<https://github.com/thediciman/FLCD/tree/master/Lab%204>

The finite automaton is represented as a class that contains 5 members:

* the set of states, which is a list of strings
* the set of symbols from the alphabet, which is a list of strings
* the initial state, which is a string
* the set of final states, which is a list of strings
* the set of transitions, which maps a pair of type (source, route) to a destination

This class has a static method that allows for a FA to be read from a file, which returns a FA created from the input file if it is valid, or throws an exception otherwise.

The structure of the file is the following:

* first line: the set of states, separated by space
* second line: the set of symbols from the alphabet, separated by space
* third line: the initial state
* fourth line: the set of final states, separated by space
* all of the remaining lines will contain a transition of the form (source\_state, route\_symbol) -> destination\_state

If we were to put this in BNF, it would have the following structure:

fa = states ‘\n’ alphabet ‘\n’ initial\_state ‘\n’ final\_states ‘\n’ transitions

states = state | state ‘ ‘ states

alphabet = symbol | symbol ‘ ‘ alphabet

initial\_state = state

final\_states = state | state ‘ ‘ final\_states

transitions = transition | transition ‘\n’ transitions

transition = ‘(‘ state ‘,’ symbol ‘)’ ‘->’ state

character = letter | digit

string = {character}

letter = uppercase\_letter | lowercase\_letter

uppercase\_letter = "A" | "B" | ... | "Z"

lowercase\_letter = "a" | "b" | ... | "z"

digit = “0” | "1" | ... | "9"

state = string

symbol = string

The FiniteAutomaton class also has the following methods:

* a method that checks if the FA is a DFA, which checks that for all destination states, there is no more than one route that goes to it.
* a method that checks if a sequence is accepted by the FA

Example input file:

s a b k  
0 1  
s  
s a k  
(s, 0) -> s  
(s, 1) -> a  
(a, 1) -> s  
(a, 0) -> b  
(b, 0) -> k

The Scanner from the previous labs has been integrated with the Finite Automaton developed now, and the FA is used to check whether a token is a constant or an identifier, by checking if the token is accepted by the FA.

The FA for identifiers contains two states q0 and q1, where q0 is the initial state and q1 is the final state, and the alphabet is made up of all english letters, uppercase and lowercase, and digits from 0 to 9.

The transitions are as following: from q0 to q1 via all letters from the alphabet, and from q1 to q1 via all symbols from the alphabet. This respects the language specification that an identifier starts with a letter, and then it can contain any number of letters or digits.

The FA for constants has been made in such a way to allow for multiple constant classes – integers, characters and strings.

It contains the states S, S\_S, S\_E, C\_S, C\_I, C\_E, I\_S, I\_I1, I\_I2, I\_E, where the ones ending with S are “starting” states for each constant class, the ones with E are “ending” states and the ones that contain I are “intermediary” states.

The initial state is S, the final states are S\_E, C\_E, I\_E, I\_I2, and the alphabet contains all uppercase and lowercase letters, digits from 0 to 9, and the characters +, -, ‘, “.

For string constants:

* from S to S\_S via “, because the string constant begins with a quote
* from S\_S to S\_S via any letter or digit, because the string constant can contain any of these between quotes, and it can loop here any number of times
* from S\_S to S\_E via “, because the string constant ends with a quote

For character constants:

* from S to C\_S via ‘, because the char const begins with a single quote
* from C\_S to C\_I via any uppercase or lowercase letter or any digit, and this allows for a single character between the single quotes
* from C\_I to C\_E via ‘, so the char const is ending with a single quote

For integer constants:

* from S to I\_E via 0, so we allow for the int const 0
* from S to I\_1 via + or -, so we allow for int consts starting with + or –
* from I\_1 to I\_2 via any non-zero digit, so we don’t allow for int consts that start with + or – followed by a 0
* from S to I\_2 via any non-zero digit, so we have int consts that do not start with +, - or 0
* from I\_2 to I\_2 via any digit, so it can loop any number of times for arbitrary length integers

The string, character and integer constants have been created in such a way to respect the language specification.