Contents

1 Introduction

JSON is a standard used to primarily transmit structured data between a server and a web application in a human-readable format. JSON parsers have been built for nearly every language and allow the data to be translated to a data structure. The data format has become incredibly popular and is used by many of the world's most popular websites for access to their data, both externally and internally. Unfortunately though, there does not exist a language specifically focused on quickly accessing and manipulating structured JSON objects. Many other languages require creating complex class definitions, or abstractions to facilitate JSON use.

As JSON has become the predominant format for transmitting data structures and arrays on many major APIs, the usefulness of a simple domain specific language drove the development of this project. A common workflow for API data interaction involves a series of steps and procedures sometimes spanning several scripting languages. As a first step, there is an initial call to the API in which JSON formatted objects are returned. Then the JSON objects string format is parsed into several objects. Finally, various manipulations and aggregations are performed depending on the task at hand.

JL (JSON Language) facilitates programming at each of the above steps, with particular emphasis on enabling navigation and item specification within a JSON objects structure. While maintaining the original JSON objects hierarchy, the language allows the hashmap structure inherit in JSON objects to be easily accessed and manipulated through simple statements or functions.

2 Lexical Conventions

2.1 Line Terminators

Expressions are terminated with a semi-colon,";"

2.2 Comments

Comments begin with the characters /* and end with the characters */ and may span multiple lines. Comments do not nest inside one another.

2.3 Whitespace

Spaces, tabs, line terminators, and comments are all considered whitespace, and are ignored, except for tokenization.

2.4 Identifiers

An identifier is defined as sequence of letter and numbers of any length, starting with a letter.

2.5 Data Types

Data types can either be atomic or compound. Types are dynamically assigned during runtime, but combinations between types will be cast automatically, and throw a casting error if impermissible.

2.5.1 Atomic Data Types

The atomic data types are:

int, boolean, float, string

2.5.2 Compound Data Types

The compound data types are:

list, hashmap

2.6 Constants

2.6.1 Numeric Int

A numeric int constant is defined as one or more digits. It can be optionally preceded by a negative sign to delineate a negative integer.

2.6.2 Numeric Floats

A numeric float constant is defined as one or more digits, followed by a decimal point, followed by one or more digits.

2.6.3 String Literals

A string literal constant is defined as anything inside of a pair of double quotation marks.

2.6.4 Booleans

Booleans are delineated by either a $true$ or $false$ token.

2.6.5 Lists

Lists are styled after JSON lists and can be constructed using: [$value1$,$value2$, ...]. Values can be any of the built in data types.

2.6.6 Hashmaps

Hashmaps are styled after JSON object syntax and can be constructed using: {*key*1:*value*1, *key*2:*value*2,...}. Keys types are restricted to strings or ints. Values can be any of the built in data types.

2.7 Keywords

The following identifiers are reserved as keywords, and may not be used:

as	break	continue	do
else	elseif	for	fun
gets	if	puts	return
true	false	to	then

2.8 Operators

2.8.1 Mathematical

The following mathematical operators are allowed:

+, -, *, /

2.8.2 Boolean

The following boolean operators are allowed:

>, <, >=, <=, ==, !

2.8.3 Logical

The following logical operators are allowed:

&&, ||

2.9 Separators

The following characters are separators:

{ } [ ] ( ) : :: ; ,

2.10 Scope Rules

A variable is defined when it is first used, and its lifetime ends out of the block in which it is defined.

2.11 Selectors

JL uses selector syntax inspired by XPath to permit quick access to to the value associated with a given key in objects or arrays. Selectors are delineated using [ ] at the end of objects/lists.

2.11.1 Root level

The simplest selector is identified by *object*[:*key*]. This selector is an alias of *object*[:*key*1] which will look only in the root level of the the object return the value associated with the provided *key*.

2.11.2 Any position

Using a double colon *object*[::*key*] descends to the bottom of an object finding all values with the listed identifier, returning a list of matching values.

2.11.3 Chaining

Selectors can be chained together like *object*[:*key*1:*key*2]. This example finds values associated with *key*2 in an object that is associated with *key*1.

2.11.4 Arrays

Arrays are treated as objects keyed by positive monotonically increasing integers and are accessed in the same way as objects: *array*[:int]. Arrays can also be chained if they are nested, *array*[:int:int]. They can also be accessed when chaining selectors with objects, *object*[:*key*:int].

2.11.5 Parents

If the value returned by a key is an object or a list, its containing compound constant is returned. Select parents by including < at the end of the selector like *object*[:*key* >] (otherwise nothing is returned)

2.11.6 Children

If the value returned by a key is an object or a list, the values of its children are returned. Select children by including > at the end of the selector like *object*[:*key* >]

2.11.7 Specific Depths

Specific depth targets can be chosen using $object$[:$key\{depth\_start/depth\_end_{opt}\}$]. In this case, the selector will return the first value associated with *key*. "::" can be used to return all matching values.

2.11.8 Restrictions

Restrictions to selectors are listed within the selector braces and delineated by commas. $object$[:$key, restriction, ...$]. Whenever a selector finds a matching *key* in the object or array, it will execute each of the provided restrictions and only include the discovered value if all of the restrictions evaluate to *true*. Restrictions can therefore be any valid boolean expression. Restrictions are executed in the scope of the discovered key. Therefore, selectors can be used in restrictions.

$object$[::*key*,:key2== "*Foo*"] selects keys matching *key*, so long as *key*2 is a sibling that has the value "Foo".

2.12 Function Definitions

Function are defined as:

fun *identifier* (*parameter-list*) *statement*

where

parameter-list :

ϵ
| *identifier*
| *identifier, parameter-list*

3 Syntax

3.1 Expressions

The following describes the expressions, in decreasing order of priority.

3.1.1 Constant

An int, boolean, float or a string constant is an expression. The value of the expressions is the constant.

3.1.2 Variable

A variable is an expression. The value of the expression is the value of the variable.

$Variable \rightarrow expression$

3.1.3 (expression)

An expression enclosed in parentheses is an expression. Parentheses can be used to indicate precedence.

3.1.4 logical-expression

A logical expression is one or more boolean expression concatenated by && (AND) or ||(OR). A boolean expression is of the form expression OP expression, where OP can be >, <, >=, <=, ==, or of the form ! expression.

BooleanExpression → *expression LogicalOp expression*

3.1.5 selector-expression

The selector expression can be applied to hashmaps or lists to access elements within those items.

SelectorExpression → ((: | ::)keyword|int) + (< | >)?(, *BooleanExpression*)∗

3.1.6 compound[selector-expression]

The compound must be a hashmap or list, and the selector-expression must be a valid selector expression. An error will be thrown if the enclosed expression is not valid.

3.1.7 identifier(expression-list-opt)

The identifier must be the name of a function. A functional call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function.

3.1.8 expression * expression

The * indicates multiplication. Both expression must be of type float. The result has type float.

3.1.9 expression / expression

The / indicates division. Both expression must be of type float. The result has type float.

3.1.10 expression + expression

The + indicates addition. Both expression must be of type float. The result has type float.

3.1.11 expression - expression

The - indicates subtraction. Both expression must be of type float. The result has type float.

3.1.12 lvalue

An lvalue is a variable or expression[:expression] where the first expression is a list and the second expression is of type float, or expression[:key] where expression is a hashmap.

$lvalue \rightarrow Variable \mid expression[SelectorExpression]$

3.2 Statements

3.2.1 Expression Statement

The most simple *statement* is created by adding a semicolon to the end of an expression.

statement → *expression*; |{*statement-list*}

Usually expression statements are assignments or function calls.

3.2.2 Compound Statement

Several statements can be used as a compound statement where one is expected. A compound statement takes the form: *statement-list*

where

statement-list:

statement
| *statement statement-list*

3.2.3 Conditional Statement

The two forms of conditional statements are:

if (*expression*) then *statement*

if (*expression*) then *statement* else *statement*

The else ambiguity is resolved by connecting an else with the last encountered else less if.

3.2.4 For Statement

The for statement has the form:

for *expression*1 to *expression*2 do *statement*

3.2.5 Return Statement

A function returns to its caller by means of the return statement, which has the form:

return *expression*;

A function may have multiple return statements. They do not need to return the same type.

3.2.6 Break Statement

The statement:

break;

terminates the current loop; control passes to the statement following the terminated statement.

3.2.7 Continue Statement

The statement:

continue;

add 1 to the expression1 and test if it is less than expression2, and if it is, run the body of the loop.

4 Examples

4.1 Convert an array to a keyed map

This is a function which takes a json object consisting of a list of objects containing name keys, and creates a new object which makes the names keys to access the objects. It assumes that names are unique.

```
fun makeNameKey(jsonObject) {
  objectsWithName = jsonObject[::name<];
  result = {};
  for i=1 to objectsWithName.length() do {
    result = result # {objectsWithName[:i:name]:objectsWithName};
  }
  return result;
}
```

input

```
[
  {
    name: "John",
    from: "New York"
   },
  {
    name:"Bob",
    from:"California"
   }
]
```

output

```
{
  "John":{
    name: "John",
    from: "New York"
  },
  "Sue": {
    name:"Bob",
    from:"California"
  }
}
```

4.2 Combine differently structured json

In this example, one data set is stored by seller, and another is stored by product. Our language allows the quick comparison of the two concepts with the following program, which returns a list of the pricing data.

```
fun comparePricingData() {
   pricingData1 = gets(dataSet1)[::price];
   pricingData2 = gets(dataSet2)[::price, :name==``Gizmo"];
   return pricingData1 @ pricingData 2;
}
```

dataSet1

```
[
  {
   "product_id": 1232131,
   "product_name": "Gizmo",
    "sellers": [
      {
        "name": "Discount Shop",
        "price": 1.5
      },
      {
        "name": "Luxury Good",
        "price": 2.75
      }
    ]
   }
]
```

dataSet2

```
[
  {"seller_id": 4324,
    "name": "Tech Deals",
    "products": [
      {
        "name": "Gizmo",
        "price": 1.4
      },
      {
        "name": "Widget",
        "price": 1.75
      }
    ]
  }
]
```

output

```
[1.5, 2.75, 1.4]
```