

Theory of Computation

Introduction

From high level

- From a high level what this course is about?
 - Given a set, say S .
 - Let $A \subseteq S$.
 - Both S and A are well defined.
 - We are given an element $x \in S$, and asked to find whether this x is in A or not.
 - That's all !

Surprises !!

- Surprising things
 - This is related to decision problems.
 - S is set all face images. A is set of images of a particular person. {Face verification}
 - S is set of all graphs. A is set of graphs with Hamiltonian cycle.
 - Sometimes this is an **unsolvable** problem. ??
 - Sometimes this is an **easy** task, sometimes quite a **difficult** one.
 - Recall, $O(n^2)$ is time consuming than $O(n)$ algorithm.

We want general enough set

- Set of strings over some alphabet like {0,1}.
- For example set of strings that end with a 0,
 $\{0, 00, 10, 000, 010, 100, 110, \dots\}$
- Eg2: Each string in the set can be seen as a positive binary number and let the set be the set of prime numbers.
 - Given a number (binary string of 0s and 1s) you want to find whether this is in the set (prime) or not (not a prime).

Why strings are chosen?

- Any data element like number or image or anything can be represented as a string.
 - Can we say DNA code represents a human being?
- Even a method which solves a problem can be represented as a string.
- A proof can be represented as a string.
- So strings over an alphabet gives us power to represent the things... that is we have *languages of strings to represent the things*.

What will you learn from this course?

- How to define a computer? **Automata theory**
- Are there problems that a computer cannot solve? If so, can we find one such problem? **Computability theory**
- For problems that a computer can solve, some problems are easy (e.g., sorting) and some are difficult (e.g., time-table scheduling). Any systematic way to classify problems? **Complexity theory**

Syllabus

- **Syllabus:**
- **Unit – 1 [8 Hours]:** Introduction - Alphabets, Strings and Languages, Automata and Grammars; Deterministic finite Automata (DFA) - Formal Definition, Simplified notation, State transition graph, Transition table, Language of DFA; Nondeterministic finite Automata (NFA) - NFA with epsilon transition, Language of NFA, Equivalence of NFA and DFA, Minimization of Finite Automata, Distinguishing one string from other
- **Unit – 2 [8 Hours]:** Regular Expression (RE) - Definition, Operators of regular expression and their precedence, Algebraic laws for Regular expressions; Relation with FA - Regular expression to FA, DFA to Regular expression; Non Regular Languages - Pumping Lemma for regular Languages, Application of Pumping Lemma; Properties - Closure properties of Regular Languages, Decision properties of Regular Languages, Applications and Limitation of FA
- **Unit – 3 [8 Hours]:** Context Free Grammar (CFG) - Definition, Examples, Derivation, Derivation trees; Ambiguity in Grammar - Inherent ambiguity, Ambiguous to Unambiguous CFG; Normal forms for CFGs - Useless symbols, Simplification of CFGs, CNF and GNF; Context Free Languages (CFL) - Closure properties of CFLs, Decision Properties of CFLs, Emptiness, Finiteness and Membership, Pumping lemma for CFLs
- **Unit – 4 [8 Hours]:** Push Down Automata (PDA) - Description and definition, Instantaneous Description, Language of PDA; Variations of PDA - Acceptance by Final state, Acceptance by empty stack, Deterministic PDA; Equivalence of PDA and CFG - CFG to PDA and PDA to CFG
- **Unit – 5 [8 Hours]:** Turing machines (TM) - Basic model, definition and representation, Instantaneous Description; Variants of Turing Machine - TM as Computer of Integer functions, Universal TM; Church's Thesis; Language acceptance by TM - Recursive and recursively enumerable languages;
- **Unit – 6 [8 Hours]:** Decidability - Halting problem, Introduction to Undecidability, Undecidable problems about TMs; Complexity - Time Complexity, Problem classes - P, NP, NP-Hard, NP-Complete.

Text Books

Text Books:

- John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman, Introduction to Automata Theory, Languages and Computation, Pearson Education, 3rd edition, 2014, ISBN: 978-0321455369
- Michael Sipser, Introduction to the Theory of Computation, Cengage Learning, 3rd Edition, 2014, ISBN: 978-8131525296

Reference Books:

- John C. Martin, Introduction to Languages and the Theory of Computation, McGraw-Hill Education, 4th edition, 2010, ISBN: 978-0073191461
- Bernard M. Moret, The Theory of Computation, Pearson Education, 2002, ISBN: 978-8131708705

Evaluation

- Quiz: Best (n-1) 20 marks
- Mid1: 20 marks
- Mid2: 25 marks
- Endsem: 35 marks
- Can be updated (and will be informed).

Mathematics Review I

(Basic Terminology)

- Unlike other CS courses, this course is a MATH course...
- We will look at a lot of definitions, theorems and proofs
- This lecture: reviews basic math notation and terminology
 - Set, Sequence, Function, Graph, String...
- Also, common proof techniques
 - By construction, induction, contradiction

Set

- A **set** is a group of items
- One way to describe a set: list every item in the group inside { }
 - E.g., { 12, 24, 5 } is a set with three items
- When the items in the set has trend: use ...
 - E.g., { 1, 2, 3, 4, ... } means the set of natural numbers
- Or, state the rule
 - E.g., { $n \mid n = m^2$ for some positive integer m } means the set { 1, 4, 9, 16, 25, ... }
- A set with no items is an **empty set** denoted by { } or \emptyset

Set

- The order of describing a set does not matter
 - $\{ 12, 24, 5 \} = \{ 5, 24, 12 \}$
- Repetition of items does not matter too
 - $\{ 5, 5, 5, 1 \} = \{ 1, 5 \}$
- Membership symbol \in
 - $5 \in \{ 12, 24, 5 \}$ $7 \notin \{ 12, 24, 5 \}$

- How many items are in each of the following set?
 - { 3, 4, 5, ..., 10 }
 - { 2, 3, 3, 4, 4, 2, 1 }
 - { 2, {2}, {{1,2,3,4,5,6}} }
 - \emptyset
 - { \emptyset }

Set

Given two sets A and B

- we say $A \subseteq B$ (read as A is a **subset** of B) if every item in A also appears in B
 - E.g., A = the set of primes, B = the set of integers
- we say $A \subsetneq B$ (read as A is a **proper subset** of B) if $A \subseteq B$ but $A \neq B$

Warning: Don't be confused with \in and \subseteq

- Let $A = \{1, 2, 3\}$. Is $\emptyset \in A$? Is $\emptyset \subseteq A$?

Union, Intersection, Complement

Given two sets A and B

- $A \cup B$ (read as the **union** of A and B) is the set obtained by combining all elements of A and B in a single set
 - E.g., $A = \{1, 2, 4\}$ $B = \{2, 5\}$
 $A \cup B = \{1, 2, 4, 5\}$
- $A \cap B$ (read as the **intersection** of A and B) is the set of common items of A and B
 - In the above example, $A \cap B = \{2\}$
- \bar{A} (read as the **complement** of A) is the set of items under consideration not in A

Set

- The **power set** of A is the set of all subsets of A , denoted by 2^A
 - E.g., $A = \{0, 1\}$
 $2^A = \{\{\}, \{0\}, \{1\}, \{0, 1\}\}$
 - How many items in the above power set of A ?
- If A has n items, how many items does its power set contain? Why?

Sequence

- A **sequence** of items is a list of these items in some order
- One way to describe a sequence: list the items inside ()
 - (5, 12, 24)
- Order of items inside () matters
 - (5, 12, 24) \neq (12, 5, 24)
- Repetition also matters
 - (5, 12, 24) \neq (5, 12, 12, 24)
- Finite sequences are also called **tuples**
 - (5, 12, 24) is a 3-tuple
 - (5, 12, 12, 24) is a 4-tuple

Sequence

Given two sets A and B

- The **Cartesian product** of A and B, denoted by $A \times B$, is the set of all possible 2-tuples with the first item from A and the second item from B
 - E.g., $A = \{1, 2\}$ and $B = \{x, y, z\}$
 $A \times B = \{(1,x), (1,y), (1,z), (2,x), (2,y), (2,z)\}$
- The Cartesian product of k sets, A_1, A_2, \dots, A_k , denoted by $A_1 \times A_2 \times \dots \times A_k$, is the set of all possible k-tuples with the i^{th} item from A_i

Functions

- A function takes an input and produces an output
- If f is a function, which gives an output b when input is a , we write
$$f(a) = b$$
- For a particular function f , the set of all possible input is called f 's domain
- The outputs of a function come from a set called f 's range

Functions

- To describe the property of a function that it has domain D and range R, we write

$$f : D \rightarrow R$$

- E.g., The function add (to add two numbers) will have an input of two integers, and output of an integer
 - We write: $\text{add} : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$

Strings

- An **alphabet** = a set of characters
 - E.g., The English Alphabet = {A,B,C,...,Z}
- A **string** = a sequence of characters
- A string over an alphabet Σ
 - A sequence of characters, with each character coming from Σ
- The **length** of a string w , denoted by $|w|$, is the number of characters in w
- The **empty string** (written as ε) is a string of length 0

Strings

Let $w = w_1w_2\dots w_n$ be a string of length n

- A **substring** of w is a consecutive subsequence of w (that is, $w_iw_{i+1}\dots w_j$ for some $i \leq j$)
- The **reverse** of w , denoted by w^R , is the string $w_n\dots w_2 w_1$
- A set of strings is called a **language**

PROOF TECHNIQUES

We look at

- Proof by contradiction
- Proof by construction
- Proof by induction
-
- But we may use some other also..
- For eg., Proof by counter example to disprove a statement...

By Contradiction

- One common way to prove a theorem is to assume that the theorem is false, and then show that this assumption leads to an obviously false consequence (also called a contradiction)
- This type of reasoning is used frequently in everyday life, as shown in the following example

By Contradiction

- Jack sees Jill, who just comes in from outdoor
- Jill looks completely dry
- Jack knows that it is not raining
- Jack's proof:
 - If it *were* raining (the assumption that the statement is false), Jill will be wet.
 - The consequence is: "Jill is wet" AND "Jill is dry", which is obviously false
 - Therefore, it must not be raining

By Contradiction [Example 1]

- Let us define a number is **rational** if it can be expressed as p/q where p and q are integers; if it cannot, then the number is called **irrational**
- E.g.,
 - 0.5 is rational because $0.5 = 1/2$
 - 2.375 is rational because $2.375 = 2375 / 1000$

By Contradiction

- Theorem: $\sqrt{2}$ (the square-root of 2) is irrational.
- How to prove?
- First thing is ...

Assume that $\sqrt{2}$ is rational

By Contradiction

- Proof: Assume that $\sqrt{2}$ is rational. Then, it can be written as p/q for some positive integers p and q .
- In fact, we can further restrict that p and q does not have common factor.
 - If D is a common factor of p and q , we use $p' = p/D$ and $q' = q/D$ so that $p'/q' = p/q = \sqrt{2}$ and there is no common factor between p' and q'
- Then, we have $p^2/q^2 = 2$, or $2q^2 = p^2$.

By Contradiction

- Since $2q^2$ is an even number, p^2 is also an even number
 - This implies that p is an even number (why?)
- So, $p = 2r$ for some integer r
- $2q^2 = p^2 = (2r)^2 = 4r^2$
 - This implies $2r^2 = q^2$
- So, q is an even number
- Something wrong happens... (what is it?)

By Contradiction

- We now have: “ p and q does not have common factor” AND “ p and q have common factor”
 - This is a contradiction
- Thus, the assumption is wrong, so that $\sqrt{2}$ is irrational

By Contradiction [Example 2]

- Theorem (Pigeonhole principle): A total of $n+1$ balls are put into n boxes. At least one box containing 2 or more balls.
 - Proof: Assume "at least one box containing 2 or more balls" is false
 - That is, each has at most 1 or fewer ball
- Consequence: total number of balls $\leq n$
- Thus, there is a contradiction (what is that?)

Proof By Construction

- Many theorem states that a particular type of object exists
- One way to prove is to find a way to construct one such object
- This technique is called **proof by construction**

- Theorem: There exists a rational number p which can be expressed as q^r , with q and r both irrational.
- How to prove?
 - Find p, q, r satisfying the above condition
- What is the irrational number we just learnt? Can we make use of it?

By Construction

- What is the following value?
 $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}}$
- If $\sqrt{2}^{\sqrt{2}}$ is rational, then $q = r = \sqrt{2}$ gives the desired answer
- Otherwise, $q = \sqrt{2}^{\sqrt{2}}$ and $r = \sqrt{2}$ gives the desired answer

By Induction

- Normally used to show that all elements in an infinite set have a specified property
- The proof consists of proving two things: The **basis**, and the **inductive step**

- Mathematical induction proves that we can climb as high as we like on a ladder, by proving that we can climb onto the bottom rung (**the basis**) and that from each rung we can climb up to the next one (**the inductive step**).

We consider only enumerable or countable sets
with a least element [well ordered sets]

1. The **base case**: prove that the statement holds for the first natural number n . Usually, $n = 0$ or $n = 1$;
 - rarely, but sometimes conveniently, the base value of n may be taken as a larger number, or even as a negative number (the statement only holds at and above that threshold).
2. The **step case or inductive step**: assume the statement holds for some natural number n , and prove that then the statement holds for $n + 1$.

By Induction [Example 1]

- Let $F(k)$ be a sequence defined as follows:
- $F(1) = 1$
- $F(2) = 1$
- for all $k \geq 3$, $F(k) = F(k-1) + F(k-2)$
- Theorem: For all $n \geq 1$,
$$F(1) + F(2) + \dots + F(n) = F(n+2) - 1$$

By Induction

- Let $P(k)$ means "the theorem is true when $n = k$ "
- Basis: To show $P(1)$ is true.
 - $F(1) = 1$, $F(3) = F(1) + F(2) = 2$
 - Thus, $F(1) = F(3) - 1$
 - Thus, $P(1)$ is true
- Inductive Step: To show for $k \geq 1$, $P(k) \rightarrow P(k+1)$
 - $P(k)$ is true means: $F(1) + F(2) + \dots + F(k) = F(k+2) - 1$
 - Then, we have
$$\begin{aligned} & F(1) + F(2) + \dots + F(k+1) \\ &= (F(k+2) - 1) + F(k+1) \\ &= F(k+3) - 1 \end{aligned}$$
 - Thus, $P(k+1)$ is true if $P(k)$ is true

Variants

- There can be many other types of basis and inductive step, as long as by proving both of them, they can cover all the cases
- For example, to show P is true for all $k > 1$, we can show
 - Basis: $P(1)$ is true, $P(2)$ is true
 - Inductive step: $P(k) \rightarrow P(k+2)$

Variants

- **Complete (strong) induction:** (in contrast to which the basic form of induction is sometimes known as **weak induction**)
makes the inductive step easier to prove by using a stronger hypothesis: one proves the statement $P(m + 1)$ under the assumption that $P(n)$ holds for all $n, n \leq m$.

Example: forming dollar amounts by coins

- Assume an infinite supply of 4 and 5 dollar coins.
- Prove that any whole amount of dollars greater than 12 can be formed by a combination of such coins.
- In more precise terms, we wish to show that for any amount $n \geq 12$ there exist natural numbers a and b such that $n = 4a + 5b$, where 0 is included as a natural number.
- The statement to be shown true is thus:

$$S(n) : n \geq 12 \Rightarrow \exists a, b \in \mathbb{N}. n = 4a + 5b$$

Base case: Show that $S(k)$ holds for $k = 12, 13, 14, 15$.

$$4 \cdot 3 + 5 \cdot 0 = 12$$

$$4 \cdot 2 + 5 \cdot 1 = 13$$

$$4 \cdot 1 + 5 \cdot 2 = 14$$

$$4 \cdot 0 + 5 \cdot 3 = 15$$

The base case holds.

Induction step:

For $j = 12, 13, \dots, 15, \dots, k$ we assume that the theorem is true.

For $j = k + 1$, we show that the theorem is true.

Since for $j = k, k - 1, k - 2, k - 3$ the theorem is true (why?).

So, $k - 3 = 4a + 5b$, for some nonnegative integers a and b .

Since $k + 1 = (k - 3) + 4$,

we have, $k + 1 = 4a + 5b + 4 = 4(a + 1) + 5b$. Q.E.D.

- The following is not a valid proof by induction!

By Induction?

- CLAIM: In any set of h horses, all horses are of the same color.
- PROOF: By induction. Let $P(k)$ means "the claim is true when $h = k$ "
- Basis: $P(1)$ is true, because in any set of 1 horse, all horses clearly are the same color.

By Induction?

- Inductive step:
 - Assume $P(k)$ is true.
 - Then we take any set of $k+1$ horses.
 - Remove one of them. Then, the remaining horses are of the same color (because $P(k)$ is true).
 - Put back the removed horse into the set, and remove another horse
 - In this new set, all horses are of same color (because $P(k)$ is true).
 - Therefore, all horses are of the same color!
- What's wrong?

More on Pigeonhole Principle

- Theorem: For any graph with more than two vertices, there exists two vertices whose degree are the same.
- How to prove?

For connected graphs

First, suppose that G is a connected finite simple graph with n vertices. Then every vertex in G has degree between 1 and $n - 1$ (the degree of a given vertex cannot be zero since G is connected, and is at most $n - 1$ since G is simple). Since there are n vertices in G with degree between 1 and $n - 1$, the pigeon hole principle lets us conclude that there is some integer k between 1 and $n - 1$ such that two or more vertices have degree k .

For arbitrary graphs

Now, suppose G is an arbitrary finite simple graph (not necessarily connected). If G has any connected component consisting of two or more vertices, the above argument shows that that component contains two vertices with the same degree, and therefore G does as well. On the other hand, if G has no connected components with more than one vertex, then every vertex in G has degree zero, and so there are multiple vertices in G with the same degree. \square

- As we go, we see various proofs.
- Proofs has to be formal.
- You can not scribble something and expect the examiner to interpret the answer !!

- As we go, we will
 - Proofs have
 - You can not expect the examiner to
- and expect the
!!





Your TA should not become like this !

General Advise

- Use standard vocabulary, simple notation, and techniques described in the class.
- Most numeric questions are toy problems, whose answer can be obtained by common sense (often simple trial and error method is enough).
- You **have to** use the method(s) described in the **classwork** or in the **text books**. Otherwise you may not get marks.

Deterministic Finite Automaton and Non-deterministic Finite Automaton

DFA and NFA
(Finite State Machines)

Notation and Definitions

- Alphabet
- String
- Language
- Operations on languages

Strings

- An **alphabet** is any **finite set of distinct symbols**
 - $\{0, 1\}$, $\{0, 1, 2, \dots, 9\}$, $\{a, b, c\}$
 - We denote a generic alphabet by Σ
- A **string** is any **finite-length sequence** of elements of Σ .
- e.g., if $\Sigma = \{a, b\}$ then a , aba , $aaaa$,,
 $abababbaab$ are some strings over the alphabet Σ

Strings

- The **length** of a string ω is the number of symbols in ω . We denote it by $|\omega|$. $|aba| = 3$.
- The symbol ϵ denotes a special string called the **empty string**
 - ϵ has length 0
- String concatenation
 - If $\omega = a_1, \dots, a_n$ and $\nu = b_1, \dots, b_m$ then $\omega \cdot \nu$ (or $\omega\nu$)
 $= a_1, \dots, a_n b_1, \dots, b_m$
 - Concatenation is associative with ϵ as the identity element.
- If $a \in \Sigma$, we use a^n to denote a string of n a 's concatenated
 - $\Sigma = \{0, 1\}$, $0^5 = 00000$
 - $a^0 =_{def} \epsilon$
 - $a^{n+1} =_{def} a^n a$

Strings

- The **reverse** of a string ω is denoted by ω^R .
 - $\omega^R = a_n, \dots, a_1$
- A **substring** y of a string ω is a string such that $\omega = xyz$ with $|x|, |y|, |z| \geq 0$ and $|x| + |y| + |z| = |\omega|$
- If $\omega = xy$ with $|x|, |y| \geq 0$ and $|x| + |y| = |\omega|$, then x is **prefix** of ω and y is a **suffix** of ω .
 - For $\omega = abaab$,
 - ϵ, a, aba , and $abaab$ are some prefixes
 - $\epsilon, abaab, aab$, and $baab$ are some suffixes.

Strings

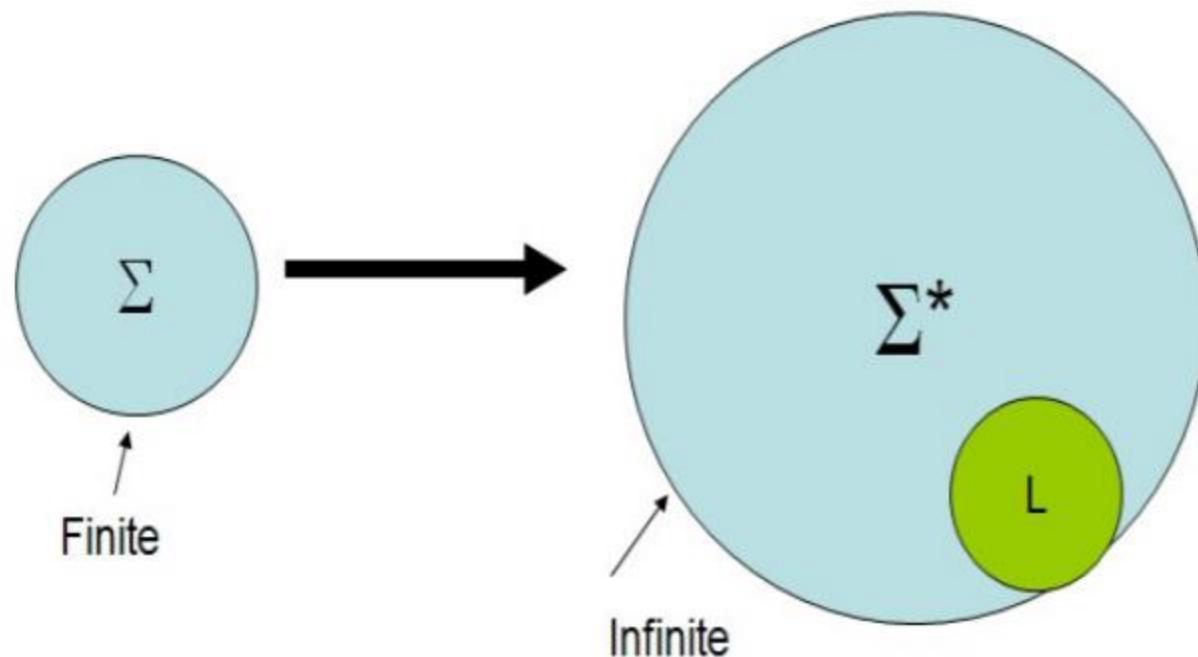
- The set of all possible strings over Σ is denoted by Σ^* .
- We define $\Sigma^0 = \{\epsilon\}$ and $\Sigma^n = \Sigma^{n-1} \cdot \Sigma$
 - with some abuse of the concatenation notation applying to sets of strings now
- So $\Sigma^n = \{\omega | \omega = xy \text{ and } x \in \Sigma^{n-1} \text{ and } y \in \Sigma\}$
- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots \Sigma^n \cup \dots = \bigcup_0^\infty \Sigma^i$
 - Alternatively, $\Sigma^* = \{x_1 x_2 \dots x_n | n \geq 0 \text{ and } x_i \in \Sigma \text{ for all } i\}$
- Φ denotes the empty set of strings $\Phi = \{\},$
 - but $\Phi^* = \{\epsilon\}$

Strings

- Σ^* is a **countably infinite set of finite length strings**
- If x is a string, we write x^n for the string obtained by concatenating n copies of x .
 - $(aab)^3 = aabaabaab$
 - $(aab)^0 = \epsilon$

Languages

- A **language** L over Σ is any subset of Σ^*



- L can be finite or (countably) infinite

Some Languages

- $L = \Sigma^*$ – The mother of all languages!
- $L = \{a, ab, aab\}$ – A fine finite language.
 - Description by enumeration
- $L = \{a^n b^n : n \geq 0\} = \{\epsilon, ab, aabb, aaabbb, \dots\}$
- $L = \{\omega | n_a(\omega) \text{ is even}\}$
 - $n_x(\omega)$ denotes the number of occurrences of x in ω
 - all strings with even number of a's.
- $L = \{\omega | \omega = \omega^R\}$
 - All strings which are the same as their reverses – palindromes.
- $L = \{\omega | \omega = xx\}$
 - All strings formed by duplicating some string once.
- $L = \{\omega | \omega \text{ is a syntactically correct Java program}\}$

Languages

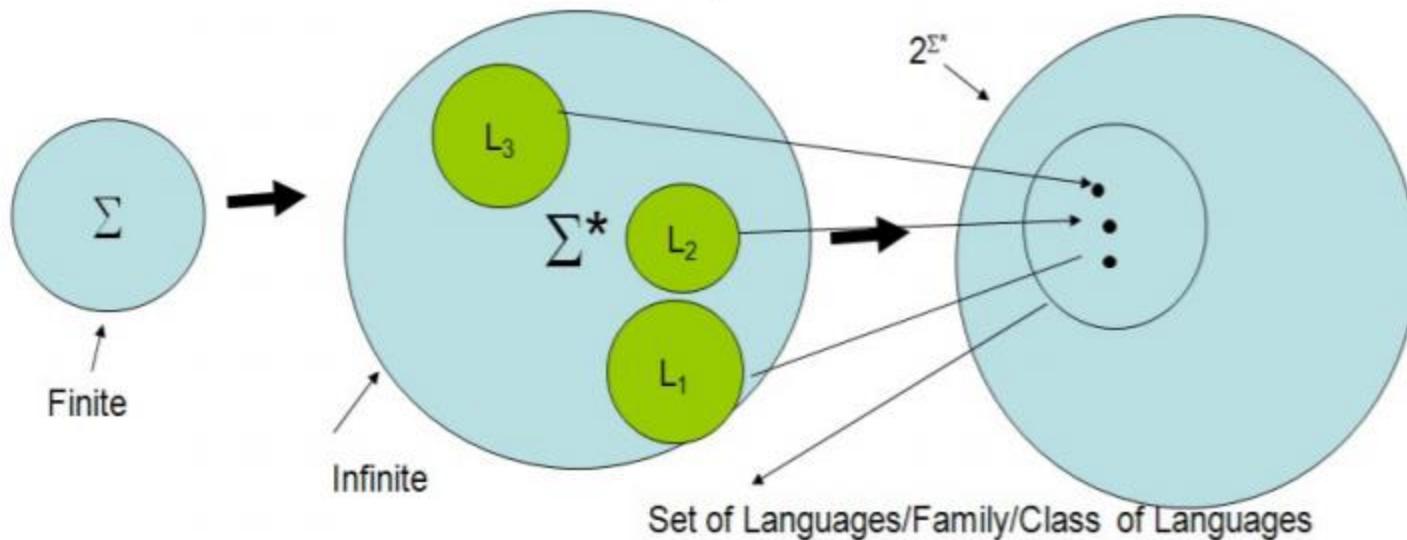
- Since languages are sets, all usual set operations such as intersection and union, etc. are defined.
- Complementation is defined with respect to the universe Σ^* : $\bar{L} = \Sigma^* - L$

Languages

- If L , L_1 and L_2 are languages:
 - $L_1 \cdot L_2 = \{xy | x \in L_1 \text{ and } y \in L_2\}$
 - $L^0 = \{\epsilon\}$ and $L^n = L^{n-1} \cdot L$
 - $L^* = \bigcup_0^{\infty} L^i$
 - $L^+ = \bigcup_1^{\infty} L^i$

Sets of Languages

- The power set of Σ^* , the set of all its subsets, is denoted as 2^{Σ^*}



AUTOMATA

- The control unit has some **finite memory** and it **keeps track of what step to execute next.**
- Additional memory (if any) is infinite - we never run out of memory!
 - Infinite but like a stack - **only the top item is accessible at a given time.**
 - Infinite but like a tape, any cell is (sequentially) accessible.

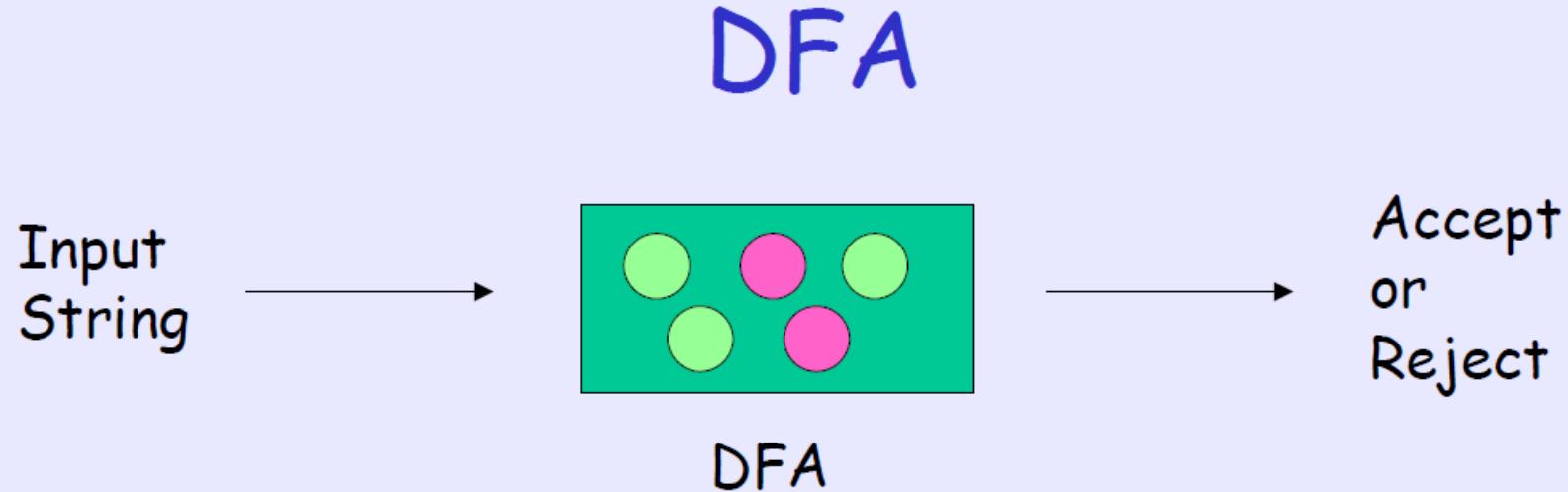
FINITE STATE AUTOMATA

- Finite State Automata (FSA) are the simplest automata.
- Only the **finite** memory in the control unit is available.
- The memory can be in one of finite **states** at a given time – hence the name.
 - One can remember only a (fixed) finite number of properties of the past input.
 - Since input strings can be of arbitrary length, **it is not possible to remember unbounded portions of the input string.**
- It comes in **Deterministic** and **Nondeterministic** flavors.

DETERMINISTIC FINITE STATE AUTOMATA (DFA)

- A DFA starts in a **start state** and is presented with an input string.
- It **moves from state to state**, reading the input string one symbol at a time.
- What state the DFA moves next depends on
 - the current state,
 - current input symbol
- **When the last input symbol is read**, the DFA decides whether it should accept the input string

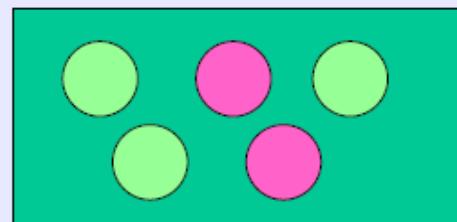
Finite State Machines



- A machine with finite number of **states**, some states are **accepting states**, others are **rejecting states**
- At any time, it is in one of the states
- It reads an input string, one character at a time

DFA

Input String

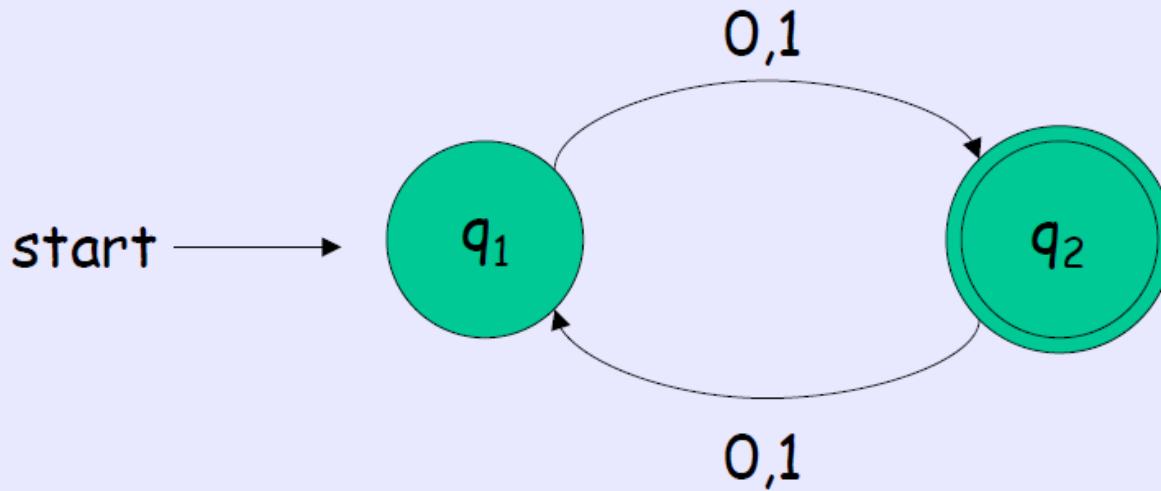


Accept
or
Reject

DFA

- After reading each character, it moves to another state depending on **what is read** and **what is the current state**
- If reading all characters, the DFA is in an accepting state, the input string is **accepted**.
- Otherwise, the input string is **rejected**.

Example of DFA



- The circles indicates the states
- If **accepting** state is marked with double circle
- The arrows pointing from a state q indicates how to move on reading a character when current state is q

DFA – FORMAL DEFINITION

- A Deterministic Finite State Acceptor (DFA) is defined as the 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where
 - Q is a finite set of states
 - Σ is a finite set of symbols – the alphabet
 - $\delta : Q \times \Sigma \rightarrow Q$ is the next-state function
 - $q_0 \in Q$ is the (label of the) start state
 - $F \subseteq Q$ is the set of final (accepting) states

DFA – FORMAL DEFINITION

- A Deterministic Finite State Acceptor (DFA) is defined as the 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where
 - Q is a finite set of states
 - Σ is a finite set of symbols – the alphabet
 - $\delta : Q \times \Sigma \rightarrow Q$ is the next-state function
 - $q_0 \in Q$ is the (label of the) start state
 - $F \subseteq Q$ is the set of final (accepting) states

Note, there must be exactly one start state.
Final states can be many or even empty !

Some Terminology

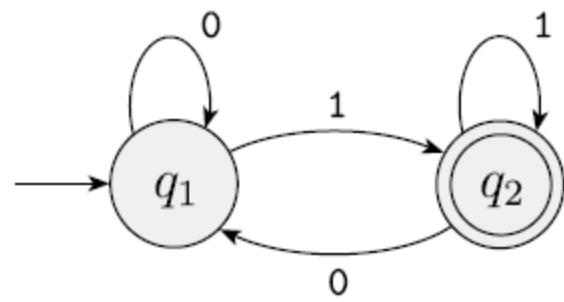
Let M be a DFA

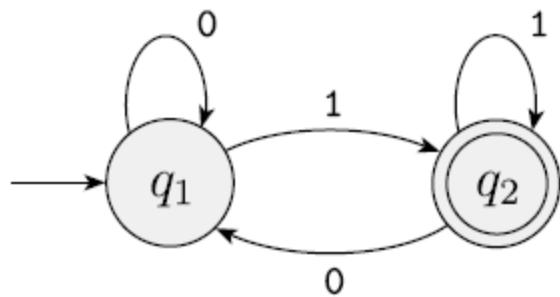
- Among all possible strings, M will accept some of them, and M will reject the remaining
- The set of strings which M accepts is called the language **recognized** by M
- That is, M **recognizes** A if

$$A = \{ w \mid M \text{ accepts } w \}$$

$$L(M)$$

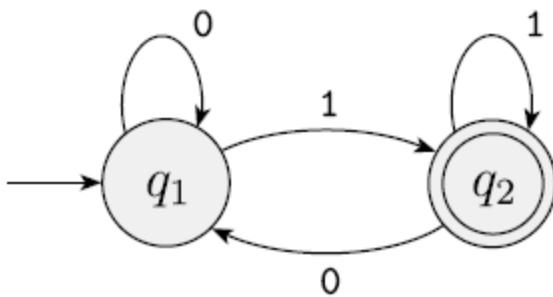
If A is the set of all strings that machine M accepts, we say that A is the *language of machine M* and write $L(M) = A$. We say that M *recognizes* A or that M *accepts* A .





In the formal description, M_2 is $(\{q_1, q_2\}, \{0,1\}, \delta, q_1, \{q_2\})$. The transition function δ is

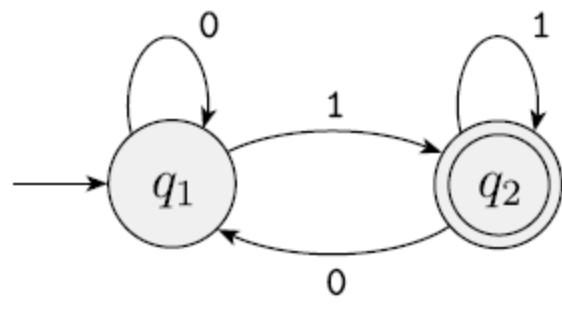
	0	1
q_1	q_1	q_2
q_2	q_1	q_2 .



In the formal description, M_2 is $(\{q_1, q_2\}, \{0,1\}, \delta, q_1, \{q_2\})$. The transition function δ is

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

Remember that the state diagram of M_2 and the formal description of M_2 contain the same information, only in different forms. You can always go from one to the other if necessary.



M_2

$L(M_2) = \{w \mid w \text{ ends in a } 1\}.$

Consider the finite automaton M_3 .

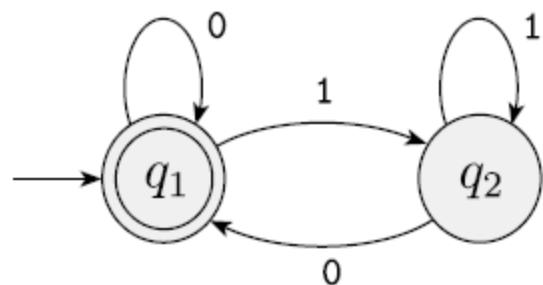


FIGURE 1.10

State diagram of the two-state finite automaton M_3

Can you describe this in the 5 tuple form?

In particular, can you write down the transition table?

Consider the finite automaton M_3 .

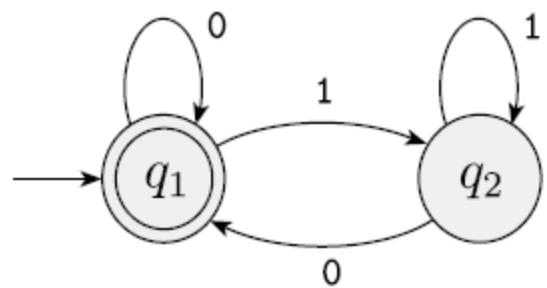


FIGURE 1.10

State diagram of the two-state finite automaton M_3

What language M_3 recognizes?

Consider the finite automaton M_3 .

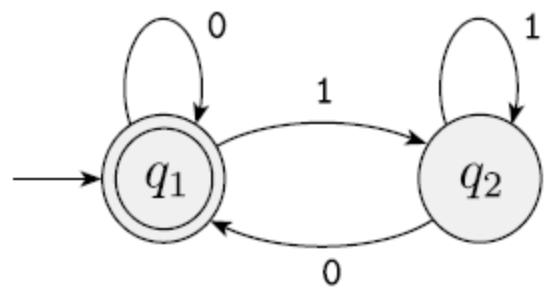


FIGURE 1.10

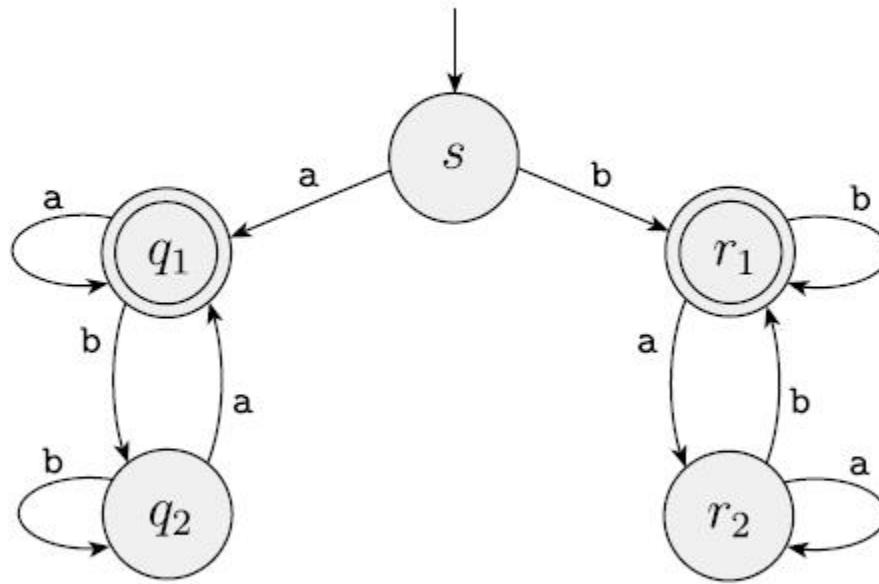
State diagram of the two-state finite automaton M_3

What language M_3 recognizes?

$$L(M_3) = \{w \mid w \text{ is the empty string } \varepsilon \text{ or ends in a } 0\}.$$

EXAMPLE 1.11

The following figure shows a five-state machine M_4 .

**FIGURE 1.12**

Finite automaton M_4

EXAMPLE 1.11

The following figure shows a five-state machine M_4 .

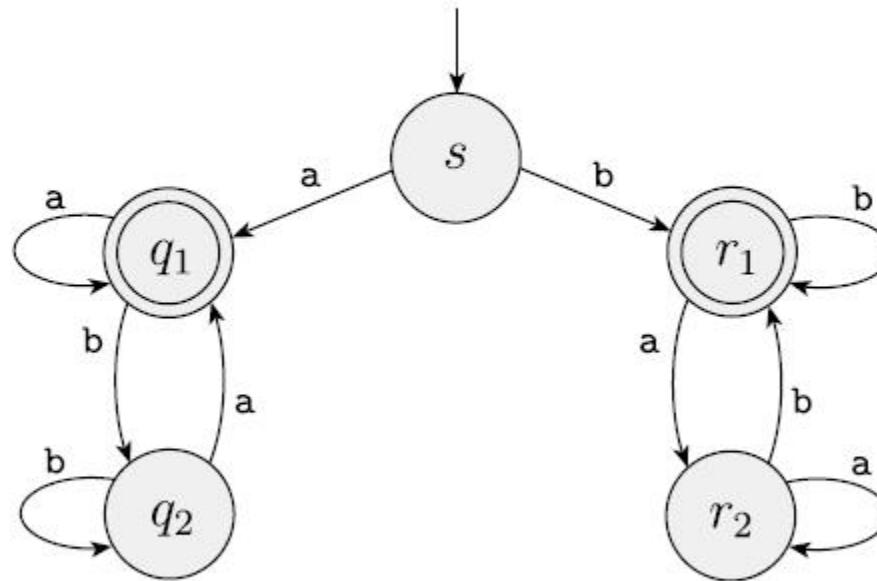


FIGURE 1.12

Finite automaton M_4

$L(M_4)$ = all strings that begin and end with the same character.

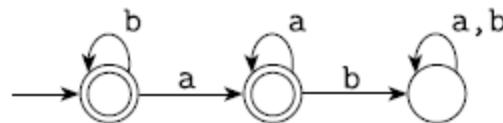
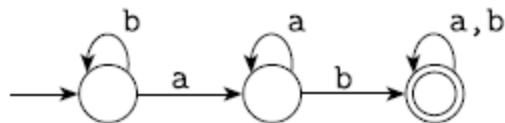
DFA for complement of a language

- Flip final and non-final states.

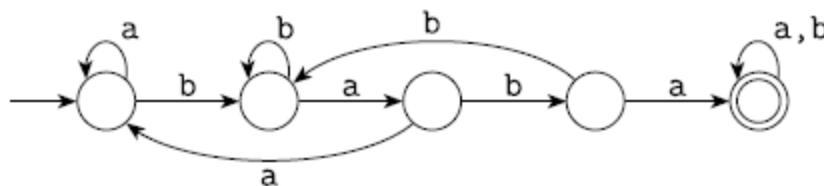
1.5 Each of the following languages is the complement of a simpler language. In each part, construct a DFA for the simpler language, then use it to give the state diagram of a DFA for the language given. In all parts, $\Sigma = \{a, b\}$.

- ^Aa. $\{w \mid w \text{ does not contain the substring } ab\}$
- ^Ab. $\{w \mid w \text{ does not contain the substring } baba\}$

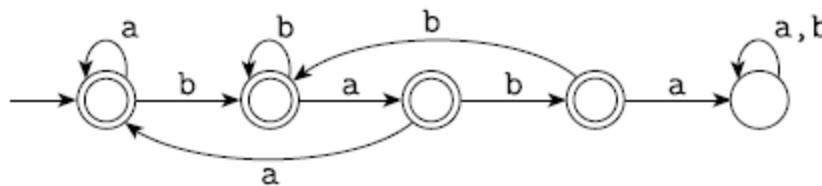
- 1.5 (a) The left-hand DFA recognizes $\{w \mid w \text{ contains } ab\}$. The right-hand DFA recognizes its complement, $\{w \mid w \text{ doesn't contain } ab\}$.



- (b) This DFA recognizes $\{w \mid w \text{ contains } baba\}$.



This DFA recognizes $\{w \mid w \text{ does not contain } baba\}$.



Designing a DFA (Quick Quiz)

- How to design a DFA that accepts all binary strings representing a multiple of 5? (E.g., 101, 1111, 11001, ...)

Formally

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where each w_i is a member of the alphabet Σ . Then M *accepts* w if a sequence of states r_0, r_1, \dots, r_n in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \dots, n - 1$, and
3. $r_n \in F$.

Regular language [Ref: Sipser Book]

DEFINITION 1.16

A language is called a *regular language* if some finite automaton recognizes it.

The regular operations

DEFINITION 1.23

Let A and B be languages. We define the regular operations *union*, *concatenation*, and *star* as follows:

- **Union:** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
- **Concatenation:** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$.
- **Star:** $A^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$.

- These are similar to arithmetic operations.
- Note, $*$ is a unary operator.

THEOREM 1.25

The class of regular languages is closed under the union operation.

In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

- The proof is by construction.
- We build a DFA for the union from the individual DFAs.
- The idea is simple: While reading the input simultaneously follow both machines.
 - Put a finger on current state. You need two fingers. You can move these two fingers as per the respective transition function.

PROOF

Let M_1 recognize A_1 , where $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, and
 M_2 recognize A_2 , where $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.

Construct M to recognize $A_1 \cup A_2$, where $M = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$.

This set is the **Cartesian product** of sets Q_1 and Q_2 and is written $Q_1 \times Q_2$.
It is the set of all pairs of states, the first from Q_1 and the second from Q_2 .

2. Σ , the alphabet, is the same as in M_1 and M_2 . In this theorem and in all subsequent similar theorems, we assume for simplicity that both M_1 and M_2 have the same input alphabet Σ . The theorem remains true if they have different alphabets, Σ_1 and Σ_2 . We would then modify the proof to let $\Sigma = \Sigma_1 \cup \Sigma_2$.

3. δ , the transition function, is defined as follows. For each $(r_1, r_2) \in Q$ and each $a \in \Sigma$, let

$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)).$$

Hence δ gets a state of M (which actually is a pair of states from M_1 and M_2), together with an input symbol, and returns M 's next state.

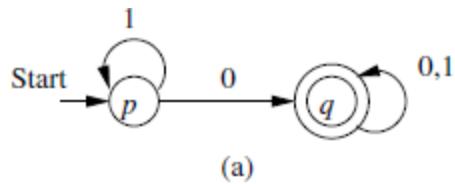
4. q_0 is the pair (q_1, q_2) .

5. F is the set of pairs in which either member is an accept state of M_1 or M_2 .
We can write it as

$$F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\}.$$

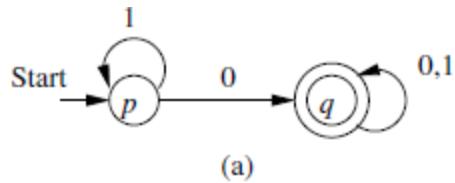
This expression is the same as $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$. (Note that it is *not* the same as $F = F_1 \times F_2$. What would that give us instead?³)

Union Example

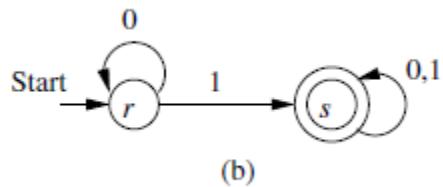


What is the language recognized by this DFA?

Union Example



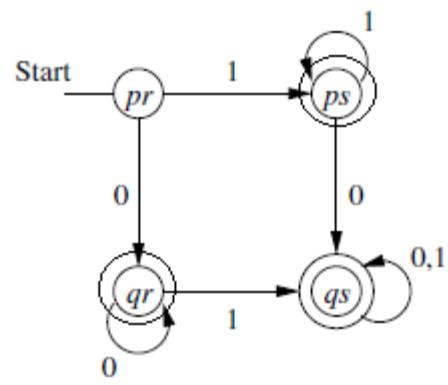
What is the language recognized by this DFA?



What is the language recognized by this DFA?

Find DFA for the union

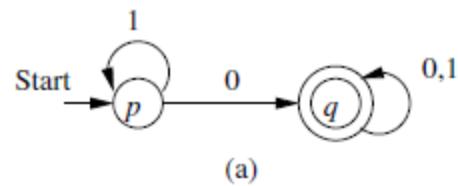
Find DFA for the union



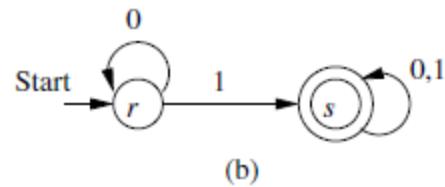
What about intersection?

- Intersection of two regular languages is also regular.
- Proof: by construction. Similar. Only final states will change.

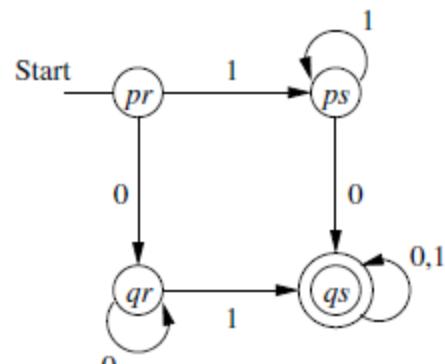
Intersection



(a)



(b)



(c)

What else we can do with product principle?

- Set difference.
 - How?

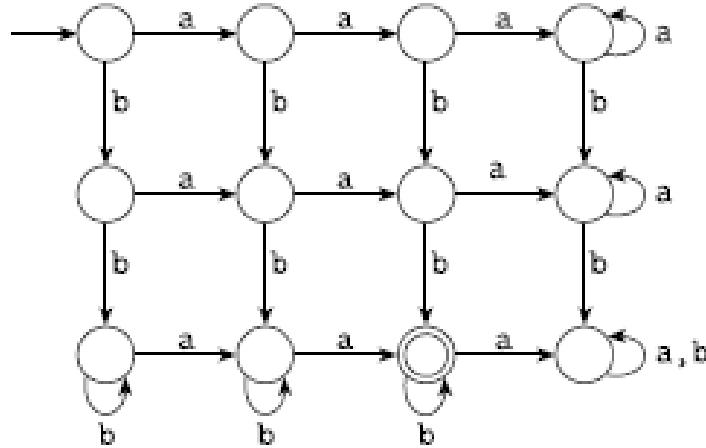
- 1.4 Each of the following languages is the intersection of two simpler languages. In each part, construct DFAs for the simpler languages, then combine them using the construction discussed in footnote 3 (page 46) to give the state diagram of a DFA for the language given. In all parts, $\Sigma = \{a, b\}$.
- a. $\{w \mid w \text{ has at least three } a\text{'s and at least two } b\text{'s}\}$
 - ^Ab. $\{w \mid w \text{ has exactly two } a\text{'s and at least two } b\text{'s}\}$
 - c. $\{w \mid w \text{ has an even number of } a\text{'s and one or two } b\text{'s}\}$
 - ^Ad. $\{w \mid w \text{ has an even number of } a\text{'s and each } a \text{ is followed by at least one } b\}$
 - e. $\{w \mid w \text{ starts with an } a \text{ and has at most one } b\}$
 - f. $\{w \mid w \text{ has an odd number of } a\text{'s and ends with a } b\}$
 - g. $\{w \mid w \text{ has even length and an odd number of } a\text{'s}\}$

1.4 (b) The following are DFAs for the two languages $\{w \mid w \text{ has exactly two a's}\}$ and $\{w \mid w \text{ has at least two b's}\}$.



- Now find product machine.

Combining them using the intersection construction gives the following DFA.



- This can be minimized. {Some states are redundant}.

NONDETERMINISM

- Useful concept, has great impact on ToC/algorithms.
- DFA is deterministic: every step of a computation follows in a unique way from the preceding step.
 - When the machine is in a given state, and upon reading the next input symbol, we know deterministically what would be the next state.
 - Only one next state.
 - No choice !!

NONDETERMINISM

- In a nondeterministic machine, several choices may exist for the next state at any point.
- Nondeterminism is a generalization of determinism.

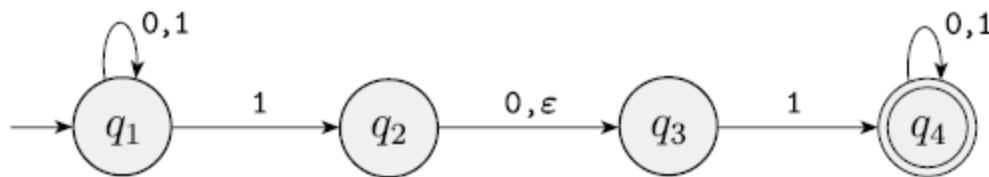


FIGURE 1.27

The nondeterministic finite automaton N_1

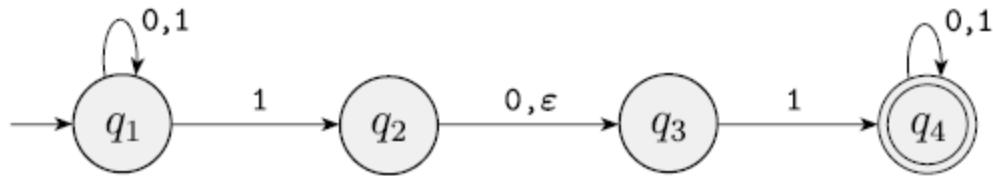


FIGURE 1.27

The nondeterministic finite automaton N_1

- More than one arrow from from q_1 on symbol 1.
- No arrow at all from q_3 on 0.
- There is ε over an arrow !

How does an NFA compute?

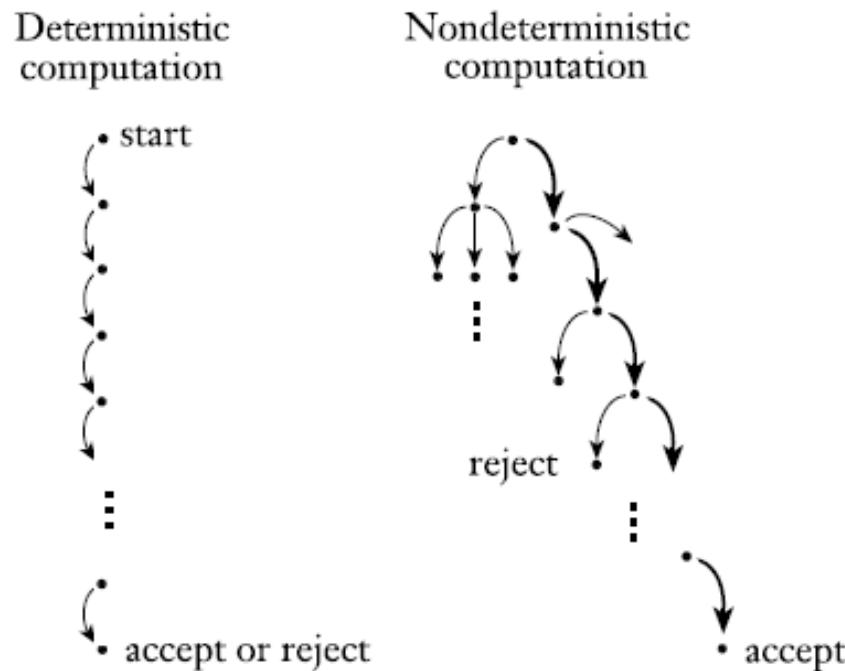


FIGURE 1.28

Deterministic and nondeterministic computations with an accepting branch

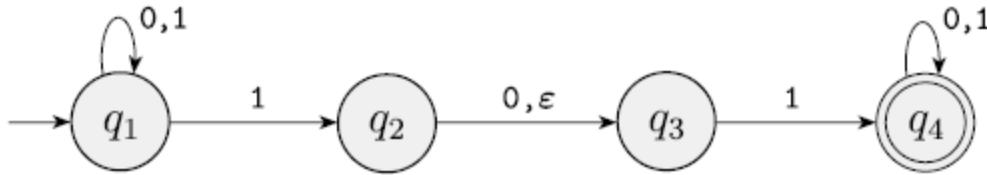


FIGURE 1.27

The nondeterministic finite automaton N_1

On input **010110**

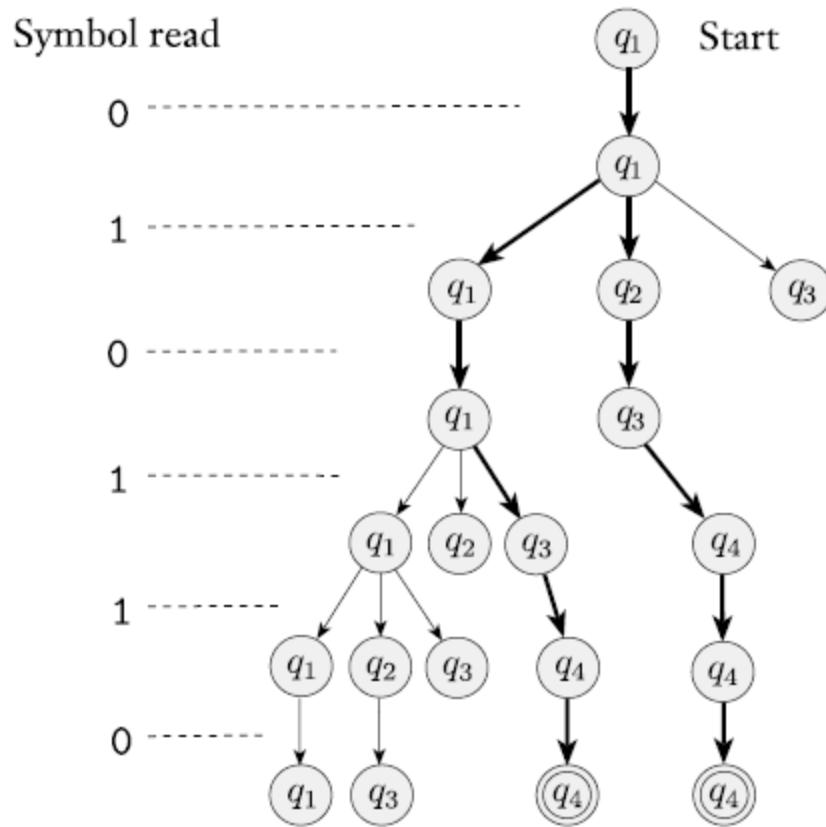


FIGURE 1.29

The computation of N_1 on input 010110

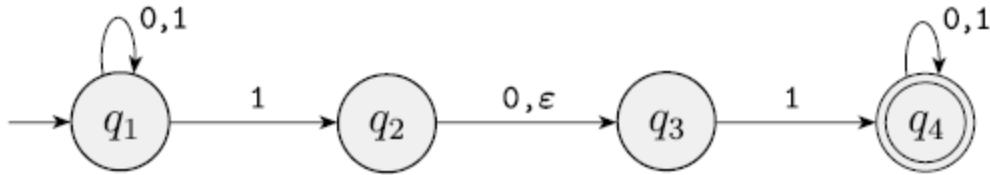


FIGURE 1.27

The nondeterministic finite automaton N_1

- What is the language accepted by this NFA?

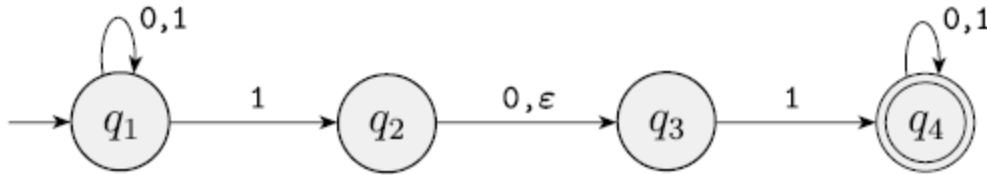


FIGURE 1.27

The nondeterministic finite automaton N_1

- It accepts all strings that contain either 101 or 11 as a substring.
- Constructing NFAs is sometimes easier than constructing DFAs.
 - Later we see that every NFA can be converted into an equivalent DFA.

EXAMPLE 1.30

Let A be the language consisting of all strings over $\{0,1\}$ containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not). The following four-state NFA N_2 recognizes A .

- Building DFA for this is possible, but difficult.
- Try this.

But NFA is easy to build.

EXAMPLE 1.30

Let A be the language consisting of all strings over $\{0,1\}$ containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not). The following four-state NFA N_2 recognizes A .

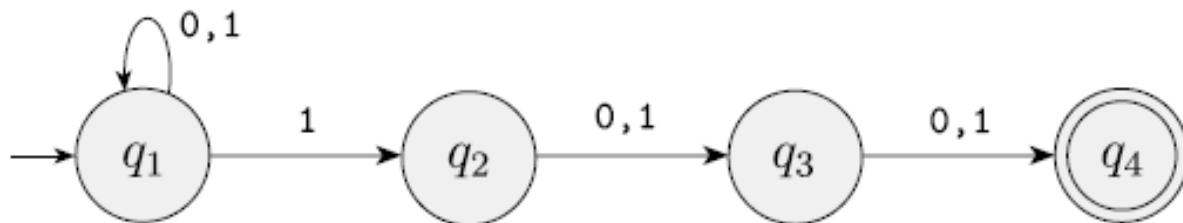


FIGURE 1.31

The NFA N_2 recognizing A

DFA for A

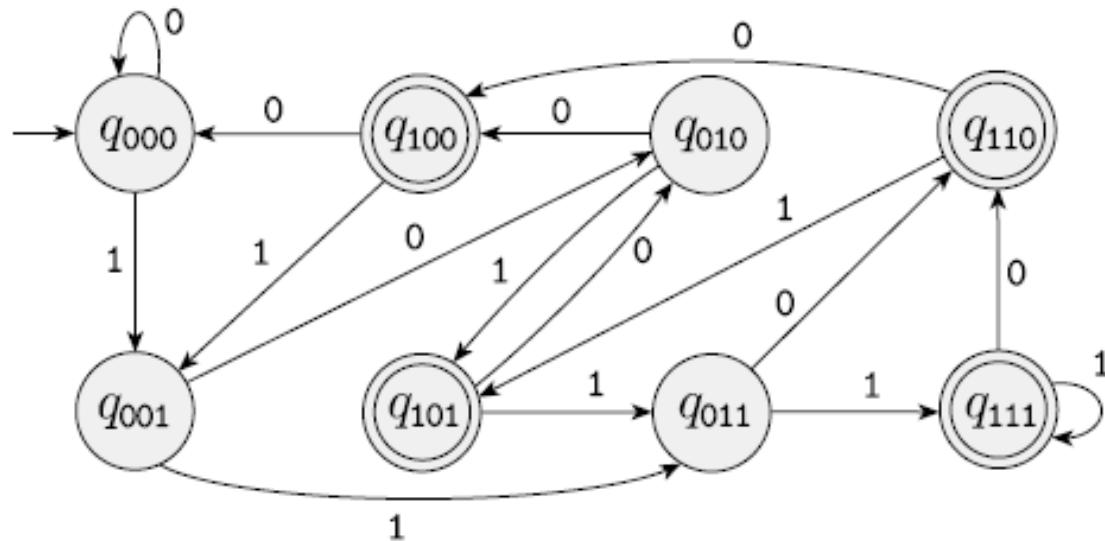


FIGURE 1.32
A DFA recognizing A

- See number of states and complexity !

Formal definition of NFA

We use Σ_ε to mean $\Sigma \cup \{\varepsilon\}$

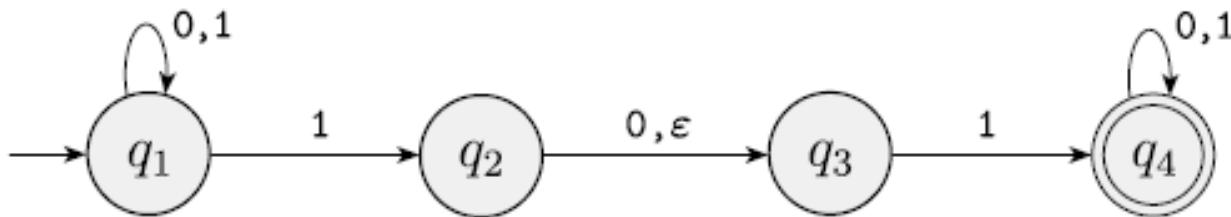
DEFINITION 1.37

A *nondeterministic finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta: Q \times \Sigma_\varepsilon \rightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

EXAMPLE 1.38

Recall the NFA N_1 :



The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0,1\}$,
3. δ is given as

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	\emptyset
q_2	$\{q_3\}$	\emptyset	$\{q_3\}$
q_3	\emptyset	$\{q_4\}$	\emptyset
q_4	$\{q_4\}$	$\{q_4\}$	\emptyset

4. q_1 is the start state, and
5. $F = \{q_4\}$.

The formal definition of computation for an NFA is similar to that for a DFA. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and w a string over the alphabet Σ . Then we say that N accepts w if we can write w as $w = y_1y_2 \cdots y_m$, where each y_i is a member of Σ_ε and a sequence of states r_0, r_1, \dots, r_m exists in Q with three conditions:

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, \dots, m - 1$, and
3. $r_m \in F$.

Equivalence of NFAs and DFAs

- We say two machines are equivalent if they recognize the same language.

THEOREM 1.39 -----

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

Proof

- Proof by construction.
 - We build an equal DFA for the given NFA

Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA recognizing some language A . We construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ recognizing A .

- First, for understanding purpose, we assume that there are no edges with ϵ transitions.

Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA recognizing some language A . We construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ recognizing A .

1. $Q' = \mathcal{P}(Q)$.

Every state of M is a set of states of N . Recall that $\mathcal{P}(Q)$ is the set of subsets of Q .

2. For $R \in Q'$ and $a \in \Sigma$, let $\delta'(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ for some } r \in R\}$.

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).$$

3. $q_0' = \{q_0\}$.

M starts in the state corresponding to the collection containing just the start state of N .

4. $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$.

The machine M accepts if one of the possible states that N could be in at this point is an accept state.

Can you convert the following

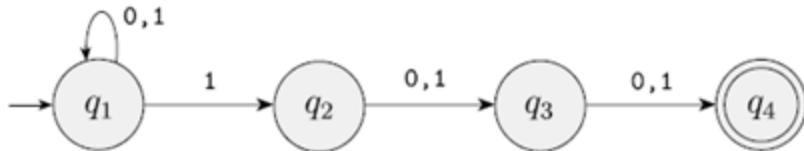


FIGURE 1.31
The NFA N_2

- What is the language accepted by this?

Now, considering ϵ arrows

- For this purpose, we define ϵ -CLOSURE of a set of states R .

Formally, for $R \subseteq Q$ let

$$E(R) = \{q \mid q \text{ can be reached from } R \text{ by traveling along 0 or more } \epsilon \text{ arrows}\}.$$

- $E(R)$ is ϵ -CLOSURE of R .

- Then the transition is defined as,

$$\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ for some } r \in R\}.$$

- Now the start state of the DFA should be

$$q_0' = E(\{q_0\})$$

Example

•

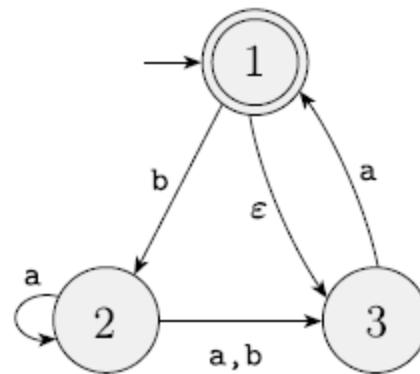
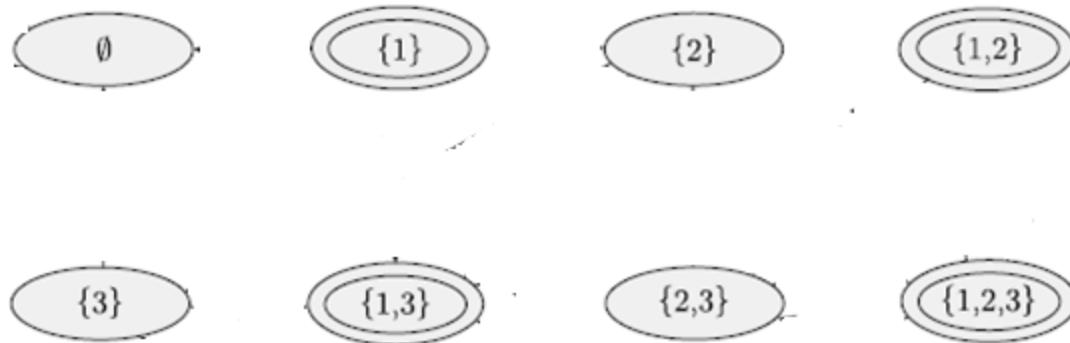
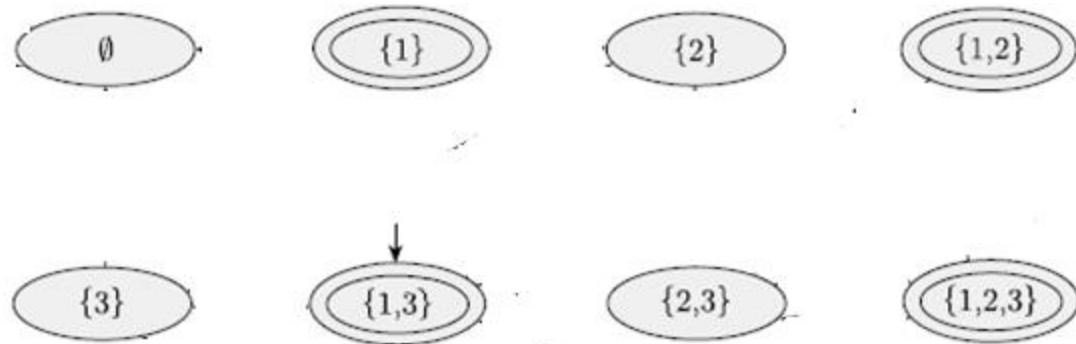


FIGURE 1.42
The NFA N_4

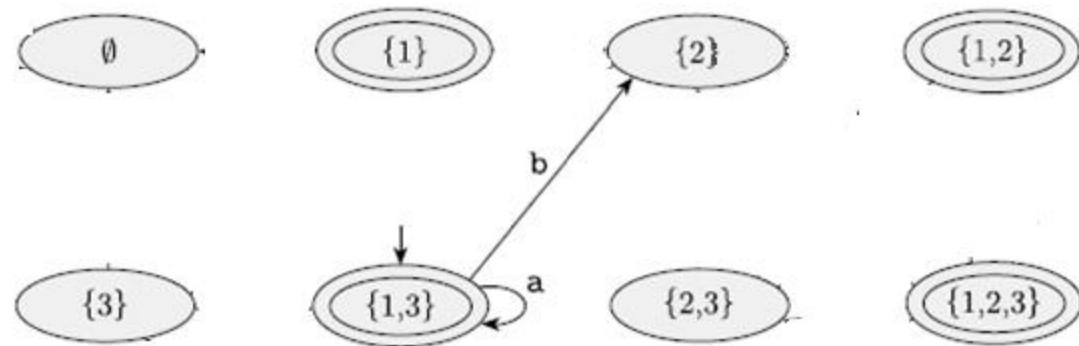


All possible states of the DFA.
(to be constructed; Final states are shown)

- Now we need to add edges, and
- identify the initial state.



- Identify the initial state.
 - Note, it is not $\{1\}$



- Adding edges, ...

After all edges ...

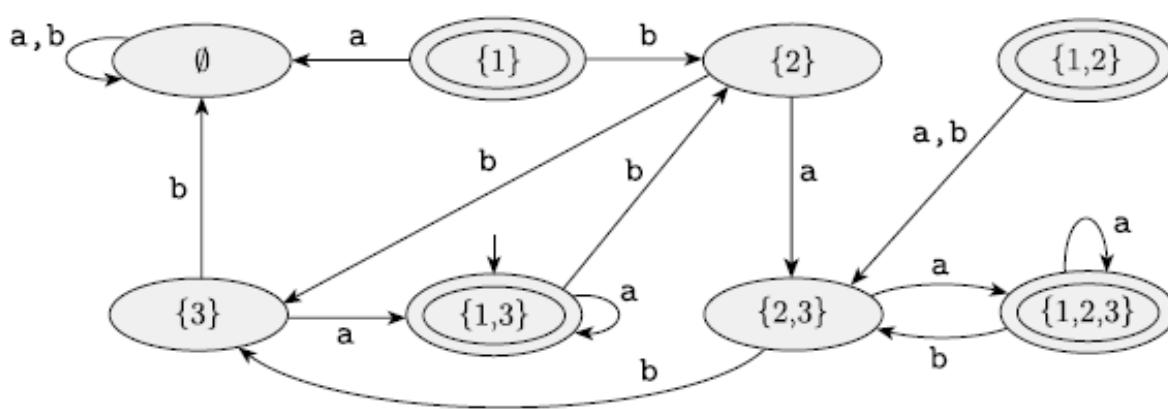


FIGURE 1.43
A DFA D that is equivalent to the NFA N_4

- But, some states are not reachable !
- Simplification can remove this.

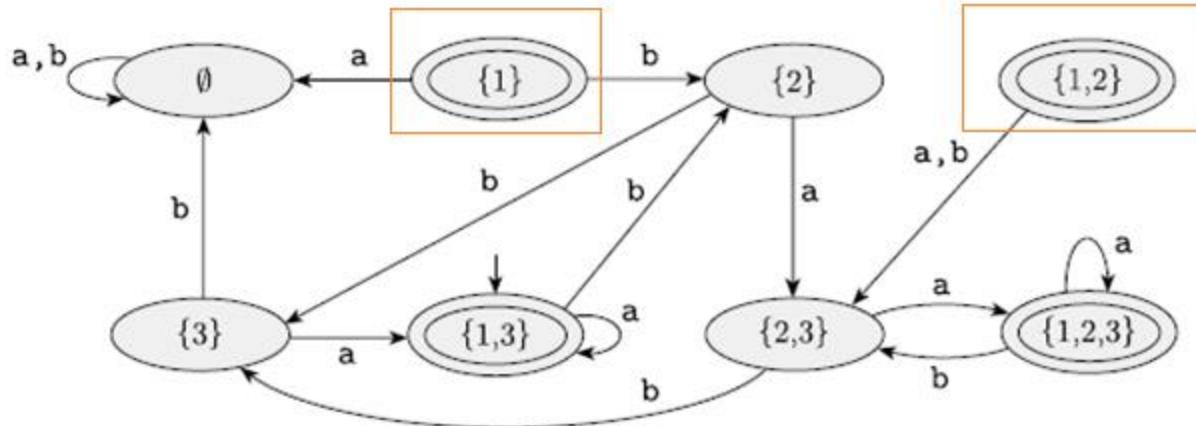
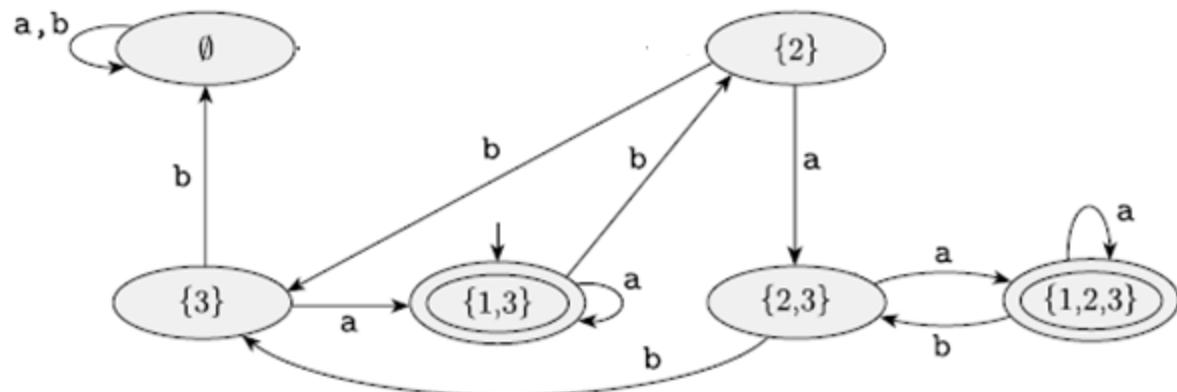


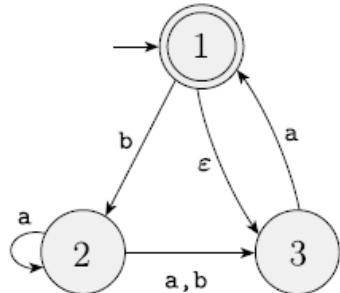
FIGURE 1.43

A DFA D that is equivalent to the NFA N_4

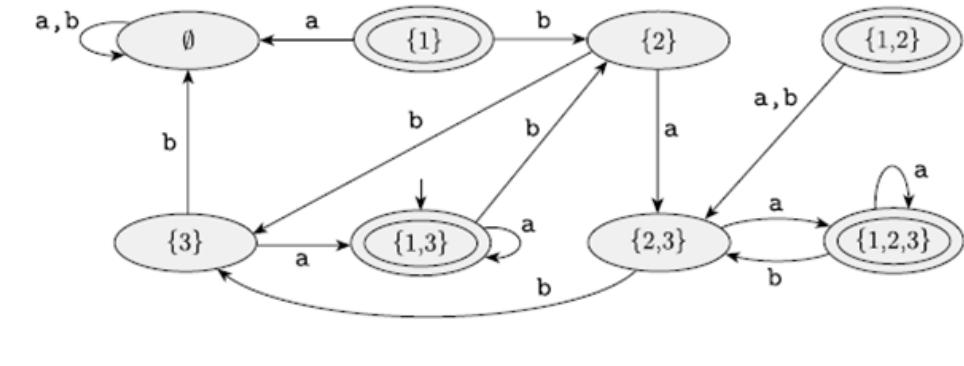


DFA D' which is equivalent to D .

Note: D' and D are different machines; but, they are equivalent.



N_4

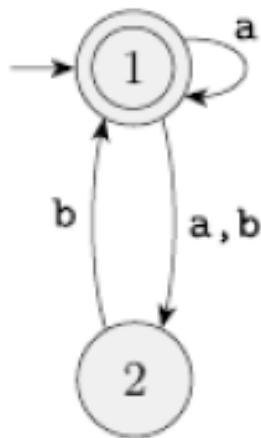


A DFA D that is equivalent to the NFA N_4

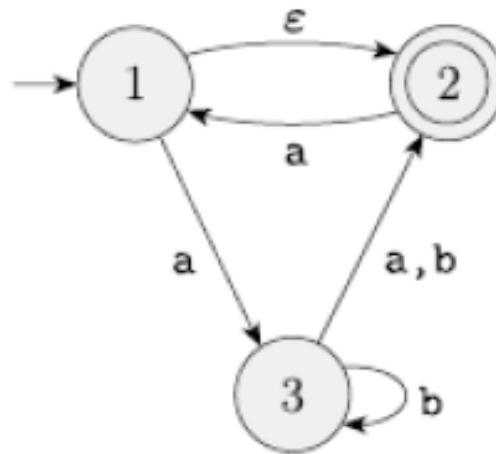
- Being in state 1 of N_4 upon reading input a the machine N_4 can be in state 1.
- Convince yourself that in the DFA D there are no mistakes.
 - From state $\{1\}$ with input a the DFA D goes to state \emptyset .

Exercise

Convert the following NFAs to equivalent DFAs.



(a)



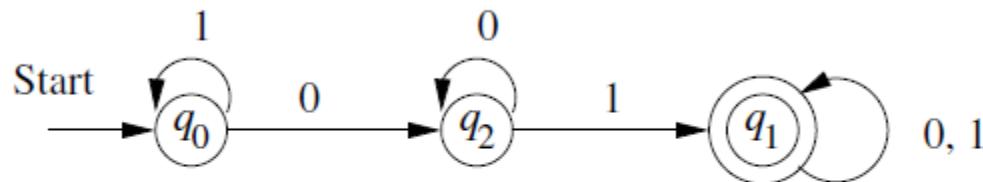
(b)

(Problem Source: Sipser's book exercise problem 1.16)

Exercise

- 1.7 Give state diagrams of NFAs with the specified number of states recognizing each of the following languages. In all parts, the alphabet is $\{0,1\}$.
 - a. The language $\{w \mid w \text{ ends with } 00\}$ with three states
 - b. The language $\{0\}$ with two states
 - c. The language $\{\epsilon\}$ with one state
 - d. The language 0^* with one state
- Can you convert each of above NFAs into a corresponding DFA.

Some Notation adapted



The transition diagram for the DFA accepting all strings with a substring 01

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

This also has all 5 components. This table is complete description of the DFA as the diagram.

Properties of Regular Languages

DEFINITION 1.16

A language is called a *regular language* if some finite automaton recognizes it.

- DFA and NFA are finite automaton
- So, a language recognized by DFA or NFA is a regular language.

Closure Properties

- **THEOREM 1.45** -----
 - The class of regular languages is closed under the union operation.
 - Product DFA construction proof, we have seen.
 - Now, we attempt using NFAs.

- Let $L(N_1) = A_1$, and $L(N_2) = A_2$

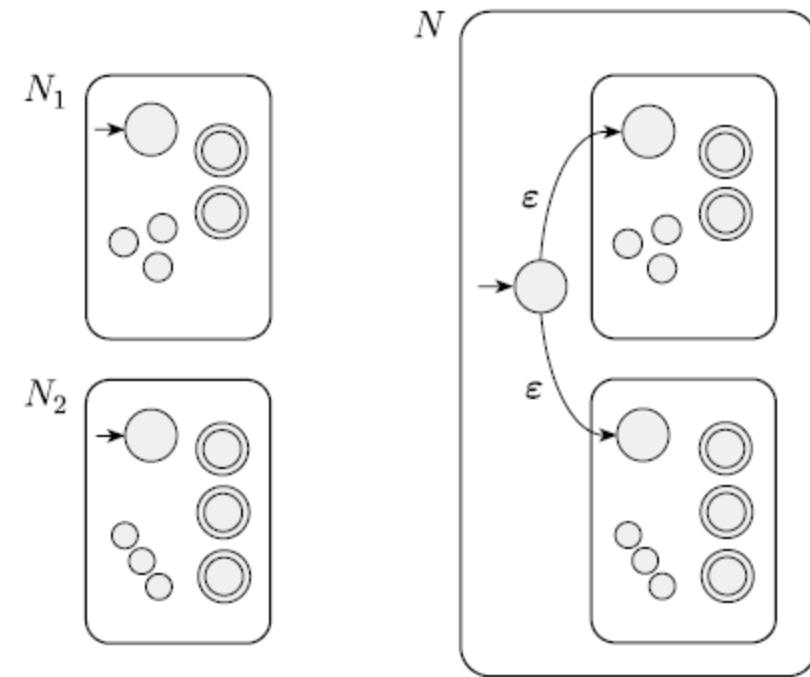


FIGURE 1.46

Construction of an NFA N to recognize $A_1 \cup A_2$

- Mathematical description of this construction is left as an exercise. {can refer to Sipser book}*

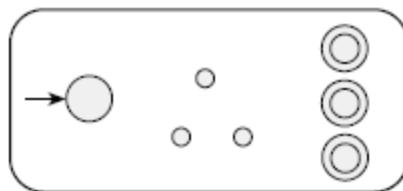
- But, for intersection, still product machine is needed. You cannot do like this for intersection.

- But, for intersection, still product machine is needed. You cannot do like this for intersection.
 - Trying to do product construction as we did for DFAs will not work.
 - Similarly complementation principle as we did with DFAs will not work for NFAs.

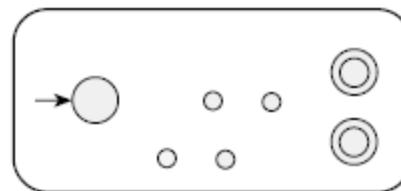
THEOREM 1.47

The class of regular languages is closed under the concatenation operation.

• N_1



N_2



N

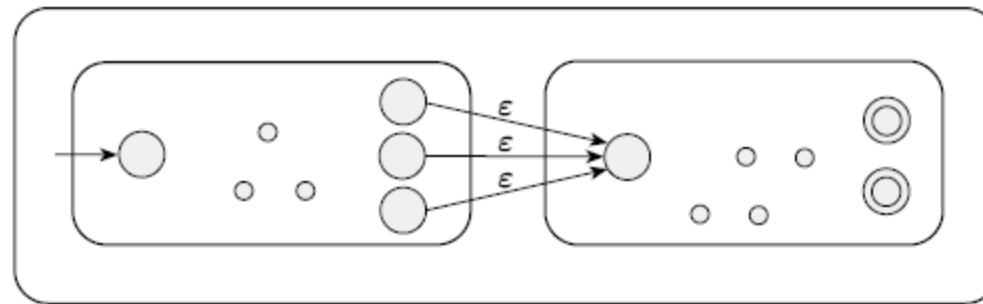


FIGURE 1.48

Construction of N to recognize $A_1 \circ A_2$

Mathematically,



PROOF

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1 , and
 $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2 .

Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognize $A_1 \circ A_2$.

1. $Q = Q_1 \cup Q_2$.
The states of N are all the states of N_1 and N_2 .
2. The state q_1 is the same as the start state of N_1 .
3. The accept states F_2 are the same as the accept states of N_2 .
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q, a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q, a) \cup \{q_2\} & q \in F_1 \text{ and } a = \varepsilon \\ \delta_2(q, a) & q \in Q_2. \end{cases}$$

THEOREM 1.49

The class of regular languages is closed under the star operation.

•

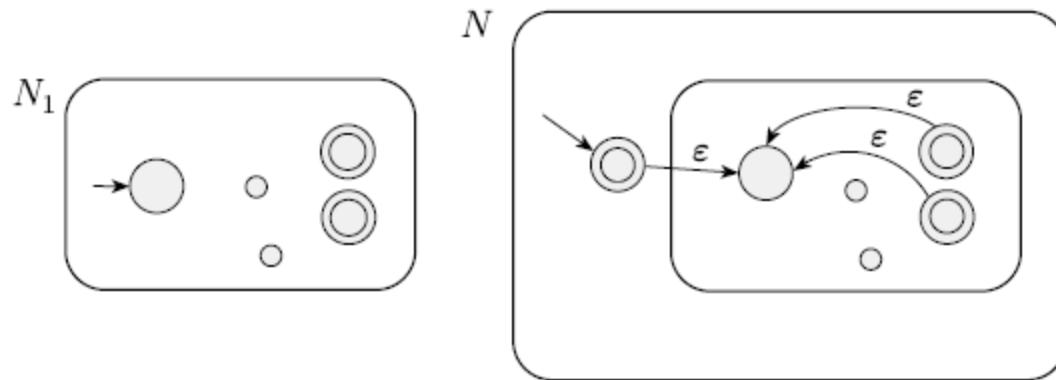


FIGURE 1.50

Construction of N to recognize A^*

PROOF Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1 .

Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize A_1^* .

1. $Q = \{q_0\} \cup Q_1$.

The states of N are the states of N_1 plus a new start state.

2. The state q_0 is the new start state.

3. $F = \{q_0\} \cup F_1$.

The accept states are the old accept states plus the new start state.

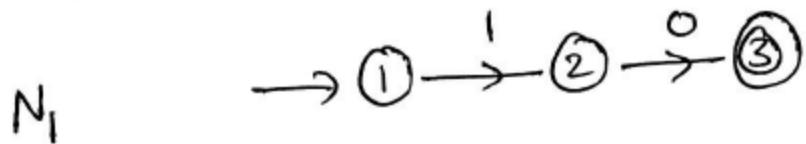
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q, a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q, a) \cup \{q_1\} & q \in F_1 \text{ and } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon. \end{cases}$$

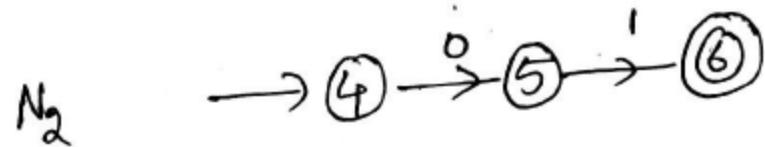
Exercise

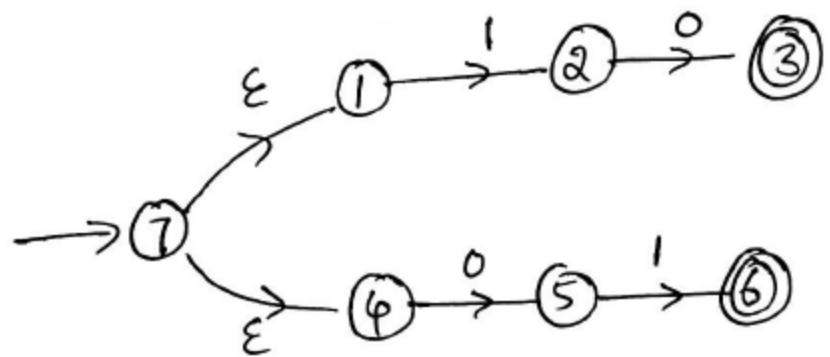
- **design a DFA to accept A^* where $A= \{10, 01\}$.**
 - Construct NFA
 - Then convert this to DFA

NFA for $\{10\}$

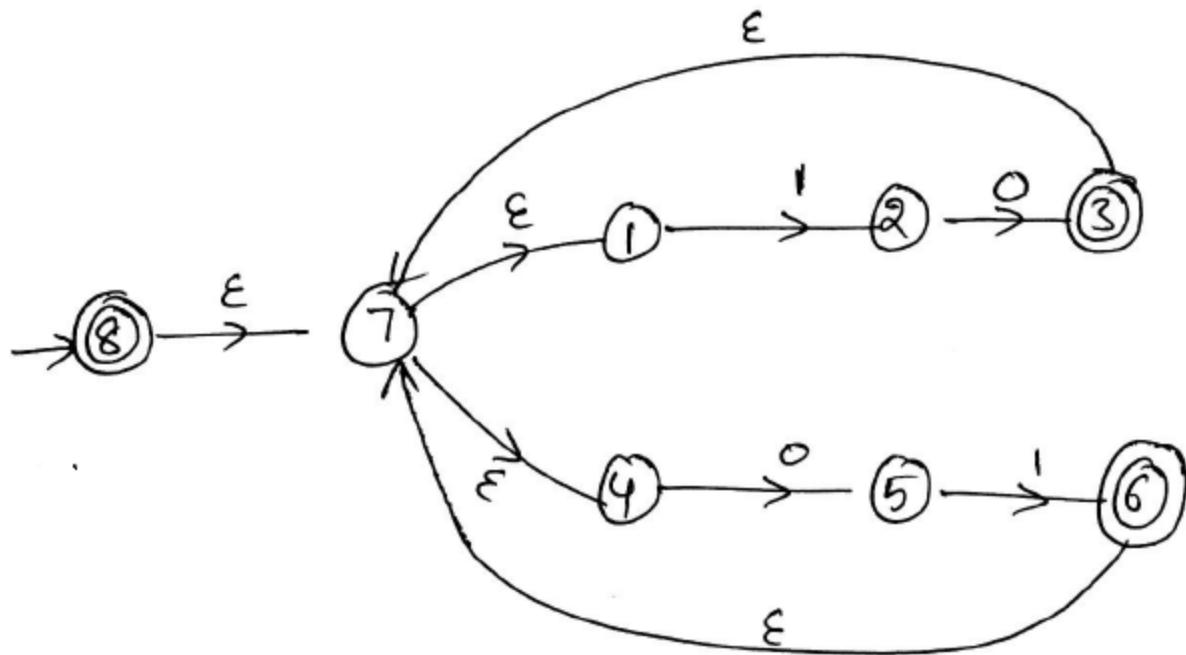


NFA for $\{01\}$





NFA N for $A = \{01, 10\}$

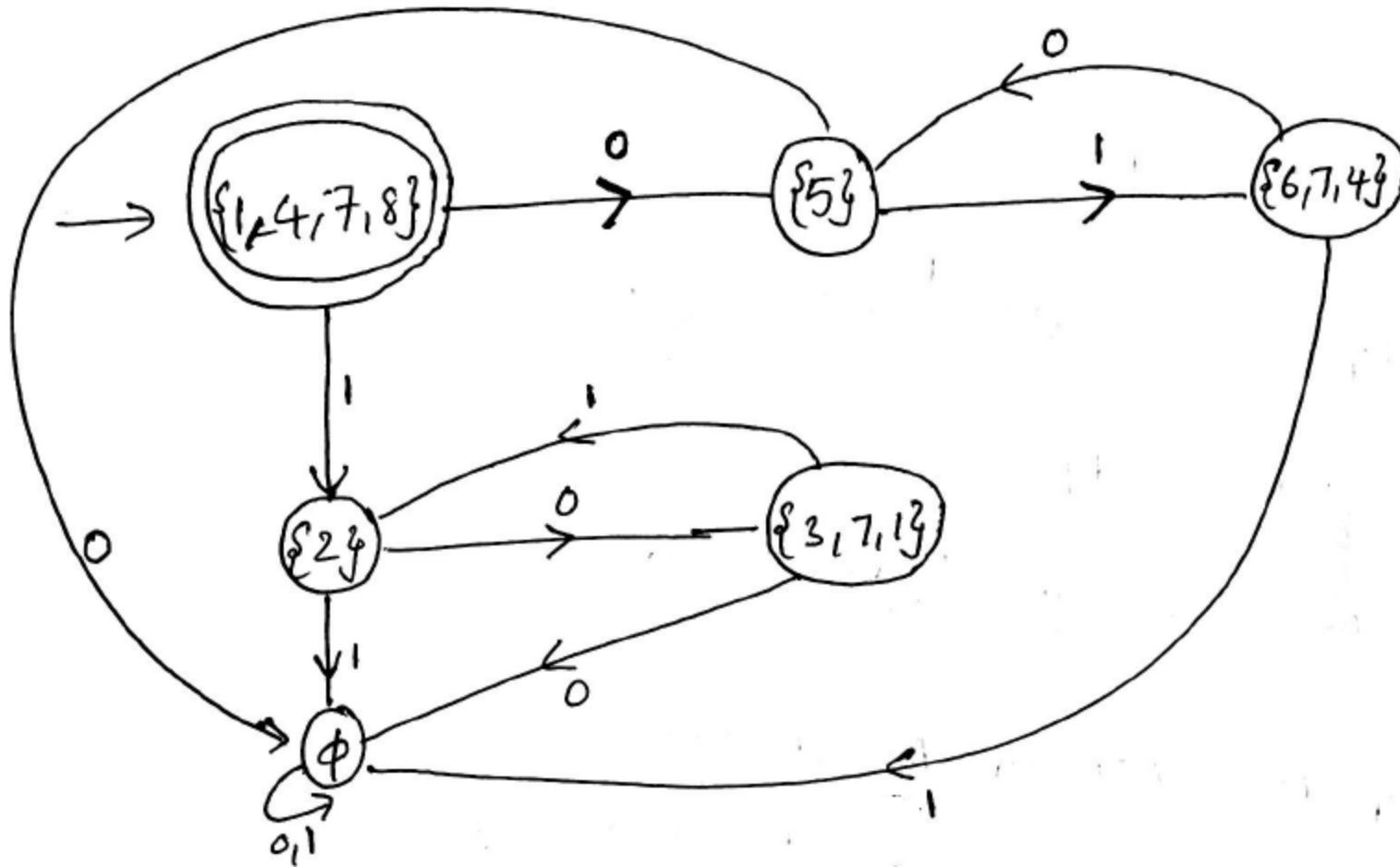


NFA for !^*

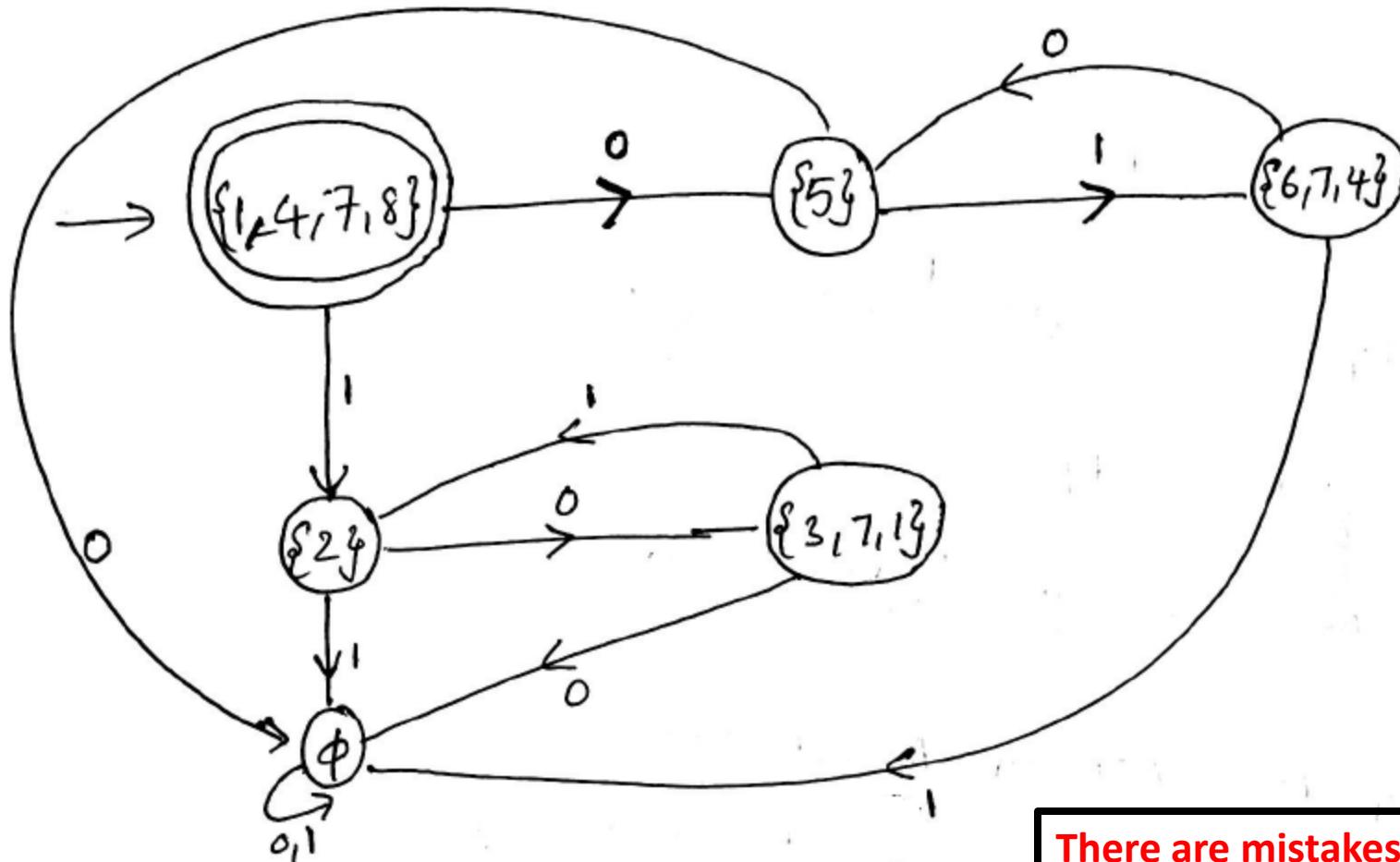
Now we should convert this to DFA.

Note that, $E(\{8\}) = \{8, 7, 1, 4\}$.

Now, can you convert this NFA in to an equivalent DFA ?



DFA for A^*



There are mistakes in this slide, try to correct them !!!

DFA for A^*

Some non-regular languages we already know are:

- Palindromes = $\{w = w^R \mid w \in \Sigma^*\}$
- Copy Language = $\{ww \mid w \in \Sigma^*\}$
- $\{a^n b^n \mid n \geq 0\}.$

Some non-regular languages we already know are:

- Palindromes = $\{w = w^R \mid w \in \Sigma^*\}$
 - Copy Language = $\{ww \mid w \in \Sigma^*\}$
 - $\{a^n b^n \mid n \geq 0\}$.
-
- But, recall $\{a^n b^n \mid 0 \leq n \leq 5\}$ is regular.
 - Every finite language is regular. {Can you prove this?}.
 - Σ^* is regular, ϕ is regular, ...

Regular Expressions

1.3

REGULAR EXPRESSIONS

In arithmetic, we can use the operations $+$ and \times to build up expressions such as

$$(5 + 3) \times 4.$$

Similarly, we can use the regular operations to build up expressions describing languages, which are called *regular expressions*. An example is:

$$(0 \cup 1)0^*.$$

- What are values of these expressions?

Where used

- Very useful to describe a set of strings having certain patterns.
 - In UNIX, rm *.c → removes all files ending with .c
 - Lex, a tool used in compiler generators
 - grep, awk available utilities in UNIX use regular expressions.

Meaning ...

0 means the language $\{0\}$

1 means $\{1\}$

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

0^* means $\{0\}^*$

$(0 \cup 1)0^*$ actually is shorthand for $(0 \cup 1) \circ 0^*$

$$(0 \cup 1)0^* = \{0, 1\}\{0\}^*$$

Inductive Definition

DEFINITION 1.52

Say that R is a *regular expression* if R is

1. a for some a in the alphabet Σ ,
2. ϵ ,
3. \emptyset ,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*) , where R_1 is a regular expression.

In items 1 and 2, the regular expressions a and ϵ represent the languages $\{a\}$ and $\{\epsilon\}$, respectively. In item 3, the regular expression \emptyset represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2 , or the star of the language R_1 , respectively.

Don't confuse between null string and null set

Don't confuse the regular expressions ϵ and \emptyset . The expression ϵ represents the language containing a single string—namely, the empty string—whereas \emptyset represents the language that doesn't contain any strings.

Parentheses in an expression may be omitted. If they are, evaluation is done in the precedence order: star, then concatenation, then union.

- Precedence is $*$, \circ , U
 - So, aUb^* is different from $(aUb)^*$
 $aUb^* = (a \cup (b))^*$
 - aUb^*c is same as $aU(b^*c)$
-
- Many authors use $+$ for U
So, $aUb = a+b$ But $+$ is overloaded.
Sipser reserved the $+$ to mean only one thing.

+ (positive closure)

$$R^* = R^0 \cup R^1 \cup R^2 \cup \dots \cup R^i \cup \dots$$

$$R^+ = R^1 \cup R^2 \cup \dots \cup R^i \cup \dots$$

$$R^+ = RR^*$$

$$R^* = R^+ \cup \epsilon$$

- The value of a regular expression R is nothing but the language represented by R
- When we want to distinguish between r.e. and the language represented by it. We use $L(R)$ to mean language represented by R .

We can write Σ as shorthand for regular expression $(0 \cup 1)^*$

From the context it should be clear for us by saying Σ do we mean alphabet or the language consisting of all possible strings of length 1.

Σ^* is a regular expression which is the language of all strings (including ϵ) over the alphabet Σ .

1. $0^*10^* = \{w \mid w \text{ contains a single } 1\}.$
2. $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}.$
3. $\Sigma^*001\Sigma^* = \{w \mid w \text{ contains the string } 001 \text{ as a substring}\}.$

Can you describe the language?

$$1^*(01^+)^* =$$

$$(\Sigma\Sigma)^* =$$

$$(\Sigma\Sigma\Sigma)^* =$$

$$01 \cup 10 =$$

$$0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 =$$

$$(0 \cup \varepsilon)(1 \cup \varepsilon) =$$

$$1^*\emptyset =$$

$$\emptyset^* =$$

Can you describe the language?

$1^*(01^+)^* = \{w \mid$ every 0 in w is followed by at least one 1 $\}.$

$(\Sigma\Sigma)^* =$

$(\Sigma\Sigma\Sigma)^* =$

$01 \cup 10 =$

$0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 =$

$(0 \cup \varepsilon)(1 \cup \varepsilon) =$

$1^*\emptyset =$

$\emptyset^* =$

Can you describe the language?

$1^*(01^+)^* = \{w \mid$ every 0 in w is followed by at least one 1 $\}.$

$(\Sigma\Sigma)^* = \{w \mid w \text{ is a string of even length}\}.$

$(\Sigma\Sigma\Sigma)^* =$

$01 \cup 10 =$

$0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 =$

$(0 \cup \varepsilon)(1 \cup \varepsilon) =$

$1^*\emptyset =$

$\emptyset^* =$

$1^*(01^+)^* = \{w \mid \text{every } 0 \text{ in } w \text{ is followed by at least one } 1\}.$

$(\Sigma\Sigma)^* = \{w \mid w \text{ is a string of even length}\}.$

$(\Sigma\Sigma\Sigma)^* = \{w \mid \text{the length of } w \text{ is a multiple of 3}\}.$

$01 \cup 10 = \{01, 10\}.$

$0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w \text{ starts and ends with the same symbol}\}.$

$(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}.$

$1^*\emptyset = \emptyset.$

$\emptyset^* = \{\varepsilon\}.$

Understood?

If we let R be any regular expression, we have the following identities. They are good tests of whether you understand the definition.

$$R \cup \emptyset = R.$$

Adding the empty language to any other language will not change it.

$$R \circ \epsilon = R.$$

Joining the empty string to any string will not change it.

Understood?

However, exchanging \emptyset and ε in the preceding identities may cause the equalities to fail.

$R \cup \varepsilon$ may not equal R .

For example, if $R = 0$, then $L(R) = \{0\}$ but $L(R \cup \varepsilon) = \{0, \varepsilon\}$.

$R \circ \emptyset$ may not equal R .

For example, if $R = 0$, then $L(R) = \{0\}$ but $L(R \circ \emptyset) = \emptyset$.

Compilers -- Tokens

- Lexical Analysis
 - Automatic tools can be used (like Lex)
 - But , you need to describe what you want.
- For Decimal Numbers:

$$(+ \cup - \cup \varepsilon) (D^+ \cup D^+ . D^* \cup D^* . D^+)$$

where $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is the alphabet of decimal digits. Examples of generated strings are: 72, 3.14159, +7 ., and -.01.

Equivalence of RE with DFA/NFA

- It is somewhat surprising to note that, RE can be used to describe any regular language.
 - This is not true for other higher level languages, like CFL {We can describe a CFL by a CFG, not by an expression}.
- Now, how is that we prove this?

Proof has two directions

- Given RE, show that we can build a DFA/NFA recognizing the language given by the RE.
- Given DFA/NFA, show that we can convert this in to a RE.

To Show

- Given RE, show that we can build a NFA recognizing the language given by the RE.
- We use inductive definition of RE.

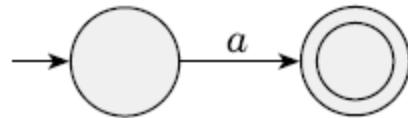
Inductive Definition of RE

DEFINITION 1.52

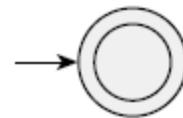
Say that R is a *regular expression* if R is

1. a for some a in the alphabet Σ ,
2. ϵ ,
3. \emptyset ,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*) , where R_1 is a regular expression.

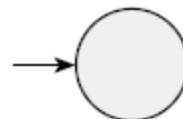
1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.



2. $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$, and the following NFA recognizes $L(R)$.



3. $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$.

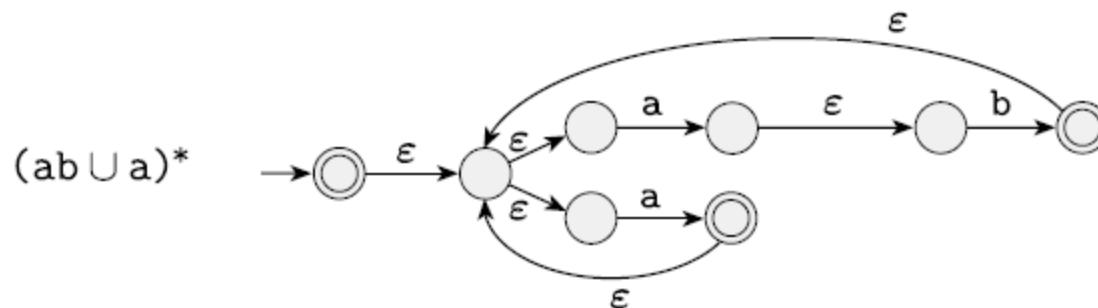
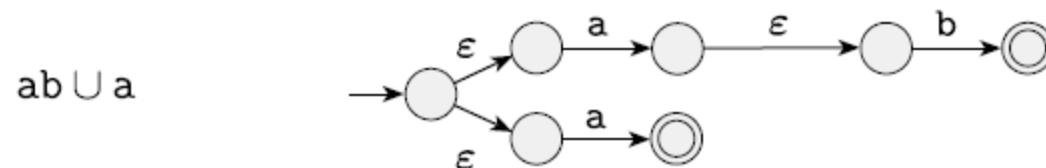


- One should be able to do these formally !!

- 4. $R = R_1 \cup R_2.$
- 5. $R = R_1 \circ R_2.$
- 6. $R = R_1^*.$

- **Use the construction proofs.**

We convert the regular expression $(ab \cup a)^*$ to an NFA in a sequence of stages.



Equivalence between RE and DFA/NFA

- The second part is,
- Given DFA/NFA convert this to equivalent RE.
- There are various ways for this.
 - The Ullman's book gives a rigorous algorithm
 - Same thing in essence is achieved by the Sipser's book in a different way.
 - We follow Sipser's book.

DFA/NFA → RE

- Go for GNFA (Generalized Nondeterministic Finite Automata)
- RE can be on arrows.

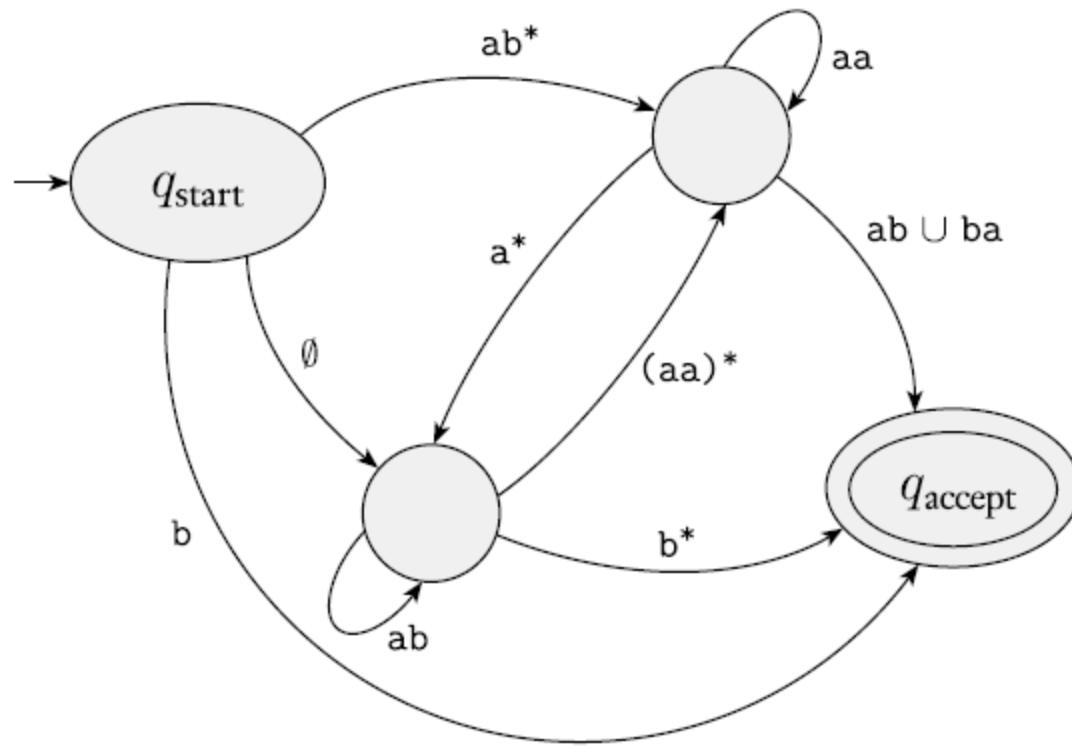


FIGURE 1.61

A generalized nondeterministic finite automaton

GNFA should -

- The start state has transition arrows going to every other state but no arrows coming in from any other state.
- There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
- Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.
- For more details, consult the Sipser's book.

What we do?

- We convert the given DFA/NFA into a GNFA.
- Then, progressively we remove all states, one by one, except for the start and accept states.

What we do?

- For example we are given a 3 state DFA

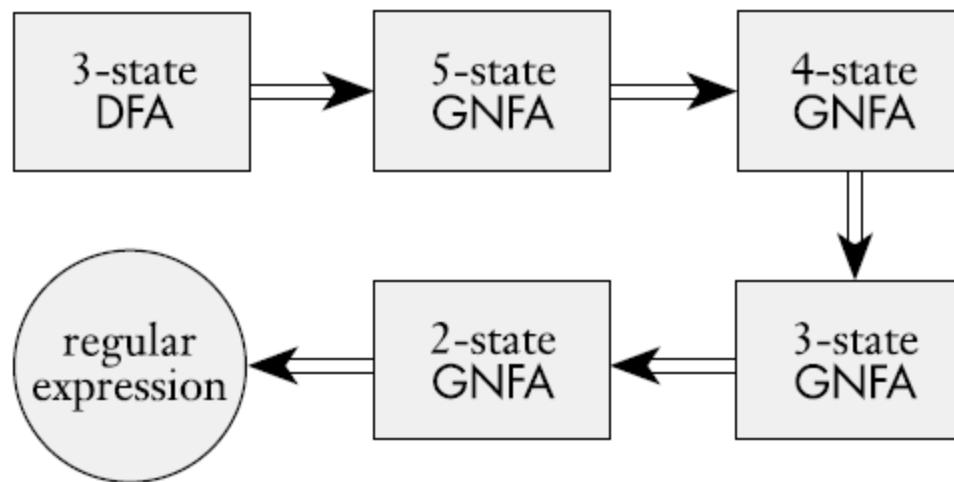


FIGURE 1.62

Typical stages in converting a DFA to a regular expression

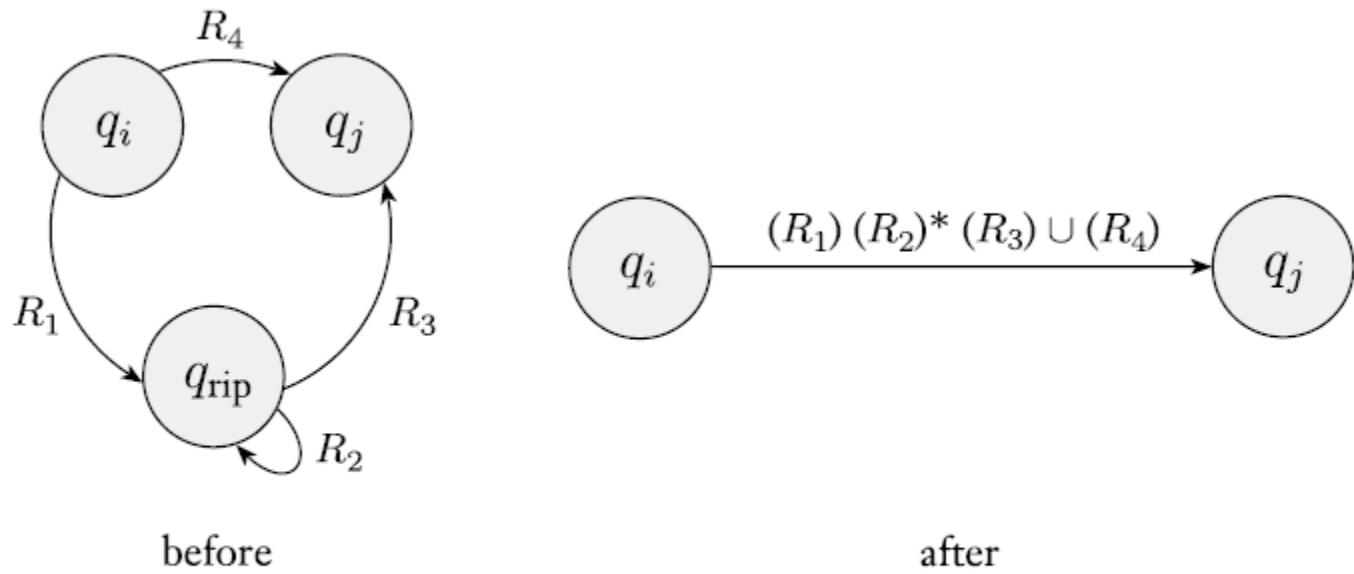
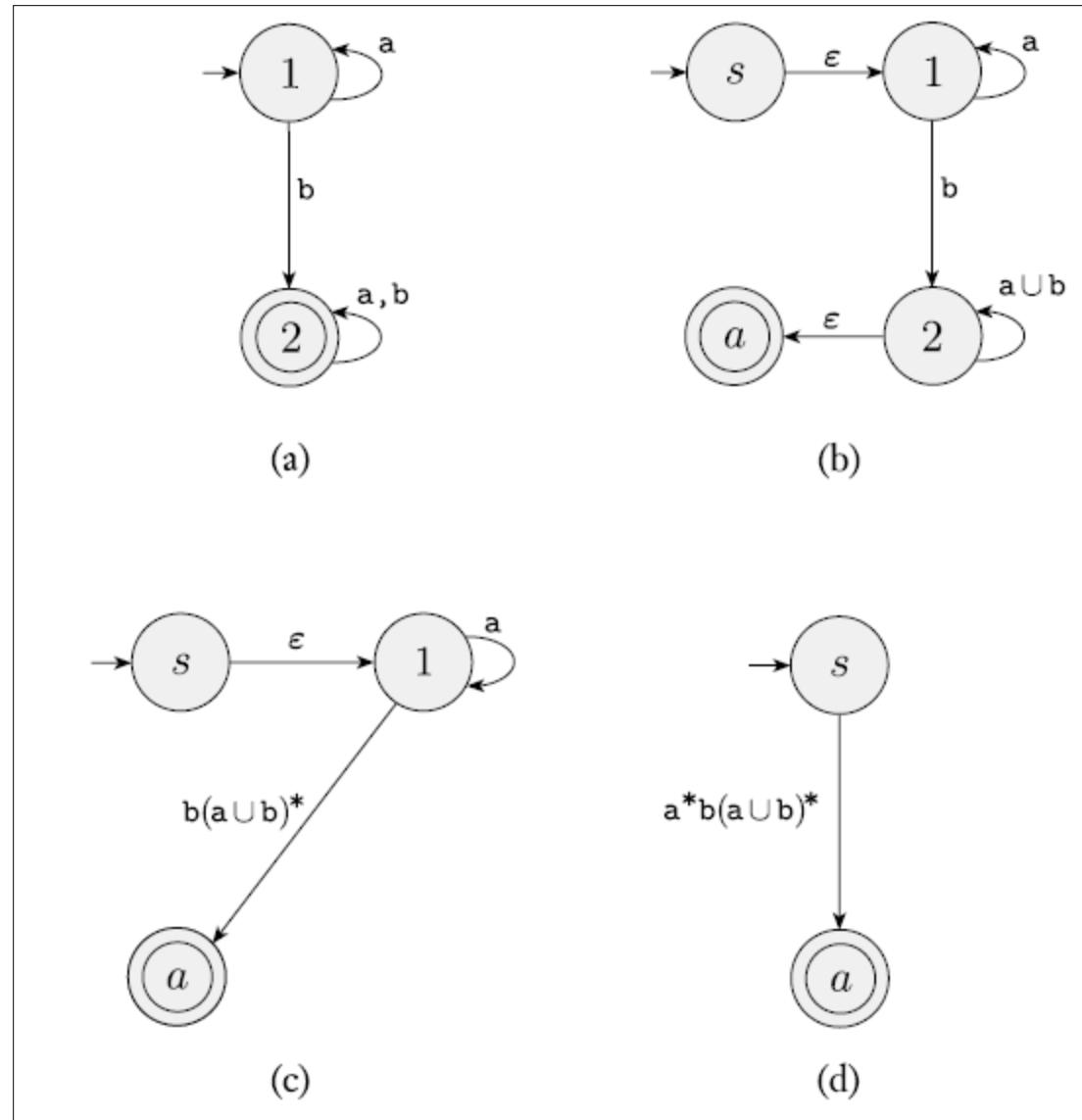


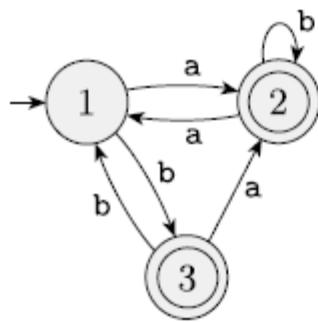
FIGURE 1.63
Constructing an equivalent GNFA with one fewer state

Example 1

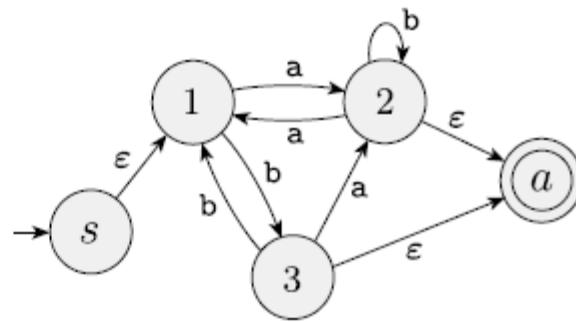


To avoid cluttering up the figure, we do not draw the arrows labeled \emptyset , even though they are present.

Example 2

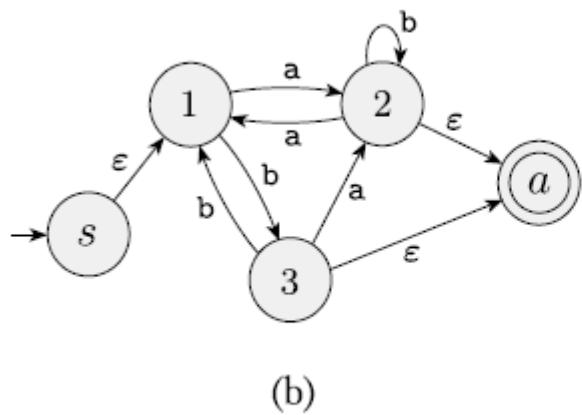


(a)

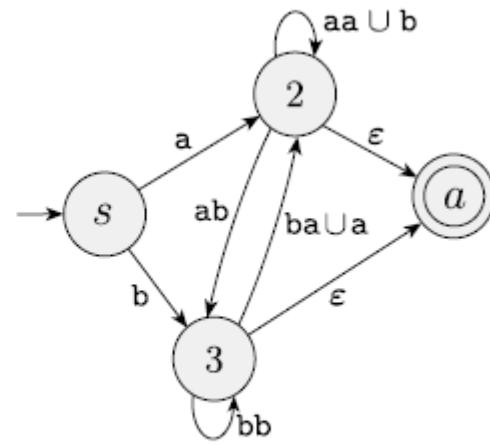


(b)

Example 2...

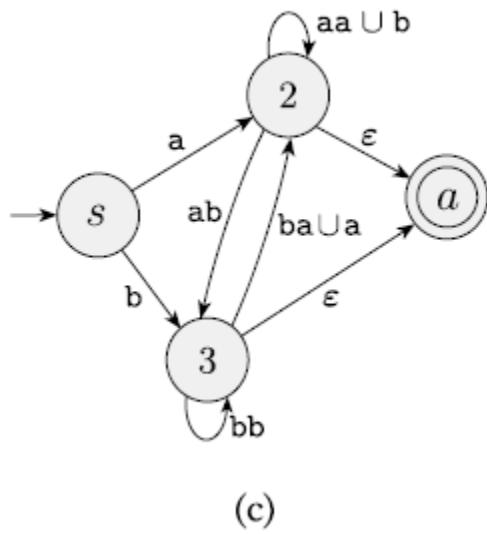


(b)

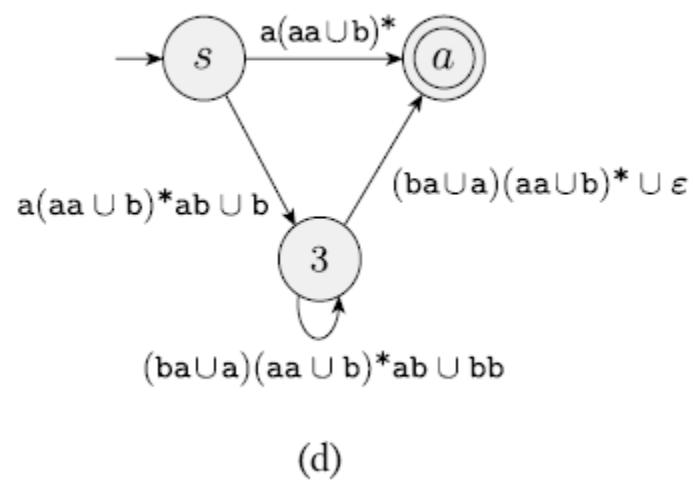


(c)

Example 2...

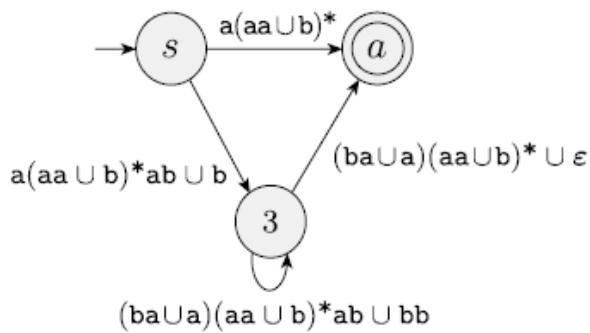


(c)

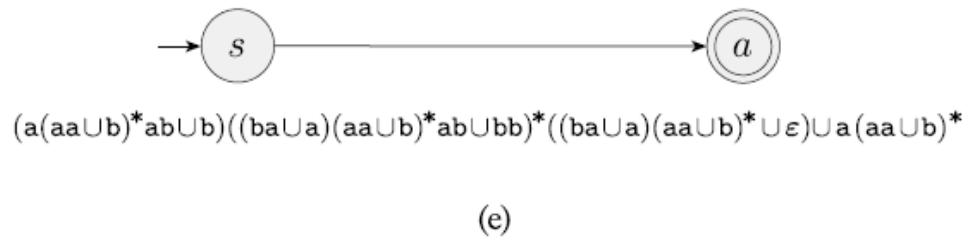


(d)

Example 2...



(d)



(e)

Laws concerning R.E.

- We overload + to mean \cup also.
 - You have to live with this notational abuse between Sipser and Ullman.
- $a+b = a \cup b$

3.4 Algebraic Laws for Regular Expressions

➤ Commutativity and Associativity

- $L + M = M + L$ (Remember, $L \cup M = M \cup L$)
- $(L + M) + R = L + (M + R) = L + M + R$
- $(LM)R = L(MR) = LMR$

➤ Left distribution and right distribution

- $L(M + N) = LM + LN$
- $(M + N)L = ML + NL$

Identities and Annihilors

- $\emptyset + L = L + \emptyset = L$. This law asserts that \emptyset is the identity for union.
- $\epsilon L = L\epsilon = L$. This law asserts that ϵ is the identity for concatenation.
- $\emptyset L = L\emptyset = \emptyset$. This law asserts that \emptyset is the annihilator for concatenation.

Idempotent Laws

- $L+L = L$
- $(L^*)^* = L^*$

How to prove?

- Inequality can be easily proved by a counter example.
- But, equality, to be proved is cumbersome.
 - You can follow your set theory knowledge to deduct that from LHS, RHS is deductible.
 - These is an established simple way of doing this.

Assuming only three regular operators,
viz., $,$ $+$, \cdot $*$ are only used.

- To test whether $E = F$ is true or false.
 1. Convert E and F to concrete regular expressions C and D , respectively, by replacing each variable by a concrete symbol.
 2. Test whether $L(C) = L(D)$. If so, then $E = F$ is a true law, and if not, then the “law” is false.
- What do you mean by **concrete r.e.** ?

Concretizing a r.e.

Let $E = P + Q(RS^*)$ be a regular expression where P, Q, R, S are some regular expressions.

Here, we say E has variables P, Q, R, S .

Concretizing E means replacing each variable in E by a distinct symbol.

In this example, we can concretize $P + Q(RS^*)$ to $a + b(cd^*)$

To prove, $P(M + N) = PM + PN$.

Concretizing L.H.S will give us $a(b + c)$

Concretizing R.H.S will give us $ab + ac$

Now, one has to show $L(a(b + c)) = L(ab + ac)$, which can be done easily.

Now, verify whether $PR = RP$ is true or false.

Concretize L.H.S in to ab

Concretize R.H.S in to ba

Now, it is clear that $L(ab) = \{ab\}$ is not equal to $L(ba) = \{ba\}$.

! Exercise 3.4.2: Prove or disprove each of the following statements about regular expressions.

- * a) $(R + S)^* = R^* + S^*$.
- b) $(RS + R)^*R = R(SR + R)^*$.
- * c) $(RS + R)^*RS = (RR^*S)^*$.
- d) $(R + S)^*S = (R^*S)^*$.
- e) $S(RS + S)^*R = RR^*S(RR^*S)^*$.

Don't use operators beyond

+ · *

- That is, stick to, regular operators only.

Extensions of the Test Beyond Regular Expressions May Fail

Consider the "law" $L \cap M \cap N = L \cap M$;

Concretizing, we get, $\{a\} \cap \{b\} \cap \{c\} = \{a\} \cap \{b\}$. This is true.

But clearly the above "law" is false.

Counter example to disprove the "law".

For example, let $L = M = \{a\}$ and $N = \emptyset$.

Clearly L.H.S and R.H.S are distinct languages.

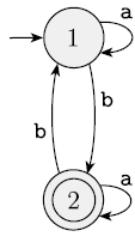
Tutorial problems on Regular Expressions

From Sipser's book –

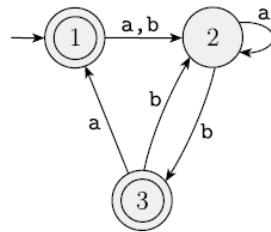
- 1.20 For each of the following languages, give two strings that are members and two strings that are *not* members—a total of four strings for each part. Assume the alphabet $\Sigma = \{a, b\}$ in all parts.

- | | |
|-------------------|--|
| a. a^*b^* | e. $\Sigma^* a \Sigma^* b \Sigma^* a \Sigma^*$ |
| b. $a(ba)^*b$ | f. $aba \cup bab$ |
| c. $a^* \cup b^*$ | g. $(\epsilon \cup a)b$ |
| d. $(aaa)^*$ | h. $(a \cup ba \cup bb)\Sigma^*$ |

- 1.21 Use the procedure described in Lemma 1.60 to convert the following finite automata to regular expressions.



(a)



(b)

- 1.28 Convert the following regular expressions to NFAs .

In all parts, $\Sigma = \{a, b\}$.

- $a(ab\bar{b})^* \cup b$
- $a^+ \cup (ab)^+$
- $(a \cup b^+)a^+b^+$

From Ullman's book –

3.4.8 Exercises for Section 3.4

Exercise 3.4.1: Verify the following identities involving regular expressions.

- * a) $R + S = S + R$.
- b) $(R + S) + T = R + (S + T)$.
- c) $(RS)T = R(ST)$.
- d) $R(S + T) = RS + RT$.
- e) $(R + S)T = RT + ST$.
- * f) $(R^*)^* = R^*$.
- g) $(\epsilon + R)^* = R^*$.
- h) $(R^*S^*)^* = (R + S)^*$.

! Exercise 3.4.2: Prove or disprove each of the following statements about regular expressions.

- * a) $(R + S)^* = R^* + S^*$.
- b) $(RS + R)^*R = R(SR + R)^*$.
- * c) $(RS + R)^*RS = (RR^*S)^*$.
- d) $(R + S)^*S = (R^*S)^*$.
- e) $S(RS + S)^*R = RR^*S(RR^*S)^*$.

Nonregular Languages

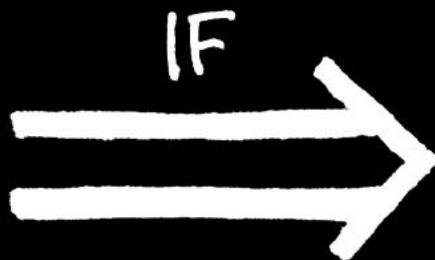
- How to show that a given language is nonregular.
- In some sense, we need to prove that No DFA is possible to recognize the language.
- How we do this?

Some properties can help us

- L is regular $\Rightarrow L$ obeys “Pumping Lemma”
- DFA must have finite number of states.
 - For the given L , we need infinite number of states in the DFA.
 - Myhill-Nerode Theorem (Gives a **necessary and sufficient** condition for regular languages).
- There are other ways ...

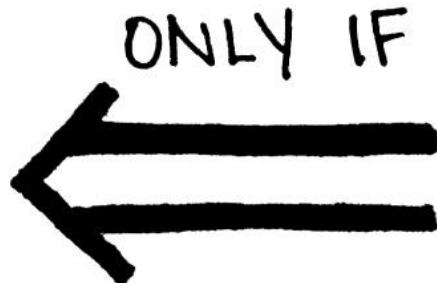
- Pumping Lemma is useful to show that L is nonregular.
- It cannot be used to show that L is regular.
- Why?

the sufficient condition



(If you assume this, you'll get what you want.)

the necessary condition



(You can't get what you want without assuming this.)

- $A \Rightarrow B$ (A being true is a sufficient condition for B to be true)
 - If A is true, we know B is true.
 - If A is false, what about B?
- $A \leq B$ (A being true is a necessary condition for B to be true)
 - If A is false, we know B is false.
 - If A is true, what about B?

- $A \Rightarrow B$ (A being true is a sufficient condition for B to be true)
 - If A is true, we know B is true.
 - If A is false, what about B?
 - If B is false, then what about A?

- L is regular $\Rightarrow L$ obeys “Pumping Lemma”
 - If L fails to obey “Pumping Lemma” then L is nonregular.
-
- Moral of the story: Never use Pumping Lemma to prove that L is regular.

- Nonregular examples

$$B = \{0^n 1^n \mid n \geq 0\}$$

$$C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$$

- But, the following is regular

$$D = \{w \mid w \text{ has an equal number of occurrences of 01 and 10 as substrings}\}$$

- Nonregular examples

$$B = \{0^n 1^n \mid n \geq 0\}$$

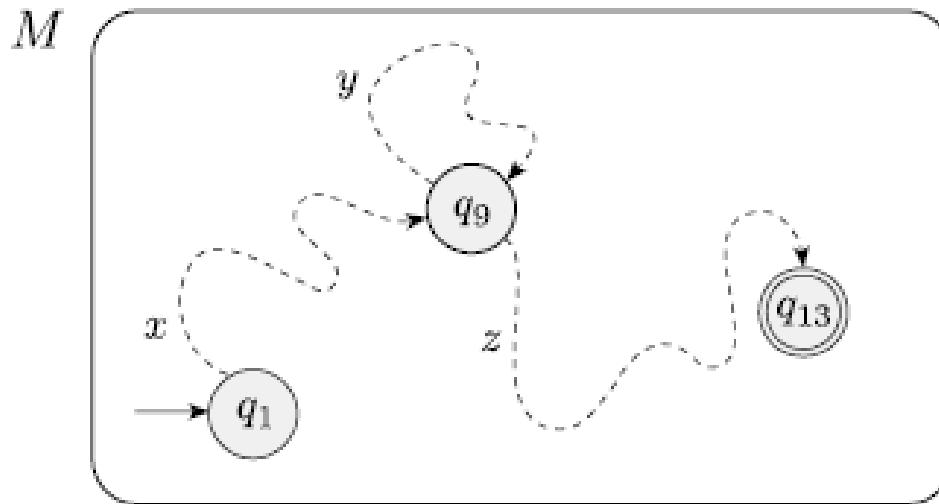
$$C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$$

- But, the following is regular

$$D = \{w \mid w \text{ has an equal number of occurrences of } 01 \text{ and } 10 \text{ as substrings}\}$$

- http://www.cs.gordon.edu/courses/cps220/Notes/nonregular_languages
 - Follow the above URL for an answer to show D is regular.

Pumping Lemma



- In any DFA, if $w = xyz$ is “long enough”, then such a loop must occur. Why?

Pigeonhole Principle



The following figure shows the string s and the sequence of states that M goes through when processing s . State q_9 is the one that repeats.

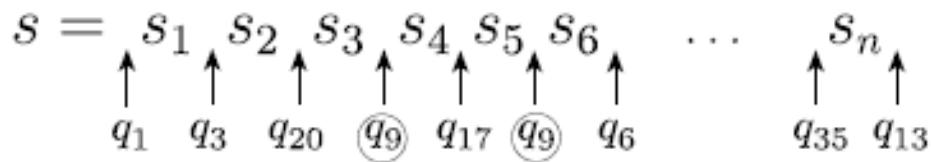


FIGURE 1.71
Example showing state q_9 repeating when M reads s

Pumping Lemma

THEOREM 1.70

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p , then s can be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

THEOREM 1.70

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p , then s can be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

When s is divided into xyz , either x or z may be ϵ , but condition 2 says that $y \neq \epsilon$.

Observe that without condition 2 the theorem would be trivially true.

Negation of Pumping Lemma

- There is **a** string w in L which of length atleast of the pumping length (p), where **every** division of w into xyz fails to satisfy at-least **one** of the following –
 - $|y| \neq 0$
 - $|xy| \leq p$
 - $x y^i z$ is in L for all i in $\{0,1,2,\dots\}$.

Negation of Pumping Lemma (we simplify)

- There is **a** string w in L which of length atleast of the pumping length (p), where **every** division of w into xyz that obeys
 - $|y| \neq 0$
 - $|xy| \leq p$Fails to satisfy the following for at-least one i .
 - $x y^i z$ is in L for all i in $\{0,1,2,\dots\}$.

Show that, $L = \{0^{n^2} | n \geq 1\}$ is nonregular.

Show that, $L = \{0^{n^2} | n \geq 1\}$ is nonregular.

Proof [by Pumping Lemma]:

Let p be the pumping length.

Let us choose $w = 0^{p^2}$ and we know, $|w| \geq p$.

Show that, $L = \{0^{n^2} | n \geq 1\}$ is nonregular.

Proof [by Pumping Lemma]:

Let p be the pumping length.

Let us choose $w = 0^{p^2}$ and we know, $|w| \geq p$.

Let us choose $x = 0^m$, $y = 0^n$, $z = 0^r$ and $m + n + r = p^2$.

Show that, $L = \{0^{n^2} | n \geq 1\}$ is nonregular.

Proof [by Pumping Lemma]:

Let p be the pumping length.

Let us choose $w = 0^{p^2}$ and we know, $|w| \geq p$.

Let us choose $x = 0^m$, $y = 0^n$, $z = 0^r$ and $m + n + r = p^2$.

Along with this, we have $1 \leq n \leq p$.

Show that, $L = \{0^{n^2} | n \geq 1\}$ is nonregular.

Proof [by Pumping Lemma]:

Let p be the pumping length.

Let us choose $w = 0^{p^2}$ and we know, $|w| \geq p$.

Let us choose $x = 0^m$, $y = 0^n$, $z = 0^r$ and $m + n + r = p^2$.

Along with this, we have $1 \leq n \leq p$.

Now, consider the string xy^2z . Note, $|xy^2z| = p^2 + n$.

We have, $p^2 < p^2 + n < (p + 1)^2$.

Show that, $L = \{0^{n^2} | n \geq 1\}$ is nonregular.

Proof [by Pumping Lemma]:

Let p be the pumping length.

Let us choose $w = 0^{p^2}$ and we know, $|w| \geq p$.

Let us choose $x = 0^m$, $y = 0^n$, $z = 0^r$ and $m + n + r = p^2$.

Along with this, we have $1 \leq n \leq p$.

Now, consider the string xy^2z . Note, $|xy^2z| = p^2 + n$.

We have, $p^2 < p^2 + n < (p + 1)^2$.

So, $|xy^2z|$ is not a perfect square, hence xy^2z is not in L .

Thus, Pumping Lemma failed for L .

EXAMPLE

Let B be the language $\{0^n 1^n \mid n \geq 0\}$. We use the pumping lemma to prove that B is not regular.

Let p be the pumping length.

Let $s = 0^{\lceil p/2 \rceil} 1^{\lceil p/2 \rceil}$

Because s is a member of B and s has length more than p , the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^i z$ is in B with conditions $y \neq \epsilon$ and $|xy| \leq p$.

We consider three cases to show that this result is impossible.

1. The string y consists only of 0s. In this case, the string $xyyz$ has more 0s than 1s and so is not a member of B , violating condition 1 of the pumping lemma. This case is a contradiction.
2. The string y consists only of 1s. This case also gives a contradiction.
3. The string y consists of both 0s and 1s. In this case, the string $xyyz$ may have the same number of 0s and 1s, but they will be out of order with some 1s before 0s. Hence it is not a member of B , which is a contradiction.

A good choice for the string

EXAMPLE

Let B be the language $\{0^n 1^n \mid n \geq 0\}$. We use the pumping lemma to prove that B is not regular.

Let p be the pumping length.

Choose s to be the string $0^p 1^p$.

Because s is a member of B and s has length more than p ,
the pumping lemma guarantees that s can be split into three
pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^i z$ is in B .

With conditions $y \neq \epsilon$ and $|xy| \leq p$,

y can be of only 0s,

and the string $xy^2 z$ will clearly have more 0s than 1s,

hence is not in the language.

Following are all nonregular.

1. Strings having equal number of 0s and 1s.

2. Dyck language.

$\Sigma = \{(), ()\}$. Dyck language is the set of all balanced strings like $\{(), (), (), ((())), \dots\}$

3. Palindromes (over any alphabet, other than unary alphabet).

4. Copy language, i.e., $L = \{ww \mid w \in \Sigma^*\}$.

5. $L = \{0^n 1 0^n \mid n \geq 0\}$.

6. $L = \{ww^R \mid w \in \Sigma^*\}$.

- Can you prove for each of these that PL fails.

What is wrong?

In order to show that the set of palindromes over $\Sigma = \{0,1\}$ regular,

I have chosen $s = 0^{\lceil p/2 \rceil} 1 0^{\lceil p/2 \rceil}$.

Now, I split $s = xyz$, with $y = 1$.

I can pump y as many times as I want and the resulting string is in the language.

So, the language is regular.

- There are two mistakes.

Mistakes.

- The argument is not for a particular division of the chosen s in to xyz .
But, for every division, satisfying the conditions $y \neq \epsilon$ and $|xy| \leq p$.

Mistakes.

- If you want to show that Pumping Lemma is true, then you have to show for all strings s , such that $|s| \geq p$, s can be divided in to xyz , satisfying the three conditions. Just showing it for one s is not enough.
- Note, on the otherhand, to show that Pumping Lemma is false, you can choose just one string s whose length is at-least p , but, now, for every division of s in to xyz , at-least one of the three conditions is not satisfied.
- But, the serious mistake is, you have not learnt the moral.
- Never use PL to show that a language is regular.

Set of primes – a nonregular language

- Let p be the pumping length.
- Consider $s = 0^n$ where n is prime and let $n \geq p$, represents a prime number.

Set of primes – a nonregular language

- Let p be the pumping length.
- Consider $s = 0^n$ where n is prime and let $n \geq p$, represents a prime number.
- Now, let us divide $s = 0^x 0^y 0^{(n-x-y)}$ and $y > 0, x + y \leq p$.

Set of primes – a nonregular language

- Let p be the pumping length.
- Consider $s = 0^n$ where n is prime and let $n \geq p$, represents a prime number.
- Now, let us divide $s = 0^x 0^y 0^{(n-x-y)}$ and $y > 0, x + y \leq p$.
- Consider $i = n + 1$.
- We show $0^x (0^y)^i 0^{(n-x-y)}$ does not represent a prime number

- $0^x (0^y)^i 0^{(n-x-y)} = 0^{x+y(n+1)+n-x-y}$
 $= 0^{n(y+1)}$

Show that $L = \{a^i b^j \mid i \neq j\}$ is non-regular.

Direct proof using pumping lemma is somewhat an involved one. All the trouble is in choosing an appropriate $s \in L$ for which the lemma is going to fail. After some investigation the following s is found which will ease out the proof.

Show that $L = \{a^i b^j \mid i \neq j\}$ is non-regular.

Direct proof using pumping lemma is somewhat an involved one. All the trouble is in choosing an appropriate $s \in L$ for which the lemma is going to fail. After some investigation the following s is found which will ease out the proof.

Proof:

Let the pumping length be p .

Choose $s = a^p b^{p! + p}$. Here $p!$ is factorial of p .

Let $s = xyz$ where $x = a^{p-n}$, $y = a^n$, $z = b^{p! + p}$ such that $1 \leq |n| \leq p$.

This division of s into xyz satisfies the constraints, viz., (i) $|y| \neq 0$, and (ii) $|xy| \leq p$.

We show that for some i , $xy^i z \notin L$.

Show that $L = \{a^i b^j \mid i \neq j\}$ is non-regular.

Direct proof using pumping lemma is somewhat an involved one. All the trouble is in choosing an appropriate $s \in L$ for which the lemma is going to fail. After some investigation the following s is found which will ease out the proof.

Proof:

Let the pumping length be p .

Choose $s = a^p b^{p! + p}$. Here $p!$ is factorial of p .

Let $s = xyz$ where $x = a^{p-n}$, $y = a^n$, $z = b^{p! + p}$ such that $1 \leq |n| \leq p$.

This division of s into xyz satisfies the constraints, viz., (i) $|y| \neq 0$, and (ii) $|xy| \leq p$.

We show that for some i , $xy^i z \notin L$.

Choose $i = \frac{p!}{n} + 1$. Note this i is a non-negative integer. Then $xy^i z =$

$$a^{p-n}(a^n)^{\frac{p!}{n}+1}b^{p!+p} = a^{p-n+p!+n}b^{p!+p} = a^{p!+p}b^{p!+p} \notin L.$$

An easy way to show $\{a^i b^j \mid i \neq j\}$ is nonregular

We know $a^* b^*$ is regular (why?)

We know $\{a^n b^n \mid n \geq 0\}$ is nonregular. Since PL fails for this.

Now assume $\{a^i b^j \mid i \neq j\}$ is regular.

Now this leads to a contradiction.

Can you prove these

1. $\{a^m b^n \mid m < n\}$ is nonregular.
2. $\{a^m b^n \mid m \leq n\}$ is nonregular.
3. $\{a^m b^n \mid m > n\}$ is nonregular.
4. $\{a^m b^n \mid m \geq n\}$ is nonregular.

- Prove or disprove: “every finite language is regular”.
- Prove or disprove: “every infinite language is nonregular”.

- Prove or disprove: “every finite language is regular”.
- True. We can build a NFA.
- Prove or disprove: “every infinite language is nonregular”.
- False. Counter example is: a^*b^*

- Prove or disprove : “nonregular languages are closed under union”.

- Prove or disprove : “nonregular languages are closed under union”.
- False.
- Counter example:

$\{a^n b^n | n \geq 0\} \cup \{a^i b^j | i \neq j\}$ is equal to $a^* b^*$, which is regular.

- Prove or disprove : “nonregular languages are closed under intersection”.

- Prove or disprove : “nonregular languages are closed under intersection”.
- False.
- Counter example:

$\{a^n b^n | n \geq 0\} \cap \{a^i b^j | i \neq j\}$ is empty language, which is regular.

- Prove or disprove : “nonregular languages are closed under complementation”.

- Prove or disprove : “nonregular languages are closed under complementation”.
- True.
- Proof: [by contradiction] using the fact that regular languages are closed under complementation.

Example of nonregular language that satisfies the pumping lemma

Let $\Sigma = \{\$\text{, } a\text{, } b\}$.

Consider the language $L = \{\$a^n b^n | n \geq 1\} \cup \{ \$^k w | k \neq 1, w \in \{a, b\}^*\}$.

Let p be the pumping length.

For every string s such that $|s| \geq p$, we show that $s = xyz$ satisfying,

1. $|y| \neq 0$,
2. $|xy| \leq p$, and
3. For all i , $xy^i z \in L$.

Example of nonregular language that satisfies the pumping lemma

Let $\Sigma = \{\$\text{, } a, b\}$.

Consider the language $L = \{\$a^n b^n | n \geq 1\} \cup \{\$^k w | k \neq 1, w \in \{a, b\}^*\}$.

Let p be the pumping length.

For every string s such that $|s| \geq p$, we show that $s = xyz$ satisfying,

1. $|y| \neq 0$,
2. $|xy| \leq p$, and
3. For all i , $xy^i z \in L$.

There are two cases. The string s might be of the form $\$a^n b^n$ or of the form $\$^k w$ where $k \neq 1$.

For all cases, except when $k = 0$, consider $y = \$$. This satisfies all three conditions.

For the case when $s \in \{\$^k w | k = 0, w \in \Sigma^*\}$. Then s can be any string from $\{a, b\}^*$.

And, y can be any nonempty substring of s . This satisfies all three conditions.

Example of nonregular language that satisfies the pumping lemma

- $L = \{\$\overline{a^n}b^n | n \geq 1\} \cup \{\$^k w | k \neq 1, w \in \{a, b\}^*\}$.

Now, how is that we show L is nonregular.

This is through proof by contradiction.

Assume that L is regular.

We know $\$a^*b^*$ is regular. (why?)

Since regular languages are closed under intersection. Intersection of L and $\$a^*b^*$ must be regular.

But, Intersection of L and $\$a^*b^*$ is $\{\$a^n b^n | n \geq 0\}$, and this is nonregular (why?).

Hence, the contradiction.

Yet another language that is nonregular,
but for which PL is satisfied.

- This language is very similar to the previous one.
- $L = \{a^i b^j c^k | (i = 1) \Rightarrow (j = k)\}$.

An Important Other way of showing that a language is nonregular

- By using Myhill-Nerode Theorem
 - DFA or NFA for a regular language must have finite number of states.
 - If you show that infinite number of states are needed, then it is equivalent to showing that the language is nonregular.
 - Apart from this, Myhill-Nerode theorem has one important application, viz., minimization of a DFA.

Myhill-Nerode Theorem is much more than the Pumping Lemma

- Myhill-Nerode theorem can be used to show that a language is regular also. Of course it can be used to show that a language is nonregular.
 - This gives a necessary and sufficient condition for a language being regular.
- Note, the pumping lemma, on the otherhand can be used only to show that a language is nonregular.
 - Pumping lemma should not be used to show that a language is regular.

1.29 Use the pumping lemma to show that the following languages are not regular.

a. $A_1 = \{0^n 1^n 2^n \mid n \geq 0\}$

b. $A_2 = \{www \mid w \in \{a, b\}^*\}$

c. $A_3 = \{a^{2^n} \mid n \geq 0\}$ (Here, a^{2^n} means a string of 2^n a's.)

- **Reading Assignment – From Sipser’s book**

1.30 Describe the error in the following “proof” that $0^* 1^*$ is not a regular language. (An error must exist because $0^* 1^*$ *is* regular.) The proof is by contradiction. Assume that $0^* 1^*$ is regular. Let p be the pumping length for $0^* 1^*$ given by the pumping lemma. Choose s to be the string $0^p 1^p$. You know that s is a member of $0^* 1^*$, but Example 1.73 shows that s cannot be pumped. Thus you have a contradiction. So $0^* 1^*$ is not regular.

CPS 220 – Theory of Computation

Non-regular Languages

Warm up Problem

Problem #1.48 (p.90)

Let $\Sigma = \{0,1\}$ and let

$D = \{w \mid w \text{ contains an equal number of occurrences of the substrings } 01 \text{ and } 10\}$.

Thus $101 \in D$ because 101 contains a single 01 and a single 10 , but $1010 \notin D$ because 1010 contains two 10 s and only one 01 . Show that D is a regular language.

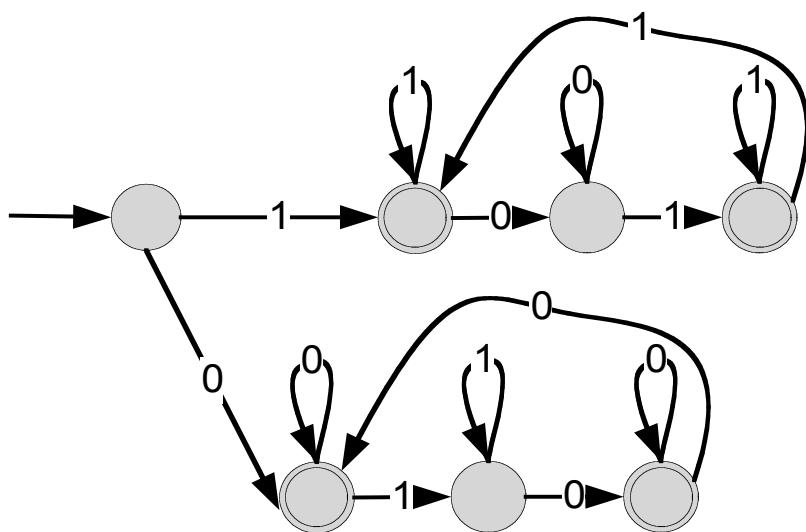
Solution:

This language is regular because it can be described by a regular expression and a FA (NFA):

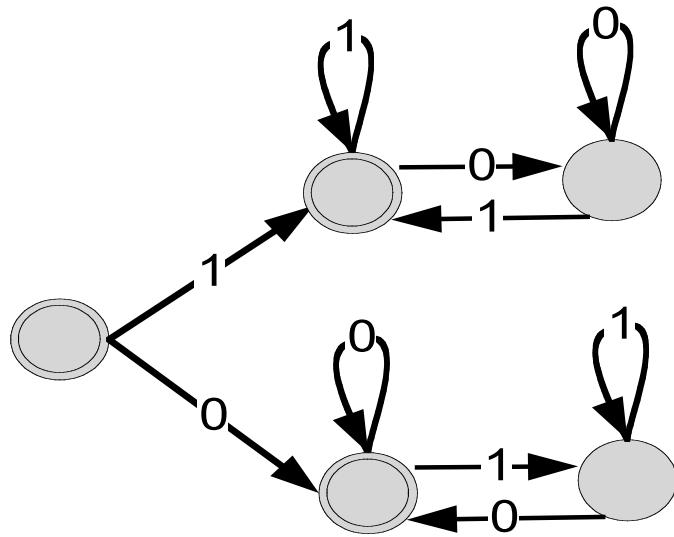
Regular Expression

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

NFA



DFA



Proving that a language is not regular—the Pumping Lemma

Consider the language $B = \{ 0^n 1^n \mid n \geq 0 \}$. Is B regular?

If B is regular, then there is a DFA M recognizing B . That means, M accepts the string $0^{2003} 1^{2003}$,

but rejects the string $0^{2003} 1^{1999}$. How can M achieve that? As it reads the input, it has to remember

how many 0s it encountered so far. Then, when it starts reading 1s, it has to count the 1s and match

them with the number of 0s. However, a DFA by definition is finite, i.e., it has limited memory, which

is just the current state in which it is. To count it would require enough bits of memory to store a number...

How can we prove that a language is not regular? We will now demonstrate a method of doing that.

+++++
+++++

Theorem. The Pumping lemma. If A is a regular language, then there is a number p (the pumping length)

where, if s is any string in A of length at least p , then s may be divided into three pieces, $s = xyz$, satisfying

the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$
2. $|y| > 0$
3. $|xy| \leq p$

Note: either x or z may be ϵ

+++++
+++++

So if s is long enough, there is a nonempty string y within s , which can be “pumped”.

=====

In other words: If a language A is accepted by a DFA M with q states, then every string s in A

with $|s| \geq q$ can be written as $s = xyz$ such that $y \neq \epsilon$ and $xy^* z \subseteq A$

=====

Proof. If A is regular, then there is a DFA M that accepts A .

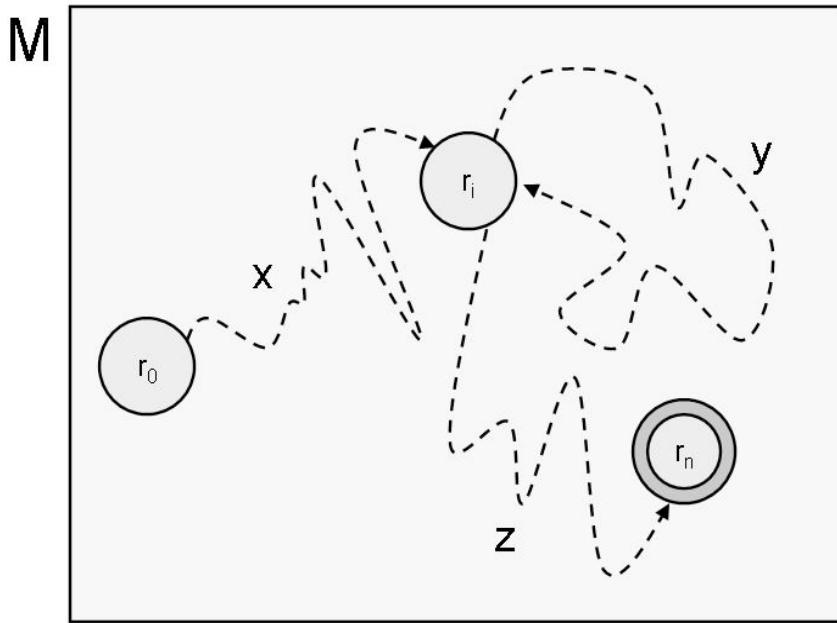
Say that M has p states.

Let s be a string of length $n \geq p$. Let $r_1 \dots r_{n+1}$ be the sequence of states of M , when processing s .

Since $n+1 > p$, at least one of M 's states appears twice: $r_i = r_j$ for $j \neq i$.

Let $x = s_1 \dots s_i$, $y = s_{i+1} \dots s_j$, $z = s_{j+1} \dots s_n$. Then, $xyz = s_1 \dots s_n = s$, and $|y| > 0$.

Here is why $xy^i z$ is accepted by M :



How can we maintain this condition - $|xy| \leq p$? We can make sure that this is true, by selecting the *smallest* i and j such that $r_i = r_j$.

The first $p+1$ states in the sequence must contain a repetition - therefore $|xy| \leq p$.

Pigeonhole principle - if p pigeons are placed into fewer than p holes, some hole has to have more than one pigeon in it.

Example of using the Pumping Lemma

Theorem. $B = \{ 0^n 1^n \mid n \geq 0 \}$ is not regular.

Proof. Proof by contradiction.

Assume that language B is regular. [If B is regular, then there exists a constant p (the pumping length) such that the conditions of the pumping lemma hold.] Let p be the pumping length and let's choose a string s that fits the conditions for our pumping lemma. Let $s = 0^p 1^p$. Because $s \in B$ and $|s| \geq p$ - then s can be broken into xyz where for any $i \geq 0$ the string $xy^i z \in B$.

Must consider 3 cases:

Case 1:

The substring y consists of only 0s. In this case the string $xyyz$ has more 0s than 1s and therefore

$xxyz \notin B$. This case is a contradiction.

Case 2:

The string y consists of only 1s. For the same basic reason, this case is a contradiction.

Case 3:

The string y consists of both 0s and 1s. In this case the string xyyz may have the same number of 1s and 0s - however it violates the basic structure of the language - where all 0 come before all 1s. This case is a contradiction

Thus a contradiction is unavoidable.

Definition. The complement of a language A , $\overline{A} = \Sigma^* - A$, is all strings except those in A .

Theorem. Regular languages are closed under complementation.

Proof. Let A be a regular language. We will show that \overline{A} is regular.

Since A is regular, a DFA M accepts it.

By turning all the accept states of M into non-accept, and all non-accept states into accept, every input in A ends up in a non-accept state, and every input not in A ends up in an accept state. Therefore, the modified M machine accepts \overline{A} .

Theorem. Regular languages are closed under intersection, \cap .

Proof. Recall that if A and B are two sets, $A \cap B$ is the set of all the common elements.

A simple logic fact that you can prove as an exercise is that:

$$A \cap B = \overline{\overline{A} \cup \overline{B}}$$

Then, the result follows by closure of regular languages under complement and union.

Theorem. $A = \{ w \mid w \text{ has an equal number of 0s and 1s} \}$ is not regular.

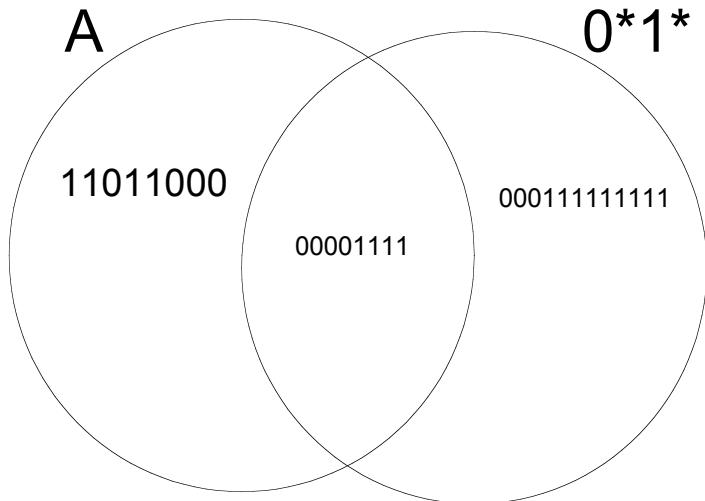
Proof. Assume that A is regular, in order to arrive at a contradiction. (Very cool proof)

We know that 0^*1^* is regular.

Therefore $A \cap (0^*1^*)$ is regular, by closure of regular languages under \cap .

However, $A \cap (0^*1^*) = \{ 0^n1^n \mid n \geq 0 \}$, which we proved not to be regular.

Therefore A cannot be regular.



Another pumping lemma proof:

Theorem. $A = \{ 0^p \mid p \text{ is a prime} \}$ is not a regular language.

Proof. Assume that A is regular, in order to arrive at a contradiction.

Then A is accepted by a DFA M . Let s be the number of states in M . Consider a prime number $p > s$.

Note that $0^p \in A$ and $|0^p|=p > s$. Therefore, by the pumping lemma, 0^p can be written as $0^p = xyz$ such that $|y| > 0$ and $xy^i z \in A$

Let $i = |x|+|z|$ and $j = |y|$. Then, the condition $xy^*z \in A$ means that, for any $k \geq 0$, $i+kj$ is a prime.

In particular, when $k = 0$ it means that i is a prime. So $i \geq 2$. When $k = i$, this means that $i(1+j)$ is a prime.

However, since $|y| > 0$ (not the empty string), we have $j = |y| \geq 1$ and so $i(1+j)$ is not prime. This is a contradiction.

References:

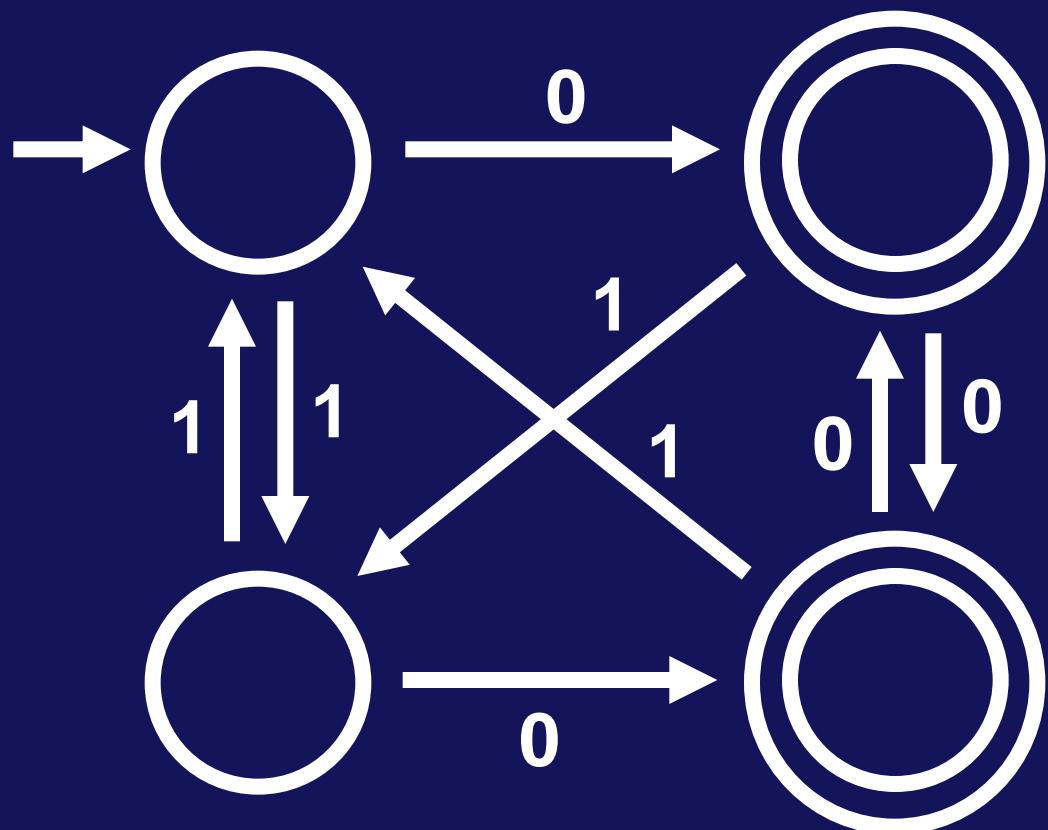
Introduction to the Theory of Computation (2nd ed.) Michael Sipser

Problem Solving in Automata, Languages, and Complexity Ding-Zhu Du and Ker-I Ko

MINIMIZING DFAs

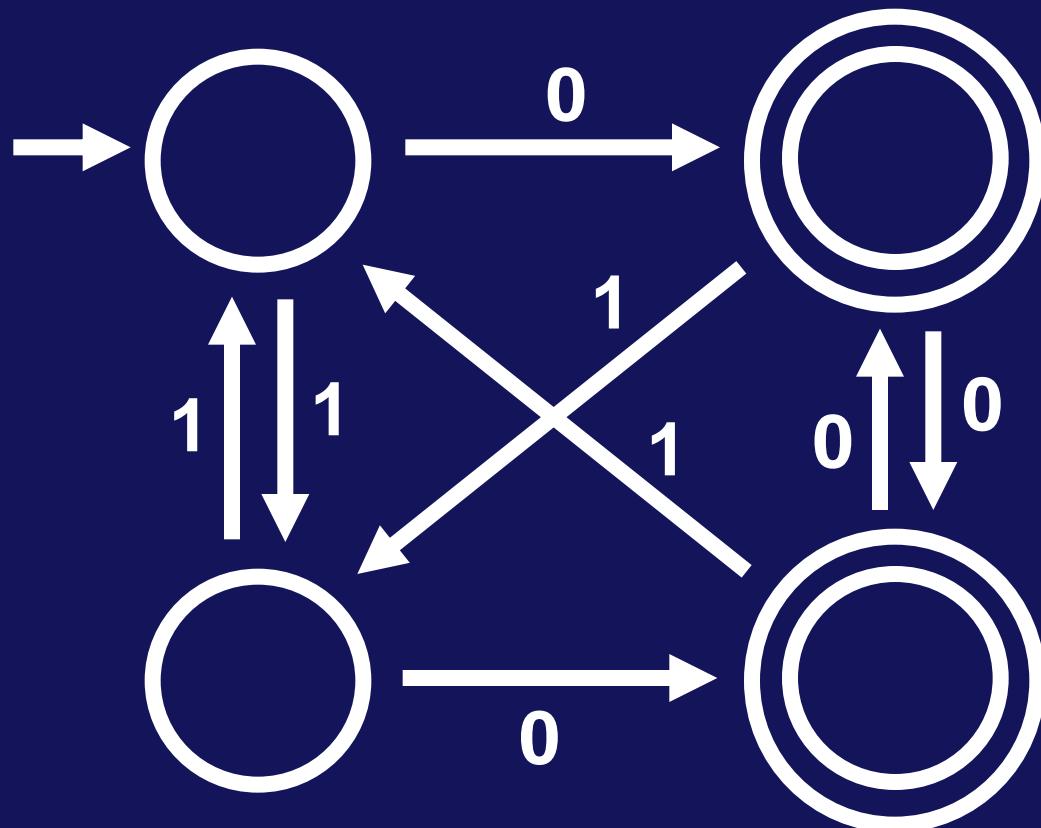
To have minimum number of states

IS THIS MINIMAL?

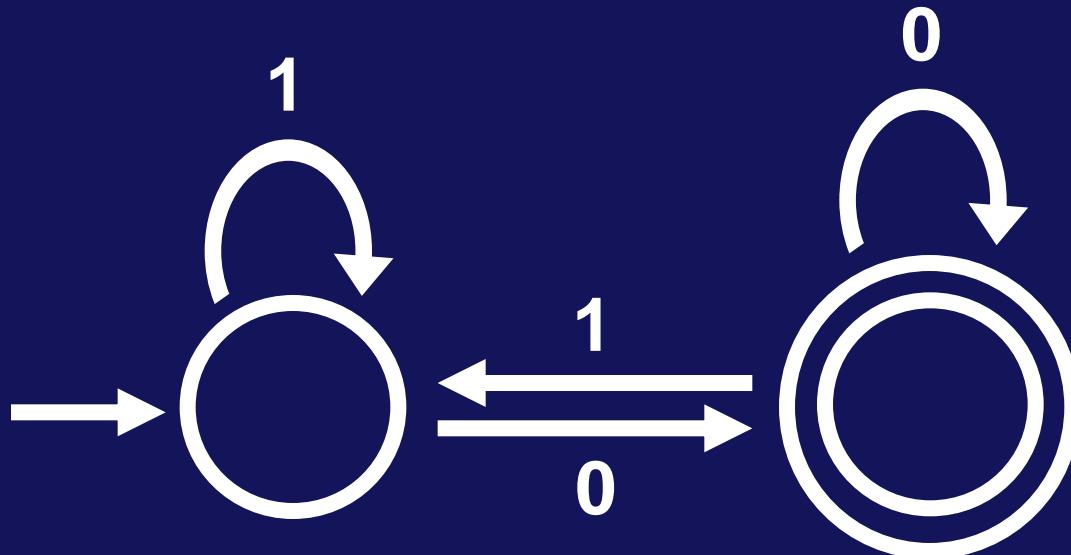


IS THIS MINIMAL?

NO



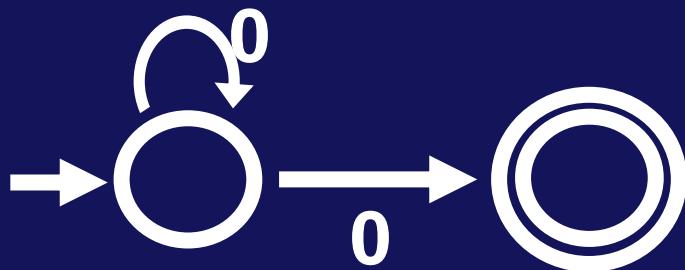
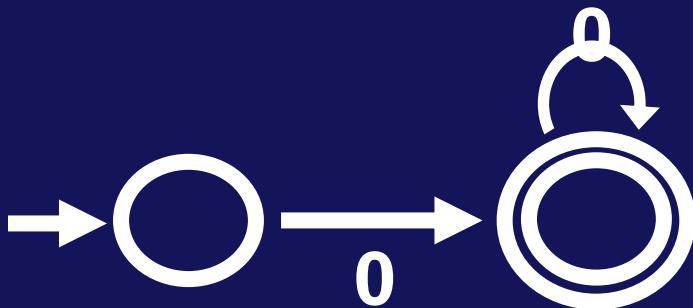
IS THIS MINIMAL?



THEOREM

For every regular language L , there exists
a **unique** (up to re-labeling of the states)
minimal DFA M such that $L = L(M)$

NOT TRUE FOR NFAs



Because of this, minimization of NFA is complicated and is out of scope of current ToC course.

EXTENDING δ

Given DFA $M = (Q, \Sigma, \delta, q_0, F)$ extend δ

to $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ as follows:

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

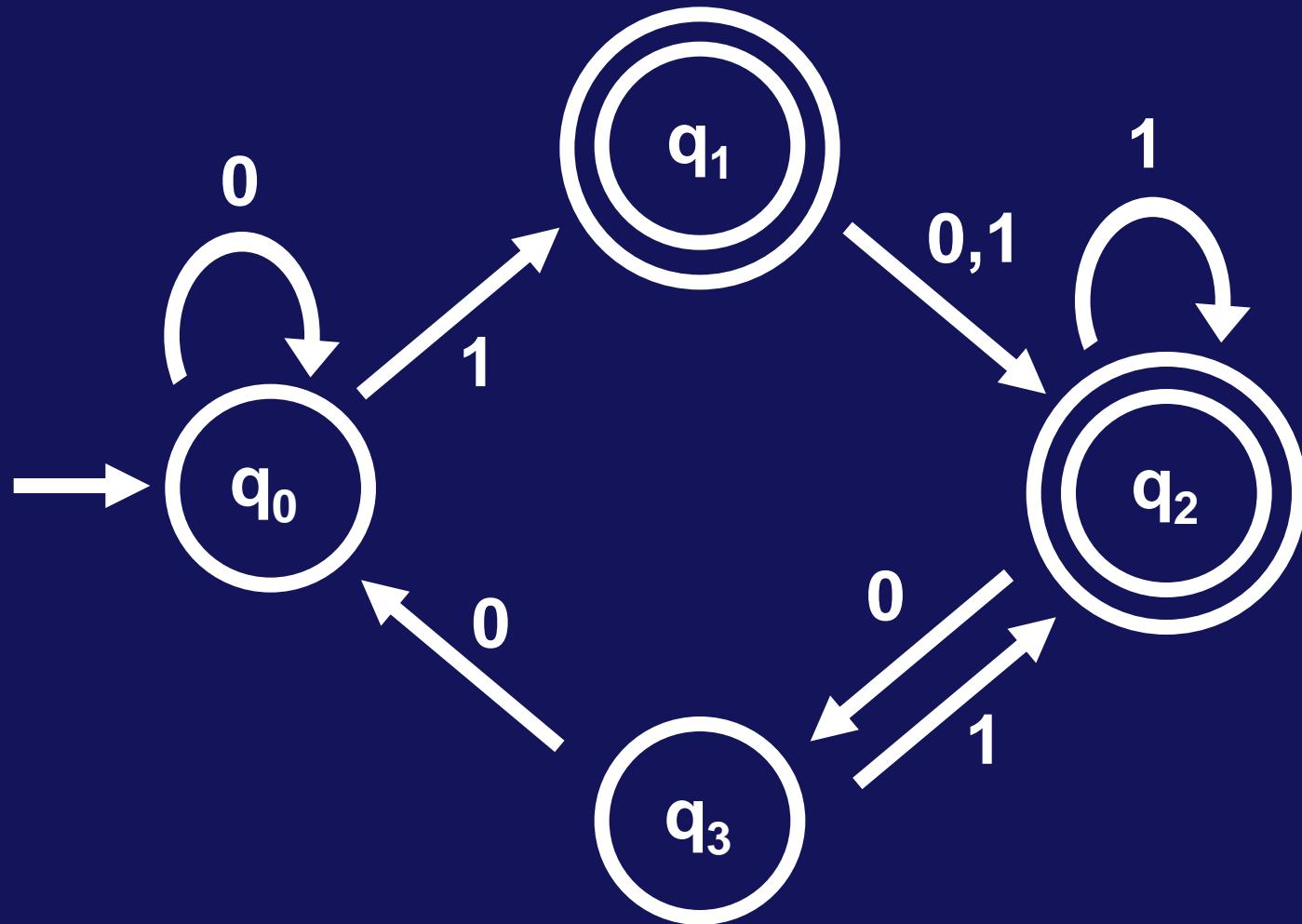
$$\hat{\delta}(q, a) = \delta(q, a) \text{ where } a \in \Sigma$$

$$\hat{\delta}(q, w_1 \dots w_{k+1}) = \delta(\hat{\delta}(q, w_1 \dots w_k), w_{k+1})$$

Note: in $\delta(q, a)$, a is a string. Context should clear this.

A string $w \in \Sigma^*$ **distinguishes states** q_1 from q_2 if

$$\widehat{\delta}(q_1, w) \in F \Leftrightarrow \widehat{\delta}(q_2, w) \notin F$$



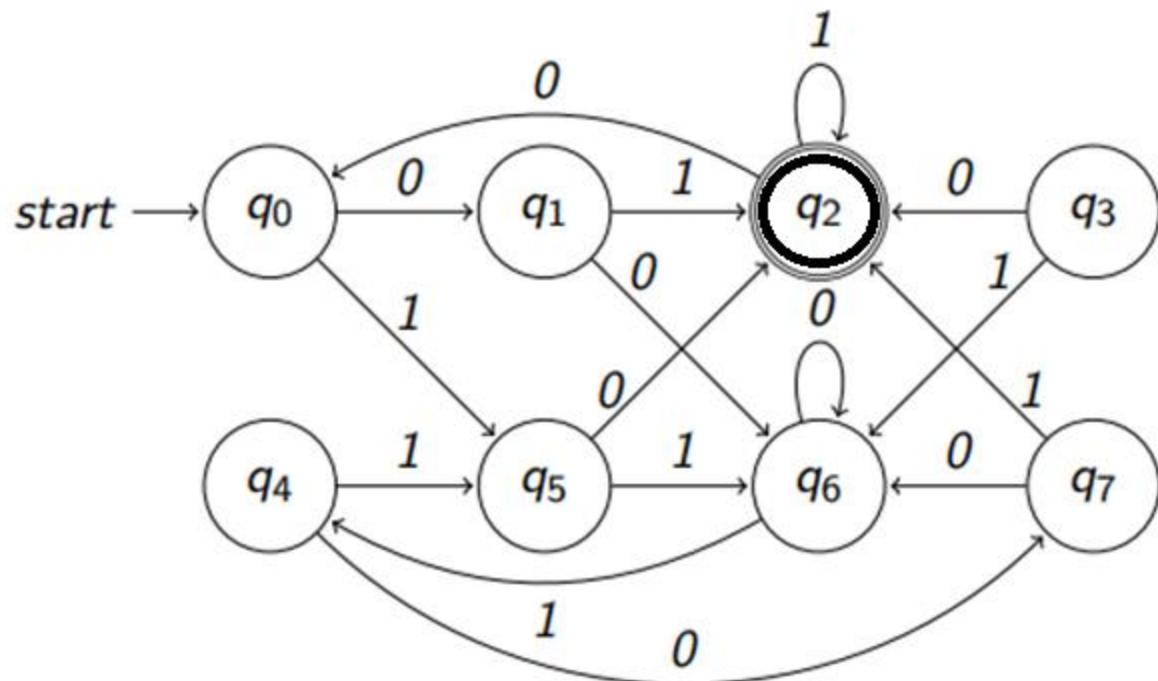
ϵ distinguishes accept from non-accept states

Let $M = (Q, \Sigma, \delta, q_0, F)$ and let $p, q, r \in Q$

Definition :

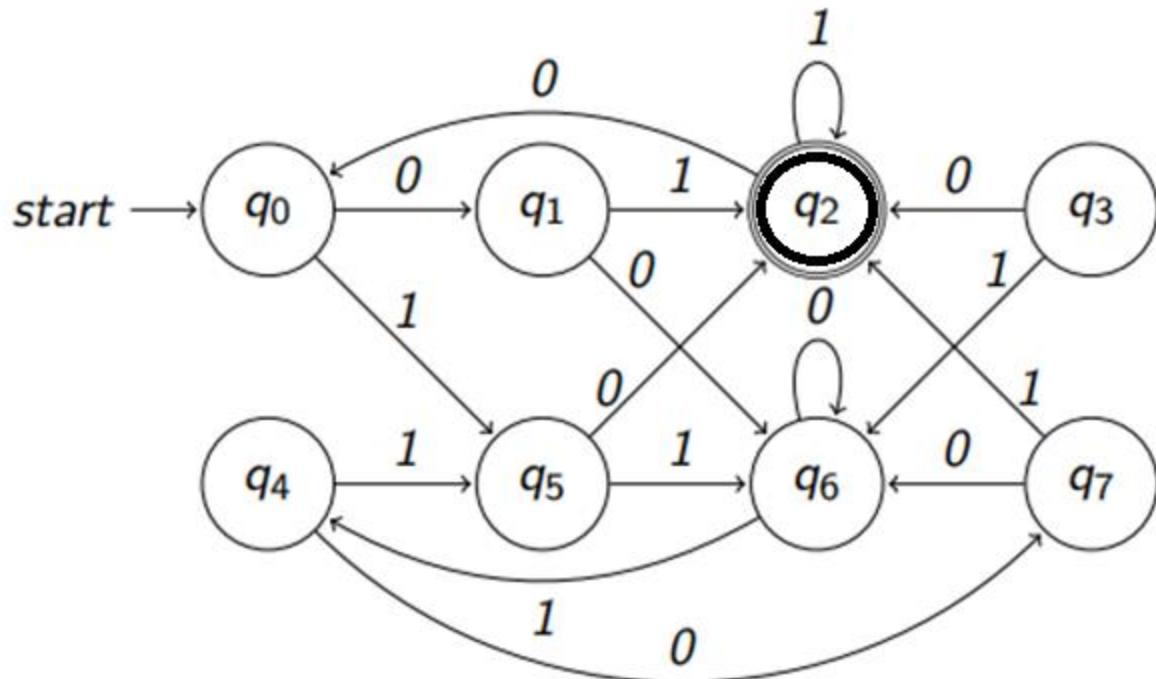
- (1) p is **equivalent** to q iff there is *no* $w \in \Sigma^*$ that distinguishes p and q ,
- (2) Otherwise p is **not equivalent** to q . In this case, we say p and q are **distinguishable**.

Example: distinguishable states



- ▶ ϵ distinguishes q_2 and q_6 .
- ▶ 01 distinguishes q_0 and q_6 .

Example: distinguishable states



- ▶ ϵ distinguishes q_2 and q_6 .
- ▶ 01 distinguishes q_0 and q_6 .

Exercise

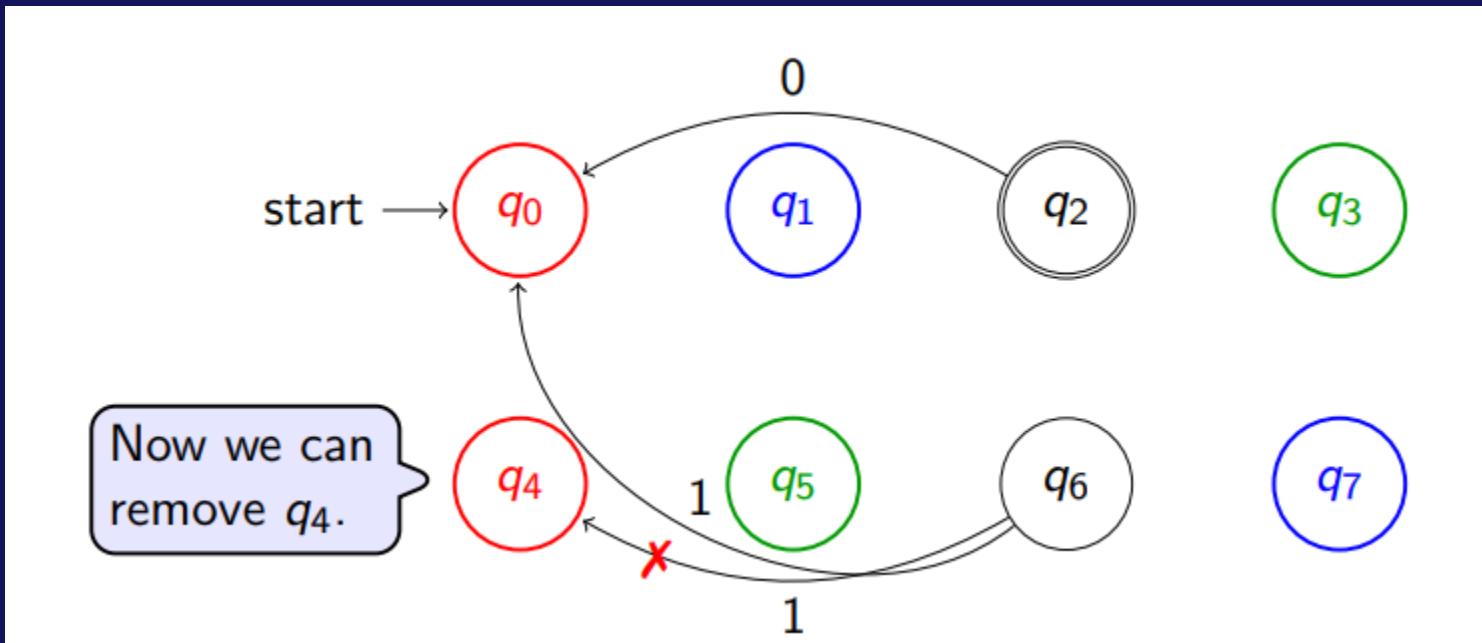
Give strings that distinguishes the following pair of states.

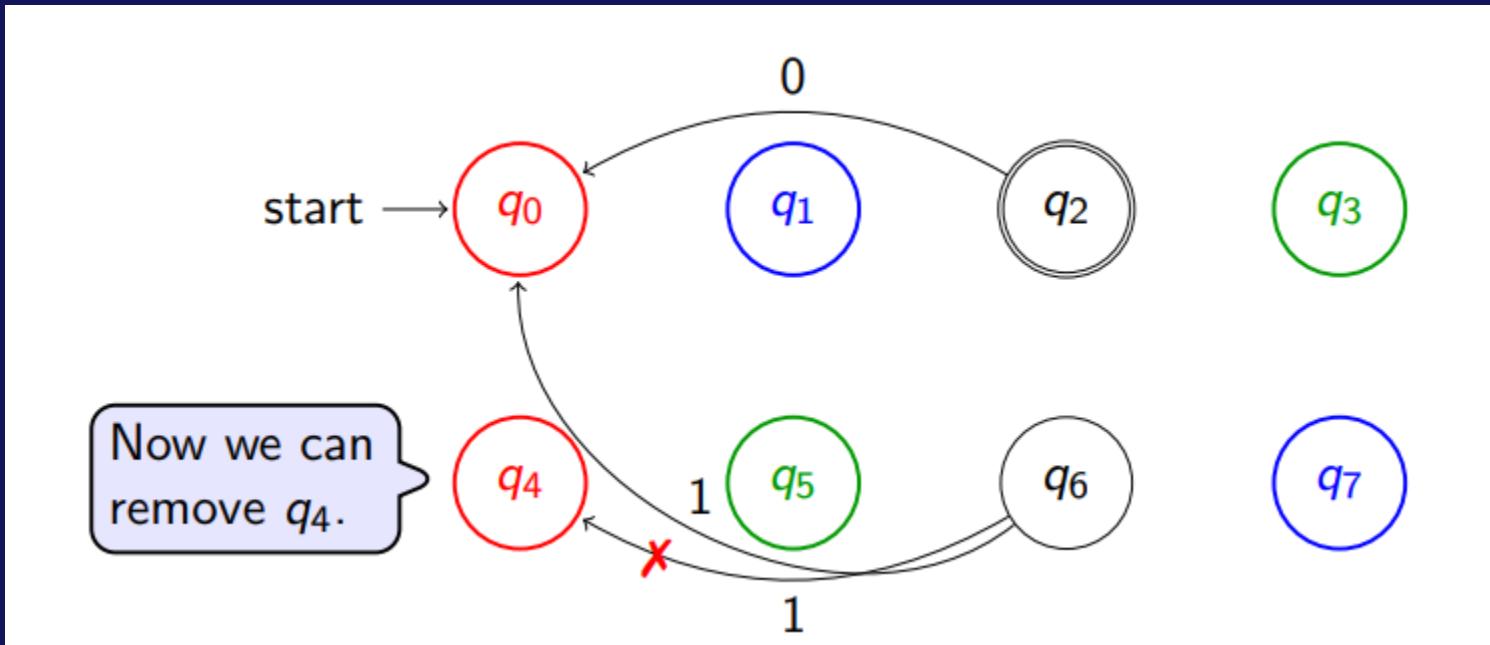
- ▶ q_1 and q_5
- ▶ q_2 and q_7
- ▶ q_4 and q_3
- ▶ q_1 and q_7

DFA Minimization, intuition

- We can remove unreachable states (**why** and **how to find unreachable states?**)
- If states q_0 and q_4 are equivalent, then, we can move all incoming transitions (arrows) from q_4 to q_0
- Because of this q_4 becomes unreachable, hence can be removed.

Let q_0 and q_4 are equivalent





- But, how is that you know
 q_0 and q_4 are equivalent?

Let $M = (Q, \Sigma, \delta, q_0, F)$ and let $p, q, r \in Q$

Define the relation “ \sim ”:

$p \sim q$ iff p is equivalent to q

$p \not\sim q$ iff p is distinguishable from q

Proposition: “ \sim ” is an equivalence relation

$p \sim p$ (reflexive)

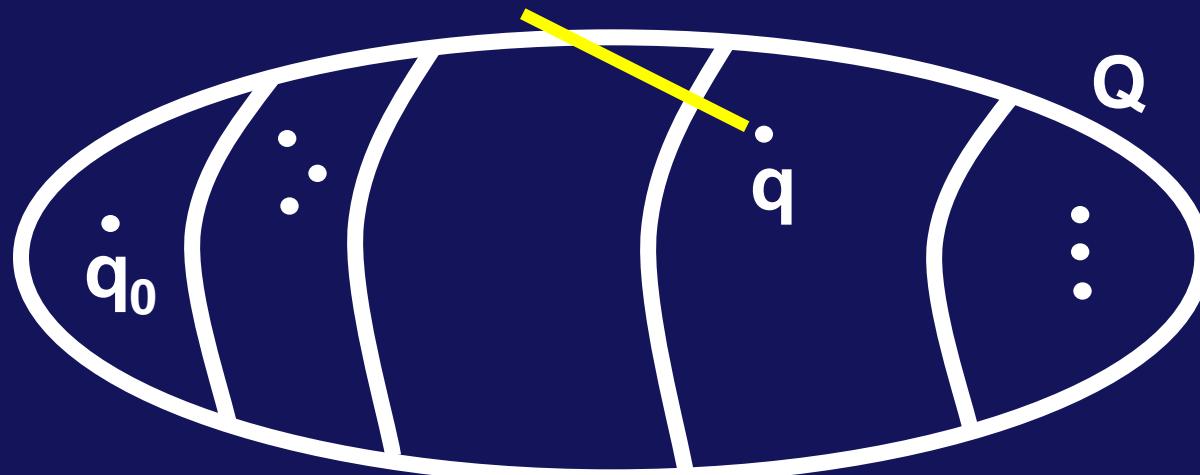
$p \sim q \Rightarrow q \sim p$ (symmetric)

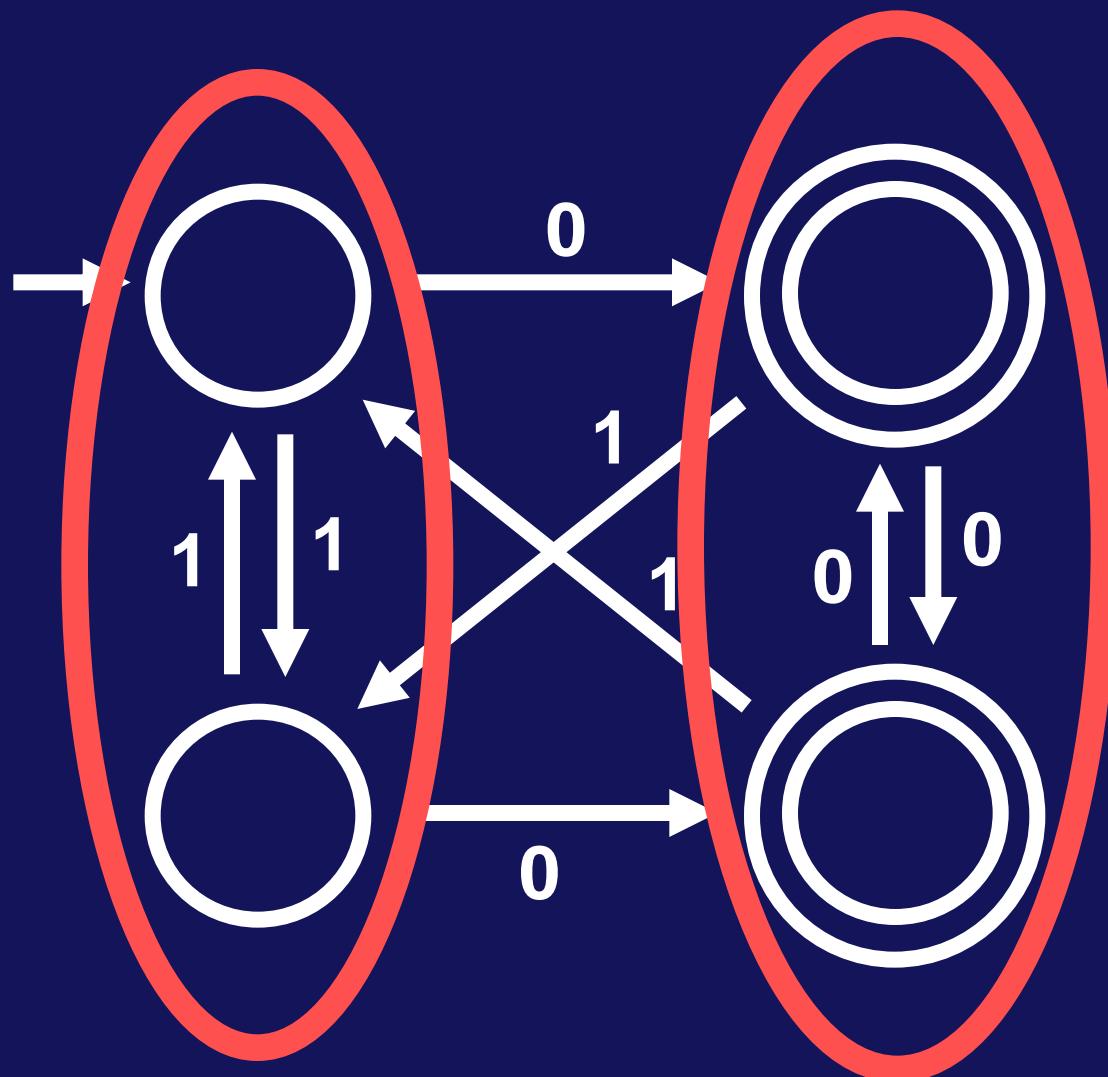
$p \sim q$ and $q \sim r \Rightarrow p \sim r$ (transitive)

Let $M = (Q, \Sigma, \delta, q_0, F)$ and let $p, q, r \in Q$

Proposition: “ \sim ” is an **equivalence relation**
so “ \sim ” partitions the set of states of M into
disjoint equivalence classes

$$[q] = \{ p \mid p \sim q \}$$





Algorithm MINIMIZE(DFA M)

Input: DFA M

Output: DFA M_{MIN} such that:

$$M \equiv M_{MIN}$$

M_{MIN} has no inaccessible states

M_{MIN} is irreducible

||

states of M_{MIN} are pairwise distinguishable

Theorem: M_{MIN} is the unique minimum

Algorithm MINIMIZE(DFA M)

(1) Remove all inaccessible states from M

(2) Apply Table-Filling algorithm to get
 $E_M = \{ [q] \mid q \text{ is an accessible state of } M \}$

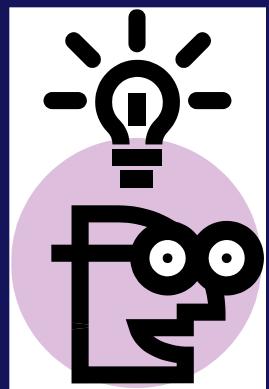
$$M_{MIN} = (Q_{MIN}, \Sigma, \delta_{MIN}, q_{0\ MIN}, F_{MIN})$$

$$Q_{MIN} = E_M, \quad q_{0\ MIN} = [q_0], \quad F_{MIN} = \{ [q] \mid q \in F \}$$

$$\delta_{MIN}([q], a) = [\delta(q, a)]$$

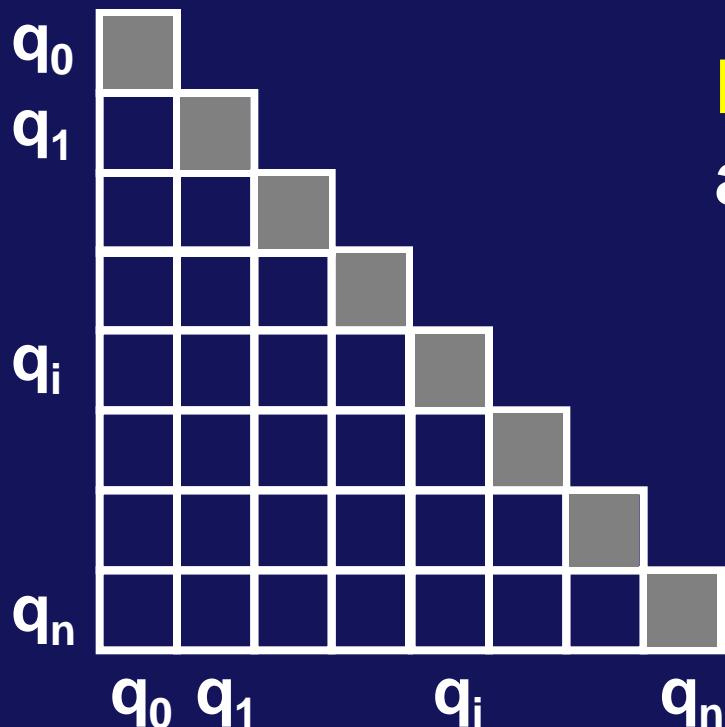
TABLE-FILLING ALGORITHM

IDEA!



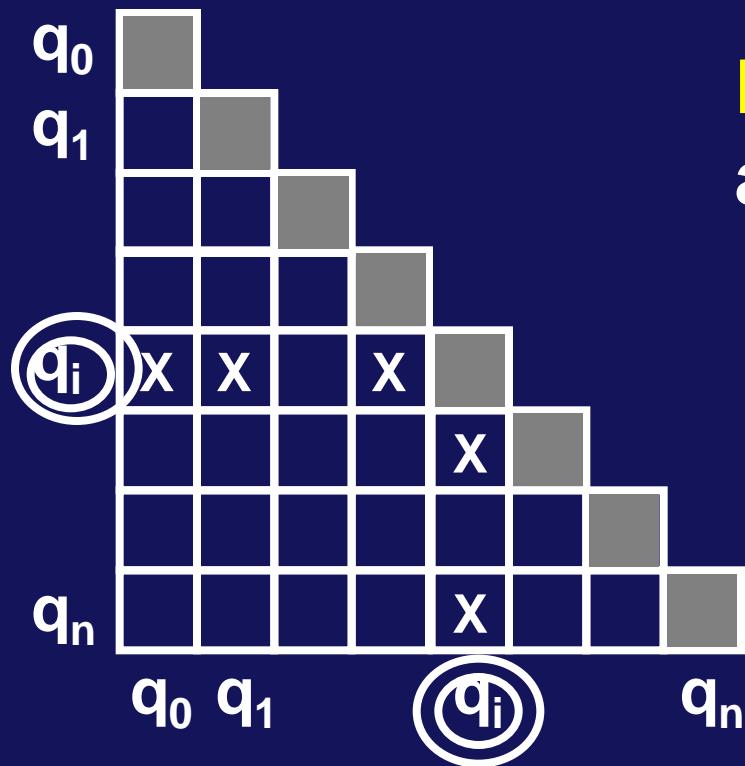
- Make best effort to find pairs of states that are distinguishable.
- Pairs leftover will help us.

TABLE-FILLING ALGORITHM



Base Case: p accepts
and q rejects $\Rightarrow p \neq q$

TABLE-FILLING ALGORITHM

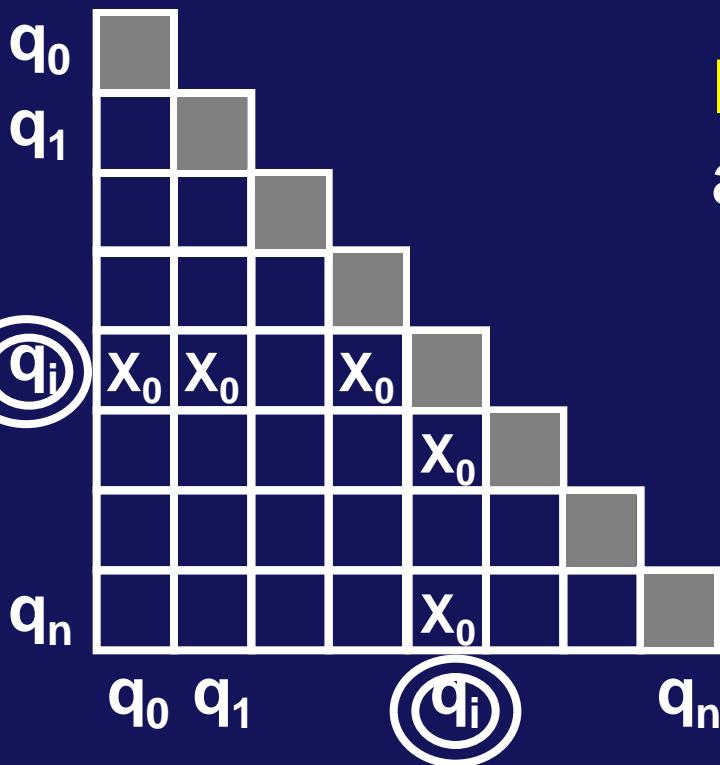


Base Case: p accepts
and q rejects $\Rightarrow p \neq q$

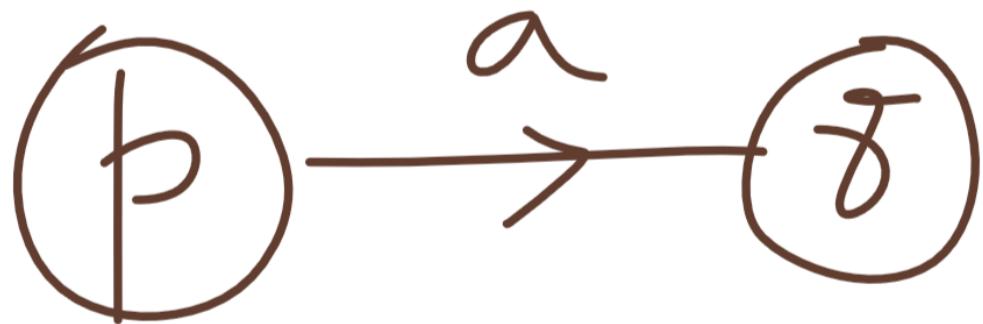
TABLE-FILLING ALGORITHM

For base case, we put X_0 in the corresponding cell.

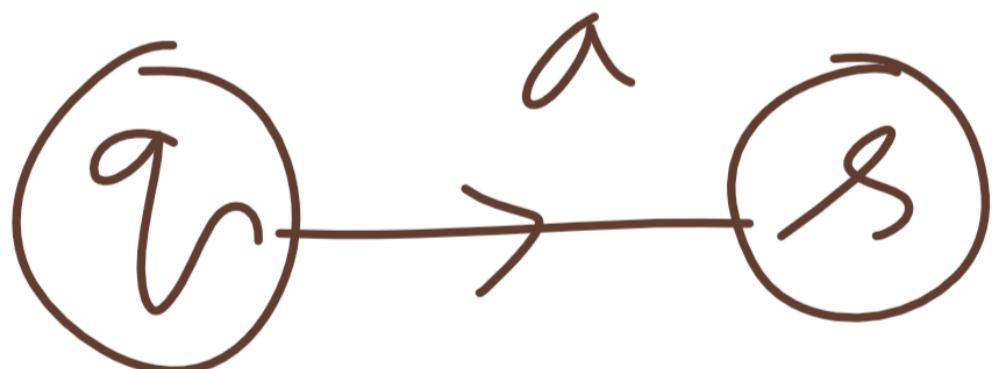
This states, that the two states are not equivalent.



**Base Case: p accepts
and q rejects $\Rightarrow p \neq q$**



$p \vdash q$



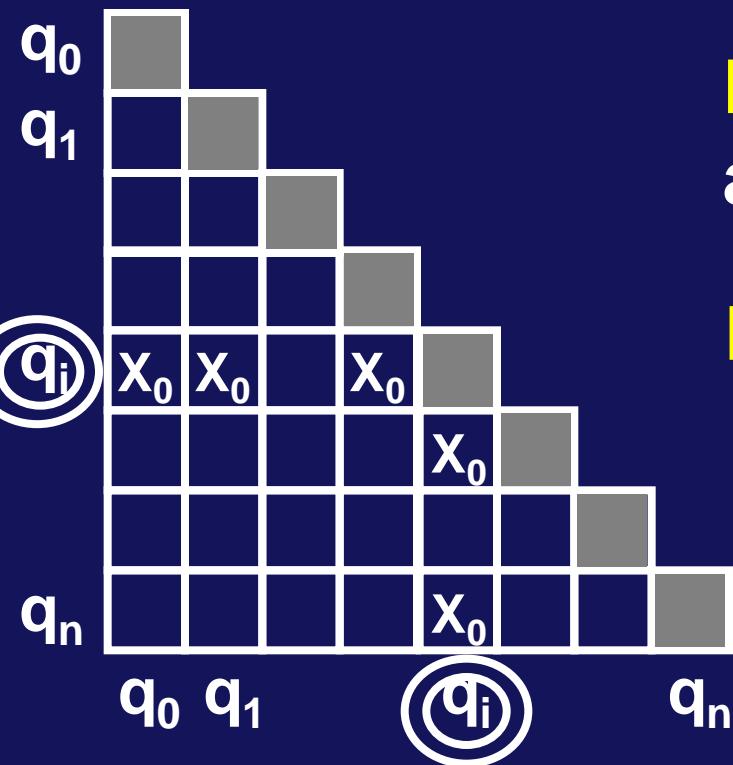
\Rightarrow

$p \vdash r$

TABLE-FILLING ALGORITHM

For base case, we put X_0 in the corresponding cell.

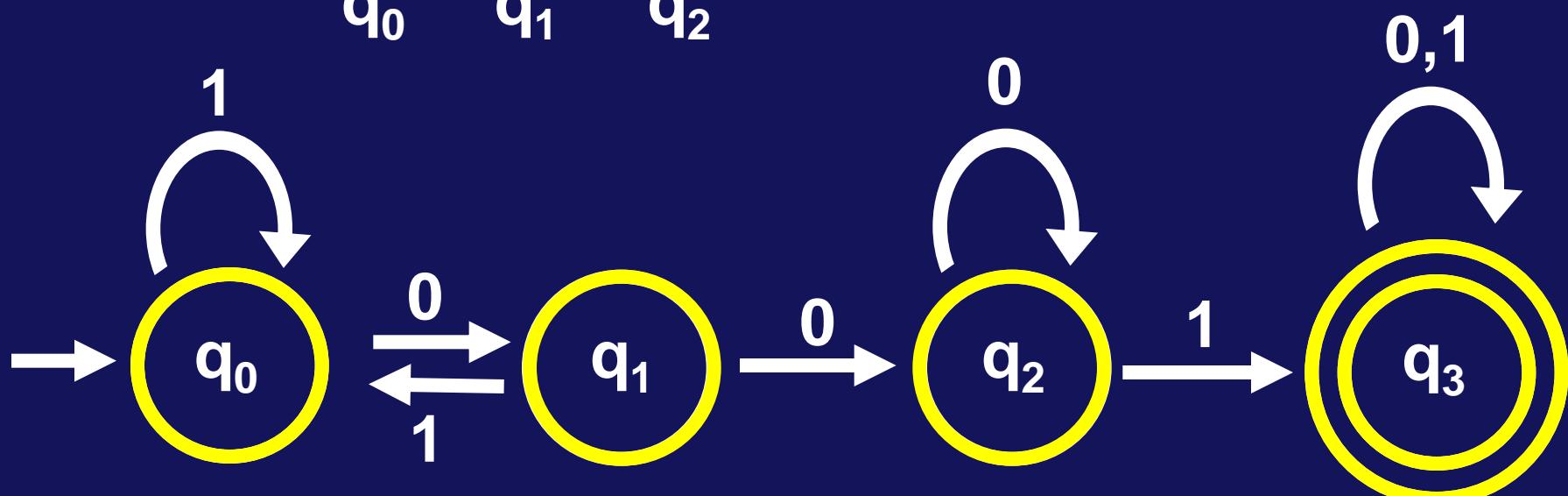
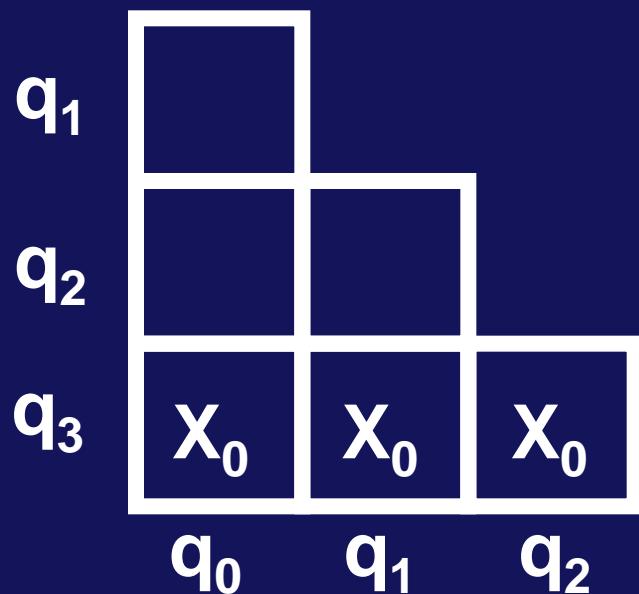
This states, that the two states are not equivalent.



Base Case: p accepts
and q rejects $\Rightarrow p \neq q$

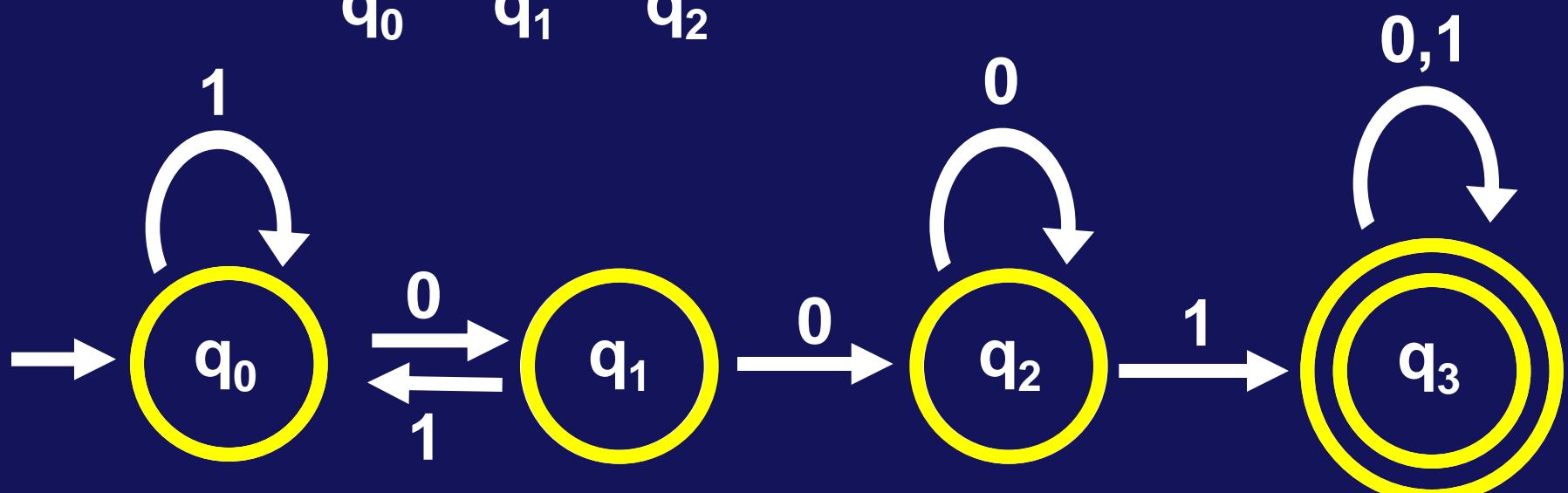
Recursion:

$$\begin{array}{c} p \xrightarrow{a} r \\ q \xrightarrow{a} s \end{array} \quad \nmid \quad \Rightarrow \quad p \nmid q$$

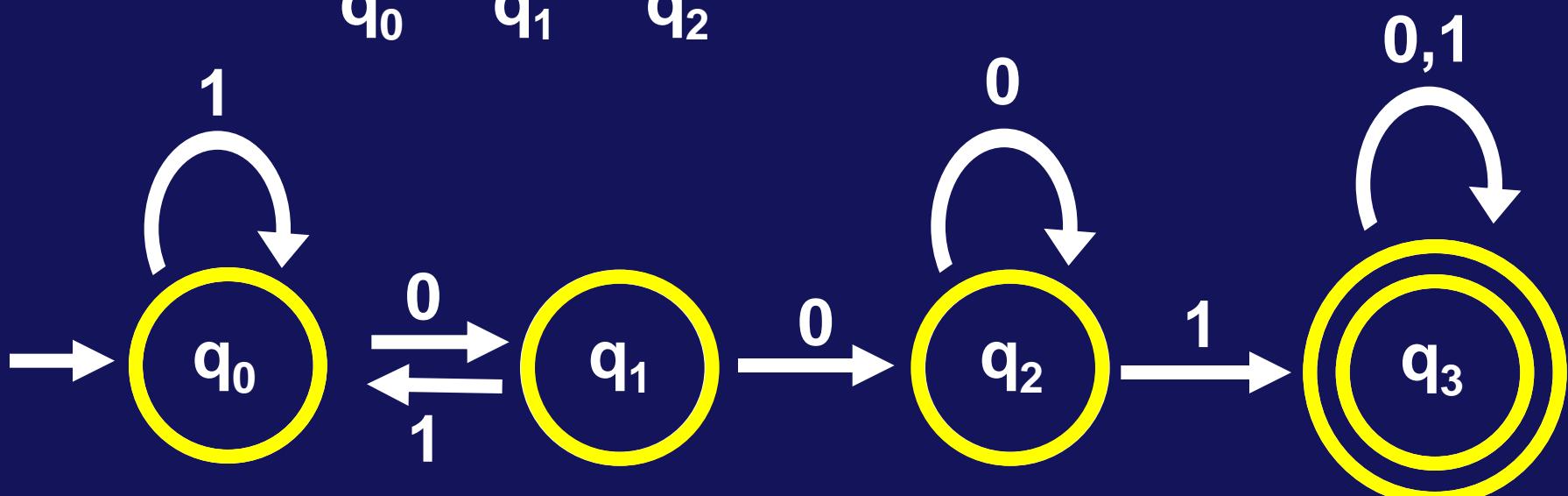


q_1		
q_2	X_1	X_1
q_3	X_0	X_0

$q_0 \quad q_1 \quad q_2$



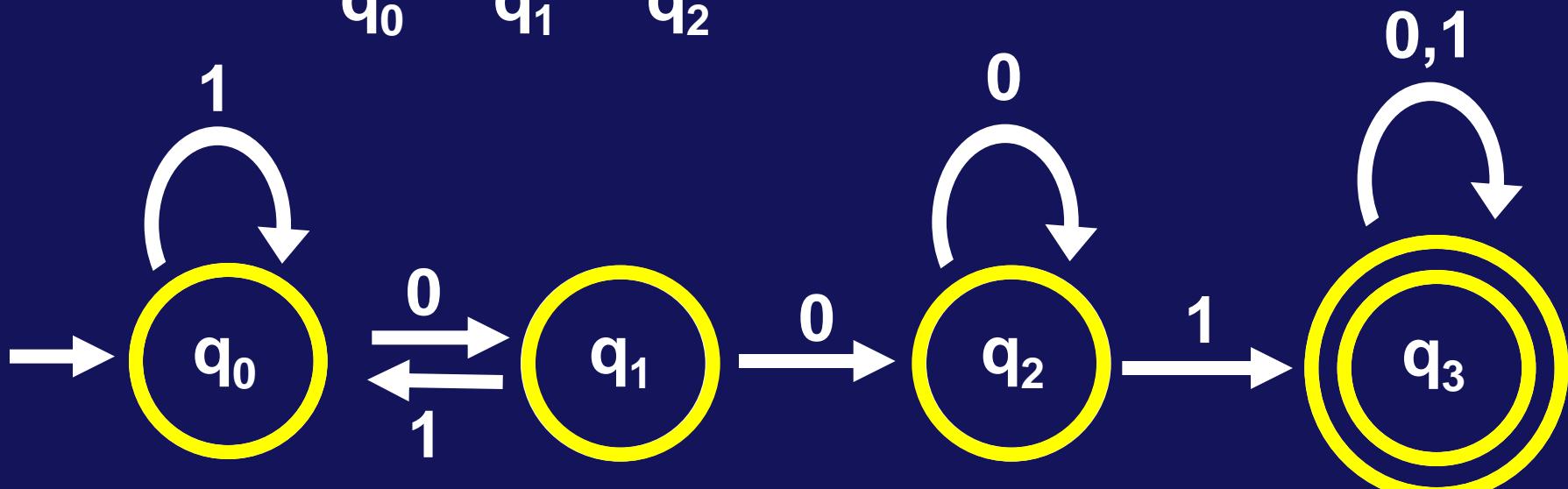
	X_2	
q_1		
q_2	X_1	X_1
q_3	X_0	X_0
	X_0	

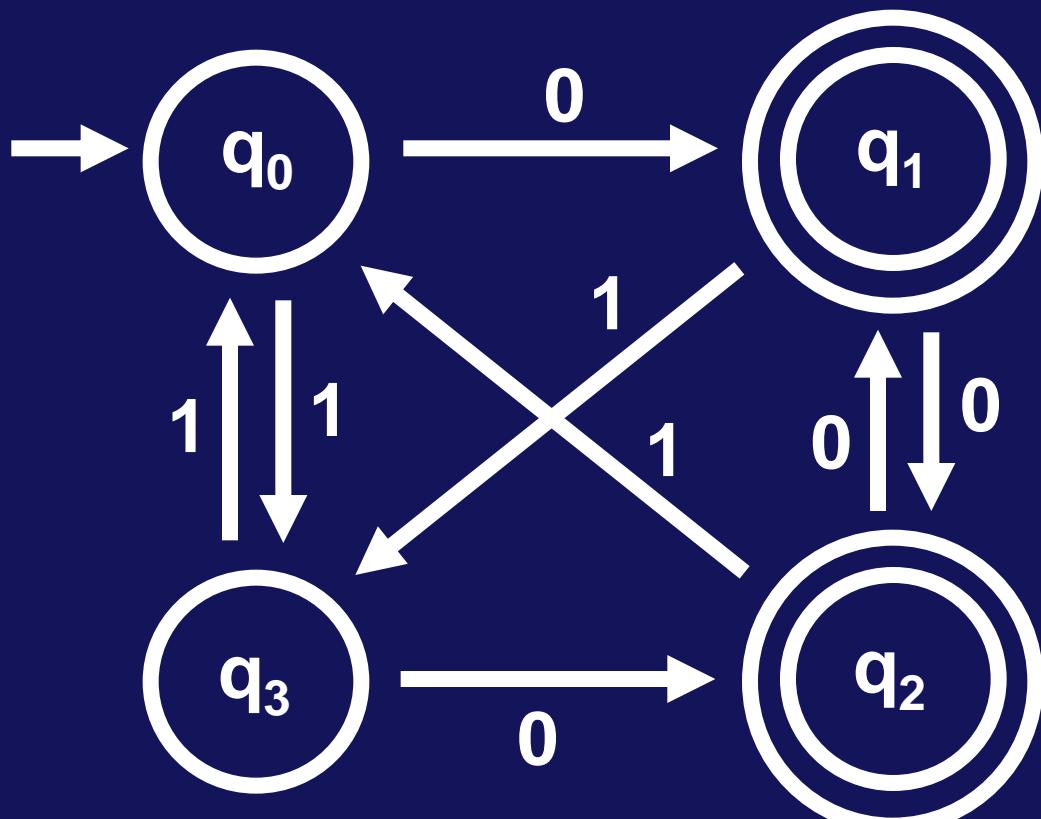
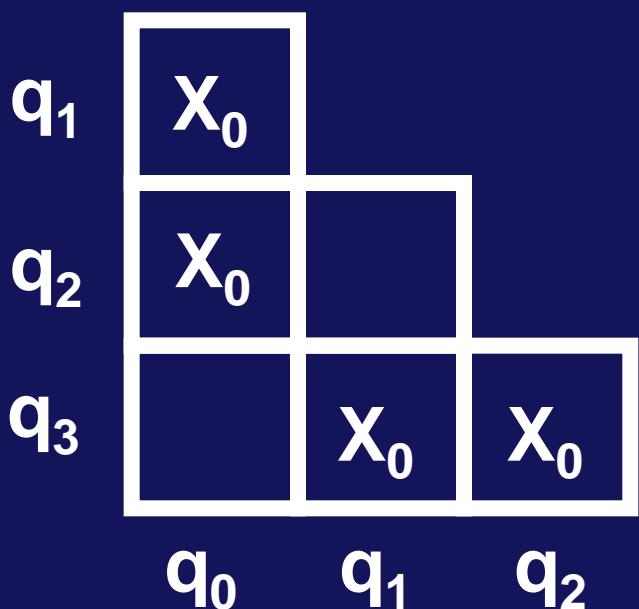


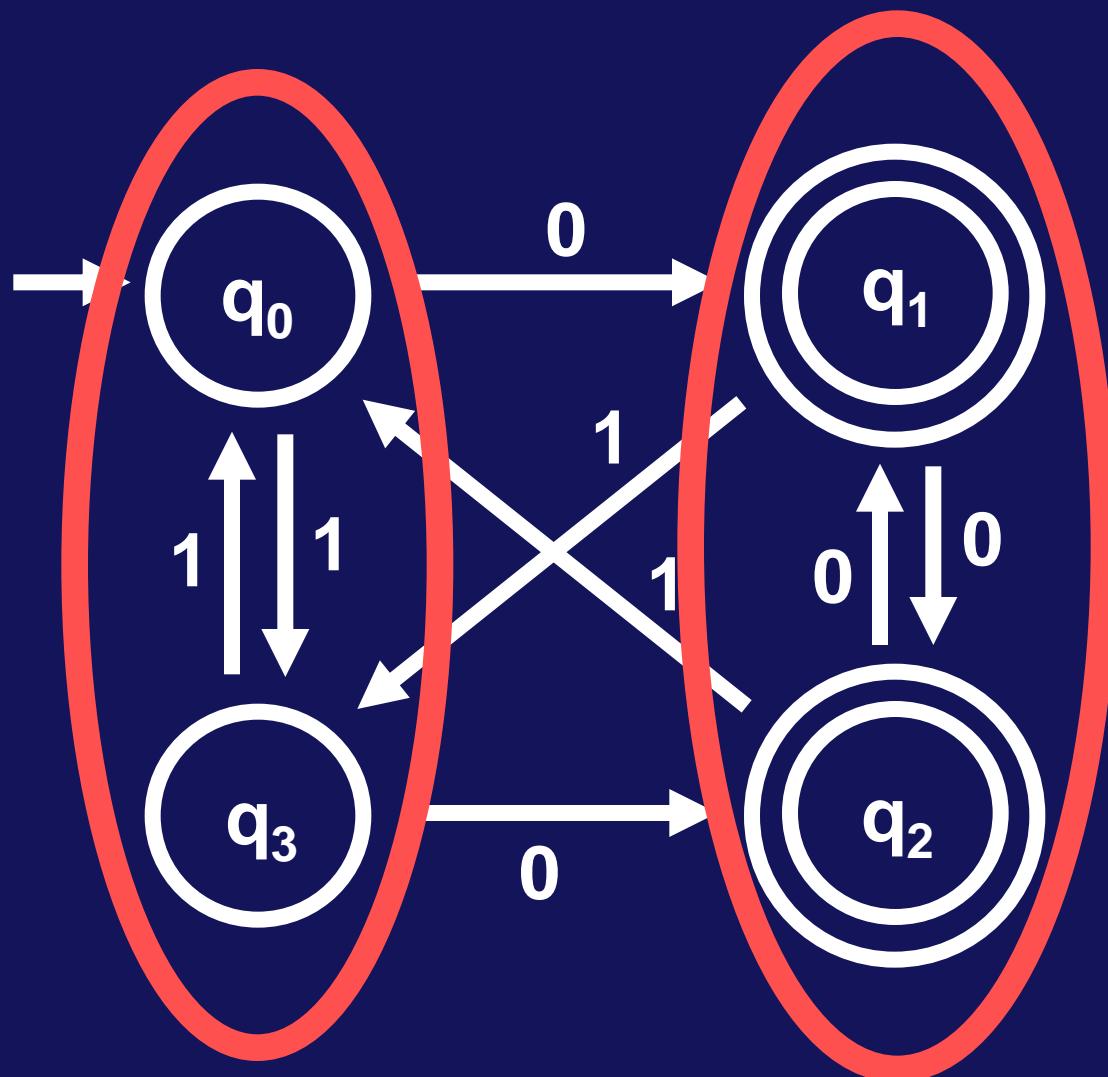
Have you noticed the reason for putting symbols X_1, X_2

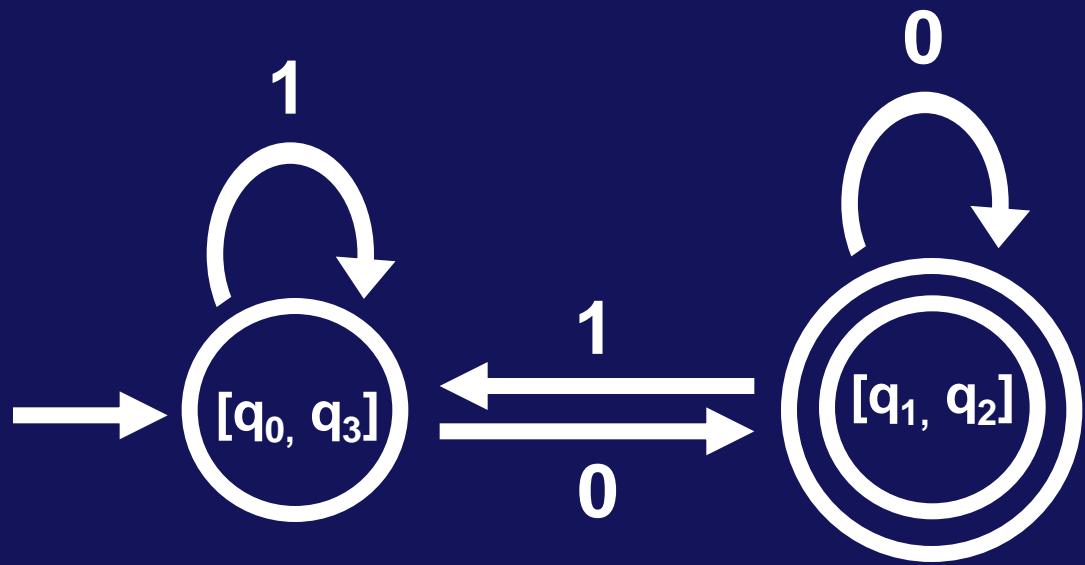
	X_2	
q_2	X_1	X_1
q_3	X_0	X_0

$q_0 \quad q_1 \quad q_2$

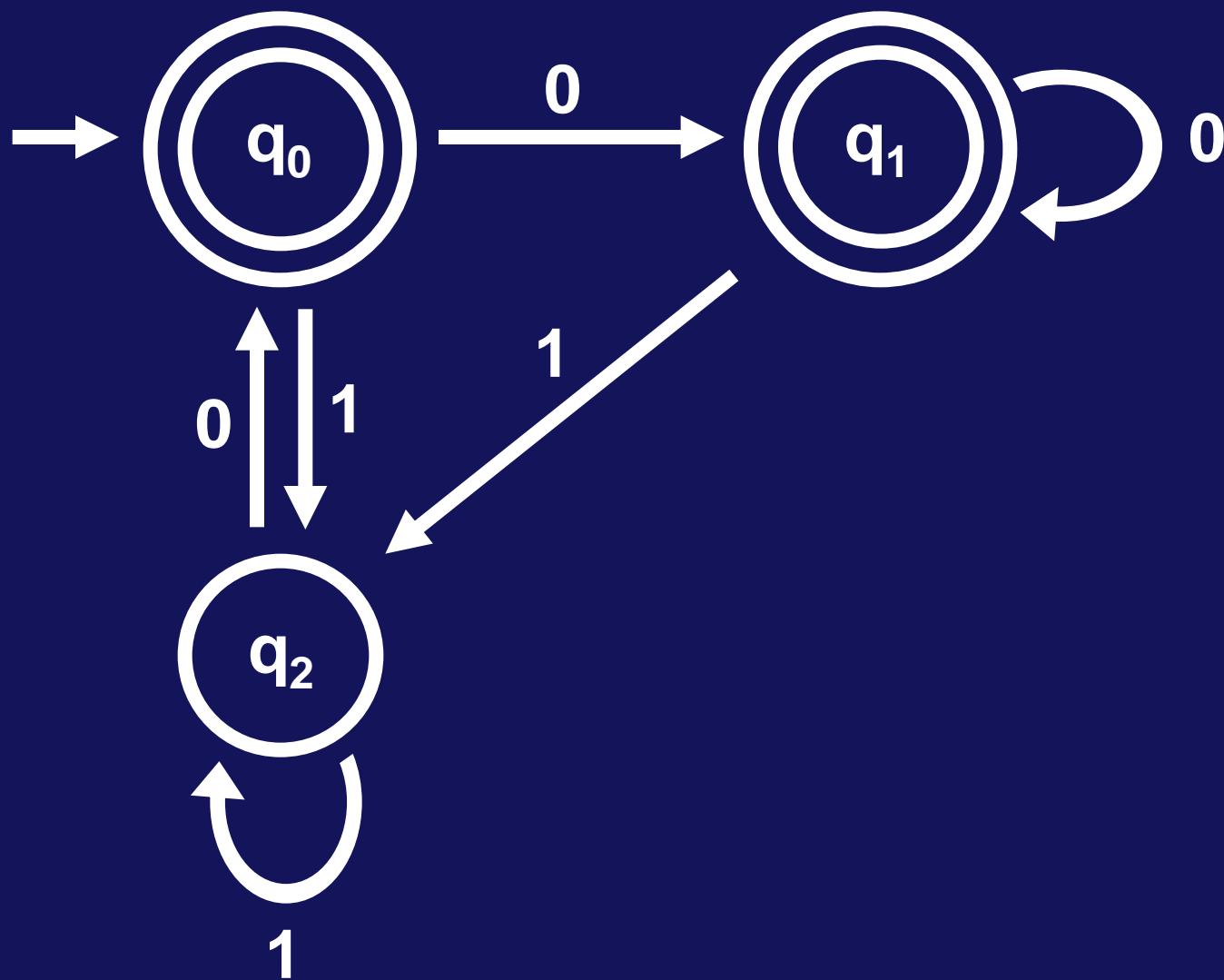




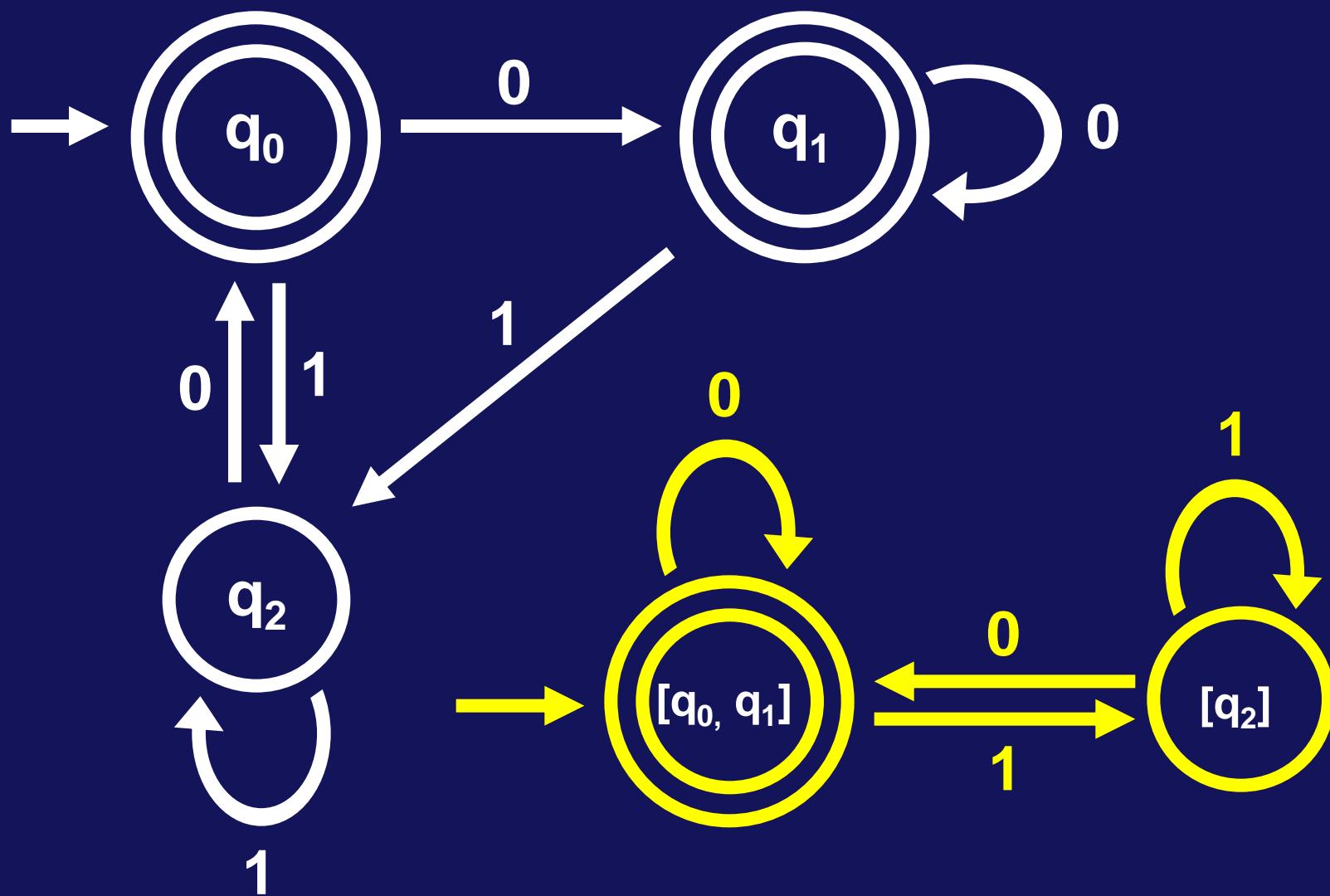


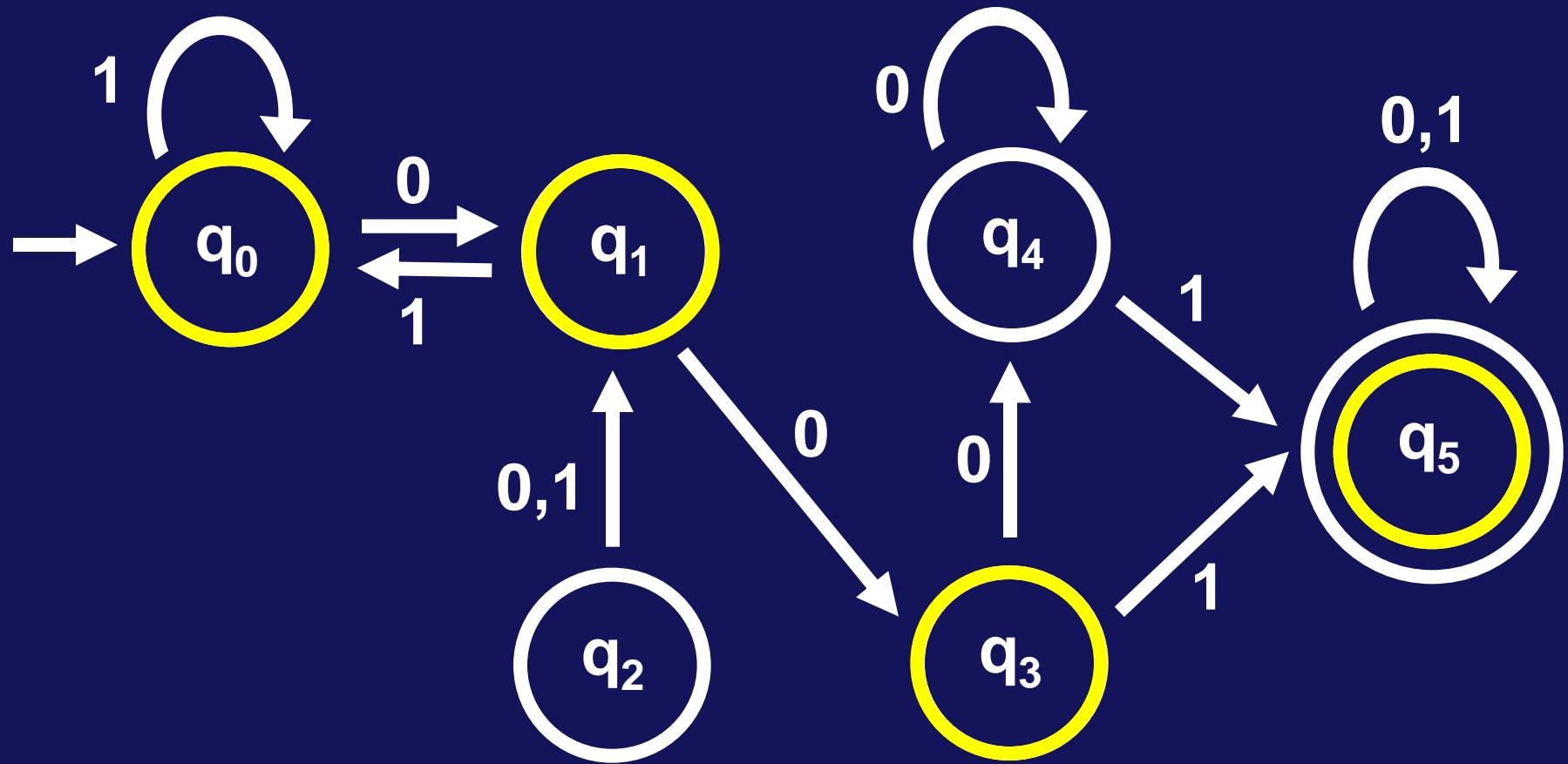


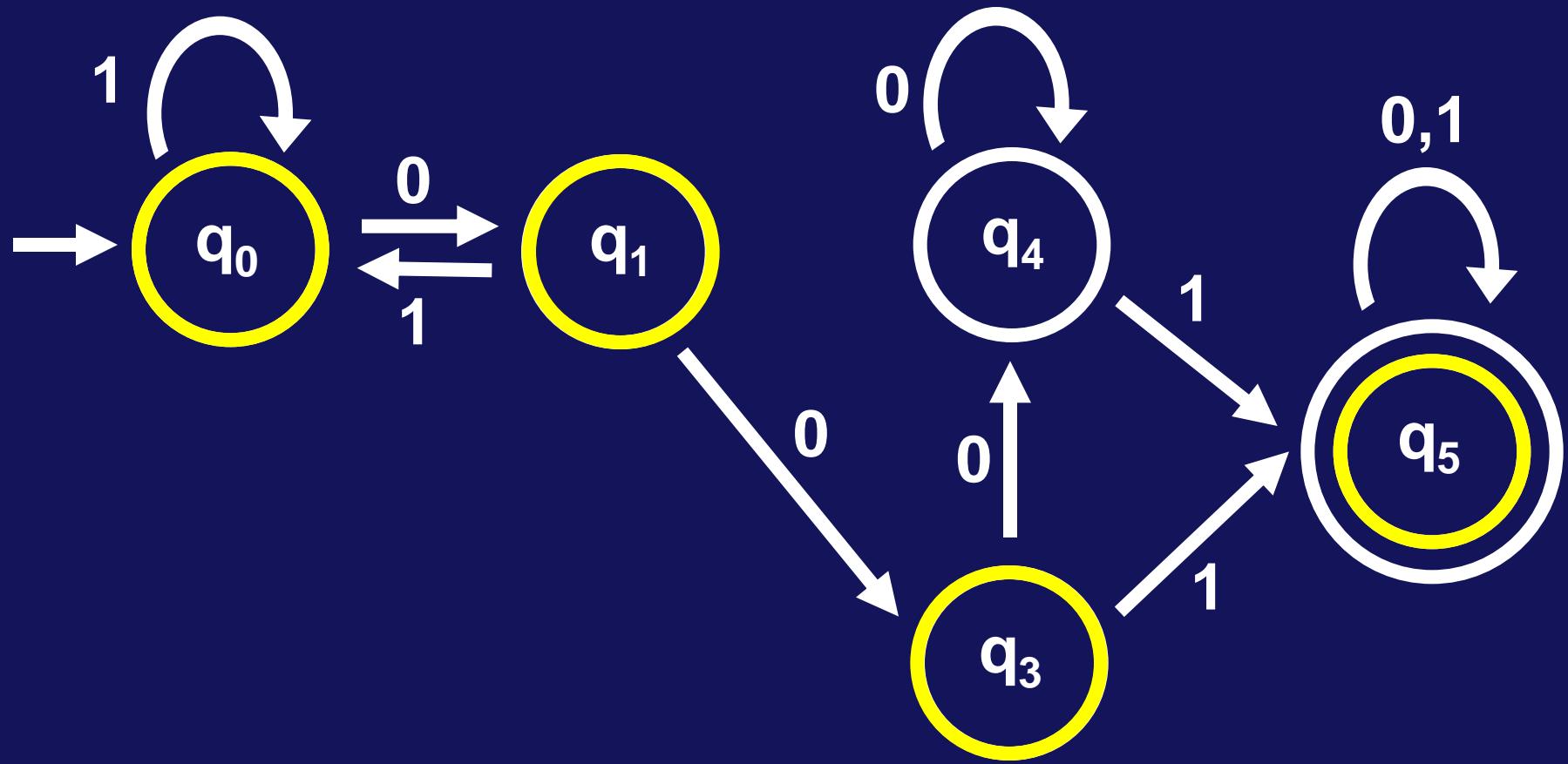
MINIMIZE

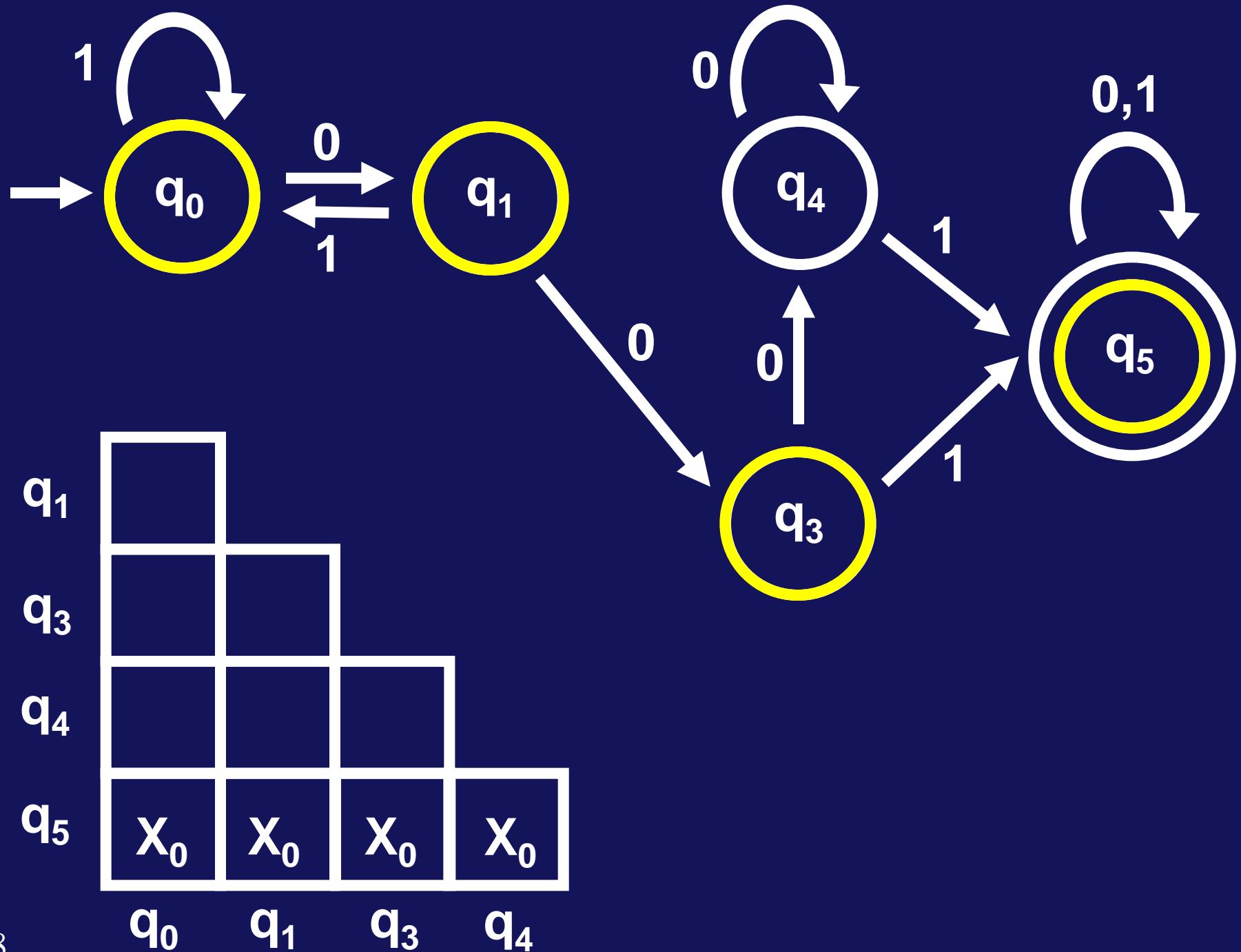


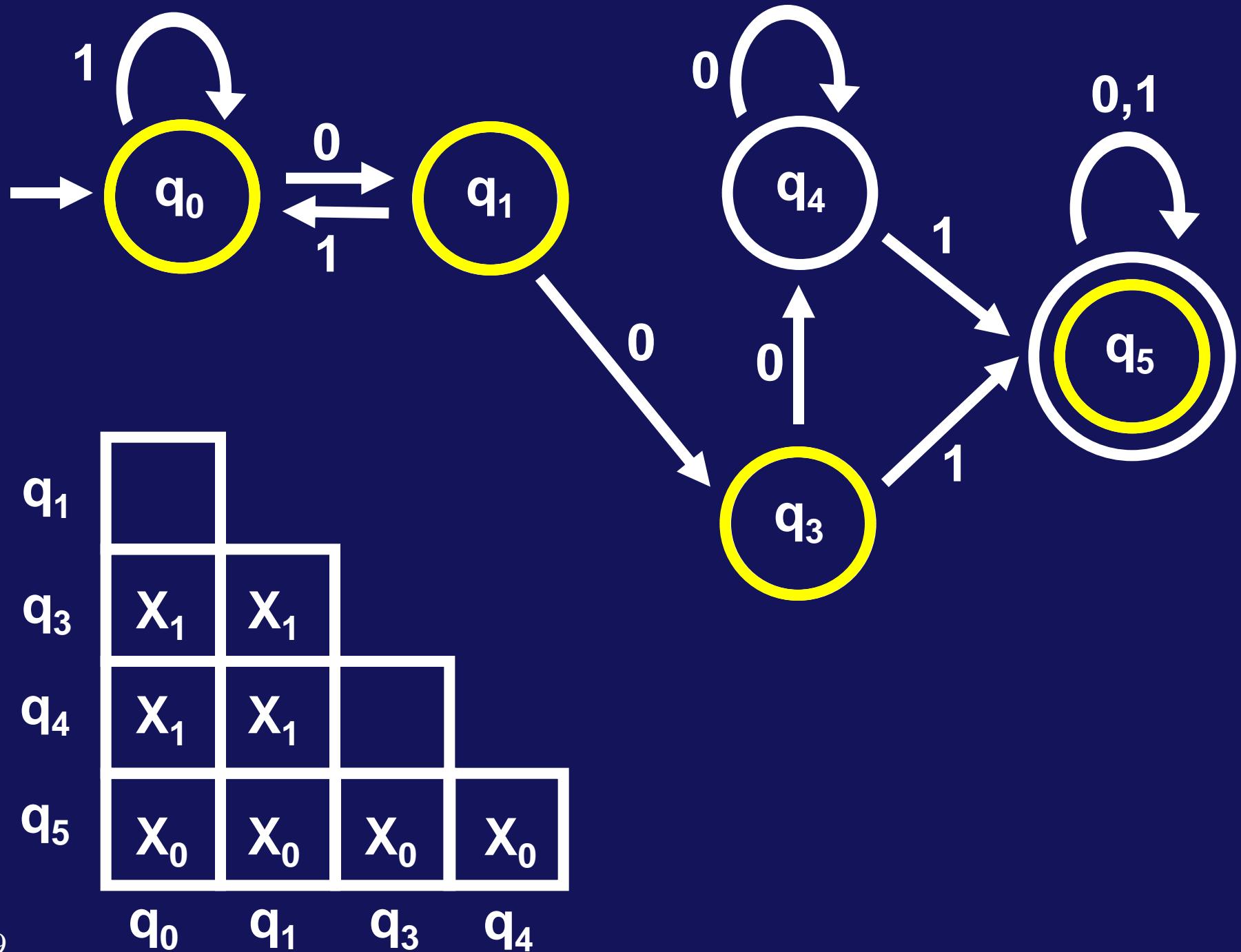
MINIMIZE

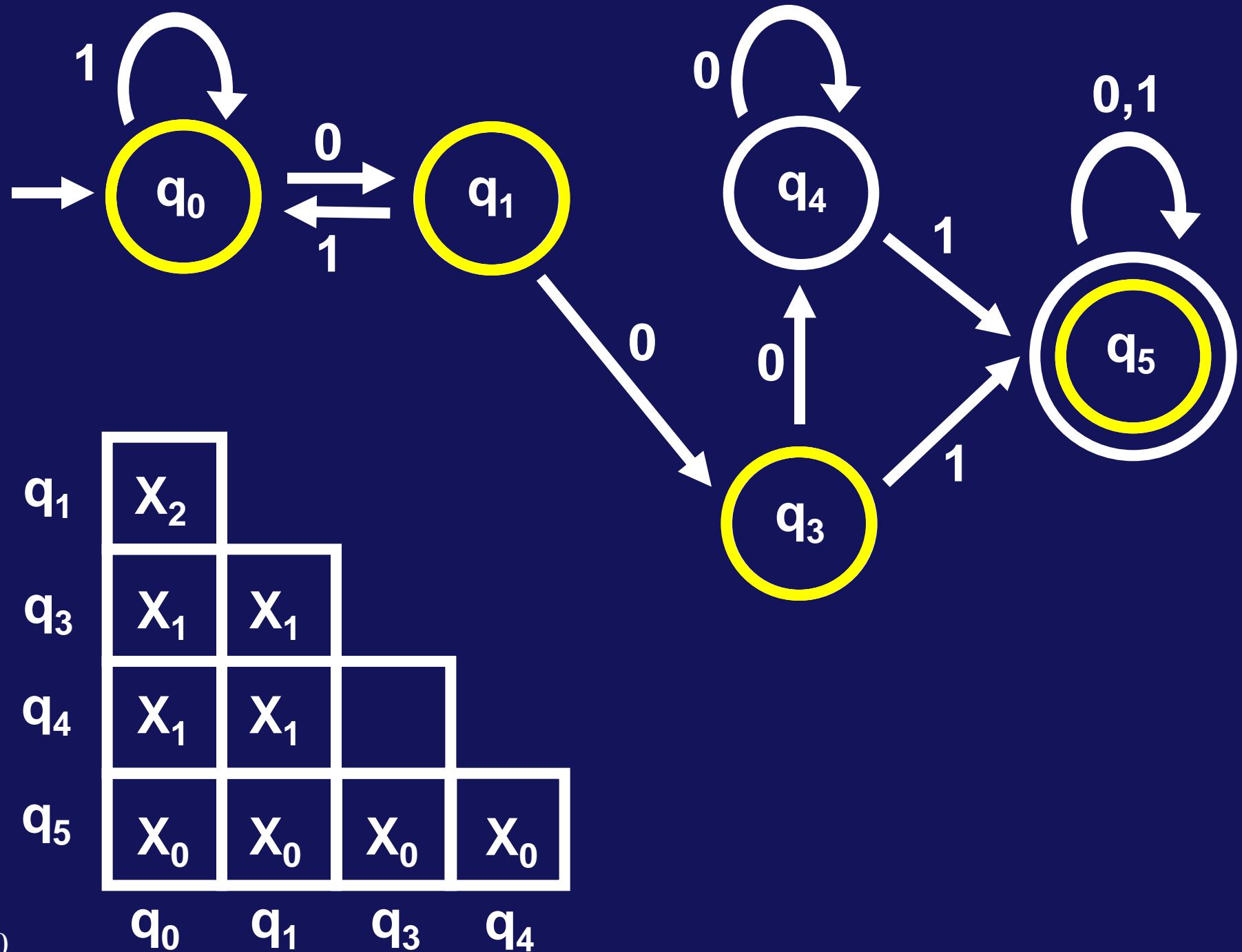


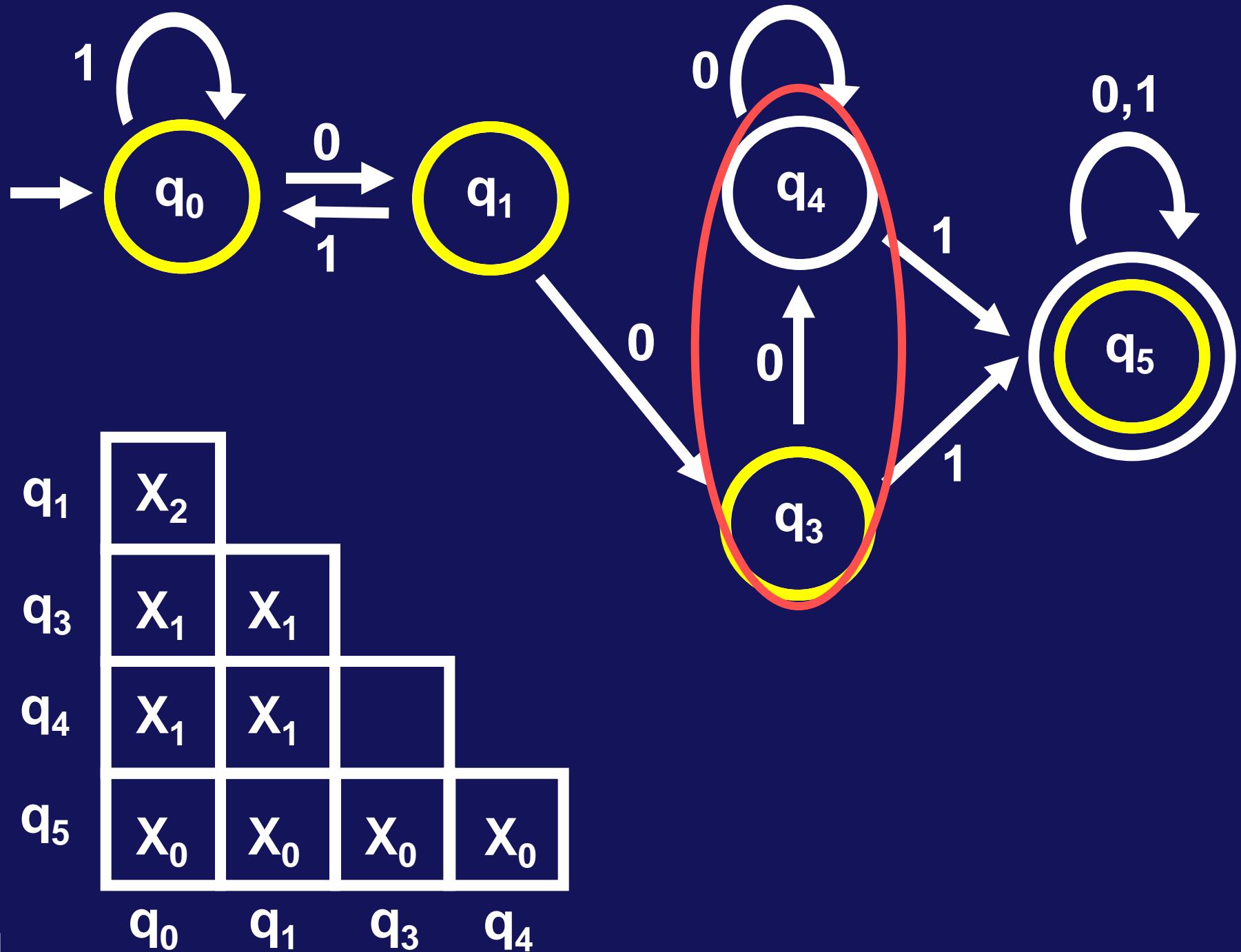


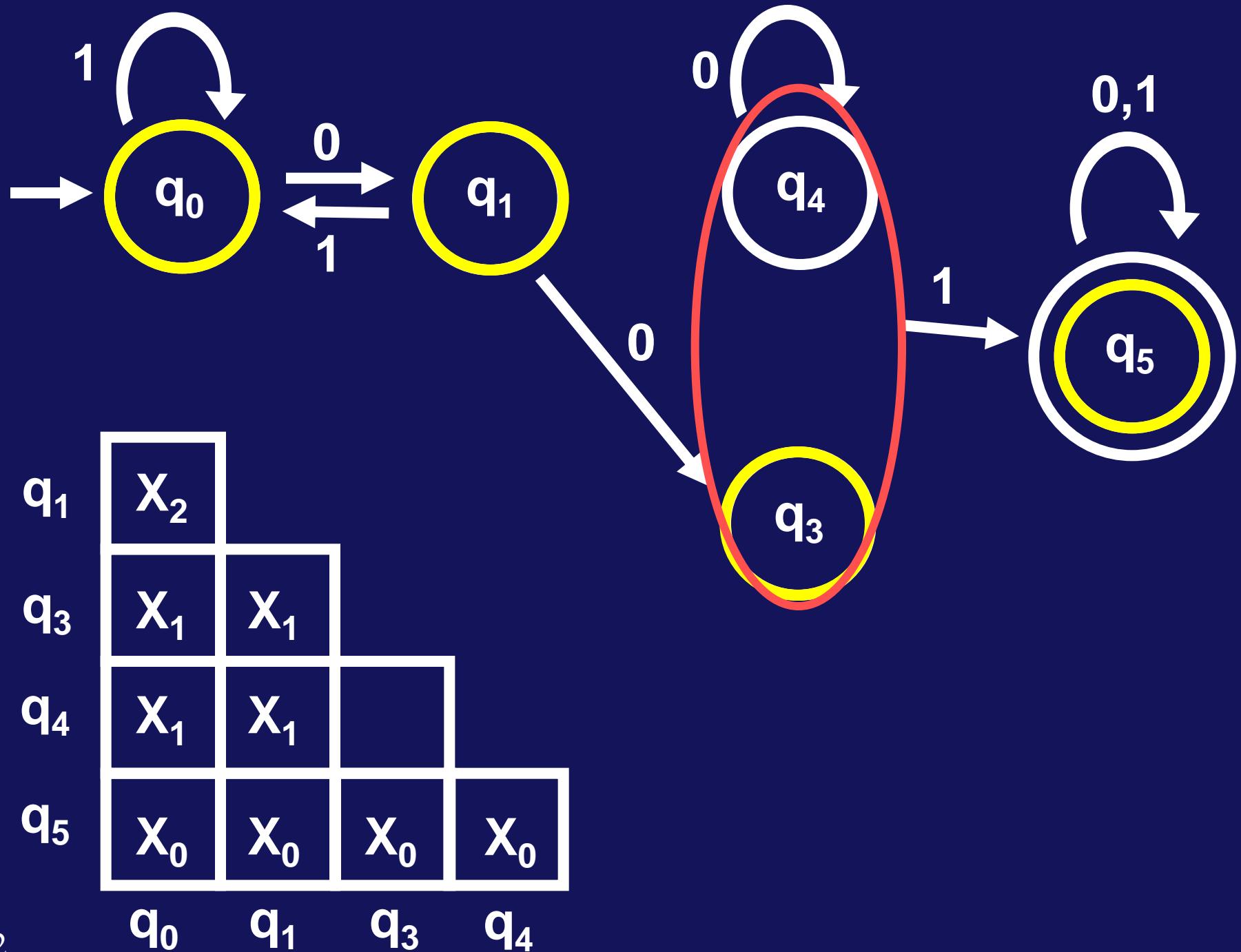


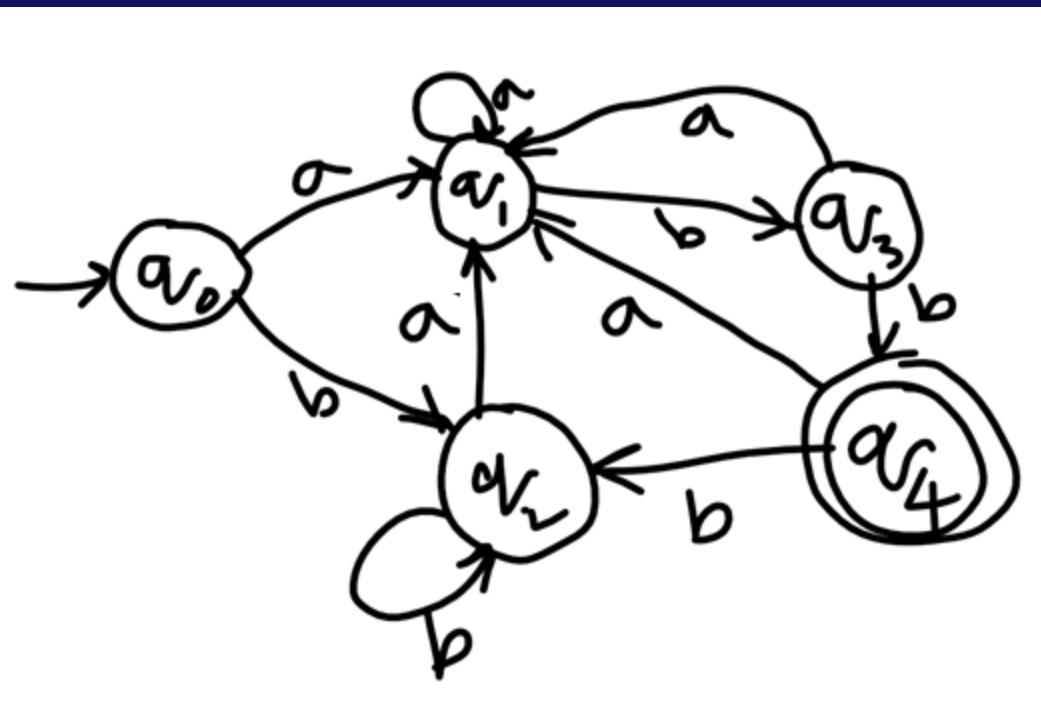


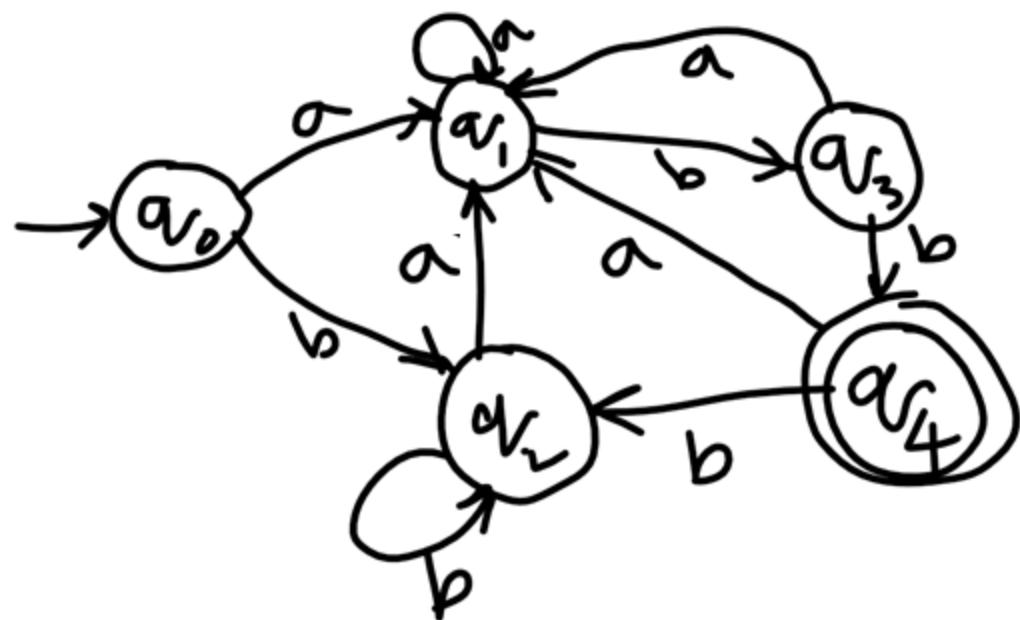




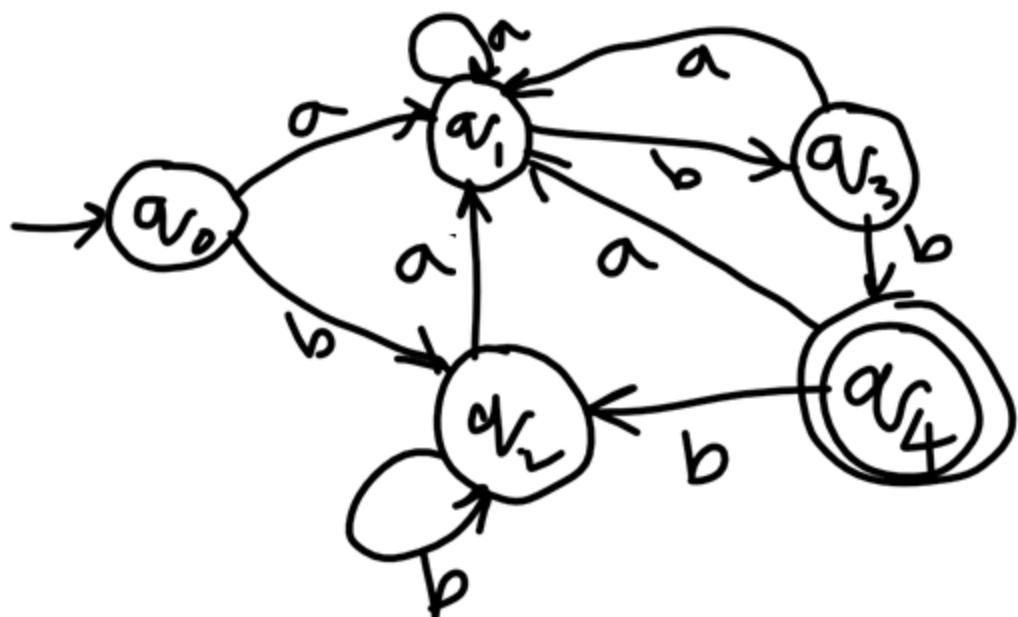








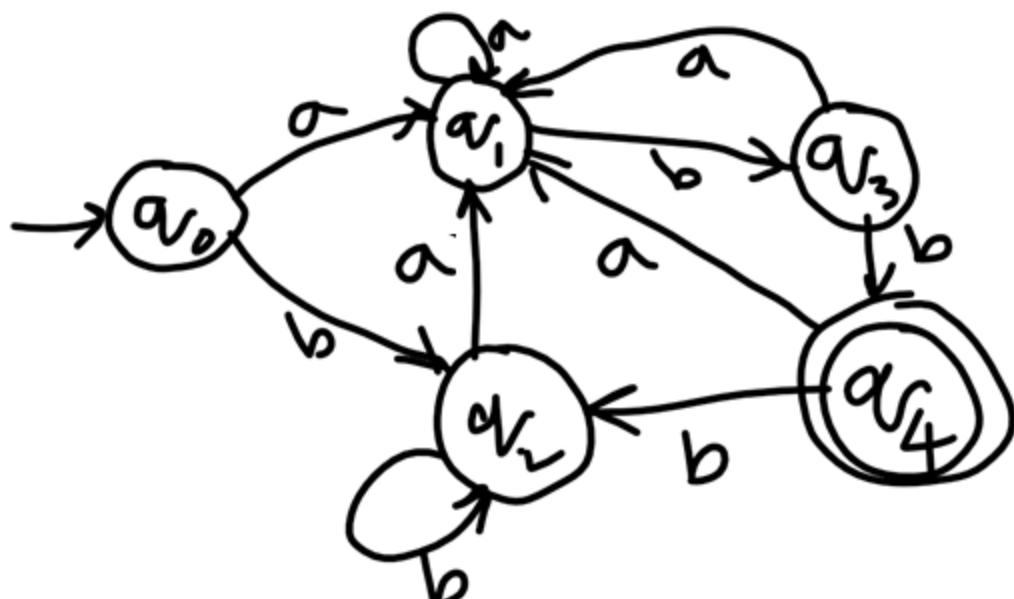
	a	b
a_0	a_1	a_2
a_1	a_1	a_3
a_2	a_1	a_2
a_3	a_1	a_4
a_4	a_1	a_2



	q_{r_4}	q_3	q_2	q_1
q_0	X0			
q_1	X0			
q_2	X0			
q_3	X0			

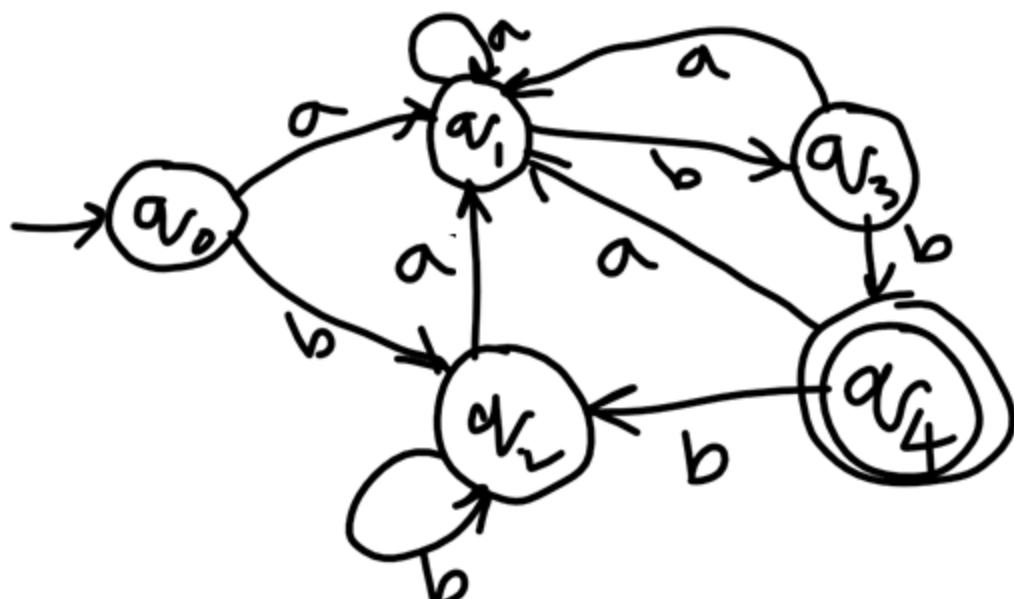
→

	a	b
q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	q_4
q_4	q_1	q_2



	q_{r_0}	q_{r_3}	q_{r_2}	q_{r_1}
q_0	x_0	x_1		
q_1	x_0	x_1		
q_2	x_0	x_1		
q_3	x_0			

	a	b
q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	q_4
q_4	$*$	q_2



	a_{1x}	a_3	a_2	a_1
a_0	x_0	x_1		x_2
a_1	x_0	x_1	x_2	
a_2	x_0	x_1		
a_3	x_0			

→

	a	b
a_0	a_1	a_2
a_1	a_1	a_3
a_2	a_1	a_2
a_3	a_1	a_4
a_4	a_1	a_2

Eg

	0	1
→ v_0	v_1	v_5
v_1	v_6	v_2
* v_2	v_0	v_2
Removed	v_3	v_2
v_4	v_7	v_5
v_5	v_2	v_6
v_6	v_6	v_7
v_7	v_6	v_2

Step 1: Identify unreachable states

$$\begin{aligned} & \{v_0\}^+ \\ &= \{v_0, v_1, v_5, v_6, v_2, v_4, v_7\} \end{aligned}$$

v_3 is not reachable

Eg

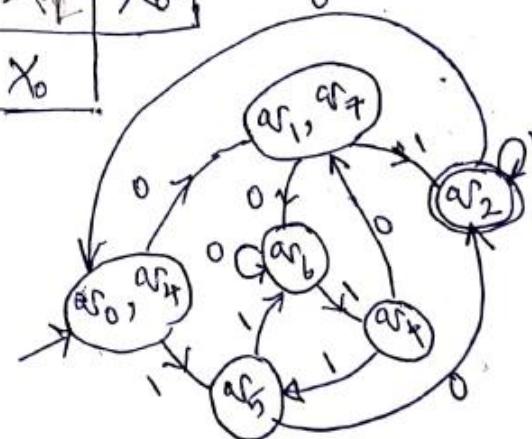
	0	1
v_0	v_1	v_5
v_1	v_6	v_2
v_2	v_0	v_2
Removed		
v_3	v_2	v_6
v_4	v_3	v_5
v_5	v_2	v_6
v_6	v_6	v_2
v_7	v_6	v_2

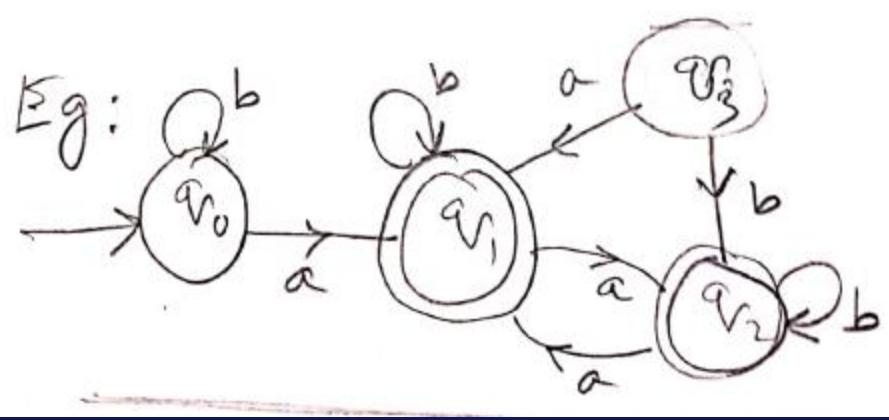
Step 1: Identify unreachable states

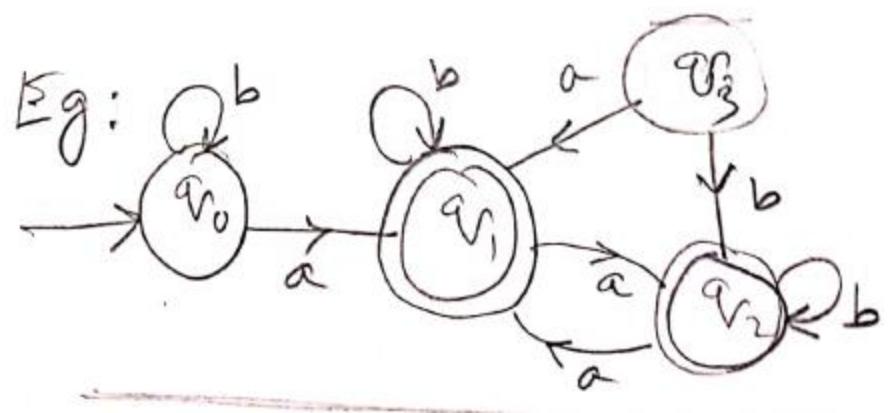
$$\{v_0\}^+ = \{v_0, v_1, v_5, v_6, v_2, v_4, v_7\}$$

v_3 is not reachable

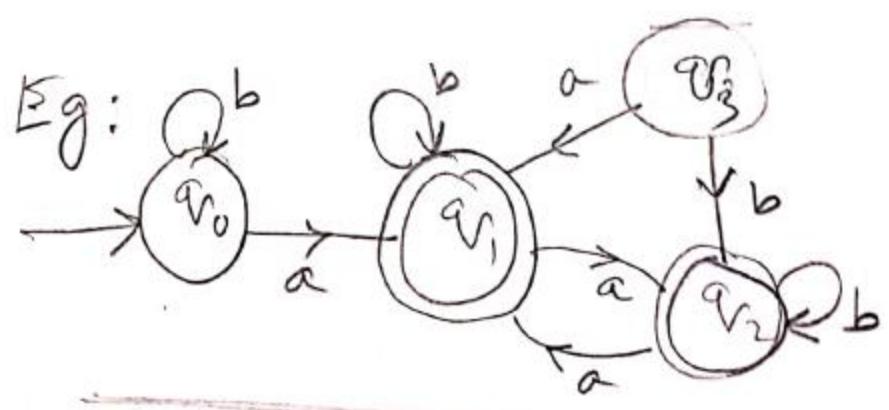
	v_7	v_6	v_5	v_4	v_2	v_1
v_0	x_1	x_2	x_1	\checkmark	x_0	x_1
v_1	\checkmark	x_1	x_1	x_1	x_0	0
v_2	x_0	x_0	x_0	x_0		
v_3	x_1	x_2	x_1			
v_4	x_1	x_1				
v_6		x_1				





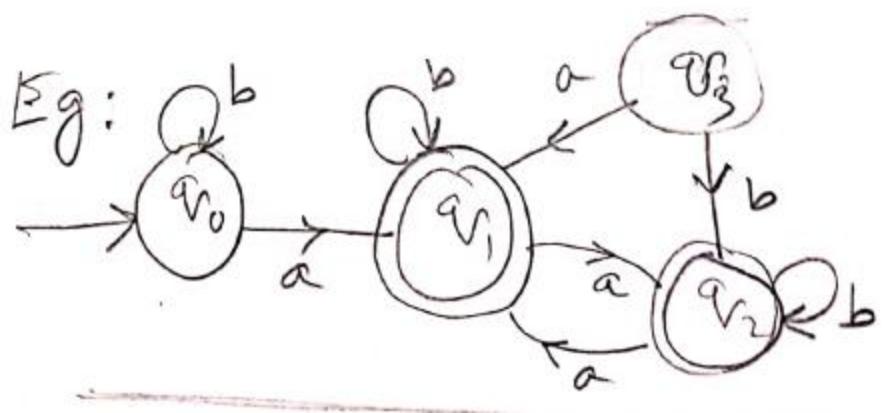


Unreachable = v_3



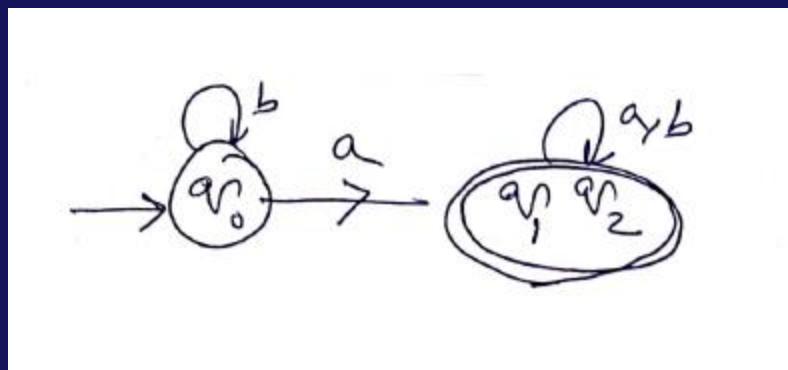
Unreachable = q_3

We get q_1 is equivalent to q_2 (how?)



Unreachable = q_3

We get q_1 is equivalent to q_2 (how?)



Minimal DFA

HOW TO PROVE THAT TWO DFAs ARE EQUIVALENT

- The following is an extract from the Ullman's book.
- Read that book for more information (Reading assignment)

4.4.2 Testing Equivalence of Regular Languages

The table-filling algorithm gives us an easy way to test if two regular languages are the same. Suppose languages L and M are each represented in some way, e.g., one by a regular expression and one by an NFA. Convert each representation to a DFA. Now, imagine one DFA whose states are the union of the states of the DFA's for L and M . Technically, this DFA has two start states, but actually the start state is irrelevant as far as testing state equivalence is concerned, so make any state the lone start state.

Now, test if the start states of the two original DFA's are equivalent, using the table-filling algorithm. If they are equivalent, then $L = M$, and if not, then $L \neq M$.

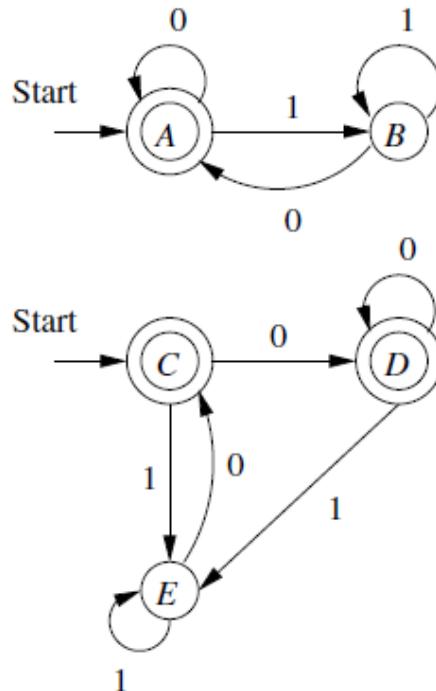
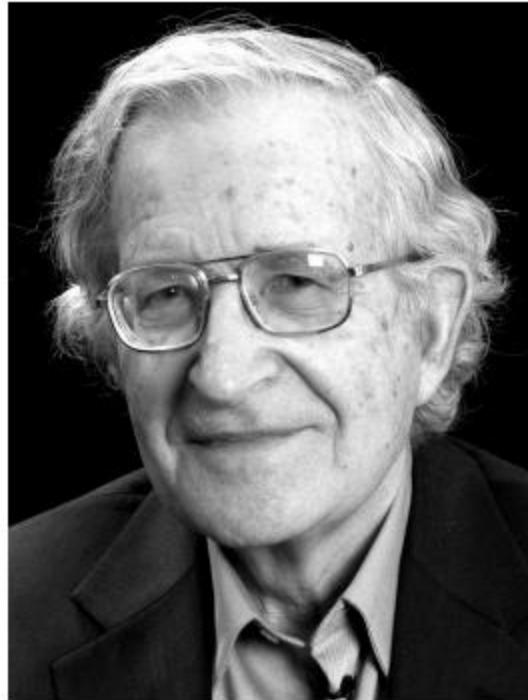


Figure 4.10: Two equivalent DFA's

Context Free Languages

Context Free Grammars

Context-Free Grammars



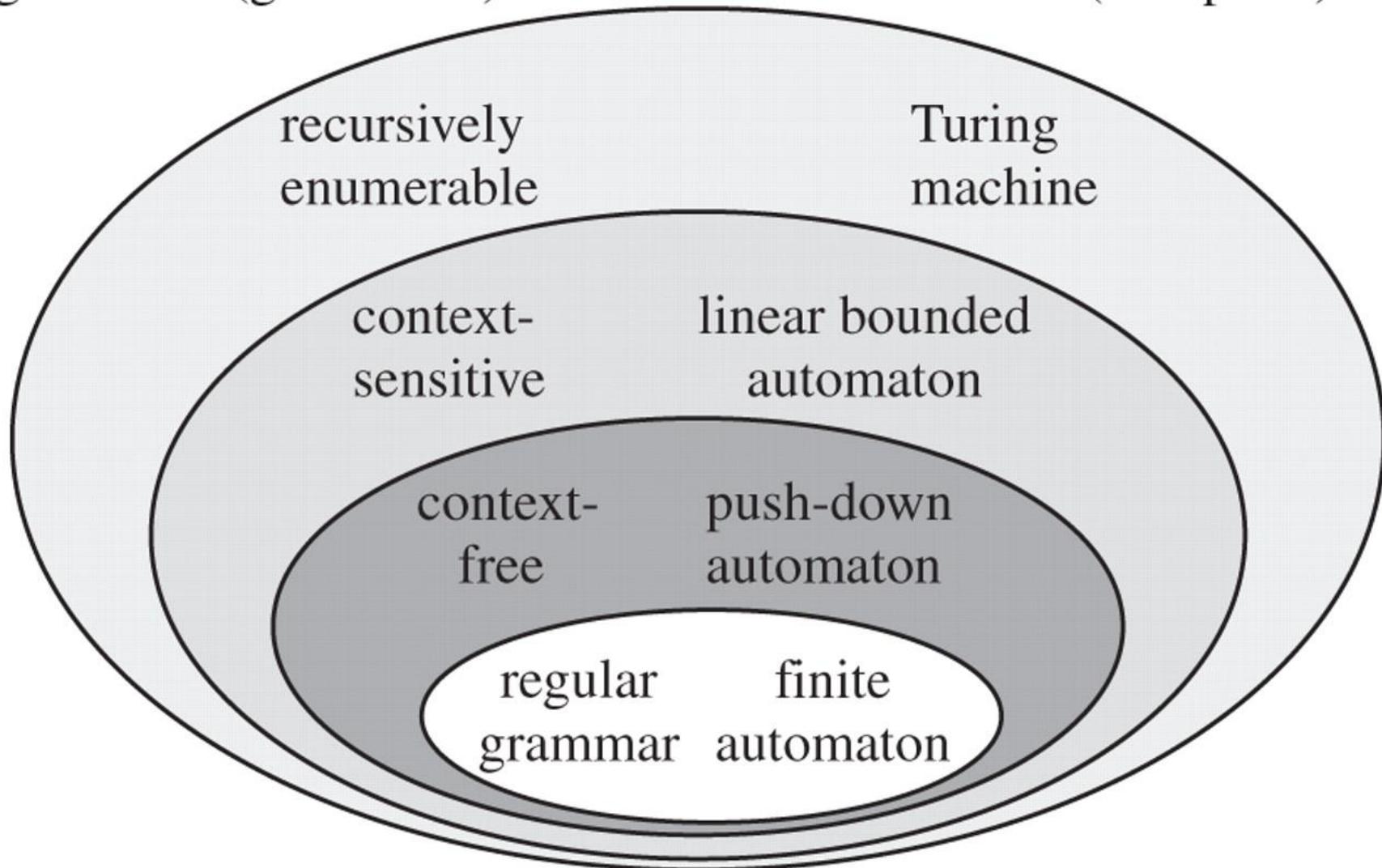
Noam Chomsky
(linguist, philosopher, logician, and activist)

In the formal languages of computer science and linguistics, the **Chomsky hierarchy** is a **hierarchy** of classes of formal grammars. This **hierarchy** of grammars was described by Noam **Chomsky** in 1956.

Chomsky Hierarchy

grammars (generators)

automata (acceptors)



The Hierarchy

Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursive Enumerable	Turing Machine
Type-1	Context Sensitive	Context Sensitive	Linear- Bound
Type-2	Context Free	Context Free	Pushdown
Type-3	Regular	Regular	Finite

How production rules look like

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 $A \rightarrow Ba$

A grammar generates sentences (strings) in a language

Examples

Consider the grammar

$$S \rightarrow AB \tag{1}$$

$$A \rightarrow C \tag{2}$$

$$CB \rightarrow Cb \tag{3}$$

$$C \rightarrow a \tag{4}$$

where $\{a, b\}$ are terminals, and $\{S, A, B, C\}$ are non-terminals.

Examples

Consider the grammar

$$S \rightarrow AB \quad (1)$$

$$A \rightarrow C \quad (2)$$

$$CB \rightarrow Cb \quad (3)$$

$$C \rightarrow a \quad (4)$$

where $\{a, b\}$ are terminals, and $\{S, A, B, C\}$ are non-terminals.

We can derive the phrase “ab” from this grammar in the following way:

$$S \rightarrow AB, \text{ from (1)}$$

$$\rightarrow CB, \text{ from (2)}$$

$$\rightarrow Cb, \text{ from (3)}$$

$$\rightarrow ab, \text{ from (4)}$$

Examples

Consider the grammar

$$S \rightarrow \text{NounPhrase VerbPhrase} \quad (5)$$

$$\text{NounPhrase} \rightarrow \text{SingularNoun} \quad (6)$$

$$\text{SingularNoun VerbPhrase} \rightarrow \text{SingularNoun comes} \quad (7)$$

$$\text{SingularNoun} \rightarrow \text{John} \quad (8)$$

We can derive the phrase “John comes” from this grammar in the following way:

$$\begin{aligned} S &\rightarrow \text{NounPhrase VerbPhrase, from (1)} \\ &\rightarrow \text{SingularNoun VerbPhrase, from (2)} \\ &\rightarrow \text{SingularNoun comes, from (3)} \\ &\rightarrow \text{John comes, from (4)} \end{aligned}$$

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 $A \rightarrow Ba$

Definition (Context-Free Grammar)

A context-free grammar is a tuple $G = (V, T, P, S)$ where

- V is a finite set of variables (nonterminals, nonterminals vocabulary);
- T is a finite set of terminals (letters);
- $P \subseteq V \times (V \cup T)^*$ is a finite set of rewriting rules called productions,
 - We write $A \rightarrow \beta$ if $(A, \beta) \in P$;
- $S \in V$ is a distinguished start or “sentence” symbol.

Definition (Context-Free Grammar)

A context-free grammar is a tuple $G = (V, T, P, S)$ where

- V is a finite set of variables (nonterminals, nonterminals vocabulary);
- T is a finite set of terminals (letters);
- $P \subseteq V \times (V \cup T)^*$ is a finite set of rewriting rules called productions,
 - We write $A \rightarrow \beta$ if $(A, \beta) \in P$;
- $S \in V$ is a distinguished start or “sentence” symbol.

Example: $G_{0^n1^n} = (V, T, P, S)$ where

- $V = \{S\}$;
- $T = \{0, 1\}$;
- P is defined as

$$\begin{array}{lcl} S & \rightarrow & \varepsilon \\ S & \rightarrow & 0S1 \end{array}$$

- $S = S$.

Palindromes

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

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

A context-free grammar for palindromes

Derivation:

- Let $G = (V, T, P, S)$ be a context-free grammar.
- Let $\alpha A \beta$ be a string in $(V \cup T)^* V (V \cup T)^*$
- We say that $\alpha A \beta$ yields the string $\alpha \gamma \beta$, and we write $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if
 $A \rightarrow \gamma$ is a production rule in G .
- For strings $\alpha, \beta \in (V \cup T)^*$, we say that α derives β and we write
 $\alpha \xrightarrow{*} \beta$ if there is a sequence $\alpha_1, \alpha_2, \dots, \alpha_n \in (V \cup T)^*$ s.t.

$$\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \cdots \alpha_n \Rightarrow \beta.$$

\Rightarrow is also called direct derivation.

\xrightarrow{i} is to mean that the i th production is used in the direct derivation.

$\xrightarrow{*}$ is reflexive and transitive closure of \Rightarrow

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$

5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

A context-free grammar for simple expressions

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$

5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

A context-free grammar for simple expressions

T is the set of symbols $\{+, *, (,), a, b, 0, 1\}$ and P is the set of productions

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$

5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

A context-free grammar for simple expressions

T is the set of symbols $\{+, *, (,), a, b, 0, 1\}$ and P is the set of productions

- Can you find how the following is true.

$$E \xrightarrow{*} (a1 + b0 * a1)$$

Compact Notation for Productions

It is convenient to think of a production as “belonging” to the variable of its head. We shall often use remarks like “the productions for A ” or “ A -productions” to refer to the productions whose head is variable A . We may write the productions for a grammar by listing each variable once, and then listing all the bodies of the productions for that variable, separated by vertical bars. That is, the productions $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_n$ can be replaced by the notation $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$. For instance, the grammar for palindromes from Fig. 5.1 can be written as $P \rightarrow \epsilon | 0 | 1 | 0P0 | 1P1$.

CFL Definition

The language $L(G)$ accepted by a context-free grammar $G = (V, T, P, S)$ is the set

$$L(G) = \{w \in T^* : S \xrightarrow{*} w\}.$$

Leftmost and Rightmost Derivations

- Derivations are not unique.
- So, to bring uniqueness, we define two special type of derivations, viz., leftmost and rightmost.

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$

5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

A context-free grammar for simple expressions

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E \underset{lm}{\Rightarrow}$$

$$a * (E) \underset{lm}{\Rightarrow} a * (E + E) \underset{lm}{\Rightarrow} a * (I + E) \underset{lm}{\Rightarrow} a * (a + E) \underset{lm}{\Rightarrow}$$

$$a * (a + I) \underset{lm}{\Rightarrow} a * (a + I0) \underset{lm}{\Rightarrow} a * (a + I00) \underset{lm}{\Rightarrow} a * (a + b00)$$

We can also summarize the leftmost derivation by saying $E \underset{lm}{\stackrel{*}{\Rightarrow}} a * (a + b00)$, or express several steps of the derivation by expressions such as $E * E \underset{lm}{\stackrel{*}{\Rightarrow}} a * (E)$.

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$

5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

A context-free grammar for simple expressions

$$\begin{aligned}
 E &\xrightarrow{rm} E * E \xrightarrow{rm} E * (E) \xrightarrow{rm} E * (E + E) \xrightarrow{rm} \\
 &E * (E + I) \xrightarrow{rm} E * (E + I0) \xrightarrow{rm} E * (E + I00) \xrightarrow{rm} E * (E + b00) \xrightarrow{rm} \\
 &E * (I + b00) \xrightarrow{rm} E * (a + b00) \xrightarrow{rm} I * (a + b00) \xrightarrow{rm} a * (a + b00)
 \end{aligned}$$

This derivation allows us to conclude $E \xrightarrow[rm]{*} a * (a + b00)$. \square

Exercise

Consider the following grammar:

$$\begin{aligned} S &\rightarrow AS \mid \varepsilon. \\ A &\rightarrow aa \mid ab \mid ba \mid bb \end{aligned}$$

Give leftmost and rightmost derivations of the string *aabbba*.

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

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

A context-free grammar for palindromes

Prove that $L(G_{pal})$ is the set of palindromes over the given alphabet.

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

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

A context-free grammar for palindromes

Prove that $L(G_{pal})$ is the set of palindromes over the given alphabet.

- This proof has two parts (\Rightarrow and \Leftarrow)

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

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

A context-free grammar for palindromes

Prove that $L(G_{pal})$ is the set of palindromes over the given alphabet.

- This proof has two parts (\Rightarrow and \Leftarrow)
 - 1) $(w = w^R) \Rightarrow w \in L(G_{pal})$
 - 2) $w \in L(G_{pal}) \Rightarrow (w = w^R)$

$$(w = w^R) \Rightarrow w \in L(G_{pal})$$

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

A context-free grammar for palindromes

- Proof [by induction on $|w|$]:

BASIS: We use lengths 0 and 1 as the basis.

If $|w| = 0$ or $|w| = 1$, then w is ϵ , 0, or 1.

Since there are productions $P \rightarrow \epsilon$, $P \rightarrow 0$, and $P \rightarrow 1$, we conclude that $P \xrightarrow{*} w$ in any of these basis cases.

- Note, $w \in L(G_{pal})$ is same $P \xrightarrow{*} w$
- Note, $(w = w^R)$ means w begins and ends with the same character.

BASIS: We use lengths 0 and 1 as the basis.

If $|w| = 0$ or $|w| = 1$, then w is ϵ , 0, or 1.

Since there are productions $P \rightarrow \epsilon$, $P \rightarrow 0$, and $P \rightarrow 1$, we conclude that $P \xrightarrow{*} w$ in any of these basis cases.

BASIS: We use lengths 0 and 1 as the basis.

If $|w| = 0$ or $|w| = 1$, then w is ϵ , 0, or 1.

Since there are productions $P \rightarrow \epsilon$, $P \rightarrow 0$, and $P \rightarrow 1$, we conclude that $P \xrightarrow{*} w$ in any of these basis cases.

Inductive Hypothesis: Let for $|w| \leq k$ where ($w = w^R$), $P \xrightarrow{*} w$ is true.

Inductive Step: We need to show for $|w| = k + 1, P \xrightarrow{*} w$ is true.

BASIS: We use lengths 0 and 1 as the basis.

If $|w| = 0$ or $|w| = 1$, then w is ϵ , 0, or 1.

Since there are productions $P \rightarrow \epsilon$, $P \rightarrow 0$, and $P \rightarrow 1$, we conclude that $P \xrightarrow{*} w$ in any of these basis cases.

Inductive Hypothesis: Let for $|w| \leq k$ where ($w = w^R$), $P \xrightarrow{*} w$ is true.

Inductive Step: We need to show for $|w| = k + 1, P \xrightarrow{*} w$ is true.

Note, $w = 0x0$ or $w = 1x1$, where $|x| = k - 1$.

Then, $P \Rightarrow 0P0 \xrightarrow{*} 0x0$ (Since $|x| \leq k$, so $P \xrightarrow{*} x$ is true).

So, $P \xrightarrow{*} w$ is true. With a similar argument, $P \Rightarrow 1P1 \xrightarrow{*} 1x1$

BASIS: We use lengths 0 and 1 as the basis.

If $|w| = 0$ or $|w| = 1$, then w is ϵ , 0, or 1.

Since there are productions $P \rightarrow \epsilon$, $P \rightarrow 0$, and $P \rightarrow 1$, we conclude that $P \xrightarrow{*} w$ in any of these basis cases.

Inductive Hypothesis: Let for $|w| \leq k$ where ($w = w^R$), $P \xrightarrow{*} w$ is true.

Inductive Step: We need to show for $|w| = k + 1, P \xrightarrow{*} w$ is true.

Note, $w = 0x0$ or $w = 1x1$, where $|x| = k - 1$.

Then, $P \Rightarrow 0P0 \xrightarrow{*} 0x0$ (Since $|x| \leq k$, so $P \xrightarrow{*} x$ is true).

So, $P \xrightarrow{*} w$ is true. With a similar argument, $P \Rightarrow 1P1 \xrightarrow{*} 1x1$

This completes our proof for :

$$(w = w^R) \Rightarrow w \in L(G_{pal})$$

$$w \in L(G_{pal}) \Rightarrow (w = w^R)$$

- Proof [by induction on number of steps in the derivation]:

BASIS: If the derivation is one step, then it must use one of the three productions that do not have P in the body. That is, the derivation is $P \Rightarrow \epsilon$, $P \Rightarrow 0$, or $P \Rightarrow 1$. Since ϵ , 0, and 1 are all palindromes, the basis is proven.

INDUCTION:

- Assume for n steps it is true.
- Then, show for $(n+1)$ steps it must be true.

Left as an exercise.

Sentential Forms

$G = (\bar{V}, \bar{T}, P, S)$ is a CFG, then any string α in $(V \cup T)^*$ such that $S \xrightarrow{*} \alpha$ is a *sentential form*.

Sentential Forms

$G = (\bar{V}, \bar{T}, P, S)$ is a CFG, then any string α in $(V \cup T)^*$ such that $S \xrightarrow{*} \alpha$ is a *sentential form*.

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$

5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

A context-free grammar for simple expressions

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$$

Sentential Forms

$G = (\bar{V}, \bar{T}, P, S)$ is a CFG, then any string α in $(V \cup T)^*$ such that $S \xrightarrow{*} \alpha$ is a *sentential form*.

If $S \xrightarrow{lm}^* \alpha$, then α is a *left-sentential form*,

and if $S \xrightarrow{rm}^* \alpha$, then α is a *right-sentential form*.

Note that the language $L(G)$ is those sentential forms that are in T^* ; i.e., they consist solely of terminals.

1.	E	\rightarrow	I
2.	E	\rightarrow	$E + E$
3.	E	\rightarrow	$E * E$
4.	E	\rightarrow	(E)

5.	I	\rightarrow	a
6.	I	\rightarrow	b
7.	I	\rightarrow	Ia
8.	I	\rightarrow	Ib
9.	I	\rightarrow	$I0$
10.	I	\rightarrow	$I1$

A context-free grammar for simple expressions

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$$

- Is this sentential form left-sentential? Or right-sentential?

1.	E	\rightarrow	I
2.	E	\rightarrow	$E + E$
3.	E	\rightarrow	$E * E$
4.	E	\rightarrow	(E)

5.	I	\rightarrow	a
6.	I	\rightarrow	b
7.	I	\rightarrow	Ia
8.	I	\rightarrow	Ib
9.	I	\rightarrow	$I0$
10.	I	\rightarrow	$I1$

A context-free grammar for simple expressions

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$$

- Is this sentential form left-sentential? Or right-sentential?
- It is a sentential form. But neither left nor right.

Exercise 5.1.2: The following grammar generates the language of regular expression $0^*1(0+1)^*$:

$$\begin{array}{lcl} S & \rightarrow & A1B \\ A & \rightarrow & 0A \mid \epsilon \\ B & \rightarrow & 0B \mid 1B \mid \epsilon \end{array}$$

Give leftmost and rightmost derivations of the following strings:

- * a) 00101.
- b) 1001.
- c) 00011.

Note, the given grammar is not a regular grammar (even-though it generates a regular language).

Can you find $L(G)$?

- $S \rightarrow aS|bS|a|b|\epsilon$

Can you find $L(G)$?

- $S \rightarrow aS|bS|a|b|\epsilon$
- Answer: All strings. Σ^*

Can you find $L(G)$?

1. $S \rightarrow S_1S|\epsilon,$
2. $S_1 \rightarrow aS_1b|ab$

Can you find $L(G)$?

1. $S \rightarrow S_1S|\epsilon,$
2. $S_1 \rightarrow aS_1b|ab$

Recall that the $S \rightarrow aSb|\epsilon$ generates $\{a^n b^n | n \geq 0\}$.

Can you find $L(G)$?

1. $S \rightarrow S_1S|\epsilon,$
2. $S_1 \rightarrow aS_1b|ab$

Recall that the $S \rightarrow aSb|\epsilon$ generates $\{a^n b^n | n \geq 0\}$.

Starting from S_1 we get $\{a^n b^n | n \geq 1\}$

Can you find $L(G)$?

1. $S \rightarrow S_1S|\epsilon,$
2. $S_1 \rightarrow aS_1b|ab$

Recall that the $S \rightarrow aSb|\epsilon$ generates $\{a^n b^n | n \geq 0\}$.

Starting from S_1 we get $\{a^n b^n | n \geq 1\}$

The answer:

$$a^{n_1}b^{n_1}a^{n_2}b^{n_2} \dots a^{n_k}b^{n_k} \in L(G)$$

$$L(G) = (\{a^n b^n | n \geq 1\})^*$$

Can you find $L(G)$?

$$S \rightarrow SS \mid [S] \mid (S) \mid [] \mid ()$$

Can you find $L(G)$?

$$S \rightarrow SS \mid [S] \mid (S) \mid [] \mid ()$$

Set of all balanced parentheses with alphabet
 $\{ (,),[],[]\}$

Can you find $L(G)$?

1. $S \rightarrow aB|bA$
2. $B \rightarrow b|bS|aBB$
3. $A \rightarrow a|aS|bAA$

Can you find $L(G)$?

1. $S \rightarrow aB|bA$
2. $B \rightarrow b|bS|aBB$
3. $A \rightarrow a|aS|bAA$

Produces strings with equal number of a's and b's.

Can you find $L(G)$?

1. $S \rightarrow SaSbS | SbSaS | \epsilon$

Can you find $L(G)$?

$$1. \ S \rightarrow SaSbS | SbSaS | \epsilon$$

Produces strings with equal number of a's and b's.

With one difference than the previous CFG. **What is it?**

Parse tree and ambiguity

Parse tree representation of the derivation.

- It is tree representation of the derivation.
- For a given derivation, there is only one parse tree.
- But, for a given parse tree, there may be many derivations.

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

Figure 5.1: A context-free grammar for palindromes

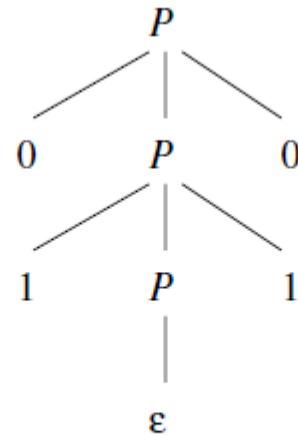


Figure 5.5: A parse tree showing the derivation $P \xrightarrow{*} 0110$

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$

5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

Figure 5.2: A context-free grammar for simple expressions

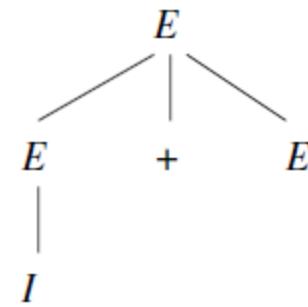


Figure 5.4: A parse tree showing the derivation of $I + E$ from E

1. Each interior node is labeled by a variable in V .
2. Each leaf is labeled by either a variable, a terminal, or ϵ . However, if the leaf is labeled ϵ , then it must be the only child of its parent.
3. If an interior node is labeled A , and its children are labeled

$$X_1, X_2, \dots, X_k$$

respectively, from the left, then $A \rightarrow X_1 X_2 \cdots X_k$ is a production in P .

5.2.2 The Yield of a Parse Tree

1. The yield is a terminal string. That is, all leaves are labeled either with a terminal or with ϵ .
2. The root is labeled by the start symbol.

1. $E \rightarrow I$
 2. $E \rightarrow E + E$
 3. $E \rightarrow E * E$
 4. $E \rightarrow (E)$

5. $I \rightarrow a$
 6. $I \rightarrow b$
 7. $I \rightarrow Ia$
 8. $I \rightarrow Ib$
 9. $I \rightarrow I0$
 10. $I \rightarrow I1$

Figure 5.2: A context-free grammar for simple expressions

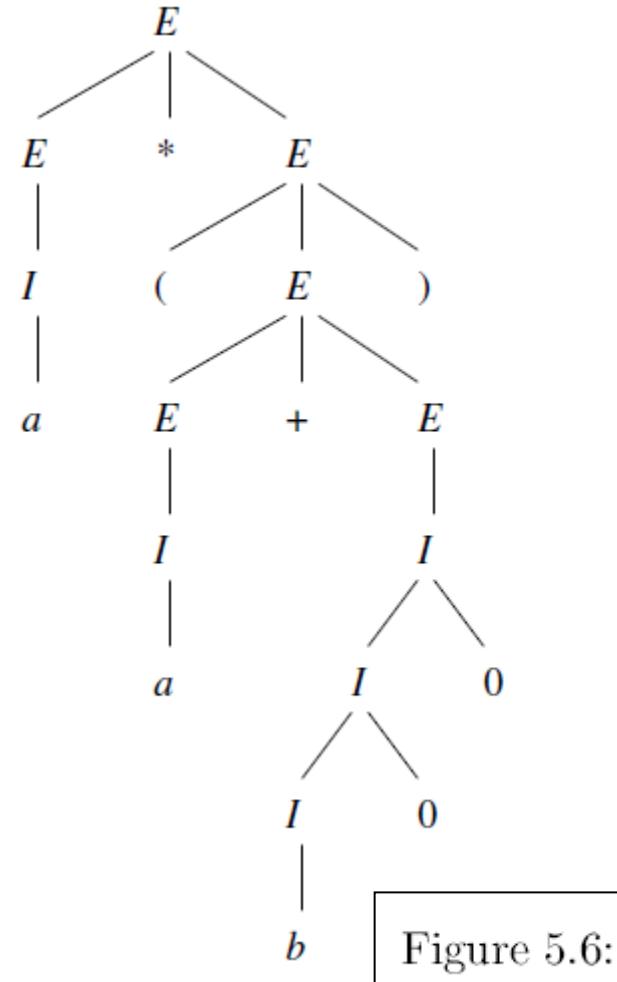


Figure 5.6:

Parse tree for the yield $a * (a + b00)$

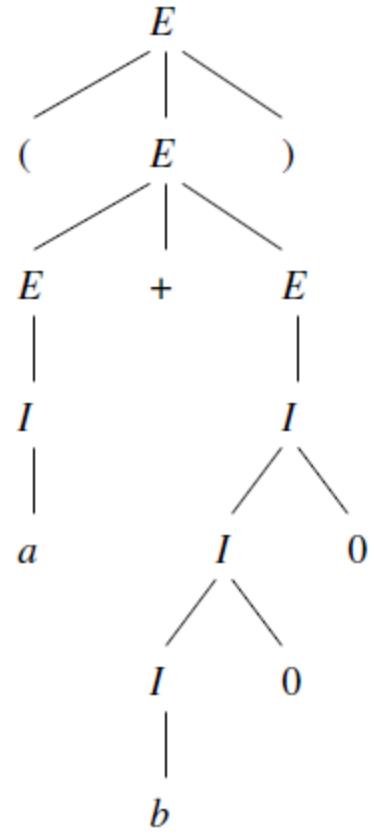
Parse tree representation of the derivation.

- It is tree representation of the derivation.
- For a given derivation, there is only one parse tree.
- But, for a given parse tree, there may be many derivations.

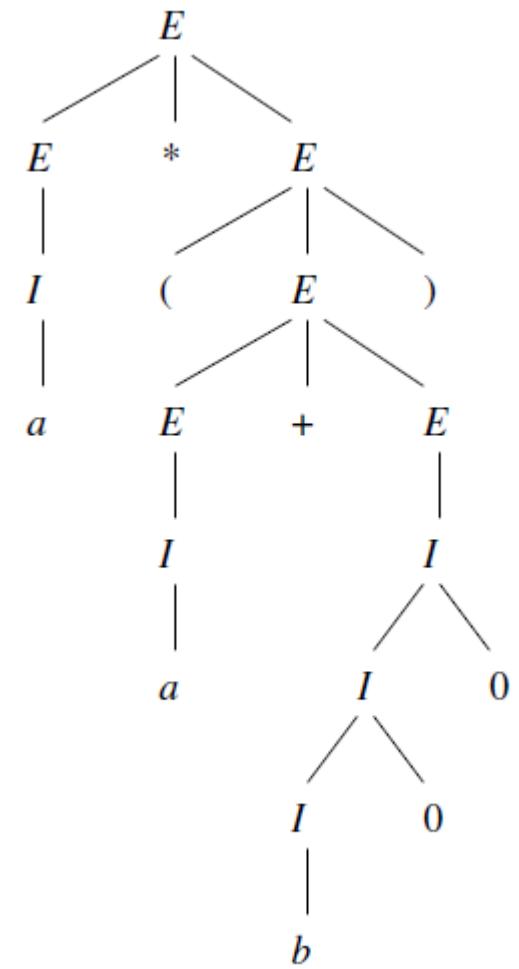
- For a given parse tree there is a unique leftmost derivation.
- Similarly, for a given parse tree there is a unique rightmost derivation.

$$E \xrightarrow{lm} (E) \xrightarrow{lm} (E + E) \xrightarrow{lm} (I + E) \xrightarrow{lm} (a + E) \xrightarrow{lm}$$

$$(a + I) \xrightarrow{lm} (a + I0) \xrightarrow{lm} (a + I00) \xrightarrow{lm} (a + b00)$$



- Can you find the rightmost derivation?

$$E \Rightarrow \underset{lm}{E * E} \Rightarrow \underset{lm}{I * E} \Rightarrow \underset{lm}{a * E} \Rightarrow$$
$$a * (E) \Rightarrow \underset{lm}{a * (E + E)} \Rightarrow \underset{lm}{a * (I + E)} \Rightarrow \underset{lm}{a * (a + E)} \Rightarrow$$
$$a * (a + I) \Rightarrow \underset{lm}{a * (a + I0)} \Rightarrow \underset{lm}{a * (a + I00)} \Rightarrow \underset{lm}{a * (a + b00)}$$


- Can you find the rightmost derivation?

- **Can you do this?**

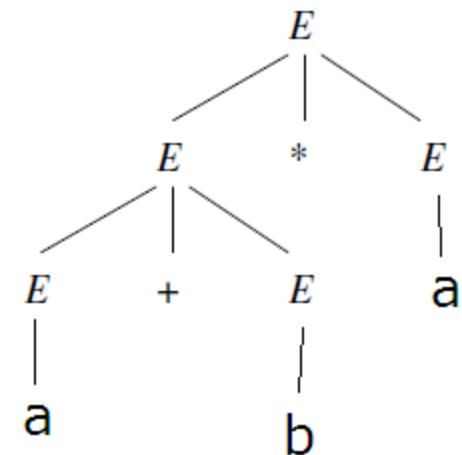
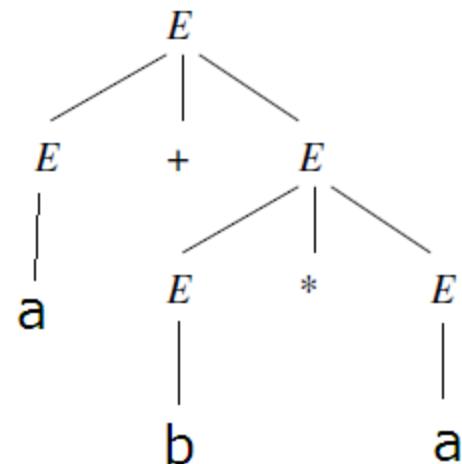
! **Exercise 5.2.2:** Suppose that G is a CFG without any productions that have ϵ as the right side. If w is in $L(G)$, the length of w is n , and w has a derivation of m steps, show that w has a parse tree with $n + m$ nodes.

Ambiguous grammar

- The CFG is ambiguous, if there is a string in the language for which there are more than one parse tree.
- This is equivalent to say, “there are more than one leftmost derivation for a string, hence the grammar is ambiguous”.
- Similarly, with rightmost derivation

- Two parse trees for the yield $a+b^*a$

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow a$
$E \rightarrow b$



Can we remove the ambiguity?

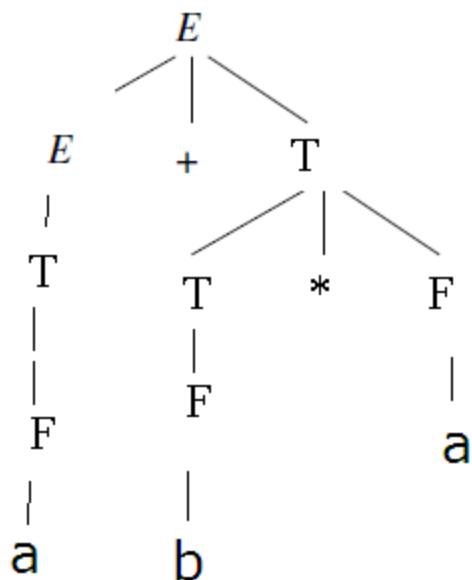
- Finding whether a given CFG is ambiguous or not is an undecidable problem !
- There are some CFLs for which it is impossible to have an unambiguous CFG.

Can we remove the ambiguity?

- Finding whether a given CFG is ambiguous or not is an undecidable problem !
- There are some CFLs for which it is impossible to have an unambiguous CFG.
- But, the situation is not so unpromising.
- For many situations in practice, we can handcraft unambiguous CFG for a given ambiguous one.

$$E \rightarrow T \mid E + T$$
$$T \rightarrow F \mid T * F$$
$$F \rightarrow a \mid b$$

An unambiguous expression grammar



This is the only parse tree for $a+b^*a$

- For an expression grammar, injecting precedence and associativity of operators can make them unambiguous.

1.	E	\rightarrow	I
2.	E	\rightarrow	$E + E$
3.	E	\rightarrow	$E * E$
4.	E	\rightarrow	(E)
5.	I	\rightarrow	a
6.	I	\rightarrow	b
7.	I	\rightarrow	Ia
8.	I	\rightarrow	Ib
9.	I	\rightarrow	$I0$
10.	I	\rightarrow	$I1$

$$\begin{array}{lcl} E & \rightarrow & T \mid E + T \\ T & \rightarrow & F \mid T * F \\ F & \rightarrow & I \mid (E) \\ I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \end{array}$$

Figure 5.19: An unambiguous expression grammar

Figure 5.2: A context-free grammar for simple expressions

$$\begin{array}{lcl}
 I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\
 F & \rightarrow & I \mid (E) \\
 T & \rightarrow & F \mid T * F \\
 E & \rightarrow & T \mid E + T
 \end{array}$$

Figure 5.19: An unambiguous expression grammar

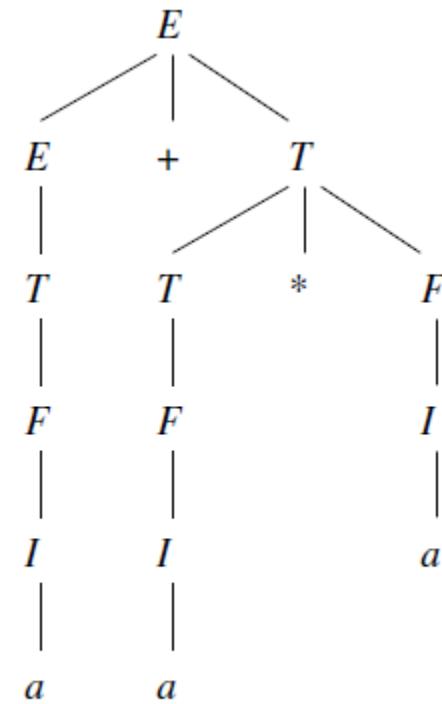


Figure 5.20: The sole parse tree for $a + a * a$

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$

5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

Figure 5.2: A context-free grammar for simple expressions

- With above we get two parse trees for the same yield.

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \rightarrow & I \mid (E) \\ T & \rightarrow & F \mid T * F \\ E & \rightarrow & T \mid E + T \end{array}$$

Figure 5.19: An unambiguous expression grammar

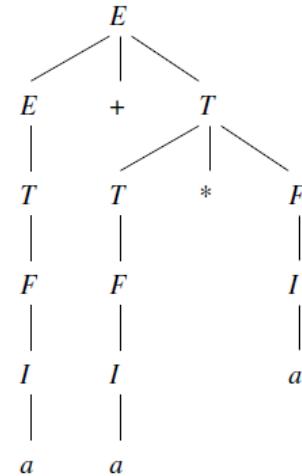


Figure 5.20: The sole parse tree for $a + a * a$

Inherent ambiguity

- A CFL is said to be inherently ambiguous, if every CFG that generates the language is ambiguous.
- Note, in this case we say CFL is ambiguous.
 - Earlier we said CFG is ambiguous.

An example of ambiguous CFL

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

That is, L consists of strings in $\mathbf{a}^+ \mathbf{b}^+ \mathbf{c}^+ \mathbf{d}^+$ such that either:

1. There are as many a 's as b 's and as many c 's as d 's, or
2. There are as many a 's as d 's and as many b 's as c 's.

An example of ambiguous CFL

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

That is, L consists of strings in $\mathbf{a}^+ \mathbf{b}^+ \mathbf{c}^+ \mathbf{d}^+$ such that either:

1. There are as many a 's as b 's and as many c 's as d 's, or
2. There are as many a 's as d 's and as many b 's as c 's.

$$\begin{array}{lcl} S & \rightarrow & AB \mid C \\ A & \rightarrow & aAb \mid ab \\ B & \rightarrow & cBd \mid cd \\ C & \rightarrow & aCd \mid aDd \\ D & \rightarrow & bDc \mid bc \end{array}$$

One CFG, for the CFL

An example of ambiguous CFL

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

That is, L consists of strings in $\mathbf{a}^+ \mathbf{b}^+ \mathbf{c}^+ \mathbf{d}^+$ such that either:

1. There are as many a 's as b 's and as many c 's as d 's, or
2. There are as many a 's as d 's and as many b 's as c 's.

- | | |
|--|--|
| 1. $S \xrightarrow[lm]{} AB \xrightarrow[lm]{} aAbB \xrightarrow[lm]{} aabbB \xrightarrow[lm]{} aabbcBd \xrightarrow[lm]{} aabbccdd$ | 2. $S \xrightarrow[lm]{} C \xrightarrow[lm]{} aCd \xrightarrow[lm]{} aaDdd \xrightarrow[lm]{} aabDcdd \xrightarrow[lm]{} aabbccdd$ |
|--|--|

Two leftmost derivations for the same string

$$\begin{array}{lcl} S & \rightarrow & AB \mid C \\ A & \rightarrow & aAb \mid ab \\ B & \rightarrow & cBd \mid cd \\ C & \rightarrow & aCd \mid aDd \\ D & \rightarrow & bDc \mid bc \end{array}$$

One CFG, for the CFL

$$\begin{array}{lcl}
 S & \rightarrow & AB \mid C \\
 A & \rightarrow & aAb \mid ab \\
 B & \rightarrow & cBd \mid cd \\
 C & \rightarrow & aCd \mid aDd \\
 D & \rightarrow & bDc \mid bc
 \end{array}$$

1. $S \xrightarrow{lm} AB \xrightarrow{lm} aAbB \xrightarrow{lm} aabbB \xrightarrow{lm} aabbcBd \xrightarrow{lm} aabbccdd$
2. $S \xrightarrow{lm} C \xrightarrow{lm} aCd \xrightarrow{lm} aaDdd \xrightarrow{lm} aabDcdd \xrightarrow{lm} aabbccdd$

One CFG, for the CFL

Two leftmost derivations for the same string

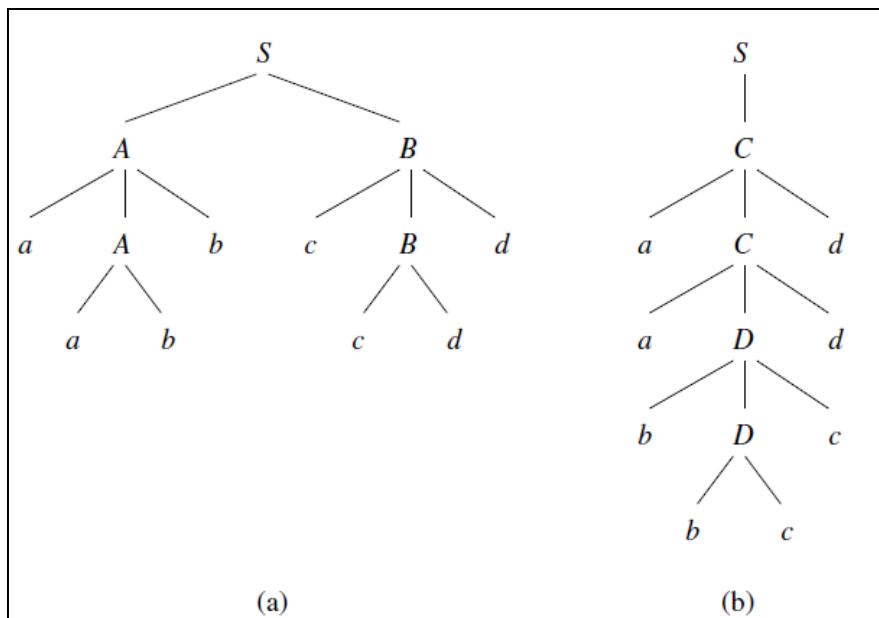


Figure 5.23: Two parse trees for *aabbccdd*

How to understand that every CFG is ambiguous.

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

That is, L consists of strings in $\mathbf{a}^+ \mathbf{b}^+ \mathbf{c}^+ \mathbf{d}^+$ such that either:

1. There are as many a 's as b 's and as many c 's as d 's, or
2. There are as many a 's as d 's and as many b 's as c 's.

- Proof is complicated.
- But the essence is, the grammar has two parts one generating strings in each of the above union.
- There are some strings that are common between these two parts. These can be generated from two ways.

* **Exercise 5.4.1:** Consider the grammar

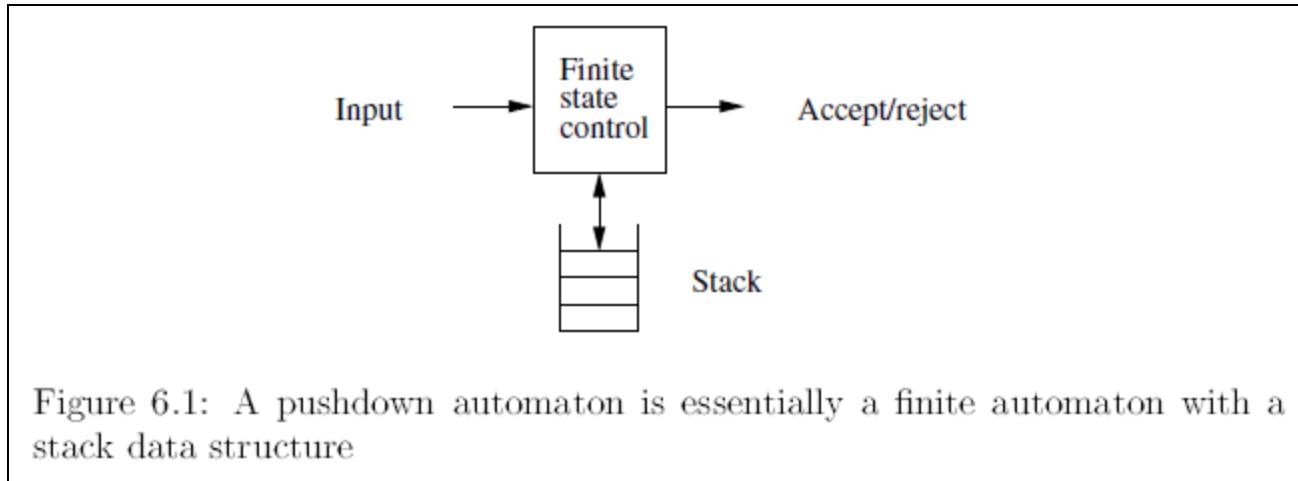
$$S \rightarrow aS \mid aSbS \mid \epsilon$$

This grammar is ambiguous. Show in particular that the string aab has two:

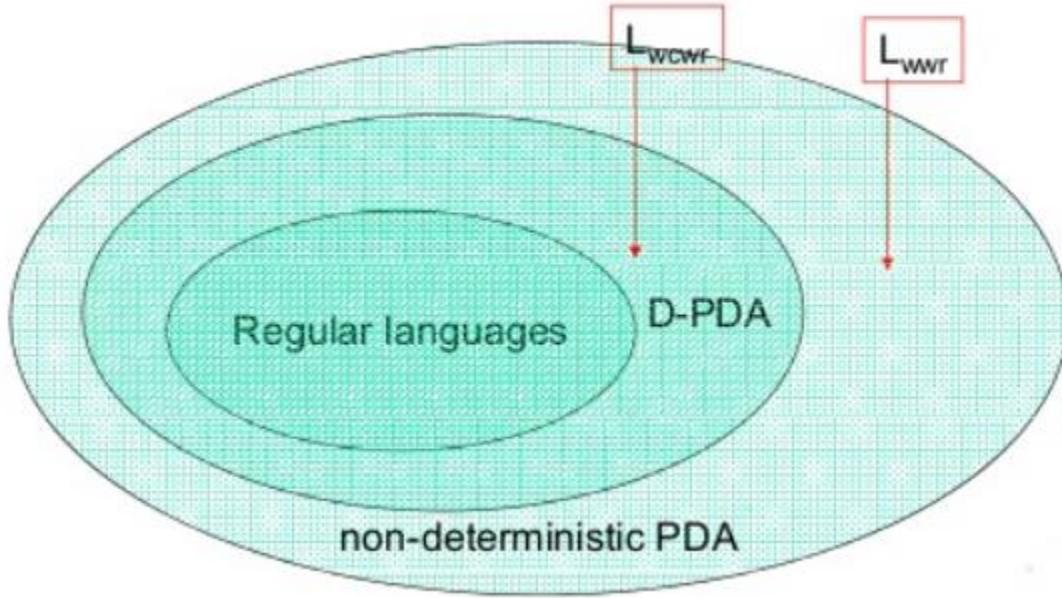
- a) Parse trees.
- b) Leftmost derivations.
- c) Rightmost derivations.

Chapter 6

Pushdown Automata



- PDA is an extension of nondeterministic finite automaton with a stack (of infinite size).
- DPDA -- deterministic version of PDA is not enough to recognize CFLs.



It holds that:

$$\left\{ \begin{array}{l} \text{Deterministic} \\ \text{Context-Free} \\ \text{Languages} \\ (\text{DPDA}) \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Context-Free} \\ \text{Languages} \\ \text{PDAs} \end{array} \right\}$$

Since every DPDA is also a PDA

6.1.2 The Formal Definition of Pushdown Automata

Our formal notation for a *pushdown automaton* (PDA) involves seven components. We write the specification of a PDA P as follows:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

The components have the following meanings:

Q : A finite set of *states*, like the states of a finite automaton.

Σ : A finite set of *input symbols*, also analogous to the corresponding component of a finite automaton.

Γ : A finite *stack alphabet*. This component, which has no finite-automaton analog, is the set of symbols that we are allowed to push onto the stack.

δ : The *transition function*.

q_0 : The *start state*. The PDA is in this state before making any transitions.

Z_0 : The *start symbol*. Initially, the PDA's stack consists of one instance of this symbol, and nothing else.

F : The set of *accepting states*, or *final states*.

δ : The *transition function*. As for a finite automaton, δ governs the behavior of the automaton. Formally, δ takes as argument a triple $\delta(q, a, X)$, where:

1. q is a state in Q .
2. a is either an input symbol in Σ or $a = \epsilon$, the empty string, which is assumed not to be an input symbol.
3. X is a stack symbol, that is, a member of Γ .

The output of δ is a finite set of pairs (p, γ) , where p is the new state, and γ is the string of stack symbols that replaces X at the top of the stack. For instance, if $\gamma = \epsilon$, then the stack is popped, if $\gamma = X$, then the stack is unchanged, and if $\gamma = YZ$, then X is replaced by Z , and Y is pushed onto the stack.

- $\delta(q, a, X) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots\}$ where $\gamma_i \in \Gamma^*$.
 - Note, it is possible that $\gamma_i = \epsilon$.
 - $p_i \in Q$.
- $a \in \Sigma \cup \{\epsilon\}$.
- $X \in \Gamma$
 - X can never be ϵ . PDA must read a symbol from stack.

(p , γ)

If $\gamma = YZ$, then Y will be on the top of the stack.

If $\gamma = \epsilon$, then do not push anything on to the stack.

L_{WWR}

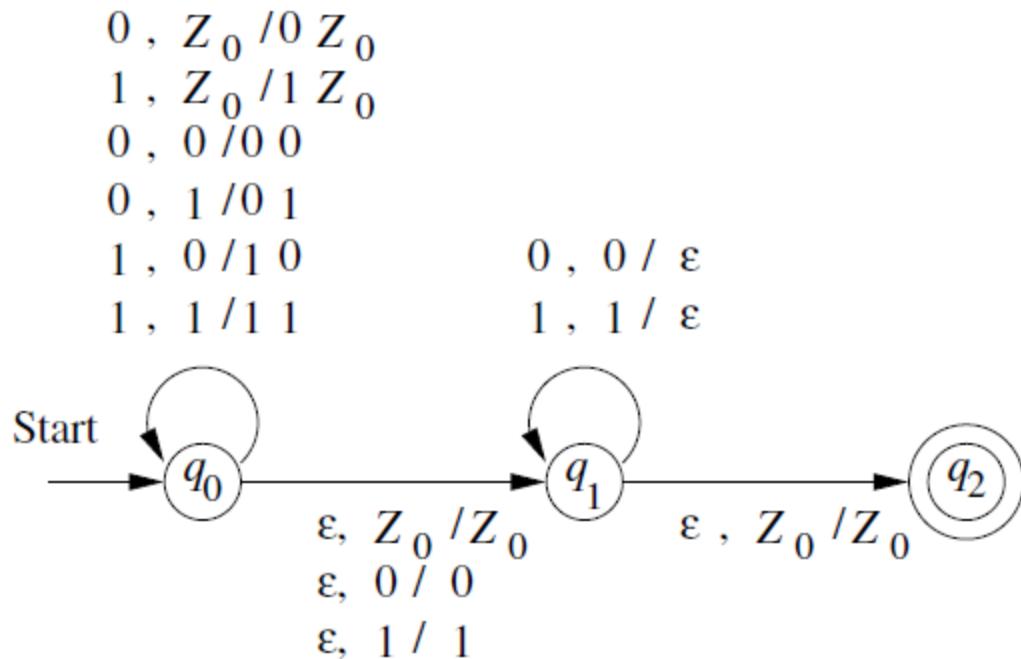


Figure 6.2: Representing a PDA as a generalized transition diagram

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

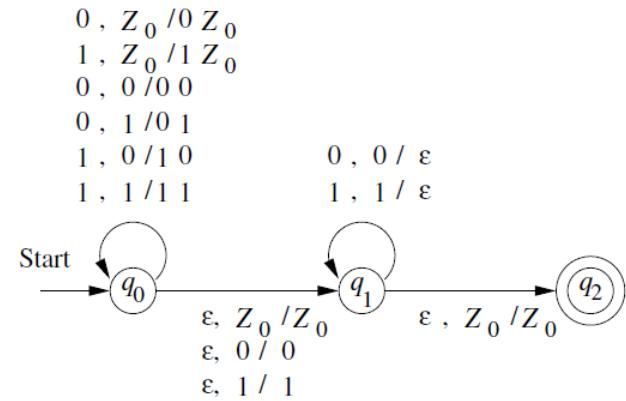


Figure 6.2: Representing a PDA as a generalized transition diagram

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$ and $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$. One of these rules applies initially, when we are in state q_0 and we see the start symbol Z_0 at the top of the stack. We read the first input, and push it onto the stack, leaving Z_0 below to mark the bottom.
2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$, $\delta(q_0, 1, 0) = \{(q_0, 10)\}$, and $\delta(q_0, 1, 1) = \{(q_0, 11)\}$. These four, similar rules allow us to stay in state q_0 and read inputs, pushing each onto the top of the stack and leaving the previous top stack symbol alone.
3. $\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$, $\delta(q_0, \epsilon, 0) = \{(q_1, 0)\}$, and $\delta(q_0, \epsilon, 1) = \{(q_1, 1)\}$. These three rules allow P to go from state q_0 to state q_1 spontaneously (on ϵ input), leaving intact whatever symbol is at the top of the stack.
4. $\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$, and $\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$. Now, in state q_1 we can match input symbols against the top symbols on the stack, and pop when the symbols match.
5. $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$. Finally, if we expose the bottom-of-stack marker Z_0 and we are in state q_1 , then we have found an input of the form ww^R . We go to state q_2 and accept.

6.1.4 Instantaneous Descriptions of a PDA

we shall represent the configuration of a PDA by a triple (q, w, γ) , where

1. q is the state,
2. w is the remaining input, and
3. γ is the stack contents.

One step, and several steps

Suppose $\delta(q, a, X)$ contains (p, α) .

Then for all strings w in Σ^* and β in Γ^* : $(q, aw, X\beta) \vdash (p, w, \alpha\beta)$

\vdash^* is reflexive and transitive closure of \vdash

6.2 The Languages of a PDA

6.2.1 Acceptance by Final State

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then $L(P)$, the *language accepted by P by final state*, is

$$\{w \mid (q_0, w, Z_0) \stackrel{P}{\vdash}^* (q, \epsilon, \alpha)\}$$

for some state q in F and any stack string α .

6.2 The Languages of a PDA

6.2.1 Acceptance by Final State

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then $L(P)$, the *language accepted by P by final state*, is

$$\{w \mid (q_0, w, Z_0) \stackrel{*}{\vdash}_P (q, \epsilon, \alpha)\}$$

for some state q in F and any stack string α .

That is, starting in the initial ID with w waiting on the input, P consumes w from the input and enters an accepting state. The contents of the stack at that time is irrelevant.

6.2 The Languages of a PDA

6.2.1 Acceptance by Final State

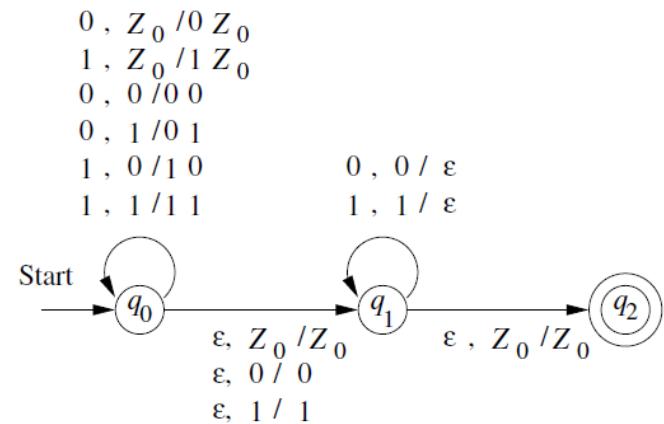
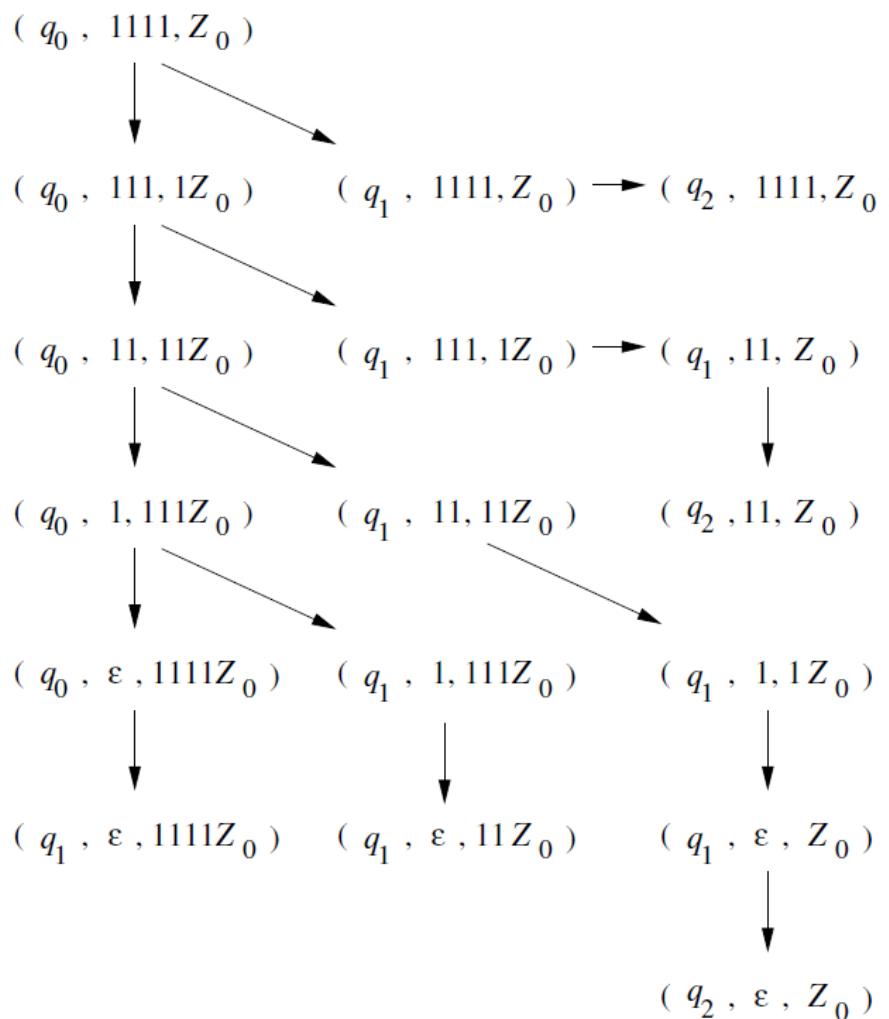
Do we have some other acceptance criterion?!

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then $L(P)$, the *language accepted by P by final state*, is

$$\{w \mid (q_0, w, Z_0) \stackrel{P}{\vdash}^* (q, \epsilon, \alpha)\}$$

for some state q in F and any stack string α .

That is, starting in the initial ID with w waiting on the input, P consumes w from the input and enters an accepting state. The contents of the stack at that time is irrelevant.



Representing a PDA as a generalized transition diagram

Figure 6.3: ID's of the PDA of Example 6.2 on input 1111

$$(q_0, 1111, Z_0) \vdash (q_0, 111, 1Z_0) \vdash (q_0, 11, 11Z_0) \vdash (q_1, 11, 11Z_0) \vdash (q_1, 1, 1Z_0) \vdash (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0).$$

$$(q_0, 1111, Z_0) \vdash (q_0, 111, 1Z_0) \vdash (q_0, 11, 11Z_0) \vdash (q_1, 11, 11Z_0) \vdash$$
$$(q_1, 1, 1Z_0) \vdash (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0).$$

- This is same as

$$(q_0, 1111, Z_0) \vdash^* (q_2, \epsilon, Z_0).$$

- So, 1111 is in the language.

6.2 The Languages of a PDA

6.2.1 Acceptance by Final State

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then $L(P)$, the *language accepted by P by final state*, is

$$\{w \mid (q_0, w, Z_0) \stackrel{*}{\vdash}_P (q, \epsilon, \alpha)\}$$

for some state q in F and any stack string α .

6.2.2 Acceptance by Empty Stack

For each PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, we also define

$$N(P) = \{w \mid (q_0, w, Z_0) \stackrel{*}{\vdash} (q, \epsilon, \epsilon)\}$$

for any state q . That is, $N(P)$ is the set of inputs w that P can consume and at the same time empty its stack.²

$L(P)$ Vs. $N(P)$

- For the same PDA P there are now two languages !!

0 ,	$Z_0 / 0 Z_0$
1 ,	$Z_0 / 1 Z_0$
0 ,	$0 / 0 0$
0 ,	$1 / 0 1$
1 ,	$0 / 1 0$
1 ,	$1 / 1 1$

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

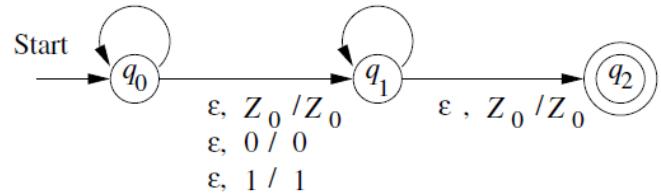


Figure 6.2: Representing a PDA as a generalized transition diagram

- $L(P) = \{ww^R \mid w \in (0+1)^*\} = L_{w wr}$
- $N(P) = ?$

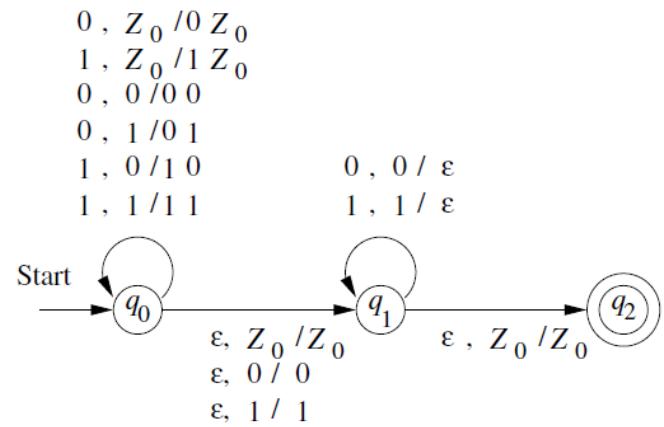


Figure 6.2: Representing a PDA as a generalized transition diagram

- $L(P) = \{ww^R \mid w \in (0 + 1)^*\} = L_{w wr}$
- $N(P) = \phi$
 - Stack never becomes empty
 - But, a small change can make $N(P) = L(P)$.
 - What is that?

$0, Z_0 / 0 Z_0$

$1, Z_0 / 1 Z_0$

$0, 0 / 0 0$

$0, 1 / 0 1$

$1, 0 / 1 0$

$1, 1 / 1 1$

$0, 0 / \epsilon$

$1, 1 / \epsilon$

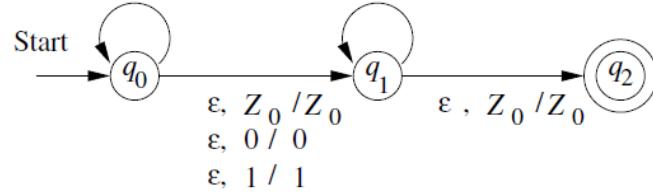


Figure 6.2: Representing a PDA as a generalized transition diagram



$0, Z_0 / 0 Z_0$

$1, Z_0 / 1 Z_0$

$0, 0 / 0 0$

$0, 1 / 0 1$

$1, 0 / 1 0$

$1, 1 / 1 1$

$0, 0 / \epsilon$

$1, 1 / \epsilon$

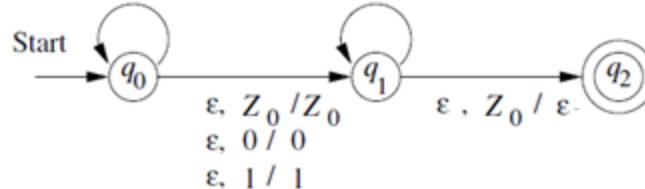


Figure 6.2(a) A modified PDA.

Acceptance by empty stack

Since the set of accepting states is irrelevant, we shall sometimes leave off the last (seventh) component from the specification of a PDA P .

Acceptance by empty stack

Since the set of accepting states is irrelevant, we shall sometimes leave off the last (seventh) component from the specification of a PDA P .

Thus, P can be written as a six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$.

Do we have two types of PDAs?

- One with final states, other with empty stack

...

Do we have two types of PDAs?

- One with final states, other with empty stack
- ...
- NO.

Do we have two types of PDAs?

- One with final states, other with empty stack
- ...
- NO.
- But, for any PDA there are two languages (both are CFLs) associated with that PDA.
 - These two (languages) may or may not be same.

Crucial thing is...

- Set of languages accepted by final state is equal to the set of languages accepted by empty stack.
 - Proof of this one is by construction.

Crucial thing is...

- Set of languages accepted by final state is equal to the set of languages accepted by empty stack.
 - Proof of this one is by construction.
- So, power of PDA is same whether recognition happens either by final state or by empty stack.

6.2.3 From Empty Stack to Final State

Theorem 6.9: If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, then there is a PDA P_F such that $L = L(P_F)$.

6.2.3 From Empty Stack to Final State

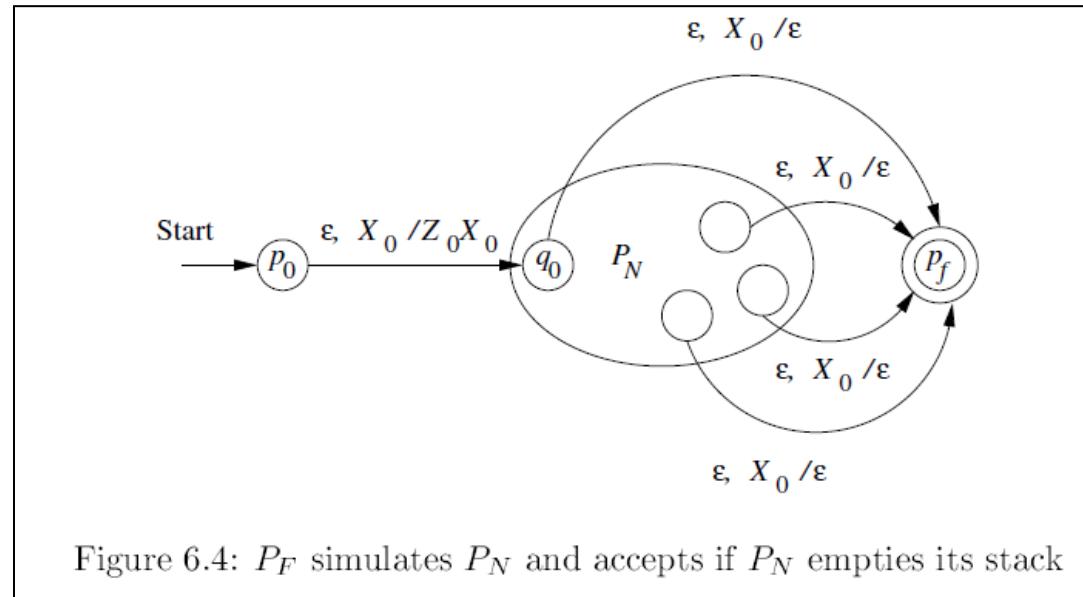
Theorem 6.9: If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, then there is a PDA P_F such that $L = L(P_F)$.

PROOF: The idea behind the proof is in Fig. 6.4. We use a new symbol X_0 , which must not be a symbol of Γ ; X_0 is both the start symbol of P_F and a marker on the bottom of the stack that lets us know when P_N has reached an empty stack.

6.2.3 From Empty Stack to Final State

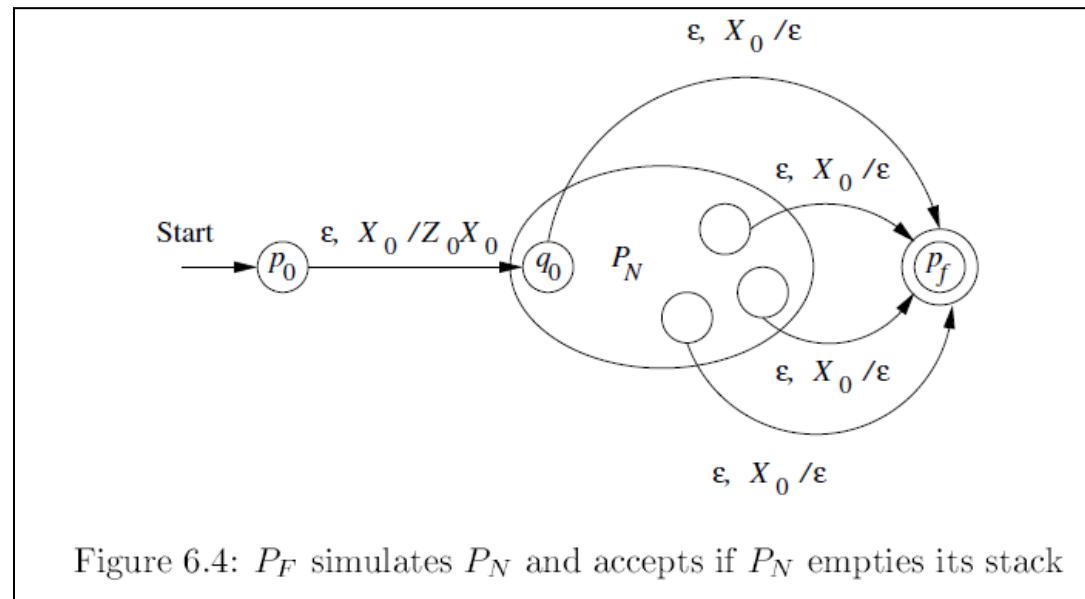
Theorem 6.9: If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, then there is a PDA P_F such that $L = L(P_F)$.

PROOF: The idea behind the proof is in Fig. 6.4. We use a new symbol X_0 , which must not be a symbol of Γ ; X_0 is both the start symbol of P_F and a marker on the bottom of the stack that lets us know when P_N has reached an empty stack.



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

1. $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0 X_0)\}$. In its start state, P_F makes a spontaneous transition to the start state of P_N , pushing its start symbol Z_0 onto the stack.
2. For all states q in Q , inputs a in Σ or $a = \epsilon$, and stack symbols Y in Γ , $\delta_F(q, a, Y)$ contains all the pairs in $\delta_N(q, a, Y)$.
3. In addition to rule (2), $\delta_F(q, \epsilon, X_0)$ contains (p_f, ϵ) for every state q in Q .



6.2.4 From Final State to Empty Stack

Theorem 6.11: Let L be $L(P_F)$ for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Then there is a PDA P_N such that $L = N(P_N)$.

6.2.4 From Final State to Empty Stack

Theorem 6.11: Let L be $L(P_F)$ for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Then there is a PDA P_N such that $L = N(P_N)$.

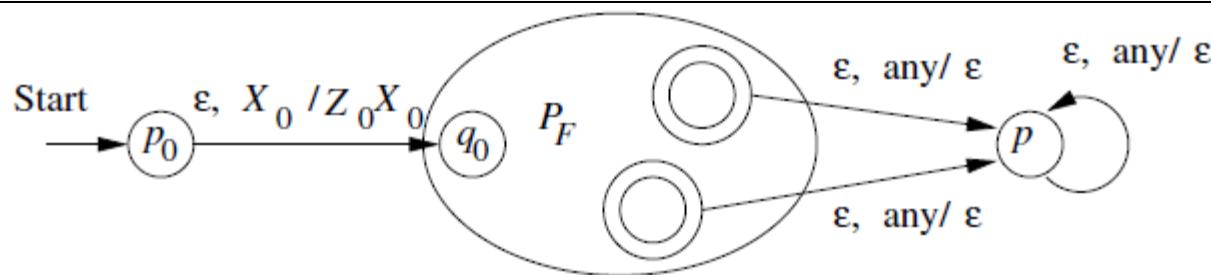


Figure 6.7: P_N simulates P_F and empties its stack when and only when P_N enters an accepting state

6.2.4 From Final State to Empty Stack

Theorem 6.11: Let L be $L(P_F)$ for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Then there is a PDA P_N such that $L = N(P_N)$.

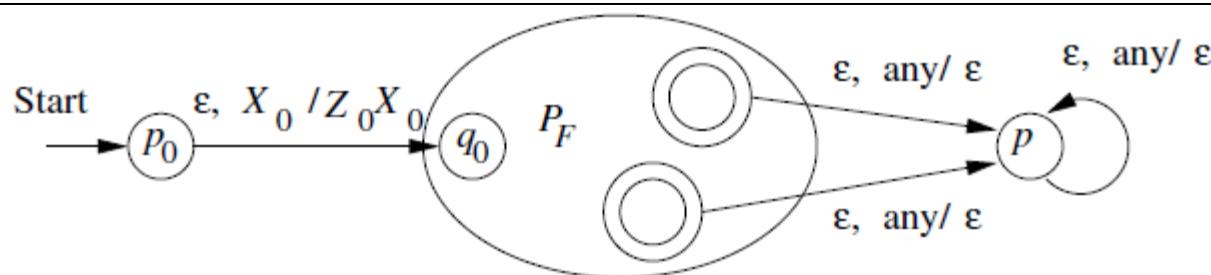


Figure 6.7: P_N simulates P_F and empties its stack when and only when P_N enters an accepting state

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

6.2.4 From Final State to Empty Stack

Theorem 6.11: Let L be $L(P_F)$ for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Then there is a PDA P_N such that $L = N(P_N)$.

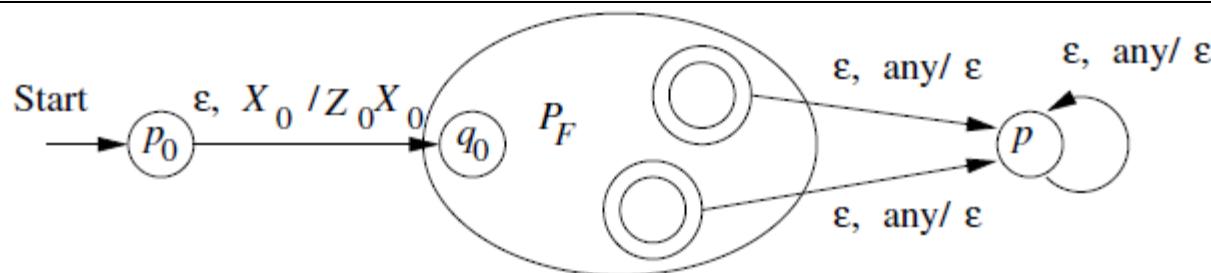


Figure 6.7: P_N simulates P_F and empties its stack when and only when P_N enters an accepting state

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

To avoid simulating a situation where P_F accidentally empties its stack without accepting, P_N must also use a marker X_0 on the bottom of its stack.

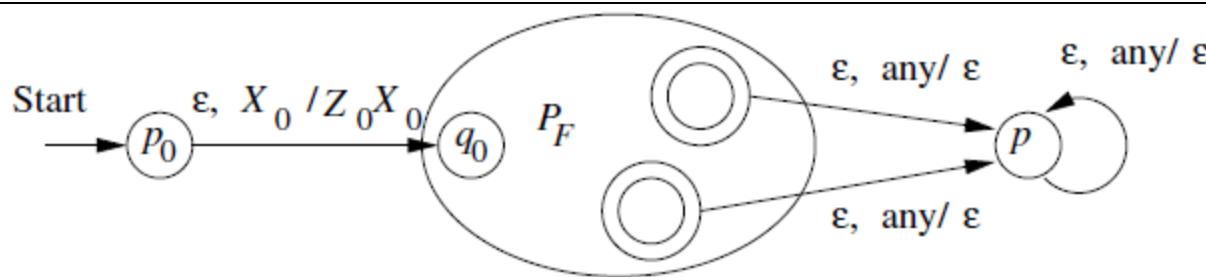


Figure 6.7: P_N simulates P_F and empties its stack when and only when P_N enters an accepting state

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

- $\delta_N(p_0, \epsilon, X_0) = \{(q_0, Z_0 X_0)\}$. We start by pushing the start symbol of P_F onto the stack and going to the start state of P_F .
- For all states q in Q , input symbols a in Σ or $a = \epsilon$, and Y in Γ , $\delta_N(q, a, Y)$ contains every pair that is in $\delta_F(q, a, Y)$. That is, P_N simulates P_F .
- For all accepting states q in F and stack symbols Y in Γ or $Y = X_0$, $\delta_N(q, \epsilon, Y)$ contains (p, ϵ) . By this rule, whenever P_F accepts, P_N can start emptying its stack without consuming any more input.
- For all stack symbols Y in Γ or $Y = X_0$, $\delta_N(p, \epsilon, Y) = \{(p, \epsilon)\}$. Once in state p , which only occurs when P_F has accepted, P_N pops every symbol on its stack, until the stack is empty. No further input is consumed.

One point ..

- In P_F if stack becomes empty (in between) then P_F gets stuck. (That PDA gets killed).

One point ..

- In P_F if stack becomes empty (in between) then P_F gets stuck. (That PDA gets killed).
- Now, in this situation P_N has X_0 in the stack,
so will this be a problem?

One point ..

- In P_F if stack becomes empty (in between) then P_F gets stuck. (That PDA gets killed).
- Now, in this situation P_N has X_0 in the stack,
so will this be a problem?
- If the state is a final state no problem.
- Otherwise, if the state is a non-final one, do we need to do something?

One point ..

- In P_F if stack becomes empty (in between) then P_F gets stuck. (That PDA gets killed).
- Now, in this situation P_N has X_0 in the stack,
so will this be a problem?
- If the state is a final state no problem.
- Otherwise, if the state is a non-final one, do we need to do something? **NO.**
- Since there is no transition (on X_0 being stack top) that PDA gets killed.

Next ...

- Equivalence of PDA's and CFG's

Next ...

- Equivalence of PDA's and CFG's

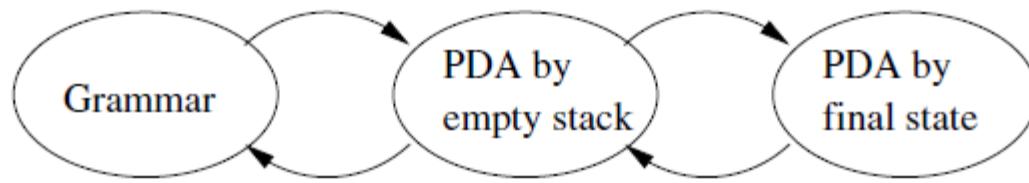


Figure 6.8: Organization of constructions showing equivalence of three ways of defining the CFL's

CFG to PDA

Let $G = (V, T, Q, S)$ be a CFG.

CFG to PDA

Let $G = (V, T, Q, S)$ be a CFG.

Construct the PDA P that accepts $L(G)$ by empty stack as follows:

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

CFG to PDA

Let $G = (V, T, Q, S)$ be a CFG.

Construct the PDA P that accepts $L(G)$ by empty stack
as follows:

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

Single state !

CFG to PDA

Let $G = (V, T, Q, S)$ be a CFG.

Construct the PDA P that accepts $L(G)$ by empty stack as follows:

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

where transition function δ is defined by:

1. For each variable A ,

$$\delta(q, \epsilon, A) = \{(q, \beta) \mid A \rightarrow \beta \text{ is a production of } G\}$$

2. For each terminal a , $\delta(q, a, a) = \{(q, \epsilon)\}$.

Example 6.12: Let us convert the expression grammar of Fig. 5.2 to a PDA.
Recall this grammar is:

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

$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

- Can you identify (V, T, Q, S) of this CFG?

Example 6.12: Let us convert the expression grammar of Fig. 5.2 to a PDA.
Recall this grammar is:

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

$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

- Can you identify (V,T,Q,S) of this CFG?
- $V = \{E, I\}$
- $T = \{a, b, 0, 1, +, *, (,)\}$
- $S = E$

$$E \rightarrow I \mid E * E \mid E + E \mid (E)$$
$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

- For the PDA stack alphabet is
 $\Gamma = \{E, I, a, b, 0, 1, +, *, (,)\}$
- Start symbol of the stack is E .
- We will have only one state, call it q .

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

$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

- For the PDA stack alphabet is
 $\Gamma = \{E, I, a, b, 0, 1, +, *, (,)\}$
- Start symbol of the stack is E .
- We will have only one state, call it q .

$$\delta(q, \epsilon, E) = \{(q, I), (q, E + E), (q, E * E), (q, (E))\}.$$

$$\delta(q, \epsilon, I) = \{(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)\}.$$

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

$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

- For the PDA stack alphabet is
 $\Gamma = \{E, I, a, b, 0, 1, +, *, (), ()\}$
- Start symbol of the stack is E .
- We will have only one state, call it q .

$$\delta(q, \epsilon, E) = \{(q, I), (q, E + E), (q, E * E), (q, (E))\}.$$

$$\delta(q, \epsilon, I) = \{(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)\}.$$

$$\begin{aligned}\delta(q, a, a) &= \{(q, \epsilon)\}; \quad \delta(q, b, b) = \{(q, \epsilon)\}; \quad \delta(q, 0, 0) = \{(q, \epsilon)\}; \quad \delta(q, 1, 1) = \\ &\{(q, \epsilon)\}; \quad \delta(q, (, ()) = \{(q, \epsilon)\}; \quad \delta(q, (),)) = \{(q, \epsilon)\}; \quad \delta(q, +, +) = \{(q, \epsilon)\}; \\ &\delta(q, *, *) = \{(q, \epsilon)\}.\end{aligned}$$

Have you noted?

- The PDA we constructed simulates, which derivation?

Have you noted?

- The PDA we constructed simulates, which derivation?
- It is “**left-most derivation**”

Have you noted?

- The PDA we constructed simulates, which derivation?
- It is “left-most derivation”
- In compilers, these are top-down parsers
 - LL parsers;
 - but non-determinism is a problem.
 - Backtracking (to try the other choice)
 - Parse table (to find feasible choices; at that time)

Compilers

- We have parsers which are bottom-up
 - Which will simulate right-most derivation
 - These are called LR parsers.
-
- One notable drawback is all these parsers:

Compilers

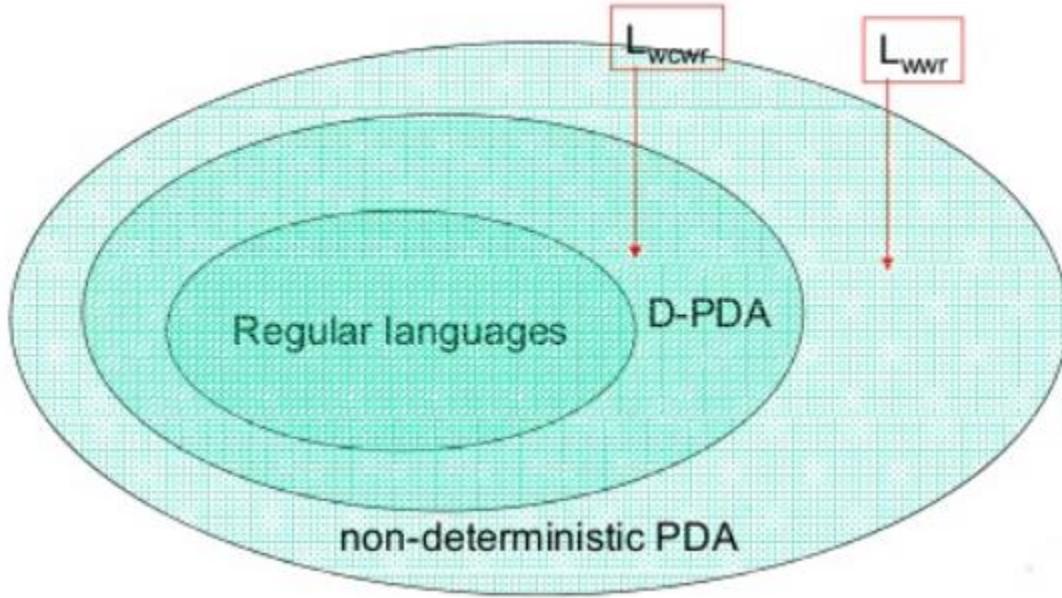
- We have parsers which are bottom-up
- Which will simulate right-most derivation
- These are called LR parsers.
- One notable drawback is all these parsers:
**each have their own limitations, and works
only for subclasses of CFLs;**
 - Some superior, some inferior ...

PDA to CFG

- We skip this in this basic course.

Deterministic PDA

- DPDA
- Can recognize a proper subset of CFLs
- Parsers (used in compilers), mostly are DPDAs.
 - Most of our programming languages are in the subclass which can be recognized by DPDAs.



It holds that:

$$\left\{ \begin{array}{l} \text{Deterministic} \\ \text{Context-Free} \\ \text{Languages} \\ (\text{DPDA}) \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Context-Free} \\ \text{Languages} \\ \text{PDAs} \end{array} \right\}$$

Since every DPDA is also a PDA

DPDA

- Remove choice.

DPDA

- Atmost one choice. But ϵ moves (should we remove them?).

DPDA

- Atmost one choice. But ϵ moves (should we remove them? **No**).
- For any $q \in Q, a \in \Sigma, or a = \epsilon, and X \in \Gamma$, we have
 - 1) $|\delta(q, a, X)| \leq 1$, and
 - 2) For any a except ϵ ,
$$|\delta(q, a, X)| = 1 \Rightarrow |\delta(q, \epsilon, X)| = 0.$$

$0, Z_0 / 0 Z_0$

$1, Z_0 / 1 Z_0$

$0, 0 / 0 0$

$0, 1 / 0 1$

$1, 0 / 1 0$

$0, 0 / \epsilon$

$1, 1 / 1 1$

$1, 1 / \epsilon$

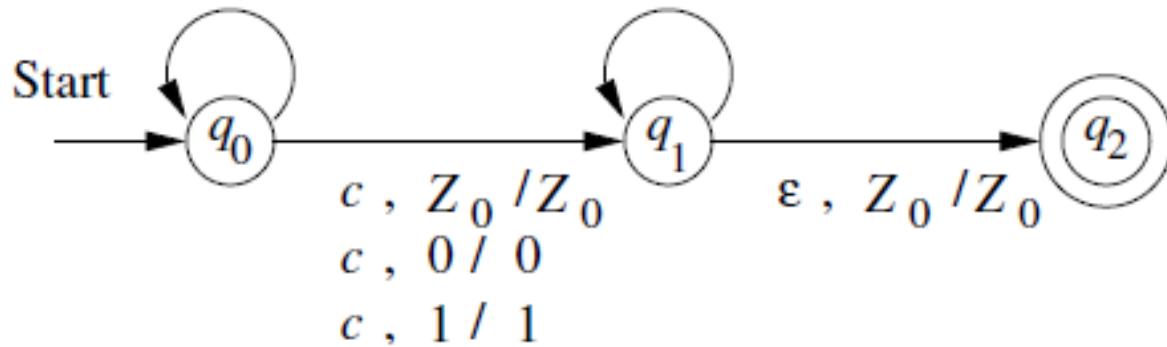


Figure 6.11: A deterministic PDA accepting L_{wcwr}

Regular Languages and DPDA

Theorem 6.17: If L is a regular language, then $L = L(P)$ for some DPDA P .

Regular Languages and DPDA

Theorem 6.17: If L is a regular language, then $L = L(P)$ for some DPDA P .

- Proof is simple.
 - DFA is there for the regular language.
 - What about stack??

Regular Languages and DPDA

Theorem 6.17: If L is a regular language, then $L = L(P)$ for some DPDA P .

- Proof is simple.
 - DFA is there for the regular language.
 - What about stack??
 - Ignore the stack.

Regular Languages and DPDA

Theorem 6.17: If L is a regular language, then $L = L(P)$ for some DPDA P .

- Proof is simple.
 - DFA is there for the regular language.
 - What about stack??
 - Ignore the stack.
- Just see, the theorem is saying about $L(P)$ only.

Formally,

let $A = (Q, \Sigma, \delta_A, q_0, F)$ be a DFA. Construct DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

Formally,

let $A = (Q, \Sigma, \delta_A, q_0, F)$ be a DFA. Construct DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

If $\delta_A(q, a) = p$ then $\delta_P(q, a, Z_0) = \{(p, Z_0)\}$ for all states p and q in Q ,

Formally,

let $A = (Q, \Sigma, \delta_A, q_0, F)$ be a DFA. Construct DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

If $\delta_A(q, a) = p$ then $\delta_P(q, a, Z_0) = \{(p, Z_0)\}$ for all states p and q in Q ,

We claim that $(q_0, w, Z_0) \vdash^* (p, \epsilon, Z_0)$ if and only if $\hat{\delta}_A(q_0, w) = p$.

DPDA and $N(P)$??

- For some regular languages, there can no DPDA by *empty stack* that recognizes the language.

DPDA and $N(P)$??

- For some regular languages, there can no DPDA by *empty stack* that recognizes the language.
- But, for a proper subset of regular languages, it is possible to build a DPDA by empty stack.
 - These are characterized by “prefix property”.
- DPDA by empty stack can recognize some non-regular languages also, provided they obey the property.

Prefix property

- A language L has the prefix property, if there are **no** two distinct strings x and y in L such that x is a prefix of y .

Prefix property

- A language L has the prefix property, if there are **no** two distinct strings x and y in L such that x is a prefix of y .
- L_{wcwr} has this property.

Prefix property

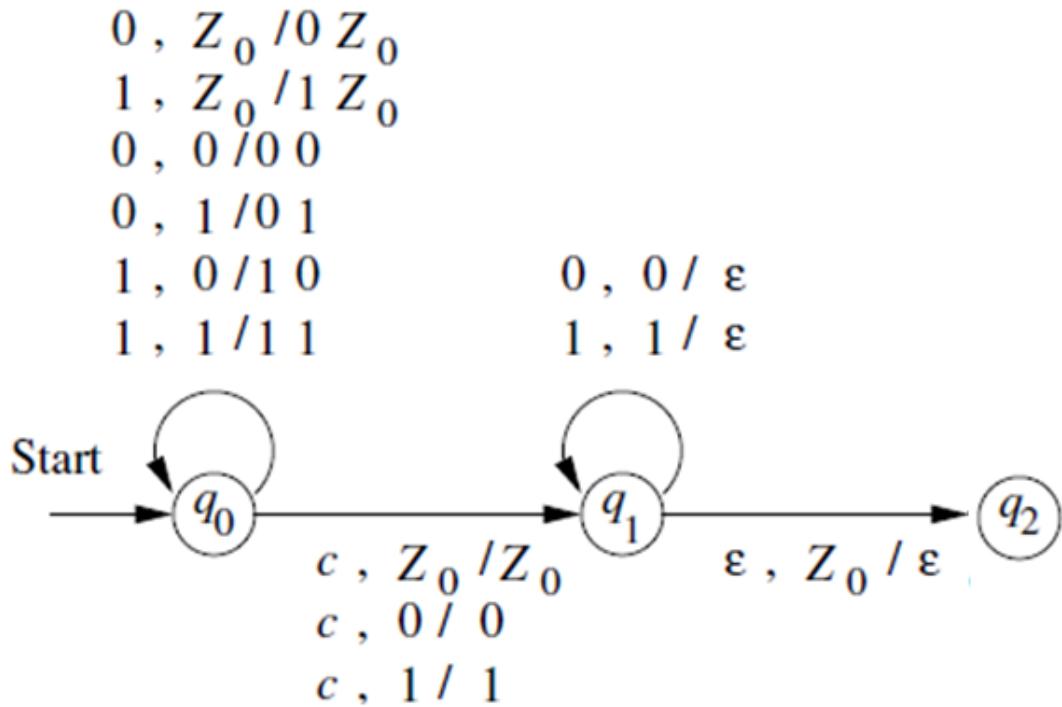
- A language L has the prefix property, if there are **no** two distinct strings x and y in L such that x is a prefix of y .
- L_{wcwr} has this property.
- 0^* violates this property. See this is regular.

Theorem 6.19: A language L is $N(P)$ for some DPDA P if and only if L has the prefix property and L is $L(P')$ for some DPDA P' . \square

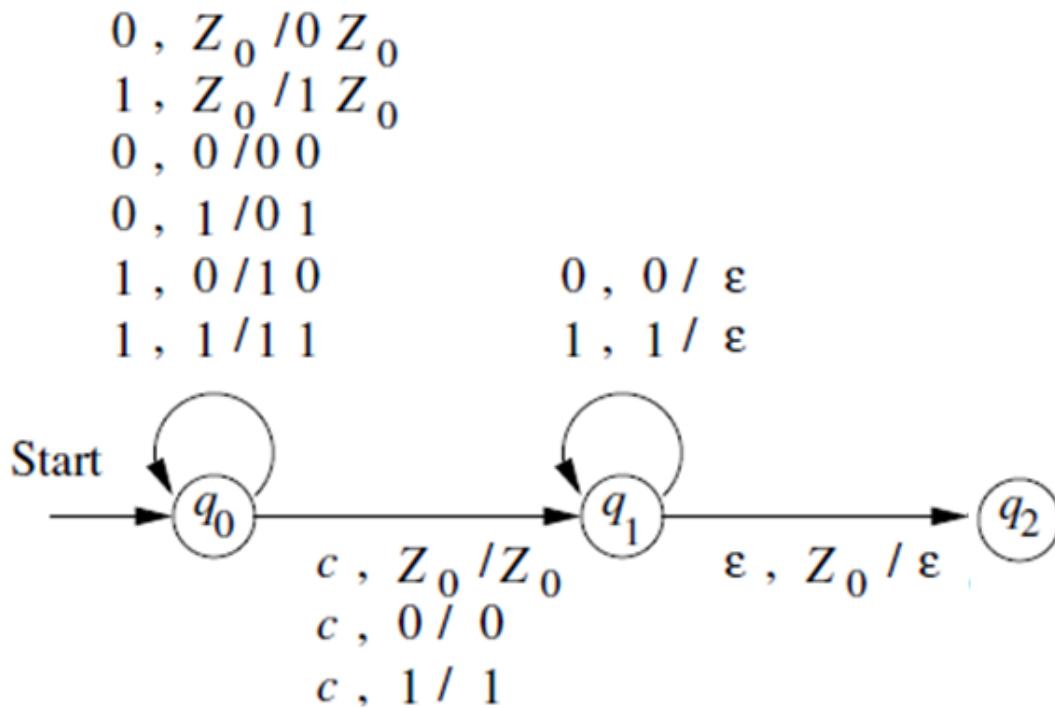
Theorem 6.19: A language L is $N(P)$ for some DPDA P if and only if L has the prefix property and L is $L(P')$ for some DPDA P' . \square

See even for 0^* (a regular language) we cannot build a DPDA by empty-stack !!

DPDA s.t. $N(P) = L_{wcwr}$

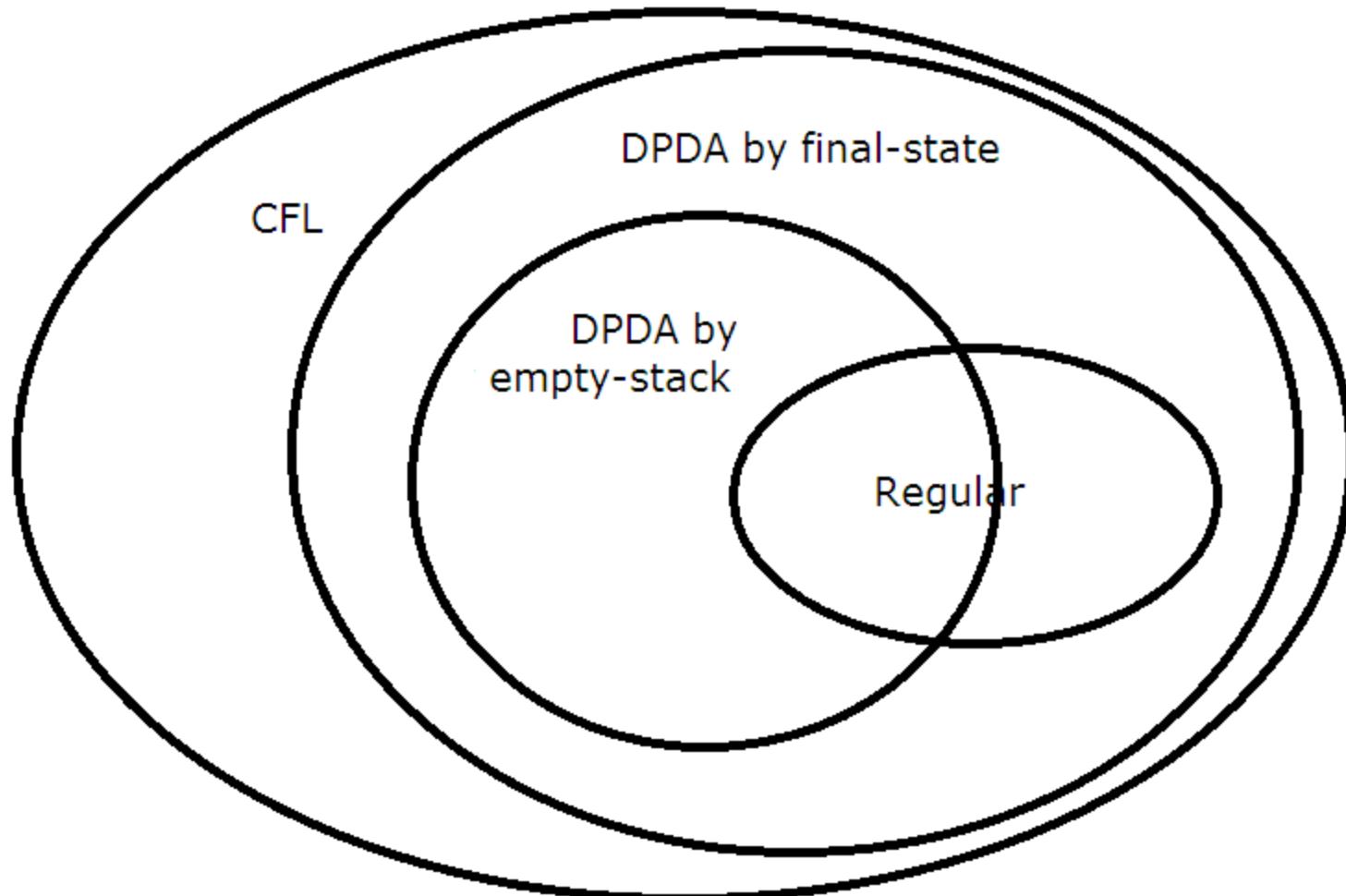


DPDA s.t. $N(P) = L_{wcwr}$



This is not a regular language.

- There is no DPDA to recognize the CFL L_{wwr}
- Proof is complex. But we can see the idea behind..



- Some points to note,

Theorem 6.20 : If $L = N(P)$ for some DPDA P , then L has an unambiguous context-free grammar.

Theorem 6.21 : If $L = L(P)$ for some DPDA P , then L has an unambiguous CFG.

- We cannot make converse of these statements.

Chapter 7

Properties of Context-Free Languages

- We first simplify CFGs
- Then we prove “pumping lemma” for CFLs
- Then closure properties and decision properties are considered.

Chomsky Normal Form (CNF)

- Every CFL (without ϵ) is generated by a CFG in which all productions are of the form

$A \rightarrow BC$ or $A \rightarrow a$

But, this requires preprocessing of the
CFG ...

- We must eliminate *useless symbols*,
- We must eliminate ϵ -*productions*, and
- We must eliminate *unit productions*.

Eliminating useless symbols

We say a symbol X is *useful* for a grammar $G = (V, T, P, S)$ if there is some derivation of the form $S \xrightarrow{*} \alpha X \beta \xrightarrow{*} w$, where w is in T^* .

- Note that X may be either a variable or a terminal symbol.
- If X is not useful, we say it is *useless*.
- This useless symbol elimination should not change the CFL. We see such a one.
 - The language is in tact, but useless are removed.

Our approach to eliminating useless symbols begins by identifying the two things a symbol has to be able to do to be useful:

Our approach to eliminating useless symbols begins by identifying the two things a symbol has to be able to do to be useful:

1. We say X is *generating* if $X \xrightarrow{*} w$ for some terminal string w . Note that every terminal is generating, since w can be that terminal itself, which is derived by zero steps.

Our approach to eliminating useless symbols begins by identifying the two things a symbol has to be able to do to be useful:

1. We say X is *generating* if $X \xrightarrow{*} w$ for some terminal string w . Note that every terminal is generating, since w can be that terminal itself, which is derived by zero steps.
2. We say X is *reachable* if there is a derivation $S \xrightarrow{*} \alpha X \beta$ for some α and β .

Order is important

1. Eliminate nongenerating symbols *first*,
2. *then*, from the remaining eliminate unreachable symbols.

Order is important

1. Eliminate nongenerating symbols *first*,
 2. *then*, from the remaining eliminate unreachable symbols.
-
- Whatever left are only useful ones.

Example 7.1: Consider the grammar:

$$\begin{array}{l} S \rightarrow AB \mid a \\ A \rightarrow b \end{array}$$

Example 7.1: Consider the grammar:

$$\begin{aligned} S &\rightarrow AB \mid a \\ A &\rightarrow b \end{aligned}$$

- Find non-generating symbols.
- How to find this?

Example 7.1: Consider the grammar:

$$\begin{aligned} S &\rightarrow AB \mid a \\ A &\rightarrow b \end{aligned}$$

- Find non-generating symbols.
- How to find this?
- We can find generating symbols, inductively.

Example 7.1: Consider the grammar:

$$\begin{aligned} S &\rightarrow AB \mid a \\ A &\rightarrow b \end{aligned}$$

- Find non-generating symbols.
- How to find this?
- We can find generating symbols, inductively.
- Basis: Every symbol of T is generating. (Why?)
- Induction: if every symbol on RHS of a production $A \rightarrow \alpha$ is generating, then A is generating.

Example 7.1: Consider the grammar:

$$\begin{aligned} S &\rightarrow AB \mid a \\ A &\rightarrow b \end{aligned}$$

- What are the generating symbols?

Example 7.1: Consider the grammar:

$$\begin{aligned} S &\rightarrow AB \mid a \\ A &\rightarrow b \end{aligned}$$

- What are the generating symbols?
- {a,b,A,S}

Example 7.1: Consider the grammar:

$$\begin{aligned} S &\rightarrow AB \mid a \\ A &\rightarrow b \end{aligned}$$

- What are the generating symbols?
- {a,b,A,S}
- So the non-generating symbol is B.

Example 7.1: Consider the grammar:

$$\begin{aligned} S &\rightarrow AB \mid a \\ A &\rightarrow b \end{aligned}$$

- What are the generating symbols?
- {a,b,A,S}
- So the non-generating symbol is B.
- So removing B we are left with $S \rightarrow a$, $A \rightarrow b$

Theorem 7.4: The algorithm above finds all and only the generating symbols of G .

- Proof is skipped.

Finding reachable symbols

- By induction, again.
- Basis. S is reachable.
- Induction. A is reachable, and $A \rightarrow \alpha$, then all symbols in α are reachable.

Example 7.1: Consider the grammar:

$$\begin{array}{l} S \rightarrow AB \mid a \\ A \rightarrow b \end{array}$$

- Find reachable.

Example 7.1: Consider the grammar:

$$\begin{array}{l} S \rightarrow AB \mid a \\ A \rightarrow b \end{array}$$

- Find reachable.
- $\{S, A, B, a, b\}$

As per order, to remove useless

$$\begin{array}{l} S \rightarrow AB \mid a \\ A \rightarrow b \end{array}$$

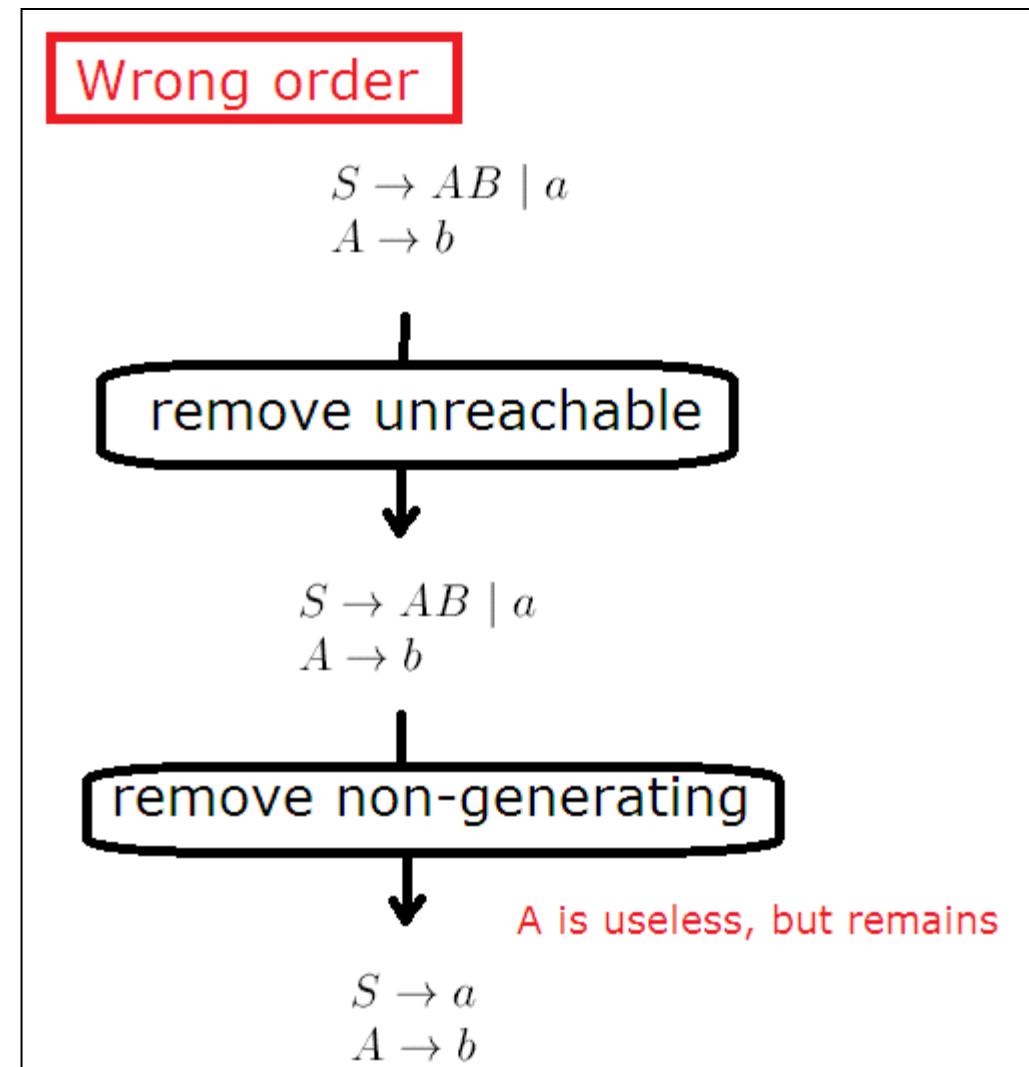
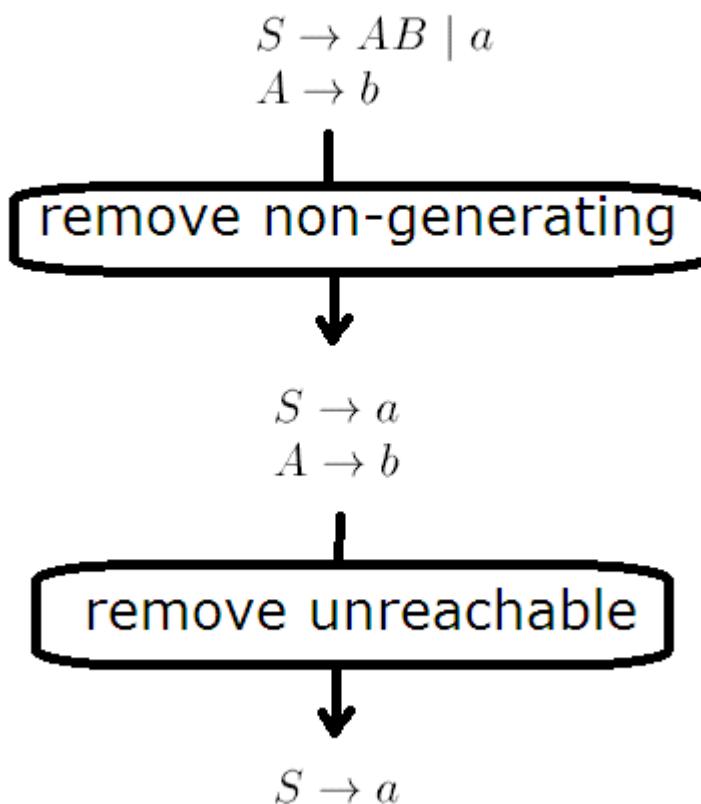
remove non-generating

$$\begin{array}{l} S \rightarrow a \\ A \rightarrow b \end{array}$$

remove unreachable

$$S \rightarrow a$$

As per order, to remove useless



Eliminating ϵ -productions

- $A \rightarrow \epsilon$ is an ϵ -production
 - Let L be a CFL.
 - For $L - \{\epsilon\}$ there is a CFG which is without ϵ -productions

Eliminating ϵ -productions

- Discover *nullable* variables.
 - A variable A is nullable if $A \xrightarrow{*} \epsilon$
 - In this case, replace $B \rightarrow CAD$ by $B \rightarrow CAD|CD$

Eliminating ϵ -productions

- Discover *nullable* variables.
 - A variable A is nullable if $A \xrightarrow{*} \epsilon$
 - In this case, replace $B \rightarrow CAD$ by $B \rightarrow CAD|CD$
- **Why this correction is needed??**

Finding the nullable

Let $G = (V, T, P, S)$ be a CFG. We can find all the nullable symbols of G by the following iterative algorithm.

Finding the nullable

Let $G = (V, T, P, S)$ be a CFG. We can find all the nullable symbols of G by the following iterative algorithm.

BASIS: If $A \rightarrow \epsilon$ is a production of G , then A is nullable.

Finding the nullable

Let $G = (V, T, P, S)$ be a CFG. We can find all the nullable symbols of G by the following iterative algorithm.

BASIS: If $A \rightarrow \epsilon$ is a production of G , then A is nullable.

INDUCTION: If there is a production $B \rightarrow C_1 C_2 \cdots C_k$, where each C_i is nullable, then B is nullable. Note that each C_i must be a variable to be nullable, so we only have to consider productions with all-variable bodies.

Finding the nullable

Let $G = (V, T, P, S)$ be a CFG. We can find all the nullable symbols of G by the following iterative algorithm.

BASIS: If $A \rightarrow \epsilon$ is a production of G , then A is nullable.

INDUCTION: If there is a production $B \rightarrow C_1 C_2 \cdots C_k$, where each C_i is nullable, then B is nullable. Note that each C_i must be a variable to be nullable, so we only have to consider productions with all-variable bodies.

Theorem 7.7: In any grammar G , the only nullable symbols are the variables found by the algorithm above.

Example 7.8: Consider the grammar

$$\begin{aligned}S &\rightarrow AB \\A &\rightarrow aAA \mid \epsilon \\B &\rightarrow bBB \mid \epsilon\end{aligned}$$

- Find the nullable symbols.

Example 7.8: Consider the grammar

$$\begin{aligned}S &\rightarrow AB \\A &\rightarrow aAA \mid \epsilon \\B &\rightarrow bBB \mid \epsilon\end{aligned}$$

- Find the nullable symbols.
- {A,B,S}

Example 7.8: Consider the grammar

$$\begin{aligned}S &\rightarrow AB \\A &\rightarrow aAA \mid \epsilon \\B &\rightarrow bBB \mid \epsilon\end{aligned}$$

- Find the nullable symbols.
- {A,B,S}
- Apply the method.

Example 7.8: Consider the grammar

$$\begin{aligned}S &\rightarrow AB \\A &\rightarrow aAA \mid \epsilon \\B &\rightarrow bBB \mid \epsilon\end{aligned}$$

- Find the nullable symbols.
- {A,B,S}
- Apply the method.

$$\begin{aligned}S &\rightarrow AB \mid A \mid B \\A &\rightarrow aAA \mid aA \mid a \\B &\rightarrow bBB \mid bB \mid b\end{aligned}$$

Example 7.8: Consider the grammar

$$\begin{aligned}S &\rightarrow AB \\A &\rightarrow aAA \mid \epsilon \\B &\rightarrow bBB \mid \epsilon\end{aligned}$$

- Find the nullable symbols.
- {A,B,S}
- Apply the method.

$$\begin{aligned}S &\rightarrow AB \mid A \mid B \\A &\rightarrow aAA \mid aA \mid a \\B &\rightarrow bBB \mid bB \mid b\end{aligned}$$

- Any thing missing??

Example 7.8: Consider the grammar

$$\begin{aligned}S &\rightarrow AB \\A &\rightarrow aAA \mid \epsilon \\B &\rightarrow bBB \mid \epsilon\end{aligned}$$

- Find the nullable symbols.
- {A,B,S}
- Apply the method.

$$\begin{aligned}S &\rightarrow AB \mid A \mid B \\A &\rightarrow aAA \mid aA \mid a \\B &\rightarrow bBB \mid bB \mid b\end{aligned}$$

- Any thing missing?? ϵ is not in the new language.

Theorem 7.9: If the grammar G_1 is constructed from G by the above construction for eliminating ϵ -productions, then $L(G_1) = L(G) - \{\epsilon\}$.

Eliminating Unit Productions

- A unit production is of the form $A \rightarrow B$, where A and B are variables.

Eliminating Unit Productions

- A unit production is of the form $A \rightarrow B$, where A and B are variables.
- They may be useful, but we can construct an equivalent grammar without them.
- This simplifies the CFG.
 - Unit productions introduce extra steps into derivations that technically need not be there.
 - This complicates proving certain facts.

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \rightarrow & I \mid (E) \\ T & \rightarrow & F \mid T * F \\ E & \rightarrow & T \mid E + T \end{array}$$

- In this $E \rightarrow T$ is a unit production.
- How to remove this?
- $E \rightarrow F \mid T * F \mid E + T$
- Still $E \rightarrow F$ is problematic
- Finally...

$$E \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F \mid E + T$$

$$A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_n \Rightarrow \alpha$$

can be replaced by

$$A \rightarrow \alpha.$$

How to do this systematically?

- Unit pairs are identified.
- Along with this, the CFG is used to produce a new CFG which is without any unit production.

Unit pair

- (A, B) is a unit pair if $A \xrightarrow{*} B$
- Note, if the CFG have $A \rightarrow BC$ and $C \rightarrow \epsilon$
- then, (A, B) is a unit pair

BASIS: (A, A) is a unit pair for any variable A . That is, $A \xrightarrow{*} A$ by zero steps.

INDUCTION: Suppose we have determined that (A, B) is a unit pair, and $B \rightarrow C$ is a production, where C is a variable. Then (A, C) is a unit pair.

$$\begin{array}{lcl}
 I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\
 F & \rightarrow & I \mid (E) \\
 T & \rightarrow & F \mid T * F \\
 E & \rightarrow & T \mid E + T
 \end{array}$$

The basis gives us the unit pairs (E, E) , (T, T) , (F, F) , and (I, I) . For the inductive step, we can make the following inferences:

1. (E, E) and the production $E \rightarrow T$ gives us unit pair (E, T) .
2. (E, T) and the production $T \rightarrow F$ gives us unit pair (E, F) .
3. (E, F) and the production $F \rightarrow I$ gives us unit pair (E, I) .
4. (T, T) and the production $T \rightarrow F$ gives us unit pair (T, F) .
5. (T, F) and the production $F \rightarrow I$ gives us unit pair (T, I) .
6. (F, F) and the production $F \rightarrow I$ gives us unit pair (F, I) .

There are no more pairs that can be inferred, and in fact these ten pairs represent all the derivations that use nothing but unit productions. \square

To eliminate unit productions, we proceed as follows. Given a CFG $G = (V, T, P, S)$, construct CFG $G_1 = (V, T, P_1, S)$:

1. Find all the unit pairs of G .
2. For each unit pair (A, B) , add to P_1 all the productions $A \rightarrow \alpha$, where $B \rightarrow \alpha$ is a nonunit production in P . Note that $A = B$ is possible; in that way, P_1 contains all the nonunit productions in P .

Given CFG:

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \rightarrow & I \mid (E) \\ T & \rightarrow & F \mid T * F \\ E & \rightarrow & T \mid E + T \end{array}$$

Non-unit productions are

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \rightarrow & (E) \\ T & \rightarrow & T * F \\ E & \rightarrow & E + T \end{array}$$

Given CFG:

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \rightarrow & I \mid (E) \\ T & \rightarrow & F \mid T * F \\ E & \rightarrow & T \mid E + T \end{array}$$

Unit Pairs

$$\begin{array}{l} \text{Pair} \\ \hline (E, E) \\ (E, T) \\ (E, F) \\ (E, I) \\ (T, T) \\ (T, F) \\ (T, I) \\ (F, F) \\ (F, I) \\ (I, I) \end{array}$$

Non-unit productions are

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \rightarrow & (E) \\ T & \rightarrow & T * F \\ E & \rightarrow & E + T \end{array}$$

Given CFG:

$$\begin{array}{l} I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F \rightarrow I \mid (E) \\ T \rightarrow F \mid T * F \\ E \rightarrow T \mid E + T \end{array}$$

Unit Pairs

Pair	Productions
(E, E)	$E \rightarrow E + T$
(E, T)	$E \rightarrow T * F$
(E, F)	$E \rightarrow (E)$
(E, I)	$E \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T, T)	$T \rightarrow T * F$
(T, F)	$T \rightarrow (E)$
(T, I)	$T \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F, F)	$F \rightarrow (E)$
(F, I)	$F \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I, I)	$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Non-unit productions are

$$\begin{array}{l} I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F \rightarrow (E) \\ T \rightarrow T * F \\ E \rightarrow E + T \end{array}$$

Given CFG:

$$\begin{array}{l} I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F \rightarrow I \mid (E) \\ T \rightarrow F \mid T * F \\ E \rightarrow T \mid E + T \end{array}$$

Unit Pairs

Pair	Productions
(E, E)	$E \rightarrow E + T$
(E, T)	$E \rightarrow T * F$
(E, F)	$E \rightarrow (E)$
(E, I)	$E \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T, T)	$T \rightarrow T * F$
(T, F)	$T \rightarrow (E)$
(T, I)	$T \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F, F)	$F \rightarrow (E)$
(F, I)	$F \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I, I)	$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Non-unit productions are

$$\begin{array}{l} I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F \rightarrow (E) \\ T \rightarrow T * F \\ E \rightarrow E + T \end{array}$$

CFG without unit productions

$$\begin{array}{l} E \rightarrow E + T \mid T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ T \rightarrow T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F \rightarrow (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \end{array}$$

The order in which these preprocessing steps can occur

1. Eliminate ϵ -productions.
2. Eliminate unit productions.
3. Eliminate useless symbols.

* **Exercise 7.1.1:** Find a grammar equivalent to

$$\begin{array}{lcl} S & \rightarrow & AB \mid CA \\ A & \rightarrow & a \\ B & \rightarrow & BC \mid AB \\ C & \rightarrow & aB \mid b \end{array}$$

with no useless symbols.

* **Exercise 7.1.1:** Find a grammar equivalent to

$$\begin{array}{lcl} S & \rightarrow & AB \mid CA \\ A & \rightarrow & a \\ B & \rightarrow & BC \mid AB \\ C & \rightarrow & aB \mid b \end{array}$$

with no useless symbols.

- Eliminate non-generating, first.

* **Exercise 7.1.1:** Find a grammar equivalent to

$$\begin{array}{lcl} S & \rightarrow & AB \mid CA \\ A & \rightarrow & a \\ B & \rightarrow & BC \mid AB \\ C & \rightarrow & aB \mid b \end{array}$$

with no useless symbols.

- Eliminate non-generating, first.
- Generating symbols = $\{a, b, A, C, S\}$
- Non-generating symbol is B .

* Exercise 7.1.1: Find a grammar equivalent to

$$\begin{array}{lcl} S & \rightarrow & AB \mid CA \\ A & \rightarrow & a \\ B & \rightarrow & BC \mid AB \\ C & \rightarrow & aB \mid b \end{array}$$

with no useless symbols.

- Eliminate non-generating, first.
- Generating symbols = $\{a, b, A, C, S\}$
- Non-generating symbol is B .
- We get $S \rightarrow CA$, $A \rightarrow a$, $C \rightarrow b$

* Exercise 7.1.1: Find a grammar equivalent to

$$\begin{array}{lcl} S & \rightarrow & AB \mid CA \\ A & \rightarrow & a \\ B & \rightarrow & BC \mid AB \\ C & \rightarrow & aB \mid b \end{array}$$

with no useless symbols.

- Eliminate non-generating, first.
- Generating symbols = $\{a, b, A, C, S\}$
- Non-generating symbol is B .
- We get $S \rightarrow CA$, $A \rightarrow a$, $C \rightarrow b$
- Remove unreachable.

* Exercise 7.1.1: Find a grammar equivalent to

$$\begin{array}{lcl} S & \rightarrow & AB \mid CA \\ A & \rightarrow & a \\ B & \rightarrow & BC \mid AB \\ C & \rightarrow & aB \mid b \end{array}$$

with no useless symbols.

- Eliminate non-generating, first.
- Generating symbols = $\{a, b, A, C, S\}$
- Non-generating symbol is B .
- We get $S \rightarrow CA$, $A \rightarrow a$, $C \rightarrow b$
- Remove unreachable.
- All are reachable. So, $S \rightarrow CA$, $A \rightarrow a$, $C \rightarrow b$ is the answer.

Chomsky Normal Form

- For a CFL without ϵ , where all productions are of the form $A \rightarrow BC, A \rightarrow a$.
- After preprocessing steps, it is quite easy to get in to CNF.

- $A \rightarrow BCDE$ can be replaced by
 $A \rightarrow BF, F \rightarrow CDE.$

- $A \rightarrow BCDE$ can be replaced by
 $A \rightarrow BF, F \rightarrow CDE.$
- Then $F \rightarrow CDE$ can be replaced by $F \rightarrow CG$
and $G \rightarrow DE$

- $A \rightarrow BCDE$ can be replaced by
 $A \rightarrow BF, F \rightarrow CDE.$
- Then $F \rightarrow CDE$ can be replaced by $F \rightarrow CG$ and $G \rightarrow DE$
- So, $A \rightarrow BCDE$ can be replaced by $A \rightarrow BF,$ $F \rightarrow CG$ and $G \rightarrow DE$

- Similarly, $A \rightarrow BabC$ can be replaced by $A \rightarrow BDEC, D \rightarrow a, E \rightarrow b.$
- Then $A \rightarrow BDEC$ can be replaced by ...

* **Exercise 7.1.2:** Begin with the grammar:

$$\begin{array}{lcl} S & \rightarrow & ASB \mid \epsilon \\ A & \rightarrow & aAS \mid a \\ B & \rightarrow & SbS \mid A \mid bb \end{array}$$

- a) Eliminate ϵ -productions.
- b) Eliminate any unit productions in the resulting grammar.
- c) Eliminate any useless symbols in the resulting grammar.
- d) Put the resulting grammar into Chomsky Normal Form.

- Theorem: Let G be a CFG in CNF form, then for any $w \in L(G)$, w can be derived in exactly $(2|w| - 1)$ steps.

- Theorem: Let G be a CFG in CNF form, then for any $w \in L(G)$, w can be derived in exactly $(2|w| - 1)$ steps.
- Proof: Parse tree when the CFG is in CNF form is a binary tree (except for a small technical issue with $A \rightarrow a$)

- Theorem: Let G be a CFG in CNF form, then for any $w \in L(G)$, w can be derived in exactly $(2|w| - 1)$ steps.
- Proof: Parse tree when the CFG is in CNF form is a binary tree (except for a small technical issue with $A \rightarrow a$)
- Let n_0 be nodes with 0 children (leaves)
- Let n_1 be nodes with 1 child
- Let n_2 be nodes with 2 children

- Theorem: Let G be a CFG in CNF form, then for any $w \in L(G)$, w can be derived in exactly $(2|w| - 1)$ steps.
- Proof: Parse tree when the CFG is in CNF form is a binary tree (except for a small technical issue with $A \rightarrow a$)
- Let n_0 be nodes with 0 children (leaves)
- Let n_1 be nodes with 1 child
- Let n_2 be nodes with 2 children
- We have $n_0 = n_2 + 1$ (can you prove this?)

- We have, $n_0 = |w|$
- $n_0 = n_1$ (why?)
- Number of steps
 - $= n_2 + n_1$
 - $= n_0 + n_0 - 1$
 - $= 2n_0 - 1$
 - $= 2|w| - 1$

Pumping Lemma for CFL

Intuition

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

Intuition

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

The size of Parse Trees

Theorem 7.17: Suppose we have a parse tree according to a Chomsky-Normal-Form grammar $G = (V, T, P, S)$, and suppose that the yield of the tree is a terminal string w . If the length of the longest path is n , then $|w| \leq 2^{n-1}$.

PROOF: The proof is a simple induction on n .

The size of Parse Trees

Theorem 7.17: Suppose we have a parse tree according to a Chomsky-Normal-Form grammar $G = (V, T, P, S)$, and suppose that the yield of the tree is a terminal string w . If the length of the longest path is n , then $|w| \leq 2^{n-1}$.

PROOF: The proof is a simple induction on n .

- If $|w| > 2^{n-1}$, then the longest path is $> n$

The size of Parse Trees

Theorem 7.17: Suppose we have a parse tree according to a Chomsky-Normal-Form grammar $G = (V, T, P, S)$, and suppose that the yield of the tree is a terminal string w . If the length of the longest path is n , then $|w| \leq 2^{n-1}$.

PROOF: The proof is a simple induction on n .

- If $|w| > 2^{n-1}$, then the longest path is $> n$
- Let $|V| = m$

The size of Parse Trees

Theorem 7.17: Suppose we have a parse tree according to a Chomsky-Normal-Form grammar $G = (V, T, P, S)$, and suppose that the yield of the tree is a terminal string w . If the length of the longest path is n , then $|w| \leq 2^{n-1}$.

PROOF: The proof is a simple induction on n .

- If $|w| > 2^{n-1}$, then the longest path is $> n$
- Let $|V| = m$
- For $|w| = 2^m$, longest path is $> m + 1$

The size of Parse Trees

Theorem 7.17: Suppose we have a parse tree according to a Chomsky-Normal-Form grammar $G = (V, T, P, S)$, and suppose that the yield of the tree is a terminal string w . If the length of the longest path is n , then $|w| \leq 2^{n-1}$.

PROOF: The proof is a simple induction on n .

- If $|w| > 2^{n-1}$, then the longest path is $> n$
- Let $|V| = m$
- For $|w| = 2^m$, longest path is $> m + 1$
- In that longest path a variable must have been repeated (since we have only m variables).

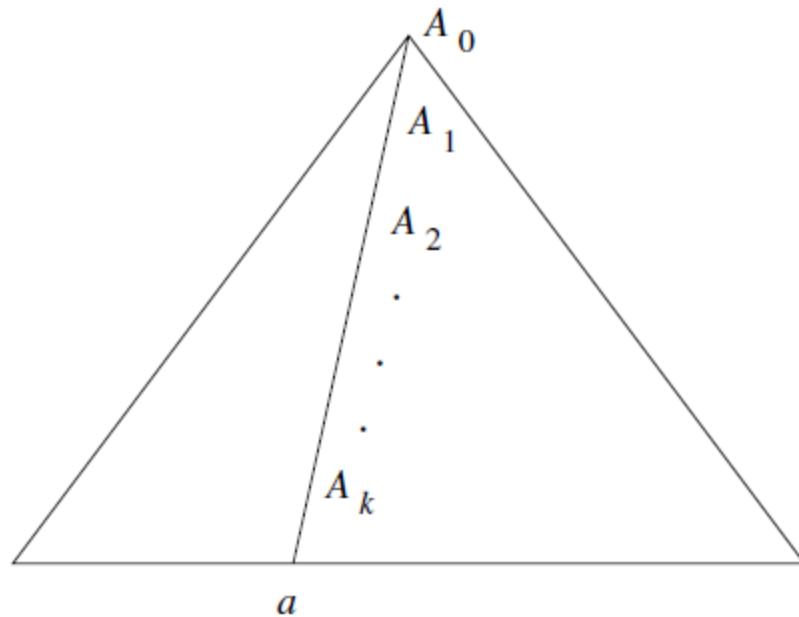


Figure 7.5: Every sufficiently long string in L must have a long path in its parse tree

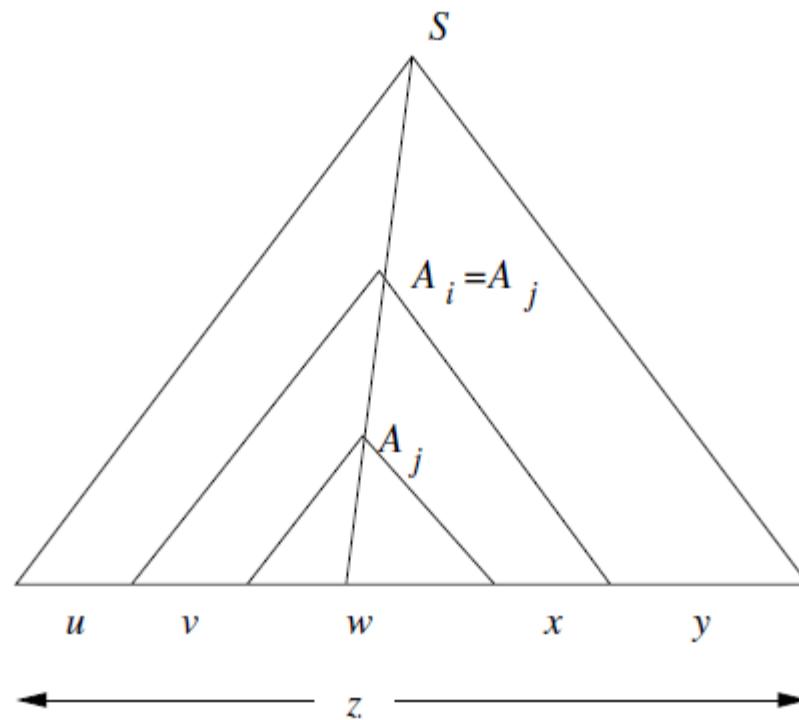
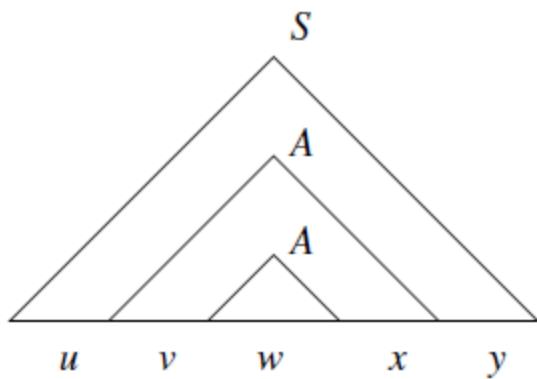
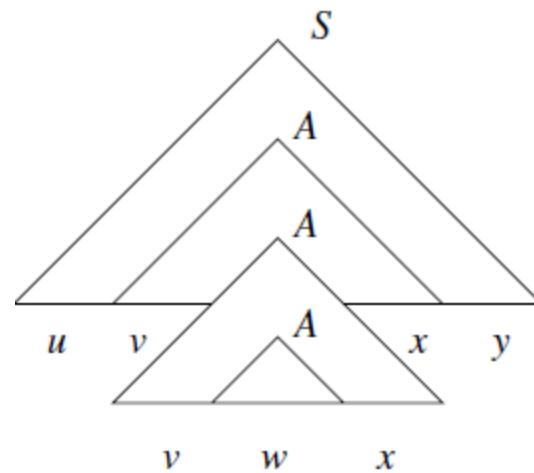
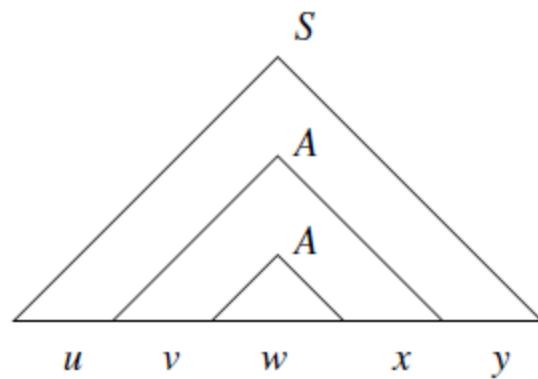
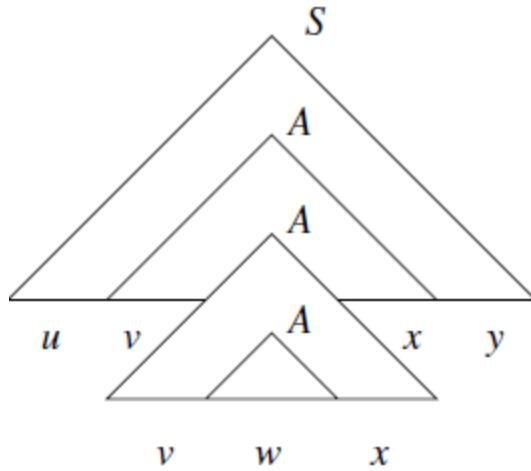
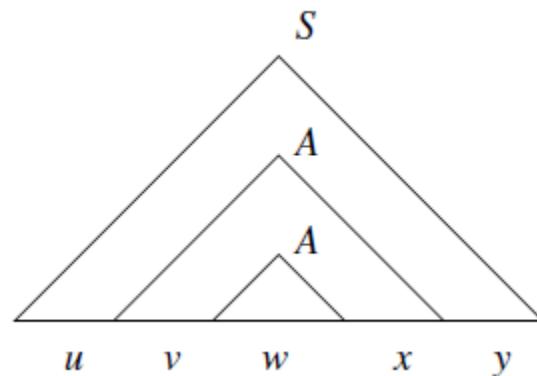
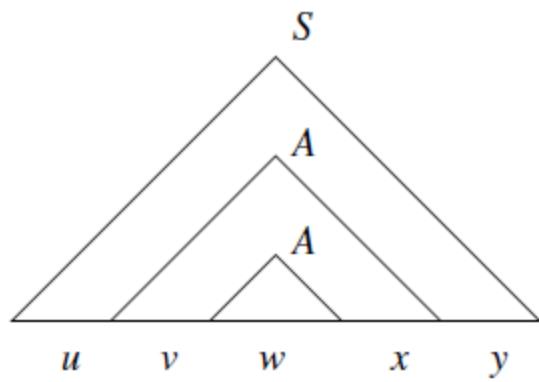
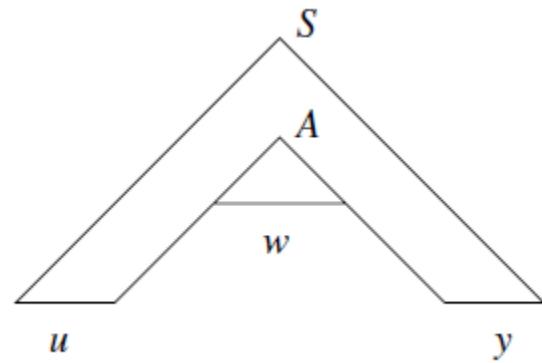
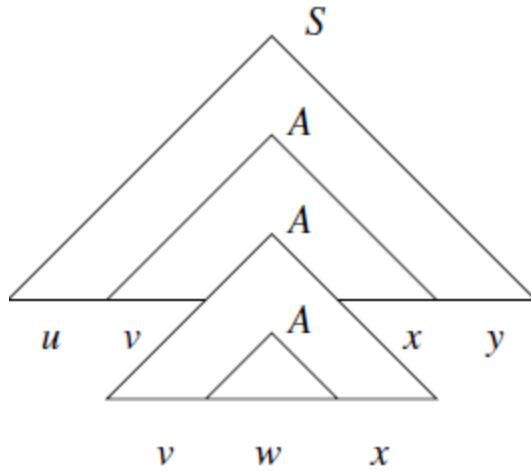
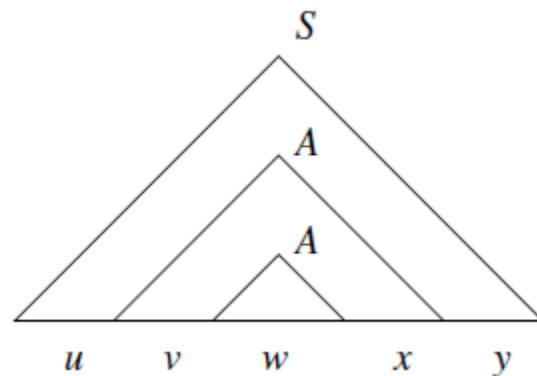
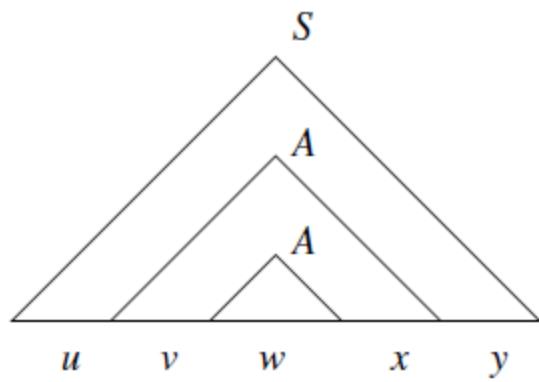


Figure 7.6: Dividing the string w so it can be pumped

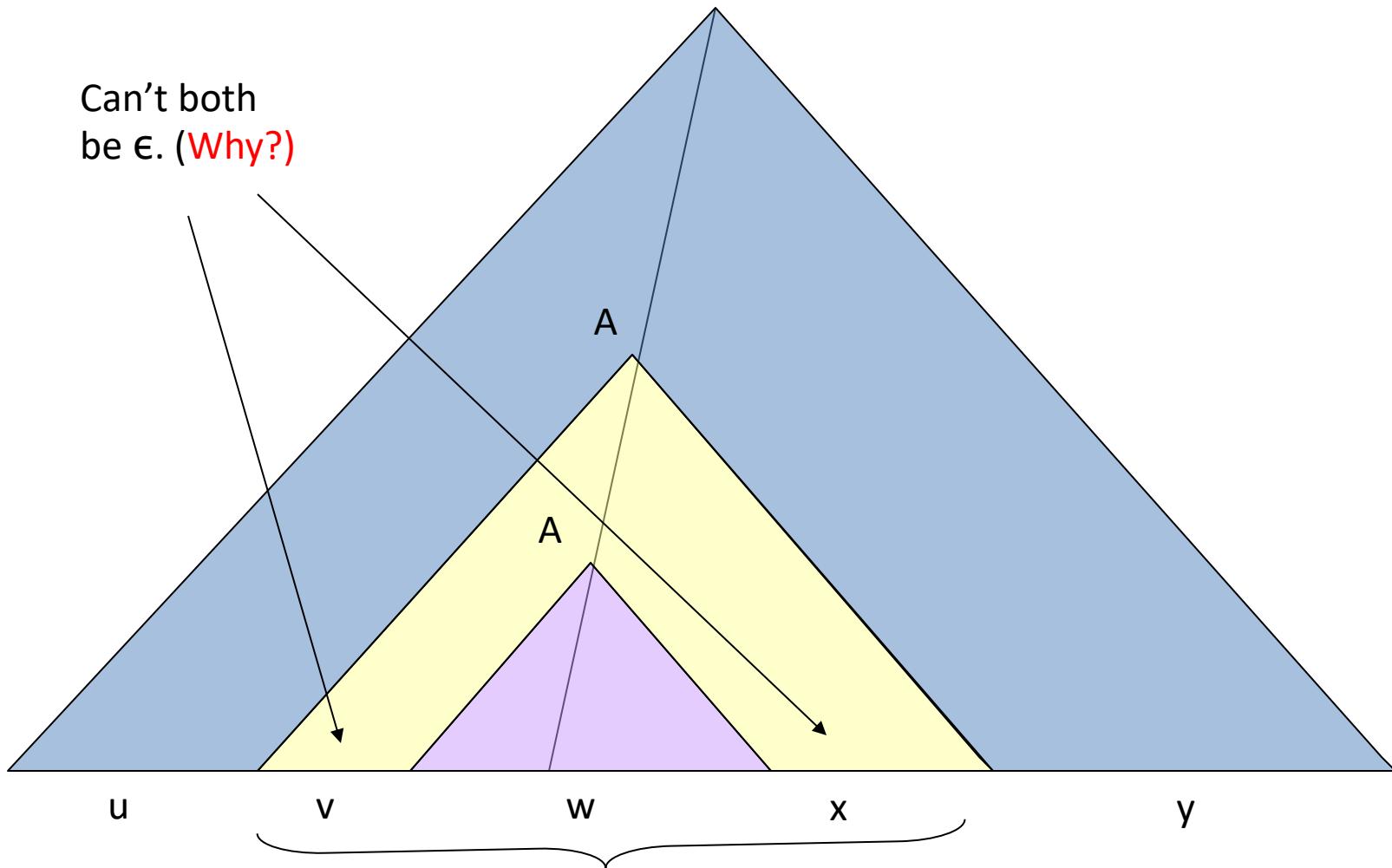




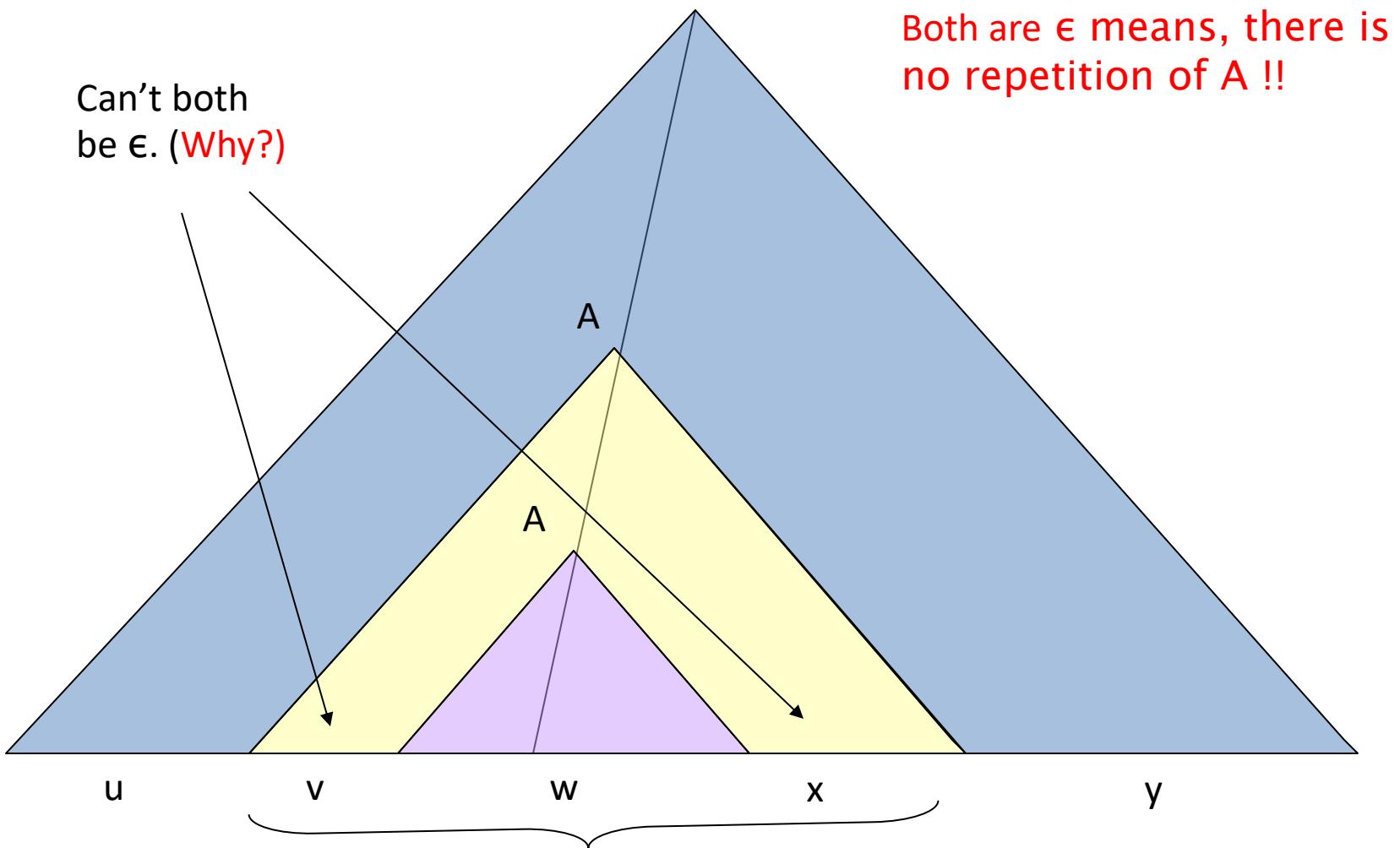




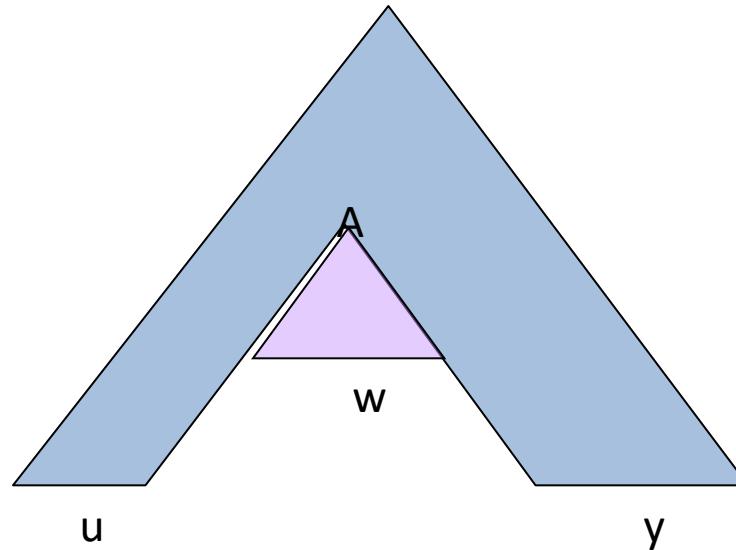
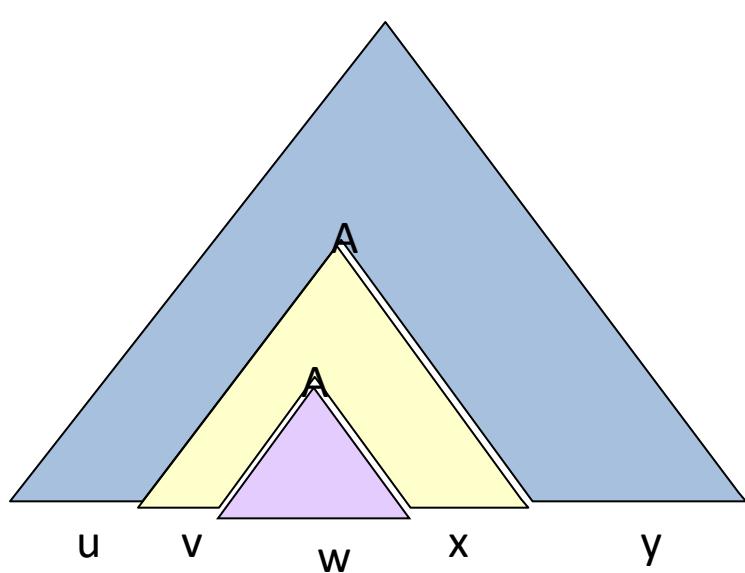
Parse Tree in the Pumping-Lemma



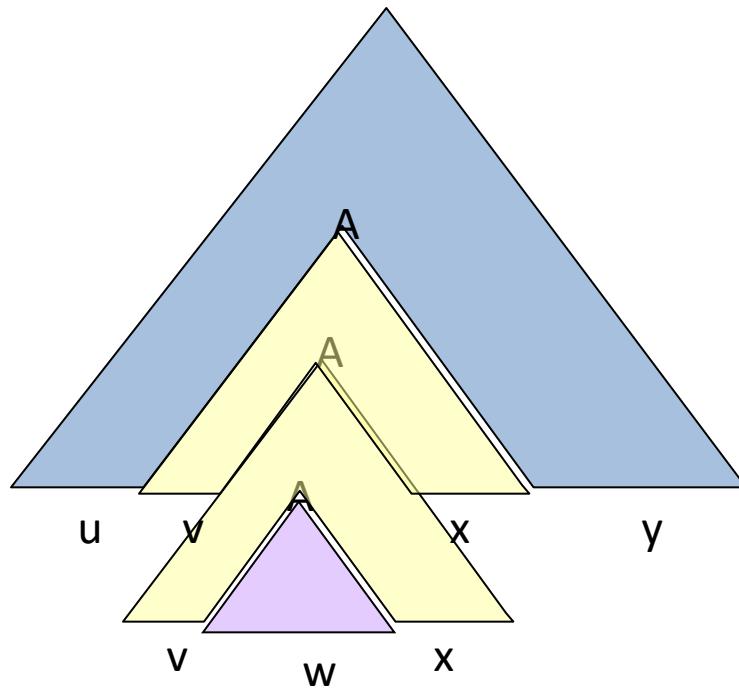
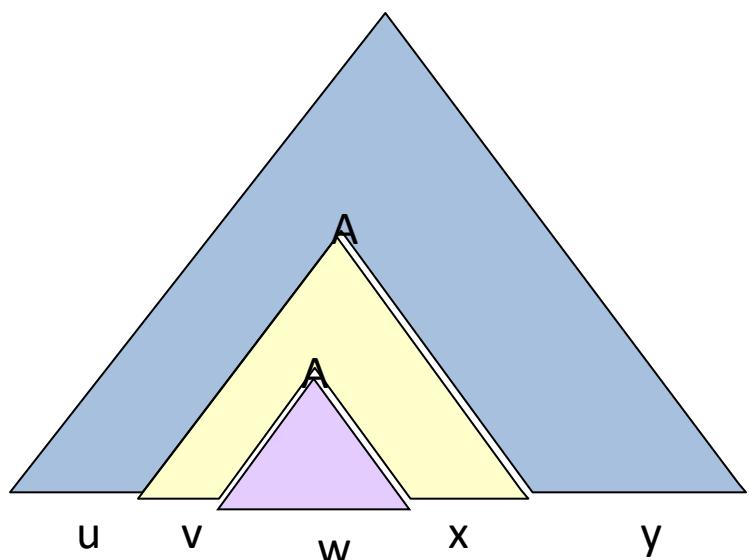
Parse Tree in the Pumping-Lemma



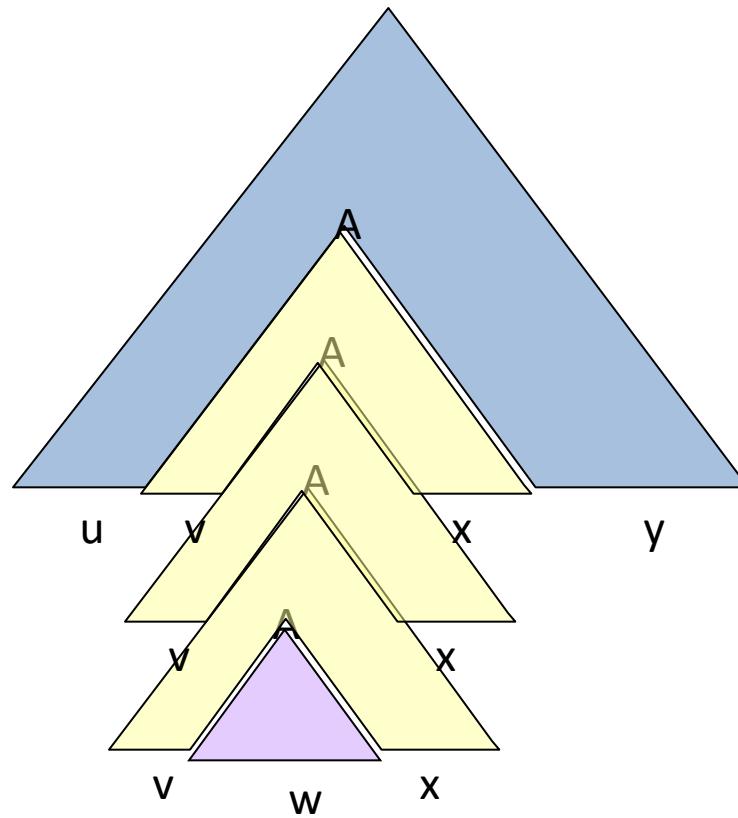
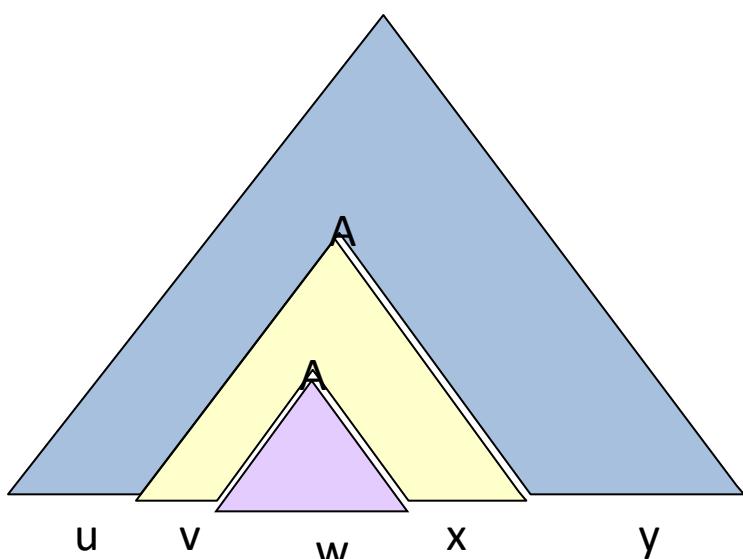
Pump Zero Times



Pump Twice



Pump Thrice Etc., Etc.



Statement

Theorem 7.18: (The pumping lemma for context-free languages) Let L be a CFL. Then there exists a constant n such that if z is any string in L such that $|z|$ is at least n , then we can write $z = uvwxy$, subject to the following conditions:

1. $|vwx| \leq n$. That is, the middle portion is not too long.
2. $vx \neq \epsilon$. Since v and x are the pieces to be “pumped,” this condition says that at least one of the strings we pump must not be empty.
3. For all $i \geq 0$, $uv^iwx^i y$ is in L . That is, the two strings v and x may be “pumped” any number of times, including 0, and the resulting string will still be a member of L .

Statement

For every context-free language L

There is an integer n , such that

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

There exists $z = uvwxy$ such that:

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

Example 7.19 : Let L be the language $\{0^n1^n2^n \mid n \geq 1\}$. That is, L consists of all strings in $0^+1^+2^+$ with an equal number of each symbol, e.g., 012, 001122, and so on. Suppose L were context-free. Then there is an integer n given to us by the pumping lemma. Let us pick $z = 0^n1^n2^n$.

Example 7.19 : Let L be the language $\{0^n 1^n 2^n \mid n \geq 1\}$. That is, L consists of all strings in $0^+ 1^+ 2^+$ with an equal number of each symbol, e.g., 012, 001122, and so on. Suppose L were context-free. Then there is an integer n given to us by the pumping lemma. Let us pick $z = 0^n 1^n 2^n$.

We can write $z = uvwxy$, where $|vwx| \leq n$ and v and x are not both ϵ .

Example 7.19 : Let L be the language $\{0^n 1^n 2^n \mid n \geq 1\}$. That is, L consists of all strings in $0^+ 1^+ 2^+$ with an equal number of each symbol, e.g., 012, 001122, and so on. Suppose L were context-free. Then there is an integer n given to us by the pumping lemma. Let us pick $z = 0^n 1^n 2^n$.

We can write $z = uvwxy$, where $|vwx| \leq n$ and v and x are not both ϵ .

Then we know that vwx cannot involve both 0's and 2's, since the last 0 and the first 2 are separated by $n + 1$ positions.

Example 7.19 : Let L be the language $\{0^n 1^n 2^n \mid n \geq 1\}$. That is, L consists of all strings in $0^+ 1^+ 2^+$ with an equal number of each symbol, e.g., 012, 001122, and so on. Suppose L were context-free. Then there is an integer n given to us by the pumping lemma. Let us pick $z = 0^n 1^n 2^n$.

We can write $z = uvwxy$, where $|vwx| \leq n$ and v and x are not both ϵ .

- There can be 5 cases where vwx is having
 - Only 0s
 - Some 0s and some 1s
 - Only 1s
 - Some 1s and some 2s
 - Only 2s.
- In all these 5 cases, $uv^2wx^2y \notin L$.

Example 7.21 : Let $L = \{ww \mid w \text{ is in } \{0, 1\}^*\}$.

- Show L is not a CFL.
- Note, $\{ww^R \mid w \in \{0, 1\}^*\}$ is a CFL.
- How can you prove this??

Example 7.21 : Let $L = \{ww \mid w \text{ is in } \{0, 1\}^*\}$.

- Let pumping length is n .
- Let the string be $z = 0^n 1^n 0^n 1^n$
- z can be written as $uvwxy$, such that $|vwx| \leq n$ and $vx \neq \epsilon$
- There are 7 cases, based on where vwx can occur in z .
- In all these cases it can be shown that uwy is not in L .

Using the Pumping Lemma

- $\{0^i 1 0^i \mid i \geq 1\}$ is a CFL.
 - We can match one pair of counts.
 - Can you give CFG??

Using the Pumping Lemma

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

Using the Pumping Lemma

- Consider $z = 0^n 1 0^n 1 0^n$.
- We can write $z = uvwxy$, where $|vwx| \leq n$, and $|vx| \geq 1$.
- **Case 1:** vx has no 0's.
 - Then at least one of them is a 1, and uw^kx has at most one 1, which no string in L does.

Using the Pumping Lemma

- Still considering $z = 0^n 1 0^n 1 0^n$.
- **Case 2:** vx has at least one 0.
 - vwx is too short ($\text{length } \leq n$) to extend to all three blocks of 0's in $0^n 1 0^n 1 0^n$.
 - Thus, uwy has at least one block of n 0's, and at least one block with fewer than n 0's.
 - Thus, uwy is not in L .

Closure properties of CFLs

- There are some differences when compared with regular languages.
- We deviate from both text books (to simplify the things).
- We want to skip homomorphism.

CFLs are --

- Closed under
 - Union
 - Concatenation
 - Kleene star
 - Reversal
 - Intersection with regular languages
 -
 -
- Not closed under
 - Intersection
 - Difference
 - Complement
 - Repetition
 -
 -

CFLs are closed under Union

- L_1 and L_2 are CFLs $\Rightarrow L_1 \cup L_2$ is a CFL.

CFLs are closed under Union

- L_1 and L_2 are CFLs $\Rightarrow L_1 \cup L_2$ is a CFL.
- Proof: Let the CFG for these are $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$.

CFLs are closed under Union

- L_1 and L_2 are CFLs $\Rightarrow L_1 \cup L_2$ is a CFL.
- Proof: Let the CFG for these are $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$.
- We rename variables so that V_1 and V_2 such that they are disjoint and does not have a variable with name S . Note T_1 and T_2 need not be disjoint. Perhaps they are same.
- Now the grammar $(V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 | S_2\}, S)$ will generate the union.

Closed under concatenation

- Add the production $S \rightarrow S_1S_2$

Closed under Kleene star

- Add productions $S \rightarrow S_1S|\epsilon$

Closed under reversal

Theorem 7.25 : If L is a CFL, then so is L^R .

PROOF: Let $L = L(G)$ for some CFL $G = (V, T, P, S)$. Construct $G^R = (V, T, P^R, S)$, where P^R is the “reverse” of each production in P . That is, if $A \rightarrow \alpha$ is a production of G , then $A \rightarrow \alpha^R$ is a production of G^R . It is an easy induction on the lengths of derivations in G and G^R to show that $L(G^R) = L^R$. Essentially, all the sentential forms of G^R are reverses of sentential forms of G , and vice-versa. We leave the formal proof as an exercise. \square

Not closed under intersection ☹

Not closed under intersection ☹

- Proof [by counter example]:

$L = \{0^n 1^n 2^n \mid n \geq 1\}$ is not a context-free language.

However, the following two languages *are* context-free:

$$\begin{aligned}L_1 &= \{0^n 1^n 2^i \mid n \geq 1, i \geq 1\} \\L_2 &= \{0^i 1^n 2^n \mid n \geq 1, i \geq 1\}\end{aligned}$$

- And, $L = L_1 \cap L_2$

A grammar for L_1 is:

$$\begin{aligned}S &\rightarrow AB \\A &\rightarrow 0A1 \mid 01 \\B &\rightarrow 2B \mid 2\end{aligned}$$

In this grammar, A generates all strings of the form 0^n1^n , and B generates all strings of 2's.

A grammar for L_2 is:

$$\begin{aligned}S &\rightarrow AB \\A &\rightarrow 0A \mid 0 \\B &\rightarrow 1B2 \mid 12\end{aligned}$$

CFLs are closed when intersected with regular languages

Theorem 7.27: If L is a CFL and R is a regular language, then $L \cap R$ is a CFL.

CFLs are closed when intersected with regular languages

Theorem 7.27: If L is a CFL and R is a regular language, then $L \cap R$ is a CFL.

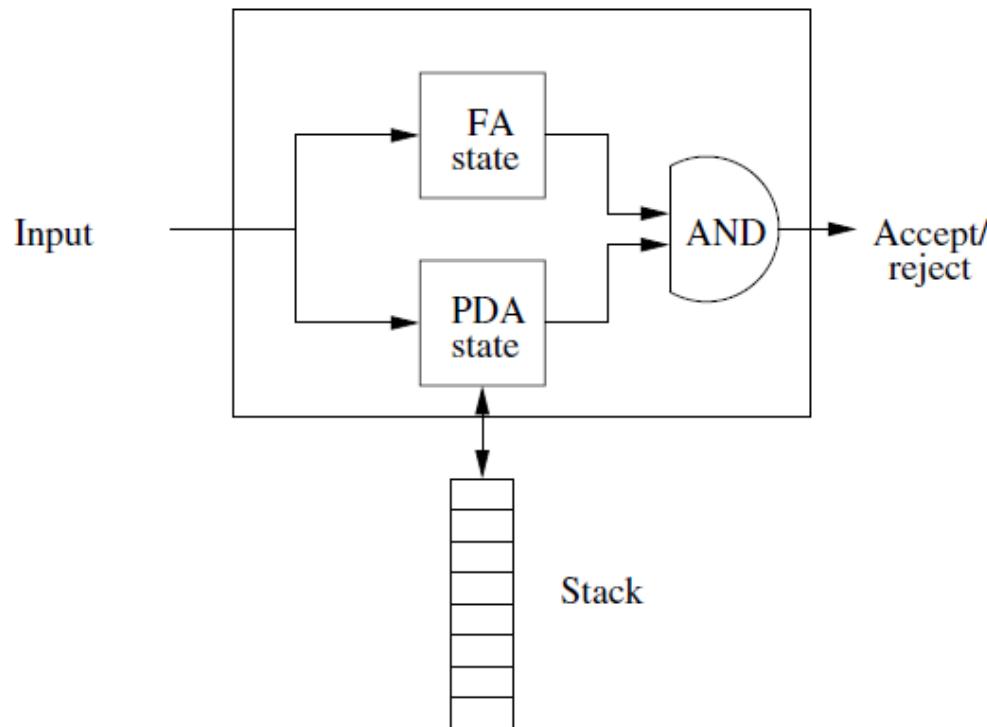


Figure 7.9: A PDA and a FA can run in parallel to create a new PDA

Formally, let $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$

be a PDA that accepts L by final state,

Formally, let $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$

be a PDA that accepts L by final state,

and let $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ be a DFA for R .

Formally, let $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$

be a PDA that accepts L by final state,

and let $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ be a DFA for R .

Construct PDA $P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$

Formally, let $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$

be a PDA that accepts L by final state,

and let $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ be a DFA for R .

Construct PDA $P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$

We need to define δ for the PDA.

Formally, let $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$

be a PDA that accepts L by final state,

and let $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ be a DFA for R .

Construct PDA $P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$

We need to define δ for the PDA.

Note that, PDA can change its state with ϵ from input, but DFA cannot. But we can use $\hat{\delta}$, the extended transition for DFA which says $\hat{\delta}(p, a) = p$, if $a = \epsilon$

Formally, let $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$

be a PDA that accepts L by final state,

and let $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ be a DFA for R .

Construct PDA $P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$

where $\delta((q, p), a, X)$ is defined to be the set of all pairs $((r, s), \gamma)$ such that:

1. $s = \hat{\delta}_A(p, a)$, and
2. Pair (r, γ) is in $\delta_P(q, a, X)$.

Formally, let $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$

be a PDA that accepts L by final state,

and let $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ be a DFA for R .

Construct PDA $P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$

where $\delta((q, p), a, X)$ is defined to be the set of all pairs $((r, s), \gamma)$ such that:

1. $s = \hat{\delta}_A(p, a)$, and
2. Pair (r, γ) is in $\delta_P(q, a, X)$.

We are allowing a to be ϵ

Formally, let $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$

be a PDA that accepts L by final state,

and let $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ be a DFA for R .

Construct PDA $P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$

where $\delta((q, p), a, X)$ is defined to be the set of all pairs $((r, s), \gamma)$ such that:

1. $s = \hat{\delta}_A(p, a)$, and
2. Pair (r, γ) is in $\delta_P(q, a, X)$.

We are allowing a to be ϵ

We could have taken NFA for R instead of a DFA, But the things will be more complicated (it can be done). Note, product of DFAs is simple; can be extended to NFAs also, but things are complicated !!

An application

- Dyck set (strings of balanced parentheses) is a CFL
- $(^*)^*$ is a regular language
- Intersection of above two is a CFL .
 - That is $\{ (^k)^k \mid k \geq 1\}$ is a CFL

To show “not CFL”

- The language L where each string has equal number of 0s, 1s and 2s (may be interleaved) is not CFL.

To show “not CFL”

- The language L where each string has equal number of 0s, 1s and 2s (may be interleaved) is not CFL.
- Take $R = 0^*1^*2^*$ and intersect with L .
- The result, if L is a CFL must be CFL.

To show “not CFL”

- The language L where each string has equal number of 0s, 1s and 2s (may be interleaved) is not CFL.
- Take $R = 0^*1^*2^*$ and intersect with L .
- The result, if L is a CFL must be CFL.
- But the result, which is $\{0^n 1^n 2^n \mid n \geq 1\}$ is not a CFL. **Contradiction.**

To show “not CFL”

- The language L where each string has equal number of 0s, 1s and 2s (may be interleaved) is not CFL.
- Take $R = 0^*1^*2^*$ and intersect with L .
- The result, if L is a CFL must be CFL.
- But the result, which is $\{0^n 1^n 2^n \mid n \geq 1\}$ is not a CFL. **Contradiction.**

Of course, pumping lemma for CFLs can directly applied on L and shown to fail.

CFLs are not closed under complementation

- We know $L = \{ww \mid w \in \{0,1\}^*\}$ is not a CFL.

CFLs are not closed under complementation

- We know $L = \{ww \mid w \in \{0,1\}^*\}$ is not a CFL.
- But its complement is a CFL !!
- This is a proof by counter example.

CFG for the \bar{L}

- $S \rightarrow S_o | S_e$
- $S_o \rightarrow 0R|1R|0|1, R \rightarrow 0S_o|1S_o$
- $S_e \rightarrow XY|YX, X \rightarrow ZXZ | 0,$
 $Y \rightarrow ZYZ | 1, Z \rightarrow 0|1$
- S_o generates odd length strings, whereas S_e generates even length strings.

Other proof

- Other proof is through contradiction.
- If it is closed under complementation then it has to be closed under intersection.

Other proof

- Other proof is through contradiction.
- If it is closed under complementation then it has to be closed under intersection.
- Depends on the identity

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

Not closed under set difference

- Proof by contradiction.
- If closed under set difference, then it has to be closed under complementation.

Not closed under set difference

- Proof by contradiction.
- If closed under set difference, then it has to be closed under complementation.
- Assuming it is closed under set difference.
- Let $L_1 = \Sigma^*$ which is a CFL
- Consider any other CFL L_2
- Then $L_1 - L_2$ which is **the complement of L_2** must be a CFL. **Contradiction.**

Set difference with regular is okay 😊

- If L is a CFL and R is a regular language, then $L - R$ is a CFL.
- $L - R = L \cap \bar{R}$
- If R is regular then \bar{R} is regular.

Exercise Problems

- Create PDA for well formed parentheses language (Dyck set).
- What is your idea ?

- Create PDA for well formed parentheses language (Dyck set).
- What is your idea ?

$$\delta(q_0, (, z_0) = (q_0, (z_0))$$

$$\delta(q_0, (, () = (q_0, (())$$

$$\delta(q_1,), () = (q_1, \varepsilon)$$

$$\delta(q_0, \varepsilon, z_0) = (q_f, \varepsilon)$$

$$\delta(q_0, (, z_0) = (q_0, (z_0 ,$$

$$\delta(q_0, (, () = (q_0, (()$$

$$\delta(q_1,), () = (q_1, \varepsilon)$$

- Draw transition diagram.

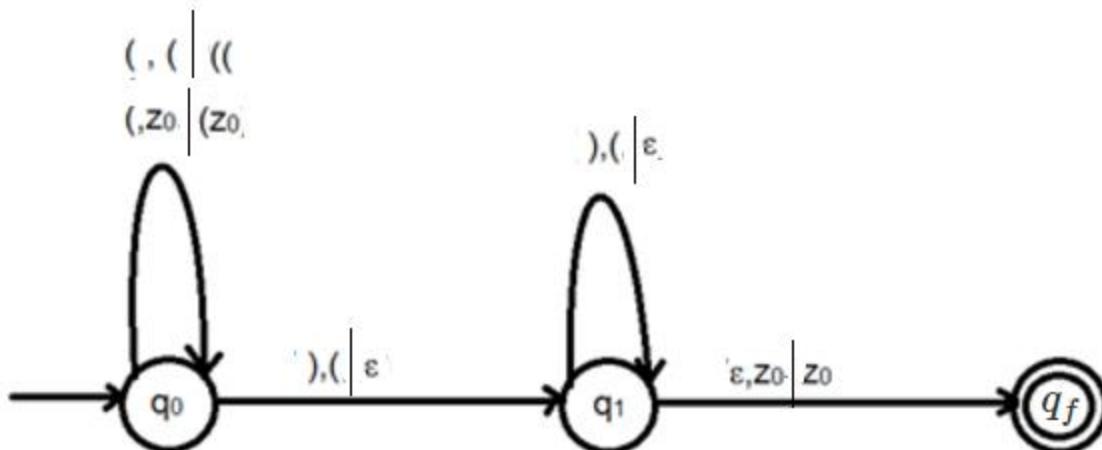
$$\delta(q_0, \varepsilon, z_0) = (q_f, \varepsilon)$$

$$\delta(q_0, (, z_0) = (q_0, (z_0))$$

$$\delta(q_0, (, () = (q_0, (())$$

$$\delta(q_1, (), () = (q_1, \varepsilon)$$

- Draw transition diagram.



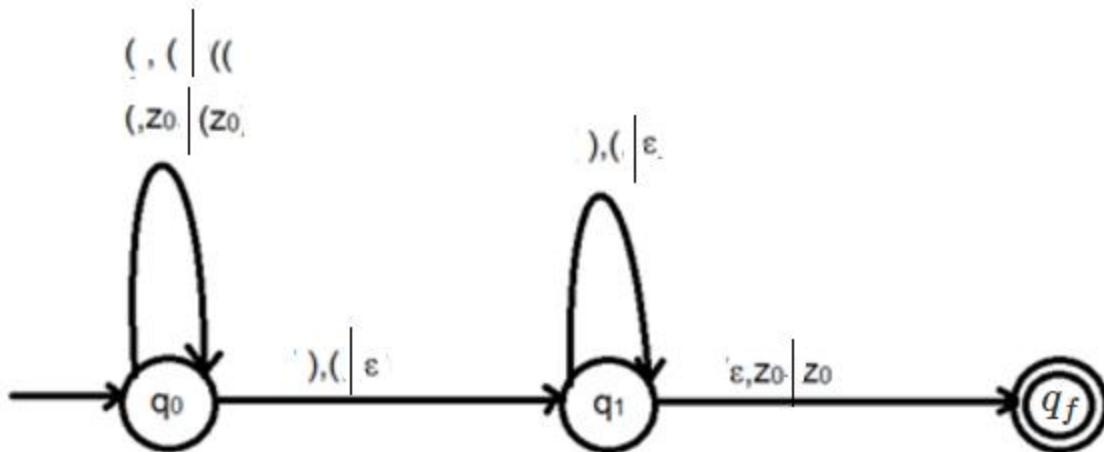
$$\delta(q_0, (, z_0) = (q_0, (z_0))$$

$$\delta(q_0, (, () = (q_0, (())$$

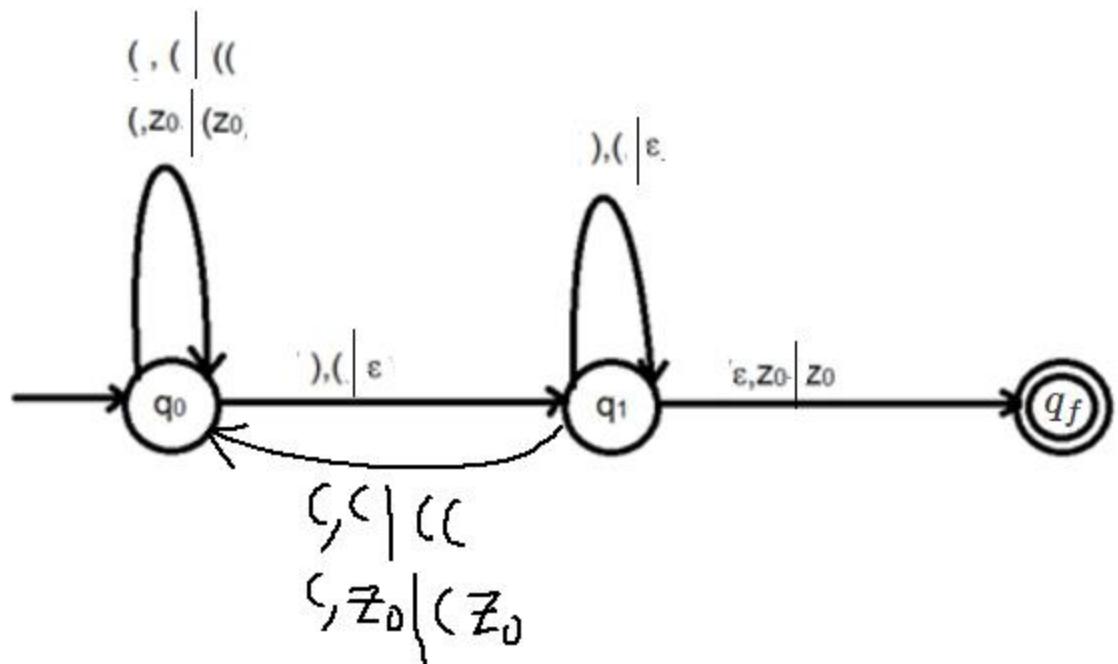
$$\delta(q_1, (), () = (q_1, \varepsilon)$$

- Draw transition diagram.

$$\delta(q_0, \varepsilon, z_0) = (q_f, \varepsilon)$$



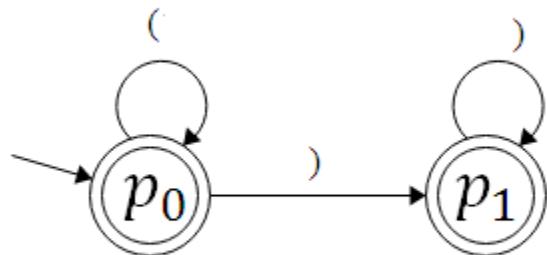
But this is not for the dyck set. It cannot recognize ()()



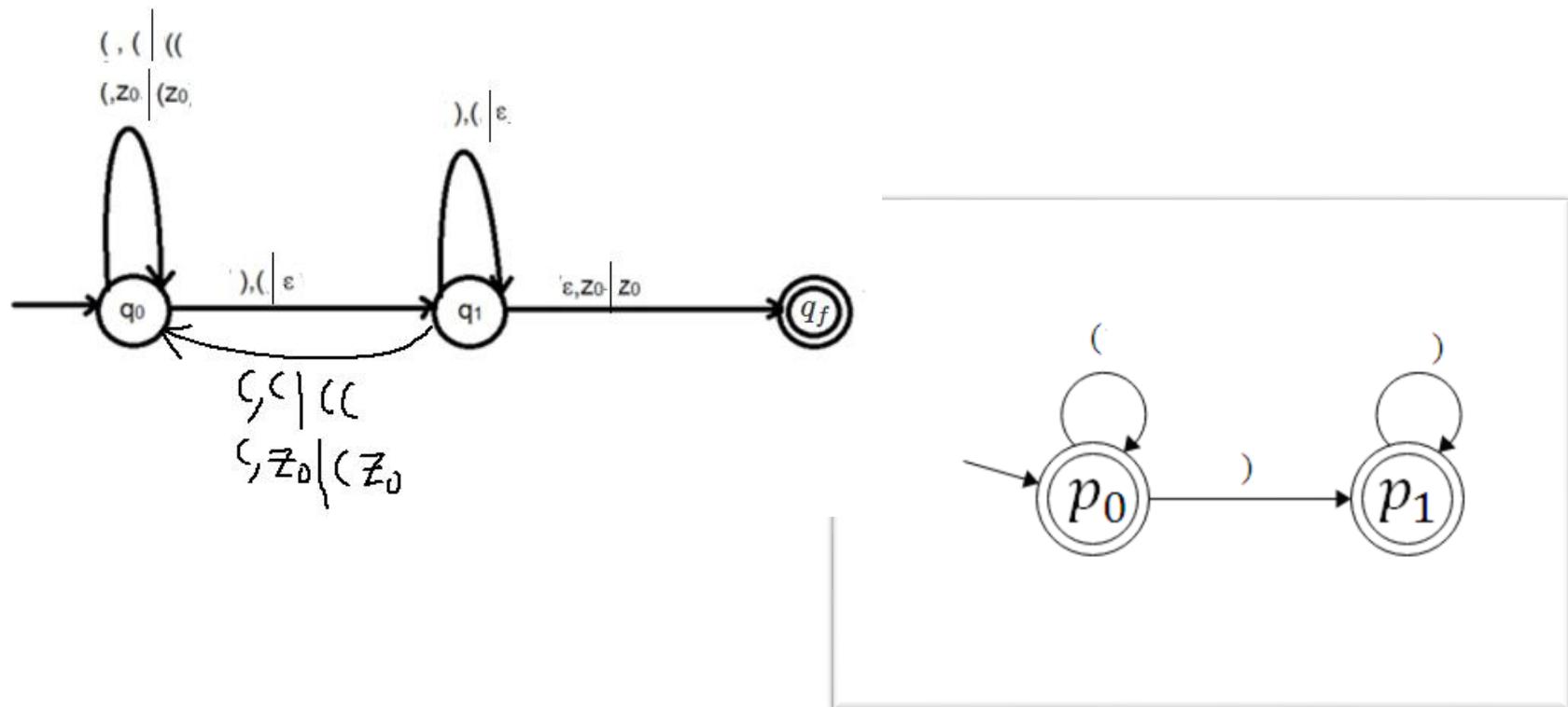
Now DFA

- Find DFA for $(^*)^*$
- Give transition diagram for this.

- Find DFA for $(^*)^*$
- Give transition diagram for this.



- Now create PDA for the intersection of the two languages.



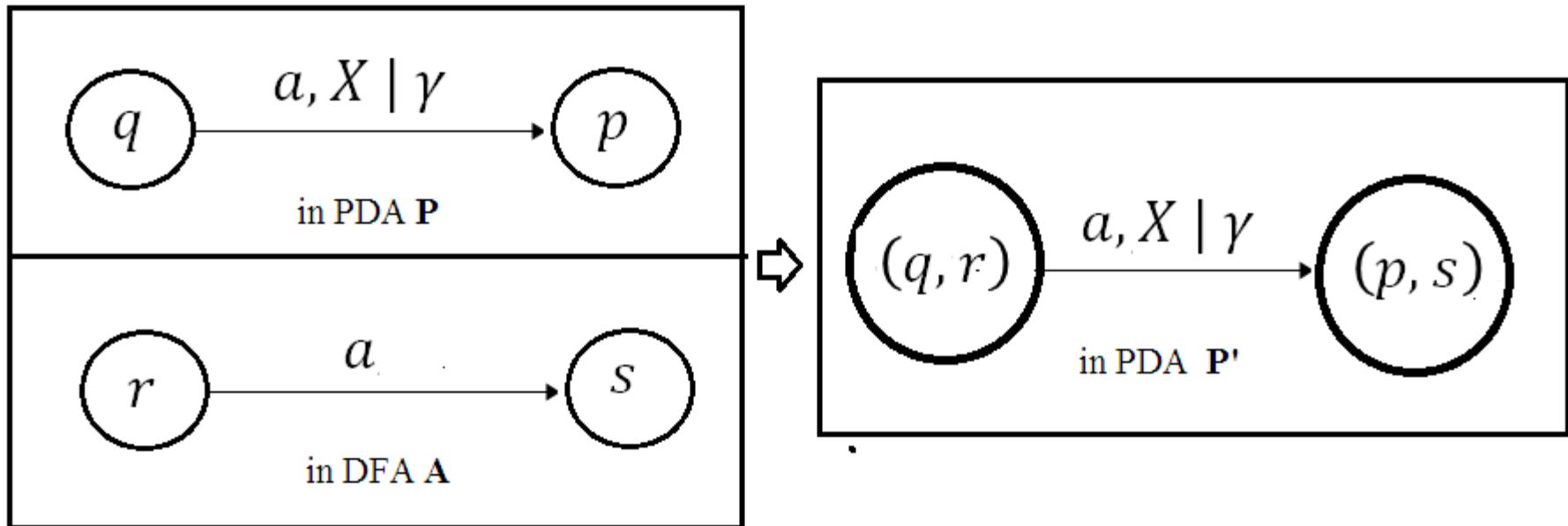
Verify your solution

- Your constructed PDA should recognize the language $\{ ()^k \mid k \geq 1 \}$.

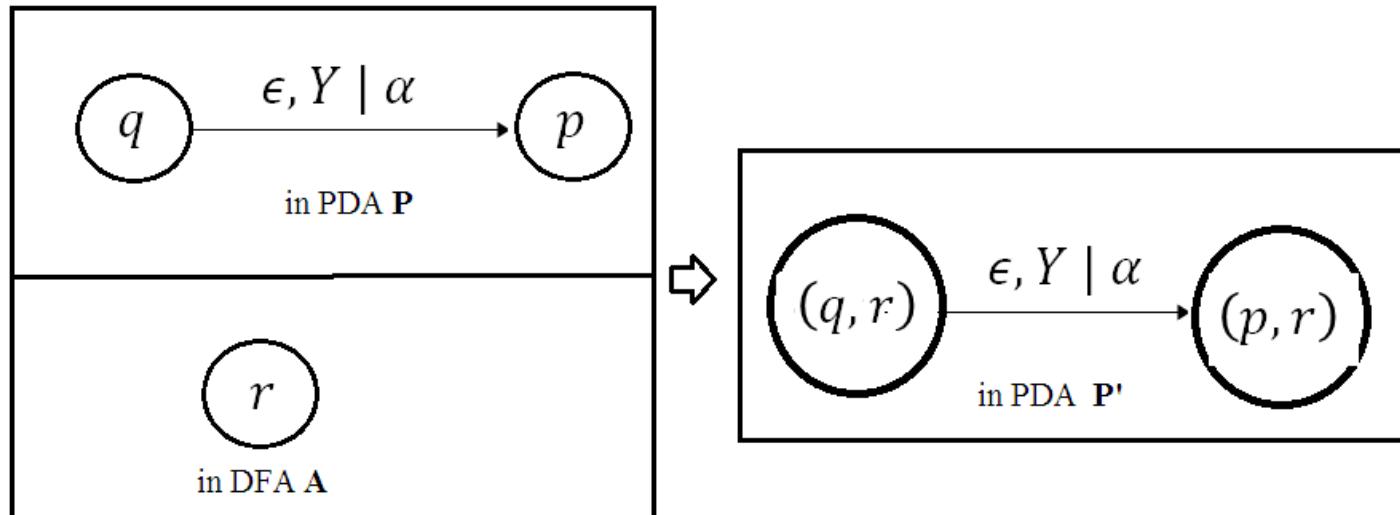
How intersection of PDA and DFA
is found?

- Let P be the PDA
- And A be the DFA
- We want to find PDA for the intersection of the languages recognized by P and A .

- When $a \in \Sigma$, i.e., an alphabet other than ϵ ,
- Then, for every possible combination, do



- When $a = \epsilon$,
- Then, for every state r of the DFA A , one has to add ... {as many as no. of states in A , that many has to be added for one arrow in P .}



Decidable properties of CFL

Several undecidable are there ...

- Membership question.
- Empty?
- Infinite/not ?

Membership question

- Given the CFG G and string w , we ask is $w \in L(G)$?
- There is a $O(n^3)$ algorithm where $|w| = n$, which is called the CYK algorithm.
 - This is a parsing technique whereby one can create the parse tree if the string is in the language.
 - Since this works for any CFG (not restricted to a subclass), this is one of universal parsers.

- If $w = \epsilon$, we verify to find whether S is nullable or not.
- Else, we convert the CFG in to CNF first.
- With CNF form the parse tree is a binary tree.
- And the string w can be derived in exactly $2|w| - 1$ steps.
- The parse tree will have exactly this many variables.

- We can list all possible derivations having $2|w| - 1$ steps.
- We verify whether, any, gave the string.
- But, this is an exponential time algorithm.

- There is a much more efficient technique based on the idea of “dynamic programming”.
- This is called the CYK algorithm.
- Also called the table-filling or tabulation algorithm.

³It is named after three people, each of whom independently discovered essentially the same idea: J. Cocke, D. Younger, and T. Kasami.

CYK algorithm

- Let $w = a_1 a_2 \cdots a_n$ be the given string.
- We fill a table, as shown, for example when
 $w = a_1 a_2 \cdots a_5$

The table entry X_{ij} is the set of variables A such that $A \xrightarrow{*} a_i a_{i+1} \cdots a_j$.

X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}

$a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5$

- If $S \in X_{1n}$ then $S \xrightarrow{*} w$
- To find X_{1n} we need to fill the table, in a bottom-up fashion.

The table entry X_{ij} is the set of variables A such that $A \xrightarrow{*} a_i a_{i+1} \cdots a_j$.

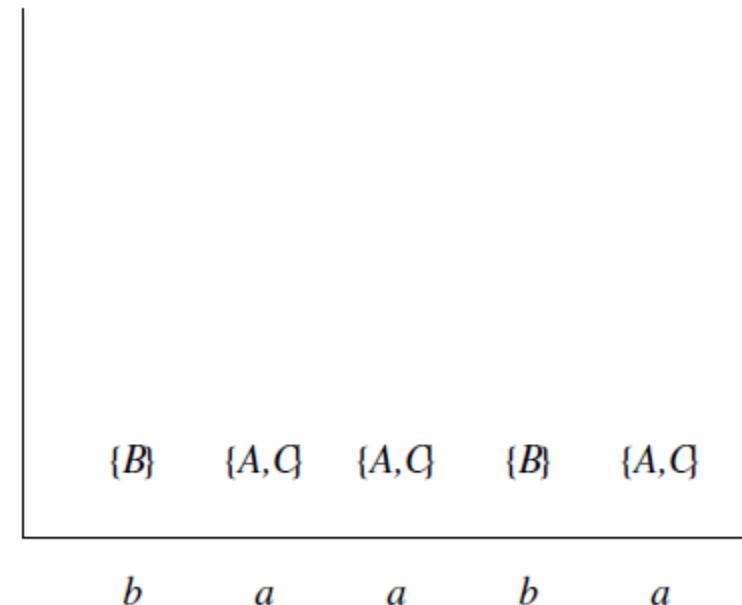
X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
a_1	a_2	a_3	a_4	a_5

$$\begin{array}{l} S \rightarrow AB \mid BC \\ A \rightarrow BA \mid a \\ B \rightarrow CC \mid b \\ C \rightarrow AB \mid a \end{array}$$

We shall test for membership in $L(G)$ the string \boxed{baaba}

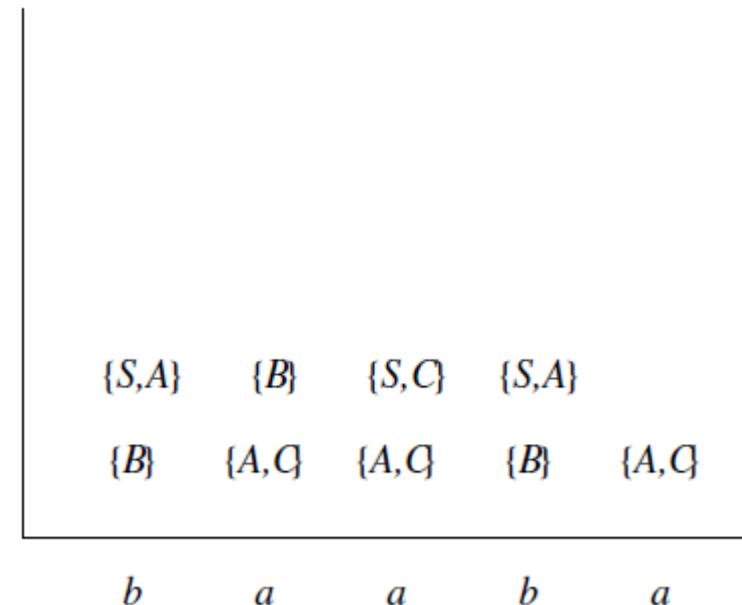
$$\begin{array}{lcl} S & \rightarrow & AB \mid BC \\ A & \rightarrow & BA \mid a \\ B & \rightarrow & CC \mid b \\ C & \rightarrow & AB \mid a \end{array}$$

We shall test for membership in $L(G)$ the string \boxed{baaba}



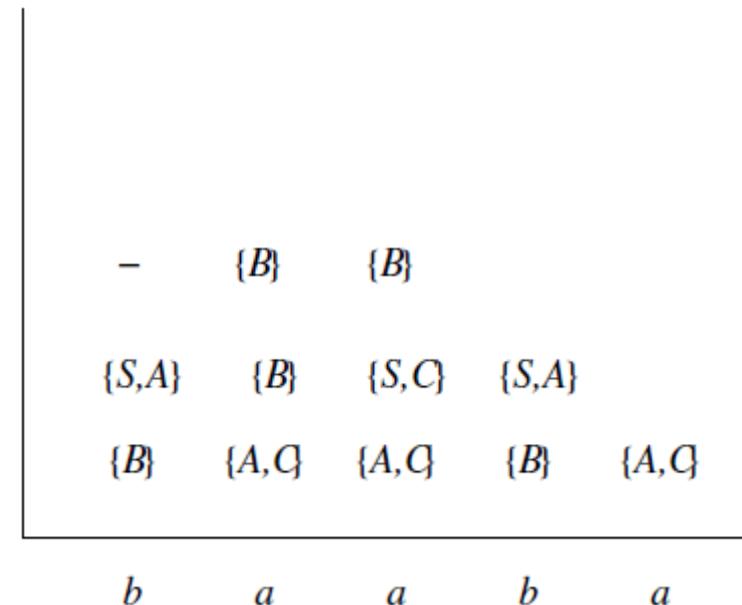
$$\begin{array}{l}
 S \rightarrow AB \mid BC \\
 A \rightarrow BA \mid a \\
 B \rightarrow CC \mid b \\
 C \rightarrow AB \mid a
 \end{array}$$

We shall test for membership in $L(G)$ the string baaba



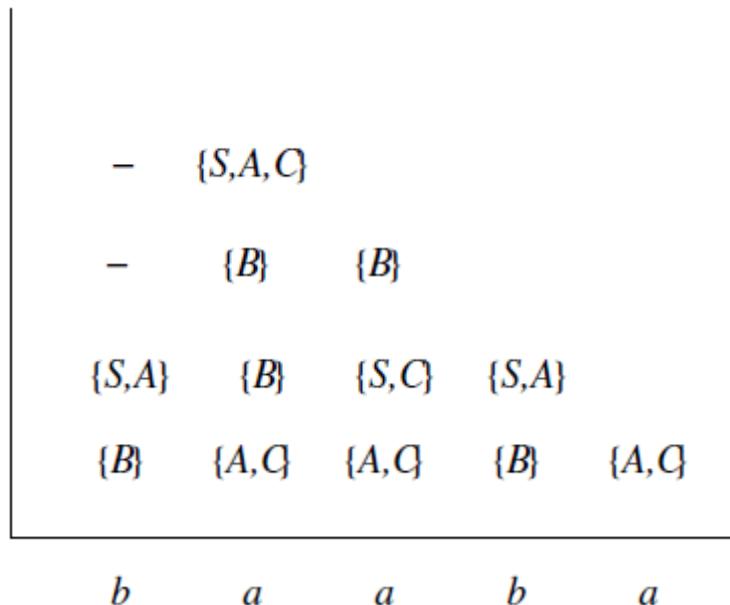
S	\rightarrow	AB	$ $	BC
A	\rightarrow	BA	$ $	a
B	\rightarrow	CC	$ $	b
C	\rightarrow	AB	$ $	a

We shall test for membership in $L(G)$ the string ***baaba***.



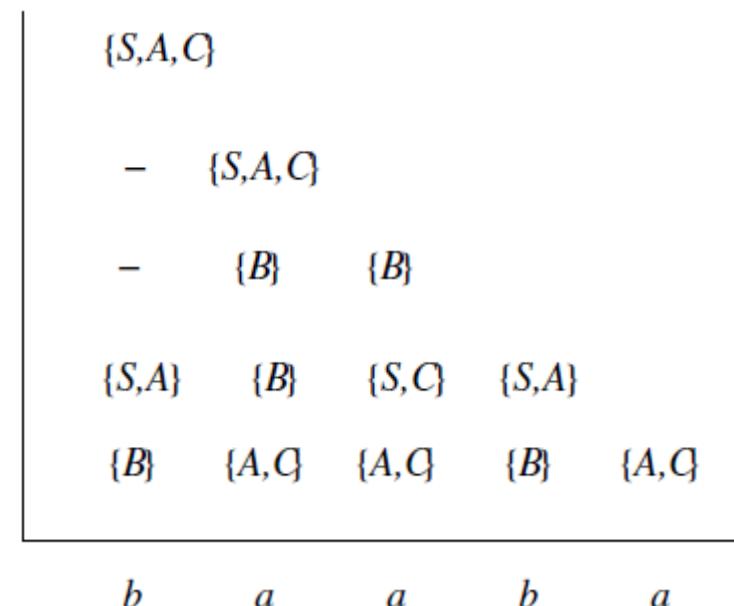
$$\begin{array}{l}
 S \rightarrow AB \mid BC \\
 A \rightarrow BA \mid a \\
 B \rightarrow CC \mid b \\
 C \rightarrow AB \mid a
 \end{array}$$

We shall test for membership in $L(G)$ the string baaba



$$\begin{array}{l}
 S \rightarrow AB \mid BC \\
 A \rightarrow BA \mid a \\
 B \rightarrow CC \mid b \\
 C \rightarrow AB \mid a
 \end{array}$$

We shall test for membership in $L(G)$ the string **$baaba$**



Parse tree

- Parse tree can be found by keeping track of some side information.

We shall test for membership in $L(G)$ the string \boxed{baaba}

S	\rightarrow	$AB \mid BC$
A	\rightarrow	$BA \mid a$
B	\rightarrow	$CC \mid b$
C	\rightarrow	$AB \mid a$

We shall test for membership in $L(G)$ the string \boxed{baaba}

S	\rightarrow	$AB \mid BC$
A	\rightarrow	$BA \mid a$
B	\rightarrow	$CC \mid b$
C	\rightarrow	$AB \mid a$

$\{S,A,C\}$

- $\{S,A,C\}$

- $\{B\}$

$\{S,A\}$

$\{B\}$

$\{B\}$

$\{S,C\}$

$\{A,C\}$

$\{B\}$

$\{A,C\}$

$\{B\}$

$\{A,C\}$

$\{B\}$

$\{A,C\}$

b

a

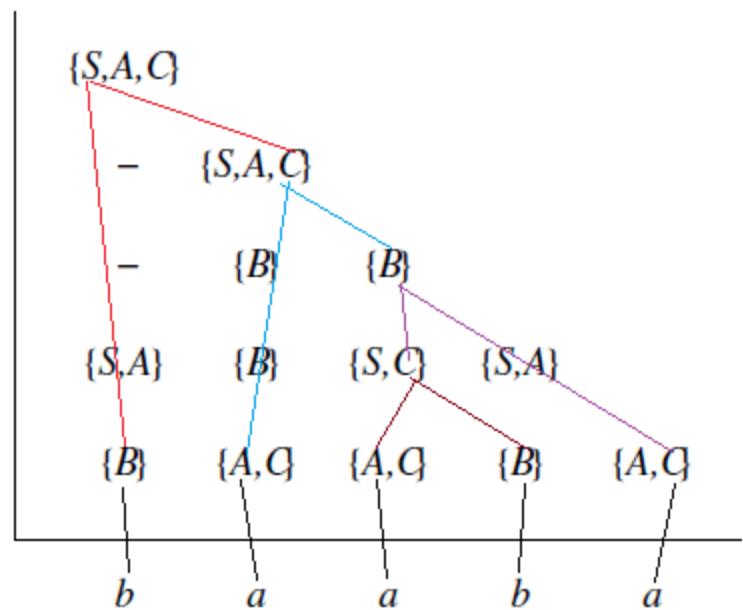
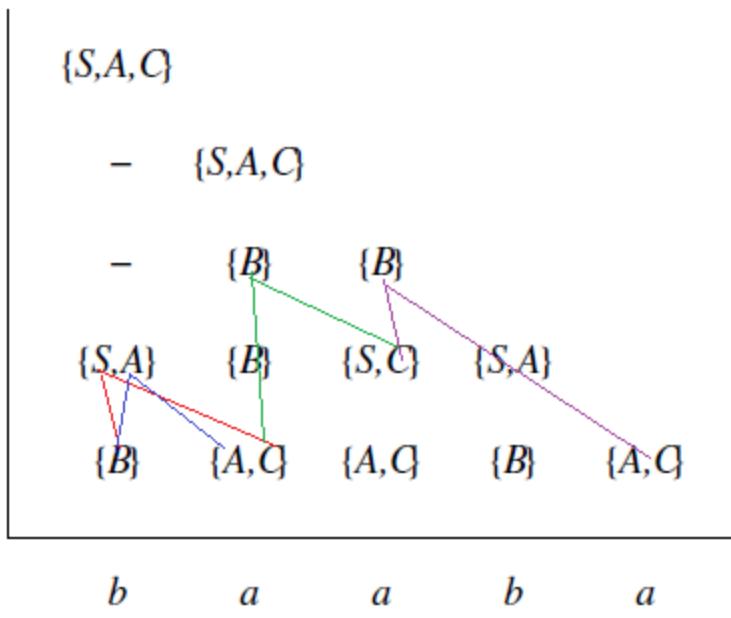
a

b

a

We shall test for membership in $L(G)$ the string **$baaba$**

S	\rightarrow	$AB \mid BC$
A	\rightarrow	$BA \mid a$
B	\rightarrow	$CC \mid b$
C	\rightarrow	$AB \mid a$



$$S \rightarrow \varepsilon \mid AB \mid XB$$
$$T \rightarrow AB \mid XB$$
$$X \rightarrow AT$$
$$A \rightarrow a$$
$$B \rightarrow b$$

1. Is $w = aaabb$ in $L(G)$?
2. Is $w = aaabbb$ in $L(G)$?

The given string is aaabb.

The grammar is CNF is :

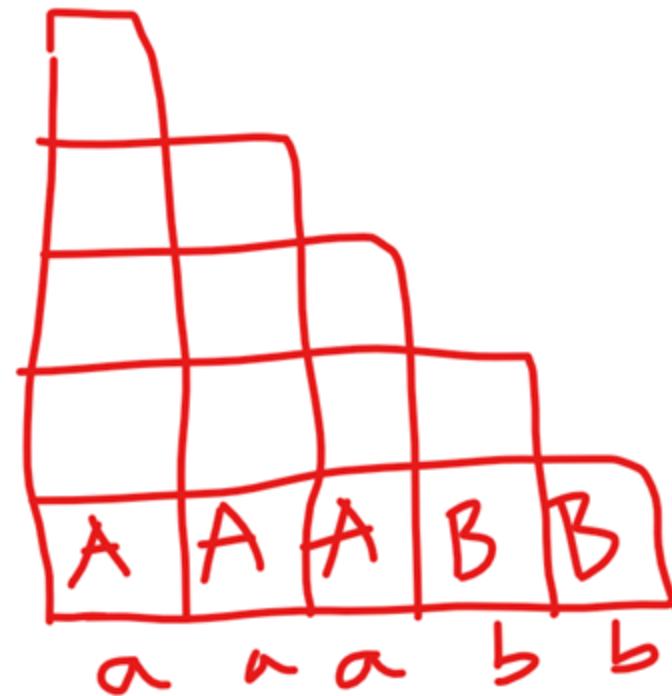
$$S \rightarrow AB \mid XB$$

$$T \rightarrow AB \mid XB$$

$$X \rightarrow AT$$

$$A \rightarrow a$$

$$B \rightarrow b$$



- Complete this..

Time complexity of CYK

- $O(n^3)$

Empty ?

- This is easy.
- Is S generating ?

Infinite or not?

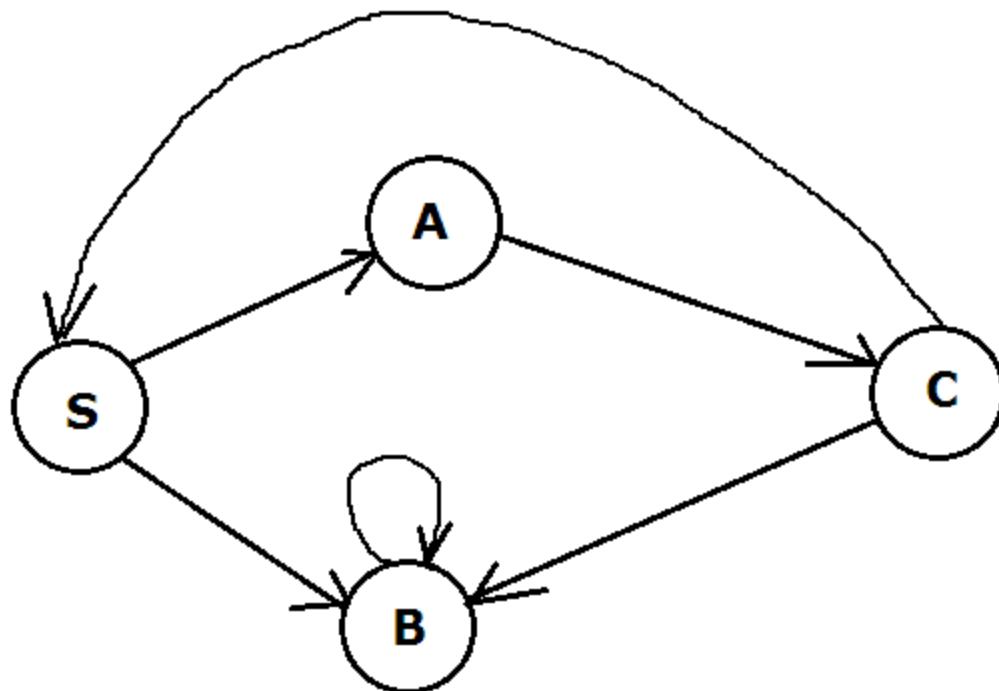
- **Algorithm.**
 1. Remove useless symbols.
 2. Remove unit and ϵ productions
 3. Create dependency graph for variables
 4. If there is a loop in the dependency graph then the language is infinite, else not.

Example

- $S \rightarrow AB, A \rightarrow aCb|a, B \rightarrow bB|b, C \rightarrow cBS$

Example

- $S \rightarrow AB, A \rightarrow aCb|a, B \rightarrow bB|b, C \rightarrow cBS$



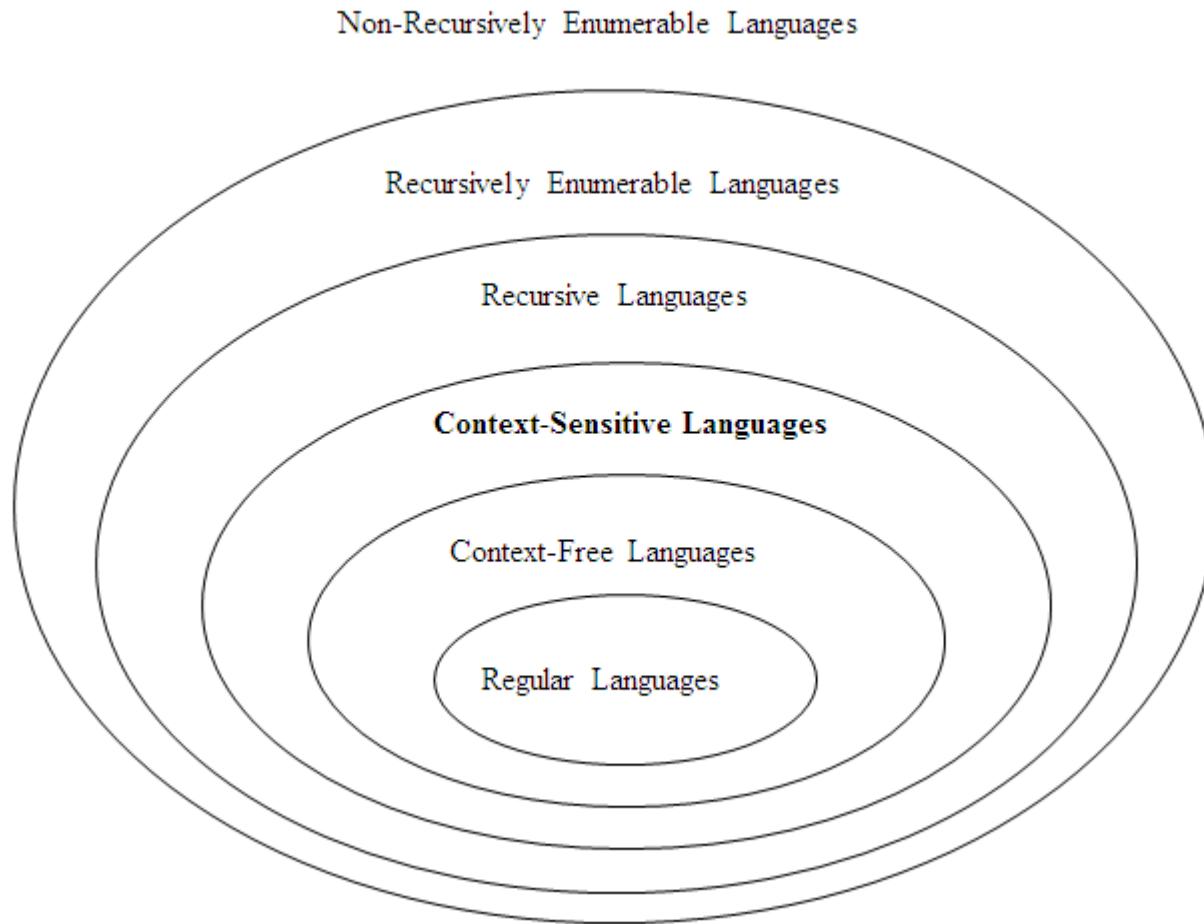
Some undecidable properties 😞

- Let G_1 and G_2 be two CFGs.
- Is $L(G_1) = \Sigma^*$?
- Is $L(G_1)$ regular ?
- Is $L(G_1) \subseteq L(G_2)$?
- Is $L(G_1) = L(G_2)$?
- Is $L(G_1) \cap L(G_2) = \phi$?
- Is $L(G_1)$ ambiguous? (inherent ambiguity)
- Is G_1 ambiguous?

Turing Machines

Recursively Enumerable and
Recursive Languages

- **The Hierarchy of Languages:**

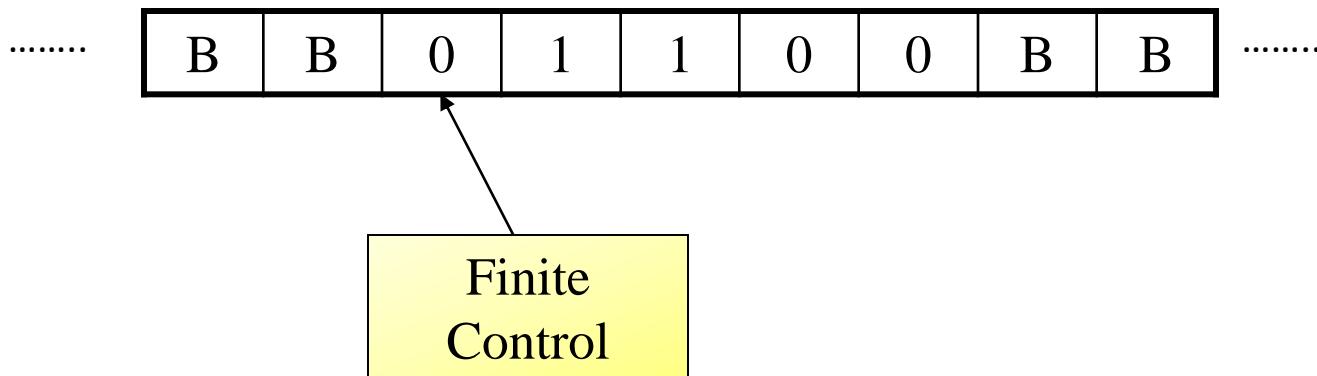


- Recursively enumerable languages are also known as *type 0* languages.
- Context-sensitive languages are also known as *type 1* languages.
- Context-free languages are also known as *type 2* languages.
- Regular languages are also known as *type 3* languages.

- TMs model the computing capability of a general purpose computer, which informally can be described as:
 - Effective procedure
 - Finitely describable
 - Well defined, discrete, “mechanical” steps
 - Always terminates
 - Computable function
 - A function computable by an effective procedure

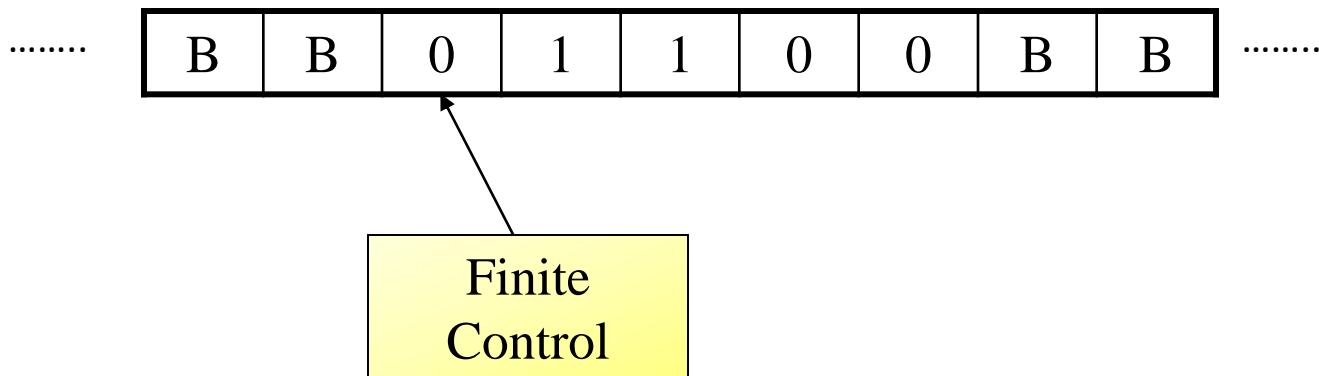
- TMs model the computing capability of a general purpose computer, which informally can be described as:
 - Effective procedure
 - Finitely describable
 - Well defined, discrete, “mechanical” steps
 - Always terminates
 - Computable function
 - A function computable by an effective procedure
- TMs formalize the above notion.
- **Church-Turing Thesis:** There is an effective procedure for solving a problem if and only if there is a TM that halts for all inputs and solves the problem.
 - There are many other computing models, but all are equivalent to or subsumed by TMs. *There is no more powerful machine* (Technically cannot be proved).
- DFAs and PDAs do not model all effective procedures or computable functions, but only a subset.

Deterministic Turing Machine (DTM)



- Two-way, infinite tape, broken into cells, each containing one symbol.
- Two-way, read/write tape head.
- An input string is placed on the tape, padded to the left and right infinitely with blanks, read/write head is positioned at the left most input character.
- Finite control (read/write head is part of this control), knows current symbol being scanned, and its current state.

Deterministic Turing Machine (DTM)



- In one move, depending on the current state and the current symbol being scanned, the TM does: (1) changes **state**, (2) **prints** a symbol over the cell being scanned, and (3) moves its' tape head one cell **left** or **right**.
- Many modifications possible, but **Church-Turing** declares equivalence of all.

Formal Definition of a DTM

- A DTM is a **seven-tuple**:

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

Q A finite set of states

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

Γ A finite tape alphabet, which is a strict superset of Σ

B A distinguished blank symbol, which is in Γ

q_0 The initial/starting state, q_0 is in Q

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

δ A next-move function, which is a *mapping* (i.e., may be undefined)
 $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$

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

- **Example #1:** $\{w \mid w \text{ is in } \{0,1\}^* \text{ and } w \text{ ends with a } 0\}$

$$= \{0, 00, 10, 10110, \dots\}$$

Note: ϵ is not in the language

- **Example #1:** $\{w \mid w \text{ is in } \{0,1\}^* \text{ and } w \text{ ends with a } 0\}$

$$= \{0, 00, 10, 10110, \dots\}$$

Note: ϵ is not in the language

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

$$\Gamma = \{0, 1, B\}$$

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

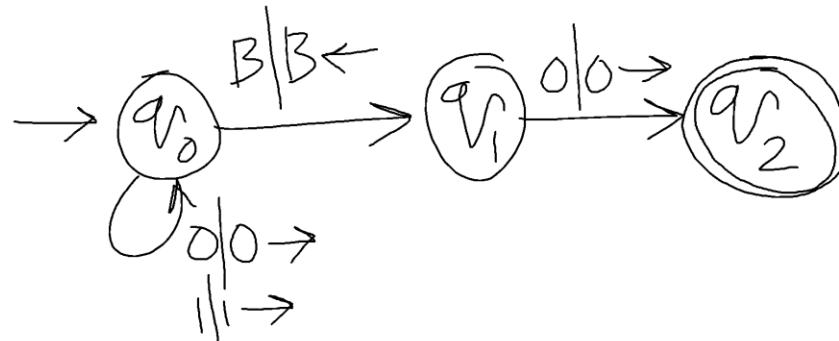
$$F = \{q_2\}$$

δ :

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

- q_0 is the start state and the “scan right” state, until hits B
- q_1 state is to begin the verification – last character is 0 or not
- q_2 is the final state

- Example #1: $\{w \mid w \text{ is in } \{0,1\}^* \text{ and } w \text{ ends with a } 0\}$



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

$$\Gamma = \{0, 1, B\}$$

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

$$F = \{q_2\}$$

δ :

	0	1	B
->q ₀	(q ₀ , 0, R)	(q ₀ , 1, R)	(q ₁ , B, L)
q ₁	(q ₂ , 0, R)	-	-
q ₂ *	-	-	-

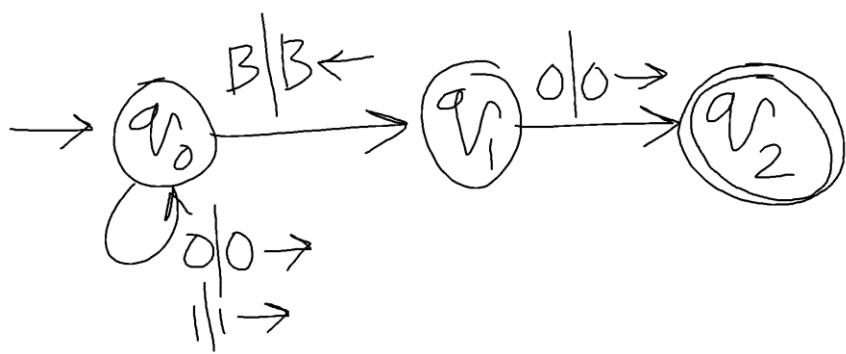
- q_0 is the start state and the “scan right” state, until hits B
- q_1 state is to begin the verification – last character is 0 or not
- q_2 is the final state

ID

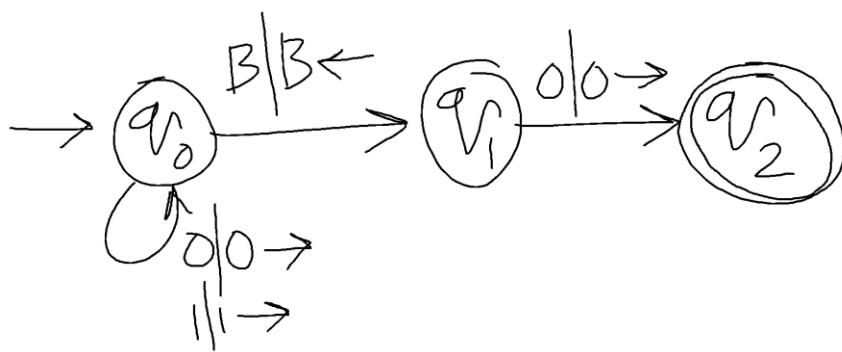
- To describe the steps we use IDs.
- An ID is $\alpha q \beta$
- $\alpha, \beta \in \Gamma^*$
- $\alpha \beta$ is on the tape. Left and right of this are blanks.
- Head is on the first character of β
- The TM is in state q

Step

- $id_1 \vdash id_2$ describes one step
- \vdash^* is reflexive and transitive closure of \vdash



- For input 1010, computation steps...



- For input 1010, computation steps...

$$\begin{aligned}
 & q_0 1010 + 1q_0 010 + 10q_0 10 \\
 & \vdash 101q_0 0 + 1010q_0 B \\
 & \vdash 101q_0 0 + 1010q_2
 \end{aligned}$$

Have you noticed

- Recognition is immediate.
 - TM, once hits one of the final states, it accepts and halts.
- No need for the input to be exhausted !!
- How TM rejects?
 - It gets stuck.
 - It goes on infinitely without ever hitting a final state.

- $\{0^n 1^n | n \geq 1\}$
- Idea??

- $\{0^n 1^n | n \geq 1\}$
- Idea??

- $\{0^n 1^n \mid n \geq 1\}$
- Idea??

B	B	0	0	0	1	1	1	B
		↑						

q_0

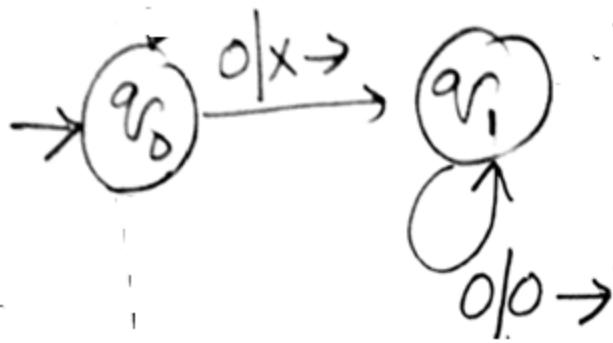
- $\{0^n 1^n \mid n \geq 1\}$

- Idea??

• We turn
0 in to X and
probe for 1.
We turn 1 to Y.

B	B	0	0	0	1	1	1	B
		↑ q_0						

B	B	X	0	0	1	1	1	B
			↑ q_1					



B	B	0	0	0	1	1	1	B
		↑ q_0						

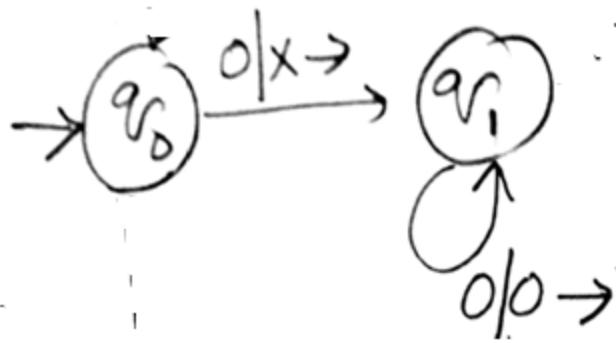
B	B	X	0	0	1	1	1	B
			↑ q_1					

q_1

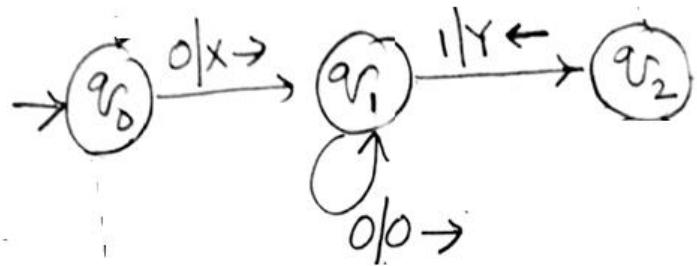
⋮

B	B	X	0	0	1	1	1	B
					↑ q_1			

q_1



B	B	0	0	0	1	1	1	B
		↑ q_0						
B	B	X	0	0	1	1	1	B
			↑ q_1					
						⋮		
B	B	X	0	0	1	1	1	B
					↑ q_1			
B	B	X	0	0	Y	1	1	B
				↑ q_2				



B	B	0	0	0	1	1	1	B
		↑ q_0						

B	B	X	0	0	1	1	1	B
			↑					

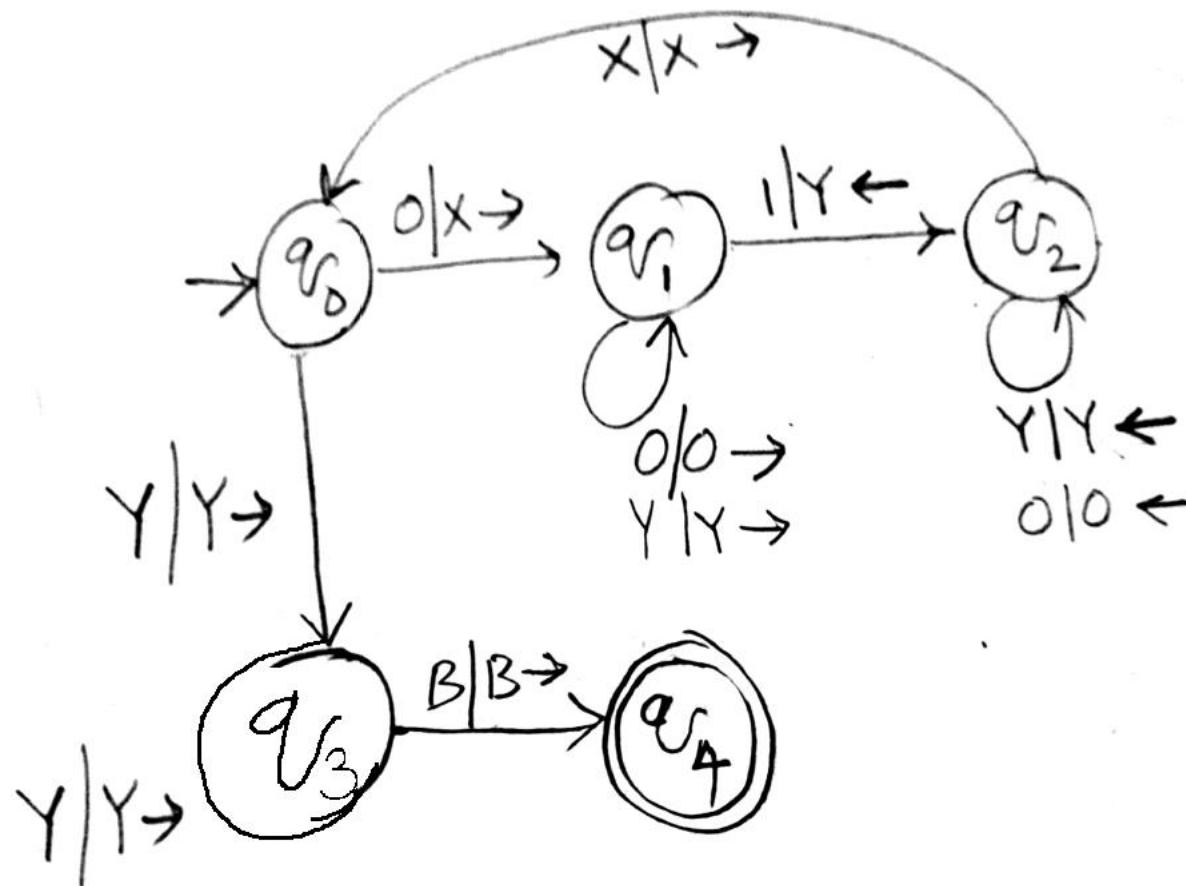
q_1

⋮

B	B	X	0	0	1	1	1	B
					↑ q_1			

B	B	X	0	0	Y	1	1	B
				↑				

q_2

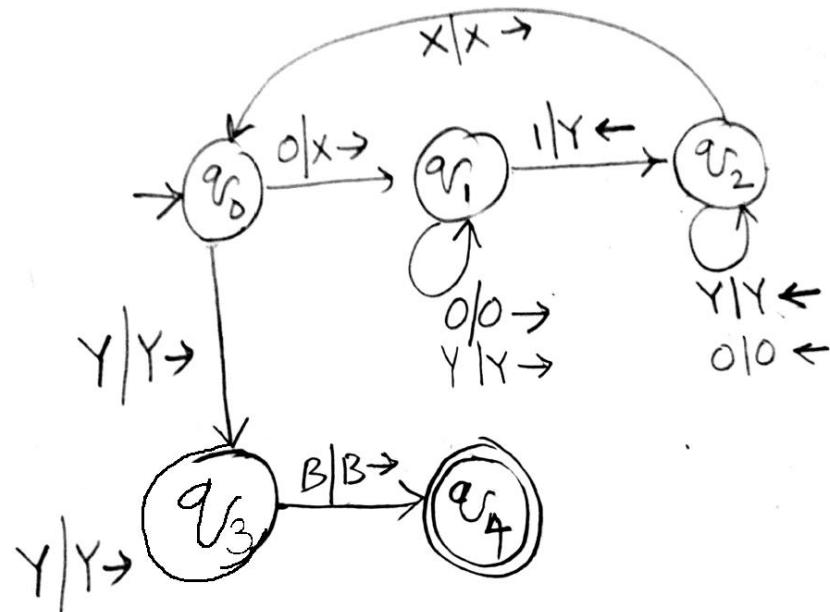


- **Example #2:** $\{0^n1^n \mid n \geq 1\}$

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

- **Sample Computation:** (on input: 0011),

$q_00011BB.. \xrightarrow{} Xq_1011$
 $\xrightarrow{} Xq_111$
 $\xrightarrow{} Xq_20Y1$
 $\xrightarrow{} q_2X0Y1$
 $\xrightarrow{} Xq_00Y1$
 $\xrightarrow{} XXq_1Y1$
 $\xrightarrow{} XXYq_11$
 $\xrightarrow{} XXq_2YY$
 $\xrightarrow{} Xq_2XXX$
 $\xrightarrow{} XXq_0YY$
 $\xrightarrow{} XXYq_3YB...$
 $\xrightarrow{} XXYYq_3BB...$
 $\xrightarrow{} XXYYBq_4$



TM

- TM can be seen as a recognizer.
- TM can be seen as a input-output mapper.
- TM can also be seen as an enumerator.
 - Generates the language one string after the other.

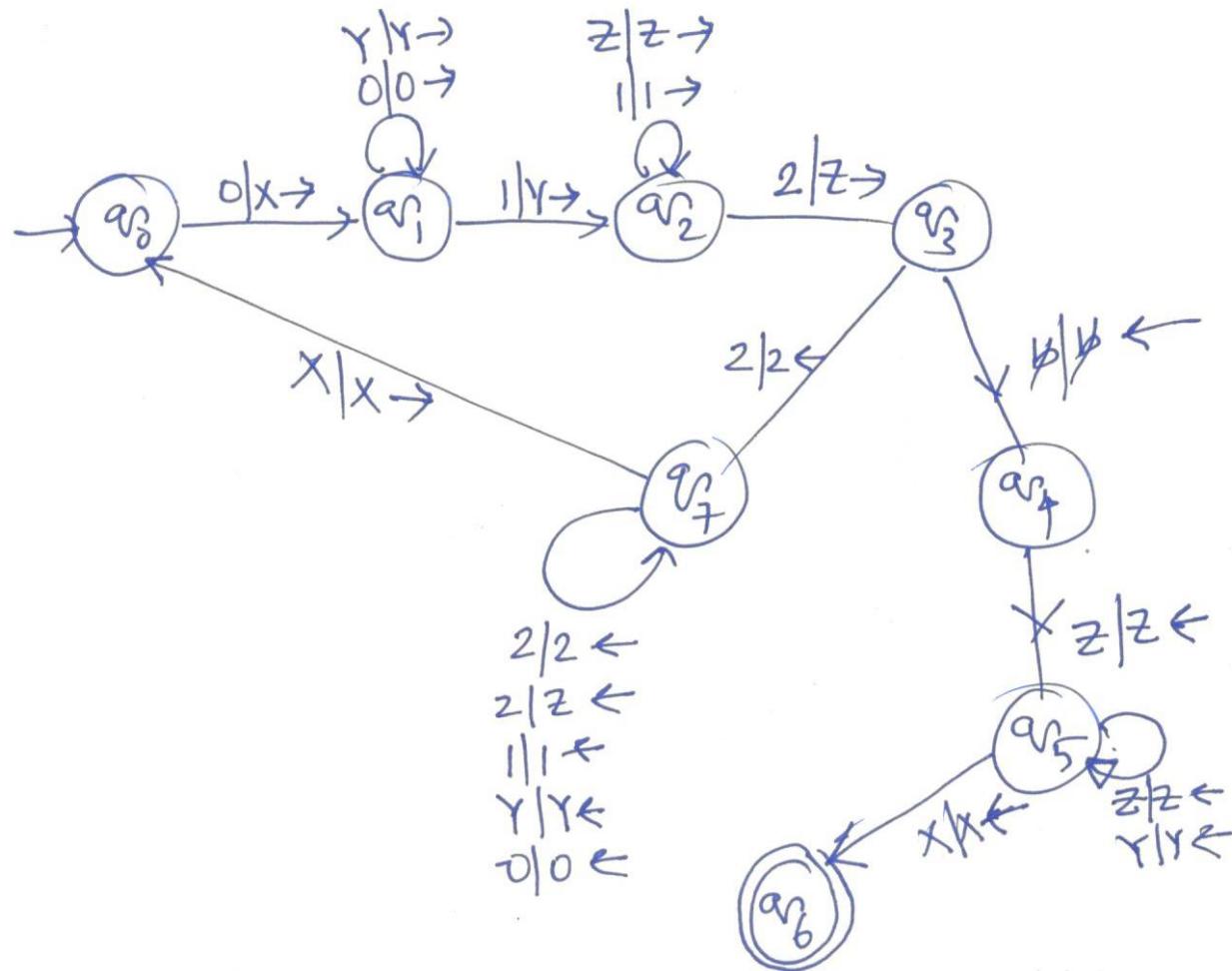
Eg: $\Sigma = \{0, 1, 2\}$, $\Gamma = \{\alpha, \gamma, z, b\}$ $L = \{0^n 1^n 2^n \mid n \geq 1\}$

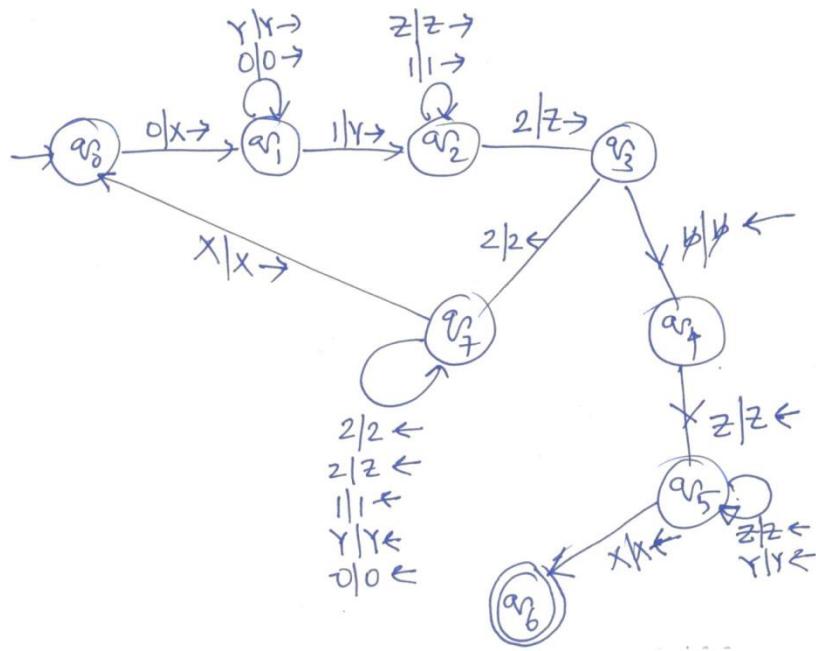
- But, there is a mistake in this.

Eg: $\Sigma = \{0,1,2\}$, $\Gamma = \{\alpha, \gamma, z, b\}$ $L = \{0^n 1^n 2^n | n \geq 1\}$

- But, there is a mistake in this.
- Γ has to be a superset of Σ

Fg: $\Sigma = \{0, 1, 2\}$, $\Gamma = \{\otimes, Y, Z, \otimes\}$ $L = \{0^n 1^n 2^n \mid n \geq 1\}$





$$w = 001122$$

$q_0 001122 \vdash X q_1 001122 \vdash X0 q_1 1122 \vdash X0Y q_2 1122$

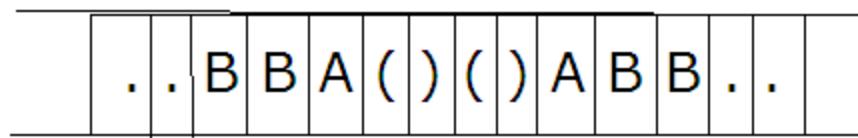
$\vdash X0Y1 q_2 22 \vdash X0Y1Z q_3 2 \vdash X0Y1 q_7 Z 2$

$\vdash^* q_7 X0Y1Z 2 \vdash X q_0 0Y1Z 2 \vdash^* XXYYZZ q_3 \beta$

$\vdash^* X q_5 XYZZ \vdash q_6 XXYYZZ$

TM that gives output

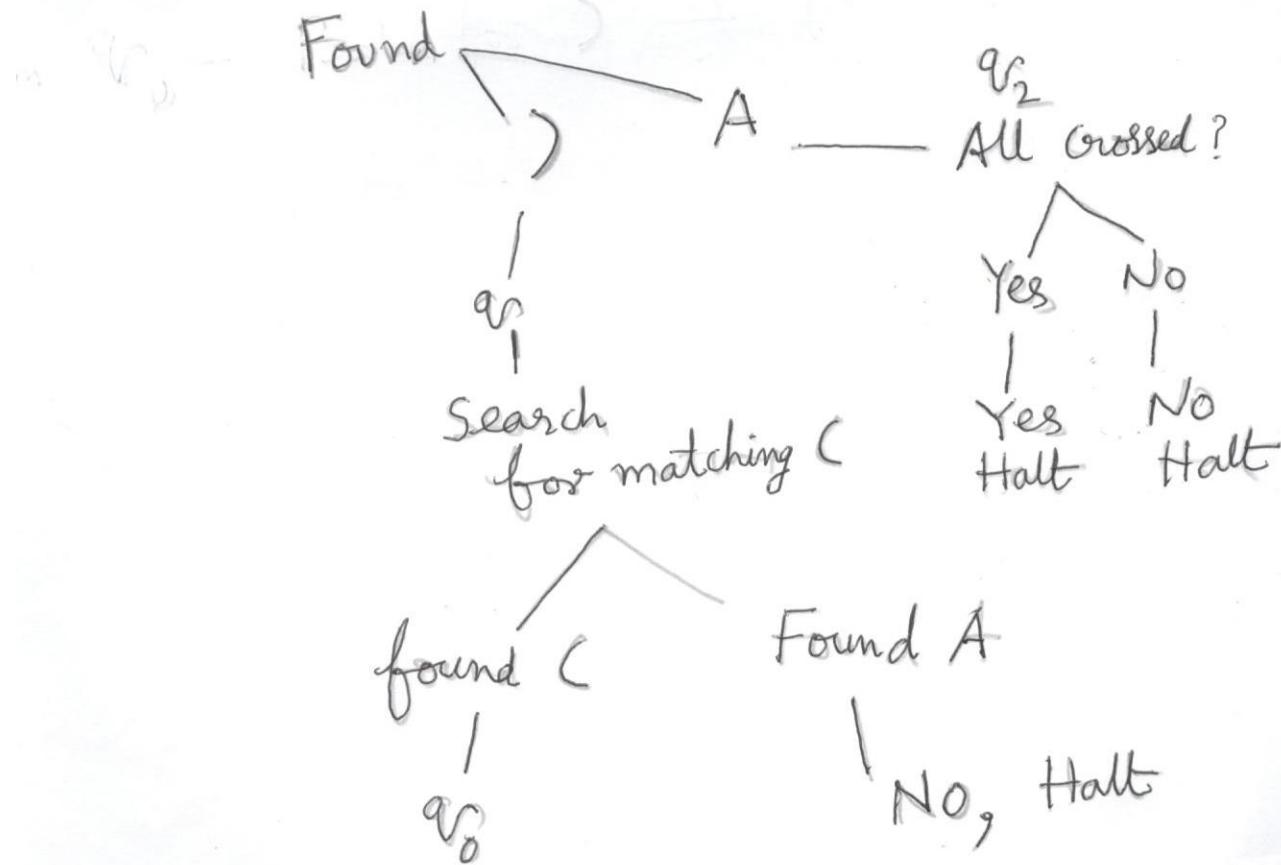
- Example:
- $\Sigma = \{(,)\}$
- L is set of well formed parentheses.
- $\Gamma = \{A, X, B, Y, N\}$
- We use B or $\textcolor{brown}{\emptyset}$ to mean the blank
- To simplify, we assume the given string is embedded between As



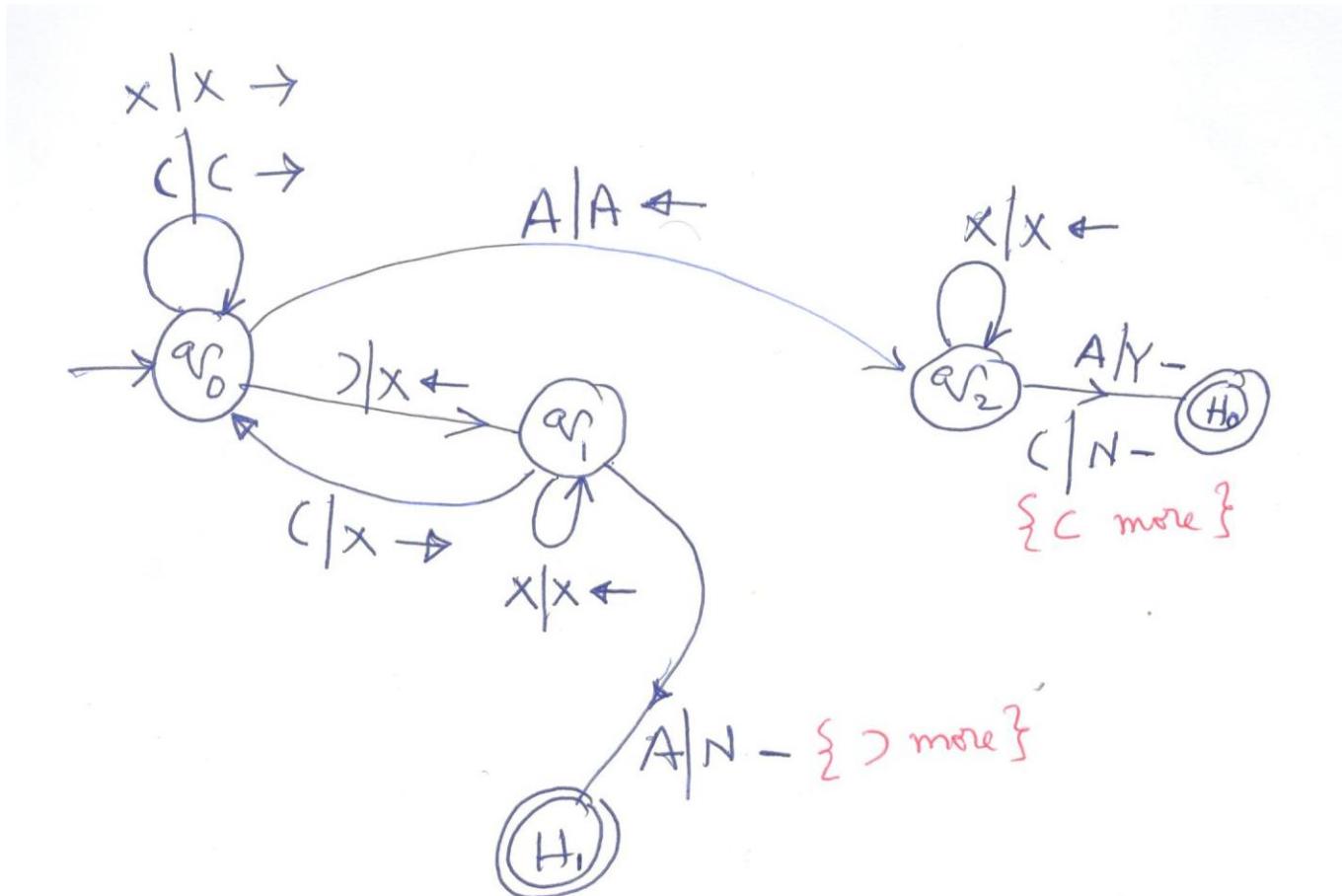
- The TM halts with a Y to mean Yes or a N to mean No.
- $Q = \{ q_0, q_1, q_2, H_0, H_1 \}$
- $F = \{H_0, H_1\}$

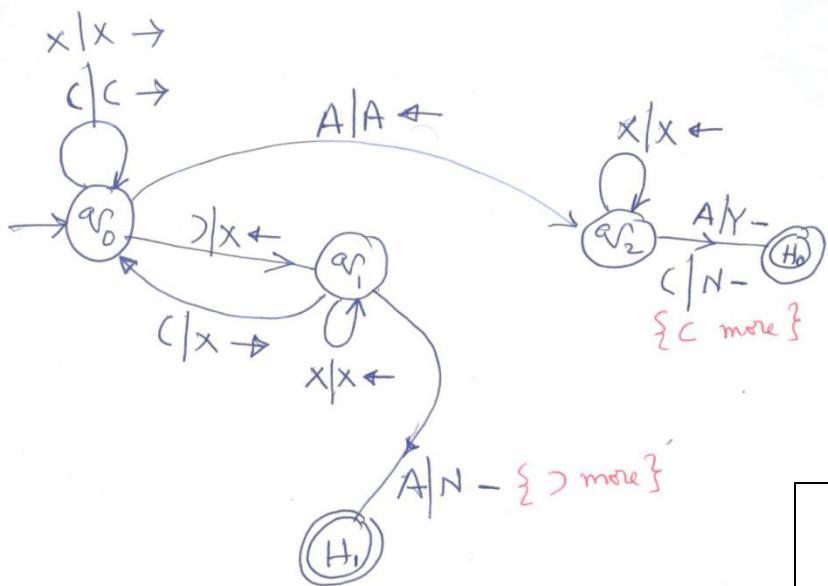
Idea

$v_0 \rightarrow \text{Begin} - \text{Probe for })$



The transition diagram





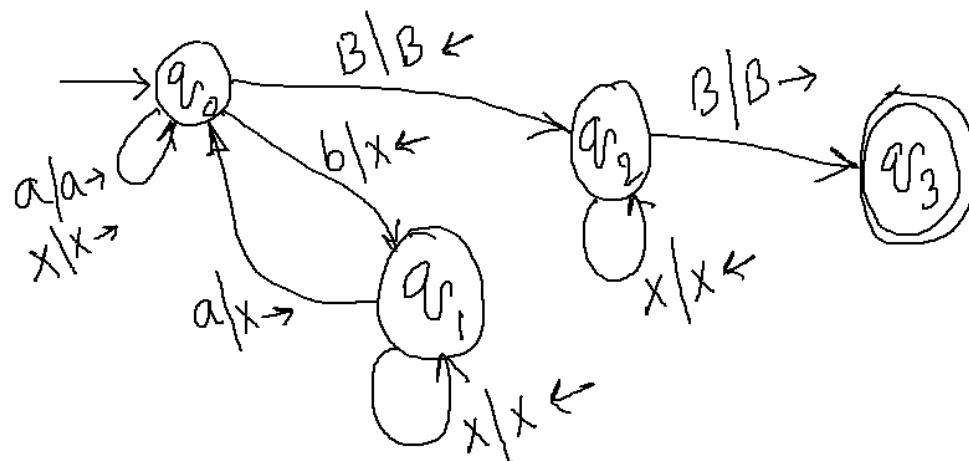
$\text{y y A () () () A y y}$
 \uparrow
 q_0

$A q_0 () () () A \vdash A (q_0) () () A$
 $\vdash A q_1 (x () ()) A \vdash A x q_0 x () () A$
 $\vdash A x x q_0 () () A \vdash A x x (q_0 () ()) A$
 $\vdash A x x (C q_0) () A \vdash A x x (q_0 C) () A$
 $\vdash A x x (X q_0) () A \vdash A x x (X X q_0) () A$
 $\vdash A x x (x x q_1 (x)) A \vdash A x x (x x x q_0 x) A$
 $\vdash A x x (x x x x q_0) A \vdash A x x (x x x x x q_0) A$
 $\vdash A x x x x x x q_0 A \vdash A x x \dots x q_2 A$
 $\vdash q_2 A x x \dots x A \vdash H_0 Y x x - x A$
 $\boxed{\begin{matrix} \swarrow & \searrow \\ \text{Halt} & \text{q1} \end{matrix}}$

For the same problem (well balanced parenthesis) as a recognition problem

- Let (and) are represented as a and b, respectively. (B is the blank symbol)

	B	X	a	b
$\rightarrow q_0$	(q2,B,L)	(q0,X,R)	(q0,a,R)	(q1,X,L)
q1		(q1,X,L)	(q0,X,R)	
q2	(q3,B,R)	(q2,X,L)		
*q3				



The language accepted by a TM M

- Let M be a TM, $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$
- $L(M) = \{w \in \Sigma^* \mid q_0 w \vdash^* ap\beta, \text{ where } p \in F, \text{ and } \alpha, \beta \in \Gamma^*\}.$
- Note that, input string getting exhausted is not present (in contrast to the acceptance criterion by PDAs, DFAs, etc).
- The language accepted by a TM is also said the language recognized by a TM.

Recursively enumerable

- If $w \in L(M)$, then M accepts/recognizes the string w.
- If, $w \notin L(M)$, then M may or may not halt.
 - No transition means M halts and rejects.
- We say $L(M)$ for a given TM M is recursively enumerable (RE).

Recursive languages

- Recursive languages (R) is a subset of RE.
- We say $L(M)$ for a TM M is recursive, if for any given input string w , M halts.
- That is, if $w \in L(M)$, M halts in an accepting (final) state.
- Else, M halts in a non-final state.
 - i.e., It gets stuck in a non-final state.
- M never goes into an infinite loop.

RE Vs. R

RE

- A TM M recognizes.

R

- A TM M decides.

RE Vs. R

RE

- A TM M recognizes.
- If $L \in RE - R$, then complement of L , that is $\bar{L} \notin RE$.

R

- A TM M decides.
- $L \in R \Leftrightarrow \bar{L} \in R$

RE Vs. R

RE

- A TM M recognizes.
- If $L \in RE - R$, then complement of L , that is $\bar{L} \notin RE$.

R

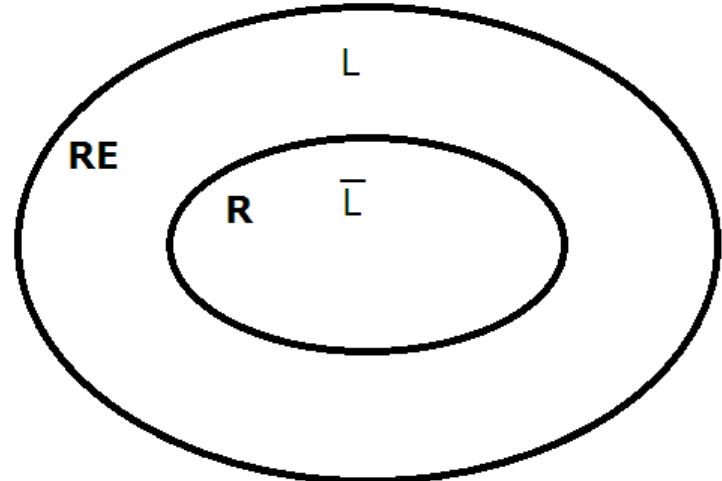
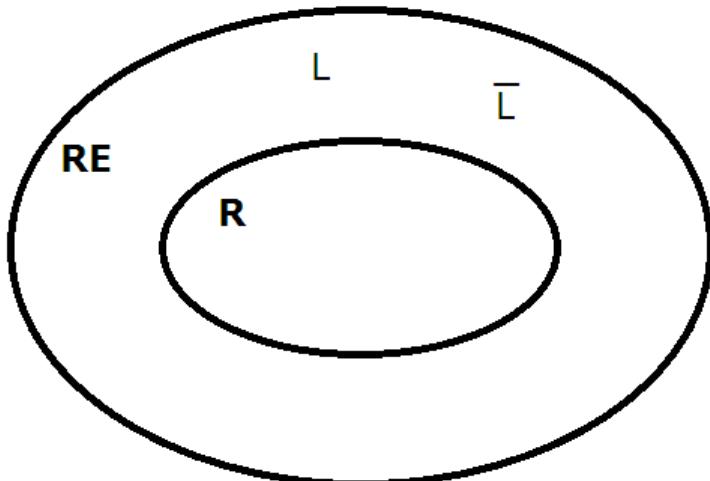
- A TM M decides.
- $L \in R \Leftrightarrow \bar{L} \in R$

$L \in RE \text{ and } \bar{L} \in RE$

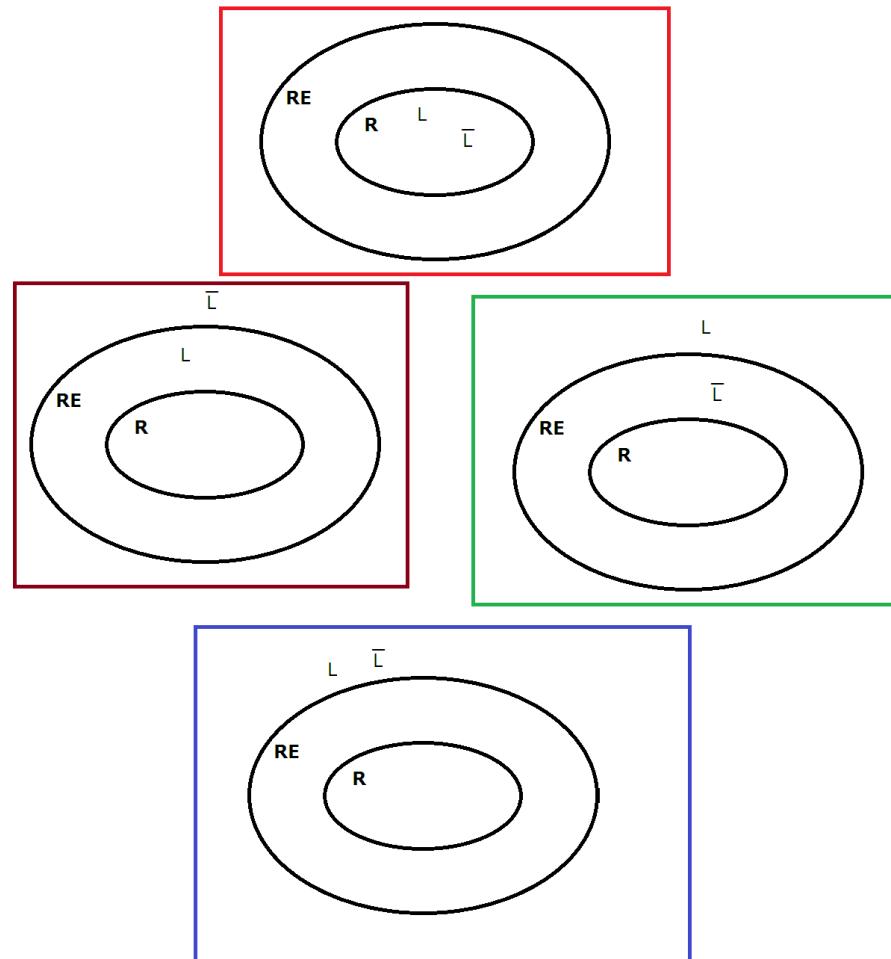
\Rightarrow

$L \in R$

Not Possible



Possible



Turing Machines-- Machines entering into infinite loop

Supplement Slides

Recognizing or accepting a language by a machine (automaton)

- Let us define ID for a DFA/NFA as
 - (q, x) where q is the current state and x is the remaining input.
- We say L is recognized/accepted by a machine M (may be a DFA/NFA/PDA/TM)
$$L = \{w \in \Sigma^* | id_0 \vdash^* id_f \text{ where } id_0 \text{ and } id_f \text{ are initial and accepting ids, respectively}\}$$

DFA/NFA

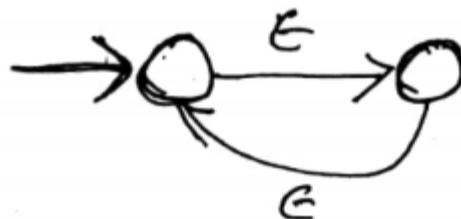
- $L = \{w \in \Sigma^* \mid (q_0, w) \vdash^* (q_f, \epsilon) \text{ where } q_f \text{ is a final state}\}$
- Input has to be exhausted. { The remaining string should become ϵ }
- This is the same criterion even for NFAs.

DFA

- A DFA **never** enters in to an infinite loop.
 - Since there are no ϵ transitions, and
 - There is exactly one choice at every stage of the computation.
 - The input has to exhaust progressively (one character at each step) and should become ϵ
 - At this stage if the state is one of final, the input string is accepted, else rejected.

NFA

- A NFA can enter in to an infinite loop.



- For any given input this NFA enters in to an infinite loop.
- The language recognized by this machine is ϕ
- The language ϕ is regular, because there is a NFA to recognize this.

For this language we can find NFA and DFA that always halts.

- NFA

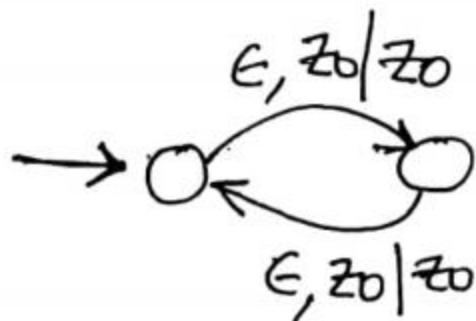


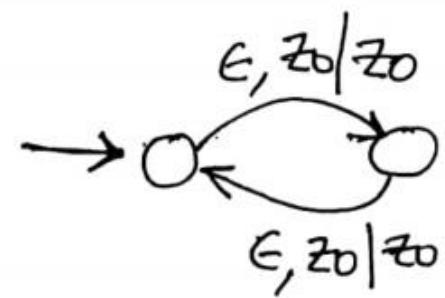
- There is a DFA (existence of either NFA or DFA is enough) also to accept ϕ . {DFA always halts}.



PDA by final state and empty stack

- For a PDA to recognize a language by final state, $L = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q_f, \epsilon, \alpha) \text{ where } q_f \text{ is a final state and } \alpha \in \Gamma^*\}$
- A PDA can also enter in to an infinite loop





- The language accepted by this PDA is also ϕ .
- we can find a PDA which recognizes the language ϕ without entering in to an infinite loop.

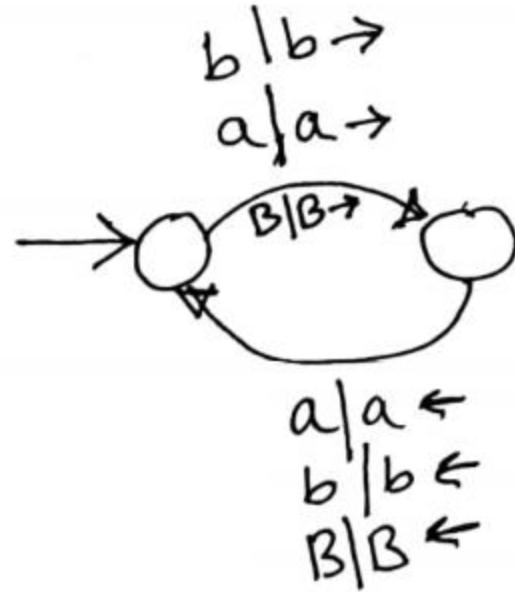


- Have you noted, both these PDAs are indeed DPDAs.

- For PDA by empty stack also similar arguments can be given.

TM

- For the given DTM
also the language
recognized is \emptyset .

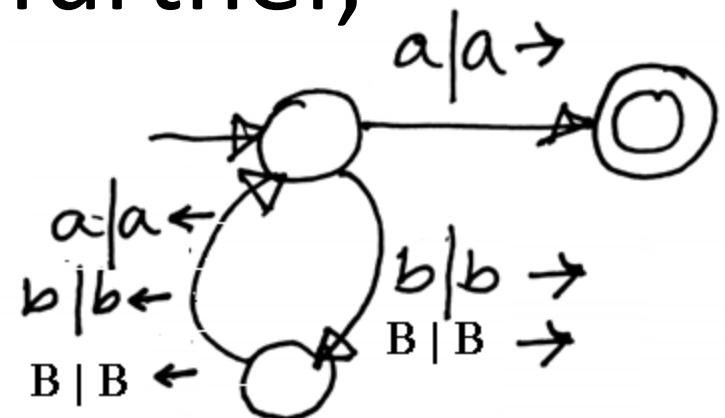


- Again there is a DTM that
Recognizes this language without entering in to
infinite loop, which is

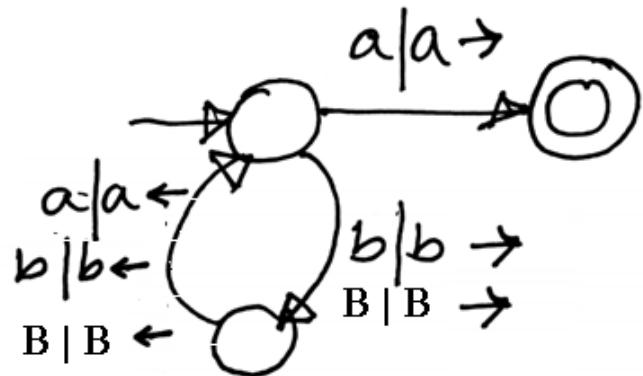


- So, $\phi \in RE$
- In fact $\phi \in R$ also.

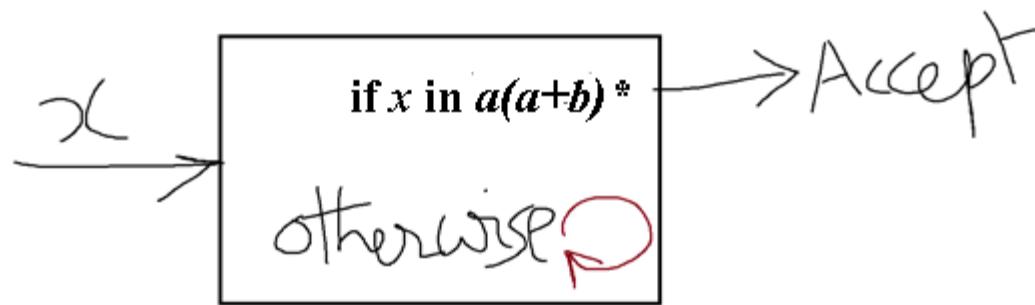
To elucidate further,



- The given DTM recognizes the language $a(a + b)^*$, but enters in to an infinite loop for all other inputs.
- So, the language recognized by this DTM is $a(a + b)^*$
- This is in RE.



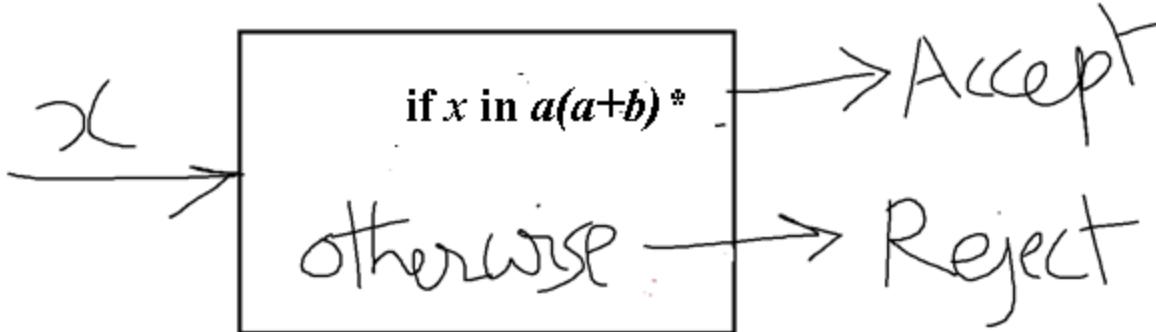
- Behaviour of this machine is



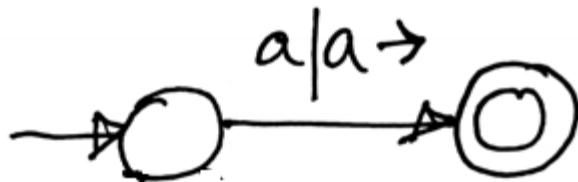
- so, the language $a(a + b)^* \in RE$

Can we say $a(a + b)^*$ is in R ?

- Yes.



- We need a TM which behaves like above



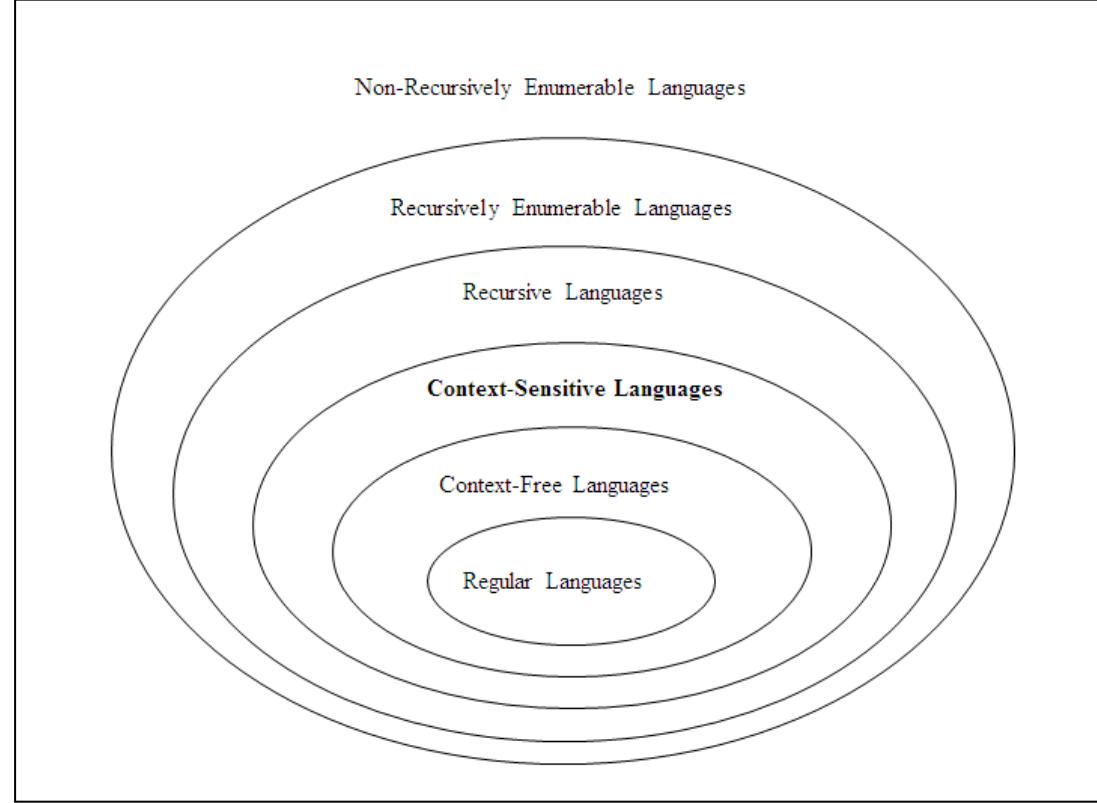
- Note, existence of one such TM is enough.

- Are there languages which are in RE but not in R ?
- Yes.
- Finding an $L \in RE - R$ is a **challenging** task.



Finding an $L \in RE - R$ is a challenging task

- That is, for any string in L, the machine should halt in a final state.
 - There is a string which is not in L, for which every possible TM (which accepts L) enters in to an infinite loop.
-
- There exists such languages !!
 - This one important aspect of the theory of computation.
 - This, indeed points at limitations of computing machines.



- This diagram is saying that regular, context free and some other languages are recursive.
- That is, all these languages are Turing Decidable.

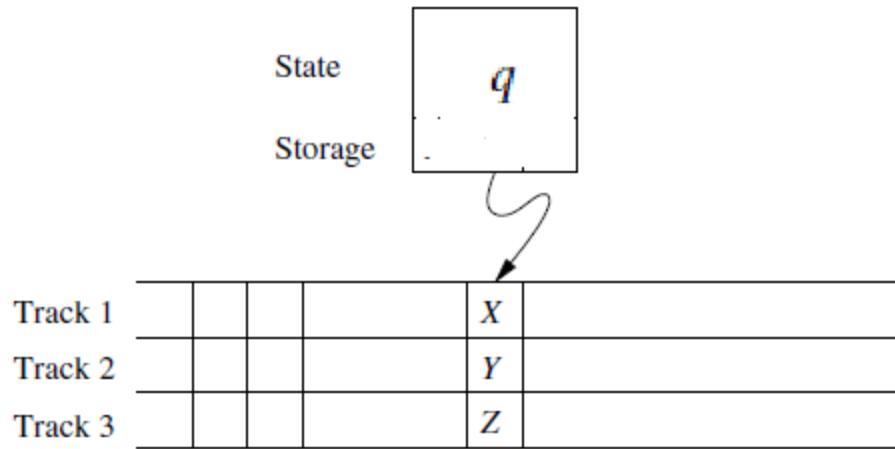
- We can say, for a regular language their exists a DFA (and also a NFA) that decides the language.
- Similarly for a CFL, a PDA that decides the language always exists.
- So, we do not need powerful TMs to decide regular and CFLs.

Variants of TM

Multi-track, multi-tape, NTM

<https://www.andrew.cmu.edu/user/ko/pdfs/lecture-14.pdf>

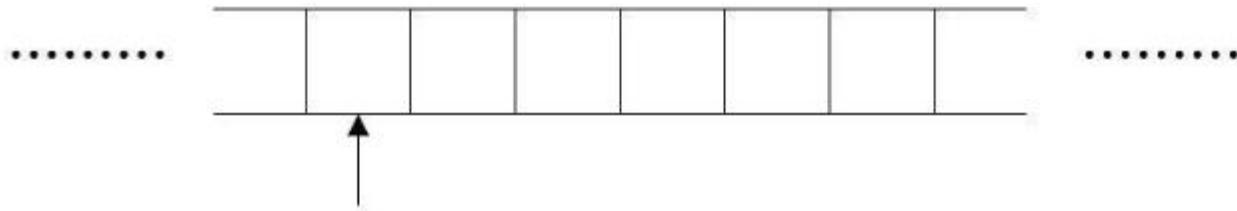
Multi-track



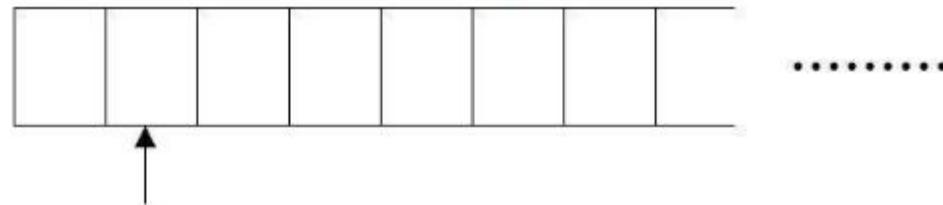
- In this example, the tape symbol is the triplet (X, Y, Z) and we can see the tape as a single-track one.

Semi-infinite tape

Standard machine

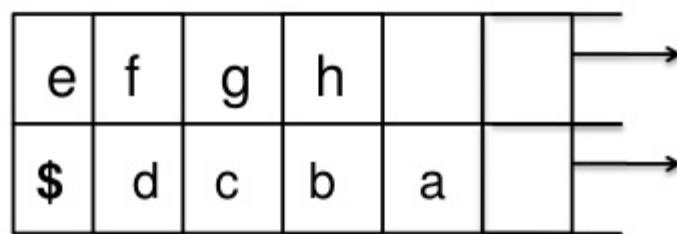
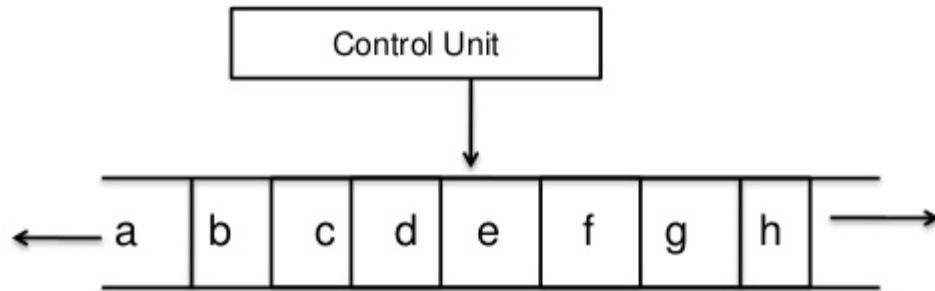


Semi-infinite tape machine



Simulation of two way infinite by semi-infinite tape

- Two way infinite tape simulated by semi -infinite tape



Multi-tape

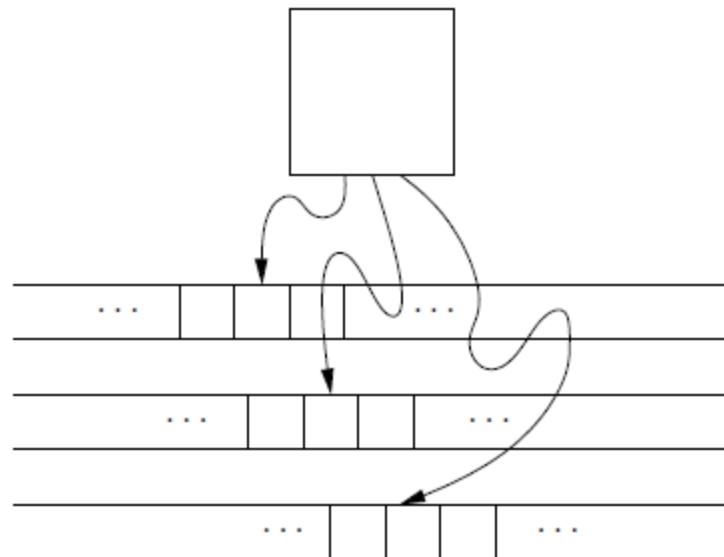


Figure 8.16: A multitape Turing machine

Simulation of multi-tape by one-tape

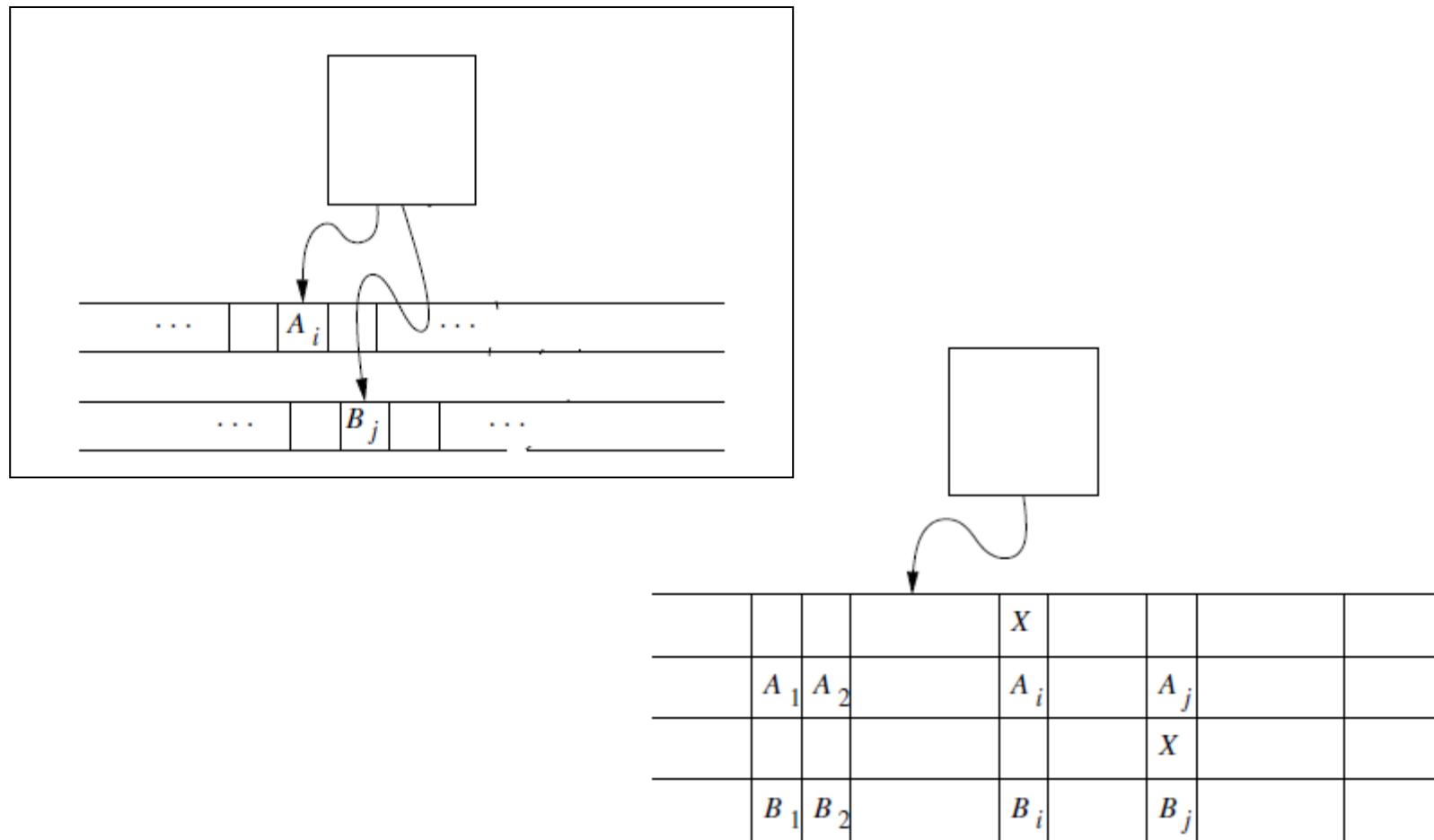


Figure 8.17: Simulation of a two-tape Turing machine by a one-tape Turing machine

NTM

NONDETERMINISTIC TM

- There is a choice in the next move.

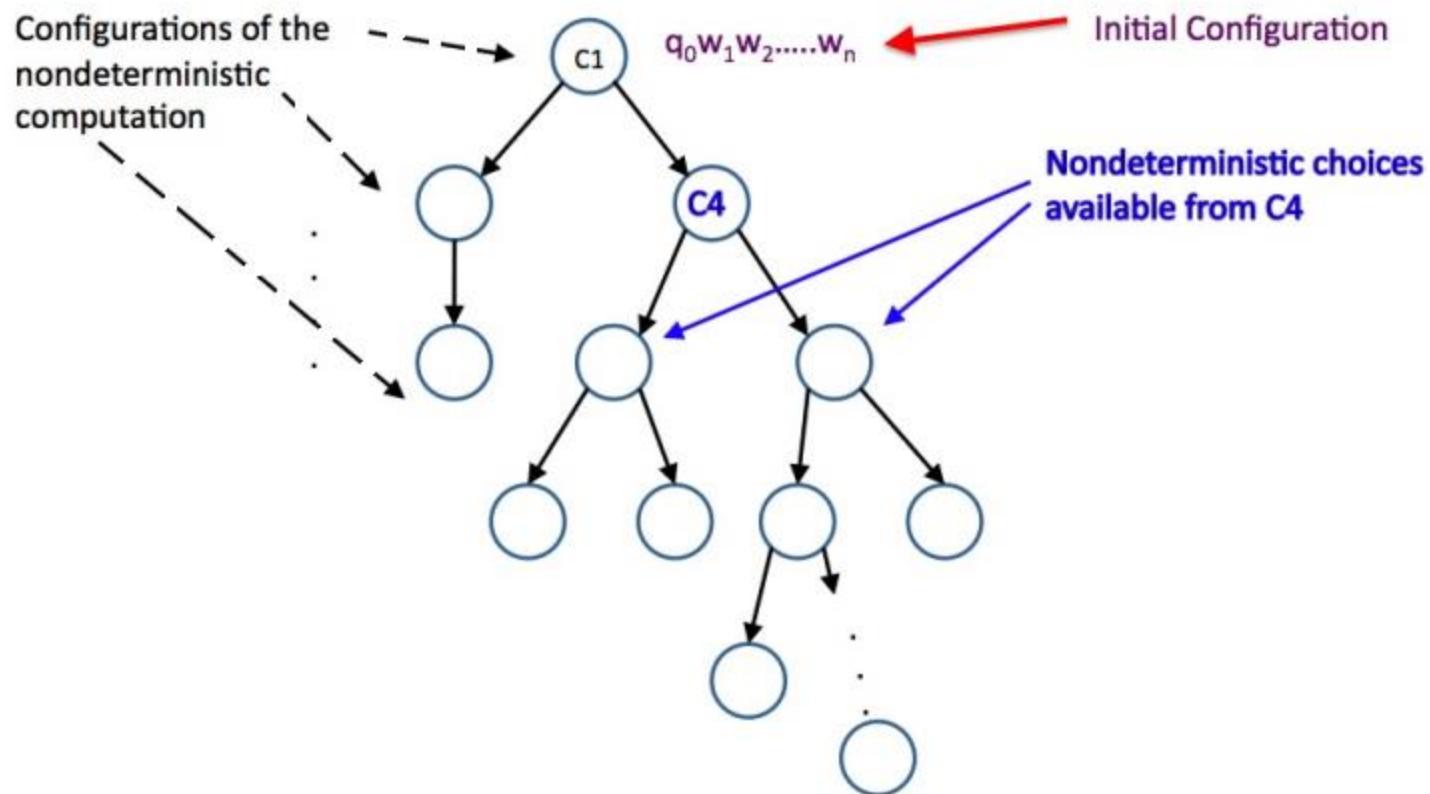
$$\delta(q, X) = \{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \dots, (q_k, Y_k, D_k)\}$$

- Here, Y_i is a tape symbol, and D_i is one from $\{L, R\}$, the direction of movement of the head.

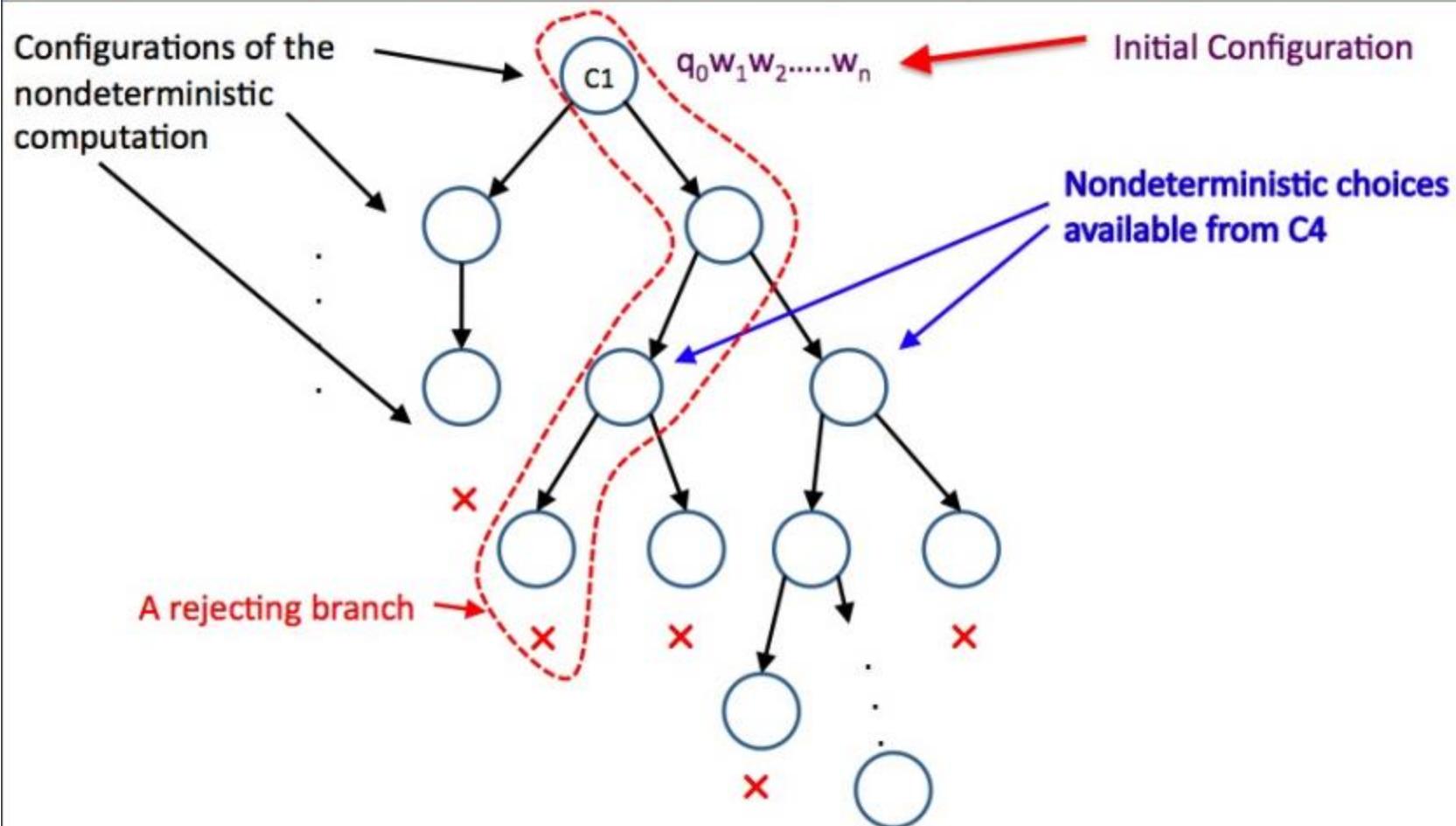
The transition function for a nondeterministic Turing machine has the form

$$\delta: Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

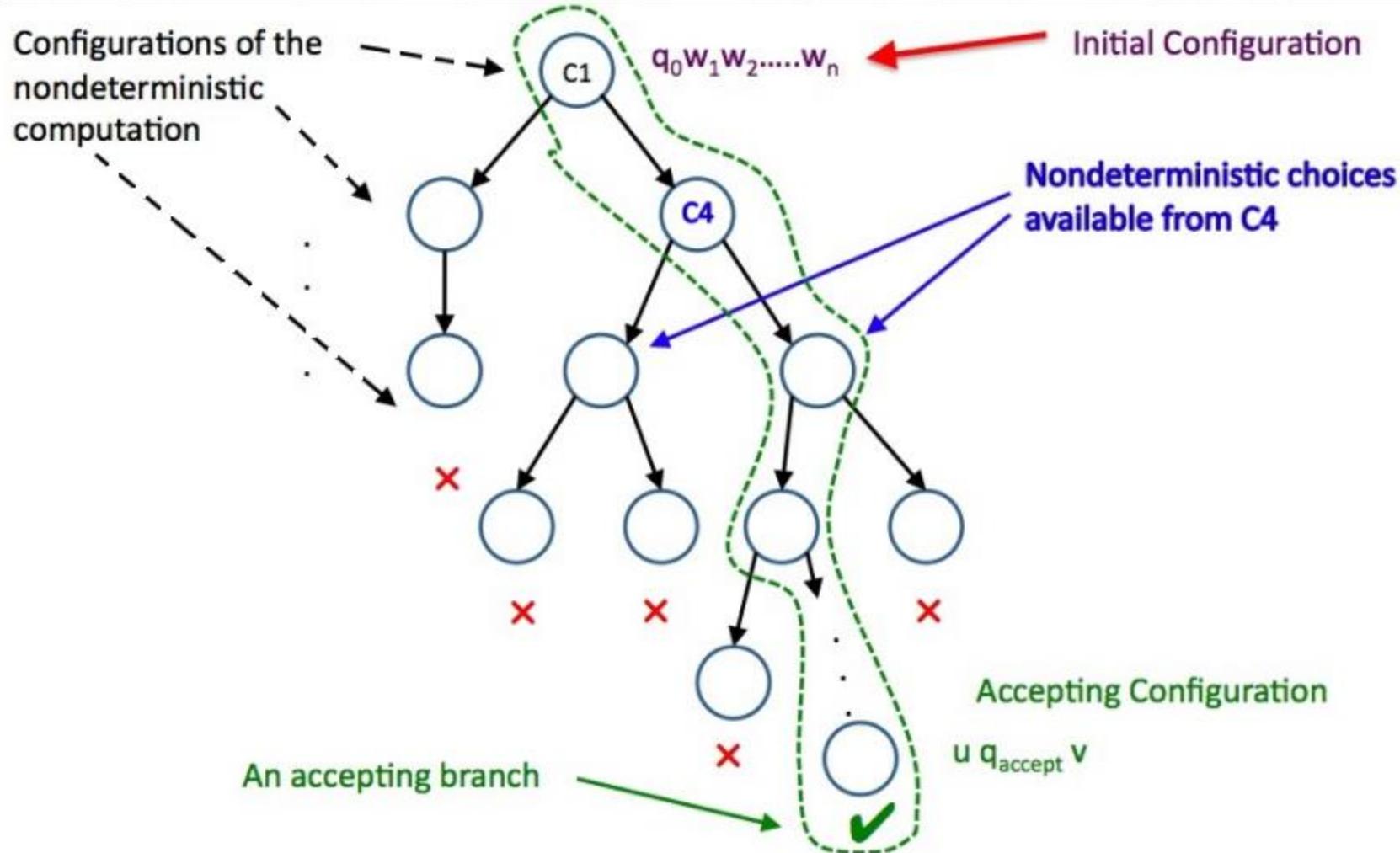
NONDETERMINISTIC COMPUTATION



NONDETERMINISTIC COMPUTATION

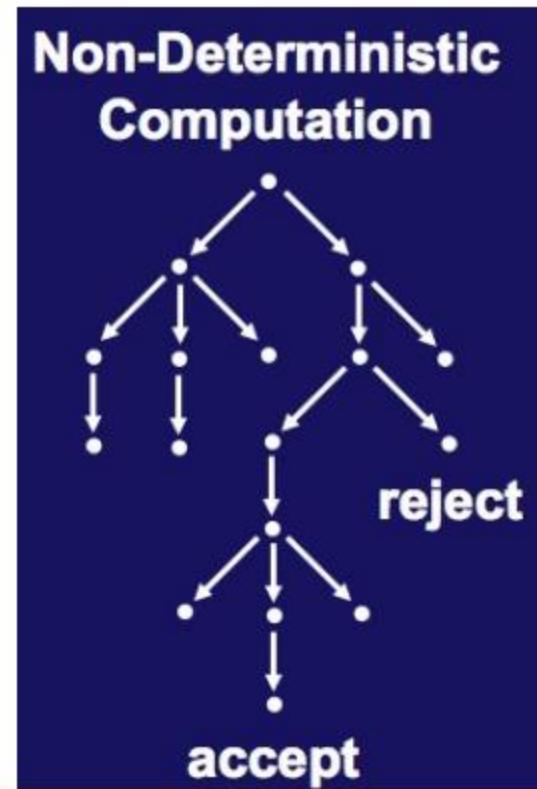
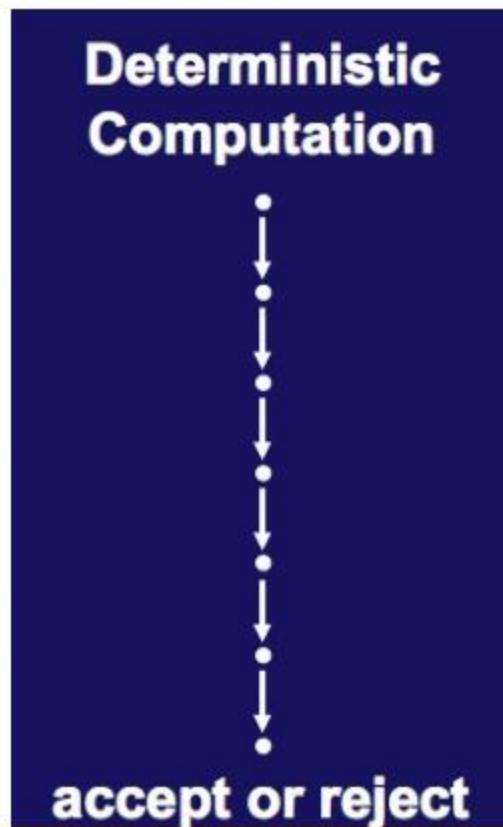


NONDETERMINISTIC COMPUTATION



NONDETERMINISTIC TURING MACHINES

- A computation of a Nondeterministic TM is a tree, where each branch of the tree is looks like a computation of an ordinary TM.



NONDETERMINISTIC TURING MACHINES

- If a single branch reaches the accepting state, the Nondeterministic TM accepts, even if other branches reach the rejecting state.
- What is the power of Nondeterministic TMs?
 - Is there a language that a Nondeterministic TM can accept but no deterministic TM can accept?
- Note, NTM rejects means
 - Each of the branch either explicitly rejects (by getting stuck in a non-final state), or goes in to an infinite loop.

NONDETERMINISTIC TURING MACHINES

THEOREM

Every nondeterministic Turing machine has an equivalent deterministic Turing Machine.

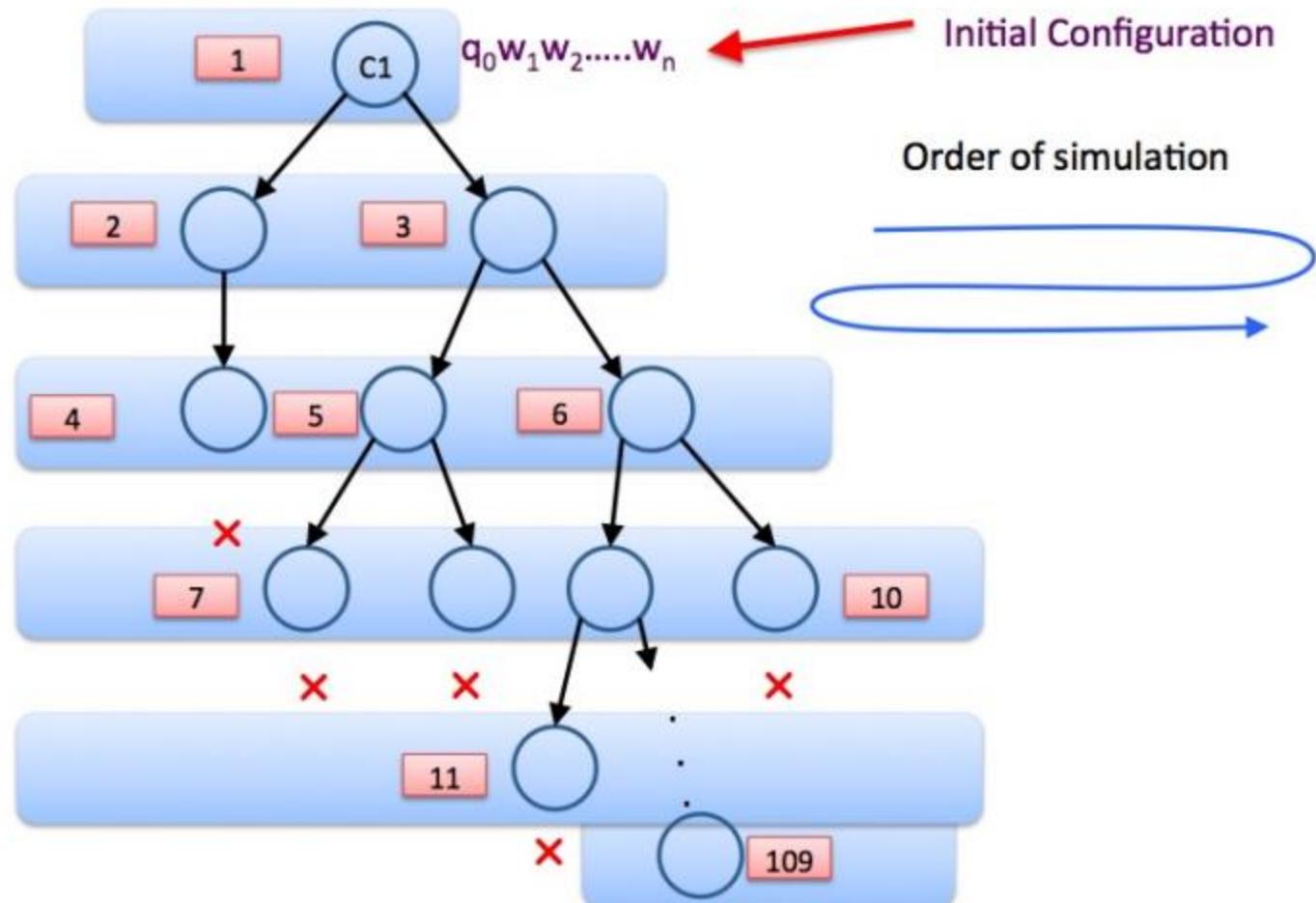
PROOF IDEA

- Timeshare a deterministic TM to different branches of the nondeterministic computation!
- Try out all branches of the nondeterministic computation until an accepting configuration is reached on one branch.
- Otherwise the TM goes on forever.

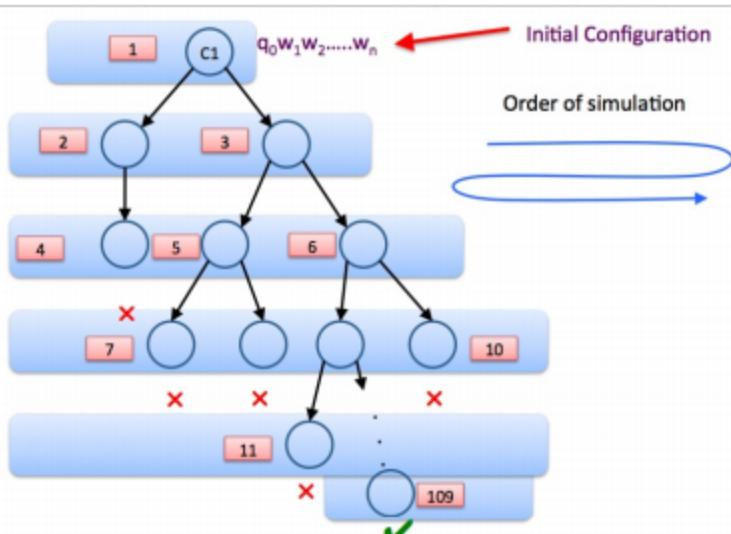
NONDETERMINISTIC TURING MACHINES

- Deterministic TM D simulates the Nondeterministic TM N .
- Some of branches of the N 's computations may be infinite, hence its computation tree has some infinite branches.
- If D starts its simulation by following an infinite branch, D may loop forever even though N 's computation may have a different branch on which it accepts.
- This is a very similar problem to processor scheduling in operating systems.
 - If you give the CPU to a (buggy) process in an infinite loop, other processes “starve”.
 - In order to avoid this unwanted situation, we want D to execute all of N 's computations concurrently.

SIMULATING NONDETERMINISTIC COMPUTATION



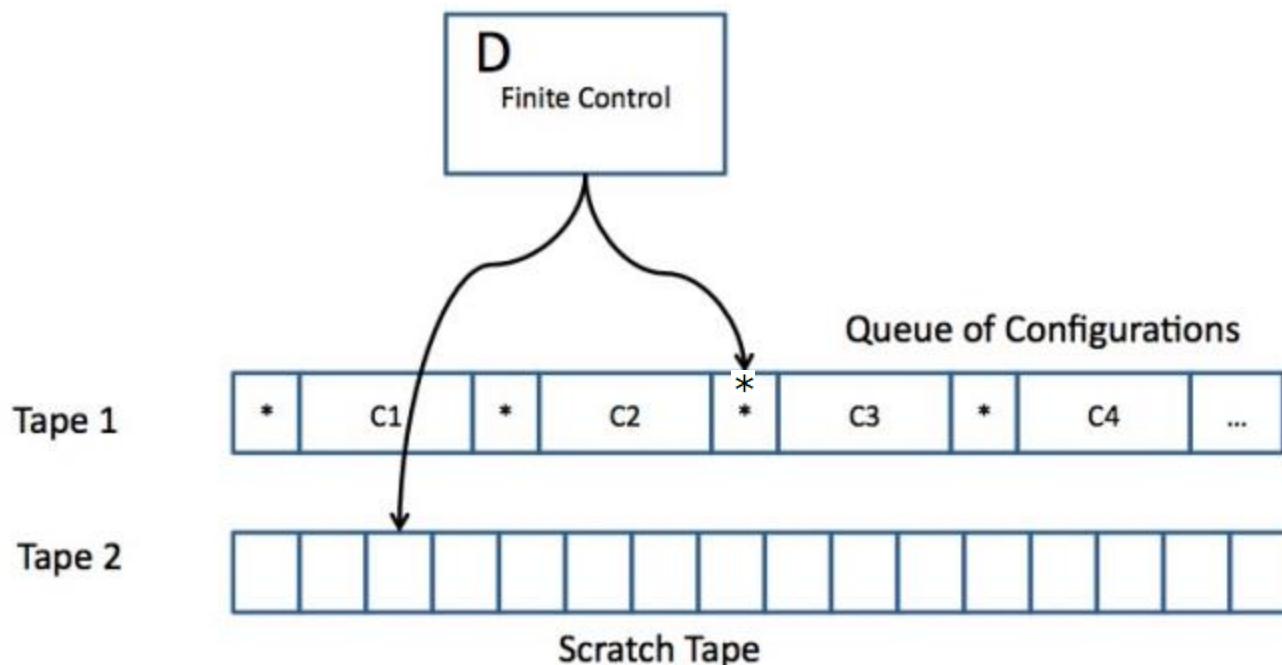
SIMULATING NONDETERMINISTIC COMPUTATION



- During simulation, D processes the configurations of N in a **breadth-first fashion**.
- Thus D needs to maintain a **queue** of N 's configurations (Remember queues?)
- D gets the next configuration from the head of the queue.
- D creates copies of this configuration (as many as needed)
- On each copy, D simulates one of the nondeterministic moves of N .
- D places the resulting configurations to the **back** of the queue.

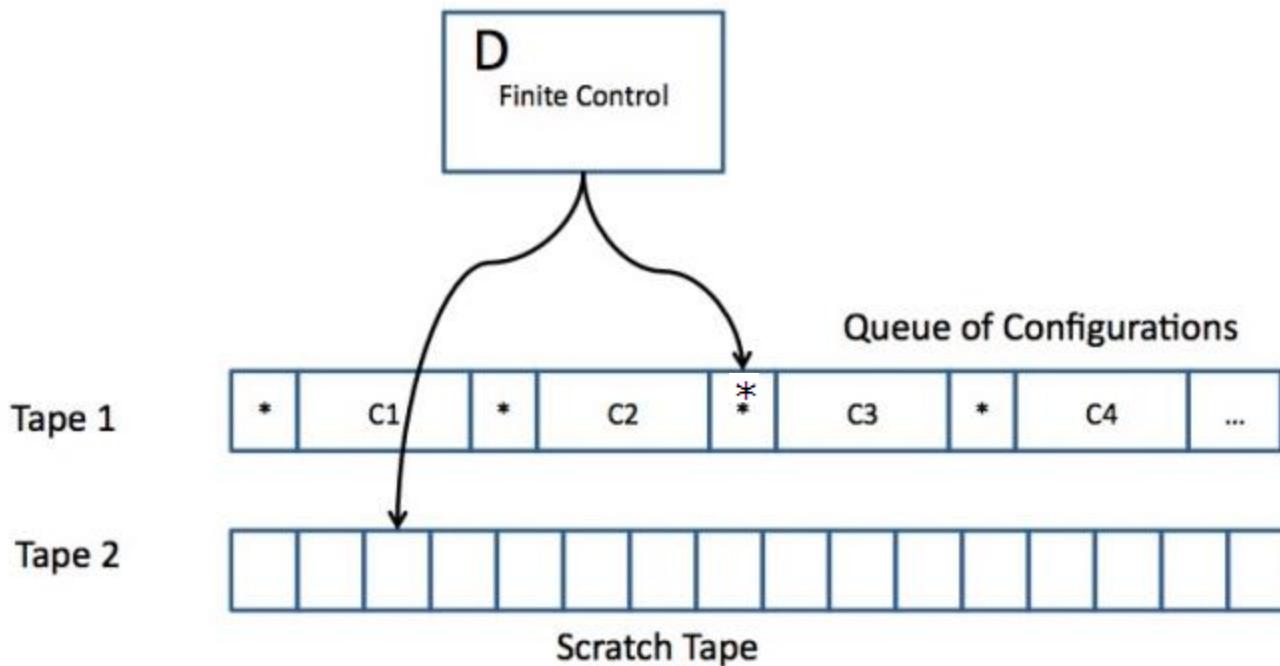
STRUCTURE OF THE SIMULATING DTM

- N is simulated with 2-tape DTM, D



STRUCTURE OF THE SIMULATING DTM

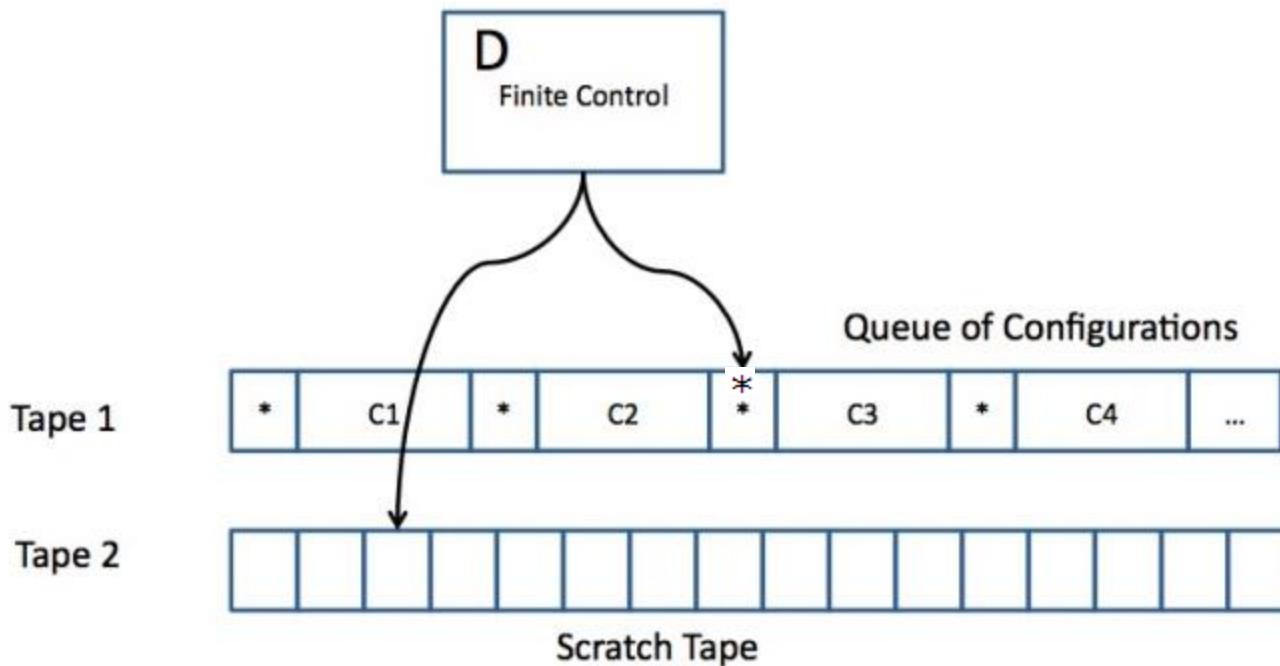
- N is simulated with 2-tape DTM, D



- Built into the finite control of D is the knowledge of what choices of moves N has for each state and input.

STRUCTURE OF THE SIMULATING DTM

- N is simulated with 2-tape DTM, D



- ➊ D examines the state and the input symbol of the current configuration (right after the dotted separator)
- ➋ If the state of the current configuration is the accept state of N , then D accepts the input and stops simulating N .

How D SIMULATES N

- Let m be the maximum number of choices N has for any of its states.
- Then, after n steps, N can reach at most $1 + m + m^2 + \dots + m^n$ configurations (which is at most nm^n)
- Thus D has to process at most this many configurations to simulate n steps of N .
- Thus the simulation can take **exponentially** more time than the nondeterministic TM.
- It is not known whether or not this exponential slowdown is necessary.

IMPLICATIONS

COROLLARY

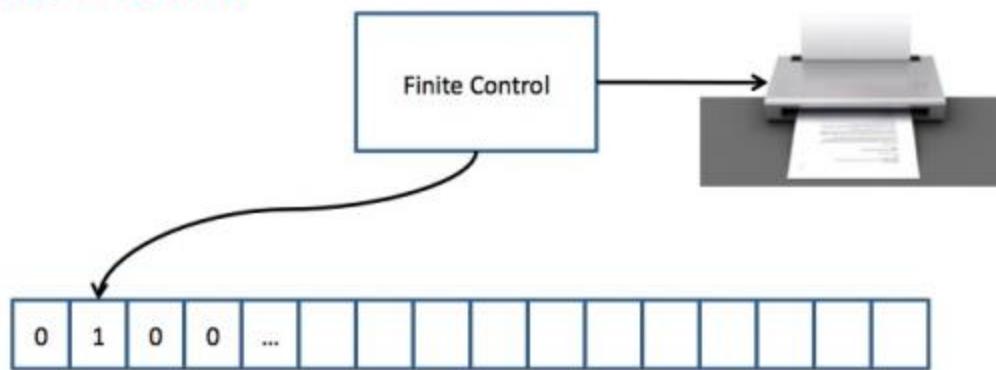
A language is Turing-recognizable if and only if some nondeterministic TM recognizes it.

COROLLARY

A language is decidable if and only if some nondeterministic TM decides it.

ENUMERATORS

- Remember we noted that some books used the term **recursively enumerable** for Turing-recognizable.
- This term arises from a variant of a TM called an **enumerator**.



- TM generates strings one by one.
- Everytime the TM wants to add a string to the list, it sends it to the printer.

ENUMERATORS

- The enumerator E starts with a blank input tape.
- If it does not halt, it may print an infinite list of strings.
- The strings can be enumerated in any order; repetitions are possible.
- The language of the enumerator is the collection of strings it eventually prints out.

- We can assume that the enumerator E writes one string at a time over a tape (it can use a tape symbol $\#$ to separate strings).

ENUMERATORS

THEOREM

A language is Turing recognizable if and only if some enumerator enumerates it.

PROOF.

The If-part: If an enumerator E enumerates the language A then a TM M recognizes A .

M = “On input w

- ① Run E . Everytime E outputs a string, compare it with w .
- ② If w ever appears in the output of E , accept.”

Clearly M accepts only those strings that appear on E 's list.



The TM M accepts w only when E produces w as one of its output strings.

ENUMERATORS

THEOREM

A language is Turing recognizable if and only if some enumerator enumerates it.

PROOF.

The Only-If-part: If a TM M recognizes a language A , we can construct the following enumerator for A .

- For each possible string $s \in \Sigma^*$ we can verify whether M accepts s , if so output s on to the tape .

Following attempt, does not work.

- Assume that there is an enumerator which enumerates Σ^* in a standard order.
- For each possible string $s \in \Sigma^*$ we can verify whether M accepts s , if so output s on to the tape .
- **The problem with this is**, for some s the TM M can enter in to an infinite loop and never returns.
 - There may be other strings which are accepted by M but will never gets a chance to be verified (and thus never is outputted).

- A feasible way of doing this (without falling in to an infinite loop) is given in the next slide.
- Basic idea:
- For example if there are two strings w_1 and w_2 and one of them makes the TM to loop infinitely. Do the following.
 1. $k = 1$
 2. Run TM for k steps on w_1 and if accept occurs then output “accept” and stop.
 3. Run TM for k steps on w_2 and if accept occurs then output “accept” and stop.
 4. $k++;$ goto step 2.

ENUMERATORS

THEOREM

A language is Turing recognizable if and only if some enumerator enumerates it.

PROOF.

The Only-If-part: If a TM M recognizes a language A , we can construct the following enumerator for A . Assume s_1, s_2, s_3, \dots is a list of possible strings in Σ^* .

E = “Ignore the input

- ① Repeat the following for $i = 1, 2, 3, \dots$
- ② Run M for i steps on each input $s_1, s_2, s_3, \dots, s_i$.
- ③ If any computations accept, print out corresponding s_j .”

If M accepts a particular string, it will appear on the list generated by E (in fact infinitely many times)

THE DEFINITION OF ALGORITHM - HISTORY

- in 1900, Hilbert posed the following problem:

“Given a polynomial of several variables with integer coefficients, does it have an integer root – an assignment of integers to variables, that make the polynomial evaluate to 0”

- For example, $6x^3yz^2 + 3xy^2 - x^3 - 10$ has a root at $x = 5, y = 3, z = 0$.
- Hilbert explicitly asked that an algorithm/procedure to be “devised”. He assumed it existed; somebody needed to find it!
- 70 years later it was shown that no algorithm exists.
- The intuitive notion of an algorithm may be adequate for giving algorithms for certain tasks, but was useless for showing no algorithm exists for a particular task.

This is known as Hilbert's Tenth Problem.

THE DEFINITION OF ALGORITHM - HISTORY

- In early 20th century, there was no formal definition of an algorithm.
- In 1936, Alonzo Church and Alan Turing came up with formalisms to define algorithms. These were shown to be equivalent, leading to the

CHURCH-TURING THESIS

Intuitive notion of algorithms ≡ Turing Machine Algorithms

THE DEFINITION OF AN ALGORITHM

- Let $D = \{p \mid p \text{ is a polynomial with integral roots}\}$
- Hilbert's 10th problem in TM terminology is "Is D decidable?" (No!)
- However D is Turing-recognizable!
- Consider a simpler version
$$D_1 = \{p \mid p \text{ is a polynomial over } x \text{ with integral roots}\}$$
- M_1 = "The input is polynomial p over x .
 - ① Evaluate p with x successively set to 0, 1, -1, 2, -2, 3, -3,
 - ② If at any point, p evaluates to 0, accept."
- D_1 is actually decidable since only a finite number of x values need to be tested (math!)
- D is also recognizable: just try systematically all integer combinations for all variables.

- For D_1 (polynomial of single variable) there is a bound and we can abandon the search beyond that and declare “No”.
- For D such a bound cannot exist (**proof is given by Matijasevich (1971)**).
 - So, if answer is “No” we enter in to an infinite loop.

In 1971, Yuri Matijasevich gave a resounding negative answer to Hilbert's tenth problem.

- This is called undecidability.
- Hilbert's tenth problem is undecidable.
- We will see the theory behind this in the next
...

Decidability

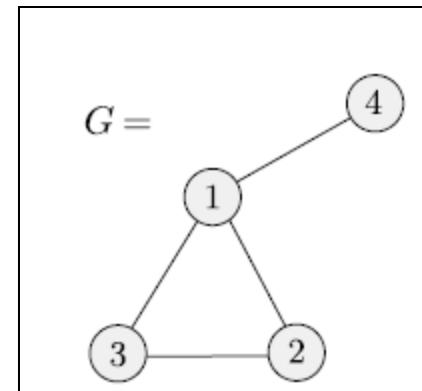
Theory behind existence of undecidable problems – Halting Problem --

Ref: <https://www.andrew.cmu.edu/user/ko/pdfs/lecture-15.pdf>

DESCRIBING TURING MACHINES AND THEIR INPUTS

- For the rest of the course we will have a rather standard way of describing TMs and their inputs.
- The input to TMs have to be strings.
- Every object O that enters a computation will be represented with an string $\langle O \rangle$, encoding the object.
- For example if G is a 4 node undirected graph with 4 edges $\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$
- Then we can define problems over graphs, e.g., as:

$$A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}$$



DESCRIBING TURING MACHINES AND THEIR INPUTS

- A TM for this problem can be given as:
- M = “On input $\langle G \rangle$, the encoding of a graph G :
 - ① Select the first node of G and mark it.
 - ② Repeat 3) until no new nodes are marked
 - ③ For each node in G , mark it, if there is edge attaching it to an already marked node.
 - ④ Scan all the nodes in G . If all are marked, the *accept*, else *reject*”

DECIDABILITY

- We investigate the power of algorithms to solve problems.
- We discuss certain problems that can be solved algorithmically and others that can not be.
- Why discuss **unsolvability**?
- Knowing a problem is unsolvable is useful because
 - you realize it must be simplified or altered before you find an algorithmic solution.
 - you gain a better perspective on computation and its limitations.

OVERVIEW

- Decidable Languages
- Diagonalization
- Halting Problem as a undecidable problem
- Turing-unrecognizable languages.

DECIDABLE LANGUAGES

SOME NOTATIONAL DETAILS

- $\langle B \rangle$ represents the encoding of the description of an automaton (DFA/NFA).
- We need to encode Q, Σ, δ and F .

DECIDABLE LANGUAGES

SOME NOTATIONAL DETAILS

- $\langle B \rangle$ represents the encoding of the description of an automaton (DFA/NFA).
- We need to encode Q, Σ, δ and F .
- Important thing to understand is that a machine (computing machine) can be represented as a string in some language.
 - It will be a string over some alphabet.
 - This is, in some sense, equivalent to say that **program is also data**.

Representation Vs. Real Thing

- DNA represents a living thing (like a human-being).
- Some people believe (!) that this is a complete description of a human being.
- You can know his personality, you can even know what he will do in future?
 - Some Hollywood movies captured this idea.

- A representation in itself is lifeless.
 - A program in itself cannot process the data.
 - It has to be realized through a processing unit or DNA has to be realized through a laboratory test tube or so.
-
- But the processing unit, now can be entirely independent of the program.
 - It can execute any program.

- We can give <program, data> to a general purpose computer which runs the program over the data and gives the output.

- A TM M also can be represented as a string.
- Call this string $\langle M \rangle$.
- $\langle M \rangle$ is like a program.
- Working of TM M on a string w can be realized by a *general purpose computer* like machine.
- This machine which can take input $\langle M, w \rangle$ and outputs what the TM M does with w , is called the **Universal Turing Machine**.

ENCODING FINITE AUTOMATA AS STRINGS

- Here is one possible encoding scheme:
- Encode Q using unary encoding:
 - For $Q = \{q_0, q_1, \dots, q_{n-1}\}$, encode q_i using $i + 1$ 0's, i.e., using the string 0^{i+1} .
 - We assume that q_0 is always the start state.
- Encode Σ using unary encoding:
 - For $\Sigma = \{a_1, a_2, \dots, a_m\}$, encode a_i using i 0's, i.e., using the string 0^i .
- With these conventions, all we need to encode is δ and F !
- Each entry of δ , e.g., $\delta(q_i, a_j) = q_k$ is encoded as

$$\underbrace{0^{i+1}}_{q_i} 1 \underbrace{0^j}_{a_j} 1 \underbrace{0^{k+1}}_{q_k}$$

ENCODING FINITE AUTOMATA AS STRINGS

- The whole δ can now be encoded as

$$\underbrace{00100001000}_\text{transition_1} 1 \underbrace{000001001000000}_\text{transition_2} \dots 1 \underbrace{000000100000010}_\text{transition_t}$$

- F can be encoded just as a list of the encodings of all the final states. For example, if states 2 and 4 are the final states, F could be encoded as

$$\underbrace{000}_\text{q_2} 1 \underbrace{00000}_\text{q_4}$$

- The whole DFA would be encoded by

$$11 \underbrace{001000100001000000 \dots 0}_\text{encoding of the transitions} 11 \underbrace{0000000010000000}_\text{encoding of the final states} 11$$

ENCODING FINITE AUTOMATA AS STRINGS

- $\langle B \rangle$ representing the encoding of the description of an automaton (DFA/NFA) would be something like

$$\langle B \rangle = 11 \underbrace{00100010000100000 \dots 0}_{\text{encoding of the transitions}} 11 \underbrace{0000000010000000}_{\text{encoding of the final states}} 11$$

- In fact, the description of all DFAs could be described by a regular expression like

$$11(0^+10^+10^+1)^*1(0^+1)^+1$$

- Similarly strings over Σ can be encoded with (the same convention)

$$\langle w \rangle = \underbrace{0000}_{a_4} 1 \underbrace{000000}_{a_6} 1 \dots \underbrace{0}_{a_1}$$

ENCODING FINITE AUTOMATA AS STRINGS

- $\langle B, w \rangle$ represents the encoding of a machine followed by an input string, in the manner above (with a suitable separator between $\langle B \rangle$ and $\langle w \rangle$).
- Now we can describe our problems over languages and automata as problems over strings (representing automata and languages).

DECIDABLE PROBLEMS

REGULAR LANGUAGES

- Does B accept w ?
- Is $L(B)$ empty?
- Is $L(A) = L(B)$?

THE ACCEPTANCE PROBLEM FOR DFAS

THEOREM 4.1

$A_{DFA} = \{\langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w\}$ is a decidable language.

PROOF

- Simulate with a two-tape TM.
 - One tape has $\langle B, w \rangle$
 - The other tape is a work tape that keeps track of which state of B the simulation is in.
- $M = \text{"On input } \langle B, w \rangle$
 - ① Simulate B on input w
 - ② If the simulation ends in an accept state of B , *accept*; if it ends in a nonaccepting state, *reject*.

THE ACCEPTANCE PROBLEM FOR NFAs

THEOREM 4.2

$A_{NFA} = \{\langle B, w \rangle \mid B \text{ is a NFA that accepts input string } w\}$ is a decidable language.

PROOF

- Convert NFA to DFA and use Theorem 4.1
- $N = \text{"On input } \langle B, w \rangle \text{ where } B \text{ is an NFA}$
 - ① Convert NFA B to an equivalent DFA C , using the determinization procedure.
 - ② Run TM M in Thm 4.1 on input $\langle C, w \rangle$
 - ③ If M accepts *accept*; otherwise *reject*.

THE GENERATION PROBLEM FOR REGULAR EXPRESSIONS

THEOREM 4.3

$A_{REX} = \{\langle R, w \rangle \mid R \text{ is a regular exp. that generates string } w\}$ is a decidable language.

PROOF

- Note R is already a string!!
- Convert R to an NFA and use Theorem 4.2
- P = “On input $\langle R, w \rangle$ where R is a regular expression
 - ① Convert R to an equivalent NFA A , using the Regular Expression-to-NFA procedure
 - ② Run TM N in Thm 4.2 on input $\langle A, w \rangle$
 - ③ If N accepts accept; otherwise reject.”

THE EMPTINESS PROBLEM FOR DFAs

THEOREM 4.4

$E_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$ is a decidable language.

PROOF

- Use the DFS algorithm to mark the states of DFA
- $T = \text{"On input } \langle A \rangle \text{ where } A \text{ is a DFA.}$
 - ① Mark the start state of A
 - ② Repeat until no new states get marked.
 - Mark any state that has a transition coming into it from any state already marked.
 - ③ If no final state is marked, *accept*; otherwise *reject*.

- Is the following a decidable language?

$$EQ_{DFA} = \{\langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

THE EQUIVALENCE PROBLEM FOR DFAS

THEOREM 4.5

$EQ_{DFA} = \{\langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$ is a decidable language.

PROOF

- Construct the machine for $L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$ and check if $L(C) = \emptyset$.
- T = “On input $\langle A, B \rangle$ where A and B are DFAs.
 - ① Construct the DFA for $L(C)$ as described above.
 - ② Run TM T of Theorem 4.4 on input $\langle C \rangle$.
 - ③ If T accepts, *accept*; otherwise *reject*.”

DECIDABLE PROBLEMS

CONTEXT-FREE LANGUAGES

- Does grammar G generate w ?
- Is $L(G)$ empty?

- Is the following language decidable?

$$A_{CFG} = \{\langle G, w \rangle \mid G \text{ is a CFG that generates string } w\}$$

THE GENERATION PROBLEM FOR CFGs

THEOREM 4.7

$A_{CFG} = \{\langle G, w \rangle \mid G \text{ is a CFG that generates string } w\}$ is a decidable language.

PROOF

- Convert G to Chomsky Normal Form and use the CYK algorithm.
- $C = \text{"On input } \langle G, w \rangle \text{ where } G \text{ is a CFG}$
 - ① Convert G to an equivalent grammar in CNF
 - ② Run CYK algorithm on w of length n
 - ③ If $S \in V_{i,n}$ accept; otherwise reject."

THE GENERATION PROBLEM FOR CFGs

ALTERNATIVE PROOF

- Convert G to Chomsky Normal Form and check all derivations up to a certain length (Why!)
- $S =$ “On input $\langle G, w \rangle$ where G is a CFG
 - ① Convert G to an equivalent grammar in CNF
 - ② List all derivations with $2n - 1$ steps where n is the length of w . If $n = 0$ list all derivations of length 1.
 - ③ If any of these strings generated is equal to w , *accept*; otherwise *reject*.”
- This works because every derivation using a CFG in CNF either increase the length of the sentential form by 1 (using a rule like $A \rightarrow BC$) or leaves it the same (using a rule like $A \rightarrow a$)
- Obviously this is not very efficient as there may be exponentially many strings of length up to $2n - 1$.

THE EMPTINESS PROBLEM FOR CFGS

THEOREM 4.8

$E_{CFG} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \Phi\}$ is a decidable language.

PROOF

- Mark variables of G systematically if they can generate terminal strings, and check if S is unmarked.
- $R = \text{"On input } \langle G \rangle \text{ where } G \text{ is a CFG.}$
 - ① Mark all terminal symbols G
 - ② Repeat until no new variable get marked.
 - Mark any variable A such that G has a rule $A \rightarrow U_1 U_2 \dots U_k$ and U_1, U_2, \dots, U_k are already marked.
 - ③ If start symbol is NOT marked, accept; otherwise reject."

THE EQUIVALENCE PROBLEM FOR CFGs

$$EQ_{CFG} = \{\langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$$

- Is this decidable?

THE EQUIVALENCE PROBLEM FOR CFGS

$$EQ_{CFG} = \{\langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$$

- It turns out that EQ_{DFA} is not a decidable language.
- The construction for DFAs does not work because CFLs are NOT closed under intersection and complementation.
- Proof comes later.

DECIDABILITY OF CFLS

THEOREM 4.9

Every context free language is decidable.

PROOF

- Design a TM M_G that has G built into it and use the result of A_{CFG} .
- M_G = “On input w
 - ① Run TM S (from Theorem 4.7) on input $\langle G, w \rangle$
 - ② If S accepts, accept, otherwise reject.

ACCEPTANCE PROBLEM FOR TMs

THEOREM 4.11

$A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ is undecidable.

- Note that A_{TM} is Turing-recognizable. Thus this theorem when proved, shows that recognizers are more powerful than deciders.
- We can encode TMs with strings just like we did for DFA's (How?)

- We shall assume the states are q_1, q_2, \dots, q_r for some r . The start state will always be q_1 , and q_2 will be the only accepting state. Note that, since we may assume the TM halts whenever it enters an accepting state, there is never any need for more than one accepting state.
- We shall assume the tape symbols are X_1, X_2, \dots, X_s for some s . X_1 always will be the symbol 0, X_2 will be 1, and X_3 will be B , the blank. However, other tape symbols can be assigned to the remaining integers arbitrarily.
- We shall refer to direction L as D_1 and direction R as D_2 .

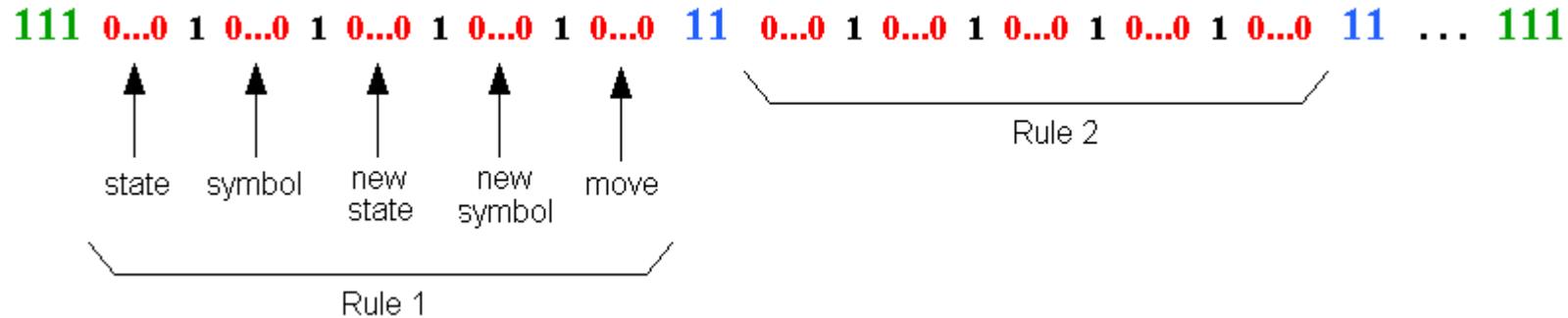
- We shall assume the states are q_1, q_2, \dots, q_r for some r . The start state will always be q_1 , and q_2 will be the only accepting state. Note that, since we may assume the TM halts whenever it enters an accepting state, there is never any need for more than one accepting state.
- We shall assume the tape symbols are X_1, X_2, \dots, X_s for some s . X_1 always will be the symbol 0, X_2 will be 1, and X_3 will be B , the blank. However, other tape symbols can be assigned to the remaining integers arbitrarily.
- We shall refer to direction L as D_1 and direction R as D_2 .

Suppose one transition rule

is $\delta(q_i, X_j) = (q_k, X_l, D_m)$, for some integers i, j, k, l , and m .

We shall code this rule by the string $0^i 1 0^j 1 0^k 1 0^l 1 0^m$.

Encoding of a TM



Three tapes



Tape 1



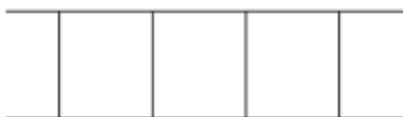
Description of M

Tape 2



Tape Contents of M

Tape 3



States of M

- Turing Machine M can be encoded as a string
 $\langle M \rangle$
- These encodings, of various Turing Machines, can be seen as a language.
- Language of TMs = { $\langle M \rangle$ | $\langle M \rangle$ is an encoding of a TM M }.
- Strings of the above language can be lexicographically ordered.
- As per the above ordering, one can say the 1st TM, the 2nd TM, so on.
 - Same machine may be encoded in multiple ways
 - 2nd TM may be same as 10032nd TM

ACCEPTANCE PROBLEM FOR TMs

- The TM U recognizes A_{TM}
- U = “On input $\langle M, w \rangle$ where M is a TM and w is a string:
 - ① Simulate M on w
 - ② If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*.
- Note that if M loops on w , then U loops on $\langle M, w \rangle$, which is why it is NOT a decider!
- U can not detect that M halts on w .
- A_{TM} is also known as the **Halting Problem**
- U is known as the **Universal Turing Machine** because it can simulate every TM (including itself!)

THE DIAGONALIZATION METHOD

SOME BASIC DEFINITIONS

- Let A and B be any two sets (not necessarily finite) and f be a function from A to B .
- f is **one-to-one** if $f(a) \neq f(b)$ whenever $a \neq b$.
- f is **onto** if for every $b \in B$ there is an $a \in A$ such that $f(a) = b$.
- We say A and B are the **same size** if there is a one-to-one and onto function $f : A \rightarrow B$.
- Such a function is called a **correspondence** for pairing A and B .
 - Every element of A maps to a unique element of B
 - Each element of B has a unique element of A mapping to it.

THE DIAGONALIZATION METHOD

- Let \mathcal{N} be the set of natural numbers $\{1, 2, \dots\}$ and let \mathcal{E} be the set of even numbers $\{2, 4, \dots\}$.
- $f(n) = 2n$ is a correspondence between \mathcal{N} and \mathcal{E} .
- Hence, \mathcal{N} and \mathcal{E} have the same size (though $\mathcal{E} \subset \mathcal{N}$).
- A set A is **countable** if it is either finite or has the same size as \mathcal{N} .
- $\mathcal{Q} = \{\frac{m}{n} \mid m, n \in \mathcal{N}\}$ is countable!
- \mathbb{Z} the set of integers is countable:

$$f(n) = \begin{cases} \frac{n}{2} & n \text{ even} \\ -\frac{n+1}{2} & n \text{ odd} \end{cases}$$

THE DIAGONALIZATION METHOD

THEOREM

\mathcal{R} is uncountable

PROOF.

- Assume f exists and every number in \mathcal{R} is listed.
- Assume $x \in \mathcal{R}$ is a real number such that x differs from the j^{th} number in the j^{th} decimal digit.
- If x is listed at some position k , then it differs from itself at k^{th} position; otherwise the premise does not hold
- f does not exist

n	$f(n)$
1	3.14159...
2	55.77777...
3	0.12345...
4	0.50000...
:	:
$x = .4527\dots$	

defined as such, can not be on this list.

DIAGONALIZATION OVER LANGUAGES

COROLLARY

Some languages are not Turing-recognizable.

PROOF

- For any alphabet Σ , Σ^* is countable. Order strings in Σ^* by length and then alphanumerically, so $\Sigma^* = \{s_1, s_2, \dots, s_i, \dots\}$
- The set of all TMs is a countable language.
 - Each TM M corresponds to a string $\langle M \rangle$.
 - Generate a list of strings and remove any strings that do not represent a TM to get a list of TMs.

DIAGONALIZATION OVER LANGUAGES

PROOF (CONTINUED)

- The set of infinite binary sequences, \mathcal{B} , is uncountable. (Exactly the same proof we gave for uncountability of \mathcal{R})
- Let \mathcal{L} be the set of all languages over Σ .
- For each language $A \in \mathcal{L}$ there is unique infinite binary sequence χ_A
 - The i^{th} bit in χ_A is 1 if $s_i \in A$, 0 otherwise.

$$\Sigma^* = \{ \epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots \}$$

$$A = \{ 0, 00, 01, 000, 001, \dots \}$$

$$\chi_A = \{ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ \dots \}$$

DIAGONALIZATION OVER LANGUAGES

PROOF (CONTINUED)

- The function $f : \mathcal{L} \longrightarrow \mathcal{B}$ is a correspondence. Thus \mathcal{L} is uncountable.
- So, there are languages that can not be recognized by some TM.
There are not enough TMs to go around.

THE HALTING PROBLEM IS UNDECIDABLE

THEOREM

$A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$, is undecidable.

PROOF

- We assume A_{TM} is decidable and obtain a contradiction.
- Suppose H decides A_{TM}

$$H(\langle M, w \rangle) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

THE HALTING PROBLEM IS UNDECIDABLE

PROOF (CONTINUED)

- We now construct a new TM D

D = “On input $\langle M \rangle$, where M is a TM

- ① Run H on input $\langle M, \langle M \rangle \rangle$.
- ② If H accepts, *reject*, if H rejects, *accept*”

- So

$$D(\langle M \rangle) = \begin{cases} \text{accept} & \text{if } M \text{ does not accept } \langle M \rangle \\ \text{reject} & \text{if } M \text{ accepts } \langle M \rangle \end{cases}$$

- When D runs on itself we get

$$D(\langle D \rangle) = \begin{cases} \text{accept} & \text{if } D \text{ does not accept } \langle D \rangle \\ \text{reject} & \text{if } D \text{ accepts } \langle D \rangle \end{cases}$$

- Neither D nor H can exist.

WHAT HAPPENED TO DIAGONALIZATION?

Consider the behaviour of all possible deciders:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$
M_1	<u>accept</u>	reject	accept	reject
M_2	<u>accept</u>	<u>accept</u>	accept	accept
M_3	reject	<u>reject</u>	<u>reject</u>	reject
M_4	accept	accept	<u>reject</u>	<u>reject</u>
:		:		
:		:		

WHAT HAPPENED TO DIAGONALIZATION?

Consider the behaviour of all possible deciders:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$
M_1	<u>accept</u>	reject	accept	reject
M_2	accept	<u>accept</u>	accept	accept
M_3	reject	<u>reject</u>	<u>reject</u>	reject
M_4	accept	accept	<u>reject</u>	<u>reject</u>
:		:		
:			:	

- D computes the opposite of the diagonal entries!

WHAT HAPPENED TO DIAGONALIZATION?

Consider the behaviour of all possible deciders:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$...	$\langle D \rangle$...
M_1	<u>accept</u>	reject	accept	reject	...	accept	...
M_2	accept	<u>accept</u>	accept	accept	...	accept	...
M_3	reject	<u>reject</u>	<u>reject</u>	reject	...	reject	...
M_4	accept	accept	<u>reject</u>	<u>reject</u>	...	accept	...
:		:		
$D = M_j$	reject	reject	accept	accept	...	?	...
:		:		

- D computes the opposite of the diagonal entries!

WHAT HAPPENED TO DIAGONALIZATION?

Consider the behaviour of all possible deciders:

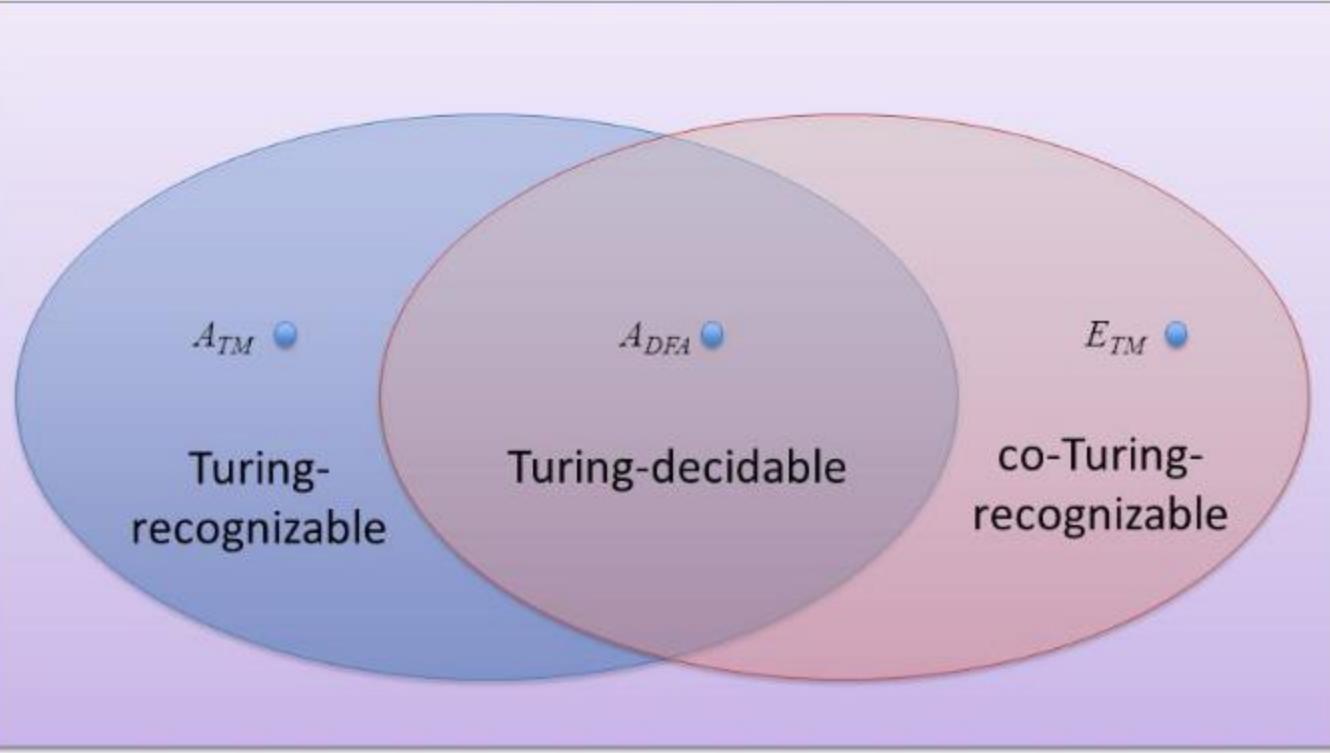
	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$...	$\langle D \rangle$...
M_1	accept	reject	accept	reject	...	accept	...
M_2	accept	accept	accept	accept	...	accept	...
M_3	reject	reject	reject	reject	...	reject	...
M_4	accept	accept	reject	reject	...	accept	...
:		:		
$D = M_j$	reject	reject	accept	accept	...	?	...
:		:		

- D computes the opposite of the diagonal entries!

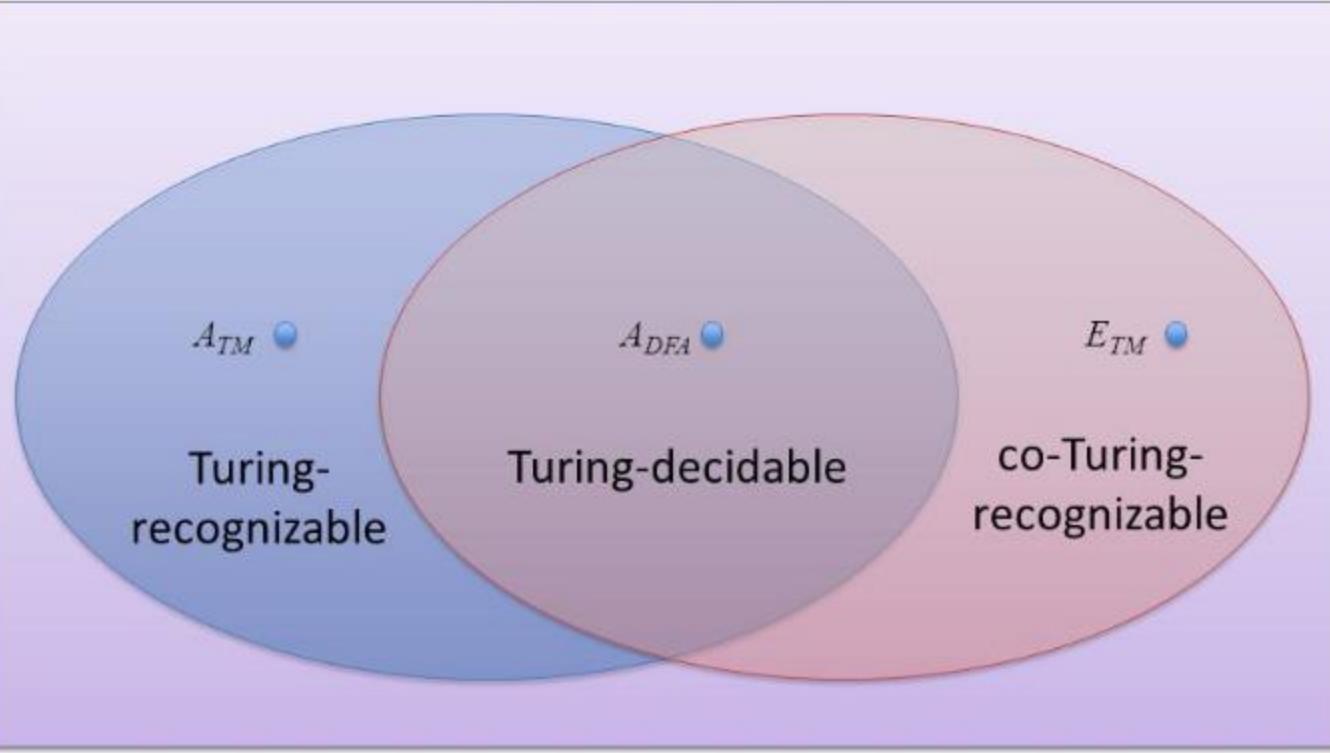
D cannot exist; After all D used only H; So if H exists then D exists;
Hence H (decider for A_{TM}) cannot exist.

A TURING UNRECOGNIZABLE LANGUAGE

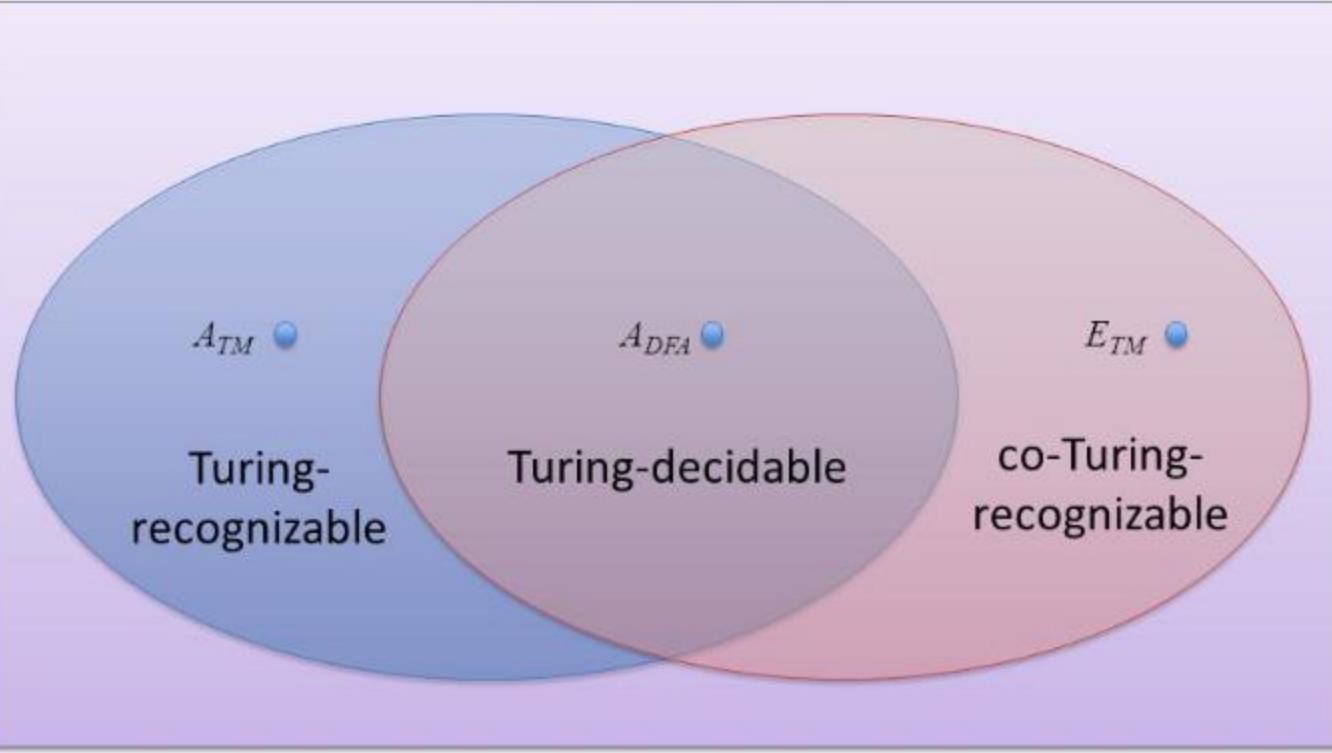
- A language is **co-Turing-recognizable** if it is the complement of a Turing-recognizable language.
- A language is **decidable** if it is Turing-recognizable and co-Turing-recognizable.
- $\overline{A_{TM}}$ is not Turing recognizable.
 - We know A_{TM} is Turing-recognizable.
 - If $\overline{A_{TM}}$ were also Turing-recognizable, A_{TM} would have to be decidable.
 - We know A_{TM} is not decidable.
 - $\overline{A_{TM}}$ must not be Turing-recognizable.



- Can you locate where $\overline{A_{TM}}$ will be?
- Can you locate $\overline{E_{TM}}$ will be?



- Can you locate where $\overline{A_{TM}}$ will be?
- Can you locate $\overline{E_{TM}}$ will be?



- Can you locate where $\overline{A_{TM}}$ will be? (**in co-RE**)
- Can you locate $\overline{E_{TM}}$ will be? (**in RE**)

Preview ...

- Are there languages which are neither RE nor co-RE ?? (i.e. both the language and its complement are not in RE).

Preview ...

- Are there languages which are neither RE nor co-RE ??
- Yes. $EQ_{TM} = \{\langle M_1, M_2 \rangle \mid L(M_1) = L(M_2)\}$.
- But, for this to understand we need the concept called “**mapping reducibility**” which is a binary relation between languages and is represented \leq_m

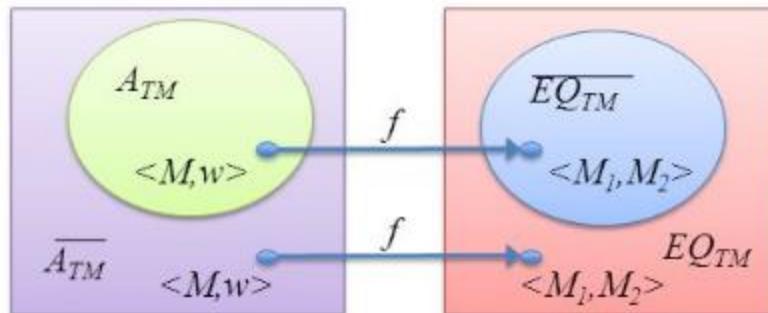
Preview ...

- Are there languages which are neither RE nor co-RE ??

EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable

- Proof:

Show $A_{TM} \leq_m \overline{EQ_{TM}}$

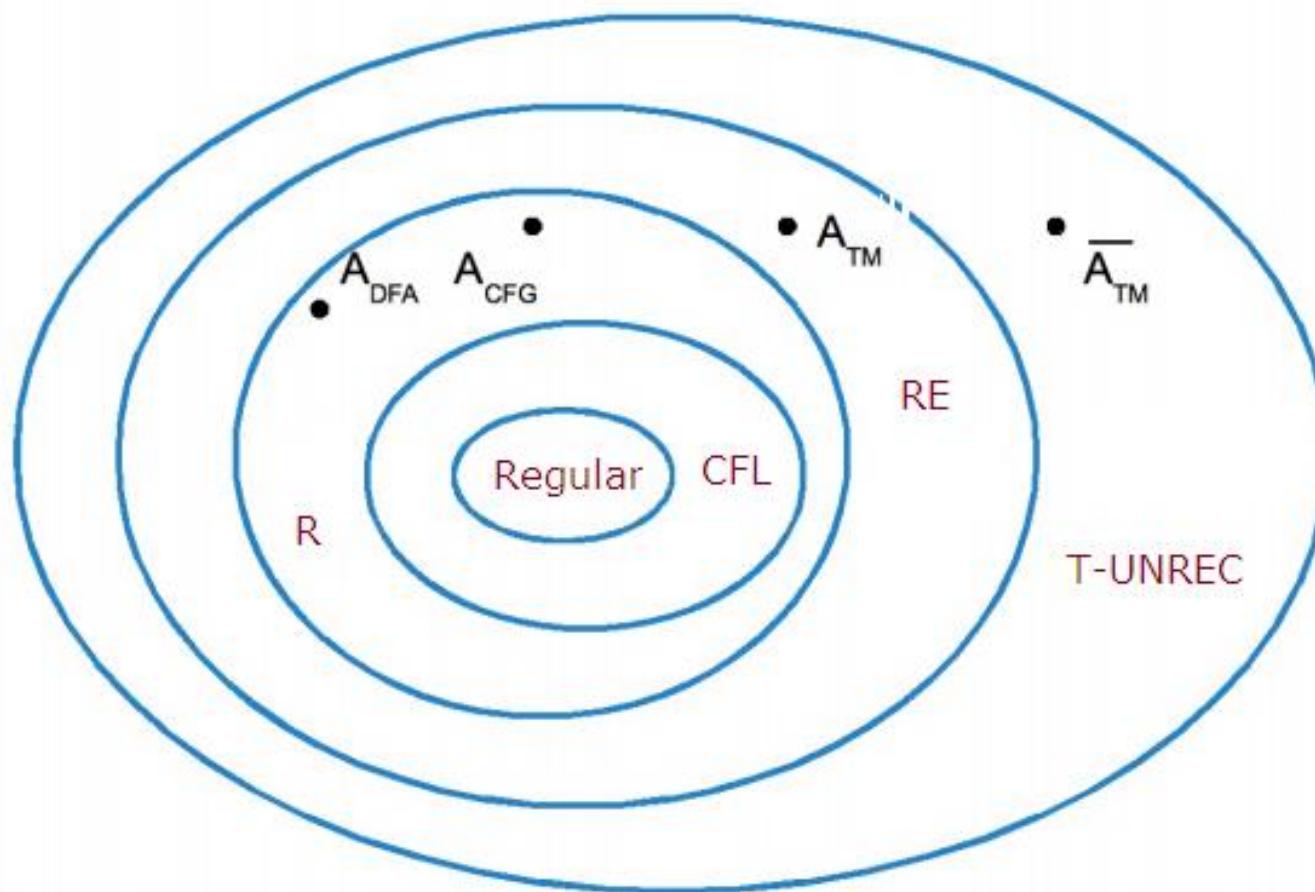


Reducibility

A way to show some languages are
undecidable !

<https://www.andrew.cmu.edu/user/ko/pdfs/lecture-16.pdf>

THE LANDSCAPE OF THE CHOMSKY HIERARCHY



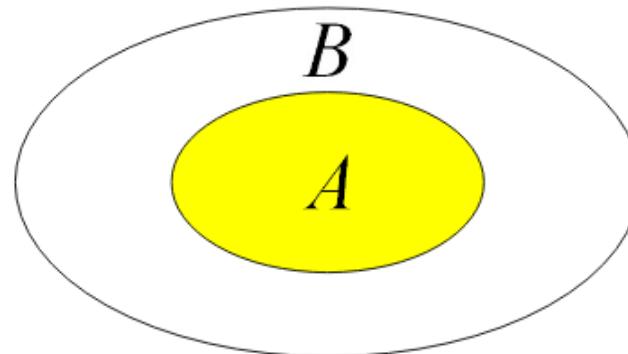
REDUCIBILITY

- A **reduction** is a way of converting one problem to another problem, so that the solution to the second problem can be used to solve the first problem.
 - Finding the area of a rectangle, reduces to measuring its width and height
 - Solving a set of linear equations, reduces to inverting a matrix.

Problem A is reduced to problem B



If we can solve problem B then
we can solve problem A .



A reduces to B

- $A \leq B$
- Find area of a rectangle \leq find length and find width of rectangle.
- Solving B means you know how to solve the fundamental ingredients (which are needed to solve A).
- A solution to B can be used to solve A.
- Note, a solution to A may not be enough to solve B.
 - Knowing area of a rectangular is not enough to find its length and width !!

A reduces to B : One more example

- $A \leq B$
- Doing injection to a patient \leq doing surgery to a patient
- Solving B means you know how to solve the fundamental ingredients (which are needed to solve A).
- A solution to B can be used to solve A.
- Note, a solution to A may not be enough to solve B.
 - Knowing area of a rectangular is not enough to find its length and width !!

A reduces to B

- $A \leq B$
- Solving B means you know how to solve the fundamental ingredients (which are needed to solve A).
- A solution to B can be used to solve A.
- An algorithm that solves B can be converted to an algorithm that solves A

A reduces to B

- $A \leq B$
- If B is decidable, then so is A.
- Contrapositive, if A is undecidable then so is B.

Problem A is reduced to problem B



If B is decidable then A is decidable.



If A is undecidable then B is undecidable.

PROVING UNDECIDABILITY VIA REDUCTIONS

THEOREM 5.1

$\text{HALT}_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$ is undecidable.

PROVING UNDECIDABILITY VIA REDUCTIONS

THEOREM 5.1

$\text{HALT}_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$ is undecidable.

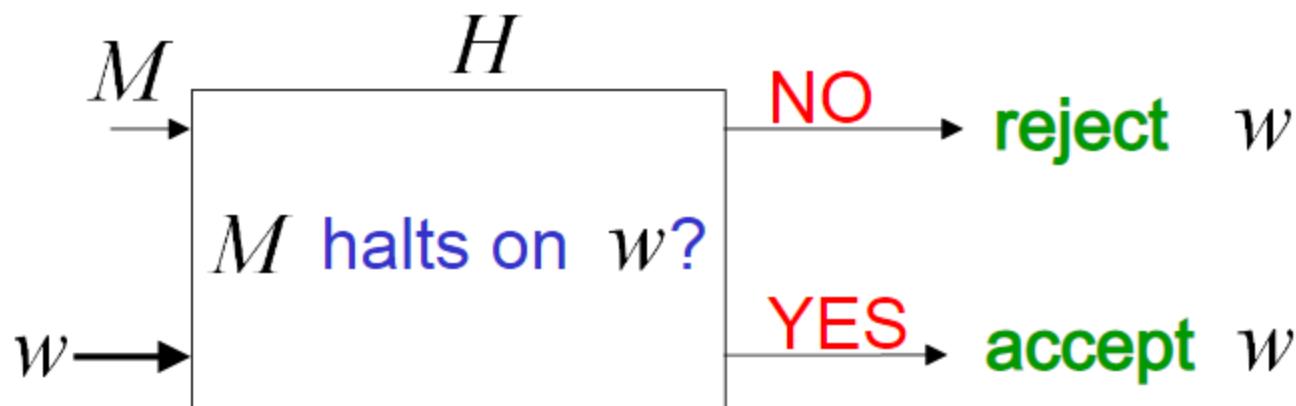
- We show that A_{TM} is reducible to HALT_{TM}
- Since A_{TM} is undecidable, so is HALT_{TM}

PROVING UNDECIDABILITY VIA REDUCTIONS

THEOREM 5.1

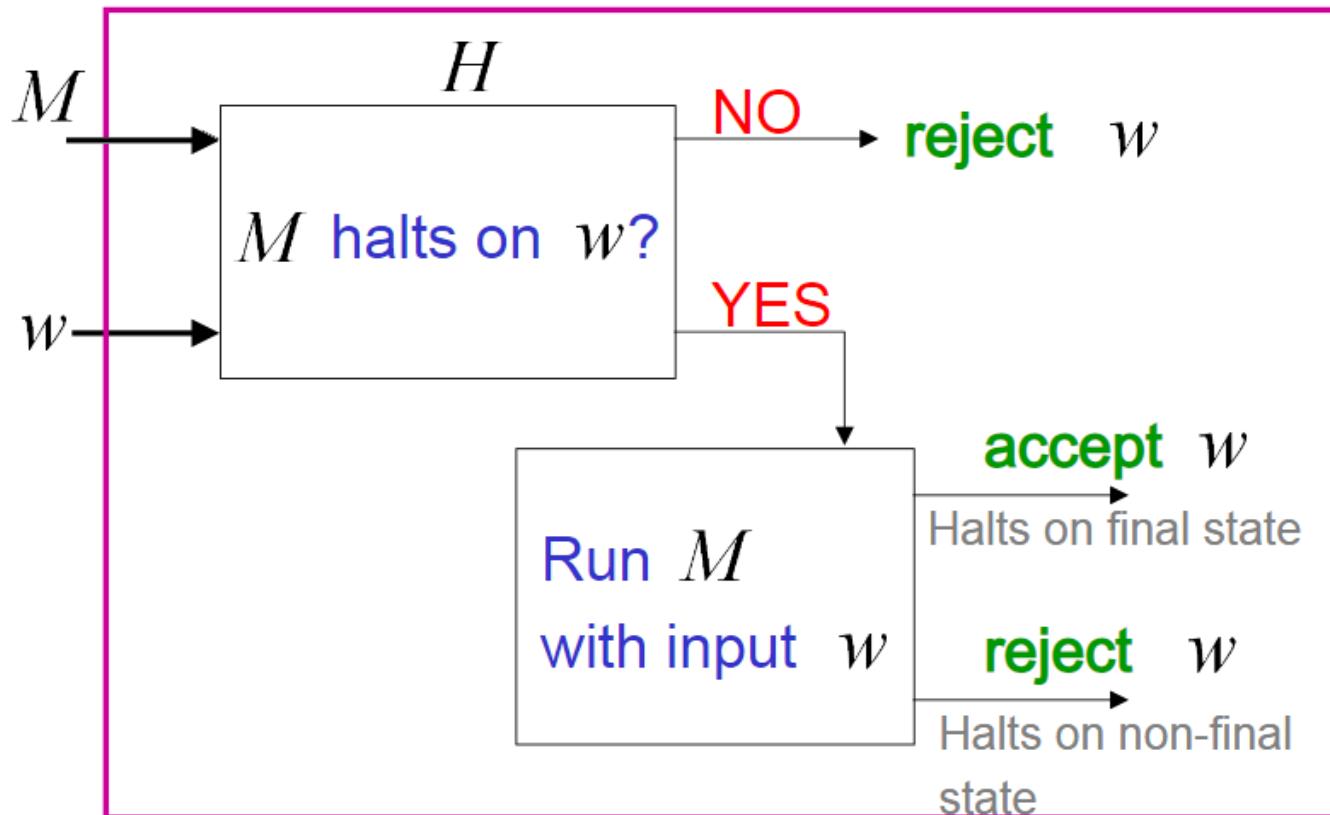
$\text{HALT}_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$ is undecidable.

- Suppose HALT_{TM} is decidable, this means

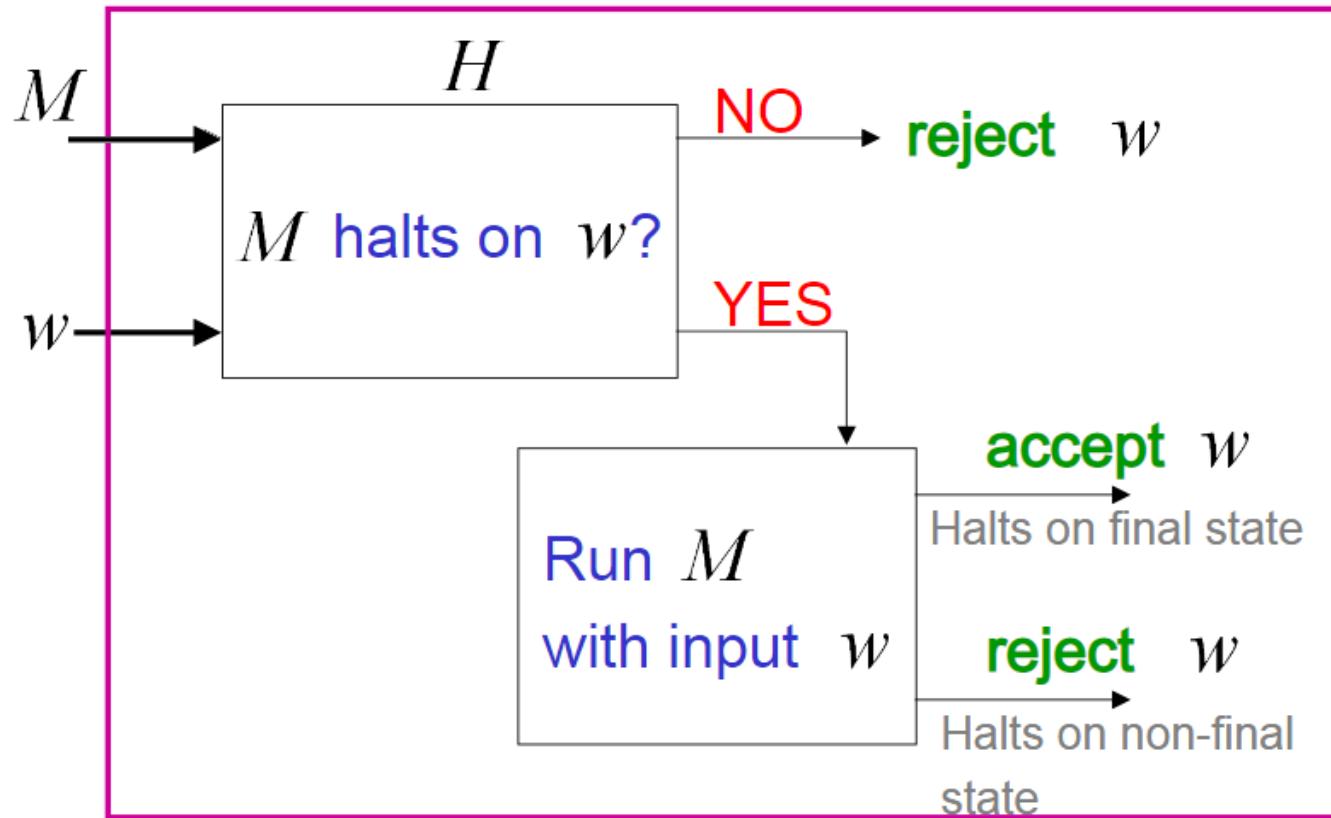


- Then A_{TM} is decidable.

- Then A_{TM} is decidable.



- Then A_{TM} is decidable.



- Contradiction

This diagram shows how A_{TM} can be reduced to $HALT_{TM}$

PROVING UNDECIDABILITY VIA REDUCTIONS

THEOREM 5.2

$E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Phi\}$ is undecidable.

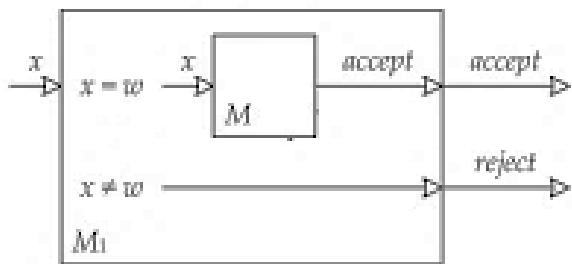
- A decider for A_{TM} via E_{TM} is possible.
- We are given $\langle M, w \rangle$, and asked to find whether $\langle M, w \rangle \in A_{TM}$?
- For this, First create M_1 (from the given $\langle M, w \rangle$)

PROVING UNDECIDABILITY VIA REDUCTIONS

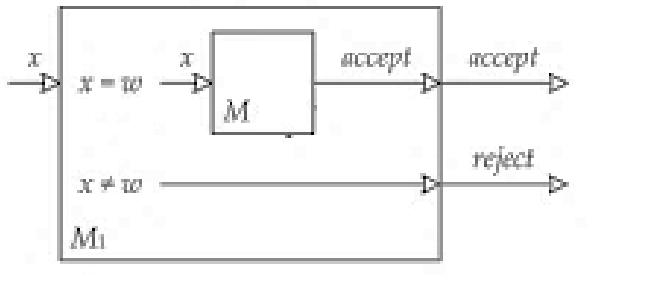
THEOREM 5.2

$E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Phi\}$ is undecidable.

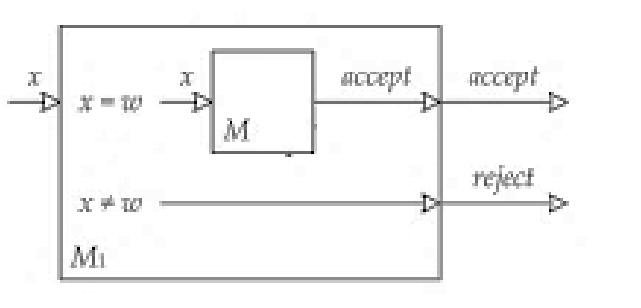
- A decider for A_{TM} via E_{TM} is possible.
- We are given $\langle M, w \rangle$, and asked to find whether $\langle M, w \rangle \in A_{TM}$?
- For this, First create M_1



Now, $L(M_1)$ is what?



- Now, $L(M_1)$ is what?



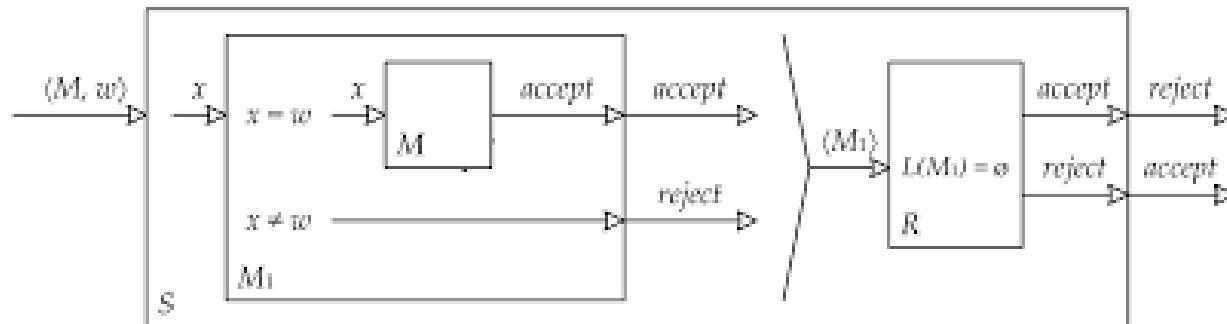
- Now, $L(M_1)$ is what?
- $L(M_1)$ is either $\{w\}$ or is \emptyset
- Now, if M_1 is in E_{TM}
 - This means, $L(M_1) = \emptyset$
 - This means, $\langle M, w \rangle \notin A_{TM}$
- Now, if M_1 is not in E_{TM}
 - This means, $L(M_1) \neq \emptyset$
 - This means, $\langle M, w \rangle \in A_{TM}$

PROVING UNDECIDABILITY VIA REDUCTIONS

THEOREM 5.2

$E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$ is undecidable.

- A decider for A_{TM} via E_{TM} is possible.



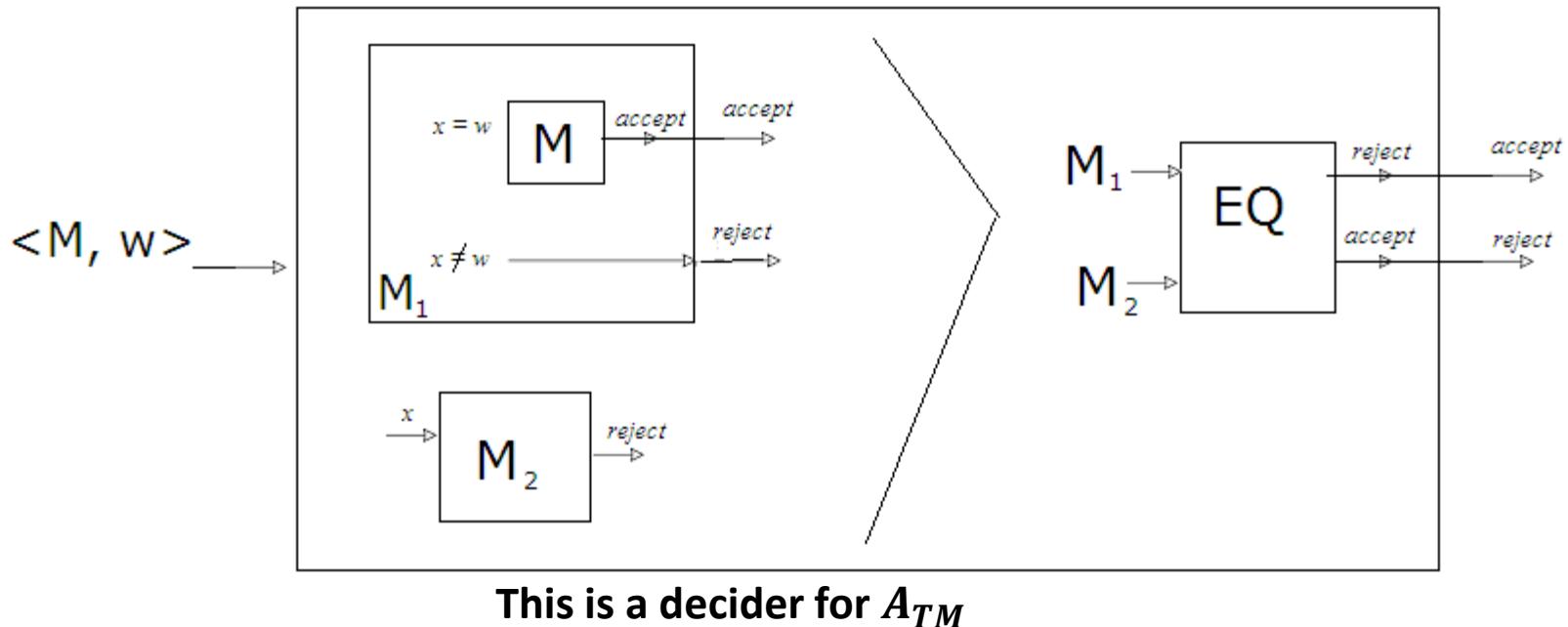
- That is, A_{TM} can be reduced to E_{TM} .
- Contradiction

TESTING FOR LANGUAGE EQUALITY

THEOREM 5.4

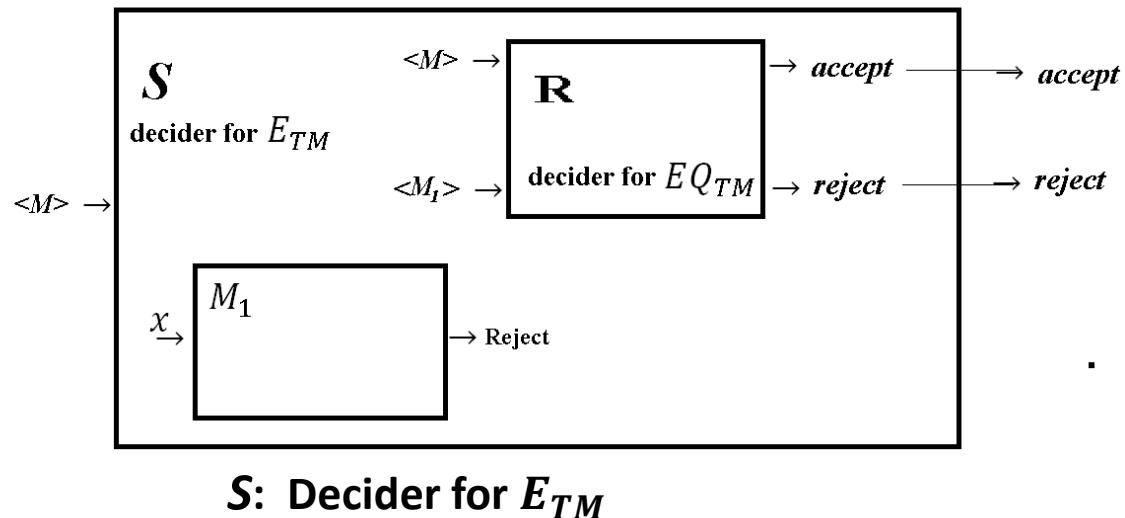
$EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ is undecidable.

- Decider for A_{TM} via EQ_{TM}
- Following is the reduction of A_{TM} to EQ_{TM}



Alternate way to show EQ_{TM} is undecidable.

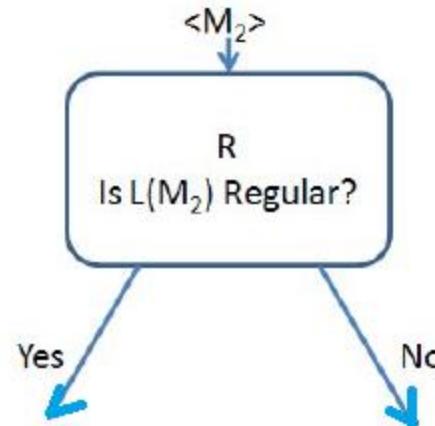
- Since, we know E_{TM} is undecidable,
- We can try to reduce E_{TM} to EQ_{TM}
- Let R be a decider for EQ_{TM}
- We can build a decider (call this S) for E_{TM} by using R



TESTING FOR REGULARITY

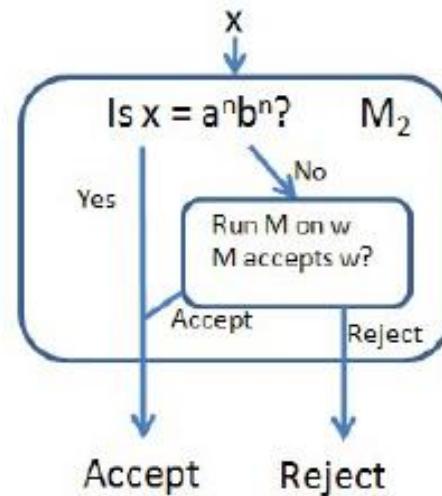
$REGULAR_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$ is undecidable.

- If $REGULAR_{TM}$ is decidable, then this can be used to decide A_{TM} .
- Let R be a decider for $REGULAR_{TM}$.



How R can be used to decide $\langle M, w \rangle \in A_{TM}$?

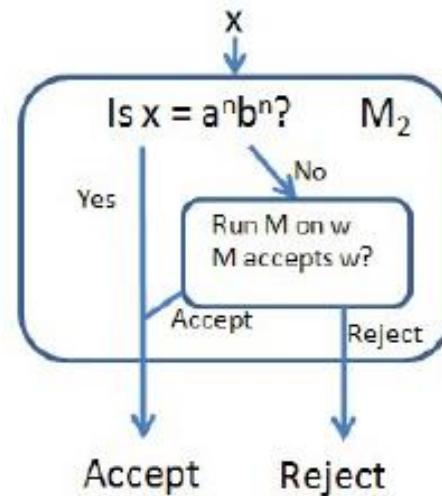
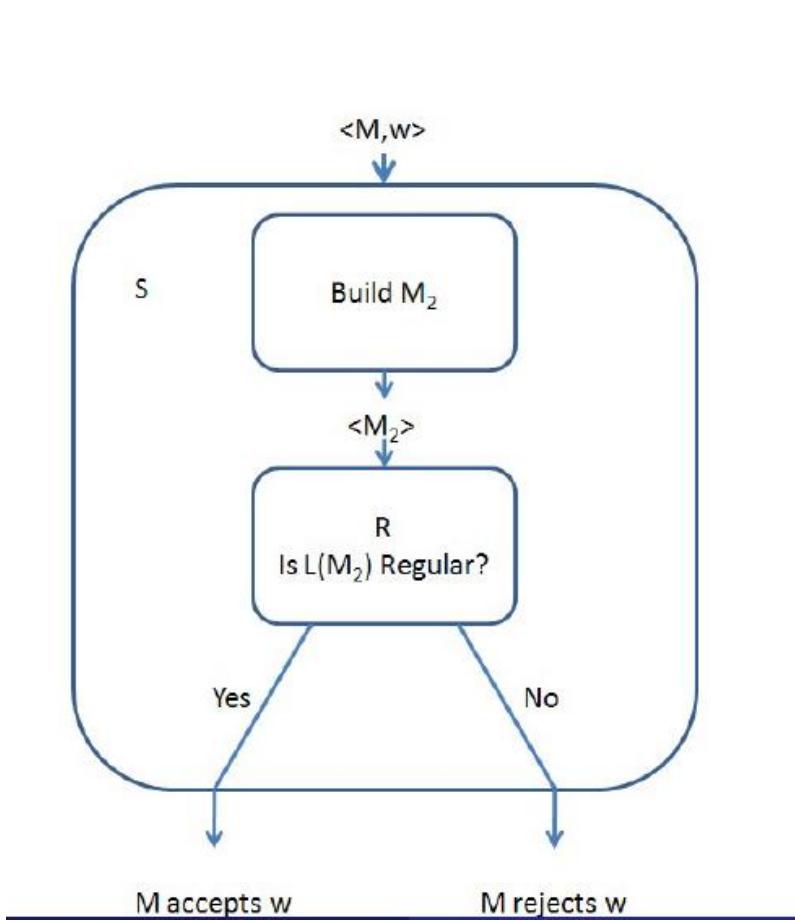
- Build M_2 as shown



So $L(M_2)$ is $= \Sigma^*$ if M accepts w
 $L(M_2)$ is $= \{a^n b^n\}$ otherwise

How R can be used to decide $\langle M, w \rangle \in A_{TM}$?

- Build M_2 as shown



So $L(M_2)$ is $= \Sigma^*$ if M accepts w
 $L(M_2)$ is $= \{a^n b^n\}$ otherwise

- Similar to $REGULAR_{TM}$, we can show CFL_{TM} is undecidable.
 - That is, finding whether a TM's language is CFL or not is undecidable.
 - In fact, we can extend this. TM's language is finite or not is undecidable.
 - General theorem in this regard is called ***The Rice's Theorem.***

More formal way of reductions

MAPPING REDUCTION

WE CAN GET MORE REFINED ANSWERS

Computable function

-

DEFINITION 5.17

A function $f: \Sigma^* \rightarrow \Sigma^*$ is a *computable function* if some Turing machine M , on every input w , halts with just $f(w)$ on its tape.

- For example,

we can make a machine that takes input $\langle m, n \rangle$ and returns $m + n$, the sum of m and n .

Mapping Reductions

Definition: Let A and B be two languages. We say that there is a **mapping reduction** from A to B , and denote

$$A \leq_m B$$

if there is a **computable function**

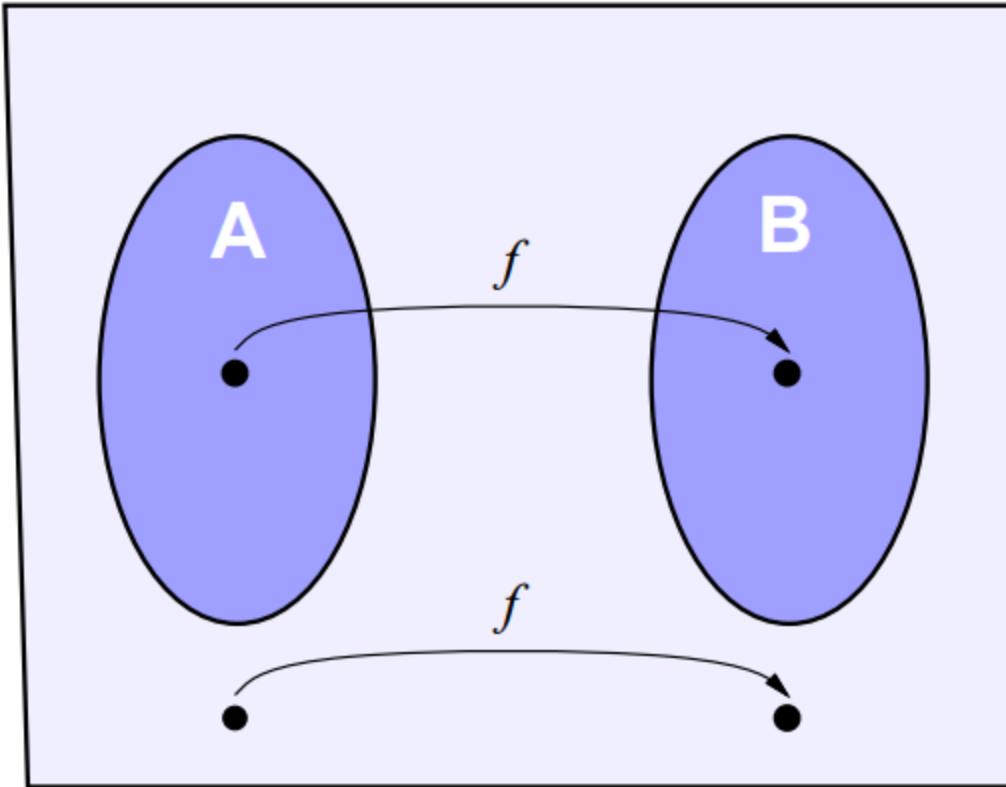
$$f : \Sigma^* \longrightarrow \Sigma^*$$

such that, for every w ,

$$w \in A \iff f(w) \in B.$$

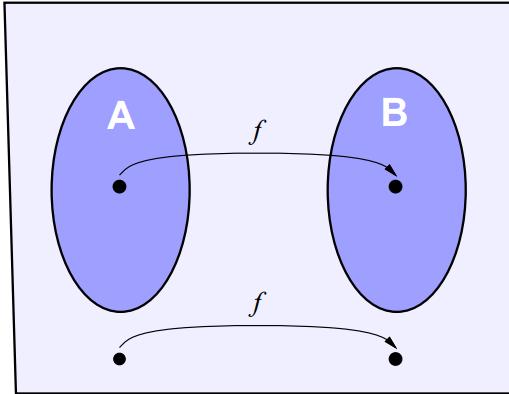
The function f is called the **reduction** from A to B .

Mapping Reductions



A mapping reduction converts questions about membership in A to membership in B

Mapping Reductions

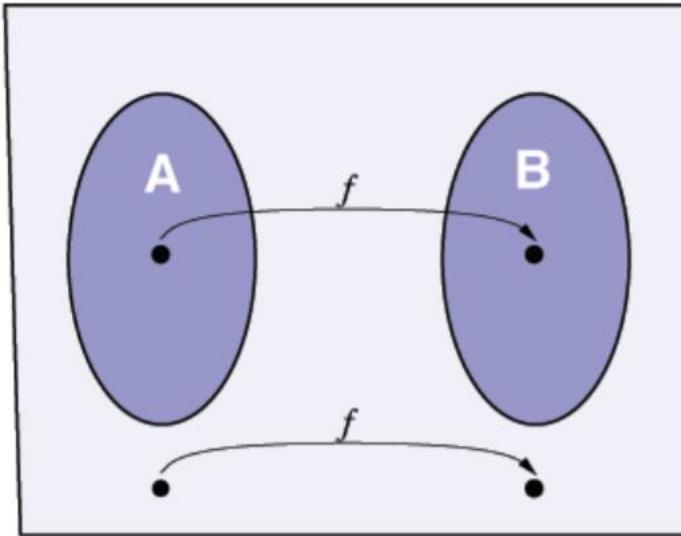


An Example:

A mapping reduction converts questions about membership in A to membership in B

Let A is 0^* , and B is $\{0, 1\}$
Let the f is defined as below.

If $w \in 0^*$ and $|w|$ is even then $f(w) = 0$,
Else if $w \in 0^*$ and $|w|$ is odd then $f(w) = 1$,
Else if $w \notin 0^*$ then $f(w) = 11$.



A mapping reduction converts questions about membership in A to membership in B

Notice that $A \leq_m B$ implies $\overline{A} \leq_m \overline{B}$.

Mapping Reductions

Theorem: If $A \leq_m B$ and B is decidable, then A is decidable.

Proof: Let

- M be the decider for B , and
- f the reduction from A to B .

Define N : On input w

1. compute $f(w)$
2. run M on input $f(w)$ and output whatever M outputs.

Mapping Reductions

Corollary: If $A \leq_m B$ and A is undecidable, then B is undecidable.

In fact, this has been our principal tool for proving undecidability of languages other than A_{TM} .

Example: Halting

Recall that

$$\begin{aligned} A_{\text{TM}} &= \{\langle M, w \rangle \mid \text{TM } M \text{ accepts input } w\} \\ H_{\text{TM}} &= \{\langle M, w \rangle \mid \text{TM } M \text{ halts on input } w\} \end{aligned}$$

Earlier we proved that

- H_{TM} undecidable
- by (de facto) reduction from A_{TM} .

Let's reformulate this.

Example: Halting

Define a computable function, f :

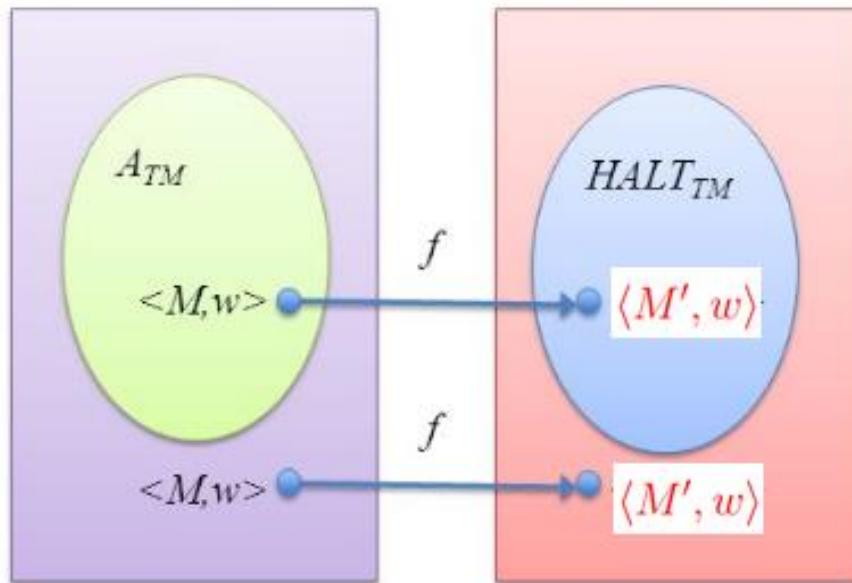
- input of form $\langle M, w \rangle$
- output of form $\langle M', w' \rangle$
- where $\langle M, w \rangle \in A_{\text{TM}} \iff \langle M', w' \rangle \in H_{\text{TM}}$.

Example: Halting

The following machine computes this function f .

$F =$ on input $\langle M, w \rangle$:

- Construct the following machine M' .
 M' : on input x
 - run M on x
 - If M accepts, *accept*.
 - if M rejects, *enter a loop*.
- output $\langle M', w \rangle$

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$
$$\leq_m$$
$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM \& } M \text{ halts on input } w\}$$


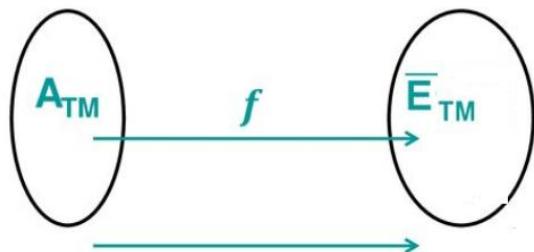
$$A_{TM} \leq_m \overline{E_{TM}}$$

- $A_{TM} = \{\langle M, w \rangle | M \text{ is a TM that accepts } w\}$
- $\overline{E_{TM}} = \{\langle M \rangle | L(M) \neq \emptyset\}$
- $f: \Sigma^* \rightarrow \Sigma^*$ can be defined as

Create M' : On input x ,

if $x \neq w$, output “Reject”;

if $x = w$, run w on M , output the result.



Mapping Reductions: Reminders

Theorem 1:

If $A \leq_m B$ and B is decidable, then A is decidable.

Theorem 2 :

If $A \leq_m B$ and B is recursively enumerable, then A is recursively enumerable.

Mapping Reductions: Corollaries

Corollary 1: If $A \leq_m B$ and A is undecidable, then B is undecidable.

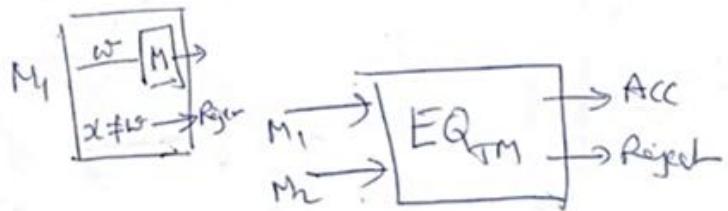
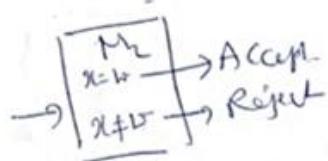
Corollary 2: If $A \leq_m B$ and A is not in \mathcal{RE} , then B is not in \mathcal{RE} .

Corollary 3: If $A \leq_m B$ and A is not in $co\mathcal{RE}$, then B is not in $co\mathcal{RE}$.

TM Equality

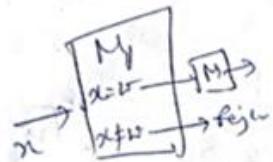
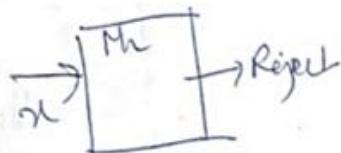
Theorem: Both EQ_{TM} and its complement, $\overline{\text{EQ}_{\text{TM}}}$, are not enumerable. Stated differently, EQ_{TM} is neither enumerable nor co-enumerable.

- We show that A_{TM} is reducible to EQ_{TM} . The same function is also a mapping reduction from $\overline{A_{\text{TM}}}$ to $\overline{\text{EQ}_{\text{TM}}}$, and thus $\overline{\text{EQ}_{\text{TM}}}$ is not enumerable.
- We then show that A_{TM} is reducible to $\overline{\text{EQ}_{\text{TM}}}$. The new function is also a mapping reduction from $\overline{A_{\text{TM}}}$ to EQ_{TM} , and thus EQ_{TM} is not enumerable.


 $A_{TM} \leq_m EQ_{TM}$


$$L(M_1) = \begin{cases} \{w\} & \text{if } M \text{ accepts } w \\ \emptyset, & \text{Otherwise} \end{cases}$$

$$L(M_2) = \{w\}$$


 $A_{TM} \leq \overline{EQ}_{TM}$


$$L(M_1) = \begin{cases} \{w\} & \text{if } M \text{ accepts } w \\ \emptyset, & \text{Otherwise} \end{cases}$$

$$L(M_2) = \emptyset$$

Alternate solutions found in the net.

$$A_{TM} \leq_m EQ_{TM}$$

Proof: The following TM computes the reduction:

F = `` On input $\langle M, w \rangle$, where M is a TM and w is a string:

1. Construct TMs M' , M'' .

M' = `` On input x ,

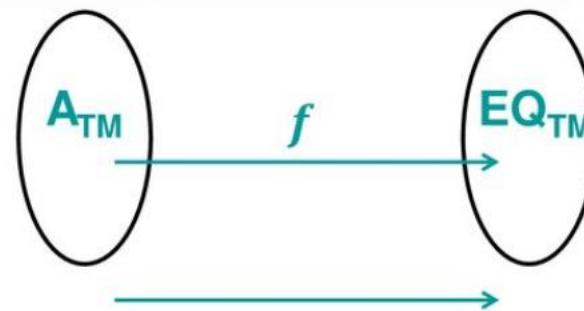
1. Ignore the input.
2. Run TM M on input w .
3. If it accepts, accept.'

M'' = `` Accept.'

2. Output $\langle M', M'' \rangle$.

$$L(M') = \begin{cases} \Sigma^*, & \text{if } M \text{ accepts } w \\ \phi, & \text{Otherwise} \end{cases}$$

$$L(M'') = \Sigma^*$$



$$A_{TM} \leq_m \overline{EQ}_{TM}$$

Proof: We give a mapping reduction $A_{TM} \leq_m \overline{EQ}_{TM}$

The following TM computes the reduction:

F = `` On input $\langle M, w \rangle$, where M is a TM and w is a string:

1. Construct TMs M', M'' .

M' = `` On input x ,

1. Ignore the input.
2. Run TM M on input w .
3. If it accepts, accept.”

M'' = `` Reject.”

2. Output $\langle M', M'' \rangle$.

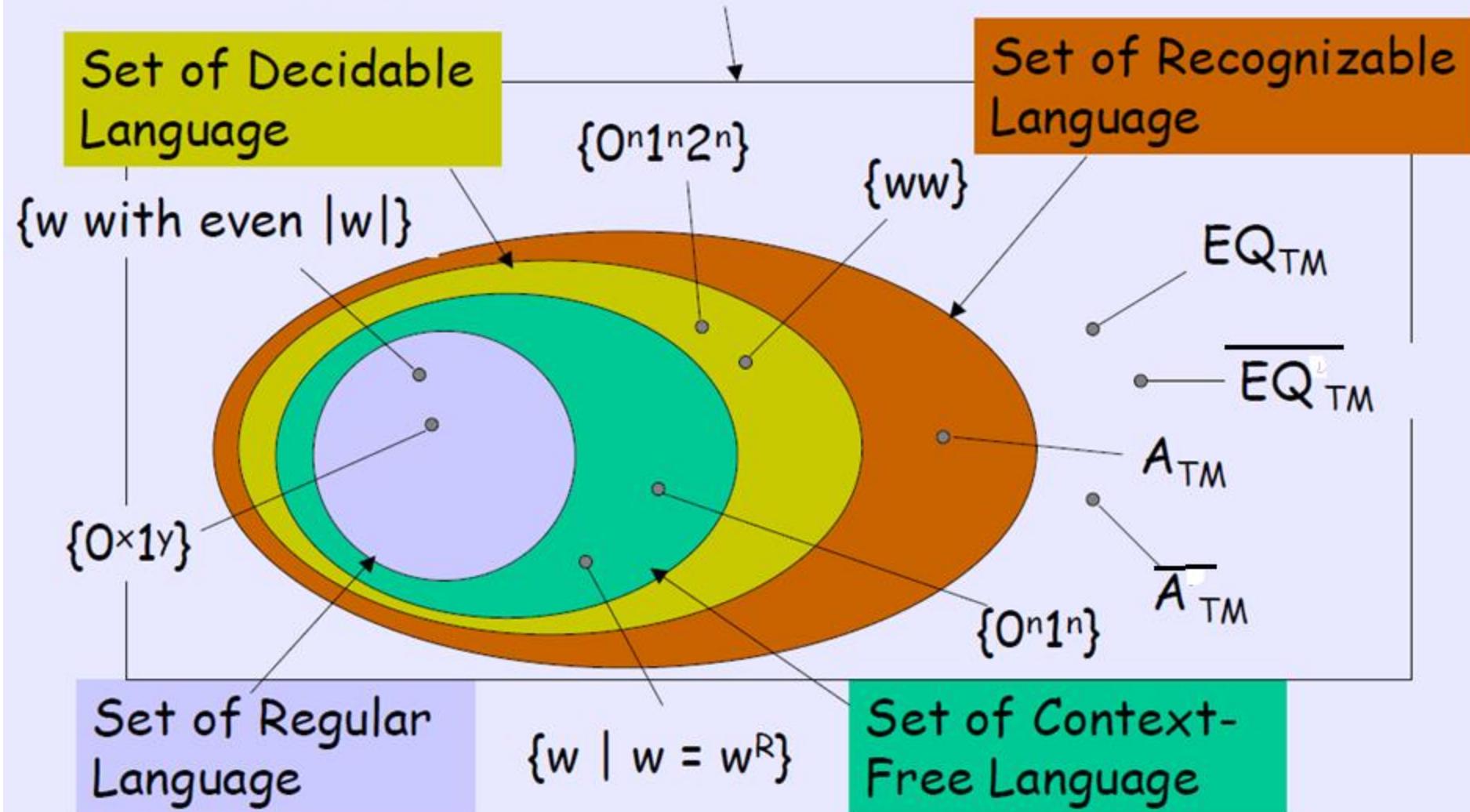


$$L(M') = \begin{cases} \Sigma^*, & \text{if } M \text{ accepts } w \\ \phi, & \text{Otherwise} \end{cases}$$

$$L(M'') = \phi$$

Language Hierarchy (revisited)

Set of Languages (= set of "set of strings")



Non Trivial Properties of \mathcal{RE} Languages

A few examples

- L is finite.
- L is infinite.
- L contains the empty string.
- L contains no prime number.
- L is co-finite.
- ...

All these are **non-trivial** properties of enumerable languages, since for each of them there is $L_1, L_2 \in \mathcal{RE}$ such that L_1 satisfies the property but L_2 does not.

Are there any **trivial** properties of \mathcal{RE} languages?

Rice's Theorem

Theorem Let \mathcal{C} be a proper non-empty subset of the set of enumerable languages. Denote by $L_{\mathcal{C}}$ the set of all TMs encodings, $\langle M \rangle$, such that $L(M)$ is in \mathcal{C} . Then $L_{\mathcal{C}}$ is undecidable.

(See problem 5.22 in Sipser's book)

Proof by reduction from A_{TM} .

Given M and w , we will construct M_0 such that:

- If M accepts w , then $\langle M_0 \rangle \in L_{\mathcal{C}}$.
- If M does not accept w , then $\langle M_0 \rangle \notin L_{\mathcal{C}}$.

<http://www.cs.tau.ac.il/~bchor/CM09/Compute9.pdf>

Has (towards end) some good slides on Rice's theorem and its consequences.

POST CORRESPONDENCE PROBLEM

- Undecidability is not just confined to problems concerning automata and languages.
- There are other “natural” problems which can be proved undecidable.
- The Post correspondence problem (PCP) is a tiling problem over strings.
- A tile or a domino contains two strings, t and b ; e.g., $\left[\begin{smallmatrix} ca \\ a \end{smallmatrix} \right]$.
- Suppose we have dominos

$$\left\{ \left[\begin{smallmatrix} b \\ ca \end{smallmatrix} \right], \left[\begin{smallmatrix} a \\ ab \end{smallmatrix} \right], \left[\begin{smallmatrix} ca \\ a \end{smallmatrix} \right], \left[\begin{smallmatrix} abc \\ c \end{smallmatrix} \right] \right\}$$

- A match is a list of these dominos so that when concatenated the top and the bottom strings are identical. For example,

$$\left[\begin{smallmatrix} a \\ ab \end{smallmatrix} \right] \left[\begin{smallmatrix} b \\ ca \end{smallmatrix} \right] \left[\begin{smallmatrix} ca \\ a \end{smallmatrix} \right] \left[\begin{smallmatrix} a \\ ab \end{smallmatrix} \right] \left[\begin{smallmatrix} abc \\ c \end{smallmatrix} \right] = \frac{abcaaabc}{abcaaabc}$$

- The set of dominos $\left\{ \left[\begin{array}{c} abc \\ ab \end{array} \right], \left[\begin{array}{c} ca \\ a \end{array} \right], \left[\begin{array}{c} acc \\ ba \end{array} \right], \right\}$ does not have a solution.

POST CORRESPONDENCE PROBLEM

AN INSTANCE OF THE PCP

A PCP instance over Σ is a finite collection P of dominos

$$P = \left\{ \left[\frac{t_1}{b_1} \right], \left[\frac{t_2}{b_2} \right], \dots, \left[\frac{t_k}{b_k} \right] \right\}$$

where for all $i, 1 \leq i \leq k, t_i, b_i \in \Sigma^*$.

MATCH

Given a PCP instance P , a **match** is a nonempty sequence

$$i_1, i_2, \dots, i_\ell$$

of numbers from $\{1, 2, \dots, k\}$ (with repetition) such that

$$t_{i_1} t_{i_2} \cdots t_{i_\ell} = b_{i_1} b_{i_2} \cdots b_{i_\ell}$$

POST CORRESPONDENCE PROBLEM

AN INSTANCE OF THE PCP

A PCP instance over Σ is a finite collection P of dominos

$$P = \left\{ \left[\frac{t_1}{b_1} \right], \left[\frac{t_2}{b_2} \right], \dots, \left[\frac{t_k}{b_k} \right] \right\}$$

where for all $i, 1 \leq i \leq k, t_i, b_i \in \Sigma^*$.

MATCH

Given a PCP instance P , a **match** is a nonempty sequence

$$i_1, i_2, \dots, i_\ell$$

of numbers from $\{1, 2, \dots, k\}$ (with repetition) such that

$$t_{i_1} t_{i_2} \cdots t_{i_\ell} = b_{i_1} b_{i_2} \cdots b_{i_\ell}$$

QUESTION:

Does a given PCP instance P have a match?

POST CORRESPONDENCE PROBLEM

QUESTION:

Does a given PCP instance P have a match?

LANGUAGE FORMULATION:

$PCP = \{\langle P \rangle \mid P \text{ is a PCP instance and it has a match}\}$

THEOREM 5.15

PCP is undecidable.

POST CORRESPONDENCE PROBLEM

QUESTION:

Does a given PCP instance P have a match?

LANGUAGE FORMULATION:

$PCP = \{\langle P \rangle \mid P \text{ is a PCP instance and it has a match}\}$

THEOREM 5.15

PCP is undecidable.

Proof: By reduction using computation histories. If PCP is decidable then so is A_{TM} . That is, if PCP has a match, then M accepts w .

- That is, A_{TM} can be reduced to PCP .
- This reduction is via a simplified PCP called $MPCP$ (modified PCP).
- Several undecidable properties of CFGs are obtained by reducing them (i.e., the corresponding languages) from PCP .

Reducibility Supplement

Relationship between reducibility
and mapping-reducibility

\leq

- $A \leq B$
 - This is saying if there is an algorithm to solve B then there is an algorithm to solve A
 - In fact, the algorithm that solves B can be converted into an algorithm that solves A.
 - If $B \in R$ then $A \in R$
 - $(B \in R) \Rightarrow (A \in R)$
 - $(A \notin R) \Rightarrow (B \notin R)$

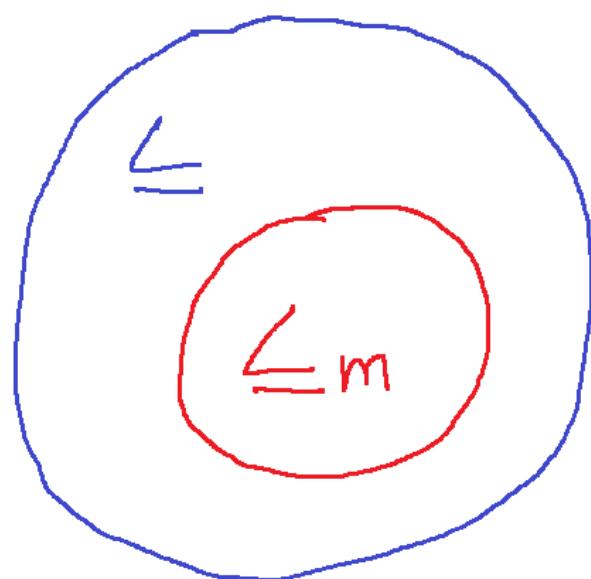
$$A \leq B \iff A \leq \bar{B}$$

- $(A \leq B) \iff ((B \in R) \Rightarrow (A \in R))$
- $(B \in R) \iff (\bar{B} \in R)$
- Since,
 $((B \in R) \Rightarrow (A \in R)) \iff ((\bar{B} \in R) \Rightarrow (A \in R))$
- Hence, $A \leq B \iff A \leq \bar{B}$

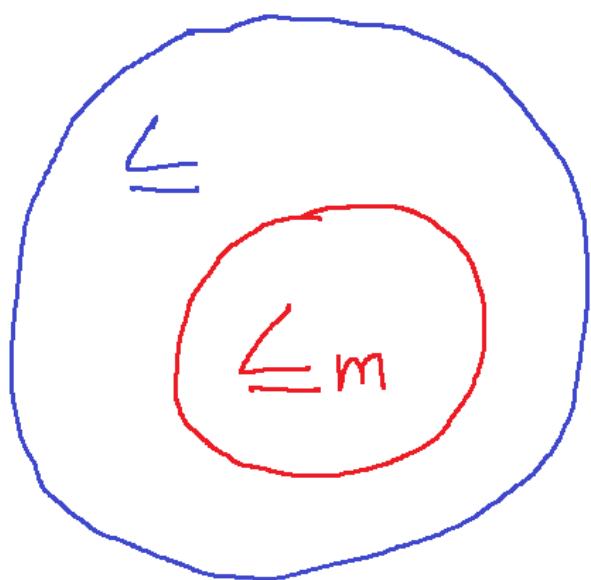
- $A \leq \bar{A}$ is trivially true.
 - But, $A \leq_m \bar{A}$ may not be true always.
- Counterexample proof:** Let $A = E_{TM}$
then if $E_{TM} \leq_m \overline{E_{TM}}$, since $\overline{E_{TM}} \in RE$, so
 $E_{TM} \in RE$. Hence $E_{TM} \in R$. This is clearly
impossible.

Relationship between the two types of reducibilities

- Mapping reducibility is specialization of reducibility (Reducibility is generalization of mapping-reducibility).



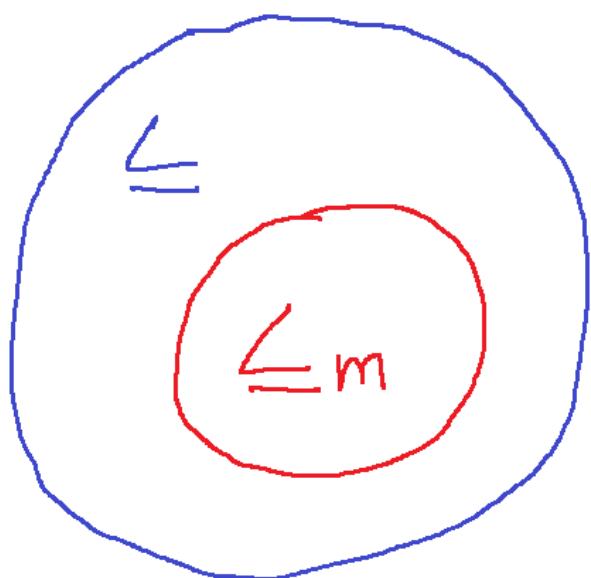
- $(A \leq_m B) \Rightarrow (A \leq B)$
- Converse is not true.



$$(A \leq_m B) \Rightarrow (A \leq B)$$

Converse is not true.

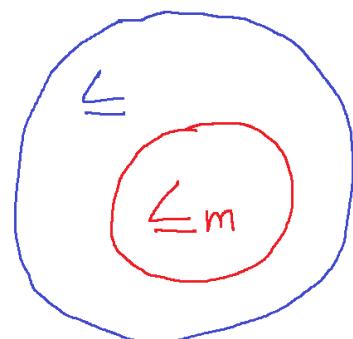
- $A_{TM} \leq_m \overline{E_{TM}}$
- Hence, we can say $A_{TM} \leq \overline{E_{TM}}$
- Note, we already know $A_{TM} \leq E_{TM}$

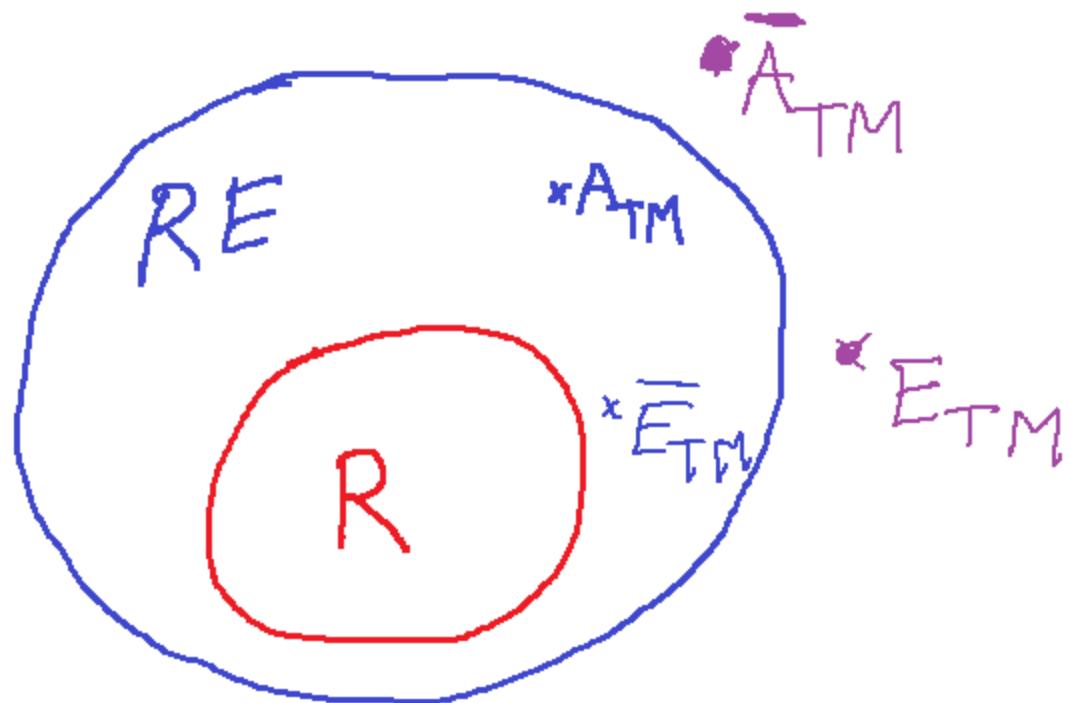


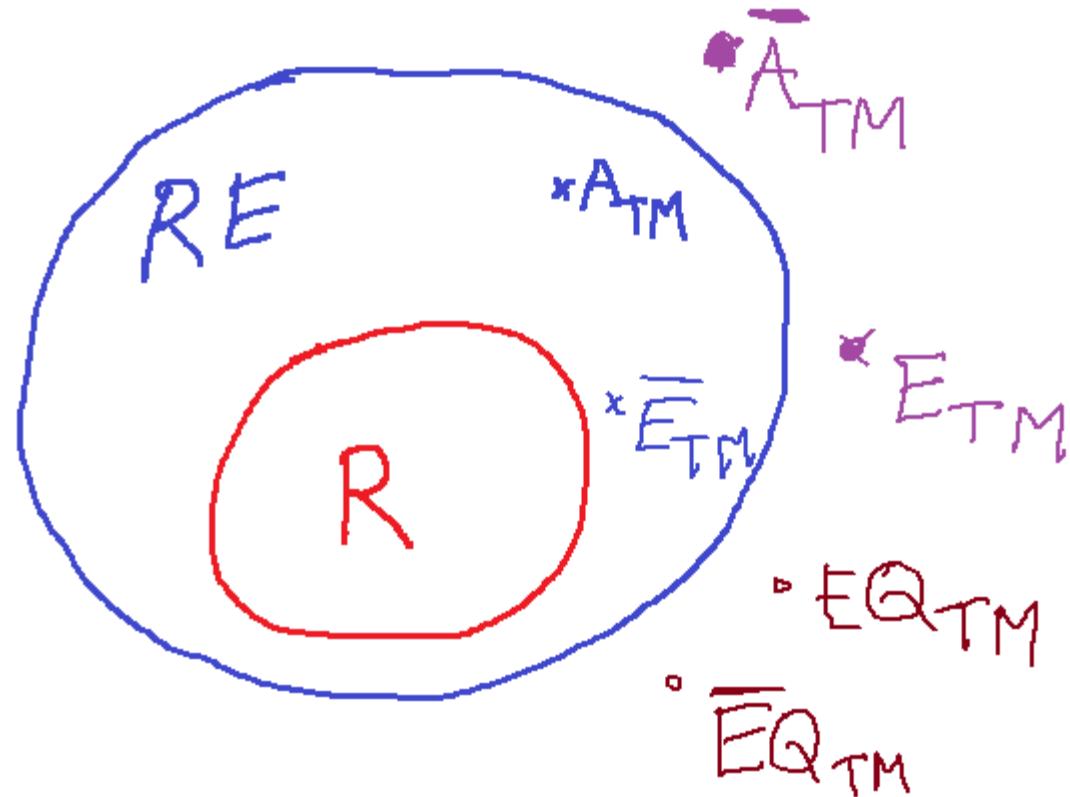
$$(A \leq_m B) \Rightarrow (A \leq B)$$

Converse is not true.

- $A_{TM} \leq_m \overline{E_{TM}}$
- Hence, we can say $A_{TM} \leq \overline{E_{TM}}$
- Note, we already know $A_{TM} \leq E_{TM}$
- It is important to understand that
 $A_{TM} \leq_m E_{TM}$ is false.







Properties of RE and R

https://youtu.be/KJD_F-QGLmo?list=PL85CF9F4A047C7BF7&t=2158

RE

- Closed under union
- Closed under intersection
- **Not closed under complementation**

R

- Closed under union
- Closed under intersection
- Closed under complementation.

Time Complexity

Objectives

- In this lecture, we focus on problems that are computable, and investigate the amount of time required to solve these problems
 - Later, we will investigate the amount of space, and other resources required to solve a problem
- Before that, we will review the big- O , small- o , big- Ω , and small- ω notations

Big-O and Big- Ω Notations

Definition: Let f and g be functions that maps \mathbb{N} to \mathbb{R}^+ . We say $f(n) = O(g(n))$ if there exists positive integers c and n' such that for every $n \geq n'$, $f(n) \leq cg(n)$.

When $f(n) = O(g(n))$, we say $g(n)$ is an asymptotic upper bound for $f(n)$

Big-O and Big-Ω Notations

We say $g(n) = \Omega(f(n))$ if $f(n) = O(g(n))$

Important: $f(n) = O(g(n))$

is a special notation, so that we will never write $O(g(n)) = f(n)$ instead

Although, we can write something like:

$f(n) = O(g(n)) = O(h(n))$, which means:

$f(n) = O(g(n))$, and $g(n) = O(h(n))$

Small-o and Small- ω Notations

Definition: Let f and g be functions that maps \mathbb{N} to \mathbb{R}^+ . We say $f(n) = o(g(n))$ if

$$\lim_{n \rightarrow \infty} f(n)/g(n) = 0$$

We say $g(n) = \omega(f(n))$ if $f(n) = o(g(n))$

Examples

Is the following true?

$$1. \ 5n^2 + 1002n + 17 = O(n^2)$$

$$2. \ \log_3 n = O(\log n)$$

$$3. \ \log n = O(\log_3 n)$$

$$4. \ \log n = O(n^{0.00001})$$

$$5. \ \log(n^2 \log n) = O(\log n)$$

$$6. \ 2^n = O(3^n)$$

$$7. \ 3^n = O(2^n)$$

$$8. \ n^{1/(\log n)} = o((n^{1/(\log n)})^2)$$

the Limit Rule

Suppose $L \leftarrow \lim_{n \rightarrow \infty} f(n)/g(n)$ exists (may be ∞)

Then if $L = 0$, then f is $O(g)$

if $0 < L < \infty$, then f is $\Theta(g)$

if $L = \infty$, then f is $\Omega(g)$

To compute the limit, the standard **L'Hopital rule** of calculus is useful: if $\lim_{x \rightarrow \infty} f(x) = \infty = \lim_{x \rightarrow \infty} g(x)$ and f, g are positive differentiable functions for $x > 0$, then $\lim_{x \rightarrow \infty} f(x)/g(x) = \lim_{x \rightarrow \infty} f'(x)/g'(x)$ where $f'(x)$ is the derivative

EXAMPLE 7.6

The following are easy to check.

1. $\sqrt{n} = o(n)$.
2. $n = o(n \log \log n)$.
3. $n \log \log n = o(n \log n)$.
4. $n \log n = o(n^2)$.
5. $n^2 = o(n^3)$.

However, $f(n)$ is never $o(f(n))$.

Analyzing Algorithms

Let A be the language $\{ 0^k 1^k \mid k \geq 0 \}$, and we have seen that A is decidable before.

Below is one such TM that decides A :

M_1 = "On input string w ,

1. Scan across the tape and **reject** if 0 appears on the right of a 1
2. Repeat if both 0s and 1s remain in tape
 - a. Scan the tape, cross off a 0 and a 1
3. If all 0s and 1s are crossed, **accept**. Otherwise, **reject**."

Analyzing Algorithms (2)

How many steps will M_1 need to decide if w is in A or not? Let n be the length of w

- Step 1 takes at most $O(n)$ steps
- Step 2 will repeat at most $n/2$ times, each time taking $O(n)$ steps
 - In total, Step 2 requires $O(n^2)$ steps
- Step 3 takes $O(n)$ steps

Thus, M_1 needs $O(n^2)$ steps to decide if w is in A or not

Running time of a DTM

- Number of steps the machine will take on input whose size is n .
- To formalize ...

Running Time

Definition: Let M be a deterministic Turing machine that halts on all inputs. The running time of M is the function $f:N \rightarrow N$, where $f(n)$ is the maximum number of steps that M uses on any input of length n

If $f(n)$ is the running time of M , we say M runs in time $f(n)$, and M is an $f(n)$ -time TM

Time Complexity Class

Definition: Let $t: \mathbb{N} \rightarrow \mathbb{R}^+$ be a function. We define the time complexity class, $\text{TIME}(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ -time Turing machine

In the previous example, M_1 is an $O(n^2)$ time TM, so that the language $A = \{0^k 1^k \mid k \geq 0\}$ is in $\text{TIME}(n^2)$

Analyzing Algorithms (3)

Can we decide $A = \{ 0^k 1^k \mid k \geq 0 \}$ faster?

Below is another TM that decides A :

M_2 = "On input string w ,

1. If 0 appears on the right of a 1, **reject**
2. Repeat if both 0s and 1s remain in tape
 - (i) If total # of 0s and 1s is odd, **reject**
 - (ii) Scan the tape, cross off every other 0.
Then cross off every other 1.
3. If all 0s and 1s are crossed, **accept**.
Otherwise, reject."

Analyzing Algorithms (4)

Question 1: Why M_2 can decide A correctly?

Question 2: What is running time of M_2 ?

- Step 1 and Step 3 takes $O(n)$ steps.
- For each time Step 2 is repeated, # of 0s is halved → repeated for $\log n$ times
- Each time Step 2 is run, it takes $O(n)$ steps → in total takes $O(n \log n)$ steps

Thus, the running time of M_2 is $O(n \log n)$

Analyzing Algorithms (5)

This implies that A is in $\text{TIME}(n \log n)$

Question 1: Earlier, we show that A is in $\text{TIME}(n^2)$... Is there a contradiction??

Question 2: Can we find a TM that decides A faster? That is, in $o(n \log n)$ time?

- The answer is NO... (if TM just have a single tape)
- In fact, it is shown that if a language can be decided by a single-tape TM in $o(n \log n)$ time, the language is regular

Analyzing Algorithms (6)

How about if we have 2 tapes?

M_3 = "On input string w ,

1. If 0 appears on the right of a 1, **reject**
2. Scan across 0s on tape 1 until first 1. At the same time, copy 0s to tape 2
3. Scan tape 1 and tape 2 together. Each time, match a 0 with a 1
4. If all 0s and 1s match, **accept**. Otherwise, **reject**."

Analyzing Algorithms (7)

The running time of M3 is $O(n)$!

What we have learnt before:

Single-tape and Multi-tape TM have the same power (in terms of computability,
I.e., whether a problem can be solved)

What we have learnt now:

Single-tape and Multi-tape does not have the same power (in terms of complexity,
I.e., how fast a problem can be solved)

Next Time

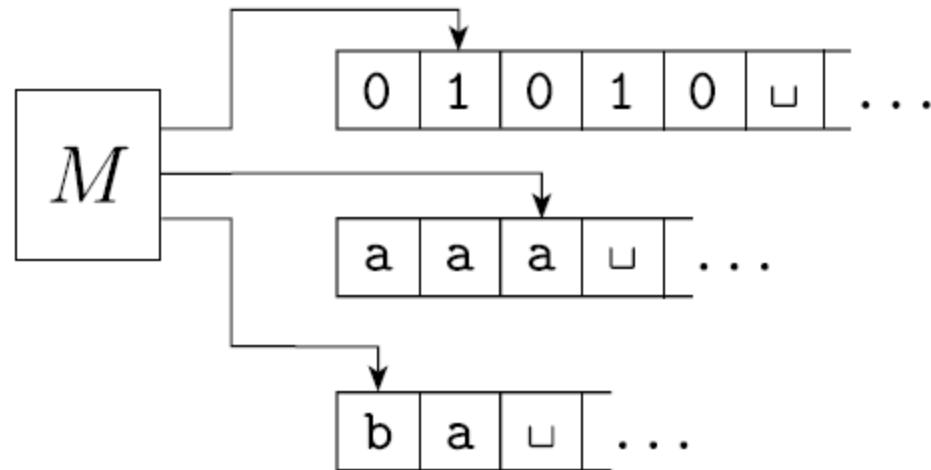
- Complexity relationship among models
 - Single-Tape versus Multi-Tape
 - Deterministic versus Non-Deterministic
- P and NP
 - Two important classes of problems in time complexity theory

Complexity Theory ...

Multi-tape, NTM, ...

MULTITAPE TURING MACHINES

A k tape Turing Machine will have k tapes (a 3 tape TM is shown in the figure)



$$\delta: Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{L, R\}^k,$$

where k is the number of tapes. The expression

$$\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, L, R, \dots, L)$$

THEOREM 3.13

Every multitape Turing machine has an equivalent single-tape Turing machine.

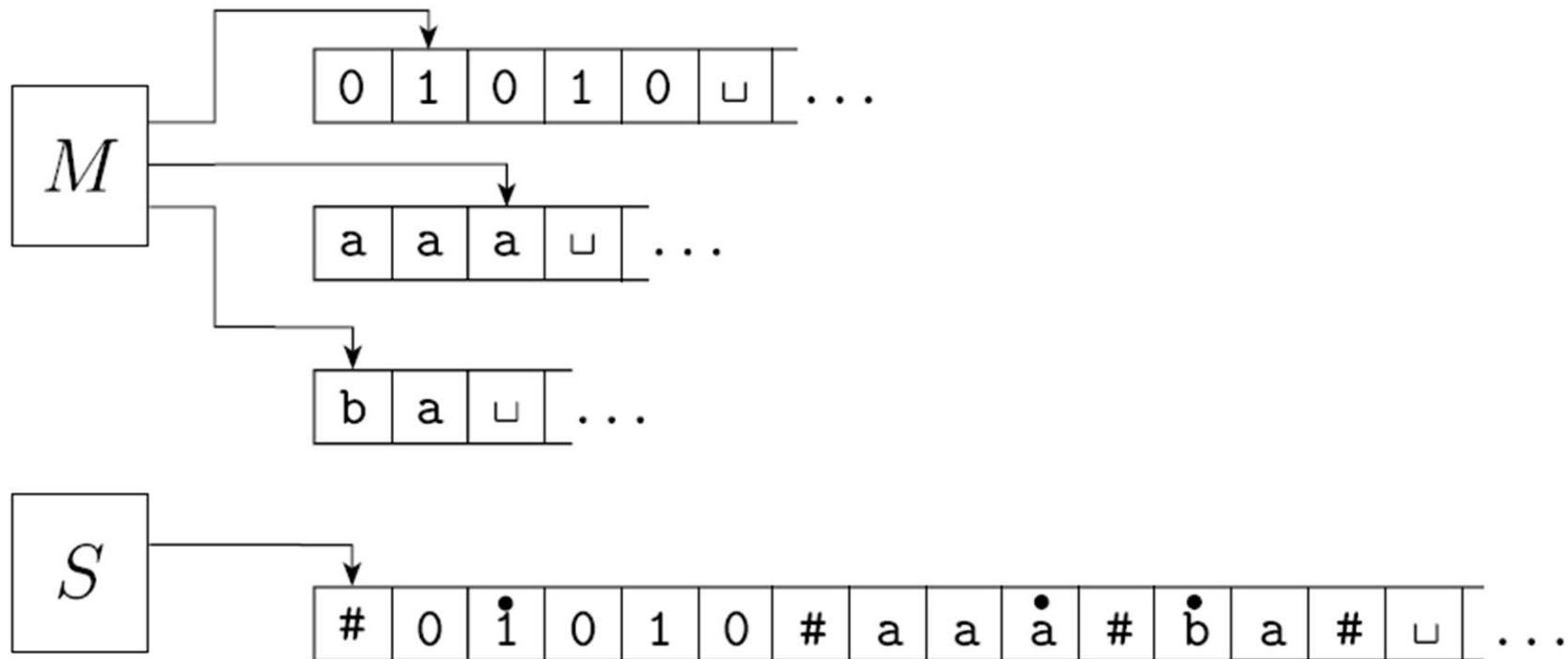


FIGURE 3.14

Representing three tapes with one

Single-Tape versus Multi-Tape

Theorem: Let $t(n)$ be a function, and $t(n) \geq n$.

Then for every $t(n)$ -time TM that works with k tapes, there is an equivalent $O(k^2 t(n)^2)$ -time TM that works with 1 tape.

Proof: Let M be a k -tape TM that runs in $t(n)$ time. We construct a single-tape TM S that runs in $O(k^2 t(n)^2)$ time.

Single-Tape vs Multi-Tape (2)

How the simulation of a k tape TM (M) can be done over a single tape TM (S):

- S uses its single tape to represent the contents of all k tapes in M
- The k tapes are stored consecutively, separated by $\#$
- Positions of tape heads are represented by “marked” symbols

Here, S uses the same way to simulate M

Single-Tape vs Multi-Tape (3)

Recall that to perform a step in M , S will do:

- Scan the tape to collect the characters under each of the tape heads in M
- Scan the tape again, update the symbol under the tape heads of M , and update the positions of the tape heads
- Special case: when a tape head of M moves rightward onto an unread portion, we add a space in the corresponding place in S 's tape (by shifting)

Single-Tape vs Multi-Tape (4)

Since M runs in $t(n)$ time, each of its tape head can access only the first $t(n)$ cells.

Thus, S will use (and access) only the first $k \times t(n) + k + 1 = O(k t(n))$ cells.

We call these $O(k t(n))$ cells
the **active portion** of S 's tape

Single-Tape vs Multi-Tape (5)

S simulates M for $O(t(n))$ steps, where each step (in the worst case) does the following:

- (1) scan the tape first
- (2) add a space to each of the k tapes,
- (3) update tape heads and its contents

For (1) and (3), S accesses only the active portion of S 's tape $\rightarrow O(k t(n))$ time

For (2), S scans the active portion for at most k times, which is $O(k^2 t(n))$ time

\rightarrow In total, it thus takes $O(k^2 t(n)^2)$ time

Polynomial Time Bounds

If the running time $t(n)$ of a machine M is $O(n^c)$ for some fixed constant $c > 0$, the running time is called **polynomial bounded**, or we say M runs in **polynomial time**. This gives the following corollary.

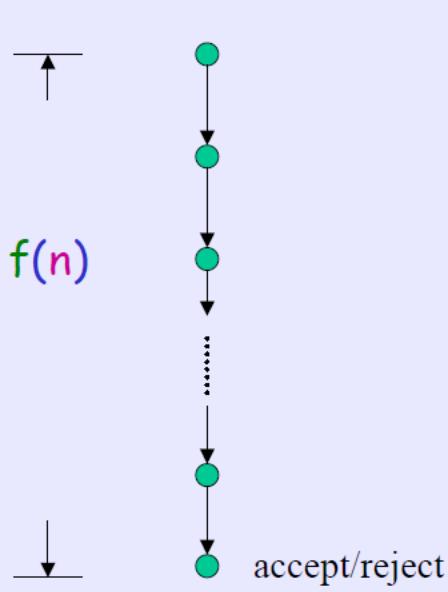
Corollary: For any k -tape TM that runs in polynomial time, it has an equivalent single-tape TM that runs in polynomial time.

NTM decider

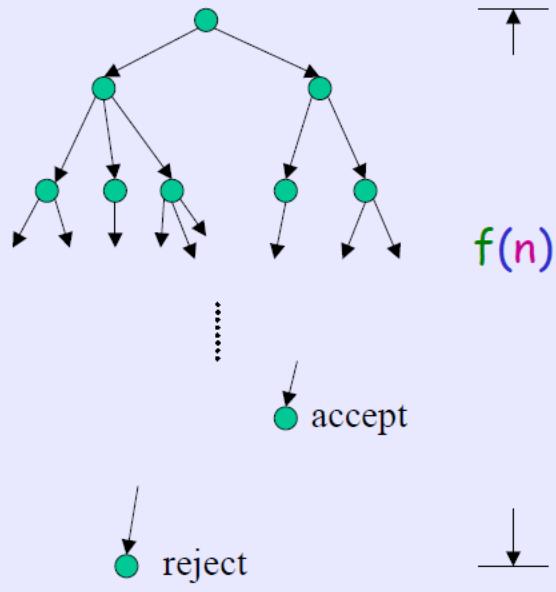
An NTM is a **decider** if all its computation branches halt on all inputs.

Definition: Let M be an NTM decider. The **running time** of M is the function $f:N \rightarrow N$, where $f(n)$ is the maximum number of steps that M uses on **any branch** of its **computation** on any input of length n

Comparison of Running Times



Deterministic time



Non-deterministic time

DTM versus NTM decider

Theorem: Let $t(n)$ be a function, $t(n) \geq n$.

Then every $t(n)$ -time single-tape NTM decider has an equivalent $2^{O(t(n))}$ -time single-tape DTM

Proof: Let M be a NTM that runs in $t(n)$ time. We construct a DTM D that simulates M by searching M 's computation tree, as described in Lecture 11. We now analyze D 's simulation.

DTM versus NTM decider (2)

- On an input of length n , every branch of computation of M has at most $t(n)$ steps
- Every node in the computation tree has at most b children, where b is the maximum number of choices in M 's transition \rightarrow number of leaves is at most $O(b^{t(n)})$
- Also, total number of nodes + leaves is at most $O(t(n) b^{t(n)})$ (why??)

DTM versus NTM decider (3)

- The simulation proceeds by visiting the nodes (including leaves) in BFS order. Here, when we visit a node v , we always travel starting from the root
→ time to visit v is $O(t(n))$

Thus, the total time for D to simulate M is
 $O(t(n)^2 b^{t(n)}) = 2^{O(t(n))}$ (why??)

Next Time

- P and NP
 - Two important classes of problems in time complexity theory

Time Complexity of NTM

Relationship with DTM

Recap of Converting a NTM to DTM

Computation of NTM

- The transition function of NTM has the form

$$\delta : Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L, R\}}$$

- For an input w , we can describe all possible computations of NTM by a **computation tree**, where

root = start configuration,

children of node C = all configurations that can be yielded by C

- The NTM **accepts** the input w if **some** branch of computation (i.e., a path from root to some node) leads to the accept state

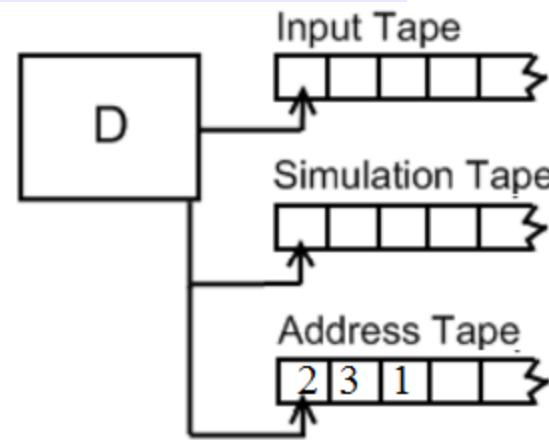
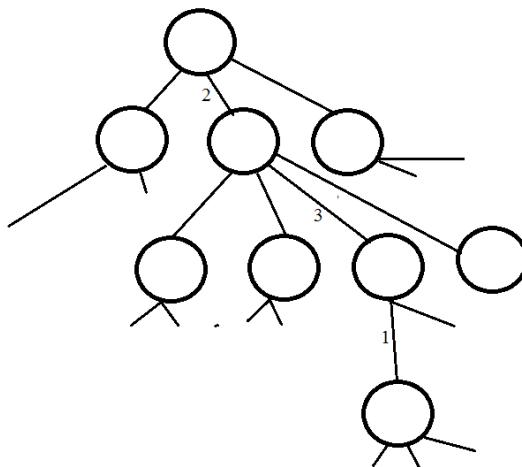
NTM = TM

Theorem: Given an NTM that recognizes a language L , we can find a TM that recognizes the same language L .

Proof: Let N be the NTM. We show how to convert N into some TM D . The idea is to simulate N by trying all possible branches of N 's computation. If one branch leads to an accept state, D accepts. Otherwise, D 's simulation will not terminate.

NTM = TM (Proof)

- To simulate the search, we use a 3-tape TM for D
 - first tape stores the input string
 - second tape is a working memory, and
 - third tape "encodes" which branch to search
- What is the meaning of "encode"?



NTM = TM (Proof)

- Let $b = |Q \times \Gamma \times \{L, R\}|$, which is the maximum number of children of a node in N's computation tree.
- We encode a branch in the tree by a string over the alphabet $\{1, 2, \dots, b\}$.
 - E.g., 231 represents the branch:
root r → r's 2nd child c →
c's 3rd child d → d's 1st child

NTM = TM (Proof)

On input string w ,

Step 1. D stores w in Tape 1 and \square in Tape 3

Step 2. Repeat

2a. Copy Tape 1 to Tape 2

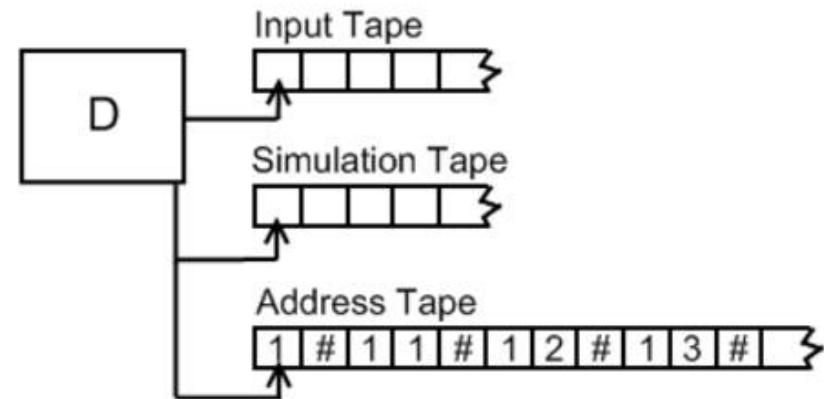
2b. Simulate N using Tape 2, with the branch of computation specified in Tape 3.

Precisely, in each step, D checks the next symbol in Tape 3 to decide which choice to make. (Special case ...)

NTM = TM (Proof)

2b [Special Case].

1. If this branch of N enters accept state, accepts w
 2. If no more chars in Tape 3, or a choice is invalid, or if this branch of N enters reject state, D aborts this branch
- 2c. Copy Tape 1 to Tape 2, and update Tape 3 to store the next branch (in Breadth-First Search order)



NTM = TM (Proof)

- In the simulation, D will first examine the branch ε (i.e., root only), then the branch 1 (i.e., root and 1^{st} child only), then the branch 2 , and then $3, 4, \dots, b$, then the branches $11, 12, 13, \dots, 1b$, then $21, 22, 23, \dots, 2b$, and so on, until the examined branch of N enters an accept state (what if N enters a reject state?)
- If N does not accept w , the simulation of D will run forever
- Note that we cannot use DFS (depth-first search) instead of BFS (why?)

TIME COMPLEXITY RELATION BETWEEN DTM & NTM

NTM decider

An NTM is a **decider** if all its computation branches halt on all inputs.

Definition: Let M be an NTM decider. The **running time** of M is the function $f:N \rightarrow N$, where $f(n)$ is the maximum number of steps that M uses on **any branch of its computation** on any input of length n .

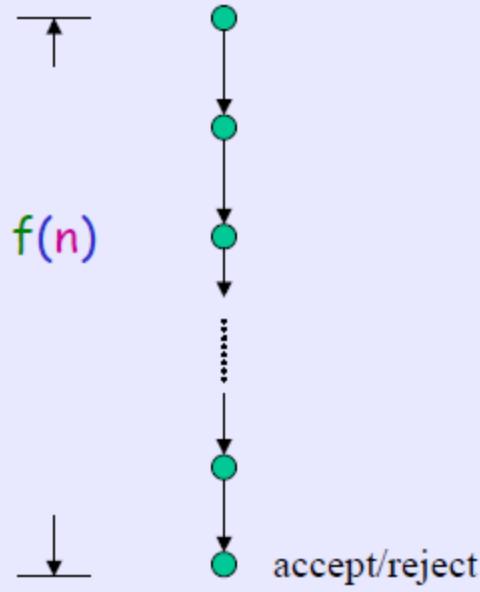
DTM versus NTM decider

Theorem: Let $t(n)$ be a function, $t(n) \geq n$.

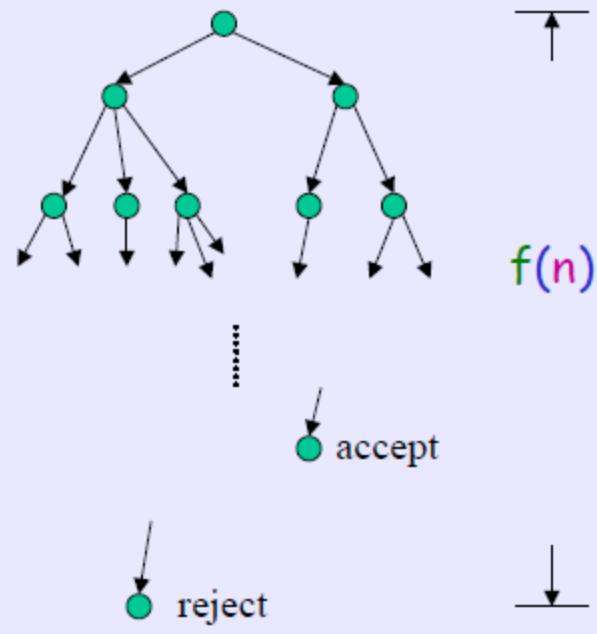
Then every $t(n)$ -time single-tape NTM decider has an equivalent $2^{O(t(n))}$ -time single-tape DTM

Proof: Let M be a NTM that runs in $t(n)$ time. We construct a DTM D that simulates M by searching M 's computation tree. We now analyze D 's simulation.

Comparison of Running Times



Deterministic time



Non-deterministic time

DTM versus NTM decider (2)

- On an input of length n , every branch of computation of M has at most $t(n)$ steps
- Every node in the computation tree has at most b children, where b is the maximum number of choices in M 's transition \rightarrow number of leaves is at most $O(b^{t(n)})$
- Total number of nodes in the tree \leq 2 times number of leaves
- Hence total number of nodes = $O(b^{t(n)})$

DTM versus NTM decider (3)

- The simulation proceeds by visiting the nodes (including leaves) in BFS order. Here, when we visit a node v , we always travel starting from the root
→ time to visit v is $O(t(n))$

- Therefore, running time = $O(t(n) b^{t(n)}) = 2^{O(t(n))}$.
- This is a 3 tape DTM, to simulate this over a single tape DTM, we need to square the upperbound, so
- Running time over single-tape DTM

$$\begin{aligned} &= \left(2^{O(t(n))}\right)^2 = 2^{O(2t(n))} \\ &= 2^{O(t(n))}. \end{aligned}$$

Time Complexity Class

Definition: Let $t: \mathbb{N} \rightarrow \mathbb{R}^+$ be a function. We define the time complexity class, $\text{TIME}(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ -time Turing machine

the language $A = \{0^k 1^k \mid k \geq 0\}$ is in $\text{TIME}(n^2)$

The Class P

Definition: **P** is the class of languages that are decidable in polynomial time on a single-tape DTM. In other words,

$$\bigcup_{k=1} \text{TIME}(n^k)$$

- **P** is invariant for all computation models that are **polynomially** equivalent to the single-tape DTM, and
- **P** roughly corresponds to the class of problems that are realistically solvable

Further points to notice

- When we describe an algorithm, we usually describe it with **stages**, just like a step in the TM, except that each stage may actually consist of many TM steps
- Such a description allows an easier (and clearer) way to analyze the running time of the algorithm

Further points to notice (2)

- So, when we analyze an algorithm to show that it runs in poly-time, we usually do:
 1. Give a polynomial upper bound on the number of stages that the algorithm uses when its input is of length n
 2. Ensure that each stage can be implemented in polynomial time on a reasonable deterministic model
- When the two tasks are done, we can say the algorithm runs in poly-time (why??)

Further points to notice (3)

- Since time is measured in terms of n , we have to be careful how to encode a string
- We continue to use the notation $\langle \rangle$ to indicate a **reasonable** encoding
- E.g., the graph encoding in (V,E) , DFA encoding in (Q,Σ,δ,q_0,F) , are reasonable
- E.g., to encode a number in unary, such as using 111111111111111 to represent 17, is not reasonable since it is exponentially larger than any base- k encoding with $k > 1$

Further points to notice (3)

- Since time is measured in terms of n , we have to be careful how to encode a string
 - We continue to use the notation $\langle \rangle$ to indicate a **reasonable** encoding
 - E.g., the graph encoding in (V,E) , DFA encoding in (Q,Σ,δ,q_0,F) , are reasonable
 - E.g., to encode a number in unary, such as using 1111111111111111 to represent 17, is not reasonable since it is exponentially larger than any base- k encoding with $k > 1$
-
- **That which works in polynomial time with a binary encoded number, may take exponential time with unary encoded number.**

Examples of Languages in P

Let PATH be the language

$\{ \langle G, s, t \rangle \mid G \text{ is a graph with path from } s \text{ to } t \}$

Theorem: PATH is in P.

How to prove??

... Find a decider for PATH that runs in polynomial time

$PATH = \{\langle G, s, t \rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}.$

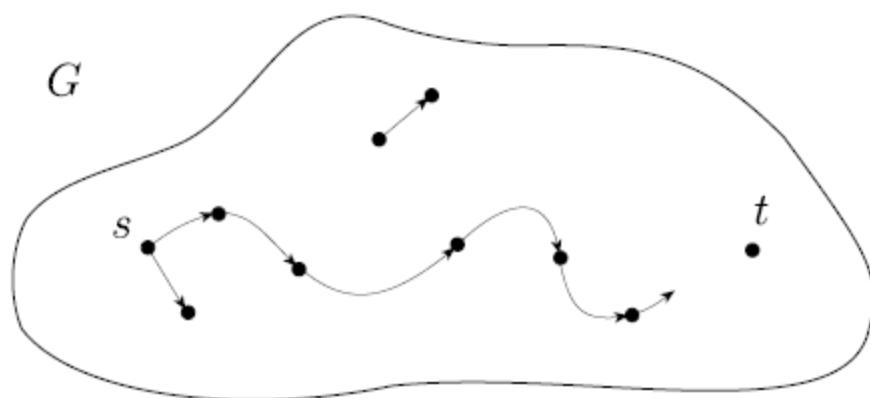


FIGURE 7.13

The $PATH$ problem: Is there a path from s to t ?

PATH is in P

Proof: A polynomial time decider M for PATH operates as follows:

M = "On input $\langle G, s, t \rangle$,

1. Mark node s
2. Repeat until no new nodes are marked
 - i. Scan all edges of G to find an edge that has exactly one marked node.
Mark the other node
3. If t is marked, accept. Else, reject."

PATH is in P (2)

What is the running time for M?

- Let m be the number of nodes in G
- Stages 1 and 3 each involves $O(1)$ scan of the input
- Stage 2 has at most m runs, each run checks at most all the edges of G . Thus, each run involves at most $O(m^2)$ scans of the input → Stage 2 involves $O(m^3)$ scans
- Since $m = O(n)$, where n = input length, the total time is polynomial in n

Every CFL is in P

Theorem: Every CFL is in P

How to prove??

... Let's recall an old idea for deciding a particular CFL ...

Every CFL is in P (2)

Proof(?): Let C be the CFL and G be the CFG in Chomsky Normal form that generates C . Define M as follows:

M = "On input $w = w_1 w_2 \dots w_n$,

1. Construct all possible derivations in G with $2n-1$ steps
2. If any derivation generates w , accept.
Else, reject."

Quick Quiz: Does M run in polynomial time?

Every CFL is in P (2)

Proof(?): Let C be the CFL and G be the CFG in Chomsky Normal form that generates C . Define M as follows:

M = "On input $w = w_1 w_2 \dots w_n$,

1. Construct all possible derivations in G with $2n-1$ steps
2. If any derivation generates w , accept.
Else, reject."

Quick Quiz: Does M run in polynomial time?

- If number of productions is p , then each step can use (in the worst case) any of these p productions. So, total number of derivations (each of size $2n - 1$) is $O(p^{2n-1})$.
- Obviously, this is not polynomial in n .

But, CYK algorithm runs in $O(n^3)$ time.

The Class NP

Definition: A **verifier** for a language A is an algorithm V , where

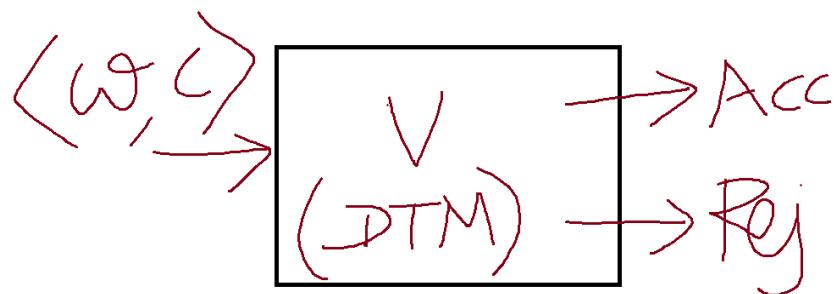
$$A = \{ w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c \}$$

A **polynomial-time verifier** is a verifier that runs in time polynomial in the length of the input w .

- **c is called a certificate or proof.**
- **If there is a polynomial-time verifier then the length of c is a polynomial in the length of w.**
 {in poly-time, the verifier can see only this much!}

V is a DTM

- It is important to understand that V is indeed a DTM that takes $\langle w, c \rangle$ as input and decides



- Further, V runs in $O(n^k)$ where $|w| = n$.
- V is called a poly-time verifier.

c is also a string over Γ

- c is also a string over the tape alphabet Γ , just like w is a string over Γ .

The Class NP

A language A is polynomially verifiable if it has a polynomial time verifier.

Definition: **NP** is the class of language that is polynomially verifiable.

The Class NP

A language A is polynomially verifiable if it has a polynomial time verifier.

Definition: NP is the class of language that is polynomially verifiable.

- Such a V exists.
- One can give the working of V as an **algorithm** and show that it runs in poly-time over the length of w .

HAMILTON is set of graphs where each graph is having a Hamiltonian cycle.

Examples of Languages in NP

Let **HAMILTON** be the language

$\{ \langle G \rangle \mid G \text{ is a Hamiltonian graph} \}$

Theorem: **HAMILTON** is in NP.

How to prove?? ... Define a polynomial time verifier **V**, and for each $\langle G \rangle$ in **HAMILTON**, define a string **c**, and show $\{ \langle G \rangle \mid V \text{ accepts } \langle G, c \rangle \} = HAMILTON$

- For given graph G , there exists a certificate c whose length is a polynomial of $|<G>|$ such that the V accepts $\langle G, c \rangle$.
- Such c exists, which is nothing but the representation of the Hamiltonian cycle, list of nodes in G (obeying to the constraint that successive nodes in the list are connected, and no node is repeated (except the first and last)). {Next slide explains this ...}
- Note, $|c| = O(|<G>|^k)$ and V runs in poly-time of $|<G>|$.

HAMILTON is in NP

Proof: Define a TM V as follows:

V = "On input $\langle G, c \rangle$,

1. If c is a cycle in G that visits each vertex once, accept
2. Else, reject."

- Note: V runs in time polynomial in length of $\langle G \rangle$ (why?)
- To show HAMILTON is in NP, it remains to show V is a verifier for HAMILTON

HAMILTON is in NP

Proof: Define a TM V as follows:

V = "On input $\langle G, c \rangle$,

1. If c is a cycle in G that visits each vertex once, accept
2. Else, reject."

- Note: V runs in time polynomial in length of $\langle G \rangle$ (why?)
- To show HAMILTON is in NP, it remains to show V is a verifier for HAMILTON

$c = (n_{i_1}, n_{i_2}, \dots, n_{i_m}, n_{i_1})$. A list of nodes.

$G = (\text{set of nodes}, \text{set of edges})$.

If number of nodes is m , what is the size of G ? For Adjacency List, it is $O(m^2)$.

So, V can work in $O(m^3)$.

V does not accidentally accept $\langle G, c \rangle$

- One has to show the correctness of the V .
- That is V accepts $\langle G, c \rangle$ if and only if $\langle G \rangle$ is in HAMILTON.
- This can be easily shown
 1. \Rightarrow (the if part)
 2. \Leftarrow (the only if part)

HAMILTON is in NP (2)

To show V is a verifier, we let $H = \{ \langle G \rangle \mid V$ accepts $\langle G, c \rangle \}$, and show $H = \text{HAMILTON}$

For every $\langle G \rangle$ in H , there is some c that V accepts $\langle G, c \rangle$. This implies $\langle G \rangle$ is a Hamiltonian graph, and $H \subseteq \text{HAMILTON}$

For every $\langle G \rangle$ in HAMILTON , let c be one of the hamilton cycle in the graph. Then, V accepts $\langle G, c \rangle$, and so $\text{HAMILTON} \subseteq H$

Similar to Hamiltonian cycle, there is Hamiltonian PATH

$HAMPATH = \{\langle G, s, t \rangle | G \text{ is a directed graph}$
with a Hamiltonian path from s to $t\}$.

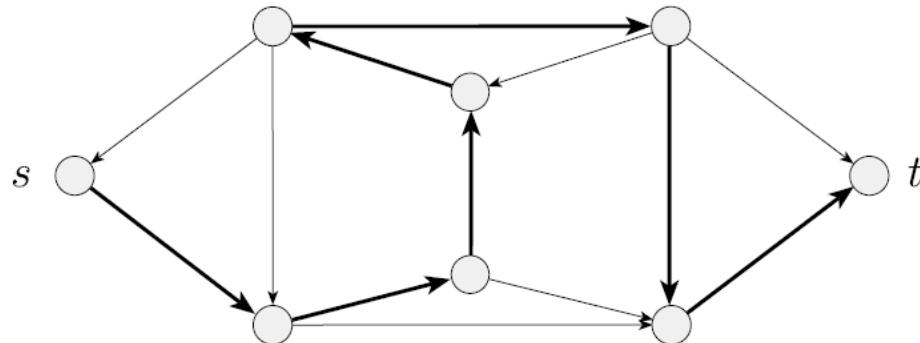


FIGURE 7.17

A Hamiltonian path goes through every node exactly once

Examples of Languages in NP (2)

Let COMPOSITE be the language

$$\{ x \mid x \text{ is a composite number} \}$$

Theorem: COMPOSITE is in NP.

How to prove?? ... Define a polynomial time verifier V , and for each x in COMPOSITE , define a string c , and show that $\{ x \mid V \text{ accepts } \langle x, c \rangle \} = \text{COMPOSITE}$

COMPOSITE is in NP

Proof: Define a TM V as follows:

V = "On input $\langle x, c \rangle$,

1. If c is not 1 or x , and c divides x ,
 accept
2. Else, reject."

- Note: V runs in time polynomial in length
 of $\langle x \rangle$ (why?)
- To show **COMPOSITE** is in NP, it remains
 to show V is a verifier for **COMPOSITE**

COMPOSITE is in NP (2)

To show V is a verifier, we let $C = \{ x \mid V$
accepts $\langle x, c \rangle \}$, and show $C = \text{COMPOSITE}$

For every x in C , there is some c that V
accepts $\langle G, c \rangle$. This implies x is a
composite number, and $C \subseteq \text{COMPOSITE}$

For every x in COMPOSITE , let c be one of
the divisor of x with $1 < c < x$. Then, V
accepts $\langle x, c \rangle$, and so $\text{COMPOSITE} \subseteq C$

Next ...

- **NP** actually means
Non-deterministically Polynomial.
- This is from NTM...
- Next, we show that for a language in NP there
is a NTM that decides in poly-time.

Objectives

- More discussion on the class NP
- Cook-Levin Theorem

The Class NP revisited

- Recall that NP is the class of language that has a polynomial time verifier
- If a language L is in NP, there is a polynomial time TM V (verifier) such that
 - For each w in L, there is some string c such that $\langle w, c \rangle$ is accepted by V, and
 - For every w not in L, $\langle w, c \rangle$ is rejected by V for all c

The Class NP revisited (2)

The reason why V is called a verifier for L :

- if a string w is in L , it has at least one certificate c to present to V , allowing V to verify that it is in L
- if a string w is not in L , it will not have any certificate c to present to V , so that V will never make a mistake to verify that it is in L

V is actually like a judge in the old days: It tends to say you are guilty, unless you can prove yourself to be innocent!

Examples (HAMPATH)

- A trivial certificate for a string $\langle G \rangle$ in HAMPATH (i.e., for a graph G to be Hamiltonian) is the order of vertices visited in a Hamiltonian path in G
 - We can find a verifier V that uses this trivial certificate to prove $\langle G \rangle$ is in HAMPATH in polynomial time (in terms of the length of $\langle G \rangle$)
 - Also, using this V , a non-Hamiltonian graph can never find a certificate to “prove” that it is Hamiltonian

Examples (COMPOSITE)

- The certificate for a string $\langle x \rangle$ in **COMPOSITE** (I.e., the number x is a composite number) is a factor of x between 2 to $x-1$
 - The corresponding verifier can use the certificate to prove $\langle x \rangle$ is in **COMPOSITE** in polynomial time (in terms of $\log x$ --- the length of $\langle x \rangle$)
- Also, when a number not composite, no certificate can fool this verifier

Properties of NP

Theorem: A language is in NP if and only if it is decided by a NTM that runs in polynomial time.

Prove idea: We show how to convert a verifier into a NTM and vice versa...

Properties of NP (2)

Proof: (\Rightarrow) Let A be a language in NP, so that it can be verified by some polynomial time verifier V . Let n^k be the running time of V . We create a polynomial time NTM N that decides A as follows:

N = "On input w ,

1. Select a string c of length at most n^k
2. Run V on $\langle w, c \rangle$
3. If V accepts, accept. Else, reject."

Properties of NP (3)

Proof: (\Leftarrow) Let A be a language decided by some polynomial time NTM N . We construct a polynomial time verifier V as follows:

V = "On input $\langle w, c \rangle$,

1. Simulate N on w , treating c as the description of the non-deterministic choice of N at each step
2. If this branch of computation in N accepts, accept. Else, reject."

Properties of NP (4)

Definition: $\text{NTIME}(t(n))$ = the set of languages that can be decided by an NTM that runs in $O(t(n))$ time

Based on the above definition and the previous theorem, we have:

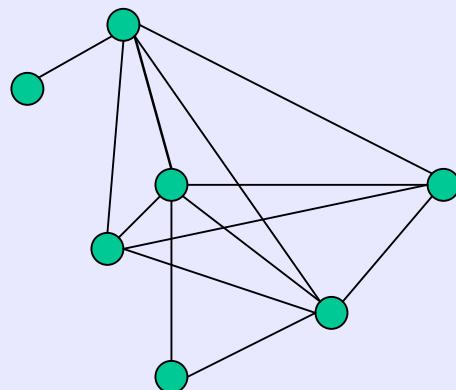
Corollary: $\text{NP} = \bigcup_k \text{NTIME}(n^k)$

More Examples of NP

Definition: A **clique** is a subgraph G' in an undirected graph G such that every two nodes in G' are connected.

Definition: A **k -clique** is a clique that contains k nodes.

E.g.,



Can you find a 5-clique here?

More Examples in NP (2)

Let CLIQUE be the language

{ $\langle G, k \rangle$ | G is a graph with a k -clique }

Theorem: CLIQUE is in NP.

How to prove??

CLIQUE is in NP

Proof 1 (using DTM verifier): What should be the certificate that a graph can prove itself is a k -clique??

The following is the verifier for CLIQUE:

V = "On input $\langle G, k, c \rangle$

1. Test if c is a set of k nodes in G
2. Test whether every two nodes in c are connected
3. If both pass, accept. Else, reject."

CLIQUE is in NP (2)

Proof 2 (using NTM decider): What should be the "guess" made by the NTM in order to check if G contains a k -clique??

The NTM below is one that decides CLIQUE:

N = "On input $\langle G, k \rangle$

1. Non-deterministically select a subset c containing k nodes of G
2. Check if every two nodes in c are connected
3. If yes, accept. Else, reject."

Another Examples in NP

Let SUBSET-SUM be the language

$\{ \langle S, t \rangle \mid S \text{ is a set of integers such that}$
 $\text{a subset of } S \text{ adds up to } t \}$

Theorem: SUBSET-SUM is in NP.

How to prove??

SUBSET-SUM is in NP

Proof 1 (using DTM verifier): What should be the certificate that S can prove itself has a subset that adds up to t ??

The verifier below is one for SUBSET-SUM:

V = "On input $\langle S, t, c \rangle$

1. Test if c is a set of numbers in S
2. Test if the numbers in c adds up to t
3. If both pass, accept. Else, reject."

SUBSET-SUM is in NP (2)

Proof 2 (using NTM decider): What should be the non-deterministic guess made by the NTM in order to check if S contains a subset that adds up to t ??

The NTM below is one that decides
SUBSET-SUM:

N = "On input $\langle S, t \rangle$

1. Non-deterministically find a subset c of S
2. Check if the numbers in c adds up to t
3. If yes, accept. Else, reject."

P versus NP

Roughly speaking:

P = the class of language that can be decided "quickly"

NP = the class of language that can be verified "quickly"

- The power of the polynomial time NTM decider seems to be much greater than the polynomial time DTM decider...

P versus NP (2)

- ... Unfortunately, so far, nobody can tell whether $P = NP$, or $P \neq NP$, is true
- A general belief (which may not be true) is $P \neq NP$ because people has input a lot of effort to find polynomial time algorithms for certain problems in NP, but fail
- What we can conclude safely so far is:

$$P \subseteq NP \subseteq \bigcup_k \text{TIME}(2^{n^k}) = \text{EXPTIME}$$

NP-Completeness

In the early 1970s, Stephen Cook and Leonid Levin (separately) discovered that:

Some languages in NP, if any of them are decidable by a DTM in polynomial time, will imply ALL problems in NP can be decided by a DTM in polynomial time

That is, they discovered some language L, such that if L is in P, then P = NP

NP-Completeness (2)

These problems are called NP-complete problems, and they form a class NP-C
(We shall give formal definition later)

The first NP-complete problem we present is called the satisfiability problem

Satisfiability Problem

Definition: A variable v is called a Boolean variable if it has a value either 1 (TRUE) or 0 (FALSE)

Definition: The Boolean operations \wedge , \vee , \neg , are defined as follows:

$$0 \wedge 0 = 0, \quad 0 \vee 0 = 0, \quad \neg 0 = 1$$

$$0 \wedge 1 = 0, \quad 0 \vee 1 = 1, \quad \neg 1 = 0$$

$$1 \wedge 0 = 0, \quad 1 \vee 0 = 1,$$

$$1 \wedge 1 = 1, \quad 1 \vee 1 = 1$$

Satisfiability Problem (2)

Definition: A Boolean formula is an expression involving Boolean variables and operations

E.g., The following is a Boolean formula:

$$F = (\neg x \wedge y) \vee (x \wedge \neg z)$$

Satisfiability Problem (2)

Definition: A Boolean formula is **satisfiable** if some assignments of 0 and 1 to the variables makes the value of the formula equal to 1

E.g., The previous Boolean formula

$$F = (\neg x \wedge y) \vee (x \wedge \neg z)$$

is satisfiable because if we set $x = 0$, $y = 1$, and $z = 1$, the value of F becomes 1

Cook-Levin Theorem

Let SAT be the language

$\{ \langle F \rangle \mid F \text{ is a satisfiable Boolean formula} \}$

Theorem: SAT is P if and only if $P = NP$

Next Time

- Polynomial Time Reducibility
- Prove Cook-Levin Theorem
- Proving other problems to be NP-Complete

Objectives

- Polynomial Time Reducibility
- Prove Cook-Levin Theorem

Polynomial Time Reducibility

- Previously, we learnt that if a problem A can be 'mapped' in finite steps into another problem B, we conclude that
 1. "if B is decidable, A is decidable"
 2. "if B is recognizable, A is recognizable"
- This is called mapping reducibility
- Suppose that we restrict the mapping reducibility to be done in polynomial time. What can we conclude?

Polynomial Time Reducibility (2)

We define (this slide + next slide):

Definition: A function $f:\Sigma^*\rightarrow\Sigma^*$ is a **polynomial-time computable function** if some polynomial-time TM M exists that halts with just $f(w)$ on its tape, when started with input w

Polynomial Time Reducibility (3)

Definition: Language A is polynomial-time mapping reducible, or simply **polynomial-time reducible**, to language B, written as $A \leq_p B$, if a polynomial-time computable function f exists, where for each w,

$$w \in A \Leftrightarrow f(w) \in B$$

The function f is called a polynomial-time reduction of A to B

Definition of NP-Complete

Definition: Language B is **NP-complete** if

1. B is in NP, and
2. every language A in NP is polynomial-time reducible to B

What is so special about NP-complete?

Question: What will happen if an NP-complete language can be decided in polynomial time?

Properties of NP-Complete

Answer: Every language in NP can be decided in polynomial time (why??)

- Naturally, a NP-complete language is the “most difficult” language in NP
- In other words, we have...

Theorem: Suppose B is NP-complete. Then, B is in P if and only if $P = NP$

Cook-Levin Theorem

Recall that Cook-Levin Theorem is the following:

Theorem: SAT is P if and only if P = NP

We have not given its proof yet. To prove this, it is equivalent if we prove:

Theorem: SAT is NP-complete

Proof of Cook-Levin

- To prove SAT is NP-complete, we need to do two things:
 1. Show SAT is in NP
 2. Show every other language in NP is polynomial time reducible to SAT

Proof of 1: Simple

Can you give a DTM verifier proof?

Can you give an NTM decider proof?

Proof of 2: Harder...

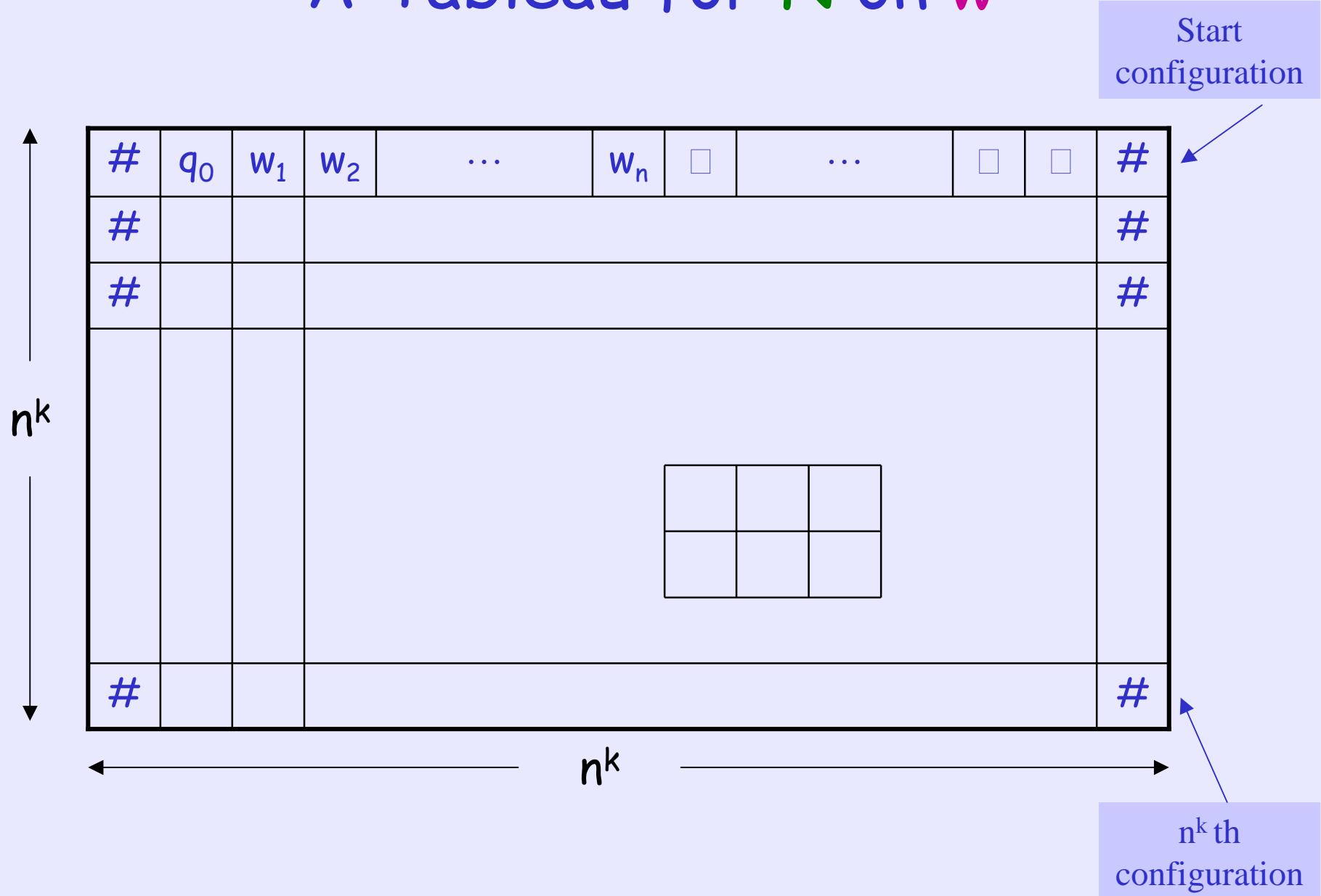
Proof of Part 2 (Idea)

- Idea: We construct a polynomial-time reduction for each A in NP to SAT
- First, let N be an NTM that decides A
- The reduction of A takes a string w and gives a Boolean formula F such that
 N accepts $w \Leftrightarrow F$ is satisfiable
- In particular, we choose (a long and strange) F such that its satisfying assignment corresponds to the (accepting) computation for N to accept w

Proof of Part 2 (Details)

- Let N be an NTM that decides A .
- Let n^k be the running time of N on input of length n , with some constant k .
- We define a **tableau** for N on input w to be an n^k by n^k table that represents a branch of computation of N on w
 - Each row stores a configuration in the branch of computation
- For instance, (see next slide)

A Tableau for N on w



More on Tableau

- For convenience, we assume each configuration starts and ends with #
- The 1st row is the starting configuration, and each row follows from the previous row legally
- A tableau is **accepting** if any row of the tableau is an accepting configuration
 - Thus, every accepting tableau corresponds to an accepting computation

Proof of Cook-Levin (cont)

- So, deciding whether N accepts w is equivalent to deciding whether an accepting tableau for N on w exists
- Our task now is to find a formula F that can check if an accepting tableau exists ...
- Let us try a formula F that contains a variable $x_{i,j,s}$ for each cell (i, j) in the tableau, and each s in $C = Q \cup \Gamma \cup \{\#\}$,
 - Later, we hope $x_{i,j,s} = 1 \Leftrightarrow$ cell (i,j) stores symbol s

Defining the Formula F

- Let us be more ambitious: we hope that when F is satisfiable, the satisfying assignment of F can tell us a valid and accepting tableau
- So, we want to ensure that the satisfying assignment (when F is satisfiable) guarantees:
 1. Each cell is occupied by exact 1 symbol
 2. The tableau has accepting configuration
 3. Each row is correct

Proof of Cook-Levin (cont)

- In particular, we will use sub-formula to represent the above three cases, so that these sub-formula is satisfiable if the corresponding three cases are correct
- The final F is obtained by "And"-ing all these formula, so that if F is satisfiable, all three cases must be correct

Each Cell has only 1 symbol

- The sub-formula $f_{i,j,1}$ ensures cell (i,j) contains at least one symbol:

$$f_{i,j,1} = \bigvee_{s \in C} x_{i,j,s}$$

- The sub-formula $f_{i,j,2}$ ensures cell (i,j) contains at most one symbol:

$$f_{i,j,2} = \bigwedge_{s,t \in C, s \neq t} ((\neg x_{i,j,s}) \vee (\neg x_{i,j,t}))$$

Thus, $f_{i,j,1} \wedge f_{i,j,2}$ will ensure cell (i,j) has exactly one symbol, if F is satisfiable

Accepting Configuration

The following sub-formula ensures the tableau has an accepting configuration if F is satisfiable:

$$f_{\text{accept}} = \bigvee_{i,j} x_{i,j,q_{\text{accept}}}$$

Row is Legal

To ensure starting row is correct, we use the following sub-formula:

$$f_{\text{start}} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \dots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\square} \wedge \dots \wedge x_{1,n^{k-1},\square} \wedge x_{1,n^k,\#}$$

To ensure the remaining rows are correct, we first define the concept of a **window** and **legal window** inside the tableau: (next slide)

Row is Legal (2)

- A **window** at (i, j) refers to the 2×3 cells of (i, j) , $(i, j+1)$, $(i, j+2)$, $(i+1, j)$, $(i+1, j+1)$, and $(i+1, j+2)$
- A **legal window** is a window that does not violate the actions specified by the **N's** transition function, assuming the configuration of each row follows legally from the configuration in the row above

Row is Legal (3)

E.g.,

a	q_1	b
q_2	a	c

This window is legal if there is
a transition $\delta(q_1, b) = (q_2, c, L)$

a	q_1	b
a	a	q_2

This window is legal if there is
a transition $\delta(q_1, b) = (q_2, a, R)$

a	a	q_1
a	a	b

This window is legal if there is
a transition $\delta(q_1, c) = (q_2, b, R)$
for some c and q_2

Row is Legal (4)

E.g.,

#	a	b
#	a	b

This window is also legal

a	b	a
a	b	q ₂

This window is legal if there is a transition $\delta(q_1, b) = (q_2, c, L)$ for some q_1 , b , and c

a	a	a
b	a	a

This window is legal if there is a transition $\delta(q_1, a) = (q_2, b, L)$ for some q_1 and q_2

Row is Legal (5)

E.g.,

a	b	b
a	a	b

a	q_1	b
q_2	a	q_2

All these windows cannot be legal, why?

a	q_1	a
q_2	c	b

Row is Legal (6)

- Note the the window containing the state symbol in the center top cell guarantees that the corresponding three lower cells are updated consistently with the transition function
- So, if a row stores a configuration c , and if all windows in that row are legal, then the row below it will store a configuration the follows legally from c

Row is Legal (7)

- Based on the legal window concept, the sub-formula f_{move} ensures that each row are following correctly:

$$f_{move} = \bigwedge_{1 \leq i, j \leq n^k - 2} (\text{window at } (i, j) \text{ is legal})$$

where "window at (i, j) is legal" is equal to:

$$\bigvee_{a_1, a_2, \dots, a_6 \text{ is a legal window}} (x_{i, j, a_1} \wedge x_{i, j+1, a_2} \wedge x_{i, j+2, a_3} \wedge x_{i+1, j, a_4} \wedge x_{i+1, j+1, a_5} \wedge x_{i+1, j+2, a_6})$$

Proof of Cook-Levin (cont)

Thus, if

$$F = (\bigwedge_{i,j} (f_{i,j,1} \wedge f_{i,j,2})) \wedge f_{\text{accept}} \wedge f_{\text{start}} \wedge f_{\text{move}}$$

then F is satisfiable implies that its satisfying assignment represents an accepting tableau $\rightarrow N$ has an accepting computation on input w $\rightarrow N$ accepts w

Conversely, if N accepts w , there must be an accepting computation, and F has a satisfying assignment $\rightarrow F$ is satisfiable

Proof of Cook-Levin (cont)

- In summary, for any w , we have found a Boolean formula F such that
 $N \text{ accepts } w \Leftrightarrow F \text{ is satisfiable}$
- That is, the construction of F gives a reduction from deciding a language in NP to deciding whether a formula is in SAT
- To show SAT is NP-complete, it remains to show that the construction of F is done in polynomial time (in terms of the length of the input w)

Proof of Cook-Levin (cont)

Given w of length n ,

- f_{start} can be constructed in $O(n^k)$ time
- sub-formula $\bigwedge_{i,j} (f_{i,j,1} \wedge f_{i,j,2})$, f_{accept} , f_{move}

can be constructed in $O(n^{2k})$ time (why??)

→ Time to construct F = polynomial time

- Thus, any language in NP is polynomial-time reducible to SAT and SAT is in NP
→ SAT is NP-complete

Next Time

- More NP-complete problems