Grammar of JSON Queries

Table of Contents

Scott McKellar

Introduction

The format of this grammar approximates Extended Backus-Naur notation. However it is intended as input to human beings, not to parser generators such as Lex or Yacc. Do not expect formal rigor. Sometimes narrative text will explain things that are clumsy to express in formal notation. More often, the text will restate or summarize the formal productions.

Conventions:

1. The grammar is a series of productions.

2. A production consists of a name, followed by "::=", followed by a definition for the name. The name identifies a grammatical construct that can appear on the right side of another production.

3. Literals (including punctuation) are enclosed in 'single quotes', or in "double quotes" if case is not significant.

4. A single quotation mark within a literal is escaped with a preceding backslash: 'dog\'s tail'.

5. If a construct can be defined more than one way, then the alternatives may appear in separate productions; or, they may appear in the same production, separated by pipe symbols. The choice between these representations is of only cosmetic significance.

6. A construct enclosed within square brackets is optional.

7. A construct enclosed within curly braces may be repeated zero or more times.

8. JSON allows arbitrary white space between tokens. To avoid ugly clutter, this grammar ignores the optional white space.

9. In many cases a production defines a JSON object, i.e. a list of name-value pairs, separated by commas. Since the order of these name/value pairs is not significant, the grammar will not try to show all the possible sequences. In general it will present the required pairs first, if any, followed by any optional elements.

Since both EBNF and JSON use curly braces and square brackets, pay close attention to whether these characters are in single quotes. If they're in single quotes, they are literal elements of the JSON notation. Otherwise they are elements of the EBNF notation.

Primitives

We'll start by defining some primitives, to get them out of the way. They're mostly just what you would expect.

[1] string ::= "" chars ""

```
[2]          chars ::= any valid sequence of UTF-8 characters, with certain
                       special characters escaped according to JSON rules
[3]                ::= [ sign ] digit { digit }
    integer_literal
[4]           sign ::= '+' | '-'
[5]          digit ::= digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
[6]                ::= "'" integer_literal "'"
    integer_string
[7]        integer ::= integer_literal | integer_string
[8]         number ::= any valid character sequence that is numeric
                       according to JSON rules
```

When json_query requires an integral value, it will usually accept a quoted string and convert it to an integer by brute force – to zero if necessary. Likewise it may truncate a floating point number to an integral value. Scientific notation will be accepted but may not give the intended results.

```
[9]        boolean ::= 'true' | 'false' | string | number
```

The preferred way to encode a boolean is with the JSON reserved word true or false, in lower case without quotation marks. The string `true`, in upper, lower, or mixed case, is another way to encode true. Any other string evaluates to false.

As an accommodation to perl, numbers may be used as booleans. A numeric value of 1 means true, and any other numeric value means false.

Any other valid JSON value, such as an array, will be accepted as a boolean but interpreted as false.

The last couple of primitives aren't really very primitive, but we introduce them here for convenience:

```
[10]  class_name ::= string
```

A class_name is a special case of a string: the name of a class as defined by the IDL. The class may refer either to a database table or to a source_definition, which is a subquery.

```
[11]  field_name ::= string
```

A field_name is another special case of a string: the name of a non-virtual field as defined by the IDL. A field_name is also a column name for the table corresponding to the relevant class.

Query

The following production applies not only to the main query but also to most subqueries.

```
[12]        query ::= '{'
                      "'from'" ':' from_list
                      [ ',' "'select'" ':' select_list ]
                      [ ',' "'where'" ':' where_condition ]
                      [ ',' "'having'" ':' where_condition ]
                      [ ',' "'order_by'" ':' order_by_list ]
                      [ ',' "'limit'" ':' integer ]
                      [ ',' "'offset'" ':' integer ]
                      [ ',' "'distinct'" ':' boolean ]
                      [ ',' "'no_i18n'" ':' boolean ]
                      '}'
```

Except for the `"distinct"` and `no_i18n` entries, each name/value pair represents a major clause of the SELECT statement. The name/value pairs may appear in any order.

There is no name/value pair for the GROUP BY clause, because json_query generates it automatically according to information encoded elsewhere.

The `"distinct"` entry, if present and true, tells json_query that it may have to create a GROUP BY clause. If not present, it defaults to false.

The `"no_i18n"` entry, if present and true, tells json_query to suppress internationalization. If not present, it defaults to false. (Note that `"no_i18n"` contains the digit one, not the letter ell.)

The values for `limit` and `offset` provide the arguments of the LIMIT and OFFSET clauses, respectively, of the SQL statement. Each value should be non-negative, if present, or else the SQL won't work.