Unit-1

1.1 Why Study Automata Theory?

1.1.1 Introduction to Automata

- Properties of *finite-state systems* ---
  - ♦ Having a finite number of states
  - ♦ With its running history not kept
  - ♦ Good to implement with a fixed set of resources
  - ♦ Easy to model by finite automata (FA)

- Example 1.1 --- an FA modeling an on/off switch (see Fig. 1.1)

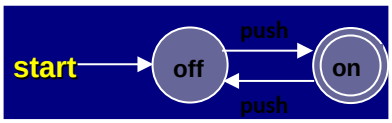


Fig. 1.1 An FA modeling an on/off switch.

- Example 1.2 --- an FA modeling recognition of the keyword "then" in a lexical analyzer
  - ♦ The double circles specify the "final" or "accepting" state.

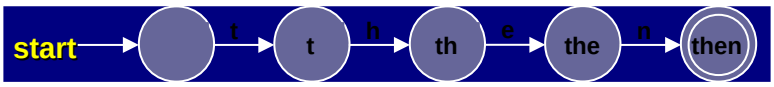


Fig. 1.2 an FA modeling recognition of the keyword "then."

1.1.2 Structural Representation

- Two other models (not automaton-like) for string data representation ---
  - ♦ Grammar --- processing data (strings of symbols) with recursive structures
    - e.g., *grammatical rule* "E → E + E" for generating arithmetic expressions like "2+ 2"

- Regular expression --- describing text strings
  - e.g., UNIX-style *regular expression* '[A-Z][a-z]*[ ][A-Z][A-Z]' for describing partial addresses Ithaca NY, Lafayette IN...

1.1.3 Automata and Complexity

- What can a computer do? ---
  This is a question which is answered by the study of *computability*!

- Computability is a study of problems which can be solved by computers, called *decidable problems*.
- *Decidability* is the main topic in the study of computability.

- What can a computer do *efficiently*? ---
  This is a question which can be answered by the study of *computational complexity*!

- Computational complexity is a study of:
  - *tractable problems* solvable with slowly growing functions (like polynomial) of input size, and
  - *intractable problems* solvable with fast growing functions (like exponential).
- Intractability is the main topic of computational complexity.

1.2 Central Concepts of Automata Theory

Three basic concepts

- *Alphabet* --- a set of symbols
- *Strings* --- a sequence of symbols from an alphabet
- *Language* --- a set of strings from the same alphabet

1.2.1 Alphabets

- Definition ---
  An *alphabet* is a finite, nonempty set of symbols.

- Conventional notation --- Σ
- The term "symbol" is usually undefined.
- Examples ---
  - Binary alphabet $\Sigma = \{0, 1\}$
  - English alphabet$\Sigma = \{a, b, ..., z\}$ ...

1.2.2 Strings

- Definition ---
  A *string* (or *word*) is a finite sequence of symbols from an alphabet.

  ♦ An example --- 1011 is a string from the binary alphabet $\Sigma = \{0, 1\}$.

- *Empty string* ε --- a string with zero occurrences of symbols.

- *Length* $|w|$ of string w --- the number of positions for symbols in w
  ♦ Examples --- $|0111| = 4$, $|\varepsilon| = 0$, ...

- *Power of a symbol a* ---
  The kth power a^k *of a is the result of* concatenating *a for k times*, *i.e.*,

  $$a^k = aa...a \text{ (}k \text{ times)}.$$

- *Power* of an alphabet Σ ---
  The *k*th power of Σ, Σ^k is a set of all strings of length *k*.

  ♦ Examples --- given $\Sigma = \{0, 1\}$, we have
  $\Sigma^0 = \{\varepsilon\}$, $\Sigma^2 = \{00, 01, 10, 11\}$

- *Power of a string x* (supplemental) ---
  ♦ Defined by *concatenation* ---
  $x^i = xx...x$ (x concatenated i times)

  ♦ Defined by *recursion* ---
  (1) $x^0 = \varepsilon$ (by definition); and

  (2) $x^i = xx^{i-1}$

  ♦ Examples --- $(10)^0 = \varepsilon$, $(011)^2 = 011011$...
  ♦ Note --- for a symbol *a*, we define $a^0 = \varepsilon$ (i.e., in this case we regard *a* as a one-symbol string).

- *Set of all strings* over Σ --- denoted as Σ^*
  ♦ It is not difficult to know that $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup ...$

- Σ^+ = the set of nonempty strings from $\Sigma = \Sigma^* - \{\varepsilon\}$
  ♦ Therefore, we have
  $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup ...$

$\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

- *Concatenation* of two strings *x* and *y* --- *xy*
  - Examples ---
    if x = 01101, y = 110, then xy = 01101110, $xx = x^2$ = 0110101101, …
  - ε is the *identity* for concatenation since $\varepsilon w = w\varepsilon = w$.

1.2.3 Languages

- Definition ---
  A language is a set of strings all chosen from some Σ^*.
  - In other words:
    if Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a language over Σ.

- Examples ---
  - The set of all legal English *words* is a language.
    - Why? What is the alphabet here?
      Answer: *the set of all letters*.
  - A legal program of C is a language.
    - Why? What is the alphabet here?
      Answer: a subset of the ASCII characters.

- More examples of languages ---
  - The set of all strings of n 0's followed by n 1's for $n \geq 0$: {ε, 01, 0011, 000111, …}
  - Σ^* is an infinite language for any alphabet Σ.
  - ϕ denotes the *empty language* (not the empty string ε) which is a language over any alphabet.
  - {ε} is a language over any alphabet (consisting of only one string, the empty string ε).

- Ways to describe languages ---
  - Description by *exhaustive listing* ---
    - L_1 = {*a*, *ab*, *abc*} (finite language; listed one by one)
    - L_2 = {*a*, *ab*, *abb*, *abbb*, …} (infinite language; listed partially)
    - $L_3 = L(ab^*)$ (infinite language; expressed by a *regular expression*)
  - Description by *generic elements* ---
    - L_4 = {x | x is over V = {*a*, *b*}, begins with *a*, followed by any number of *b*, possible none}
    - Note: $L_4 = L_3 = L_2$
  - Description by *integer parameters* ---

- $L_5 = \{ab^n \mid n \geq 0\}$
- Note: $L_5 = L_4 = L_3 = L_2$

Operations on Languages (supplemental)

- Languages are sets, and operations of sets may be applied to them:
  (1) *union* --- $A \cup B = \{a \mid a \in A \text{ or } a \in B\}$

  (2) *intersection* --- $A \cap B = \{a \mid a \in A \text{ and } a \in B\}$

  (3) *difference* --- $A - B = \{a \mid a \in A \text{ and } a \notin B\}$

  (4) *product* --- $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$

  (5) *complement* --- $\bar{A} = \{a \mid a \in U \text{ and } a \notin A\}$

  (6) *power set* --- $2^A = \{B \mid B \subset A\}$

  - Note: U above is the universal set, just like Σ^* which is the *closure* of an alphabet (defined later)

More Operations on Languages (supplemental)

- *Concatenation* of two languages L_1 and L_2 ---
  $L_1L_2 = \{x_1x_2 \mid x_1 \in L_1 \text{ and } x_2 \in L_2\}$

- *Power* of a language L ---
  - Defined directly ---
    $L^k = \{x_1x_2...x_k \mid x_1, x_2, ..., x_k \in L\}$

  - Defined by recursion ---
    (1) $L^0 = \{\varepsilon\}$; and

    (2) $L^i = LL^{i-1}$

- *Closure* of language L ---
  $L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup ...$

- *Positive closure* of a language L ---
  $L^+ = L^1 \cup L^2 \cup ...$

- Note: $L^* - L^0 = L^* - \{\varepsilon\}$

1.2.4 Problems

- A problem in automata theory ---
  deciding whether a given string is a member of some particular language.

♦ That is, if Σ is an alphabet, and L is a language over Σ, the *problem* L is:
given a string w in Σ^*, decide if $w \in L$ or not.

♦ The solution will be studied later in the topic of decidability.

1.3 Automata:

A algorithm or program that automatically recognizes if a particular string belongs to the language or not, by checking the grammar of the string.

An automata is an abstract computing device (or machine).

There are different varities of such abstract machines (also called models of computation) which can be defined mathematically.

Every Automaton fulfills the three basic requirements.

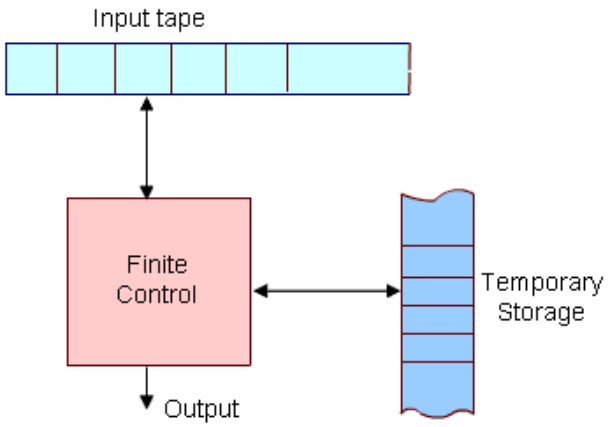
♦ Every automaton consists of some essential features as in real computers. It has a mechanism for reading input. The input is assumed to be a sequence of symbols over a given alphabet and is placed on an input tape(or written on an input file). The simpler automata can only read the input one symbol at a time from left to right but not change. Powerful versions can both read (from left to right or right to left) and change the input.

♦ The automaton can produce output of some form. If the output in response to an input string is binary (say, accept or reject), then it is called an accepter. If it produces an outputsequence in response to an input sequence, then it is called a transducer(or automatonwith output).

♦ The automaton may have a temporary storage, consisting of an unlimited number of cells, each capable of holding a symbol from an alphabet (whcih may be different from the input alphabet). The automaton can both read and change the contents of the storage cells in the temporary storage. The

accusing capability of this storage varies depending on the type of the storage.

♦ The most important feature of the automaton is its control unit, which can be in any one of a finite number of interval states at any point. It can change state in some defined manner determined by a transition function.



The figure above shows a diagrammatic representation of a generic automation.

Operation of the automation is defined as follows.

- At any point of time the automaton is in some integral state and is reading a particular symbol from the input tape by using the mechanism for reading input. In the next time step the automaton then moves to some other integral (or remain in the same state) as defined by the transition function.

- The transition function is based on the current state, input symbol read, and the content of the temporary storage. At the same time the content of the storage may be changed and the input read may be modifed. The automation may also produce some output during this transition.

- The internal state, input and the content of storage at any point defines the configuration of the automaton at that point. The transition from one configuration to the next (as defined by the transition function) is called a *move.* Finite state machine or *Finite Automation* is the simplest type of abstract machine we consider. Any system that is at any point of time in one of a finite number of interval state and moves among these states in a defined manner in response to some input, can be modeled by a finite automaton. It doesnot have any temporary storage and hence a restricted model of computation.

Finite Automata

Automata (singular : automation) are a particularly simple, but useful, model of computation.

They were initially proposed as a simple model for the behavior of neurons.

States, Transitions and Finite-State Transition System :

Let us first give some intuitive idea about *a state of a system* and *state transitions* before describing finite automata.

Informally, *a state of a system* is an instantaneous description of that system which gives all relevant information necessary to determine how the system can evolve from that point on.

Transitions are changes of states that can occur spontaneously or in response to inputs to the states. Though transitions usually take time, we assume that state transitions are instantaneous (which is an abstraction).

Some examples of state transition systems are: digital systems, vending machines, etc. A systemcontaining only a finite number of states and transitions among them is calleda *finite-state transition system*.

Finite-state transition systems can be modeled abstractly by a mathematical model called*finite automation*

Deterministic Finite (-state) Automata

Informally, a DFA (Deterministic Finite State Automaton) is a simple machine that reads an input

string -- one symbol at a time -- and then, after the input has been completely read, decides whether to accept or reject the input. As the symbols are read from the tape, the automaton can change its state, to reflect how it reacts to what it has seen so far.

A machine for which a deterministic code can be formulated, and if there is only one unique way to formulate the code, then the machine is called deterministic finite automata.

Thus, a DFA conceptually consists of 3 parts:

1.A *tape* to hold the input string. The tape is divided into a finite number of cells.

Each cell holds a symbol from .

2. A *tape head* for reading symbols from the tape

3. A *control* , which itself consists of 3 things:

- finite number of states that the machine is allowed to be in (zero or more states
  are designated as *accept* or *final* states),
- a current state, initially set to a start state,
- a state transition function for changing the current state.

An automaton processes a string on the tape by repeating the following actions until the tape head has traversed the entire string:

1. The tape head reads the current tape cell and sends the symbol s found there to the control. Then the tape head moves to the next cell.

2. he control takes s and the current state and consults the state transition function to get the next state, which becomes the new current state.

Once the entire string has been processed, the state in which the automation enters is examined.

If it is an accept state , the input string is accepted ; otherwise, the string is rejected . Summarizing all the above we can formulate the following formal definition:

Deterministic Finite State Automaton : A Deterministic Finite State Automaton (DFA) isa 5-tuple:

$$(Q, \Sigma, \delta, q_0, F)$$

Definition: A deterministic finite automaton (DFA) consists of

1. a finite set of *states* (often denoted Q)
2. a finite set Σ of *symbols* (alphabet)
3. a *transition function* that takes as argument a state and a symbol and returns a state (often denoted δ)
4. a *start state* often denoted q_0
5. a set of *final* or *accepting* states (often denoted F)

We have $q_0 \in Q$ and $F \subseteq Q$

The transition function δ is a function in

$Q \times \Sigma \rightarrow Q$

$Q \times \Sigma$ is the set of 2-tuples (q, a) with $q \in Q$ and $a \in \Sigma$

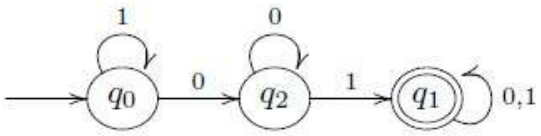
How to present a DFA? With a *transition table*

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

The $\rightarrow$ indicates the *start* state: here q_0

The $*$ indicates the final state(s) (here only one final state q_1)

This defines the following *transition diagram*



For this example

$Q = \{q_0, q_1, q_2\}$

start state q_0

$F = \{q_1\}$

$\Sigma = \{0, 1\}$

δ is a *function* from $Q \times \Sigma$ to Q

$\delta : Q \times \Sigma \rightarrow Q$

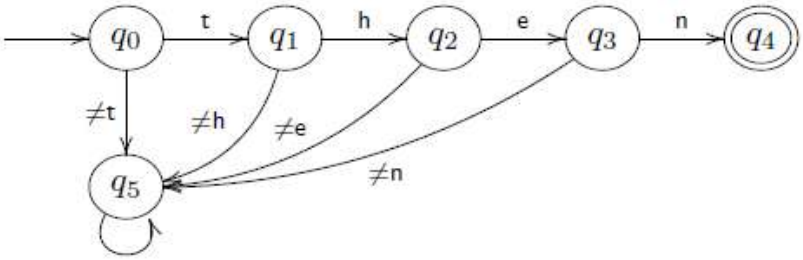
$\delta(q_0, 1) = q_0$

$\delta(q_0, 0) = q_2$

e Example:

When does the automaton accepts a word??

It reads the word and accepts it if it stops in an accepting state



Only the word then is accepted

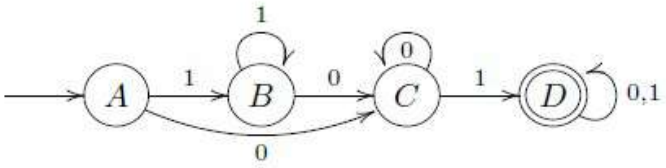
Here $Q = \{q_0, q_1, q_2, q_3, q_4\}$

Σ is the set of all characters

$F = \{q_4\}$

We have a "stop" or "dead" state q_5, *not* accepting

We get the following automaton



Transition table

	0	1
→A	C	B
B	C	B
C	C	D
*D	D	D

Q = {A,B,C,D}, Σ = {0,1}, start state A, final state(s) {D}

Extending the Transition Function to Strings

In the previous example, what happens if we get 011? 100? 10101?

We define $\hat{\delta}(q, x)$ by induction

$\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

BASIS $\hat{\delta}(q, \epsilon) = q$ for $|x| = 0$

INDUCTION suppose $x = ay$ (y is a string, a is a symbol)

$\hat{\delta}(q, ay) = \hat{\delta}(\delta(q, a), y)$

Notice that if $x = a$ we have

$\hat{\delta}(q, a) = \delta(q, a)$ since $a = a\epsilon$ and $\hat{\delta}(\delta(q, a), \epsilon) = \delta(q, a)$

$\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

We write $q.x$ instead of $\hat{\delta}(q, x)$

We can now define mathematically the *language* accepted by a given automaton $Q, \Sigma, \delta, q_0, F$

$L = \{x \in \Sigma^* \mid q_0.x \in F\}$

On the previous example 100 is not accepted and 10101 is accepted

Product of automata

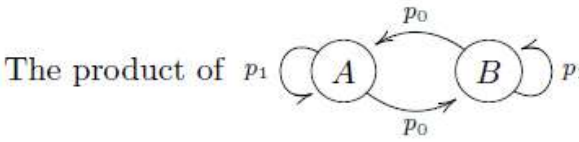
How do we represent *interaction* between machines?

This is via the *product* operation

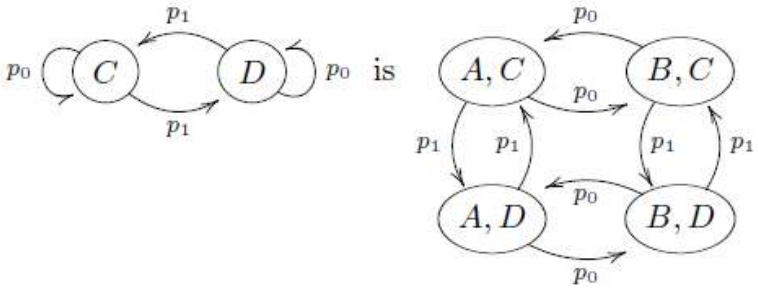
There are different kind of products

We may then have *combinatorial explosion*: the product of n automata with 2 states has 2^n states!

Product of automata (example)

The product of 

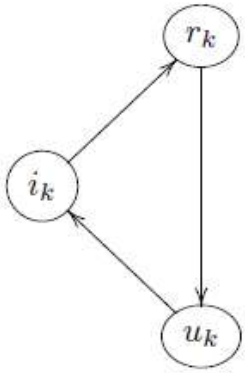
and



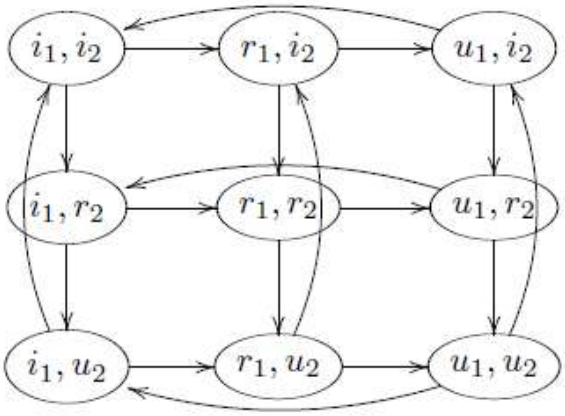
If we start from A, C and after the word w we are in the state A,D we know that w contains an even number of p_0s and odd number of p_1s

Model of a system of *users* that have three states I(dle), R(equesting) and U(sing). We have two users for $k = 1$ or $k = 2$

Each user is represented by a simple automaton



The complete system is represented by the product of these two automata; it has $3 \times 3 = 9$ states



The Product Construction

Given $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ two DFAs *with the same alphabet* Σ we can define the product $A = A_1 \times A_2$

set of state $Q = Q_1 \times Q_2$

transition function $(r_1, r_2).a = (r_1.a, r_2.a)$

intial state $q_0 = (q_1, q_2)$

accepting states $F = F_1 \times F_2$

Complement

If $A = (Q, \Sigma, \delta, q_0, F)$ we define the *complement* $\bar{A}$ of A as the automaton

$\bar{A} = (Q, \Sigma, \delta, q_0, Q - F)$

Theorem: *If A is a DFA, then $L(\bar{A}) = \Sigma^* - L(A)$*

Remark: We have $A \oplus A' = \overline{\overline{A} \times \overline{A'}}$

Languages

Given an alphabet Σ

A *language* is simply a *subset* of Σ^*

Common languages, programming languages, can be seen as sets of words

Definition: *A language $L \subseteq \Sigma^*$ is* regular *iff there exists a DFA A, on the same alphabet Σ such that $L = L(A)$*

Theorem: *If L_1, L_2 are regular then so are*
$L_1 \cap L_2$, $L_1 \cup L_2$, $\Sigma^* - L_1$

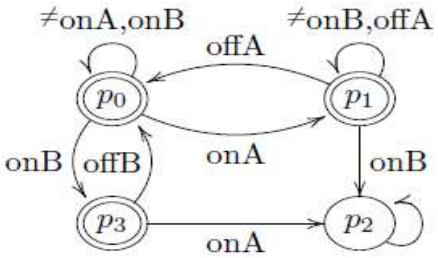
Application: control system

We have several machines working concurrently

We need to forbid some sequence of actions. For instance, if we have two machines MA and MB, we may want to say that MB cannot be on when MA is on. The alphabets will contain: onA, offA, onB, offB

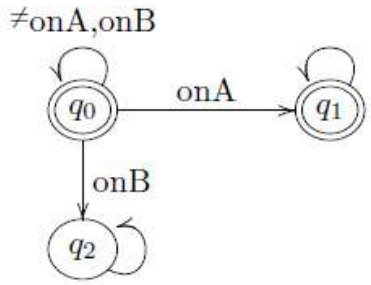
Between onA, on2 there should be at least one offA

The automaton expressing this condition is



What is interesting is that we can use the product construction to combine several conditions

For instance, another condition maybe that onA should appear before onB appear. One automaton representing this condition is



We can take the product of the two automata to express the two conditions as one automaton, which may represent the control system

NFA: Given a current state of the machine and input symbol to be read, the next state is not uniquely determined.
NFAs have 3 features when compared with DFAs.
1. Ability to take a step without reading any input symbol
2. A state may have no transition on a particular symbol
3. Ability to transition to more than one state on a given symbol

ϵ-Transitions

Transitions without reading input symbols
Example 1. The British spelling of \color" is \colour". In a web search application, you may want
to recognize both variants.

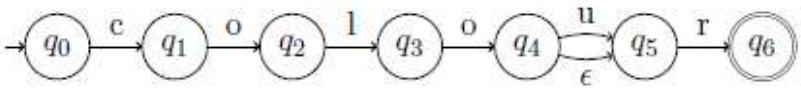


Figure 3: NFA with ϵ-transitions

No transitions

Example 2.

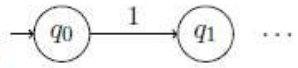


Figure 4: No 0-transition out of initial state

In the above automaton, if the string starts with a 0 then the string has no computation (i.e., rejected).

Multiple Transitions

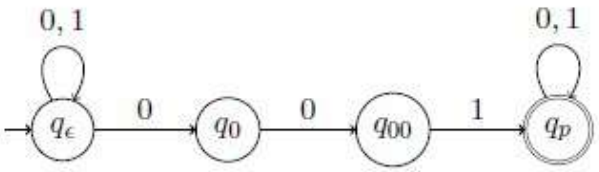


Figure 5: q_ϵ has two 0-transitions

Nondeterministic Finite Automata (NFA)

Formal Definition

Definition 3. A nondeterministic finite automaton (NFA) is $M = (Q, \Sigma, \delta, q_0, F)$, where

- Q is the finite set of states
- Σ is the finite alphabet
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is the powerset of Q
- $q_0 \in Q$ initial state
- $F \subseteq Q$ final/accepting states

Example of NFA

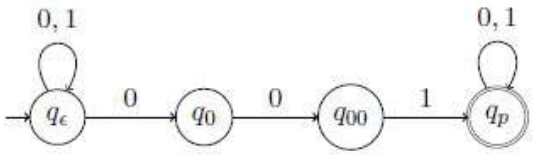


Figure 9: Transition Diagram of NFA

Formally, the NFA is $M_{001} = (\{q_\epsilon, q_0, q_{00}, q_p\}, \{0,1\}, \delta, q_\epsilon, \{q_p\})$ where δ is given by

$$\delta(q_\epsilon, 0) = \{q_\epsilon, q_0\} \qquad \delta(q_\epsilon, 1) = \{q_\epsilon\} \qquad \delta(q_0, 0) = \{q_{00}\}$$
$$\delta(q_{00}, 1) = \{q_p\} \qquad \delta(q_p, 0) = \{q_p\} \qquad \delta(q_p, 1) = \{q_p\}$$

δ is $\emptyset$ in all other cases.

Converting NFA to DFA:

Let **X = (Q_x, ∑, δ_x, q_0, F_x)** be an NDFA which accepts the language L(X). We have to design an equivalent DFA **Y = (Q_y, ∑, δ_y, q_0, F_y)** such that **L(Y) = L(X)**. The following procedure converts the NDFA to its equivalent DFA

Step 1 − Create state table from the given NDFA.

Step 2 − Create a blank state table under possible input alphabets for the equivalent DFA.

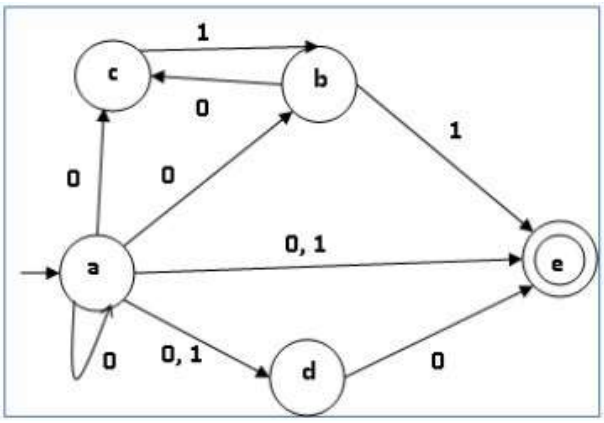
Step 3 − Mark the start state of the DFA by q0 (Same as the NDFA).

Step 4 − Find out the combination of States {Q_0, Q_1,... , Q_n} for each possible input alphabet.

Step 5 − Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

Step 6 − The states which contain any of the final states of the NDFA are the final states of the equivalent DFA.

Let us consider the NDFA shown in the figure below.

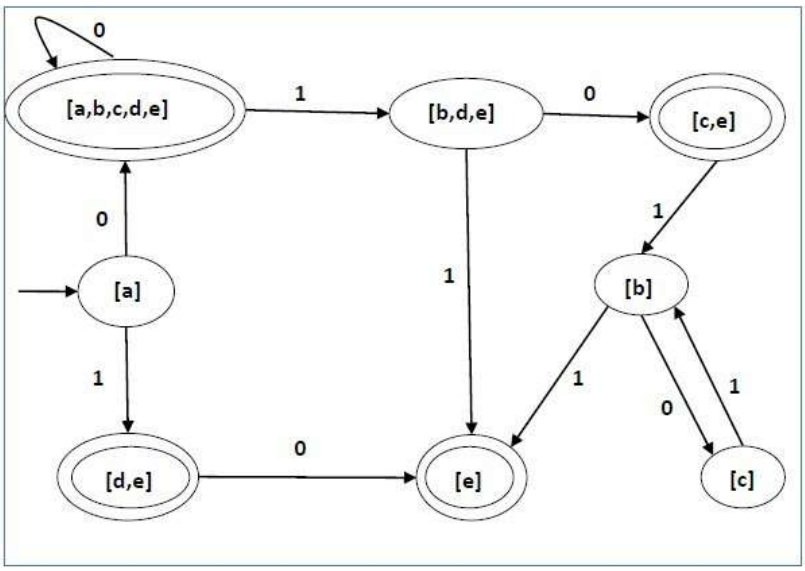


q	δ(q,0)	δ(q,1)
a	{a,b,c,d,e}	{d,e}
b	{c}	{e}
c	∅	{b}
d	{e}	∅
e	∅	∅

Using the above algorithm, we find its equivalent DFA. The state table of the DFA is shown in below.

q	δ(q,0)	δ(q,1)
[a]	[a,b,c,d,e]	[d,e]
[a,b,c,d,e]	[a,b,c,d,e]	[b,d,e]
[d,e]	[e]	∅
[b,d,e]	[c,e]	[e]
[e]	∅	∅
[c, e]	∅	[b]
[b]	[c]	[e]
[c]	∅	[b]

The state diagram of the DFA is as follows −



∈- transitions :

In an -transition, the tape head doesn't do anything- it doesnot read and it doesnot move. However, the state of the automata can be changed - that is can go to zero, one or more states. This is written formally as $\delta(q,\in)=\{q_1, q_2,\cdots,q_k\}$ implying that the next state could by any one of $q_1, q_2,\cdots,q_k$ w/o consuming the next input symbol.

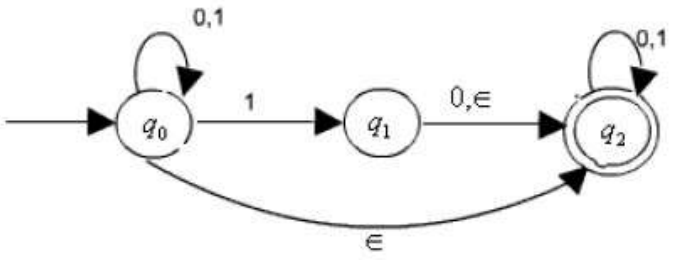
Formal definition of NFA :

Formally, an NFA is a quituple $M=(Q,\Sigma,\delta,q_0,F)$ where Q, Σ, q_0, and F bear the same meaning as for a DFA, but δ, the transition function is redefined as follows:

$\delta: Q\times(\Sigma\cup\{\in\})\rightarrow P(Q)$

where $P(Q)$ is the power set of Q i.e. 2^Q.

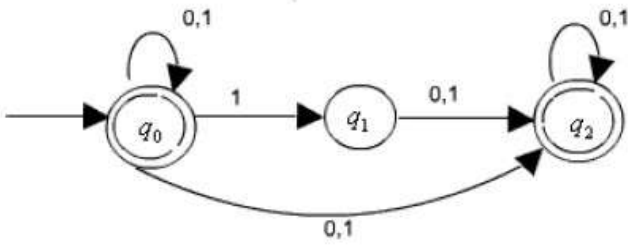
Ex: Consider the following *NFA* with ∈- transition.



Transition Diagram δ

	0	1	$\in$
$\rightarrow q_0$	$\{q_0\}$	$\{q_0, q_1\}$	$\{q_2\}$
q_1	$\{q_2\}$	ϕ	$\{q_2\}$
$F \quad q_2$	$\{q_2\}$	ϕ	$\{q_2\}$

Transition diagram for δ' for the equivalent *NFA* without $\in$- moves



	0	1
$\rightarrow_F q_0$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$
q_1	$\{q_2\}$	$\{q_2\}$
$F \quad q_2$	$\{q_2\}$	$\{q_2\}$

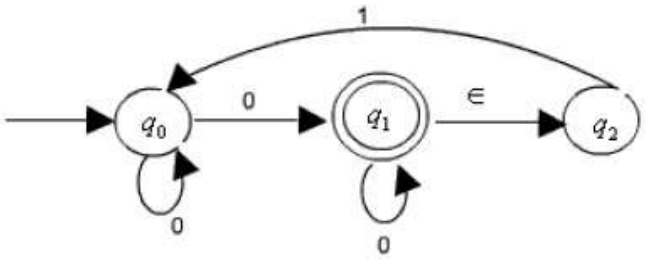
$\in$-closures:

The concept used in the above construction can be made more formal by defining the $\in$-closure for a state (or a set of states). The idea of $\in$-closure is that, when moving from a state *p* to a state *q* (or from a set of states S_i to a set of states S_j) an input $a \in \Sigma$, we need to take account of all $\in$-moves that could be made after the transition. Formally, for a given state *q*,

$\in$-closures: $(q) = \{p \mid p \text{ can be reached from } q \text{ by zero or more } \in\text{-moves}\}$

Similarly, for a given set $R \subseteq Q$

Example : Consider the NFA given below.



	0	1	∈
$\rightarrow q_0$	$\{q_0, q_1\}$	ϕ	ϕ
$F\ q_1$	$\{q_1\}$	ϕ	$\{q_2\}$
q_2	ϕ	ϕ	$\{q_0\}$

	0	1
ϕ	ϕ	ϕ
$\rightarrow q_0$	$\{q_0, q_1, q_2\}$	ϕ
$F\{q_1\}$	$\{q_1, q_2\}$	$\{q_0\}$
$\{q_2\}$	ϕ	$\{q_0\}$
$F\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
$F\{q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_0\}$
$F\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

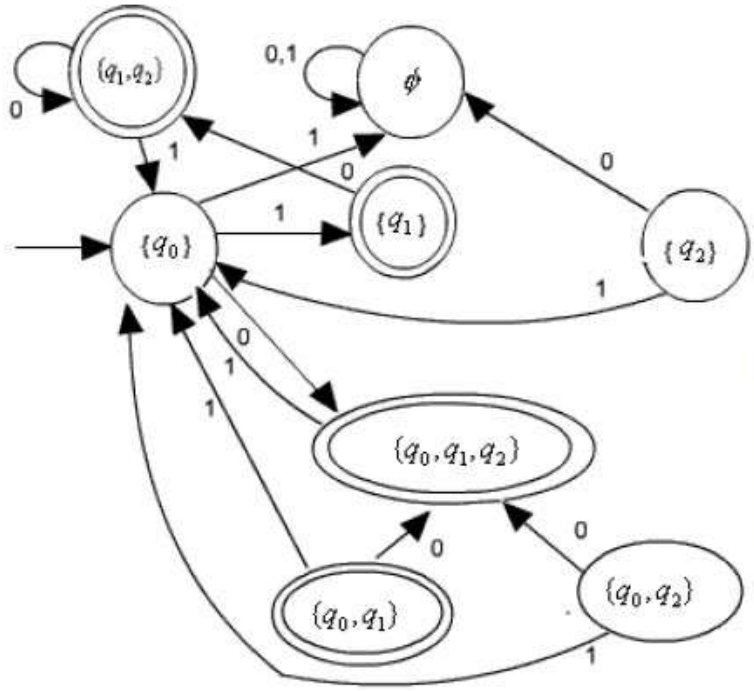
The start state of the *DFA* is $\in$- closures $(q_0) = \{q_0\}$

The final states are all those subsets that contains q_1 (since $q_1 \in F$ in the *NFA*).

Let us compute one entry,

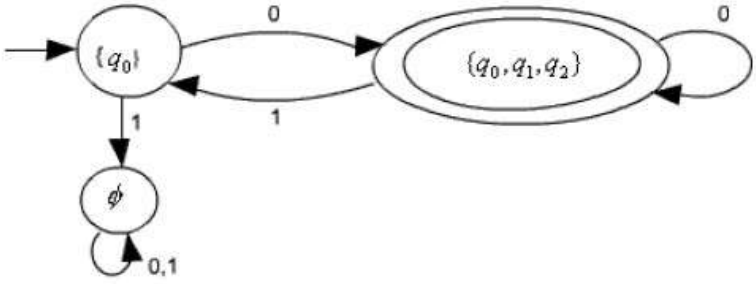
$$\delta^D(\{q_0, 0\}) = \in\text{-closure}(\delta(q_0, 0))$$
$$= \in\text{-closure}(\{q_0, q_1\})$$
$$= \{q_0, q_1, q_2\}$$

Similarly, all other transitions can be computed



	0	1
$\rightarrow\{q_0\}$	$\{q_0,q_1,q_2\}$	ϕ
$\{q_0,q_1,q_2\}$	$\{q_0,q_1,q_2\}$	$\{q_0\}$

Corresponding Transition fig. for DFA. Note that states $\{q_1\},\{q_2\},\{q_1,q_2\},\{q_0,q_2\}$ and $\{q_0,q_1\}$ are not accessible and hence can be removed. This gives us the following simplified *DFA* with only 3 states.



Finite State Machines with Output (Mealy and Moore Machines)

Finite automata are like computers in that they receive input and process the input by changing states. The only output that we have seen finite automata produce so far is a yes/no at the end of processing.

We will now look at two models of finite automata that produce more output than a yes/no.

Moore machine:

- In Moore machine. the value of output function is depend on the present state only.
- Moore machine is described by 6-tuples - **(Q, Σ, Δ, δ, λ, q_0)**

where

Q = Finite non-empty set of states;

Σ = Set of input alphabets.

Δ = Set of output alphabets.

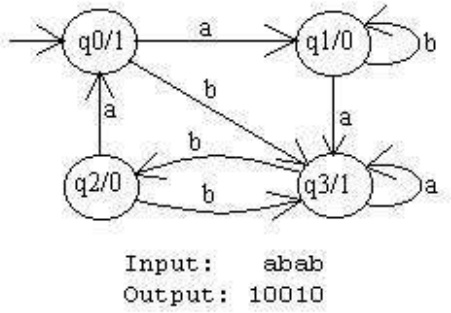
δ = Transition function mapping Q X Σ → Q

λ = Output function mapping Q → Δ

q_0 = Initial state.

Sample Transition Table:

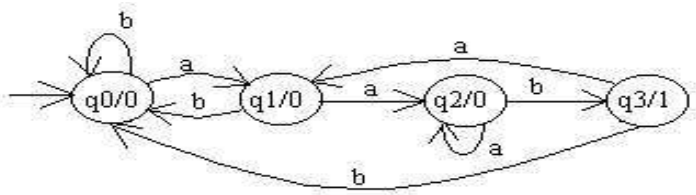
Present State	Next State		Output
	a = 0	a = 1	
-> q0	q3	q1	1
q1	q0	q3	0
q2	q2	q2	0
q3	q1	q0	1



Basically a Moore machine is just a FA with two extras.

1. It has TWO alphabets, an input and output alphabet.

2. It has an output letter associated with each state. The machine writes the appropriate output letter as it enters each state.




```
Input:   babaababaab
Output: 000000100001
```

The output produced by the machine contains a 1 for each occurrence of the substring aab found in the input string.

Mealy machine

- In Mealy machine. the value of output function is depend on the present state and present input.
- Mealy machine is described by 6-tuples - **$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$**

where

Q = Finite non-empty set of states;

Σ = Set of input alphabets.

Δ = Set of output alphabets.

δ = Transitional function mapping Q X $\Sigma \rightarrow$ Q

λ = Output function mapping Q X $\Sigma \rightarrow \Delta$

q_0 = Initial state.

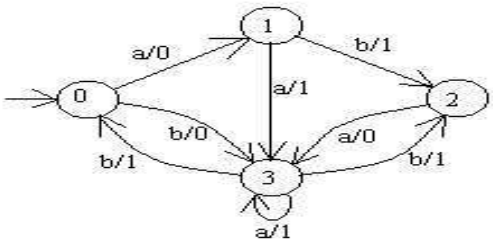
Sample Transition table:

Present State	Next State			
	a = 0		a = 1	
	State	Output	State	Output
-> q0	q3	0	q1	1
q1	q0	1	q3	0
q2	q2	1	q2	0
q3	q1	0	q0	1

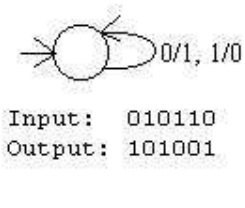
Mealy Machines are exactly as powerful as Moore machines

(we can implement any Mealy machine using a Moore machine, and vice versa).

However, Mealy machines move the output function from the state to the transition. This turns out to be easier to deal with in practice, making Mealy machines more practical.



A Mealy machine produces output on a transition instead of on entry into a state.



Transitions are labelled *i/o* where

- ***i*** is a character in the input alphabet and
- ***o*** is a character in the output alphabet.

- The following Mealy machine takes the one's complement of its binary input. In other words, it flips each digit from a 0 to a 1 or from a 1 to a 0.
- Mealy machines are complete in the sense that there is a transition for each character in the input alphabet leaving every state.

There are no accept states in a Mealy machine because it is not a language recogniser, it is an output producer. Its output will be the same length as its input.

(Parts of the) output of a Mealy/Moore automaton can be input to another automaton

- Pipeline
- Dataflow architecture
- Feedback network

Conversion from Mealy machine to Moore machine:

Steps:

1. Determine the number of different output associated with qi in the next state column.

2. we split qi into different states according to different output associated with it. for ex. suppose in the next state column of the above sample transition table of mealy machine, the output associated with q1 is "0" in the first next state column and "1" in the second next state column. so we split q1 into q10 and q11 states. similarly check others and split them.

Example 1: consider the above sample transition table of the mealy machine. convert it into corresponding Moore machine.

Solution: After applying the conversion steps, we get two states (q1 and q2) that are associated with different outputs (0 and 1). so we split both states into q10 , q11 and q20, q21.

Now the transition table becomes

Present State	Next State			
	a = 0		a = 1	
	State	Output	State	Output
-> q0	q3	0	q11	1
q10	q0	1	q3	0
q11	q0	1	q3	0
q20	q21	1	q20	0
q21	q21	1	q20	0
q3	q10	0	q0	1

Here

- whole row of q1 is copied to q10 , q11 and whole row of q2 is copied to q20 and q21 of the sample transition table of mealy machine.
- The outputs of the next state columns of q1 and q2 are depend on the previous output. For ex. in the first row, q1 becomes q11 because the out of q1 is 1. in the fourth row, q2 becomes

q21 because the output of the q2 is 1. and in the subsequent column q2 becomes q20 because the output of q2 in that column was 0. and so on

now in moore machine format, we copied all the states and common output because in moore machine. the outputs of the next state are common.

Present State	Next State		Output
	a = 0	a = 1	
-> q0	q3	q11	1
q10	q0	q3	0
q11	q0	q3	1
q20	q21	q20	0
q21	q21	q20	1
q3	q10	q0	0

This table is moore machine table corresponding to the sample mealy machine.

Exercise : convert the following mealy machine to corresponding moore machine

Present State	Next State			
	a = 0		a = 1	
	State	Output	State	Output
->q0	q1	0	q3	0
q1	q3	1	q2	0
q2	q4	1	q0	0
q3	q0	0	q4	1
q4	q2	0	q1	1

Conversion from Moore machine to Mealy machine:

For understanding the conversion of moore to mealy machine, let us take an example:

suppose the moore machine transition table is:

Present State	Next State		Output
	a = 0	a = 1	
-> q0	q3	q1	1
q1	q0	q3	0
q2	q2	q2	0
q3	q1	q0	1

convert this transition table into mealy machine.

Solution: First of all take the mealy machine transition table format, i.e.,

Present State	Next State			
	a = 0		a = 1	
	State	Output	State	Output
-> q0	q3	1	q1	0
q1	q0	1	q3	1
q2	q2	0	q2	0
q3	q1	0	q0	1

Minimization of Deterministic Finite Automata:

Minimization of automata refers to detect those states of automata whose presence or absence in a automata does not affect the language accepted by automata. these states are like Unreachable states, Dead states, Non-distinguishable states etc.

Example:

Consider the following transition table example :

State	0	1
-> q0	q1	q5
q1	q6	q2
q2 (Final state)	q0	q2
q3	q2	q6
q4	q7	q5
q5	q2	q6
q6	q6	q4
q7	q6	q2

Minimize the above finite automata.

Solution:

For minimizing the above automata, we will have to make Π (pi) sets.

Π0 = { {q2}, {q0, q1, q3,q4,q5,q6,q7}}

Π1 = { {q2}, {q0, q4,q6}, {q1,q7}, {q3,q5}}

Π2 = { {q2}, {q0, q4}, {q6}, {q1,q7}, {q3,q5}}

Π3 = { {q2}, {q0, q4}, {q6}, {q1,q7}, {q3,q5}}

here Π2 = Π3 so stop making Π sets.

[**Trick:** for making Π sets, first of all make Π0 set. for this, make two set, first set contains final state and second set contains non final states.

For Π1 set , there are two sets in Π0:

First set = {q2}, that can not be partitioned further.

Second set = {q0, q1, q3,q4,q5,q6,q7}, that can be partitioned.

For partitioned second set, consider q0 and q1, the entries corresponding to q0 and q1under 0-column are q1 and q6. these entries belongs to second set.so no problem.

the entries corresponding to q0 and q1under 1-column are q5 and q2. in this q2 entry belongs to first set and q5 belong to second set. so problem occur.

due to this, we can not make combination of q0 and q1.

similarly check (q0,q3), {q0,q4), (q0,q5), (q0, q6), (q0,q7) states.

after checking all the states with q0, we make a new set {q0, q4, q6}.

after making new set, the remaining states are {q1,q3,q5,q7}.

Now we check states (q1,q3), (q1,q5), (q1,q7) states by following above procedure. after this, we get a new set {q1, q7}. and remaining states are {q3, q5}.

finally we check states (q3, q5) by following above procedure.

after checking all this, we make new set $\Pi 2$,

$\Pi 2$ = { {q2}, {q0, q4}, {q6}, {q1,q7}, {q3,q5}}

similar procedure applied to $\Pi 2$ set.

then we get $\Pi 3$ set. we make Π sets until we find last two Π sets equal.

in this example, we find $\Pi 2 = \Pi 3$ so stop making Π sets.

Final minimized DFA table is :

State	0	1
[q0, q4]	[q1, q7]	[q3, q5]
[q1, q7]	[q6]	[q2]
[q2]	[q0, q4]	[q2]
[q3, q5]	[q2]	[q6]
[q6]	[q6]	[q0, q4]

we take states from $\Pi 3$ set in minimized DFA table. after making minimized DFA table, we can make finite automata.