There are three approaches to building a lexical analyzer

- Write a formal description of the token patterns of the language using a descriptive language related to regular expressions.

  These descriptions are used as inputs to a software tool that automatically generates a lexical analyzer.

  example of such tools:

  Generates Lexical Analyzer

  - Lex: Included in UNIX system
  - Flex: Free and open-source alternative of lex

- Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram.

  → One can use C-language to do this job.

- Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram.

State Transition Diagram

- known as state diagram and it is a directed graph
- Nodes of a state diagram are state names
- Arcs are labeled with characters that cause transition between the states.

- Represented using mathematical machines called finite automata

Finite automata can be designed to recognize regular languages

Tokens of programming languages are regular languages, and the lexical analyzer is finite automata

So, in short, a lexical analyzer does the following:

- Extracts lexemes from a given input string and produce the corresponding tokens
- It is now implemented as subprogram, and is called by the syntax analyzer.

  ↳ in each call it returns a single lexeme and the corresponding token

  ↓

  So, a syntax analyzer sees the the output of a lexical analyzer

- Hexical-analysis skips comments and white space.

  ↓ not relevant to the meaning of the program

- It also detect syntactic errors in tokens. For example, ill-formed floating-point literals

  ↓

  5.3876e4 ≡ 53876

  4e-11 ≡ 0.00000000004

LANGUAGE RECOGNITION

Finite-state automata can recognize language as evident from the previous discussion. A valid question is—

What sets can be recognized by these Finite-state automata or machines?

↓ Answer

Solved by mathematician Stephen Kleene [A simple characterization of the sets can be recognized by finite state automata.

↳ Precisely,

A set is recognized if this set can be built up from the null set
empty string
singleton ~~set~~ string

by taking

operations [concatenations, unions, and kleen closures

the above string set

So, Performing these operations on build up a set, known as, regular set

↓

This terminology is very similar to the term REGULAR GRAMMAR

Regular Sets:

Regular sets are those that can be formed using the operations of

- concatenation
- Union
- Kleene closure

in arbitrary order, starting with the

- empty set
- empty string, and
- singleton sets

To define regular sets, we first need to define regular expressions—

Regular expressions over a set I are defined recursively by:

- $\emptyset$ is a regular expression
- λ is a regular expression
- x is a regular expression whenever $x \in I$

AB, $(A+B)$ and A^* are regular expressions whenever $A \in I$ and $B \in I$

(AB: concatenation; $(A+B)$: Union; A^*: Kleene closure)

Each regular expression represents a set specified by these rules:

- $\emptyset$ represents the empty set
  ↳ set with no strings
- λ represents the set $\{\lambda\}$, which is the set containing the empty string.
- $\{\lambda\}$: Also, a singleton set

- x represents the set $\{x\}$ containing the string with one symbol x
- (AB) represents the concatenation of sets A and B
- $(A+B)$ represents the union of the sets represented by A and B.
- A^* is the Kleene closure of the set A.

So, sets represented by regular expressions are called regular sets.

⊞ Example: Regular sets and the corresponding Regular expression:

ab^*	Single a followed by any number of bs
$(ab)^*$	any number of copies of ab, including null string
$b+ba$	String b or the string ba
$b(b+a)^*$	Any string beginning with b
$(b^*a)^*$	Any string not ending with b

⊞ Example: bit string

Set of bit strings with even length
- length multiple of two
- Zero length

Two bit strings are 00, 01, 11, 10.

So, concatenation among any of these strings will generate strings with even length

Thus, the set is

$$\underbrace{(00 + 01 + 10 + 11)^*}_{\text{zero or more repetition}}$$

(union)

↳ concatenated form

- Set of bit strings ending with a '0' and they do not contain 11

Answer: 11 means two consequitive 1s.
ending with zero is 10, or ,
it could be 0 itself

So, $\underbrace{(0 + 10)^*}_{\text{zero times or more times in concatenated form}} (0 + 10)$

- Observation

Often we see that we see alternative paths to recognize the language. That is, existence of "Union" in the regular language which is suggestive of a non-deterministic transition to the next step.

Regular Grammar

Partly From cs.odu.edu

Regular grammar describes a regular language.

- Right regular grammar
- Left regular grammar

Formally,

a grammar consists of a

- set of terminals Σ (alphabet)
- set of nonterminals V (variables)
- start symbol S (non terminal)

and a set of rewrite rules (Production Rules) of the form

General form $\gamma \rightarrow \alpha$

where, γ could be a string of terminals and nonterminals but there must be at least one non-terminal,

α is strings of terminals and non-terminals

Example: Let's say a language is $\{a, aa, aaa, \ldots\}$

Production rules:

$S \rightarrow aS$

$S \rightarrow a$

so,

$S \rightarrow aS$
$\rightarrow aaS$
$\rightarrow aaaS$
$\rightarrow aaaa$

Now, A grammar is regular if and only if γ is a single nonterminal and α is a single terminal or a single terminal followed by a single nonterminal

Right Regular Grammar

↳ Definition depends largely on the format of production rules.

forms of grammar Rule allowed

$S \to a$,
$S \to aP$
$S \to \epsilon$

where a is terminal, S, P are non terminal and ϵ is empty string.

↳ this is non-terminal

This grammar is also known as Right Linear Grammar

Left-Regular Grammar

Allowed production rules are of the form

$S \to a$
$S \to Pa$
$S \to \epsilon$

where. a is terminal, S, P are non terminal and ϵ is empty string.

Nondeterministic Finite Automata

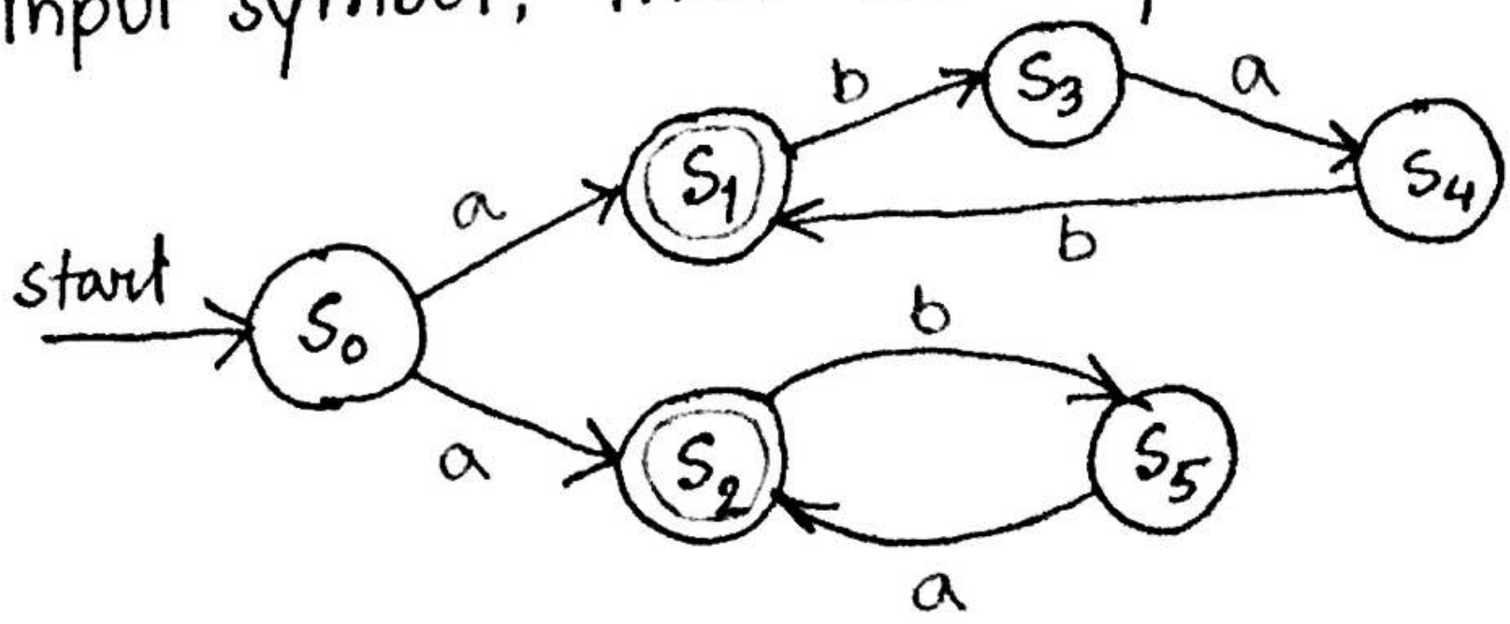
For the deterministic finite automata (DFA), we have known that

> Each transition from current state to the next state is uniquely determined by the current state and input
>
> Also, input character is essential for each state transition.

But, a nondeterministic finite automata (NFA) does not need to obey these restrictions.

Precisely, in NFA, for each state there can be zero, one, two, or more transitions corresponding to the particular input symbol

If NFA reach to a state with more than one possible transition corresponding to the input symbol, then we say that it branches



The above NFA recognizes

$$a(bab)^* \underset{\text{or}}{\cup} a\underbrace{(ba)^*}_{+}$$

0 or more repetition

So, $a(bab)^* \cup a\underbrace{(ba)^*}$
↳ 0 or more repetition

so, recognized strings would be

a, a bab, a babbab, a bab bab bab ... ...

In short,

In case of deterministic finite automata,

for each pair of (current) state and input value, there is a unique next state given by the transition function.

Whereas,

In non-deterministic finite-state automata.

for each pair of state and input value, there may be several possible next states.

↓

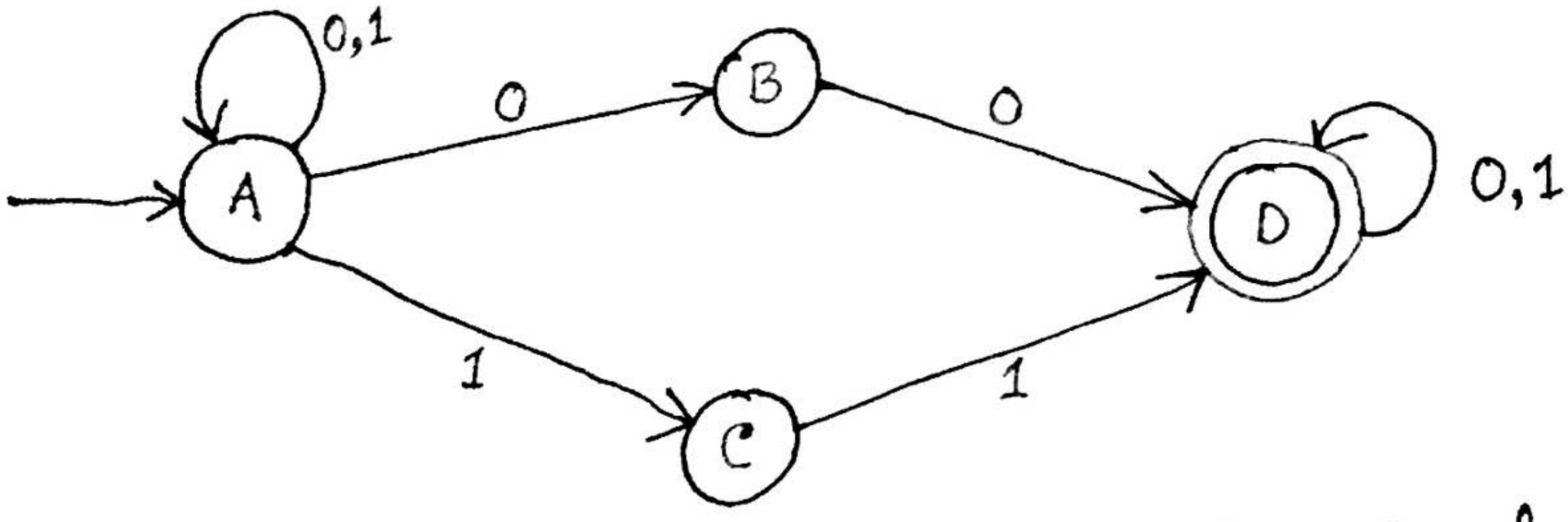
Provides flexiblity & the state-transition diagram is less complex. Overall, it is important in determining the language a specific finite-automata can recognize.

Definition:

An NFA $M=(S, I, f, s_0, F)$ consists of a set S of states, an input alphabet I, a transition fn^c f, starting state s_0, and the final state F. Here,

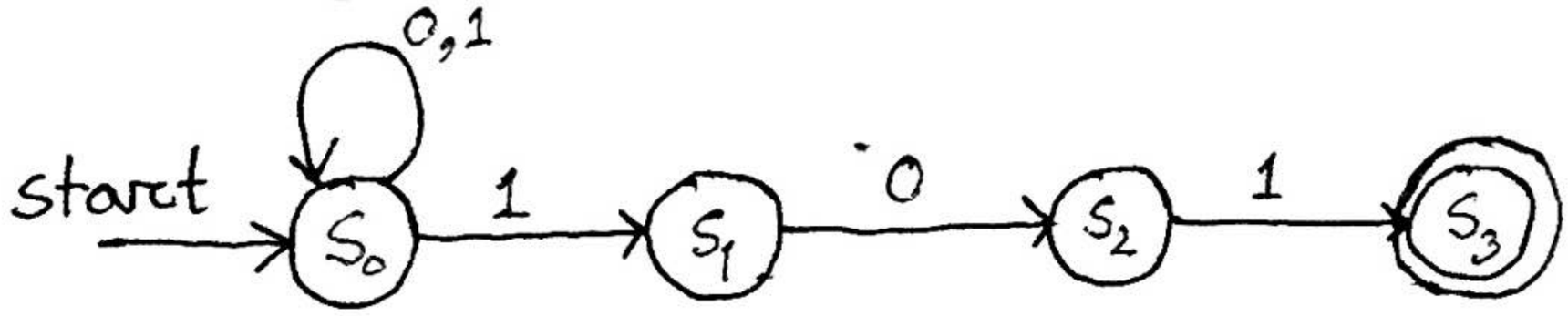
Transition fn^c $f : S \times I \rightarrow \underbrace{P(S)}_{\text{set of states}}$

NFA : Example



The above NFA accepts any string that contains 00 or 11

Design an NFA that accepts all binary strings that end with 101



Example

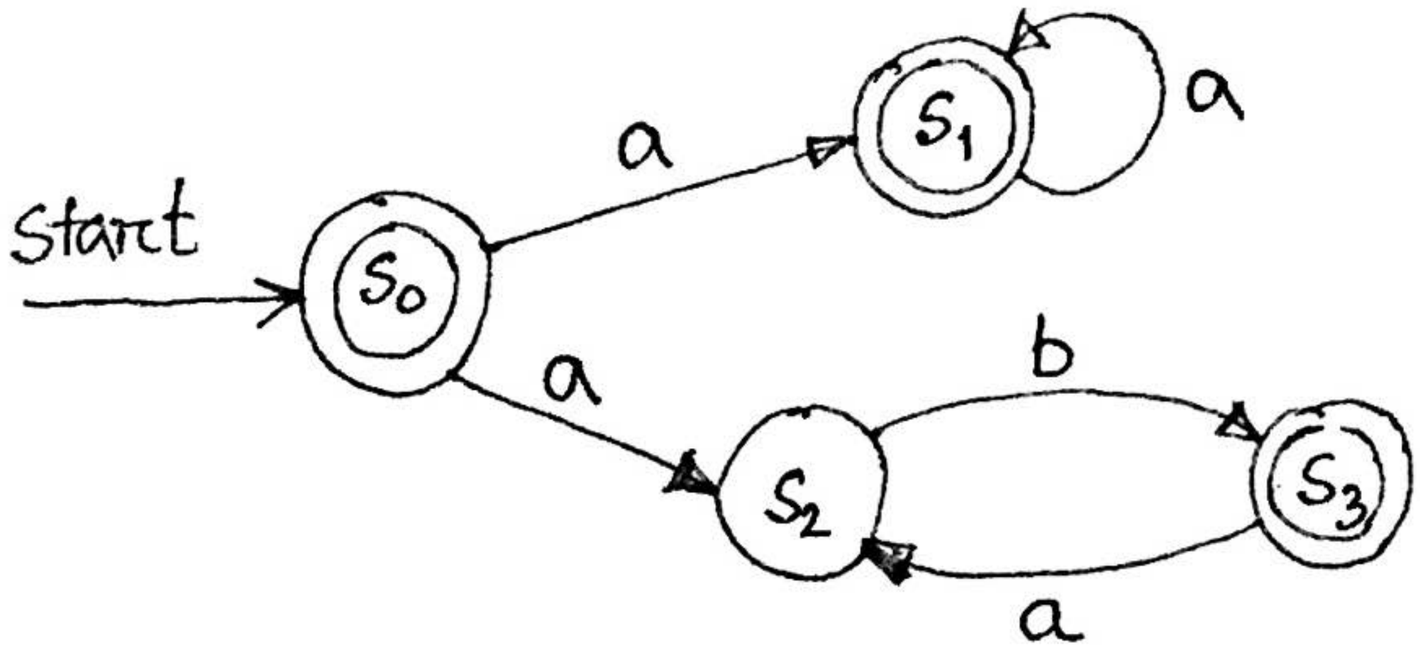
000 101

010 010 101

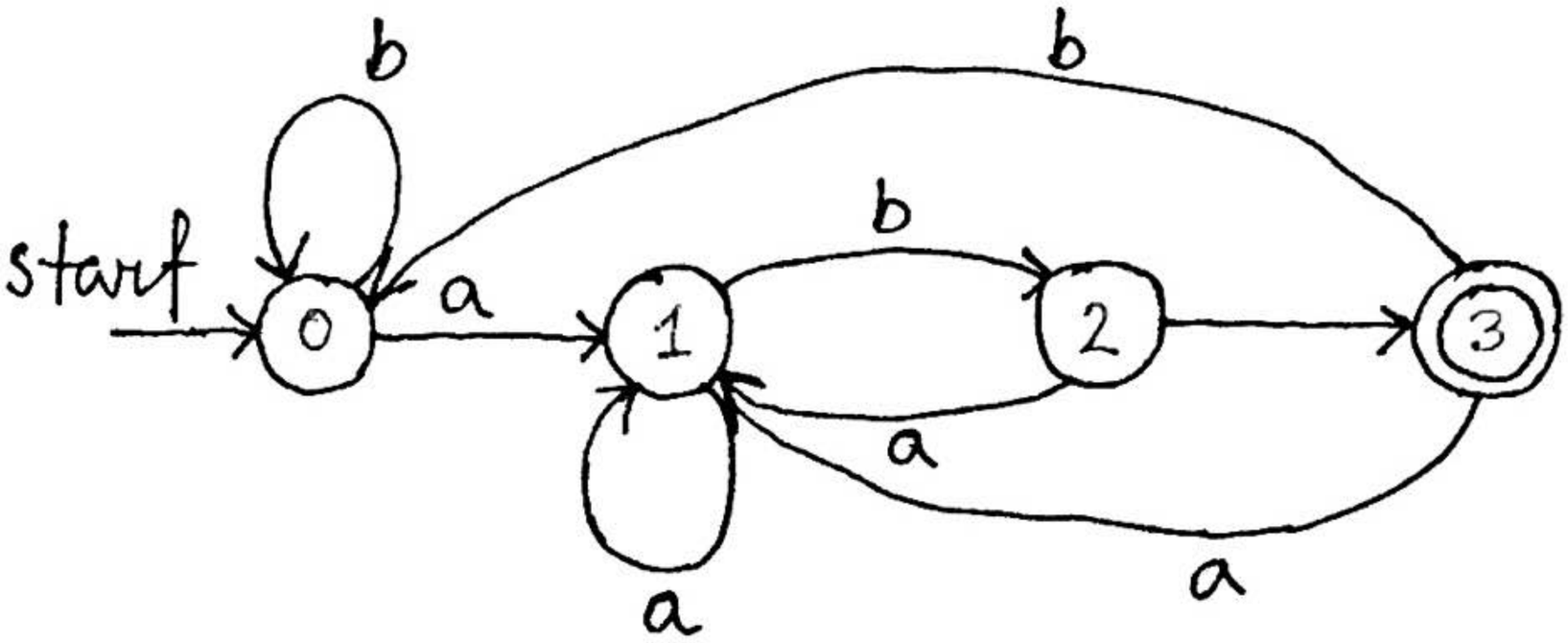
We reach to accepting state

Draw the NFA to recognize simultaneous pattern —

$a^* + (ab)^*$ [it includes null string]



Example of a state transition diagram

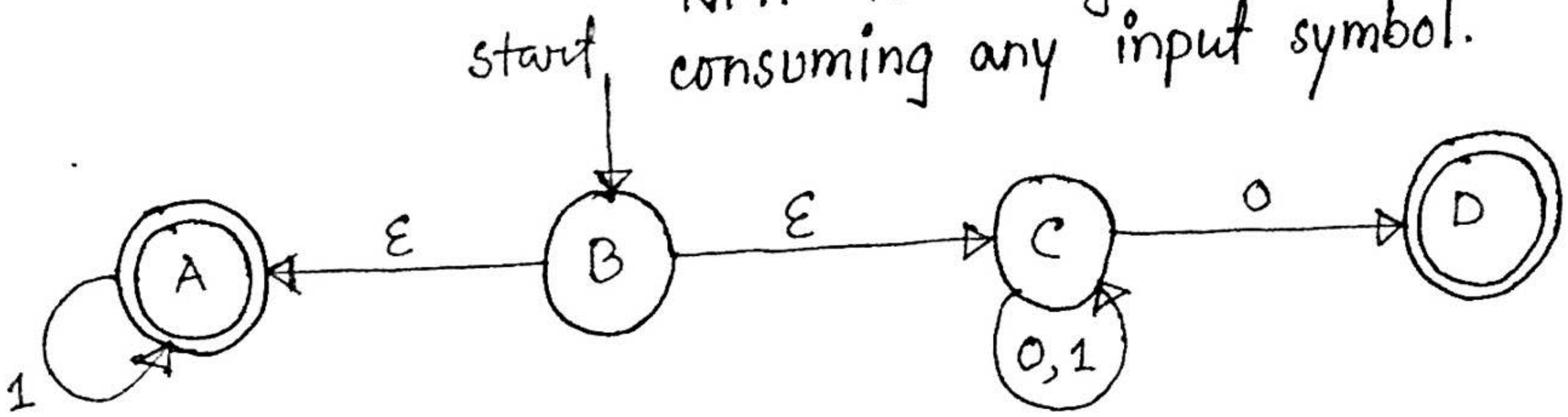


- Precisely, this is a Deterministic Finite Automata (DFA)
- The above diagram recognizes strings of the form $(a|b)^* abb$

ε-transition

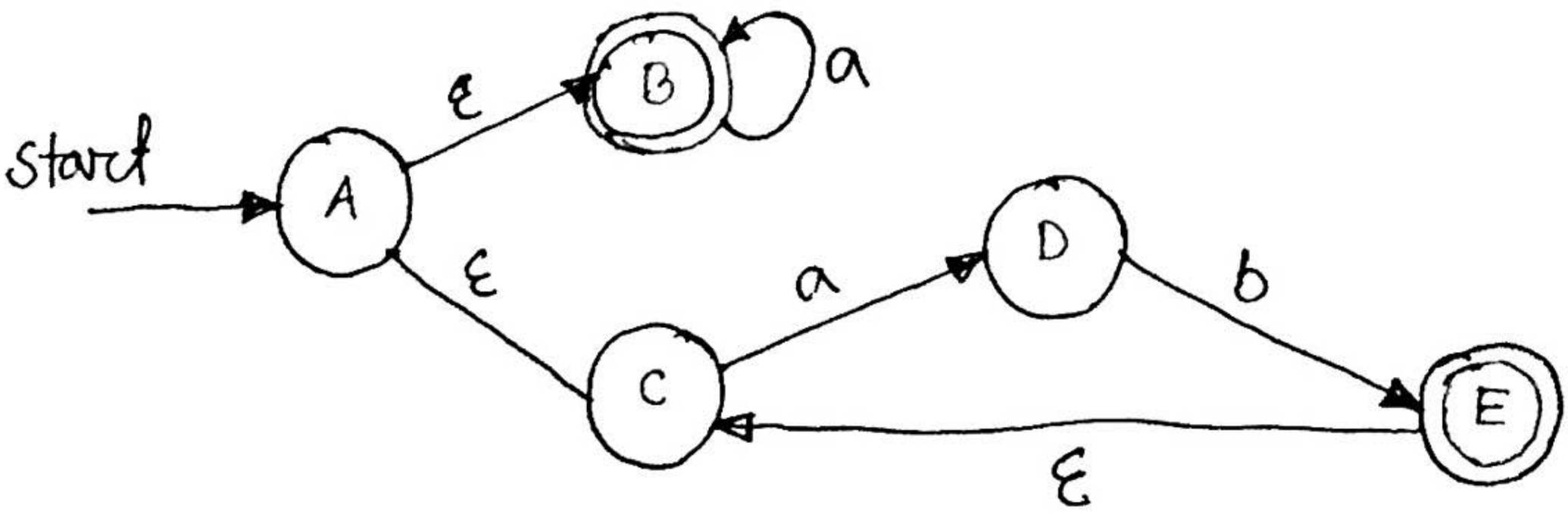
↳ arrows labeled with empty string

This sort of transition allows NFA to change state without consuming any input symbol.

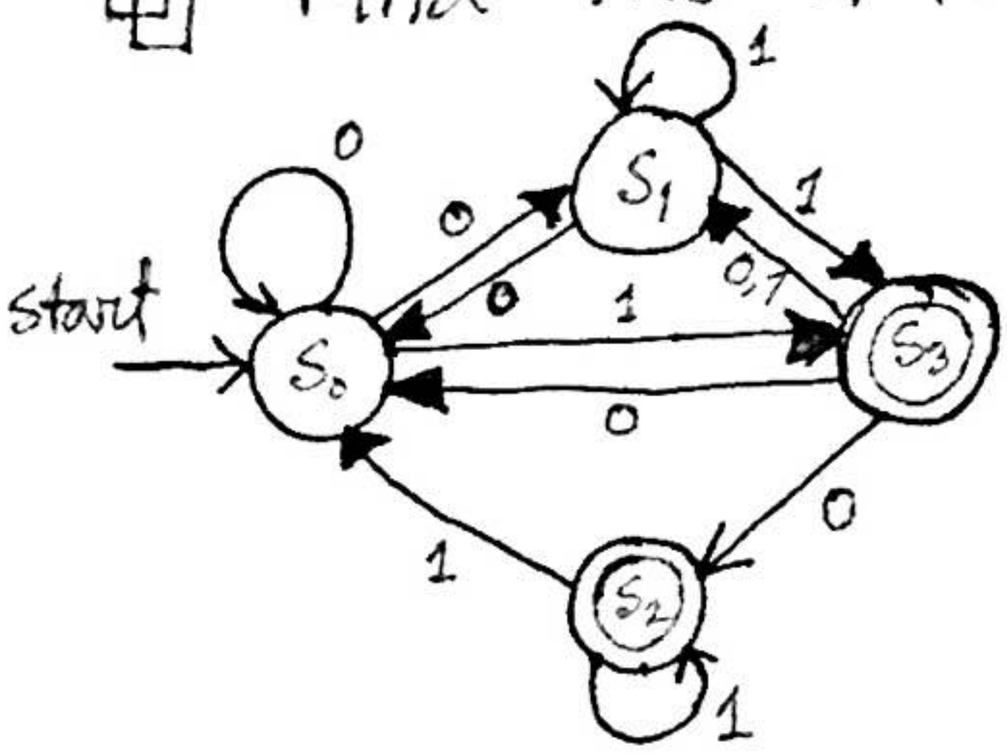


The above NFA recognizes all binary strings where the last symbol is 0 or that contain only 1's

Another example: Recognize $a^* + (ab)^*$ using NFA with ε-transition



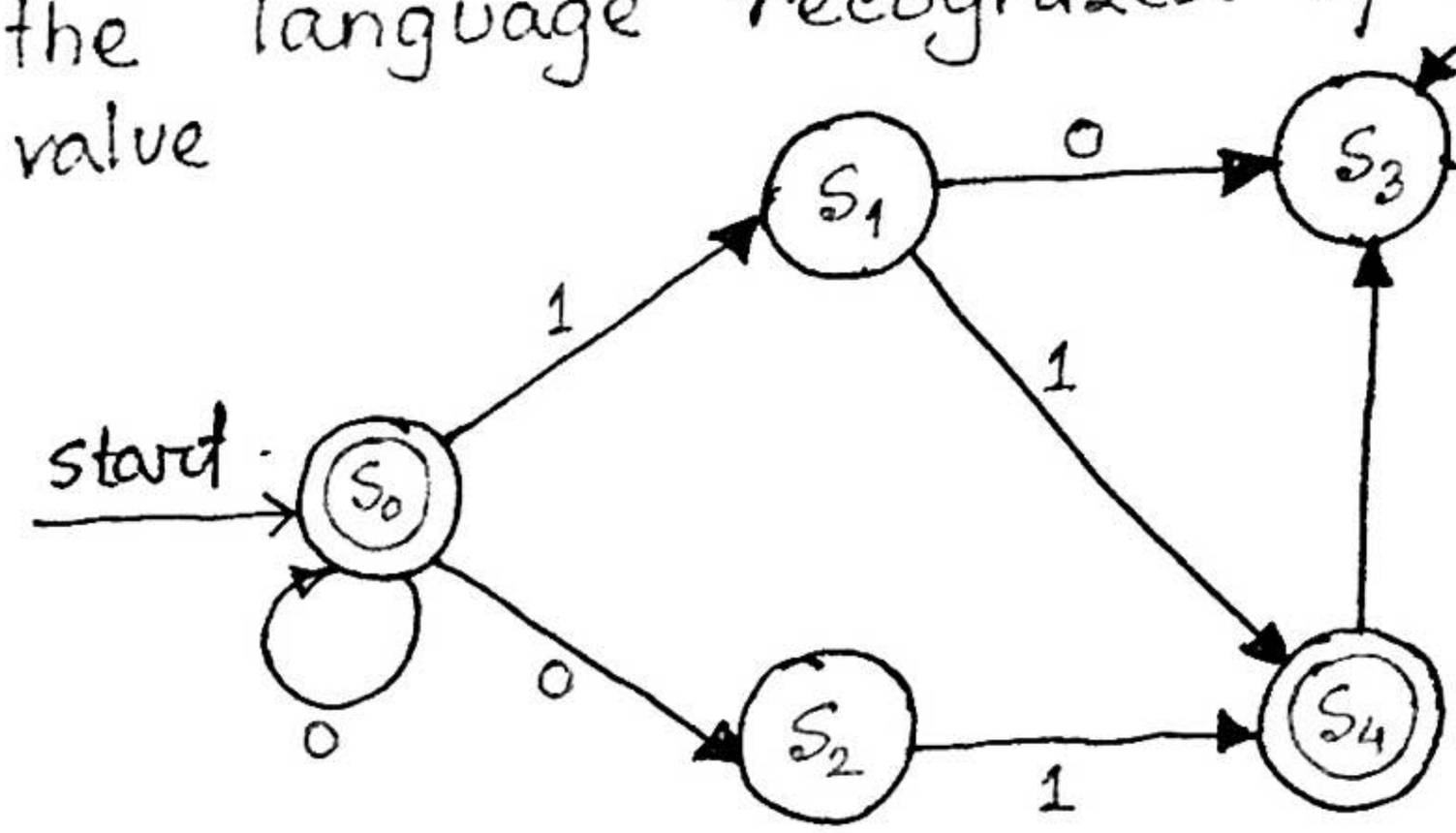
Find the state-table of the given NFA



State-trasition Table

State	input 0	input 1
S_0	S_0, S_1	S_3
S_1	S_0	S_1, S_3
S_2	—	S_2, S_2
S_3	S_0, S_1, S_2	S_1

Find the language recognized by the NFA given value



Language: $\{0^n, 0^n01, 0^n11 \mid n \geq 0\}$ Null could be recognized as well.

↓ corresponding DFA

Transition Table

	input 0	input 1
S_0 →	S_0, S_2	S_1
S_1	S_3	S_4
S_2		S_4
S_3	S_3	
S_4	S_3	S_3

Important :

Language recognized by a NFA can also be recognized by DFA.

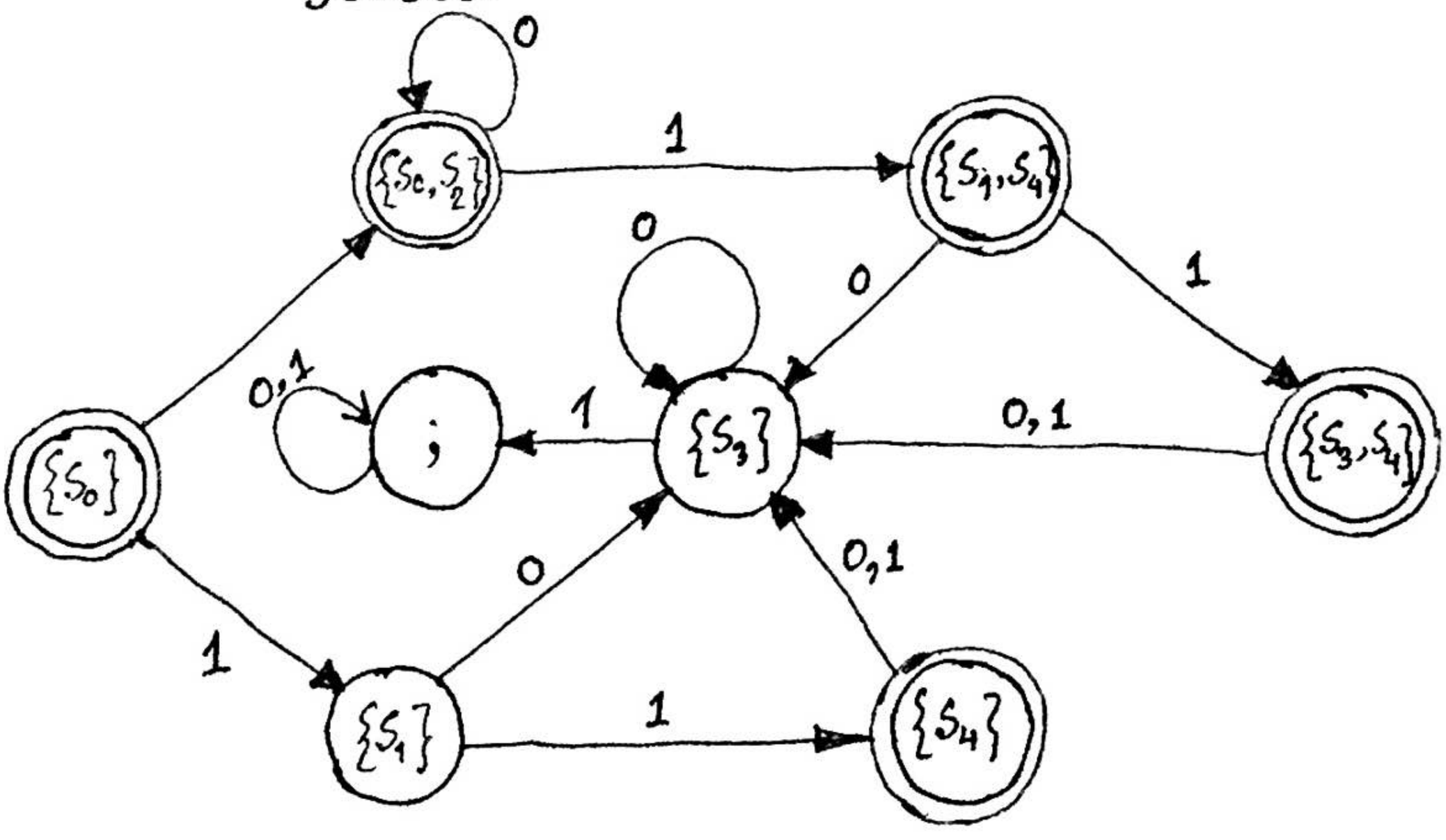
↓

Every NFA can be converted to DFA and they do recognize same regular expression.

DFA from NFA

- states of the deterministic FA are the subsets of the set of all ~~sets~~ states of the NFA.

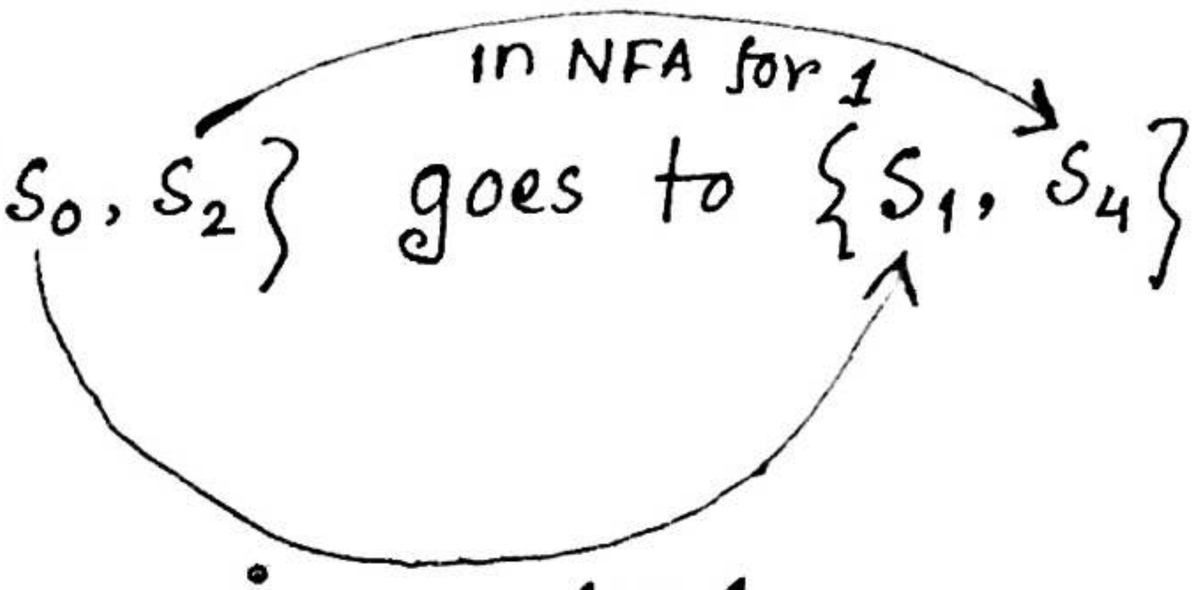
- Next state of a subset under an input symbol is the subset containing the next states in the NFA machine of elements in this subset.



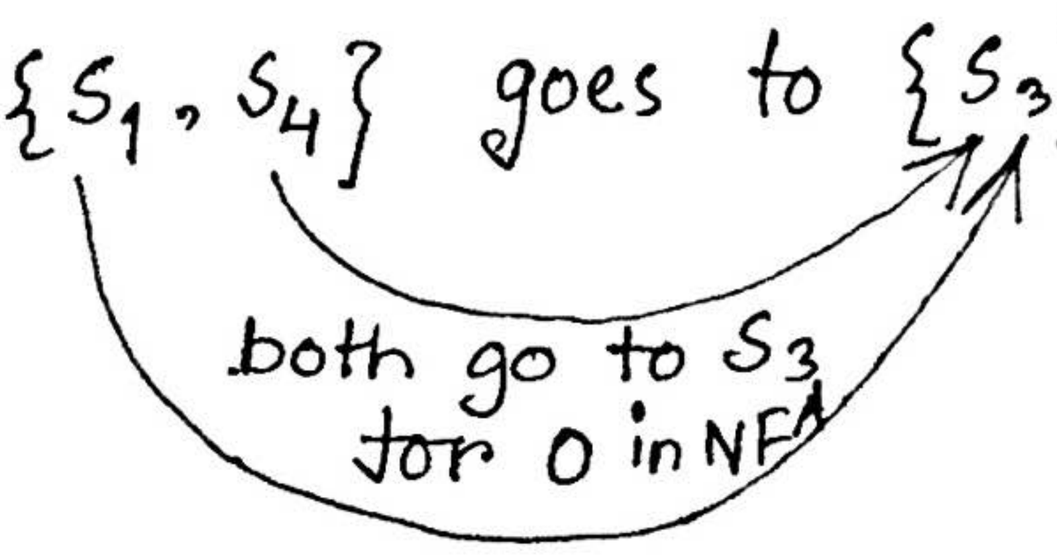
- In NFA, S_0 has transition to itself and S_0 for 0 has another transition to S_2

In DFA, S_0 thus go to $\{S_0, S_2\}$ for "0"

- NOW, in DFA $\{S_0, S_2\}$ goes to $\{S_1, S_4\}$

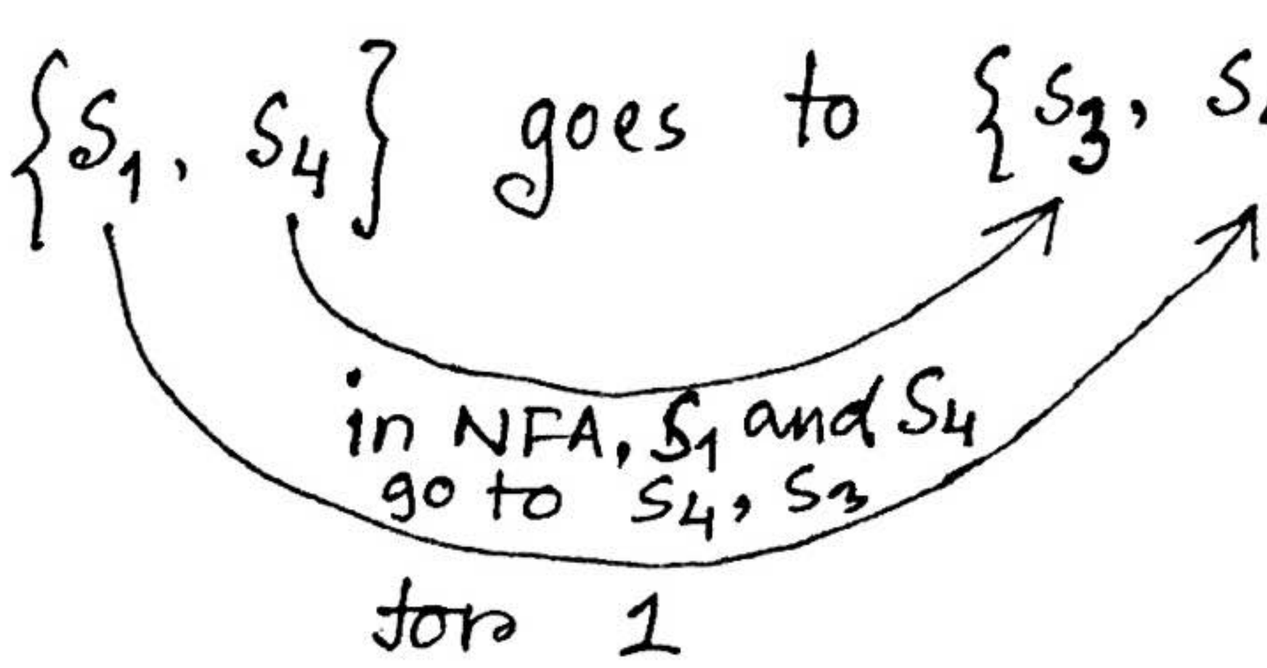


- NOW, in DFA $\{S_1, S_4\}$ goes to $\{S_3\}$ for "0"



Also.

$\{S_1, S_4\}$ goes to $\{S_3, S_4\}$ for 1



- Empty state becomes one of the states of DFA because it is the subset of containing all the next states of $\{S_3\}$

- Final state: Set of final states are those that include S_2 or S_4