Project Proposal

Convert an Input Context-Free Grammar to Chomsky Normal Form

Penny Silliman and Milton Nguy

Table of Contents

Description 3

Expected Inputs 3

Expected Outputs 3

Intended Programming Language 4

Program Design 4

Roles and Responsibilities 4

Test Cases 6

- Test Case 1: 6
- Test Case 2: 7
- Test Case 3: 8
- Test Case 4: 9
- Test Case 5: 10
- Test Case 6: 11

Appendix A 12

Appendix B 13

Description

This project will implement a Context Free Grammar to Chomsky Normal Form conversion. Context Free Grammars (CFG) are non-regular languages or also known as Context Free Languages (CFL). Chomsky Normal Form (CNF) is a Context Free Grammar where all productions rules satisfy one of the following conditions[1]:

1. A non-terminal generating terminal. (e.g., X -> x)
2. A non-terminal generating two non-terminals. (e.g., X -> YZ)
3. Start symbol generating ε (e.g., S -> ε)

Note that there can be more than one CNF for each CFG. CNF's produce the same language as generated by the CFG's they are converted from. For generating a string W of length 'n' requires '2n-1' production or steps in the CNF. So, you would think that the conversion is not helpful as it takes more steps to complete the same string. However, CNF's are used in algorithms for CFG's like bottom-up parsers and CYK algorithms, to reverse engineer from a string to a CFG. Then the question arises can all CFG's be represented in CNF? In fact, all CFG's that do not include an ε in its language have an equivalent CNF.

Expected Inputs

The program will expect one CFG as input; therefore, the program will need one text file as its input. The text file will contain structed data representing a CFG's 4-tuple including the production rules. The program will then read the 4-tuple for the CFG and perform the conversion.

Notes: The empty string (ε) is included as a terminal by default and should not be included in the JSON file. Variables such as "S_0" will be represented as "S0", thus the terminals must also not be comprised of numbers; this is true for both the input and output.

See Appendix A (page 12) for input file type.

Expected Outputs

The program will output one CNF. This text file will represent the resultant CNF from the conversion from the input CFG. The text file will contain structured data representing the CNF's 4-tuple, including the production rules.

Notes: The empty string (ε) is included as a terminal by default and should not be included in the JSON file. Variables such as "S_0" will be represented as "S0", thus the terminals must also not be comprised of numbers; this is true for both the input and output.

See Appendix B (page 13) for output file type.

[1] *GeeksforGeeks. (2019, May 21). Converting Context Free Grammar to Chomsky Normal Form. GeeksforGeeks.* *https://www.geeksforgeeks.org/converting-context-free-grammar-chomsky-normal-form/*

Intended Programming Language

This project will be utilizing Python. This choice was an easy one as Python's library support and extensive documentation make it great for handling structured data from the JSON input file. Which is all in addition to, a personal focus on learning Python as per my career readiness plan. Other programming languages may be considered in the future if there are additional implementations needed or ideas, we want to add on top of it that cannot be coded in Python, but we are confident that we will be able to make the project and produce expected results using Python alone.

Program Design

The program can be broken down into eight functions (see Figure 1). The program execution will start at the main function. The main function will call helper function to do each of the following in order:

1. Load the input file (*see Appendix A*) and parse it for necessary data including the CFG's variables, terminals, production rules, and the start state.
2. Make a new start rule with an additional variable that has not been used already (this step is technically only necessary if the existing start state variable shows up on the RHS of any production rule, otherwise it can be ignored).
3. Eliminate useless productions, these will look like variables that do not exist in any RHS NOT including the start variable.
4. Eliminate epsilons and make relevant variable substitutions.
5. Eliminate unit productions.
6. Eliminate RHS with terminals and non-terminals.
7. Eliminate RHS with more than two non-terminals.
8. Map all results of previous functions to the output file (*see Appendix B*), resulting in the depiction of the 4-tuple for the CNF.

Roles and Responsibilities

We plan to use GitHub in this project to work collaboratively and pair program during times when we are available to meet in person. We will meet on a weekly basis to discuss updates for the project. We have decided to pair program for the entire project but if circumstances arise and we aren't able to meet in person, we will use Microsoft Teams to effectively communicate and discuss work.

Penny and Milton will create a set of tasks that we need to implement for the project. To stay organized, tools that may be used are Trello or other list-making applications. If there are tasks that we could not complete when we programmed in person, we will either meet at another day or finish it offline and push all changes to GitHub once task is completed. We are both responsible for the construction of the program that will allow users to convert input CFGS to Chomsky Normal Form.

This diagram is not very readable in this form, I have attached another way to view this diagram with the Project Proposal submission on Moodle. Please view that to get a good look at the diagram. Thank you and sorry for the inconvenience.

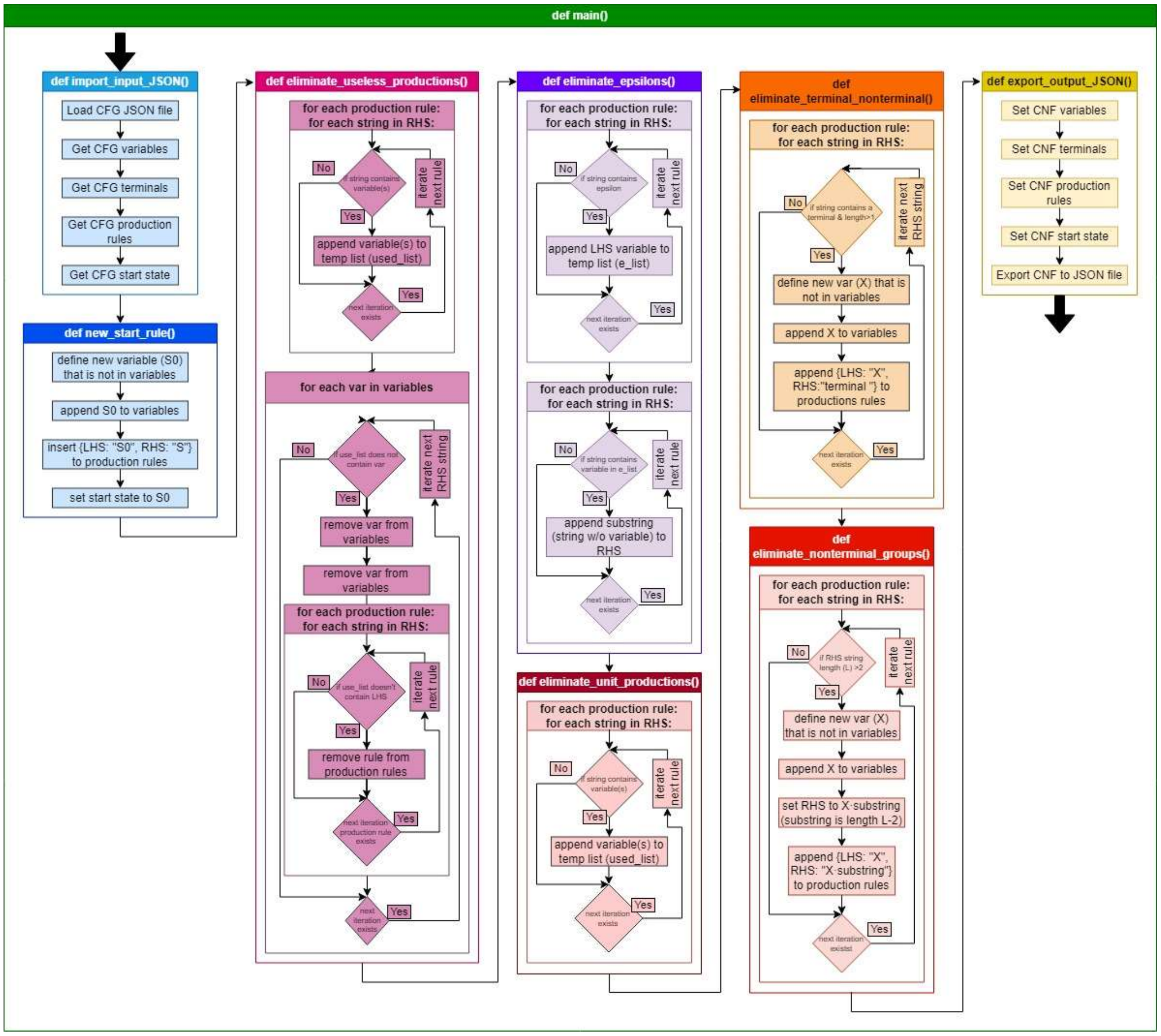

def main()
def import_input_JSON()
Load CFG JSON file
Get CFG variables
Get CFG terminals
Get CFG production rules
Get CFG start state
def new_start_rule()
define new variable (S0) that is not in variables
append S0 to variables
insert {LHS: "S0", RHS: "S"} to production rules
set start state to S0
def eliminate_useless_productions()
for each production rule: for each string in RHS:
if string contains variable(s)
append variable(s) to temp list (used_list)
next iteration exists
iterate next rule
for each var in variables
if use_list does not contain var
remove var from variables
for each production rule: for each string in RHS:
if use_list doesn't contain LHS
remove rule from production rules
next iteration production rule exists
iterate next RHS string
def eliminate_epsilons()
if string contains epsilon
append LHS variable to temp list (e_list)
if string contains variable in e_list
append substring (string w/o variable) to RHS
def eliminate_unit_productions()
def eliminate_terminal_nonterminal()
if string contains a terminal & length>1
define new var (X) that is not in variables
append X to variables
append {LHS: "X", RHS:"terminal "} to productions rules
def eliminate_nonterminal_groups()
if RHS string length (L) >2
set RHS to X·substring (substring is length L-2)
append {LHS: "X", RHS: "X·substring"} to production rules
def export_output_JSON()
Set CNF variables
Set CNF terminals
Set CNF production rules
Set CNF start state
Export CNF to JSON file
Yes
No

Figure 1: Program Execution Block Diagram

Test Cases

Five initial test cases will be used to determine the functionality of the program. The test cases demonstrate a breadth of CFGs functionality with varying number of variables and terminals as well as various kinds of production rules. Using these test cases will allow for observation between achieved results in comparison to expected. It should be noted that additional test cases will be used, but these are the initial test cases that will be considered. It is also possible that the most effective way to implement and code the program design may vary from the by-hand solutions given in these test case examples; however, they should be equivalent to the Python results in order to prove effectiveness of the program.

Test Case 1:

Input:

```
{
  "variables":["S", "T"],
  "terminals":["a", "b"],
  "rules":[
    {"LHS": "S", "RHS": ["aSb", "ST", "Tb"]},
    {"LHS": "T", "RHS": ["bTa", "a", "ε"]}],
  "start": "S"
}
```

Output:

```
{
  "variables":["S", "T", "U", "V", "W", "X"],
  "terminals":["a", "b"],
  "rules":[
    {"LHS": "S", "RHS": ["WV", "b"]},
    {"LHS": "T", "RHS": ["VX", "a"]},
    {"LHS": "U", "RHS": ["a"]},
    {"LHS": "V", "RHS": ["b"]},
    {"LHS": "W", "RHS": ["US"]},
    {"LHS": "X", "RHS": ["TU"]}],
  "start": "S"
}
```

Test Case 2:

Input:

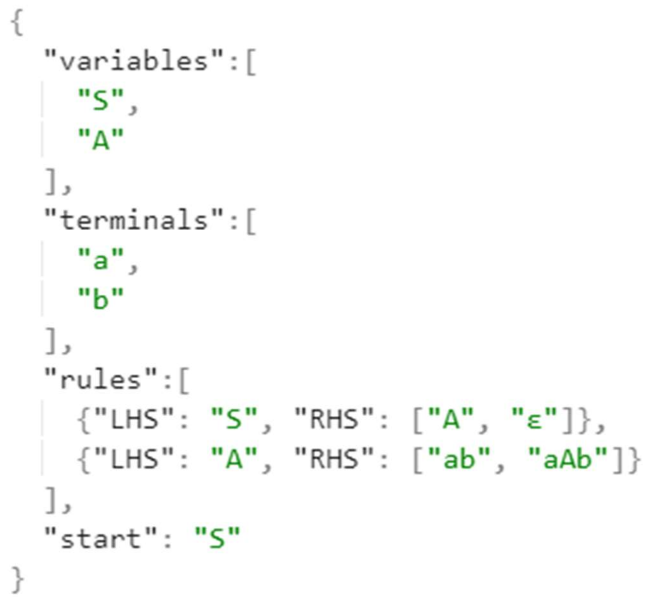
```
{
  "variables":[
    "S",
    "A",
    "B"
  ],
  "terminals":[
    "a",
    "b"
  ],
  "rules":[
    {"LHS": "S", "RHS": ["ASA", "aB"]},
    {"LHS": "A", "RHS": ["B", "S"]},
    {"LHS": "B", "RHS": ["ε", "b"]}
  ],
  "start": "S"
}
```

Output:

```
{
  "variables":[
    "S0",
    "S",
    "A",
    "B",
    "A1",
    "U"
  ],
  "terminals":[
    "a",
    "b"
  ],
  "rules":[
    {"LHS": "S0", "RHS": ["AA1", "UB", "a", "SA", "AS"]},
    {"LHS": "S", "RHS": ["AA1", "UB", "a", "SA", "AS"]},
    {"LHS": "A", "RHS": ["b", "AA1", "UB", "a", "SA", "AS"]},
    {"LHS": "B", "RHS": ["b"]},
    {"LHS": "A1", "RHS": ["SA"]},
    {"LHS": "U", "RHS": ["a"]}
  ],
  "start": "S0"
}
```

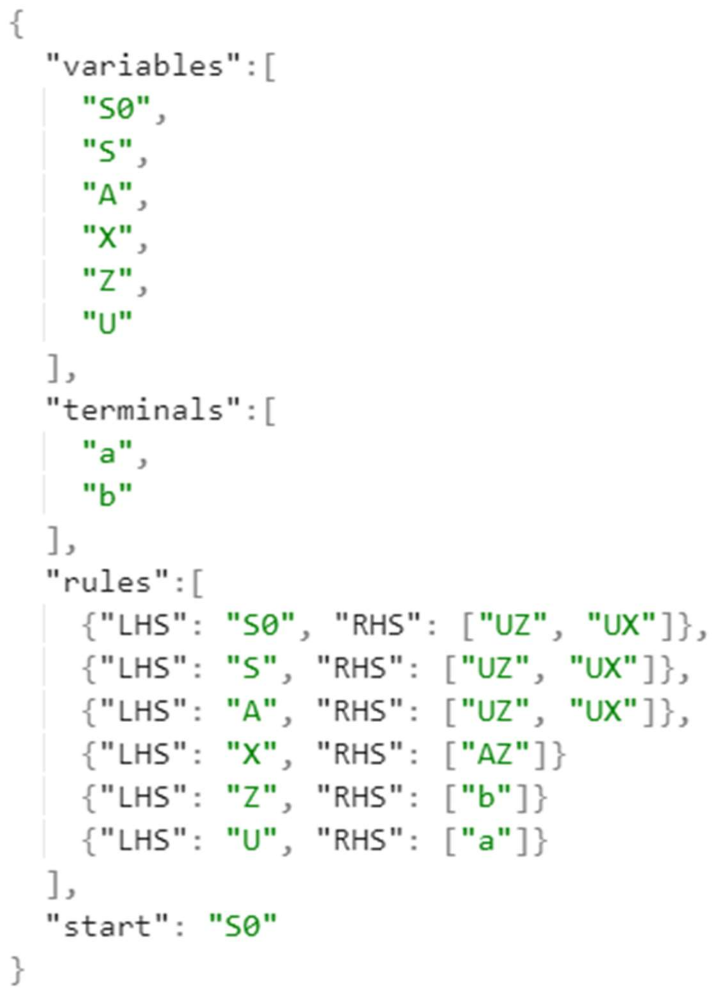
Test Case 3:

Input:



```
{
  "variables":[
    "S",
    "A"
  ],
  "terminals":[
    "a",
    "b"
  ],
  "rules":[
    {"LHS": "S", "RHS": ["A", "ε"]},
    {"LHS": "A", "RHS": ["ab", "aAb"]}
  ],
  "start": "S"
}
```

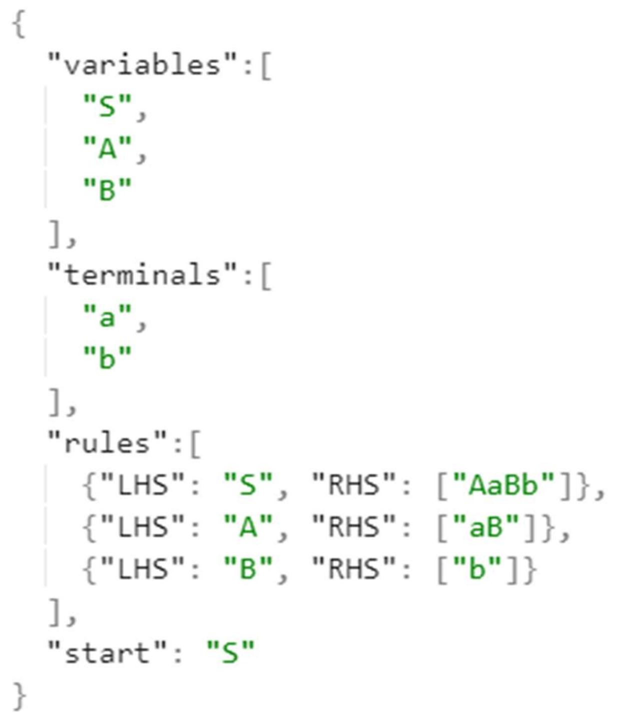
Output:



```
{
  "variables":[
    "S0",
    "S",
    "A",
    "X",
    "Z",
    "U"
  ],
  "terminals":[
    "a",
    "b"
  ],
  "rules":[
    {"LHS": "S0", "RHS": ["UZ", "UX"]},
    {"LHS": "S", "RHS": ["UZ", "UX"]},
    {"LHS": "A", "RHS": ["UZ", "UX"]},
    {"LHS": "X", "RHS": ["AZ"]}
    {"LHS": "Z", "RHS": ["b"]}
    {"LHS": "U", "RHS": ["a"]}
  ],
  "start": "S0"
}
```

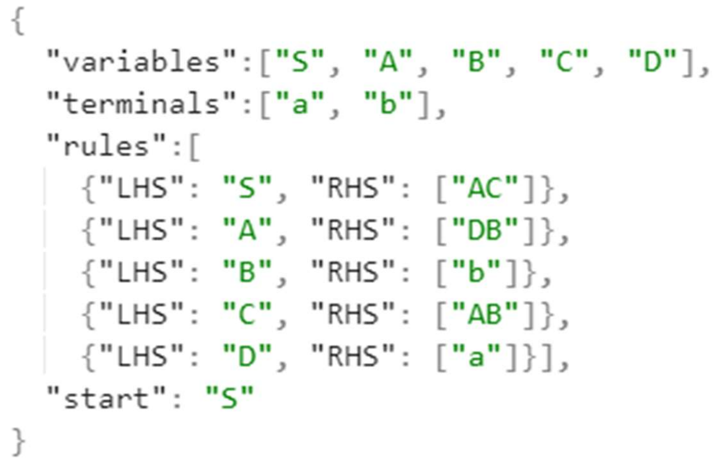
Test Case 4:

Input:



```
{
  "variables":[
    "S",
    "A",
    "B"
  ],
  "terminals":[
    "a",
    "b"
  ],
  "rules":[
    {"LHS": "S", "RHS": ["AaBb"]},
    {"LHS": "A", "RHS": ["aB"]},
    {"LHS": "B", "RHS": ["b"]}
  ],
  "start": "S"
}
```

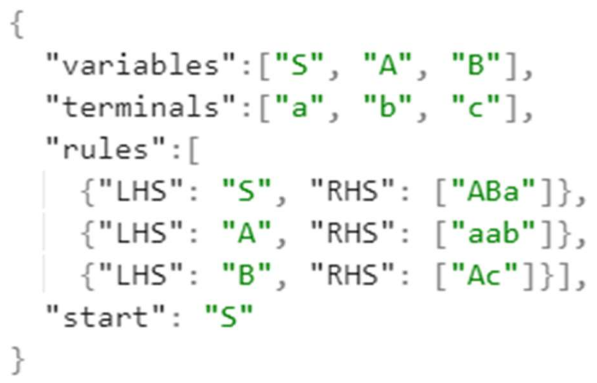
Output:



```
{
  "variables":["S", "A", "B", "C", "D"],
  "terminals":["a", "b"],
  "rules":[
    {"LHS": "S", "RHS": ["AC"]},
    {"LHS": "A", "RHS": ["DB"]},
    {"LHS": "B", "RHS": ["b"]},
    {"LHS": "C", "RHS": ["AB"]},
    {"LHS": "D", "RHS": ["a"]}],
  "start": "S"
}
```

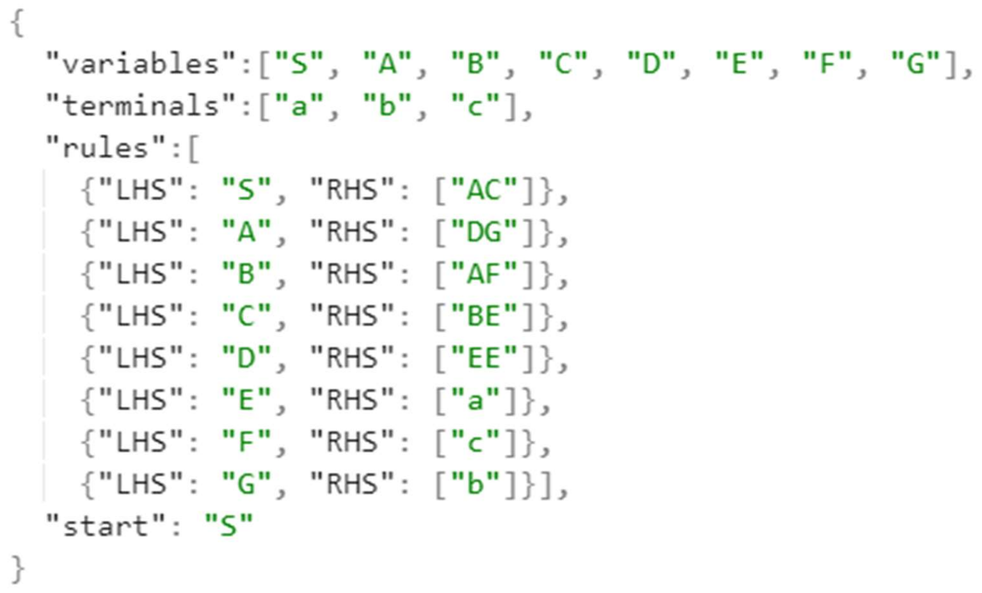
Test Case 5:

Input:



```
{
  "variables":["S", "A", "B"],
  "terminals":["a", "b", "c"],
  "rules":[
    {"LHS": "S", "RHS": ["ABa"]},
    {"LHS": "A", "RHS": ["aab"]},
    {"LHS": "B", "RHS": ["Ac"]}],
  "start": "S"
}
```

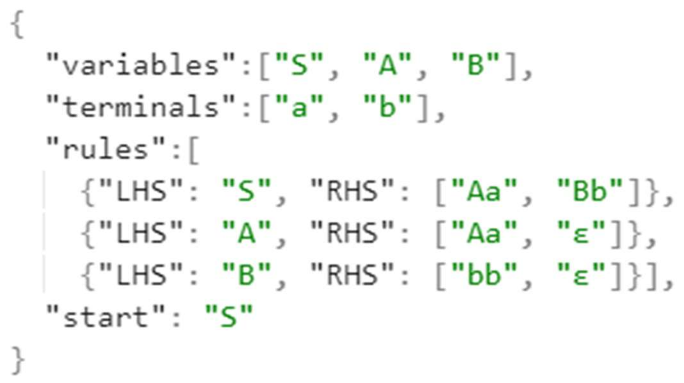
Output:



```
{
  "variables":["S", "A", "B", "C", "D", "E", "F", "G"],
  "terminals":["a", "b", "c"],
  "rules":[
    {"LHS": "S", "RHS": ["AC"]},
    {"LHS": "A", "RHS": ["DG"]},
    {"LHS": "B", "RHS": ["AF"]},
    {"LHS": "C", "RHS": ["BE"]},
    {"LHS": "D", "RHS": ["EE"]},
    {"LHS": "E", "RHS": ["a"]},
    {"LHS": "F", "RHS": ["c"]},
    {"LHS": "G", "RHS": ["b"]}],
  "start": "S"
}
```

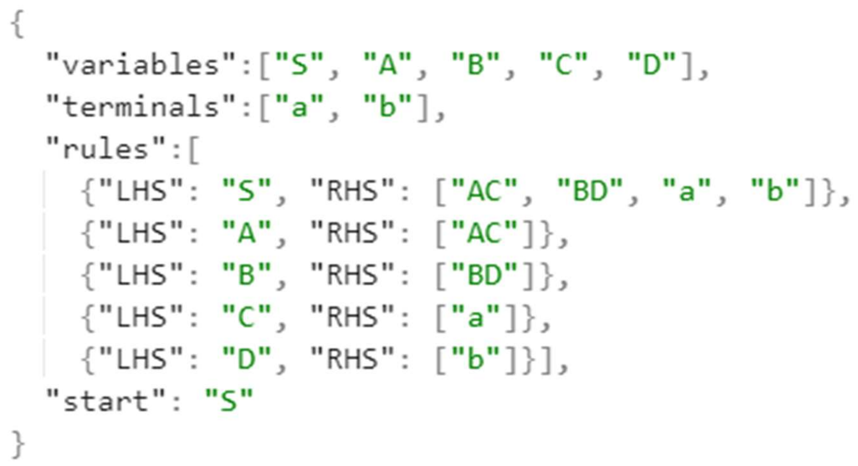
Test Case 6:

Input:



```
{
  "variables":["S", "A", "B"],
  "terminals":["a", "b"],
  "rules":[
    {"LHS": "S", "RHS": ["Aa", "Bb"]},
    {"LHS": "A", "RHS": ["Aa", "ε"]},
    {"LHS": "B", "RHS": ["bb", "ε"]}],
  "start": "S"
}
```

Output:



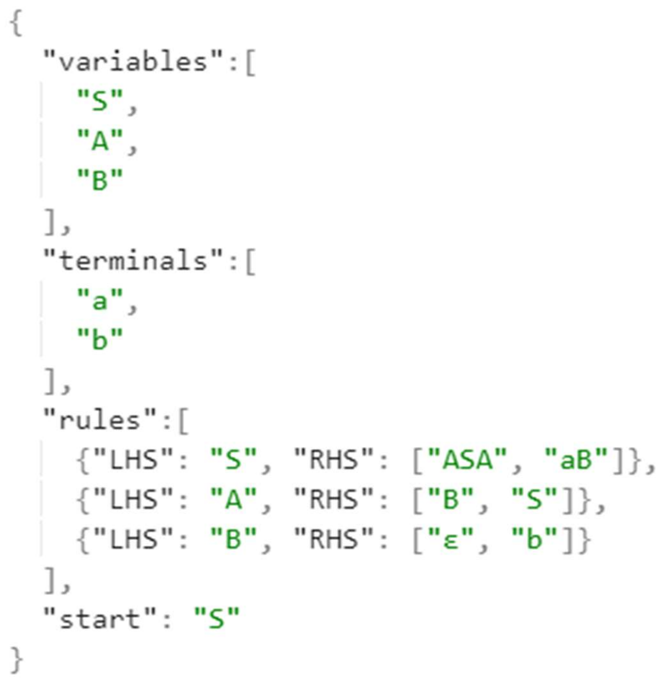
```
{
  "variables":["S", "A", "B", "C", "D"],
  "terminals":["a", "b"],
  "rules":[
    {"LHS": "S", "RHS": ["AC", "BD", "a", "b"]},
    {"LHS": "A", "RHS": ["AC"]},
    {"LHS": "B", "RHS": ["BD"]},
    {"LHS": "C", "RHS": ["a"]},
    {"LHS": "D", "RHS": ["b"]}],
  "start": "S"
}
```

Appendix A

The input file type will be of JSON format (Figure 2). Due to the structuring of the data by using JSON it will be possible to represent the full 4-tuple for a CFG. The input JSON file will include:

1. An array for the finite set of variables (which are non-terminal),
2. An array for the finite set of terminal symbols (which are disjoint from the variables),
3. An array for the set of production rules where each production rule maps a variable to a string, which is shown with 2-tuples:
    a. The first of the 2-tuple is an object representing the Left-Hand Side (LHS) as a single variable,
    b. And the second of the 2-tuple is an array representing the Right-Hand Side (RHS) of the production rule:
        i. Each item in the RHS array is an object representing one string that the LHS variable maps too.
4. And finally, an object representing the start variable.

By using this format, the entire 4-tuple is represented (Variables, Terminals, Production Rules, Start Variable).



```
{
  "variables":[
    "S",
    "A",
    "B"
  ],
  "terminals":[
    "a",
    "b"
  ],
  "rules":[
    {"LHS": "S", "RHS": ["ASA", "aB"]},
    {"LHS": "A", "RHS": ["B", "S"]},
    {"LHS": "B", "RHS": ["ε", "b"]}
  ],
  "start": "S"
}
```

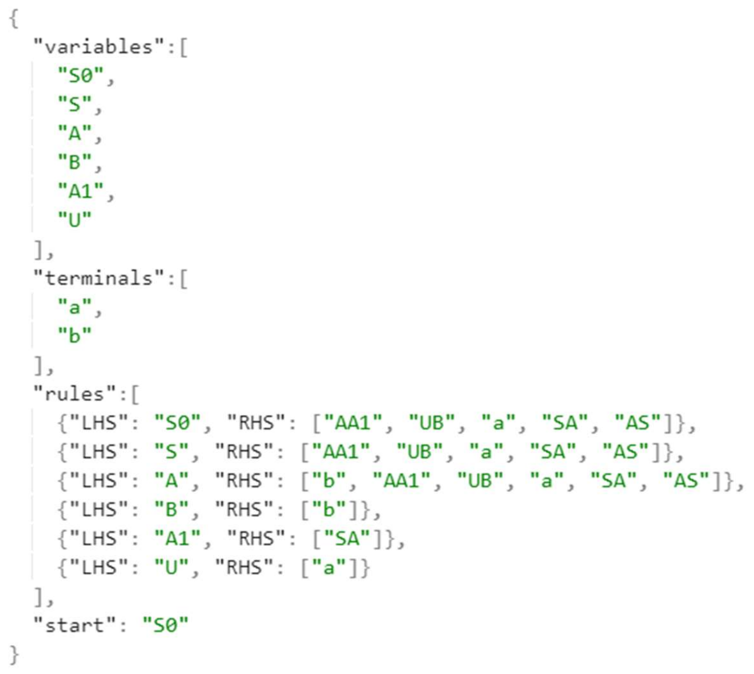
Figure 2: Example Input File (Test Case 2, Input)

Appendix B

The output file will also be of JSON format (Figure 3). Similar to the input file, it will also be preferable to represent the output CNF in JSON format. If fact since a CNF is simply a form of CFG the resulting JSON file will include the same formatting of the input text file. However, for the sake of completeness: the output JSON file will include:

1. An array for the finite set of variables (which are non-terminal),
2. An array for the finite set of terminal symbols (which are disjoint from the variables),
3. An array for the set of production rules where each production rule maps a variable to a string, which is shown with 2-tuples:
    a. The first of the 2-tuple is an object representing the Left-Hand Side (LHS) as a single variable,
    b. And the second of the 2-tuple is an array representing the Right-Hand Side (RHS) of the production rule:
        i. Each item in the RHS array is an object representing one string that the LHS variable maps too.
4. And finally, an object representing the start variable.

By using this format, the entire 4-tuple is represented (Variables, Terminals, Production Rules, Start Variable). Note: the empty string (ε) is included as a terminal by default and should not be included in the JSON file.



```
{
  "variables":[
    "S0",
    "S",
    "A",
    "B",
    "A1",
    "U"
  ],
  "terminals":[
    "a",
    "b"
  ],
  "rules":[
    {"LHS": "S0", "RHS": ["AA1", "UB", "a", "SA", "AS"]},
    {"LHS": "S", "RHS": ["AA1", "UB", "a", "SA", "AS"]},
    {"LHS": "A", "RHS": ["b", "AA1", "UB", "a", "SA", "AS"]},
    {"LHS": "B", "RHS": ["b"]},
    {"LHS": "A1", "RHS": ["SA"]},
    {"LHS": "U", "RHS": ["a"]}
  ],
  "start": "S0"
}
```

Figure 3: Example Output File (Test Case 2, Output)