COMP2212 Coursework Report

Tom Humphrey, Lewis Hawkins, Tom Hoad

April-May 2021

Team

- Tom Humphrey (th1g19)
- Lewis Hawkins (lh5g19)
- Tom Hoad (tdh1g19)

1 Research on Query Languages

For our query language, we reflected on the methodologies of SQL as our main source of inspiration. SQL query statements rely incredibly heavily on single-line select statements providing many options of filtering and transforming database tables. SQL has a lot of expressive power but can be very complicated very quickly by long statements. Based on this research, we wanted to create a language where it was easier to read these complicated statements whilst retaining the querying abilities of SQL.

2 CFlat Query Language

Our language combines some methodologies from both declarative and iterative languages. The iterative nature relates to the general program structure, allowing the user to use multiple lines and assign variables. We took this approach as it was easier to keep track of the program, and made commenting more useful. Individual statements are then more declarative, where the user only has to provide simple commands for the interpreter to do most of the computing.

2.1 Syntax

Shown below is the Backus-Naur Form notation that was written at the beginning of the project that we based the final language upon.

⟨*Exp1*⟩ ::= load ⟨*word*⟩ ‘=’ ‘"’ ⟨*word*⟩ .csv ‘"’ ⟨*Exp1*⟩
| var ⟨*word*⟩ ‘=’ ⟨*Exp2*⟩ ⟨*Exp1*⟩
| preach word

⟨*Exp2*⟩ ::= select all of ⟨*word*⟩
| ⟨*word*⟩ add ⟨*word*⟩
| ⟨*word*⟩ add ‘(’ ⟨*Words*⟩ ‘)’
| select ‘(’ ⟨*Cols*⟩ ‘)’ of ⟨*word*⟩
| select all of ⟨*word*⟩ where ‘(’ ⟨*Wheres*⟩ ‘)’

| select '(' ⟨*Cols*⟩ ')' of ⟨*word*⟩ where '(' ⟨*Wheres*⟩ ')'
| select distinct '(' ⟨*Cols*⟩ ')' of ⟨*word*⟩
| select top ⟨*int*⟩ of ⟨*word*⟩
| unite ⟨*word*⟩ ⟨*word*⟩
| arrange ⟨*word*⟩ asc
| arrange ⟨*word*⟩ desc
| arrange ⟨*word*⟩ asc ⟨*int*⟩
| arrange ⟨*word*⟩ desc ⟨*int*⟩
| append ⟨*word*⟩ ⟨*word*⟩
| append ⟨*word*⟩ '"' ⟨*word*⟩ '"'
| append ⟨*word*⟩ '"' ⟨*int*⟩ '"'
| append ⟨*word*⟩ '"' ⟨*int*⟩ ⟨*word*⟩ '"'
| update ⟨*word*⟩ '(' ⟨*Sets*⟩ ')' where '(' ⟨*Wheres*⟩ ')'
| delete of ⟨*word*⟩ where '(' ⟨*Wheres*⟩ ')'

⟨*Words*⟩ ::= ⟨*word*⟩ ',' ⟨*Words*⟩
| ⟨*word*⟩

⟨*Sets*⟩ ::= ⟨*Set*⟩ ',' ⟨*Sets*⟩
| ⟨*Set*⟩

⟨*Set*⟩ ::= ⟨*int*⟩ '=' '"' ⟨*word*⟩ '"'

⟨*Cols*⟩ ::= ⟨*Col*⟩ ',' ⟨*Cols*⟩
| ⟨*Col*⟩

⟨*Col*⟩ ::= ⟨*int*⟩ nullCase ⟨*int*⟩
| ⟨*int*⟩

⟨*Wheres*⟩ ::= ⟨*Where*⟩ ',' ⟨*Wheres*⟩
| ⟨*Where*⟩

⟨*Where*⟩ ::= ⟨*int*⟩ '==' ⟨*int*⟩
| ⟨*int*⟩ '>=' ⟨*int*⟩
| ⟨*int*⟩ '<=' ⟨*int*⟩
| ⟨*int*⟩ '>' ⟨*int*⟩
| ⟨*int*⟩ '<' ⟨*int*⟩
| ⟨*int*⟩ '!=' ⟨*int*⟩
| ⟨*int*⟩ '==' ⟨*word*⟩
| ⟨*int*⟩ '==' notNull

2.2 Syntax Sugar

We used a small bit of syntactic sugar in our program. The first bit is in our select statements as the user can use the keyword all to select all of the columns rather than the user manually noting all of them. For example, the following statements achieve the same result for a 3 column table.

```
var B = Select all of A
var B = Select (1, 2, 3) of A
```

Our arrange statement also allows the user to sort on a specific column. The syntax also allows the user to not select a specific column and will automatically sort on the first. The following example will sort the table in the same order lexicographically.

```
var B = arrange A asc
var B = arrange A asc 1
```

2.3 Scoping

In our Haskell interpreter, we use an internal data structure to store the CSV files as lists of lists paired with the variable identifier. All variables are final so cannot be modified and hence, the rules of the program require each line to be a new variable declaration where it will store the

resulting table too. We believe this is better for the user as it allows them to test the output after every step in the program as a sort of debugging process. Every variable from statements and load assignment are public for all following statements.

2.4 Error Messages

To make debugging easier for the programmer, we first made sure that parse errors displayed the line number and column number of the offending code. This ensured that any errors could be identified far fast than without this feature. Our final error messages included the previous feature and also some custom error messages for run-time errors such as:

- Variable not found
- ArrayIndexOutOfBounds: column value
- ArrayIndexOutOfBounds: where value

Here, the out of bounds error for column values is when the index is too large in a select statement and the where value is when an index is too large in a where statement.

An additional error message was considered when an invalid file name was used to load a csv. As Haskell has its own 'openFile: does not exist' error message for this situation, it was decided that our language would not have an error and message in addition to the message that Haskell would output.

3 Execution Model

Our language's execution model consists of three stages: identifying tokens, parsing the tokens and performing the actions of the result of parsing. Our language consists of 33 tokens that relate to keywords and operators. When the regular expressions for keywords such as 'select' or 'load' are found in the input file, they are parsed to the next stage. Once the input file is converted into a list of tokens, the list is parsed to generate lexemes where the input is valid. These lexemes form an abstract syntax tree for the execution of the program which the interpreter then reads to perform the correct actions.