QL Language Reference Manual

Anshul Gupta (akg2155), Evan Tarrh (ert2123), Gary Lin (gml2153), Matt Piccolella (mjp2220), Mayank Mahajan (mm4399)

1.0 Introduction

JavaScript Object Notation (JSON) is an open-standard format that uses human-readable format to capture attribute-value pairs. JSON has gained prominence replacing XML encoded-data in browser-server communication, particularly with the explosion of RESTful APIs and AJAX requests that often make use of JSON.

While domain-specific languages like SQL and PostgreSQL work with relational databases, languages like AWK specialize in processing datatables, especially tab-separated files. We noticed a need for a language designed to interact with JSON data, to quickly search through JSON structures and run meaningful queries on the JSON data, all the while using a syntax that aligned much more closely with the actual structure of the data we were using.

2.0 Data Types

2.1 Primitive Types

All primitive data types are passed by value. They can each be declared and then initialized later (their value is null in the interim) or declared and initialized in-line.

2.1.1 Integers (`int`)

Integers are signed, 8-byte literals denoting a number as a sequence of digits e.g. `5`,`6`,`-1`,`0`.

2.1.2 Floating Point Numbers (`float`)

Floats are signed, 8-byte single-precision floating point numbers e.g. `-3.14`, `4e10`, `.1`, `2.`.

2.1.3 Boolean (`bool`)

Booleans are defined by the `true` and `false` keywords. Only boolean types can be used in logical expressions e.g. `true`, `false`.

2.1.4 String (`string`)

Since our language doesn't contain characters, strings are the only way of expressing zero or more characters in the language. Each string is enclosed by two quotation marks e.g. `"e"`, "Hello, world!".

2.2 Non-Primitive Types

All non-primitive data types are passed by a reference in memory. They can each be declared and initialized later (their value is null in the interim) or declared and initialized in line.

2.2.1 Arrays (`array`)

Arrays represent multiple instances of one of the primitive data types represente as contiguous memory. The square bracket notation is used to create an array and then get direct access to elements. Each array must contain only a single type of primitives; for example, we can have either an array of `int`, an array of `float`, an array of `bool`, and an array of `string`, but no combinations of these types. The size of the array is fixed at the time of its creation e.g. `array(10)`.

2.2.2 JSON (`json`)

Since the language must search and return results from JSON files, it supports Jsons as a non-primitive type. A `json` object can be created through multiple mechanisms. The first is directly from a filename of a valid JSON. For example, one could write: `json a = json("file1.json")`. This will check `file1.json` to ensure it is a valid JSON, and if so, will store the JSON in the variable `a`. The second way to obtain a JSON object is by using a subset of a current JSON. For example, say the following variable is already set:

```
b = {
    "size":10,
    "links": {
        "1": 1,
        "2": 2,
        "3": 3
    }
}
```

QL then allows for commands like `json links = b["links"]`. The links variable would then look as follows:

```
links = {
    "1" : 1,
    "2" : 2,
    "3" : 3
}
```

3.0 Lexical Conventions

3.1 Identifiers

Identifiers are combinations of letters and numbers. They must start with a lowercase letter, and can be any combination of lowercase letters, uppercase letters, and numbers. Lowercase letters and uppercase letters are seen as being distinct. We also reject dashes in identifiers. Identifiers can refer to three things in our language: variables, functions, and function arguments.

3.2 Keywords

The following words are defined as keywords and are reserved for the use of the language; thus, they cannot be used as identifiers to name either a variable, a function, or a function argument:

```
int, float, bool, string, json, array, where, in, as, for, while, return, funct
ion, true, false, if, elseif, else, void, not
```

3.3 Comments

We reserve the symbol `#~~` to introduce a comment and the symbol `~~#` to close a comment. Comments cannot be nested, and they do not occur within string literals. A comment looks as follows:

```
#~~ This is a comment. ~~#
```

3.4 Literals

Our language supports several different types of literals.

3.4.1 `int` literals

A string of numeric digits of arbitrary size that does not contain a decimal point with an optional '-' to indicate a negative number.

3.4.2 `float` literals

A string of numeric digits of arbitrary size, followed by a single '.' digit character, followed by another string of numeric digits of arbitrary size. It can also contain an optional '-' to indicate a negative number. In addition, we are following Brian Kernighan and Dennis Ritchie's explanation in *The C Programming Language*: "A floating constant consists of an integer part, a decimal part, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part, or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing."

3.4.3 `boolean` literals

Booleans can take on one of two values: `true` or `false`. `true` evaluates to an integer value of 1 and `false` evaluates to an integer value of 0. Thus, something like `true == 1` would evaluate to `true`, and something like `if(1)` would be valid.

3.4.4 `string` literals

A sequence of ASCII characters surrounded by double quotation marks on both sides.

4.0 Syntax

The following sections define the specifics of the syntax of our language.

4.1 Punctuation

QL employs several different types of punctuation to signal certain directions of workflow or special blocks of code within programs.

4.1.1 `()`: hierarchical evaluation, function arguments, `where` clauses

Parentheses can be used in three main cases:

- Numerical or Boolean statements: Forces the expression inside the parentheses to be evaluated before interacting with tokens outside of the parentheses. For example, in `1*(2-3)`, the expression `2-3` will be evaluated, and its result will then be multiplied with `1`. These can also be nested, e.g. : (1 + (4-(5/3)*2)).

- Function arguments: When providing arguments during a function call, the arguments must be listed within parentheses directly after the name of the function. For examples, foo(array a, int b) involves a function foo() that takes in an array and an integer enclosed in parentheses. The parentheses are also used for marking the argument list in the function definition, i.e.

```
function foo(array a, int b) : array {
    #~~ code goes here ~~#
}

foo(arr1, myInt)
```

- `Where` clauses: In a `where` clause, the search criteria must be enclosed within parentheses, and the expression within the parentheses should evaluate to a boolean value. For example,

```
where(["size"] > 10 & ["weight"] < 4) as item {
    #~~ code goes here ~~#
}
```

4.1.2 `{}`: function definitions, `where` clauses

Curly braces have two uses:

- Function definitions: When a function is defined, the procedural code to be run must be enclosed in curly braces.

- `where` clauses: In a `where` clause, immediately following the search criteria, curly braces enclose the code to be implemented. Using the `where` clause outlined above. The open and closed curly braces should contain all of the code to be run for each entry within the JSON that passes the filter.

4.1.3 : : function return types

The colon has use in our language as the specifier of a function return type. Separated between our language identifier and its argument list, we specify a `:` to mark that we will not be specifying a return type. Immediately after this colon, then, comes our function return type, which can be any of the data types we described above.

4.2 Operators (listed in order of precedence)

4.2.1 `[]` : attribute access

This can be used in two different ways:

- [int `index`]: accesses value at `index` of array
    - Return type is the same as the array's type.
- [string `key`]: accesses value at `key` of JSON
    - Return type is inferred from the value in JSON. The type can be one of three things: a value (int, float, bool, string), an array, or a JSON.

This operator can nest, e.g.: ["data"]["views"]["total"]. It associates from left to right.

Here is a program containing different examples of the `[]` operator and their return values based on the following JSON:

```
#~~ ["data"]["views"]["total"] returns an int. ~~#

#~~ We iterate through each "data" object with a total viewcount less than 100 ~~#

where (["data"]["views"]["total"] < 80) as item {
    #~~ item["data"]["users"] returns an array ~~#
    array users = item["data"]["users"]

    #~~ iterate through the array ~~#
    for (int i = 0; i < users.length; i++) {
        #~~ print the user at index i in the array ~~#
        print users[i]
    }

    #~~ item["data"]["items"]["category"] returns a string ~~#
    if (item["data"]["items"]["category"] == "News") {
        where (true) as name {
            print "name"
        } in users
    }
```

```
} in json("file1.json")


file1.json:

[{"data": {
    "views": {
        "total": 80
    },
    "items": {
        "category": "News"
    },
    "users": [
        "Matt",
        "Evan",
        "Gary"
    ]
},
{"data": {
    "views": {
        "total": 1000
    },
    "items": {
        "category": "Sports"
    }
}]
```

4.2.2 `%` : mod

- `int` % `int`: returns int (the remainder of ($1 divided by $3))

For all other combinations of types, we throw an error (incompatible data types).

4.2.3 `*` : multiplication

- `int` * `int`: returns int ($1 multiplied by $3)
- `float` *int*, *int* `float`, `float` * `float`: returns float ($1 multiplied by $3)

For all other combinations of types, we throw an error (incompatible data types).

4.2.4 `/` : division

- `int` / `int`: returns an int (the floor of ($1 divided by $3))
- `float` / `int`, `int` / `float`, `float` / `float`: returns float ($1 divided by $3)

For all other combinations of types, we throw an error (incompatible data types).

4.2.5 `+` : addition, concatenation

- `int` + `int`: returns int ($1 added to $3)

- `float` + `int`, `int` + `float`, `float` + `float`: returns float ($1 added to $3)
- `string` + `int`, `int` + `string`, `float` + `string`, `string` + `float`: returns string ($1 concatenated with $3)

For all other combinations of types, we throw an error (incompatible data types).

4.2.6 – : subtraction

- `int` – `int`: returns int ($1 minus $3)
- `float` – `int`, `int` – `float`, `float` – `float`: returns float ($1 minus $3)

For all other combinations of types, we throw an error (incompatible data types).

4.2.7 = : assignment

- `anytype` = `anytype`: sets value of $1 to $3.

If the type of $1 is different from the type of $3, we throw an error.

4.2.8 `not` : negation

- not `expr` = evaluates `expr` as a boolean (throws error if this is not possible); returns the opposite of `expr` (if `expr` was true, return false; if `expr` was false, return true)

If this operator is used on anything other than a bool, we throw an error.

4.2.9 Equivalency operators

- == : equivalence,
- != : non-equivalence,
- > : greater than,
- < : less than,
- >= : greater than or equal to,
- <= : less than or equal to

`anytype` OP `anytype`: returns a bool (true if $1 OP $3 e.g. `3 == 3` returns true)

- if $1 and $3 are strings, we do a lexical comparison
- if $1 and $3 are both ints, or both floats, we see if they are equal

If the types are anything other than these specified combinations, we throw an error.

4.2.10 Logical operators

- `expr1` & `expr2`: evaluates `expr1` and `expr2` as booleans (throws error if this is not possible), and returns true if they both evaluate to true; otherwise, returns false.
- `expr1` | `expr2`: evaluates `expr1` and `expr2` as booleans (throws error if this is not possible), and returns true if either evaluate to true; otherwise, returns false.

4.3 Statements

There are several different kinds of statements in QL, including both basic and compound statements. Basic statements can consist of three different types of expressions, including assignments, mathematical operations, and function calls. Statements are separated by the newline character `\n`, as follows:

```
expression \n
```

The effects of the expression are evaluated prior to the next expression being evaluated. The precedence of operators within the expression goes from highest to lowest. To determine which operator binds tighter than another, check the operator precedence above.

4.3.1 Declaration of Variables

To declare a variable, a data type must be specified followed by the variable name and an equals sign. After the equal sign, the user has to specify the datatype with the corresponding parameters to be passed into the constructor in parentheses.

```
<data_type> <variable_name> = <data_type>(<parameter>)
<parameter> = <identifier> | <literal>
```

Some examples of the declaration of variables would be:

```
array testArr = array(10)
int i = int(0)
float f = float(1.4e10)
bool b = bool(true)
string s = string("foo")
```

4.3.2 Function Calls

A function-call invokes a previously declared function by matching the unique function name and the list of arguments, as follows:

```
<function_identifier>(<arg1>,<arg2>,...)
```

This transfers the control of the program execution to the invoked function and waits for it to return before proceeding with computation. Some examples of possible function calls are:

```
sort(a)
array a = append(a, int(2))
```

4.3.3 Conditional Statements

Our conditional statements behave as conditional statements in other languages do. They check the truth of a condition, executing a list of statements if the boolean condition provided is true. Only the `if` statement is required. We can provide an arbitrary number of `elseif` statements following the `if`, though there can also be none. Finally, we can follow an `if`/combination of `elseif`'s with a single `else`, though there can be only one.

An example conditional statement is as follows:

```
if (__boolean condition__) {
    #~~ List of statements ~~#
}
elseif (__boolean condition__) {
    #~~ List of statements ~~#
} else {
    #~~ List of statements ~~#
}
```

4.3.4 Return statements

A return statement ends the definition of a function which has a non-void return type. If there is no return statement at the bottom of the function block, it is evidence that there is a `void` return type for the function; if it's not a `void` return type, then we return a compiler error.

4.3.5 Loop statements

4.3.5.1 `where` clauses

The where clause allows the user to search through a JSON and find all of the elements within that JSON that match a certain boolean condition. This condition can be related to the structure of the element; for example, the condition can impose a condition of the certain property or key of the element itself.

A where condition must start with the `where` keyword, followed by a boolean condition enclosed in parentheses. This condition will be checked against every element in the JSON. The next element is the `as __identifier__`, which allows the user to identify the element within the JSON that is currently being processed. This must be included. Following this is an `{`, which marks the beginning of the body code which is applied to each element. A closing `}` signifies the end of the body. The last section is the "in" keyword, which is followed by the JSON through which the clause will iterate to extract elements.

```
where (__boolean condition__) as __identifier__ {
    #~~ List of statements ~~#
} in __json__
```

4.3.5.2 `for` loops

The for loop starts with the `for` keyword, followed by a set of three expressions separated by commas and enclosed by parentheses. The first expression is the initialization, where temporary variables can be initialized. The second expression is the boolean condition; at each iteration through the loop, the boolean condition will be checked. The loop will execute as long as the boolean condition is satisfied, and will exit as soon as the condition is evaluated to false. The third expression is the afterthought, where variables can be updated at each stage of the loop. Following these three expressions is an open `{` , followed by a list of statements, and then a close `}`.

```
for (__initialization__, __boolean condition__, __update__) {
    #~~ List of statements ~~#
}
```

4.3.5.3 `while` loops

The while loop is initiated by the `while` keyword, followed by an open paren `(`, followed by a boolean expression, which is then followed by a close paren `)`. After this, there is a block of statements, enclosed by `{` and `}`, which are executed in succession until the condition inside the `while` parentheses is no longer satisfied. This behaves as `while` loops do in other languages.

```
while (__boolean condition__) {
    #~~ List of statements ~~#
}
```

5 Standard Library Functions

Standard library functions are included with the language for convenience for the user. The first few of these functions will give users the ability to perform basic modifying operations with arrays.

5.1 append

```
function append(array arr, int x) : array {

}
```

The above function takes in an array and an integer as arguments and returns an array with size increased by 1 that contains that integer at the last index.

5.2 `unique`

```
function unique(array arr) : array {

}
```

The above function receives an array as argument and returns a copy of the array with duplicate values removed. Only the first appearance of each element will be conserved, and a resulting array is returned.

5.3 `sort`

```
function sort(array arr) : array {

}
```

The above function receives an array as argument and returns a copy of the array with all of the elements sorted in ascending order. To compare the elements of the array, the > operator is used. For example, the array `[1,4,3,5,2]` passed into the sort() method would return `[1,2,3,4,5]`. The array `["c","e","a","c","f"]` would return `["a","c","d","e","f"]`.

5.4 `print`

We also include a built-in print function to print strings and primitive types.

```
print(toPrint)
```

Multiple primitives may be printed to console in one statement, concatenated by a `+`:

```
print(toPrint1 + toPrint2)
```

Attempting to print something that is not a primitive will result in an error.