Part 1:

You are required to build a command line tool that converts a regular expression into its corresponding NFA **(Strictly)** using Thompson's construction algorithm. You must implement the algorithm yourself without using any libraries. The output should be **exactly as the algorithm output**, don't optimize by yourself. Your tool should run like this for example: convert "[A-Za-z]+[0-9]*"

Input format:

The input to the tool is simply a regular expression in text form. The expression may be invalid, so you need to validate it before transforming it first. Example of an invalid regular expression: [A-Za-z

The allowed regular expressions syntax is listed in the following table:

Input	Description	Example	Explanation
|	Or	a|b|c	Matches either a or b or c
(…)	Group	((ABC)|(abc))	Matches either the first group ABC or the second group abc
[…]	Match any character/range of characters within the bracket	[a-cA-C5]	Matches characters in the ranges a-c, A-C and 5
.	Any single character	a.c	Matches any alphanumeric string of length 3 that starts with a and ends with c
?	Zero or one repetition (Optional)	a?	Matches a or empty string
*	0 or more repetition	A*	Matches empty string or any number of As
+	1 or more repetition	A+	Matches any number of As

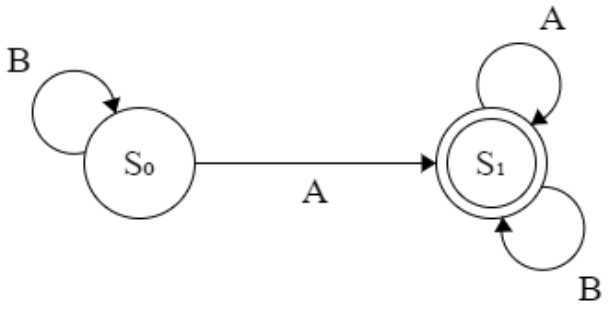
Note: input *literals* are alphanumeric. you **DO NOT** have to handle anything else.

Output format:

You are required to output a JSON file representing the FSM states and transition using the following format:

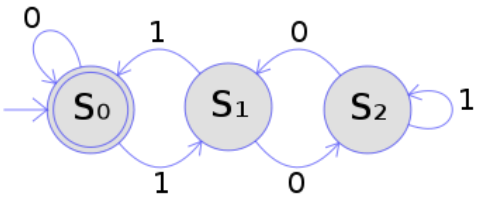
```
{
  "startingState": "S0",
  "S0": {
    "isTerminatingState": false,
    "A": "S1",
    "B": "S0"
  },
  "S1": {
    "isTerminatingState": true,
    "A": "S1",
    "B": "S1"
  }
}
```

Which corresponds to the following FSM



Example:

The following NFA



Would be corresponding to the following JSON:

```
{
  "startingState": "S0",
  "S0": {
    "isTerminatingState": true,
    "0": "S0",
    "1": "S1"
  },
  "S1": {
    "isTerminatingState": false,
    "0": "S2",
    "1": "S0"
  },
  "S2": {
    "isTerminatingState": false,
    "0": "S1",
    "1": "S2"
  }
}
```

You are also required to output an image containing the graph of the NFA, using any graphics library of your choice. You must distinguish between accepting and nonaccepting states. Neat and non-overlapping graphs are mandatory

Part 2:

You are required to implement a program that converts an NFA to a **minimized** DFA. You must implement the algorithm yourself without using any libraries.

Input format:

The input to the tool is a file containing the JSON representation of the NFA using the following format:

```json
{
  "startingState": "S0",
  "S0": {
    "isTerminatingState": false,
    "A": "S1",
    "B": "S0"
  },
  "S1": {
    "isTerminatingState": true,
    "A": "S1",
    "B": "S1"
  }
}
```

Output format:

You are required to output a JSON file representing the **minimized** DFA states and transitions using the same format.

You are also required to output an image containing the graph of the **minimized** DFA, using any graphics library of your choice. You must distinguish between accepting and non-accepting states. Neat and non-overlapping graphs are mandatory.

Notes:

- You are required to follow the output format (json) as it is without any changes to it.
- You are required to use google colab.
- At the end of the notebook in colab you are required to prepare one cell that will accept an input text (that will represent the input test case). and will call the function that will apply the algorithm producing 4 outputs: json file for part1, graph for part 1, json file for part2, and graph for part 2.
- The assignment will be in pairs.
- **Deadline**: exact deadline is to be posted on google classroom