1

Encoding a Grammar

The grammar is encoded in a JSON file this was particularly done to increase the amount of structures that can be represented including array and the object key/value pairs. Further reading the JSON file is automated through the python JSON library. Test values can also be inserted into the JSON array to parse through the grammar. The JSON object format allows to describe the grammar rule derivations with much ease compared to other formats, for example describing it using without a object notation format would make it difficult to read the file and involves creating objects for reading. Using the JSON object the key/value pairs abstract the process and are within the rules of the program. Further in the encoding of the rules object in the JSON format the rules are written as follows,

1. All rules in the grammar must have a terminal as the first symbol on the right hand side of the rule.
2. No variable in the grammar can have two rules with the same terminal as the first symbol on the right hand side.

2

Reading the Grammar

The grammars are read through the python json library. They are read to their respective notation representing the grammar object. For example using the key, "nonterminals", all the nonterminals are read into the nonterminals array. Once the file containing the representation of the grammar is read and tests are done to compare the strings to the alphabet and parse the characters of the string through the derivation function. The algorithms used for the derivation function are as follows,

1. Maintain stack and input buffer.
2. Push the start variable onto the stack.
3. Then repeat the following until either the input buffer is empty:
   (a) Pop an element from the stack.
   (b) If the popped element is a variable, look at the next character in the input and use that to determine which rule to apply. Then push the right hand side of that rule onto the stack. If no rule matches the next input reject the string.
   (c) If the popped element is a terminal, make sure it matches the next character in the input and remove both. If it does not match reject the string.
   (d) If the stack is empty and the input buffer is not, reject the string.
   (e) If both the stack and the input buffer are empty, accept the string.

3

Testing

The string is processed through two main functions, the first function compares the string by parsing the characters to be compared against the alphabet, any discrepancies lead to the function returning a false '0' value which at that point the program will cease execution. The second function parses the characters in the string using the derivation function with the algorithm described above. Further more strings are checked to evaluate the grammar encoding and the program itself.The grammars tested and encoded in the json files are,

$$\{a^n\#b^n \mid n > 0\}$$

$$\{w\#w^R \mid w \in \{0,1\}^*\}$$
$$\{a^i\#b^j\#c^k\# \mid i = j \text{ and } i,j,k > 0\}$$
$$\{a^n bb^n \mid n > 0\}$$

to see the string used to test the grammar please see the scripts of the programming running.

4

Writeup

1. Why were those particular restrictions placed on the grammars?
   The restrictions placed greatly simplified the algorithm and allowed a simpler data structure such as the stack to be used compared to a much harder data structure (graph) in its place to test the derivation. Further, say that the rule on RHS had the same terminal as the first character. This would cause some sort of branching to determine if the output of the grammar would be possible.
2. If those restrictions weren't there, how would your program need to change?
   The program would need some method of branching, and need a recursive depth first search using a graph to determine if the application of the rules will result in a correct encoding of the grammar. Further it would be a very hard problem to check if the parse tree generated on the recursive descent will be non-ambiguous. For example, another recursionmight produce a correct result. This is taken care of when the terminals on RHS have different first chars.
3. What would happen if the grammar were ambigious?
   This approach would not work since the ambiguity is taken care of by requiring different first character in the RHS of the rule.
4. What other approaches might be used to tell if a string can be generated by a given grammar?
   One way would be to make an actual pushdown autmaton. Otherwise see 2.