|  |  |  |
| --- | --- | --- |
| **AIN SHAMS UNIVERSITY**  **FACULTY OF ENGINEERING**  **International Credit Hours Engineering Programs (i.CHEP)** |  | |
|  | | |
| Automata & Compatibility Project Documentation  **Prepared by Team 2**  **Submitted to**  **Prof. Dr. Gamal A. Ebrahim**  **T.A Eng. Sally E. Shaker** | | |
|  | |  |
|  | |

# Team Namelist:

|  |  |
| --- | --- |
| Name | ID |
| Abdulrahman Ayman Mahmoud | *19P6458* |
| *Ahmed Sameh Mohamed Mourad* | *19P5861* |
| Osama Ayman Mokhtar Amin | *19P1609* |
| *Amr Haithem Abd Almalek Shams* | *19P2352* |

**Table of Contents**

[Team Namelist: ii](#_Toc134546394)

[1. NFA TO DFA: 5](#_Toc134546395)

[1.1 Conversion Steps 5](#_Toc134546396)

[1.2 Output Screenshots 6](#_Toc134546397)

[2. CFG TO PDA: 7](#_Toc134546398)

[2.1 Conversion Steps 7](#_Toc134546399)

[2.2 Output Screenshots 9](#_Toc134546400)

**Table of Figures:**

[Figure 1 shows an example of an NFA. 7](#_Toc134549821)

[Figure 2 shows the converted DFA. 8](#_Toc134549822)

[Figure 3 shows an example of a CFG. 9](#_Toc134549822)

[Figure 4 shows the generated PDA. 10](#_Toc134549822)

1. NFA TO DFA:
   1. Conversion Steps
2. **Inputting The Graph:** The program allows the user to input the graph in the following format: Node, Node, Label. Alternatively, the user can enter the nodes separately and then connect them by the first format. This provides the necessary input to generate the DFA.
3. **Selecting Start and Final States**: The user must select one start state and at least one final state. If the user does identify them, a pop-up error message will show up.
4. **Construct Transition Table**: To convert a Non-Deterministic Finite Automaton (NFA) to a Deterministic Finite Automaton (DFA), a transition table can be constructed using the Subset Construction Algorithm. The algorithm works by constructing a DFA state for each subset of the NFA's states, where each DFA state corresponds to a set of NFA states that can be reached by following a particular input symbol. The algorithm also calculates the transitions between DFA states by following the transitions between the corresponding sets of NFA states. The resulting transition table will have rows corresponding to the DFA states and columns corresponding to the input symbols, with each entry indicating the next DFA state to transition to when a particular input symbol is encountered. The final DFA state is determined by the set of NFA states that includes the initial state and any other states that can be reached from it by following epsilon transitions.
5. **Display DFA**: To obtain the Non-Deterministic Finite Automaton (NFA) from the transition table, the transition table can be used to determine the set of states and transitions for the NFA. Each row in the table corresponds to a state in the NFA, and the entries in that row specify the set of states that can be reached from that state on each input symbol. To create the NFA, a state is created for each row in the table, and the transitions between the states are determined by the entries in the corresponding row. If multiple entries in a row specify transitions on the same input symbol, then the resulting NFA will have non-deterministic transitions. Additionally, if the table includes epsilon transitions, then the NFA will have epsilon transitions between states. Once the states and transitions have been determined, the initial and final states of the NFA can be identified based on the initial and final states of the corresponding DFA.
   1. Output Screenshots

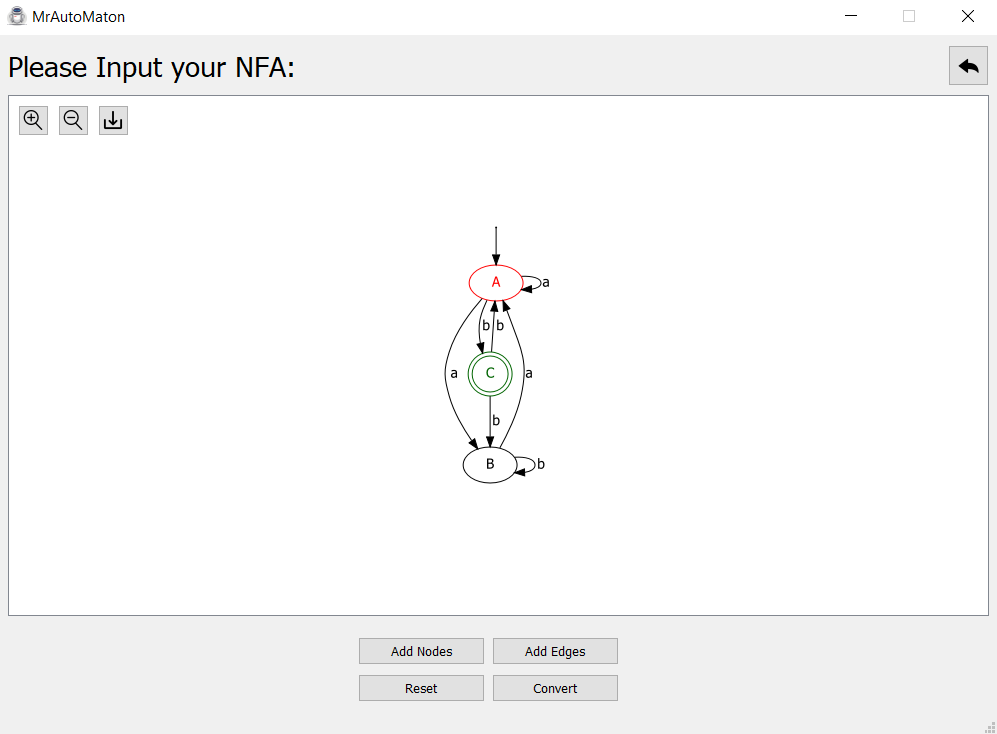


Figure 1 shows an example of an NFA.

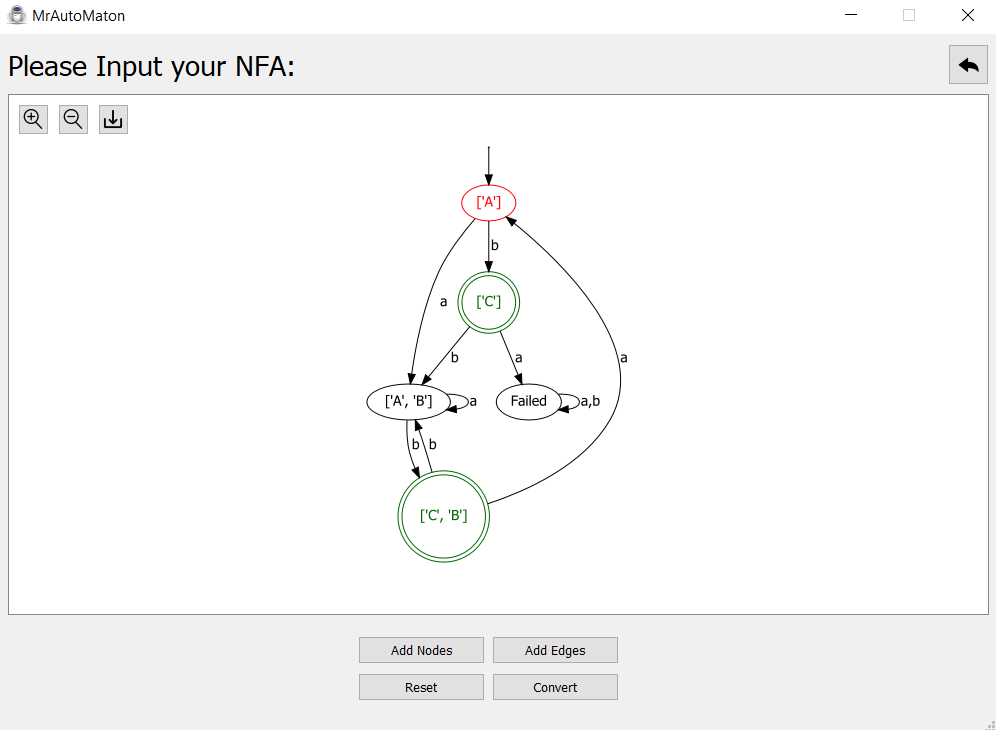
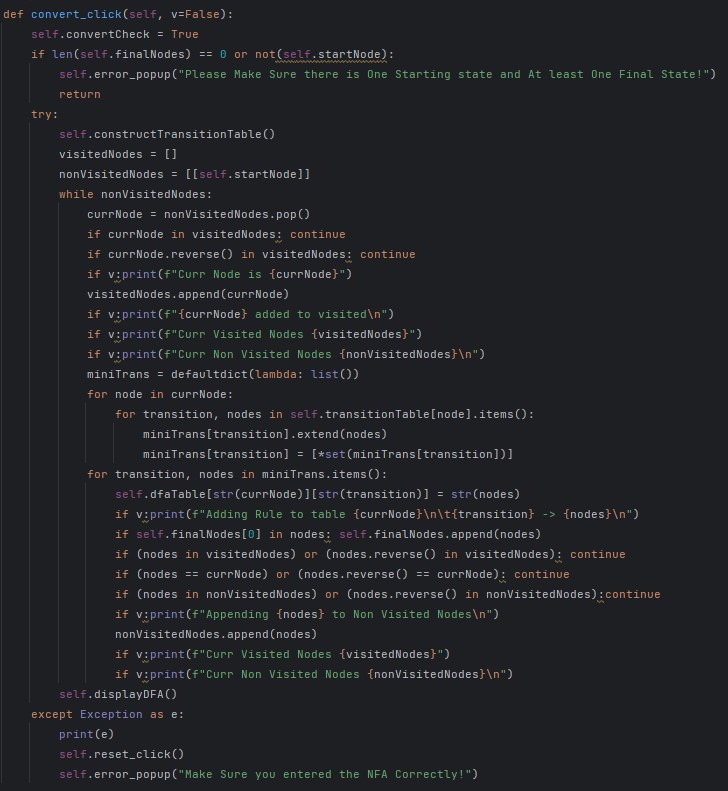
****

Figure 2 shows the converted DFA.

## Code Documentation

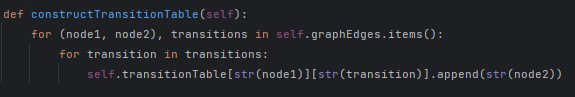
**Handling Convert Click**



**Description:** This function is responsible for converting the entered NFA (Non-Deterministic Finite Automata) to a DFA (Deterministic Finite Automata). Here's a breakdown of its functionality:

1. Set the flag **self.convertCheck** to **True**. This flag is used to check if the conversion process has already been initiated.
2. Check if the input NFA has at least one starting state and one final state. If not, display an error message and return from the function.
3. Call the **constructTransitionTable** function to build a transition table for the NFA.
4. Initialize an empty list **visitedNodes** to keep track of the nodes that have already been visited during the conversion process.
5. Initialize a list **nonVisitedNodes** with the starting state of the NFA.
6. Loop through **nonVisitedNodes** until it is empty.
7. Pop a node **currNode** from **nonVisitedNodes** and check if it has already been visited. If so, continue to the next iteration of the loop.
8. Check if the reverse of **currNode** has already been visited. If so, continue to the next iteration of the loop.
9. Add **currNode** to **visitedNodes**.
10. Create a **miniTrans** dictionary to store the transitions for each state in **currNode**.
11. Loop through each node in **currNode** and its transitions. Add the nodes that can be reached from each transition to **miniTrans**.
12. Add the transitions and their corresponding nodes to the DFA table **self.dfaTable**.
13. If any of the nodes reached by a transition contain a final state, add them to the list of final states.
14. Check if the nodes reached by a transition have already been visited. If not, add them to **nonVisitedNodes**.
15. If **v** (verbosity) is **True**, print some debugging information.
16. Once all nodes have been visited, call **displayDFA** function to display the converted DFA.

**Constructing Transition Table**

****

**Description:** This function constructs a transition table for the input NFA. It loops through each edge of the graph and adds the transition to the corresponding dictionary in **self.transitionTable**.

**Displaying the DFA**

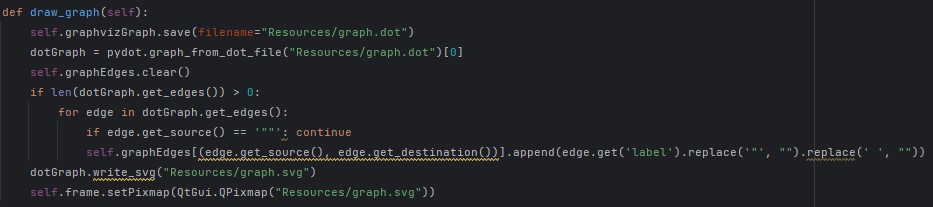
A screen shot of a computer code

Description automatically generated with low confidence

**Description:** This function displays the converted DFA in a GUI window. Here's a breakdown of its functionality:

1. Reset all the variables used during the conversion process.
2. Loop through each node and transition in **self.dfaTable** and add them to the **graphvizGraph** object.
3. Add a "Failed" state to the graph if any transitions are missing.
4. Highlight the starting state in red and the final states in green.
5. Save the graph in a file and display it in the GUI window.

**Drawing the Graph**



**Description:** This function is called by **displayDFA** to save the graph in a file and display it in the GUI window. Here's a breakdown of its functionality:

1. Save the graph in a file named "graph.dot".
2. Read the graph from the file.
3. Loop through each edge in the graph and add it to **self.graphEdges**.
4. Save the graph as an SVG file named "graph.svg".
5. Display the SVG file in the GUI window.

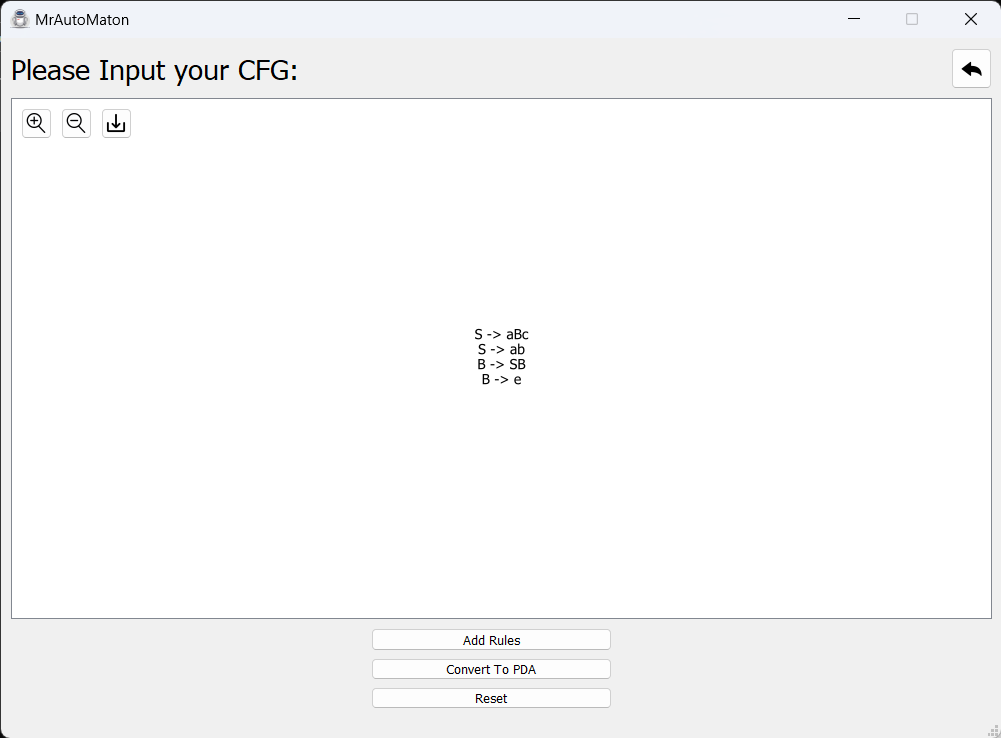
And that's a summary of how each of these functions work together to convert an NFA to a DFA and display it in a GUI window.

# 2. CFG TO PDA:

## Conversion Steps

1. **Inputting CFG Rules:** The program allows the user to input the context-free grammar rules in the following format: {S -> Aa, S -> c, A -> bc}, etc. This provides the necessary input to generate a pushdown automaton.
2. **Extracting The Start State**: The user has the option to input the starting state or leave it empty. If no starting state is specified, the program automatically selects the first encountered non-terminal in the input CFG rules as the starting state. This ensures that the pushdown automaton is generated correctly based on the provided rules.
3. **Extracting the Terminals & Non-terminals**: The program iterates through the input CFG rules and extracts the terminals, storing them in a list. It then performs the same operation to extract the non-terminals in the CFG rules.
4. **Adding the first two States**: The program adds two initial states to the pushdown automaton. The first state, qstart, is added as (ε, ε -> $), while the second state, q1, is added as (ε, ε -> S). These initial states serve as starting points for the pushdown automaton to begin processing the input strings.
5. **Adding Variable Productions to qloop:** The program adds all variable productions to the qloop state. It does this by popping the variable from the top of the stack and sequentially adding its corresponding productions. This process continues until all variable productions have been added to the qloop state.
6. **Adding Terminals to qloop:** The program adds all terminals to the qloop state. It does this by reading the terminal input from the user and popping it from the stack.
7. **Adding Accepting State (qaccept):** The program adds an accepting state, qaccept, to the pushdown automaton. It transitions to this newly added state by popping the $ symbol from the top of the stack and leaving the qloop state. This finalizes the process of generating the pushdown automaton from the input CFG rules.

## 2.2 Output Screenshots



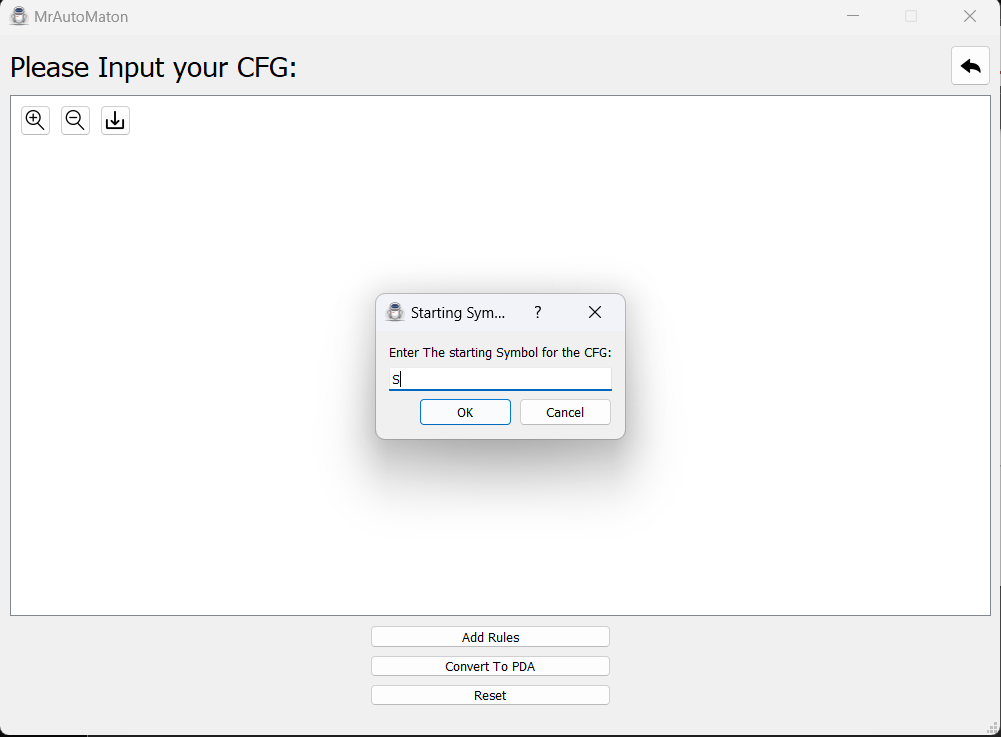


Figure 3 shows an example of a CFG.

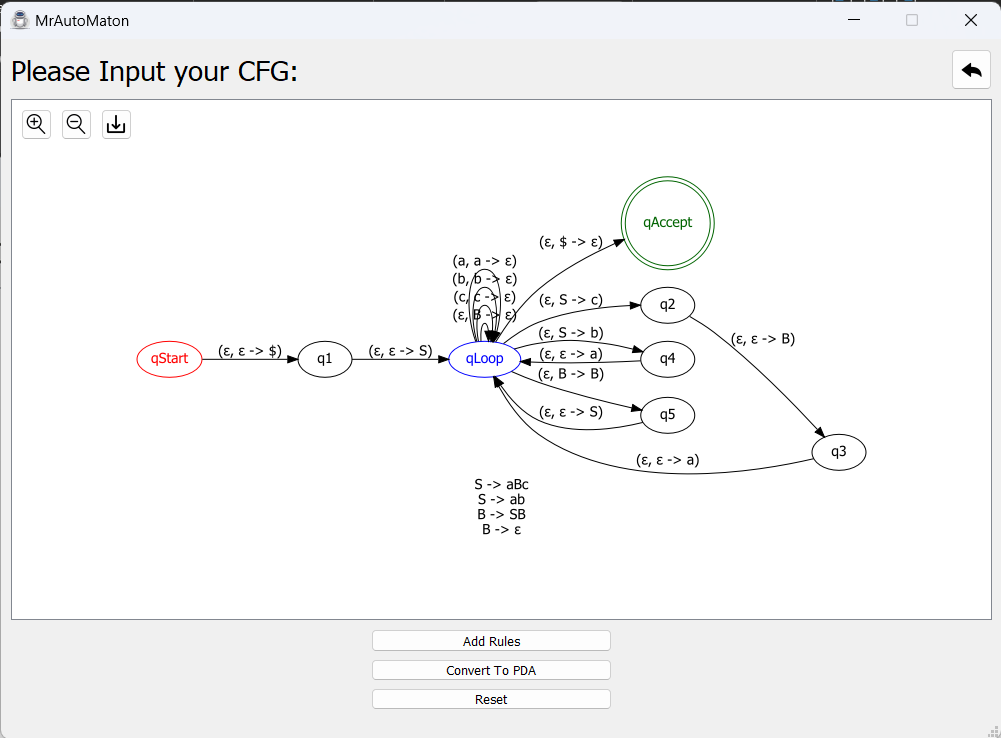


Figure 4 shows the Generated PDA.

## 2.3: Code Documentation

**1st Function: initGrammar()**

A picture containing text, screenshot

Description automatically generated

Description: This function initializes the context-free grammar (CFG) entered by the user. The function takes a reference to the object "self" as the parameter.

It starts by iterating over the grammar rules stored in the "rules" list of the object. For each rule, it splits the string into two parts, the variable and the production part, by using the arrow symbol "->" as the separator. The split result is stored in the "decompose" variable as a list of two strings.

The variable and the production part are then extracted from the "decompose" list and assigned to the "variable" and "production\_list" variables, respectively. The "strip()" method is used to remove any leading or trailing whitespace from the variable and each production in the production list.

The "variables" and "allSymbols" sets are updated by adding the variable and all productions in the production list to the corresponding sets. The "reverse()" method is used to reverse the order of the productions in the list. This is done to facilitate the creation of edges in the PDA graph later in the "convert\_click()" function.

The production list is appended to the list of productions associated with the variable in the "productionRules" dictionary.

The function then iterates over the "allSymbols" set to identify the terminals. It checks if each symbol in the set is a variable by checking if it exists in the "variables" set. If not, it is added to the "terminals" set.

The user is then prompted to enter the starting symbol of the CFG using a dialog box. The input is stored in the "startingSymbol" variable of the object. The function checks if the starting symbol exists in the set of all symbols and if the "rules" list is not empty. If either condition is not met, the "startingSymbol" variable is set to an empty string and the function returns False. Otherwise, it returns True.

Parameters:

* self: reference to the object

Return:

* True if the initialization is successful, False otherwise

------------------------------------------------------------------------------------------------------------------------------

**2nd Function: convert\_click()**

**A computer code on a black background

Description automatically generated with low confidence**

**A screen shot of a computer program

Description automatically generated with low confidence**

Description: This function is called when the "Convert" button is clicked in the GUI. It converts a context-free grammar (CFG) to a pushdown automaton (PDA) and draws a graph of the PDA using the Graphviz library. The function begins by calling the **initGrammar()** function, which initializes the grammar and returns **True** if it is valid and **False** otherwise. If the grammar is not valid, the function displays an error message and returns without doing anything else.

If the grammar is valid, the function creates three nodes in the graphvizGraph object representing the start state, the accept state, and a loop state. It then adds two edges to the graphvizGraph object: an epsilon transition from the start state to the loop state, and an epsilon transition from the loop state to a new state representing the starting symbol of the grammar.

The function then iterates over each variable in the productionRules dictionary, which maps variables to the lists of their productions. For each production, the function creates a copy of the production list and sets **popped** to **False**. It then enters a loop that iterates over each symbol in the production list.

If the current symbol is not the last symbol in the production list, the function creates a new state in the graphvizGraph object representing the current symbol and adds an epsilon transition from the previous state to the new state. If **popped** is **False**, the function adds a non-epsilon transition from the previous state to the new state, labeling it with the current variable and symbol. If **popped** is **True**, the function adds an epsilon transition from the previous state to the new state. The function then sets **nextNode** to the new state and increments **currMaxNode**.

If the current symbol is the last symbol in the production list, the function adds a transition from the previous state to the loop state, labeling it with the current variable and symbol if **popped** is **False**, and labeling it with just the symbol if **popped** is **True**.

After iterating over all productions, the function adds transitions from the loop state to itself for each terminal symbol in the grammar, except for epsilon. Finally, the function adds an epsilon transition from the loop state to the accept state, and calls the **draw\_graph()** function to draw the PDA.

Parameters:

* self: reference to the object

Return:

* None

------------------------------------------------------------------------------------------------------------------------------

**3rd Function: draw\_graph()**

A screen shot of a computer

Description automatically generated with low confidence

Description: This function saves the graphvizGraph object to a file named "PDA.dot" in the "Resources" folder, reads the file into a pydot graph object, writes the graph to an SVG file named "PDA.svg" in the "Resources" folder, and displays the SVG file in the GUI. It uses the Graphviz library to perform these operations.

Parameters:

* self: reference to the object

Return:

* None