Regular Expressions & Finite State Machines

1. Regular Expressions (Regex)

Regular expressions are strings of characters that represent patterns. They are built based on a grammar, just like a programming language. These strings are used to "recognize" and manipulate strings. Analogous to arithmetic expressions, a regular expression is built by combining smaller expressions with the help of operators.

The following metacharacters used in the construction of a regular expression:

^ - used to search for a specific pattern at the beginning of the line. Example: "^abc" will match "abc", "abcde", "abc f123c", but will not match "eabc" because abc does not is at the beginning of the line

$ - used to search for a specific pattern at the end of the line. Example: "abc$" will match with "abc", "deabc", "f123c abc", but will not match "abcde" because abc is not at the end of the line

. - matches any character. Example: "a.c" will match "abc", "adc", "a0c", but it won't match "abbbc". Notice that since I didn't put start or end anchors, the expression can also match "edabcde".

[] - matches a single character, but that character will be inside the square brackets. Example: "grade [789]" will match "grade 9", "I got grade 7", but it will not match " grade 4" or " grade 789". Another useful element is the hyphen (-) that can be used to specify ranges (considering the ASCII order of the characters). For example, if we want to write an expression that will match any lowercase letter, we can write [a-z], if we want to write an expression that will match any lowercase, uppercase letter or number we write [a-zA-Z0-9].

* - matches 0 or more times. Example: if we write "ab*c" it will match with "ac", "abc", "abbbbbbbc", "12345abbbc54321".

+ - matches 1 or more times. Example: if we write "ab+c" it will match with ”abc”, ”abbbbbbbc”, ”12345abbbc54321”, but it will no longer match ”ac”.

? - matches 0 or 1 times. Example: if we write "ab?c" it will match "ac", "abc", but it will not match "abbbbbbbc", "12345abbbc54321".

{,} - allows us to specify a minimum and maximum number of repetitions for a pattern. Example: [a-z]{2,5} will match ”ab”, ”abc”, ”abcd”, ”abcde”, but will not match with "a" or with "abcdef".

() - used for grouping regular expressions. For example, (ab)* will match "abab", "ab", "abababababab".

| - will match one of the patterns to its left of right. For example, “(positive|negative) ion” will match either "positive ion" or "negative ion".

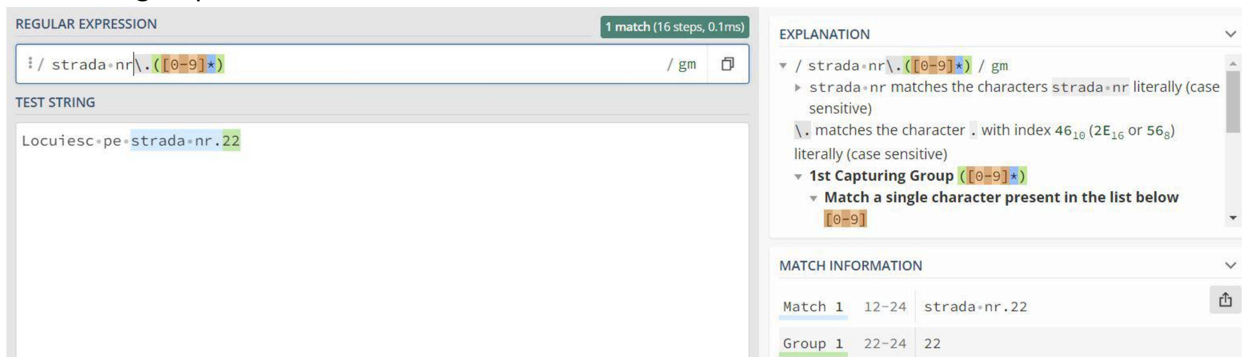
\ - in case we want a special character to not be interpreted in a special way (if we want to use \ or . inside the expression as a character). For example, we saw above that . matches any character, but if I want it not to be interpreted (I want to use the character .), I can write this: "book\.". If I want to write the $ symbol (without its meaning as the final anchor), I can proceed similarly. Example: ”\$[0-9]*” matches "$300", "I won $1000 yesterday" ($ is not interpreted as a final anchor, because I put \ in front of him in the regular expression).

Regex101

You can access an online interpreter for regular expressions at this link:
https://regex101.com/
In the field named REGULAR EXPRESSION you enter the regular expression written by you, and in the field named TEST STRING you write several examples of words, each on a different line. The words that will be underlined will match with the regular expression entered by you.

Other uses for Regex

Another important advantage of regular expressions is its ability to extract information, aside from finding patterns. For example, if I define a regular expression "strada nr\.([0-9]*)" I have defined a group for the street number:



We can observe in the bottom right that next to matching, in the Group 1 section I have extracted the street number with the help of the group we defined.

Examples

Example 1

Write a regular expression for words that contain only the characters 0 and 1 (may appear in any order and combination).

Solution:

”^(0|1)*$” sau ”^[01]*$”

Warning: If we hadn't put ^ and $, our regular expression would have matched strings of the form "abc 01 abc", which would have been wrong, because this string contains s, and other characters apart from 0 and 1.

Example 2

Write a regular expression for words that contain only the characters 0 and 1 and end with with 1.

Solution:

”^(0|1)*1$” sau ”^[01]*1$”

Example 3

Write a regular expression for the words that contain only the characters a and b and contain the substring bab.

Solution:

”^(a|b)*bab(a|b)*$” sau ”^[ab]*bab[ab]*$”

Example 4

Write a regular expression for the words that contain only the characters a and b and contain the substring bab.

Solution:

”^(a|b)*bab(a|b)*$” sau ”^[ab]*bab[ab]*$”

Example 5

Write a regular expression for the words that contain only the characters a and b and can contain as many b's, but they must be delimited by 2 a's.

Accepted examples:

aba, abbbbbbba, abaaba, abbbaabaabba

Solution:

”^(ab*a)*$”

Example 6

Write a regular expression for the words that contain only the characters a and b and have the length divisible by 2.

Accepted examples:

aba, abbbbbbba, abaaba, abbbaabaabba

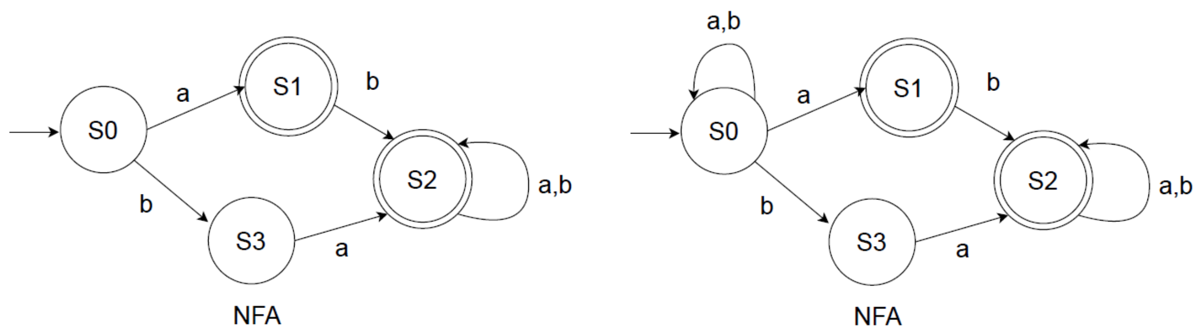
Solution:

”^([ab][ab])*$”

Finite State Machines (FSM)

A finite state machine is defined by its input symbols (formally Σ - the input symbols, e.g. characters or pressing a button), states (formally S – set of states), transitions or rules for passing from one state to another (formally a function $\delta : S \times \Sigma \rightarrow S$), initial state (formally $s_0 \in S$), set of accepting states (formally $F \subseteq S$, so an automaton can have several final states). Formally we can define the automaton as a tuple with the 5 elements mentioned above $(\Sigma, S, s_0, \delta, F)$.

DFA = Deterministic Finite Automaton: can have only one active state at a time
NFA = Nondeterministic Finite Automaton: can have several active states at the same time
Every DFA is NFA, but not vice versa.
Every NFA can be converted to DFA.

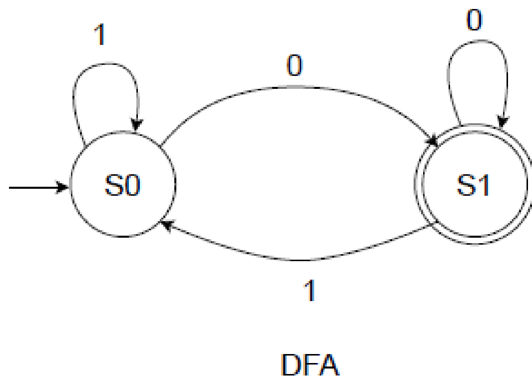


In the above example, we notice that the initial state is S0, and both automata have accepting states S1 and S2. The second automaton is of the NFA type because from state S0 it can stay in state S0 or go to state S1, so the automaton can follow several paths (unlike the example in the first figure). An NFA will accept the input if at least one accepting state is active.

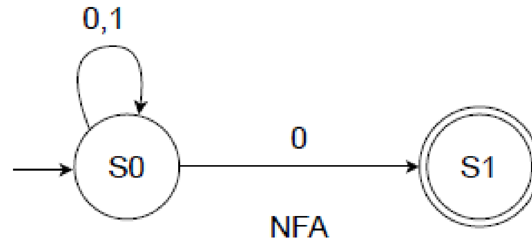
For quickly creating and testing state machines, you can use **Automaton Simulator**:
https://automatonsimulator.com/

Example 1

Considering the set of symbols Σ = {0, 1}, create an FSM that accepts the words which end with 0.



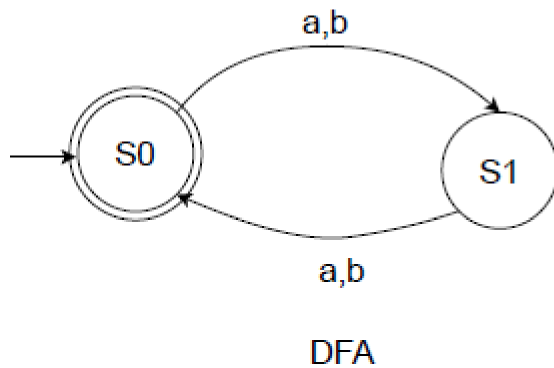
(c) DFA (ex.1)



(d) NFA (ex.1)

Example 2

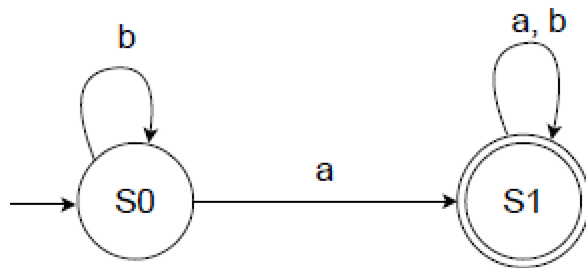
Considering the set of symbols Σ = {a, b}, create an FSM that accepts the words with the length divisible by 2.



(e) Ex.2

Example 3

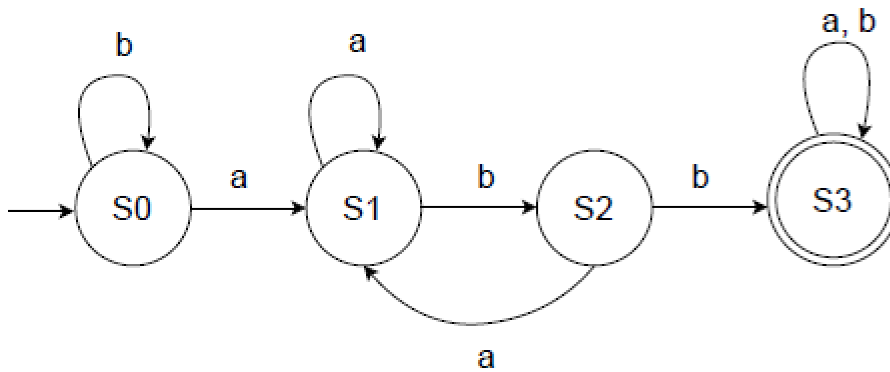
Considering the set of symbols Σ = {a, b}, create an FSM that accepts the words which contain the letter a (must necessarily include an a).



(f) Ex.3

Example 4

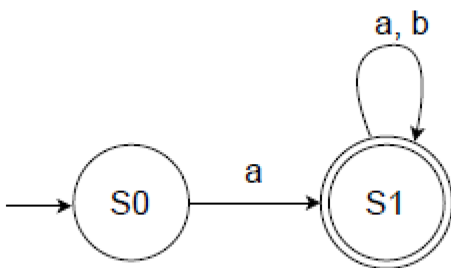
Considering the set of symbols Σ = {a, b}, create an FSM that accepts the words which contain the substring abb.



(g) Ex.4

Example 5

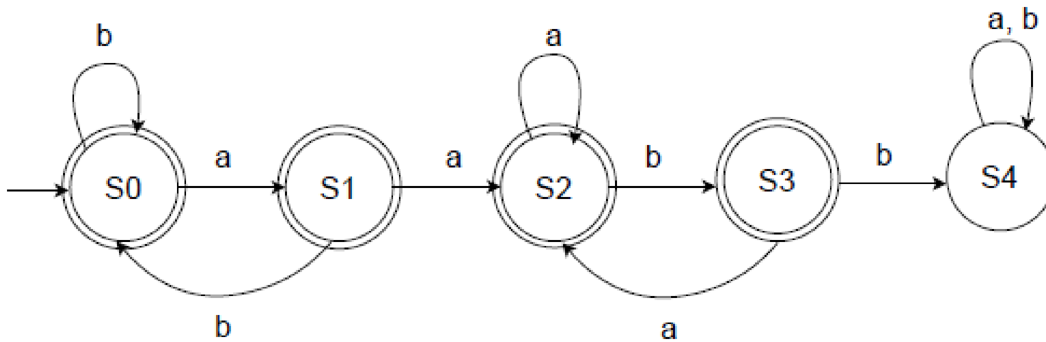
Considering the set of symbols Σ = {a, b}, create an FSM that accepts the words that begin with a.



(h) Ex.5

Example 6

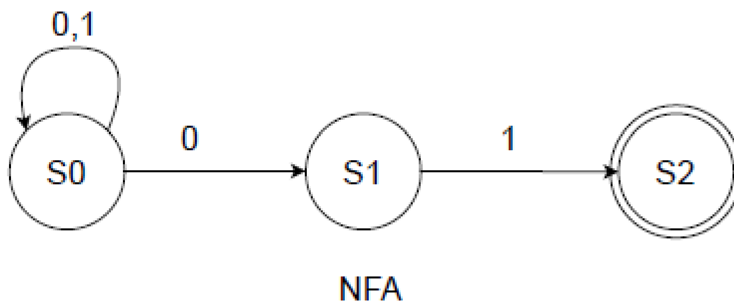
Considering the set of symbols Σ = {a, b}, create an FSM that accepts the words in which if bb appears, then aa did not appear before. (so in which we will not accept words that contain the substring aabb)



(i) Ex.6

Converting NFA-DFA

Convert the following FSM from NFA to DFA:



NFA

Solution:

We build a table like this: we start from the initial state and for each symbol we will write the set of states that can be reached. Every time we find a new set (written it below in red), add a new line to the table and repeat the operations. Every newly obtained set will become a new state in the DFA. The table corresponding to the conversion of the above example, written step by step, is:

state	0	1
S0	{S0,S1}	S0

Step 1

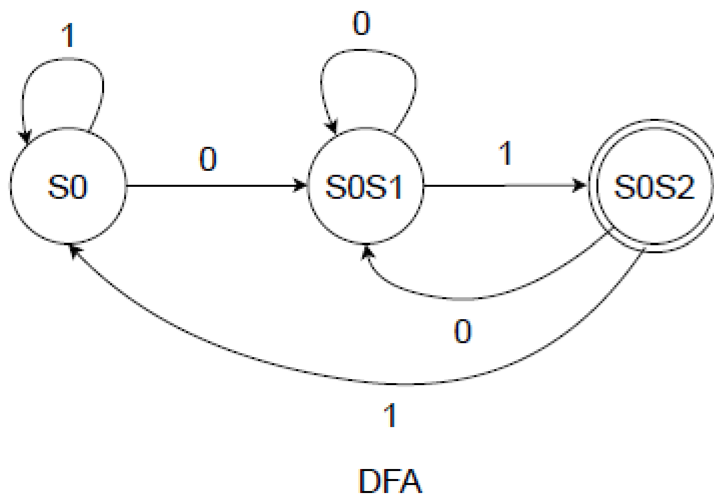
state	0	1
S0	{S0,S1}	S0
{S0,S1}	{S0,S1}	{S0,S2}

Step 2

state	0	1
S0	{S0,S1}	S0
{S0,S1}	{S0,S1}	{S0,S1}
{S0,S2}	{S0,S1}	S0

Step 3

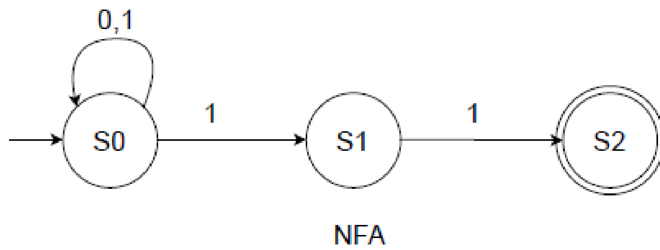
At step 3, no new states have been obtained, so we'll stop there. The initial state will still be S0. The acceptor states in the obtained DFA are the states that contain at least an acceptor state from the initial FSM (in our case, since S2 was the accepting state, then S0S2 will be the accepting state). Based on the transition table, we create the DFA:



DFA

Example 2

Convert the following FSM from NFA to DFA:

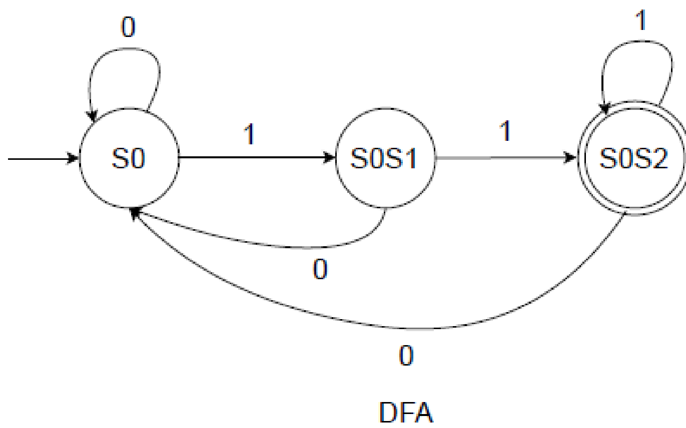


NFA

Solution:

We build the table in a similar way to the one from the previous exercise, and based on it, we create the DFA.

state	0	1
S0	S0	{S0,S1}
{S0,S1}	S0	{S0,S1,S2}
{S0,S1,S2}	S0	{S0,S1,S2}



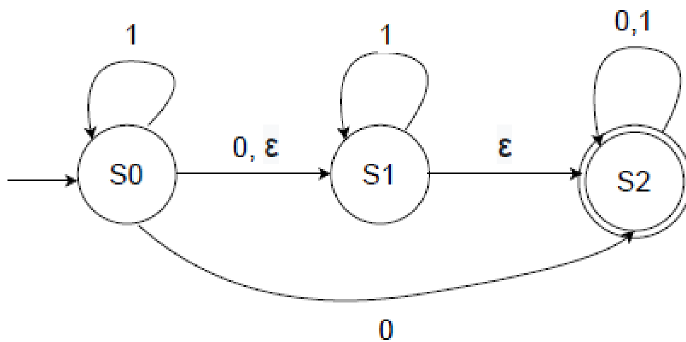
DFA

ε transition

A ε transition (empty string) does not consume a symbol:

Example 3

Convert the following FSM from NFA with ε transitions to DFA:



Solution:

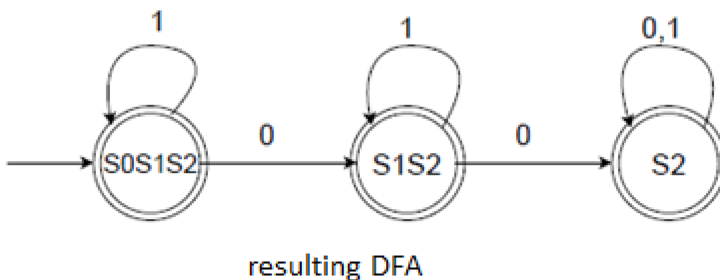
We build the transition table for the FSM, but also take into account the fact that we have ε transitions that can be realized without consuming symbols.

state	0	1
S0	{S1,S2}	{S0,S1,S2}
S1	S2	{S1,S2}
S2	S2	S2

For the new 3 sets of states that appeared (these will be the states in the DFA) we create the table of transitions:

state	0	1
{S0,S1,S2}	{S1,S2}	{S0,S1,S2}
{S1,S2}	S2	{S1,S2}
S2	S2	S2

Since all 3 of them contain S2 (which was the accepting state), all 3 will be accepting states in the DFA. The initial state will be S0S1S2.



resulting DFA

Homework:

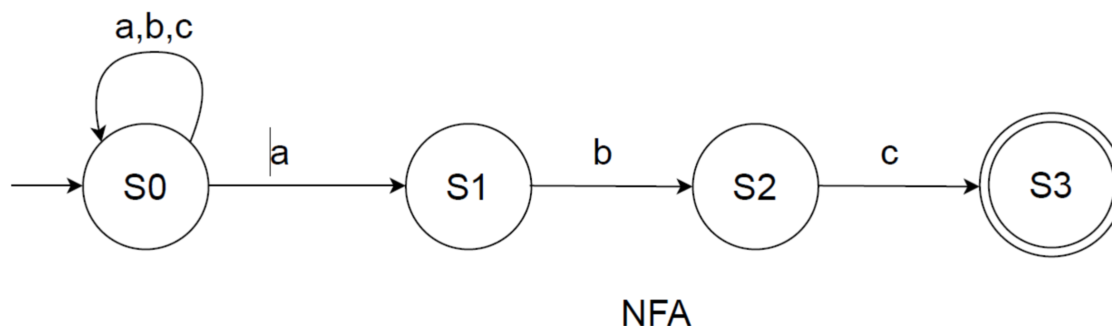
Note regarding uploading the homework: For this homework, upload a single file (.doc, .docx, .pdf, etc.). If you solve it on a sheet of paper, scan the sheets and upload the pdf (don't upload pictures).

Homework exercise 1:

a) Create an FSM that accepts words containing a, b or c and containing the substrings ac or bc, followed by the substring bac. Write a regular expression for the same type of words.
b) How would you modify the FSM to accept any word containing a, b, or c, but without containing the substrings ac or bc followed by bac.

Homework exercise 2:

Convert the following FSM from NFA to DFA:



NFA

Homework exercise 3 (Optional):

Write a regular expression that can be used to extract flight data from an airport. We consider that they will be of the form: City1 - City2: DepartureDate. City name starts with a capital letter, then it can contain any number of lowercase letters. The date will be of the form DD.MM.YYYY (where D,M,Y are numbers).
Example of a string that will be accepted:
"Timisoara - Barcelona: 25.12.2022"
Example of a string that will not be accepted:
"Timisoara - Barcelona: 250.120.20212"

Optionally, you can try to improve the validation for the date (not to accept the date 31.31.30000, for example)