**Compilers Coursework: Parser for JSON**

I used the Eclipse plug-in developed by Technische Universität München to develop my code, rather than editing the ‘minimal’ example due to issues I faced with the UCL Linux system. When creating a new CUP project using Eclipse, it creates all the necessary files for testing and creating a lexer and parser using a simple calculating example. I then edited the CUP, JFLEX and test files, whilst retaining the base format from the example.

**JFLEX**

I used regular expressions extensively in my JFLEX file, namely to describe all numbers that are allowed in JSON and also the characters which are allowed within a string in JSON. – Complete explanation of my thinking is given within the JFLEX file.

As found in the JFLEX specification: “The lexical state YYINITIAL is predefined and is also the state in which the lexer begins scanning.”, hence was appropriate to define YYINITIAL as { and } so that the scanning only begins within these curly brackets – which is JSON syntax. Also the rules are defined only within the YYINITIAL state so that the sets of expressions are only allowed when they follow the initial ({) conditions.

I defined the prime elements that are the base of all the grammar rules in the CUP files. – namely: Comma for the *members* and *elements* definition, Colon for the *pair* definition, Left & Right square brackets for the *array* definition*,* Quote for the string, Booleans true and false, null, and numbers for the  *value* definition, and finally the characters – for the *chars* definition.

One note, since there are no return values, if the input.txt file is correct JSON syntax no message is displayed. But, an error is shown if it the txt file has incorrect syntax.

**CUP**

Within the CUP file I defined all the tokens I defined in the JFLEX file, as terminals. The majority of the characters had no specific type when created. But numbers were defined as a Double, hence the terminal also needed to be defined as a Double. Similarly for true & false being Boolean, and Char being a String.

The non terminals were defined using the JSON structure and name conventions as found on their website – Each of these non terminals had definitions in the grammar rules for JSON each of which is fully described in my CUP file.

**Testing**

Due to the original issues I faced, in order to make it easier to find out where any errors were being caused, I edited my test file (input.txt) and tested it by running firstly the build.xml to test whether the build was successful and then the driver class which checks the test file pretty much after every edit I made to the lexer and parser.

What I decided to do, was rather than creating various test files, I would just add to the existing file, so eventually the test file would include nearly every situation acceptable in JSON.

The final output I received within Eclipse when running the build.xml was:

Buildfile: C:\Users\Nick\Documents\UCL\GC04\Attempt\build.xml

generate:

[cup] This is CUP v0.11b 20141204 (SVN rev 60)

[cup] Authors : Scott E. Hudson, Frank Flannery, Michael Petter and C. Scott Ananian

[cup] Bugreports to petter@cs.tum.edu

[cup] checking, whether this run is necessary

[cup] Parserfile and symbolfile are existing

[cup] Parserfile C:\Users\Nick\Documents\UCL\GC04\Attempt\src\cup\example\Parser.java is actual

[cup] SymbolfileC:\Users\Nick\Documents\UCL\GC04\Attempt\src\cup\example\sym.java is actual

[cup] skipping generation of C:\Users\Nick\Documents\UCL\GC04\Attempt/parser.cup

[cup] use option force="true" to override

compile:

BUILD SUCCESSFUL

Total time: 297 milliseconds

Details on the specific tests I performed are in the Testing document.