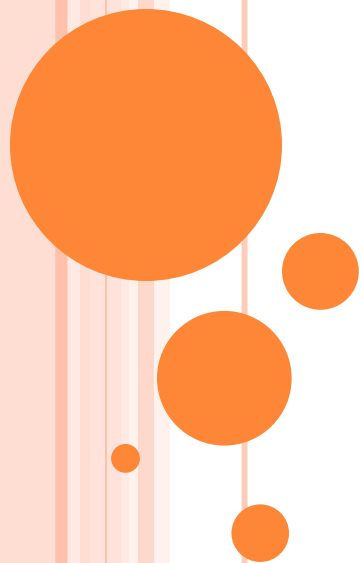


Unit 2



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 $(Q, \Sigma, O, \delta, X, q_0)$

where –

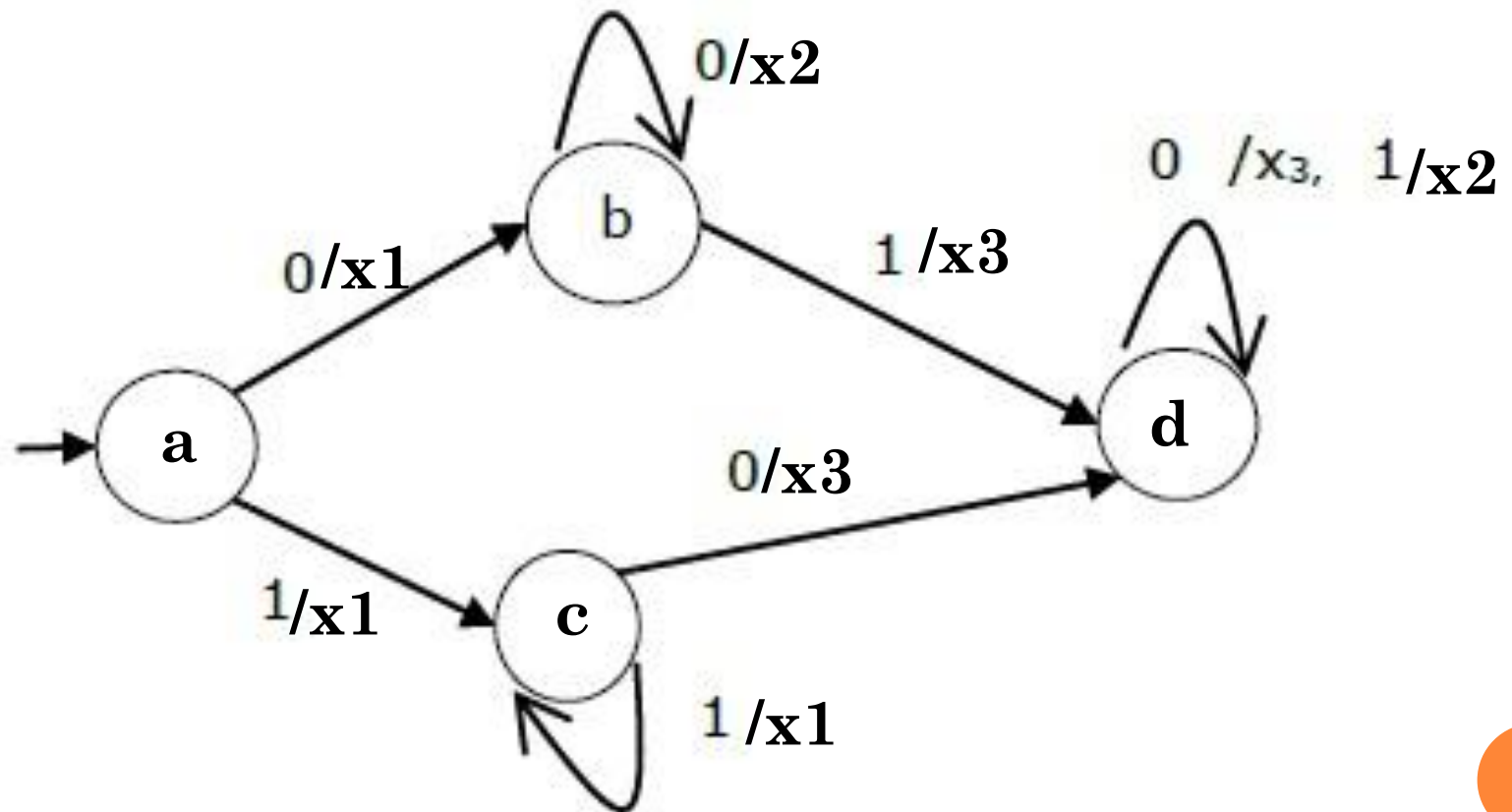
- **Q** is a finite set of states.
- Σ is a finite set of symbols called the input alphabet.
- **O** is a finite set of symbols called the output alphabet.
- **δ** is the input transition function where $\delta: Q \times \Sigma \rightarrow Q$
- **X** is the output transition function where $X: Q \times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



The state table of a Mealy Machine is shown below –

Present state	Next state			
	input = 0		input = 1	
	State	Output	State	Output
→ 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

The state diagram of the above Mealy Machine is –



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 $(Q, \Sigma, O, \delta, X, q_0)$ where –
 - **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 ($q_0 \in Q$).

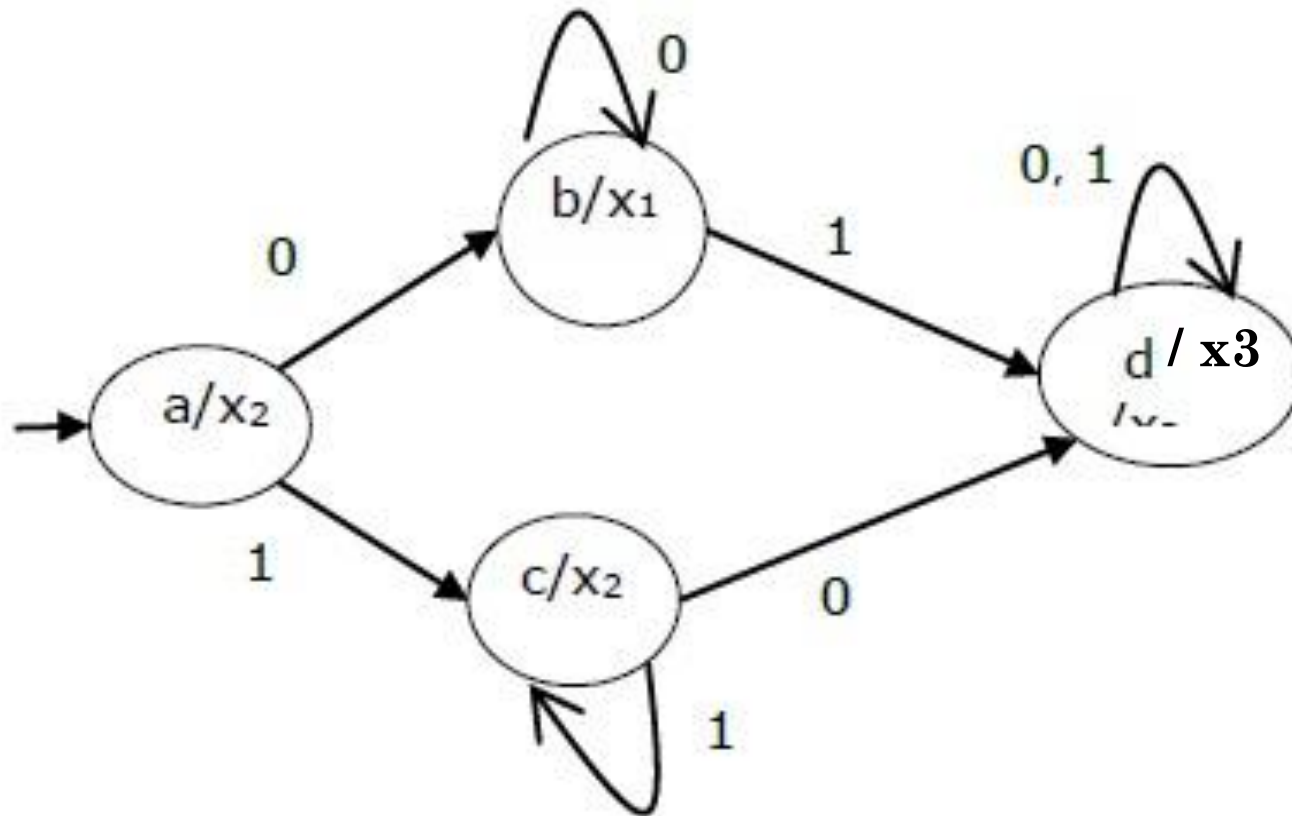


The state table of a Moore Machine is shown below

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



The state diagram of the above Moore Machine is



Mealy Machine vs. Moore Machine

Mealy Machine	Moore Machine
Output depends both upon the present state and the present input	Output depends only upon the present state.
Generally, it has fewer states than Moore Machine.	Generally, it has more states than Mealy Machine.
The value of the output function is a function of the transitions and the changes, when the input logic on the present state is done.	The value of the output function is a function of the current state and the changes at the clock edges, whenever state changes occur.
Mealy machines react faster to inputs. They generally react in the same clock cycle.	In Moore machines, more logic is required to decode the outputs resulting in more circuit delays. They generally react one clock cycle later.



THE PUMPING LEMMA

- We have, almost accidentally, proved a statement that is quite useful for showing certain languages are not regular.
- Called the *pumping lemma for regular languages*.



STATEMENT OF THE PUMPING LEMMA

For every regular language L

There is an integer n , such that

Number of
states of
DFA for L

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

We can write $w = xyz$ such that:

1. $|xy| \leq n$.
2. $|y| > 0$.
3. For all $i \geq 0$, xy^iz is in L .

Labels along
first cycle on
path labeled w



EXAMPLE: USE OF PUMPING LEMMA

- We have claimed $\{0^k 1^k \mid k \geq 1\}$ is not a regular language.
- Suppose it were. Then there would be an associated n for the pumping lemma.
- Let $w = 0^n 1^n$. We can write $w = xyz$, where x and y consist of 0's, and $y \neq \epsilon$.
- But then $xyyz$ would be in L , and this string has more 0's than 1's.



CLOSURE PROPERTIES

- Recall a closure property is a statement that a certain operation on languages, when applied to languages in a class (e.g., the regular languages), produces a result that is also in that class.
- For regular languages, we can use any of its representations to prove a closure property.



CLOSURE UNDER UNION

- If L and M are regular languages, so is $L \cup M$.
- **Proof:** Let L and M be the languages of regular expressions R and S , respectively.
- Then $R+S$ is a regular expression whose language is $L \cup M$.



CLOSURE UNDER CONCATENATION AND KLEENE CLOSURE

- Same idea:
 - RS is a regular expression whose language is LM .
 - R^* is a regular expression whose language is L^* .

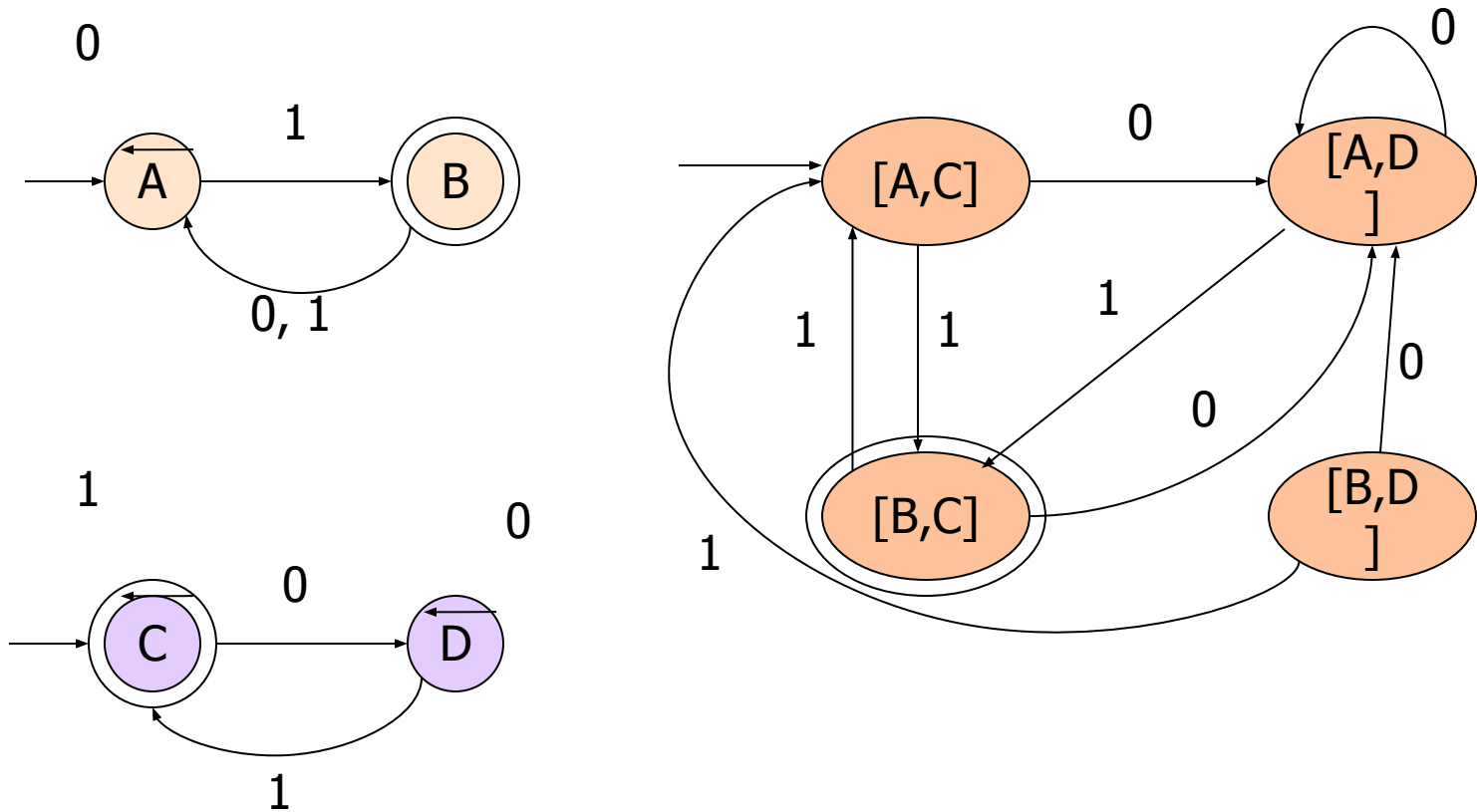


CLOSURE UNDER INTERSECTION

- If L and M are regular languages, then so is $L \cap M$.
- **Proof:** Let A and B be DFA's whose languages are L and M , respectively.
- Construct C , the product automaton of A and B .
- Make the final states of C be the pairs consisting of final states of both A and B .



EXAMPLE: PRODUCT DFA FOR INTERSECTION

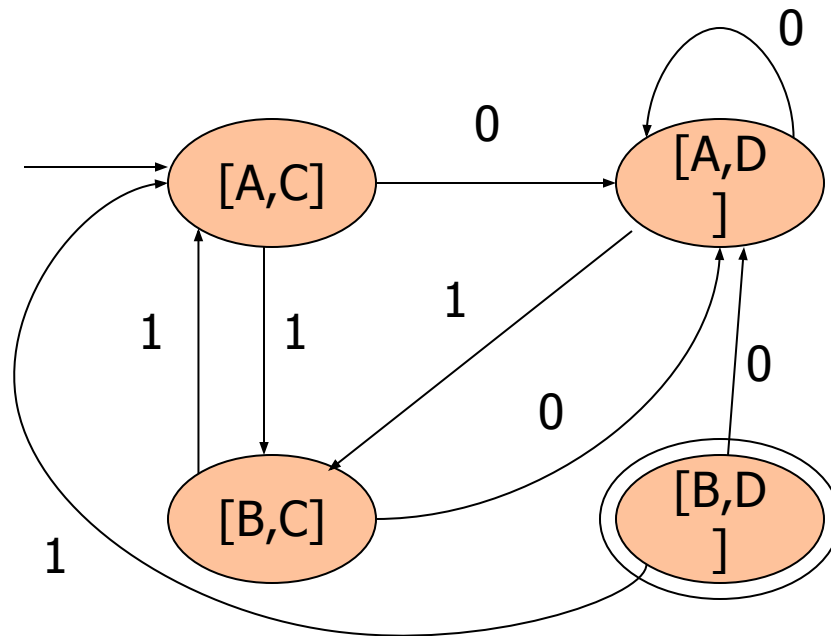
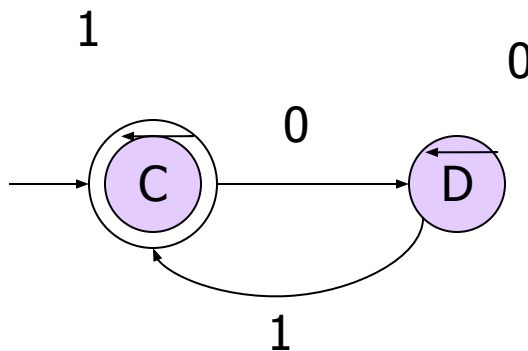
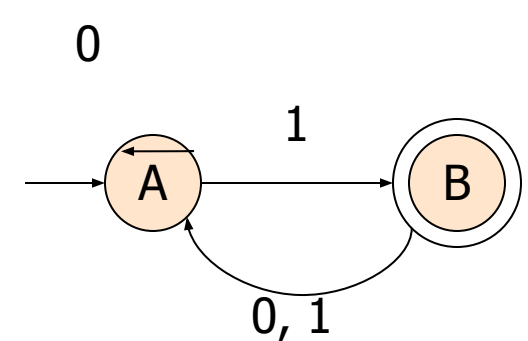


CLOSURE UNDER DIFFERENCE

- If L and M are regular languages, then so is $L - M =$ strings in L but not M .
- **Proof:** Let A and B be DFA's whose languages are L and M , respectively.
- Construct C , the product automaton of A and B .
- Make the final states of C be the pairs where A -state is final but B -state is not.



EXAMPLE: PRODUCT DFA FOR DIFFERENCE



Notice: difference
is the empty language



CLOSURE UNDER COMPLEMENTATION

- The *complement* of a language L (with respect to an alphabet Σ such that Σ^* contains L) is $\Sigma^* - L$.
- Since Σ^* is surely regular, the complement of a regular language is always regular.



CLOSURE UNDER REVERSAL

- Recall example of a DFA that accepted the binary strings that, as integers were divisible by 23.
- We said that the language of binary strings whose reversal was divisible by 23 was also regular, but the DFA construction was very tricky.
- Good application of reversal-closure.



CLOSURE UNDER REVERSAL – (2)

- Given language L , L^R is the set of strings whose reversal is in L .
- **Example:** $L = \{0, 01, 100\}$; $L^R = \{0, 10, 001\}$.
- **Proof:** Let E be a regular expression for L .
- We show how to reverse E , to provide a regular expression E^R for L^R .



REVERSAL OF A REGULAR EXPRESSION

- **Basis:** If E is a symbol a , ϵ , or \emptyset , then $E^R = E$.
- **Induction:** If E is
 - $F+G$, then $E^R = F^R + G^R$.
 - FG , then $E^R = G^R F^R$
 - F^* , then $E^R = (F^R)^*$.



EXAMPLE: REVERSAL OF A RE

- Let $E = 01^* + 10^*$.
- $E^R = (01^* + 10^*)^R = (01^*)^R + (10^*)^R$
- $= (1^*)^R 0^R + (0^*)^R 1^R$
- $= (1^R)^* 0 + (0^R)^* 1$
- $= 1^* 0 + 0^* 1.$



HOMOMORPHISMS

- A *homomorphism* on an alphabet is a function that gives a string for each symbol in that alphabet.
- **Example:** $h(0) = ab$; $h(1) = \varepsilon$.
- Extend to strings by $h(a_1 \dots a_n) = h(a_1) \dots h(a_n)$.
- **Example:** $h(01010) = ababab$.




CLOSURE UNDER HOMOMORPHISM

- If L is a regular language, and h is a homomorphism on its alphabet, then $h(L) = \{h(w) \mid w \text{ is in } L\}$ is also a regular language.
- **Proof:** Let E be a regular expression for L .
- Apply h to each symbol in E .
- Language of resulting RE is $h(L)$.



EXAMPLE: CLOSURE UNDER HOMOMORPHISM

- Let $h(0) = ab$; $h(1) = \varepsilon$.
- Let L be the language of regular expression $01^* + 10^*$.
- Then $h(L)$ is the language of regular expression $ab\varepsilon^* + \varepsilon(ab)^*$.


Note: use parentheses
to enforce the proper
grouping.



EXAMPLE – CONTINUED

- $\mathbf{ab}\epsilon^* + \epsilon(\mathbf{ab})^*$ can be simplified.
- $\epsilon^* = \epsilon$, so $\mathbf{ab}\epsilon^* = \mathbf{ab}\epsilon$.
- ϵ is the identity under concatenation.
 - That is, $\epsilon E = E\epsilon = E$ for any RE E .
- Thus, $\mathbf{ab}\epsilon^* + \epsilon(\mathbf{ab})^* = \mathbf{ab}\epsilon + \epsilon(\mathbf{ab})^* = \mathbf{ab} + (\mathbf{ab})^*$.
- Finally, $L(\mathbf{ab})$ is contained in $L((\mathbf{ab})^*)$, so a RE for $h(L)$ is $(\mathbf{ab})^*$.



INVERSE HOMOMORPHISMS

- Let h be a homomorphism and L a language whose alphabet is the output language of h .
- $h^{-1}(L) = \{w \mid h(w) \text{ is in } L\}$.



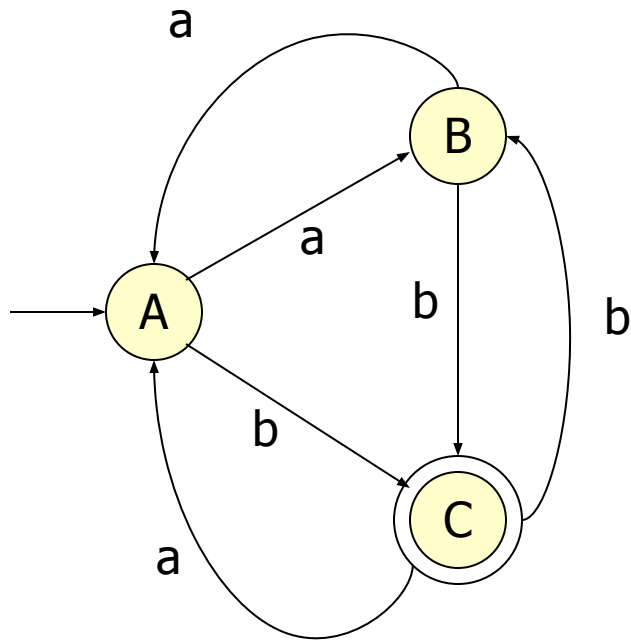
EXAMPLE: INVERSE HOMOMORPHISM

- Let $h(0) = ab$; $h(1) = \varepsilon$.
- Let $L = \{abab, baba\}$.
- $h^{-1}(L)$ = the language with two 0's and any number of 1's = $L(1^*01^*01^*)$.

Notice: no string maps to baba; any string with exactly two 0's maps to abab.

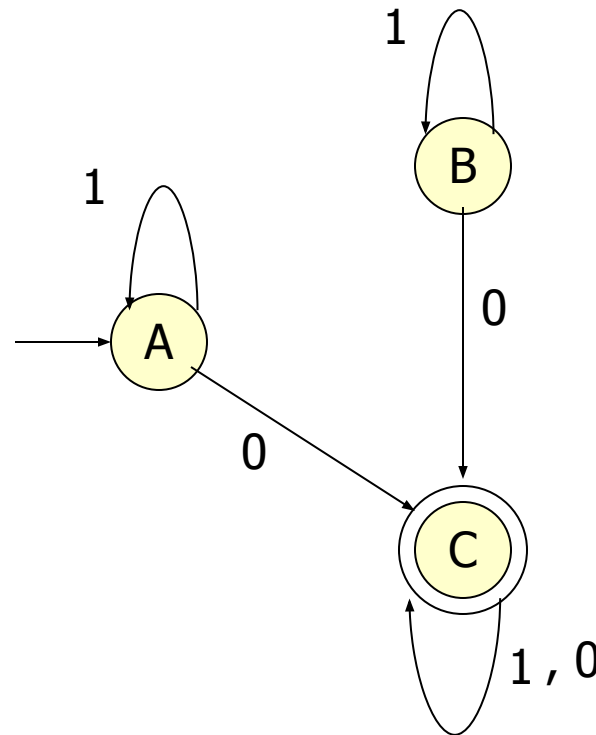


EXAMPLE: INVERSE HOMOMORPHISM CONSTRUCTION



$$h(0) = ab$$

$$h(1) = \varepsilon$$



$$\text{Since } h(1) = \varepsilon$$

$$\text{Since } h(0) = ab$$

CONTEXT FREE GRAMMAR

- 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.



INFORMAL COMMENTS

- 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

- 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.
 - α , β , γ ,... 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 – INTUITION

- 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 .



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.



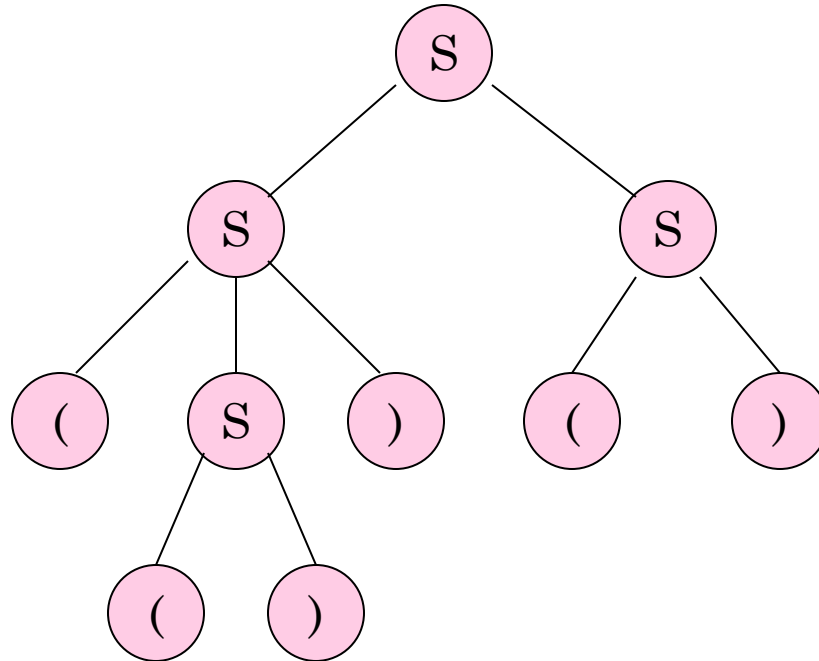
PARSE TREES

- *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.



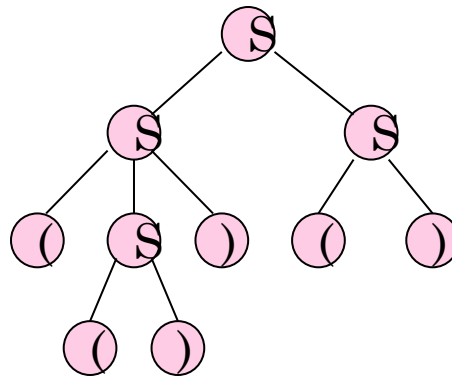
EXAMPLE: PARSE TREE

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



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, LEFT- 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 $A \Rightarrow_{lm}^* w$.
 2. If $A \Rightarrow_{lm}^* w$, then there is a parse tree with root A and yield w .



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 $A \Rightarrow^* \varepsilon$.
- **Basis**: If there is a production $A \rightarrow \varepsilon$, 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:

$S \rightarrow ABC \mid AB \mid AC \mid BC \mid A \mid B \mid C$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$



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.



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 – 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.



END

