**Abu Abukar** **CSE 530 Term Project** **7/27/2022**

## Project Statement

For the term project I have chosen to use Coco/R EBNF to define the syntax of JSON (JavaScript Object Notation), the open standard file format and data interchange format that is used commonly in web applications. In my implementation I chose to use Coco/R for C# as it was what I have been using the most this course when working with Coco/R through samples and the homeworks.

## Language design and syntax definition in EBNF

In designing language and syntax definition in EBNF I followed the rules of valid JSON structure. JSON is a collection of key value pairs that are surrounded by curly braces. Each key value pair has a colon character in between them and all keys must be strings. However the values have more options on what it could be, such options include: string, numbers, arrays, booleans, null, or even another nested json object. Also an edge case that needs to be considered is a JSON file that only consists of a value and nothing else, this is technically correct syntax according to JSON.

### Syntax Definition:

**DATA** → **JSON** .

**JSON** → ‘ { ’ [**Object**] ‘ } ’ | **Value** .

**Object** → string ‘ : ’ **Value** { ‘ , ’ **Object** } .

**Value** → ( number | string | bool | null | **Arr** | ‘ { ’ [**Object**] ‘ } ’ ) .

**Arr** → ‘ [ ‘ [ **Value** { ‘ , ‘ **Value** } ] ‘ ] ’ .

Each type of value that is defined must follow certain rules that will be specified in the character section of the ATG file. For example, numbers can't have leading zeros unless it is the number zero or begins with a positive symbol, positive numbers are indicated without a leading symbol. This differs compared to some of the numbers that were defined in the calculator implementations we have seen in the course. However exponents can be defined with an e character. Also another interesting thing to observe is that Arrays can also have mixed data types unlike languages such as C or Java.

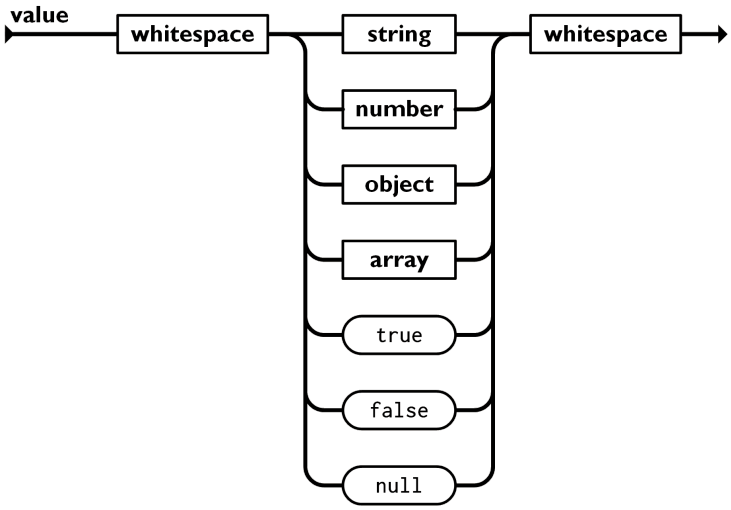


**Figure 1.1:** Value Definition From JSON.org

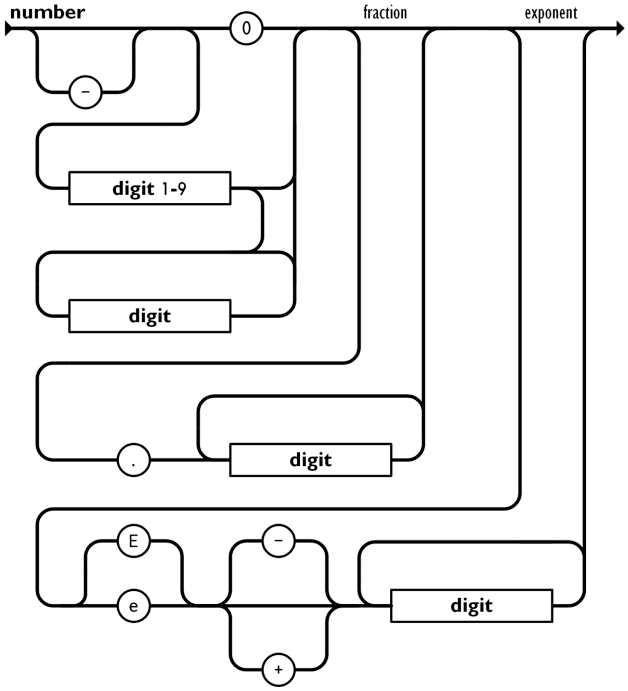


**Figure 1.2:** Number Definition From JSON.org

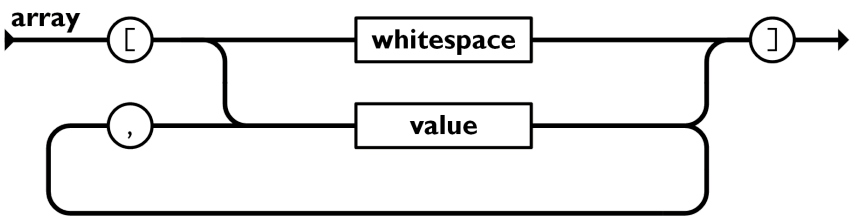


**Figure 1.3:** Array Definition From JSON.org

## Implementation

To implement this language in Coco/R in C#. A visual studio project was created with the Coco executable. parser.frame , and scanner.frame files. Other files were used as well to generate the abstract syntax tree such as AST.cs and the bin folder to generate the GraphViz tree with a dot file.

In the ATG file a character section was defined that would be used to tokenize the input that defined the json structure as well as to ignore whitespace. The token section defined the four major types, numbers, strings, bool, and null. The number token is similar to the one in the calculator samples but does not allow for leading zeroes or leading positive symbols. Strings are any characters encapsulated with double quotes, which also includes double quotes. However in JSON the only way to have nested double quotes in a string is to use escape characters. It would be difficult to tokenize strings without the escape characters as it would mistake a nested double quote as the final double quote. White space is also ignored as it is in JSON.

```
10  CHARACTERS
11
12      UpperLetter = 'A'..'Z'.
13      LowerLetter = 'a'..'z'.
14      letter = UpperLetter + LowerLetter.
15      nonzero = "123456789" .
16      zero = '0'.
17      cr     = '\r' .
18      lf     = '\n' .
19      tab    = '\t' .
20      space    = ' ' .
21      dquote = '"'.
22      escape = '\"'.
23      backslash = '\\'.
24      special = ':' + space + '!' + '@' + '#' + '$' + '%' + '^' + '&' +
25          '*' + '(' + ')' + '-' + '+' + '[' + ']' + '~' + '`' + '_' +
26          '?' + '<' + '>' + ',' + '.' + '|' + ':' + '\\' + '/' + '=' +
27          ';' + '\'' + '{' + '}'.
28      a = ANY - dquote.
29
30
31  TOKENS
32
33    number = ['-'] (zero | (nonzero {(nonzero | zero)}) )
34             ['.' (zero | nonzero) {(nonzero | zero)} ]
35             [('E'|'e')['+'|'-'] (nonzero | zero) {(nonzero | zero)}]
36           .
37    string = dquote {a | backslash '\"'} dquote.
38    bool = "true" | "false" .
39    null = "null".
40
41
42  IGNORE cr + tab + lf
```

**Figure 2:** Characters, Tokens, and Ignore sections of the ATG file

Next was implementing the productions for producing the parser specifications. Beginning with the Data production would produce the Json production, following that begins the three ways of describing a Json object: a value on it's own, an empty curly braces, or one or more key value pair objects encapsulated in curly braces.The root ast parent is passed into Json and AST parents and terminals are created in the Object production.

```
44  PRODUCTIONS
45
46  DATA                                (. root = new AST(null,"DATA");
47                                      TerminalAST terminal = null;.)
48  =
49  JSON<root>
50  .
51
52  JSON <AST parent>                   (. AST nt = new AST(parent, "JSON");
53                                      TerminalAST terminal = null; .)
54   =
55  (
56      '{'                             (. terminal = new TerminalAST(nt, "", t.val); .)
57      [Object<nt>]
58      '}'                             (. terminal = new TerminalAST(nt, "", t.val); .)
59  |
60      Value<nt>
61  )
62
63  End <>
64   .
```

**Figure 3:** First two productions for DATA and JSON

In the Object production a Object AST parent and produces a string, colon, and Value production. As well as the iterative metasymbol portion to add more objects which are separated with commas.

```
66  Object <AST parent>                 (. AST nt = new AST(parent, "Object");
67                                      TerminalAST terminal = null;.)
68   =
69
70      string                          (. StringAST str = new StringAST(nt);
71                                      str.SetValue (t.val); .)
72
73      ":"                             (. terminal = new TerminalAST(nt, "", t.val); .)
74
75      Value<nt>
76  {
77
78          ','                         (. terminal = new TerminalAST(nt, "", ","); .)
79              Object<nt>
80
81
82  }
83   .
```

**Figure 4:** Production of Object for key value pairs

Following that is the value production which is the value of the key value pairs. The value can be either a number, string, boolean, null, array, or nested json object. Depending on which case the respective AST child would be created for the visualization.

```
85  Value <AST parent>                    (. AST nt = new AST(parent, "Value");
86                                        TerminalAST terminal = null;.)
87   =
88  (
89       number                           (. NumberAST number = new NumberAST(nt);
90                                        number.Value = t.val;              .)
91  |
92      string                            (. StringAST str = new StringAST(nt);
93                                        str.SetValue (t.val); .)
94  |
95       bool                             (. BoolAST b = new BoolAST(nt);
96                                        b.Value = (t.val); .)
97  |
98       null                             (. terminal = new TerminalAST(nt, "", "null"); .)
99  |
100      Arr<nt>
101 |
102     '{'                               (. terminal = new TerminalAST(nt, "", t.val); .)
103     [Object<nt>]
104     '}'                               (. terminal = new TerminalAST(nt, "", t.val); .)
105 )
106
107  .
```

**Figure 5:** Value production which can be one of multiple types including a nested object

The array production is similar to the object production but is nested within brackets and zero or more value productions separated with commas.

```
109   Arr<AST parent>                     (. AST nt = new AST(parent, "Arr");
110                                       TerminalAST terminal = null;.)
111   =
112      '['                              (. terminal = new TerminalAST(nt, "", t.val); .)
113
114   [
115
116      Value<nt>
117
118      {
119
120          ','                          (. terminal = new TerminalAST(nt, "", ","); .)
121
122          Value<nt>
123
124      }
125
126   ]
127
128      ']'                              (. terminal = new TerminalAST(nt, "", t.val); .)
129    .
130
131
132   End<> =
133   .
134
135   END DATA.
```

**Figure 6:** Ending of the productions including the array production

The imputed data can be given through a JSON file or typed in a text field. This was implemented with an OpenFileDialog, TextFields, Buttons, in a Windows Form. The user can click the browse button to find a json file to parse or type in the text field and click validate input. Optionally the AST display can be turned on or off, changing a boolean.

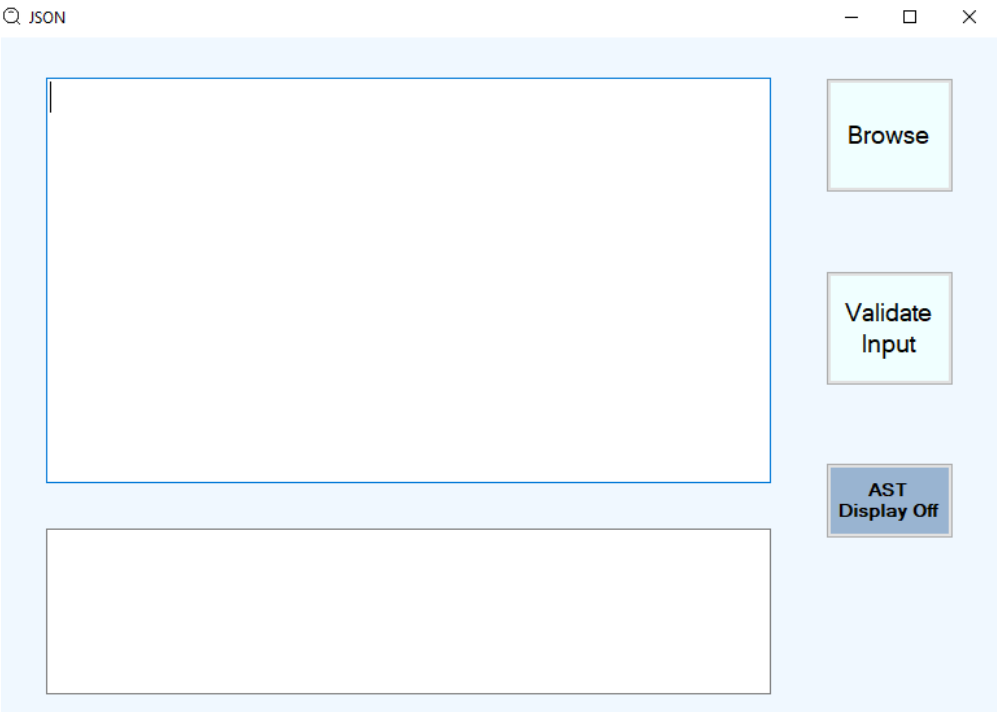


**Figure 7:** Windows Form to validate Json files or Json input

## Testing

To test this project JSON files were found online and allowed to find different edge cases that were not considered in initial versions. Multiple testing files are included in the Json folder of this project. These can be used to validate and display the condensed Abstract Syntax Tree

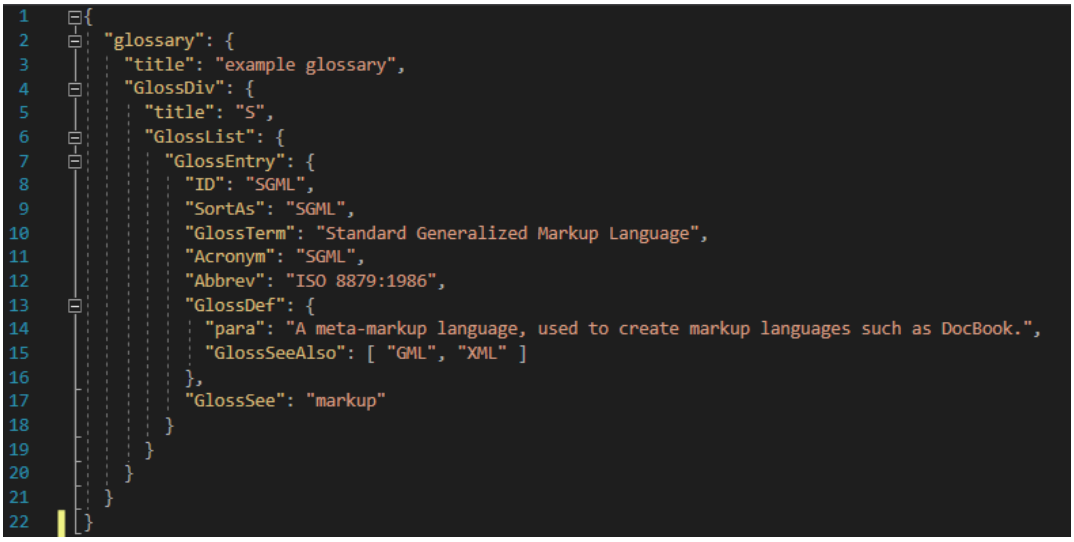


**Figure 8:** Glossary.json


**Figure 9:** Validating the input

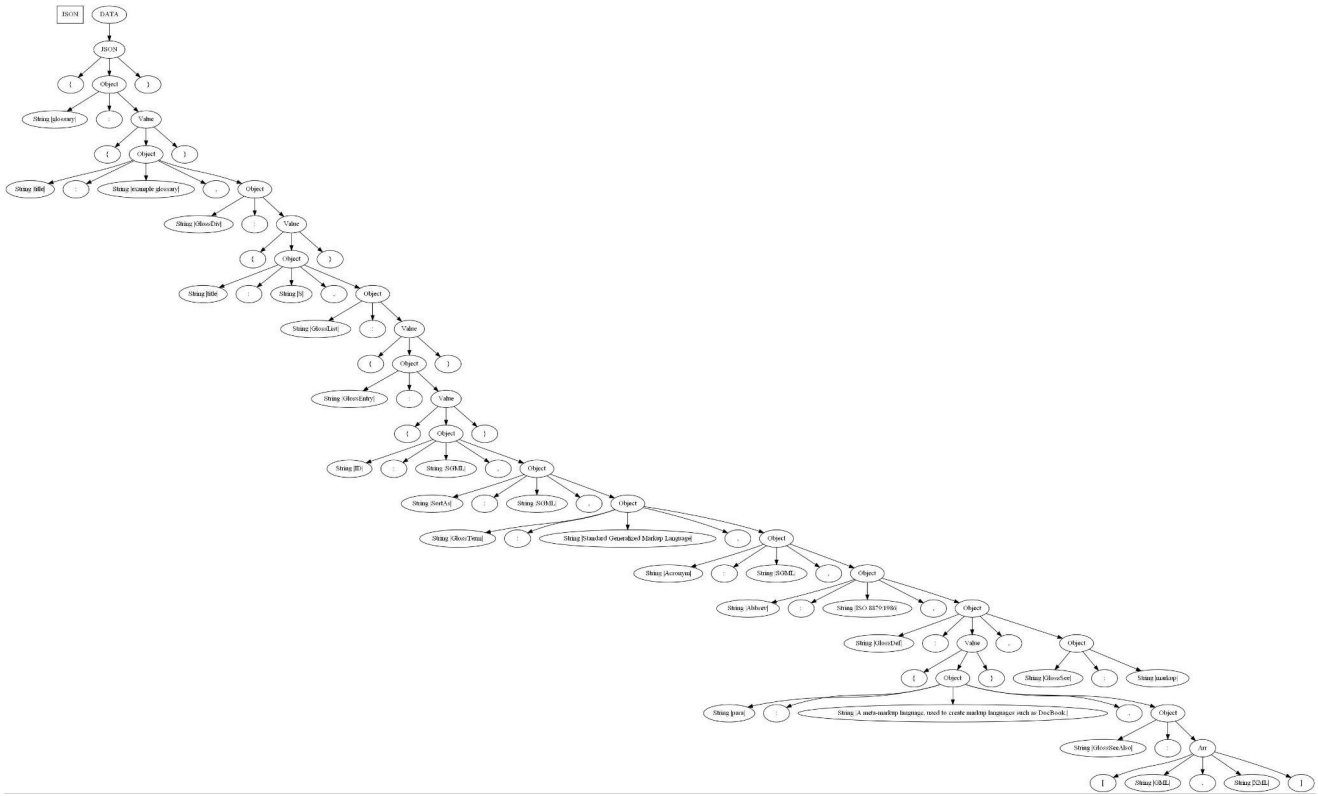


**Figure 10: Abstract Syntax Tree of the json file**

Incorrect test cases were examined as well such as strings with nested double quotes that do not have the escape character such as “”string”” instead of “\”string\””. A controlled testing website for validating json was used in combination with the developed application to make sure decisions were being made properly. This JSON validator can be found at https://jsonlint.com/

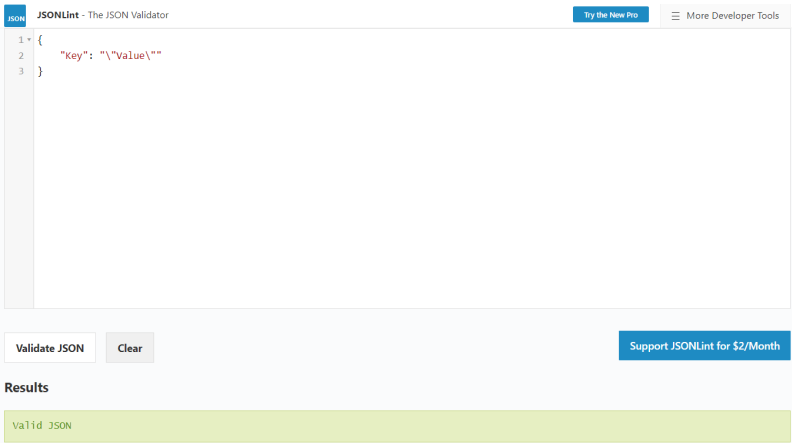


**Figure 11.1:** Controlled with valid data

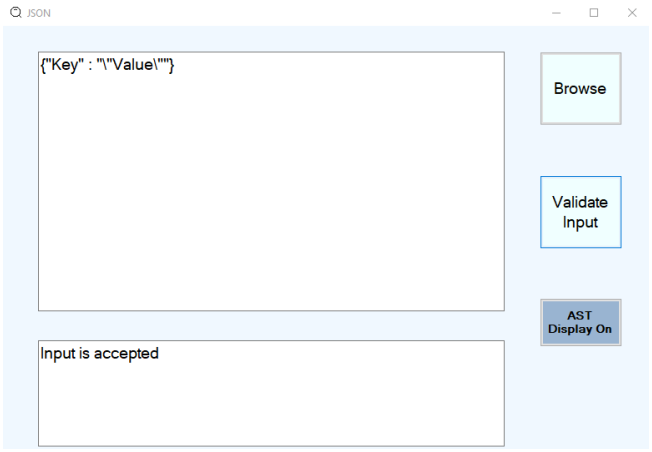


**Figure 11.2:** Experimental with valid data

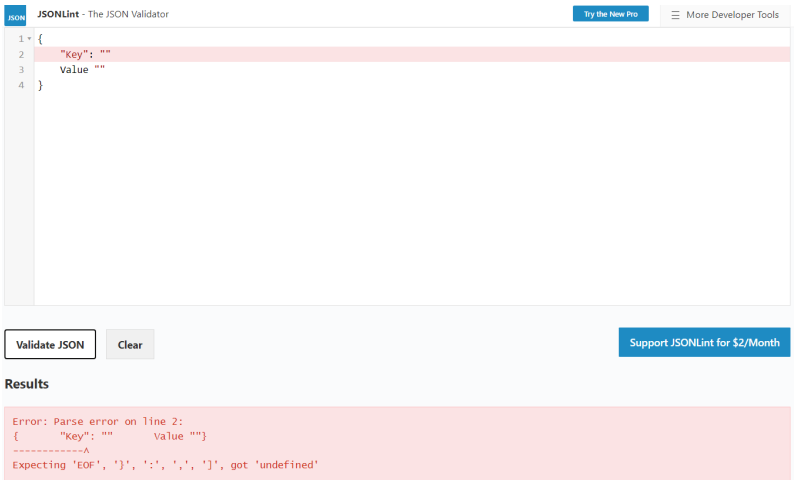


**Figure 12.1:** Controlled with invalid data

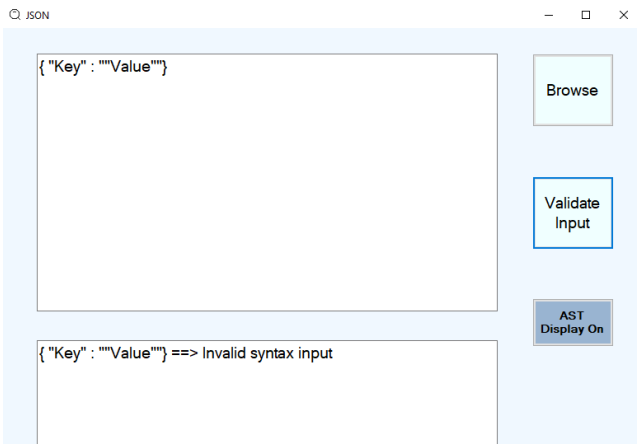


**Figure 12.2:** Experimental with invalid data

## Contributions and Conclusion

The contributions of this project include the insightful samples and projects that were provided in this course, lecture materials, Coco/R user manual, jsonlint.com validator, and Json testing files. My work in combination allowed for a successful application to developed for validating Json and displaying the abstract syntax tree

In conclusion, this term project to develop a language based application allowed me to explore the various topics of this course including: languages, lexical analysis, parsing techniques, and compiler constructions using Coco/R. This helps understand the inner workings of a compiler, what a compiler truly is, and the structure of a compiler. Overall a depth of knowledge was gained in this course and applied in this project which uses Coco/R EBNF to define the syntax of JSON and generate a parser to generate and visualizat an AST upon an error free input.

Moving forward more additions can be made in the implementation such as utilizing the symbol table from Coco/R user manual and checking for duplicates as well as improving efficiency as some slowdown can be observed for larger json files.

## References


**[1] Dr. Dar-jen Chang: Lecture materials and samples**

**[2] Json.org: Json structure visualizations**

**[3] Jsonlint.com: Json controlled validation**

**[4] https://project-awesome.org/jdorfman/awesome-json-datasets: Json testing files**

**[5] Coco User Manual: Helping understanding and provided samples such as symbol table sample code**