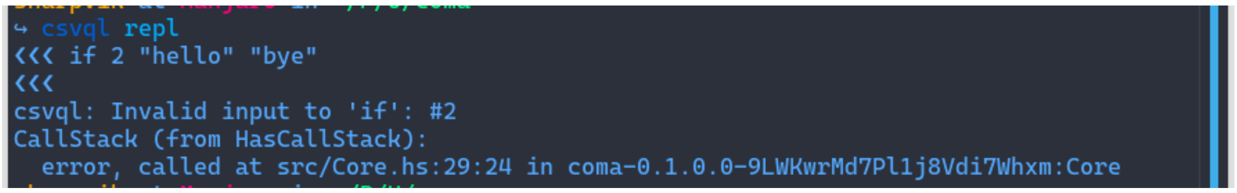
It has to return `IO Coma` since some functions in our language (namely the `read` function) require OS interactions.

The topmost call to `execWithEnv` is given the initial `Env` that contains all of the standard library functions. Then, expressions are evaluated following a very straightforward logic.

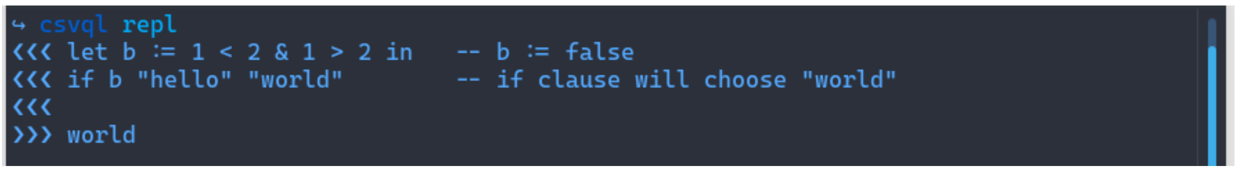
As the program progresses the `Env` will contain a greater map of strings (variable/function names) to comas. This is used in execution to lookup the meanings of references defined in the code. More are added to and removed from the `Env` as more references go in and out of scope.

Appendix

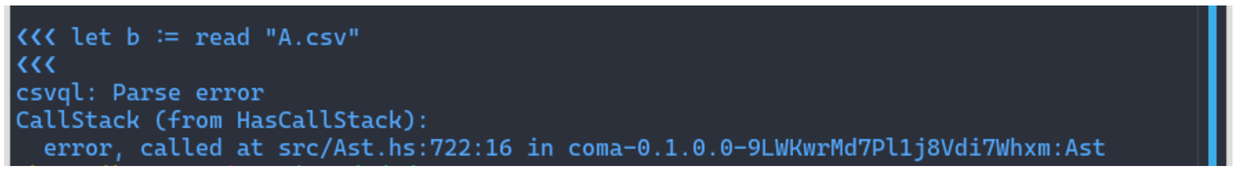
Here are some screenshots of the REPL in action:



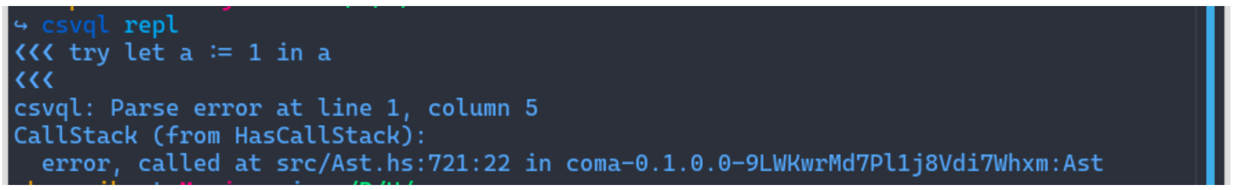
```
↪ csvql repl
<<< if 2 "hello" "bye"
<<<
csvql: Invalid input to 'if': #2
CallStack (from HasCallStack):
  error, called at src/Core.hs:29:24 in coma-0.1.0.0-9LWKwrMd7Pl1j8Vdi7Whxm:Core
```



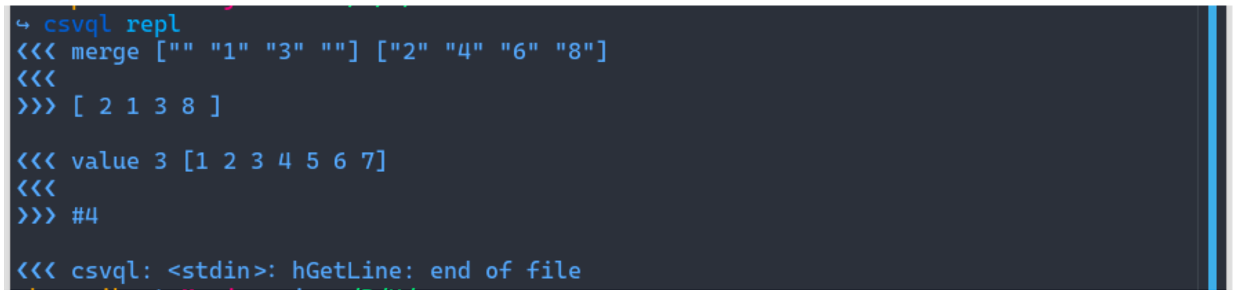
```
↪ csvql repl
<<< let b := 1 < 2 & 1 > 2 in    -- b := false
<<< if b "hello" "world"         -- if clause will choose "world"
<<<
>>> world
```



```
<<< let b := read "A.csv"
<<<
csvql: Parse error
CallStack (from HasCallStack):
  error, called at src/Ast.hs:722:16 in coma-0.1.0.0-9LWKwrMd7Pl1j8Vdi7Whxm:Ast
```



```
↪ csvql repl
<<< try let a := 1 in a
<<<
csvql: Parse error at line 1, column 5
CallStack (from HasCallStack):
  error, called at src/Ast.hs:721:22 in coma-0.1.0.0-9LWKwrMd7Pl1j8Vdi7Whxm:Ast
```



```
↪ csvql repl
<<< merge ["" "1" "3" ""] ["2" "4" "6" "8"]
<<<
>>> [ 2 1 3 8 ]

<<< value 3 [1 2 3 4 5 6 7]
<<<
>>> #4

<<< csvql: <stdin>: hGetLine: end of file
```