	THEORY OF COMPUTATION		Semester	V
	Course Code	BCS503	CIE Marks	50
	Teaching Hours/Week (L: T:P: S)	(3:2:0:0)	SEE Marks	50
	Total Hours of Pedagogy	50	Total Marks	100
	Credits	04	Exam Hours	3
	Examination type (SEE)	Theory		

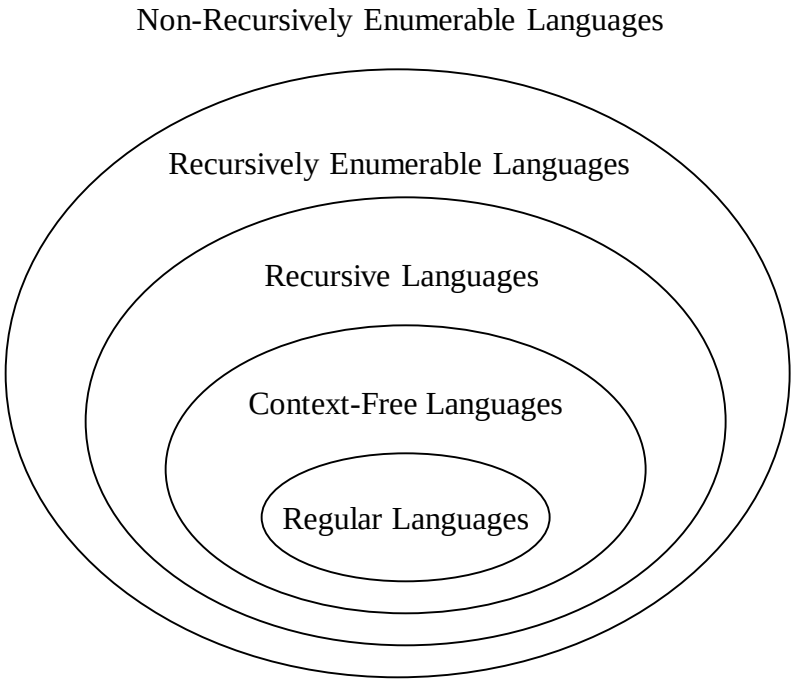
Course outcome (Course Skill Set)

At the end of the course, the student will be able to:

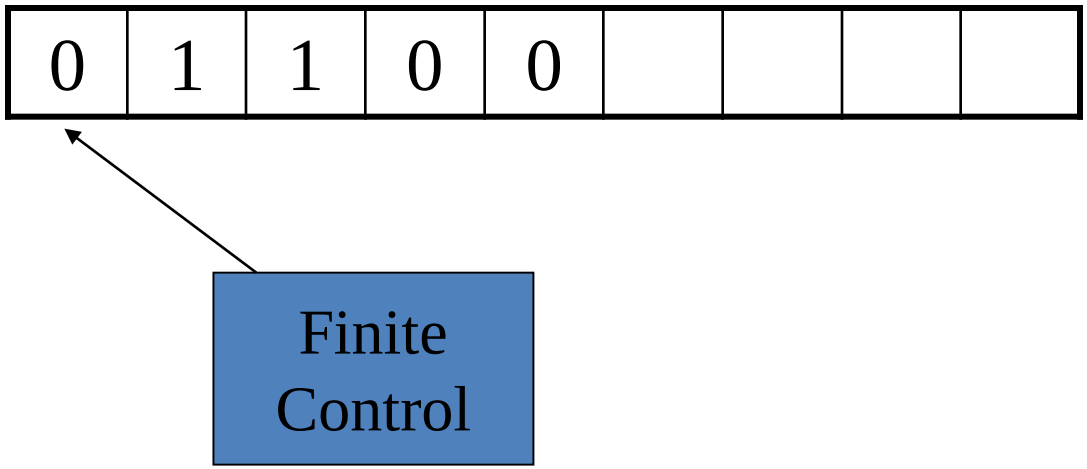
1. Apply the fundamentals of automata theory to write DFA, NFA, Epsilon-NFA and conversion between them.
2. Prove the properties of regular languages using regular expressions.
3. Design context-free grammars (CFGs) and pushdown automata (PDAs) for formal languages.
4. Design Turing machines to solve the computational problems.
5. Explain the concepts of decidability and undecidability.

	Module-1	10 Hours
	Introduction to Finite Automata, Structural Representations, Automata and Complexity. The Central Concepts of Automata Theory. Deterministic Finite Automata, Nondeterministic Finite Automata, An Application: Text Search, Finite Automata with Epsilon-Transitions. **TEXT BOOK: Sections 1.1, 1.5, 2.2,2.3,2.4,2.5**	

Hierarchy of languages



Deterministic Finite State Automata (DFA)

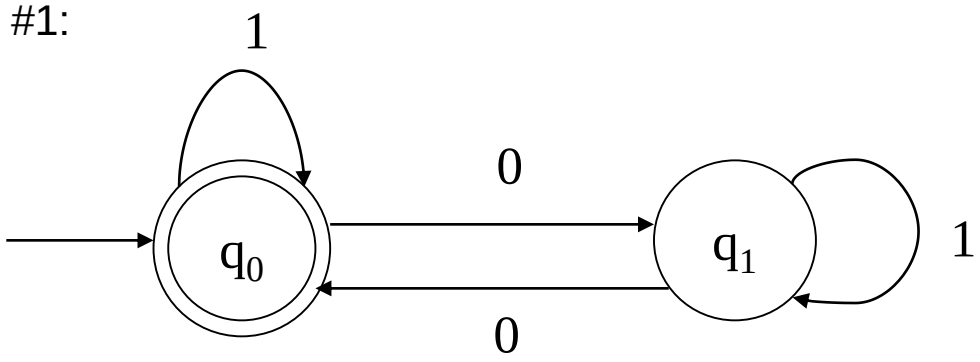


……..

One-way, infinite tape, broken into cells
One-way, read-only tape head.
Finite control, i.e.,
finite number of states, and
transition rules between them, i.e.,
a program, containing the position of the read head, current symbol being scanned, and the current "state."
A string is placed on the tape, read head is positioned at the left end, and the DFA will read the string one symbol at a time until all symbols have been read. The DFA will then either *accept* or *reject* the string.

The finite control can be described by a transition diagram or table:

Example #1:



1 0 0 1 1

q_0 q_0 q_1 q_0 q_0 q_0

One state is final/accepting, all others are rejecting.
The above DFA accepts those strings that contain an even number of 0's, including the *null* string, over *Sigma* = {0,1}

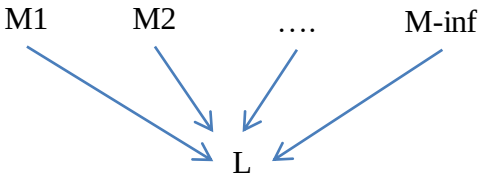
L = {all strings with zero or more 0's}

Note, the DFA must reject all other strings

Note:

- *Machine is for accepting a language, language is the purpose!*

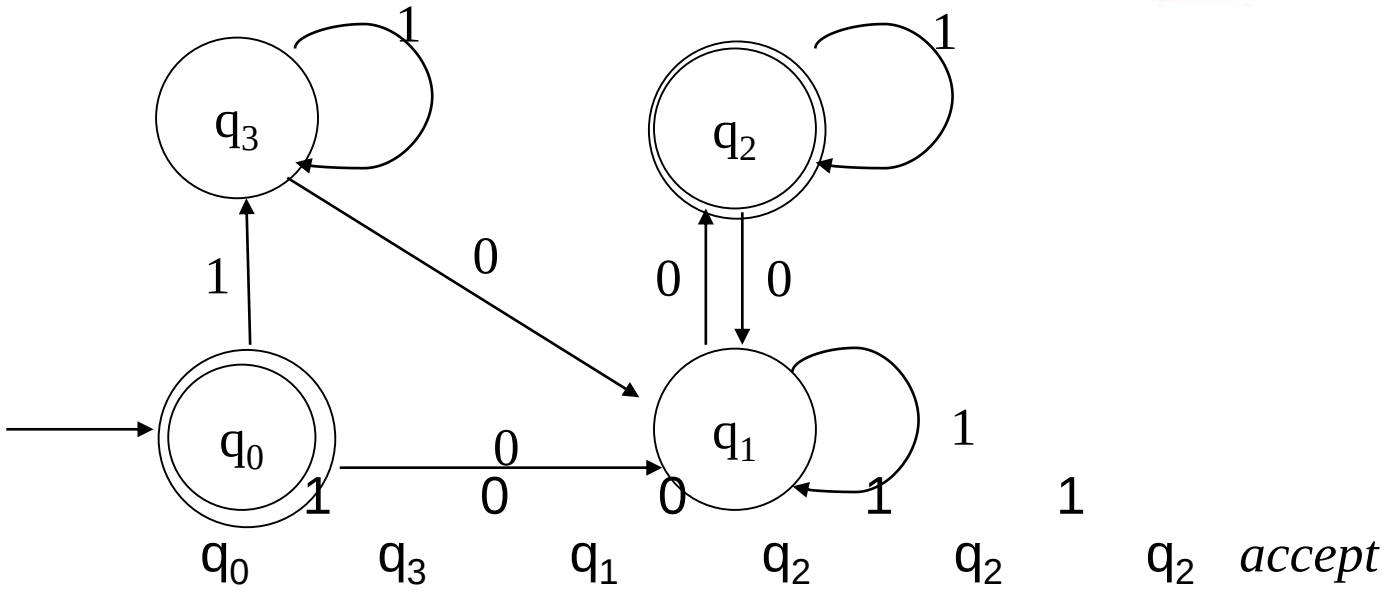
- *Many equivalent machines may accept the same language, but a machine cannot accept multiple languages!*



- *Id's of the characters or states are irrelevant, you can call them by any names!*

  Sigma = {0, 1} ≡ {a, b}

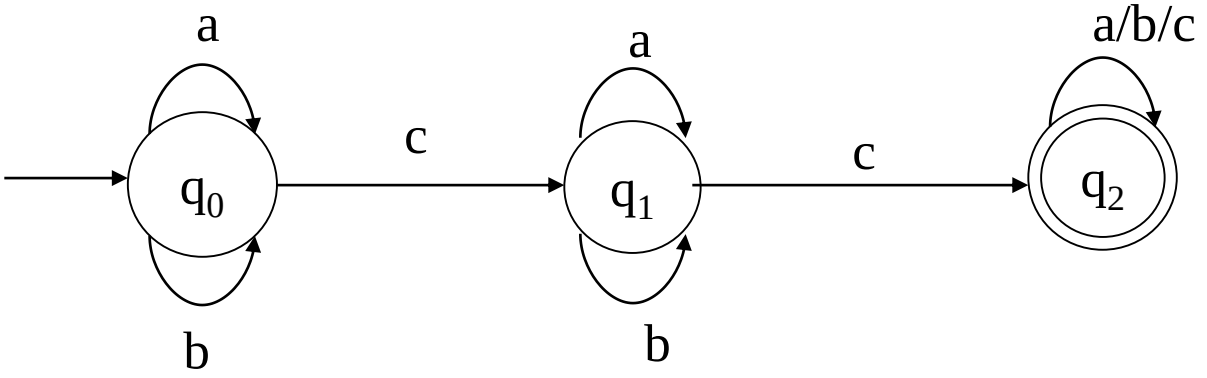
  States = {q0, q1} ≡ {u, v}, as long as they have identical (isomorphic) transition table



string

One state is final/accepting, all others are rejecting.
The above DFA accepts those strings that contain an even number of 0's, including null string, over *Sigma* = {0,1}
Can you draw a machine for a language by excluding the null string from the language? L = {all strings with 2 or more *0*'s}

Example#2:



a c c c b

<u>accepted</u>

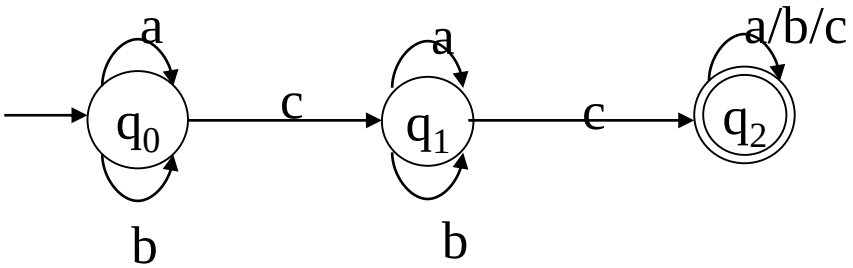
q_0 q_0 q_1 q_2 q_2 q_2

a a c

<u>rejected</u>

q_0 q_0 q_0 q_1

Accepts those strings that contain <u>at least</u> two *c*'s



Inductive Proof (sketch): that the machine correctly accepts strings with at least two *c*'s
Proof goes over the length of the string.

Base: x a string with |x|=0. state will be q0 => rejected.
Inductive hypothesis: |x|= integer *k*, & string *x* is *rejected* - in state q0 (x must have *zero* c),
OR, rejected – in state q1 (x must have *one* c),
OR, accepted – in state q2 (x has already with *two* c's)

Inductive steps: Each case for symbol *p*, for string *xp* (|*xp*| = *k*+1), the last symbol *p* = *a*, *b* or *c*

Formal Definition of a DFA

A DFA is a five-tuple:

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

Q	A finite set of states
Σ	A finite input alphabet
q_0	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, which is a total function from Q x Σ to Q

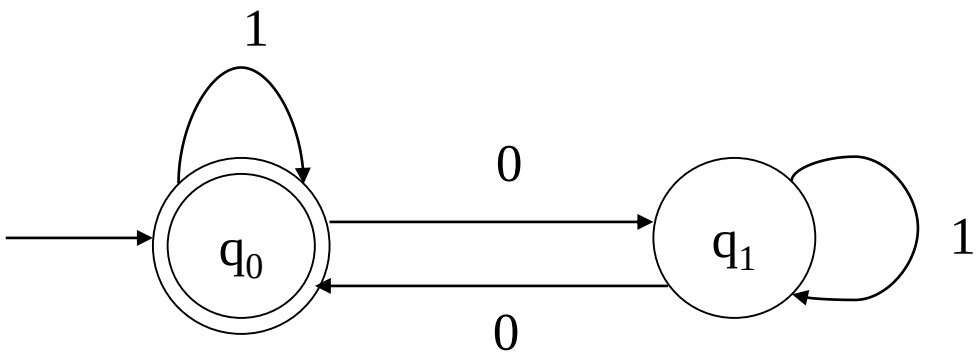
$\delta: (Q \times \Sigma) \rightarrow Q$ δ is defined for any q in Q and s in Σ, and

$\delta(q,s) = q'$ is equal to some state q' in Q, could be q'=q

Intuitively, δ(q,s) is the state entered by M after reading symbol s while in state q.

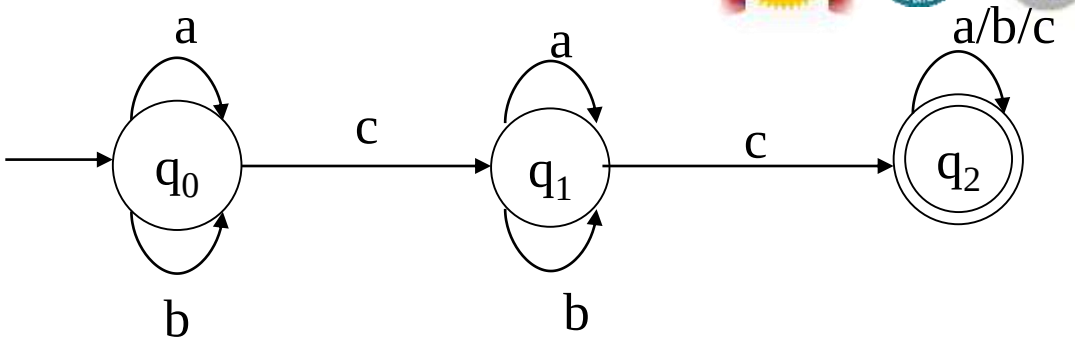
Revisit example #1:

$Q = \{q_0, q_1\}$
$\Sigma = \{0, 1\}$
Start state is q_0
$F = \{q_0\}$



δ:

	0	1
q_0	q_1	q_0
q_1	q_0	q_1



	a	b	c
q_0	q_0	q_0	q_1
q_1	q_1	q_1	q_2
q_2	q_2	q_2	q_2

Since δ is a function, at each step M has exactly one option.

It follows that for a given string, there is exactly one computation.

Extension of δ to Strings

$\delta^{\wedge} : (Q \times \Sigma^{*}) \rightarrow Q$

$\delta^{\wedge}(q,w)$ – The state entered after reading string *w* having started in state *q*.

Formally:

1) $\delta^{\wedge}(q, \varepsilon) = q$, and
2) For all w in Σ^{*} and a in Σ

$$\delta^{\wedge}(q,wa) = \delta(\delta^{\wedge}(q,w), a)$$

Note that:

$\delta^{\wedge}(q,a) = \delta(\delta^{\wedge}(q, \varepsilon), a)$ by definition of $\delta^{\wedge}$, rule #2

$= \delta(q, a)$ by definition of $\delta^{\wedge}$, rule #1

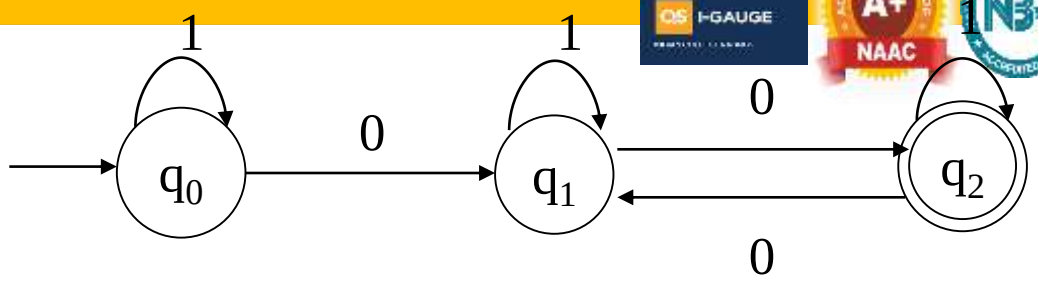
Therefore:

$$\delta^{\wedge}(q, a_1a_2\ldots a_n) = \delta(\delta(\ldots\delta(\delta(q, a1), a2)\ldots), a_n)$$

However, we will abuse notations, and use δ in place of $\delta^{\wedge}$:

$$\delta^{\wedge}(q, a_1a_2\ldots a_n) = \delta(q, a_1a_2\ldots a_n)$$

Example #3:



What is $\delta(q_0, 011)$? Informally, it is the state entered by M after processing 011 having started in state q_0.

Formally:

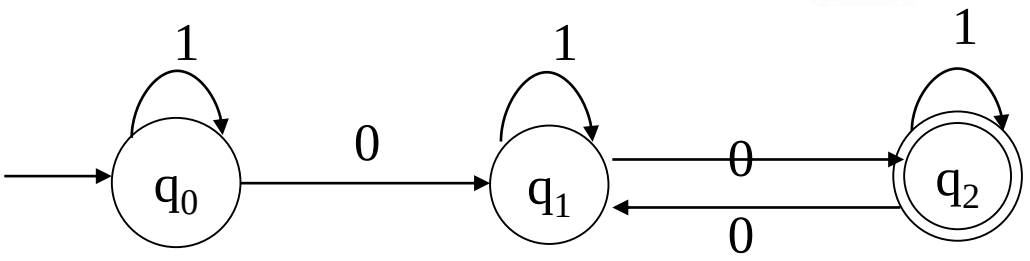
$\delta(q_0, 011)$	$= \delta(\delta(q_0,01), 1)$	by rule #2
	$= \delta(\delta(\delta(q_0,0), 1), 1)$	by rule #2
	$= \delta(\delta(q_1, 1), 1)$	by definition of δ
	$= \delta(q_1, 1)$	by definition of δ
	$= q_1$	by definition of δ

Is 011 accepted? No, since $\delta(q_0, 011) = q_1$ is not a final state.

Language?

L ={ all strings over {0,1} that has 2 or more *0* symbols}

Recall Example #3:



What is $\delta(q_1, 10)$?

$\delta(q_1, 10) = \delta(\delta(q_1,1), 0)$ by rule #2

$= \delta(q_1, 0)$ by definition of δ

$= q_2$ by definition of δ

Is 10 accepted? No, since $\delta(q_0, 10) = q_1$ is not a final state. The fact that $\delta(q_1, 10) = q_2$ is irrelevant, q1 is not the start state!

Definitions related to DFAs

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let w be in Σ^*. Then w is *accepted* by M iff $\delta(q_0, w) = p$ for some state p in F.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Then the *language accepted* by M is the set:

$$L(M) = \{w \mid w \text{ is in } \Sigma^* \text{ and } \delta(q_0, w) \text{ is in } F\}$$

Another equivalent definition:

$$L(M) = \{w \mid w \text{ is in } \Sigma^* \text{ and w is accepted by M}\}$$

Let L be a language. Then L is a ***regular language*** iff there exists a DFA M such that $L = L(M)$.

Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_0, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, p_0, F_2)$ be DFAs. Then M_1 and M_2 are *equivalent* iff $L(M_1) = L(M_2)$.

Notes:

A DFA $M = (Q, \Sigma, \delta, q_0, F)$ partitions the set Σ^* into two sets: L(M) and Σ^* - L(M).

If L = L(M) then L is a subset of L(M) and L(M) is a subset of L (def. of set equality).

Similarly, if $L(M_1) = L(M_2)$ then $L(M_1)$ is a subset of $L(M_2)$ and $L(M_2)$ is a subset of $L(M_1)$.

Some languages are regular, others are not. For example, if

Regular: $L_1 = \{x \mid x$ is a string of 0's and 1's containing an even number of 1's$\}$ and

Not-regular: $L_2 = \{x \mid x = 0^n1^n$ for some $n >= 0\}$

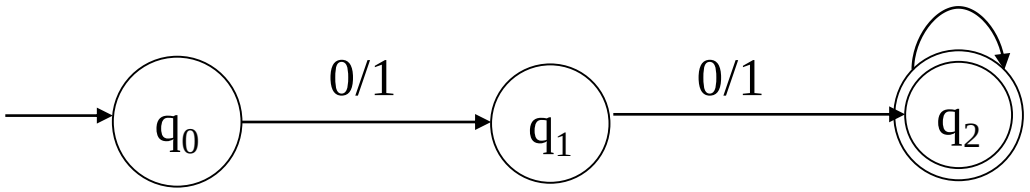
Can you write a program to "simulate" a given DFA, or any arbitrary input DFA?

Question we will address later:

How do we determine whether or not a given language is regular?

Give a DFA M such that:

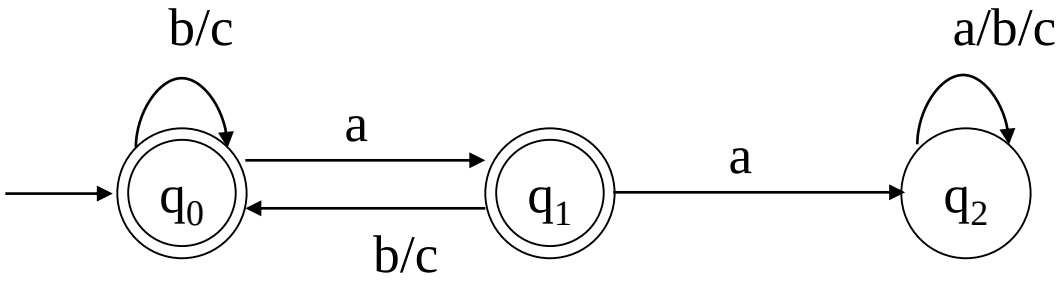
L(M) = {x | x is a string of 0's and 1's and |x| >= 2}



Prove this by induction

Give a DFA M such that:

L(M) = {x | x is a string of (zero or more) a's, b's and c's such that x does *not* contain the substring *aa*}



Logic:

In Start state (q0): b's and c's: ignore – stay in same state

q0 is also "accept" state

First 'a' appears: get ready (q1) to reject

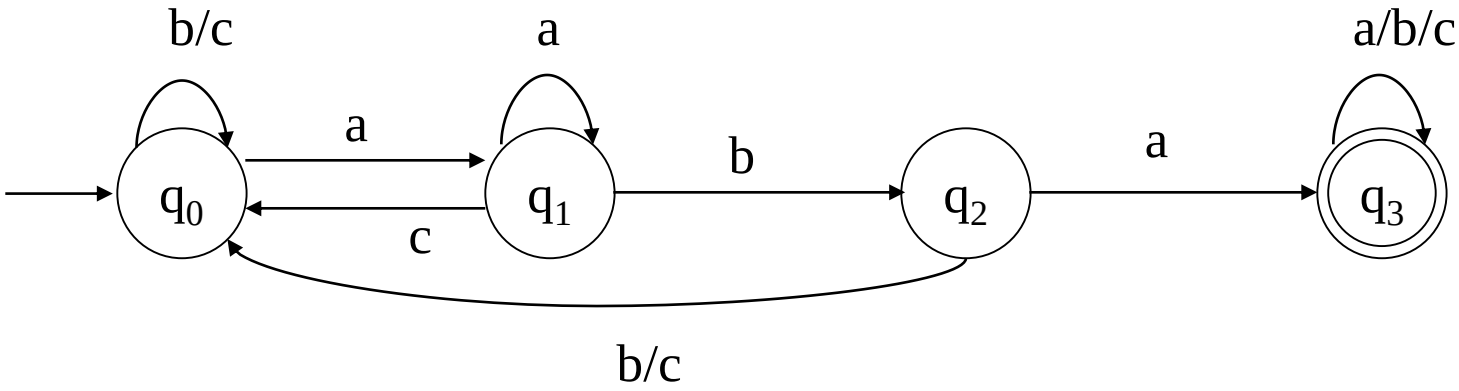
But followed by a 'b' or 'c': go back to start state q0

When second 'a' appears after the "ready" state: go to reject state q2

Ignore everything after getting to the "reject" state q2

Give a DFA M such that:

L(M) = {x | x is a string of a's, b's and c's such that x contains the substring *aba*}

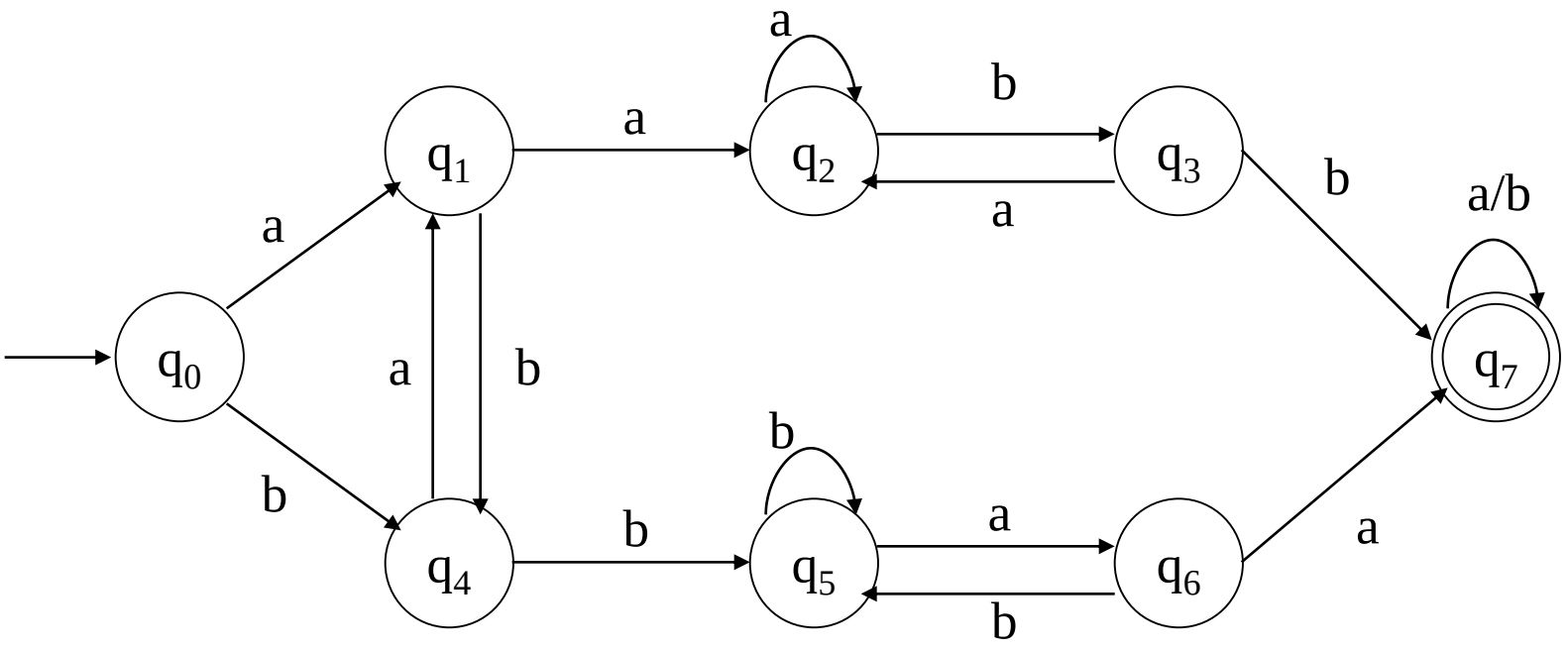


Logic: acceptance is straight forward, progressing on each expected symbol However, rejection needs special care, in each state (for DFA, we will see this becomes easier in NFA, non-deterministic machine)

Give a DFA M such that:

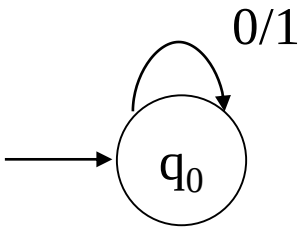
L(M) = {x | x is a string of a's and b's such that x contains both *aa* and *bb*}

First do, for a language where 'aa' comes before 'bb'
Then do its reverse; and then parallelize them.

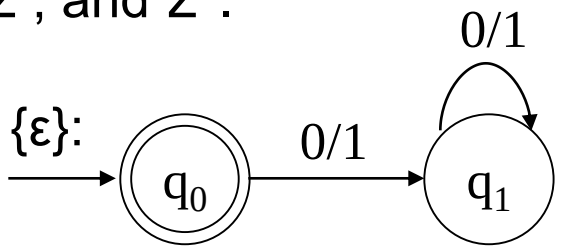


Let Σ = {0, 1}. Give DFAs for {}, {ε}, Σ^*, and Σ^+.

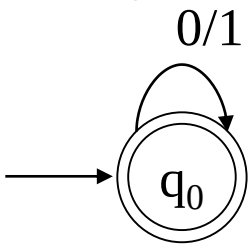
For {}:



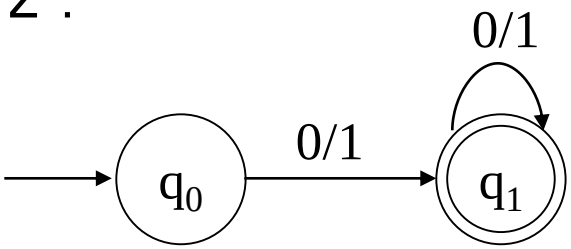
For {ε}:



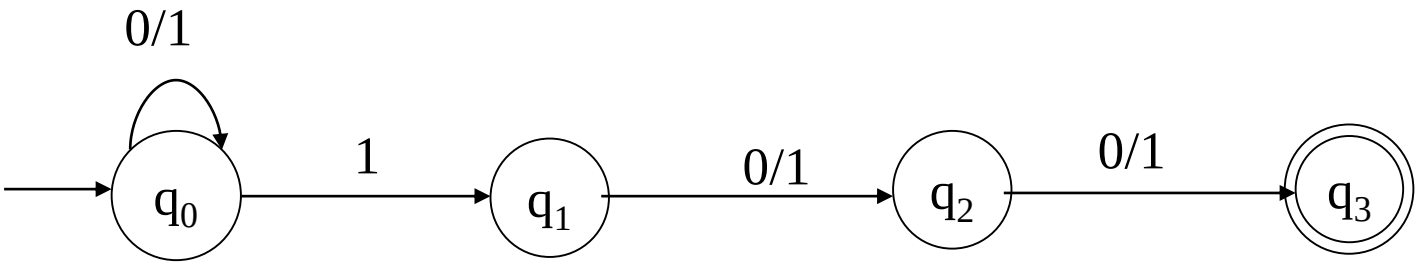
For Σ^*:



For Σ^+:



Problem: Third symbol from last is *1*



Is this a DFA?

No, but it is a *Non-deterministic Finite Automaton*

Nondeterministic Finite State Automata (NFA)

An NFA is a five-tuple:

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

Q	A finite set of states
Σ	A finite input alphabet
q_0	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, which is a total function from Q x Σ to 2^Q

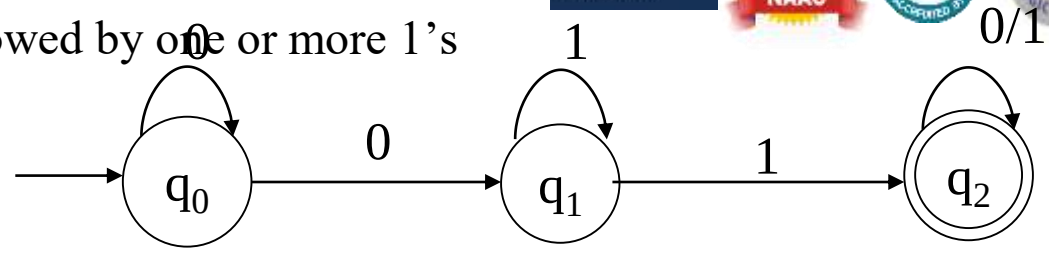
$\delta: (Q \times \Sigma) \rightarrow \mathbf{2^Q}$: 2^Q is the power set of Q, the set of *all subsets* of Q

$\delta(q,s)$:The set of all states p such that there is a transition labeled s from q to p

$\delta(q,s)$ is a function from Q x S to 2^Q (but not only to Q)

Example #1: one or more 0's followed by one or more 1's

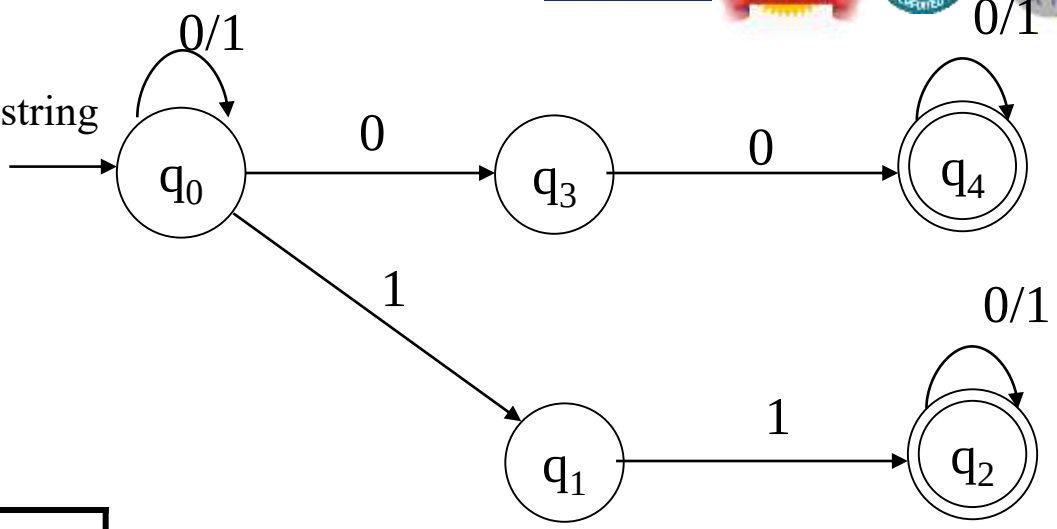
$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{0, 1\}$
Start state is q_0
$F = \{q_2\}$



δ:

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

Example #2: pair of 0's *or* pair of 1's as substring

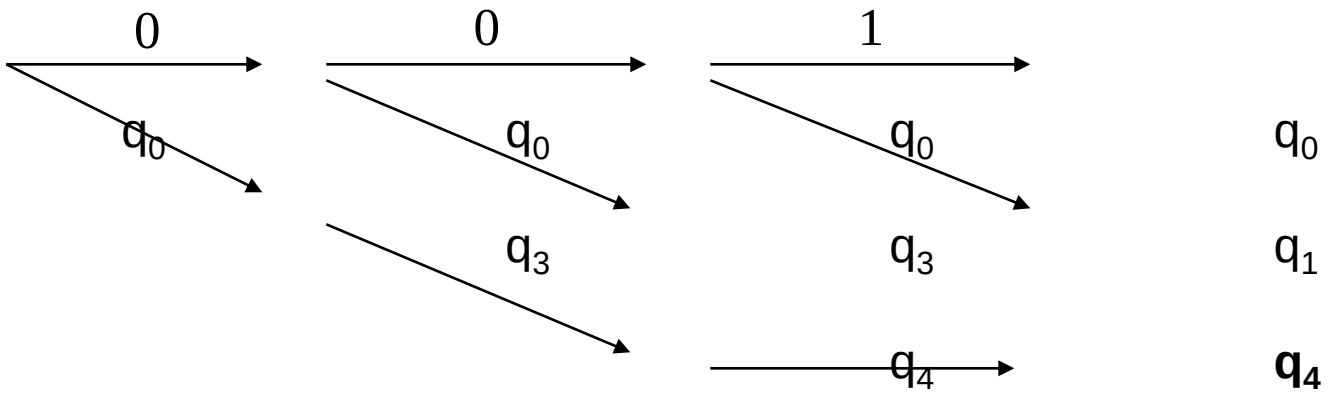


$Q = \{q_0, q_1, q_2, q_3, q_4\}$
$\Sigma = \{0, 1\}$
Start state is q_0
$F = \{q_2, q_4\}$

δ:

	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\{\}$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\{\}$
q_4	$\{q_4\}$	$\{q_4\}$

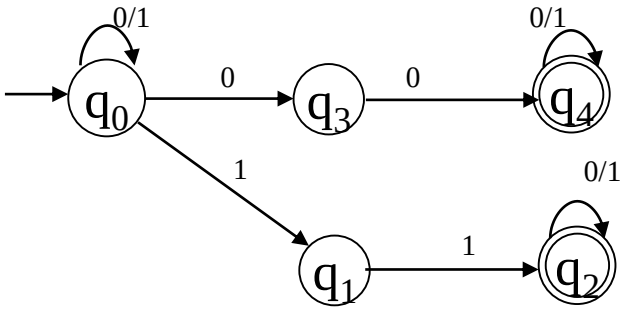
Determining if a given NFA (example #2) accepts a given string (001) can be done algorithmically:



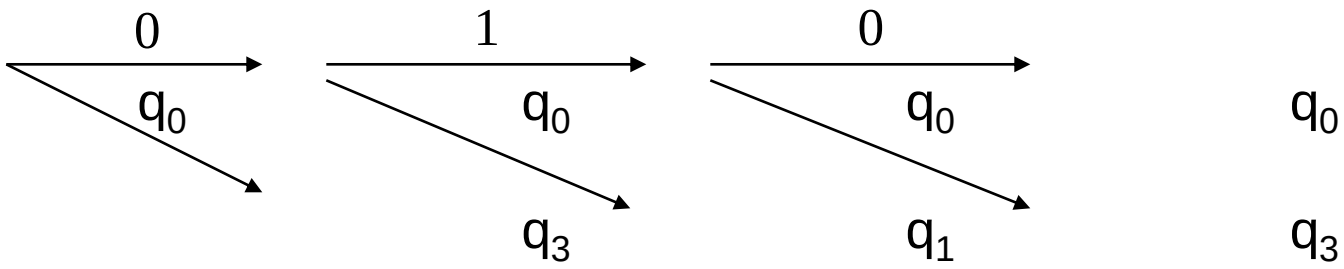
accepted

Each level will have at most *n* states:

Complexity: O(|x|*n), for running over a string *x*

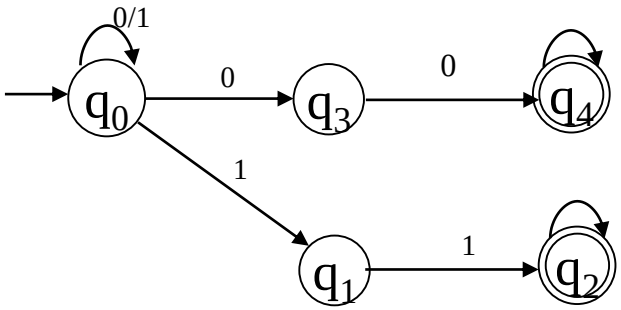


Another example (010):



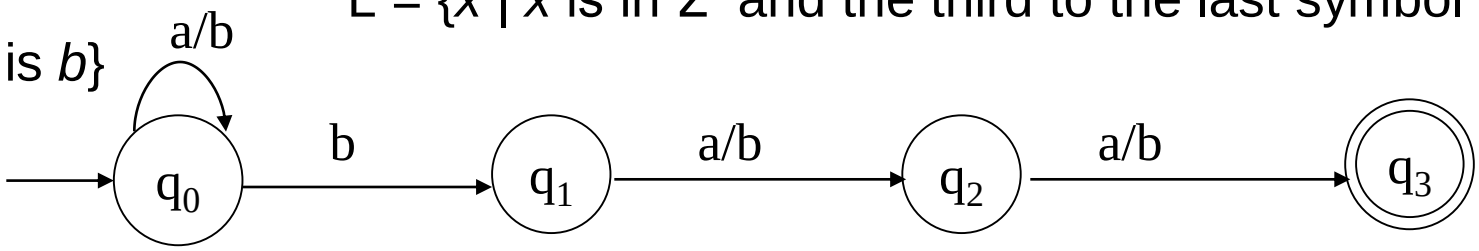
not accepted

All paths have been explored, and none lead to an accepting state.



Let Σ = {a, b}. Give an NFA M that accepts:

L = {*x* | *x* is in Σ* and the third to the last symbol in *x* is *b*}



Is L a subset of L(M)?
Is L(M) a subset of L?

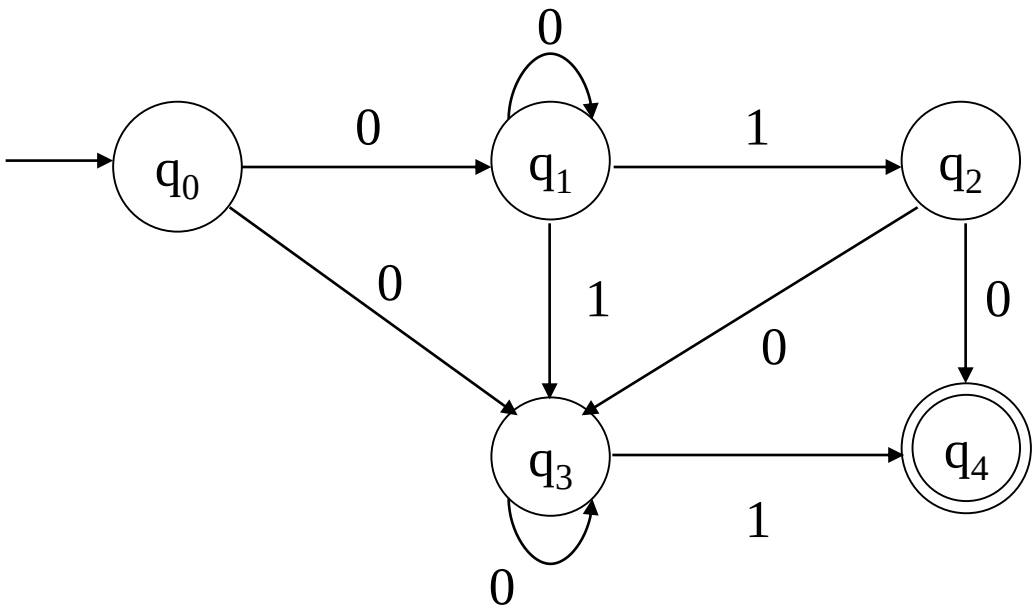
Give an equivalent DFA as an exercise.

Extension of δ to Strings and Sets of States

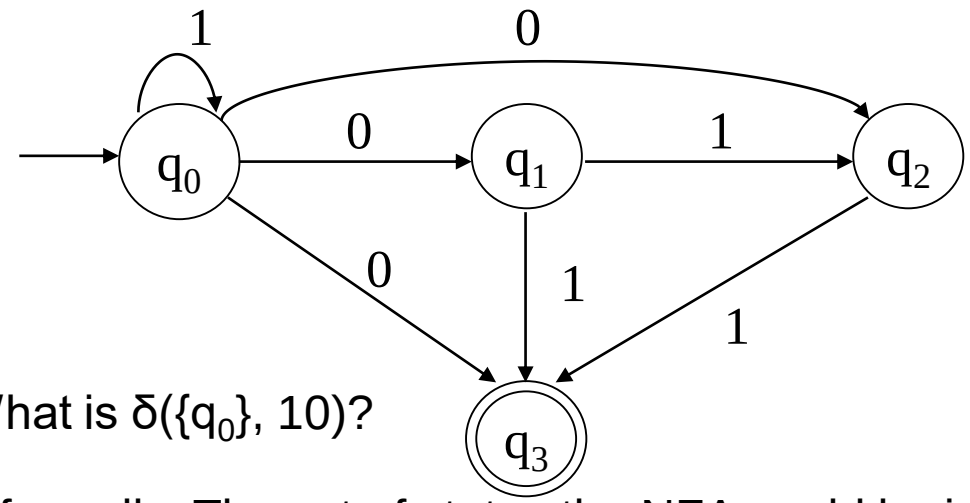
What we currently have: $\delta : (Q \times \Sigma) \rightarrow 2^Q$

What we want (why?): $\delta : (2^Q \times \Sigma^*) \rightarrow 2^Q$

We will do this in two steps, which will be slightly different from the book, and we will make use of the following NFA.



Example:



What is $\delta(\{q_0\}, 10)$?

Informally: The set of states the NFA could be in after processing 10,

having started in state q_0, i.e., $\{q_1, q_2, q_3\}$.

Formally: $\delta(\{q_0\}, 10) = \delta(\delta(\{q_0\}, 1), 0)$

$= \delta(\{q_0\}, 0)$

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

Is 10 accepted? Yes!

Example:

What is $\delta(\{q_0, q_1\}, 1)$?

$\delta(\{q_0 , q_1\}, 1) = \delta(\{q_0\}, 1) \cup \delta(\{q_1\}, 1)$
$= \{q_0\} \cup \{q_2, q_3\}$
$= \{q_0, q_2, q_3\}$

What is $\delta(\{q_0, q_2\}, 10)$?

$\delta(\{q_0 , q_2\}, 10) = \delta(\delta(\{q_0 , q_2\}, 1), 0)$
$= \delta(\delta(\{q_0\}, 1) \cup \delta(\{q_2\}, 1), 0)$
$= \delta(\{q_0\} \cup \{q_3\}, 0)$
$= \delta(\{q_0,q_3\}, 0)$
$= \delta(\{q_0\}, 0) \cup \delta(\{q_3\}, 0)$
$= \{q_1, q_2, q_3\} \cup \{\}$
$= \{q_1, q_2, q_3\}$

Example:

$\delta(\{q_0\}, 101) = \delta(\delta(\{q_0\}, 10), 1)$
$= \delta(\delta(\delta(\{q_0\}, 1), 0), 1)$
$= \delta(\delta(\{q_0\}, 0), 1)$
$= \delta(\{q_1, q_2, q_3\}, 1)$
$= \delta(\{q_1\}, 1) \cup \delta(\{q_2\}, 1) \cup \delta(\{q_3\}, 1)$
$= \{q_2, q_3\} \cup \{q_3\} \cup \{\}$
$= \{q_2, q_3\}$

Is 101 accepted? Yes! q_3 is a final state.

Equivalence of DFAs and NFAs

Do DFAs and NFAs accept the same *class* of languages?

Is there a language L that is accepted by a DFA, but not by any NFA?

Is there a language L that is accepted by an NFA, but not by any DFA?

Observation: Every DFA is an NFA, DFA is only restricted NFA.

Therefore, if L is a regular language then there exists an NFA M such that L = L(M).

It follows that NFAs accept all regular languages.

But do NFAs accept more?

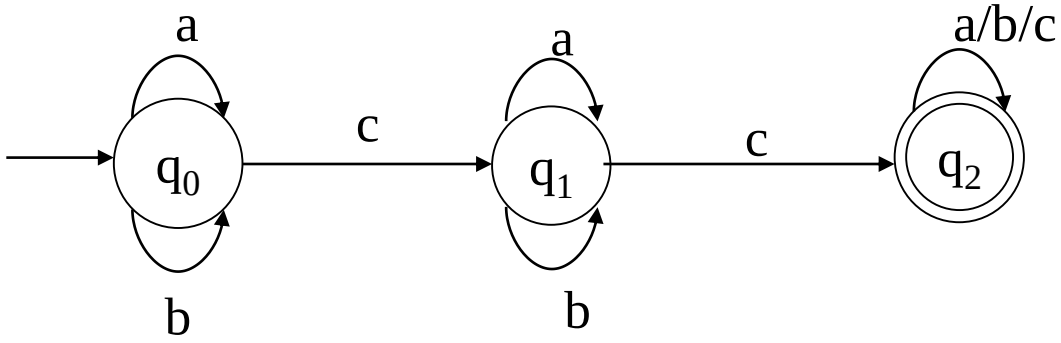
Consider the following DFA: 2 or more c's

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

$\Sigma = \{a, b, c\}$

Start state is q_0

$F = \{q_2\}$



δ:

	a	b	c
q_0	q_0	q_0	q_1
q_1	q_1	q_1	q_2
q_2	q_2	q_2	q_2

Lemma 1: Let M be an DFA. Then there exists a NFA M' such that L(M) = L(M').

Proof: Every DFA is an NFA. Hence, if we let M' = M, then it follows that L(M') = L(M).

The above is just a formal statement of the observation from the previous slide.

Lemma 2: Let M be an NFA. Then there exists a DFA M' such that L(M) = L(M').

Proof: (sketch)

Let $M = (Q, \Sigma, \delta, q_0, F)$.

Define a DFA $M' = (Q', \Sigma, \delta', q'_0, F')$ as:

$Q' = 2^Q$ Each state in M' corresponds to a

$= \{Q_0, Q_1, \ldots,\}$ subset of states from M

where $Q_u = [q_{i0}, q_{i1}, \ldots q_{ij}]$

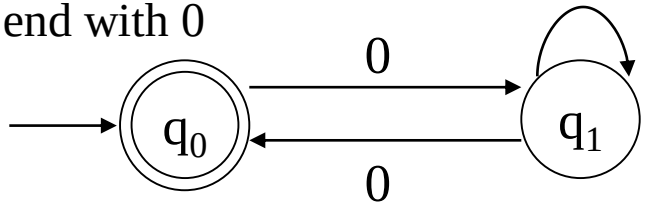
$F' = \{Q_u \mid Q_u$ contains at least one state in $F\}$

$q'_0 = [q_0]$

$\delta'(Q_u, a) = Q_v$ iff $\delta(Q_u, a) = Q_v$

0/1

Example: empty string or start and end with 0



$Q = \{q_0, q_1\}$
$\Sigma = \{0, 1\}$
Start state is q_0
$F = \{q_0\}$

δ:

	0	1
q_0	$\{q_1\}$	$\{\}$
q_1	$\{q_0, q_1\}$	$\{q_1\}$

Theorem: Let L be a language. Then there exists an DFA M such that L = L(M) iff there exists an NFA M' such that L = L(M').

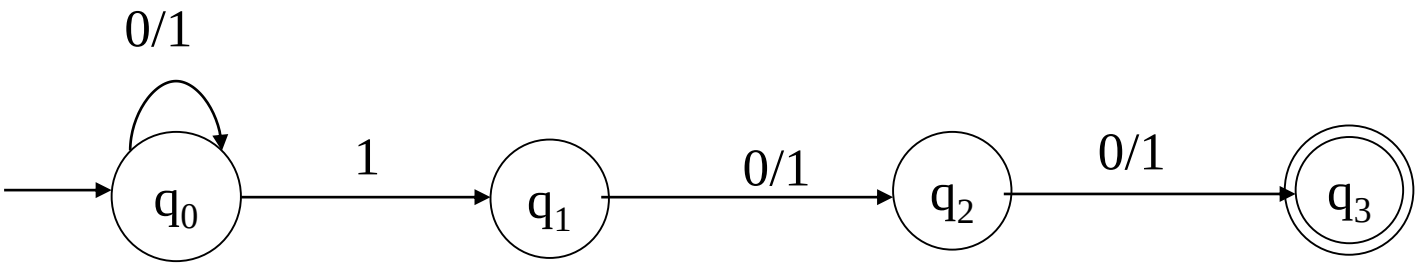
Proof:

(if) Suppose there exists an NFA M' such that L = L(M'). Then by Lemma 2 there exists an DFA M such that L = L(M).

(only if) Suppose there exists an DFA M such that L = L(M). Then by Lemma 1 there exists an NFA M' such that L = L(M').

Corollary: The NFAs define the regular languages.

Problem: Third symbol from last is 1



Now, can you convert this NFA to a DFA?

NFAs with ε Moves

An NFA-ε is a five-tuple:

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

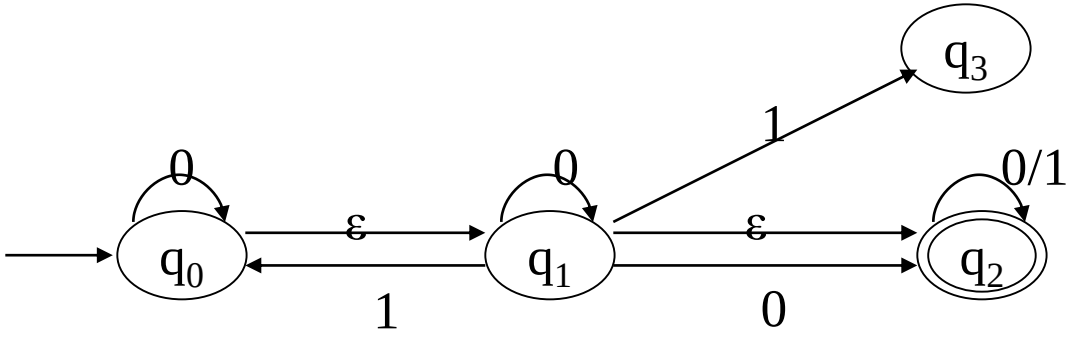
Q	A finite set of states
Σ	A finite input alphabet
q_0	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, which is a total function from Q x Σ U {ε} to 2^Q

$\delta: (Q \times (\mathbf{\Sigma \cup \{\varepsilon\}})) \rightarrow 2^Q$

$\delta(q,s)$ -The set of all states p such that there is a transition labeled a from q to p, where a is in Σ U {ε}

Sometimes referred to as an NFA-ε other times, simply as an NFA.

Example



δ:	0	1	ε
q_0	$\{q_0\}$	$\{\ \}$	$\{q_1\}$
q_1	$\{q_1, q_2\}$	$\{q_0, q_3\}$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$	$\{\ \}$
q_3	$\{\ \}$	$\{\ \}$	$\{\ \}$

- A string $w = w_1w_2\ldots w_n$ is processed as $\varepsilon^*w_1\varepsilon^*w_2\varepsilon^* \ldots \varepsilon^*w_n\varepsilon^*$

- Example: all computations on 00:

0	ε	0
q_0	q_0	q_1 q_2
	:	

Informal Definitions

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA-ε.

A String w in Σ^* is *accepted* by M iff there exists a path in M from q_0 to a state in F labeled by w and zero or more ε transitions.

The language accepted by M is the set of all strings from Σ^* that are accepted by M.

ε-closure

Define ε-closure(q) to denote the set of all states reachable from q by zero or more ε transitions.

Examples: (for the previous NFA)

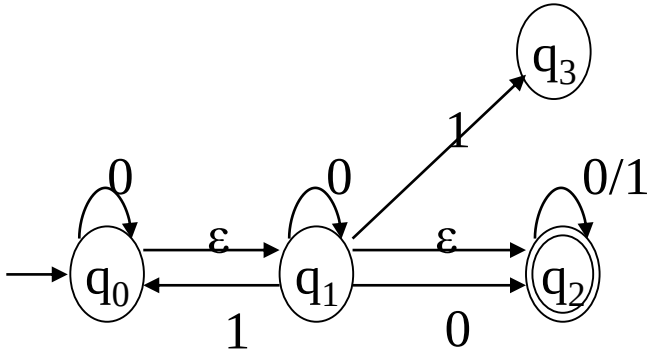
ε-closure(q_0) = {q_0, q_1, q_2} ε-closure(q_2) = {q_2}
ε-closure(q_1) = {q_1, q_2} ε-closure(q_3) = {q_3}

ε-closure(q) can be extended to sets of states by defining:

$$\text{ε-closure}(P) = \bigcup_{q \in P} \text{ε-closure}(q)$$

Examples:

ε-closure({q_1, q_2}) = {q_1, q_2}
ε-closure({q_0, q_3}) = {q_0, q_1, q_2, q_3}



Equivalence of NFAs and NFA-εs

Do NFAs and NFA-ε machines accept the same *class* of languages?

Is there a language L that is accepted by a NFA, but not by any NFA-ε?

Is there a language L that is accepted by an NFA-ε, but not by any DFA?

Observation: Every NFA is an NFA-ε.

Therefore, if L is a regular language then there exists an NFA-ε M such that L = L(M).

It follows that NFA-ε machines accept all regular languages.

But do NFA-ε machines accept more?

Lemma 1: Let M be an NFA. Then there exists a NFA-ε M' such that L(M) = L(M').

Proof: Every NFA is an NFA-ε. Hence, if we let M' = M, then it follows that L(M') = L(M).

The above is just a formal statement of the observation from the previous slide.

Lemma 2: Let M be an NFA-ε. Then there exists a NFA M' such that L(M) = L(M').

Proof: (sketch)

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA-ε.

Define an NFA $M' = (Q, \Sigma, \delta', q_0, F')$ as:

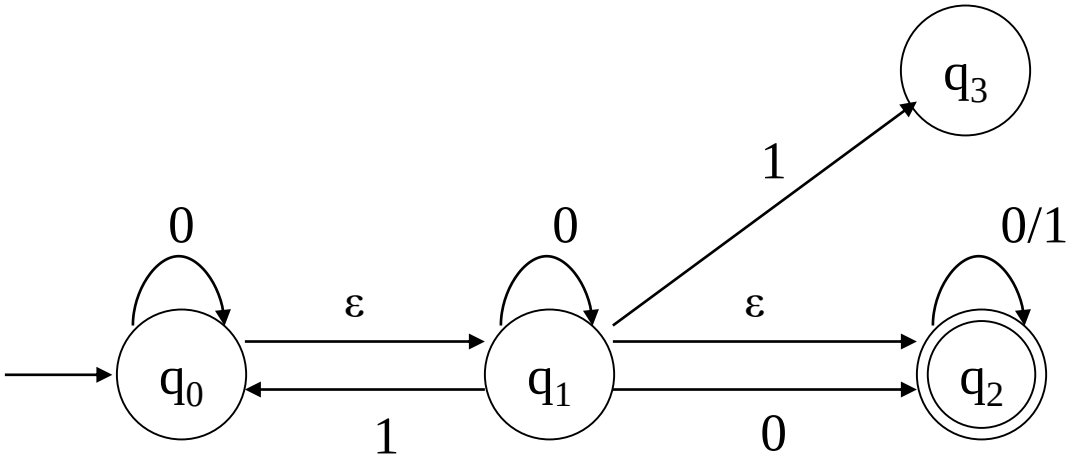
F' = F U {q} if ε-closure(q) contains at least one state from F
F' = F otherwise

$\delta'(q, a) = \delta^{\wedge}(q, a)$ - for all q in Q and a in Σ

Notes:

δ': $(Q \times \Sigma) \rightarrow 2^Q$ is a function
M' has the same state set, the same alphabet, and the same start state as M
M' has no ε transitions

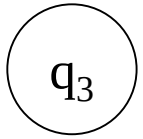
Example

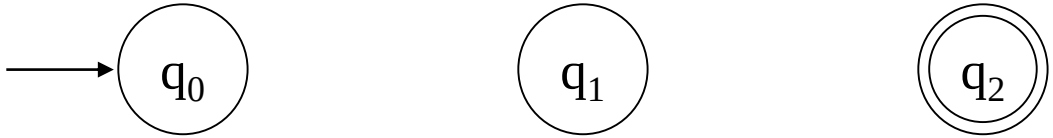


Step #1:

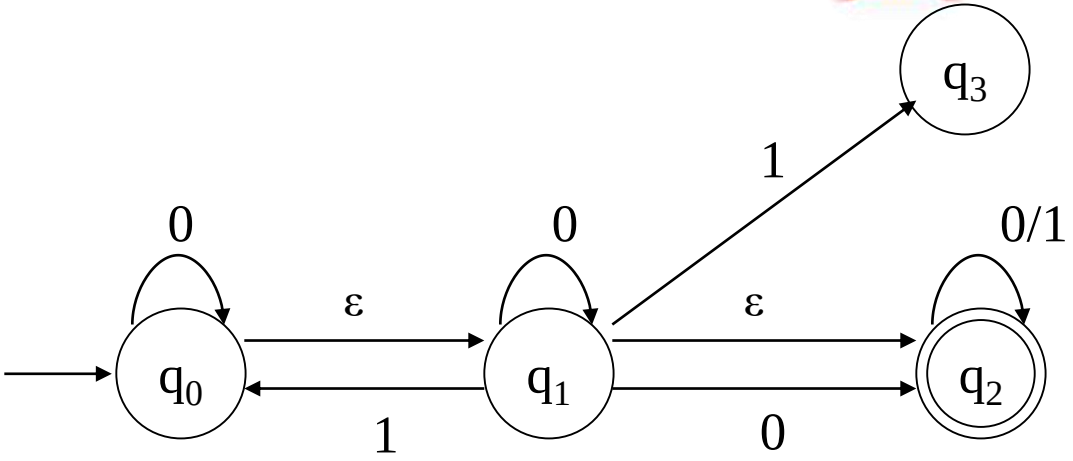
Same state set as M

q_0 is the starting state



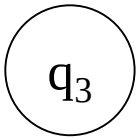


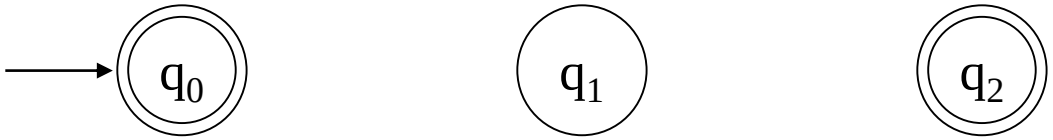
Example



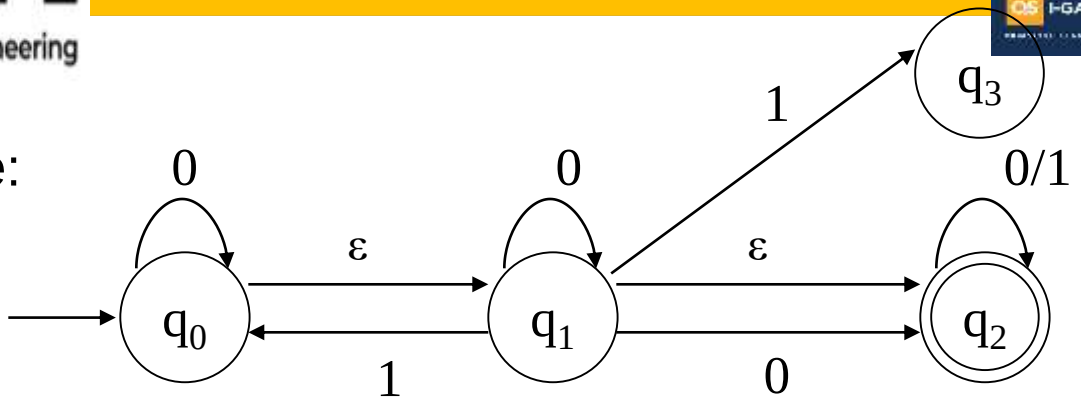
Step #2:

q_0 becomes a final state

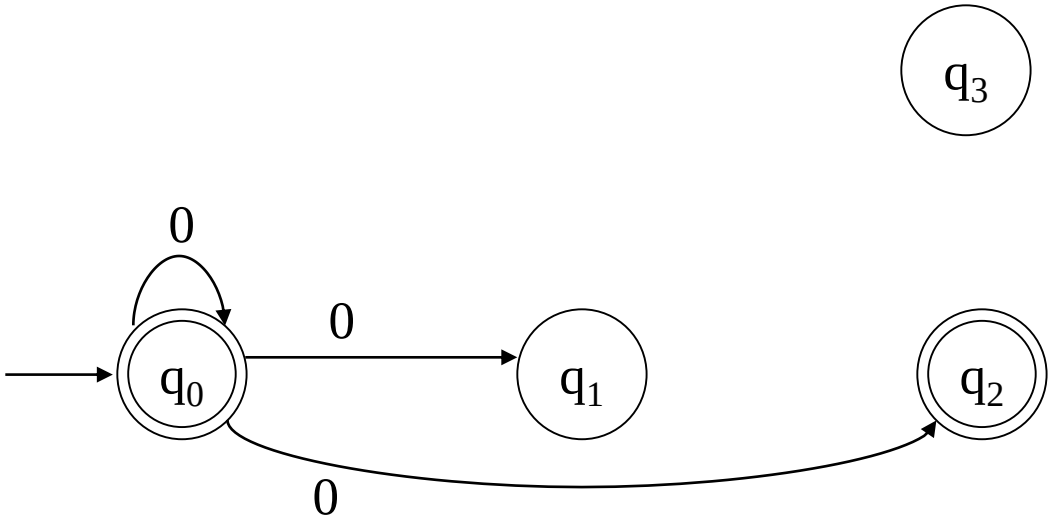




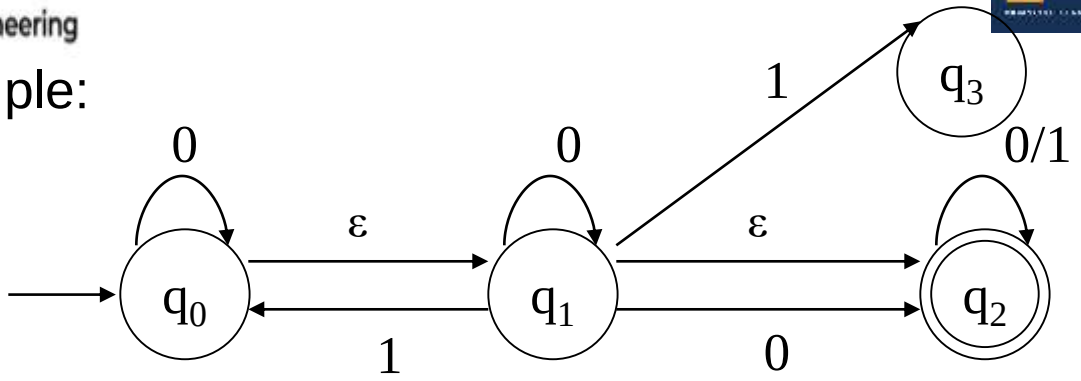
Example:



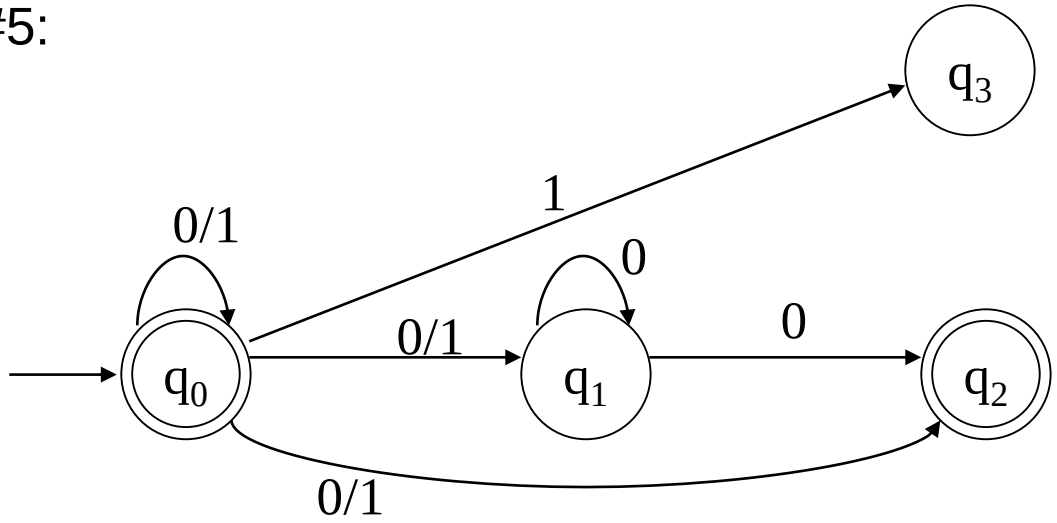
Step #3:



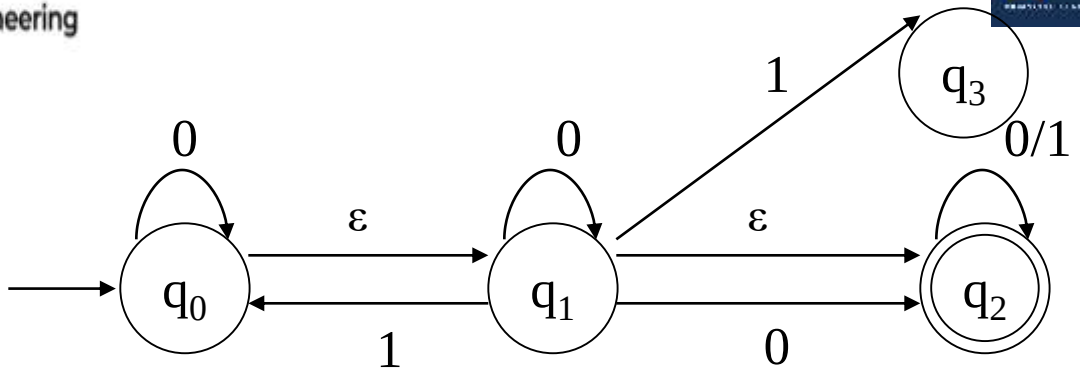
Example:



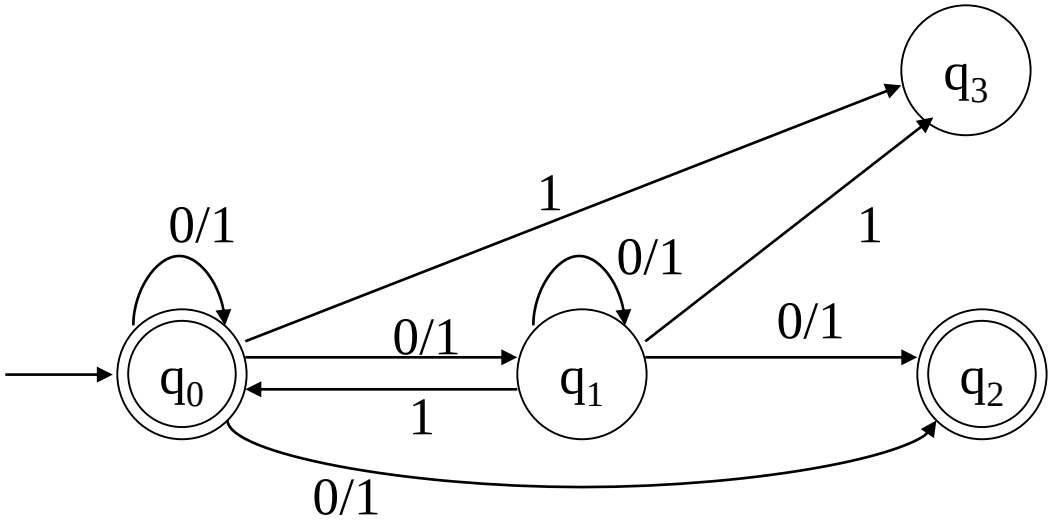
Step #5:



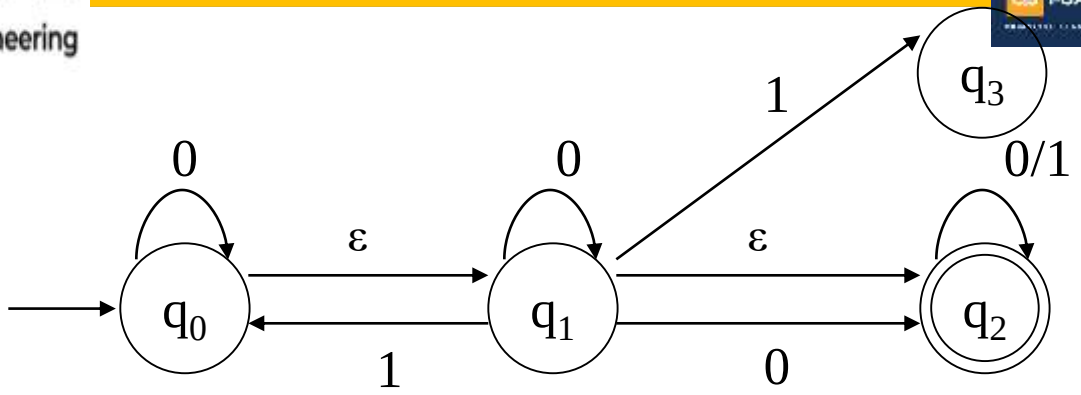
Example



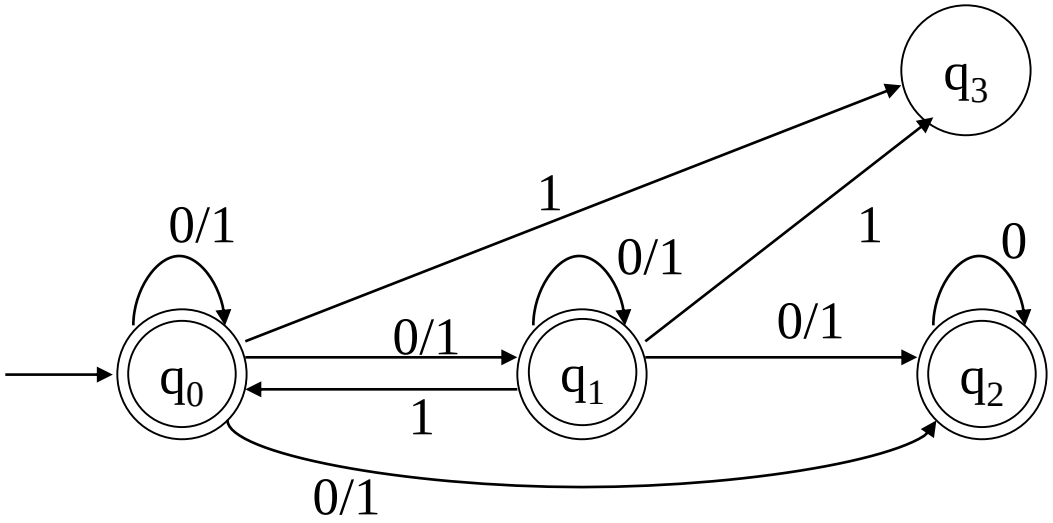
Step #6:

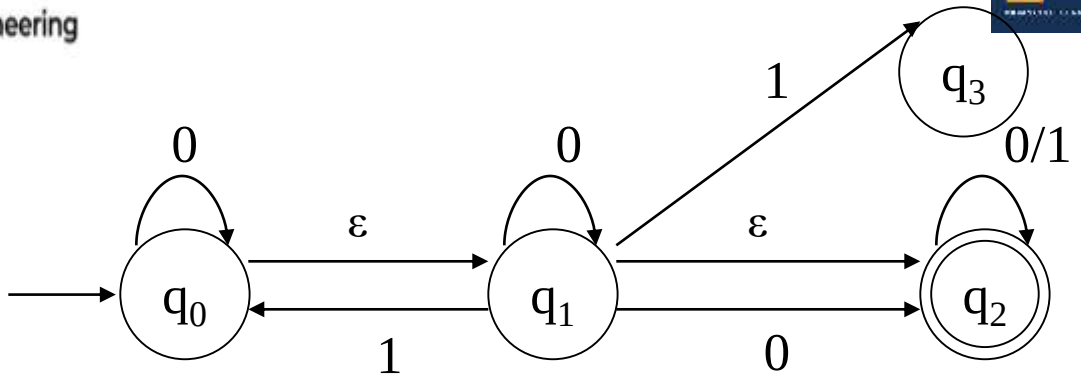


Example



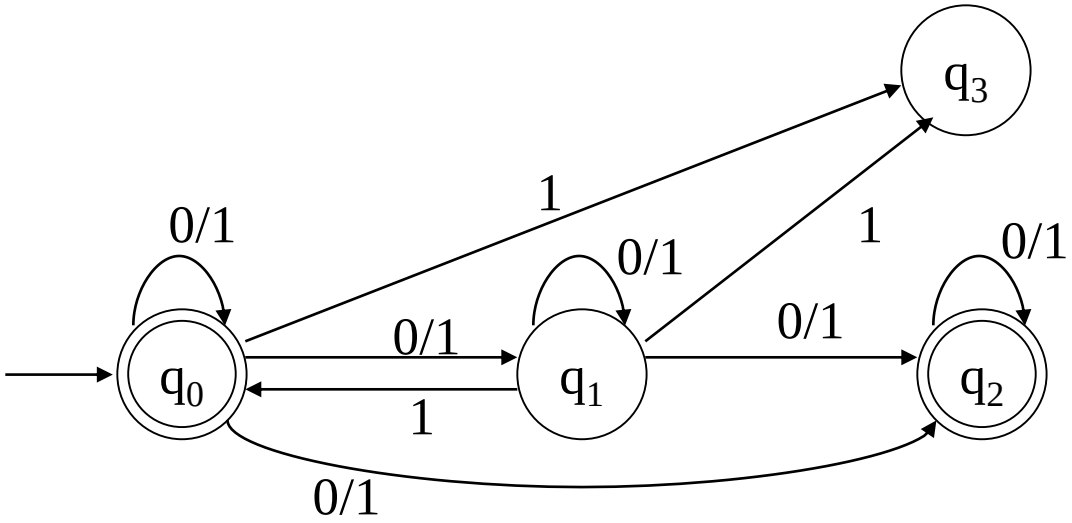
Step #7:





Step #8: [use table of e-closure]

Done!



Theorem: Let L be a language. Then there exists an NFA M such that L= L(M) iff there exists an NFA-ε M' such that L = L(M').

Proof:

(if) Suppose there exists an NFA-ε M' such that L = L(M'). Then by Lemma 2 there exists an NFA M such that L = L(M).

(only if) Suppose there exists an NFA M such that L = L(M). Then by Lemma 1 there exists an NFA-ε M' such that L = L(M').

Corollary: The NFA-ε machines define the regular languages.

THEORY OF COMPUTATION		Semester	V
Course Code	**BCS503**	CIE Marks	50
Teaching Hours/Week (L: T:P: S)	(3:2:0:0)	SEE Marks	50
Total Hours of Pedagogy	50	Total Marks	100
Credits	04	Exam Hours	3
Examination type (SEE)	Theory		

Course outcome (Course Skill Set)

At the end of the course, the student will be able to:

1. Apply the fundamentals of automata theory to write DFA, NFA, Epsilon-NFA and conversion between them.
2. Prove the properties of regular languages using regular expressions.
3. Design context-free grammars (CFGs) and pushdown automata (PDAs) for formal languages.
4. Design Turing machines to solve the computational problems.
5. Explain the concepts of decidability and undecidability.

Module-2	10 Hours
Regular Expressions, Finite Automata and Regular Expressions, Proving Languages not to be Regular. Closure Properties of Regular Languages, Equivalence and Minimization of Automata, Applications of Regular Expressions **TEXT BOOK: Sections 3.1, 3.2 (Except 3.2.1), 3.3, 4.1, 4.2, 4.4**	

RE's: Introduction

•*Regular expressions* are algebraic ways to describe sets of strings that are regular languages (denoted by L(RE)).

•RE's and their languages are defined recursively.

Operations in RE's

3 basic operations between languages (i.e., sets of strings) derived from RE's:

- **Union** denoted by L(RE1)+L(RE2)
- **Concatenation** denoted by L(RE1).L(RE2) or L(RE1)L(RE2)
- **Closure** denoted by L*(RE).

Definition of + . and * operations

L + M the is set of all strings either in L or in M or in both
Example: {001,10,111} + {e,001}={e,001,10,111}

L.M or simply LM is the set of all string that can be formed by concatenating any string in L with any string in M.
Example: {001,10,111}.{e,001}=
{001,10,111,001001,10001,111001}
Note! left-right order is preserved

L* is set of strings obtained by taking any number of strings from L and forming all possible concatenation.

L* in relation to powers of L

$L^* = U_{k \geq 0} L^k$

Union of all powers of L (including zero)

$L^0 = \{e\}$; hence, L* contains {e} for any L

$L^1 = L$

L^k (k>1) concatenation of k copies of L

If $L=\{0,11\}$, $L^2 = \{0,11\}\{0,11\} = \{00,011,110,1111\}$

L(∅) is the empty language (no strings)

$L(\emptyset)^* = \{e\}$ rare example of finite closure

L^+ is the same as L* except no empty string

Elementary components of RE's

- Basis 1: any symbol, *a*, is a RE.
  - L(RE)={*a*} is language containing one string of length 1.
- Basis 2: e is a RE.
  - L(RE)={e} consists of empty string only
- Basis 3: ∅ is a RE.
  - L(RE) = ∅ has no strings

Recursive Definitions of RE's

Induction 1: If E_1 and E_2 are RE's, then E_1+E_2 is a RE, and $L(E_1+E_2) = L(E_1)+L(E_2)$

Induction 2: If E_1 and E_2 are RE's then E_1E_2 is a RE, and $L(E_1E_2) = L(E_1)L(E_2)$

Induction 3: If E is a RE, then E* is a RE, and L(E*) = (L(E))* or simply L(E)*

Building regular expressions

Precedence of operations

- * highest
- . (or juxtaposition) next
- + lowest

Parentheses are used as needed to influence the precedence of operators.

* operates on smallest sequence of symbols to its left that is a legal RE

Example: **01*** closure on **1** only

After grouping all *'s to their operands, group all concatenations to their operands

Example: **0** to **1*** in RE=**01***

Finally, group unions (+) with operands;

01*+**1**=0{e,1,11,…}+1={0,01,011,…}+1

={0,1,01,011,…}

Precedence matters:

L(**01***+**1**)=0{e,1,11,…}+1={0,01,011…}+1={0,1,01,011..}

When we override precedence by ()

L(**0**(**1***+**1**)) = **01***= 0{e,1,11,…}={0,01,011,…}

Note: **1*** and (**1***+**1**) are the same

Enumerate the strings in these L(RE)

L(**01**) = ?
L(**01**+**0**) = ?
L(**0**(**1**+**0**)) = ?
L(**0***) = ?
L(**01***) = ?
L((**01**)*) = ?
L((**01**)$^{+}$) = ?

Enumerate the strings in L(RE)

L(**01**) = {01}.
L(**01**+**0**) = {01, 0}.
L(**0**(**1**+**0**)) = {0}{0,1}={00, 01}.
L(**0***) = {ϵ, 0, 00, 000,… }.
L(**01***) = 0{e,1,11,…}={0,01,011,…}
L((**01**)*) = {e,01,0101,…}
L((**01**)$^{+}$) = {01,0101,…}

Given a description of L, what RE will generate the strings in L?

Example: L = strings of alternating 0's and 1's
Start by enumerating strings in L

L={e,0,1}+strings alternating 0's and 1's length >1

L={e,0,1,01,10} + strings alternating 0's and 1's length >2

L={e,0,1,01,10,010,101} + strings alternating 0's and 1's length > 3

Generalize: {e,0,1} +
Strings with even number of characters that begin 0 and end 1
Strings with even number of characters that begin 1 and end 0
Strings with odd number of characters that begin 0 and end 0
Strings with odd number of characters that begin 1 and end 1

Build each from closure and concatenation

Strings with even number of characters that begin 0 and end 1
(**01**)*={e,01,0101,010101,…}

Strings with even number of characters that begin 1 and end 0
(**10**)*={e,10,1010,101010,…}

Strings with odd number of characters that begin 0 and end 0
0(**10**)*=0{e,10,1010…}={0,010,01010,…}

Strings with odd number of characters that begin 1 and end 1
1(**01**)*=1{e,01,0101…}={1,101 10101,…}

L is the union of 4 cases
RE = (**01**)*+(**10**)*+**0**(**10**)*+**1**(**01**)*

L is the union of 4 cases

RE = (**01**)*+(**10**)*+**0**(**10**)*+**1**(**01**)*

Find a different expression for RE using the distribution of concatenation over union.

L is the union of 4 cases

RE = (**01**)*+(**10**)*+**0**(**10**)*+**1**(**01**)*

Find a different expression for RE using the distribution of concatenation over union.

RE = (e+1)(01)* + (e+0)(10)*

RE = (**01**)*+(**10**)*+**0**(**10**)*+**1**(**01**)*

is not the only RE that defines

L={e,0,1}+strings alternating 0’ and 1’s length >1

Enumerate and describe string defined by

(**01**)***0**

1(**01**)***0**

RE = (**01**)*+(**10**)*+**0**(**10**)*+**1**(**01**)*

is not the only RE that defines

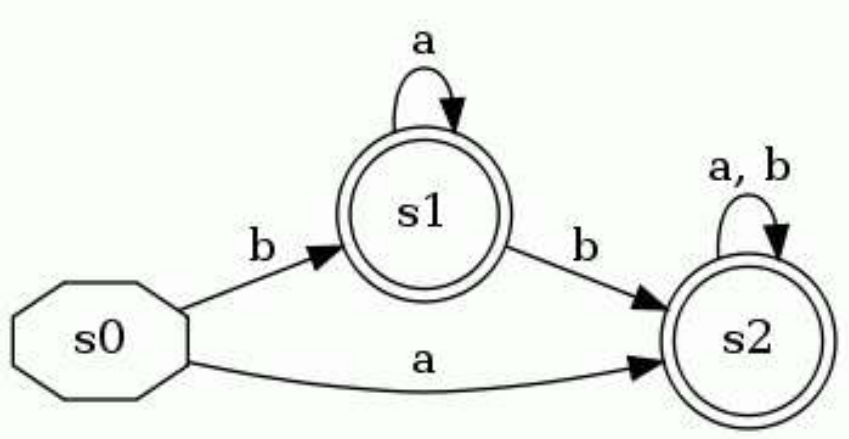
L={e,0,1}+strings alternating 0’ and 1’s length >1

(**01**)***0**={0,010,…} odd, begin=0, end=0

1(**01**)***0** ={10,1010,…} even, begin=1, end=0

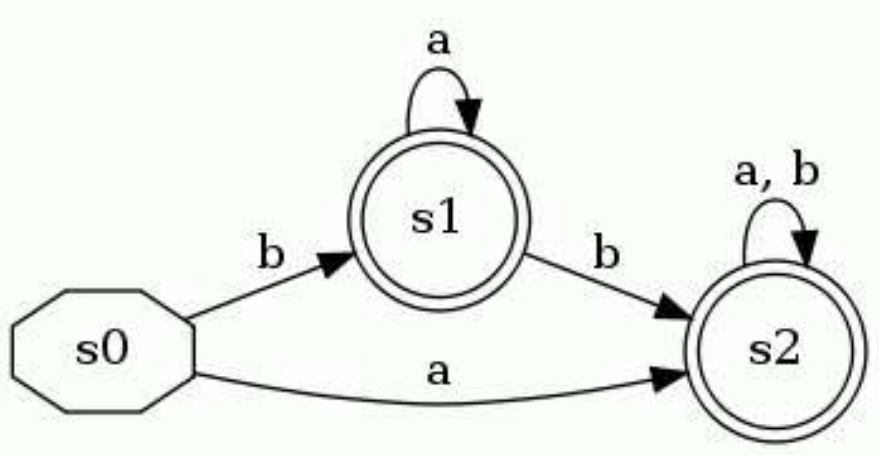
DFA Minimization

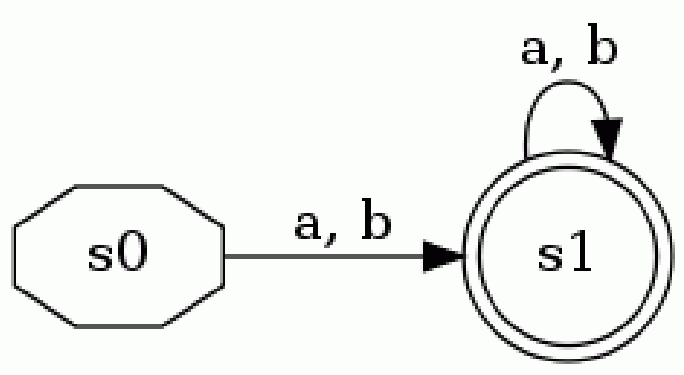
- Some states can be redundant:
  - The following DFA accepts (a|b)+
  - State s1 is not necessary



DFA Minimization

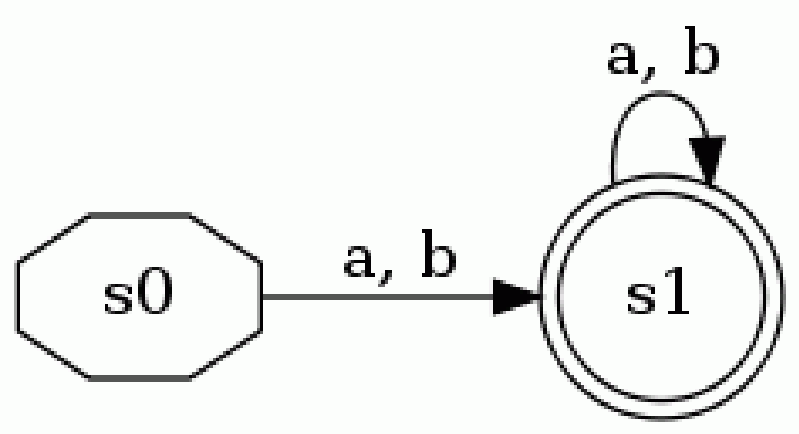
- So these two DFAs are *equivalent*:





DFA Minimization

- This is a *state-minimized* (or just *minimized*) DFA
  - Every remaining state is necessary



DFA Minimization

- The task of *DFA minimization*, then, is to automatically transform a given DFA into a state-minimized DFA
  - Several algorithms and variants are known
  - Note that this also in effect can minimize an NFA (since we know algorithm to convert NFA to DFA)

DFA Minimization Algorithm

- Create lower-triangular table DISTINCT, initially blank
- For every pair of states (p,q):
  - If p is final and q is not, or vice versa
    - DISTINCT(p,q) = ε
- Loop until no change for an iteration:
  - For every pair of states (p,q) and each symbol α
    - If DISTINCT(p,q) is blank and

      DISTINCT(δ(p,α), δ(q,α)) is not blank
      - DISTINCT(p,q) = α
- Combine all states that are not distinct

THEORY OF COMPUTATION		Semester	V
Course Code	**BCS503**	CIE Marks	50
Teaching Hours/Week (L: T:P: S)	(3:2:0:0)	SEE Marks	50
Total Hours of Pedagogy	50	Total Marks	100
Credits	04	Exam Hours	3
Examination type (SEE)	Theory		

Course outcome (Course Skill Set)

At the end of the course, the student will be able to:

1. Apply the fundamentals of automata theory to write DFA, NFA, Epsilon-NFA and conversion between them.
2. Prove the properties of regular languages using regular expressions.
3. Design context-free grammars (CFGs) and pushdown automata (PDAs) for formal languages.
4. Design Turing machines to solve the computational problems.
5. Explain the concepts of decidability and undecidability.

Context-Free Grammars, Parse Trees, Ambiguity in Grammars and Languages, Ambiguity in Grammars and Languages, Definition of the Pushdown Automaton, The Languages of a PDA, Equivalence of PDA's and CFG's, Deterministic Pushdown Automata.

TEXT BOOK: Sections 5.1, 5.2, 5.4, 6.1,6.2,6.3.1,6.4

Pushdown Automata (PDA)

Informally:

A PDA is an NFA-ε with a stack.

Transitions are modified to accommodate stack operations.

Questions:

What is a stack?

How does a stack help?

A DFA can “remember” only a finite amount of information, whereas a PDA can “remember” an infinite amount of (certain types of) information, in one memory-stack

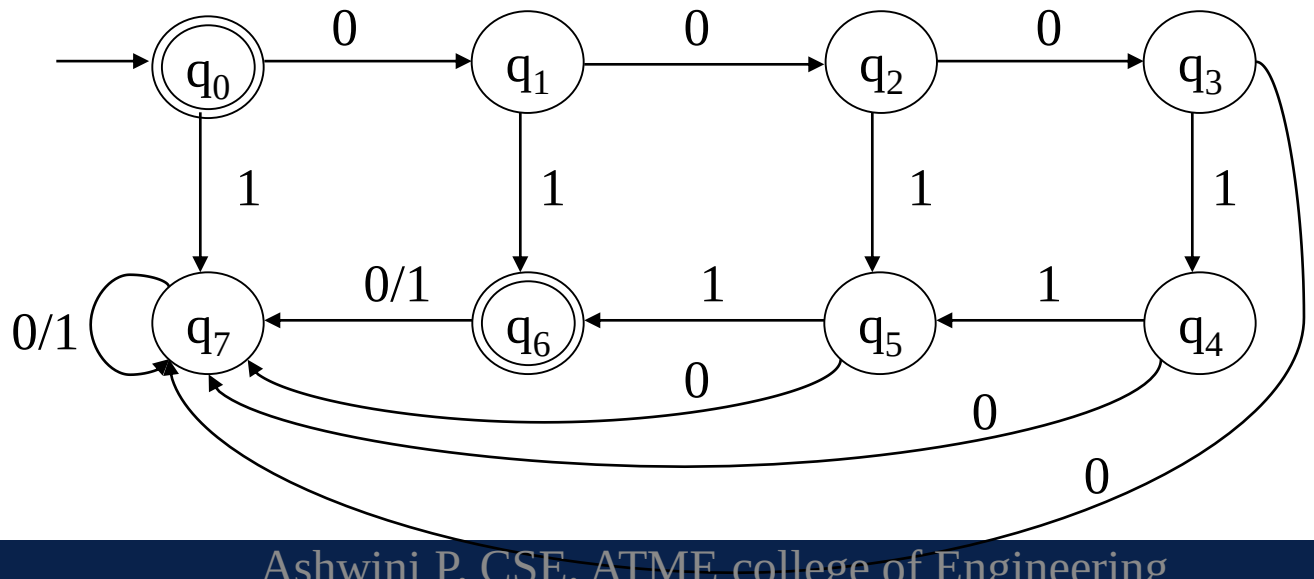
Example:

$\{0^n1^n \mid 0=<n\}$ is *not* regular, but

$\{0^n1^n \mid 0 \leq n \leq k$, for some fixed k$\}$ is regular, for any fixed k.

For k=3:

L = {ε, 01, 0011, 000111}



In a DFA, each state remembers a finite amount of information.

To get $\{0^n1^n \mid 0 \leq n\}$ with a DFA would require an infinite number of states using the preceding technique.

An infinite stack solves the problem for $\{0^n1^n \mid 0 \leq n\}$ as follows:

Read all 0's and place them on a stack

Read all 1's and match with the corresponding 0's on the stack

Only need two states to do this in a PDA

Similarly for $\{0^n1^m0^{n+m} \mid n,m \geq 0\}$

Formal Definition of a PDA

A pushdown automaton (PDA) is a seven-tuple:

$M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$

Q	A finite set of states
Σ	A finite input alphabet
Γ	A finite stack alphabet
q_0	The initial/starting state, q_0 is in Q
z_0	A starting stack symbol, is in Γ // need not always remain at the bottom of stack
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, where

$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow$ finite subsets of $Q \times \Gamma^*$

Consider the various parts of δ:

$$Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$$

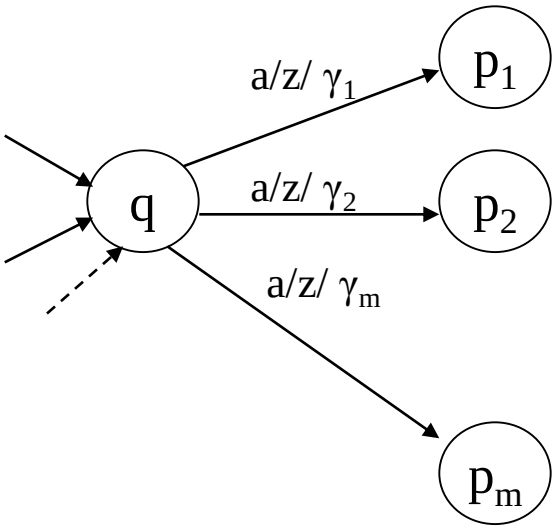
Q on the LHS means that at each step in a computation, a PDA must consider its' current state.
Γ on the LHS means that at each step in a computation, a PDA must consider the symbol on top of its' stack.
Σ U {ε} on the LHS means that at each step in a computation, a PDA may or may not consider the current input symbol, i.e., it may have epsilon transitions.

"Finite subsets" on the RHS means that at each step in a computation, a PDA may have several options.
Q on the RHS means that each option specifies a new state.
Γ* on the RHS means that each option specifies zero or more stack symbols that will replace the top stack symbol, but *in a specific sequence*.

Two types of PDA transitions:

$$\delta(q, a, z) = \{(p_1,\gamma_1), (p_2,\gamma_2),\ldots, (p_m,\gamma_m)\}$$

Current state is q
Current input symbol is a
Symbol currently on top of the stack z
Move to state p_i from q
Replace z with γ_i on the stack (leftmost symbol on top)
Move the input head to the next input symbol

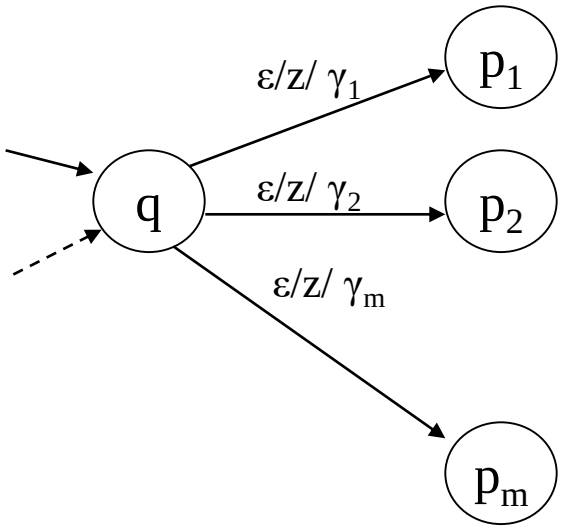


:

Two types of PDA transitions:

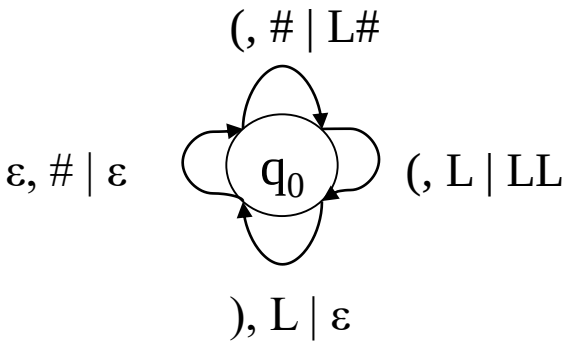
$$\delta(q, \varepsilon, z) = \{(p_1,\gamma_1), (p_2,\gamma_2),\ldots, (p_m,\gamma_m)\}$$

Current state is q
Current input symbol is not considered
Symbol currently on top of the stack z
Move to state p_i from q
Replace z with γ_i on the stack (leftmost symbol on top)
No input symbol is read



:

Transition Diagram:



Example Computation:

Current Input	Stack	Transition	
(())	#	-- initial status	
())	L#	(1)	- Could have applied rule (5), but
))	LL#	(3)	it would have done no good
)	L#	(4)	
ε	#	(4)	
ε	-	(5)	

Example PDA #1: For the language {x | x = wcw^r and *w* in {0,1}*, but sigma={0,1,c}}
Is this a regular language?

Note: length |x| is odd

M = ({q_1, q_2}, {0, 1, c}, {#, B, G}, δ, q_1, #, Ø)

δ:

(1)	δ(q_1, 0, #) = {(q_1, B#)}	(9)	δ(q_1, 1, #) = {(q_1, G#)}
(2)	δ(q_1, 0, B) = {(q_1, BB)}	(10)	δ(q_1, 1, B) = {(q_1, GB)}
(3)	δ(q_1, 0, G) = {(q_1, BG)}	(11)	δ(q_1, 1, G) = {(q_1, GG)}
(4)	δ(q_1, c, #) = {(q_2, #)}		
(5)	δ(q_1, c, B) = {(q_2, B)}		
(6)	δ(q_1, c, G) = {(q_2, G)}		
(7)	δ(q_2, 0, B) = {(q_2, ε)}	(12)	δ(q_2, 1, G) = {(q_2, ε)}
(8)	δ(q_2, ε, #) = {(q_2, ε)}		

Notes:

Stack grows leftwards
Only rule #8 is non-deterministic.
Rule #8 is used to pop the final stack symbol off at the end of a computation.

Example Computation:

(1) $\delta(q_1, 0, \#) = \{(q_1, B\#)\}$
(2) $\delta(q_1, 0, B) = \{(q_1, BB)\}$
(3) $\delta(q_1, 0, G) = \{(q_1, BG)\}$
(4) $\delta(q_1, c, \#) = \{(q_2, \#)\}$
(5) $\delta(q_1, c, B) = \{(q_2, B)\}$
(6) $\delta(q_1, c, G) = \{(q_2, G)\}$
(7) $\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$
(8) $\delta(q_2, \varepsilon, \#) = \{(q_2, \varepsilon)\}$
(9) $\delta(q_1, 1, \#) = \{(q_1, G\#)\}$
(10) $\delta(q_1, 1, B) = \{(q_1, GB)\}$
(11) $\delta(q_1, 1, G) = \{(q_1, GG)\}$
(12) $\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$

State	Input	Stack	Rule Applied	Rules Applicable
q_1	**0**1c10	#		(1)
q_1	**1**c10	B#	(1)	(10)
q_1	**c**10	GB#	(10)	(6)
q_2	**1**0	GB#	(6)	(12)
q_2	**0**	B#	(12)	(7)
q_2	**ε**	#	(7)	(8)
q_2	ε	ε	(8)	-

Example Computation:

(1) $\delta(q_1, 0, \#) = \{(q_1, B\#)\}$
(2) $\delta(q_1, 0, B) = \{(q_1, BB)\}$
(3) $\delta(q_1, 0, G) = \{(q_1, BG)\}$
(4) $\delta(q_1, c, \#) = \{(q_2, \#)\}$
(5) $\delta(q_1, c, B) = \{(q_2, B)\}$
(6) $\delta(q_1, c, G) = \{(q_2, G)\}$
(7) $\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$
(8) $\delta(q_2, \varepsilon, \#) = \{(q_2, \varepsilon)\}$
(9) $\delta(q_1, 1, \#) = \{(q_1, G\#)\}$
(10) $\delta(q_1, 1, B) = \{(q_1, GB)\}$
(11) $\delta(q_1, 1, G) = \{(q_1, GG)\}$
(12) $\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$

State	Input	Stack	Rule Applied
q_1	**1**c1	#	
q_1	**c**1	G#	(9)
q_2	**1**	G#	(6)
q_2	**ε**	#	(12)
q_2	ε	ε	(8)

Questions:

Why isn't $\delta(q_2, 0, G)$ defined?
Why isn't $\delta(q_2, 1, B)$ defined?

TRY: 11c1

Example PDA #2: For the language $\{x \mid x = ww^r \text{ and } w \text{ in } \{0,1\}^*\}$

Note: length |x| is even

$M = (\{q_1, q_2\}, \{0, 1\}, \{\#, B, G\}, \delta, q_1, \#, \emptyset)$

δ:

(1) $\delta(q_1, 0, \#) = \{(q_1, B\#)\}$

(2) $\delta(q_1, 1, \#) = \{(q_1, G\#)\}$

(3) $\delta(q_1, 0, B) = \{(q_1, BB), (q_2, \varepsilon)\}$

(4) $\delta(q_1, 0, G) = \{(q_1, BG)\}$

(5) $\delta(q_1, 1, B) = \{(q_1, GB)\}$

(6) $\delta(q_1, 1, G) = \{(q_1, GG), (q_2, \varepsilon)\}$

(7) $\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$

(8) $\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$

(9) $\delta(q_1, \varepsilon, \#) = \{(q_2, \#)\}$

(10) $\delta(q_2, \varepsilon, \#) = \{(q_2, \varepsilon)\}$

Notes:

Rules #3 and #6 are non-deterministic: two options each

Rules #9 and #10 are used to pop the final stack symbol off at the end of a computation.

Example Computation:

(1) $\delta(q_1, 0, \#) = \{(q_1, B\#)\}$
(2) $\delta(q_1, 1, \#) = \{(q_1, G\#)\}$
(3) $\delta(q_1, 0, B) = \{(q_1, BB), (q_2, \varepsilon)\}$
(4) $\delta(q_1, 0, G) = \{(q_1, BG)\}$
(5) $\delta(q_1, 1, B) = \{(q_1, GB)\}$
(6) $\delta(q_1, 1, G) = \{(q_1, GG), (q_2, \varepsilon)\}$
(7) $\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$
(8) $\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$
(9) $\delta(q_1, \varepsilon, \#) = \{(q_2, \varepsilon)\}$
(10) $\delta(q_2, \varepsilon, \#) = \{(q_2, \varepsilon)\}$

State	Input	Stack	Rule Applied	Rules Applicable
q_1	000000	#		(1), (9)
q_1	00000	B#	(1)	(3), both options
q_1	0000	BB#	(3) option #1	(3), both options
q_1	000	BBB#	(3) option #1	(3), both options
q_2	00	BB#	(3) option #2	(7)
q_2	0	B#	(7)	(7)
q_2	ε	#	(7)	(10)
q_2	ε	ε	(10)	

Questions:

What is rule #10 used for?
What is rule #9 used for?
Why do rules #3 and #6 have options?
Why don't rules #4 and #5 have similar options? [transition not possible if the previous input symbol was different]

Negative Example Computation:

(1) $\delta(q_1, 0, \#) = \{(q_1, B\#)\}$
(2) $\delta(q_1, 1, \#) = \{(q_1, G\#)\}$
(3) $\delta(q_1, 0, B) = \{(q_1, BB), (q_2, \varepsilon)\}$
(4) $\delta(q_1, 0, G) = \{(q_1, BG)\}$
(5) $\delta(q_1, 1, B) = \{(q_1, GB)\}$
(6) $\delta(q_1, 1, G) = \{(q_1, GG), (q_2, \varepsilon)\}$
(7) $\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$
(8) $\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$
(9) $\delta(q_1, \varepsilon, \#) = \{(q_2, \varepsilon)\}$
(10) $\delta(q_2, \varepsilon, \#) = \{(q_2, \varepsilon)\}$

State	Input	Stack	Rule Applied
q_1	000	#	
q_1	00	B#	(1)
q_1	0	BB#	(3) option #1
			(q2, 0, #) by option 2
q_1	ε	BBB#	(3) option #1 -crashes, no-rule to apply-
			(q2, ε, B#) by option 2
			-rejects: end of string but not empty stack-

Example Computation:

(1) $\delta(q_1, 0, \#) = \{(q_1, B\#)\}$
(2) $\delta(q_1, 1, \#) = \{(q_1, G\#)\}$
(3) $\delta(q_1, 0, B) = \{(q_1, BB), (q_2, \varepsilon)\}$
(4) $\delta(q_1, 0, G) = \{(q_1, BG)\}$
(5) $\delta(q_1, 1, B) = \{(q_1, GB)\}$
(6) $\delta(q_1, 1, G) = \{(q_1, GG), (q_2, \varepsilon)\}$
(7) $\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$
(8) $\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$
(9) $\delta(q_1, \varepsilon, \#) = \{(q_2, \varepsilon)\}$
(10) $\delta(q_2, \varepsilon, \#) = \{(q_2, \varepsilon)\}$

State	Input	Stack	Rule Applied	
q_1	010010	#		
q_1	10010	B#	(1)	From (1) and (9)
q_1	0010	GB#	(5)	
q_1	010	BGB#	(4)	
q_2	10	GB#	(3) option #2	
q_2	0	B#	(8)	
q_2	ε	#	(7)	
q_2	ε	ε	(10)	

Exercises:

0011001100 *// how many total options the machine (or you!) may need to try before rejection?*
011110
0111

Formal Definitions for PDAs

Let M = (Q, Σ, Γ, δ, q_0, z_0, F) be a PDA.

Definition: An *instantaneous description* (ID) is a triple (q, w, γ), where q is in Q, w is in Σ* and γ is in Γ*.

q is the current state
w is the unused input
γ is the current stack contents

Example: (for PDA #2)

(q_1, 111, GBR) (q_1, 11, GGBR)

(q_1, 111, GBR) (q_2, 11, BR)

(q_1, 000, GR) (q_2, 00, R)

Let M = (Q, Σ, Γ, δ, q_0, z_0, F) be a PDA.

Definition: Let a be in Σ U {ε}, w be in Σ*, z be in Γ, and α and β both be in Γ*. Then:

$$(q, aw, z\alpha) \vdash_M (p, w, \beta\alpha)$$

if δ(q, a, z) contains (p, β).

Intuitively, if I and J are instantaneous descriptions, then I |— J means that J follows from I by one transition.

Examples: (PDA #2)

(q_1, 111, GBR) |— (q_1, 11, GGBR) β=GG, w=11, and — (6) option #1, with a=1, z=G, α= BR

(q_1, 111, GBR) |— (q_2, 11, BR) w=11, and — (6) option #2, with a=1, z=G, β= ε, α= BR

(q_1, 000, GR) |— (q_2, 00, R) — Is *not* true, For any a, z, β, w and α

Examples: (PDA #1)

(q_1, (())), L#) |— (q_1, ())),LL#) — (3)

Definition: |—* is the reflexive and transitive closure of |—.

I |—* I for each instantaneous description I

If I |— J and J |—* K then I |—* K

Intuitively, if I and J are instantaneous descriptions, then I |—* J means that J follows from I by zero or more transitions.

Definition: Let M = (Q, Σ, Γ, δ, q_0, z_0, F) be a PDA. The *language accepted by empty stack*, denoted L_E(M), is the set

$$\{w \mid (q_0, w, z_0) \vdash^* (p, \varepsilon, \varepsilon) \text{ for some } p \text{ in } Q\}$$

Definition: Let M = (Q, Σ, Γ, δ, q_0, z_0, F) be a PDA. The *language accepted by final state*, denoted L_F(M), is the set

$$\{w \mid (q_0, w, z_0) \vdash^* (p, \varepsilon, \gamma) \text{ for some } p \text{ in } F \text{ and } \gamma \text{ in } \Gamma^*\}$$

Definition: Let M = (Q, Σ, Γ, δ, q_0, z_0, F) be a PDA. The *language accepted by empty stack and final state*, denoted L(M), is the set

$$\{w \mid (q_0, w, z_0) \vdash^* (p, \varepsilon, \varepsilon) \text{ for some } p \text{ in } F\}$$

Lemma 1: Let $L = L_E(M_1)$ for some PDA M_1. Then there exits a PDA M_2 such that $L = L_F(M_2)$.

Lemma 2: Let $L = L_F(M_1)$ for some PDA M_1. Then there exits a PDA M_2 such that $L = L_E(M_2)$.

Theorem: Let L be a language. Then there exits a PDA M_1 such that $L = L_F(M_1)$ if and only if there exists a PDA M_2 such that $L = L_E(M_2)$.

Corollary: The PDAs that accept by empty stack and the PDAs that accept by final state define the same class of languages.

Note: Similar lemmas and theorems could be stated for PDAs that accept by both final state and empty stack.

Back to CFG again:
PDA equivalent to CFG

Definition: Let G = (V, T, P, S) be a CFL. If every production in P is of the form

$$A \rightarrow a\alpha$$

Where A is in V, *a* is in T, and α is in V*, then G is said to be in Greibach Normal Form (GNF).

Only one non-terminal in front.

Example:

S –> aAB | bB
A –> aA | a
B –> bB | c

Language: $(aa^+ + b)b^+c$

Theorem: Let L be a CFL. Then L – {ε} is a CFL.

Theorem: Let L be a CFL not containing {ε}. Then there exists a GNF grammar G such that L = L(G).

Lemma 1: Let *L* be a CFL. Then there exists a PDA *M* such that L = L_E(M).

Proof: Assume without loss of generality that ε is not in *L*. The construction can be modified to include ε later.

Let G = (V, T, P, S) be a CFG, and assume without loss of generality that *G* is in GNF. Construct M = (Q, Σ, Γ, δ, q, z, Ø) where:

Q = {q}
Σ = T
Γ = V
z = S
δ: for all *a* in Σ and *A* in Γ, δ(q, a, A) contains (q, γ)
if A –> aγ is in *P* or rather:
δ(q, a, A) = {(q, γ) | A –> aγ is in *P* and γ is in Γ*},
for all *a* in Σ and *A* in Γ

For a given string *x* in Σ* , *M* will attempt to simulate a leftmost derivation of *x* with *G*.

Example #1: Consider the following CFG in GNF.

S –> aS
S –> a

G is in GNF
$L(G) = a^+$

Construct M as:

Q = {q}
Σ = T = {a}
Γ = V = {S}
z = S

δ(q, a, S) = {(q, S), (q, ε)}
δ(q, ε, S) = Ø

Is δ complete?

Example #2: Consider the following CFG in GNF.

(1) S –> aA
(2) S –> aB
(3) A –> aA
(4) A –> aB
(5) B –> bB
(6) B –> b

G is in GNF
$L(G) = a^+ b^+$ // This looks ok to me, one, two or more *a*'s in the start

[Can you write a simpler equivalent CFG? Will it be GNF?]

Construct M as:
Q = {q}
Σ = T = {a, b}
Γ = V = {S, A, B}
z = S

(1) δ(q, a, S) = {(q, A), (q, B)} From productions #1 and 2, S->aA, S->aB
(2) δ(q, a, A) = {(q, A), (q, B)} From productions #3 and 4, A->aA, A->aB
(3) δ(q, a, B) = Ø
(4) δ(q, b, S) = Ø
(5) δ(q, b, A) = Ø
(6) δ(q, b, B) = {(q, B), (q, ε)} From productions #5 and 6, B->bB, B->b
(7) δ(q, ε, S) = Ø
(8) δ(q, ε, A) = Ø
(9) δ(q, ε, B) = Ø *Is δ complete?*

For a string w in L(G) the PDA M will simulate a leftmost derivation of w.

If w is in L(G) then $(q, w, z_0) \vdash^* (q, \varepsilon, \varepsilon)$

If $(q, w, z_0) \vdash^* (q, \varepsilon, \varepsilon)$ then w is in L(G)

Consider generating a string using G. Since G is in GNF, each sentential form in a *leftmost* derivation has form:

$$\Rightarrow t_1 t_2 \ldots t_i A_1 A_2 \ldots A_m$$

terminals non-terminals

And each step in the derivation (i.e., each application of a production) adds a terminal and some non-terminals.

$$A_1 \rightarrow t_{i+1}\alpha$$

$$\Rightarrow t_1 t_2 \ldots t_i t_{i+1} \alpha A_1 A_2 \ldots A_m$$

Each transition of the PDA simulates one derivation step. Thus, the i^{th} step of the PDAs' computation corresponds to the i^{th} step in a corresponding leftmost derivation with the grammar.

After the i^{th} step of the computation of the PDA, $t_1 t_2 \ldots t_{i+1}$ are the symbols that have already been read by the PDA and $\alpha A_1 A_2 \ldots A_m$ are the stack contents.

For each leftmost derivation of a string generated by the grammar, there is an equivalent accepting computation of that string by the PDA.

Each sentential form in the leftmost derivation corresponds to an instantaneous description in the PDA's corresponding computation.

For example, the PDA instantaneous description corresponding to the sentential form:

$$=> t_1 t_2 \ldots t_i A_1 A_2 \ldots A_m$$

would be:

$$(q, t_{i+1} t_{i+2} \ldots t_n , A_1 A_2 \ldots A_m)$$

Example: Using the grammar from example #2:

S => aA (1)
=> aaA (3)
=> aaaA (3)
=> aaaaB (4)
=> aaaabB (5)
=> aaaabb (6)

Grammar:

(1) S –> aA
(2) S –> aB
(3) A –> aA
(4) A –> aB
(5) B –> bB
(6) B –> b

G is in GNF
$L(G) = a^+b^+$

(1) δ(q, a, S) = {(q, A), (q, B)}
(2) δ(q, a, A) = {(q, A), (q, B)}
(3) δ(q, a, B) = Ø
(4) δ(q, b, S) = Ø
(5) δ(q, b, A) = Ø
(6) δ(q, b, B) = {(q, B), (q, ε)}
(7) δ(q, ε, S) = Ø
(8) δ(q, ε, A) = Ø
(9) δ(q, ε, B) = Ø

The corresponding computation of the PDA:

	(rule#)/right-side#
(q, aaaabb, S) |— (q, aaabb, A)	(1)/1
|— (q, aabb, A)	(2)/1
|— (q, abb, A)	(2)/1
|— (q, bb, B)	(2)/2
|— (q, b, B)	(6)/1
|— (q, ε, ε)	(6)/2

String is read
Stack is emptied
Therefore the string is accepted by the PDA

Another Example: Using the PDA from example #2:

(q, aabb, S) |— (q, abb, A) (1)/1
|— (q, bb, B) (2)/2
|— (q, b, B) (6)/1
|— (q, ε, ε) (6)/2

The corresponding derivation using the grammar:

S => aA (1)
=> aaB (4)
=> aabB (5)
=> aabb (6)

Example #3: Consider the following CFG in GNF.

(1) S –> aABC
(2) A –> a
(3) B –> b
(4) C –> cAB
(5) C –> cC

G is in GNF

$aab\ cc^*\ ab$

Language?

Construct M as:

Q = {q}
Σ = T = {a, b, c}
Γ = V = {S, A, B, C}
z = S

(1) δ(q, a, S) = {(q, ABC)} S->aABC
(2) δ(q, a, A) = {(q, ε)} A->a
(3) δ(q, a, B) = Ø
(4) δ(q, a, C) = Ø
(5) δ(q, b, S) = Ø
(6) δ(q, b, A) = Ø
(7) δ(q, b, B) = {(q, ε)} B->b
(8) δ(q, b, C) = Ø
(9) δ(q, c, S) = Ø
(10) δ(q, c, A) = Ø
(11) δ(q, c, B) = Ø
(12) δ(q, c, C) = {(q, AB), (q, C))} // C->cAB|cC
(13) δ(q, ε, S) = Ø
(14) δ(q, ε, A) = Ø
(15) δ(q, ε, B) = Ø
(16) δ(q, ε, C) = Ø

Notes:

Recall that the grammar *G* was required to be in GNF before the construction could be applied.

As a result, it was assumed at the start that ε was not in the context-free language *L*.

What if ε is in *L*? You need to add ε back.

Suppose ε is in L:

1) First, let L' = L – {ε}

Fact: If *L* is a CFL, then L' = L – {ε} is a CFL.

By an earlier theorem, there is GNF grammar *G* such that L' = L(G).

2) Construct a PDA *M* such that L' = $L_E(M)$

How do we modify *M* to accept ε?

Add δ(q, ε, S) = {(q, ε)}? *NO!!*

Counter Example:

Consider L = {ε, b, ab, aab, aaab, …}= ε + a*b Then L' = {b, ab, aab, aaab, …} = a*b

The GNF CFG for L':

P:

(1) S –> aS
(2) S –> b

The PDA M Accepting L':

Q = {q}
Σ = T = {a, b}
Γ = V = {S}
z = S

δ(q, a, S) = {(q, S)}
δ(q, b, S) = {(q, ε)}
δ(q, ε, S) = Ø

How to add ε to L' now?

δ(q, a, S) = {(q, S)}
δ(q, b, S) = {(q, ε)}
δ(q, ε, S) = Ø

If δ(q, ε, S) = {(q, ε)} is added then:
L(M) = {ε, a, aa, aaa, …, b, ab, aab, aaab, …}, wrong!

It is like, S -> aS | b | ε
which is wrong!
Correct grammar should be:
(0) S_1 -> ε | S, with new starting non-terminal S_1
(1) S –> aS
(2) S –> b

For PDA, add a new *Stack-bottom symbol* S_1, with new transitions:
$\delta(q, \varepsilon, S_1) = \{(q, \varepsilon), (q, S)\}$, where *S* was the previous stack-bottom of *M*

Alternatively, add a new *start* state q' with transitions:
δ(q', ε, S) = {(q', ε), (q, S)}

Lemma 1: Let *L* be a CFL. Then there exists a PDA *M* such that L = L_E(M).

Lemma 2: Let *M* be a PDA. Then there exists a CFG grammar *G* such that L_E(M) = L(G).

Can you prove it?

First step would be to transform an arbitrary PDA to a single state PDA!

Theorem: Let *L* be a language. Then there exists a CFG *G* such that L = L(G) iff there exists a PDA *M* such that L = L_E(M).

Corollary: The PDAs define the CFLs.

0^n1^n, n>=1
S -> 0S1 | 01

GNF:
S -> $0SS_1$ | $0S_1$
S_1 -> 1

Note: in PDA the symbol S will float on top, rather than stay at the bottom!
Acceptance of string by removing last S_1 at stack bottom

Ignore this slide

How about language like: (((())())(), nested

M = ({q_1,q_2}, {"(", ")"}, {L, #}, δ, q_1, #, Ø)

δ:

(1) δ(q_1, (, #) = {(q_1, L#)}

(2) δ(q_1,), #) = Ø // illegal, string rejected

(3) δ(q_1, (, L) = {(q_1, LL)}

(4) δ(q_1,), L) = {(q_2, ε)}

(5) δ(q_2,), L) = {(q_2, ε)}

(6) δ(q_2, (, L) = {(q_1, LL)} // not balanced yet, but start back anyway

(7) δ(q_2, (, #) = {(q_1, L#)} // start afresh again

(8) δ(q_2, ε, #) = {(q_2, ε)} // end of string & stack hits bottom, accept

(9) δ(q_1, ε, #) = {(q_1, ε)} // special rule for empty string

(10) δ(q_1, ε, L) = Ø // illegal, end of string but more L in stack

Total number of transitions? Verify all carefully.

THEORY OF COMPUTATION		Semester	V
Course Code	**BCS503**	CIE Marks	50
Teaching Hours/Week (L: T:P: S)	(3:2:0:0)	SEE Marks	50
Total Hours of Pedagogy	50	Total Marks	100
Credits	04	Exam Hours	3
Examination type (SEE)	Theory		

Course outcome (Course Skill Set)

At the end of the course, the student will be able to:

1. Apply the fundamentals of automata theory to write DFA, NFA, Epsilon-NFA and conversion between them.
2. Prove the properties of regular languages using regular expressions.
3. Design context-free grammars (CFGs) and pushdown automata (PDAs) for formal languages.
4. Design Turing machines to solve the computational problems.
5. Explain the concepts of decidability and undecidability.

Module-4	10 Hours
Normal Forms for Context-Free Grammars, The Pumping Lemma for Context-Free Languages, Closure Properties of Context-Free Languages. **TEXT BOOK: Sections 7.1, 7.2, 7.3**	

- Introduction
- Chomsky normal form
  - Preliminary simplifications
  - Final steps
- Greibach Normal Form
  - Algorithm (Example)
- Summary

Introduction

Grammar: G = (V, T, P, S)

Terminals	T = { a, b }
Variables	V = A, B, C
Start Symbol	S
Production	P = S → A

Grammar example

S → aBSc
S → abc
Ba → aB
Bb → bb

$L = \{ a^n b^n c^n \mid n \geq 1 \}$

$S \Rightarrow aBSc \Rightarrow aBabcc \Rightarrow aaBbcc \Rightarrow aabbcc$

Context free grammar

The head of any production contains only one non-terminal symbol

$S \rightarrow P$

$P \rightarrow aPb$

$P \rightarrow \varepsilon$

$$L = \{ a^n b^n \mid n \geq 0 \}$$

- Introduction
- Chomsky normal form
  - Preliminary simplifications
  - Final simplification
- Greibach Normal Form
  - Algorithm (Example)
- Summary

A context free grammar is said to be in **Chomsky Normal Form** if all productions are in the following form:

$$A \rightarrow BC$$
$$A \rightarrow \alpha$$

- A, B and C are non terminal symbols
- α is a terminal symbol

- Introduction
- Chomsky normal form
  - Preliminary simplifications
  - Final steps
- Greibach Normal Form
  - Algorithm (Example)
- Summary

There are three preliminary simplifications

1. Eliminate Useless Symbols
2. Eliminate ε productions
3. Eliminate unit productions

Eliminate Useless Symbols

We need to determine if the symbol is useful by identifying if a symbol is **generating** and is **reachable**

- X is **generating** if $X \overset{*}{\Rightarrow} \omega$ for some terminal string ω.
- X is **reachable** if there is a derivation $S \overset{*}{\Rightarrow} \alpha X \beta$ for some α and β

Example: Removing **non-generating** symbols

$S \rightarrow AB \mid a$
$A \rightarrow b$

Initial CFL grammar

$S \rightarrow AB \mid a$
$A \rightarrow b$

Identify generating symbols

$S \rightarrow a$
$A \rightarrow b$

Remove non-generating

Example: Removing **non-reachable** symbols

$S \rightarrow a$
$A \rightarrow b$

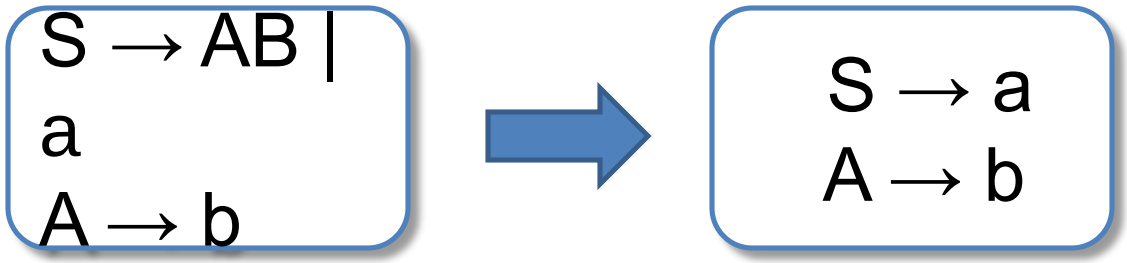
Identify reachable symbols

$S \rightarrow a$

Eliminate non-reachable

The order is important.

Looking first for non-reachable symbols and then for non-generating symbols can still leave some useless symbols.



Finding **generating** symbols

If there is a production $A \rightarrow \alpha$, and every symbol of α is already known to be generating. Then A is generating

$S \rightarrow AB \mid a$

$A \rightarrow b$

We cannot use S → AB because B has not been established to be generating

Finding **reachable** symbols

S is surely reachable. All symbols in the body of a production with S in the head are reachable.

$S \rightarrow AB \mid a$

$A \rightarrow b$

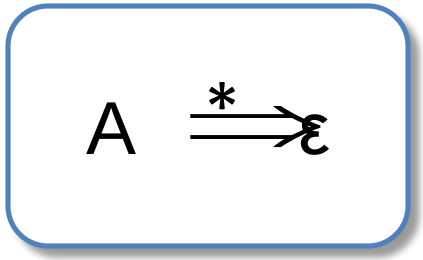
In this example the symbols {S, A, B, a, b} are reachable.

There are three preliminary simplifications

1. Eliminate Useless Symbols
2. Eliminate ε productions
3. Eliminate unit productions

Eliminate ε Productions

- In a grammar ε productions are convenient but not essential
- If L has a CFG, then L – {ε} has a CFG



Nullable variable

If A is a nullable variable

- Whenever A appears on the body of a production A might or might not derive ε

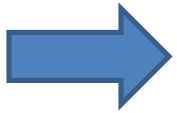
S → ASA | aB
A → B | S
B → b | ε

Nullable: {A, B}

Eliminate ε Productions

- Create two version of the production, one with the nullable variable and one without it
- Eliminate productions with ε bodies

$S \rightarrow ASA \mid aB$
$A \rightarrow B \mid S$
$B \rightarrow b \mid \varepsilon$



$S \rightarrow ASA \mid aB \mid AS \mid SA \mid S \mid a$
$A \rightarrow B \mid S$
$B \rightarrow b$

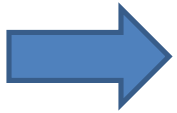
Eliminate ε Productions

- Create two version of the production, one with the nullable variable and one without it
- Eliminate productions with ε bodies

$S \rightarrow ASA \mid aB$

$A \rightarrow B \mid S$

$B \rightarrow b \mid \varepsilon$



$S \rightarrow ASA \mid aB \mid AS \mid SA \mid S \mid a$

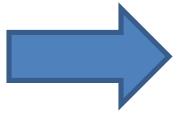
$A \rightarrow B \mid S$

$B \rightarrow b$

Eliminate ε Productions

- Create two version of the production, one with the nullable variable and one without it
- Eliminate productions with ε bodies

$S \rightarrow ASA \mid aB$
$A \rightarrow B \mid S$
$B \rightarrow b \mid \varepsilon$



$S \rightarrow ASA \mid aB \mid AS \mid SA \mid S \mid a$
$A \rightarrow B \mid S$
$B \rightarrow b$

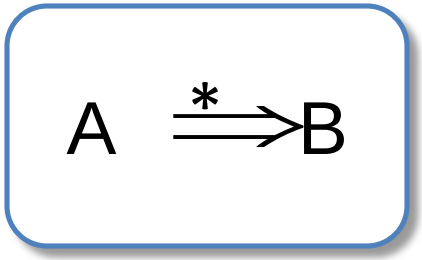
There are three preliminary simplifications

1. Eliminate Useless Symbols
2. Eliminate ε productions
3. Eliminate unit productions

Eliminate unit productions

A unit production is one of the form $A \rightarrow B$ where both A and B are variables

Identify **unit pairs**



$A \rightarrow B$, $B \rightarrow \omega$, then $A \rightarrow \omega$

Example:

T = {*, +, (,), a, b, 0, 1}

I → a | b | Ia | Ib | I0 | I1
F → I | (E)
T → F | T * F
E → T | E + T

Basis: (A, A) is a unit pair of any variable A, if $A \overset{*}{\Longrightarrow} A$ by 0 steps.

Pairs	Productions
(E, E)	E → E + T
(E, T)	E → T * F
(E, F)	E → (E)
(E, I)	E → a | b | Ia | Ib | I0 | I1
(T, T)	T → T * F
(T, F)	T → (E)
(T, I)	T → a | b | Ia |Ib | I0 | I1
(F, F)	F → (E)
(F, I)	F → a | b | Ia | Ib | I0 | I1
(I, I)	I → a | b | Ia | Ib | I0 | I1

Example:

Pairs	Productions
…	…
(T, T)	**T → T * F**
(T, F)	**T → (E)**
(T, I)	**T → a | b | Ia |Ib | I0 | I1**
…	…

I → a | b | Ia | Ib | I0 | I1

E → E + T | T * F | (E) | a | b | Ia | Ib | I0 | I1

T → T * F | (E) | a | b | Ia | Ib | I0 | I1

F → (E) | a | b | Ia | Ib | I0 | I1

- Introduction
- Chomsky normal form
  - Preliminary simplifications
  - Final steps
- Greibach Normal Form
  - Algorithm (Example)
- Summary

A context free grammar is said to be in **Chomsky Normal Form** if all productions are in the following form:

$$A \rightarrow BC$$
$$A \rightarrow \alpha$$

- A, B and C are non terminal symbols
- α is a terminal symbol

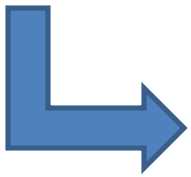
Chomsky Normal Form (CNF)

Starting with a CFL grammar with the preliminary simplifications performed

1. Arrange that all bodies of length 2 or more to consists only of variables.
2. Break bodies of length 3 or more into a cascade of productions, each with a body consisting of two variables.

Step 1: For every terminal α that appears in a body of length 2 or more create a new variable that has only one production.

E → E + T | T * F | (E) | a | b | Ia | Ib | I0 | I1
T → T * F | (E) | a | b | Ia | Ib | I0 | I1
F → (E) | a | b | Ia | Ib | I0 | I1
I → a | b | Ia | Ib | I0 | I1



E → EPT | TMF | LER | a | b | IA | IB | IZ | IO
T → TMF | LER | a | b | IA | IB | IZ | IO
F → LER | a | b | IA | IB | IZ | IO
I → a | b | IA | IB | IZ | IO
A → a B → b Z → 0 O → 1
P → + M → * L → (R →)

Step 2: Break bodies of length 3 or more adding more variables

E → E**PT** | T**MF** | L**ER** | a | b | IA | IB | IZ | IO

T → T**MF** | L**ER** | a | b | IA | IB | IZ | IO

F → L**ER** | a | b | IA | IB | IZ | IO

I → a | b | IA | IB | IZ | IO

A → a B → b Z → 0 O → 1

P → + M → * L → (R →)

C_1 → PT

C_2 → MF

C_3 → ER

- Introduction
- Chomsky normal form
  - Preliminary simplifications
  - Final steps
- Greibach Normal Form
  - Algorithm (Example)
- Summary

A context free grammar is said to be in **Greibach Normal Form** if all productions are in the following form:

$$A \rightarrow \alpha X$$

- A is a non terminal symbols
- α is a terminal symbol
- X is a sequence of non terminal symbols. It may be empty.

- Introduction
- Chomsky normal form
  - Preliminary simplifications
  - Final steps
- Greibach Normal Form
  - Algorithm (Example)
- Summary

Example:

CNF	New Labels	Updated CNF
$S \rightarrow XA \mid BB$	$S = A_1$	$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$B \rightarrow b \mid SB$	$X = A_2$	$A_4 \rightarrow b \mid A_1A_4$
$X \rightarrow b$	$A = A_3$	$A_2 \rightarrow b$
$A \rightarrow a$	$B = A_4$	$A_3 \rightarrow a$

Example:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow b \mid A_1A_4$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

First Step

$$A_i \rightarrow A_jX_k \quad j > i$$

X_k is a string of zero or more variables

✗ $A_4 \rightarrow A_1A_4$

Example:

First Step $A_i \rightarrow A_jX_k \quad j > i$

$A_4 \rightarrow \underline{A_1}A_4$

$A_4 \rightarrow \underline{A_2}A_3A_4 \mid A_4A_4A_4 \mid b$

$A_4 \rightarrow bA_3A_4 \mid A_4A_4A_4 \mid b$

$A_1 \rightarrow A_2A_3 \mid A_4A_4$

$A_4 \rightarrow b \mid A_1A_4$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Example:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow bA_3A_4 \mid A_4A_4A_4 \mid b$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

Second Step

Eliminate Left Recursions

$A \rightarrow A\,\alpha \mid \beta$

Can be written as

$A \rightarrow \beta\, A'$
$A' \rightarrow \alpha\, A' \mid \varepsilon$

✗ $A_4 \rightarrow A_4A_4A_4$

Example:

Second Step

Eliminate Left Recursions

$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$

$Z \rightarrow A_4A_4 \mid A_4A_4Z$

$A_1 \rightarrow A_2A_3 \mid A_4A_4$

$A_4 \rightarrow bA_3A_4 \mid A_4A_4A_4 \mid b$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Example:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$

$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$

$Z \rightarrow A_4A_4 \mid A_4A_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

$A \rightarrow \alpha X$

GNF

Example:

$A_1 \rightarrow \underline{A_2}A_3 \mid \underline{A_4}A_4$

$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$

$Z \rightarrow A_4A_4 \mid A_4A_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

$A_1 \rightarrow bA_3 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$

$Z \rightarrow bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$

Example:

$A_1 \rightarrow bA_3 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$
$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$
$Z \rightarrow bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

Grammar in Greibach Normal Form

Summary (Some properties)

- Every CFG that doesn't generate the empty string can be simplified to the Chomsky Normal Form and Greibach Normal Form
- The derivation tree in a grammar in CNF is a binary tree
- In the GNF, a string of length n has a derivation of exactly n steps
- Grammars in normal form can facilitate proofs
- CNF is used as starting point in the algorithm CYK

Thank You!

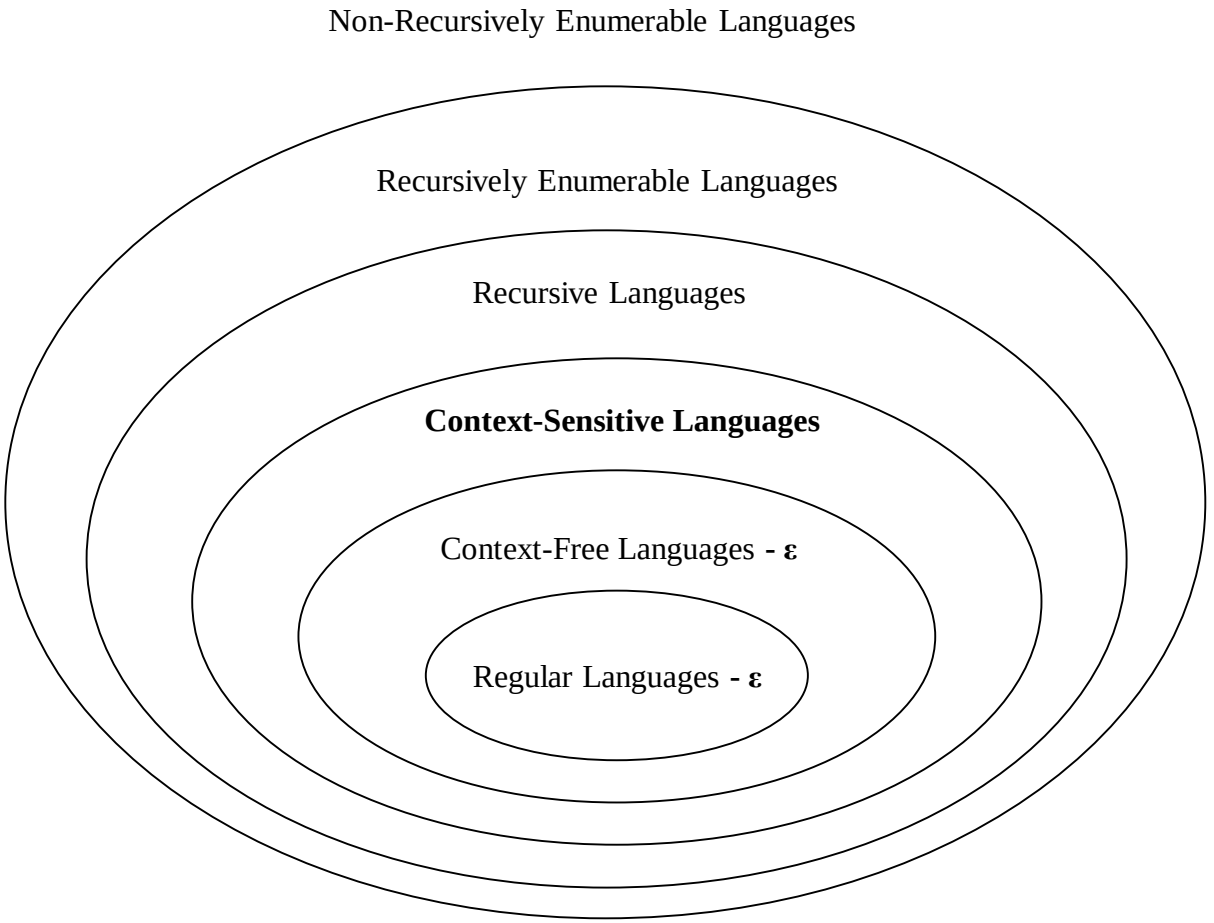
THEORY OF COMPUTATION		Semester	V
Course Code	**BCS503**	CIE Marks	50
Teaching Hours/Week (L: T:P: S)	(3:2:0:0)	SEE Marks	50
Total Hours of Pedagogy	50	Total Marks	100
Credits	04	Exam Hours	3
Examination type (SEE)	Theory		

Course outcome (Course Skill Set)

At the end of the course, the student will be able to:

1. Apply the fundamentals of automata theory to write DFA, NFA, Epsilon-NFA and conversion between them.
2. Prove the properties of regular languages using regular expressions.
3. Design context-free grammars (CFGs) and pushdown automata (PDAs) for formal languages.
4. Design Turing machines to solve the computational problems.
5. Explain the concepts of decidability and undecidability.

Module-5	10 Hours
Introduction to Turing Machines: Problems That Computers Cannot Solve, The Turing Machine, Programming Techniques for Turing Machines, Extensions to the Basic Turing Machine, Undecidability: A Language That Is Not Recursively Enumerable. **TEXT BOOK: Sections 8.1,8.2, 8.3,8.4, 9.1, 9.2**	


Non-Recursively Enumerable Languages
Recursively Enumerable Languages
Recursive Languages
Context-Sensitive Languages
Context-Free Languages - ε
Regular Languages - ε

Recursively enumerable languages are also known as *type 0* languages.

Context-sensitive languages are also known as *type 1* languages.

Context-free languages are also known as *type 2* languages.

Regular languages are also known as *type 3* languages.

TMs model the computing capability of a general purpose computer, which informally can be described as:

- Effective procedure
  - Finitely describable
  - Well defined, discrete, “mechanical” steps
  - Always terminates
- Computable function
  - A function computable by an effective procedure

TMs formalize the above notion.

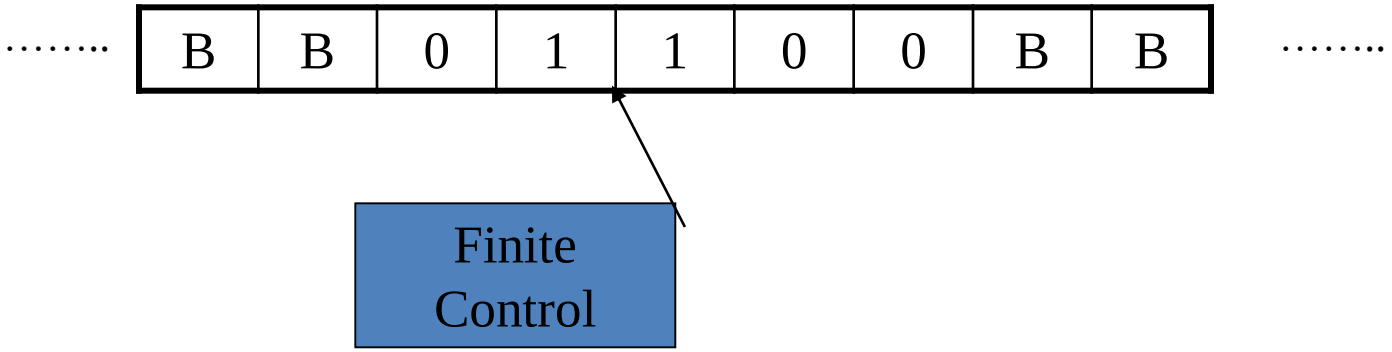
Church-Turing Thesis: There is an effective procedure for solving a problem if and only if there is a TM that halts for all inputs and solves the problem.

There are many other computing models, but all are equivalent to or subsumed by TMs.

There is no more powerful machine (Technically cannot be proved).

DFAs and PDAs do not model all effective procedures or computable functions, but only a subset.

Deterministic Turing Machine (DTM)



Two-way, infinite tape, broken into cells, each containing one symbol.

Two-way, read/write tape head.

An input string is placed on the tape, padded to the left and right infinitely with blanks, read/write head is positioned at the left end of input string.

Finite control, i.e., a program, containing the position of the read head, current symbol being scanned, and the current state.

In one move, depending on the current state and the current symbol being scanned, the TM 1) changes state, 2) prints a symbol over the cell being scanned, and 3) moves its' tape head one cell left or right.

Many modifications possible, but Church-Turing declares equivalence of all.

Formal Definition of a DTM

A DTM is a seven-tuple:

$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Q	A finite set of states
Σ	A finite input alphabet, which is a subset of Γ– {B}
Γ	A finite tape alphabet, which is a strict superset of Σ
B	A distinguished blank symbol, which is in Γ
q_0	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A next-move function, which is a *mapping* (i.e., may be undefined) from

$Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$

Intuitively, δ(q,s) specifies the next state, symbol to be written, and the direction of tape head movement by M after reading symbol s while in state q.

Example #1: {w | w is in {0,1}* and w ends with a 0}

0
00
10
10110
Not ε

$Q = \{q_0, q_1, q_2\}$
$\Gamma = \{0, 1, B\}$
$\Sigma = \{0, 1\}$
$F = \{q_2\}$
δ:

	0	1	B
$\rightarrow q_0$	$(q_0, 0, R)$	$(q_0, 1, R)$	(q_1, B, L)
q_1	$(q_2, 0, R)$	-	-
q_2^*	-	-	-

q_0 is the start state and the "scan right" state, until hits B
q_1 is the verify 0 state
q_2 is the final state

Exercises: Construct a DTM for each of the following.

{w | w is in {0,1}* and w ends in 00}
{w | w is in {0,1}* and w contains at least two 0's}
{w | w is in {0,1}* and w contains at least one 0 and one 1}
Just about anything else (simple) you can think of

Formal Definitions for DTMs

Let M = (Q, Σ, Γ, δ, q_0, B, F) be a TM.

Definition: An *instantaneous description* (ID) is a triple $\alpha_1 q \alpha_2$, where:

q, the current state, is in Q
$\alpha_1\alpha_2$, is in Γ*, and is the current tape contents up to the rightmost non-blank symbol, or the symbol to the left of the tape head, whichever is rightmost
The tape head is currently scanning the first symbol of α_2
At the start of a computation $\alpha_1 = \varepsilon$
If $\alpha_2 = \varepsilon$ then a blank is being scanned

Example: (for TM #1)

$q_0$0011 ⊢ X$q_1$011 ⊢ X0$q_1$11 ⊢ X$q_2$0Y1 ⊢ q_2X0Y1

X$q_0$0Y1 ⊢ XXq_1Y1 ⊢ XXY$q_1$1 ⊢ XXq_2YY ⊢ Xq_2XYY

XXq_0YY ⊢ XXYq_3Y ⊢ XXYYq_3 ⊢ XXYYBq_4

Suppose the following is the current ID of a DTM

$$x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n$$

Case 1) $\delta(q, x_i) = (p, y, L)$

(a) if i = 1 then $qx_1x_2\ldots x_{i-1}x_ix_{i+1}\ldots x_n \vdash pByx_2\ldots x_{i-1}x_ix_{i+1}\ldots x_n$

(b) else $x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n \vdash x_1x_2\ldots x_{i-2}px_{i-1}yx_{i+1}\ldots x_n$

If any suffix of $x_{i-1}yx_{i+1}\ldots x_n$ is blank then it is deleted.

Case 2) $\delta(q, x_i) = (p, y, R)$

$x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n \vdash x_1x_2\ldots x_{i-1}ypx_{i+1}\ldots x_n$

If i>n then the ID increases in length by 1 symbol

$x_1x_2\ldots x_nq \vdash x_1x_2\ldots x_nyp$

Definition: Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM, and let w be a string in Σ^*. Then w is *accepted* by M iff

$$q_0 w \vdash^* \alpha_1 p \alpha_2$$

where p is in F and α_1 and α_2 are in Γ^*

Definition: Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM. The *language accepted by M*, denoted L(M), is the set

$$\{w \mid w \text{ is in } \Sigma^* \text{ and } w \text{ is accepted by } M\}$$

Notes:

In contrast to FA and PDAs, if a TM simply *passes through* a final state then the string is accepted.

Given the above definition, no final state of a TM need to have any transitions.

Henceforth, this is our assumption.

If x is NOT in L(M) then M may enter an infinite loop, or halt in a non-final state.

Some TMs halt on ALL inputs, while others may not. In either case the language defined by TM is still well defined.

Definition: Let *L* be a language. Then *L* is *recursively enumerable* if there exists a TM *M* such that L = L(M).

If *L* is r.e. then L = L(M) for some TM *M*, and
If *x* is in *L* then *M* halts in a final (accepting) state.
If *x* is not in *L* then *M* may halt in a non-final (non-accepting) state or no transition is available, or loop forever.

Definition: Let *L* be a language. Then *L* is *recursive* if there exists a TM *M* such that L = L(M) and M halts on all inputs.

If *L* is recursive then L = L(M) for some TM *M*, and
If *x* is in *L* then *M* halts in a final (accepting) state.
If *x* is not in *L* then *M* halts in a non-final (non-accepting) state or no transition is available (does not go to infinite loop).

Notes:

The set of all recursive languages is a subset of the set of all recursively enumerable languages

Terminology is easy to confuse: A *TM* is not recursive or recursively enumerable, rather a *language* is recursive or recursively enumerable.

Closure Properties for Recursive and Recursively Enumerable Languages

TMs model General Purpose (GP) Computers:

If a TM can do it, so can a GP computer

If a GP computer can do it, then so can a TM

If you want to know if a TM can do X, then some equivalent question are:

Can a general purpose computer do X?

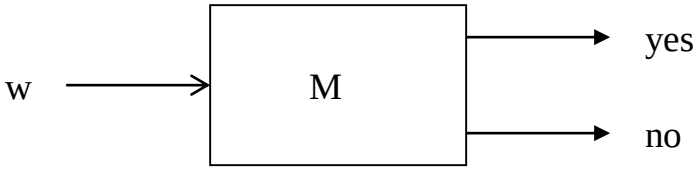
Can a C/C++/Java/etc. program be written to do X?

For example, is a language L recursive?

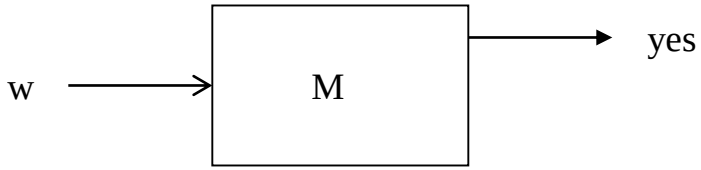
Can a C/C++/Java/etc. program be written that always halts and accepts L?

TM Block Diagrams:

If *L* is a recursive language, then a TM *M* that accepts *L* and always halts can be pictorially represented by a "chip" or "box" that has one input and two outputs.



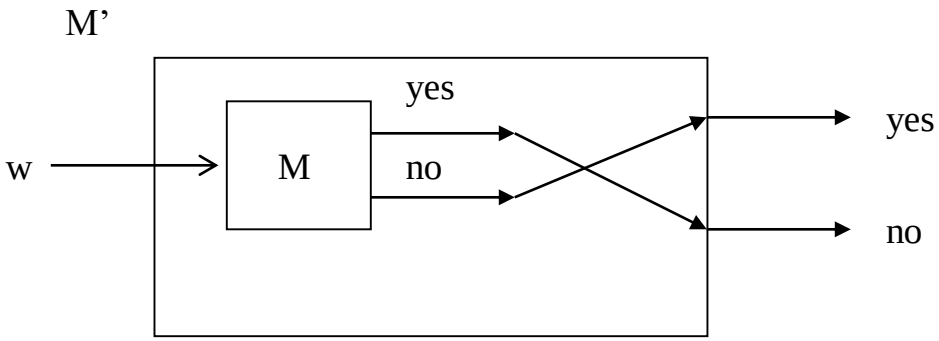
If *L* is a recursively enumerable language, then a TM *M* that accepts *L* can be pictorially represented by a "box" that has one output.



Conceivably, *M* could be provided with an output for "no," but this output cannot be counted on. Consequently, we simply ignore it.

Theorem 1: The recursive languages are closed with respect to complementation, i.e., if L is a recursive language, then so is $\overline{L} = \Sigma^* - L$

Proof: Let M be a TM such that L = L(M) and M always halts. Construct TM M' as follows:



Note That:

M' accepts iff M does not

M' always halts since M always halts

From this it follows that the complement of L is recursive. •

Question: How is the construction achieved? Do we simply complement the final states in the TM? No! A string in L could end up in the complement of L.

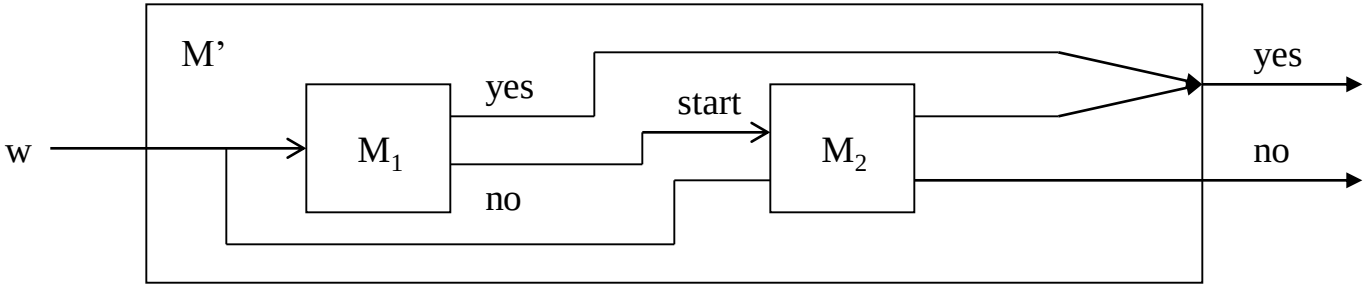
Suppose q_5 is an accepting state in M, but q_0 is not.

If we simply complemented the final and non-final states, then q_0 would be an accepting state in M' but q_5 would not.

Since q_0 is an accepting state, by definition all strings are accepted by M'

Theorem 2: The recursive languages are closed with respect to union, i.e., if L_1 and L_2 are recursive languages, then so is $L_3 = L_1 \cup L_2$

Proof: Let M_1 and M_2 be TMs such that $L_1 = L(M_1)$ and $L_2 = L(M_2)$ and M_1 and M_2 always halts. Construct TM M' as follows:



Note That:

$L(M') = L(M_1) \quad L(M_2)$

$L(M')$ is a subset of $L(M_1) \cup L(M_2)$

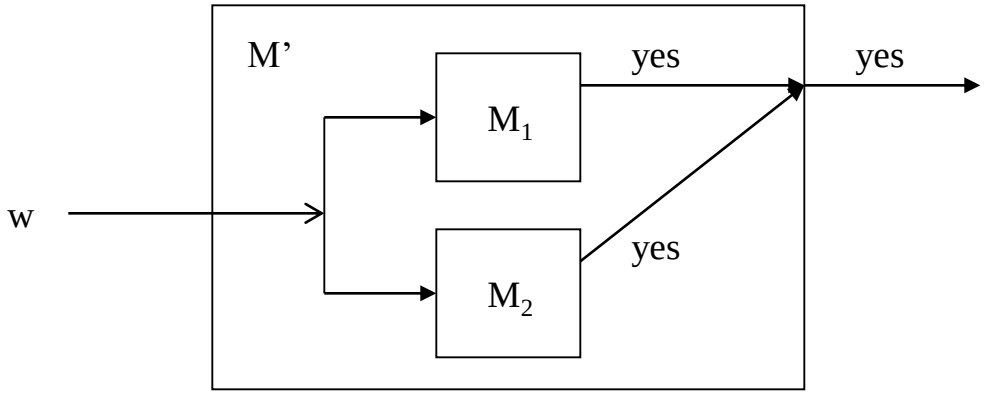
$L(M_1) \cup L(M_2)$ is a subset of $L(M')$

M' always halts since M_1 and M_2 always halt

It follows from this that $L_3 = L_1 \cup L_2$ is recursive. •

Theorem 3: The *recursive enumerable languages* are closed with respect to union, i.e., if L_1 and L_2 are recursively enumerable languages, then so is $L_3 = L_1 \cup L_2$

Proof: Let M_1 and M_2 be TMs such that $L_1 = L(M_1)$ and $L_2 = L(M_2)$. Construct M' as follows:



Note That:

$L(M') = L(M_1) \cup L(M_2)$

$L(M')$ is a subset of $L(M_1) \cup L(M_2)$

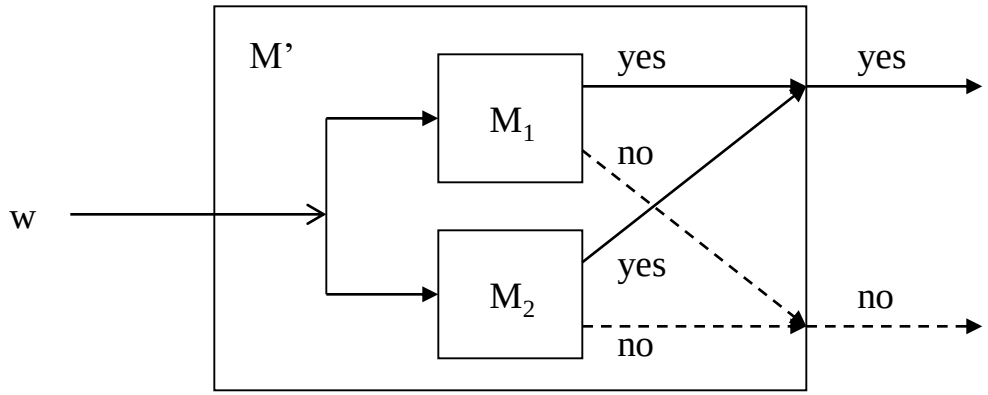
$L(M_1) \cup L(M_2)$ is a subset of $L(M')$

M' halts and accepts iff M_1 or M_2 halts and accepts

It follows from this that $L_3 = L_1 \cup L_2$ is recursively enumerable. •

Question: *How do you run two TMs in parallel?*

Suppose, M_1 and M_2 had outputs for "no" in the previous construction, and these were transferred to the "no" output for M'



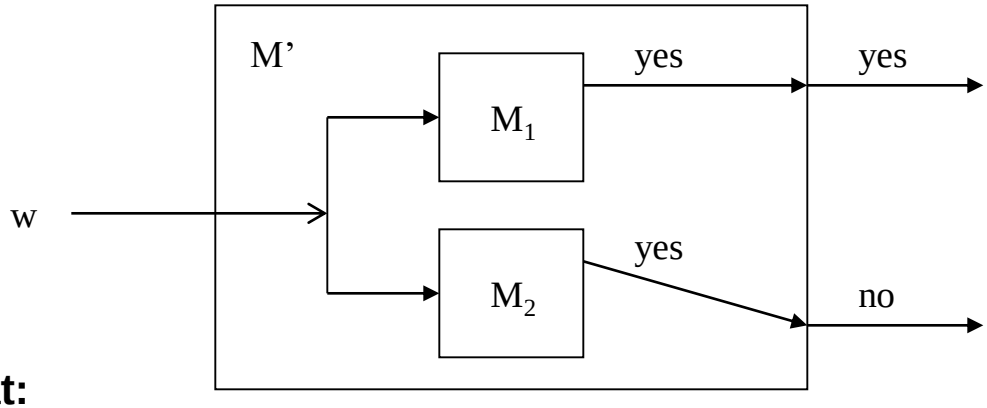
Question: What would happen if w is in $L(M_1)$ but not in $L(M_2)$?

Answer: You could get two outputs – one "yes" and one "no."

At least M_1 will halt and answer accept, M_2 may or may not halt.
As before, for the sake of convenience the "no" output will be ignored.

Theorem 4: If L and $\overline{L}$ are both recursively enumerable then L (and therefore $\overline{L}$) is recursive.

Proof: Let M_1 and M_2 be TMs such that L = L(M_1) and $\overline{L}$ = L(M_2). Construct M' as follows:



Note That:

L(M') = L

L(M') is a subset of L

L is a subset of L(M')

M' is TM for L

M' always halts since either M_1 or M_2 halts for any given string

M' shows that L is recursive

It follows from this that L (and therefore its' complement) is recursive.

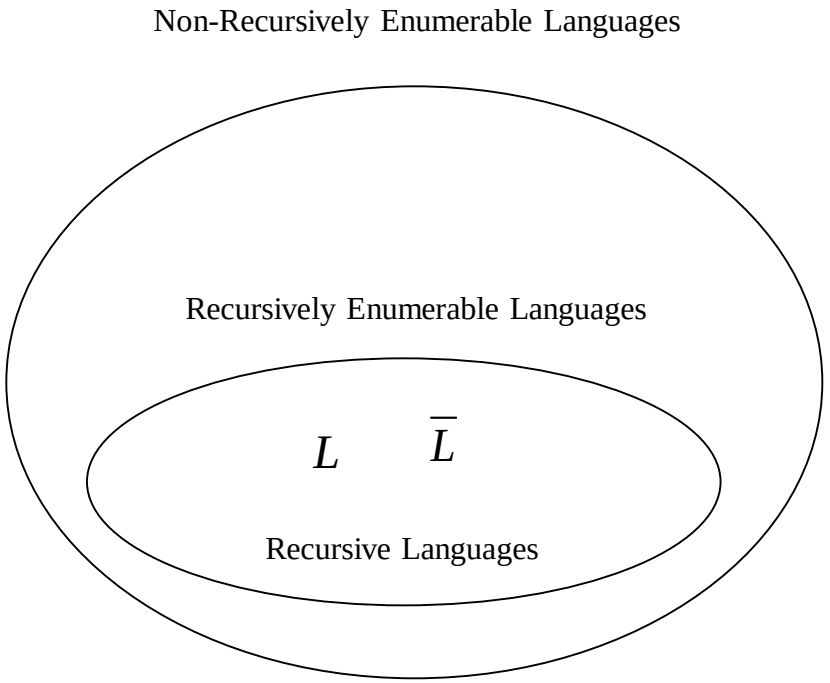
So, $\overline{L}$ is also recursive (we proved it before). •

Corollary of Thm 4: Let *L* be a subset of Σ*. Then one of the following must be true:

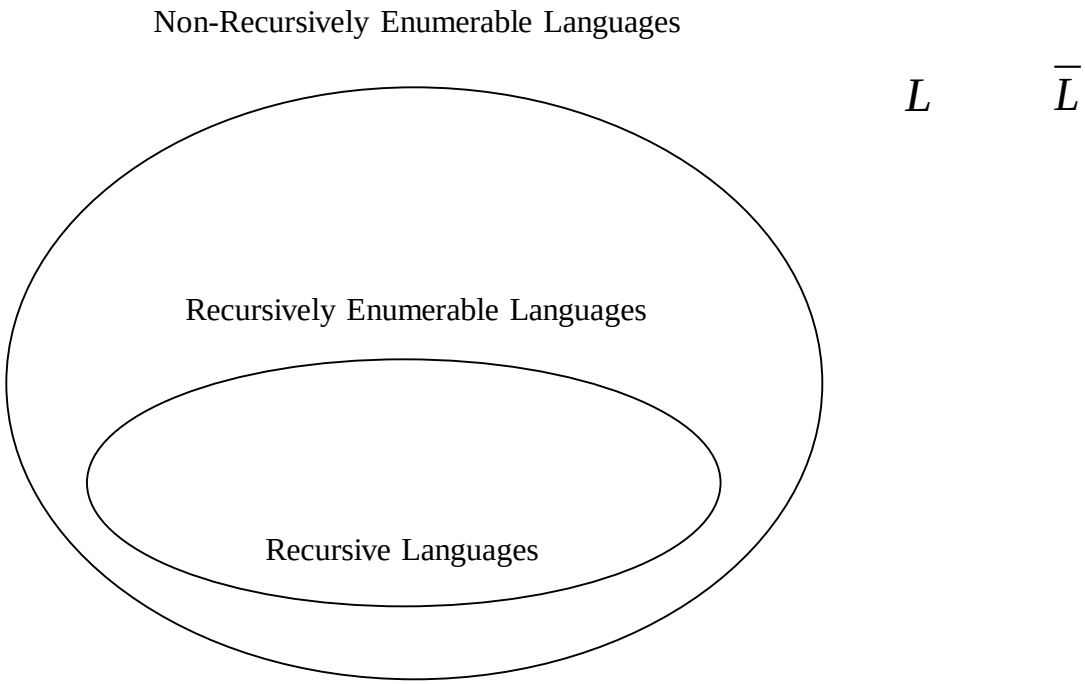
Both *L* and $\overline{L}$ are recursive.
One of *L* and $\overline{L}$ is recursively enumerable but not recursive, and the other is not recursively enumerable, or
Neither *L* nor $\overline{L}$ is recursively enumerable

In other words, it is impossible to have both L and $\overline{L}$ *r.e. but not recursive*

In terms of the hierarchy: (possibility #1)

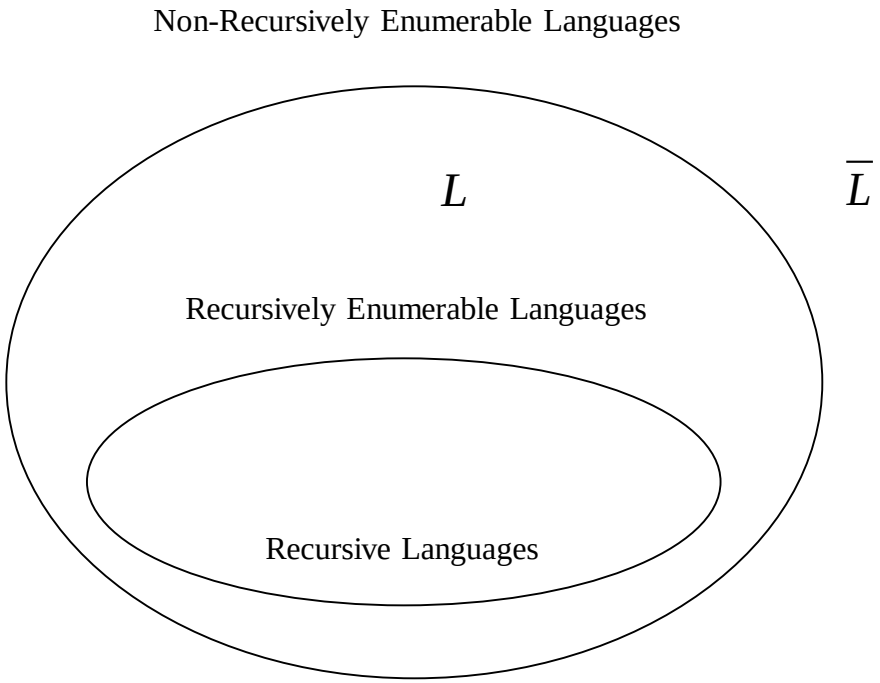


In terms of the hierarchy: (possibility #2)

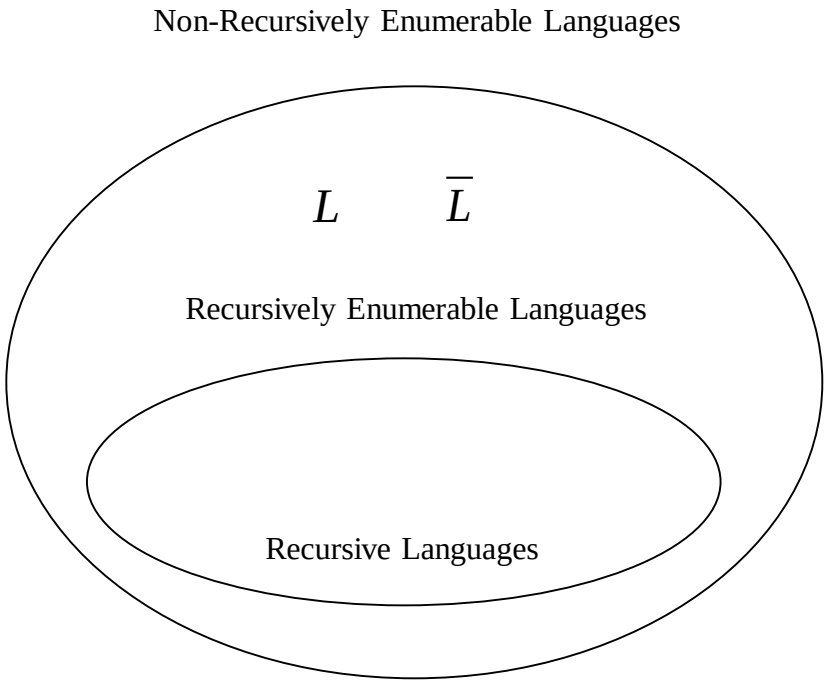


L $\overline{L}$

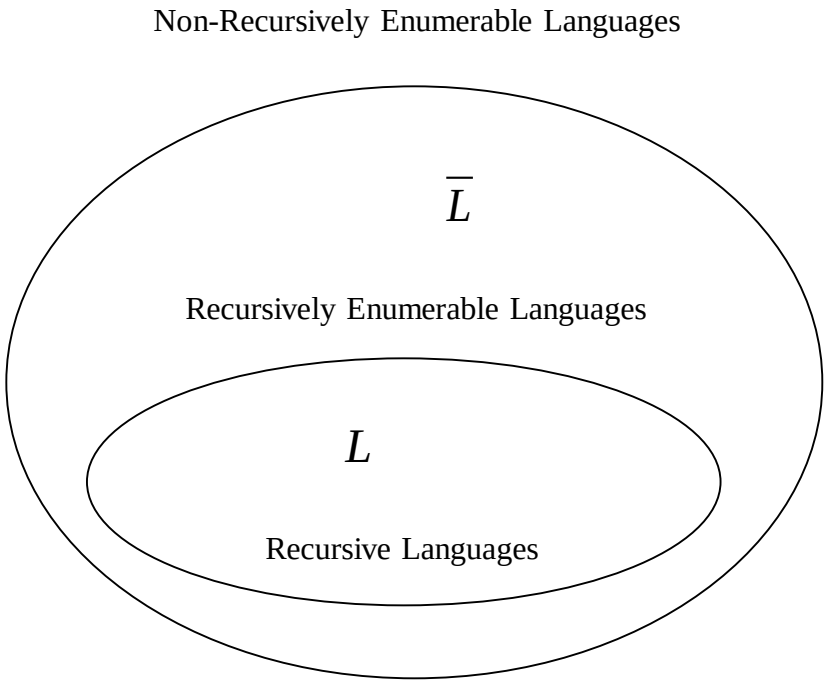
In terms of the hierarchy: (possibility #3)



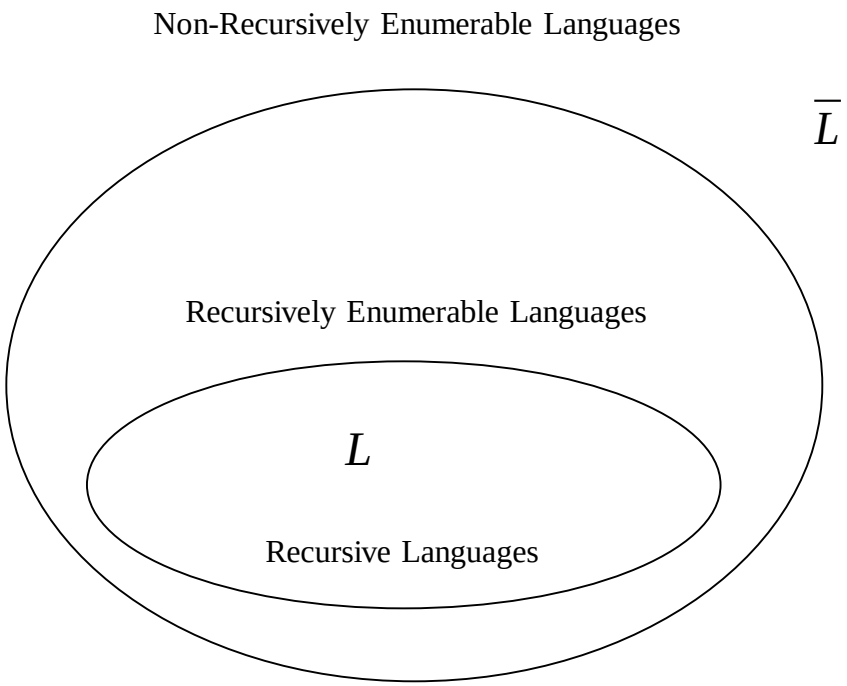
In terms of the hierarchy: (Impossibility #1)



In terms of the hierarchy: (Impossibility #2)



In terms of the hierarchy: (Impossibility #3)



Note: This gives/identifies three approaches to show that a language is not recursive.

- Show that the language's complement is not recursive, in one of the two ways:
  - Show that the language's complement is recursively enumerable but not recursive
  - Show that the language's complement is not even recursively enumerable

The Halting Problem - Background

Definition: A decision problem is a problem having a yes/no answer (that one presumably wants to solve with a computer). Typically, there is a list of parameters on which the problem is based.

Given a list of numbers, is that list sorted?
Given a number x, is x even?
Given a C program, does that C program contain any syntax errors?
Given a TM (or C program), does that TM contain an infinite loop?

From a practical perspective, many decision problems do not seem all that interesting. However, from a theoretical perspective they are for the following two reasons:

Decision problems are more convenient/easier to work with when proving complexity results.
Non-decision *counter-parts* can always be created & are typically at least as difficult to solve.

Notes:

The following terms and phrases are analogous:

Algorithm	-	A halting TM program
Decision Problem	-	A language *(will show shortly)*
(un)Decidable	-	**(non)Recursive**

Statement of the Halting Problem

Practical Form: (P1)

Input: Program P and input I.
Question: Does P terminate on input I?

Theoretical Form: (P2)

Input: Turing machine M with input alphabet Σ and string w in Σ*.
Question: Does M halt on w?

A Related Problem We Will Consider First: (P3)

Input: Turing machine M with input alphabet Σ and one final state, and string w in Σ*.
Question: Is w in L(M)?

Analogy:

Input: DFA M with input alphabet Σ and string w in Σ*.
Question: Is w in L(M)?
Is this problem (*regular language*) decidable? Yes! DFA always accepts or rejects.

Over-All Approach:

We will show that a language L_d is not recursively enumerable
From this it will follow that $\overline{L_d}$ is not recursive
Using this we will show that a language L_u is not recursive
From this it will follow that the Halting problem is undecidable.

As We Will See:

P3 will correspond to the language L_u
Proving P3 (un)decidable is equivalent to proving L_u (non)recursive

The Universal Language

Define the language L_u as follows:

L_u = {x | x is in {0, 1}* and x = <M,w> where *M* is a TM encoding and *w* is in L(M)}

Let *x* be in {0, 1}*. Then either:

1. *x* doesn't have a TM prefix, in which case *x* is **not** in L_u
2. *x* has a TM prefix, i.e., x = <M,w> and either:
   a) *w* is not in L(M), in which case *x* is **not** in L_u
   b) *w* is in L(M), in which case *x* is in L_u