THEORY OF COMPUTATION

Course Code: AITB03

B.Tech IV Semester

Regulation: IARE R-18

By

Dr. K Srinivasa Reddy

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous) DUNDIGAL,

HYDERABAD - 500 043.

Course Outcomes

CO 1	Understand the functionality of deterministic finite automata and Non-deterministic finite automata
CO 2	Apply the regular languages , regular expressions to construct finite automata
CO 3	Apply the context free grammars to construct derivation trees and the accept various strings
CO 4	Compare the functionality of push down automata with deterministic finite automata
CO 5	Apply the concept of Turing machines to solve the complex functions

UNIT– I
FINITE AUTOMATA

Course Learning Outcome

CLOs	Course Learning Outcome
CLO 1	Able to show the importance of alphabets, strings and languages to construct finite automata
CLO 2	Analyze the construction of finite automata for a given regular expressions
CLO 3	Apply the Chomsky normal form and Greibach normal forms to eliminate the Ambiguity in context free grammars
CLO 4	Able to construct the deterministic push down automata to accept the context free languages
CLO 5	Analyze the functionality of different types of Turing machines

Syllabus:

Fundamentals: Alphabet, strings, language, operations; Introduction to finite automata: The central concepts of automata theory, deterministic finite automata, nondeterministic finite automata, an application of finite automata, finite automata with epsilon transitions.

What is Automata Theory?

- Study of abstract computing devices, or "machines"
- Automaton = an abstract computing device
  - <u>Note:</u> A "device" need not even be a physical hardware!
- A fundamental question in computer science:
  - Find out what different models of machines can do and cannot do
  - The *theory of computation*
- Computability vs. Complexity

TOC :A Historical Perspective

1930s	• Alan Turing studies Turing machines • Decidability • Halting problem
1940-1950s	• "Finite automata" machines studied • Noam Chomsky proposes the "Chomsky Hierarchy" for formal languages
1969	Cook introduces "intractable" problems or "NP-Hard" problems
1970-	Modern computer science: compilers, computational & complexity theory evolve

5

An **alphabet** is a set of symbols:

$$\{0,1\}$$

Sentences are strings of symbols:

$$0,1,00,01,10,1,...$$

A **language** is a set of sentences:

$$L = \{000,0100,0010,..\}$$

A **grammar** is a finite list of rules defining a language.

S ⟶ 0A	B ⟶ 1B
A ⟶ 1A	B ⟶ 0F
A ⟶ 0B	F ⟶ ε

- <u>Languages</u>: "*A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols*"

- <u>Grammars</u>: "*A grammar can be regarded as a device that enumerates the sentences of a language*" - nothing more, nothing less

- *N. Chomsky, Information and Control, Vol 2, 1959*

Alphabet

An alphabet is a finite, non-empty set of symbols

- We use the symbol ∑ (sigma) to denote an alphabet
- Examples:
  - Binary: ∑ = {0,1}
  - All lower case letters: ∑ = {a,b,c,..z}
  - Alphanumeric: ∑ = {a-z, A-Z, 0-9}
  - DNA molecule letters: ∑ = {a,c,g,t}
  - ...

A string or word is a finite sequence of symbols chosen from ∑

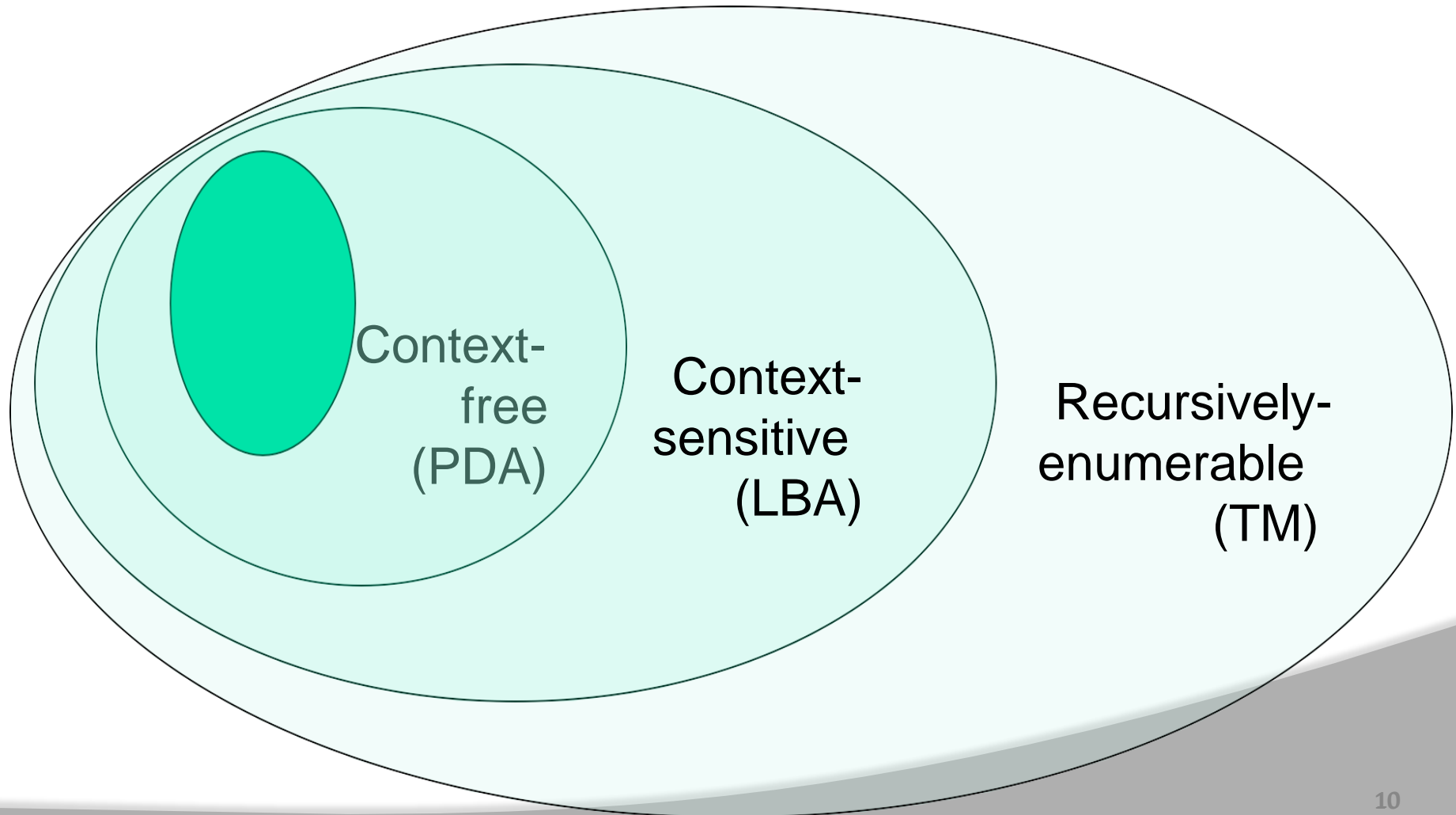- ***Empty string is ε (or "epsilon")***

- Length of a string *w,* denoted by "|*w*|", is equal to the *number of (non- ε) characters in the string*
  - *E.g., x = 010100*
  - *x = 01 ε 0 ε 1 ε 00 ε*

  |x| = 6
  |x| = ?

  - *xy = c*oncatentation of two strings *x* and *y*

The Chomsky Hierachy

- A containment hierarchy of classes of formal languages

Context-free (PDA)

Context-sensitive (LBA)

Recursively-enumerable (TM)

Let ∑ be an alphabet.

- $\sum^k$ = the set of all strings of length k

- $\sum^* = \sum^0 \cup \sum^1 \cup \sum^2 \cup \ \dots$

- $\sum^+ = \sum^1 \cup \sum^2 \cup \sum^3 \cup \ \dots$

L is a said to be a language over alphabet ∑, only if L ϵ ∑*

this is because ∑* is the set of all strings (of all possible length including 0) over the given alphabet ∑

Examples:

1. Let L be *the* language of all strings consisting of *n* 0's followed by *n* 1's:
$$L = \{\epsilon, 01, 0011, 000111,...\}$$

2. Let L be *the* language of all strings of with equal number of 0's and 1's:L = $\{\epsilon, 01, 10, 0011, 1100, 0101, 1010, 1001,...\}$ Canonical ordering of strings in the language

Definition: Ø denotes the Empty language

- Let L = $\{\varepsilon\}$; Is L=Ø?   NO

$W = a_1 a_2 \epsilon\ a_n$

abba

$v = b_1 b_2 \epsilon b_m$

bbbaaa

Concatenation

$$wv = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m$$

abbabbbaaa

$w = a_1 a_2 \cdots a_n$ $ababaaabbb$

Reverse

$w^R = a_n \cdots a_2 a_1$ $bbbaaababa$

String Length

$$w = a_1 a_2 \cdots a_n$$

- Length: $|w| = n$

- Examples:

$$|abba| = 4$$

$$|aa| = 2$$

$$|a| = 1$$

- For any letter: $|a| = 1$

$$|wa| = |w| + 1$$

- For any string   : $|abba| = |abb| + 1$

$$= |ab| + 1 + 1$$

- Example:

$$= |a| + 1 + 1 + 1$$

$$= 1 + 1 + 1 + 1$$

$$= 4^{16}$$

Length of Concatenation

$$|uv| = |u| + |v|$$

$$u = aab, \quad |u| = 3$$

$$v = abaab, \quad |v| = 5$$

- Example:

$$|uv| = |aababaab| = 8$$

$$|uv| = |u| + |v| = 3 + 5 = 8$$

Deterministic Finite State Automata (DFA)

| 0 | 1 | 1 | 0 | 0 | | | | |

...
....
.
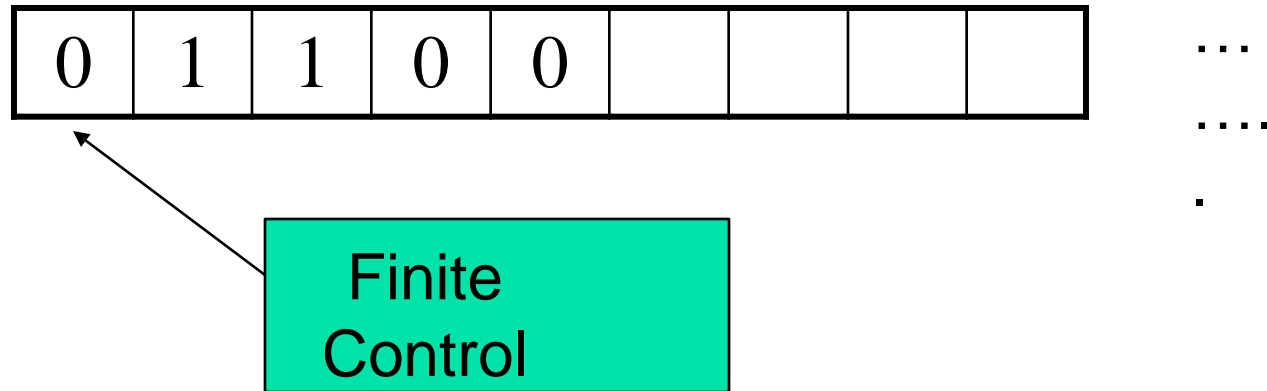
Finite Control

- 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 <u>transition diagram</u> or <u>table</u>

Example #1:

$$q_0 \quad q_0 \quad q_1 \quad q_0 \quad q_0 \quad q_0$$
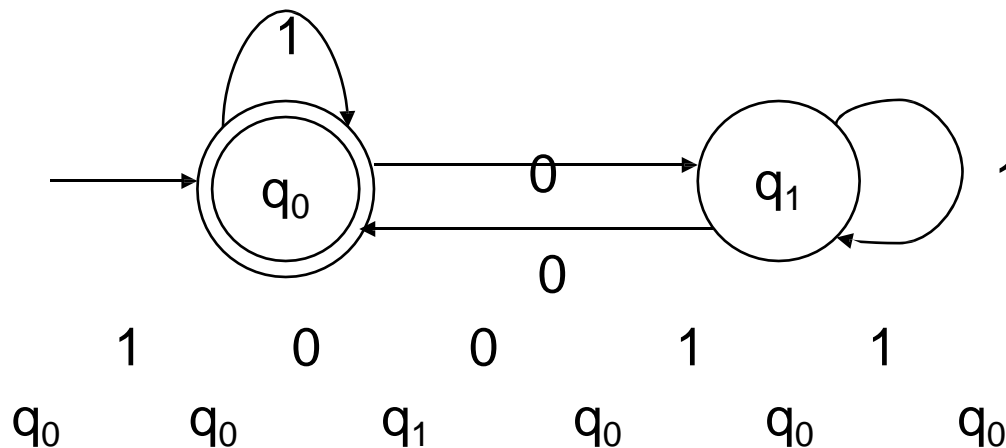
- One state is final/accepting, all others are rejecting.
- The above DFA accepts those strings that contain an even number of 0's, including the *null* string, over *Sigma* = {0,1}

    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!

M1   M2   ….   M-inf

L
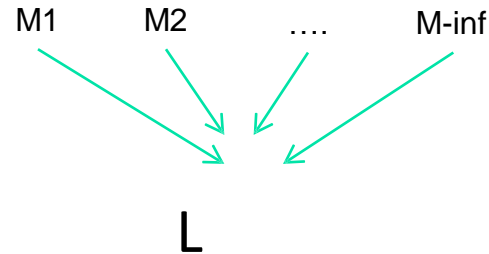
- *Id's of the characters or states are irrelevant,*
    you can call them by any names!
    Sigma = {0, 1} ≡ {a, b}
    States = {q0, q1} ≡ {u, v}, as long as they have
        identical (isomorphic) transition table
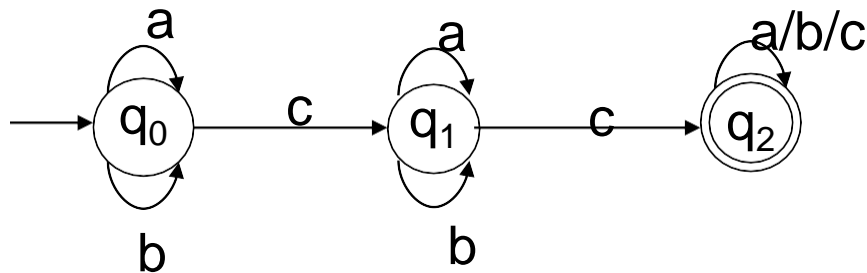
Inductive Proof (sketch): that the machine correctly accepts strings with at least two c's

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

bol p =a, b or c

- A DFA is a five-tuple:

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

Q   A <u>finite</u> set of states Σ

      A <u>finite</u> 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

δ: (Q x Σ) –> Q

$\delta(q,s) = q'$

δ is defined for any q in Q and s in Σ, and is equal to some state q' in Q, could be q'=q

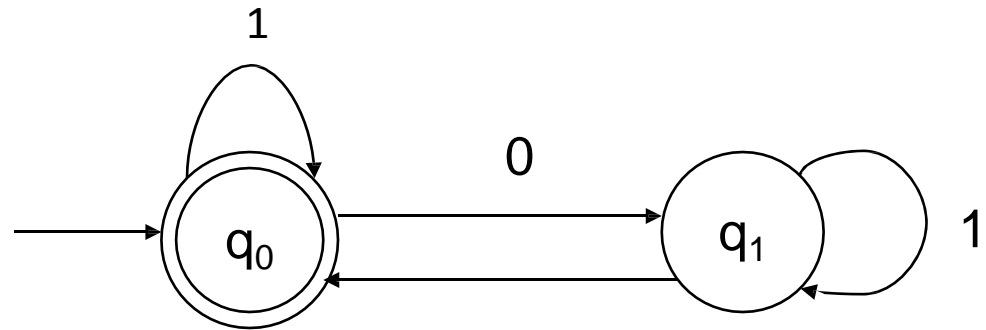Intuitively, $\delta(q,s)$ is the state entered by M after reading symbol s while in state q.

- **Revisit example #1:**

  $Q = \{q_0, q_1\}$

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

  Start state is q

  $F = \{q_0\}$

	0	1
q_0	q_1	q_0
q_1	q_0	q_1

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

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

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
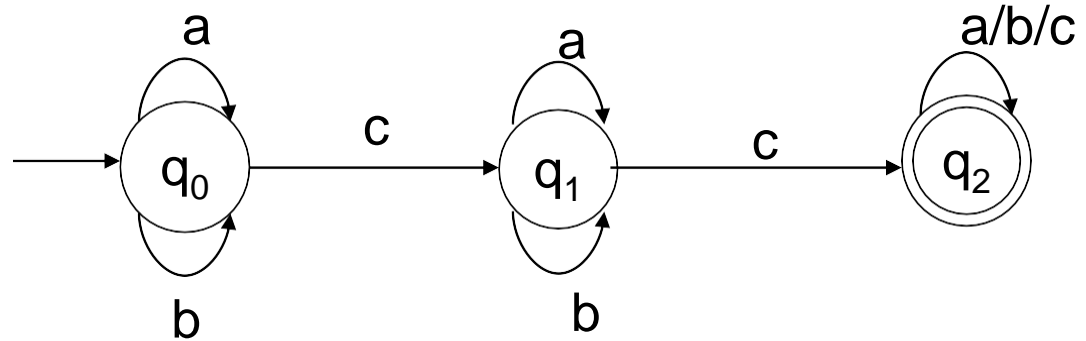
- Since δ is a function, at each step M has exactly one option.
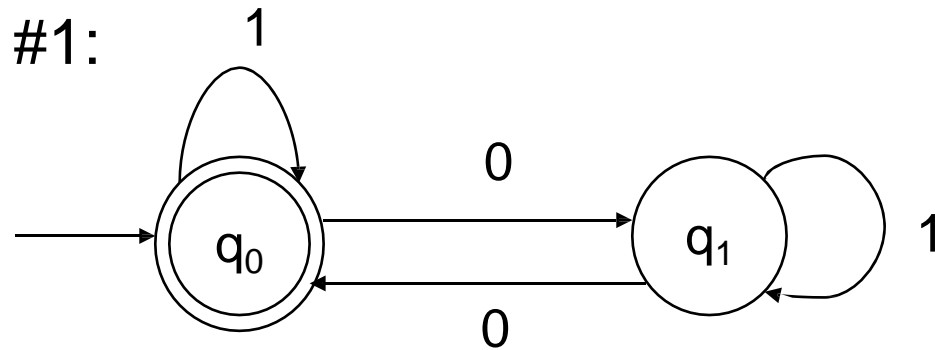- It follows that for a given string, there is exactly one computation.

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

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

Formally:

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

- **Recall Example #1:**

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

- Formally:

$\delta^\wedge(q_0, 011)$

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

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

  - Note that:

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

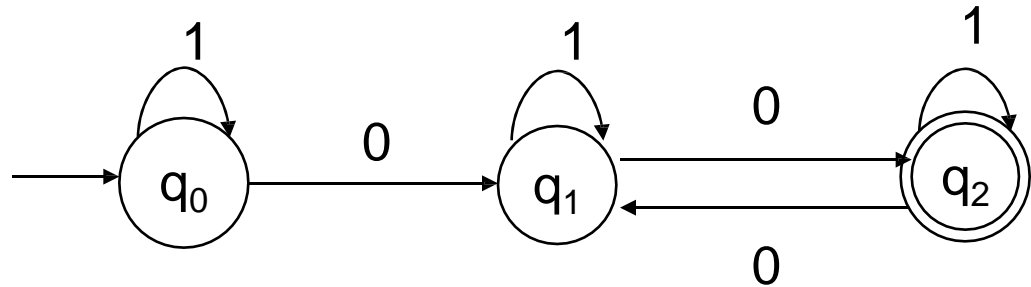 $$= \delta(q, a) \quad \text{by definition of } \delta^\wedge, \text{ rule \#1}$$

  - Therefore:

    $$\delta^\wedge(q, a_1 a_2 ... a_n) = \delta(\delta(...\delta(\delta(q, a1), a2)...), a_n)$$

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

    $$\delta^\wedge(q, a_1 a_2 ... a_n) = \delta(q, a_1 a_2 ... a_n)$$

- Example #3:

$$1 \qquad 1 \qquad 1$$
$$q_0 \xrightarrow{0} q_1 \xrightarrow[0]{0} q_2$$

- 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}

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:

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

- 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:* $L_2 = \{x \mid x = 0^n 1^n$ 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 (zero or more) a's, b's and c's such

  that x does *not* contain the substring *aa*}
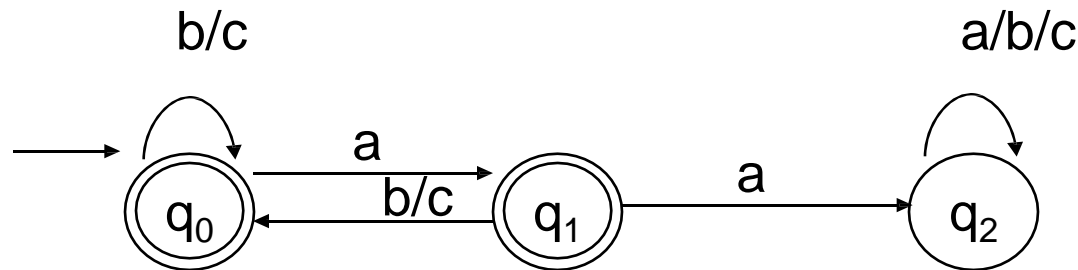
Logic:

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

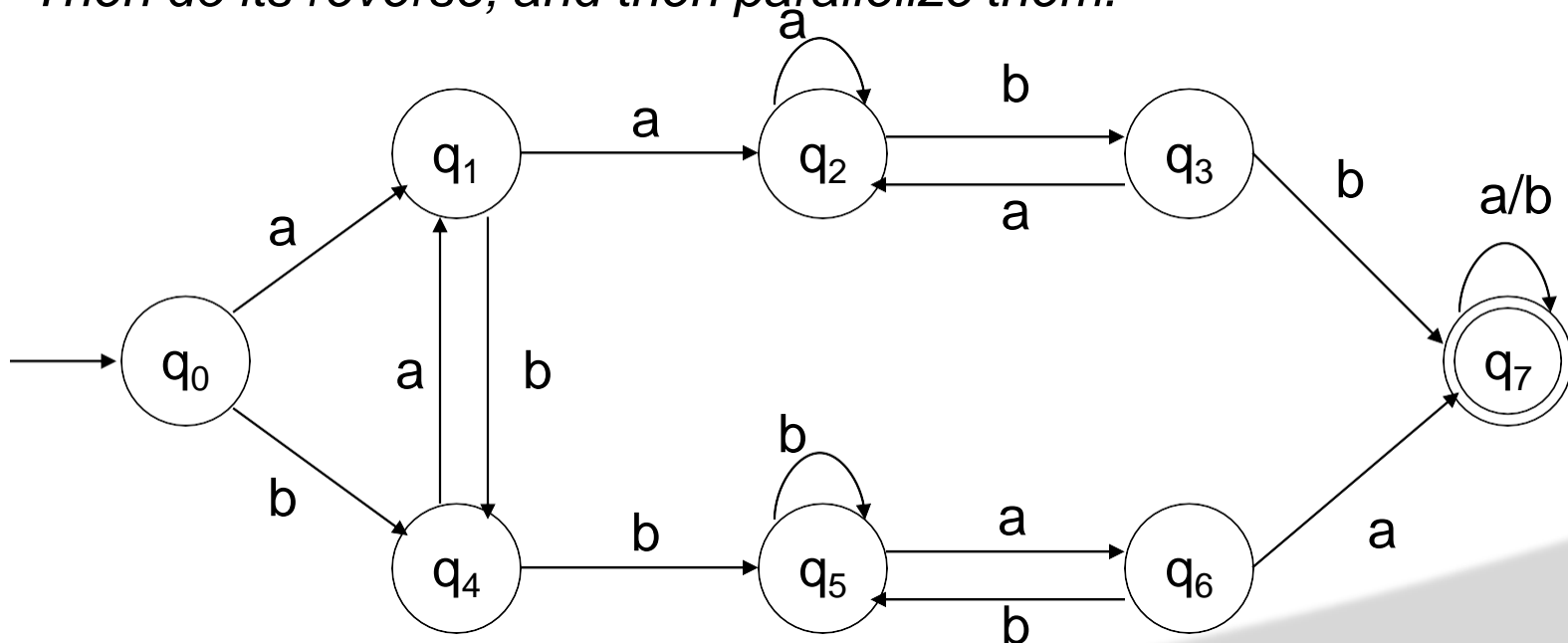*When second 'a' appears after the "ready" state: go to reject state*

DFA

- 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*}

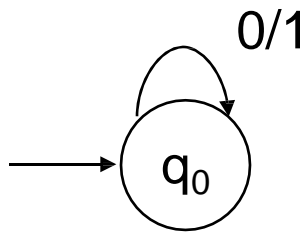*First do, for a language where 'aa' comes before 'bb'*
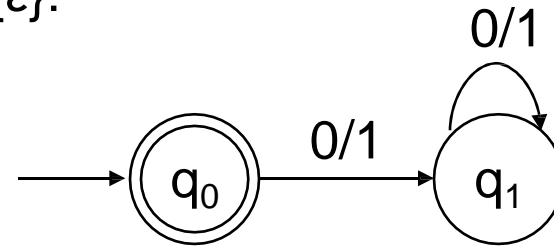
Then do its reverse; and then parallelize them.

Remember, you may have multiple "final" states, but only one "start" state

- Let $\Sigma = \{0, 1\}$. Give DFAs for $\{\}$, $\{\varepsilon\}$, Σ^*, and Σ^+.
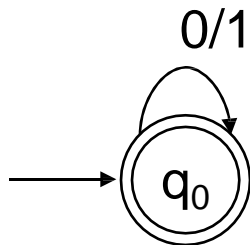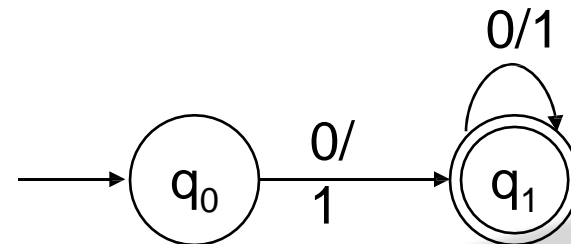
For $\{\}$:
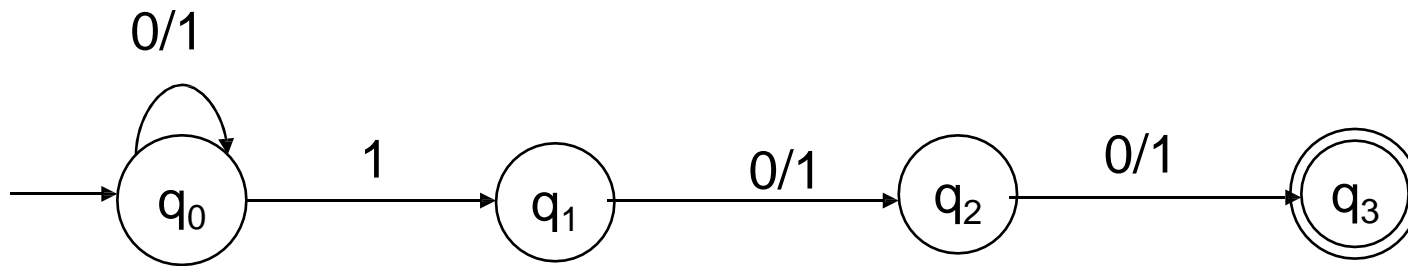

For $\{\varepsilon\}$:

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

- An NFA is a five-tuple:

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

    $Q =$ A <u>finite</u> set of
    states

    $\Sigma =$ A <u>finite</u> 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

    $\delta =$ A transition function, which is a total function
    from $Q \times \Sigma$ to 2^Q

δ: (Q x Σ) –> **2^Q**
of δ(q,s) transition

:2^Q is the power set of Q, the set of *all subsets*
:The set of all states p such that there is a
labeled s from q to p
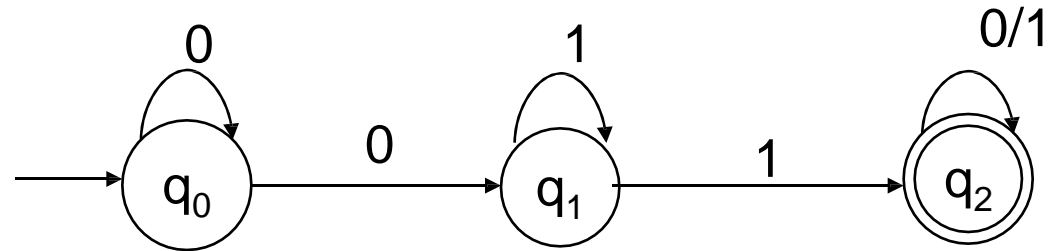
δ(q,s) is a function from Q x S to 2^Q (but
not only to Q)

- 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	$\{q_2\}$	$\{q_2\}$

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

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

Start state is q_0

$F = \{q_2, q_4\}$

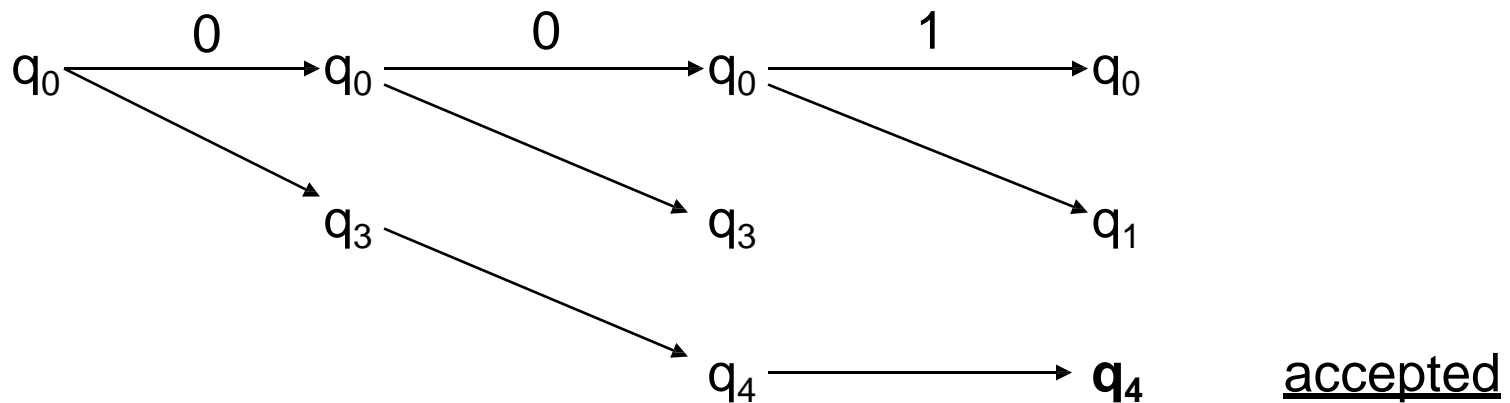| δ:	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\{\}$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\{\}$
q_4	$\{q_4\}$	$\{q_4\}$
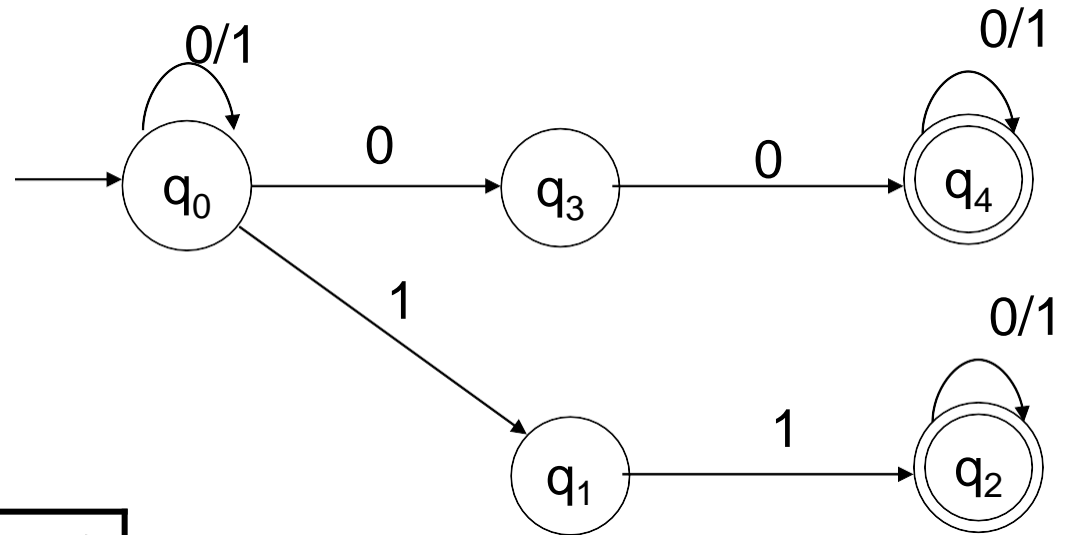
- Notes:
  - δ(q,s) may not be defined for some *q* and *s* (*what does that mean*?)
  - δ(q,s) may map to multiple *q*'s
  - A string is said to be accepted *if there exists* a path from q_0 to some state in *F*
  - A string is rejected *if there exist NO path* to any state in *F*
  - The language accepted by an NFA is the set of all accepted strings

- Question: How does an NFA find the correct/accepting path for
  a given string?
  - NFAs are a non-intuitive computing model
  - You may use *backtracking* to find if there exists a path to a final state (following slide)
- Why NFA?
  - We are *primarily* interested in NFAs as language defining capability, i.e., do NFAs accept languages that DFAs do not?
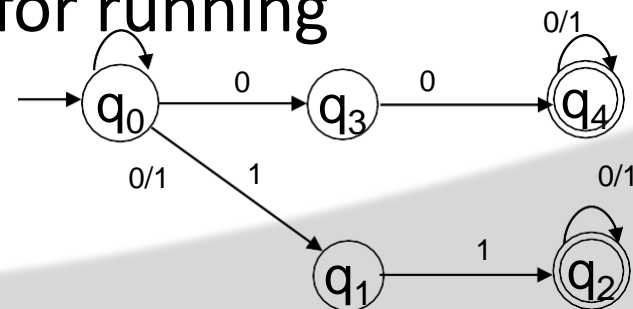 - Other secondary questions include practical ones such as whether

- Determining if a given NFA (example #2) accepts a given string

  (1)   can be done algorithmically:

$q_0 \xrightarrow{\ 0\ } q_0 \xrightarrow{\ 0\ } q_0 \xrightarrow{\ 1\ } q_0$

$q_3 \qquad\qquad q_3 \qquad\qquad q_1$

$q_4 \longrightarrow \mathbf{q_4} \qquad \underline{accepted}$

- Each level will have at most *n* states:

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

- **Question: Why non-determinism is useful?**
  - Non-determinism   =   Backtracking
  - Compressed information
  - Non-determinism hides backtracking
  - Programming languages, e.g., Prolog, hides backtracking => Easy to program at a higher level: *what we want to do, rather than how to do it*
  - Useful in algorithm complexity study

  - Is NDA more "powerful" than DFA, i.e., accepts type of languages that any DFA cannot?

- Let Σ = {a, b, c}. Give an NFA M that accepts:
  L = {x | x is in Σ^* and x contains ab}
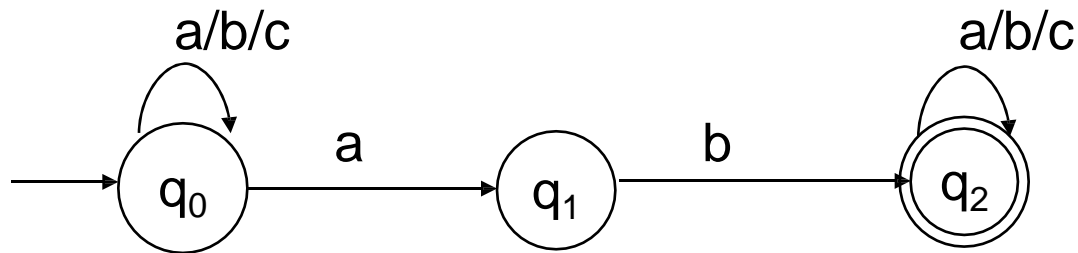
Is L a subset of L(M)?Or, does M accepts all string in L? Is

L(M) a subset of L? Or, does M rejects all strings not in L?

- Is an NFA necessary? Can you draw a DFA for this L?
- Designing NFAs is not as trivial as it seems: easy to create bug accepting string outside language

- 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}
  a/b
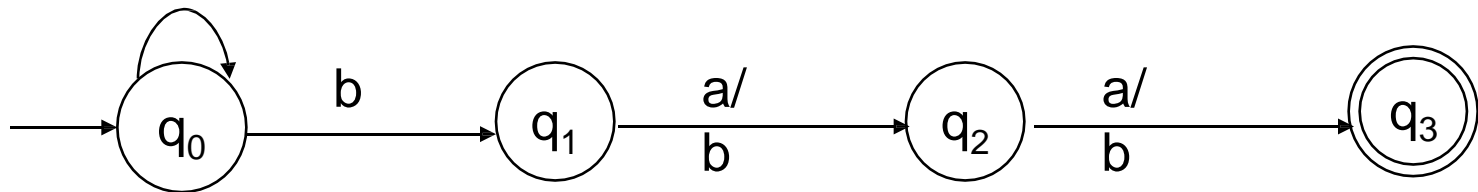
Is L a subset of L(M)? Is L(M) subset of L?

Give an equivalent DFA as an exercise.

- 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.
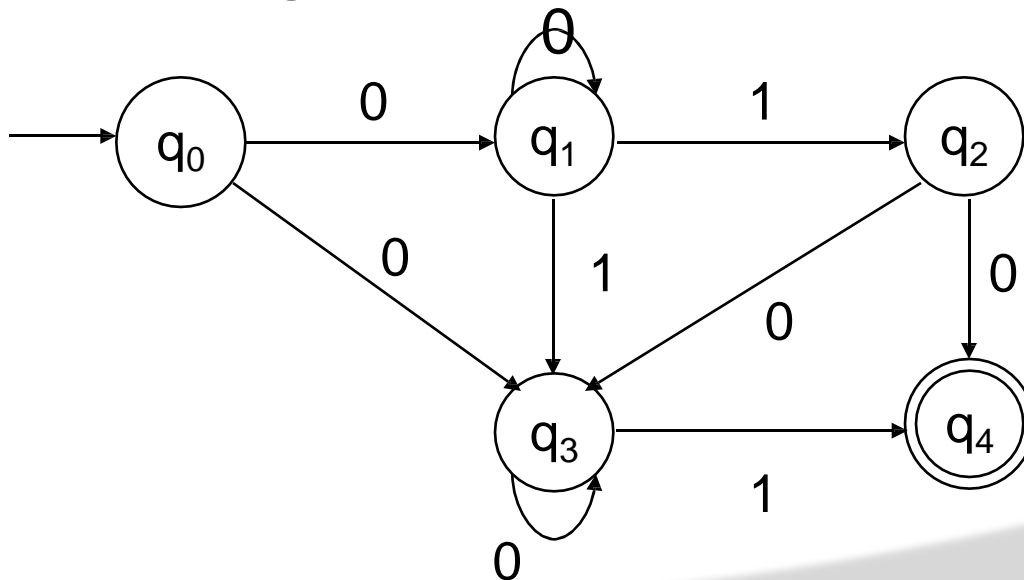
- **Step #1:**
Given $\delta: (Q \times \Sigma) \rightarrow 2^Q$ define $\delta^{\#}: (2^Q \times \Sigma) \rightarrow 2^Q$ as follows:

1) $\delta^{\#}(R, a) = \bigcup_{q \in R}$ $\delta(q, a)$ for all subsets R of Q, and symbols a in Σ

- **Note that:**

$$\delta^{\#}(\{p\}, a) = \bigcup_{q \in \{p\}} \delta(q, a)$$
$$= \delta(p, a)$$

by definition of $\delta^{\#}$, rule #1 above

- Hence, we can use δ for $\delta^{\#}$

$\delta(\{q_0, q_2\}, 0)$
   previously
$\delta(\{q_0, q_1, q_2\}, 0)$

These now make sense, but they did not.

- **Example:**

$$\delta(\{q_0, q_2\}, 0) = \delta(q_0, 0) \cup \delta(q_2, 0)$$
$$= \{q_1, q_3\} \cup \{q_3, q_4\}$$
$$= \{q_1, q_3, q_4\}$$

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

- Step #2:

  Given $\delta: (2^Q \times \Sigma) \rightarrow 2^Q$ define $\delta\hat{}: (2^Q \times \Sigma^*) \rightarrow 2^Q$ as follows:

  $\delta\hat{}(R,w)$ – The set of states M could be in after processing string w, having started from any state in R.

  Formally:

2) $\delta\hat{}(R, \varepsilon) = R$	for any subset R of Q
3) $\delta\hat{}(R,wa) = \delta (\delta\hat{}(R,w), a)$	for any w in Σ^*, a in Σ, and subset R of Q

- Note that:

  $\delta\hat{}(R, a) = \delta(\delta\hat{}(R, \varepsilon), a)$

  $\quad = \delta(R, a)$ by definition of $\delta\hat{}$, rule #3 above by definition of $\delta\hat{}$, rule #2 above

- Hence, we can use δ for $\delta\hat{}$

  $\delta(\{q_0, q_2\}, 0110)$
    previously

  $\delta(\{q_0, q_1, q_2\}, 101101)$ These now make sense, but

  they did not.
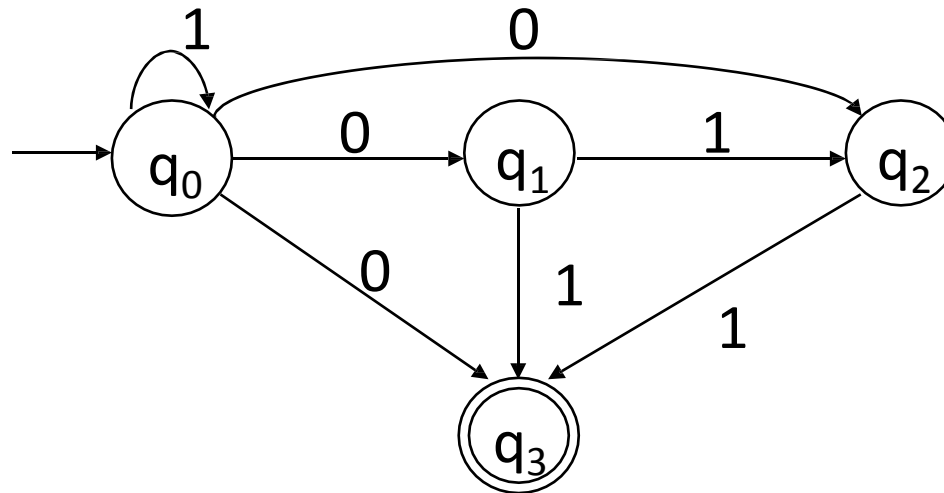
- Example:

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

Formall
y:

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

Is 10 accepted? Yes!

- **Example:**
  What is $\delta(\{q_0, q_1\}, 1)$?
  $\delta(\{q_0, q_1\}, 1) = \delta(\{q_0\}, 1) \cup \delta(\{q_1\}, 1)$
  $\qquad\qquad = \{q_0\} \cup \{q_2, q_3\}$
  $\qquad\qquad = \{q_0, q_2, q_3\}$
  What is $\delta(\{q_0, q_2\}, 10)$?

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

- **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.

- Let M = (Q, Σ, δ, q_0, F) be an NFA and let w be in $Σ^*$. Then w is *accepted* by M iff δ({q_0}, w) contains at least one state in F.
- Let M = (Q, Σ, δ, q_0, F) be an NFA. Then the *language accepted* by M is the set:
  L(M) = {w | w is in $Σ^*$ and δ({q_0}, w) contains at least one state in F}
- Another equivalent definition:
  L(M) = {w | w is in $Σ^*$ and w is accepted by M}

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).
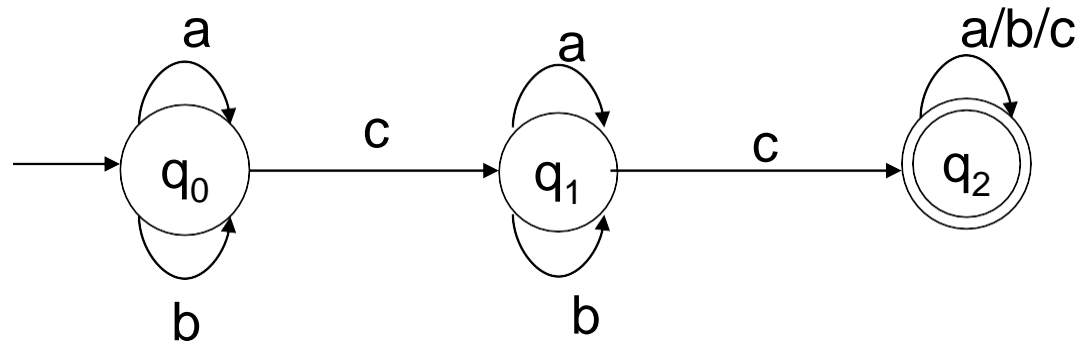- It follows that NFAs accept all regular languages.
- But do NFAs accept more?

- **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

- **An Equivalent NFA:**

$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{a, b, c\}$
Start state is q_0
$F = \{q_2\}$

δ:
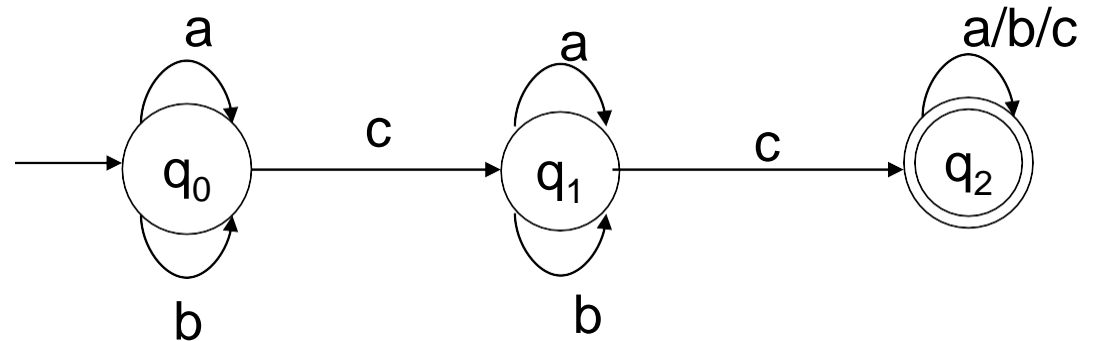
	a	b	c
q_0	$\{q_0\}$	$\{q_0\}$	$\{q_1\}$
q_1	$\{q_1\}$	$\{q_1\}$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$

- **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, Σ, δ, q_0, F).
  Define a DFA M' = (Q', Σ, δ', q_0 , F') as:

Q' = 2^Q
   = {Q_0, Q_1,...,}

Each state in M' corresponds to a subset of states from M

where Q_u = [q_{i0}, q_{i1},...q_{ij}]

F' = {Q_u | Q_u contains at least one state in F}
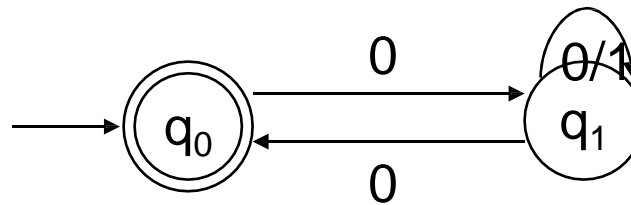
q'_0 = [q_0]

δ'(Q_u, a) = Q_v iff δ(Q_u, a) = Q_v

- **Example: empty string or start and end with 0**

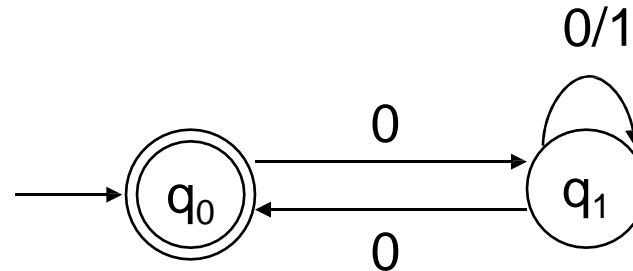$Q = \{q_0, q_1\}$

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

Start state is q_0

$F = \{q_0\}$

δ:

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

- Example of creating a DFA out of an NFA (as per the constructive proof):

$$0/1$$

-->q_0	$\{q_1\}$	$\{\}$
q_1	$\{q_0, q_1\}$	$\{q_1\}$

δ for DFA:

	0	1
->q_0	$\{q_1\}$ write as $[q_1]$	$\{\}$ write as $[\,]$
$[q_1]$	$[q_1]$	$[\,]$
$[\,]$		

■ Example of creating a DFA out of an NFA (as per the constructive proof):

0/1

0

q_0

q_1

0

$\{q_1\}$	$\{\}$
$\{q_0, q_1\}$	$\{q_1\}$

δ:
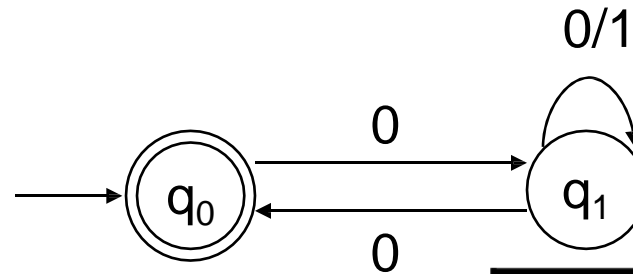
	0	1
->q_0	$\{q_1\}$ write as $[q_1]$	$\{\}$
$[q_1]$	$\{q_0,q_1\}$ write as $[q_{01}]$	$\{q_1\}$
[ ]		
$[q_{01}]$		

- **Example of creating a DFA out of an NFA (as per the constructive proof):**

0/1

0

q_0 q_1

0

$\{q_1\}$	$\{\}$
$\{q_0, q_1\}$	$\{q_1\}$

δ: 0   1

->q_0 $\{q_1\}$ $\{\}$
          write as
          $[q_1]$

$[q_1]$ $\{q_0,q_1\}$ $\{q_1\}$
          write as
          $[q_{01}]$

$[\ ]$ $[\ ]$ $[\ ]$

$[q_{01}]$

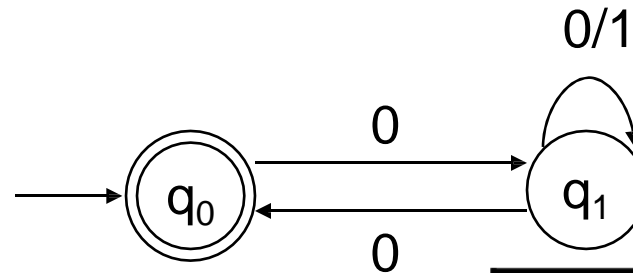- Example of creating a DFA out of an NFA (as per the constructive proof):
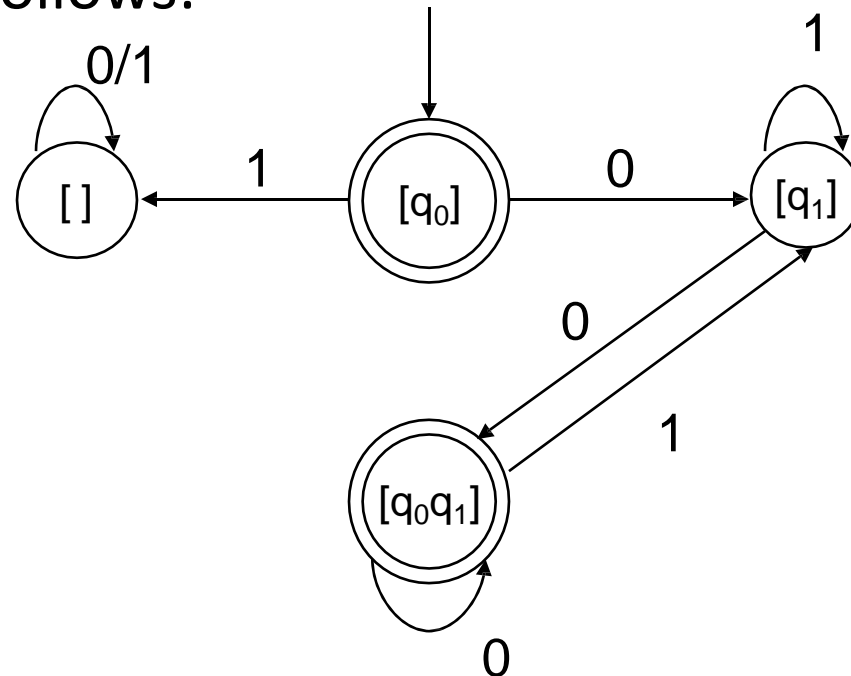

δ:

	0	1
->q_0	$\{q_1\}$ write as $[q_1]$	$\{\}$
$[q_1]$	$\{q_0,q_1\}$ write as $[q_{01}]$	$\{q_1\}$
$[\ ]$	$[\ ]$	$[\ ]$
$[q_{01}]$	$[q_{01}]$	$[q_1]$

■ Construct DFA M' as follows:

$\delta(\{q_0\}, 0) = \{q_1\}$	=>	$\delta'([q_0], 0) = [q_1]$
$\delta(\{q_0\}, 1) = \{\}$	=>	$\delta'([q_0], 1) = [\ ]$
$\delta(\{q_1\}, 0) = \{q_0, q_1\}$	=>	$\delta'([q_1], 0) = [q_0 q_1]$
$\delta(\{q_1\}, 1) = \{q_1\}$	=>	$\delta'([q_1], 1) = [q_1]$
$\delta(\{q_0, q_1\}, 0) = \{q_0, q_1\}$	=>	$\delta'([q_0 q_1], 0) = [q_0 q_1]$
$\delta(\{q_0, q_1\}, 1) = \{q_1\}$	=>	$\delta'([q_0 q_1], 1) = [q_1]$
$\delta(\{\}, 0) = \{\}$	=>	$\delta'([\ ], 0) = [\ ]$
$\delta(\{\}, 1) = \{\}$	=>	$\delta'([\ ], 1) = [\ ]$

- **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.

- Note: Suppose R = {}

$$\begin{aligned}
\delta(R, 0) \quad &= \delta(\delta(R, \varepsilon), 0) \\
&= \delta(R, 0) \\
&= \bigcup_{q \in R} \delta(q, 0) \\
&= \{\ \} \qquad\qquad \text{Since R = \{\}}
\end{aligned}$$

- Exercise - Convert the following NFA to a DFA:
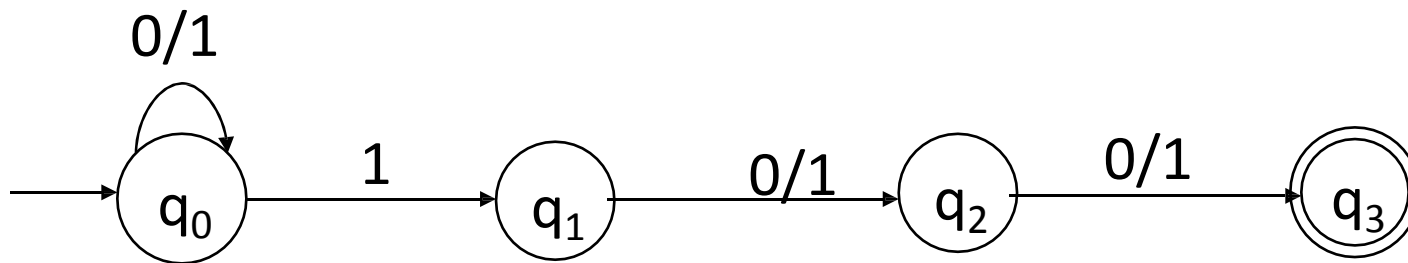
$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{0, 1\}$
Start state is q_0
$F = \{q_0\}$

δ:

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

- Problem: Third symbol from last is 1

Now, can you convert this NFA to a DFA?

Finite Automata

- Some Applications
  - Software for designing and checking the behavior of digital circuits
  - Lexical analyzer of a typical compiler
  - Software for scanning large bodies of text (e.g., web pages) for pattern finding
  - Software for verifying systems of all types that have a finite number of states (e.g., stock market transaction, communication/network protocol)

- An NFA-ε is a five-tuple:

  M = (Q, Σ, δ, q_0, F)

  Q = A <u>finite</u> set of

  states

  Σ= A <u>finite</u> 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 $_2Q$

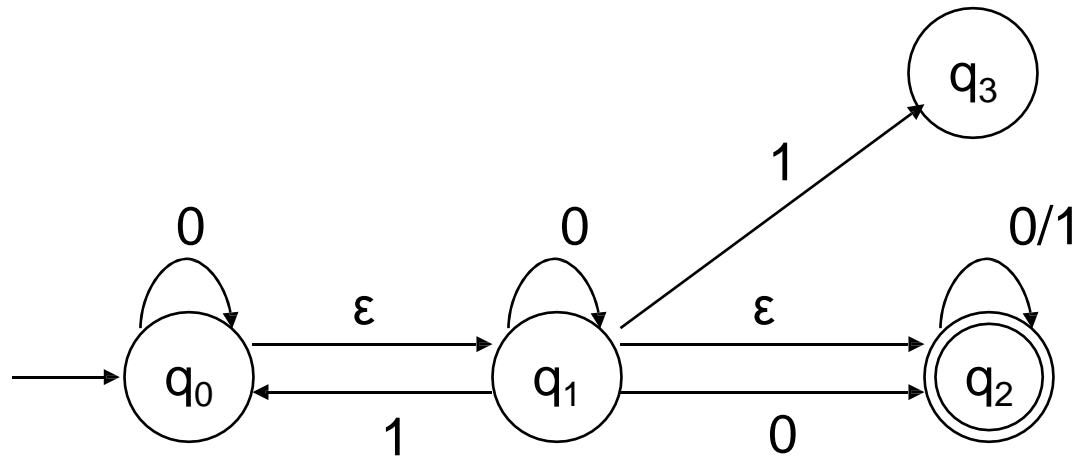   δ: (Q x (**Σ U {ε}**)) –> 2^Q -The set of all states p such that there
  is a   δ(q,s)

                              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 $w = w_1 w_2 \ldots w_n$ is processed

    as $w = \varepsilon^* w_1 \varepsilon^* w_2 \varepsilon^* \ldots \varepsilon^* w_n \varepsilon^*$

- 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.

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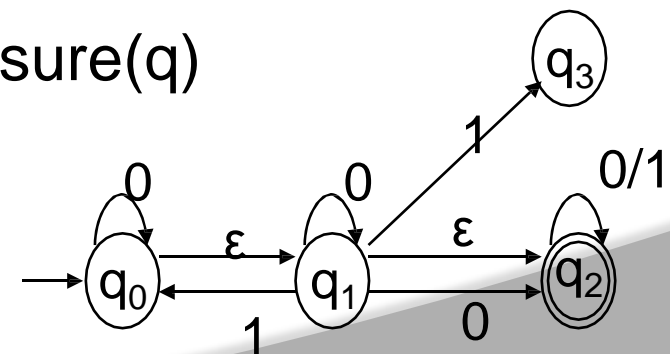 ε-closure(q_1) = {q_1, q_2}   ε-closure(q_3) = {q_3}

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

$$\varepsilon\text{-closure}(P) = \bigcup_{q \in P} \varepsilon\text{-closure}(q)$$
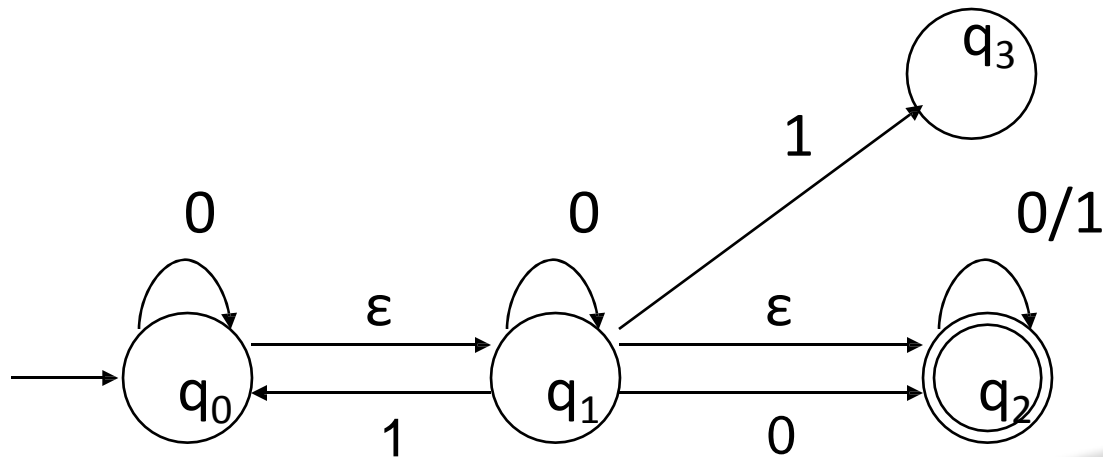
- Examples:

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

75

- What we currently have: $\delta : (Q \times (\Sigma \cup \{\varepsilon\})) \rightarrow 2^Q$

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

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

- Step #1:
  Given $\delta: (Q \times (\Sigma \cup \{\varepsilon\})) \rightarrow 2^Q$ define $\delta^{\#}: (2^Q \times (\Sigma \cup \{\varepsilon\})) \rightarrow 2^Q$ as follows:

  1) $\delta^{\#}(R, a) = \bigcup_{q \in R} \delta(q, a)$ for all subsets R of Q, and symbols a in $\Sigma \cup \{\varepsilon\}$

- Note that:

  $$\delta^{\#}(\{p\}, a) = \bigcup_{q \in \{p\}} \delta(q, a) \text{ by definition of } \delta^{\#}, \text{ rule \#1 above}$$

  $$= \delta(p, a)$$

- Hence, we can use δ for $\delta^{\#}$

  $\delta(\{q_0, q_2\}, 0)$ These now make sense, but previously

  $\delta(\{q_0, q_1, q_2\}, 0)$ they did not.

- Examples:

What is $\delta(\{q_0, q_1, q_2\}, 1)$?

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

$= \{\ \} \cup \{q_0, q_3\} \cup \{q_2\}$

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

What is $\delta(\{q_0, q_1\}, 0)$?

$\delta(\{q_0, q_1\}, 0) = \delta(q_0, 0) \cup \delta(q_1, 0)$

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

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

- Step #2:
  Given $\delta: (2^Q \times (\Sigma \cup \{\varepsilon\})) \rightarrow 2^Q$ define $\delta^\wedge: (2^Q \times \Sigma^*) \rightarrow 2^Q$ as
  follows:
  $\delta^\wedge(R,w)$ – The set of states M could be in after processing string w, having starting from any state in R.
  Formally:
  2) $\delta^\wedge(R, \varepsilon) = \varepsilon\text{-closure}(R)$ - for any subset R of Q
  3) $\delta^\wedge(R,wa) = \varepsilon\text{-closure}(\delta(\delta^\wedge(R,w), a))$ - for any w in Σ^*, a in Σ, and subset R of Q

- Can we use δ for $\delta^\wedge$?

- Consider the following example:

$\delta(\{q_0\}, 0) = \{q_0\}$ By rule #3

$\delta^\wedge(\{q_0\}, 0) = \varepsilon\text{-closure}(\delta(\delta^\wedge(\{q_0\}, \varepsilon), 0))$ By rule #2

$\qquad\qquad = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(\{q_0\}), 0))$ By ε-closure

$\qquad\qquad = \varepsilon\text{-closure}(\delta(\{q_0, q_1, q_2\}, 0))$ By rule #1

$\qquad\qquad = \varepsilon\text{-closure}(\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0))$

$\qquad\qquad = \varepsilon\text{-closure}(\{q_0\} \cup \{q_1, q_2\} \cup \{q_2\})$

$\qquad\qquad = \varepsilon\text{-closure}(\{q_0, q_1, q_2\})$

$\qquad\qquad = \varepsilon\text{-closure}(\{q_0\}) \cup \varepsilon\text{-closure}(\{q_1\}) \cup \varepsilon\text{-closure}(\{q_2\})$

$\qquad\qquad = \{q_0, q_1, q_2\} \cup \{q_1, q_2\} \cup \{q_2\}$

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

$\delta(q_0, 0)$ So what is the difference? - Processes 0 as a single symbol, without ε transitions.

$\delta^\wedge(q_0, 0)$ possible. - Processes 0 using as many ε transitions as are

- **Example :**

$\delta^\wedge(\{q_0\}, 01) = \varepsilon\text{-closure}(\delta(\delta^\wedge(\{q_0\}, 0), 1))$ #3

$\qquad\qquad = \varepsilon\text{-closure}(\delta(\{q_0, q_1, q_2\}), 1)$

$\qquad\qquad = \varepsilon\text{-closure}(\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1))$

#1 $\qquad = \varepsilon\text{-closure}(\{\ \} \cup \{q_0, q_3\} \cup \{q_2\})$

$\qquad\qquad = \varepsilon\text{-closure}(\{q_0, q_2, q_3\})$

$\qquad\qquad\quad = \varepsilon\text{-closure}(\{q_0\}) \cup \varepsilon\text{-closure}(\{q_2\}) \cup \varepsilon\text{-closure}(\{q_3\})$

$\qquad\qquad\quad = \{q_0, q_1, q_2\} \cup \{q_2\} \cup \{q_3\}$

$\qquad\qquad\quad = \{q_0, q_1, q_2, q_3\}$

- Let = $(Q, \Sigma, \delta, q_0, F)$ be an NFA-ε and let w be in Σ^*. Then w is *accepted* by M iff $\delta^\wedge(\{q_0\}, w)$ contains at least one state in F.

- Let M = $(Q, \Sigma, \delta, q_0, F)$ be an NFA-ε. Then the *language accepted* by M is M the set:
  L(M) = {w | w is in Σ^* and $\delta^\wedge(\{q_0\}, w)$ contains at least one state in F}

- Another equivalent definition:
  L(M) = {w | w is in Σ^* and w is accepted by M}

- 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)

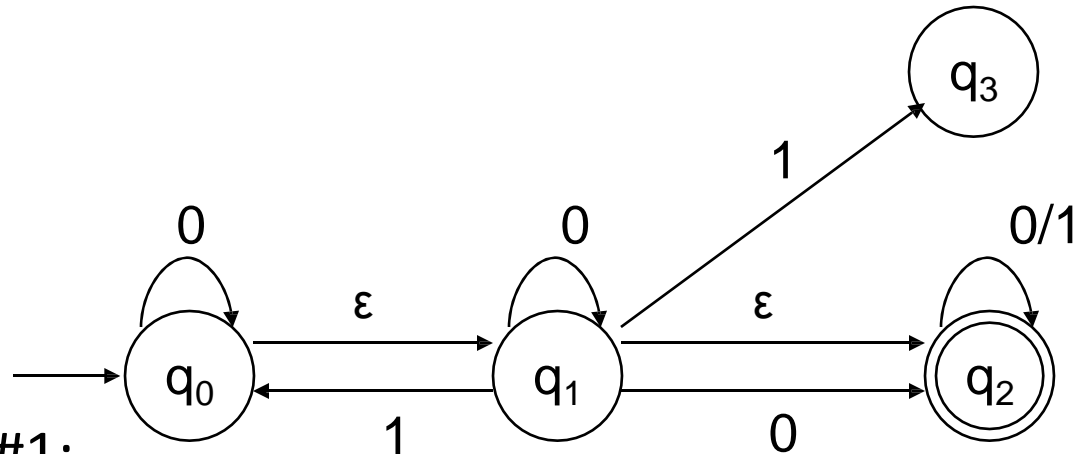 Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA-ε.

  Define an NFA $M' = (Q, \Sigma, \delta', q_0, F')$ as: $F' = F \cup \{q\}$ if ε-closure(q) contains at least one state from F

      $F' = F$ otherwise

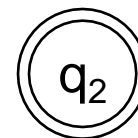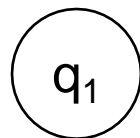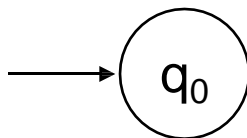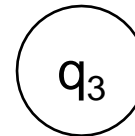 $\delta'(q, a) = \delta^\wedge(q, a)$ for all q in Q and a in Σ

- Notes:
  - $\delta': (Q \times \Sigma) \rightarrow 2^Q$ is a function
  - M' has the same state set, the same alphabet, and the same start state as M
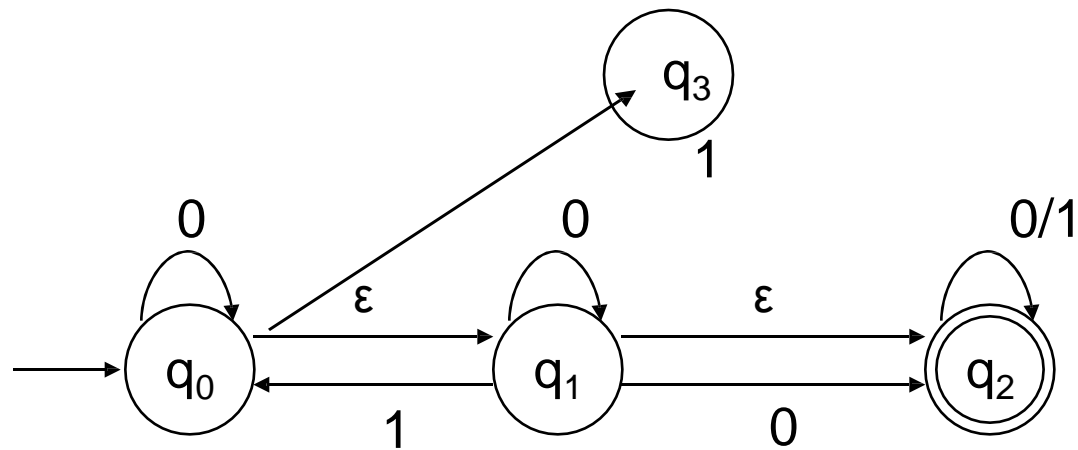
- **Step #1:**
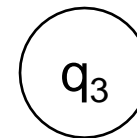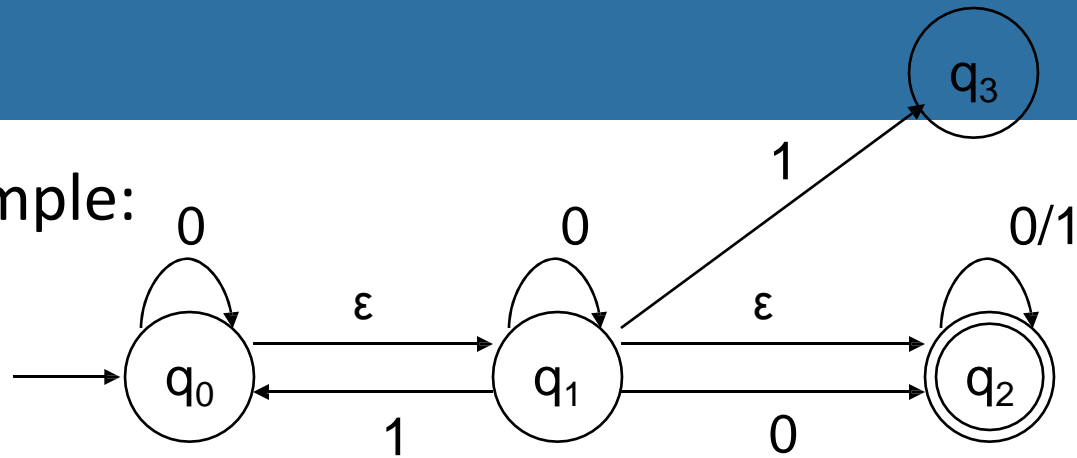 - Same state set as M
  - q_0 is the starting state

- Step #2:
  - q_0 becomes a final state

- Example:

- Step #3:
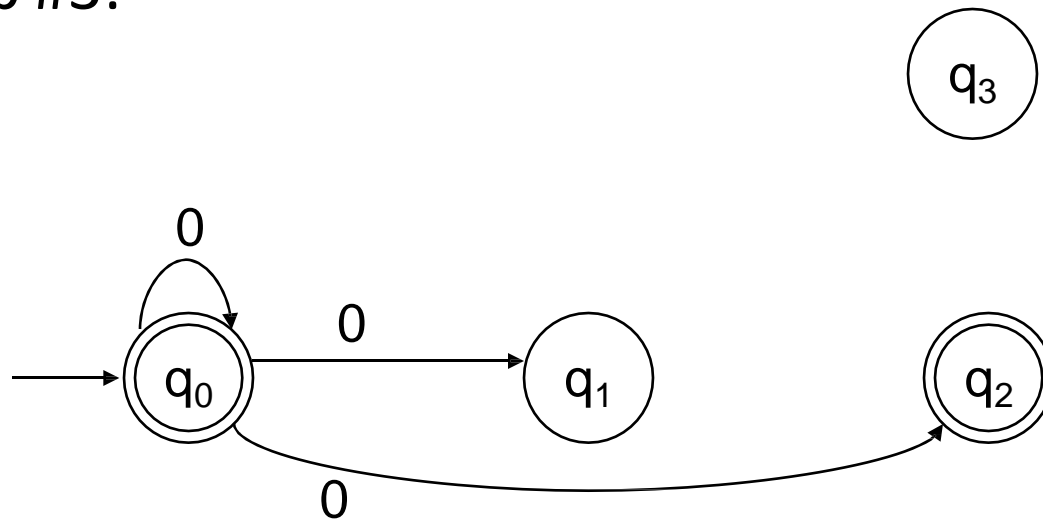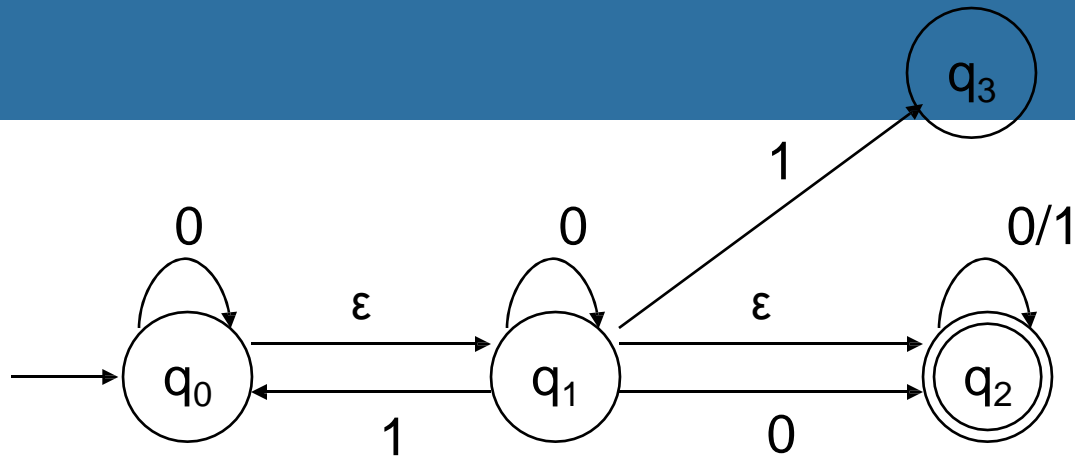
- Step #4:
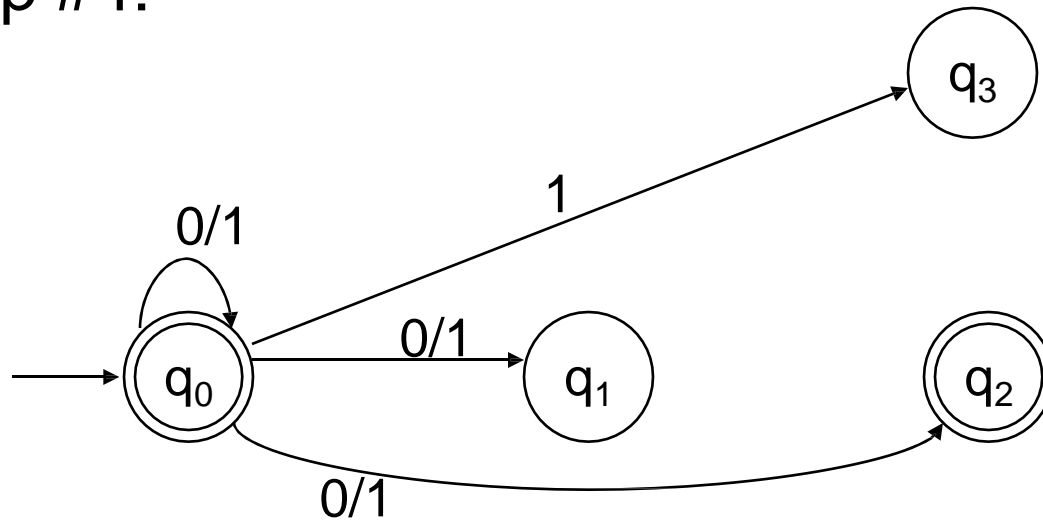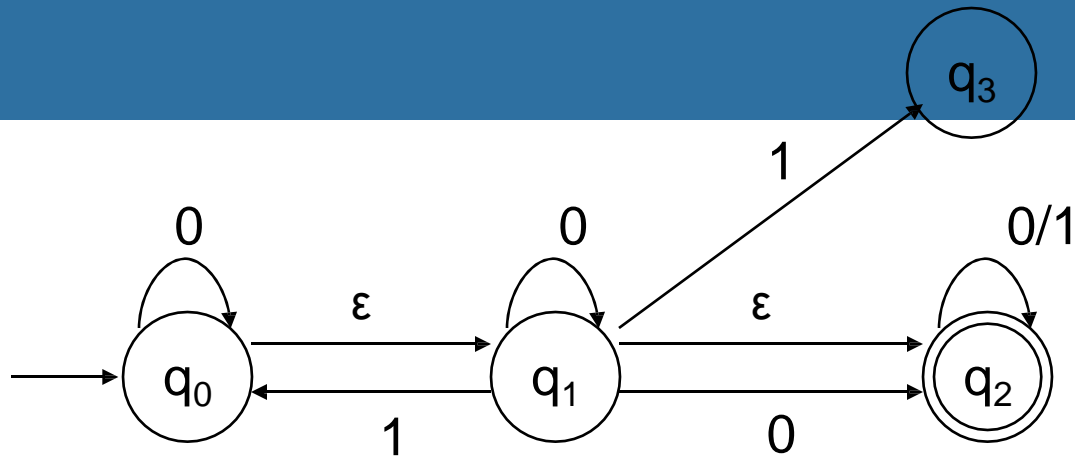
- Step #5:
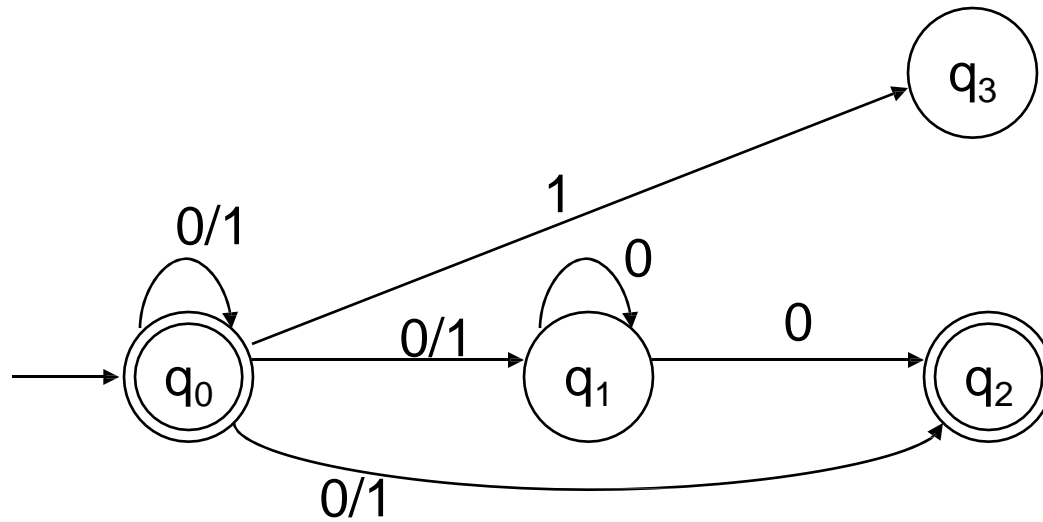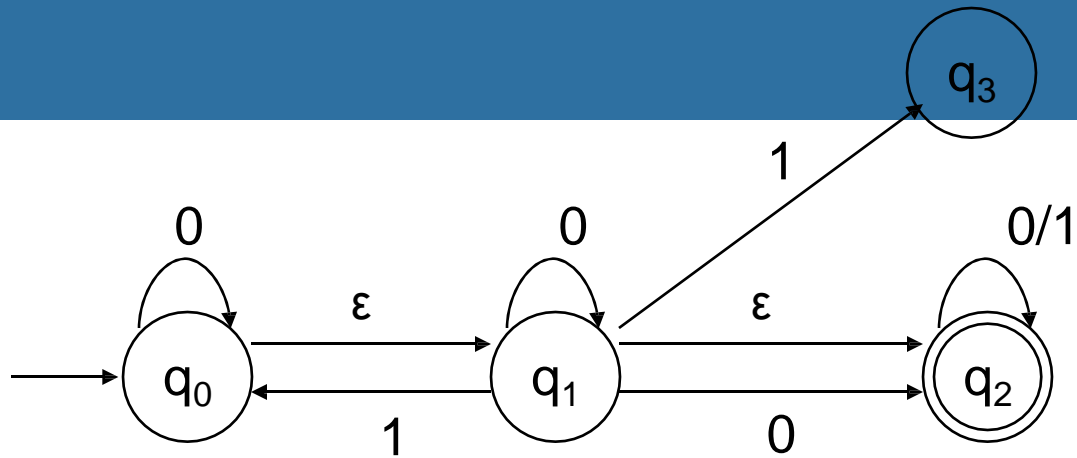
- Step #6:
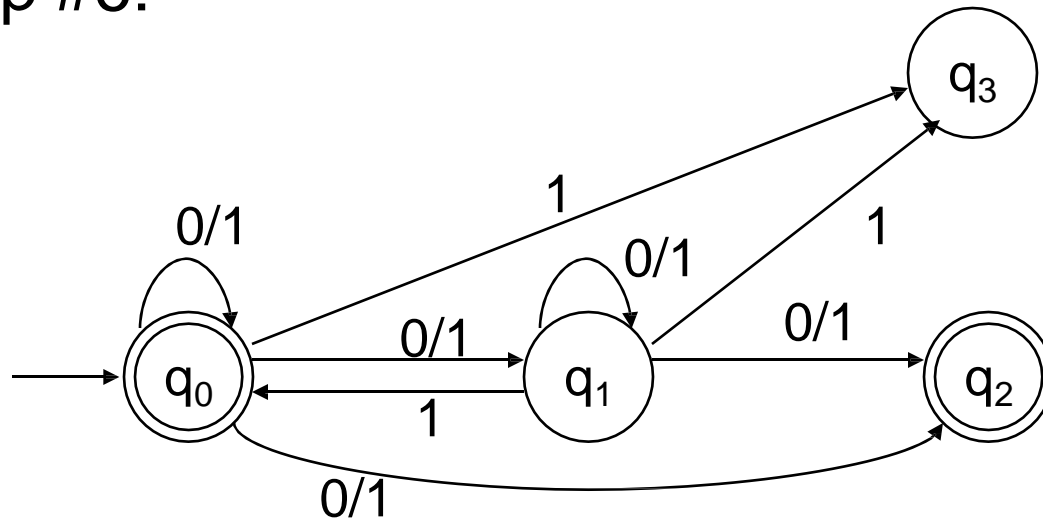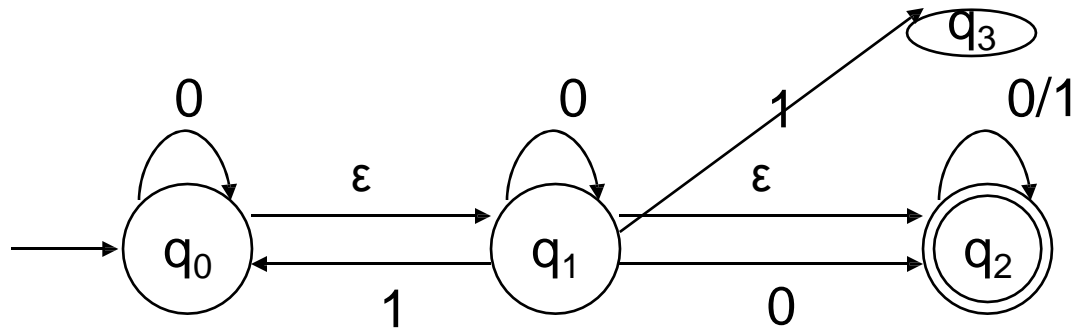
■ Step #7:
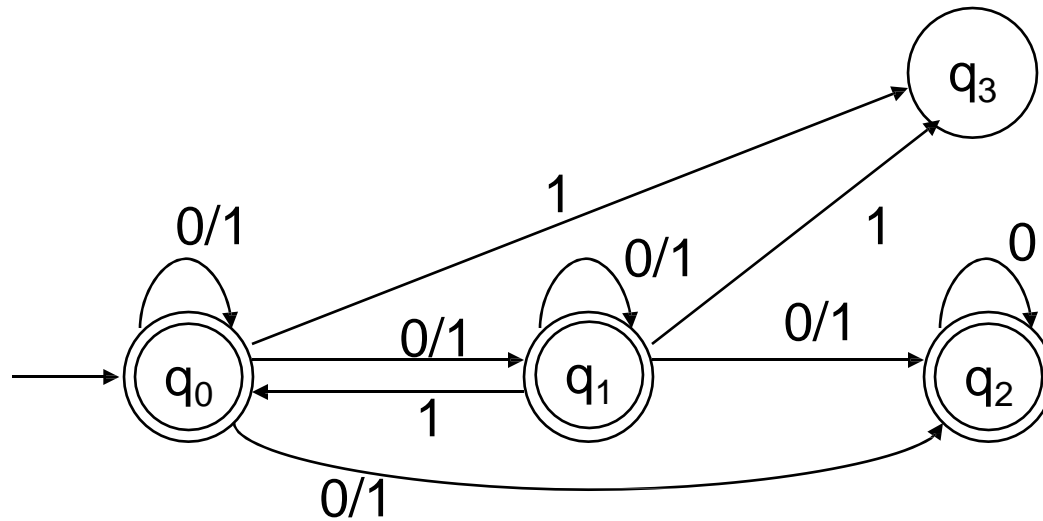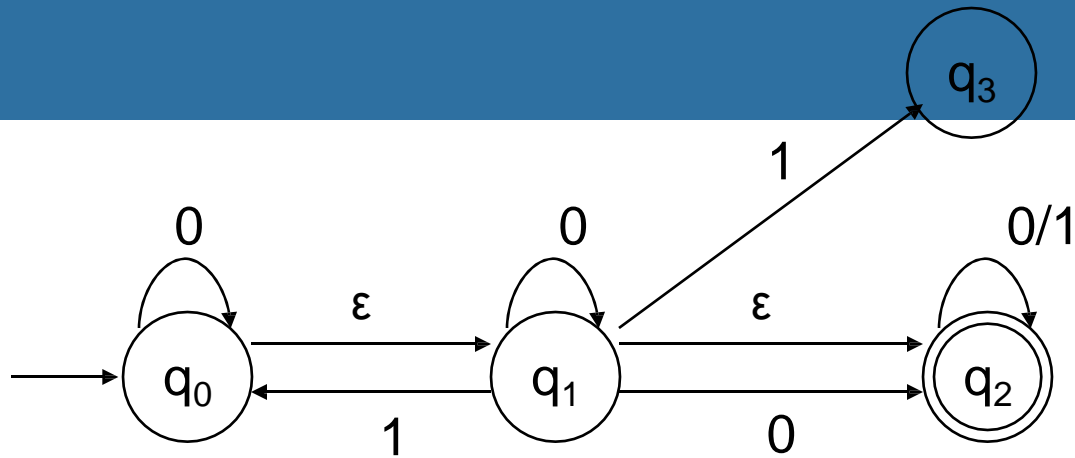
- **Step #8:** [use table of e-closure]
  - Done!
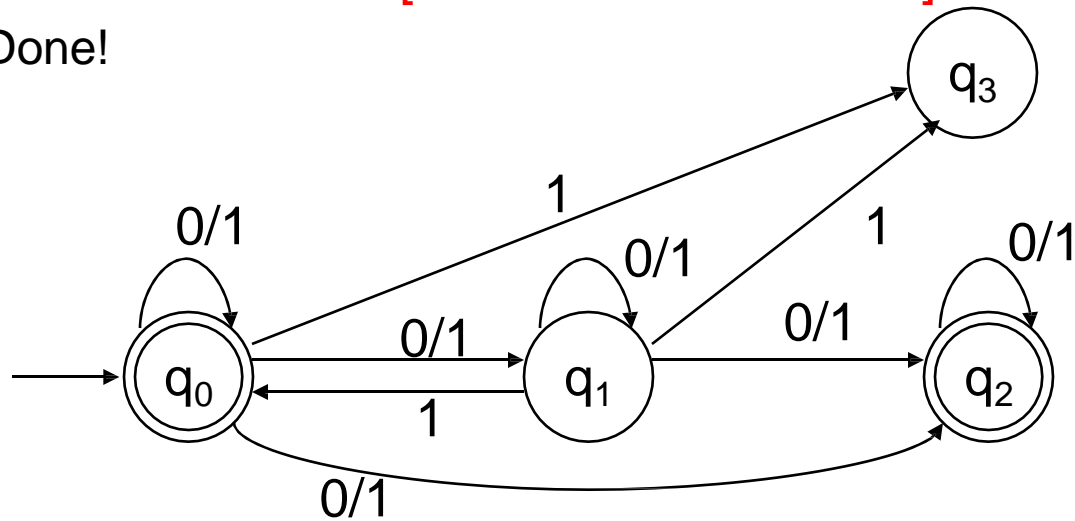
- **Theorem:**Let L be a language.   Then there exists an NFA M such that L= L(M) iff there exists an NFA-ε M' such that L = L(M').
- **Proof:**

  (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
Unit- II

Syllabus:

Regular sets, regular expressions, identity rules, constructing finite automata for a given regular expressions, conversion of finite automata to regular expressions, pumping lemma of regular sets, closure properties of regular sets (proofs not required), regular grammars-right linear and left linear grammars, equivalence between regular linear grammar and finite automata, inter conversion.

.

Family of languages

- **Seed elements:**
    - Empty language
    - Language containing the empty string
    - Singleton language for each letter in the alphabet
- **Closure Operations:**
    - Union: collects strings from languages
    - Concatenation: generates longer strings
    - Kleene Star: generates infinite languages

Basis: $\phi, \{\lambda\}, \text{ and } \forall a \in \Sigma : \{a\}$ are

- regular sets over Σ.

- **Inductive Step**: Let X and Y be regular sets over $\sum$ Then so are: $X \cup Y$

  XY

  x^*

- **Closure**:…

Examples

- Bit strings containing at least a "1"

$$(\{0\}\cup\{1\})*\{1\}(\{0\}\cup\{1\})*$$

- Bit strings containing exactly one "1"

$$\{0\}*\{1\}\{0\}*$$

- Bit strings beginning or ending with a "1"

$$\{1\}\{0,1\}* \cup \{0,1\}*\{1\}$$

Regular Expressions

- Regular expressions
  are an algebraic way to describe languages.

- They describe exactly the regular languages.

- If E is a regular expression, then L(E) is the language it defines.

- We'll describe RE's and their languages recursively.

Definition

- **Basis 1**: If *a* is any symbol, then **a** is a RE, and L(**a**) = {a}.
  - **Note**: {a} is the language containing one string, and that string is of length 1.
- **Basis 2**: ε is a RE, and L(ε) = {ε}.
- **Basis 3**: ∅ is a RE, and L(∅) = ∅.

- Induction 1: If E_1 and E_2 are regular expressions, then E_1+E_2 is a regular expression, and $L(E_1+E_2) = L(E_1)\cup L(E_2)$.

- Induction 2: If E_1 and E_2 are regular expressions, then E_1E_2 is a regular expression, and $L(E_1E_2) = L(E_1)L(E_2)$.

Concatenation : the set of strings wx such that w Is in $L(E_1)$ and x is in $L(E_2)$.
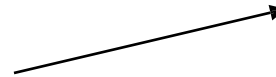
- Induction 3: If E is a RE, then E* is a RE, and $L(E^*) = (L(E))^*$.
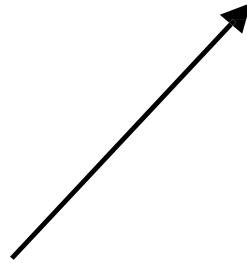
Closure, or "Kleene closure" = set of strings $w_1 w_2 ... w_n$, for some $n \geq 0$, where each w_i is in $L(E)$.
Note: when n=0, the string is ε.

- Parentheses may be used wherever needed to influence the grouping of operators.

- Order of precedence is * (highest), then concatenation, then + (lowest).

Examples

- L(**01**) = {01}.

- L(**01**+**0**) = {01, 0}.

- L(**0**(**1**+**0**)) = {01, 00}.

  - Note order of precedence of operators.

- L(**0***) = {ε, 0, 00, 000,… }.

- L((**0**+**10**)*(ε+**1**)) = all strings of 0's and 1's without two consecutive 1's.

- We need to show that for every RE, there is an automaton that accepts the same language.

  - Pick the most powerful automaton type: the ε-NFA.

- And we need to show that for every automaton, there is a RE defining its language.

  - Pick the most restrictive type: the DFA.

- Proof is an induction on the number of operators (+, concatenation, *) in the RE.

- We always construct an automaton of a special form (next slide).
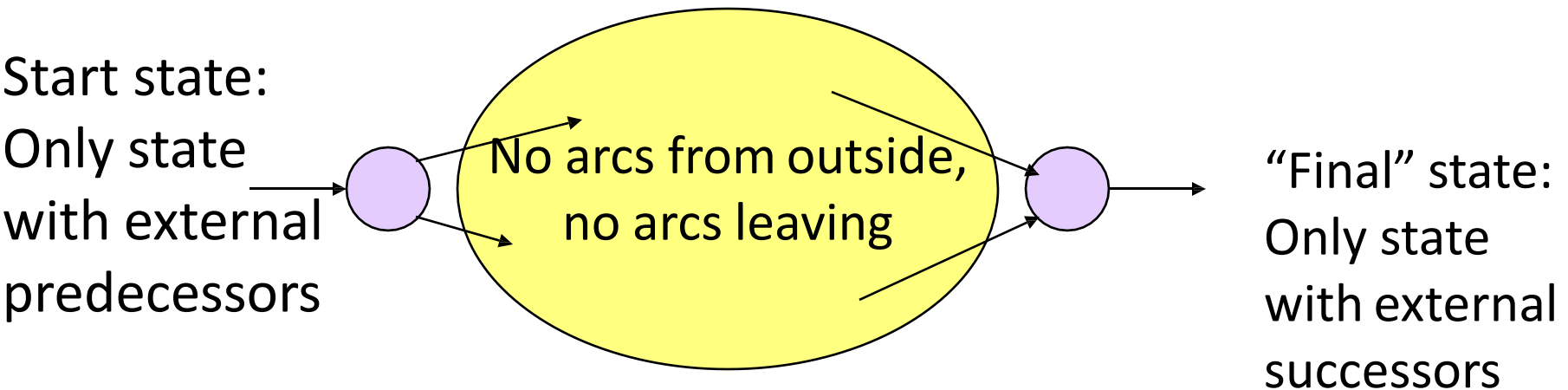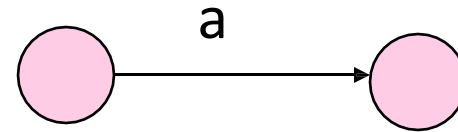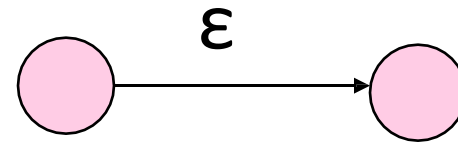
Start state:
Only state
with external
predecessors

No arcs from outside,
no arcs leaving

"Final" state:
Only state
with external
successors

- # Symbol **a**:

- # ε:

- ∅:

For E$_1$

For E$_2$

For E$_1 \cup$ E$_2$

For E_1

ϵ

For E_2

For E_1E_2
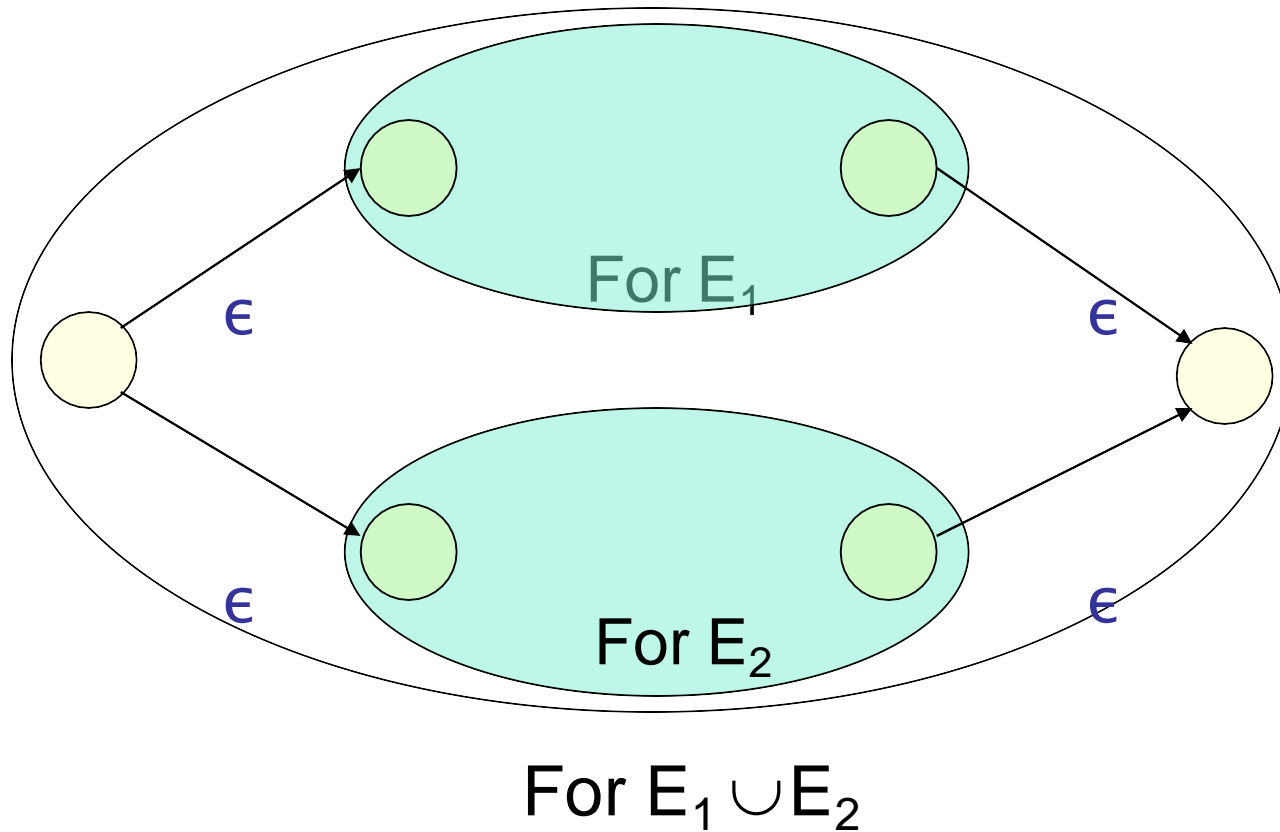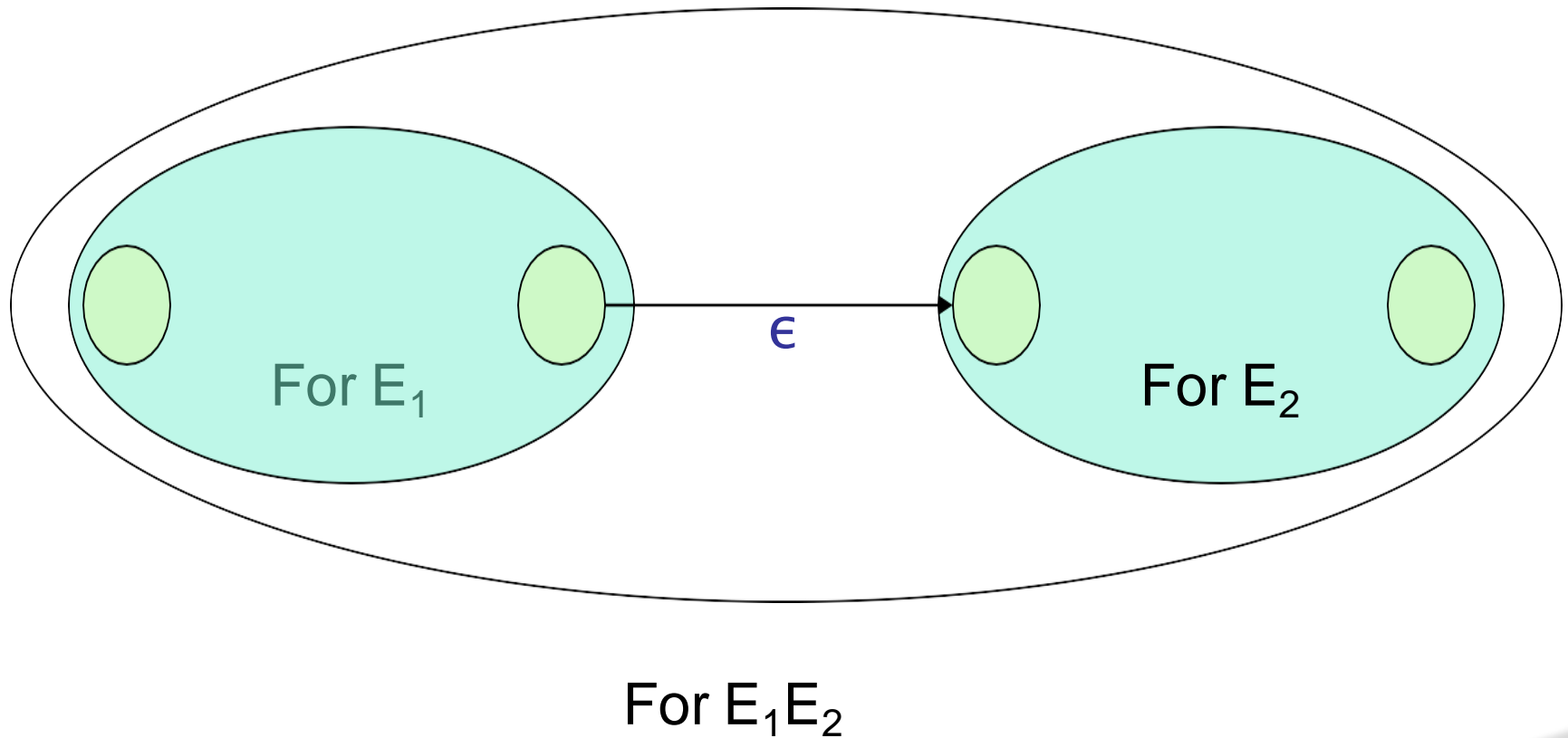
For E*
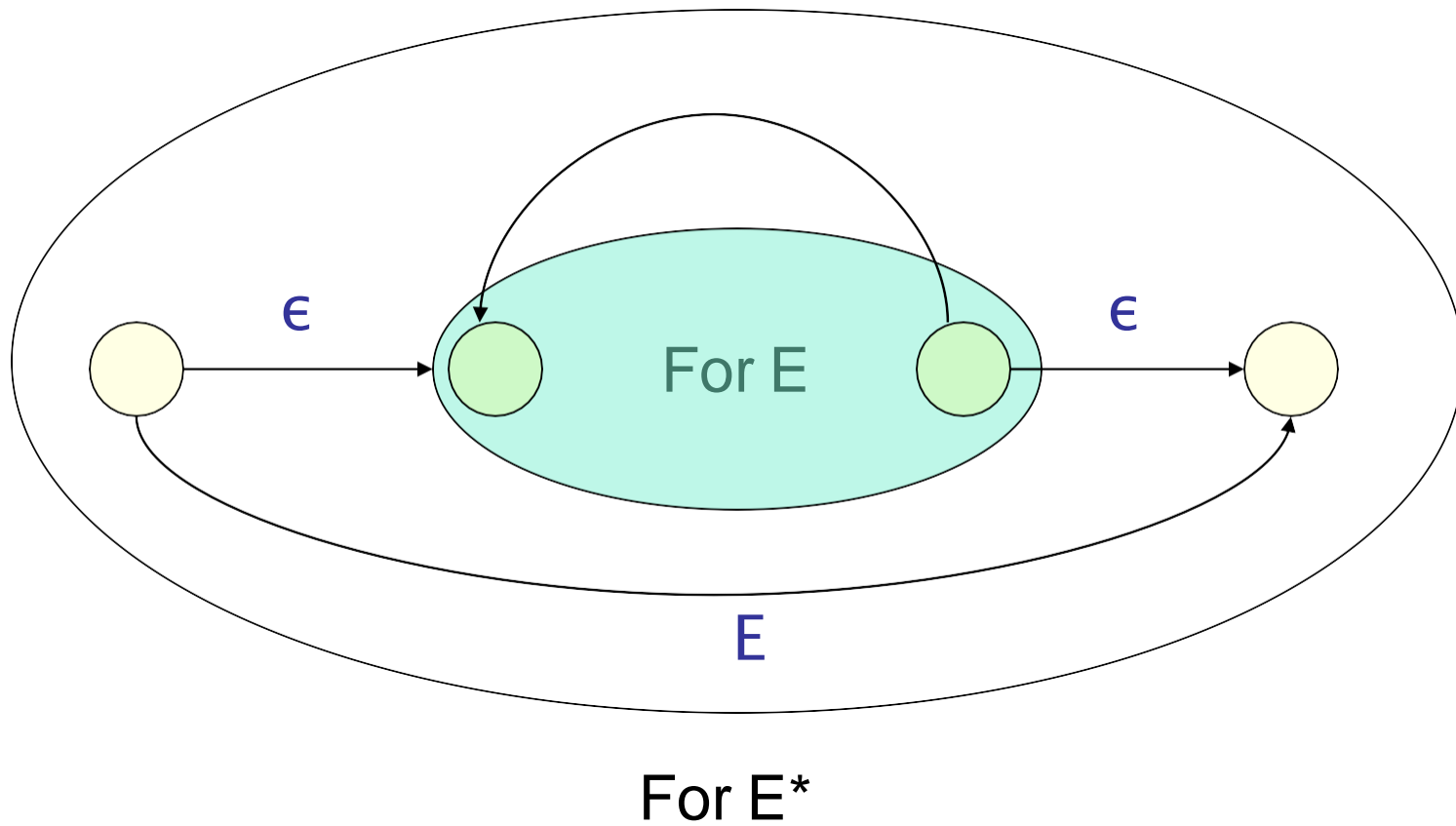
- A strange sort of induction.

- States of the DFA are assumed to be 1,2,…,n.

- We construct RE's for the labels of restricted sets of paths.

  - Basis: single arcs or no arc at all.

  - Induction: paths that are allowed to traverse next state in order.

- A k-path is a path through the graph of the DFA that goes though no state numbered higher than k.

- Endpoints are not restricted; they can be any state.

Example: k-Paths
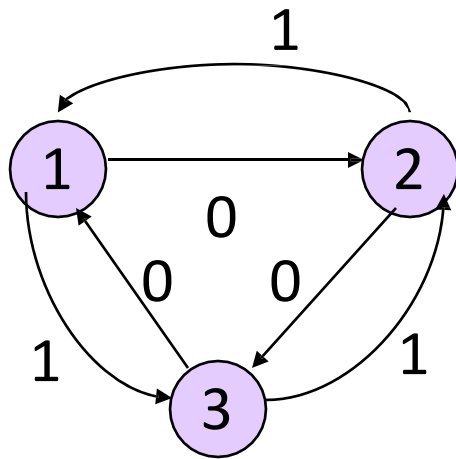
0-paths from 2 to 3:

RE for labels = **0**.

1-paths from 2 to 3:
RE for labels = **0**+**11**.

2paths from 2 to 3:
RE for labels =
(**10**)*****0**+**1**(**01**)*****1**

3paths from 2 to 3:
RE for labels = ??

- Let R_{ij}^k be the regular expression for the

set of labels of k-paths from state i to state j.

- **Basis**: k=0. R_{ij}^0 = sum of labels of arc from i to j.
  - ∅ if no such arc.
  - But add ε if i=j.

Example: Basis

- $R_{12}{}^0 = \mathbf{0}.$
- $R_{11}{}^0 = \varnothing + \varepsilon = \varepsilon.$
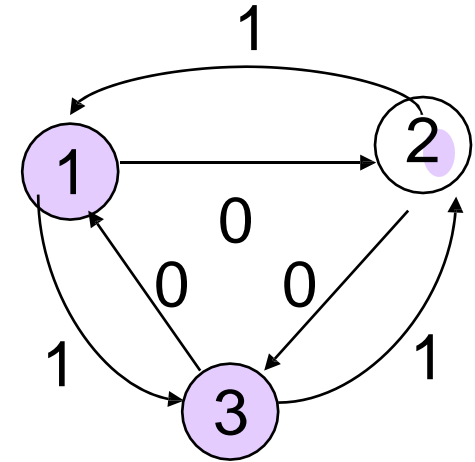
- A k-path from i to j either:
  1. Never goes through state k, or
  2. Goes through k one or more times.

$$R^{k}_{ij} = R^{k-1}_{ij} + R^{k-1}_{ik} (R^{k-1}_{kk})^{*} R^{k-1}_{kj}.$$

Doesn't go through k

Goes from i to k the first time

Zero or more times from k to k

Then, from k to j
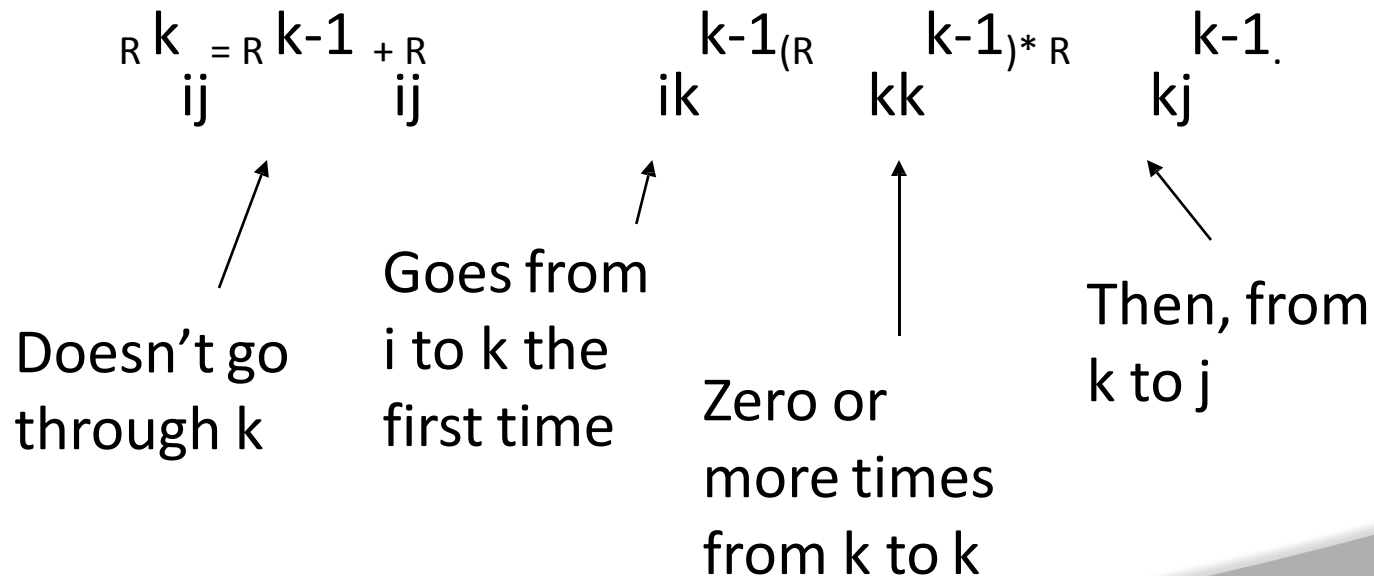
Illustration of Induction
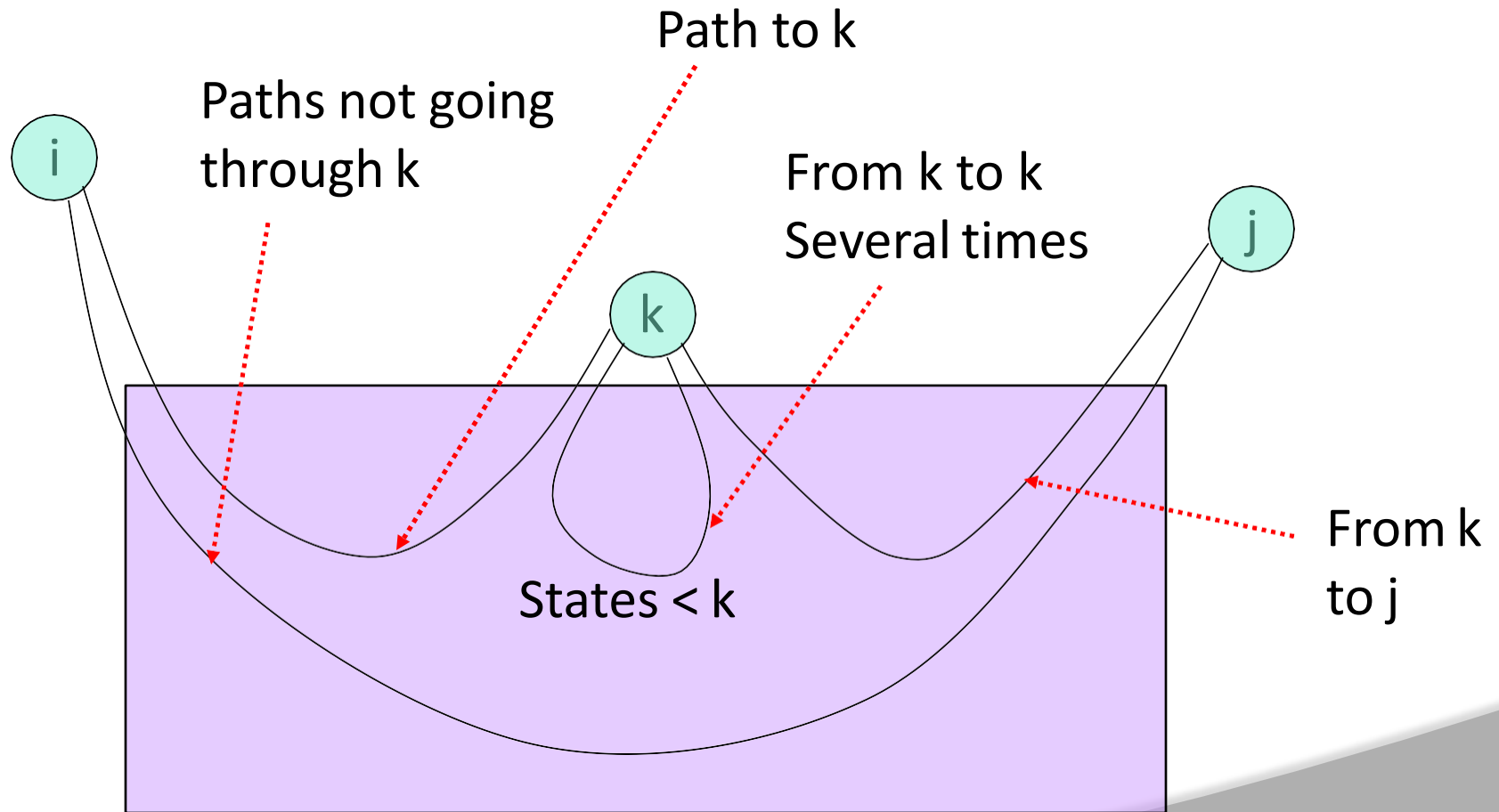
Final Step

■   The RE with the same language as
the DFA is the sum (union) of R_{ij}^{n},

   where:

1. n is the number of states; i.e., paths are unconstrained.

2. i is the start state.

3. j is one of the final states.

Example

- $R_{23}{}^3 = R_{23}{}^2 + R_{23}{}^2 (R_{33}{}^2)^* R_{33}{}^2 = R_{23}{}^2 (R_{33}{}^2)^*$

- $R_{23}{}_2 = (10)^*0+1(01)^*1$

- $R_{33}{}_2 = 0(01)^*(1+00) + 1(10)^*(0+11)$

- $R_{23}{}^3 = [(10)^*0+1(01)^*1] [(0(01)^*(1+00)+$

   $1(10)^*(0+11))]^*$
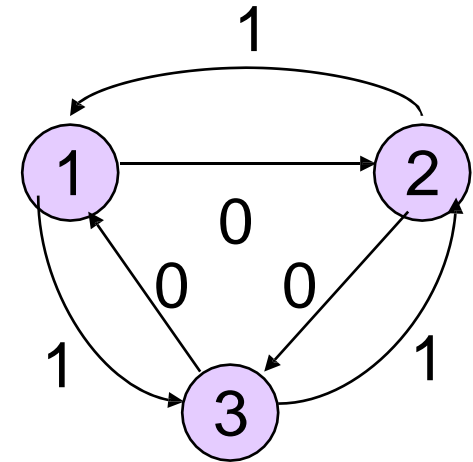
- Union and concatenation behave sort of like addition and multiplication.

  - + is commutative and associative; concatenation is associative.

  - Concatenation distributes over +.

  - Exception: Concatenation is not commutative.

Identities and Annihilators

- $\emptyset$ is the identity for +.
  - $R + \emptyset = R$.
- ε is the identity for concatenation.
  - $\varepsilon R = R\varepsilon = R$.
- $\emptyset$ is the annihilator for concatenation.
  - $\emptyset R = R\emptyset = \emptyset$.

Union, Intersection, Difference, Concatenation,
Kleene Closure, Reversal, Homomorphism,
Inverse Homomorphism

- Recall a closure property is a statement that a certain operation on languages, when applied to languages in a class (e.g., the regular languages), produces a result that is also in that class.

- For regular languages, we can use any of its representations to prove a closure property.

- If L and M are regular languages, so is L ⎕ M.

- Proof: Let L and M be the languages of regular expressions R and S, respectively.

- Then R+S is a regular expression whose language is L ⎕ M.

- Same idea:

    - RS is a regular expression whose language is LM.

    - R* is a regular expression whose language is L*.

Closure Under Intersection

- If L and M are regular languages, then so is L ⌧ M.

- Proof: Let A and B be DFA's whose languages are L and M, respectively.

- Construct C, the product automaton of A and B.

- Make the final states of C be the pairs consisting of final states of both A and B.
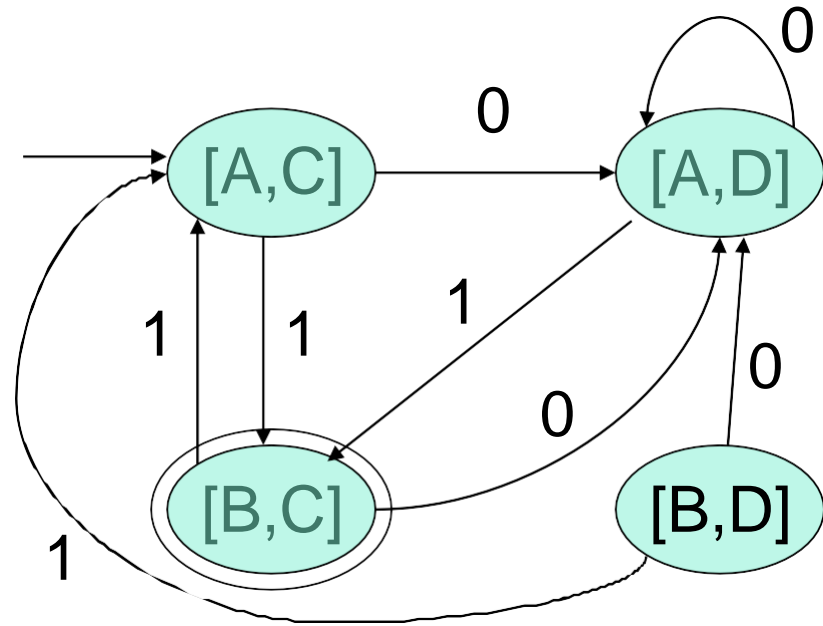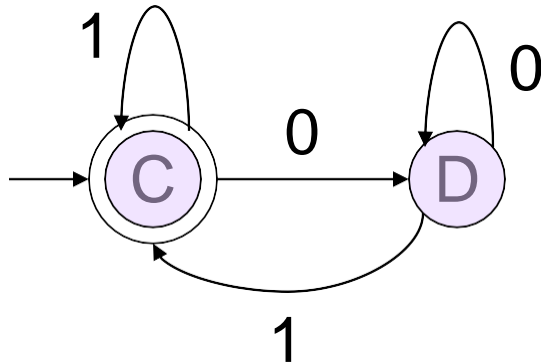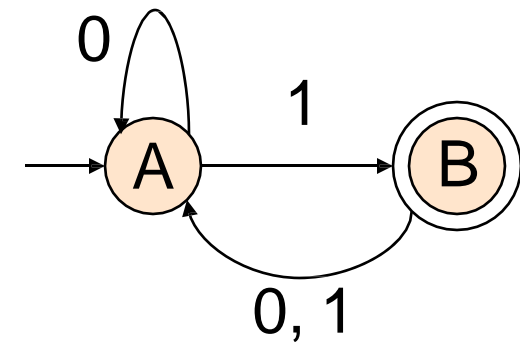
- If L and M are regular languages, then so is $L - M$ = strings in L but not M.

- Proof: Let A and B be DFA's whose languages are L and M, respectively.

- Construct C, the product automaton of A and B.

- Make the final states of C be the pairs where A-state is final but B-state is not.
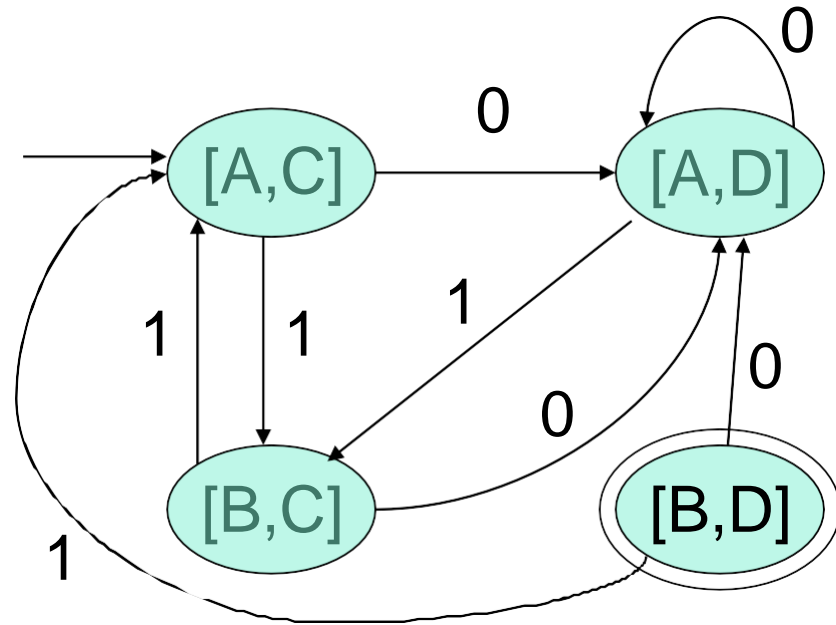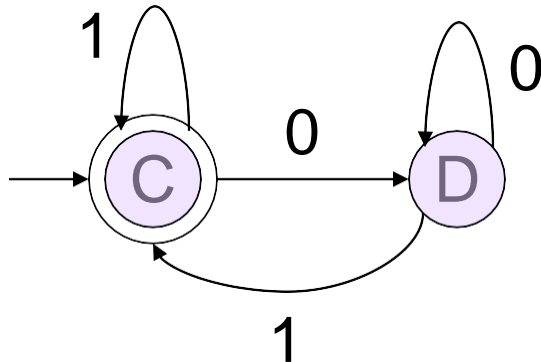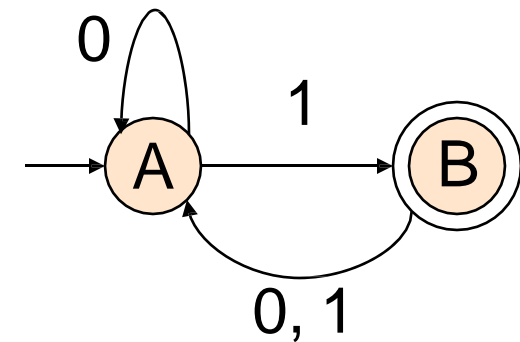
Notice: difference is the empty language

- The *complement* of a language L (with respect to an alphabet Σ such that Σ* contains L) is Σ* − L.

- Since Σ* is surely regular, the complement of a regular language is always regular.

- Recall example of a DFA that accepted the binary strings that, as integers were divisible by 23.

- We said that the language of binary strings whose reversal was divisible by 23 was also regular, but the DFA construction was very tricky.

- Good application of reversal-closure.

- Given language L, L^R is the set of strings whose reversal is in L.

- Example: L = {0, 01, 100};
  L^R = {0, 10, 001}.

- Proof: Let E be a regular expression for L.

- We show how to reverse E, to provide a regular expression E^R for L^R.

- **Basis**: If E is a symbol a, ε, or ∅, then $E^R = E$.

- **Induction**: If E is

  - F+G, then $E^R = F^R + G^R$.

  - FG, then $E^R = G^R F^R$

  - F*, then $E^R = (F^R)*$.

Example: Reversal of a RE

- Let $E = \mathbf{01^* + 10^*}$.

- $E^R = (\mathbf{01^* + 10^*})^R = (\mathbf{01^*})^R + (\mathbf{10^*})^R$

  $= (\mathbf{1^*})^R\mathbf{0}^R + (\mathbf{0^*})^R\mathbf{1}^R$

  $= (\mathbf{1}^R)\mathbf{^*0} + (\mathbf{0}^R)\mathbf{^*1}$

  $= \mathbf{1^*0 + 0^*1}$.

Homomorphisms

- A *homomorphism* on an alphabet is a function that gives a string for each symbol in that alphabet.

- Example: $h(0) = ab$; $h(1) = \varepsilon$.

- Extend to strings by $h(a_1 \ldots a_n) = h(a_1) \ldots h(a_n)$.

- Example: $h(01010) = ababab$.

Closure Under Homomorphism

- If L is a regular language, and h is a homomorphism on its alphabet, then h(L)
  = {h(w) | w is in L} is also a regular language.

- Proof: Let E be a regular expression for L.

- Apply h to each symbol in E.

- Language of resulting RE is h(L).

Example: Closure under Homomorphism

- Let h(0) = ab; h(1) = ε.

- Let L be the language of regular expression 01* + 0*.

- Then h(L) is the language of regular expression abε* + ε(ab)*.

- Note: use parentheses to enforce the proper grouping.

- abε* + ε(ab)* can be simplified.

- ε* = ε, so abε* = abε.

- ε is the identity under concatenation.

  - That is, εE = Eε = E for any RE E.

- Thus, abε* + ε(ab)* = abε + ε(ab)* = ab+ (ab)*.

- Finally, L(ab) is contained in L((ab)*),

- so a RE for h(L) is (ab)*.

- Let h be a homomorphism and L a language whose alphabet is the output language of h.

- $h^{-1}(L)$ = {w | h(w) is in L}.

- Let $h(0) = ab$; $h(1) = \varepsilon$.

- Let $L = \{abab, baba\}$.

- $h^{-1}(L)$ = the language with two 0's and any number of 1's
  = $L(1*01*01*)$.

  Notice: no string maps to baba; any string with exactly
  two 0's maps to abab.

Linear Grammars

- Grammars with

- at most one variable at the right side

- of a production

$$S \rightarrow aSb \quad S \rightarrow \lambda$$

$$S \rightarrow Ab \quad A \rightarrow aAb \quad A \rightarrow \lambda$$

Grammar G :
$$S \rightarrow SS$$
$$S \rightarrow \lambda$$

$$S \rightarrow aSb \quad S$$

$$\rightarrow bSa$$

$$L(G) = \{w : \quad n_a(w) = n_b(w)\}$$

$$G: \qquad S \rightarrow A$$

$$A \rightarrow aB \mid \lambda$$

- Grammar :

$$B \rightarrow Ab$$

$$L(G) = \{a^n b^n : n \geq 0\}$$

- All productions have form:

$$A \rightarrow xB$$

or

$$A \rightarrow x$$

- Example:

$$S \rightarrow abS$$

$$S \rightarrow a$$

- All productions have form:

$$A \rightarrow Bx$$

or

$$A \rightarrow x$$

- Example : $S \rightarrow Aab$

$$A \rightarrow Aab \mid B$$

$$B \rightarrow a$$

Regular Grammars

- A regular grammar is any

- right-linear or left-linear grammar

- Examples:

$$G_1$$
$$S \rightarrow abS$$
$$S \rightarrow a$$

$$G_2$$
$$S \rightarrow Aab$$
$$A \rightarrow Aab \mid B \quad B \rightarrow a$$

■ Regular grammars generate regular languages

G_1 G2

$S\text{->}abS, S \rightarrow a$ $S \rightarrow Aab$

                                         $A \rightarrow Aab \mid B \quad B \rightarrow a$

$L(G_1) = (ab)*a$

                                         $L(G_2) = aab(ab)*$

Regular Grammars Generate Regular Languages

$$\left\{ \begin{array}{l} \text{Language generated} \\ \text{by Regular} \\ \text{Grammars} \end{array} \right\} \qquad \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Grammars} \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$
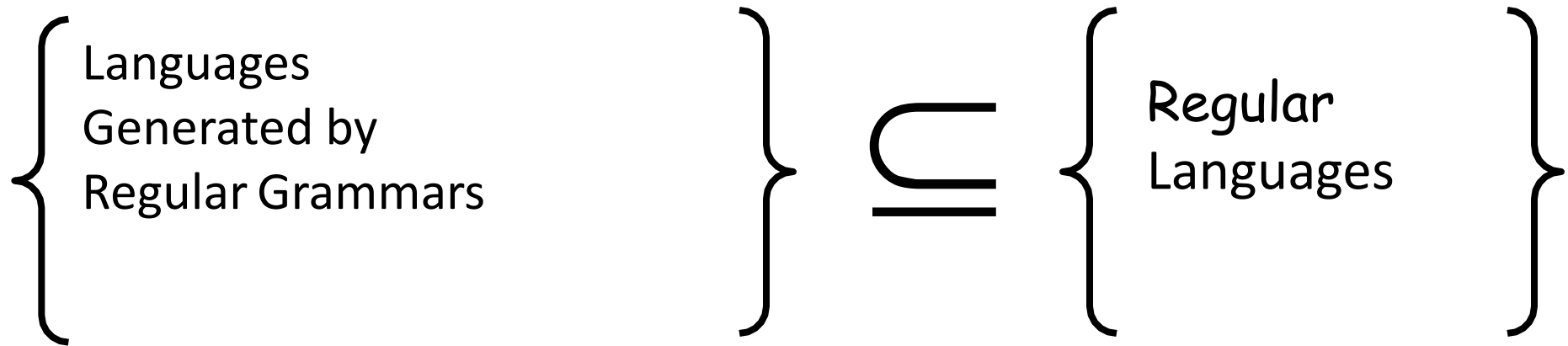
Any regular grammar generates
a regular language

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Grammars} \end{array} \right\} \supseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$
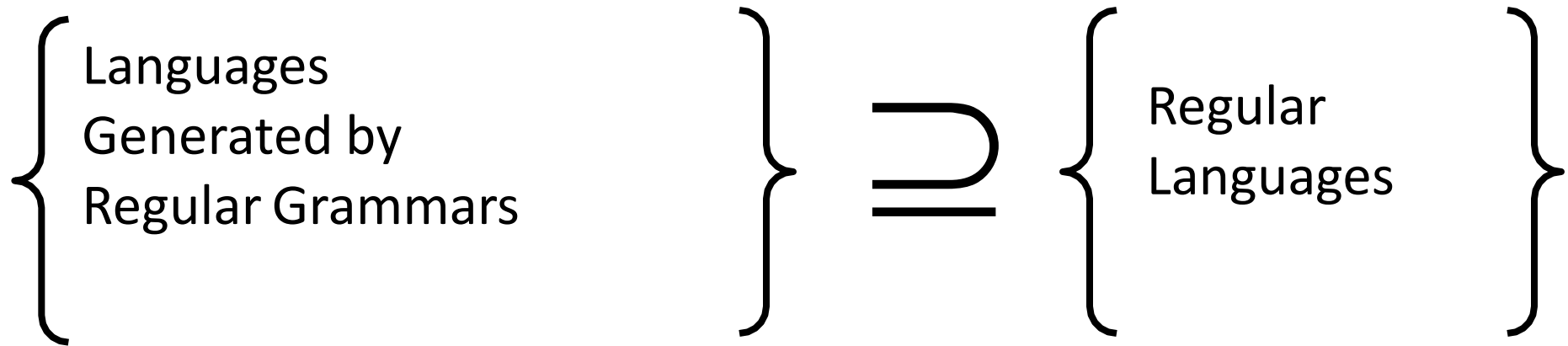
Any regular language is generated
by a regular grammar
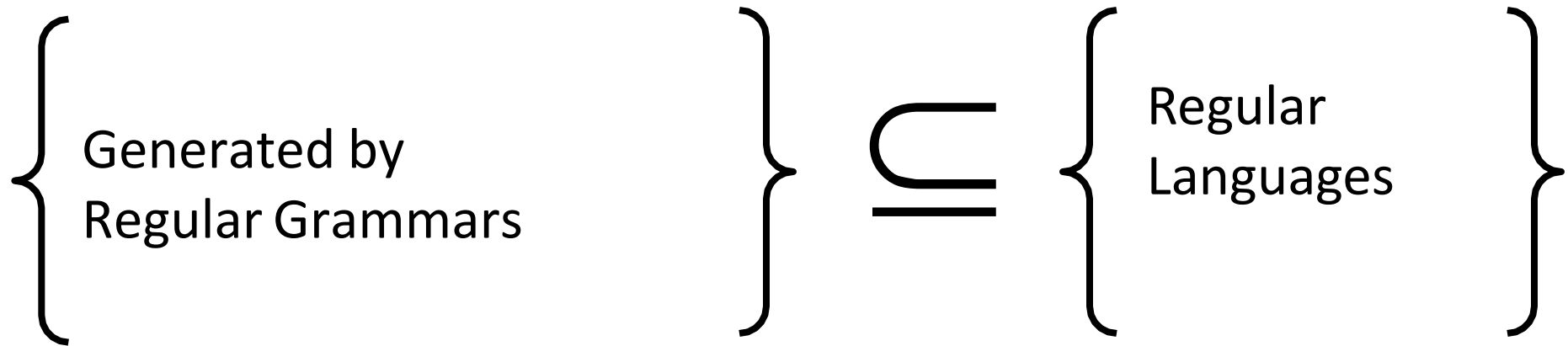
$$\left\{ \begin{array}{l} \text{Generated by} \\ \text{Regular Grammars} \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

The language   generated by
any regular grammar   is regular

$$G \qquad\qquad L(G)$$

G:

- Let   be a right-linear grammar $L(G)$

- We will prove: M is regular

$$L(M) = L(G)$$

- **Proof idea:**   We will construct NFA

  with

Example:

- Grammar G is right-linear

$$S \rightarrow aA \mid B \quad A \rightarrow$$
$$aa\,B \quad B \rightarrow b\,B \mid a$$

- Construct NFA M such that

- every state is a grammar variable:
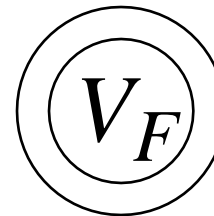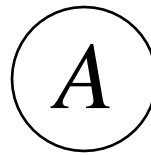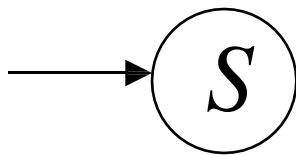
special final state
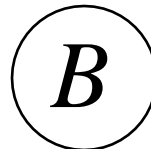
$S \rightarrow aA \mid B \ A \rightarrow$
$aa \ B \ B \rightarrow b \ B \mid a$

- **Add edges for each production:**
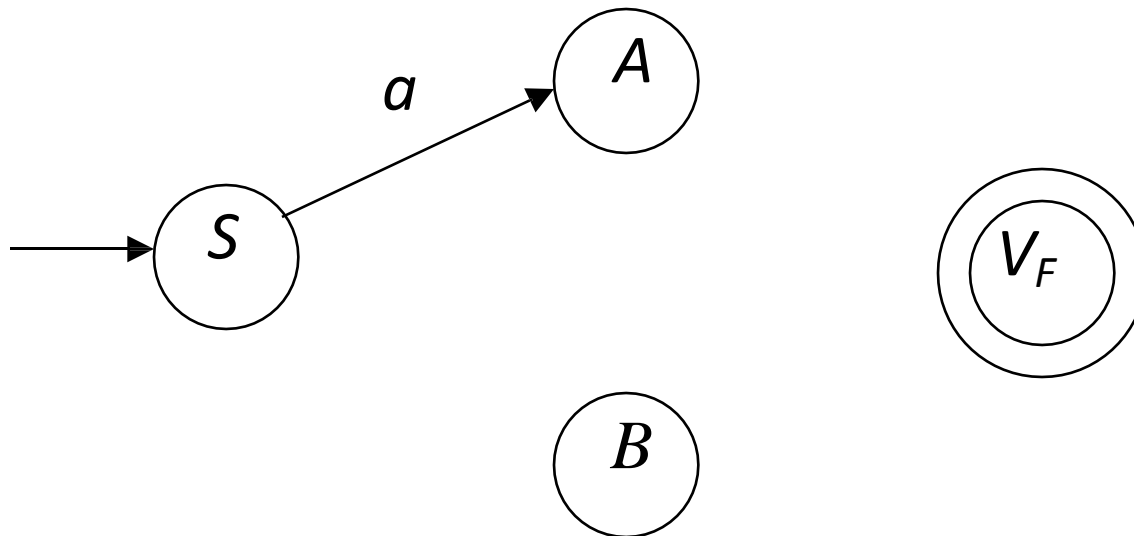
$$S \rightarrow aA$$
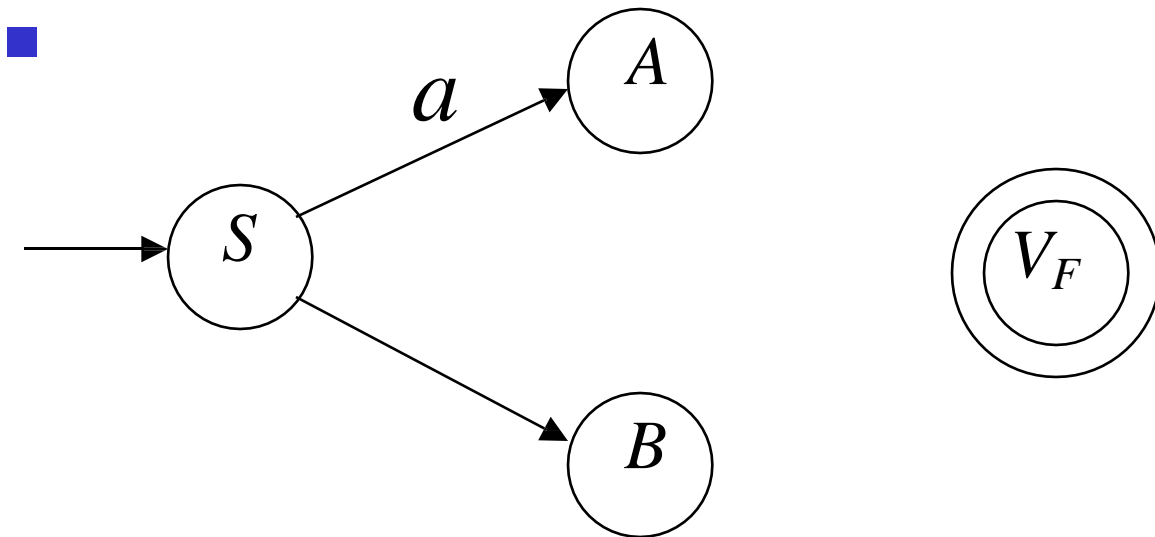
$S \rightarrow aA \mid B$
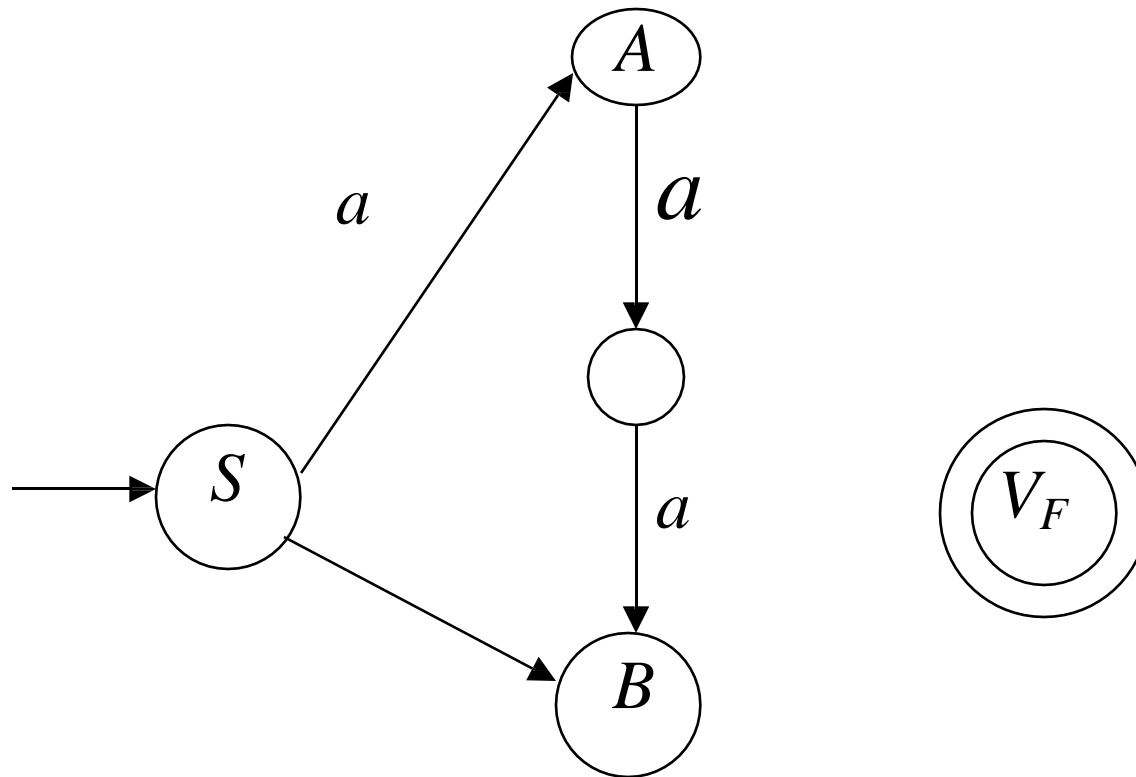
$S \rightarrow aA| \ B$

$A \rightarrow aaB$
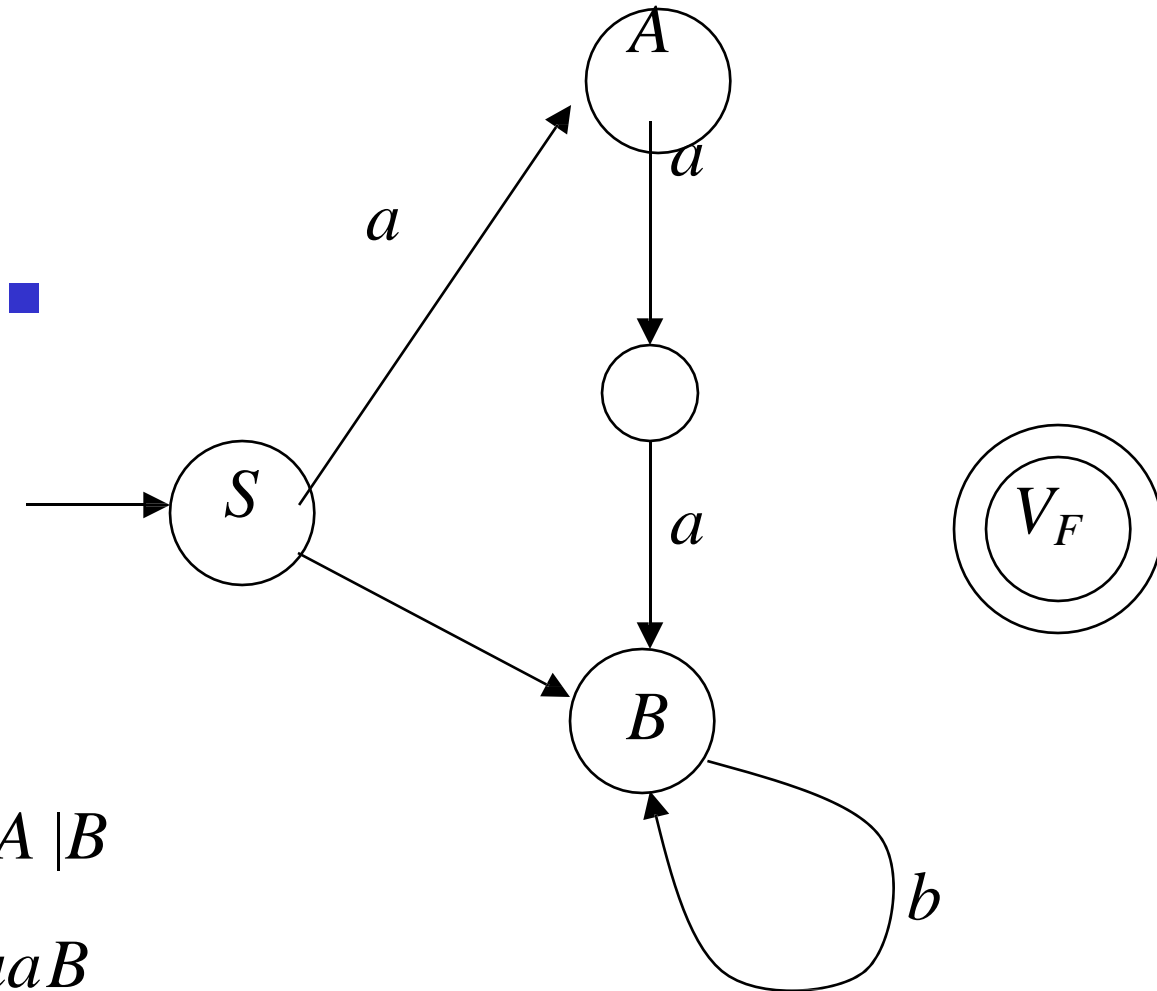
$S \to aA \mid B$

$A \to aaB$

$B \to bB$
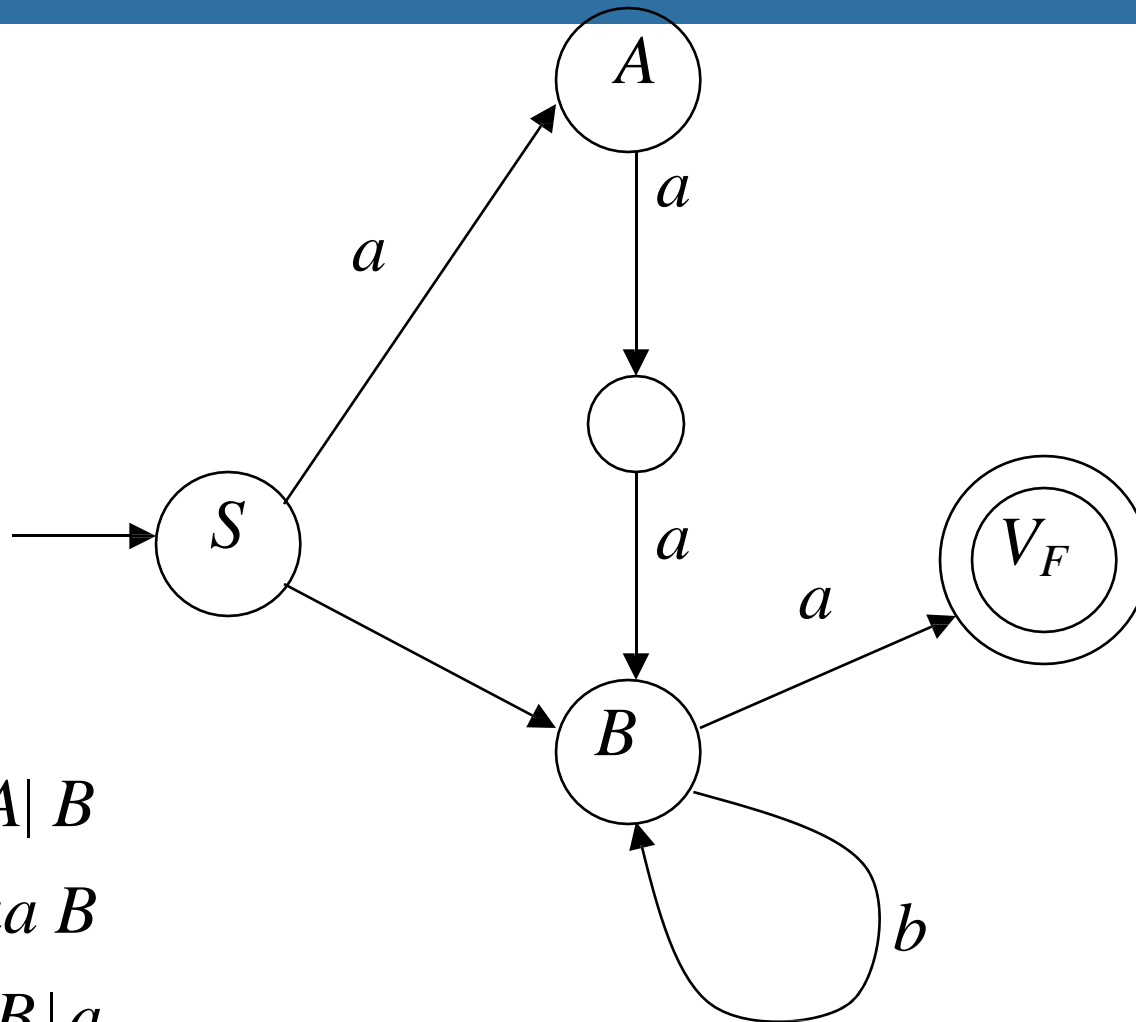
$$S \rightarrow aA \mid B$$

$$A \rightarrow aa\,B$$

$$B \rightarrow bB \mid a$$
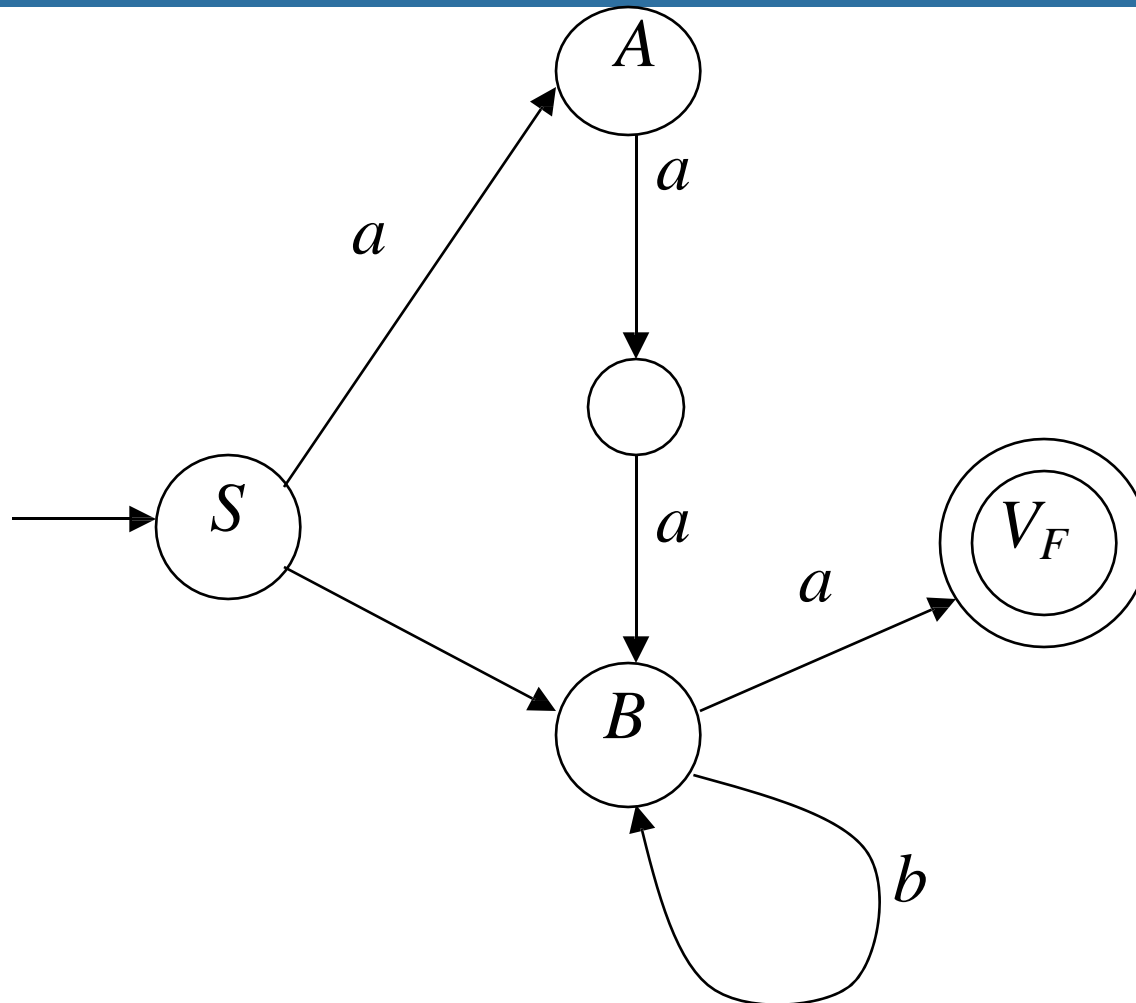
$$S \Rightarrow aA \Rightarrow aaaB \Rightarrow aaabB \Rightarrow aaaba$$

A

M

a

a

S

a

B

a

V_F

b
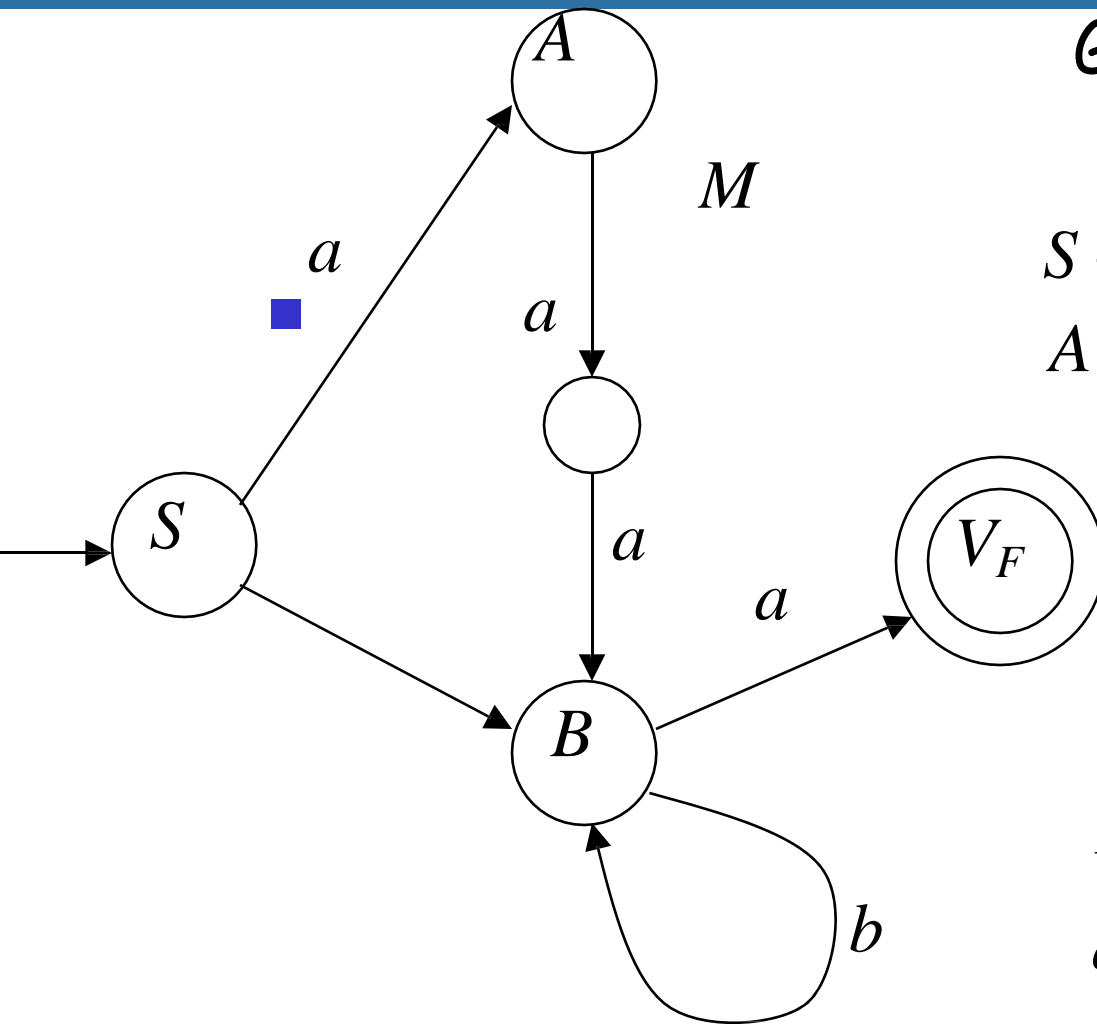
Grammar

G

$S \rightarrow aA|\ B$

$A \rightarrow aa\ B \quad B \rightarrow bB\,|\,a$

$L(M) = L(G) =$

$aaab*a + b*a$

In General

- A right-linear grammar $V_0, V_1, V_2, \ldots$

- has variables: G

$$V_i \rightarrow a_1 a_2 \cdots a_m V_j$$

or

- and productions:

$$V_i \rightarrow a_1 a_2 \cdots a_m$$

- We construct the NFA M such that:

- Each variable node: v_i corresponds to a

V_1

V_3

V_0

V_F

V_2

V_4

special
final
state
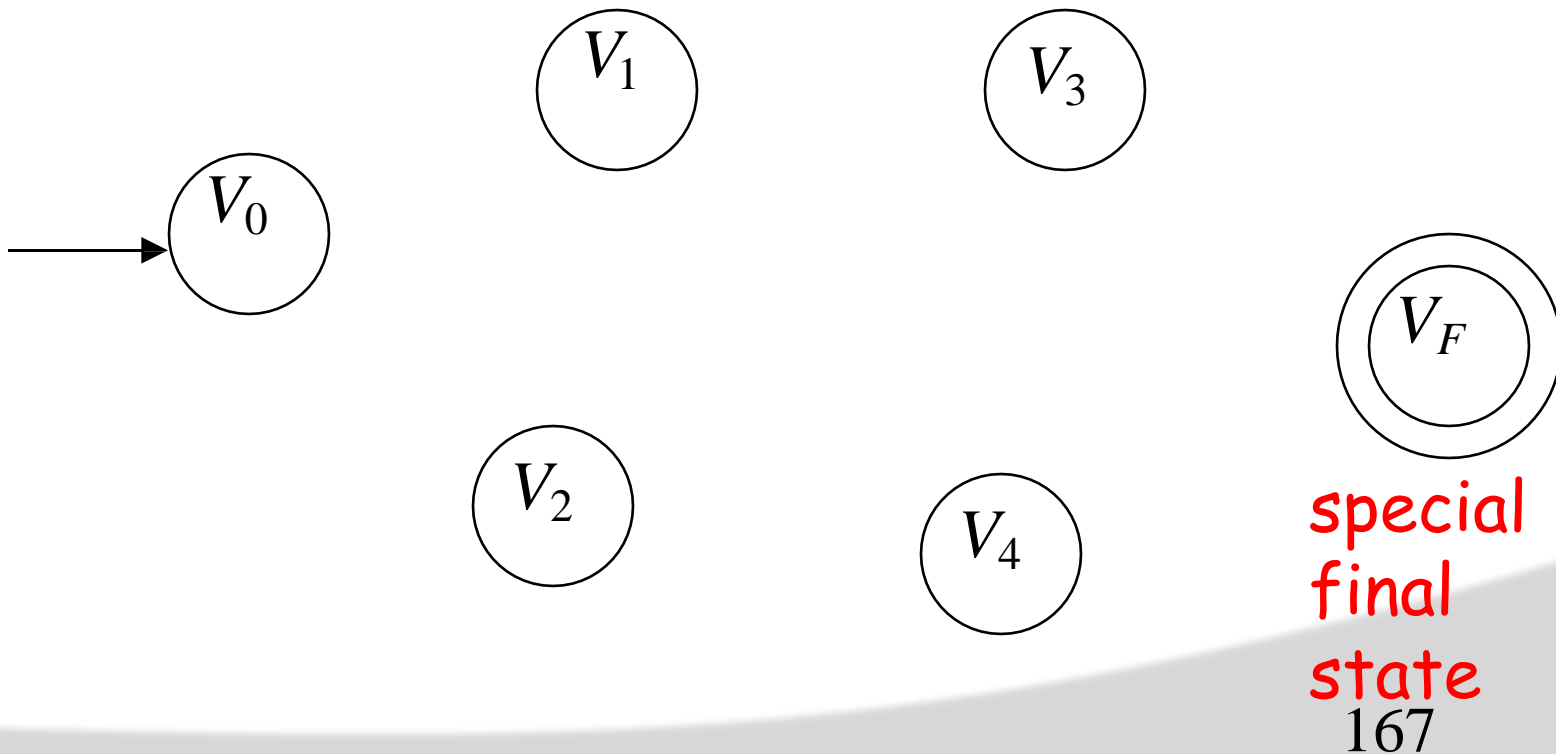
- For each production: v_i

- we add transitions and intermediate

nodes

$$V_i \xrightarrow{1} \bigcirc \xrightarrow{a_2} \bigcirc \rightarrow \cdots\cdots\cdots \rightarrow \bigcirc \xrightarrow{a_m} V_j$$
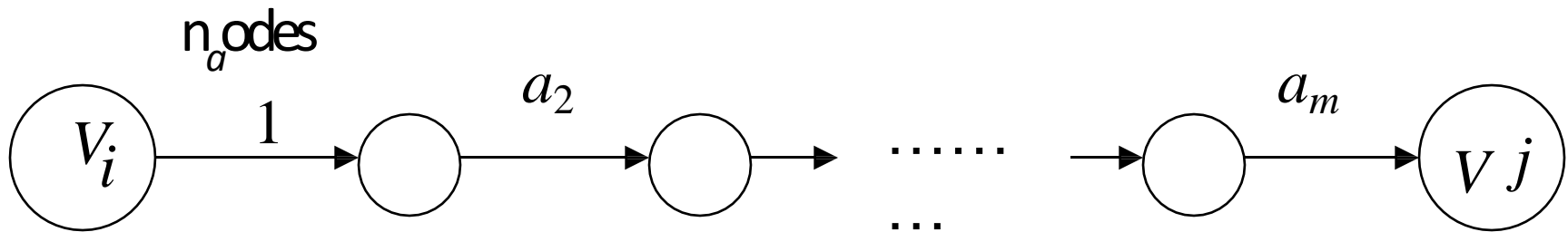
- For each production:
- we add transitions and intermediate

  nodes
  $$V_i \quad \rightarrow a_1 a_2 \cdots a_m$$
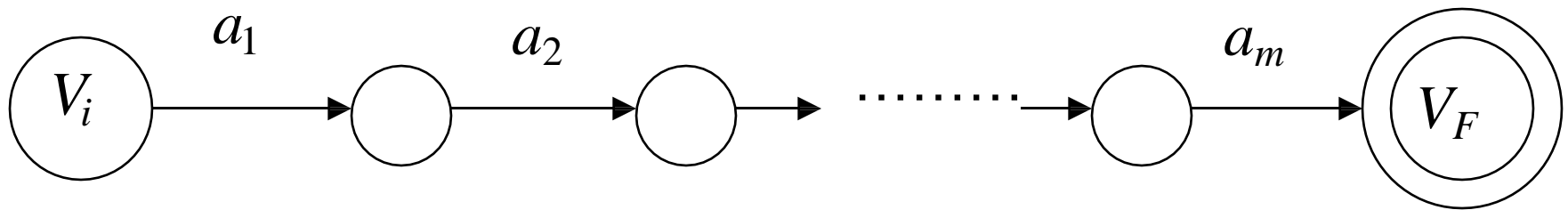
- Resulting NFA M looks like this:
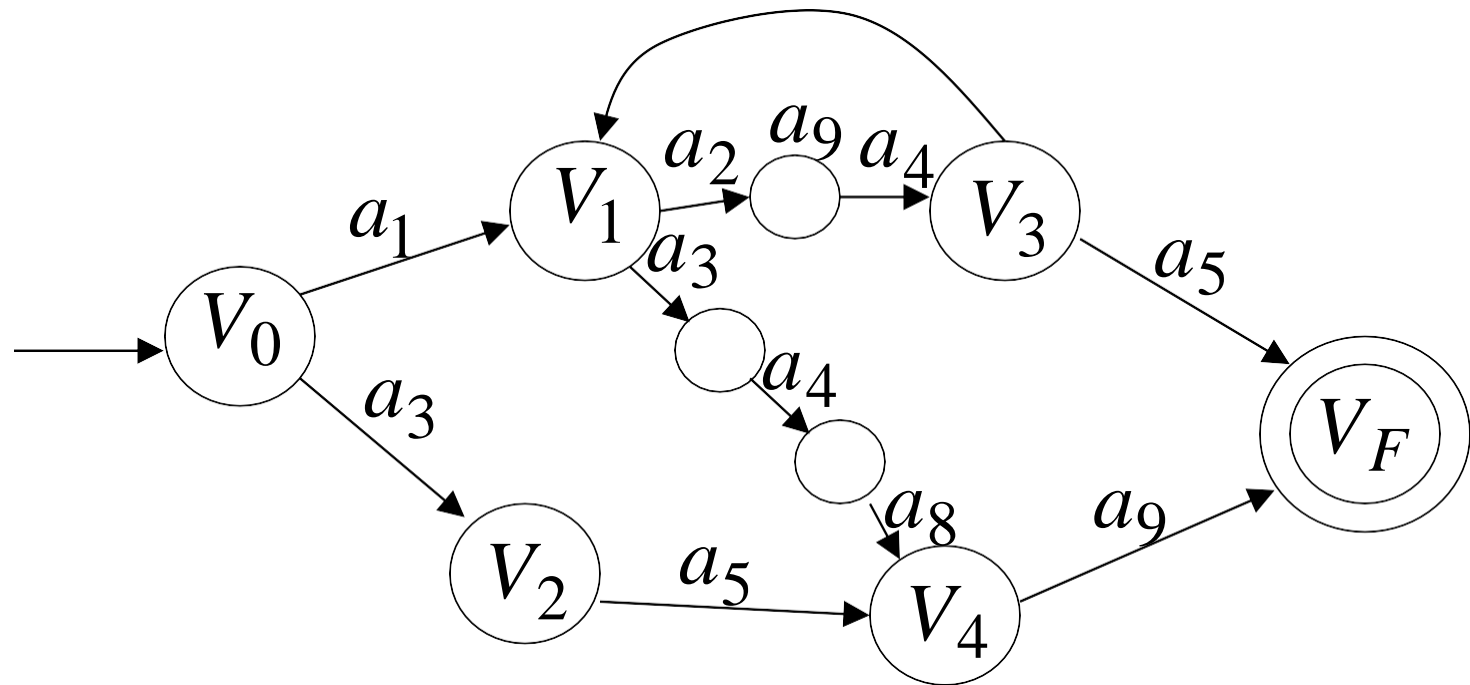
It holds that: $L(G) = L(M)$

Proof - Part 2

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Grammars} \end{array} \right\} \supseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

Any regular language L is generated G by some regular grammar

Any regular language L is generated G
by some regular grammar

Proof idea:

Let M be the NFA with $L = L(M)$

Construct from M a regular grammar G
such that

$$L(M) = L(G)$$

- Since L is regular

- there is an NFA M M_b such that

$$L = L(M)$$

Example:
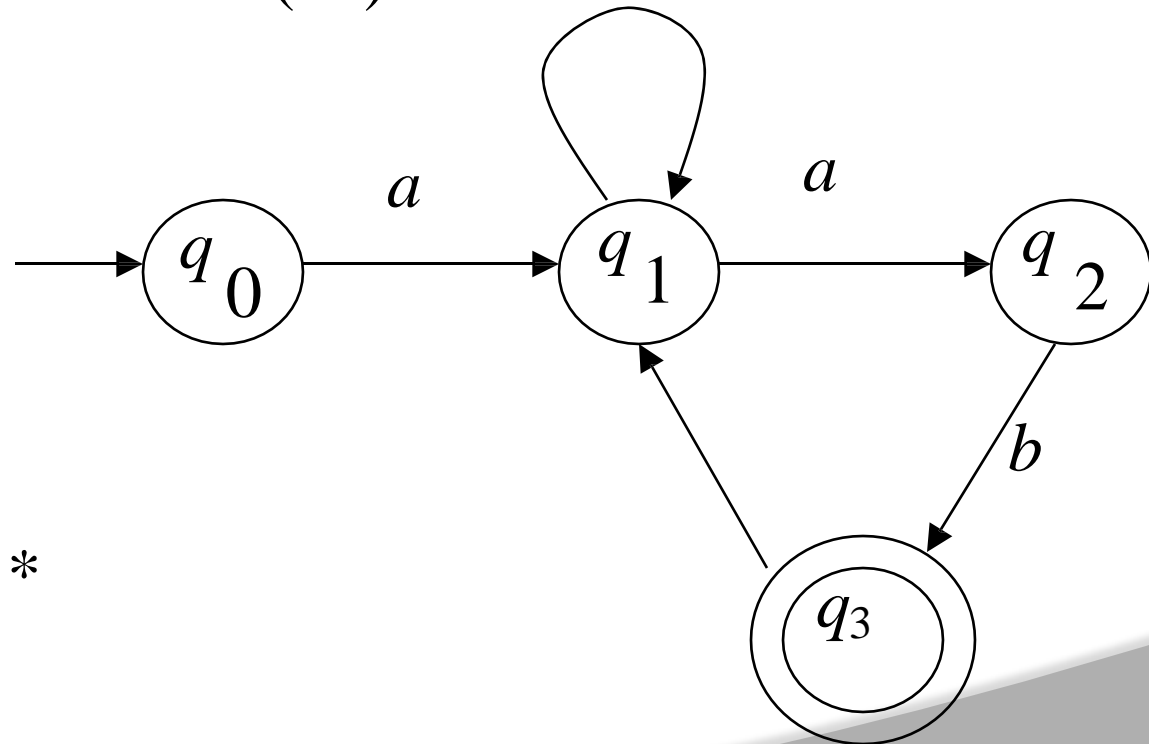
$$L = ab*ab(b*ab)*$$

$$L = L(M)$$

- **Convert** M **to a right-linear grammar**

M
$q_0 \rightarrow aq_1$

M

-

$q_0 \rightarrow aq_1$

$q_1 \rightarrow bq_1$

$q_1 \rightarrow aq_2$

b

M

$q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_2$

$q_1 \to q_1$ (loop with b)

$q_2 \xrightarrow{b} q_3$

$q_3 \to q_1$

$q_0 \to aq_1$

$q_1 \to bq_1$

$q_1 \to aq_2$

$q_2 \to bq_3$

176
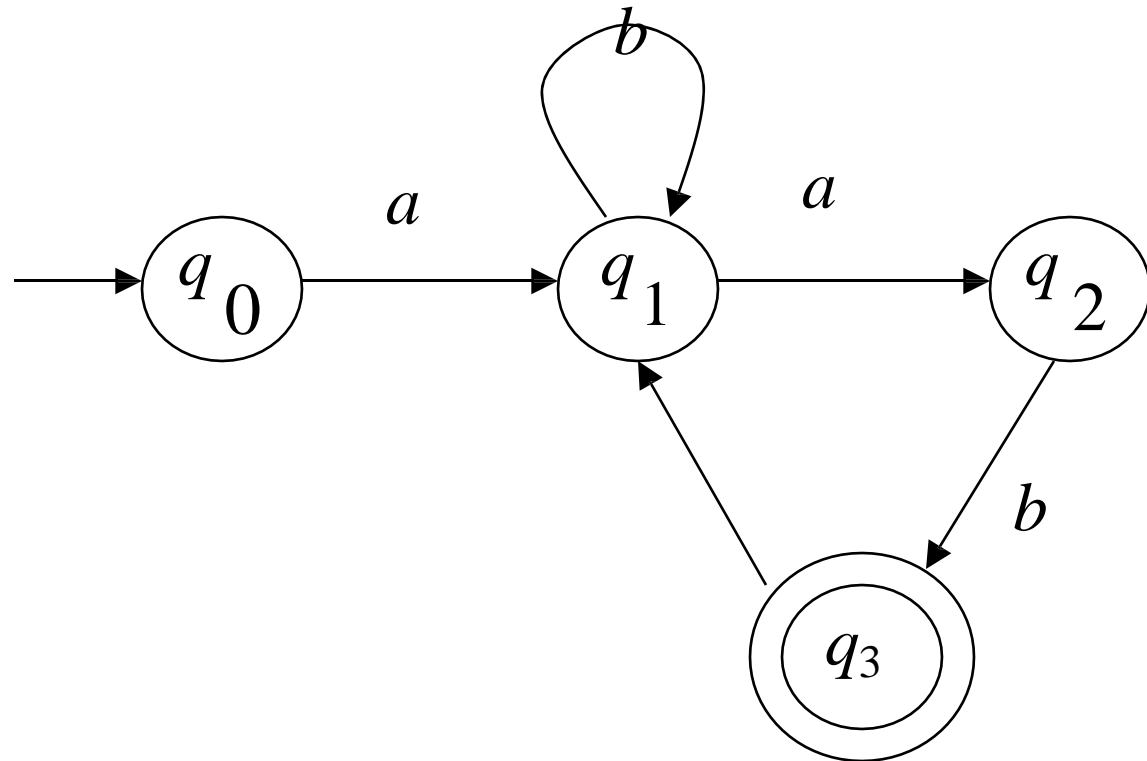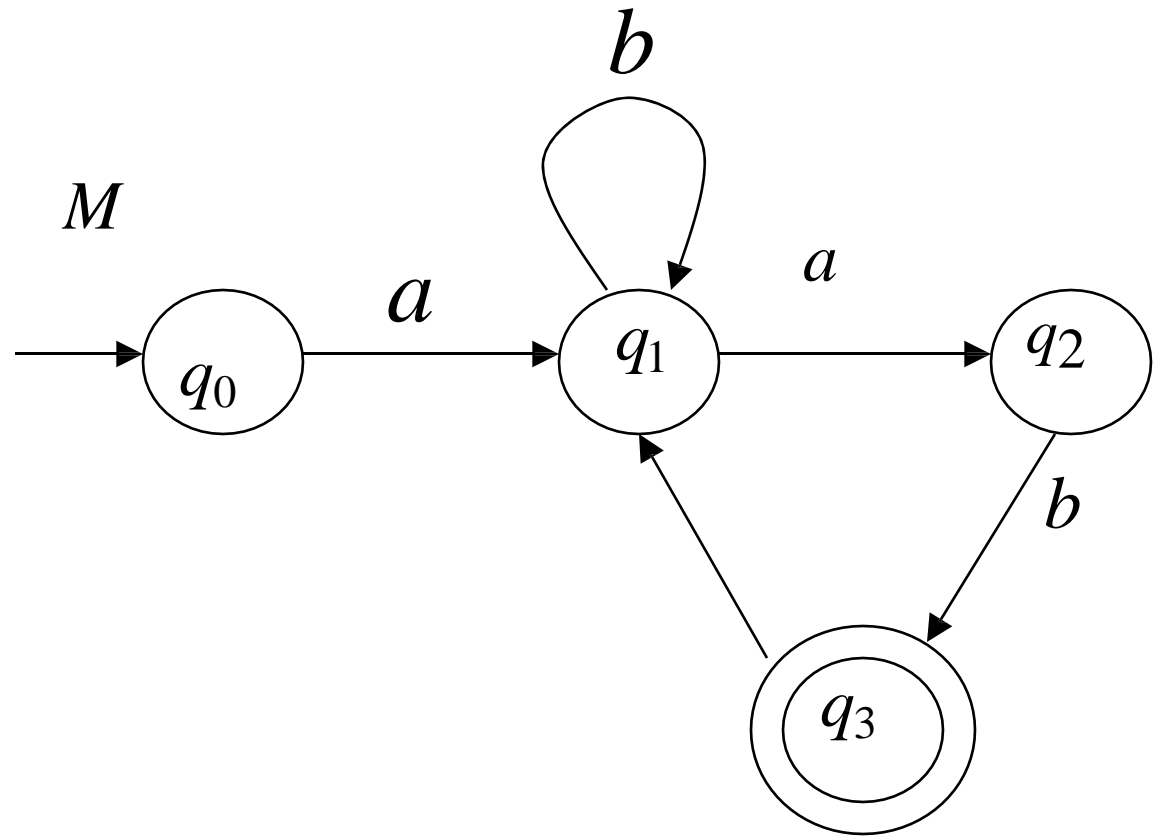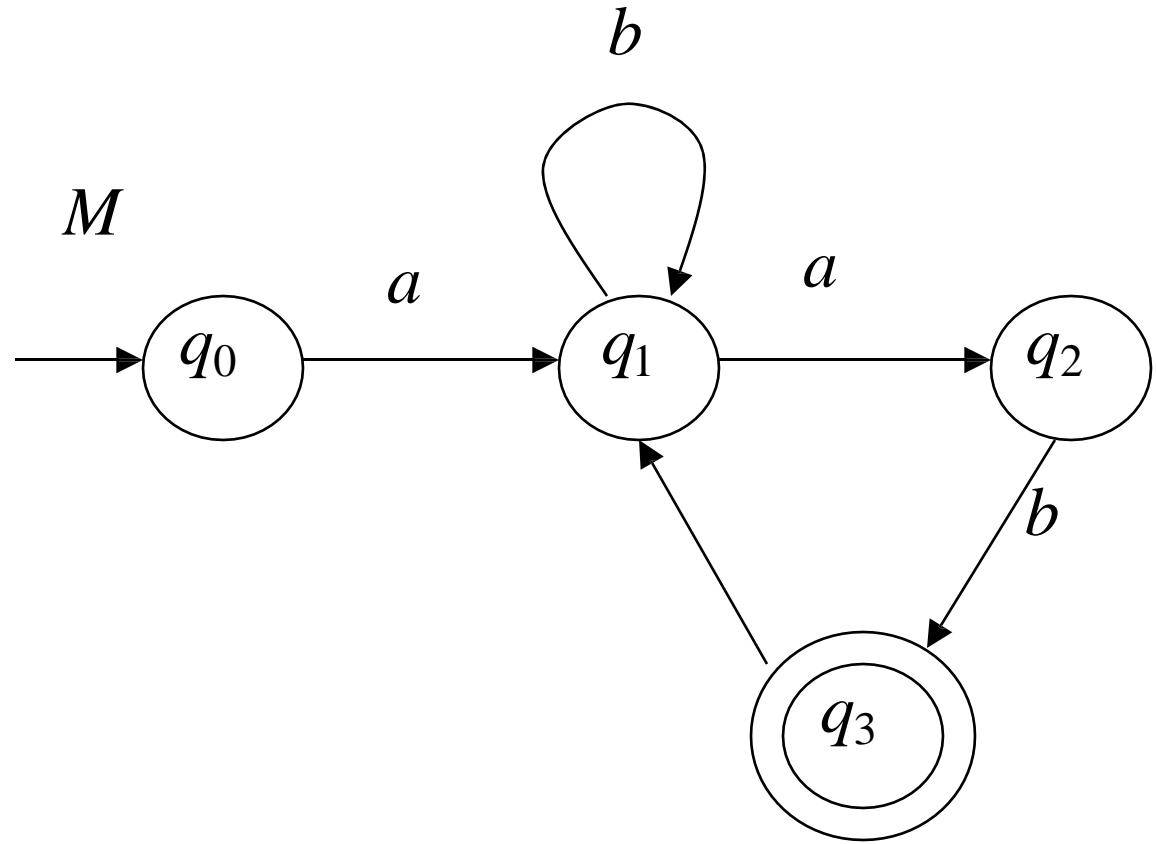
G

$q_0 \rightarrow aq_1$

$q_1 \rightarrow \quad bq_1$

$q_1 \rightarrow aq_2$

$q_2 \rightarrow bq_3$

$q_3 \rightarrow q_1$

$q_3 \rightarrow \lambda$

For any transition:

$$q \xrightarrow{a} p$$

Add production:

$$q \rightarrow ap$$

variable   terminal   variable
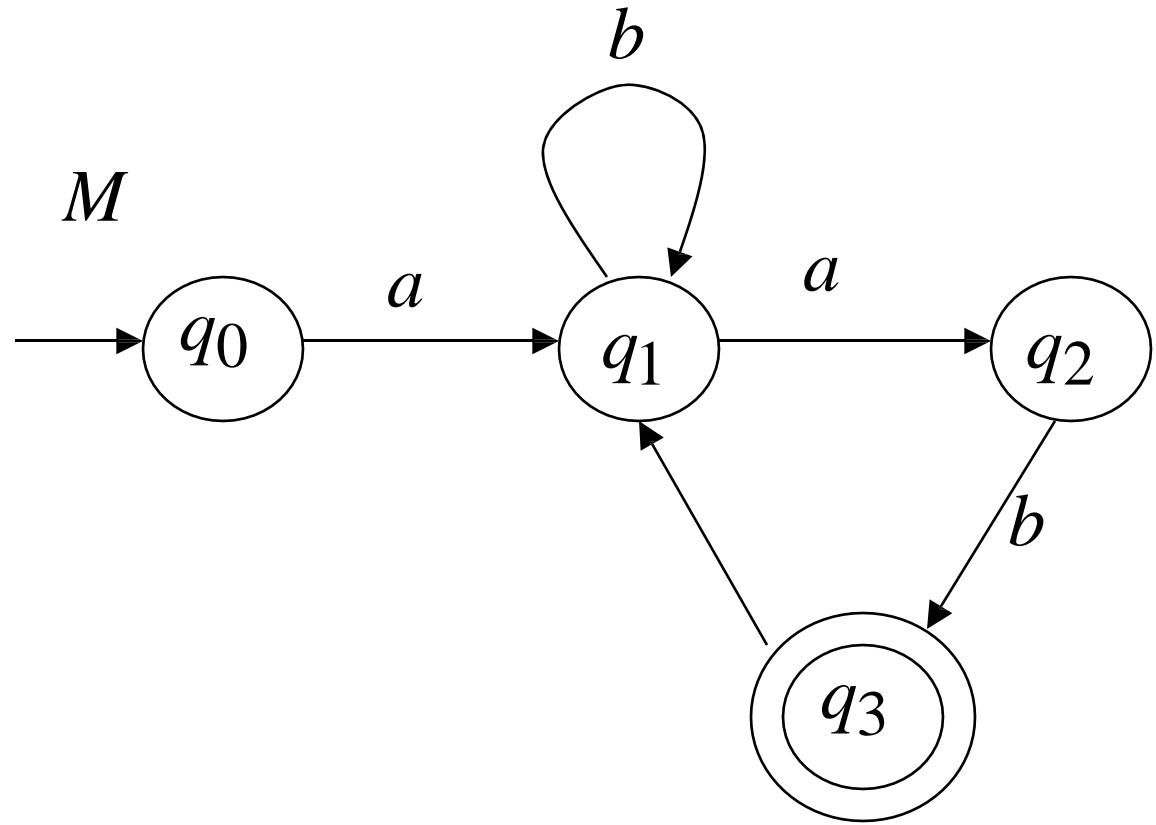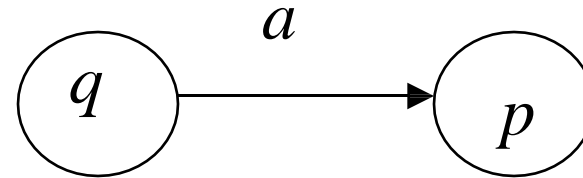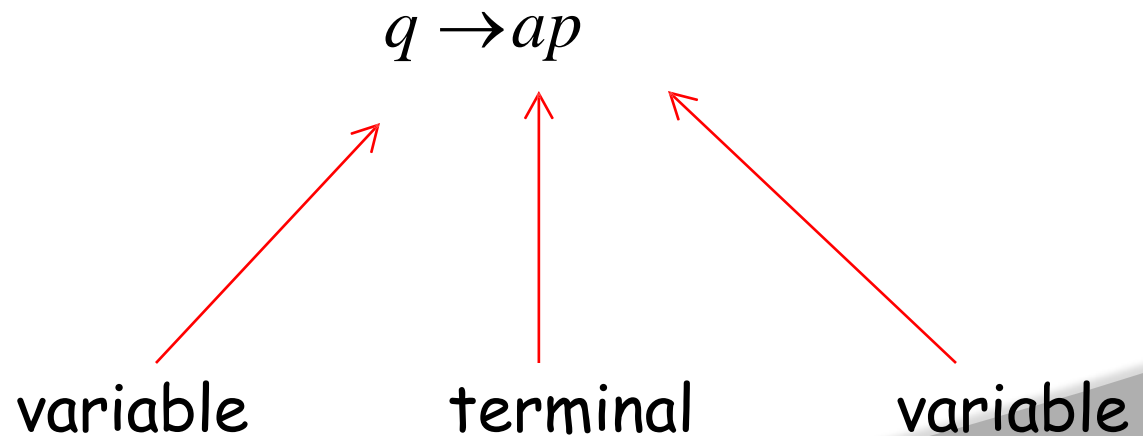
For any final state:

$$q_f$$
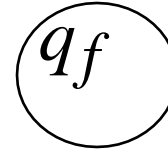
Add production:

$$q_f \rightarrow \lambda$$

- Since G is right-linear grammar

- G is also a regular grammar

$$L(G) = L(M) = L$$

Computer Science and Engineering Department
IV Semester

Theory of Computation
Unit- III

Syllabus:

Context free grammars and languages: Context free grammar, derivation trees, sentential forms, right most and leftmost derivation of strings, applications.

Ambiguity in context free grammars, minimization of context free grammars, Chomsky normal form, Greibach normal form, pumping lemma for context free languages, enumeration of properties of context free language (proofs omitted)..

- A *context-free grammar:* is a notation for describing languages.

- It is more powerful than finite automata or RE's, but still cannot define all possible languages.

- Useful for nested structures, e.g., parentheses in programming languages.

- Basic idea is to use "variables" to stand for sets of strings (i.e., languages).

- These variables are defined recursively, in terms of one another.

- Recursive rules ("productions") involve only concatenation.

- Alternative rules for a variable allow union.

Example: CFG for $\{\, 0^n 1^n \mid n \geq 1 \}$

- Productions:

  S -> 01 S -

  > 0S1

- **Basis**: 01 is in the language.

- **Induction**: if w is in the language, then so is 0w1.

CFG Formalism

- *Terminals*   = symbols of the alphabet of the language being defined.

- *Variables*   = *nonterminals*   = a finite set of other symbols, each of which represents a language.

- *Start symbol*   = the variable whose language is the one being defined.

Productions

- A *production* has the form variable -> string of variables and terminals.

- Convention:

  - A, B, C,… are variables.

  - a, b, c,… are terminals.

  - …, X, Y, Z are either terminals or variables.

  - …, w, x, y, z are strings of terminals only.

  - ▯, ▯, ▯,… are strings of terminals and/or variables.

Example: Formal CFG

- Here is a formal CFG for $\{ 0^n 1^n \mid n \geq 1 \}$.

- Terminals = $\{0, 1\}$.

- Variables = $\{S\}$.

- Start symbol = S.

- Productions =
  S -> 01   S -
  > 0S1

Derivations – Intuition

- We *derive* strings in the language of a CFG by starting with the start symbol, and repeatedly replacing some variable A by the right side of one of its productions.

  - That is, the "productions for A" are those that have A on the left side of the ->.

- We say αAβ => αγβ if A -> γ is a production.

- Example: S -> 01; S -> 0S1.

- S => 0S1 => 00S11 => 000111.

- =>* means "zero or more derivation steps."

- Basis: ⍰ =>* ⍰ for any string ⍰.

- Induction: if ⍰ =>* ⍰ and ⍰ => ⍰, then ⍰ =>* ⍰.

Example: Iterated Derivation

- S -> 01; S -> 0S1.

- S => 0S1 => 00S11 => 000111.

- So S =>* S; S =>* 0S1; S =>* 00S11; S =>* 000111.

- Any string of variables and/or terminals derived from the start symbol is called a *sentential form*.

- Formally, ⬚ is a sentential form iff   S =>* ⬚.

- If G is a CFG, then L(G), the *language of G*, is {w | S =>* w}.

  - Note: w must be a terminal string, S is the start symbol.
- Example: G has productions S -> ε and S -> 0S1.

- L(G) = {0^n1^n | n $\geq$ 0}.

Note: ε is a legitimate right side.

- A language that is defined by some CFG is called a *context-free language*.

- There are CFL's that are not regular languages, such as the example just given.

- But not all languages are CFL's.

- Intuitively: CFL's can count two things, not three.

BNF Notation

- Grammars for programming languages are often written in BNF (*Backus-Naur Form*).

- Variables are words in <…>; Example: <statement>.

- Terminals are often multicharacter strings indicated by boldface or underline; Example: **while** or WHILE.

- Symbol ::= is often used for ->.

- Symbol | is used for "or."

  - A shorthand for a list of productions with the same left side.

- Example: S -> 0S1 | 01 is shorthand for S -> 0S1 and S -> 01.

- Symbol … is used for "one or more."

- Example: <digit> ::= 0|1|2|3|4|5|6|7|8|9

<unsigned integer> ::= <digit>…

  - Note: that's not exactly the * of RE's.

- Translation: Replace ▯… with a new variable A and productions A -> A▯ | ▯.

Example: Kleene Closure

- Grammar for unsigned integers can be replaced by:

    U -> UD | D

    D -> 0|1|2|3|4|5|6|7|8|9

- Surround one or more symbols by [...] to make them optional.

- Example: <statement> ::= **if** <condition> **then** <statement> [; **else** <statement>]

- Translation: replace [α] by a new variable A with productions A -> α | ϵ.

Example: Optional Elements

- Grammar for if-then-else can be replaced by:

S -> iCtSA

A -> ;eS | ε

- Use {...} to surround a sequence of symbols that need to be treated as a unit.

  - Typically, they are followed by a ... for "one or more."

- Example: <statement list> ::= <statement> [{;<statement>}...]

Translation: Grouping

- You may, if you wish, create a new variable A for $\{\alpha\}$.

- One production for A: A -> α.

- Use A in place of $\{\alpha\}$.

Example: Grouping

<p style="color:red">L -> S [{;S}...]</p>

- Replace by L -> S [A...]   A -> ;S
  - A stands for {;S}.

- Then by L -> SB   B -> A... | ε   A -> ;S

  - B stands for [A...] (zero or more A's).

- Finally by L -> SB   B -> C | ε

  C -> AC | A   A -> ;S

  - C stands for A... .

- Derivations allow us to replace any of the variables in a string.

- Leads to many different derivations of the same string.

- By forcing the leftmost variable (or alternatively, the rightmost variable) to be replaced, we avoid these "distinctions without a difference."

- Say $wA\alpha \Rightarrow_{lm} w\beta\alpha$ if w is a string of terminals only and A -> β is a production.

- Also, $\alpha \Rightarrow^*_{lm} \beta$ if α becomes β by a sequence of 0 or more $\Rightarrow_{lm}$ steps.

- Balanced-parentheses grammmar: S -> SS | (S) | ()

- $S \Rightarrow_{lm} SS \Rightarrow_{lm} (S)S \Rightarrow_{lm} (())S \Rightarrow_{lm} (())()$

- Thus, $S \Rightarrow^*_{lm} (())()$

- $S \Rightarrow SS \Rightarrow S() \Rightarrow (S)() \Rightarrow (())()$ is a derivation, but not a leftmost derivation.

- Say $\alpha A w \Rightarrow_{rm} \alpha \beta w$ if w is a string of terminals only and $A \rightarrow \beta$ is a production.

- Also, $\alpha \Rightarrow^*_{rm} \beta$ if α becomes β by a sequence of 0 or more $\Rightarrow_{rm}$ steps.

- Balanced-parentheses grammmar: S -> SS | (S) | ()

- S $=>_{rm}$ SS $=>_{rm}$ S() $=>_{rm}$ (S)() $=>_{rm}$ (())()

- Thus, S $=>*_{rm}$ (())()

- S => SS => SSS => S()S => ()()S => ()()() is neither a rightmost nor a leftmost derivation.

- *Parse trees* are trees labeled by symbols of a particular CFG.

- Leaves: labeled by a terminal or ε.

- Interior nodes: labeled by a variable.

  - Children are labeled by the right side of a production for the parent.

- Root: must be labeled by the start symbol.

Example: Parse Tree

S -> SS | (S) | ()
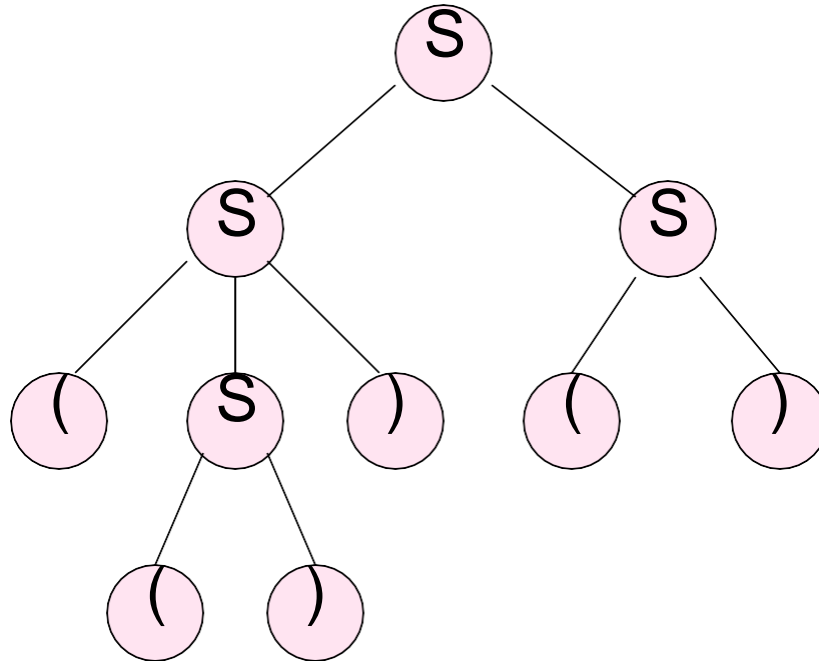
- The concatenation of the labels of the leaves in left-to-right order

  - That is, in the order of a preorder traversal.

  is called the *yield* of the parse tree.
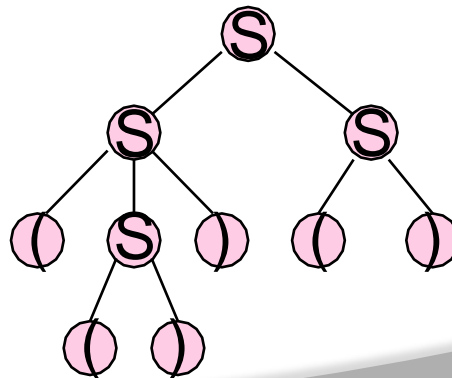
- Example: yield of is (())()

- For every parse tree, there is a unique leftmost, and a unique rightmost derivation.

- We'll prove:

    1. If there is a parse tree with root labeled A and yield w, then A =>*$_{lm}$ w.

    2. If A =>*$_{lm}$ w, then there is a parse tree with root A and yield w.

- Induction on the *height* (length of the longest path from the root) of the tree.

- Basis: height 1. Tree looks like

- $A \to a_1 \ldots a_n$ must be a production.

- Thus, $A \Rightarrow^*_{lm} a_1 \ldots a_n$.
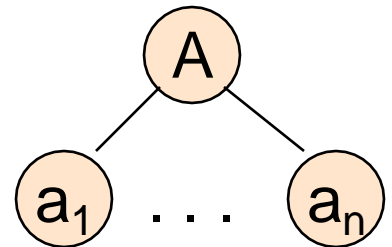
Part 1 – Induction

- Assume (1) for trees of height < h, and let this tree have height h:
- By IH, $X_i =>^*_{lm} w_i$.
    - Note: if X_i is a terminal, then $= w_i$.

- Thus, $A =>_{lm} X_1...X_n =>^*_{lm} w_1 X_2...X_n =>^*_{lm} w_1 w_2 X_3...X_n =>^*_{lm} ... =>^*_{lm} w_1...w_n$.
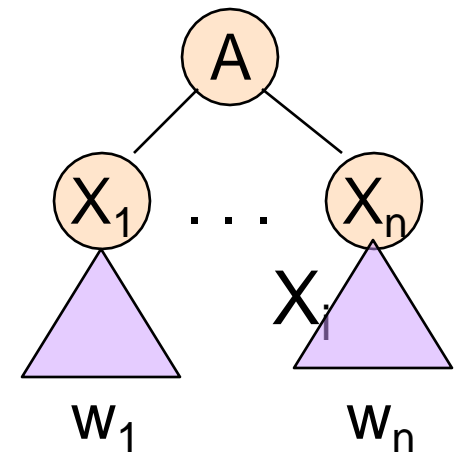
- Given a leftmost derivation of a terminal string, we need to prove the existence of a parse tree.

- The proof is an induction on the length of the derivation.

- If $A \Rightarrow^*_{lm} a_1 \ldots a_n$ by a one-step derivation, then there must be a parse tree
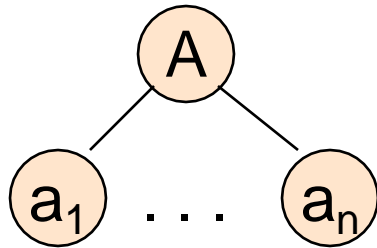
- Assume (2) for derivations of fewer than k > 1 steps, and let A =>*$_{lm}$ w be a k- step derivation.

- First step is A =>$_{lm}$ $X_1...X_n$.

- Key point: w can be divided so the first portion is derived from X_1, the next is derived from X_2, and so on.

  - If X_i is a terminal, then w_i =X_i.

Induction – (2)

- That is, $X_i =>^*_{lm} w_i$ for all i such that X_i is a variable.
  - And the derivation takes fewer than k steps.
- By the IH, if X_i is a variable, then there is a parse tree with root X_i and yield w_i.

- Thus, there is a parse tree

A

X_1 . . . X_n

w_1 220 w_n
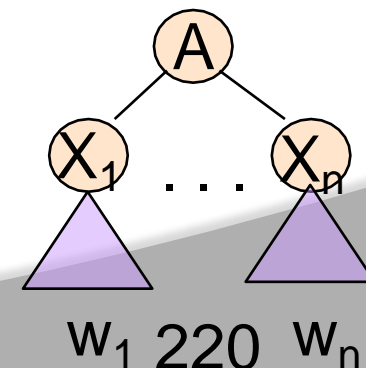
- The ideas are essentially the mirror image of the proof for leftmost derivations.

- Left to the imagination.

- The proof that you can obtain a parse tree from a leftmost derivation doesn't really depend on "leftmost."

- First step still has to be A => $X_1 \dots X_n$.

- And w still can be divided so the first portion is derived from X_1, the next is derived from X_2, and so on.

- A CFG is *ambiguous* if there is a string in the language that is the yield of two or more parse trees.

- Example: S -> SS | (S) | ()

- Two parse trees for ()()() on next slide.

Example – Continued
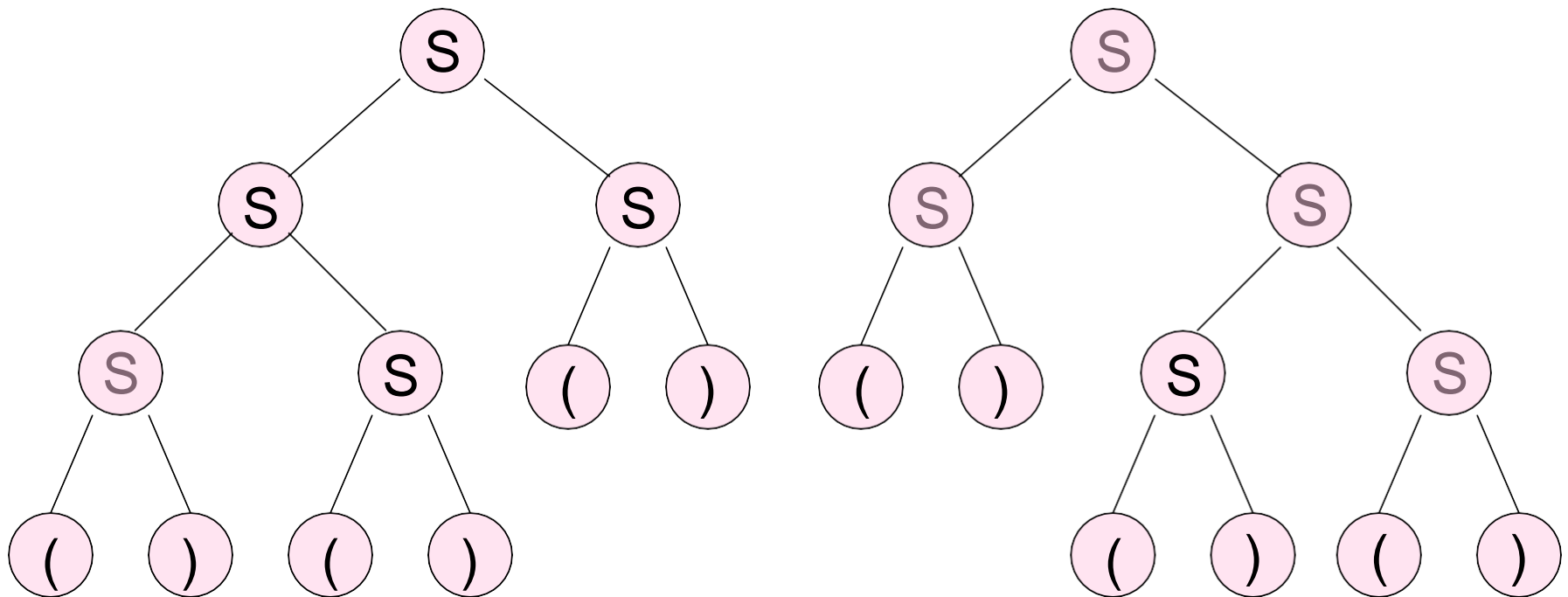
- If there are two different parse trees, they must produce two different leftmost derivations by the construction given in the proof.

- Conversely, two different leftmost derivations produce different parse trees by the other part of the proof.

- Likewise for rightmost derivations.

Ambiguity, etc. – (2)

- Thus, equivalent definitions of "ambiguous grammar'' are:

  1. There is a string in the language that has two different leftmost derivations.

  2. There is a string in the language that has two different rightmost derivations.

- For the balanced-parentheses language, here is another CFG, which is unambiguous.

B -> (RB | ε

R ->) | (RR

B, the start symbol, derives balanced strings.

R generates strings that have one more right paren than left.

Example: Unambiguous Grammar

B -> (RB | ε   R ->) | (RR

- Construct a unique leftmost derivation for a given balanced string of parentheses by scanning the string from left to right.

  - If we need to expand B, then use B -> (RB if the next symbol is "(" and ε if at the end.

  - If we need to expand R, use R ->) if the next symbol is ")" and (RR if it is "(".

Remaining Input: (())()

Steps of leftmost derivation:

B

Next symbol

B -> (RB | ϵ   R ->) | (RR

Remaining Input: ())()

Steps of leftmost
   derivation:

B

(RB

Next
symbol

B -> (RB | ε   R ->) | (RR

Remaining Input:

))()

Next
symbol

Steps of leftmost
derivation:

B

(RB

((RRB

B -> (RB | ε   R ->) | (RR

Remaining Input:

)()

Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

(()RB

B -> (RB | ε   R ->) | (RR

Remaining Input: ()

Steps of leftmost
derivation:

B

(RB

((RRB

(()RB

(())B

Next
symbol

B -> (RB | ε   R ->) | (RR

Remaining Input:

)

↑

Next
symbol

Steps of leftmost
derivation:

B   (())(RB

(RB

((RRB

(()RB

(())B

B -> (RB | ε   R ->) | (RR

Remaining Input:

Steps of leftmost derivation:

B

(RB

((RRB

(()RB

(())B

(())(RB

(())()B

↑

Next symbol

B -> (RB | ε   R ->) | (RR

Remaining Input:

Steps of leftmost derivation:

↑

Next symbol

B	(())(RB
(RB	(())()B
((RRB	(())()
(()RB	
(())B	

B -> (RB | ε   R ->) | (RR

- As an aside, a grammar such B -> (RB | ε R ->) | (RR, where you can always figure out the production to use in a leftmost derivation by scanning the given string left-to-right and looking only at the next one symbol is called LL(1).

  - "Leftmost derivation, left-to-right scan, one symbol of lookahead."

- Most programming languages have LL(1) grammars.

- LL(1) grammars are never ambiguous.

- It would be nice if for every ambiguous grammar, there were some way to "fix" the ambiguity, as we did for the balanced-parentheses grammar.

- Unfortunately, certain CFL's are *inherently ambiguous*, meaning that every grammar for the language is ambiguous.

Example: Inherent Ambiguity

- The language $\{0^i 1^j 2^k \mid i = j \text{ or } j = k\}$ is inherently ambiguous.

- Intuitively, at least some of the strings of the form $0^n 1^n 2^n$ must be generated by two different parse trees, one based on checking the 0's and 1's, the other based on checking the 1's and 2's.

S -> AB | CD

A -> 0A1 | 01 B -

> 2B | 2

C -> 0C | 0

D -> 1D2 | 12

A generates equal 0's and 1's

B generates any number of 2's

C generates any number of 0's

D generates equal 1's and 2's

And there are two derivations of every string with equal numbers of 0's, 1's, and 2's. E.g.:

S => AB => 01B =>012

S => CD => 0D => 012

241

Normal Forms for CFG's

Eliminating Useless Variables Removing Epsilon

Removing Unit Productions Chomsky Normal

Form

242

- Consider: S -> AB, A -> aA | a, B -> AB

- Although A derives all strings of a's, B derives no terminal strings (can you prove this fact?).

- Thus, S derives nothing, and the language is empty.

- **Basis**: If there is a production A -> w, where w has no variables, then A derives a terminal string.

- **Induction**: If there is a production A -> α, where α consists only of terminals and variables known to derive a terminal string, then A derives a terminal string.

Testing – (2)

- Eventually, we can find no more variables.

- An easy induction on the order in which variables are discovered shows that each one truly derives a terminal string.

- Conversely, any variable that derives a terminal string will be discovered by this algorithm.

- The proof is an induction on the height of the least-height parse tree by which a variable A derives a terminal string.

- Basis: Height = 1.   Tree looks like:

- Then the basis of the algorithm tells us that

    A will be discovered.

```
        A
       / \
     a_1 ... a_n
```
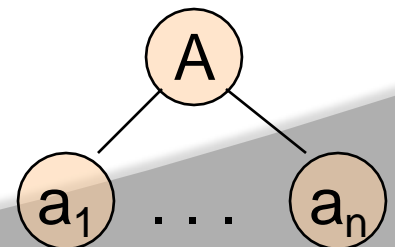
- Assume IH for parse trees of height < h, and suppose A derives a terminal string via a parse tree of height h:

- By IH, those X_i's that are

  variables are discovered.

- Thus, A will also be discovered,

because it has a right side of terminals and/or discovered variables.

A

X_1 $\cdots$ X_n

w_1 w_n
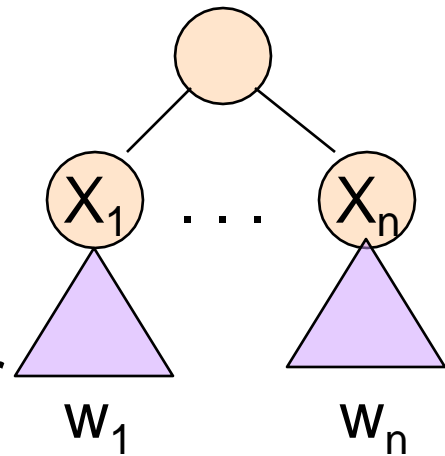
Algorithm to Eliminate Variables That Derive Nothing

1. Discover all variables that derive terminal strings.

2. For all other variables, remove all productions in which they appear either on the left or the right.

Example: Eliminate Variables

S -> AB | C, A -> aA | a, B -> bB, C -> c

- **Basis**: A and C are identified because of A -> a and C -> c.

- **Induction**: S is identified because of S -> C.

- Nothing else can be identified.

- **Result**: S -> C, A -> aA | a, C -> c

- Another way a terminal or variable deserves to be eliminated is if it cannot appear in any derivation from the start symbol.

- Basis: We can reach S (the start symbol).

- Induction: if we can reach A, and there is a production $A \rightarrow \alpha$, then we can reach all symbols of α.

- Easy inductions in both directions show that when we can discover no more symbols, then we have all and only the symbols that appear in derivations from S.

- Algorithm: Remove from the grammar all symbols not discovered reachable from S and all productions that involve these symbols.

- A symbol is *useful* if it appears in some derivation of some terminal string from the start symbol.

- Otherwise, it is *useless*. Eliminate all useless symbols by:

    1. Eliminate symbols that derive no terminal string.

    2. Eliminate unreachable symbols.

Example: Useless Symbols – (2)

$$S \to AB, A \to C, C \to c, B \to bB$$

- If we eliminated unreachable symbols first, we would find everything is reachable.

- A, C, and c would never get eliminated.

Why It Works

- After step (1), every symbol remaining derives some terminal string.

- After step (2) the only symbols remaining are all derivable from S.

- In addition, they still derive a terminal string, because such a derivation can only involve symbols reachable from S.

- We can almost avoid using productions of the form A -> ε (called *ε-productions*).
  - The problem is that ε cannot be in the language of any grammar that has no ε–productions.
- Theorem: If L is a CFL, then L-{ε} has a CFG with no ε-productions.

Nullable Symbols

- To eliminate ε-productions, we first need to discover the *nullable variables* = variables A such that A =>* ε.

- Basis: If there is a production A -> ε, then A is nullable.

- Induction: If there is a production A -> α, and all symbols of ▯ are nullable, then A is nullable.

Example: Nullable Symbols

S -> AB, A -> aA | ε, B -> bB | A

- **Basis**: A is nullable because of A -> ε.

- **Induction**: B is nullable because of   B -> A.

- Then, S is nullable because of S -> AB.

Proof of Nullable-Symbols Algorithm

- The proof that this algorithm finds all and only the nullable variables is very much like the proof that the algorithm for symbols that derive terminal strings works.

- Do you see the two directions of the proof?

- On what is each induction?

Eliminating ε-Productions

- **Key idea**: turn each production A -> $X_1 \ldots X_n$ into a family of productions.

- For each subset of nullable X's, there is one production with those eliminated from the right side "in advance."

  - Except, if all X's are nullable, do not make a production with ε as the right side.

S -> ABC, A -> aA | ε, B -> bB | ε, C -> ε

- A, B, C, and S are all nullable.

- New grammar:

S -> ~~ABC~~ | AB | AC | ~~BC~~ | A | B | C ~~~~

A -> aA | a

B -> bB | b

Note: C is now useless.
Eliminate its productions.

- Prove that for all variables A:

    1. If $w \neq \varepsilon$ and $A \Rightarrow^*_{old} w$, then $A \Rightarrow^*_{new} w$.

    2. If $A \Rightarrow^*_{new} w$ then $w \neq \varepsilon$ and $A \Rightarrow^*_{old} w$.

- Then, letting A be the start symbol proves that $L(new) = L(old) - \{\varepsilon\}$.

- (1) is an induction on the number of steps by which A derives w in the old grammar.

Proof of 1 − Basis

- If the old derivation is one step, then   A -> w must be a production.
- Since w ≠ ϵ, this production also appears in the new grammar.
- Thus, A =>$_{new}$ w.

Proof of 1 – Induction

- Let $A \Rightarrow^*_{old} w$ be an n-step derivation, and assume the IH for derivations of less than n steps.

- Let the first step be $A \Rightarrow_{old} X_1 \ldots X_n$.

- Then w can be broken into $w = w_1 \ldots w_n$,

- where $X_i \Rightarrow^*_{old} w_i$, for all i, in fewer than n steps.

Induction – Continued

- By the IH, if $w_i \neq \epsilon$, then $X_i \Rightarrow^*_{new} w_i$.

- Also, the new grammar has a production with A on the left, and just those X_i's on the right such that $w_i \neq \epsilon$.

    - Note: they all can't be ϵ, because $w \neq \epsilon$.

- Follow a use of this production by the derivations $X_i \Rightarrow^*_{new} w_i$ to show that A derives w in the new grammar.

Proof of Converse

- We also need to show part (2) – if w is derived from A in the new grammar, then it is also derived in the old.

- Induction on number of steps in the derivation.

- We'll leave the proof for reading in the text.

Unit Productions

- A *unit production* is one whose right side consists of exactly one variable.
- These productions can be eliminated.
- Key idea: If A =>* B by a series of unit productions, and B -> α is a non-unit-production, then add production A -> α.
- Then, drop all unit productions.

Unit Productions – (2)

- Find all pairs (A, B) such that A =>* B by a sequence of unit productions only.
- Basis: Surely (A, A).
- Induction: If we have found (A, B), and B -> C is a unit production, then add (A, C).

Proof That We Find Exactly the Right Pairs

- By induction on the order in which pairs (A, B) are found, we can show A =>* B by unit productions.

- Conversely, by induction on the number of steps in the derivation by unit productions of A =>* B, we can show that the pair (A, B) is discovered.

- **Basic idea**: there is a leftmost derivation A $=>^*_{lm}$ w in the new grammar if and only if there is such a derivation in the old.

- A sequence of unit productions and a non-unit production is collapsed into a single production of the new grammar.

Cleaning Up a Grammar

- **Theorem**: if L is a CFL, then there is a CFG for L − {ε} that has:

    1. No useless symbols.
    2. No ε-productions.
    3. No unit productions.

- I.e., every right side is either a single terminal or has length ≥ 2.

Cleaning Up – (2)

- **Proof**: Start with a CFG for L.
- Perform the following steps in order:
  1. Eliminate є-productions.
  2. Eliminate unit productions.
  3. Eliminate variables that derive no terminal string.
  4. Eliminate variables not reached from the start symbol.

Must be first. Can create unit productions or useless variables.

271

Chomsky Normal Form

- A CFG is said to be in *Chomsky Normal Form* if every production is of one of these two forms:
    1. A -> BC (right side is two variables).
    2. A -> a (right side is a single terminal).
- Theorem: If L is a CFL, then L – {ϵ} has a CFG in CNF.

Proof of CNF Theorem

- Step 1: "Clean" the grammar, so every production right side is either a single terminal or of length at least 2.

- Step 2: For each right side $\neq$ a single terminal, make the right side all variables.

  - For each terminal a create new variable A_a and production $A_a \rightarrow a$.

  - Replace a by A_a in right sides of length > 2.

Example: Step 2

- Consider production A -> BcDe.
- We need variables A_c and A_e. with productions A_c -> c and A_e -> e.
  - Note: you create at most one variable for each terminal, and use it everywhere it is needed.
- Replace A -> BcDe by A -> BA_cDA_e.

CNF Proof – Continued

- Step 3: Break right sides longer than 2 into a chain of productions with right sides of two variables.

- Example: A -> BCDE is replaced by A -> BF, F -> CG, and G -> DE.
  - F and G must be used nowhere else.

Example of Step 3 – Continued

- Recall A -> BCDE is replaced by A -> BF, F -> CG, and G -> DE.

- In the new grammar, A => BF => BCG
=> BCDE.

- More importantly: Once we choose to replace A by BF, we must continue to BCG and BCDE.

  - Because F and G have only one

    production.

CNF Proof – Concluded

- We must prove that Steps 2 and 3 produce new grammars whose languages are the same as the previous grammar.

- Proofs are of a familiar type and involve inductions on the lengths of derivations.

The Pumping Lemma for CFL's

Statement

Applications

Intuition

- Recall the pumping lemma for regular languages.
- It told us that if there was a string long enough to cause a cycle in the DFA for the language, then we could "pump" the cycle and discover an infinite sequence of strings that had to be in the language.

Intuition – (2)

- For CFL's the situation is a little more complicated.
- We can always find two pieces of any sufficiently long string to "pump" in tandem.
  - That is: if we repeat each of the two pieces the same number of times, we get another string in the language.

For every context-free language L

There is an integer n, such that

For every string z in L of length $\geq$ n

There exists z = uvwxy such that:

1. $|vwx| \leq n$.
2. $|vx| > 0$.
3. For all $i \geq 0$, $uv^i wx^i y$ is in L.

Proof of the Pumping Lemma

- Start with a CNF grammar for L – {ϵ}.

- Let the grammar have m variables.

- Pick n = 2^m.

- Let |z| $\geq$ n.

- We claim ("*Lemma 1*") that a parse tree with yield z must have a path of length m+2 or more.

Proof of Lemma 1

■ If all paths in the parse tree of a CNF grammar are of length $\leq$ m+1, then the longest yield has length 2^{m-1}, as in:

m variables

one terminal

2^{m-1} terminals

283

- Now we know that the parse tree for z has a path with at least m+1 variables.

- Consider some longest path.

- There are only m different variables, so among the lowest m+1 we can find two nodes with the same label, say A.

- The parse tree thus looks like:

Can't both be ϵ.

$\leq 2^m = n$ because a longest path chosen

A

A

u   v   _w   x _   y
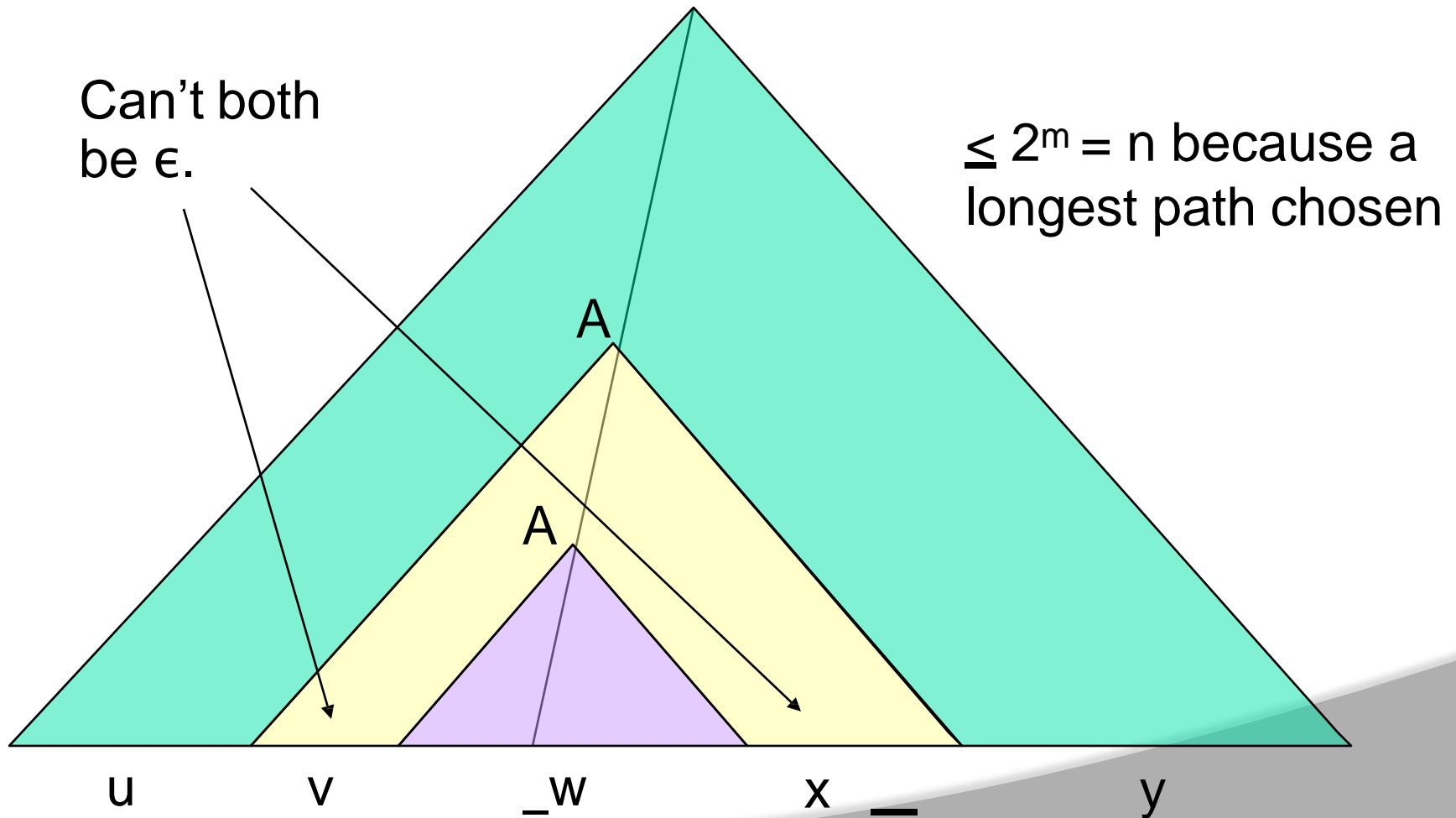
Pump Zero Times
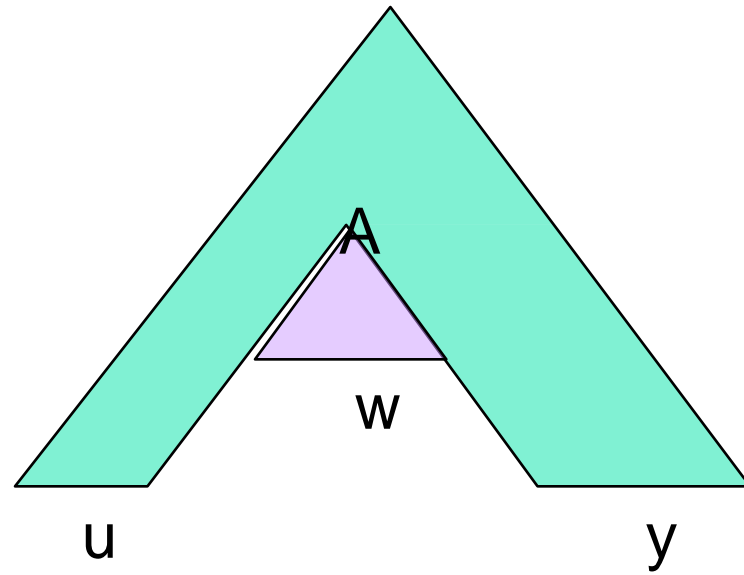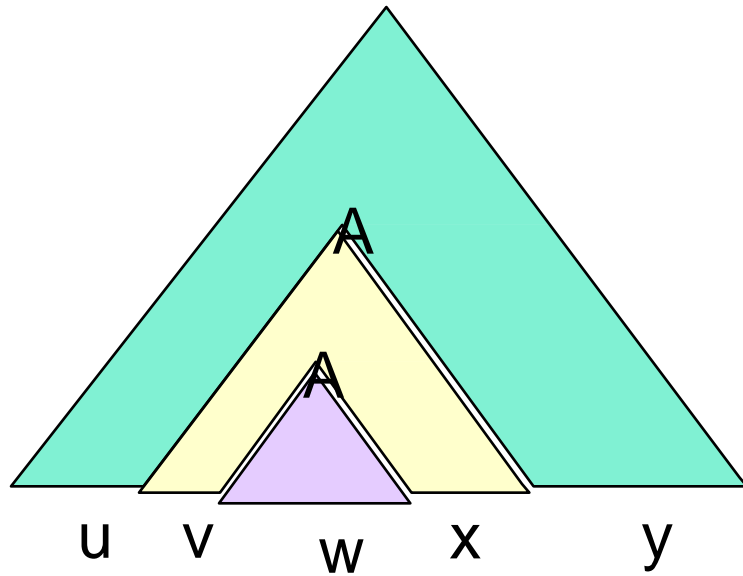
Pump Twice
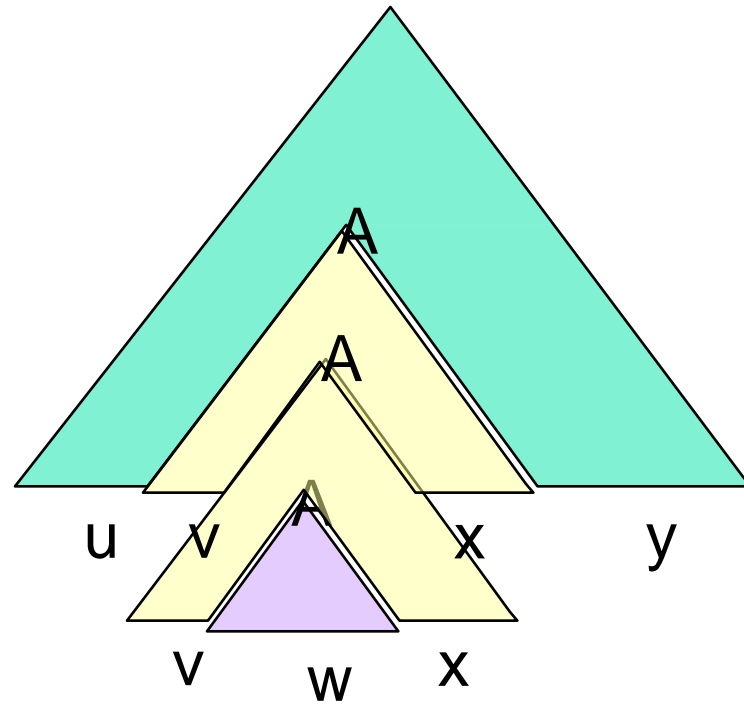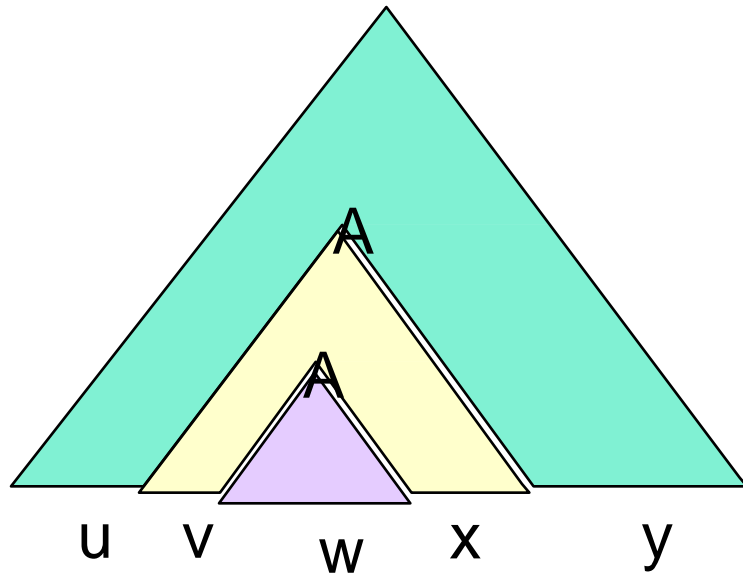
Pump Thrice
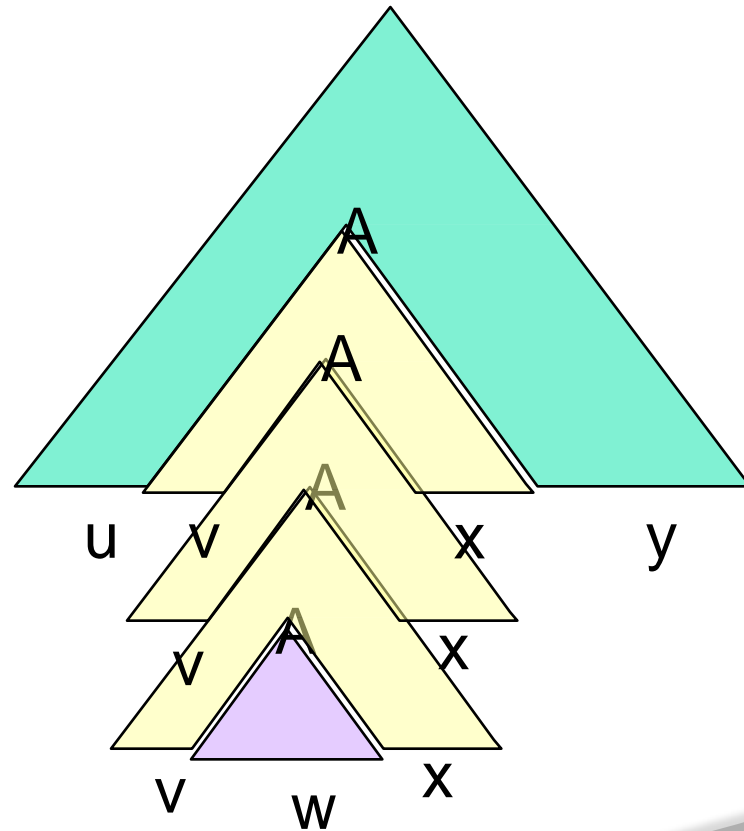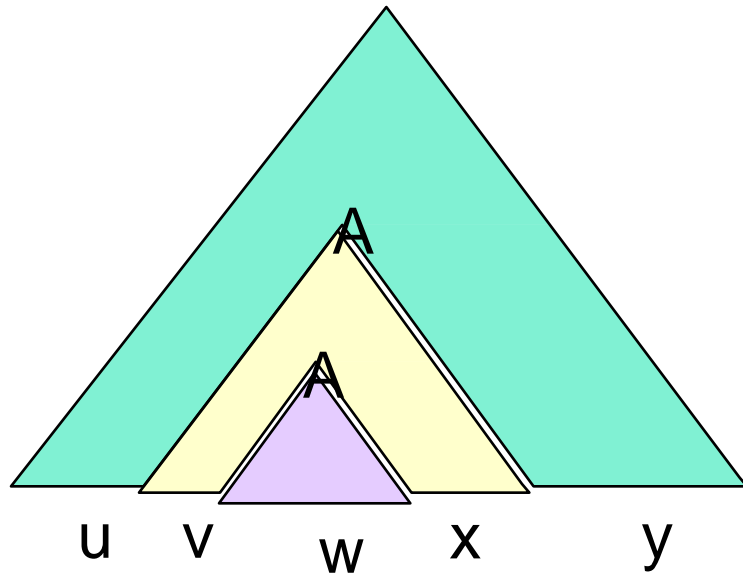
Using the Pumping Lemma

- Non-CFL's typically involve trying to match two pairs of counts or match two strings.

- Example: The text uses the pumping lemma to show that {ww | w in (**0+1**)*} is not a CFL.

Using the Pumping Lemma – (2)

- $\{0^i 10^i \mid i \geq 1\}$ is a CFL.
  - We can match one pair of counts.
- But $L = \{0^i 10^i 10^i \mid i \geq 1\}$ is not.
  - We can't match two pairs, or three counts as a group.
- Proof using the pumping lemma.
- Suppose L were a CFL.
- Let n be L's pumping-lemma constant.

Using the Pumping Lemma – (3)

- Consider $z = 0^n 1 0^n 1 0^n$.

- We can write $z = uvwxy$, where $|vwx| \leq n$, and $|vx| \geq 1$.

- Case 1: vx has no 0's.
  - Then at least one of them is a 1, and uwy has at most one 1, which no string in L does.

Using the Pumping Lemma – (4)

- Still considering $z = 0^n10^n10^n$.
- Case 2: vx has at least one 0.
  - vwx is too short (length $\leq$ n) to extend to all three blocks of 0's in $0^n10^n10^n$.
  - Thus, uwy has at least one block of n 0's, and at least one block with fewer than n 0's.
  - Thus, uwy is not in L.

Properties of Context-Free Languages

Decision Properties Closure Properties

Summary of Decision Properties

- As usual, when we talk about "a CFL" we really mean "a representation for the CFL, e.g., a CFG or a PDA accepting by final state or empty stack.

- There are algorithms to decide if:

  1. String w is in CFL L.
  2. CFL L is empty.
  3. CFL L is infinite.

Non-Decision Properties

- Many questions that can be decided for regular sets cannot be decided for CFL's.
- Example: Are two CFL's the same?
- Example: Are two CFL's disjoint?
  - How would you do that for regular languages?
- Need theory of Turing machines and decidability to prove no algorithm exists.

Testing Emptiness

- We already did this.
- We learned to eliminate variables that generate no terminal string.
- If the start symbol is one of these, then the CFL is empty; otherwise not.

Testing Membership

- Want to know if string w is in L(G).
- Assume G is in CNF.
  - Or convert the given grammar to CNF.
  - w = ε is a special case, solved by testing if the start symbol is nullable.
- Algorithm (*CYK*) is a good example of dynamic programming and runs in time $O(n^3)$, where n = |w|.

CYK Algorithm

- Let $w = a_1 \ldots a_n$.

- We construct an n-by-n triangular array of sets of variables.

- $X_{ij} = \{\text{variables } A \mid A \Rightarrow^* a_i \ldots a_j\}$.

- Induction on $j - i + 1$.

  - The length of the derived string.

- Finally, ask if S is in X_{1n}.

CYK Algorithm – (2)

- **Basis**: $X_{ii} = \{A \mid A \to a_i$ is a production$\}$.

- **Induction**: $X_{ij} = \{A \mid$ there is a production $A \to BC$ and an integer k, with $i \leq k < j$, such that B is in X_{ik} and C is in $X_{k+1,j}$.

Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b
String w = ababa

$X_{12}=\{B,S\}$ $X_{23}=\{A\}$ $X_{34}=\{B,S\}$ $X_{45}=\{A\}$

$X_{11}=\{A,C\}$ $X_{22}=\{B,C\}$ $X_{33}=\{A,C\}$ $X_{44}=\{B,C\}$ $X_{55}=\{A,C\}$
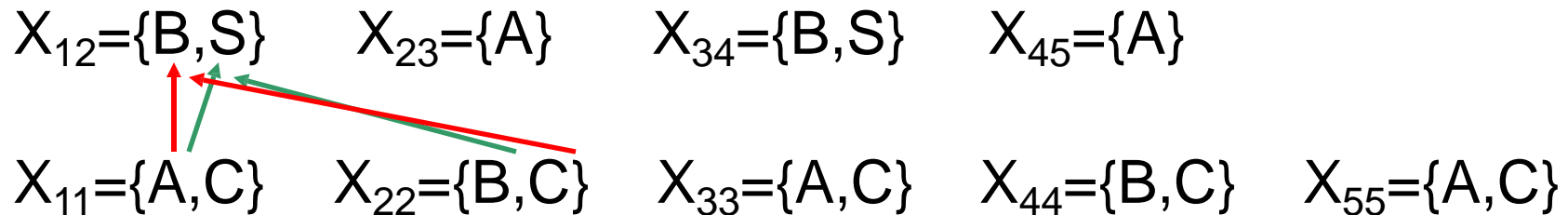
Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b
String w = ababa

$X_{13} = \{\}$
Yields nothing

$X_{12} = \{B,S\}$ $X_{23} = \{A\}$ $X_{34} = \{B,S\}$ $X_{45} = \{A\}$

$X_{11} = \{A,C\}$ $X_{22} = \{B,C\}$ $X_{33} = \{A,C\}$ $X_{44} = \{B,C\}$ $X_{55} = \{A,C\}$

Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b
String w = ababa

$X_{13}=\{A\}$ $X_{24}=\{B,S\}$ $X_{35}=\{A\}$

$X_{12}=\{B,S\}$ $X_{23}=\{A\}$ $X_{34}=\{B,S\}$ $X_{45}=\{A\}$

$X_{11}=\{A,C\}$ $X_{22}=\{B,C\}$ $X_{33}=\{A,C\}$ $X_{44}=\{B,C\}$ $X_{55}=\{A,C\}$

Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b
String w = ababa
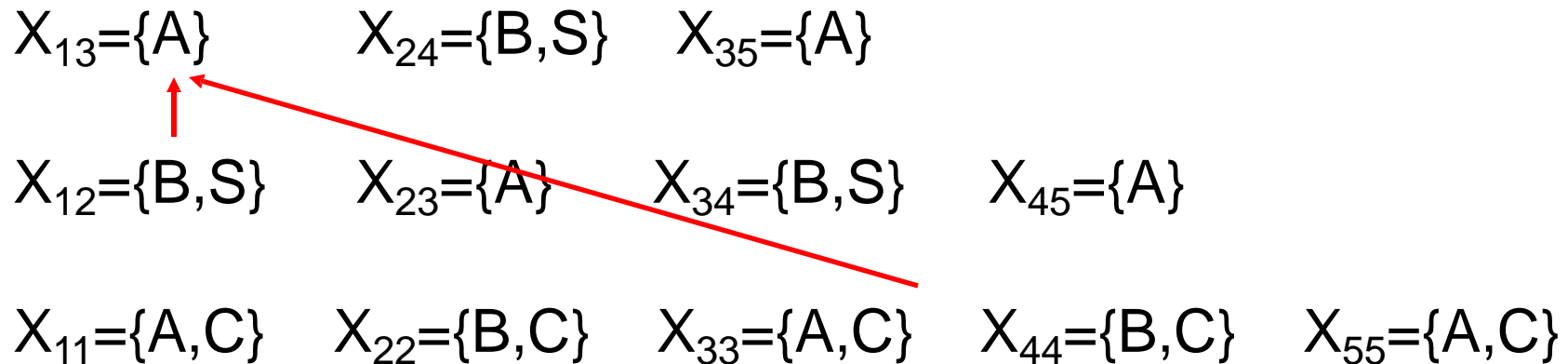
$X_{14} = \{B,S\}$

$X_{13} = \{A\}$ $X_{24} = \{B,S\}$ $X_{35} = \{A\}$

$X_{12} = \{B,S\}$ $X_{23} = \{A\}$ $X_{34} = \{B,S\}$ $X_{45} = \{A\}$

$X_{11} = \{A,C\}$ $X_{22} = \{B,C\}$ $X_{33} = \{A,C\}$ $X_{44} = \{B,C\}$ $X_{55} = \{A,C\}$
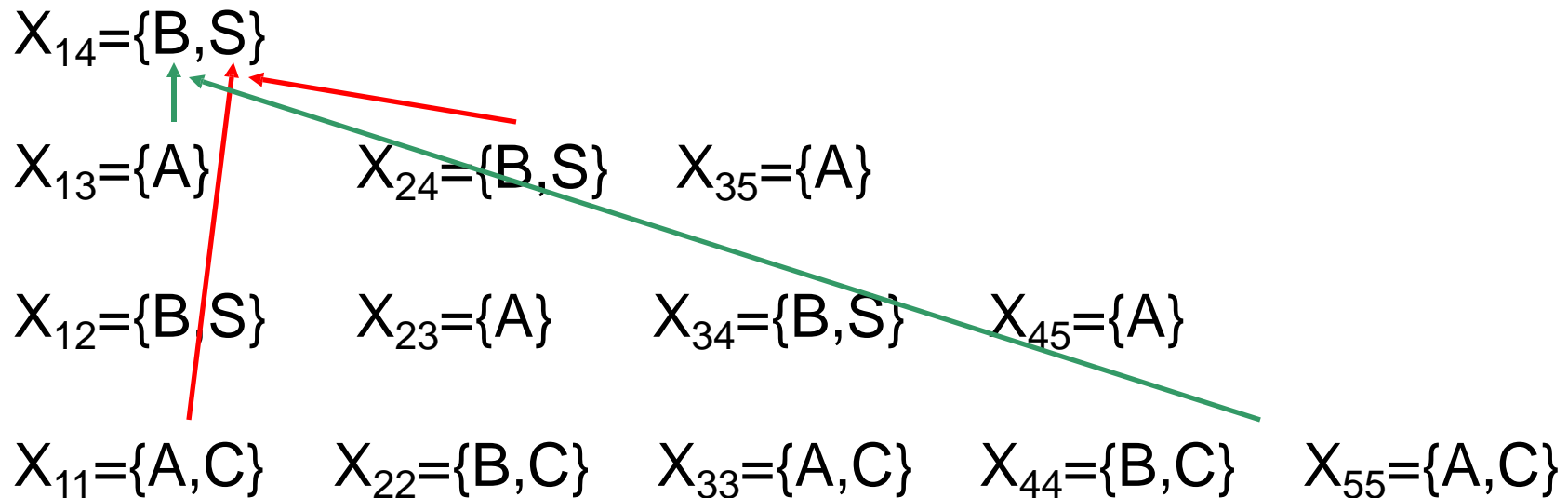
Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b

String w = ababa

$X_{15}=\{A\}$

$X_{14}=\{B,S\}$ $X_{25}=\{A\}$

$X_{13}=\{A\}$ $X_{24}=\{B,S\}$ $X_{35}=\{A\}$

$X_{12}=\{B,S\}$ $X_{23}=\{A\}$ $X_{34}=\{B,S\}$ $X_{45}=\{A\}$

$X_{11}=\{A,C\}$ $X_{22}=\{B,C\}$ $X_{33}=\{A,C\}$ $X_{44}=\{B,C\}$ $X_{55}=\{A,C\}$
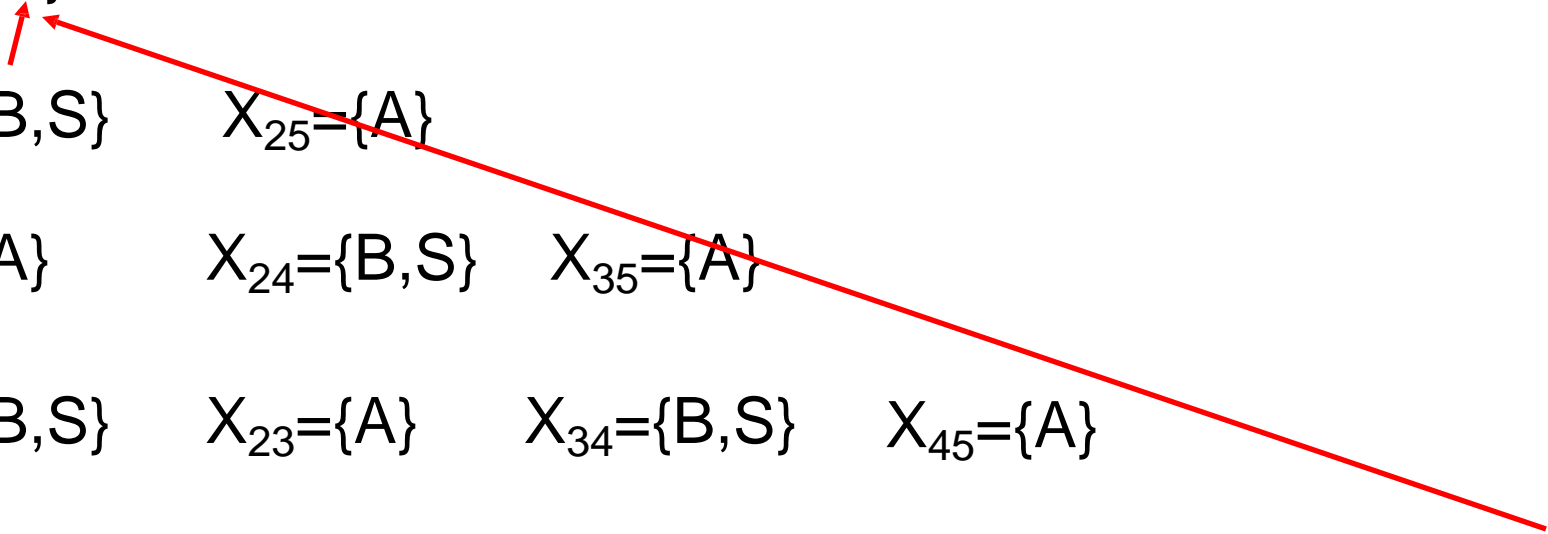
Testing Infiniteness

- The idea is essentially the same as for regular languages.
- Use the pumping lemma constant n.
- If there is a string in the language of length between n and 2n-1, then the language is infinite; otherwise not.
- Let's work this out in class.

Closure Properties of CFL's

- CFL's are closed under union, concatenation, and Kleene closure.
- Also, under reversal, homomorphisms and inverse homomorphisms.
- But not under intersection or difference.

Closure of CFL's Under Union

- Let L and M be CFL's with grammars G and H, respectively.
- Assume G and H have no variables in common.
  - Names of variables do not affect the language.
- Let S_1 and S_2 be the start symbols of G and H.

Closure Under Union – (2)

- Form a new grammar for $L \cup M$ by combining all the symbols and productions of G and H.

- Then, add a new start symbol S.

- Add productions $S \rightarrow S_1 \mid S_2$.

Closure Under Union – (3)

- In the new grammar, all derivations start with S.

- The first step replaces S by either S_1 or S_2.

- In the first case, the result must be a string in $L(G) = L$, and in the second case a string in $L(H) = M$.

Closure of CFL's Under Concatenation

- Let L and M be CFL's with grammars G and H, respectively.

- Assume G and H have no variables in common.

- Let S_1 and S_2 be the start symbols of G and H.

Closure Under Concatenation – (2)

- Form a new grammar for LM by starting with all symbols and productions of G and H.

- Add a new start symbol S.

- Add production $S \rightarrow S_1 S_2$.

- Every derivation from S results in a string in L followed by one in M.

Closure Under Star

- Let L have grammar G, with start symbol S_1.

- Form a new grammar for L* by introducing to G a new start symbol S and the productions $S \to S_1 S \mid \epsilon$.

- A rightmost derivation from S generates a sequence of zero or more S_1's, each of which generates some string in L.

Closure of CFL's Under Reversal

- If L is a CFL with grammar G, form a grammar for L^R by reversing the right side of every production.

- Example: Let G have S -> 0S1 | 01.

- The reversal of L(G) has grammar S -> 1S0 | 10.

Closure of CFL's Under Homomorphism

- Let L be a CFL with grammar G.
- Let h be a homomorphism on the terminal symbols of G.
- Construct a grammar for h(L) by replacing each terminal symbol *a* by h(a).

Example:

Closure Under Homomorphism

- G has productions S -> 0S1 | 01.
- h is defined by h(0) = ab, h(1) = ε.
- h(L(G)) has the grammar with productions S -> abS | ab.

Closure of CFL's Under Inverse Homomorphism

- Here, grammars don't help us.
- But a PDA construction serves nicely.
- Intuition: Let L = L(P) for some PDA P.
- Construct PDA P' to accept $h^{-1}(L)$.
- P' simulates P, but keeps, as one component of a two-component state a buffer that holds the result of applying h to one input symbol.

Architecture of P'

Input: 0 0 1 1
h(0)
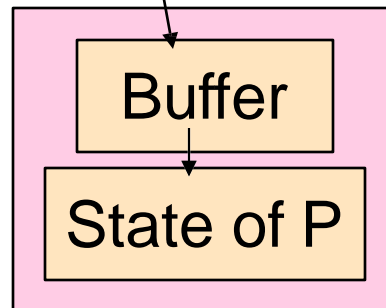
Buffer

State of P

Read first remaining
symbol in buffer as
if it were input to P.

Stack
of P

Formal Construction of P'

- ## States are pairs [q, b], where:
    1. q is a state of P.
    2. b is a suffix of h(a) for some symbol *a*.
        - Thus, only a finite number of possible values for b.

- ## Stack symbols of P' are those of P.
- ## Start state of P' is [q_0 ,ϵ].

Construction of P' – (2)

- Input symbols of P' are the symbols to which h applies.
- Final states of P' are the states [q, ϵ] such that q is a final state of P.

Transitions of P'

1. δ'([q, ϵ], a, X) = {([q, h(a)], X)} for any input symbol *a* of P' and any stack symbol X.

    ◆ When the buffer is empty, P' can reload it.

2. δ'([q, bw], ϵ, X) contains ([p, w], α) if δ(q, b, X) contains (p, α), where b is either an input symbol of P or ϵ.

    ◆ Simulate P from the buffer.

Proving Correctness of P'

- We need to show that $L(P') = h^{-1}(L(P))$.

- Key argument: P' makes the transition $([q_0, \epsilon], w, Z_0) \vdash^* ([q, x], \epsilon, \alpha)$ if and only if P makes transition $(q_0, y, Z_0) \vdash^* (q, \epsilon, \alpha)$, $h(w) = yx$, and x is a suffix of the last symbol of w.

- Proof in both directions is an induction on the number of moves made.

Nonclosure Under Intersection

- Unlike the regular languages, the class of CFL's is not closed under $\cap$.

- We know that $L_1 = \{0^n 1^n 2^n \mid n \geq 1\}$ is not a CFL (use the pumping lemma).

- However, $L_2 = \{0^n 1^n 2^i \mid n \geq 1, i \geq 1\}$ is.
  - CFG: S -> AB, A -> 0A1 | 01, B -> 2B | 2.

- So is $L_3 = \{0^i 1^n 2^n \mid n \geq 1, i \geq 1\}$.

- But $L_1 = L_2 \cap L_3$.

Nonclosure Under Difference

- We can prove something more general:
    - Any class of languages that is closed under difference is closed under intersection.

- Proof: $L \cap M = L - (L - M)$.

- Thus, if CFL's were closed under difference, they would be closed under intersection, but they are not.

- Intersection of two CFL's need not be context free.

- But the intersection of a CFL with a regular language is always a CFL.

- Proof involves running a DFA in parallel with a PDA, and noting that the combination is a PDA.

  - PDA's accept by final state.

DFA and PDA in Parallel

DFA

Input

PDA

Accept
if both
accept

Stack

Looks like the
state of one PDA
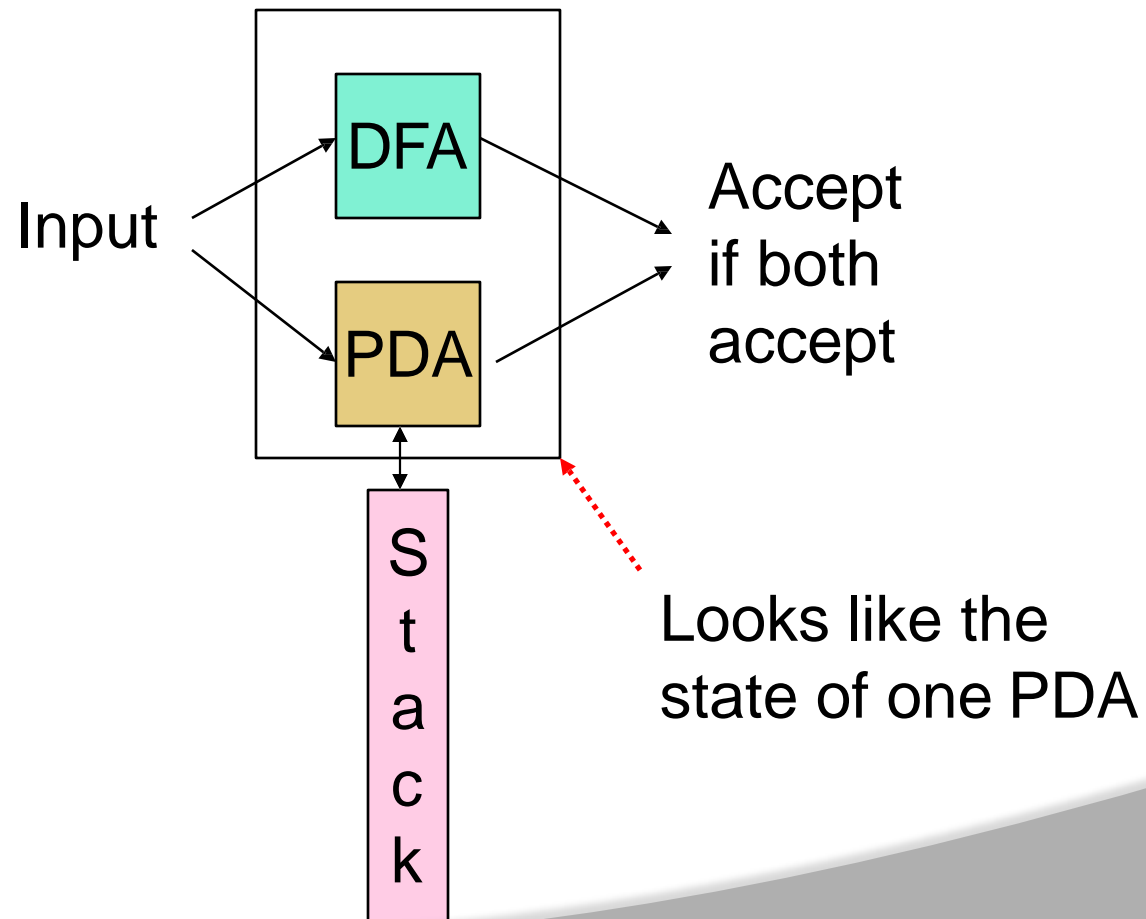
Formal Construction

- Let the DFA A have transition function δ_A.
- Let the PDA P have transition function δ_P.

- States of combined PDA are [q,p], where q is a state of A and p a state of P.
- $\delta([q,p], a, X)$ contains $([\delta_A(q,a),r], \alpha)$ if $\delta_P(p, a, X)$ contains (r, α).
  - Note a could be ε, in which case $\delta_A(q,a) = q$.

Formal Construction – (2)

- Accepting states of combined PDA are those [q,p] such that q is an accepting state of A and p is an accepting state of P.

- Easy induction: $([q_0,p_0], w, Z_0) \vdash^* ([q,p], \varepsilon, \alpha)$ if and only if $\delta_A(q_0,w) = q$ and in P: $(p_0, w, Z_0) \vdash^* (p, \varepsilon, \alpha)$.

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad - 500 043

Computer Science and Engineering Department
IV Semester

Theory of Computation
Unit- IV

Unit – IV

Syllabus:

Pushdown automata, definition, model, acceptance of context free language, acceptance by final state and acceptance by empty stack and its equivalence, equivalence of context free language and pushdown automata, inter conversion;(Proofs not required);Introduction to deterministic context free languages and deterministic pushdown automata

Hierarchy of languages

Regular Languages ➜ Finite State Machines, Regular Expression

Context Free Languages ➜ Context Free Grammar, **Push-down Automata**

Non-Recursively Enumerable Languages

Recursively Enumerable Languages

Recursive Languages

Context-Free Languages

Regular Languages
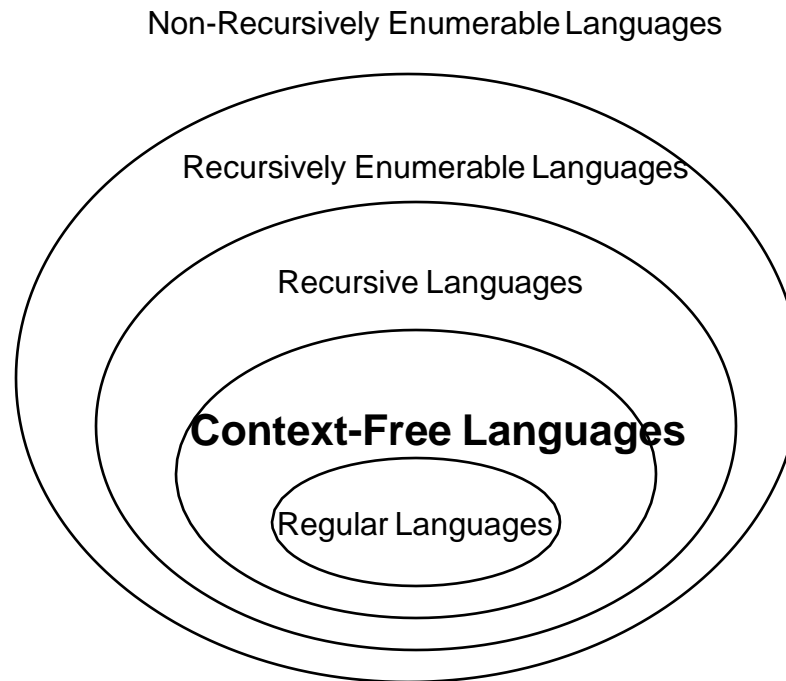
Pushdown Automata (PDA)

- **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:**

  $\{0^n 1^n \mid 0 =< n\}$ is *not* regular, but

  $\{0^n 1^n \mid 0 \leq n \leq k,\ \text{for some fixed } k\}$ is regular, for any fixed k.

- **For k=3:**

  $L = \{\varepsilon,\ 01,\ 0011,\ 000111\}$
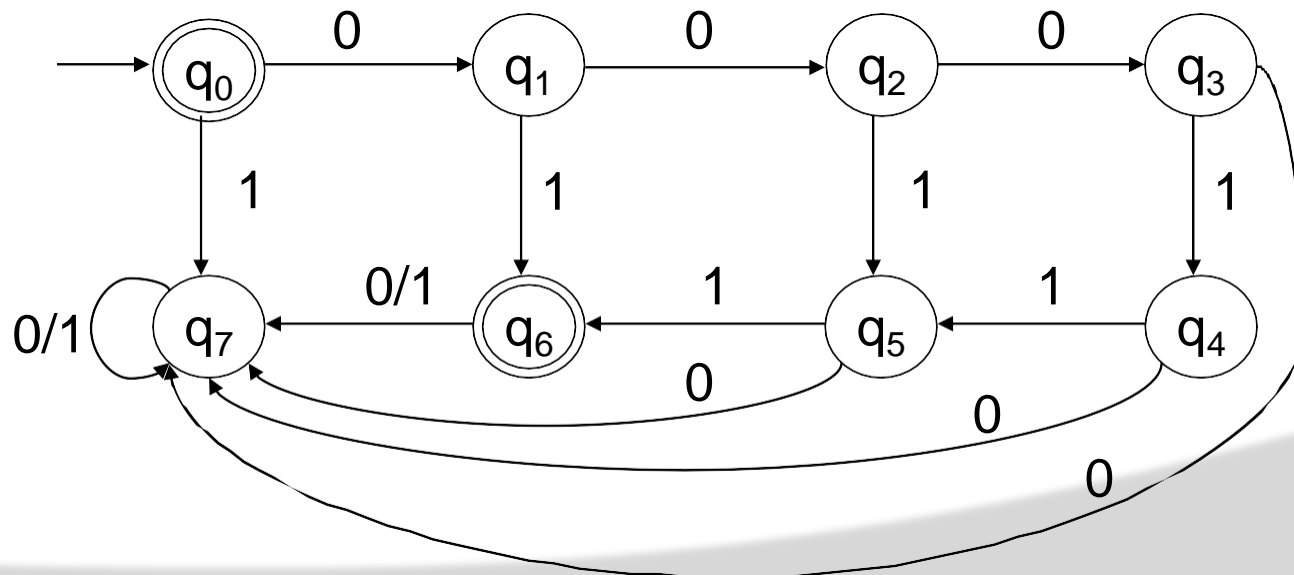
- In a DFA, each state remembers a finite amount of information.

- 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 <u>pushdown automaton (PDA)</u> is a seven-tuple:

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

Q   A <u>finite</u> set of states

Σ   A <u>finite</u> input alphabet

Γ   A <u>finite</u> 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

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

- Consider the various parts of δ:

  Q x (Σ U {ε}) x Γ –> finite subsets of Q x Γ*

  - 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*.

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

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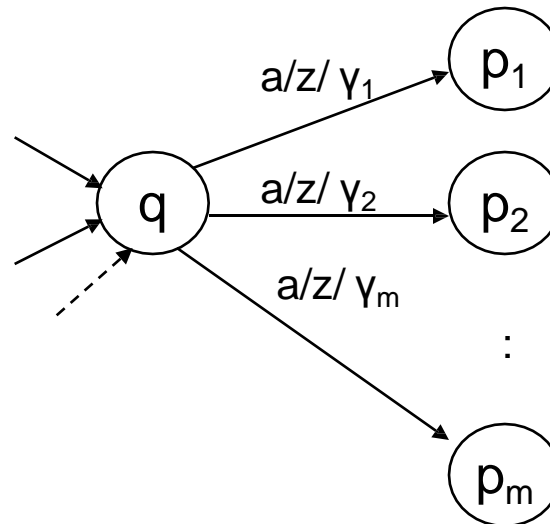
$$\delta(q, \varepsilon, z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_m, \gamma_m)\}$$

- 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**
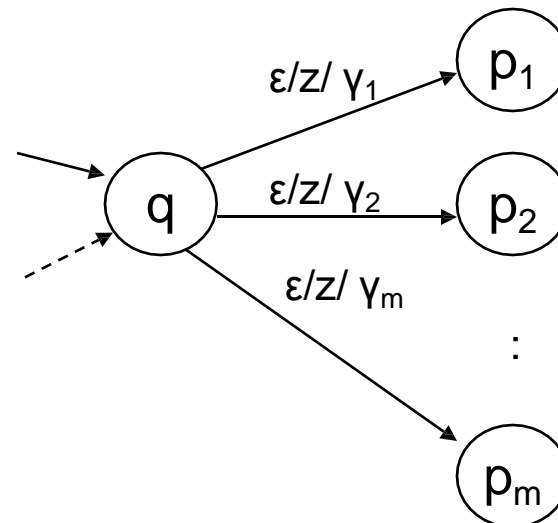
- **Example:** 0^n1^n, $n>=0$

  $M = (\{q_1, q_2\}, \{0, 1\}, \{L, \#\}, δ, q_1, \#, Ø)$

  δ:

  (1) $δ(q_1, 0, \#) = \{(q_1, L\#)\}$ // stack order: L on top, then # below

  (2) $δ(q_1, 1, \#) = Ø$ // illegal, string rejected, *When will it happen?*

         $δ(q_1, 0, L) = \{(q_1, LL)\}$

         $δ(q_1, 1, L) = \{(q_2, ε)\}$

         $δ(q_2, 1, L) = \{(q_2, ε)\}$

         $δ(q_2, ε, \#) = \{(q_2, ε)\}$ //if ε read & stack hits bottom, accept

         $δ(q_2, ε, L) = Ø$ // illegal, string rejected

  (3)(8) $δ(q_1, ε, \#) = \{(q_2, ε)\}$ // n=0, accept

- **Goal:** (acceptance)
  - Read the entire input string
  - Terminate with an empty stack

- Informally, a string is accepted if there exists a computation that uses up all the input and leaves the stack empty.
- *How many rules should be there in delta?*

- **Language: $0^n 1^n$, n>=0**

  δ:

  (1) $\delta(q_1, 0, \#) = \{(q_1, L\#)\}$ // stack order: L on top, then # below

  (2) $\delta(q_1, 1, \#) = \emptyset$ // illegal, string rejected, *When will it happen?*

  (3) $\delta(q_1, 0, L) = \{(q_1, LL)\}$ ←

  (4) $\delta(q_1, 1, L) = \{(q_2, \varepsilon)\}$

  (5) $\delta(q_2, 1, L) = \{(q_2, \varepsilon)\}$

  (6) $\delta(q_2, \varepsilon, \#) = \{(q_2, \varepsilon)\}$ //if ε read & stack hits bottom, accept

  (7) $\delta(q_2, \varepsilon, L) = \emptyset$ // illegal, string rejected

  (8) $\delta(q_1, \varepsilon, \#) = \{(q_2, \varepsilon)\}$ // n=0, accept

- **0011**

- *(q1, 0 011, #) |-*

  (q1, 0 11, L#) |-

  (q1, 1 1, LL#) |-

  (q2, 1, L#) |-

  (q2, e, #) |-

  (q2, e, e):   accept

- *011*

- *(q1, 0 11, #) |-*

  (q1, 1 1, L#) |-

  (q2, 1, #) |-

  Ø :   reject

- *Try   001*

- **Example:** balanced parentheses,
- e.g. in-language: ((())()), or (())(), but not-in-language: ((())

$M = (\{q_1\}, \{"(", ")"\}, \{L, \#\}, \delta, q_1, \#, \varnothing)$

δ:

(1) $\delta(q_1, (, \#) = \{(q_1, L\#)\}$ // stack order: L-on top-then- # lower

(2) $\delta(q_1,), \#) = \varnothing$ // illegal, string rejected

(3) $\delta(q_1, (, L) = \{(q_1, LL)\}$

(4) $\delta(q_1,), L) = \{(q_1, \varepsilon)\}$

(5) $\delta(q_1, \varepsilon, \#) = \{(q_1, \varepsilon)\}$ //if ε read & stack hits bottom, accept

(6) $\delta(q_1, \varepsilon, L) = \varnothing$ // illegal, string rejected

                                       // *What does it mean? When will it*

happen?

- **Goal:** (acceptance)
    - Read the entire input string
    - Terminate with an empty stack

- Informally, a string is accepted if there exists a computation that uses up all the input and leaves the stack empty.
- *How many rules should be in delta?*

340

$$(, \# \mid L\#$$

$$\varepsilon, \# \mid \varepsilon \quad q_0 \quad (, L \mid LL$$

$$), L \mid \varepsilon$$
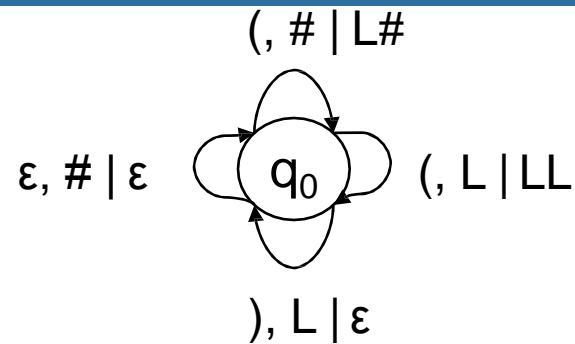
■ Example Computation:

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)

341

- **Example PDA #1:** For the language $\{x \mid x = wcw^r$ and w in $\{0,1\}^*$, but sigma$=\{0,1,c\}\}$
- *Is this a regular language?*
- *Note: length $|x|$ is odd*

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

δ:

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

- **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

342

- **Example Computation:**

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

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

(2) $\delta(q_1, 0, B) = \{(q_1, BB)\}$

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

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

(11) $\delta(q_1, 1, G) = \{(q_1, GG)\}$

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

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

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

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)	-

343

- **Example Computation:**

(1) $\delta(q_1, 0, \#) = \{(q_1, B\#)\}$ (9) $\delta(q_1, 1, \#) = \{(q_1, G\#)\}$

(2) $\delta(q_1, 0, B) = \{(q_1, BB)\}$ (10) $\delta(q_1, 1, B) = \{(q_1, GB)\}$

(3) $\delta(q_1, 0, G) = \{(q_1, BG)\}$ (11) $\delta(q_1, 1, G) = \{(q_1, GG)\}$

(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) $\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$ (12) $\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$

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

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

344

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

- *Note: length $|x|$ is even*

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

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.

(1) $\delta(q_1, 0, \#) = \{(q_1, B\#)\}$ (6) $\delta(q_1, 1, G) = \{(q_1, GG), (q_2, \varepsilon)\}$

(2) $\delta(q_1, 1, \#) = \{(q_1, G\#)\}$ (7) $\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$

(3) $\delta(q_1, 0, B) = \{(q_1, BB), (q_2, \varepsilon)\}$ (8) $\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$

(4) $\delta(q_1, 0, G) = \{(q_1, BG)\}$ (9) $\delta(q_1, \varepsilon, \#) = \{(q_2, \varepsilon)\}$

(5) $\delta(q_1, 1, B) = \{(q_1, GB)\}$ (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
			$(q_2, 0, \#)$ by option 2
q_1	ε	BBB#	(3) option #1 -crashes, no-rule to apply-
			$(q_2, \varepsilon, B\#)$ 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 $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ 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:

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

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)

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

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

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

- **Examples:** (PDA #1)

$(q_1, (())), L\#) \mid— (q_1, ())),LL\#)$ (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 <u>Greibach Normal Form</u> (GNF).

Only one non-terminal in front.

- **Example:**

  S $\rightarrow$ aAB | bB
  A $\rightarrow$ aA | a
  B $\rightarrow$ 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, $\emptyset$) where:
  Q = {q}
  Σ = T
  Γ = V
  z = S
  δ: for all a in Σ and A in Γ, $\delta(q, a, A)$ contains (q, γ)
  if $A \rightarrow a\gamma$ is in P or rather:
  $\delta(q, a, A) = \{(q, \gamma) \mid A \rightarrow a\gamma$ is in P and γ is in $\Gamma^*\}$,
  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.

$S \rightarrow aS$ G is in GNF

$S \rightarrow a$ $L(G) = a^+$

Construct M as:

$Q = \{q\}$

$\Sigma = T = \{a\}$

$\Gamma = V = \{S\}$

$z = S$

$\delta(q, a, S) = \{(q, S), (q, \varepsilon)\}$

$\delta(q, \varepsilon, S) = \varnothing$

- *Is δ complete?*

(1) S –> aA
(2) S –> aB
(3) A –> aA   G is in GNF
(4)A –> aB   L(G) = a⁺ b⁺   // This looks ok to me, one, two or more *a*'s in the start
(5) B –> bB
(6) B –> b   *[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 $(q, w, z_0) \models^* (q, \varepsilon, \varepsilon)$

  - If $(q, w, z_0) \models^* (q, \varepsilon, \varepsilon)$ 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 ith step of the PDAs' computation corresponds to the ith step in a corresponding leftmost derivation with the grammar.

- After the ith 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.

360

- 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:

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

would be:

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

S => aA   (1)

  => aaA   (3)

  => aaaA   (3)

  => aaaaB   (4)

  => aaaabB   (5)

  => aaaabb   (6)

Grammar:
(1)   S –> aA
(2)   S –> aB
(3)   A –> aA   G is in GNF
(4)   A –> aB   L(G) = a⁺b⁺
(5)   B –> bB
(6)   B –> b

$L(G) = a^+b^+$

- 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

(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

- 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   G is in GNF

(3) B –> b

(4) C –> cAB

(5) C –> cC   *Language?* $aab\ cc^*ab$

Construct M as:

$Q = \{q\}$

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

$\Gamma = V = \{S, A, B, C\}$

$z = S$

(1) $\delta(q, a, S) = \{(q, ABC)\}$ S->aABC

(2) $\delta(q, a, A) = \{(q, \varepsilon)\}$ A->a

(3) $\delta(q, a, B) = \emptyset$

(4) $\delta(q, a, C) = \emptyset$

(5) $\delta(q, b, S) = \emptyset$

(6) $\delta(q, b, A) = \emptyset$

(7) $\delta(q, b, B) = \{(q, \varepsilon)\}$ B->b

(8) $\delta(q, b, C) = \emptyset$

(9) $\delta(q, c, S) = \emptyset$

(10) $\delta(q, c, A) = \emptyset$

(11) $\delta(q, c, B) = \emptyset$

(12) $\delta(q, c, C) = \{(q, AB), (q, C)\}$ // C->cAB|cC

(13) $\delta(q, \varepsilon, S) = \emptyset$

(14) $\delta(q, \varepsilon, A) = \emptyset$

(15) $\delta(q, \varepsilon, B) = \emptyset$

(16) $\delta(q, \varepsilon, C) = \emptyset$

- **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!!*

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

- **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

$\delta(q, a, S) = \{(q, S)\}$

$\delta(q, b, S) = \{(q, \varepsilon)\}$

$\delta(q, \varepsilon, S) = \varnothing$

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:

  (1) S_1 -> ε | S, with new starting non-terminal S_1

  (2) S –> aS

  (3) S –> b

For PDA, add a new *Stack-bottom symbol* S_1, with new transitions:

  δ(q, ε, S_1) = {(q, ε), (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.

Sample CFG to GNF transformation:

- $0^n 1^n$, 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*

(Autonomous)

Dundigal, Hyderabad - 500 043

Computer Science and Engineering Department
IV Semester

Theory of Computation
Unit- V

Syllabus:

Turing machine: Turing machine, definition, model, design of Turing machine, computable functions, recursively enumerable languages, Church's hypothesis, counter machine, types of Turing machines (proofs not required), linear bounded automata and context sensitive language, Chomsky hierarchy of languages.

- Generalize the class of CFLs:

Non-Recursively Enumerable Languages

Recursively Enumerable Languages

Recursive Languages

Context-Free Languages
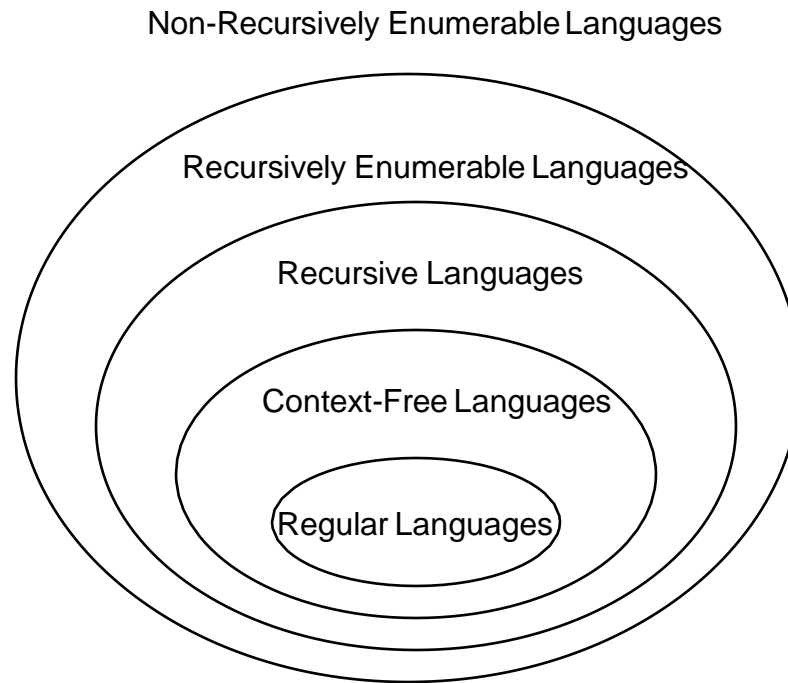
Regular Languages

- **Another Part of the Hierarchy:**

Non-Recursively Enumerable Languages

Recursively Enumerable Languages

Recursive Languages

Context-Sensitive Languages

Context-Free Languages **- ε**

Regular Languages **- ε**
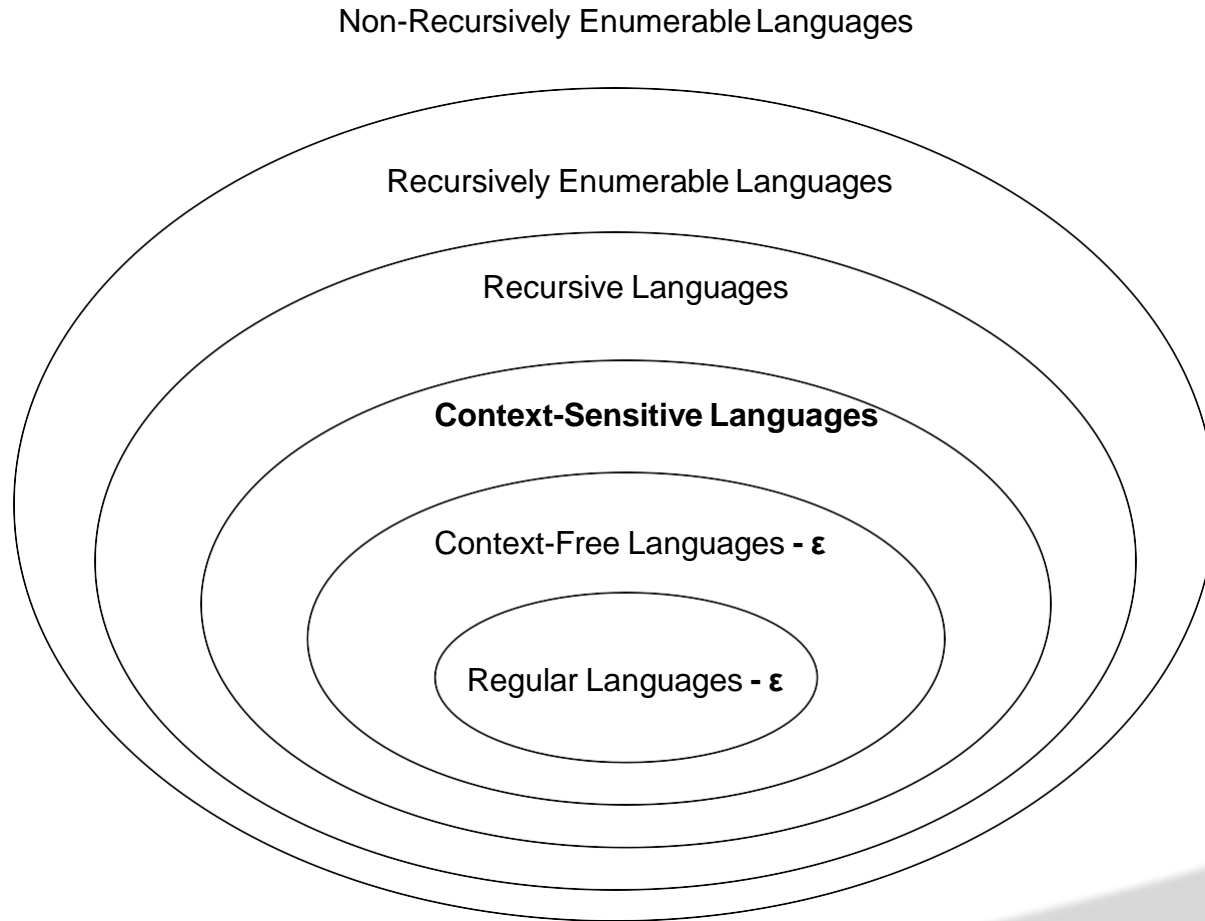
- Recursively enumerable languages are also known as *type 0* languages.

- 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.

375

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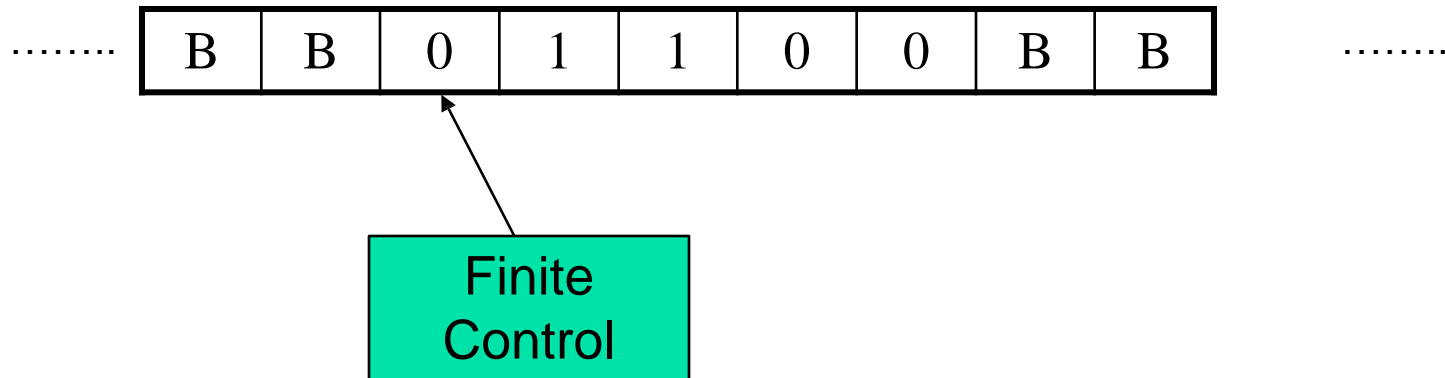
- Two-way, infinite tape, broken into cells, each containing one symbol.

- 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 possible, but Church-Turing declares equivalence of all

- A DTM is a seven-tuple:

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

Q   A <u>finite</u> set of states

Σ   A <u>finite</u> input alphabet, which is a subset of $\Gamma - \{B\}$

Γ   A <u>finite</u> tape alphabet, which is a strict <u>superset</u> 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, $\delta(q, s)$ 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
->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

Example #2: $\{0^n 1^n \mid n \geq 1\}$

	0	1	X	Y	B
->q_0	(q_1, X, R)	-	-	(q_3, Y, R) *0's finished*	-
q_1	$(q_1, 0, R)$ *ignore1*	(q_2, Y, L)	-	(q_1, Y, R) *ignore2*	- (more 0's)
q_2	$(q_2, 0, L)$ *ignore2*	-	(q_0, X, R)	(q_2, Y, L) *ignore1*	-
q_3	-	- (more 1's)	-	(q_3, Y, R) *ignore*	(q_4, B, R)
q_4*	-	-	-	-	-

Sample Computation: (on 0011), *presume state q looks rightward*

$q_0$0011BB.. |— X$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 B…

|— XXYYq_3BB…

|— XXYYBq_4

Making a TM for $\{0^n 1^n \mid n >= 1\}$

Try n=2 or 3 first.
- q0 is on 0, replaces with the character to X, changes state to q1, moves right
- q1 sees next 0, ignores (both 0's and X's) and keeps moving right
- q1 hits a 1, replaces it with Y, state to q2, moves left
- q2 sees a Y or 0, ignores, continues left
- when q2 sees X, moves right, returns to q0 for looping step 1 through 5
- when finished, q0 sees Y (no more 0's), changes to pre-final state q3
- q3 scans over all Y's to ensure there is no extra 1 at the end (to crash on seeing any 0 or 1)
- when q3 sees B, all 0's matched 1's, done, changes to final state q4
- blank line for final state q4

Try n=1 next.

Make sure unbalanced 0's and 1's, or mixture of 0-1's,
        "crashes" in a state not q4, as it should be

$q_0$0011BB.. |— X$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 B...

|— XXYYq_3 BB...

|— XXYYBq_4

380

	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

Logic: cross 0's with X's, scan right to look for corresponding 1, on finding it cross it with Y, and scan left to find next leftmost 0, keep iterating until no more 0's, then scan right looking for B.

- The TM matches up 0's and 1's
- q_1 is the "scan right" state, looking for 1
- q_2 is the "scan left" state, looking for X
- q_3 is "scan right", looking for B
- q_4 is the final state

Can you extend the machine to include n=0?
How does the input-tape look like for string epsilon?

- **Other Examples:**

    000111   00

    11   001

    011

	0	1	B
q_0	(q_1, B, R)		(q_4, B, R)
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	-	(q_3, B, L)	-
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4^*	-	-	-

Logic: Keep deleting 0 and corresponding 1 from extreme ends, until none left.
- q_0 deletes a leftmost 0 and let q_1 scan through end of string, q_0 accepts on epsilon
- q_1 scans over the string and makes q_2 expecting 1 on the left
- q_2 deletes 1 and let q_3 "scan left" looking for the start of current string
- q_3 lets q_0 start the next iteration
- q_4 is the final state

Any bug?

Try on:

000111	00
11	001
011	

	0	1	B
q_0	(q_1, B, R)	$(q_3, 1, L)$	(q_4, B, R)
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	-	(q_3, B, L)	-
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4^*	-	-	-

Logic: This machine still works correctly for all strings in the language, but start
a string with 1 (not in the language),
and it loops on B1 for ever.

- **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 $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ 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 $\alpha_1 = \boldsymbol{\varepsilon}$
  - If $\alpha_2 = \boldsymbol{\varepsilon}$ then a blank is being scanned

Example: (for TM #1)

$q_0 0011$ $X q_1 011$ $X0 q_1 11$ $X q_2 0Y1$ $q_2 X0Y1$

$X q_0 0Y1$ $XX q_1 Y1$ $XXY q_1 1$ $XX q_2 YY$ $X q_2 XYY$

$XX q_0 YY$ $XXY q_3 Y$ $XXYY q_3$ $XXYYB q_4$

- Suppose the following is the current ID of a DTM

$$x1x2\ldots xi\text{-}1qxixi+1\ldots xn$$

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

$$q_0 w \mathbin{|}{-}^* \alpha_1 p \alpha_2$$

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 <u>there exists</u> 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 <u>not</u> 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.

- **Recall the Hierarchy:**

Non-Recursively Enumerable Languages

Recursively Enumerable Languages

Recursive Languages Context-Sensitive Languages

Context-Free Languages - **ε**

Regular Languages - **ε**
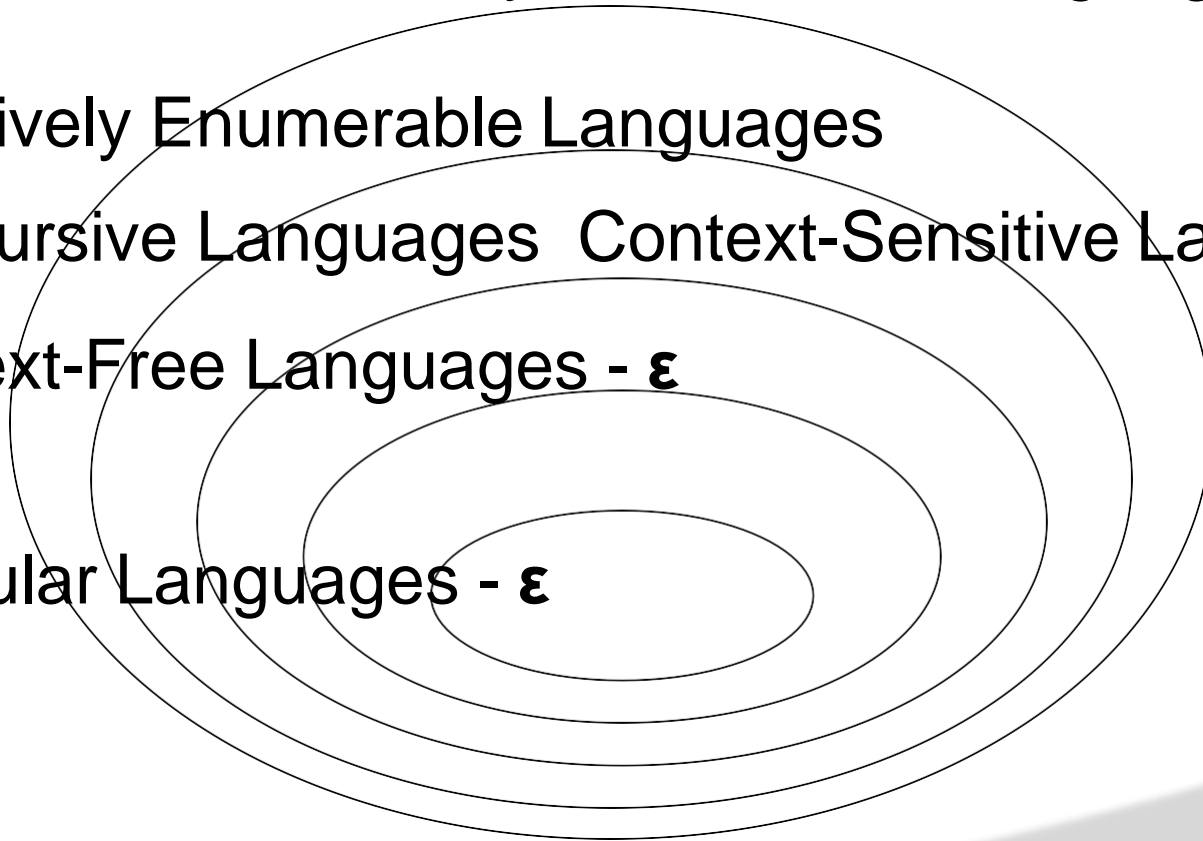
- **Observation:** Let L be an r.e. language. Then there is an infinite list $M_0, M_1, \ldots$ of TMs such that $L = L(M_i)$.

- **Question:** Let L be a **recursive** language, and $M_0, M_1, \ldots$ a list of all TMs such that $L = L(M_i)$, and choose any $i \geq 0$. Does M_i always halt?

- **Answer:** Maybe, maybe not, but *at least one in the list does*.

- **Question:** Let L be a **recursive enumerable** language, and $M_0, M_1, \ldots$ a list of all TMs such that $L = L(M_i)$, and choose any $i \geq 0$. Does M_i always halt?

- **Answer:** Maybe, maybe not. Depending on L, none might halt or some may halt.

  - If L is also recursive then L is recursively enumerable, *recursive is subset of r.e.*

- **Question:** Let L be a r.e. language that is not recursive (L is in r.e. − r), and M_0, M_1, ... a list of all TMs such that L = $L(M_i)$, and choose any i>=0. Does M_i always halt?

- **Answer:** No! If it did, then L would not be in r.e. − r

L is Recursively enumerable:

    TM exist: M_0, M_1, …

    They accept string in L, and do not accept any string outside L

L is Recursive:

    at least one TM halts on L and on $\sum^*$-L, others may or may not

L is Recursively enumerable but not Recursive:

    TM exist: M_0, M_1, …

    but <u>none</u> halts on <u>all</u> x in $\sum^*$-L

    M_0 goes on infinite loop on a string p in $\sum^*$-L, while M_1 on q in $\sum^*$-L

    However, each correct TM accepts each string in L, and none in $\sum^*$-L

L is not R.E:

    no TM exists

- **Let *M* be a TM.**

  - Question: Is L(M) r.e.?
  - Answer: Yes! By definition it is!

  - Question: Is L(M) recursive?
  - Answer: Don't know, we don't have enough information.

  - Question: Is L(M) in r.e – r?
  - Answer: Don't know, we don't have enough information.

- **Let *M* be a TM that <u>halts</u> on all inputs:**

  - Question: Is L(M) recursively enumerable?
  - Answer: Yes! By definition it is!

  - Question: Is L(M) recursive?
  - Answer: Yes! By definition it is!

  - Question: Is L(M) in r.e – r?
  - Answer: No! It can't be. Since *M* always halts, L(M) is recursive.

- **Let _M_ be a TM.**
  - As noted previously, L(M) is recursively enumerable, but may or may not be recursive.
  - Question: Suppose, we know L(M) is recursive. Does that mean _M_ always halts?
  - Answer: Not necessarily. However, some TM _M'_ must exist such that L(M') = L(M) and _M'_ always halts.

  - Question: Suppose that L(M) is in r.e. − r. Does _M_ always halt?
  - Answer: No! If it did then L(M) would be recursive and therefore not in r.e. −r.

- **Let *M* be a TM, and suppose that *M* loops forever on some string *x*.**
    - Question: Is L(M) recursively enumerable?
    - Answer: Yes! By definition it is. But, obviously *x* is not in L(M).
    - Question: Is L(M) recursive?
    - Answer: Don't know. Although *M* doesn't always halt, some other TM *M'*<u>may</u> exist such that L(M') = L(M) and *M'* always halts.
    - Question: Is L(M) in r.e. – r?
    - Answer: Don't know.

    May be another *M'* will halt on *x*, and on all strings!
    May be no TM for this
 L(M) does halt on all strings! We just do not know!

- **Other (Extended) TM Models:**
  - One-way infinite tapes
  - Multiple tapes and tape heads
  - Non-Deterministic TMs
  - Multi-Dimensional TMs (n-dimensional tape)
  - Multi-Heads
  - Multiple tracks

  All of these extensions are equivalent to the basic DTM model

- **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.

$$w \longrightarrow \quad M \quad \begin{array}{l} \longrightarrow \text{ yes} \\ \\ \longrightarrow \text{ no} \end{array}$$

  - 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.

$$w \longrightarrow \quad M \quad \longrightarrow \text{ yes}$$

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

398

- **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:
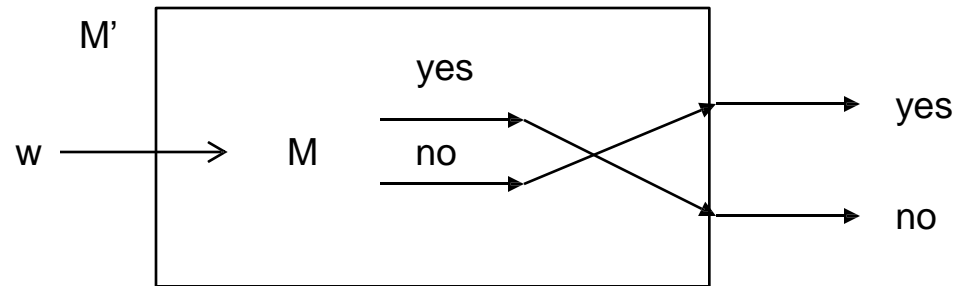
- **Note That:**
  - *M'* accepts iff *M* does not
  - *M'* always halts since *M* always halts

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:
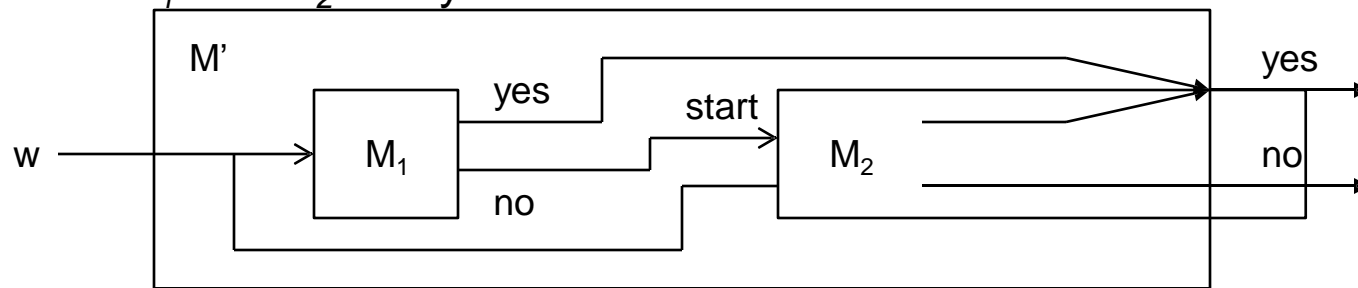
- **Note That:**
    - $L(M') = L(M_1) \cup L(M_2)$
        - $L(M')$ is a subset of $L(M_1) \cup L(M_2)$
        - $L(M_1) \cup L(M_2)$ is a subset of $L(M')$
    - M' always halts since M_1 and M_2 always halt

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

- **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 <u>either</u> M_1 <u>or</u> 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, *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)

Non-Recursively Enumerable Languages

Recursively Enumerable Languages

L $\bar{L}$

Recursive Languages

- **In terms of the hieararchy:** (possibility #2)

Non-Recursively Enumerable Languages

L L

Recursively Enumerable Languages

Recursive Languages

- **In terms of the hierarchy:** (possibility #3)

Non-Recursively Enumerable Languages

L $\bar{L}$

Recursively Enumerable Languages

Recursive Languages

■ **In terms of the hierarchy:** (Impossibility #1)

Non-Recursively Enumerable Languages

$$L \qquad \bar{L}$$

Recursively Enumerable Languages

Recursive Languages

- **In terms of the hierarchy:** (Impossibility #2)

Non-Recursively Enumerable Languages

$$\bar{L}$$

Recursively Enumerable Languages

$$L$$

Recursive Languages

- **In terms of the hierarchy:** (Impossibility #3)

Non-Recursively Enumerable Languages

$\overline{L}$

Recursively Enumerable Languages

L

Recursive Languages

- **Note:** This gives/identifies three approaches to show that a language is not recursive.
    - Show that the language's <u>complement</u> is not recursive, in one of the two ways:
        - Show that the language's <u>complement</u> is recursively enumerable but not recursive
        - Show that the language's <u>complement</u> is not even recursively enumerable

- **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.

The Halting Problem - Background

- **Definition:** A <u>decision problem</u> 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?

-

- **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.

- 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.

- **Over-All Approach:**

  - We will show that a language L_d is not recursively enumerable
  - From this it will follow that   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.

  $$\overline{L_d}$$

- **As We Will See:**
  - P3 will correspond to the language L_u
  - Proving P3 (un)decidable is equivalent to proving L_u (non)recursive

- Let $M = (Q, \Sigma, \Gamma, \delta, q_1, B, \{q_n\})$ be a TM, where

  $Q = \{q_1, q_2, \ldots, q_n\}$, order the states from 1 through n $\Sigma = \{x_1, x_2\} = \{0, 1\}$
  $\Gamma = \{x_1, x_2, x_3\} = \{0, 1, B\}$

- Encode each transition:

  $\delta(q_i, x_j) = (q_k, x_l, d_m)$ where q_i and q_k are in ordered Q
       x_j and x_l are in Σ,
       and d_m is in $\{L, R\} = \{d_1, d_2\}$

as:

     $0^i10^j10^k10^l10^m$ where the number of 0's indicate the corresponding id, and single 1 acts as a barrier

- The TM M can then be encoded as:

  $111code_111code_211code_311 \ldots 11code_r111$

where each $code_i$ is one transitions' encoding, and 11's are barriers between transitions from the table row-major. Let this encoding of M be denoted by $<M>$.

- Every state, tape symbol, and movement symbol is encoded as a sequence of 0's:

q_1,   0
q_2,   00
q_3 000
    :

0   0
1   00
B   000

L   0
R   00

- Note that 1's are not used to represent the above, since 1 is used as a special separator symbol.

- Example:

$\delta(q_2, 1) = (q_3, 0, R)$

Is encoded as:

00100100010100

	0	1	B
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	$(q_3, 0, R)$	-	-
q_3	-	-	-

What is the L(M)?

Coding for the above table:

*111*0101010100**11**0100101001001**11**01000100100010**11**0010100010100*111*

Are the followings correct encoding of a TM?

01100001110001

111111

$L_t = \{x \mid x$ is in $\{0, 1\}^*$ and x encodes a TM$\}$

- Question: Is L_t recursive?
- Answer: Yes. [Check only for format, i.e. the order and number of 0's and 1's, syntax checking]

- Question: Is L_t decidable:
- Answer: Yes (same question).

- Define the language L_u as follows:

  $L_u = \{x \mid x$ 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

- **Recall:**

	0	1	B
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	$(q_3, 0, R)$	-	-
q_3	-	-	-

- **Which of the following are in L_u?**

1110101010100110100101001001101000100100010110010100010100**111**

1110101010100110100101001001101000100100010110010100010100**111**011
10

1110101010100110100101001001101000100100010110010100010100**111**001
10111

0110000111 0001

111111

(P3):

Input: Turing machine *M* with input alphabet Σ and one final state, and string *w* in Σ*.

Question: Is *w* in L(M)?

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

■ Universal TM *(UTM)* is the machine for L_u
  - presuming it is r.e.!   *Can you write a program to accept strings in Lu?*
■ **Notes:**
  - L_u is P3 expressed as a language
  - *Asking if L_u is recursive is the same as asking if P3 is decidable.*
    - *Can you write a Halting program for accept/reject of strings in Sigma* ?*
  - We will show that L_u is *not recursive*, and from this it will follow that P3 is *un-decidable*.
  - From this we can further show that the *Halting problem is un-decidable*.
  => A general concept: *a decision problem ≡ a formal language*

- Define another language L_d as follows:
- *[L_d_bar = {self accepting TM encodings}, everything else is L_d]*

  L_d = {x | x is in {0, 1}* and (a) either x is **not** a TM,

  (b) or x is a TM, call it M, and x is **not** in L(M)}   (1)

  - *Note, there is only one string x*
  - *And, the question really is the complement of "does a TM accept its own encoding?" (Ld-bar's complement)*

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

  1. x is **not** a TM, in which case x is *in* L_d
  2. x is a TM, call it M, and either:
     a) x is **not** in L(M), in which case x is *in* L_d
     b) x is in L(M), in which case x is **not** in L_d

	0	1	B
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	$(q_3, 0, R)$	-	-
q_3	-	-	-

■ **Which of the following are in L_d?**

1110101010100110100101001001101000100010001011001010100010100111

01100001110001

Change above machine to accept strings ending with 1: the encoding will not be in L_d

- **Lemma:** L_d *is not recursively enumerable.*   *[No TM for L_d!!!]*

- **Proof:** (by contradiction)

Suppose that L_d is recursively enumerable. In other words, there exists a TM M such that:

$$L_d = L(M) \qquad (2)$$

Now suppose that w is a string encoding of M. $\qquad (3)$

Case 1) **w is in L_d** $\qquad (4)$

By definition of L_d given in (1), either w does not encode a TM, or w does encode a TM, call it M, and w is not in L(M). But we know that w encodes a TM (3: that's where it came from). Therefore:

$$w \text{ is not in } L(M) \qquad (5)$$

But then (2) and (5) imply that **w is not in L_d** contradicting (4).

Case 2) **w is not in L_d** $\qquad (6)$

By definition of L_d given in (1), w encodes a TM, call it M, and:

$$w \text{ is in } L(M) \qquad (7)$$

But then (2) and (7) imply that **w is in L_d** contradicting (6).

Since both case 1) and case 2) lead to a contradiction, no TM M can exist such that $L_d = L(M)$. Therefore L_d is not recursively enumerable.

L_u is recursively enumerable
(you may ignore this slide, for now)

Input the string
   Decode the TM prefix, if it doesn't have one then the string is not in Lu
   Otherwise, run/simulate the encoded TM on the suffix
   If it terminates and accepts then the original string is in Lu.

If a given string is in Lu, then the above algorithm will correctly determine that, halt and say *yes*.

If the given string is not in Lu, then there are three cases:

1) the string doesn't have a TM as a prefix. In this case the above algo correctly detects this fact, and reports the string is not in Lu.
2) the string has a TM prefix, and the TM halts and rejects on the suffix. In this case the above algo correctly reports the string is not in Lu.
3) the string has a TM prefix, but it goes into an infinite loop on the suffix. In this case the above algo also goes into an infinite loop, but that's ok since the string as a whole is not in Lu anyway, and we are just trying to show there exists a TM for only accepting strings in Lu.

From this proof note that if the prefix TM is a DFA or PDA, then our machine will also halt in the 3rd case above, no matter what the suffix is.

-- due to Dr. Bernhard (edited by me)

- **Define another language L_h:**

  $L_h = \{x \mid x$ is in $\{0, 1\}^*$ and $x = <M,w>$ where M is a TM encoding and M halts on $w\}$

  Note that L_h is P2 expressed as a language:

  (P2):
  Input: Turing machine M with input alphabet Σ and string w in Σ^*. Question: Does M halt on w?

■ **Proof:** (by contradiction)
Suppose that L_h is recursive. Recall that:

   $L_h = \{x \mid x$ is in $\{0, 1\}^*$ and $x = <M,w>$ where M is a TM encoding and M halts on $w\}$

and

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

Suppose that $L_h = L(M')$ where M' is a TM that always halts. Construct an algorithm (i.e., a TM that always halts) for L_u as follows:

430
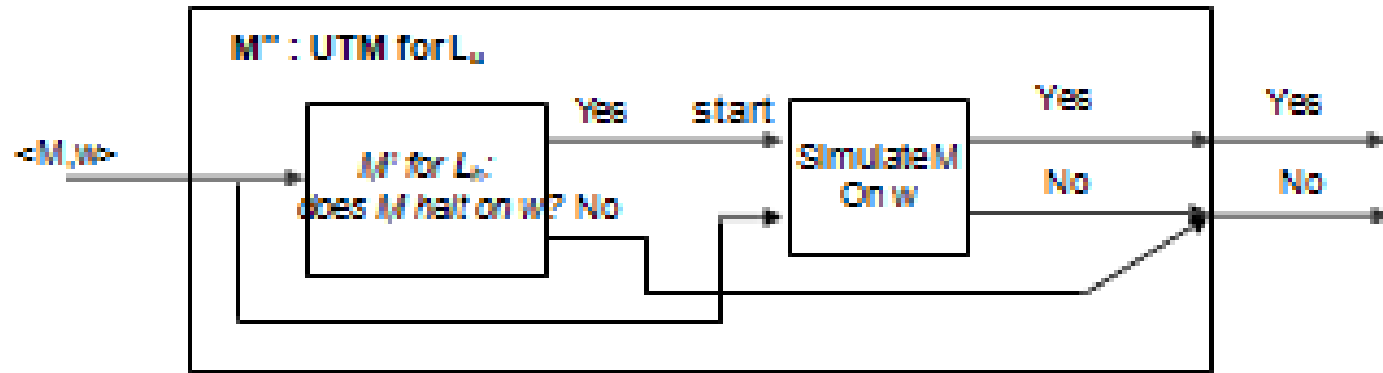
Suppose that *M'* always halts and L_h = L(M'). It follows that:

- M'' always halts
- $L(M'') = L_u$

- **The over-all logic of the proof is as follows:**
    1. If L_h is recursive, then so is L_u
    2. L_u is not recursive
    3. It follows that L_h is not recursive.

    The second point was established previously.

    The first point was established by the theorem on the preceding slide.

    This proof is also a reduction. Specifically, the problem of recognizing L_u was *reduced*
        to the problem of recognizing L_h.

    [L_u and L_h both are recursively enumerable: for proof see Dr. Shoaff!]

Examples of non-halting program:

http://cs.fit.edu/~ryan/tju/russell.c
http://cs.fit.edu/~ryan/tju/russell.scm
http://cs.fit.edu/~ryan/tju/russell.py

L$_q$ = {x | x is in {0, 1}*, x encodes a TM M, and M does contain an infinite loop} Or equivalently:

L$_q$ = {x | x is in {0, 1}*, x encodes a TM M, and there exists **no** string w in {0, 1}* such that M does **not** terminate on w}

Note that:

$\overline{L_q}$ = {x | x is in {0, 1}*, and either x does **not** encode a TM, or it does encode a TM, call

it M,

and there exists a string w in {0, 1}* such that M does **not** terminate on w}

Note that the above languages correspond to the following problem: (P0):

Input: Program P.

Question: Does P contain an infinite loop?

Using the techniques discussed, what can we prove about L$_q$ or its' complement?

More examples of non-recursive languages:

L_{ne} = {x | x is a TM M and L(M) is not empty} is r.e. but not recursive.

L_e = {x | x is a TM M and L(M) is empty} is not r.e.

L_r = {x | x is a TM M and L(M) is recursive} is not r.e

Note that L_r is not the same as L_h = {x | x is a TM M that always halts}

but L_h is in L_r.

L_{nr} = {x | x is a TM M and L(M) is not recursive} is not r.e.

Roger's TM for balanced parenthesis:	(	)	B
findPair	(findPair2, "(", R)	-	(final, B, R)
findPair2	(findPair2, "(", R)	(removePair, ")", L)	-
removePair	(fetch, "(", R)	(fetch, ")", R)	(goBack, B, L)
fetch	(retrieve, "(", R)	(retreive, ")", R)	(retreive, B, R)
retreive	(returnOpen, "(", L)	(returnClosed, ")", L)	(returnBlank, B, L)
returnOpen	(writeOpen, "(", L)	(writeOpen, ")", L)	(writeOpen, B, L)
returnClosed	(writeClosed, "(", L)	(writeClosed, ")", L)	(writeClosed, B, L)
returnBlank	(writeBlank "(", L)	(writeBlank, ")", L)	(writeBlank, B, L)
writeOpen	(removePair, "(", R)	(removePair, "(", R)	-
writeClosed	(removePair, ")", R)	(removePair, ")", R)	-
writeBlank	(removePair, B, R)	(removePair, B, R)	-
goBack	-	-	(backAgain, B, L)
backAgain	-	-	(seekFront, B, L)
seekFront	(seekFront, "(", L)	(seekFront, ")", L)	(findPair, B, R)
final*	-	-	-

<Quote> It was ambiguous, in my opinion, based on the definition in the Hopcroft book, i.e., the definition in the Hopcroft book was not clear/precise enought to

account this special case. I don't have the book in front of me right now, but I think this is the example I used in class: Consider the TM that has exactly one state, but no transitions. Perfectly valid TM, and it would give us this encoding (111111). In that case the encoded machine would accept sigma because the highest numbered state would be q0, the only state, and that would be the final state under the Hopcroft encoding. Now consider the TM that has exactly two states, but no transitions. Also a perfectly valid TM, and it would give us the same encoding. In that case the encoded machine would not accept anything because the final state is q1 (highest numbered state), and there is no way to get to it. I used it only as a way to raise that issue in class, i.e., the the Hopcroft definition is a bit ambiguous in this case.*

One way to resolve the ambiguity is to require the encoding to specifically specify the final state (at the end or something). In that case, 111111 isn't even a valid TM, since it doesn't specify the final state. Another related question is, does a TM even have to have any states at all to be a valid TM? The encoding would have to be able to isolate that as a unique string also. <End Quote>

Phil Bernhard