LOON

THE LANGUAGE OF OBJECT NOTATION

Language Reference Manual

Kyle Hughes, Jack Ricci,
Chelci Houston-Borroughs, Niles Christensen, Habin Lee

October 2017

Contents

1 Introduction

Over the past decade, JavaScript Object Notation (JSON) has arguably become the format of choice for transferring data between web applications and services. With the rise of AJAX-powered sites, developers everywhere are using JSON to pass updates between client and server quickly & asynchronously. JSON has achieved its immense popularity in large part due to its flexibility and lightweight nature, but also due to its independence from any particular language. An application programmed in Java can send JSON data to a client running on a JavaScript engine, who can pass it along to a different application coded in C#, and so on and so forth. However, programs written in these languages generally utilize libraries to convert JSON data to native objects in order to manipulate the data. When the program outputs the modified JSON data, it must convert it back to JSON format from the created native objects.

LOON, the Language of Object Notation, provides a simple and efficient way to construct and manipulate JSON data for such transfers. Developers will be able to import large data sets and craft them into JSON without needing to import standard libraries or perform tedious string conversions. In addition to generating JSON format from other data formats, programmers will be able to employ LOON to operate on JSON data while maintaining valid JSON format during all iterations of the programming cycle. In other words, LOON eliminates the JSON-to-Native Object-to-JSON conversion process. This feature provides valuable debugging capabilities to developers, who will be able to log their output at any given point of their code and see if the JSON is being formatted properly. LOON is simple by nature. It resembles C-based languages in its support for standard data types such as int, float, char, boolean, and string. Arrays in LOON are dynamic and can hold any type. The language introduces two new types: Pair and JSON. These two types will be at the heart of most every piece of LOON source code. Where LOON truly separates itself is in its provision of operators, such as the "+=" operator for concatenation, for usage on values of the JSON and Pair types. This allows for more intuitive code to be written when working with JSON data.

2 Types

2.1 JSON

An object of type json is formatted according to the official JSON standard. That is, any object of type json is the concatenation of:

1. An open brace character, **{**
2. Any number of pair objects, with the comma character spliced in between each instance of two consecutive pairs
3. A closed brace character, **}**

Contents nested inside of the json object can be accessed through two methods:

1. The key-value access notation described in section 3f.
2. Loop iteration as described in section 6.

Objects of type json are initialized by two methods:

1. An open brace followed by a closed brace after the = operator. This is the default style of initializing a json object.
2. Entering any valid JSON object (defined above) after the = operator. If an invalid JSON object is assigned as the initial value of an identifier of type json, the compiler will throw an error.

2.2 Pair

The pair type represents a key-value pair. The key will always be a String (as described below). The value can be an object of any type described in this language, except for an object of type pair. In totality, a valid pair object consists of the concatentation of the following:

1. A quotation mark, **"**
2. The concatenation of any number of valid characters
3. A quotation mark, **"**
4. At least one white space character
5. A colon
6. At least one white space character
7. An object of any valid type, excluding pair

If two pairs are concatenated using the + operator, the resulting object is of type json.
Pairs are declared using carat notation, where a valid LOON type name must be spliced between the carats. An example is:

```
1 pair<int> intPair
```

Pairs are defined and initialized using carat notation as well. Pair initialization occurs when the following is entered to the right of the = operator:

1. At least one whitespace character
2. An open carat character, <
3. Either the empty string or a valid pair object as defined above
4. A closed carat character, >

The compiler will throw an error on the following initialization miscues:

1. Passing in an empty string or a string containing invalid characters as part of the key
2. Type mismatch between the pair's declared value and the initialized value.
3. Attempting to assign the concatenation of two pair objects to an identifier of type pair, as the concatenation of two pair objects is of type json.

The value of a pair's key can be retrieved using the notation described in section 3f. The value of a pair's key can be modified by specifying the key using the key-value access notation of section 3f and entering the new value to the right of the assignment operator, =.

2.3 Int

A 32-bit two's complement integer. Standard mathematical operations will be implemented.

2.4 Float

A 32-bit float. Standard mathematical operations will be implemented.

2.5 Char

An 8 bit integer. Can be used as an integer, but should typically be understood to represent an ASCII character.

2.6 Boolean

A 1-byte object that can have two values: True and False. Can use standard boolean operators.

2.7 Array

An array of any of the other types of values, including Array itself. JSON format allows for type flexibility within a single array, so LOON offers developers the ability to craft both statically and dynamically typed arrays. Creating an array containing arbitrary types will be done with a special notation.

```
1 // Create a new array
2 Array<Int> myIntArray = [5, 4, 3, 2, 1]
3
4 // Append the integer 4 to the end of the array
5 myIntArray += 4
6
```

```
// Create a two-dimensional array with characters
Array<Array<Char>> twoDimensionalCharArray = [['a', 'b'], ['c', 'd']]

Array<?> miscArray = [5, 'FOO', 3.14]
```

2.8 String

Strings are an immutable, array sequence of characters. To modify an existing string, it is necessary to create a new one that results from some use of legal string operations. The code snippet below details the declaration and manipulation of strings in LOON:

```
// Create a string
String newLang = 'LOON'
```

In its declaration, memory is allocated precisely according to the string's size. The string is able to handle any further addition of characters that go beyond its initial size. The goal in this is for developer's to be able to load data into and out of strings seamlessly.

3 Lexical Conventions

3.1 Identifiers

LOON identifier refers to the name given to entities such as variables, functions, and objects. They give unique name to an entity to identify it during the execution of the program. You can choose any name for an identifier outside of the keywords.
For example:

```
int count = 0\\
String myString = 'hello'\\
json result
```

Here *count*, *myString* and *result* are identifers.

3.2 Keywords

The following are reserved words in LOON and cannot be used as identifiers to define variables or functions: *if, elseif, else, for, while, return, break, continue, int, float, char, boolean, string, array, json, pair.* LOON also reserves the keywords *abstract, FILL*; a short description of each of their use-cases follows:

- arbitrary
  - Use to declare an array that accepts any type.
    `Array<arbitrary> a = []`
  - Arrays of standard types (int, double, char, etc.) can be incorporated within arbitrary arrays, but not vice versa (arbitrary arrays cannot be incorporated within typed arrays).

3.3 Literals

LOON literals refer to fixed values that are immutable during program execution. They can be of any of the primitive data types such as *integer, float, string, and boolean.*

1. Integer Literal
   An integer literal is a sequence of one or more integers from 0-9.
2. Float Literal
   A float literal has an integer part, decimal point, fractional part and exponential part.
3. String Literal
   String literals are sequences of characters enclosed in single quotes.
4. Boolean Literal
   Boolean literals are either *true* or *false*. If the user assigns a different value an error will be raised.

3.4 Comments

LOON allows for multiline/nested comments, as well as single-line comments. The table below summarizes the convention for both comment formats:

Comment Symbol	Description	Example
/* */	Multiline comments	/* This /* is legally */ commented */
//	Single-line comment	// This is a legal comment in LOON

3.5 Whitespace

The newline is significant in the LOON language; otherwise, whitespace is discarded.

3.6 Functions

3.6.1 Function Definitions

The general format used to define a function in LOON is as follows:

```
return_type function_name(parameter list ) {
   body of the function
}
```

Here are all the parts of a function in LOON -

1. Return Type
   The return type is the data type of the value the function returns. If the function does not return a value, users should use return type *void*.

2. Name
   This is the identifier for the function. The name paired with the parameters is the function signature.

3. Parameters
   Parameters act as placeholders in LOON. When a function is invoked, the user passes a value into the parameter. The parameter list refers to the type, order and number of parameters of a function. A function may also contain no parameters.

4. Body
   The body of a function contains a collection of statements that logically define what a function does.

3.6.2 Function Declaration

Functions in LOON are called by their identifiers. To call a function, pass the required parameters with the function name. If the function returns a value, you can store it in a variable.

3.7 Operators/Punctuation

Operator	Usage	Function
+	x + y (binary operator)	Depends on types added. When adding two Floats, or two Integers, or a Float and an Integer, it returns the sum of the two values. If both numbers added are Integers, then the expression evaluates to type Integer. Otherwise the expression evaluates to type Float. If both x and y are Arrays, then the result is a new Array that is y concatenated to the end of x. If x is an Array and y is an object of the same type as x stores, then the expression evaluates to a new array with all the values in x, and the value of y appended to its end. If either x or y is a String and the other is a character, then this concatenates the second value to the first and represents a String representation. If both x and y are Characters, then the expression evaluates to a String consisting of x first, and then y second. If both x and y are of type JSON, then it evaluates to a new object of type JSON that contains all the keys in both JSON objects. If adding 2 Pair objects, the expression evaluates to a JSON object containing both Pairs. If adding a Pair object to a JSON object, then the result is a JSON object containing all the Pairs from the JSON object as well as the Pair being added. If both x and y are Strings, then this is the concatenation operator.
-	x - y (binary operator)	Depends on types used on. If subtracting an Integer from an Integer, then this evaluates to an Integer representing the difference between the two values. If subtracting an Integer from a Float, a Float from an Integer, or a Float from a Float, then the expression evaluates to type Float, but is still the difference between the two values.
	- x (unary operator)	Valid when x is either an Integer or a Float. Returns the value of x * -1. Evaluated expression is of same type as x.

*	x * y (binary operator)	Depends on types used on. If multiplying an Integer by an Integer, then this evaluates to an Integer representing the product of the two values. If multiplying an integer by a Float, a Float by an Integer, or a Float by a Float, then the expression evaluates to type Float, but is still the product between the two values.
/	x / y (binary operator)	Depends on types used on. If dividing an integer by an integer, then this evaluates to an integer representing the result of dividing x by y, with the remainder discarded. If dividing an integer by a float, a float by an integer, or a float by a float, then the expression evaluates to type Float, and is x divided by y as a decimal as well as can be approximated.
%	x % y (binary operator)	Modulo operator. Both x and y must be Integers. Evaluates to x modulo y.
[]	x[y]	Used to access values. There are a few possible valid combinations of types for x and y. If x is an Array and y is an Integer, this evaluates to the value that is stored at location y in x. Type will vary depending on what the Array stores. If x is a String and y is an Integer, this evaluates to the Character in location y of x. If x is a JSON object, and y is a String, then this expression evaluates to the value of the object in x with key y.
==	x == y	Evaluates to True if x is equal to y, and False otherwise.
!=	x != y	Evaluates to True if x is not equal to y, and False otherwise.
!	!x (unary operator)	Valid when x is a boolean. Evaluates to False if x is True and True if x is False.
>	x >y	Evaluates to True if x is greater than y. Valid for an combination of Integer, Float, and Char.
>=	x >= y	Evaluates to True if x is greater than or equal to y. Valid for an combination of Integer, Float, and Char.

<	x <y	Evaluates to True if x is less than y. Valid for an combination of Integer, Float, and Char.
<=	x <= y	Evaluates to True if x is less than or equal to y. Valid for an combination of Integer, Float, and Char.
&&	x && y	Logical and. Evaluates to True if both x and y are True, and False otherwise.
||	x || y	Logical or. Evaluates to True if either x or y are True, and False otherwise.

Note: Any of the above binary operators have a level of syntactic sugar that can be added by appending an equals sign to the operator, with the exception of the comparison operators. For example, += or *=. In this case, a b= c for an arbitrary binary operator b should be understood as a = a b c. Note that this will not always lead to a valid expression.

4 Syntax

4.1 Program Structure

In LOON, there is no main function. Rather, any code that is not contained within the scope of a function is executed by default (and this code can call defined functions). Furthermore, statements and expressions are distinguished by newlines. The following code block highlights the structural flow of a standard .LOON program (note that it is not doing anything useful):

```
/* Begin Example Code */

// Import functions XML_Manipulation.LOON
from XML_Manipulation import textify_XML, http_XML_toJSON
String xfilePath = '"/.../XMLData/test.xml"'
String jfilePath = '"/.../JSONData/test.JSON"'

JSON XMLobj_toJSON(String filePath)
{
    String buffer
    JSON newObject

    String xmlFile = readFile(filePath, buffer)

    String removeXMLTags = textify_XML(xmlFile)

    JSON newObject = string_to_JSON(removeXMLTags)
    return newObject

}

JSON firstJSON = {"name": "Just Jack"}
Pair agePair = <"age", 20>
JSON secondJSON = firstJSON + agePair
// secondJSON == {"name": "Just Jack", "age": 20}

// Output JSON to user
printJSON(secondJSON)
// Write JSON object to request file
writeJson(jfilePath, secondJSON)
```

4.2 Expressions

4.2.1 Declaration

Declaration of variables are achieved as follows. Type specification is mandatory for Array and pair.

```
JSON myJSON
Pair<int> intPair
String myName
Array<Int> myIntArray
```

4.2.2 Assignment

Assignment : Assignment can be done using where lvalue is the variable and rvalue is the value.

```
myIntArray = [1, 2, 3, 4, 5]
myIntArray += 4
```

4.2.3 Precedence

All operators follow the standard precedence rules. Every operation, apart from assignment (right-to-left associative), is left-to-right associative.

4.3 Statements

4.3.1 Expression Statements

Statements in within a line (not escaped) are treated as one statement

4.4 Loops

Loop Type	Usage	Function
While	while(x) {y}	If x is false, nothing happens. If x is true, then the block of code y is executed. Once y is finished, the loop is evaluated again. If x is now false, then the loop ends (breaks). However, if it is still true, then the entire process is repeated. The loop can carry on potentially infinitely if x never becomes false.
For	for(a | b | c) {y}	Upon encountering this loop, the command a is evaluated. A is only evaluated the first time through. Then, b is evaluated. If b is true, then the block of code y is evaluated. Once y is finished, c is evaluated. Then, b is evaluated again. If b is still true, then y is executed again. This process of y $\rightarrow$ c $\rightarrow$ b is repeated for as long as b is true when evaluated.
For-each	for(a in b){y}	b must be an object of type Array or JSON. If not, an exception is raised. If b is empty, then nothing happens. Otherwise, a is assigned the value of either the first object in the array, or one of the pairs in the JSON object. y is then executed. Then, a is assigned either the second value in b if b is an Array, or another Pair if b is a JSON object. Then, y is evaluated. This is repeated until has taken on all the values in b, and y has been executed for each case. Note that because the JSON object makes no guarantee about ordering, there can be no guarantee for the order in which a takes on values when b is a JSON object. However, if b is an array, a is guaranteed to take on values in order.

4.5 Scope

1. Single file scope
   Identifiers declared within a LOON file are either declared external to any function or inside of a function. Identifiers declared external to all functions within the file are accessible for all functions defined within the file. The identifiers persist from beginning to end of program runtime. Identifiers delcared within any function block or conditional block are accessible only within their block of declaration. They persist from the moment that they are declared to the moment that their block of declaration is no longer being executed. The only exception to this standard is that variables declared inside of a loop conditional definition persist until the loop's conclusion.

2. Multiple file scope
   Multiple file scope concerns the relationship between identifiers external to any function block in multiple files. An external identifer in one file is not accessible for methods in other files. As a result, there is no conflict between two external identifiers of the same name in two different files. The compiler will recognize that these are independent identifiers and treat them accordingly. Functions are utilized to pass data to and from external identifiers in different files. Functions native to one file are accessible to functions in another file by including the first file at the beginning of the second as defined in section 4a.

4.6 Casts

LOON allows for a limited implementation of casts. A cast may be performed for objects of type float to objects of type int and for objects of type int to objects of type float. Casts are written as the concatenation of an open parenthesis, the type name, and a closed parenthesis. The written cast is placed after the assignment operator, =, but before the object value to be converted, with whitespace separating the three entities. An example of a valid cast would be:

```
1  int intVal = 10
2  float floatVal = (float) intVal //Has value of 10.0
```

JSON manipulations tend to be short on the need to cast objects of certain types to other types, which is why LOON is so limited in this facet. LOON offers a standard library of functions, convert.loon, that allow objects of one type (i.e. json or pair) to be converted to a format that is valid for another type (i.e. string).

5 Input/Output

LOON contains the ability to read from and write to sources and destinations of text data. The ability to perform read and write operations is offered through the language's standard library. Input/output functions are held in the io.loon library file. The following functions are defined:

1. readFile(String filePath, String buffer)
   Reads the specified text file into a string. The string must already have been declared in the program prior to the readFile function call.

2. readLine(String filePath, String buffer)
   Reads the next line from the specified text file into a previously declared string. Once invoked for the first time in a program on a particular file, the readLine function maintains a notion

of position within the specified file. Subsequent calls to readLine will return the next line of text from the file and update the position in the file that is pointed to.

3. readJson(String filePath, JSON buffer)
   Reads the specified text file into a previously declared JSON object. The string must already have been declared in the program prior to the readFile function call. The readJson function is for reading in JSON files or other files that the user can ensure are in proper JSON format. The compiler will evaluate the validity of the JSON read in from file just as it would check the validity of a JSON object initialized by the user in the source code. An error will be thrown on a readJson attempt on a file that does not consist of a valid JSON sequence of text.

4. writeFile(String filePath, String buffer)
   Writes the text contents of the string to the specified file. Once invoked for the first time in a program on a particular file, the writeFile function maintains a notion of position within the specified file. Subsequent calls to writeFile will append the text contents of the string to the end of the file.

5. writeJson(String filePath, JSON buffer)
   Writes the JSON object to the specified file, maintaining proper JSON format.