Luke Harries
GCO4
Tuesday, 7 November 2017

Compilers Coursework

The GCO4 Compilers coursework involved creating a lexer and parser for JavaScript Object Notation (JSON). Compilers translate human-readable source code into hardware optimised machine code. The steps involved can be grouped into two overall stages: analysis and synthesis. The following piece of coursework will focus on the first stage, analysis. The analysis stage involves three steps: the first step is lexical analysis, as performed by a lexer; the second step is syntax analysis, as performed by a parser; and the third step is semantic analysis. For the GCO4 coursework, I was tasked with creating a lexer and parser. The following write up will explain the theory behind both the lexer and parser, how I created them and the testing involved.

Lexical Analysis

Lexical analysis describes the recognition and grouping of words and symbols found in the source program into tokens. These tokens are composed of a sequence of characters, forming a unit of grammar in the programming language.

To create the lexer, I used JFlex which generates the lexer from a spec file, such as the attached Scanner.jflex, containing regular expressions. I created the regular expressions based off of the JSON specification[1]. For example, to declare Unicode characters I wrote the following regular expression to match Unicode strings: \\u[a-fA-F0-9]{4}, specifically matching the characters \u followed by any four hexadecimal characters.

During the generation of the lexer, the nondeterministic finite automata are converted into a minimised deterministic fine state automata, reducing backtracking and hence increasing speed[2].

[1] http://www.json.org/

[2] http://jflex.de/

Syntax Analysis

Syntax analysis determines that the sequence of tokens generated from Lexical Analysis follows the syntax and grammar of the specified language. The parser uses Context Free Grammer (CFG) which is composed of four components: terminals, the input tokens; non-terminals, syntactic tokens; producers, rules by which non-terminals and terminals combine to form non-terminals; and a start symbol, the non-terminal to start with. By repeatedly applying the producers in the form LHS ->RHS it ensures that the input token stream is syntactically valid.

To create the parser, I used Java based Constructor of Useful Parsers (CUP). CUP generates a parser from a spec file, such as the attached Parser.cup. To create the spec file I first defined the non-terminals and terminals, and then the procedures. The contents of Parser.cup file was similarly based off of the JSON specification and my implementation was designed such that it could accept multiple JSON 'blobs', to allow testing of multiple JSON objects/arrays in one input file.

An example of a declared procedure is:

object ::= LCUBRACE key_value_pair_list RCUBRACE | LCUBRACE RCUBRACE;

Where the uppercase terms "LCUBRACE" and "RCUBRACE" are terminals, and the lowercase "object" and "key_value_pair_list" are non-terminals.

Starting with the start symbol of json_blobs each non-terminal symbol is subsequently defined, from the most general to the most specific, to create the Abstract Syntax Tree.

Testing

During development, I frequently tested my lexer and parser with small snippets of code to ensure that the functionality I was implementing was achieved. Following completion, I performed further tests with the five testing files attached (identifiable by their .test extension). The role of each testing file can be summarised in the table below.

File Name	What I was testing
one.test	Nested arrays and objects are valid (data from http://json.org/example.html)
two.test	Different types of numbers are valid
three.test	The lexer and parser can handle booleans, null values and unicode strings
four.test	The specified escaped characters are valid
five.test	Other escaped characters are invalid