RAJALAKSHMI ENGINEERING COLLEGE

THEORY OF COMPUTATION

UNIT- I

AUTOMATA

Why Study Theory of Computation

- Computational devices appear all over the place – besides the computers, several products have computational devices inside them: TV, vending machines, mobile phones, . . .
- Though models and forms of computation may differ, essential idea underlying them remain the same.

  Example: vending machines may be despensing soft drinks, snacks, train tickets, . . .
- Theory of computation helps you develop an understanding of what computation is, what are its limits, what can be done and what cannot be done, how much resources (time, memory, . . .) are needed to do a job, etc.
- We will mostly focus on acceptance of languages: automata, grammars, expressions, etc.

Formal Proofs

If you studied geometry by traditional methods, you would have seen how one proves validity of a statement by using a chain of arguments.

- Deductive proofs:

  Example: if $x \geq 4$, then $2^x \geq x^2$.

- Modus Ponens:

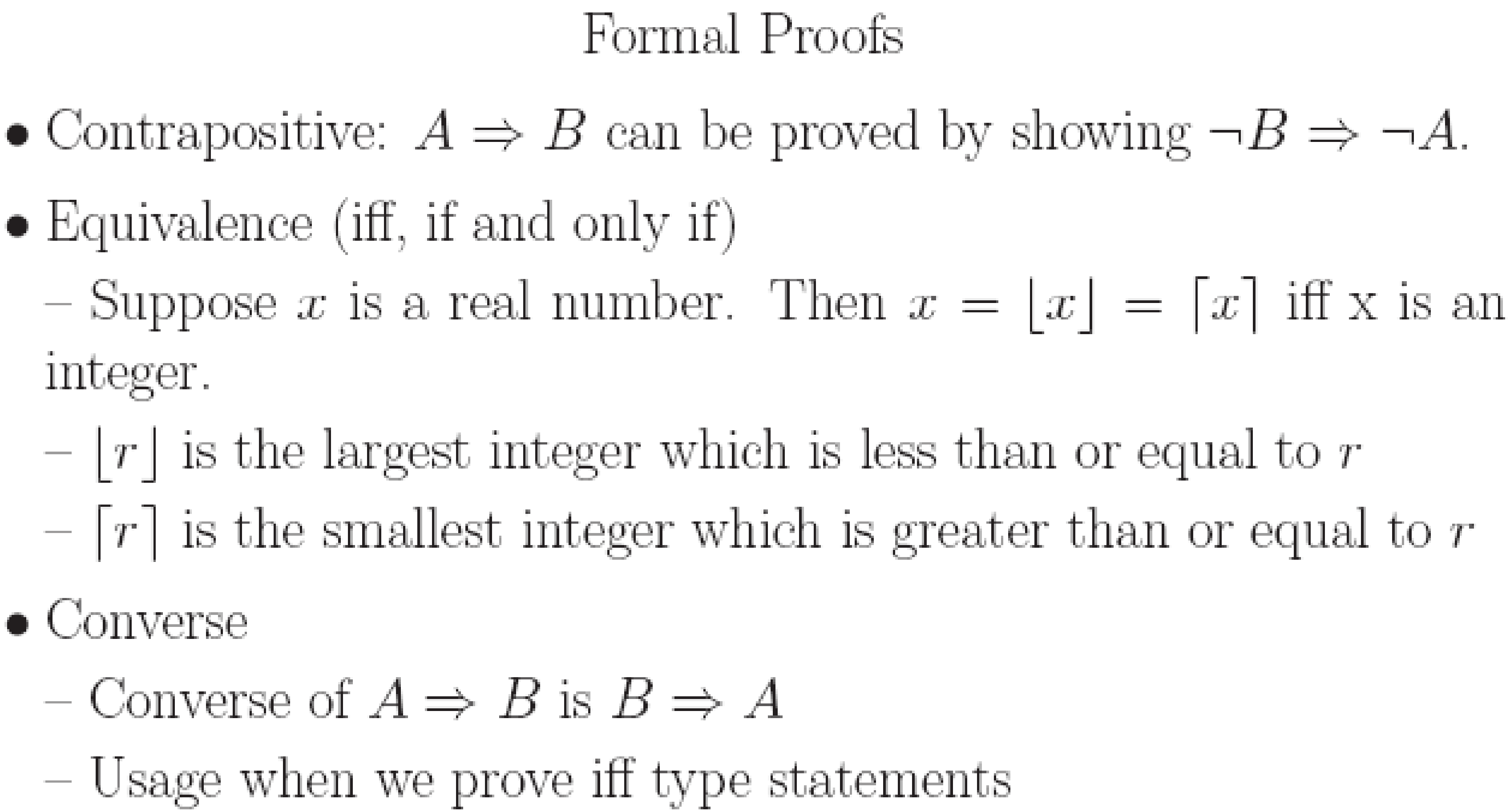
  $A \Rightarrow B$, A, then we can conclude B.

- Proof by Contradiction.

  Example: Suppose U, the universal set, is infinite. Then for any set A, either A is infinite of $\overline{A}$ is infinite (where $\overline{A} = U - A$).

- Counterexample.

  Example: All primes are odd (False: counterexample is 2).



Formal Proofs

- Contrapositive: $A \Rightarrow B$ can be proved by showing $\neg B \Rightarrow \neg A$.
- Equivalence (iff, if and only if)
  - Suppose x is a real number. Then $x = \lfloor x \rfloor = \lceil x \rceil$ iff x is an integer.
  - $\lfloor r \rfloor$ is the largest integer which is less than or equal to r
  - $\lceil r \rceil$ is the smallest integer which is greater than or equal to r
- Converse
  - Converse of $A \Rightarrow B$ is $B \Rightarrow A$
  - Usage when we prove iff type statements

Formal Proofs

- Inductive Proofs.
  - $1 + 2 + 3 + \ldots + n = n(n+1)/2.$

Formal Proofs

- Inductive Proofs

  Basic form: prove

  - base case ($n = 1$)
  - induction step: If statement holds for $n = k$, then it holds for $n = k + 1$.

  Course of values induction: prove

  - base case ($n = 1$)
  - induction step: If statement holds for smaller values of n ($n = 1$ to k), then it holds for $n = k + 1$.

  Structural Induction

  Example: If a claim holds for all trees of height at most k, then the claim holds for all trees of height $k + 1$.

Formal Proofs

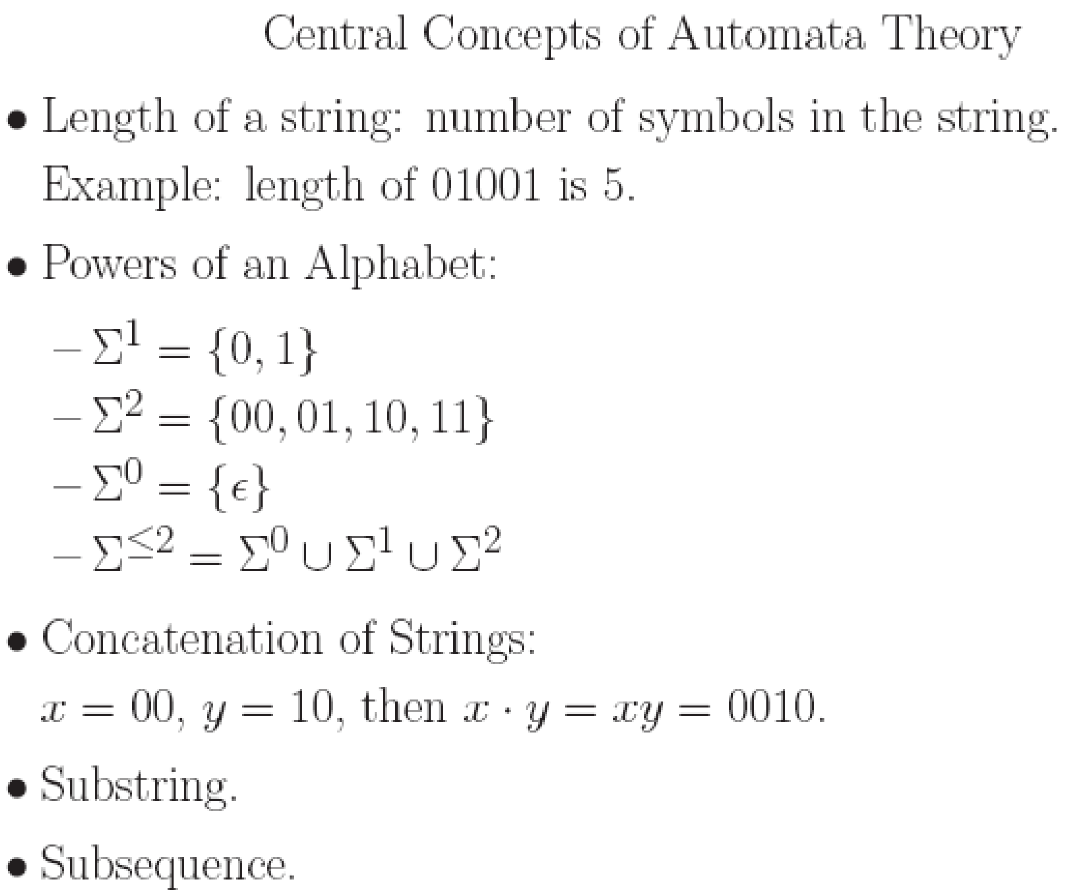
Mutual Induction: showing several claims to be true simultaneously.

Quantifiers: $(\exists x)[P(x)]$ and $(\forall x)[P(x)]$
Sometimes also use $(\overset{\infty}{\exists}\ x)[P(x)]$, $(\exists ! x)[P(x)]$.

Central Concepts of Automata Theory

- Alphabets
  - Alphabet is a finite non-empty set of symbols. We usually use the symbol Σ to denote the alphabet.
  - $\{0, 1\}$.
  - $\{A, B, \ldots, Z\}$
  - $\{0, A, s\}$
- Strings:

  Finite sequence of symbols chosen from a given alphabet. For example, $010001, ACB, 0sAss0$.
- Empty String: ϵ. Sometimes Λ is also used.



Central Concepts of Automata Theory

- Length of a string: number of symbols in the string.

  Example: length of 01001 is 5.

- Powers of an Alphabet:
  - $\Sigma^1 = \{0, 1\}$
  - $\Sigma^2 = \{00, 01, 10, 11\}$
  - $\Sigma^0 = \{\epsilon\}$
  - $\Sigma^{\leq 2} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2$

- Concatenation of Strings:

  $x = 00$, $y = 10$, then $x \cdot y = xy = 0010$.

- Substring.

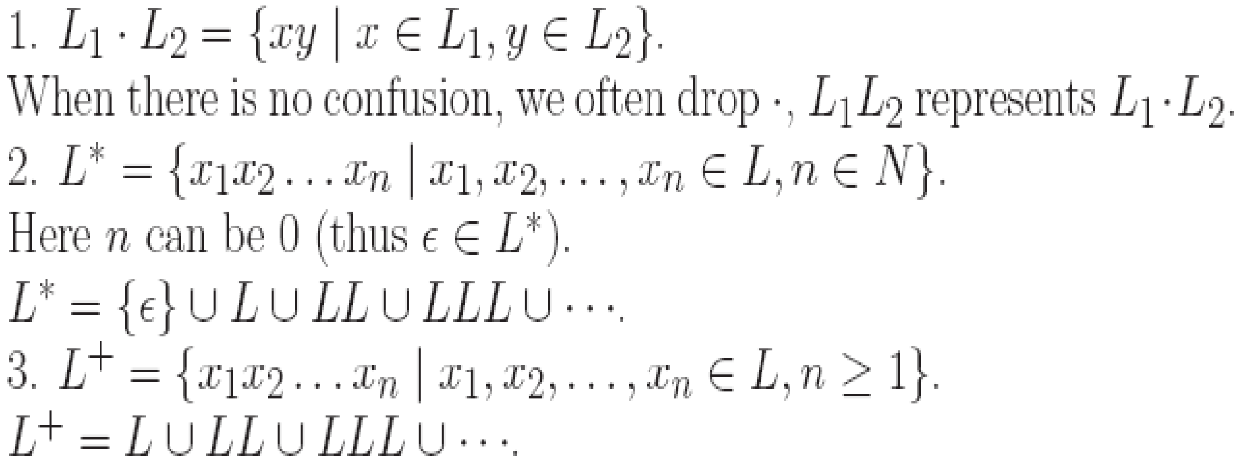
- Subsequence.

- Languages: A set of strings (over an alphabet).

  $L = \{00, 11, 01, 110\}$.

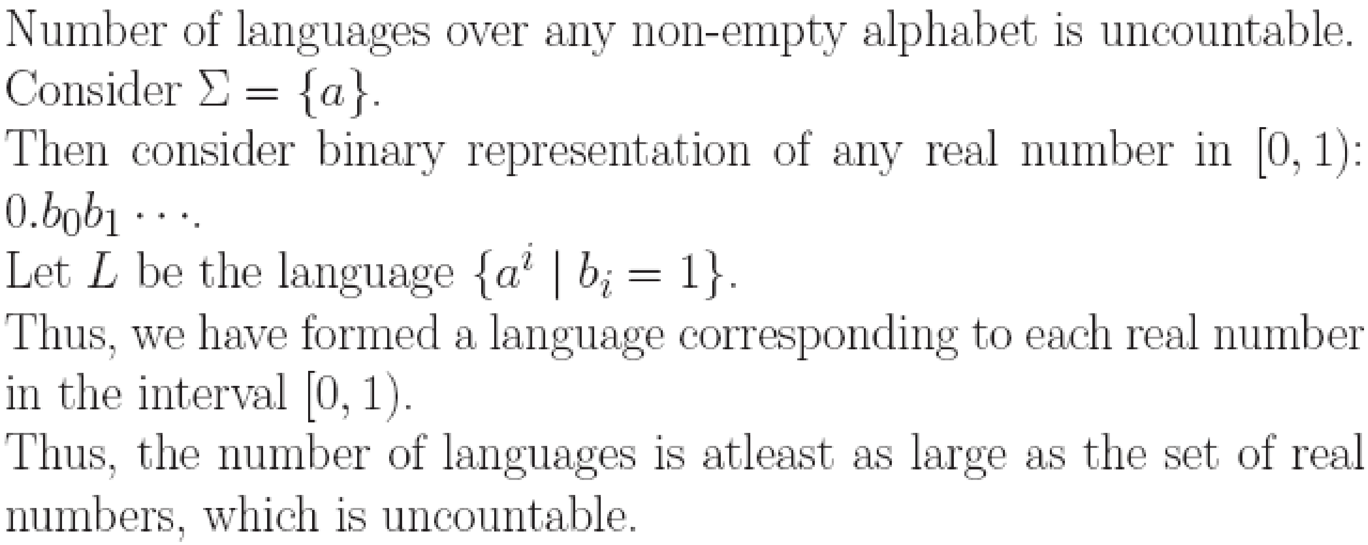
  $L = \emptyset$.

  $L = \{x \mid x \text{ is a binary representation of a prime number }\}$.



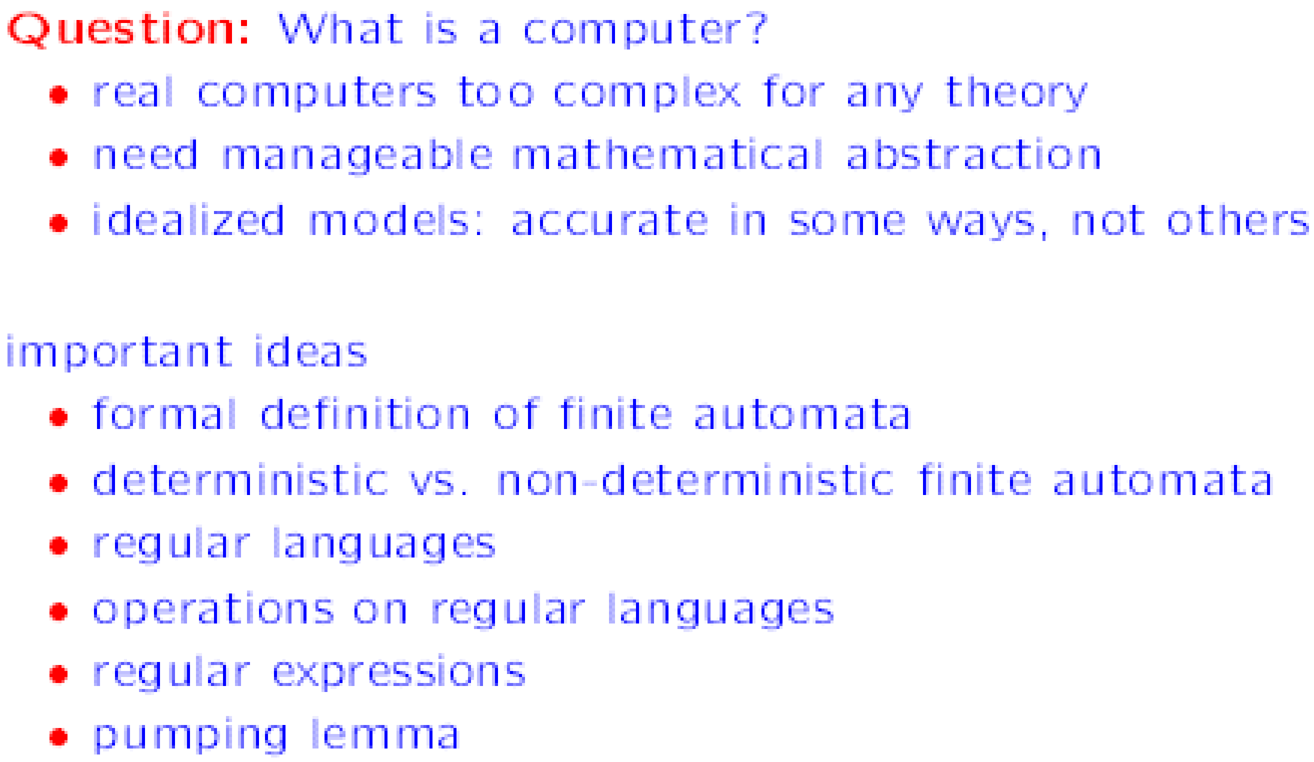
1. $L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$.
When there is no confusion, we often drop $\cdot$, L_1L_2 represents $L_1 \cdot L_2$.
2. $L^* = \{x_1x_2 \ldots x_n \mid x_1, x_2, \ldots, x_n \in L, n \in N\}$.
Here n can be 0 (thus $\epsilon \in L^*$).
$L^* = \{\epsilon\} \cup L \cup LL \cup LLL \cup \cdots$.
3. $L^+ = \{x_1x_2 \ldots x_n \mid x_1, x_2, \ldots, x_n \in L, n \geq 1\}$.
$L^+ = L \cup LL \cup LLL \cup \cdots$.

Number of strings over any fixed finite alphabet Σ is countable:
We do the proof for $\Sigma = \{0, 1\}$.
The same idea can be generalized to prove the result for larger alphabets.
For any string x, let $m(x)$ = value of $1x$ as binary number.
Thus we are mapping string ϵ to 1, string 0 to 2,
string 1 to 3, string 00 to 4 and so on.
Thus, number of strings is countable.



Number of languages over any non-empty alphabet is uncountable.
Consider $\Sigma = \{a\}$.
Then consider binary representation of any real number in $[0, 1)$: $0.b_0b_1\cdots$.
Let L be the language $\{a^i \mid b_i = 1\}$.
Thus, we have formed a language corresponding to each real number in the interval $[0, 1)$.
Thus, the number of languages is atleast as large as the set of real numbers, which is uncountable.

Finite Automata



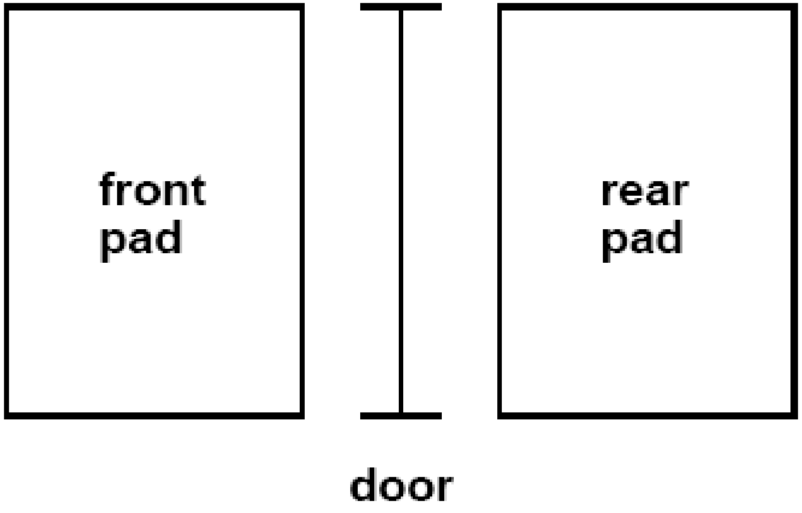
Question: What is a computer?

- real computers too complex for any theory
- need manageable mathematical abstraction
- idealized models: accurate in some ways, not others

important ideas

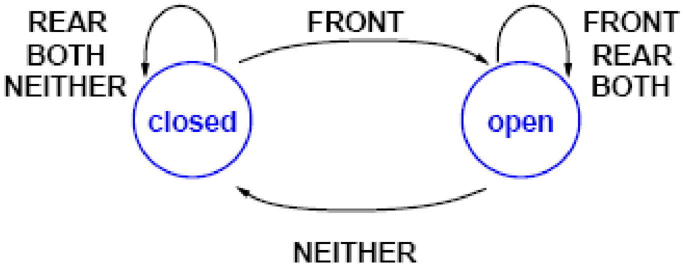
- formal definition of finite automata
- deterministic vs. non-deterministic finite automata
- regular languages
- operations on regular languages
- regular expressions
- pumping lemma

Example: An Automatic Door



- open when person approaches
- hold open until person clears
- don't open when someone standing behind the door

Example: An Automatic Door



- States

  Open

  Closed

- Sensors

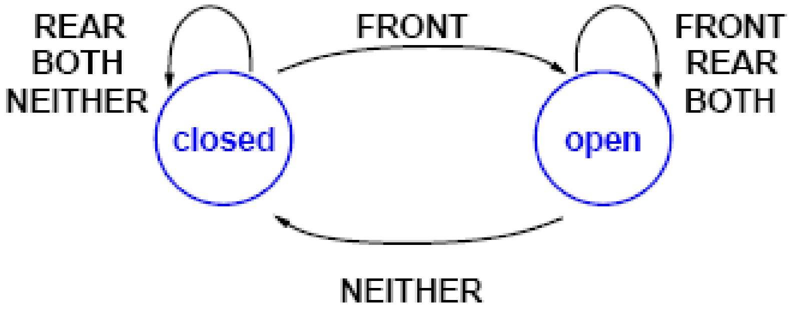
Front – Someone on the Front pad

Rear – Someone on the Rear pad

Both – Someone on both the pads

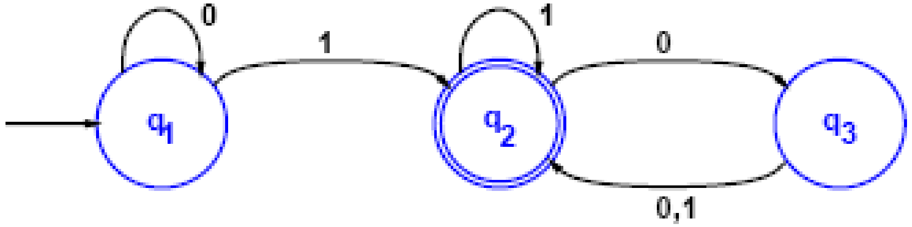
Neither – No one on either pad

Example: An Automatic Door



	NEITHER	FRONT	REAR	BOTH
closed	closed	open	closed	closed
open	closed	open	open	open

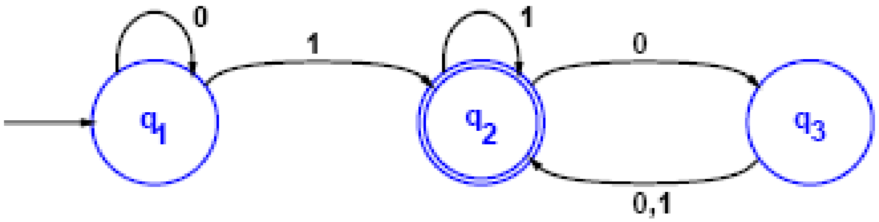
Informal Definition



The machine M_1:

- *states*: q_1, q_2, and q_3.
- *start state*: q_1 (arrow from nowhere).
- *accept state*: q_2 (double circle).
- *state transitions*: arrows.

Informal Definition



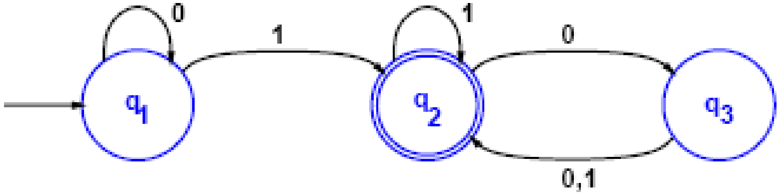
On an input string

- begins in start state q_1
- after reading each symbol, M_1 takes transition with matching label.

After reading last symbol, M_1 produces output:

- *accept* if M_1 is an accepting state.
- *reject* otherwise.

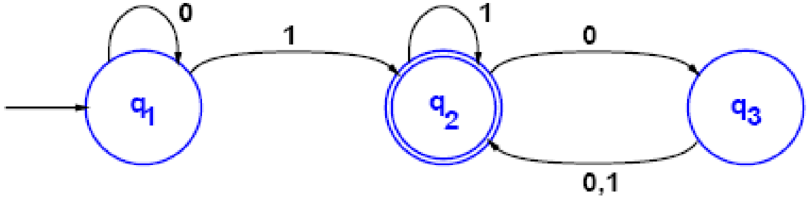
Informal Definition



What happens on input strings

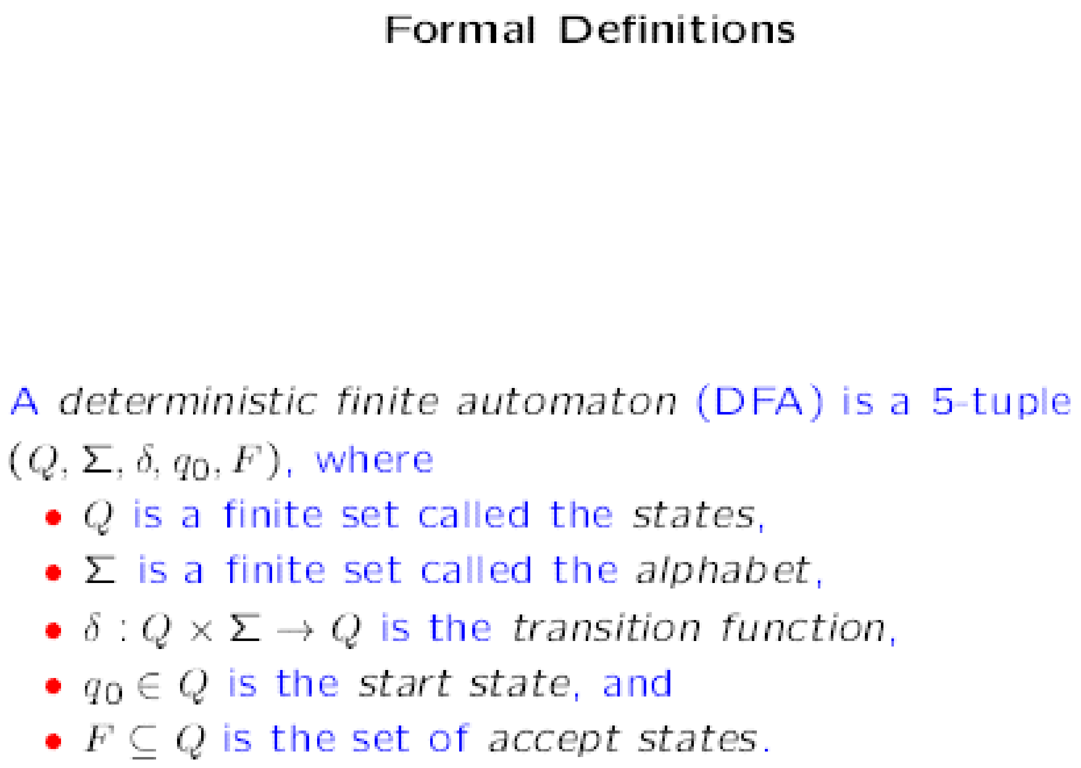
- 1101
- 0010
- 01100

Informal Definition



M_1 accepts

- all input strings that end with a 1
- all input strings that end with an even number of 0's
- no other strings

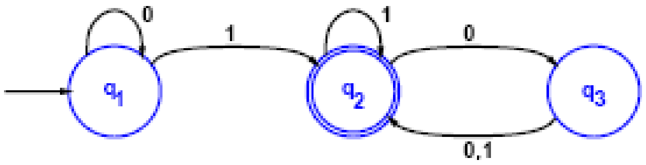


Formal Definitions

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

- Q is a finite set called the *states*,
- Σ is a finite set called the *alphabet*,
- $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *accept states*.

Back to M_1



$M_1 = (Q, \Sigma, \delta, q_1, F)$ where

- $Q = \{q_1, q_2, q_3\}$,
- $\Sigma = \{0, 1\}$,
- δ is

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

- q_1 is the start state, and
- $F = \{q_2\}$.

Languages

Definition The *language of machine* M is the set of strings A that M accepts.

$$L(M) = A$$

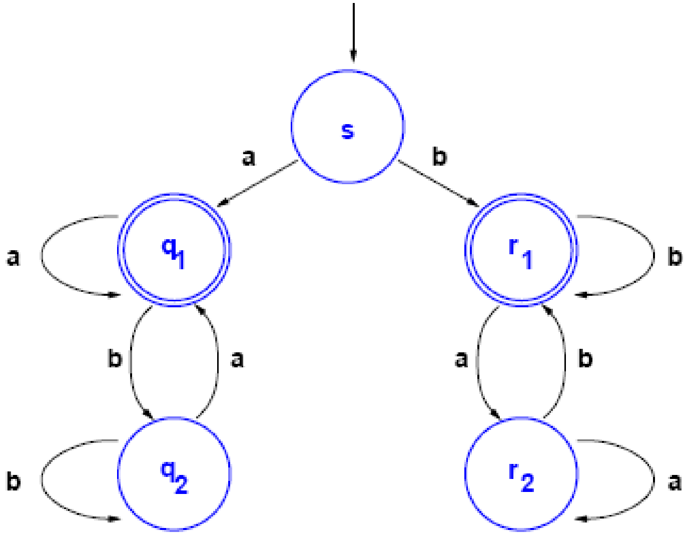
Note that

- M may accept many strings, but
- M accepts only one language

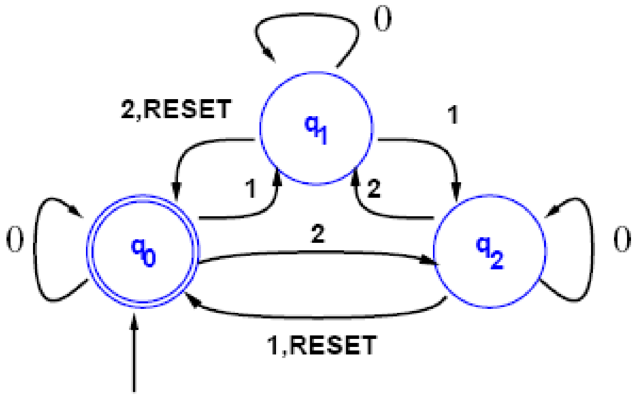
Question: What language does M accept if it accepts no strings?

Definition: A language is *regular* if some deterministic finite automaton accepts it.

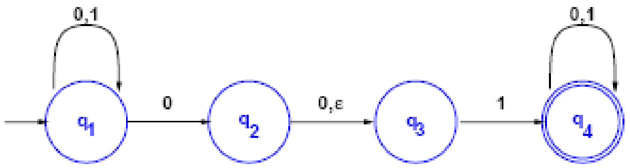
Example



Example



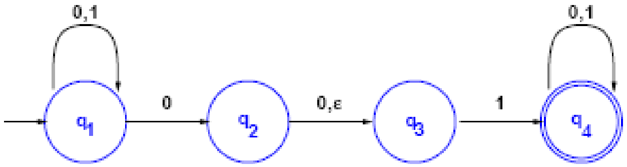
Non-Deterministic Finite Automata



Every DFA is also a non-deterministic finite automaton (NFA).

- a NFA may have more than one transition labeled with a symbol,
- a NFA may have no transitions labeled with a symbol, and
- transitions may be labeled with ε, the empty string.

Non-Deterministic Computation

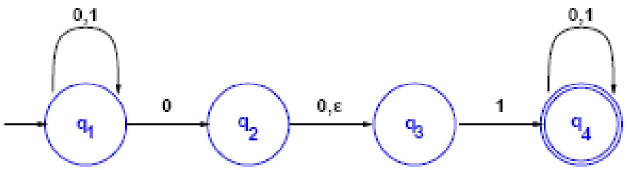


Question: What happens when more than one transition is possible?

- The machine "splits" into multiple copies
- each branch follows one possibility
- together, branches follow *all* possibilities.
- If the input doesn't appear, that branch "dies".
- Automaton accepts if *any* branch accepts.

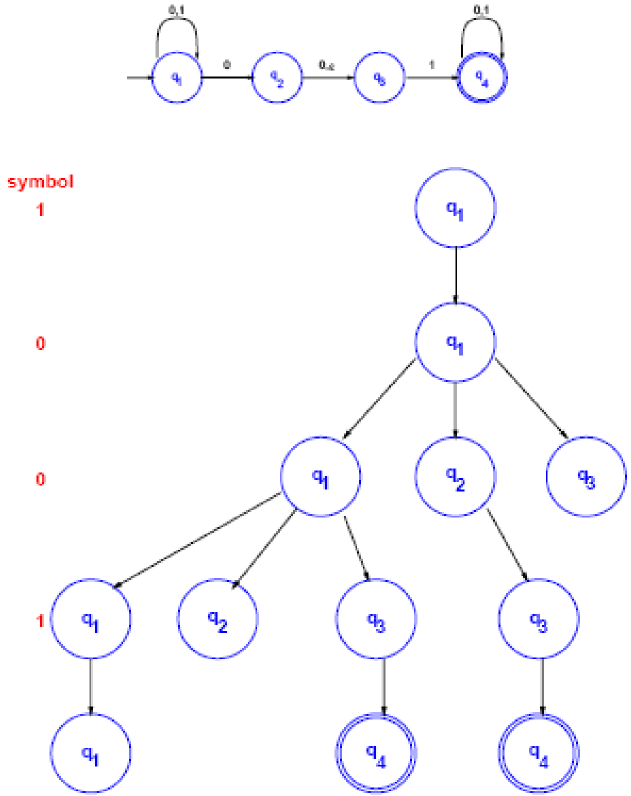
Question: What does an ε transition do?

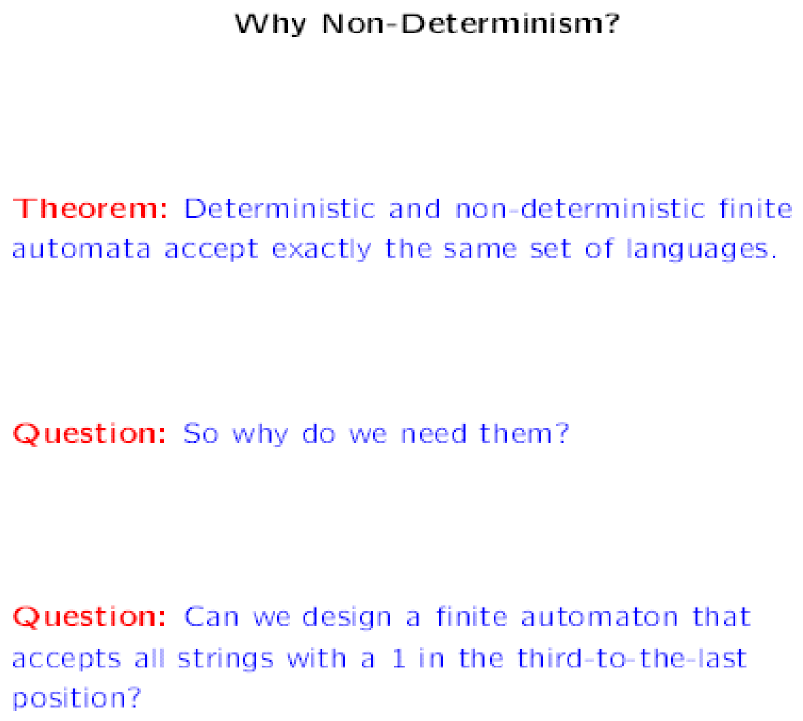
Non-Deterministic Computation



Question: What happens on string 1001?

String 1001


0,1
0
0,ε
1
0,1
q1
q2
q3
q4
symbol
1
0
0
1
q1
q1
q1
q2
q3
q1
q2
q3
q3
q1
q4
q4



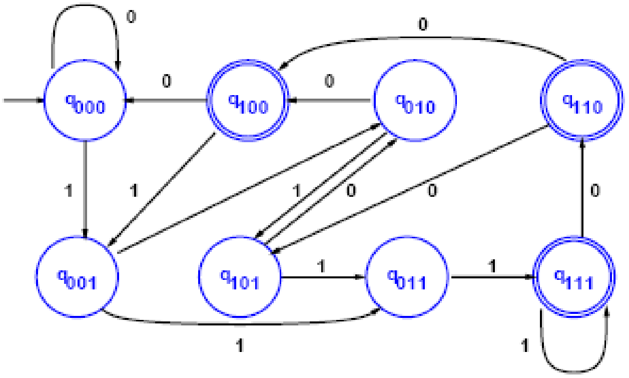
Why Non-Determinism?

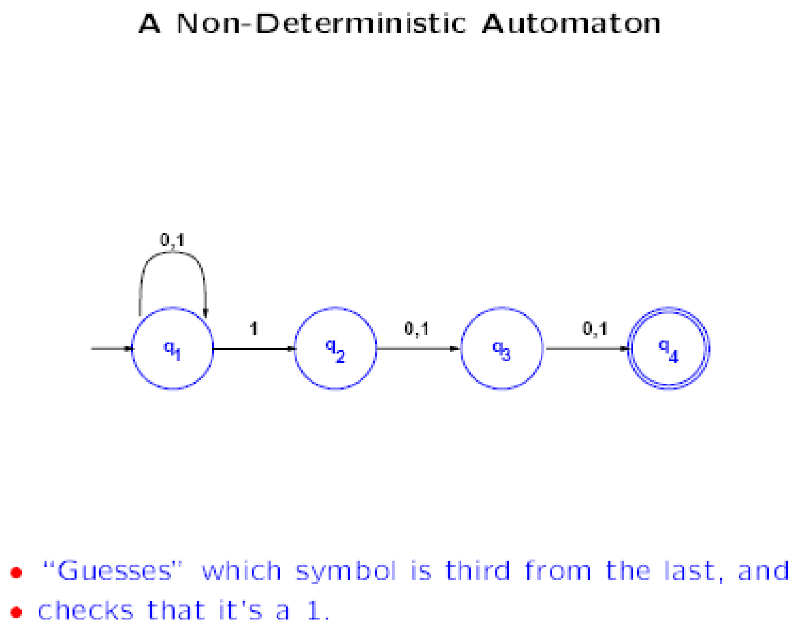
Theorem: Deterministic and non-deterministic finite automata accept exactly the same set of languages.

Question: So why do we need them?

Question: Can we design a finite automaton that accepts all strings with a 1 in the third-to-the-last position?

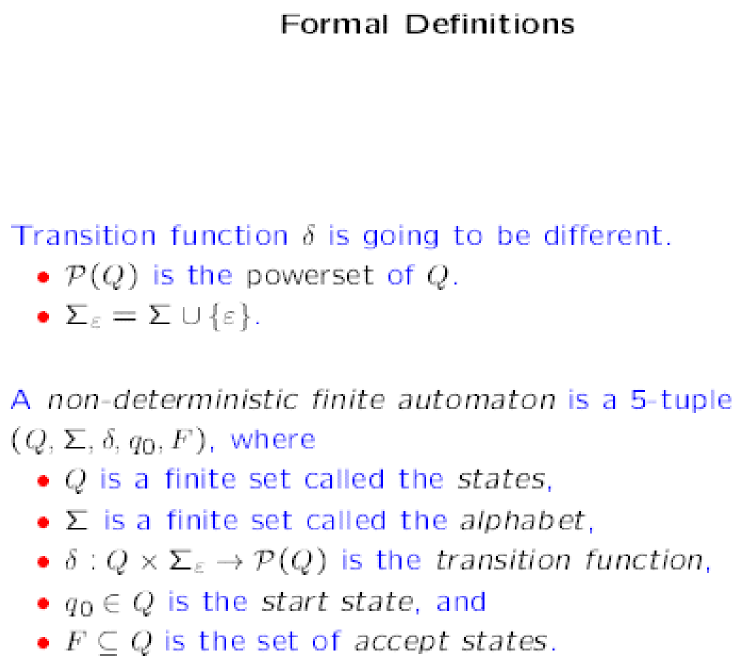
A Deterministic Automaton





A Non-Deterministic Automaton

- "Guesses" which symbol is third from the last, and
- checks that it's a 1.



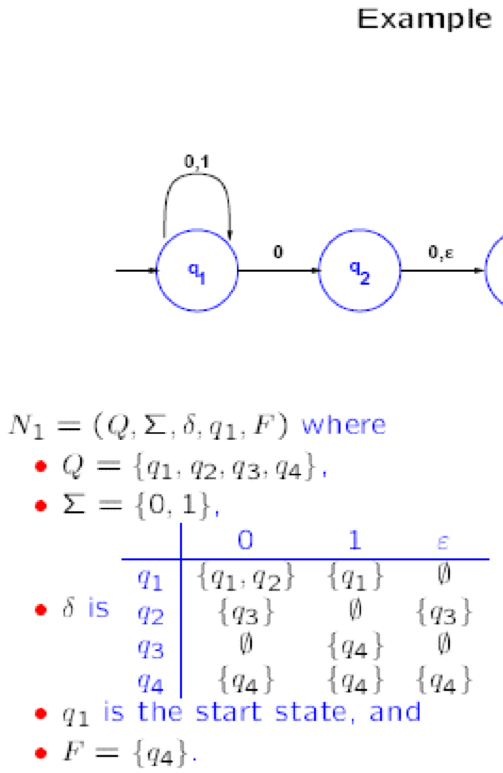
Formal Definitions

Transition function δ is going to be different.

- $\mathcal{P}(Q)$ is the powerset of Q.
- $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.

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

- Q is a finite set called the *states*,
- Σ is a finite set called the *alphabet*,
- $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *accept states*.



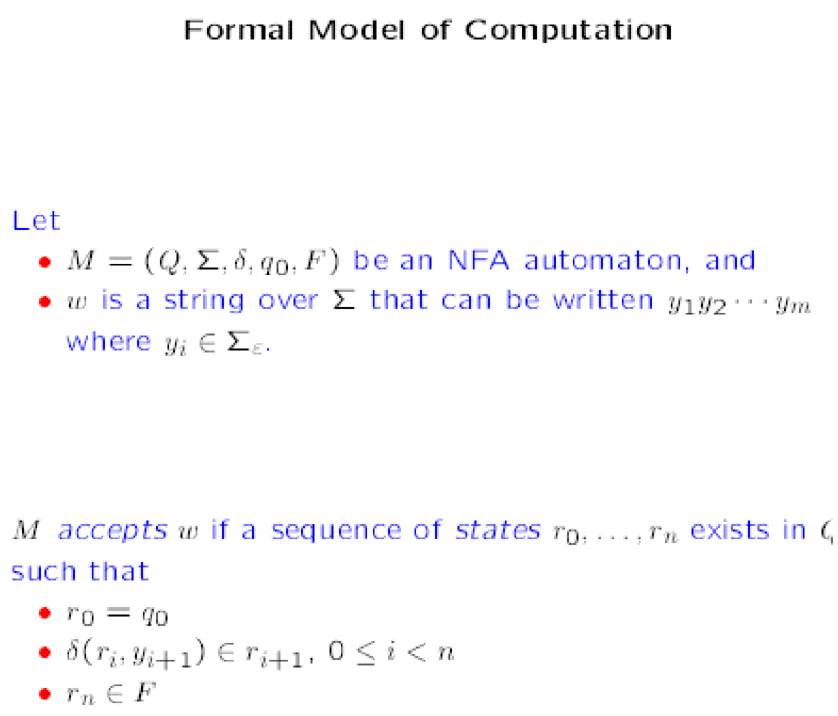
Example

$N_1 = (Q, \Sigma, \delta, q_1, F)$ where

- $Q = \{q_1, q_2, q_3, q_4\}$,
- $\Sigma = \{0, 1\}$,
- δ is

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

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



Formal Model of Computation

Let

- $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA automaton, and
- w is a string over Σ that can be written $y_1 y_2 \cdots y_m$ where $y_i \in \Sigma_\varepsilon$.

M *accepts* w if a sequence of *states* $r_0, \ldots, r_n$ exists in Q such that

- $r_0 = q_0$
- $\delta(r_i, y_{i+1}) \in r_{i+1}$, $0 \leq i < n$
- $r_n \in F$

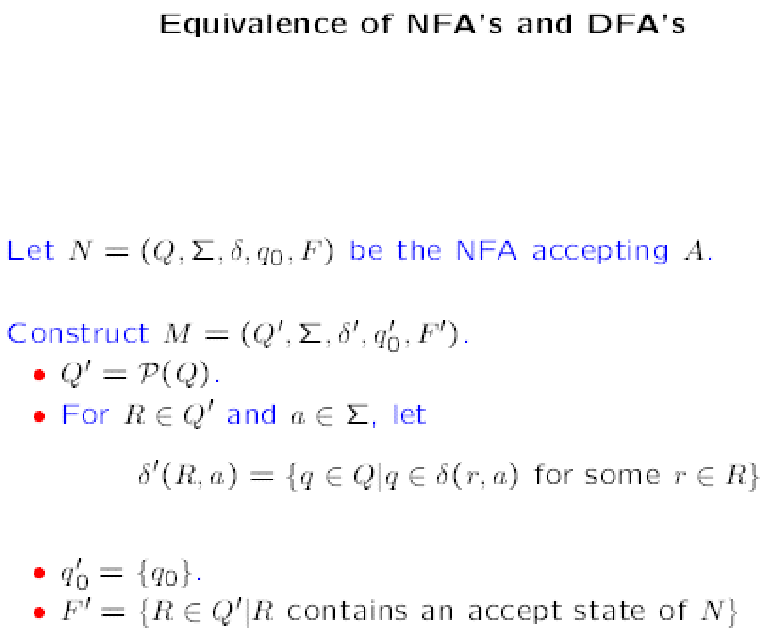
Equivalence of NFA's and DFA's

If a language is accepted by a NFA, then we must construct a DFA that accepts the same language.

Make things easier by ignoring ε transitions (for now).

Have DFA simulate all possible NFA states.

If the NFA has k states, the DFA has 2^k states.



Equivalence of NFA's and DFA's

Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA accepting A.

Construct $M = (Q', \Sigma, \delta', q'_0, F')$.

- $Q' = \mathcal{P}(Q)$.
- For $R \in Q'$ and $a \in \Sigma$, let

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

- $q'_0 = \{q_0\}$.
- $F' = \{R \in Q' | R \text{ contains an accept state of } N\}$

ε-Transitions

For any state R of M, define $E(R)$ to be the collection of states reachable from R by ε transitions only.

$E(R) = \{q | q$ can be reached from R by 0 or more ε trans

Now define transition:

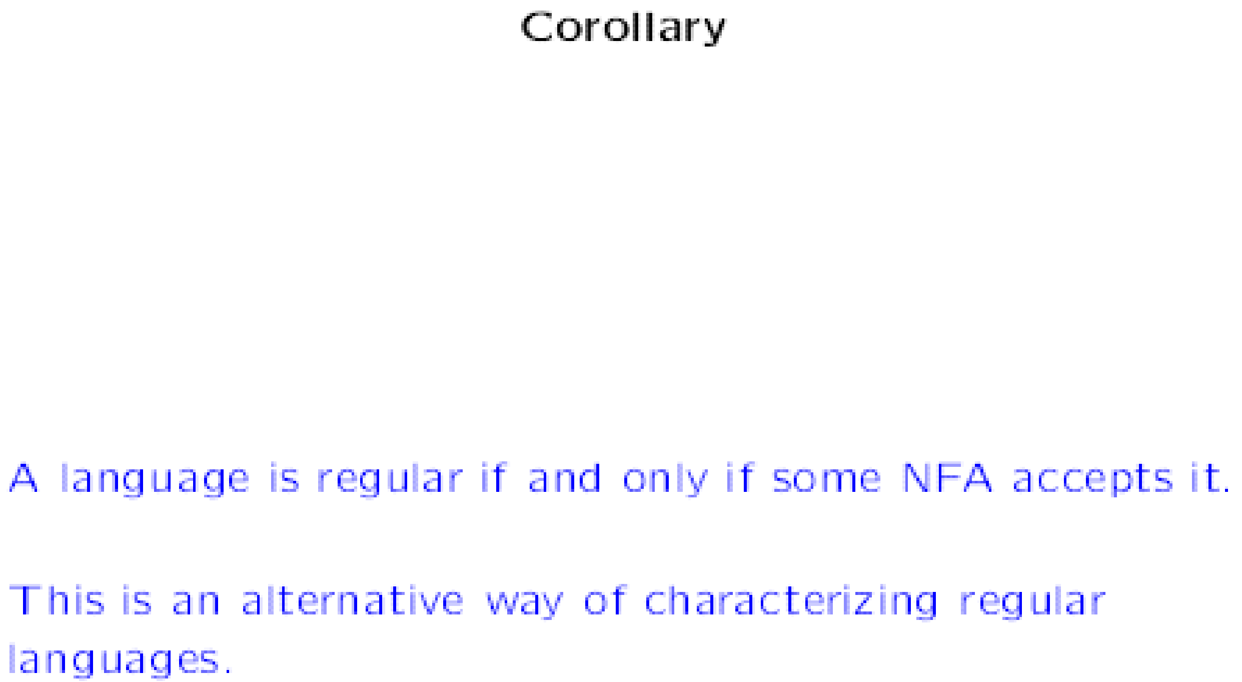
$$\delta'(R, a) = \{q \in Q | q \in E(\delta(r, a)) \text{ for some } r \in R\}$$

Change start state to

$$q'_0 = E(\{q_0\})$$

UNIT - II

REGULAR EXPRESSIONS
AND
LANGUAGES

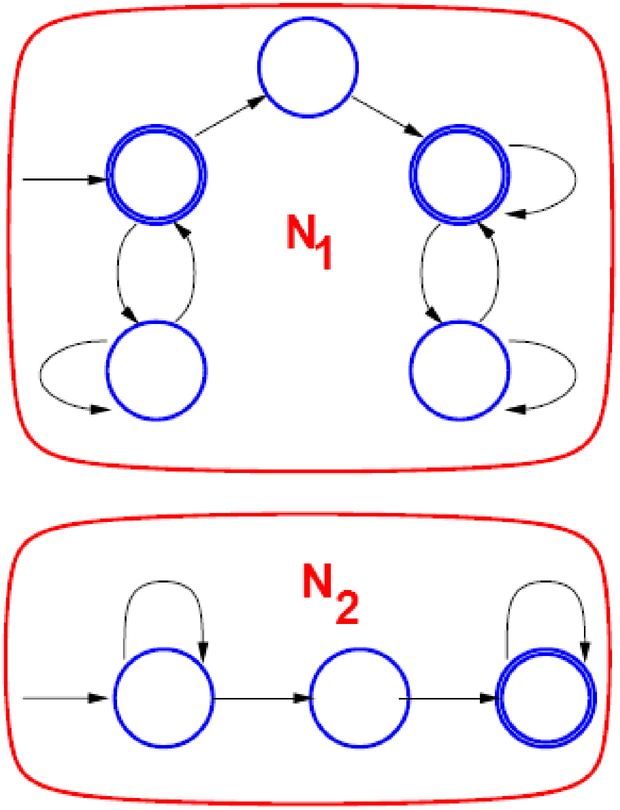


Corollary

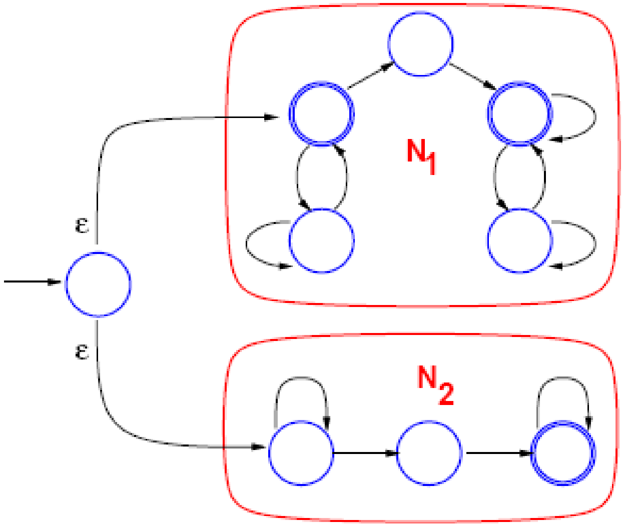
A language is regular if and only if some NFA accepts it.

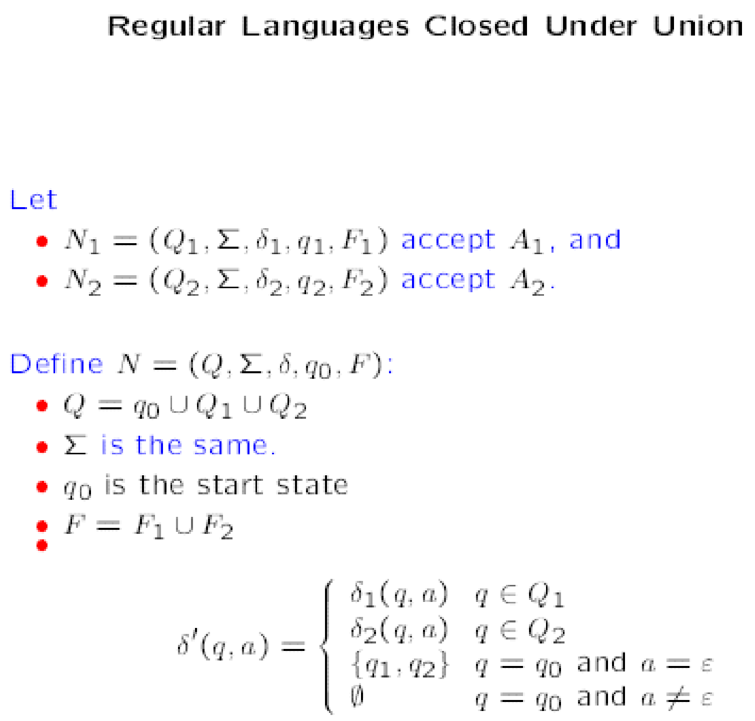
This is an alternative way of characterizing regular languages.

Regular Languages Closed Under Union



Regular Languages Closed Under Union





Regular Languages Closed Under Union

Let

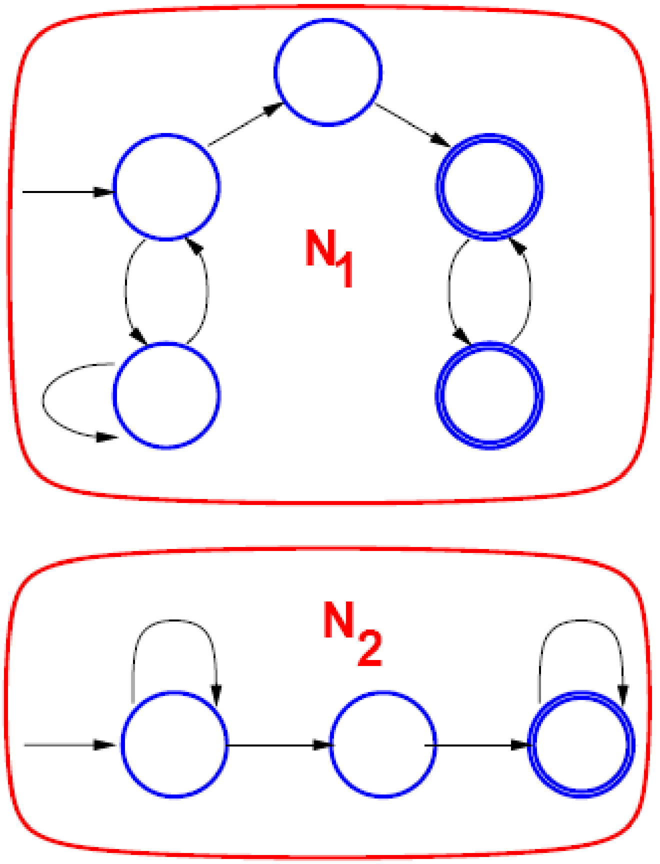
- $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ accept A_1, and
- $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ accept A_2.

Define $N = (Q, \Sigma, \delta, q_0, F)$:

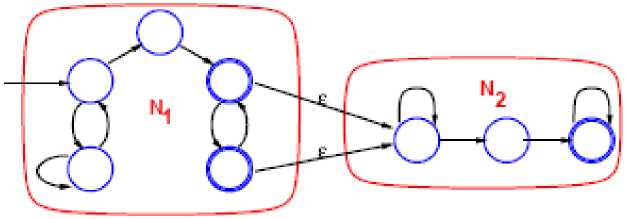
- $Q = q_0 \cup Q_1 \cup Q_2$
- Σ is the same.
- q_0 is the start state
- $F = F_1 \cup F_2$
-

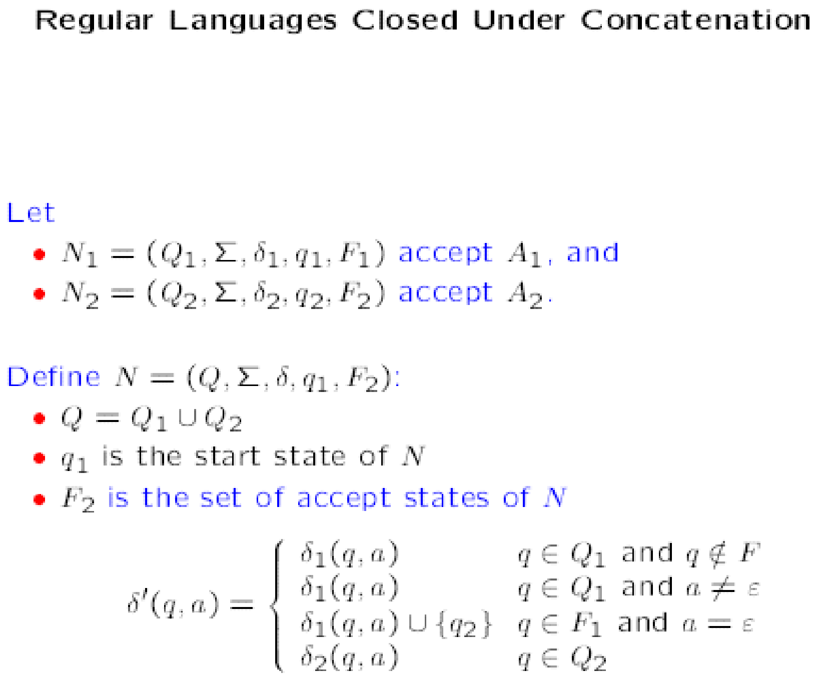
$$\delta'(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon \end{cases}$$

Regular Languages Closed Under Concatenation



Regular Languages Closed Under Concatenation





Regular Languages Closed Under Concatenation

Let

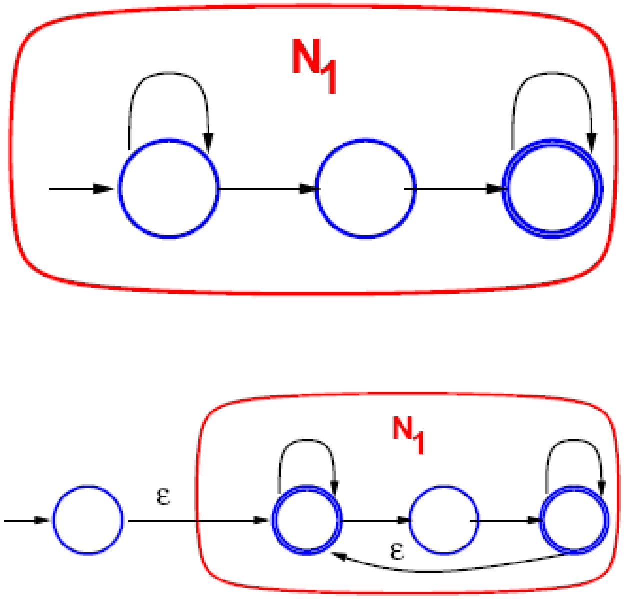
- $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ accept A_1, and
- $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ accept A_2.

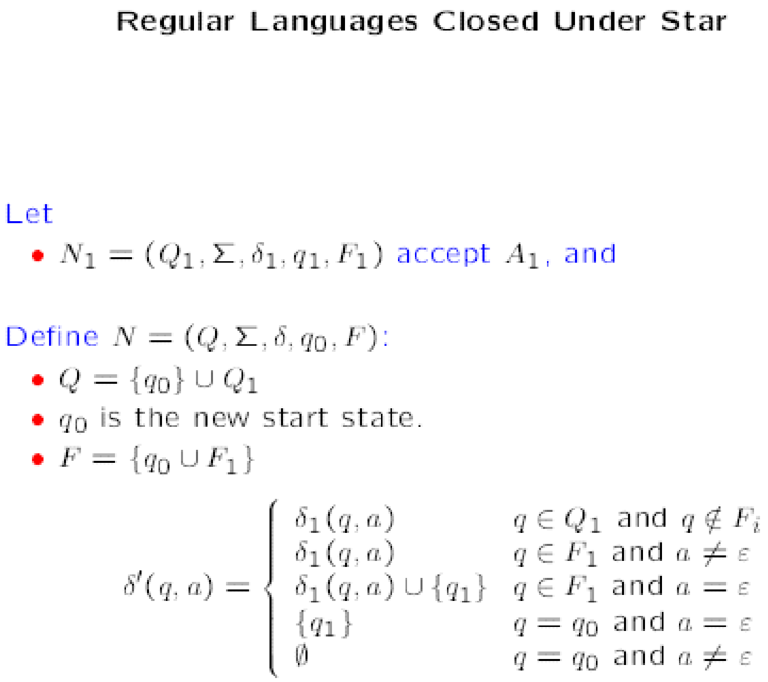
Define $N = (Q, \Sigma, \delta, q_1, F_2)$:

- $Q = Q_1 \cup Q_2$
- q_1 is the start state of N
- F_2 is the set of accept states of N

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

Regular Languages Closed Under Star





Regular Languages Closed Under Star

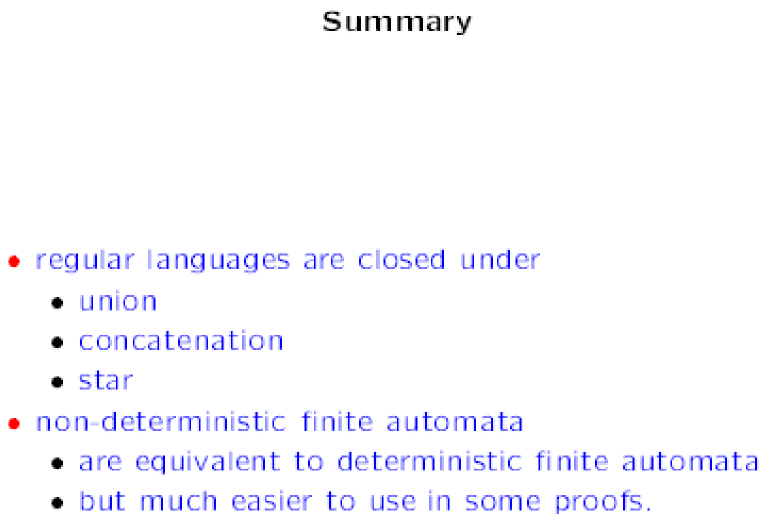
Let

- $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ accept A_1, and

Define $N = (Q, \Sigma, \delta, q_0, F)$:

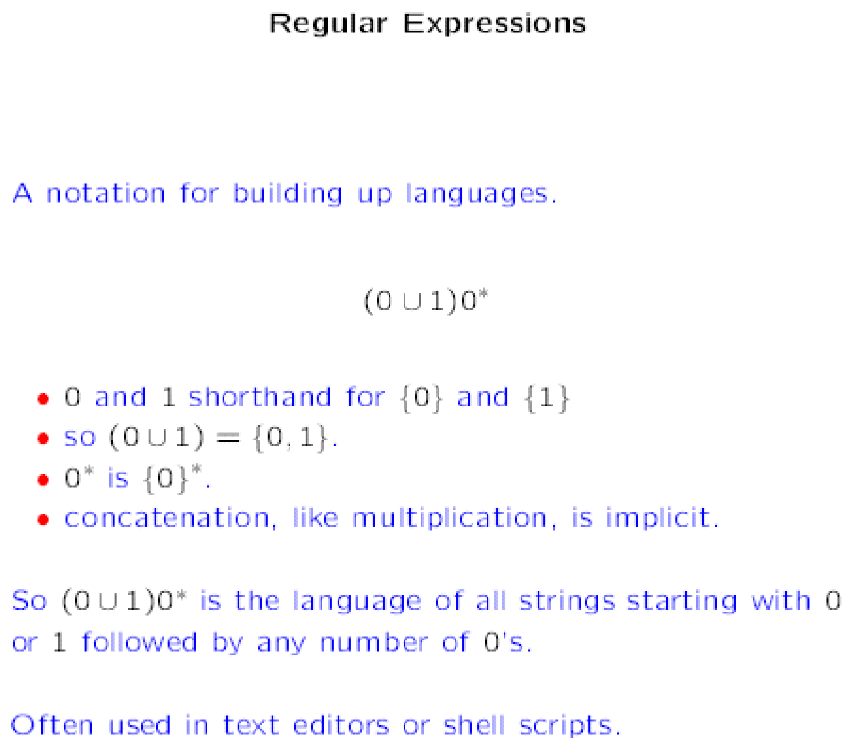
- $Q = \{q_0\} \cup Q_1$
- q_0 is the new start state.
- $F = \{q_0 \cup F_1\}$

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



Summary

- regular languages are closed under
  - union
  - concatenation
  - star
- non-deterministic finite automata
  - are equivalent to deterministic finite automata
  - but much easier to use in some proofs.



Regular Expressions

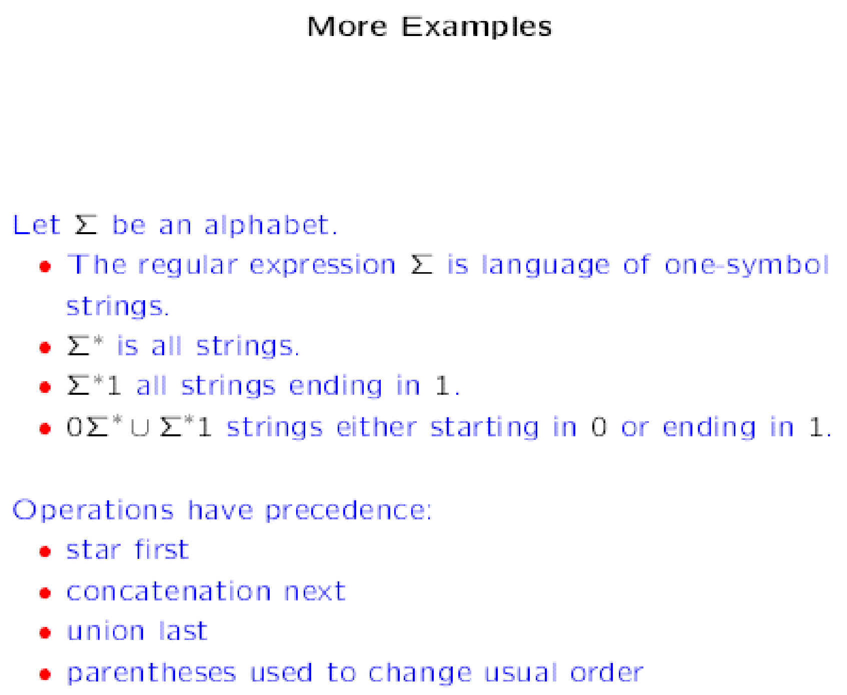
A notation for building up languages.

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

- 0 and 1 shorthand for $\{0\}$ and $\{1\}$
- so $(0 \cup 1) = \{0, 1\}$.
- 0^* is $\{0\}^*$.
- concatenation, like multiplication, is implicit.

So $(0 \cup 1)0^*$ is the language of all strings starting with 0 or 1 followed by any number of 0's.

Often used in text editors or shell scripts.



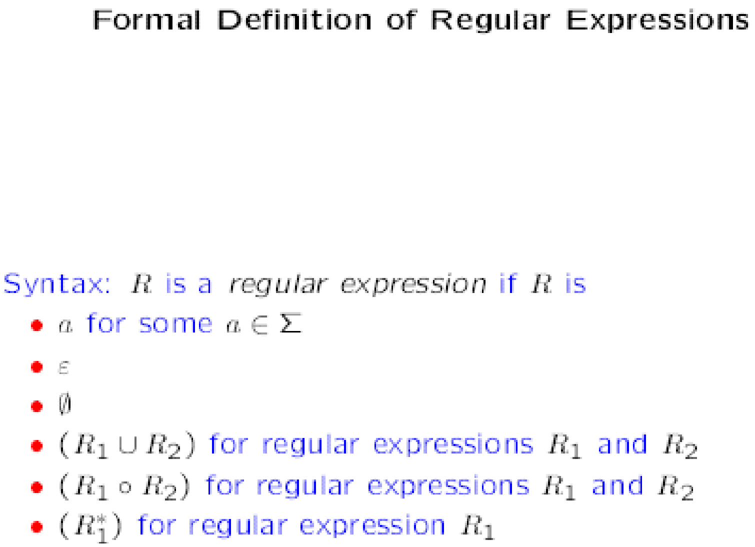
More Examples

Let Σ be an alphabet.

- The regular expression Σ is language of one-symbol strings.
- Σ^* is all strings.
- Σ^*1 all strings ending in 1.
- $0\Sigma^* \cup \Sigma^*1$ strings either starting in 0 or ending in 1.

Operations have precedence:

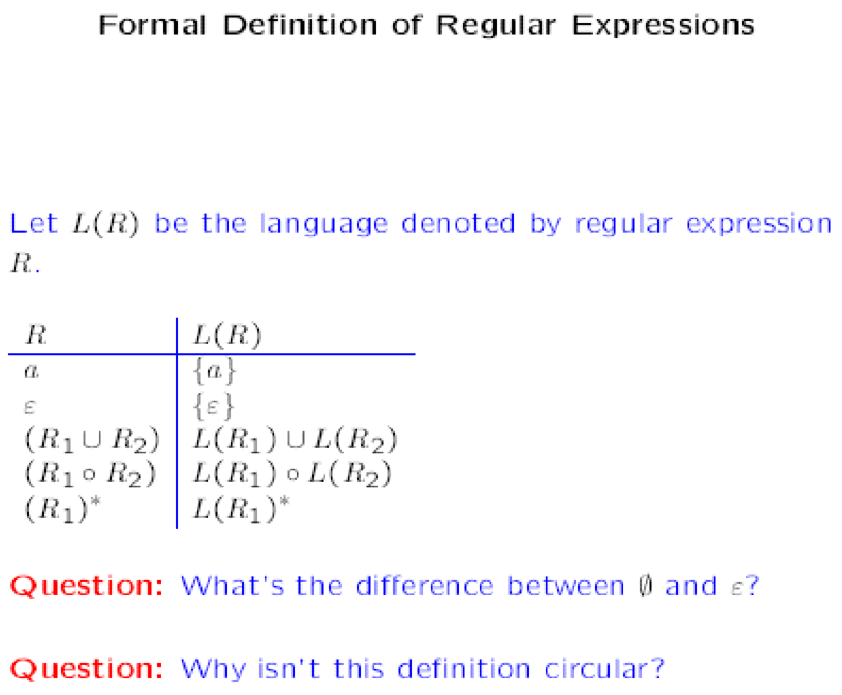
- star first
- concatenation next
- union last
- parentheses used to change usual order



Formal Definition of Regular Expressions

Syntax: R is a *regular expression* if R is

- a for some $a \in \Sigma$
- ε
- $\emptyset$
- $(R_1 \cup R_2)$ for regular expressions R_1 and R_2
- $(R_1 \circ R_2)$ for regular expressions R_1 and R_2
- (R_1^*) for regular expression R_1



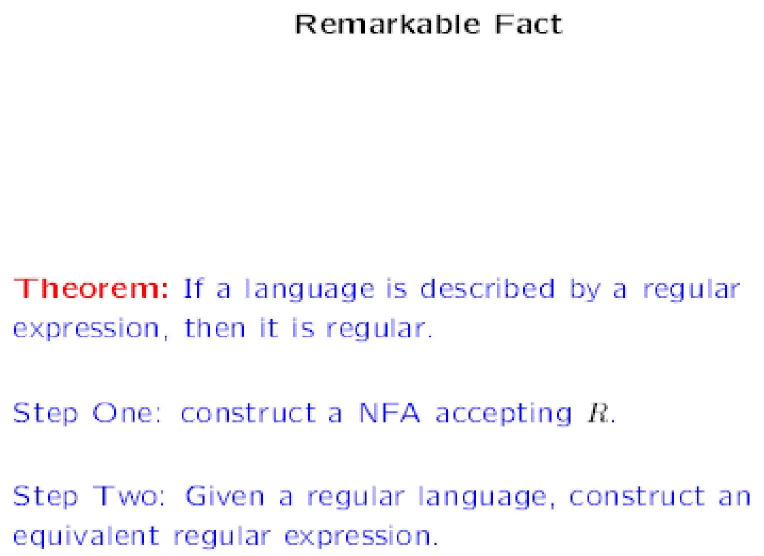
Formal Definition of Regular Expressions

Let $L(R)$ be the language denoted by regular expression R.

R	$L(R)$
a	$\{a\}$
ε	$\{\varepsilon\}$
$(R_1 \cup R_2)$	$L(R_1) \cup L(R_2)$
$(R_1 \circ R_2)$	$L(R_1) \circ L(R_2)$
$(R_1)^*$	$L(R_1)^*$

Question: What's the difference between $\emptyset$ and ε?

Question: Why isn't this definition circular?



Remarkable Fact

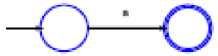
Theorem: If a language is described by a regular expression, then it is regular.

Step One: construct a NFA accepting R.

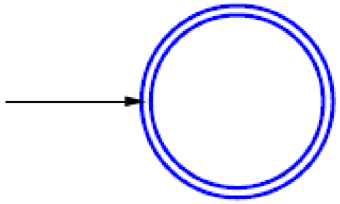
Step Two: Given a regular language, construct an equivalent regular expression.

NFA Accepting Regular Expression

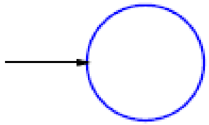
$R = a \in \Sigma$



$R = \varepsilon$

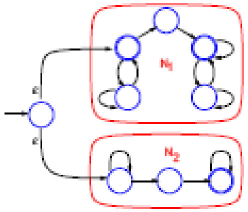


$R = \emptyset$

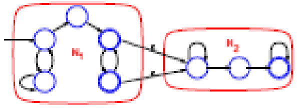


NFA Accepting Regular Expression

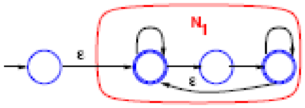
$(R_1 \cup R_2)$:



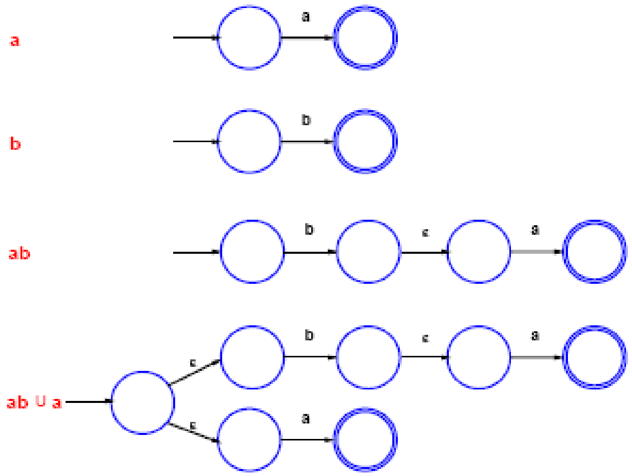
$(R_1 \circ R_2)$:

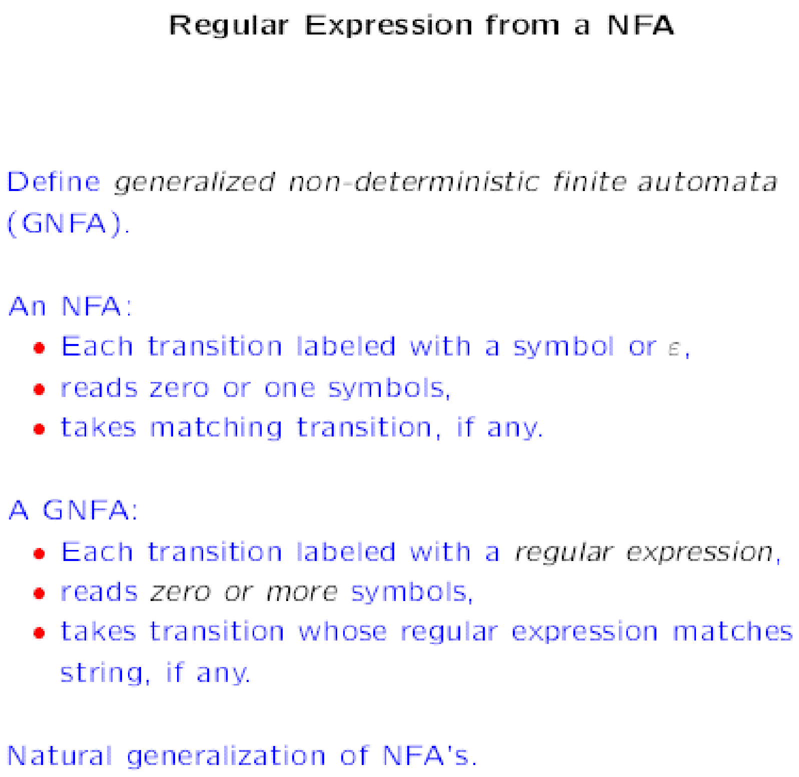


$(R_1)^*$:



Example





Regular Expression from a NFA

Define *generalized non-deterministic finite automata* (GNFA).

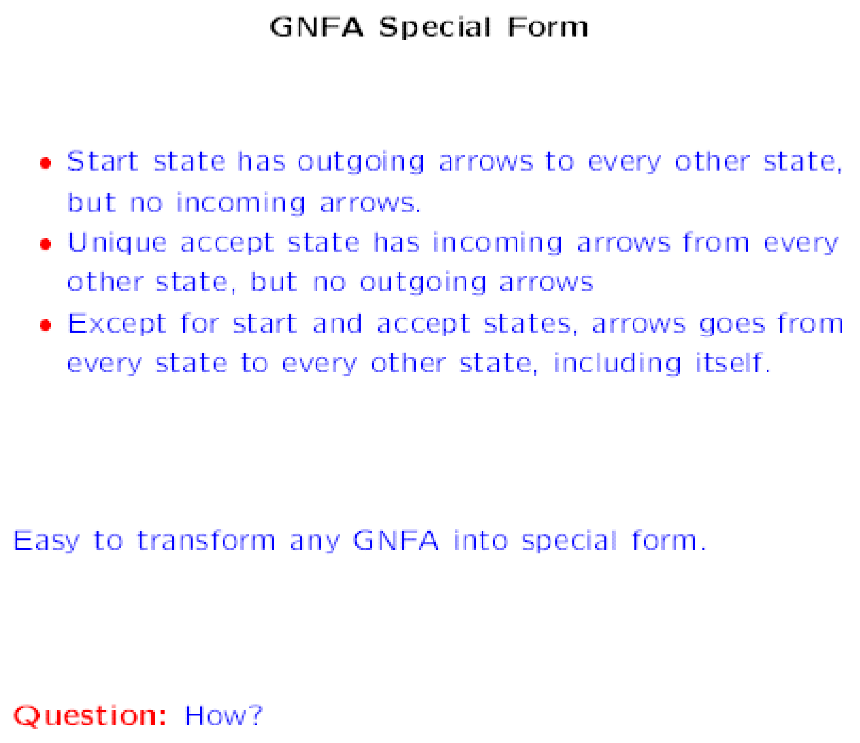
An NFA:

- Each transition labeled with a symbol or ε,
- reads zero or one symbols,
- takes matching transition, if any.

A GNFA:

- Each transition labeled with a *regular expression*,
- reads *zero or more* symbols,
- takes transition whose regular expression matches string, if any.

Natural generalization of NFA's.

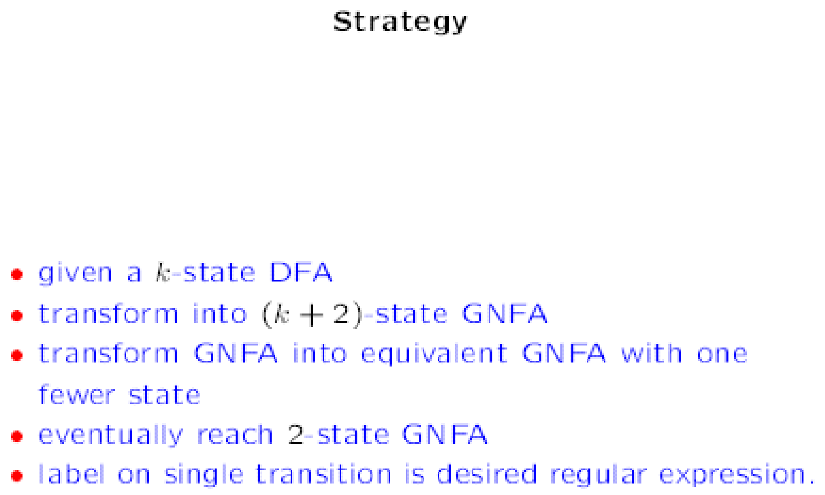


GNFA Special Form

- Start state has outgoing arrows to every other state, but no incoming arrows.
- Unique accept state has incoming arrows from every other state, but no outgoing arrows
- Except for start and accept states, arrows goes from every state to every other state, including itself.

Easy to transform any GNFA into special form.

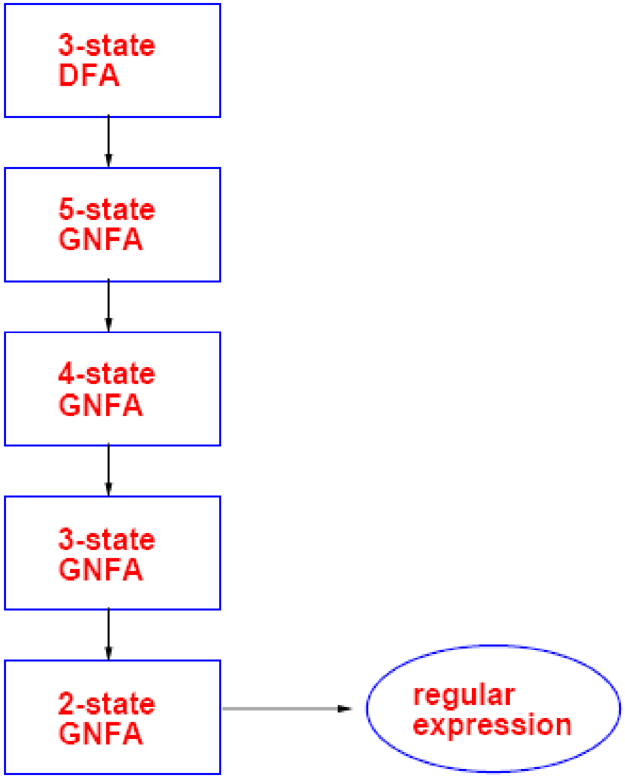
Question: How?



Strategy

- given a k-state DFA
- transform into $(k+2)$-state GNFA
- transform GNFA into equivalent GNFA with one fewer state
- eventually reach 2-state GNFA
- label on single transition is desired regular expression.

Strategy



3-state DFA

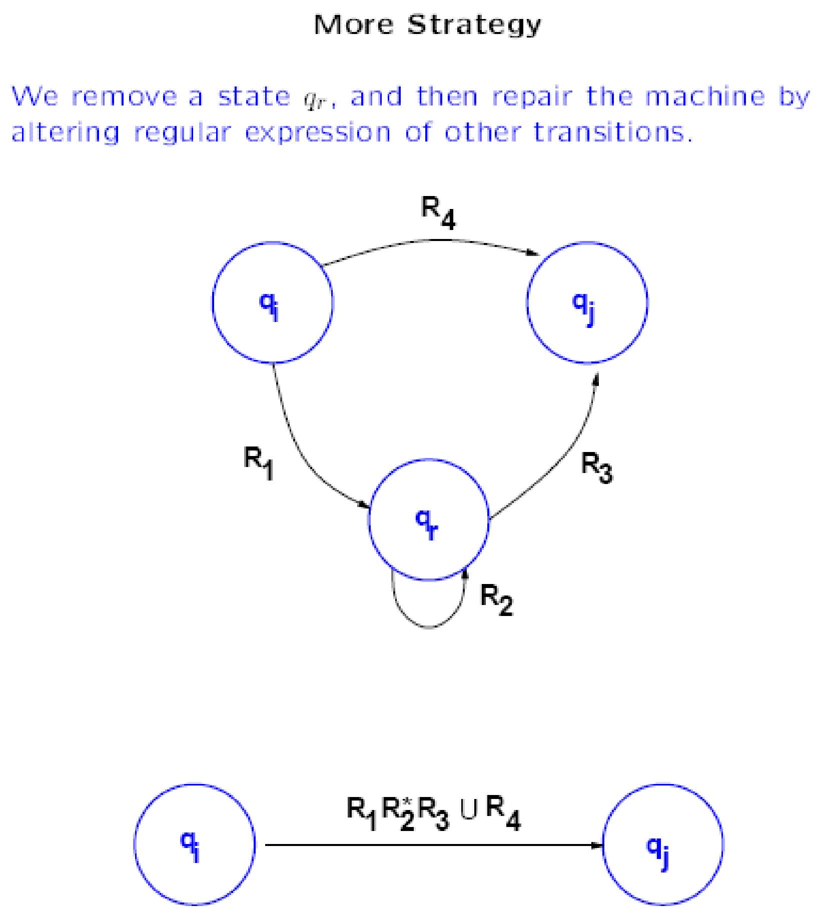
5-state GNFA

4-state GNFA

3-state GNFA

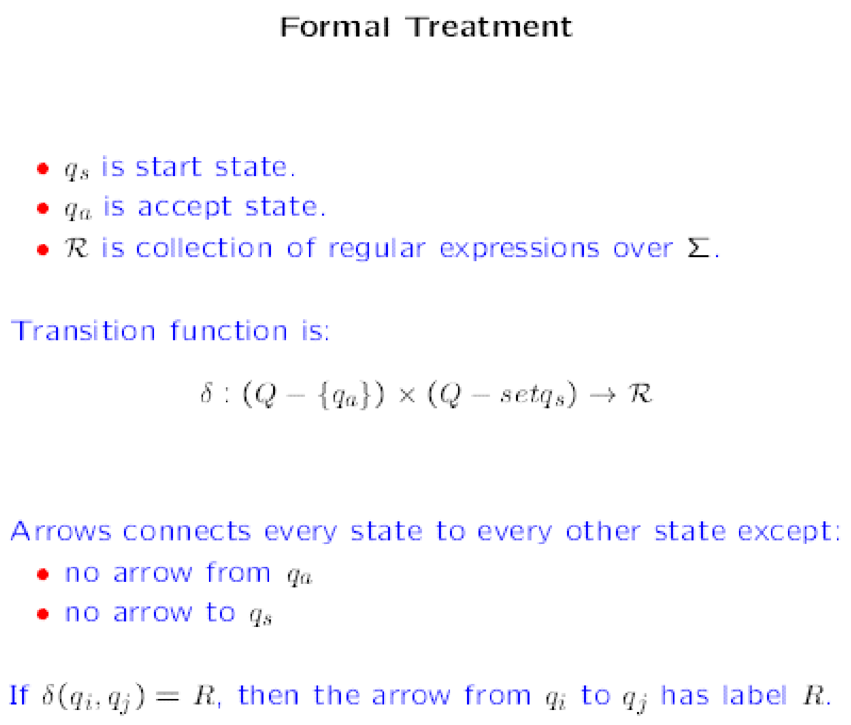
2-state GNFA

regular expression



More Strategy

We remove a state q_r, and then repair the machine by altering regular expression of other transitions.



Formal Treatment

- q_s is start state.
- q_a is accept state.
- $\mathcal{R}$ is collection of regular expressions over Σ.

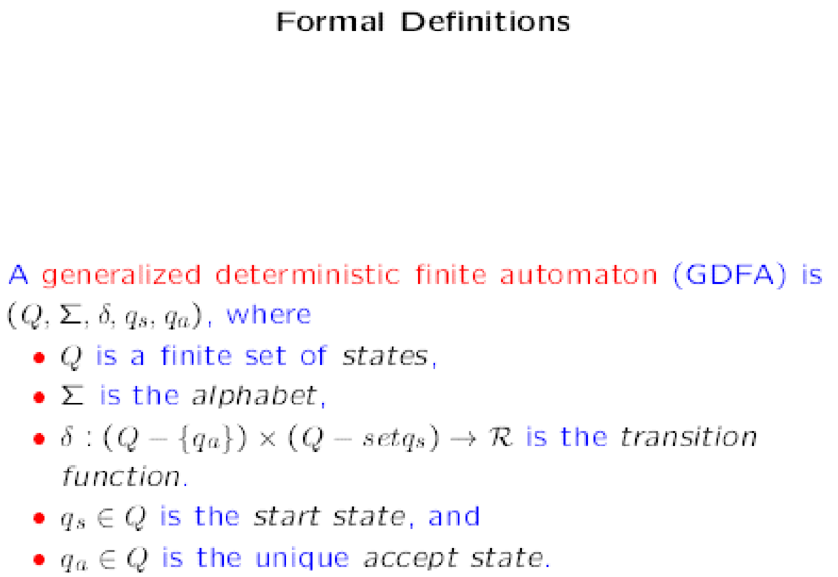
Transition function is:

$$\delta : (Q - \{q_a\}) \times (Q - setq_s) \to \mathcal{R}$$

Arrows connects every state to every other state except:

- no arrow from q_a
- no arrow to q_s

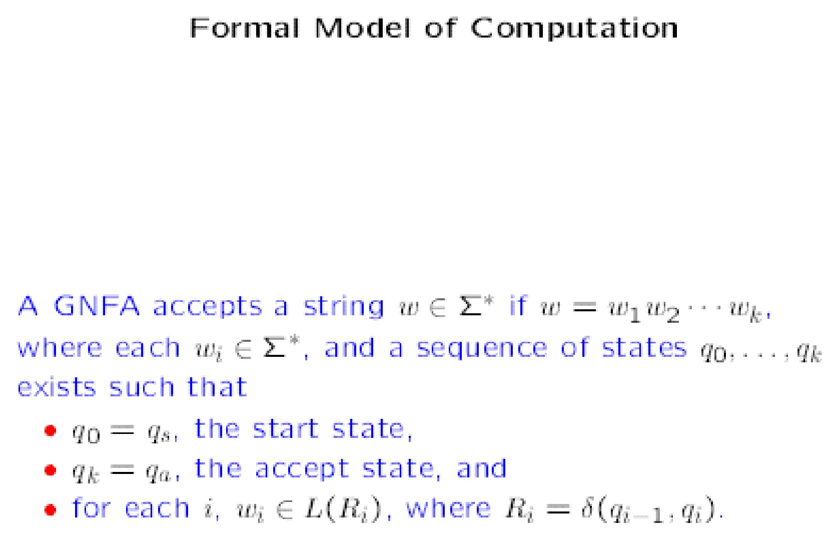
If $\delta(q_i, q_j) = R$, then the arrow from q_i to q_j has label R.



Formal Definitions

A generalized deterministic finite automaton (GDFA) is $(Q, \Sigma, \delta, q_s, q_a)$, where

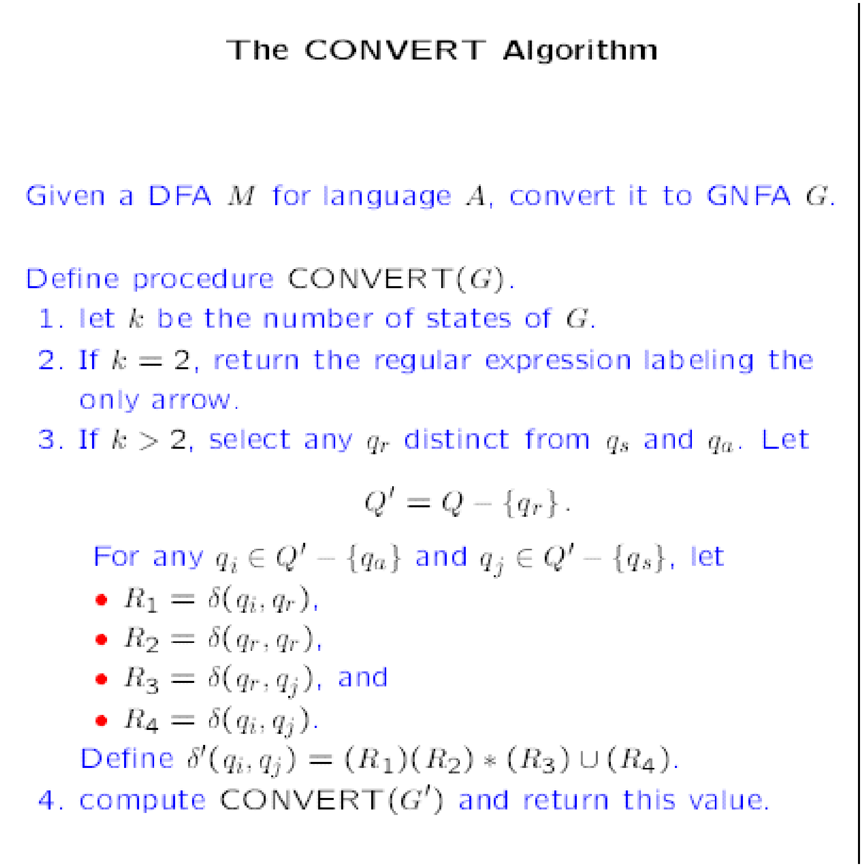
- Q is a finite set of *states*,
- Σ is the *alphabet*,
- $\delta : (Q - \{q_a\}) \times (Q - setq_s) \to \mathcal{R}$ is the *transition function*.
- $q_s \in Q$ is the *start state*, and
- $q_a \in Q$ is the unique *accept state*.



Formal Model of Computation

A GNFA accepts a string $w \in \Sigma^*$ if $w = w_1 w_2 \cdots w_k$, where each $w_i \in \Sigma^*$, and a sequence of states $q_0, \ldots, q_k$ exists such that

- $q_0 = q_s$, the start state,
- $q_k = q_a$, the accept state, and
- for each i, $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$.



The CONVERT Algorithm

Given a DFA M for language A, convert it to GNFA G.

Define procedure CONVERT(G).

1. let k be the number of states of G.
2. If $k = 2$, return the regular expression labeling the only arrow.
3. If $k > 2$, select any q_r distinct from q_s and q_a. Let

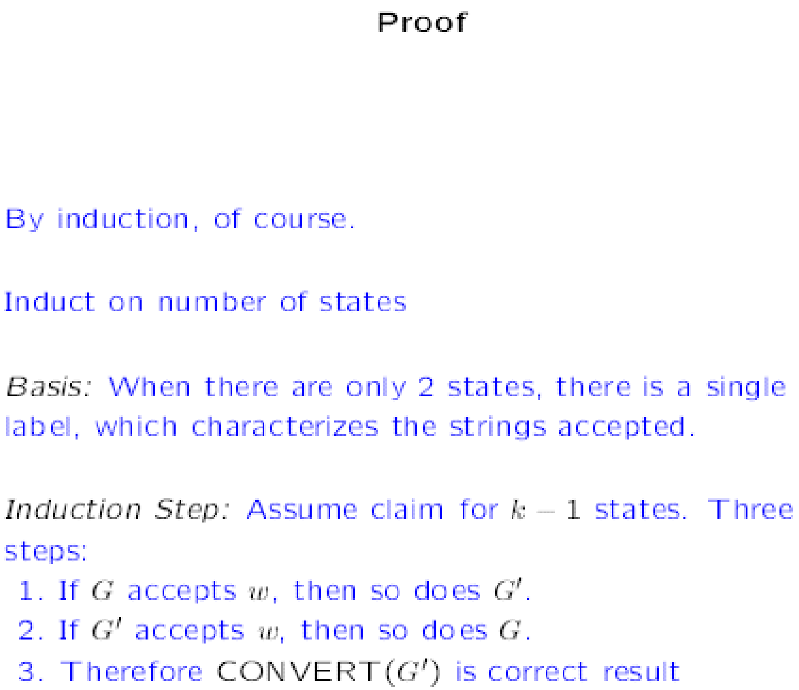
$$Q' = Q - \{q_r\}.$$

   For any $q_i \in Q' - \{q_a\}$ and $q_j \in Q' - \{q_s\}$, let

   - $R_1 = \delta(q_i, q_r)$,
   - $R_2 = \delta(q_r, q_r)$,
   - $R_3 = \delta(q_r, q_j)$, and
   - $R_4 = \delta(q_i, q_j)$.

   Define $\delta'(q_i, q_j) = (R_1)(R_2) * (R_3) \cup (R_4)$.

4. compute CONVERT(G') and return this value.



Proof

By induction, of course.

Induct on number of states

Basis: When there are only 2 states, there is a single label, which characterizes the strings accepted.

Induction Step: Assume claim for $k-1$ states. Three steps:

1. If G accepts w, then so does G'.
2. If G' accepts w, then so does G.
3. Therefore CONVERT(G') is correct result

Theorem: If G accepts w, then so does G'.

If G accepts w, then it traverses states

$$q_s, q_1, q_2, \ldots, q_a$$

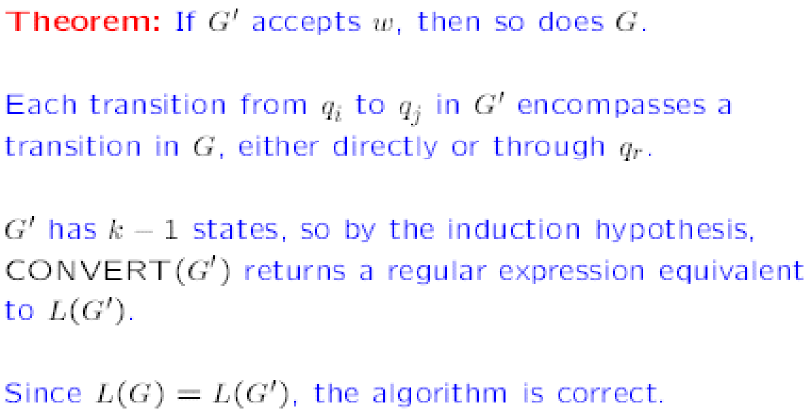
If q_r does not appear, then G' also accepts w, because new regular expression contains old regular expression in union.

If q_r appears, then consider the subsequence:

$$\ldots q_i, q_r, \ldots, q_r, q_j \ldots$$

The new regular expression linking q_i and q_j encompasses any such string.

Either way, the claim holds.

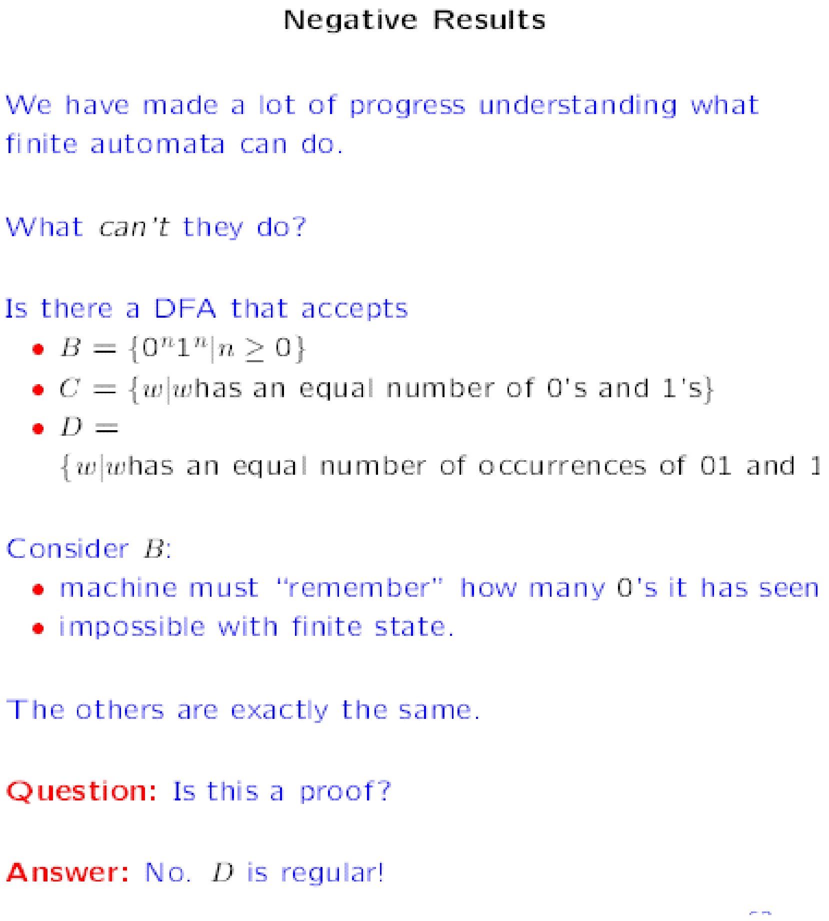


Theorem: If G' accepts w, then so does G.

Each transition from q_i to q_j in G' encompasses a transition in G, either directly or through q_r.

G' has $k-1$ states, so by the induction hypothesis, CONVERT(G') returns a regular expression equivalent to $L(G')$.

Since $L(G) = L(G')$, the algorithm is correct.



Negative Results

We have made a lot of progress understanding what finite automata can do.

What *can't* they do?

Is there a DFA that accepts

- $B = \{0^n 1^n | n \geq 0\}$
- $C = \{w | w$has an equal number of 0's and 1's$\}$
- $D =$ $\{w | w$has an equal number of occurrences of 01 and 1

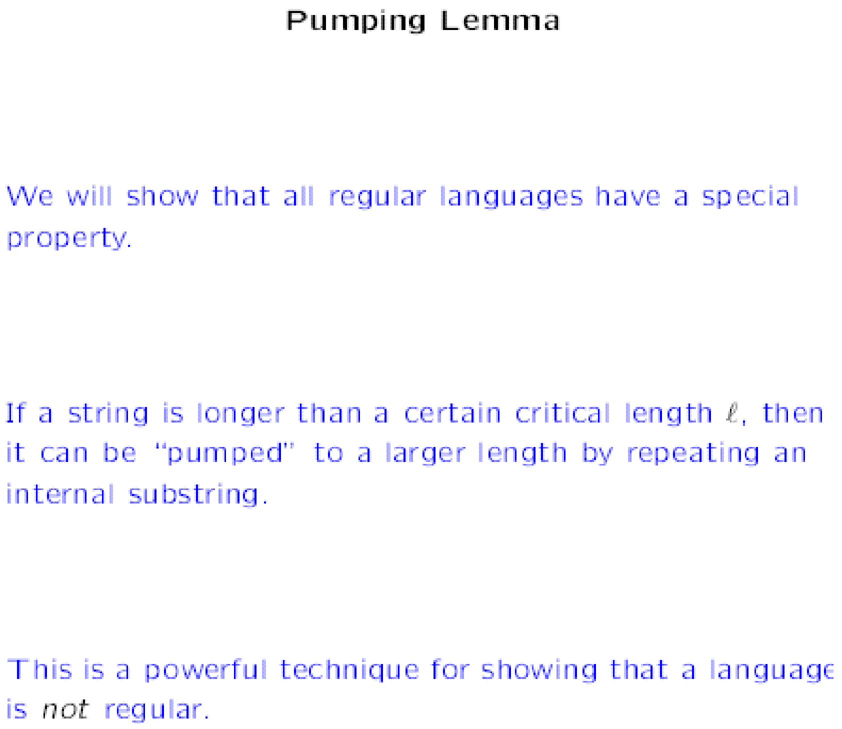
Consider B:

- machine must "remember" how many 0's it has seen
- impossible with finite state.

The others are exactly the same.

Question: Is this a proof?

Answer: No. D is regular!

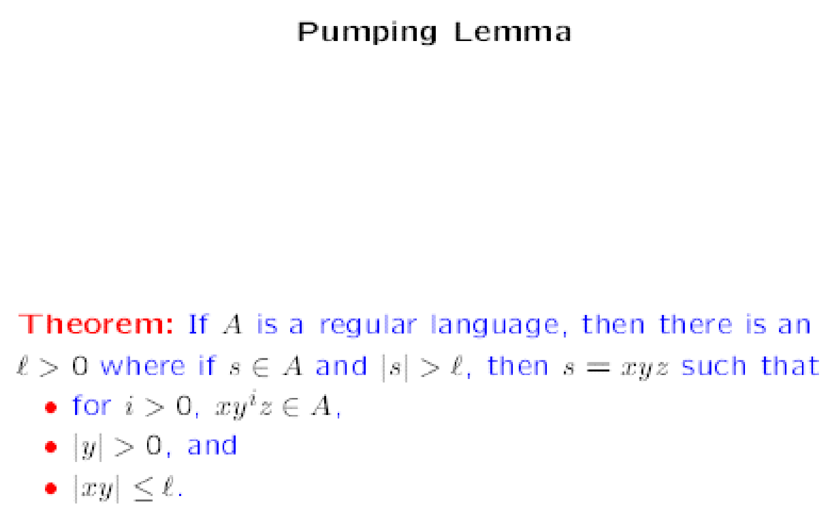


Pumping Lemma

We will show that all regular languages have a special property.

If a string is longer than a certain critical length ℓ, then it can be "pumped" to a larger length by repeating an internal substring.

This is a powerful technique for showing that a language is *not* regular.



Pumping Lemma

Theorem: If A is a regular language, then there is an $\ell > 0$ where if $s \in A$ and $|s| > \ell$, then $s = xyz$ such that

- for $i > 0$, $xy^iz \in A$,
- $|y| > 0$, and
- $|xy| \leq \ell$.

Pumping Lemma Proof

Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA that accepts A.

Let ℓ be $|Q|$, the number of states of M.

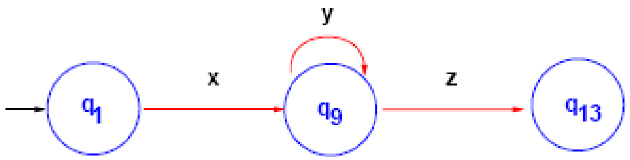
If $s \in A$ has length at least ℓ, consider the sequence of states:

$$s = \begin{array}{cccccccccccccc} s_1 & & s_2 & & s_3 & & s_4 & & s_5 & & s_6 & \dots & & s \\ & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & & \uparrow & \\ & q_1 & & q_20 & & q_9 & & q_17 & & q_9 & & & & q \end{array}$$

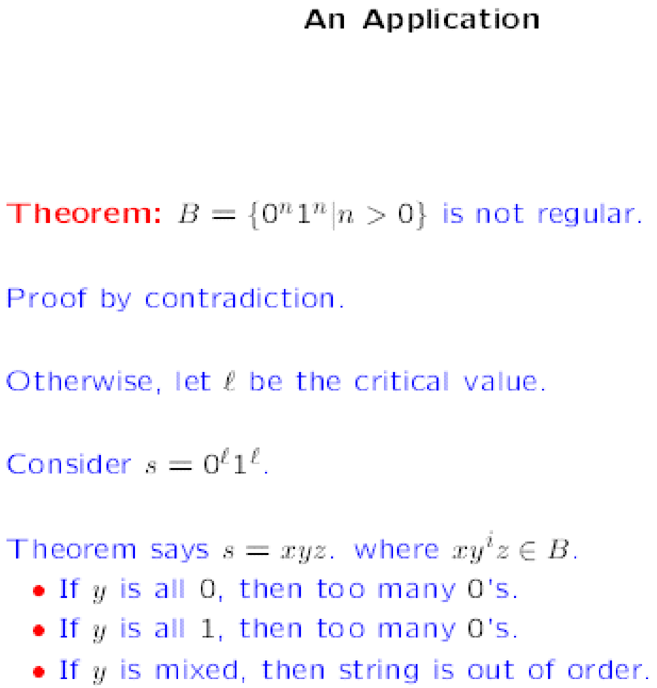
Since there are more then ℓ states, at least one is repeated. *pigeonhole principle*.

Pumping Lemma Proof

We can divide $s = xyz$, where



- from inspection, M accepts xy^kz.
- $|y| > 0$ because state is repeated.
- pick first state repetition to ensure that $|xy| < \ell$.



An Application

Theorem: $B = \{0^n 1^n | n > 0\}$ is not regular.

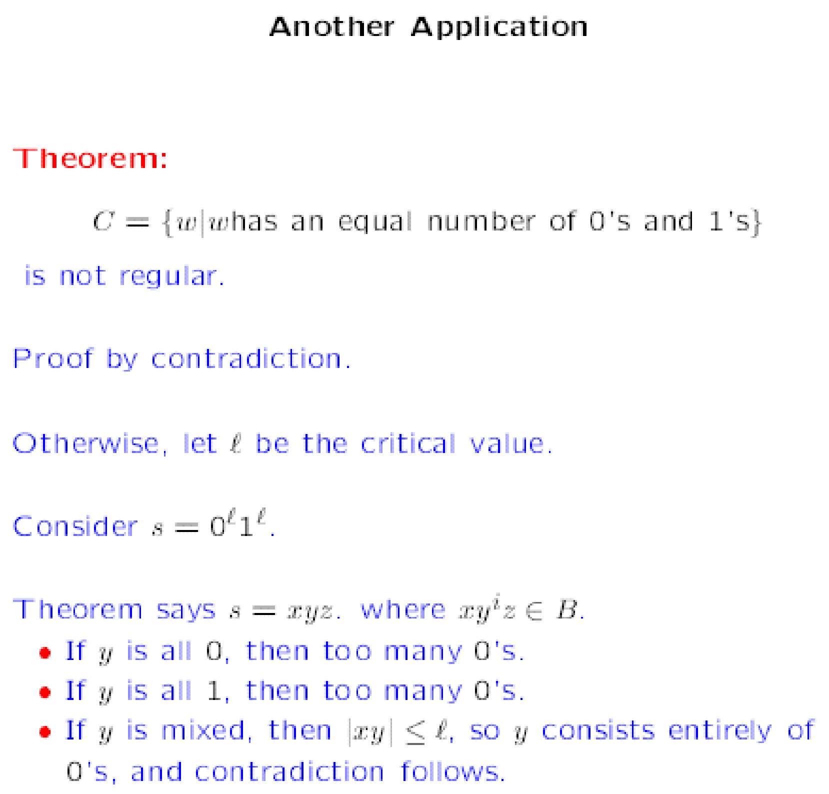
Proof by contradiction.

Otherwise, let ℓ be the critical value.

Consider $s = 0^\ell 1^\ell$.

Theorem says $s = xyz$. where $xy^i z \in B$.

- If y is all 0, then too many 0's.
- If y is all 1, then too many 0's.
- If y is mixed, then string is out of order.



Another Application

Theorem:

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

is not regular.

Proof by contradiction.

Otherwise, let ℓ be the critical value.

Consider $s = 0^\ell 1^\ell$.

Theorem says $s = xyz$. where $xy^iz \in B$.

- If y is all 0, then too many 0's.
- If y is all 1, then too many 0's.
- If y is mixed, then $|xy| \leq \ell$, so y consists entirely of 0's, and contradiction follows.

Algorithms for NDA's

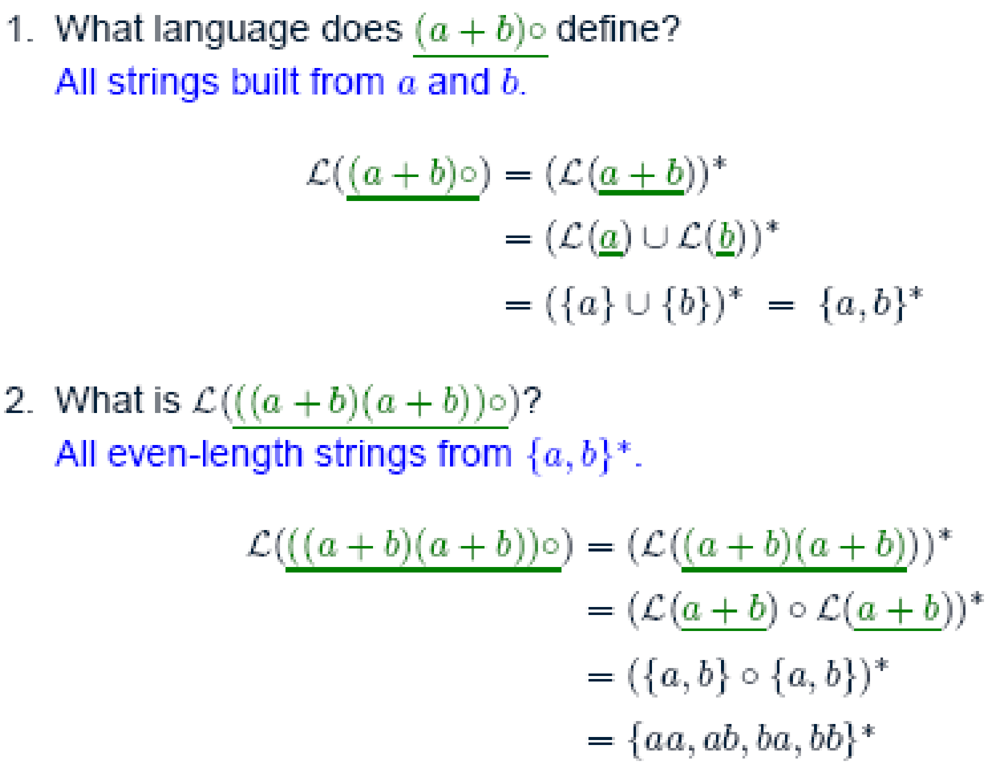
Question: Given NDA N and a string s, is $s \in L(N)$?

Answer: Construct the DFA equivalent to N and run it on w.

Question: Is $L(N) = \emptyset$?

Answer: By Pumping Lemma, suffices to check all strings of length less than the number of states of N.

Questions about Regular Languages



1. What language does $\underline{(a+b)\circ}$ define?

   All strings built from a and b.

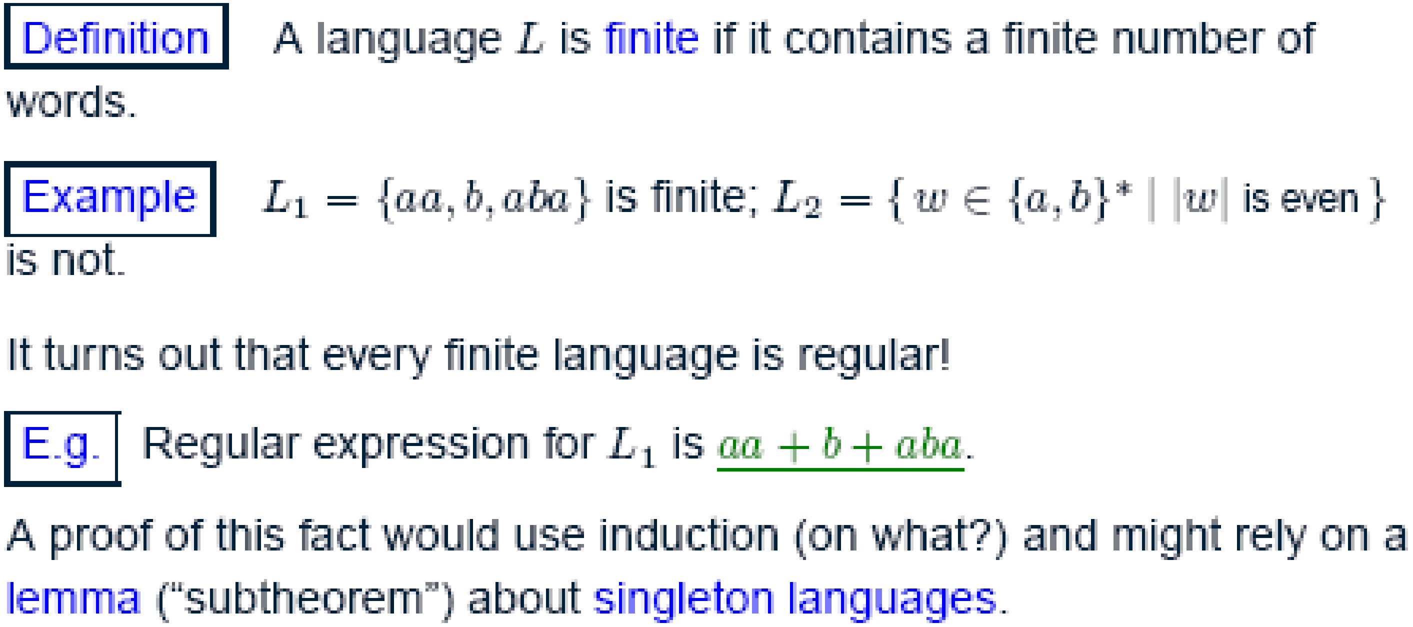
$$\begin{aligned}\mathcal{L}(\underline{(a+b)\circ}) &= (\mathcal{L}(\underline{a+b}))^{*}\\ &= (\mathcal{L}(\underline{a}) \cup \mathcal{L}(\underline{b}))^{*}\\ &= (\{a\} \cup \{b\})^{*} \;=\; \{a,b\}^{*}\end{aligned}$$

2. What is $\mathcal{L}(\underline{((a+b)(a+b))\circ})$?

   All even-length strings from $\{a,b\}^{*}$.

$$\begin{aligned}\mathcal{L}(\underline{((a+b)(a+b))\circ}) &= (\mathcal{L}(\underline{(a+b)(a+b)}))^{*}\\ &= (\mathcal{L}(\underline{a+b}) \circ \mathcal{L}(\underline{a+b}))^{*}\\ &= (\{a,b\} \circ \{a,b\})^{*}\\ &= \{aa, ab, ba, bb\}^{*}\end{aligned}$$

Question contd…



Definition A language L is finite if it contains a finite number of words.

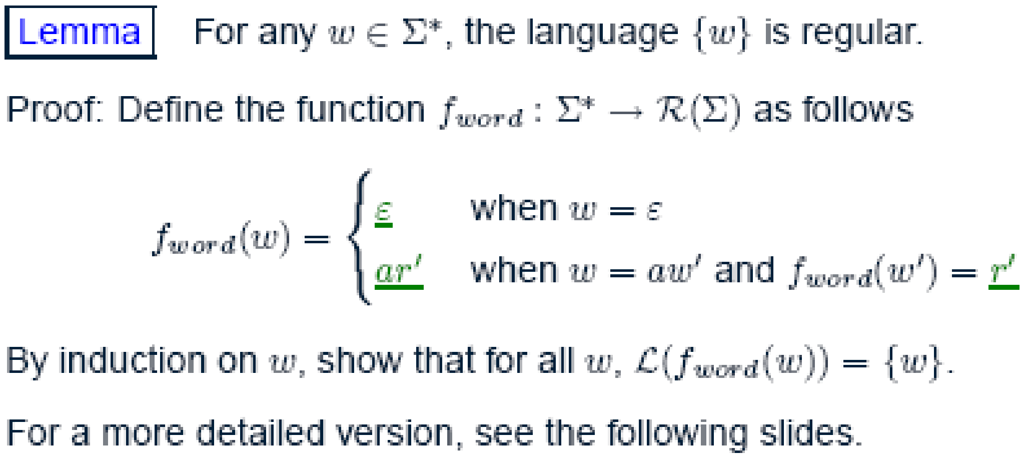
Example $L_1 = \{aa, b, aba\}$ is finite; $L_2 = \{\, w \in \{a, b\}^* \mid |w| \text{ is even} \,\}$ is not.

It turns out that every finite language is regular!

E.g. Regular expression for L_1 is $\underline{aa + b + aba}$.

A proof of this fact would use induction (on what?) and might rely on a lemma (“subtheorem”) about singleton languages.

Singleton Languages are Regular



Lemma For any $w \in \Sigma^*$, the language $\{w\}$ is regular.

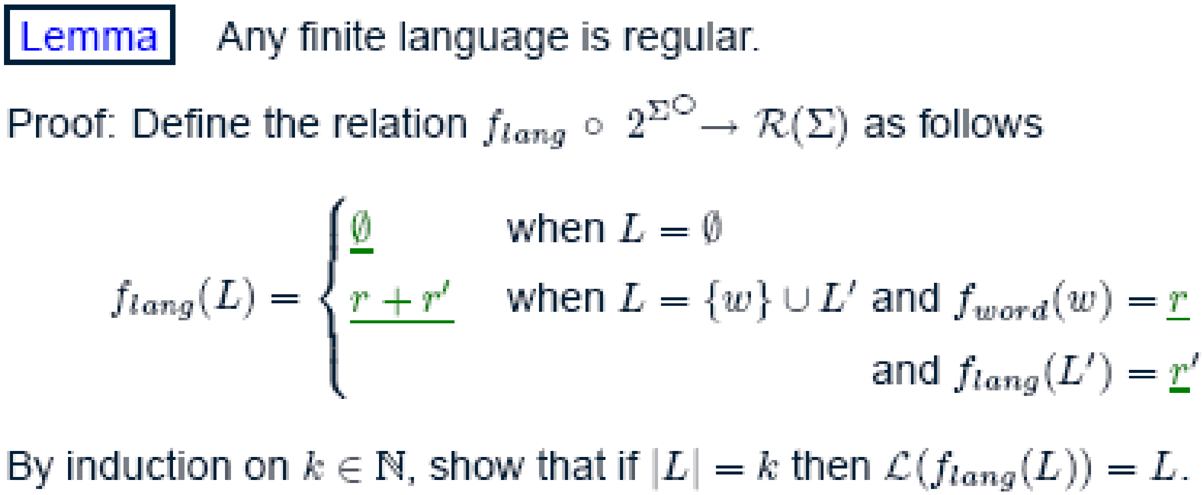
Proof: Define the function $f_{word} : \Sigma^* \to \mathcal{R}(\Sigma)$ as follows

$$f_{word}(w) = \begin{cases} \underline{\varepsilon} & \text{when } w = \varepsilon \\ \underline{ar'} & \text{when } w = aw' \text{ and } f_{word}(w') = \underline{r'} \end{cases}$$

By induction on w, show that for all w, $\mathcal{L}(f_{word}(w)) = \{w\}$.

For a more detailed version, see the following slides.

Finite Languages are regular

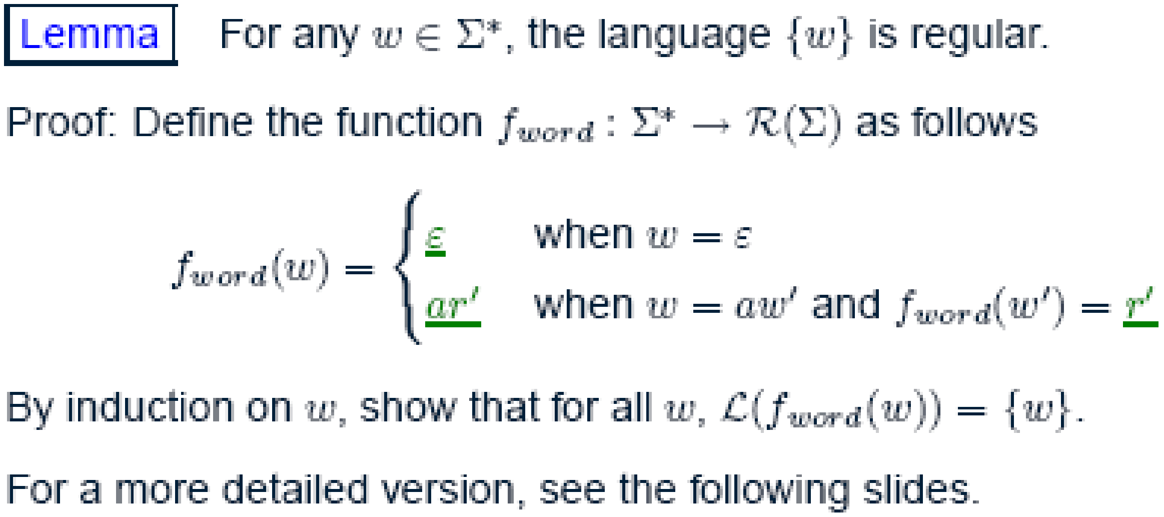


Lemma Any finite language is regular.

Proof: Define the relation $f_{lang} \circ 2^{\Sigma\bigcirc} \to \mathcal{R}(\Sigma)$ as follows

$$f_{lang}(L) = \begin{cases} \underline{\emptyset} & \text{when } L = \emptyset \\ \underline{r + r'} & \text{when } L = \{w\} \cup L' \text{ and } f_{word}(w) = \underline{r} \\ & \qquad\qquad\qquad\qquad \text{and } f_{lang}(L') = \underline{r'} \end{cases}$$

By induction on $k \in \mathbb{N}$, show that if $|L| = k$ then $\mathcal{L}(f_{lang}(L)) = L$.



Lemma For any $w \in \Sigma^*$, the language $\{w\}$ is regular.

Proof: Define the function $f_{word} : \Sigma^* \to \mathcal{R}(\Sigma)$ as follows

$$f_{word}(w) = \begin{cases} \underline{\varepsilon} & \text{when } w = \varepsilon \\ \underline{ar'} & \text{when } w = aw' \text{ and } f_{word}(w') = \underline{r'} \end{cases}$$

By induction on w, show that for all w, $\mathcal{L}(f_{word}(w)) = \{w\}$.

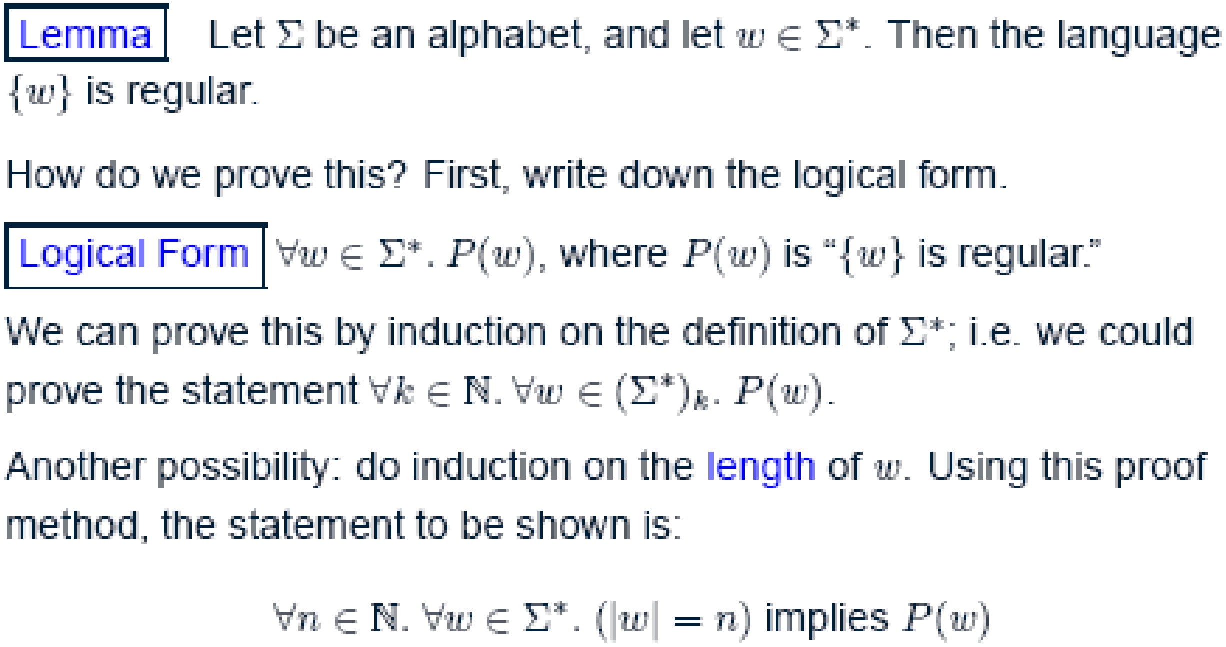
For a more detailed version, see the following slides.

Lemma Any finite language is regular.

Proof: Define the relation $f_{lang} \circ 2^{\Sigma^{\bigcirc}} \longrightarrow \mathcal{R}(\Sigma)$ as follows

$$f_{lang}(L) = \begin{cases} \underline{\emptyset} & \text{when } L = \emptyset \\ \underline{r + r'} & \text{when } L = \{w\} \cup L' \text{ and } f_{word}(w) = \underline{r} \\ & \qquad\qquad\qquad \text{and } f_{lang}(L') = \underline{r'} \end{cases}$$

By induction on $k \in \mathbb{N}$, show that if $|L| = k$ then $\mathcal{L}(f_{lang}(L)) = L$.



Lemma Let Σ be an alphabet, and let $w \in \Sigma^*$. Then the language $\{w\}$ is regular.

How do we prove this? First, write down the logical form.

Logical Form $\forall w \in \Sigma^*.\ P(w)$, where $P(w)$ is "$\{w\}$ is regular."

We can prove this by induction on the definition of Σ^*; i.e. we could prove the statement $\forall k \in \mathbb{N}.\ \forall w \in (\Sigma^*)_k.\ P(w)$.

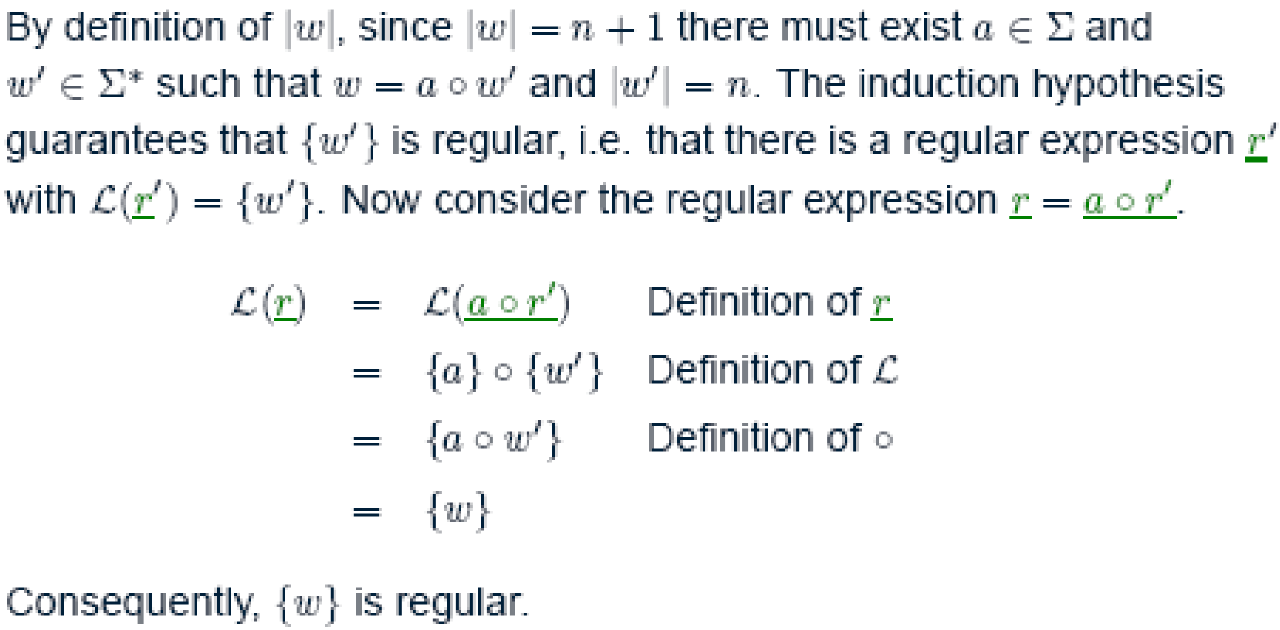
Another possibility: do induction on the length of w. Using this proof method, the statement to be shown is:

$$\forall n \in \mathbb{N}.\ \forall w \in \Sigma^*.\ (|w| = n) \text{ implies } P(w)$$

The proof proceeds by induction on word length; the statement to be proved is $\forall n \in \mathbb{N}.\ Q(n)$, where $Q(n)$ is "$\forall w \in \Sigma^*.\ (|w| = n) \circ\!\!\longrightarrow \{w\}$ "is regular".

Base case. We must show $Q(0)$, i.e. that for any word w, if $|w| = 0$, then $\{w\}$ is regular. So fix w and assume that $|w| = 0$. This implies that $w = \varepsilon$. But $\{\varepsilon\}$ is regular, since the regular expression $\underline{\varepsilon}$ is such that $\mathcal{L}(\underline{\varepsilon}) = \{\varepsilon\}$.

Induction step. We must show that for any n, $Q(n) \circ\!\!\longrightarrow Q(n+1)$. So fix n and assume (induction hypothesis) that $Q(n)$ holds. We must prove $Q(n+1)$, i.e. that for any w of length $n+1$, $\{w\}$ is regular. Now fix w and assume that $|w| = n+1$; we must find a regular expression $\underline{r}$ such that $\mathcal{L}(\underline{r}) = \{w\}$.

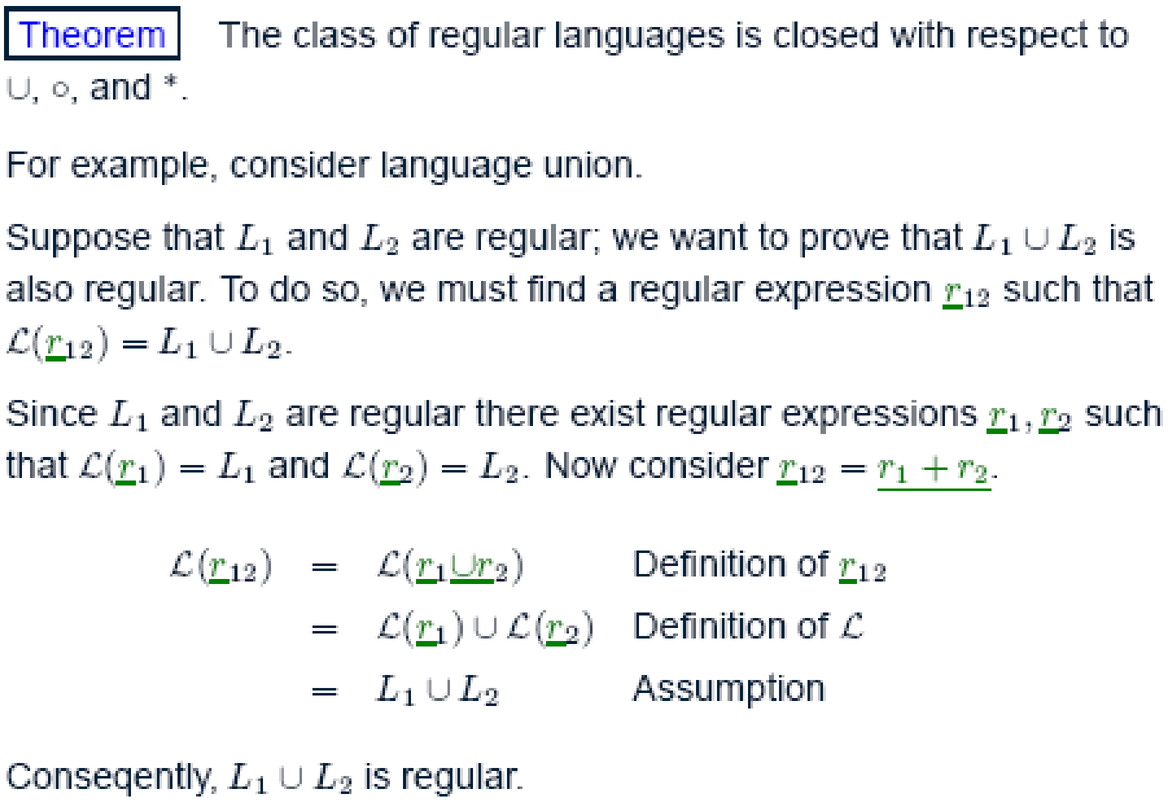


By definition of $|w|$, since $|w| = n + 1$ there must exist $a \in \Sigma$ and $w' \in \Sigma^*$ such that $w = a \circ w'$ and $|w'| = n$. The induction hypothesis guarantees that $\{w'\}$ is regular, i.e. that there is a regular expression $\underline{r}'$ with $\mathcal{L}(\underline{r}') = \{w'\}$. Now consider the regular expression $\underline{r} = \underline{a \circ r'}$.

$$\begin{aligned} \mathcal{L}(\underline{r}) &= \mathcal{L}(\underline{a \circ r'}) && \text{Definition of } \underline{r} \\ &= \{a\} \circ \{w'\} && \text{Definition of } \mathcal{L} \\ &= \{a \circ w'\} && \text{Definition of } \circ \\ &= \{w\} \end{aligned}$$

Consequently, $\{w\}$ is regular.

Closure Properties for Regular Languages



Theorem The class of regular languages is closed with respect to $\cup$, $\circ$, and $*$.

For example, consider language union.

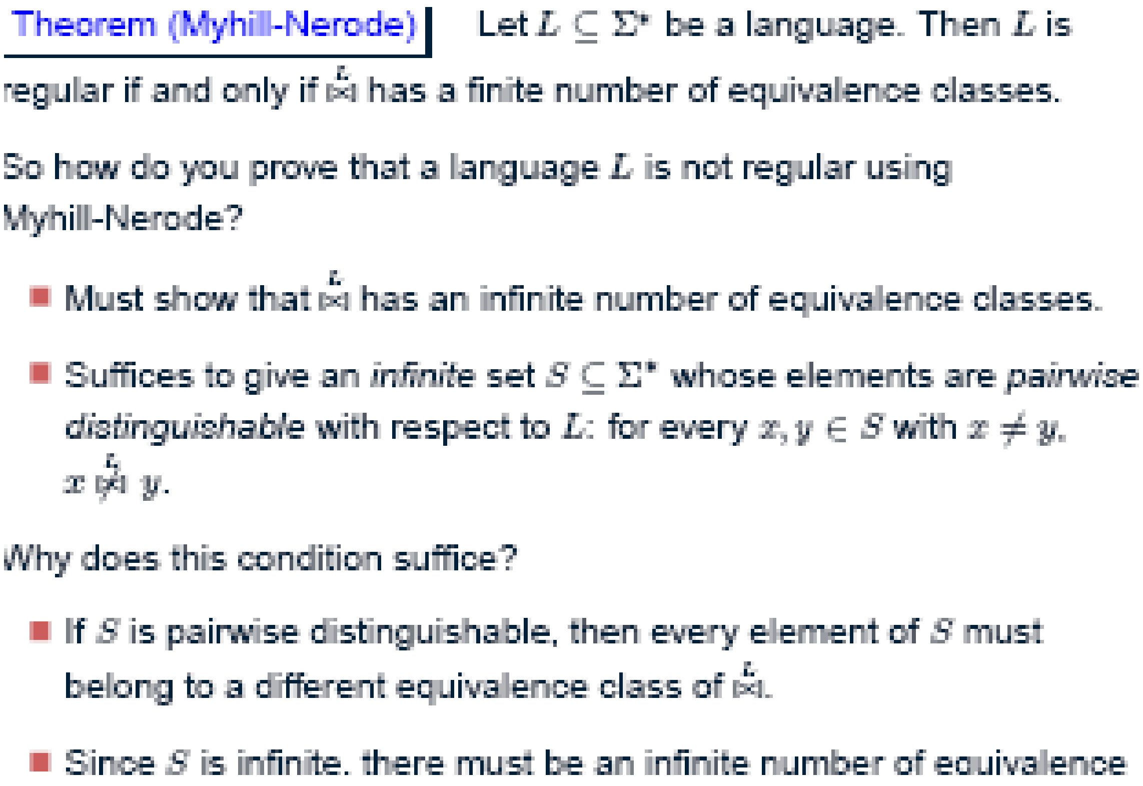
Suppose that L_1 and L_2 are regular; we want to prove that $L_1 \cup L_2$ is also regular. To do so, we must find a regular expression r_{12} such that $\mathcal{L}(r_{12}) = L_1 \cup L_2$.

Since L_1 and L_2 are regular there exist regular expressions r_1, r_2 such that $\mathcal{L}(r_1) = L_1$ and $\mathcal{L}(r_2) = L_2$. Now consider $r_{12} = r_1 + r_2$.

$$\begin{aligned} \mathcal{L}(r_{12}) &= \mathcal{L}(r_1 \cup r_2) && \text{Definition of } r_{12} \\ &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) && \text{Definition of } \mathcal{L} \\ &= L_1 \cup L_2 && \text{Assumption} \end{aligned}$$

Conseqently, $L_1 \cup L_2$ is regular.

Myhill-Nerode Theorem



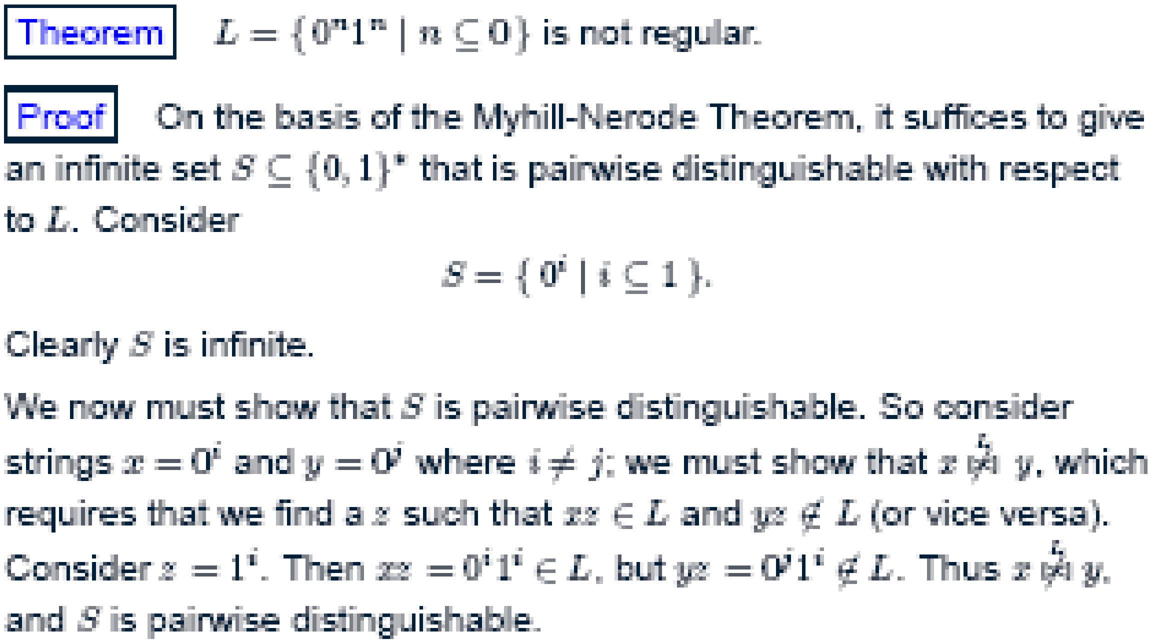
Theorem (Myhill-Nerode) Let $L \subseteq \Sigma^*$ be a language. Then L is regular if and only if $\overset{L}{\bowtie}$ has a finite number of equivalence classes.

So how do you prove that a language L is not regular using Myhill-Nerode?

- Must show that $\overset{L}{\bowtie}$ has an infinite number of equivalence classes.
- Suffices to give an *infinite* set $S \subseteq \Sigma^*$ whose elements are *pairwise distinguishable* with respect to L: for every $x, y \in S$ with $x \neq y$, $x \overset{L}{\not\bowtie} y$.

Why does this condition suffice?

- If S is pairwise distinguishable, then every element of S must belong to a different equivalence class of $\overset{L}{\bowtie}$.
- Since S is infinite, there must be an infinite number of equivalence



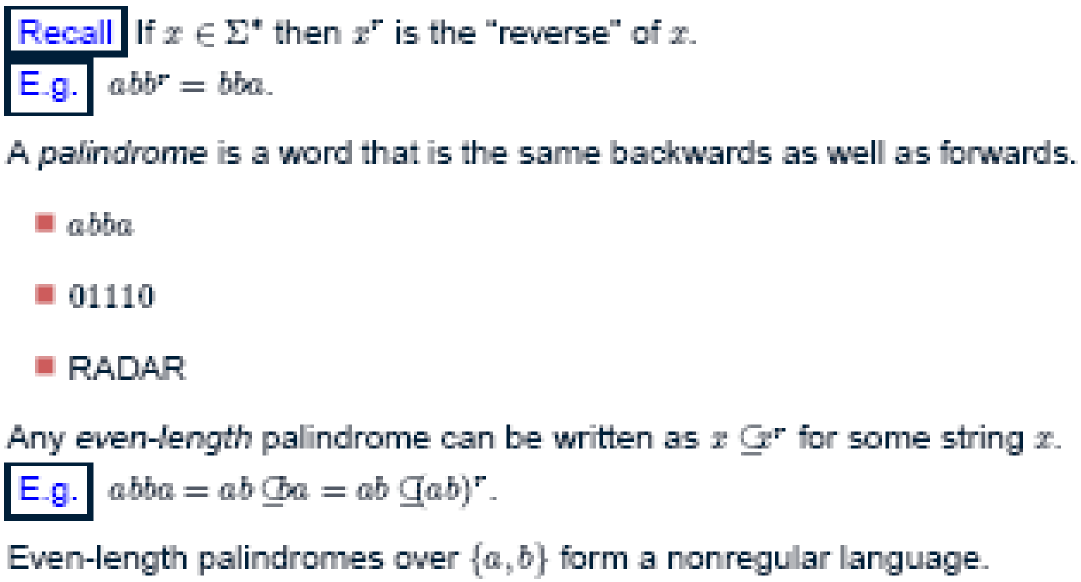
Theorem $L = \{ 0^n 1^n \mid n \subseteq 0 \}$ is not regular.

Proof On the basis of the Myhill-Nerode Theorem, it suffices to give an infinite set $S \subseteq \{0,1\}^*$ that is pairwise distinguishable with respect to L. Consider

$$S = \{ 0^i \mid i \subseteq 1 \}.$$

Clearly S is infinite.

We now must show that S is pairwise distinguishable. So consider strings $x = 0^i$ and $y = 0^j$ where $i \neq j$; we must show that $x \overset{L}{\not\equiv} y$, which requires that we find a z such that $xz \in L$ and $yz \notin L$ (or vice versa). Consider $z = 1^i$. Then $xz = 0^i 1^i \in L$, but $yz = 0^j 1^i \notin L$. Thus $x \overset{L}{\not\equiv} y$, and S is pairwise distinguishable.



Recall If $x \in \Sigma^*$ then x^r is the "reverse" of x.

E.g. $abb^r = bba$.

A *palindrome* is a word that is the same backwards as well as forwards.

- $abba$
- 01110
- RADAR

Any *even-length* palindrome can be written as $x \subseteq x^r$ for some string x.

E.g. $abba = ab \subseteq ba = ab \subseteq (ab)^r$.

Even-length palindromes over $\{a, b\}$ form a nonregular language.

Theorem Let $E = \{ x \subseteq x^r \mid x \in \{a,b\}^* \}$. Then E is not regular.

Proof On the basis of the Myhill-Nerode Theorem it suffices to come up with an infinite set $S \subseteq \{a,b\}^*$ that is pairwise distinguishable with respect to E. Consider

$$S = \{ a^i b \mid i \subseteq 0 \}.$$

Clearly S is infinite.

To show pairwise distinguishability, consider $x = a^i b$ and $y = a^j b$ where $i \neq j$; we must show $x \overset{E}{\not\equiv}_L y$, i.e. we must find a z with $xz \in L$ and $yz \notin L$, or vice versa. Consider

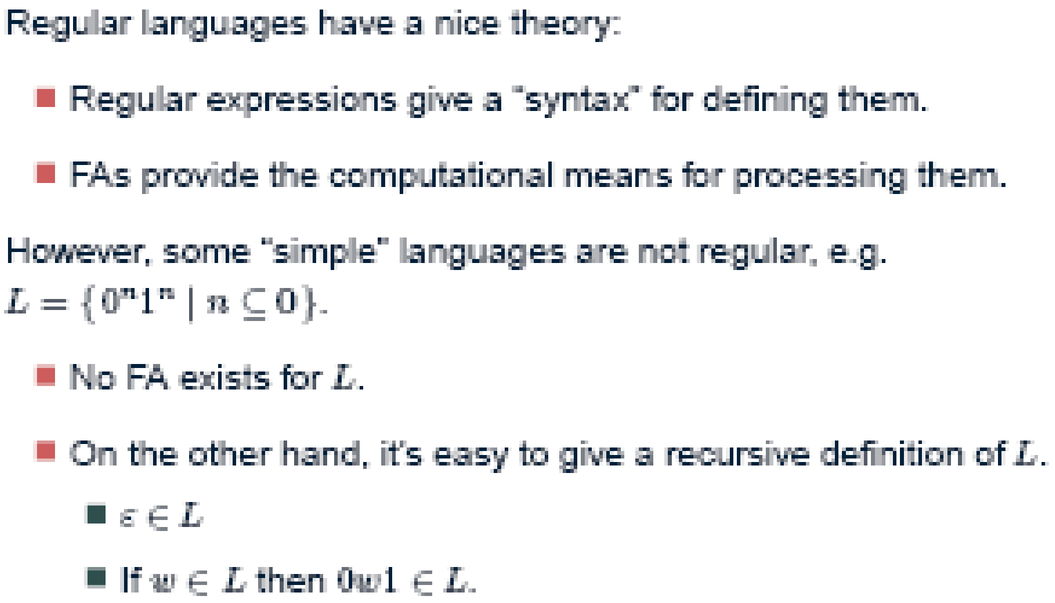
$$z = x^r = ba^i.$$

By definition $xz \in L$. However, $yz = a^j bba^i \notin L$ since $j \neq i$.

UNIT III

CONTEXT-FREE GRAMMAR
AND
LANGUAGES

Context Free Grammars



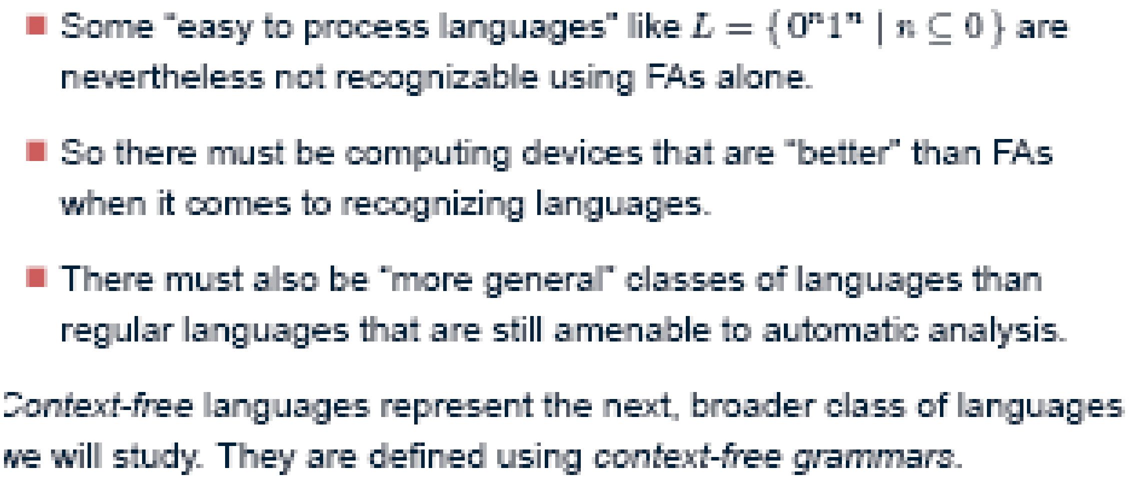
Regular languages have a nice theory:

- Regular expressions give a "syntax" for defining them.
- FAs provide the computational means for processing them.

However, some "simple" languages are not regular, e.g.
$L = \{0^n1^n \mid n \subseteq 0\}$.

- No FA exists for L.
- On the other hand, it's easy to give a recursive definition of L.
  - $\varepsilon \in L$
  - If $w \in L$ then $0w1 \in L$.

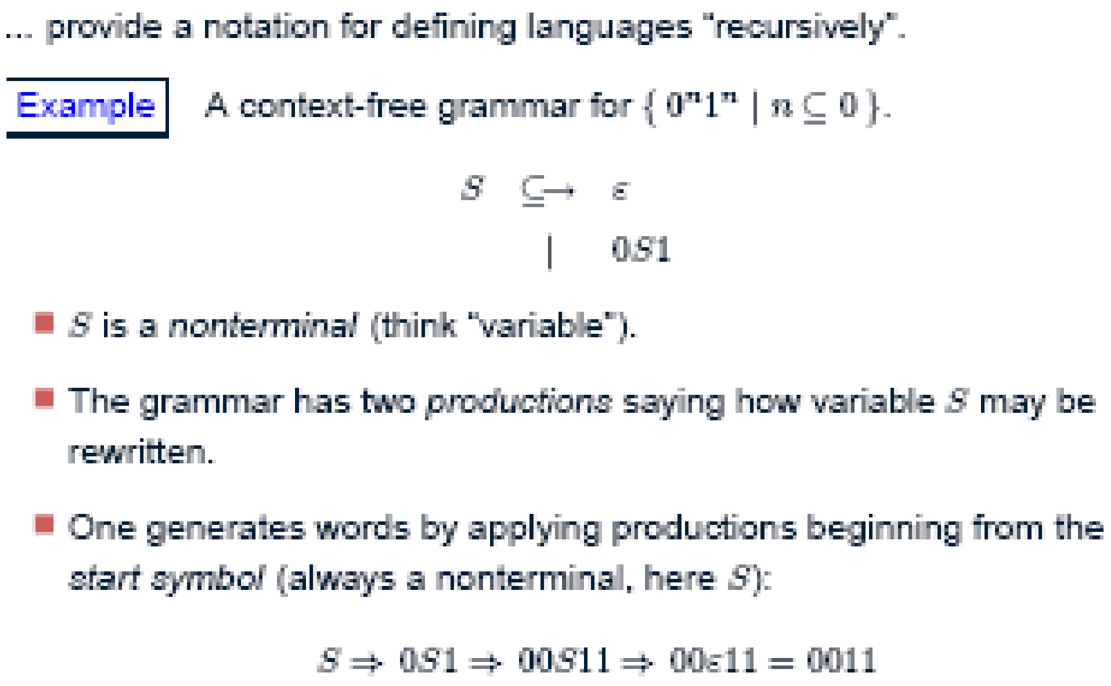
Cont…



- Some "easy to process languages" like $L = \{0^n1^n \mid n \subseteq 0\}$ are nevertheless not recognizable using FAs alone.
- So there must be computing devices that are "better" than FAs when it comes to recognizing languages.
- There must also be "more general" classes of languages than regular languages that are still amenable to automatic analysis.

Context-free languages represent the next, broader class of languages we will study. They are defined using *context-free grammars*.

Cont…



... provide a notation for defining languages "recursively".

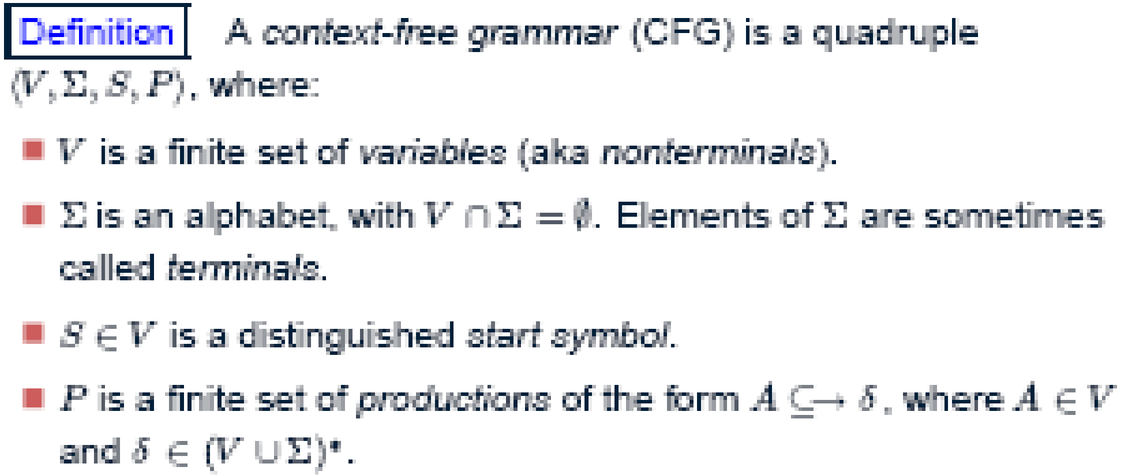
Example A context-free grammar for $\{ 0^n1^n \mid n \subseteq 0 \}$.

$$\begin{aligned} S \;&\subseteq\!\rightarrow\; \varepsilon \\ &\mid\; 0S1 \end{aligned}$$

- S is a *nonterminal* (think "variable").
- The grammar has two *productions* saying how variable S may be rewritten.
- One generates words by applying productions beginning from the *start symbol* (always a nonterminal, here S):

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 00\varepsilon 11 = 0011$$

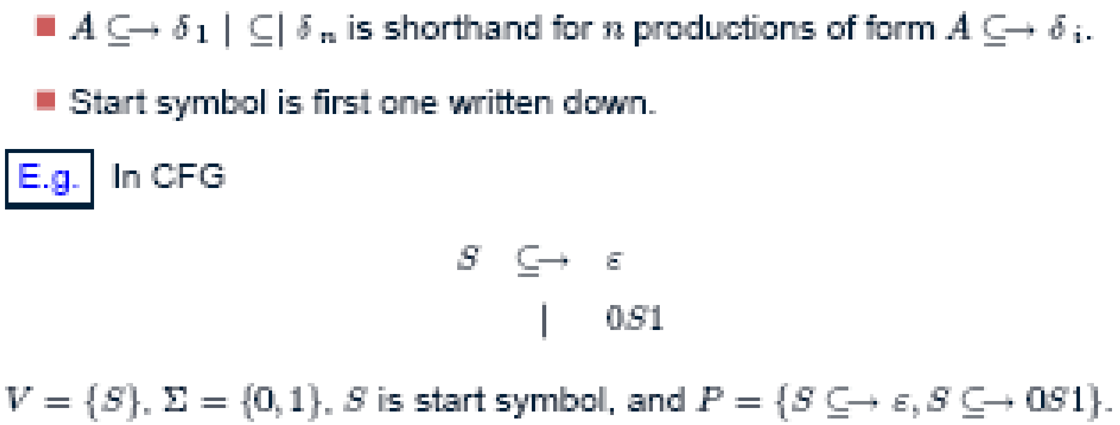
Defining CFG



Definition A *context-free grammar* (CFG) is a quadruple (V, Σ, S, P), where:

- V is a finite set of *variables* (aka *nonterminals*).
- Σ is an alphabet, with $V \cap \Sigma = \emptyset$. Elements of Σ are sometimes called *terminals*.
- $S \in V$ is a distinguished *start symbol*.
- P is a finite set of *productions* of the form $A \subsetneq\!\rightarrow \delta$, where $A \in V$ and $\delta \in (V \cup \Sigma)^*$.

Notational conventions For CFGs



- $A \subsetneq\rightarrow \delta_1 \mid \subsetneq\mid \delta_n$ is shorthand for n productions of form $A \subsetneq\rightarrow \delta_i$.
- Start symbol is first one written down.

E.g. In CFG

$$\begin{array}{lll} S & \subsetneq\rightarrow & \varepsilon \\ & \mid & 0S1 \end{array}$$

$V = \{S\}$, $\Sigma = \{0, 1\}$, S is start symbol, and $P = \{S \subsetneq\rightarrow \varepsilon, S \subsetneq\rightarrow 0S1\}$.

OTHER CFG EXAMPLES

Palindromes over $\Sigma = \{a, b\}$

$$\begin{aligned} S &\rightarrow \varepsilon \mid a \mid b \\ &\mid \quad aSa \mid bSb \end{aligned}$$

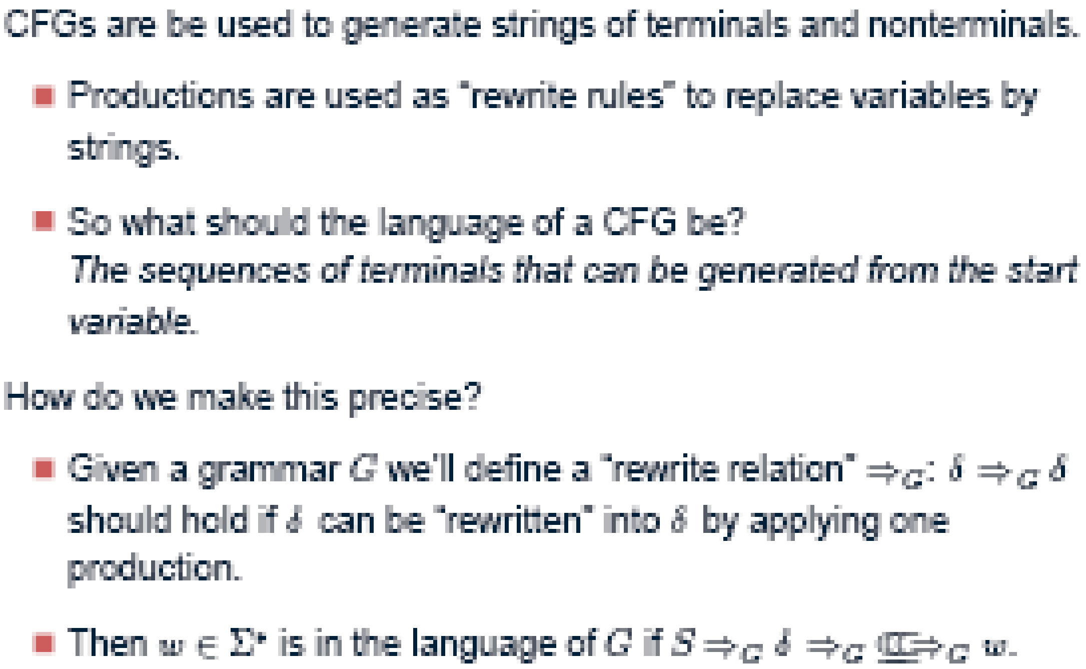
Sample word: $S \Rightarrow aSa \Rightarrow abSba \Rightarrow ababa$

Nonpalindromes over $\Sigma = \{a, b\}$

$$\begin{aligned} S &\rightarrow aSa \mid bSb \mid aAb \mid bAa \\ A &\rightarrow \varepsilon \mid aA \mid bA \end{aligned}$$

Sample word: $S \Rightarrow aSa \Rightarrow aaAba \Rightarrow aa\varepsilon ba = aaba$

Languages of CFG



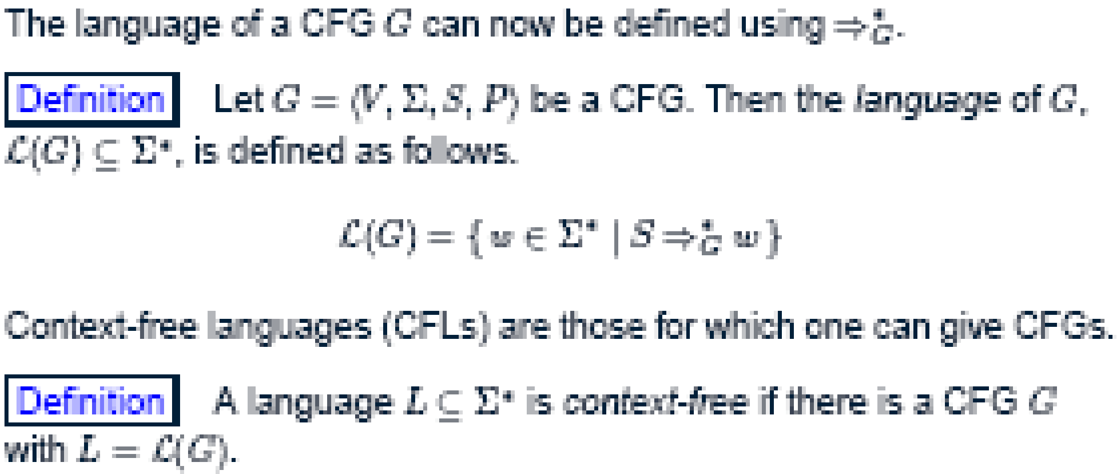
CFGs are be used to generate strings of terminals and nonterminals.

- Productions are used as "rewrite rules" to replace variables by strings.
- So what should the language of a CFG be?
  The sequences of terminals that can be generated from the start variable.

How do we make this precise?

- Given a grammar G we'll define a "rewrite relation" $\Rightarrow_G$: $\delta \Rightarrow_G \delta$ should hold if δ can be "rewritten" into δ by applying one production.
- Then $w \in \Sigma^*$ is in the language of G if $S \Rightarrow_G \delta \Rightarrow_G \cdots \Rightarrow_G w$.

Languages of CFG



The language of a CFG G can now be defined using $\Rightarrow_G^*$.

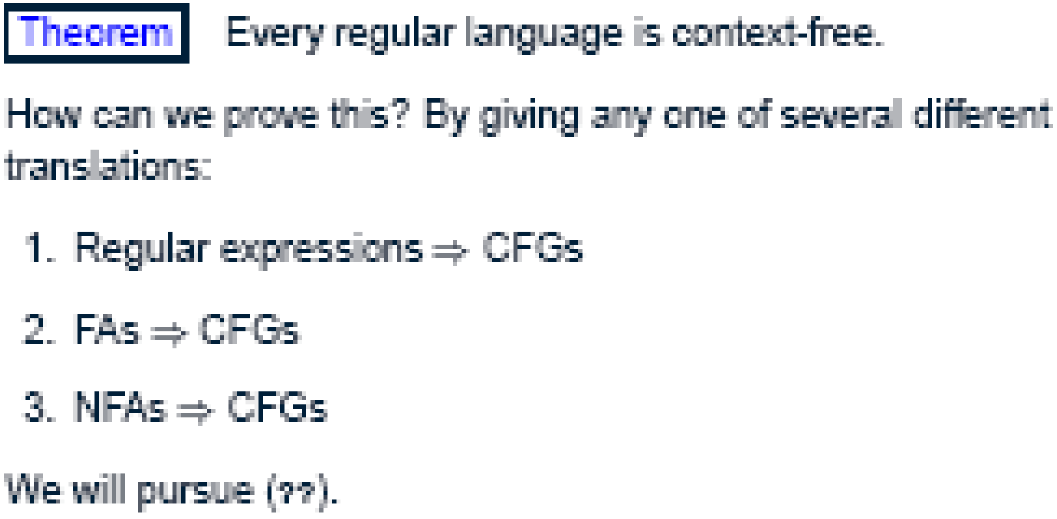
Definition Let $G = (V, \Sigma, S, P)$ be a CFG. Then the *language of* G, $\mathcal{L}(G) \subseteq \Sigma^*$, is defined as follows.

$$\mathcal{L}(G) = \{ w \in \Sigma^* \mid S \Rightarrow_G^* w \}$$

Context-free languages (CFLs) are those for which one can give CFGs.

Definition A language $L \subseteq \Sigma^*$ is *context-free* if there is a CFG G with $L = \mathcal{L}(G)$.

Regular Languages and CFL



Theorem Every regular language is context-free.

How can we prove this? By giving any one of several different translations:

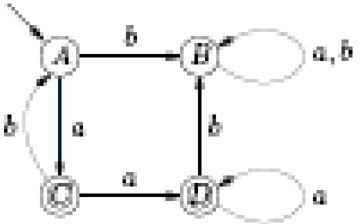
1. Regular expressions ⇒ CFGs
2. FAs ⇒ CFGs
3. NFAs ⇒ CFGs

We will pursue (??).

Translating FAs into CFG

How do we do this? By turning:

- states into variables;
- transitions into productions; and
- acceptance into ε-productions.



$$
\begin{aligned}
A &\subseteq\rightarrow aC \mid bB \\
B &\subseteq\rightarrow aB \mid bB \\
C &\subseteq\rightarrow aD \mid bA \mid \varepsilon \\
D &\subseteq\rightarrow aD \mid bB \mid \varepsilon
\end{aligned}
$$

Note $\delta^*(A, aab) = B$, and $A \Rightarrow^*_G aabB$.

Formalizing the Translation

Given a FA $M = \langle Q, \Sigma, q_0, \delta, A \rangle$, we want to define CFG $G_M = \langle V, \Sigma, S, P \rangle$ so that $\mathcal{L}(M) = \mathcal{L}(G_M)$. Assume without loss of generality that $Q \cap \Sigma = \emptyset$.

- $V = Q$
- $S = q_0$
- $P = \{ q \to a\,\delta(q, a) \mid q \in Q \} \cup \{ q \to \varepsilon \mid q \in A \}$

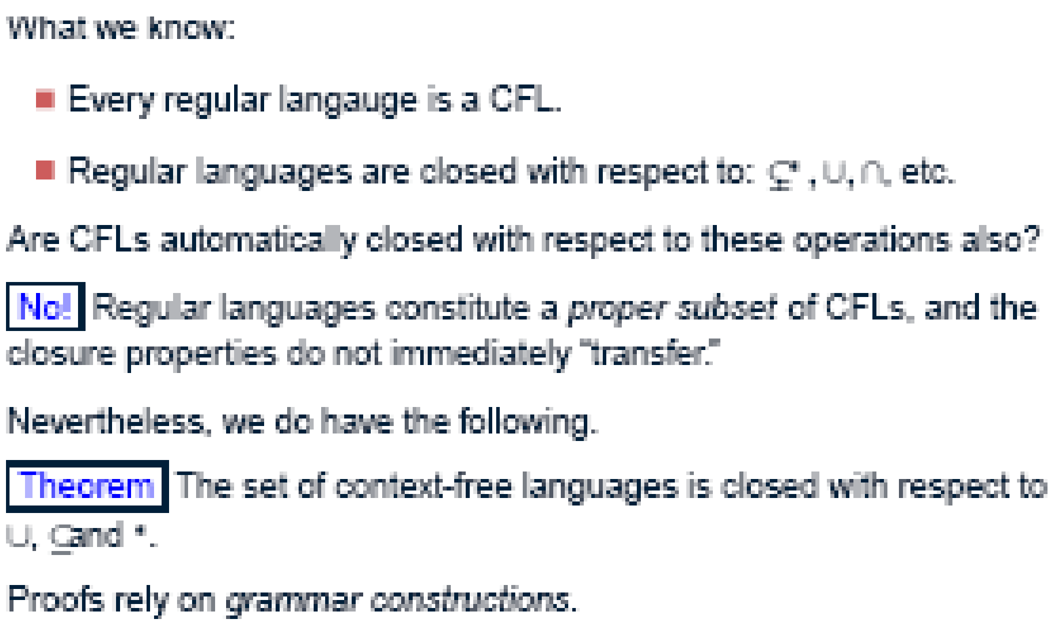
To prove that $\mathcal{L}(M) = \mathcal{L}(G_M)$ we can first argue that:

For every $x \in \Sigma^*, q, q' \in Q$,

$$\delta^*(q, x) = q' \text{ iff } q \Rightarrow^*_{G_M} x\,q'.$$

Then $x \in \mathcal{L}(M)$ iff $x \in \mathcal{L}(G_M)$! (Why?)

Closure Properties of CFL



What we know:

- Every regular langauge is a CFL.
- Regular languages are closed with respect to: ⊆ , ∪, ∩, etc.

Are CFLs automatically closed with respect to these operations also?

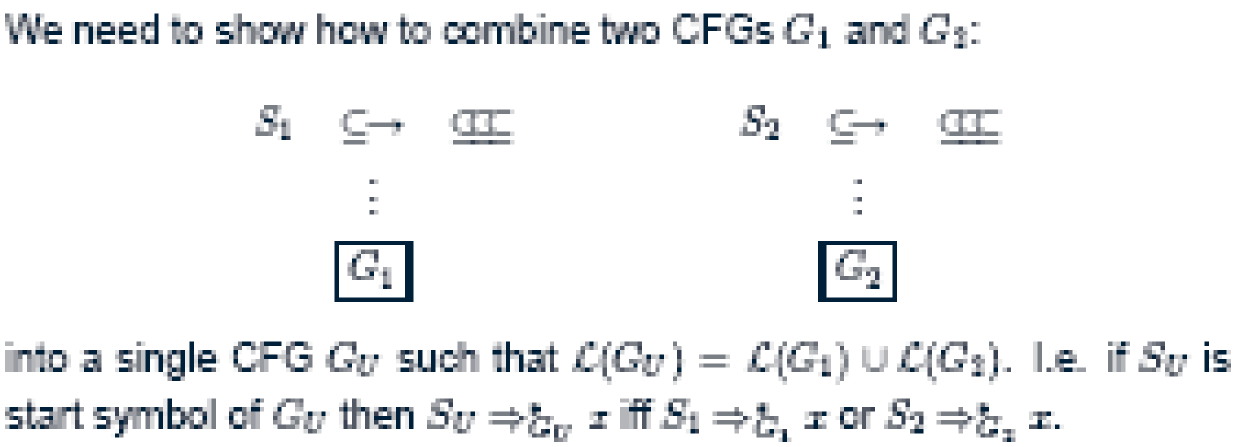
No! Regular languages constitute a *proper subset* of CFLs, and the closure properties do not immediately "transfer."

Nevertheless, we do have the following.

Theorem The set of context-free languages is closed with respect to ∪, ⊆and *.

Proofs rely on *grammar constructions.*

Proving CFLs closed under Union



We need to show how to combine two CFGs G_1 and G_2:

$$S_1 \quad \subseteq\to \quad ⊏⊏ \qquad\qquad S_2 \quad \subseteq\to \quad ⊏⊏$$

$$\vdots \qquad\qquad\qquad \vdots$$

$$\boxed{G_1} \qquad\qquad\qquad \boxed{G_2}$$

into a single CFG G_U such that $\mathcal{L}(G_U) = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$. I.e. if S_U is start symbol of G_U then $S_U \Rightarrow^*_{G_U} x$ iff $S_1 \Rightarrow^*_{G_1} x$ or $S_2 \Rightarrow^*_{G_2} x$.

IDEA

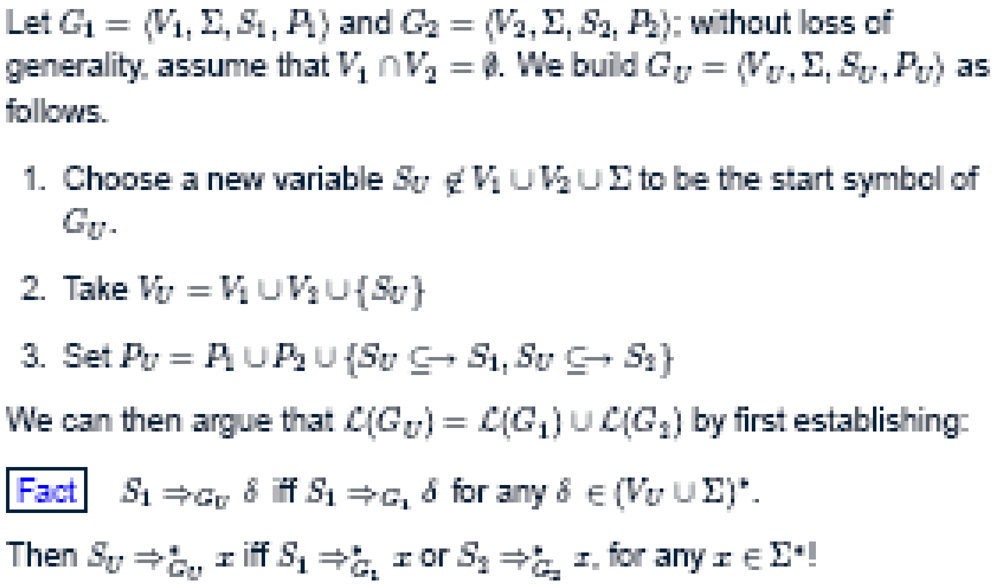
Why not introduce a new variable S_U as follows?

$$\begin{array}{lcl} S_U & \subseteq\!\rightarrow & S_1 \mid S_2 \\ S_1 & \subseteq\!\rightarrow & \underline{\sqsubset\sqsubset\sqsubset} \\ & \vdots & \\ S_2 & \subseteq\!\rightarrow & \underline{\sqsubset\sqsubset\sqsubset} \\ & \vdots & \end{array}$$

- If $S_1 \Rightarrow^*_{G_1} w$ then $S \Rightarrow_{G_U} S_1 \Rightarrow^*_{G_U} x$.
- If $S \Rightarrow^*_{G_U} x$ then $S \Rightarrow_{G_U} S_i \Rightarrow^*_{G_i} x$ for $i = 1$ or $i = 2$.

For the latter to hold we need to ensure that G_1, G_2 don't interfere with one another (i.e. share variables).

Formal Construction of Gu



Let $G_1 = \langle V_1, \Sigma, S_1, P_1 \rangle$ and $G_2 = \langle V_2, \Sigma, S_2, P_2 \rangle$; without loss of generality, assume that $V_1 \cap V_2 = \emptyset$. We build $G_U = \langle V_U, \Sigma, S_U, P_U \rangle$ as follows.

1. Choose a new variable $S_U \notin V_1 \cup V_2 \cup \Sigma$ to be the start symbol of G_U.
2. Take $V_U = V_1 \cup V_2 \cup \{S_U\}$
3. Set $P_U = P_1 \cup P_2 \cup \{S_U \hookrightarrow S_1, S_U \hookrightarrow S_2\}$

We can then argue that $\mathcal{L}(G_U) = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$ by first establishing:

Fact $S_1 \Rightarrow_{G_U} \delta$ iff $S_1 \Rightarrow_{G_1} \delta$ for any $\delta \in (V_U \cup \Sigma)^*$.

Then $S_U \Rightarrow^*_{G_U} x$ iff $S_1 \Rightarrow^*_{G_1} x$ or $S_2 \Rightarrow^*_{G_2} x$, for any $x \in \Sigma^*$!

Derivations

$\alpha A \beta \Rightarrow_G \alpha\gamma\beta$, if there is a production of form $A \rightarrow \gamma$.

We now define $\alpha \Rightarrow_G^* \beta$.

Basis: $\alpha \Rightarrow_G^* \alpha$ for all $\alpha \in (V \cup T)^*$.

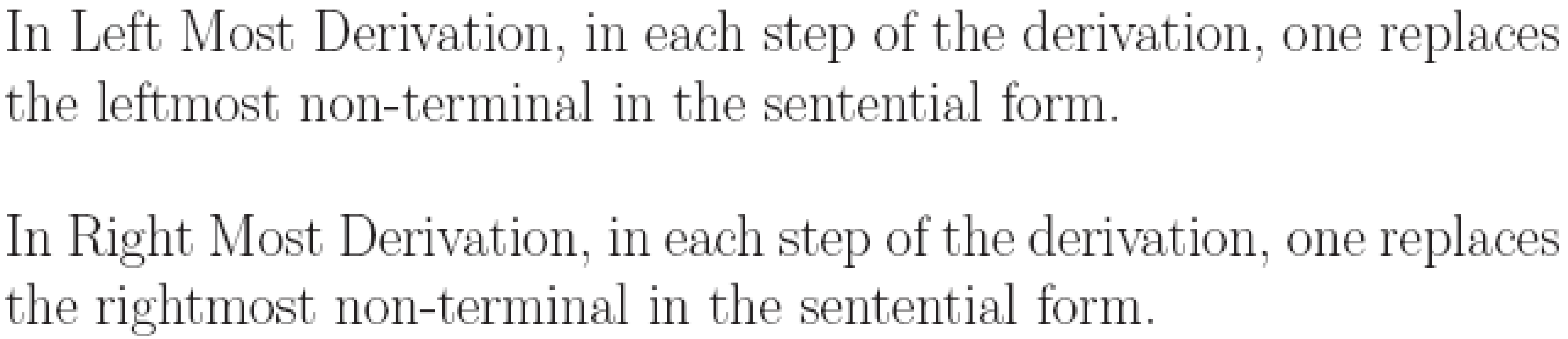
Induction: If $\alpha \Rightarrow_G^* \beta$ and $\beta \Rightarrow_G \gamma$, then $\alpha \Rightarrow_G^* \gamma$.

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

Sentential Form

If $S \Rightarrow^*_G \alpha$, then α is called a sentential form.

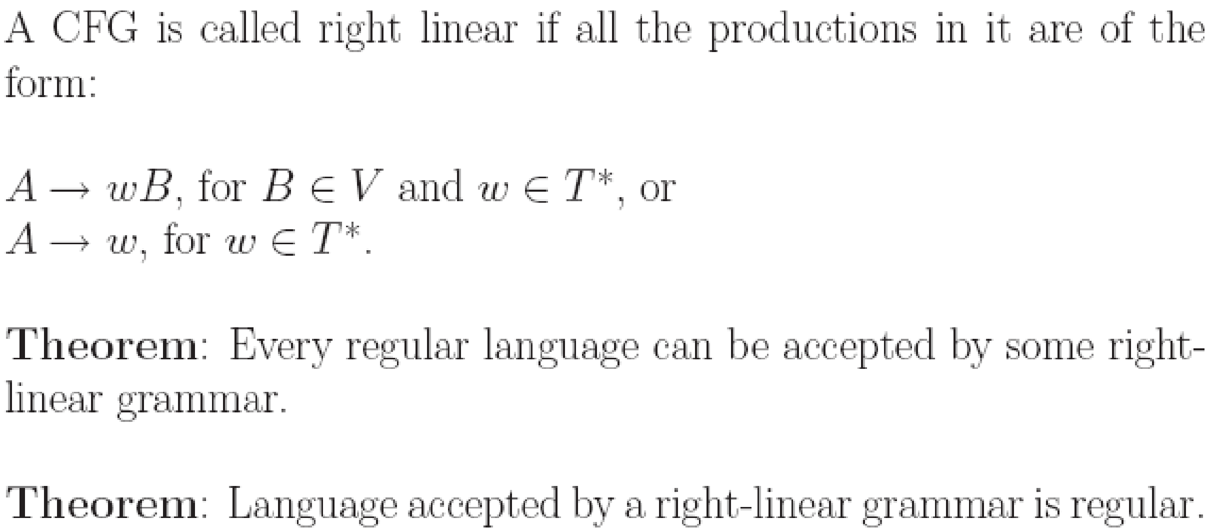
Left Most and Right Most Derivation



In Left Most Derivation, in each step of the derivation, one replaces the leftmost non-terminal in the sentential form.

In Right Most Derivation, in each step of the derivation, one replaces the rightmost non-terminal in the sentential form.

Right-Linear Grammars



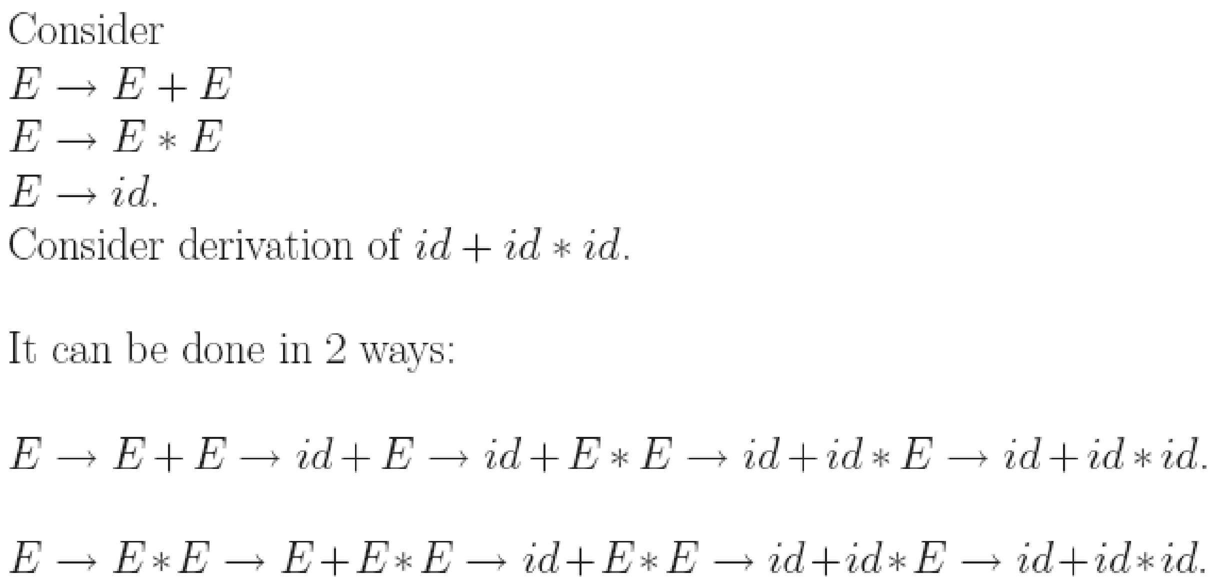
A CFG is called right linear if all the productions in it are of the form:

$A \rightarrow wB$, for $B \in V$ and $w \in T^*$, or
$A \rightarrow w$, for $w \in T^*$.

Theorem: Every regular language can be accepted by some right-linear grammar.

Theorem: Language accepted by a right-linear grammar is regular.

Ambiguous Grammars



Consider

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id.$

Consider derivation of $id + id * id$.

It can be done in 2 ways:

$$E \rightarrow E + E \rightarrow id + E \rightarrow id + E * E \rightarrow id + id * E \rightarrow id + id * id.$$

$$E \rightarrow E * E \rightarrow E + E * E \rightarrow id + E * E \rightarrow id + id * E \rightarrow id + id * id.$$

Ambiguous Grammars

$S \rightarrow S + T$
$S \rightarrow T$
$T \rightarrow T * id$
$T \rightarrow id$

Inherently ambiguous languages.
$L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$.
Any grammar for above language is ambiguous.

Pushdown Automata

Recall our study of regular languages.

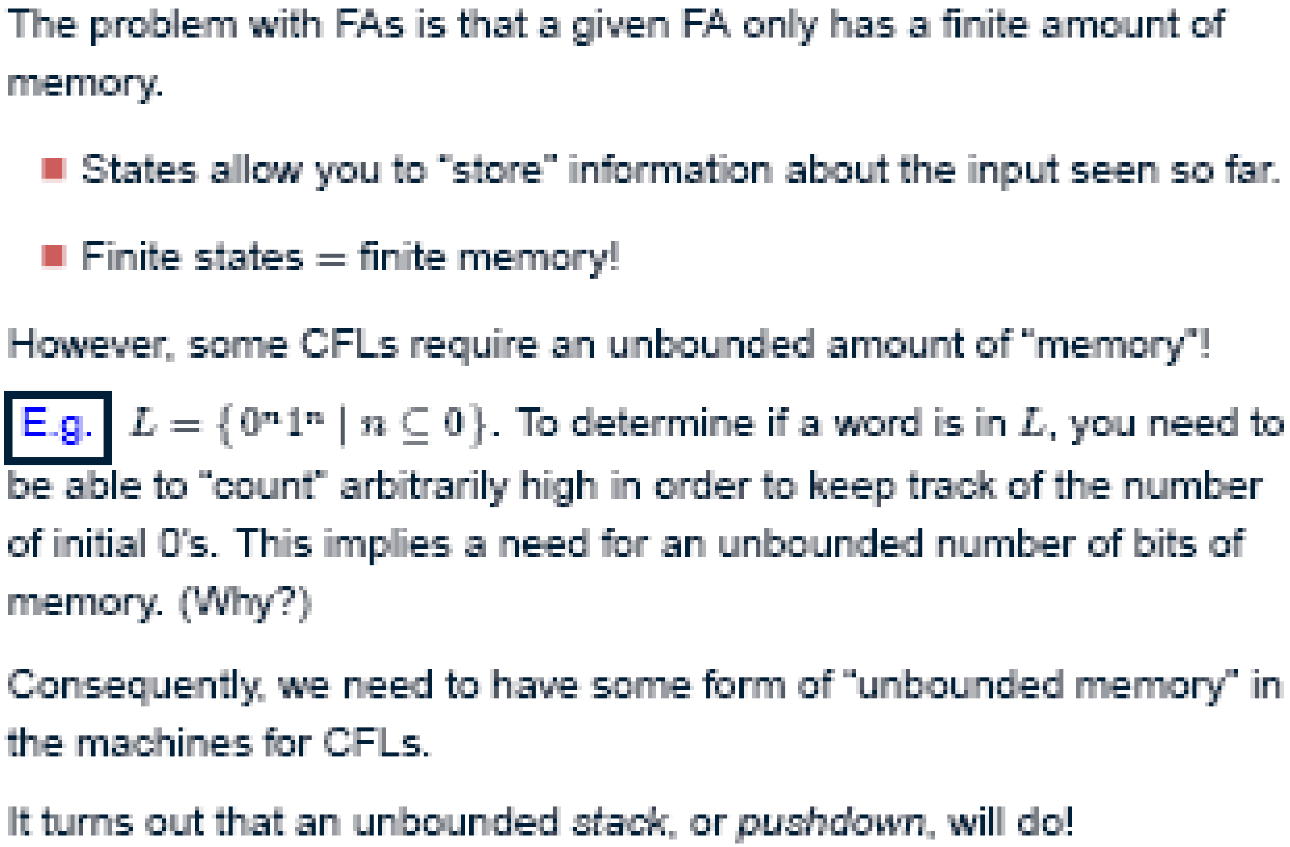
- They were defined in terms of regular expressions (syntax).
- We then showed that FAs provide the computational power needed to process them.

We would like to mimic this line of development for CFLs.

- We have a "syntactic" definition of CFLs in terms of CFGs.
- What kind of computing power is needed to "process" (i.e. recognize) CFLs?

Do FAs suffice?

Cont…



The problem with FAs is that a given FA only has a finite amount of memory.

- States allow you to "store" information about the input seen so far.
- Finite states = finite memory!

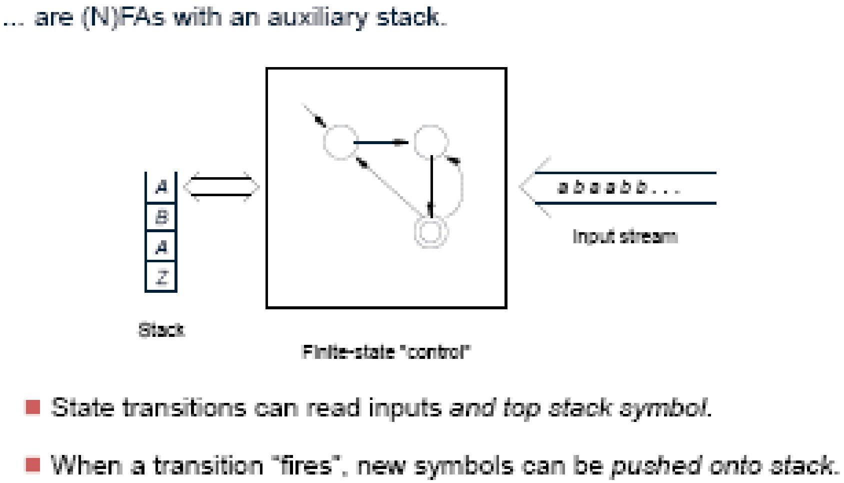
However, some CFLs require an unbounded amount of "memory"!

E.g. $L = \{0^n 1^n \mid n \subseteq 0\}$. To determine if a word is in L, you need to be able to "count" arbitrarily high in order to keep track of the number of initial 0's. This implies a need for an unbounded number of bits of memory. (Why?)

Consequently, we need to have some form of "unbounded memory" in the machines for CFLs.

It turns out that an unbounded *stack*, or *pushdown*, will do!

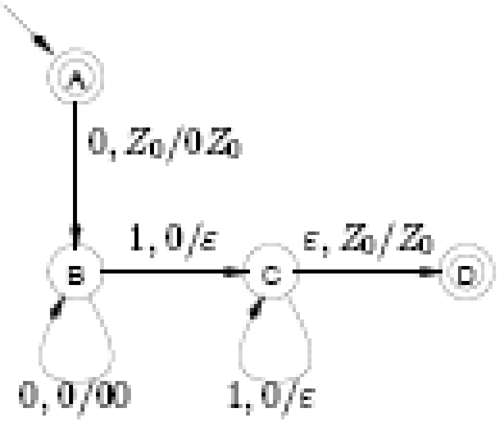
Cont…



... are (N)FAs with an auxiliary stack.

- State transitions can read inputs *and top stack symbol.*
- When a transition "fires", new symbols can be *pushed onto stack.*

Example PDA for $\{0^n1^n \mid n \geq 0\}$



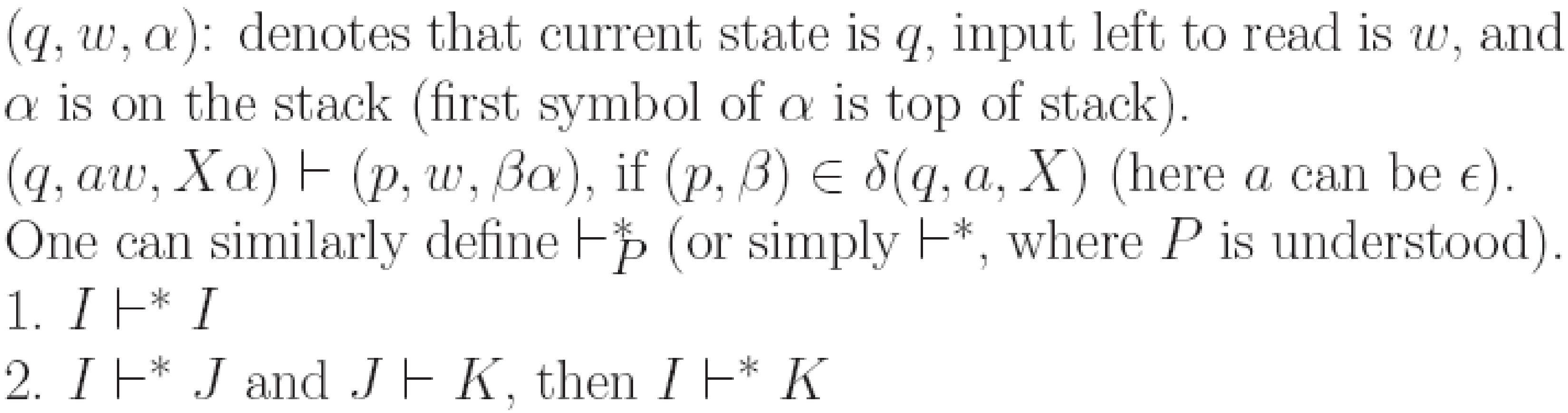
Formal Definition

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

- Q: Finite set of states; q_0 is the start state;
  F is the set of final/accepting states.
- Σ: Alphabet set; Γ: Stack alphabet
- Z_0 is the (only) initial symbol on the stack.
- δ: transition function.

δ takes as input a state q, an input letter a (or ϵ), and a stack symbol (top of stack) X. $\delta(q, a, X)$ is then a finite subset of $Q \times \Gamma^*$. $(p, \gamma) \in \delta(q, a, X)$ denotes that when in state q, reading symbol a (or ϵ), with top of stack being X, the machine's new state is p, X at the top of stack is popped and γ is pushed to the stack. (By convention, if $\gamma = RS$, then S is pushed first, and then R is pushed on the stack).

Instantaneous Description



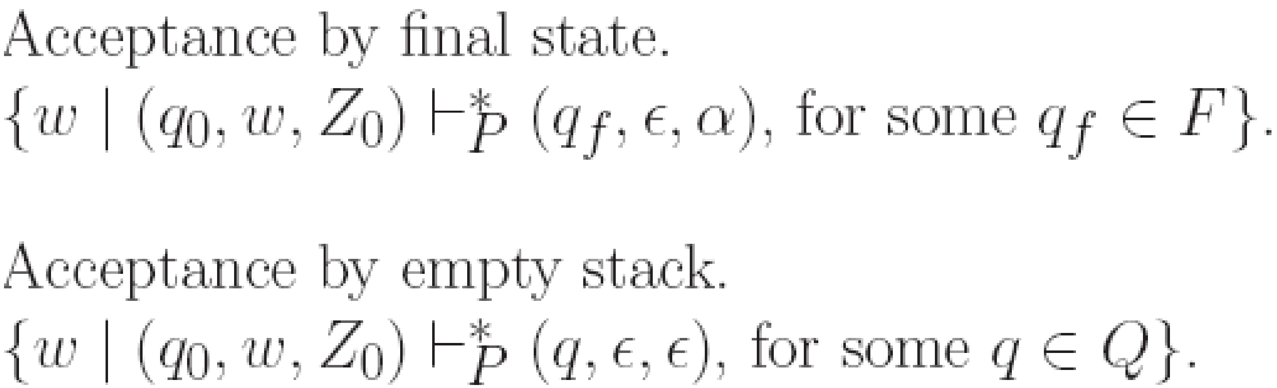
(q, w, α): denotes that current state is q, input left to read is w, and α is on the stack (first symbol of α is top of stack).

$(q, aw, X\alpha) \vdash (p, w, \beta\alpha)$, if $(p, \beta) \in \delta(q, a, X)$ (here a can be ϵ).

One can similarly define $\vdash_P^*$ (or simply $\vdash^*$, where P is understood).

1. $I \vdash^* I$
2. $I \vdash^* J$ and $J \vdash K$, then $I \vdash^* K$

Language accepted by PDA



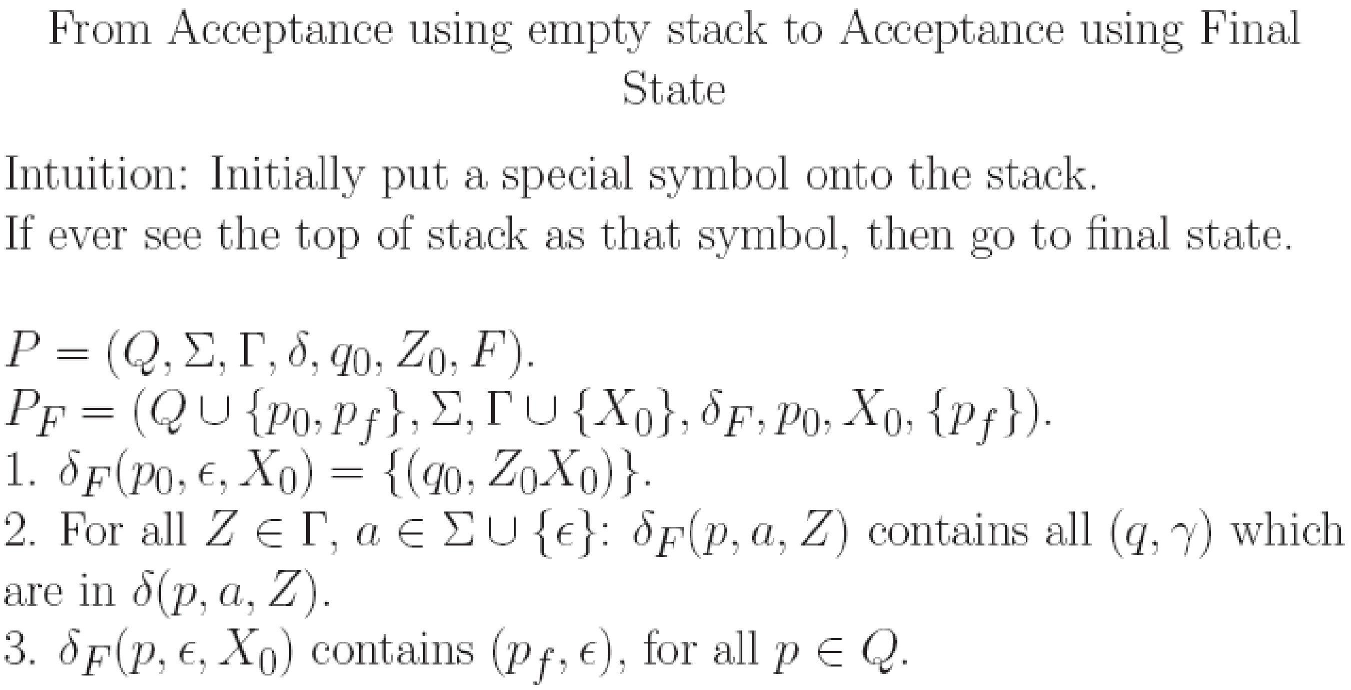
Acceptance by final state.

$\{w \mid (q_0, w, Z_0) \vdash_P^* (q_f, \epsilon, \alpha)$, for some $q_f \in F\}$.

Acceptance by empty stack.

$\{w \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \epsilon)$, for some $q \in Q\}$.

Language accepted by PDA

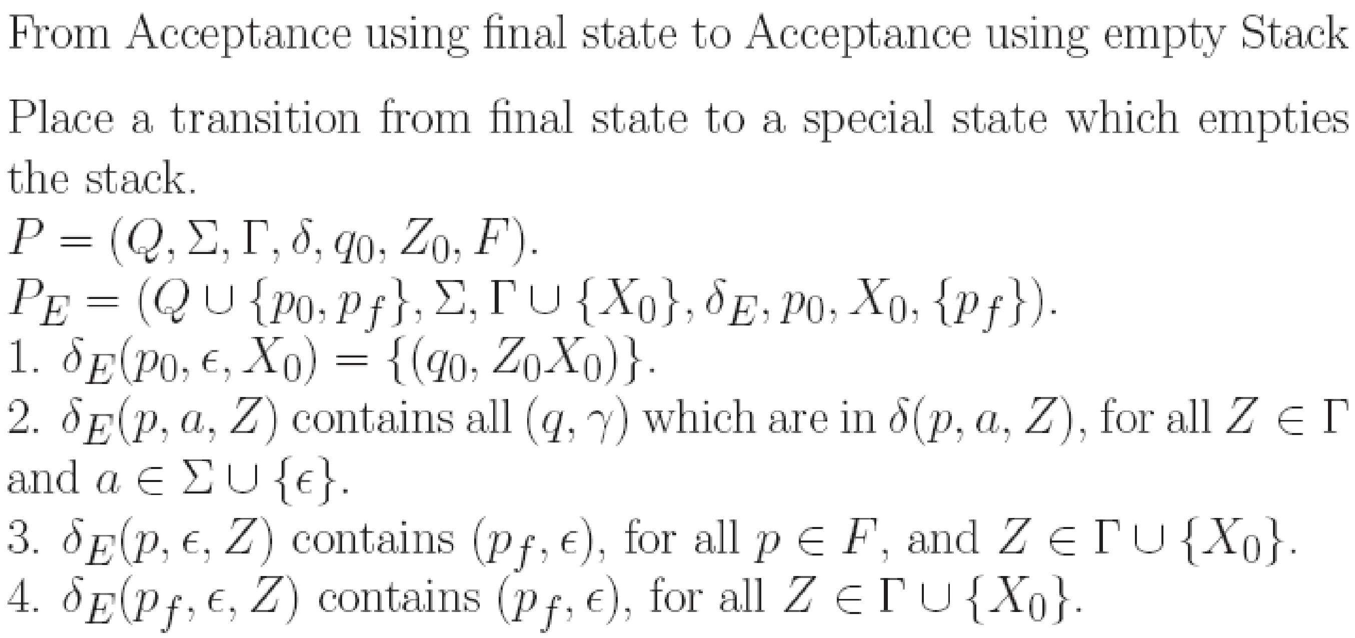


From Acceptance using empty stack to Acceptance using Final State

Intuition: Initially put a special symbol onto the stack.
If ever see the top of stack as that symbol, then go to final state.

$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$.
$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$.
1. $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$.
2. For all $Z \in \Gamma$, $a \in \Sigma \cup \{\epsilon\}$: $\delta_F(p, a, Z)$ contains all (q, γ) which are in $\delta(p, a, Z)$.
3. $\delta_F(p, \epsilon, X_0)$ contains (p_f, ϵ), for all $p \in Q$.

Language accepted by PDA



From Acceptance using final state to Acceptance using empty Stack

Place a transition from final state to a special state which empties the stack.

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

$P_E = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_E, p_0, X_0, \{p_f\})$.

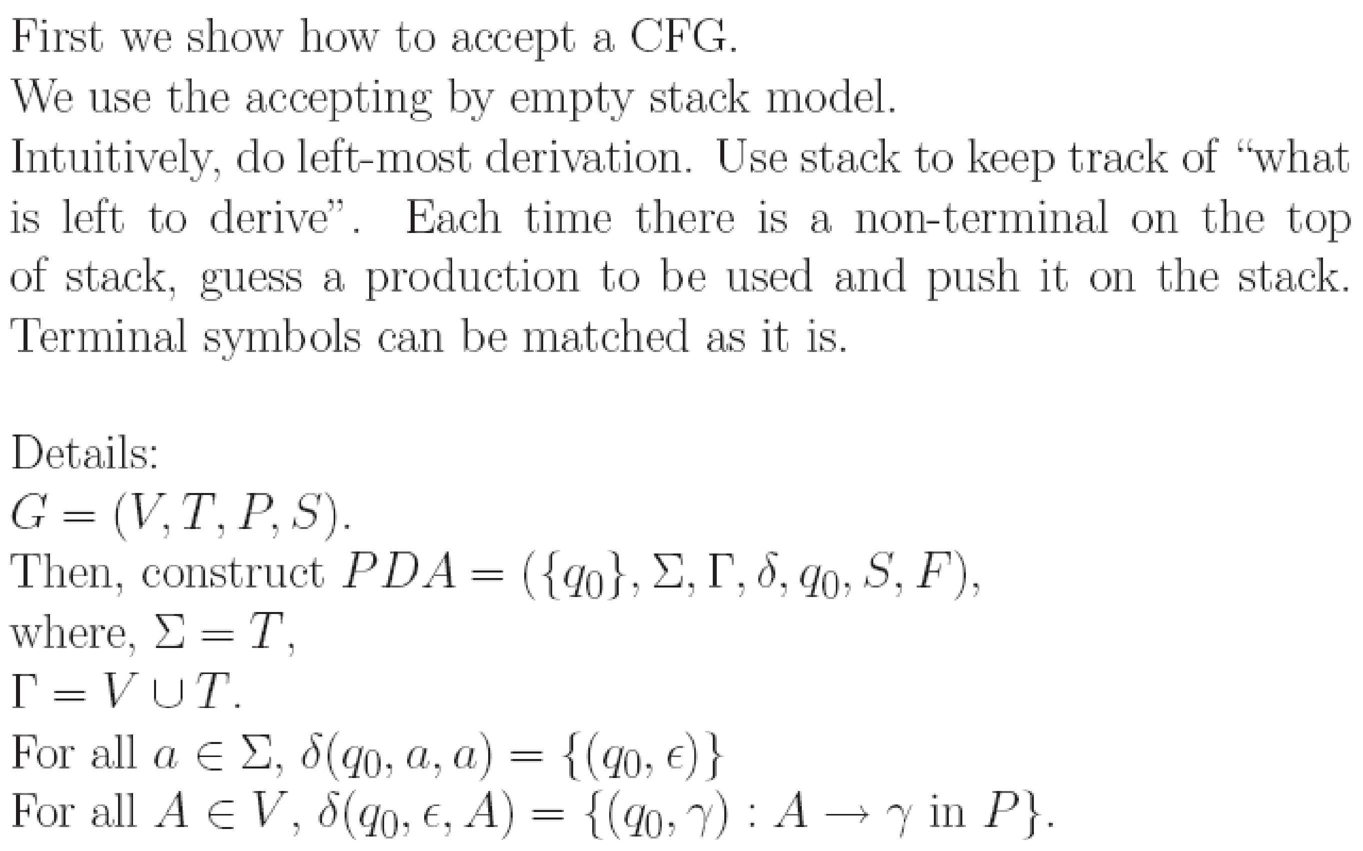
1. $\delta_E(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$.

2. $\delta_E(p, a, Z)$ contains all (q, γ) which are in $\delta(p, a, Z)$, for all $Z \in \Gamma$ and $a \in \Sigma \cup \{\epsilon\}$.

3. $\delta_E(p, \epsilon, Z)$ contains (p_f, ϵ), for all $p \in F$, and $Z \in \Gamma \cup \{X_0\}$.

4. $\delta_E(p_f, \epsilon, Z)$ contains (p_f, ϵ), for all $Z \in \Gamma \cup \{X_0\}$.

Equivalence of CFGs and PDAs



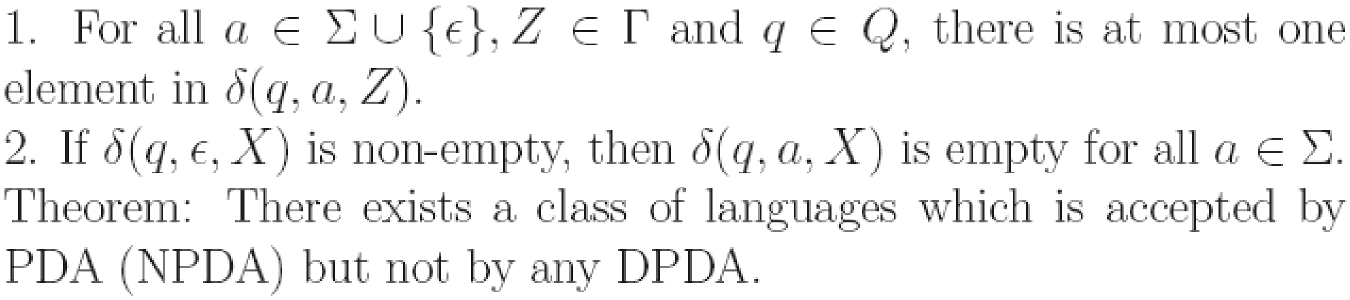
First we show how to accept a CFG.
We use the accepting by empty stack model.
Intuitively, do left-most derivation. Use stack to keep track of "what is left to derive". Each time there is a non-terminal on the top of stack, guess a production to be used and push it on the stack. Terminal symbols can be matched as it is.

Details:
$G = (V, T, P, S)$.
Then, construct $PDA = (\{q_0\}, \Sigma, \Gamma, \delta, q_0, S, F)$,
where, $\Sigma = T$,
$\Gamma = V \cup T$.
For all $a \in \Sigma$, $\delta(q_0, a, a) = \{(q_0, \epsilon)\}$
For all $A \in V$, $\delta(q_0, \epsilon, A) = \{(q_0, \gamma) : A \rightarrow \gamma \text{ in } P\}$.

Cont…

Now we show that each language accepted by a PDA (using empty stack) can be accepted by a CFG.
Suppose PDA is $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$.
We define grammar $G = (V, \Sigma, R, S)$ as follows.
$V = \{S\} \cup \{[qZp] : q, p \in Q, Z \in \Gamma\}$.
$S \rightarrow [q_0 Z_0 p]$, for each $p \in Q$.
If $\delta(q, a, X)$ contains $(r, Y_1 \ldots Y_k)$, then we have productions of form:
$[qXr_k] \rightarrow a[rY_1r_1][r_1Y_2r_2] \ldots [r_{k-1}Y_kr_k]$,
for all $r_1, r_2, \ldots, r_k \in Q$.

Deterministic PDA



1. For all $a \in \Sigma \cup \{\epsilon\}, Z \in \Gamma$ and $q \in Q$, there is at most one element in $\delta(q, a, Z)$.
2. If $\delta(q, \epsilon, X)$ is non-empty, then $\delta(q, a, X)$ is empty for all $a \in \Sigma$.

Theorem: There exists a class of languages which is accepted by PDA (NPDA) but not by any DPDA.

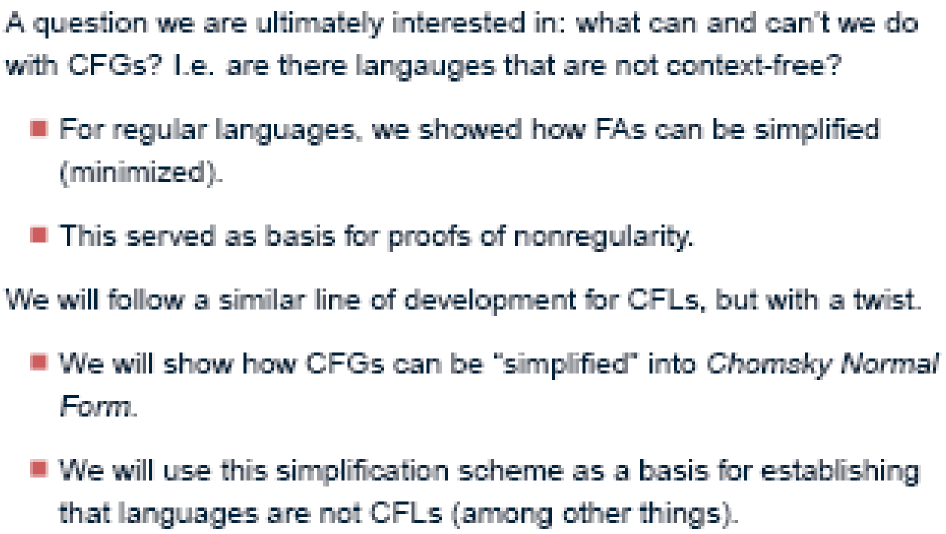
UNIT - IV

PROPERTIES

OF

CONTEXT-FREE LANGUAGES

Chomsky Normal Form



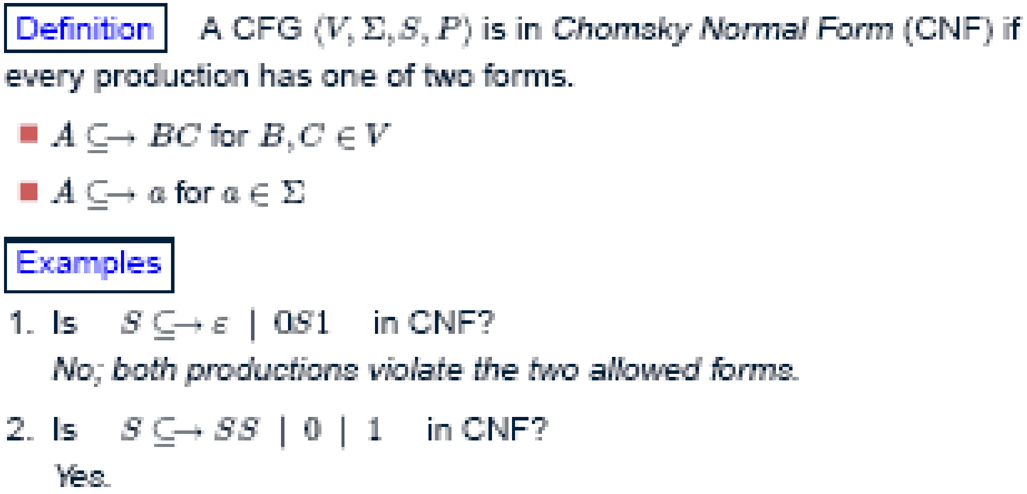
A question we are ultimately interested in: what can and can't we do with CFGs? I.e. are there langauges that are not context-free?

- For regular languages, we showed how FAs can be simplified (minimized).
- This served as basis for proofs of nonregularity.

We will follow a similar line of development for CFLs, but with a twist.

- We will show how CFGs can be "simplified" into *Chomsky Normal Form.*
- We will use this simplification scheme as a basis for establishing that languages are not CFLs (among other things).

Defining CNF



Definition A CFG (V, Σ, S, P) is in *Chomsky Normal Form* (CNF) if every production has one of two forms.

- $A \subseteq\rightarrow BC$ for $B, C \in V$
- $A \subseteq\rightarrow a$ for $a \in \Sigma$

Examples

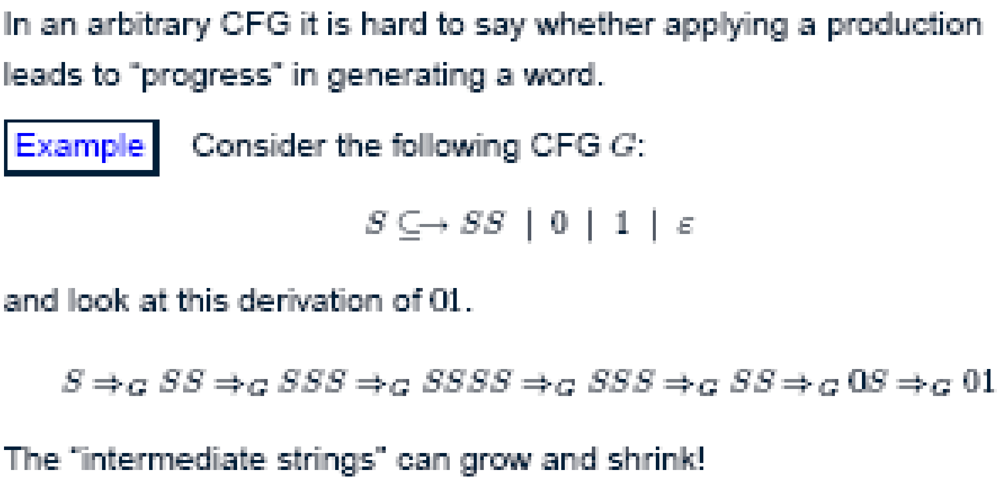
1. Is $S \subseteq\rightarrow \varepsilon \mid 0S1$ in CNF?

   No; both productions violate the two allowed forms.

2. Is $S \subseteq\rightarrow SS \mid 0 \mid 1$ in CNF?

   Yes.

What is the Big Deal about CNF?



In an arbitrary CFG it is hard to say whether applying a production leads to "progress" in generating a word.

Example Consider the following CFG G:

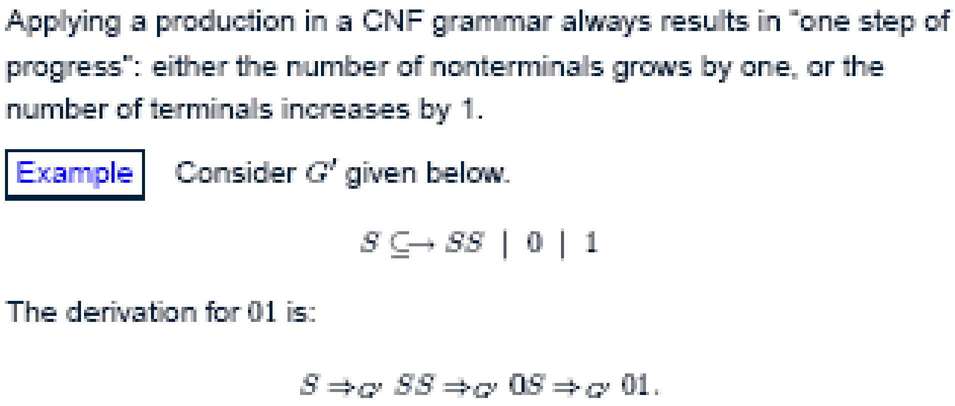
$$S \subseteq\!\!\longrightarrow SS \mid 0 \mid 1 \mid \varepsilon$$

and look at this derivation of 01.

$$S \Rightarrow_G SS \Rightarrow_G SSS \Rightarrow_G SSSS \Rightarrow_G SSS \Rightarrow_G SS \Rightarrow_G 0S \Rightarrow_G 01$$

The "intermediate strings" can grow and shrink!

Cont…



Applying a production in a CNF grammar always results in "one step of progress": either the number of nonterminals grows by one, or the number of terminals increases by 1.

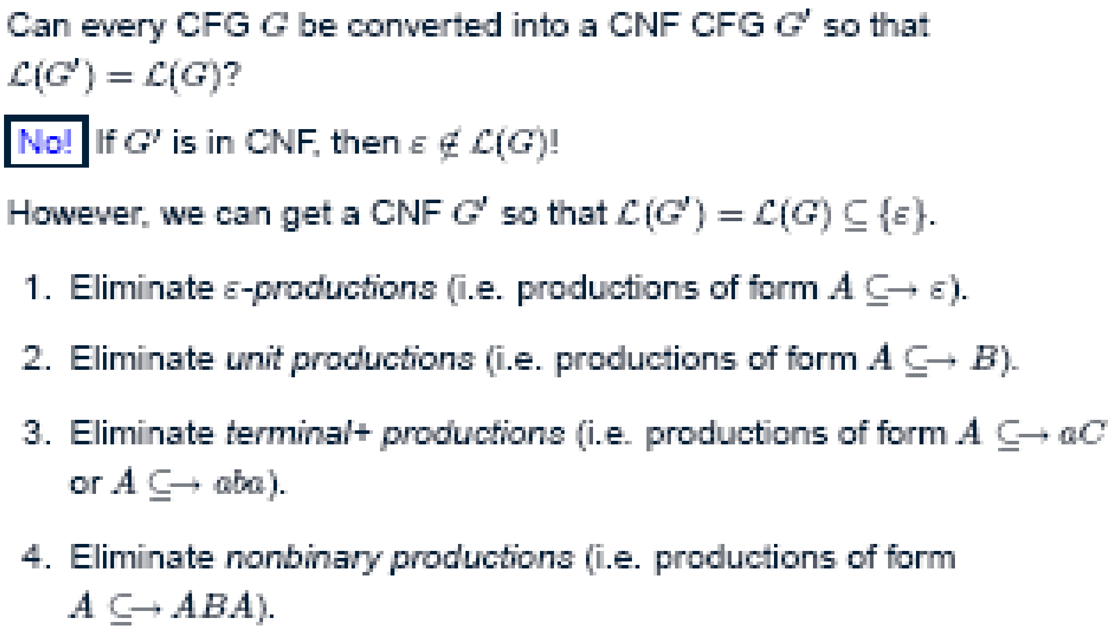
Example Consider G' given below.

$$S \subseteq\!\!\!\longrightarrow SS \mid 0 \mid 1$$

The derivation for 01 is:

$$S \Rightarrow_{G'} SS \Rightarrow_{G'} 0S \Rightarrow_{G'} 01.$$

Converting CFGs into CNF



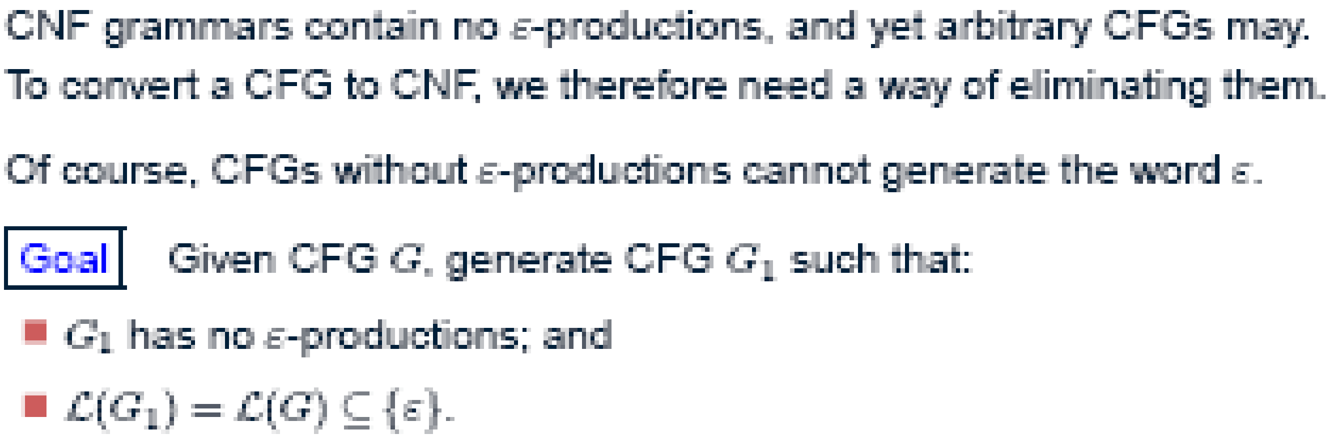
Can every CFG G be converted into a CNF CFG G' so that $\mathcal{L}(G') = \mathcal{L}(G)$?

$\boxed{\text{No!}}$ If G' is in CNF, then $\varepsilon \notin \mathcal{L}(G)$!

However, we can get a CNF G' so that $\mathcal{L}(G') = \mathcal{L}(G) \subseteq \{\varepsilon\}$.

1. Eliminate *ε-productions* (i.e. productions of form $A \subseteq\rightarrow \varepsilon$).
2. Eliminate *unit productions* (i.e. productions of form $A \subseteq\rightarrow B$).
3. Eliminate *terminal+ productions* (i.e. productions of form $A \subseteq\rightarrow aC$ or $A \subseteq\rightarrow aba$).
4. Eliminate *nonbinary productions* (i.e. productions of form $A \subseteq\rightarrow ABA$).

Eliminating ε-Productions



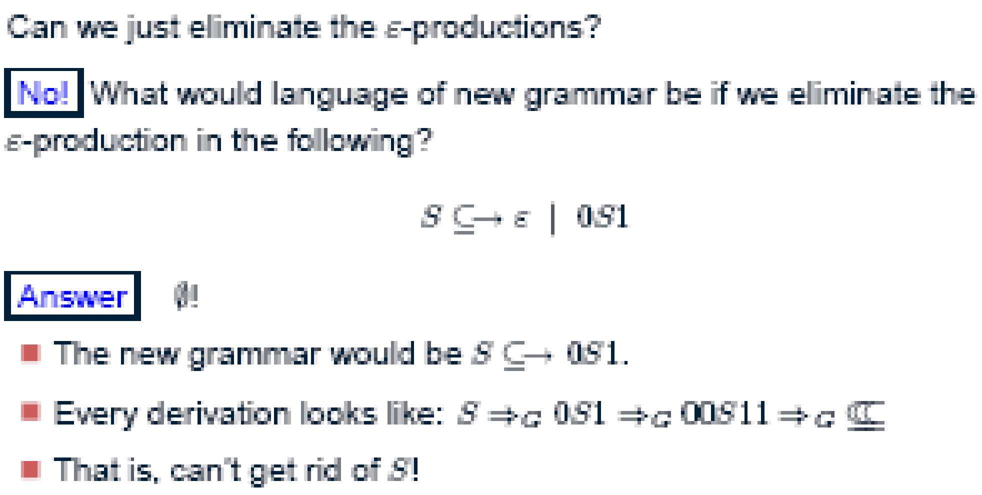
CNF grammars contain no ε-productions, and yet arbitrary CFGs may. To convert a CFG to CNF, we therefore need a way of eliminating them.

Of course, CFGs without ε-productions cannot generate the word ε.

Goal Given CFG G, generate CFG G_1 such that:

- G_1 has no ε-productions; and
- $\mathcal{L}(G_1) = \mathcal{L}(G) \subseteq \{\varepsilon\}$.

Cont…



Can we just eliminate the ε-productions?

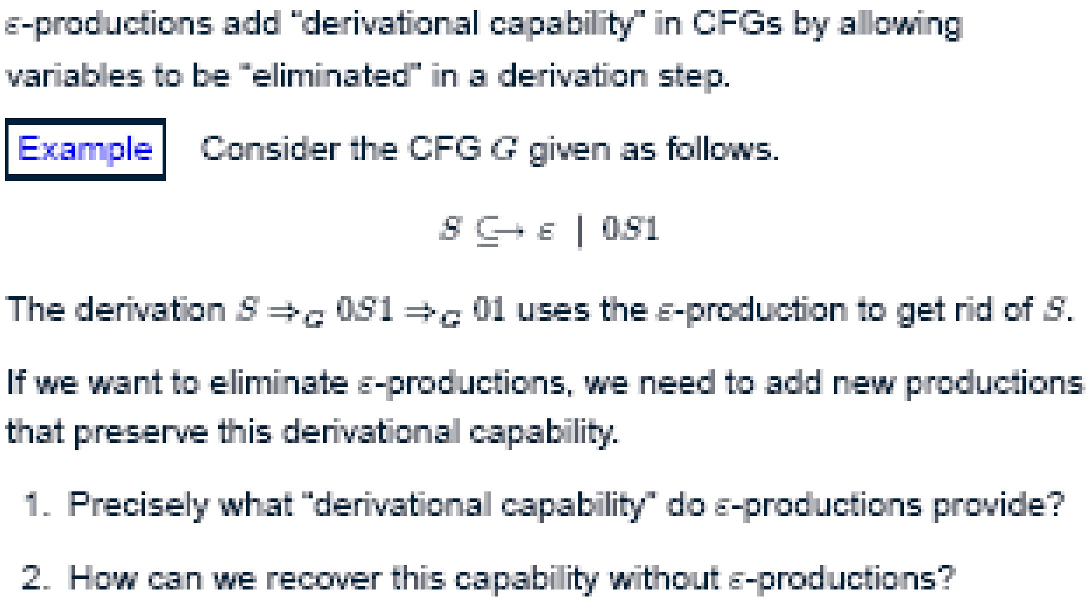
No! What would language of new grammar be if we eliminate the ε-production in the following?

$$S \subseteq\!\rightarrow \varepsilon \mid 0S1$$

Answer $\emptyset$!

- The new grammar would be $S \subseteq\!\rightarrow 0S1$.
- Every derivation looks like: $S \Rightarrow_G 0S1 \Rightarrow_G 00S11 \Rightarrow_G \subseteq\!\subseteq$
- That is, can't get rid of S!

Cont…



ε-productions add "derivational capability" in CFGs by allowing variables to be "eliminated" in a derivation step.

Example Consider the CFG G given as follows.

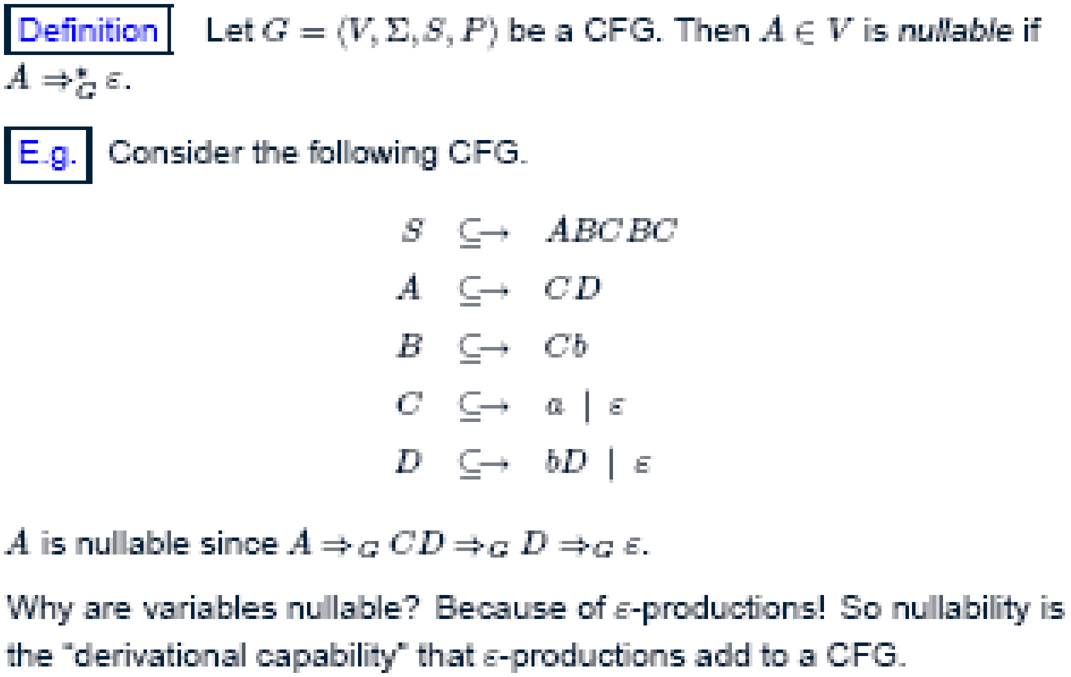
$$S \subseteq\rightarrow \varepsilon \mid 0S1$$

The derivation $S \Rightarrow_G 0S1 \Rightarrow_G 01$ uses the ε-production to get rid of S.

If we want to eliminate ε-productions, we need to add new productions that preserve this derivational capability.

1. Precisely what "derivational capability" do ε-productions provide?
2. How can we recover this capability without ε-productions?

Nullability



Definition Let $G = \langle V, \Sigma, S, P \rangle$ be a CFG. Then $A \in V$ is *nullable* if $A \Rightarrow_G^* \varepsilon$.

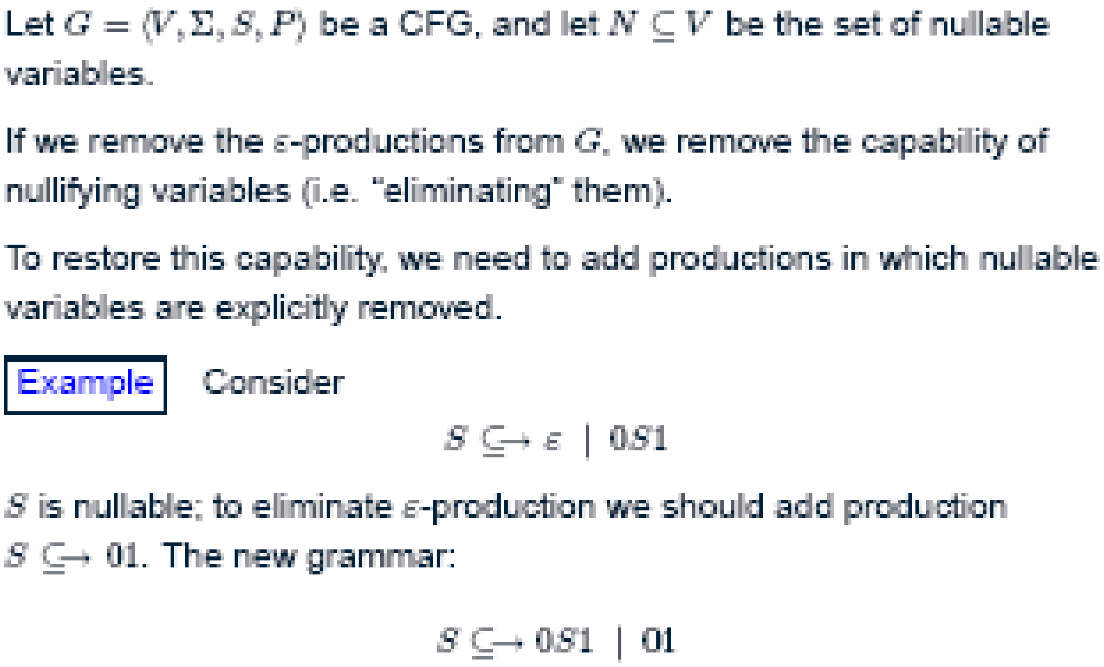
E.g. Consider the following CFG.

$$\begin{aligned} S &\hookrightarrow ABCBC \\ A &\hookrightarrow CD \\ B &\hookrightarrow Cb \\ C &\hookrightarrow a \mid \varepsilon \\ D &\hookrightarrow bD \mid \varepsilon \end{aligned}$$

A is nullable since $A \Rightarrow_G CD \Rightarrow_G D \Rightarrow_G \varepsilon$.

Why are variables nullable? Because of ε-productions! So nullability is the "derivational capability" that ε-productions add to a CFG.

Generating an ε-Production-free CFGs



Let $G = (V, \Sigma, S, P)$ be a CFG, and let $N \subseteq V$ be the set of nullable variables.

If we remove the ε-productions from G, we remove the capability of nullifying variables (i.e. "eliminating" them).

To restore this capability, we need to add productions in which nullable variables are explicitly removed.

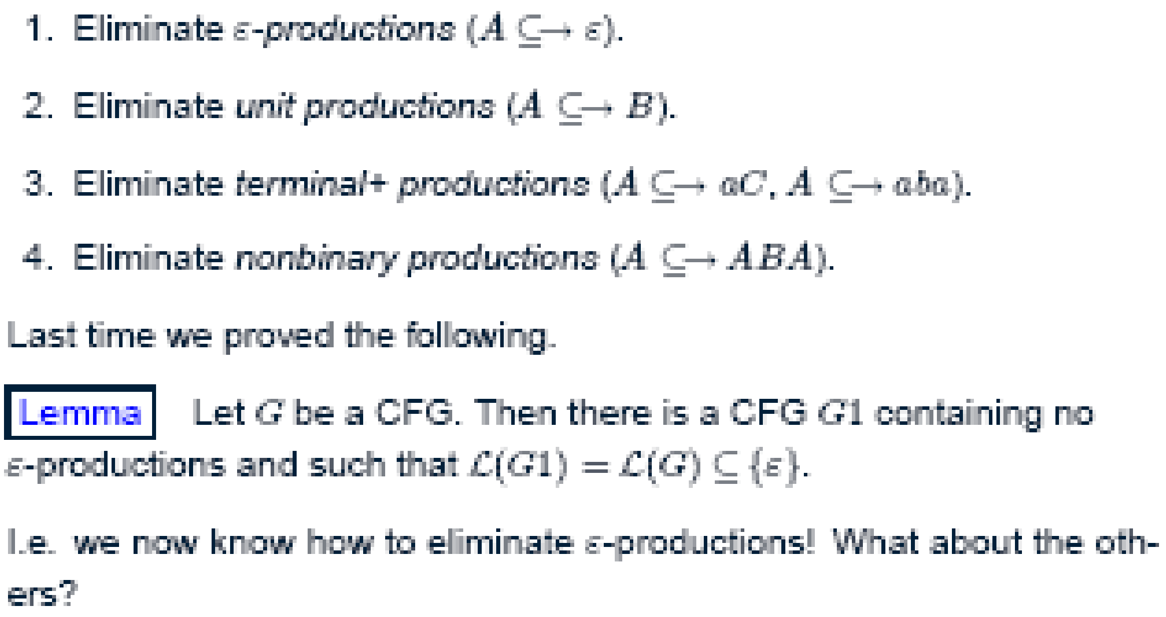
Example Consider

$$S \subsetneq\!\rightarrow \varepsilon \mid 0S1$$

S is nullable; to eliminate ε-production we should add production $S \subsetneq\!\rightarrow 01$. The new grammar:

$$S \subsetneq\!\rightarrow 0S1 \mid 01$$

Converting CFGs into CNF



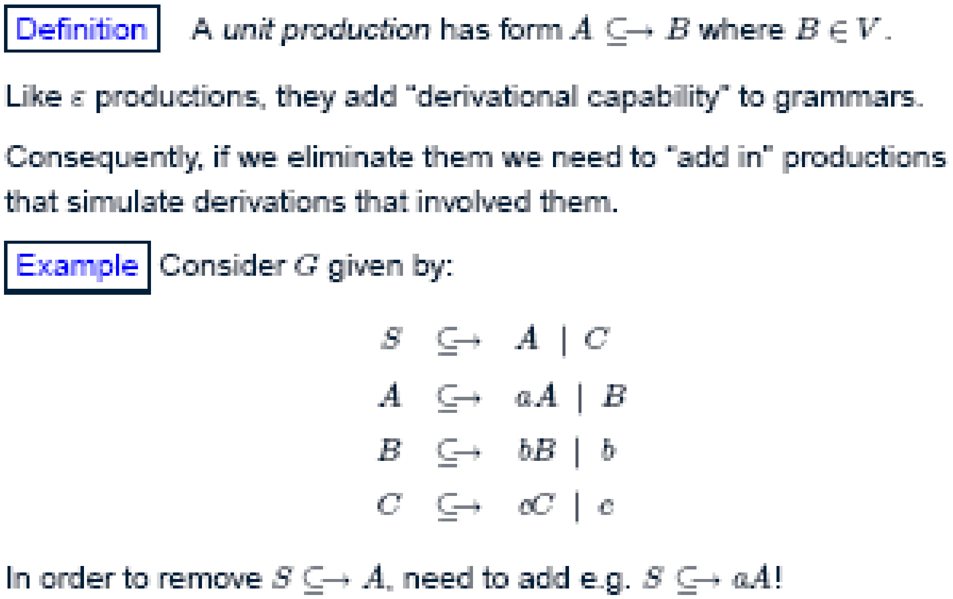
1. Eliminate ε-*productions* ($A \subseteq\rightarrow \varepsilon$).
2. Eliminate *unit productions* ($A \subseteq\rightarrow B$).
3. Eliminate *terminal+ productions* ($A \subseteq\rightarrow aC$, $A \subseteq\rightarrow aba$).
4. Eliminate *nonbinary productions* ($A \subseteq\rightarrow ABA$).

Last time we proved the following.

Lemma Let G be a CFG. Then there is a CFG $G1$ containing no ε-productions and such that $\mathcal{L}(G1) = \mathcal{L}(G) \subseteq \{\varepsilon\}$.

I.e. we now know how to eliminate ε-productions! What about the others?

Eliminating Unit Productions



Definition A *unit production* has form $A \subsetneq\rightarrow B$ where $B \in V$.

Like ε productions, they add "derivational capability" to grammars.

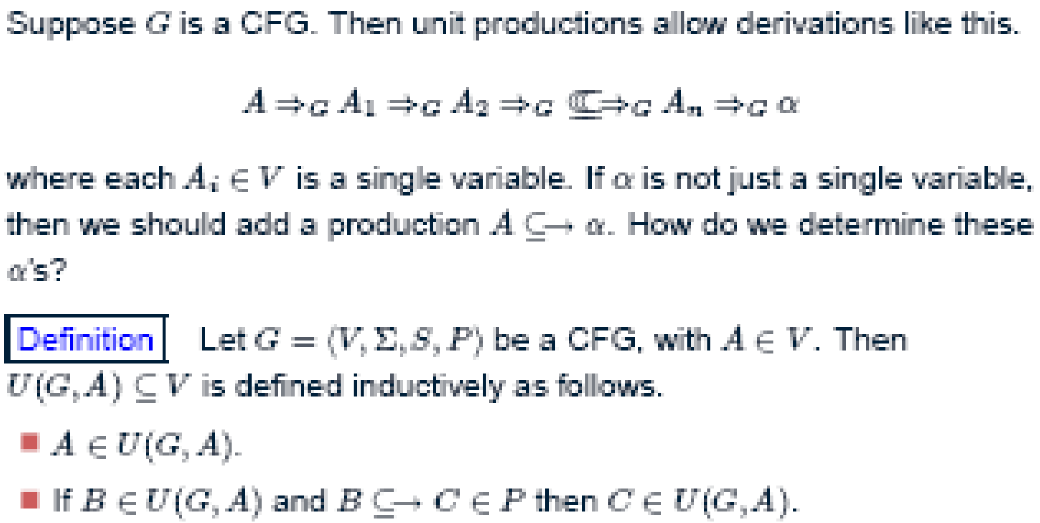
Consequently, if we eliminate them we need to "add in" productions that simulate derivations that involved them.

Example Consider G given by:

$$\begin{array}{lcl} S & \subsetneq\rightarrow & A \mid C \\ A & \subsetneq\rightarrow & aA \mid B \\ B & \subsetneq\rightarrow & bB \mid b \\ C & \subsetneq\rightarrow & cC \mid c \end{array}$$

In order to remove $S \subsetneq\rightarrow A$, need to add e.g. $S \subsetneq\rightarrow aA$!

Cont…



Suppose G is a CFG. Then unit productions allow derivations like this.

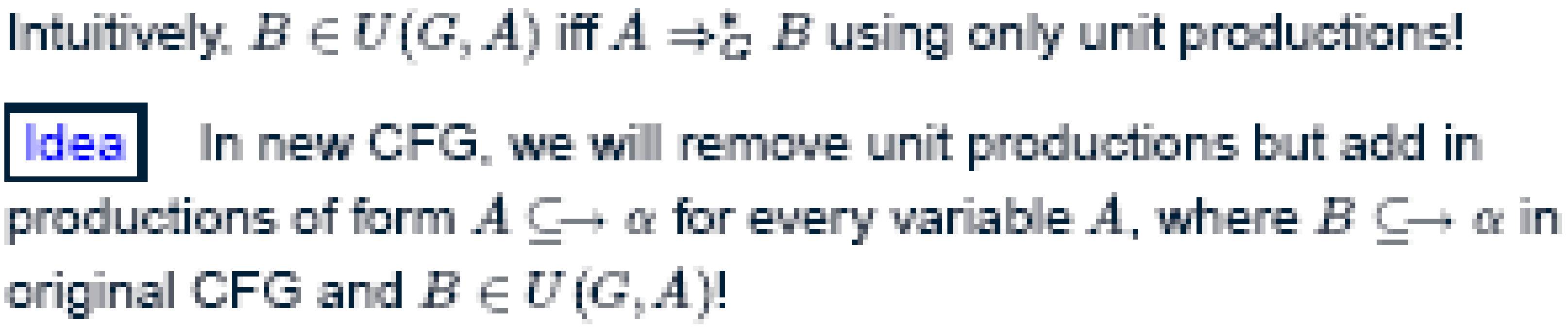
$$A \Rightarrow_G A_1 \Rightarrow_G A_2 \Rightarrow_G \subseteq\!\Rightarrow_G A_n \Rightarrow_G \alpha$$

where each $A_i \in V$ is a single variable. If α is not just a single variable, then we should add a production $A \subseteq\!\rightarrow \alpha$. How do we determine these α's?

Definition Let $G = (V, \Sigma, S, P)$ be a CFG, with $A \in V$. Then $U(G, A) \subseteq V$ is defined inductively as follows.

- $A \in U(G, A)$.
- If $B \in U(G, A)$ and $B \subseteq\!\rightarrow C \in P$ then $C \in U(G, A)$.

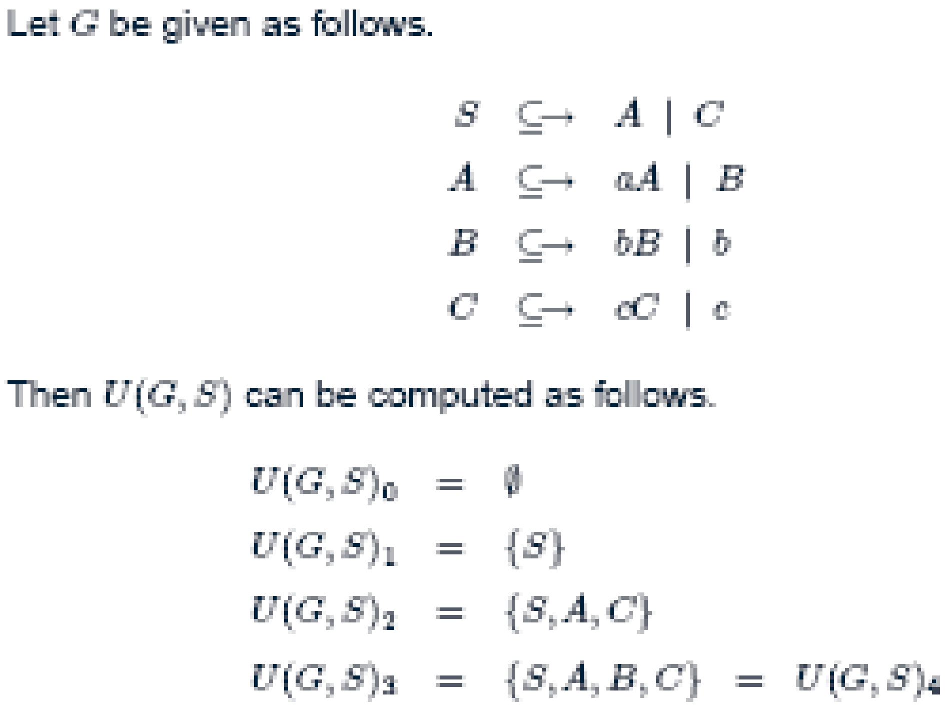
U(G,A) and New Productions



Intuitively, $B \in U(G, A)$ iff $A \Rightarrow_G^* B$ using only unit productions!

Idea In new CFG, we will remove unit productions but add in productions of form $A \subseteq\rightarrow \alpha$ for every variable A, where $B \subseteq\rightarrow \alpha$ in original CFG and $B \in U(G, A)$!

Example



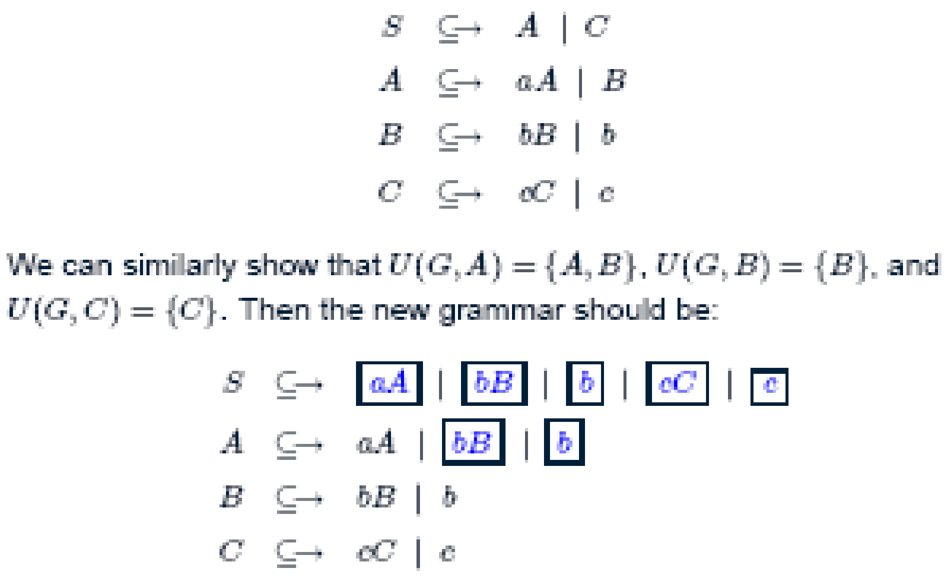
Let G be given as follows.

$$\begin{aligned} S &\subseteq\!\rightarrow A \mid C \\ A &\subseteq\!\rightarrow aA \mid B \\ B &\subseteq\!\rightarrow bB \mid b \\ C &\subseteq\!\rightarrow cC \mid c \end{aligned}$$

Then $U(G,S)$ can be computed as follows.

$$\begin{aligned} U(G,S)_0 &= \emptyset \\ U(G,S)_1 &= \{S\} \\ U(G,S)_2 &= \{S,A,C\} \\ U(G,S)_3 &= \{S,A,B,C\} = U(G,S)_4 \end{aligned}$$

Cont…

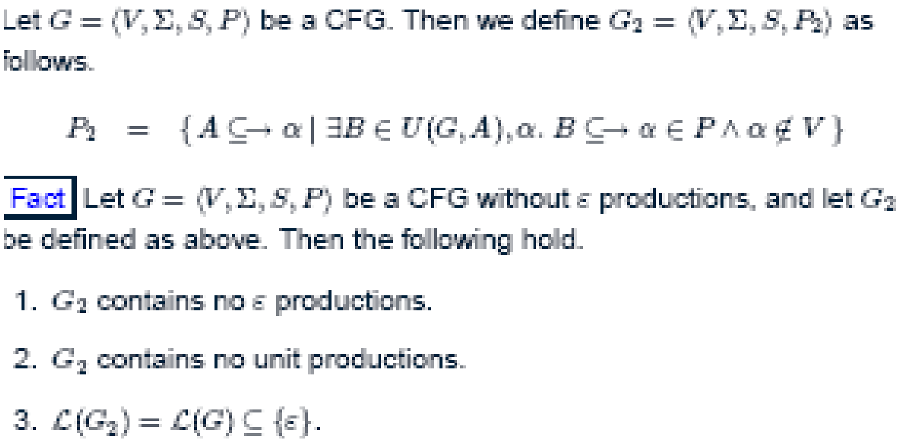


$$\begin{array}{lcl} S & \subsetneq\!\rightarrow & A \mid C \\ A & \subsetneq\!\rightarrow & aA \mid B \\ B & \subsetneq\!\rightarrow & bB \mid b \\ C & \subsetneq\!\rightarrow & cC \mid c \end{array}$$

We can similarly show that $U(G,A) = \{A, B\}$, $U(G,B) = \{B\}$, and $U(G,C) = \{C\}$. Then the new grammar should be:

$$\begin{array}{lcl} S & \subsetneq\!\rightarrow & \boxed{aA} \mid \boxed{bB} \mid \boxed{b} \mid \boxed{cC} \mid \boxed{c} \\ A & \subsetneq\!\rightarrow & aA \mid \boxed{bB} \mid \boxed{b} \\ B & \subsetneq\!\rightarrow & bB \mid b \\ C & \subsetneq\!\rightarrow & cC \mid c \end{array}$$

Formal Construction



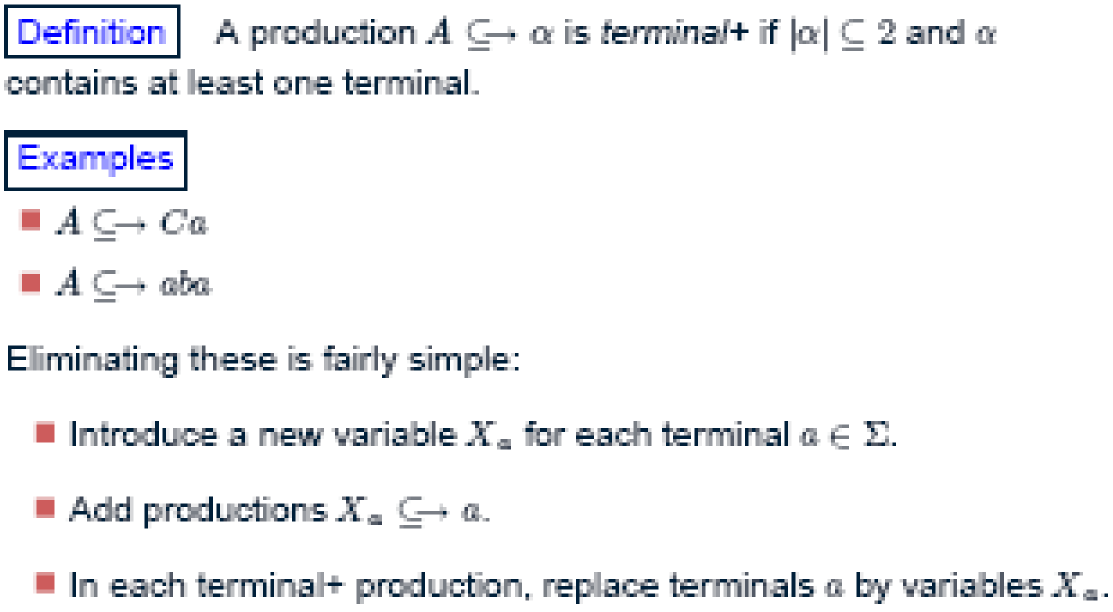
Let $G = \langle V, \Sigma, S, P \rangle$ be a CFG. Then we define $G_2 = \langle V, \Sigma, S, P_2 \rangle$ as follows.

$$P_2 \quad = \quad \{\, A \subseteq\!\!\rightarrow \alpha \mid \exists B \in U(G, A), \alpha.\ B \subseteq\!\!\rightarrow \alpha \in P \wedge \alpha \notin V \,\}$$

Fact Let $G = \langle V, \Sigma, S, P \rangle$ be a CFG without ε productions, and let G_2 be defined as above. Then the following hold.

1. G_2 contains no ε productions.
2. G_2 contains no unit productions.
3. $\mathcal{L}(G_2) = \mathcal{L}(G) \subseteq \{\varepsilon\}$.

Eliminating Terminal Productions



Definition A production $A \subseteq\!\!\to \alpha$ is *terminal+* if $|\alpha| \subseteq 2$ and α contains at least one terminal.

Examples

- $A \subseteq\!\!\to Ca$
- $A \subseteq\!\!\to aba$

Eliminating these is fairly simple:

- Introduce a new variable X_a for each terminal $a \in \Sigma$.
- Add productions $X_a \subseteq\!\!\to a$.
- In each terminal+ production, replace terminals a by variables X_a.

Example

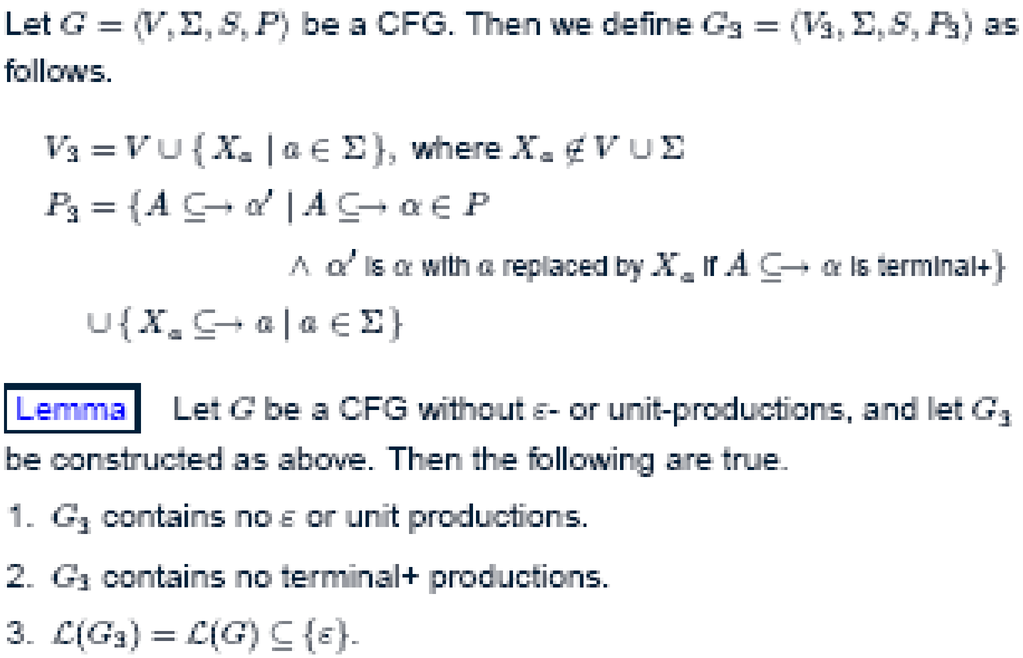
Let G be given by:

$$S \subseteq\!\rightarrow aSb \mid aS \mid Sb \mid a \mid b$$

Then G_3 is:

$$\begin{aligned} S &\subseteq\!\rightarrow X_aSX_b \mid X_aS \mid SX_b \mid a \mid b \\ X_a &\subseteq\!\rightarrow a \\ X_b &\subseteq\!\rightarrow b \end{aligned}$$

Formal Construction



Let $G = \langle V, \Sigma, S, P\rangle$ be a CFG. Then we define $G_3 = \langle V_3, \Sigma, S, P_3\rangle$ as follows.

$$V_3 = V \cup \{\, X_a \mid a \in \Sigma \,\}, \text{ where } X_a \notin V \cup \Sigma$$

$$\begin{aligned} P_3 = \{A \hookrightarrow \alpha' \mid A \hookrightarrow \alpha \in P \\ \wedge\ \alpha' \text{ is } \alpha \text{ with } a \text{ replaced by } X_a \text{ if } A \hookrightarrow \alpha \text{ is terminal+}\} \\ \cup \{\, X_a \hookrightarrow a \mid a \in \Sigma \,\} \end{aligned}$$

Lemma Let G be a CFG without ε- or unit-productions, and let G_3 be constructed as above. Then the following are true.

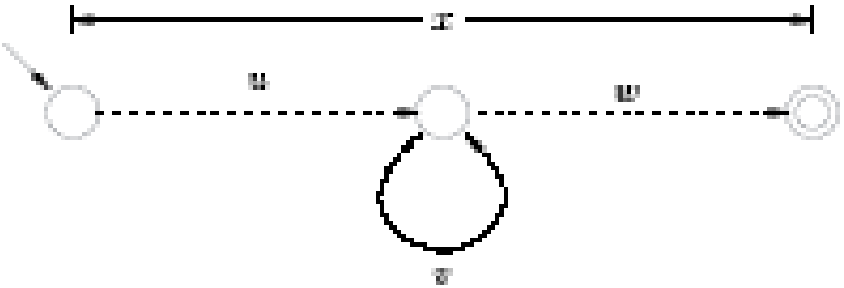
1. G_3 contains no ε or unit productions.
2. G_3 contains no terminal+ productions.
3. $\mathcal{L}(G_3) = \mathcal{L}(G) \subseteq \{\varepsilon\}$.

Pumping Lemma For CFLs

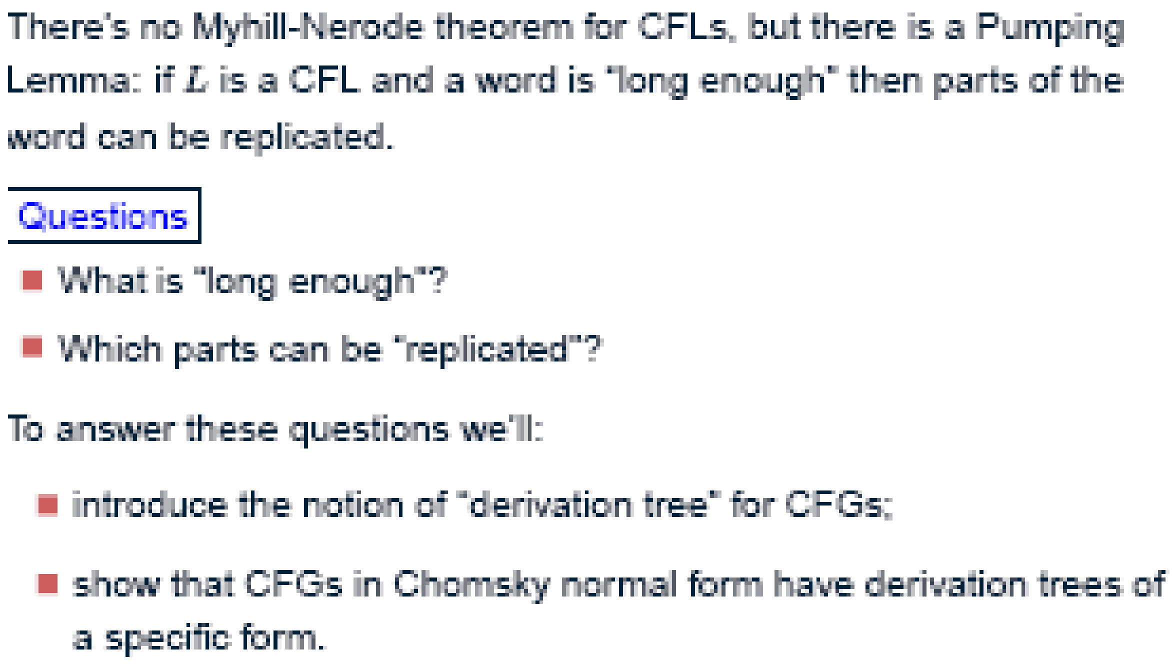
Recall how we proved languages to be nonregular.

Myhill-Nerode: A language L is regular iff its indistinguishability relation I_L has finitely many equivalence classes.

Pumping Lemma: If L is regular, and $x \in L$ is "long enough", then x can be split into u, v, w so that $uv^i w \in L$ all i.



Cont…



There's no Myhill-Nerode theorem for CFLs, but there is a Pumping Lemma: if L is a CFL and a word is "long enough" then parts of the word can be replicated.

Questions

- What is "long enough"?
- Which parts can be "replicated"?

To answer these questions we'll:

- introduce the notion of "derivation tree" for CFGs;
- show that CFGs in Chomsky normal form have derivation trees of a specific form.

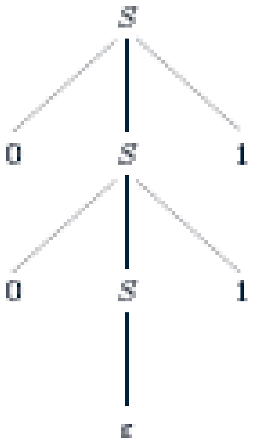
Derivation Trees

"Derivation sequences" show how CFGs generate words.

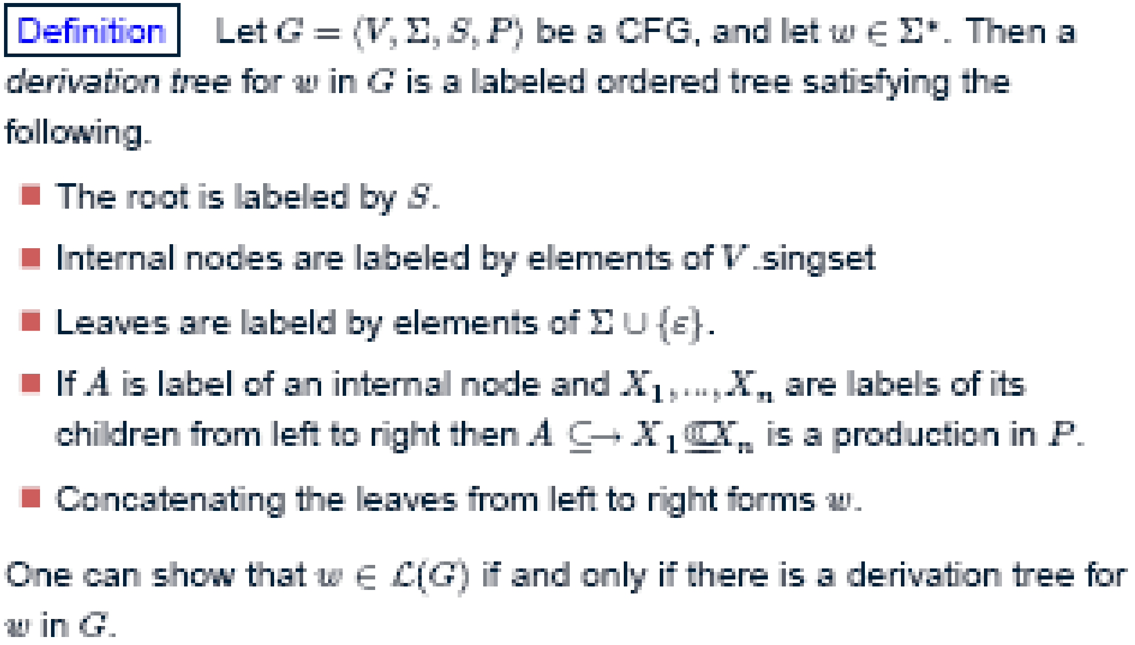
Example Let G be $S \subseteq\!\rightarrow \varepsilon \mid 0S1$. Then to show that G generates 0011:

$$S \Rightarrow_G 0S1 \Rightarrow_G 00S11 \Rightarrow_G 00\ \underline{\subset\!\varepsilon}\ \underline{\subset}11 = 0011$$

A *derivation tree* is a tree-like representation of a derivation sequence.



Defining Derivation Tree



Definition Let $G = \langle V, \Sigma, S, P \rangle$ be a CFG, and let $w \in \Sigma^*$. Then a *derivation tree* for w in G is a labeled ordered tree satisfying the following.

- The root is labeled by S.
- Internal nodes are labeled by elements of V.singset
- Leaves are labeld by elements of $\Sigma \cup \{\varepsilon\}$.
- If A is label of an internal node and $X_1, \ldots, X_n$ are labels of its children from left to right then $A \subseteq\!\rightarrow X_1 \underline{\mathbb{C}}X_n$ is a production in P.
- Concatenating the leaves from left to right forms w.

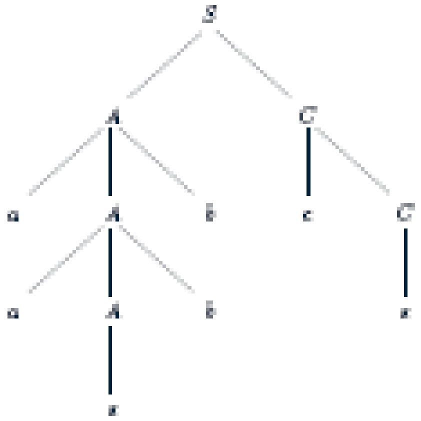
One can show that $w \in \mathcal{L}(G)$ if and only if there is a derivation tree for w in G.

Example

Let G be:

$$\begin{aligned} S &\hookrightarrow AC \\ A &\hookrightarrow aAb \mid \varepsilon \\ C &\hookrightarrow cC \mid \varepsilon \end{aligned}$$

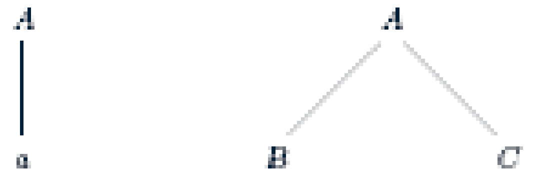
Then a derivation tree for $aabbc$ is:



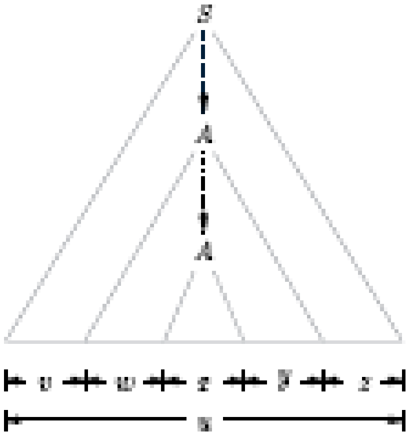
Derivation Trees and CNF

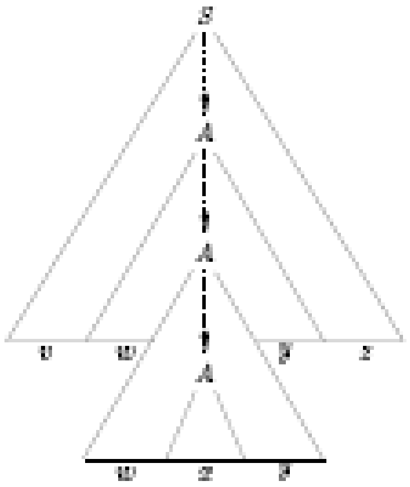
Suppose G is in CNF; what property do the derivation trees for words have?

- No leaves are labeled by ε.
- Every internal node has either one child, which must be a leaf, or two children, which must both be internal.



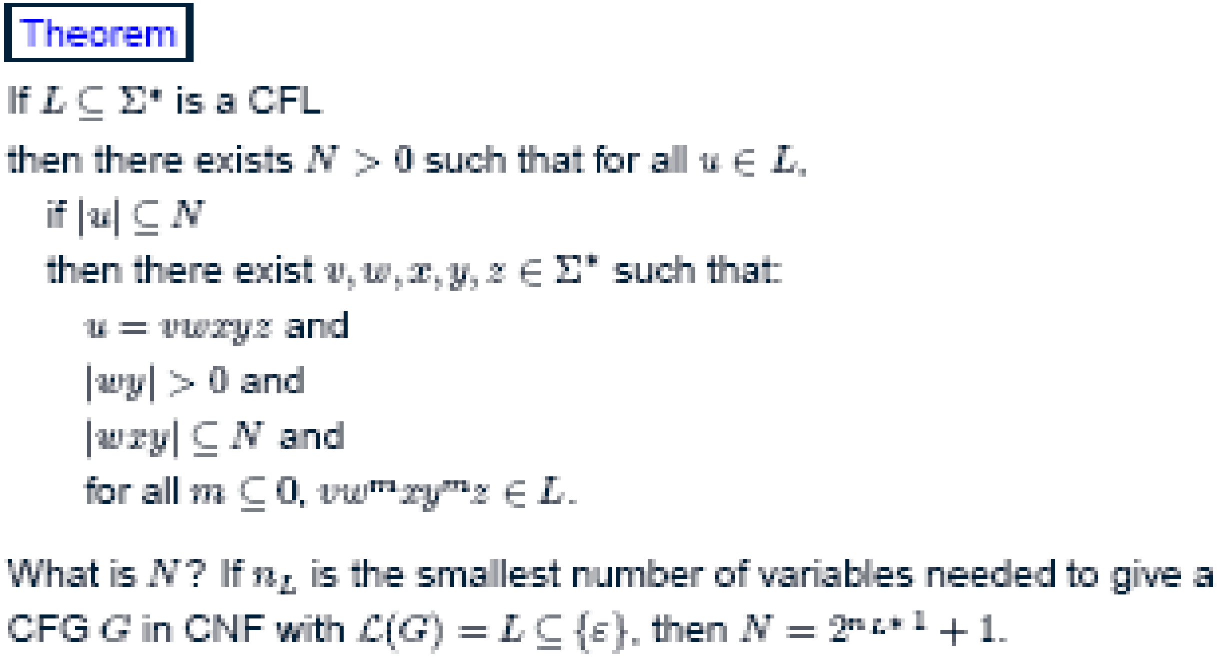
If derivation tree for u is... ... then the following derivation tree also e





If the CFG is in CNF, one can characterize when words are "long enough" to have such trees!

Pumping Lemma for CFL



Theorem

If $L \subseteq \Sigma^*$ is a CFL

then there exists $N > 0$ such that for all $u \in L$,

if $|u| \subseteq N$

then there exist $v, w, x, y, z \in \Sigma^*$ such that:

$u = vwxyz$ and

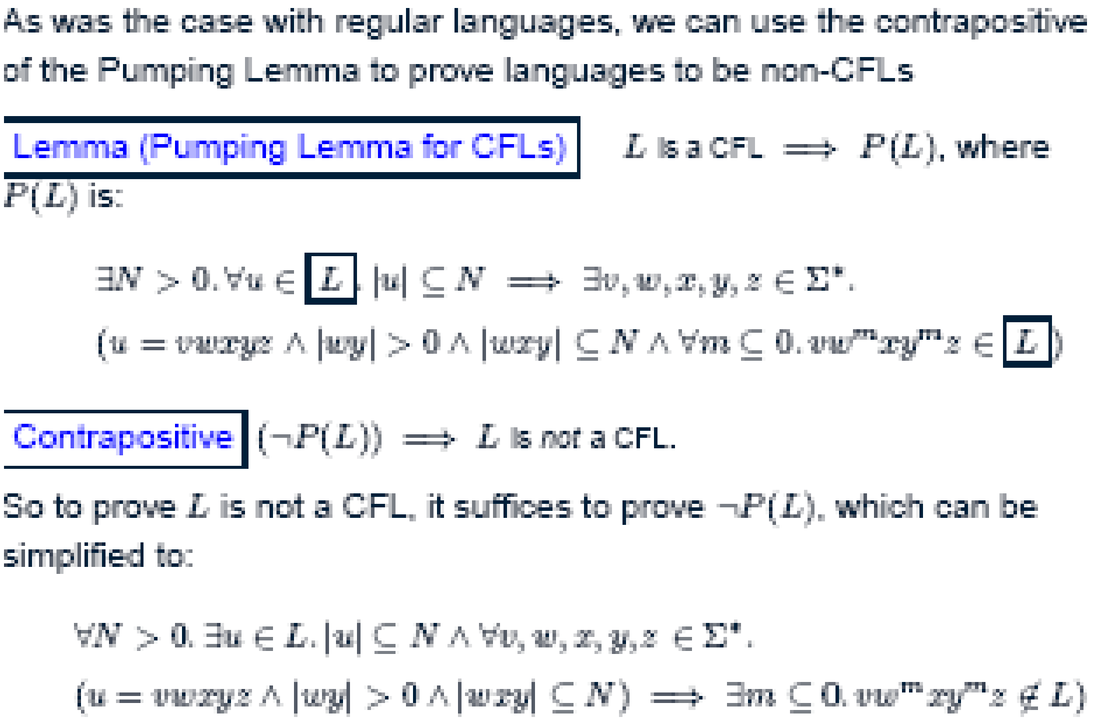
$|wy| > 0$ and

$|wxy| \subseteq N$ and

for all $m \subseteq 0$, $vw^m xy^m z \in L$.

What is N? If n_L is the smallest number of variables needed to give a CFG G in CNF with $\mathcal{L}(G) = L \subseteq \{\varepsilon\}$, then $N = 2^{n_L+1} + 1$.

Proving Languages Non Context Free using Pumping Lemma



As was the case with regular languages, we can use the contrapositive of the Pumping Lemma to prove languages to be non-CFLs

Lemma (Pumping Lemma for CFLs) L is a CFL $\implies P(L)$, where $P(L)$ is:

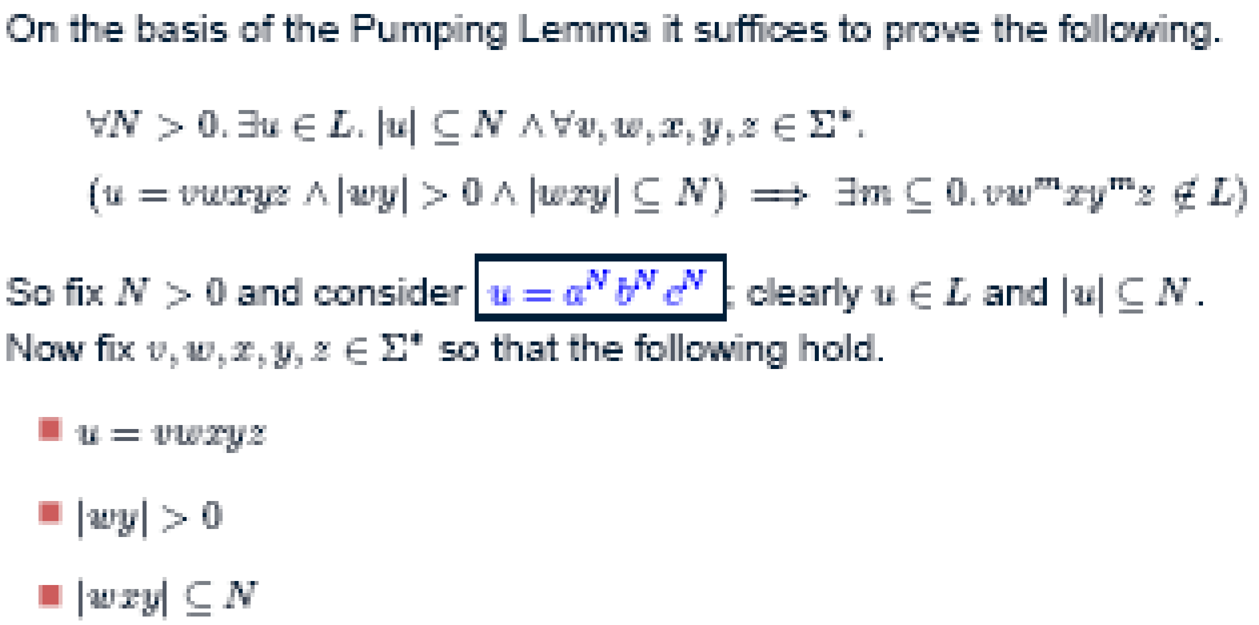
$$\exists N > 0. \forall u \in \boxed{L}. |u| \subseteq N \implies \exists v, w, x, y, z \in \Sigma^*.$$
$$(u = vwxyz \wedge |wy| > 0 \wedge |wxy| \subseteq N \wedge \forall m \subseteq 0. vw^m xy^m z \in \boxed{L})$$

Contrapositive $(\neg P(L)) \implies L$ is *not* a CFL.

So to prove L is not a CFL, it suffices to prove $\neg P(L)$, which can be simplified to:

$$\forall N > 0. \exists u \in L. |u| \subseteq N \wedge \forall v, w, x, y, z \in \Sigma^*.$$
$$(u = vwxyz \wedge |wy| > 0 \wedge |wxy| \subseteq N) \implies \exists m \subseteq 0. vw^m xy^m z \notin L)$$

Prove that L= $\{a^N b^N c^N \mid N \geq 0\}$ is Not a CFL



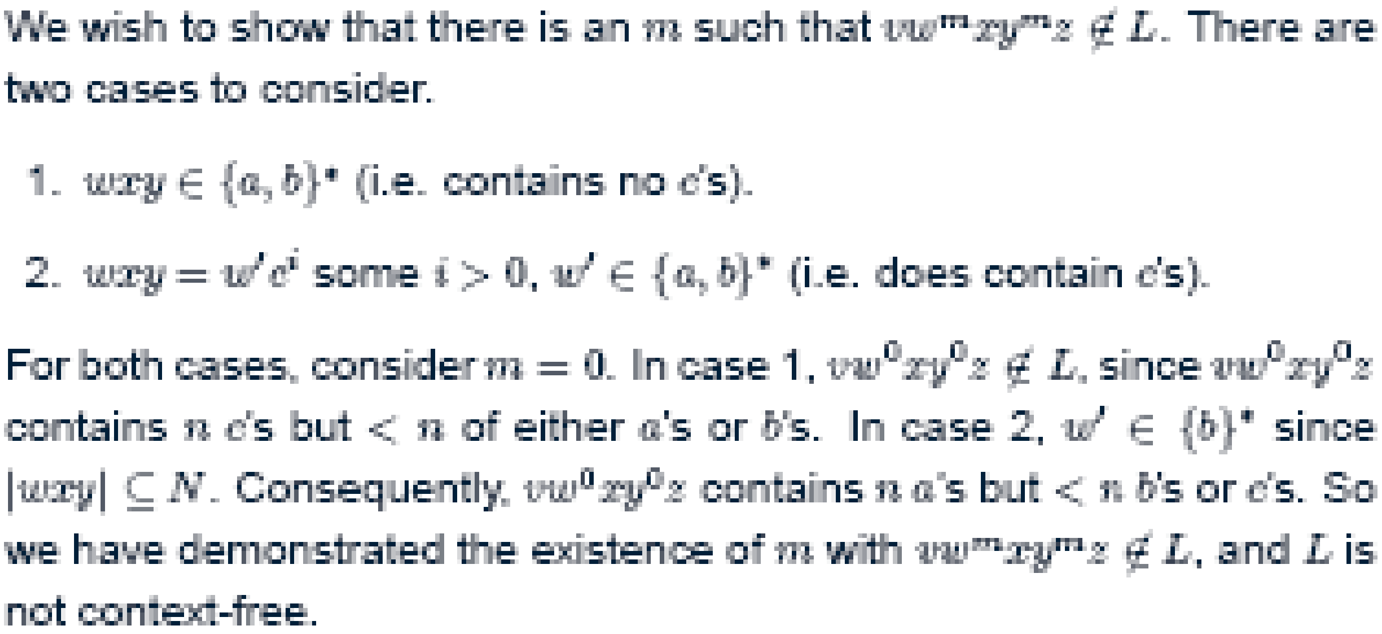
On the basis of the Pumping Lemma it suffices to prove the following.

$$\forall N > 0.\, \exists u \in L.\, |u| \subseteq N \wedge \forall v, w, x, y, z \in \Sigma^*.$$
$$(u = vwxyz \wedge |wy| > 0 \wedge |wxy| \subseteq N) \implies \exists m \subseteq 0.\, vw^m xy^m z \notin L)$$

So fix $N > 0$ and consider $\boxed{u = a^N b^N c^N}$; clearly $u \in L$ and $|u| \subseteq N$.
Now fix $v, w, x, y, z \in \Sigma^*$ so that the following hold.

- $u = vwxyz$
- $|wy| > 0$
- $|wxy| \subseteq N$

Proof Cont…



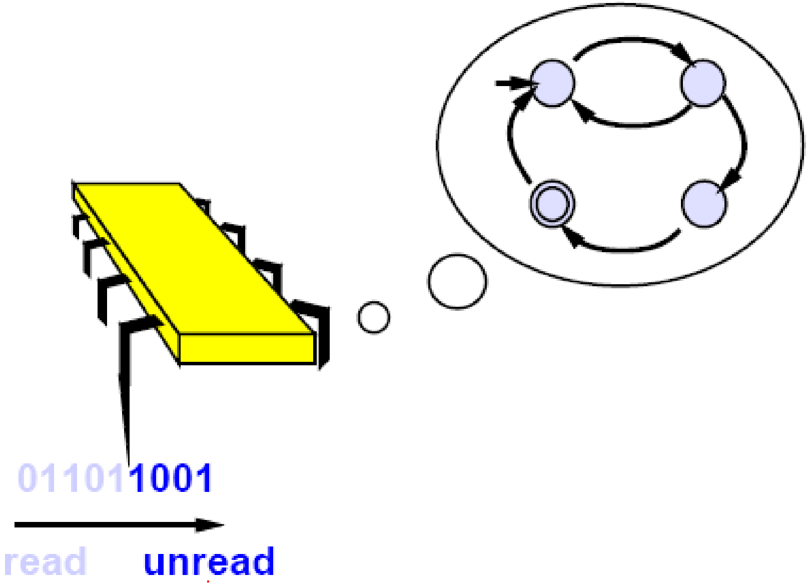
We wish to show that there is an m such that $vw^m xy^m z \notin L$. There are two cases to consider.

1. $wxy \in \{a, b\}^*$ (i.e. contains no c's).
2. $wxy = w'c^i$ some $i > 0$, $w' \in \{a, b\}^*$ (i.e. does contain c's).

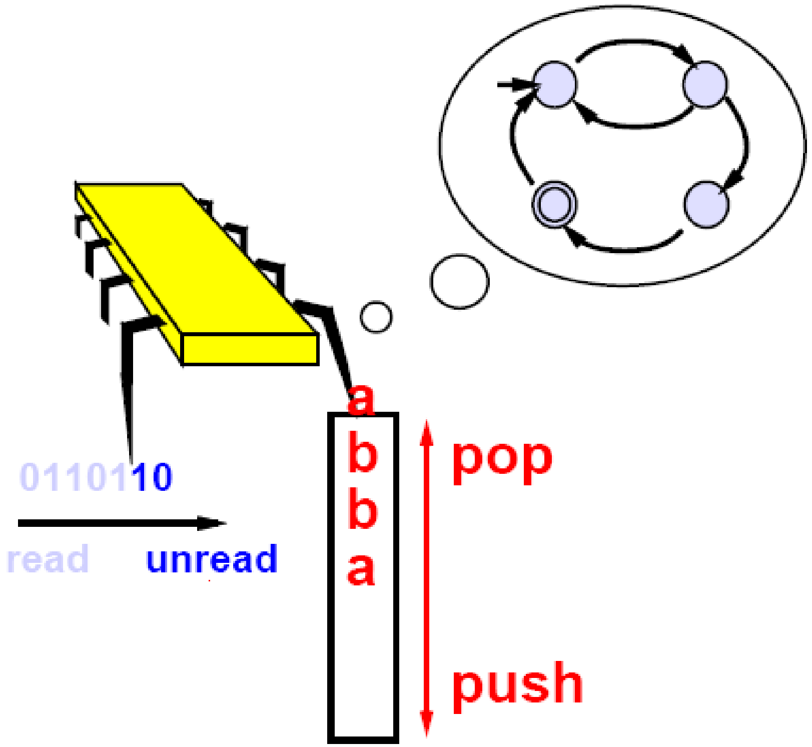
For both cases, consider $m = 0$. In case 1, $vw^0 xy^0 z \notin L$, since $vw^0 xy^0 z$ contains n c's but $<$ n of either a's or b's. In case 2, $w' \in \{b\}^*$ since $|wxy| \subseteq N$. Consequently, $vw^0 xy^0 z$ contains n a's but $< n$ b's or c's. So we have demonstrated the existence of m with $vw^m xy^m z \notin L$, and L is not context-free.

Turing Machines

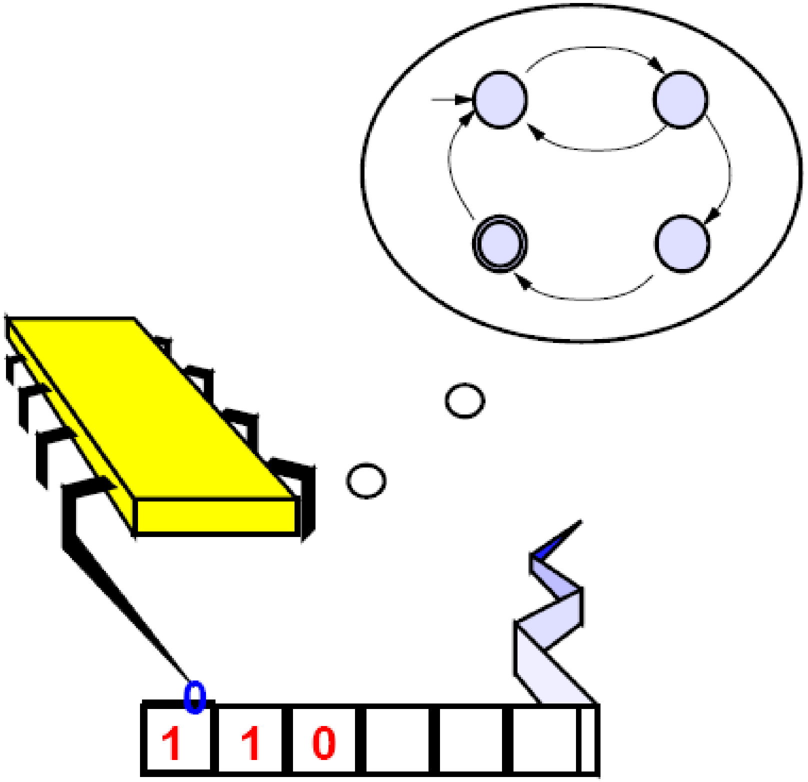
A Finite Automaton



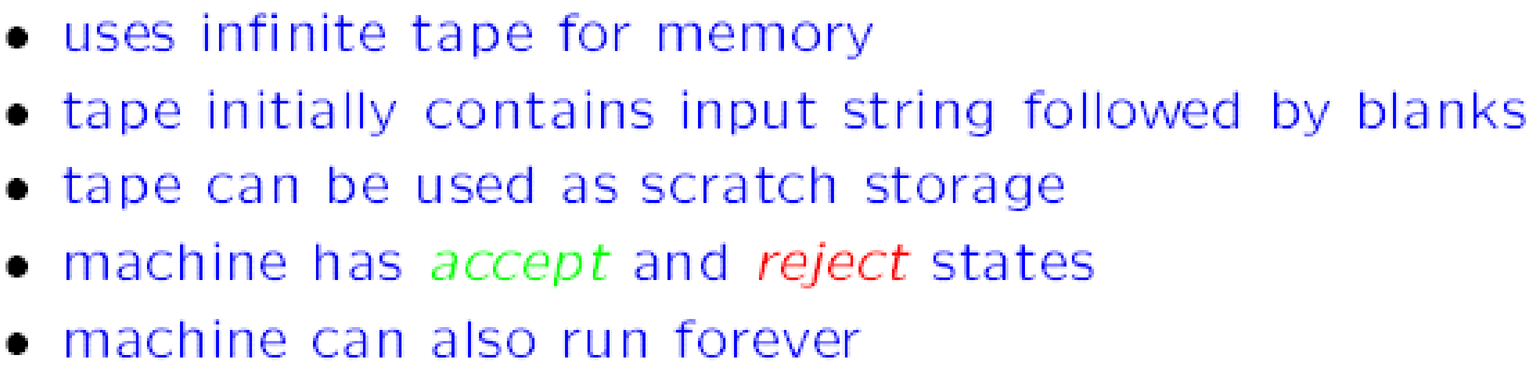
A Pushdown Automaton



A Turing Machine

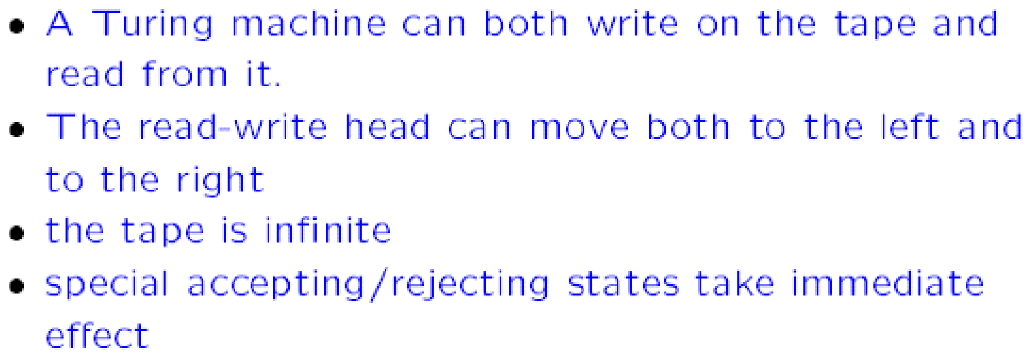


Cont..



- uses infinite tape for memory
- tape initially contains input string followed by blanks
- tape can be used as scratch storage
- machine has *accept* and *reject* states
- machine can also run forever

Differences



- A Turing machine can both write on the tape and read from it.
- The read-write head can move both to the left and to the right
- the tape is infinite
- special accepting/rejecting states take immediate effect

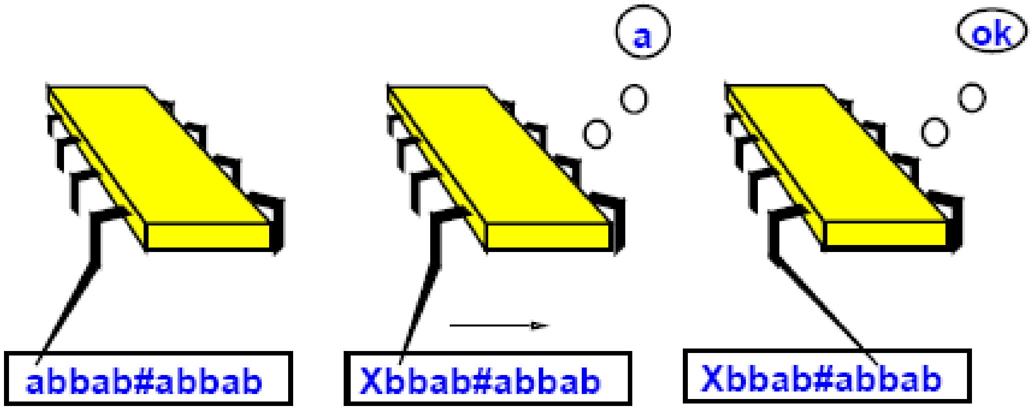
Example

A machine that tests for membership in the language

$$A = \{w\#w | w \in \{0,1\}^*\}$$

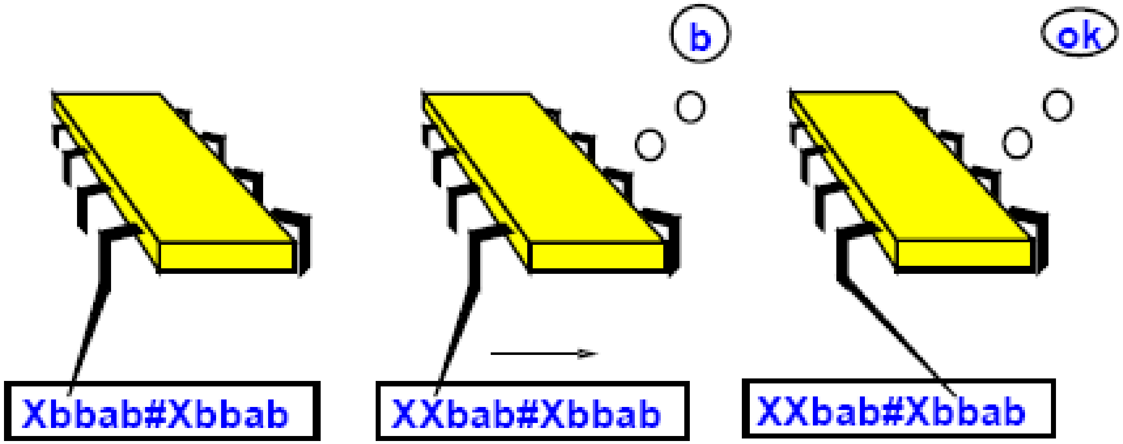
Zig-zag across tape, crossing of matching symbols.

Cont…



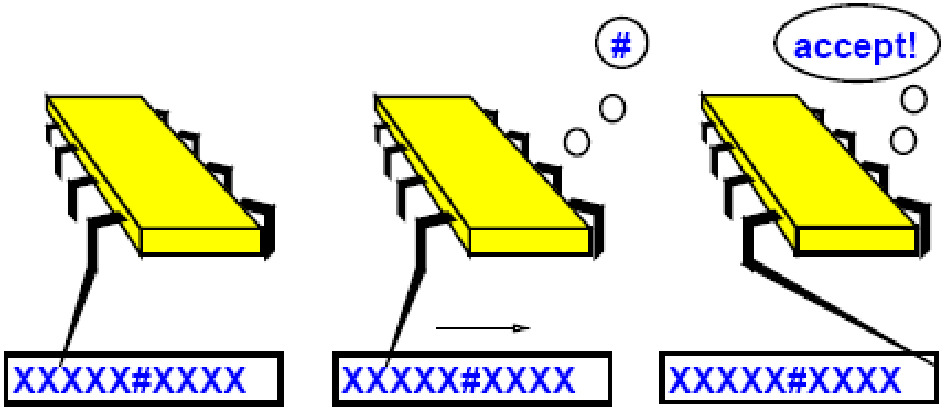
- tape head starts over leftmost symbol
- record symbol in control and overwrite X
- scan right: *reject*if blank encountered before #
- when # encountered, move right one space
- if symbols don't match, *reject*.

Cont…



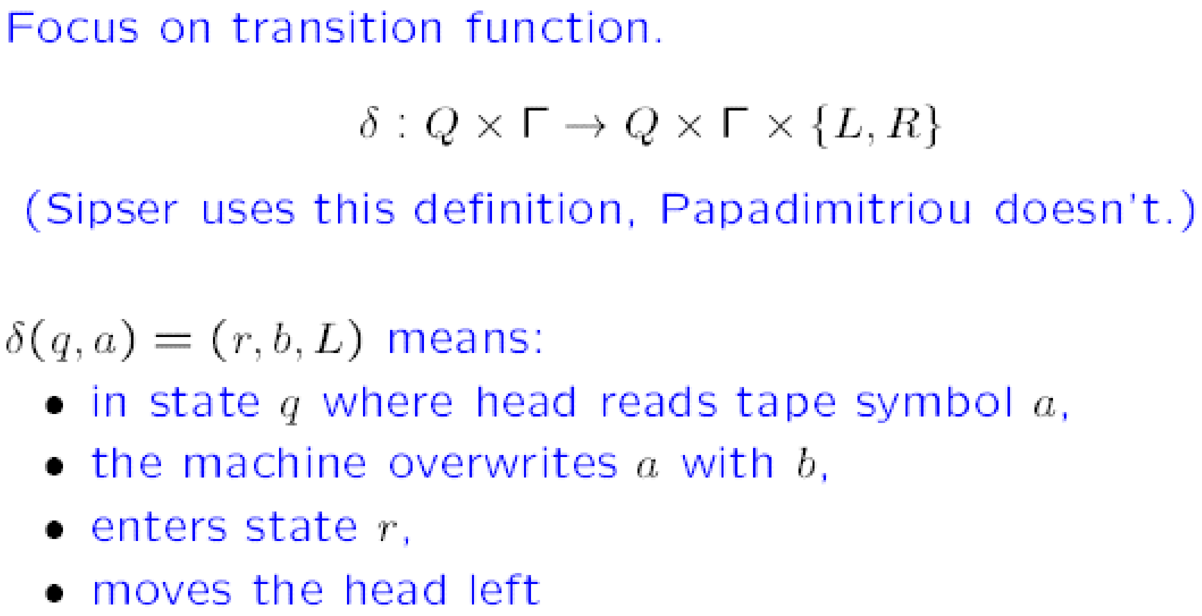
- overwrite X
- scan left, past # to X
- move one space right
- record symbol and overwrite X
- scan right past # to X . . .

Cont…



- finally, scan left
- if 0 or 1 encountered, *reject*
- when blank encountered, *accept*

Formal Definition



Focus on transition function.

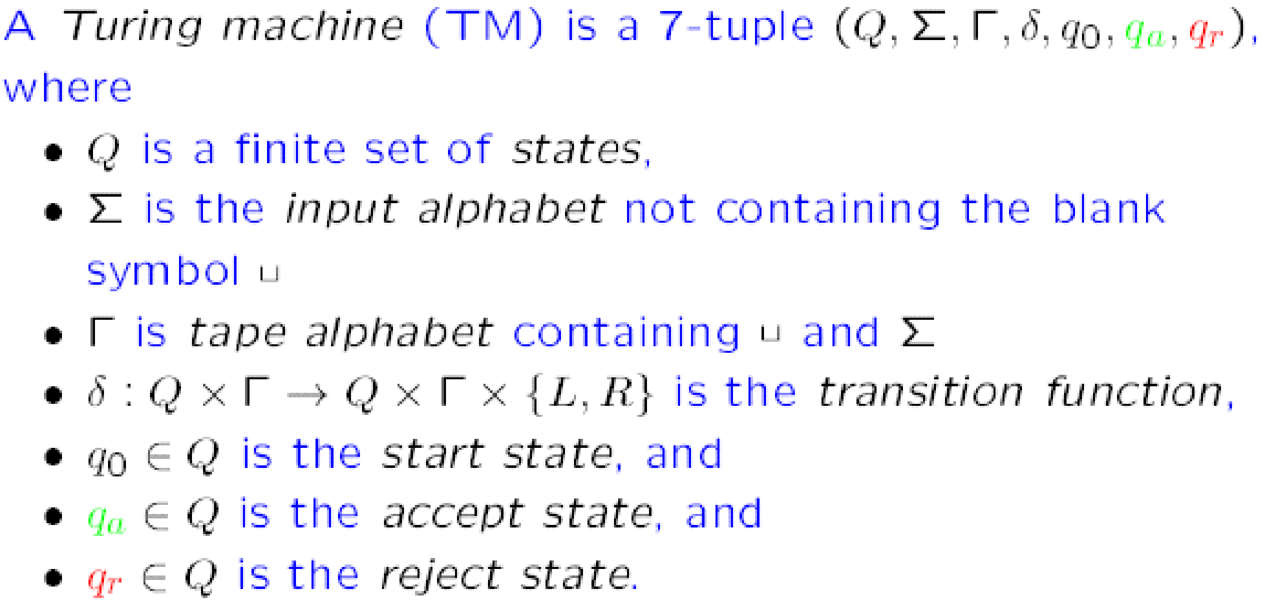
$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

(Sipser uses this definition, Papadimitriou doesn't.)

$\delta(q, a) = (r, b, L)$ means:

- in state q where head reads tape symbol a,
- the machine overwrites a with b,
- enters state r,
- moves the head left

Cont..



A *Turing machine* (TM) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$, where

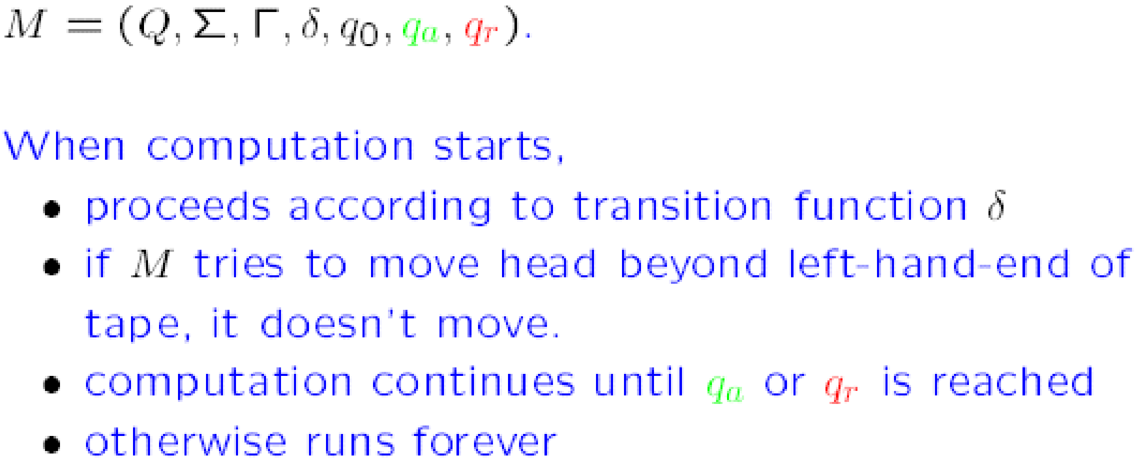
- Q is a finite set of *states*,
- Σ is the *input alphabet* not containing the blank symbol $\sqcup$
- Γ is *tape alphabet* containing $\sqcup$ and Σ
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $q_a \in Q$ is the *accept state*, and
- $q_r \in Q$ is the *reject state*.

Cont…

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ computes as follows

- input $w = w_1 w_2 \ldots w_n \in \Sigma^*$
- is on leftmost n tape squares
- rest of tape is blank $\sqcup$
- head is on leftmost square of tape
- because $\sqcup \notin \Sigma$, blank is end of tape

Cont…

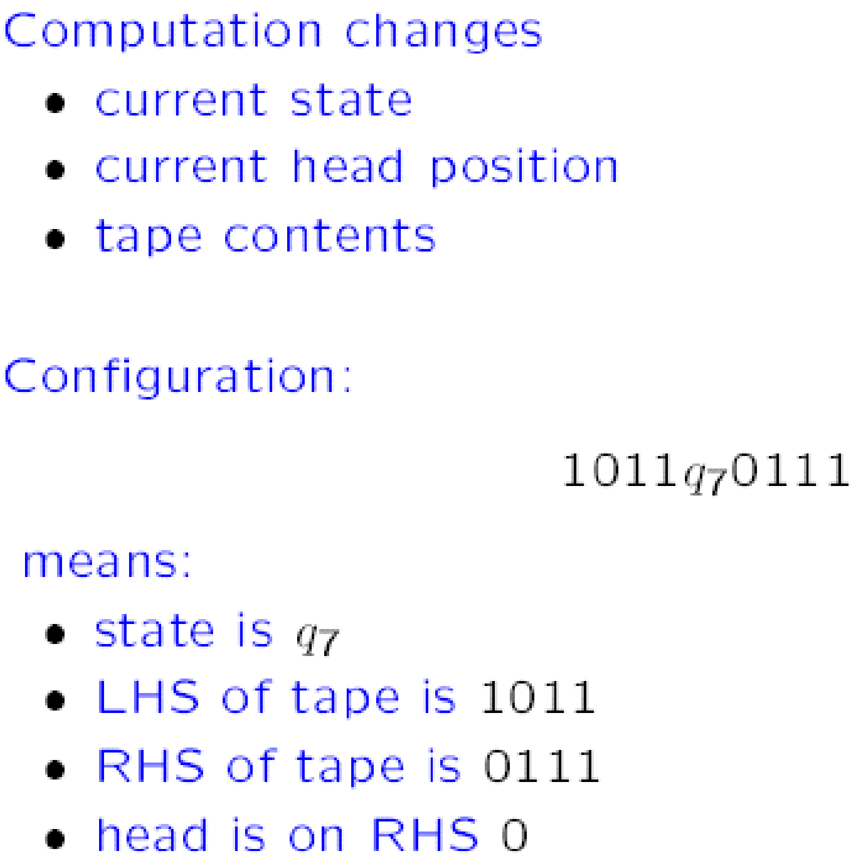


$M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$.

When computation starts,

- proceeds according to transition function δ
- if M tries to move head beyond left-hand-end of tape, it doesn't move.
- computation continues until q_a or q_r is reached
- otherwise runs forever

Configurations



Computation changes

- current state
- current head position
- tape contents

Configuration:

$$1011q_70111$$

means:

- state is q_7
- LHS of tape is 1011
- RHS of tape is 0111
- head is on RHS 0

Cont…

uaq_ibv yields uq_jacv if

$$\delta(q_i, b) = (q_j, c, L)$$

uaq_ibv yields $uacq_jv$ if

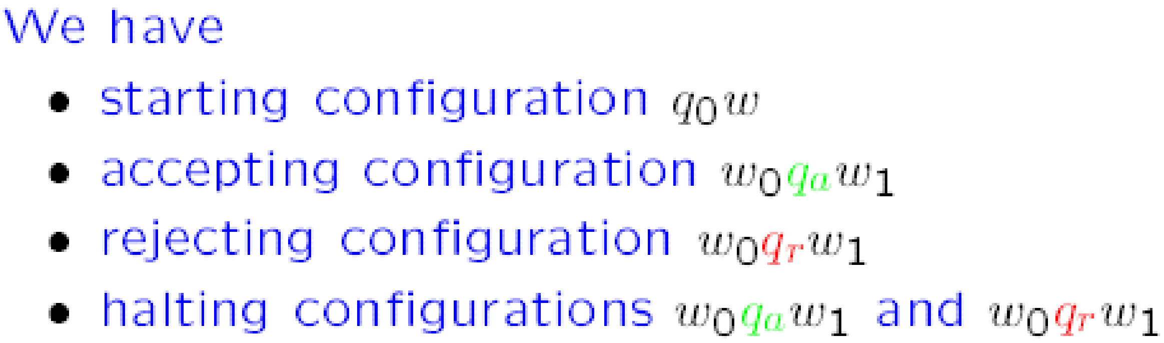
$$\delta(q_i, b) = (q_j, c, R)$$

Special case: q_ibv yields q_jcv if

$$\delta(q_i, b) = (q_j, c, L)$$

wq_i is the same as $wq_i\sqcup$.

More Configuration



We have

- starting configuration $q_0 w$
- accepting configuration $w_0 q_a w_1$
- rejecting configuration $w_0 q_r w_1$
- halting configurations $w_0 q_a w_1$ and $w_0 q_r w_1$

Accepting a Language

Turing machine M *accepts* input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exist

- C_1 is start configuration of M on w
- each C_i yields C_{i+1}
- C_k is an accepting configuration.

The collection of strings that M accepts is the *language* of M, denoted $L(M)$.

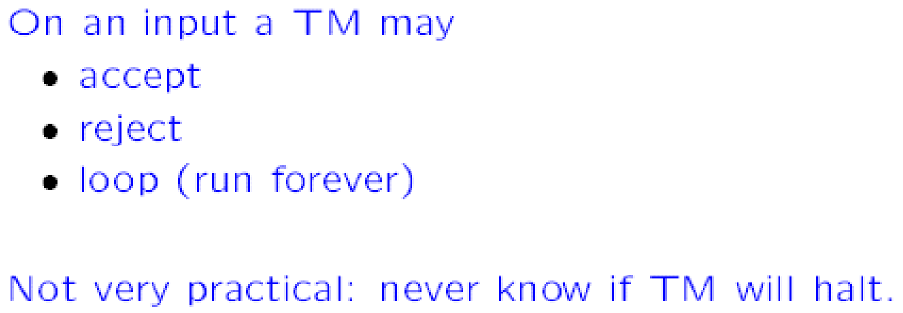
Enumerable Languages

Definition: A language is *(recursively) enumerable* if some Turing machine accepts it.

UNIT – V

UNDECIDABILITY

Enumerable Languages

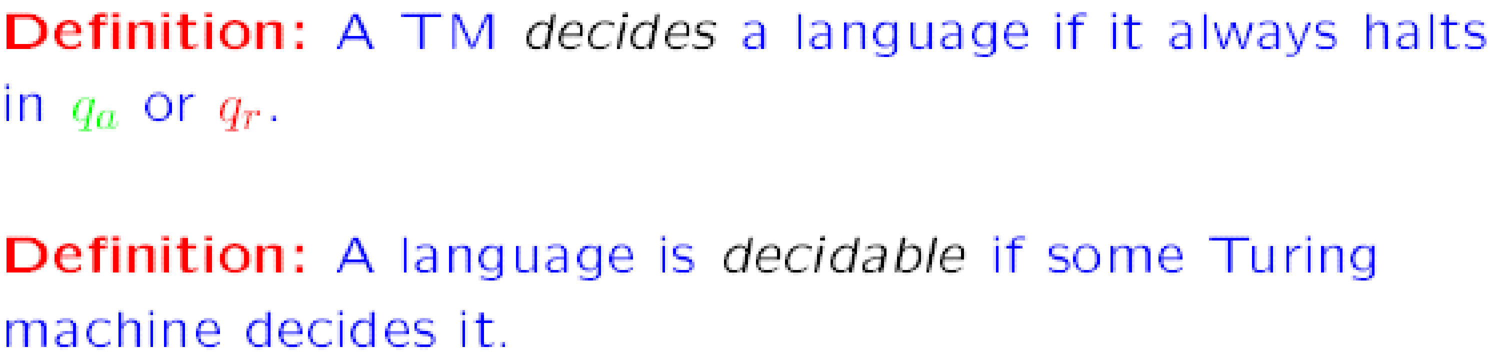


On an input a TM may

- accept
- reject
- loop (run forever)

Not very practical: never know if TM will halt.

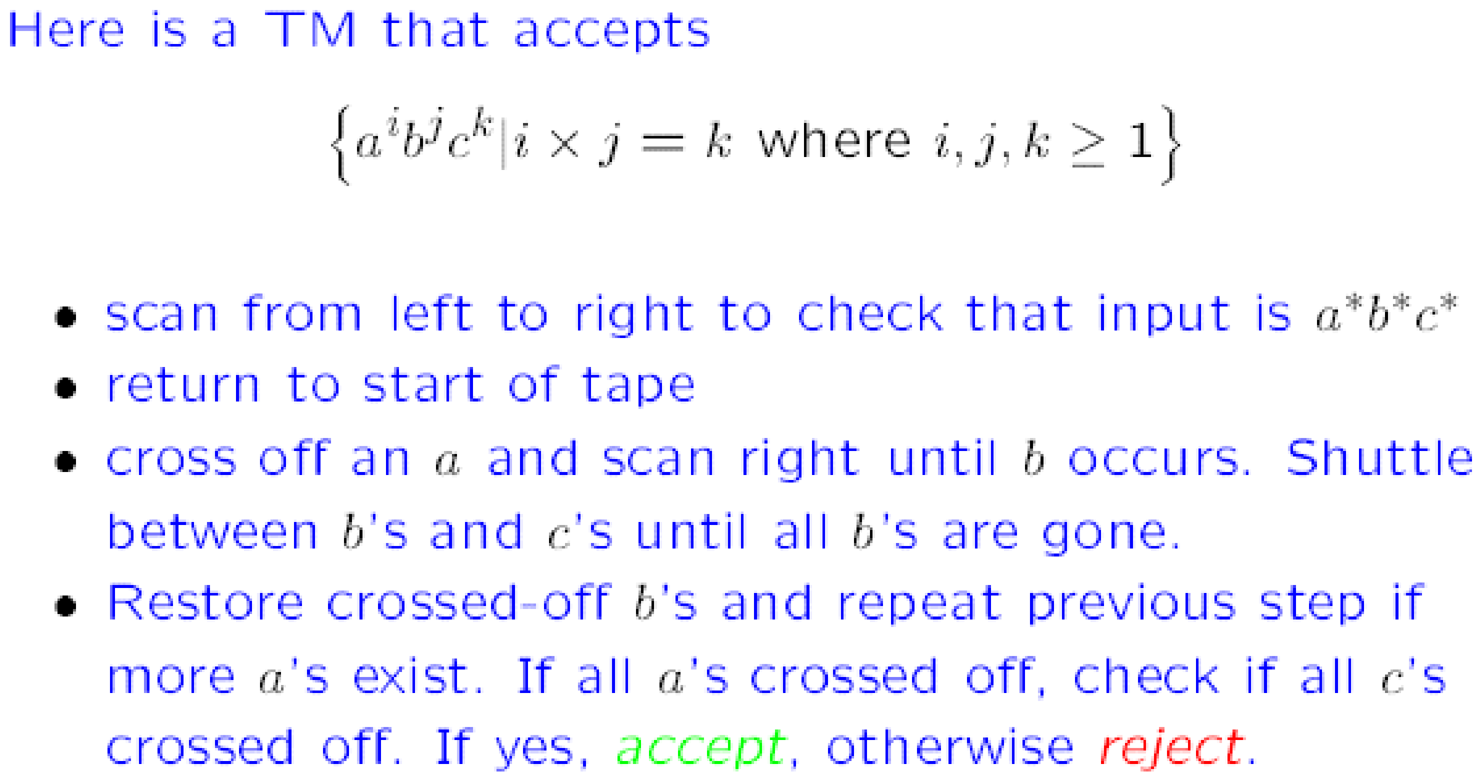
Enumerable Languages



Definition: A TM *decides* a language if it always halts in q_a or q_r.

Definition: A language is *decidable* if some Turing machine decides it.

Example

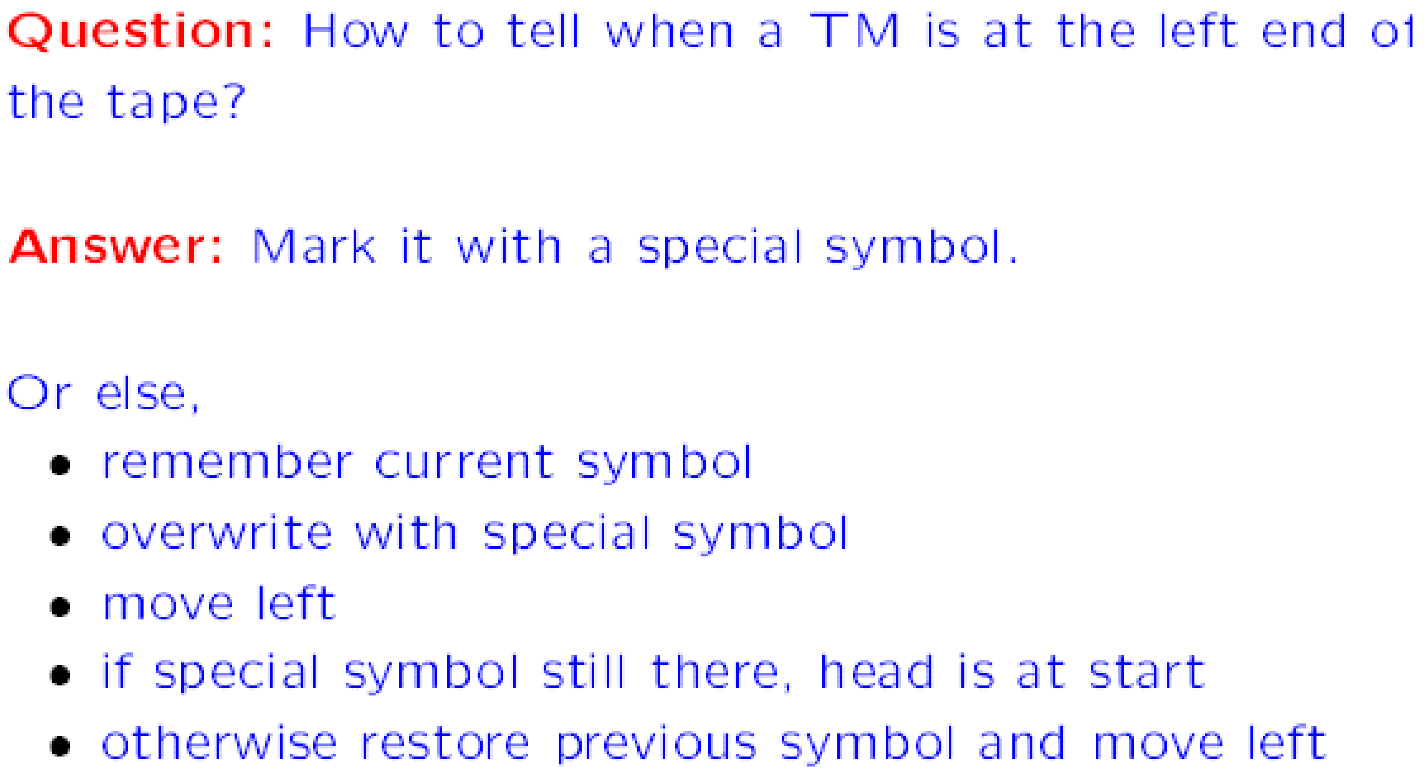


Here is a TM that accepts

$$\left\{a^i b^j c^k | i \times j = k \text{ where } i, j, k \geq 1\right\}$$

- scan from left to right to check that input is $a^* b^* c^*$
- return to start of tape
- cross off an a and scan right until b occurs. Shuttle between b's and c's until all b's are gone.
- Restore crossed-off b's and repeat previous step if more a's exist. If all a's crossed off, check if all c's crossed off. If yes, *accept*, otherwise *reject*.

Example



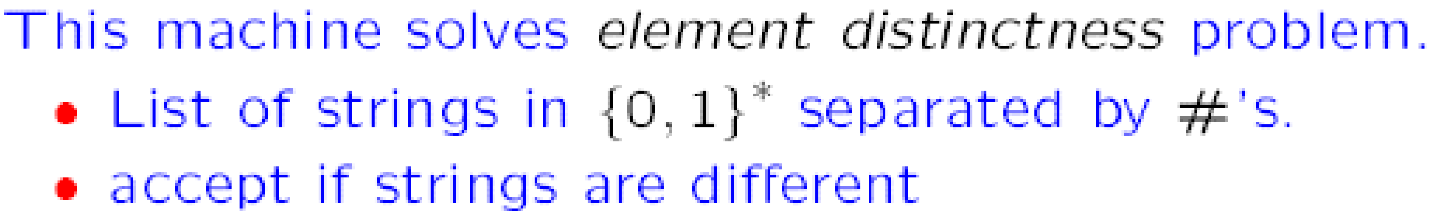
Question: How to tell when a TM is at the left end of the tape?

Answer: Mark it with a special symbol.

Or else,

- remember current symbol
- overwrite with special symbol
- move left
- if special symbol still there, head is at start
- otherwise restore previous symbol and move left

Example



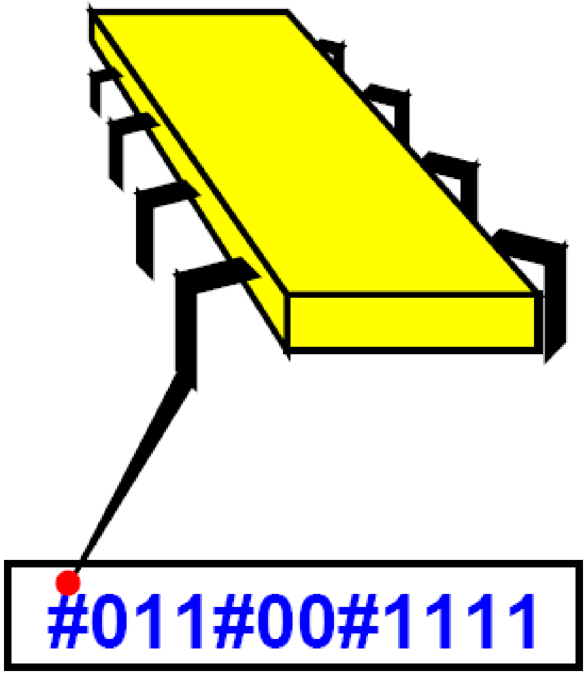
This machine solves *element distinctness* problem.

- List of strings in $\{0,1\}^*$ separated by $\#$'s.
- accept if strings are different

Element Distinctness

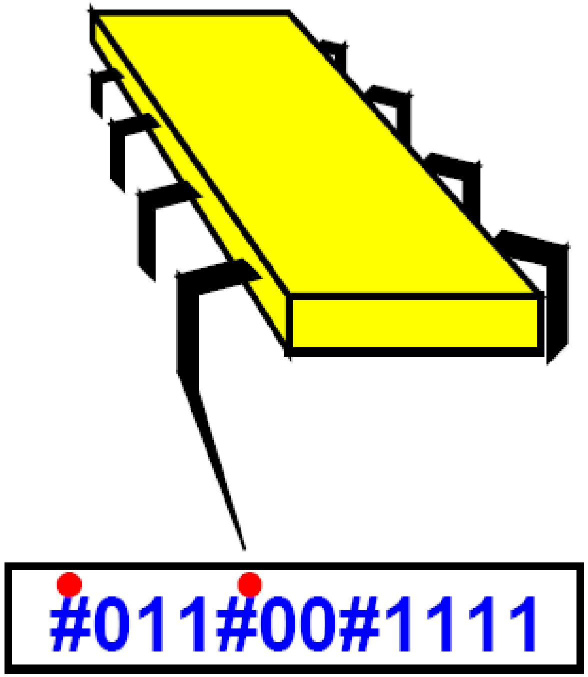
On input w

- place a mark on leftmost tape symbol. If symbol not #, reject.



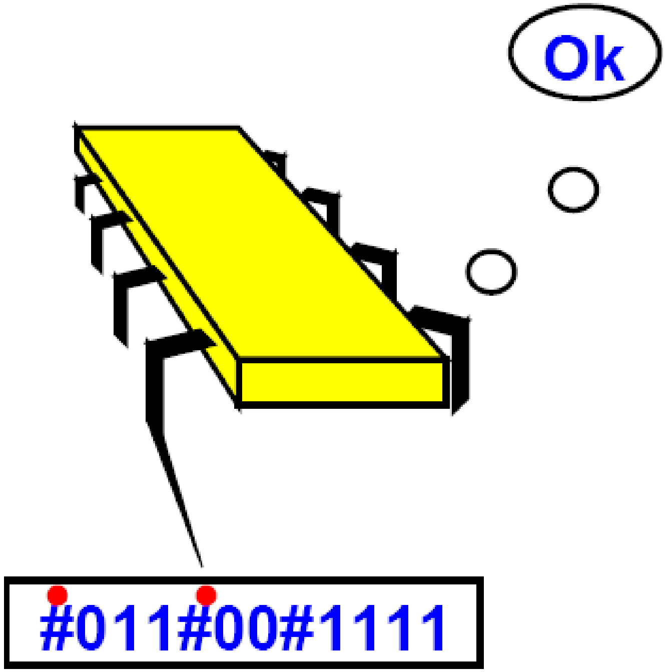
Element Distinctness

- scan right to next # and place mark on top. If none encountered, reject



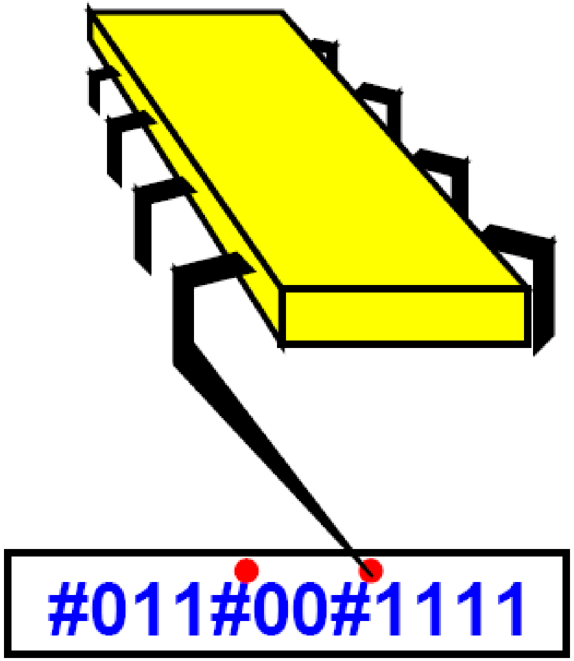
Element Distinctness

- By zig-zagging, compare the two strings to the left of the marked #'s. If equal, reject.



Element Distinctness

- Move rightmost mark to next #, if any.
- otherwise move leftmost mark to next # on right and rightmost mark to # after that.
- if not possible, accept.



Element Distinctness

Question: How do we "mark" a symbol?

Answer: For each tape symbol $\#$, add tape symbol $\dot{\#}$.

Variants

Alternative Turing machine definitions abound.

Suppose the Turing machine head is allowed to stay put.

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

Question: Does this add any power?

Answer: No. Replace each S transition with two transitions: R then L. (Why not vice-versa?)

Notion of simulation important.

Multitape Turing Machine

- each tape has its own head
- initially, input string on tape 1 and rest blank

Transition function:

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$$

Expression

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$$

- machine starts in q_i
- if heads 1 through k reading $a_1, \ldots, a_k$,
- then machine goes to q_j,
- heads 1 thought k write $b_1, \ldots, b_k$,
- and moves each head right or left as specified.

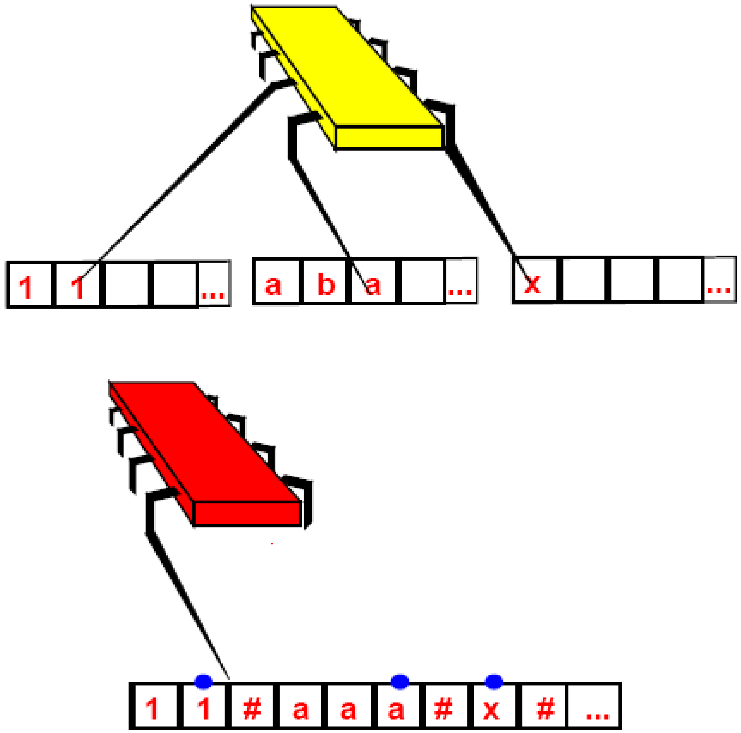
Equivalence

Theorem: A language is enumerable if and only if some multitape Turing machine accepts it.

One direction is trivial.

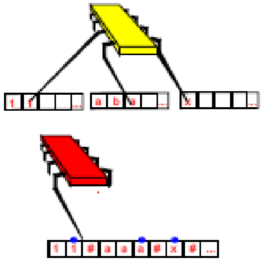
Show how to convert a multitape TM M into a single-tape TM S.

Simulation



- M has k tapes.
- S simulates k tapes by storing them all on a single tape with delimiter $\#$.
- S "marks" current head position

Simulation



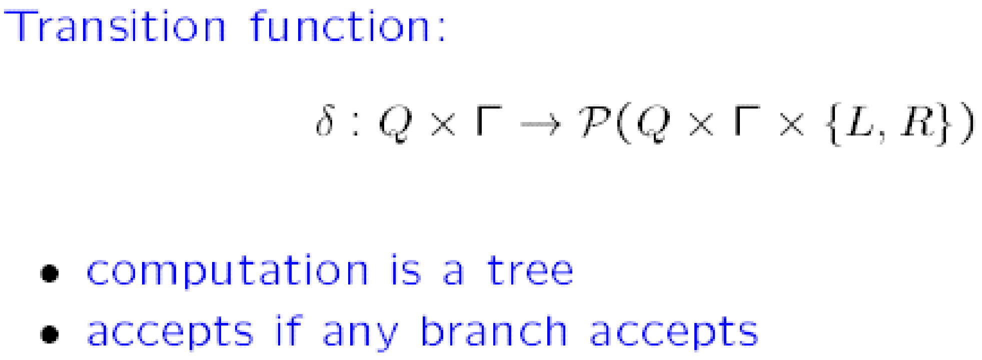
On input $w = w_1 \cdots w_n$, S:

- puts on its tape:

$$\# \overset{\bullet}{w}_1 \; w_2 \cdot w_n \# \overset{\bullet}{\sqcup} \# \overset{\bullet}{\sqcup} \# \cdots \#$$

- scans its tape from first # to $k + 1$-st # to read symbols under "virtual" heads.
- rescans to write new symbols and move heads
- if S tries to move virtual head onto #, then M is trying to move head onto unused blank square. S writes blank symbol on tape, and shifts the rest of the tape right one square.

Non Deterministic Turing Machine

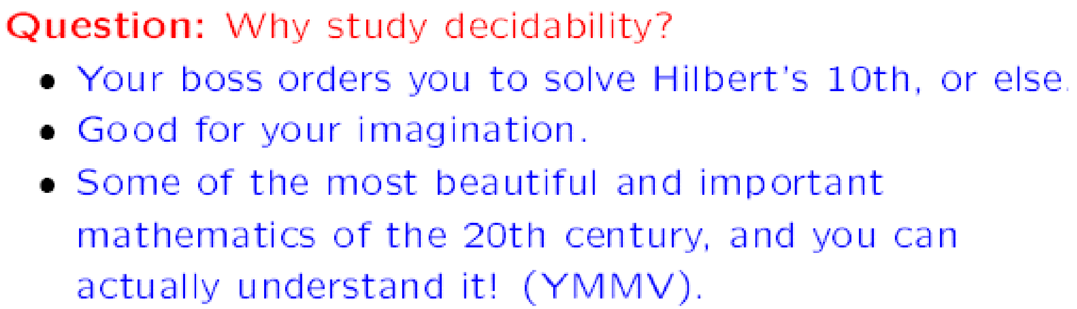


Transition function:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

- computation is a tree
- accepts if any branch accepts

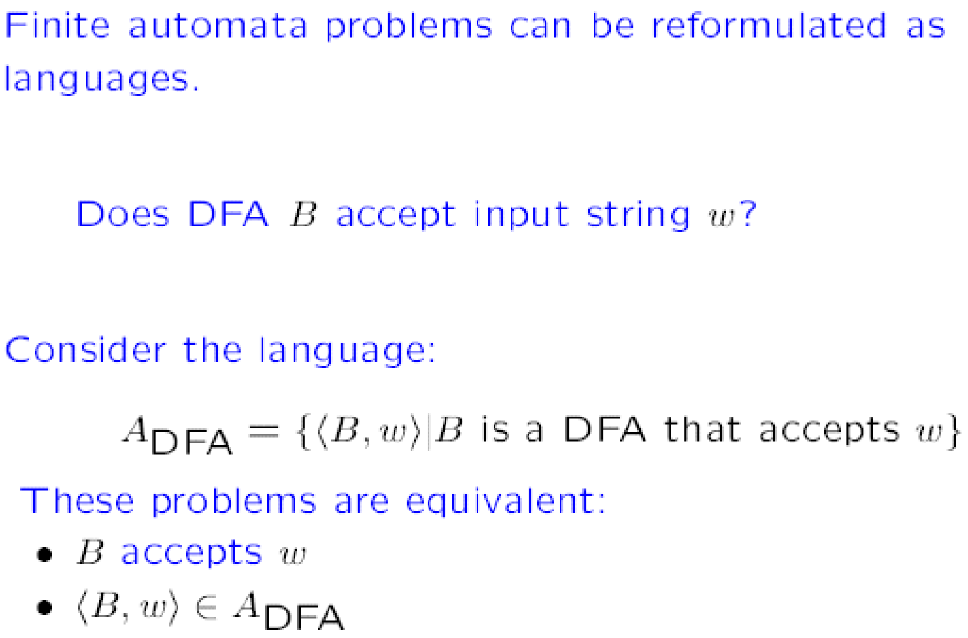
Decidability



Question: Why study decidability?

- Your boss orders you to solve Hilbert's 10th, or else.
- Good for your imagination.
- Some of the most beautiful and important mathematics of the 20th century, and you can actually understand it! (YMMV).

Example



Finite automata problems can be reformulated as languages.

Does DFA B accept input string w?

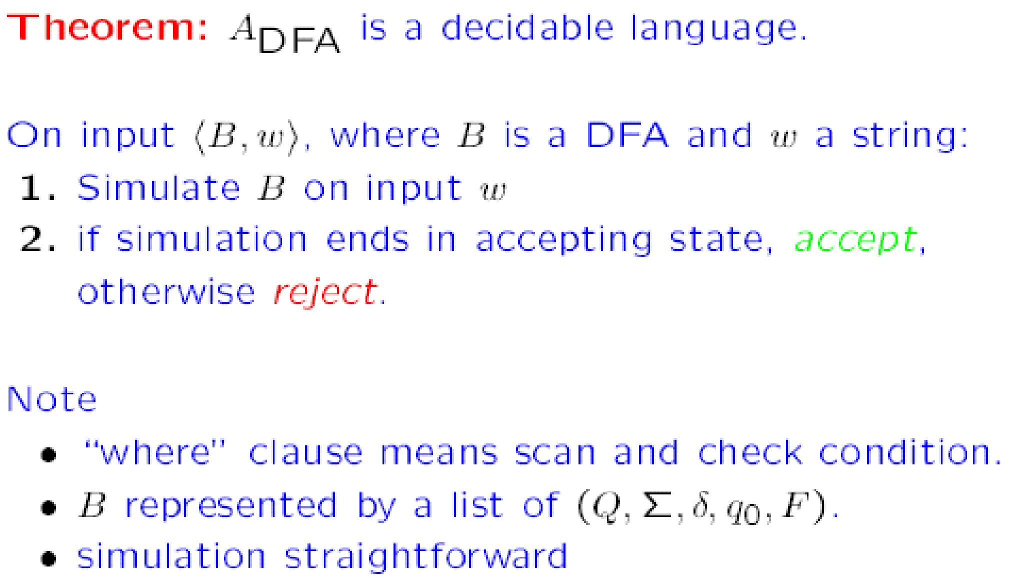
Consider the language:

$$A_{\text{DFA}} = \{\langle B, w\rangle | B \text{ is a DFA that accepts } w\}$$

These problems are equivalent:

- B accepts w
- $\langle B, w\rangle \in A_{\text{DFA}}$

Theorem



Theorem: A_{DFA} is a decidable language.

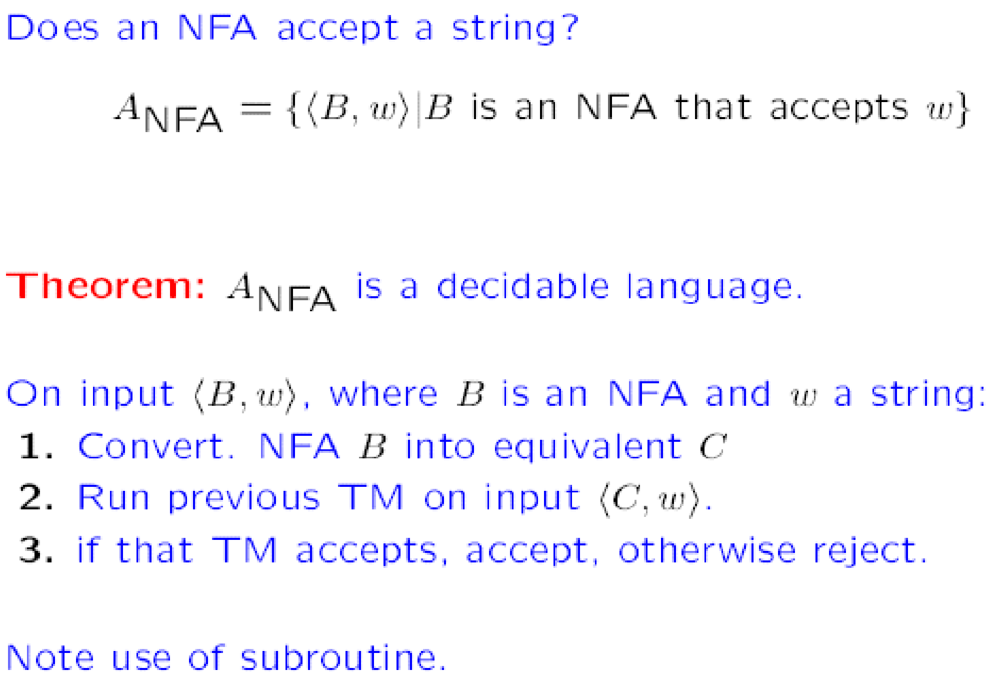
On input $\langle B, w \rangle$, where B is a DFA and w a string:

1. Simulate B on input w
2. if simulation ends in accepting state, *accept*, otherwise *reject*.

Note

- "where" clause means scan and check condition.
- B represented by a list of $(Q, \Sigma, \delta, q_0, F)$.
- simulation straightforward

Theorem



Does an NFA accept a string?

$$A_{\mathsf{NFA}} = \{\langle B, w\rangle | B \text{ is an NFA that accepts } w\}$$

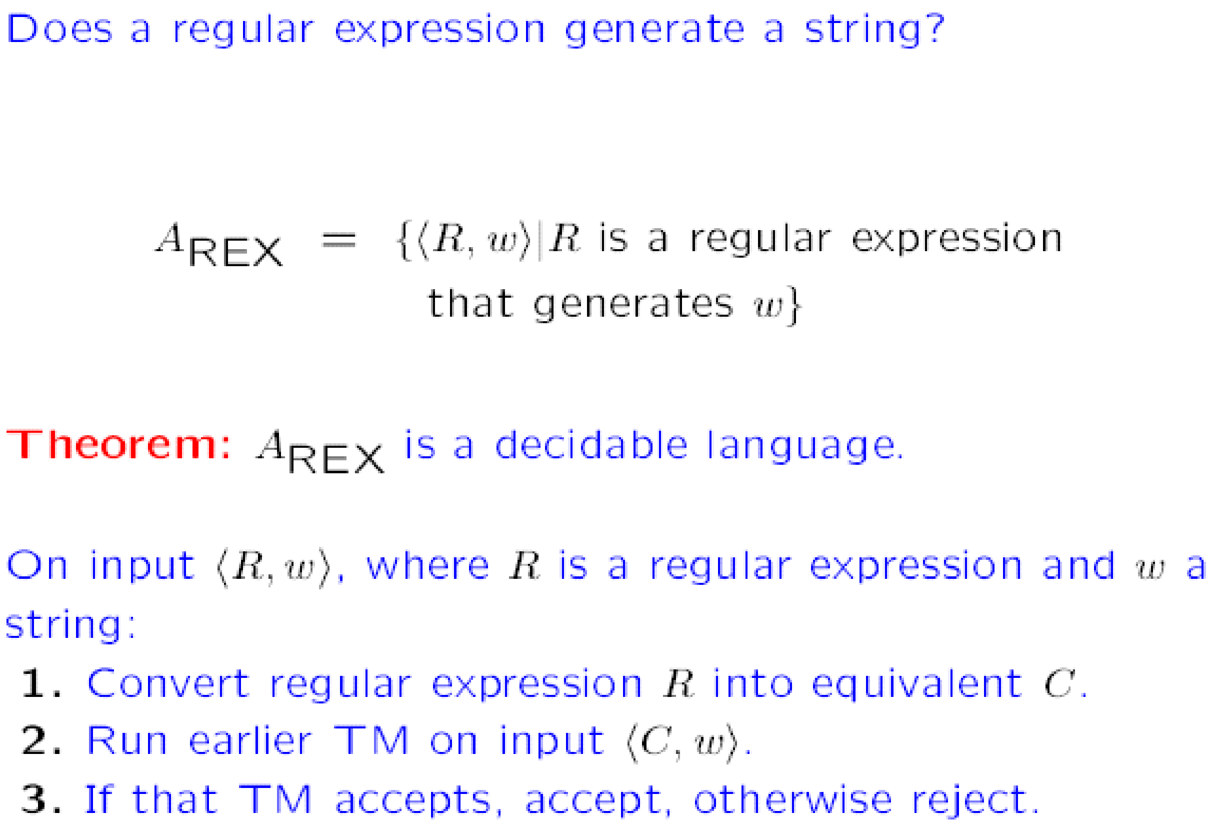
Theorem: A_{NFA} is a decidable language.

On input $\langle B, w\rangle$, where B is an NFA and w a string:

1. Convert. NFA B into equivalent C
2. Run previous TM on input $\langle C, w\rangle$.
3. if that TM accepts, accept, otherwise reject.

Note use of subroutine.

Theorem



Does a regular expression generate a string?

$$\begin{aligned} A_{\mathsf{REX}} \;=\; & \{\langle R, w\rangle | R \text{ is a regular expression} \\ & \quad \text{that generates } w\} \end{aligned}$$

Theorem: A_{REX} is a decidable language.

On input $\langle R, w\rangle$, where R is a regular expression and w a string:

1. Convert regular expression R into equivalent C.
2. Run earlier TM on input $\langle C, w\rangle$.
3. If that TM accepts, accept, otherwise reject.

Theorem

Does a DFA accept the empty language?

$$E_{\mathsf{DFA}} = \{\langle A\rangle | A \text{ is a DFA and } L(A) = \emptyset\}$$

Theorem: E_{DFA} is a decidable language.

```
On input $\langle A\rangle$, where $A$ is a DFA:
     Mark. the start state of $A$.
 2.  Repeat until no new states are marked:
 3.  Mark any state that has a transition coming into it
     from any already marked state.
 4.  If no accept state is marked, accept, otherwise
     reject.
```

This TM actually just tests whether any accepting state is reachable from initial state.

Halting Problem

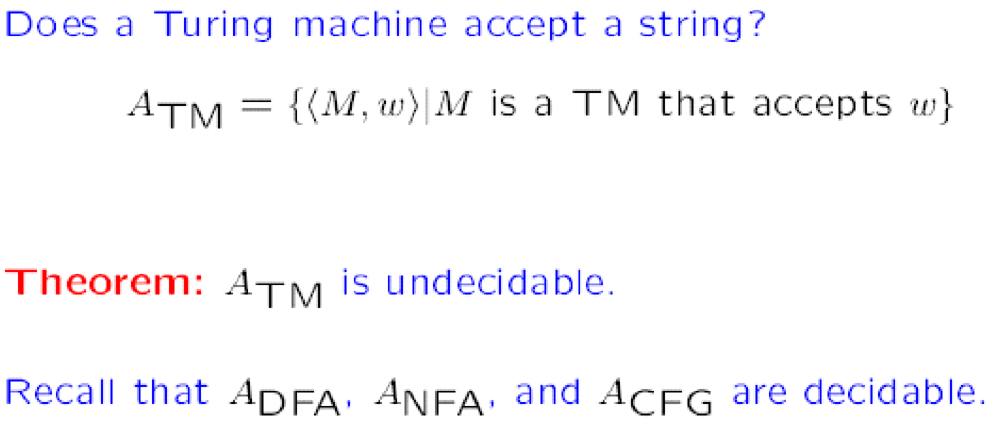
One of the most philosophically important theorems of the theory of computation.

Computers (and computation) are limited in a very fundamental way.

Common, every-day problems are unsolvable.

- does a program sort an array of integers?
- both program and specification are precise mathematical objects.
- proving program $\cong$ specification should be just like proving that triangle 1 $\cong$ triangle 2 . . .

Cont..



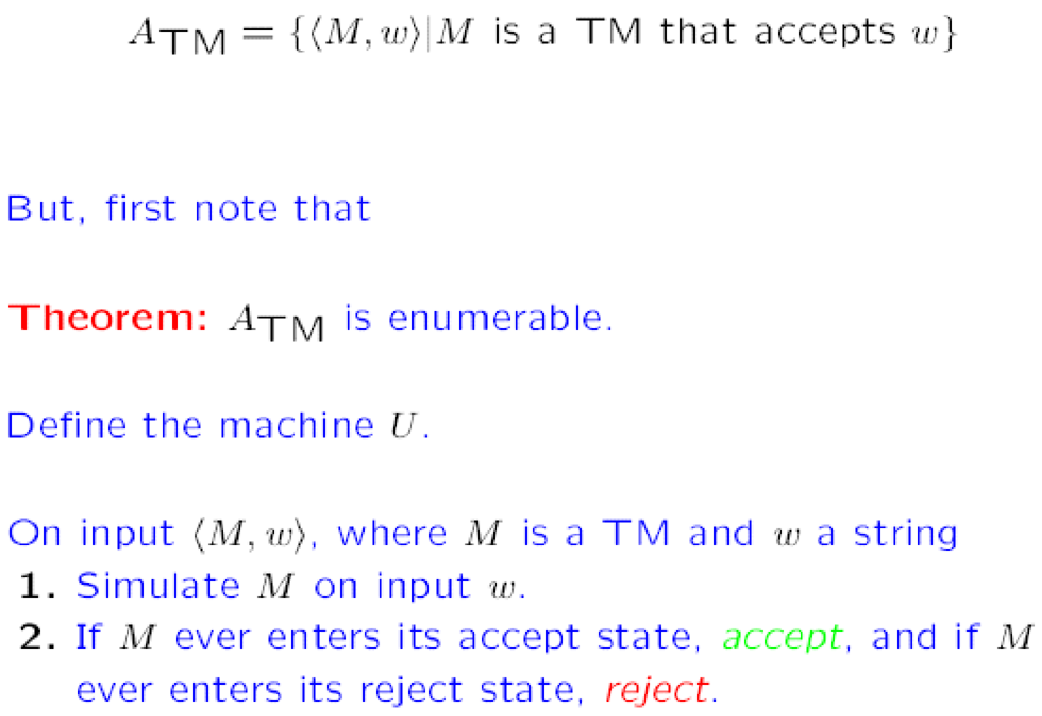
Does a Turing machine accept a string?

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM that accepts } w\}$$

Theorem: A_{TM} is undecidable.

Recall that A_{DFA}, A_{NFA}, and A_{CFG} are decidable.

Cont..



$$A_{\mathsf{TM}} = \{\langle M, w\rangle | M \text{ is a TM that accepts } w\}$$

But, first note that

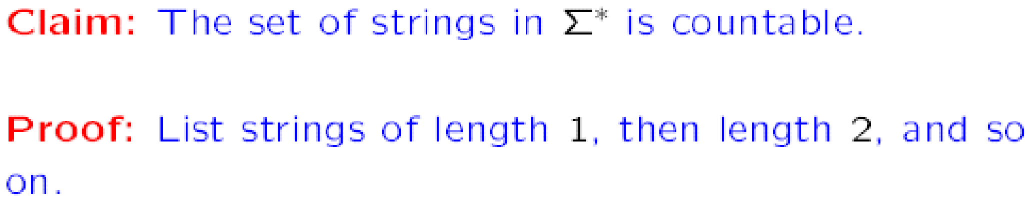
Theorem: A_{TM} is enumerable.

Define the machine U.

On input $\langle M, w\rangle$, where M is a TM and w a string

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*, and if M ever enters its reject state, *reject*.

Turing Machines are countable



Claim: The set of strings in Σ^* is countable.

Proof: List strings of length 1, then length 2, and so on.

Cont..

Claim: The set of Turing machines is countable.

Proof: Each TM M has an encoding as a string $\langle M \rangle$.

We can list all strings, and just omit the ones that are not legal TM encodings.

Post Correspondence Problem

An instance of PCP is a collection of dominos

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \left[\frac{t_3}{b_3}\right], \ldots\ldots, \left[\frac{t_k}{b_k}\right] \right\}$$

A match is a sequence $i_1, i_2, \ldots, i_r$ such that $t_{i_1} t_{i_2} \ldots t_{i_r} = b_{i_1} b_{i_2} \ldots b_{i_r}$.

E.g. $P = \left\{ \left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right] \right\}$

A match $\left[\frac{a}{ab}\right] \left[\frac{b}{ca}\right] \left[\frac{ca}{a}\right] \left[\frac{a}{ab}\right] \left[\frac{abc}{c}\right]$

Top string = Bottom String = abcaaabc .

PCP = { ⟨P⟩ : P is a collection of dominos with a match }.

Cont…

Thm: PCP is undecidable.

Proof: We show reduction from A_{TM} via accepting computation histories

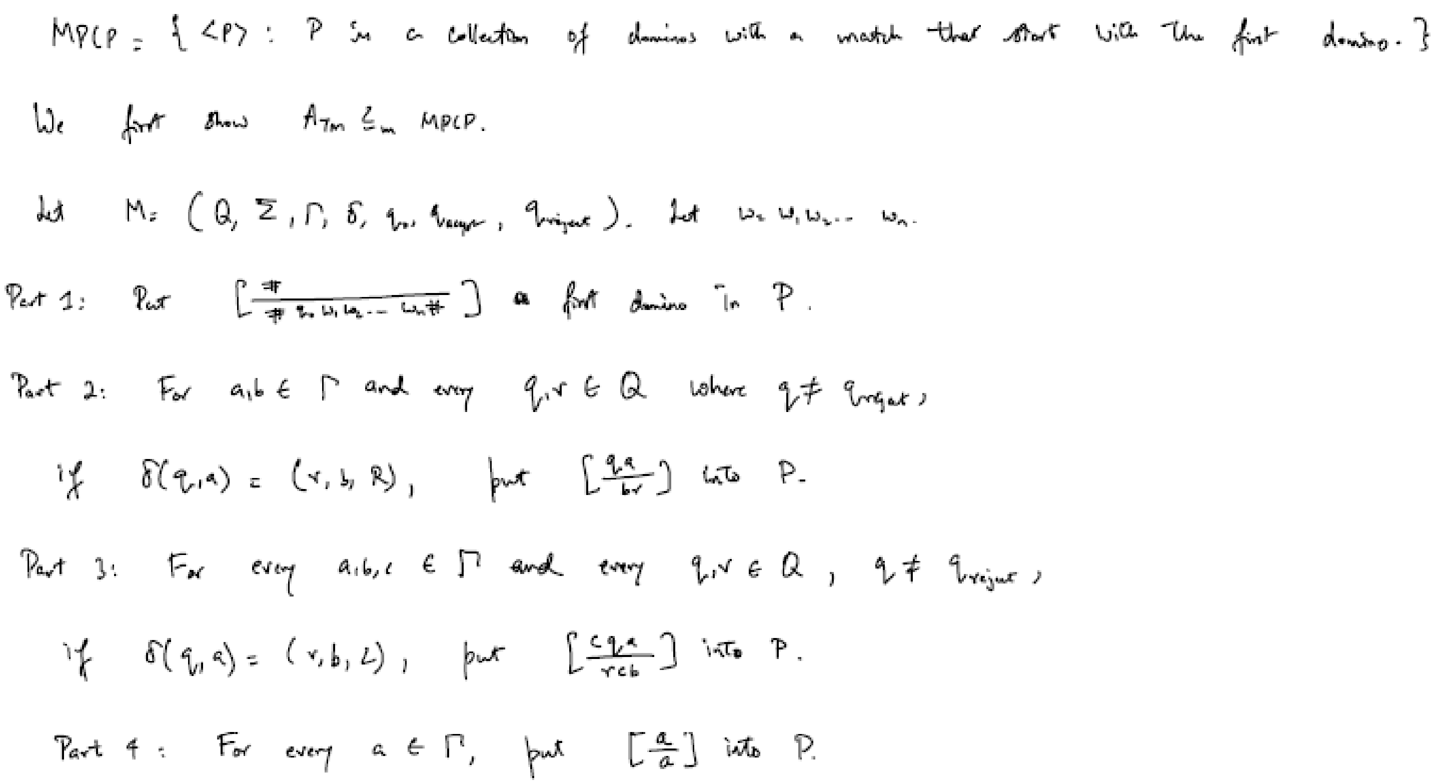
For any TM M and string w, we construct instance P of PCP where a match is an accepting computation history for M on w.

Simplifying assumptions: 1) M on w never attempts to move its head off the left hand end of the tape.

2) If $w = \epsilon$, we use $w = \sqcup$ in the construction of P.

3) We require that match starts with first domino $\left[\frac{t_1}{b_1}\right]$ on P.

Cont...



MPCP = { <P> : P is a collection of dominos with a match that start with the first domino. }

We first show $A_{TM} \le_m$ MPCP.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$. Let $w = w_1 w_2 \ldots w_n$.

Part 1: Put $\left[\frac{\#}{\# q_0 w_1 w_2 \ldots w_n \#}\right]$ a first domino in P.

Part 2: For $a, b \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{reject}$,

if $\delta(q, a) = (r, b, R)$, put $\left[\frac{qa}{br}\right]$ into P.

Part 3: For every $a, b, c \in \Gamma$ and every $q, r \in Q$, $q \neq q_{reject}$,

if $\delta(q, a) = (r, b, L)$, put $\left[\frac{cqa}{rcb}\right]$ into P.

Part 4: For every $a \in \Gamma$, put $\left[\frac{a}{a}\right]$ into P.

Cont…

Part 5: Put $\left[\frac{\#}{\#}\right]$ and $\left[\frac{\#}{\sqcup\#}\right]$ into P.

Part 6: For every $a \in \Gamma$, put

$\left[\frac{a\, q_{accept}}{q_{accept}}\right]$ and $\left[\frac{q_{accept}\, a}{q_{accept}}\right]$ into P.

Part 7: Add $\left[\frac{q_{accept}\, \#\#}{\#}\right]$ into P.

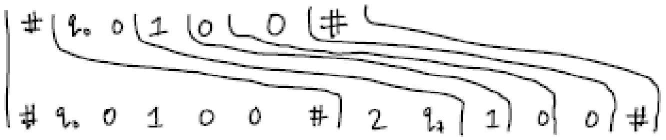
Match beginning, let $w = 0100$.

$$\left|\begin{array}{l} \# \\ \# q_0 0 1 0 0 \# \end{array}\right|$$

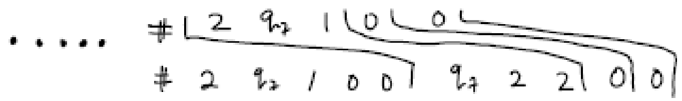
Let $\delta(q_0, 0) = (q_7, 2, R)$, so we have domino $\left[\frac{q_0\, 0}{2\, q_7}\right]$ in P.

We also have $\left[\frac{0}{0}\right]$, $\left[\frac{1}{1}\right]$, $\left[\frac{2}{2}\right]$ and $\left[\frac{\sqcup}{\sqcup}\right]$ in P.

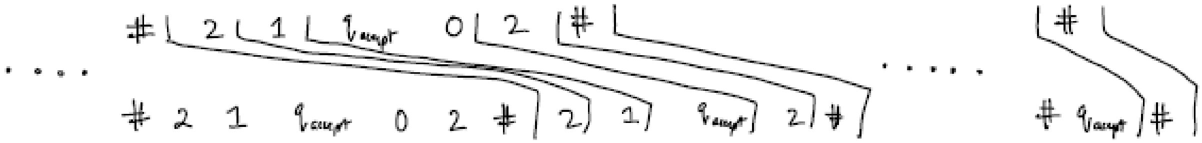
Cont…

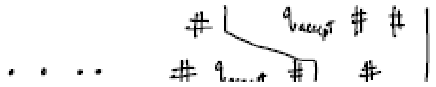


Let $\delta(q_7, 1) = (q_9, 2, L)$; we have $\left[\frac{2\, q_7 1}{q_7 2 2}\right]$ in P.



Suppose we reach q_{accept}





Cont…

Converting instance P of MPCP to instance P' of PCP

For string $u = u_1 \cdots u_n$, let

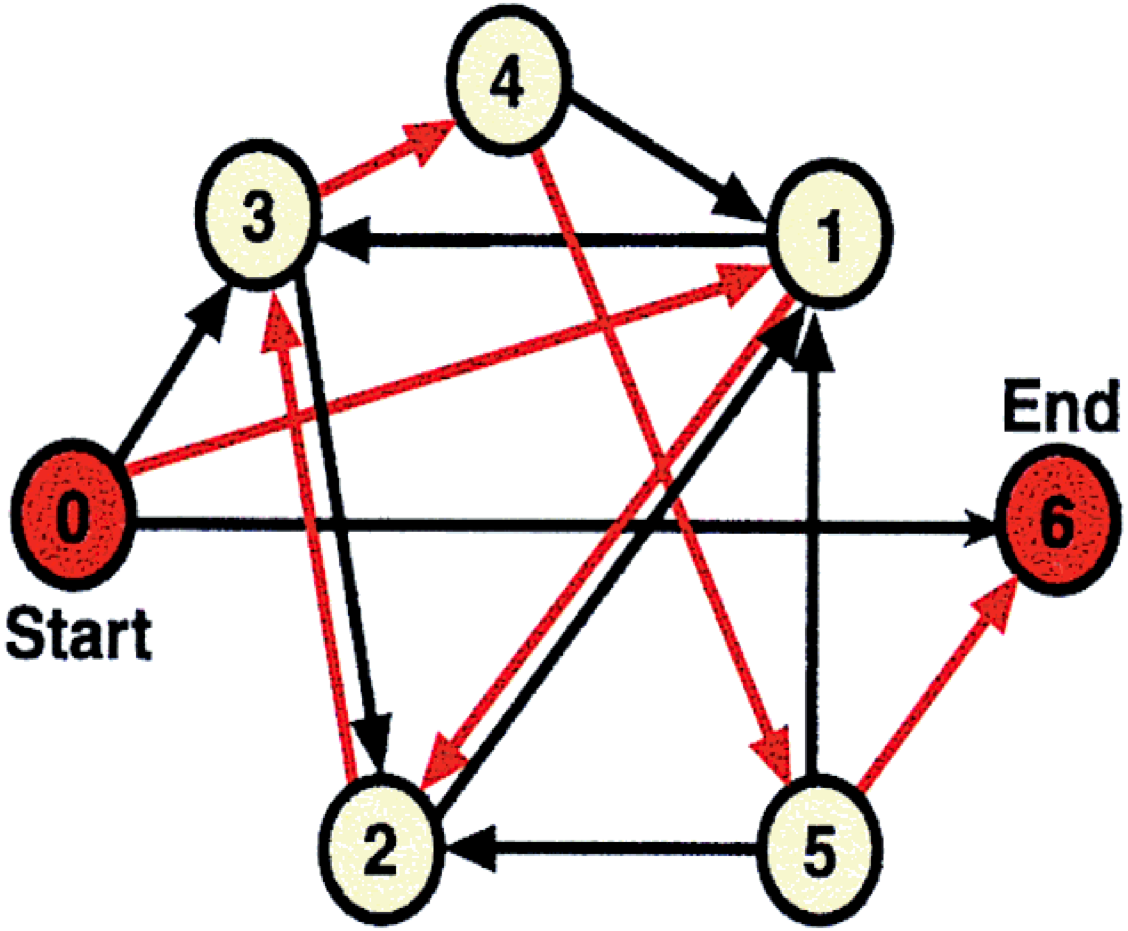
$*u = *u_1 * u_2 \cdots * u_n$; $u* = u_1 * u_2 * \cdots u_n *$; $*u* = *u_1 * u_2 \cdots * u_n *$.

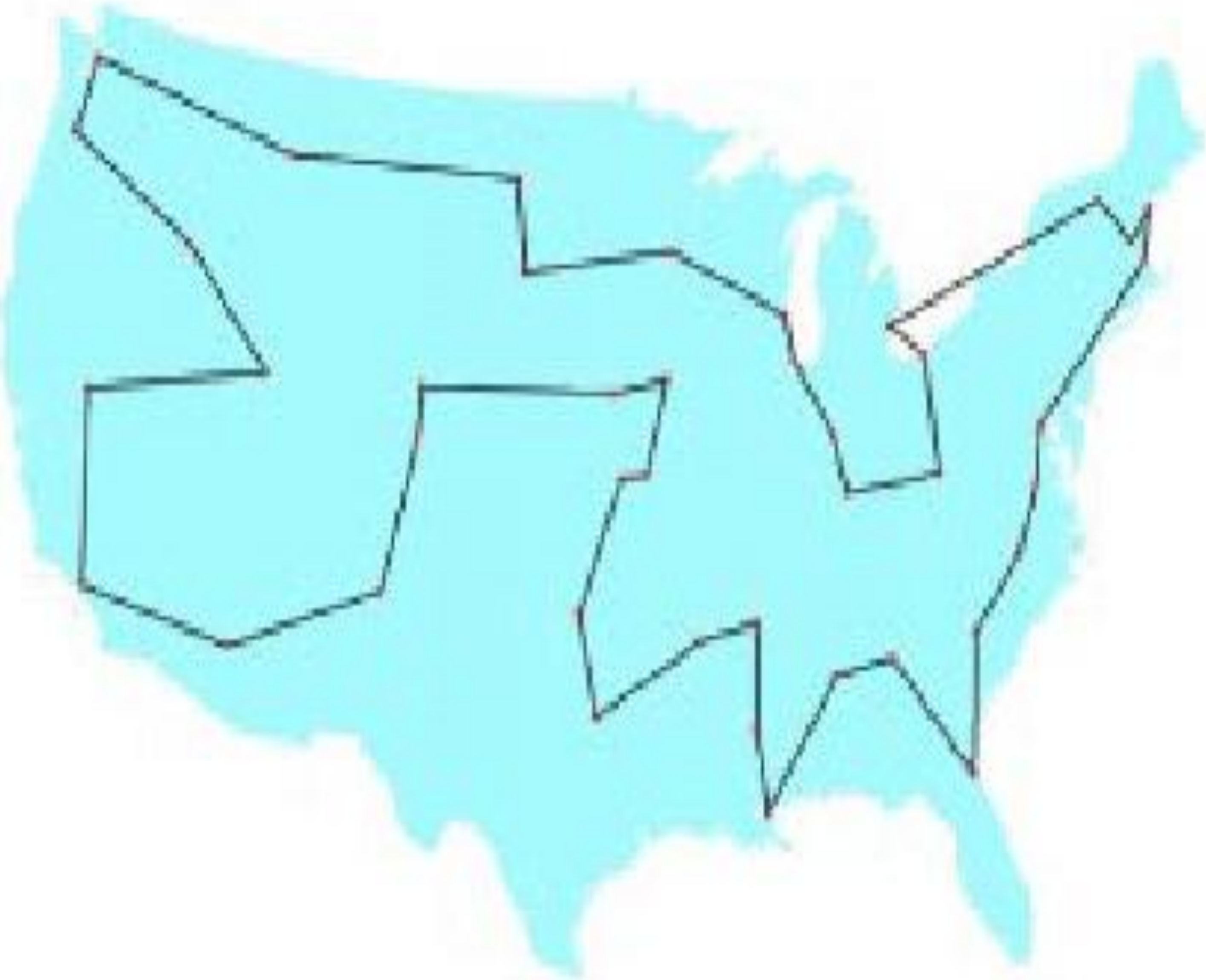
Let $P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \cdots, \left[\frac{t_k}{b_k}\right] \right\}$ instance starting with $\left[\frac{t_1}{b_1}\right]$.

Then let $P' = \left\{ \left[\frac{*t_1}{*b_1*}\right], \left[\frac{*t_1}{b_1*}\right], \left[\frac{*t_2}{b_2*}\right], \cdots, \left[\frac{*t_k}{b_k*}\right], \left[\frac{*\Diamond}{\Diamond}\right] \right\}$.

NP-complete Problem

Hamiltonian Path





The traveling salesperson problem: find the shortest tour connecting n cities?

Decision Problems

To keep things simple, we will mainly concern ourselves with decision problems. These problems only require a single bit output: ``yes" and ``no".

How would you solve the following decision problems?

Is this directed graph acyclic?

Is there a spanning tree of this undirected graph with total weight less than w?

Does this bipartite graph have a perfect (all nodes matched) matching?

Does the pattern p appear as a substring in text t?

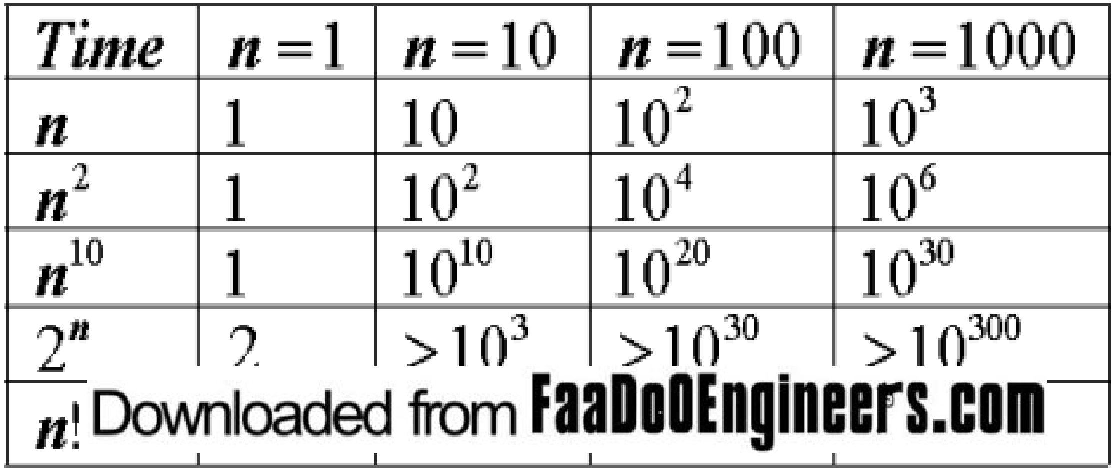
Algorithm Running Time

Given a size n problem, an algorithm runs $\boldsymbol{O(f(n))}$ time:

$\boldsymbol{1.O(f(n))}$: upper bound.   (Ω :lower θ: equal)

2.Polynomial: $f(n)=1$(constant), n(linear), n^2(quadratic), n^k.

3.Exponential: $f(n)=2^n$, $n!$, n^n.



Time	$n=1$	$n=10$	$n=100$	$n=1000$
n	1	10	10^2	10^3
n^2	1	10^2	10^4	10^6
n^{10}	1	10^{10}	10^{20}	10^{30}
2^n	2	$>10^3$	$>10^{30}$	$>10^{300}$
$n!$	[illegible]	[illegible]	[illegible]	[illegible]

P

P is the set of decision problems that can be solved in worst-case polynomial time:

If the input is of size n, the running time must be O(nk).
Note that k can depend on the problem class, but not the particular instance.

All the decision problems mentioned above are in P.

Nice Puzzle

The class NP (meaning non-deterministic polynomial time) is the set of problems that might appear in a puzzle magazine: ``Nice puzzle."

What makes these problems special is that they might be hard to solve, but a short answer can always be printed in the back, and it is easy to see that the answer is correct once you see it.

Example... Does matrix A have an LU decomposition?

No guarantee if answer is ``no".

Intractable Problems

Types of problems

- □ Decision problems: Given an input x and a poperty P we wish to decide if $P(x)$ is true (1) or false (0).

  Examples. Given $G=(V,E)$ and $u,v \in V$, decide if there is a path between u and v.

  Given $G=(V,E)$ and $w: E \rightarrow \mathbb{R}^{+}$ and $k \in \mathbb{N}$, is theree a minimum spanning tree of weight $\leq k$?.

- □ Function problems: Given an input x and a predicate Q we wish to compute y such that $Q(x,y)$.

  Examples: Given $G=(V,E)$ and $u,v \in V$, compute the shortest path between u and v.

  Given $G=(V,E)$ and $w: E \rightarrow \mathbb{R}^{+}$ find a spanning tree of minimum weight.

NP

Technically speaking:

A problem is in NP if it has a short accepting certificate.

An accepting certificate is something that we can use to quickly show that the answer is ``yes" (if it is yes).

Quickly means in polynomial time.

Short means polynomial size.

This means that all problems in P are in NP (since we don't even need a certificate to quickly show the answer is ``yes").

But other problems in NP may not be in P. Given an integer x, is it composite? How do we know this is in NP?

Good Guessing

Another way of thinking of NP is it is the set of problems that can solved efficiently by a really good guesser.

The guesser essentially picks the accepting certificate out of the air (Non-deterministic Polynomial time). It can then convince itself that it is correct using a polynomial
time algorithm. (Like a right-brain, left-brain sort of thing.)

Clearly this isn't a practically useful characterization: how could we build such a machine?

Exponential Upperbound

Another useful property of the class NP is that all NP problems can be solved in exponential time (EXP).

This is because we can always list out all short certificates in exponential time and check all O(2nk) of them.

Thus, P is in NP, and NP is in EXP. Although we know that P is not equal to EXP, it is possible that NP = P, or EXP, or neither. Frustrating!

NP-hardness

As we will see, some problems are at least as hard to solve as any problem in NP. We call such problems NP-hard.

How might we argue that problem X is at least as hard (to within a polynomial factor) as problem Y?

If X is at least as hard as Y, how would we expect an algorithm that is able to solve X to behave?

Algorithm Complexity

- P = solutions in polynomial deterministic time.
  - e.g. dynamic programming
- NP = (non-deterministic polynomial time) solutions checkable in deterministic polynomial time.
  - e.g. RSA code breaking by factoring
- NP-complete = most complex subset of NP
  - e.g. traveling all vertices with mileage < x
- NP-hard = optimization versions of above
  - e.g. Minimum mileage for traveling all vertices
- Undecidable = no way even with unlimited time & space
  - e.g. program halting problem

P and NP problems

- Assume we have a "conventional" deterministic computer.
  - The class of problems which can be solved on such a computer in polynomial time is called P (for Polynomial).
- Suppose we have a (theoretical) non-deterministic computer that can "guess" the right option when faced with choices.
  - The class of problems which can be solved on a non-deterministic computer in polynomial time is called NP (for Nondeterministic Polynomial).

- Partition Given $A = \{a_1, \ldots, a_n\}$ each a_i with $s(a_i) \in \int$ is there a $S \subset [n]$ s.t. $\sum_{i \in S} s(a_i) = \sum_{j \notin S} s(a_j)$?

certificate: S. To verify check in $O(n)$ that $\sum_{i \in S} s(a_i) = \sum_{j \notin S} s(a_j)$

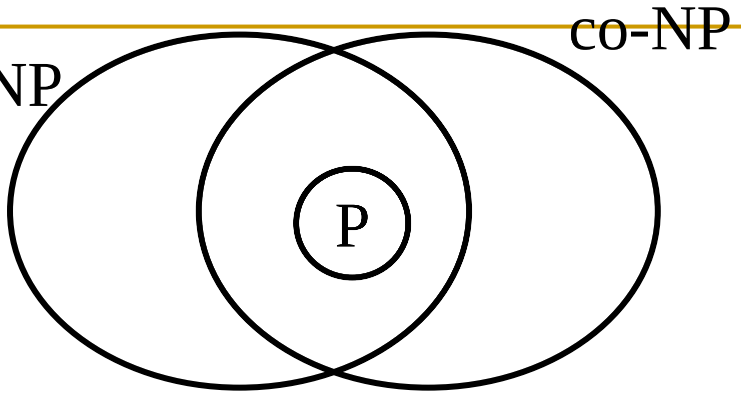
Theorem: P $\subseteq$ NP.

The US$ 10^6 Question: Is P $\neq$ NP or P = NP ?

http://www.claymath.org/prizeproblems/pvsnp.html

Is NP larger than P?

- Clearly, if a problem is in P it is also in NP. But what about the other way round?
- One might expect that such non-deterministic machines are more powerful (that is, that NP is larger than P).
- However, no one has found *a single problem* that is proven to be in NP but not in P.
- That is, if a problem is in NP, it might or might not be in P, so far as we know at present.
- In theory there *could be* efficient solutions to "hard" atisfiability.



One of the central (and widely and intensively studied 30 years) problems of (theoretical) computer science is to prove that

(a) **P✹NP** (b) **NP ✹ co-NP**.

- All evidence indicates that these conjectures are true.
- Disproving any of these two conjectures would not only be considered truly spectacular, but would also come as a tremendous surprise (with a variety of far-reaching counterintuitive consequences).

NP-complete: Collection Z of problems is NP-complete if (a) it is NP and (b) if polynomial-time algorithm existed for solving problems in Z, then P=NP.

NP-completeness

A problem $A \in$ NP is NP-complete if for every Bin NP, $B \leq A$. If for Bin NP, $B \leq A$ but $B \notin$ NP then A is said to be NP-hard.

Lemma: If A is NP-complete, the A in P iff $P = NP$.

So once we prove that a problem is NP-complete, either A has no efficient algorithm or all NP problemas are in P.

Majority conjecture: P ≠ NP

NP-complete problems

- The boolean satisfiability problem has been proved (by S.A. Cook (1971)) to be NP-complete.
- That is, Cook proved that if the boolean satisfiability problem (which was discussed above) is in P, then so are all problems in NP.
- Researchers have since shown that many other problems are NP-complete.
  - Proof of NP-completeness consists of mapping a given problem to the boolean satisfiability problem (or s oblem).

Some NP-complete problems

- Many practical problems are NP-complete.
  - Given a linear program (a set of linear inequalities) is there an integer solution to the variables?
  - Given a set of integers, can they be divided into two sets whose sum is equal?
  - Given two identical processors, a set of tasks of varying length, and a deadline, can the tasks be scheduled so that they finish before the deadline?
  - If there is an efficient solution to any of these, then all NP problems have efficient solutions! This would have a major impact

P=NP or P≠NP?

- Proving whether P=NP or P≠NP is one of the most important open problems in computer science.
- If someone showed that P=NP, then many "hard" problems (i.e. The NP-complete problems) would be tractable.
- However most computer scientists believe that P≠NP, largely because there are many problems which are in NP but for which no one has found an efficient solution.
    - That is, absence of evidence that P=NP counts as evi[illegible]

Summary: P and NP

- Some problems seem to be intrinsically very complex (NP). The only “efficient” known solutions require a non-deterministic computer.
- At present we have no proof that such problems do not have efficient solutions (they could be in P).
- Some NP problems are significant in the sense that if they are in P, then so are all NP problems.

NP-Complete Problems

A formal-language framework:

- Alphabet Σ
- String (empty string ϵ)
- Language L over Σ (empty language $\emptyset$, Σ^*)
- Decision problem as a language $L = \{x \in \Sigma^* : Q(x) = 1\}$
- Accept, Decide (in polynomial time, by an algorithm)
  - common points
  - distinctions
- Define class P:

P $= \{L \subseteq \{$

algorithm}

Polynomial time verification (2):

- Hamiltonian Cycle Problem

  1. instances: (undirected, simple) graphs G
  2. query: is there a Hamiltonian cycle in G?

  So far we don't if the problem is in P or not ...

- Suppose your answer is 'yes'. I can still verify the truth if you can pass me a cycle C, no matter how you get it:

  – C is a simple cycle

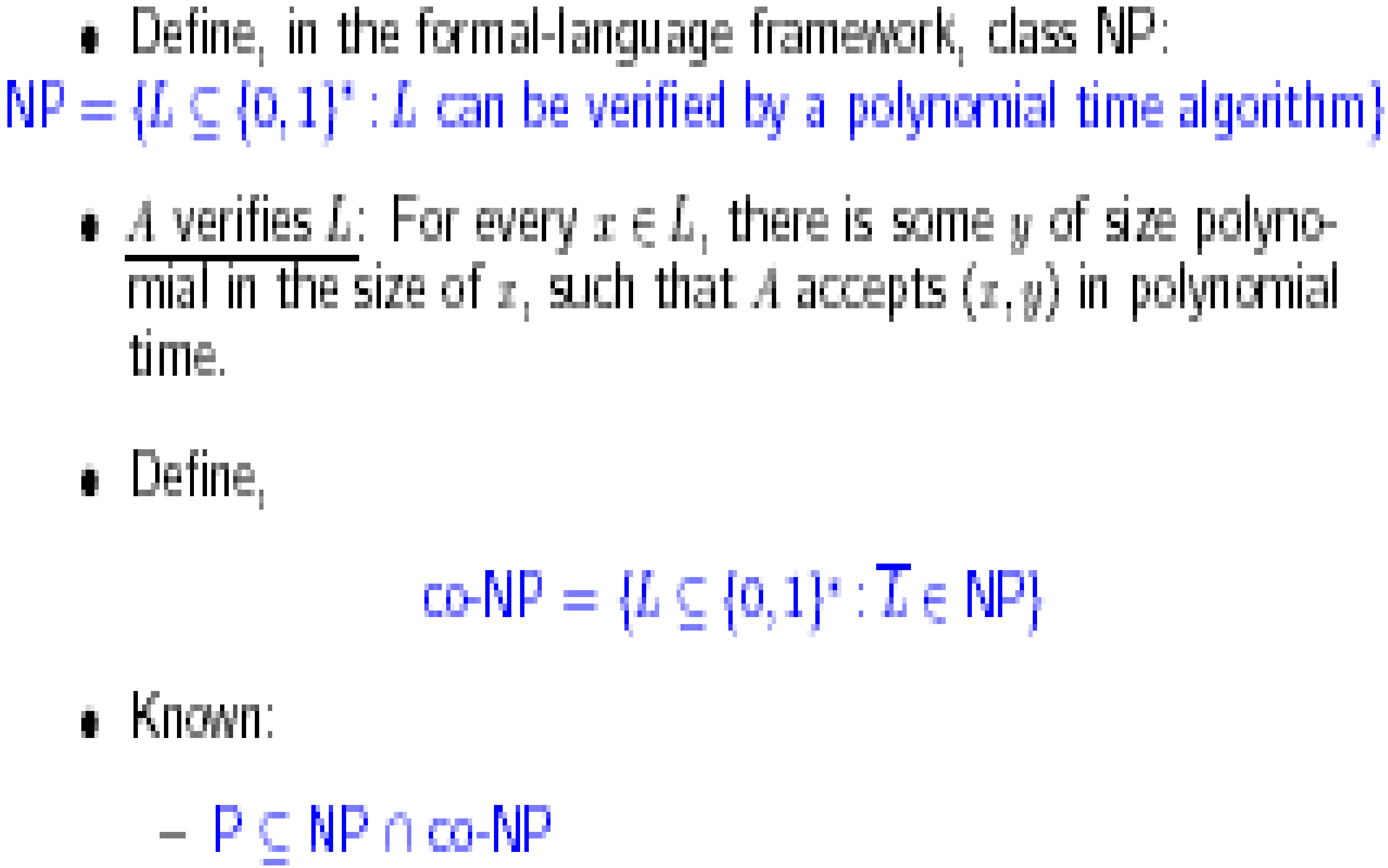
  – C passes every vertex

  – amount of time spent: polynomial — $\Theta(|V| + |E|)$

- Verification algorithm

  – input: the instance of the decision problem and the certificate

  – output: 1 if the certificate supports the answer "yes"



- Define, in the formal-language framework, class NP:

$\text{NP} = \{L \subseteq \{0,1\}^* : L \text{ can be verified by a polynomial time algorithm}\}$

- A verifies L: For every $x \in L$, there is some y of size polynomial in the size of x, such that A accepts (x, y) in polynomial time.

- Define,

$$\text{co-NP} = \{L \subseteq \{0,1\}^* : \overline{L} \in \text{NP}\}$$

- Known:
  - $\text{P} \subseteq \text{NP} \cap \text{co-NP}$

NP-complete:

- Informally, the hardest problem in class NP
  - if an NP-complete problem can be solved in polynomial time, then every problem in NP can be too

- Formally,
  - a language L_1 is polynomial-time reducible to a language L_2, if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

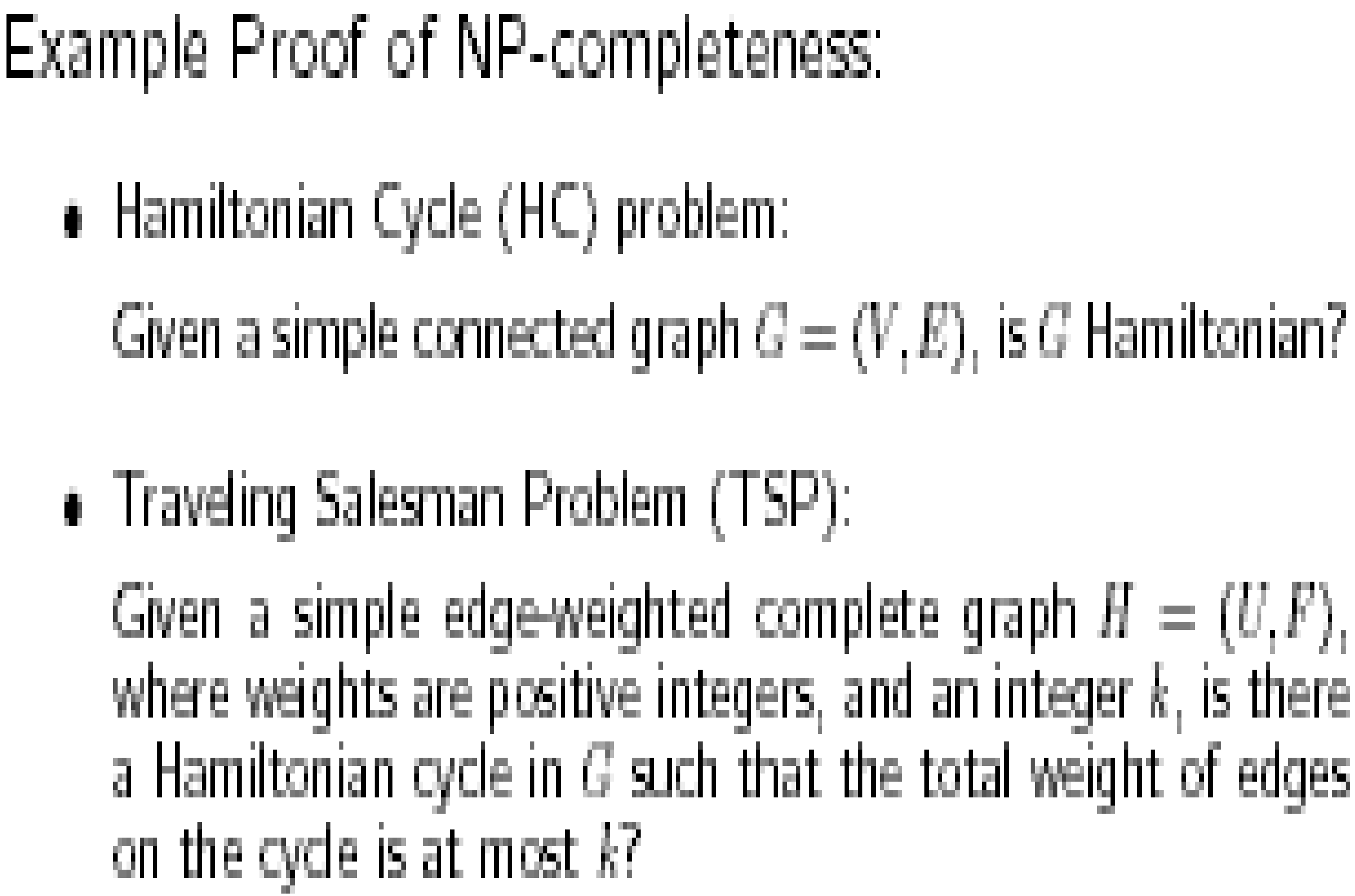
$$x \in L_1 \text{ if and only if } f(x) \in L_2.$$

written as $L_1 \leq_p L_2$

NP-completeness:

- Class NPC:
- Some conclusions:
  1. if one NP-complete is solvable in polynomial time, then $P = NP$
  2. if $P \neq NP$, then $NPC \neq \emptyset$
- Where or not $P = NP$ is one of the most fundamental problems in CS.
- Since there are so many smart people who cannot solve the problem, if we are lazy then we show people the problems are NP-complete.

No, showing NP-completeness doesn't end the story. We will see later.



Example Proof of NP-completeness:

- Hamiltonian Cycle (HC) problem:

  Given a simple connected graph $G = (V, E)$, is G Hamiltonian?

- Traveling Salesman Problem (TSP):

  Given a simple edge-weighted complete graph $H = (U, F)$, where weights are positive integers, and an integer k, is there a Hamiltonian cycle in G such that the total weight of edges on the cycle is at most k?

Assume that HC is NP-complete, prove that TSP is NP-complete too.

Proof.

1. TSP $\in$ NP
2. HC is NP-complete
3. Given an instance of HC, *i.e.*, a simple graph $G = (V, E)$, need to construct an instance of TSP $(H = (U, F), k)$:
   (a) $U \leftarrow V$
   (b) H is complete on U
   (c) edges in E get weight 1, edges not in E get weight 2
   (d) $k \leftarrow |U|$

   This construction takes polynomial time — the F
4. if G is Hamiltonian, then there is an HC in H with weight $k = |U|$
5. if there is an HC in H with weight $k = |U|$, then G is Hamilto