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. Our interpreted language hopes to change that by allowing easy access to specific objects, and allowing calculations to be done on specific objects within the single structure via command chaining. In this sense, it is a spiritual successor to *awk*, which tried to achieve similar goals for plain text tables at the time of its creation.

2 Language Goals

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.

This 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.

In addition to the low level structural elements of the language, several command line utilities for aggregations and manipulations will be built into the language. These utilities will allow a user to chain together several transformations and manipulations. As an example, a single line of code could be used to call Twitters API, sort the JSON data based on a first field within the object, and then calculate the median value for a second field as grouped by the first field. In effect, the process of accessing and utilizing JSON data will be made more efficient. Taking inspiration from AWKs simple and elegant functionality, the language will enable flexible access to JSON objects and will allow a programmer to quickly achieve required aggregations.

3 Example Program

In a hypothetical research project our developer seeks to gather pricing data from a number of different pricing websites all providing data through JSON. These JSON formats are all represented differently, and this would normally require that the developer handle each api separately, depending on the implementation language.

From one API:

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

From a second API:

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

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.

```
def comparePricingData()
   pricingData1 = :price{2}(gets(readurl("http://firstapi"))
      )
   pricingData2 = :price{2}[name=="Gizmo"](gets(readurl("
      http://secondapi")))
   return pricingData1 @ pricingData 2
end
```

In this example ":price" is a keyword

4 Syntax

4.1 Basic types:

map object, int, float, string, keyword(symbol in LISP or ruby)
A keyword starts with a colon, e.g., :price Example: a json object called priceData

```
priceData = {
    price : {
      apples : 1.50
     }
}
```

A keyword is automatically a special type of function :keyword(jsonObject). So *:price:apples(priceData)* returns the price of apples in this case

Keywords can also have depth and adjacent child restrictions placed on them. *:price{2}* is a function which finds only values of price at depth two. *:price[name="Gizmo"]* finds only the price of objects which have the name "Gizmo".
Keywords will also likely have other modifiers which help restrict which types to access and also return additional information (ie. parents, adjacent elements, etc..)

4.2 Mathematical operations:

+, -, *, /
Mathematical operations work with keywords
:price(a) + :price(b) if *:price* is a keyword of object a and b
:price(a) is also a valid lvalue, so *:price(a) = :price(b) + 2*. If *:price(b)* returns multiple price values, then only the first will be used for assignation. The result, however, will be assigned to all referenced objects returned by *:price(a)*.

4.3 Logical operators:

<, >, <=, >=, ==
Logical operators will do pairwise comparisons of all values returned by a keyword. So

in order for *:price(a)* < *:price(b)* to return true, all of the values returned by *price(a)* must also be less than all of the values returned by *price(b)*

4.4 Examples of valid expressions

a json object, a function call that returns a json object, a json object + a json object, a json object - a json object, a json object * a json object, a json object / a json object

4.5 Control flows

if boolean *then* expr1 *elseif* expr2 *else* expr3

4.6 Function Definition

```
def function ( argument)
        return 1
end
```

def return and end are keywords in the language. function can be recursive.

4.7 Built-in Functions

gets
An operation that takes a string representation of a json object, returns a internal representation of the object, with meta type deduced from the input or provided by the user

```
jsonObj = gets( [] )
jsonObj = gets(readurl( http://... ))
jsonObj = gets( [] , meta)
gets(readurl( ... )).add(gets(readurl( ... )))
```

puts

```
str = puts(jsonObj)
```