**Assignment 1**

**Phase1: Lexical Analyzer Generator**

* **Data structures:**
  + Lexical rules 🡪 expressions:
    - **Exp\_Node**: each node has
      * name to indicate the expression type (class).
      * value to store the expression value.
      * Bool exp to indicate if the expression may contain reference to definitions and has to be resolved first.
    - **In class Rules:**
      * List of Exp\_Node contain all of the expressions.
      * List of string to store all of the rules that has been read from the rules file.
      * String contains the path of the rules file
      * List of strings contains all the names of the expressions in order according to their priority.
      * Private list of Exp\_Node contains all of the definitions.
  + Expressions 🡪 nondeterministic finite automata:
    - **NFA state:**
      * Class line:
        + int to indicate the id of the state that this line is pointing to.
        + string contains the input which resulting in the occurrence of this transition.
      * Int to determine the id of the state.
      * Bool to indicate if this state is an acceptance state or not.
      * String contains the type of the expression of this state is an acceptance state.
      * List of lines contains all possible transitions of this state.
    - **NFA:**
      * Int to indicate the id of the start state.
      * Int to indicate the id of the end state.
    - **Final NFA:**
      * Int to indicate the id of the start state of the combined NFA.
      * List of NFA states contains all of the states found in the NFA.
  + nondeterministic finite automata 🡪 deterministic automata:
    - DFA state:
      * **Ids:** list of integers to keep the number of each nondeterministic state within that state.
      * **Stable:** Boolean variable to check if the state is stable.
      * **name:** string representing the name of the state.
      * **id:** integer represents the number of the state.
      * **trans:** list of lines which represents transitions out of that state.
      * Class **line** which contains:
        + **Input** string which represents the input of this transition line.
        + **to** Deterministic state which represents the destination.
    - DFA:
      * **allStates**: list of all deterministic automata states.
      * **num**: number of states in the deterministic automata.
      * **start:** integer to represent the number of the start state.
  + Minimize deterministic automata:
    - **Class group**
      * int to indicate the id of the group.
      * List of int contains the ids of the states found in this group.
    - List of group contains all of the classes.
    - List of string contains the transition table of the minimal DFA.
    - DFA which is the minimal DFA.
  + Input matching:
    - **In Tokens class:**
      * List of string contains the input that has been read from the input file.
      * List of string contains the recognized token classes that are ready to be written in the output file.
    - **In check class:**
      * Data structure for table of ids: list of nodes each has name, type, and value.
* **Algorithms:**
  + Lexical rules 🡪 expressions:
    - **read\_from\_file():** read the rules from the file and push them them to the rules list.
    - **convert\_to\_expressions():**
      * iterate over the rules list if the string start with:
        + ‘{‘ push them in the expression list as a exp node of name keywords.
        + ‘[‘ push them in the expression list as a exp node of name punctuations.
      * If it contains ‘:’ push it in the definitions list, else push it in the expressions list
    - **resolve\_definitions():**
      * iterate over all definitions and if one of them reference to a previous one, replace it with its value.
      * Iterate over all expressions and if it references to a definition, replace the definition with its value.
      * If the expression contains (a-z) or (A-Z) or (0-9) replace them by “/a”, “/A”, “/0” respectively.
    - **get\_priority(list<string>):** return the string that comes first in the priority list (which is ordered according to the listing of expressions in the rules file).
  + Expressions 🡪 nondeterministic finite automata:
    - **In Final NFA class:**
      * **get\_state(int id):** return a pointer pointing to the NFA state of the given id
      * **add\_state():** add a new state to the all states list, give it a new id and return a pointer pointing on it.
      * **construct\_NDFA(list<Exp\_Node>):** 
        + iterate over all expressions and call a NFA function to construct the NFA corresponding to this expression and push the resulting NFA in a list.
        + Combine all of the NFAs by calling combine function of the NFA class
    - **In NFA class:**
      * **get\_from\_expression(string exp,Final\_NDFA\* fNdfa):**
        + for each char in the exp if it is:

‘(‘: call function get from expression with parameter exp equal to the string between ().

‘+’: Add a new transition from the end state to the start state under the input is lambda.

‘\*’:

Add a new transition from the end state to the start state under the input is lambda.

Add a new start and end state.

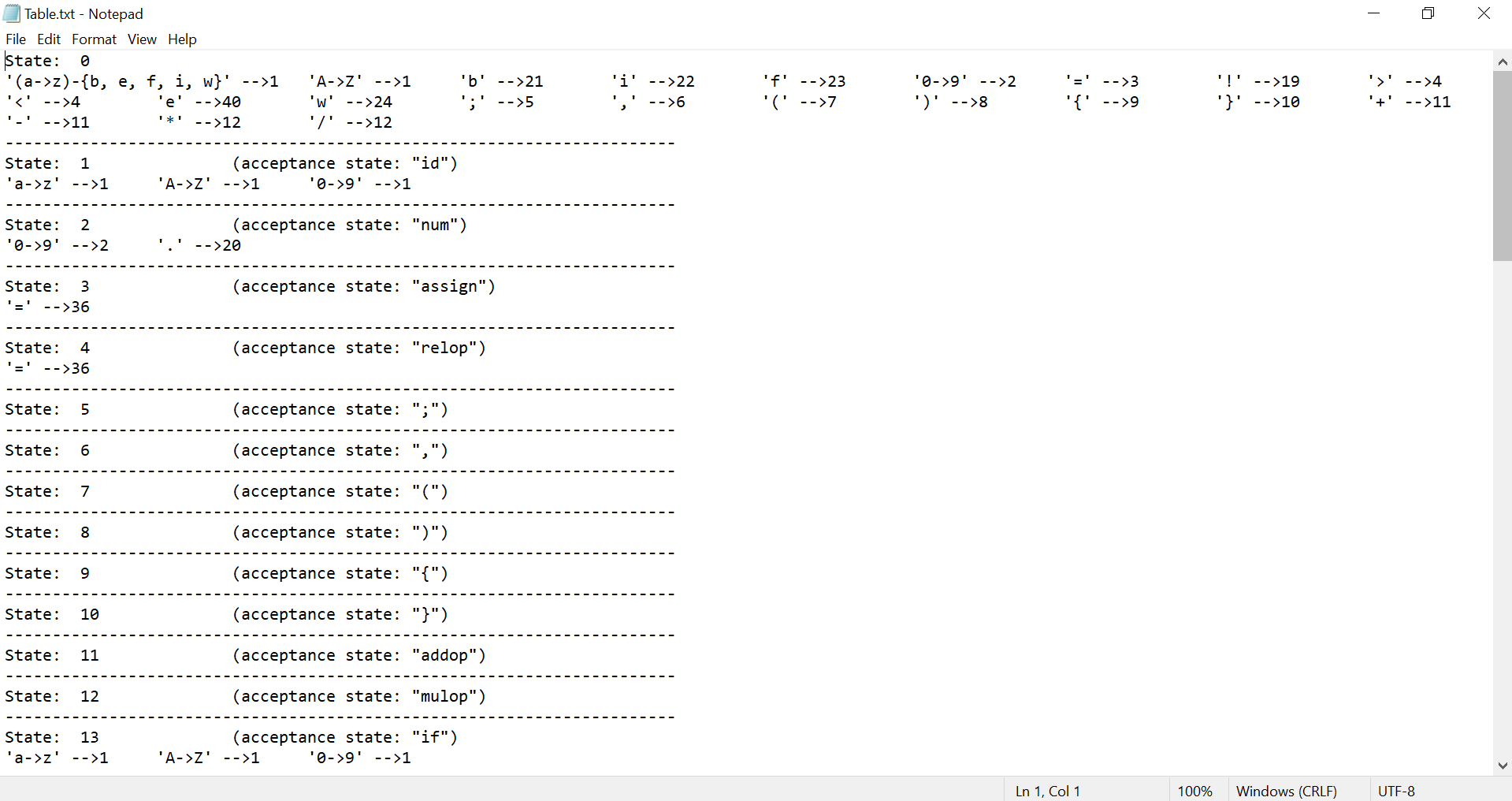
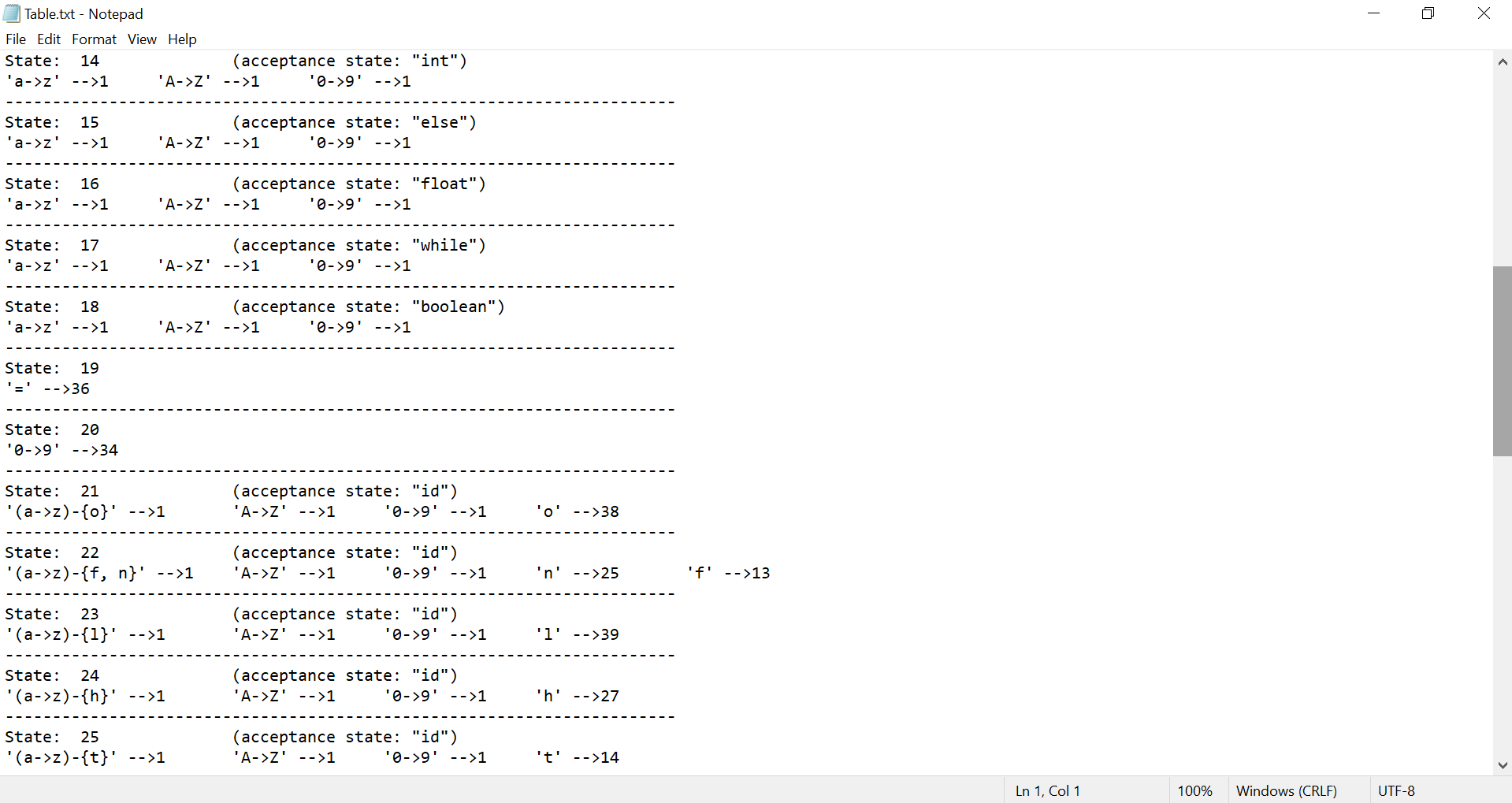
Add a new transition from the new start state to the previous start state under the input is lambda.

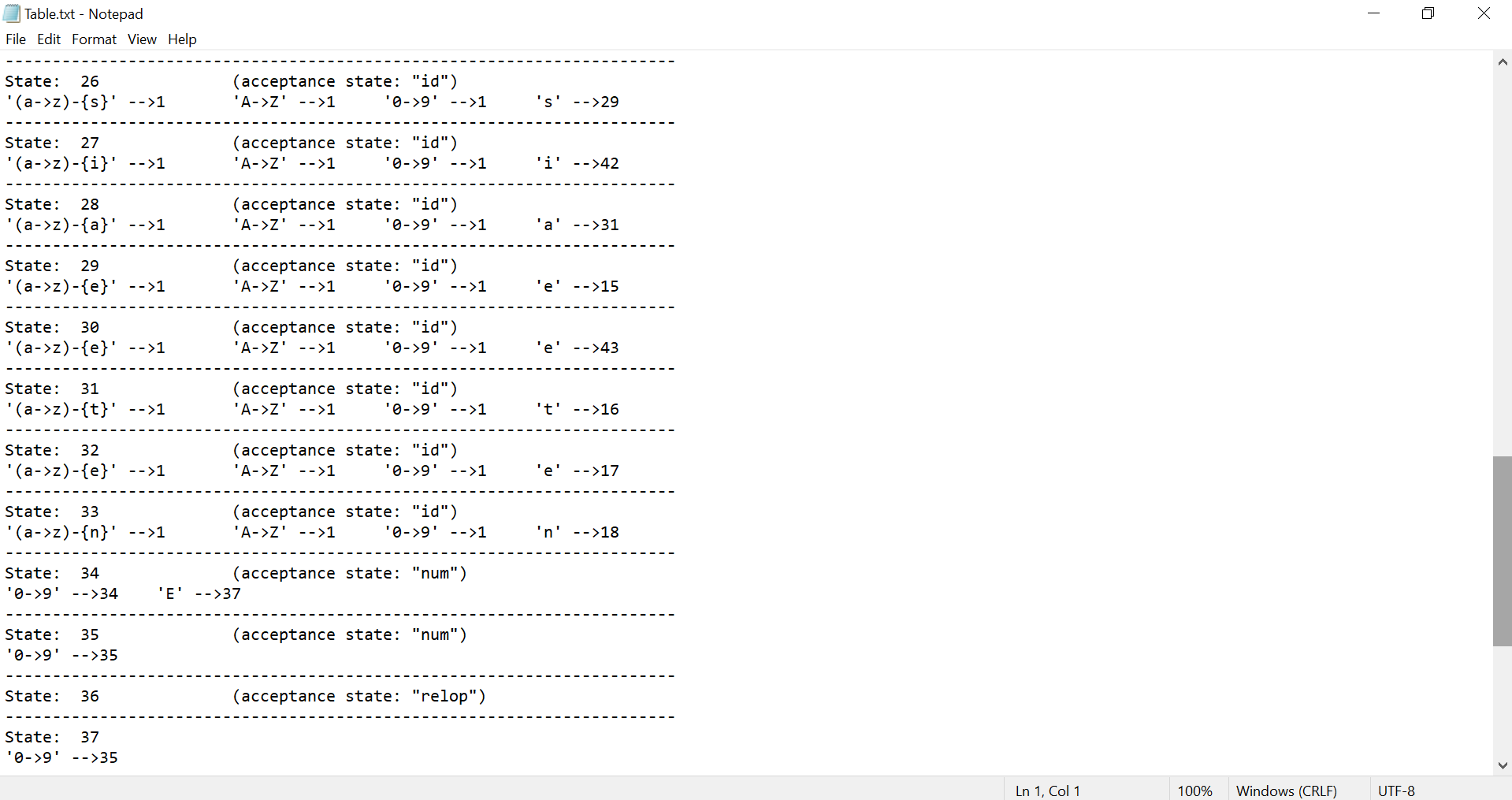
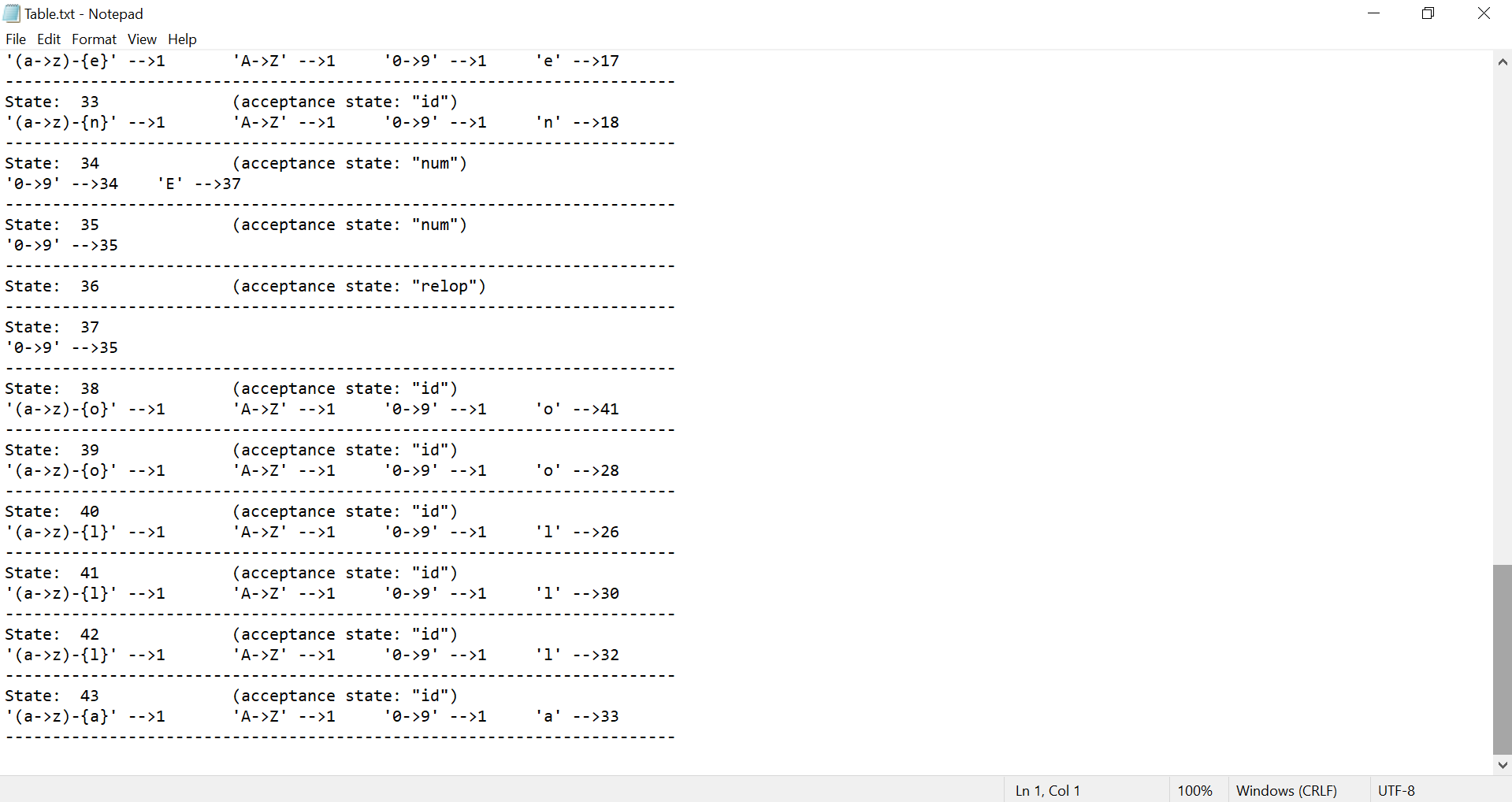
Add a new transition from the previous end state to the new end state under the input is lambda.

Add a new transition from the new start state to the new end state under the input is lambda.

‘|’: add the NFA to a list called ors.

Else: add a new transition from the previous end state to a new state under the input is that char then change the end state to be the new state.

* + - * + If the ors list is not empty combine these states.
        + Return the resulting NFA.
      * **get\_from\_non\_expression(string exp,Final\_NDFA\* fNdfa):** for each char in the exp add a new transition from the previous end state to a new state under the input is that char then change the end state to be the new state.
      * **combine(list<NDFA> states,Final\_NDFA\* fNdfa):**
        + Add a new start and end states.
        + Add a transition from the new state to each start state of the NFAs in the list.
        + Add a transition from each end state of the NFAs in the list to the new end state.
        + Return the resulting NFA.
  + nondeterministic finite automata 🡪 deterministic automata:
    - **getEps:** function to find all states which the current state reaches with epsilon transition.
      * Queue (q) is used to keep track of the id of each nondeterministic state needed to get epsilon transitions out of it.
      * As the queue is not empty, we take the begin of that queue.
      * Then iterate over all lines out of this state.
      * If the transition is epsilon, then we add it to the end of the queue.
    - **getstates():** returns list of all deterministic states.
    - **exists():** function to check if the given deterministic state exists in the automata.
      * In this function we loop over all the deterministic states.
      * Compare the values of ids of each nondeterministic state included in the state.
      * If all ids are the same in both states, then return true.
      * Which means the state is already in the deterministic automata
    - **construct\_DFA():** Build the deterministic automata from given nondeterministic one.
      * Get the start state of the nondeterministic automata and all states reached by epsilon transitions.
      * Create new deterministic state and assign these ids to that state.
      * Create queue of deterministic states and push the start state into it.
      * As the queue is not empty do the following:
        + Extract all the possible inputs of all transitions from this state.
        + Remove all duplicates using unique () function.
        + Check if the list of valid inputs is empty, then we check the stability of the nondeterministic states.
        + If any one of them is stable, then set the current state to be stable.
        + For each valid input, we loop over all the nondeterministic states of the current state.
        + Check if this state has transition under that input, save all the destination states and states reached by epsilon transition out of them.
        + If there are transitions out of that deterministic state under that input, add new deterministic state to the queue.
  + Minimize deterministic automata:
    - **mini(int id,DFA dfa):**
      * iterate over each state present in the group which its id is given as a parameter.
      * If there exist a state which its transitions are different from the others, remove it from this group and add a new group and add this state to it.
      * Call mini() again for all groups until there is new classes are added or all states becomes in a separate group.
    - **get\_class\_of\_state(int id):** return the id of the group which this state is belong to.
    - **lines\_equal(list<DFA\_state::line> t1,list<DFA\_state::line> t2):** 
      * iterate over each list to determine if all lines are present in the other list.
      * Return a bool which indicate if the two lists are equal or not.
    - **get\_by\_id(int id):** return a pointer pointing to the group which have this id.
    - **print\_table(DFA dfa):** push back each state in the minimal DFA with its transitions to the table list.
    - **write\_to\_file():** write the table list to a new file named “Table”.
    - **contain(list<DFA\_state::line> lst,DFA\_state::line l):** check if the given line is in the given list or not by making check that there is a line in the list under the same input of the given line transfer to the same DFA node as the given line.
    - **minimize(DFA dfa):**
      * iterate over each state in the DFA and add it to the acceptance states group or the nonacceptance states group.
      * For the acceptance state group if there are acceptance states that have a different class of tokens divide them into different groups.
      * Minimize each group by calling function mini().
      * For each group add a state to the minimal DFA with the same id and transitions of this group.
  + Input matching:
    - **In Tokens class:**
      * **read\_from\_file():** to read the input from the file which is path is given and store them in the input list.
      * **Write\_to\_file():** to write the output list to a new file call “Output”.
    - **In check class:**
      * **Check():** which takes a list of string each represent line in the input file and minimal DFA.
        + It iterates over each line in the input file then takes each character to generate an accepted token by standing in the start state of DFA and compare if this character can move us to the next state. if yes, then take another character and move through the DFA and if we reach an acceptance state, we save that there is an accepted state and continue till we can’t move again.
        + Then check if we found an acceptance token, we accept this as a token and add it to list of outputs and begin again from after that token.
        + If there is no accepted token, it removes one character then iterate one time again to generate an accepted token and report that there is an error in line (print number of line).
        + we also check if the acceptance token was an id we add it to the table of ids.
      * **Report\_Error():** it checks if we report an error at this line before, don’t report again. Otherwise display an error message.
* **The resultant transition table for the minimal DFA:**

****

* **The resultant stream of tokens:**

