

Introduction to Automata Theory



Reading: Chapter 1



What is Automata Theory?

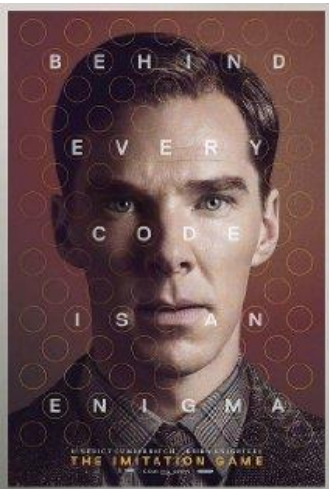
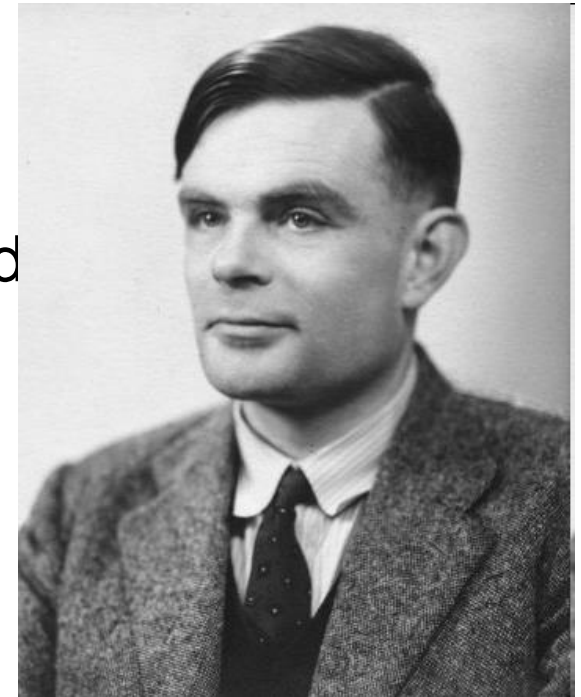
- *Study of abstract computing devices, or “machines”*
- **Automaton = an abstract computing device**
 - Note: A “device” need not even be a physical hardware!
- **A fundamental question in computer science:**
 - Find out what different models of machines can do and cannot do
 - The *theory of computation*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

(A pioneer of automata theory)

Alan Turing (1912-1954)

- Father of Modern Computer Science
- English mathematician
- Studied abstract machines called **Turing machines** even before computers existed

Heard of the Turing test?





Theory of Computation: A Historical Perspective

1930s	<ul style="list-style-type: none">• Alan Turing studies Turing machines• Decidability• Halting problem
1940-1950s	<ul style="list-style-type: none">• “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

Languages & Grammars

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

$\{0,1\}$

Or “**words**”

↓
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.

$S \longrightarrow 0A$

$B \longrightarrow 1B$

$A \longrightarrow 1A$

$B \longrightarrow 0F$

$A \longrightarrow 0B$

$F \longrightarrow \epsilon$

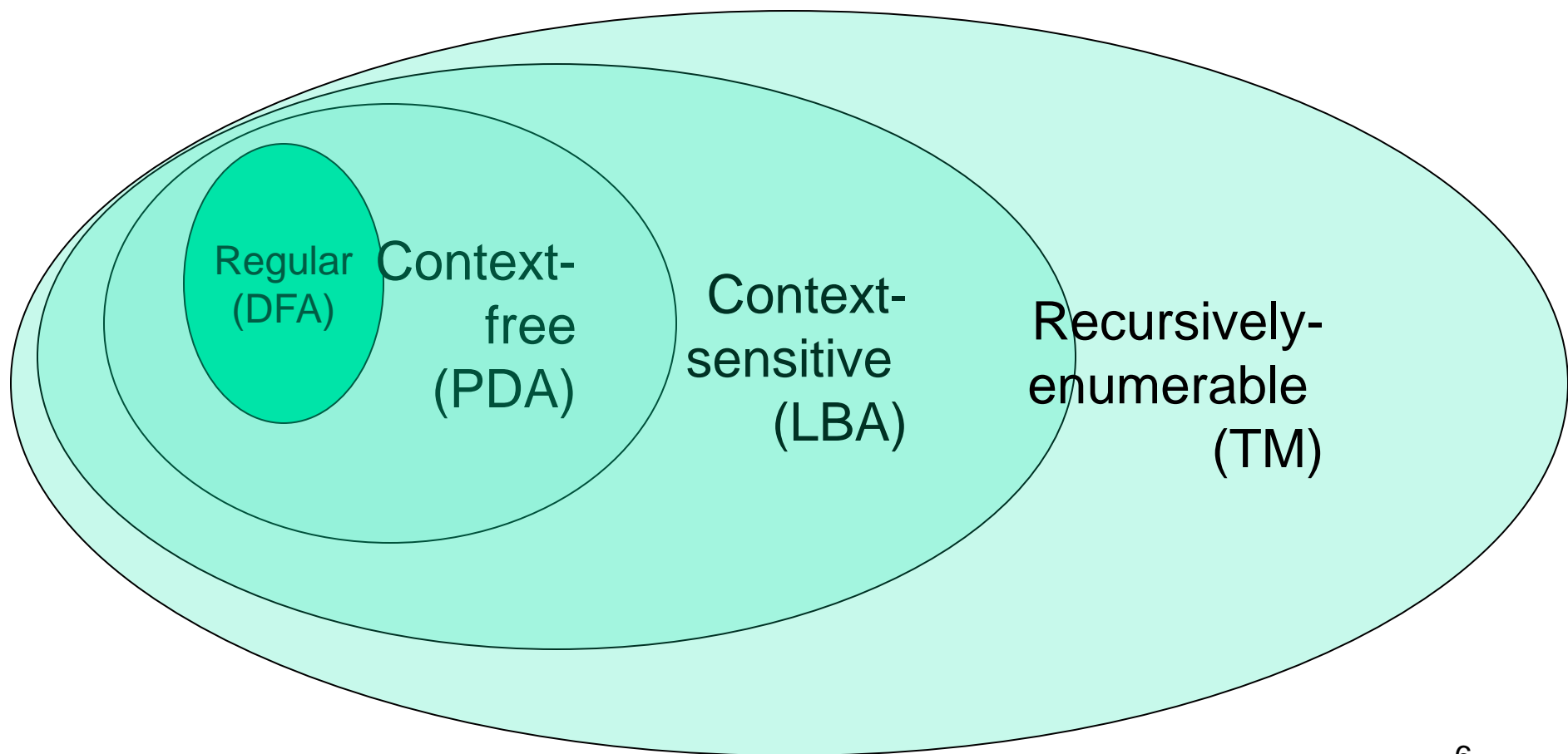
- Languages: “A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols”
- Grammars: “A grammar can be regarded as a device that enumerates the sentences of a language” - nothing more, nothing less
- N. Chomsky, *Information and Control*, Vol 2, 1959



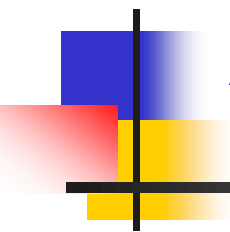
The Chomsky Hierarchy



- A containment hierarchy of classes of formal languages



The Central Concepts of Automata Theory





Alphabet

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

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



Strings

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

- **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., $x = 010100$ $|x| = 6$
 - $x = 01\ \varepsilon\ 0\ \varepsilon\ 1\ \varepsilon\ 00\ \varepsilon$ $|x| = ?$
- xy = concatenation of two strings x and y



Powers of an alphabet

Let Σ be an alphabet.

- Σ^k = the set of all strings of length k
- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$
- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$



Languages

L is said to be a language over alphabet Σ , only if $L \subseteq \Sigma^$*

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

Examples:

1. Let L be *the* language of all strings consisting of n 0's followed by n 1's:
2. Let L be *the* language of all strings of with equal number of 0's and 1's:

$$L = \{\epsilon, 01, 0011, 000111, \dots\}$$

$$L = \{\epsilon, 01, 10, 0011, 1100, 0101, 1010, 1001, \dots\}$$

→
Canonical ordering of strings in the language

Definition: \emptyset denotes the Empty language

- Let $L = \{\epsilon\}$; Is $L = \emptyset$?

NO



The Membership Problem

Given a string $w \in \Sigma^$ and a language L over Σ , decide whether or not $w \in L$.*

Example:

Let $w = 100011$

Q) Is $w \in$ the language of strings with equal number of 0s and 1s?

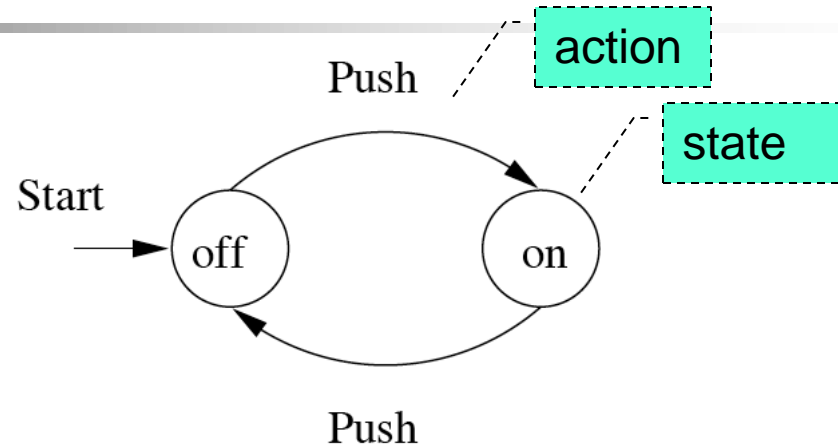


Finite Automata

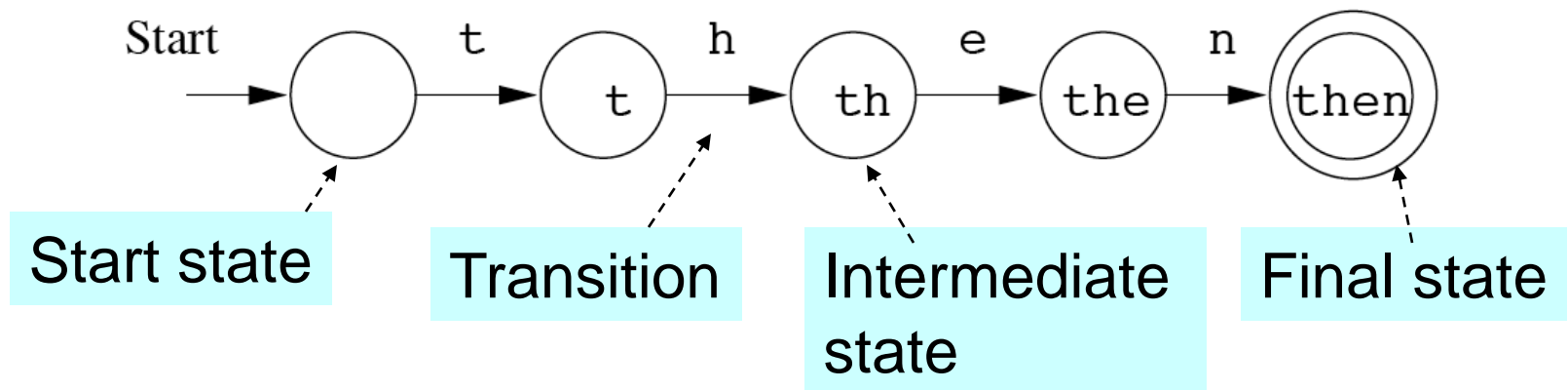
- Some Applications
 - Software for designing and checking the behavior of digital circuits
 - Lexical analyzer of a typical compiler
 - Software for scanning large bodies of text (e.g., web pages) for pattern finding
 - Software for verifying systems of all types that have a finite number of states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

Finite Automata : Examples

- On/Off switch



- Modeling recognition of the word “*then*”



Structural expressions

- Grammars
- Regular expressions
 - E.g., unix style to capture city names such as “Palo Alto CA”:

■ `[A-Z][a-z]*([][A-Z][a-z]*)*[][A-Z][A-Z]`

Start with a letter

A string of other letters (possibly empty)

Other space delimited words (part of city name)

Should end w/ 2-letter state code



Formal Proofs



Deductive Proofs

From the given statement(s) to a conclusion statement (what we want to prove)

- Logical progression by direct implications

Example for parsing a statement:

- “If $y \geq 4$, then $2^y \geq y^2$.”

given

conclusion

(there are other ways of writing this).



Example: Deductive proof

Let Claim 1: If $y \geq 4$, then $2^y \geq y^2$.

Let x be any number which is obtained by adding the squares of 4 positive integers.

Claim 2:

Given x and assuming that Claim 1 is true, prove that $2^x \geq x^2$

■ Proof:

1) Given: $x = a^2 + b^2 + c^2 + d^2$

2) Given: $a \geq 1, b \geq 1, c \geq 1, d \geq 1$

3) $\rightarrow a^2 \geq 1, b^2 \geq 1, c^2 \geq 1, d^2 \geq 1$

(by 2)

4) $\rightarrow x \geq 4$

(by 1 & 3)

5) $\rightarrow 2^x \geq x^2$

(by 4 and Claim 1)

“implies” or “follows”



On Theorems, Lemmas and Corollaries

We typically refer to:

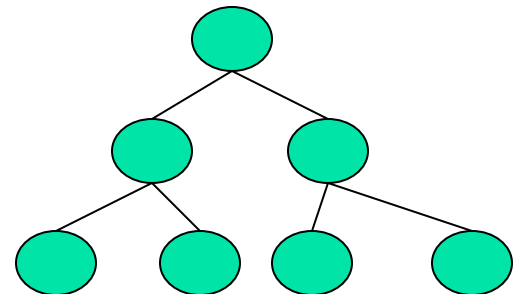
- A major result as a “**theorem**”
- An intermediate result that we show to prove a larger result as a “**lemma**”
- A result that follows from an already proven result as a “**corollary**”

An example:

Theorem: *The height of an n -node binary tree is at least $\text{floor}(\lg n)$*

Lemma: *Level i of a perfect binary tree has 2^i nodes.*

Corollary: *A perfect binary tree of height h has $2^{h+1}-1$ nodes.*





Quantifiers

“For all” or “For every”

- Universal proofs
- Notation= \forall

“There exists”

- Used in existential proofs
- Notation= \exists

Implication is denoted by \Rightarrow

- E.g., “IF A THEN B” can also be written as “ $A \Rightarrow B$ ”



Proving techniques

- **By contradiction**

- Start with the statement contradictory to the given statement
- E.g., To prove $(A \Rightarrow B)$, we start with:
 - $(A \text{ and } \sim B)$
 - ... and then show that could never happen

What if you want to prove that “ $(A \text{ and } B \Rightarrow C \text{ or } D)$ ”?

- **By induction**

- (3 steps) Basis, inductive hypothesis, inductive step

- **By contrapositive statement**

- If A then B \equiv If $\sim B$ then $\sim A$



Proving techniques...

- By counter-example
 - Show an example that disproves the claim
- Note: There is no such thing called a “proof by example”!
 - So when asked to prove a claim, an example that satisfied that claim is *not* a proof



Different ways of saying the same thing

- “*If H then C*”:
 - i. *H implies C*
 - ii. $H \Rightarrow C$
 - iii. *C if H*
 - iv. *H only if C*
 - v. *Whenever H holds, C follows*



“If-and-Only-If” statements

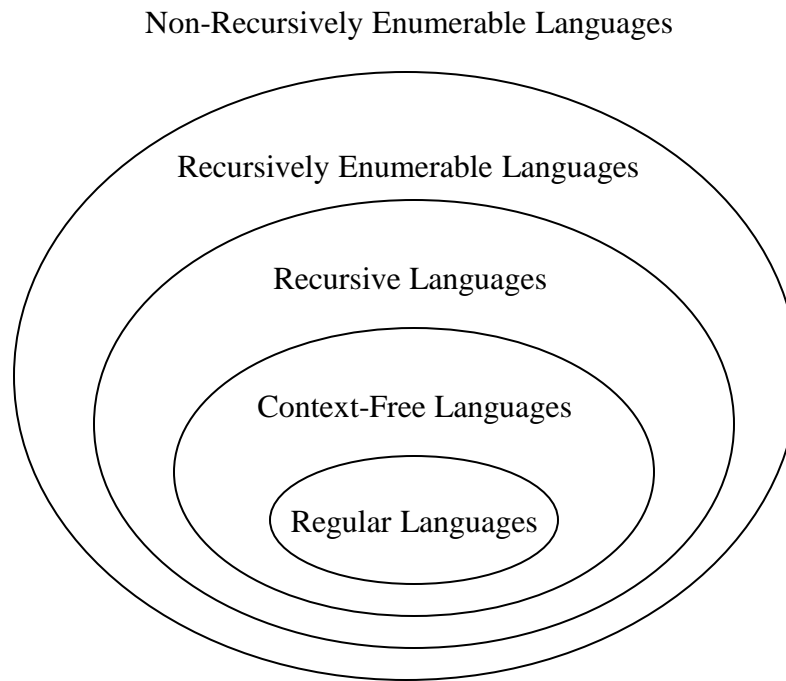
- “A if and only if B” ($A \iff B$)
 - (if part) if B then A (\implies)
 - (only if part) A only if B (\implies)
(same as “if A then B”)
- “If and only if” is abbreviated as “iff”
 - i.e., “A iff B”
- Example:
 - Theorem: *Let x be a real number. Then floor of x = ceiling of x if and only if x is an integer.*
- Proofs for iff have two parts
 - One for the “if part” & another for the “only if part”



Summary

- Automata theory & a historical perspective
- Chomsky hierarchy
- Finite automata
- Alphabets, strings/words/sentences, languages
- Membership problem
- Proofs:
 - Deductive, induction, contrapositive, contradiction, counterexample
 - If and only if
- Read chapter 1 for more examples and exercises

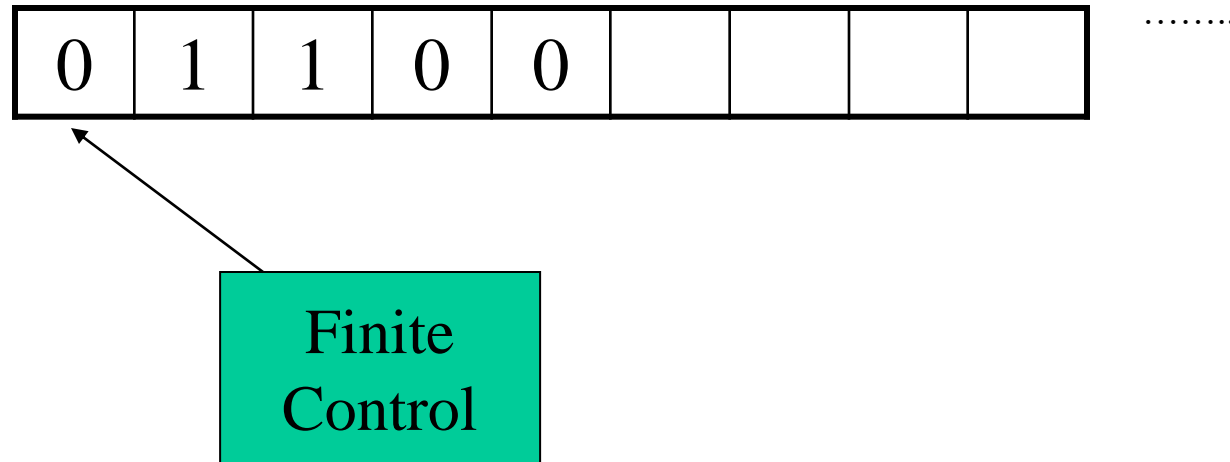
Hierarchy of languages



FA

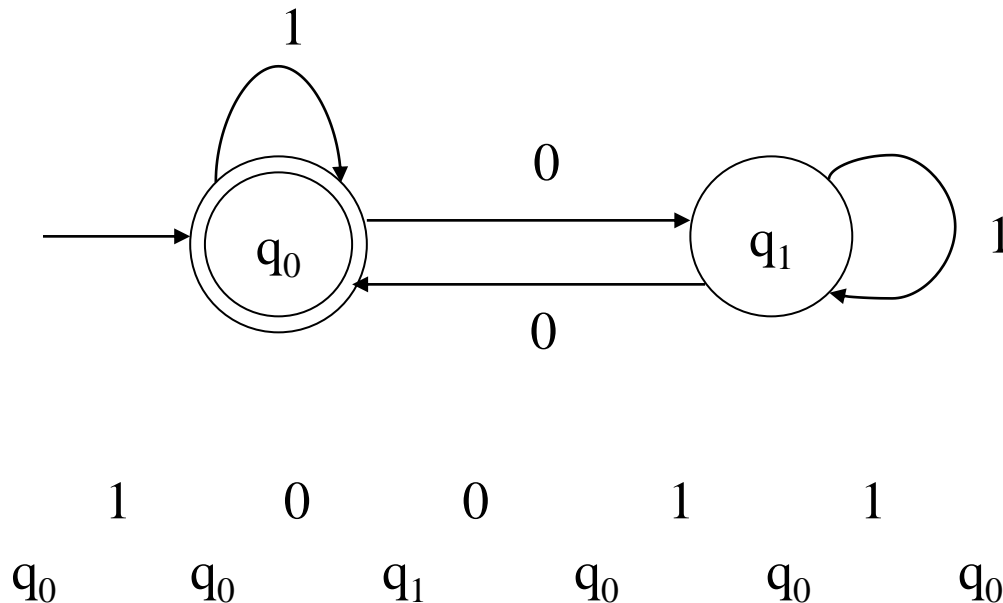
- DFA
- NFA /epsilon NFA

Deterministic Finite State Automata (DFA)



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

- The finite control can be described by a transition diagram or table:
- Example #1:

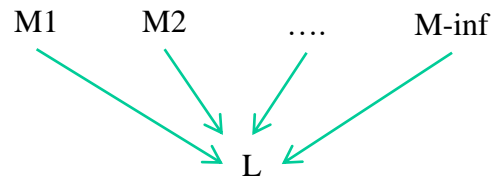


- One state is final/accepting, all others are rejecting.
- The above DFA accepts those strings that contain an even number of 0's, including the *null* string, over $\Sigma = \{0,1\}$

$$L = \{\text{all strings with zero or more 0's}\}$$
- Note, the DFA must reject all other strings

Note:

- *Machine is for accepting a language, language is the purpose!*
- *Many equivalent machines may accept the same language, but a machine cannot accept multiple languages!*

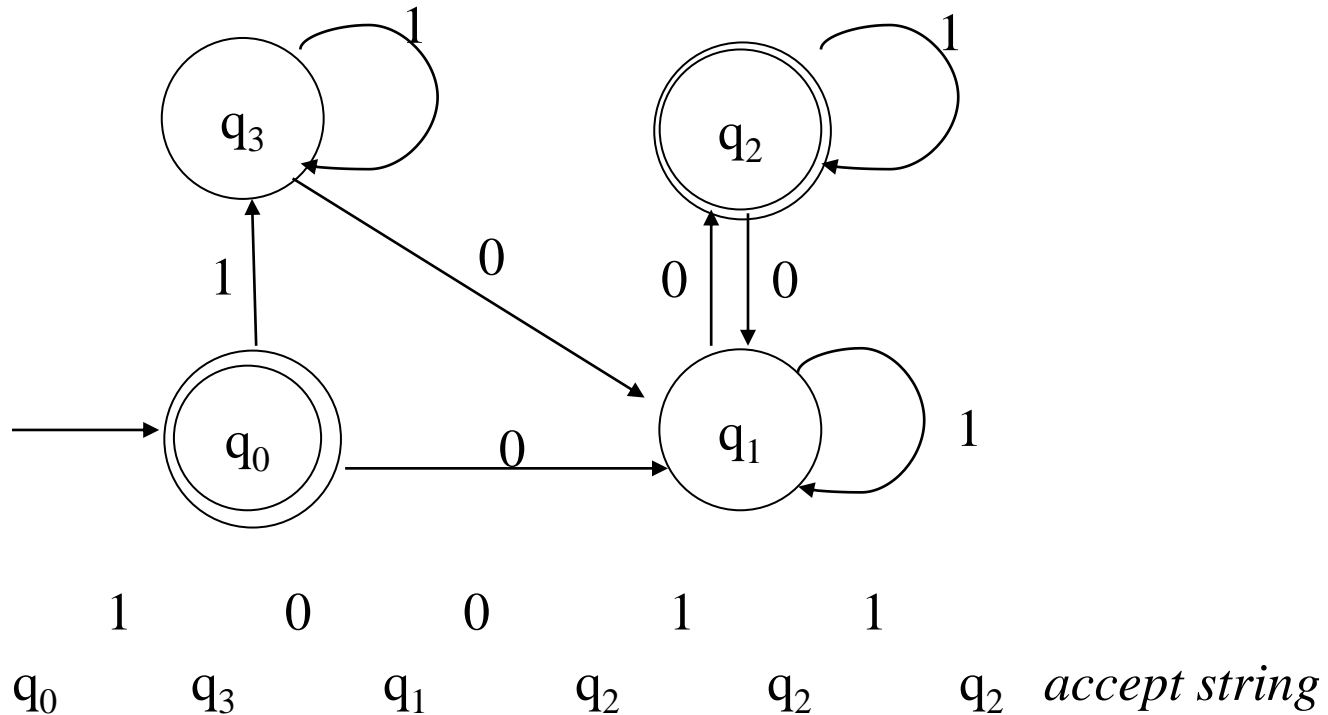


- *Id's of the characters or states are irrelevant, you can call them by any names!*

$\Sigma = \{0, 1\} \equiv \{a, b\}$

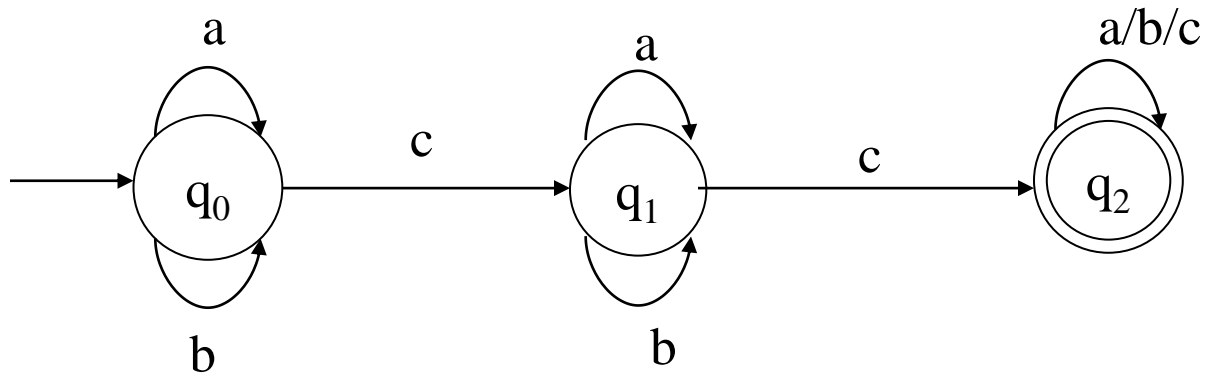
$States = \{q_0, q_1\} \equiv \{u, v\}$, as long as they have identical (isomorphic) transition table

- An equivalent machine to the previous example (DFA for even number of 0's):



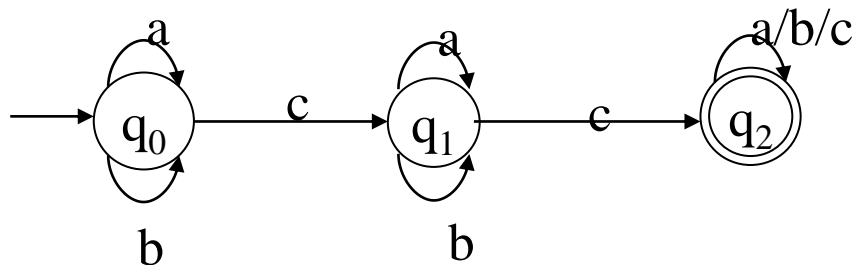
- One state is final/accepting, all others are rejecting.
- The above DFA accepts those strings that contain an even number of 0's, including null string, over $\Sigma = \{0,1\}$
- Can you draw a machine for a language by excluding the null string from the language? $L = \{\text{all strings with 2 or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

- Example #2:



	a	c	c	c	b	<u>accepted</u>
q ₀	q ₀	q ₁	q ₂	q ₂	q ₂	
	a	a	c			<u>rejected</u>
q ₀	q ₀	q ₀	q ₁			

- Accepts those strings that contain at least two *c*'s



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

Base: x a string with $|x|=0$. state will be $q_0 \Rightarrow$ rejected.

Inductive hypothesis: $|x| = \text{integer } k$, & string x is *rejected* - in state q_0 (x must have zero c),
OR, rejected - in state q_1 (x must have one c),
OR, accepted - in state q_2 (x has already with two c 's)

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

	xa	xb	xc
x ends in q_0	$q_0 \Rightarrow$ reject (still zero $c \Rightarrow$ should reject)	$q_0 \Rightarrow$ reject (still zero $c \Rightarrow$ should reject)	$q_1 \Rightarrow$ reject (still zero $c \Rightarrow$ should reject)
x ends in q_1	$q_1 \Rightarrow$ reject (still one $c \Rightarrow$ should reject)	$q_1 \Rightarrow$ reject (still one $c \Rightarrow$ should reject)	$q_2 \Rightarrow$ accept (two c now \Rightarrow should accept)
x ends in q_2	$q_2 \Rightarrow$ accept (two c already \Rightarrow should accept)	$q_2 \Rightarrow$ accept (two c already \Rightarrow should accept)	$q_2 \Rightarrow$ accept (two c already \Rightarrow should accept)

Formal Definition of a DFA

- A DFA is a five-tuple:

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

Q A finite set of states

Σ A finite input alphabet

q_0 The initial/starting state, q_0 is in Q

F A set of final/accepting states, which is a subset of Q

δ A transition function, which is a total function from $Q \times \Sigma$ to Q

$\delta: (Q \times \Sigma) \rightarrow Q$ δ is defined for any q in Q and s in Σ , and
 $\delta(q,s) = q'$ is equal to some state q' in Q , could be $q'=q$

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

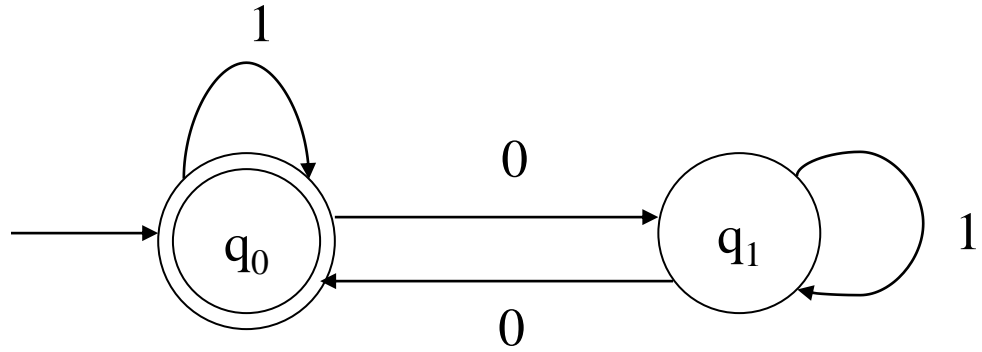
- Revisit example #1:

$Q = \{q_0, q_1\}$

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

Start state is q_0

$F = \{q_0\}$



δ :

	0	1
q_0	q_1	q_0
q_1	q_0	q_1

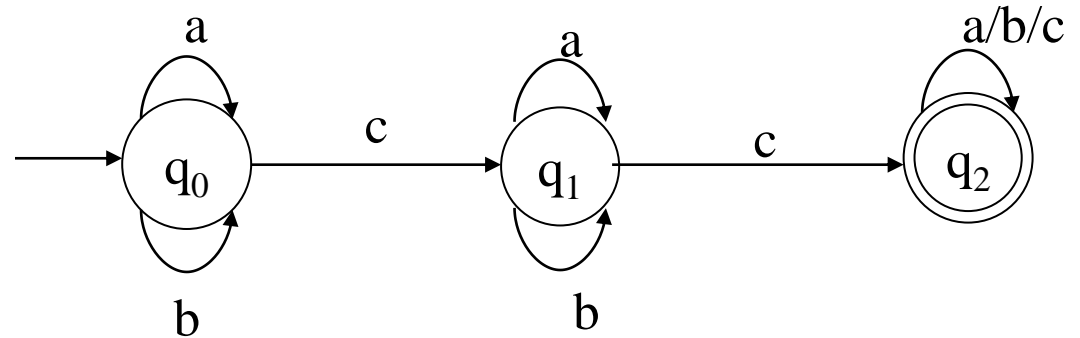
- Revisit example #2:

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

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

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.

Extension of δ to Strings

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

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

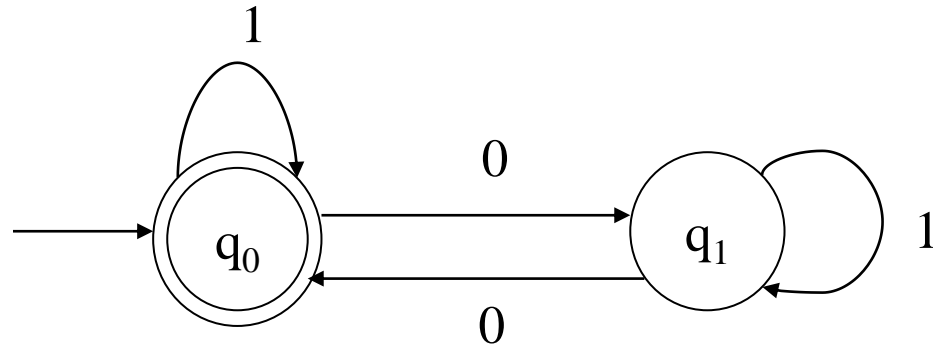
Formally:

1) $\delta^{\wedge}(q, \epsilon) = q$, and

2) For all w in Σ^* and a in Σ

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

- Recall Example #1:



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

$$\begin{aligned}
 \delta^*(q_0, 011) &= \delta(\delta^*(q_0, 01), 1) && \text{by rule \#2} \\
 &= \delta(\delta(\delta^*(q_0, 0), 1), 1) && \text{by rule \#2} \\
 &= \delta(\delta(\delta(\delta^*(q_0, \lambda), 0), 1), 1) && \text{by rule \#2} \\
 &= \delta(\delta(\delta(q_0, 0), 1), 1) && \text{by rule \#1} \\
 &= \delta(\delta(q_1, 1), 1) && \text{by definition of } \delta \\
 &= \delta(q_1, 1) && \text{by definition of } \delta \\
 &= q_1 && \text{by definition of } \delta
 \end{aligned}$$

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

- Note that:

$$\begin{aligned}\delta^{\wedge}(q, a) &= \delta(\delta^{\wedge}(q, \varepsilon), a) && \text{by definition of } \delta^{\wedge}, \text{ rule \#2} \\ &= \delta(q, a) && \text{by definition of } \delta^{\wedge}, \text{ rule \#1}\end{aligned}$$

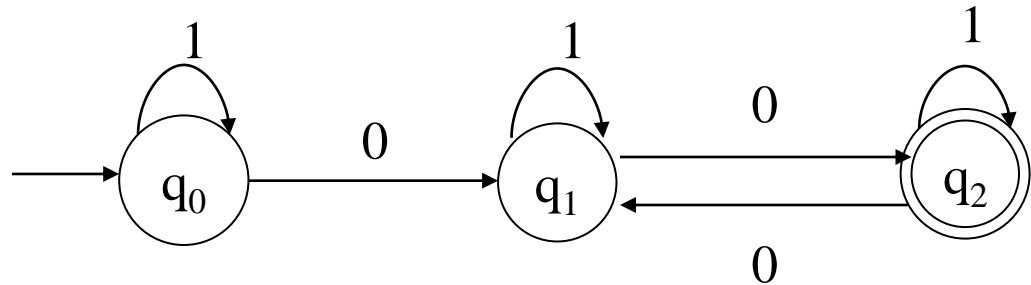
- Therefore:

$$\delta^{\wedge}(q, a_1 a_2 \dots a_n) = \delta(\delta(\dots \delta(\delta(q, a_1), a_2) \dots), a_n)$$

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

$$\delta^{\wedge}(q, a_1 a_2 \dots a_n) = \delta(q, a_1 a_2 \dots a_n)$$

- Example #3:

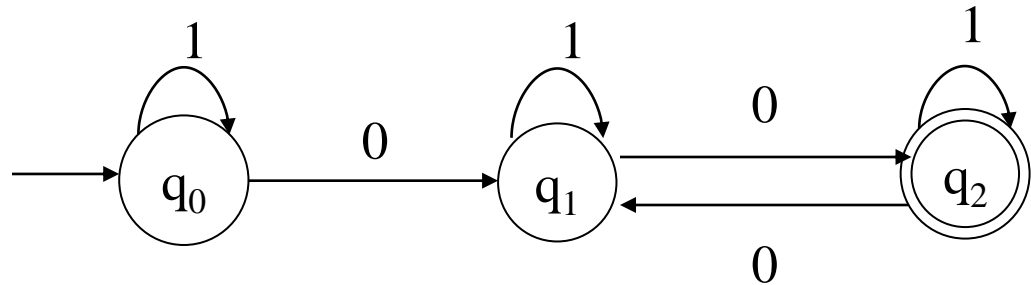


- What is $\delta(q_0, 011)$? Informally, it is the state entered by M after processing 011 having started in state q_0 .
- Formally:

$$\begin{aligned}
 \delta(q_0, 011) &= \delta(\delta(q_0, 01), 1) && \text{by rule \#2} \\
 &= \delta(\delta(\delta(q_0, 0), 1), 1) && \text{by rule \#2} \\
 &= \delta(\delta(q_1, 1), 1) && \text{by definition of } \delta \\
 &= \delta(q_1, 1) && \text{by definition of } \delta \\
 &= q_1 && \text{by definition of } \delta
 \end{aligned}$$

- Is 011 accepted? No, since $\delta(q_0, 011) = q_1$ is not a final state.
- *Language?*
- $L = \{ \text{all strings over } \{0,1\} \text{ that has 2 or more } 0 \text{ symbols} \}$

- Recall Example #3:



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

$$\begin{aligned}
 \delta(q_1, 10) &= \delta(\delta(q_1, 1), 0) && \text{by rule \#2} \\
 &= \delta(q_1, 0) && \text{by definition of } \delta \\
 &= q_2 && \text{by definition of } \delta
 \end{aligned}$$

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

Definitions related to DFAs

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

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

- Another equivalent definition:

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

- 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 $\Sigma^* - L(M)$.
 - 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

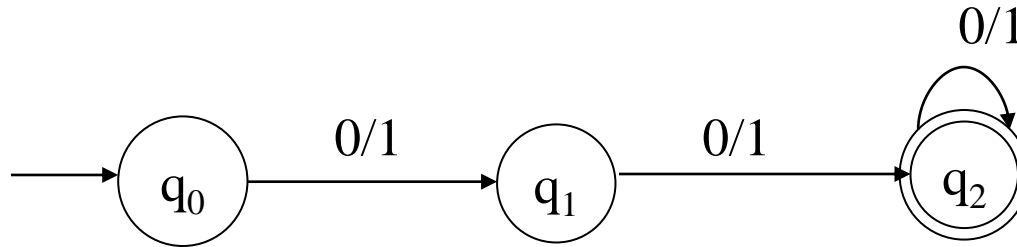
Regular: $L_1 = \{x \mid x \text{ is a string of 0's and 1's containing an even number of 1's}\}$ and

Not-regular: $L_2 = \{x \mid x = 0^n 1^n \text{ for some } n \geq 0\}$

- *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:

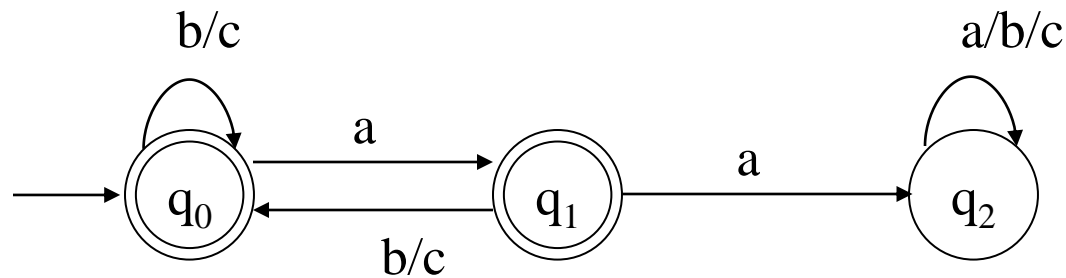
$$L(M) = \{x \mid x \text{ is a string of 0's and 1's and } |x| \geq 2\}$$



Prove this by induction

- Give a DFA M such that:

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



Logic:

In Start state (q_0): b's and c's: ignore – stay in same state

q_0 is also “accept” state

First ‘a’ appears: get ready (q_1) to reject

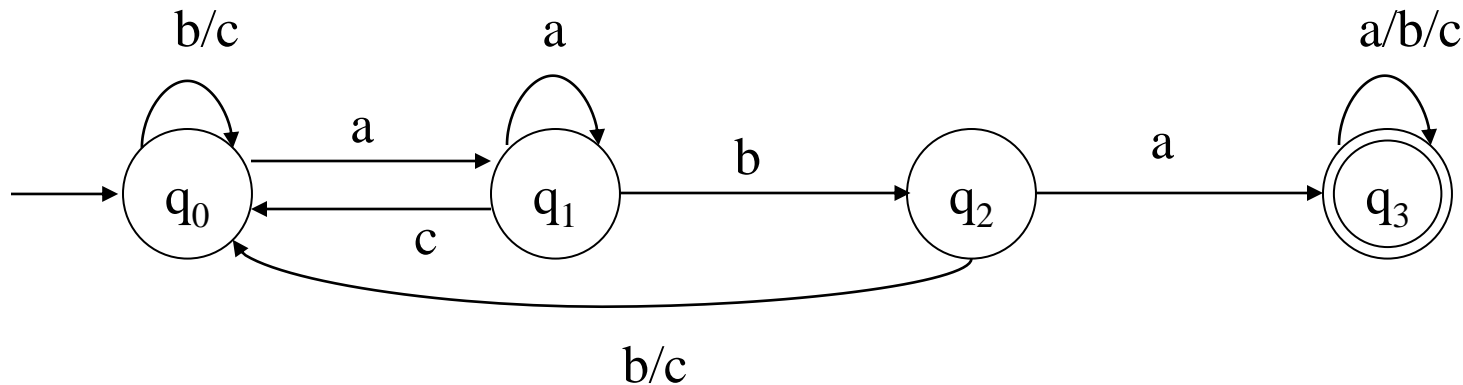
But followed by a ‘b’ or ‘c’: go back to start state q_0

When second ‘a’ appears after the “ready” state: go to reject state q_2

Ignore everything after getting to the “reject” state q_2

- Give a DFA M such that:

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



Logic: acceptance is straight forward, progressing on each expected symbol

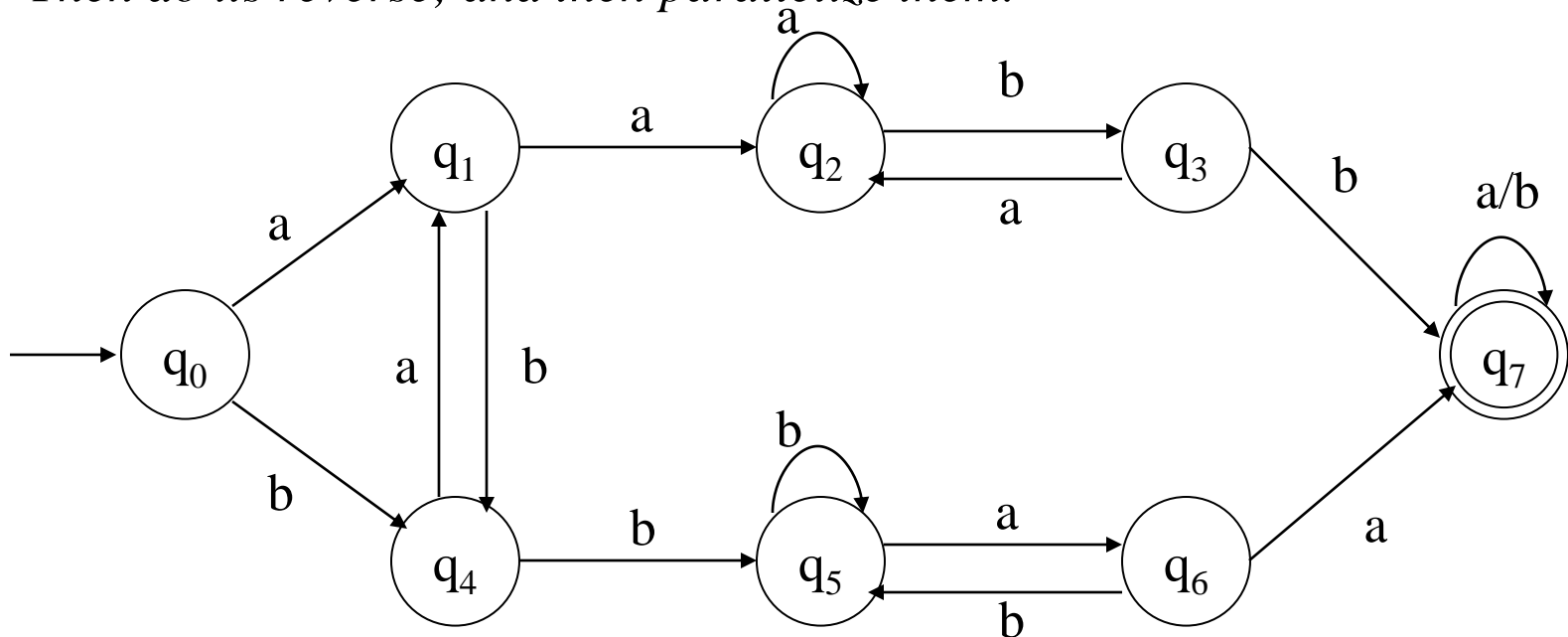
However, rejection needs special care, in each state (for DFA, we will see this becomes easier in NFA, non-deterministic machine)

- Give a DFA M such that:

$$L(M) = \{x \mid x \text{ is a string of } a\text{'s and } b\text{'s such that } x \text{ contains both } aa \text{ and } bb\}$$

First do, for a language where 'aa' comes before 'bb'

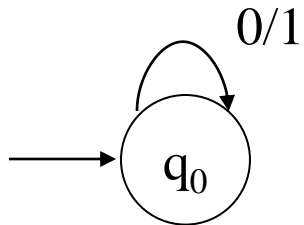
Then do its reverse; and then parallelize them.



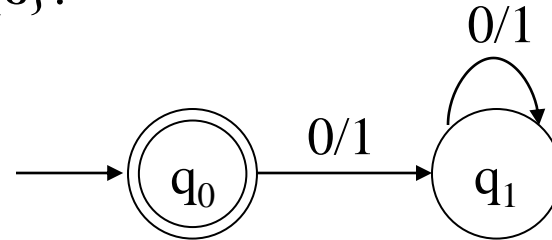
Remember, you may have multiple “final” states, but only one “start” state

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

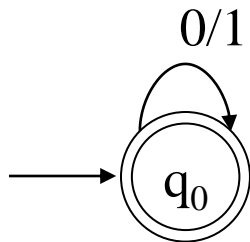
For $\{\}$:



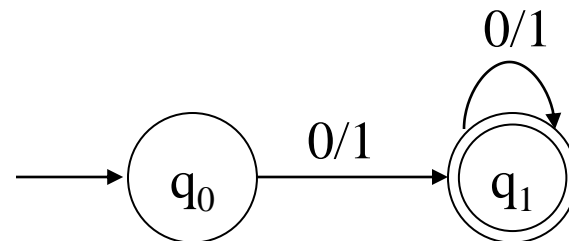
For $\{\epsilon\}$:



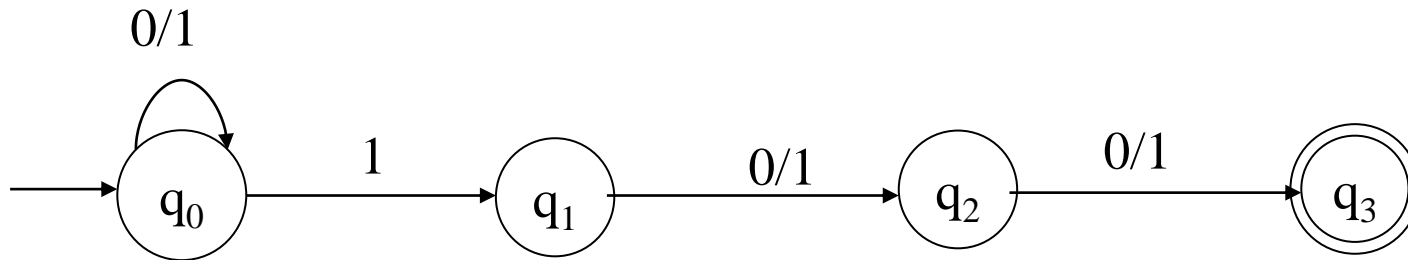
For Σ^* :



For Σ^+ :



- Problem: Third symbol from last is *1*



Is this a DFA?

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

Nondeterministic Finite State Automata (NFA)

- An NFA is a five-tuple:

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

Q A finite set of states

Σ A finite input alphabet

q_0 The initial/starting state, q_0 is in Q

F A set of final/accepting states, which is a subset of Q

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

$\delta: (Q \times \Sigma) \rightarrow 2^Q$: 2^Q is the power set of Q , the set of *all subsets* of Q
 $\delta(q,s)$: The **set of all states** p such that there is a transition labeled s from q to p

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

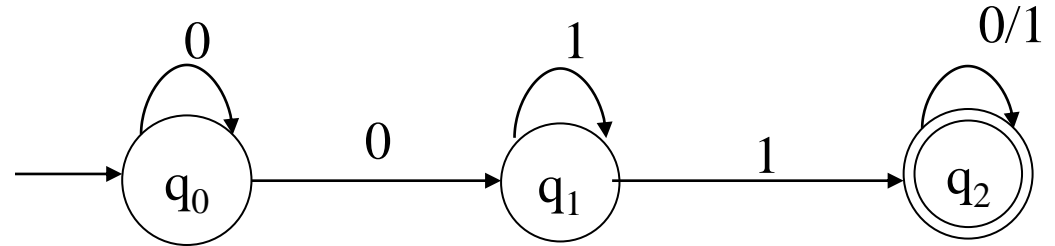
- Example #1: one or more 0's followed by one or more 1's

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

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

Start state is q_0

$F = \{q_2\}$



δ :

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

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

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

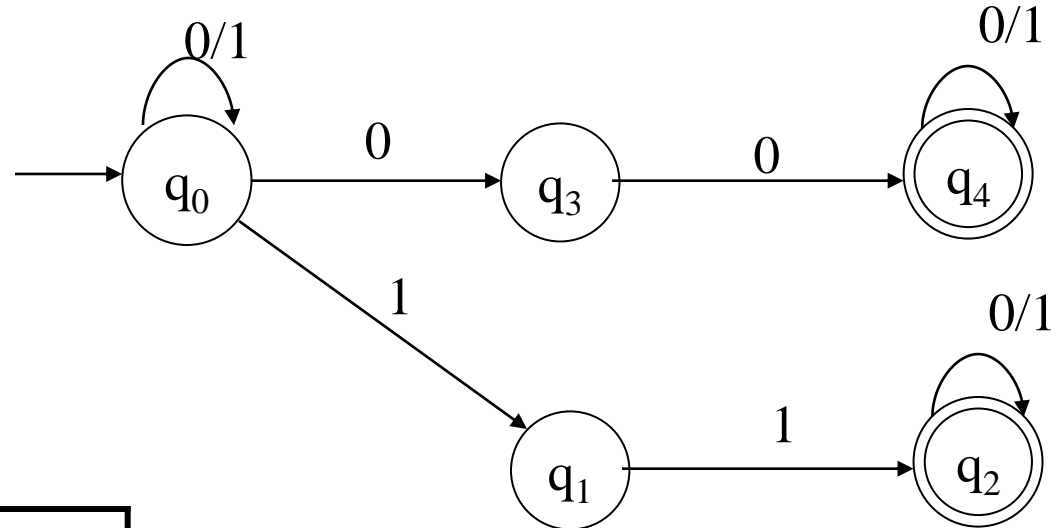
$\Sigma = \{0, 1\}$

Start state is q_0

$F = \{q_2, q_4\}$

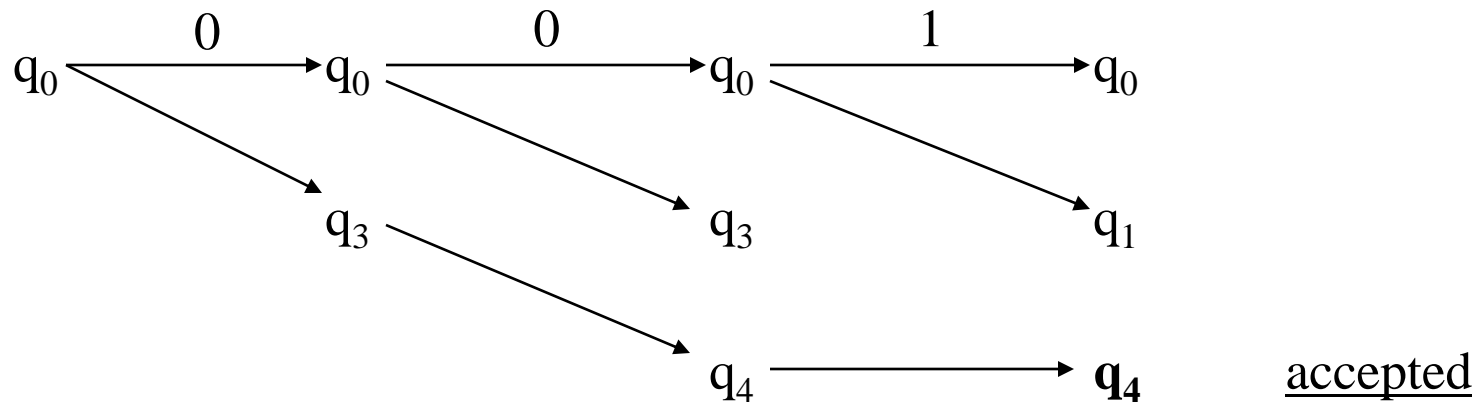
δ :

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



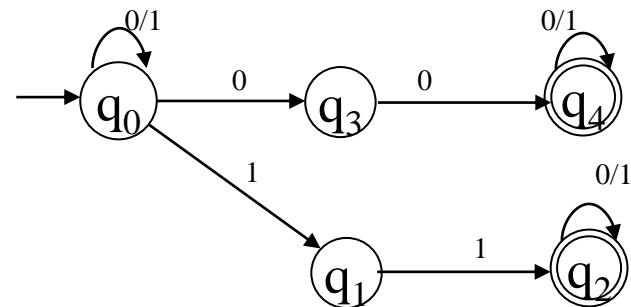
- Notes:
 - $\delta(q,s)$ may not be defined for some q and s (*what does that mean?*)
 - $\delta(q,s)$ may map to multiple q 's
 - A string is said to be accepted *if there exists* a path from q_0 to some state in F
 - A string is rejected *if there exist NO path* to any state in F
 - The language accepted 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 or not NFA is easier to develop, or how does one implement NFA

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

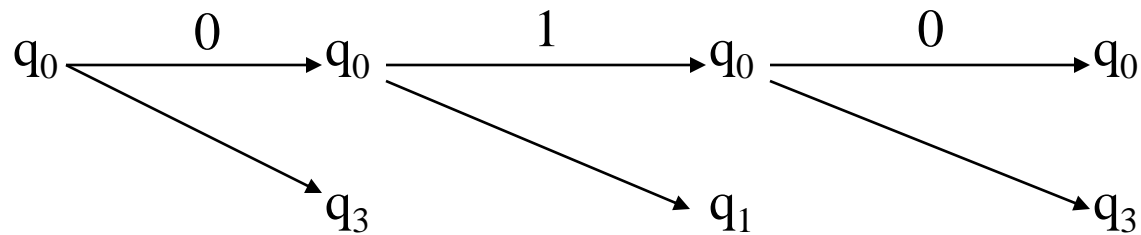


- Each level will have at most n states:

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

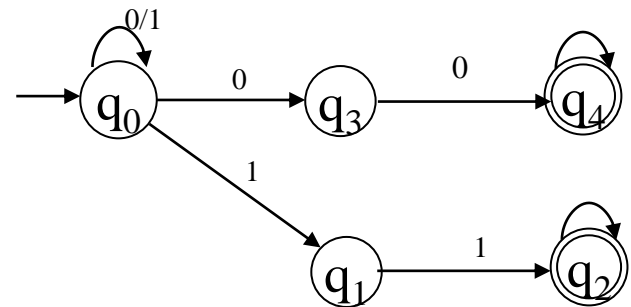


- Another example (010):



not accepted

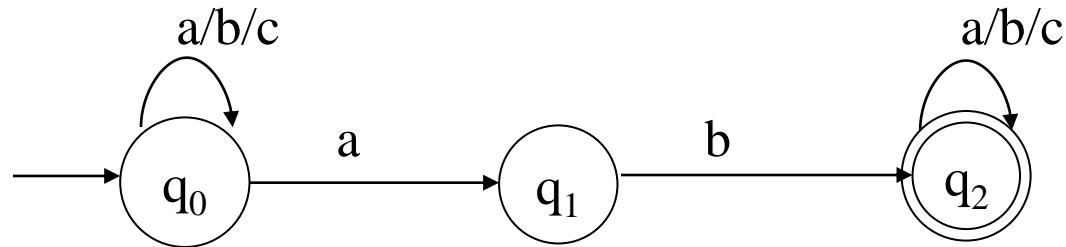
- All paths have been explored, and none lead to an accepting state.



- 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 $\Sigma = \{a, b, c\}$. Give an NFA M that accepts:

$$L = \{x \mid x \text{ is in } \Sigma^* \text{ and } x \text{ contains } ab\}$$



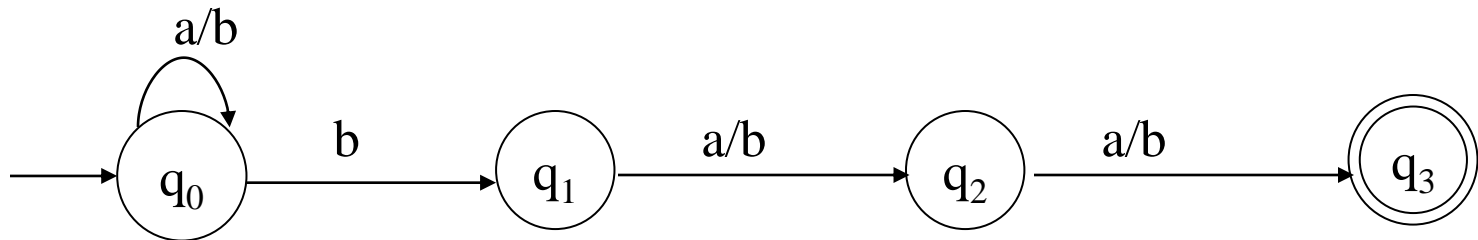
Is L a subset of $L(M)$? Or, does M accept all strings in L ?

Is $L(M)$ a subset of L ? Or, does M reject 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 $\Sigma = \{a, b\}$. Give an NFA M that accepts:

$$L = \{x \mid x \text{ is in } \Sigma^* \text{ and the third to the last symbol in } x \text{ is } b\}$$



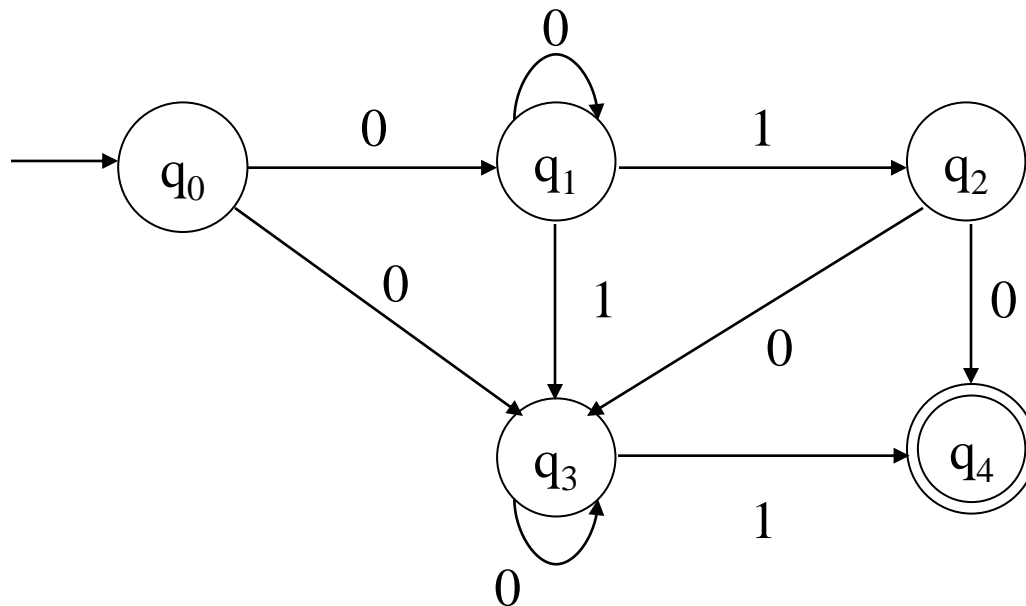
Is L a subset of $L(M)$?

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

- Give an equivalent DFA as an exercise.

Extension of δ to Strings and Sets of States

- What we currently have: $\delta : (Q \times \Sigma) \rightarrow 2^Q$
- What we want (why?): $\delta : (2^Q \times \Sigma^*) \rightarrow 2^Q$
- We will do this in two steps, which will be slightly different from the book, and we will make use of the following NFA.



Extension of δ to Strings and Sets of States

- Step #1:

Given $\delta: (Q \times \Sigma) \rightarrow 2^Q$ define $\delta^\#: (2^Q \times \Sigma) \rightarrow 2^Q$ as follows:

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

- Note that:

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

- Hence, we can use δ for $\delta^\#$

$$\begin{aligned} &\delta(\{q_0, q_2\}, 0) \\ &\delta(\{q_0, q_1, q_2\}, 0) \end{aligned}$$

These now make sense, but previously they did not.

- Example:

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

$$\begin{aligned}\delta(\{q_0, q_1, q_2\}, 1) &= \delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1) \\ &= \{\} \cup \{q_2, q_3\} \cup \{\} \\ &= \{q_2, q_3\}\end{aligned}$$

- Step #2:

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

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

Formally:

$$\begin{aligned} 2) \delta^*(R, \epsilon) &= R && \text{for any subset } R \text{ of } Q \\ 3) \delta^*(R, wa) &= \delta(\delta^*(R, w), a) && \text{for any } w \text{ in } \Sigma^*, a \text{ in } \Sigma, \text{ and} \\ &&& \text{subset } R \text{ of } Q \end{aligned}$$

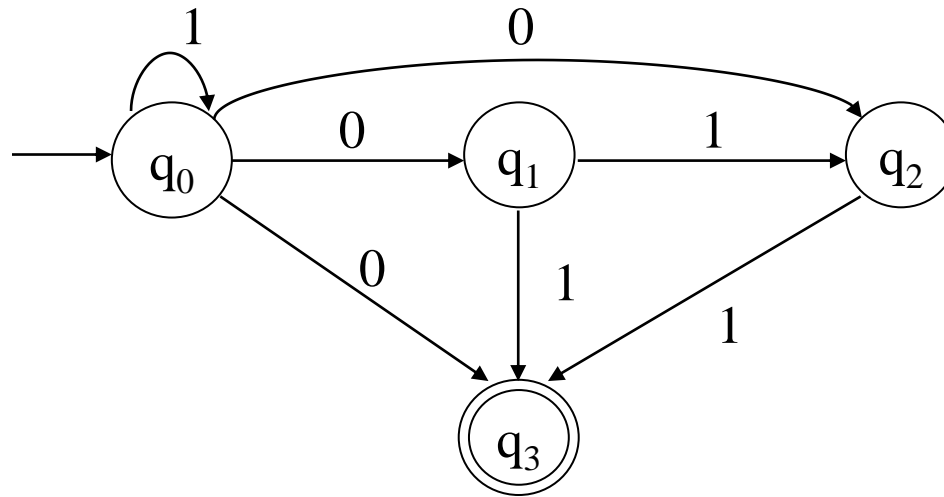
- Note that:

$$\begin{aligned} \delta^*(R, a) &= \delta(\delta^*(R, \epsilon), a) && \text{by definition of } \delta^*, \text{ rule \#3 above} \\ &= \delta(R, a) && \text{by definition of } \delta^*, \text{ rule \#2 above} \end{aligned}$$

- Hence, we can use δ for δ^*

$$\begin{aligned} \delta(\{q_0, q_2\}, 0110) &&& \text{These now make sense, but previously} \\ \delta(\{q_0, q_1, q_2\}, 101101) &&& \text{they did not.} \end{aligned}$$

- Example:



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

Informally: The set of states the NFA could be in after processing 10, having started in state q_0 , i.e., $\{q_1, q_2, q_3\}$.

Formally:

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

Is 10 accepted? Yes!

- Example:

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

$$\begin{aligned}\delta(\{q_0, q_1\}, 1) &= \delta(\{q_0\}, 1) \cup \delta(\{q_1\}, 1) \\ &= \{q_0\} \cup \{q_2, q_3\} \\ &= \{q_0, q_2, q_3\}\end{aligned}$$

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

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

- Example:

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

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

Definitions for NFAs

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

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

- Another equivalent definition:

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

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?

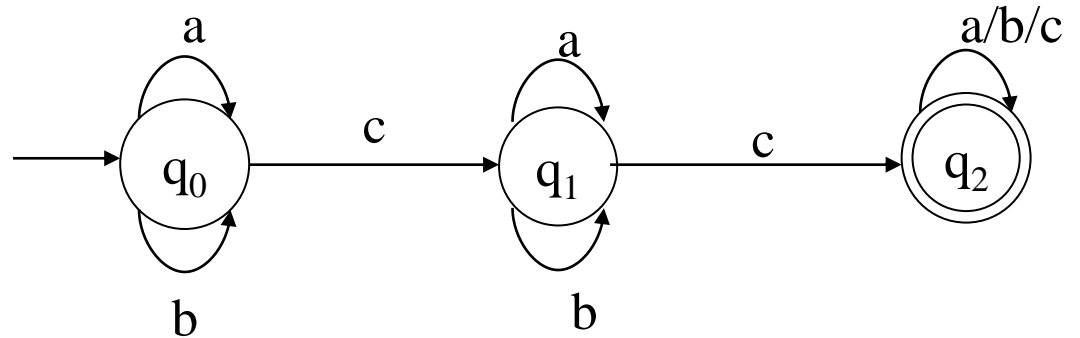
- Consider the following DFA: 2 or more c's

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

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

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

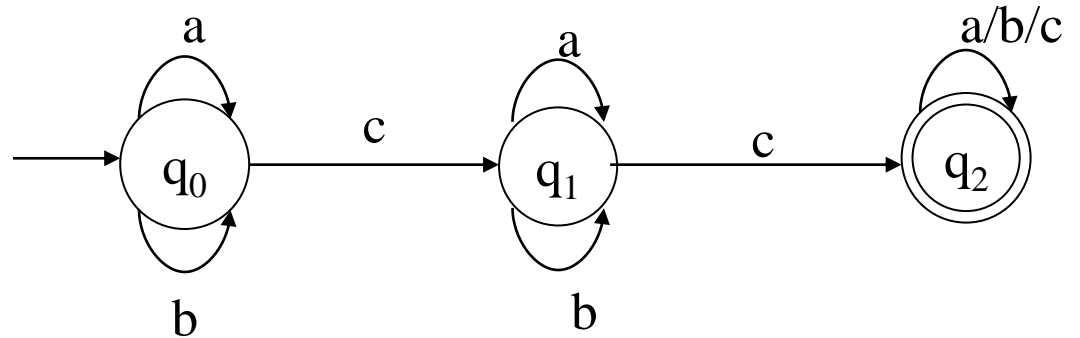
- An Equivalent NFA:

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

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

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

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

$$\begin{aligned} Q' &= 2^Q && \text{Each state in } M' \text{ corresponds to a} \\ &= \{Q_0, Q_1, \dots\} && \text{subset of states from } M \end{aligned}$$

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

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

$$q'_0 = [q_0]$$

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

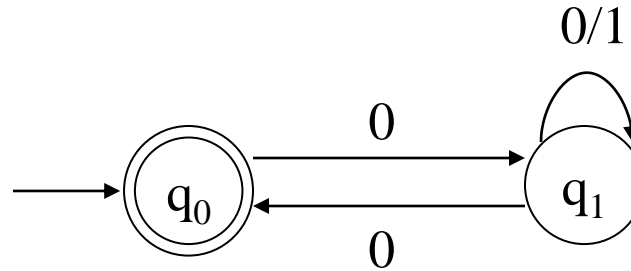
- Example: empty string or start and end with 0

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

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

Start state is q_0

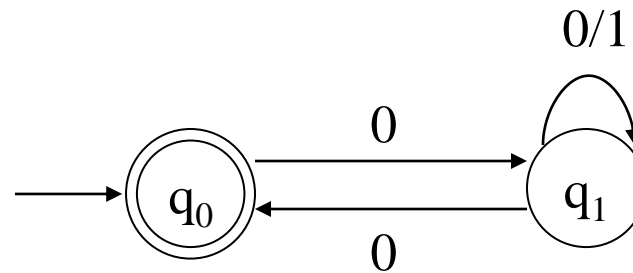
$$F = \{q_0\}$$



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):

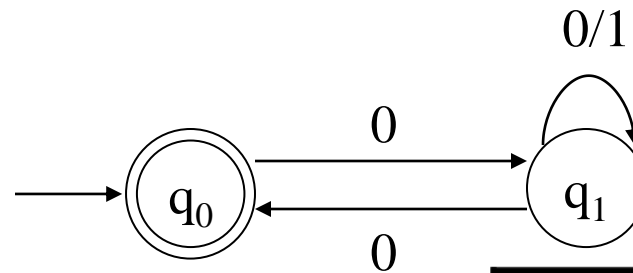


δ for DFA:

	0	1
$\rightarrow q_0$	$\{q_1\}$ write as $[q_1]$	$\{\}$ write as $[]$
$[q_1]$		
$[]$		

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

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

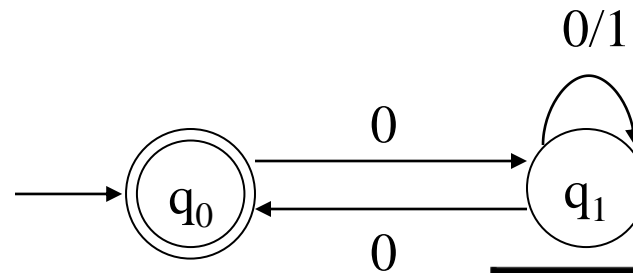


$\{q_1\}$	$\{\}$
$\{q_0, q_1\}$	$\{q_1\}$

δ :

	0	1
$\rightarrow q_0$	$\{q_1\}$ 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):

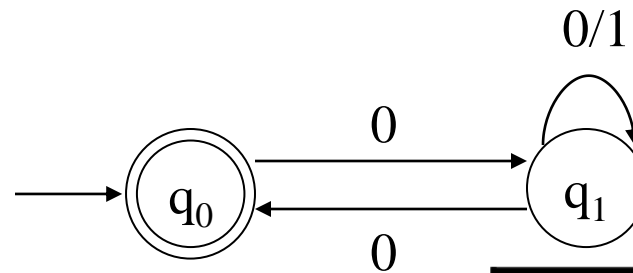


$\{q_1\}$	$\{\}$
$\{q_0, q_1\}$	$\{q_1\}$

δ :

	0	1
$\rightarrow q_0$	$\{q_1\}$ 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):

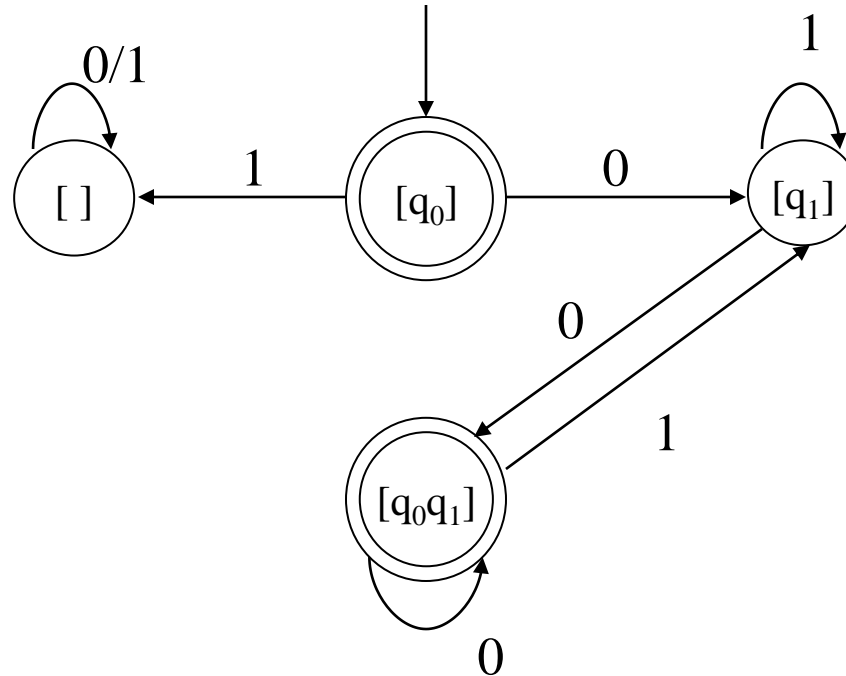


$\{q_1\}$	$\{\}$
$\{q_0, q_1\}$	$\{q_1\}$

δ :

	0	1
$\rightarrow q_0$	$\{q_1\}$ 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:



$\delta(\{q_0\}, 0) = \{q_1\}$	\Rightarrow	$\delta'([q_0], 0) = [q_1]$
$\delta(\{q_0\}, 1) = \{\}$	\Rightarrow	$\delta'([q_0], 1) = []$
$\delta(\{q_1\}, 0) = \{q_0, q_1\}$	\Rightarrow	$\delta'([q_1], 0) = [q_0q_1]$
$\delta(\{q_1\}, 1) = \{q_1\}$	\Rightarrow	$\delta'([q_1], 1) = [q_1]$
$\delta(\{q_0, q_1\}, 0) = \{q_0, q_1\}$	\Rightarrow	$\delta'([q_0q_1], 0) = [q_0q_1]$
$\delta(\{q_0, q_1\}, 1) = \{q_1\}$	\Rightarrow	$\delta'([q_0q_1], 1) = [q_1]$
$\delta(\{\}, 0) = \{\}$	\Rightarrow	$\delta'([], 0) = []$
$\delta(\{\}, 1) = \{\}$	\Rightarrow	$\delta'([], 1) = []$

- **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 $R = \{ \}$

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

- Exercise - Convert the following NFA to a DFA:

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

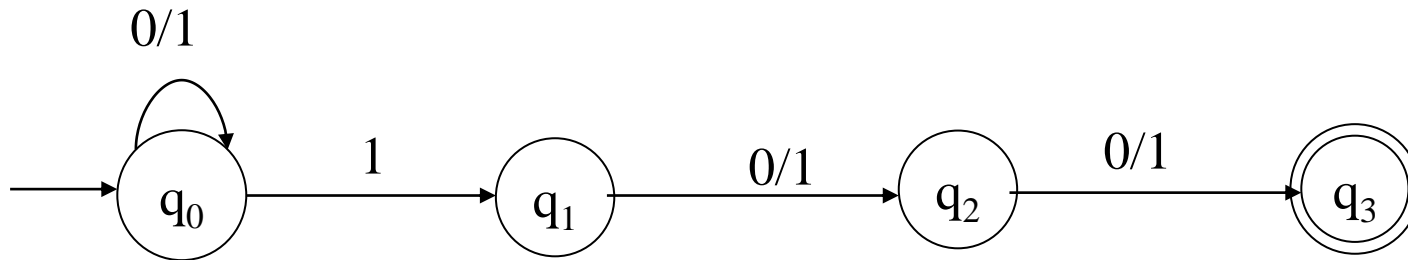
$$\Sigma = \{0, 1\}$$

Start state is q_0

$$F = \{q_0\}$$

$\delta:$	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?

NFAs with ε Moves

- An NFA- ε is a five-tuple:

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

Q A finite set of states

Σ A finite input alphabet

q_0 The initial/starting state, q_0 is in Q

F A set of final/accepting states, which is a subset of Q

δ A transition function, which is a total function from $Q \times \Sigma \cup \{\varepsilon\}$ to 2^Q

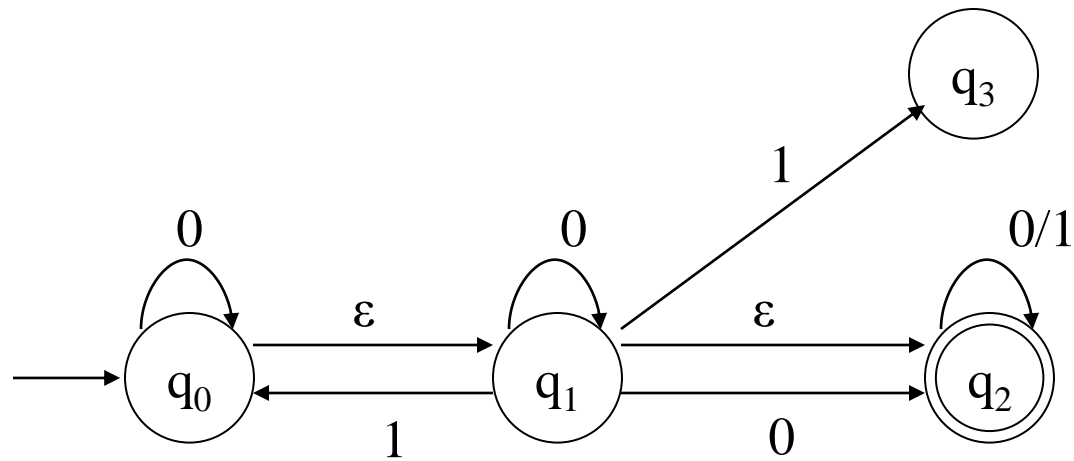
$$\delta: (Q \times (\Sigma \cup \{\varepsilon\})) \rightarrow 2^Q$$

$$\delta(q, s)$$

-The set of all states p such that there is a transition labeled a from q to p , where a is in $\Sigma \cup \{\varepsilon\}$

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

- Example:



δ :

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

- A string $w = w_1w_2\dots w_n$ is processed as $w = \epsilon^*w_1\epsilon^*w_2\epsilon^*\dots\epsilon^*w_n\epsilon^*$
- Example: all computations on 00:

$0 \quad \epsilon \quad 0$
 $q_0 \quad q_0 \quad q_1 \quad q_2$
 \vdots

Informal Definitions

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA- ϵ .
- A String w in Σ^* is *accepted* by M iff there exists a path in M from q_0 to a state in F labeled by w and zero or more ϵ transitions.
- The language accepted by M is the set of all strings from Σ^* that are accepted by M .

ϵ -closure

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

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

$$\epsilon\text{-closure}(q_2) = \{q_2\}$$

$$\epsilon\text{-closure}(q_1) = \{q_1, q_2\}$$

$$\epsilon\text{-closure}(q_3) = \{q_3\}$$

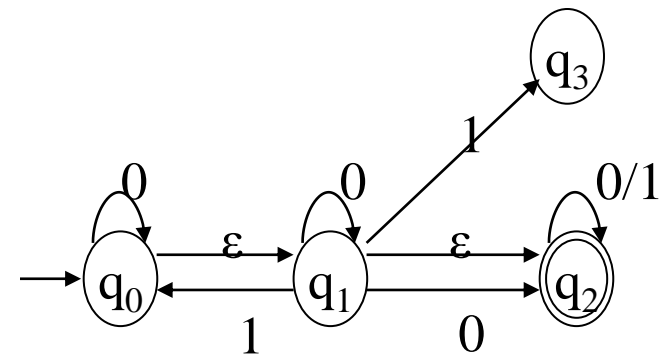
- ϵ -closure(q) can be extended to sets of states by defining:

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

- Examples:

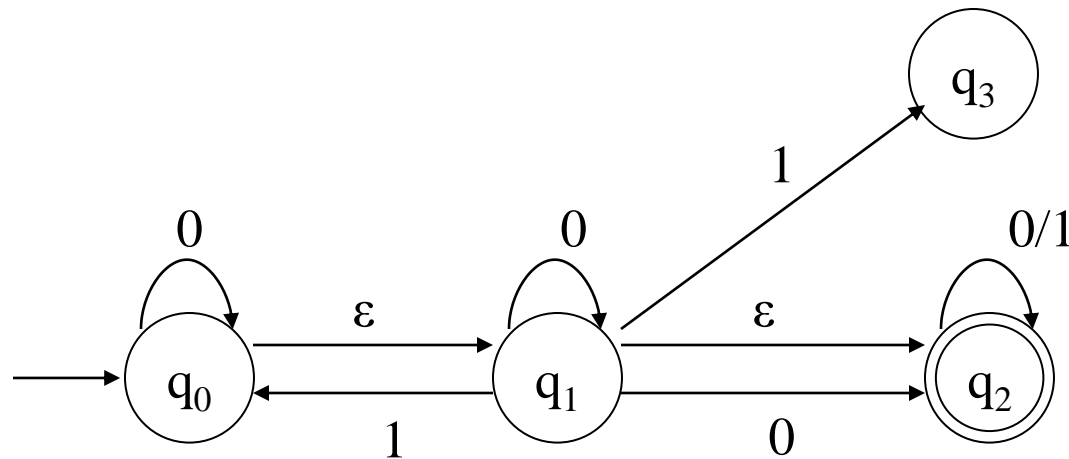
$$\epsilon\text{-closure}(\{q_1, q_2\}) = \{q_1, q_2\}$$

$$\epsilon\text{-closure}(\{q_0, q_3\}) = \{q_0, q_1, q_2, q_3\}$$



Extension of δ to Strings and Sets of States

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

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

- Note that:

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

- Hence, we can use δ for $\delta^\#$

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

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

These now make sense, but previously they did not.

- Examples:

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

$$\begin{aligned}\delta(\{q_0, q_1, q_2\}, 1) &= \delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1) \\ &= \{ \} \cup \{q_0, q_3\} \cup \{q_2\} \\ &= \{q_0, q_2, q_3\}\end{aligned}$$

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

$$\begin{aligned}\delta(\{q_0, q_1\}, 0) &= \delta(q_0, 0) \cup \delta(q_1, 0) \\ &= \{q_0\} \cup \{q_1, q_2\} \\ &= \{q_0, q_1, q_2\}\end{aligned}$$

- Step #2:

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

$\delta^*(R, w)$ – The set of states M could be in after processing string w , having starting from any state in R .

Formally:

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

- Can we use δ for δ^* ?

- Consider the following example:

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

$$\begin{aligned}
 \delta^(\{q_0\}, 0) &= \varepsilon\text{-closure}(\delta(\delta^(\{q_0\}, \varepsilon), 0)) && \text{By rule \#3} \\
 &= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(\{q_0\}), 0)) && \text{By rule \#2} \\
 &= \varepsilon\text{-closure}(\delta(\{q_0, q_1, q_2\}, 0)) && \text{By } \varepsilon\text{-closure} \\
 &= \varepsilon\text{-closure}(\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0)) && \text{By rule \#1} \\
 &= \varepsilon\text{-closure}(\{q_0\} \cup \{q_1, q_2\} \cup \{q_2\}) \\
 &= \varepsilon\text{-closure}(\{q_0, q_1, q_2\}) \\
 &= \varepsilon\text{-closure}(\{q_0\}) \cup \varepsilon\text{-closure}(\{q_1\}) \cup \varepsilon\text{-closure}(\{q_2\}) \\
 &= \{q_0, q_1, q_2\} \cup \{q_1, q_2\} \cup \{q_2\} \\
 &= \{q_0, q_1, q_2\}
 \end{aligned}$$

- So what is the difference?

$\delta(q_0, 0)$ - Processes 0 as a single symbol, without ε transitions.
 $\delta^(\{q_0\}, 0)$ - Processes 0 using as many ε transitions as are possible.

- Example:

$$\begin{aligned}
\delta^(\{q_0\}, 01) &= \varepsilon\text{-closure}(\delta(\delta^(\{q_0\}, 0), 1)) && \text{By rule \#3} \\
&= \varepsilon\text{-closure}(\delta(\{q_0, q_1, q_2\}), 1) && \text{Previous slide} \\
&= \varepsilon\text{-closure}(\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1)) && \text{By rule \#1} \\
&= \varepsilon\text{-closure}(\{ \} \cup \{q_0, q_3\} \cup \{q_2\}) \\
&= \varepsilon\text{-closure}(\{q_0, q_2, q_3\}) \\
&= \varepsilon\text{-closure}(\{q_0\}) \cup \varepsilon\text{-closure}(\{q_2\}) \cup \varepsilon\text{-closure}(\{q_3\}) \\
&= \{q_0, q_1, q_2\} \cup \{q_2\} \cup \{q_3\} \\
&= \{q_0, q_1, q_2, q_3\}
\end{aligned}$$

Definitions for NFA- ϵ Machines

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA- ϵ and let w be in Σ^* . Then w is *accepted* by M iff $\delta^+(\{q_0\}, w)$ 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 the set:

$$L(M) = \{w \mid w \text{ is in } \Sigma^* \text{ and } \delta^+(\{q_0\}, w) \text{ contains at least one state in } F\}$$

- Another equivalent definition:

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

Equivalence of NFAs and NFA- ϵ s

- 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:

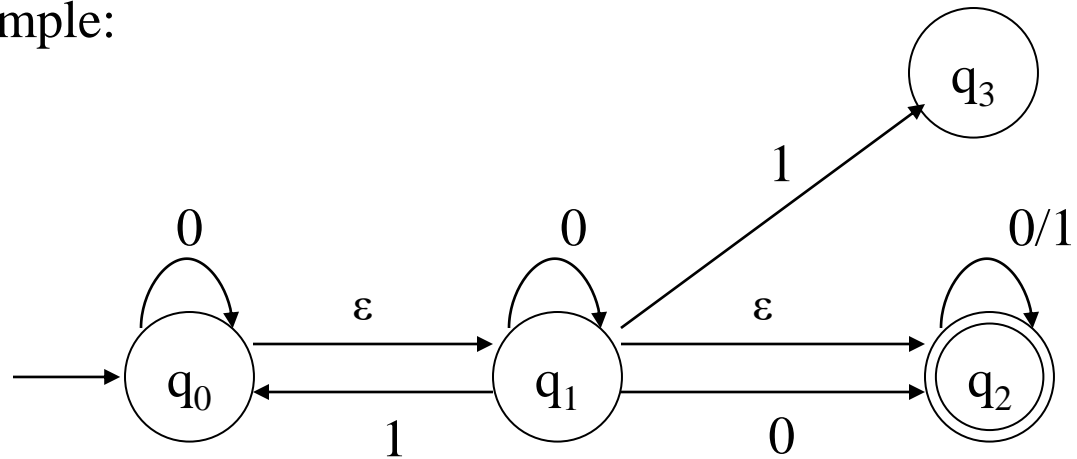
$F' = F \cup \{q\}$ if ϵ -closure(q) contains at least one state from F

$F' = F$ otherwise

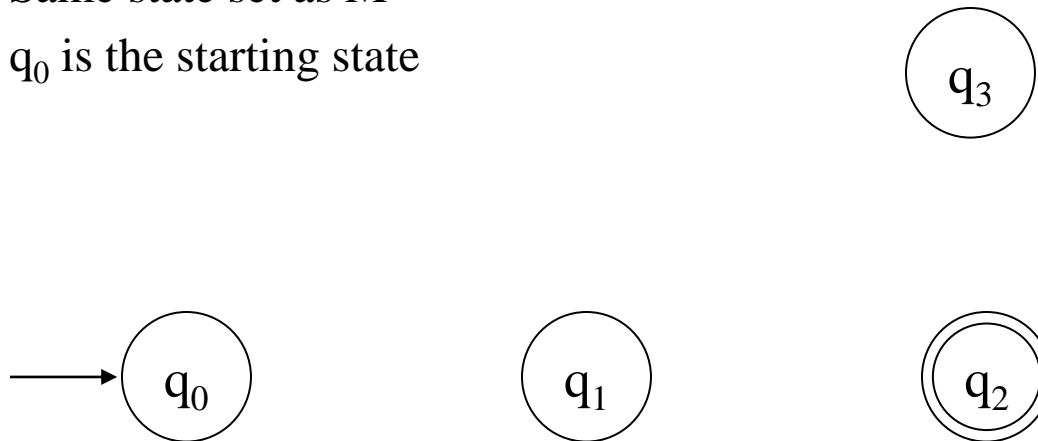
$\delta'(q, a) = \delta^+(q, a)$ - for all q in Q and a in Σ

- 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
 - M' has no ϵ transitions

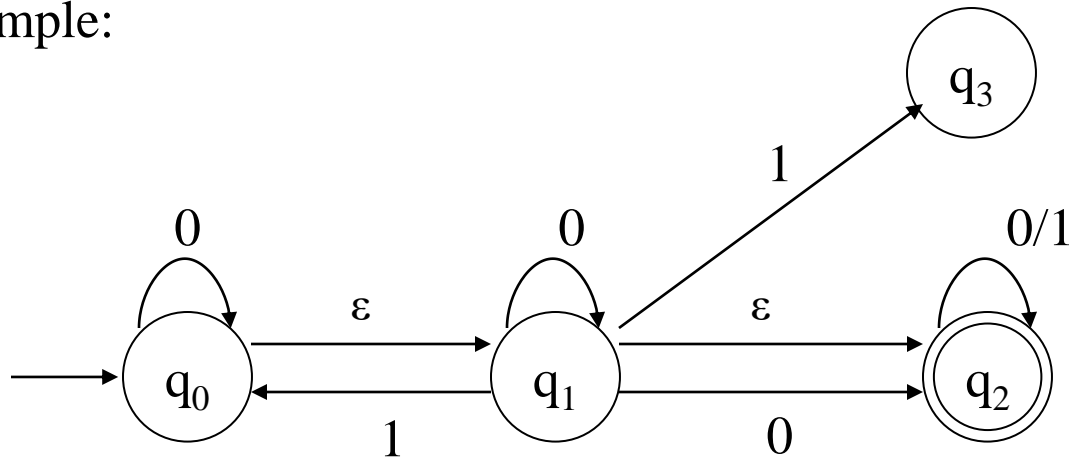
- Example:



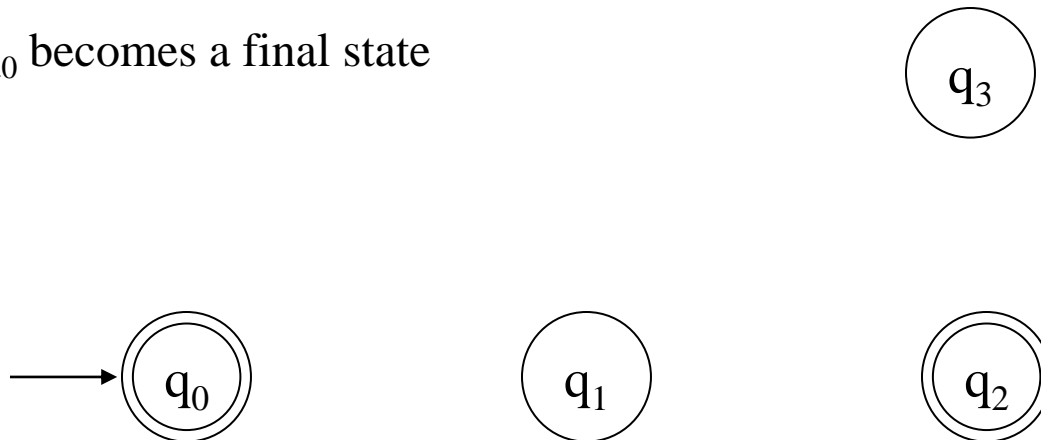
- Step #1:
 - Same state set as M
 - q_0 is the starting state



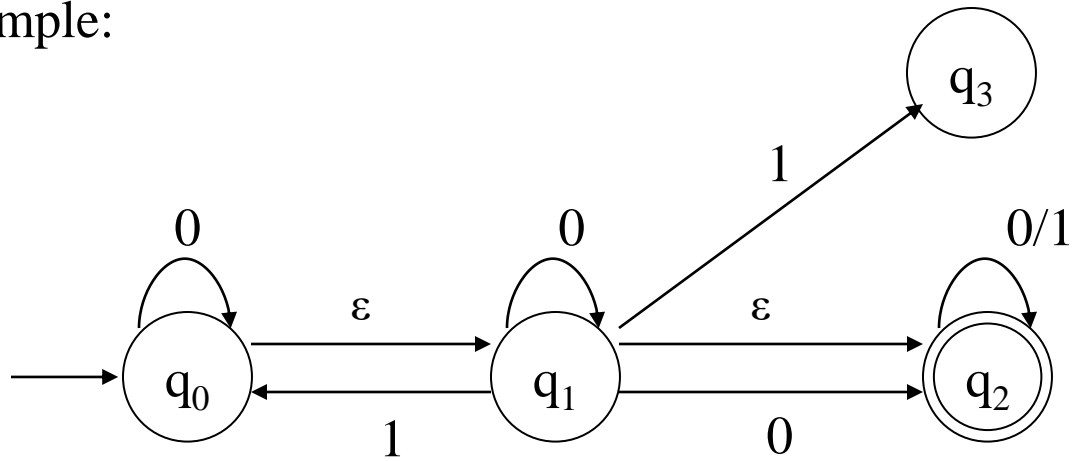
- Example:



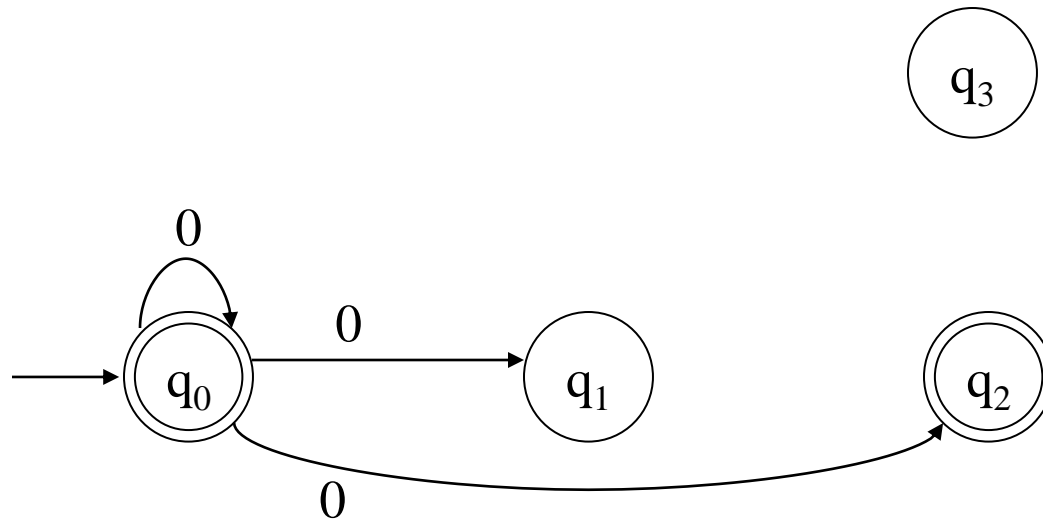
- Step #2:
 - q_0 becomes a final state



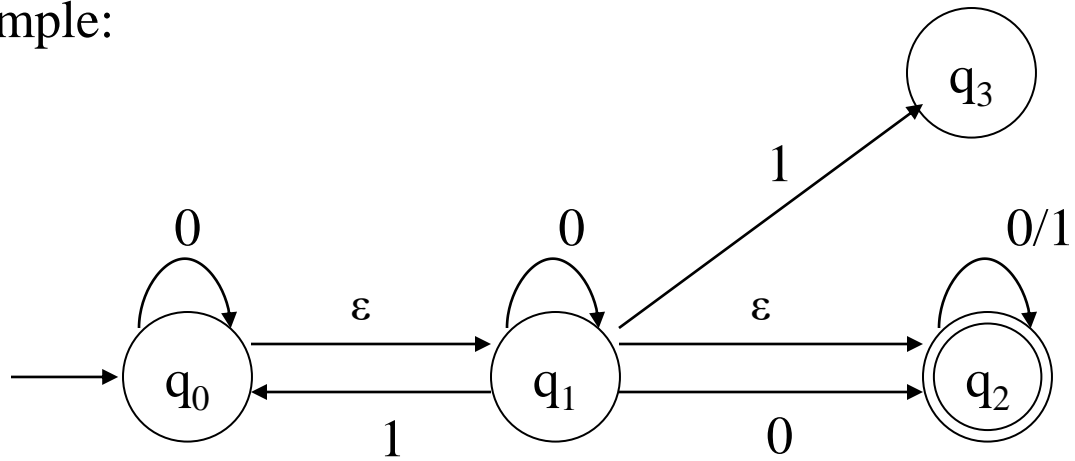
- Example:



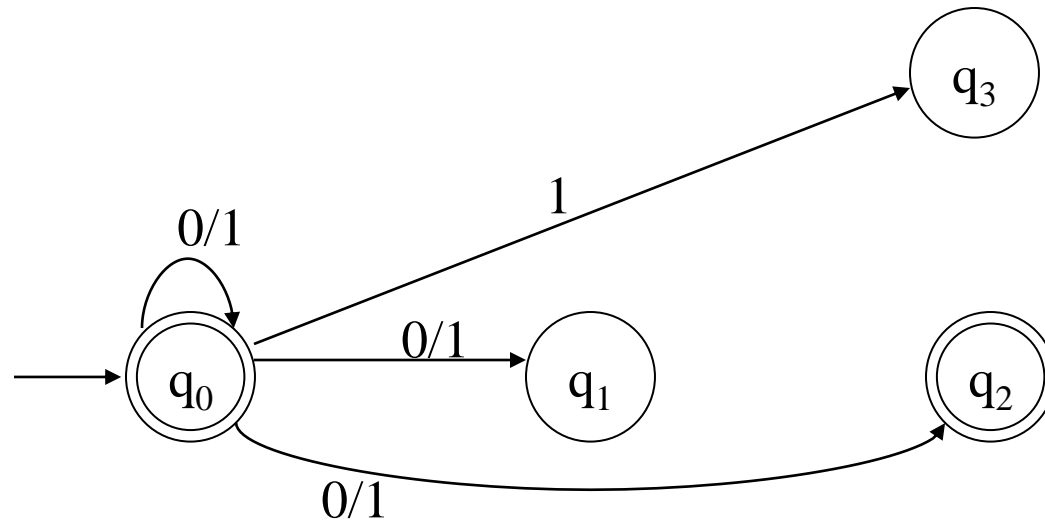
- Step #3:



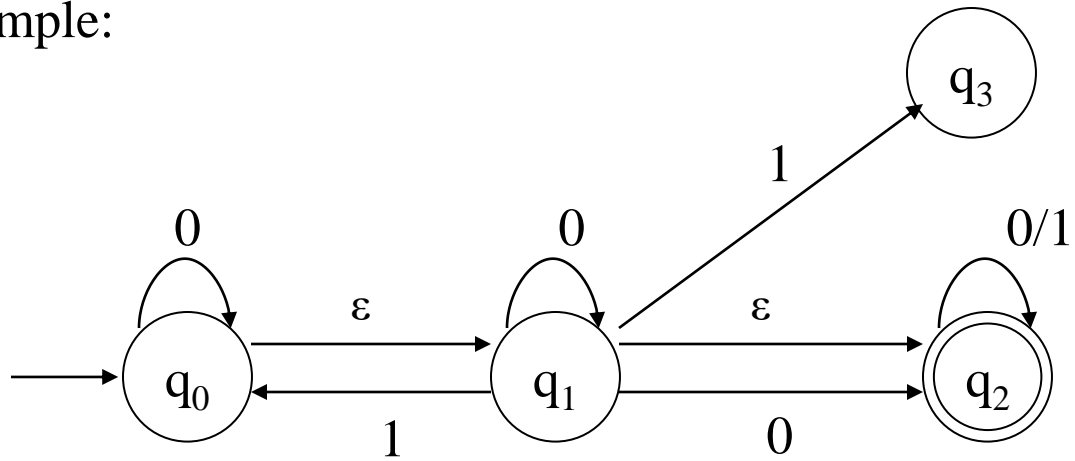
- Example:



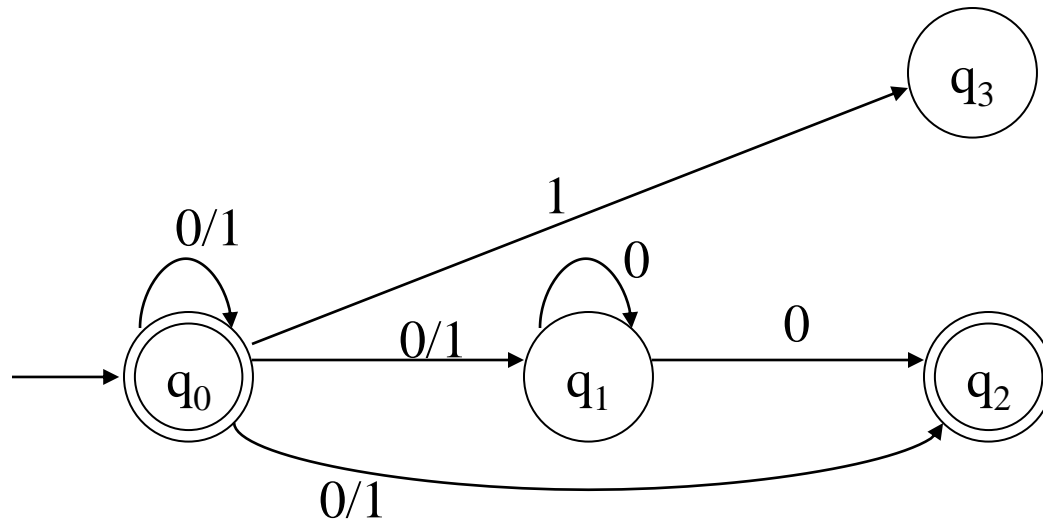
- Step #4:



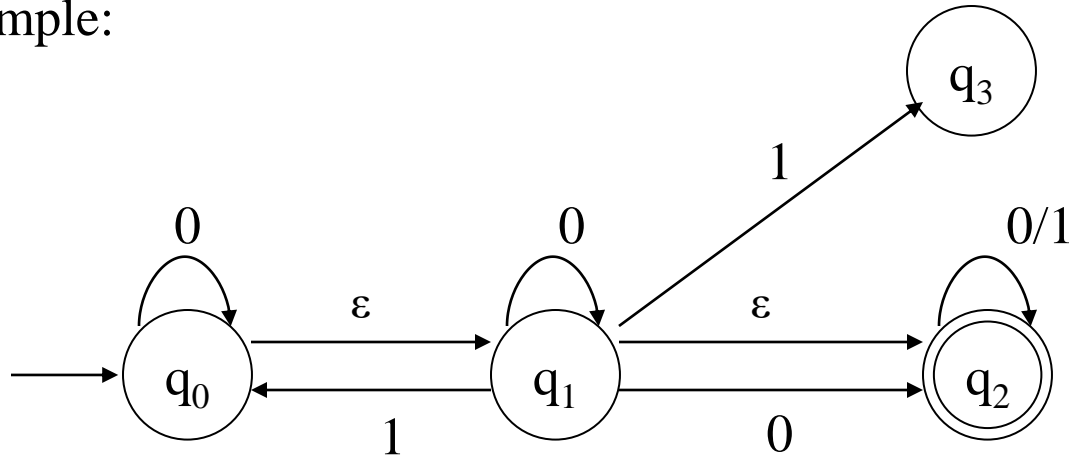
- Example:



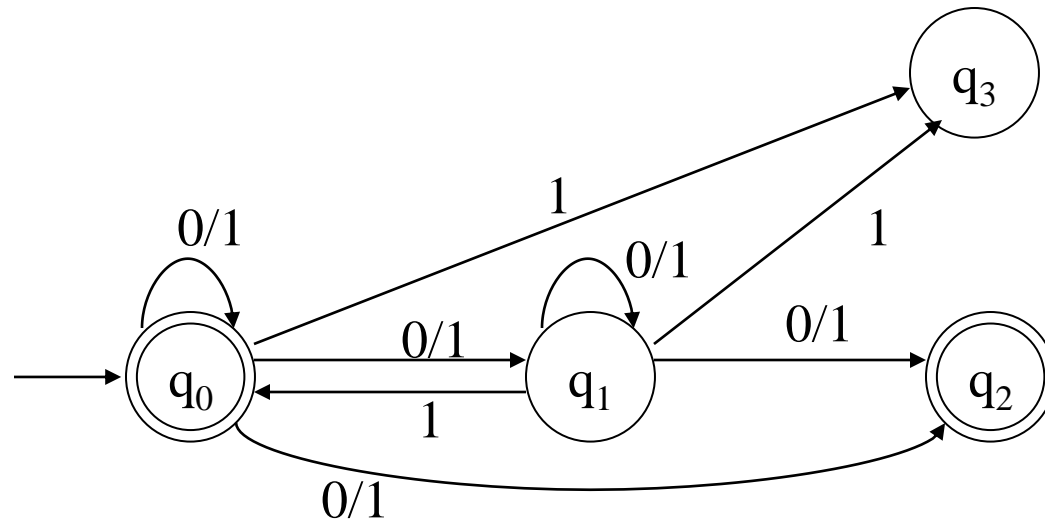
- Step #5:



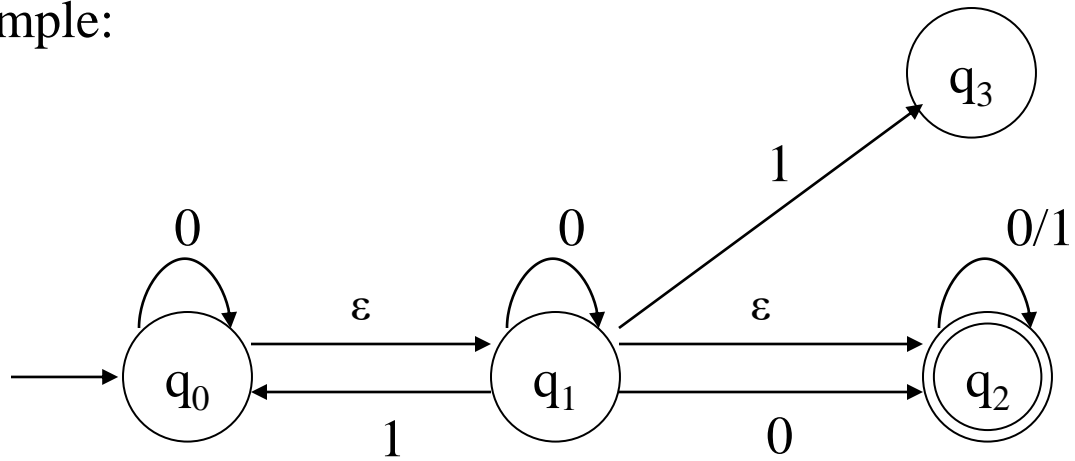
- Example:



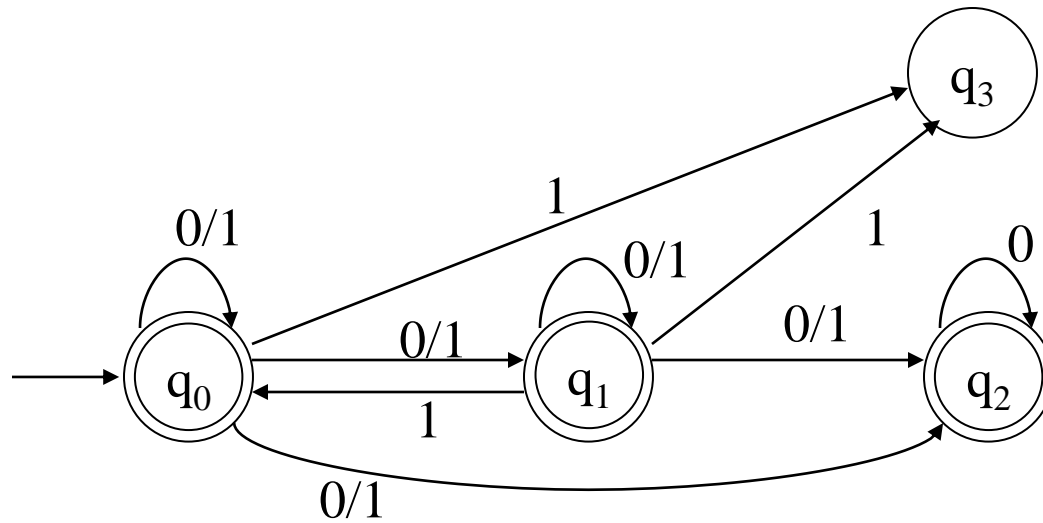
- Step #6:



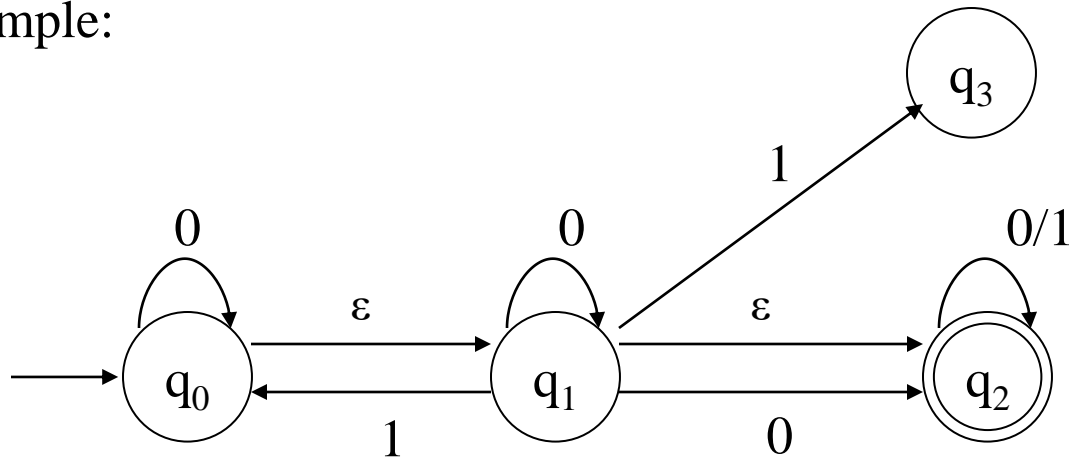
- Example:



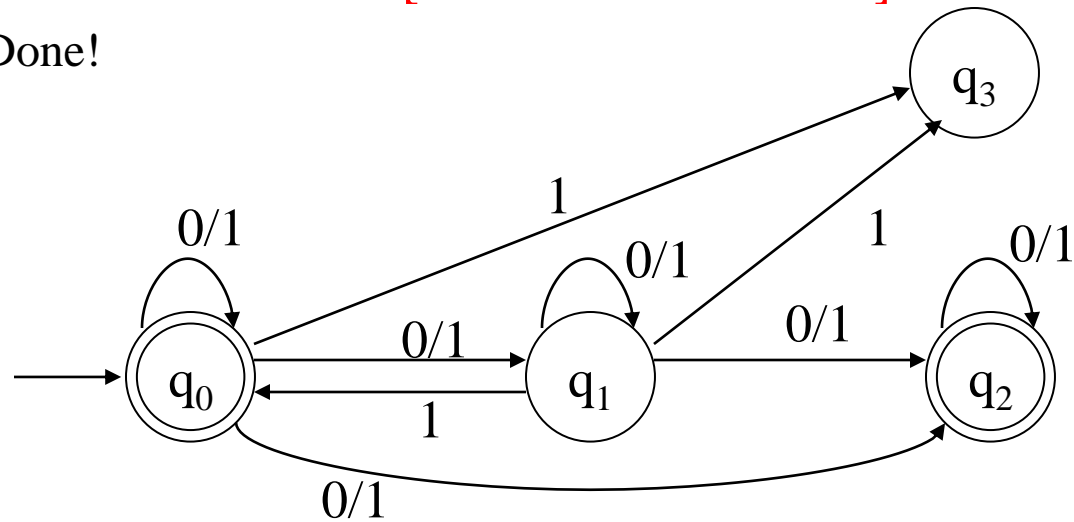
- Step #7:



- Example:



- 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.



Finite Automata

Reading: Chapter 2



Finite Automaton (FA)

- Informally, a state diagram that comprehensively captures all possible states and transitions that a machine can take while responding to a stream or sequence of input symbols
- Recognizer for “Regular Languages”
- Deterministic Finite Automata (DFA)
 - The machine can exist in only one state at any given time
- Non-deterministic Finite Automata (NFA)
 - The machine can exist in multiple states at the same time



Deterministic Finite Automata

- Definition

- A Deterministic Finite Automaton (DFA) consists of:
 - $Q \implies$ a finite set of states
 - $\Sigma \implies$ a finite set of input symbols (alphabet)
 - $q_0 \implies$ a start state
 - $F \implies$ set of accepting states
 - $\delta \implies$ a transition function, which is a mapping between $Q \times \Sigma \implies Q$
- A DFA is defined by the 5-tuple:
 - $\{Q, \Sigma, q_0, F, \delta\}$



What does a DFA do on reading an input string?

- Input: a word w in Σ^*
- Question: Is w acceptable by the DFA?
- Steps:
 - Start at the “start state” q_0
 - For every input symbol in the sequence w do
 - Compute the next state from the current state, given the current input symbol in w and the transition function
 - If after all symbols in w are consumed, the current state is one of the accepting states (F) then *accept* w ;
 - Otherwise, *reject* w .

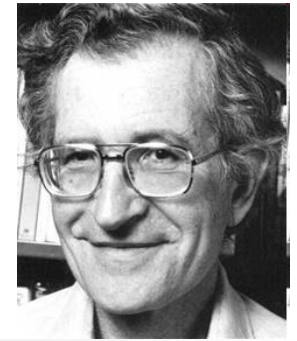


Regular Languages

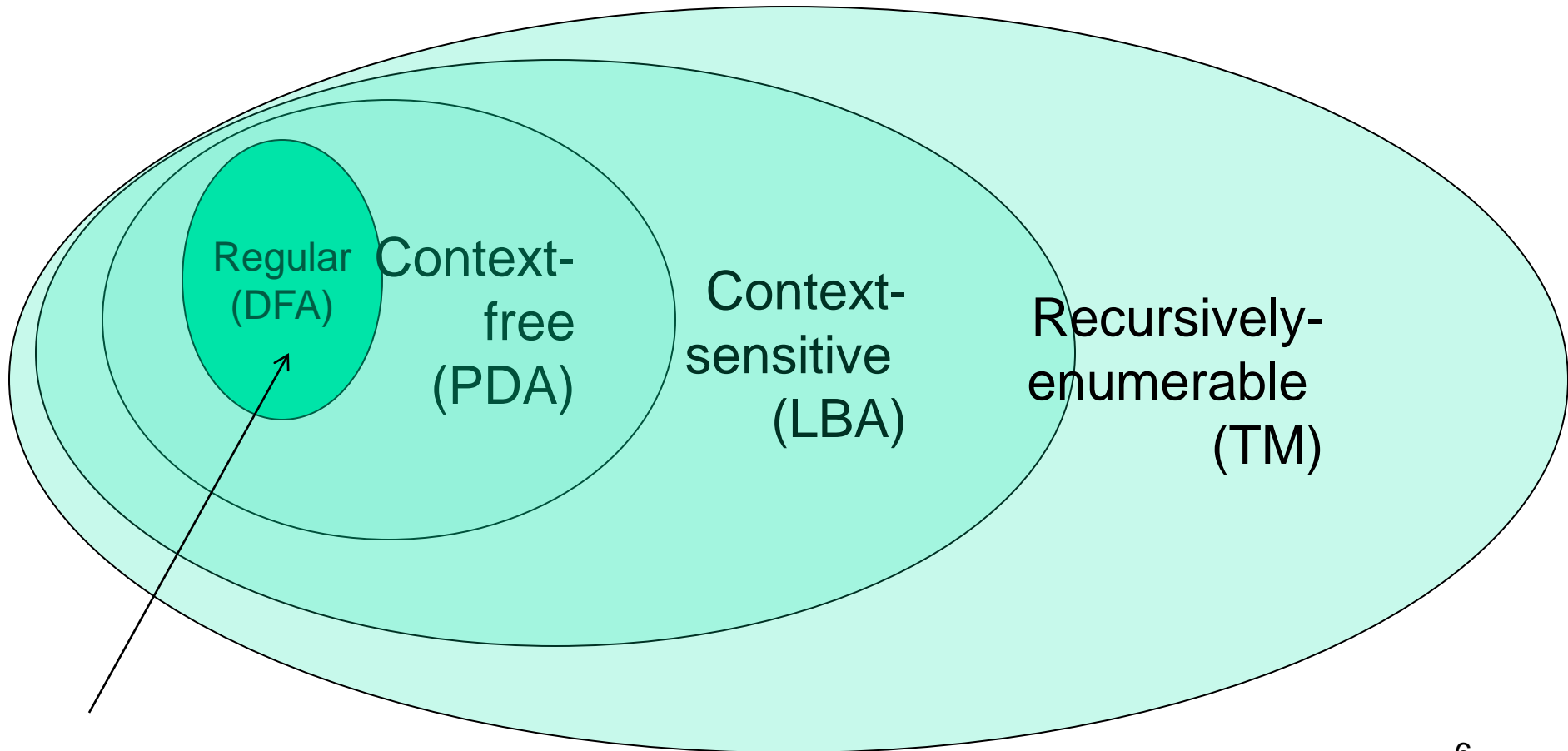
- Let $L(A)$ be a language *recognized* by a DFA A .
 - Then $L(A)$ is called a “*Regular Language*”.
- Locate regular languages in the Chomsky Hierarchy



The Chomsky Hierarchy



- A containment hierarchy of classes of formal languages





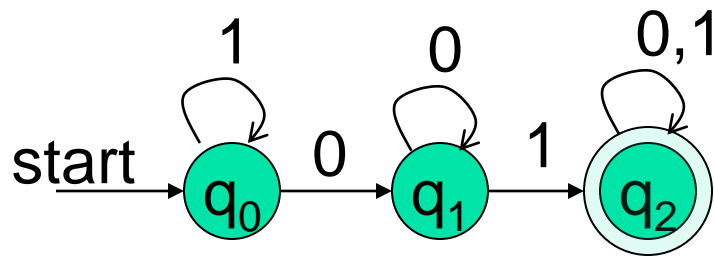
Example #1

- Build a DFA for the following language:
 - $L = \{w \mid w \text{ is a binary string that contains } 01 \text{ as a substring}\}$
- Steps for building a DFA to recognize L:
 - $\Sigma = \{0,1\}$
 - Decide on the states: Q
 - Designate start state and final state(s)
 - δ : Decide on the transitions:
- “Final” states == same as “accepting states”
- Other states == same as “non-accepting states”

Regular expression: $(0+1)^*01(0+1)^*$

DFA for strings containing 01

- What makes this DFA deterministic?



Accepting state

- What if the language allows empty strings?

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- start state = q_0
- $F = \{q_2\}$

Transition table

		symbols	
δ		0	1
states	q_0	q_1	q_0
	q_1	q_1	q_2
	$*q_2$	q_2	q_2



Example #2

Clamping Logic:

- A clamping circuit waits for a "1" input, and turns on forever. However, to avoid clamping on spurious noise, we'll design a DFA that waits for *two consecutive 1s* in a row before clamping on.
- Build a DFA for the following language:
$$L = \{ w \mid w \text{ is a bit string which contains the substring } 11 \}$$
- State Design:
 - q_0 : start state (initially off), also means the most recent input was not a 1
 - q_1 : has never seen 11 but the most recent input was a 1
 - q_2 : has seen 11 at least once



Example #3

- Build a DFA for the following language:
$$L = \{ w \mid w \text{ is a binary string that has even number of 1s and even number of 0s} \}$$
- ?



Extension of transitions (δ) to Paths ($\hat{\delta}$)

- $\hat{\delta}(q, w) = \text{destination state from state } q \text{ on input string } w$
- $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$
- Work out example #3 using the input sequence $w=10010$, $a=1$:
 - $\hat{\delta}(q_0, wa) = ?$



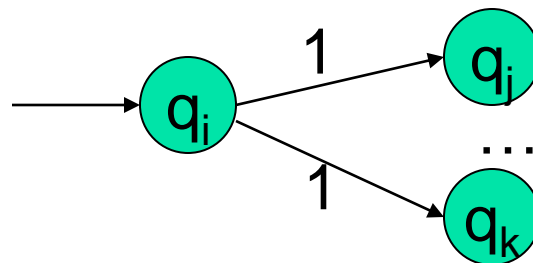
Language of a DFA

A DFA A accepts string w if there is a path from q_0 to an accepting (or final) state that is labeled by w

- *i.e., $L(A) = \{ w \mid \hat{\delta}(q_0, w) \in F \}$*
- *I.e., $L(A) = \text{all strings that lead to an accepting state from } q_0$*

Non-deterministic Finite Automata (NFA)

- A Non-deterministic Finite Automaton (NFA)
 - is of course “non-deterministic”
 - Implying that the machine can exist in more than one state at the same time
 - Transitions could be non-deterministic



- Each transition function therefore maps to a set of states



Non-deterministic Finite Automata (NFA)

- A Non-deterministic Finite Automaton (NFA) consists of:
 - $Q \implies$ a finite set of states
 - $\Sigma \implies$ a finite set of input symbols (alphabet)
 - $q_0 \implies$ a start state
 - $F \implies$ set of accepting states
 - $\delta \implies$ a transition function, which is a mapping between $Q \times \Sigma \implies$ subset of Q
- An NFA is also defined by the 5-tuple:
 - $\{Q, \Sigma, q_0, F, \delta\}$



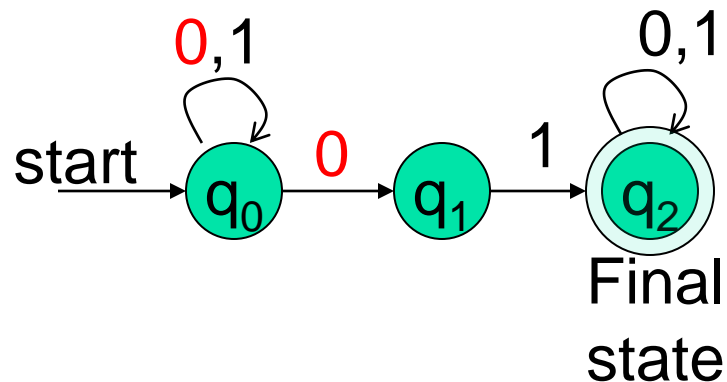
How to use an NFA?

- Input: a word w in Σ^*
- Question: Is w acceptable by the NFA?
- Steps:
 - Start at the “start state” q_0
 - For every input symbol in the sequence w do
 - Determine **all possible next states from all current states**, given the current input symbol in w and the transition function
 - If after all symbols in w are consumed and if at least **one of** the current states is a final state then *accept* w ;
 - Otherwise, *reject* w .

Regular expression: $(0+1)^*01(0+1)^*$

NFA for strings containing 01

Why is this non-deterministic?



What will happen if at state q_1 an input of 0 is received?

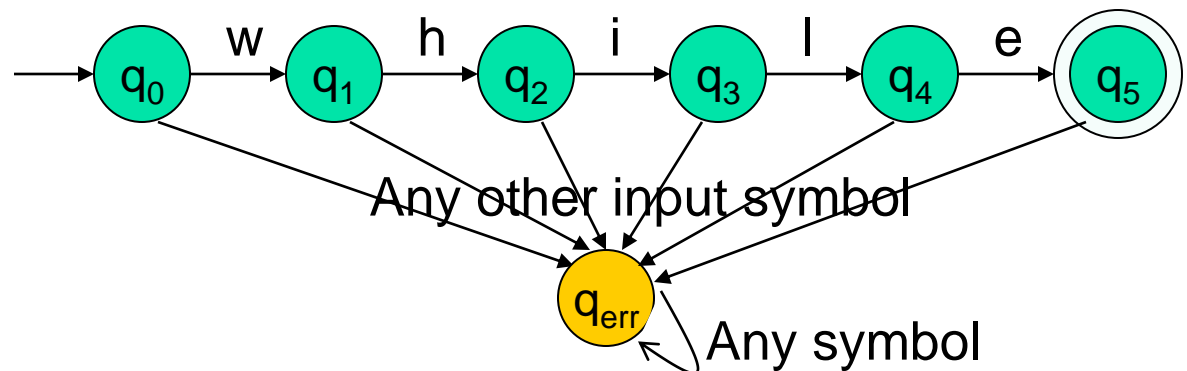
- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- start state = q_0
- $F = \{q_2\}$
- Transition table

symbols		
δ	0	1
states $\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	Φ	$\{q_2\}$
$*q_2$	$\{q_2\}$	$\{q_2\}$

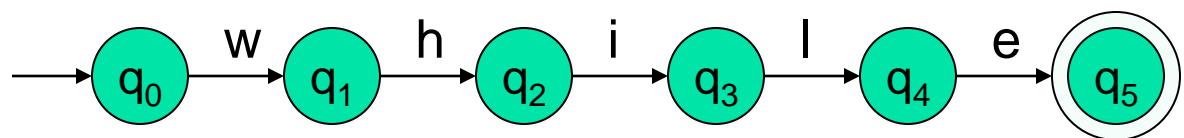
Note: Omitting to explicitly show error states is just a matter of design convenience (one that is generally followed for NFAs), and i.e., this feature should not be confused with the notion of non-determinism.

What is an “error state”?

- A DFA for recognizing the key word “*while*”



- An NFA for the same purpose:



Transitions into a dead state are implicit



Example #2

- Build an NFA for the following language:
 $L = \{ w \mid w \text{ ends in } 01 \}$
- ?
- Other examples
 - Keyword recognizer (e.g., if, then, else, while, for, include, etc.)
 - Strings where the first symbol is present somewhere later on at least once



Extension of $\hat{\delta}$ to NFA Paths

- Basis: $\hat{\delta}(q, \varepsilon) = \{q\}$
- Induction:
 - Let $\hat{\delta}(q_0, w) = \{p_1, p_2, \dots, p_k\}$
 - $\delta(p_i, a) = S_i$ for $i=1, 2, \dots, k$
 - Then, $\hat{\delta}(q_0, wa) = S_1 \cup S_2 \cup \dots \cup S_k$



Language of an NFA

- An NFA accepts w if *there exists at least one* path from the start state to an accepting (or final) state that is labeled by w
- $L(N) = \{ w \mid \hat{\delta}(q_0, w) \cap F \neq \Phi \}$



Advantages & Caveats for NFA

- Great for modeling regular expressions
 - String processing - e.g., grep, lexical analyzer
- Could a non-deterministic state machine be implemented in practice?
 - Probabilistic models could be viewed as extensions of non-deterministic state machines (e.g., toss of a coin, a roll of dice)
 - They are not the same though
 - A parallel computer could exist in multiple “states” at the same time

Technologies for NFAs

- Micron's Automata Processor (introduced in 2013)
- 2D array of MISD (multiple instruction single data) fabric w/ thousands to millions of processing elements.
- 1 input symbol = fed to all states (i.e., cores)
- Non-determinism using circuits
- <http://www.micronautomata.com/>



But, DFAs and NFAs are equivalent in their power to capture languages !!

Differences: DFA vs. NFA

■ DFA

1. All transitions are deterministic
 - Each transition leads to exactly one state
2. For each state, transition on all possible symbols (alphabet) should be defined
3. Accepts input if the last state visited is in F
4. Sometimes harder to construct because of the number of states
5. Practical implementation is feasible

■ NFA

1. Some transitions could be non-deterministic
 - A transition could lead to a subset of states
2. Not all symbol transitions need to be defined explicitly (if undefined will go to an error state – this is just a design convenience, not to be confused with “non-determinism”)
3. Accepts input if *one of* the last states is in F
4. Generally easier than a DFA to construct
5. Practical implementations limited but emerging (e.g., Micron automata processor)



Equivalence of DFA & NFA

- Theorem:

Should be
true for
any L

- → A language L is accepted by a DFA if and only if it is accepted by an NFA.

- Proof:

1. If part:

- Prove by showing every NFA can be converted to an equivalent DFA (in the next few slides...)

2. Only-if part is trivial:

- Every DFA is a special case of an NFA where each state has exactly one transition for every input symbol. Therefore, if L is accepted by a DFA, it is accepted by a corresponding NFA. □



Proof for the if-part

- If-part: A language L is accepted by a DFA if it is accepted by an NFA
 - rephrasing...
 - Given any NFA N , we can construct a DFA D such that $L(N)=L(D)$
-
- How to convert an NFA into a DFA?
 - Observation: In an NFA, each transition maps to a *subset* of states
 - Idea: Represent:
each “subset of NFA_states” \rightarrow a single “DFA_state”

Subset construction



NFA to DFA by subset construction

- Let $N = \{Q_N, \Sigma, \delta_N, q_0, F_N\}$
- Goal: Build $D = \{Q_D, \Sigma, \delta_D, \{q_0\}, F_D\}$ s.t. $L(D) = L(N)$
- Construction:
 1. Q_D = all subsets of Q_N (i.e., power set)
 2. F_D = set of subsets S of Q_N s.t. $S \cap F_N \neq \emptyset$
 3. δ_D : for each subset S of Q_N and for each input symbol a in Σ :
 - $\delta_D(S, a) = \bigcup \delta_N(p, a)$

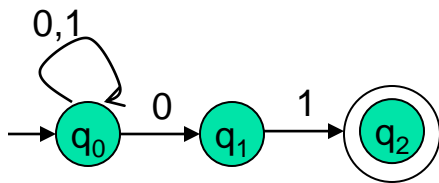
$p \in s$

Idea: To avoid enumerating all of power set, do "lazy creation of states"

NFA to DFA construction: Example

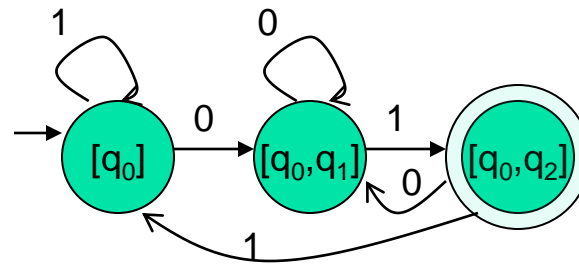
- $L = \{w \mid w \text{ ends in } 01\}$

NFA:



δ_N	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$
$*q_2$	\emptyset	\emptyset

DFA:



δ_D	
\emptyset	
$\rightarrow [q_0]$	
$[q_1]$	
$*[q_2]$	
$[q_0, q_1]$	
$*[q_0, q_2]$	
$*[q_1, q_2]$	
$*[q_0, q_1, q_2]$	

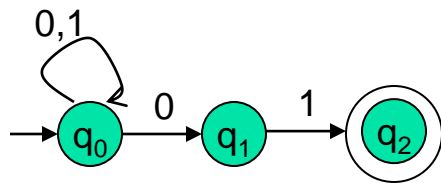
δ_D	0	1
$\rightarrow [q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_2]$
$*[q_0, q_2]$	$[q_0, q_1]$	$[q_0]$

- Enumerate all possible subsets
- Determine transitions
- Retain only those states reachable from $\{q_0\}$

NFA to DFA: Repeating the example using *LAZY CREATION*

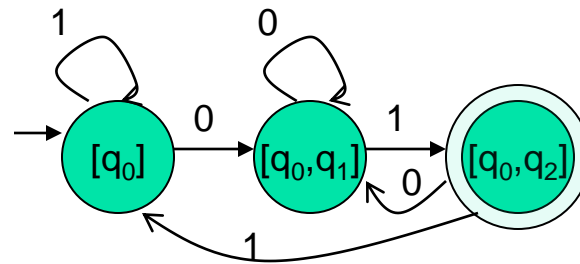
- $L = \{w \mid w \text{ ends in } 01\}$

NFA:



δ_N	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$
$*q_2$	\emptyset	\emptyset

DFA:



δ_D	0	1
$[q_0]$	$[q_0, q_1]$	$[q_0]$

Main Idea:

Introduce states as you go
(on a need basis)

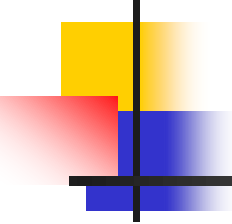


Correctness of subset construction

Theorem: *If D is the DFA constructed from NFA N by subset construction, then $L(D)=L(N)$*

■ Proof:

- Show that $\hat{\delta}_D(\{q_0\}, w) \equiv \hat{\delta}_N(q_0, w)$, for all w
- Using induction on w 's length:
 - Let $w = xa$
 - $\hat{\delta}_D(\{q_0\}, xa) \equiv \hat{\delta}_D(\hat{\delta}_N(q_0, x), a) \equiv \hat{\delta}_N(q_0, w)$



A bad case where $\#states(DFA) \gg \#states(NFA)$

- $L = \{w \mid w \text{ is a binary string s.t., the } k^{\text{th}} \text{ symbol from its end is a } 1\}$
 - NFA has $k+1$ states
 - But an equivalent DFA needs to have at least 2^k states

(Pigeon hole principle)

- m holes and $>m$ pigeons
 - \Rightarrow at least one hole has to contain two or more pigeons



Applications

- Text indexing
 - inverted indexing
 - For each unique word in the database, store all locations that contain it using an NFA or a DFA
- Find pattern P in text T
 - Example: Google querying
- Extensions of this idea:
 - PATRICIA tree, suffix tree



A few subtle properties of DFAs and NFAs

- The machine never really terminates.
 - It is always waiting for the next input symbol or making transitions.
- The machine decides when to consume the next symbol from the input and when to ignore it.
 - (but the machine can never skip a symbol)
- \Rightarrow A transition can happen even *without* really consuming an input symbol (think of consuming ε as a free token) – if this happens, then it becomes an ε -NFA (see next few slides).
- A single transition *cannot* consume more than one (non- ε) symbol.



FA with ε -Transitions

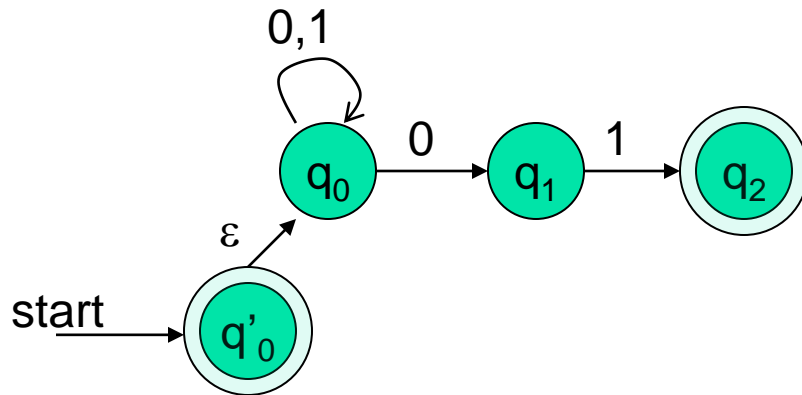
- We can allow explicit ε -transitions in finite automata
 - i.e., a transition from one state to another state without consuming any additional input symbol
 - Explicit ε -transitions between different states introduce non-determinism.
 - Makes it easier sometimes to construct NFAs

Definition: ε -NFAs are those NFAs with at least one explicit ε -transition defined.

- ε -NFAs have one more column in their transition table

Example of an ε -NFA

$L = \{w \mid w \text{ is empty, or if non-empty will end in } 01\}$



- ε -closure of a state q , **ECLOSE(q)**, is the set of all states (including itself) that can be *reached* from q by repeatedly making an arbitrary number of ε -transitions.

δ_E	0	1	ε	
$\rightarrow *q'_0$	\emptyset	\emptyset	$\{q'_0, q_0\}$	ECLOSE(q'_0)
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$	ECLOSE(q_0)
q_1	\emptyset	$\{q_2\}$	$\{q_1\}$	ECLOSE(q_1)
$*q_2$	\emptyset	\emptyset	$\{q_2\}$	ECLOSE(q_2)

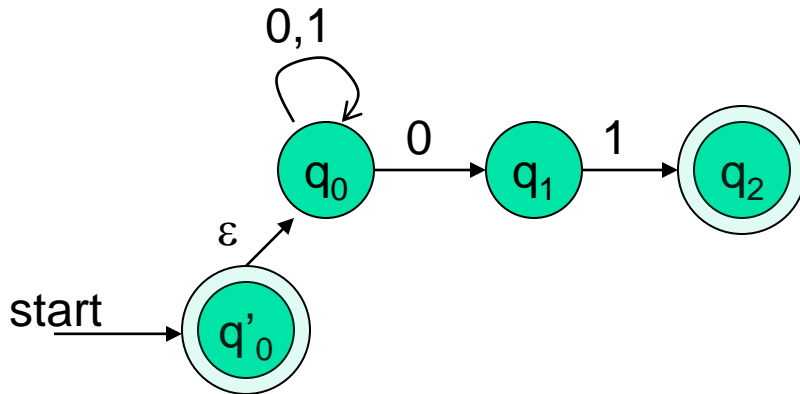
To simulate any transition:

Step 1) Go to all immediate destination states.

Step 2) From there go to all their ε -closure states as well.

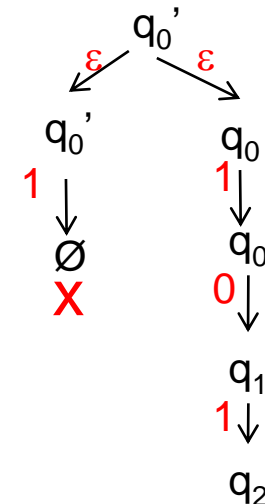
Example of an ε -NFA

$L = \{w \mid w \text{ is empty, or if non-empty will end in } 01\}$



Simulate for $w=101$:

δ_E	0	1	ε
$*q'_0$	\emptyset	\emptyset	$\{q'_0, q_0\}$ ← ECLOSE(q'_0)
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$ ← ECLOSE(q_0)
q_1	\emptyset	$\{q_2\}$	$\{q_1\}$
$*q_2$	\emptyset	\emptyset	$\{q_2\}$

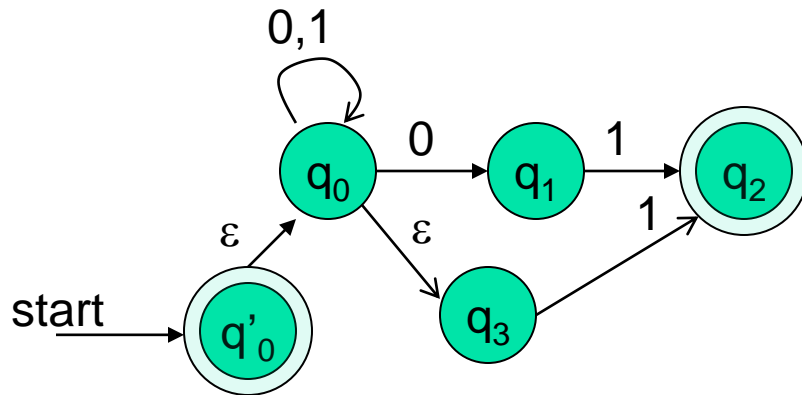


To simulate any transition:

Step 1) Go to all immediate destination states.

Step 2) From there go to all their ε -closure states as well.

Example of another ε -NFA



Simulate for $w=101$:

?

δ_E	0	1	ε
$\rightarrow *q'_0$	\emptyset	\emptyset	$\{q'_0, q_0, q_3\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0, q_3\}$
q_1	\emptyset	$\{q_2\}$	$\{q_1\}$
$*q_2$	\emptyset	\emptyset	$\{q_2\}$
q_3	\emptyset	$\{q_2\}$	$\{q_3\}$



Equivalency of DFA, NFA, ϵ -NFA

- Theorem: A language L is accepted by some ϵ -NFA if and only if L is accepted by some DFA
- Implication:
 - $\text{DFA} \equiv \text{NFA} \equiv \epsilon\text{-NFA}$
 - (all accept Regular Languages)



Eliminating ε -transitions

Let $E = \{Q_E, \Sigma, \delta_E, q_0, F_E\}$ be an ε -NFA

Goal: To build DFA $D = \{Q_D, \Sigma, \delta_D, \{q_D\}, F_D\}$ s.t. $L(D) = L(E)$

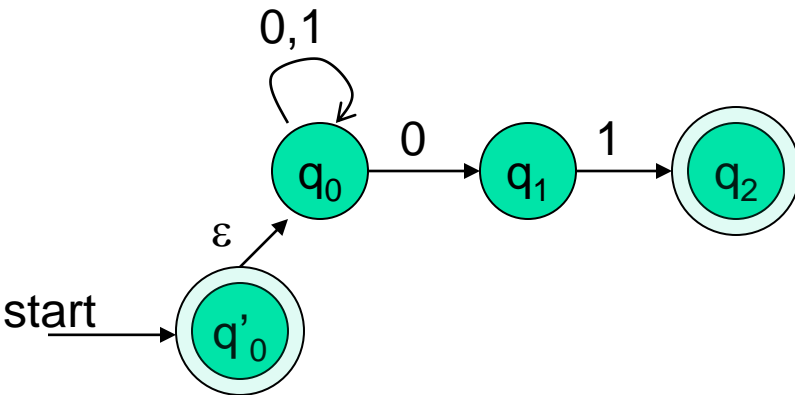
Construction:

1. Q_D = all reachable subsets of Q_E factoring in ε -closures
2. $q_D = \text{ECLOSE}(q_0)$
3. F_D = subsets S in Q_D s.t. $S \cap F_E \neq \emptyset$
4. δ_D : for each subset S of Q_E and for each input symbol $a \in \Sigma$:
 - Let $R = \bigcup \delta_E(p, a)$ // go to destination states
 - $\delta_D(S, a) = \bigcup_{r \in R}^{\text{p in s}} \text{ECLOSE}(r)$ // from there, take a union of all their ε -closures

Reading: Section 2.5.5 in book

Example: ε -NFA \rightarrow DFA

$L = \{w \mid w \text{ is empty, or if non-empty will end in } 01\}$

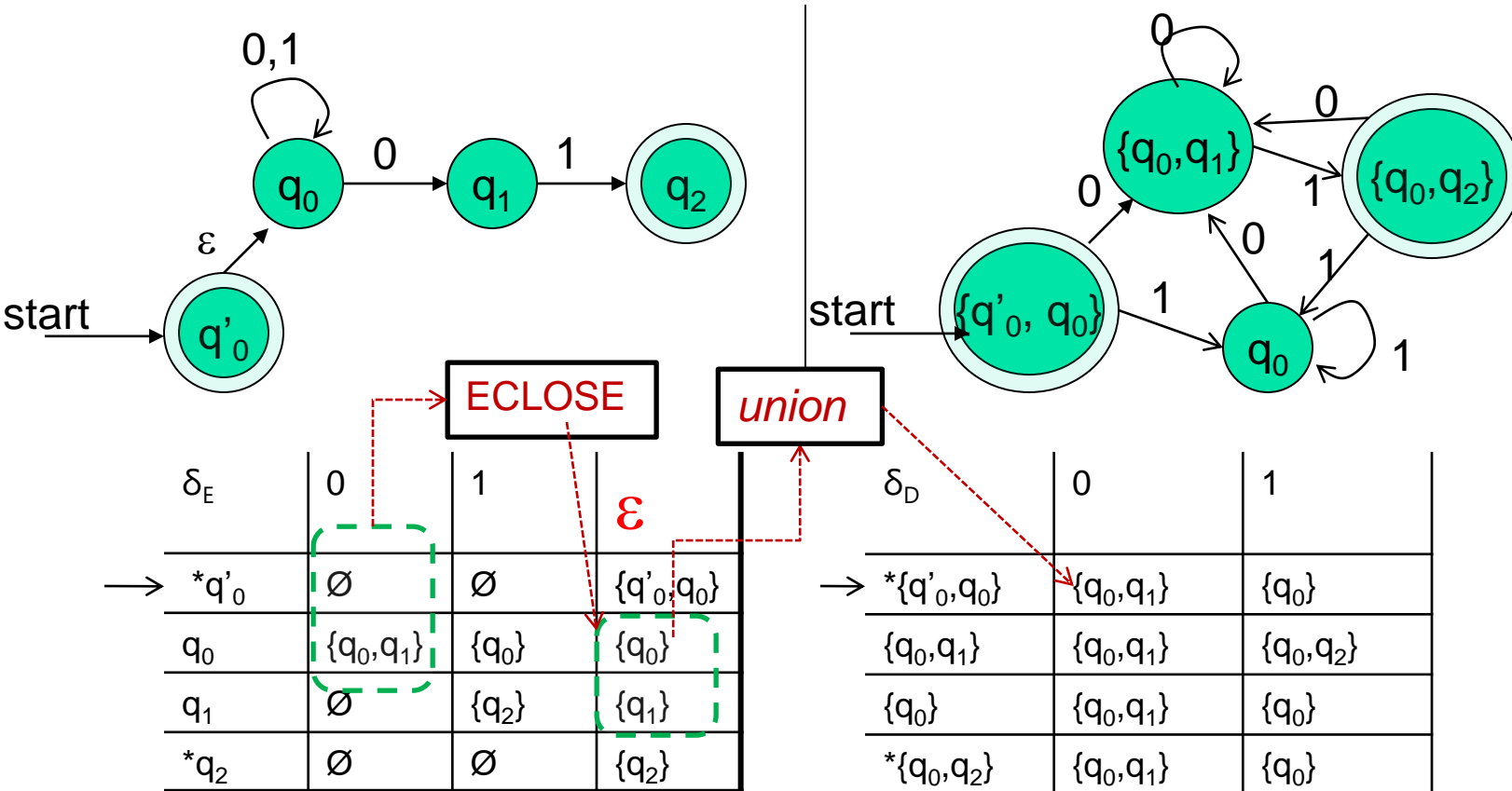


δ_E	0	1	ε
$\rightarrow *q'_0$	\emptyset	\emptyset	$\{q'_0, q_0\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$	$\{q_1\}$
$*q_2$	\emptyset	\emptyset	$\{q_2\}$

δ_D	0	1
$\rightarrow * \{q'_0, q_0\}$		
...		

Example: ε -NFA \rightarrow DFA

$L = \{w \mid w \text{ is empty, or if non-empty will end in } 01\}$





Summary

- DFA
 - Definition
 - Transition diagrams & tables
- Regular language
- NFA
 - Definition
 - Transition diagrams & tables
- DFA vs. NFA
- NFA to DFA conversion using subset construction
- Equivalency of DFA & NFA
- Removal of redundant states and including dead states
- ϵ -transitions in NFA
- Pigeon hole principles
- Text searching applications

REGULAR EXPRESSION AND LANGUAGES

-
- FA can also be defined as m/c dividing class of i/p strings into two groups ; those which are acceptable and those that are not.
 - A language is a **regular set(or regular)** if it is a set accepted by some FA.

EG: following are the regular set S :

1. $S1 = \{\epsilon, 1, 11, 111, 1111, \dots\}$
2. $S2 = \{\epsilon, 0, 1, 00, 11, 01, 10, 000, 111, \dots\}$
3. $S3 = \{1, 01, 11, 0000001, 1111111, 10101, 0101, \dots\}$

So on....

Regular Expressions

- In maths one can use operations $+$ and \times to build up expressions like: $(12 + 5) \times 2$ ($= 34$)
- Similarly, can build regular operations to build expressions describing languages.
- An example is: $(0 \cup 1)1^*$ = a language!!
- In this case it is the language consisting of all strings starting with a 0 or 1 followed by zero or more 1's (1^*).

Regular Expression : Definition

- Regular languages are defined by regular Expression.
- A **regular expression**, or **RE**, describes strings of characters. It's a pattern that matches certain strings and doesn't match others. **A regular expression is a set of characters that specify a pattern.**
- A regular expression is a “user-friendly” declarative way of describing a language.
- Example: 01^*+10^*

Inductive Definition of RE

- Given a particular alphabet Σ and using three basic operations Union, Concatenation, and Kleene closure we can build RE representing Regular language.
- R is a regular expression if R is
 1. ε , the empty string
 2. a , for some $a \in \Sigma$
 3. $R_1 + R_2$, where R_1 and R_2 are reg. exp.
 4. $R_1 \bullet R_2$, where R_1 and R_2 are reg. exp.
 5. R_1^* , where R_1 is a regular expression

Here " $+$ " is Union operation

" \bullet " is Concatenation operation

" $*$ " is Kleene Closure operation

Language of a RE

- A regular expression R describes the language $L(R)$
 - $L(\emptyset) = \emptyset$
 - $L(\varepsilon) = \{\varepsilon\}$
 - $L(a) = \{a\}$
 - $L(R_1 + R_2) = L(R_1) \cup L(R_2)$ ← union
 - $L(R_1 R_2) = L(R_1) L(R_2)$ ← concatenation
 - $L(R_1^*) = (L(R_1))^*$ ← closure

Concatenation of Languages

- If L_1 and L_2 are languages, we can define the concatenation

$$L_1 L_2 = \{w \mid w=xy, x \in L_1, y \in L_2\}$$

- Examples:
 - $\{ab, ba\}\{cd, dc\} =? \{abcd, abdc, bacd, badc\}$
 - $\emptyset\{ab\} =? \emptyset$

Precedence of Operators used in regular expressions

- Closure has the highest precedence, followed by concatenation, followed by union.

Regular Expressions

- Which of these are value languages for the regular expression: $(00)^*1(0 \cup 1)^*?$
- 0, 1, 010, 0010, 000011, 0011010, 0000101010
 - 0 No - must be at least one 1
 - 1 Yes
 - 010 No – must be two 0's
 - 0010 Yes
 - 000011 Yes
 - 0011010 Yes
 - 0000101010 Yes

Regular Expressions

- The expression $(0 \cup 1)^*$ is the language consisting of all possible strings of 0's and 1's.
- If the alphabet $\Sigma = \{0,1\}$, we can write Σ as shorthand for $(0 \cup 1)$.
- If Σ is any alphabet, the regular expression Σ describes the language of all strings of length 1.
- Σ^* is the language of all strings over the alphabet.
- Σ^*0 is the language that contains all strings ending in a 0.
- The language $(0\Sigma^*) \cup (\Sigma^*1)$ consists of all strings that either start with a 0 or end with a 1.

Some RE

1. Strings of a's with zero or more length.

ie $S = \{\epsilon, a, aa, aaa, aaaa, \dots\}$

This is written in terms of RE as : a^*

2. Strings of length of one. $S = \{a, b\}$

RE = $a+b$

3. Strings of even length. $S = \{aa, ab, ba, bb\}$

RE = $(a+b)(a+b)$

4. Strings made up of any combination of a's and b's of any length (zero length also)

ie $S = \{\epsilon, a, b, ab, ba, aa, bb, aab, baa, aba, bab, aaa, bbb, \dots\}$

This is written in terms of RE as : $(a+b)^*$

Examples

1. The set of strings over $\{0,1\}$ that end in 3 consecutive 1's.

ans: $(0 \mid 1)^* 111$
OR $(0 + 1)^* 111$

2. The set of strings over $\{0,1\}$ that have at least one 1.

ans: $0^* 1 (0 + 1)^*$

3. The set of strings over $\{0,1\}$ that have at most one 1.

ans: $0^* \mid 0^* 1 0^*$

4. The set of strings over $\{A..Z,a..z\}$ that contain the word "main".

Let $\langle \text{letter} \rangle = A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$

ans: $\langle \text{letter} \rangle^* \text{main} \langle \text{letter} \rangle^*$

5. The set of strings over $\{A..Z,a..z\}$ that contain 3 x's.

ans: $\langle \text{letter} \rangle^* x \langle \text{letter} \rangle^* x \langle \text{letter} \rangle^* x \langle \text{letter} \rangle^*$

6. All strings of 0's and 1's

Ans: $(0|1)^*$

7. All strings of 0's and 1's with at least 2 consecutive 0's

Ans: $(0|1)^*00(0|1)^*$

8. All strings of 0's and 1's beginning with 1 and not having two consecutive 0's

Ans: $(1+10)^*$

Identities of regular Expression

- Two RE P and Q are equivalent (ie $P=Q$) if and only if P represents the same set of strings as Q does.
- For this we need to show some identities of RE.

Let P,Q,R are RE then identity rules as :

1. $\epsilon R = R \epsilon = R$
2. $\epsilon^* = \epsilon$
3. $(\Phi)^* = \epsilon$
4. $\Phi R = R \Phi = \Phi$
5. $\Phi + R = R$
6. $R + R = R$
7. $RR^* = R^*R = R +$
8. $(R^*)^* = R^*$
9. $\epsilon + RR^* = R^*$

-
10. $(P+Q)R=PR+QR$
 11. $(P+Q)^*=(P^*Q^*)=(P^*+Q^*)^*$
 12. $R^*(\epsilon +R)= (\epsilon +R)R^*=R^*$
 13. $(R+ \epsilon)^*=R^*$
 14. $\epsilon +R^*=R^*$
 15. $(PQ)^*P=P(QP)^*$
 16. $R^*R+R=R^*R$

Ardens Theorem : let P, Q be two RE over i/p set Σ . If P does not contain ϵ then there exists RE R such that **$R = Q+RP$** which has a unique solution as **$R = QP^*$**

Equivalence of two RE

- Two RE P and Q are equivalent (ie $P=Q$) if and only if P represents the same set of strings as Q does.

$$R1 = R2 \quad \text{iff} \quad L(R1) = L(R2)$$

Properties of Regular Languages (Closure Properties)

- If certain language is regular and language L is formed from them by certain operations then L is also regular
- Certain operations on regular languages are guaranteed to produce regular languages
 - Union: $L \cup M$
 - Intersection: $L \cap M$
 - Complement: \bar{L}
 - Difference: $L - M$
 - Reversal: $L^R = \{w^R \mid w \in L\}$
 - Closure: L^*
 - Concatenation: LM
 - Homomorphism:
 $h(L) = \{h(w) \mid w \in L, h \text{ is a homomorphism}\}$
 - Inverse homomorphism:
 $h^{-1}(L) = \{w \mid h(w) \in L, h \text{ is a homomorphism}\}$

Limits on the Power of FA

1. An FA is limited on finite memory, i.e. finite set of states
2. Finite automata can't count.

Consider the language

$$a^n b^n = \{\varepsilon, ab, aabb, aaabbbb, \dots\}$$

i.e., a bunch of *a*'s followed by an equal number of *b*'s

No finite automaton accepts this language.

Can you prove this?

Pumping Lemma

- If L is a regular language, then there **exists** a constant n (pumping length) such that **every** string w in L with $|w| \geq n$, can be written as $w = xyz$, where:
 1. $|y| > 0$
 2. $|xy| \leq n$
 3. For **all** $i \geq 0$, xy^iz is also in L

Example 1

Question : $L = \{0^n 1^n \mid n \geq 0\}$

Solution:

Proof by method of contradiction .

Assume L is regular, let p be the pumping length

Choose $w = 0^p 1^p$ in L ($|w| > p$)

Applying PL, $w = xyz$, where $|y| > 0$, such that $xy^i z$ in L for all $i \geq 0$

■ Three possible cases **for y** :

1. Case 1: only 0's

eg : let $w = 000111$

can be broken as $w = 0 \ 00 \ 111$

2. Case 2: y contains only 1's

eg : $w = 000\ 11\ 1$

3. Case 3: y contains 0's and 1's

eg : $w = 00\ 01\ 11$

We need to find 'i' such that $w = xy^iz \in R$

Let $i=2$

Therefore case 1 : $w = 00 (0)^2 111$

$= 00\ 00\ 111$ not in L (as more no. of 0's than 1)

case 2 : $w = 000 (1)^2 11$

$= 000\ 1111$ not in L (as more no. of 1's than 0)

Case3: $w = 00 (01)^2 11$
 $= 00 0101 11$ not in L (as alternate 0's and 1's)

Thus L is not Regular



Regular Expressions

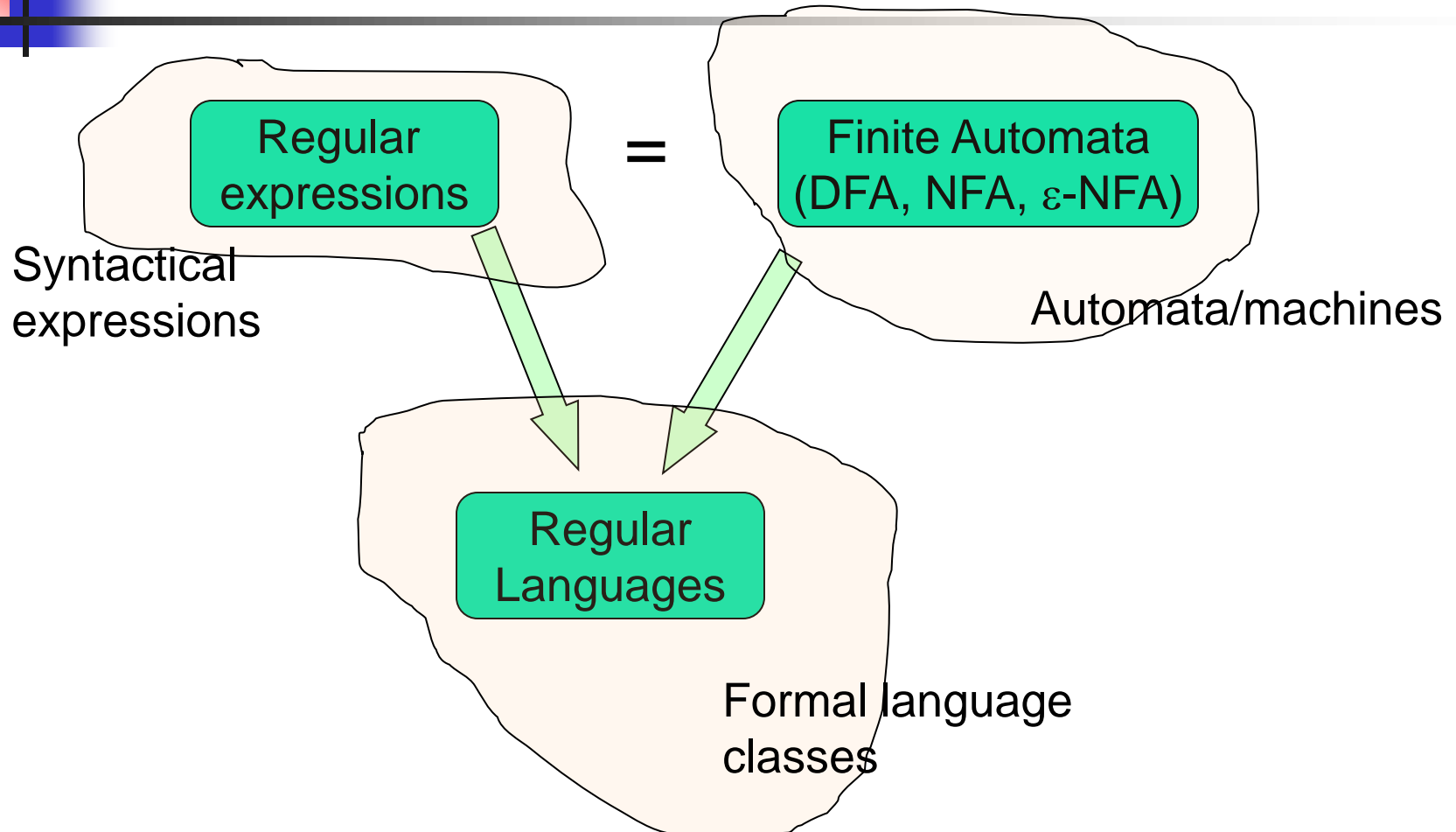
Reading: Chapter 3



Regular Expressions vs. Finite Automata

- Offers a declarative way to express the pattern of any string we want to accept
 - E.g., $01^* + 10^*$
- Automata \Rightarrow more machine-like
 - < input: string , output: [accept/reject] >
- Regular expressions \Rightarrow more program syntax-like
- Unix environments heavily use regular expressions
 - E.g., bash shell, grep, vi & other editors, sed
- Perl scripting – good for string processing
- Lexical analyzers such as Lex or Flex

Regular Expressions





Language Operators

- Union of two languages:
 - $L \cup M$ = all strings that are either in L or M
 - Note: A union of two languages produces a third language
- Concatenation of two languages:
 - $L . M$ = all strings that are of the form xy
s.t., $x \in L$ and $y \in M$
 - The *dot* operator is usually omitted
 - i.e., LM is same as $L.M$

“i” here refers to how many strings to concatenate from the parent language L to produce strings in the language L^i

Kleene Closure (the * operator)

- Kleene Closure of a given language L:
 - $L^0 = \{\epsilon\}$
 - $L^1 = \{w \mid \text{for some } w \in L\}$
 - $L^2 = \{w_1w_2 \mid w_1 \in L, w_2 \in L \text{ (duplicates allowed)}\}$
 - $L^i = \{w_1w_2\dots w_i \mid \text{all } w\text{'s chosen are } \in L \text{ (duplicates allowed)}\}$
 - (Note: the choice of each w_i is independent)
 - $L^* = \bigcup_{i \geq 0} L^i$ (arbitrary number of concatenations)

Example:

- Let $L = \{1, 00\}$
 - $L^0 = \{\epsilon\}$
 - $L^1 = \{1, 00\}$
 - $L^2 = \{11, 100, 001, 0000\}$
 - $L^3 = \{111, 1100, 1001, 10000, 000000, 00001, 00100, 0011\}$
 - $L^* = L^0 \cup L^1 \cup L^2 \cup \dots$



Kleene Closure (special notes)

- L^* is an infinite set iff $|L| \geq 1$ and $L \neq \{\varepsilon\}$ **Why?**
- If $L = \{\varepsilon\}$, then $L^* = \{\varepsilon\}$ **Why?**
- If $L = \Phi$, then $L^* = \{\varepsilon\}$ **Why?**

Σ^* denotes the set of all words over an alphabet Σ

- Therefore, an abbreviated way of saying there is an arbitrary language L over an alphabet Σ is:
 - $L \subseteq \Sigma^*$



Building Regular Expressions

- Let E be a regular expression and the language represented by E is $L(E)$
- Then:
 - $(E) = E$
 - $L(E + F) = L(E) \cup L(F)$
 - $L(E F) = L(E) L(F)$
 - $L(E^*) = (L(E))^*$

Example: how to use these regular expression properties and language operators?

- $L = \{ w \mid w \text{ is a binary string which does not contain two consecutive 0s or two consecutive 1s anywhere} \}$
 - E.g., $w = 01010101$ is in L , while $w = 10010$ is not in L
- Goal: Build a regular expression for L
- Four cases for w :
 - Case A: w starts with 0 and $|w|$ is even
 - Case B: w starts with 1 and $|w|$ is even
 - Case C: w starts with 0 and $|w|$ is odd
 - Case D: w starts with 1 and $|w|$ is odd
- Regular expression for the four cases:
 - Case A: $(01)^*$
 - Case B: $(10)^*$
 - Case C: $0(10)^*$
 - Case D: $1(01)^*$
- Since L is the union of all 4 cases:
 - Reg Exp for $L = (01)^* + (10)^* + 0(10)^* + 1(01)^*$
- If we introduce ε then the regular expression can be simplified to:
 - Reg Exp for $L = (\varepsilon + 1)(01)^*(\varepsilon + 0)$



Precedence of Operators

- Highest to lowest

- * operator (star)
- . (concatenation)
- + operator

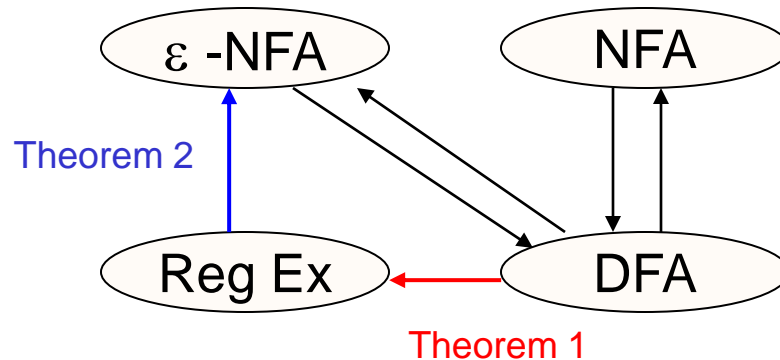
- Example:

- $01^* + 1 = (0 \cdot ((1)^*)) + 1$

Finite Automata (FA) & Regular Expressions (Reg Ex)

- To show that they are interchangeable, consider the following theorems:
 - Theorem 1: For every DFA A there exists a regular expression R such that $L(R)=L(A)$
 - Theorem 2: For every regular expression R there exists an ε -NFA E such that $L(E)=L(R)$

Proofs
in the book



Kleene Theorem

DFA

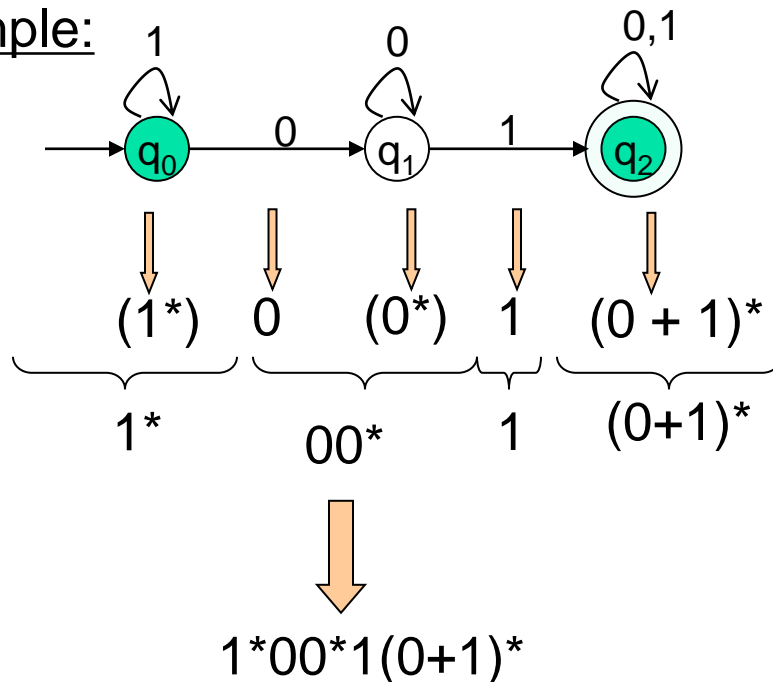
Theorem 1

Reg Ex

DFA to RE construction

Informally, trace all distinct paths (traversing cycles only once) from the start state to *each of the* final states and enumerate all the expressions along the way

Example:



Q) What is the language?

Reg Ex

Theorem 2

ϵ -NFA

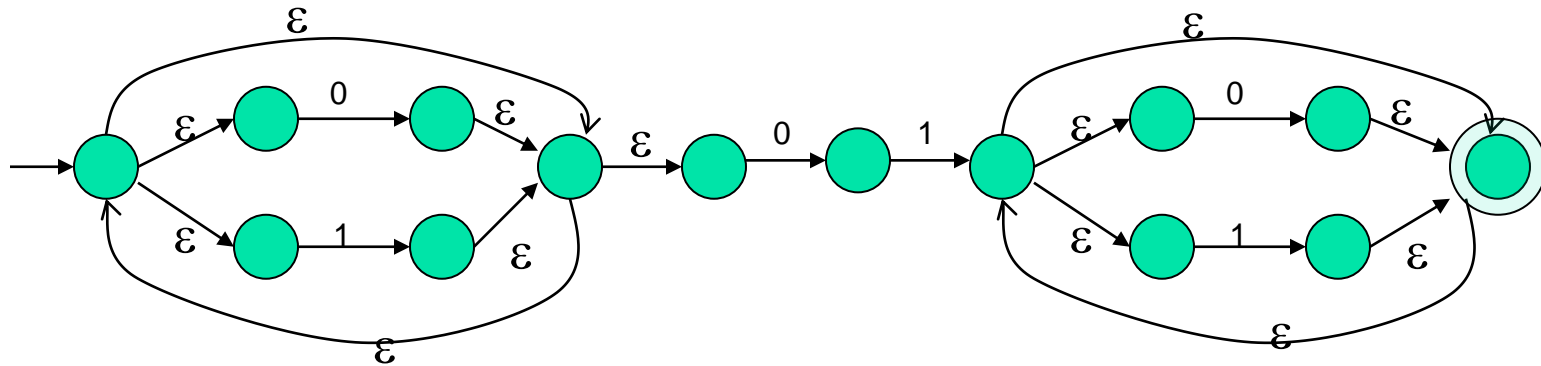
RE to ϵ -NFA construction

Example: $(0+1)^*01(0+1)^*$

$(0+1)^*$

01

$(0+1)^*$





Algebraic Laws of Regular Expressions

- Commutative:
 - $E + F = F + E$
- Associative:
 - $(E + F) + G = E + (F + G)$
 - $(EF)G = E(FG)$
- Identity:
 - $E + \Phi = E$
 - $\varepsilon E = E \varepsilon = E$
- Annihilator:
 - $\Phi E = E\Phi = \Phi$



Algebraic Laws...

- Distributive:
 - $E(F+G) = EF + EG$
 - $(F+G)E = FE+GE$
- Idempotent: $E + E = E$
- Involving Kleene closures:
 - $(E^*)^* = E^*$
 - $\Phi^* = \varepsilon$
 - $\varepsilon^* = \varepsilon$
 - $E^+ = EE^*$
 - $E? = \varepsilon + E$



True or False?

Let R and S be two regular expressions. Then:

1. $((R^*)^*)^* = R^*$?

2. $(R+S)^* = R^* + S^*$?

3. $(RS + R)^* RS = (RR^*S)^*$?



Summary

- Regular expressions
- Equivalence to finite automata
- DFA to regular expression conversion
- Regular expression to ε -NFA conversion
- Algebraic laws of regular expressions
- Unix regular expressions and Lexical Analyzer

Context-Free Languages

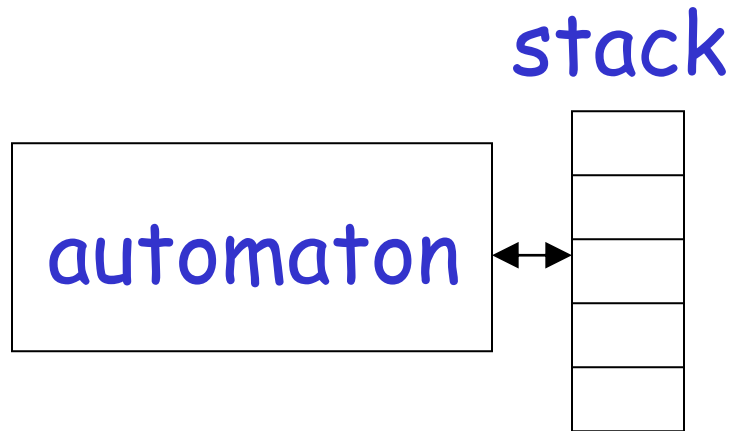
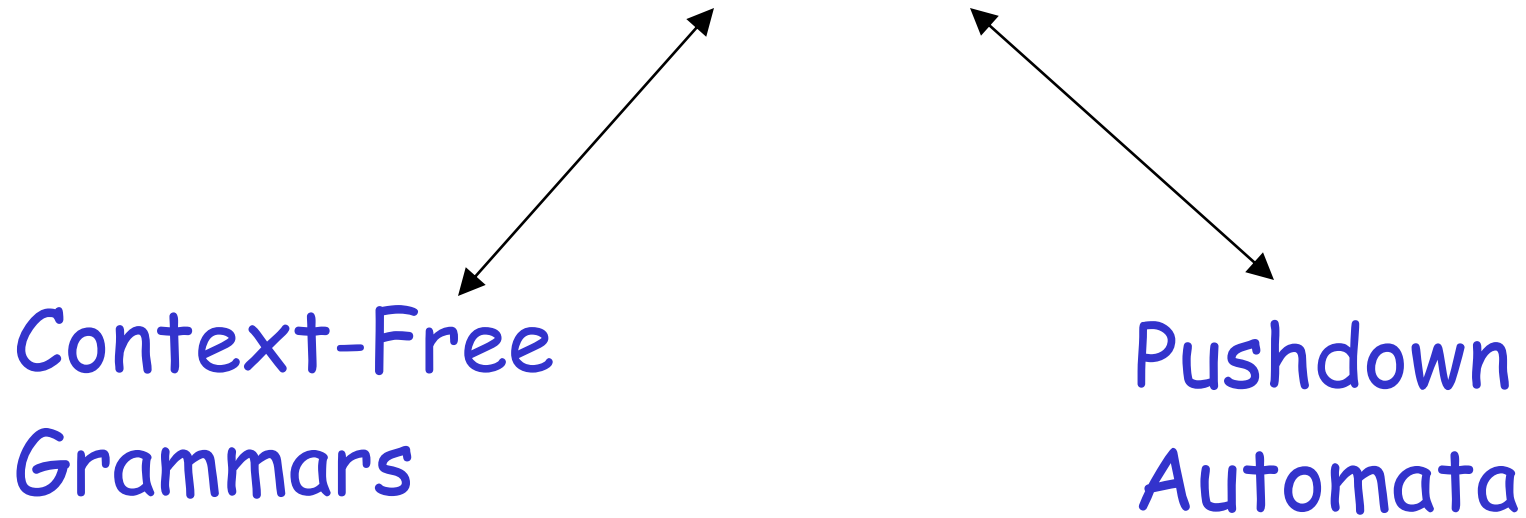
Context-Free Languages

$$\{a^n b^n : n \geq 0\} \quad \{ww^R\}$$

Regular Languages

$$a^* b^* \quad (a + b)^*$$

Context-Free Languages



Context-Free Grammars

Grammars

Grammars express languages

Example: the English language grammar

$$\langle sentence \rangle \rightarrow \langle noun_phrase \rangle \langle predicate \rangle$$
$$\langle noun_phrase \rangle \rightarrow \langle article \rangle \langle noun \rangle$$
$$\langle predicate \rangle \rightarrow \langle verb \rangle$$

$\langle \textit{article} \rangle \rightarrow a$

$\langle \textit{article} \rangle \rightarrow \textit{the}$

$\langle \textit{noun} \rangle \rightarrow \textit{cat}$

$\langle \textit{noun} \rangle \rightarrow \textit{dog}$

$\langle \textit{verb} \rangle \rightarrow \textit{runs}$

$\langle \textit{verb} \rangle \rightarrow \textit{sleeps}$

Derivation of string "the dog walks":

$\langle sentence \rangle \Rightarrow \langle noun_phrase \rangle \langle predicate \rangle$
 $\Rightarrow \langle noun_phrase \rangle \langle verb \rangle$
 $\Rightarrow \langle article \rangle \langle noun \rangle \langle verb \rangle$
 $\Rightarrow the \langle noun \rangle \langle verb \rangle$
 $\Rightarrow the \ dog \langle verb \rangle$
 $\Rightarrow the \ dog \ sleeps$

Derivation of string "a cat runs":

$\langle sentence \rangle \Rightarrow \langle noun_phrase \rangle \langle predicate \rangle$
 $\Rightarrow \langle noun_phrase \rangle \langle verb \rangle$
 $\Rightarrow \langle article \rangle \langle noun \rangle \langle verb \rangle$
 $\Rightarrow a \langle noun \rangle \langle verb \rangle$
 $\Rightarrow a \ cat \langle verb \rangle$
 $\Rightarrow a \ cat \ runs$

Language of the grammar:

$L = \{ \text{"a cat runs"},$
 $\text{"a cat sleeps"},$
 $\text{"the cat runs"},$
 $\text{"the cat sleeps"},$
 $\text{"a dog runs"},$
 $\text{"a dog sleeps"},$
 $\text{"the dog runs"},$
 $\text{"the dog sleeps"} \}$

Productions

Sequence of
Terminals (symbols)

$\langle \textit{noun} \rangle \rightarrow \textit{cat}$

$\langle \textit{sentence} \rangle \rightarrow \langle \textit{noun_phrase} \rangle \langle \textit{predicate} \rangle$

Variables

Sequence of Variables

Formal Definitions

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

Set of
variables



Set of
terminal
symbols

Start
variable

Set of
productions

Another Example

Sequence of
terminals and variables

Grammar:

$$S \rightarrow \overbrace{aSb}$$

$$S \rightarrow \lambda$$

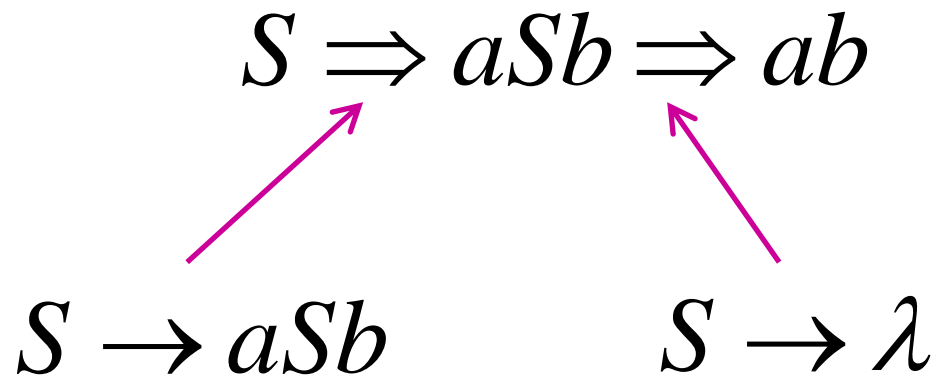
Variable

The right side
may be λ

Grammar: $S \rightarrow aSb$

$S \rightarrow \lambda$

Derivation of string ab :



Grammar: $S \rightarrow aSb$

$S \rightarrow \lambda$

Derivation of string $aabb$:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$



$S \rightarrow aSb$

$S \rightarrow \lambda$

Grammar: $S \rightarrow aSb$

$S \rightarrow \lambda$

Other derivations:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbbb$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$
 $\Rightarrow aaaaSbbbb \Rightarrow aaabbbbb$

Grammar:

$$S \rightarrow aSb$$

$$S \rightarrow ab$$

Language of the grammar:

$$L = \{a^n b^n : n > 0\}$$

A Convenient Notation

We write: $S \stackrel{*}{\Rightarrow} aaabbb$

for zero or more derivation steps

Instead of:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbbb$$

In general we write: $w_1 \xRightarrow{*} w_n$

If: $w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \cdots \Rightarrow w_n$

in zero or more derivation steps

Trivially: $w \xRightarrow{*} w$

Example Grammar

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

Possible Derivations

$$\begin{array}{c} * \\ S \Rightarrow \lambda \end{array}$$

$$\begin{array}{c} * \\ S \Rightarrow ab \end{array}$$

$$\begin{array}{c} * \\ S \Rightarrow aaabbb \end{array}$$

$$S \xRightarrow{*} aaSbb \xRightarrow{*} aaaaaaSbbbbbb$$

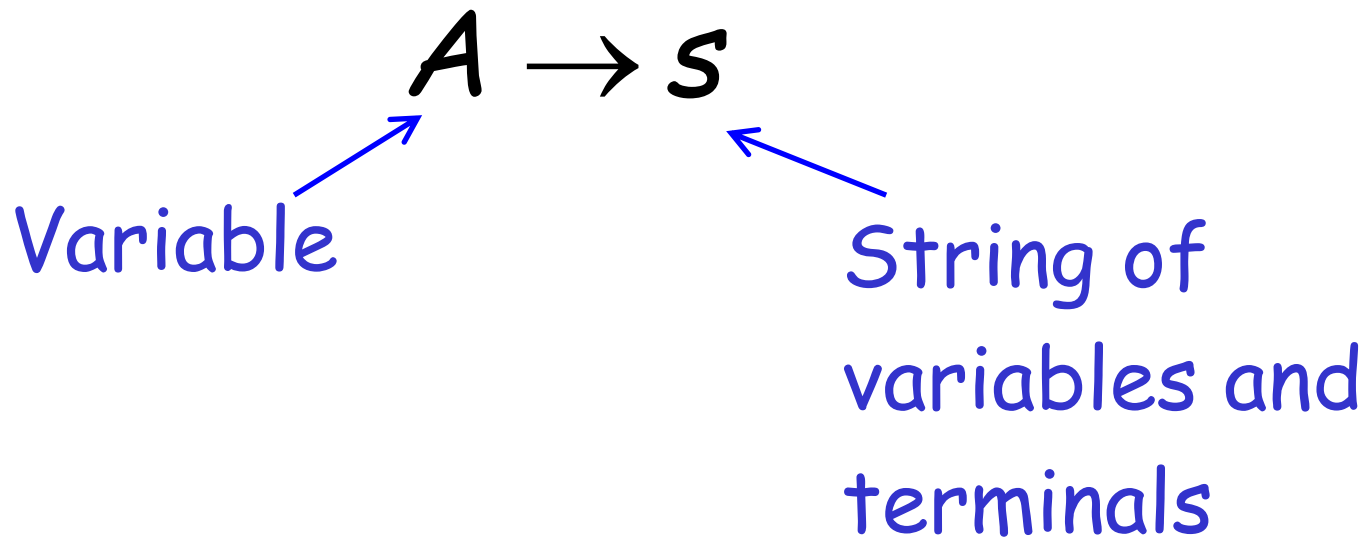
Another convenient notation:

$$\begin{array}{l} S \rightarrow aSb \\ S \rightarrow \lambda \end{array} \quad \longrightarrow \quad S \rightarrow aSb \mid \lambda$$

$$\begin{array}{l} \langle \textit{article} \rangle \rightarrow a \\ \langle \textit{article} \rangle \rightarrow \textit{the} \end{array} \quad \longrightarrow \quad \langle \textit{article} \rangle \rightarrow a \mid \textit{the}$$

Context-Free Grammar: $G = (V, T, S, P)$

All productions in P are of the form



Example of Context-Free Grammar

$$S \rightarrow aSb \mid \lambda$$

productions

$$P = \{S \rightarrow aSb, S \rightarrow \lambda\}$$

$$G = (V, T, S, P)$$

$V = \{S\}$
variables

$T = \{a, b\}$
terminals

start variable

Language of a Grammar:

For a grammar G with start variable S

$$L(G) = \{w : S \xRightarrow{*} w, \quad w \in T^*\}$$

String of terminals or λ



Example:

context-free grammar $G : \boxed{S \rightarrow aSb \mid \lambda}$

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

Since, there is derivation

$$S \xRightarrow{*} a^n b^n \quad \text{for any } n \geq 0$$

Context-Free Language:

A language L is context-free
if there is a context-free grammar G
with $L = L(G)$

Example:

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

is a context-free language

since context-free grammar G :

$$S \rightarrow aSb \mid \lambda$$

generates $L(G) = L$

Another Example

Context-free grammar G :

$$S \rightarrow aSa \mid bSb \mid \textit{epsilon}$$

Example derivations:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$$

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaba$$

$$L(G) = \{ ww^R : w \in \{a,b\}^* \}$$

Palindromes of even length

Another Example

Context-free grammar G :

$$S \rightarrow aSb \mid SS \mid \lambda$$

Example derivations:

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow ab$$

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab$$

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

$$\text{and } n_a(v) \geq n_b(v)$$

in any prefix v }

Describes
matched

parentheses:

$() ((())) (())$ $a = (, \quad b =)$

Derivation Order and Derivation Trees

Derivation Order

Consider the following example grammar with 5 productions:

- | | | |
|-----------------------|----------------------------|----------------------------|
| 1. $S \rightarrow AB$ | 2. $A \rightarrow aaA$ | 4. $B \rightarrow Bb$ |
| | 3. $A \rightarrow \lambda$ | 5. $B \rightarrow \lambda$ |

- | | | |
|-----------------------|----------------------------|----------------------------|
| 1. $S \rightarrow AB$ | 2. $A \rightarrow aaA$ | 4. $B \rightarrow Bb$ |
| | 3. $A \rightarrow \lambda$ | 5. $B \rightarrow \lambda$ |

Leftmost derivation order of string *aab*:

$$\begin{array}{ccccccccc} & 1 & & 2 & & 3 & & 4 & & 5 \\ S & \Rightarrow & AB & \Rightarrow & aaAB & \Rightarrow & aaB & \Rightarrow & aaBb & \Rightarrow & aab \end{array}$$

At each step, we substitute the
leftmost variable

- | | | |
|-----------------------|----------------------------|----------------------------|
| 1. $S \rightarrow AB$ | 2. $A \rightarrow aaA$ | 4. $B \rightarrow Bb$ |
| | 3. $A \rightarrow \lambda$ | 5. $B \rightarrow \lambda$ |

Rightmost derivation order of string aab :

$$\begin{array}{ccccccccc}
 & 1 & & 4 & & 5 & & 2 & & 3 \\
 S & \Rightarrow & AB & \Rightarrow & ABb & \Rightarrow & Ab & \Rightarrow & aaAb & \Rightarrow & aab
 \end{array}$$

At each step, we substitute the
rightmost variable

- | | | |
|-----------------------|----------------------------|----------------------------|
| 1. $S \rightarrow AB$ | 2. $A \rightarrow aaA$ | 4. $B \rightarrow Bb$ |
| | 3. $A \rightarrow \lambda$ | 5. $B \rightarrow \lambda$ |

Leftmost derivation of aab :

$$\begin{array}{ccccccccc}
 1 & & 2 & & 3 & & 4 & & 5 \\
 S & \Rightarrow & AB & \Rightarrow & aaAB & \Rightarrow & aaB & \Rightarrow & aaBb & \Rightarrow & aab
 \end{array}$$

Rightmost derivation of aab :

$$\begin{array}{ccccccccc}
 1 & & 4 & & 5 & & 2 & & 3 \\
 S & \Rightarrow & AB & \Rightarrow & ABb & \Rightarrow & Ab & \Rightarrow & aaAb & \Rightarrow & aab
 \end{array}$$

Derivation Trees

Consider the same example grammar:

$$S \rightarrow AB \quad A \rightarrow aaA \mid \lambda \quad B \rightarrow Bb \mid \lambda$$

And a derivation of aab :

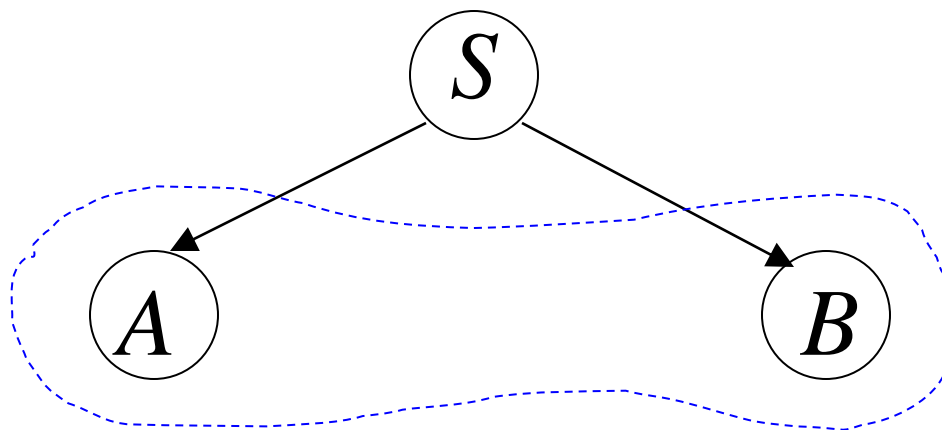
$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB$$



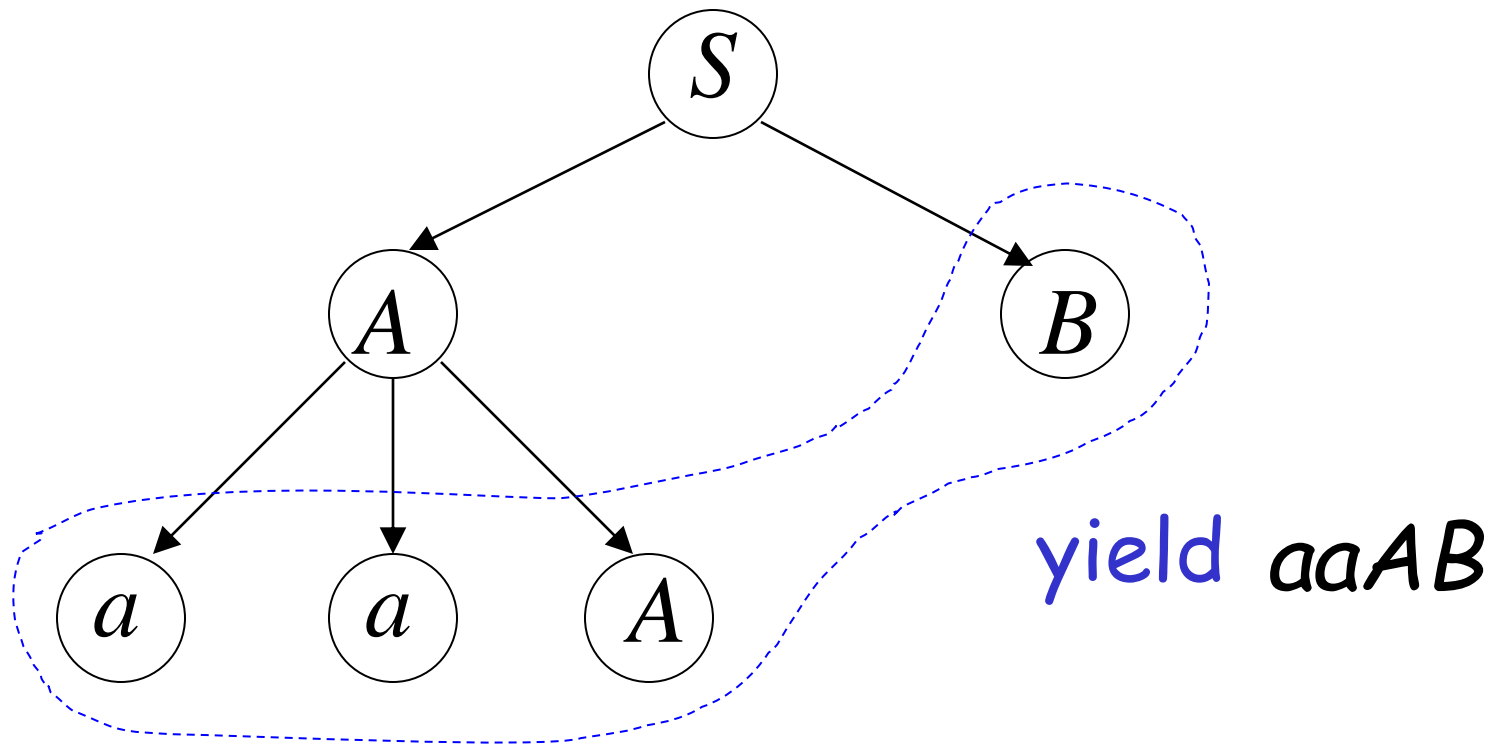
yield AB

$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

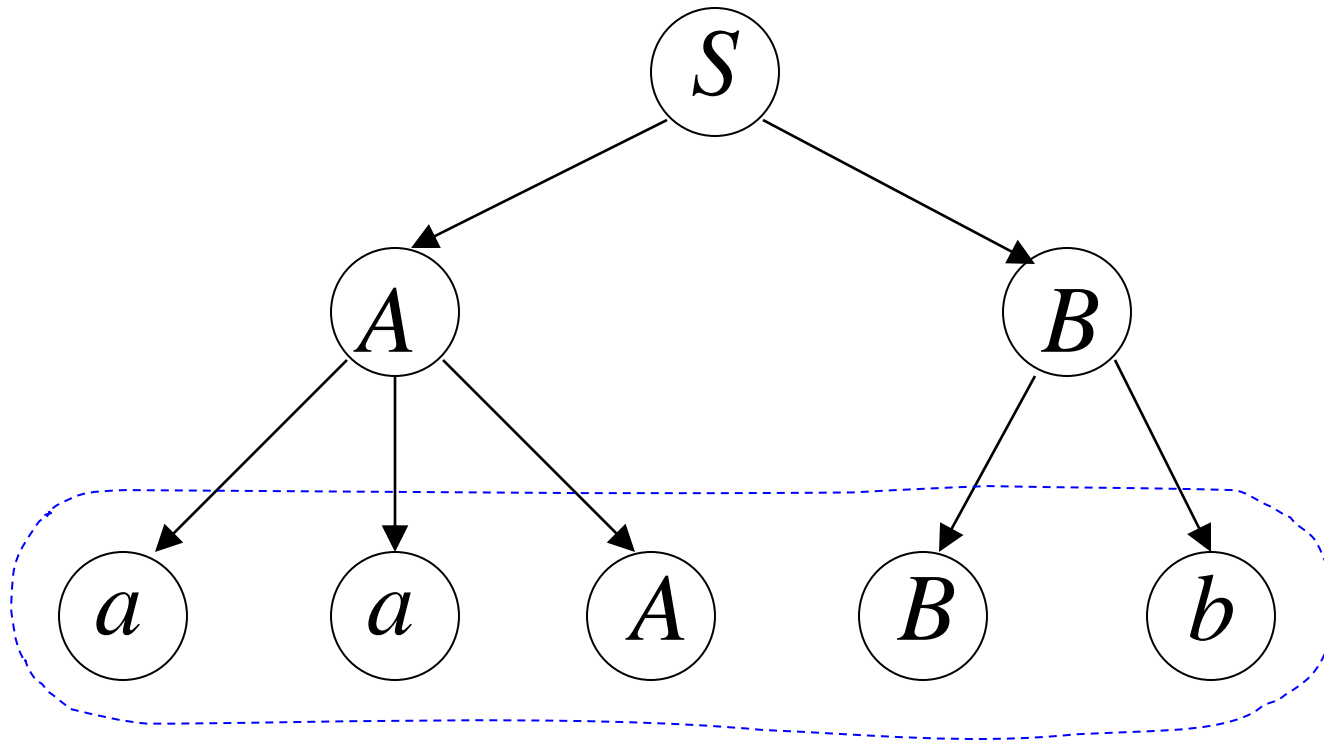
$$B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB$$



$$S \rightarrow AB \quad A \rightarrow aaA \mid \lambda \quad B \rightarrow Bb \mid \lambda$$

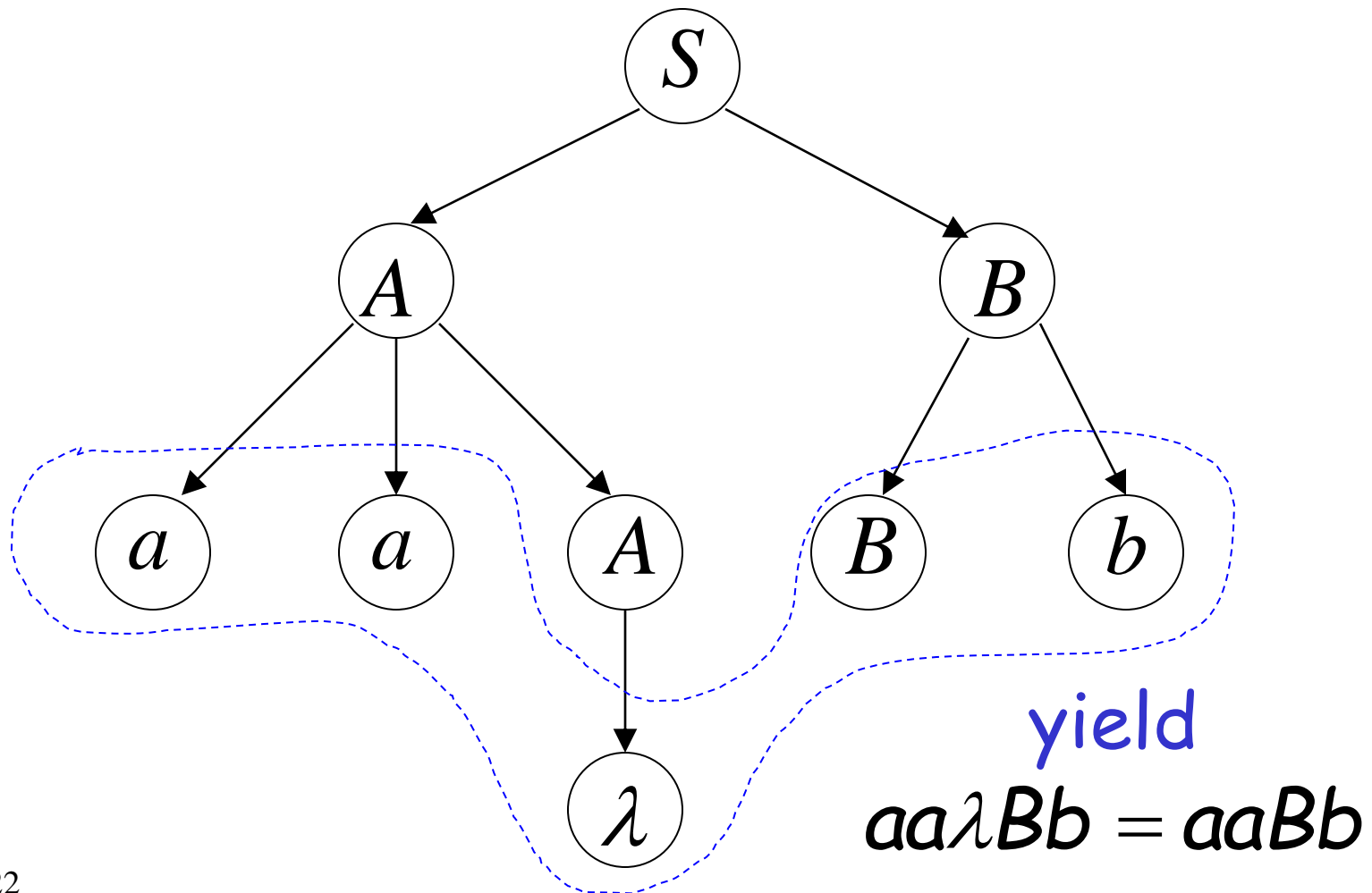
$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb$$



yield $aaABb$

$$S \rightarrow AB \quad A \rightarrow aaA \mid \lambda \quad B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb$$



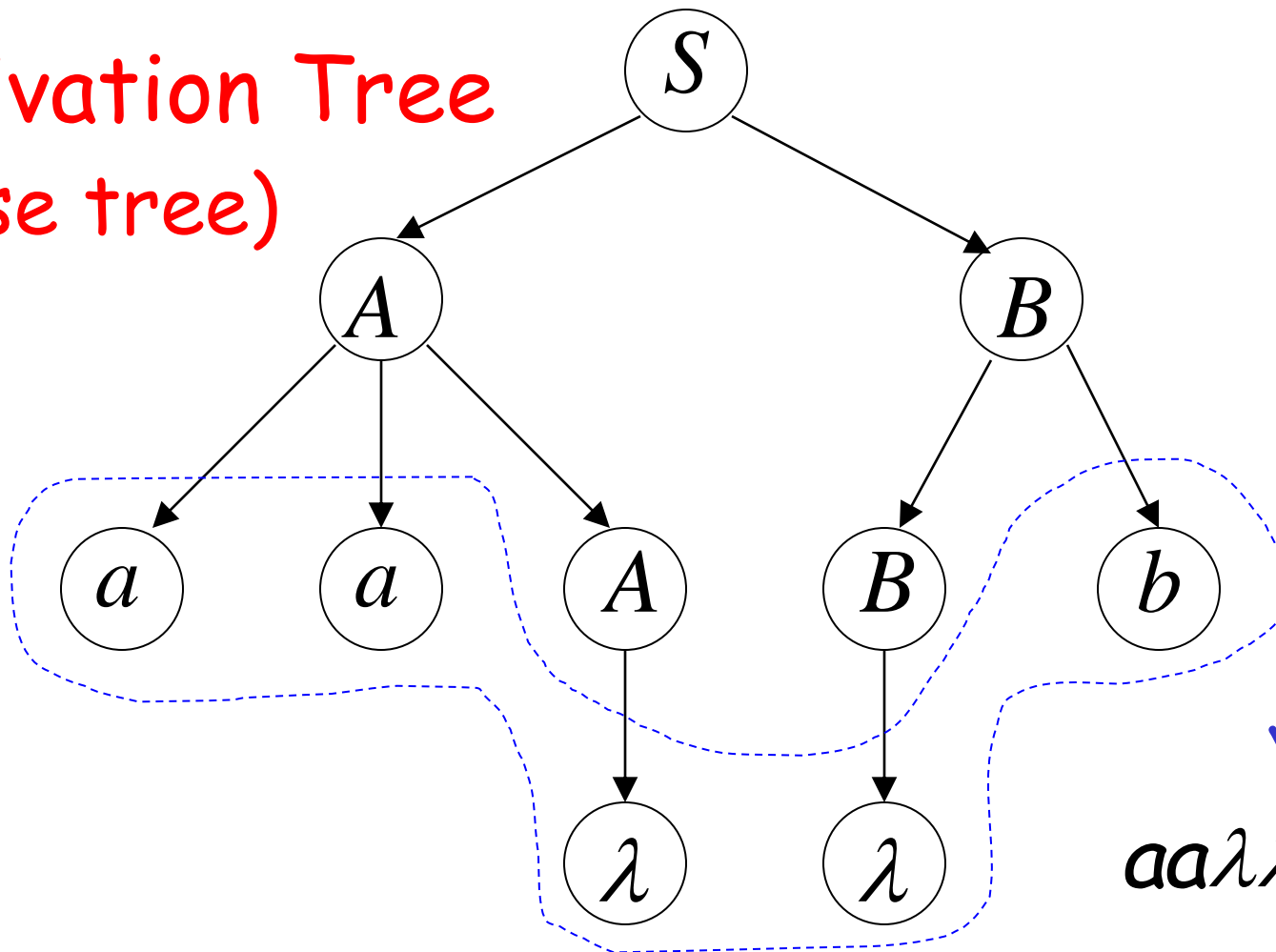
$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

Derivation Tree
(parse tree)



yield

$$aa\lambda\lambda b = aab$$

Sometimes, derivation order doesn't matter

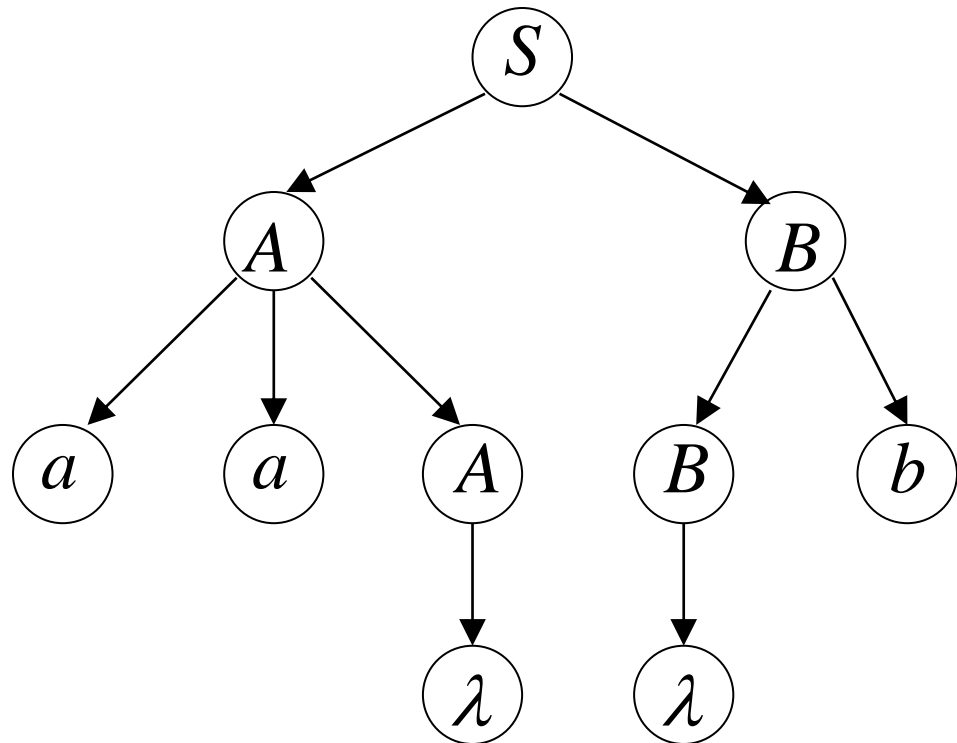
Leftmost derivation:

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aaBb \Rightarrow aab$$

Rightmost derivation:

$$S \Rightarrow AB \Rightarrow ABb \Rightarrow Ab \Rightarrow aaAb \Rightarrow aab$$

Give same
derivation tree



Ambiguity

Grammar for mathematical expressions

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

Example strings:

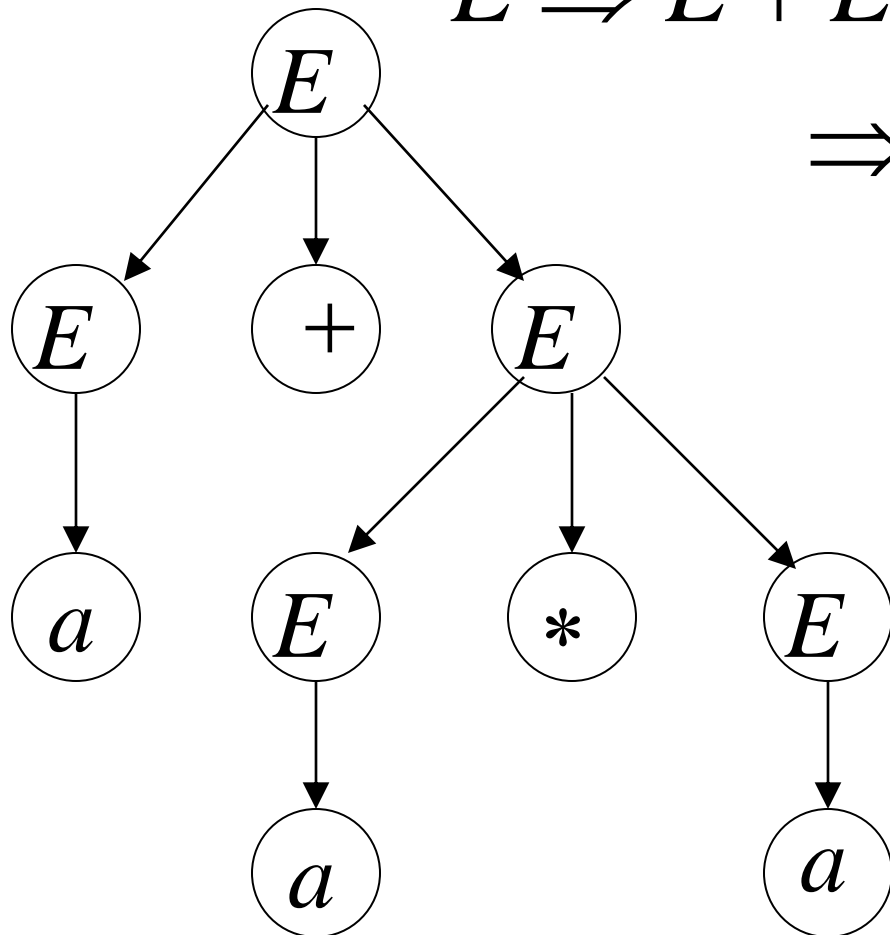
$$(a + a) * a + (a + a * (a + a))$$



Denotes any number

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \\ &\Rightarrow a + a * E \Rightarrow a + a * a \end{aligned}$$

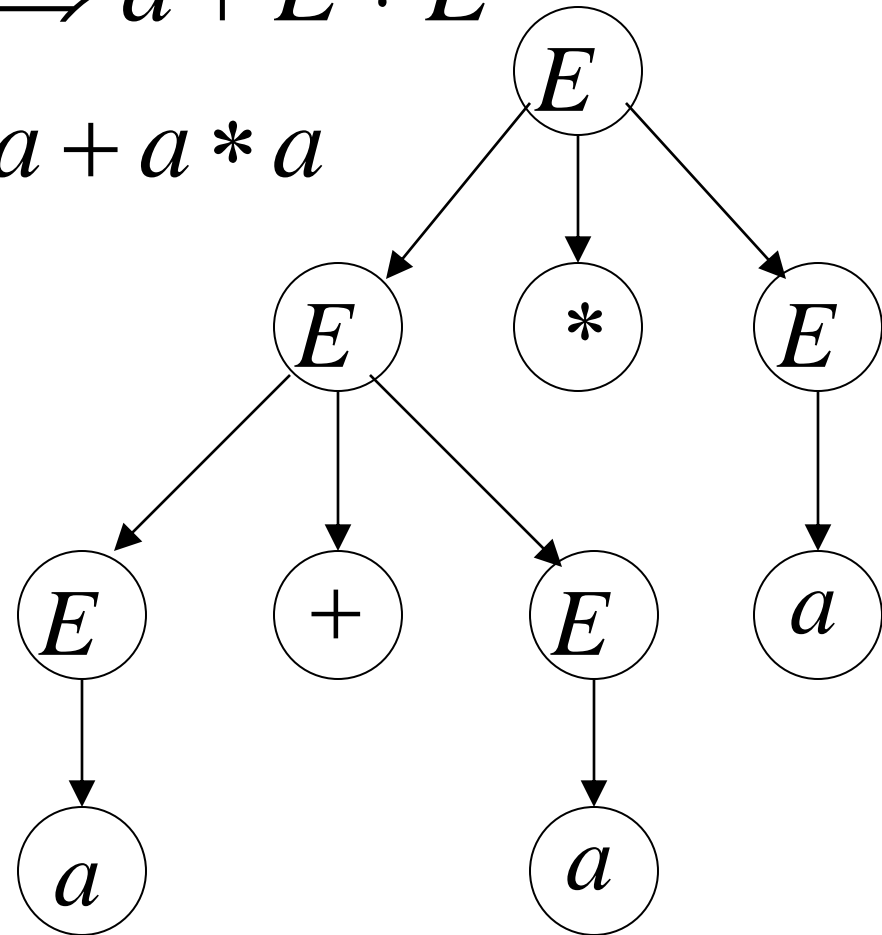


A leftmost derivation
for $a + a * a$

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

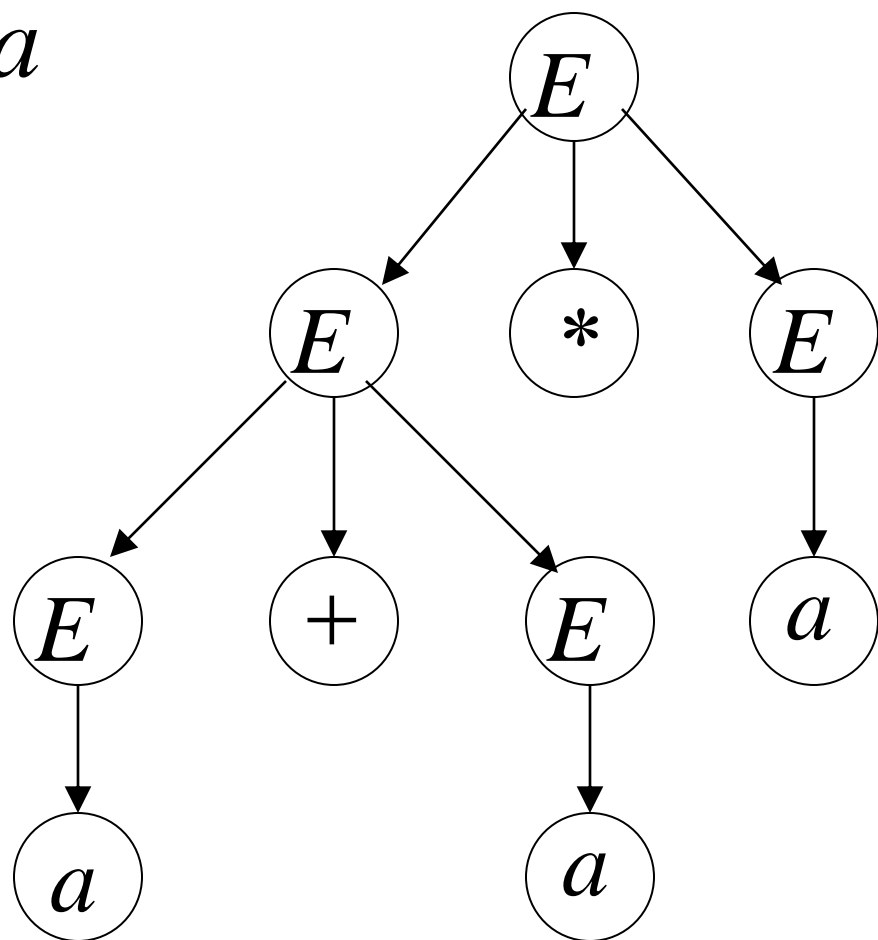
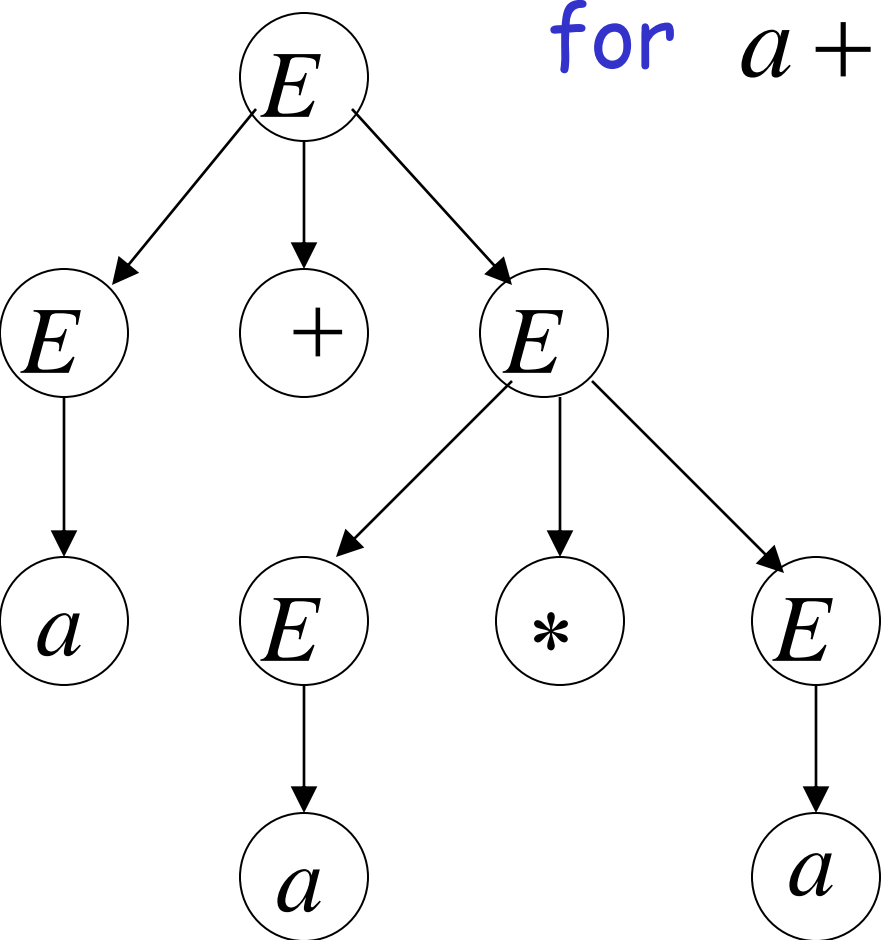
$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \\ \Rightarrow a + a * E \Rightarrow a + a * a$$

Another
leftmost derivation
for $a + a * a$



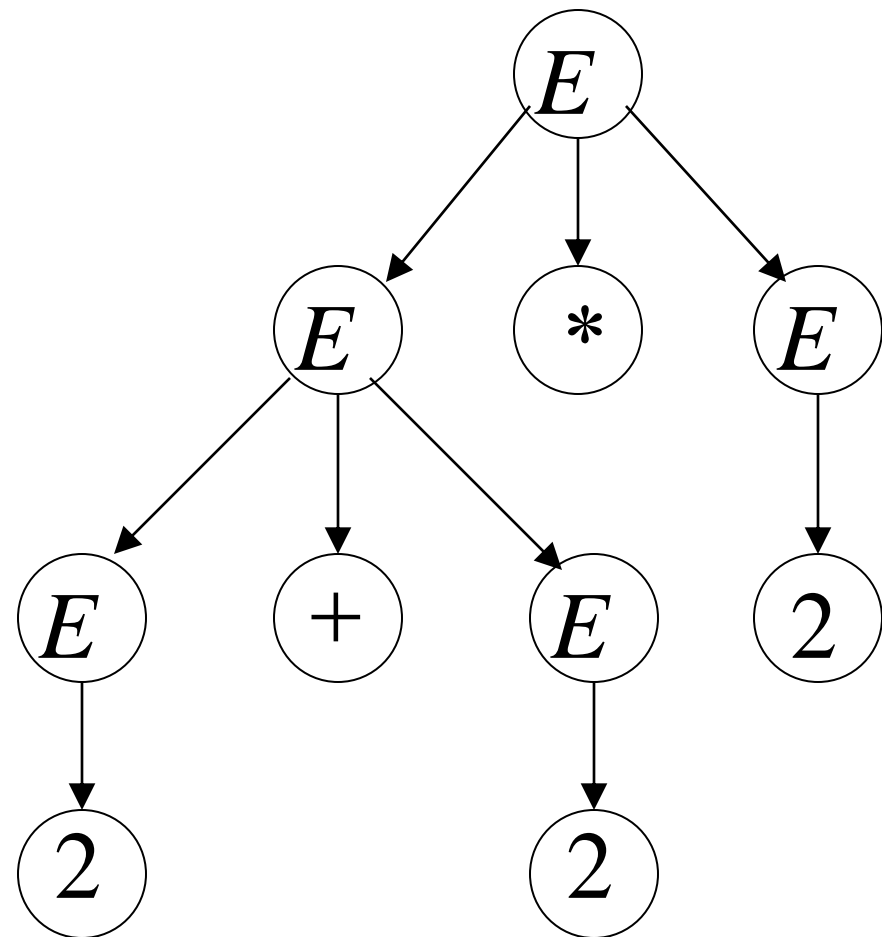
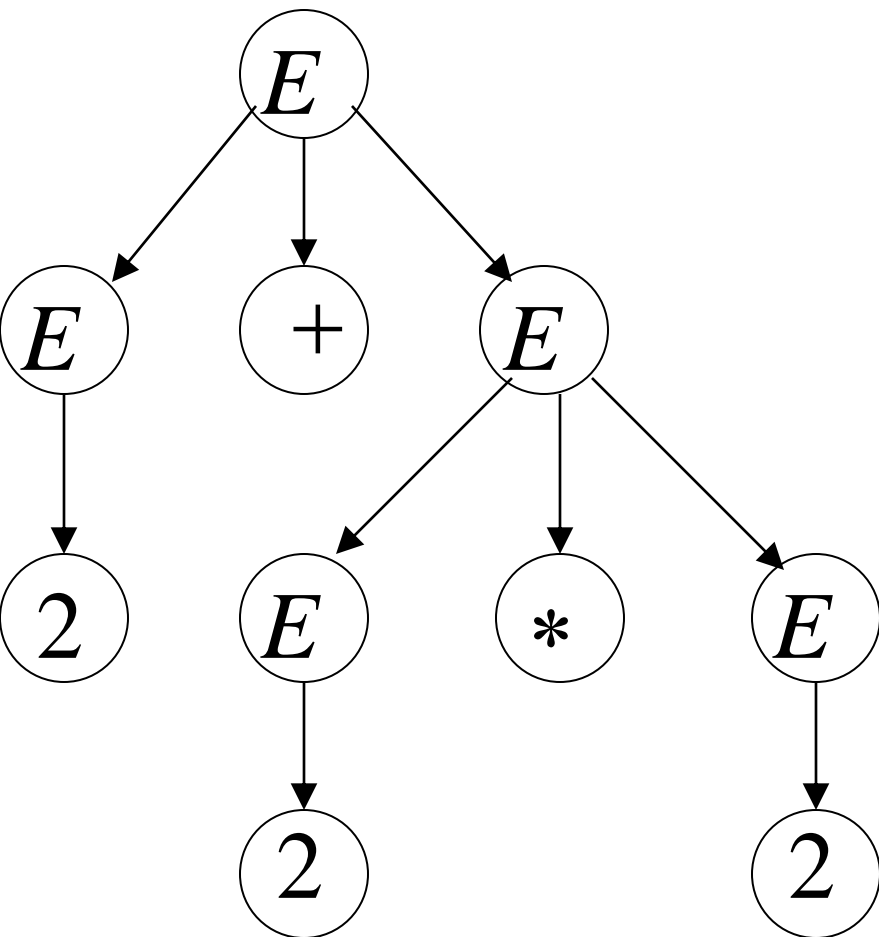
$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

Two derivation trees
for $a + a * a$



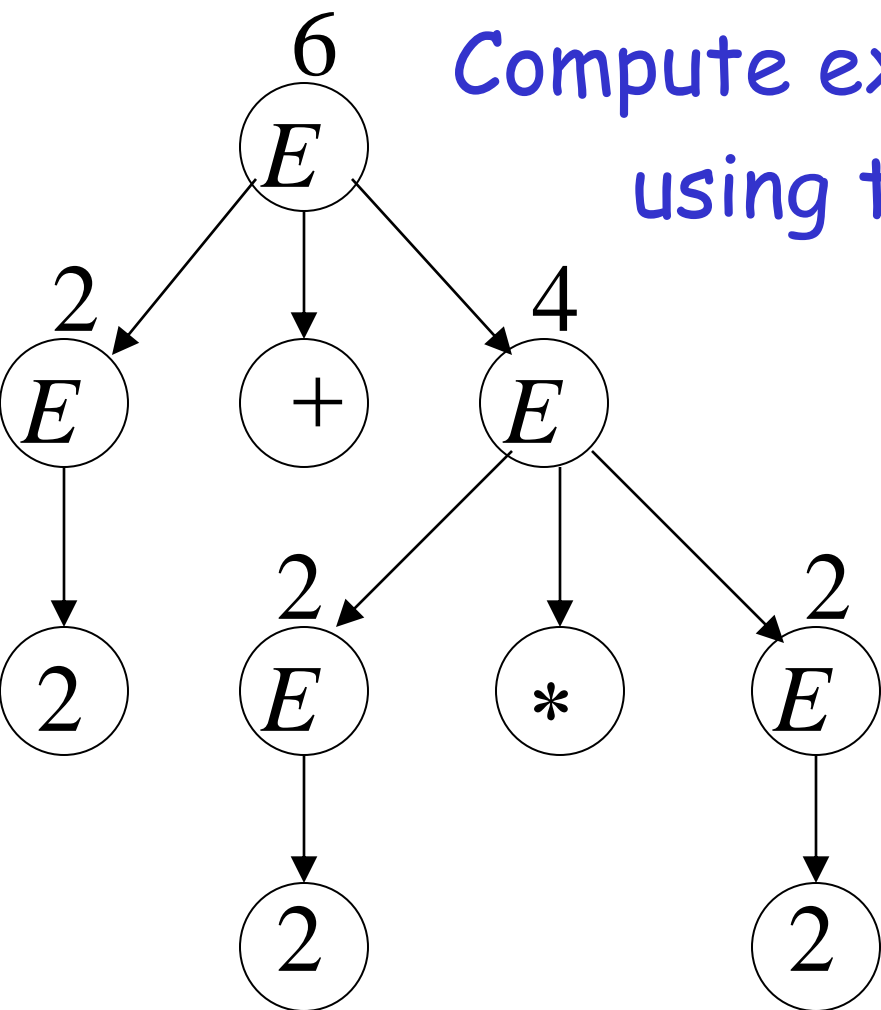
take $a = 2$

$$a + a * a = 2 + 2 * 2$$



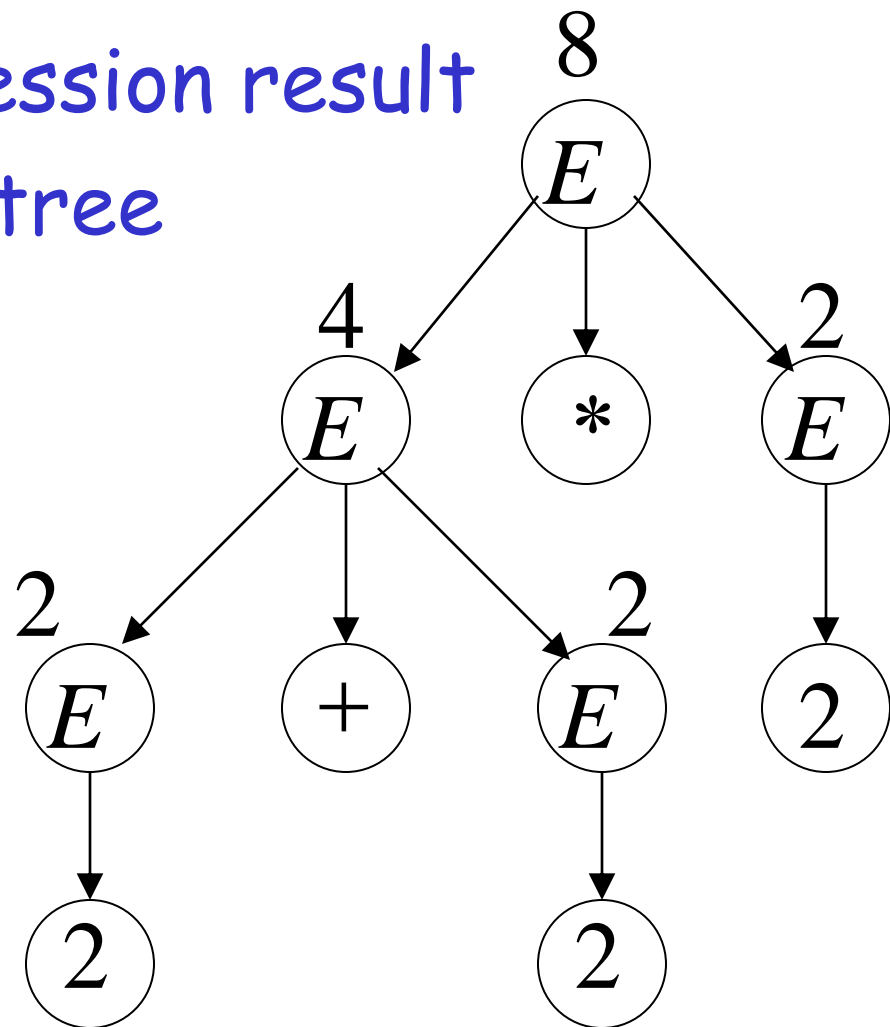
Good Tree

$$2 + 2 * 2 = 6$$



Bad Tree

$$2 + 2 * 2 = 8$$



Compute expression result
using the tree

Two different derivation trees
may cause problems in applications which
use the derivation trees:

- Evaluating expressions
- In general, in compilers
for programming languages

Ambiguous Grammar:

A context-free grammar G is ambiguous if there is a string $w \in L(G)$ which has:

two different derivation trees

or

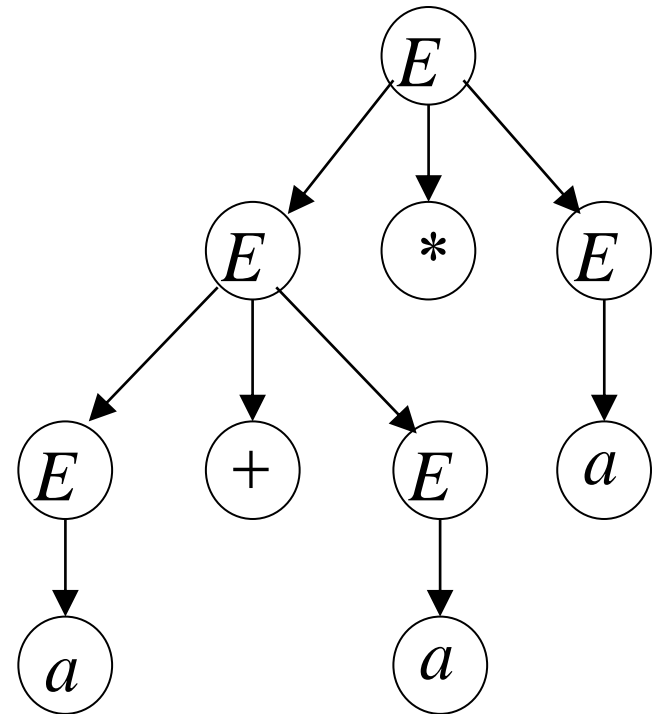
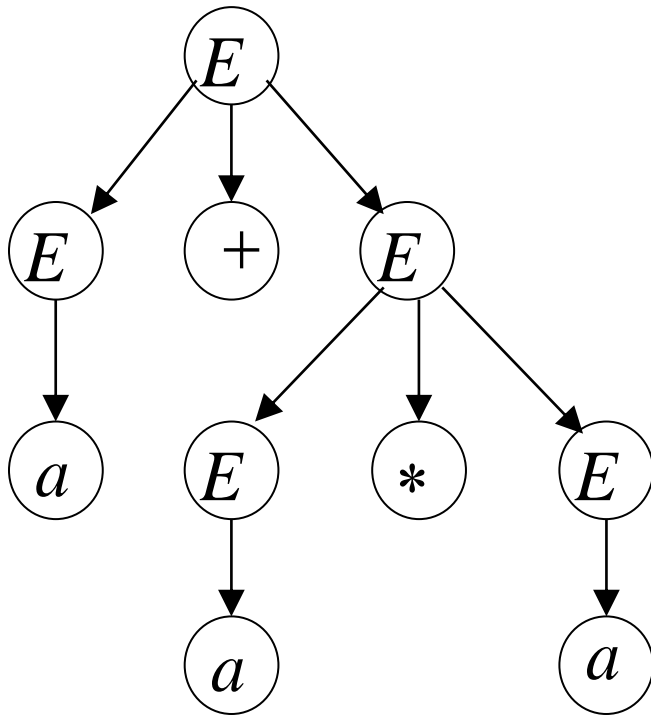
two leftmost derivations

(Two different derivation trees give two different leftmost derivations and vice-versa)

Example:

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

this grammar is ambiguous since
string $a + a * a$ has two derivation trees



$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

this grammar is ambiguous also because
string $a + a * a$ has two leftmost derivations

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \\ &\Rightarrow a + a * E \Rightarrow a + a * a \end{aligned}$$

$$\begin{aligned} E &\Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \\ &\Rightarrow a + a * E \Rightarrow a + a * a \end{aligned}$$

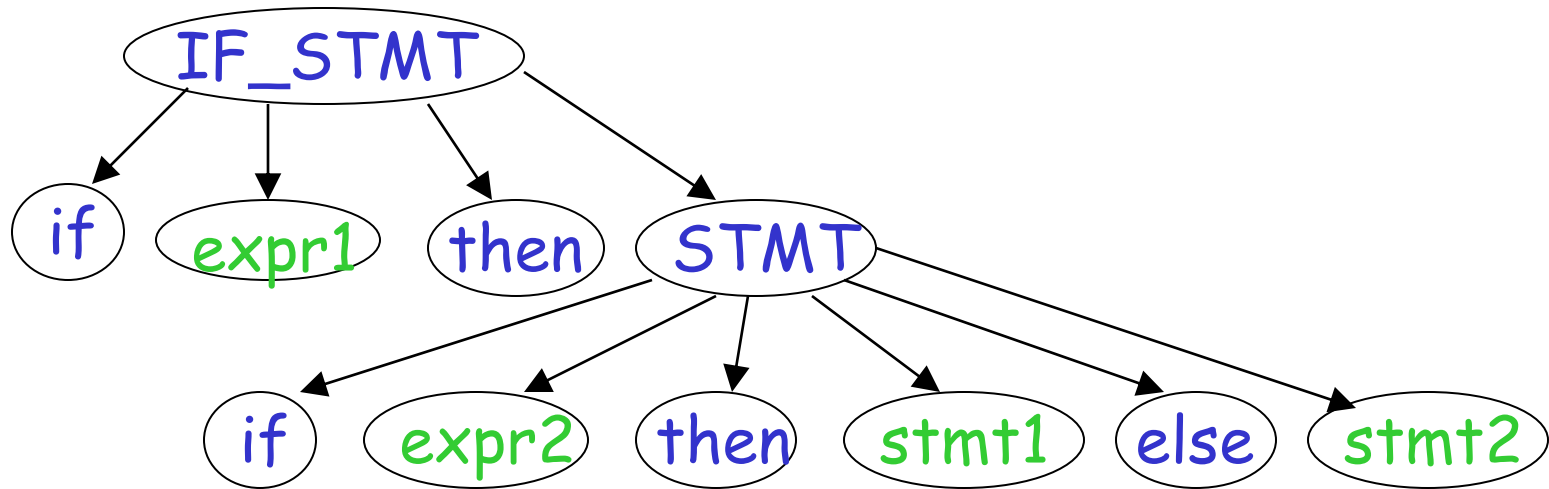
Another ambiguous grammar:

IF_STMT \rightarrow if EXPR then STMT
 | if EXPR then STMT else STMT

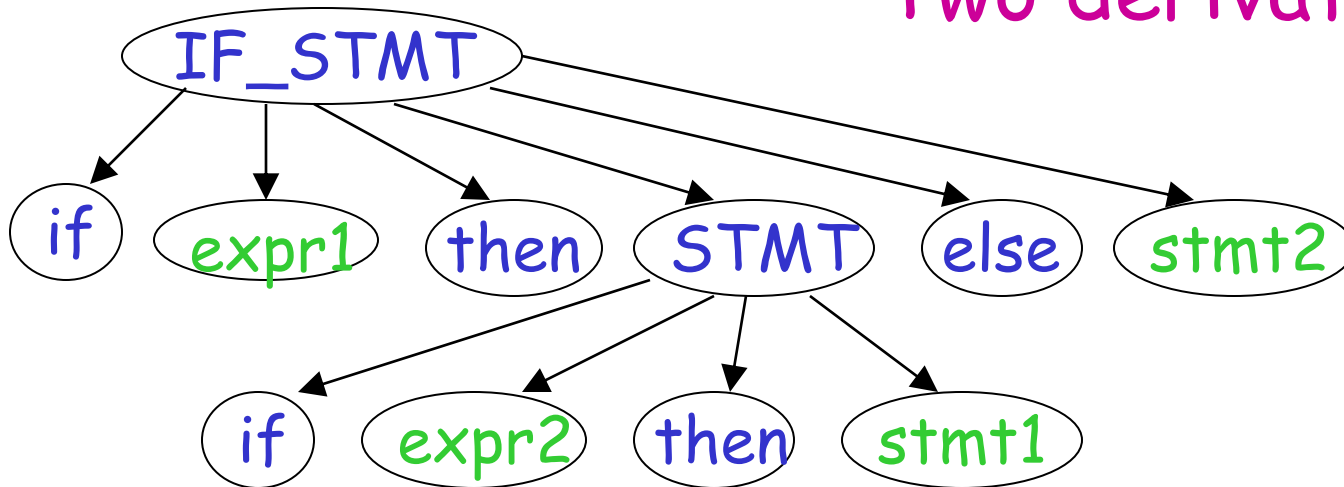
Variables Terminals

Very common piece of grammar
in programming languages

If expr1 then if expr2 then stmt1 else stmt2



Two derivation trees



In general, ambiguity is bad
and we want to remove it

Sometimes it is possible to find
a non-ambiguous grammar for a language

But, in general we cannot do so

A successful example:

Ambiguous Grammar

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ E &\rightarrow a \end{aligned}$$

Equivalent

Non-Ambiguous Grammar

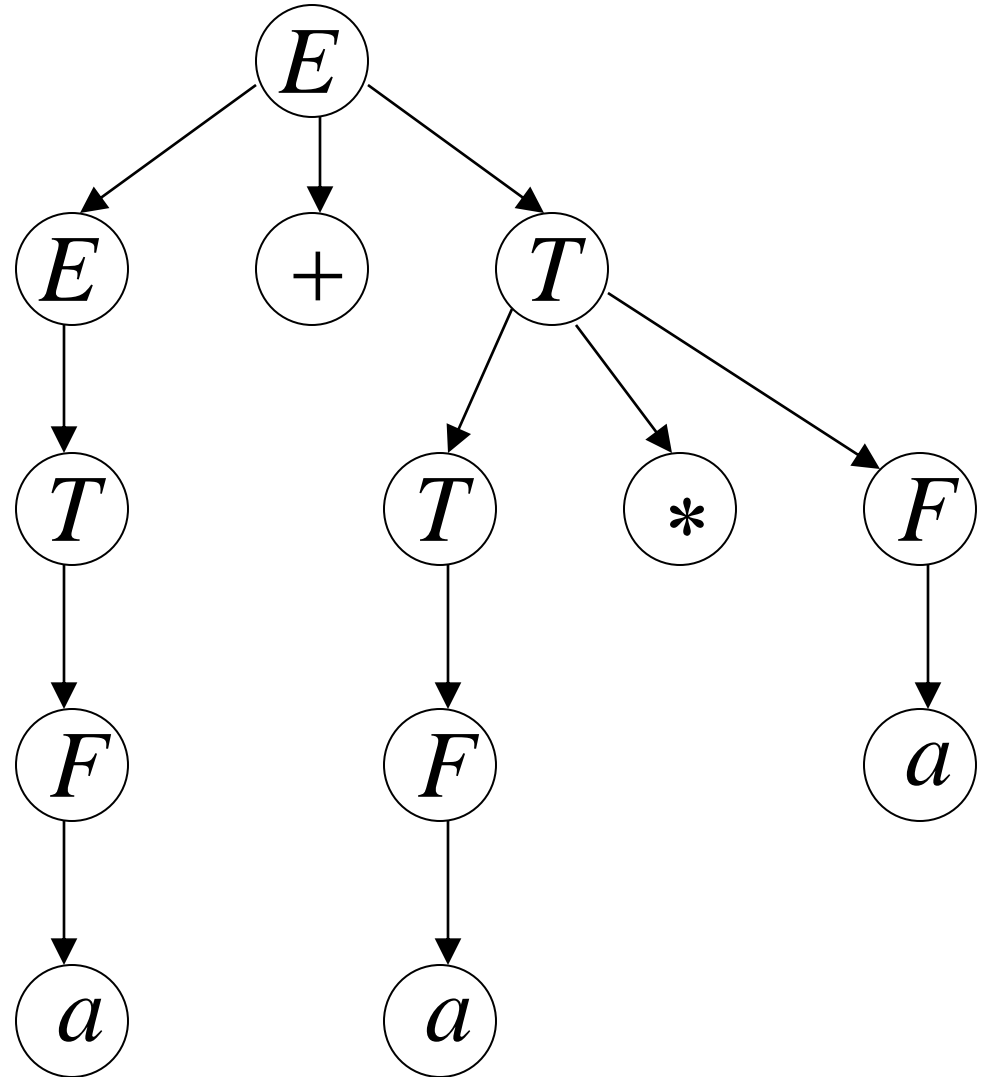
$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid a \end{aligned}$$

generates the same language

$$\begin{aligned}
 E &\Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T * F \\
 &\Rightarrow a + F * F \Rightarrow a + a * F \Rightarrow a + a * a
 \end{aligned}$$

$$\begin{aligned}
 E &\rightarrow E + T \mid T \\
 T &\rightarrow T * F \mid F \\
 F &\rightarrow (E) \mid a
 \end{aligned}$$

Unique
derivation tree
for $a + a * a$



An un-successful example:

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$$

$$n, m \geq 0$$

L is inherently ambiguous:

every grammar that generates this language is ambiguous

Example (ambiguous) grammar for L :

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$$


$$S \rightarrow S_1 \mid S_2$$


$$S_1 \rightarrow S_1 c \mid A$$

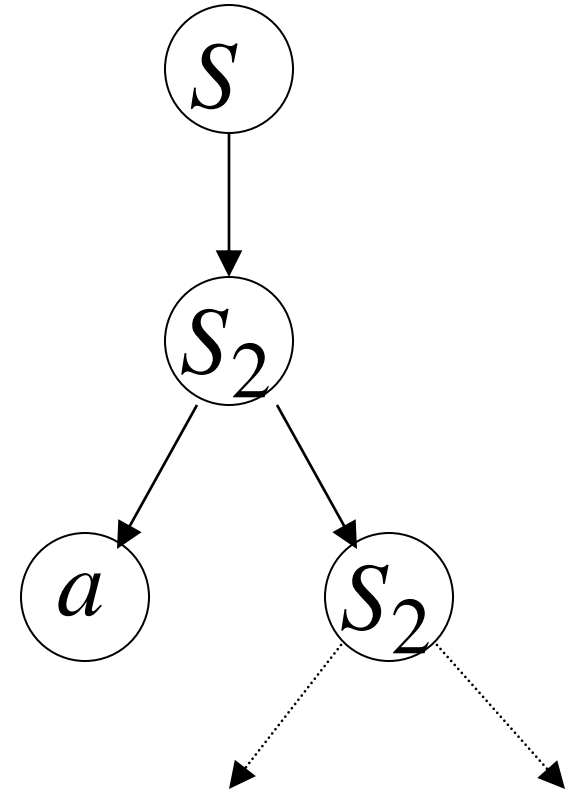
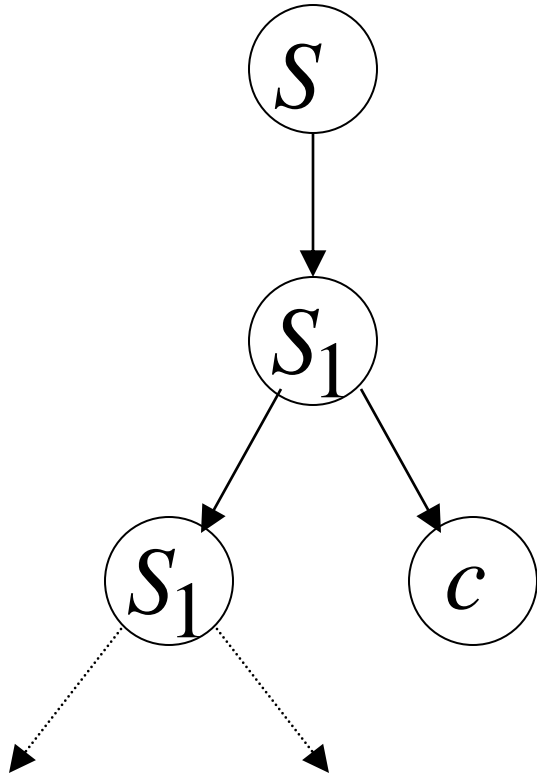
$$A \rightarrow aAb \mid \lambda$$


$$S_2 \rightarrow aS_2 \mid B$$

$$B \rightarrow bBc \mid \lambda$$

The string $a^n b^n c^n \in L$
has always two different derivation trees
(for any grammar)

For example



Applications of CFG

1. Context Free Grammars are used in Compilers (like GCC) for parsing. In this step, it takes a program (a set of strings).
2. Context Free Grammars are used to define the High Level Structure of a Programming Languages.
3. Every Context Free Grammars can be converted to a Parser which is a component of a Compiler that identifies the structure of a Program and converts the Program into a Tree.
4. Document Type Definition in XML is a Context Free Grammars which describes the HTML tags and the rules to use the tags in a nested fashion.

Applications of CFG

Pushdown Automata

Definition

Moves of the PDA

Languages of the PDA

Deterministic PDA's

Pushdown Automata

- The PDA is an automaton equivalent to the CFG in language-defining power.
- Two types of PDAs: Deterministic and non deterministic PDA
- Only the nondeterministic PDA defines all the CFL's.
- But the deterministic version models parsers.
 - Most programming languages have deterministic PDA's.

PDA Formal Definition

□ A PDA is described by:

$$M = \{Q, \Sigma, \overset{\checkmark}{\Gamma}, \delta, q_0, \underline{Z_0}, \underline{F}\}$$



1. A finite set of *states* (Q , typically).
2. An *input alphabet* (Σ , typically).
3. A *stack alphabet* (Γ , typically).
4. A *transition function* (δ , typically).
5. A *start state* (q_0 , in Q , typically).
6. A *start symbol* (Z_0 , in Γ , typically).
7. A set of *final states* ($F \subseteq Q$, typically).

Conventions

- a, b, \dots are input symbols.
 - But sometimes we allow ϵ as a possible value.
- \dots, X, Y, Z are stack symbols.
- \dots, w, x, y, z are strings of input symbols.
- α, β, \dots are strings of stack symbols.

The Transition Function

- Takes three arguments:
 1. A state, in Q .
 2. An input, which is either a symbol in Σ or ϵ .
 3. A stack symbol in Γ .
- $\delta(q, a, Z)$ is a set of zero or more actions of the form (p, α) .
 - p is a state; α is a string of stack symbols.

Actions of the PDA

- If $\delta(q, a, Z)$ contains (p, α) among its actions, then one thing the PDA can do in state q , with a at the front of the input, and Z on top of the stack is:
 1. Change the state to p .
 2. Remove a from the front of the input (but a may be ϵ).
 3. Replace Z on the top of the stack by α .

Example: PDA

- Design a PDA to accept $\{0^n 1^n \mid n \geq 1\}$.
- The states:
 - q = start state. We are in state q if we have seen only 0's so far.
 - p = we've seen at least one 1 and may now proceed only if the inputs are 1's.
 - f = final state; accept.

Example: PDA – (2)

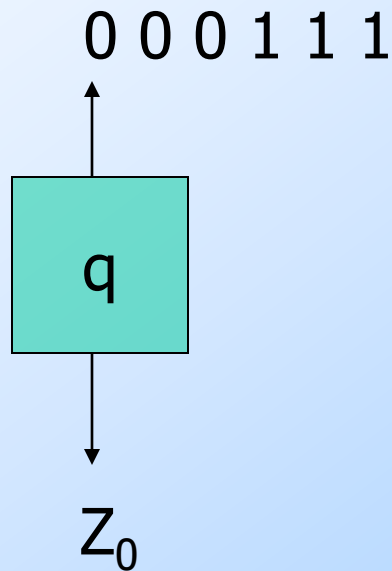
- The stack symbols:
 - Z_0 = start symbol. Also marks the bottom of the stack, so we know when we have counted the same number of 1's as 0's.
 - X = marker, used to count the number of 0's seen on the input.

Example: PDA – (3)

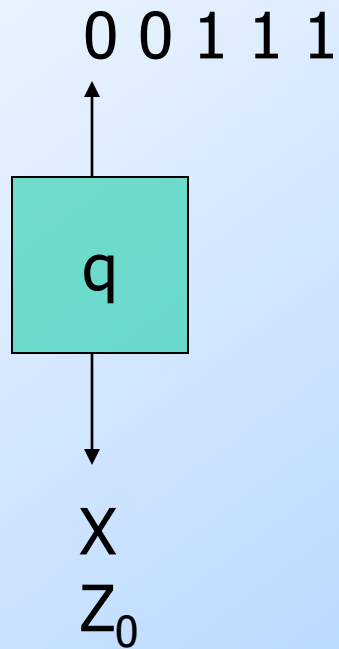
□ The transitions:

- $\delta(q, 0, Z_0) = \{(q, XZ_0)\}$.
- $\delta(q, 0, X) = \{(q, XX)\}$. These two rules cause one X to be pushed onto the stack for each 0 read from the input.
- $\delta(q, 1, X) = \{(p, \epsilon)\}$. When we see a 1, go to state p and pop one X.
- $\delta(p, 1, X) = \{(p, \epsilon)\}$. Pop one X per 1.
- $\delta(p, \epsilon, Z_0) = \{(f, Z_0)\}$. Accept at bottom.

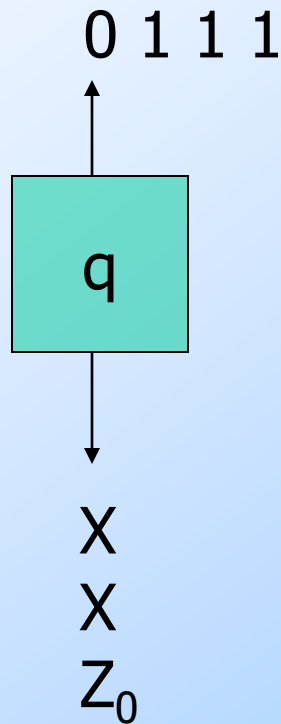
Actions of the Example PDA



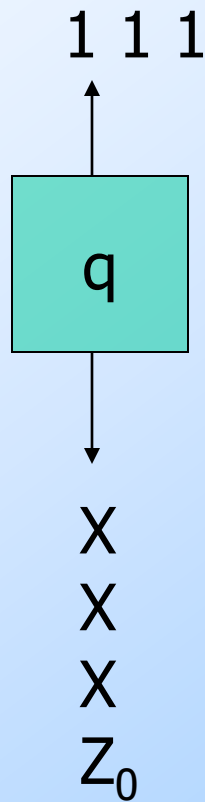
Actions of the Example PDA



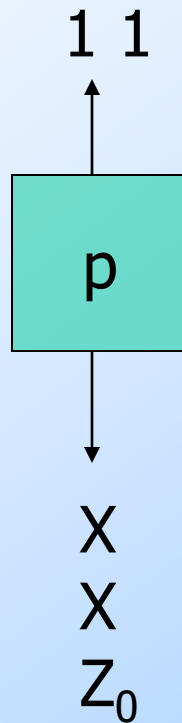
Actions of the Example PDA



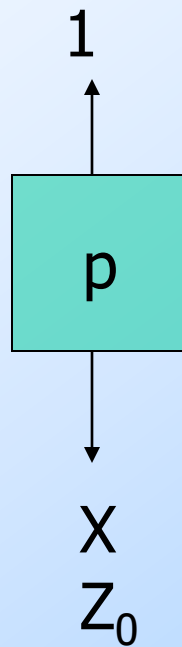
Actions of the Example PDA



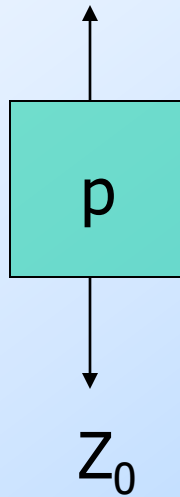
Actions of the Example PDA



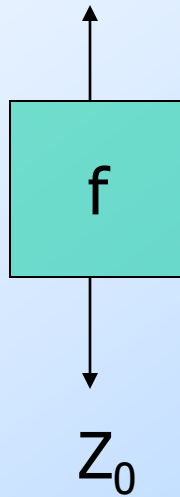
Actions of the Example PDA



Actions of the Example PDA



Actions of the Example PDA



Instantaneous Descriptions

- We can formalize the pictures just seen with an *instantaneous description* (ID).
- A ID is a triple (q, w, α) , where:
 1. q is the current state.
 2. w is the remaining input.
 3. α is the stack contents, top at the left.

The “Goes-To” Relation


- To say that ID I can become ID J in one move of the PDA, we write $I \vdash J$.
- Formally, $(q, aw, X\alpha) \vdash (p, w, \beta\alpha)$ for any w and α , if $\delta(q, a, X)$ contains (p, β) .
- Extend \vdash to \vdash^* , meaning “zero or more moves,” by:
 - **Basis:** $I \vdash^* I$.
 - **Induction:** If $I \vdash^* J$ and $J \vdash K$, then $I \vdash^* K$.

Example: Goes-To

- Using the previous example PDA, we can describe the sequence of moves by:
 $(q, 000111, Z_0) \vdash (q, 00111, XZ_0) \vdash$
 $(q, 0111, XXZ_0) \vdash (q, 111, XXXZ_0) \vdash$
 $(p, 11, XXZ_0) \vdash (p, 1, XZ_0) \vdash (p, \epsilon, Z_0) \vdash$
 (f, ϵ, Z_0)
- Thus, $(q, 000111, Z_0) \vdash^* (f, \epsilon, Z_0)$.
- What would happen on input 0001111?

Answer

Legal because a PDA can use ϵ input even if input remains.



- $(q, 0001111, Z_0) \vdash (q, 001111, XZ_0) \vdash (q, 01111, XXZ_0) \vdash (q, 1111, XXXZ_0) \vdash (p, 111, XXZ_0) \vdash (p, 11, XZ_0) \vdash (p, 1, Z_0) \vdash (f, 1, Z_0)$
- Note the last ID has no move.
- 0001111 is **not** accepted, because the input is not completely consumed.

Aside: FA and PDA Notations

- We represented moves of a FA by an extended δ , which did not mention the input yet to be read.
- We could have chosen a similar notation for PDA's, where the FA state is replaced by a state-stack combination, like the pictures just shown.

FA and PDA Notations – (2)

- Similarly, we could have chosen a FA notation with ID's.
 - Just drop the stack component.
- Why the difference? **My theory:**
- FA tend to model things like protocols, with indefinitely long inputs.
- PDA model parsers, which are given a fixed program to process.

Language of a PDA

- The common way to define the language of a PDA is by *final state*.
- If P is a PDA, then $L(P)$ is the set of strings w such that $(q_0, w, Z_0) \vdash^* (f, \epsilon, \alpha)$ for final state f and any α .

Language of a PDA – (2)

- Another language defined by the same PDA is by *empty stack*.
- If P is a PDA, then $N(P)$ is the set of strings w such that $(q_0, w, Z_0) \vdash^* (q, \epsilon, \epsilon)$ for any state q .

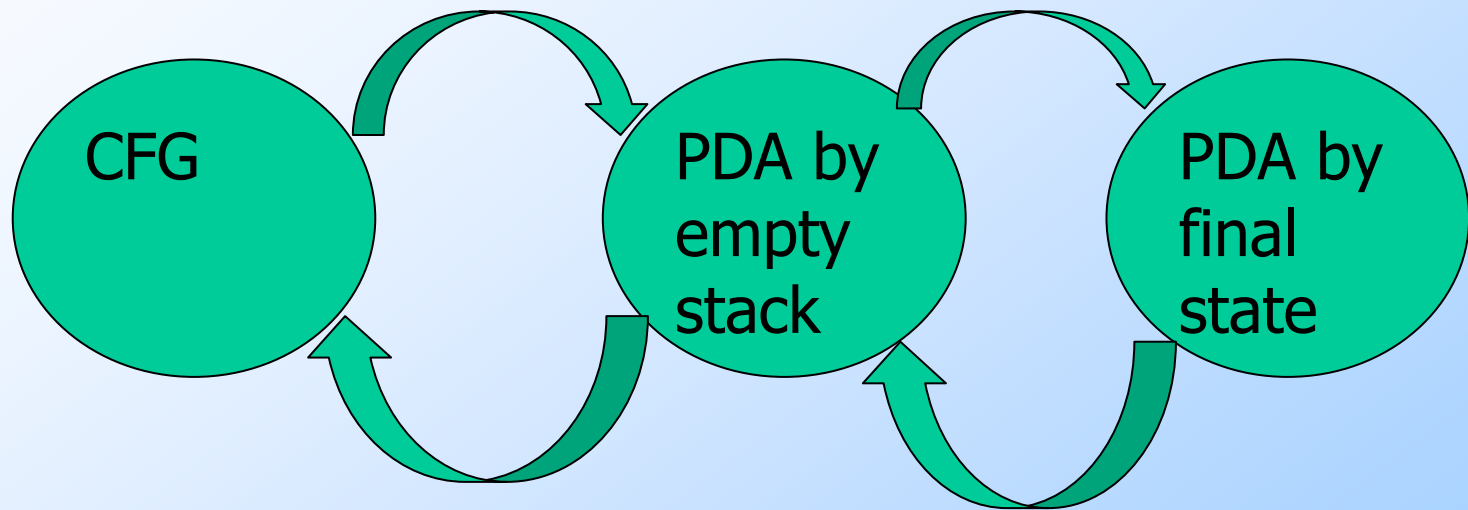
Equivalence of Language Definitions

1. If $L = L(P)$, then there is another PDA P' such that $L = N(P')$.
2. If $L = N(P)$, then there is another PDA P'' such that $L = L(P'')$.

Deterministic PDA's

- To be deterministic, there must be at most one choice of move for any state q , input symbol a , and stack symbol X .
- In addition, there must not be a choice between using input ϵ or real input.
- Formally, $\delta(q, a, X)$ and $\delta(q, \epsilon, X)$ cannot both be nonempty.

PDA and CFG



Construction of PDA from CFG

From a given CFG $G = (V, T, P, S)$, we can construct a PDA, M that simulates the leftmost derivation of G .

The PDA accepting $L(G)$ by empty stack is given by :

$$M = (\{q\}, T, V \cup T, \delta, q, S, \phi)$$

[M is a PDA for $L(G)$]

Where δ is defined by :

1. For each variable $A \in V$, include a transition,

$$\delta(q, \epsilon, A) \Rightarrow \{(q, \alpha) \mid A \rightarrow \alpha \text{ is a production in } G\}$$

2. For each terminal $a \in T$, include a transition

$$\delta(q, a, a) \Rightarrow \{(q, \epsilon)\}$$

Ex 1: Construction of PDA from CFG

Find a PDA for the given grammar
 $S \rightarrow 0S1 \mid 00 \mid 11$

Solution :

The equivalent PDA, M is given by :

$M = (\{q\}, \{0, 1\}, \{0, 1, S\}, \delta, q, S, \phi)$, where δ is given by :

$$\delta(q, \epsilon, S) = \{(q, 0S1), (q, 00), (q, 11)\}$$

$$\delta(q, 0, 0) = \{(q, \epsilon)\}$$

$$\delta(q, 1, 1) = \{(q, \epsilon)\}$$

Ex 2: Construction of PDA from CFG

Convert the grammar

$S \rightarrow 0S1 \mid A$

$A \rightarrow 1A0 \mid S \mid \epsilon$

to PDA that accepts the same language by empty stack.

Solution :

Step 1 : for each variable $A \in V$, include a transition

$\delta(q, \epsilon, A) \Rightarrow \{(q, \alpha) \mid A \rightarrow \alpha \text{ is a production in } G\}$

$\delta(q, \epsilon, S) \Rightarrow \{(q, 0S1), (q, A)\}$

$\delta(q, \epsilon, A) \Rightarrow \{(q, 1A0), (q, S), (q, \epsilon)\}$

Step 2 : For each terminal $a \in T$, include a transition $\delta(q, a, a) \Rightarrow (q, \epsilon)$

$\therefore \delta(q, 0, 0) = \{(q, \epsilon)\}$

$\delta(q, 1, 1) = \{(q, \epsilon)\}$

Therefore, the PDA is given by :

$M = (\{q\}, \{0, 1\}, \{S, A, 0, 1\}, \delta, q, S, \phi)$

where δ is :

$\delta(q, \epsilon, S) = \{(q, 0S1), (q, A)\}$

$\delta(q, \epsilon, A) = \{(q, 1A0), (q, S), (q, \epsilon)\}$

$\delta(q, 0, 0) = \{(q, \epsilon)\}$

$\delta(q, 1, 1) = \{(q, \epsilon)\}$

Ex 3: Construction of PDA from CFG

Let G be the grammar given by

$S \rightarrow aABB \mid aAA, A \rightarrow aBB \mid a, B \rightarrow bBB \mid A.$

Construct NPDA that accepts the language generated by this grammar.

Solution : The equivalent PDA, M is given by :

$$M = (\{q\}, \{a, b\}, \{a, b, S, A, B\}, \delta, q, S, \phi)$$

where δ is given by :

$$\delta(q, \epsilon, S) \Rightarrow \{(q, aABB), (q, aAA)\}$$

$$\delta(q, \epsilon, A) \Rightarrow \{(q, aBB), (q, a)\}$$

$$\delta(q, \epsilon, B) \Rightarrow \{(q, bBB), (q, A)\}$$

$$\delta(q, a, a) \Rightarrow (q, \epsilon)$$

$$\delta(q, b, b) \Rightarrow (q, \epsilon)$$

For each production in given grammar

For each terminal in T .

Union of Regular languages with CFL

- **Union of Regular language with context free language –**
- As all regular languages are context-free.
- The union of both results in a context-free language.
- But it is always good to understand with the help of an example.
Let's take a language $L1 = \{0^*1^*\}$ (regular) and
 $L2 = \{0^n1^n \mid n \geq 0\}$ (context-free)

And let $L = L1 \cup L2$ which will result in union of both these languages and that will be :

**** $L = \{0^*1^*\}$** which is regular language, but since every regular language is context-free. So, we can say the union of two always results in context-free language.

Intersection of Regular languages with CFL

- the intersection of a regular and a context-free language always result in a context-free language.
- Let's take a language $L1 = \{0^*1^*\}$ (regular) and $L2 = \{0^n1^n \mid n \geq 0\}$ (context-free).
- Let $L = L1 \cap L2$ which will easily result in $L = \{0^n1^n \mid n \geq 0\}$ which is context-free .

Context Sensitive Grammars

- Context-sensitive grammars are the grammars those generate languages that can be recognized with a restricted class of Turing machines called linear-bounded automata.
- A **grammar** $G = (V, T, S, P)$ is context-sensitive, if all productions are of the form $x \rightarrow y$, where $x, y \in (V \cup T)^+$ and $|x| \leq |y|$.
- G is context-sensitive, if all rules in P are of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$, where $A \in V$, $\alpha, \beta \in (\Sigma \cup V)^*$, $\gamma \in (\Sigma \cup V)^+$. Context-sensitive grammar is also called 1-type grammar

Chomsky hierarchy

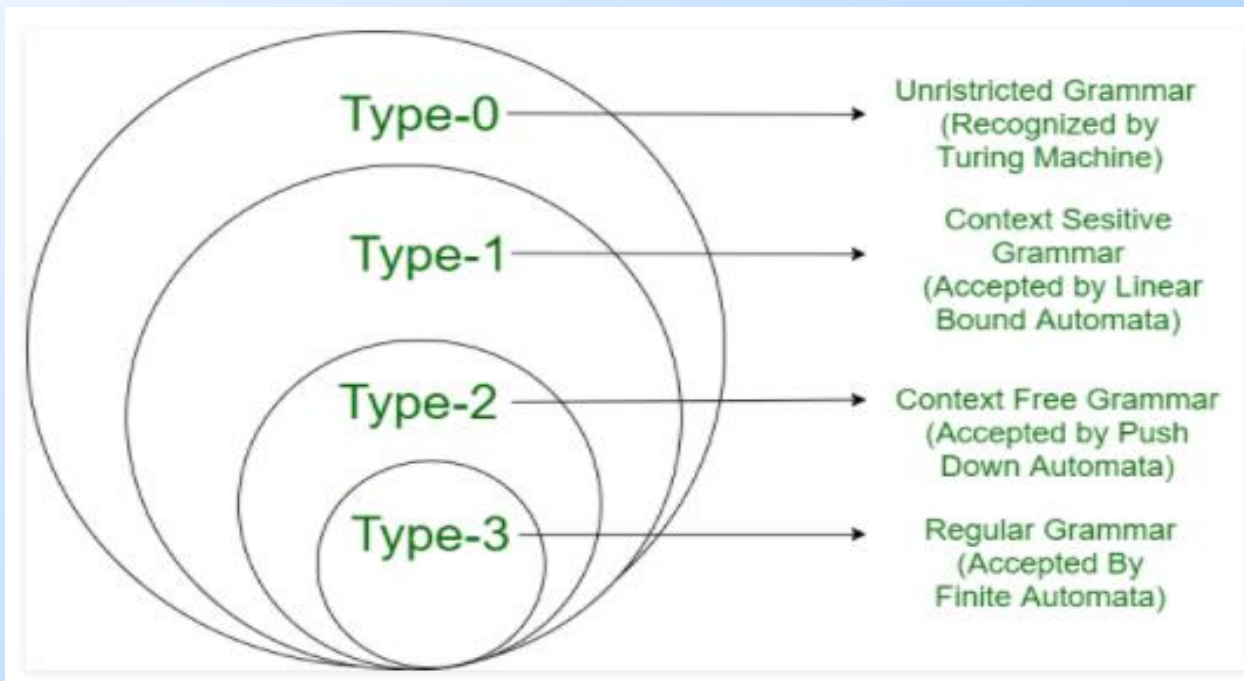
□ According to Chomsky hierarchy, grammars is divided into 4 types:

Type 0 known as **unrestricted grammar**.

Type 1 known as **context sensitive grammar**.

Type 2 known as **context free grammar**.

Type 3 **Regular Grammar**.



Unrestricted grammar

- No restrictions are made on the productions of an unrestricted grammar, other than each of their left-hand sides being non-empty.

Regular Grammar

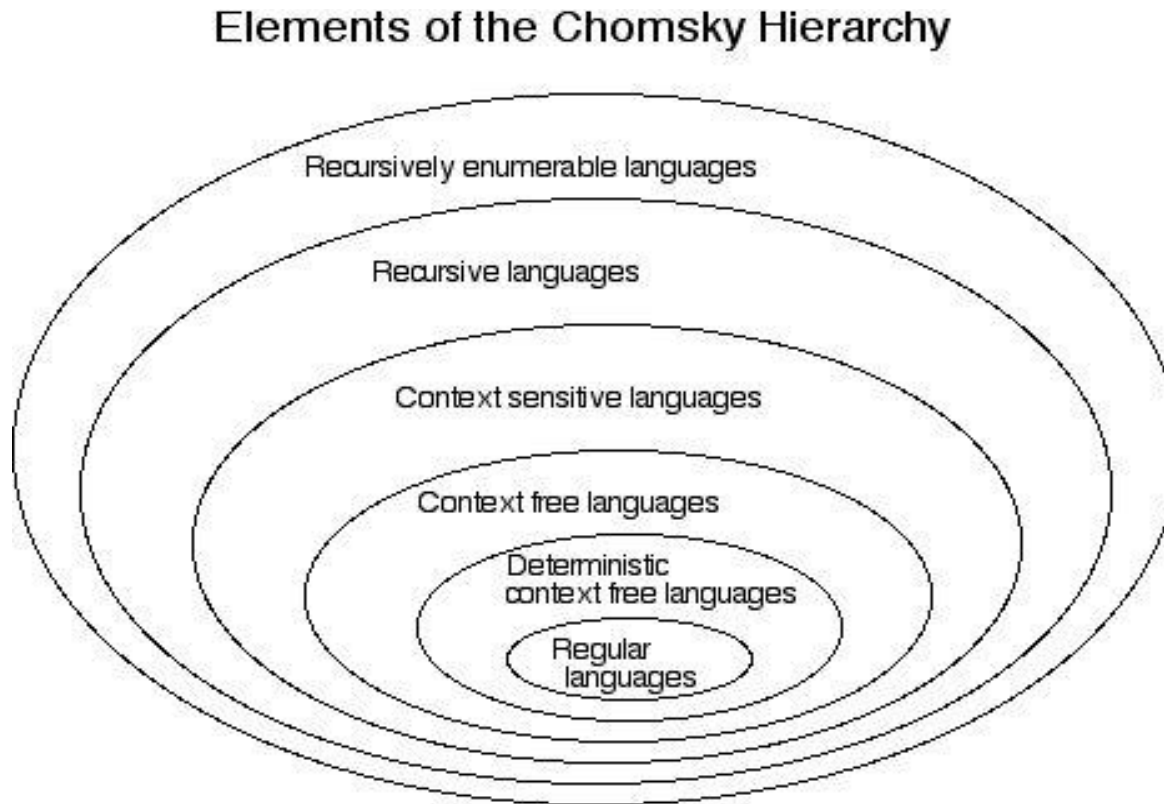
Undecidability & Intractable Problems

—

Syllabus

A Language that is not recursively enumerable, An undecidable problem that is RE, Post Correspondence Problem, The Classes P and NP : Problems Solvable in Polynomial Time, An Example: Kruskal's Algorithm, Nondeterministic Polynomial Time, An NP Example: The Traveling Salesman Problem, Polynomial-Time Reductions NP Complete Problems, An NP-Complete Problem: The Satisfiability Problem, Tractable and Intractable, Representing Satisfiability, Instances, NP Completeness of the SAT Problem, A Restricted Satisfiability Problem: Normal Forms for Boolean Expressions, Converting Expressions to CNF, The Problem of Independent Sets, The Node-Cover Problem.

Elements of the Chomsky Hierarchy



Decidable

A problem P is *decidable* if it can be solved by a Turing machine T that always halt. (We say that P has an effective algorithm.)

Note that the corresponding language of a decidable problem is *recursive*.

Un-decidable

A problem is *un-decidable* if it **cannot** be solved by any Turing machine that halts on all inputs.

Note that the corresponding language of an un-decidable problem is *non-recursive*.

Recursively Enumerable(RE) Language

A recursively enumerable language is a **formal language** for which there **exists a Turing machine** (or other computable function) that will **halt and accept** when presented with **any string** in the language as input but may **either halt and reject or loop forever** when presented with **a string not in the language**.

All regular, context-free, context-sensitive and recursive languages are recursively enumerable.

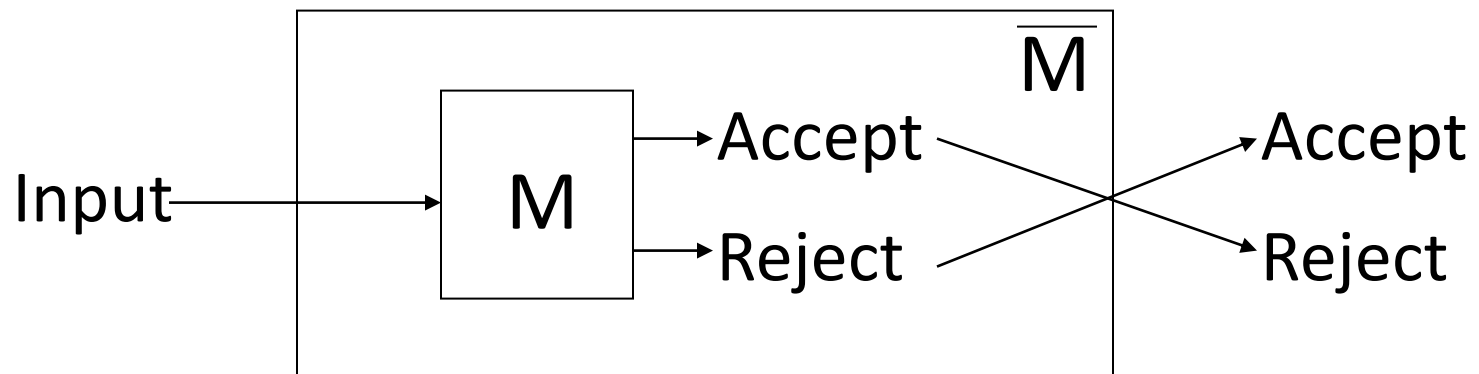
Difference between Recursive & RE Language

- **Recursively enumerable languages** as one for which a **partial decider** exists; that is, a Turing machine which, given as input a word over your alphabet, will either **correctly accept/reject** the word according to your language, or if the word is not in your language, it may loop forever.
- A **recursive language**, in contrast, is one for which a **total decider** exists, i.e. *one that will never loop, and always halt in either an accepting or a rejecting state.*

Complements of Recursive Languages

Theorem: If L is a recursive language, \overline{L} is also recursive.

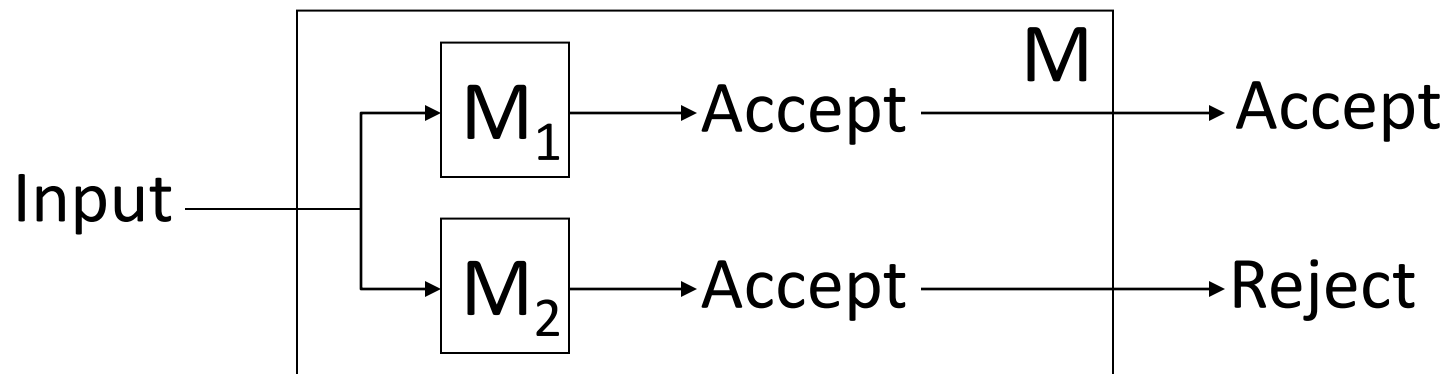
Proof: Let M be a TM for L that always halt. We can construct another TM \overline{M} for \overline{L} that always halts as follows:

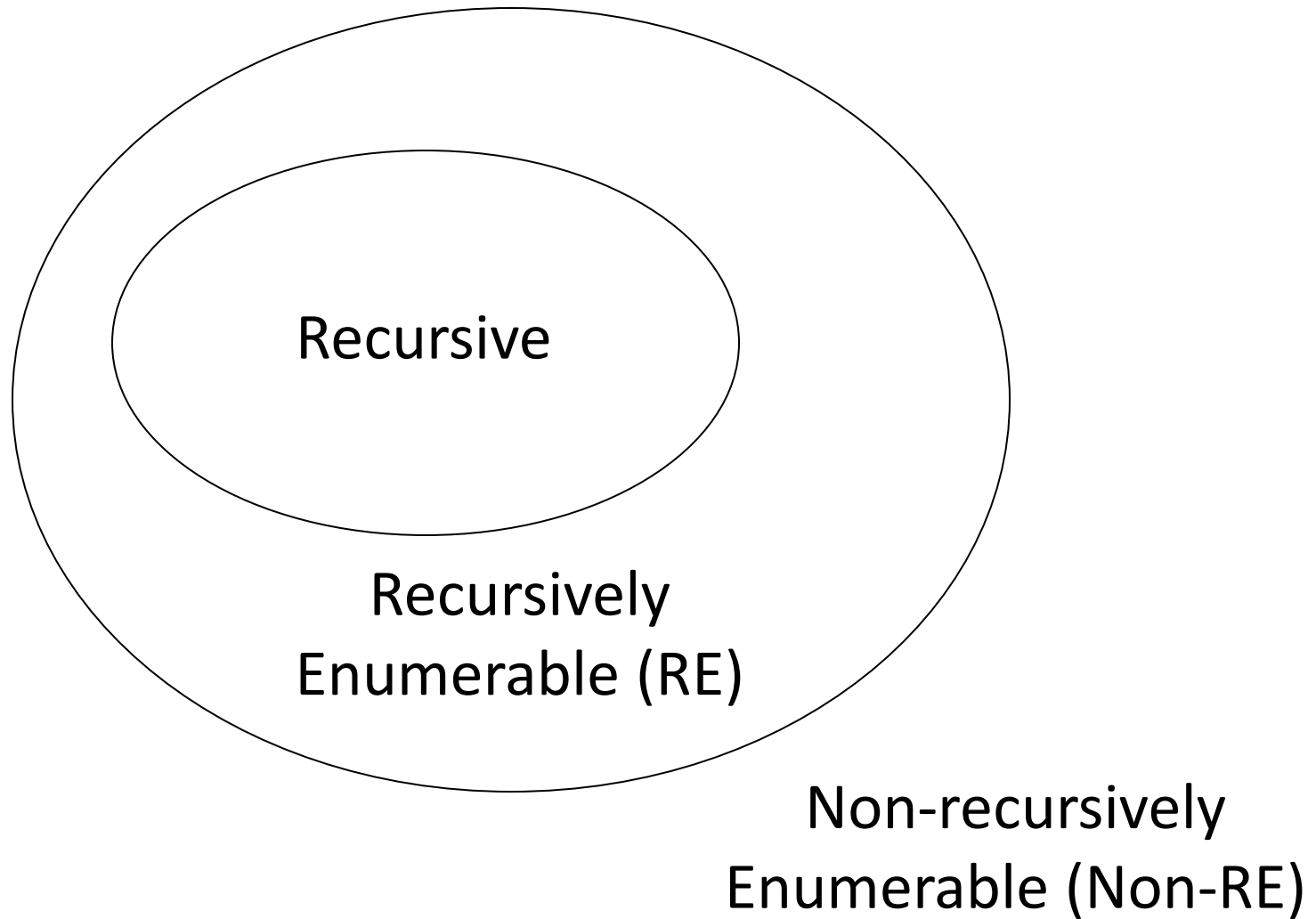


Complements of RE Languages

Theorem: If both a language L and its complement \bar{L} are RE, **L is recursive.**

Proof: Let M_1 and M_2 be TM for L and \bar{L} respectively. We can construct a TM M from M_1 and M_2 for L **that always halt** as follows:





A Non-recursive RE Language

- Recall that we can encode each TM uniquely as a binary number and enumerate all TM's as $T_1, T_2, \dots, T_k, \dots$ where the encoded value of the k^{th} TM, i.e., T_k , is k .
- Consider the language L_u :
$$L_u = \{(k, w) \mid T_k \text{ accepts input } w\}$$

This is called the *universal language*.

Universal Language

- Note that designing a TM to recognize L_u is the same as solving the problem of *given k and w , decide whether T_k accepts w as its input.*
- L_u is RE but non-recursive, i.e., L_u can be accepted by a TM, but there is no TM for L_u that always halt.

Modified Post Correspondence Problem

- We have seen an **undecidable** problem, that is, given a **Turing machine M** and an **input w** , determine whether M will accept w (universal language problem).
- We will study another **un-decidable** problem that is not related to Turing machine directly.

Post Correspondence Problem (PCP)

Given two lists A and B:

$$A = w_1, w_2, \dots, w_k \quad B = x_1, x_2, \dots, x_k$$

The problem is to determine if there is a sequence of one or more integers i_1, i_2, \dots, i_m such that:

$$w_1 w_{i_1} w_{i_2} \dots w_{i_m} = x_1 x_{i_1} x_{i_2} \dots x_{i_m}$$

(w_i, x_i) is called a corresponding pair.

Example:

	A	B
i	w_i	x_i
1	100	001
2	11	111
3	111	11

PC-solution:

$$A_2 A_1 A_3 = B_2 B_1 B_3$$

11100111

Difference between PCP & MPCP

*In the MPCP, a solution is required to start with the **first string** on each list.*

Example

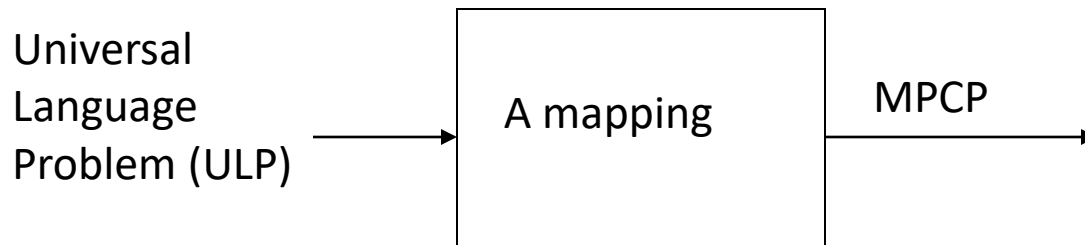
	A	B
i	w_i	x_i
1	11	1
2	1	111
3	0111	10
4	10	0

This PCP instance has a solution: **3, 2, 2, 4**:

$$w_1 w_3 w_2 w_2 w_4 = x_1 x_3 x_2 x_2 x_4 = 110111110$$

Un-decidability of MPCP

To show that MPCP is un-decidable, we will reduce the **Universal Language Problem (ULP)** to MPCP:



If MPCP can be solved, ULP can also be solved.
Since we have already shown that ULP is un-decidable, MPCP must also be un-decidable.

Problem?????



Def. :A Situation, Person or Thing that needs attention & needs to be dealt with or solved?

Types of Complexity

- There are two types of Complexity
 - Space Complexity
 - Time Complexity

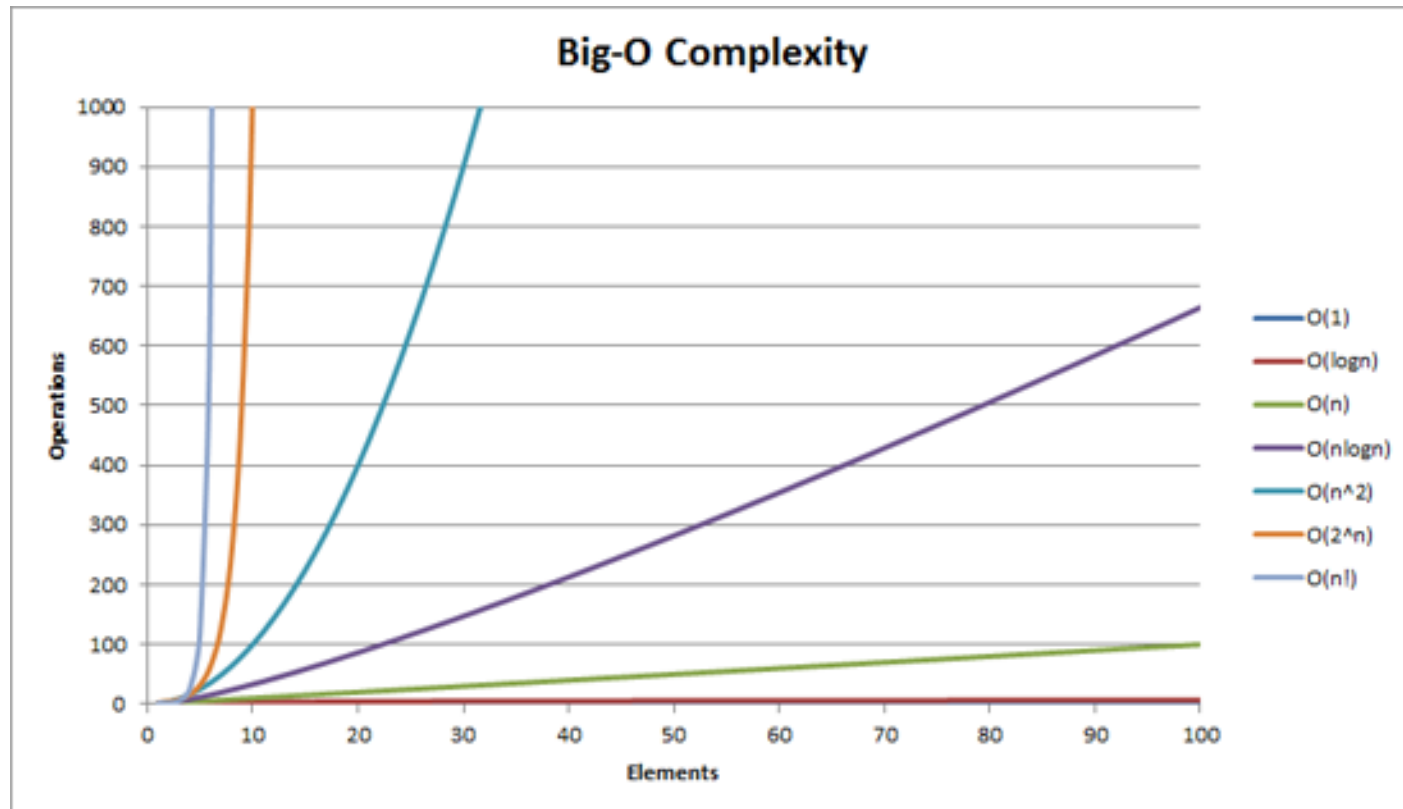
Algorithmic vs. Problem Complexity

- The ***Algorithmic complexity*** of a computation is some **measure of how difficult** is to perform the computation (i.e., **specific to an algorithm**)
- The **complexity of a computational *problem* or *task*** is the ***complexity of the algorithm*** with the **lowest** order of growth of complexity for solving that problem or performing that task.
- **Computational Complexity**: deals with classifying problems by ***how hard they are?***.

Complexity definitions

Usually, the time required by an algorithm falls under three types –

- **Best Case** – **Minimum** time required for program execution. **(Ω)**
- **Average Case** – **Average** time required for program execution. **(θ)**
- **Worst Case** – **Maximum** time required for program execution. **(O)**



Tractable vs. Intractable

- Polynomial Time Algorithm

-**worst case** complexity belongs to $O(n^k)$ for fixed integer k and input size is n .

Tractable:

Set of all problems that can be **solved in polynomial** time are called Tractable problem.

Intractable:

Set of all problems that **can't be solved** in polynomial time are called Intractable problem.

Optimization & Decision Problems

- **Decision problems**
 - Given an input and a question regarding a problem, determine if the answer is **yes** or **no**.
- **Optimization problems**
 - Find a solution with the “**best**” value.
- Optimization problems can be cast as decision problems that are easier to study
 - *E.g.:* Shortest path: G = unweighted directed graph
 - Find a path between u and v that uses the fewest edges

Class of “P” Problems

- **Class P** consists of (decision) problems that are solvable in polynomial time
- Polynomial-time algorithms
 - Worst-case running time is $O(n^k)$, for some constant k
(where n is the size of the input to the problem)
- Examples of polynomial time:
 - $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \log n)$
- Examples of non-polynomial time:
 - $O(2^n)$, $O(n^n)$, $O(n!)$

Tractable/Intractable Problems

- Problems in P are also called **tractable**
- Problems **not** in P are **intractable or unsolvable**
 - Can be solved in reasonable time only for small inputs
 - Or, can not be solved at all.

Example of Unsolvable Problem

- Turing discovered in the 1930's that there are problems **unsolvable** by *any* algorithm. The most famous of them is the ***halting problem***
 - Given an arbitrary algorithm and its input, will that algorithm eventually halt, or will it continue forever in an “***infinite loop***”

Intractable Problems

- Can be classified in various categories based on their degree of difficulty, e.g.,
 - NP
 - NP-complete
 - NP-hard
- Let's define NP algorithms and NP problems ...

Nondeterministic and NP Algorithms

NP is the set of all decision problems solvable by a **non-deterministic algorithm in polynomial time**.

Nondeterministic algorithm = two stage procedure:

1) Nondeterministic (“guessing”) stage:

generate randomly an arbitrary string that can be assumed of as a candidate solution (“certificate”)

2) Deterministic (“verification”) stage:

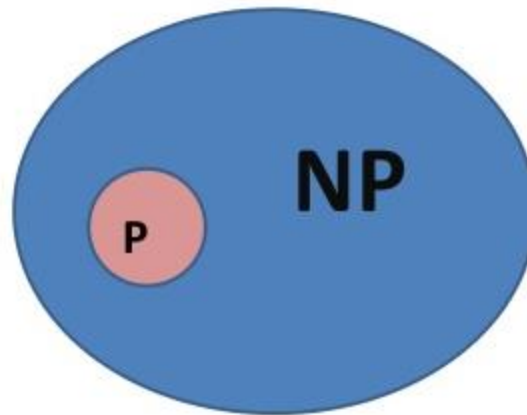
take the certificate and the instance to the problem and returns YES if the certificate represents a solution.

(Verification stage is polynomial)

- Warning: NP does **not** mean “non-polynomial”

Relation between P & NP

- NP is the set of all **decision problems** solvable by **nondeterministic algorithms** in **polynomial time**.
- *Deterministic algorithms is just a special case of non deterministic one..*



P=NP????

- The **P** versus **NP** problem is a major **unsolved problem** in computer science.
- Computer scientists do not know for certain whether **P = NP** or not.

TRAVELLING SALESMAN

Example of P Class

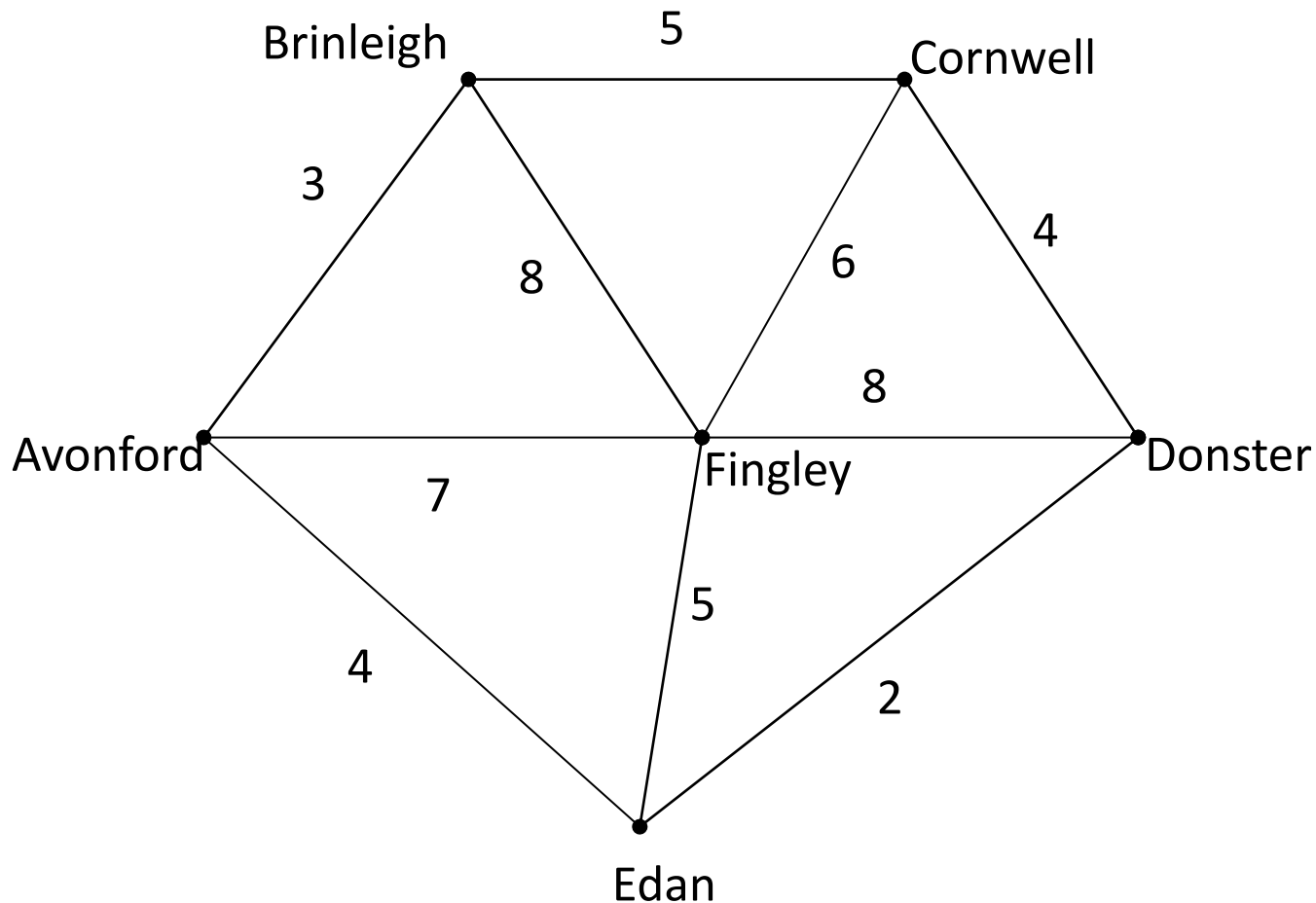
Kruskal's algorithm

1. Select the shortest edge in a network.
2. Select the next shortest edge which does **not create a cycle**.
3. Repeat step 2 until all vertices have been connected.

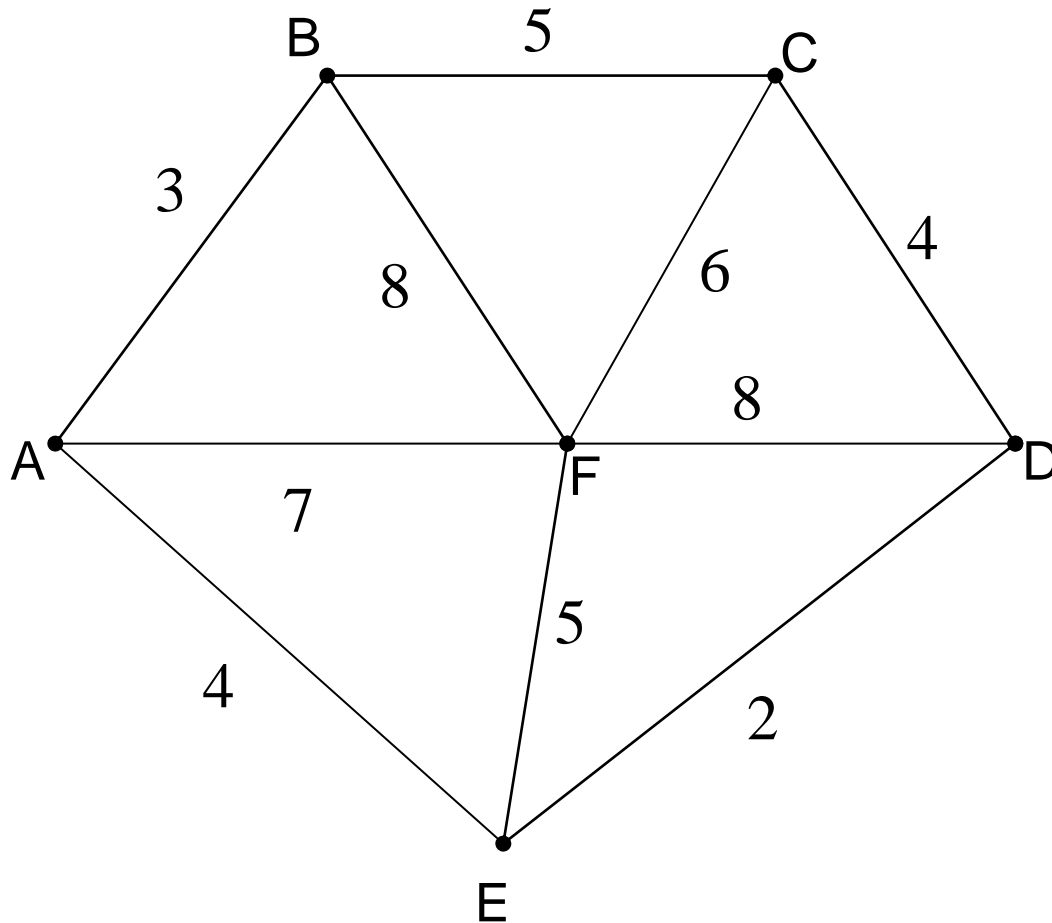
Complexity is $O(e \log e)$ where e is the number of edges

Example

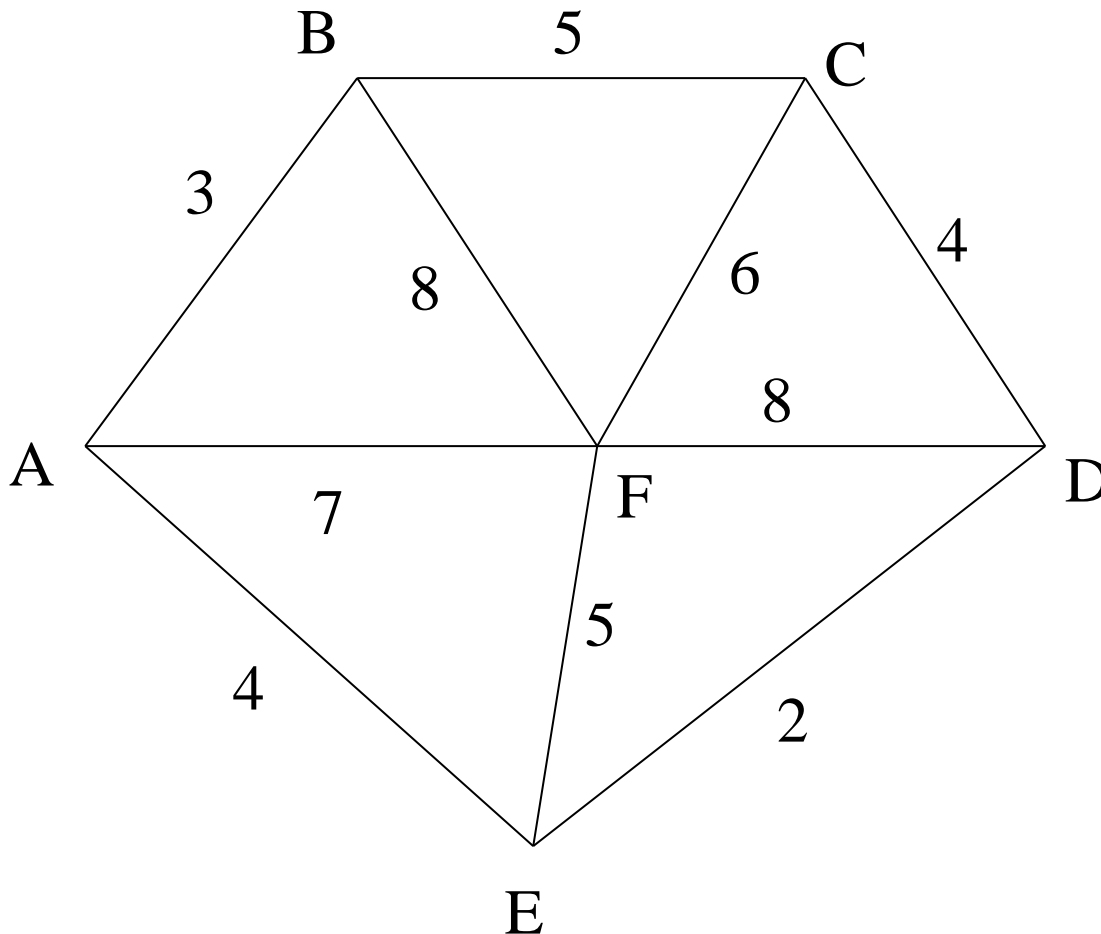
A cable company want to connect five villages to their network which currently extends to the market town of Avonford. What is the minimum length of cable needed?



We model the situation as a network, then the problem is to find the minimum connector for the network



Kruskal's Algorithm

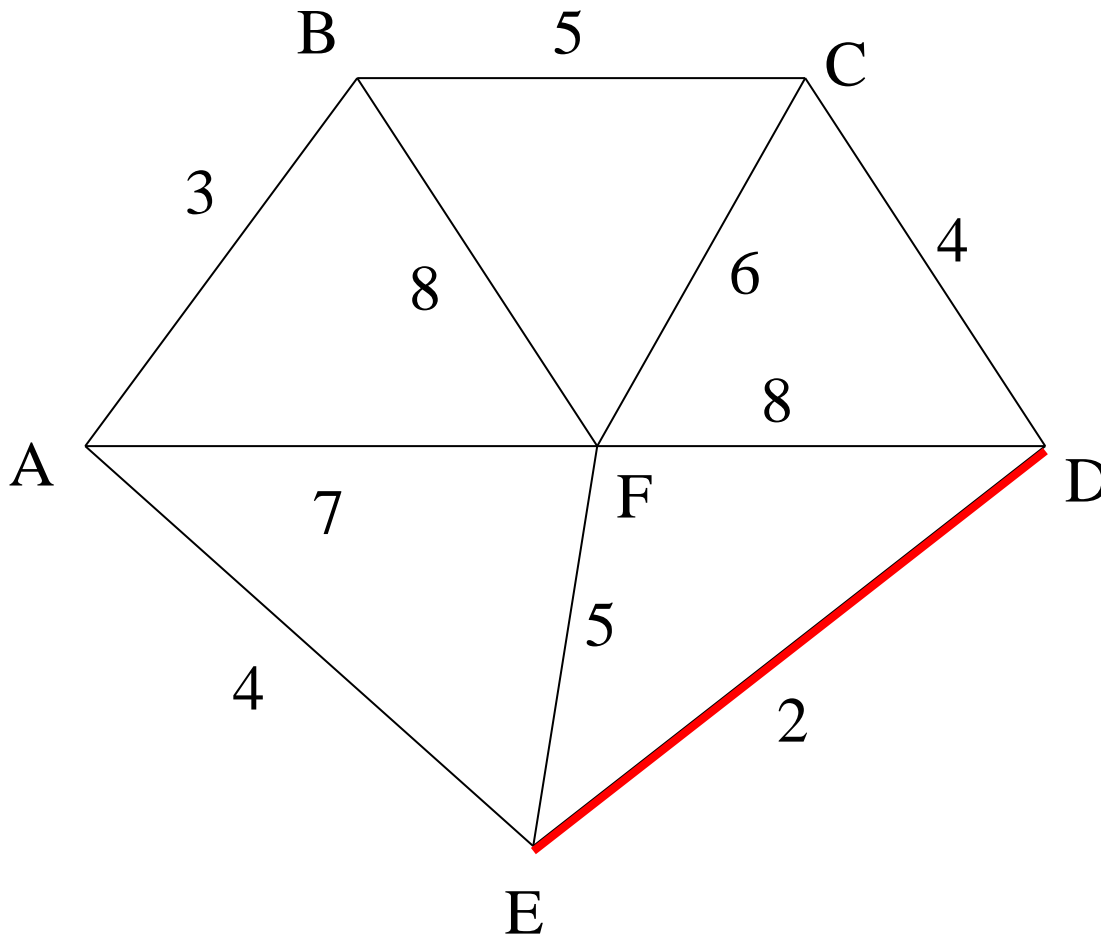


List the edges in
order of size:

ED 2
AB 3
AE 4
CD 4
BC 5
EF 5
CF 6
AF 7
BF 8
CF 8

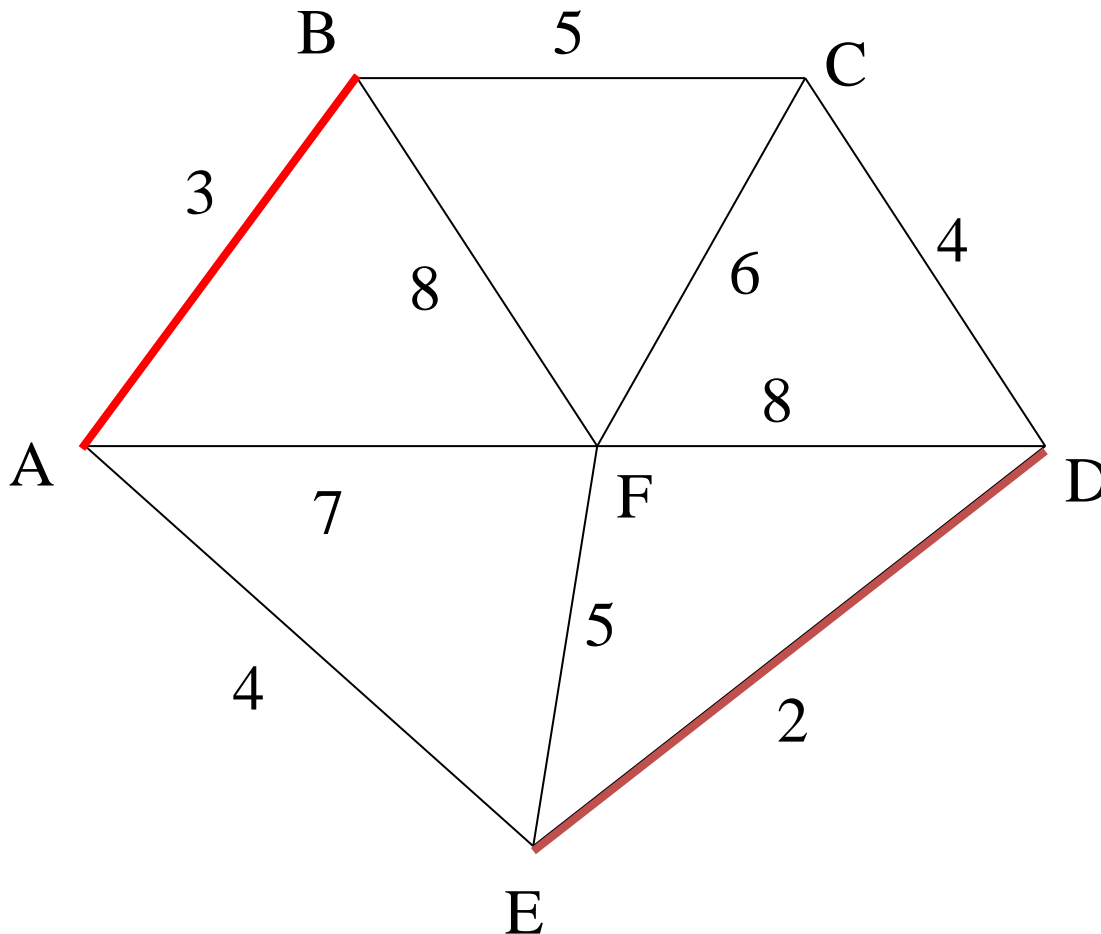
Kruskal's Algorithm

Select the shortest edge in the network



ED 2

Kruskal's Algorithm

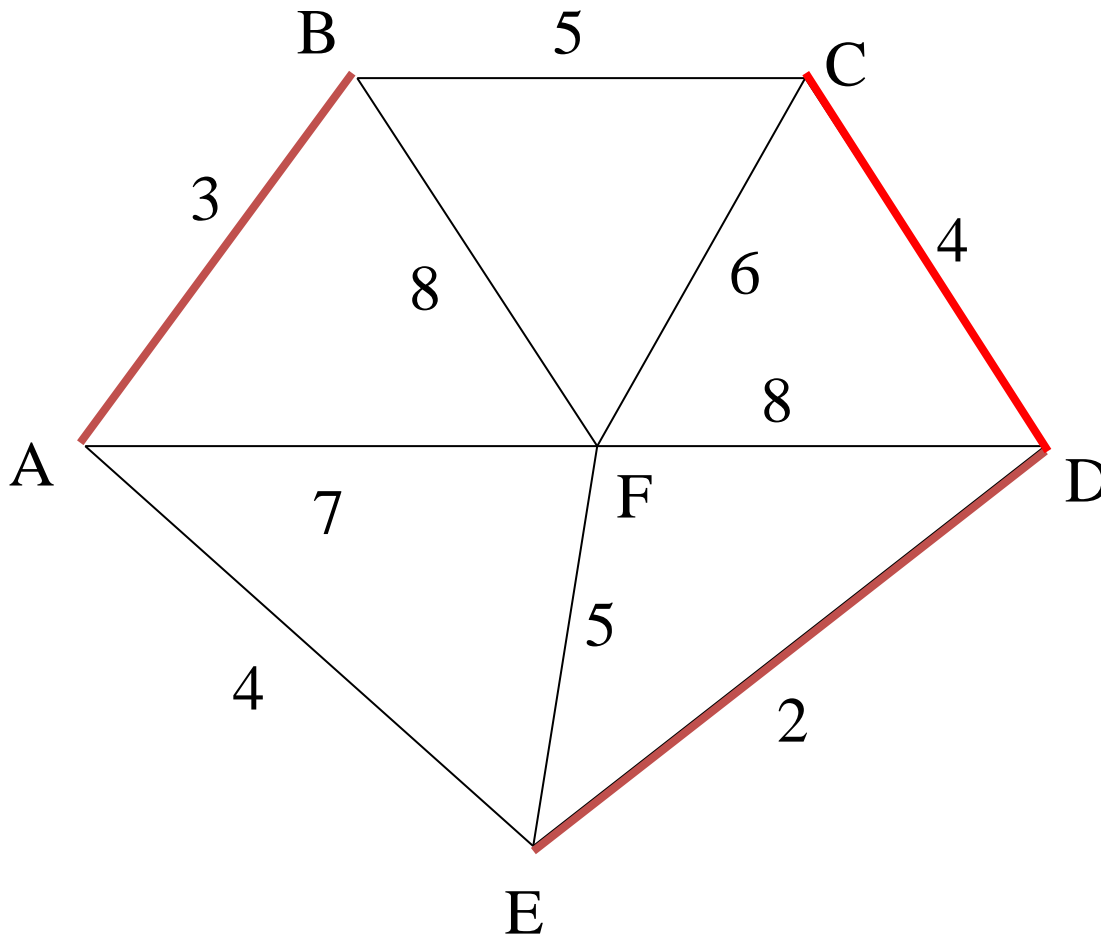


Select the next
shortest edge
which does not
create a cycle

ED 2

AB 3

Kruskal's Algorithm



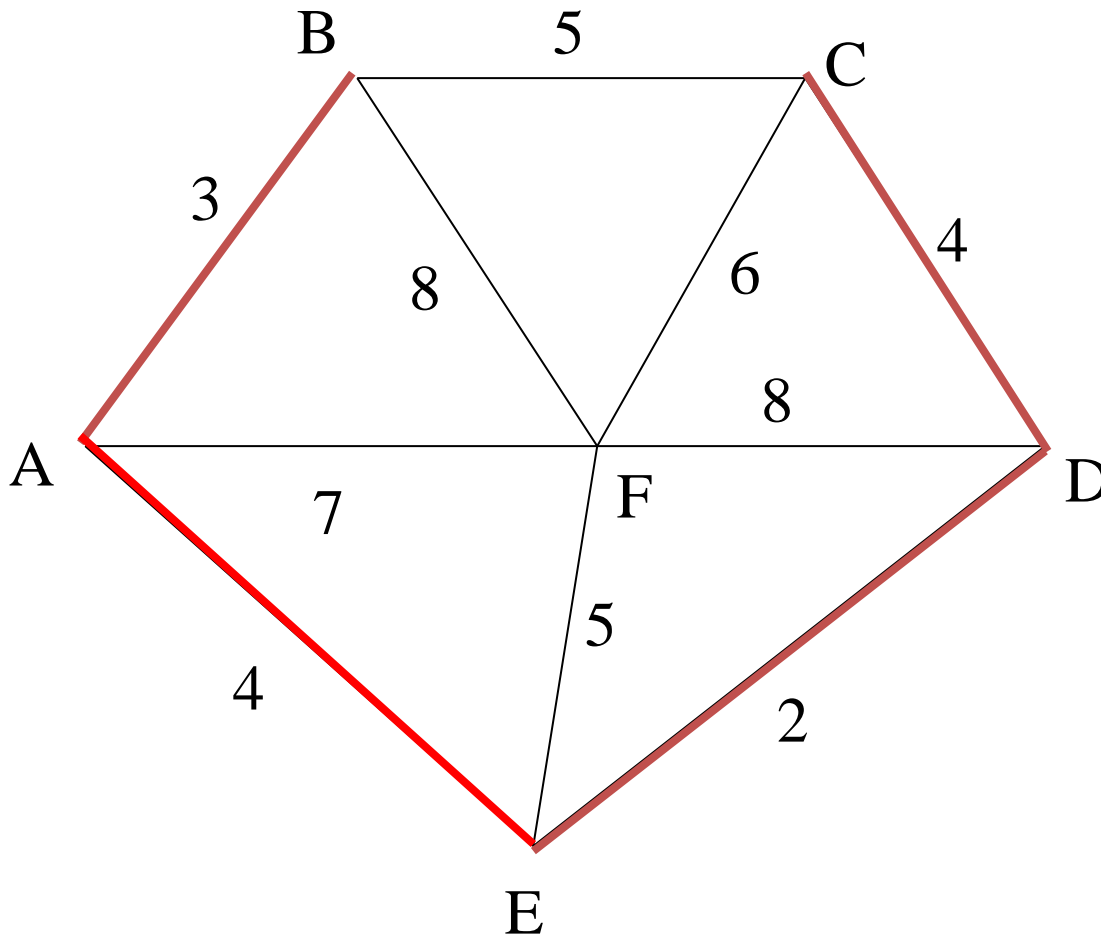
Select the next
shortest edge
which does not
create a cycle

ED 2

AB 3

CD 4 (or AE 4)

Kruskal's Algorithm



Select the next
shortest edge
which does not
create a cycle

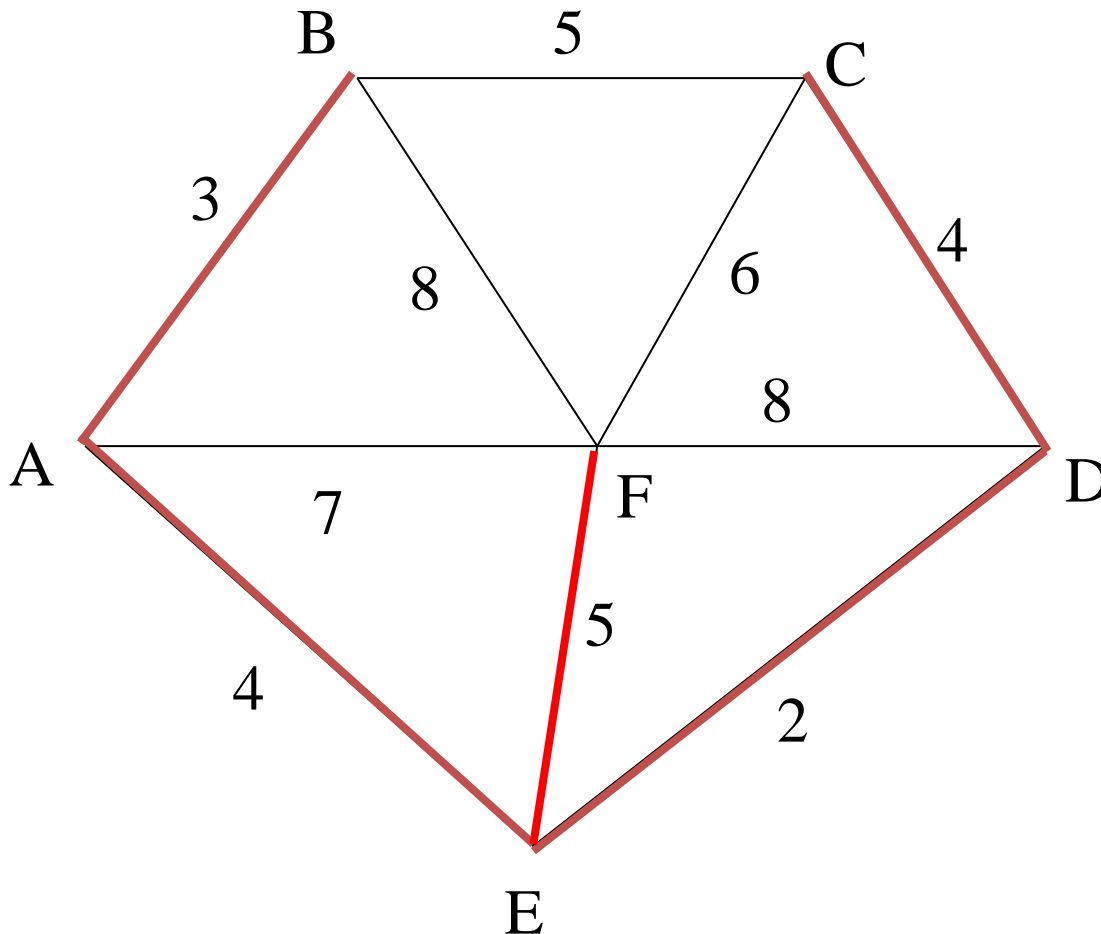
ED 2

AB 3

CD 4

AE 4

Kruskal's Algorithm



Select the next
shortest edge
which does not
create a cycle

ED 2

AB 3

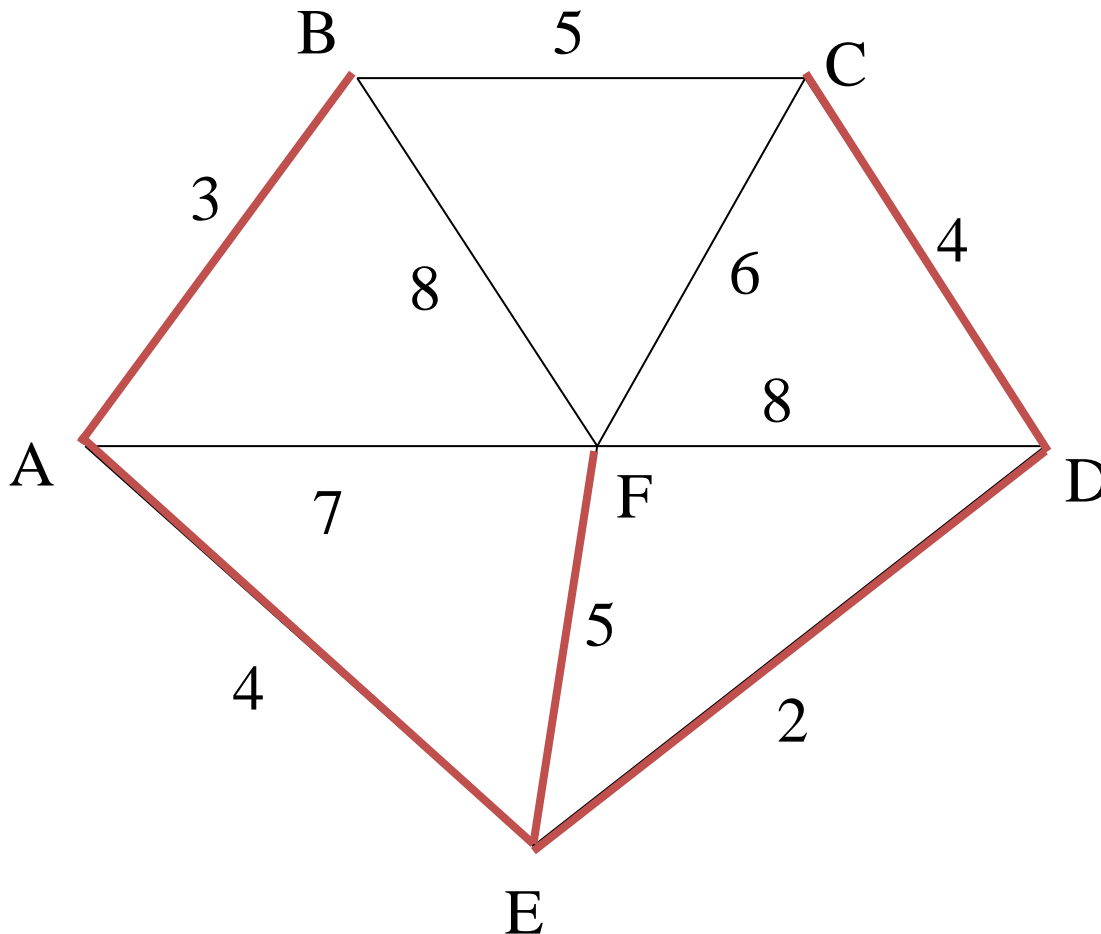
CD 4

AE 4

**BC 5 – forms a
cycle**

EF 5

Kruskal's Algorithm



All vertices have
been
connected.

The solution is

ED 2

AB 3

CD 4

AE 4

EF 5

Total weight of
tree: 18

Example of NP Class

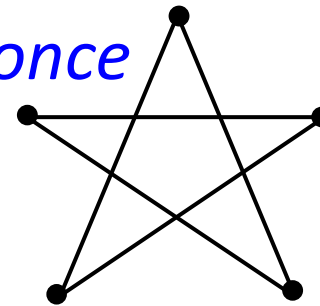
The **Travelling Salesman Problem** describes a salesman who must travel between N cities. The order in which he does so is something he does not care about, as long as he visits each one during his trip, and finishes where he was at first.

E.g.: Hamiltonian Cycle

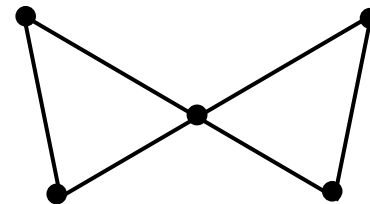
- **Given:** a directed graph $G = (V, E)$, determine a **simple cycle** that contains each vertex in V
 - *Each vertex can only be visited once*

- **Certificate:**

- Sequence: $\langle v_1, v_2, v_3, \dots, v_{|V|} \rangle$



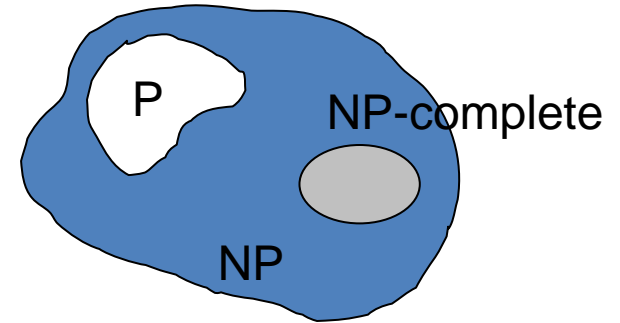
hamiltonian



not
hamiltonian

NP-Completeness (informally)

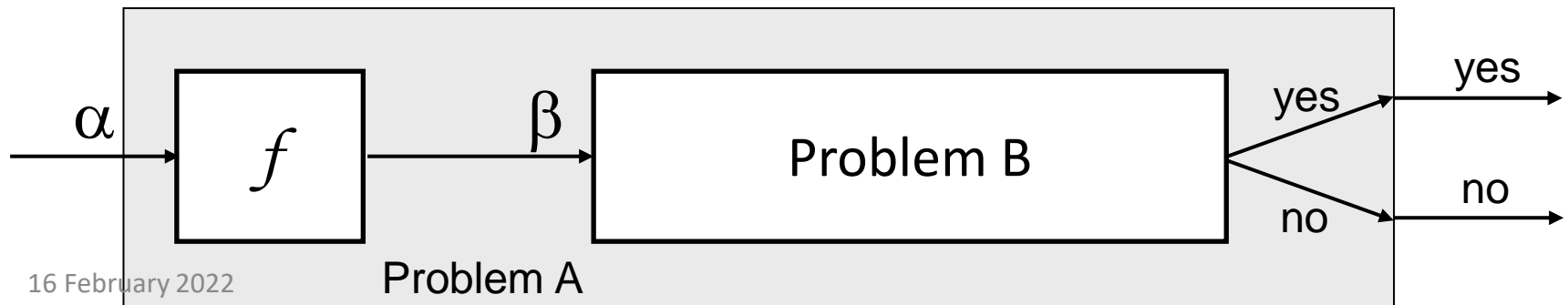
- **NP-complete** problems are defined as the hardest problems in NP



- Most practical problems turn out to be either P or NP-complete.
- Study NP-complete problems ...

Reductions

- Reduction is a way of saying that one problem is “**easier**” than another.
- We say that problem A is easier than problem B, (i.e., we write “ **$A \leq B$** ”) if we can solve A using the algorithm that solves B.
- **Idea:** transform the inputs of A to inputs of B



Polynomial Reductions

- Given two problems A , B , we say that A is polynomially **reducible** to B ($A \leq_p B$) if:
 1. There exists a function f that converts the input of A to inputs of B in **polynomial time**
 2. $A(i) = \text{YES} \Leftrightarrow B(f(i)) = \text{YES}$

NP Complete & Hard

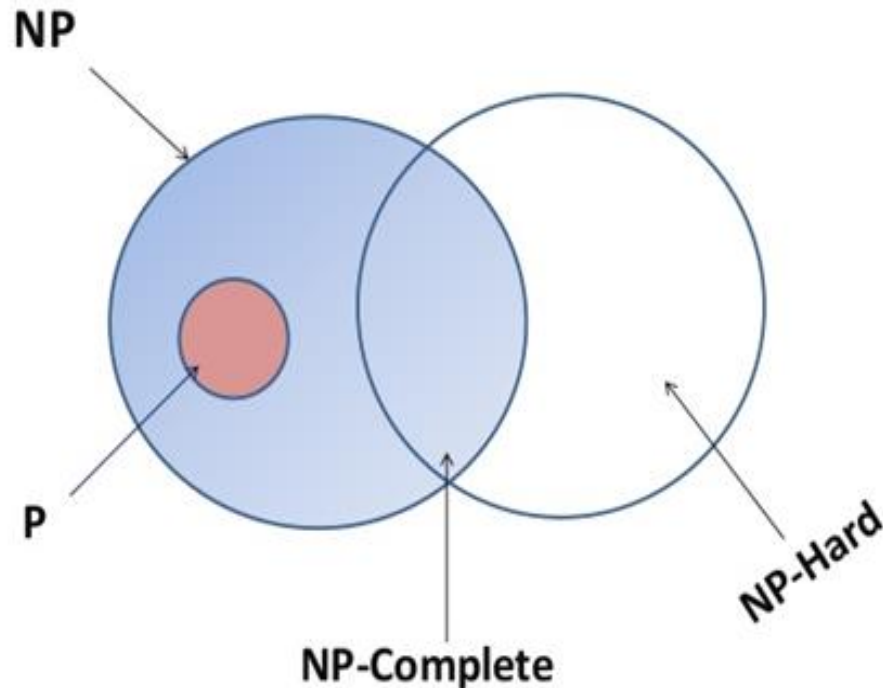
- A problem B is **NP-complete** if:

$$(1) B \in \mathbf{NP}$$

$$(2) A \leq_p B \text{ for all } A \in \mathbf{NP}$$

- If B satisfies only property (2) we say that B is **NP-hard**

Relation between P, NP, NP-Complete, NP-Hard



NP-Complete & NP-Hard Problems

- **NP-Complete Problems**

Following are some NP-Complete problems.

- I. Determining whether a graph has a **Hamiltonian cycle**.
- II. Determining whether a **Boolean formula is satisfiable** etc.

- **NP-Hard Problems**

The following problems are NP-Hard

- I. The circuit-satisfiability problem
- II. Set Cover
- III. Vertex Cover

TSP is NP-Complete

First we have to prove that ***TSP belongs to NP***. In TSP, we find a tour and check that the tour contains **each vertex once**. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is **minimum**. This can be completed in polynomial time. Thus ***TSP belongs to NP***.

TSP reduce into Hamiltonian Cycle where **simple cycle** that contains ***each vertex can only be visited once***.

Satisfiability Problem (SAT)

- **Satisfiability problem:** given a logical expression Φ , find an assignment of values (F, T) to variables x_i that causes Φ to evaluate to T

$$\Phi = x_1 \vee \neg x_2 \wedge x_3 \vee \neg x_4$$

- SAT was the first problem shown to be NP-complete!
- [Video](#)

Normal Forms for Boolean Expression

Essential definitions

- ✓ A **literal**: is either a variable, or a negated
e.g. $\neg x$, x
- ✓ A **clause**: is the **logical OR** of one or more literals.
e.g. x , $x \vee \neg y \vee z$.
- ✓ A **boolean expression**: is said to be in conjunctive normal form or CNF, if it is the AND of clauses.

Important notation

- ✓ **OR** (\vee) is treated as a Sum, using the (+) operator.
- ✓ **AND** (\wedge) is treated as a product.
 - ✓ normally use juxtaposition (no operator).
- ✓ **Not** = (\neg)

Example

The expression $(x \vee \neg y) \wedge (\neg x \vee z)$:

will be written in CNF as $(x + \neg y)(\neg x + z)$

It is in **Conjunctive Normal Form(CNF)**, since it is **the AND (product) of the clauses:**

$(x + \neg y)$ and $(\neg x + z)$.

Question?

are the expressions

$(x + y\neg z)(x + y + z)(\neg y + \neg z)$, xyz in CNF?

✓ The first expression not in CNF.

It is the AND of three subexpressions, the last two are clauses, but the first is not, it is the sum of a literal and a product of two literals.

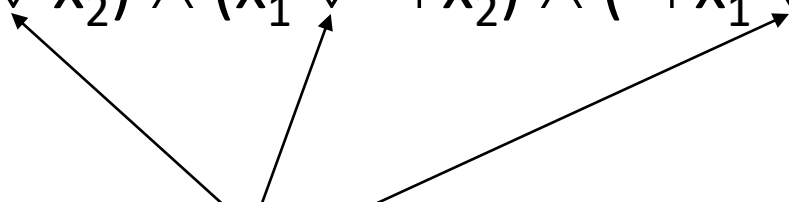
✓ The second expression in CNF.

the clause can have only one literal, so it is the product of three clauses, (x) , (y) , and (z)

CNF Satisfiability

- CNF is a special case of SAT.
- Φ is in “Conjunctive Normal Form” (CNF)
 - “AND” of expressions (i.e., clauses)
 - Each clause contains only “OR”s of the variables and their complements

E.g.: $\Phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$



clauses

DeMorgan's Laws

I. $\neg(E \wedge F) \Rightarrow \neg(E) \vee \neg(F)$

II. $\neg(E \vee F) \Rightarrow \neg(E) \wedge \neg(F)$

III. $\neg(\neg E) \Rightarrow E$: the double negation.

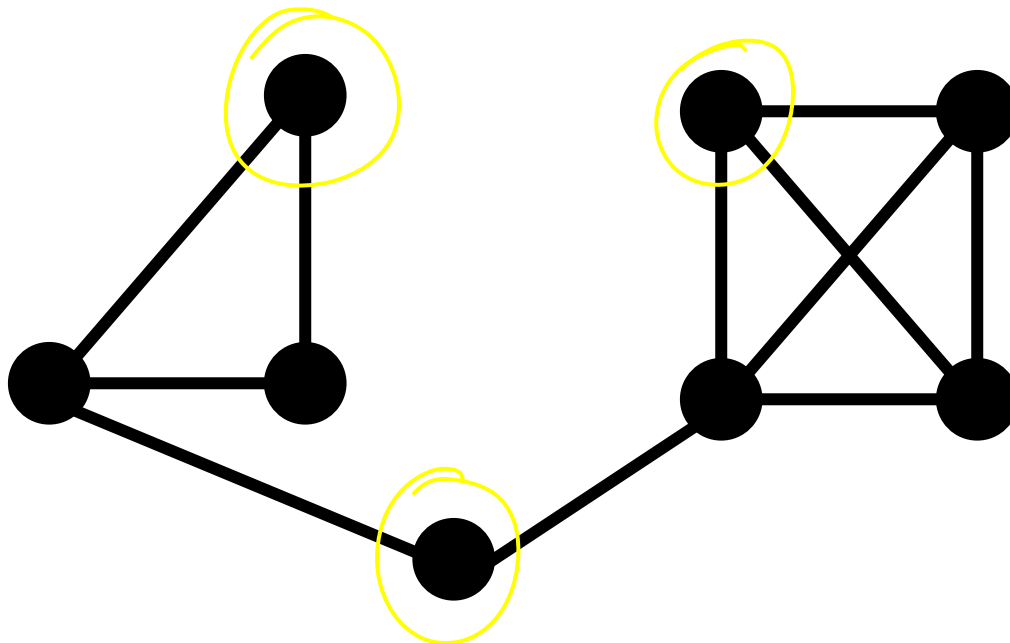
Example

$$E = \neg((\neg(x + y))(\neg x + y))$$

$\neg((\neg(x + y))(\neg x + \neg y))$	Start
$\neg(\neg(x + y)) + \neg(\neg x + \neg y)$	(1)
$x + y + \neg(\neg x + \neg y)$	(3)
$x + y + (\neg(\neg x))(\neg(\neg y))$	(2)
$x + y + x + y$	(3)

Independent Set

An independent set is a set of nodes with no edges between them



This graph
contains an
independent
set of size 3

3-CNF Satisfiability

A subcase of CNF problem:

- Contains three clauses

- *E.g.:*

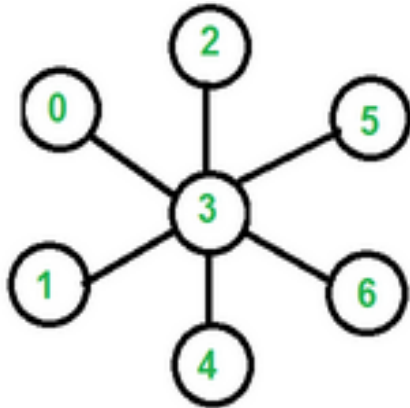
$$\Phi = (x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

- **3-CNF** is NP-Complete

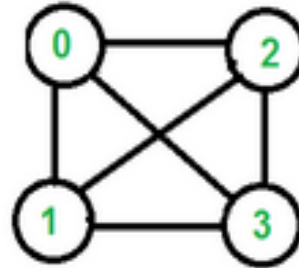
Vertex/Node Cover Problem

- A vertex cover of an undirected graph is a **subset of its vertices** such that for every edge (u, v) of the graph, either 'u' or 'v' is in vertex cover. Although the name is Vertex Cover, the **set covers all edges of the given graph.**

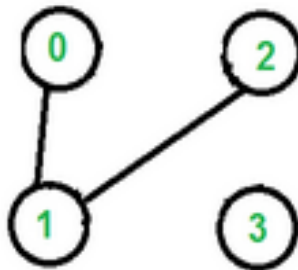
Example:



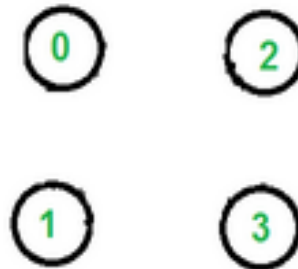
Minimum Vertex Cover is {3}



Minimum Vertex Cover is {0, 1, 2} or {0, 1, 3} or {1, 2, 3}



Minimum Vertex Cover is {1}



Minimum Vertex Cover is empty {}

