



Introduction to Automata Theory

Reading: Chapter 1



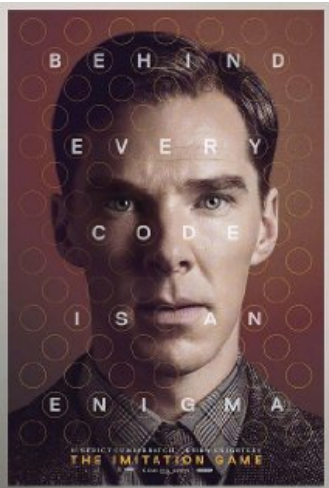
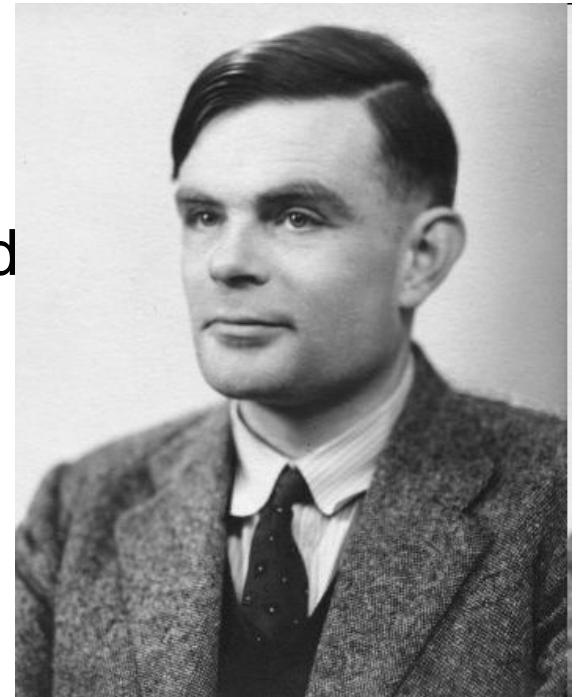
What is Automata Theory?

- *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.

$S \longrightarrow 0A$

$B \longrightarrow 1B$

$A \longrightarrow 1A$

$B \longrightarrow 0F$

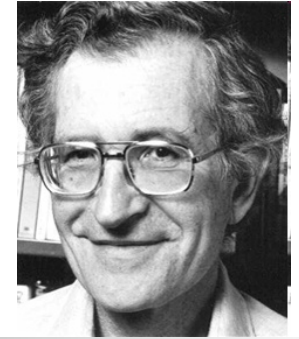
$A \longrightarrow 0B$

$F \longrightarrow \epsilon$

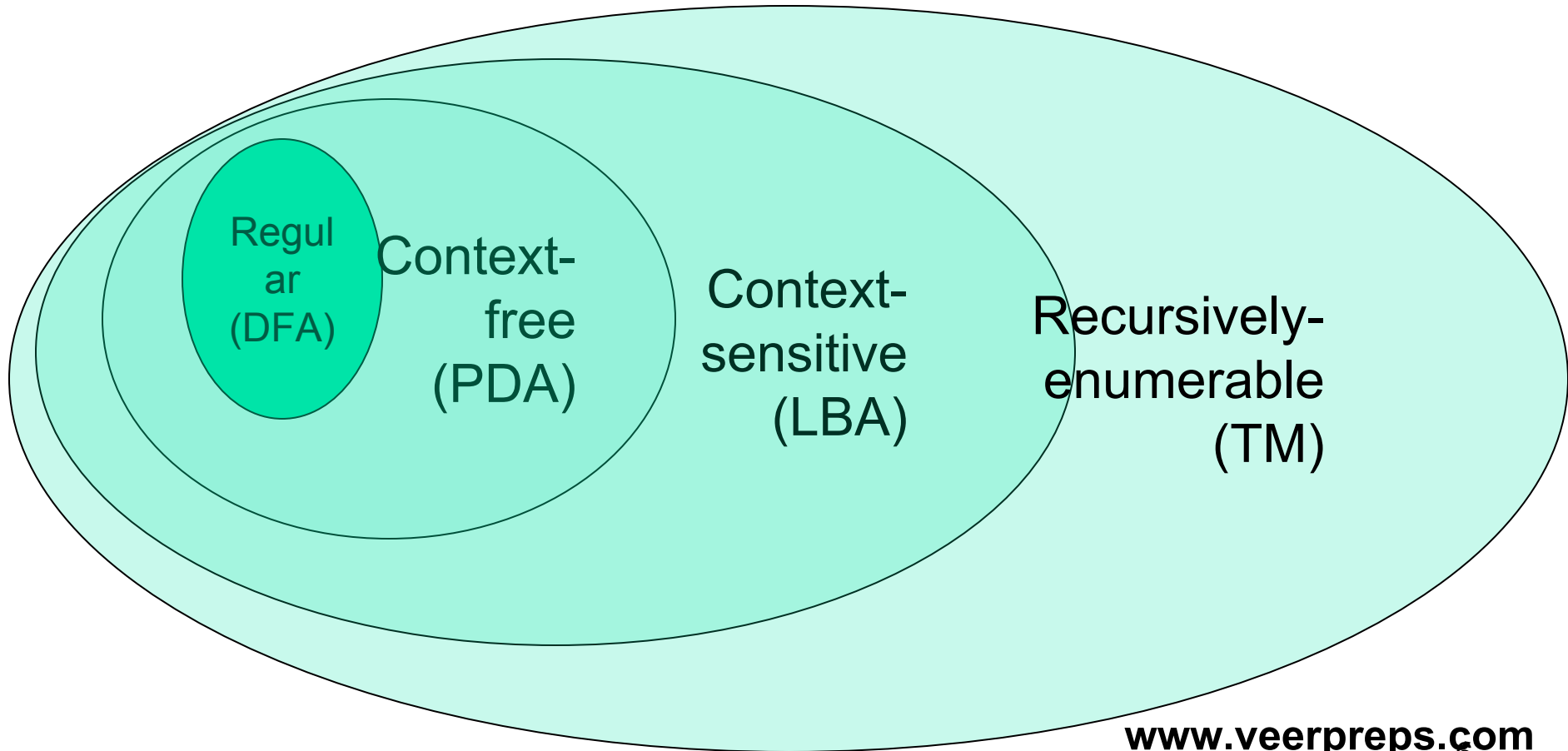
- Languages: “A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols”
- Grammars: “A grammar can be regarded as a device that enumerates the sentences of a language” - nothing more, nothing less
- N. Chomsky, *Information and Control*, Vol 2, 1959



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: $\Sigma = \{a,b,c,..z\}$
 - 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:
 $L = \{\epsilon, 01, 0011, 000111, \dots\}$
2. Let L be *the* language of all strings of with equal number of 0's and 1's:
 $L = \{\epsilon, 01, 10, 0011, 1100, 0101, 1010, 1001, \dots\}$

→
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?

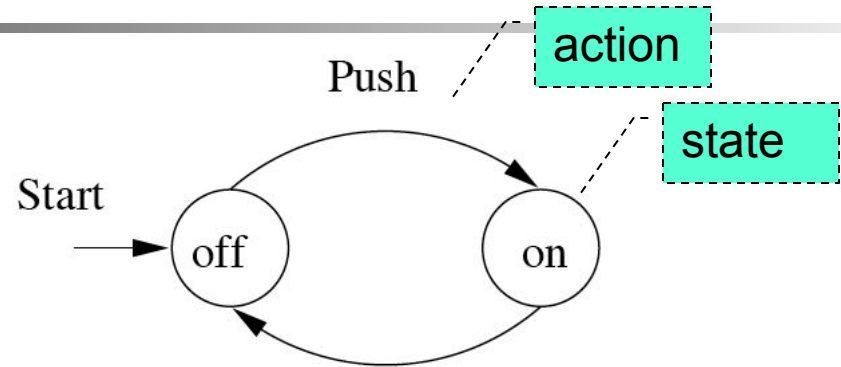


Finite Automata

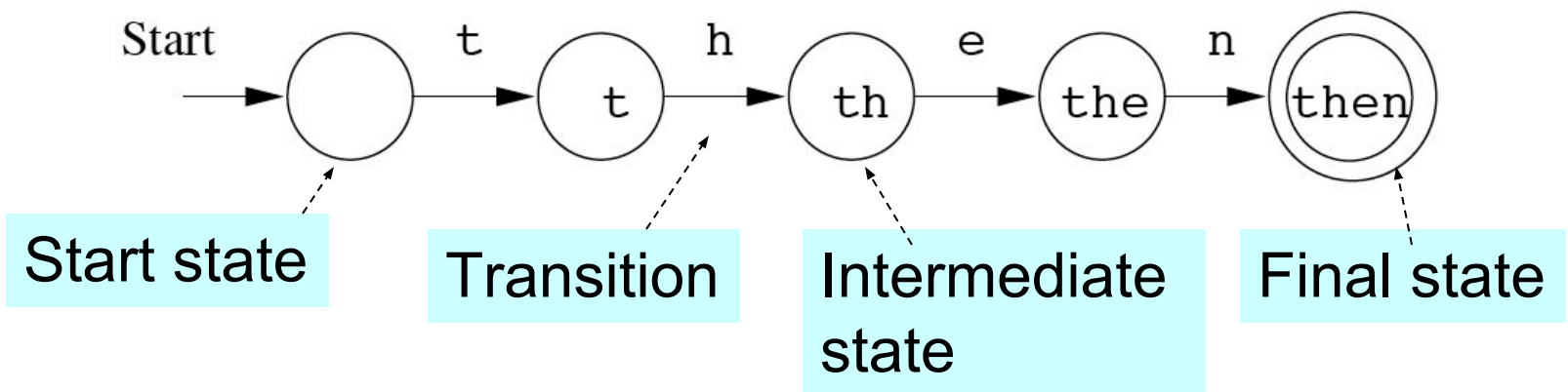
- Some Applications
 - Software for designing and checking the behavior of digital circuits
 - Lexical analyzer of a typical compiler
 - Software for scanning large bodies of text (e.g., web pages) for pattern finding
 - Software for verifying systems of all types that have a finite number of states (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) $\square a^2 \geq 1, b^2 \geq 1, c^2 \geq 1, d^2 \geq 1$ (by 2)

4) $\square x \geq 4$ (by 1 & 3)

5) $\square 2^x \geq x^2$ (by 4 and Claim 1)

“implies” or “follows”

On Theorems, Lemmas and Corollaries

We typically refer to:

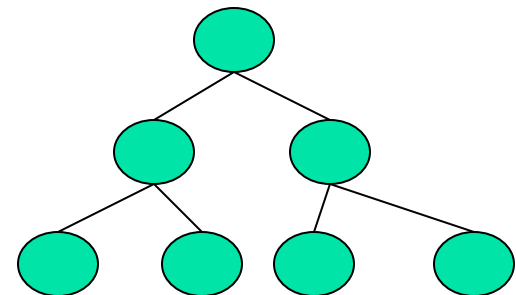
- A major result as a “**theorem**”
- An intermediate result that we show to prove a larger result as a “**lemma**”
- A result that follows from an already proven result as a “**corollary**”

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 $B \quad \equiv \quad \text{If } \sim B \text{ then } \sim A$



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



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:
 - $Q \Rightarrow$ a finite set of states
 - $\Sigma \Rightarrow$ a finite set of input symbols (alphabet)
 - $q_0 \Rightarrow$ a start state
 - $F \Rightarrow$ set of accepting states
 - $\delta \Rightarrow$ a transition function, which is a mapping between $Q \times \Sigma \Rightarrow Q$
- 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 .

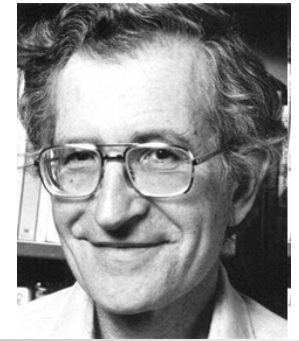


Regular Languages

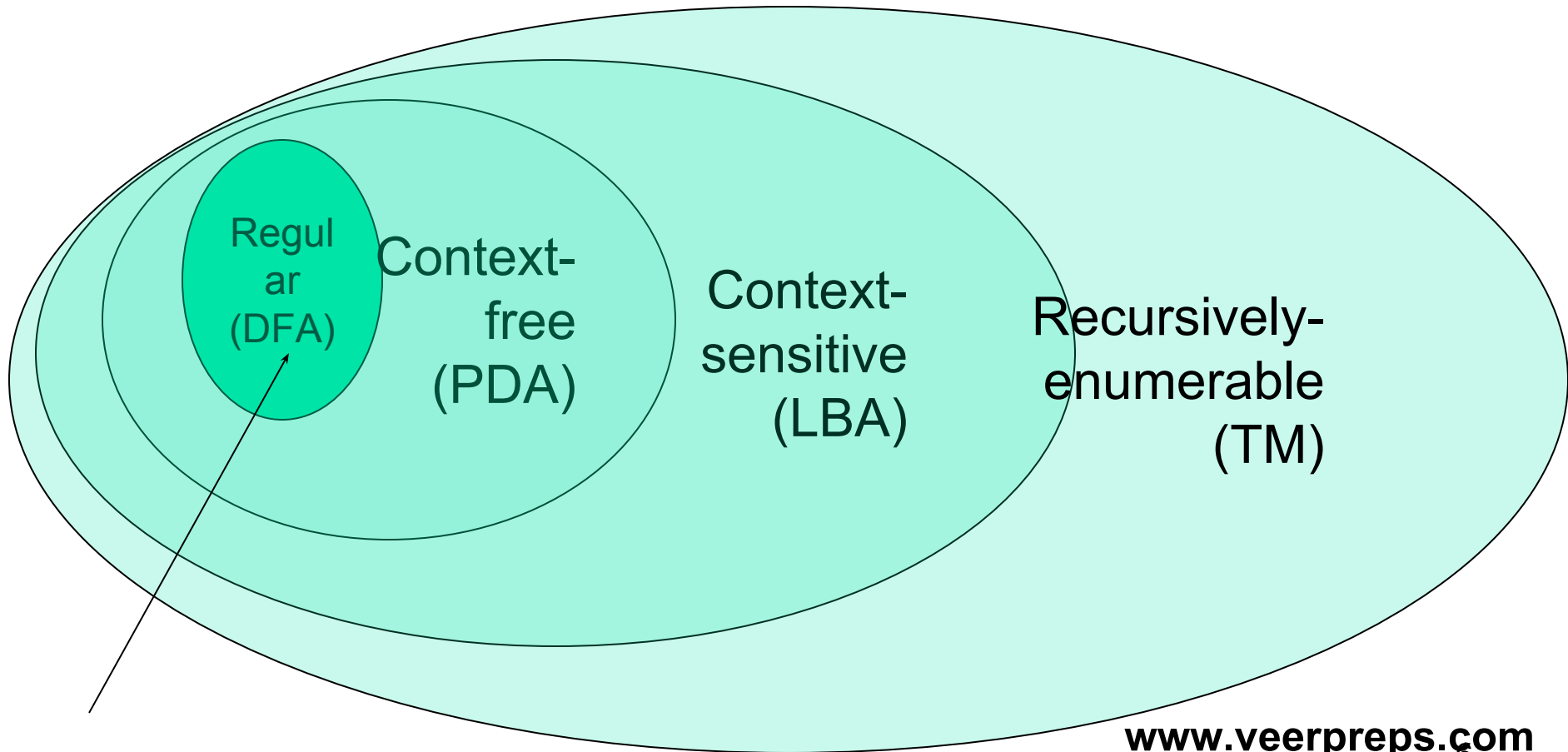
- Let $L(A)$ be a language *recognized* by a DFA A .
 - Then $L(A)$ is called a “*Regular Language*”.
- Locate regular languages in the Chomsky Hierarchy



The Chomsky Hierarchy



- A containment hierarchy of classes of formal languages





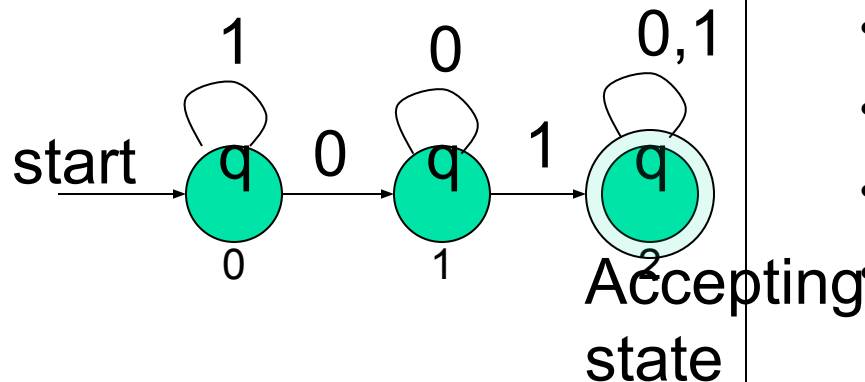
Example #1

- 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?



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

Transition table

		symbols	
		0	1
states	q_0	q_1	q_0
	q_1	q_1	q_2
	q_2	q_2	q_2

- What if the language allows empty strings?



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 $w=10010$, $a=1$:
 - $\hat{\delta}(q_0, wa) = ?$



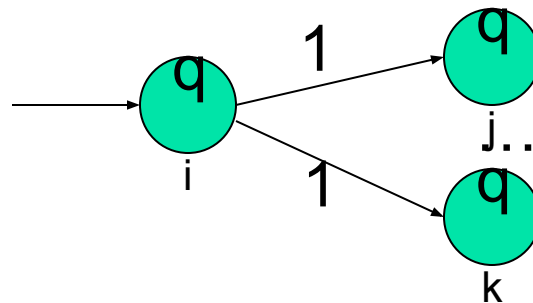
Language of a DFA

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., $L(A) = \{ w \mid \hat{\delta}(q_0, w) \in F \}$*
- *L.e., $L(A) =$ 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:
 - $Q \Rightarrow$ a finite set of states
 - $\Sigma \Rightarrow$ a finite set of input symbols (alphabet)
 - $q_0 \Rightarrow$ a start state
 - $F \Rightarrow$ set of accepting states
 - $\delta \Rightarrow$ a transition function, which is a mapping between $Q \times \Sigma \Rightarrow$ subset of Q
- An NFA is also defined by the 5-tuple:
 - $\{Q, \Sigma, q_0, F, \delta\}$



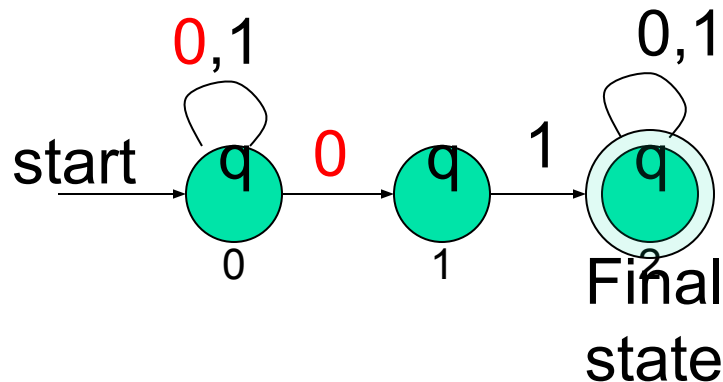
How to use an NFA?

- 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?

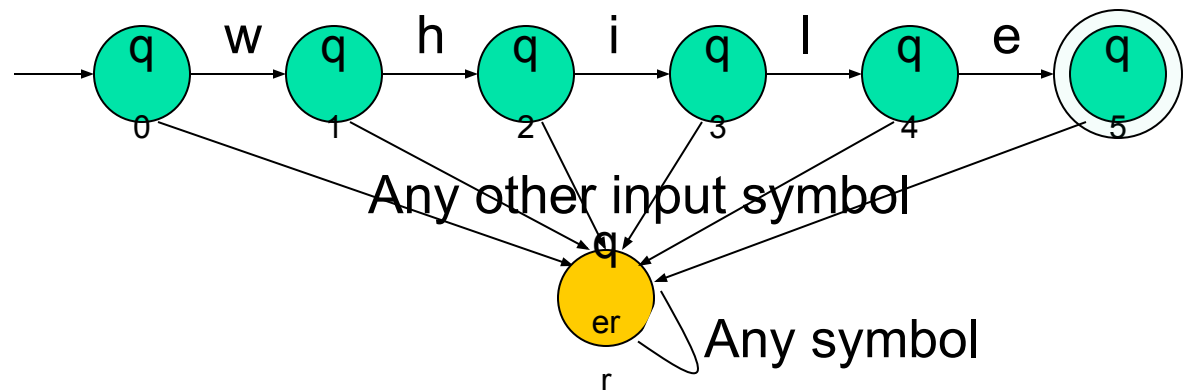
- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- start state = q_0
- $F = \{q_2\}$
- Transition table

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

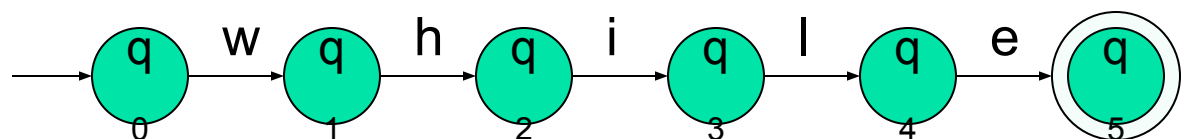
Note: Omitting to explicitly show error states is just a matter of design convenience (one that is generally followed for NFAs), and i.e., this feature should not be confused with the notion of non-determinism.

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. www.veerpreps.com



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 δ to NFA Paths

- Basis: $\hat{\delta}(q, \epsilon) = \{q\}$
- 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” \square a single “DFA_state”

Subset construction



NFA to DFA by subset construction

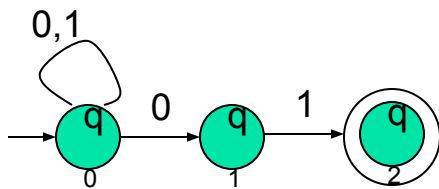
- Let $N = \{Q_N, \Sigma, \delta_N, q_0, F_N\}$
- Goal: Build $D = \{Q_D, \Sigma, \delta_D, \{q_0\}, F_D\}$ s.t.
 $L(D) = L(N)$
- Construction:
 1. Q_D = all subsets of Q_N (i.e., power set)
 2. F_D = set of subsets S of Q_N s.t. $S \cap F_N \neq \emptyset$
 3. δ_D : for each subset S of Q_N and for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

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

NFA to DFA construction: Example

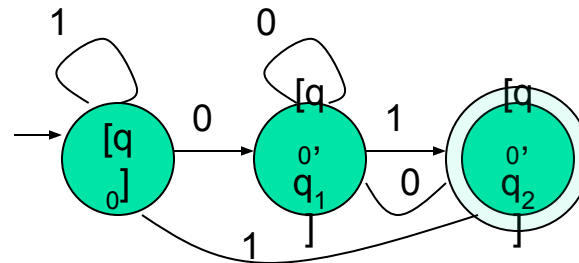
- $L = \{w \mid w \text{ ends in } 01\}$

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

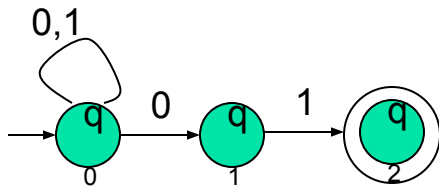
δ_D	0	1
$\rightarrow [q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_2]$
$*[q_0, q_2]$	$[q_0, q_1]$	$[q_0]$

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

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

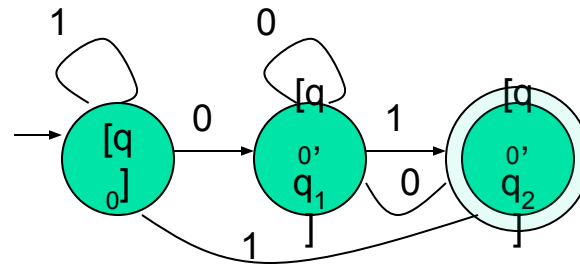
- $L = \{w \mid w \text{ ends in } 01\}$

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)

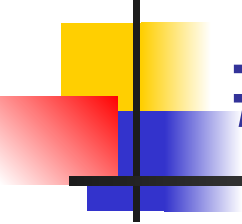


Correctness of subset construction

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 $w = xa$
 - $\hat{\delta}_D(\{q_0\}, xa) \equiv \delta_D(\hat{\delta}_N(q_0, x), a) \equiv \hat{\delta}_N(q_0, w)$



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)
- => 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

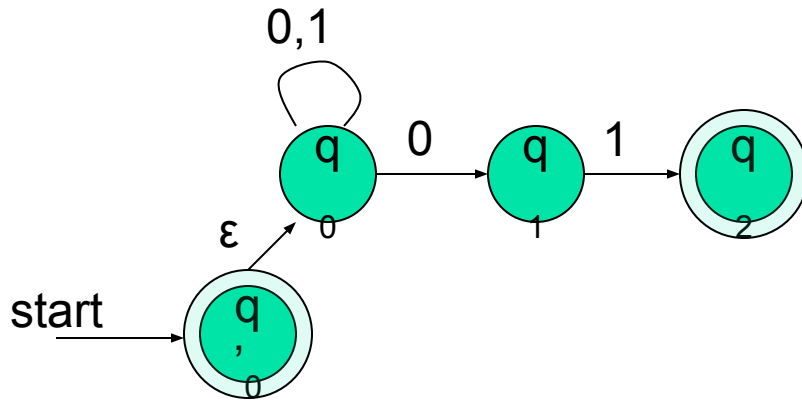
- We can allow explicit ϵ -transitions in finite automata
 - i.e., a transition from one state to another state without consuming any additional input symbol
 - Explicit ϵ -transitions between different states introduce non-determinism.
 - Makes it easier sometimes to construct NFAs

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
$*q'_0$	\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	$\{q_2\}$

ECLOSE(q'_0)

ECLOSE(q_0)

ECLOSE(q_1)

ECLOSE(q_2)

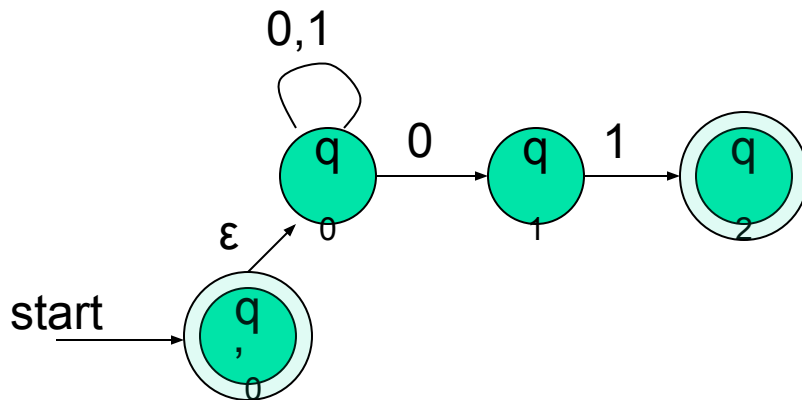
To simulate any transition:

Step 1) Go to all immediate destination states.

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

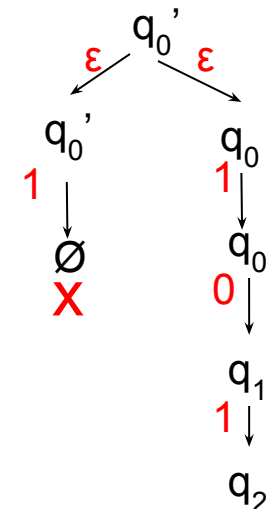
Example of an ϵ -NFA

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



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	$\{q_2\}$

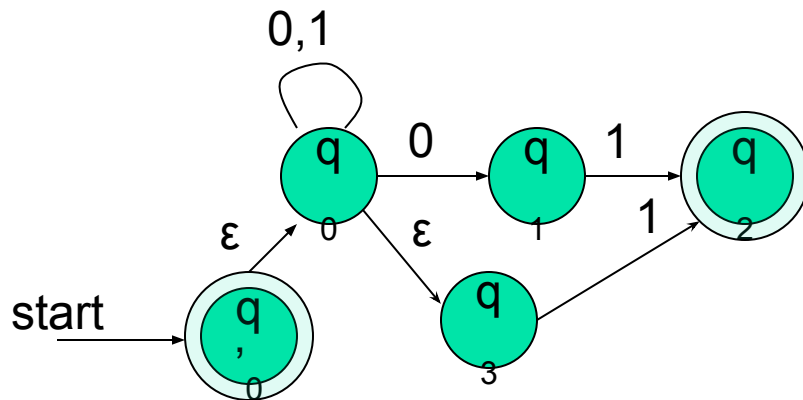


To simulate any transition:

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. $L(D) = L(E)$

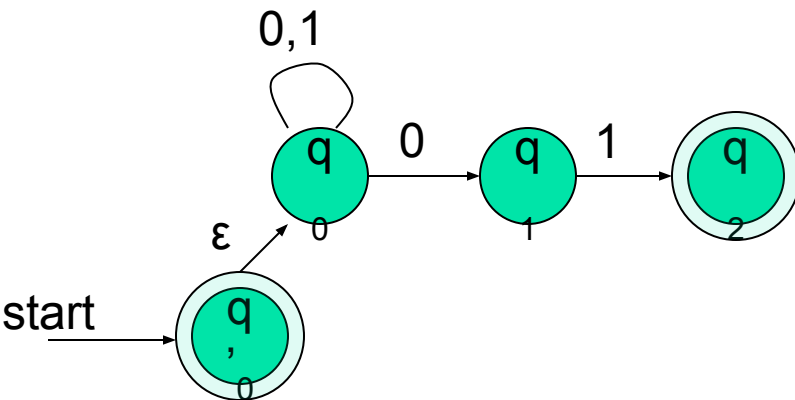
Construction:

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
 - $\delta_D(S, a) = \bigcup_{r \in R} \text{ECLOSE}(r)$ // from there, take a union of all their ϵ -closures

$r \in R$

Example: ϵ -NFA \square DFA

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

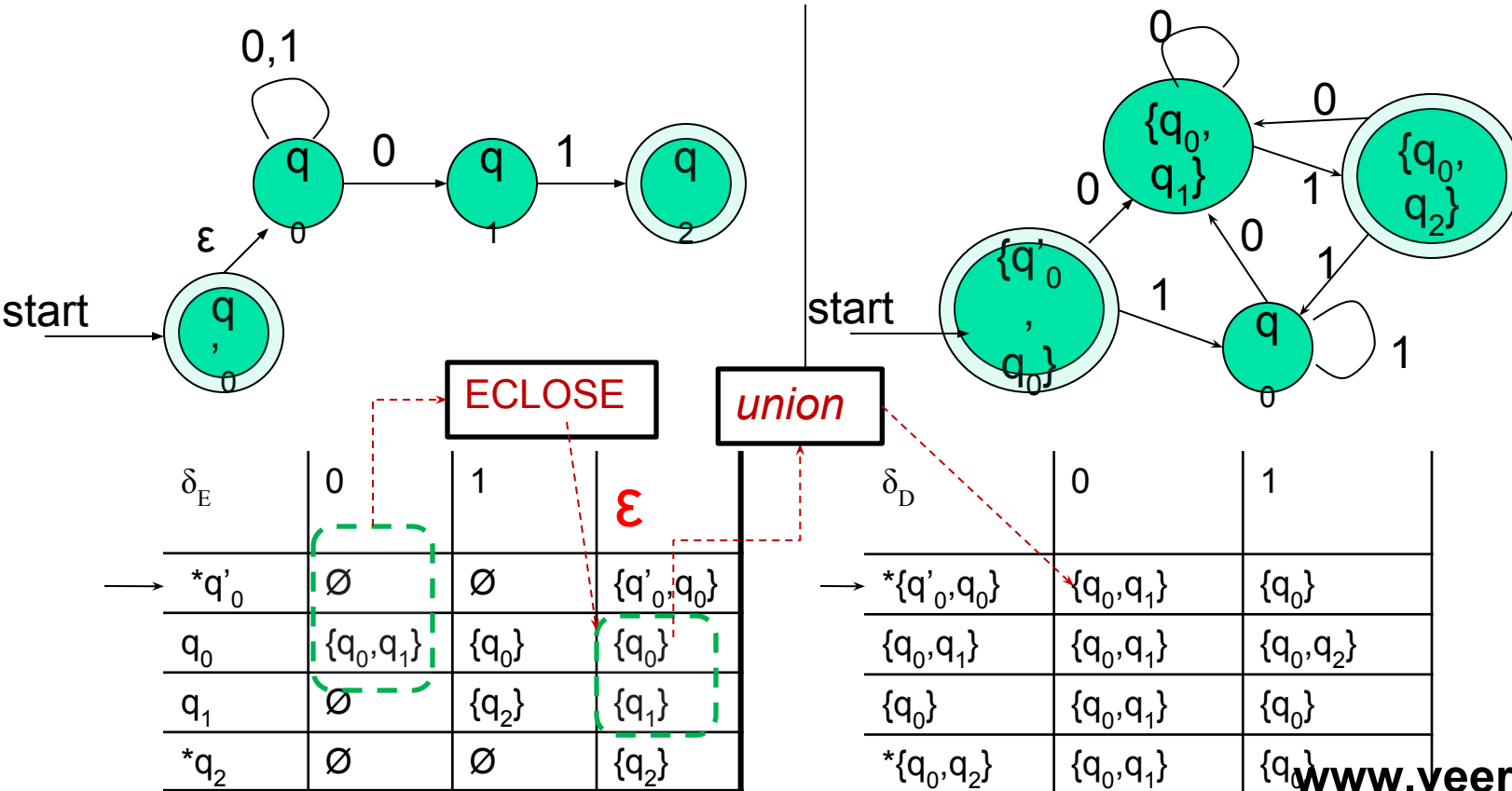


δ_E	0	1	ϵ
$\rightarrow *q'_0$	\emptyset	\emptyset	$\{q'_0, q_0\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$	$\{q_1\}$
$*q_2$	\emptyset	\emptyset	$\{q_2\}$

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

Example: ϵ -NFA \square 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



FA with Output string

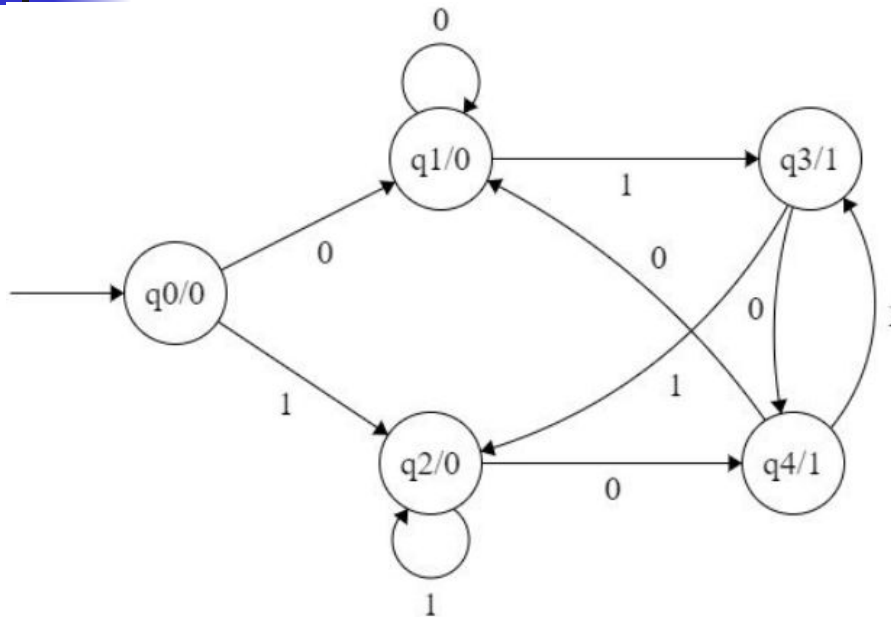
- Moore and Mealy Machines are Transducers that help in producing outputs based on the input of the current state or previous state.
- The output function λ and output alphabet Δ is added here
- In Moore Machine the output function is $Q \rightarrow \Delta$
- In Mealy Machine the output function is $Q \times \Sigma \rightarrow \Delta$



Moore Machines

- Moore Machines are finite state machines with output value and its output depends only on the present state.
- Formally it can be defined as a 6 tuples $M=(Q,q_0,\Sigma,\Delta,\delta,\lambda)$ where:
 - Q is a finite set of states.
 - q_0 is the initial state.
 - Σ is the input alphabet.
 - Δ is the output alphabet.
 - δ is the transition function which maps $Q \times \Sigma \rightarrow Q$.
 - λ is the output function which maps $Q \rightarrow \Delta$.

Moore Machines (Cont.)



- **Input:** 1,1
- **Transition:** $\delta(q0,1,1) \Rightarrow \delta(q2,1) \Rightarrow q2$
- **Output:**
000 (0 for q0, 0 for q2 and again 0 for q2)

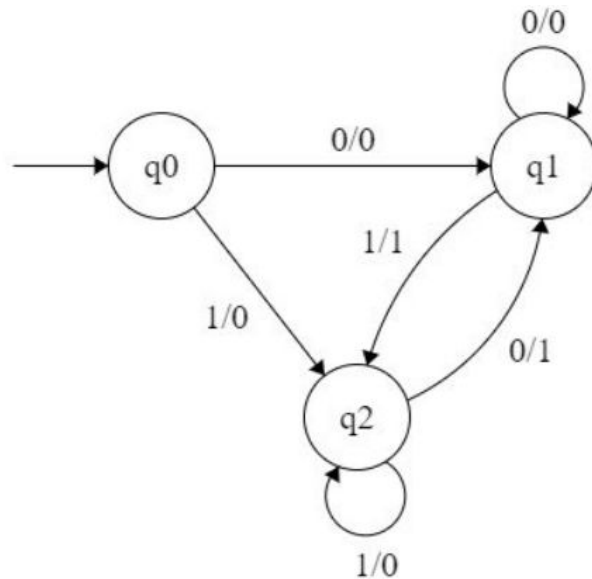
	Input =0	Input =1	
Present State	Next State	Next State	Output
q0	q1	q2	0
q1	q1	q3	0
q2	q4	q2	0
q3	q4	q2	1
q4	q1	q3	1



Mealy Machine

- Mealy Machines are finite state machines with output value and its output depends on the present state and current input symbol.
- Formally it can be defined as a 6 tuples $M=(Q,q_0,\Sigma,\Delta,\delta,\lambda)$ where:
 - Q is a finite set of states.
 - q_0 is the initial state.
 - Σ is the input alphabet.
 - Δ is the output alphabet.
 - δ is the transition function which maps $Q \times \Sigma \rightarrow Q$.
 - λ is the output function which maps $Q \times \Sigma \rightarrow \Delta$.

Mealy Machines (Cont.)



- **Input:** 1,1
- **Transition:** $\delta(q0,1,1) \Rightarrow \delta(q2,1) \Rightarrow q2$
- **Output:**
00 (q0 to q2 transition has Output 0 and q2 to q2 transition also has Output 0)

	Input=0		Input=1	
Present State	Next State	Output	Next State	Output
q0	q1	0	q2	0
q1	q1	0	q2	1
q2	q1	1	q2	0

Moore Machines vs Mealy Machines

Aspect	Moore Machines	Mealy Machines
Output	Outputs depend only on the current state.	Outputs depend on the current state and input.
Number of States	Tends to require more states due to separate output behavior.	Might require fewer states as outputs are tied to transitions.
Response Time	Slower response to input changes as outputs update on state changes.	Faster response to input changes due to immediate output updates.
Complexity	Can be simpler due to separation of output behavior.	Can be more complex due to combined state-input cases.
H/W requirement for ckt implementation	More	Less
Sync/Async	Synchronous	Asynchronous
Design	Easy	Difficult

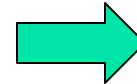


Conversion From Mealy to Moore Machine

- **Step 1.** First, find out those states which have more than 1 output associated with them. q_1 and q_2 are the states which have both output 0 and 1 associated with them.
- **Step 2** Create two states for these states. For q_1 , two states will be q_{10} (a state with output 0) and q_{11} (a state with output 1). Similarly, for q_2 , two states will be q_{20} and q_{21} .
- **Step 3:** Fill in the entries of the next state using the mealy machine transition table. For q_0 on input 0, the next state is q_{10} (q_1 with output 0). Similarly, for q_0 on input 1, the next state is q_{20} (q_2 with output 0). For q_1 (both q_{10} and q_{11}) on input 0, the next state is q_{10} . Similarly, for q_1 (both q_{10} and q_{11}), next state is q_{21} . For q_{10} , the output will be 0 and for q_{11} , the output will be 1. Similarly, other entries can be filled.

Conversion From Mealy to Moore Machine

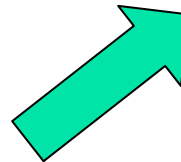
Present State	Input=0		Input=1	
	Next State	Output	Next State	Output
q0	q1	0	q2	0
q1	q1	0	q2	1
q2	q1	1	q2	0



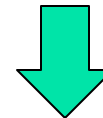
	Input =0	Input =1	
Present State	Next State	Next State	Output
q0	q10	q20	0
q10	q10	q21	0
q11	q10	q21	1
q20	q11	q20	0
q21	q11	q20	1

Conversion of Moore to Mealy Machine

	Input =0	Input =1	
Present State	Next State	Next State	Output
A	A	B	0
B	C	B	0
C	D	B	0
D	A	E	0
E	C	B	1



	Input=0		Input=1	
Present State	Next S tate	Output	Next S tate	Output
A	A		B	
B	C		B	
C	D		B	
D	A		E	
E	C		B	



	Input=0		Input=1	
Present State	Next S tate	Output	Next S tate	Output
A	A	0	B	0
B	C	0	B	0
C	D	0	B	0
D	A	0	E	1
E	C	0	B	0



Problems on Moore and Mealy Machine

- Design a Moore machine to generate 1's complement of a given binary number
- Design a Moore machine for a binary input sequence such that if it has a substring 101, the machine output A, if the input has substring 110, it outputs B other it outputs C.
- Design a Moore machine that determines whether an input string contains an even and odd number of 1's. The machine should give 1 as output of an even number of 1's are in the string and 0 otherwise
- Design a Mealy machine that scans sequence of input 0 and 1 and generates output 'A' if the input string terminates in 00, outputs 'B' if the string terminates in 11, and output 'C' otherwise.

Minimization of DFA

Dr. Kishore Kumar Sahu
Dept. of CSE, VSSUT, Burla

Minimization of DFA

- Suppose there is a DFA $D (Q, \Sigma, \delta, q_0, F)$ which recognizes a language L . Then the minimized DFA $D (Q', \Sigma, \delta, q_0, F')$ can be constructed for language L as:

Step 1: We will divide Q (set of states) into two sets. One set will contain all final states and other set will contain non-final states. This partition is called P_0 .

Step 2: Initialize $k = 1$

Step 3: Find P_k by partitioning the different sets of P_{k-1} . In each set of P_{k-1} , we will take all possible pair of states. If two states of a set are distinguishable, we will split the sets into different sets in P_k .

Step 4: Stop when $P_k = P_{k-1}$ (No change in partition)

Step 5: All states of one set are merged into one. No. of states in minimized DFA will be equal to no. of sets in P_k .

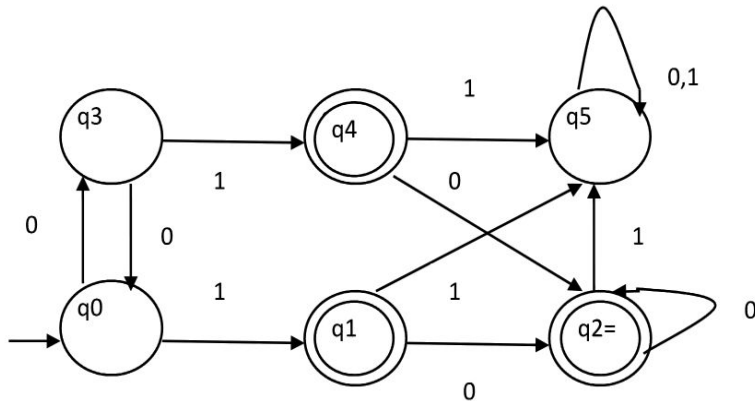
Partition class

- **How to find whether two states in partition P_k are distinguishable ?**

Two states (q_i, q_j) are distinguishable in partition P_k if for any input symbol a , $\delta(q_i, a)$ and $\delta(q_j, a)$ are in different sets in partition P_{k-1} .

Example

- Consider the following DFA shown in figure.



Q\Σ	0	1
->q0	q3	q1
*q1	q2	q5
*q2	q2	q5
q3	q0	q4
*q4	q4	q5
q5	q5	q5

Example (Cont.)

- **For set { q1, q2, q4 } :**
 $\delta(q1, 0) = \delta(q2, 0) = q2$ and
 $\delta(q1, 1) = \delta(q2, 1) = q5$, So
q1 and q2 are not distinguishable.
Similarly, $\delta(q1, 0) = \delta(q4, 0) = q2$ and $\delta(q1, 1) = \delta(q4, 1) = q5$, So q1 and q4 are not distinguishable.
Since, q1 and q2 are not distinguishable and q1 and q4 are also not distinguishable, So q2 and q4 are not distinguishable. So, { q1, q2, q4 } set will not be partitioned in P1.

Q\Σ	0	1
->q0	q3	q1
*q1	q2	q5
*q2	q2	q5
q3	q0	q4
*q4	q4	q5
q5	q5	q5

Example (Cont.)

- **ii) For set { q0, q3, q5 } :**
 $\delta^*(q_0, 0) = q_3$ and $\delta^*(q_3, 0) = q_0$
 $\delta^*(q_0, 1) = q_1$ and $\delta^*(q_3, 1) = q_4$
 So, q0 and q3 are not distinguishable
- $\delta^*(q_0, 0) = q_3$ and $\delta^*(q_5, 0) = q_5$ and $\delta^*(q_0, 1) = q_1$ and $\delta^*(q_5, 1) = q_5$
- So, q0 and q5 are distinguishable
- $P1 = \{ \{ q_1, q_2, q_4 \}, \{ q_0, q_3 \}, \{ q_5 \} \}$

$Q \backslash \Sigma$	0	1
$\rightarrow q_0$	q3	q1
*q1	q2	q5
*q2	q2	q5
q3	q0	q4
*q4	q4	q5
q5	q5	q5

Example (Cont.)

- To calculate P2, we will check whether sets of partition P1 can be partitioned or not:
iii) For set { q1, q2, q4 } :
 $\delta(q1, 0) = \delta(q2, 0) = q2$
and $\delta(q1, 1) = \delta(q2, 1) = q5$,
- $\delta(q1, 0) = \delta(q4, 0) = q2$
and $\delta(q1, 1) = \delta(q4, 1) = q5$
- So, { q1, q2, q4 } set will not be partitioned in P2

Q\Σ	0	1
->q0	q3	q1
*q1	q2	q5
*q2	q2	q5
q3	q0	q4
*q4	q4	q5
q5	q5	q5

Example (Cont.)

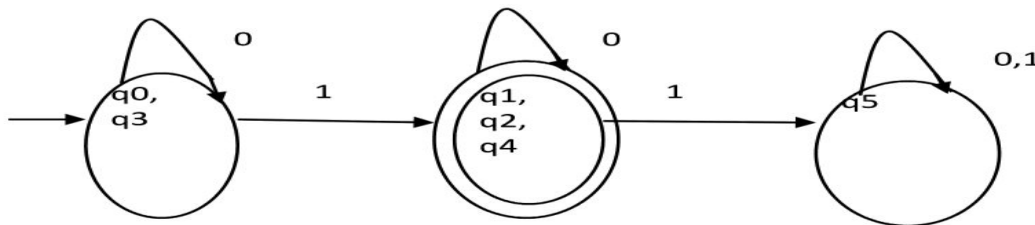
- To calculate P2, we will check whether sets of partition P1 can be partitioned or not:
iii) For set { q1, q2, q4 } :
 $\delta(q1, 0) = \delta(q2, 0) = q2$ and $\delta(q1, 1) = \delta(q2, 1) = q5$,
- $\delta(q1, 0) = \delta(q4, 0) = q2$ and $\delta(q1, 1) = \delta(q4, 1) = q5$
- So, { q1, q2, q4 } set will not be partitioned in P2
- iv) For set { q0, q3 } :**
 $\delta(q0, 0) = q3$ and $\delta(q3, 0) = q0$
 $\delta(q0, 1) = q1$ and $\delta(q3, 1) = q4$
- So, q0 and q3 are not distinguishable

Q\Σ	0	1
->q0	q3	q1
*q1	q2	q5
*q2	q2	q5
q3	q0	q4
*q4	q4	q5
q5	q5	q5

Example (Cont.)

- Since, $P1=P2$. So, this is the final partition.
Partition $P2$ means that $q1$, $q2$ and $q4$ states are merged into one.
Similarly, $q0$ and $q3$ are merged into one.

$Q \backslash \Sigma$	0	1
$\rightarrow q0$	$q3$	$q1$
$*q1$	$q2$	$q5$
$*q2$	$q2$	$q5$
$q3$	$q0$	$q4$
$*q4$	$q4$	$q5$
$q5$	$q5$	$q5$



Example 2

Given DFA

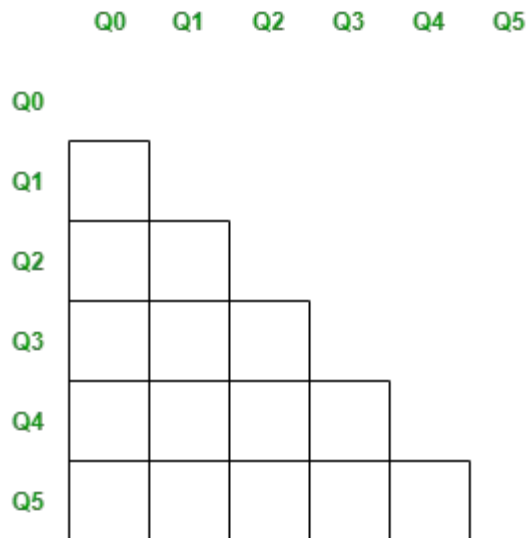
State	Input = a	Input = b
->q0	q1	q3
q1	q2	q4
q2	q1	q1
q3	q2	q4
*q4	q4	q4

Minimized DFA

State	Input = a	Input = b
->{q0,q2}	{q1,q3}	{q1,q3}
{q1,q3}	{q0,q2}	{q4}
{q4}	{q4}	{q4}

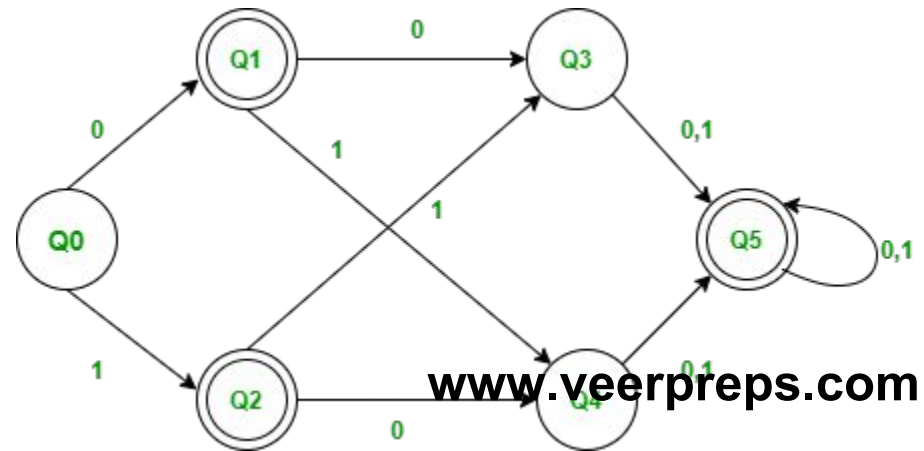
Myhill-Nerode Theorem

Create the pairs of all the states involved in DFA.



Given DFA

States	Inputs	
	0	1
Q0	Q1	Q2
Q1	Q3	Q4
Q2	Q4	Q3
Q3	Q5	Q5
Q4	Q5	Q5
Q5	Q5	Q5



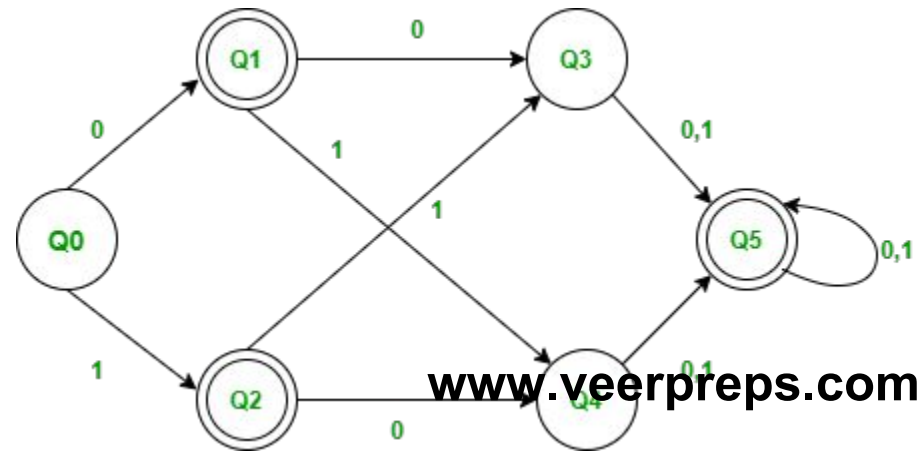
Myhill-Nerode Theorem (Cont.)

Mark all the pairs (Q_a, Q_b) such a that Q_a is Final state and Q_b is Non-Final State.

	Q0	Q1	Q2	Q3	Q4	Q5
Q0						
Q1	✓					
Q2	✓					
Q3		✓	✓			
Q4		✓	✓			
Q5	✓			✓	✓	

Given DFA

States	Inputs	
	0	1
Q0	Q1	Q2
Q1	Q3	Q4
Q2	Q4	Q3
Q3	Q5	Q5
Q4	Q5	Q5
Q5	Q5	Q5



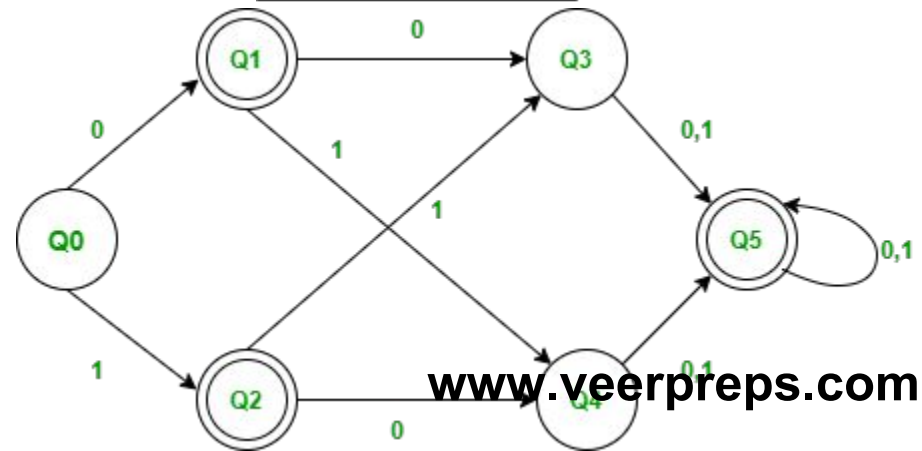
Myhill-Nerode Theorem (Cont.)

Mark all the pairs (Q_a, Q_b) such a that Q_a is Final state and Q_b is Non-Final State.

	Q0	Q1	Q2	Q3	Q4	Q5
Q0						
Q1	✓					
Q2	✓					
Q3		✓	✓			
Q4		✓	✓			
Q5	✓	✓		✓	✓	

Given DFA

States	Inputs	
	0	1
Q0	Q1	Q2
Q1	Q3	Q4
Q2	Q4	Q3
Q3	Q5	Q5
Q4	Q5	Q5
Q5	Q5	Q5



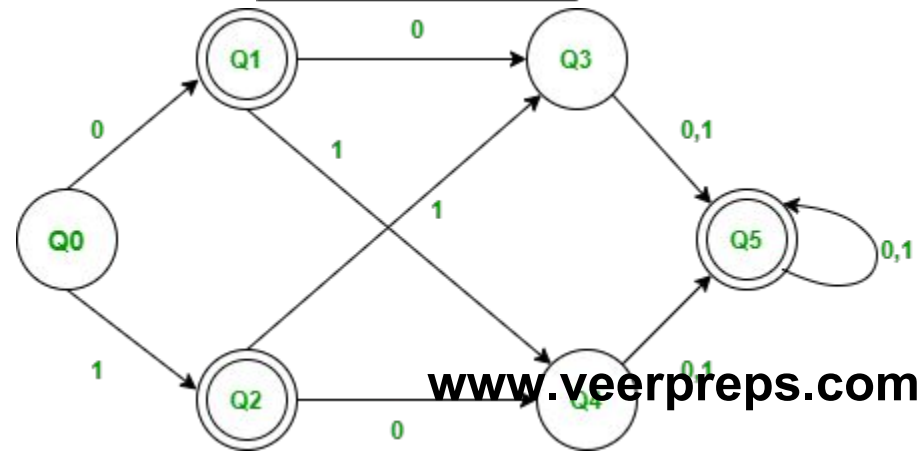
Myhill-Nerode Theorem (Cont.)

Mark all the pairs (Q_a, Q_b) such a that Q_a is Final state and Q_b is Non-Final State.

	Q0	Q1	Q2	Q3	Q4	Q5
Q0						
Q1	✓					
Q2	✓					
Q3		✓	✓			
Q4		✓	✓			
Q5	✓	✓	✓	✓	✓	

Given DFA

States	Inputs	
	0	1
Q0	Q1	Q2
Q1	Q3	Q4
Q2	Q4	Q3
Q3	Q5	Q5
Q4	Q5	Q5
Q5	Q5	Q5



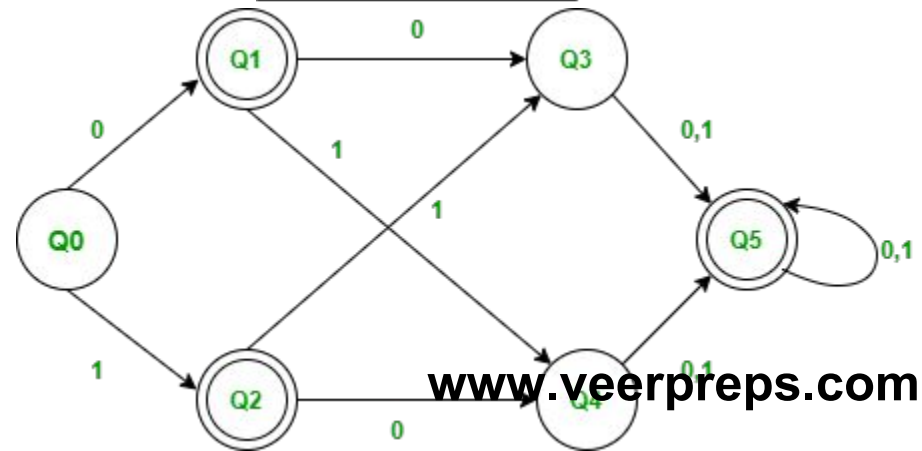
Myhill-Nerode Theorem (Cont.)

Mark all the pairs (Q_a, Q_b) such a that Q_a is Final state and Q_b is Non-Final State.

	Q0	Q1	Q2	Q3	Q4	Q5
Q0						
Q1	✓					
Q2	✓					
Q3	✓	✓	✓			
Q4		✓	✓			
Q5	✓	✓	✓	✓	✓	

Given DFA

States	Inputs	
	0	1
Q0	Q1	Q2
Q1	Q3	Q4
Q2	Q4	Q3
Q3	Q5	Q5
Q4	Q5	Q5
Q5	Q5	Q5



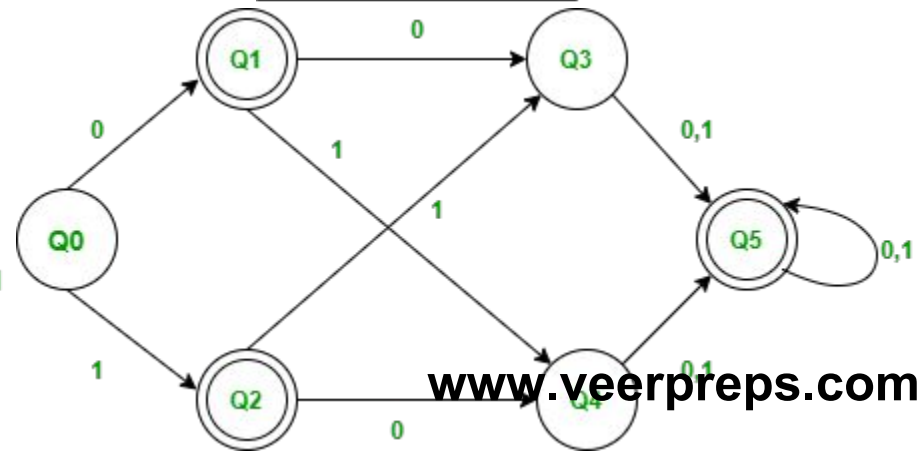
Myhill-Nerode Theorem (Cont.)

Mark all the pairs (Q_a, Q_b) such a that Q_a is Final state and Q_b is Non-Final State.

Given DFA

	Q0	Q1	Q2	Q3	Q4	Q5
Q0						
Q1	✓					
Q2	✓					
Q3	✓	✓	✓			
Q4	✓	✓	✓			
Q5	✓	✓	✓	✓	✓	

States	Inputs	
	0	1
Q0	Q1	Q2
Q1	Q3	Q4
Q2	Q4	Q3
Q3	Q5	Q5
Q4	Q5	Q5
Q5	Q5	Q5





Regular Expressions

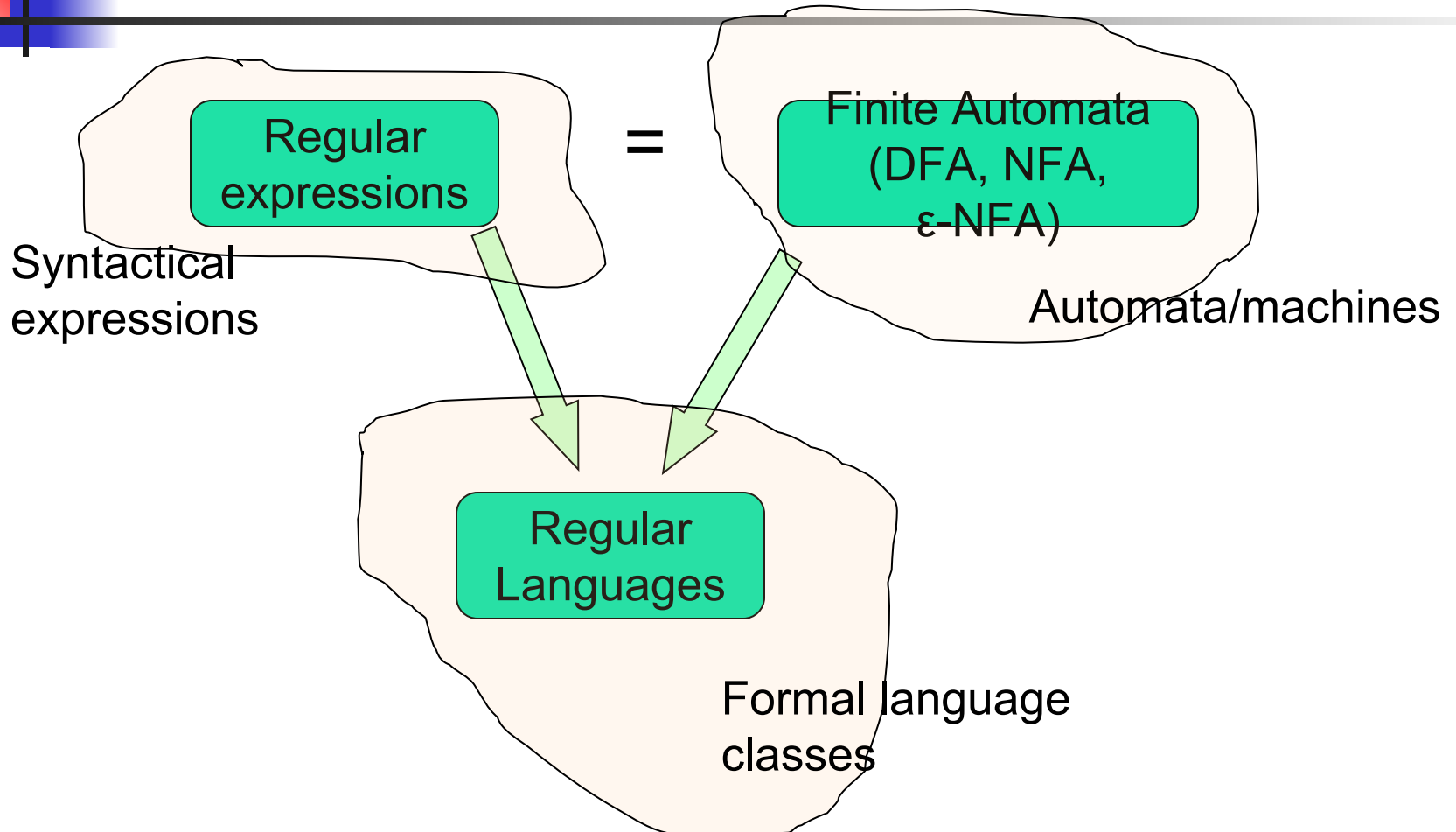
Reading: Chapter 3



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:
 - $L^0 = \{\epsilon\}$
 - $L^1 = \{w \mid \text{for some } w \in L\}$
 - $L^2 = \{w_1 w_2 \mid w_1 \in L, w_2 \in L \text{ (duplicates allowed)}\}$
 - $L^i = \{w_1 w_2 \dots w_i \mid \text{all } w\text{'s chosen are } \in L \text{ (duplicates allowed)}\}$
 - (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 $L \neq \{\epsilon\}$
- If $L = \{\epsilon\}$, then $L^* = \{\epsilon\}$
- If $L = \Phi$, then $L^* = \{\epsilon\}$

Why?

Why?

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 $L = (\epsilon + 1)(01)^*(\epsilon + 0)$



Precedence of Operators

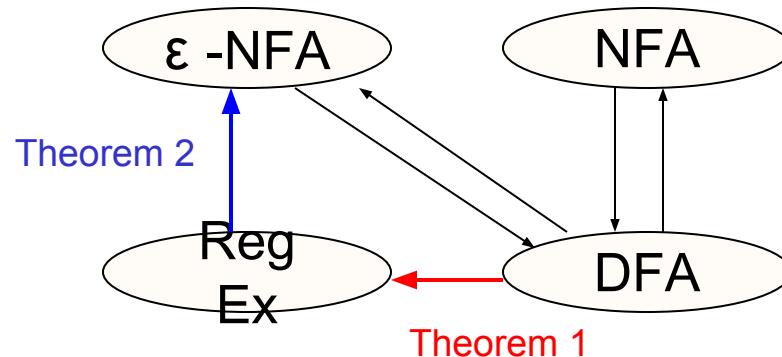
- Highest to lowest
 - * operator (star)
 - . (concatenation)
 - + operator

- 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

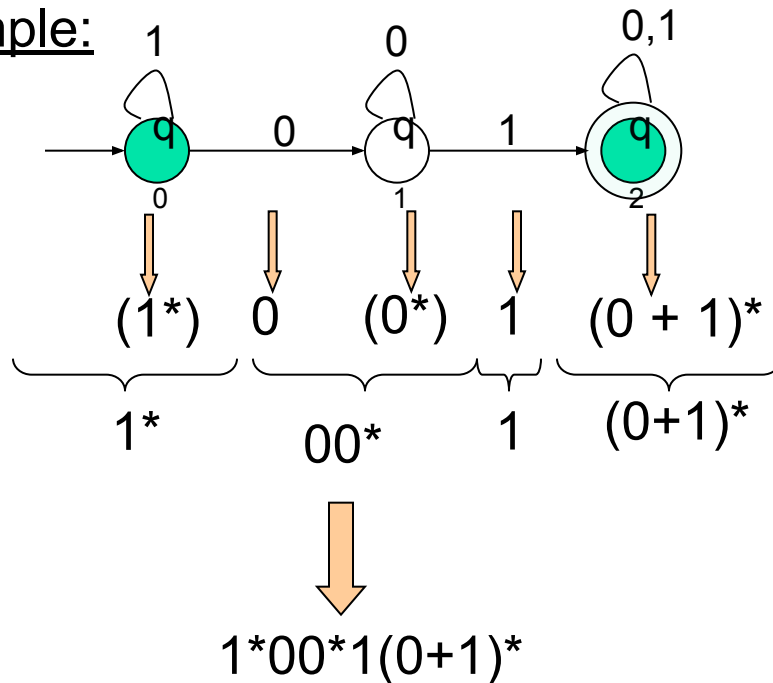
Theorem 1

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

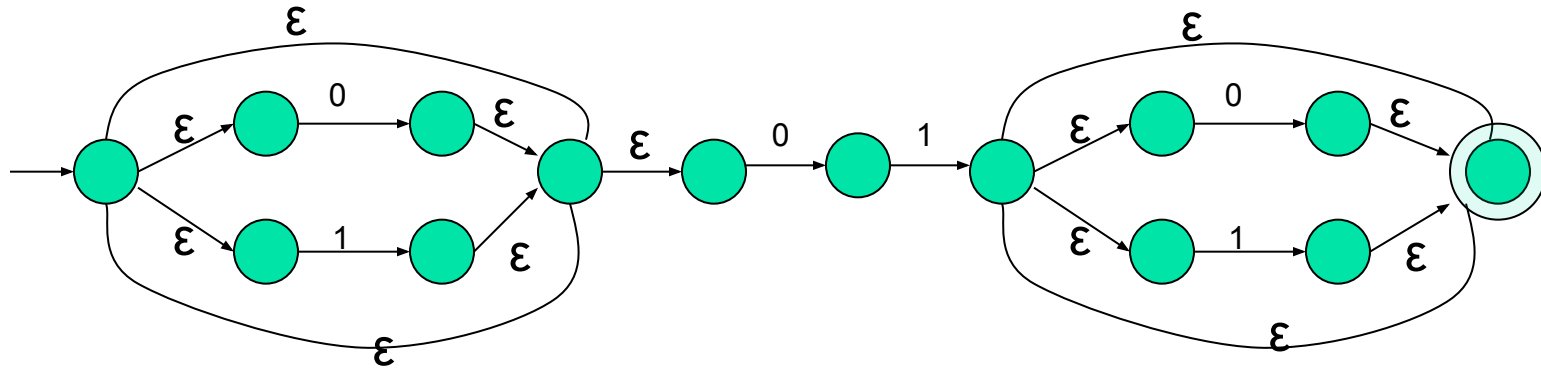
RE to ϵ -NFA construction

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

IDENTITIES OF RE AND ITS APPLICATIONS

Dr. Kishore Kumar Sahu
Dept. of CSE, VSSUT, Burla

IDENTITIES FOR RE

We now give the identities for regular expressions; these are useful for simplifying regular expressions.

$$I_1 \quad \emptyset + R = R$$

$$I_2 \quad \emptyset R = R \emptyset = \emptyset$$

$$I_3 \quad \Lambda R = R \Lambda = R$$

$$I_4 \quad \Lambda^* = \Lambda \text{ and } \emptyset^* = \Lambda$$

$$I_5 \quad R + R = R$$

$$I_6 \quad R^* R^* = R^*$$

$$I_7 \quad R R^* = R^* R$$

$$I_8 \quad (R^*)^* = R^*$$

$$I_9 \quad \Lambda + R R^* = R^* = \Lambda + R^* R$$

$$I_{10} \quad (PQ)^* P = P(QP)^*$$

$$I_{11} \quad (P + Q)^* = (P^* Q^*)^* = (P^* + Q^*)^*$$

$$I_{12} \quad (P + Q)R = PR + QR \quad \text{and} \quad R(P + Q) = RP + RQ$$

Arden's theorem

Theorem 5.1 (Arden's theorem) Let P and Q be two regular expressions over Σ . If P does not contain Λ , then the following equation in R , namely

$$R = Q + RP \quad (5.1)$$

has a unique solution (i.e. one and only one solution) given by $R = QP^*$.

Proof $Q + (QP^*)P = Q(\Lambda + P^*P) = QP^*$ by I_9

Hence (5.1) is satisfied when $R = QP^*$. This means $R = QP^*$ is a solution of (5.1).

To prove uniqueness, consider (5.1). Here, replacing R by $Q + RP$ on the R.H.S., we get the equation

$$\begin{aligned} Q + RP &= Q + (Q + RP)P \\ &= Q + QP + RPP \\ &= Q + QP + RP^2 \\ &= Q + QP + QP^2 + \dots + QP^i + RP^{i+1} \\ &= Q(\Lambda + P + P^2 + \dots + P^i) + RP^{i+1} \end{aligned}$$

Arden's theorem

From (5.1).

$$\mathbf{R} = \mathbf{Q}(\Lambda + \mathbf{P} + \mathbf{P}^2 + \dots + \mathbf{P}^i) + \mathbf{R}\mathbf{P}^{i+1} \quad \text{for } i \geq 0 \quad (5.2)$$

We now show that any solution of (5.1) is equivalent to \mathbf{QP}^* . Suppose \mathbf{R} satisfies (5.1), then it satisfies (5.2). Let w be a string of length i in the set \mathbf{R} . Then w belongs to the set $\mathbf{Q}(\Lambda + \mathbf{P} + \mathbf{P}^2 + \dots + \mathbf{P}^i) + \mathbf{R}\mathbf{P}^{i+1}$. As \mathbf{P} does not contain Λ , $\mathbf{R}\mathbf{P}^{i+1}$ has no string of length less than $i + 1$ and so w is not in the set $\mathbf{R}\mathbf{P}^{i+1}$. This means that w belongs to the set $\mathbf{Q}(\Lambda + \mathbf{P} + \mathbf{P}^2 + \dots + \mathbf{P}^i)$, and hence to \mathbf{QP}^* .

Consider a string w in the set \mathbf{QP}^* . Then w is in the set \mathbf{QP}^k for some $k \geq 0$, and hence in $\mathbf{Q}(\Lambda + \mathbf{P} + \mathbf{P}^2 + \dots + \mathbf{P}^k)$. So w is on the R.H.S. of (5.2). Therefore, w is in \mathbf{R} (L.H.S. of (5.2)). Thus \mathbf{R} and \mathbf{QP}^* represent the same set. This proves the uniqueness of the solution of (5.1). **I**

Example

- Q Give an r.e. for representing the set L of strings in which every 0 is immediately followed by at least two 1's.
- Soln:

If w is in L , then either (a) w does not contain any 0, or (b) it contains a 0 preceded by 1 and followed by 11. So w can be written as $w_1w_2 \dots w_n$, where each w_i is either 1 or 011. So L is represented by the r.e. $(1 + 011)^*$.

Example

- Q Prove that the regular expression $R = \Lambda + 1^*(011)^*(1^*(011)^*)^*$ also describes the same set of strings.
- Soln:

$$\begin{aligned} R &= \Lambda + P_1 P_1^*, \text{ where } P_1 = 1^*(011)^* \\ &= P_1^* && \text{using } I_9 \\ &= (1^*(011)^*)^* \\ &= (P_2^* P_3^*)^* && \text{letting } P_2 = 1, P_3 = 011 \\ &= (P_2 + P_3)^* && \text{using } I_{11} \\ &= (1 + 011)^* \end{aligned}$$

Example

Prove $(1 + 00^*1) + (1 + 00^*1)(0 + 10^*1)^* (0 + 10^*1) = 0^*1(0 + 10^*1)^*$.

Solution

$$\begin{aligned}\text{L.H.S.} &= (1 + 00^*1) (\Lambda + (0 + 10^*1)^* (0 + 10^*1)) && \text{using } I_{12} \\ &= (1 + 00^*1) (0 + 10^*1)^* && \text{using } I_9 \\ &= (\Lambda + 00^*)1 (0 + 10^*1)^* && \text{using } I_{12} \text{ for } 1 + 00^*1 \\ &= 0^*1(0 + 10^*1)^* && \text{using } I_9 \\ &= \text{R.H.S.}\end{aligned}$$

DFA TO REGULAR USING ARDEN'S THEOREM

This is used to find the r.e. recognized by a transition system.

The following assumptions are made regarding the transition system:

- (i) The transition graph does not have Λ -moves.
- (ii) It has only one initial state, say v_1 .
- (iii) Its vertices are $v_1 \dots v_n$.
- (iv) V_i the r.e. represents the set of strings accepted by the system even though v_i is a final state.
- (v) α_{ij} denotes the r.e. representing the set of labels of edges from v_i to v_j . When there is no such edge, $\alpha_{ij} = \emptyset$. Consequently, we can get the following set of equations in $V_1 \dots V_n$:

$$V_1 = V_1\alpha_{11} + V_2\alpha_{21} + \dots + V_n\alpha_{n1} + \Lambda$$

$$V_2 = V_1\alpha_{12} + V_2\alpha_{22} + \dots + V_n\alpha_{n2}$$

$$\vdots$$

$$V_n = V_1\alpha_{1n} + V_2\alpha_{2n} + \dots + V_n\alpha_{nn}$$

By repeatedly applying substitutions and Theorem 5.1 (Arden's theorem), we can express V_i in terms of α_{ij} 's.

For getting the set of strings recognized by the transition system, we have to take the 'union' of all V_i 's corresponding to final states.

EXAMPLE

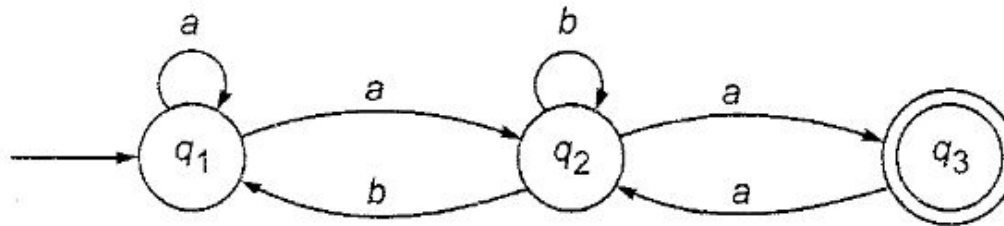


Fig. 5.13 Transition system of Example 5.8.

Solution

We can directly apply the above method since the graph does not contain any Λ -move and there is only one initial state.

The three equations for q_1 , q_2 and q_3 can be written as

$$q_1 = q_1a + q_2b + \Lambda, \quad q_2 = q_1a + q_2b + q_3a, \quad q_3 = q_2a$$

It is necessary to reduce the number of unknowns by repeated substitution. By substituting q_3 in the q_2 -equation, we get by applying Theorem 5.1

$$\begin{aligned}
 q_2 &= q_1a + q_2b + q_2aa \\
 &= q_1a + q_2(b + aa) \\
 &= q_1a(b + aa)^*
 \end{aligned}$$

EXAMPLE (Cont.)

Substituting q_2 in q_1 , we get

$$\begin{aligned}q_1 &= q_1 a + q_1 a(b + aa)^* b + \Lambda \\ &= q_1(a + a(b + aa)^* b) + \Lambda\end{aligned}$$

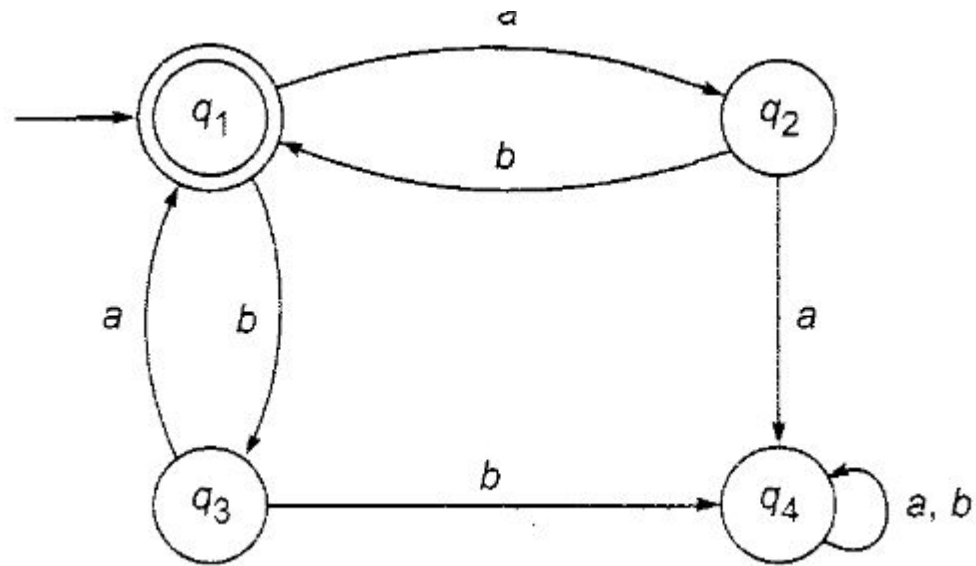
Hence,

$$\begin{aligned}q_1 &= \Lambda(a + a(b + aa)^* b)^* \\ q_2 &= (a + a(b + aa)^* b)^* a(b + aa)^* \\ q_3 &= (a + a(b + aa)^* b)^* a(b + aa)^* a\end{aligned}$$

Since q_3 is a final state, the set of strings recognized by the graph is given by

$$(a + a(b + aa)^* b)^* a(b + aa)^* a$$

EXAMPLE



EXAMPLE (Cont.)

$$q_1 = q_2b + q_3a + \Lambda$$

$$q_2 = q_1a$$

$$q_3 = q_1b$$

$$q_4 = q_2a + q_3b + q_4a + q_4b$$

As q_1 is the only final state and the q_1 -equation involves only q_2 and q_3 , we use only q_2 - and q_3 -equations (the q_4 -equation is redundant for our purposes). Substituting for q_2 and q_3 , we get

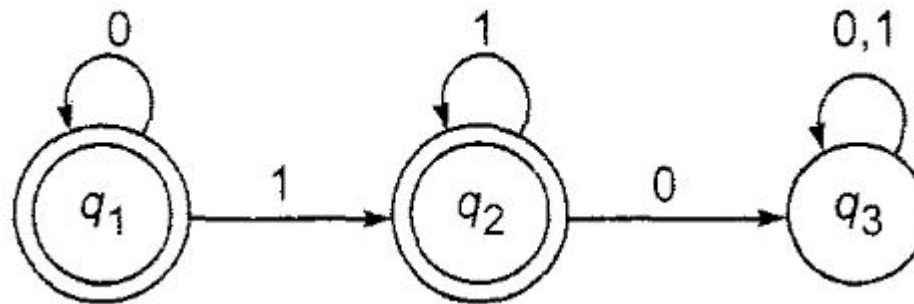
$$q_1 = q_1ab + q_1ba + \Lambda = q_1(ab + ba) + \Lambda$$

By applying Theorem 5.1, we get

$$q_1 = \Lambda(ab + ba)^* = (ab + ba)^*$$

As q_1 is the only final state, the strings accepted by the given finite automaton are the strings given by $(ab + ba)^*$. As any such string is a string of ab 's, and ba 's, we get an equal number of a 's and b 's. If a prefix x of a sentence

EXAMPLE



EXAMPLE (Cont.)

$$q_1 = q_1 0 + \Lambda$$

$$q_2 = q_1 1 + q_2 1$$

$$q_3 = q_2 0 + q_3(0 + 1)$$

By applying Theorem 5.1 to the q_1 -equation, we get

$$q_1 = \Lambda 0^* = 0^*$$

So,

$$q_2 = q_1 1 + q_2 1 = 0^* 1 + q_2 1$$

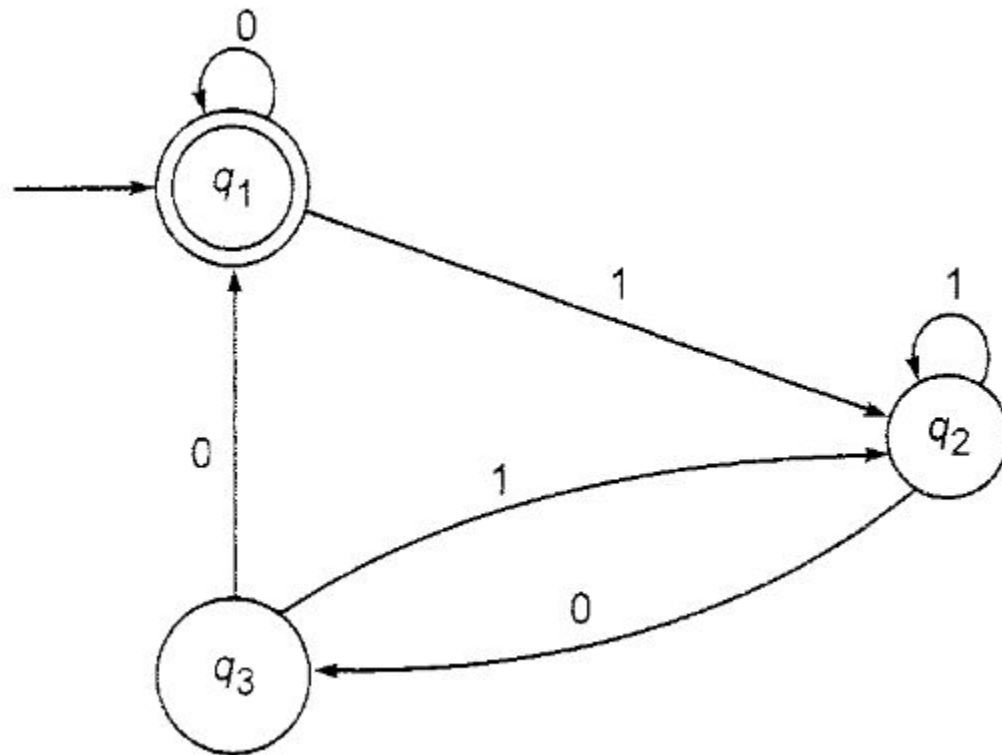
Therefore,

$$q_2 = (0^* 1) 1^*$$

As the final states are q_1 and q_2 , we need not solve for q_3 :

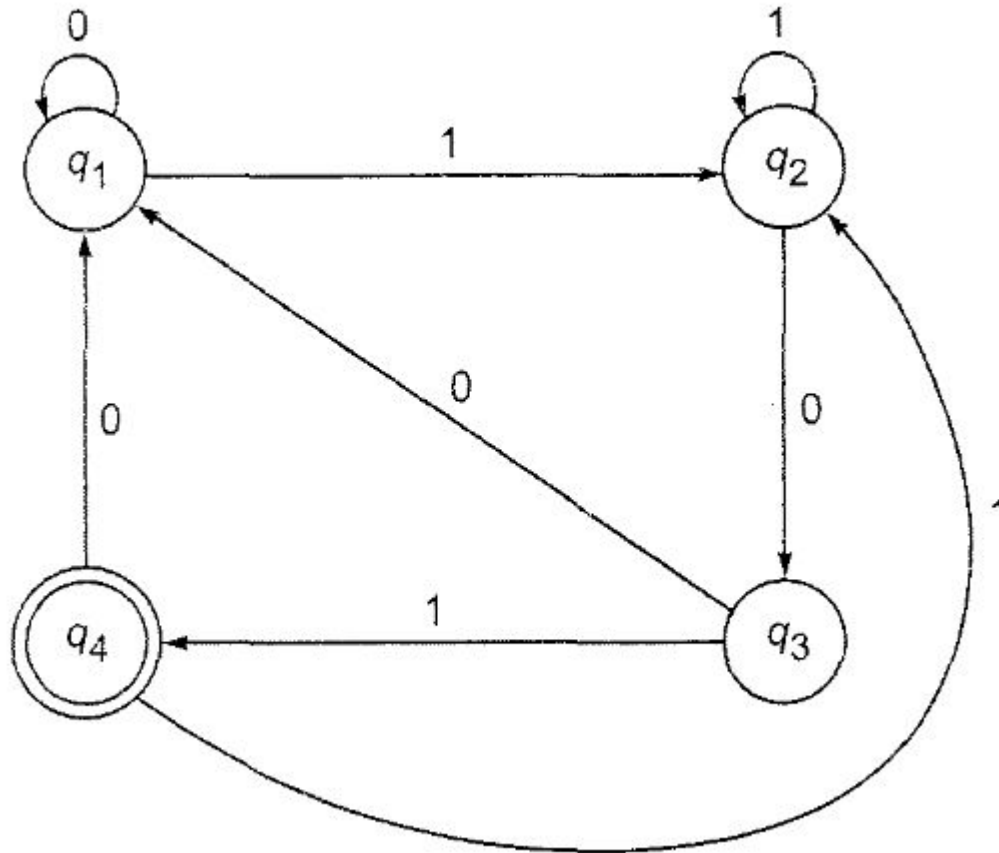
$$q_1 + q_2 = 0^* + 0^*(11^*) = 0^*(\Lambda + 11^*) = 0^*(1^*) \quad \text{by } I_9$$

Assignment



$$q_1 = \Lambda(0 + 1(1 + 01)^* 00)^* = (0 + 1(1 + 01)^* 00)^*$$

Assignment



$$\begin{aligned}
 q_4 &= q_2 01 = q_1 (1 + 011)^* 01 \\
 &= (0 + 1(1 + 011)^*(00 + 010))^* (1(1 + 011)^* 01)
 \end{aligned}$$

State Elimination Method

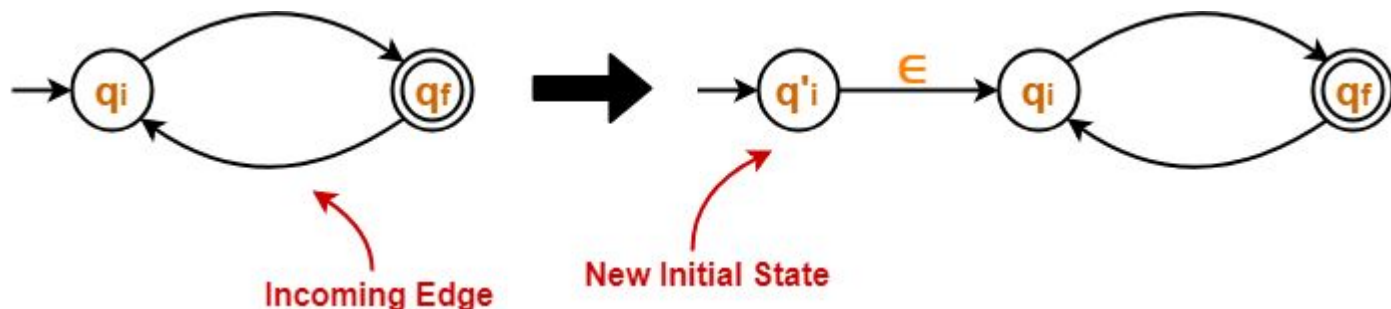
- This method involves the following steps in finding the regular expression for any given DFA

Step-01:

Thumb Rule (The initial state of the DFA must not have any incoming edge)

- If there exists any incoming edge to the initial state, then create a new initial state having no incoming edge to it

Example-



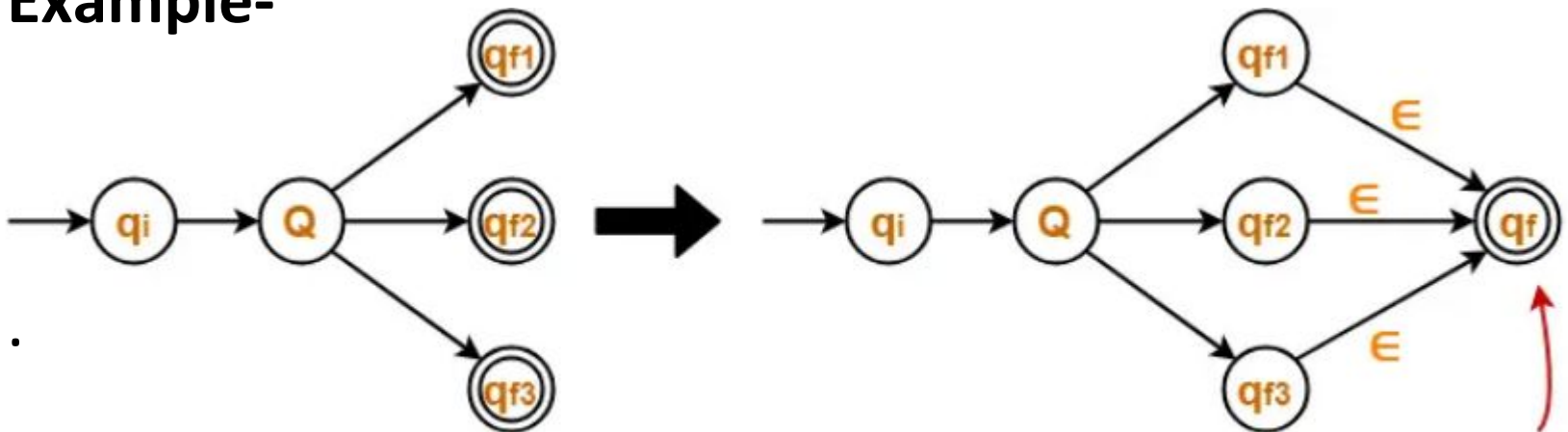
State Elimination Method (Cont.)

- **Step-02:**

Thumb Rule (There must exist only one final state in the DFA)

- If there exists multiple final states in the DFA, then convert all the final states into non-final states and create a new single final state.

Example-



Multiple Final States

New Final State

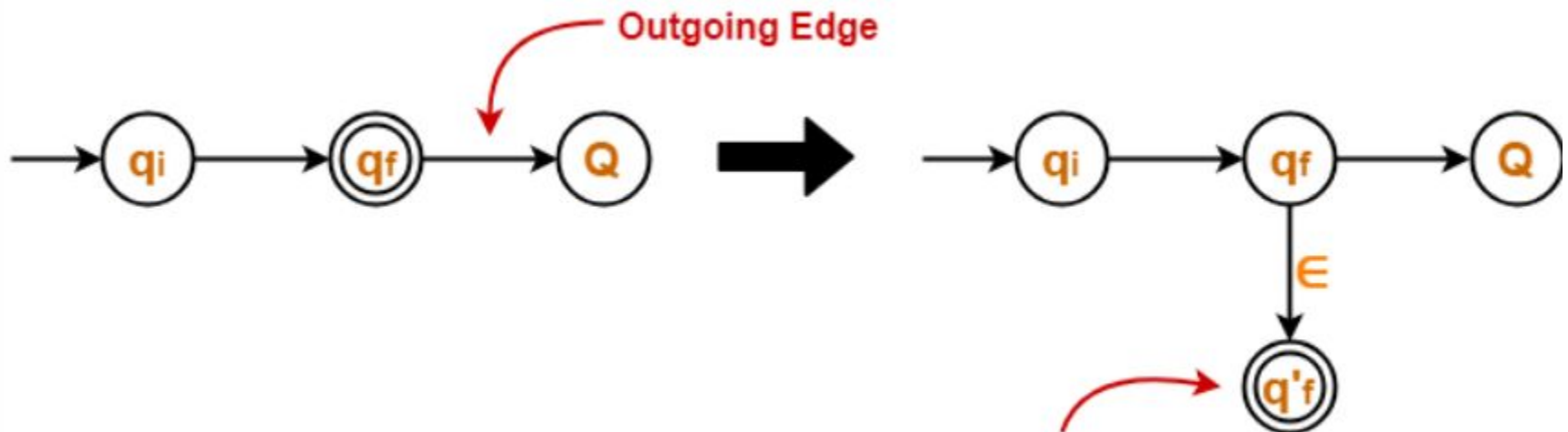
State Elimination Method (Cont.)

- **Step-03:**

Thumb Rule (The final state of the DFA must not have any outgoing edge)

- If there exists any outgoing edge from the final state, then create a new final state having no outgoing edge from it.

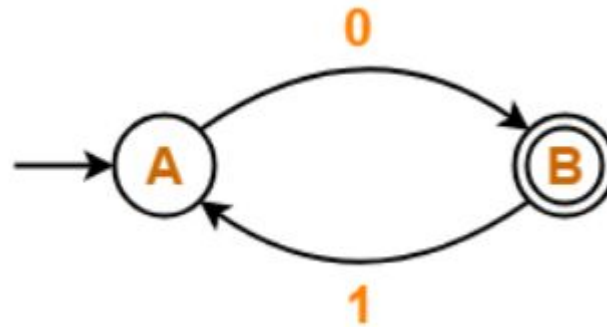
Example-



State Elimination Method (Cont.)

- **Step-04:**
 - Eliminate all the intermediate states one by one.
 - These states may be eliminated in any order.
- In the end,
 - Only an initial state going to the final state will be left.
 - The cost of this transition is the required regular expression.
- **NOTE**
 - The state elimination method can be applied to any finite automata (NFA, ϵ -NFA, DFA etc)

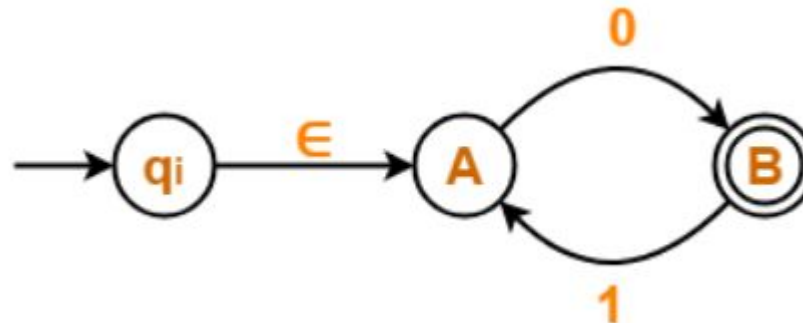
Example 01



Step-01:

- Initial state A has an incoming edge.
- So, we create a new initial state q_i .

The resulting DFA is-

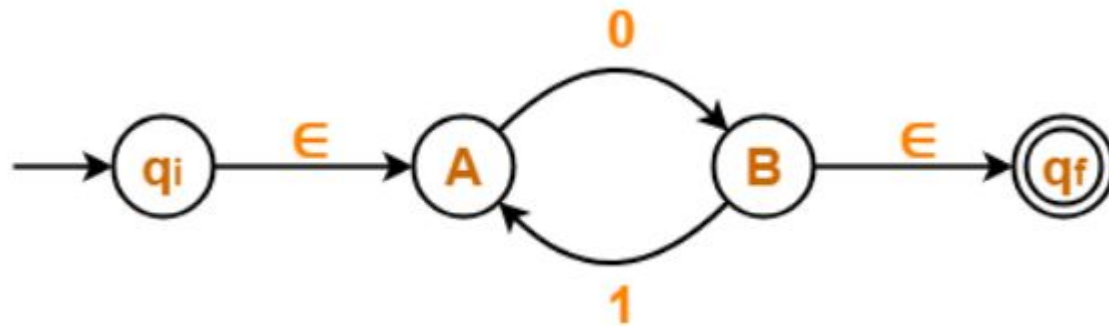


Example 01 (Cont.)

Step-02:

- Final state B has an outgoing edge.
- So, we create a new final state q_f .

The resulting DFA is-



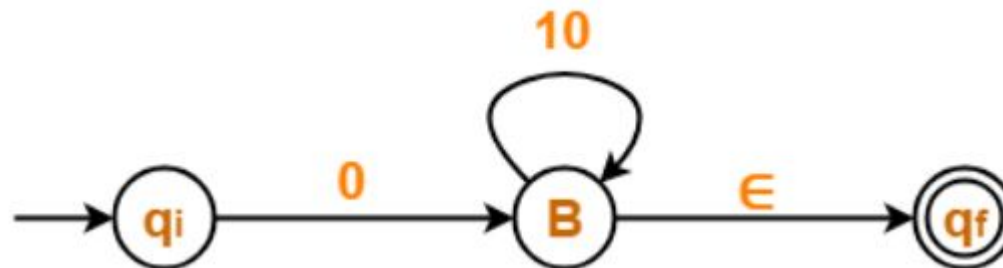
Example 01 (Cont.)

Step-03: Now, we start eliminating the intermediate states.

First, let us eliminate state A.

- There is a path going from state q_i to state B via state A.
- So, after eliminating state A, we put a direct path from state q_i to state B having cost $\in .0 = 0$
- There is a loop on state B using state A.
- So, after eliminating state A, we put a direct loop on state B having cost $1.0 = 10$.

Eliminating state A, we get-



Example 01 (Cont.)

Step-04: Now, let us eliminate state B.

- There is a path going from state q_i to state q_f via state B.
- So, after eliminating state B, we put a direct path from state q_i to state q_f having cost $0.(10)^*.\in = 0(10)^*$

Eliminating state B, we get-

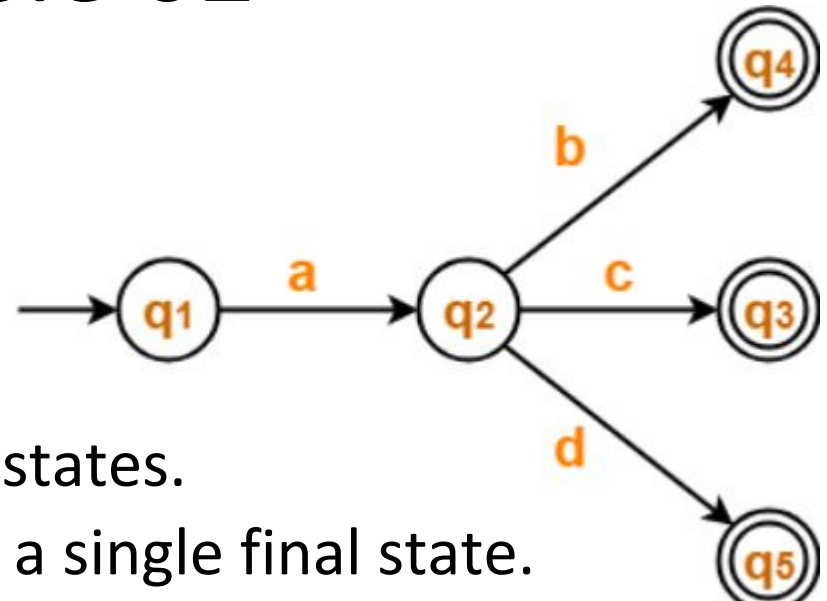


Regular Expression = $0(10)^*$

NOTE- In the above question,

- If we first eliminate state B and then state A, then regular expression would be $= (01)^*0$.
- This is also the same and correct.

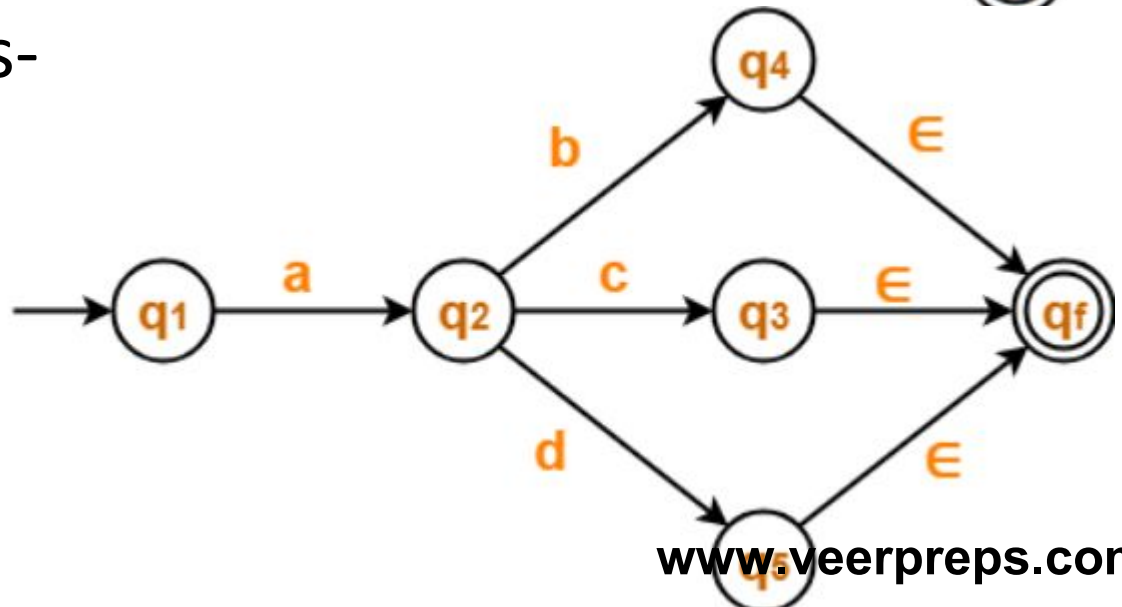
Example 02



Step-01:

- There exist multiple final states.
- So, we convert them into a single final state.

The resulting DFA is-



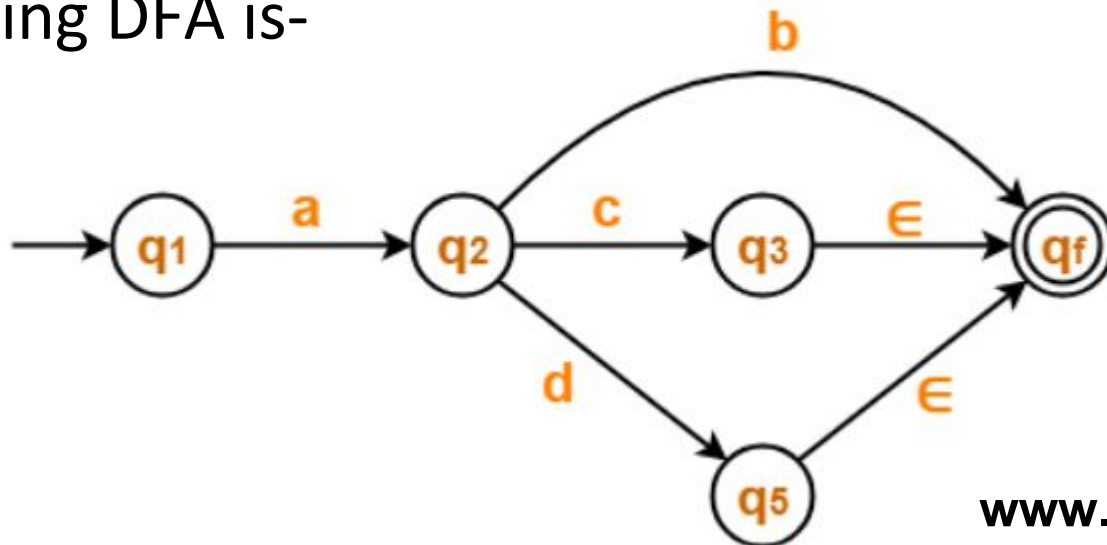
Example 02 (Cont.)

Step-02: Now, we start eliminating the intermediate states

First, let us eliminate state q_4 .

- There is a path going from state q_2 to state q_f via state q_4 .
- So, after eliminating state q_4 , we put a direct path from state q_2 to state q_f having cost $b. \in = b$.

The resulting DFA is-

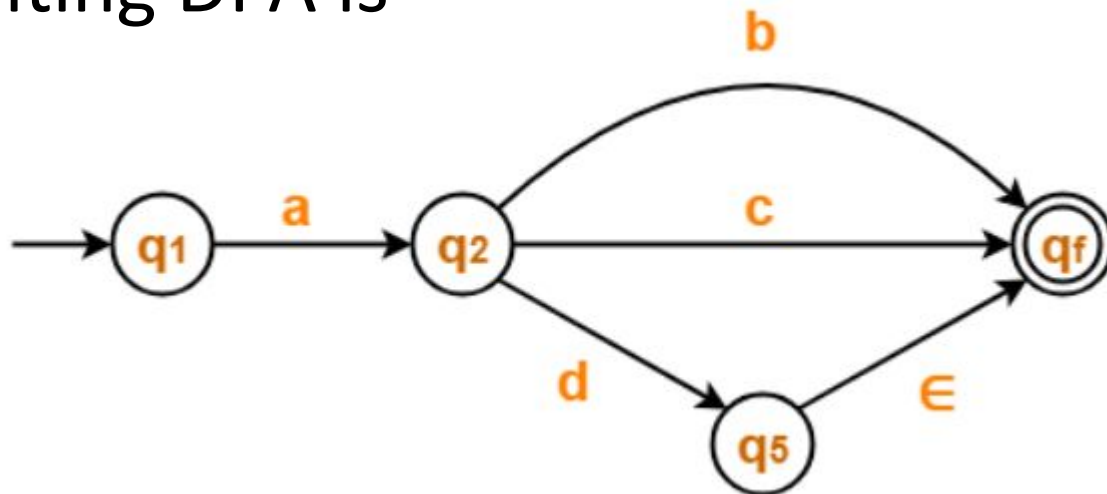


Example 02 (Cont.)

Step-03: Now, let us eliminate state q_3 .

- There is a path going from state q_2 to state q_f via state q_3 .
- So, after eliminating state q_3 , we put a direct path from state q_2 to state q_f having cost $c. \in = c$.

The resulting DFA is-

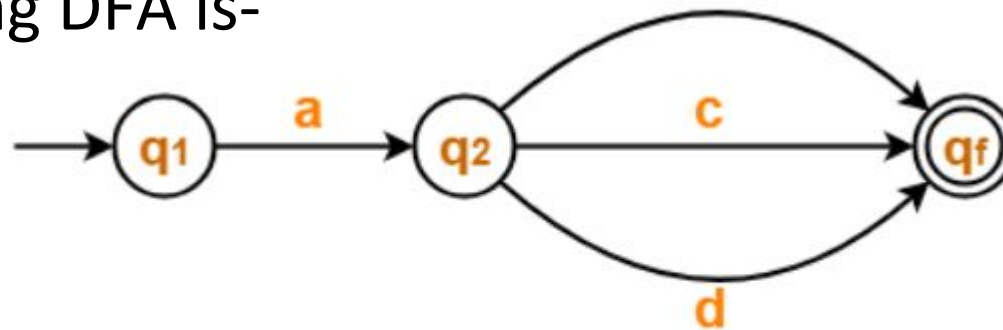


Example 02 (Cont.)

Step-04: Now, let us eliminate state q_5 .

- There is a path going from state q_2 to state q_f via state q_5 .
- So, after eliminating state q_5 , we put a direct path from state q_2 to state q_f having cost $d \cup c = d + c$.

The resulting DFA is-



Example 02 (Cont.)

Step-05: Now, let us eliminate state q_2 .

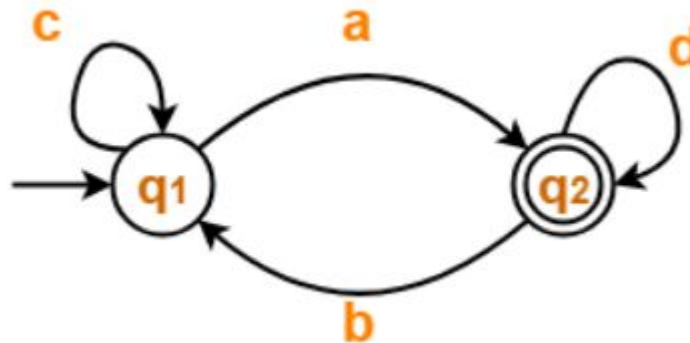
- There is a path going from state q_1 to state q_f via state q_2 .
- So, after eliminating state q_2 , we put a direct path from state q_1 to state q_f having cost $a.(b+c+d)$.

The resulting DFA is-



Regular Expression = $a(b+c+d)$

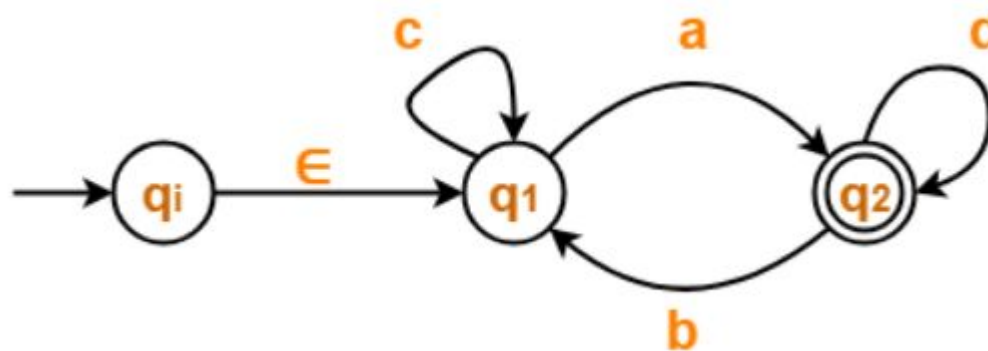
Example 03



Step-01:

- Initial state q_1 has an incoming edge.
- So, we create a new initial state q_i .

The resulting DFA is-

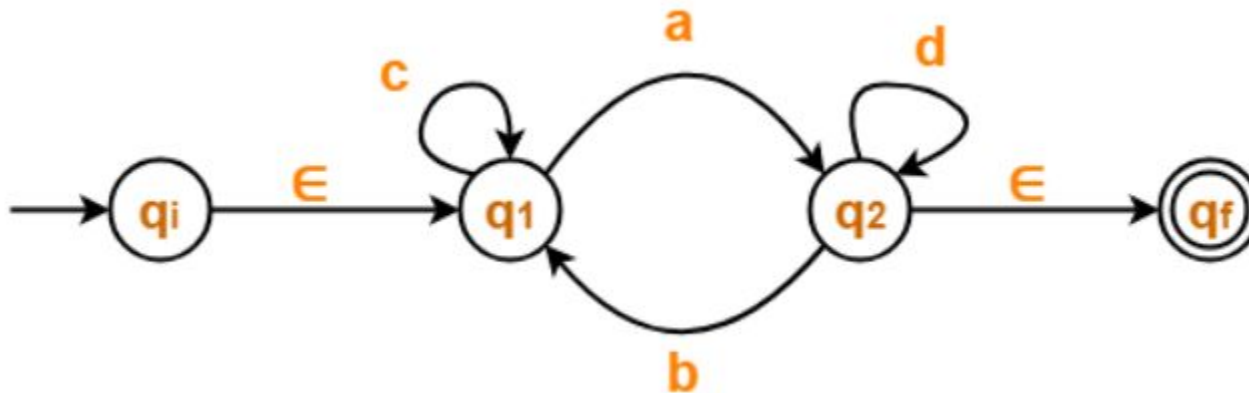


Example 03 (Cont.)

Step-02:

- Final state q_2 has an outgoing edge.
- So, we create a new final state q_f .

The resulting DFA is-

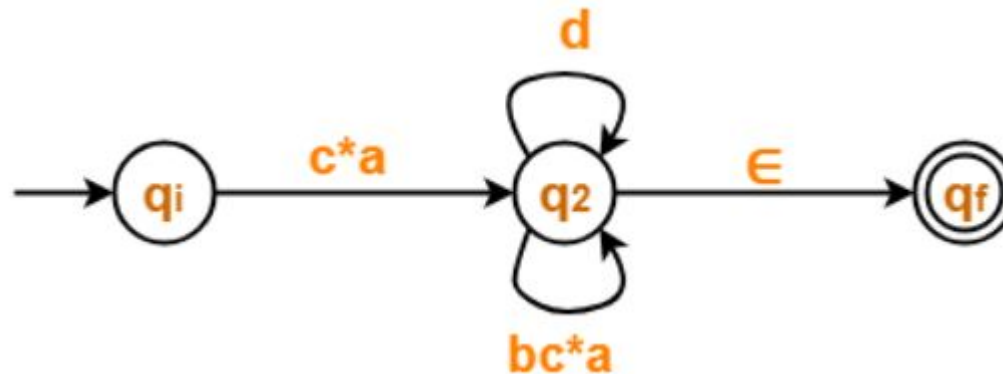


Example 03 (Cont.)

Step-03: Now, we start eliminating the intermediate states.

First, let us eliminate state q_1 .

- There is a path going from state q_i to state q_2 via state q_1 .
 - So, after eliminating state q_1 , we put a direct path from state q_i to state q_2 having cost $\in .c^*.a = c^*a$
 - There is a loop on state q_2 using state q_1 .
 - So, after eliminating state q_1 , we put a direct loop on state q_2 having cost $b.c^*.a = bc^*a$
- Eliminating state q_1 , we get-



Example 03 (Cont.)

Step-04: Now, let us eliminate state q_2 .

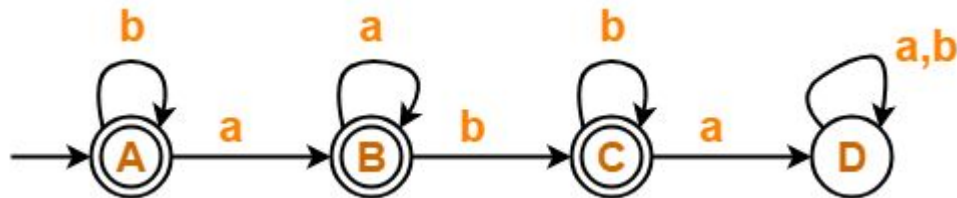
- There is a path going from state q_i to state q_f via state q_2 .
- So, after eliminating state q_2 , we put a direct path from state q_i to state q_f having cost $c^*a(d+bc^*a)^* \in = c^*a(d+bc^*a)^*$

The resulting DFA is-



Regular Expression = $c^*a(d+bc^*a)^*$

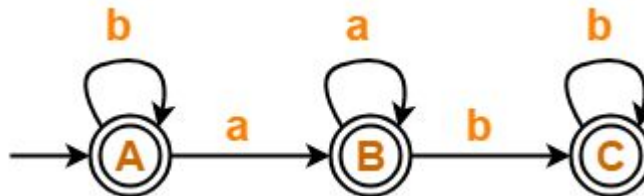
Example 04



Step-01:

- State D is a dead state as it does not reach to any final state.
- So, we eliminate state D and its associated edges.

The resulting DFA is-

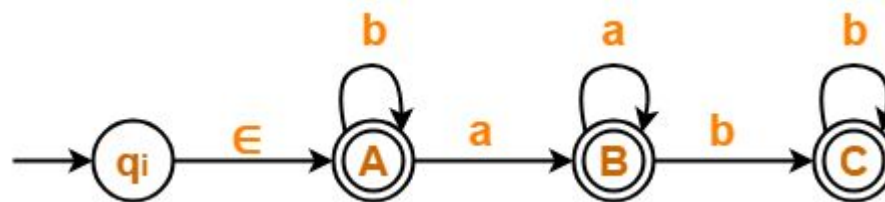


Example 04 (Cont.)

Step-02:

- Initial state A has an incoming edge (self loop).
- So, we create a new initial state q_i .

The resulting DFA is-

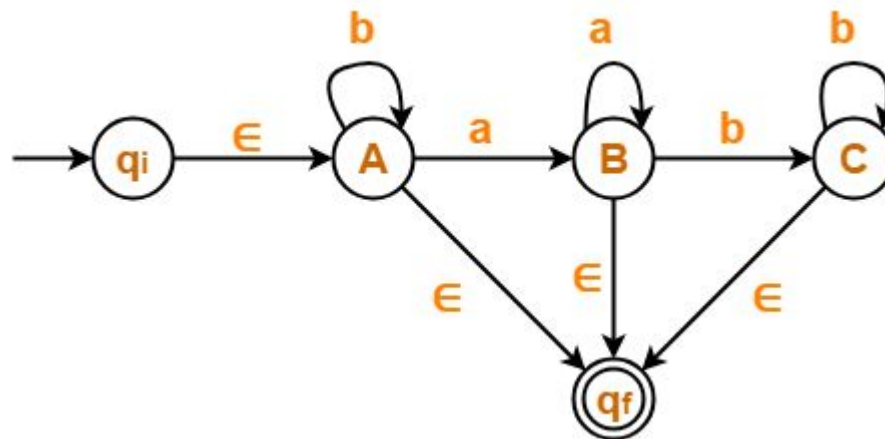


Example 04 (Cont.)

Step-03:

- There exist multiple final states.
- So, we convert them into a single final state.

- The resulting DFA is-



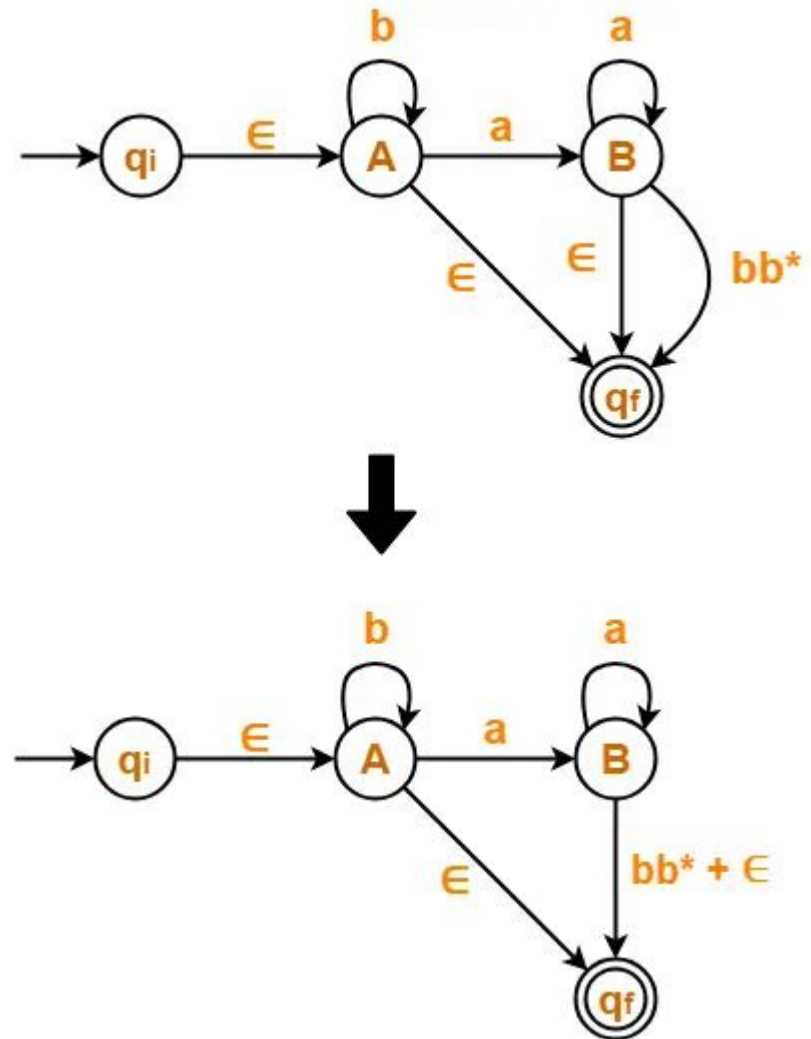
Example 04 (Cont.)

Step-04: Now, we start eliminating the intermediate states.

First, let us eliminate state C.

- There is a path going from state B to state q_f via state C.
- So, after eliminating state C, we put a direct path from state B to state q_f having cost $b.b^*.\epsilon = bb^*$

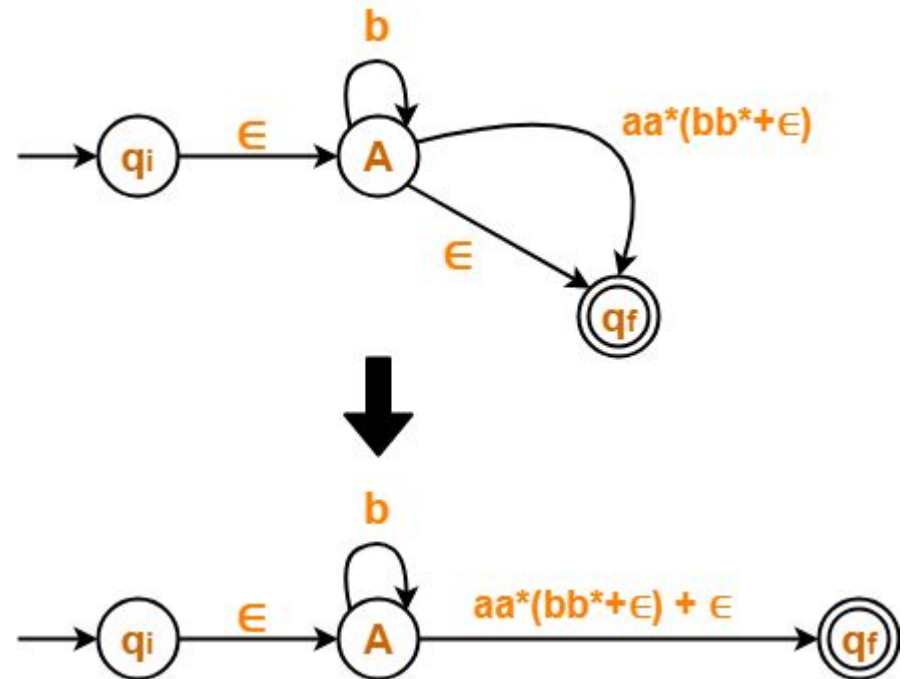
Eliminating state C, we get-



Example 04 (Cont.)

Step-05: Now, let us eliminate state B.

- There is a path going from state A to state q_f via state B.
- So, after eliminating state B, we put a direct path from state A to state q_f having cost $a.a^*. (bb^* + \epsilon)$
 $= aa^*(bb^* + \epsilon)$



- Eliminating state B, we get-

Example 04 (Cont.)

Step-06: Now, let us eliminate state A.

- There is a path going from state q_i to state q_f via state A.
- So, after eliminating state A, we put a direct path from state q_i to state q_f having cost $\in .b^*.(aa^*(bb^*+\in)+\in) = b^*(aa^*(bb^*+\in)+\in)$

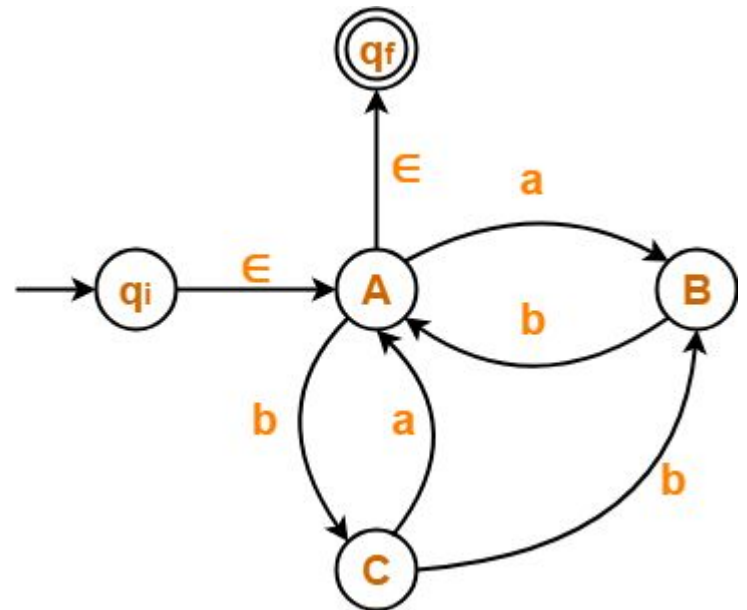
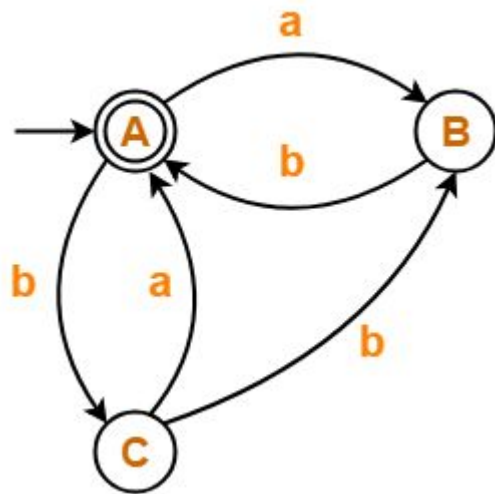
- Eliminating state A, we get-



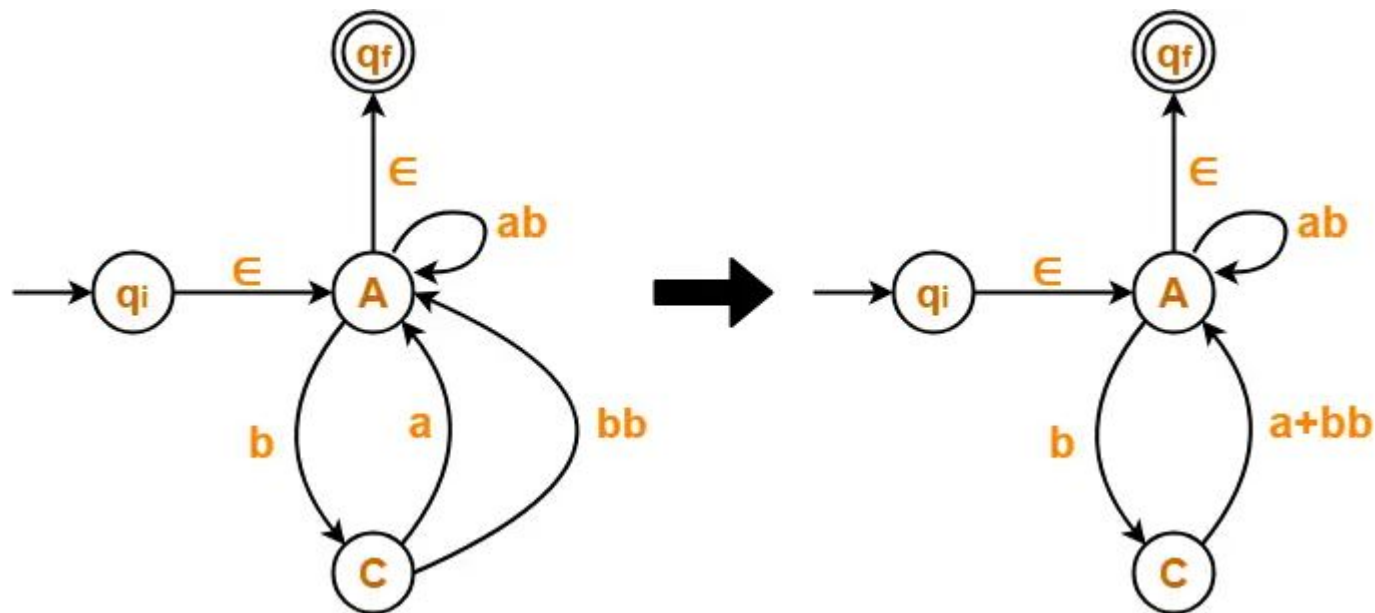
Regular Expression = $b^*(aa^*(bb^*+\in)+\in)$

- We know, $bb^* + \in = b^*$, So, we can also write-
- Regular Expression = $b^*(aa^*b^*+\in)$**

Example 05

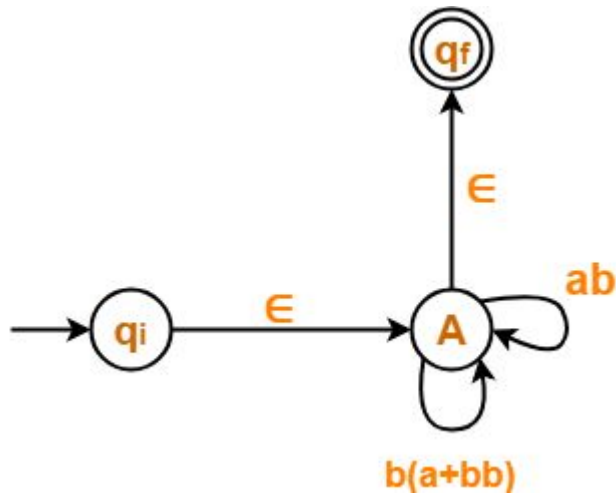


Example 05 (Cont.)



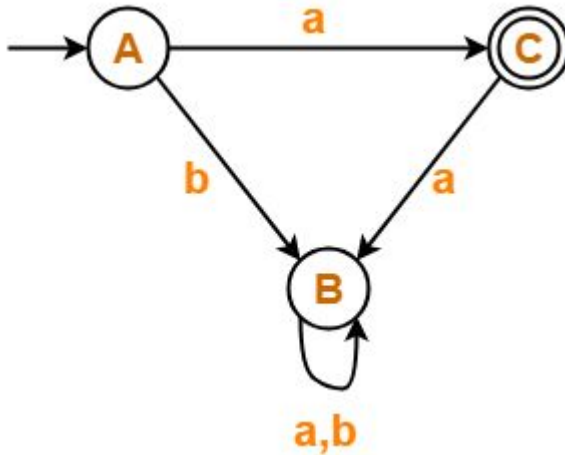
Example 05 (Cont.)

- Regular Expression = $(ab + b(a+bb))^*$

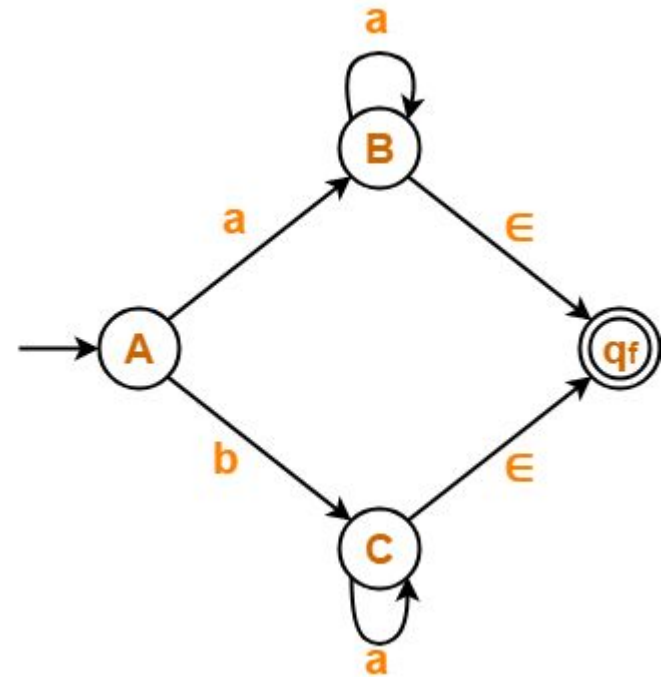
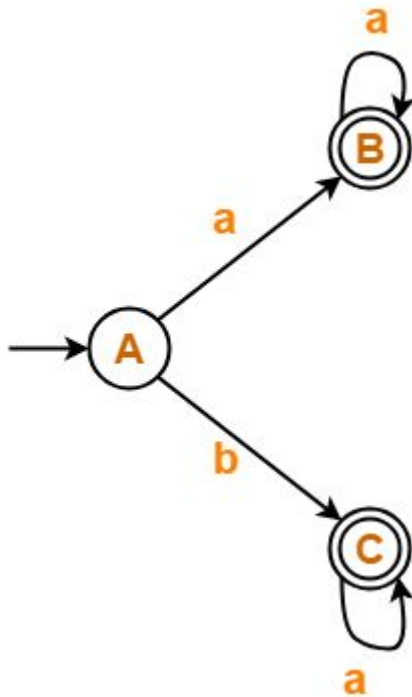


Example 06

- Regular Expression = a

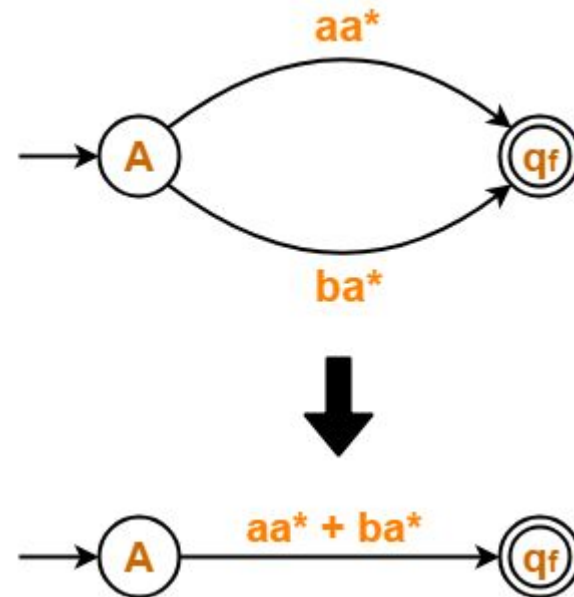
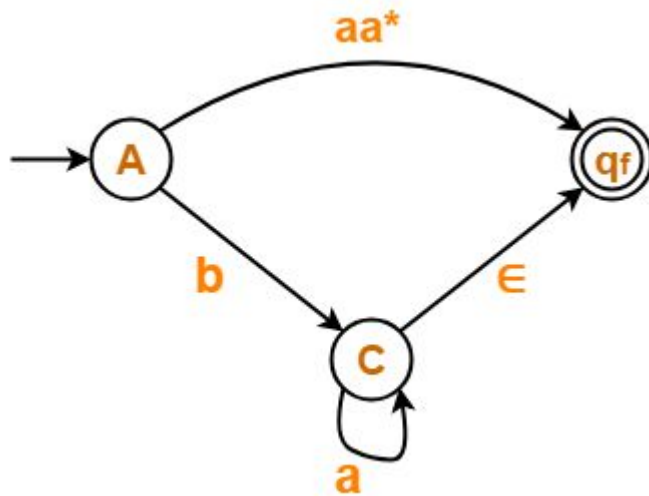


Example 07



Example 07

- Regular Expression = $aa^* + ba^*$



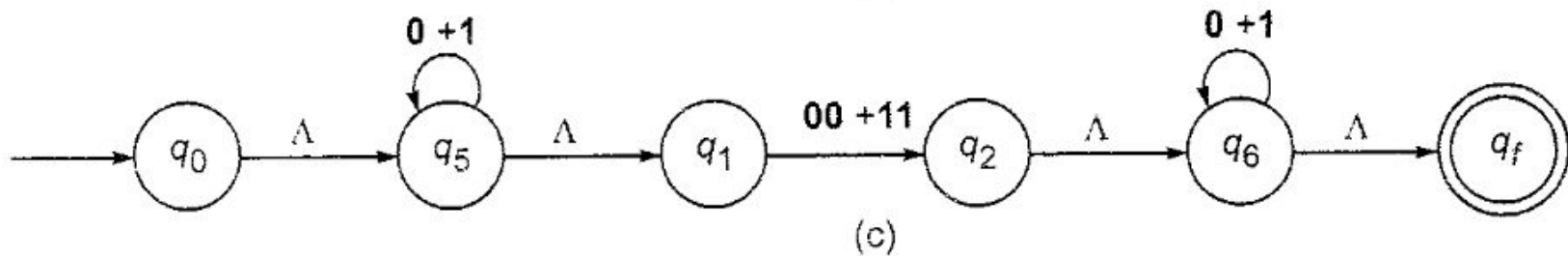
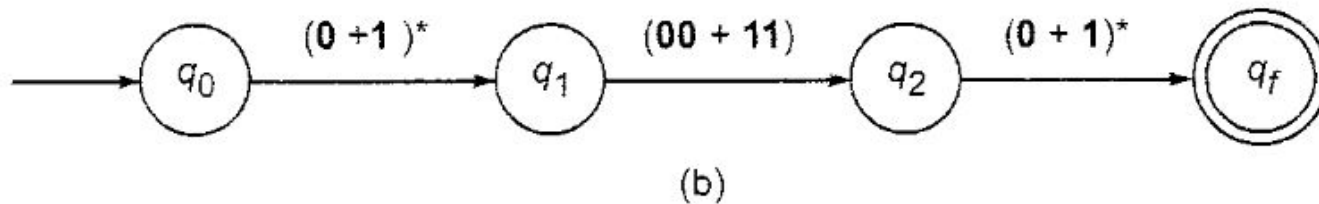
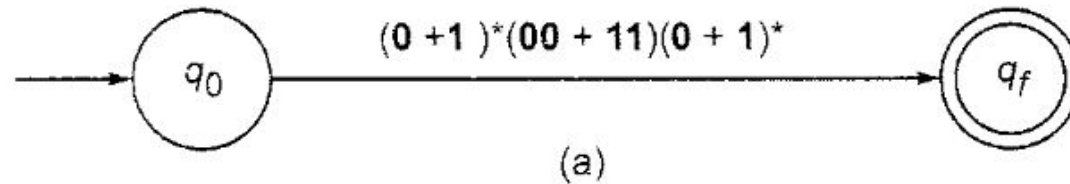
RE TO DFA

Dr. Kishore Kumar Sahu
Dept. of CSE, VSSUT, Burla

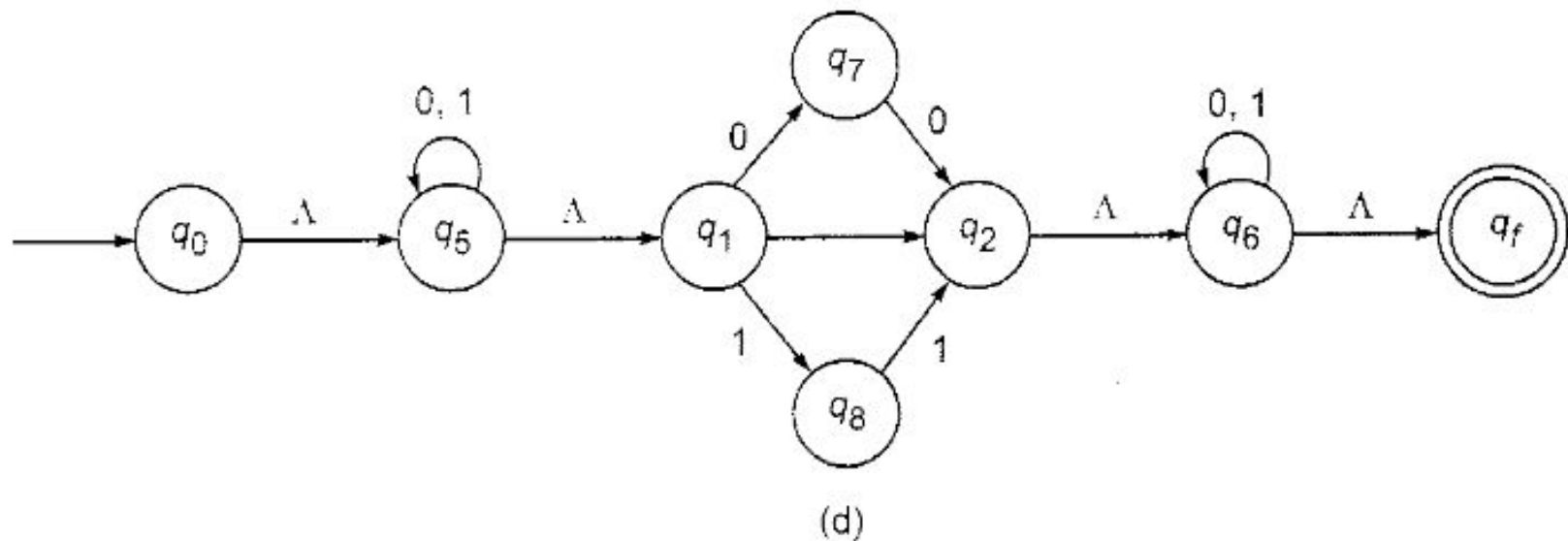
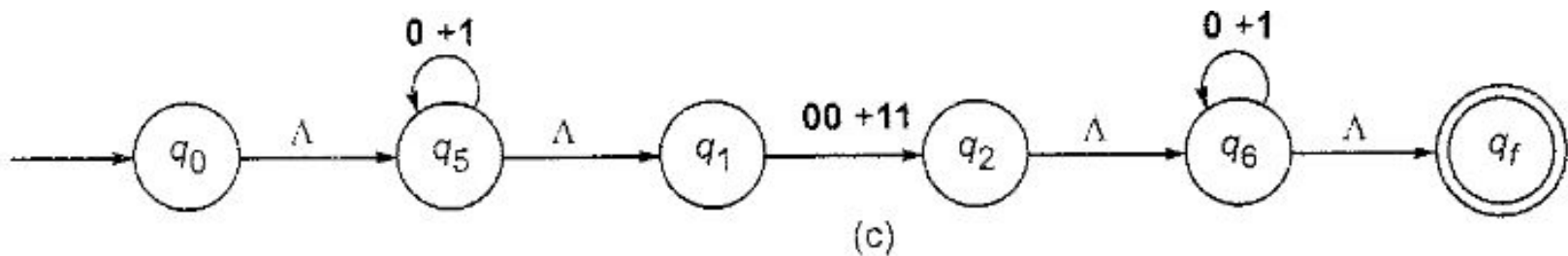
CONSTRUCTION OF FA TO RE

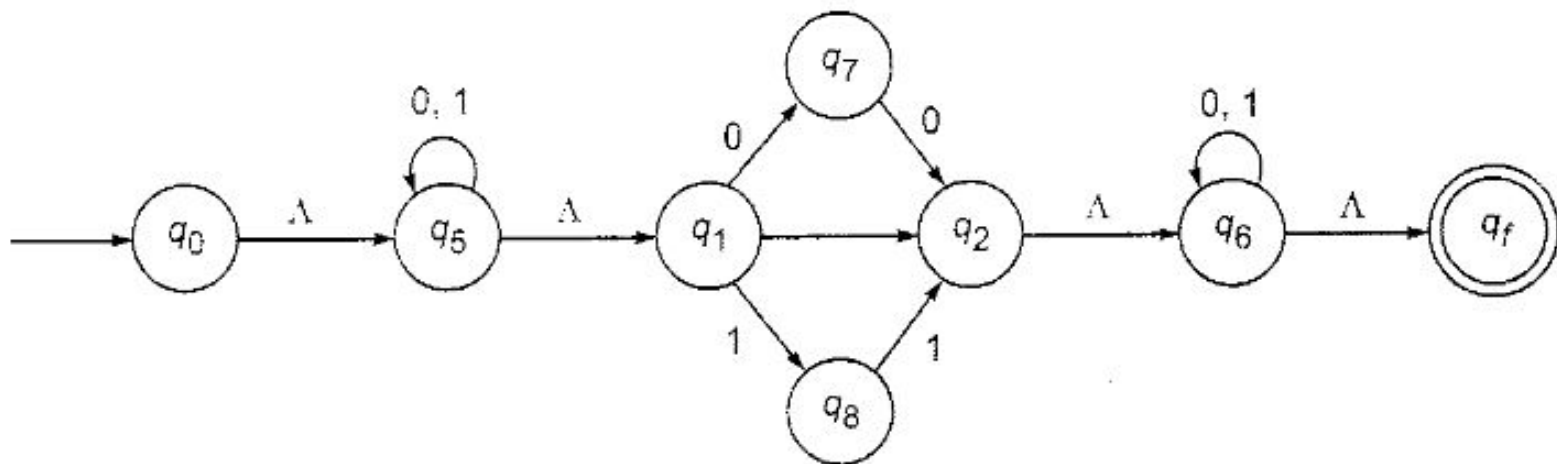
Construct the finite automaton equivalent to the regular expression

$$(0 + 1)^*(00 + 11)(0 + 1)^*$$

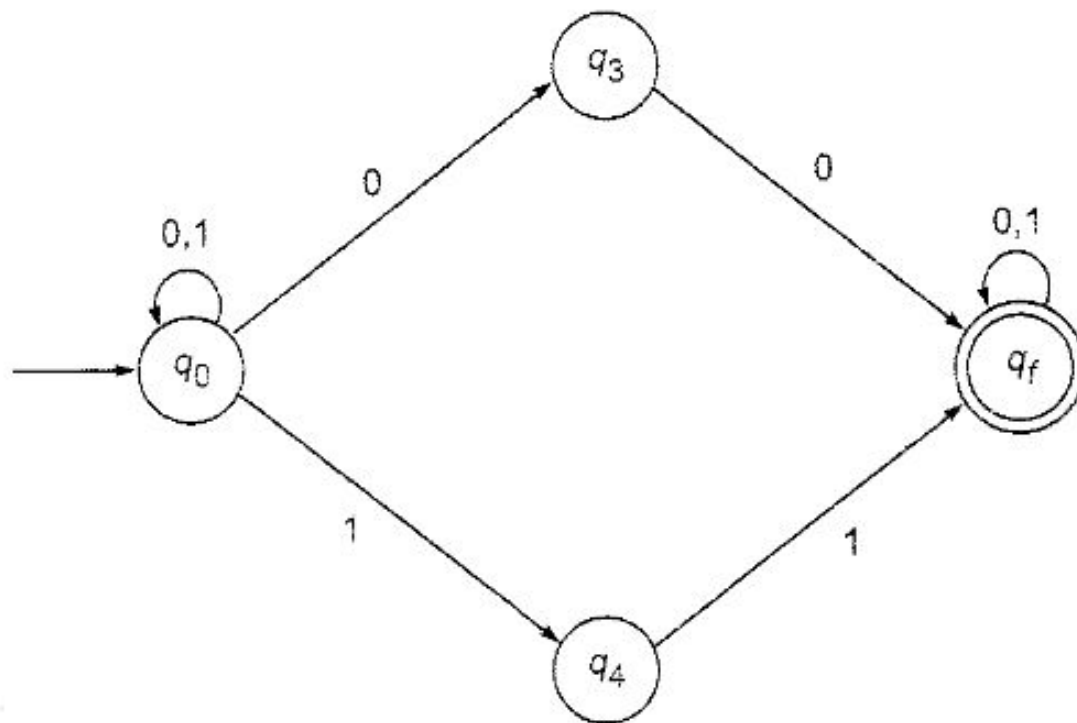


Example 01 (Cont.)



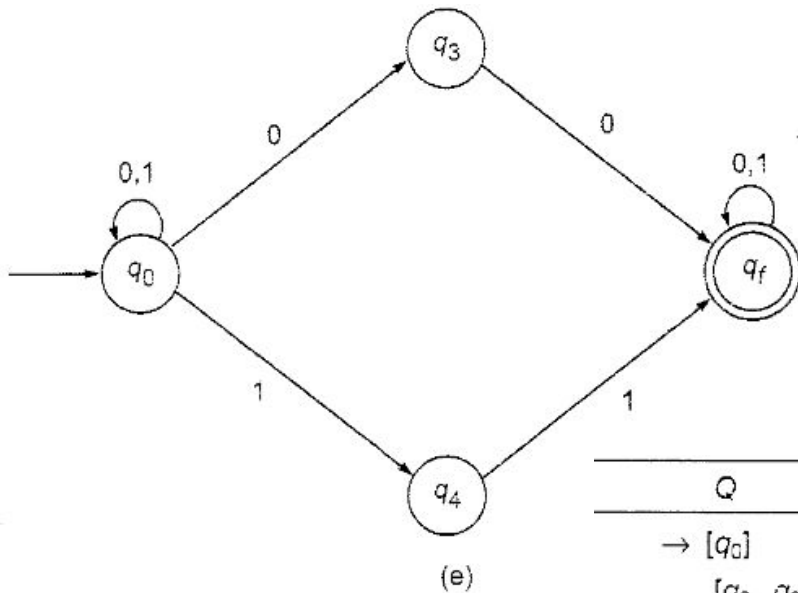


(d)



(e)

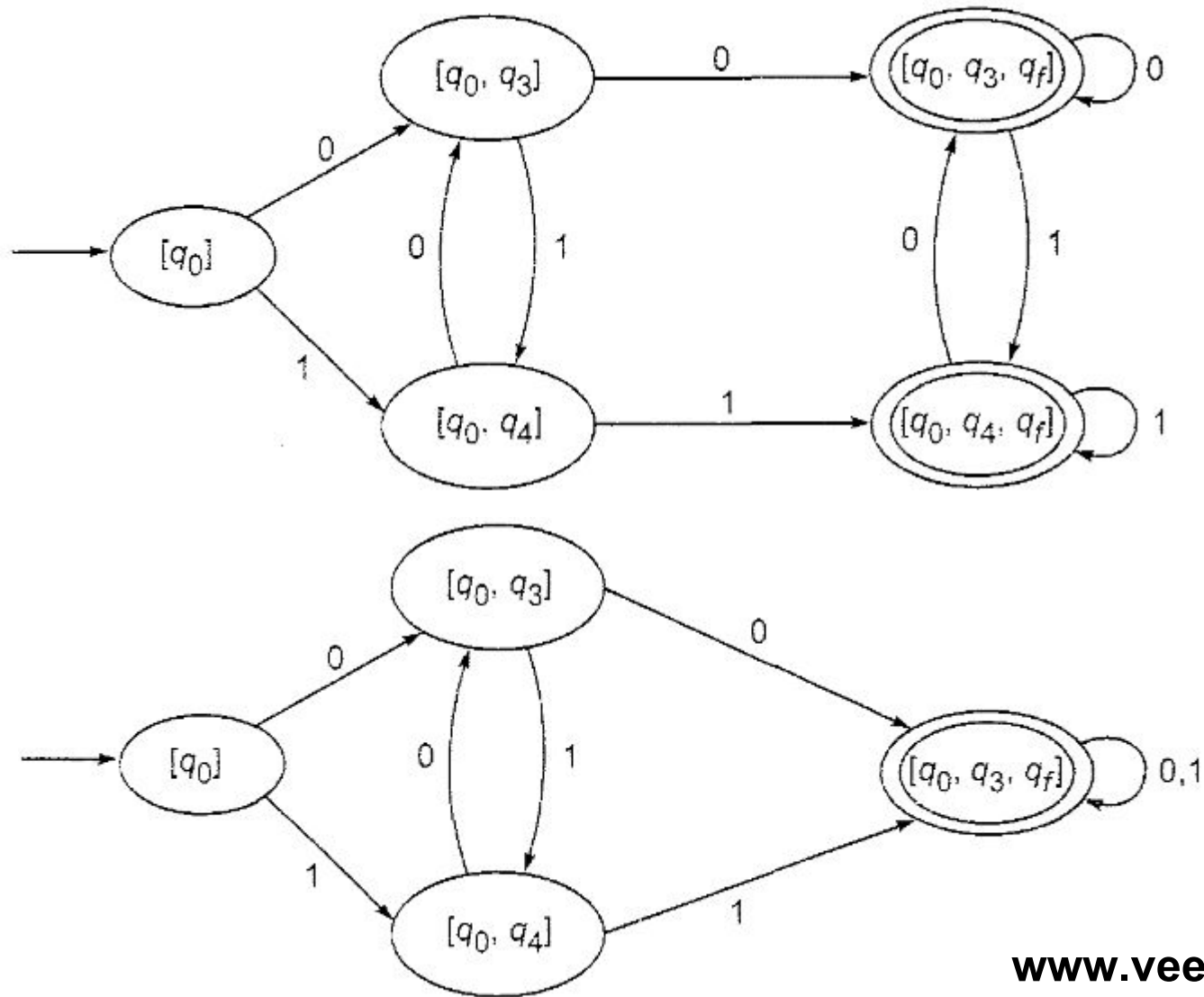
Example 01 (Cont.)



State/ Σ	0	1
$\rightarrow q_0$	q_0, q_3	q_0, q_4
q_3	q_f	
q_4		q_f
q_f	q_f	q_f

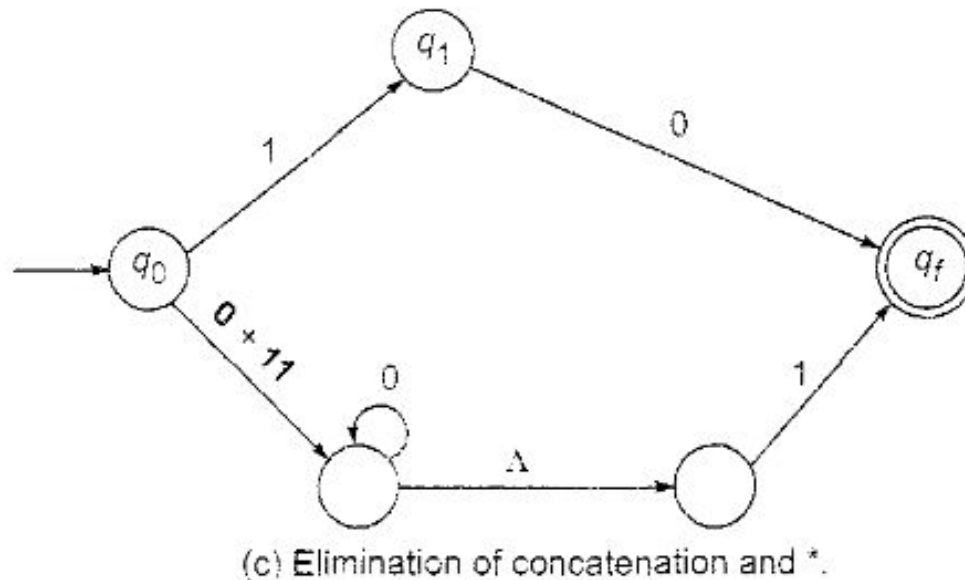
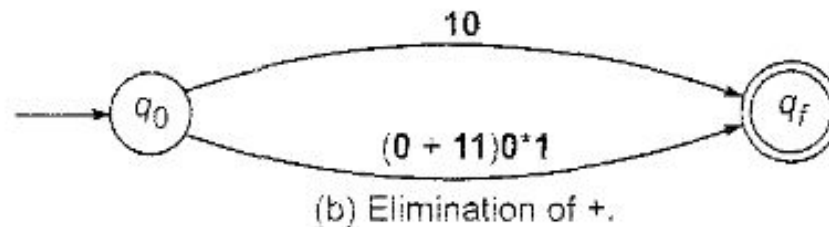
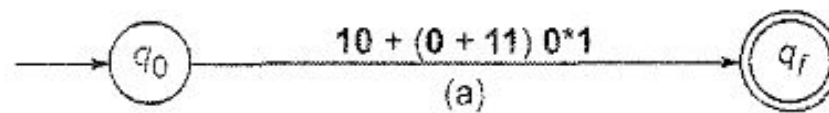
Q	Q_0	Q_1
$\rightarrow [q_0]$	$[q_0, q_3]$	$[q_0, q_4]$
$[q_0, q_3]$	$[q_0, q_3, q_f]$	$[q_0, q_4]$
$[q_0, q_4]$	$[q_0, q_3]$	$[q_0, q_4, q_f]$
$[q_0, q_3, q_f]$	$[q_0, q_3, q_f]$	$[q_0, q_4, q_f]$
$[q_0, q_4, q_f]$	$[q_0, q_3, q_f]$	$[q_0, q_4, q_f]$

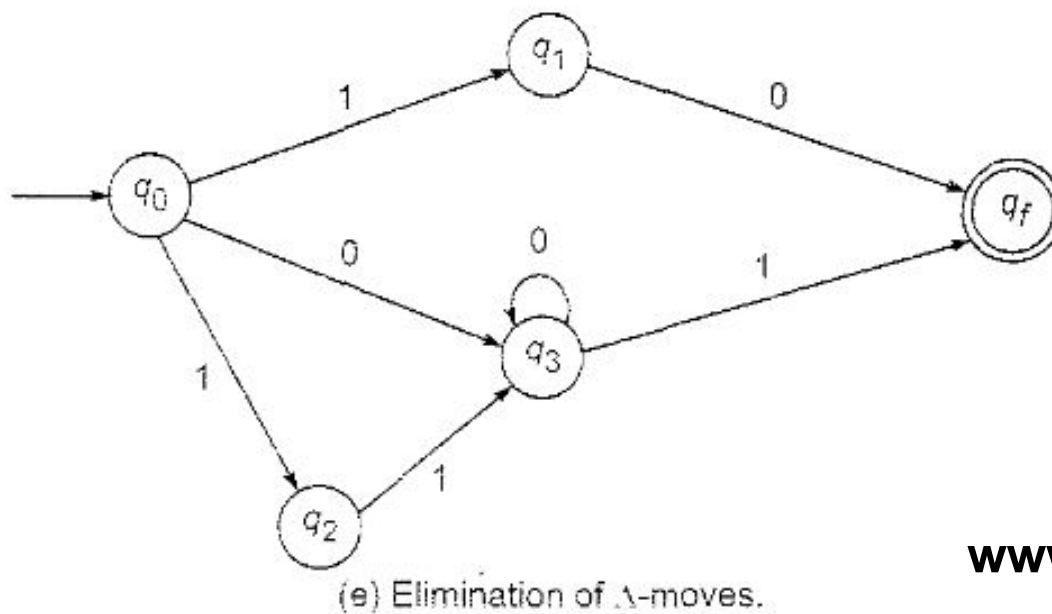
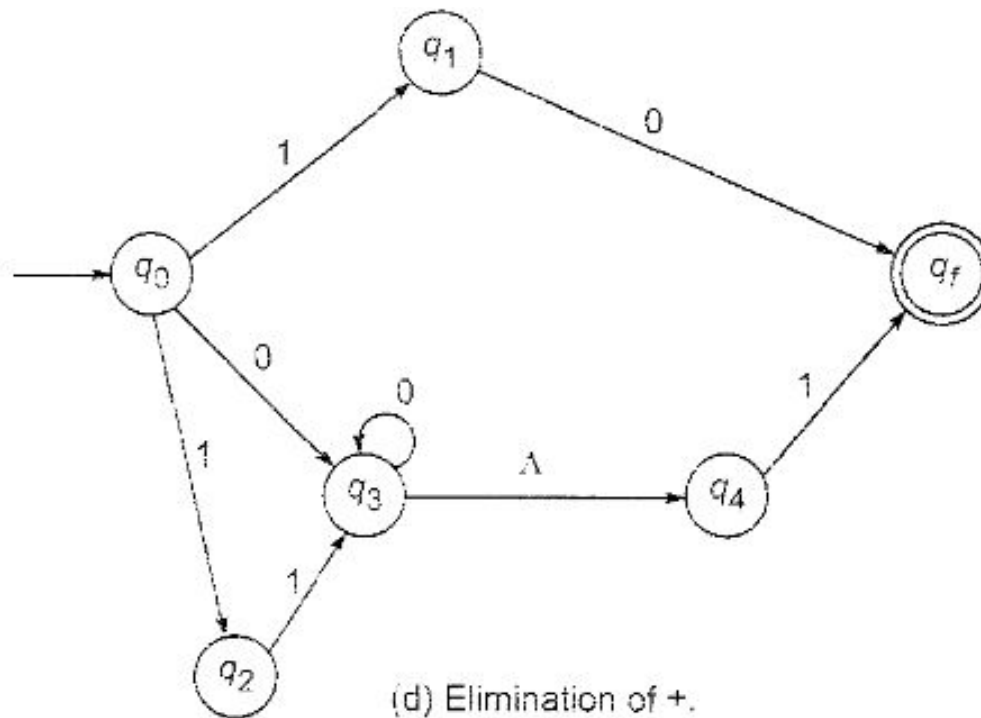
Example 01 (Cont.)



Example 02

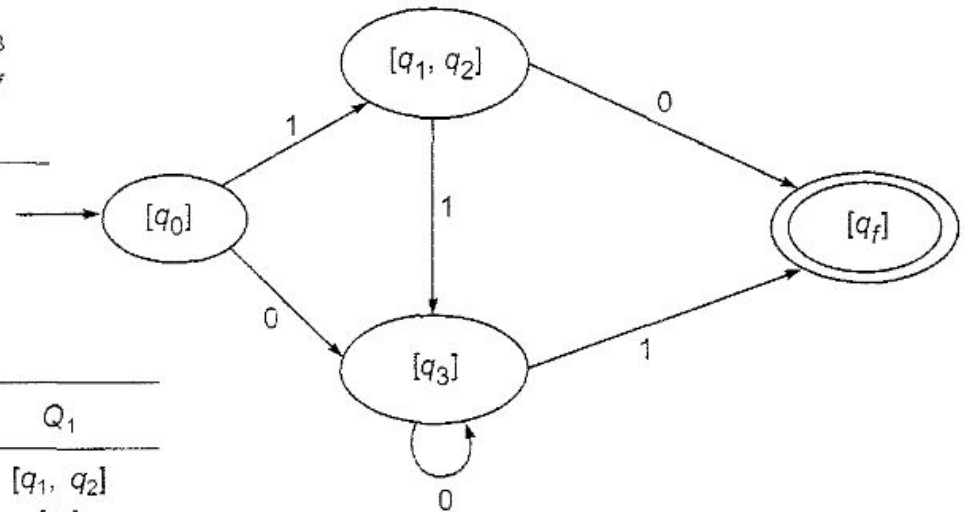
Construct a DFA with reduced states equivalent to the r.e. $10 + (0 + 11)0^*1$.





Example 02 (Cont.)

State/ Σ	0	1
$\rightarrow q_0$	q_3	q_1, q_2
q_1	q_f	
q_2		q_3
q_3	q_3	q_f
q_f		



Q	Q_0	Q_1
$\rightarrow [q_0]$	$[q_3]$	$[q_1, q_2]$
$[q_3]$	$[q_3]$	$[q_f]$
$[q_1, q_2]$	$[q_f]$	$[q_3]$
$[q_f]$	\emptyset	\emptyset
\emptyset	\emptyset	\emptyset



Properties of Regular Languages

Reading: Chapter 4



Topics

- 1) How to prove whether a given language is regular or not?
- 2) Closure properties of regular languages
- 3) Minimization of DFAs



Some languages are *not* regular

When is a language is regular?

if we are able to construct one of the following: DFA *or* NFA *or* ϵ -NFA *or* regular expression

When is it not?

If we can show that no FA can be built for a language



How to prove languages are *not* regular?

What if we cannot come up with any FA?

A) Can it be language that is not regular?

B) Or is it that we tried wrong approaches?

How do we *decisively* prove that a language is not regular?

“The hardest thing of all is to find a black cat in a dark room, especially if there is no cat!” -Confucius


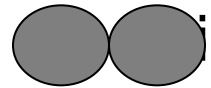


Example of a non-regular language

Let $L = \{w \mid w \text{ is of the form } 0^n 1^n, \text{ for all } n \geq 0\}$

- Hypothesis: L is not regular
- Intuitive rationale: How do you keep track of a running count in an FA?
- A more formal rationale:
 - By contradiction, if L is regular then there should exist a DFA for L .
 - Let k = number of states in that DFA.
 - Consider the special word $w = 0^k 1^k \Rightarrow w \in L$
 - DFA is in some state p_i , after consuming the first i symbols in w

Rationale...

- Let $\{p_0, p_1, \dots, p_k\}$ be the sequence of states that the DFA should have visited after consuming the first k symbols in w which is 0^k
- But there are only k states in the DFA!
- \implies at least one state should repeat somewhere along the path (by  +  principle)
- \implies Let the repeating state be $p_i = p_j$ for $i < j$
- \implies We can fool the DFA by inputting $0^{(k-(j-i))}1^k$ and still get it to accept (note: $k-(j-i)$ is at most $k-1$).
- \implies DFA accepts strings w with unequal number of 0s and 1s, implying that the DFA is wrong!



The Pumping Lemma for Regular Languages

What it is?

The Pumping Lemma is a property of all regular languages.

How is it used?

A technique that is used to show that a given language is not regular



Pumping Lemma for Regular Languages

Let L be a regular language

Then there exists some constant N such that for every string $w \in L$ s.t. $|w| \geq N$, there exists a way to break w into three parts, $w = xyz$, such that:

1. $y \neq \varepsilon$
2. $|xy| \leq N$
3. For all $k \geq 0$, all strings of the form $xy^kz \in L$

This property should hold for all regular languages.

Definition: N is called the “Pumping Lemma Constant”



Pumping Lemma: Proof

- L is regular \Rightarrow it should have a DFA.
 - Set $N :=$ number of states in the DFA
- Any string $w \in L$, s.t. $|w| \geq N$, should have the form: $w = a_1 a_2 \dots a_m$, where $m \geq N$
- Let the states traversed after reading the first N symbols be: $\{p_0, p_1, \dots, p_N\}$
 - \Rightarrow There are $N+1$ p -states, while there are only N DFA states
 - \Rightarrow at least one state has to repeat
i.e, $p_i = p_j$ where $0 \leq i < j \leq N$ (by PHP)

Pumping Lemma: Proof...

□ \Rightarrow We should be able to break $w = \mathbf{x} \mathbf{y} \mathbf{z}$ as follows:

□ $\mathbf{x} = a_1 a_2 \dots a_i$; $\mathbf{y} = a_{i+1} a_{i+2} \dots a_j$; $\mathbf{z} = a_{j+1} a_{j+2} \dots a_m$

□ \mathbf{x} 's path will be $p_0 \dots p_i$

□ \mathbf{y} 's path will be $p_i p_{i+1} \dots p_j$ (but $p_i = p_j$ implying a loop)

□ \mathbf{z} 's path will be $p_j p_{j+1} \dots p_m$

□ Now consider another string $w_k = \mathbf{x} \mathbf{y}^k \mathbf{z}$, where $k \geq 0$

□ Case $k=0$

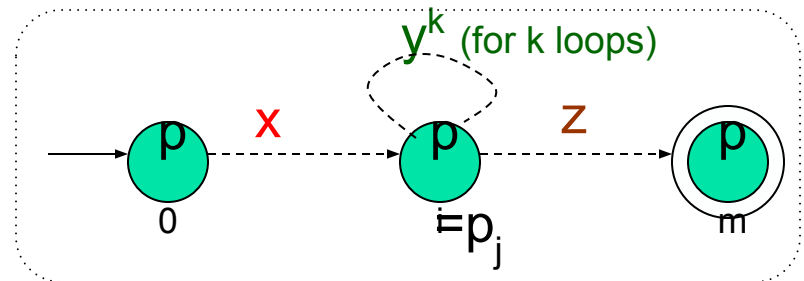
□ DFA will reach the accept state p_m

□ Case $k > 0$

□ DFA will loop for \mathbf{y}^k , and finally reach the accept state p_m for \mathbf{z}

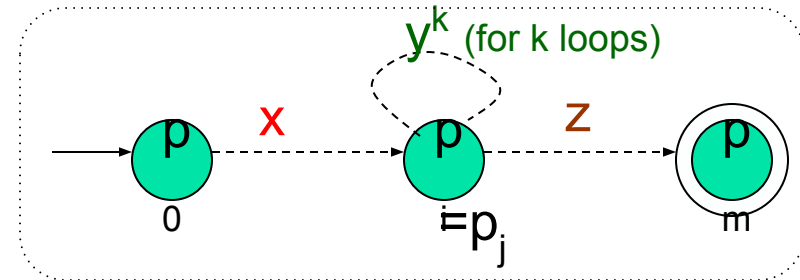
□ In either case, $w_k \in L$

This proves part (3) of the lemma



Pumping Lemma: Proof...

- For part (1):
 - Since $i < j$, $y \neq \varepsilon$



- For part (2):
 - By PHP, the repetition of states has to occur within the first N symbols in w
 - $\implies |xy| \leq N$

□



The Purpose of the Pumping Lemma for RL

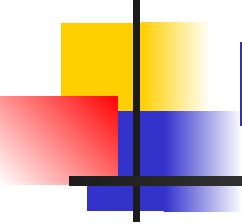
- To prove that some languages *cannot* be regular.



How to use the pumping lemma?

Think of playing a 2 person game

- Role 1: **We** claim that the language cannot be regular
- Role 2: An **adversary** who claims the language is regular
- We show that the adversary's statement will lead to a contradiction that implies pumping lemma *cannot* hold for the language.
- We win!!



How to use the pumping lemma? (The Steps)

1. (we) L is not regular.
 2. (adv.) Claims that L is regular and gives you a value for N as its P/L constant
 3. (we) Using N , choose a string $w \in L$ s.t.,
 1. $|w| \geq N$,
 2. Using w as the template, construct other words w_k of the form xy^kz and show that at least one such $w_k \notin L$
 \Rightarrow this implies we have successfully broken the pumping lemma for the language, and hence that the adversary is wrong.
- (Note: In this process, we may have to try many values of k , starting with $k=0$, and then 2, 3, .. so on, until $w_k \notin L$.)

Note: We don't have any control over N , except that it is positive.

We also don't have any control over how to split $w=xyz$,
but xyz should respect the P/L conditions (1) and (2).



Using the Pumping Lemma

- What WE do?
 3. Using N , we construct our template string w
 4. Demonstrate to the adversary, either through pumping up or down on w , that some string $w_k \notin L$
(this should happen regardless of $w=xyz$)
- What the Adversary does?
 1. Claims L is regular
 2. Provides N

Note: This N can be anything (need not necessarily be the #states in the DFA.
It's the adversary's choice.)

Example of using the Pumping Lemma to prove that a language is not regular

Let $L_{eq} = \{w \mid w \text{ is a binary string with equal number of 1s and 0s}\}$

- Your Claim: L_{eq} is not regular

- Proof:

- By contradiction, let L_{eq} be regular

□ adv.

- P/L constant should exist

□ adv.

- Let $N =$ that P/L constant

- Consider input $w = 0^N 1^N$

□ you

(your choice for the template string)

- By pumping lemma, we should be able to break $w = xyz$, such that:

□ you


- 1) $y \neq \epsilon$

- 2) $|xy| \leq N$

- 3) For all $k \geq 0$, the string xy^kz is also in L

Template string $w = 0^N 1^N = \underbrace{00}_{\cdot \ddot{N} \cdot} \underbrace{011}_{\ddot{N} \cdot} 1$

Proof...

- Because $|xy| \leq N$, xy should contain only 0s
 - (This and because $y \neq \varepsilon$, implies $y = 0^+$)
- Therefore x can contain *at most* $N-1$ 0s
- Also, all the N 1s must be inside z
- By (3), any string of the form $xy^kz \in L_{eq}$ for all $k \geq 0$
- Case $k=0$: xz has at most $N-1$ 0s but has N 1s
- Therefore, $xy^0z \notin L_{eq}$
- This violates the P/L (a contradiction) 

□ you

Setting $k=0$ is referred to as "pumping down"

Setting $k>1$ is referred to as "pumping up"

Another way of proving this will be to show that if the #0s is arbitrarily pumped up (e.g., $k=2$), then the #0s will become exceed the #1s



Exercise 2

Prove $L = \{0^n 1 0^n \mid n \geq 1\}$ is not regular

Note: This n is not to be confused with the pumping lemma constant N . That *can* be different.

In other words, the above question is same as proving:

- $L = \{0^m 1 0^m \mid m \geq 1\}$ is not regular



Example 3: Pumping Lemma

Claim: $L = \{ 0^i \mid i \text{ is a perfect square} \}$ is not regular

■ Proof:

- By contradiction, let L be regular.
- P/L should apply
- Let $N = P/L$ constant
- Choose $w = 0^{N^2}$
- By pumping lemma, $w = xyz$ satisfying all three rules
- By rules (1) & (2), y has between 1 and N 0s
- By rule (3), any string of the form xy^kz is also in L for all $k \geq 0$
- Case $k=0$:
 - $\#zeros(xy^0z) = \#zeros(xyz) - \#zeros(y)$
 - $N^2 - N \leq \#zeros(xy^0z) \leq N^2 - 1$
 - $(N-1)^2 < N^2 - N \leq \#zeros(xy^0z) \leq N^2 - 1 < N^2$
 - $xy^0z \notin L$
 - But the above will complete the proof ONLY IF $N > 1$.
 - ... (proof contd.. Next slide)



Example 3: Pumping Lemma

- (proof contd...)
 - If the adversary pick $N=1$, then $(N-1)^2 \leq N^2 - N$, and therefore the #zeros(xy^0z) could end up being a perfect square!
 - This means that pumping down (i.e., setting $k=0$) is not giving us the proof!
 - So lets try pumping up next...
- Case $k=2$:
 - #zeros (xy^2z) = #zeros (xyz) + #zeros (y)
 - $N^2 + 1 \leq \text{\#zeros}(xy^2z) \leq N^2 + N$
 - $N^2 < N^2 + 1 \leq \text{\#zeros}(xy^2z) \leq N^2 + N < (N+1)^2$
 - $xy^2z \notin L$
- (Notice that the above should hold for all possible N values of $N>0$. Therefore, this completes the proof.)

Closure properties of Regular Languages



Closure properties for Regular Languages (RL)

This is different from Kleene closure

- Closure property:
 - If a set of regular languages are combined using an operator, then the resulting language is also regular
- Regular languages are closed under:
 - Union, intersection, complement, difference
 - Reversal
 - Kleene closure
 - Concatenation
 - Homomorphism
 - Inverse homomorphism

Now, let's prove all of this!



RLs are closed under union

- IF L and M are two RLs THEN:
 - they both have two corresponding regular expressions, R and S respectively
 - $(L \cup M)$ can be represented using the regular expression $R+S$
 - Therefore, $(L \cup M)$ is also regular

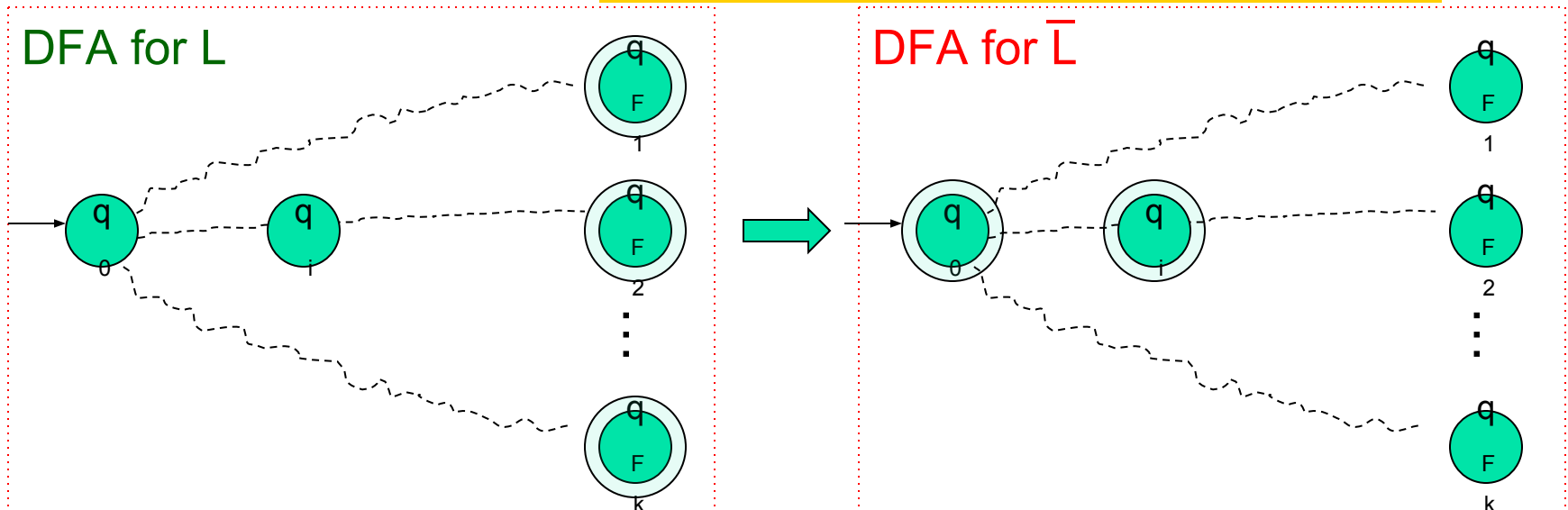


How can this be proved using FAs?

RLs are closed under complementation

- If L is an RL over Σ , then $\bar{L} = \Sigma^* - L$
- To show \bar{L} is also regular, make the following construction

Convert every final state into non-final, and every non-final state into a final state



Assumes q_0 is a non-final state. If not, do the opposite.



RLs are closed under intersection

- A quick, indirect way to prove:
 - By DeMorgan's law:
 - $L \cap M = \overline{(\overline{L} \cup \overline{M})}$
 - Since we know RLs are closed under union and complementation, they are also closed under intersection
- A more direct way would be construct a finite automaton for $L \cap M$

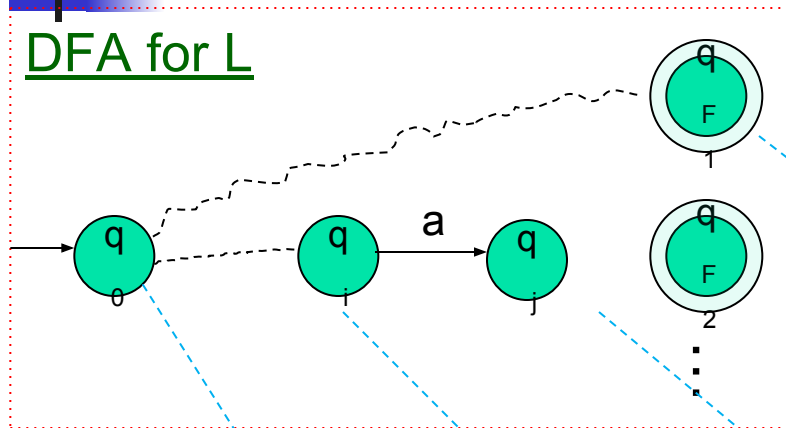


DFA construction for $L \cap M$

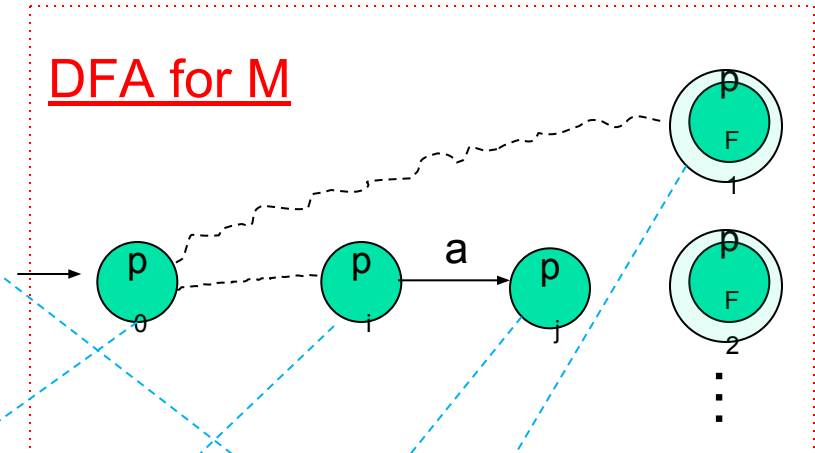
- $A_L = \text{DFA for } L = \{Q_L, \Sigma, q_L, F_L, \delta_L\}$
- $A_M = \text{DFA for } M = \{Q_M, \Sigma, q_M, F_M, \delta_M\}$
- Build $A_{L \cap M} = \{Q_L \times Q_M, \Sigma, (q_L, q_M), F_L \times F_M, \delta\}$ such that:
 - $\delta((p, q), a) = (\delta_L(p, a), \delta_M(q, a))$, where p in Q_L , and q in Q_M
- This construction ensures that a string w will be accepted if and only if w reaches an accepting state in both input DFAs.

DFA construction for $L \cap M$

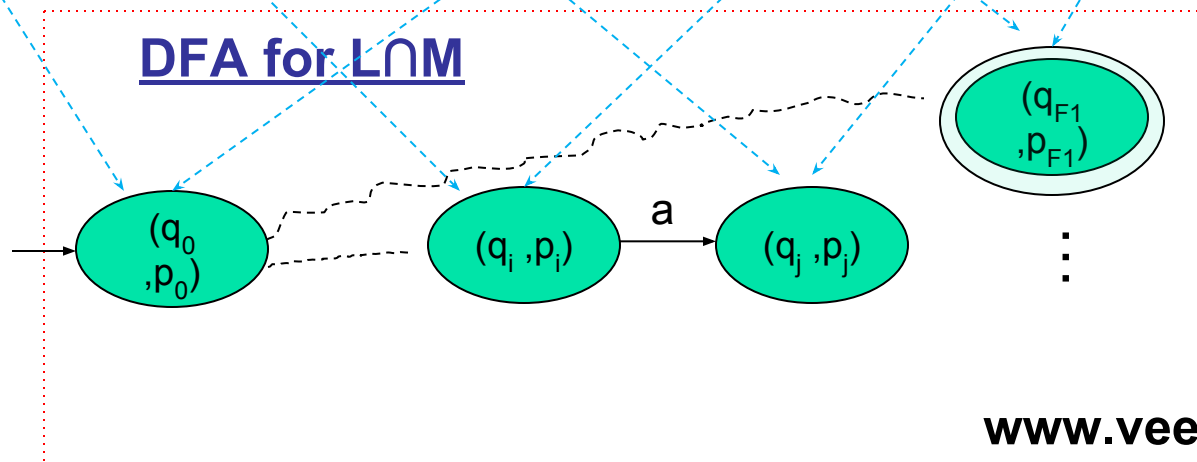
DFA for L



DFA for M



DFA for $L \cap M$



RLs are closed under set difference

- We observe:

- $L - M = L \cap \overline{M}$

Closed under intersection

Closed under
complementation

- Therefore, $L - M$ is also regular



RLs are closed under reversal

Reversal of a string w is denoted by w^R

- E.g., $w=00111$, $w^R=11100$

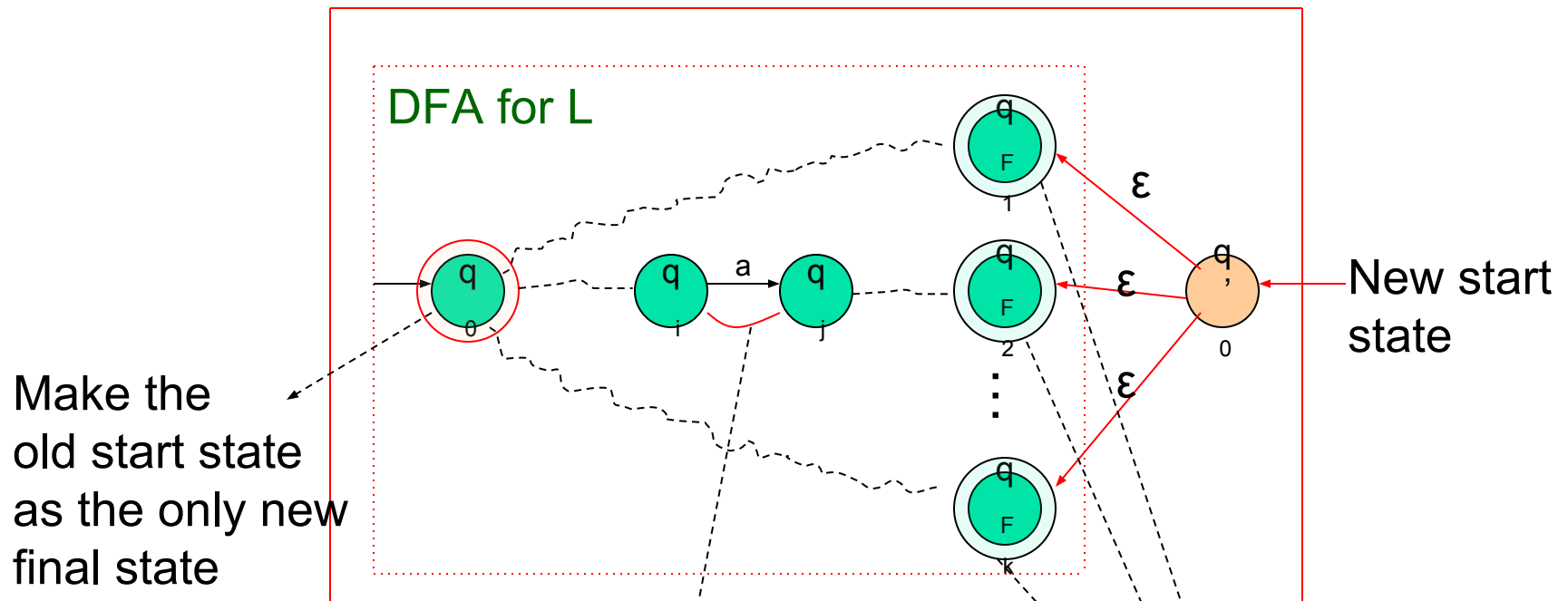
Reversal of a language:

- L^R = The language generated by reversing all strings in L

Theorem: If L is regular then L^R is also regular

ϵ -NFA Construction for L^R

New ϵ -NFA for L^R



What to do if q_0 was one of the final states in the input DFA?

Reverse all transitions

Convert the old set of final states into non-final states



If L is regular, L^R is regular (proof using regular expressions)

- Let E be a regular expression for L
- Given E , how to build E^R ?
- Basis: If $E = \varepsilon$, \emptyset , or a , then $E^R = E$
- Induction: Every part of E (refer to the part as “ F ”) can be in only *one* of the three following forms:
 1. $F = F_1 + F_2$
 - $F^R = F_1^R + F_2^R$
 2. $F = F_1 F_2$
 - $F^R = F_2^R F_1^R$
 3. $F = (F_1)^*$
 - $(F^R)^* = (F_1^R)^*$



Homomorphisms

- Substitute each symbol in Σ (main alphabet) by a corresponding string in T (another alphabet)
 - $h: \Sigma \rightarrow T^*$
- Example:
 - Let $\Sigma = \{0, 1\}$ and $T = \{a, b\}$
 - Let a homomorphic function h on Σ be:
 - $h(0) = ab, h(1) = \epsilon$
 - If $w = 10110$, then $h(w) = \epsilon ab \epsilon \epsilon ab = abab$
- In general,
 - $h(w) = h(a_1) h(a_2) \dots h(a_n)$



RLs are closed under homomorphisms

- Theorem: If L is regular, then so is $h(L)$
- Proof: If E is a RE for L , then show $L(h(E)) = h(L(E))$
- Basis: If $E = \varepsilon, \emptyset$, or a , then the claim holds.
- Induction: There are three forms of E :

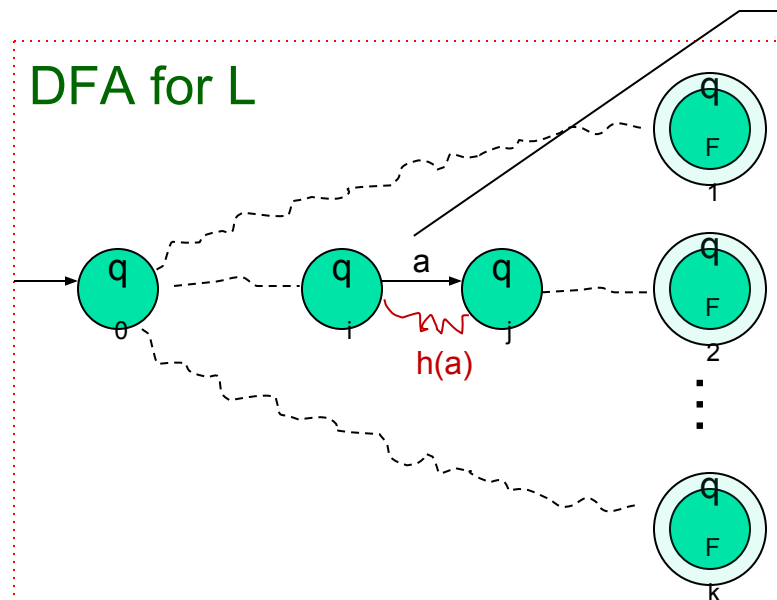
1. $E = E_1 + E_2$
 - $L(h(E)) = L(h(E_1) + h(E_2)) = L(h(E_1)) \cup L(h(E_2)) \text{ ----- (1)}$
 - $h(L(E)) = h(L(E_1) + L(E_2)) = h(L(E_1)) \cup h(L(E_2)) \text{ ----- (2)}$
 - By inductive hypothesis, $L(h(E_1)) = h(L(E_1))$ and $L(h(E_2)) = h(L(E_2))$
 - Therefore, $L(h(E)) = h(L(E))$

2. $E = E_1 E_2$
 3. $E = (E_1)^*$
- } Similar argument

Think of a DFA based construction

Given a DFA for L , how to convert it into an FA for $h(L)$?

FA Construction for $h(L)$



Replace every edge “ a ” by a path labeled $h(a)$ in the new DFA

- Build a new FA that simulates $h(a)$ for every symbol a transition in the above DFA
- The resulting FA may or may not be a DFA, but will be a FA for $h(L)$

Given a DFA for M , how to convert it into an FA for $h^{-1}(M)$?

The set of strings in Σ^* whose homomorphic translation results in the strings of M

Inverse homomorphism

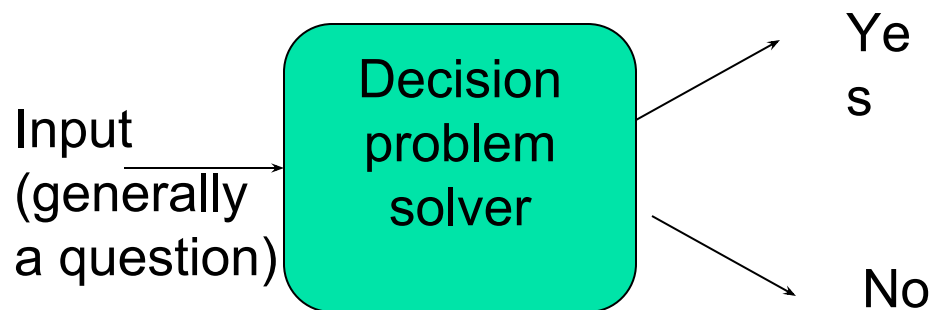
- Let $h: \Sigma \rightarrow T^*$
- Let M be a language over alphabet T
- $h^{-1}(M) = \{w \mid w \in \Sigma^* \text{ s.t., } h(w) \in M\}$

Claim: If M is regular, then so is $h^{-1}(M)$

- Proof:
 - Let A be a DFA for M
 - Construct another DFA A' which encodes $h^{-1}(M)$
 - A' is an exact replica of A , except that its transition functions are s.t. for any input symbol a in Σ , A' will simulate $h(a)$ in A .
 - $\delta'(p, a) = \delta(\hat{p}, h(a))$

Decision properties of regular languages

Any “decision problem” looks like this:





Membership question

- Decision Problem: Given L , is w in L ?
- Possible answers: Yes or No
- Approach:
 1. Build a DFA for L
 2. Input w to the DFA
 3. If the DFA ends in an accepting state, then yes; otherwise no.



Emptiness test

- Decision Problem: Is $L = \emptyset$?
- Approach:
On a DFA for L:
 1. From the start state, run a *reachability* test, which returns:
 1. success: if there is at least one final state that is reachable from the start state
 2. failure: otherwise
 2. $L = \emptyset$ if and only if the reachability test fails

How to implement the reachability test?



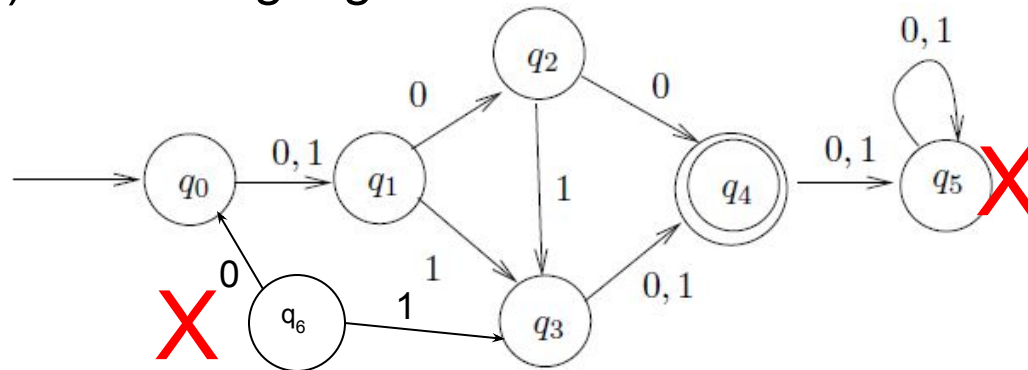
Finiteness

- Decision Problem: Is L finite or infinite?
- Approach:
 - On a DFA for L:
 1. Remove all states unreachable from the start state
 2. Remove all states that cannot lead to any accepting state.
 3. After removal, check for cycles in the resulting FA
 4. L is finite if there are no cycles; otherwise it is infinite
- Another approach
 - Build a regular expression and look for Kleene closure

How to implement steps 2 and 3?

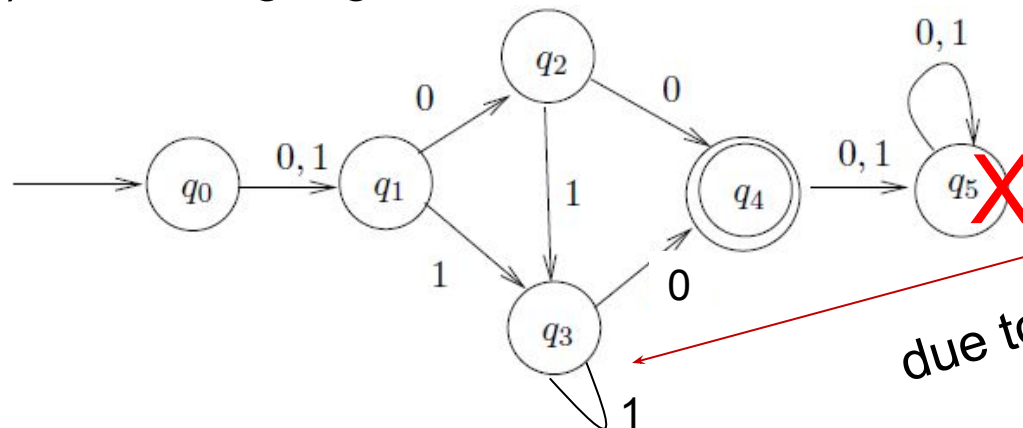
Finiteness test - examples

Ex 1) Is the language of this DFA finite or infinite?



FINITE

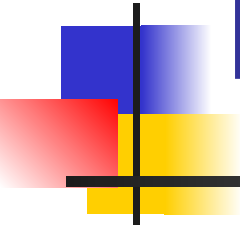
Ex 2) Is the language of this DFA finite or infinite?



INFINITE

due to this

Equivalence & Minimization of DFAs





Applications of interest

- Comparing two DFAs:
 - $L(\text{DFA}_1) == L(\text{DFA}_2)$?

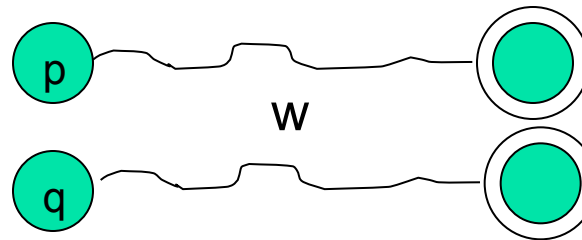
- How to minimize a DFA?
 1. Remove unreachable states
 2. Identify & condense equivalent states into one

When to call two states in a DFA “equivalent”?

Past doesn't matter - only future does!

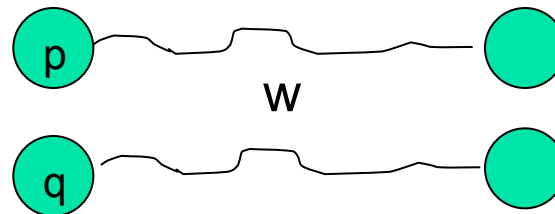
Two states p and q are said to be *equivalent* iff:

- i) Any string w accepted by starting at p is also accepted by starting at q ;



AND

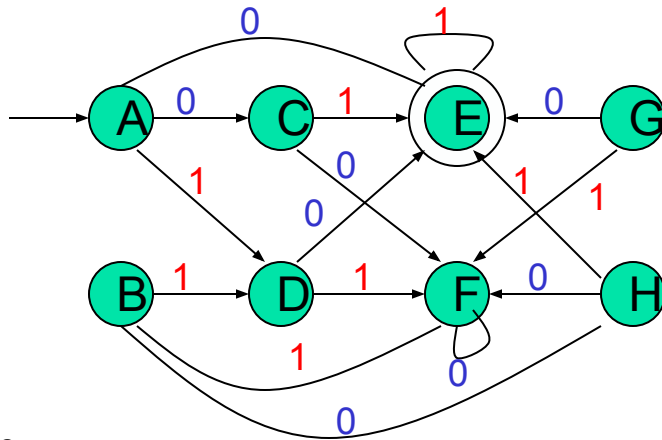
- ii) Any string w rejected by starting at p is also rejected by starting at q .



$\square p \equiv q$

Computing equivalent states in a DFA

Table Filling Algorithm



Pass #0

1. Mark accepting states \neq non-accepting states

Pass #1

2. Compare every pair of states
3. Distinguish by one symbol transition
4. Mark = or \neq or blank(tbd)

Pass #2

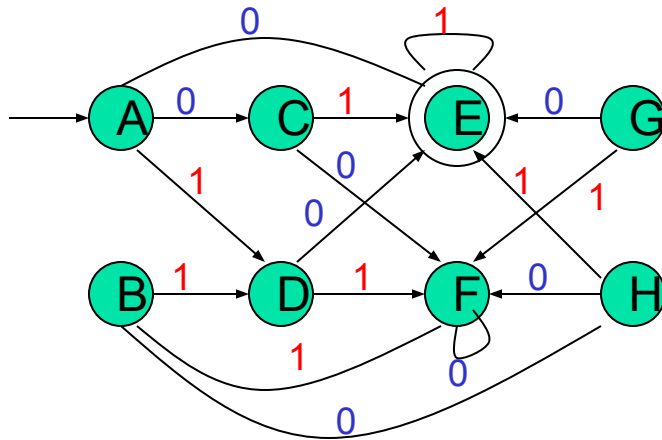
5. Compare every pair of states
6. Distinguish by up to two symbol transitions (until different or same or tbd)

....

(keep repeating until table complete)

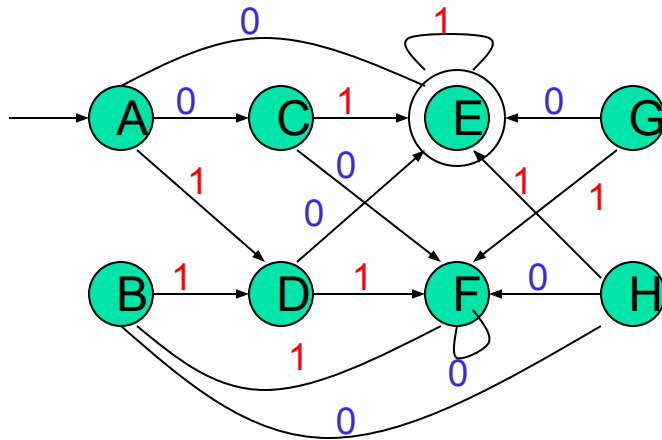
A	=							
B	=	=						
C	x	x	=					
D	x	x	x	=				
E	x	x	x	x	=			
F	x	x	x	x	x	=		
G	x	x	x	=	x	x	=	
H	x	x	=	x	x	x	x	=
	A	B	C	D	E	F	G	H

Table Filling Algorithm - step by step



A	=							
B		=						
C			=					
D				=				
E					=			
F						=		
G							=	
H								=
	A	B	C	D	E	F	G	H

Table Filling Algorithm - step by step

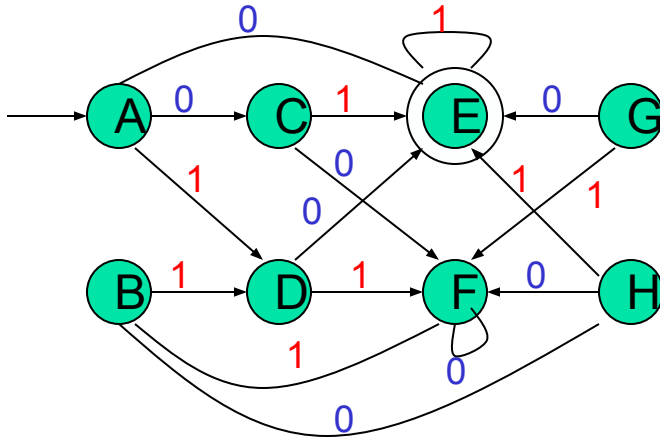


1. Mark **X** between accepting vs. non-accepting state



A	=							
B		=						
C			=					
D				=				
E	X	X	X	X	=			
F					X	=		
G					X		=	
H					X			=
	A	B	C	D	E	F	G	H

Table Filling Algorithm - step by step

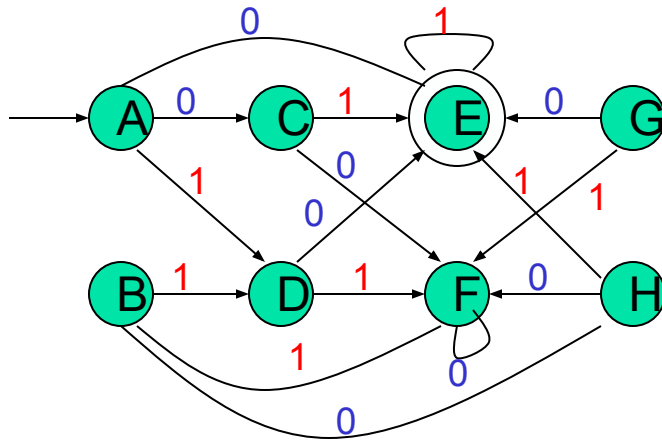


1. Mark **X** between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

A	=							
B		=						
C	X		=					
D	X			=				
E	X	X	X	X	=			
F					X	=		
G	X				X		=	
H	X				X			=
	A	B	C	D	E	F	G	H

↑

Table Filling Algorithm - step by step

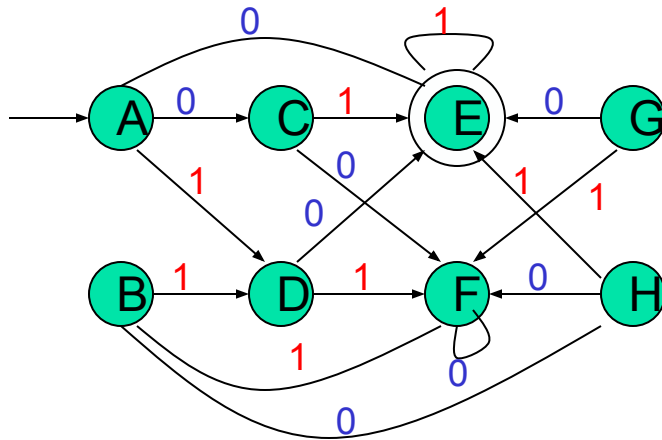


1. Mark **X** between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

A	=							
B		=						
C	X	X	=					
D	X	X		=				
E	X	X	X	X	=			
F					X	=		
G	X	X			X		=	
H	X	X			X			=
	A	B	C	D	E	F	G	H

↑

Table Filling Algorithm - step by step

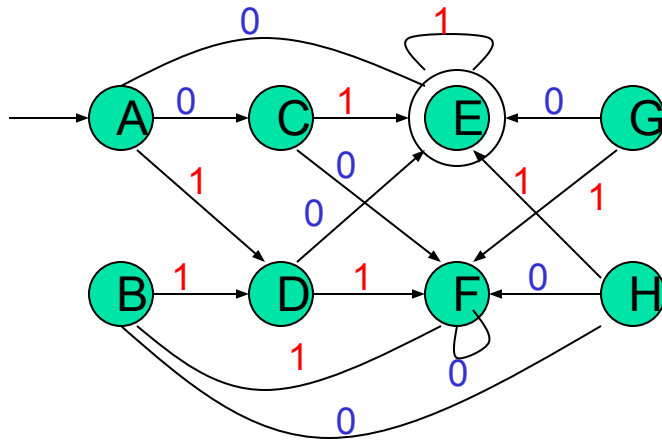


1. Mark **X** between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F			X		X	=		
G	X	X	X		X		=	
H	X	X	=		X			=
	A	B	C	D	E	F	G	H



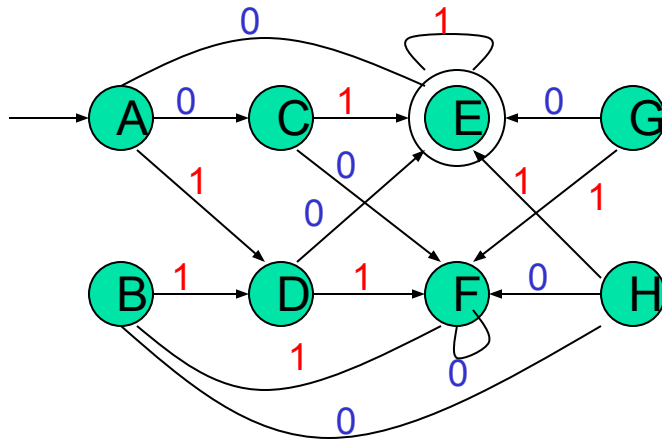
Table Filling Algorithm - step by step



1. Mark **X** between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F			X	X	X	=		
G	X	X	X	=	X		=	
H	X	X	=	X	X			=
	A	B	C	D	E	F	G	H

Table Filling Algorithm - step by step

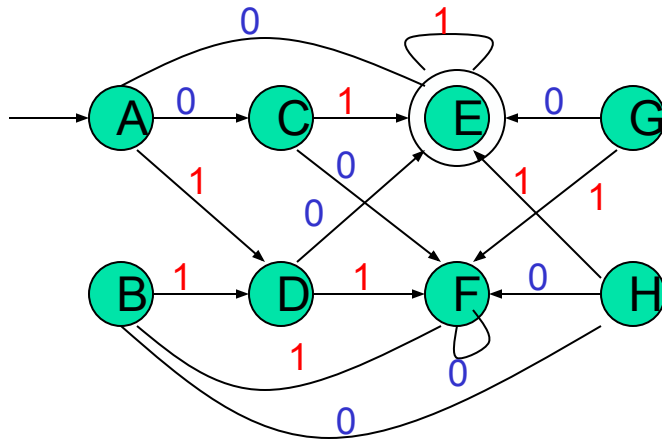


1. Mark **X** between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F			X	X	X	=		
G	X	X	X	=	X	X	=	
H	X	X	=	X	X	X		=
	A	B	C	D	E	F	G	H



Table Filling Algorithm - step by step

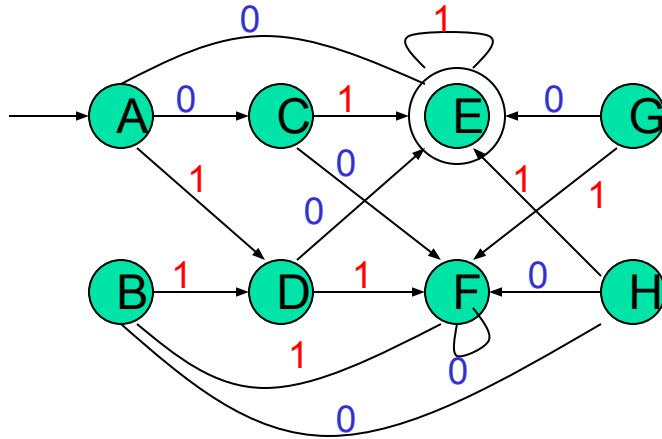


1. Mark **X** between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F			X	X	X	=		
G	X	X	X	=	X	X	=	
H	X	X	=	X	X	X	X	=
	A	B	C	D	E	F	G	H



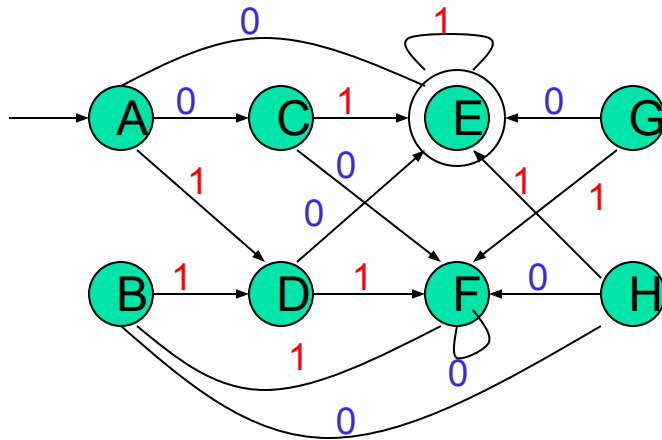
Table Filling Algorithm - step by step



A	=							
B	=	=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F	X	X	X	X	X	=		
G	X	X	X	=	X	X	=	
H	X	X	=	X	X	X	X	=
	A	B	C	D	E	F	G	H

1. Mark **X** between accepting vs. non-accepting state
2. Pass 1:
Look 1- hop away for distinguishing states or strings
3. Pass 2:
Look 1-hop away again for distinguishing states or strings
continue....

Table Filling Algorithm - step by step



A	=							
B	=	=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F	X	X	X	X	X	=		
G	X	X	X	=	X	X	=	
H	X	X	=	X	X	X	X	=
	A	B	C	D	E	F	G	H

1. Mark **X** between accepting vs. non-accepting state

2. Pass 1:

Look 1- hop away for distinguishing states or strings

3. Pass 2:

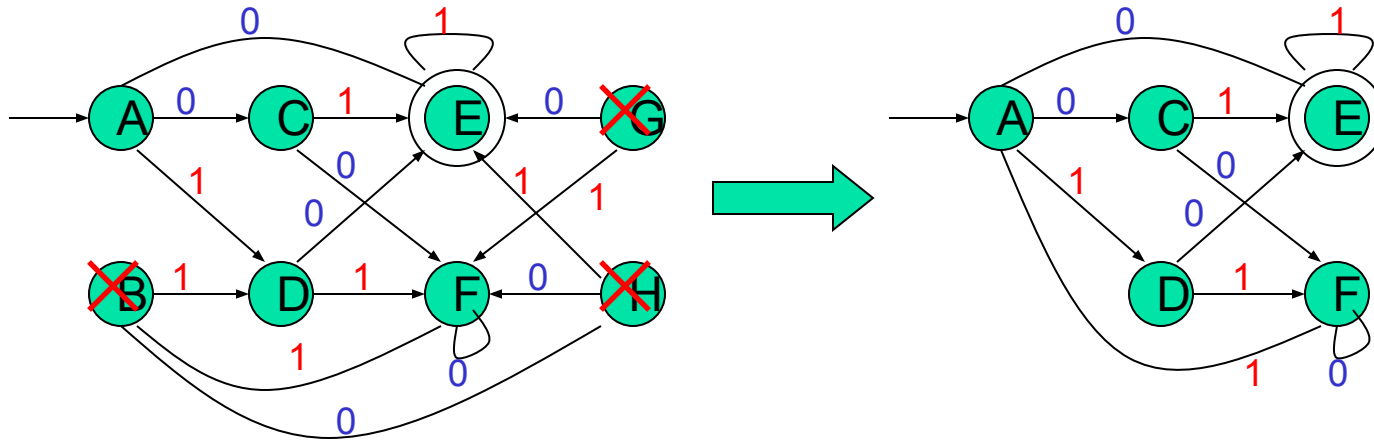
Look 1-hop away again for distinguishing states or strings

continue....

Equivalences:

- A=B
- C=H
- D=G

Table Filling Algorithm - step by step

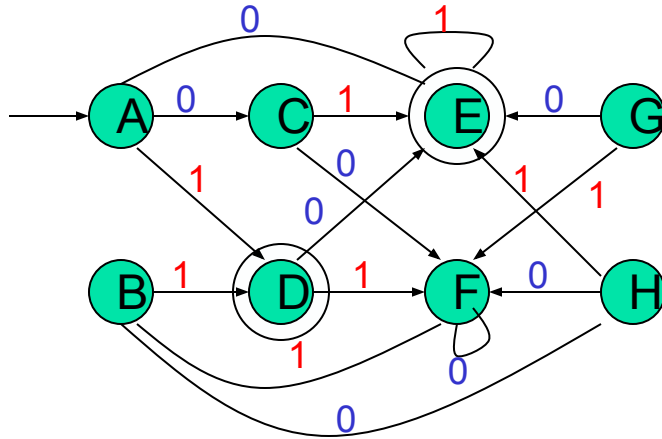


Retrain only one copy for
each equivalence set of states

Equivalences:

- A=B
- C=H
- D=G

Table Filling Algorithm – special case



A	=							
B		=						
C			=					
D				=				
E				?	=			
F						=		
G							=	
H								=
	A	B	C	D	E	F	G	H

Q) What happens if the input DFA has more than one final state?
Can all final states initially be treated as equivalent to one another?

Putting it all together ...



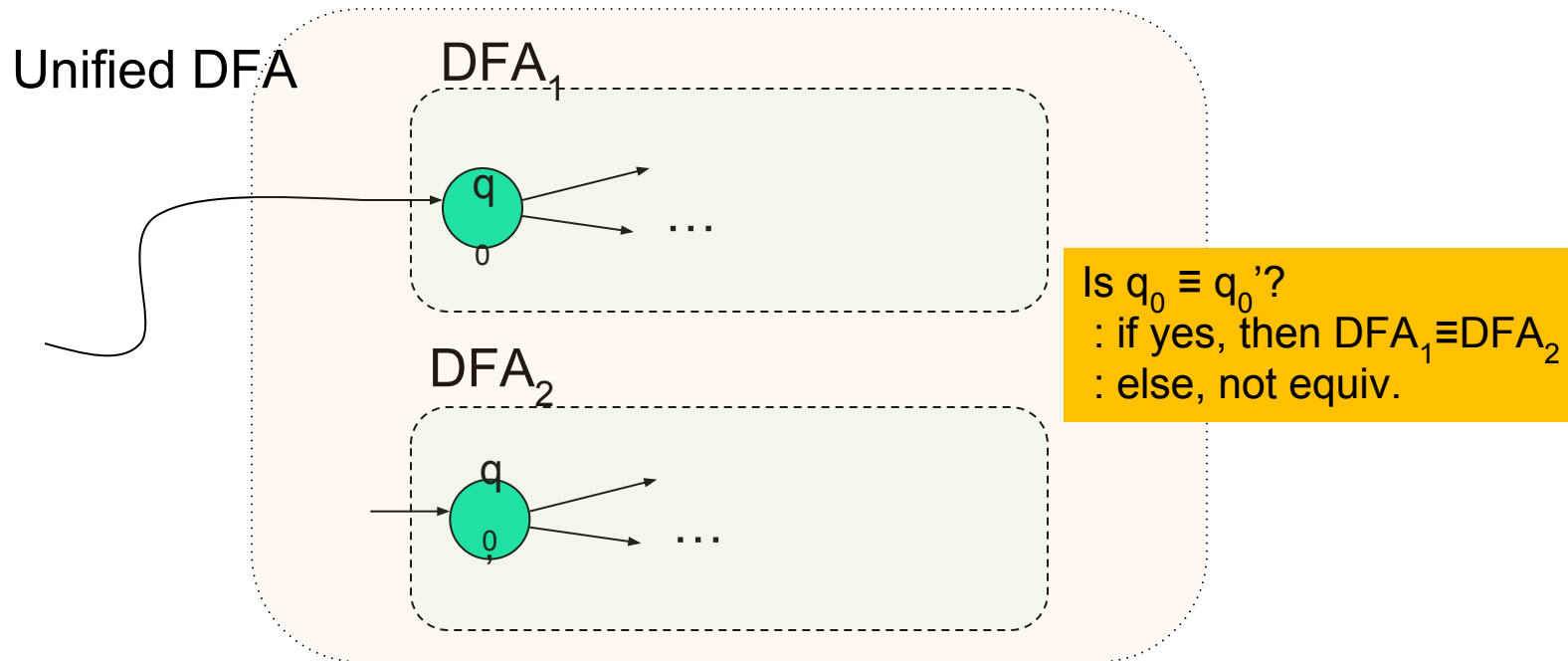
How to minimize a DFA?

- Goal: Minimize the number of states in a DFA
- Algorithm:
 - 1. Eliminate states unreachable from the start state
 - 2. Identify and remove equivalent states
 - 3. Output the resultant DFA

Depth-first traversal from the start state

Table filling algorithm

Are Two DFAs Equivalent?



1. Make a new dummy DFA by just putting together both DFAs
2. Run table-filling algorithm on the unified DFA
3. IF the start states of both DFAs are found to be equivalent,
 THEN: $DFA_1 \equiv DFA_2$
 ELSE: different



Summary

- How to prove languages are not regular?
 - Pumping lemma & its applications
- Closure properties of regular languages
- Simplification of DFAs
 - How to remove unreachable states?
 - How to identify and collapse equivalent states?
 - How to minimize a DFA?
 - How to tell whether two DFAs are equivalent?

Regular Properties and Regular Grammar

Equivalence of Two Finite Automata

Let M and M' be two finite automata over Σ . We construct a comparison table consisting of $n + 1$ columns, where n is the number of input symbols. The first column consists of pairs of vertices of the form (q, q') , where $q \in M$ and $q' \in M'$. If (q, q') appears in some row of the first column, then the corresponding entry in the a -column ($a \in \Sigma$) is (q_a, q'_a) , where q_a and q'_a are reachable from q and q' , respectively on application of a (i.e. by a -paths).

The comparison table is constructed by starting with the pair of initial vertices q_{in}, q'_{in} of M and M' in the first column. The first elements in the subsequent columns are (q_a, q'_a) , where q_a and q'_a are reachable by a -paths from q_{in} and q'_{in} . We repeat the construction by considering the pairs in the second and subsequent columns which are not in the first column.

The row-wise construction is repeated. There are two cases:

Case 1 If we reach a pair (q, q') such that q is a final state of M , and q' is a nonfinal state of M' or vice versa, we terminate the construction and conclude that M and M' are not equivalent.

Case 2 Here the construction is terminated when no new element appears in the second and subsequent columns which are not in the first column (i.e. when all the elements in the second and subsequent columns appear in the first column). In this case we conclude that M and M' are equivalent.

EXAMPLE 1

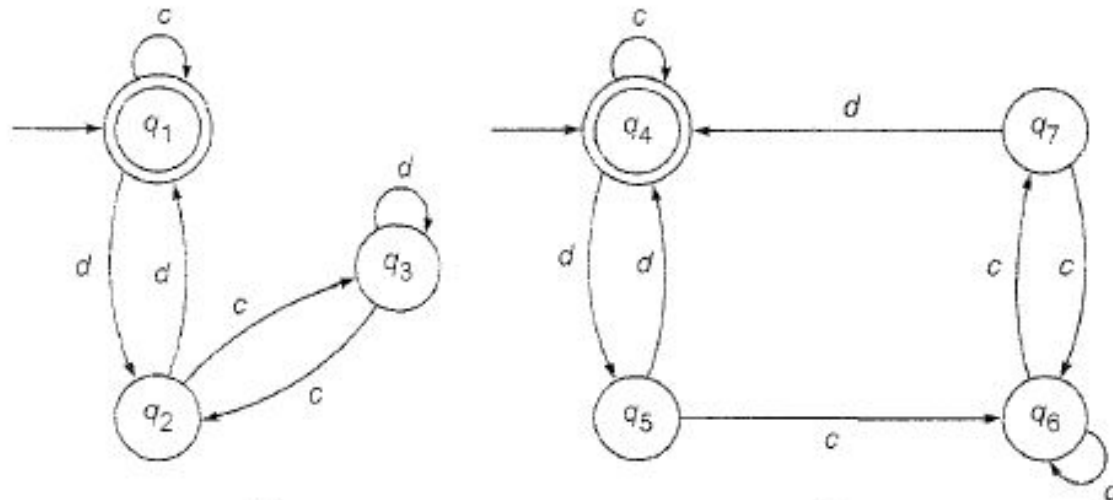


TABLE 5.7 Comparison Table for Example 5.15

(q, q')	(q_c, q'_c)	(q_d, q'_d)
(q_1, q_4)	(q_1, q_4)	(q_2, q_5)
(q_2, q_5)	(q_3, q_6)	(q_1, q_4)
(q_3, q_6)	(q_2, q_7)	(q_3, q_6)
(q_2, q_7)	(q_3, q_6)	(q_1, q_4)

EXAMPLE 2

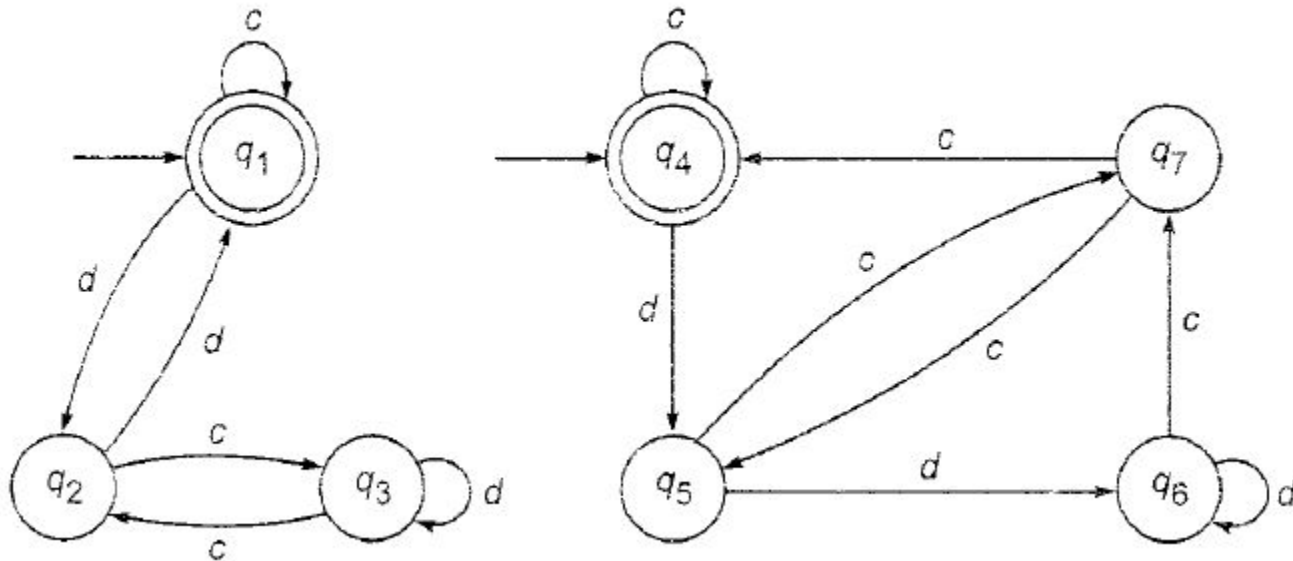
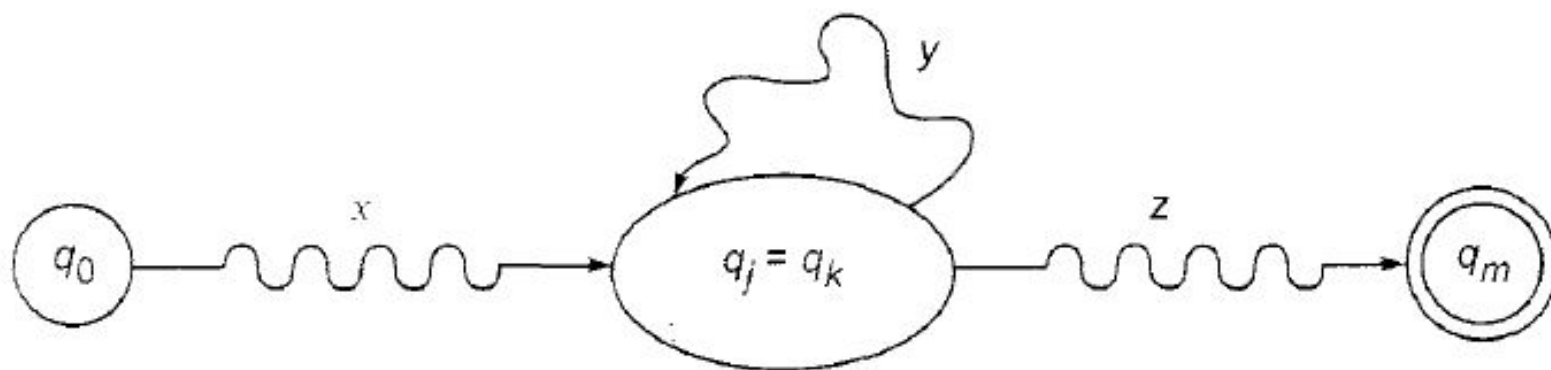


TABLE 5.8 Comparison Table for Example 5.16

(q, q')	(q_c, q'_c)	(q_d, q'_d)
(q_1, q_4)	(q_1, q_4)	(q_2, q_5)
(q_2, q_5)	(q_3, q_7)	(q_1, q_6)

Pumping Lemma for Regular Language

Theorem 5.5 (Pumping Lemma) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton with n states. Let L be the regular set accepted by M . Let $w \in L$ and $|w| \geq m$. If $m \geq n$, then there exists x, y, z such that $w = xyz$, $y \neq \Lambda$ and $xy^iz \in L$ for each $i \geq 0$.



Pumping Lemma for Regular Language

Proof - Let

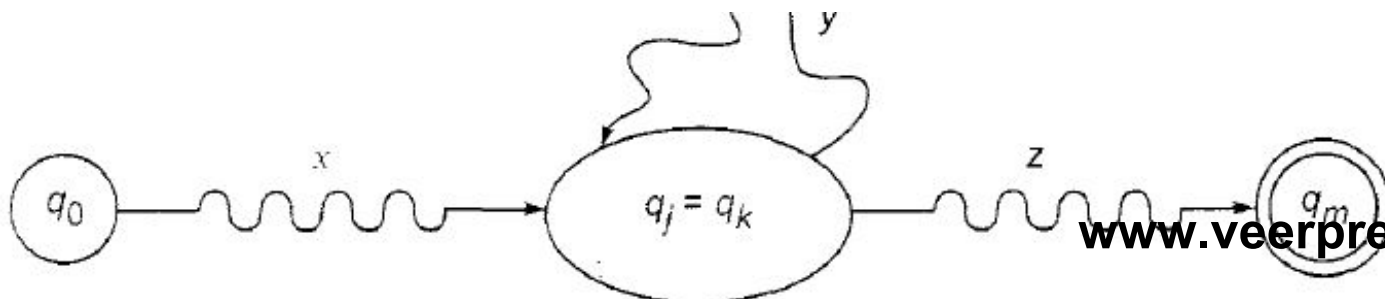
$$w = a_1 a_2 \dots a_m, \quad m \geq n$$

$$\delta(q_0, a_1 a_2 \dots a_i) = q_i \quad \text{for } i = 1, 2, \dots, m; \quad Q_1 = \{q_0, q_1, \dots, q_m\}$$

That is, Q_1 is the sequence of states in the path with path value $w = a_1 a_2 \dots a_m$. As there are only n distinct states, at least two states in Q_1 must coincide. Among the various pairs of repeated states, we take the first pair. Let us take them as q_j and q_k ($q_j = q_k$). Then j and k satisfy the condition $0 \leq j < k \leq n$.

The string w can be decomposed into three substrings $a_1 a_2 \dots a_j$, $a_{j+1} \dots a_k$ and $a_{k+1} \dots a_m$. Let x , y , z denote these strings $a_1 a_2 \dots a_j$, $a_{j+1} \dots a_k$, $a_{k+1} \dots a_m$ respectively. As $k \leq n$, $|xy| \leq n$ and $w = xyz$. The path with the path value w in the transition diagram of M is shown in Fig. 5.27.

The automaton M starts from the initial state q_0 . On applying the string x , it reaches $q_j (= q_k)$. On applying the string y , it comes back to $q_j (= q_k)$. So after application of y^i for each $i \geq 0$, the automaton is in the same state q_j . On applying z , it reaches q_m , a final state. Hence, $xy^i z \in L$. As every state in Q_1 is obtained by applying an input symbol, $y \neq \Lambda$. **■**



Example

5.4 APPLICATION OF PUMPING LEMMA

This theorem can be used to prove that certain sets are not regular. We now give the steps needed for proving that a given set is not regular.

Step 1 Assume that L is regular. Let n be the number of states in the corresponding finite automaton.

Step 2 Choose a string w such that $|w| \geq n$. Use pumping lemma to write $w = xyz$, with $|xy| \leq n$ and $|y| > 0$.

Step 3 Find a suitable integer i such that $xy^iz \notin L$. This contradicts our assumption. Hence L is not regular.

Note: The crucial part of the procedure is to find i such that $xy^iz \notin L$. In some cases we prove $xy^iz \notin L$ by considering $|xy^iz|$. In some cases we may have to use the 'structure' of strings in L .

Example 1

Show that the set $L = \{a^{i^2} \mid i \geq 1\}$ is not regular.

Solution

Step 1 Suppose L is regular. Let n be the number of states in the finite automaton accepting L .

Step 2 Let $w = a^{n^2}$. Then $|w| = n^2 > n$. By pumping lemma, we can write $w = xyz$ with $|xy| \leq n$ and $|y| > 0$.

Step 3 Consider xy^2z . $|xy^2z| = |x| + 2|y| + |z| > |x| + |y| + |z|$ as $|y| > 0$. This means $n^2 = |xyz| = |x| + |y| + |z| < |xy^2z|$. As $|xy| \leq n$, we have $|y| \leq n$. Therefore,

$$|xy^2z| = |x| + 2|y| + |z| \leq n^2 + n$$

i.e.

$$n^2 < |xy^2z| \leq n^2 + n < n^2 + n + n + 1$$

Hence, $|xy^2z|$ strictly lies between n^2 and $(n+1)^2$, but is not equal to any one of them. Thus $|xy^2z|$ is not a perfect square and so $xy^2z \notin L$. But by pumping lemma, $xy^2z \in L$. This is a contradiction.

Example 2

Show that $L = \{a^p \mid p \text{ is a prime}\}$ is not regular.

Solution

Step 1 We suppose L is regular. Let n be the number of states in the finite automaton accepting L .

Step 2 Let p be a prime number greater than n . Let $w = a^p$. By pumping lemma, w can be written as $w = xyz$, with $|xy| \leq n$ and $|y| > 0$. x, y, z are simply strings of a 's. So, $y = a^m$ for some $m \geq 1$ (and $\leq n$).

Step 3 Let $i = p + 1$. Then $|xy^iz| = |xyz| + |y^{i-1}| = p + (i - 1)m = p + pm$. By pumping lemma, $xy^iz \in L$. But $|xy^iz| = p + pm = p(1 + m)$, and $p(1 + m)$ is not a prime. So $xy^iz \notin L$. This is a contradiction. Thus L is not regular.

Example 3

Show that $L = \{0^i 1^i \mid i \geq 1\}$ is not regular.

Solution

Step 1 Suppose L is regular. Let n be the number of states in the finite automaton accepting L .

Step 2 Let $w = 0^n 1^n$. Then $|w| = 2n > n$. By pumping lemma, we write $w = xyz$ with $|xy| \leq n$ and $|y| \neq 0$.

Step 3 We want to find i so that $xy^i z \notin L$ for getting a contradiction. The string y can be in any of the following forms:

Case 1 y has 0's, i.e. $y = 0^k$ for some $k \geq 1$.

Case 2 y has only 1's, i.e. $y = 1^l$ for some $l \geq 1$.

Case 3 y has both 0's and 1's, i.e. $y = 0^k 1^j$ for some $k, j \geq 1$.

In Case 1, we can take $i = 0$. As $xyz = 0^n 1^n$, $xz = 0^{n-k} 1^n$. As $k \geq 1$, $n - k \neq n$. So, $xz \notin L$.

In Case 2, take $i = 0$. As before, xz is $0^n 1^{n-l}$ and $n \neq n - l$. So, $xz \notin L$.

In Case 3, take $i = 2$. As $xyz = 0^{n-k} 0^k 1^j 1^{n-j}$, $xy^2 z = 0^{n-k} 0^k 1^j 0^k 1^j 1^{n-j}$. As $xy^2 z$ is not of the form $0^i 1^i$, $xy^2 z \notin L$.

Thus in all the cases we get a contradiction. Therefore, L is not regular.

Closure Properties of Regular Sets

In this section we discuss the closure properties of regular sets under (i) set union, (ii) concatenation, (iii) closure (iteration), (iv) transpose, (v) set intersection, and (vi) complementation.

Closure Properties of Regular Sets

Theorem 5.6 If L is regular then L^T is also regular.

Proof As L is regular by (vii), given at the end of Section 5.2.7, we can construct a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ such that $T(M) = L$.

We construct a transition system M' by starting with the state diagram of M , and reversing the direction of the directed edges. The set of initial states of M' is defined as the set F , and q_0 is defined as the (only) final state of M' , i.e. $M' = (Q, \Sigma, \delta', F, \{q_0\})$.

If $w \in T(M)$, we have a path from q_0 to some final state in F with path value w . By 'reversing the edges', we get a path in M' from some final state in F to q_0 . Its path value is w^T . So $w^T \in T(M')$. In a similar way, we can see that if $w_1 \in T(M')$, then $w_1^T \in T(M)$. Thus from the state diagram it is easy to see that $T(M') = T(M)^T$. We can prove rigorously that $w \in T(M)$ iff $w^T \in T(M')$ by induction on $|w|$. So $T(M)^T = T(M')$. By (viii) of Section 5.2.7, $T(M')$ is regular, i.e. $T(M)^T$ is regular. ■

Closure Properties of Regular Sets

Theorem 5.7 If L is a regular set over Σ , then $\Sigma^* - L$ is also regular over Σ .

Proof As L is regular by (vii), given at the end of Section 5.2.7, we can construct a DFA $M = (Q, \Sigma, \delta, q_0, F)$ accepting L , i.e. $L = T(M)$.

We construct another DFA $M' = (Q, \Sigma, \delta, q_0, F')$ by defining $F' = Q - F$, i.e. M and M' differ only in their final states. A final state of M' is a nonfinal state of M and vice versa. The state diagrams of M and M' are the same except for the final states.

$w \in T(M')$ if and only if $\delta(q_0, w) \in F' = Q - F$, i.e. iff $w \notin L$. This proves $T(M') = \Sigma^* - L$. ■

Closure Properties of Regular Sets

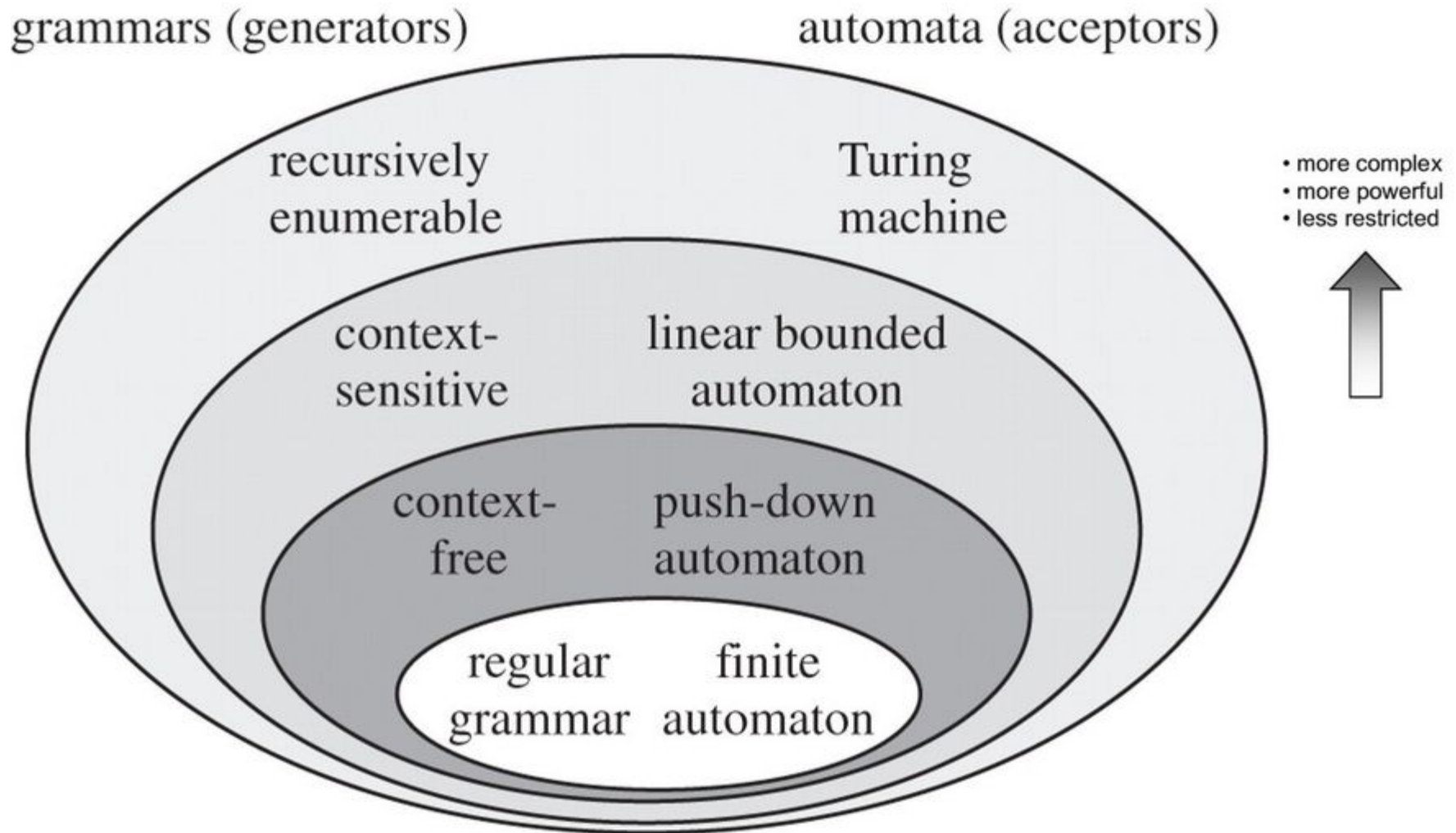
Theorem 5.8 If X and Y are regular sets over Σ , then $X \cap Y$ is also regular over Σ .

Proof By DeMorgan's law for sets, $X \cap Y = \Sigma^* - ((\Sigma^* - X) \cup (\Sigma^* - Y))$. By Theorem 5.7, $\Sigma^* - X$ and $\Sigma^* - Y$ are regular. So, $(\Sigma^* - X) \cup (\Sigma^* - Y)$ is also regular. By applying Theorem 5.7, once again $\Sigma^* - ((\Sigma^* - X) \cup (\Sigma^* - Y))$ is regular. i.e. $X \cap Y$ is regular. ■

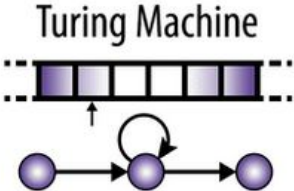

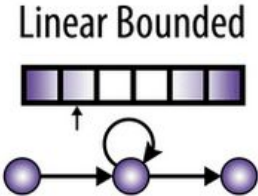

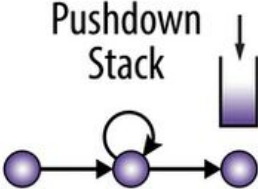
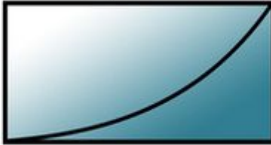
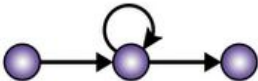

Classification of Grammar

Type	Grammar	Production rules
Type 0	unrestricted	$\alpha \rightarrow \beta$
Type 1	context-sensitive	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type 2	context-free	$A \rightarrow \gamma$
Type 3	regular	$A \rightarrow aB$ or $A \rightarrow Ba$

Chomsky Hierarchy



Languages, Automaton, Grammar, Recognition

Language	Automaton	Grammar	Recognition
Recursively Enumerable Languages	<p>Turing Machine</p> 	<p>Unrestricted</p> $Baa \rightarrow A$	<p>Undecidable</p> 
Context-Sensitive Languages	<p>Linear Bounded</p> 	<p>Context Sensitive</p> $A t \rightarrow aA$	<p>Exponential?</p> 
Context-Free Languages	<p>Pushdown Stack</p> 	<p>Context Free</p> $S \rightarrow gS c$	<p>Polynomial</p> 
Regular Languages	<p>Finite-State Automaton</p> 	<p>Regular</p> $A \rightarrow cA$	<p>Linear</p> 

Regular Set and Regular Grammar

- Regular grammar is a type of grammar that describes a regular language. A regular grammar is a mathematical object, G , which consists of four components, $G = (N, E, P, S)$, where
- **N**: non-empty, finite set of non-terminal symbols,
- **E**: a finite set of terminal symbols, or alphabet, symbols,
- **P**: a set of grammar rules, each of one having one of the forms
 - $A \rightarrow xB$
 - $A \rightarrow x$
 - $A \rightarrow \epsilon$, Here ϵ =empty string, $A, B \in N, x \in \Sigma^*$
- **S** $\in N$ is the start symbol.

Regular Set and Regular Grammar

- This grammar can be of two forms:
 - Right Linear Regular Grammar
 - Left Linear Regular Grammar

Regular Set and Regular Grammar

- **Right Linear Regular Grammar**

- In this type of regular grammar, all the non-terminals on the right-hand side exist at the rightmost place, i.e; right ends.

- **Examples :**

- $A \rightarrow a, A \rightarrow aB, A \rightarrow \epsilon$ where, A and B are non-terminals, a is terminal, and ϵ is empty string
- $S \rightarrow 00B \mid 11S, B \rightarrow 0B \mid 1B \mid 0 \mid 1$ where, S and B are non-terminals, and 0 and 1 are terminals

Regular Set and Regular Grammar

- **Left Linear Regular Grammar**

- In this type of regular grammar, all the non-terminals on the left-hand side exist at the leftmost place, i.e; left ends.

- **Examples :**

- $A \rightarrow a, A \rightarrow \mathbf{Ba}, A \rightarrow \epsilon$ where, A and B are non-terminals, a is terminal, and ϵ is empty string
- $S \rightarrow \mathbf{B00} \mid \mathbf{S11}, B \rightarrow \mathbf{B0} \mid \mathbf{B1} \mid 0 \mid 1$, where S and B are non-terminals, and 0 and 1 are terminals

Regular Set and Regular Grammar

- **Left linear to Right Linear Regular Grammar**
 - In this type of conversion, we have to shift all the left-handed non-terminals to right as shown in example given below:
- Left linear
 - $A \rightarrow Ba, B \rightarrow ab$
- Right linear
 - $A \rightarrow abaB, B \rightarrow \epsilon$
 - OR
 - $A \rightarrow abB, B \rightarrow a$
- So, this can be done to give multiple answers. Example explained above have multiple answers other than the given once.

Regular Set and Regular Grammar

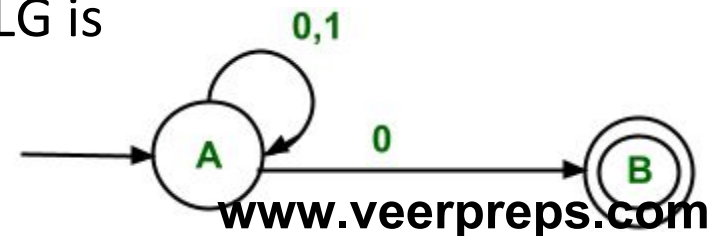
- **Right linear to Left Linear Regular Grammar**
 - In this type of conversion, we have to shift all the right-handed non-terminals to left as shown in example given below:
- Right linear
 - $A \rightarrow Ab, B \rightarrow ab$
- Left linear
 - $A \rightarrow \mathbf{B}aba, B \rightarrow \text{epsilon}$
 - OR
 - $A \rightarrow \mathbf{B}ab B \rightarrow a$
- So, this can be done to give multiple answers. Example explained above have multiple answers other than the given once.

Regular Grammar and DFA

- **Conversion of RLG to FA:**
 - Start from the first production.
 - From every left alphabet (or variable) go to the symbol followed by it.
 - **Start state:** It will be the first production state.
 - **Final state:** Take those states which end up with terminals without further non-terminals.

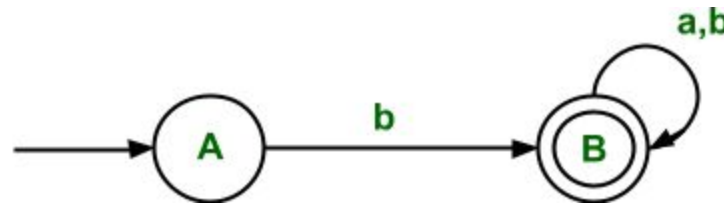
Regular Grammar and DFA

- **Example:** The RLL grammar for Language(L), represents a set of all strings which end with 0.
 - $A \rightarrow 0A/1A/0B, B \rightarrow \epsilon$
- **So the FA for corresponding to RLG can be found out as**
Start with variable A and use its production.
 - For production $A \rightarrow 0A$, this means after getting input symbol 0, the transition will remain in the same state.
 - For production, $A \rightarrow 1A$, this means after getting input symbol 1, the state transition will take place from State A to A.
 - For production $A \rightarrow 0B$, this means after getting input symbol 0, the state transition will take place from State A to B.
 - For production $B \rightarrow \epsilon$, this means there is no need for state transition. This means it would be the final state in the corresponding FA as RHS is terminal.
- So the final NFA for the corresponding RLG is

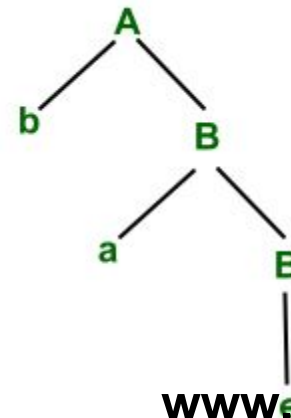
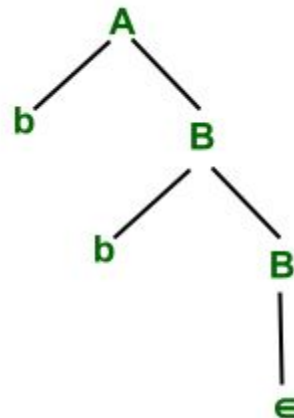
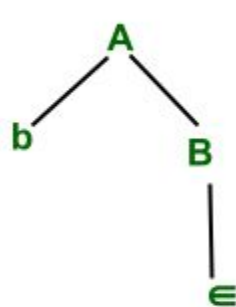


Regular Grammar and DFA

- **Example:** FA for accepting strings that start with b



- $\Sigma = \{a,b\}$ Initial state(q_0) = A Final state(F) = B
- **The RLG corresponding to FA is**
- **$A \rightarrow bB, B \rightarrow \epsilon / aB / bB$**
- The above grammar is RLG, which can be written directly through FA.



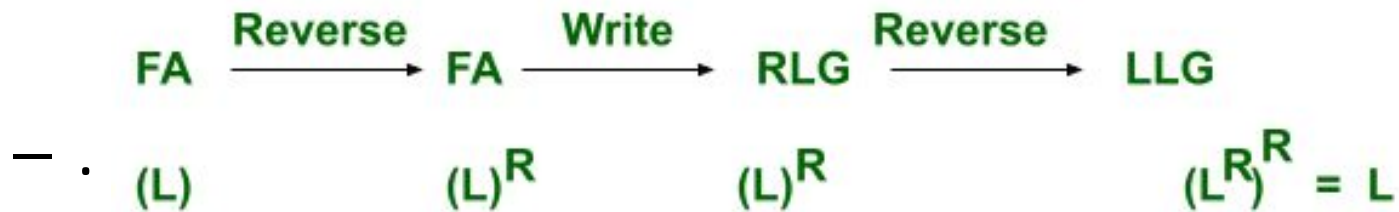
- .

RLG To LLG

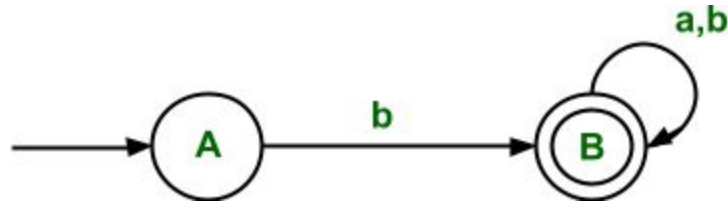
- The above RLG can derive strings that start with b and after that any input symbol(i.e. $\Sigma = \{a, b\}$ can be accepted).
- The regular language corresponding to RLG is
 - $L = \{b, ba, bb, baa, bab, bba, bbb, \dots\}$
- If we reverse the above production of the above RLG, then we get
 - $A \rightarrow Bb, B \rightarrow \epsilon / Ba / Bb$ It derives the language that contains all the strings which end with b.
 - i.e. $L' = \{b, bb, ab, aab, bab, abb, bbb, \dots\}$
- So we can conclude that if we have FA that represents language L and if we convert it, into RLG, which again represents language L, but after reversing RLG we get LLG which represents language L'(i.e. reverse of L).

RLG To LLG

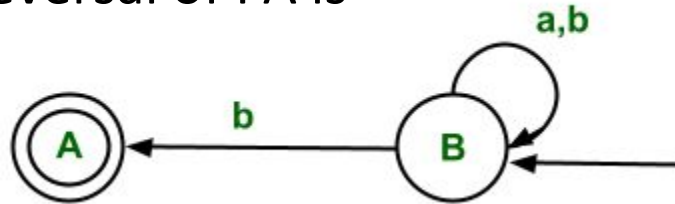
- **For converting the RLG into LLG for language L, the following procedure needs to be followed:**
 - Step 1: Reverse the FA for language L
 - Step 2: Write the RLG for it.
 - Step 3: Reverse the right linear grammar. after this we get the grammar that generates the language that represents the LLG for the same language L.



RLG To LLG



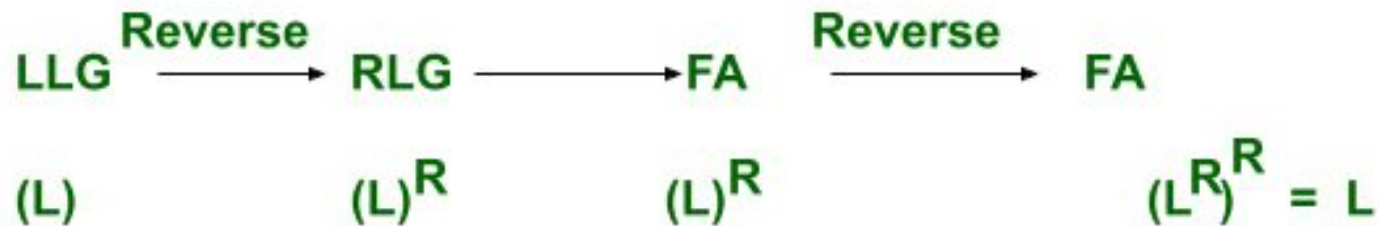
- **Step1:** The reversal of FA is



- **Step 2:** The corresponding RLG for this reversed FA is
 - $B \rightarrow aB/bB/bA, A \rightarrow \epsilon$
- **Step 3:** The reversing the above RLG we get
 - $B \rightarrow Ba/Bb/Ab, A \rightarrow \epsilon$
- So this is LLG for language L(which represents all strings that start with b).
 - $L = \{b, ba, bb, baa, bab, bba, bbb, \dots\}$

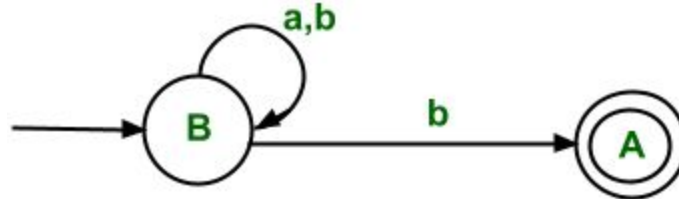
LLG To RLG

- **Explanation:** First convert the LLG which represents the Language(L) to RLG, which represents, the reversal of language L(i.e. L^R) then design FA corresponding to it(i.e. FA for Language L^R). Then reverse the FA. Then the final FA is FA for language L).

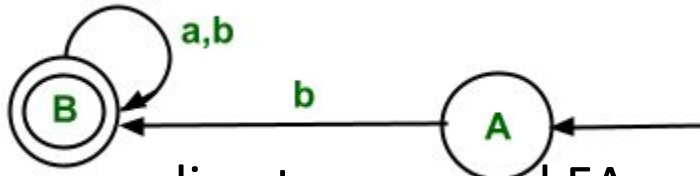


LLG To RLG

- **Conversion of LLG to RLG:** For example, the above grammar is taken which represents language L(i.e. set of all strings that start with b)
 - The LLG for this grammar is $B \rightarrow Ba/Bb/Ab, A \rightarrow \epsilon$
- **Step 1:** Convert the LLG into FA(i.e. the conversion procedure is the same as above)



- **Step 2:** Reverse the FA(i.e. initial state is converted into final state and convert final state to initial state and reverse all edges)



- **Step 3:** Write RLG corresponding to reversed FA.
 - $A \rightarrow bB \quad B \rightarrow aB/bB/\epsilon$

Regular Expressions

Definitions

Equivalence to Finite Automata

RE's: Introduction

- *Regular expressions* are an algebraic way to describe languages.
- They describe exactly the regular languages.
- If E is a regular expression, then $L(E)$ is the language it defines.
- We'll describe RE's and their languages recursively.

RE's: Definition

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

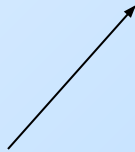
RE's: Definition – (2)

- **Induction 1:** If E_1 and E_2 are regular expressions, then $E_1 + E_2$ is a regular expression, and $L(E_1 + E_2) = L(E_1) \cup L(E_2)$.
- **Induction 2:** If E_1 and E_2 are regular expressions, then $E_1 E_2$ is a regular expression, and $L(E_1 E_2) = L(E_1) L(E_2)$.

Concatenation : the set of strings wx such that w is in $L(E_1)$ and x is in $L(E_2)$.

RE's: Definition – (3)

- **Induction 3:** If E is a RE, then E^* is a RE, and $L(E^*) = (L(E))^*$.



Closure, or “Kleene closure” = set of strings $w_1w_2\dots w_n$, for some $n \geq 0$, where each w_i is in $L(E)$.

Note: when $n=0$, the string is ϵ .

Precedence of Operators

- Parentheses may be used wherever needed to influence the grouping of operators.
- Order of precedence is $*$ (highest), then concatenation, then $+$ (lowest).

Examples: RE's

- $L(\mathbf{01}) = \{01\}$.
- $L(\mathbf{01+0}) = \{01, 0\}$.
- $L(\mathbf{0(1+0)}) = \{01, 00\}$.
 - Note order of precedence of operators.
- $L(\mathbf{0^*}) = \{\varepsilon, 0, 00, 000, \dots\}$.
- $L((\mathbf{0+10})^*(\varepsilon+\mathbf{1})) =$ all strings of 0's and 1's without two consecutive 1's.

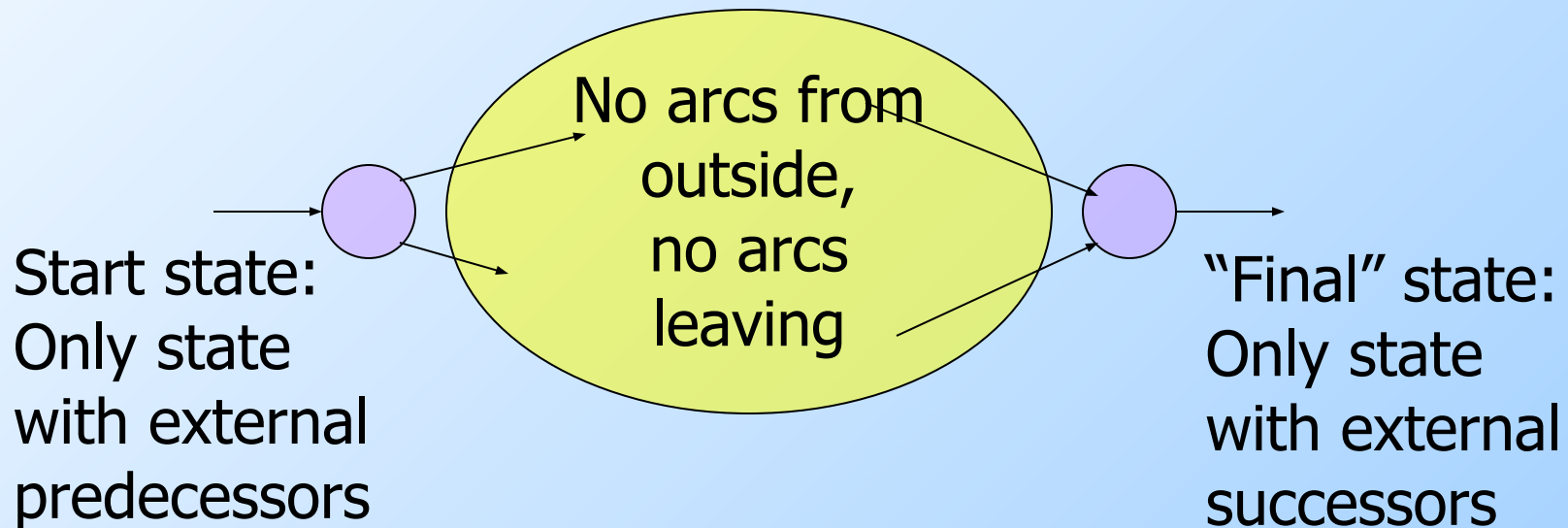
Equivalence of RE's and Automata

- We need to show that for every RE, there is an automaton that accepts the same language.
 - Pick the most powerful automaton type: the ϵ -NFA.
- And we need to show that for every automaton, there is a RE defining its language.
 - Pick the most restrictive type: the DFA.

Converting a RE to an ε -NFA

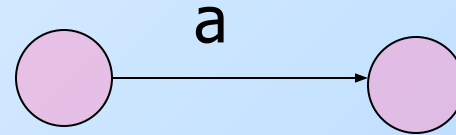
- Proof is an induction on the number of operators (+, concatenation, *) in the RE.
- We always construct an automaton of a special form (next slide).

Form of ε -NFA's Constructed

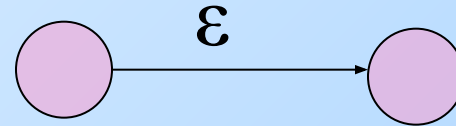


RE to ϵ -NFA: Basis

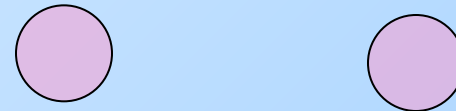
- Symbol **a**:



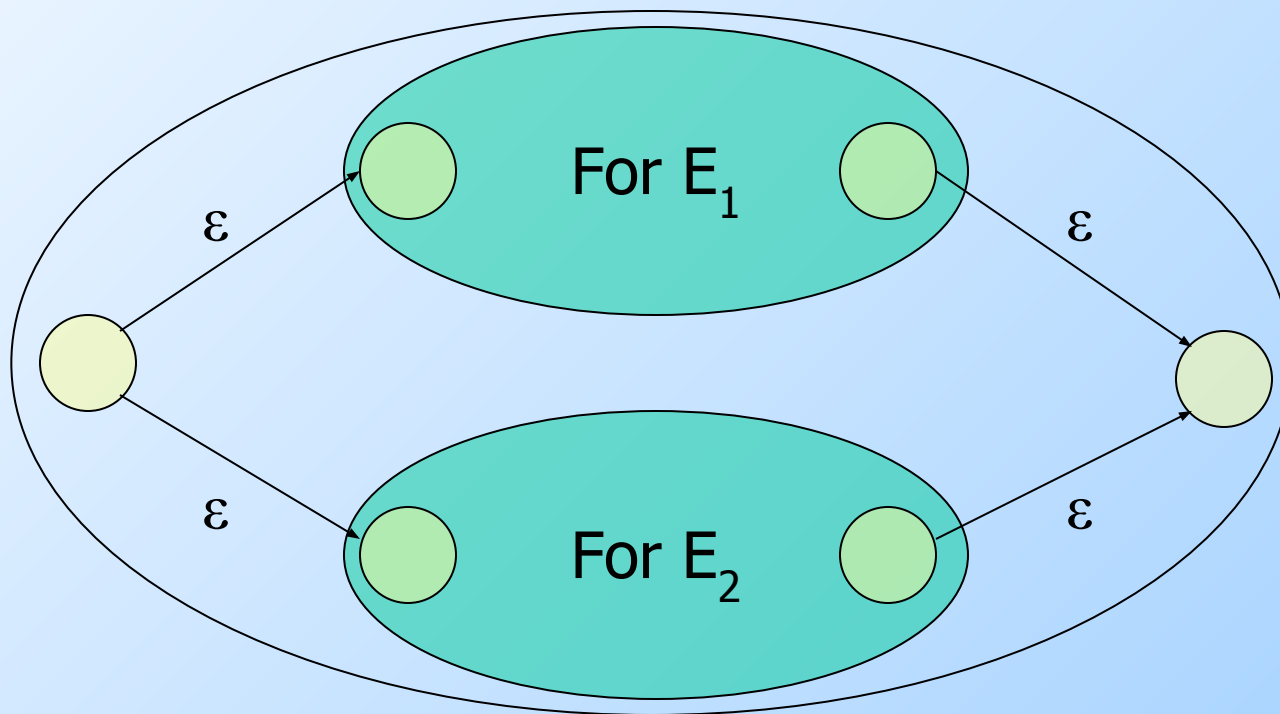
- ϵ :



- \emptyset :

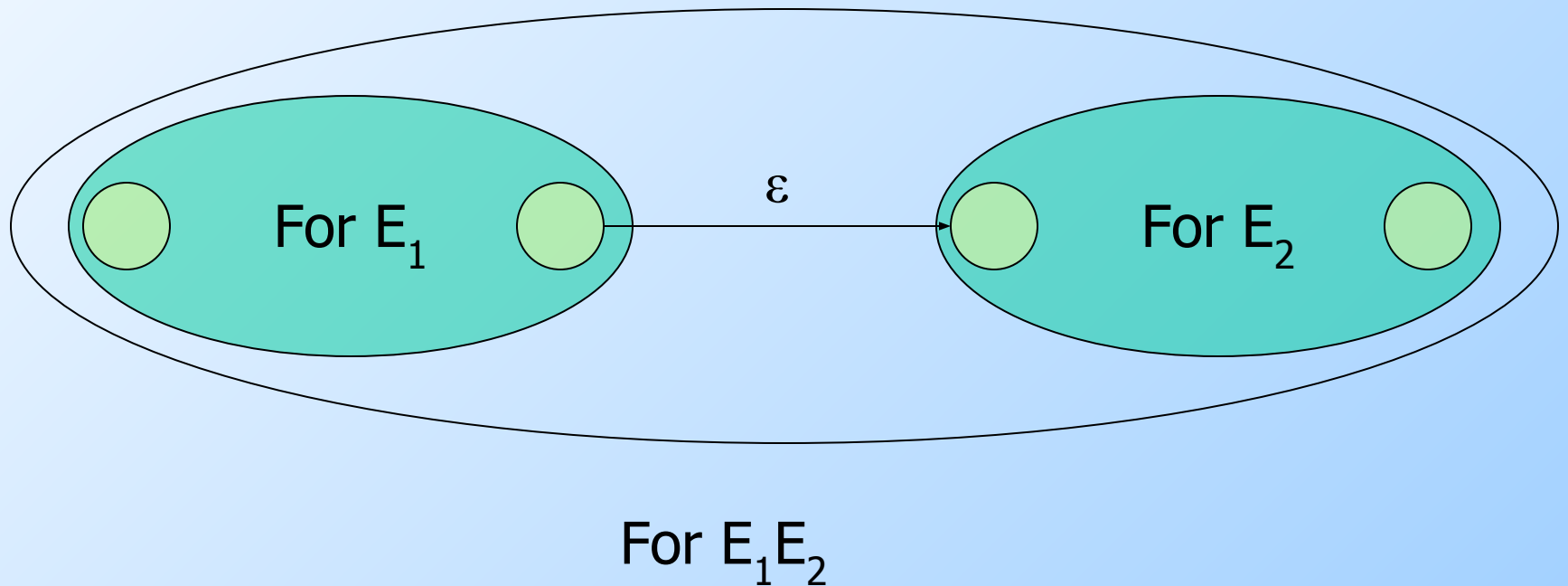


RE to ϵ -NFA: Induction 1 – Union

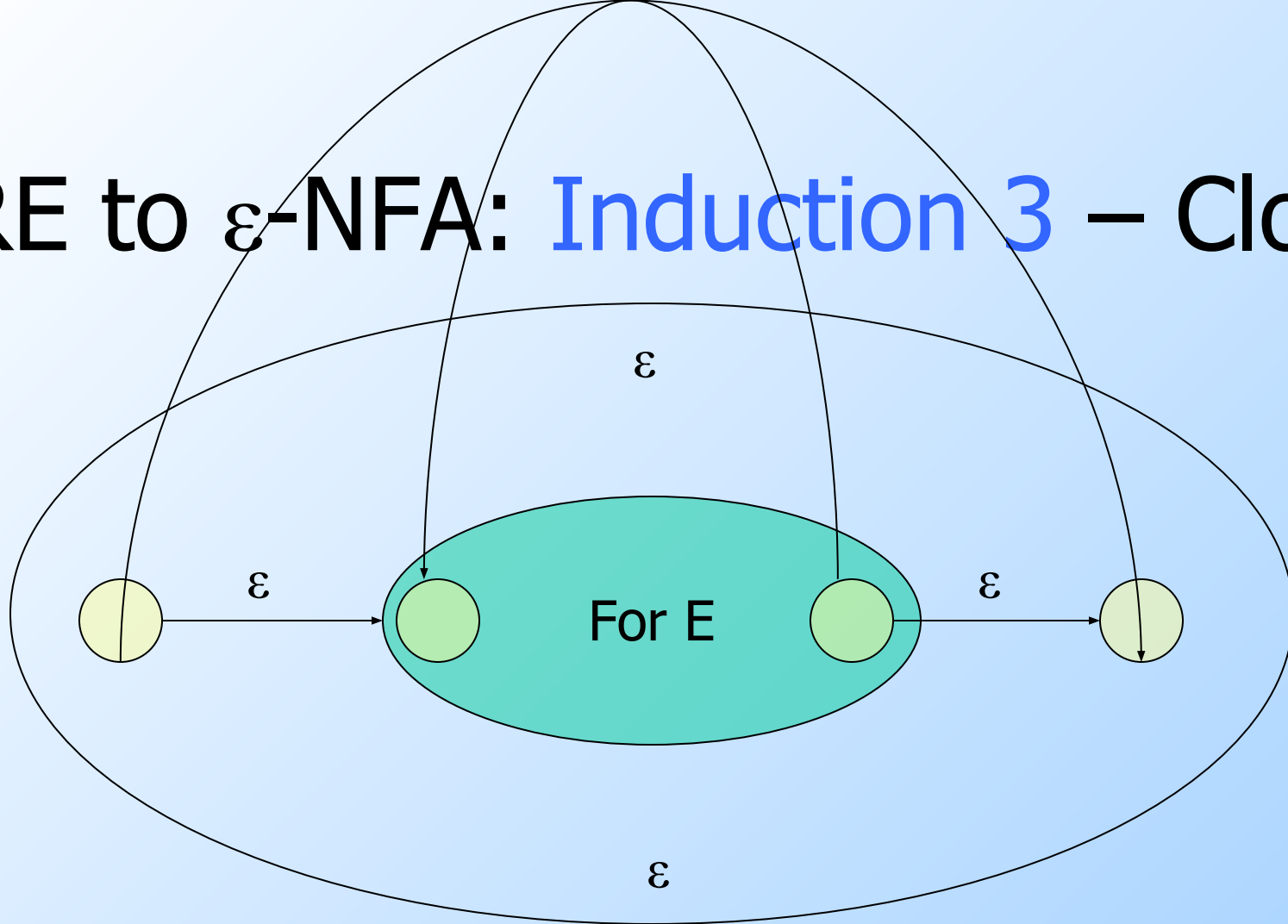


For $E_1 \cup E_2$

RE to ϵ -NFA: Induction 2 – Concatenation



RE to ϵ -NFA: Induction 3 – Closure



For E^*

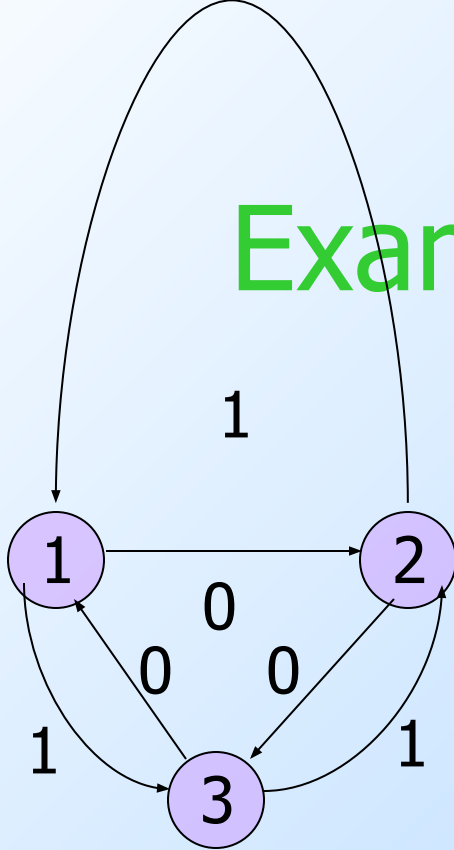
DFA-to-RE

- A strange sort of induction.
- States of the DFA are assumed to be $1, 2, \dots, n$.
- We construct RE's for the labels of restricted sets of paths.
 - **Basis**: single arcs or no arc at all.
 - **Induction**: paths that are allowed to traverse next state in order.

k-Paths

- A k-path is a path through the graph of the DFA that goes **through** no state numbered higher than k.
- Endpoints are not restricted; they can be any state.

Example: k-Paths



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

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

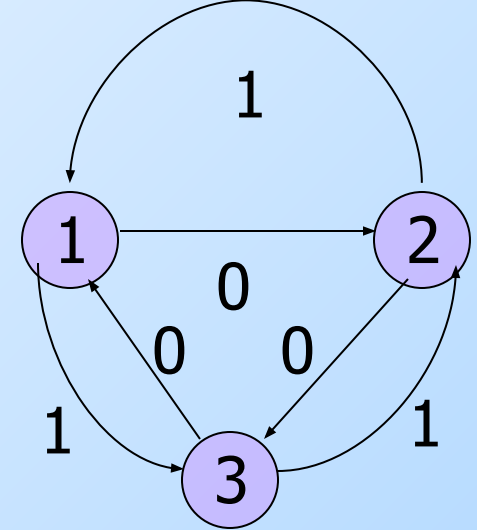
2-paths from 2 to 3:
RE for labels =
(10)*0+1(01)*1

3-paths from 2 to 3:
RE for labels = **111**

k-Path Induction

- Let R_{ij}^k be the regular expression for the set of labels of k-paths from state i to state j .
- **Basis:** $k=0$. R_{ij}^0 = sum of labels of arc from i to j .
 - \emptyset if no such arc.
 - But add ϵ if $i=j$.

Example: Basis



- $R_{12}^0 = \mathbf{0}.$
- $R_{11}^0 = \emptyset + \varepsilon = \varepsilon.$

k-Path Inductive Case

- A k-path from i to j either:
 1. Never goes through state k, or
 2. Goes through k one or more times.

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

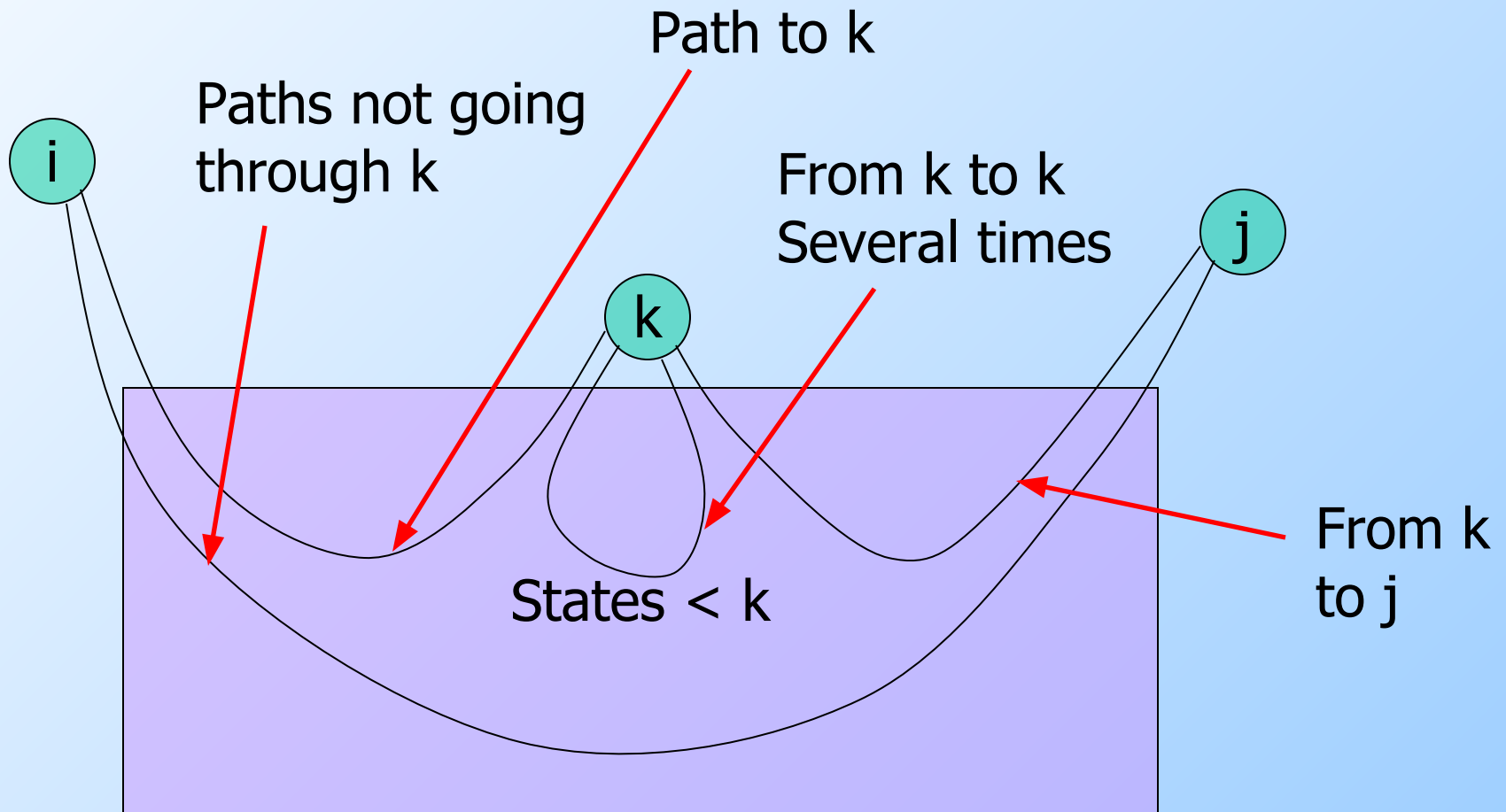
Doesn't go
through k

Goes from
i to k the
first time

Zero or
more times
from k to k

Then, from
k to j

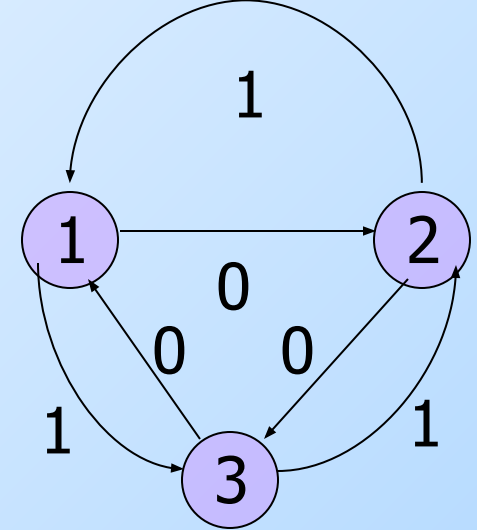
Illustration of Induction



Final Step

- The RE with the same language as the DFA is the sum (union) of R_{ij}^n , where:
 1. n is the number of states; i.e., paths are unconstrained.
 2. i is the start state.
 3. j is one of the final states.

Example



- $R_{23}^3 = R_{23}^2 + R_{23}^2(R_{33}^2)*R_{33}^2 =$
 $R_{23}^2(R_{33}^2)*$
- $R_{23}^2 = (\mathbf{10})*\mathbf{0} + \mathbf{1}(\mathbf{01})*\mathbf{1}$
- $R_{33}^2 = \mathbf{0}(\mathbf{01})*(\mathbf{1} + \mathbf{00}) + \mathbf{1}(\mathbf{10})*(\mathbf{0} + \mathbf{11})$
- $R_{23}^3 = [(\mathbf{10})*\mathbf{0} + \mathbf{1}(\mathbf{01})*\mathbf{1}]$
 $[(\mathbf{0}(\mathbf{01})*(\mathbf{1} + \mathbf{00}) + \mathbf{1}(\mathbf{10})*(\mathbf{0} + \mathbf{11}))]*$

Summary

- Each of the three types of automata (DFA, NFA, ϵ -NFA) we discussed, and regular expressions as well, define exactly the same set of languages: the regular languages.

Algebraic Laws for RE's

- Union and concatenation behave sort of like addition and multiplication.
 - $+$ is commutative and associative; concatenation is associative.
 - Concatenation distributes over $+$.
 - **Exception**: Concatenation is not commutative.

Identities and Annihilators

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



Midterm I review

Reading: Chapters 1-4



Test Details

- In class, Wednesday, Feb. 25, 2015
3:10pm-4pm
- Comprehensive
- Closed book, closed notes



Syllabus

- Formal proofs
- Finite Automata
 - NFA, DFA, ϵ -NFA
- Regular expressions
- Regular languages & properties
 - Pumping lemma for regular languages



Finite Automata

- **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
- ϵ -NFA is an NFA that allows ϵ -transitions
- What are their differences?



Deterministic Finite Automata

- A DFA is defined by the 5-tuple:
 - $\{Q, \Sigma, q_0, F, \delta\}$
- Two ways to define:
 - State-diagram (preferred)
 - State-transition table
- DFA construction checklist:
 - Associate states with their meanings
 - Capture all possible combinations/input scenarios
 - break into cases & subcases wherever possible
 - Are outgoing transitions defined for every symbol from every state?
 - Are final/accepting states marked?
 - Possibly, dead/error-states will have to be included depending on the design.



Non-deterministic Finite Automata

- A NFA is defined by the 5-tuple:
 - $\{Q, \Sigma, q_0, F, \delta\}$
- Two ways to represent:
 - State-diagram (preferred)
 - State-transition table
- NFA construction checklist:
 - Has *at least* one nondeterministic transition
 - Capture only valid input transitions
 - Can ignore invalid input symbol transitions (paths will die implicitly)
 - Outgoing transitions defined only for valid symbols from every state
 - Are final/accepting states marked?



NFA to DFA conversion

- Checklist for NFA to DFA conversion
 - Two approaches:
 - Enumerate all possible subsets, or
 - Use **lazy construction** strategy (to save time)
 - Introduce subset states only as needed
 - In your solutions, use the lazy construction procedure by default unless specified otherwise.
 - Any subset containing an accepting state is also accepting in the DFA
 - Have you made a special entry for Φ , the empty subset?
 - This will correspond to the dead/error state



ϵ -NFA to DFA conversion

- Checklist for ϵ -NFA to DFA conversion
 - First take ECLOSE(start state)
 - New start state = ECLOSE(start state)
 - Remember: ECLOSE(q) include q
- Then convert to DFA:
 - Use *lazy construction* strategy for introducing subset states only as needed (same as NFA to DFA), but ...
 - Only difference : take ECLOSE after transitions and also include those states in the subset corresponding to your destination state.
 - E.g., if q_i goes to $\{q_j, q_k\}$, then your subset must be: $\text{ECLOSE}(q_j) \cup \text{ECLOSE}(q_k)$
- Again, check for a special entry for Φ if needed



Regular Expressions

- A way to express accepting patterns
- Operators for Reg. Exp.
 - (E) , $L(E+F)$, $L(EF)$, $L(E^*)$..
- Reg. Language \square Reg. Exp. (checklist):
 - Capture all cases of valid input strings
 - Express each case by a reg. exp.
 - Combine all of them using the $+$ operator
 - Pay attention to operator precedence
 - Try to reuse previously built regular expressions wherever possible



Regular Expressions...

- DFA to Regular expression
 - Enumerate all paths from start to every final state
 - Generate regular expression for each segment, and concatenate
 - Combine the reg. exp. for all each path using the + operator
- Reg. Expression to ϵ -NFA conversion
 - Inside-to-outside construction
 - Start making states for every atomic unit of RE
 - Combine using: concatenation, + and * operators as appropriate
 - For connecting adjacent parts, use ϵ -transitions
 - Remember to note down final states



Regular Expressions...

- Algebraic laws
 - Commutative
 - Associative
 - Distributive
 - Identity
 - Annihilator
 - Idempotent
 - Involving Kleene closures (* operator)



English description of lang.

- Finite automata □ english description
- Regular expression □ english description

“**English description**” should be similar to how we have been describing languages in class

- E.g., languages of strings over $\{a,b\}$ that end in b ; or
- Languages of binary strings that have 0 in its even position, etc.

Thumbrule: the simpler the description is, the better.

However, make sure that the description should accurately capture the language.



Pumping Lemma

- Purpose: Regular or not? Verification technique
- Steps/Checklist for Pumping Lemma (in order):
 - 1) Let N \square pumping lemma constant
 - 2) Choose a template string w in L , such that $|w| \geq N$.
(Note: the string you choose should depend on N . And the choice of your w will affect the rest of the proof. So select w judiciously. Generally, a simple choice of w would be a good starting point. But if that doesn't work, then go for others.)
 - 3) Now w should satisfy P/L, and therefore, all three conditions of the lemma. Specifically, using conditions $|xy| \leq N$ and $y \neq \epsilon$, try to conclude something about the property of the xy part and y part separately.
 - 4) Next, use one of these two below strategies to arrive at the conclusion of $xy^kz \notin L$ (for some value of k):
 - Pump down ($k=0$)
 - Pump up ($k \geq 2$)Note: arriving at a contradiction using either pumping up OR down is sufficient. No need to show both.

Working out pumping lemma based proofs as a 2-player game:

- Steps (think of this 2-party game):

Good guy (us)

Builds w using N
(without assuming
any particular value of N)

Tries to break the third condition
of P/L without assuming any
particular $\{x,y,z\}$ split

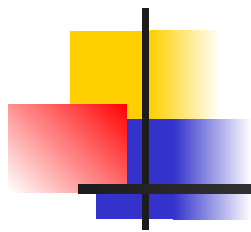
- this is done by first pumping down ($k=0$)
- if that does not work, then try pumping up ($k \geq 2$)

Bad guy (someone else)

Claims L is regular

\Rightarrow Knows N and has the freedom
to choose any value of $N \geq 1$

Comes up with $\{x,y,z\}$ combination,
s.t. $w=xyz$
(again, has the freedom to choose
any xyz split, but meeting
the two conditions of P/L:
i.e., $|xy| \leq N$ and $y \neq \epsilon$)



GOOD LUCK!

Not in syllabus for this Midterm I



Reg. Lang. Properties

- Closed under:
 - Union
 - Intersection
 - Complementation
 - Set difference
 - Reversal
 - Homomorphism & inverse homomorphism
- Look at all DFA/NFA constructions for the above
 - Expect example questions

Not in syllabus for this Midterm I



Other Reg. Lang. Properties

- Membership question
- Emptiness test
 - Reachability test
- Finiteness test
 - Remove states that are:
 - Unreachable, or cannot lead to accepting
 - Check for cycle in left-over graph
 - Or the reg. expression approach

Not in syllabus for this Midterm I



DFA minimization

- Steps:
 - Remove unreachable states first
 - Detect equivalent states
- Table-filing algorithm

Not in syllabus for this Midterm I



Other properties

- Are 2 DFAs equivalent?
 - Application of table filling algo

Context-Free Languages & Grammars (CFLs & CFGs)



Reading: Chapter 5



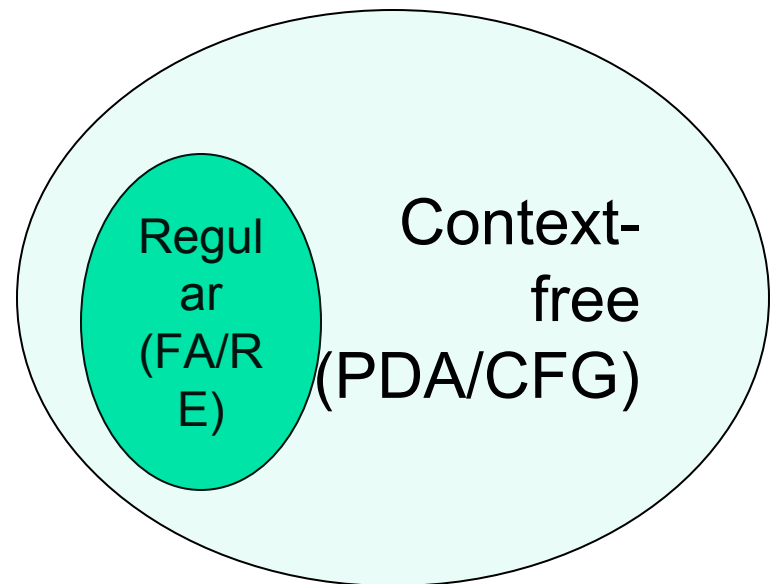
Not all languages are regular

- So what happens to the languages which are not regular?
- Can we still come up with a language recognizer?
 - i.e., something that will accept (or reject) strings that belong (or do not belong) to the language?



Context-Free Languages

- A language class larger than the class of regular languages
- Supports natural, recursive notation called “context-free grammar”
- Applications:
 - Parse trees, compilers
 - XML





An Example

- A palindrome is a word that reads identical from both ends
 - E.g., $\overrightarrow{\text{madam}}$, $\overleftarrow{\text{redivider}}$, $\overrightarrow{\text{malayalam}}$, $\overleftarrow{010010010}$
- Let $L = \{ w \mid w \text{ is a binary palindrome} \}$
- Is L regular?
 - No.
 - Proof:
 - Let $w = 0^N 1 0^N$ (assuming N to be the p/l constant)
 - By Pumping lemma, w can be rewritten as xyz , such that xy^kz is also L (for any $k \geq 0$)
 - But $|xy| \leq N$ and $y \neq \epsilon$
 - $\implies y = 0^+$
 - $\implies xy^kz$ *will NOT* be in L for $k=0$
 - \implies Contradiction

But the language of palindromes...

is a CFL, because it supports recursive substitution (in the form of a CFG)

- This is because we can construct a “grammar” like this:

- Productions
1. $A \Rightarrow \epsilon$
 2. $A \Rightarrow 0$
 3. $A \Rightarrow 1$
 4. $A \Rightarrow 0A0$
 5. $A \Rightarrow 1A1$

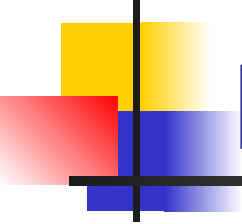
Terminal

Variable or non-terminal

Same as:

$A \Rightarrow 0A0 \mid 1A1 \mid 0 \mid 1 \mid \epsilon$

How does this grammar work?



How does the CFG for palindromes work?

An input string belongs to the language (i.e., accepted) iff it can be generated by the CFG

- Example: $w=01110$
- G can generate w as follows:

G :

$A \Rightarrow 0A0 \mid 1A1 \mid 0 \mid 1 \mid \epsilon$

1. $A \Rightarrow 0A0$
2. $\Rightarrow 01A10$
3. $\Rightarrow 01110$

Generating a string from a grammar:

1. Pick and choose a sequence of productions that would allow us to generate the string.
2. At every step, substitute one variable with one of its productions.



Context-Free Grammar: Definition

- A context-free grammar $G=(V,T,P,S)$, where:
 - V : set of variables or non-terminals
 - T : set of terminals (= alphabet $\cup \{\epsilon\}$)
 - P : set of *productions*, each of which is of the form
 $V \Rightarrow \alpha_1 \mid \alpha_2 \mid \dots$
 - Where each α_i is an arbitrary string of variables and terminals
 - $S \Rightarrow$ start variable

CFG for the language of binary palindromes:

$G=({A},\{0,1\},P,A)$

$P: A \Rightarrow 0 A 0 \mid 1 A 1 \mid 0 \mid 1 \mid \epsilon$



More examples

- Parenthesis matching in code
- Syntax checking
- In scenarios where there is a general need for:
 - Matching a symbol with another symbol, or
 - Matching a count of one symbol with that of another symbol, or
 - Recursively substituting one symbol with a string of other symbols



Example #2

- Language of balanced paranthesis
e.g., $()(((((())((()))))((()))))\dots$
- CFG?

G:
 $S \Rightarrow (S) \mid SS \mid \epsilon$

How would you “interpret” the string “ $(((((())((()))))((()))))$ ” using this grammar?



Example #3

- A grammar for $L = \{0^m 1^n \mid m \geq n\}$
- CFG?

G:
 $S \Rightarrow 0S1 \mid A$
 $A \Rightarrow 0A \mid \varepsilon$

How would you interpret the string “00000111”
using this grammar?



Example #4

A program containing **if-then(-else)** statements

if *Condition* **then** *Statement* **else** *Statement*

(Or)

if *Condition* **then** *Statement*

CFG?



More examples

- $L_1 = \{0^n \mid n \geq 0\}$
- $L_2 = \{0^n \mid n \geq 1\}$
- $L_3 = \{0^i 1^j 2^k \mid i=j \text{ or } j=k, \text{ where } i, j, k \geq 0\}$
- $L_4 = \{0^i 1^j 2^k \mid i=j \text{ or } i=k, \text{ where } i, j, k \geq 1\}$



Applications of CFLs & CFGs

- Compilers use parsers for syntactic checking
- Parsers can be expressed as CFGs
 1. Balancing paranthesis:
 - $B \Rightarrow BB \mid (B) \mid \textit{Statement}$
 - $\textit{Statement} \Rightarrow \dots$
 2. If-then-else:
 - $S \Rightarrow SS \mid \textit{if Condition then Statement else Statement} \mid \textit{if Condition then Statement} \mid \textit{Statement}$
 - $\textit{Condition} \Rightarrow \dots$
 - $\textit{Statement} \Rightarrow \dots$
 3. C paranthesis matching $\{ \dots \}$
 4. Pascal *begin-end* matching
 5. YACC (Yet Another Compiler-Compiler)



More applications

- Markup languages

- Nested Tag Matching

- HTML

- `<html> ...<p> </p> ... </html>`

- XML

- `<PC> ... <MODEL> ... </MODEL> .. <RAM> ... </RAM> ... </PC>`



Tag-Markup Languages

Roll \Rightarrow **<ROLL>** Class Students **</ROLL>**

Class \Rightarrow **<CLASS>** Text **</CLASS>**

Text \Rightarrow Char Text | Char

Char \Rightarrow **a | b | ... | z | A | B | .. | Z**

Students \Rightarrow Student Students | ϵ

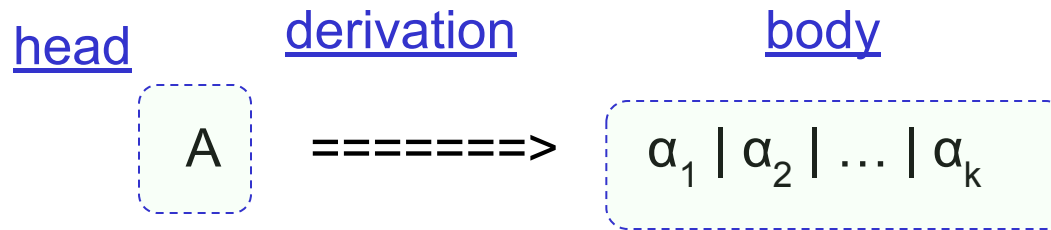
Student \Rightarrow **<STUD>** Text **</STUD>**

Here, the left hand side of each production denotes one non-terminals
(e.g., “Roll”, “Class”, etc.)

Those symbols on the right hand side for which no productions (i.e.,
substitutions) are defined are terminals (e.g., ‘a’, ‘b’, ‘|’, ‘<’, ‘>’, “ROLL”,
etc.)



Structure of a production



The above is same as:

1. $A \Longrightarrow \alpha_1$
2. $A \Longrightarrow \alpha_2$
3. $A \Longrightarrow \alpha_3$
- ...
- K. $A \Longrightarrow \alpha_k$

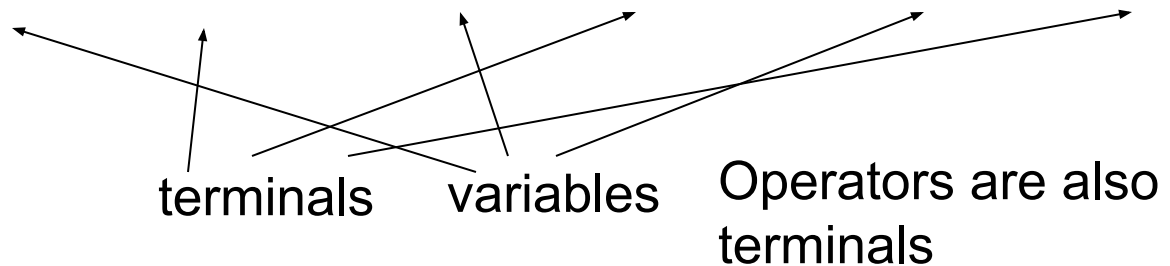


CFG conventions

- Terminal symbols $\leq a, b, c \dots$
- Non-terminal symbols $\leq A, B, C, \dots$
- Terminal or non-terminal symbols $\leq X, Y, Z$
- Terminal strings $\leq w, x, y, z$
- Arbitrary strings of terminals and non-terminals $\leq \alpha, \beta, \gamma, \dots$

Syntactic Expressions in Programming Languages

*result = a*b + score + 10 * distance + c*



Regular languages have only terminals

- Reg expression = $[a-z][a-z0-1]^*$
- If we allow only letters a & b, and 0 & 1 for constants (for simplification)
 - Regular expression = $(a+b)(a+b0+1)^*$

String membership

How to say if a string belong to the language defined by a CFG?

1. Derivation
 - Head to body
2. Recursive inference
 - Body to head

Both are equivalent forms

Example:

- $w = 01110$
- Is w a palindrome?

G:
 $A \Rightarrow 0A0 \mid 1A1 \mid 0 \mid 1 \mid \epsilon$

$A \Rightarrow 0A0$
 $\Rightarrow 01A10$
 $\Rightarrow 01110$



Simple Expressions...

- We can write a CFG for accepting simple expressions
- $G = (V, T, P, S)$
 - $V = \{E, F\}$
 - $T = \{0, 1, a, b, +, *, (,)\}$
 - $S = \{E\}$
 - $P:$
 - $E \Rightarrow E + E \mid E * E \mid (E) \mid F$
 - $F \Rightarrow aF \mid bF \mid 0F \mid 1F \mid a \mid b \mid 0 \mid 1$



Generalization of derivation

- Derivation is *head* \Rightarrow *body*
- $A \Rightarrow X$ (A derives X in a single step)
- $A \Rightarrow_G^* X$ (A derives X in a multiple steps)
- Transitivity:
IF $A \Rightarrow_G^* B$, and $B \Rightarrow_G^* C$, THEN $A \Rightarrow_G^* C$



Context-Free Language

- The language of a CFG, $G=(V,T,P,S)$, denoted by $L(G)$, is the set of terminal strings that have a derivation from the start variable S .
 - $L(G) = \{ w \text{ in } T^* \mid S \Rightarrow^*_G w \}$

Left-most & Right-most Derivation Styles

G:

$E \Rightarrow E + E \mid E * E \mid (E) \mid F$
 $F \Rightarrow aF \mid bF \mid 0F \mid 1F \mid \varepsilon$

Derive the string $a^*(ab+10)$ from G:

$E \xRightarrow{*}_G a^*(ab+10)$

Left-most derivation:

Always substitute leftmost variable

```

E
==> E * E
==> F * E
==> aF * E
==> a * E
==> a * (E)
==> a * (E + E)
==> a * (F + E)
==> a * (aF + E)
==> a * (abF + E)
==> a * (ab + E)
==> a * (ab + F)
==> a * (ab + 1F)
==> a * (ab + 10F)
==> a * (ab + 10)
    
```

Right-most derivation:

Always substitute rightmost variable

```

E
==> E * E
==> E * (E)
==> E * (E + E)
==> E * (E + F)
==> E * (E + 1F)
==> E * (E + 10F)
==> E * (E + 10)
==> E * (F + 10)
==> E * (aF + 10)
==> E * (abF + 0)
==> E * (abF + 10)
==> E * (ab + 10)
==> F * (ab + 10)
==> aF * (ab + 10)
==> a * (ab + 10)
    
```



Leftmost vs. Rightmost derivations

Q1) For every leftmost derivation, there is a rightmost derivation, and vice versa. True or False?

True - will use parse trees to prove this

Q2) Does every word generated by a CFG have a leftmost and a rightmost derivation?

Yes – easy to prove (reverse direction)

Q3) Could there be words which have more than one leftmost (or rightmost) derivation?

Yes – depending on the grammar



How to prove that your CFGs are correct?

(using induction)



CFG & CFL

$$\frac{G_{\text{pal}}}{A \Rightarrow 0A0 \mid 1A1 \mid 0 \mid 1 \mid \varepsilon}$$

- Theorem: A string w in $(0+1)^*$ is in $L(G_{\text{pal}})$, if and only if, w is a palindrome.

- Proof:
 - Use induction
 - on string length for the IF part
 - On length of derivation for the ONLY IF part



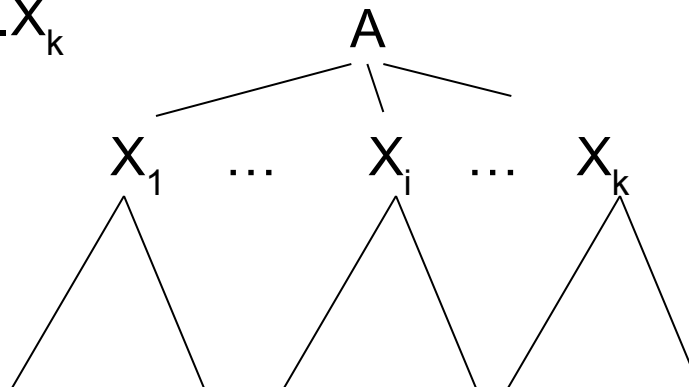
Parse trees

Parse Trees

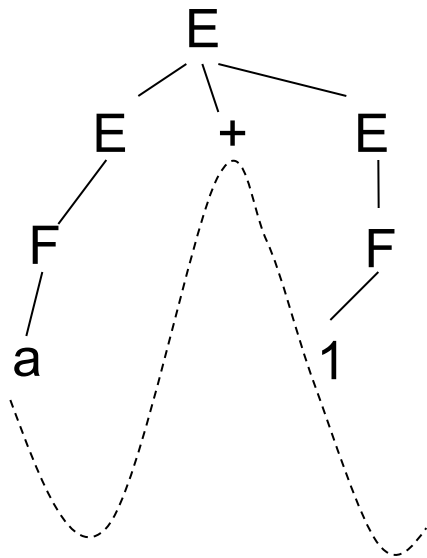
- Each CFG can be represented using a *parse tree*:
 - Each internal node is labeled by a variable in V
 - Each leaf is terminal symbol
 - For a production, $A \Rightarrow X_1 X_2 \dots X_k$, then any internal node labeled A has k children which are labeled from X_1, X_2, \dots, X_k from left to right

Parse tree for production and all other subsequent productions:

$A \Rightarrow X_1 \dots X_i \dots X_k$



Examples



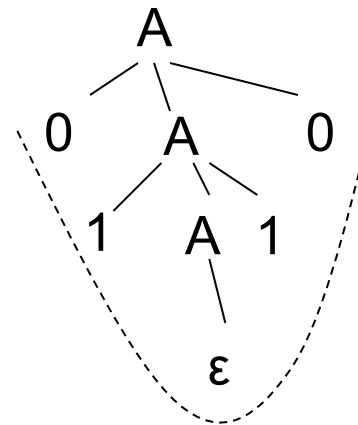
Parse tree for $a + 1$

G:

$E \Rightarrow E + E \mid E * E \mid (E) \mid F$

$F \Rightarrow aF \mid bF \mid 0F \mid 1F \mid 0 \mid 1 \mid a \mid b$

Recursive inference



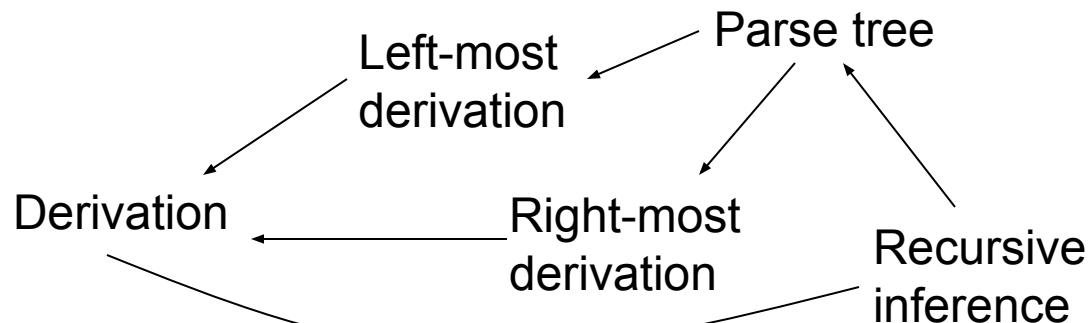
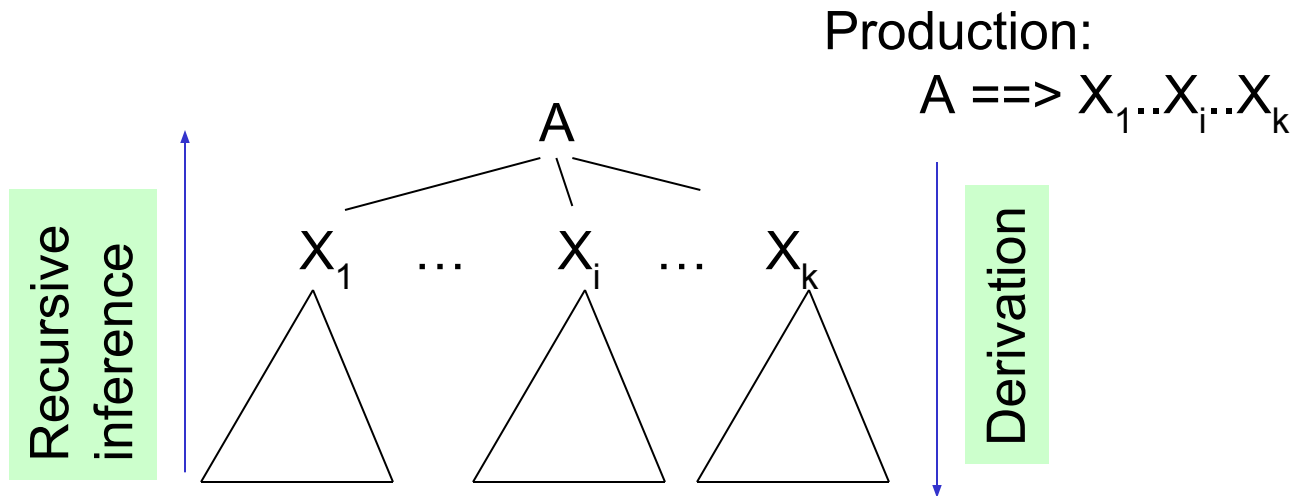
Parse tree for 0110

Derivation

G:

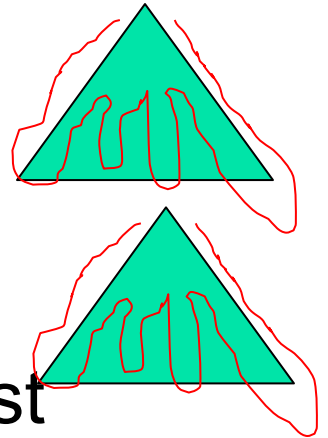
$A \Rightarrow 0A0 \mid 1A1 \mid 0 \mid 1 \mid \epsilon$

Parse Trees, Derivations, and Recursive Inferences



Interchangeability of different CFG representations

- Parse tree \Rightarrow left-most derivation
 - DFS left to right
- Parse tree \Rightarrow right-most derivation
 - DFS right to left
- \Rightarrow left-most derivation $=$ right-most derivation
- Derivation \Rightarrow Recursive inference
 - Reverse the order of productions
- Recursive inference \Rightarrow Parse trees
 - bottom-up traversal of parse tree



Connection between CFLs and RLs



What kind of grammars result for regular languages?

CFLs & Regular Languages

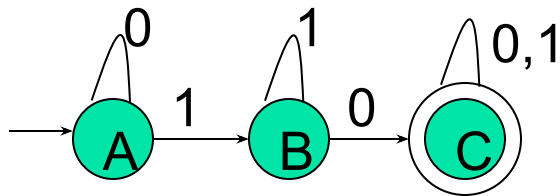
- A CFG is said to be *right-linear* if all the productions are one of the following two forms: $A \Rightarrow wB$ (or) $A \Rightarrow w$

Where:

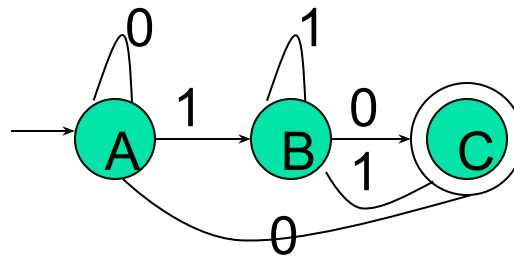
- A & B are variables,
- w is a string of terminals

- Theorem 1: Every right-linear CFG generates a regular language
- Theorem 2: Every regular language has a right-linear grammar
- Theorem 3: Left-linear CFGs also represent RLs

Some Examples



Right linear CFG?



Right linear CFG?

$\square A \Rightarrow 01B \mid C$
 $B \Rightarrow 11B \mid 0C \mid 1A$
 $C \Rightarrow 1A \mid 0 \mid 1$

Finite Automaton?



Ambiguity in CFGs and CFLs

Ambiguity in CFGs

- A CFG is said to be *ambiguous* if there exists a string which has more than one left-most derivation

Example:

$S \Rightarrow AS \mid \epsilon$

$A \Rightarrow A1 \mid 0A1 \mid 01$

Input string: 00111

Can be derived in two ways

LM derivation #1:

$S \Rightarrow AS$

$\Rightarrow 0A1S$

$\Rightarrow 0A11S$

$\Rightarrow 00111S$

$\Rightarrow 00111$

LM derivation #2:

$S \Rightarrow AS$

$\Rightarrow A1S$

$\Rightarrow 0A11S$

$\Rightarrow 00111S$

$\Rightarrow 00111$

Why does ambiguity matter?

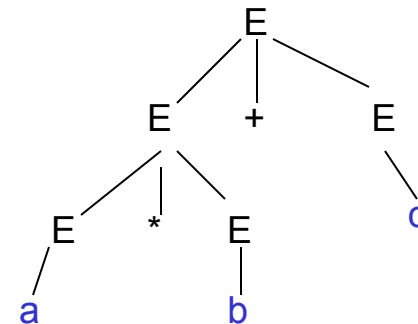
$E \Rightarrow E + E \mid E * E \mid (E) \mid a \mid b \mid c \mid 0 \mid 1$

Values are
different !!!

string = $a * b + c$

- LM derivation #1:

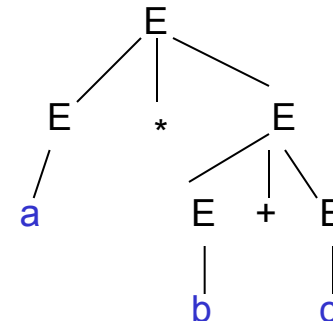
• $E \Rightarrow E + E \Rightarrow E * E + E$
 $\Rightarrow * a * b + c$



$(a*b)+c$

- LM derivation #2

• $E \Rightarrow E * E \Rightarrow a * E \Rightarrow$
 $a * E + E \Rightarrow * a * b + c$



$a*(b+c)$

The calculated value depends on which
of the two parse trees is actually used.

Removing Ambiguity in Expression Evaluations

- It MAY be possible to remove ambiguity for some CFLs
 - E.g., in a CFG for expression evaluation by imposing rules & restrictions such as precedence
 - This would imply rewrite of the grammar

- Precedence: $()$, $*$, $+$

Modified unambiguous version:

$$\begin{aligned} E &\Rightarrow E + T \mid T \\ T &\Rightarrow T * F \mid F \\ F &\Rightarrow I \mid (E) \\ I &\Rightarrow a \mid b \mid c \mid 0 \mid 1 \end{aligned}$$

Ambiguous version:

$$E \Rightarrow E + E \mid E * E \mid (E) \mid a \mid b \mid c \mid 0 \mid 1$$

How will this avoid ambiguity?



Inherently Ambiguous CFLs

- However, for some languages, it may not be possible to remove ambiguity
- A CFL is said to be *inherently ambiguous* if every CFG that describes it is ambiguous

Example:

- $L = \{ a^n b^n c^m d^m \mid n, m \geq 1 \} \cup \{ a^n b^m c^m d^n \mid n, m \geq 1 \}$
- L is inherently ambiguous
- Why?

Input string: $a^n b^n c^n d^n$



Summary

- Context-free grammars
- Context-free languages
- Productions, derivations, recursive inference, parse trees
- Left-most & right-most derivations
- Ambiguous grammars
- Removing ambiguity
- CFL/CFG applications
 - parsers, markup languages

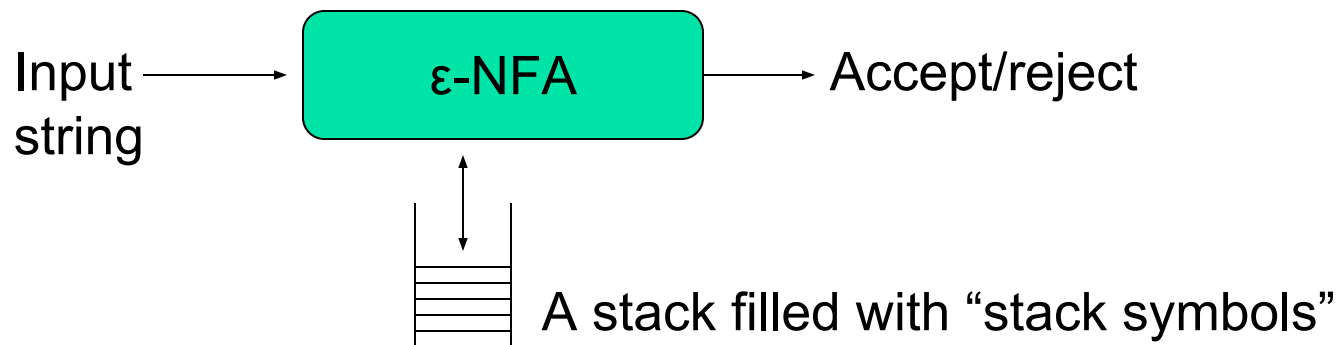


Pushdown Automata (PDA)

Reading: Chapter 6

PDA - the automata for CFLs

- What is?
 - FA to Reg Lang, PDA is to CFL
- $\text{PDA} == [\epsilon\text{-NFA} + \text{"a stack"}]$
- Why a stack?





Pushdown Automata - Definition

- A PDA $P := (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$:
 - Q : states of the ε -NFA
 - Σ : input alphabet
 - Γ : stack symbols
 - δ : transition function
 - q_0 : start state
 - Z_0 : Initial stack top symbol
 - F : Final/accepting states

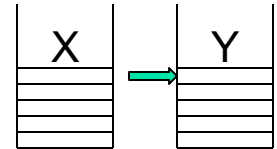
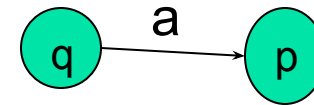
$$\delta : \overset{\text{old state}}{Q} \times \overset{\text{input symb.}}{\Sigma} \times \overset{\text{Stack top}}{\Gamma} \Rightarrow \overset{\text{new state(s)}}{Q} \times \overset{\text{new Stack top(s)}}{\Gamma}$$

δ : The Transition Function

$$\delta(q, a, X) = \{(p, Y), \dots\}$$

state transition from q to p
a is the next input symbol
X is the current stack *top* symbol
Y is the replacement for X;
it is in Γ^* (a string of stack symbols)

- i. Set $Y = \epsilon$ for: Pop(X)
- ii. If $Y = X$: stack top is unchanged
- iii. If $Y = Z_1 Z_2 \dots Z_k$: X is popped and is replaced by Y in reverse order (i.e., Z_1 will be the new stack top)



	Y = ?	Action
i)	$Y = \epsilon$	Pop(X)
ii)	$Y = X$	Pop(X) Push(X)
iii)	$Y = Z_1 Z_2 \dots Z_k$	Pop(X) Push(Z_k) Push(Z_{k-1}) ... Push(Z_2) Push(Z_1)



Example

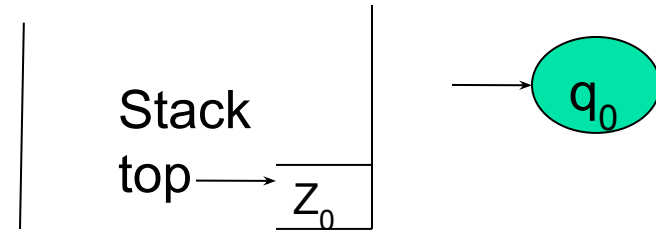
Let $L_{ww^R} = \{ww^R \mid w \text{ is in } (0+1)^*\}$

- CFG for L_{ww^R} : $S \Rightarrow 0S0 \mid 1S1 \mid \varepsilon$
- PDA for L_{ww^R} :
- $P := (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

$$= (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

PDA for L_{ww^R}

Initial state of the PDA:



1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$
2. $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$

First symbol push on stack

3. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$
4. $\delta(q_0, 0, 1) = \{(q_0, 01)\}$
5. $\delta(q_0, 1, 0) = \{(q_0, 10)\}$
6. $\delta(q_0, 1, 1) = \{(q_0, 11)\}$

Grow the stack by pushing new symbols on top of old (w-part)

7. $\delta(q_0, \epsilon, 0) = \{(q_1, 0)\}$
8. $\delta(q_0, \epsilon, 1) = \{(q_1, 1)\}$
9. $\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$

Switch to popping mode, nondeterministically (boundary between w and w^R)

10. $\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$
11. $\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$

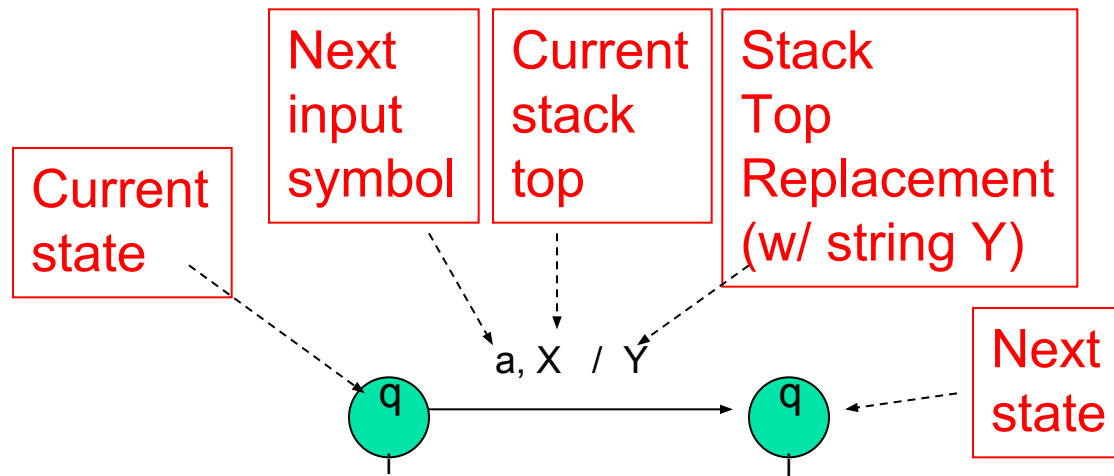
Shrink the stack by popping matching symbols (w^R -part)

12. $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$

Enter acceptance state

PDA as a state diagram

$$\delta(q_i, a, X) = \{(q_j, Y)\}$$



PDA for L_{wwr} : Transition Diagram

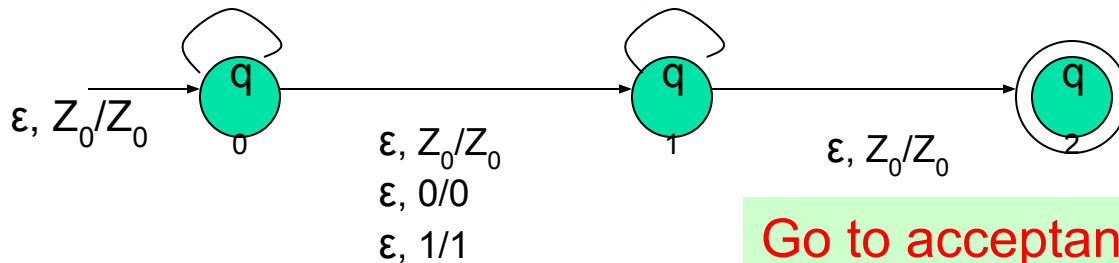
Grow stack

$0, Z_0/0Z_0$
 $1, Z_0/1Z_0$
 $0, 0/00$
 $0, 1/01$
 $1, 0/10$
 $1, 1/11$

Pop stack for
matching symbols

$0, 0/\epsilon$
 $1, 1/\epsilon$

$\Sigma = \{0, 1\}$
 $\Gamma = \{Z_0, 0, 1\}$
 $Q = \{q_0, q_1, q_2\}$



Switch to
popping mode

Go to acceptance

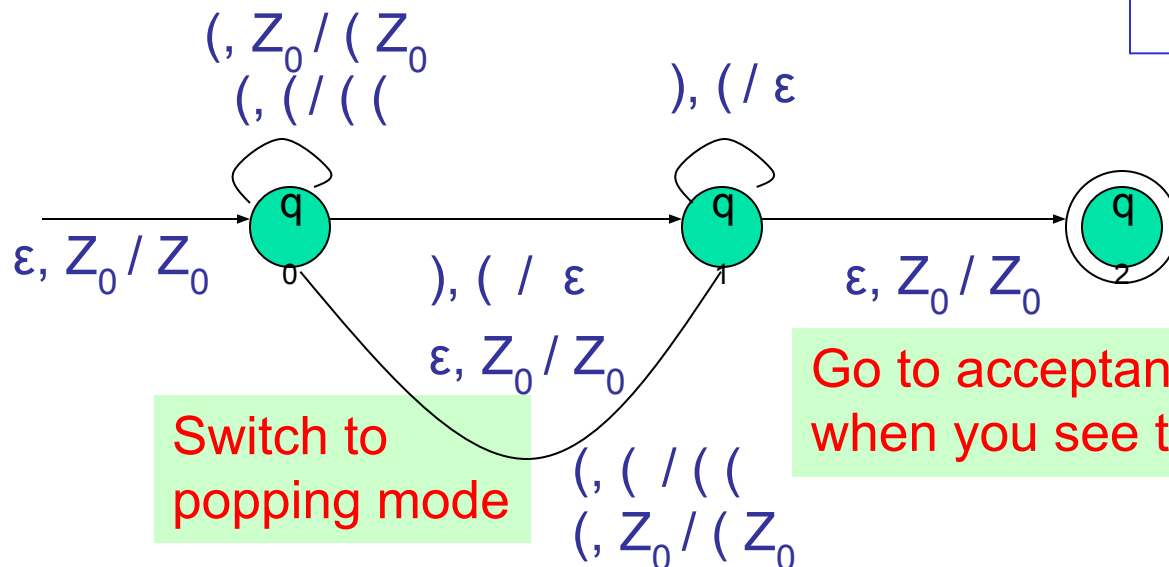
This would be a non-deterministic PDA

Example 2: language of balanced paranthesis

Grow stack

Pop stack for matching symbols

$$\begin{aligned}\Sigma &= \{ (,) \} \\ \Gamma &= \{ Z_0, (\} \\ Q &= \{ q_0, q_1, q_2 \}\end{aligned}$$

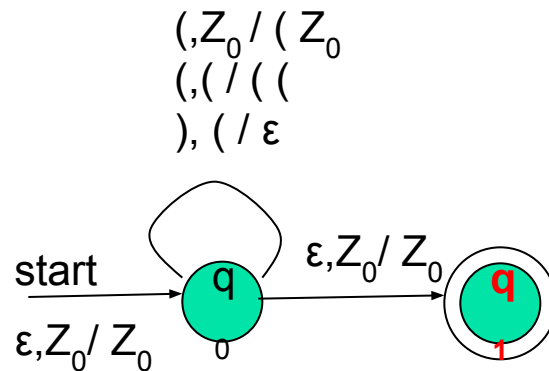


Switch to popping mode

Go to acceptance (by final state) when you see the stack bottom symbol

To allow adjacent blocks of nested parenthesis

Example 2: language of balanced paranthesis (another design)



$$\begin{aligned}\Sigma &= \{ (,) \} \\ \Gamma &= \{ Z_0, (\} \\ Q &= \{ q_0, q_1 \}\end{aligned}$$



PDA's Instantaneous Description (ID)

A PDA has a configuration at any given instance:

(q, w, y)

- q - current state
- w - remainder of the input (i.e., unconsumed part)
- y - current stack contents as a string from top to bottom of stack

If $\delta(q, a, X) = \{(p, A)\}$ is a transition, then the following are also true:

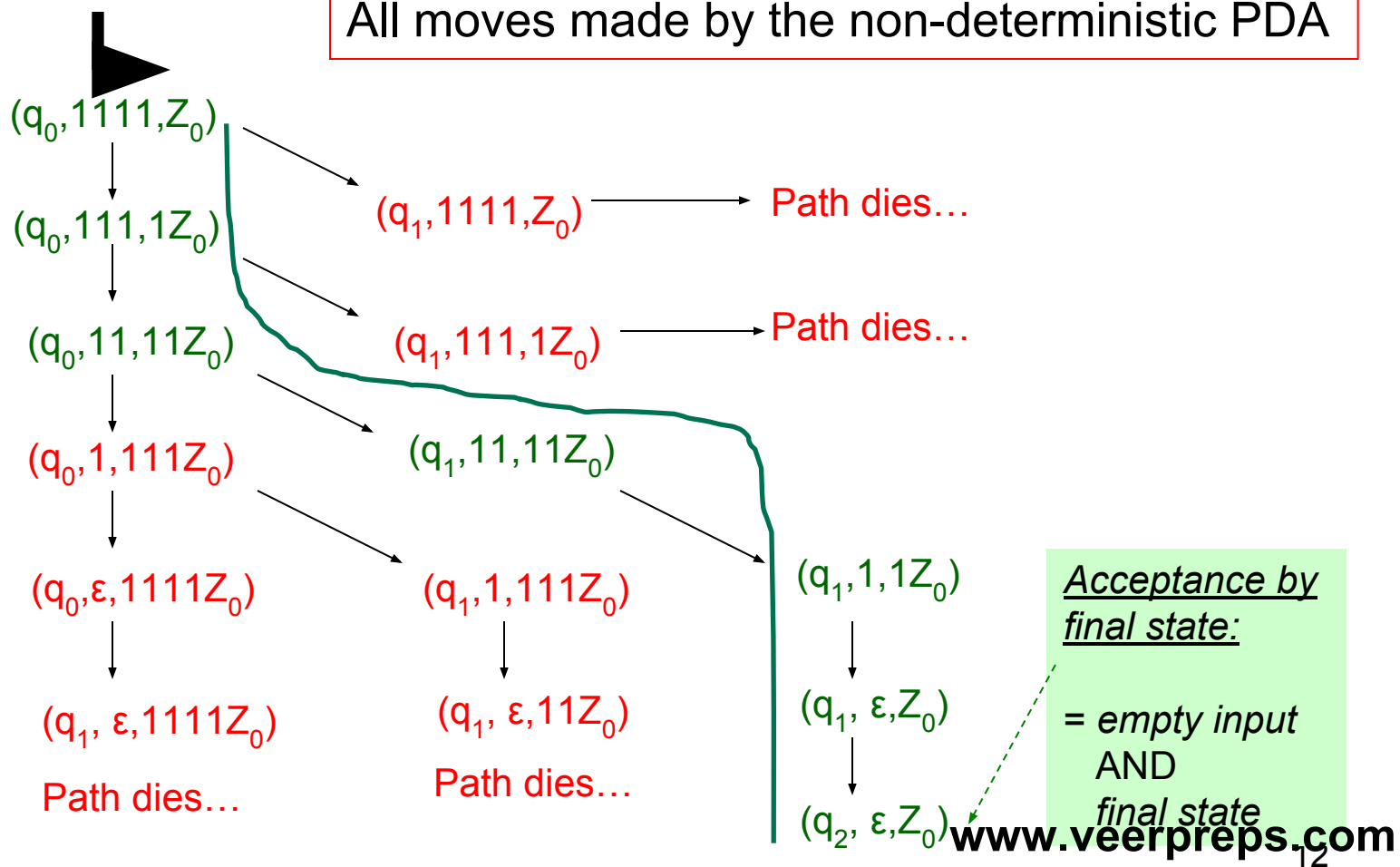
- $(q, a, X) \vdash (p, \epsilon, A)$
- $(q, aw, XB) \vdash (p, w, AB)$

\vdash sign is called a “turnstile notation” and represents one move

\vdash^* sign represents a sequence of moves

How does the PDA for L_{wwr} work on input “1111”?

All moves made by the non-deterministic PDA





Principles about IDs

- Theorem 1: If for a PDA,
 $(q, x, A) \vdash^{*} (p, y, B)$, then for any string $w \in \Sigma^{*}$ and $\gamma \in \Gamma^{*}$, it is also true that:
 - $(q, xw, A\gamma) \vdash^{*} (p, yw, B\gamma)$
- Theorem 2: If for a PDA,
 $(q, xw, A) \vdash^{*} (p, yw, B)$, then it is also true that:
 - $(q, x, A) \vdash^{*} (p, y, B)$

There are two types of PDAs that one can design:
those that accept by final state or by empty stack

Acceptance by...

- *PDAs that accept by final state:*

- For a PDA P , the language accepted by P , denoted by $L(P)$ by *final state*, is:

- $\{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, A)\}, \text{ s.t., } q \in F$

Checklist:

- input exhausted?
- in a final state?

- *PDAs that accept by empty stack:*

- For a PDA P , the language accepted by P , denoted by $N(P)$ by *empty stack*, is:

- $\{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}, \text{ for any } q \in Q.$

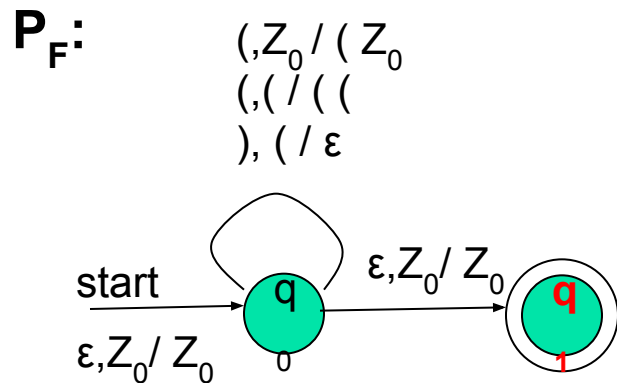
Q) Does a PDA that accepts by empty stack need any final state specified in the design?

Checklist:

- input exhausted?
- is the stack empty?

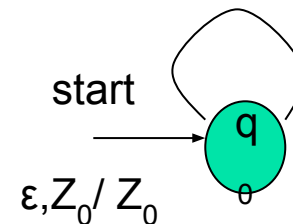
Example: L of balanced parenthesis

PDA that accepts by final state



An equivalent PDA that accepts by empty stack

P_N :

$$\begin{array}{l} (, Z_0 / (Z_0 \\ (, (/ ((\\), (/ \epsilon \\ \epsilon, Z_0 / \epsilon \end{array}$$


How will these two PDAs work on the input: $((()) ()) ()$



PDA for L_{ww^R} : Proof of correctness

- Theorem: The PDA for L_{ww^R} accepts a string x by final state **if and only if** x is of the form ww^R .
- Proof:
 - *(if-part)* If the string is of the form ww^R then there exists a sequence of IDs that leads to a final state:
 $(q_0, ww^R, Z_0) \vdash^* (q_0, w^R, wZ_0) \vdash^* (q_1, w^R, wZ_0) \vdash^* (q_1, \varepsilon, Z_0) \vdash^* (\mathbf{q_2}, \varepsilon, Z_0)$
 - *(only-if part)*
 - Proof by induction on $|x|$

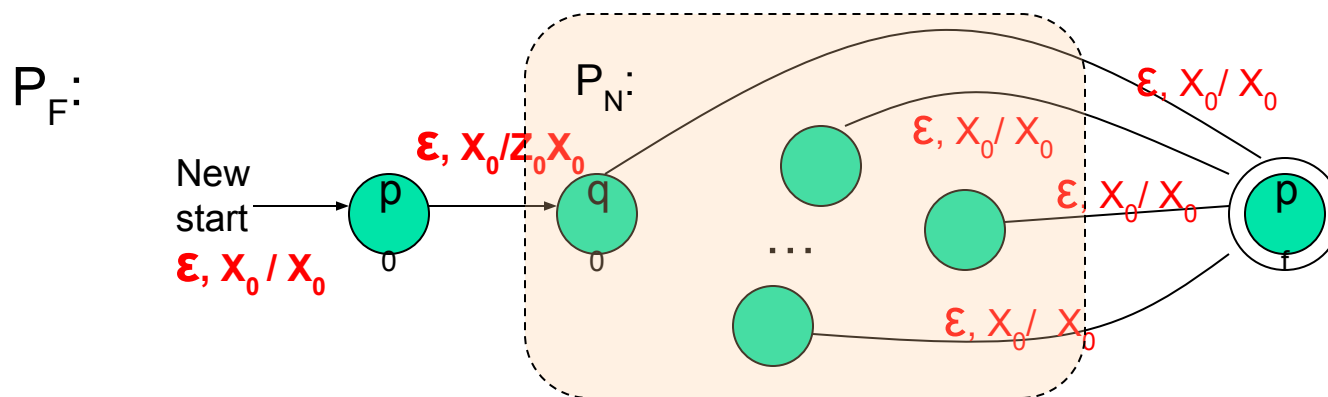


PDAs accepting by final state and empty stack are equivalent

- $P_F \leq$ PDA accepting by final state
 - $P_F = (Q_F, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$
- $P_N \leq$ PDA accepting by empty stack
 - $P_N = (Q_N, \Sigma, \Gamma, \delta_N, q_0, Z_0)$
- Theorem:
 - $(P_N \Rightarrow P_F)$ For every P_N , there exists a P_F s.t. $L(P_F) = L(P_N)$
 - $(P_F \Rightarrow P_N)$ For every P_F , there exists a P_N s.t. $L(P_F) = L(P_N)$

$P_N \Rightarrow P_F$ construction

- Whenever P_N 's stack becomes empty, make P_F go to a final state without consuming any addition symbol
- To detect empty stack in P_N : P_F pushes a new stack symbol X_0 (not in Γ of P_N) initially before simulating P_N

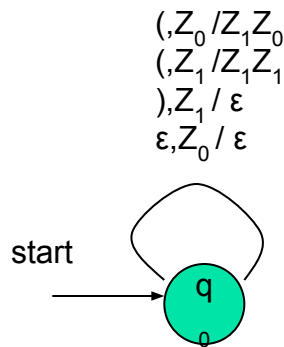


$$P_F = (Q_N \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

Example: Matching parenthesis “(” “)”

$P_N: (\{q_0\}, \{(\,,)\}, \{Z_0, Z_1\}, \delta_N, q_0, Z_0)$

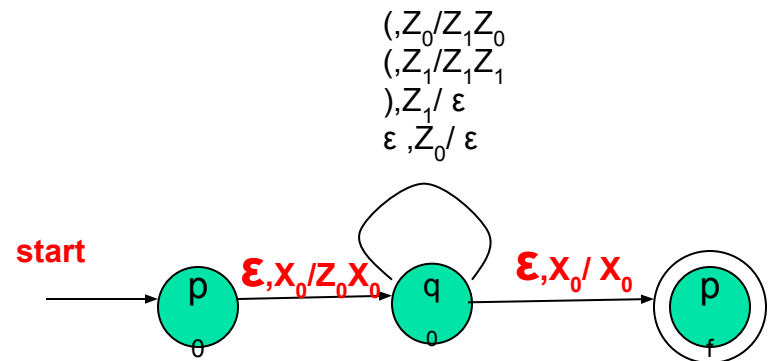
$\delta_N:$
 $\delta_N(q_0, (, Z_0) = \{ (q_0, Z_1 Z_0) \}$
 $\delta_N(q_0, (, Z_1) = \{ (q_0, Z_1 Z_1) \}$
 $\delta_N(q_0,), Z_1) = \{ (q_0, \epsilon) \}$
 $\delta_N(q_0, \epsilon, Z_0) = \{ (q_0, \epsilon) \}$



Accept by empty stack

$P_f: (\{p_0, q_0, p_f\}, \{(\,,)\}, \{X_0, Z_0, Z_1\}, \delta_f, p_0, X_0, p_f)$

$\delta_f:$
 $\delta_f(p_0, \epsilon, X_0) = \{ (q_0, Z_0) \}$
 $\delta_f(q_0, (, Z_0) = \{ (q_0, Z_1 Z_0) \}$
 $\delta_f(q_0, (, Z_1) = \{ (q_0, Z_1 Z_1) \}$
 $\delta_f(q_0,), Z_1) = \{ (q_0, \epsilon) \}$
 $\delta_f(q_0, \epsilon, Z_0) = \{ (q_0, \epsilon) \}$
 $\delta_f(p_0, \epsilon, X_0) = \{ (p_f, X_0) \}$



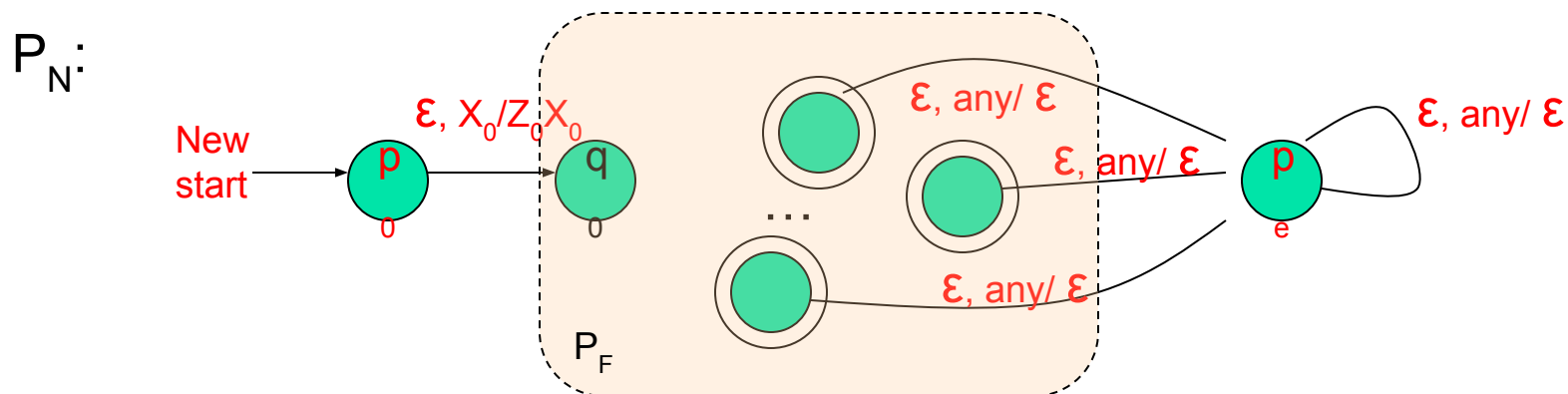
Accept by final state

$P_F \Rightarrow P_N$ construction

■ Main idea:

- Whenever P_F reaches a final state, just make an ϵ -transition into a new end state, clear out the stack and accept
- Danger: What if P_F design is such that it clears the stack midway *without* entering a final state?
 - to address this, add a new start symbol X_0 (not in Γ of P_F)

$$P_N = (Q \cup \{p_0, p_e\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

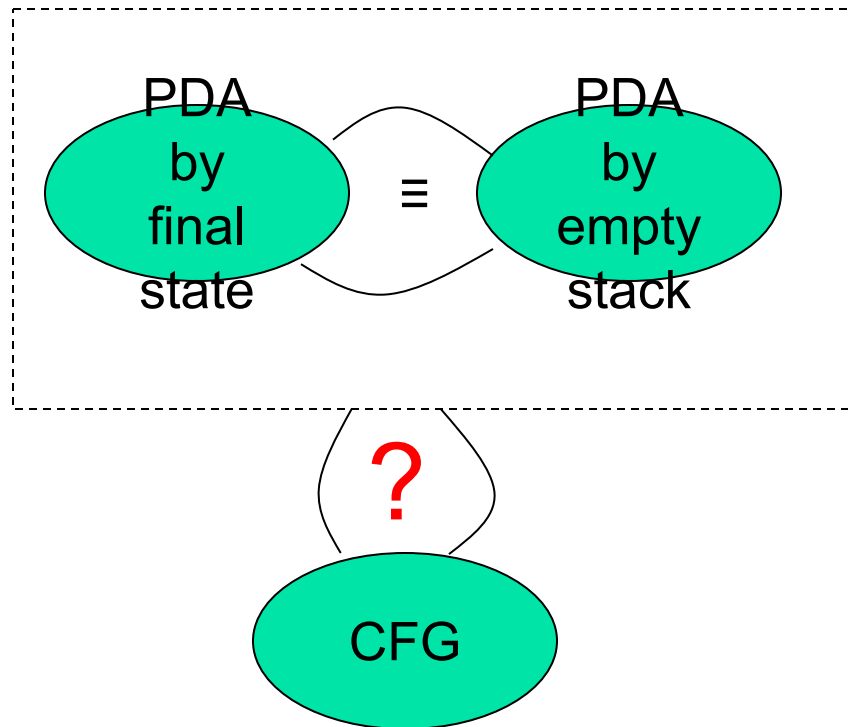


Equivalence of PDAs and CFGs





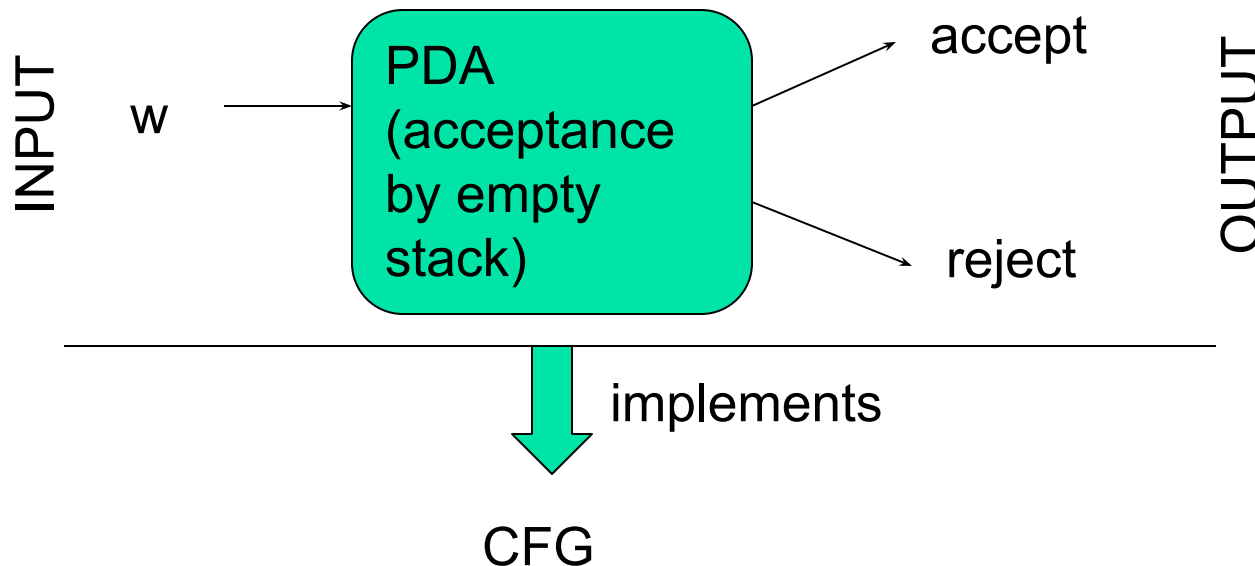
CFGs == PDAs ==> CFLs



This is same as: “implementing a CFG using a PDA”

Converting CFG to PDA

Main idea: The PDA simulates the leftmost derivation on a given w , and upon consuming it fully it either arrives at acceptance (by empty stack) or non-acceptance.

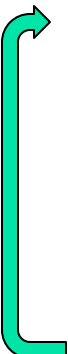




Converting a CFG into a PDA

Main idea: The PDA simulates the leftmost derivation on a given w , and upon consuming it fully it either arrives at acceptance (by empty stack) or non-acceptance.

Steps:

- 
1. Push the right hand side of the production onto the stack, with leftmost symbol at the stack top
 2. If stack top is the leftmost variable, then replace it by all its productions (each possible substitution will represent a distinct path taken by the non-deterministic PDA)
 3. If stack top has a terminal symbol, and if it matches with the next symbol in the input string, then pop it

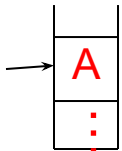
State is inconsequential (only one state is needed)

Formal construction of PDA from CFG

Note: Initial stack symbol (S) same as the start variable in the grammar

- Given: $G = (V, T, P, S)$
- Output: $P_N = (\{q\}, T, V \cup T, \delta, q, S)$
- δ :

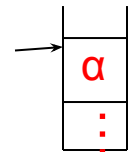
Before:



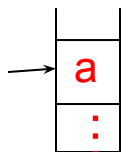
- For all $A \in V$, add the following transition(s) in the PDA:

- $\delta(q, \varepsilon, A) = \{ (q, \alpha) \mid "A \Rightarrow \alpha" \in P \}$

After:



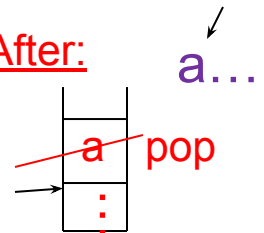
Before:



- For all $a \in T$, add the following transition(s) in the PDA:

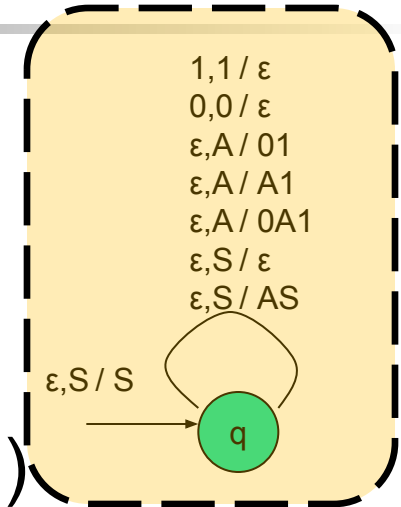
- $\delta(q, a, a) = \{ (q, \varepsilon) \}$

After:



Example: CFG to PDA

- $G = (\{S,A\}, \{0,1\}, P, S)$
- P :
 - $S \Rightarrow AS \mid \varepsilon$
 - $A \Rightarrow 0A1 \mid A1 \mid 01$
- $PDA = (\{q\}, \{0,1\}, \{0,1,A,S\}, \delta, q, S)$
- δ :
 - $\delta(q, \varepsilon, S) = \{ (q, AS), (q, \varepsilon) \}$
 - $\delta(q, \varepsilon, A) = \{ (q, 0A1), (q, A1), (q, 01) \}$
 - $\delta(q, 0, 0) = \{ (q, \varepsilon) \}$
 - $\delta(q, 1, 1) = \{ (q, \varepsilon) \}$



How will this new PDA work?

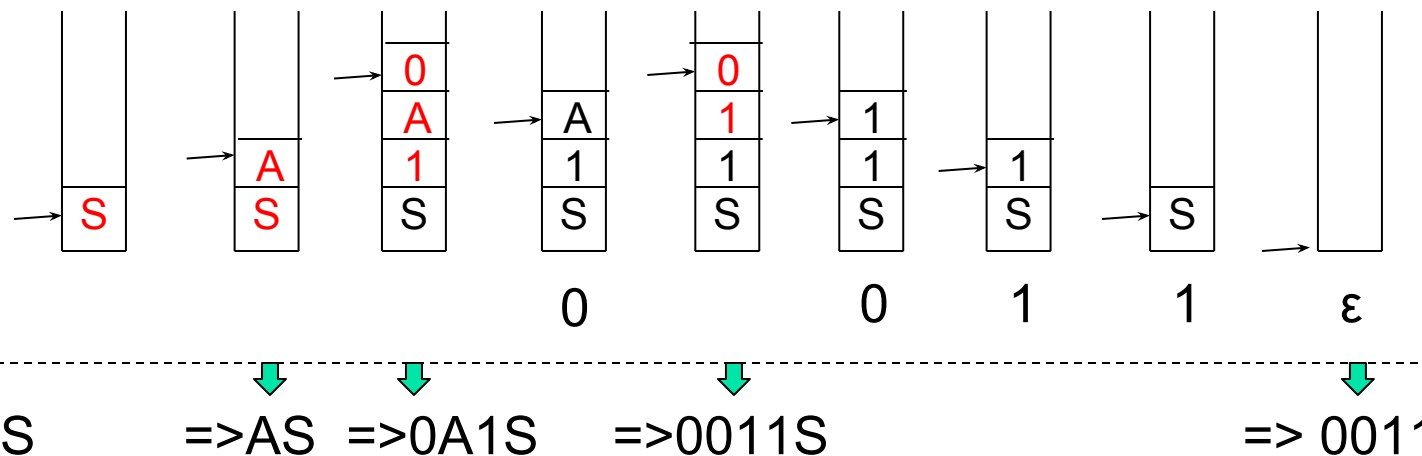
Lets simulate string 0011

Simulating string 0011 on the new PDA ...

PDA (δ):

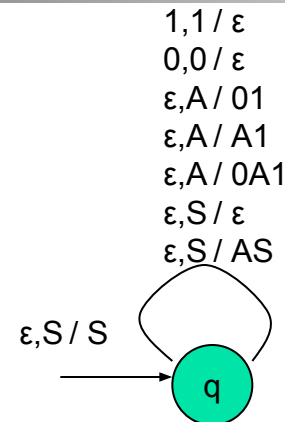
$\delta(q, \epsilon, S) = \{ (q, AS), (q, \epsilon) \}$
 $\delta(q, \epsilon, A) = \{ (q, 0A1), (q, A1), (q, 01) \}$
 $\delta(q, 0, 0) = \{ (q, \epsilon) \}$
 $\delta(q, 1, 1) = \{ (q, \epsilon) \}$

Stack moves (shows only the successful path):



Leftmost deriv.:

$$\begin{aligned}
 S &\Rightarrow AS \\
 &\Rightarrow 0A1S \\
 &\Rightarrow 0011S \\
 &\Rightarrow 0011
 \end{aligned}$$



Accept by empty stack



Proof of correctness for CFG \Rightarrow PDA construction

- Claim: A string is accepted by G iff it is accepted (by empty stack) by the PDA
- Proof:
 - *(only-if part)*
 - Prove by induction on the number of derivation steps
 - *(if part)*
 - If $(q, wx, S) \vdash^* (q, x, B)$ then $S \Rightarrow_{\text{Im}}^* wB$



Converting a PDA into a CFG

- Main idea: Reverse engineer the productions from transitions

If $\delta(q, a, Z) \Rightarrow (p, Y_1 Y_2 Y_3 \dots Y_k)$:

1. State is changed from q to p ;
2. Terminal a is consumed;
3. Stack top symbol Z is popped and replaced with a sequence of k variables.

- Action: Create a grammar variable called “[qZp]” which includes the following production:

- $[qZp] \Rightarrow a[pY_1q_1] [q_1Y_2q_2] [q_2Y_3q_3] \dots [q_{k-1}Y_kq_k]$

- Proof discussion (in the book) www.veerpreps.com

Example: Bracket matching

- To avoid confusion, we will use $b = "("$ and $e = ")"$

$P_N: (\{q_0\}, \{b, e\}, \{Z_0, Z_1\}, \delta, q_0, Z_0)$

1. $\delta(q_0, b, Z_0) = \{(q_0, Z_1, Z_0)\}$
2. $\delta(q_0, b, Z_1) = \{(q_0, Z_1, Z_1)\}$
3. $\delta(q_0, e, Z_1) = \{(q_0, \epsilon)\}$
4. $\delta(q_0, \epsilon, Z_0) = \{(q_0, \epsilon)\}$

0. $S \Rightarrow [q_0 Z_0 q_0]$
1. $[q_0 Z_0 q_0] \Rightarrow b [q_0 Z_1 q_0] [q_0 Z_0 q_0]$
2. $[q_0 Z_1 q_0] \Rightarrow b [q_0 Z_1 q_0] [q_0 Z_1 q_0]$
3. $[q_0 Z_1 q_0] \Rightarrow e$
4. $[q_0 Z_0 q_0] \Rightarrow \epsilon$

Let $A = [q_0 Z_0 q_0]$
Let $B = [q_0 Z_1 q_0]$

0. $S \Rightarrow A$
1. $A \Rightarrow b B A$
2. $B \Rightarrow b B B$
3. $B \Rightarrow e$
4. $A \Rightarrow \epsilon$

Simplifying,

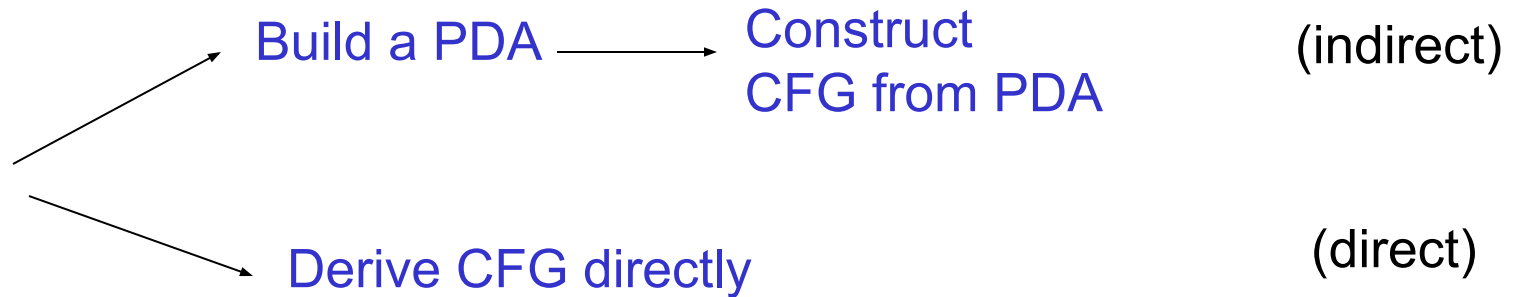
0. $S \Rightarrow b B S \mid \epsilon$
1. $B \Rightarrow b B B \mid e$

If you were to directly write a CFG:

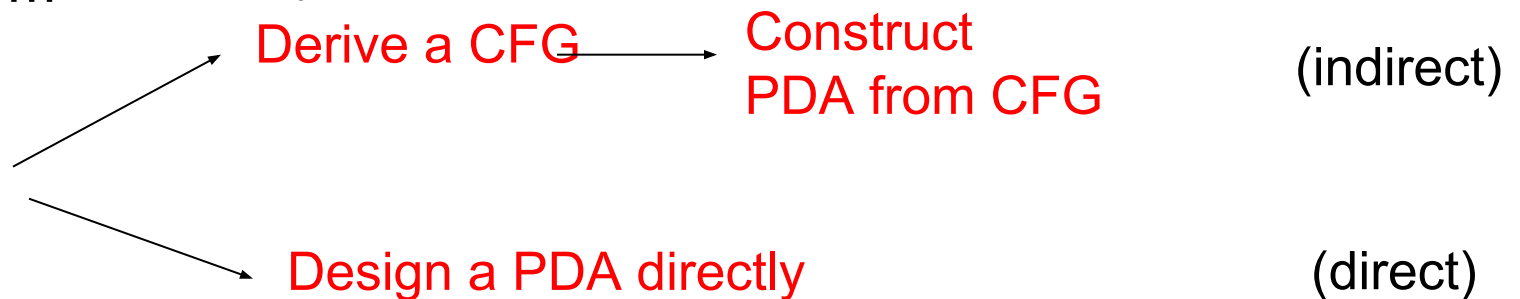
$S \Rightarrow b S e S \mid \epsilon$



Two ways to build a CFG



Similarly... Two ways to build a PDA





Deterministic PDAs

This PDA for L_{wwr} is non-deterministic

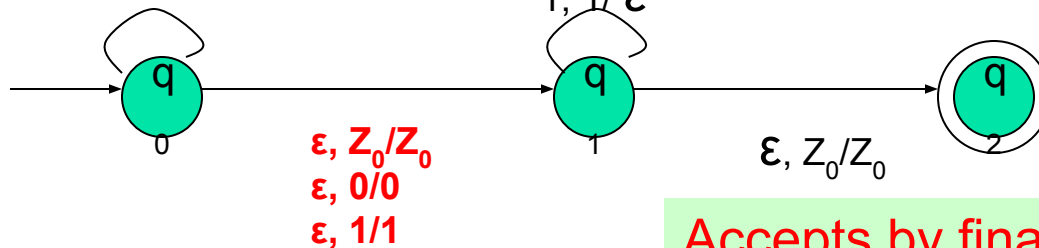
Grow stack

$0, Z_0/0Z_0$
 $1, Z_0/1Z_0$
 $0, 0/00$
 $0, 1/01$
 $1, 0/10$
 $1, 1/11$

Pop stack for
matching symbols

$0, 0/\epsilon$
 $1, 1/\epsilon$

Why does it have to
be
non-deterministic?



Switch to
popping mode

Accepts by final state

To remove
guessing,
impose the user
to insert c in the
middle

Example shows that: Nondeterministic PDAs \neq D-PDAs

D-PDA for $L_{wcw^R} = \{wcw^R \mid c \text{ is some special symbol not in } w\}$

Grow stack

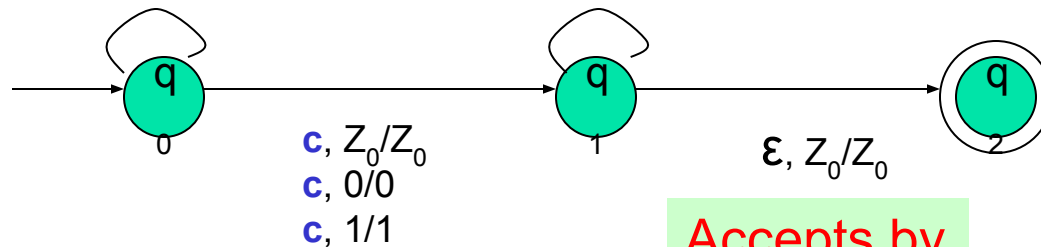
0, $Z_0/0Z_0$
1, $Z_0/1Z_0$
0, 0/00
0, 1/01
1, 0/10
1, 1/11

Pop stack for
matching symbols

0, 0/ ϵ
1, 1/ ϵ

Note:

- all transitions have become deterministic



Switch to
popping mode

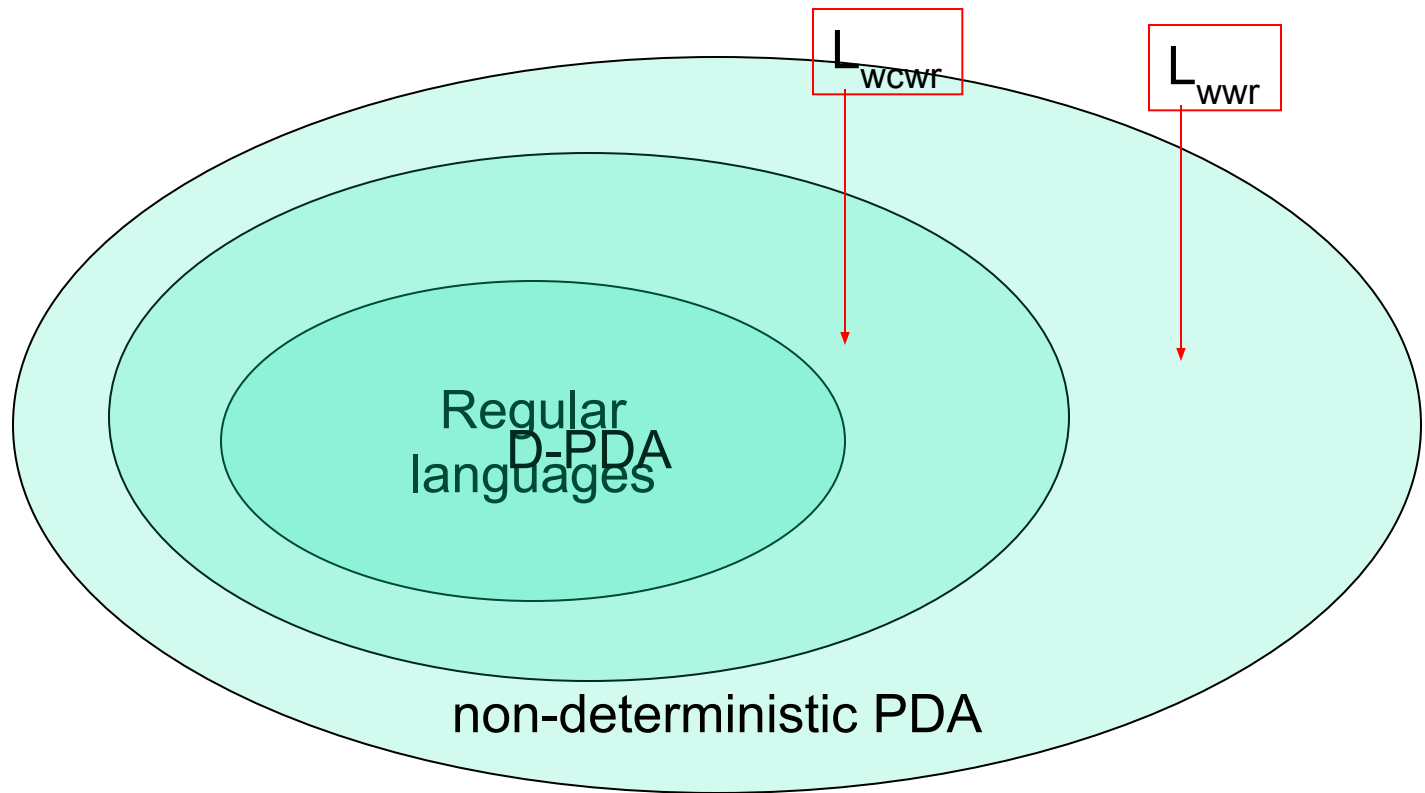
Accepts by
final state



Deterministic PDA: Definition

- A PDA is *deterministic* if and only if:
 1. $\delta(q, a, X)$ has *at most one* member for any $a \in \Sigma \cup \{\epsilon\}$
- If $\delta(q, a, X)$ is non-empty for some $a \in \Sigma$, then $\delta(q, \epsilon, X)$ must be empty.

PDA vs DPDA vs Regular languages





Summary

- PDAs for CFLs and CFGs
 - Non-deterministic
 - Deterministic
- PDA acceptance types
 1. By final state
 2. By empty stack
- PDA
 - IDs, Transition diagram
- Equivalence of CFG and PDA
 - $\text{CFG} \Rightarrow \text{PDA}$ construction
 - $\text{PDA} \Rightarrow \text{CFG}$ construction



Properties of Context-free Languages

Reading: Chapter 7



Topics

- 1) Simplifying CFGs, Normal forms
- 2) Pumping lemma for CFLs
- 3) Closure and decision properties of CFLs



How to “simplify” CFGs?



Three ways to simplify/clean a CFG

(clean)

1. Eliminate *useless symbols*

(simplify)

2. Eliminate ϵ -productions

$A \not\Rightarrow \epsilon$

3. Eliminate *unit productions*

$A \not\Rightarrow B$



Eliminating useless symbols

Grammar cleanup



Eliminating *useless symbols*

A symbol X is reachable if there exists:

- $S \xRightarrow{*} \alpha X \beta$

A symbol X is generating if there exists:

- $X \xRightarrow{*} w,$
 - for some $w \in T^*$

For a symbol X to be “useful”, it has to be both reachable *and* generating

- $S \xRightarrow{*} \alpha X \beta \xRightarrow{*} w', \text{ for some } w' \in T^*$

reachable generating



Algorithm to detect useless symbols

1. First, eliminate all symbols that are *not* generating
2. Next, eliminate all symbols that are *not* reachable

Is the order of these steps important,
or can we switch?



Example: Useless symbols

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

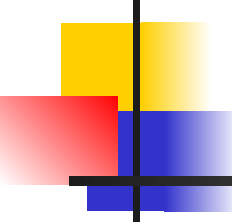
1. A, S are generating
2. B is *not generating* (and therefore B is useless)
3. \Rightarrow Eliminating B ... (i.e., remove all productions that involve B)
 1. $S \rightarrow a$
 2. $A \rightarrow b$
4. Now, A is *not reachable* and therefore is useless

5. Simplified G :

1. $S \rightarrow a$

What would happen if you reverse the order:
i.e., test reachability before generating?

Will fail to remove:
 $A \rightarrow b$


$$X \square^* w$$

Algorithm to find all generating symbols

- Given: $G=(V,T,P,S)$
- Basis:
 - Every symbol in T is obviously generating.
- Induction:
 - Suppose for a production $A \square \alpha$, where α is generating
 - Then, A is also generating

$$S \Rightarrow^* \alpha X \beta$$

Algorithm to find all reachable symbols

- Given: $G=(V,T,P,S)$
- Basis:
 - S is obviously reachable (from itself)
- Induction:
 - Suppose for a production $A \Rightarrow \alpha_1 \alpha_2 \dots \alpha_k$, where A is reachable
 - Then, all symbols on the right hand side, $\{\alpha_1, \alpha_2, \dots, \alpha_k\}$ are also reachable.



Eliminating ϵ -productions

$A \Rightarrow \epsilon$

X

What's the point of removing ϵ -productions?

$A \Rightarrow \epsilon$

Eliminating ϵ -productions

Caveat: It is *not* possible to eliminate ϵ -productions for languages which include ϵ in their word set

So we will target the grammar for the rest of the language

Theorem: If $G=(V,T,P,S)$ is a CFG for a language L , then $L \setminus \{\epsilon\}$ has a CFG without ϵ -productions

Definition: A is “nullable” if $A \Rightarrow^* \epsilon$

- If A is nullable, then any production of the form “ $B \Rightarrow CAD$ ” can be simulated by:
 - $B \Rightarrow CD \mid CAD$
 - This can allow us to remove ϵ transitions for A



Algorithm to detect all nullable variables

- Basis:
 - If $A \Rightarrow \varepsilon$ is a production in G , then A is nullable
(note: A can still have other productions)
- Induction:
 - If there is a production $B \Rightarrow C_1 C_2 \dots C_k$, where *every* C_i is nullable, then B is also nullable



Eliminating ϵ -productions

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

Algorithm:

1. Detect all nullable variables in G
2. Then construct $G_1=(V,T,P_1,S)$ as follows:
 - i. For each production of the form: $A \rightarrow X_1 X_2 \dots X_k$, where $k \geq 1$, suppose m out of the k X_i 's are nullable symbols
 - ii. Then G_1 will have 2^m versions for this production
 - i. i.e, all combinations where each X_i is either present or absent
 - iii. Alternatively, if a production is of the form: $A \rightarrow \epsilon$, then remove it

Example: Eliminating ϵ -productions

- Let L be the language represented by the following CFG G :

- i. $S \rightarrow AB$
- ii. $A \rightarrow aAA \mid \epsilon$
- iii. $B \rightarrow bBB \mid \epsilon$

Simplified
grammar

Goal: To construct G_1 , which is the grammar for $L - \{\epsilon\}$

- Nullable symbols: $\{A, B\}$
- G_1 can be constructed from G as follows:
 - $B \rightarrow b \mid bB \mid bB \mid bBB$
 - $\Rightarrow B \rightarrow b \mid bB \mid bBB$
 - Similarly, $A \rightarrow a \mid aA \mid aAA$
 - Similarly, $S \rightarrow A \mid B \mid AB$

- Note: $L(G) = L(G_1) \cup \{\epsilon\}$

G_1 :

- $S \rightarrow A \mid B \mid AB$
- $A \rightarrow a \mid aA \mid aAA$
- $B \rightarrow b \mid bB \mid bBB$

+

- $S \rightarrow \epsilon$

Eliminating unit productions

$A \Rightarrow B$

X

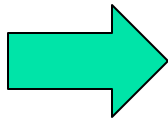
← B has to be a variable

What's the point of removing unit transitions ?

Will save #substitutions

E.g.,

```
A=>B | ...  
B=>C | ...  
C=>D | ...  
D=>xxx | yyy | zzz
```



```
A=>xxx | yyy | zzz | ...  
B=> xxx | yyy | zzz | ...  
C=> xxx | yyy | zzz | ...  
D=>xxx | yyy | zzz
```

before

after

$$A \Rightarrow B$$

Eliminating unit productions

- Unit production is one which is of the form $A \Rightarrow B$, where both A & B are variables

E.g.,

- $E \Rightarrow T \mid E+T$
- $T \Rightarrow F \mid T^*F$
- $F \Rightarrow I \mid (E)$
- $I \Rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

- How to eliminate unit productions?

- Replace $E \Rightarrow T$ with $E \Rightarrow F \mid T^*F$

- Then, upon recursive application wherever there is a unit production:

- $E \Rightarrow F \mid T^*F \mid E+T$ (substituting for T)
- $E \Rightarrow I \mid (E) \mid T^*F \mid E+T$ (substituting for F)
- $E \Rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T^*F \mid E+T$ (substituting for I)
- Now, E has no unit productions

- Similarly, eliminate for the remainder of the unit productions



The Unit Pair Algorithm: to remove unit productions

- Suppose $A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \dots \Rightarrow B_n \Rightarrow \alpha$
- Action: Replace all intermediate productions to produce α directly
 - i.e., $A \Rightarrow \alpha$; $B_1 \Rightarrow \alpha$; ... $B_n \Rightarrow \alpha$;

Definition: (A, B) to be a “**unit pair**” if $A \Rightarrow^* B$

- We can find all unit pairs inductively:
 - Basis: Every pair (A, A) is a unit pair (by definition). Similarly, if $A \Rightarrow B$ is a production, then (A, B) is a unit pair.
 - Induction: If (A, B) and (B, C) are unit pairs, and $A \Rightarrow C$ is also a unit pair.



The Unit Pair Algorithm: to remove unit productions

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

Goal: to build $G_1=(V,T,P_1,S)$ devoid of unit productions

Algorithm:

1. Find all unit pairs in G
2. For each unit pair (A,B) in G :
 1. Add to P_1 a new production $A \rightarrow \alpha$, for every $B \rightarrow \alpha$ which is a *non-unit* production
 2. If a resulting production is already there in P , then there is no need to add it.

Example: eliminating unit productions

G:

1. $E \rightarrow T \mid E+T$
2. $T \rightarrow F \mid T^*F$
3. $F \rightarrow I \mid (E)$
4. $I \rightarrow a \mid b \mid la \mid lb \mid l0 \mid l1$

Unit pairs

Only non-unit productions to be added to P_1

(E,E)	$E \rightarrow E+T$
(E,T)	$E \rightarrow T^*F$
(E,F)	$E \rightarrow (E)$
(E,I)	$E \rightarrow a \mid b \mid la \mid lb \mid l0 \mid l1$
(T,T)	$T \rightarrow T^*F$
(T,F)	$T \rightarrow (E)$
(T,I)	$T \rightarrow a \mid b \mid la \mid lb \mid l0 \mid l1$
(F,F)	$F \rightarrow (E)$
(F,I)	$F \rightarrow a \mid b \mid la \mid lb \mid l0 \mid l1$
(I,I)	$I \rightarrow a \mid b \mid la \mid lb \mid l0 \mid l1$

G_1 :

1. $E \rightarrow E+T \mid T^*F \mid (E) \mid a \mid b \mid la \mid lb \mid l0 \mid l1$
2. $T \rightarrow T^*F \mid (E) \mid a \mid b \mid la \mid lb \mid l0 \mid l1$
3. $F \rightarrow (E) \mid a \mid b \mid la \mid lb \mid l0 \mid l1$
4. $I \rightarrow a \mid b \mid la \mid lb \mid l0 \mid l1$



Putting all this together...

- Theorem: If G is a CFG for a language that contains at least one string other than ϵ , then there is another CFG G_1 , such that $L(G_1) = L(G) - \epsilon$, and G_1 has:
 - no ϵ -productions
 - no unit productions
 - no useless symbols
- Algorithm:
 - Step 1) eliminate ϵ -productions
 - Step 2) eliminate unit productions
 - Step 3) eliminate useless symbols

Again,
the order is
important!

Why?



Normal Forms



Why normal forms?

- If all productions of the grammar could be expressed in the same form(s), then:
 - a. It becomes easy to design algorithms that use the grammar
 - b. It becomes easy to show proofs and properties



Chomsky Normal Form (CNF)

Let G be a CFG for some $L - \{\epsilon\}$

Definition:

G is said to be in **Chomsky Normal Form** if all its productions are in one of the following two forms:

i. $A \rightarrow BC$ where A, B, C are variables, or

ii. $A \rightarrow a$ where a is a terminal

- G has no useless symbols
- G has no unit productions
- G has no ϵ -productions



CNF checklist

Is this grammar in CNF?

G_4 :

1. $E \rightarrow E+T \mid T^*F \mid (E) \mid Ia \mid Ib \mid I0 \mid I1$
2. $T \rightarrow T^*F \mid (E) \mid Ia \mid Ib \mid I0 \mid I1$
3. $F \rightarrow (E) \mid Ia \mid Ib \mid I0 \mid I1$
4. $I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Checklist:

- G has no ϵ -productions ✓
- G has no unit productions ✓
- G has no useless symbols ✓
- But...
 - the normal form for productions is violated



So, the grammar is not in CNF

How to convert a G into CNF?

- Assumption: G has no ϵ -productions, unit productions or useless symbols
- 1) For every terminal a that appears in the body of a production:
 - i. create a unique variable, say X_a , with a production $X_a \rightarrow a$, and
 - ii. replace all other instances of a in G by X_a
- 2) Now, all productions will be in one of the following two forms:
 - $A \rightarrow B_1 B_2 \dots B_k$ ($k \geq 3$) or $A \rightarrow a$
- 4) Replace each production of the form $A \rightarrow B_1 B_2 B_3 \dots B_k$ by:

$$\begin{array}{c}
 B_1 \quad B_2 \quad B_3 \quad \dots \quad B_k \\
 \xleftarrow{C_1} \quad \xrightarrow{C_2}
 \end{array}$$

and so on...

 - $A \rightarrow B_1 C_1 \quad C_1 \rightarrow B_2 C_2 \quad \dots \quad C_{k-3} \rightarrow B_{k-2} C_{k-2} \quad C_{k-2} \rightarrow B_{k-1} B_k$

Example #1

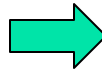
G:

$S \Rightarrow AS \mid BABC$

$A \Rightarrow A1 \mid 0A1 \mid 01$

$B \Rightarrow 0B \mid 0$

$C \Rightarrow 1C \mid 1$



G in CNF:

$X_0 \Rightarrow 0$

$X_1 \Rightarrow 1$

$S \Rightarrow AS \mid BY_1$

$Y_1 \Rightarrow$

AY_2

$A \Rightarrow BC \mid X_0Y_3 \mid X_0X_1$

$Y_3 \Rightarrow AX_1$

$B \Rightarrow X_0B \mid 0$

$C \Rightarrow X_1C \mid 1$

All productions are of the form: $A \Rightarrow BC$ or $A \Rightarrow a$

Example #2

G:

1. $E \sqsubseteq E+T \mid T^*F \mid (E) \mid I_a \mid I_b \mid I_0 \mid I_1$
2. $T \sqsubseteq T^*F \mid (E) \mid I_a \mid I_b \mid I_0 \mid I_1$
3. $F \sqsubseteq (E) \mid I_a \mid I_b \mid I_0 \mid I_1$
4. $I \sqsubseteq a \mid b \mid I_a \mid I_b \mid I_0 \mid I_1$

Step (1)

1. $E \sqsubseteq EX_+T \mid TX_*F \mid X(EX_*) \mid IX_a \mid IX_b \mid IX_0 \mid IX_1$
2. $T \sqsubseteq TX_*F \mid X(EX_*) \mid IX_a \mid IX_b \mid IX_0 \mid IX_1$
3. $F \sqsubseteq X(EX_*) \mid IX_a \mid IX_b \mid IX_0 \mid IX_1$
4. $I \sqsubseteq X_a \mid X_b \mid IX_a \mid IX_b \mid IX_0 \mid IX_1$
5. $X_+ \sqsubseteq +$
6. $X_* \sqsubseteq *$
7. $X_+ \sqsubseteq +$
8. $X_() \sqsubseteq ($
9.

Step (2)

1. $E \sqsubseteq EC_1 \mid TC_2 \mid X(C_3) \mid IX_a \mid IX_b \mid IX_0 \mid IX_1$
2. $C_1 \sqsubseteq X_+T$
3. $C_2 \sqsubseteq X_*F$
4. $C_3 \sqsubseteq EX_()$
5. $T \sqsubseteq \dots\dots\dots$
6.

Languages with ϵ

- For languages that include ϵ ,
 - Write down the rest of grammar in CNF
 - Then add production “ $S \Rightarrow \epsilon$ ” at the end

E.g., consider:

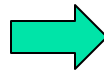
G:

$S \Rightarrow AS \mid BABC$

$A \Rightarrow A1 \mid 0A1 \mid 01 \mid \epsilon$

$B \Rightarrow 0B \mid 0 \mid \epsilon$

$C \Rightarrow 1C \mid 1 \mid \epsilon$



G in CNF:

$X_0 \Rightarrow 0$

$X_1 \Rightarrow 1$

$S \Rightarrow AS \mid BY_1 \mid \epsilon$

$Y_1 \Rightarrow$

AY_2

$Y_2 \Rightarrow BC$

$A \Rightarrow AX_1 \mid X_0Y_3 \mid X_0X_1$

$Y_3 \Rightarrow AX_1$

$B \Rightarrow X_0B \mid 0$

$C \Rightarrow X_1C \mid 1$



Other Normal Forms

- Griebach Normal Form (GNF)
 - All productions of the form

$$A \Rightarrow a \alpha$$



Return of the Pumping Lemma !!

Think of languages that cannot be CFL

== think of languages for which a stack will not be enough

e.g., the language of strings of the form ww



Why pumping lemma?

- A result that will be useful in proving languages that *are not* CFLs
 - (just like we did for regular languages)
- But before we prove the pumping lemma for CFLs
 - Let us first prove an important property about parse trees

Observe that any parse tree generated by a CNF will be a binary tree, where all internal nodes have exactly two children (except those nodes connected to the leaves).

The “parse tree theorem”

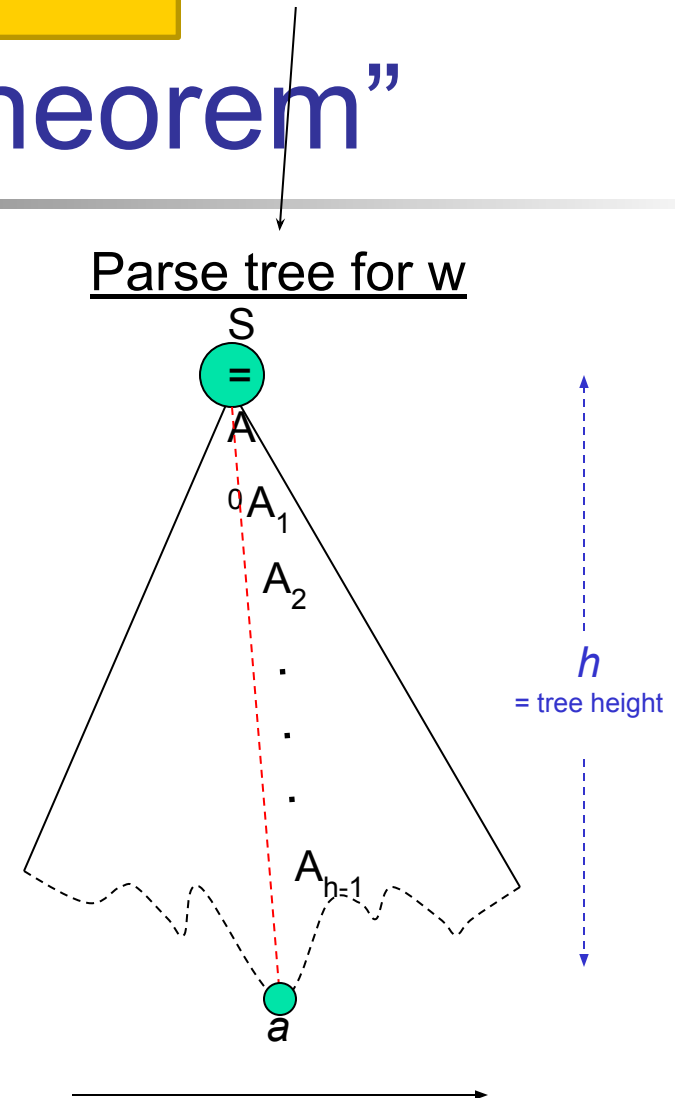
Given:

- Suppose we have a parse tree for a string w , according to a CNF grammar, $G=(V,T,P,S)$
- Let h be the height of the parse tree

Implies:

- $|w| \leq 2^{h-1}$

In other words, a CNF parse tree's string yield (w) can no longer be 2^{h-1}



To show: $|w| \leq 2^{h-1}$

Proof...The size of parse trees

Proof: (using induction on h)

Basis: $h = 1$

- Derivation will have to be " $S \Rightarrow a$ "
- $|w| = 1 = 2^{1-1}$.

Ind. Hyp: $h = k-1$

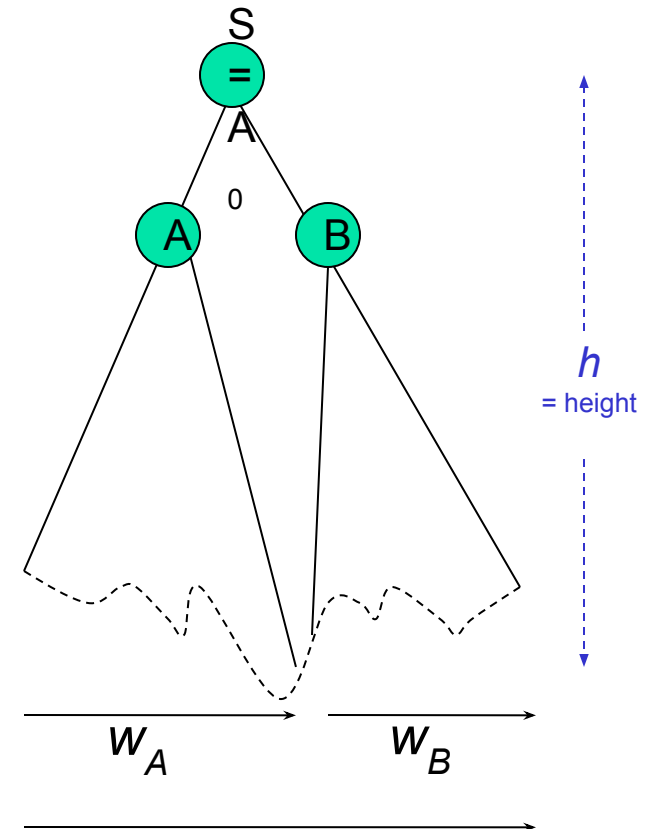
- $|w| \leq 2^{k-2}$

Ind. Step: $h = k$

S will have exactly two children:
 $S \Rightarrow AB$

- Heights of A & B subtrees are at most $h-1$
- $w = w_A w_B$, where $|w_A| \leq 2^{k-2}$ and $|w_B| \leq 2^{k-2}$
- $|w| \leq 2^{k-1}$

Parse tree for w





Implication of the Parse Tree Theorem (assuming CNF)

Fact:

- If the height of a parse tree is h , then
 - $\implies |w| \leq 2^{h-1}$

Implication:

- If $|w| \geq 2^m$, then
 - Its parse tree's height is *at least* $m+1$



The Pumping Lemma for CFLs

Let L be a CFL.

Then there exists a constant N , s.t.,

- if $z \in L$ s.t. $|z| \geq N$, then we can write $z = uvwx y$, such that:

1. $|vwx| \leq N$
2. $vx \neq \epsilon$
3. For all $k \geq 0$: $uv^kwx^ky \in L$

Note: we are pumping in two places (v & x)



Proof: Pumping Lemma for CFL

- If $L = \Phi$ or contains only ϵ , then the lemma is trivially satisfied (as it cannot be violated)

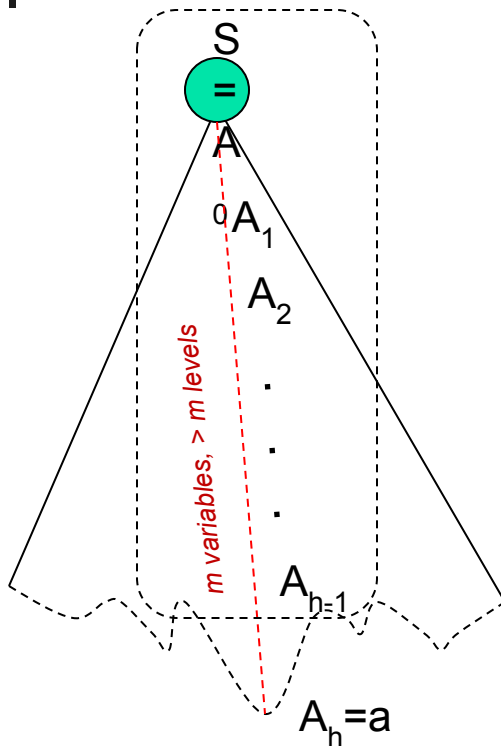
- For any other L which is a CFL:
 - Let G be a CNF grammar for L
 - Let m = number of variables in G
 - Choose $N = 2^m$.
 - Pick any $z \in L$ s.t. $|z| \geq N$
 - the parse tree for z should have a height $\geq m+1$
(by the parse tree theorem)

Parse tree for z

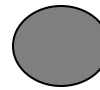
Meaning:

Repetition in the last $m+1$ variables

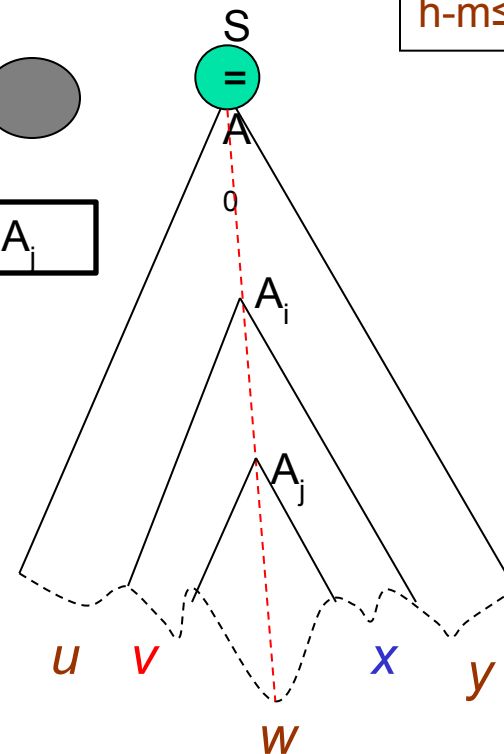
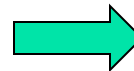
$$h-m \leq i < j \leq h$$



+



$$A_i = A_j$$



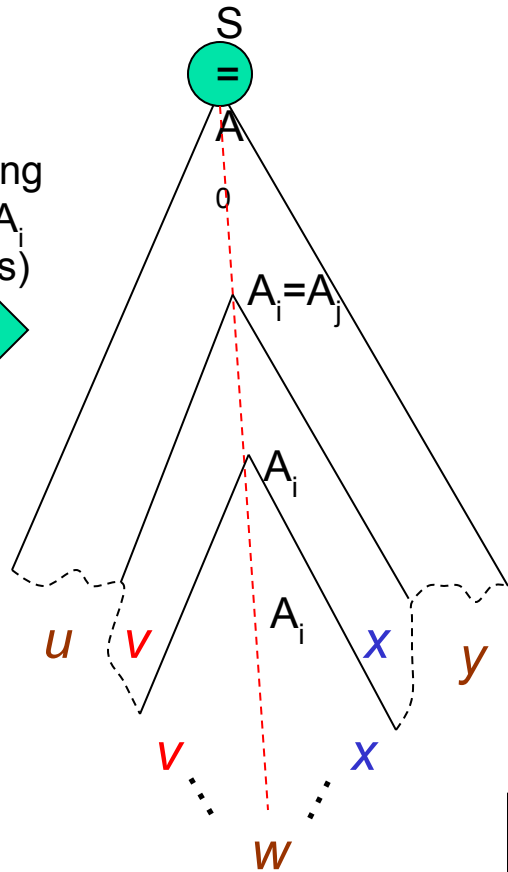
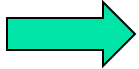
$m+1$

$$z = uvwx y$$

- Therefore $vx \neq \epsilon$

Extending the parse tree...

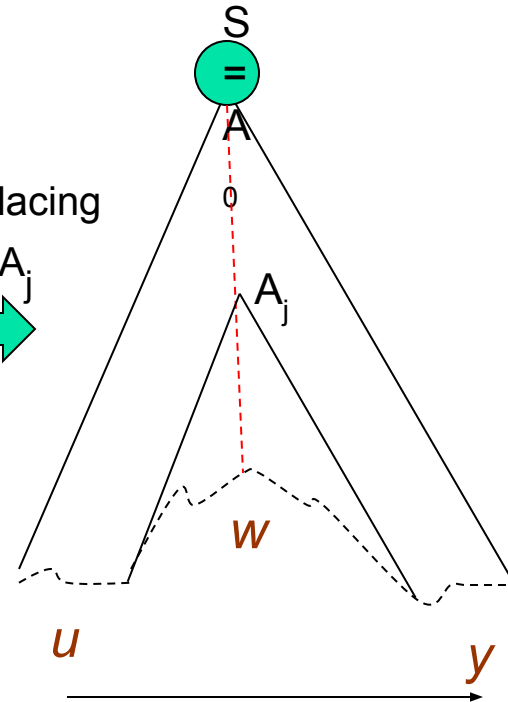
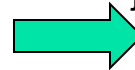
Replacing A_i with A_i (k times)



$$z = uv^kwx^ky$$

$h \geq m+1$

Or, replacing A_i with A_j



$$z = uwy$$

\Rightarrow

For all $k \geq 0$: $uv^kwx^ky \in L$



Proof contd..

- Also, since A_i 's subtree no taller than $m+1$

\Rightarrow the string generated under A_i 's subtree, which is vwx , cannot be longer than $2^m (=N)$

But, $2^m = N$

$\Rightarrow |vwx| \leq N$

This completes the proof for the pumping lemma.



Application of Pumping Lemma for CFLs

Example 1: $L = \{a^m b^m c^m \mid m > 0\}$

Claim: L is not a CFL

Proof:

- Let $N \leq P/L$ constant
- Pick $z = a^N b^N c^N$
- Apply pumping lemma to z and show that there exists at least one other string constructed from z (obtained by pumping up or down) that is $\notin L$



Proof contd...

- $z = uvwxy$
- As $z = a^N b^N c^N$ and $|vwx| \leq N$ and $vx \neq \epsilon$
 - $\implies v, x$ cannot contain all three symbols (a, b, c)
 - \implies we can pump up or pump down to build another string which is $\notin L$



Example #2 for P/L application

- $L = \{ ww \mid w \text{ is in } \{0,1\}^* \}$
- Show that L is not a CFL
 - Try string $z = 0^N 0^N$
 - what happens?
 - Try string $z = 0^N 1^N 0^N 1^N$
 - what happens?



Example 3

- $L = \{ 0^{k^2} \mid k \text{ is any integer} \}$
- Prove L is not a CFL using Pumping Lemma



Example 4

- $L = \{a^i b^j c^k \mid i < j < k\}$
- Prove that L is not a CFL



CFL Closure Properties



Closure Property Results

- CFLs are closed under:
 - Union
 - Concatenation
 - Kleene closure operator
 - Substitution
 - Homomorphism, inverse homomorphism
 - reversal

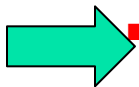
-
- CFLs are *not* closed under:
 - Intersection
 - Difference
 - Complementation

Note: Reg languages
are closed
under
these
operators

Strategy for Closure Property Proofs

- First prove “closure under **substitution**”
- Using the above result, prove other closure properties
- CFLs are closed under:
 - Union
 - Concatenation
 - Kleene closure operator
 - Substitution
 - Homomorphism, inverse homomorphism
 - Reversal

Prove
this first



Note: $s(L)$ can use a different alphabet

The *Substitution* operation

For each $a \in \Sigma$, then let $s(a)$ be a language
If $w = a_1 a_2 \dots a_n \in L$, then:

$$\bullet s(w) = \{x_1 x_2 \dots\} \in s(L), \text{ s.t., } x_i \in s(a_i)$$

Example:

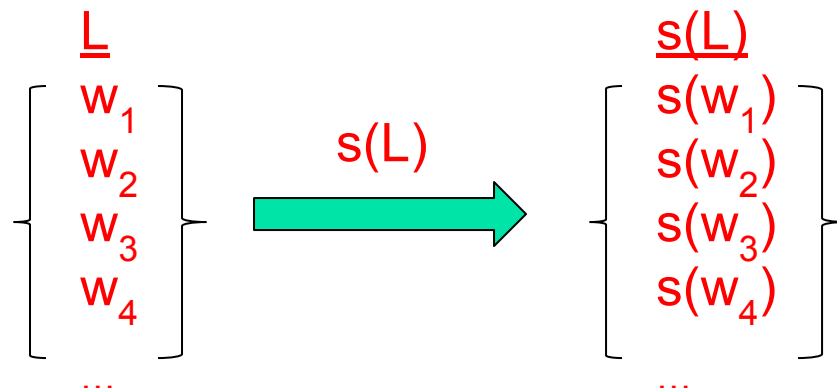
- Let $\Sigma = \{0, 1\}$
- Let: $s(0) = \{a^n b^n \mid n \geq 1\}$, $s(1) = \{aa, bb\}$
- If $w = 01$, $s(w) = s(0).s(1)$
 - E.g., $s(w)$ contains $a^1 b^1 aa$, $a^1 b^1 bb$,
 $a^2 b^2 aa$, $a^2 b^2 bb$,
... and so on.

CFLs are closed under Substitution

IF L is a CFL and a substitution defined on L , $s(L)$, is s.t., $s(a)$ is a CFL for every symbol a , THEN:

- $s(L)$ is also a CFL

What is $s(L)$?

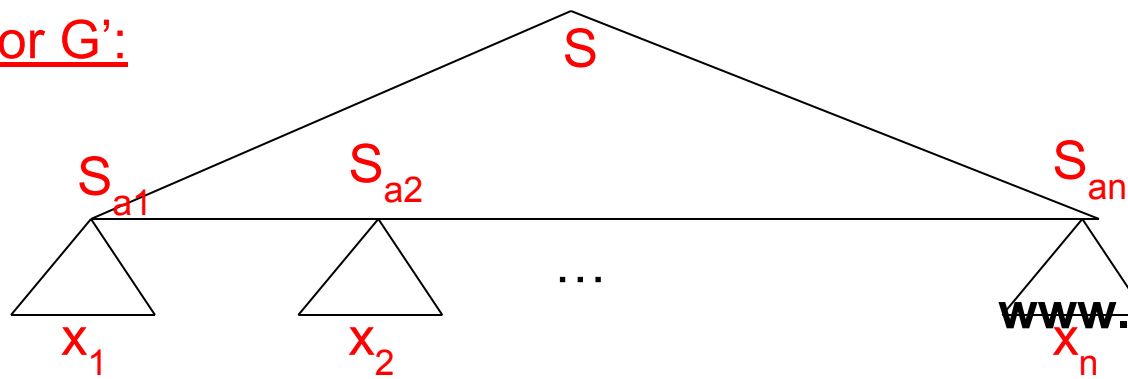


Note: each $s(w)$ is itself a set of strings

CFLs are closed under *Substitution*

- $G=(V,T,P,S)$: CFG for L
- Because every $s(a)$ is a CFL, there is a CFG for each $s(a)$
 - Let $G_a = (V_a, T_a, P_a, S_a)$
- Construct $G'=(V',T',P',S)$ for $s(L)$
- P' consists of:
 - The productions of P , but with every occurrence of terminal “ a ” in their bodies replaced by S_a .
 - All productions in any P_a , for any $a \in \Sigma$

Parse tree for G' :



Substitution of a CFL: example

- Let L = language of binary palindromes s.t., substitutions for 0 and 1 are defined as follows:
 - $s(0) = \{a^n b^n \mid n \geq 1\}$, $s(1) = \{xx, yy\}$
- Prove that $s(L)$ is also a CFL.

CFG for L :

$S \Rightarrow 0S0 \mid 1S1 \mid \epsilon$

CFG for $s(0)$:

$S_0 \Rightarrow aS_0b \mid ab$

CFG for $s(1)$:

$S_1 \Rightarrow xx \mid yy$



Therefore, CFG for $s(L)$:

$S \Rightarrow S_0 S S_0 \mid S_1 S S_1 \mid \epsilon$

$S_0 \Rightarrow aS_0b \mid ab$

$S_1 \Rightarrow xx \mid yy$



CFLs are closed under *union*

Let L_1 and L_2 be CFLs

To show: $L_1 \cup L_2$ is also a CFL

Let us show by using the result of *Substitution*

- Make a new language:

- $L_{\text{new}} = \{a, b\}$ s.t., $s(a) = L_1$ and $s(b) = L_2$
 $\implies s(L_{\text{new}}) == \text{same as} == L_1 \cup L_2$



- A more direct, alternative proof

- Let S_1 and S_2 be the starting variables of the grammars for L_1 and L_2
 - Then, $S_{\text{new}} \Rightarrow S_1 \mid S_2$



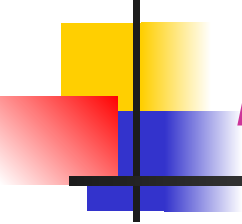
CFLs are closed under *concatenation*

- Let L_1 and L_2 be CFLs

Let us show by using the result of *Substitution*

- Make $L_{\text{new}} = \{ab\}$ s.t.,
 $s(a) = L_1$ and $s(b) = L_2$
 $\implies L_1 L_2 = s(L_{\text{new}})$

-
- A proof without using substitution?



CFLs are closed under *Kleene Closure*

- Let L be a CFL
- Let $L_{\text{new}} = \{a\}^*$ and $s(a) = L_1$
 - Then, $L^* = s(L_{\text{new}})$

CFLs are closed under *Reversal*

- Let L be a CFL, with grammar $G=(V,T,P,S)$
- For L^R , construct $G^R=(V,T,P^R,S)$ s.t.,
 - If $A \Rightarrow \alpha$ is in P , then:
 - $A \Rightarrow \alpha^R$ is in P^R
 - (that is, reverse every production)

CFLs are *not* closed under Intersection

- Existential proof:
 - $L_1 = \{0^n 1^n 2^i \mid n \geq 1, i \geq 1\}$
 - $L_2 = \{0^i 1^n 2^n \mid n \geq 1, i \geq 1\}$
- Both L_1 and L_2 are CFLs
 - Grammars?
- But $L_1 \cap L_2$ *cannot* be a CFL
 - Why?
- We have an example, where intersection is not closed.
- Therefore, CFLs are not closed under intersection

CFLs are not closed under complementation

- Follows from the fact that CFLs are not closed under intersection

- $$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Logic: if CFLs were to be closed under complementation

- the whole right hand side becomes a CFL (because CFL is closed for union)
- the left hand side (intersection) is also a CFL
- but we just showed CFLs are NOT closed under intersection!
- CFLs cannot be closed under complementation.



CFLs are not closed under difference

- Follows from the fact that CFLs are not closed under complementation
- Because, if CFLs are closed under difference, then:
 - $\bar{L} = \Sigma^* - L$
 - So \bar{L} has to be a CFL too
 - Contradiction



Decision Properties

- Emptiness test
 - Generating test
 - Reachability test
- Membership test
 - PDA acceptance



“Undecidable” problems for CFL

- Is a given CFG G ambiguous?
- Is a given CFL inherently ambiguous?
- Is the intersection of two CFLs empty?
- Are two CFLs the same?
- Is a given $L(G)$ equal to Σ^* ?



Summary

- Normal Forms
 - Chomsky Normal Form
 - Greibach Normal Form
 - Useful in proving P/L
- Pumping Lemma for CFLs
 - Main difference: $z = uv^iwx^iy$
- Closure properties
 - Closed under: union, concatenation, reversal, Kleen closure, homomorphism, substitution
 - Not closed under: intersection, complementation, difference

Simplification of CFG and Normal Forms

Dr. Kishore Kumar Sahu,
Assistant Professor, CSE,
VSSUT, Burla.

SIMPLIFICATION OF CONTEXT-FREE GRAMMARS

Consider, for example,

$$G = (\{S, A, B, C, E\}, \{a, b, c\}, P, S)$$

where

$$P = \{S \rightarrow AB, A \rightarrow a, B \rightarrow b, B \rightarrow C, E \rightarrow c \mid \Lambda\}$$

following points regarding the symbols and productions which are eliminated:

- (i) C does not derive any terminal string.
- (ii) E and c do not appear in any sentential form.
- (iii) $E \rightarrow \Lambda$ is a null production.
- (iv) $B \rightarrow C$ simply replaces B by C .

VARIABLE NOT DERIVING TERMINAL STRING

Let $G = (V_N, \Sigma, P, S)$ be given by the productions $S \rightarrow AB$, $A \rightarrow a$, $B \rightarrow b$, $B \rightarrow C$, $E \rightarrow c$. Find G' such that every variable in G' derives some terminal string.

Solution

(a) *Construction of V'_N :*

$W_1 = \{A, B, E\}$ since $A \rightarrow a$, $B \rightarrow b$, $E \rightarrow c$ are productions with a terminal string on the R.H.S.

$$\begin{aligned} W_2 &= W_1 \cup \{A_1 \in V_N \mid A_1 \rightarrow \alpha \text{ for some } \alpha \in (\Sigma \cup \{A, B, E\})^*\} \\ &= W_1 \cup \{S\} = \{A, B, E, S\} \end{aligned}$$

$$\begin{aligned} W_3 &= W_2 \cup \{A_1 \in V_N \mid A_1 \rightarrow \alpha \text{ for some } \alpha \in (\Sigma \cup \{S, A, B, E\})^*\} \\ &= W_2 \cup \emptyset = W_2 \end{aligned}$$

Therefore,

$$V'_N = \{S, A, B, F\}$$

VARIABLE NOT DERIVING TERMINAL STRING

(b) *Construction of P' :*

$$\begin{aligned} P' &= \{A_1 \rightarrow \alpha \mid A_1, \alpha \in (V'_N \cup \Sigma)^*\} \\ &= \{S \rightarrow AB, A \rightarrow a, B \rightarrow b, E \rightarrow c\} \end{aligned}$$

Therefore,

$$G' = (\{S, A, B, E\}, \{a, b, c\}, P', S)$$

SYMBOLS NOT APPEARING IN SENTENTIAL FORM

Consider $G = (\{S, A, B, E\}, \{a, b, c\}, P, S)$, where P consists of $S \rightarrow AB$, $A \rightarrow a$, $B \rightarrow b$, $E \rightarrow c$.

Solution

$$W_1 = \{S\}$$

$$W_2 = \{S\} \cup \{X \in V_N \cup \Sigma \mid \text{there exists a production } A \rightarrow \alpha \text{ with } A \in W_1 \text{ and } \alpha \text{ containing } X\}$$

$$= \{S\} \cup \{A, B\}$$

$$W_3 = \{S, A, B\} \cup \{a, b\}$$

$$W_4 = W_3$$

$$V'_N = \{S, A, B\} \quad \Sigma' = \{a, b\}$$

$$P' = \{S \rightarrow AB, A \rightarrow a, B \rightarrow b\}$$

Thus the required grammar is $G' = (V'_N, \Sigma', P', S)$.

Example 01

Find a reduced grammar equivalent to the grammar G whose productions are

$$S \rightarrow AB|CA, \quad B \rightarrow BC|AB, \quad A \rightarrow a, \quad C \rightarrow aB|b$$

Solution

Step 1 $W_1 = \{A, C\}$ as $A \rightarrow a$ and $C \rightarrow b$ are productions with a terminal string on R.H.S.

$$\begin{aligned} W_2 &= \{A, C\} \cup \{A_1 \mid A_1 \rightarrow \alpha \text{ with } \alpha \in (\Sigma \cup \{A, C\})^*\} \\ &= \{A, C\} \cup \{S\} \text{ as we have } S \rightarrow CA \end{aligned}$$

$$\begin{aligned} W_3 &= \{A, C, S\} \cup \{A_1 \mid A_1 \rightarrow \alpha \text{ with } \alpha \in (\Sigma \cup \{S, A, C\})^*\} \\ &= \{A, C, S\} \cup \emptyset \end{aligned}$$

As $W_3 = W_2$,

$$V'_N = W_2 = \{S, A, C\}$$

$$\begin{aligned} P' &= \{A_1 \rightarrow \alpha \mid A_1, \alpha \in (V'_N \cup \Sigma)^*\} \\ &= \{S \rightarrow CA, A \rightarrow a, C \rightarrow b\} \end{aligned}$$

Thus,

$$G_1 = (\{S, A, C\}, \{a, b\}, \{S \rightarrow CA, A \rightarrow a, C \rightarrow b\}, S)$$

Example 01

Step 2 We have to apply Theorem 6.4 to G_1 . Thus,

$$W_1 = \{S\}$$

As we have production $S \rightarrow CA$ and $S \in W_1$, $W_2 = \{S\} \cup \{A, C\}$

As $A \rightarrow a$ and $C \rightarrow b$ are productions with $A, C \in W_2$, $W_3 = \{S, A, C, a, b\}$

$$\text{As } W_3 = V'_N \cup \Sigma, P'' = \{S \rightarrow a \mid A_1 \in W_3\} = P'$$

Therefore,

$$G' = (\{S, A, C\}, \{a, b\}, \{S \rightarrow CA, A \rightarrow a, C \rightarrow b\}, S)$$

is the reduced grammar.

Example 02

Construct a reduced grammar equivalent to the grammar

$$\begin{aligned} S &\rightarrow aAa, & A &\rightarrow Sb \mid bCC \mid DaA, & C &\rightarrow abb \mid DD, \\ E &\rightarrow aC, & D &\rightarrow aDA \end{aligned}$$

Solution

Step 1 $W_1 = \{C\}$ as $C \rightarrow abb$ is the only production with a terminal string on the R.H.S.

$$W_2 = \{C\} \cup \{E, A\}$$

as $E \rightarrow aC$ and $A \rightarrow bCC$ are productions with R.H.S. in $(\Sigma \cup \{C\})^*$

$$W_3 = \{C, E, A\} \cup \{S\}$$

as $S \rightarrow aAa$ and aAa is in $(\Sigma \cup W_2)^*$

$$W_4 = W_3 \cup \emptyset$$

Hence,

$$V'_N = W_3 = \{S, A, C, E\}$$

$$P' = \{A_1 \rightarrow \alpha \mid \alpha \in (V'_N \cup \Sigma)^*\}$$

$$= \{S \rightarrow aAa, A \rightarrow Sb \mid bCC, C \rightarrow abb, E \rightarrow aC\}$$

$$G_1 = (V'_N, \{a, b\}, P', S)$$

Example 02

Step 2 We have to apply Theorem 6.4 to G_1 . We start with

$$W_1 = \{S\}$$

As we have $S \rightarrow aAa$,

$$W_2 = \{S\} \cup \{A, a\}$$

As $A \rightarrow Sb \mid bCC$,

$$W_3 = \{S, A, a\} \cup \{S, b, C\} = \{S, A, C, a, b\}$$

As we have $C \rightarrow abb$,

Hence,

$$\begin{aligned} P'' &= \{A_1 \rightarrow \alpha \mid A_1 \in W_3\} \\ &= \{S \rightarrow aAa, A \rightarrow Sb \mid bCC, C \rightarrow abb\} \end{aligned}$$

Therefore,

$$G' = (\{S, A, C\}, \{a, b\}, P'', S)$$

is the reduced grammar.

ELIMINATION OF NULL PRODUCTION

Consider the grammar G whose productions are $S \rightarrow aS \mid AB$, $A \rightarrow \Lambda$, $B \rightarrow \Lambda$, $D \rightarrow b$. Construct a grammar G_1 without null productions generating $L(G) - \{\Lambda\}$.

Solution

Step 1 *Construction of the set W of all nullable variables:*

$$\begin{aligned} W_1 &= \{A_1 \in V_N \mid A_1 \rightarrow \varepsilon \text{ is a production in } G\} \\ &= \{A, B\} \end{aligned}$$

$$\begin{aligned} W_2 &= \{A, B\} \cup \{S\} \text{ as } S \rightarrow AB \text{ is a production with } AB \in W_1^* \\ &= \{S, A, B\} \end{aligned}$$

$$W_3 = W_2 \cup \emptyset = W_2$$

Thus,

$$W = W_2 = \{S, A, B\}$$

ELIMINATION OF NULL PRODUCTION

Step 2 *Construction of P' :*

- (i) $D \rightarrow b$ is included in P' .
- (ii) $S \rightarrow aS$ gives rise to $S \rightarrow aS$ and $S \rightarrow a$.
- (iii) $S \rightarrow AB$ gives rise to $S \rightarrow AB$, $S \rightarrow A$ and $S \rightarrow B$.

(**Note:** We cannot erase both the nullable variables A and B in $S \rightarrow AB$ as we will get $S \rightarrow \Lambda$ in that case.)

Hence the required grammar without null productions is

$$G_1 = (\{S, A, B, D\}, \{a, b\}, P, S)$$

where P' consists of

$$D \rightarrow b, S \rightarrow aS, S \rightarrow AB, S \rightarrow a, S \rightarrow A, S \rightarrow B$$

ELIMINATION OF UNIT PRODUCTION

Let G be $S \rightarrow AB, A \rightarrow a, B \rightarrow C|b, C \rightarrow D, D \rightarrow E$ and $E \rightarrow a$. Eliminate unit productions and get an equivalent grammar.

Solution

Step 1 $V'_0(S) = \{S\}, \quad W_1(S) = W_0(S) \cup \emptyset$

Hence $W(S) = \{S\}$. Similarly,

$$W(A) = \{A\}, \quad W(E) = \{E\}$$

$$W_0(B) = \{B\}, \quad W_1(B) = \{B\} \cup \{C\} = \{B, C\}$$

$$W_2(B) = \{B, C\} \cup \{D\}, \quad W_3(B) = \{B, C, D\} \cup \{E\}, \quad W_4(B) = W_3(B)$$

Therefore,

$$W(B) = \{B, C, D, E\}$$

Similarly,

$$W_0(C) = \{C\}, \quad W_1(C) = \{C, D\}, \quad W_2(C) = \{C, D, E\} = W_3(C)$$

Therefore,

$$W(C) = \{C, D, E\}, \quad W_0(D) = \{D\}$$

Hence,

$$W_1(D) = \{D, E\} = W_2(D)$$

Thus,

$$W(D) = \{D, E\}$$

ELIMINATION OF UNIT PRODUCTION

Step 2 The productions in G_1 are

$$\begin{array}{lll} S \rightarrow AB, & A \rightarrow a, & E \rightarrow a \\ B \rightarrow b \mid a, & C \rightarrow a, & D \rightarrow a \end{array}$$

By construction, G_1 has no unit productions.

To complete the proof we have to show that $L(G') = L(G_1)$.

NORMAL FORMS FOR CFGs

- Chomsky Normal Form (CNF)
 - $A \rightarrow a$, $A \rightarrow BC$ and $S \rightarrow \epsilon$
- Greibach Normal Form (GNF)
 - $A \rightarrow a\alpha$ where α belongs to V^*

CHOMSKY NORMAL FORM

Reduce the following grammar G to CNF. G is $S \rightarrow aAD$, $A \rightarrow aB \mid bAB$, $B \rightarrow b$, $D \rightarrow d$.

Solution

As there are no null productions or unit productions, we can proceed to step 2.

Step 2 Let $G_1 = (V'_N, \{a, b, d\}, P_1, S)$, where P_1 and V'_N are constructed as follows:

- (i) $B \rightarrow b$, $D \rightarrow d$ are included in P_1 .
- (ii) $S \rightarrow aAD$ gives rise to $S \rightarrow C_aAD$ and $C_a \rightarrow a$.
 $A \rightarrow aB$ gives rise to $A \rightarrow C_aB$.
 $A \rightarrow bAB$ gives rise to $A \rightarrow C_bAB$ and $C_b \rightarrow b$.
 $V'_N = \{S, A, B, D, C_a, C_b\}$.

CHOMSKY NORMAL FORM

Step 3 P_1 consists of $S \rightarrow C_aAD$, $A \rightarrow C_aB \mid C_bAB$, $B \rightarrow b$, $D \rightarrow d$, $C_a \rightarrow a$, $C_b \rightarrow b$.

$A \rightarrow C_aB$, $B \rightarrow b$, $D \rightarrow d$, $C_a \rightarrow a$, $C_b \rightarrow b$ are added to P_2

$S \rightarrow C_aAD$ is replaced by $S \rightarrow C_aC_1$ and $C_1 \rightarrow AD$.

$A \rightarrow C_bAB$ is replaced by $A \rightarrow C_bC_2$ and $C_2 \rightarrow AB$.

Let

$$G_2 = (\{S, A, B, D, C_a, C_b, C_1, C_2\}, \{a, b, d\}, P_2, S)$$

where P_2 consists of $S \rightarrow C_aC_1$, $A \rightarrow C_aB \mid C_bC_2$, $C_1 \rightarrow AD$, $C_2 \rightarrow AB$, $B \rightarrow b$, $D \rightarrow d$, $C_a \rightarrow a$, $C_b \rightarrow b$. G_2 is in CNF and equivalent to G .

Example 01

Reduce the following grammar G to CNF. G is $S \rightarrow aAD$, $A \rightarrow aB \mid bAB$, $B \rightarrow b$, $D \rightarrow d$.

Solution

As there are no null productions or unit productions, we can proceed to step 2.

Step 2 Let $G_1 = (V'_N, \{a, b, d\}, P_1, S)$, where P_1 and V'_N are constructed as follows:

- (i) $B \rightarrow b$, $D \rightarrow d$ are included in P_1 .
- (ii) $S \rightarrow aAD$ gives rise to $S \rightarrow C_aAD$ and $C_a \rightarrow a$.
 $A \rightarrow aB$ gives rise to $A \rightarrow C_aB$.
 $A \rightarrow bAB$ gives rise to $A \rightarrow C_bAB$ and $C_b \rightarrow b$.
 $V'_N = \{S, A, B, D, C_a, C_b\}$.

Example 01

Step 3 P_1 consists of $S \rightarrow C_a AD$, $A \rightarrow C_a B \mid C_b AB$, $B \rightarrow b$, $D \rightarrow d$, $C_a \rightarrow a$, $C_b \rightarrow b$.

$A \rightarrow C_a B$, $B \rightarrow b$, $D \rightarrow d$, $C_a \rightarrow a$, $C_b \rightarrow b$ are added to P_2

$S \rightarrow C_a AD$ is replaced by $S \rightarrow C_a C_1$ and $C_1 \rightarrow AD$.

$A \rightarrow C_b AB$ is replaced by $A \rightarrow C_b C_2$ and $C_2 \rightarrow AB$.

Let

$$G_2 = (\{S, A, B, D, C_a, C_b, C_1, C_2\}, \{a, b, d\}, P_2, S)$$

where P_2 consists of $S \rightarrow C_a C_1$, $A \rightarrow C_a B \mid C_b C_2$, $C_1 \rightarrow AD$, $C_2 \rightarrow AB$, $B \rightarrow b$, $D \rightarrow d$, $C_a \rightarrow a$, $C_b \rightarrow b$. G_2 is in CNF and equivalent to G .

Example 02

Find a grammar in Chomsky normal form equivalent to $S \rightarrow aAbB$, $A \rightarrow aA \mid a$, $B \rightarrow bB \mid b$.

Solution

As there are no unit productions or null productions, we need not carry out step 1. We proceed to step 2.

Step 2 Let $G_1 = (V'_N \setminus \{a, b\}, P_1, S)$, where P_1 and V'_N are constructed as follows:

- (i) $A \rightarrow a$, $B \rightarrow b$ are added to P_1 .
- (ii) $S \rightarrow aAbB$, $A \rightarrow aA$, $B \rightarrow bB$ yield $S \rightarrow C_aAC_bB$, $A \rightarrow C_aA$, $B \rightarrow C_bB$, $C_a \rightarrow a$, $C_b \rightarrow b$.

$$V'_N = \{S, A, B, C_a, C_b\}.$$

Example 02

Step 3 P_1 consists of $S \rightarrow C_a A C_b B$, $A \rightarrow C_a A$, $B \rightarrow C_b B$, $C_a \rightarrow a$, $C_b \rightarrow b$, $A \rightarrow a$, $B \rightarrow b$.

$S \rightarrow C_a A C_b B$ is replaced by $S \rightarrow C_a C_1$, $C_1 \rightarrow A C_2$, $C_2 \rightarrow C_b B$

The remaining productions in P_1 are added to P_2 . Let

$$G_2 = (\{S, A, B, C_a, C_b, C_1, C_2\}, \{a, b\}, P_2, S),$$

where P_2 consists of $S \rightarrow C_a C_1$, $C_1 \rightarrow A C_2$, $C_2 \rightarrow C_b B$, $A \rightarrow C_a A$, $B \rightarrow C_b B$, $C_a \rightarrow a$, $C_b \rightarrow b$, $A \rightarrow a$, and $B \rightarrow b$.

G_2 is in CNF and equivalent to the given grammar.

GREIBACH NORMAL FORM

Lemma 6.1 Let $G = (V_N, \Sigma, P, S)$ be a CFG. Let $A \rightarrow B\gamma$ be an A -production in P . Let the B -productions be $B \rightarrow \beta_1 | \beta_2 | \dots | \beta_s$. Define

$$P_1 = (P - \{A \rightarrow B\gamma\}) \cup \{A \rightarrow \beta_i\gamma \mid 1 \leq i \leq s\}.$$

Then, $G_1 = (V_N, \Sigma, P_1, S)$ is a context-free grammar equivalent to G .

Lemma 6.2 Let $G = (V_N, \Sigma, P, S)$ be a context-free grammar. Let the set of A -productions be $A \rightarrow A\alpha_1 | \dots | A\alpha_r | \beta_1 | \dots | \beta_s$ (β_i 's do not start with A).

Let Z be a new variable. Let $G_1 = (V_N \cup \{Z\}, \Sigma, P_1, S)$, where P_1 is defined as follows:

(i) The set of A -productions in P_1 are $A \rightarrow \beta_1 | \beta_2 | \dots | \beta_s$

$$A \rightarrow \beta_1 Z | \beta_2 Z | \dots | \beta_s Z$$

(ii) The set of Z -productions in P_1 are $Z \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_r$

$$Z \rightarrow \alpha_1 Z | \alpha_2 Z \dots | \alpha_r Z$$

(iii) The productions for the other variables are as in P . Then G_1 is a CFG and equivalent to G .

Example 01

Apply Lemma 6.2 to the following A-productions in a context-free grammar G .

$$A \rightarrow aBD \mid bDB \mid c, \quad A \rightarrow AB \mid AD$$

Solution

In this example, $\alpha_1 = B$, $\alpha_2 = D$, $\beta_1 = aBD$, $\beta_2 = bDB$, $\beta_3 = c$. So the new productions are:

- (i) $A \rightarrow aBD \mid bDB \mid c, \quad A \rightarrow aBDZ \mid bDBZ \mid cZ$
- (ii) $Z \rightarrow B, Z \rightarrow D, \quad Z \rightarrow BZ \mid DZ$

Example 01

Apply Lemma 6.2 to the following A-productions in a context-free grammar G .

$$A \rightarrow aBD \mid bDB \mid c, \quad A \rightarrow AB \mid AD$$

Solution

In this example, $\alpha_1 = B$, $\alpha_2 = D$, $\beta_1 = aBD$, $\beta_2 = bDB$, $\beta_3 = c$. So the new productions are:

- (i) $A \rightarrow aBD \mid bDB \mid c, \quad A \rightarrow aBDZ \mid bDBZ \mid cZ$
- (ii) $Z \rightarrow B, Z \rightarrow D, \quad Z \rightarrow BZ \mid DZ$

Example 02

Construct a grammar in Greibach normal form equivalent to the grammar $S \rightarrow AA \mid a$, $A \rightarrow SS \mid b$.

Solution

The given grammar is in CNF. S and A are renamed as A_1 and A_2 , respectively. So the productions are $A_1 \rightarrow A_1A_2 \mid a$ and $A_2 \rightarrow A_1A_1 \mid b$. As the given grammar has no null productions and is in CNF we need not carry out step 1. So we proceed to step 2.

Step 2 (i) A_1 -productions are in the required form. They are $A_1 \rightarrow A_2A_2 \mid a$.

(ii) $A_2 \rightarrow b$ is in the required form. Apply Lemma 6.1 to $A_2 \rightarrow A_1A_1$. The resulting productions are $A_2 \rightarrow A_2A_2A_1$, $A_2 \rightarrow aA_1$. Thus the A_2 -productions are

$$A_2 \rightarrow A_2A_2A_1, \quad A_2 \rightarrow aA_1, \quad A_2 \rightarrow b$$

Example 02

$$A_2 \rightarrow A_2A_2A_1, \quad A_2 \rightarrow aA_1, \quad A_2 \rightarrow b$$

Step 3 We have to apply Lemma 6.2 to A_2 -productions as we have $A_2 \rightarrow A_2A_2A_1$. Let Z_2 be the new variable. The resulting productions are

$$\begin{aligned} A_2 &\rightarrow aA_1, & A_2 &\rightarrow b \\ A_2 &\rightarrow aA_1Z_2, & A_2 &\rightarrow bZ_2 \\ Z_2 &\rightarrow A_2A_1, & Z_2 &\rightarrow A_2A_1Z_2. \end{aligned}$$

Step 4 (i) The A_2 -productions are $A_2 \rightarrow aA_1 | b | aA_1Z_2 | bZ_2$.

(ii) Among the A_1 -productions we retain $A_1 \rightarrow a$ and eliminate $A_1 \rightarrow A_2A_2$ using Lemma 6.1. The resulting productions are $A_1 \rightarrow aA_1A_2 | bA_2$. $A_1 \rightarrow aA_1Z_2A_2 | bZ_2A_2$. The set of all (modified) A_1 -productions is

$$A_1 \rightarrow a | aA_1A_2 | bA_2 | aA_1Z_2A_2 | bZ_2A_2$$

Step 5 The Z_2 -productions to be modified are $Z_2 \rightarrow A_2A_1$, $Z_2 \rightarrow A_2A_1Z_2$. We apply Lemma 6.1 and get

$$\begin{aligned} Z_2 &\rightarrow aA_1A_1 | bA_1 | aA_1Z_2A_1 | bZ_2A_1 \\ Z_2 &\rightarrow aA_1A_1Z_2 | bA_1Z_2 | aA_1Z_2A_1Z_2 | bZ_2A_1Z_2 \end{aligned}$$

Example 02

Hence the equivalent grammar is

$$G' = (\{A_1, A_2, Z_2\}, \{a, b\}, P_1, A_1)$$

where P_1 consists of

$$A_1 \rightarrow a \mid aA_1A_2 \mid bA_2 \mid aA_1Z_2A_1 \mid bZ_2A_2$$

$$A_2 \rightarrow aA_1 \mid b \mid aA_1Z_2 \mid bZ_2$$

$$Z_2 \rightarrow aA_1A_1 \mid bA_1 \mid aA_1Z_2A_1 \mid bZ_2A_1$$

$$Z_2 \rightarrow aA_1A_1Z_2 \mid bA_1Z_2 \mid aA_1Z_2A_1Z_2 \mid bZ_2A_1Z_2$$

Example 03

Convert the grammar $S \rightarrow AB, A \rightarrow BS|b, B \rightarrow SA|a$ into GNF.

Solution

As the given grammar is in CNF, we can omit step 1 and proceed to step 2 after renaming S, A, B as A_1, A_2, A_3 , respectively. The productions are $A_1 \rightarrow A_2A_3, A_2 \rightarrow A_3A_1|b, A_3 \rightarrow A_1A_2|a$.

Step 2 (i) The A_1 -production $A_1 \rightarrow A_2A_3$ is in the required form.

(ii) The A_2 -productions $A_2 \rightarrow A_3A_1|b$ are in the required form.

(iii) $A_3 \rightarrow a$ is in the required form.

Apply Lemma 6.1 to $A_3 \rightarrow A_1A_2$. The resulting productions are $A_3 \rightarrow A_2A_3A_2$. Applying the lemma once again to $A_3 \rightarrow A_2A_3A_2$, we get

$$A_3 \rightarrow A_3A_1A_3A_2|bA_3A_2.$$

Example 03

Step 3 The A_3 -productions are $A_3 \rightarrow a \mid bA_3A_2$ and $A_3 \rightarrow A_3A_1A_3A_2$. As we have $A_3 \rightarrow A_3A_1A_3A_2$, we have to apply Lemma 6.2 to A_3 -productions. Let Z_3 be the new variable. The resulting productions are

$$\begin{aligned} A_3 &\rightarrow a \mid bA_3A_2, & A_3 &\rightarrow aZ_3 \mid bA_3A_2Z_3 \\ Z_3 &\rightarrow A_1A_3A_2, & Z_3 &\rightarrow A_1A_3A_2Z_3 \end{aligned}$$

Step 4 (i) The A_3 -productions are

$$A_3 \rightarrow a \mid bA_3A_2 \mid aZ_3 \mid bA_3A_2Z_3 \quad (6.9)$$

(ii) Among the A_2 -productions, we retain $A_2 \rightarrow b$ and eliminate $A_2 \rightarrow A_3A_1$ using Lemma 6.1. The resulting productions are

$$A_2 \rightarrow aA_1 \mid bA_3A_2A_1 \mid aZ_3A_1 \mid bA_3A_2Z_3A_1$$

The modified A_2 -productions are

$$A_2 \rightarrow b \mid aA_1 \mid bA_3A_2A_1 \mid aZ_3A_1 \mid bA_3A_2Z_3A_1 \quad (6.10)$$

(iii) We apply Lemma 6.1 to $A_1 \rightarrow A_2A_3$ to get

$$A_1 \rightarrow bA_3 \mid aA_1A_3 \mid bA_3A_2A_1A_3 \mid aZ_3A_1A_3 \mid bA_3A_2Z_3A_1A_3 \quad (6.11)$$

Example 03

Step 5 The Z_3 -productions to be modified are

$$Z_3 \rightarrow A_1A_3A_2 \mid A_1A_3A_2Z_3$$

We apply Lemma 6.1 and get

$$Z_3 \rightarrow bA_3A_3A_2 \mid bA_3A_2Z_3$$

$$Z_3 \rightarrow aA_1A_3A_3A_2 \mid aA_1A_3A_3A_2Z_3$$

$$Z_3 \rightarrow bA_3A_2A_1A_3A_3A_2 \mid bA_3A_2A_1A_3A_3A_2Z_3 \quad (6.12)$$

$$Z_3 \rightarrow aZ_3A_1A_3A_3A_2 \mid aZ_3A_1A_3A_3A_2Z_3$$

$$Z_3 \rightarrow bA_3A_2Z_3A_1A_3A_3A_2 \mid bA_3A_2Z_3A_1A_3A_3A_2Z_3$$



Turing Machines

Reading: Chapter 8



Turing Machines are...

- Very powerful (abstract) machines that could simulate any modern day computer (although very, very slowly!)
- Why design such a machine?
 - If a problem cannot be “solved” even using a TM, then it implies that the problem is *undecidable*
- Computability vs. Decidability

For every input,
answer YES or NO

A Turing Machine (TM)

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

This is like the CPU & program counter

Finite control

Tape is the memory

Tape head

Infinite tape with tape symbols



Input & output tape symbols

B: blank symbol (special symbol reserved to indicate data boundary)

You can also use:

□ for R

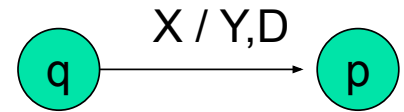
□ for L

Transition function

- One move (denoted by |---) in a TM does the following:

- $\delta(q, X) = (p, Y, D)$

- q is the current state
- X is the current tape symbol pointed by tape head
- State changes from q to p
- After the move:
 - X is replaced with symbol Y
 - If D="L", the tape head moves "left" by one position.
Alternatively, if D="R" the tape head moves "right" by one position.



ID of a TM

- Instantaneous Description or ID :

- $$X_1 X_2 \dots X_{i-1} q X_i X_{i+1} \dots X_n$$

means:

- q is the current state
- Tape head is pointing to X_i
- $X_1 X_2 \dots X_{i-1} X_i X_{i+1} \dots X_n$ are the current tape symbols

- $\delta(q, X_i) = (p, Y, R)$ is same as:

$$X_1 \dots X_{i-1} q X_i \dots X_n \quad | \text{---} \quad X_1 \dots X_{i-1} Y p X_{i+1} \dots X_n$$

- $\delta(q, X_i) = (p, Y, L)$ is same as:

$$X_1 \dots X_{i-1} q X_i \dots X_n \quad | \text{---} \quad X_1 \dots p X_{i-1} Y X_{i+1} \dots X_n$$

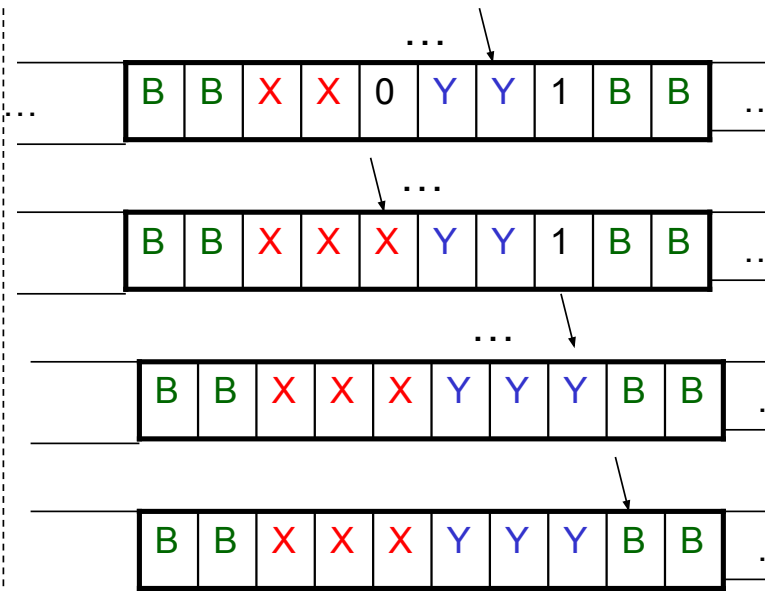
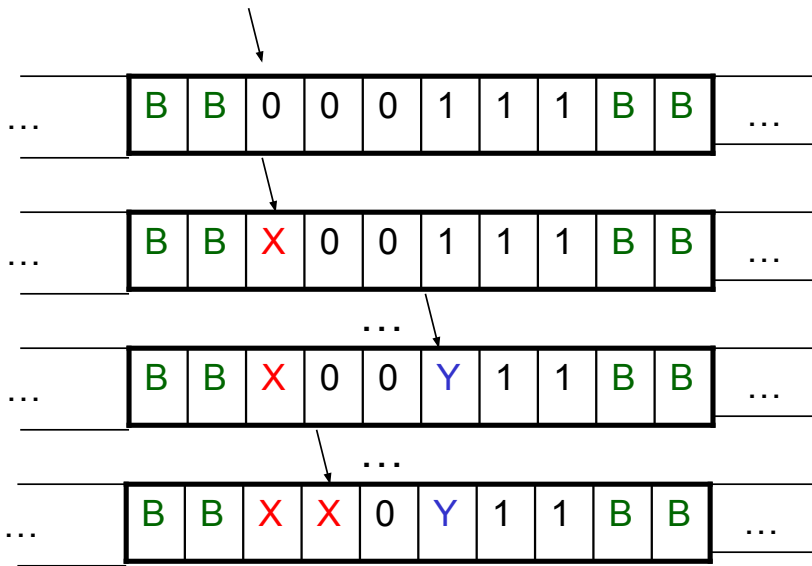


Way to check for Membership

- Is a string w accepted by a TM?
- Initial condition:
 - The (whole) input string w is present in TM, preceded and followed by infinite blank symbols
- Final acceptance:
 - Accept w if TM enters final state and halts
 - If TM halts and not final state, then reject

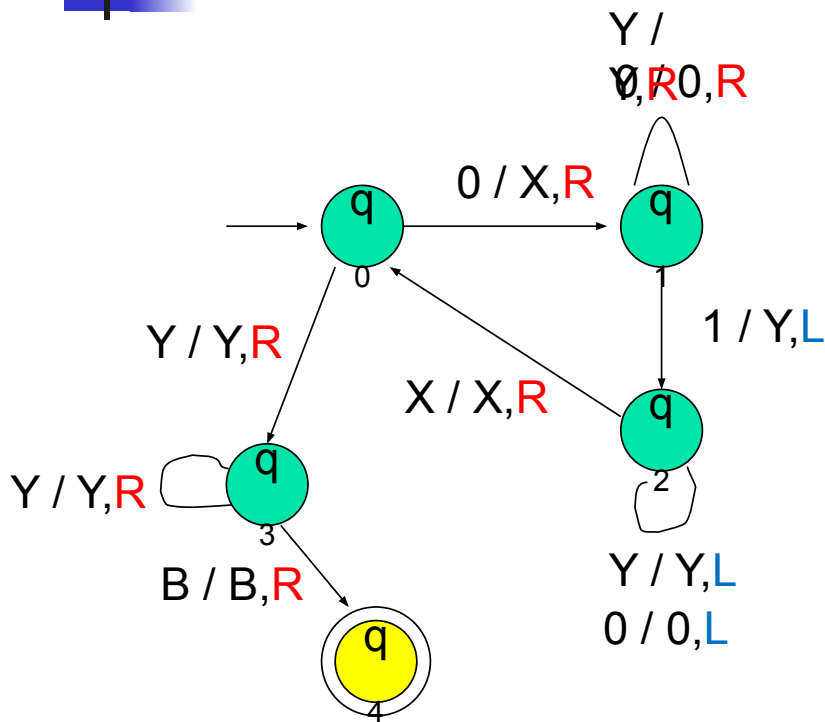
Example: $L = \{0^n 1^n \mid n \geq 1\}$

- Strategy: $w = 000111$



Accept

TM for $\{0^n 1^n \mid n \geq 1\}$



1. Mark next unread 0 with X and move right
2. Move to the right all the way to the first unread 1, and mark it with Y
3. Move back (to the left) all the way to the last marked X, and then move one position to the right
4. If the next position is 0, then goto step 1.
Else move all the way to the right to ensure there are no excess 1s. If not move right to the next blank symbol and stop & accept.

TM for $\{0^n 1^n \mid n \geq 1\}$

Curr. State	Next Tape Symbol				
	0	1	X	Y	B
→ q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
$*q_4$	-	--	-	-	-

Table representation of the state diagram



TMs for calculations

- TMs can also be used for calculating values
 - Like arithmetic computations
 - Eg., addition, subtraction, multiplication, etc.

Example 2: monus subtraction

$$“m \dot{-} n” = \max\{m-n, 0\}$$

$$0^m 1 0^n \square \quad \dots B 0^{m-n} B.. \text{ (if } m > n\text{)}$$
$$\dots BB\dots B.. \text{ (otherwise)}$$

1. For every 0 on the left (mark X), mark off a 0 on the right (mark Y)
2. Repeat process, until one of the following happens:
 1. // No more 0s remaining on the left of 1
Answer is 0, so flip all excess 0s on the right of 1 to Bs (and the 1 itself) and halt
 2. // No more 0s remaining on the right of 1
Answer is m-n, so simply halt after making 1 to B

Give state diagram



Example 3: Multiplication

- $0^m 1 0^n 1$ (input), $0^{mn} 1$ (output)
- Pseudocode:
 1. Move tape head back & forth such that for every 0 seen in 0^m , write n 0s to the right of the last delimiting 1
 2. Once written, that zero is changed to B to get marked as finished
 3. After completing on all m 0s, make the remaining n 0s and 1s also as Bs

Give state diagram

Calculations vs. Languages

A “calculation” is one that takes an input and outputs a value (or values)

The “language” for a certain calculation is the set of strings of the form “<input, output>”, where the output corresponds to a valid calculated value for the input

A “language” is a set of strings that meet certain criteria

E.g., The language L_{add} for the addition operation

“<0#0,0>”

“<0#1,1>”

...

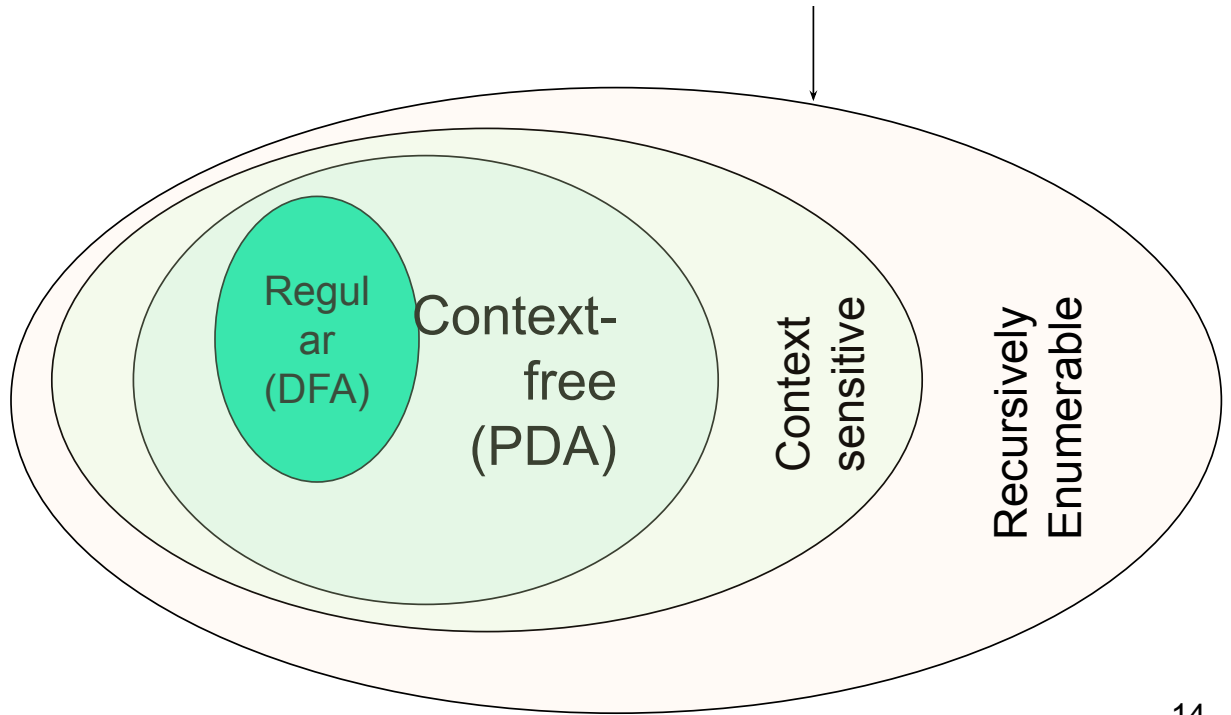
“<2#4,6>”

...

Membership question == verifying a solution
e.g., is “<15#12,27>” a member of L_{add} ?

Language of the Turing Machines

- *Recursive Enumerable (RE) language*





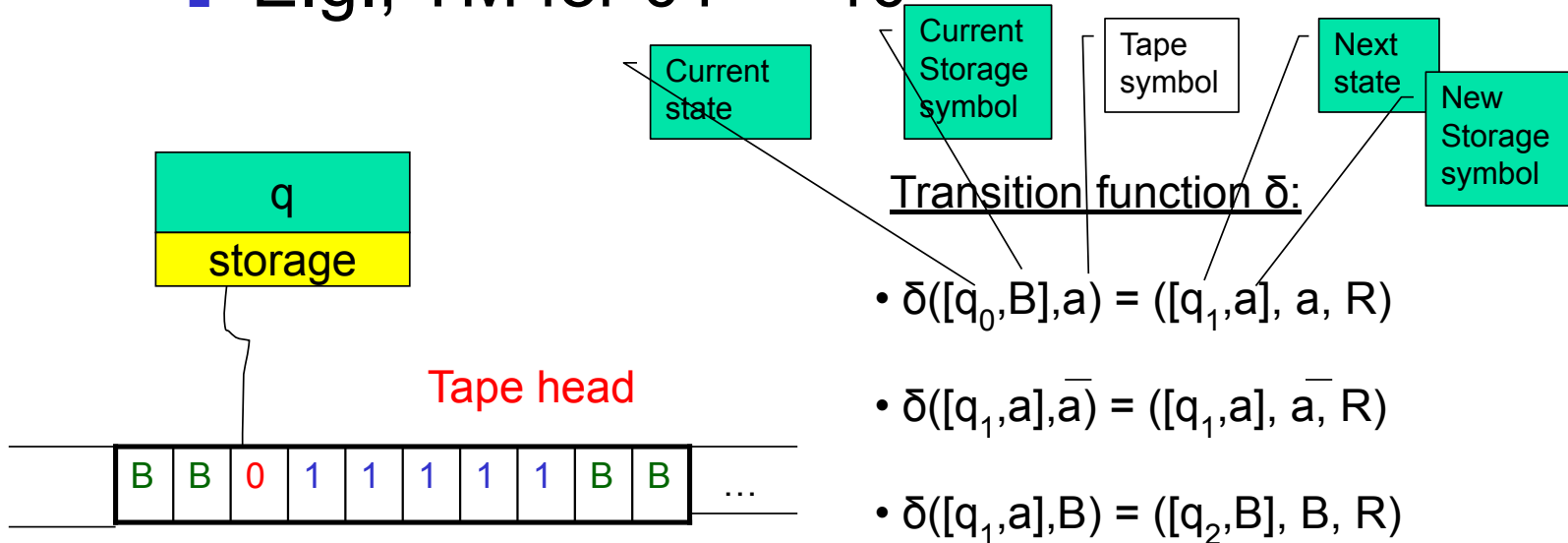
Variations of Turing Machines

TMs with *storage*

Generic description

Will work for both $a=0$ and $a=1$

- E.g., TM for $01^* + 10^*$



$[q, a]$: where q is current state,
 a is the symbol in storage

Are the standard TMs
equivalent to TMs with storage?

Yes



Standard TMs are equivalent to TMs with storage - Proof

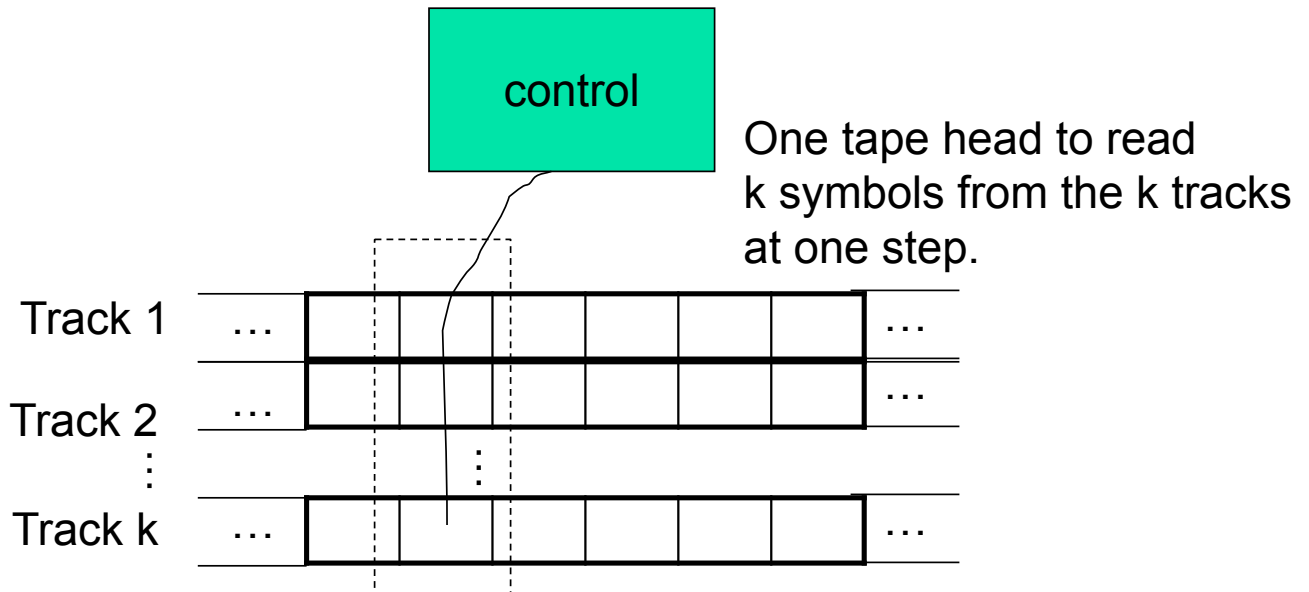
Claim: Every TM w/ storage can be simulated by a TM w/o storage as follows:

- For every [state, symbol] combination in the TM w/ storage:
 - Create a new state in the TM w/o storage
 - Define transitions induced by TM w/ storage

Since there are only finite number of states and symbols in the TM with storage, the number of states in the TM without storage will also be finite

Multi-track Turing Machines

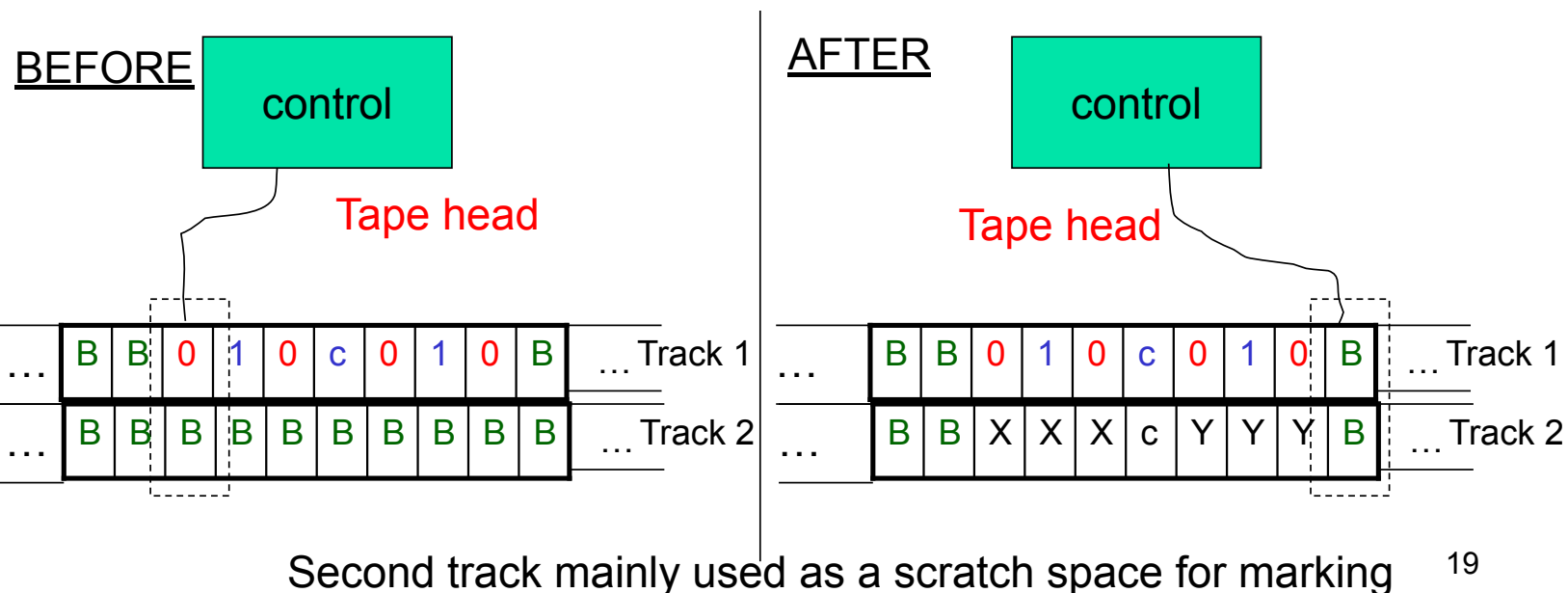
- TM with multiple tracks,
but just one unified tape head



Multi-Track TMs

- TM with multiple “tracks” but just one head

E.g., TM for $\{wcw \mid w \in \{0,1\}^*\}$
but w/o modifying original input string





Multi-track TMs are equivalent to basic (single-track) TMs

- Let M be a single-track TM
 - $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$
- Let M' be a multi-track TM (k tracks)
 - $M' = (Q', \Sigma', \Gamma', \delta', q'_0, B, F')$
 - $\delta'(q_i, \langle a_1, a_2, \dots, a_k \rangle) = (q_j, \langle b_1, b_2, \dots, b_k \rangle, L/R)$
- Claims:
 - *For every M , there is an M' s.t. $L(M) = L(M')$.*
 - (proof trivial here)

Multi-track TM \Rightarrow TM (proof)

- For every M' , there is an M s.t. $L(M')=L(M)$.

- $M = (Q, \Sigma, \Gamma, \delta, q_0, [B, B, \dots], F)$
- Where:
 - $Q = Q'$
 - $\Sigma = \Sigma' \times \Sigma' \times \dots$ (k times for k-track)
 - $\Gamma = \Gamma' \times \Gamma' \times \dots$ (k times for k-track)
 - $q_0 = q'_0$
 - $F = F'$
 - $\delta(q_i, [a_1, a_2, \dots, a_k]) = \delta'(q_i, \langle a_1, a_2, \dots, a_k \rangle)$

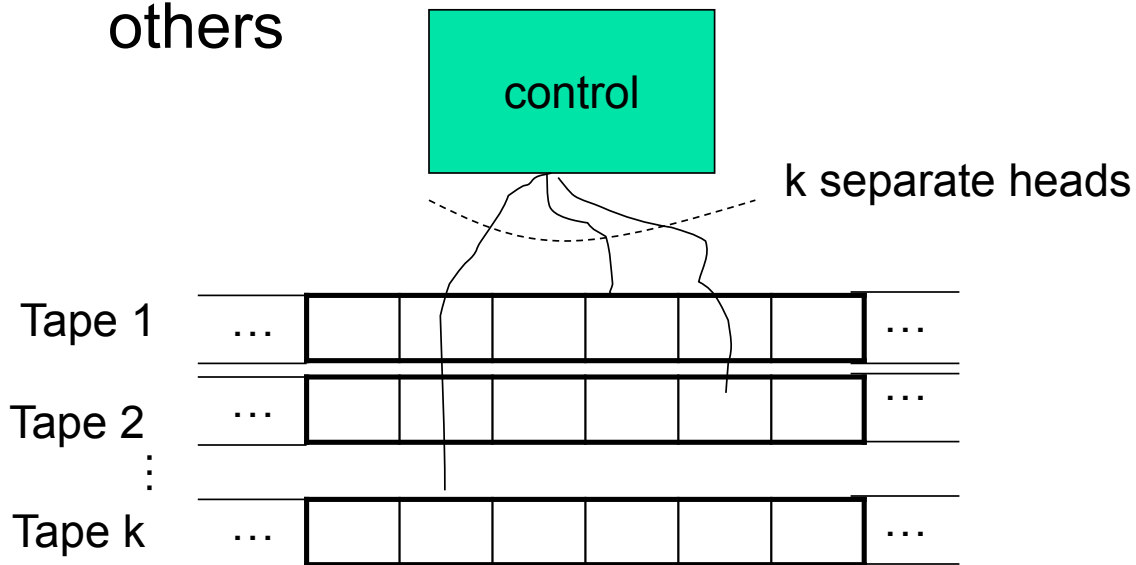
Main idea:

Create one composite symbol to represent every combination of k symbols

- Multi-track TMs are just a different way to represent single-track TMs, and is a matter of design convenience.

Multi-tape Turing Machines

- TM with multiple tapes, *each tape with a separate head*
 - Each head can move independently of the others





On how a Multi-tape TM would operate

- Initially:
 - The input is in tape #1 surrounded by blanks
 - All other tapes contain only blanks
 - The tape head for tape #1 points to the 1st symbol of the input
 - The heads for all other tapes point at an arbitrary cell (doesn't matter because they are all blanks anyway)
- A move:
 - Is a function (current state, the symbols pointed by all the heads)
 - After each move, each tape head can move independently (left or right) of one another

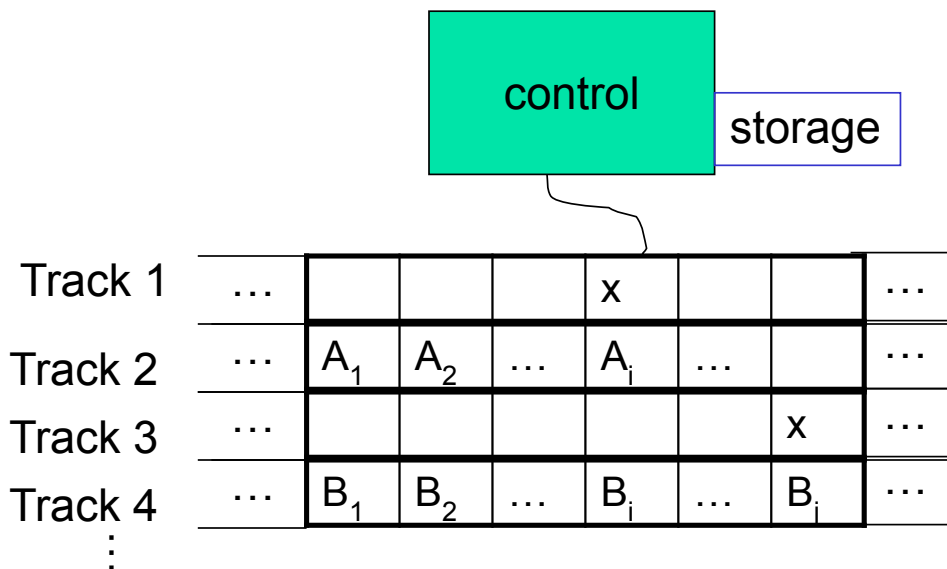


Multitape TMs \equiv Basic TMs

- Theorem: Every language accepted by a k-tape TM is also accepted by a single-tape TM
- Proof by construction:
 - Construct a single-tape TM with $2k$ tracks, where each tape of the k-tape TM is simulated by 2 tracks of basic TM
 - k out of the $2k$ tracks simulate the k input tapes
 - The other k out of the $2k$ tracks keep track of the k tape head positions

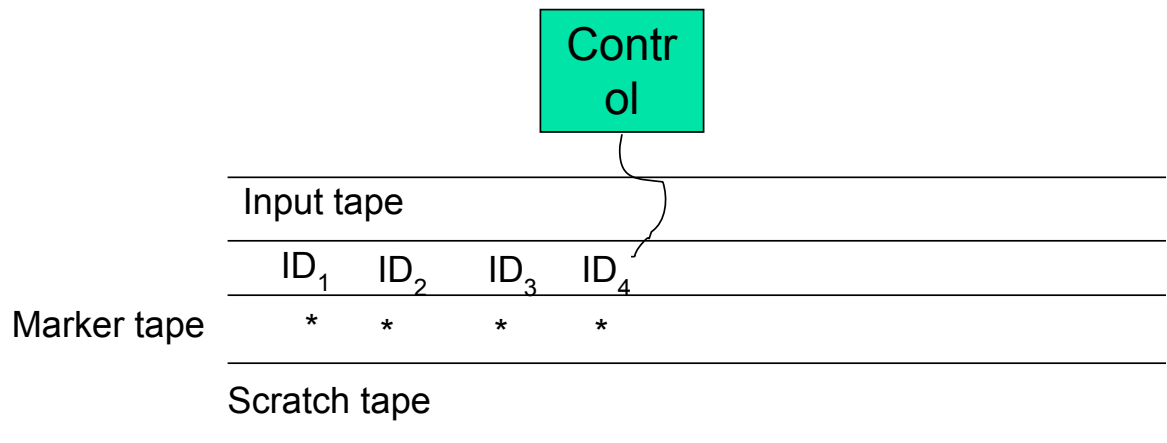
Multitape TMs \equiv Basic TMs ...

- To simulate one move of the k-tape TM:
 - Move from the leftmost marker to the rightmost marker (k markers) and in the process, gather all the input symbols into storage
 - Then, take the action same as done by the k-tape TM (rewrite tape symbols & move L/R using the markers)



Non-deterministic TMs

- A TM can have non-deterministic moves:
 - $\delta(q, X) = \{ (q_1, Y_1, D_1), (q_2, Y_2, D_2), \dots \}$
- Simulation using a multitape deterministic TM:





Summary

- TMs == Recursively Enumerable languages
- TMs can be used as both:
 - Language recognizers
 - Calculators/computers
- ***Basic TM is equivalent to all the below:***
 1. *TM + storage*
 2. *Multi-track TM*
 3. *Multi-tape TM*
 4. *Non-deterministic TM*
- TMs are like universal computing machines with unbounded storage

WHY TURING MACHINES?

- Turing Machine is the most general model.
- It accepts type-0 languages.
- It can also used to compute functions.
- It turns out to be mathematical model of partial recursive functions.
- It is used to determine the undecidablity of certain languages and measuring the space and time complexity of problems.

TURING MACHINE MODEL

- Turing machine is supposed to be having the following components.
 - Infinite length tape
 - Read/Write head
 - Finite control
- Input tape is divided into number of cells to which the head can read as well as write.
- R/W head is able to move in both the directions.
- The finite control consists of internal states.

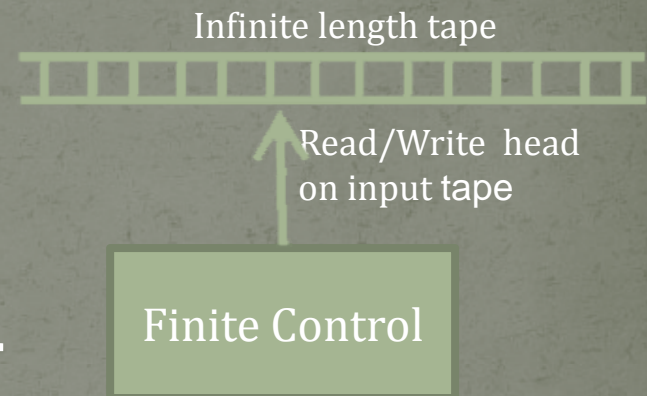


Fig.-Turing Machine

FORMAL DEFINITION OF TM

- A Turing Machine M is a 7-tuple, namely $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$ where
 - Q is a finite non empty set of states,
 - Γ is a finite non empty set of tape symbols,
 - b belongs to Γ is a blank symbol,
 - δ is the transition function mapping (q, x) onto (q', y, D) where D denotes the direction of movement of R/W head; $D=L$ or R according as the movement is to the left or right.
 - q_0 belongs to Q is the initial state, and
 - F a subset of Q is the set of final states.

WORKING PRINCIPAL OF TM

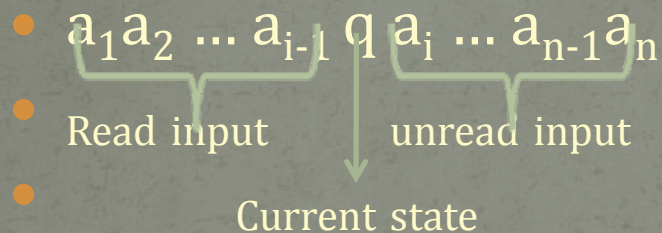
- Each cell on the input tape is capable of one symbol.
- At any instance of time the FC is in a particular state and reads the symbol just above the R/W head .
- After reading the input symbol the TM does one of the following according to the transition defined for a given TM.
 - A new symbol is written on to the input tape in the cell pointed by the R/W head.
 - R/W head moves in the direction R (right) or L (left).
 - The FC moves to the next state.
 - Whether to halt or not.

REPRESENTATION OF TM

- Turing machine can be represented in one of the following notation:
 - State diagram
 - Transition table
 - Instantaneous description.

REPRESENTATION OF TM (CONDT...)

- ID notation is used to represent the state of a TM at a particular instant of time as follows:



- transition between IDs takes place with help of move relation (\vdash) as follows

- $a_1 a_2 \dots a_{i-1} q a_i \dots a_{n-1} a_n \vdash a_1 a_2 \dots a_{i-2} b_i q' a_{i+1} \dots a_{n-1} a_n$ (right move)
- $a_1 a_2 \dots a_{i-1} q a_i \dots a_{n-1} a_n \vdash a_1 a_2 \dots a_{i-2} q' a_{i-1} b_i a_{i+1} \dots a_{n-1} a_n$ (left move)

CONSTRUCTION OF TM FOR 0^n1^n

- The language is context free.
- It has # of 0's equal to the # of 1's and all the 1's follows the 0's.
- The logic behind the construction is that,
 - For each 0 read the head moves to the right of the input string and finds a matching 1.
 - It keeps tracks of the 0's read by marking them as 'x' and that of matching 1's by 'y'.
 - It continues the above process till,
 - All 0's and 1's are marked (accept)
 - Some 0's are unmarked i.e. # of 0's > # of 1's (reject)
 - Some 1's are unmarked i.e. # of 0's < # of 1's (reject)

CONSTRUCTION OF TM FOT 0^n1^n (Cont...)

- Let us take $w=0011$, then we can write the sequence of moves as follows:
 - $0011 \rightarrow x011 \rightarrow x0y1 \rightarrow xxy1 \rightarrow xxyy$ (accept)
- Let us take $w=00011$, then we can write the sequence of moves as follows:
 - $00011 \rightarrow x0011 \rightarrow x00y1 \rightarrow xx0y1 \rightarrow xx0yy$ (reject)
- Let us take $w=00111$, then we can write the sequence of moves as follows:
 - $00111 \rightarrow x0111 \rightarrow x0y11 \rightarrow xxy11 \rightarrow xxyy1$ (reject)

CONSTRUCTION OF TM FOT 0^n1^n (Cont...)

- Let us implement this logic in the construction as follows:
 - Initially the FC/CU is in state q_0 and on reading a 0 it changes to q_1 and marks(replace) the 0 to 'x' and moves right.
 - Being in state q_1 it moves right and on encountering an 1 the FC go to q_2 and marks(replace) the 1 to 'y' and moves left.
 - Being in state q_2 it moves left till a 'x' is read and the FC go to q_0 and continues the last two steps till

CONSTRUCTION OF TM FOT 0^n1^n (Cont...)

- All the 0's are marked i.e. q_0 reads a 'y' after immediately reading a 'x'. Then we need check if any 1 is left unmarked. This can be done by changing to q_3 and moving right till a 'b' is encountered. On reading a 'b' the CU changes to q_4 which is an accepting state.
- When the # of 0's > that of 1, after marking the first excess 0 the CU will not find a matching 1 i.e. q_1 on moving extreme right will encounter a 'b' instead of 1. Hence the TM will come to a halt(reject).
- When the # of 0's < that of 1, after marking the 1 corresponding to the last 0 CU will have excess 1's i.e. q_3 on moving right will encounter a excess 1 instead of 'b'. Hence the TM will come to a halt(reject).

CONSTRUCTION OF TM FOT 0^n1^n (Cont...)

- Let us use this TM for the three input strings discussed earlier.
- When $w=0011$, then we can write the sequence of moves with ID's as follows:
 - $q_00011b \rightarrow xq_1011b \rightarrow x0q_111b \rightarrow xq_20y1b \rightarrow q_2x0y1b \rightarrow xq_00y1b \rightarrow xxq_1y1b \rightarrow xxyq_11b \rightarrow xxq_2yyb \rightarrow xq_2xyyb \rightarrow xxq_0yyb \rightarrow xxyq_3yb \rightarrow xxyyq_3b \rightarrow xxyyqb_4b$ (accept)

CONSTRUCTION OF TM FOT 0^n1^n (Cont...)

- When $w=00011$, then we can write the sequence of moves with ID's as follows:
 - $q_000011b \rightarrow xq_10011b \rightarrow x0q_1011b \rightarrow x00q_111b \rightarrow x0q_20y1b \rightarrow xq_200y1b \rightarrow q_2x00y1b \rightarrow xq_000y1b \rightarrow xxq_10y1b \rightarrow xx0q_1y1b \rightarrow xx0yq_11b \rightarrow xx0q_2yyb \rightarrow xq_2x0yyb \rightarrow xxq_00yyb \rightarrow xxxq_1yyb \rightarrow xxxyq_1yb \rightarrow xxxyyq_1b$ (reject)

CONSTRUCTION OF TM FOT 0^n1^n (Cont...)

- When $w=00111$, then we can write the sequence of moves with ID's as follows:
 - $q_000111b \rightarrow xq_10111b \rightarrow x0q_1111b \rightarrow xq_20y11b \rightarrow q_2x0y11b \rightarrow xq_00y11b \rightarrow xxq_1y11b \rightarrow xxyq_111b \rightarrow xxq_2yy1b \rightarrow xq_2xyy1b \rightarrow xxq_0yy1b \rightarrow xxyq_3y1b \rightarrow xxyyq_31b$ (reject)

CONSTRUCTION OF TM FOT 0^n1^n (Cont...)

Present State	Tape Symbols				
	0	1	x	y	b
$\rightarrow q_0$	xq_1R			yq_3R	bq_4R
q_1	$0q_1R$	yq_2L		yq_1R	
q_2	$0q_2L$		xq_0R	yq_2L	
q_3				yq_3R	bq_4R
(q_4)					

CONSTRUCTION OF TM FOT $0^n1^n2^n$

- The language is context sensitive.
- It has equal # of 0's, 1's and 2's and all the 1's follows the 0's and all the 2's follows the 1's.
- The logic behind the construction is that,
 - For each 0 read the head moves to the right of the input string and finds a matching 1 and again moves right to find a matching 2.
 - It keeps tracks of the 0's, 1's and 2's by marking them as 'x', 'y' and 'z' respectively.
 - It continues the above process till,
 - All 0's , 1's and 2's are marked (accept)
 - Unequal # of 0's, 1's and 2's marked(reject)

CONSTRUCTION OF TM FOT $0^n1^n2^n$

- Let $w=001122$ and find the moves by applying the same logic.
 - $q_0001122b \rightarrow xq_101122b \rightarrow x0q_11122b \rightarrow x0yq_2122b \rightarrow x0y1q_222b \rightarrow x0yq_31z2b \rightarrow x0q_3y1z2b \rightarrow xq_30y1z2b \rightarrow q_3x0y1z2b \rightarrow xq_00y1z2b \rightarrow xxq_1y1z2b \rightarrow xxyq_11z2b \rightarrow xxyyq_2z2b \rightarrow xxyyzq_22b \rightarrow xxyyq_3zzb \rightarrow xxyq_3yzzb \rightarrow xxq_3yyzzb \rightarrow xq_3xyyzzb \rightarrow xxq_0yyzzb \rightarrow xxyq_4yzzb \rightarrow xxyyq_4zzb \rightarrow xxyyzq_4zb \rightarrow xxyyzzq_4b \rightarrow xxyyzzbq_5b$
 - q_5 is the accepting state hence w is accepted.

CONSTRUCTION OF TM FOT $0^n1^n2^n$

Present State	Tape Symbols						
	0	1	2	x	y	z	b
$\rightarrow q_0$	xq_1R				yq_4R		bq_5R
q_1	$0q_1R$	yq_2R			yq_1R		
q_2		$1q_2R$	zq_3L		yq_2L		
q_3	zq_3L	zq_3L		xq_0R	yq_3R		
q_4					yq_4R	yq_4R	bq_5R
(q_5)							

TECHNIQUES FOR TM CONSTRUCTION OR VARIANTS OF TM

- To make the construction easier we make use of high-level description called as *Implementation description* or *high-level description*.
- The *implementation description* are of the following type:
 - Non Deterministic TM
 - TM with stationary head (stay option)
 - Storage in state
 - Subroutines
 - Multiple track TM
 - Multitape TM
 - Multidimensional TM
 - TM with Semi-Infinite Tape
 - Off-Line TM
 - Universal TM

NONDETERMINISTIC TM

- We can make the TM to behave nondeterministic ally by making the FC to have more than one configuration on reading a single tape symbol.
- This is possible by expanding the definition of δ as follows: $Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L, R\}}$ (it represents the power set of $Q \times \Gamma \times \{L, R\}$), where as the standard TM had δ as follows: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$.

TM WITH STATIONARY HEAD

- In some cases we may come across a situation where may not want the head to move in either direction.
- This is possible by expanding the definition of δ as follows: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$ (where S means, the head is stationary), where as the standard TM had δ as follows: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$.
- $uqaxv \vdash uq'bxv$ (a move corresponding to the S)
- $uqaxv \vdash ubq''xv \vdash uq'bxv$ (a move corresponding first L and then R move in a standard TM)

STORAGE IN STATE

- We can always use a state to store a input symbol.
- Then TM can be defined as follows:
 - $M=(Q \times \Gamma, \Sigma, \Gamma, \delta, q_0, b, F)$ the new set of states is $Q \times \Gamma$ and containing elements of the form $[q_0, a]$.
 - $M=(Q, \Sigma, \Gamma, \delta, q_0, b, F)$ being the description of standard TM.
 - Using this concept we can construct a TM for the RE 10^*+01^* .

STORAGE IN STATE: An Example

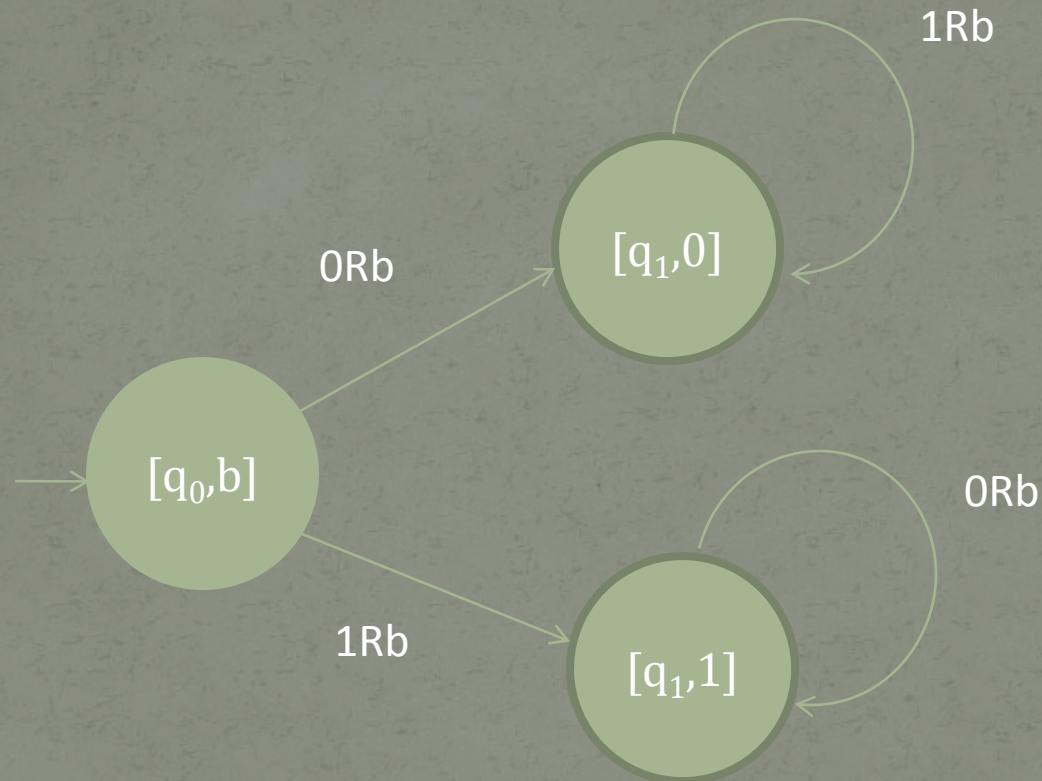


Fig. TM representing the RE 01^*+10^*

SUBROUTINES

- In the construction of TM we have some steps which are repeated a # of times.
- Hence these can be treated as the subroutines which can be designed as a separate TM and can be called when ever required.
- When ever a subroutine TM is called then the main TM comes to a temporary halt condition, and the later TM resumes it process when the subroutine TM finishes it's work.
- A subroutine TM is always associated with an initial state and a return state.

SUBROUTINES: An Example

- Design a TM which can multiply two +ve integer.
 - The input (m, n) being given, the +ve integer are represented as $0^m 1 0^n 1$.
 - Input: $b 0^m 1 0^n 1 b$, Output: $b 0^m 0^n b$.
 - The steps involved in the process are as follows:
 1. $0^m 1 0^n 1$ is placed on the tape.
 2. The leftmost 0 is erased (by replacing with 'b').
 3. A block of n 0's is copied onto the right end (by COPY subroutine).
 4. Step 2. and 3. are repeated m times and $1 0^n 1 0^{mn}$ is obtained on the tape.
 5. The prefix $1 0^m 1$ of $1 0^n 1 0^{mn}$ is erased, leaving the product mn as the output.

SUBROUTINES: An Example

TM for COPY subroutine

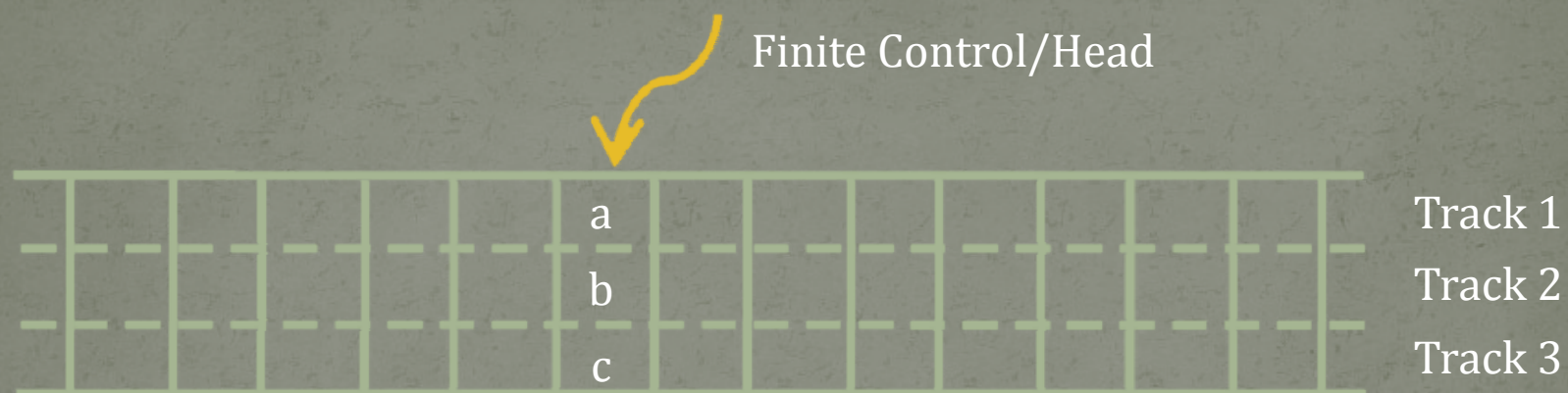
State	Tape Symbols			
	0	1	2	b
$\rightarrow q_1$	$q_2 2R$	$q_4 1L$		
q_2	$q_2 0R$	$q_2 1R$		$q_3 0L$
q_3	$q_3 2L$	$q_3 1L$	$q_1 2R$	
q_4		$q_5 1R$	$q_4 0L$	
(q_5)				

SUBROUTINES: An Example

TM for Multiplication of two numbers

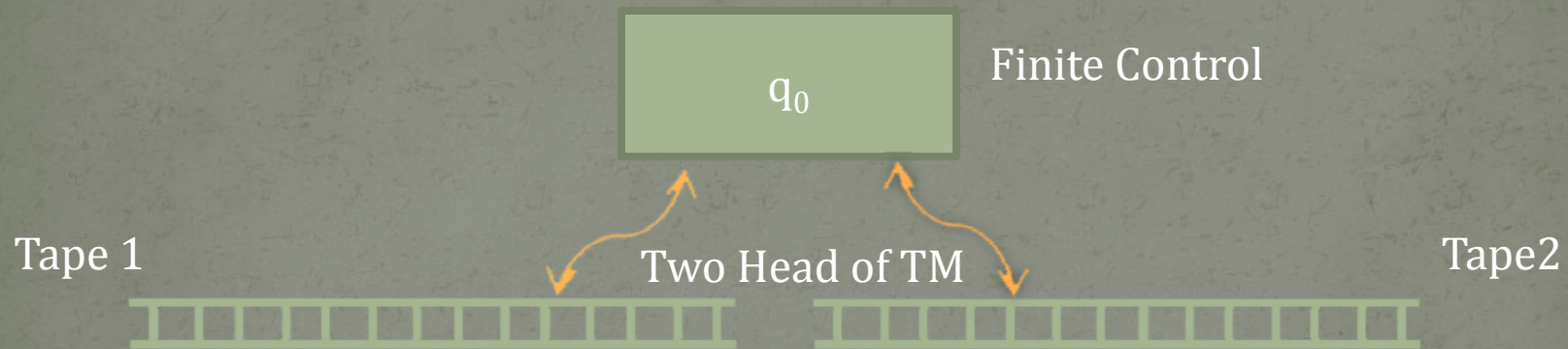
State	Tape Symbols			
	0	1	2	b
$\rightarrow q_0$	q_6bR			
q_6	q_60R	q_11R		
q_5	q_70L			
q_7		q_81L		
q_8	q_90L			$q_{10}bR$
q_9	q_90L			q_0bR
q_{10}		$q_{11}bR$		
(q_{11})	$q_{11}bR$	$q_{12}bR$		

MULTIPLE TRACK TM



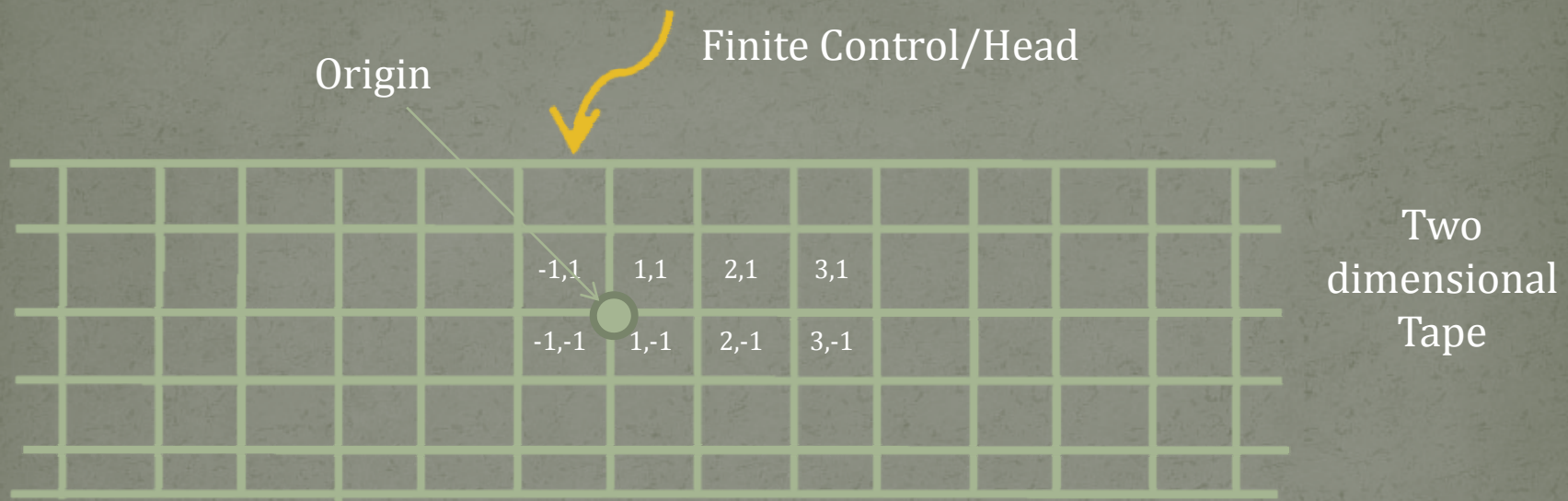
- Here a single tape is divided into k tracks and Γ is consisting of elements composed of k tape symbols (each element is a k -tuple), where ' k ' is a constant associated with a TM.
- Here the TM is defined as $M = (Q, \Sigma, \Gamma^k, \delta, q_0, b, F)$.
- Here $\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}$
- Fig, has $\Gamma = \{(a, b, c), (a, b, e), (d, c, f), \dots\}$.

MULTITAPE TM



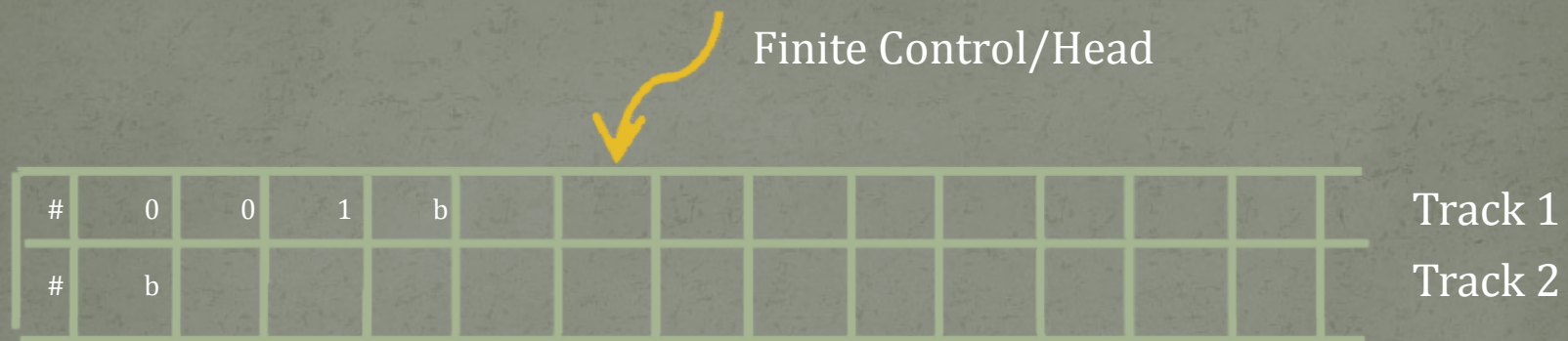
- Here we have more than one tape all having a single tape symbol as Γ .
- Here the TM is defined as $M=(Q, \Sigma, \Gamma, \delta, q_0, b, F)$.
- Here $\delta: Q \times \Gamma^n \rightarrow Q \times \Gamma^n \times \{L,R\}^n$ ('n' = # of tapes).
- n # of heads reads n different input symbols.
- Although it has n heads corresponding to n tapes but has a single finite control head.

MULTIDIMENSIONAL TM



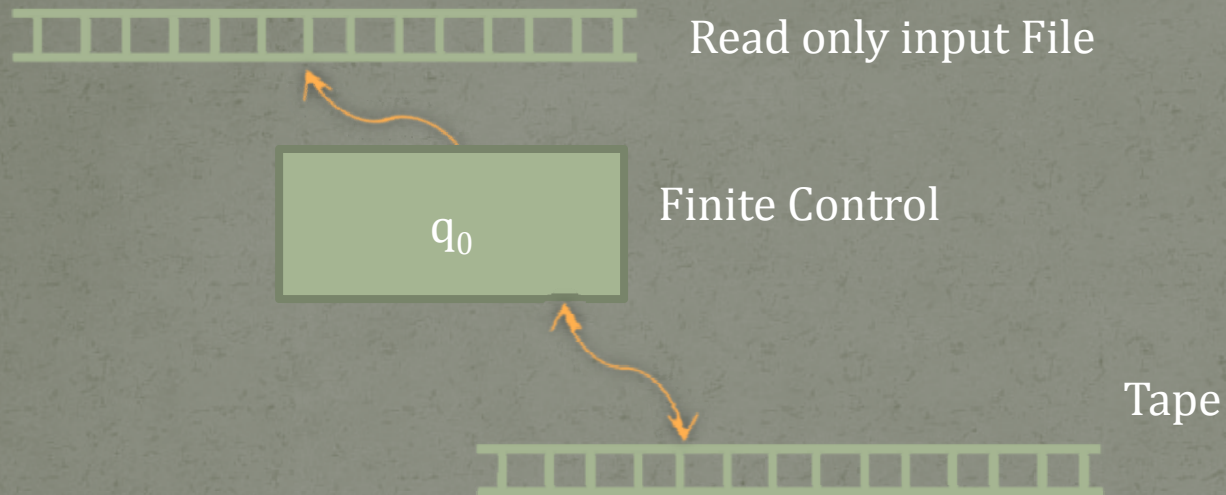
- Here $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R,U,D\}$ (U, D corresponds to the movement of the head in up and down direction).
- The reference point is taken as origin of the multidimensional tape and from this position the head can move in L, R, U, D directions.

TM WITH SEMI-INFINITE TAPE



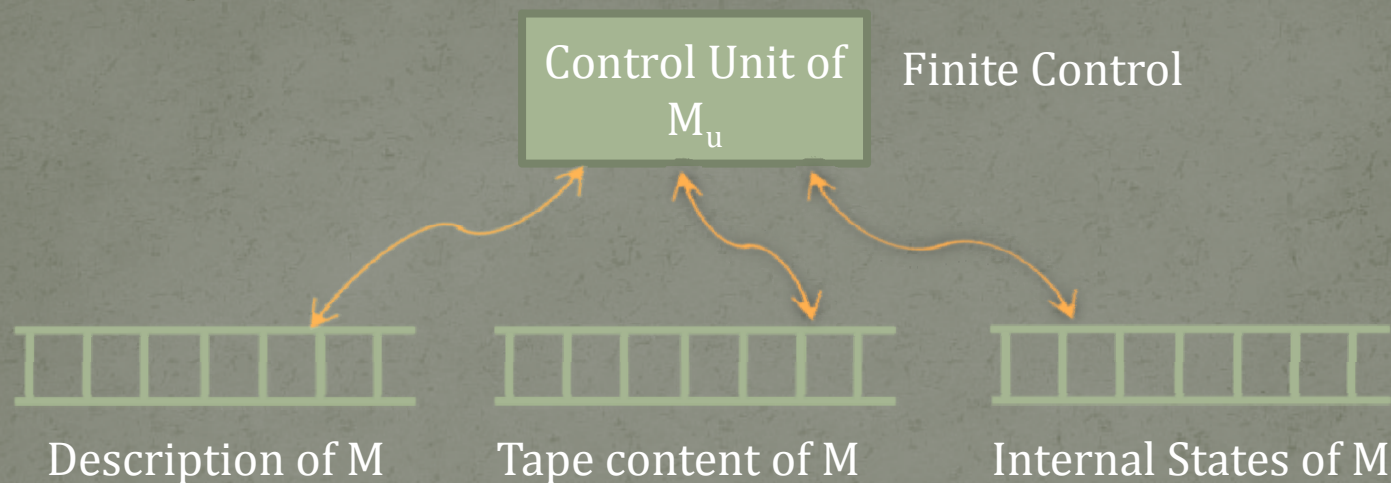
- Here the TM has only one tape which is bounded on the left and is also divided into two tracks.
- The upper track contains the input string after the '#' symbol which is considered as the reference point.
- Generally the TM works on the upper track but when it has to move to the left of # symbol, then the lower track is the working tape for the TM.

OFFLINE TM



- It has an additional read only input tape to that of standard TM.
- The work of these kind of TM is to write the contents of the input tape onto the standard tape.
- Hence these are also called as offline printers.

UNIVERSAL TM



- It is called as universal TM as it can simulate any given language (simulates a general purpose computer).
- Here the description, tape contents and the internal state of any TM is encoded in terms of 0's and 1's and are placed on three different tape.
- The control unit in this case behave just like the control unit of encoded TM.

DEFINITION OF ALGORITHM

- ✗ *Algorithm* is a collection of simple instruction for carrying out some task.
- ✗ Until 20th century the mathematicians used the intuitive notation of algorithm, which was not sufficient in gaining a deeper understanding of algorithm.
- ✗ In 1900 David Hilbert identified 23 mathematical problems and posed them as challenge for the coming century.
- ✗ The 10th problem in list concerned algorithm (devise an algorithm that test whether a polynomial has an integral roots).

DEFINITION OF ALGORITHM(CONT...)

- ✖ We know that no algorithm exist for this task as it is algorithmically unsolvable.
- ✖ We can conclude this only if we have clear definition of algorithm.
- ✖ This definition came in the 1936 paper of Alonzo Church (used λ -calculus) and Alan Turing (used Turing Machines) to define algorithm.
- ✖ These two definition were shown to be equivalent . This is known as “*Church-Turing Thesis*”.

★ *(Intuitive notation of algorithms = Turing machine algorithm)*

DEFINITION OF ALGORITHM(CONT...)

- ✖ We use TM for defining our algorithms.
- ✖ We have three standard ways to describe TM algorithm which are as follows:
 - + *Formal definition:* Low level description of TM in terms of states, transition function and so on.
 - + *Implementation description:* We make use of English prose to describe the movement of head and storage of data on the Tape.
 - + *High-level description:* We use English prose to describe and algorithm, ignoring the implantation details.
- ✖ Now onwards we will be using the high-level description for a TM.
- ✖ The TM usually take input strings 'w' as input, but we can also pass objects like polynomials, graphs, grammar etc.
- ✖ Let O be an object then $\langle O \rangle$ is encoded input to the TM. If O_1, O_2, \dots, O_n are the objects the $\langle O_1, O_2, \dots, O_n \rangle$ is the encoded input to the TM.

DEFINITION OF ALGORITHM(CONT...)

✖ Example:

- + Write the high-level description of a TM that decides A . Where A is a language consisting of all representing the undirected graphs that are connected.

✖ Solution:

- + The set A is represented as follows $A = \{ \langle G \rangle \mid G \text{ is a connected undirected graph} \}$.
- + Then the high-level description of the TM that decides A is as follows:
- + $M =$ "On input $\langle G \rangle$, the encoding of a graph G .
- + 1. Select the first node of G and mark it.
- + 2. Repeat the following stages until no new nodes are marked.
- + 3. For each node in G , mark it if it is attached by an edge to a node that is already marked.
- + 4. Scan all the nodes of G to determine whether they all are marked. If they are, *accept*; otherwise *reject*."

DEFINITION OF ALGORITHM(CONT...)

- ✗ Under decidability we will try to investigate the power of algorithms to solve problems.
- ✗ Intern we will be exploring the limit of algorithmically solvability (finding unsolvable problems).
- ✗ The unsolvability of some problems are subject of concern for two reasons:
 - + Problem can be altered or simplified before they turn unsolvable.
 - + Stimulates our imagination and helps us to gain an important perspective on computation

DECIDABILITY

- ✗ The term is associated with a TM representing a set or language.
- ✗ The TM is said to a *decider* for a language as it decides whether a particular element or string belongs to the language or not.
- ✗ A problem is said to be solvable if we have a TM acting as the *decider* for the given problem else it is unsolvable.
- ✗ Implies if a problem is solvable then there exist a TM acting as the *decider* for the problem. And the problem is decidable else undecidable.
- ✗ Hence decidability divides our problem set into:
 - + Decidable and
 - + Undecidable.

DECIDABLE PROBLEMS CONCERNING L_{RL}

- ✗ Computational problems concerning FA are as follows:
 - + FA accepting a string 'w' (A_{DFA}),
 - + NFA accepting a string 'w' (A_{NFA}),
 - + RE generating a string 'w' (A_{REX}),
 - + FA is empty (E_{DFA}),
 - + Two FAs are equivalent (EQ_{DFA}).
- ✗ All the above problems are decidable as we have a TM acting as a *decider* for each problem.

A_{DFA} IS DECIDABLE

- ✗ $A_{\text{DFA}} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \}.$
- ✗
- ✗ $M =$ “On input $\langle B, w \rangle$, where B is a DFA and w is a string:
 - + 1. Simulate B on input w .
 - + 2. If the simulation ends in an accept state, *accept*. If it ends in a non accepting state, *reject*.”

A_{NFA} IS DECIDABLE

- ✗ $A_{\text{NFA}} = \{ \langle B, w \rangle \mid B \text{ is an NFA that accepts input string } w \}.$
- ✗
- ✗ $N =$ “On input $\langle B, w \rangle$, where B is a NFA and w is a string:
 - + 1. Convert NFA B to an equivalent DFA C .
 - + 2. Run TM M on input $\langle C, w \rangle$.
 - + 3. If M accepts, *accept*; otherwise rejects, *reject*.”

A_{REX} IS DECIDABLE

- ✗ $A_{\text{REX}} = \{ \langle R, w \rangle \mid R \text{ is regular expression that generates input string } w \}.$
- ✗
- ✗ $P =$ “On input $\langle R, w \rangle$, where R is RE and w is a string:
 - + 1. Convert RE R to an equivalent DFA A .
 - + 2. Run TM M on input $\langle A, w \rangle$.
 - + 3. If M accepts, *accept*; otherwise rejects *reject*.”

E_{DFA} IS DECIDABLE

- ✗ T=“On input $\langle A \rangle$, where A is a DFA:
 - + 1. Mark the start state of A .
 - + 2. Repeat until no new states get marked.
 - + 3. Mark any state that has a transition coming into it from any state that is already marked.
 - + 4. If no accept state is marked, *accept*; otherwise *reject*.”

EQ_{DFA} IS DECIDABLE

- ✗ $L(C) = (L(A) \cap L(B)) \cup (L(A)^c \cap L(B)^c)$
- ✗ F = "On input $\langle A, B \rangle$, where A and B are DFAs:
 - + 1. Construct DFA C as Described.
 - + 2. Run TM T from previous TM on input $\langle C \rangle$.
 - + 3. If T accept, *accept*. If T rejects, *reject*."

DECIDABLE PROBLEMS CONCERNING L_{CFL}

- ✗ Computational problems concerning CFG are as follows:
 - + CFG accepting a string 'w' (A_{CFG}),
 - + CFG is empty (E_{CFG}),
 - + Two CFGs are equivalent (EQ_{CFG}),
 - + Every CFG is decidable(A).

A_{CFG} IS DECIDABLE

- ✗ $A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is an CFG that accepts input string } w \}.$
- ✗
- ✗ $S =$ “On input $\langle G, w \rangle$, where G is a CFG and w is a string:
 - + 1. Convert CFG G to an equivalent grammar in CNF.
 - + 2. List all derivations with $(2n-1)$ steps, where ‘ n ’ is the length of w ; except if $n=0$, then instead list all derivation with the 1 step.
 - + 3. If any of these derivation generates w , *accept*; otherwise *reject*.”

E_{CFG} IS DECIDABLE

- ✗ $E_{CFG} = \{ \langle G \rangle \mid G \text{ is an CFG and } L(G) = \emptyset \}.$
- ✗
- ✗ $R =$ “On input $\langle G \rangle$, where G is a CFG:
 - + 1. Mark all terminal symbols in G .
 - + 2. Repeat until no new variable get marked:
 - + 3. Mark any variable A where G has a rule $A \rightarrow U_1 U_2 \dots U_k$ and each symbol U_1, U_2, \dots, U_k has already been marked.
 - + 4. If any start symbol is not marked, *accept*; otherwise *reject*.”

EQ_{CFG} IS UNDECIDABLE

- ✗ $EQ_{CFG} = \{ \langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H) \}$.
- ✗ The class of context-free languages are not closed under intersection and complementation operation.
- ✗ Hence the ways we have proved for FA with the help of intersection and complementation operation will not work for CFG.
- ✗ So it turns out to be difficult for us to find a TM to decide EQ_{CFG} . Hence EQ_{CFG} is undecidable.

A IS DECIDABLE

- ✗ Let G be CFG for A and design a TM M_G that decides A . we build a copy of G into M_G . It works as follows:
- ✗ $M_G =$ “On input ‘ w ’:
 - + 1.Run TM S on input $\langle G, w \rangle$,
 - + 2.If this machine accepts, *accept*; otherwise if it rejects, *reject*.”

THE HALTING PROBLEM

- ✗ Not all the problems are solved by a computer.
- ✗ Now we will come across several problems that are computational unsolvable.
- ✗ Eg. TM accepting a string w (A_{TM}).
- ✗ $A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$
- ✗ $U =$ “On input $\langle M, w \rangle$, where M is a TM and w is a string;
 - + 1. Simulate M on input w .
 - + 2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*.”
- ✗ U is not a decider as it may loop on input w and don't halt. (*Halting Problem*)
- ✗ Hence U is not a decider but a *recognizer* for A_{TM} .
- ✗ We conclude that A_{TM} is undecidable.

A TURING-UNRECOGNIZABLE LANGUAGE

- ✗ There are some problems that are even not recognized by a TM.
- ✗ Eg. A_{TM} is a Turing unrecognizable language (i.e. the compliment of A_{TM}).
- ✗ If the complement of a language is recognized by a TM, then the language is called as co Turing recognizable.
- ✗ Theorem: “A language is decidable iff both it and its compliment is Turing Recognizable.”
- ✗ Proof:(Forward proof)
 - + (Method1). A is a decidable language (implies A is also recognizable) then A' is also decidable(compliment of a decidable language is also decidable).
 - + If A' is decidable, implies A' is also recognizable.
- ✗ (Backward proof)
 - + (Method2). A and A' are Turing recognizable (represented by M_1, M_2). Then is designed as a decider as follows:
 - + M=“On input w:
 - ✗ Run both M_1 and M_2 on input w in parallel.
 - ✗ If M_1 accepts, *accept*; if M_2 accepts, *reject*.”



Undecidability

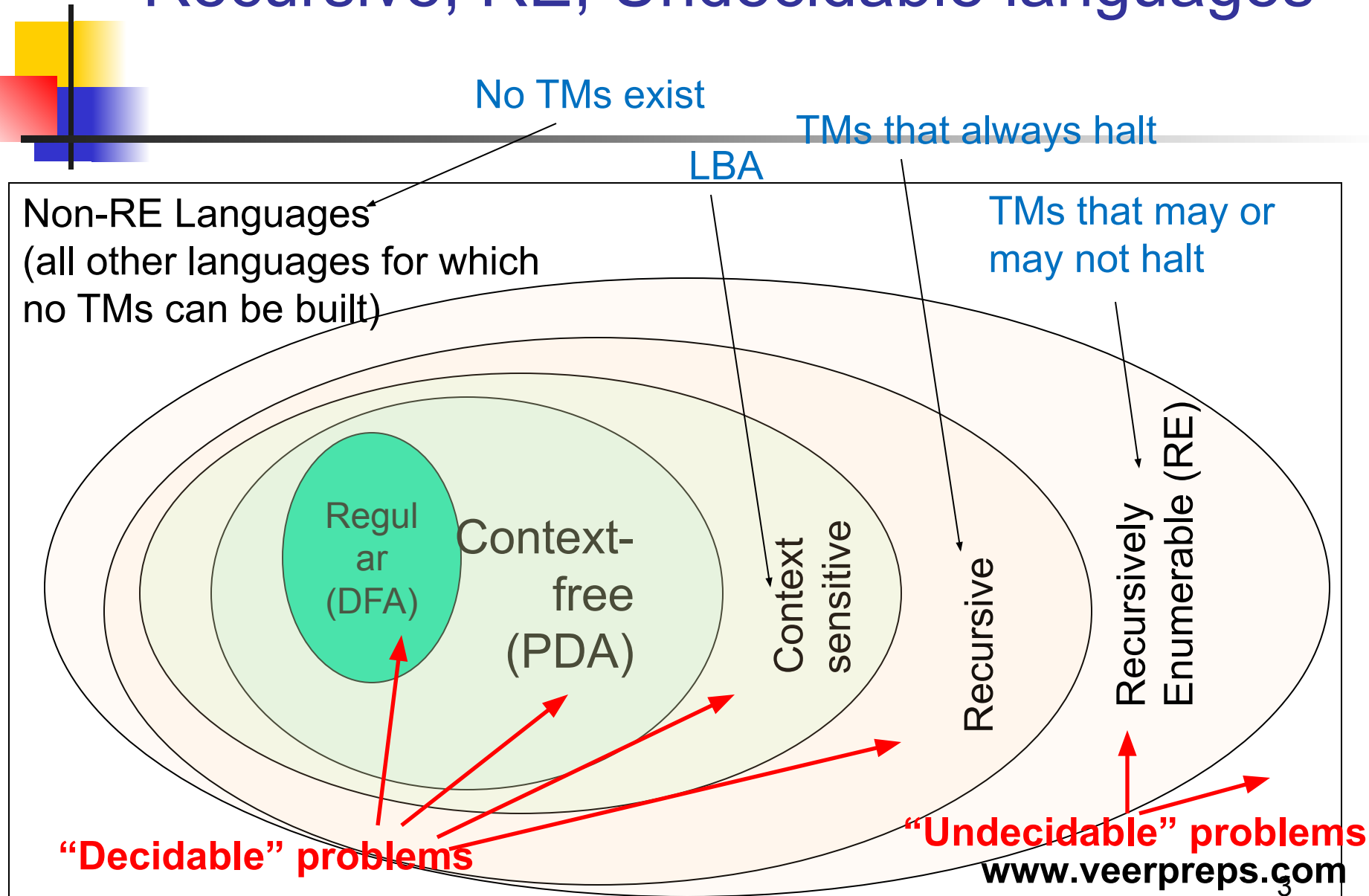
Reading: Chapter 8 & 9



Decidability vs. Undecidability

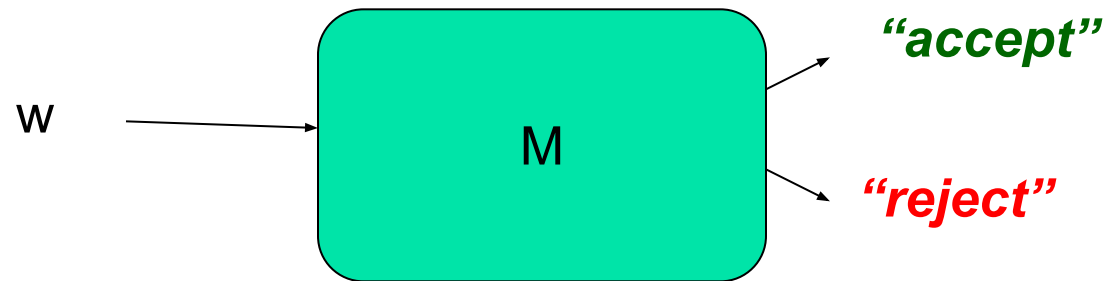
- There are two types of TMs (based on halting):
 - (*Recursive*)
 - TMs that *always* halt**, no matter accepting or non-accepting \equiv **DECIDABLE PROBLEMS**
 - (*Recursively enumerable*)
 - TMs that *are guaranteed to halt only on acceptance***. If non-accepting, it may or may not halt (i.e., could loop forever).
- **Undecidability:**
 - Undecidable problems are those that are not recursive

Recursive, RE, Undecidable languages



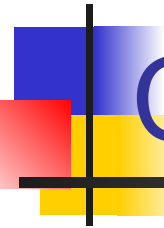
Recursive Languages & Recursively Enumerable (RE) languages

- Any TM for a Recursive language is going to look like this:



- Any TM for a Recursively Enumerable (RE) language is going to look like this:



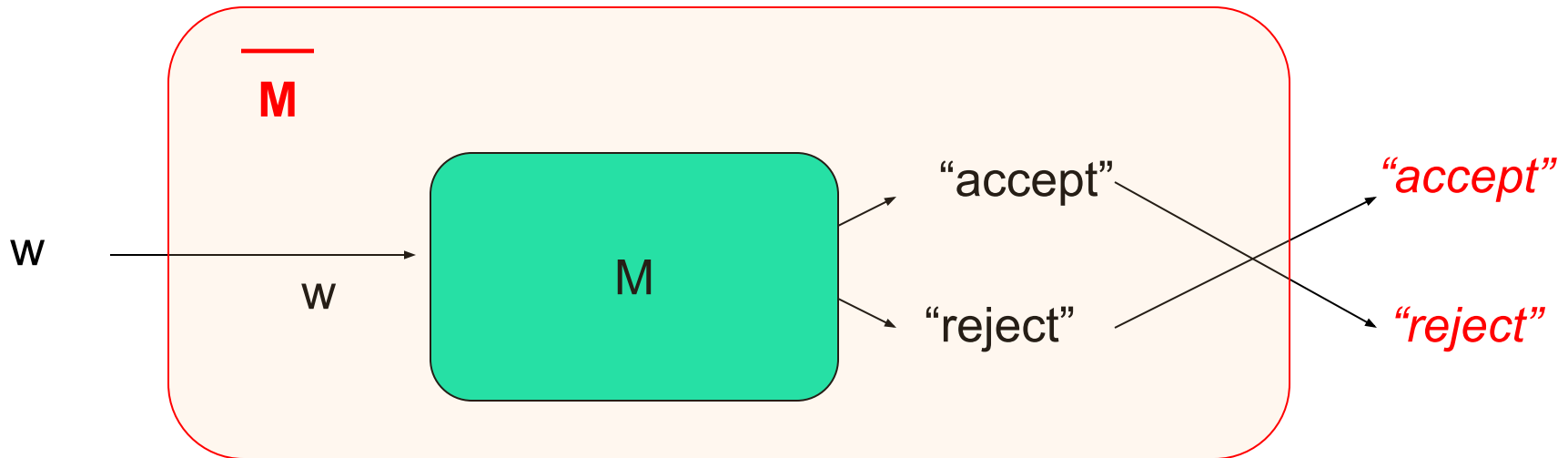


Closure Properties of:

- the Recursive language class, and
- the Recursively Enumerable language class

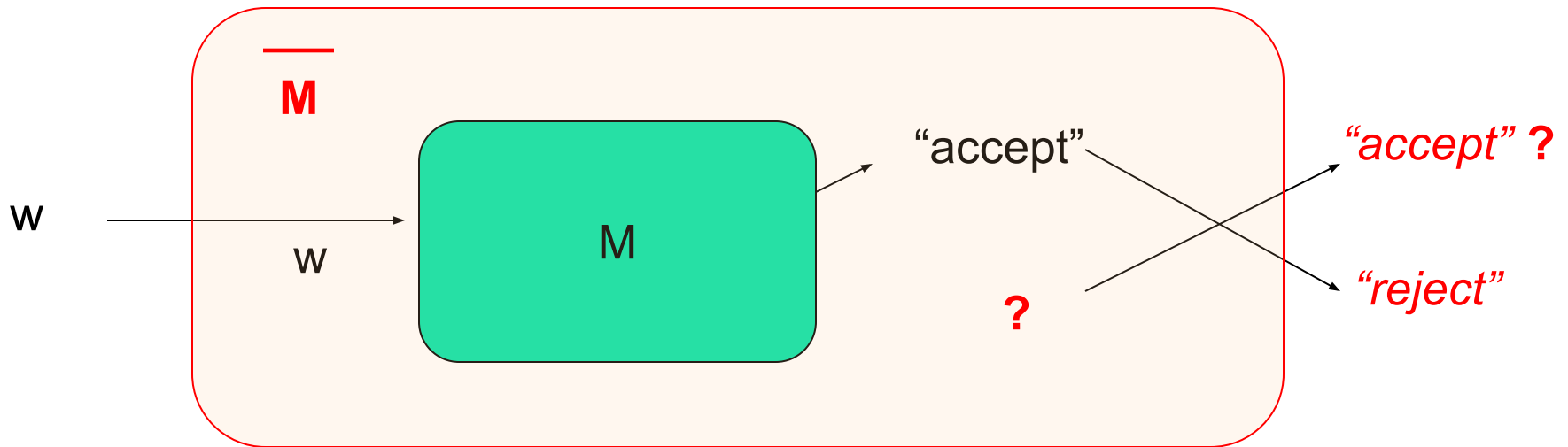
Recursive Languages are closed under complementation

- If L is Recursive, \overline{L} is also Recursive



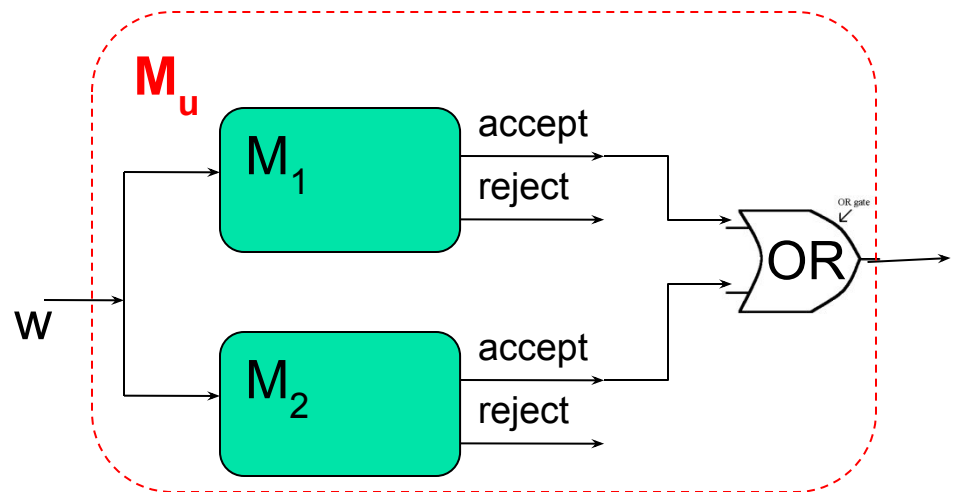
Are Recursively Enumerable Languages closed under complementation? (NO)

- If L is RE, \overline{L} need not be RE



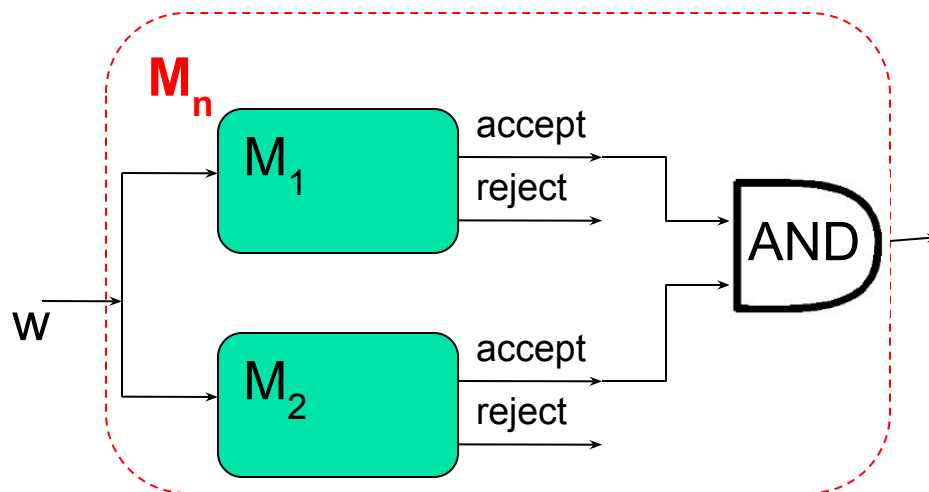
Recursive Langs are closed under Union

- Let $M_u = \text{TM for } L_1 \cup L_2$
- M_u construction:
 1. Make 2-tapes and copy input w on both tapes
 2. Simulate M_1 on tape 1
 3. Simulate M_2 on tape 2
 4. If either M_1 or M_2 accepts, then M_u accepts
 5. Otherwise, M_u rejects.



Recursive Langs are closed under Intersection

- Let $M_n = \text{TM for } L_1 \cap L_2$
- M_n construction:
 1. Make 2-tapes and copy input w on both tapes
 2. Simulate M_1 on tape 1
 3. Simulate M_2 on tape 2
 4. If M_1 AND M_2 accepts, then M_n accepts
 5. Otherwise, M_n rejects.





Other Closure Property Results

- Recursive languages are also closed under:
 - Concatenation
 - Kleene closure (star operator)
 - Homomorphism, and inverse homomorphism
 - RE languages are closed under:
 - Union, intersection, concatenation, Kleene closure
-
- RE languages are *not* closed under:
 - complementation



“Languages” vs. “Problems”

A “language” is a set of strings

Any “problem” can be expressed as a set of all strings that are of the form:

- “<input, output>”

e.g., Problem (a+b) \equiv Language of strings of the form { “a#b, a+b” }

\Rightarrow Every problem also corresponds to a language!!

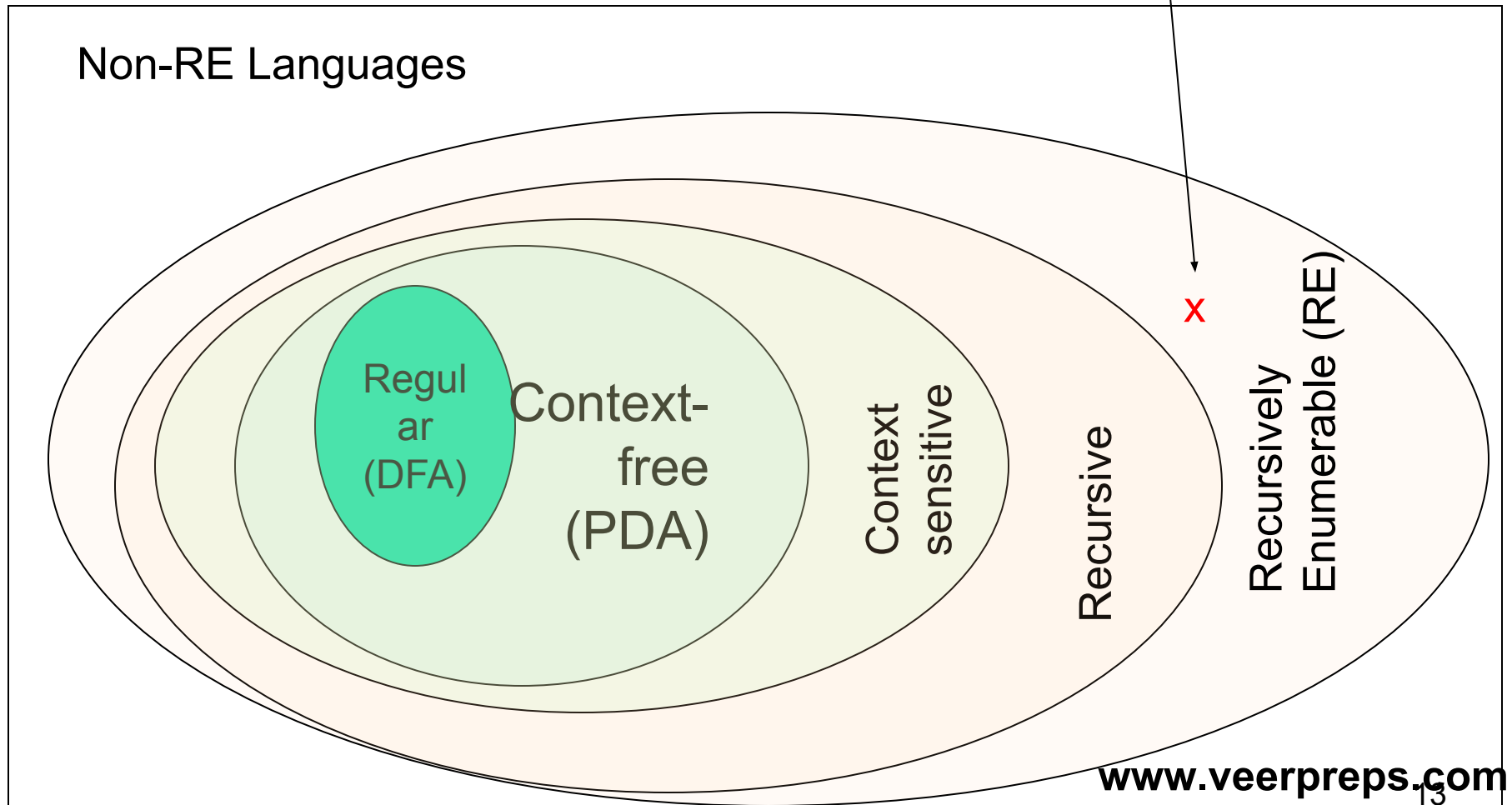
Think of the language for a “problem” \equiv a *verifier* for the problem



The Halting Problem

**An example of a recursive
enumerable problem that is
also undecidable**

The Halting Problem

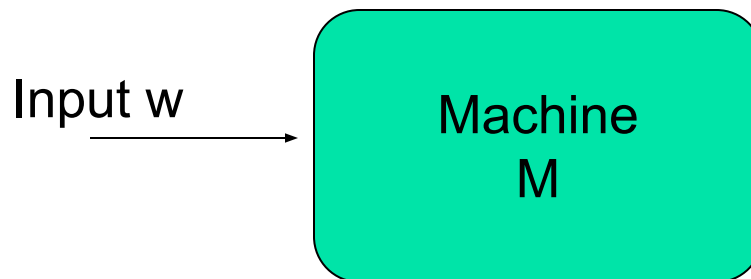




What is the Halting Problem?

Definition of the “halting problem”:

- *Does a given Turing Machine M halt on a given input w ?*



The Universal Turing Machine

- Given: TM **M** & its input **w**
- Aim: Build another TM called “H”, that will output:
 - “*accept*” if M accepts w, and
 - “*reject*” otherwise

- An algorithm for H:
 - Simulate M on w

Implies: H is in RE

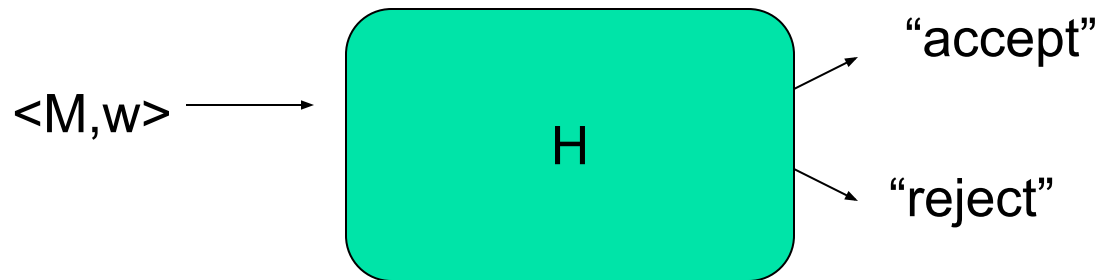
$$\begin{aligned} \text{■ } H(\langle M, w \rangle) = \begin{cases} \text{accept,} & \text{if } M \text{ accepts } w \\ \text{reject,} & \text{if } M \text{ does not accept } w \end{cases} \end{aligned}$$

Question: If M does *not* halt on w, what will happen to H?



A Claim

- Claim: No H that is always guaranteed to halt, can exist!
- Proof: (Alan Turing, 1936)
 - By contradiction, let us assume H exists

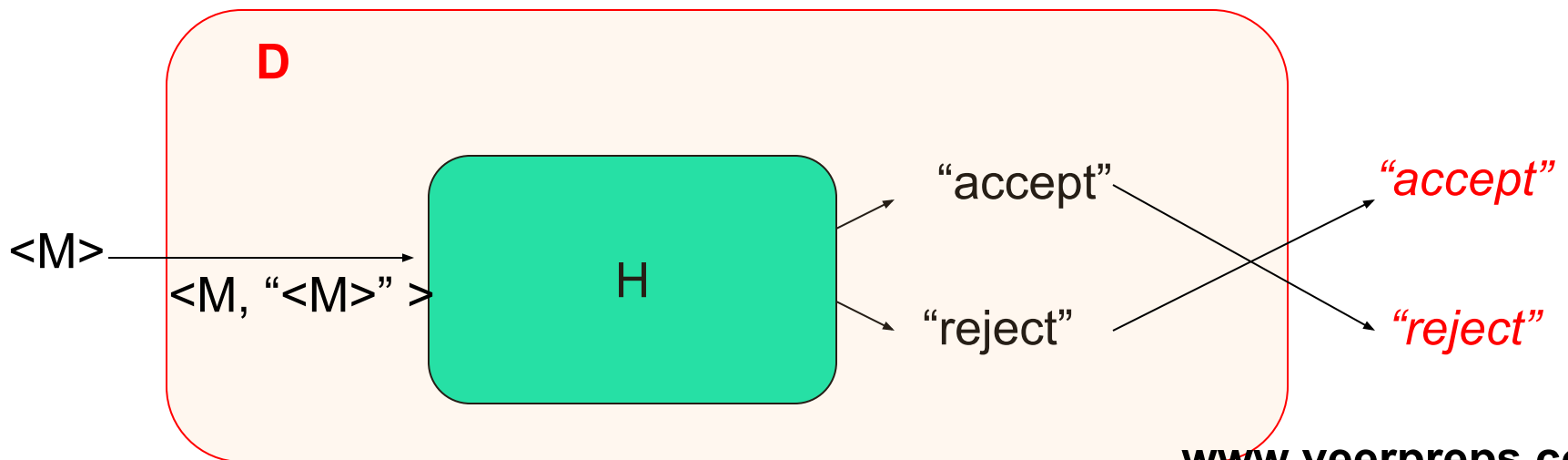


Therefore, if H exists \square D also should exist.

But can such a D exist? (if not, then H also cannot exist)

HP Proof (step 1)

- Let us construct a new TM **D** using H as a subroutine:
 - On input $\langle M \rangle$:
 1. Run H on input $\langle M, \langle M \rangle \rangle$; //(i.e., run M on M itself)
 2. Output the *opposite* of what H outputs;





HP Proof (step 2)

- The notion of inputting “<M>” to M itself
 - A program can be input to itself (e.g., a compiler is a program that takes any program as input)

$$D(<M>) = \begin{cases} \text{accept,} & \text{if M does not accept } <M> \\ \text{reject,} & \text{if M accepts } <M> \end{cases}$$

Now, what happens if D is input to itself?

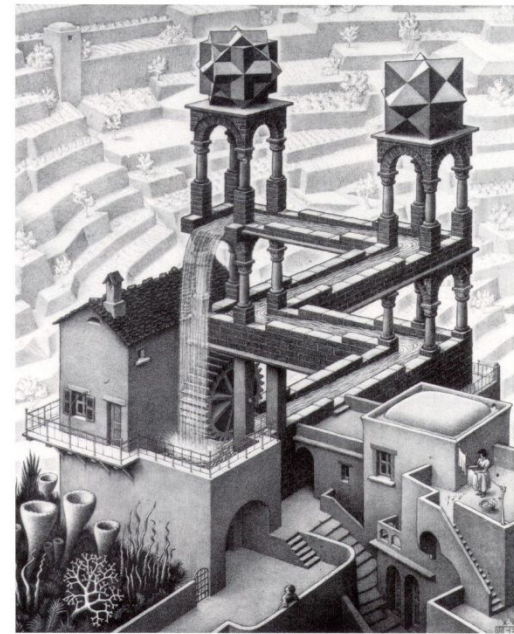
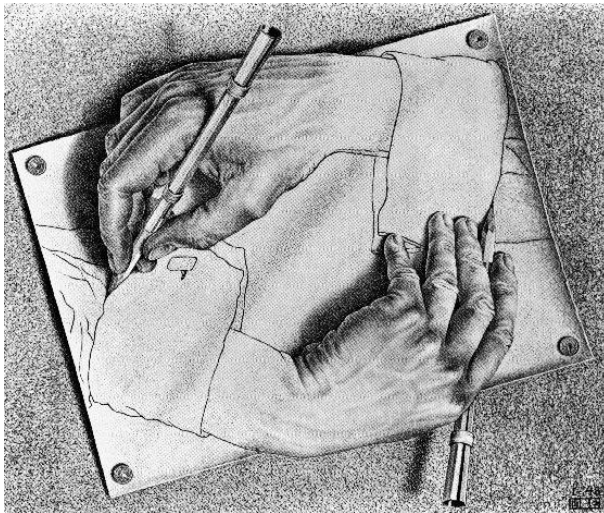
$$D(<D>) = \begin{cases} \text{accept,} & \text{if D does not accept } <D> \\ \text{reject,} & \text{if D accepts } <D> \end{cases}$$

A contradiction!!!

==> Neither D nor H can exist

Of Paradoxes & Strange Loops

E.g., Barber's paradox, Achilles & the Tortoise (Zeno's paradox)
MC Escher's paintings



A fun book for further reading:

"Godel, Escher, Bach: An Eternal Golden Braid"

by Douglas Hofstadter (Pulitzer winner, 1980)

www.veerpreps.com



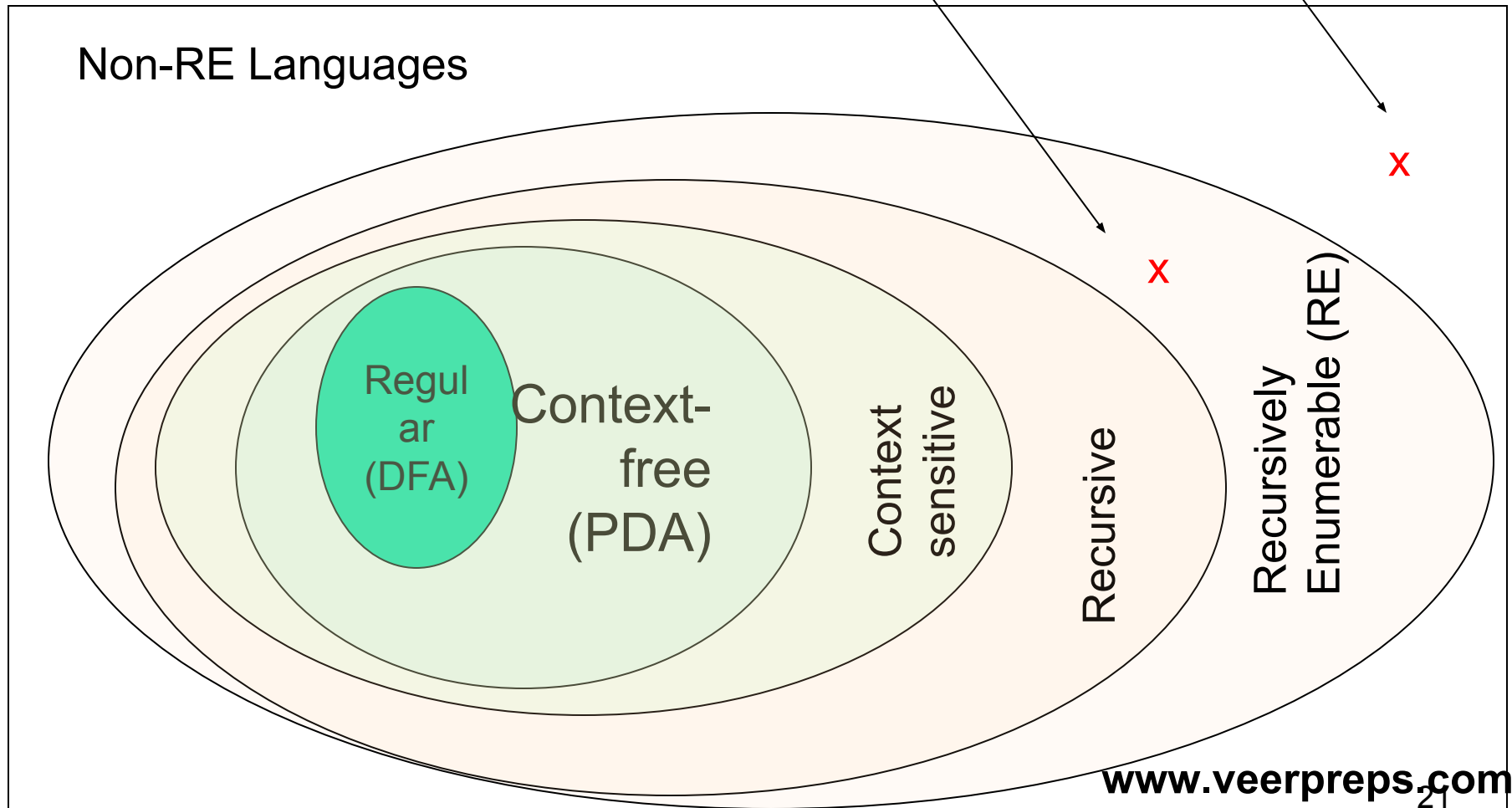
The Diagonalization Language

**Example of a language that is
not recursive enumerable**

(i.e, no TMs exist)

The Diagonalization language

The Halting Problem





A Language about TMs & acceptance

- Let L be the language of all strings $\langle M, w \rangle$ s.t.:
 1. M is a TM (coded in binary) with input alphabet also binary
 2. w is a binary string
 3. M accepts input w .



Enumerating all binary strings

- Let w be a binary string
- Then $1w \equiv i$, where i is some integer
 - E.g., If $w=\epsilon$, then $i=1$;
 - If $w=0$, then $i=2$;
 - If $w=1$, then $i=3$; so on...
- If $1w \equiv i$, then call w as the i^{th} word or i^{th} binary string, denoted by w_i .
- **\Rightarrow A canonical ordering of all binary strings:**
 - $\{\epsilon, 0, 1, 00, 01, 10, 11, 000, 100, 101, 110, \dots\}$
 - $\{w_1, w_2, w_3, w_4, \dots, w_i, \dots\}$



Any TM M can also be binary-coded

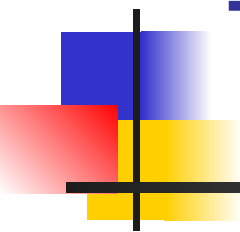
- $M = \{ Q, \{0,1\}, \Gamma, \delta, q_0, B, F \}$
 - Map all states, tape symbols and transitions to integers (\Rightarrow binary strings)
 - $\delta(q_i, X_j) = (q_k, X_l, D_m)$ will be represented as:
 - $\Rightarrow 0^i 1 0^j 1 0^k 1 0^l 1 0^m$
- Result: Each TM can be written down as a long binary string
- \Rightarrow Canonical ordering of TMs:
 - $\{M_1, M_2, M_3, M_4, \dots, M_i, \dots\}$



L_d is not RE (i.e., has no TM)

- Proof (by contradiction):
- Let M be the TM for L_d
- $\implies M$ has to be equal to some M_k s.t.
$$L(M_k) = L_d$$
- \implies Will w_k belong to $L(M_k)$ or not?
 1. If $w_k \in L(M_k) \implies T[k,k]=1 \implies w_k \notin L_d$
 2. If $w_k \notin L(M_k) \implies T[k,k]=0 \implies w_k \in L_d$
- A contradiction either way!!

Why should there be
languages that do not have
TMs?

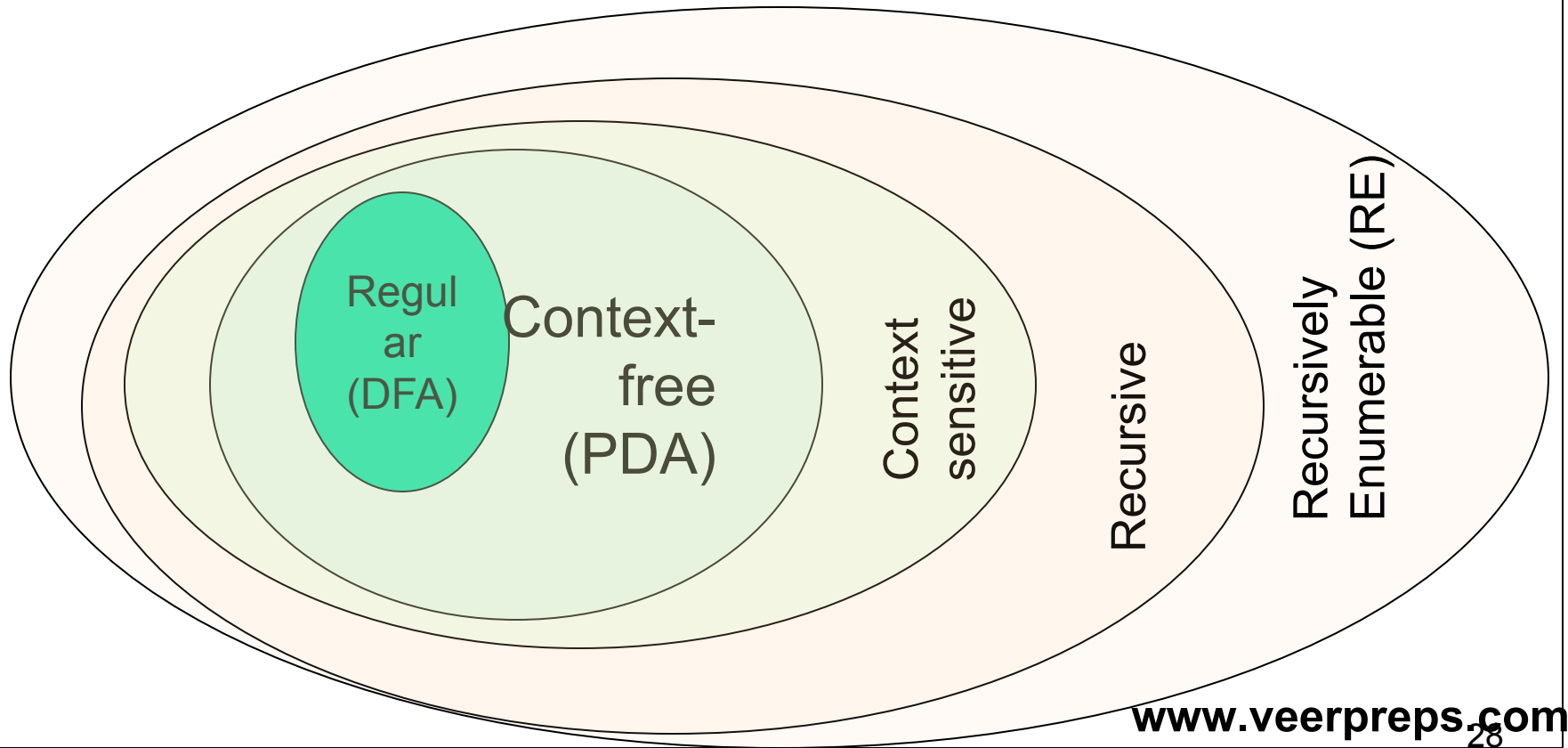


We thought TMs can solve
everything!!

Non-RE languages

How come there are languages here?
(e.g., diagonalization language)

Non-RE Languages

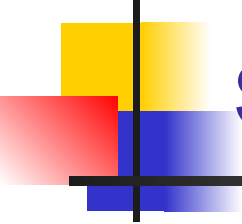




One Explanation

There are more languages than TMs

- By pigeon hole principle:
 - \Rightarrow some languages cannot have TMs
- But how do we show this?
- Need a way to “*count & compare*” two infinite sets (languages and TMs)



How to count elements in a set?

Let A be a set:

- If A is finite \implies counting is trivial
- If A is infinite \implies how do we count?
- And, how do we compare two infinite sets by their size?

Cantor's definition of set "size" for infinite sets (1873 A.D.)

Let $N = \{1, 2, 3, \dots\}$ (all natural numbers)

Let $E = \{2, 4, 6, \dots\}$ (all even numbers)

Q) Which is bigger?

■ A) Both sets are of the same size

■ "**Countably infinite**"

■ Proof: Show by one-to-one, onto set correspondence from

$N \Rightarrow E$

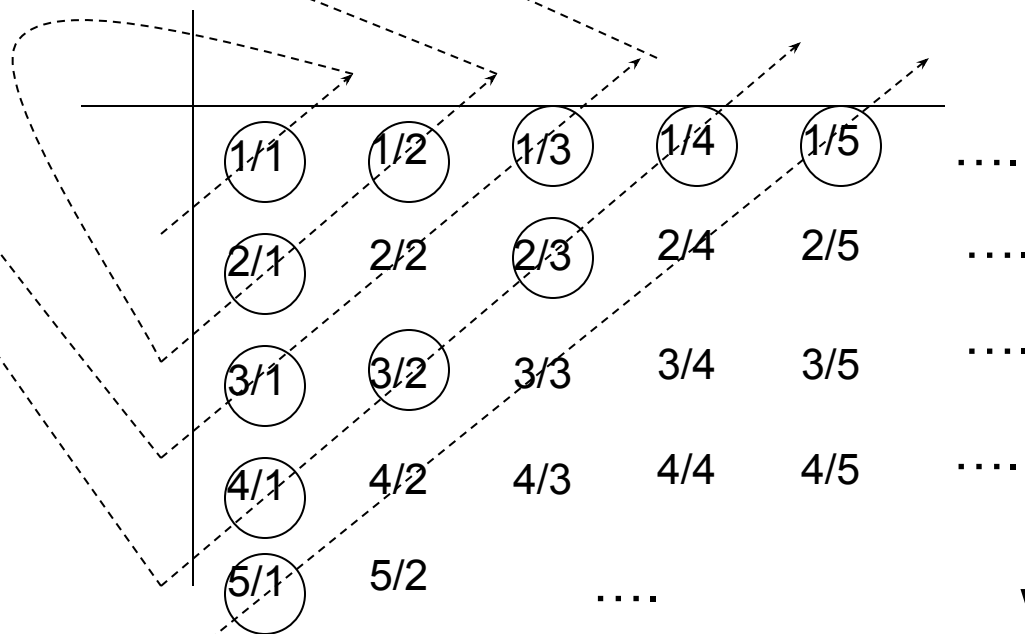
n	$f(n)$
1	2
2	4
3	6
⋮	⋮
⋮	⋮

i.e, for every element in N ,
there is a unique element in E ,
and vice versa.



Example #2

- Let Q be the set of all rational numbers
- $Q = \{ m/n \mid \text{for all } m, n \in \mathbb{N} \}$
- Claim: Q is also countably infinite; $\Rightarrow |Q| = |\mathbb{N}|$



Really, really big sets!
(even bigger than countably infinite sets)



Uncountable sets

Example:

- Let R be the set of all real numbers
- Claim: R is uncountable

n	$f(n)$
1	3 . <u>1</u> 4 1 5 9 ...
2	5 . 5 <u>5</u> 5 5 5 ...
3	0 . 1 2 <u>3</u> 4 5 ...
4	0 . 5 1 4 <u>3</u> 0 ...
.	
.	
.	

Build x s.t. x cannot possibly
occur in the table

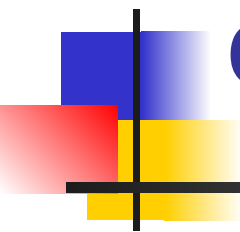
E.g. $x = 0 . 2 6 4 4 \dots$



Therefore, some languages cannot have TMs...

- The set of all TMs is countably infinite
- The set of all Languages is uncountable
- \implies There should be some languages without TMs (by PHP)

The problem reduction technique, and reusing other constructions



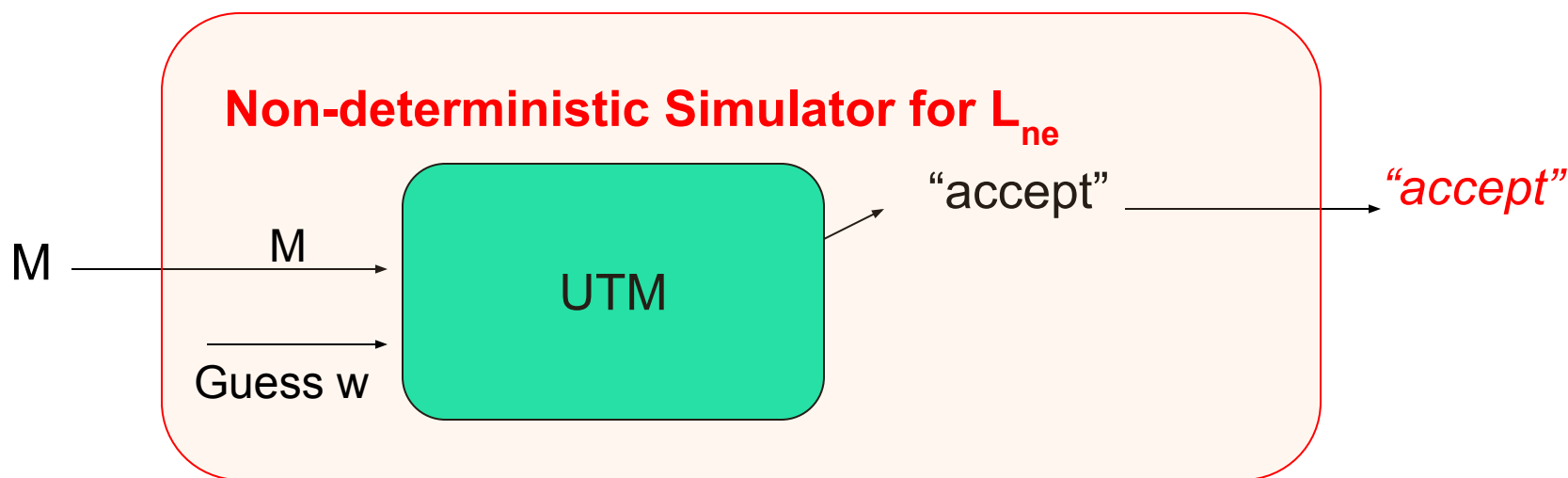


Languages that we know about

- *Language of a Universal TM (“UTM”)*
 - $L_u = \{ \langle M, w \rangle \mid M \text{ accepts } w \}$
 - Result: L_u is in RE but not recursive
- *Diagonalization language*
 - $L_d = \{ w_i \mid M_i \text{ does not accept } w_i \}$
 - Result: L_d is non-RE

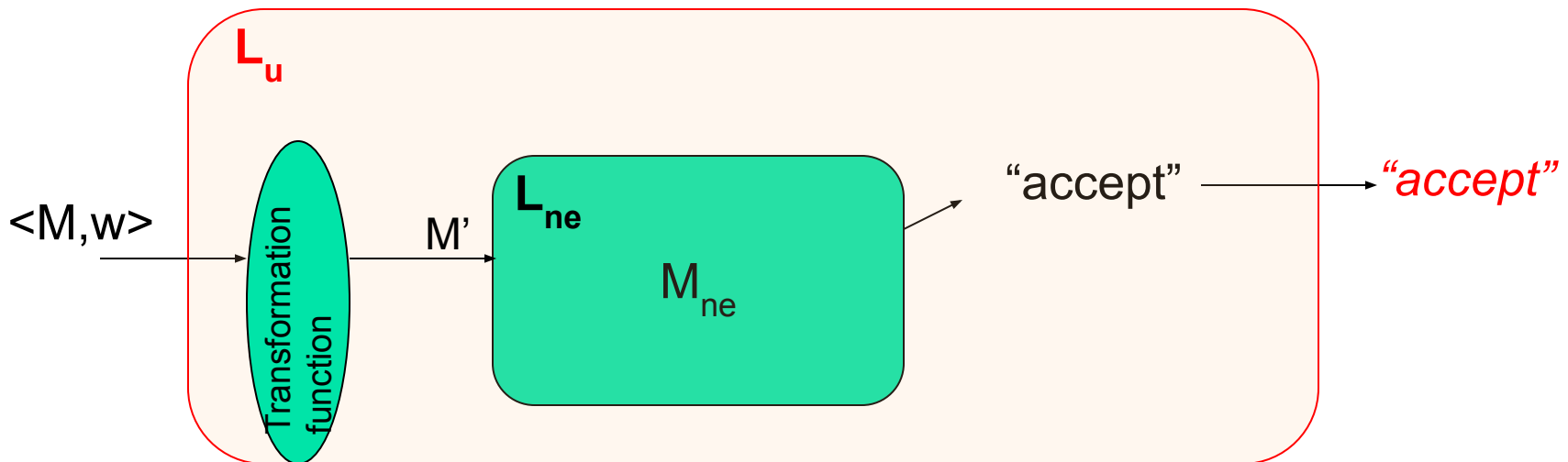
TMs that accept non-empty languages

- $L_{ne} = \{ M \mid L(M) \neq \emptyset \}$
- L_{ne} is RE
- Proof: (build a TM for L_{ne} using UTM)



TMs that accept non-empty languages

- L_{ne} is not recursive
- Proof: (“Reduce” L_u to L_{ne})
 - Idea: M accepts w if and only if $L(M') \neq \emptyset$





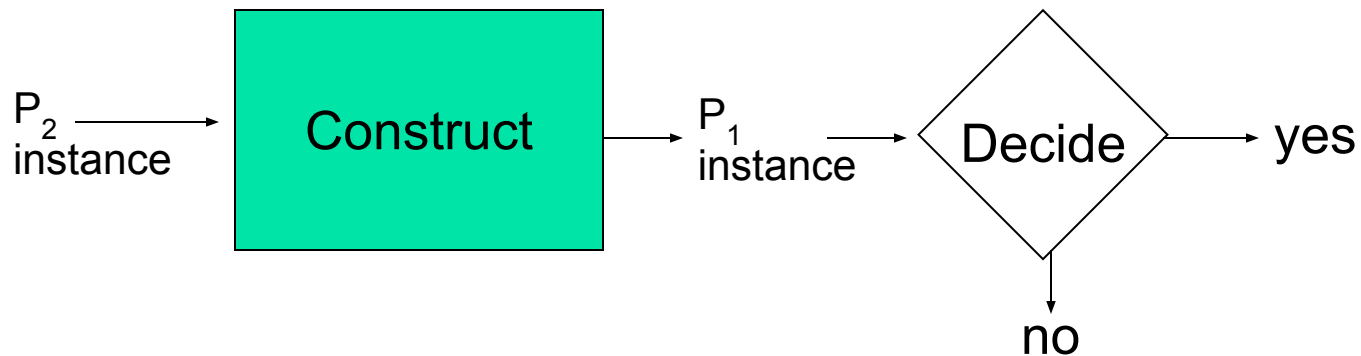
Reducability

- To prove: Problem P_1 is undecidable
- Given/known: Problem P_2 is undecidable
- Reduction idea:
 1. “Reduce” P_2 to P_1 :
 - Convert P_2 ’s input instance to P_1 ’s input instance s.t.
 - i) P_2 decides only if P_1 decides
 2. Therefore, P_2 is decidable
 3. A contradiction
 4. Therefore, P_1 has to be undecidable

The Reduction Technique

Reduce P_2 to P_1 :

Note:
not same as
 $P_1 \implies P_2$



Conclusion: If we could solve P_1 , then we can solve P_2 as well



Summary

- Problems vs. languages
- Decidability
 - Recursive
- Undecidability
 - Recursively Enumerable
 - Not RE
 - Examples of languages
- The diagonalization technique
- Reducability

MEASURING COMPLEXITY

- Let take $A = \{0^k 1^k \mid k \geq 0\}$ and we know A is decidable.
- How much time does a single-tape TM needs to decide A ?
- We do this giving a low level description of TM, so that we can count the number of steps the TM takes when it runs on input w . The description is as follows:
- $M_1 =$ "On input string w :
 - 1. Scan across the tape and *reject* if a 0 is found to the right of 1.
 - 2. Repeat the following if both 0s and 1s remain on the tape.
 - 3. Scan across the tape, crossing off a single 0 and a single 1.
 - 4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."
- For the above TM M_1 we need to find the analyze the algorithm for it as
 - *Worst-case analysis*: We consider the longest running time of all inputs of a particular length.
 - *Average-case analysis*: We consider the average of all the running times of inputs of a particular length.

MEASURING COMPLEXITY (Cont...)

- Let us now find the time taken for each step in the algorithm
- Step 1: requires to check the input is of the form 0^*1^* . This needs n steps, hence it uses $O(n)$.
- Step 2&3: requires to repeatedly scan the tape and cross off a 0 and 1 on each scan. This is an operation of $O(n)$. This process is to be done for half of the input string i.e. $n/2$. Hence the total time required for Step 2&3 is $(n/2) * O(n) = O(n^2)$.
- Step 4: requires a single scan to accept or reject. Hence it is of $O(n)$.
- Hence the total time required to run on an input w is equals to $O(n) + O(n^2) + O(n) = O(n^2)$.

TIME COMPLEXITY CLASS

- Let $t:N \rightarrow N$ be a function, then the time complexity class $TIME(t(n))$ is defined as:
- $TIME(t(n)) = \{L \mid L \text{ is a language decided in an } O(t(n)) \text{ time Turing Machine}\}.$
- The last $A \in TIME(n^2)$ as the A was decided in $O(n^2)$ time.
- We will try to find an algorithm that decides A asymptotically faster.

TIME COMPLEXITY CLASS(Cont...)

- ⦿ We may devise on more algorithm that puts A in $\text{TIME}(n/\log n)$.
- ⦿ M_2 ="On input string w:
 - 1. Scan across the tape and *reject* if a 0 is found to the right of 1.
 - 2. Repeat the following as long as some 0s and 1s remain on the tape.
 - 3. Scan across the tape, checking whether the total # of 0s and 1s remaining is even or odd. If it is odd, *reject*.
 - 4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
 - 5. If # of 0s and # of 1s remain on the tape, *accept*. Otherwise *reject*."

TIME COMPLEXITY CLASS(Cont...)

- Step 1 & 5 is only executed once and are of $O(n)$.
- Step 4 is executed only $1 + \log n$ times as each time the input size is reduced to half of the previous, hence the total time of execution of Step 2, 3 & 4 is $(1 + \log n)O(n)$ or $O(n \log n)$.
- Hence the running time of M_2 is $O(n) + O(n \log n) = O(n \log n)$.
- Hence this TM M_2 runs faster than M_1 and now we can say that $A \in \text{TIME}(n \log n)$.

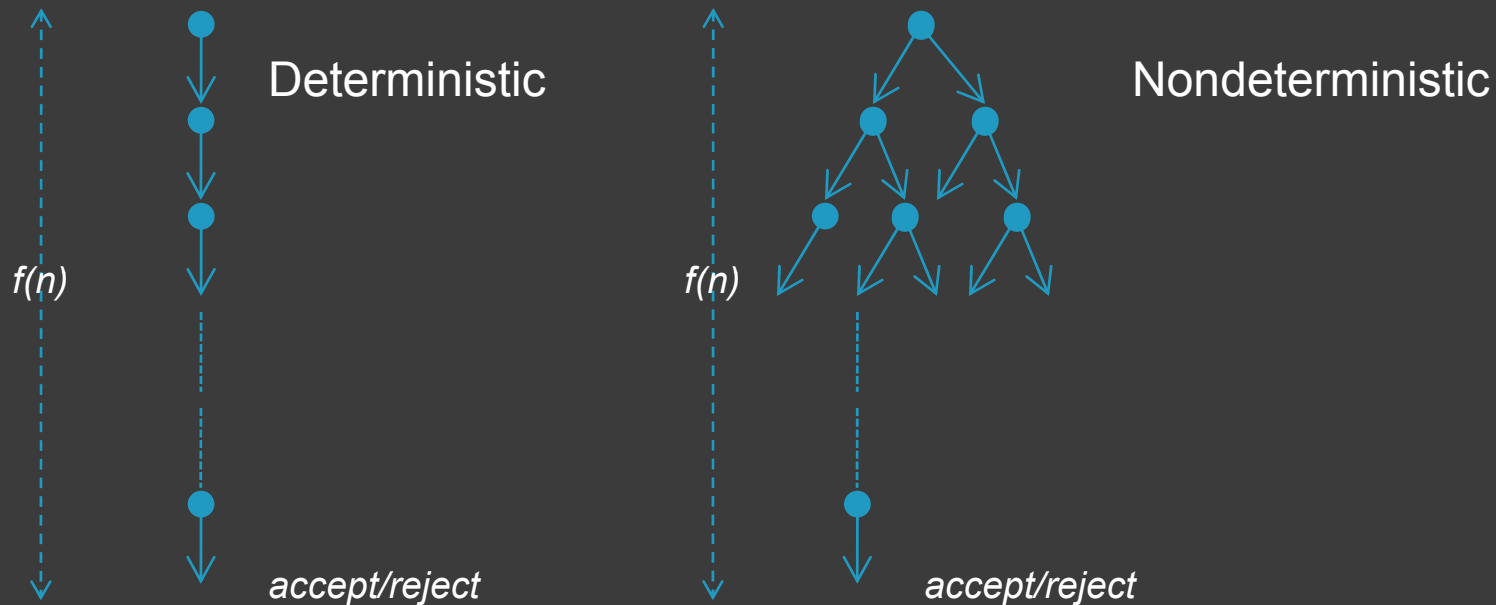
TIME COMPLEXITY CLASS(Cont...)

- ⦿ We may devise on more algorithm that puts A in $\text{TIME}(n)$.
- ⦿ M_3 ="On input string w:
 - 1.Scan across the tape and *reject* if a 0 is found to the right of 1.
 - 2.Scan across the 0s on Tape1 until the first 1. At the same time, copy the 0s onto Tape2.
 - 3.Scan across the 1s on Tape1, until the end of the input. For each 1 read on the Tape1, cross off a 0 on Tape2. If all 0s are crossed off before all the 1s are read, *reject*.
 - 4.If all the 0s have now been crossed off *accept*. If any 0s remain, *reject*."

TIME COMPLEXITY CLASS(Cont...)

- Each Step in the algorithm is of the $O(n)$ and hence the total time of the algorithm is of $O(n)$.
- We can concluding that M_1 a TM with single-tape take $O(n^2)$ to decide a language A and M_3 a TM with two-tape takes $O(n)$ to decide the same language A .
- Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multi-tape TM has an equivalent $O(t(n^2))$ time single-tape TM.

COMPLEXITY RELATIONSHIPS AMONG MODELS



- Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape TM has an equivalent $2^{O(t(n))}$ time deterministic single-tape TM.

THE CLASS P

- ⦿ P is the class of languages that are decidable in polynomial time on a deterministic single-tape TM. In other words, $P = \bigcup_k \text{TIME}(n^k)$.
- ⦿ The class P plays a central role in our theory and is important because:
 - 1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape TM and,
 - P roughly corresponds to the class of problems that are realistically solvable on a computer.

EXAMPLES OF PROBLEMS IN P

- PATH Problem: Is there a path from s to t ?
- RELPRIME Problem: Are two numbers x and y relatively prime?
- These problems are in P class as we can devise an algorithm that can run in polynomial time.

PATH PROBLEM

- ◎ $PATH = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t \}$.
- ◎ $M =$ “On input $\langle G, s, t \rangle$ where G is a directed graph with nodes s and t :
 - 1. Place a mark on node s .
 - 2. Repeat the following until no additional nodes are marked.
 - 3. Scan all the edges of G . If an edge (a, b) is found going from a marked node a to an unmarked node b , mark node b .
 - 4. If t is marked, *accept*. Otherwise, *reject*.”

RELPRIME PROBLEM

- ⊙ RELPRIME = { $\langle x, y \rangle$ | x and y are relatively prime}. We can have and an Euclidean algorithm for this.
- ⊙ E = "On input $\langle x, y \rangle$ where x and y are natural numbers in binary:
 - 1. Repeat until $y = 0$.
 - 2. Assign $x \leftarrow x \bmod y$.
 - 3. Exchange x and y .
 - 4. Output x ."
- ⊙ Algorithm R solves RELPRIME using E as a subroutine:
- ⊙ R = "On input $\langle x, y \rangle$, where x and y are natural numbers in binary:
 - 1. Run E on $\langle x, y \rangle$.
 - 2. If the result is 1, *accept*. Otherwise, *reject*."

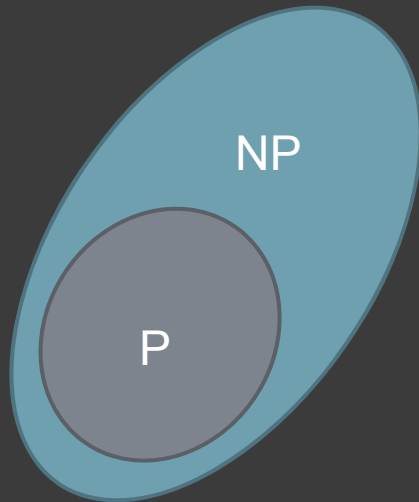
THE CLASS NP

- A language is in NP iff it is decided by some nondeterministic polynomial time TM.
- These are the problems that are decidable in exponential time.
- $N =$ "On input w of length n :
 - 1. Nondeterministically select string c of length n^k .
 - 2. Run V on input $\langle w, c \rangle$.
 - 3. If V accepts, *accept*. Otherwise, *reject*."
- $V =$ "On input $\langle w, c \rangle$, where w and c are strings:
 - 1. Simulate N on input w , treating each symbol of c as a description of the nondeterministic choice to make at each step.
 - 2. If this branch of N 's computation accepts, *accept*. Otherwise, *reject*."
- Here c is called the *certificate or proof of membership in language* and V is the verifier of *polynomial time verifier*.
- We define nondeterministic time complexity class $NTIME(t(n)) = \{L \mid L \text{ is a language decided by a } O(t(n)) \text{ time nondeterministic TM}\}$. $NP = \bigcup_k NTIME(n^k)$.

EXAMPLES OF PROBLEMS IN NP

- HAMPATH Problem: Is there a Hamiltonian path from s to t in a directed graph G ?
 - $\text{HAMPATH} = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t \}$.
- COMPOSITES Problem: Can a number x be expressed as a product of two numbers p and $q \mid p, q > 1$?
 - $\text{COMPOSITES} = \{ x \mid x = pq, \text{ for integer } p, q > 1 \}$.
- CLIQUE Problem: Whether a graph G contains a clique of specified size (A clique in an undirected graph is a sub-graph, wherein every two nodes are connected by an edge).
 - $\text{CLIQUE} = \{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \}$.
- SUBSET-SUM Problem: Can a number be represented as a sum of numbers from a set of numbers?
 - $\text{SUBSET-SUM} = \{ \langle S, t \rangle \mid S = \{x_1, \dots, x_k\} \text{ and for some } \{y_1, \dots, y_i\} \text{ subset of } \{x_1, \dots, x_k\}, \text{ we have } \sum y_i = t \}$.
- These problems are in NP class as we can devise an algorithm that can polynomially verify them.

P VERSUS NP QUESTION



$$NP \subseteq EXPTIME = \bigcup_k TIME(2^{n^k})$$

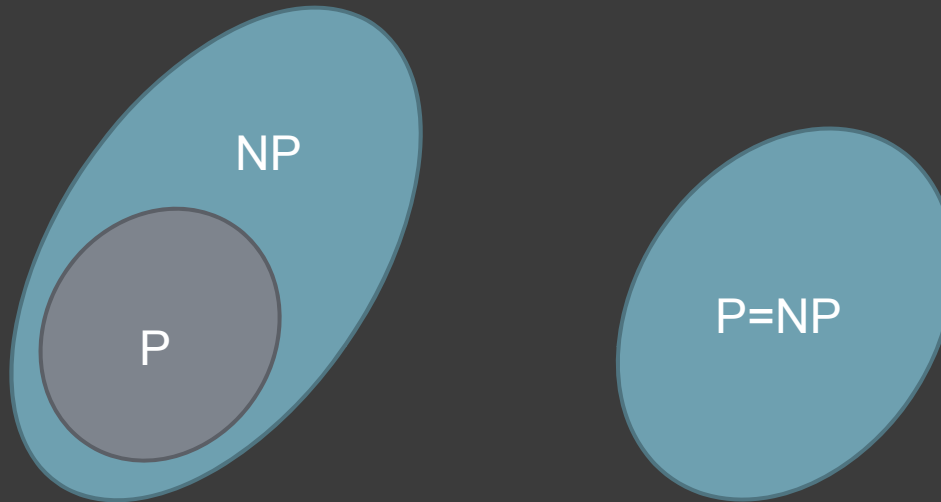
$$P \subseteq POLYTIME = \bigcup_k TIME(n^k)$$

- P= The class of language where membership can be *decided* quickly (polynomial time).
- NP= The class of language where membership can be *verified* quickly (but decided in exponential time).

NP-COMPLETENESS

- There are some problems in NP whose individual complexity is related to that of the entire class.
- If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable.
- These problems are called as *NP-complete*.
- If any problem in NP requires more than polynomial time, an NP-complete one does.
- An attempt to prove that P and NP are equal only needs to find a polynomial time algorithm for an NP-complete problem to achieve this goal.

DEFINITION OF NP-COMPLETENESS



- ⦿ A language B is NP-complete if it satisfies two conditions:
 - 1. B is in NP, and
 - 2. Every A in NP is polynomial time reducible to B .
- ⦿ If B is NP-complete and $B \in P$, then $P=NP$.

POLYNOMIAL TIME REDUCIBILITY

- A function $f : \Sigma^* \rightarrow \Sigma^*$ is a *polynomial time computable function* if some polynomial time TM M exist that halts with just $f(w)$ on its tape, when started on any input w .
- A language A is *polynomial time mapping reducible* or simply *polynomial time reducible*, to language B written $A \leq_p B$, if there is a polynomial time computable function $f : \Sigma^* \rightarrow \Sigma^*$, where for every w , $w \in A \Leftrightarrow f(w) \in B$. The function f is called the *polynomial time reduction* of A to B .
- If $A \leq_p B$ and $B \in P$, then $A \in P$.
- $N =$ "On input w :
 - 1. Compute $f(w)$.
 - 2. Run M on input $f(w)$ and output whatever M outputs."
- Clearly if $w \in A$, then $f(w) \in B$ because f is a reduction from A to B . Thus M accepts $f(w)$ whenever $w \in A$. Therefore N works as desired in polynomial time.

PROBLEMS IN NP-COMplete

- SAT Problem: *satisfiability* problem.
 - $SAT = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula} \}$.
 - A Boolean formula is satisfiable if some assignments of 0s and 1s to the variables makes the formula ϕ evaluate 1.
- 3SAT = $\{ \langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf formula} \}$.
 - It is called 3cnf as all the clause in the Boolean formula have three literals.
 - Eg.
$$(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_3} \vee \overline{x_5} \vee \overline{x_6}) \wedge (\overline{x_3} \vee \overline{x_6} \vee \overline{x_4}) \wedge (\overline{x_4} \vee \overline{x_5} \vee \overline{x_6})$$

REDUCIBLE PROBLEMS

- ③ 3SAT is polynomial time reducible to CLIQUE.
 - CLIQUE being a problem in NP can be used as reduction for 3SAT problem
- ③ THE COOK-LEVIN THEOREM
 - SAT is NP-complete.
- ③ Additional NP-complete problems:
 - Vertex cover Problem.

Completion of Theory of Computation

REDUCIBILITY

2

- It is a technique used to prove that some problem are computationally unsolvable.
- It involves a process called *reduction*, which is way of converting one problem 'A' into second problem 'B' in such a way that a solution to the second problem 'B' can be used to solve the first problem 'A' (A is reduced to B).
- Eg: A="The problem of going from BAM to LA" can be reduced to B="Buying the air tickets between the two cities". And this intern can be reduced to C=" Earning money" and further can be reduced to D="Finding a job".
- So the chain of problem over here is $A \rightarrow B \rightarrow C \rightarrow D$. And solution to the problem D act as the solution to A.

UNDECIDABLE PROBLEMS FROM LANGUAGE THEORY

3

- The solution of A_{TM} (is undecidable) is used to find the solution of these problems.
- The problem falling under this categories are as follows:
 - TM halting on input 'w' ($HALT_{TM}$),
 - TM is empty (E_{TM}),
 - TM simulating a FA ($REGULAR_{TM}$),
 - Two TMs are equivalent (EQ_{TM}).

HALT_{TM} IS UNDECIDABLE

4

- $S =$ “On input $\langle M, w \rangle$, an encoding of TM M and a string w ;
 - 1. Run TM R on input $\langle M, w \rangle$. // R decides HALT_{TM}
 - 2. If R rejects, *reject*.
 - 3. If R accepts, simulates M on w until it halts.
 - 4. If M has accepted, *accept*; if M has rejected, *reject*.”
- Clearly if R decides HALT_{TM} , then S decides A_{TM} . Because A_{TM} is undecidable, HALT_{TM} also must be undecidable.

E_{TM} IS UNDECIDABLE

5

- M_1 = “On input x ; // M_1 accepts only string w ”
 - 1.If $x \neq w$, *reject*.
 - 2.If $x = w$, run M on input w and accept if M does.”
- S = “On input $\langle M, w \rangle$, an encoding of a TM and a string w . ”
 - 1.Use the description of M and w to construct the TM M_1 just described.
 - 2.Run R on input $\langle M_1 \rangle$.
 - 3.If R accepts, *rejects*; if R rejects, *accept*.”
- If R were a decider for E_{TM} , S would be a decider for A_{TM} .
- A decider for A_{TM} cannot exist, so we know that E_{TM} must be undecidable.

REGULAR_{TM} IS UNDECIDABLE

6

- S=“On input $\langle M, w \rangle$, where M is a TM and w is a string.
 - 1. Construct the following TM M_2 .
 - ✦ M_2 = “On input x ;
 - 1. If x has the form $0^n 1^n$, *accept*.
 - 2. If x does not have this form, run M on input w and *accept* if M accepts w .”
 - 2. Run R on input $\langle M_2 \rangle$.
 - 3. If R accepts, *accept*; if R rejects, *reject*.”
- Similarly the language of the TM is CF, is undecidable can also be proved in a similar manner.
- This is Rice's Theorem “testing any property of the languages recognized by TM is undecidable”.

EQ_{TM} IS UNDECIDABLE

7

- $EQ_{TM} = \{ \langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2) \}$
- $S =$ “On input $\langle M \rangle$, where M is a TM;
 - 1. Run R on input $\langle M, M_1 \rangle$, where M_1 is a TM that rejects all inputs.
 - 2. If R accepts, *accept*; if R rejects, *reject*.”
- If R decides EQ_{TM} , S decides E_{TM} . But E_{TM} is undecidable so EQ_{TM} also must be undecidable.

REDUCTION BY COMPUTATION HISTORIES

8

- It is an important technique for proving that A_{TM} is reducible to certain language. E.g. Hilbert's 10th problem (testing for existence of integral roots of a polynomial).
- *Computational history* for a TM on an input is simply the sequence of *configurations* that the machine goes through as it processes the input.
- **Definition:**
- Let M be the TM and 'w' be the input string, then an *accepting computation history* for M on w is a sequence of configurations C_1, C_2, \dots, C_l where
 - C_1 is the start configuration and
 - C_l is an accepting configuration of M and each of C_i legally follows from C_{i-1} according to the rules of M .
- A *rejecting computation history* for M on w is defined similarly, except that
 - C_l is a rejecting configuration.

REDUCTION BY COMPUTATION HISTORIES

9

- Computation histories are finite sequences.
- If M doesn't halt on w , no *accepting* or *rejecting computation history* exist for M on w .
- Deterministic machines have at most one computation histories on any given input.
- Non deterministic machines have more than one computation histories on a single input.
- Here we will be discussing about the undecidability of *linear bounded automaton*.

LINEAR BOUNDED AUTOMATON

10

- LBA is a restricted type of TM where in the tape head isn't permitted to move off the tape containing the input.
- $M = (Q, \Sigma, \Gamma, \delta, q_0, b, \Phi, \$, F)$,
- Where $\Phi, \$$ are the left and right end markers respectively.
- The ID = (q, w, k) where k is an integer from 1 to n (size of w). The value of k changes to $k+1$ if the head moves to right and $k-1$ if the head moves to left.
- The language accepted by a LBA is defined as follows:
 $\{w \in (\Sigma - \{\Phi, \$\})^* \mid (q_0, \Phi w \$, 1) \vdash (q, \alpha, i)\}$
 where $q \in F$ and $1 < i < n$.

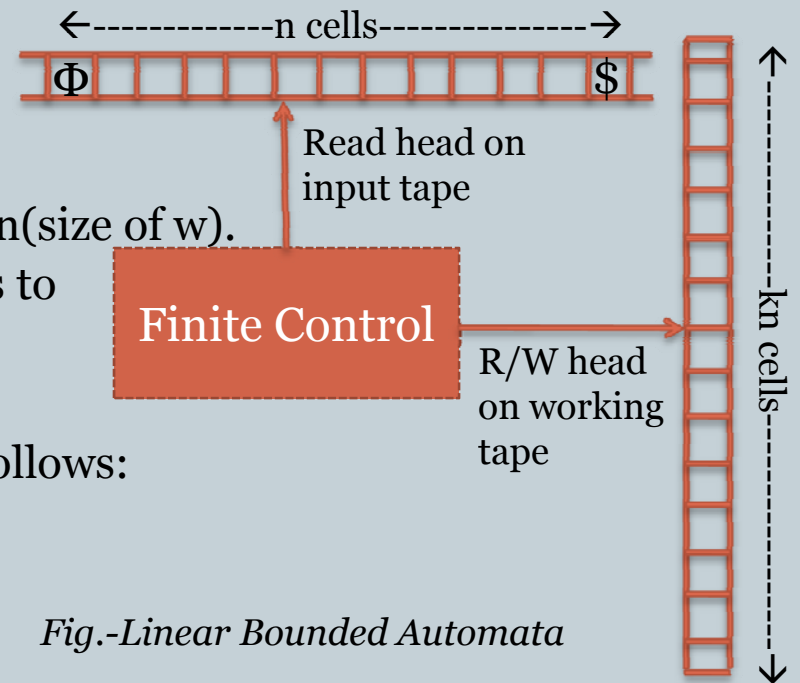


Fig.-Linear Bounded Automata

A_{LBA} IS DECIDABLE

11

- $A_{\text{LBA}} = \{ \langle M, w \rangle \mid M \text{ is an LBA that accepts string } w \}$
- $L =$ “On input $\langle M, w \rangle$, where M is a LBA and w is a string:
 - 1. Simulate M on w for qng^n steps or until it halts.
 - 2. If M halted, *accept* if it has accepted and *reject* if it has rejected. If it has not halted, *reject*.”
- q : # of states in the LBA,
- n : Length of the tape (the head can be only n position)
- g^n : Possible strings of tape symbols appearing on the tape.
- qng^n : Total # of different configurations of M with tape of length n .
- LBAs are different from TMs in a way that, the acceptance problem of the former is decidable.

E_{LBA} IS UNDECIDABLE

12

- We construct a LBA 'B' that accepts the computation histories of w on M .
- The computation history for w on M is a set of configurations C_1, C_2, \dots, C_l , so the input to the LBA B is $x = \#C_1\#C_2\#\dots\#C_l\#$.
- The LBA on input x checks for the following:
 - C_1 is the start configuration for M on w .
 - Each C_{i+1} , legally follows from C_i .
 - C_l is the accepting configuration for M on input w .
- $S =$ "On input $\langle M, w \rangle$, where M is a TM and w is a string:
 - 1. Construct LBA B from M and w as described in the proof idea.
 - 2. Run R on input $\langle B \rangle$.
 - If R reject, *accept*; if R accepts, *reject*."

ALL_{CFG} IS UNDECIDABLE

13

- $ALL_{CFG} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \Sigma^* \}$
- We construct a CFG that accepts all strings over Σ if M rejects w and vice versa.
- An accepting computation history for M on input w appears as follows $\#C_1\#C_2\#\dots\#C_l\#$. So the CFG is designed to generate all strings that:
 - Do not start with C_1 .
 - Do not end with an accepting configuration i.e. C_l .
 - Where some C_i does not properly yield C_{i+1} under the rules of M .
- We construct a PDA 'D' instead of CFG as its construction is easier.
- $S =$ "On input $\langle M, w \rangle$, where M is a TM and w is a string:
 - 1. Construct PDA D from M and w as described in the proof idea.
 - 2. Run R on input $\langle D \rangle$.
 - If R reject, *accept*; if R accepts, *reject*."

POST CORRESPONDENCE PROBLEM (PCP)

14

- It is an *undecidable* problem concerning simple manipulations of string.
- We have a collection of dominos, each containing two strings, one on each side.

$$\left\{ \begin{bmatrix} b \\ ca \end{bmatrix}, \begin{bmatrix} a \\ ab \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} abc \\ c \end{bmatrix} \right\}$$

- The task is to make a list of dominos (reputations permitted) so that the string we are reading off the symbols on the top is the same as the string of symbol on the bottom. This is called as a *match*.

$$\left\{ \begin{bmatrix} a \\ ab \end{bmatrix}, \begin{bmatrix} b \\ ca \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} a \\ ab \end{bmatrix}, \begin{bmatrix} abc \\ c \end{bmatrix} \right\}$$

- There are some collection for which finding a match is impossible.

$$\left\{ \begin{bmatrix} abc \\ ab \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} acc \\ ba \end{bmatrix} \right\}$$

- The problem of PCP is algorithmically unsolvable.
- An instance of PCP is a collection P of dominos:
and a match is a sequence of i_1, i_2, \dots, i_l , where

$$P = \left\{ \begin{bmatrix} t_1 \\ b_1 \end{bmatrix}, \begin{bmatrix} t_2 \\ b_2 \end{bmatrix}, \dots, \begin{bmatrix} t_k \\ b_k \end{bmatrix} \right\}$$

$t_{i1}, t_{i2}, \dots, t_{il} = b_{i1}, b_{i2}, \dots, b_{il}$. The problem is to determine whether P has a match.

- $PCP = \{ \langle P \rangle, \mid P \text{ is an instance of the PCP with a match} \}$
- PCP is undecidable.

MAPPING REDUCIBILITY

15

- A function $f: \Sigma^* \rightarrow \Sigma^*$ is a *computable function* if some TM M , on every input w , halts with $f(w)$ on its tape.
- Eg. All arithmetic operation on integer are computable functions.
- A language A is *mapping reducible* to language B written $A \leq_m B$, if there is a computable function $f: \Sigma^* \rightarrow \Sigma^*$, where for every w , $w \in A \Leftrightarrow f(w) \in B$. The function f is called *reduction* of A to B .
- If $A \leq_m B$ and B is decidable, then A is decidable.
- $N =$ "On input w :
 - 1. Compute $f(w)$.
 - 2. Run M on input $f(w)$ and output whatever M outputs."
- Clearly if $w \in A$, then $f(w) \in B$ because f is a reduction from A to B . Thus M accepts $f(w)$ whenever $w \in A$. Therefore N works as desired.