AUTOMATA THEORY- GRAMMAR, LANGUAGES & FINITE STATE MACHINES

- Terminologies of TOC
- Languages
- Grammar
- Noam Chomsky hierarchy
- Parse Tree
- BNF Notation
- Finite automation
- Regular Expressions
- Theory of Automata
- DFA & NFA
- Pumping lemma
- Languages which are not Regular
- FSM with Output
- Turing Machine

Terminologies of TOC

- **Alphabet** – finite set of symbols

  – denoted by Σ.

  Examples :

  $\Sigma = \{a,b, c,\ldots, z\}$
  $\Sigma = \{0, 1\}$
  $\Sigma = \{a\}$
  $\Sigma = \{\varepsilon\}$
  $\Sigma = \emptyset$

Terminologies of TOC

- **String** (word) – *Finite* sequence of symbols from an alphabet., usually denoted by *w*.

Ex: *English* is a string from $\Sigma = \{a,b, c,\ldots, z\}$

01110 is a string from $\Sigma = \{0,1\}$

- **Empty string** – String with no symbols

ε or e

Terminologies of TOC

- **Length of a string *w*,** denoted by $|w|$ - is the number of symbols in a string.

  Eg:

  $|abc| = 3$

  $|b| = 1$

  $|\varepsilon| = 0$

  $|ababxyz| = 7$

Terminologies of TOC

- **Substring** – string contained in a string
  Eg: *a,ab,abc,* ε are substrings of string *abc*

- A string may have many occurrences of the same substring, Ex: *ababababc* contains the substring *ab* three times
- If w is a string then
  - $W^0 = \varepsilon$
  - $W^1 = w$
  - $W^2 = w\ w\ \ldots$
  - $W^{i+1} = w^i\ w$

String Operations

1. **Concatenation:**
   - binary operator

- concatenation of x and y, denoted by xy or $x \circ y$, is the string x followed by y
- Eg: $dog \circ house \Rightarrow doghouse$

$abc \circ \varepsilon \Rightarrow abc$

2. **Reversal:**
   - unary in nature
   - denoted by w^R, is the string w spelled backwards
   - Eg: $(top)^R = pot$

Terminologies of TOC

- (Formal) **Language** – set of strings over an alphabet, Σ, ie. any subset of Σ^* is called a language.

  Eg: If Σ = {0,1}, then

  - L_1= {*0, 1, 00, 11*} is a language
  - L_2= set of palindromes is a language, ie
  - L_2= {ε, *0, 1, 00, 11, 010, 101,...*}
  - Σ^*, {ε}, Ø

- A language may be finite or infinite
- Infinite language is written as L = { $w \in \Sigma^*$: *w* has property P}

Eg: L = { $w \in \{0,1\}^*$; *w* has equal number of 1's and 0's}

Operations on Languages

If L, L_1, L_2 are languages over Σ

1. Union
   $L = L_1 \cup L_2$
1. Intersection
   $L = L_1 \cap L_2$
1. Complement
   $L^c = \Sigma^* - L$
1. Concatenation
   $L = L_1.L_2 = \{w : w = x.y$, for some $x \in L_1$ and $y \in L_2\}$
1. Closure (Kleen closure)
   $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$
1. Positive closure
   $L^+ = L^* - \varepsilon$, $L^+ = L\,L^*$

Regular Expressions

A **Regular Expression** is a string that is used to describe or match a set of strings, according to certain syntax rules.

Eg: **0*** = {$\mathcal{E}$, *0*, *00*, *000*,…}

0+1 = {*0*, *1*}

01 = {*01*}

ab(a+b) = {*aba*, *abb*}

a(c+ d) b = {*acb*, *adb*}

a*b = {$\mathcal{E}$, *a*, *aa*, *aaa*, …} b = {*b*, *ab*, *aab*, *aaab*, …}

Regular Language – Language generated by a regular expression.

Regular Expressions

Operation	Symbol	Example	Regular Expression	
concatenation		ab	ab	
		[a-c][AB]	aA∪aB∪bA∪bB∪cA∪CB	
Kleene star	*	[ab]*	(a∪b)*	
disjunction	|	[ab]*|A	(a∪b)*∪A	
zero or more	+	a?	(a∪λ)	
one character	.	a.a	a(a∪b)a	if Σ={a,b}
n times	{n}	a{4}	aaaa=a^4	
n or more times	{n,}	a{4,}	aaaaa*	
n to m times	{n,m}	a{4,6}	aaaa∪aaaaa∪aaaaaa	

#> *man grep*

Regular Expressions

- Describe the following sets by regular expression

(1) {ε, *a*, *aa*, *aaa*, …}

a*

(2) {01,10}

01+10

(3) {01,011,0101}

01+011+0101 = 01(ε+1+01)

(4) Strings that end with a over Σ ={a, b}

(a+b)*a

(5) Strings of 0's and 1's containing 00 as substring

(0+1)*00(0+1)*

(6) Strings of 0's and 1's beginning with 0 and ending with 1

0(0+1)*1

(7) L= {ε, 11, 1111, 111111, …}

(11)*`

Noam Chomsky described the languages into four classes, namely

- Type 0 Language (Recursively Enumerable Language)
- Type 1 Language (Context Sensitive Language)
- Type 2 Language (Context Free language)
- Type 3 Language (Regular Language)

Grammar

Generates strings in language by a process of replacing symbols
- Similar to regular expressions.

Four elements.
- Terminal symbols:
  characters in alphabet – denote by 0 or 1 for binary alphabet.
- Nonterminal symbols:
  local variables for internal use – denote by <name>.
- Start Symbol: one special nonterminal.
  (analogous to start state in FSA)
- Production rules:
  replacement rules – denote by <A> c => <D> b <B>

A Familiar Example

Terminals: horse, dog, cat, saw, heard, the

Nonterminals: <sentence>, <subject>, <verb>, <object>

Start Symbol: <sentence>

Production rules: <sentence> => <subject> <verb><object>

<subject> => the horse

<subject> => the dog

<subject> => the cat

<object> => the horse

<object> => the dog

<object> => the cat

<verb> => saw

<verb> => heard

Some Strings: the horse saw the dog

the dog heard the cat

the cat saw the horse

Generating a String in Language

Start with the start Symbol.

<sentence>

Generating a string in language:

<sentence>

Generating a String in Language

Start with the start Symbol.

Use any applicable production rule.

<sentence> => <subject> <verb> <object>

Generating a string in language:

<sentence> => <subject> <verb> <object>

Generating a String in Language

Start with the start Symbol.

Use any applicable production rule.

```
<sentence> => <subject> <verb> <object>
                the horse     saw     the dog
```

Generating a string in language:

```
<sentence> => <subject> <verb> <object>
           => the horse saw the dog
```

The C Language Grammar

Terminals:

```
if do while for switch break continue typedef struct return main int
   long char float double void static ; ( ) a b c A B C 0 1 2 + * - / _
   # include += ++ …
```

Nonterminals:

```
<statement> <expression> <C source file> <identifier> <digit> <nondigit>
   <identifier> <selection-statement> <loop-statement>
```

Start symbol:

```
<C source file>
```

A String:

```
#include <stdio.h>
int main(void) {
   printf("Hello World!\n");
   return 0;
}
```

The C Language: Identifier

Production Rules:

```
<identifier>  =>  <nondigit>
              =>  <identifier> <nondigit>
               => <identifier> <digit>
<nondigit>    =>   a | b | … | Y | Z | _
<digit>       => 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Some identifiers:

```
x
f
temp
temp1
done
_CanStartWithUnderscoreButNor7
```

The C Language: Expressions

Production rules:

```
<expression>          => <identifier>
                      => <constant>
                      => <cond-expression>
                      => <assign-expression>
<cond-expression>     => <expression> > <expression>
                      => <expression> != <expression>
<assign-expression> => <expression> = <expression>
                      => <expression> += <expression>
```

Some expression:

- `x`
- `x > 4`
- `done != 1`
- `x = y = z = 0`
- `x += 2.0`

This grammar also considers `4 = x` a valid expression

The C Language: Statements

Production rules:

```
<statement>          => <select-statement>
                     => <loop-statement>
                     => <compound-statement>
                     => <express-statement>
<select-statement>   => if (<expression>)
                     => if (<expression>) <statement>
                          else <statement>
<loop-statement>     => while (<expression>) <statement>
                     => do <statement> while (<expression>)
<express-statement>  => <expression>;
```

A Statement:

```
while (done != 1)
    if(f(x) > 4.0)
     done = 1;
    else
     x += 2.0;
```

Grammar

In practice, need to write out grammar for C.

- Compiler check to see if your program is a valid "string" in the C language.
- The C Standard formalizes what it means to be a valid ANSI C program using grammar
- Compiler implementation: simulate FSA and PDA machine to recognize valid C programs.

Type III Grammar (Regular)

Limit production rules to have exactly one nonterminal on LHS and at most one nonterminal and terminal on RHS.

```
<A> => <B> a
<A> => <A> b
<B> => c
<C> => ϵ
```

Example:

```
<A> => <B> 0
<B> => <A> 1
<A> => ϵ
```

```
Start = <A>
```

String generated:

```
ϵ, 10, 1010, 101010, 10101010, …
```

Grammar GENERATES language = set of all strings derivable from applying production rules.

Type II Grammar (Context Free)

Limit production rules to have exactly one nonterminal on LHS, but anything on RHS.

```
<A> => b <B> <C> a <C>
<A> => <A> b c a <A>
```

Example:

```
<PAL> => 0 <PAL> 0          Start = <PAL>
      => 1 <PAL> 1
      => 0
       => 1
      => ϵ
```

String generated:

ϵ, 1, 0, 101, 001100, 111010010111, …

Type II Grammar (Context Free)

Example:

```
<S>  => ( <S> )                    Start = <S>
     => { <S> }
     => [ <S> ]
      => <S> <S>
     => ϵ
```

String generated:

ϵ, (), ()[()], (([]{()})[](())), …

Type I Grammar (Context Sensitive)

Add production rules of the type:

```
[A] <B> [C] => [A] a [C]
```

where [A] and [C] represent some fixed sequence of nonterminals and terminals.

```
<A> <B> <C> => <A> b <C>
<A> hi <B> <C> => <A> hib <C> <D>
```

Type 0 Grammar (Recursive)

No limit on production rules: at least one nonterminal on LHS.

Example:

Start = <S>

<S> => <S> <S>

<S> => <A> <B> <C>

<A> => a

<B> => b

<C> => c

<S> => ϵ

<A><B> => <B><A>

<B><A> => <A><B>

<A><C> => <C><A>

<C><A> => <A><C>

<B><C> => <C><B>

Strings Generated:

ϵ, abc, aabbcc, cabcab, acacacacacabbbbbb, …

Chomsky Hierarchy

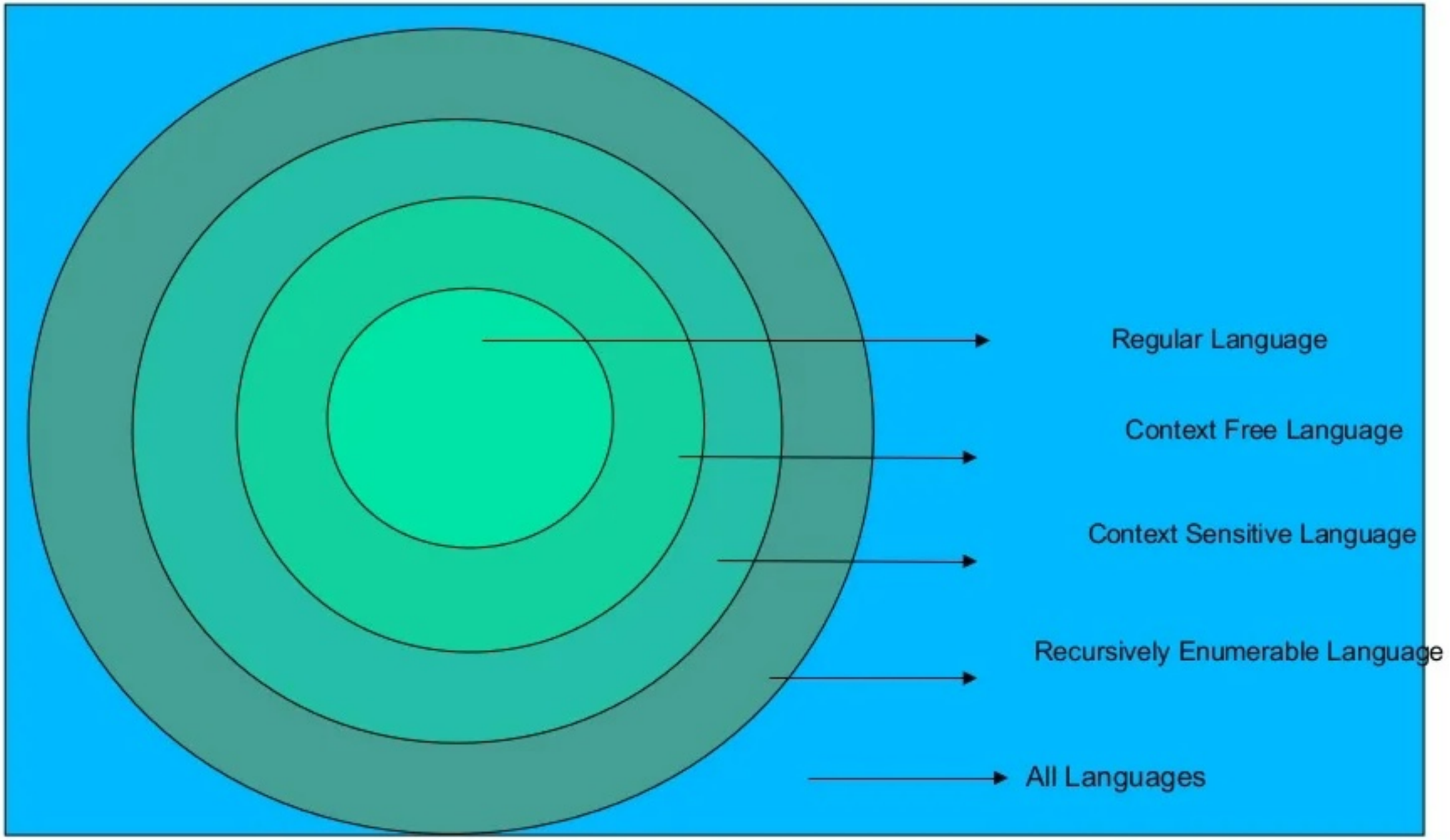
Powerful Machines

Type	Machine	Grammar
III	Finite State Automata	Regular
II	Push Down Automata	Context Free
I	Least Bound Automata	Context Sensitive
0	Turing Machine	Recursive

Expressive Languages

Essential one-to-one correspondence between machine and languages.

Chomsky Hierarchy



FSA and Type III Equivalence

FSA's and Type III grammar are equally powerful.

- Given an FSA, can construct Type III grammar to generate same language.
- Given Type III language, can construct FSA that accepts same language.

Proof Idea:

FSA	Type III Grammar
Start state	Start symbol
States	Nonterminals
Transition arcs	Production rules: <A> => <B> a
Accept state	Production rules: <A> => a

Compilers and Grammar

Compiler: translates program from high-language to native machine language.

Three basic phases.

- Lexical analysis (tokenizing).
  - convert input into "tokens" or terminal symbols
  - `# include <stdio.h> int main ( void ) { printf ( "Hello World!\n" ) ; return 0 ; }`
  - implement with FSA
  - Unix program lex

Compilers and Grammar

Compiler: translates program from high-language to native machine language.

Three basic phases.

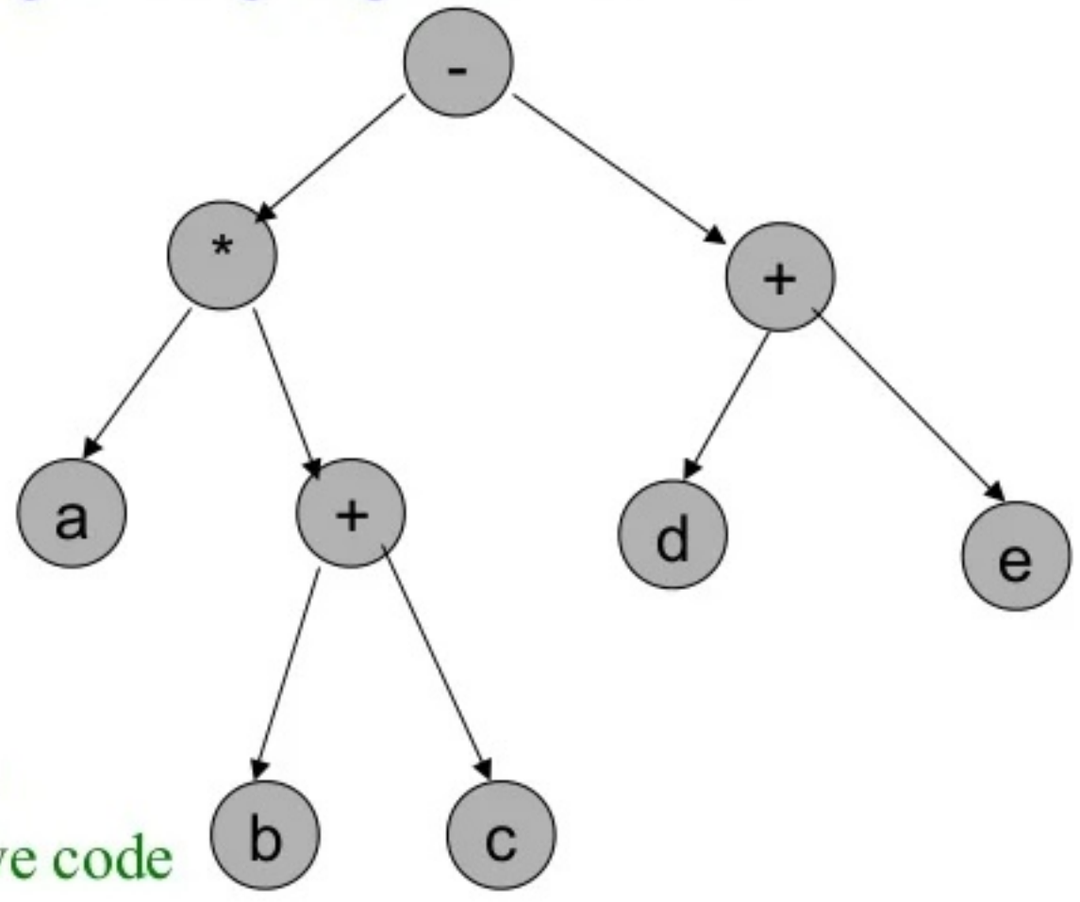
- Lexical analysis (tokenizing).
- Syntax analysis (parsing).
  - implemented using pushdown automata since C language is (almost) completely described with context-free grammar
  - Unix program `yacc`

Compilers and Grammar

Compiler: translates program from high-language to native machine language.

Three basic phases.

- Lexical analysis (tokenizing).
- Syntax analysis (parsing).
- Code generation.
  - parse tree gives structure of computation
  - traverse tree in postorder and create native code



Parse tree for expression:
`(a * (b + c)) - (d + e)`

Theory of Automata

- Dictionary defines ***Automaton*** as a machine/mechanism that operates as per encoded instructions.

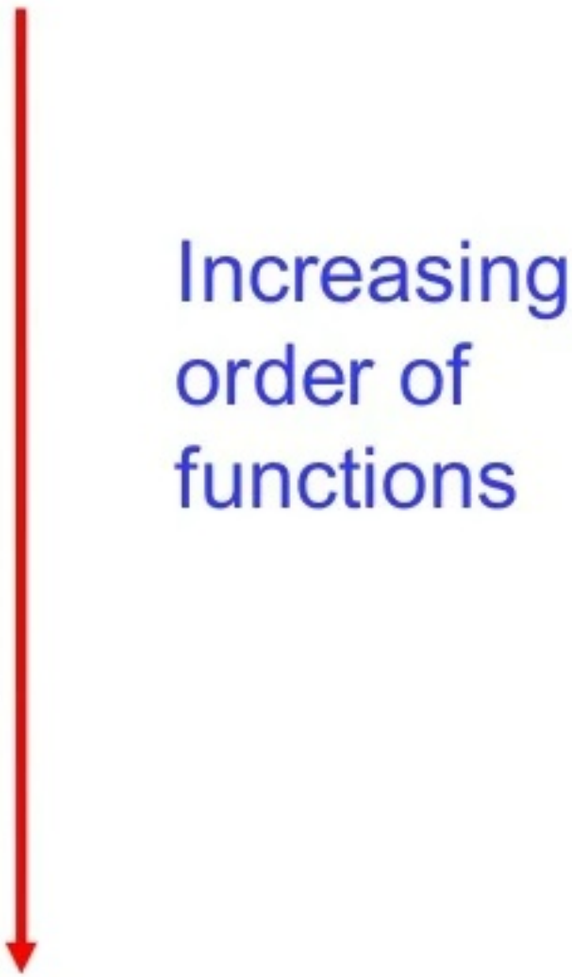
- ***Theory of Automata*** deals with the study of automata (machines), their states, and state transitions with respect to functions, without human intervention ie., as per encoded instructions.

Major Automata categories:

- Finite Automata
  - Deterministic Finite Automata (DFA)
  - Non-Deterministic Finite Automata (NFA)

- Pushdown Automata
  - Deterministic Pushdown Automata
  - Nondeterministic Pushdown Automata

- Turing Machine



Deterministic Finite Automata(DFA)

- A DFA consists of a finite set of states and a set of transitions from state to state that occur on input symbols chosen from an alphabet $\sum$.

DFA

- Formal Definition:

DFA A= (Q, $\sum$,δ, q0, F), where

Q is a finite set of states,

$\sum$ is a finite input alphabet

q0 $\in$ Q is the initial state

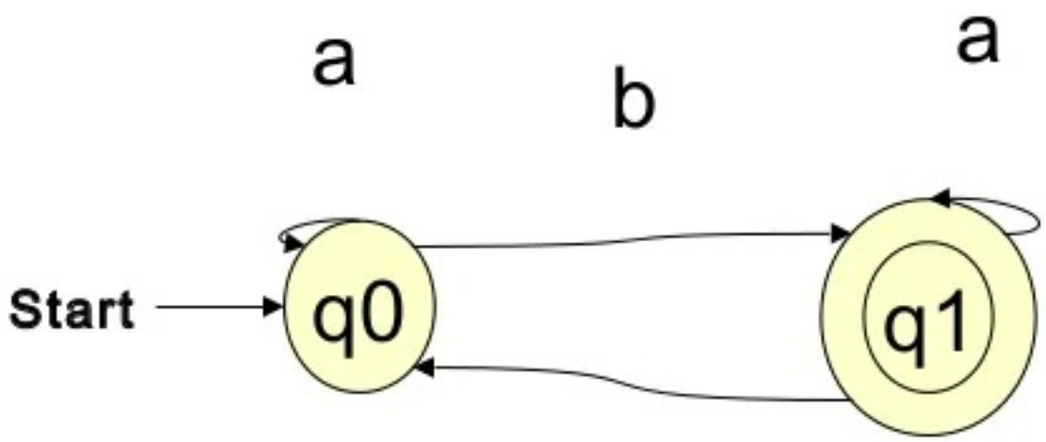
F $\in$ Q is the set of final states

δ is the transition function mapping Q X $\sum$ to Q

DFA

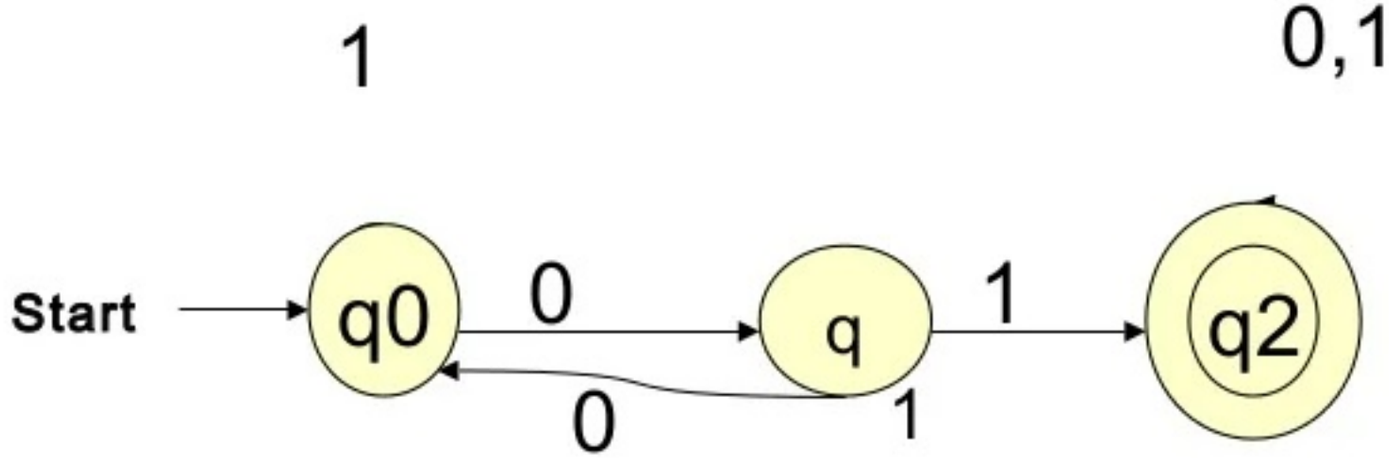
Draw the DFA for ,D= (Q, ∑,δ,S, F), when Q = {q0,q1}, ∑ = {a,b}, S = q0,F = {q1} and δ is given by

δ	a	b
→ q0	q0	q1
*q1	q1	q0



DFA

Draw DFA for $\{w : w \text{ is of the form } x01y, \text{ for some } x,y \in \{0,1\}\}$



So,

Q = {q0,q1,q2}

∑ = {0,1}

S = q0

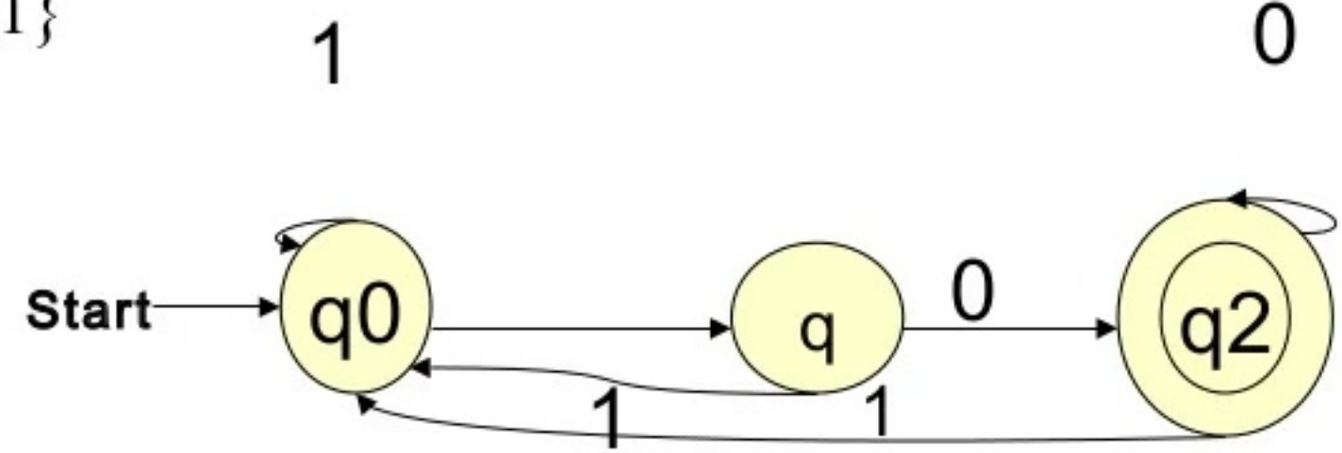
F = {q2}

δ	0	1
→ q0	q1	q0
q1	q0	q2
* q2	q2	q2

DFA

Give DFA for the set of all strings ending in 00, over the alphabet

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



So,

$Q = \{q0,q1,q2\}$

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

$S = q0$

$F = \{q2\}$

1

δ	0	1
→ q0	q1	q0
q1	q2	q0
* q2	q2	q0

DFA

- Language accepted by a DFA is called ***Regular language.***
- DFA is an FA with ***determinism.***
- ***Determinism*** means for any DFA there will be *one and only one transition* from every state for every symbol of $\sum$

NonDeterministic Finite Automata(NFA)

- NFA is an FA showing Nondeterminism.
- A NFA can be different from a DFA in that
  - for any input symbol, nondeterministic one can transit to more than one states.
  - for any input symbol, nondeterministic one may not transit to another state

NFA

- Formal Definition:

NFA A = (Q, $\sum$,δ, q0, F), where

Q is a finite set of states,

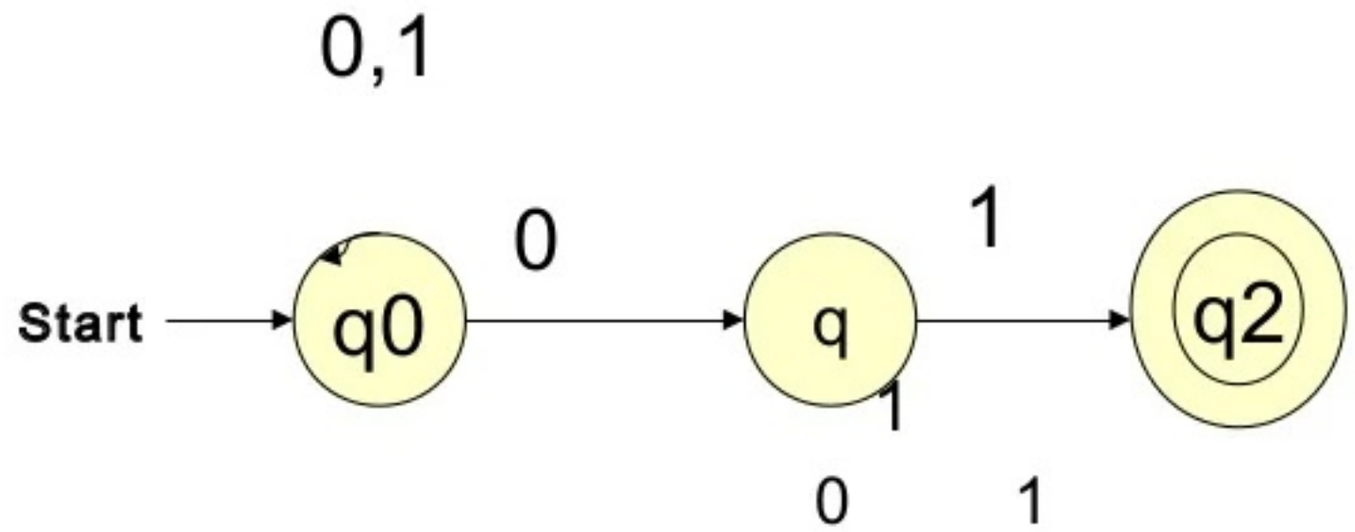
$\sum$ is a finite input alphabet

q0 $\in$ Q is the initial state

F $\subset$ Q is the set of final states

δ is the transition function mapping Q X $\sum^*$ -> Q

NFA



This diagram represents an NFA which accepts strings ending in 01, ie., **(0+1)*01**

NFA

- Corresponding Transition Diagram

	0	1
→ {q0}	{q0,q1}	{q0}
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2*}	{q0,q1}	{q0}

Equivalence of DFA and NFA

- *For every NFA there exists a DFA which simulates the behavior of NFA*

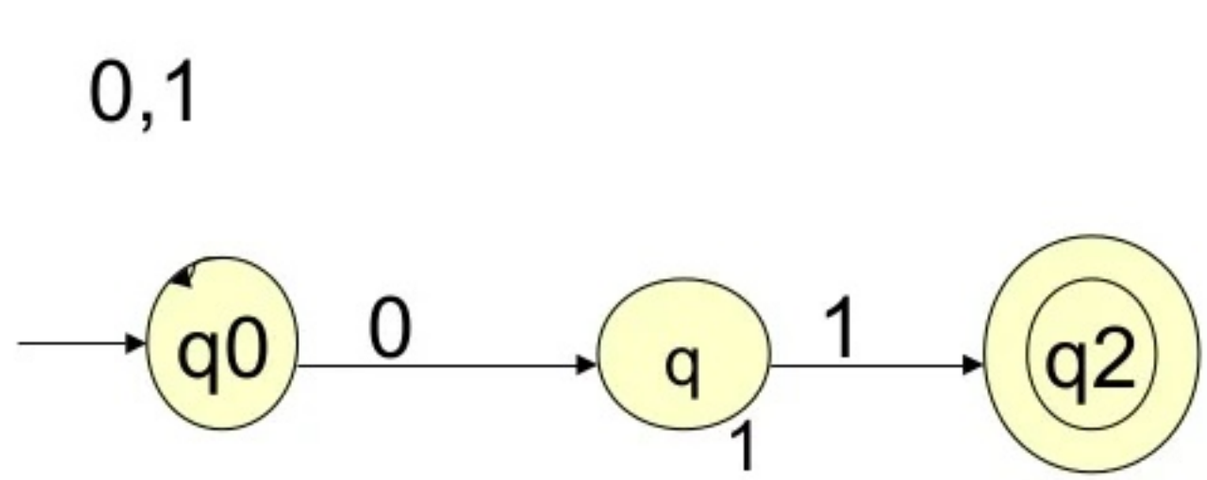
- Language accepted by NFA will be equal to that accepted by DFA.

- Language accepted by NFA is regular language.

How to construct Equivalent DFA for an NFA

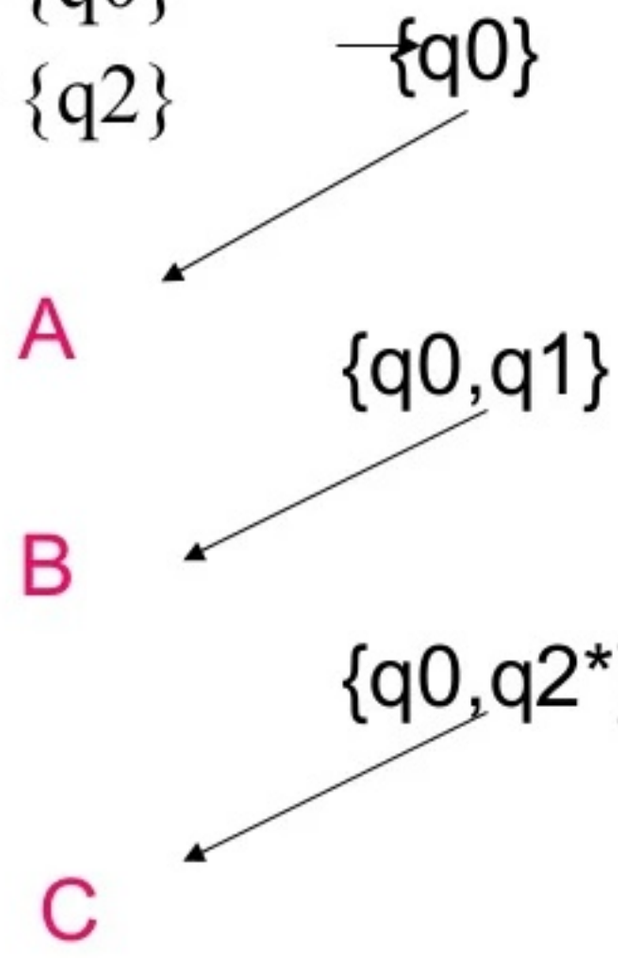
(1) Construct a DFA which accepts strings ending in 01

NFA



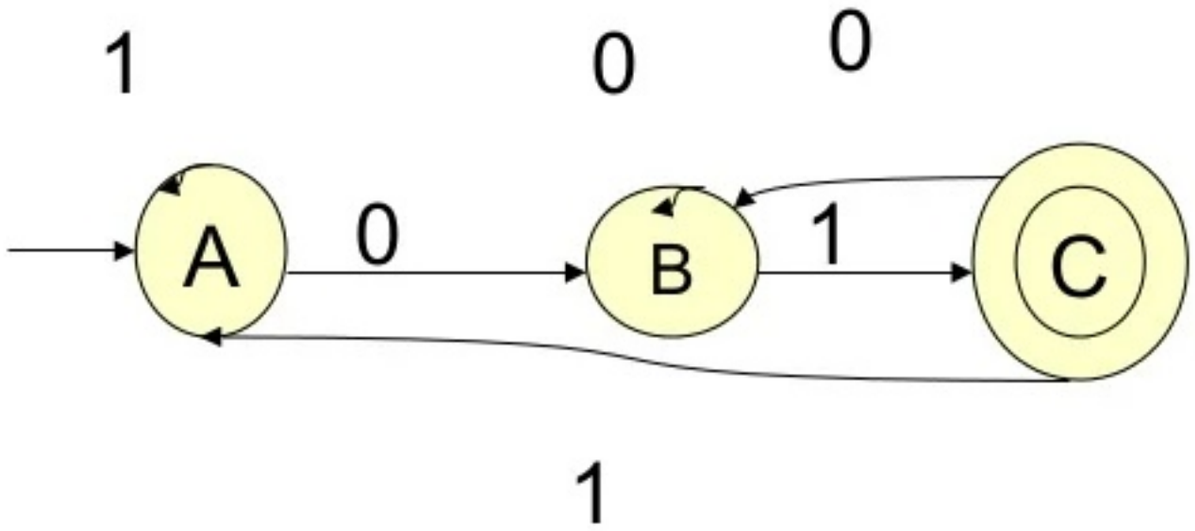
Transition Diagram

- Q = {q0,q1,q2}
- ∑ = {0,1}
- S = {q0}
- F = {q2}



	0	1
{q0}	{q0,q1}	{q0}
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2*}	{q0,q1}	{q0}

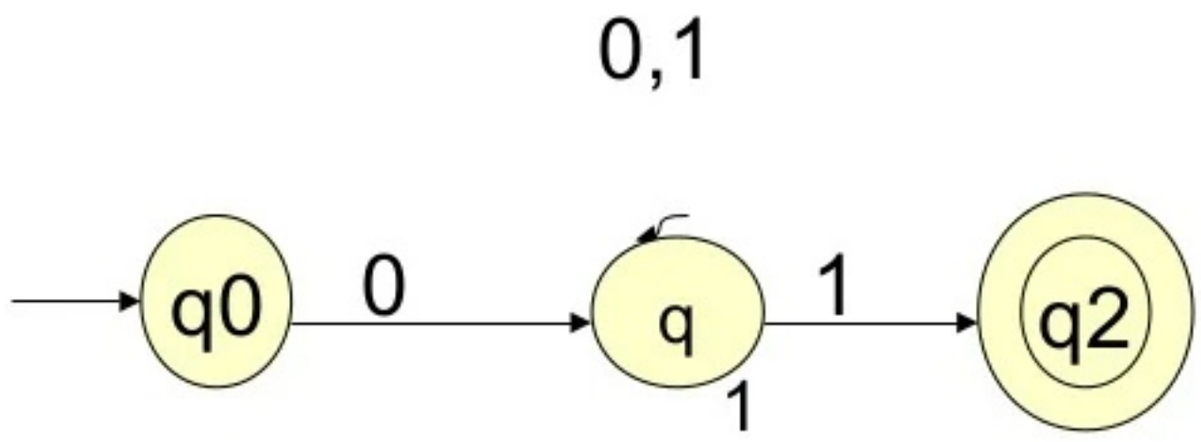
DFA



How to construct Equivalent DFA for an NFA

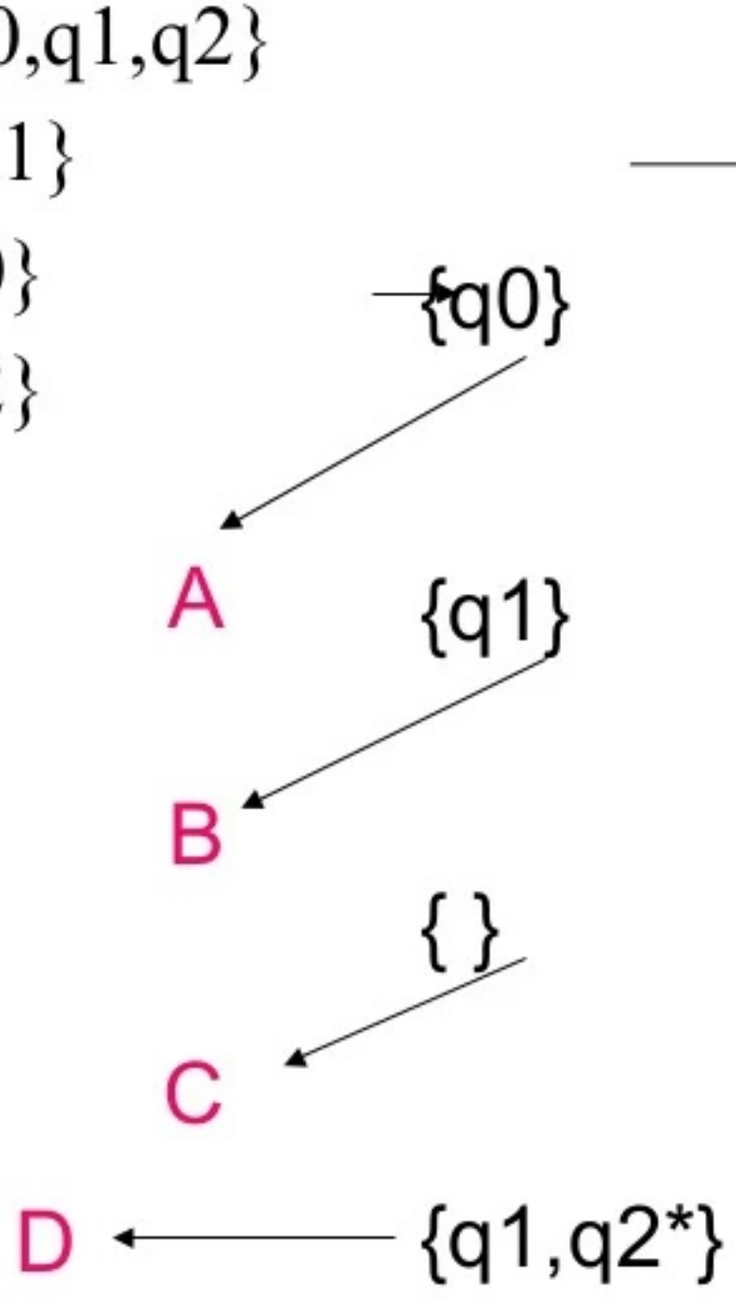
- (2) Construct an NFA which accepts 0(0+1)*1 and then convert it into corresponding DFA

NFA:



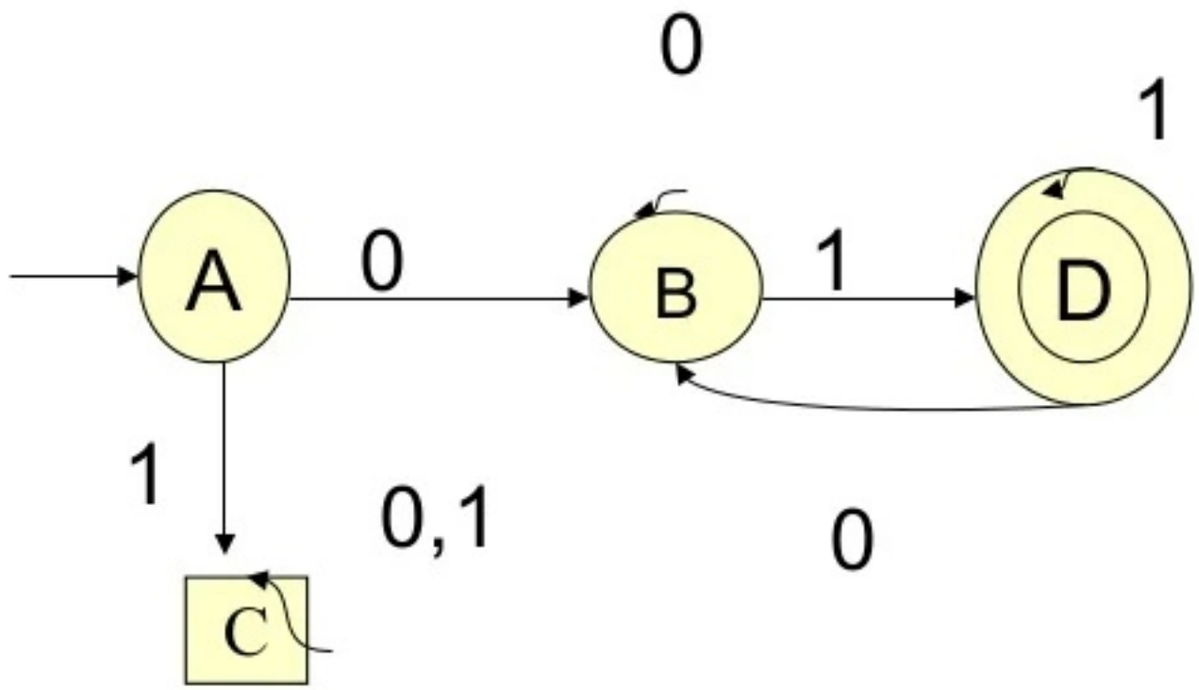
Transition Diagram (for 0(0+1)*1)

- Q = {q0,q1,q2}
- ∑ = {0,1}
- S = {q0}
- F = {q2}



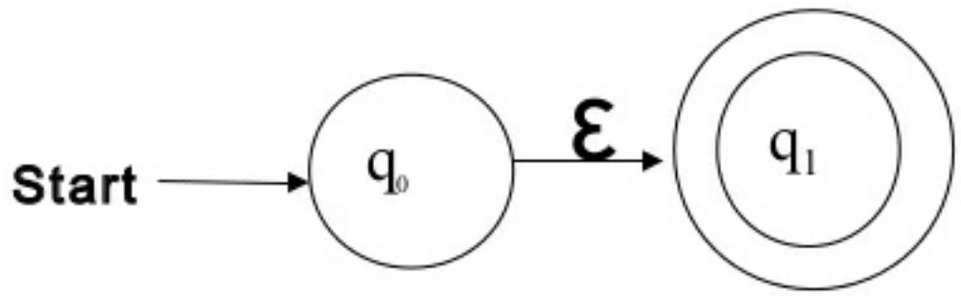
	0	1
{q0}	{q1}	{ }
{q1}	{q1}	{q1,q2}
{ }	{ }	{ }
{q1,q2*}	{q1}	{q1,q2}

DFA

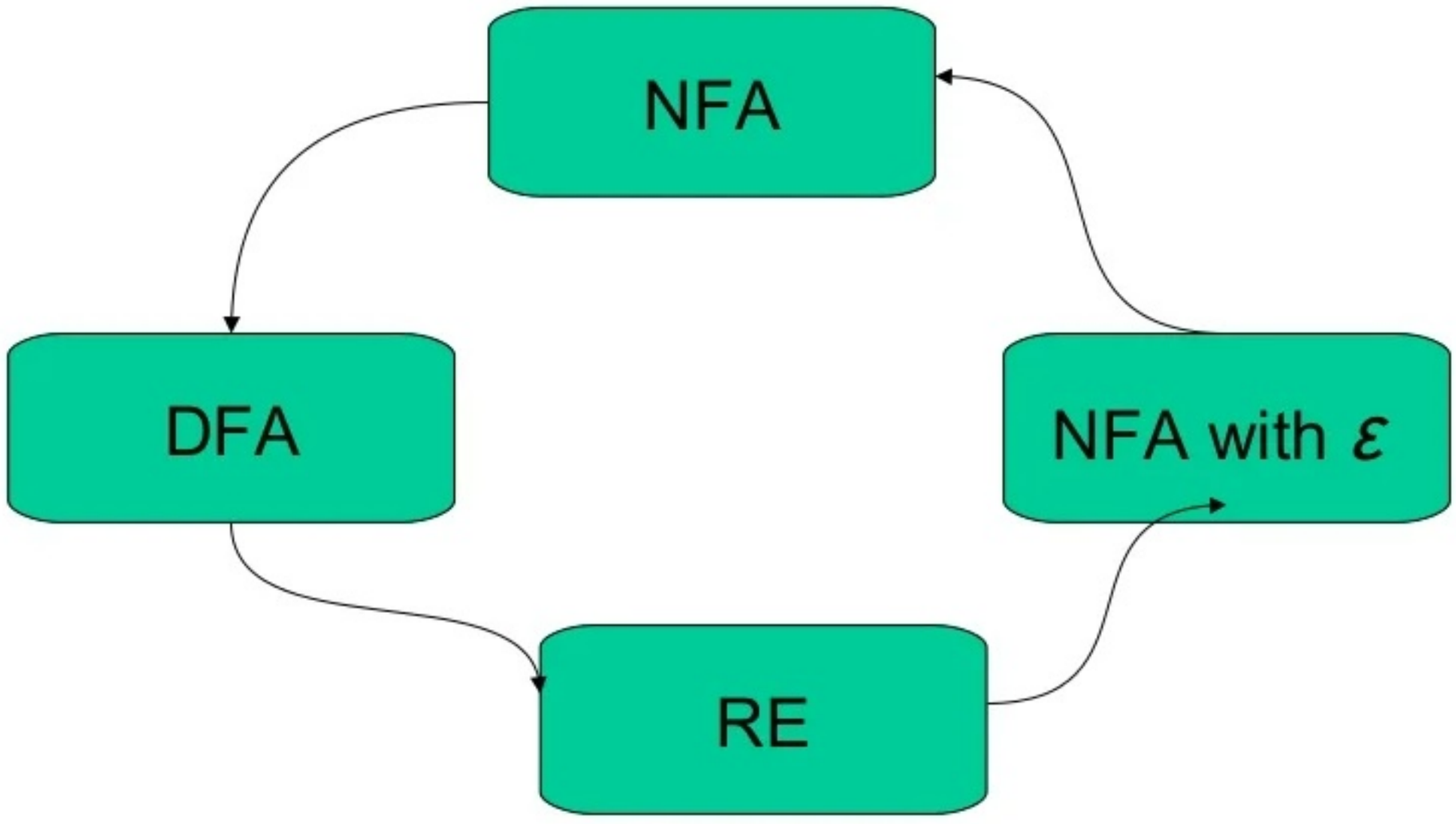


Finite Automata with Epsilon transitions

- NFAs are sometimes allowed to have a transition accepting ε string.
- Such NFAs are called FA with ε transition.



Equivalence relations



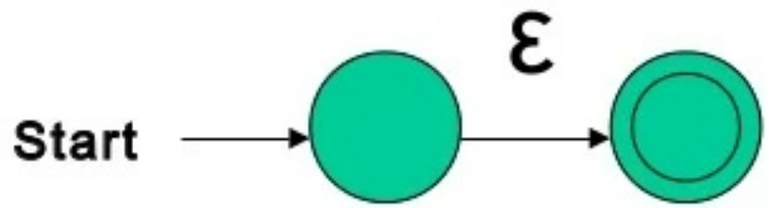
Equivalence of RE and FA

- Every language defined by a regular expression is also defined by a finite automaton.
- In other words, there exists a finite automaton for every regular expression.

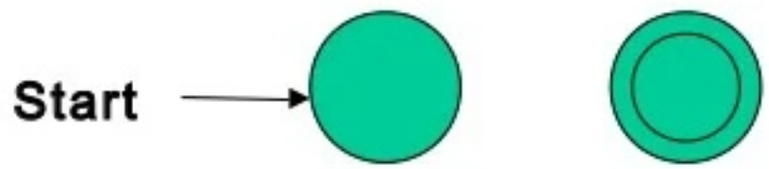
Equivalence of RE and FA (trivial cases)

Let R be a regular expression, L(R) be the language generated by R, then

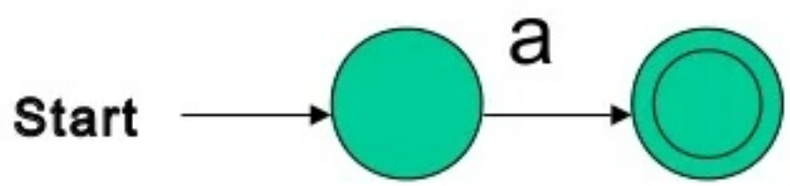
- $L(R) = \{\varepsilon\}$, $R = \varepsilon$



- $L(R) = \emptyset$



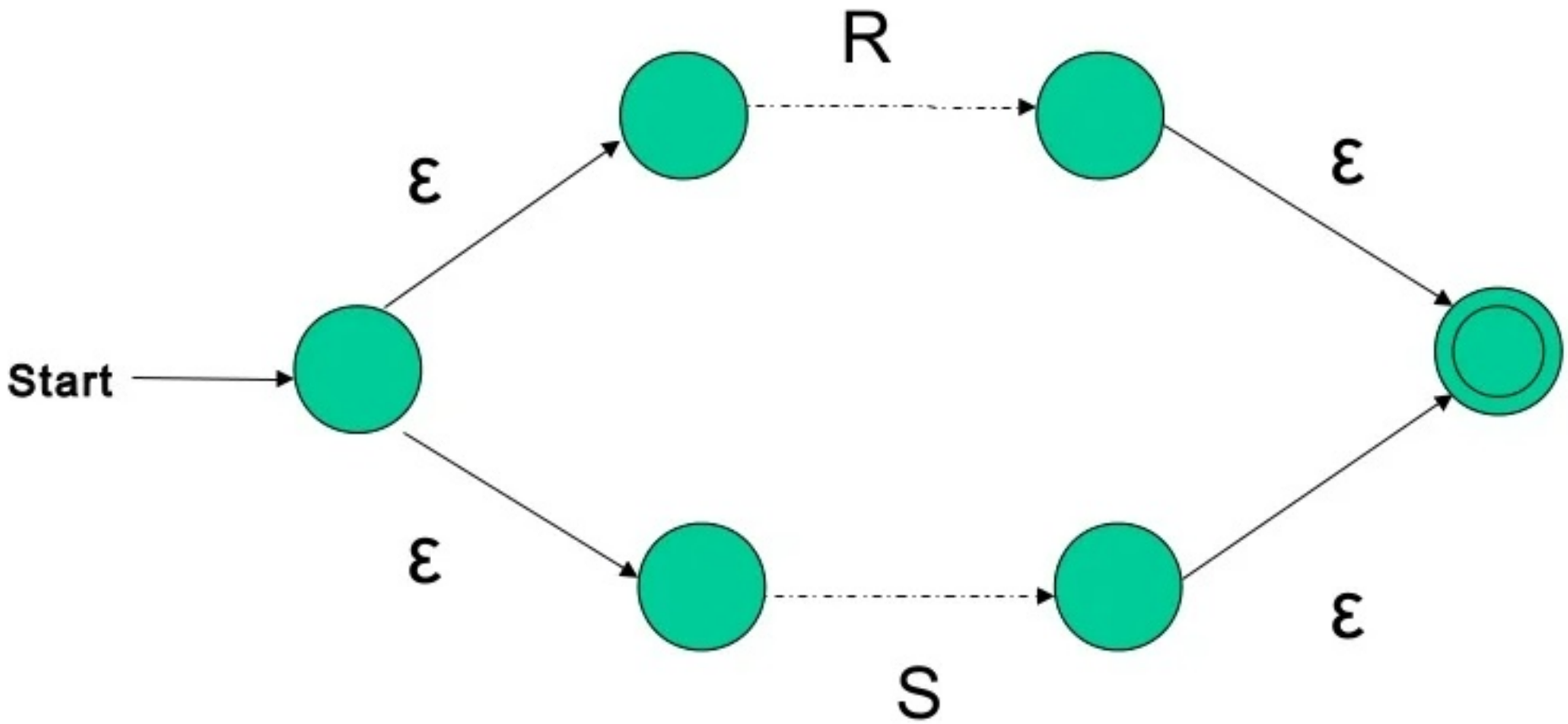
- $L(R) = L(a) = \{a\}$, $R = a$



Equivalence of RE and FA

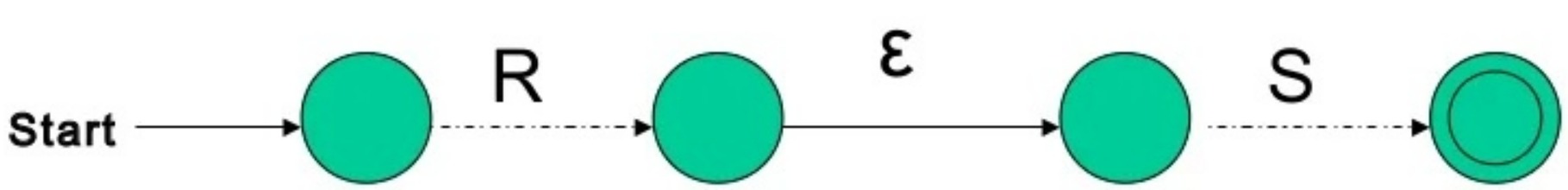
Let R and S be regular expressions, then:

1. Let M = R+S, then L(M) =L(R) U L(S)



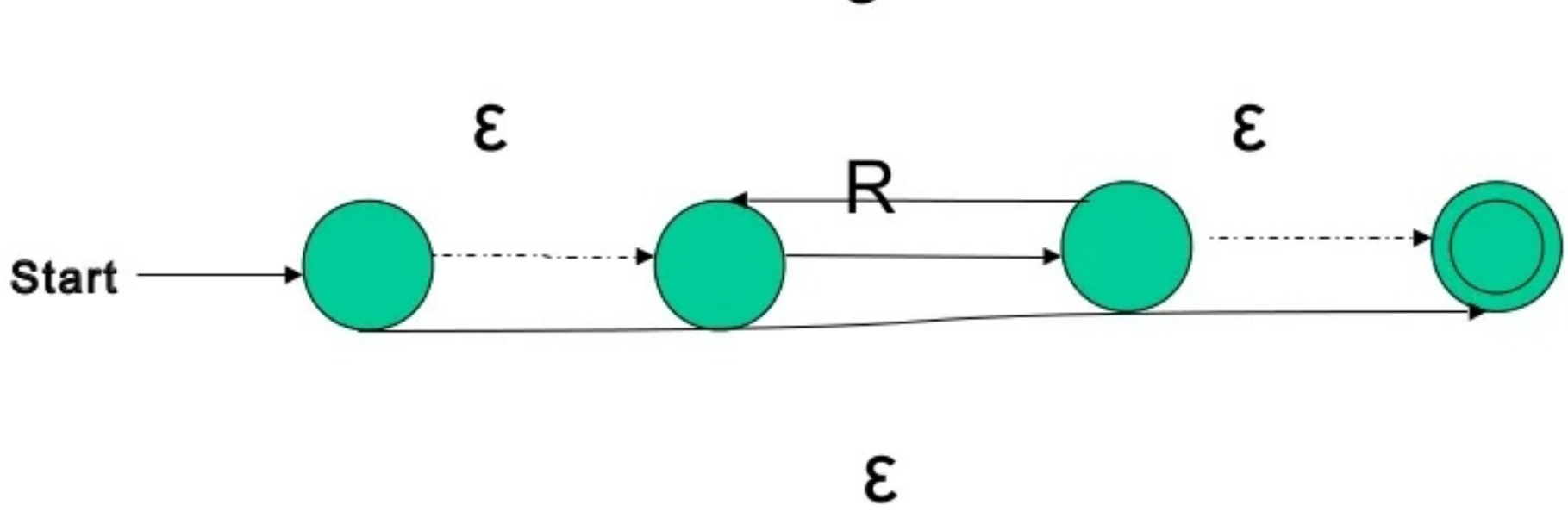
Equivalence of RE and FA

1. Let M = R.S, then L(M) =L(R)L(S)



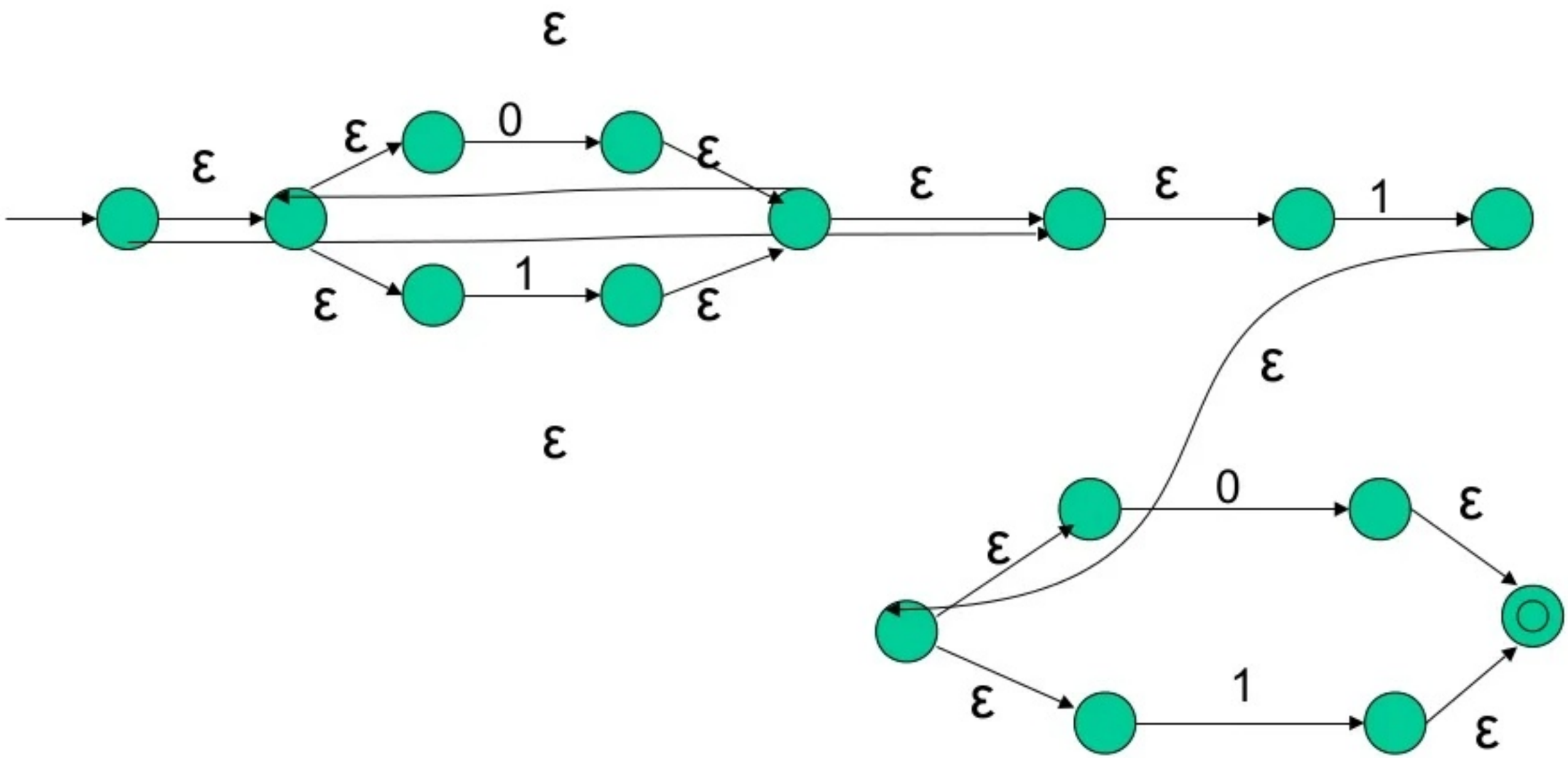
Equivalence of RE and FA

3. Let M = R*, then L(M) =L(R)UL(R)UL(R)UL(R)…



Example

Convert the RE = (0+1)*1(0+1) to an ε-NFA



Algebraic laws for Regular Expressions

If L, M, N are 3 regular expressions, then

1. Commutative Law for Union – L +M = M + L

1. Associative Law for Union – (L+M) + N = L + (M + N)

1. Associative Law for concatenation – (LM)N = L(MN)

Algebraic laws for Regular Expressions

- $\emptyset + L = L + \emptyset = L$, $\emptyset$ is the *identity* for union.
- $\varepsilon L = L\,\varepsilon = L$, ε is the *identity* for concatenation.
- $\emptyset L = L\emptyset = \emptyset$, $\emptyset$ is the *annihilator* for concatenation.

Algebraic laws for Regular Expressions

1. Distributive Law –

$$L(M+N) = LM + LN$$

$$(M+N)\ L = ML + NL$$

1. Idempotent law -

$$L + L = L$$

1. Laws involving closures –

$$(L^*)^* = L^*$$

$$\emptyset^* = \varepsilon$$

$$\varepsilon^* = \varepsilon$$

Common Regular Expression Rules

If P, Q ,R are regular expressions, then

(1) (PQ)*P = P(QP)*

(2) R*R* = R*

(3)If R = Q + PR, then R = QP* (Arden's theorem)

(4) (P+Q)* = (P*Q*)*

Pumping Lemma (for Regular Languages)

- Not every language is a regular language

Eg: $L_{01} = \{0^n1^n \mid n>= 1\}$

- L_{01} is a language that consists of the strings comprising n number of zeroes followed by n number of ones.
- If there is an FA which accepts L_{01} then it should,

  - accept n number of zeroes from start state
  - remember n, so that it can accept <u>exactly</u> n number of ones
  - accept n number of ones and reach the final state.
- But FA lacks the capability to remember value n (lacks memory), so cannot accept L_{01}.
- L_{01} cannot be accepted by any FA => L_{01} is not regular => there exists languages which are not regular.

Pumping Lemma (for Regular Languages)

(The *pumping lemma for regular languages*) Let L be a regular language. Then there exists a constant n (which depends on L) such that for every string *w* in L such that $|w| \geq n$, we can break *w* into three strings, $w=xyz$, such that

1) $y \mathrel{!}= \epsilon$
2) $|xy| \leq n$.
3) For all $k \geq 0$, the string xy^kz is also in L.

Pumping Lemma (for Regular Languages)

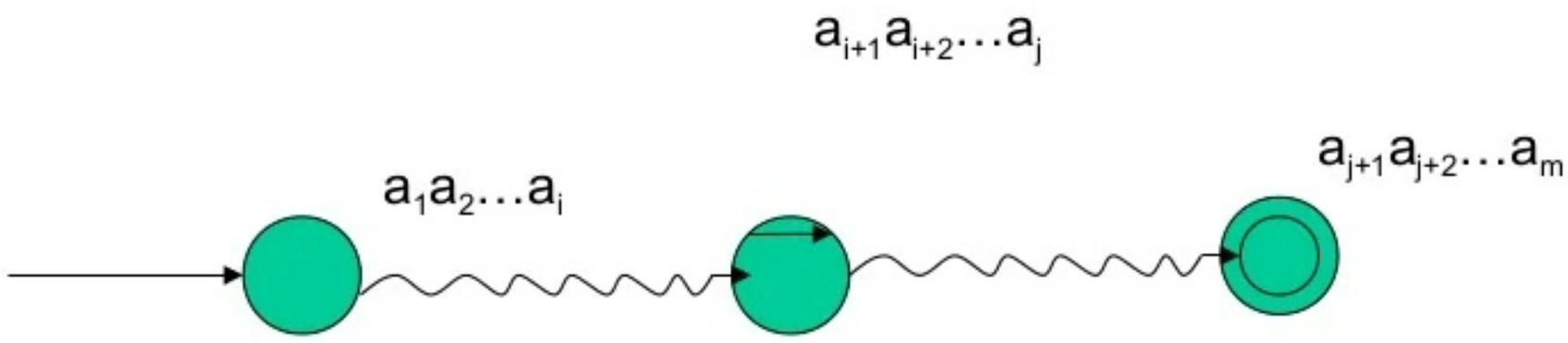
Say, $w = a_1 a_2 \ldots a_m$

let, $x = a_1 a_2 \ldots a_i$

$y = a_{i+1} a_{i+2} \ldots a_j$

$z = a_{j+1} a_{j+2} \ldots a_m$

Then the FA can be viewed as,



Pumping Lemma (for Regular Languages)

Prove that $L_{01} = \{0^n1^n \mid n >= 1\}$ is not regular.

Assume that L_{01} is regular.

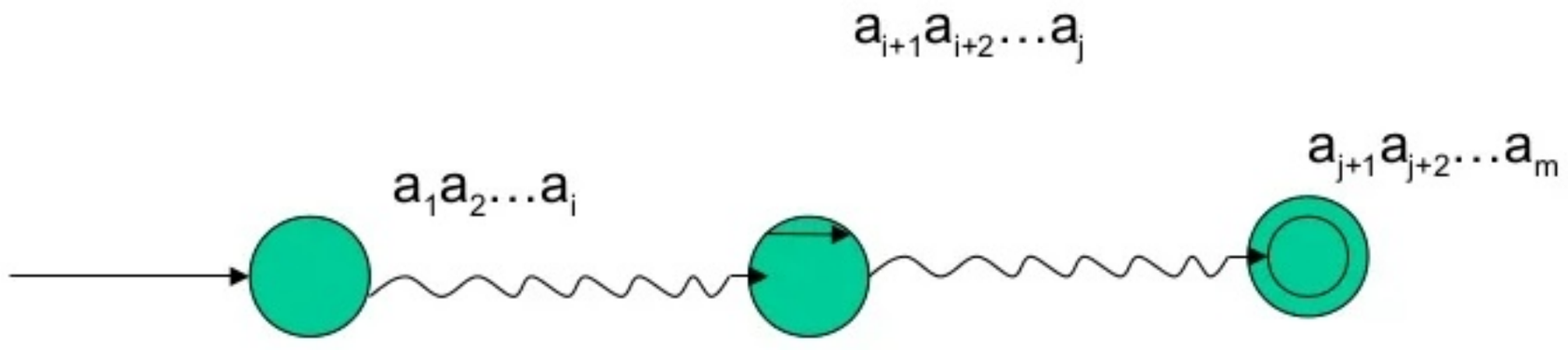
So by Pumping lemma, we can have $n \geq 0$, such that

$w \in L_{01}$, $|w| = n$ and we can break w into three sub strings x, y , z such that,

$w = xyz$ such that, $y \mathrel{!}= \epsilon$, $|xy| \leq n$ and $xy^kz \in L_{01}$.

Assume, $xy=0^n$ and $z=1^n$.

So FA can be drawn as,



Pumping Lemma (for Regular Languages)

Here length of y is k, $k \geq 0$.

When k=0, $xz \in L_{01}$, => $|x|=|z|$ =n ----------→(1)

When k=1, xyz must be an element of L_{01}, => $|xy|=|z|$,

but from (1), $|x|=|z|$, and also $y \neq \varepsilon$,

which is a contradiction.

So the assumption that the language, L_{01} is regular is not correct.

Examples of languages, not regular

- $L = \{ 0^{i2} \mid i>=0\}$
- $L_{pr} = \{0^p \mid p \text{ is a prime number}\}$
- $L = \{0^n 1 0^n \mid n>=1\}$
- $L = \{0^n 1^m \mid n<=m\}$
- $L = \{0^n \mid n \text{ is a perfect cube}\}$

Finite state Machine with Output

- Mealy Machine :- Outputs corresponds to transition between states
- Moore Machine :- Output determined only by state.

Mathematical Representation of Mealy Machine

$M = (Q, \Sigma, \lambda, \Delta, q_0, \delta)$

where

Q =A nonempty finite set of state in M.

Σ = A nonempty finite set of input symbols.

Δ = A nonempty finite set of outputs.

δ = It is the transition function which takes two arguments input state and input symbol.

q_0 = Initial state of M.

λ = It is a mapping function which maps $Q * \Sigma$ to Δ giving output associated with each transition.

Representation of Mealy Machine

Let M be the Mealy Machine and $a_1, a_2, a_3, \ldots a_n$ be input symbols where $n > 0$ then output of M is $\lambda(q_0, a_1), \lambda(q_1, a_2), \lambda(q_2, a_3), \lambda(q_3, a_4), \lambda(q_4, a_5) \ldots \lambda(q_{n-1}, a_n)$, such that

$\delta(q_{i-1}, a_i) = q_i \quad$ for $1 < i < n$.

Example of Mealy Machine

Transition Table

Present State	Input a = 0		Input a = 1	
	State	output	State	Output
q_1	q_3	0	Q_2	0
q_2	q_1	1	q_4	0
q_3	q_2	1	q_1	1
q_4	q_4	1	q_3	0

Values of Q, Σ, Δ

- $Q = \{q1, q2, q3, q4\}$
- $\Sigma = \{0, 1\}$
- $\Delta = \{0, 1\}$

$\lambda(q1, 0) = 0$ $\delta(q1, 0) = q3$

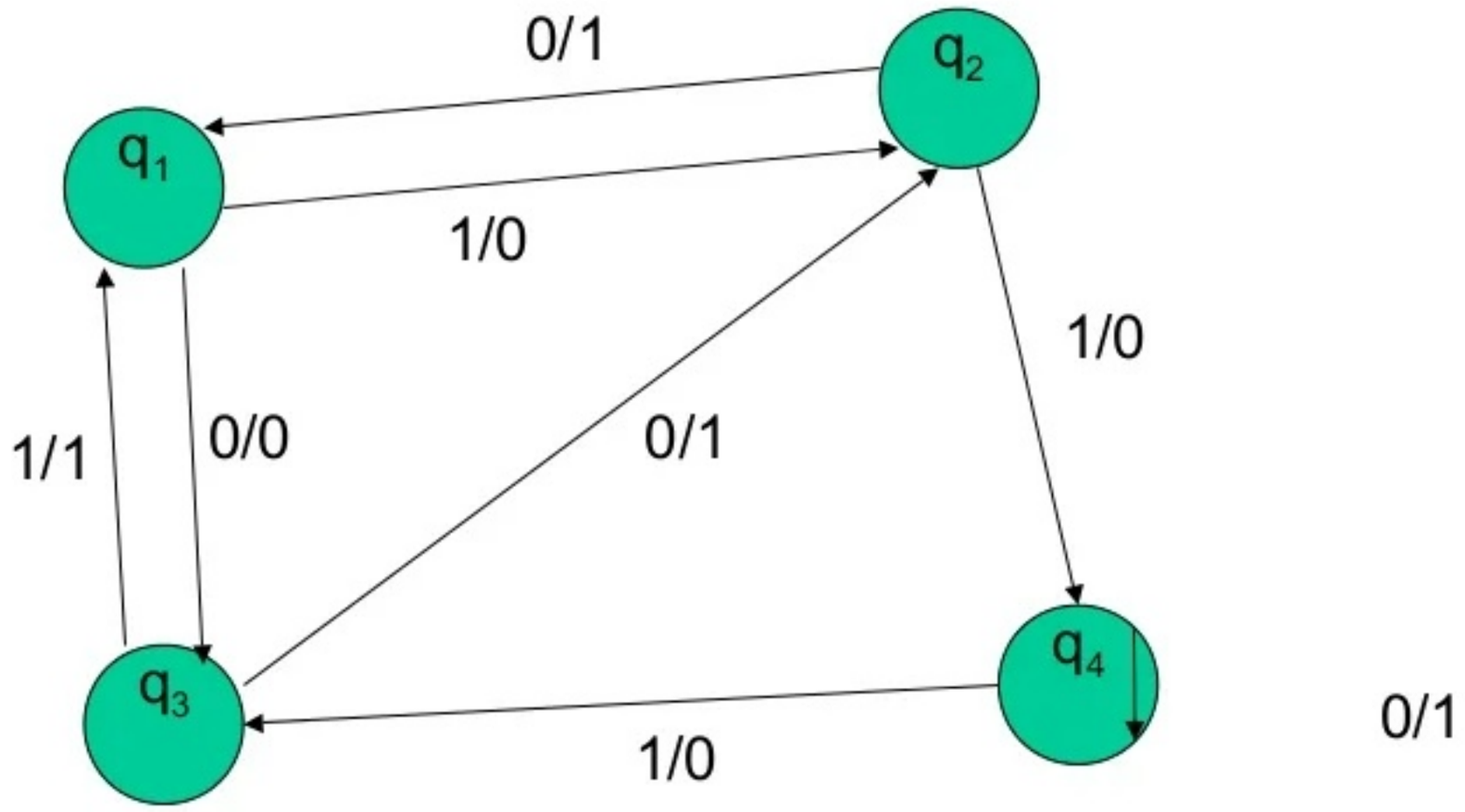
$\lambda(q2, 0) = 1$ $\delta(q2, 0) = q1$

$\lambda(q3, 0) = 1$ $\delta(q3, 0) = q2$

$\lambda(q4, 0) = 1$ $\delta(q4, 0) = q4$

$\lambda(q1, 1) = 0$ $\delta(q1, 1) = q2$

Mealy Machine Transition Diagram



e.g. In 1/1, a = 1 and λ(q, a) = 1

Mathematical Representation of Moore Machine

$M = (Q, \Sigma, \lambda, \Delta, q_0, \delta)$

where

Q =A nonempty finite set of state in M.

Σ = A nonempty finite set of input symbols.

Δ = A nonempty finite set of outputs.

δ = It is the transition function which takes two arguments input state and input symbol.

q_0 = Initial state of M belongs to Q.

λ = It is a mapping function which maps Q to Δ giving output associated with each state.

Representation of Moore Machine

- Let M be a Moore machine and a1 , a2 , a3 , …….a_n be input symbols where n>0 then output of M is $\lambda(q_1)$, $\lambda(q_2)$, $\lambda(q_3)$, $\lambda(q_4)$ …… $\lambda(q_n)$, such that
- $\delta(q_{i-1}, a_i) = q_i$ for $1<i<n$.

Example of Moore Machine

Transition Table

Present State	Input a = 0	Input a = 1	Output
q_0	q_3	q_1	0
q_1	q_1	q_2	1
q_2	q_2	q_3	0
q_3	q_3	q_0	0

Values of
Q, Σ, Δ

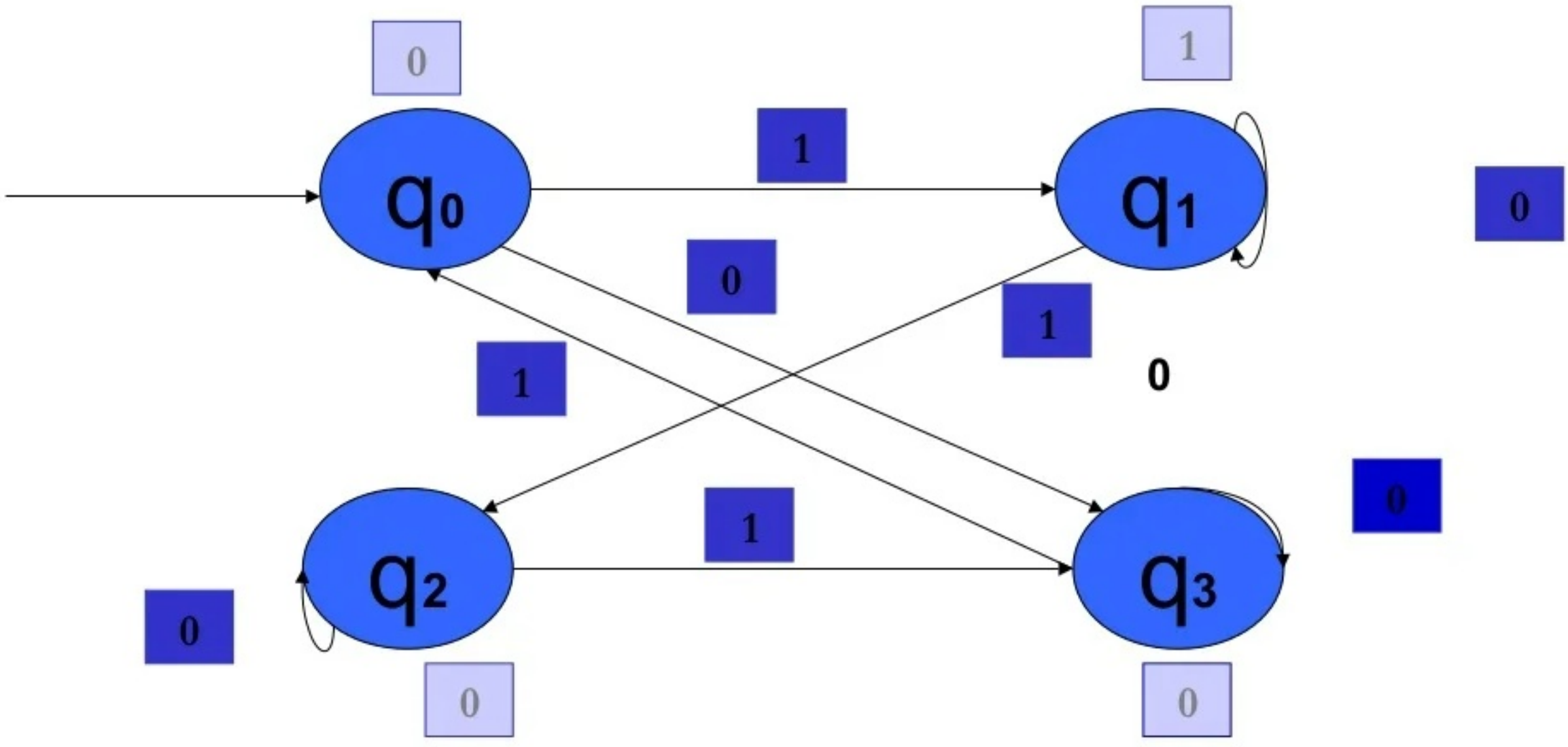
$$Q = \{q_0, q_1, q_2, q_3\}$$

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

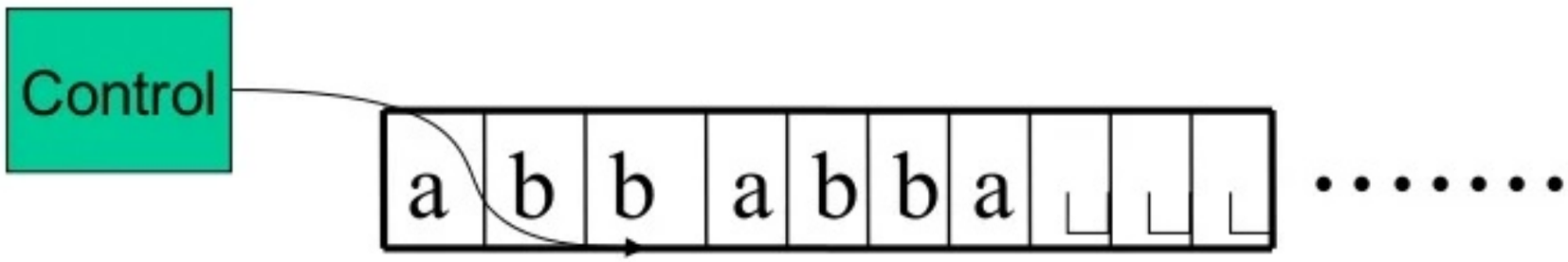
$$\Delta = \{ 0 , 1 \}$$

$$\lambda(q_0) = 0 , \ \lambda(q_1) = 1 , \ \lambda(q_2) = 0 , \ \lambda(q_3) = 0$$

Transition Diagram



Turing Machines



- Infinite tape to the right
- Finite alphabets for input and tape
- Finite number of states
- Can read from and write to the tape
- Can accept or reject at any time
- May run forever

Turing Machine

Designed to satisfy three criteria:

- They should be automata, with the same general characteristics as DFAs and PFAs
- They should be as simple as possible to describe and reason about
- They should be as general as possible in terms if the computations they can carry out

Turing Machine

- A Turing Machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where
- 1. Q is the set of states,
- 2. Σ is the input alphabet not containing the blank symbol
- 3. Γ is the tape alphabet, where $\in \Gamma$ and $\Sigma \subseteq \Gamma$,
- 4. $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
- 5. $q_0 \in Q$ is the start state,
- 6. $q_{accept} \in Q$ is the accept state,
- 7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Turing Machine

- The input is placed on the leftmost squares of the tape, so the first blank symbol marks the end of the input.
- The head starts on the leftmost square.
- The head cannot move off the left end of the tape.
- The machine runs until it enters the accept or reject state; or it runs forever.
- For a given state and tape symbol, the transition function specifies the next state, the symbol written on the tape to replace the scanned symbol, and a direction of movement of the head, L or R.

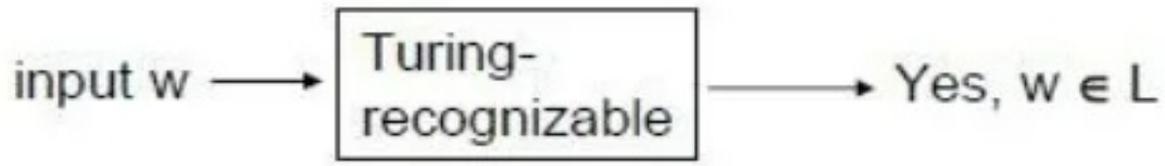
Turing Machine

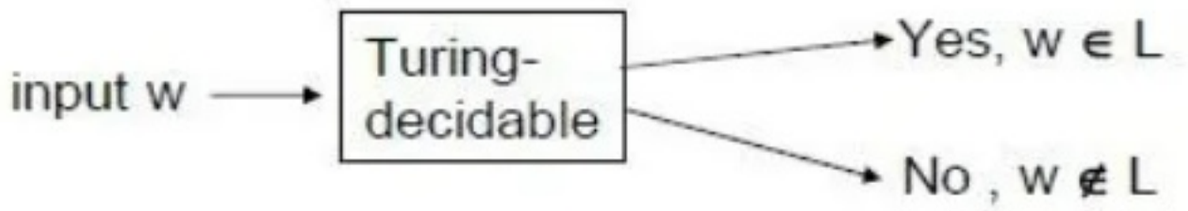
- The collection of strings accepted by a Turing machine M is called the language of M, or the language recognized by M, and is denoted by L(M).
- A language is Turing-recognizable if some Turing machine recognizes it.
- A Turing machine may reject an input by entering the qreject state or by looping forever. Note that it is difficult to distinguish a machine that is looping from one that is taking a long time to accept.
- A machine that halts on all input is called a decider and is said to decide its language.
- A language is Turing-decidable (or simply decidable) is some Turing machine decides it.

Definitions

Turing-recognizable languages are also called recursively enumerable languages.

Turing-decidable are also called recursive languages.





References

- Introduction to Automata Theory, Languages, and Computation

  John E Hopcroft, Rajeev Motwani & Jeffrey D Ullman
- An Introduction to Formal Languages and Automata, Peter Linz
- Theory of Computer Science (Automata, Languages and Computation) ,K L P Mishra & N Chandrasekaran.
- Discrete Mathematics and it's applications, Kenneth H. Rosen.