JSON Parser using JFlex and CUP

Dhruv Ghulati

November 23, 2015

The system I have built allows JSON formatted strings to be parsed, and rejects any strings which are not formatted in terms of JSON.

A Parser file creates the context-free grammar which allows a parse tree to be built up, and via providing different optionality over different nodes, allows various JSON elements to be identified and correctly passed through the system as valid. This is in my Parser.cup file as attached. The key purpose of a parser file is to generate the rules by which a non-terminal element can be defined in reference to other non-terminal elements or terminals.

In this way, a whole language can be defined such that any keyword string allowed in the language should pass through this parse tree without facing any barriers, and any keyword string not allowed in the language should face a barrier or roadblock and not be able to traverse further.

The Scanner file serves the purpose of defining the terminals which are utilised as terminals or tree endpoints within the Parser.cup file. The main importance of the Scanner file is the Regex definitions, which use real regex characters to define what actually is a specific symbol. I decided to create new Symbols and ignore what SymbolFactory defined as a digit, among other overrides.

1 Scanner File

My Scanner file is split into the definition of terminals which are simple, for example `LCURLYBRACE` and other terminals which are much harder, for example `STRING` or `NUMBER`, which require complex definitions.

The way I set up my file was to first define terminals, then define the beginning of a `STRING`object and initiate it, and then define the rules that define a string object, which is appended to and built over time. When a string object ends via `LCURLYBRACE`, we go back to the initial state.

The use of `yytext()`or `Integer(yytext())`was not necessary, as I would not need to parse or use the input strings inputted by the user in any way for maniputlation. In fact, even the use of states in the Scanner was not necessary, but here I decided to use the `<STRING>`state when I actually could have defined a string symbol in one line.

References are available in the file as to how I defined my string and number objects.

2 Parser File

The Parser file contains no precedence rules over left or right parsing for specific types of terminals, as there is no manipulation of objects or elements when parsing JSON (for example, multiplication or addition of integers).

For my Parser file, I used the exact notation given within the JSON definition link provided in the coursework. This parse tree ends where we have a value, which also then allows an object and array to allow us to create multiple objects within the JSON. I thus ensured that the structure of my parsing was per the official definition.

3 Tests

The tests I used include the official tests within the examples provided on the JSON website. I also have tests conducted for negative and positive exponent numbers with decimal places, multi-array objects, and nested objects, among others. My coursework successfully passes all valid JSON and fails invalid JSON (I tested if I removed a comma, forgot a colon, misaligned brackets etc.). I have decided not to screenshot the results of every test from my terminal, but you will be able to view the .test files within my submission and test these in my terminal.

4 Problems tackled during coursework

- I did not know that I needed to initiate a new StringBuffer in order to append items to my str object, and add this to the Scanner class within Scanner.jflex.
- I did not know I needed to define `%state`STRING within my Scanner.jflex file.
- I initially started with a very simple definition of NUMBER, then built it up to allow exponents and optional decimal numbers. I had issues with forgetting about optional +- within my exponents, as well as not recognising that JFlex 1.6.1 doesn't allow `\d` as a valid Regex definition for digit, only [0-9].
- There were several syntactical mistakes e.g. forgetting to comment out text comments, putting commas instead of semi-colons within multiple executions within curly brackets etc, putting single quotes instead of double quotes in my STRING definition.
- There were several inbuilt SymbolFactory definitions for things like COMMA, QUOTEMARK etc already, but to be on the safe side I defined these myself as new Symbols.
- I did not realise that JFlex 1.6.1, my version, did not allow for certain regex definitions e.g. `\d`.Thus, I had to play around to define these the long way.
- The use of definitions, e.g.

  ```
  digits = 0|[1-9][0-9]*
  letter = [a-zA-Z]
  ```

  was not allowed within the syntax of my Scanner.jflex file, so my Regex strings had to be very long just for compilation to be successful.

5 Appendix

5.1 Scanner.jflex

```
package Example;

import java_cup.runtime.SymbolFactory;
%%
%cup
%class Scanner

%{
  public Scanner(java.io.InputStream r, SymbolFactory sf){
    this(r);
    this.sf=sf;
  }
  private SymbolFactory sf;
  private StringBuffer str = new StringBuffer();
%}//End of class definition for Scanner

%eofval{
  return sf.newSymbol("EOF",sym.EOF);//This denotes the end of your string
%eofval}

%state STRING

%%
<YYINITIAL> "{" { return sf.newSymbol("Left Curly Brace",sym.LCURLYBRACE);}
<YYINITIAL> "}" { return sf.newSymbol("Right Curly Brace",sym.RCURLYBRACE);}

//Here I define the terminals to be defined in my cup file, and what symbols to
    generate.
//http://stackoverflow.com/questions/32155133/regex-to-match-a-json-string
//http://stackoverflow.com/questions/13340717/json-numbers-regular-expression
<YYINITIAL> {

-?(0|[1-9][0-9]*)/*Positive or negative zero or any digits starting with 1-9*/
(\.[0-9]+)?/*Followed by one or zero optional decimal places followed by lots of
    digits, both of which optional*/
([eE][+|-]?[0-9]+)? /*Optional exponents with one or more digits*/
{ return sf.newSymbol("Number",sym.NUMBER);}
[ \t\r\n\f] { /* ignore white space. */ }
"[" { return sf.newSymbol("Left Square Bracket",sym.LSQBRACKET);}
"]" { return sf.newSymbol("Right Square Bracket",sym.RSQBRACKET);}
"," { return sf.newSymbol("Comma",sym.COMMA);}
":" { return sf.newSymbol("Colon",sym.COLON);}//Autocompletion to sym.Colon, but I
    kept my convention to define all terminals in capitals
/*A double quotation starts the definition for a STRING*/
\" {str.setLength(0); yybegin(STRING);}
"null" { return sf.newSymbol("Null", sym.NULL); }
"true" | "false" { return sf.newSymbol("Boolean", sym.BOOLEAN); }
}

<STRING> {
  //This is when the string ends, and it has been defined
```

```
49    \" { yybegin(YYINITIAL); return sf.newSymbol("String", sym.STRING, str.toString())
        ; }
      /*Account for actions within string literal*/
51    [^\n\r\"\\]+ { str.append( yytext() ); } //A string is anything that is not a new
        line, return, a quotation or a backslash.
      /*Tab within a string literal*/
53    \\t { str.append('\t'); }
      /*New line within a string literal*/
55    \\n { str.append('\n'); }
      /*Returns within a string literal*/
57    \\r  { str.append('\r'); }
      /*Quotation within a string literal*/
59    \\\" { str.append('\"'); }
      /*Backslash within a string literal*/
61    \\  { str.append('\\'); }
      //How to complete a string?
63  }

65  . {System.err.println("Illegal character: " + yytext());}
```

Listing 1: /Users/dhruv/Dropbox/uclcs/gc_004_compilers/coursework/minimal/jflex/Scanner.jflex

5.2 Parser.cup

```
1   package Example;

3   import java_cup.runtime.*;

5   parser code {:
      public static void main(String args[]) throws Exception {
7       SymbolFactory sf = new DefaultSymbolFactory();
        if (args.length==0) new Parser(new Scanner(System.in,sf),sf).parse();
9       else new Parser(new Scanner(new java.io.FileInputStream(args[0]),sf),sf).parse()
        ;
      }
11  :}

13  terminal COMMA, LSQBRACKET, RSQBRACKET, RCURLYBRACE, LCURLYBRACE, COLON, QUOTEMARK;
    terminal NUMBER, STRING;
15  terminal BOOLEAN, NULL; //Should you define a new STRING, or just use the standard
        documentation?

17  non terminal object, members, pair,array, value, elements;//char autocompletes so
        need another terminal at the end

19  //An object is an unordered set of name/value pairs. An object begins with { (left
        brace) and ends with } (right brace).
    object ::= LCURLYBRACE RCURLYBRACE | LCURLYBRACE members RCURLYBRACE {: System.out.
        println("Object found"); :};
21  // | LCURLYBRACE RCURLYBRACE | LCURLYBRACE members RCURLYBRACE {: System.out.println
        ("Object found"); :};
    members ::= pair | pair COMMA members {: System.out.println("Members found"); :};
23  // //Each name is followed by : (colon) and the name/value pairs are separated by ,
        (comma).
    pair ::= STRING COLON value {: System.out.println("Pair found"); :};
```

```
25 //An array is an ordered collection of values. An array begins with [ (left bracket)
       and ends with ] (right bracket).
   array ::= LSQBRACKET elements RSQBRACKET | LSQBRACKET RSQBRACKET {: System.out.
       println("Array found"); :};
27 // //Values are separated by , (comma).
   elements ::= value | value COMMA elements {: System.out.println("Elements found");
       :};
29 //A value can be a string in double quotes, or a number, or true or false or null,
       or an object or an array. These structures can be nested.
   value ::= object| array| NULL | BOOLEAN | NUMBER | STRING {: System.out.println("
       Value found"); :};
```

Listing 2: /Users/dhruv/Dropbox/uclcs/gc_004_compilers/coursework/minimal/cup/Parser.cup

5.3 Test files

boolean.test
booleanfail.test
coursework.test
courseworkfail.test
curlyfail.test
numbers.test
numbersexp.test
simple.test
simplefail.test
test1.test
test1fail.test
test2.test
test2fail.test
test3.test
test3fail.test
test4.test
test4fail.test
underscore.test
underscorefail.test