COMS W4115: Programming Languages and Translators
Language Project Proposal
LOON: Language of Object Notation

Kyle Hughes (keh2184), Niles Christensen (nnc2117), Chelci Houston-Burroughs (ceh2193),
Jack Ricci (jr3663), Habin Lee (hl3020)

September 26, 2017

1 Language Overview

Over the past decade, JavaScript Object Notation (JSON) has arguably become the format of choice for transferring data between web applications and services. 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.

Our proposed language, LOON (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 Potential Applications

LOON has the capacity to serve a variety of use cases, especially as JSON's popularity grows in comparison to rival data transportation formats. LOON's potential applications and programs are listed below, with a brief explanation of the problem's background and where LOON comes into play.

1. Format Control
    - Converting some form of dictionary/array/hash, etc. to JSON text.
2. Parsing
    - Converting JSON text to some form of dictionary/array/hash, etc.
3. Streamlined JSON data processing
    - Easy and intuitive manipulation of data; no longer need JSON-to-Native Object-to-JSON conversion(s) that you would need in a Java, C etc. program.
4. Data Generation
    - Generation of useful test data through LOON-based algorithms.
5. Ease of conversion between alternate data formats
    - LOON algorithms to efficiently convert JSON objects to XML, etc. (and vice versa).
6. Ease of Data Comparison
    - With LOON, developers can operate on valid JSON at all points in their programming cycle; keys & values between different JSON objects can be easily manipulated, operated on, and compared.
7. Ease of JSON data exchange (send, receive) to/from large data sets
    - LOON enables developers to easily integrate external JSON data to existing data sets and, in the other direction, pull interesting JSON data from external data sets.

3 Technical Details

3.1 JSON

This is the most important type in the language. The main advantage that this language has over other languages that have JSON libraries will be a very optimized representation of a JSON object. This will be stored in memory as an array containing a series of references to Pairs (per the description in the next subsection). This will allow efficient iteration over the Pairs in the JSON object, in addition to the ability to fetch, for example, the value corresponding to any key through a hash-map setup. Values can be accessed through the formal *jsonName.keyName*, as demonstrated below. Any variable of type JSON has a value that is always in valid JSON format. This is of great value to developers when debugging at different stages of their program.

```
// Initialize an empty JSON object
JSON testJsonString = {}

// Initialize a JSON object with values
JSON initializedJson = {'myIntegerValue': 5}

// Changes myIntegerValue to 6
initializedJson.myIntegerValue ++
```

3.2 Pair

Pair represents a key-value pair. The key will always be a String (as described below). The value can be any type described in this language, except for a Pair. It might be possible to include syntactic sugar such that if an attempt is made to make the value of a Pair be another Pair, the second Pair could be wrapped in its own JSON object. This would cause reasonable behavior in most cases, as it will then be its own dictionary with only one value.
Similarly, if two Pairs are concatenated, the resulting object should be a JSON object, as it is the most reasonable way to store two Pairs together. Since this language is strongly typed, carat notation can be used to indicate the type of the value in the Pair.

```
// Creates a new Pair with an Int as the value
Pair<Int> intPair = <'Test value': 5>
// Creates a new Pair
Pair<String> stringPair = <'String value': 'Hello!'>

// Should throw an error
Pair<Int> errorPair = <'Other string value': 'Goodbye!'>

//Pair with value of type JSON
Pair<JSON> jsonPair = <'jsonValue', {name: Jack, age: 20}>
/*jsonPair's value is now:
{"jsonValue": {
                    "name": "Jack",
                "age" : 20
            }
```

```
}*/


// Combining Pairs
JSON combination = intPair + stringPair
//The value of combination is now {'Test value': 5, 'String value': 'Hello!'}

// Adding Pairs to JSON objects
combination = combination + <'New Value!': 7>
```

3.3 Int

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

3.4 Float

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

3.5 Char

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

3.6 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.

```
// Create a new array
Array<Int> myIntArray = [5, 4, 3, 2, 1]

// Append the integer 4 to the end of the array
myIntArray += 4

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

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

3.7 String

An array of Chars.

3.8 Functions

LOON will support functions to allow developers to modularize the tasks that they perform on similar types data sets. Functions will be employed using the familiar C-style function syntax, though the flexible nature of JSON data may lead to a looser enforcement of that style.

4 Source Code Example

Consider the following scenario: a developer has a set of structured data available in a CSV file called intro.csv. intro.csv contains information about a set of dog lovers, their ages, what kind of dog they own, and whether or not it is friendly. The contents of intro.csv appear below:

```
Jack, 22, German Shepard, true
Habin, 24, Golden Retriever, NULL
Kyle, 15, NULL, NULL
Niles, 23, Terrier, false
Chelci, NULL, Golden Doodle, true
```

The developer knows maintaining the data in this format is wasteful due to all the null values, so he or she decides that it should be converted to JSON format and stored in a document called intro.JSON. The developer writes the following program using LOON to convert the data to JSON and create intro.JSON:

intro.LOON:

```
String readLine(String fileString)
Array breakLine(char delimChar, String textLine)
String readFile(String FileName)

int main(){
    //Read CSV file into a string
    String myCSV = readFile(myCSV.csv)

    //Set up your custom field names
    Array fieldNames = ["name", "age", "dog"]
    Array dogInfo = ["type", "isFriendly"]


    String csvLine = ""

    //Make a list of people who own dogs
    JSON result = {};
    result += <"dogPeople", []>

    //For each line, delimit by commas and manipulate data accordingly
    while(csvLine = readLine(myCSV)){
        JSON individualResult = {}
        Array values = breakLine(',', csvLine)
```

```
        //For each value, match it to the appropriate field name
        for(int i = 0, k = 0; i < size(values) && k < size(fieldNames), i++, k++){
            if(fieldNames[k] == "dogs"){

                //Initialize dogs field to be of type JSON
                individualResult += <fieldNames[k], {}>

                //Knowing that next two fields contain info about
                //person's dog, we can iterate through and add them
                //to dog object
                for(int j = 0; j < size(dogInfo) && i < size(values); j++, i++){
                    if(values[i]){
                        individualResult.dogs += <dogInfo[j], values[i]>
                    }
                }
            }else{
                //Concatenate to individual's object
                if(values[i]){
                    individualResult += <fieldNames[k], values[i]>
                }
            }
        }

        //Add individual's data to dogPeople array
        result.dogPeople += individualResult
    }

    printJSON(result, intro.JSON)
}
```

After running intro.LOON on the intro.csv input file, the developer has created **intro.JSON:**

```
{
  "dogPeople" : [{
    "name" : "Jack",
    "age" : 22,
    "dog" : {
      "type" : "German Shepard",
      "isFriendly" : true
    }
  }, {
    "name" : "Habin",
    "age" : 25,
    "dog" : {
      "type" : "Golden Retriever"
    }
  }, {
    "name" : "Kyle",
```

```
    "age" : 15,
    "dog" : {}
  }, {
    "name" : "Niles",
    "age" : 23,
    "dog" : {
      "type" : "Terrier",
      "isFriendly" : false
    }
  }, {
    "name" : "Chelci",
    "dog" : {
      "type" : "German Shepard",
      "isFriendly" : true
    }
  }]
}
```