Aquery to Q Compiler: Parser Grammar

Jose Cambronero

January 21, 2015

1 Introduction

As part of implementing a compiler from Aquery to q[1], we have developed the BNF grammar below. This grammar is eventually used in implementing a flex/bison parser for Aquery.

2 Grammar

An Aquery program consists of a list of semi-colon separated queries with global context. Global queries in turn consist of a potential list of local queries followed by a global query.

⟨*program*⟩ ::= ⟨*global_queries*⟩

⟨*global_queries*⟩ ::= ⟨*global_query*⟩ ⟨*global_queries*⟩
| ϵ

⟨*global_query*⟩ ::= ⟨*local_queries*⟩ ⟨*query*⟩ ';'

We proceed to define what constitutes a local query, those that can solely be used within queries between the **WITH** keyword and the following global query. Note that aside from the necessary declarations at the beginning, the remainder of the query is a normal query, and thus refers to the grammar rule associated with the query non-terminal.

⟨*comma_identifier_list*⟩ ::= ⟨*identifier*⟩ ⟨*comma_identifier_list_tail*⟩

⟨*comma_identifier_list_tail*⟩ ::= ',' ⟨*identifier*⟩ ⟨*comma_identifier_list_tail*⟩
| ϵ

⟨*local_queries*⟩ ::= 'WITH' ⟨*local_query*⟩ ⟨*local_queries_tail*⟩
| ϵ

⟨*local_queries_tail*⟩ ::= ⟨*local_query*⟩ ⟨*local_queries_tail*⟩
| ϵ

[1]Aquery is an ordered database query language developed by Alberto Lerner and Dennis Shasha, for more information please see `https://cs.nyu.edu/web/Research/TechReports/TR2003-836/TR2003-836.pdf`

⟨*local_query*⟩ ::= ⟨*identifier*⟩ ⟨*col_aliases*⟩ 'AS' '(' ⟨*query*⟩ ')'

⟨*col_aliases*⟩ :: = '(' ⟨*comma_identifier_list*⟩ ')'
| ϵ

A query requires a select clause and a from clause, there are additional optional clauses including an ordering clause (the base of declarative order in Aquery), a where clause, and a group by clause

⟨*query*⟩ ::= ⟨*select_clause*⟩ ⟨*from_clause*⟩ ⟨*order_clause*⟩ ⟨*where_clause*⟩ ⟨*groupby_clause*⟩

We breakout the grammar for each relevant clause below

⟨*select_clause*⟩ ::= ⟨*select_elem*⟩ ⟨*select_clause_tail*⟩

⟨*select_elem*⟩ ::= ⟨*expression*⟩ 'as' ⟨*identifier*⟩
| ⟨*expression*⟩

⟨*select_clause_tail*⟩ ::= ',' ⟨*select_elem*⟩ ⟨*select_clause_tail*⟩
| ϵ

⟨*from_clause*⟩ ::= 'FROM' ⟨*table_expressions*⟩

⟨*order_clause*⟩ ::= 'ASSUMING' 'ORDER' ⟨*comma_identifier_list*⟩ (*comment: does the order clause allow expression on the fly, similarly to group by? *)
| ϵ

⟨*where_clause*⟩ ::= 'WHERE' ⟨*and_expression_list*⟩
| ϵ

⟨*groupby_clause*⟩ ::= 'GROUP' 'BY' ⟨*comma_expression_list*⟩
| ϵ

We now proceed to define what table expressions constitute. Table expression can be an identifier associated with a table, or an operation on a table (such as flatten).

⟨*table_expressions*⟩ ::= ⟨*table_expression*⟩ ⟨*table_expression_tail*⟩

⟨*table_expression_tail*⟩ ::= ',' ⟨*table_expression*⟩ ⟨*table_expression_tail*⟩
| ϵ

⟨*table_expression*⟩ ::= ⟨*table_exp*⟩ ⟨*identifier*⟩
| ⟨*table_exp*⟩

⟨*table_exp*⟩ ::= 'FLATTEN' '(' ⟨*identifier*⟩ ')'
| ⟨*identifier*⟩

We encode operator precedence and associativity into the grammar itself. This section of the grammar draws inspiration from `http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`.

⟨*literal*⟩ ::= ⟨*identifier*⟩ | '*' | ⟨*column_access*⟩ | ⟨*integer*⟩ | ⟨*float*⟩ | ⟨*date*⟩ | ⟨*string*⟩ | '(' ⟨*expression*⟩ ')'

⟨*column_access*⟩ ::= ⟨*identifier*⟩ '.' ⟨*identifier*⟩

⟨*call*⟩ ::= ⟨*literal*⟩
| ⟨*literal*⟩ '[' ⟨*indexing*⟩ ']'
| ⟨*built_in*⟩ '(' ')'
| ⟨*built_in*⟩ '(' ⟨*comma_expression_list*⟩ ')'

⟨*indexing*⟩ ::= ODD | EVEN | EVERY ⟨*integer*⟩

⟨*built_in*⟩ ::= 'abs' | 'avg' | 'count' | 'deltas' | 'distinct' | 'drop' | 'first' | 'last' | 'max' | 'maxs' | 'min' | 'mins' | 'mod' | 'next' | 'prev' | 'prd' | 'prds' | 'reverse' | 'sum' | 'sums' | 'stddev'

⟨*mult_expression*⟩ ::= ⟨*call*⟩
| ⟨*mult_expression*⟩ ⟨*times_op*⟩ ⟨*call*⟩
| ⟨*mult_expression*⟩ ⟨*div_op*⟩ ⟨*call*⟩

⟨*add_expression*⟩ ::= ⟨*mult_expression*⟩
| ⟨*add_expression*⟩ ⟨*plus_op*⟩ ⟨*mult_expression*⟩
| ⟨*add_expression*⟩ ⟨*minus_op*⟩ ⟨*mult_expression*⟩

⟨*rel_expression*⟩ ::= ⟨*add_expression*⟩
| ⟨*rel_expression*⟩ ⟨*l_op*⟩ ⟨*add_expression*⟩
| ⟨*rel_expression*⟩ ⟨*g_op*⟩ ⟨*add_expression*⟩
| ⟨*rel_expression*⟩ ⟨*le_op*⟩ ⟨*add_expression*⟩
| ⟨*rel_expression*⟩ ⟨*ge_op*⟩ ⟨*add_expression*⟩

⟨*eq_expression*⟩ ::= ⟨*rel_expression*⟩
| ⟨*eq_expression*⟩ ⟨*eq_op*⟩ ⟨*rel_expression*⟩
| ⟨*eq_expression*⟩ ⟨*neq_op*⟩ ⟨*rel_expression*⟩

⟨*and_expression*⟩ ::= ⟨*eq_expression*⟩
| ⟨*and_expression*⟩ ⟨*and_op*⟩ ⟨*eq_expression*⟩

⟨*or_expression*⟩ ::= ⟨*and_expression*⟩
| ⟨*or_expression*⟩ ::= ⟨*or_expression*⟩ ⟨*or_op*⟩ ⟨*and_expression*⟩

⟨*logical_expression*⟩ ::= ⟨*and_expression*⟩ | ⟨*or_expression*⟩

⟨*expression*⟩ ::= ⟨*logical_expression*⟩

Now that we have expressions defined, we define 2 forms of expression lists, comma and **AND** separated expression lists.

⟨*comma_expression_list*⟩ ::= ⟨*expression*⟩ ⟨*comma_expression_list_tail*⟩

⟨*comma_expression_list_tail*⟩ ::= ',' ⟨*expression*⟩ ⟨*comma_expression_list_tail*⟩
| ϵ

⟨*and_expression_list*⟩ ::= ⟨*expression*⟩ ⟨*and_expression_list_tail*⟩

⟨*and_expression_list_tail*⟩ ::= 'AND' ⟨*expression*⟩ ⟨*and_expression_list_tail*⟩
| ϵ

This concludes the formal outline of the Aquery grammar. Note that this grammar maybe revised and changed as necessary throughout development if need be.

For a flex/bison implementation of this grammar please see `https://www.github.com/josepablocam/aquery2q/parser/`