15-453

FORMAL LANGUAGES, AUTOMATA AND COMPUTABILITY

REVIEW for MIDTERM 1

THURSDAY Feb 6

Midterm 1 will cover everything we have seen so far

The PROBLEMS will be from Sipser,

Chapters 1, 2, 3

It will be Closed-Book, Closed-Everything

- **1. Deterministic Finite Automata and Regular Languages**
- **2. Non-Deterministic Finite Automata**
- **3. Pumping Lemma for Regular Languages; Regular Expressions**
- **4. Minimizing DFAs**
- **5. PDAs, CFGs;
Pumping Lemma for CFLs**
- **6. Equivalence of PDAs and CFGs**
- **7. Chomsky Normal Form**
- **8. Turing Machines**

Machines	Syntactic Rules
DFAs	
NFAs	Regular Expressions
PDAs	Context-Free Grammars

THE REGULAR OPERATIONS

Union: $A \cup B = \{ w \mid w \in A \text{ or } w \in B \}$

Intersection: $A \cap B = \{ w \mid w \in A \text{ and } w \in B \}$

Negation: $\neg A = \{ w \in \Sigma^* \mid w \notin A \}$

Reverse: $A^R = \{ w_1 \ldots w_k \mid w_k \ldots w_1 \in A \}$

Concatenation: $A \cdot B = \{ vw \mid v \in A \text{ and } w \in B \}$

Star: $A^* = \{ s_1 \ldots s_k \mid k \geq 0 \text{ and each } s_i \in A \}$

REGULAR EXPRESSIONS

σ is a regexp representing $\{\sigma\}$

ε is a regexp representing $\{\varepsilon\}$

$\varnothing$ is a regexp representing $\varnothing$

If R_1 and R_2 are regular expressions representing L_1 and L_2 then:

(R_1R_2) represents $L_1 \cdot L_2$

$(R_1 \cup R_2)$ represents $L_1 \cup L_2$

$(R_1)^*$ represents L_1^*

How can we test if two regular expressions are the same?

Length n	R_1		R_2
O(n) states	N_1		N_2
$O(2^n)$ states	M_1		M_2
	$M_{1\ MIN}$	?=	$M_{2\ MIN}$

How can we test if two regular expressions are the same?

Another way???

THEOREMS
and
CONSTRUCTIONS

THE PUMPING LEMMA

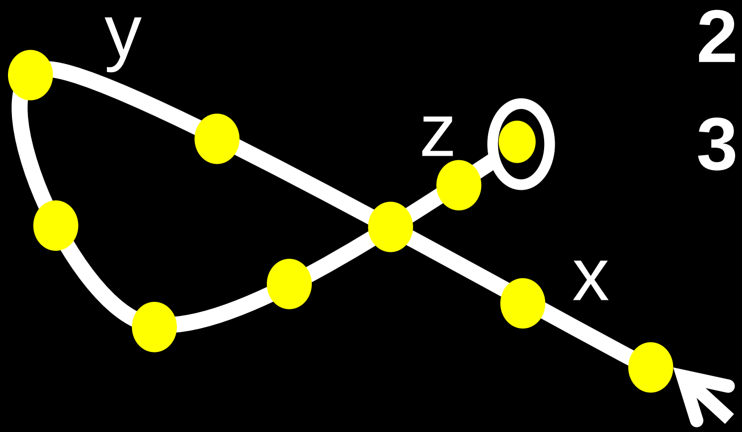
(for Regular Languages)

Let L be a regular language with |L| = ∞

Then there is an integer P such that

if $w \in L$ and $|w| \geq P$

then can write w = xyz, where:



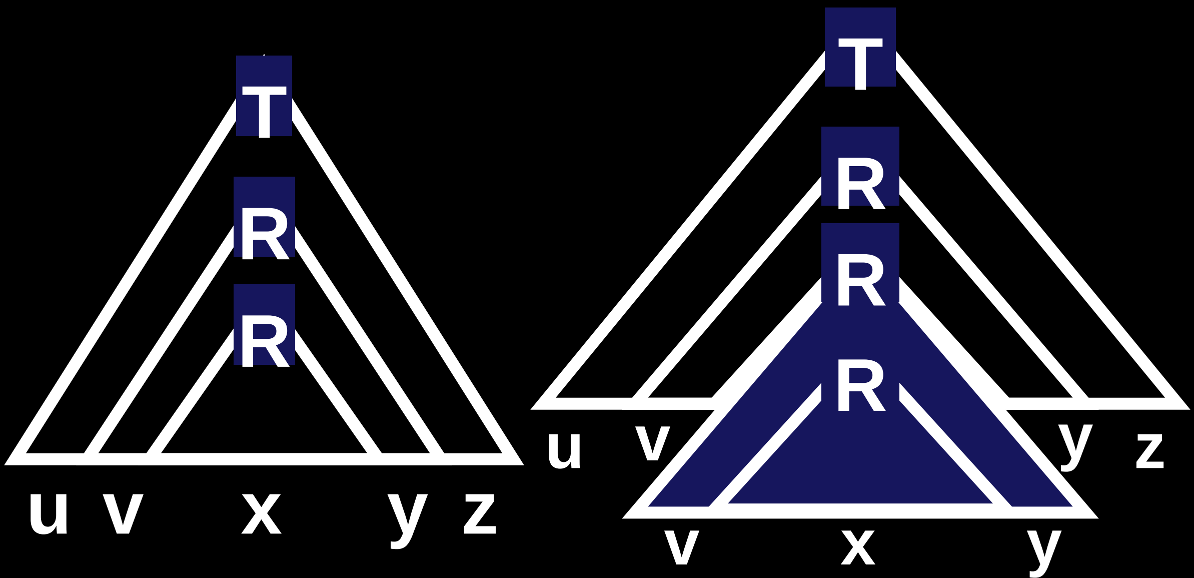
1. **$|y| > 0$**
2. **$|xy| \leq P$**
3. **$xy^iz \in L$ for any $i \geq 0$**

THE PUMPING LEMMA
(for Context Free Grammars)

Let L be a context-free language with |L| = ∞

Then there is an integer P such that
if w ∈ L and |w| ≥ P

then can write w = uvxyz, where:



1. **|vy| > 0**
2. **|vxy| ≤ P**
3. **uv^ixy^iz ∈ L, for any i ≥ 0**

CONVERTING NFAs **TO DFAs**

Input: NFA N = $(Q, \Sigma, \delta, Q_0, F)$

Output: DFA M = $(Q', \Sigma, \delta', q_0', F')$

$$Q' = 2^Q$$

$$\delta' : Q' \times \Sigma \rightarrow Q'$$

$$\delta'(R,\sigma) = \bigcup_{r \in R} \varepsilon(\delta(r,\sigma)) \quad *$$

$$q_0' = \varepsilon(Q_0)$$

$$F' = \{ R \in Q' \mid f \in R \text{ for some } f \in F \}$$

*

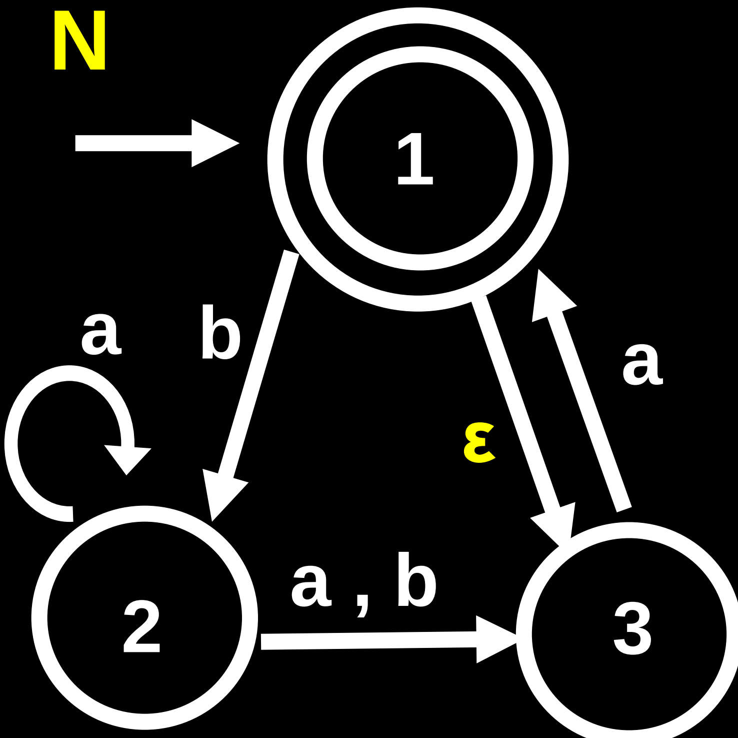
For **R** $\subseteq$ **Q,** the **ε-closure of R, ε(R)** = **{q** that can be reached from some r $\in$ **R** by traveling along zero or more **ε** arrows}

Given: NFA **N** = ({1,2,3}, {a,b}, δ , {1}, {1})

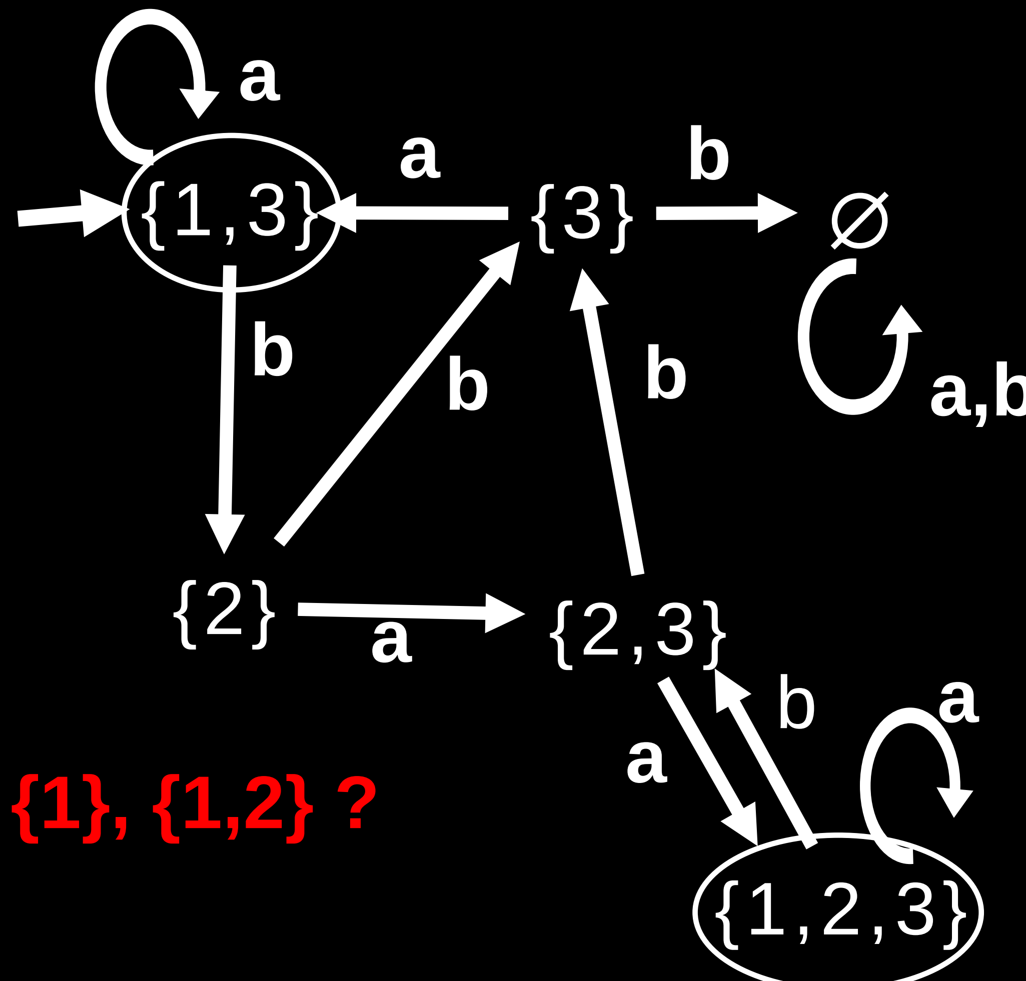
Construct: Equivalent DFA **M**

M = ($2^{\{1,2,3\}}$, {a,b}, δ′, {1,3}, …)

N



ε({1}) = {1,3}



{1}, {1,2} ?

EQUIVALENCE

L can be represented by a regexp

⇔

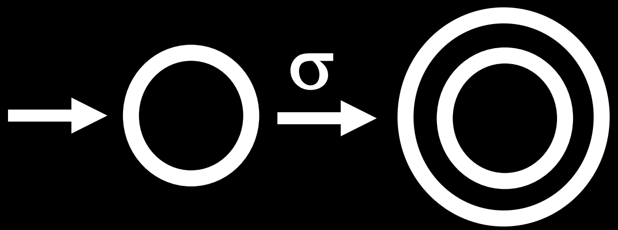
L is a regular language

✓ L can be represented by a regexp $\Rightarrow$ L is a regular language

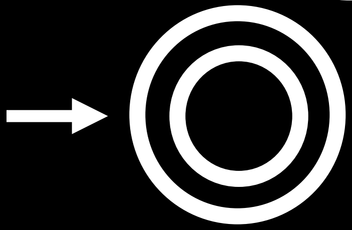
Induction on the length of R:

Base Cases (R has length 1):

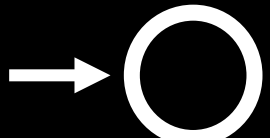
R = σ



R = ε



R = $\emptyset$



Inductive Step:

Assume R has length $k > 1$, and that every regexp of length $< k$ represents a regular language

Three possibilities for what R can be:

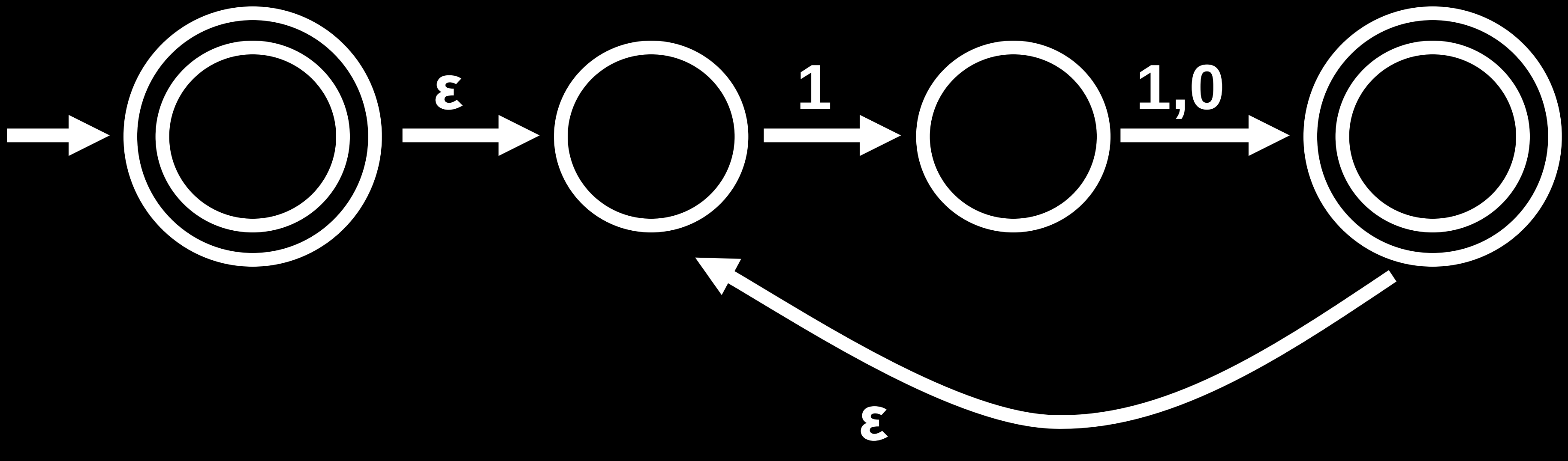
$R = R_1 \cup R_2$ (Closure under Union)

$R = R_1 R_2$ (Closure under Concat.)

$R = (R_1)^*$ (Closure under Star)

Therefore: L can be represented by a regexp $\Rightarrow$ L is regular

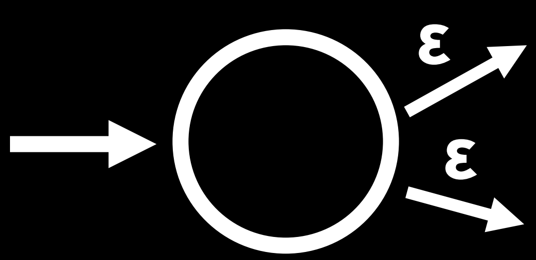
Transform (1(0 ∪ 1))* to an NFA

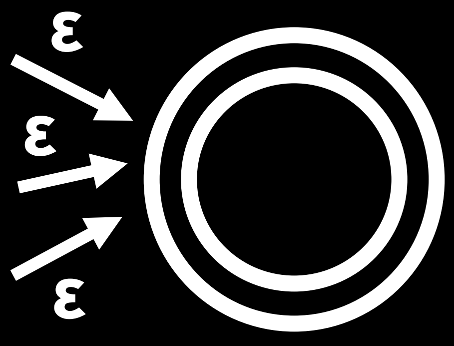


✓ L is a regular language ⇒ L can be represented by a regexp

Proof idea: Transform an NFA for L into a regular expression by **removing states** and re-labeling the arrows with regular expressions

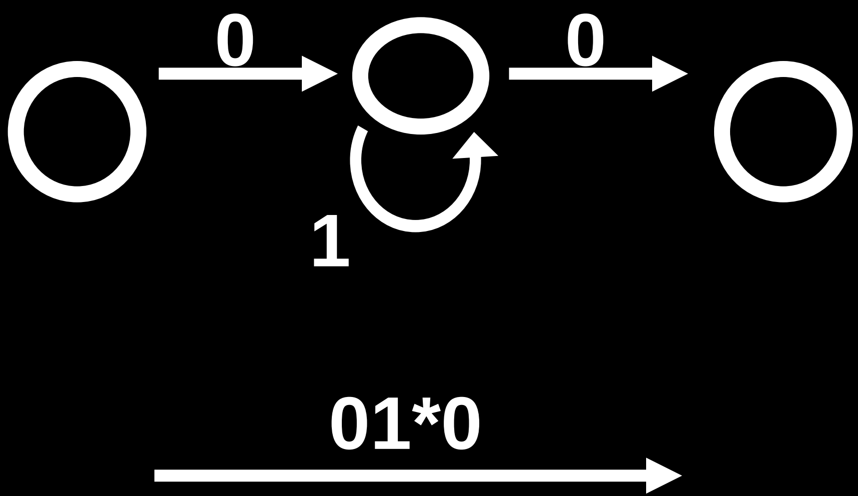
Add unique and distinct start and accept states

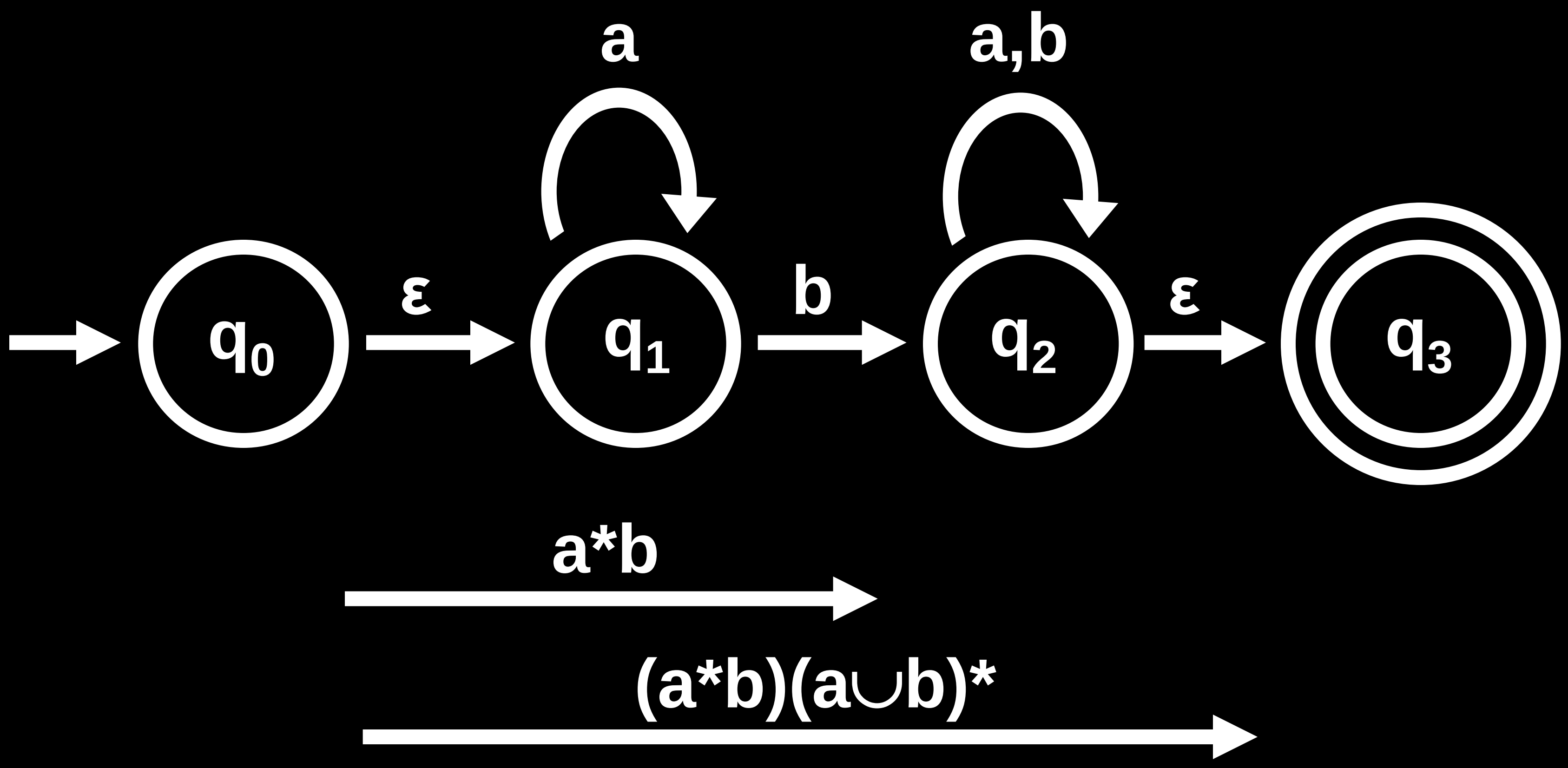




While machine has more than 2 states:

Pick an internal state, rip it out and re-label the arrows with regexps, to account for the missing state





$R(q_0,q_3) = (a^*b)(a \cup b)^*$

THEOREM

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

EXTENDING δ

Given DFA M = $(Q, \Sigma, \delta, q_0, F)$, extend δ to $\hat{\delta} : Q \times \Sigma^* \to Q$ as follows:

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

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

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

Note: $\delta(q_0, w) \in F \Leftrightarrow$ M accepts w

String $w \in \Sigma^*$ distinguishes states q_1 and q_2 iff exactly ONE of $\hat{\delta}(q_1, w)$, $\hat{\delta}(q_2, w)$ is a final state

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

Definition:

$p \sim q$ iff p is indistinguishable from q

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

Proposition: $\sim$ is an equivalence relation

$p \sim p$ (reflexive)

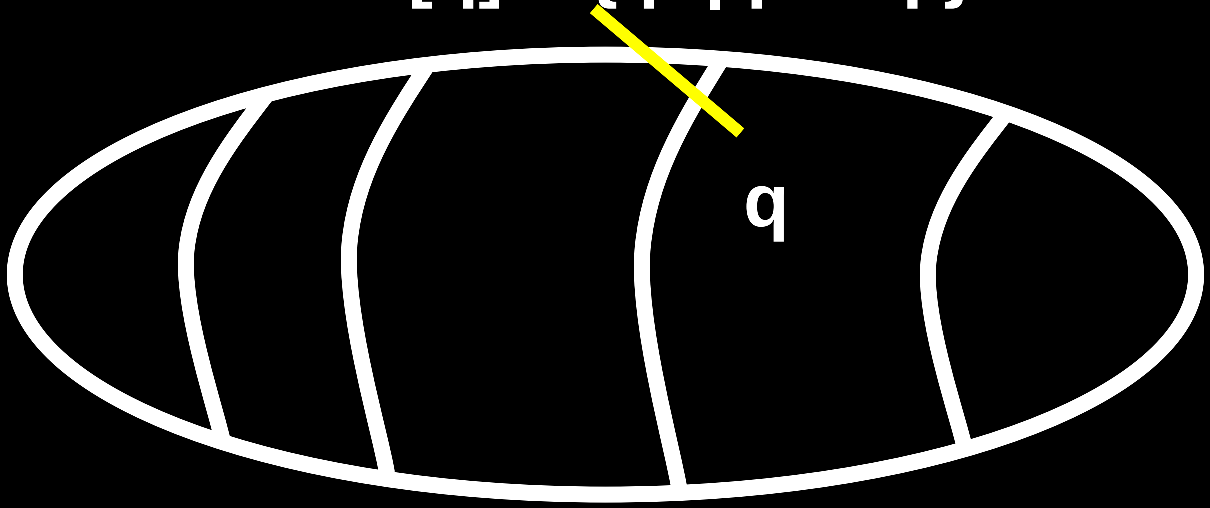
$p \sim q \Rightarrow q \sim p$ (symmetric)

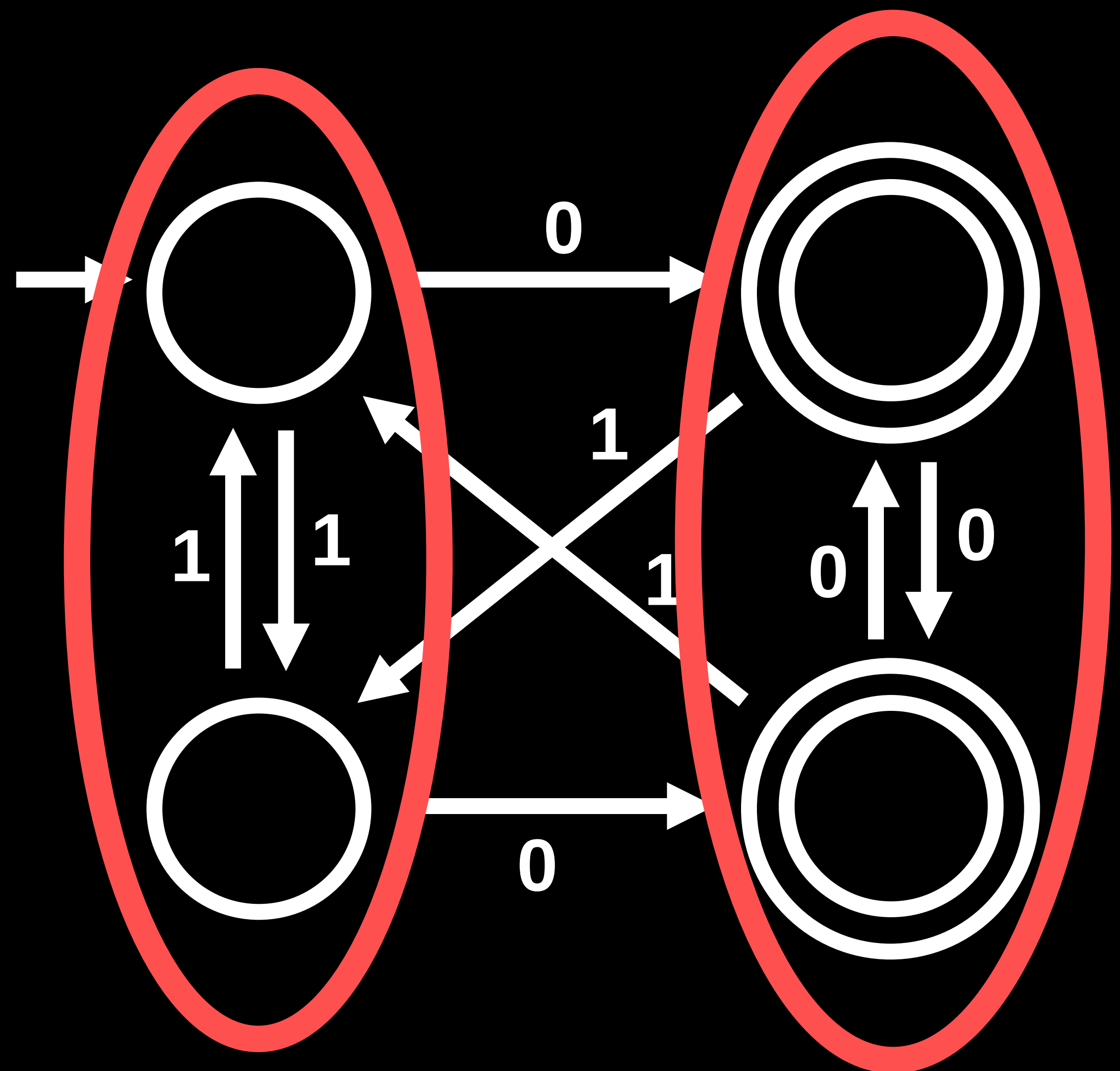
$p \sim q$ and $q \sim r \Rightarrow p \sim r$ (transitive)

Proposition: ~ is an equivalence relation

so ~ partitions the set of states of M into disjoint equivalence classes

[q] = { p | p ~ q }





0
1
1
1
1
0
0
0

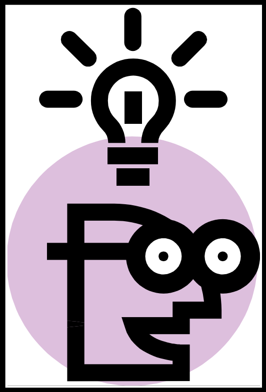
TABLE-FILLING ALGORITHM

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

Output: (1) $D_M = \{ (p,q) \mid p,q \in Q \text{ and } p \not\sim q \}$

(2) $E_M = \{ [q] \mid q \in Q \}$

IDEA:



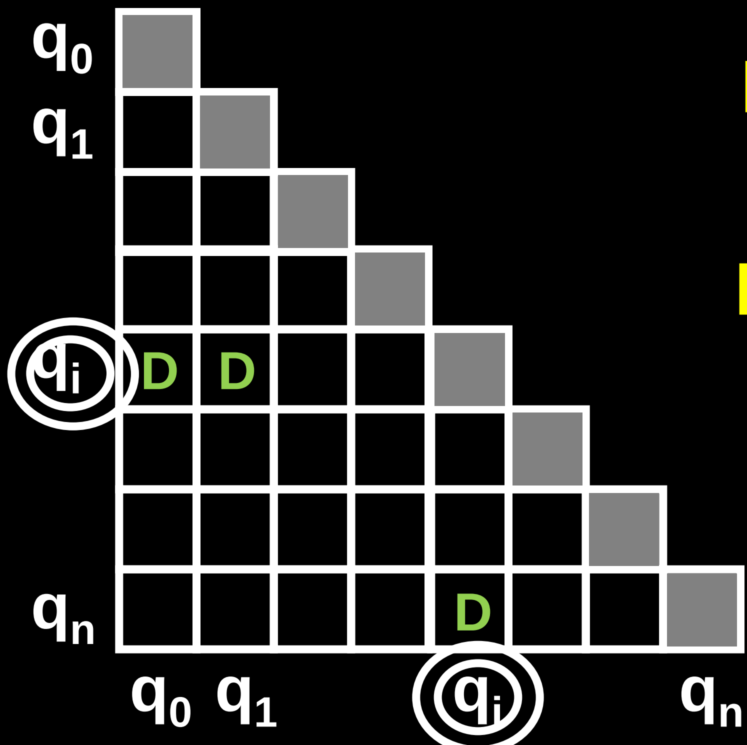
- **We know how to find those pairs of states that ε distinguishes…**
- **Use this and recursion to find those pairs distinguishable with *longer* strings**
- **Pairs left over will be indistinguishable**

TABLE-FILLING ALGORITHM

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

Output: (1) $D_M = \{ (p,q) \mid p,q \in Q \text{ and } p \not\sim q \}$

(2) $E_M = \{ [q] \mid q \in Q \}$



Base Case: **p accepts and q rejects $\Rightarrow p \not\sim q$**

Recursion: **if there is $\sigma \in \Sigma$ and states p', q' satisfying**

$$\delta(p, \sigma) = p'$$
$$\not\sim \quad \Rightarrow \quad p \not\sim q$$
$$\delta(q, \sigma) = q'$$

Repeat until no more new D's

Algorithm MINIMIZE

Input: DFA M

Output: DFA M_{MIN}

(1) Remove all inaccessible states from M

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

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

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

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

Claim: $M_{MIN} \equiv M$

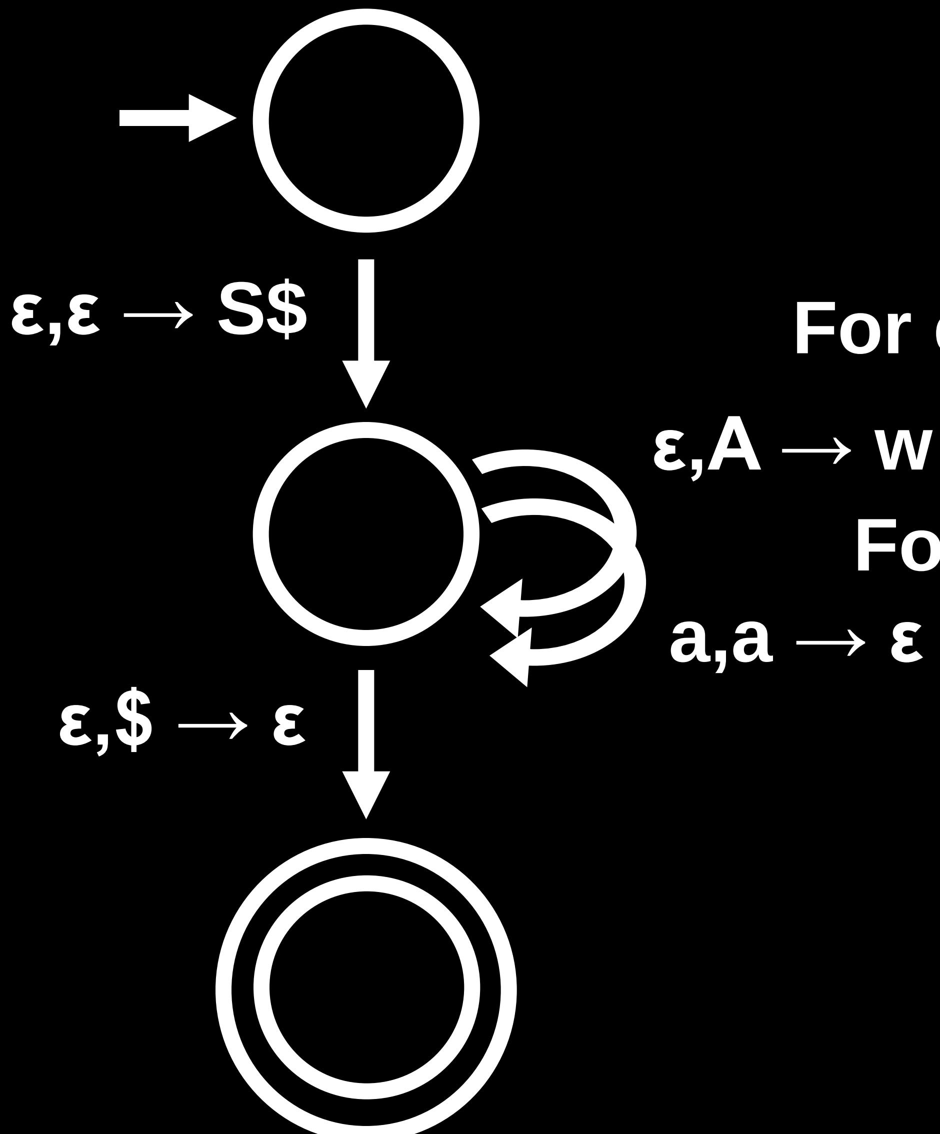
A Language L is generated by a CFG

$\Leftrightarrow$

L is recognized by a PDA

Suppose L is generated by a CFG G = (V, Σ, R, S)

Construct P = (Q, Σ, Γ, δ, q, F) that recognizes L

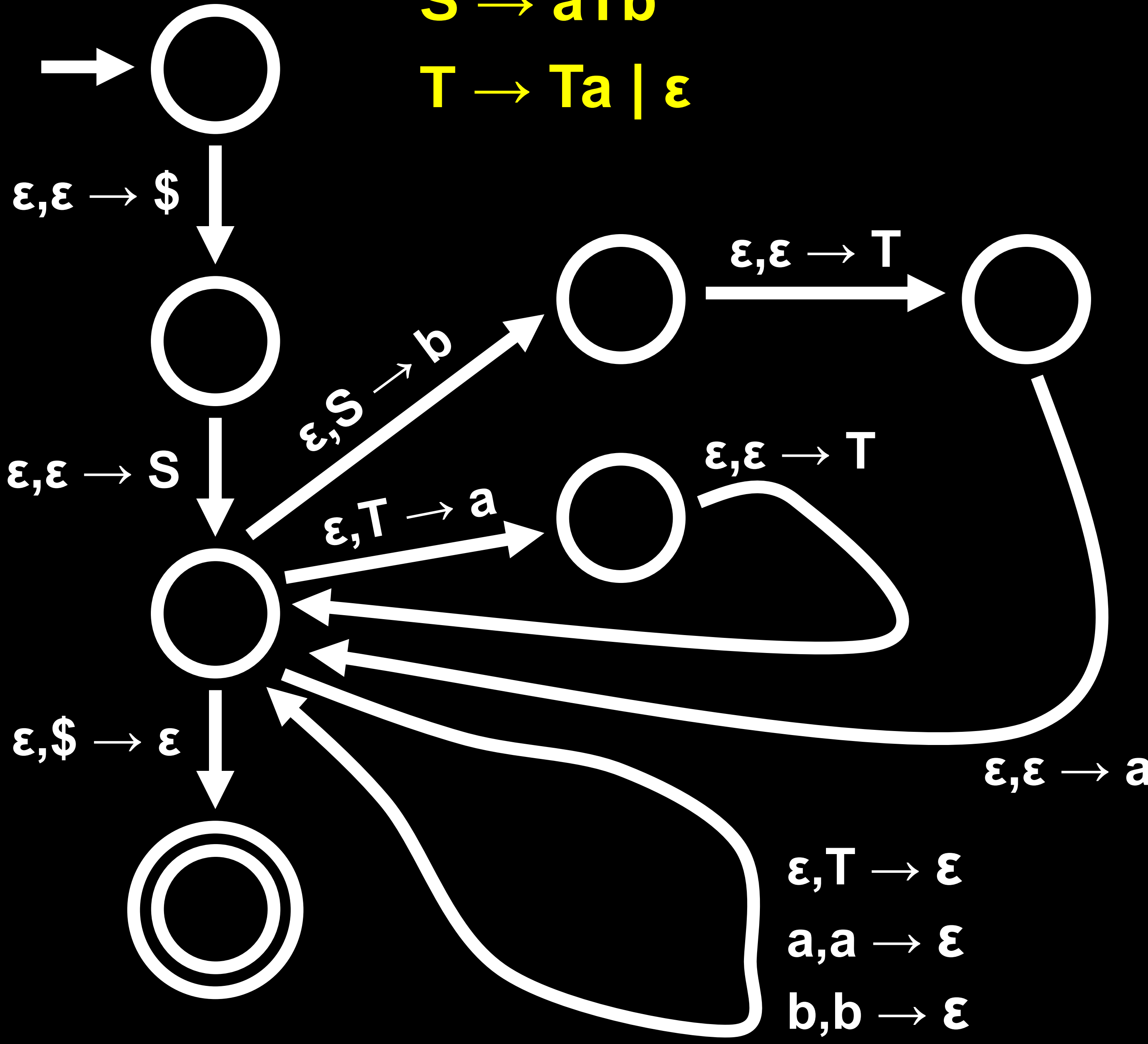


For each rule 'A → w' ∈ R:

For each terminal a ∈ Σ:

S → aTb

T → Ta | ε



A Language L is generated by a CFG
⇐
L is recognized by a PDA

Given PDA **P** = (Q, Σ, Γ, δ, q, F)

Construct a CFG **G** = (V, Σ, R, S) such that L(**G**) = L(**P**)

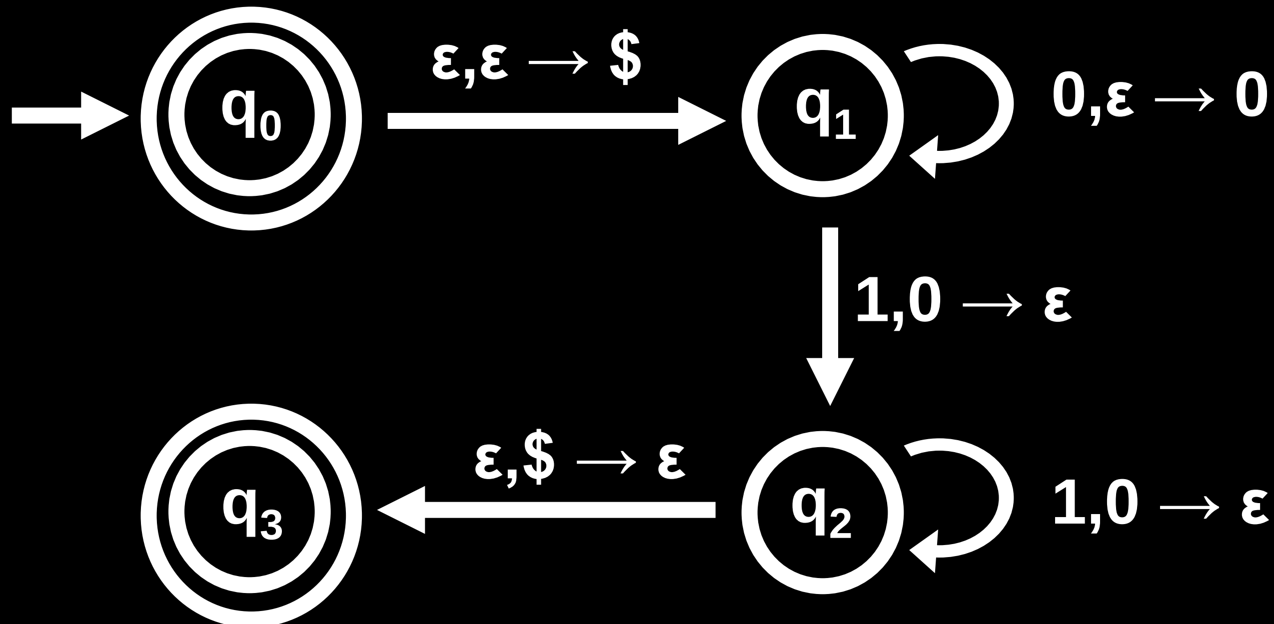
First, **simplify P** to have the following form:

(1) It has a single accept state, q_{accept}

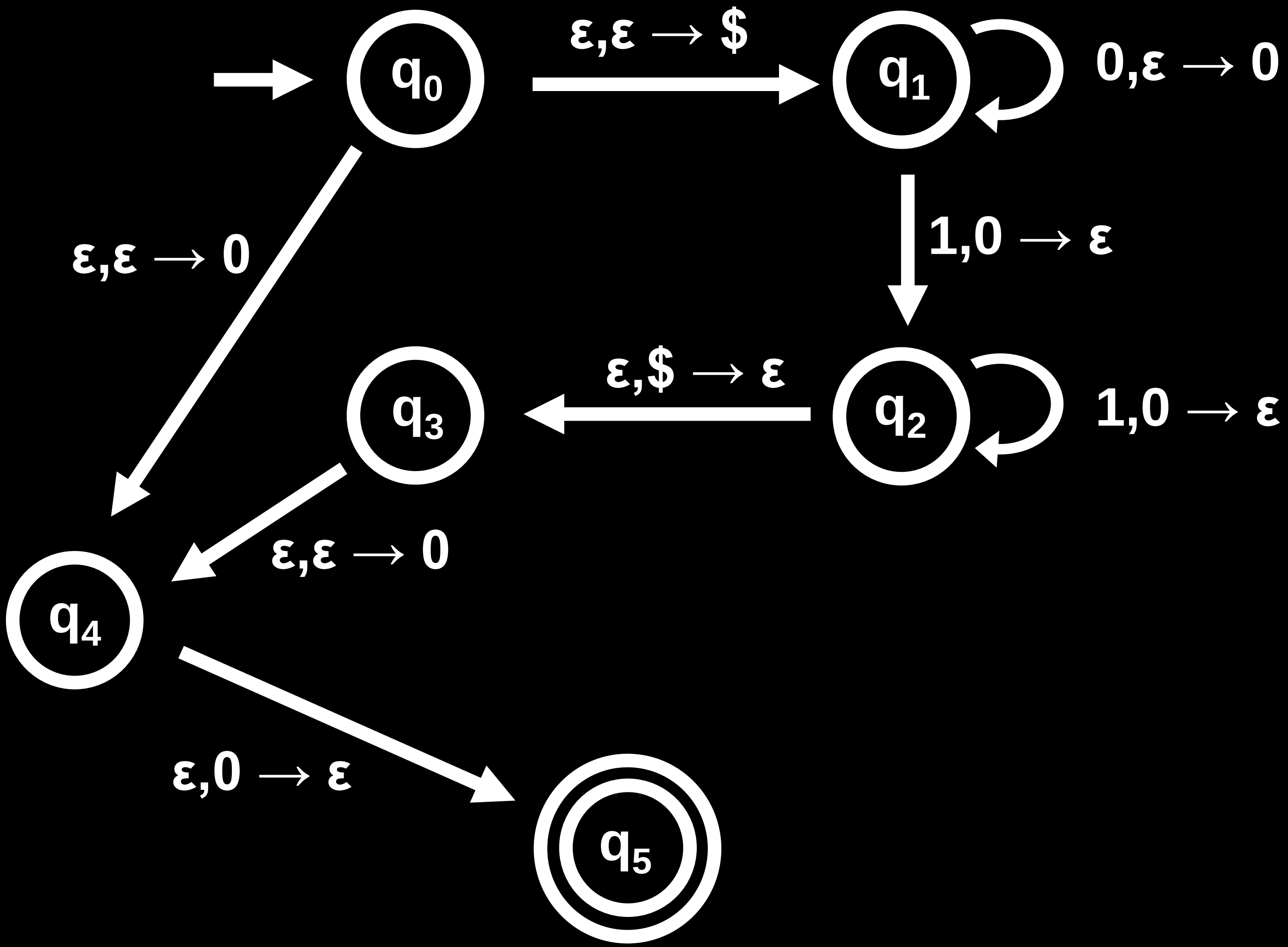
(2) It empties the stack before accepting

(3) Each transition either pushes a symbol or pops a symbol, but not both at the same time

SIMPLIFY



SIMPLIFY



Idea For Our Grammar G:

For every pair of states p and q in PDA P,

G will have a variable A_{pq} which generates all strings x that can take:

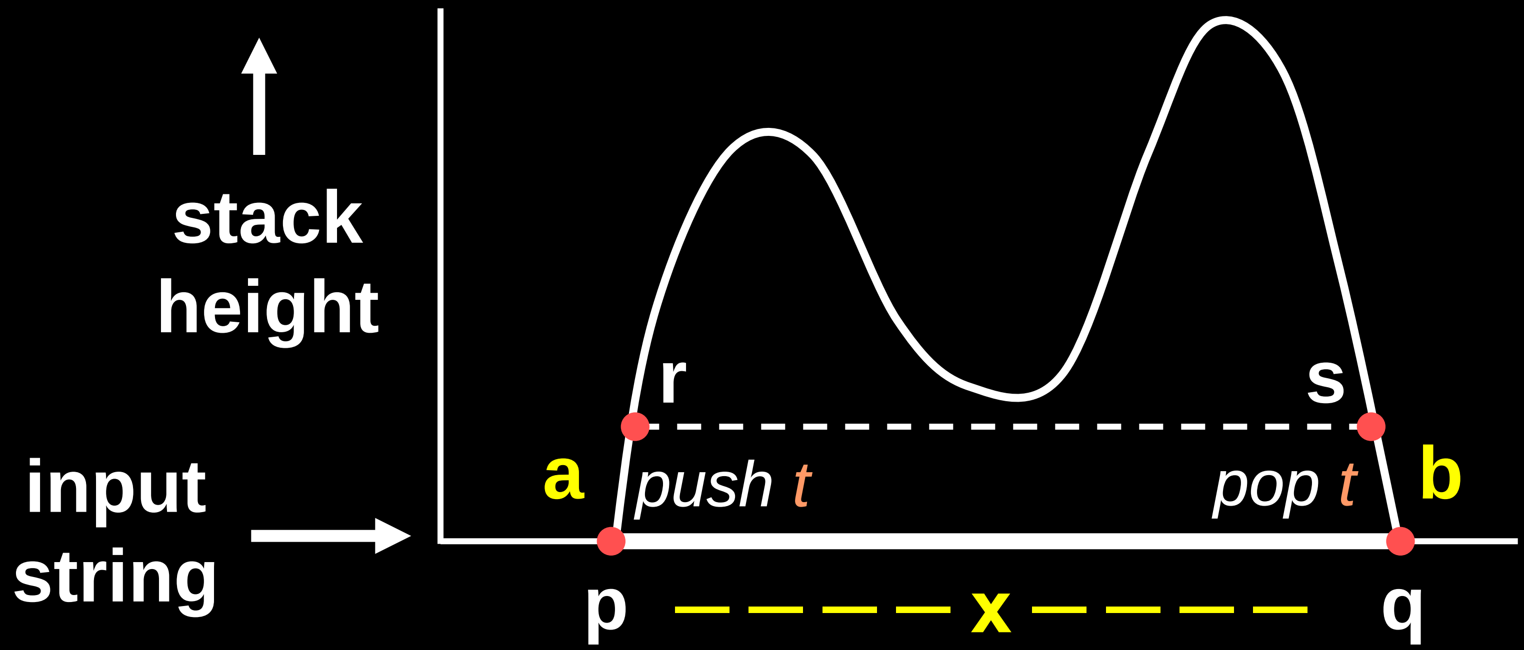
P from p with an empty stack
to q with an empty stack

$$V = \{A_{pq} \mid p,q \in Q\}$$

$$S = A_{q_0 q_{acc}}$$

x = ayb takes **p** with empty stack to **q** with empty stack

1. The symbol t popped at the end is exactly the one pushed at the beginning

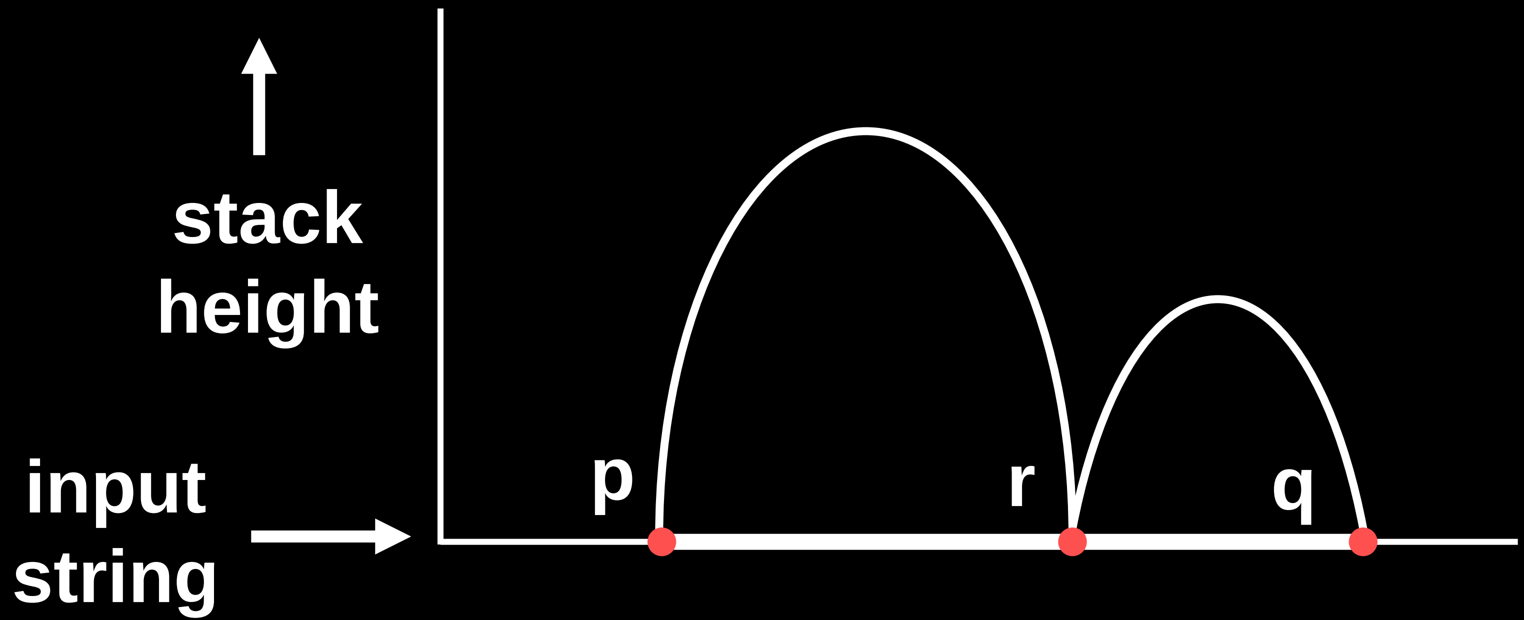


$\delta(p, a, \varepsilon) \rightarrow (r, t)$

$\delta(s, b, t) \rightarrow (q, \varepsilon)$

$A_{pq} \rightarrow aA_{rs}b$

2. The symbol popped at the end is not the one pushed at the beginning



$$A_{pq} \rightarrow A_{pr}A_{rq}$$

Formally:

$V = \{A_{pq} \mid p, q \in Q\}$

$S = A_{q_0 q_{acc}}$

For every $p, q, r, s \in Q$, $t \in \Gamma$ and $a, b \in \Sigma_\varepsilon$

If $(r, t) \in \delta(p, a, \varepsilon)$ and $(q, \varepsilon) \in \delta(s, b, t)$

Then add the rule $A_{pq} \to aA_{rs}b$

For every $p, q, r \in Q$,

add the rule $A_{pq} \to A_{pr}A_{rq}$

For every $p \in Q$,

add the rule $A_{pp} \to \varepsilon$

THE CHOMSKY NORMAL FORM

A context-free grammar is in Chomsky normal form if every rule is of the form:

$A \rightarrow BC$	B, C are variables (not the start var)
$A \rightarrow a$	a is a terminal
$S \rightarrow \varepsilon$	S is the start variable

$S \rightarrow 0S1$

$S \rightarrow TT$

$T \rightarrow \varepsilon$

$S_0 \rightarrow TU \mid \varepsilon$

$T \rightarrow 0$

$U \rightarrow SV \mid 1$

$S \rightarrow TU$

$V \rightarrow 1$

Theorem: If G is in CNF, $w \in L(G)$ and $|w| > 0$, then any derivation of w in G has length $2|w| - 1$

Theorem: Any context-free language can be generated by a context-free grammar in Chomsky normal form

"Can transform any CFG into Chomsky normal form"

Theorem: **Any CFL can be generated by a CFG in Chomsky normal form**

Algorithm:

1. Add a new start variable (**$S_0 \rightarrow S$**)
2. Eliminate all **$A \rightarrow \varepsilon$** rules:

   For each occurrence of A on the RHS of a rule, add a new rule that removes that occurrence *(unless this new rule was previously removed)*
3. Eliminate all **$A \rightarrow B$** rules:

   For each rule with B on LHS of a rule, add a new rule that puts A on the LHS instead *(unless this new rule was previously removed)*
4. Convert **$A \rightarrow u_1 u_2 \ldots u_k$** to **$A \rightarrow u_1 A_1, A_1 \rightarrow u_2 A_2, \ldots$**

If **u_i** is a terminal, replace **u_i** with **U_i** and add **$U_i \rightarrow u_i$**

Convert the following into Chomsky normal form:

$$A \to BAB \mid B \mid \varepsilon$$
$$B \to 00 \mid \varepsilon$$

$$S_0 \to A$$
$$A \to BAB \mid B \mid \varepsilon$$
$$B \to 00 \mid \varepsilon$$

$$S_0 \to A \mid \varepsilon$$
$$A \to BAB \mid B \mid BB \mid AB \mid BA$$
$$B \to 00$$

$$S_0 \to BAB \mid 00 \mid BB \mid AB \mid BA \mid \varepsilon$$
$$A \to BAB \mid 00 \mid BB \mid AB \mid BA$$
$$B \to 00$$

$$S_0 \to BC \mid DD \mid BB \mid AB \mid BA \mid \varepsilon, \quad C \to AB,$$
$$A \to BC \mid DD \mid BB \mid AB \mid BA, \quad B \to DD, \quad D \to 0$$

FORMAL DEFINITIONS

deterministic DFA

A ^ finite automaton ^ is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$

Q is the set of states (finite)

Σ is the alphabet (finite)

$\delta : Q \times \Sigma \rightarrow Q$ is the transition function

$q_0 \in Q$ is the start state

$F \subseteq Q$ is the set of accept states

Let $w_1, \ldots, w_n \in \Sigma$ and $w = w_1 \ldots w_n \in \Sigma^*$

Then **M accepts w** if there are $r_0, r_1, \ldots, r_n \in Q$, s.t.

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

A *non-deterministic* finite automaton (NFA) is a 5-tuple $N = (Q, \Sigma, \delta, Q_0, F)$

Q is the set of states

Σ is the alphabet

$\delta : Q \times \Sigma_\varepsilon \rightarrow 2^Q$ is the transition function

$Q_0 \subseteq Q$ is the set of start states

$F \subseteq Q$ is the set of accept states

2^Q is the set of all possible subsets of Q

$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$

Let $w \in \Sigma^*$ and suppose w can be written as $w_1 \ldots w_n$ where $w_i \in \Sigma_\varepsilon$ (ε = empty string)

Then N accepts w if there are $r_0, r_1, \ldots, r_n \in Q$ such that

1. $r_0 \in Q_0$
2. $r_{i+1} \in \delta(r_i, w_{i+1})$ for $i = 0, \ldots, n-1$, and
3. $r_n \in F$

L(N) = the language recognized by N
= set of all strings machine N accepts

A language L is recognized by an NFA N if L = L(N).

Definition: A (non-deterministic) PDA is a tuple P = (Q, Σ, Γ, δ, q_0, F), where:

Q is a finite set of states

Σ is the input alphabet

Γ is the stack alphabet

$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow 2^{Q \times \Gamma_\varepsilon}$

$q_0 \in Q$ is the start state

$F \subseteq Q$ is the set of accept states

2^Q is the set of subsets of Q and $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$

Let $w \in \Sigma^*$ and suppose w can be written as $w_1 \ldots w_n$ where $w_i \in \Sigma_\varepsilon$ (recall $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$)

Then P accepts w if there are

$r_0, r_1, \ldots, r_n \in Q$ and

$s_0, s_1, \ldots, s_n \in \Gamma^*$ (sequence of stacks) **such that**

1. **$r_0 = q_0$ and $s_0 = \varepsilon$ (P** starts in q_0 with empty stack)

2. **For i = 0, ..., n-1:**
$(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where **$s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$**
(**P** moves correctly according to state, stack and symbol read)

3. **$r_n \in F$** (**P** is in an accept state at the end of its input)

CONTEXT-FREE GRAMMARS

A context-free grammar (CFG) is a tuple G = (V, Σ, R, S), where:

V is a finite set of variables

Σ is a finite set of terminals (disjoint from V)

R is set of production rules of the form $A \rightarrow W$, where $A \in V$ and $W \in (V \cup \Sigma)^*$

$S \in V$ is the start variable

$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$ Strings Generated by G

Definition: A Turing Machine is a 7-tuple
$T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where:

Q is a finite set of states

Σ is the input alphabet, where $\square \notin \Sigma$

Γ is the tape alphabet, where $\square \in \Gamma$ and $\Sigma \subseteq \Gamma$

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

$q_0 \in Q$ is the start state

$q_{accept} \in Q$ is the accept state

$q_{reject} \in Q$ is the reject state, and $q_{reject} \neq q_{accept}$

A Turing Machine M accepts input w if there is a sequence of configurations $C_1, \ldots, C_k$ such that

1. C_1 is a *start* configuration of M on input w, ie C_1 is q_0w
2. each C_i *yields* C_{i+1}, ie M can legally go from C_i to C_{i+1} in a single step
3. C_k is an *accepting* configuration, ie the state of the configuration is q_{accept}

Accepting and *rejecting* configurations are *halting* configurations and do not yield further configurations

TMs VERSUS FINITE AUTOMATA

TM can both *write* to and read from the tape

The head can move *left and right*

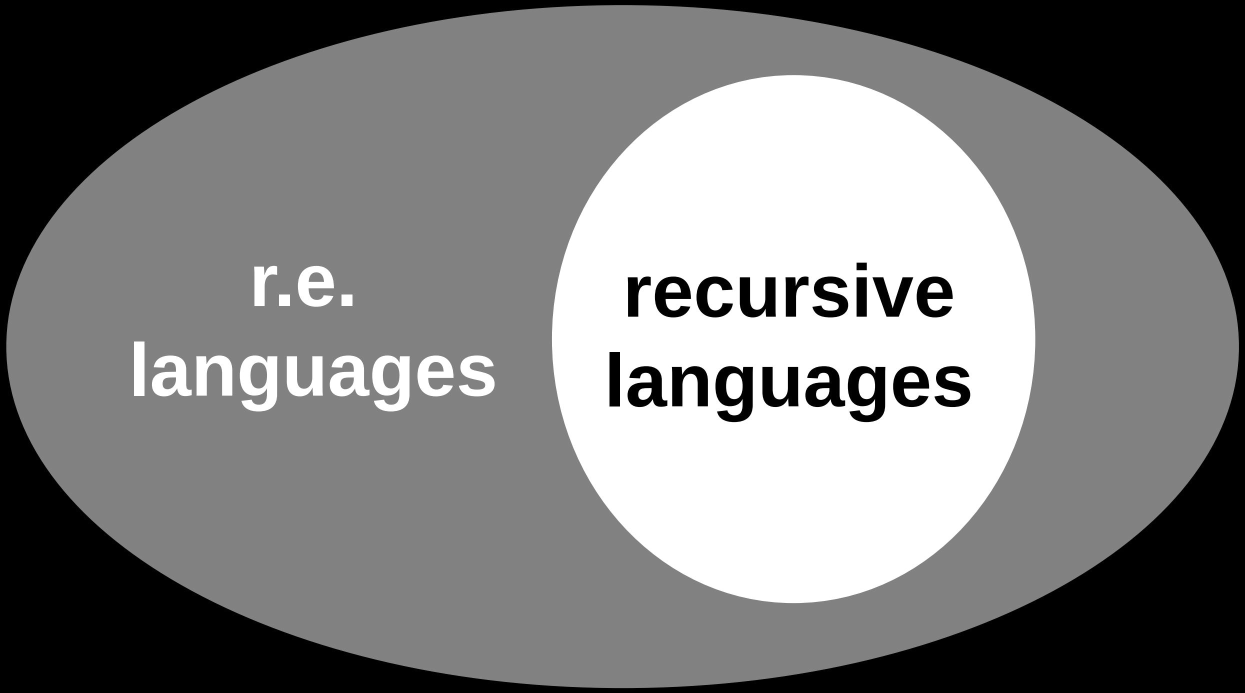
The input doesn't have to be read entirely,

and the computation can continue after all the input has been read

Accept and Reject take immediate effect

A language is called **Turing-recognizable** or **recursively enumerable (r.e.)** or **semi-decidable** if some TM **recognizes** it

A language is called **decidable** or **recursive** if some TM **decides** it



Theorem: If A and ¬A are r.e. then A is recursive

A_{DFA} = { (B, w) | B is a DFA that accepts string w }

Theorem: A_{DFA} is decidable

Proof Idea: Simulate B on w

A_{NFA} = { (B, w) | B is an NFA that accepts string w }

Theorem: A_{NFA} is decidable

A_{CFG} = { (G, w) | G is a CFG that generates string w }

Theorem: A_{CFG} is decidable

Proof Idea: Transform G into Chomsky Normal Form. Try all derivations of length up to 2|w|-1