https://github.com/RavaszTamas/FormalLanguagesLaboratories/tree/main/lab4

Example automaton in EBNF format, FA.in:

State = “A” | “B” | … | “Z” | “q0” | “q1” | …

digit_or_letter = “a” | “b” | … | “z” | “A” | … | “Z” | “0” | … | “9” | … (any other character)

Q = “”””State”””” { “|” “””” State “””” }

E = “”" digit_or_letter “””” {“|” “””digit_or_letter””””}

DELTA = “”" “(“State”,” digit_or_letter”)” “->” State””" {“|” “”" “(“State”,” digit_or_letter”)” “->” State””" }

q0 = “”””State””””

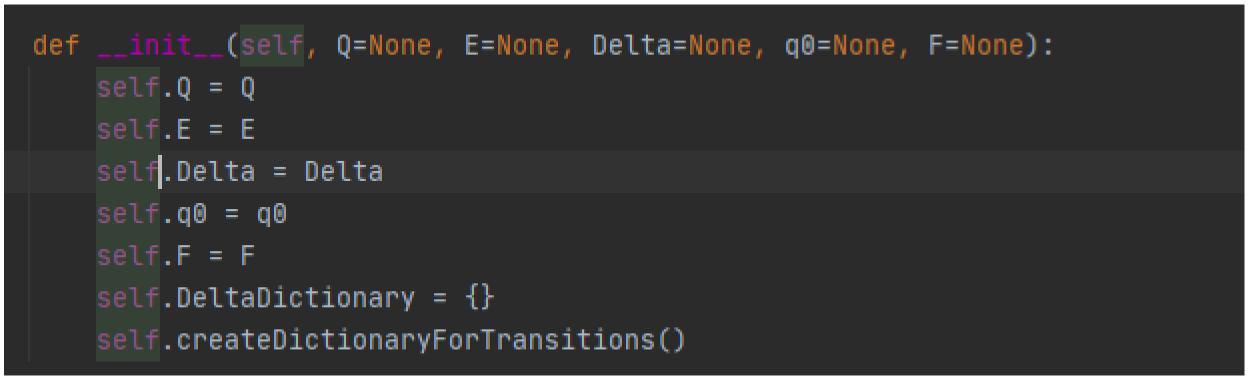
F = “”””State”””” { “|” “””” State “””” }

Initialization of the automaton happens in these lines, the user can choose any file it deems proper for an automaton.

file_name = input("Enter the name of the file containing the automata:")

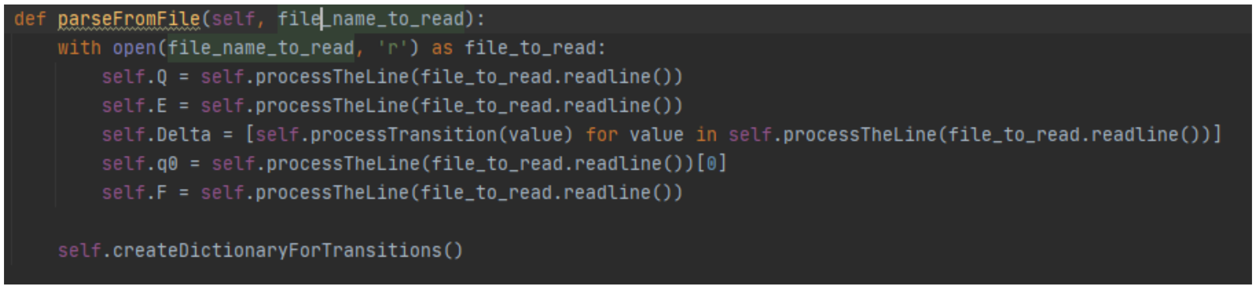
finite_automata.parseFromFile(file_name)

The user can initialize the automaton if he needs it, if either of the fields remain null the final method call will return without performing any computations



```
def __init__(self, Q=None, E=None, Delta=None, q0=None, F=None):
    self.Q = Q
    self.E = E
    self.Delta = Delta
    self.q0 = q0
    self.F = F
    self.DeltaDictionary = {}
    self.createDictionaryForTransitions()
```

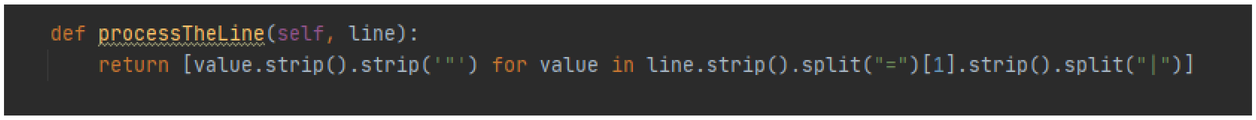
The parsing of the input file:



```
def parseFromFile(self, file_name_to_read):
    with open(file_name_to_read, 'r') as file_to_read:
        self.Q = self.processTheLine(file_to_read.readline())
        self.E = self.processTheLine(file_to_read.readline())
        self.Delta = [self.processTransition(value) for value in self.processTheLine(file_to_read.readline())]
        self.q0 = self.processTheLine(file_to_read.readline())[0]
        self.F = self.processTheLine(file_to_read.readline())

    self.createDictionaryForTransitions()
```

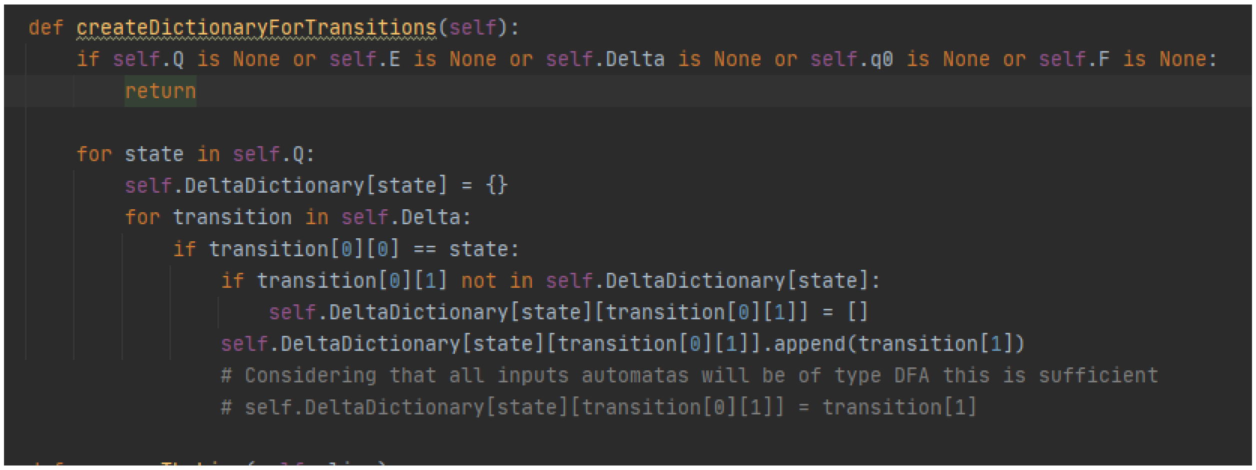
The line is processed as such, where the first symbol is split off and the rest are split by the "|" character, and stripped of they " characters from the end and the start so that it can be interpreted, I used EBNF in the file also:



```python
def processTheLine(self, line):
    return [value.strip().strip('"') for value in line.strip().split("=")[1].strip().split("|")]
```

Process transition behaves in a similar way, it will try to create a tuple of like ((a,b),c) to represent a transition.

This method is called after a proper initialization of the automaton

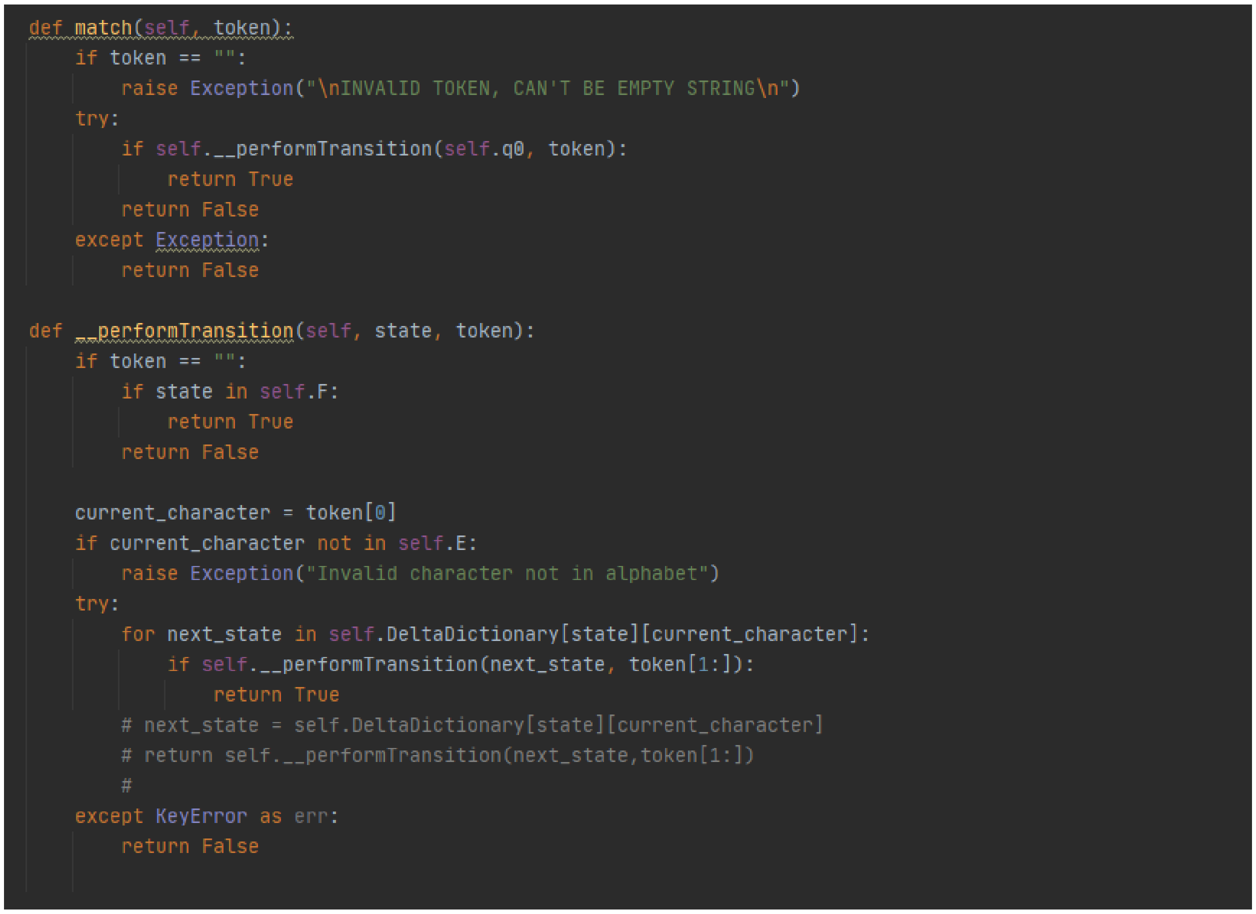


```python
def createDictionaryForTransitions(self):
    if self.Q is None or self.E is None or self.Delta is None or self.q0 is None or self.F is None:
        return

    for state in self.Q:
        self.DeltaDictionary[state] = {}
        for transition in self.Delta:
            if transition[0][0] == state:
                if transition[0][1] not in self.DeltaDictionary[state]:
                    self.DeltaDictionary[state][transition[0][1]] = []
                self.DeltaDictionary[state][transition[0][1]].append(transition[1])
                # Considering that all inputs automatas will be of type DFA this is sufficient
                # self.DeltaDictionary[state][transition[0][1]] = transition[1]
```

It will be a dictionary where each state will also have it's dictionary, which represents for a given input where does the automaton arrive. As it can be seen a second implementation is there for the case when we are sure that all inputs are DFA. The current one works also with NFA's.

This the match method will try to detect if the token is part of the language generated by the automata or not it will recursively call itself and tries to check every possible path until it finds a solution. If not solution is obtained then it will return False as the entered token is not part of the language generated by the automata



```python
def match(self, token):
    if token == "":
        raise Exception("\nINVALID TOKEN, CAN'T BE EMPTY STRING\n")
    try:
        if self.__performTransition(self.q0, token):
            return True
        return False
    except Exception:
        return False

def __performTransition(self, state, token):
    if token == "":
        if state in self.F:
            return True
        return False

    current_character = token[0]
    if current_character not in self.E:
        raise Exception("Invalid character not in alphabet")
    try:
        for next_state in self.DeltaDictionary[state][current_character]:
            if self.__performTransition(next_state, token[1:]):
                return True
        # next_state = self.DeltaDictionary[state][current_character]
        # return self.__performTransition(next_state,token[1:])
        #
    except KeyError as err:
        return False
```