

**SIDDARTHA INSTITUTE OF SCIENCE AND
TECHNOLOGY
(AUTONOMOUS)**

**DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING**



**FORMAL LANGUAGES AND AUTOMATA THEORY
(20CS0512)**

COURSE OBJECTIVES

The objectives of this course:

- *Explain formal definitions of machine models.*
- *Classify machines by their power to recognize languages and understanding of formal grammars, analysis.*
- *Illustrate hierarchical organization of problems depending on their complexity.*
- *Explain logical limits to computational capacity.*
- *Describe decidable and un-decidable problems.*

COURSE OUTCOMES (COs)

On successful completion of this course, students will be able to

- *Compare, understand and analyse different types of automata based on its functioning and acceptance of languages and grammars.*
- *Construct finite Automats for various problems.*
- *Design regular expressions and illustrate its identities, equivalence and closure properties.*
- *Understand the various properties of the CFG and can perform its simplification and normal forms.*
- *Define all the properties of Pushdown Automata and create it for various problems understanding the equivalence of PDA and CFGs.*
- *Infer and design solutions to the problems using Turing machines with ability to distinguish between computability, decidability and undecidability problems.*

Unit-1

Introduction: Basics of set theory, Relations on sets, Alphabet, Strings, languages and grammars, Chomsky hierarchy of languages.

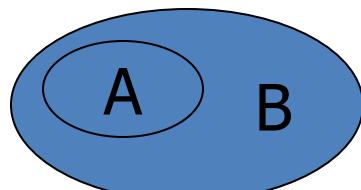
Finite Automata: History of Automata theory, Characteristics of Automata, Graphical notation of FA, DFA and NFA, Conversion of an NFA to DFA, NFA with e(null)Move, Equivalence of DFA and NFA, Finite Automata with Output, Conversion from Moore to Mealy and Mealy to Moore Machine, Minimization of Finite Automata, Myhill-Nerode Theorem, Applications and Limitations FA.

Basics of Set Theory

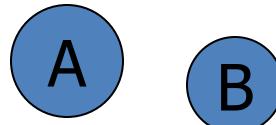
- $S=\{a, b, c\}$ refers to the set
whose elements are a, b and c.
- $a \in S$ means “a is an element of set S”.
- $d \notin S$ means “d is *not* an element of set S”.
- $\{x \in S \mid P(x)\}$ is the set of all those x from S such
that $P(x)$ is true. *E.g.*, $T=\{x \in \mathbf{Z} \mid 0 < x < 10\}$.

Relations between sets

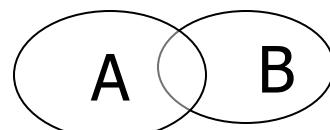
- **Definition:** Suppose A and B are sets. Then
A is called a **subset** of B: $A \subseteq B$
iff every element of A is also an element of B.
Symbolically,
 $A \subseteq B \Leftrightarrow \forall x, \text{ if } x \in A \text{ then } x \in B.$
- $A \not\subseteq B \Leftrightarrow \exists x \text{ such that } x \in A \text{ and } x \notin B.$



$$A \subseteq B$$



$$A \not\subseteq B$$



$$A \not\subseteq B$$

Relations between sets

- **Definition:** Suppose A and B are sets. Then

A **equals** B: $A = B$

iff every element of A is in B and
every element of B is in A.

Symbolically,

$$A=B \Leftrightarrow A \subseteq B \text{ and } B \subseteq A .$$

- **Example:** Let $A = \{m \in \mathbf{Z} \mid m=2k+3 \text{ for some integer } k\}$;
 $B =$ the set of all odd integers.
Then $A=B$.

Operations on Sets

Definition: Let A and B be subsets of a set U.

1. Union of A and B: $A \cup B = \{x \in U \mid x \in A \text{ or } x \in B\}$

2. Intersection of A and B:

$$A \cap B = \{x \in U \mid x \in A \text{ and } x \in B\}$$

3. Difference of B minus A: $B - A = \{x \in U \mid x \in B \text{ and } x \notin A\}$

4. Complement of A: $A^c = \{x \in U \mid x \notin A\}$

Ex.: Let $U = \mathbf{R}$, $A = \{x \in \mathbf{R} \mid 3 < x < 5\}$, $B = \{x \in \mathbf{R} \mid 4 < x < 9\}$.

Then

$$1) A \cup B = \{x \in \mathbf{R} \mid 3 < x < 9\}.$$

$$2) A \cap B = \{x \in \mathbf{R} \mid 4 < x < 5\}.$$

$$3) B - A = \{x \in \mathbf{R} \mid 5 \leq x < 9\}, A - B = \{x \in \mathbf{R} \mid 3 < x \leq 4\}.$$

$$4) A^c = \{x \in \mathbf{R} \mid x \leq 3 \text{ or } x \geq 5\}, B^c = \{x \in \mathbf{R} \mid x \leq 4 \text{ or } x \geq 9\}$$

Properties of Sets

➤ **Theorem 1** (*Some subset relations*):

- 1) $A \cap B \subseteq A$
- 2) $A \subseteq A \cup B$
- 3) If $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$.

➤ To prove that $A \subseteq B$ use the “element argument”:

1. suppose that x is a particular but arbitrarily chosen element of A ,
2. show that x is an element of B .

Proving a Set Property

- **Theorem 2 (*Distributive Law*):**

For any sets A,B and C:

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C).$$

- **Proof:** We need to show that

$$(I) A \cup (B \cap C) \subseteq (A \cup B) \cap (A \cup C) \text{ and}$$

$$(II) (A \cup B) \cap (A \cup C) \subseteq A \cup (B \cap C).$$

Let's show (I).

Suppose $x \in A \cup (B \cap C)$ (1)

We want to show that $x \in (A \cup B) \cap (A \cup C)$ (2)

Proving a Set Property

- **Proof (cont.):**

$$x \in A \cup (B \cap C) \Rightarrow x \in A \text{ or } x \in B \cap C.$$

(a) Let $x \in A$. Then

$$x \in A \cup B \text{ and } x \in A \cup C \Rightarrow x \in (A \cup B) \cap (A \cup C)$$

(b) Let $x \in B \cap C$. Then $x \in B$ and $x \in C$.

$$\left. \begin{array}{l} x \in B \Rightarrow x \in A \cup B \\ x \in C \Rightarrow x \in A \cup C \end{array} \right\} \Rightarrow x \in (A \cup B) \cap (A \cup C)$$

Thus, (2) is true, and we have shown (I).

(II) is shown similarly (*left as exercise*). ■

Set Properties

- Commutative Laws:
 - (a) $A \cap B = B \cap A$
 - (b) $A \cup B = B \cup A$
- Associative Laws:
 - (a) $(A \cap B) \cap C = A \cap (B \cap C)$
 - (b) $(A \cup B) \cup C = A \cup (B \cup C)$
- Distributive Laws:
 - (a) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
 - (b) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Set Properties

- Double Complement Law:

$$(A^c)^c = A$$

- De Morgan's Laws:

$$(a) (A \cap B)^c = A^c \cup B^c$$

$$(b) (A \cup B)^c = A^c \cap B^c$$

- Absorption Laws:

$$(a) A \cup (A \cap B) = A$$

$$(b) A \cap (A \cup B) = A$$

Showing that an alleged set property is false

- **Statement:** For all sets A,B and C,

$$A - (B - C) = (A - B) - C .$$

The following counterexample
shows that the statement is false.

- **Counterexample:**

Let $A=\{1,2,3,4\}$, $B=\{3,4,5,6\}$, $C=\{3\}$.

Then $B - C = \{4,5,6\}$ and $A - (B - C) = \{1,2,3\}$.

On the other hand,

$A - B = \{1,2\}$ and $(A - B) - C = \{1,2\}$.

Thus, for this example

$$A - (B - C) \neq (A - B) - C .$$

Empty Set

- The unique set with no elements is called **empty set** and denoted by \emptyset .
- Set Properties that involve \emptyset .

For all sets A ,

1. $\emptyset \subseteq A$
2. $A \cup \emptyset = A$
3. $A \cap \emptyset = \emptyset$
4. $A \cap A^c = \emptyset$

Disjoint Sets

- A and B are called **disjoint** iff $A \cap B = \emptyset$.
- Sets A_1, A_2, \dots, A_n are called **mutually disjoint** iff for all $i, j = 1, 2, \dots, n$
$$A_i \cap A_j = \emptyset \text{ whenever } i \neq j.$$
- Examples:
 - 1) $A=\{1,2\}$ and $B=\{3,4\}$ are disjoint.
 - 2) The sets of even and odd integers are disjoint.
 - 3) $A=\{1,4\}$, $B=\{2,5\}$, $C=\{3\}$ are mutually disjoint.
 - 4) $A-B$, $B-A$ and $A \cap B$ are mutually disjoint.

Partitions

- **Definition:** A collection of nonempty sets $\{A_1, A_2, \dots, A_n\}$ is a **partition** of a set A iff
 1. $A = A_1 \cup A_2 \cup \dots \cup A_n$
 2. A_1, A_2, \dots, A_n are mutually disjoint.
- *Examples:*
 - 1) $\{\mathbf{Z}^+, \mathbf{Z}^-, \{0\}\}$ is a partition of \mathbf{Z} .
 - 2) Let $S_0 = \{n \in \mathbf{Z} \mid n=3k \text{ for some integer } k\}$
 $S_1 = \{n \in \mathbf{Z} \mid n=3k+1 \text{ for some integer } k\}$
 $S_2 = \{n \in \mathbf{Z} \mid n=3k+2 \text{ for some integer } k\}$
Then $\{S_0, S_1, S_2\}$ is a partition of \mathbf{Z} .

Power Sets

- **Definition:** Given a set A,
the **power set** of A, denoted $P(A)$,
is the set of all subsets of A.
- *Example:* $P(\{a,b\}) = \{\emptyset, \{a\}, \{b\}, \{a,b\}\}$.
- **Properties:**
 - 1) If $A \subseteq B$ then $P(A) \subseteq P(B)$.
 - 2) If a set A has n elements
then $P(A)$ has 2^n elements.

Relations *on* a Set

- A relation *on* a set A is a relation from A to A .
- Examples of relations *on* \mathbf{R} :
 - $R_1 = \{ (a, b) \mid a \leq b \}.$
 - $R_2 = \{ (a, b) \mid b = +\sqrt{a} \}.$
 - Are R_1 & R_2 functions?

Properties of Relations

A relation R on A is:

- *Reflexive*: $\forall a (aRa)$.

Are either R_1 or R_2 reflexive?

- *Symmetric*: $\forall a \forall b (aRb \rightarrow bRa)$.

– Let S be a set of people.

– Let R & T be relations on S ,

$$R = \{ (a, b) \mid a \text{ is a sibling of } b \}.$$

$$T = \{ (a, b) \mid a \text{ is a brother of } b \}.$$

Is R symmetric?

Is T symmetric?

(Cont.,)

- *Antisymmetric:*

$$1. \forall a \forall b ((aRb \wedge bRa) \rightarrow (a = b)).$$

$$2. \forall a \forall b ((a \neq b) \rightarrow ((a, b) \notin R \vee (b, a) \notin R)).$$

Example: $L = \{ (a, b) \mid a \leq b \}.$

Can a relation be symmetric & antisymmetric?

- *Transitive:*

$$\forall a \forall b \forall c ((aRb \wedge bRc) \rightarrow aRc).$$

Are any of the previous examples transitive?

Composition

- Let R be a relation from A to B .
- Let S be a relation from B to C .
- The *composition* is

$$S \circ R = \{ (a, c) \mid \exists b (aRb \wedge bSc) \}.$$

- Let R be a relation *on* A .

$$R^1 = R$$

$$R^n = R^{n-1} \circ R.$$

- Let $R = \{ (1, 1), (2, 1), (3, 2), (4, 3) \}$.

What is R^2, R^3 ?

Introduction to Formal Proof

- In this class, sometimes we will give formal proofs and at other times intuitive “proofs”
- Mostly inductive proofs
- First, a bit about deductive proofs

Deductive Proofs

- Given a hypothesis H, and some statements, generate a conclusion C
- Sherlock Holmes style of reasoning
- Example: consider the following theorem
 - **If $x \geq 4$ then $2^x \geq x^2$**
 - Here, H is $x \geq 4$ and C is $2^x \geq x^2$
 - Intuitive deductive proof
 - Each time x increases by one, the left hand side doubles in size. However, the right side increases by the ratio $((x+1)/x)^2$. When $x=4$, this ratio is 1.56. As x increases and approaches infinity, the ratio $((x+1)/x)^2$ approaches 1. This means the ratio gets smaller as x increases. Consequently, 1.56 is the largest that the right hand side will increase. Since $1.56 < 2$, the left side is increasing faster than the right side

Basic Formal Logic (1)

- An “If H then C” statement is typically expressed as:
 $H \Rightarrow C$ or H implies C
- The logic truth table for implication is:

H	C	$H \Rightarrow C$ (i.e. $\neg H \vee C$)
F	F	T
F	T	T
T	F	F
T	T	T

Basic Formal Logic (2)

- If and Only If statements, e.g. “If and only if H then C” means that $H \Rightarrow C$ and $C \Rightarrow H$.
- Sometimes this will be written as
 $H \Leftrightarrow C$ or “H iff C”.

The truth table is:

H	C	$H \Leftrightarrow C$ (i.e. H equals C)
F	F	T
F	T	F
T	F	F
T	T	T

Modus Ponens

- **Modus Ponens** (Latin for “method of affirming”) can be used to form chains of logic to reach a desired conclusion.
- In other words, given:

$$H \Rightarrow C \quad \text{and}$$

$$H$$

Then we can infer C

- Example: given “If Joe and Sally are siblings then Joe and Sally are related” as a true assertion, and also given “Joe and Sally are siblings” as a true assertion, then we can conclude “Joe and Sally are related.”

Modus Tollens

- **Modus Tollens** (Latin for “method of denying”). This reasons backwards across the implication.
 - Cognitive psychologists have shown that under 60% of college students have a solid intuitive understanding of Modus Tollens versus almost 100% for Modus Ponens
- If we are given:

$$H \Rightarrow C \quad \text{and} \\ \neg C$$

then we can infer $\neg H$.

- For example, given: “If Joe and Sally are siblings then Joe and Sally are related” as a true assertion, and also given “Joe and Sally are not related” as a true assertion, then we can conclude “Joe and Sally are not siblings.”
 - What if we are told Joe and Sally are not siblings? Can we conclude anything?

Proof by Contradiction

- Suppose that we want to prove H and we know that C is true. Instead of proving H directly, we may instead show that assuming $\neg H$ leads to a contradiction. Therefore H must be true.
- Example:
 - A large sum of money has been stolen from the bank. The criminal(s) were seen driving away from the scene. From questioning criminals A, B, and C we know:
 - No one other than A, B, or C were involved in the robbery.
 - C never pulls a job without A
 - B does not know how to drive

Proof by Contrapositive

- Proof by contrapositive takes advantage of the logical equivalence between "H implies C" and "Not C implies Not H".
- For example, the assertion "If it is my car, then it is red" is equivalent to "If that car is not red, then it is not mine".
- To prove "If P, Then Q" by the method of contrapositive means to prove "If Not Q, Then Not P".

Contrapositive Example

- An integer x is called even (respectively odd) if there is another integer k for which $x = 2k$ (respectively $2k+1$).
- Two integers are said to have the same parity if they are both odd or both even.
- Theorem. If x and y are two integers for which $x+y$ is even, then x and y have the same parity

Contrapositive Example

- Proof of the theorem
 - The contrapositive version of this theorem is "If x and y are two integers with opposite parity, then their sum must be odd."
 - Assume x and y have opposite parity.
 - Since one of these integers is even and the other odd, there is no loss of generality to suppose x is even and y is odd.
 - Thus, there are integers k and m for which $x = 2k$ and $y = 2m+1$. Then, we compute the sum $x+y = 2k + 2m + 1 = 2(k+m) + 1$, which is an odd integer by definition.

Contrapositive vs. Contradiction

- Both methods somewhat similar, but different.
- In contrapositive, we assume $\neg C$ and prove $\neg H$, given $H \Rightarrow C$.
 - The method of Contrapositive has the advantage that your goal is clear: Prove Not H.
- In the method of Contradiction, your goal is to prove a contradiction, but it is not always clear what the contradiction is going to be at the start.
 - Indeed, one may never be found (and will never be found if the hypothesis is false).

Proof by Induction

- Essential for proving recursively defined objects
- We can perform induction on integers, automata, and concepts like trees or graphs.
- To make an inductive proof about a statement $S(X)$ we need to prove two things:
 1. Basis: Prove for one or several small values of X directly.
 2. Inductive step: Assume $S(Y)$ for Y "smaller than" X ; then prove $S(X)$ using that assumption.

Familiar Induction Example?

- For all $n \geq 0$, prove that:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

- First prove the basis. We pick $n=0$. When $n=0$, there is a general principle that when the upper limit (0) of a sum is less than the lower limit (1) then the sum is over no terms and therefore the sum is 0. That is:

$$\sum_{i=1}^0 i = 0$$

Familiar Induction Example

- Next prove the induction. Assume $n \geq 0$. We must prove that the theorem implies the same formula when n is larger. For integers, we will use $n+1$ as the next largest value. This means that the formula should hold with $n + 1$ substituted for n :

$$\begin{aligned}\sum_{i=1}^{n+1} i &= \frac{(n+1)(n+2)}{2} \\ &= \frac{n^2 + 3n + 2}{2}\end{aligned}$$

- This should equal what we came up with previously if we just add on an extra $n+1$ term:

$$\sum_{i=1}^{n+1} i = \left(\sum_{i=1}^n i \right) + (n+1)$$

Familiar Induction Example

- Continued:
$$\sum_{i=1}^{n+1} i = \left(\sum_{i=1}^n i \right) + (n+1)$$

$$\begin{aligned}& \left(\sum_{i=1}^n i \right) + (n+1) \\&= \frac{n(n+1)}{2} + (n+1) \\&= \frac{n^2 + n}{2} + \frac{2n+2}{2} \\&= \frac{n^2 + 3n + 2}{2}\end{aligned}$$

This matches what we got from the inductive step, and the proof is complete.

Second Induction Example

- If $x \geq 4$ then $2^x \geq x^2$
- Basis: If $x=4$, then 2^x is 16 and x^2 is 16. Thus, the theorem holds.
- Induction: Suppose for some $x \geq 4$ that $2^x \geq x^2$. With this statement as the hypothesis, we need to prove the same statement, with $x+1$ in place of x : $2^{(x+1)} \geq (x+1)^2$

Second Induction Example

- $2^{(x+1)} \geq (x+1)^2$? (i)
- Rewrite in terms of $S(x)$
 - $2^{(x+1)} = 2 * 2^x$
 - We are assuming $2^x \geq x^2$
 - So therefore $2^{(x+1)} = 2 * 2^x \geq 2x^2$ (ii)
- Substitute (ii) into (i)
 - $2x^2 \geq (x+1)^2$
 - $2x^2 \geq (x^2 + 2x + 1)$
 - $x^2 \geq 2x + 1$
 - $x \geq 2 + 1/x$
 - Since $x \geq 4$, we get some value ≥ 4 on the left side. The right side will equal at most 2.25 and in fact gets smaller and approaches 2 as x increases. Consequently, we have proven the theorem to be true by induction.

Alphabets

- An *alphabet* is any finite set of symbols.
- Examples: ASCII, Unicode, $\{0,1\}$ (*binary alphabet*), $\{a,b,c\}$.

Strings

- The set of *strings* over an alphabet Σ is the set of lists, each element of which is a member of Σ .
 - Strings shown with no commas, e.g., abc.
- Σ^* denotes this set of strings.
- ϵ stands for the *empty string* (string of length 0).

Example: Strings

- $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$
- Substring: 0 as a string, 0 as a symbol look the same.
 - Context determines the type.

Languages

- A *language* is a subset of Σ^* for some alphabet Σ .
- Example: The set of strings of 0's and 1's with no two consecutive 1's.
- $L = \{\epsilon, 0, 1, 00, 01, 10, 000, 001, 010, 100, 101, 0000, 0001, 0010, 0100, 0101, 1000, 1001, 1010, \dots\}$

Grammar Formalism

- A Grammar:
 - A set of production rules.
 - In addition to the vocabulary, the production rules can use other symbols
 - N (noun)
 - V (verb)
 - NP (noun phrase)
 - VP (verb phrase)
 - One symbol is special:
 - S (sentence)

What is a grammar formalism?

Production Rules

- $S \rightarrow NP\ VP$
 - $NP \rightarrow Det\ N$
 - $VP \rightarrow V\ NP$
 - $DET \rightarrow \text{the}$
 - $DET \rightarrow a$
 - $N \rightarrow \text{boy}$
 - $N \rightarrow \text{girl}$
 - $V \rightarrow \text{saw}$
 - $V \rightarrow \text{sees}$
- These production rules have a non-terminal symbol (one that isn't from the vocabulary) on the left, then an arrow, then some terminal (from the vocabulary) and non-terminal symbols on the right.
 - This is one instance of a grammar formalism.
 - We will see that other grammar formalisms use other types of symbols and production rules.

Grammar Formalism- Derivation

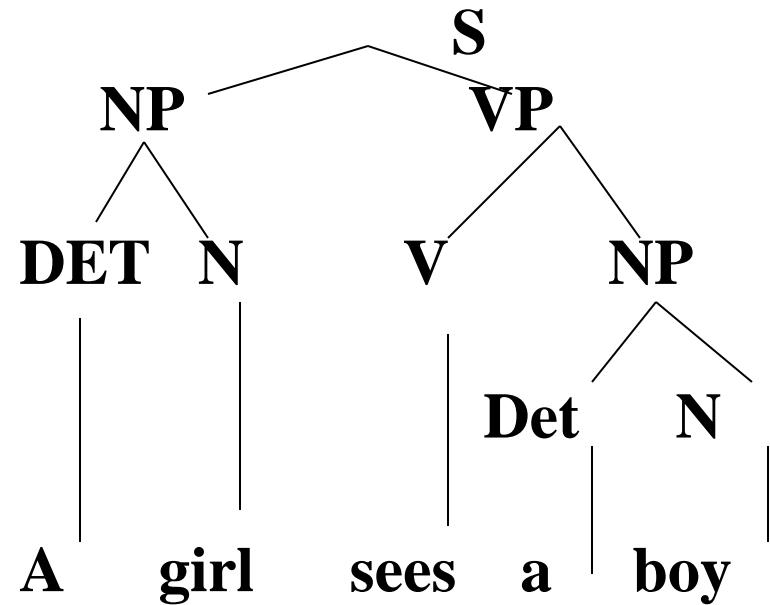
- The production rules are interpreted as instructions:
 - Parsing: when you find the string on the right hand side, replace it with the string on the left hand side.
 - Generation: when you find the symbol on the left hand side, replace it with the string on the right hand side.
 - Different grammar formalisms will have different instructions.

Grammar Formalism

- A derivation is the ordered list of production rules that you use to get from the special symbol to the terminal string or vice versa.
 - S
 - NP VP
 - Det N VP
 - Det N V NP
 - Det N V Det N
 - The N V Det N
 - The girl V Det N
 - The girl sees Det N
 - The girl sees a N
 - The girl sees a boy
- $S \rightarrow NP\ VP$
 $NP \rightarrow Det\ N$
 $VP \rightarrow V\ NP$
 $DET \rightarrow the$
 $DET \rightarrow a$
 $N \rightarrow boy$
 $N \rightarrow girl$
 $V \rightarrow saw$
 $V \rightarrow sees$

Structure of GF

- S
- NP VP
- Det N VP
- Det N V NP
- Det N V Det N
- The N V Det N
- The girl V Det N
- The girl sees Det N
- The girl sees a N
- The girl sees a boy



The Chomsky Hierarchy

Non Turing-Acceptable

Turing-Acceptable

decidable

Context-sensitive

Context-free

Regular

Finite Automata – DFA & NFA

Finite Automata

- Two types – both describe what are called **regular languages**
 - Deterministic (DFA) – There is a fixed number of states and we can only be in one state at a time
 - Nondeterministic (NFA) –There is a fixed number of states but we can be in multiple states at one time
- While NFA's are more expressive than DFA's, we will see that adding nondeterminism does not let us define any language that cannot be defined by a DFA.
- One way to think of this is we might write a program using a NFA, but then when it is “compiled” we turn the NFA into an equivalent DFA.

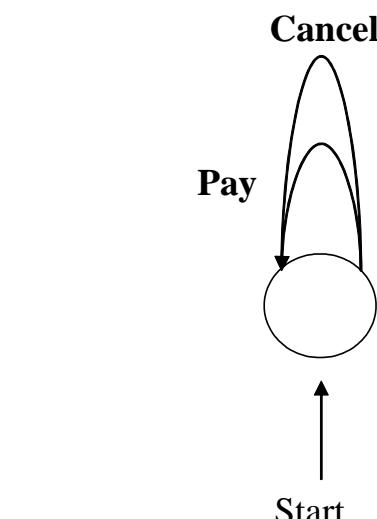
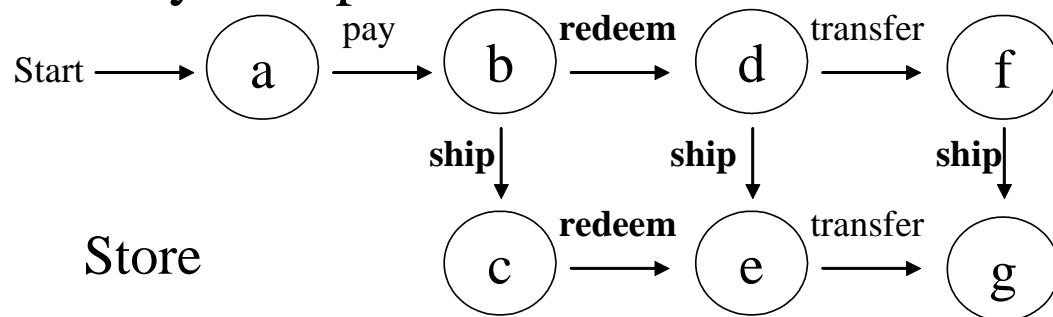
Informal Example

- Customer shopping at a store with an electronic transaction with the bank
 - The customer may *pay* the e-money or *cancel* the e-money at any time.
 - The store may *ship* goods and *redeem* the electronic money with the bank.
 - The bank may *transfer* any redeemed money to a different party, say the store.
- Can model this problem with three automata

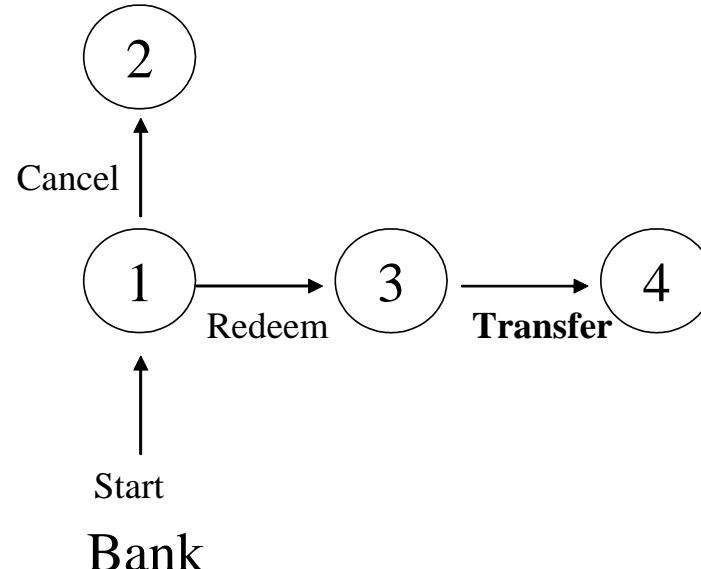
Bank Automata

Actions in bold are initiated by the entity.

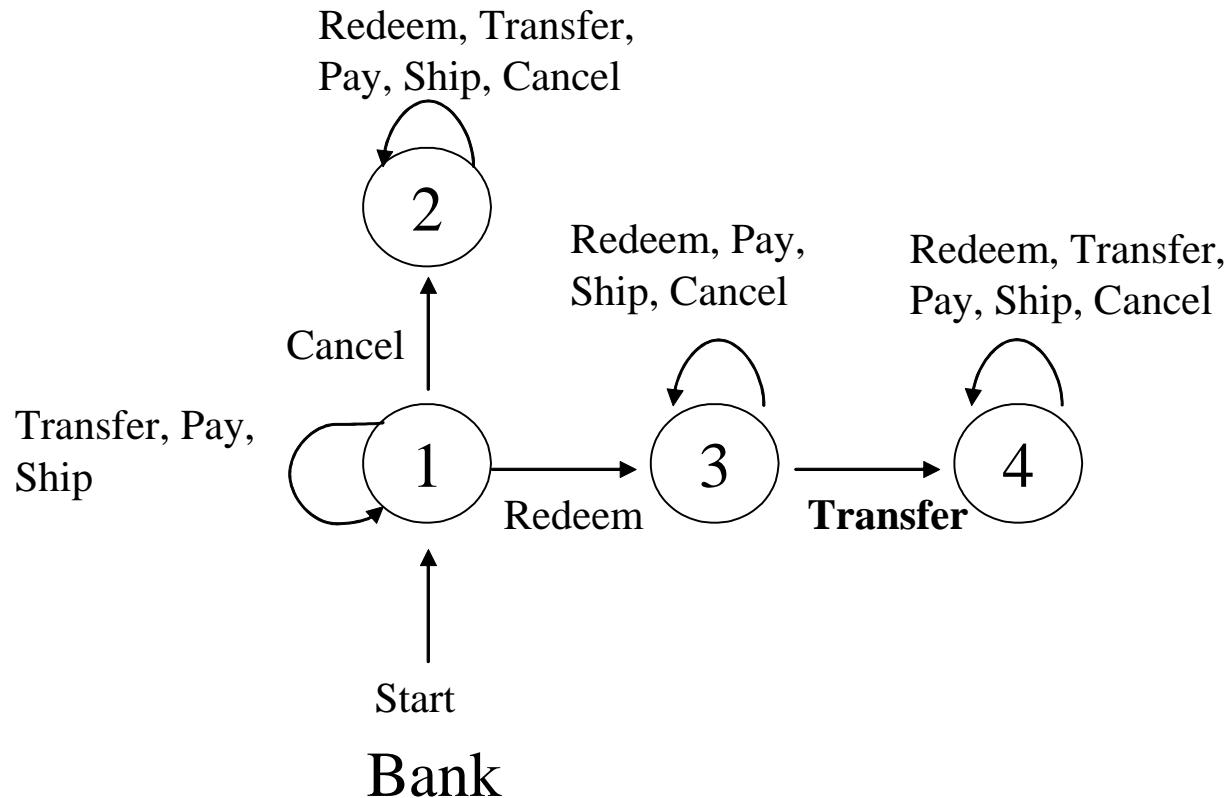
Otherwise, the actions are initiated by someone else and received by the specified automata



Customer



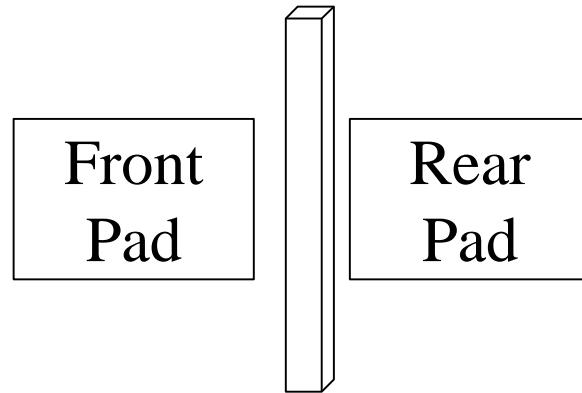
Complete Bank Automaton



Ignores other actions that may be received

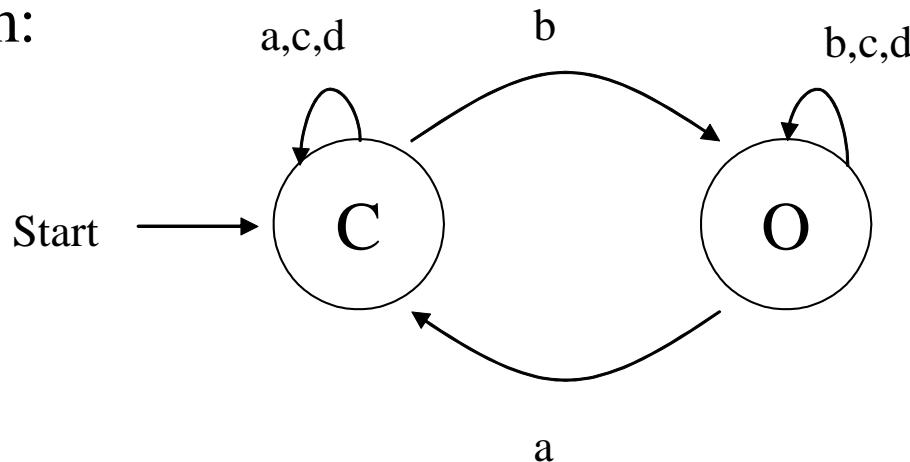
Simple Example – 1 way door

- As an example, consider a one-way automatic door. This door has two pads that can sense when someone is standing on them, a front and rear pad. We want people to walk through the front and toward the rear, but not allow someone to walk the other direction:



One Way Door

- Let's assign the following codes to our different input cases:
 - Nobody on either pad
 - Person on front pad
 - Person on rear pad
 - Person on front and rear pad
- We can design the following automaton so that the door doesn't open if someone is still on the rear pad and hit them:



Formal Definition of a Finite Automaton

1. Finite set of states, typically Q .
2. Alphabet of input symbols, typically Σ
3. One state is the start/initial state, typically q_0 // $q_0 \in Q$
4. Zero or more final/accepting states; the set is typically F . // $F \subseteq Q$
5. A transition function, typically δ . This function
 - Takes a state and input symbol as arguments.
 - Returns a state.
 - One “rule” would be written $\delta(q, a) = p$, where q and p are states, and a is an input symbol.
 - Intuitively: if the FA is in state q , and input a is received, then the FA goes to state p (note: $q = p$ OK).
6. A FA is represented as the five-tuple: $A = (Q, \Sigma, \delta, q_0, F)$.
Here, F is a set of accepting states.

Formal Definition of DFA

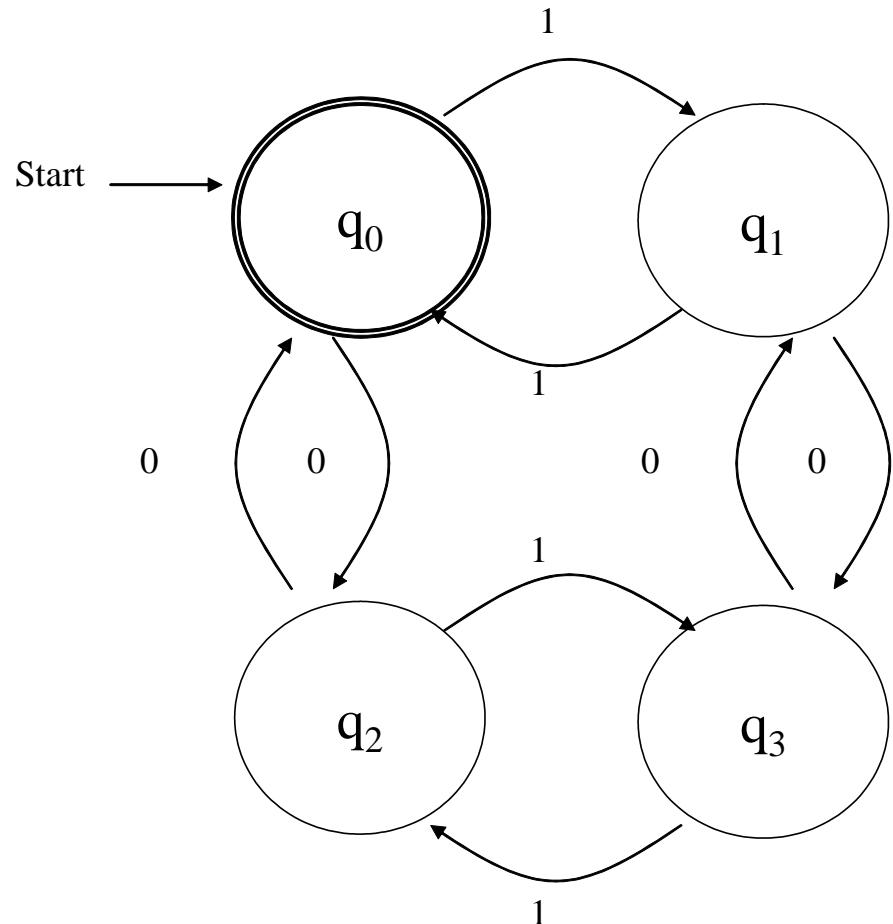
- Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1w_2\dots w_n$ be a string where each w_i is a member of alphabet Σ .
- **M accepts w** if a sequence of states $r_0r_1\dots r_n$ in Q exists with three conditions:
 1. $r_0 = q_0$
 2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i=0, \dots, n-1$
 3. $r_n \in F$

We say that **M recognizes language A** if $A = \{ w \mid M \text{ accepts } w \}$

In other words, the language is all of those strings that are accepted by the finite automata.

DFA

- Here is a DFA for the language that is the set of all strings of 0's and 1's whose numbers of 0's and 1's are both even:

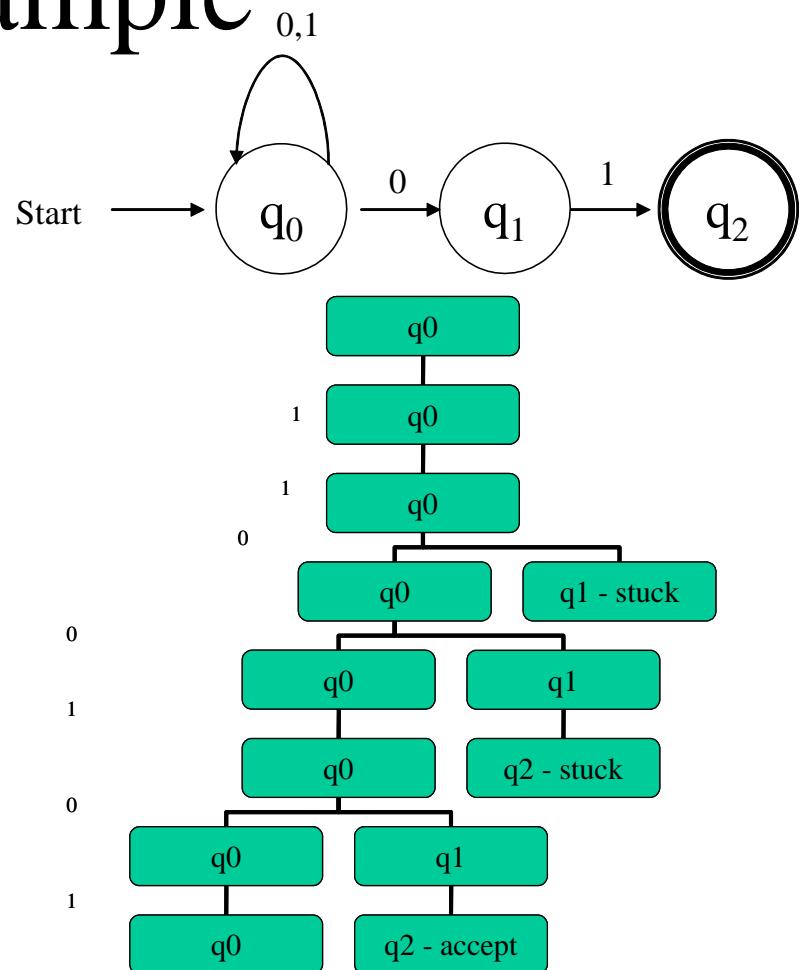


Nondeterministic Finite Automata

- A NFA (nondeterministic finite automata) is able to be in several states at once.
 - In a DFA, we can only take a transition to a single deterministic state
 - In a NFA we can accept multiple destination states for the same input.
- NFA is a useful tool
 - More expressive than a DFA.
 - BUT we will see that it is not more powerful! For any NFA we can construct a corresponding DFA
 - Another way to think of this is the DFA is how an NFA would actually be implemented (e.g. on a traditional computer)

NFA Example

- This NFA accepts only those strings that end in 01
- Running in “parallel threads” for string 1100101



Formal Definition of an NFA

- Similar to a DFA
1. Finite set of states, typically Q .
 2. Alphabet of input symbols, typically Σ
 3. One state is the start/initial state, typically q_0
 4. Zero or more final/accepting states; the set is typically F .
 5. A transition function, typically δ . This function:
 - Takes a state and input symbol as arguments.
 - Returns a **set of states** instead of a single state, as a DFA
 6. A FA is represented as the five-tuple: $A = (Q, \Sigma, \delta, q_0, F)$. Here, F is a set of accepting states.

Previous NFA in Formal Notation

The previous NFA could be specified formally as:

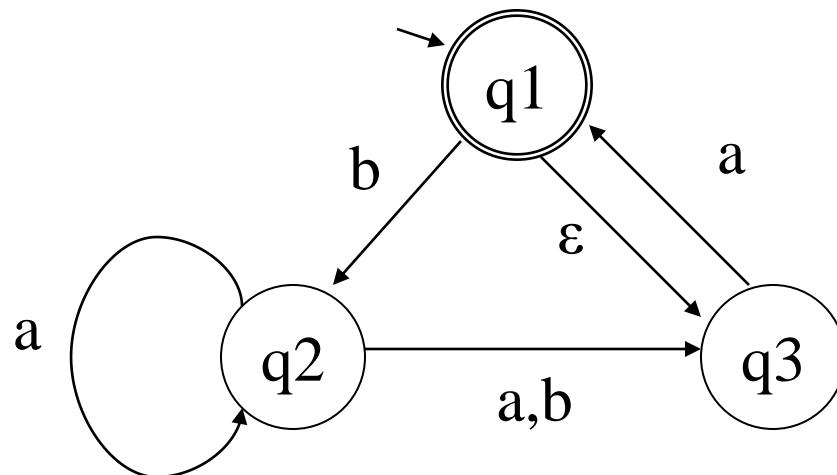
$$(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$$

The transition table is:

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

NFA Example

- Practice with the following NFA to satisfy yourself that it accepts ϵ , a, baba, baa, and aa, but that it doesn't accept b, bb, and babba.



Equivalence of DFA's and NFA's

- For most languages, NFA's are easier to construct than DFA's
- But it turns out we can build a corresponding DFA for any NFA
 - The downside is there may be up to 2^n states in turning a NFA into a DFA. However, for most problems the number of states is approximately equivalent.
- Theorem: A language L is accepted by some DFA if and only if L is accepted by some NFA; i.e. : $L(\text{DFA}) = L(\text{NFA})$ for an appropriately constructed DFA from an NFA.
 - Informal Proof: It is trivial to turn a DFA into an NFA (a DFA is already an NFA without nondeterminism). The following slides will show how to construct a DFA from an NFA.

NFA to DFA : Subset Construction

Let an NFA N be defined as $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$.

The equivalent DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ where:

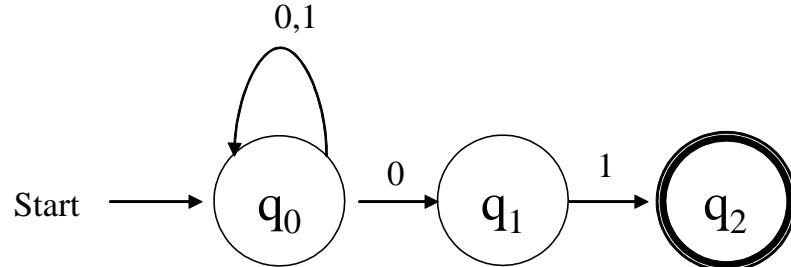
1. $Q_D = 2^{Q_N}$; i.e. Q_D is the set of all subsets of Q_N ; that is, it is the power set of Q_N . Often, not all of these states are accessible from the start state; these states may be “thrown away.”
2. F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \emptyset$. That is, F_D is all sets of N ’s states that include at least one accepting state of N .
3. For each set $S \subseteq Q_N$ and for each input symbol a in Σ :

$$\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$$

That is, to compute $\delta_D(S, a)$ we look at all the states p in S , see what states N goes to starting from p on input a , and take the union of all those states.

Subset Construction Example (1)

- Consider the NFA:



The power set of these states is: $\{ \emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\} \}$

New transition function with all of these states and go to the set of possible inputs:

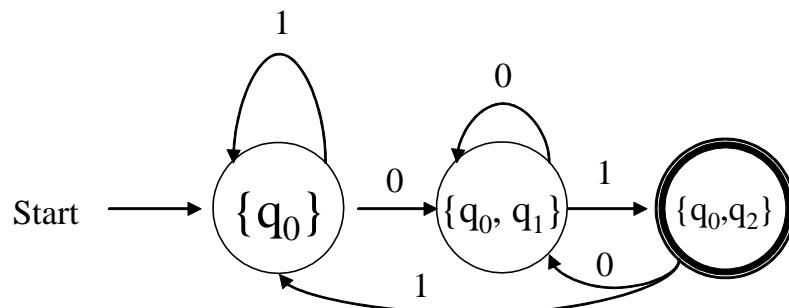
	0	1
\emptyset	\emptyset	\emptyset
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	\emptyset	$\{q_2\}$
$*\{q_2\}$	\emptyset	\emptyset
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	\emptyset	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0,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

Subset Construction (2)

- Many states may be unreachable from our start state. A good way to construct the equivalent DFA from an NFA is to start with the start states and construct new states on the fly as we reach them.

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

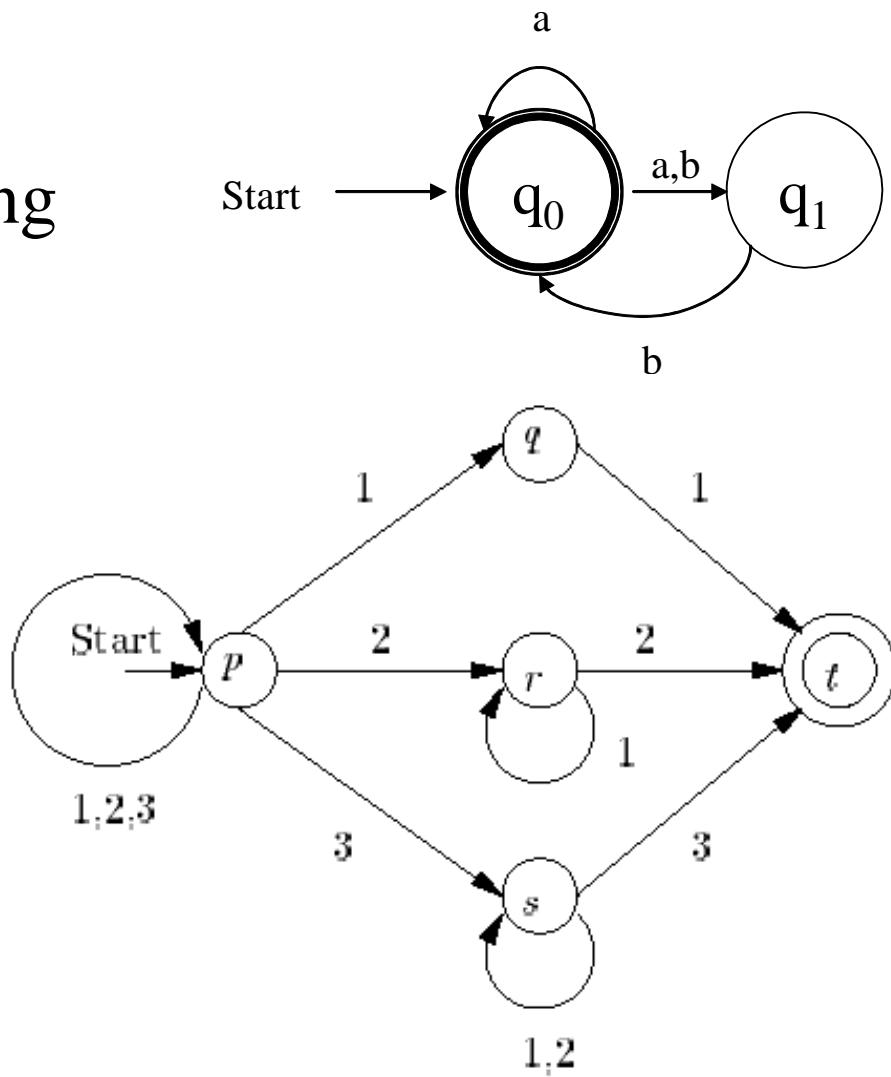
Graphically:



NFA to DFA Exercises

- Convert the following NFA's to DFA's

15 possible states on this second one (might be easier to represent in table format)



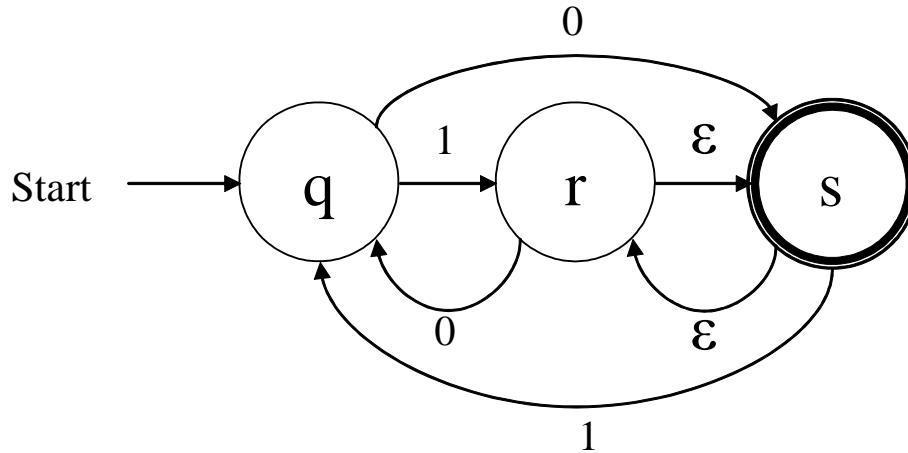
Epsilon Transitions

- Extension to NFA – a “feature” called epsilon transitions, denoted by ϵ , the empty string
- The ϵ transition lets us spontaneously take a transition, without receiving an input symbol
- Another mechanism that allows our NFA to be in multiple states at once.
 - Whenever we take an ϵ edge, we must fork off a new “thread” for the NFA starting in the destination state.
- While sometimes convenient, has no more power than a normal NFA
 - Just as a NFA has no more power than a DFA

Formal Notation – Epsilon Transition

- Transition function δ is now a function that takes as arguments:
 - A state in Q and
 - A member of $\Sigma \cup \{\varepsilon\}$; that is, an input symbol or the symbol ε . We require that ε not be a symbol of the alphabet Σ to avoid any confusion.

Epsilon NFA Example

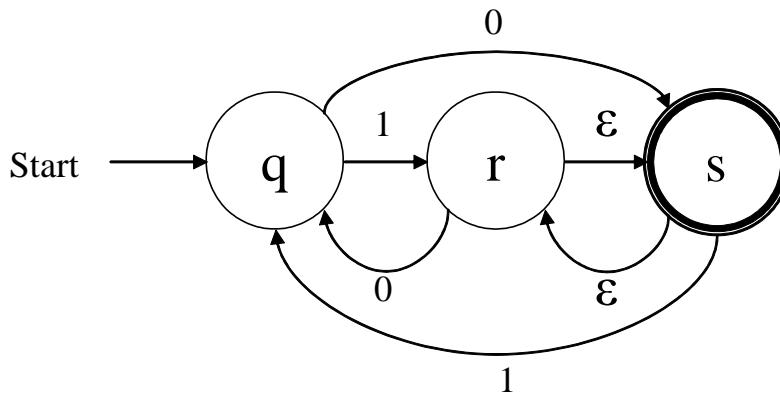


In this ϵ -NFA, the string “001” is accepted by the path qsrqrs, where the first qs matches 0, sr matches ϵ , rq matches 0, qr matches 1, and then rs matches ϵ . In other words, the accepted string is $0\epsilon 0\epsilon 1\epsilon$.

Epsilon Closure

- Epsilon closure of a state is simply the set of all states we can reach by following the transition function from the given state that are labeled ε .

Example:



- $\varepsilon\text{-closure}(q) = \{ q \}$
- $\varepsilon\text{-closure}(r) = \{ r, s \}$

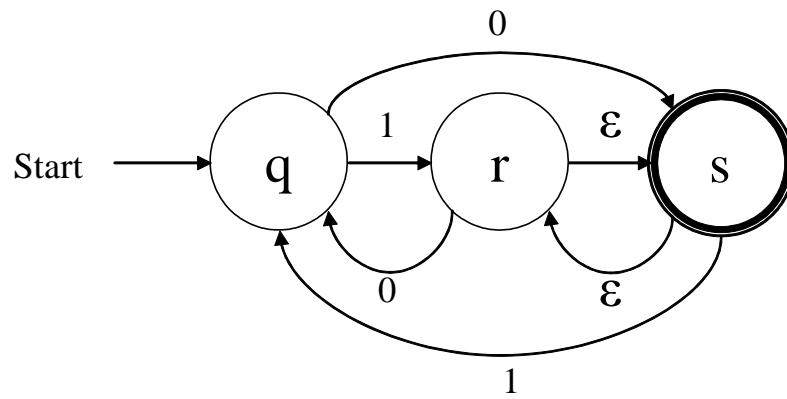
Eliminating Epsilon Transitions

To eliminate ϵ -transitions, use the following to convert to a DFA:

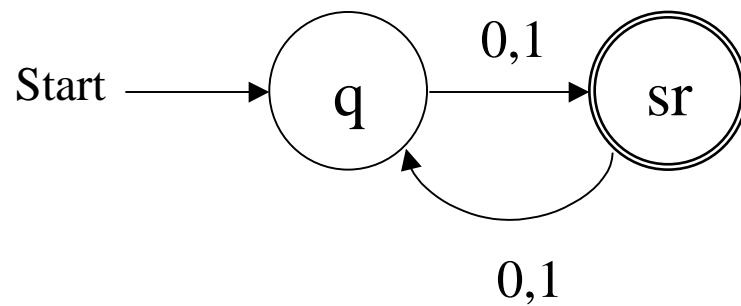
1. Compute ϵ -closure for the current state, resulting in a set of states S.
2. $\delta_D(S, a)$ is computed for all a in Σ by
 - a. Let $S = \{p_1, p_2, \dots, p_k\}$
 - b. Compute $\bigcup_{i=1}^k \delta(p_i, a)$ and call this set $\{r_1, r_2, r_3, \dots, r_m\}$ This set is achieved by following input a , not by following any ϵ -transitions
 - c. Add the ϵ -transitions in by computing $\delta(S, a) = \bigcup_{i=1}^m \epsilon\text{-closure}(r_i)$
3. Make a state an accepting state if it includes any final states in the ϵ -NFA.

In simple terms: Just like converting a regular NFA to a DFA except follow the epsilon transitions whenever possible after processing an input

Epsilon Elimination Example

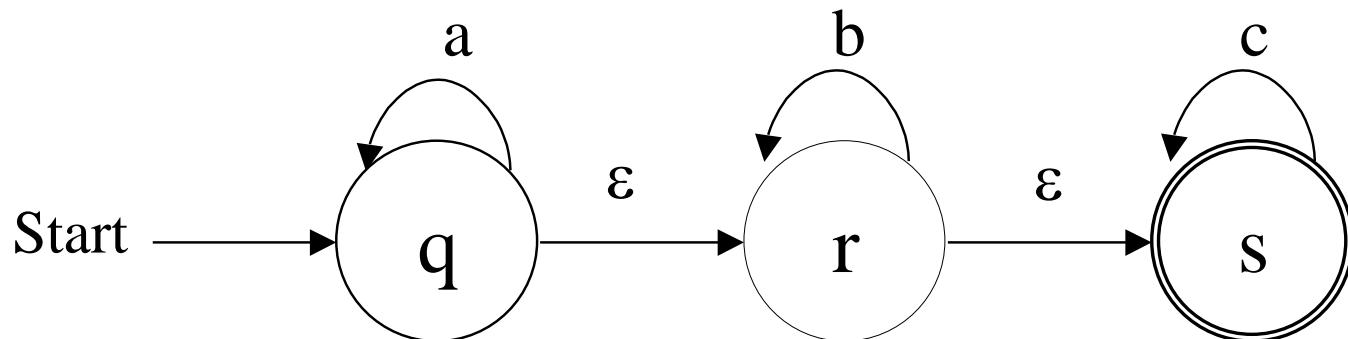


Converts to:



Epsilon Elimination Exercise

- Exercise: Here is the ϵ -NFA for the language consisting of zero or more a's followed by zero or more b's followed by zero or more c's.
- Eliminate the epsilon transitions.



FSM with Outputs

- The goal of is to describe a circuit with inputs and outputs.
- Finite automata may have outputs corresponding to each transition. There are two types of finite state machines that generate output –
 - Mealy Machine
 - Moore machine

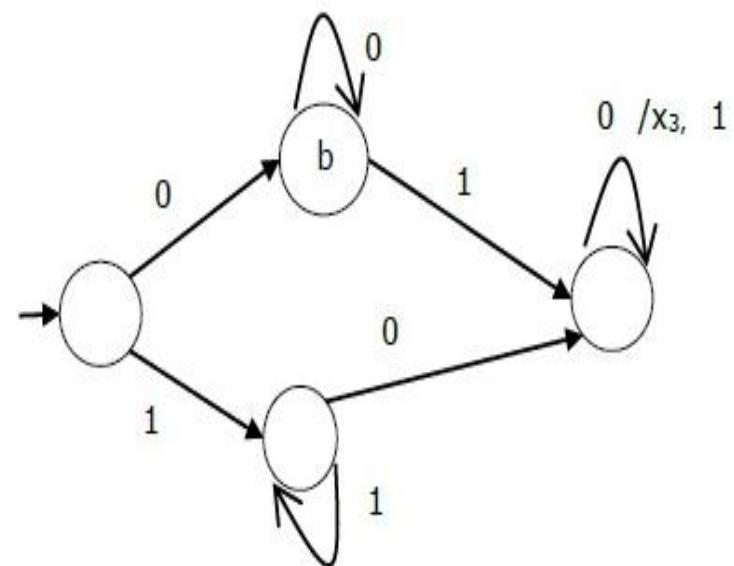
Mealy Machine

- A Mealy Machine is an FSM whose output depends on the present state as well as the present input.
- It can be described by a 6 tuple $(Q, \Sigma, O, \delta, X, q_0)$

- Q is a finite set of states.
- Σ is a finite set of symbols called the input alphabet.
- O is a finite set of symbols called the output alphabet.
- δ is the input transition function where $\delta: Q \times \Sigma \rightarrow Q$
- X is the output transition function where $X: Q \times \Sigma \rightarrow O$
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).

The state Table & Diagram of a Mealy Machine

Present state	Next state			
	input = 0		input = 1	
	State	Output	State	Output
$\rightarrow a$	b	x_1	c	x_1
b	b	x_2	d	x_3
c	d	x_3	c	x_1
d	d	x_3	d	x_2



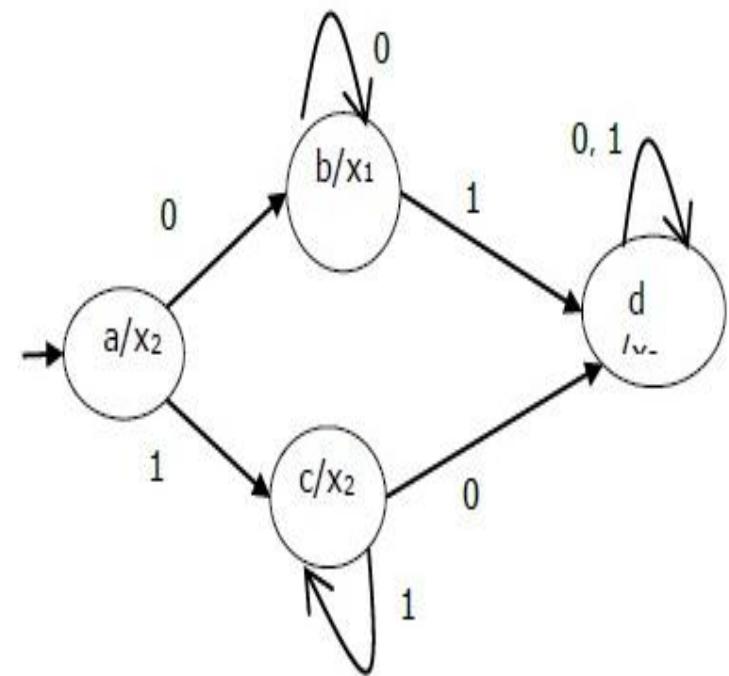
Moore Machine

- Moore machine is an FSM whose outputs depend on only the present state.
- A Moore machine can be described by a 6 tuple $(Q, \Sigma, O, \delta, X, q_0)$

- Q is a finite set of states.
- Σ is a finite set of symbols called the input alphabet.
- O is a finite set of symbols called the output alphabet.
- δ is the input transition function where $\delta: Q \times \Sigma \rightarrow Q$
- X is the output transition function where $X: Q \rightarrow O$
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).

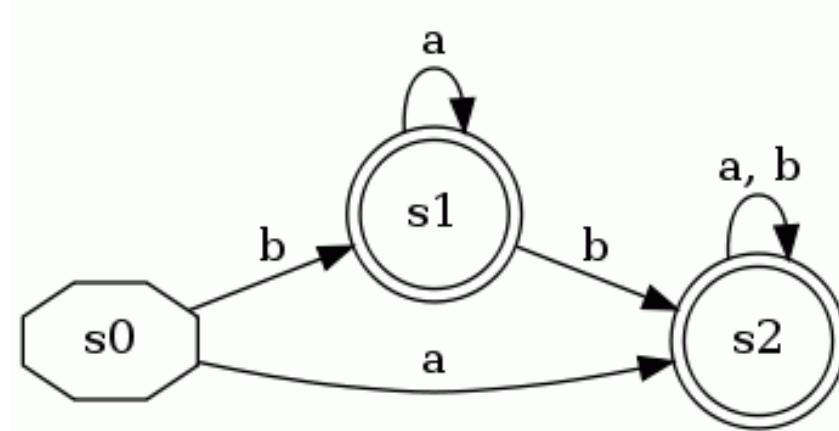
The state Table & Diagram of a Moore Machine

Present state	Next State		Output
	Input = 0	Input = 1	
$\rightarrow a$	b	c	x_2
b	b	d	x_1
c	c	d	x_2
d	d	d	x_3



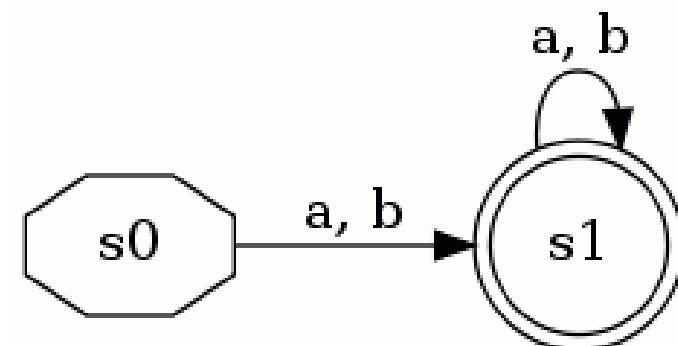
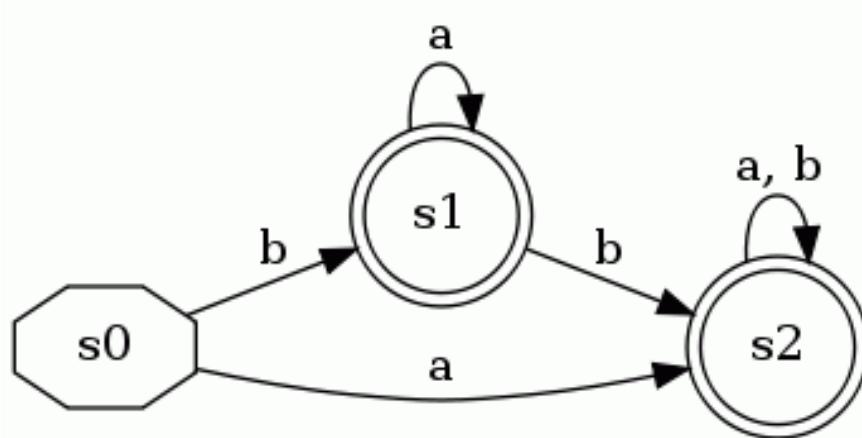
DFA Minimization

- Some states can be redundant:
 - The following DFA accepts $(a|b)^+$
 - State s_1 is not necessary



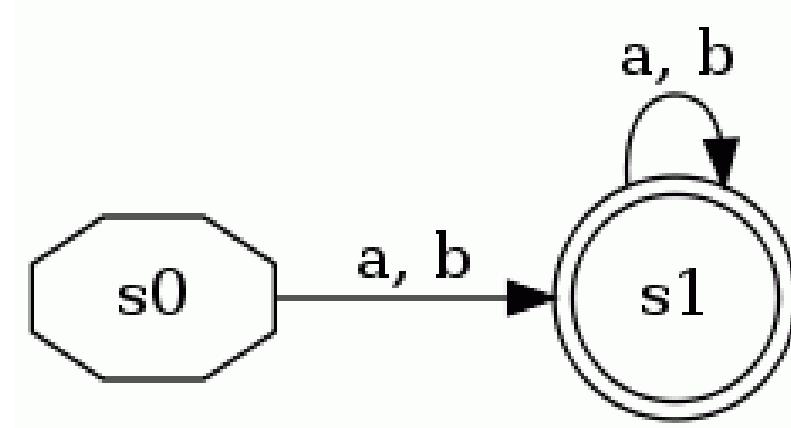
DFA Minimization

- So these two DFAs are *equivalent*:



DFA Minimization

- This is a *state-minimized* (or just *minimized*) DFA
 - Every remaining state is necessary



DFA Minimization

- The task of *DFA minimization*, then, is to automatically transform a given DFA into a state-minimized DFA
 - Several algorithms and variants are known
 - Note that this also in effect can minimize an NFA (since we know algorithm to convert NFA to DFA)

DFA Minimization Algorithm

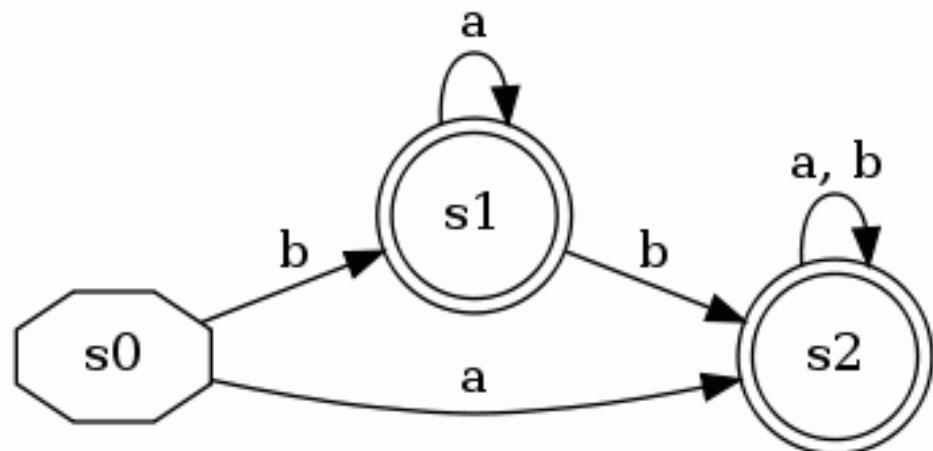
- Recall that a DFA $M=(Q, \Sigma, \delta, q_0, F)$
- Two states p and q are distinct if
 - p in F and q not in F or vice versa, or
 - for some α in Σ , $\delta(p, \alpha)$ and $\delta(q, \alpha)$ are distinct
- Using this inductive definition, we can calculate which states are distinct

DFA Minimization Algorithm

- Create lower-triangular table DISTINCT, initially blank
- For every pair of states (p,q) :
 - If p is final and q is not, or vice versa
 - $\text{DISTINCT}(p,q) = \varepsilon$
- Loop until no change for an iteration:
 - For every pair of states (p,q) and each symbol α
 - If $\text{DISTINCT}(p,q)$ is blank and $\text{DISTINCT}(\delta(p,\alpha), \delta(q,\alpha))$ is not blank
 - $\text{DISTINCT}(p,q) = \alpha$
- Combine all states that are not distinct

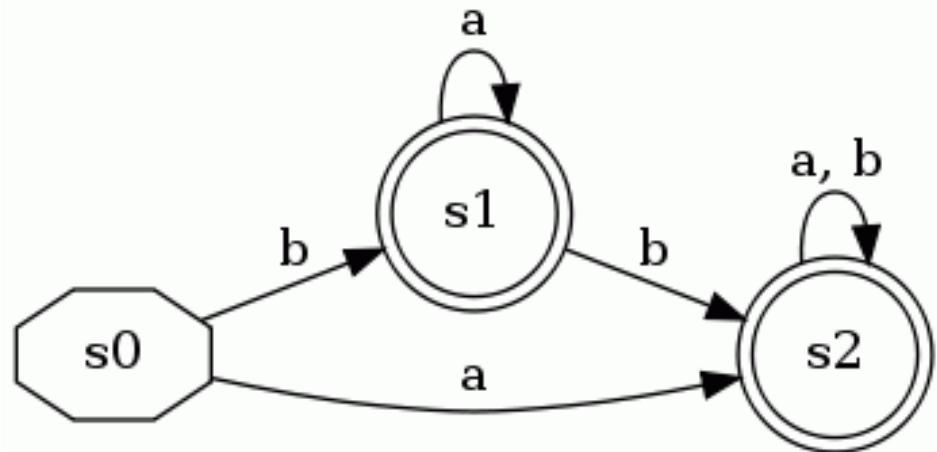
Example

s_0			
s_1			
s_2			
	s_0	s_1	s_2



Example

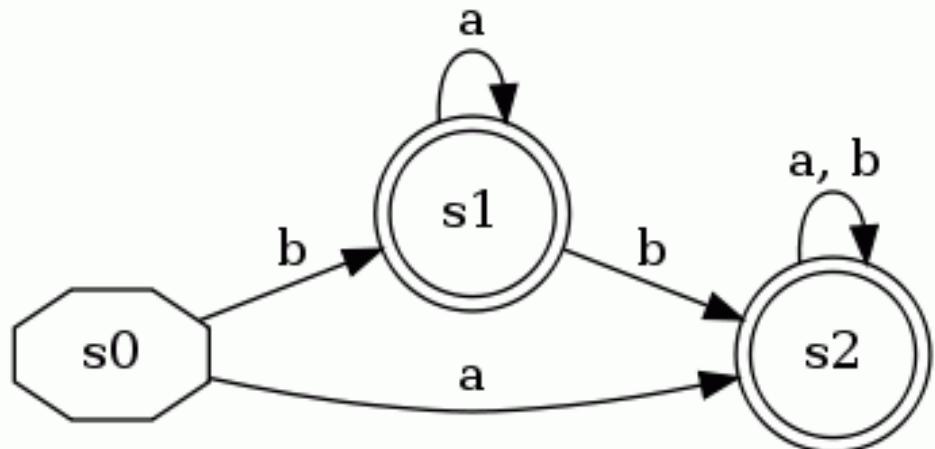
s_0			
s_1	ϵ		
s_2	ϵ		
	s_0	s_1	s_2



Label pairs with ϵ where one is a final state and the other is not

Example

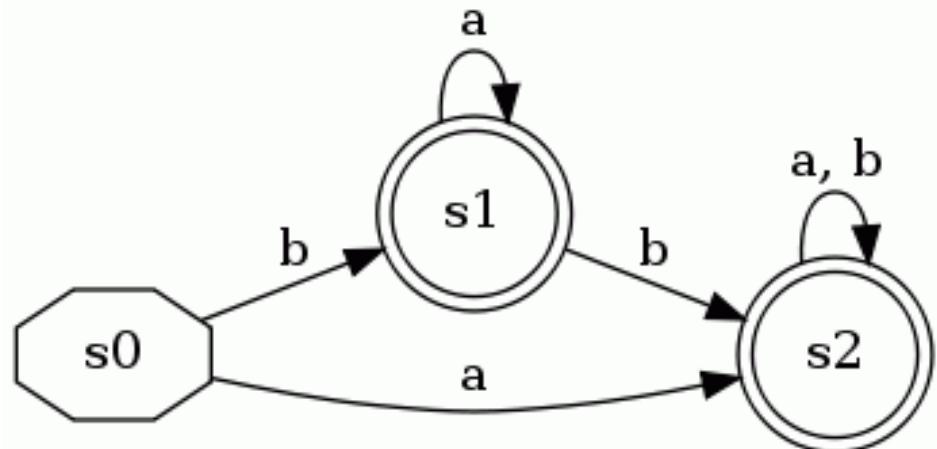
s_0			
s_1	ϵ		
s_2	ϵ		
	s_0	s_1	s_2



Main loop (no changes occur)

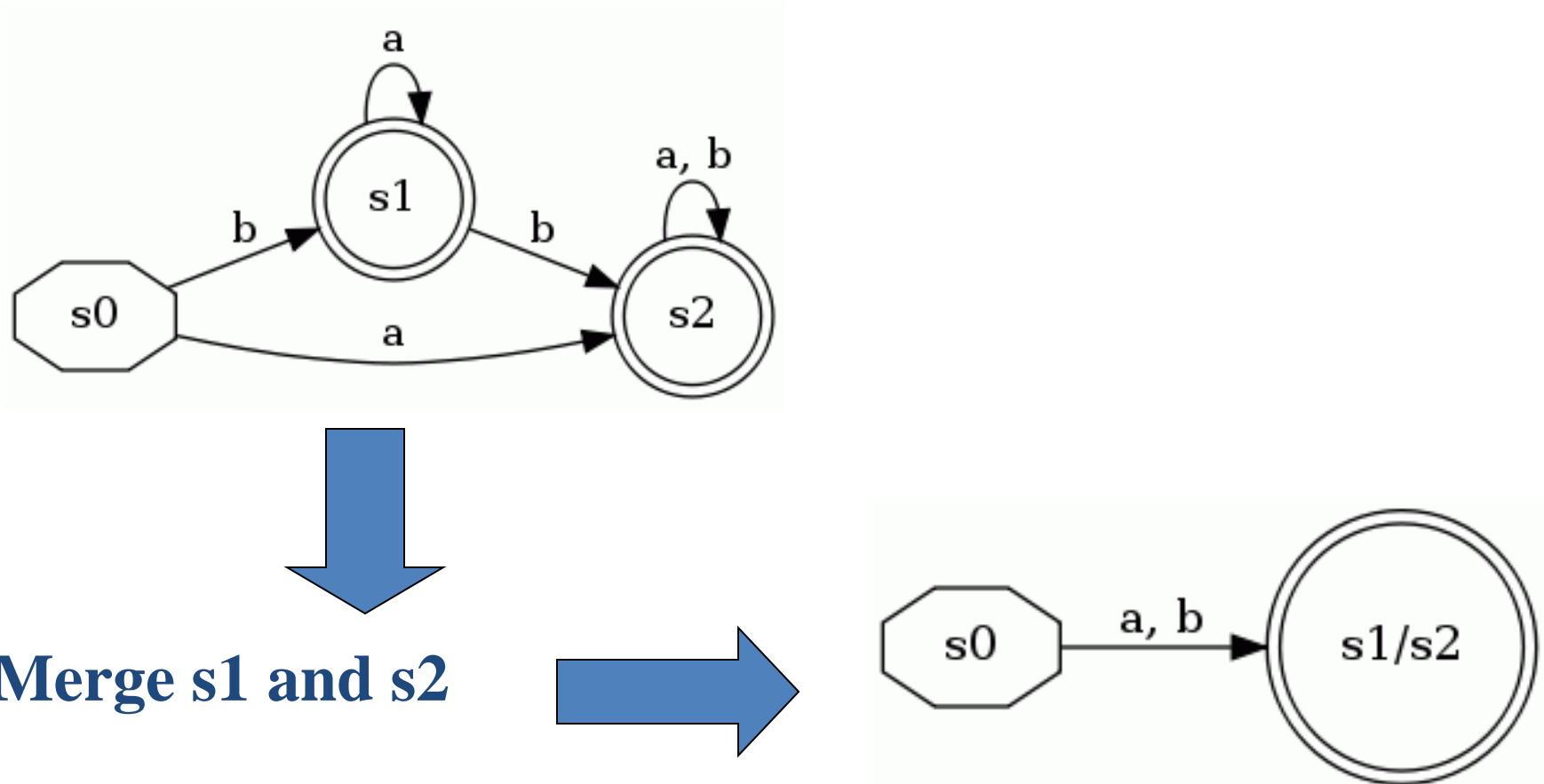
Example

s_0			
s_1	ϵ		
s_2	ϵ		
	s_0	s_1	s_2



$\text{DISTINCT}(s_1, s_2)$ is empty, so s_1 and s_2 are equivalent states

Example



Myhill Nerode Theorem:

The following three statements are equivalent

1. The set $L \in \Sigma^*$ is accepted by a FSA
2. L is the union of some of the equivalence classes of a right invariant equivalence relation of finite index.
3. Let equivalence relation R_L be defined by :
 xR_Ly iff for all z in Σ^* xz is in L exactly when yz is in L .
Then R_L is of finite index.

Theorem Proof:

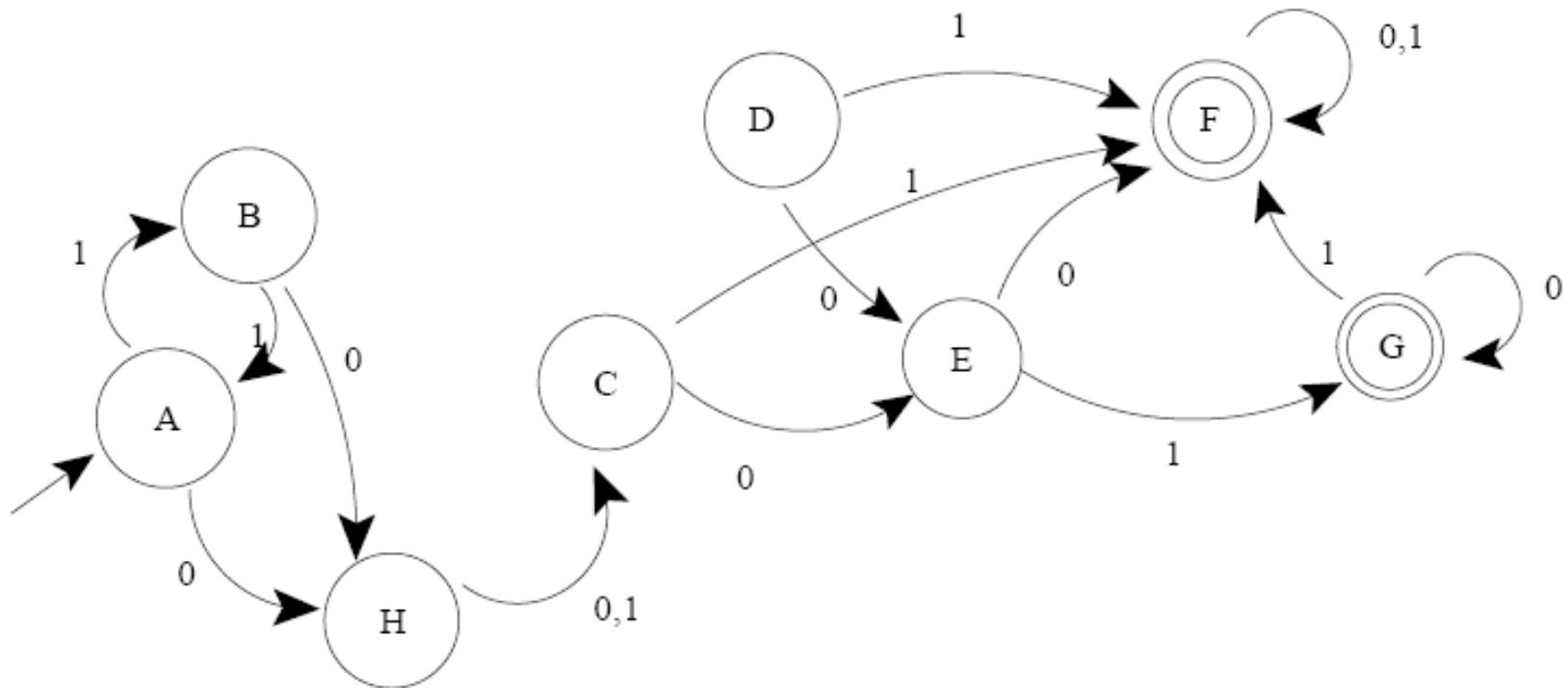
- There are three conditions:
 1. Condition (i) implies condition (ii)
 2. Condition (ii) implies condition (iii)
 3. Condition (iii) implies condition (i)

Equivalence Relation

A binary relation \sim over a set X is an equivalence relation if it satisfies

- Reflexivity
- Symmetry
- Transitivity

Myhill Nerode Theorem - Example



Example

- Check for pairs with one state final and one not:

b								
c								
d								
e								
f	ϵ	ϵ	ϵ	ϵ	ϵ			
g	ϵ	ϵ	ϵ	ϵ	ϵ			
h						ϵ	ϵ	
	a	b	c	d	e	f	g	

Example

- First iteration of main loop:

b							
c	1	1					
d	1	1					
e	0	0	0				
f	ϵ	ϵ	ϵ	ϵ			
g	ϵ	ϵ	ϵ	ϵ			
h			1	1	0	ϵ	ϵ
	a	b	c	d	e	f	g

Example

- Second iteration of main loop:

b							
c	1	1					
d	1	1					
e	0	0	0	0			
f	ϵ	ϵ	ϵ	ϵ	ϵ		
g	ϵ	ϵ	ϵ	ϵ	ϵ		
h	1	1	1	1	0	ϵ	ϵ
	a	b	c	d	e	f	g

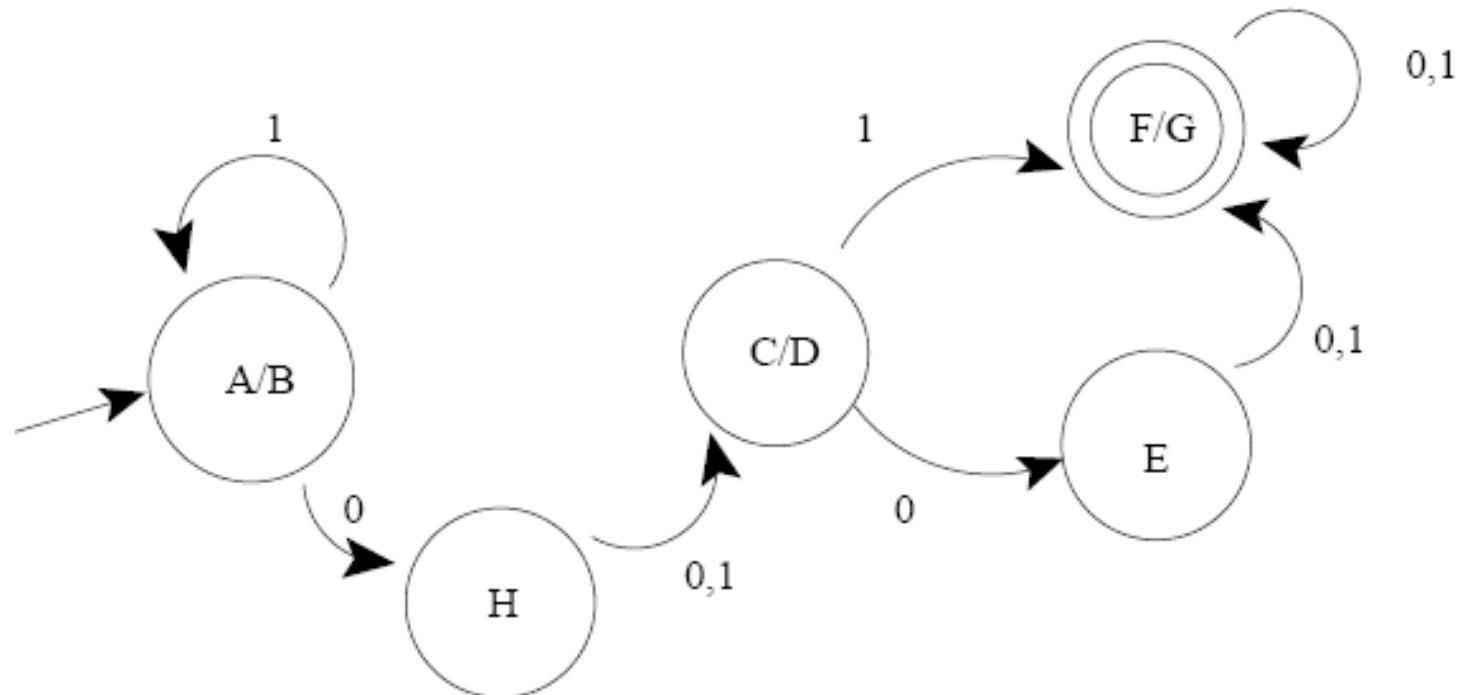
Example

- Third iteration makes no changes
 - Blank cells are equivalent pairs of states

	a	b	c	d	e	f	g
b							
c	1	1					
d	1	1					
e	0	0	0	0			
f	ϵ	ϵ	ϵ	ϵ	ϵ		
g	ϵ	ϵ	ϵ	ϵ	ϵ		
h	1	1	1	1	0	ϵ	ϵ

Example

- Combine equivalent states for minimized DFA:



Unit-2

Regular Languages: Basics of Regular Expressions, Identities of Regular Expression, The Arden's Theorem, Construct RE from FA, Equivalence of Two FAs, Construct FA from RE, Equivalence of Two REs, Regular grammars and equivalence of two finite automata, Pumping Lemma for RLs, Applications of Pumping Lemma, Closure properties of Regular Sets, Applications of Regular Expressions.

Basics of Regular Expressions

Regular expression over alphabet Σ

- \emptyset is a regular expression.
- ϵ is a regular expression.
- For any $a \in \Sigma$, a is a regular expression.
- If r_1 and r_2 are regular expressions, then
 - $(r_1 + r_2)$ is a regular expression.
 - $(r_1 \cdot r_2)$ is a regular expression.
 - (r_1^*) is a regular expression.
- Nothing else is a regular expression.

Regular Languages

- \emptyset is a regular language corresponding to the regular expression \emptyset .
- $\{\varepsilon\}$ is a regular language corresponding to the regular expression ε .
- For any symbol $a \in \Sigma$, $\{a\}$ is a regular language corresponding to the regular expression a .
- If L_1 and L_2 are regular languages corresponding to the regular expression r_1 and r_2 , then
 - $L_1 \cup L_2$, $L_1 \cdot L_2$, and L_1^* are regular languages corresponding to $(r_1 + r_2)$, $(r_1 \cdot r_2)$, and (r_1^*) .

Simple examples

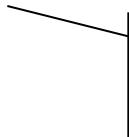
Let $\Sigma = \{0,1\}$.

- $\{\alpha \in \Sigma^* | \alpha \text{ does not contain } 1's\}$
 - (0^*)
- $\{\alpha \in \Sigma^* | \alpha \text{ contains } 1's \text{ only}\}$
 - $(1 \cdot (1^*))$ (which can be denoted by (1^+))
- Σ^*
 - $((0+1)^*)$
- $\{\alpha \in \Sigma^* | \alpha \text{ contains only } 0's \text{ or only } 1's\}$
 - $((00^*) + (11^*))$ $0^* + 1^* \neq (0+1)^*$

Some more Notations of RE

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

- Parentheses in regular expressions can be omitted when the order of evaluation is clear.
 - $((0+1)^*) \neq 0+1^*$
 - $((0^*)+(1^*)) = 0^* + 1^*$
- For concatenation, \cdot can be omitted.
- $r \cdot r \cdot r \dots r$ is denoted by r^n .



n times

More complex examples

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

- $\{\alpha \in \Sigma^* \mid \alpha \text{ contains odd number of } 1\text{'s}\}$
 - $0^*(10^*10^*)^*10^*$
- $\{\alpha \in \Sigma^* \mid \text{any two } 0\text{'s in } \alpha \text{ are separated by three } 1\text{'s}\}$
 - $1^*(0111)^*01^* + 1^*$
- $\{\alpha \in \Sigma^* \mid \alpha \text{ is a binary number divisible by } 4\}$
 - $(0+1)^*00$
- $\{\alpha \in \Sigma^* \mid \alpha \text{ does not contain } 11\}$
 - $0^*(10^+)^* (1+\varepsilon) \text{ or } (0+10)^* (1+\varepsilon)$

Notation

Let r be a regular expression.

The regular language corresponding to the regular expression r is denoted by $L(r)$.

Some rules/ identities for language operations

Let r, s and t be languages over $\{0,1\}$

$$r + \emptyset = \emptyset + r = r$$

$$r + s = s + r$$

$$r \cdot \varepsilon = \varepsilon \cdot r = r$$

$$r \cdot \emptyset = \emptyset \cdot r = \emptyset$$

$$r \cdot (s + t) = r \cdot s + r \cdot t$$

$$r^+ = r r^*$$

Rules for Regular Expressions

Let r, s and t be regular expressions over $\{0,1\}$.

$$\emptyset^* = \varepsilon$$

$$\varepsilon^* = \varepsilon$$

$$(r + \varepsilon)^+ = r^*$$

$$r^* = r^*(r + \varepsilon) = r^* \quad r^* = (r^*)^*$$

$$(r^*s^*)^* = (r + s)^*$$

Closure properties of the class of regular languages (Part 1)

Theorem: The class of regular languages is closed under union, concatenation, and Kleene's star.

Proof: Let L_1 and L_2 be regular languages over Σ .

Then, there are regular expressions r_1 and r_2 corresponding to L_1 and L_2 .

By the definition of regular expression and regular languages, $r_1 + r_2$, $r_1 \cdot r_2$, and r_1^* are regular expressions corresponding to $L_1 \cup L_2$, $L_1 \cdot L_2$, and L_1^* .

Thus, the class of regular languages is closed under union, concatenation, and Kleene's star.

Equivalence of language accepted by FA and regular languages

To show that the languages accepted by FA and regular languages are equivalent, we need to prove:

- For any regular language L , there exists an FA M such that $L = L(M)$.
- For any FA M , $L(M)$ is a regular language.

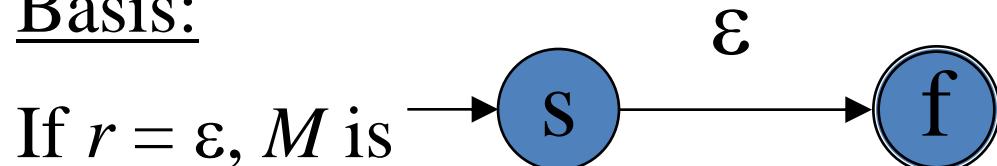
For any regular language L , there exists an FA M such that $L = L(M)$

Proof: Let L be a regular language.

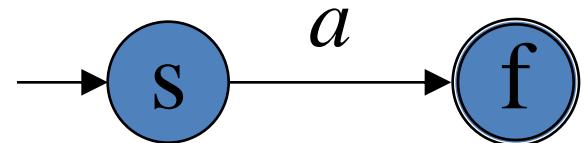
Then, \exists a regular expression r corresponding to L .

We construct an NFA M , from the regular expression r , such that $L=L(M)$.

Basis:



If $r = \{a\}$ for some $a \in \Sigma$, M is



Proof (cont'd)

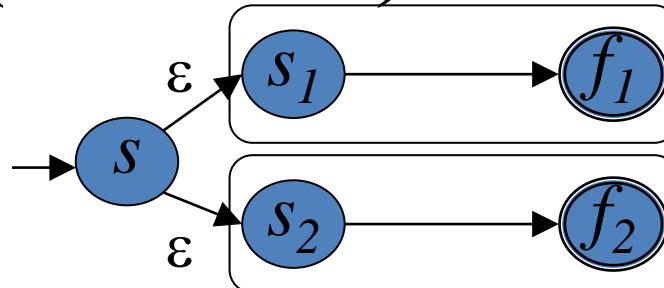
Induction hypotheses: Let r_1 and r_2 be regular expressions with less than n operations. And, there are NFA's M_1 and M_2 accepting regular languages corresponding to $L(r_1)$ and $L(r_2)$.

Induction step: Let r be a regular expression with n operations. We construct an NFA accepting $L(r)$.

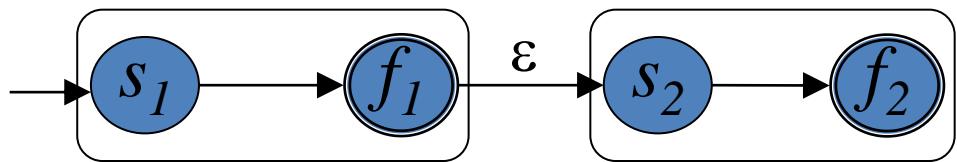
r can be in the form of either $r_1 + r_2$, $r_1 \cdot r_2$, or r_1^* , for regular expressions r_1 and r_2 with less than n operations.

Proof (cont'd)

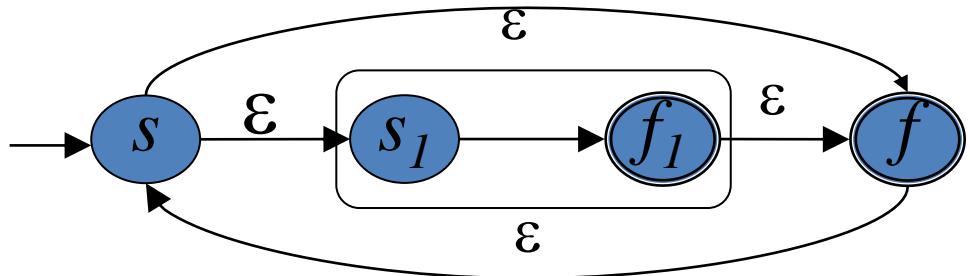
If $r = r_1 + r_2$, then M is



If $r = r_1 \cdot r_2$, then M is



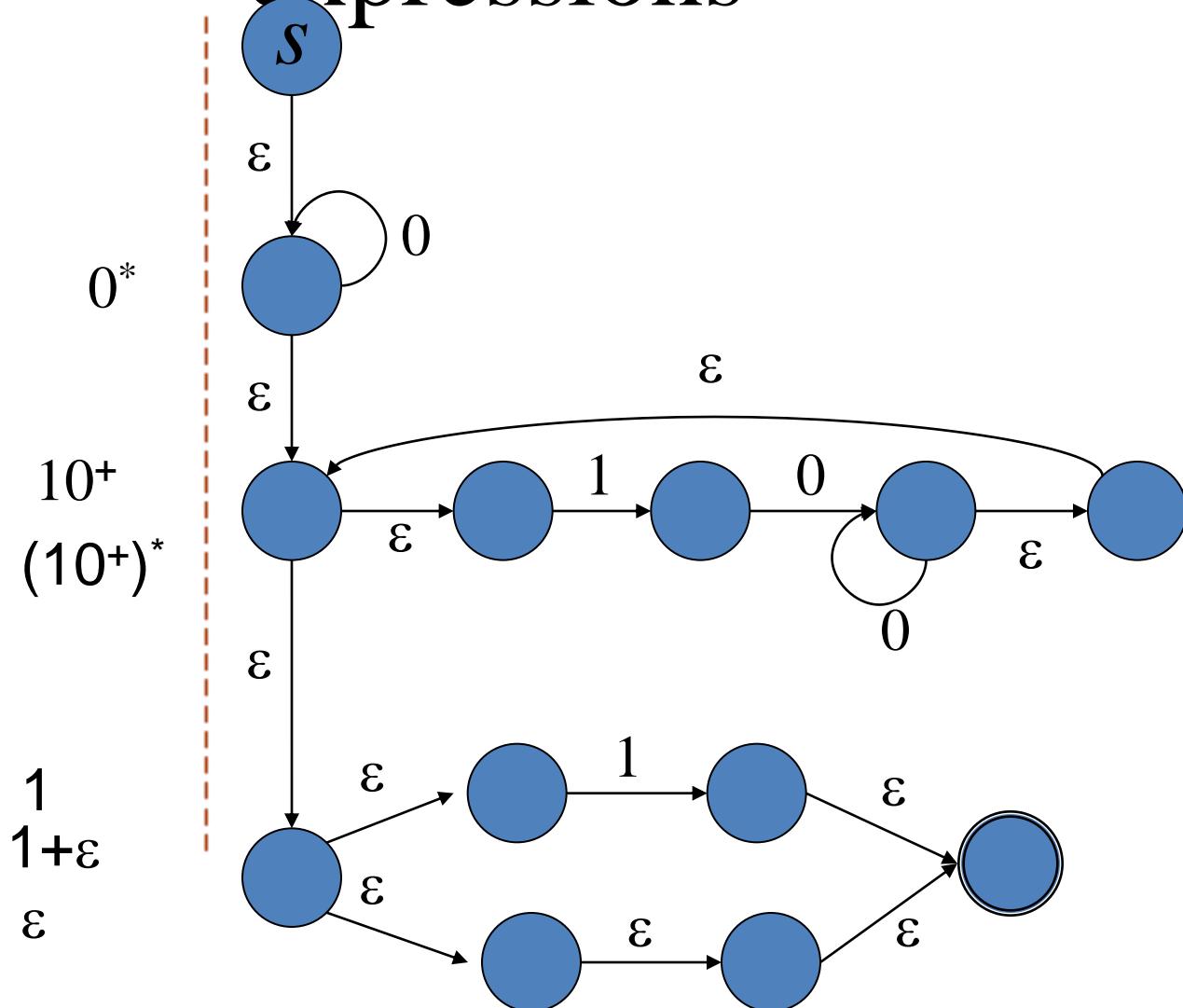
If $r = r_1^*$, then M is



Therefore, there is an NFA accepting $L(r)$ for any regular expression r .

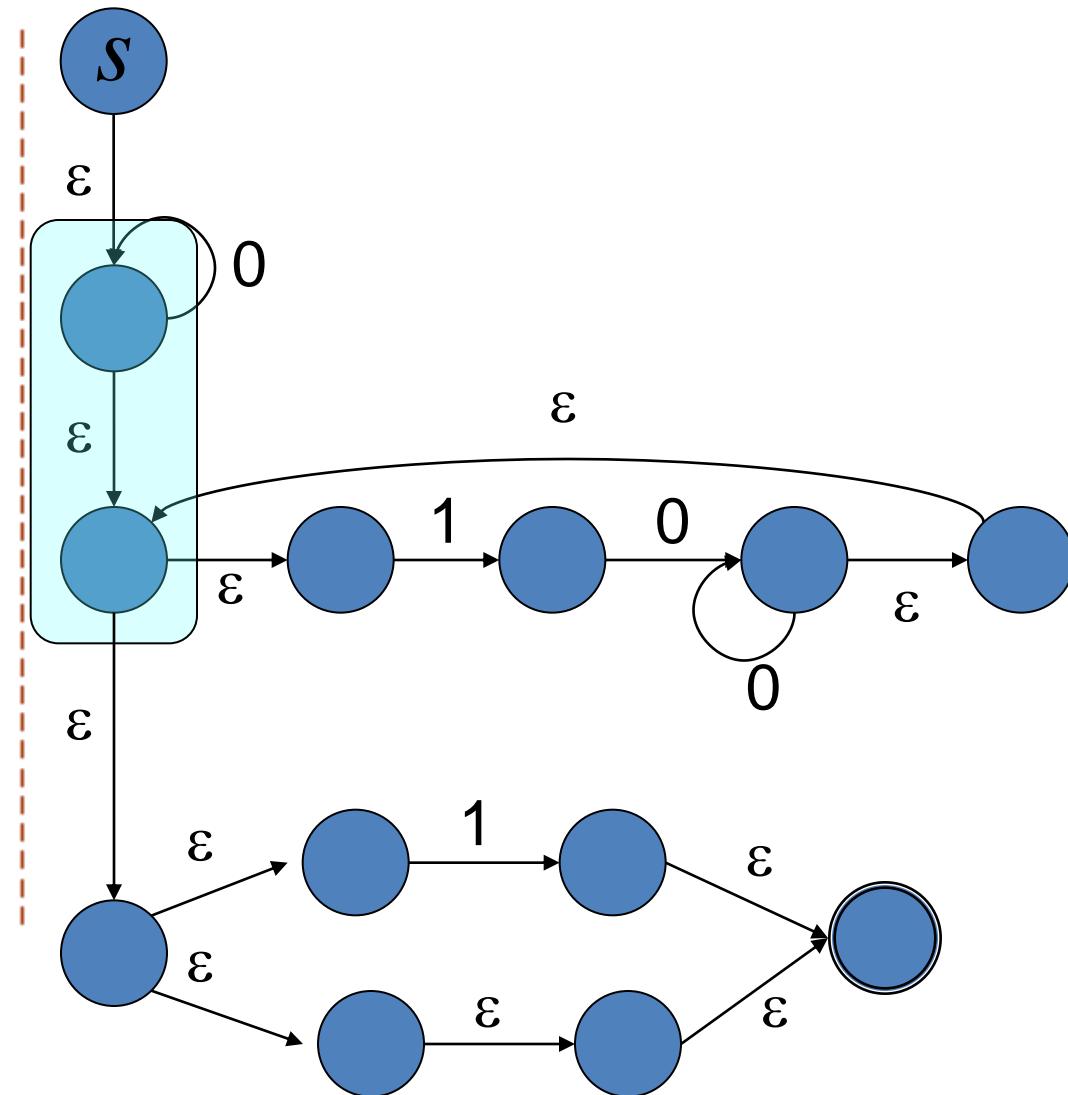
Constructing NFA for regular expressions

- $0^*(10^+)^*$
 $(1+\varepsilon)$



Some Observations

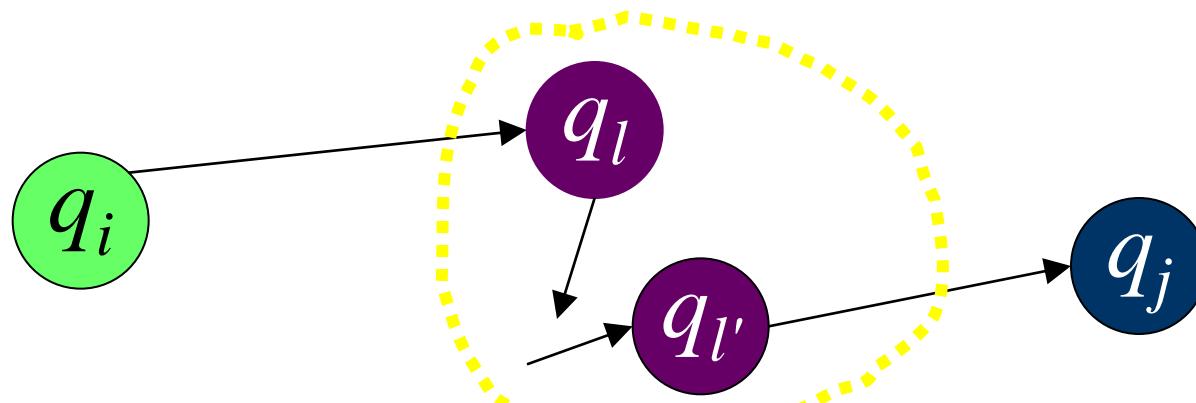
- Can these two states be merged?
NO
- Be careful when you decide to merge some states.



For any FA M , $L(M)$ is a regular language

Proof: Let $M = (Q, \Sigma, \delta, q_I, F)$ be an FA, where $Q = \{q_i \mid 1 \leq i \leq n\}$ for some positive integer n .

Let $R(i, j, k)$ be the set of all strings in that drive M from state q_i to state q_j while passing through any state q_l , for $l \leq k$. (i and j can be any states)

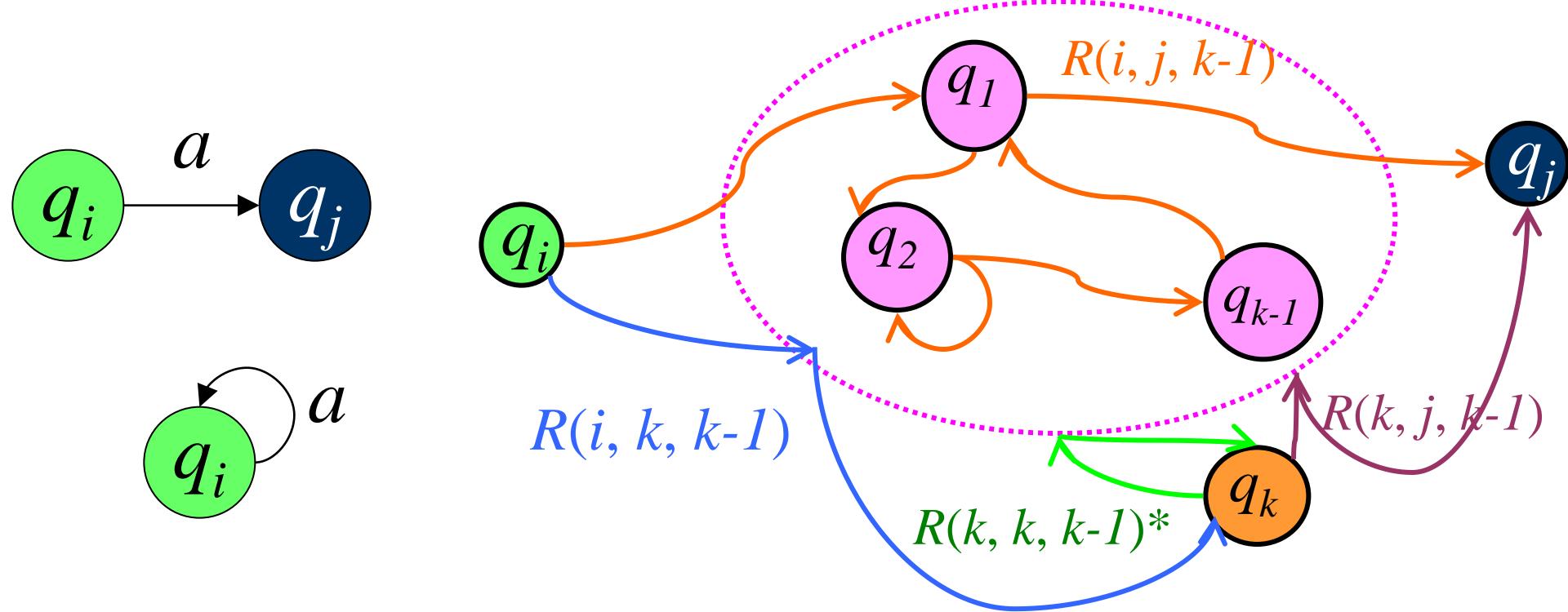


Proof (cont'd)

$$R(i, j, 0) = +_{qj \in (qi, a)} a \text{ if } i \neq j$$

$$R(i, j, 0) = +_{qj \in (qi, a)} a + \varepsilon \text{ if } i = j$$

$$R(i, j, k) = R(i, j, k-1) + R(i, k, k-1) R(k, k, k-1)^* R(k, j, k-1)$$



Proof (cont'd)

Then, $L(M) = + R(\textcolor{green}{1}, f, \textcolor{magenta}{n})$ for all q_f in F .

$$R(\textcolor{green}{i}, \textcolor{blue}{j}, 0) = +_{q_j \in (q_i, a)} a \text{ if } \textcolor{green}{i} \neq \textcolor{blue}{j}$$

$$R(\textcolor{green}{i}, \textcolor{blue}{j}, 1) = \{a \in \Sigma \mid (q_i, a) = q_j\} + \varepsilon \text{ if } \textcolor{green}{i} = \textcolor{blue}{j}$$

$$R(\textcolor{green}{i}, \textcolor{blue}{j}, \textcolor{orange}{k}) = R(\textcolor{green}{i}, \textcolor{blue}{j}, \textcolor{magenta}{k-1}) + R(\textcolor{green}{i}, \textcolor{orange}{k}, \textcolor{magenta}{k-1}) R(\textcolor{orange}{k}, \textcolor{orange}{k}, \textcolor{magenta}{k-1})^* R(\textcolor{orange}{k}, \textcolor{blue}{j}, \textcolor{magenta}{k-1})$$

Proof (cont'd)

We prove that $L(M)$ is a regular language by showing that there is a regular expression corresponding to $L(M)$, by induction.

Basis: $R(i, j, 0)$ corresponds to a regular expression a if $i \neq j$ and $a + \epsilon$ if $i = j$ for some $a \in \Sigma$.

Induction hypotheses: Let $R(i, j, k-1)$ correspond to a regular expression, for any $i, j, k \leq n$.

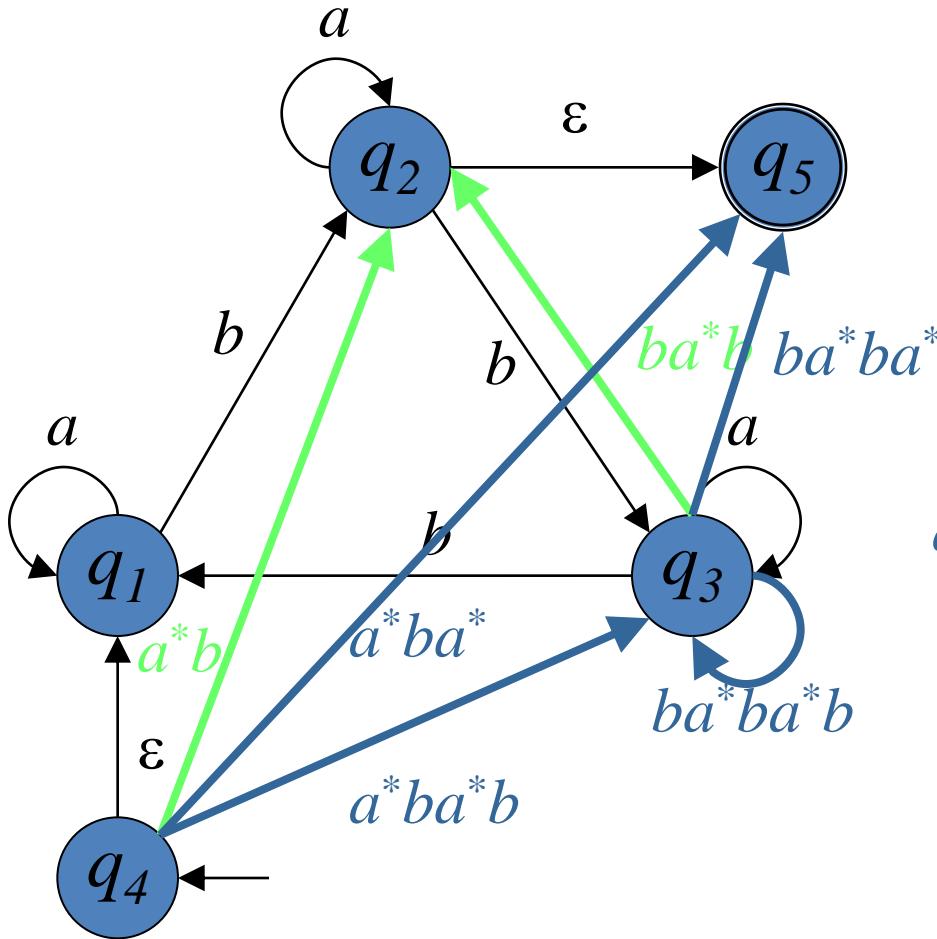
Proof (cont'd)

Induction step: $R(i, j, k) = R(i, j, k-1) \cup R(i, k, k-1)$

$R(k, k, k-1)^*$ $R(k, j, k-1)$ also corresponds to a regular expression because $R(i, j, k-1)$, $R(i, k, k-1)$, $R(k, k, k-1)$ and $R(k, j, k-1)$ correspond to some regular expressions and union, concatenation, and Kleene's star are allowed in regular expressions.

Therefore, $L(M)$ is also a regular language because $L(M) = + R(1, f, n)$ for all q_f in F .

Find a regular expression for a DFA



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

$a^*ba^* (\epsilon+b(ba^*ba^*b+a)^* ba^*ba^*)$

Arden's Theorem

- *Statement* –
- Let P and Q be two regular expressions.
- If P does not contain null string, then $R = Q + RP$ has a unique solution that is $R = QP^*$

(cont.,)

- **Proof –**
- $R = Q + (Q + RP)P$ [After putting the value $R = Q + RP$]
- $= Q + QP + RPP$
- When we put the value of **R** recursively again and again, we get the following equation –
- $R = Q + QP + QP^2 + QP^3 \dots$
- $R = Q (\varepsilon + P + P^2 + P^3 + \dots)$
- $R = QP^*$ [As $P^* = \varepsilon + P_1 + P_2 + P_3 + \dots$]
- Hence, proved.

Steps –Arden’s Theorem

- **Assumptions for Applying Arden’s Theorem**
- The transition diagram must not have NULL transitions
- It must have only one initial state
- **Method**
- **Step 1** – Create equations as the following form for all the states of the DFA having n states with initial state q_1 .
 - $q_1 = q_1R_{11} + q_2R_{21} + \dots + q_nR_{n1} + \varepsilon$
 - $q_2 = q_1R_{12} + q_2R_{22} + \dots + q_nR_{n2}$
 - $q_n = q_1R_{1n} + q_2R_{2n} + \dots + q_nR_{nn}$

Steps(Cont.,)

- R_{ij} represents the set of labels of edges from q_i to q_j , if no such edge exists, then $R_{ij} = \emptyset$
- **Step 2** – Solve these equations to get the equation for the final state in terms of R_{ij}

Pumping Lemma

Let L be a regular language.

Then, there exists an integer $n \geq 0$ such that for every string x in L that $|x| \geq n$, there are strings u , v , and w such that

- $x = u v w$,
- $v \neq \epsilon$,
- $|u v| \leq n$, and
- for all $k \geq 0$, $u v^k w$ is also in L .

Any language L is not a regular language *if* for any integer $n \geq 0$, there is a string x in L such that $|x| \geq n$, for any strings u , v , and w ,

- $x \neq u v w$, or
- $v = \epsilon$, or
- Not ($|u v| \leq n$), or
- there is $k \geq 0$, $u v^k w$ is not in L

Pumping Lemma

Any language L is not a regular language *if*

- for any integer $n \geq 0$,
- there is a string x in L such that $|x| \geq n$,
- for any strings u , v and w , such that $x = u v w$,
 $v \neq \varepsilon$, and $|u v| \leq n$,
 - there is $k \geq 0$, $u v^k w$ is not in L

Using Pumping Lemma

- Given a language L.
- Let n be any integer ≥ 0 .
- Choose a string x in L that $|x| \geq n$.
- Consider all possible ways to chop x into u , v and w such that $v \neq \epsilon$, and $|uv| \leq n$.
- For all possible u , v , and w , show that there is $k \geq 0$ such that $u v^k w$ is not in L .
- Then, we can conclude that L is not regular.

If you pick a wrong x , your proof will fail ! It does not mean that L is regular !!!!!

Prove $\{0^i 1^i \mid i \geq 0\}$ is not regular

Let $L = \{0^i 1^i \mid i \geq 0\}$.

Let n be any integer ≥ 0 .

Let $x = 0^n 1^n$.

Make sure that x is in L and $|x| \geq n$.

The only possible way to chop x into u , v , and w such that $v \neq \epsilon$, and $|u v| \leq n$ is:

$u = 0^p$, $v = 0^q$, $w = 0^{n-p-q} 1^n$, where $0 \leq p < n$ and $0 < q \leq n$

Show that there is $k \geq 0$, $u v^k w$ is not in L .

$u v^k w = 0^p 0^{qk} 0^{n-p-q} 1^n = 0^{p+qk+(n-p-q)} 1^n = 0^{n+q(k-1)} 1^n$

If $k \neq 1$, then $n+q(k-1) \neq n$ and $u v^k w$ is not in L .

Then, L is not regular.

Prove $\{1^i | i \text{ is prime}\}$ is not regular

Let $L = \{1^i | i \text{ is prime}\}.$

Let n be any integer $\geq 0.$

Let p be a prime $\geq n,$ and $w = 1^p.$

Only one possible way to chop w into $x, y,$ and z such that $y \neq \varepsilon,$ and $|x y| \leq n$ is:

$x = 1^q, y = 1^r, z = 1^{p-q-r},$ where $0 \leq q < n$ and $0 < r < n$

Show that there is $k \geq 0, x y^k z$ is not in $L.$

$$x y^k z = 1^q 1^{rk} 1^{p-q-r} = 1^{q+rk+(p-q-r)} = 1^{p+r(k-1)}$$

If $k=p+1,$ then $p+r(k-1) = p(r+1),$ which is not a prime.

Then, $x y^k z$ is not in $L.$

Then, L is not regular.

Closure property

Let ∇ be a binary operation on languages and the class of regular languages is closed under ∇ . (∇ can be \cup , \cap , or $-$)

- If L_1 and L_2 are regular, then $L_1 \nabla L_2$ is regular.
- If $L_1 \nabla L_2$ is not regular, then L_1 or L_2 are not regular.
- If $L_1 \nabla L_2$ is not regular but L_2 is regular, then L_1 is not regular.

Prove that $\{w \in \{0,1\}^* \mid \text{the number of 0's and 1's in } w \text{ are equal}\}$ is not regular

Let $L = \{w \in \{0,1\}^* \mid \text{the number of 0's and 1's in } w \text{ are equal}\}$.

Let $R = \{0^i 1^i \mid i \geq 0\}$.

$R = 0^* 1^* \cap L$

We already prove that R is not regular.

But $0^* 1^*$ is regular.

Then, L is not regular.

Decision properties of RLs

- Consider the following typical and important question:
- w and a regular language L , is an element of L ?
- The answer is either yes or no.

- While w is represent explicitly, we wonder how L given to us. Obviously, L cannot be given as an enumeration of strings (L may be infinite). L will be represented either by a *DFA* , *NFA* or regular expression.
- The question presented above is called the “membership problem” for the corresponding regular language L .
- If L is represented by a *DFA* , the problem has an easy solution.

Application of Finite Automata

- Used to design Lexical Analyser
- Text Editor designing
- Useful in spell checker
- Used to design Sequential Circuit design

Applications of Regular Expression

- Web Search Engines
- Software Engineering
- Lexical Analysis

UNIT -3

Context Free Grammar: Context-free grammars (CFG) , Derivation trees, Ambiguity in CFG , Left recursion and Left factoring, Simplification of CFGs, Chomsky Normal form and Greibach Normal form, Pumping lemma for Context-free languages, closure properties of CFLs.

CFG

- A *context-free grammar* is a notation for describing languages.
- It is more powerful than finite automata or RE's, but still cannot define all possible languages.
- Useful for nested structures, e.g., parentheses in programming languages.

(cont.,)

- Basic idea is to use “variables” to stand for sets of strings (i.e., languages).
- These variables are defined recursively, in terms of one another.
- Recursive rules (“productions”) involve only concatenation.
- Alternative rules for a variable allow union.

Example: CFG for $\{ 0^n 1^n \mid n \geq 1 \}$

- Productions:

$S \rightarrow 01$

$S \rightarrow 0S1$

- Basis: 01 is in the language.
- Induction: if w is in the language, then so is $0w1$.

CFG Formalism

- *Terminals* = symbols of the alphabet of the language being defined.
- *Variables* = *nonterminals* = a finite set of other symbols, each of which represents a language.
- *Start symbol* = the variable whose language is the one being defined.

Productions of CFG

- A *production* has the form variable \rightarrow string of variables and terminals.
- Convention:
 - A, B, C,... are variables.
 - a, b, c,... are terminals.
 - ..., X, Y, Z are either terminals or variables.
 - ..., w, x, y, z are strings of terminals only.
 - $\alpha, \beta, \gamma, \dots$ are strings of terminals and/or variables.

Example: Formal CFG

- Here is a formal CFG for $\{ 0^n 1^n \mid n \geq 1 \}$.
- Terminals = {0, 1}.
- Variables = {S}.
- Start symbol = S.
- Productions =
 $S \rightarrow 01$
 $S \rightarrow 0S1$

Derivations

- We *derive* strings in the language of a CFG by starting with the start symbol, and repeatedly replacing some variable A by the right side of one of its productions.
 - That is, the “productions for A” are those that have A on the left side of the \rightarrow .

Derivations – Formalism

- We say $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production.
- Example: $S \rightarrow 01; S \rightarrow 0S1.$
- $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000111.$



Iterated Derivation

- \Rightarrow^* means “zero or more derivation steps.”
- Basis: $\alpha \Rightarrow^* \alpha$ for any string α .
- Induction: if $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \Rightarrow^* \gamma$.

Example: Iterated Derivation

- $S \rightarrow 01; S \rightarrow 0S1.$
- $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000111.$
- So $S \Rightarrow^* S; S \Rightarrow^* 0S1; S \Rightarrow^* 00S11; S \Rightarrow^*$
 $000111.$

Sentential Forms

- Any string of variables and/or terminals derived from the start symbol is called a *sentential form*.
- Formally, α is a sentential form iff $S \Rightarrow^* \alpha$.

Language of a Grammar

- If G is a CFG, then $L(G)$, the *language of G*, is $\{w \mid S \Rightarrow^* w\}$.
 - Note: w must be a terminal string, S is the start symbol.
- Example: G has productions $S \rightarrow \epsilon$ and $S \rightarrow 0S1$.
- $L(G) = \{0^n 1^n \mid n \geq 0\}$.

↑
Note: ϵ is a legitimate right side.

Context-Free Languages

- A language that is defined by some CFG is called a *context-free language*.
- There are CFL's that are not regular languages, such as the example just given.
- But not all languages are CFL's.
- Intuitively: CFL's can count two things, not three.

BNF Notation

- Grammars for programming languages are often written in BNF (*Backus-Naur Form*).
- Variables are words in <...>; Example: <statement>.
- Terminals are often multicharacter strings indicated by boldface or underline; Example: **while** or WHILE.

BNF Notation – (2)

- Symbol $::=$ is often used for \rightarrow .
- Symbol $|$ is used for “or.”
 - A shorthand for a list of productions with the same left side.
- Example: $S \rightarrow 0S1 | 01$ is shorthand for $S \rightarrow 0S1$ and $S \rightarrow 01$.

BNF Notation – Kleene Closure

- Symbol ... is used for “one or more.”
- Example: $\langle \text{digit} \rangle ::= 0|1|2|3|4|5|6|7|8|9$
 $\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \dots$
 - Note: that’s not exactly the * of RE’s.
- Translation: Replace $\alpha\dots$ with a new variable A and productions $A \rightarrow A\alpha \mid \alpha$.

Example: Kleene Closure

- Grammar for unsigned integers can be replaced by:

$$U \rightarrow UD \mid D$$
$$D \rightarrow 0|1|2|3|4|5|6|7|8|9$$

BNF Notation: Optional Elements

- Surround one or more symbols by [...] to make them optional.
- Example: $\langle \text{statement} \rangle ::= \text{if } \langle \text{condition} \rangle \text{ then } \langle \text{statement} \rangle [; \text{else } \langle \text{statement} \rangle]$
- Translation: replace $[\alpha]$ by a new variable A with productions $A \rightarrow \alpha \mid \varepsilon$.

Example: Optional Elements

- Grammar for if-then-else can be replaced by:

$$S \rightarrow iCtSA$$
$$A \rightarrow ;eS \mid \epsilon$$

BNF Notation – Grouping

- Use $\{\dots\}$ to surround a sequence of symbols that need to be treated as a unit.
 - Typically, they are followed by a \dots for “one or more.”
- Example: $\langle \text{statement list} \rangle ::= \langle \text{statement} \rangle [\{\text{;} \langle \text{statement} \rangle\} \dots]$

Translation: Grouping

- You may, if you wish, create a new variable A for $\{\alpha\}$.
- One production for A: $A \rightarrow \alpha$.
- Use A in place of $\{\alpha\}$.

Example: Grouping

$L \rightarrow S [\{;S\} \dots]$

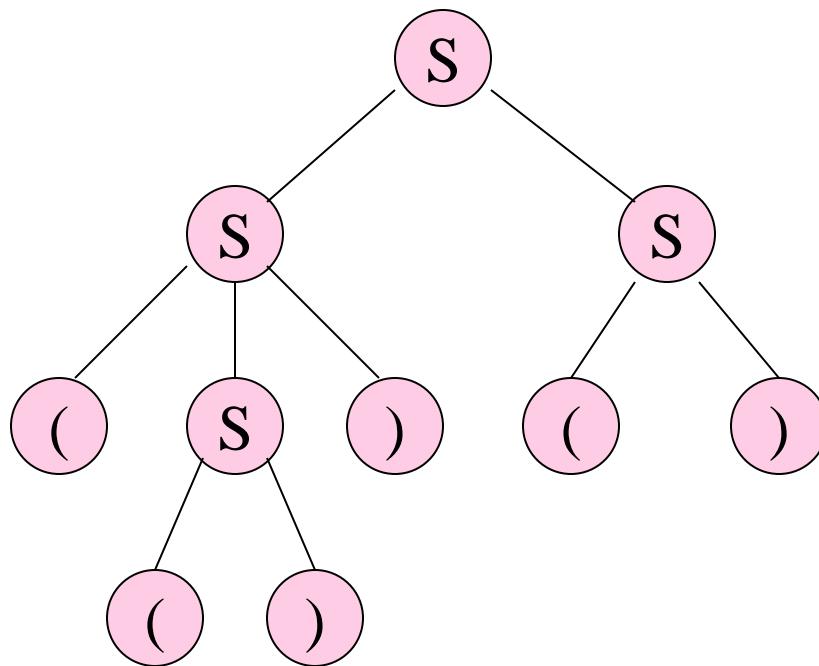
- Replace by $L \rightarrow S [A\dots] \quad A \rightarrow ;S$
 - A stands for $\{;S\}$.
- Then by $L \rightarrow SB \quad B \rightarrow A\dots \mid \epsilon \quad A \rightarrow ;S$
 - B stands for $[A\dots]$ (zero or more A's).
- Finally by $L \rightarrow SB \quad B \rightarrow C \mid \epsilon \quad C \rightarrow AC \mid A \quad A \rightarrow ;S$
 - C stands for $A\dots$.

Parse Trees

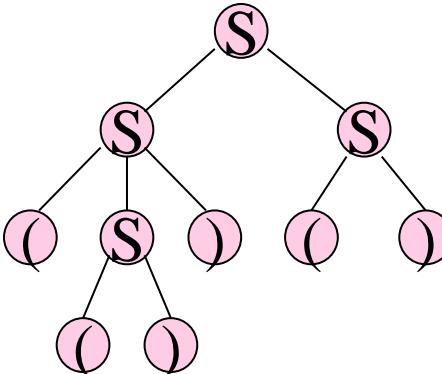
- *Parse trees* are trees labeled by symbols of a particular CFG.
- Leaves: labeled by a terminal or ϵ .
- Interior nodes: labeled by a variable.
 - Children are labeled by the right side of a production for the parent.
- Root: must be labeled by the start symbol.

Example: Parse Tree

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



Yield of a Parse Tree

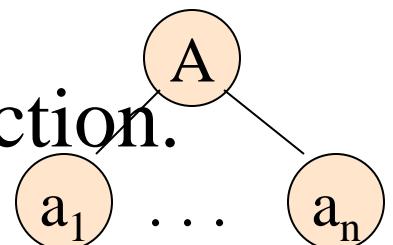
- The concatenation of the labels of the leaves in left-to-right order
 - That is, in the order of a preorder traversal.is called the *yield* of the parse tree.
- Example: yield of  is $((())()$

Parse Trees, Leftmost and Rightmost Derivations

- For every parse tree, there is a unique leftmost, and a unique rightmost derivation.
- We'll prove:
 1. If there is a parse tree with root labeled A and yield w, then $A \Rightarrow^*_{\text{lm}} w$.
 2. If $A \Rightarrow^*_{\text{lm}} w$, then there is a parse tree with root A and yield w.

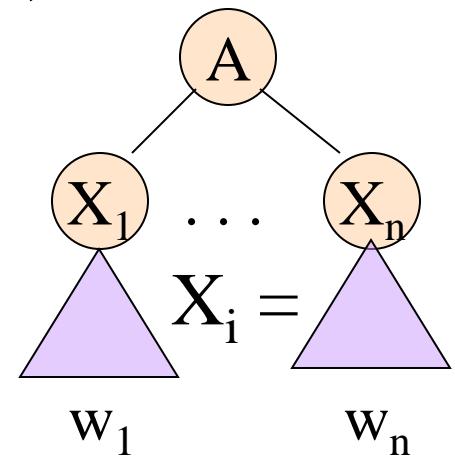
Proof – Part 1

- Induction on the *height* (length of the longest path from the root) of the tree.
- Basis: height 1. Tree looks like
- $A \rightarrow a_1 \dots a_n$ must be a production.
- Thus, $A \Rightarrow^*_{\text{lm}} a_1 \dots a_n$.



Part 1 – Induction

- Assume (1) for trees of height $< h$, and let this tree have height h :
- By IH, $X_i \Rightarrow_{lm}^* w_i$.
 - Note: if X_i is a terminal, then w_i .
- Thus, $A \Rightarrow_{lm} X_1 \dots X_n \Rightarrow_{lm}^* w_1 X_2 \dots X_n \Rightarrow_{lm}^* w_1 w_2 X_3 \dots X_n \Rightarrow_{lm}^* \dots \Rightarrow_{lm}^* w_1 \dots w_n$.

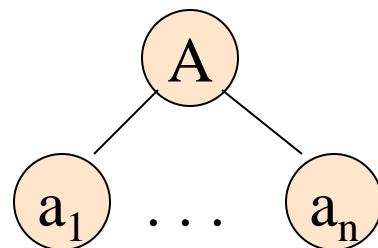


Proof: Part 2

- Given a leftmost derivation of a terminal string, we need to prove the existence of a parse tree.
- The proof is an induction on the length of the derivation.

Part 2 – Basis

- If $A \Rightarrow^*_{lm} a_1 \dots a_n$ by a one-step derivation, then there must be a parse tree

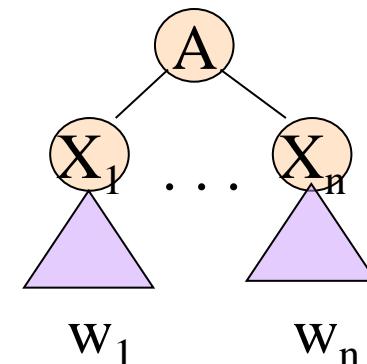


Part 2 – Induction

- Assume (2) for derivations of fewer than $k > 1$ steps, and let $A \Rightarrow_{lm}^* w$ be a k -step derivation.
- First step is $A \Rightarrow_{lm} X_1 \dots X_n$.
- Key point: w can be divided so the first portion is derived from X_1 , the next is derived from X_2 , and so on.
 - If X_i is a terminal, then $w_i = X_i$.

Induction – (2)

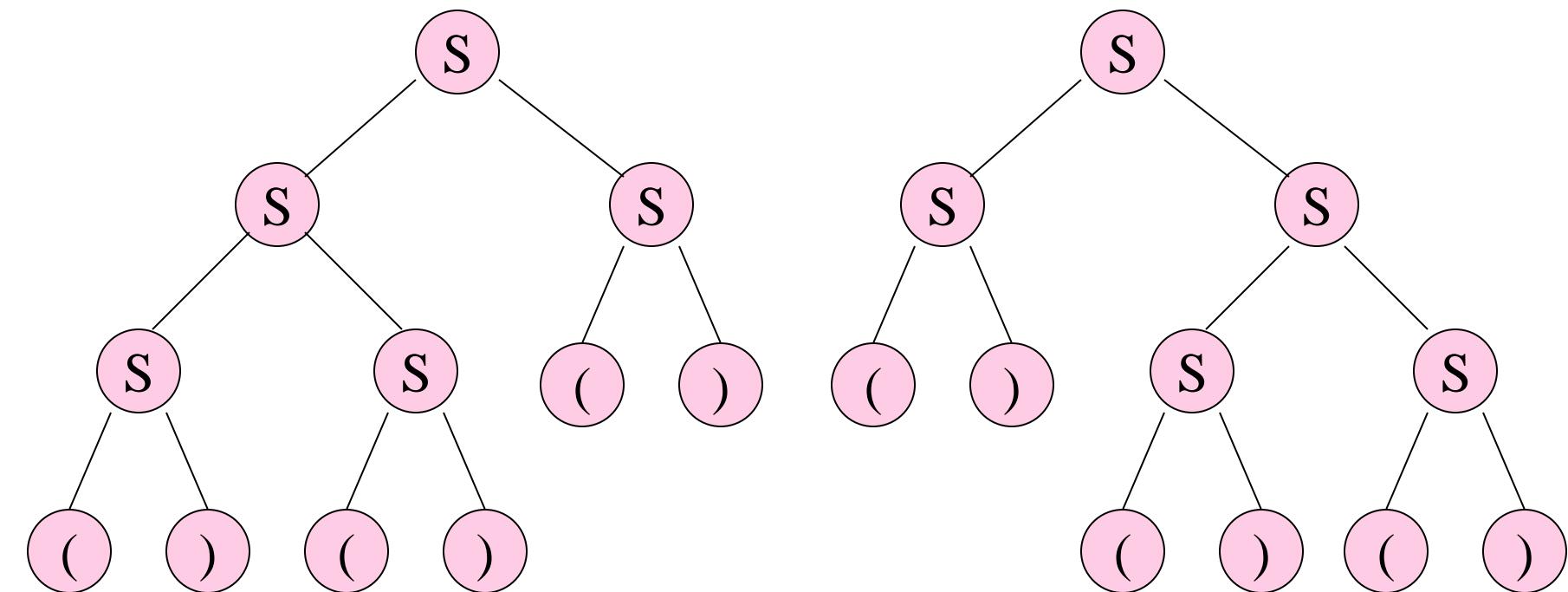
- That is, $X_i \Rightarrow^*_{\text{lm}} w_i$ for all i such that X_i is a variable.
 - And the derivation takes fewer than k steps.
- By the IH, if X_i is a variable, then there is a parse tree with root X_i and yield w_i .
- Thus, there is a parse tree



Ambiguous Grammars

- A CFG is *ambiguous* if there is a string in the language that is the yield of two or more parse trees.
- Example: $S \rightarrow SS \mid (S) \mid ()$
- Two parse trees for $()()()$ on next slide.

Example – Continued



Ambiguity, Left- and Rightmost Derivations

- If there are two different parse trees, they must produce two different leftmost derivations by the construction given in the proof.
- Conversely, two different leftmost derivations produce different parse trees by the other part of the proof.
- Likewise for rightmost derivations.

Ambiguity, etc. – (2)

- Thus, equivalent definitions of “ambiguous grammar” are:
 1. There is a string in the language that has two different leftmost derivations.
 2. There is a string in the language that has two different rightmost derivations.

Ambiguity

- Ambiguity is a Property of Grammars, not Languages
- For the balanced-parentheses language, here is another CFG, which is unambiguous.

$$B \xrightarrow{-} (RB \mid \epsilon)$$

$$R \rightarrow) \mid (RR$$

B, the start symbol,
derives balanced strings.

R generates strings that
have one more right paren
than left.

Example: Unambiguous Grammar

$$B \rightarrow (RB \mid \epsilon \quad R \rightarrow) \mid (RR$$

- Construct a unique leftmost derivation for a given balanced string of parentheses by scanning the string from left to right.
 - If we need to expand B, then use $B \rightarrow (RB$ if the next symbol is "(" and ϵ if at the end.
 - If we need to expand R, use $R \rightarrow)$ if the next symbol is ")" and $(RR$ if it is "(".

Leftmost Derivations

- Say $wA\alpha \Rightarrow_{lm} w\beta\alpha$ if w is a string of terminals only and $A \rightarrow \beta$ is a production.
- Also, $\alpha \Rightarrow_{lm}^* \beta$ if α becomes β by a sequence of 0 or more \Rightarrow_{lm} steps.

Rightmost Derivations

- Say $\alpha A w \Rightarrow_{rm} \alpha \beta w$ if w is a string of terminals only and $A \rightarrow \beta$ is a production.
- Also, $\alpha \Rightarrow_{rm}^* \beta$ if α becomes β by a sequence of 0 or more \Rightarrow_{rm} steps.

Ambiguity

- In general, we try to eliminate ambiguity by rewriting the grammar.
- Example:
 - $S \rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{if } E \text{ then } S \mid \text{other}$
becomes:

$$S \rightarrow E_{\text{withElse}} \mid E_{\text{noElse}}$$

$$E_{\text{withElse}} \rightarrow \text{if } E \text{ then } E_{\text{withElse}} \text{ else } E_{\text{withElse}} \mid \text{other}$$

$$E_{\text{noElse}} \rightarrow \text{if } E \text{ then } S$$

$$\mid \text{if } E \text{ then } E_{\text{withElse}} \text{ else } E_{\text{noElse}}$$

Grammar problems

- Because we try to generate a leftmost derivation by scanning the input from left to right, grammars of the form $A \rightarrow A x$ may cause endless recursion.
- Such grammars are called **left-recursive** and they must be transformed if we want to use a top-down parser.

Left recursion

- A grammar is left recursive if for a non-terminal A , there is a derivation $A \Rightarrow^+ A\alpha$
- There are three types of left recursion:
 - direct ($A \rightarrow A x$)
 - indirect ($A \rightarrow B C, B \rightarrow A$)
 - hidden ($A \rightarrow B A, B \rightarrow \varepsilon$)

Left recursion

- To eliminate direct left recursion replace

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

with

$$A \rightarrow \beta_1 B \mid \beta_2 B \mid \dots \mid \beta_n B$$

$$B \rightarrow \alpha_1 B \mid \alpha_2 B \mid \dots \mid \alpha_m B \mid \varepsilon$$

Left recursion

- How about this:

$S \rightarrow E$
 $E \rightarrow E+T$
 $E \rightarrow T$
 $T \rightarrow E-T$
 $T \rightarrow \text{id}$

There is direct recursion: $E \rightarrow E+T$

There is indirect recursion: $T \rightarrow E+T$, $E \rightarrow T$

Algorithm for eliminating indirect recursion

List the nonterminals in some order A_1, A_2, \dots, A_n
for $i=1$ to n

 for $j=1$ to $i-1$

 if there is a production $A_i \rightarrow A_j \alpha$,

 replace A_j with its rhs

 eliminate any direct left recursion on A_i

Eliminating indirect left recursion

ordering: S, E, T, F

$S \rightarrow E$
 $E \rightarrow E+T$
 $E \rightarrow T$
 $T \rightarrow E-T$
 $T \rightarrow F$
 $F \rightarrow E^*F$
 $F \rightarrow \text{id}$

i=S

$S \rightarrow E$
 $E \rightarrow E+T$
 $E \rightarrow T$
 $T \rightarrow E-T$
 $T \rightarrow F$
 $F \rightarrow E^*F$
 $F \rightarrow \text{id}$

i=E

$S \rightarrow E$
 $E \rightarrow TE'$
 $E' \rightarrow +TE'|\varepsilon$
 $T \rightarrow E-T$
 $T \rightarrow F$
 $F \rightarrow E^*F$
 $F \rightarrow \text{id}$

i=T, j=E

$S \rightarrow E$
 $E \rightarrow TE'$
 $E' \rightarrow +TE'|\varepsilon$
 $T \rightarrow TE'-T$
 $T \rightarrow F$
 $F \rightarrow E^*F$
 $F \rightarrow \text{id}$



$S \rightarrow E$
 $E \rightarrow TE'$
 $E' \rightarrow +TE'|\varepsilon$
 $T \rightarrow FT'$
 $T' \rightarrow E'-TT'|\varepsilon$
 $F \rightarrow E^*F$
 $F \rightarrow \text{id}$

Eliminating indirect left recursion

i=F, j=E

i=F, j=T

$S \rightarrow E$
 $E \rightarrow TE'$
 $E' \rightarrow +TE'|\epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow E'-TT'|\epsilon$
 $F \rightarrow TE'^*F$
 $F \rightarrow \mathbf{id}$

$S \rightarrow E$
 $E \rightarrow TE'$
 $E' \rightarrow +TE'|\epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow E'-TT'|\epsilon$
 $F \rightarrow FT'E'^*F$
 $F \rightarrow \mathbf{id}$



$S \rightarrow E$
 $E \rightarrow TE'$
 $E' \rightarrow +TE'|\epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow E'-TT'|\epsilon$
 $F \rightarrow \mathbf{id}F'$
 $F' \rightarrow T'E'^*FF'|\epsilon$

Grammar problems

- Consider $S \rightarrow \mathbf{if}\ E\ \mathbf{then}\ S\ \mathbf{else}\ S\ |\ \mathbf{if}\ E\ \mathbf{then}\ S$
 - Which of the two productions should we use to expand non-terminal S when the next token is **if**?
 - We can solve this problem by factoring out the common part in these rules. This way, we are postponing the decision about which rule to choose until we have more information (namely, whether there is an **else** or not).
 - This is called **left factoring**

Left factoring

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$$

becomes

$$A \rightarrow \alpha B \mid \gamma$$

$$B \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

Normal Forms for CFG's

Eliminating Useless Variables

Removing Epsilon

Removing Unit Productions

Chomsky Normal Form

Testing Whether a Variable Derives Some Terminal String

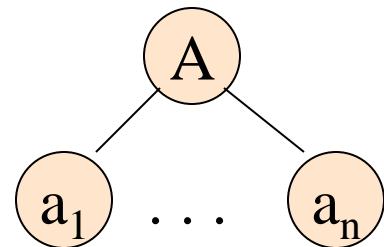
- Basis: If there is a production $A \rightarrow w$, where w has no variables, then A derives a terminal string.
- Induction: If there is a production $A \rightarrow \alpha$, where α consists only of terminals and variables known to derive a terminal string, then A derives a terminal string.

Testing – (2)

- Eventually, we can find no more variables.
- An easy induction on the order in which variables are discovered shows that each one truly derives a terminal string.
- Conversely, any variable that derives a terminal string will be discovered by this algorithm.

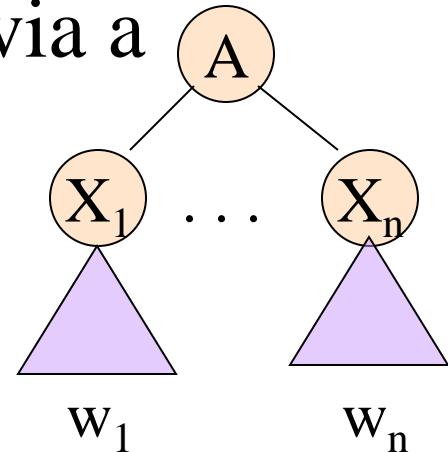
Proof of Converse

- The proof is an induction on the height of the least-height parse tree by which a variable A derives a terminal string.
- Basis: Height = 1. Tree looks like:
- Then the basis of the algorithm tells us that A will be discovered.



Induction for Converse

- Assume IH for parse trees of height $< h$, and suppose A derives a terminal string via a parse tree of height h :
- By IH, those X_i 's that are variables are discovered.
- Thus, A will also be discovered, because it has a right side of terminals and/or discovered variables.



Algorithm to Eliminate Variables That Derive Nothing

1. Discover all variables that derive terminal strings.
2. For all other variables, remove all productions in which they appear either on the left or the right.

Example: Eliminate Variables

$S \rightarrow AB \mid C, A \rightarrow aA \mid a, B \rightarrow bB, C \rightarrow c$

- Basis: A and C are identified because of $A \rightarrow a$ and $C \rightarrow c$.
- Induction: S is identified because of $S \rightarrow C$.
- Nothing else can be identified.
- Result: $S \rightarrow C, A \rightarrow aA \mid a, C \rightarrow c$

Unreachable Symbols

- Another way a terminal or variable deserves to be eliminated is if it cannot appear in any derivation from the start symbol.
- Basis: We can reach S (the start symbol).
- Induction: if we can reach A , and there is a production $A \rightarrow \alpha$, then we can reach all symbols of α .

Unreachable Symbols – (2)

- Easy inductions in both directions show that when we can discover no more symbols, then we have all and only the symbols that appear in derivations from S .
- Algorithm: Remove from the grammar all symbols not discovered reachable from S and all productions that involve these symbols.

Eliminating Useless Symbols

- A symbol is *useful* if it appears in some derivation of some terminal string from the start symbol.
- Otherwise, it is *useless*.

Eliminate all useless symbols by:

1. Eliminate symbols that derive no terminal string.
2. Eliminate unreachable symbols.

Example: Useless Symbols – (2)

$S \rightarrow AB, A \rightarrow C, C \rightarrow c, B \rightarrow bB$

- If we eliminated unreachable symbols first, we would find everything is reachable.
- A, C, and c would never get eliminated.

Epsilon Productions

- We can almost avoid using productions of the form $A \rightarrow \epsilon$ (called *ϵ -productions*).
 - The problem is that ϵ cannot be in the language of any grammar that has no ϵ –productions.
- Theorem: If L is a CFL, then $L - \{\epsilon\}$ has a CFG with no ϵ -productions.

Nullable Symbols

- To eliminate ϵ -productions, we first need to discover the *nullable variables* = variables A such that $A \Rightarrow^* \epsilon$.
- Basis: If there is a production $A \rightarrow \epsilon$, then A is nullable.
- Induction: If there is a production $A \rightarrow \alpha$, and all symbols of α are nullable, then A is nullable.

Example: Nullable Symbols

$S \rightarrow AB, A \rightarrow aA \mid \epsilon, B \rightarrow bB \mid A$

- Basis: A is nullable because of $A \rightarrow \epsilon$.
- Induction: B is nullable because of $B \rightarrow A$.
- Then, S is nullable because of $S \rightarrow AB$.

Eliminating ϵ -Productions

- Key idea: turn each production
- $A \rightarrow X_1\dots X_n$ into a family of productions.
- For each subset of nullable X 's, there is one production with those eliminated from the right side “in advance.”
 - Except, if all X 's are nullable, do not make a production with ϵ as the right side.

Example: Eliminating ϵ -Productions

$S \rightarrow ABC, A \rightarrow aA \mid \epsilon, B \rightarrow bB \mid \epsilon, C \rightarrow \epsilon$

- A, B, C, and S are all nullable.
- New grammar:

$S \rightarrow \cancel{ABC} \mid AB \mid \cancel{AC} \mid \cancel{BC} \mid \cancel{A} \mid B \mid C \cancel{\times}$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

Note: C is now useless.
Eliminate its productions.

Unit Productions

- A *unit production* is one whose right side consists of exactly one variable.
- These productions can be eliminated.
- Key idea: If $A \Rightarrow^* B$ by a series of unit productions, and $B \rightarrow \alpha$ is a non-unit-production, then add production $A \rightarrow \alpha$.
- Then, drop all unit productions.

Unit Productions – (2)

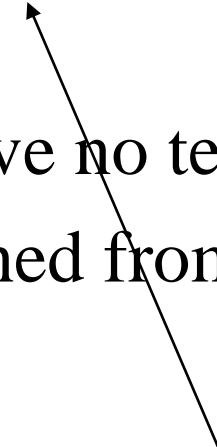
- Find all pairs (A, B) such that $A \Rightarrow^* B$ by a sequence of unit productions only.
- Basis: Surely (A, A) .
- Induction: If we have found (A, B) , and $B \rightarrow C$ is a unit production, then add (A, C) .

Cleaning Up a Grammar

- Theorem: if L is a CFL, then there is a CFG for $L - \{\epsilon\}$ that has:
 1. No useless symbols.
 2. No ϵ -productions.
 3. No unit productions.
- I.e., every right side is either a single terminal or has length ≥ 2 .

Cleaning Up – (2)

- Proof: Start with a CFG for L.
- Perform the following steps in order:
 1. Eliminate ϵ -productions.
 2. Eliminate unit productions.
 3. Eliminate variables that derive no terminal string.
 4. Eliminate variables not reached from the start symbol.



Must be first. Can create unit productions or useless variables.

Chomsky Normal Form

- A CFG is said to be in *Chomsky Normal Form* if every production is of one of these two forms:
 1. $A \rightarrow BC$ (right side is two variables).
 2. $A \rightarrow a$ (right side is a single terminal).
- Theorem: If L is a CFL, then $L - \{\epsilon\}$ has a CFG in CNF.

Proof of CNF Theorem

- Step 1: “Clean” the grammar, so every production right side is either a single terminal or of length at least 2.
- Step 2: For each right side \neq a single terminal, make the right side all variables.
 - For each terminal a create new variable A_a and production $A_a \rightarrow a$.
 - Replace a by A_a in right sides of length > 2 .

Example: Step 2

- Consider production $A \rightarrow BcDe$.
- We need variables A_c and A_e . with productions $A_c \rightarrow c$ and $A_e \rightarrow e$.
 - Note: you create at most one variable for each terminal, and use it everywhere it is needed.
- Replace $A \rightarrow BcDe$ by $A \rightarrow BA_cDA_e$.

CNF Proof – Continued

- Step 3: Break right sides longer than 2 into a chain of productions with right sides of two variables.
- Example: $A \rightarrow BCDE$ is replaced by $A \rightarrow BF$, $F \rightarrow CG$, and $G \rightarrow DE$.
 - F and G must be used nowhere else.

Example of Step 3 – Continued

- Recall $A \rightarrow BCDE$ is replaced by
- $A \rightarrow BF$, $F \rightarrow CG$, and $G \rightarrow DE$.
- In the new grammar, $A \Rightarrow BF \Rightarrow BCG \Rightarrow BCDE$.
- More importantly: Once we choose to replace A by BF , we must continue to BCG and $BCDE$.
 - Because F and G have only one production.

CNF Proof – Concluded

- We must prove that Steps 2 and 3 produce new grammars whose languages are the same as the previous grammar.
- Proofs are of a familiar type and involve inductions on the lengths of derivations.

The Pumping Lemma for CFL's

Statement
Applications

Statement of the CFL Pumping Lemma

For every context-free language L

There is an integer n , such that

For every string z in L of length $\geq n$

There exists $z = uvwxy$ such that:

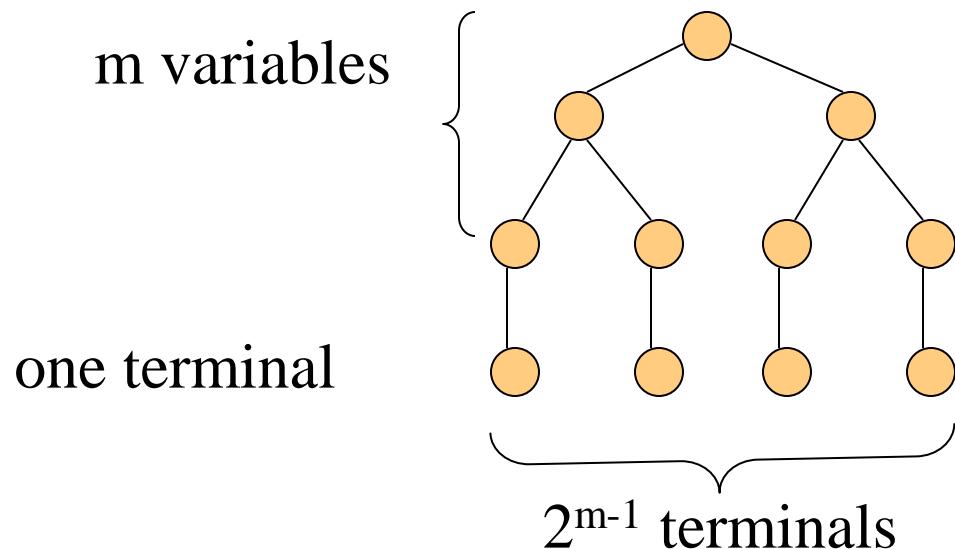
1. $|vwx| \leq n$.
2. $|vx| > 0$.
3. For all $i \geq 0$, $uv^iwx^i y$ is in L .

Proof of the Pumping Lemma

- Start with a CNF grammar for $L - \{\epsilon\}$.
- Let the grammar have m variables.
- Pick $n = 2^m$.
- Let $|z| \geq n$.
- We claim (“*Lemma 1*”) that a parse tree with yield z must have a path of length $m+2$ or more.

Proof of Lemma 1

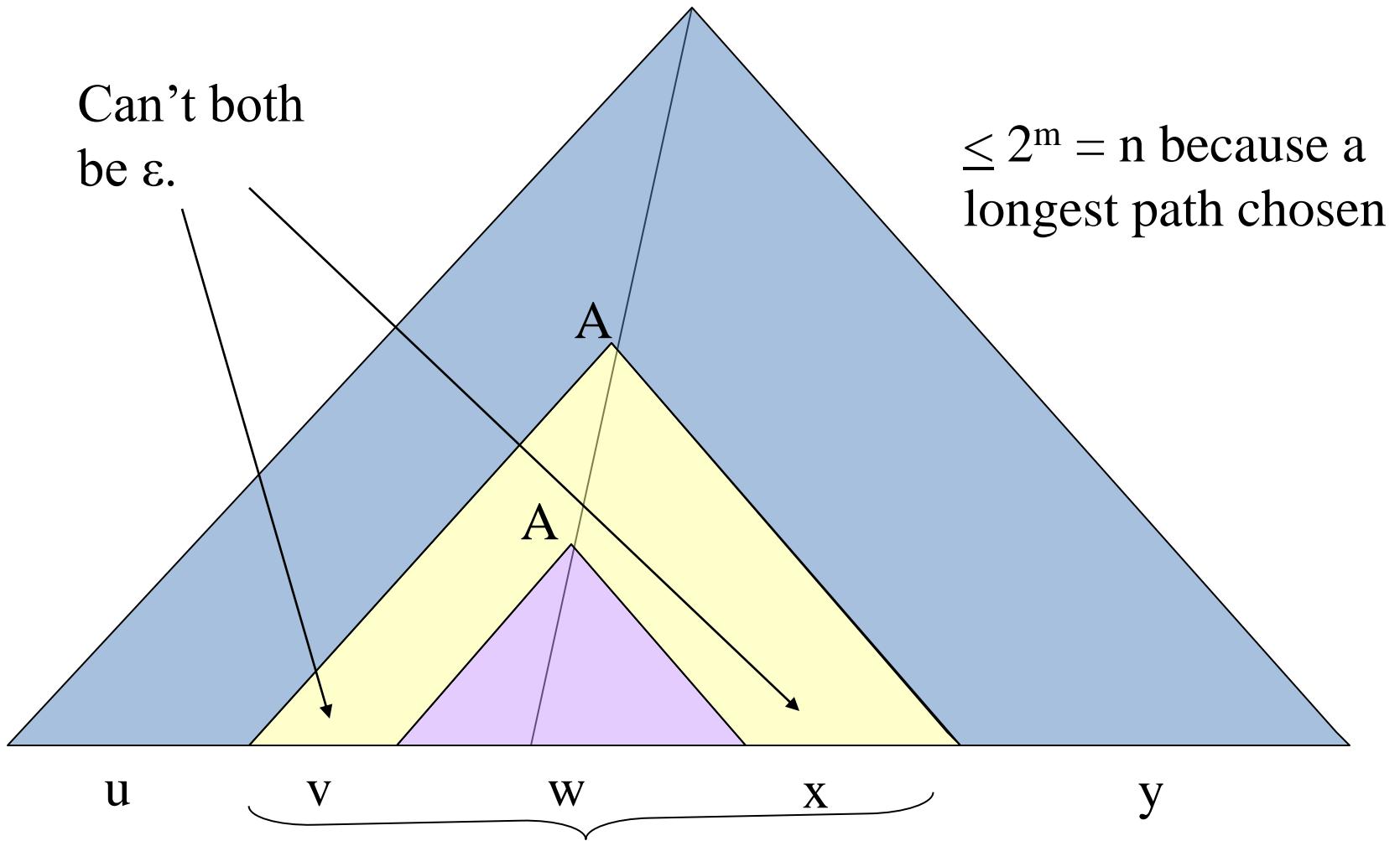
- If all paths in the parse tree of a CNF grammar are of length $\leq m+1$, then the longest yield has length 2^{m-1} , as in:



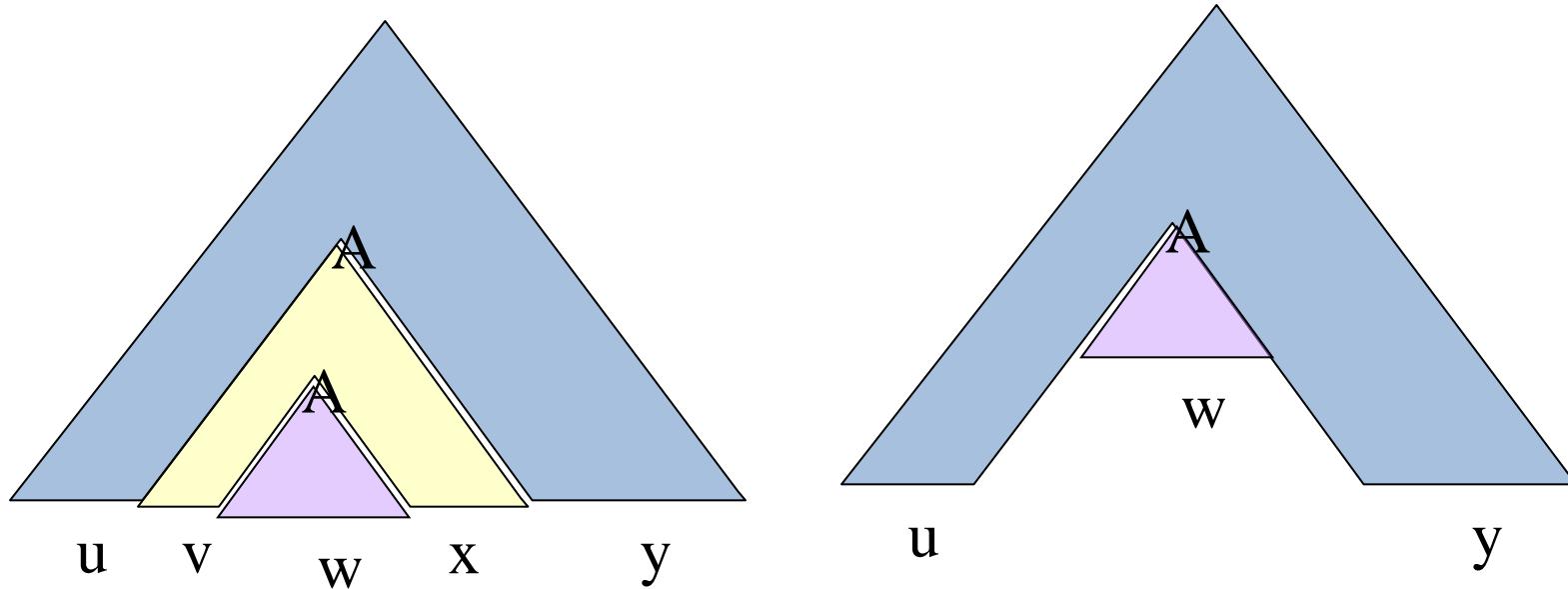
Back to the Proof of the Pumping Lemma

- Now we know that the parse tree for z has a path with at least $m+1$ variables.
- Consider some longest path.
- There are only m different variables, so among the lowest $m+1$ we can find two nodes with the same label, say A .
- The parse tree thus looks like:

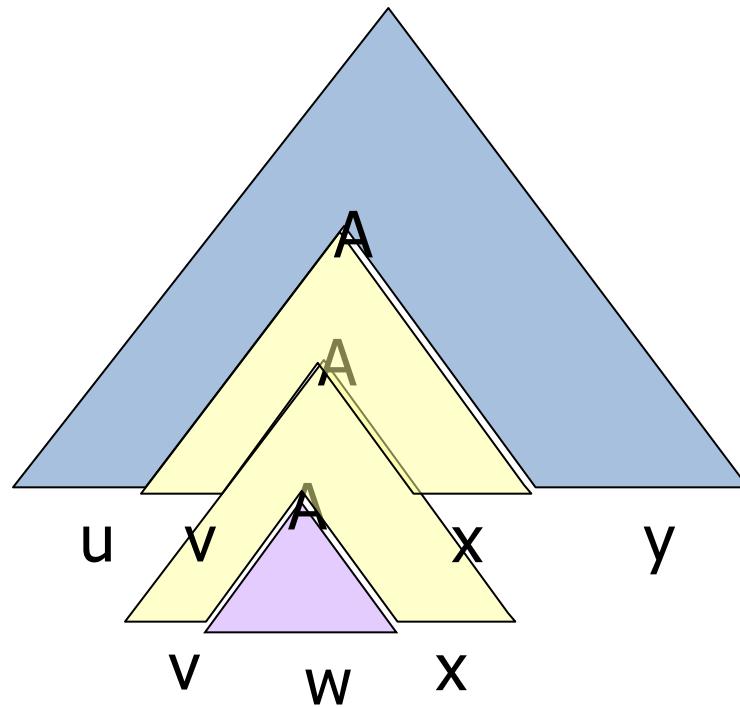
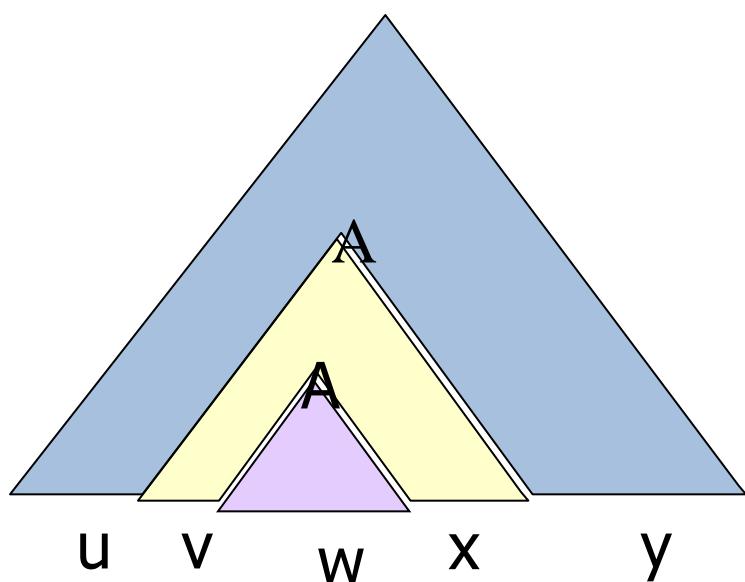
Parse Tree in the Pumping-Lemma Proof



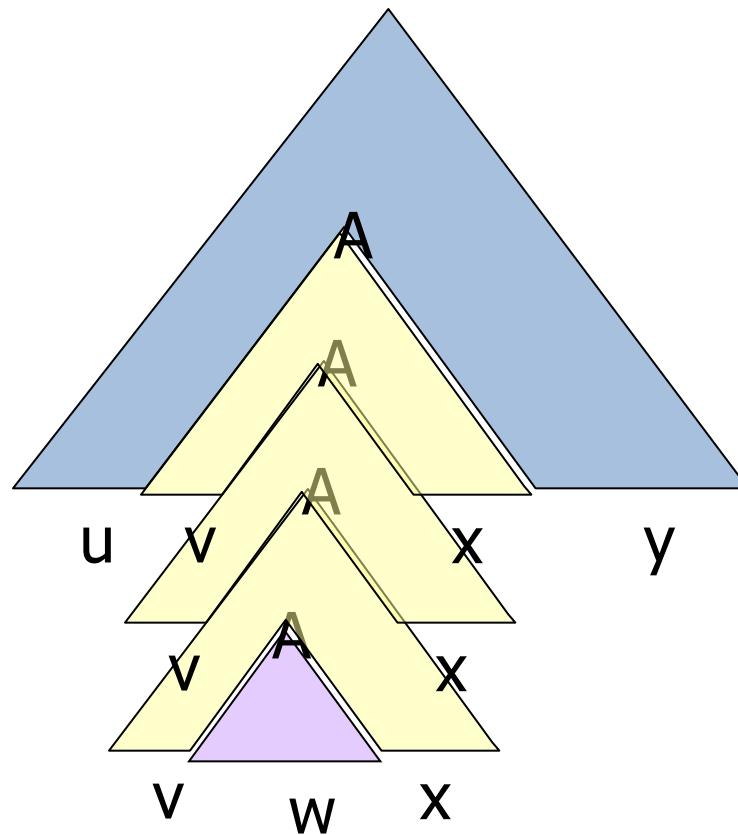
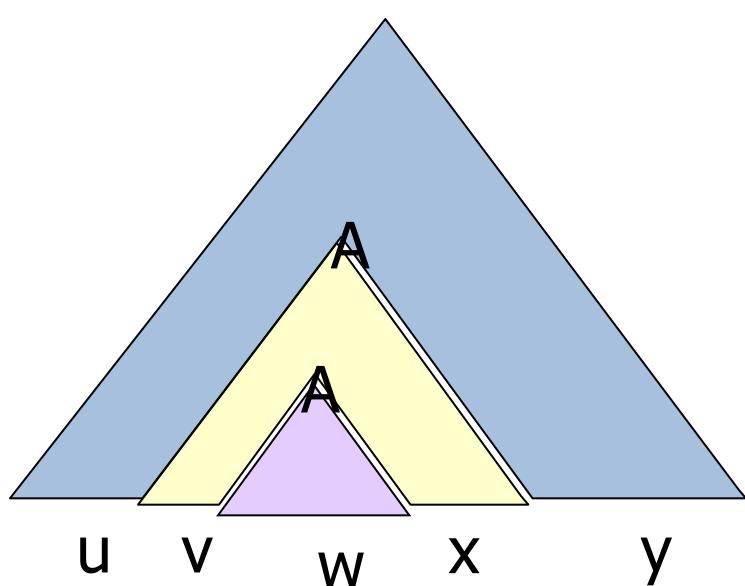
Pump Zero Times



Pump Twice



Pump Thrice Etc., Etc.



Using the Pumping Lemma

- Non-CFL's typically involve trying to match two pairs of counts or match two strings.
- **Example:** The text uses the pumping lemma to show that $\{ww \mid w \text{ in } (0+1)^*\}$ is not a CFL.

Using the Pumping Lemma – (2)

- $\{0^i 1 0^i \mid i \geq 1\}$ is a CFL.
 - We can match one pair of counts.
- But $L = \{0^i 1 0^i 1 0^i \mid i \geq 1\}$ is not.
 - We can't match two pairs, or three counts as a group.
- Proof using the pumping lemma.
- Suppose L were a CFL.
- Let n be L 's pumping-lemma constant.

Using the Pumping Lemma – (3)

- Consider $z = 0^n 1 0^n 1 0^n$.
- We can write $z = uvwxy$, where $|vwx| \leq n$, and $|vx| \geq 1$.
- Case 1: vx has no 0's.
 - Then at least one of them is a 1, and uwy has at most one 1, which no string in L does.

Using the Pumping Lemma – (4)

- Still considering $z = 0^n 1 0^n 1 0^n$.
- Case 2: vx has at least one 0.
 - vwx is too short ($\text{length } \leq n$) to extend to all three blocks of 0's in $0^n 1 0^n 1 0^n$.
 - Thus, uwy has at least one block of n 0's, and at least one block with fewer than n 0's.
 - Thus, uwy is not in L .

Decision Properties of CFLs

- As usual, when we talk about “a CFL” we really mean “a representation for the CFL, e.g., a CFG or a PDA accepting by final state or empty stack.
- There are algorithms to decide if:
 1. String w is in CFL L .
 2. CFL L is empty.
 3. CFL L is infinite.

Closure Properties of CFL's

- CFL's are closed under union, concatenation, and Kleene closure.
- Also, under reversal, homomorphisms and inverse homomorphisms.
- But not under intersection or difference.

Closure of CFL's Under Union

- Let L and M be CFL's with grammars G and H , respectively.
- Assume G and H have no variables in common.
 - Names of variables do not affect the language.
- Let S_1 and S_2 be the start symbols of G and H .

Closure Under Union – (2)

- Form a new grammar for $L \cup M$ by combining all the symbols and productions of G and H.
- Then, add a new start symbol S.
- Add productions $S \rightarrow S_1 \mid S_2$.

Closure Under Union – (3)

- In the new grammar, all derivations start with S.
- The first step replaces S by either S_1 or S_2 .
- In the first case, the result must be a string in $L(G) = L$, and in the second case a string in $L(H) = M$.

Closure of CFL's Under Concatenation

- Let L and M be CFL's with grammars G and H , respectively.
- Assume G and H have no variables in common.
- Let S_1 and S_2 be the start symbols of G and H .

Closure Under Concatenation – (2)

- Form a new grammar for LM by starting with all symbols and productions of G and H .
- Add a new start symbol S .
- Add production $S \rightarrow S_1S_2$.
- Every derivation from S results in a string in L followed by one in M .

Closure Under Star

- Let L have grammar G , with start symbol S_1 .
- Form a new grammar for L^* by introducing to G a new start symbol S and the productions $S \rightarrow S_1S \mid \epsilon$.
- A rightmost derivation from S generates a sequence of zero or more S_1 's, each of which generates some string in L .

Closure of CFL's Under Reversal

- If L is a CFL with grammar G , form a grammar for L^R by reversing the right side of every production.
- **Example:** Let G have $S \rightarrow 0S1 \mid 01$.
- The reversal of $L(G)$ has grammar
- $S \rightarrow 1S0 \mid 10$.

Closure of CFL's Under Homomorphism

- Let L be a CFL with grammar G .
- Let h be a homomorphism on the terminal symbols of G .
- Construct a grammar for $h(L)$ by replacing each terminal symbol a by $h(a)$.

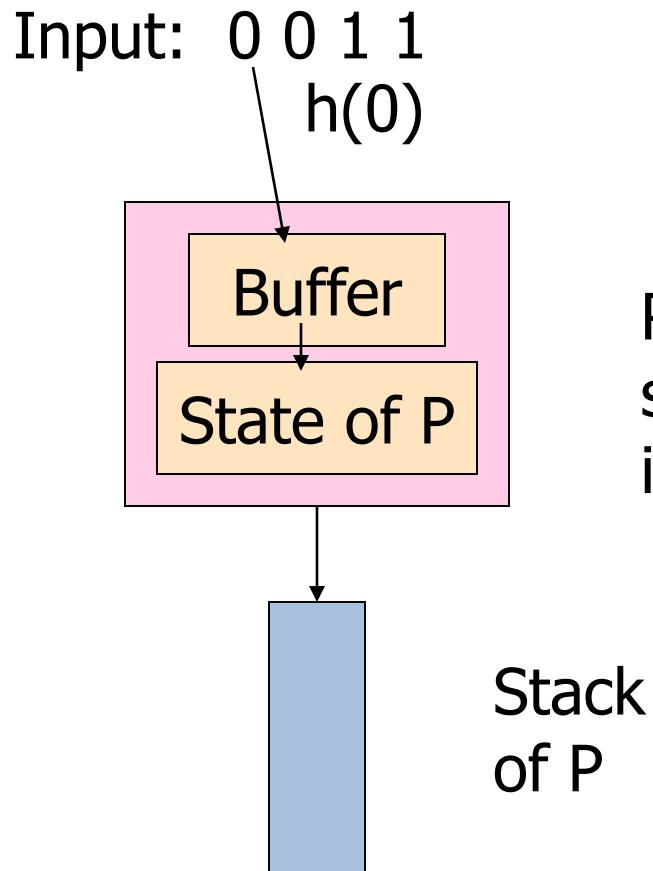
Example: Closure Under Homomorphism

- G has productions $S \rightarrow 0S1 \mid 01$.
- h is defined by $h(0) = ab$, $h(1) = \epsilon$.
- $h(L(G))$ has the grammar with productions $S \rightarrow abS \mid ab$.

Closure of CFL's Under Inverse Homomorphism

- Here, grammars don't help us.
- But a PDA construction serves nicely.
- **Intuition:** Let $L = L(P)$ for some PDA P .
- Construct PDA P' to accept $h^{-1}(L)$.
- P' simulates P , but keeps, as one component of a two-component state a buffer that holds the result of applying h to one input symbol.

Architecture of P'



Read first remaining symbol in buffer as if it were input to P.

Formal Construction of P'

- States are pairs $[q, b]$, where:
 1. q is a state of P .
 2. b is a suffix of $h(a)$ for some symbol a .
 - ◆ Thus, only a finite number of possible values for b .
- Stack symbols of P' are those of P .
- Start state of P' is $[q_0, \varepsilon]$.

Construction of P' – (2)

- Input symbols of P' are the symbols to which h applies.
- Final states of P' are the states $[q, \varepsilon]$ such that q is a final state of P .

Transitions of P'

1. $\delta'([q, \varepsilon], a, X) = \{([q, h(a)], X)\}$ for any input symbol a of P' and any stack symbol X .
 - ◆ When the buffer is empty, P' can reload it.
2. $\delta'([q, bw], \varepsilon, X)$ contains $([p, w], \alpha)$ if $\delta(q, b, X)$ contains (p, α) , where b is either an input symbol of P or ε .
 - ◆ Simulate P from the buffer.

Proving Correctness of P'

- We need to show that $L(P') = h^{-1}(L(P))$.
- **Key argument:** P' makes the transition $([q_0, \varepsilon], w, Z_0) \vdash^* ([q, x], \varepsilon, \alpha)$ if and only if P makes transition $(q_0, y, Z_0) \vdash^* (q, \varepsilon, \alpha)$, $h(w) = yx$, and x is a suffix of the last symbol of w .
- **Proof** in both directions is an induction on the number of moves made.

Nonclosure Under Intersection

- Unlike the regular languages, the class of CFL's is not closed under \cap .
- We know that $L_1 = \{0^n 1^n 2^n \mid n \geq 1\}$ is not a CFL (use the pumping lemma).
- However, $L_2 = \{0^n 1^n 2^i \mid n \geq 1, i \geq 1\}$ is.
 - CFG: $S \rightarrow AB, A \rightarrow 0A1 \mid 01, B \rightarrow 2B \mid 2$.
- So is $L_3 = \{0^i 1^n 2^n \mid n \geq 1, i \geq 1\}$.
- But $L_1 = L_2 \cap L_3$.

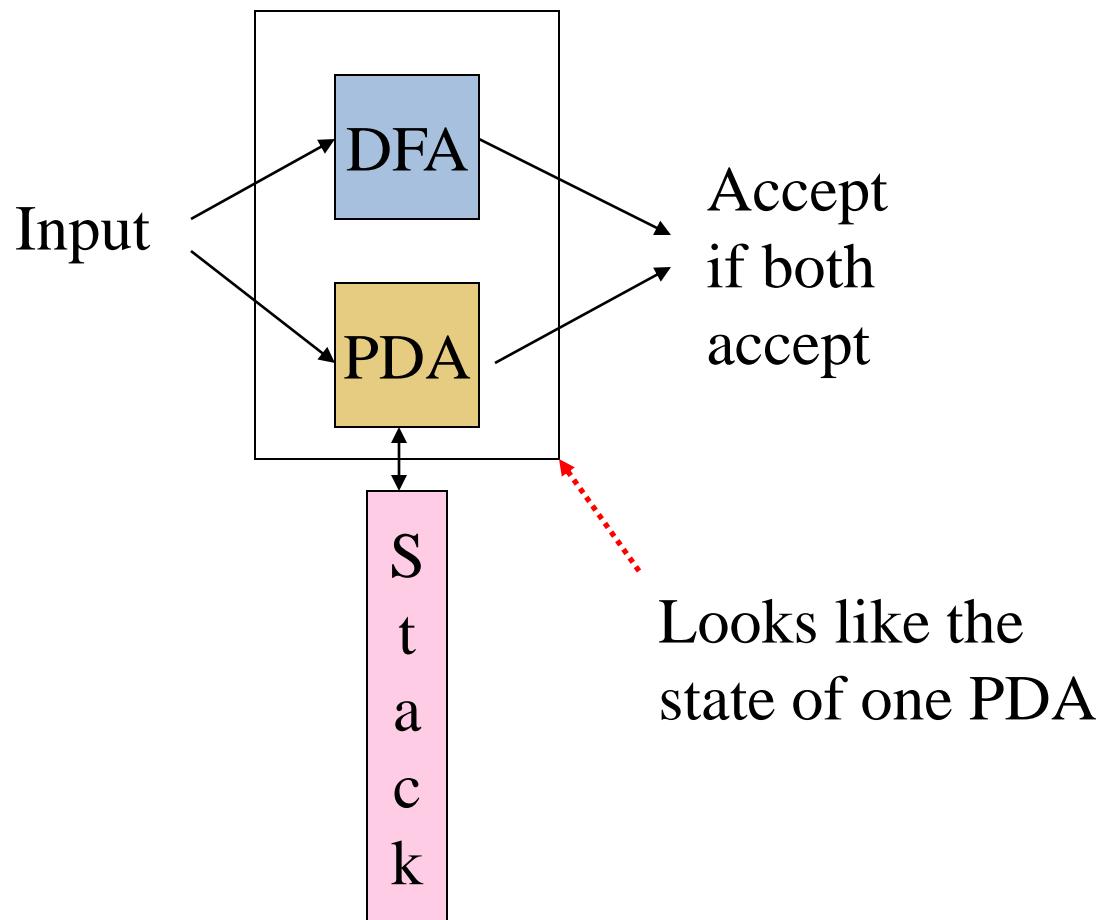
Nonclosure Under Difference

- We can prove something more general:
 - Any class of languages that is closed under difference is closed under intersection.
- Proof: $L \cap M = L - (L - M)$.
- Thus, if CFL's were closed under difference, they would be closed under intersection, but they are not.

Intersection with a Regular Language

- Intersection of two CFL's need not be context free.
- But the intersection of a CFL with a regular language is always a CFL.
- Proof involves running a DFA in parallel with a PDA, and noting that the combination is a PDA.
 - PDA's accept by final state.

DFA and PDA in Parallel



Formal Construction

- Let the DFA A have transition function δ_A .
- Let the PDA P have transition function δ_P .
- States of combined PDA are $[q,p]$, where q is a state of A and p a state of P.
- $\delta([q,p], a, X)$ contains $([\delta_A(q,a),r], \alpha)$ if $\delta_P(p, a, X)$ contains (r, α) .
 - Note a could be ε , in which case $\delta_A(q,a) = q$.

Formal Construction – (2)

- Accepting states of combined PDA are those $[q,p]$ such that q is an accepting state of A and p is an accepting state of P .
- Easy induction: $([q_0,p_0], w, Z_0) \vdash^* ([q,p], \varepsilon, \alpha)$ if and only if $\delta_A(q_0, w) = q$ and in P : $(p_0, w, Z_0) \vdash^*(p, \varepsilon, \alpha)$.

Unit – 4

Push Down Automata (PDA): The Formal Definition, Graphical notation, Instantaneous description, The Languages of a PDA, Equivalence of PDAs and CFGs, Deterministic Push Down Automata, Non-Deterministic Push Down Automata.

Pushdown Automata – Informal Definition

- The PDA is an automaton equivalent to the CFG in language-defining power.
- Only the nondeterministic PDA defines all the CFL's.
- But the deterministic version models parsers.
 - Most programming languages have deterministic PDA's.

PDA

- Think of an ϵ -NFA with the additional power that it can manipulate a stack.
- Its moves are determined by:
 1. The current state (of its “NFA”),
 2. The current input symbol (or ϵ), and
 3. The current symbol on top of its stack.

PDA – (2)

- Being nondeterministic, the PDA can have a choice of next moves.
- In each choice, the PDA can:
 1. Change state, and also
 2. Replace the top symbol on the stack by a sequence of zero or more symbols.
 - ◆ Zero symbols = “pop.”
 - ◆ Many symbols = sequence of “pushes.”

PDA Formalism

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

Conventions

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

The Transition Function

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

Actions of the PDA

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

Example: PDA

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

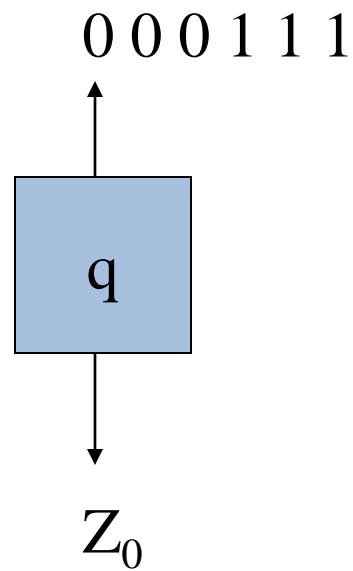
Example: PDA – (2)

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

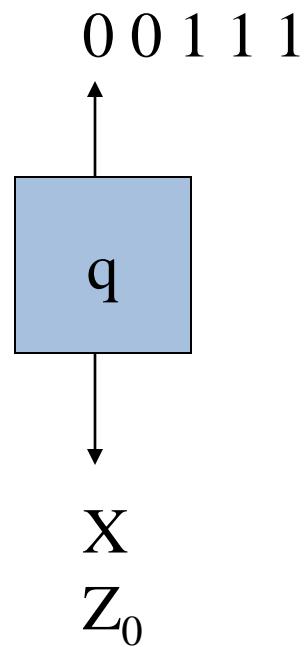
Example: PDA – (3)

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

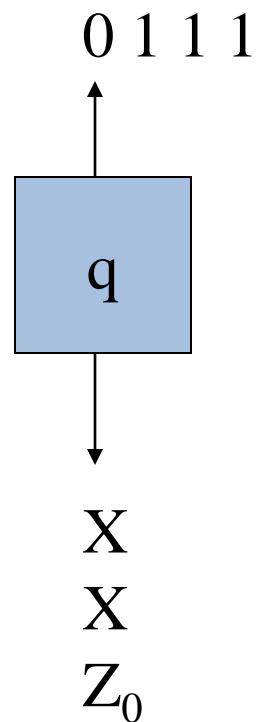
Actions of the Example PDA



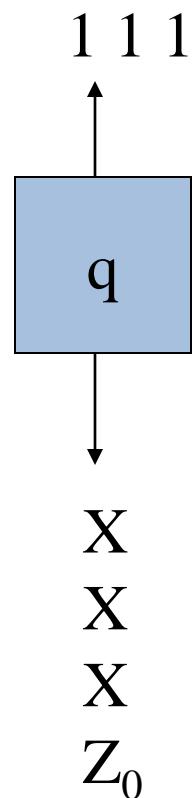
Actions of the Example PDA



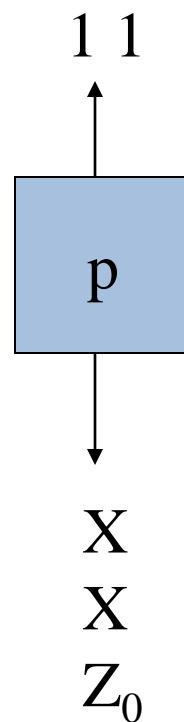
Actions of the Example PDA



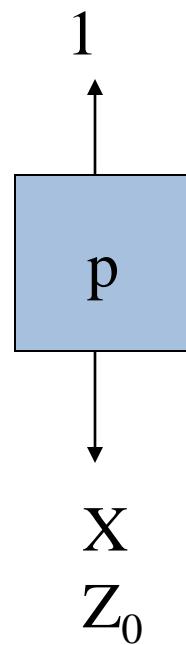
Actions of the Example PDA



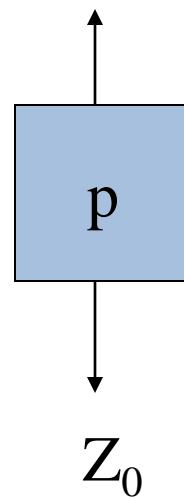
Actions of the Example PDA



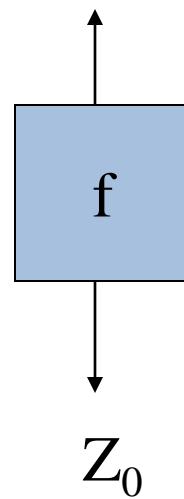
Actions of the Example PDA



Actions of the Example PDA



Actions of the Example PDA



Instantaneous Descriptions

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

The “Goes-To” Relation

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

Example: Goes-To

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

Answer

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

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

Aside: FA and PDA Notations

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

FA and PDA Notations – (2)

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

Language of a PDA

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

Language of a PDA – (2)

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

Equivalence of Language Definitions

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

Proof: $L(P) \rightarrow N(P')$ Intuition

- P' will simulate P .
- If P accepts, P' will empty its stack.
- P' has to avoid accidentally emptying its stack, so it uses a special bottom-marker to catch the case where P empties its stack without accepting.

Proof: $L(P) \rightarrow N(P')$

- P' has all the states, symbols, and moves of P , plus:
 1. Stack symbol X_0 , used to guard the stack bottom against accidental emptying.
 2. New start state s and “erase” state e .
 3. $\delta(s, \varepsilon, X_0) = \{(q_0, Z_0 X_0)\}$. Get P started.
 4. $\delta(f, \varepsilon, X) = \delta(e, \varepsilon, X) = \{(e, \varepsilon)\}$ for any final state f of P and any stack symbol X .

Proof: $N(P) \rightarrow L(P'')$ Intuition

- P'' simulates P .
- P'' has a special bottom-marker to catch the situation where P empties its stack.
- If so, P'' accepts.

Proof: $N(P) \rightarrow L(P'')$

- P'' has all the states, symbols, and moves of P , plus:
 1. Stack symbol X_0 , used to guard the stack bottom.
 2. New start state s and final state f .
 3. $\delta(s, \varepsilon, X_0) = \{(q_0, Z_0 X_0)\}$. Get P started.
 4. $\delta(q, \varepsilon, X_0) = \{(f, \varepsilon)\}$ for any state q of P .

Deterministic PDA's

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

Equivalence of PDA, CFG

Converting a CFG to a PDA

- Let $L = L(G)$.
- Construct PDA P such that $N(P) = L$.
- P has:
 - One state q .
 - Input symbols = terminals of G .
 - Stack symbols = all symbols of G .
 - Start symbol = start symbol of G .

(cont.,)

- Given input w , P will step through a leftmost derivation of w from the start symbol S .
- Since P can't know what this derivation is, or even what the end of w is, it uses nondeterminism to "guess" the production to use at each step.

(cont.,)

- At each step, P represents some *left-sentential form* (step of a leftmost derivation).
- If the stack of P is α , and P has so far consumed x from its input, then P represents left-sentential form $x\alpha$.
- At empty stack, the input consumed is a string in $L(G)$.

Transition Function of P

1. $\delta(q, a, a) = (q, \epsilon)$. (*Type 1* rules)
 - ◆ This step does not change the LSF represented, but “moves” responsibility for a from the stack to the consumed input.
2. If $A \rightarrow \alpha$ is a production of G , then $\delta(q, \epsilon, A)$ contains (q, α) . (*Type 2* rules)
 - ◆ Guess a production for A , and represent the next LSF in the derivation.

Proof That $L(P) = L(G)$

- We need to show that $(q, wx, S) \vdash^* (q, x, \alpha)$ for any x if and only if $S \Rightarrow^*_{lm} w\alpha$.
- **Part 1:** “only if” is an induction on the number of steps made by P .
- **Basis:** 0 steps.
 - Then $\alpha = S$, $w = \varepsilon$, and $S \Rightarrow^*_{lm} S$ is surely true.

Induction for Part 1

- Consider n moves of P : $(q, wx, S) \vdash^* (q, x, \alpha)$ and assume the IH for sequences of $n-1$ moves.
- There are two cases, depending on whether the last move uses a Type 1 or Type 2 rule.

Use of a Type 1 Rule

- The move sequence must be of the form $(q, yax, S) \vdash^* (q, ax, a\alpha) \vdash (q, x, \alpha)$, where $ya = w$.
- By the IH applied to the first $n-1$ steps, $S \Rightarrow_{lm}^* ya\alpha$.
- But $ya = w$, so $S \Rightarrow_{lm}^* wa\alpha$.

Use of a Type 2 Rule

- The move sequence must be of the form $(q, wx, S) \vdash^* (q, x, A\beta) \vdash (q, x, \gamma\beta)$, where $A \rightarrow \gamma$ is a production and $\alpha = \gamma\beta$.
- By the IH applied to the first $n-1$ steps, $S \Rightarrow_{lm}^* wA\beta$.
- Thus, $S \Rightarrow_{lm}^* w\gamma\beta = w\alpha$.

Proof of Part 2 (“if”)

- We also must prove that if $S \Rightarrow^*_{lm} w\alpha$, then $(q, wx, S) \vdash^* (q, x, \alpha)$ for any x .
- Induction on number of steps in the leftmost derivation.
- Ideas are similar; read in text.

Proof – Completion

- We now have $(q, wx, S) \vdash^* (q, x, \alpha)$ for any x if and only if $S \Rightarrow^*_{\text{lm}} w\alpha$.
- In particular, let $x = \alpha = \varepsilon$.
- Then $(q, w, S) \vdash^* (q, \varepsilon, \varepsilon)$ if and only if $S \Rightarrow^*_{\text{lm}} w$.
- That is, w is in $N(P)$ if and only if w is in $L(G)$.

From a PDA to a CFG

- Now, assume $L = N(P)$.
- We'll construct a CFG G such that $L = L(G)$.
- **Intuition:** G will have variables generating exactly the inputs that cause P to have the net effect of popping a stack symbol X while going from state p to state q .
 - P never gets below this X while doing so.

Variables of G

- G's variables are of the form $[pXq]$.
- This variable generates all and only the strings w such that $(p, w, X) \vdash^*(q, \varepsilon, \varepsilon)$.
- Also a start symbol S we'll talk about later.

Productions of G

- Each production for $[pXq]$ comes from a move of P in state p with stack symbol X.
- Simplest case: $\delta(p, a, X)$ contains (q, ϵ) .
- Then the production is $[pXq] \rightarrow a$.
 - Note a can be an input symbol or ϵ .
- Here, $[pXq]$ generates a , because reading a is one way to pop X and go from p to q.

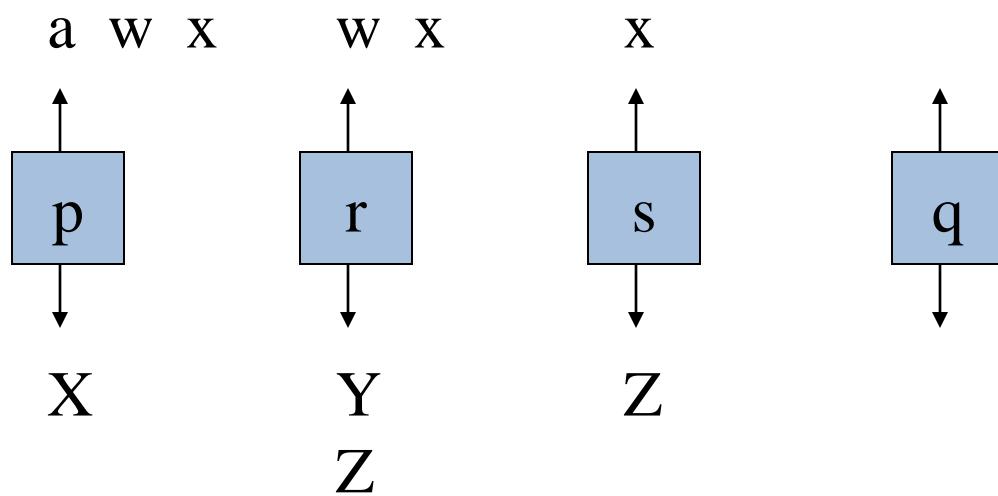
Productions of G – (2)

- Next simplest case: $\delta(p, a, X)$ contains (r, Y) for some state r and symbol Y .
- G has production $[pXq] \rightarrow a[rYq]$.
 - We can erase X and go from p to q by reading a (entering state r and replacing the X by Y) and then reading some w that gets P from r to q while erasing the Y .
- Note: $[pXq] \Rightarrow^* aw$ whenever $[rYq] \Rightarrow^* w$.

Productions of G – (3)

- Third simplest case: $\delta(p, a, X)$ contains (r, YZ) for some state r and symbols Y and Z .
- Now, P has replaced X by YZ .
- To have the net effect of erasing X , P must erase Y , going from state r to some state s , and then erase Z , going from s to q .

Picture of Action of P



Third-Simplest Case – Concluded

- Since we do not know state s, we must generate a family of productions:
 $[pXq] \rightarrow a[rYs][sZq]$
for all states s.
- $[pXq] \Rightarrow^* awx$ whenever $[rYs] \Rightarrow^* w$ and
 $[sZq] \Rightarrow^* x$.

Productions of G: General Case

- Suppose $\delta(p, a, X)$ contains (r, Y_1, \dots, Y_k) for some state r and $k \geq 3$.
- Generate family of productions

$[pXq] \rightarrow$

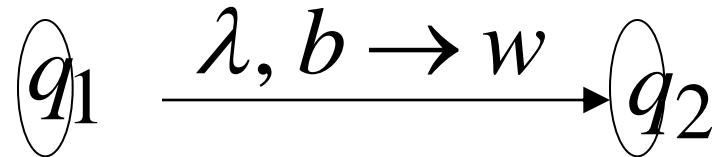
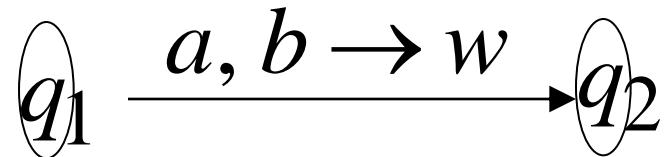
$a[rY_1s_1][s_1Y_2s_2]\dots[s_{k-2}Y_{k-1}s_{k-1}][s_{k-1}Y_kq]$

Completion of the Construction

- We can prove that $(q_0, w, Z_0) \vdash^*(p, \varepsilon, \varepsilon)$ if and only if $[q_0 Z_0 p] \Rightarrow^* w$.
 - Proof is in text; it is two easy inductions.
- But state p can be anything.
- Thus, add to G another variable S , the start symbol, and add productions
- $S \rightarrow [q_0 Z_0 p]$ for each state p .

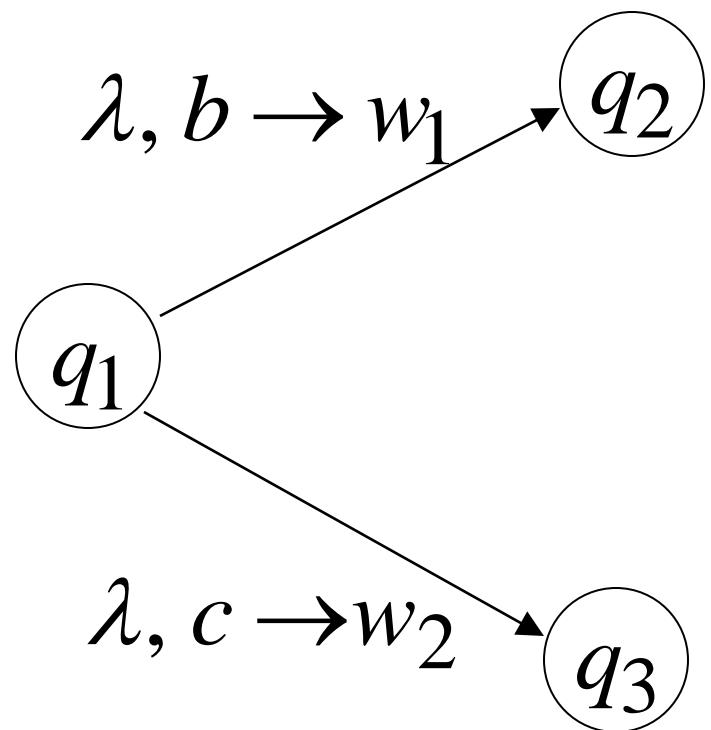
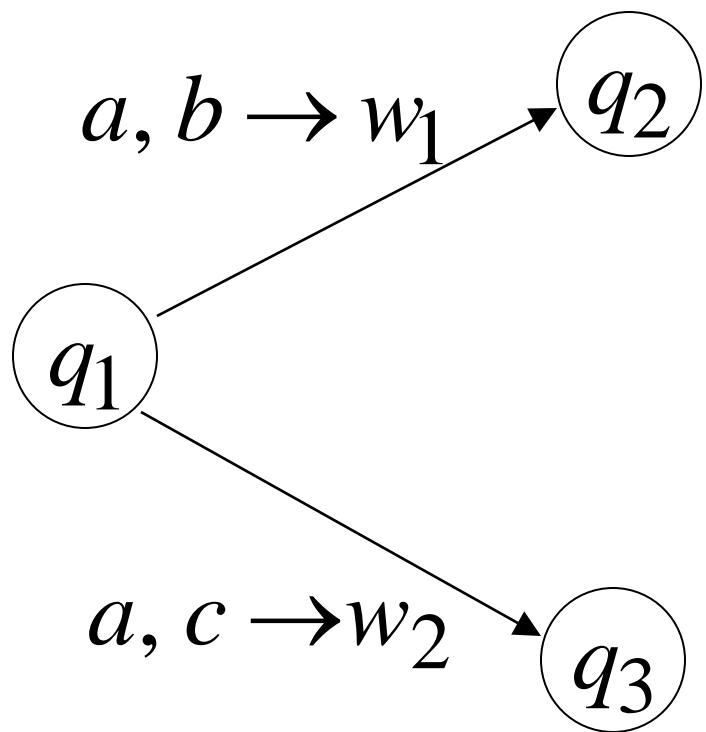
Deterministic PDA: DPDA

Allowed transitions:



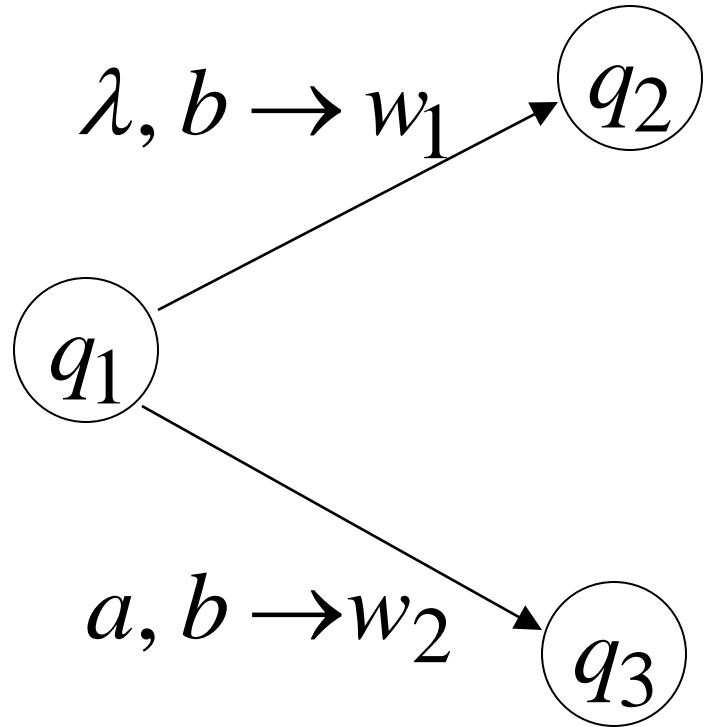
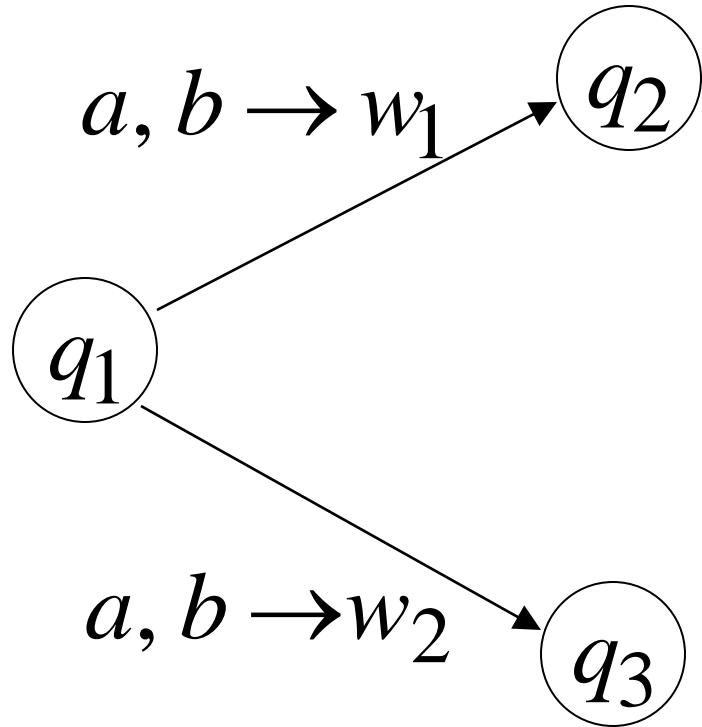
(deterministic choices)

Allowed transitions:



(deterministic choices)

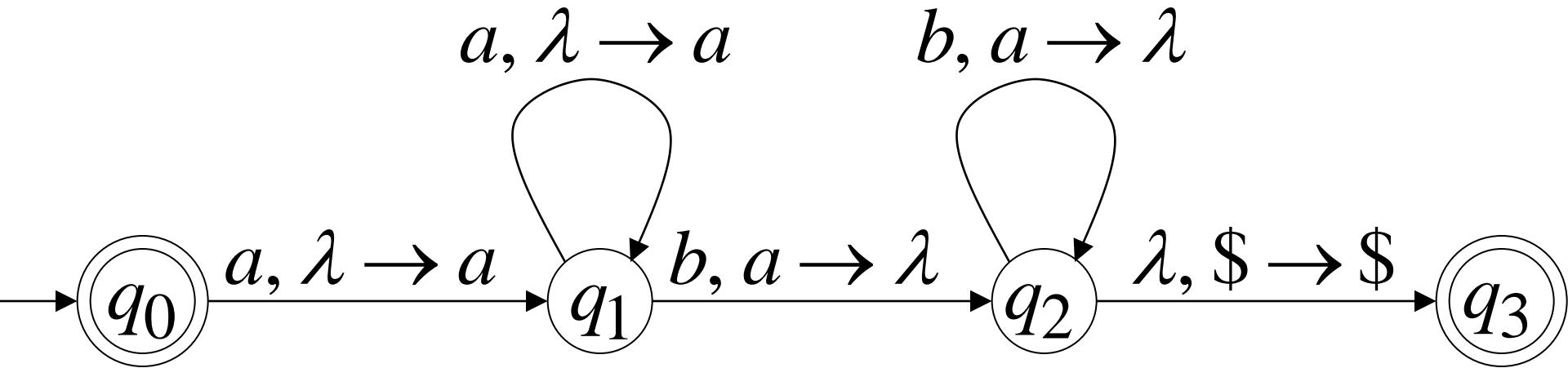
Not allowed:



(non deterministic choices)

DPDA example

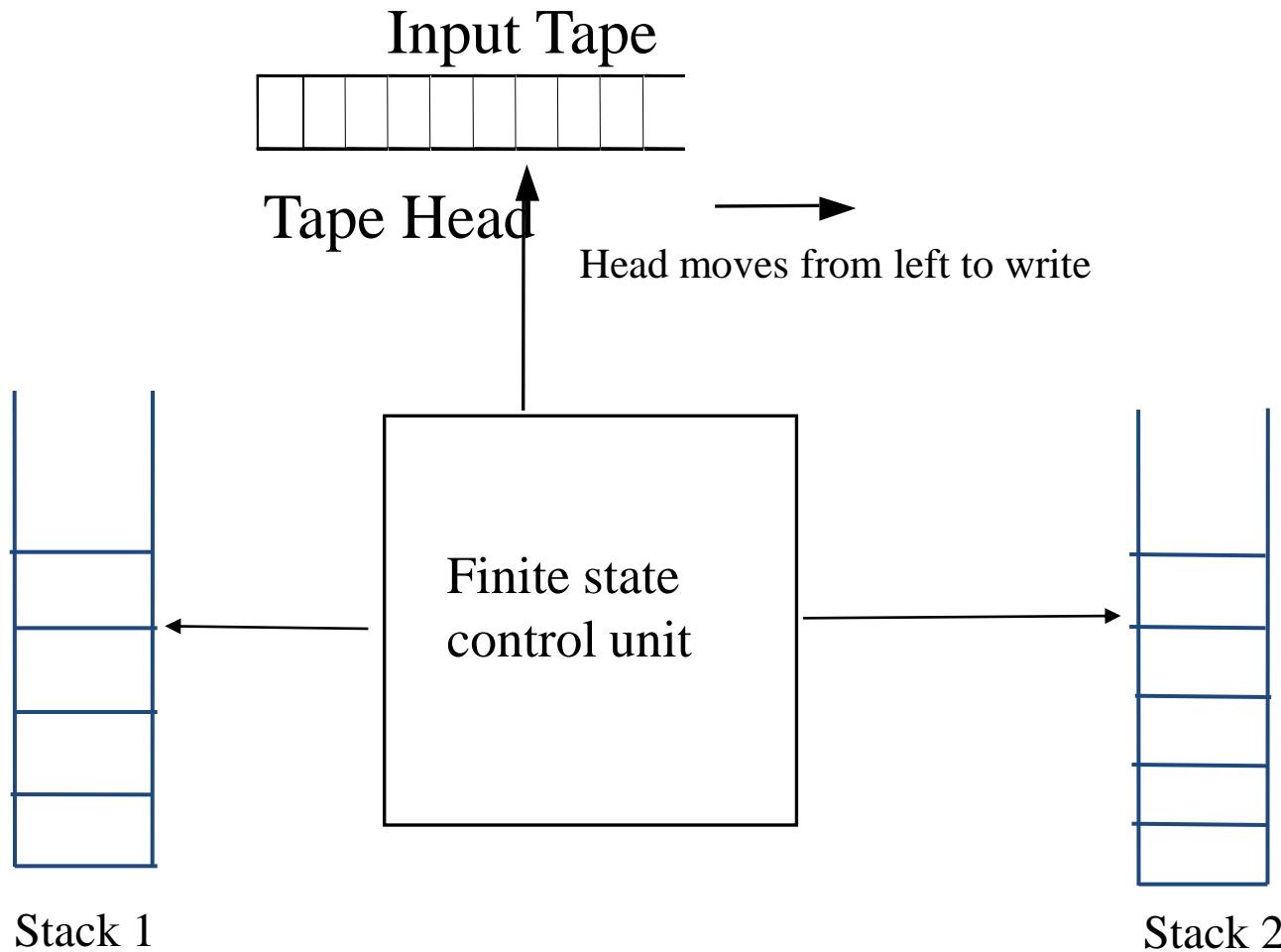
$$L(M) = \{a^n b^n : n \geq 0\}$$



Two-Stack PDA

- A Turing machine can accept languages not accepted by any PDA with one stack.
- The strength of pushdown automata can be increased by adding additional (extra) stacks.
- Actually, a PDA with two stacks has the same computation power as a Turing Machine.

Two-Stack PDA



Two-Stack PDA

k -Stack PDA is also called k -stack machine.

Adding more than two stacks to multistack machine does not increase the power of computation.

Theorem (8.13) (Hopcroft and Ullman [1]): If a language L is accepted by a Turing machine, then L is accepted by a two-stack machine.

Two-Stack PDA

- Two-Stack PDA is a computational model based on the generalization of Pushdown Automata (PDA).
- Non-deterministic Two-Stack PDA is equivalent to a deterministic Two-Stack PDA.
- The move of the Two-Stack PDA is based on
 - The state of the finite control.
 - The input symbol read.
 - The top stack symbol on each of its stacks.

UNIT- 5

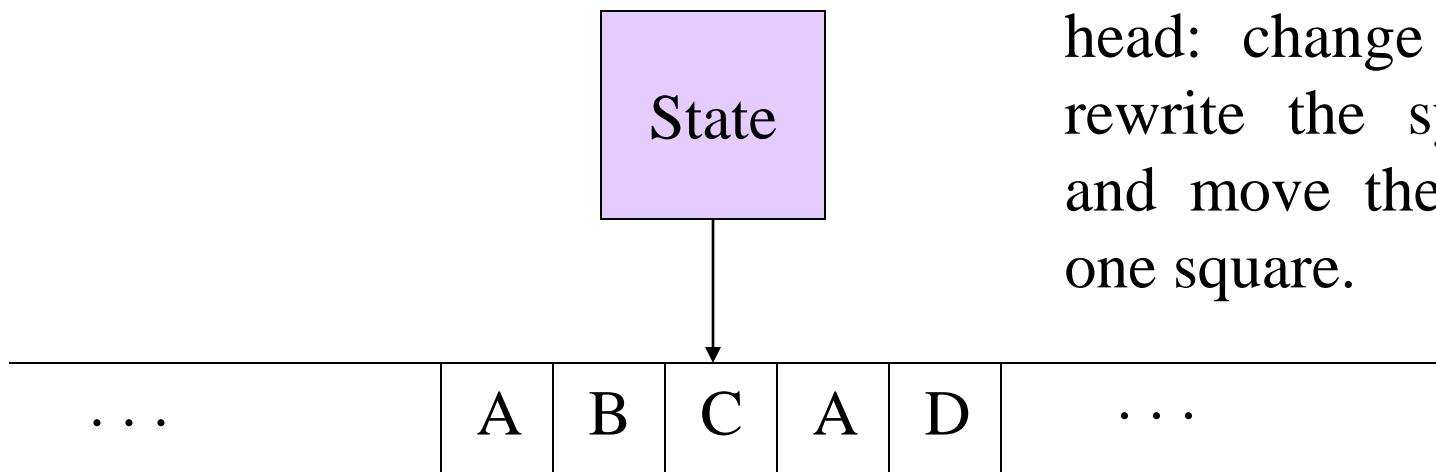
Turing Machines and Undecidability:

The basic model of Turing machines (TM), Instantaneous Description, Variants of Turing Machines, Conversion from RE to TM, LBA, Universal Turing Machine, Recursive and Recursively Enumerable Languages.

The Basic Model of Turing Machine

- The purpose of the theory of Turing machines is to prove that certain specific languages have no algorithm.
- Start with a language about Turing machines themselves.
- Reductions are used to prove more common questions undecidable.

Picture of a Turing Machine



Action: based on the state and the tape symbol under the head: change state, rewrite the symbol and move the head one square.

Infinite tape with squares containing tape symbols chosen from a finite alphabet

Why Turing Machines?

- Why not deal with C programs or something like that?
- **Answer:** You can, but it is easier to prove things about TM's, because they are so simple.
 - And yet they are as powerful as any computer.
 - More so, in fact, since they have infinite memory.

Turing-Machine Formalism

- A TM is described by:
 1. A finite set of *states* (Q , typically).
 2. An *input alphabet* (Σ , typically).
 3. A *tape alphabet* (Γ , typically; contains Σ).
 4. A *transition function* (δ , typically).
 5. A *start state* (q_0 , in Q , typically).
 6. A *blank symbol* (B , in $\Gamma - \Sigma$, typically).
 - ◆ All tape except for the input is blank initially.
 7. A set of *final states* ($F \subseteq Q$, typically).

Conventions

- a, b, \dots are input symbols.
- \dots, X, Y, Z are tape symbols.
- \dots, w, x, y, z are strings of input symbols.
- α, β, \dots are strings of tape symbols.

The Transition Function

- Takes two arguments:
 1. A state, in Q .
 2. A tape symbol in Γ .
- $\delta(q, Z)$ is either undefined or a triple of the form (p, Y, D) .
 - p is a state.
 - Y is the new tape symbol.
 - D is a *direction*, L or R.

Actions of the PDA

- If $\delta(q, Z) = (p, Y, D)$ then, in state q , scanning Z under its tape head, the TM:
 1. Changes the state to p .
 2. Replaces Z by Y on the tape.
 3. Moves the head one square in direction D .
 - ◆ $D = L$: move left; $D = R$: move right.

Example: Turing Machine

- This TM scans its input right, looking for a 1.
- If it finds one, it changes it to a 0, goes to final state f , and halts.
- If it reaches a blank, it changes it to a 1 and moves left.

Example: Turing Machine – (2)

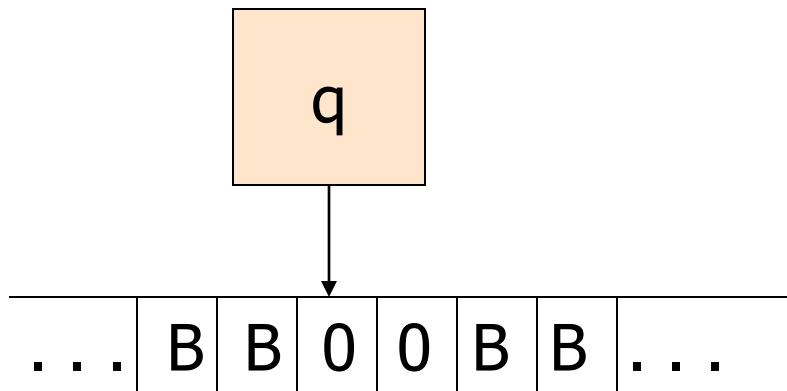
- States = {q (start), f (final)}.
- Input symbols = {0, 1}.
- Tape symbols = {0, 1, B}.
- $\delta(q, 0) = (q, 0, R)$.
- $\delta(q, 1) = (f, 0, R)$.
- $\delta(q, B) = (q, 1, L)$.

Simulation of TM

$$\delta(q, 0) = (q, 0, R)$$

$$\delta(q, 1) = (f, 0, R)$$

$$\delta(q, B) = (q, 1, L)$$

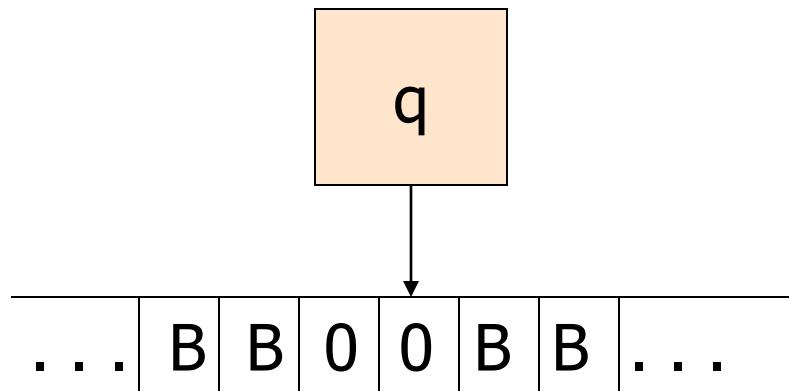


Simulation of TM

$$\delta(q, 0) = (q, 0, R)$$

$$\delta(q, 1) = (f, 0, R)$$

$$\delta(q, B) = (q, 1, L)$$

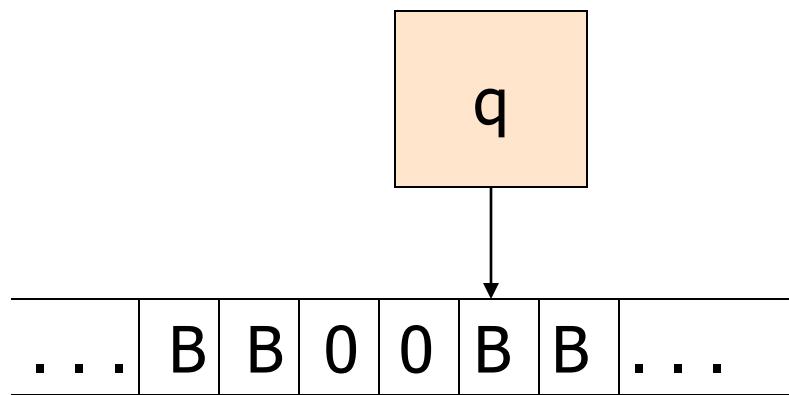


Simulation of TM

$$\delta(q, 0) = (q, 0, R)$$

$$\delta(q, 1) = (f, 0, R)$$

$$\delta(q, B) = (q, 1, L)$$

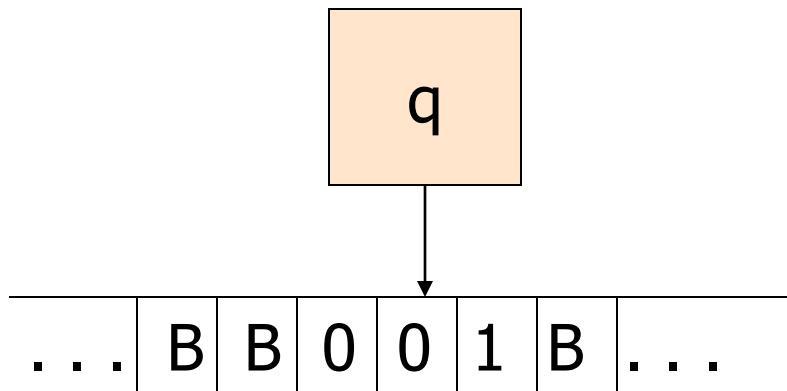


Simulation of TM

$$\delta(q, 0) = (q, 0, R)$$

$$\delta(q, 1) = (f, 0, R)$$

$$\delta(q, B) = (q, 1, L)$$

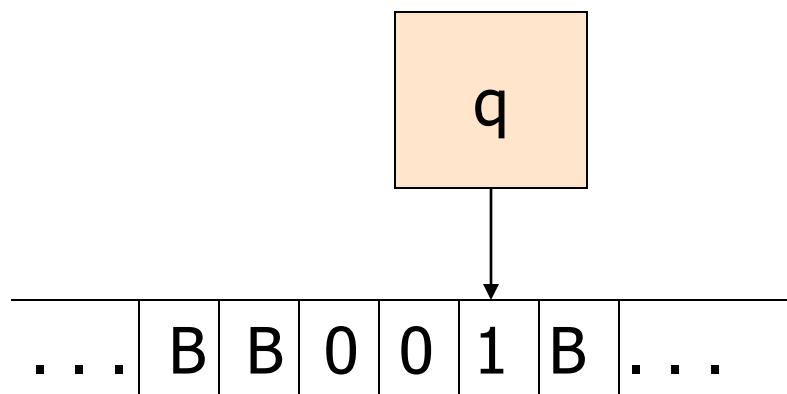


Simulation of TM

$$\delta(q, 0) = (q, 0, R)$$

$$\delta(q, 1) = (f, 0, R)$$

$$\delta(q, B) = (q, 1, L)$$

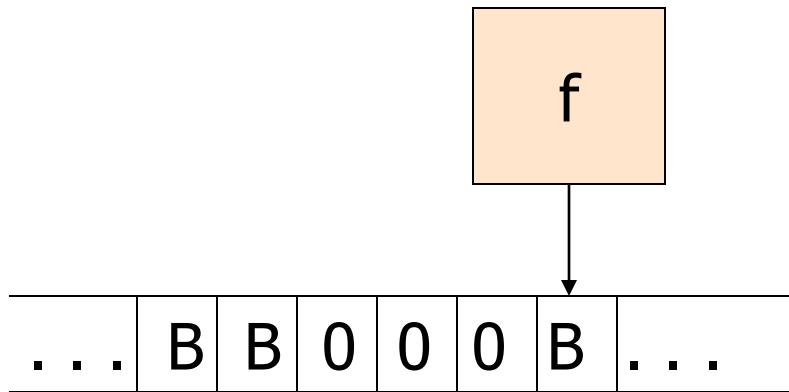


Simulation of TM

$$\delta(q, 0) = (q, 0, R)$$

$$\delta(q, 1) = (f, 0, R)$$

$$\delta(q, B) = (q, 1, L)$$



No move is possible.
The TM halts and
accepts.

Instantaneous Descriptions of a Turing Machine

- Initially, a TM has a tape consisting of a string of input symbols surrounded by an infinity of blanks in both directions.
- The TM is in the start state, and the head is at the leftmost input symbol.

TM ID's – (2)

- An ID is a string $\alpha q \beta$, where $\alpha \beta$ is the tape between the leftmost and rightmost nonblanks (inclusive).
- The state q is immediately to the left of the tape symbol scanned.
- If q is at the right end, it is scanning B .
 - If q is scanning a B at the left end, then consecutive B 's at and to the right of q are part of α .

TM ID's – (3)

- As for PDA's we may use symbols \vdash and \vdash^* to represent “becomes in one move” and “becomes in zero or more moves,” respectively, on ID's.
- **Example:** The moves of the previous TM are
 $q00 \vdash 0q0 \vdash 00q \vdash 0q01 \vdash 00q1 \vdash 000f$

Formal Definition of Moves

1. If $\delta(q, Z) = (p, Y, R)$, then
 - ◆ $\alpha q Z \beta \vdash \alpha Y p \beta$
 - ◆ If Z is the blank B , then also $\alpha q \vdash \alpha Y p$
2. If $\delta(q, Z) = (p, Y, L)$, then
 - ◆ For any X , $\alpha X q Z \beta \vdash \alpha p X Y \beta$
 - ◆ In addition, $q Z \beta \vdash p B Y \beta$

Languages of a TM

- A TM defines a language by final state, as usual.
- $L(M) = \{w \mid q_0 w \vdash^* I\}$, where I is an ID with a final state}.
- Or, a TM can accept a language by halting.
- $H(M) = \{w \mid q_0 w \vdash^* I\}$, and there is no move possible from ID $I\}$.

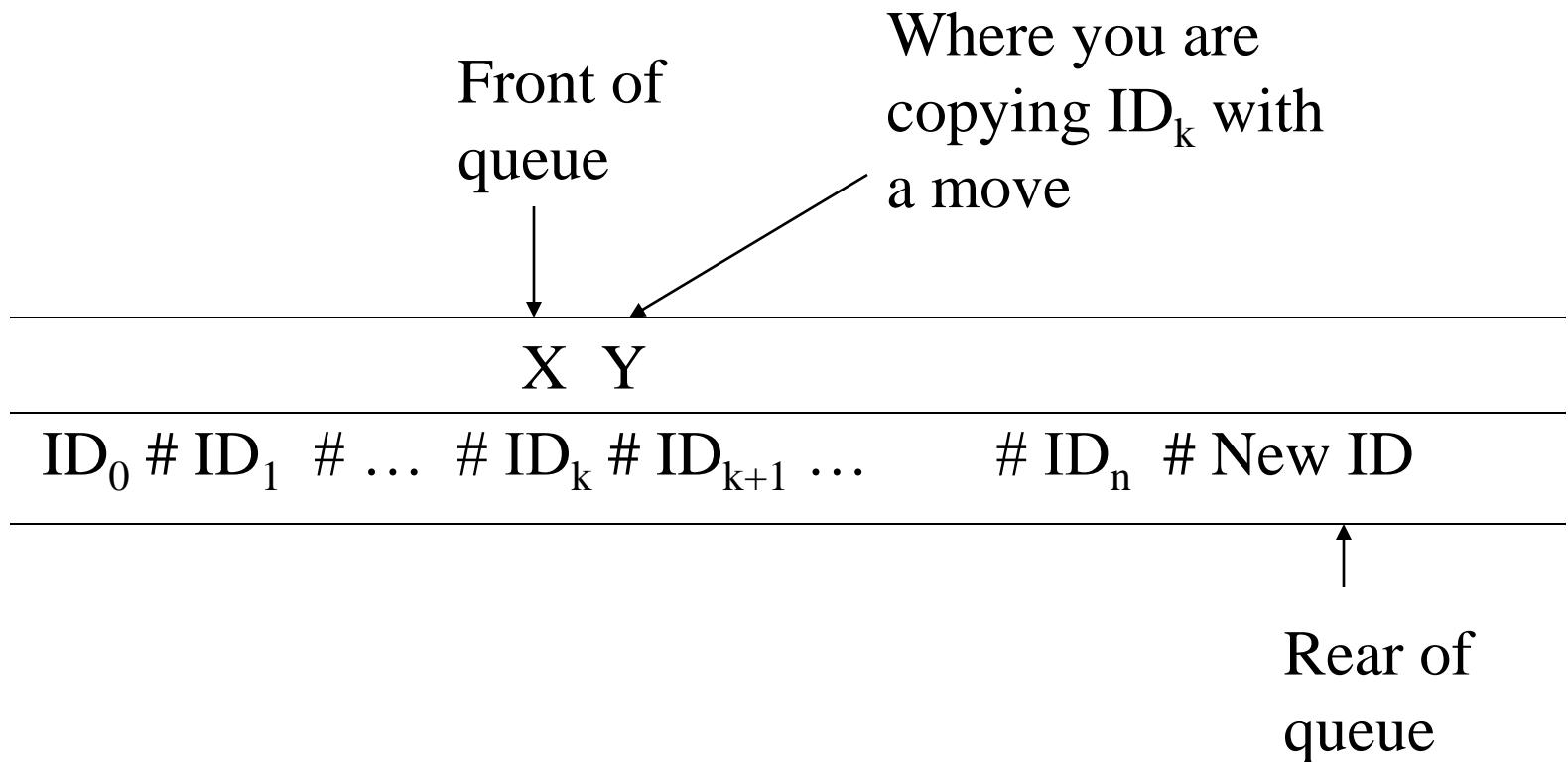
Nondeterministic TM's

- Allow the TM to have a choice of move at each step.
 - Each choice is a state-symbol-direction triple, as for the deterministic TM.
- The TM accepts its input if any sequence of choices leads to an accepting state.

Simulating a NTM by a DTM

- The DTM maintains on its tape a queue of ID's of the NTM.
- A second track is used to mark certain positions:
 1. A mark for the ID at the head of the queue.
 2. A mark to help copy the ID at the head and make a one-move change.

Picture of the DTM Tape



Operation of the Simulating DTM

- The DTM finds the ID at the current front of the queue.
- It looks for the state in that ID so it can determine the moves permitted from that ID.
- If there are m possible moves, it creates m new ID's, one for each move, at the rear of the queue.

Operation of the DTM – (2)

- The m new ID's are created one at a time.
- After all are created, the marker for the front of the queue is moved one ID toward the rear of the queue.
- However, if a created ID has an accepting state, the DTM instead accepts and halts.

Taking Advantage of Extensions

- We now have a really good situation.
- When we discuss construction of particular TM's that take other TM's as input, we can assume the input TM is as simple as possible.
 - E.g., one, semi-infinite tape, deterministic.
- But the simulating TM can have many tapes, be nondeterministic, etc.

The Universal Language

- An example of a recursively enumerable, but not recursive language is the language L_u of a *universal Turing machine*.
- That is, the UTM takes as input the code for some TM M and some binary string w and accepts if and only if M accepts w .

Designing the UTM

- Inputs are of the form:
Code for M 111 w
- Note: A valid TM code never has 111, so we can split M from w.
- The UTM must accept its input if and only if M is a valid TM code and that TM accepts w.

The UTM – (2)

- The UTM will have several tapes.
- Tape 1 holds the input $M111w$
- Tape 2 holds the tape of M .
 - Mark the current head position of M .
- Tape 3 holds the state of M .

The UTM – (3)

- Step 1: The UTM checks that M is a valid code for a TM.
 - E.g., all moves have five components, no two moves have the same state/symbol as first two components.
- If M is not valid, its language is empty, so the UTM immediately halts without accepting.

The UTM – (4)

- **Step 2:** The UTM examines M to see how many of its own tape squares it needs to represent one symbol of M .
- **Step 3:** Initialize Tape 2 to represent the tape of M with input w , and initialize Tape 3 to hold the start state.

The UTM – (5)

- Step 4: Simulate M.
 - Look for a move on Tape 1 that matches the state on Tape 3 and the tape symbol under the head on Tape 2.
 - If found, change the symbol and move the head marker on Tape 2 and change the State on Tape 3.
 - If M accepts, the UTM also accepts.

Recursively Enumerable Languages

- We now see that the classes of languages defined by TM's using final state and halting are the same.
- This class of languages is called the *recursively enumerable languages*.
 - Why? The term actually predates the Turing machine and refers to another notion of computation of functions.

Recursive Languages

- An *algorithm* is a TM that is guaranteed to halt whether or not it accepts.
- If $L = L(M)$ for some TM M that is an algorithm, we say L is a *recursive language*.
 - Why? Again, don't ask; it is a term with a history.

Example: Recursive Languages

- Every CFL is a recursive language.
 - Use the CYK (Cocke–Younger–Kasami) algorithm.
- Every regular language is a CFL (think of its DFA as a PDA that ignores its stack); therefore every regular language is recursive.

Closure Properties of Recursive and RE Languages

- Both closed under union, concatenation, star, reversal, intersection, inverse homomorphism.
- Recursive closed under difference, complementation.
- RE closed under homomorphism.

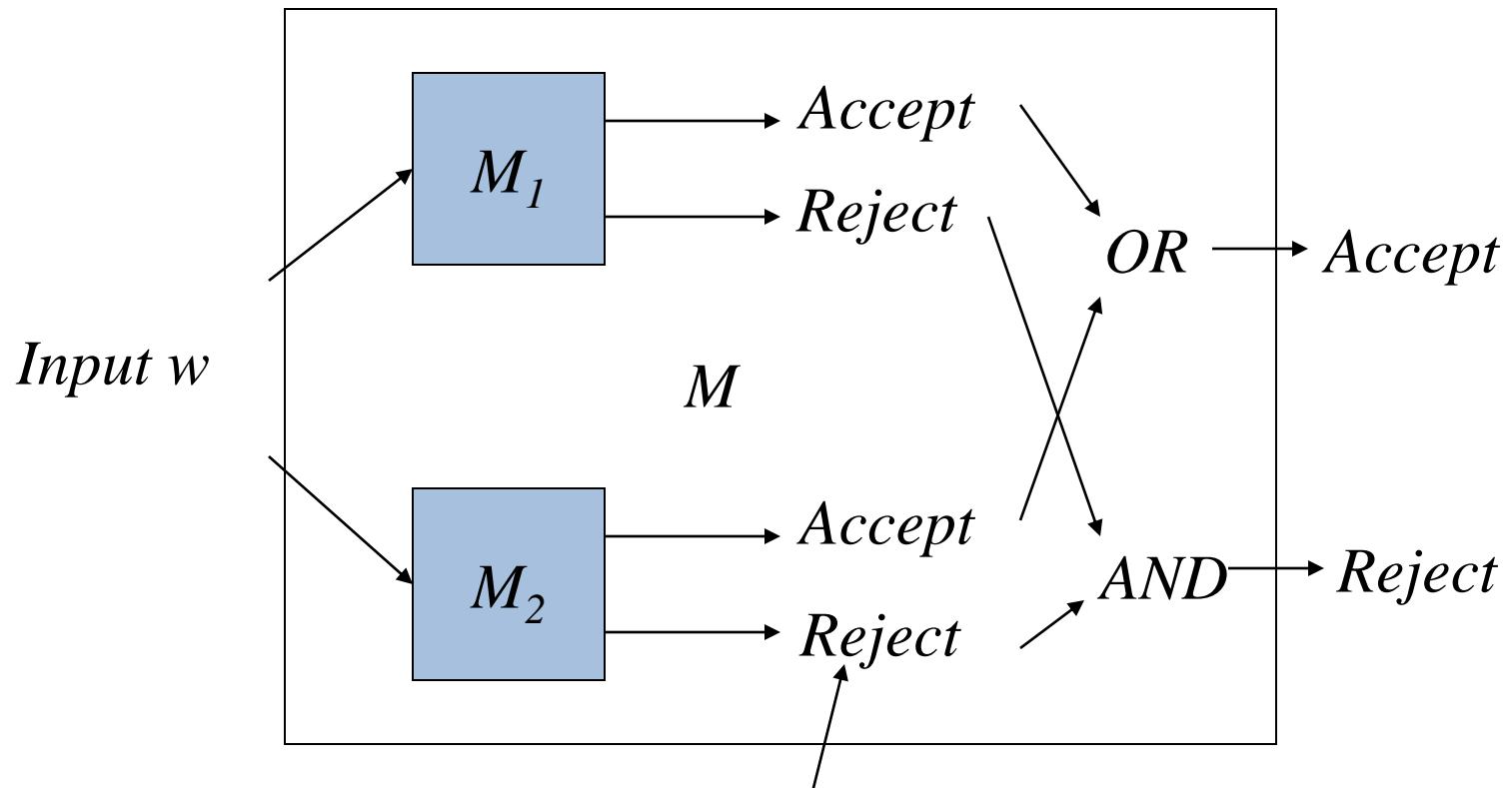
Union

- Let $L_1 = L(M_1)$ and $L_2 = L(M_2)$.
- Assume M_1 and M_2 are single-semi-infinite-tape TM's.
- Construct 2-tape TM M to copy its input onto the second tape and simulate the two TM's M_1 and M_2 each on one of the two tapes, “in parallel.”

Union – (2)

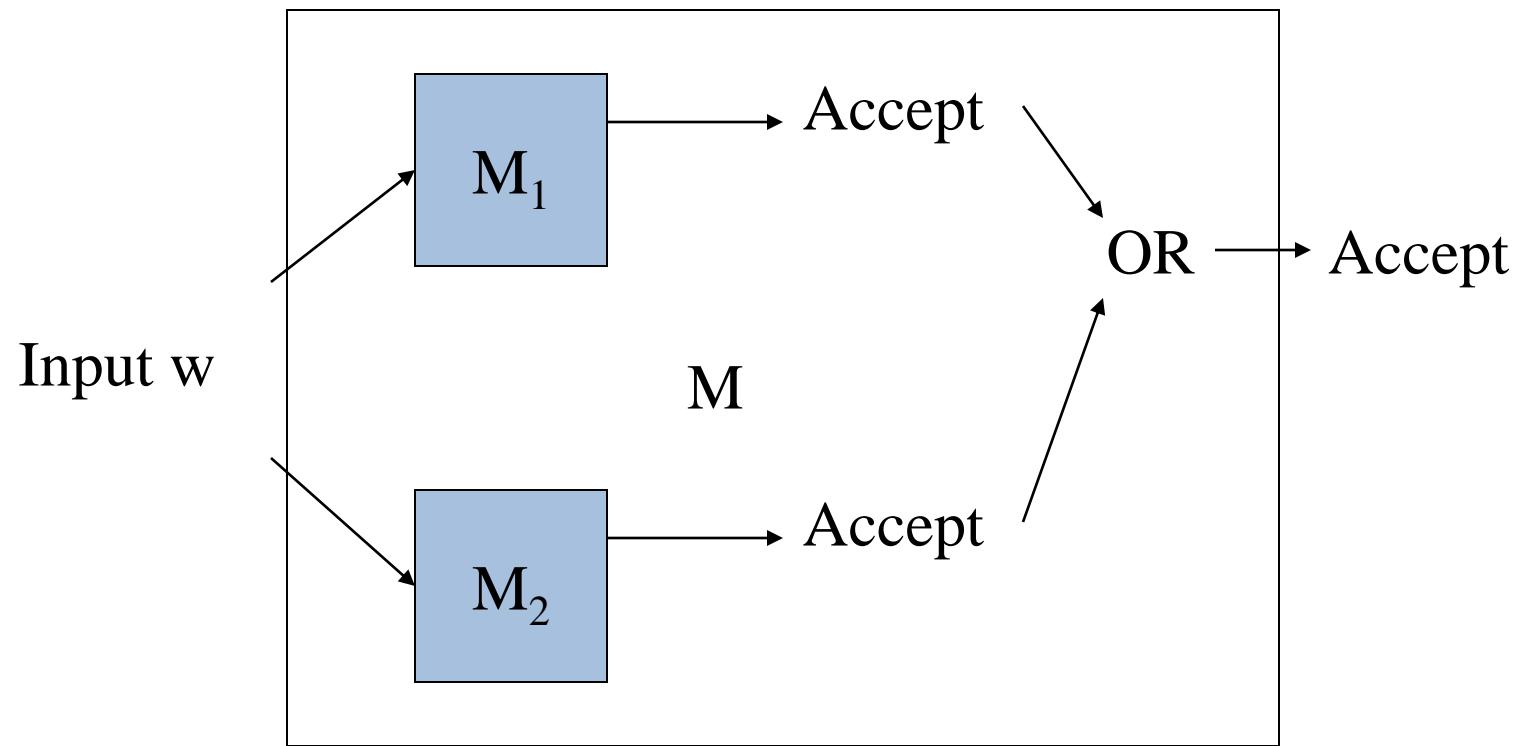
- **Recursive languages:** If M_1 and M_2 are both algorithms, then M will always halt in both simulations.
- Accept if either accepts.
- **RE languages:** accept if either accepts, but you may find both TM's run forever without halting or accepting.

Picture of Union/Recursive

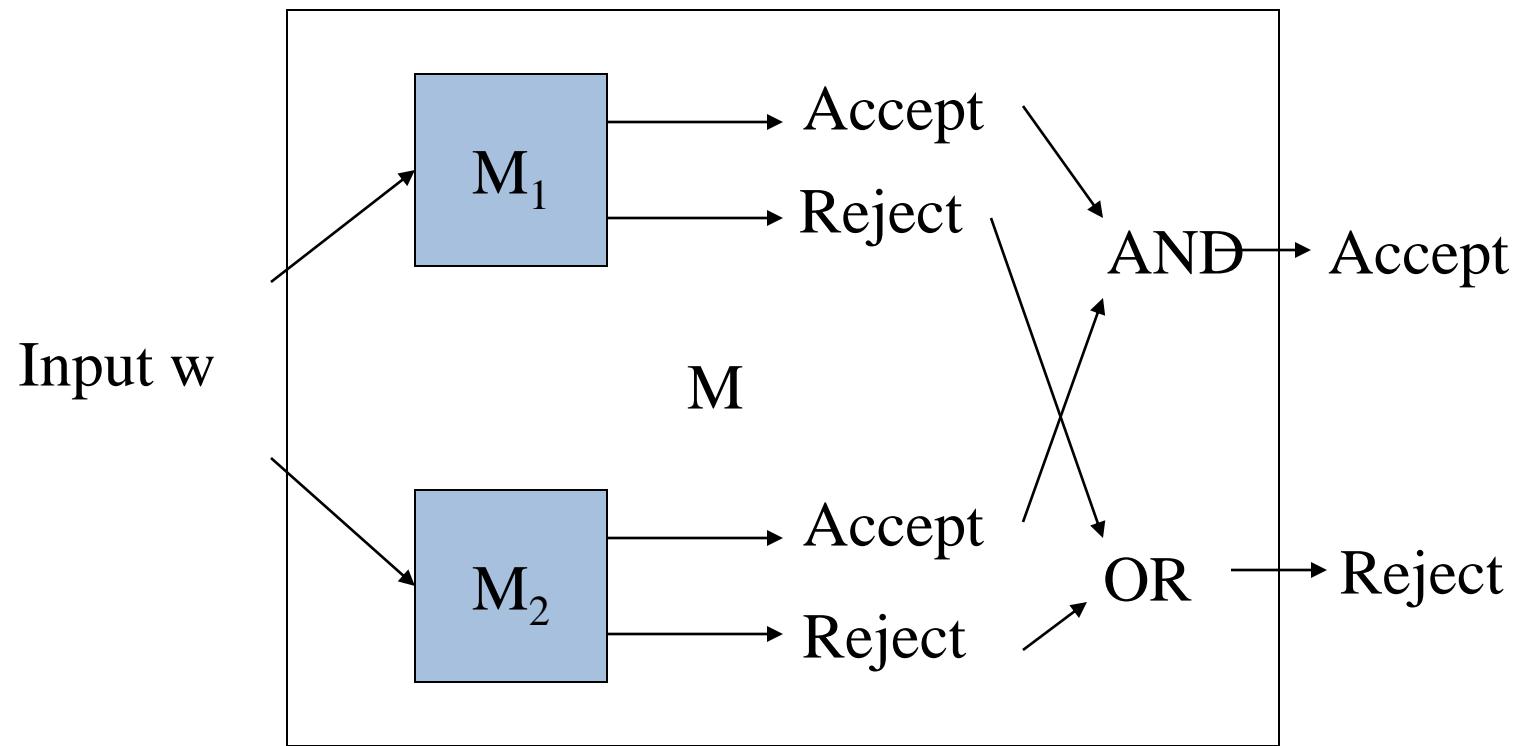


Remember: = “halt
without accepting

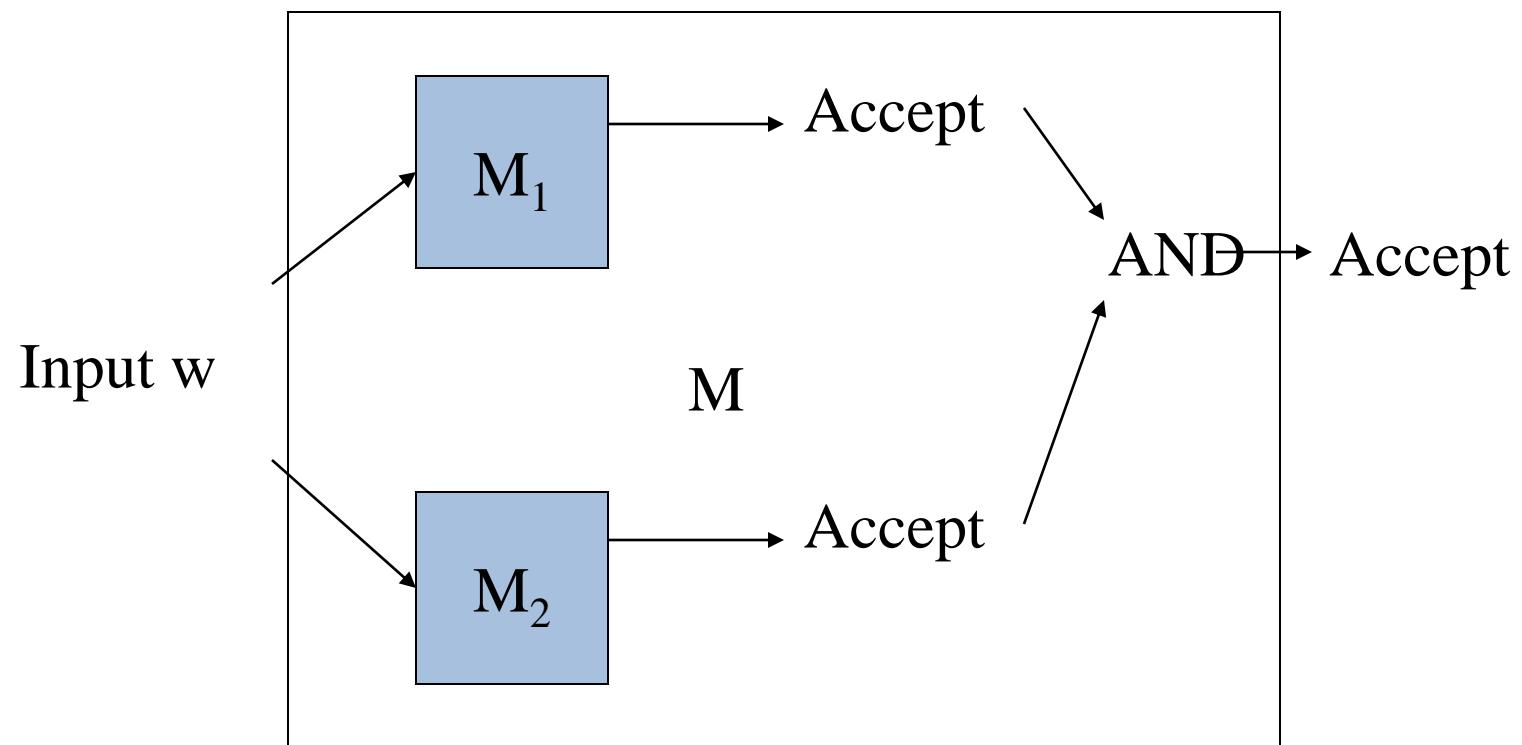
Picture of Union/RE



Intersection/Recursive – Same Idea



Intersection/RE



Difference, Complement

- **Recursive languages:** both TM's will eventually halt.
- Accept if M_1 accepts and M_2 does not.
 - **Corollary:** Recursive languages are closed under complementation.
- **RE Languages:** can't do it; M_2 may never halt, so you can't be sure input is in the difference.

Concatenation/RE

- Let $L_1 = L(M_1)$ and $L_2 = L(M_2)$.
- Assume M_1 and M_2 are single-semi-infinite-tape TM's.
- Construct 2-tape Nondeterministic TM M :
 1. Guess a break in input $w = xy$.
 2. Move y to second tape.
 3. Simulate M_1 on x , M_2 on y .
 4. Accept if both accept.

Concatenation/Recursive

- Can't use a NTM.
- Systematically try each break $w = xy$.
- M_1 and M_2 will eventually halt for each break.
- Accept if both accept for any one break.
- Reject if all breaks tried and none lead to acceptance.

Star

- Same ideas work for each case.
- **RE**: guess many breaks, accept if M_1 accepts each piece.
- **Recursive**: systematically try all ways to break input into some number of pieces.

Reversal

- Start by reversing the input.
- Then simulate TM for L to accept w if and only w^R is in L .
- Works for either Recursive or RE languages.

Inverse Homomorphism

- Apply h to input w .
- Simulate TM for L on $h(w)$.
- Accept w iff $h(w)$ is in L .
- Works for Recursive or RE.

Homomorphism/RE

- Let $L = L(M_1)$.
- Design NTM M to take input w and guess an x such that $h(x) = w$.
- M accepts whenever M_1 accepts x .
- **Note:** won't work for Recursive languages.

Decidable Problems

- A problem is *decidable* if there is an algorithm to answer it.
 - Recall: An “algorithm,” formally, is a TM that halts on all inputs, accepted or not.
 - Put another way, “decidable problem” = “recursive language.”
- Otherwise, the problem is *undecidable*.

Bullseye Picture

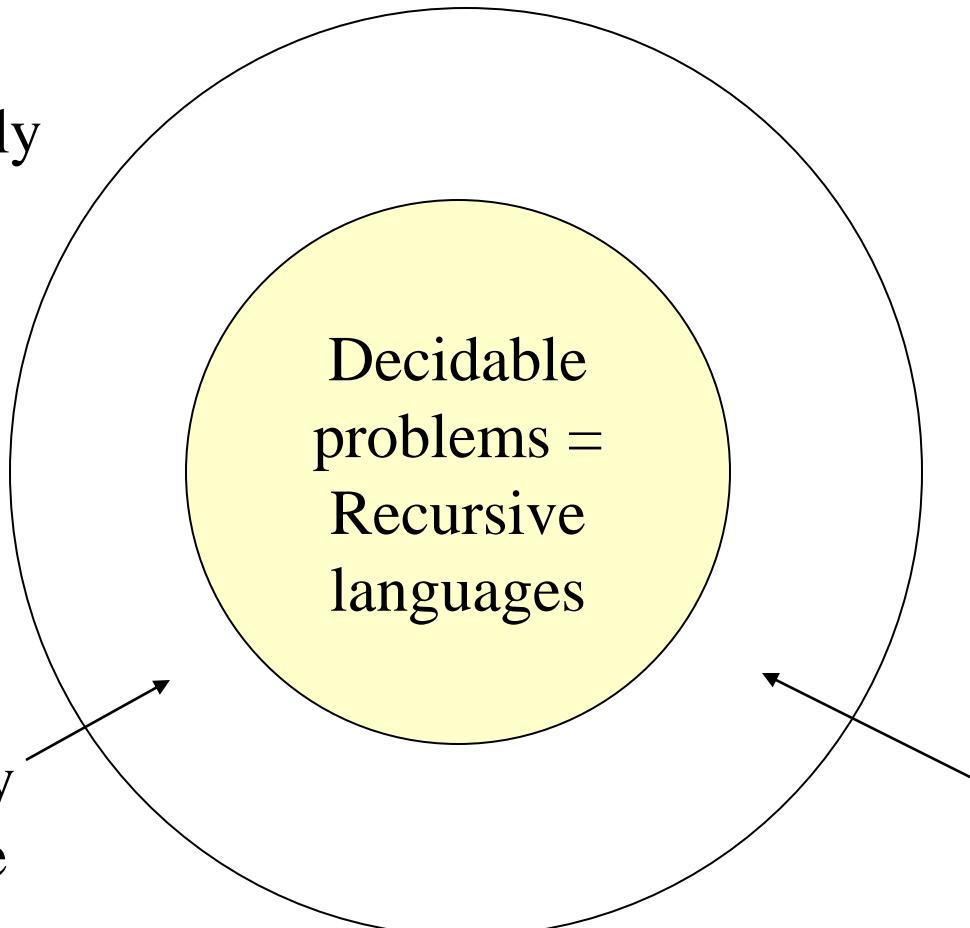
Not recursively
enumerable
languages

Decidable
problems =
Recursive
languages

Recursively
enumerable
languages

L_d

Are there
any languages
here?



From the Abstract to the Real

- While the fact that L_d is undecidable is interesting intellectually, it doesn't impact the real world directly.
- We first shall develop some TM-related problems that are undecidable, but our goal is to use the theory to show some real problems are undecidable.

Examples: Undecidable Problems

- Can a particular line of code in a program ever be executed?
- Is a given context-free grammar ambiguous?
- Do two given CFG's generate the same language?

Post's Correspondence Problem

- But we're still stuck with problems about Turing machines only.
- Post's Correspondence Problem (PCP) is an example of a problem that does not mention TM's in its statement, yet is undecidable.
- From PCP, we can prove many other non-TM problems undecidable.

PCP Instances

- An instance of PCP is a list of pairs of nonempty strings over some alphabet Σ .
 - Say $(w_1, x_1), (w_2, x_2), \dots, (w_n, x_n)$.
- The answer to this instance of PCP is “yes” if and only if there exists a nonempty sequence of indices i_1, \dots, i_k , such that $w_{i1} \dots w_{in} = x_{i1} \dots x_{in}$.

In text: a pair of lists.



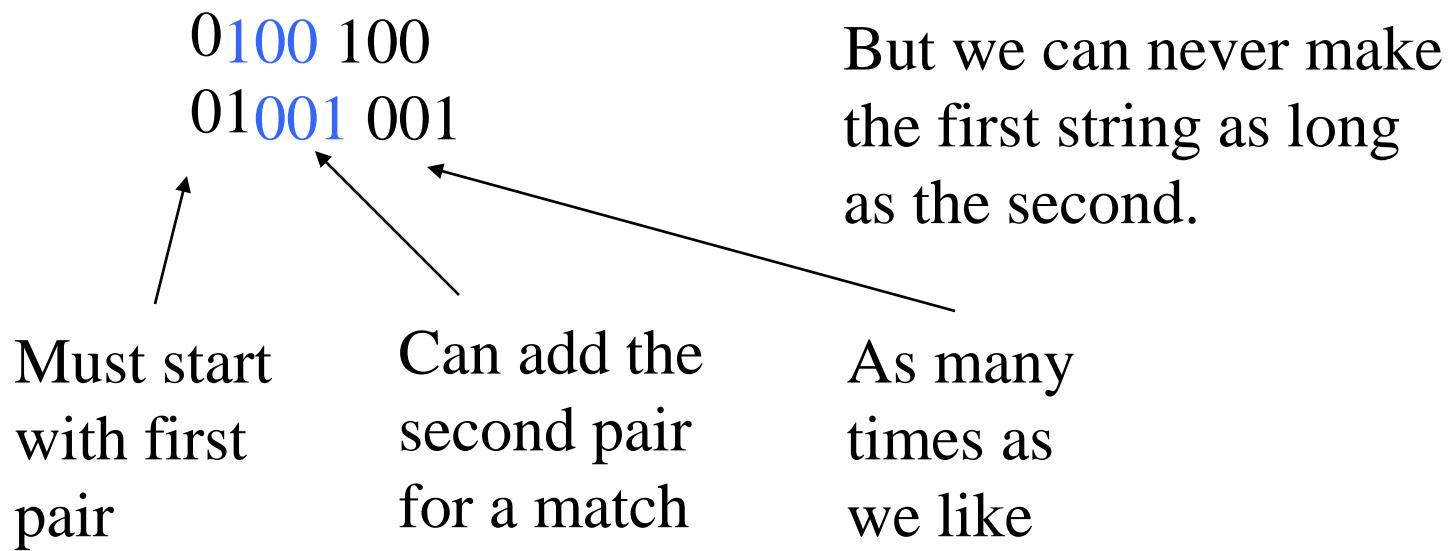
Should be “ w_{i1} ,” etc.

Example: PCP

- Let the alphabet be $\{0, 1\}$.
- Let the PCP instance consist of the two pairs $(0, 01)$ and $(100, 001)$.
- We claim there is no solution.
- You can't start with $(100, 001)$, because the first characters don't match.

Example: PCP – (2)

Recall: pairs are $(0, 01)$ and $(100, 001)$



Example: PCP – (3)

- Suppose we add a third pair, so the instance becomes: $1 = (0, 01); 2 = (100, 001); 3 = (110, 10)$.
- Now $1,3$ is a solution; both strings are 0110 .
- In fact, any sequence of indexes in $\mathbf{12*3}$ is a solution.

Proving PCP is Undecidable

- We'll introduce the *modified* PCP (MPCP) problem.
 - Same as PCP, but the solution must start with the first pair in the list.
- We reduce L_u to MPCP.
- But first, we'll reduce MPCP to PCP.

Example: MPCP

- The list of pairs $(0, 01)$, $(100, 001)$, $(110, 10)$, as an instance of MPCP, has a solution as we saw.
- However, if we reorder the pairs, say $(110, 10)$, $(0, 01)$, $(100, 001)$ there is no solution.
 - No string $110\dots$ can ever equal a string $10\dots$.

Representing PCP or MPCP Instances

- Since the alphabet can be arbitrarily large, we need to code symbols.
- Say the i -th symbol will be coded by “ a ” followed by i in binary.
- Commas and parentheses can represent themselves.

Representing Instances – (2)

- Thus, we have a finite alphabet in which all instances of PCP or MPCP can be represented.
- Let L_{PCP} and L_{MPCP} be the languages of coded instances of PCP or MPCP, respectively, that have a solution.

Reducing L_{MPCP} to L_{PCP}

- Take an instance of L_{MPCP} and do the following, using new symbols * and \$.
 1. For the first string of each pair, add * **after** every character.
 2. For the second string of each pair, add * **before** every character.
 3. Add pair (\$, *\$).
 4. Make another copy of the first pair, with *'s and an extra * prepended to the first string.

Example: L_{MPCP} to L_{PCP}

MPCP instance,
in order:

(110, 10)

(0, 01)

(100, 001)

PCP instance:

(1*1*0*, *1*0)

(0*, *0*1)

(1*0*0*, *0*0*1)

(\$, *\$)

(*1*1*0*, *1*0)

Add
*'s

Ender

Special pair version of
first MPCP choice – only
possible start for a PCP
solution.