Welcome

Course Code: CSE 217

Course Name: Theory of Computation

Credit Hour: 3.00

Instructor: Maj Mahbub

COURSE CONTENT

COURSE CONTENT
Regular language: deterministic finite automata, nondeterministic finite automata, equivalence and conversion of deterministic and nondeterministic finite automata, regular expressions, non-regular languages, the pumping lemma; **Context-free language:** Context free grammars, Chomsky normal form, Greibach Normal Form, Pushdown automata; **Turing Machines:** basic machines, configuration, computing with Turing machines, combining Turing machines; **Decidability:** decidable languages, undecidability.

Course Schedule

COURSE SCHEDULE			
Week	**Lecture**	**Topics**	**Assessment Methods**
1	Lec 1 Lec 2 Lec 3	Automata, Computability, and Complexity, Mathematical Notation and Terminology, Sets, Sequences and Tuples, Functions and Relations, Strings and Languages, Definitions, Theorems and Proofs.	Class Test 1
2	Lec 4 Lec 5 Lec 6	Finite Automata Formal Definition of a Finite Automaton Examples of Finite Automata	
3	Lec 7 Lec 8 Lec 9	Formal Definition of Computation Designing Deterministic Finite Automata	
4	Lec 10 Lec 11 Lec 12	The Regular Operations Union operation, Concatenation operation, Star operation, Closure under the Regular Operations	Class Test 2

Course Schedule

5	Lec 13 Lec 14 Lec 15	Nondeterminism Equivalence of NFAs and DFAs Closure under the Regular Operations	
6	Lec 16 Lec 17 Lec 18	Regular expressions Formal definition of a regular expression	
7	Lec 19 Lec 20 Lec 21	Nonregular Languages, The Pumping Lemma for Regular Languages.	
8	Lec 22 Lec 23 Lec 24	Context-Free Languages Context-Free Grammars Formal Definition of CFG	Mid Term Exam
9	Lec 25 Lec 26 Lec 27	Examples of CFG, Designing CFG Ambiguity	
10	Lec 31 Lec 32 Lec 33	Chomsky Normal Form I Chomsky Normal Form II	

Course Schedule

11	Lec 28 Lec 29 Lec 30	Pushdown Automata Formal Definition of a Pushdown Automaton Examples of Pushdown Automata.	Class Test 3
12	Lec 34 Lec 35 Lec 36	Non-context-free languages The pumping lemma for context-free languages and proofs	
13	Lec 37 Lec 38 Lec 39	Turning Machines, Formal Definition of a Turing Machine, Examples of Turing Machines.	
14	Lec 40 Lec 41 Lec 42	Decidability, decidable languages, Decidable problems concerning Regular languages	

Assessment Strategy

Components		Grading
Continuous Assessment (40%)	Test 1-3	20%
	Class Participation	5%
	Mid term	15%
Final Exam		60%
Total Marks		100%

ATTENDANCE POLICY

- Class attendance must be ensured within 24 hrs

Google Class Room

selcyxd

Reference Books

REFERENCE BOOKS
1. Introduction to the Theory of Computation, 3rd edition, 2012- Michael Sipser.
2. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2008 - J. E. Hopcroft, R. Motwani, and J. D. Ullman.
3. Elements of the Theory of Computation. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd edition, 1997- H. R. Lewis and C. H. Papadimitriou.

Welcome

Course Code: CSE 217

Course Name: Theory of Computation

Credit Hour: 3.00

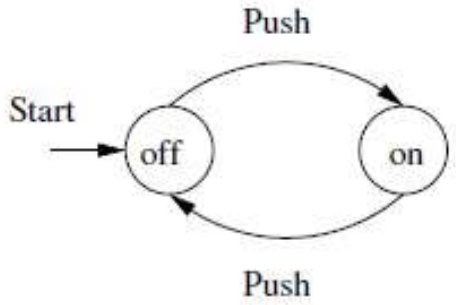
Instructor: Maj Mahbub

Outline

- Automata
- Why automata is important
- Computability and Complexity
- Set
- Relations
- Graphs
- Functions
- Strings and Languages
- Definition, Theorems, Proofs

Automata

- Theoretical Study of computer science and mathematics
- Automata Theory is a study of abstract computing devices or machines.
- Deals with logic of computation w.r.t computing devices
- Enables to understand how machines computes the functions and solve problems



- State is represented by circle e.g *on* and *off*
- Arcs represent inputs e.g *Push*
- One of the state is designated as start state
- Final/accepted state is represented by double circle

Why Automata is important?

- Allows to understand how machines solve problems
- Software for designing and checking the behavior of digital circuit
- Designing of lexical analyzer of compiler
- Web scrapping and Searching patterns
- Software for verifying systems that have finite number of states

Computability and Complexity

- Computability refers to the problems that are computable by the computing devices.
- Complexity- Easy or Hard
- Computability- Solvable or Unsolvable

Set

- Defined by as an unordered collection of definite and distinct objects
- Objects are element/member of a set

  $x \in A$, x is member of set A

  $x \notin A$, x is not member of set A

  Empty set=Φ

  Universal set= U

  Power set = P(X)

Set Operations

- $X \subseteq Y,\ X\ is\ a\ subset\ of\ Y$
- $X \cup Y = x \in (X \cup Y)\ if\ and\ only\ if\ (x \in X)\ or\ (x \in Y)$
- $X \cap Y = \Phi, if\ X\ and\ Y\ are\ disjoint$
- A' is the set of all elements that are in the universal set U but are not in A.
- $A - B$- *Subtraction*
- $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ *-Distributive*
- $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
- *De Morgan* $\overline{A \cup B} = \overline{A} \cap \overline{B},$

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

Relation

- *Connection or association of objects*
- *Tuple*
  - *Ordered collection of objects*
  - *Set whose objects are given in a specified order*
  - *N- tuple of n elements as* $(x_1, x_2, \ldots\ldots x_n)$
  - *2-tuple- ordered pair*
  - *3-tuple- triple*

- *Two tuples* $(x_1, x_2, \ldots\ldots x_n)$ *and* $(y_1, y_2, \ldots\ldots y_n)$ *are said equal if* $x_i = y_i$ *for all i:* $1 \leq i \leq n$

Graph

- Graph G is a pair of two sets <V,E>
- V={v1,v2,…vn}-non-empty (nodes/vertices)
- E=multi set of two vertices (edges)

  $E=\{<v_i,v_j> | v_i,v_j \in V \text{ and } <v_i,v_j>=<v_j,v_i>\}$

  If $v_i=v_j$, then it is called self loop

Self Study: Directed graph, Undirected graph, Simple graph, Finite graph, Infinite graph, Subgraph, Spanning subgraph, Isomorphism, Eulerian path, Hamilton path

Function

- Object that sets up input-output relationship
- f(a)=b
- a is input value and b is the output for a
- Set of possible inputs- Domain
- Set of outputs- Range
- $f: D \rightarrow R$

Strings and Languages

- String is fundamental building block of CS
- Alphabet: non-empty finite set . The members of alphabet are the symbols of alphabet
- Alphabet is denoted by $\sum$; $\sum$={0,1},{a,b,c}
- Empty String: zero occurrences of symbols. Denoted by $\in$
- Length of String: no. of positions for symbols in the string

  |s|=|01110|=5
- Power of alphabet: set of all strings of certain length
- Denoted by $\sum^k$; where k is length of string
- $\sum^0 = \in$
- $\sum$={a,b,c}, $\sum^2$ ={aa, ac, bb, ac, …….}
- $\sum^*$={0,1}*={$\in$, 0,1,00,11,10,101, 0001,……..}

Strings and Languages

- Concatenation of string: x and y be strings then xy denotes concatenation of x and y
- Length of xy= length(x)+length(y)
- Lexicographic order: ~ is same as dictionary order

Definition, Theorems, Proofs

- Definitions: ~ describes object and notions that we use. Precision is essential to any mathematical definitions
- Proof: ~ is convincing logical argument that a statement is true
- Theorem: ~ is a mathematical statement proved true

Languages

- A set of strings that are chosen from $\sum^*$ where $\sum$ is a particular alphabet
- $L \subseteq \sum^*$, then L is language
- Language of all strings consisting of n 0's followed by n 1's for some $n \geq 0$ {∈, 01,00111,000111…….}
- Set of strings of 0's and 1's with equal no of each {∈, 01,10,0011,0101,……}
- Set of binary number whose value is a prime {10,11,101,111……..}
- {∈}, the language consisting of only empty string

Reference Books

REFERENCE BOOKS
1. Introduction to the Theory of Computation, 3rd edition, 2012- Michael Sipser.
2. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2008 - J. E. Hopcroft, R. Motwani, and J. D. Ullman.
3. Elements of the Theory of Computation. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd edition, 1997- H. R. Lewis and C. H. Papadimitriou.

Welcome

Course Code: CSE 217

Course Name: Theory of Computation

Credit Hour: 3.00

Instructor: Maj Mahbub

Outline

- Finite Automata
- Formal Definition of Finite Automata
- Examples of Finite Automata

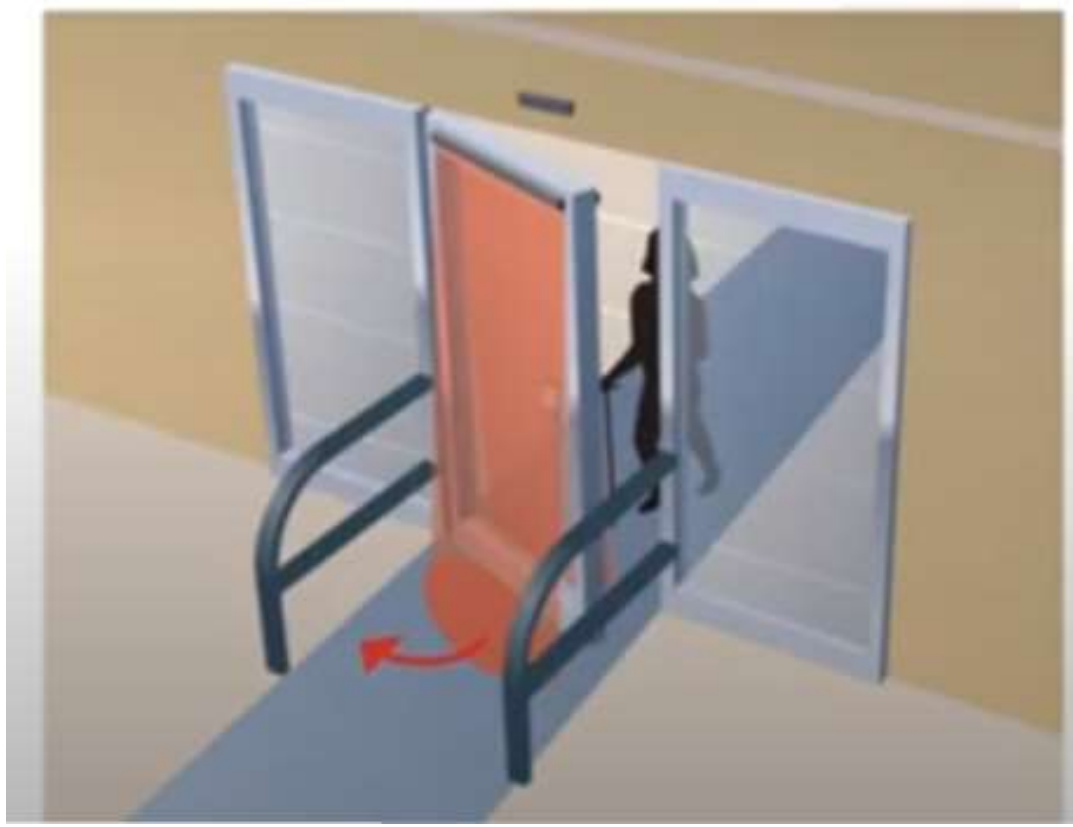
Introduction

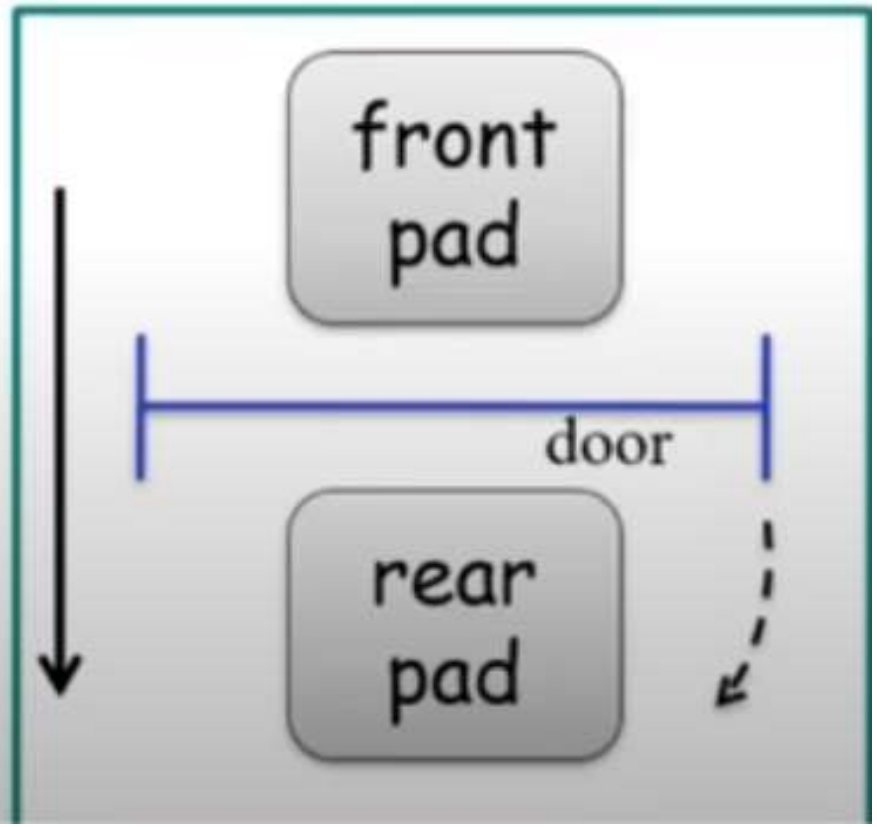
- Computer
  - Use computational model
  - Complicated in computation
  - Finite state machine/finite automata

Finite Automata

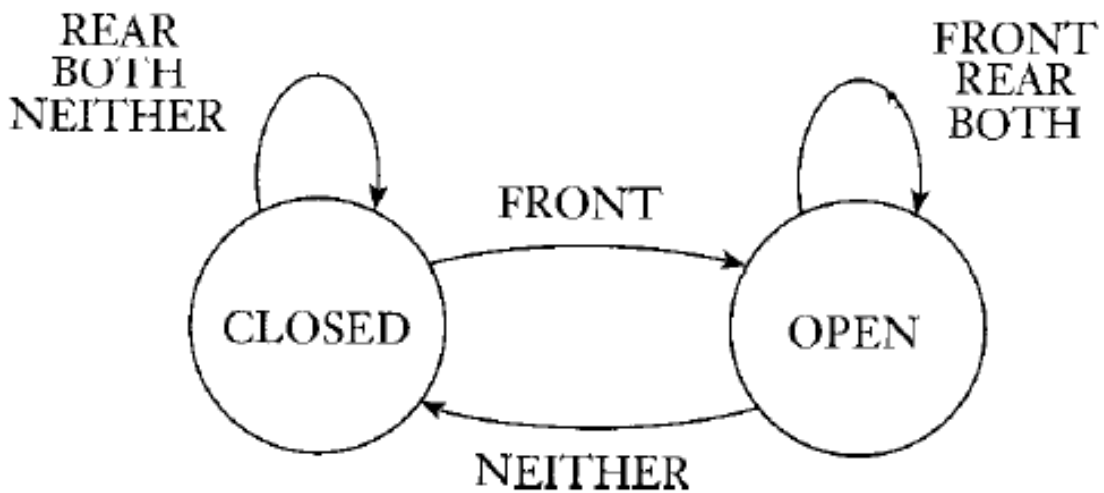
- Good for computer for using less amount of memory and time
- Automatic Door Controller
  - Two states: open or closed
  - Four possible input conditions
    - Front- a person is standing on the front pad sensor
    - Rear-a person is standing on the rear pad sensor
    - Both- people are standing both of the doorways
    - Neither- no one standing on either pad

Automatic Door Controller



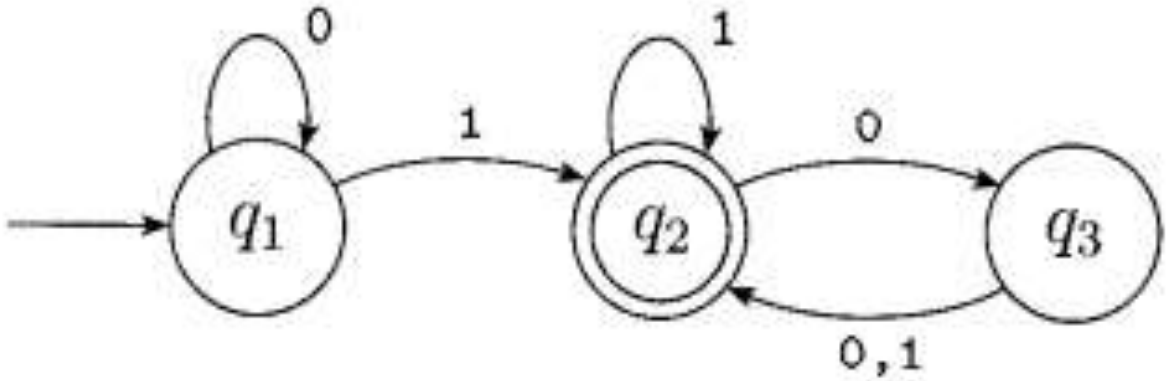


Automatic Door Controller



state	input signal: NEITHER	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

Finite Automata



- Finite automata M
- 3 state, start state q_1 , accepted state q_2
- Arrows indicate transition between states
- Output is accept or reject
- Accepting strings- 001, 11, 01010101,…..
- Accept any strings that ended with 1
- Accept any strings that end with even number of 0's -100, 00100, 10000
- Rejects other strings

Formal Definition Finite Automata

- Defined by using 5-tuple $(Q, \Sigma, \delta, \mathrm{q}_{0,} \mathrm{F})$

  $Q = finite\ set\ of\ states$

  $\Sigma = alphabet$

  $\delta = \mathrm{Q} \times \Sigma \rightarrow Q\ (transition\ function)$

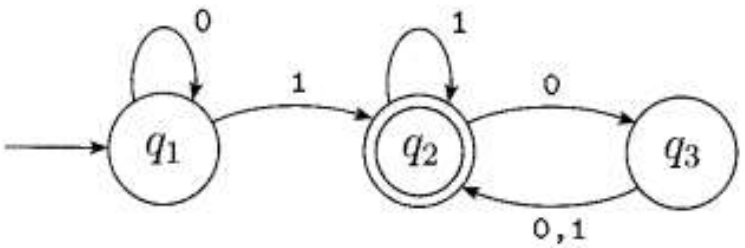
  $q_0 = start\ state$

  $F = final\ state, F \subseteq Q$

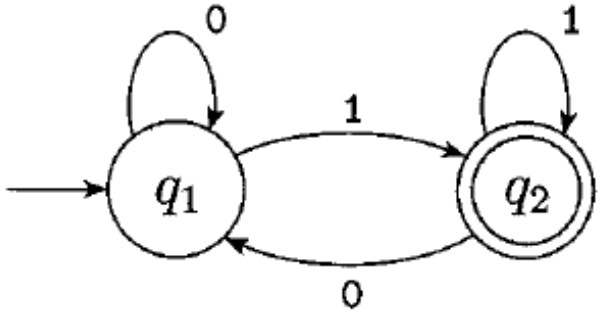
- Describe M formally: ????
- Language of M: If A is the set of all strings that accepted by M then A is the language of the machine M. $L(M) = A$
- A machine may accept several language but always recognize only one language
- Machine that accept no strings, it still recognize one language – empty language ϕ

Examples Finite Automata

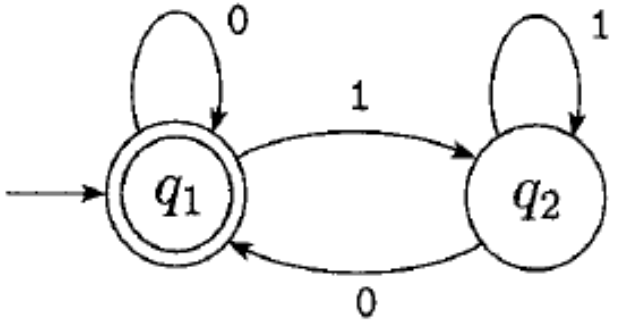
- A={w|w contains at least one 1 and d an even number of 0s after last 1}



- A={w|w ends in a 1}

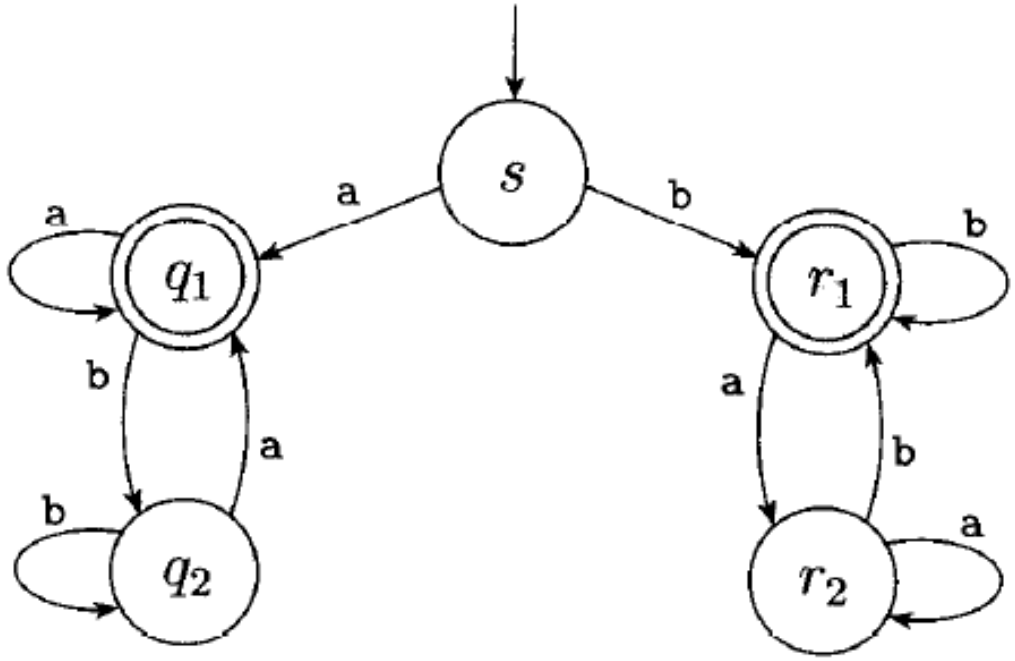


- A={w|w is the empty string or ends in a 0}



Examples Finite Automata

- Design a automata over an alphabet $\Sigma = \{a, b\}$ that start and ends with same symbol
- Sample accepted strings: a, b, aa, bb, aba, bab…. Etc
- Non accepted strings: ab, ba, bbba etc



Applications of Deterministic Finite Automata

- Pattern matching: DFAs can be used to search for patterns in text or data. For example: Recognize email addresses or phone numbers in a block of text, or to identify certain types of programming constructs in source code.
- Lexical analysis: Lexical analysis is the process of identifying the tokens in a source code file. Recognize the different types of tokens, such as keywords, identifiers, and operators.
- Network security: DFAs can be used to recognize and block malicious network traffic. For example, a DFA can be used to identify known malware signatures or suspicious network activity, and can trigger an alert or block the traffic.

Applications of Deterministic Finite Automata

- DNA sequencing: DFAs can be used to analyze DNA sequences and identify patterns that correspond to specific genetic traits or diseases. By constructing a DFA that recognizes a particular sequence, researchers can more easily identify the presence of that sequence in a DNA sample.
- Natural language processing: DFAs can be used in natural language processing to recognize patterns in human language. For example, a DFA can be used to identify named entities like people, places, or organizations in a block of text, or to recognize certain types of grammar constructs.

Reference Books

REFERENCE BOOKS
1. Introduction to the Theory of Computation, 3rd edition, 2012- Michael Sipser.
2. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2008 - J. E. Hopcroft, R. Motwani, and J. D. Ullman.
3. Elements of the Theory of Computation. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd edition, 1997- H. R. Lewis and C. H. Papadimitriou.

Welcome

Course Code: CSE 217

Course Name: Theory of Computation

Credit Hour: 3.00

Instructor: Maj Mahbub

Outline

- Formal Definition of Computation
- Designing of Deterministic Finite Automata

Formal Definition of Computation

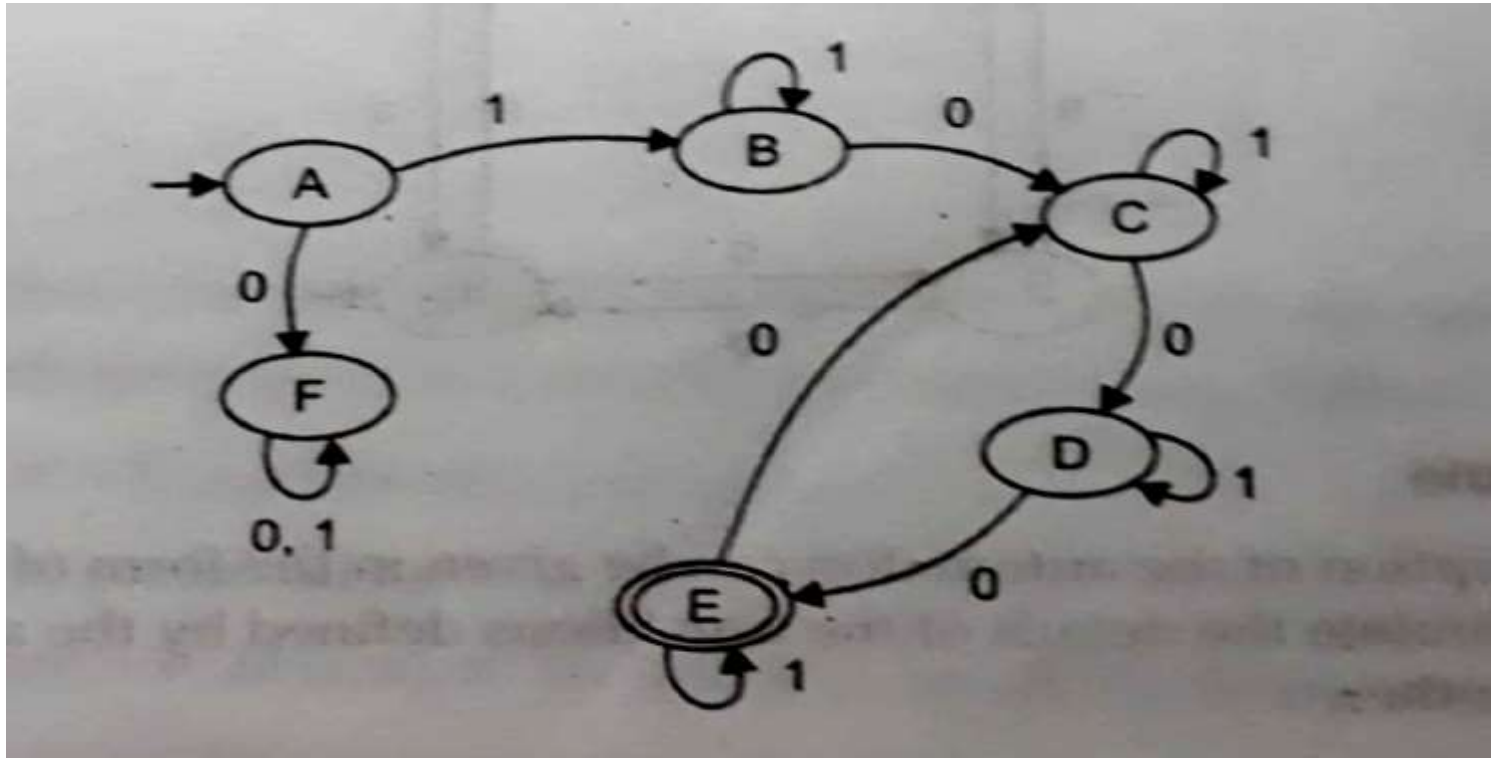
- Let M =($Q, \Sigma, \delta,$ $\mathrm{q}_{0,}$ F) be a finite automata and let $w = w_1, w_2, w_3 \ldots \ldots wn$ be a string, w_k is a member of the alphabet Σ, then M accept w if a sequence of states $r_0, r_1, r_2 \ldots . r$ in Q exist with the following three conditions:
  - $r_0 = q_0$
  - $\delta(\mathrm{r_i},\ w_{i+1}) = r_{i+1}$for $i = 0,1,2,3 \ldots . . n - 1$
  - $r_n \epsilon F$

  - Condition 1 states that machine start with the start state
  - Condition 2 states that the machine goes state to state according to the transition function
  - Condition 3 states that the machine accepts its input

- We can say that M recognizes A if $A = \{w | M\ accepts\ w\}$

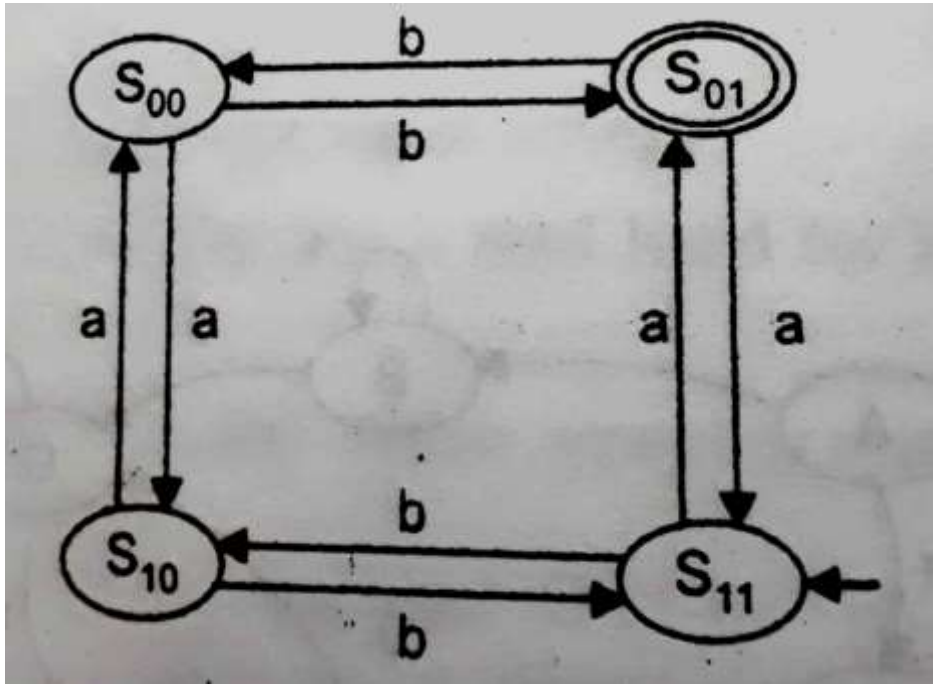
Deterministic Finite Automata

- Design a automata over an alphabet $\Sigma = \{0,1\}$ that start with 1 such that the number of 0s is divisible by 3.
- Sample accepted strings: 1000, 11000000, 101010 etc
- Non accepted strings: 100, 010, 10000 etc



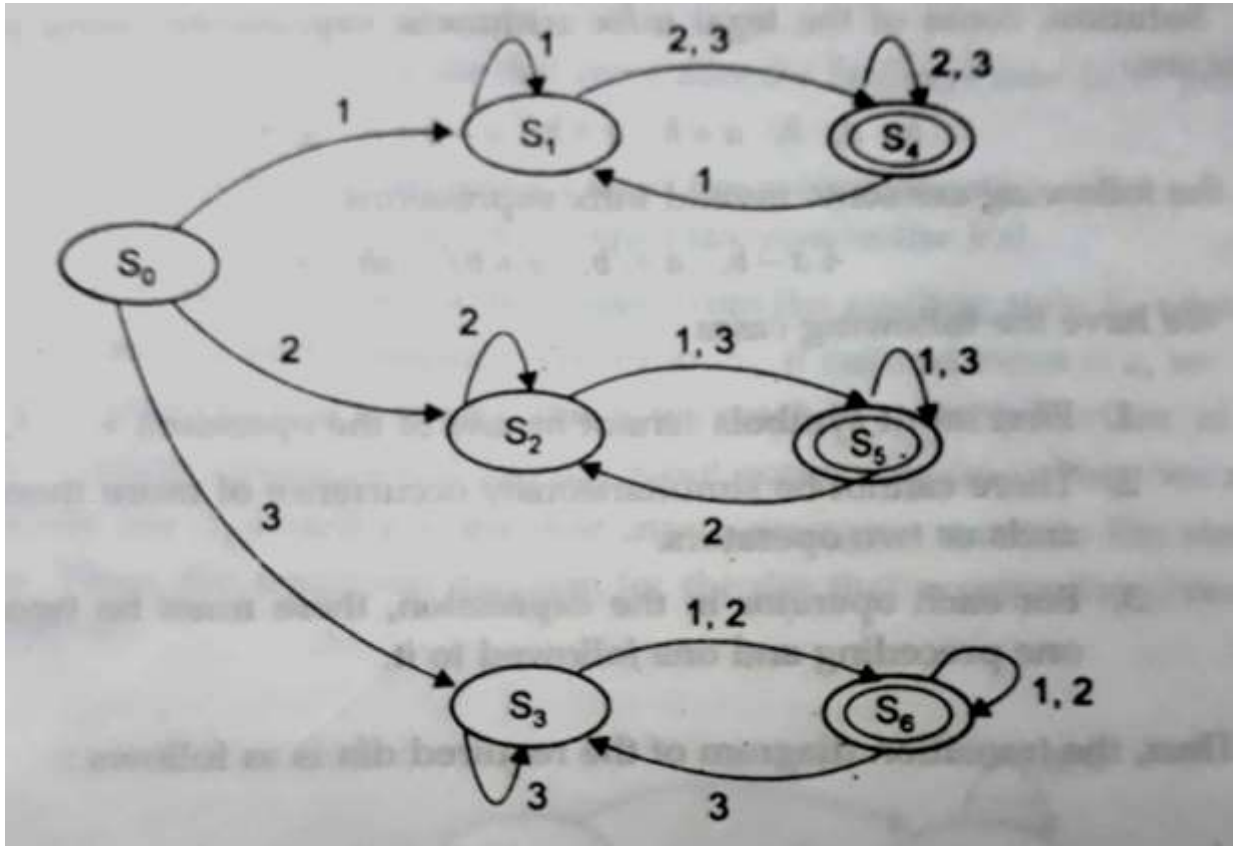
Deterministic Finite Automata

- Construct a DFA over an alphabet $\Sigma = \{a, b\}$ that accept the language: $\{x \epsilon \{a, b\}: |x|_a = odd \ and \ |x|_b = even\}$
- Sample accepted strings: abb, aaabbbb, aaabb etc
- Non accepted strings: ab, aabb, etc
- Four possible state: (odd,odd), (odd,even), (even,odd), (even,even)
- States can be denoted by S_{00}, S_{01}, S_{10} and S_{11}



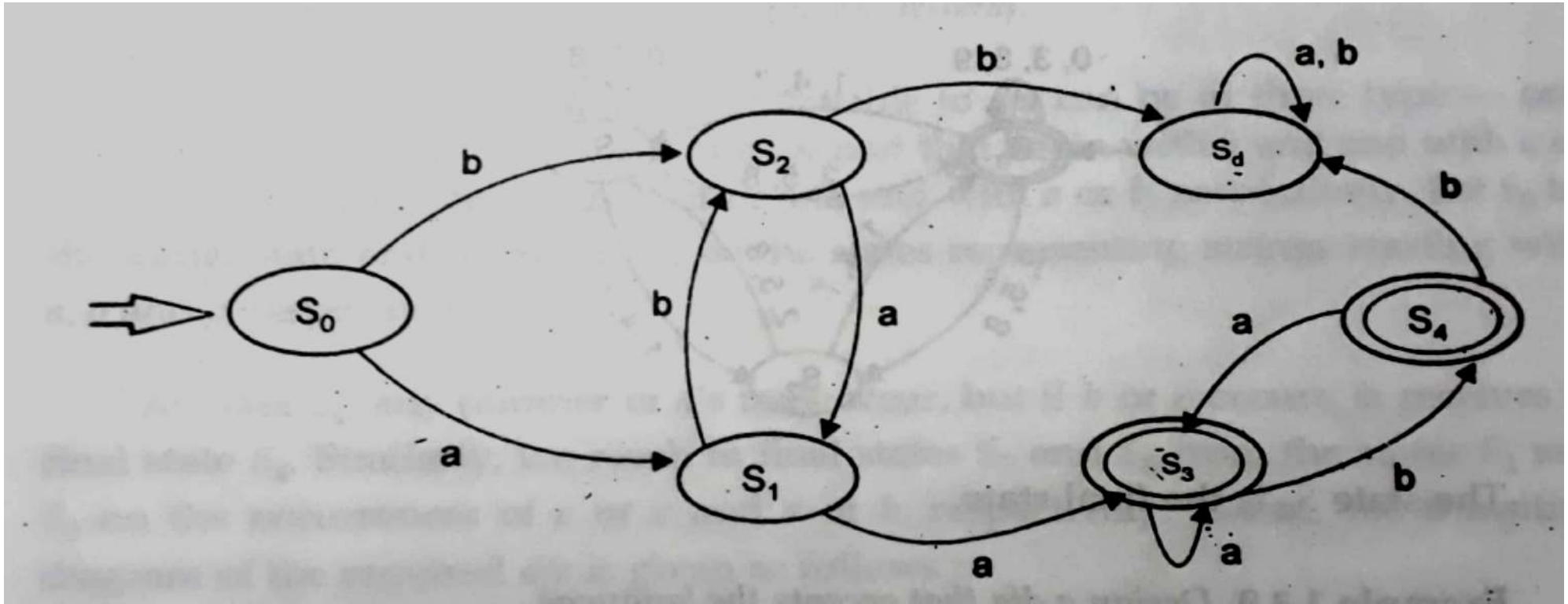
Deterministic Finite Automata

- Construct a DFA over an alphabet $\Sigma = \{1,2,3\}$ that accept the language: $\{x\epsilon\ \{1,2,3\}: x\ begin\ and\ end\ with\ differnt\ symbols\}$
- Sample accepted strings: 123, 132, 1112233, 321, 213 etc
- Non accepted strings: 11221,121, 131, 2312 etc



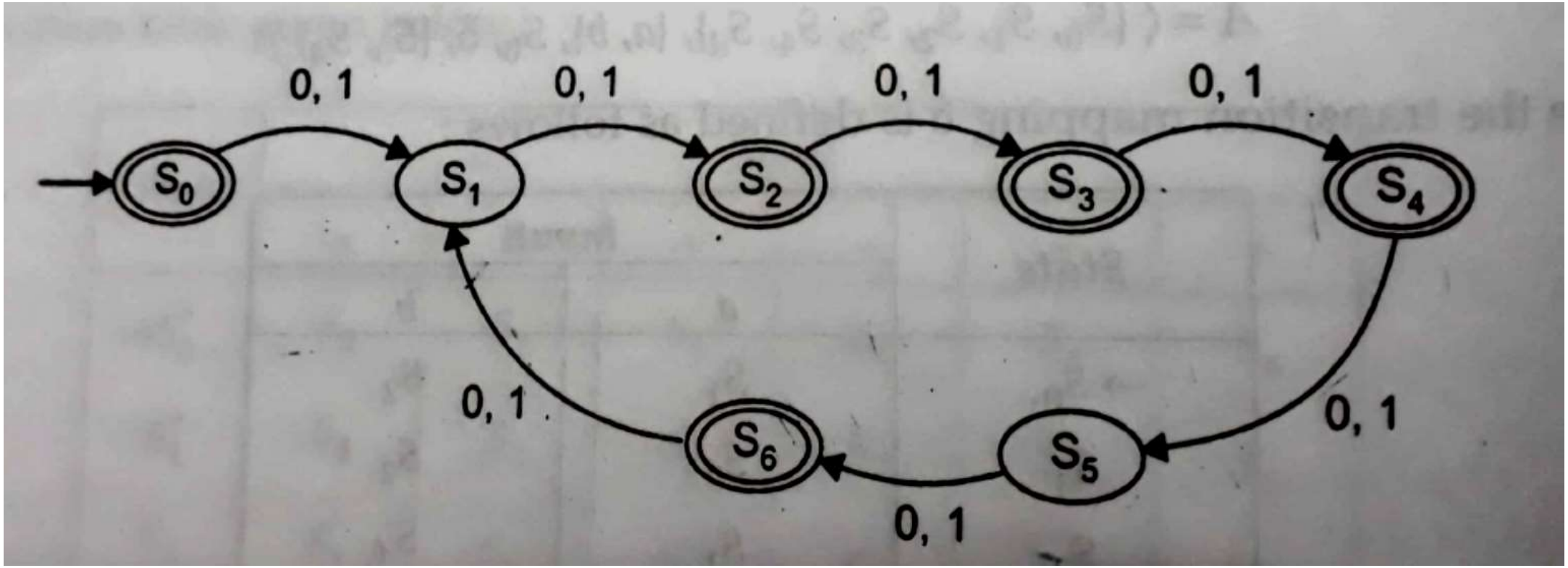
Deterministic Finite Automata

- Construct a DFA over an alphabet $\Sigma = \{a, b\}$ that accept the language: $\{x \epsilon \{a, b\}: x\ contains\ at\ least\ two\ consecutive\ a's\ and\ x\ does\ not\ contain\ two\ consecutive\ b's\}$
- Sample accepted strings: aa, aab, abaa, babaa, aaabaab etc
- Non accepted strings: bb, abb, aabb, abab, aba etc



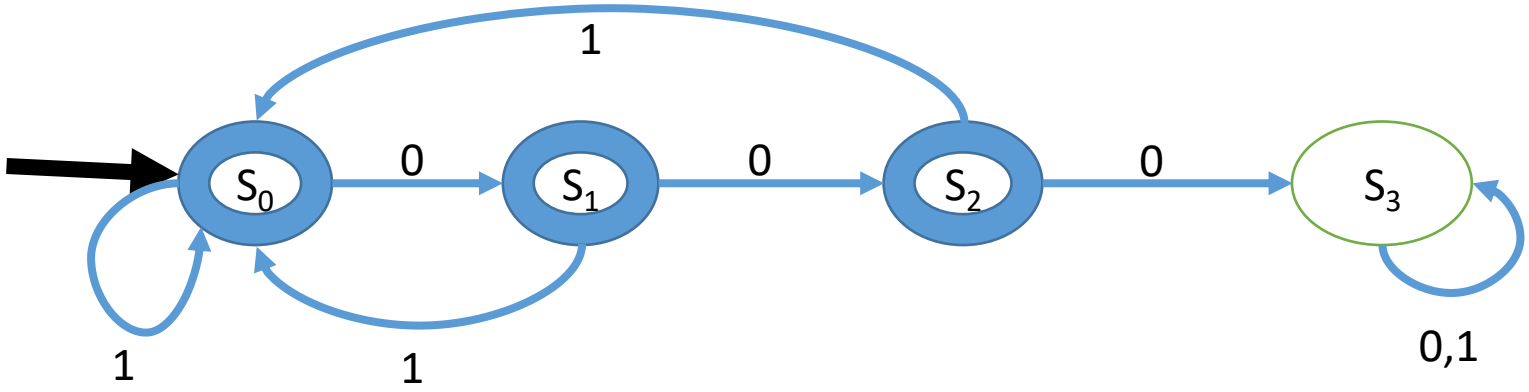
Deterministic Finite Automata

- Construct a DFA over an alphabet $\Sigma = \{0,1\}$ that accept the language: $\{x \epsilon \{0,1\}: x \; is \; multiple \; of \; 2 \; or \; 3\}$
- Sample accepted strings: 00,11, 01, 10, 000,111, 101 ,010 etc
- Non accepted strings: 0,1, 01011, etc



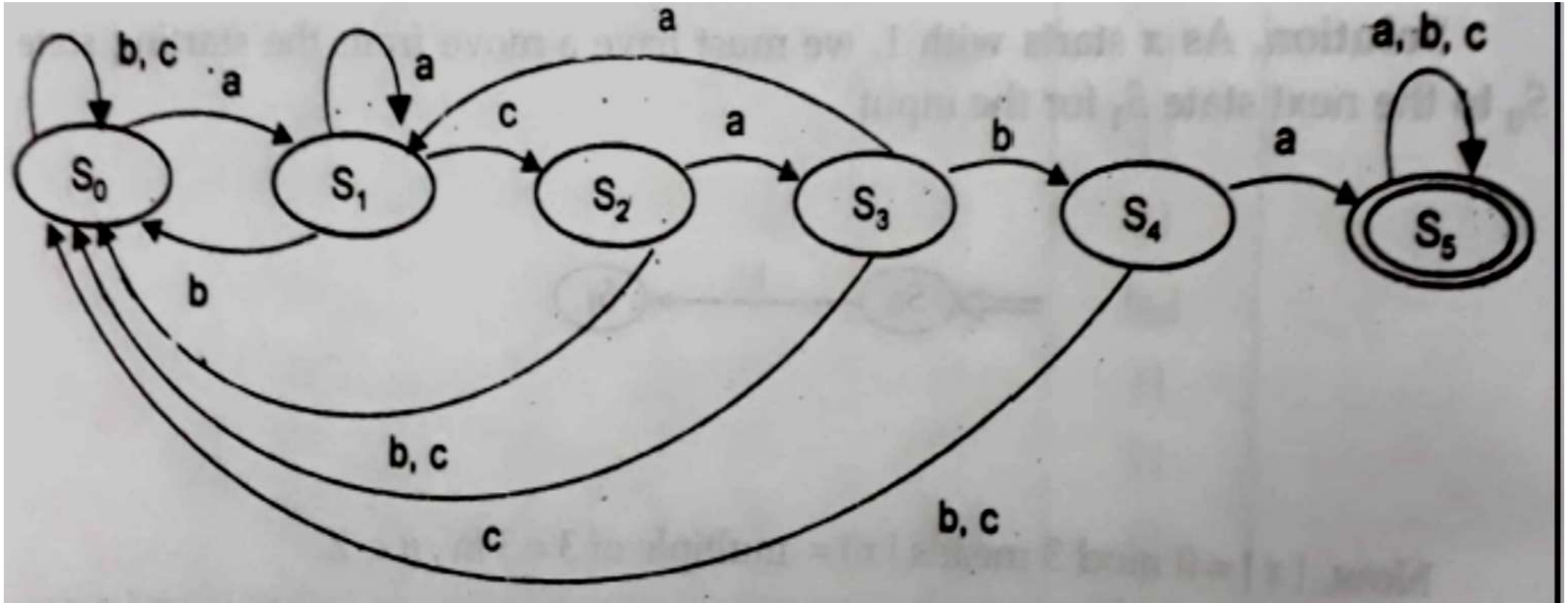
Deterministic Finite Automata

- Construct a DFA over an alphabet $\Sigma = \{0,1\}$ that accept the language: $\{x \epsilon \{0,1\}: x \ \ do\ not\ contain\ three\ consequtive\ 0's\}$
- Sample accepted strings: 0,1, 00,11, 01, 10, 0010,111, 101 ,010 etc
- Non accepted strings: 0001,1000, 00100011, etc



Deterministic Finite Automata

- Construct a DFA over an alphabet $\Sigma = \{a, b, c\}$ that accept the language: $\{x \epsilon \{a, b, c\} : x\ contain\ substring\ acaba\}$
- Sample accepted strings: acaba, bbacabacc, acabaccbbaetc
- Non accepted strings: acacbaa,acaabca etc



Reference Books

REFERENCE BOOKS
1. Introduction to the Theory of Computation, 3rd edition, 2012- Michael Sipser.
2. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2008 - J. E. Hopcroft, R. Motwani, and J. D. Ullman.
3. Elements of the Theory of Computation. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd edition, 1997- H. R. Lewis and C. H. Papadimitriou.

Welcome

Course Code: CSE 217

Course Name: Theory of Computation

Credit Hour: 3.00

Instructor: Maj Mahbub

Outline

- Regular Operation
- Union, Concatenation and Star Operation
- Closure under regular Operation

Formal Definition of Computation(Recap)

- Let M =$(Q, \Sigma, \delta, \mathrm{q}_{0,} \mathrm{F})$ be a finite automata and let $w = w_1, w_2, w_3 \ldots \ldots wn$ be a string, w_k is a member of the alphabet Σ, then M accept w if a sequence of states $r_0, r_1, r_2 \ldots . r$ in Q exist with the following three conditions:
  - $r_0 = q_0$
  - $\delta(\mathrm{r_i},\ w_{i+1}) = r_{i+1}$for $i = 0,1,2,3 \ldots . . n - 1$
  - $r_n \epsilon F$

  - Condition 1 states that machine start with the start state
  - Condition 2 states that the machine goes state to state according to the transition function
  - Condition 3 states that the machine accepts its input

- We can say that M recognizes A if $A = \{w | M\ accepts\ w\}$

SAMPLE DFA

EXAMPLE 1.5

The following diagram shows machine M_5, which has a four-symbol input alphabet, $\Sigma = \{\langle \text{RESET} \rangle, 0, 1, 2\}$. We treat $\langle \text{RESET} \rangle$ as a single symbol.

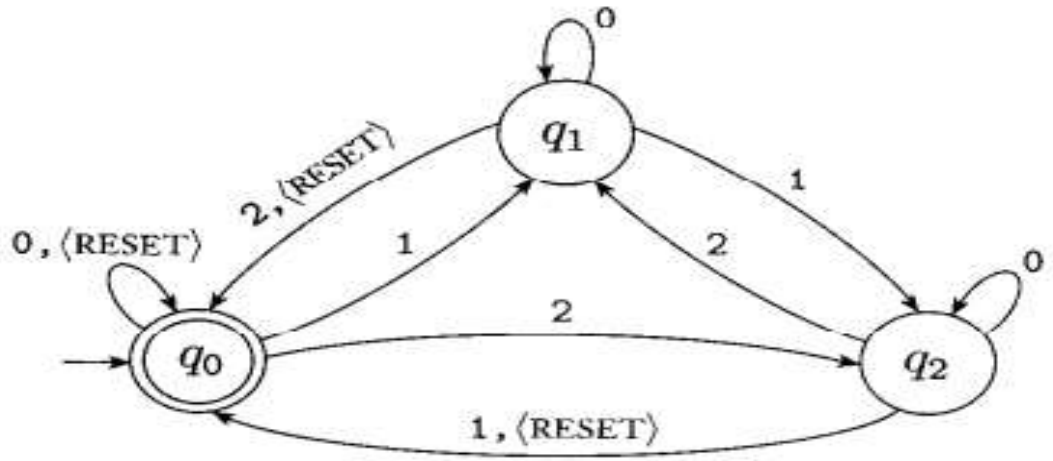


FIGURE 1.9
Finite automaton M_5

M_5 keeps a running count of the sum of the numerical input symbols it reads, modulo 3. Every time it receives the $\langle \text{RESET} \rangle$ symbol it resets the count to 0. It accepts if the sum is 0, modulo 3, or in other words, if the sum is a multiple of 3.

PROOF

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

Take machine M_5 from Example 1.5. Let w be the string

$$10\langle\text{RESET}\rangle 22\langle\text{RESET}\rangle 012$$

Then M_5 accepts w according to the formal definition of computation because the sequence of states it enters when computing on w is

$$q_0, q_1, q_1, q_0, q_2, q_1, q_0, q_0, q_1, q_0,$$

which satisfies the three conditions. The language of M_5 is

$$L(M_5) = \{w | \text{ the sum of the symbols in } w \text{ is 0 modulo 3, except that } \langle\text{RESET}\rangle \text{ resets the count to 0}\}.$$

As M_5 recognizes this language, it is a regular language.

The Regular Operation

- To investigate the properties of DFA
- To investigate the regular language of DFA
- Used as toolbox for manipulating the language of FA
  - Let $A\ and\ B$ be languages then we can define the following three regular operations as:
    i. Union: $AUB = \{x|x\epsilon A\ or\ x\epsilon B\}$
    ii. Concatenation: $A \circ B = \{xy|x\epsilon A\ and\ y\epsilon B\}$
    iii. Star: $A^{\star} = \{x_1, x_2, x_3 \ldots . x_k|k \geq 0\ and\ each\ x_i\epsilon A\}$
- Union and Concatenation are binary operation and Star operation is unary operation

The Regular Operation

Union: $AUB = \{x | x \epsilon A \ or \ x \epsilon B\}$

Concatenation: $A \circ B = \{xy | x \epsilon A \ and \ y \epsilon B\}$

Star: $A^{\star} = \{x_1, x_2, x_3 \ldots . x_k | k \geq 0 \ and \ each \ x_i \epsilon A\}$

- Union and Concatenation are binary operation and Star operation is unary operation

Let the alphabet Σ be the standard 26 letters $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. If $A = \{\mathtt{good}, \mathtt{bad}\}$ and $B = \{\mathtt{boy}, \mathtt{girl}\}$, then

$A \cup B = \{\mathtt{good}, \mathtt{bad}, \mathtt{boy}, \mathtt{girl}\}$,

$A \circ B = \{\mathtt{goodboy}, \mathtt{goodgirl}, \mathtt{badboy}, \mathtt{badgirl}\}$, and

$A^* = \{\varepsilon, \mathtt{good}, \mathtt{bad}, \mathtt{goodgood}, \mathtt{goodbad}, \mathtt{badgood}, \mathtt{badbad}, \mathtt{goodgoodgood}, \mathtt{goodgoodbad}, \mathtt{goodbadgood}, \mathtt{goodbadbad}, \ldots\}$.

The Regular Operation

- Prove that the class of regular languages closed under union operation or If $A\ and\ B$ are regular languages , so is $A \cup B$
- Proof Idea: Let two finite automata $M_1\ and\ M_2$ recognize $A1\ and\ A2$ respectively, then another FA M will recognizes $A_1 \cup A_2$

The Regular Operation

- Let M_1 recognize A_1 where M_1=(Q_1 $\Sigma, \delta_1, \mathrm{q}_{1,} \mathrm{F}_1$) and M_2 recognize A_2 where M_2=(Q_2 $\Sigma, \delta_2, \mathrm{q}_{2,} \mathrm{F}_2$), Now construct M to recognize $A_1 \cup A_2$ where M =($Q, \Sigma, \delta, \mathrm{q}_{0,}$ F)

1. $Q = \{(r_1, r_2) | r_1 \epsilon Q_1 \; and \; r_2 \epsilon Q_2\}$; Cartesian product of $Q_1 \; and \; Q_2 \; ie. \; Q_1 \times Q_2$
2. Σ is same for $M_1 and \; M_2$ (For different alphabet $\Sigma_1$and Σ_2 let $\Sigma = \Sigma_1 \cup \Sigma_2$)
3. δ, the transition function is defined by: for each (r_1, r_2) ϵ Q and each a ϵ Σ, let $\delta\big((\mathrm{r}_1, \mathrm{r}_2)\mathrm{a}\big) = (\delta_1(\mathrm{r}_1, \mathrm{a}), \delta_2(\mathrm{r}_2, \mathrm{a}))$ δ gets a state of M which actually is a pair of states from M_1 and M_2 together with an input symbol and returns $M's$ next state
4. $q_0 \; is \; the \; pair \; (q1, q2)$
5. F is the set of pairs in which either members is an accepted state of $M1$ or $M2$. We can write as follows: $F = \{(r_1, r_2 | r_1 \epsilon F_1 \; or \; r_2 \epsilon F_2)\}$

Reference Books

REFERENCE BOOKS
1. Introduction to the Theory of Computation, 3rd edition, 2012- Michael Sipser.
2. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2008 - J. E. Hopcroft, R. Motwani, and J. D. Ullman.
3. Elements of the Theory of Computation. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd edition, 1997- H. R. Lewis and C. H. Papadimitriou.

Welcome

Course Code: CSE 217

Course Name: Theory of Computation

Credit Hour: 3.00

Instructor: Maj Mahbub

Outline

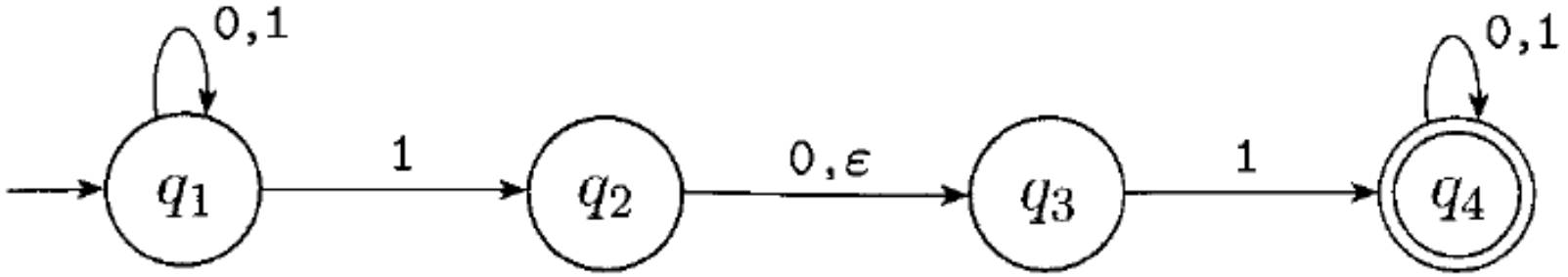
- Non determinism
- Equivalence of NFAs and DFAs
- Closure under regular Operation

Nondeterminism

- In DFA for a given state, we know what will be the next state that's why it is called deterministic computation
- In non deterministic computation several choices exist for the next state at any point
- NFA is automatically a DFA
- Difference:
  - DFA has only one exiting transition arrow for each symbol in the alphabet
  - NFA may have several exiting transition arrow for each symbol in the alphabet (0, 1 or more)
  - In DFA, labels on the transition arrows are symbols from the alphabet
  - NFA may have arrows labelled with member of alphabet or ϵ

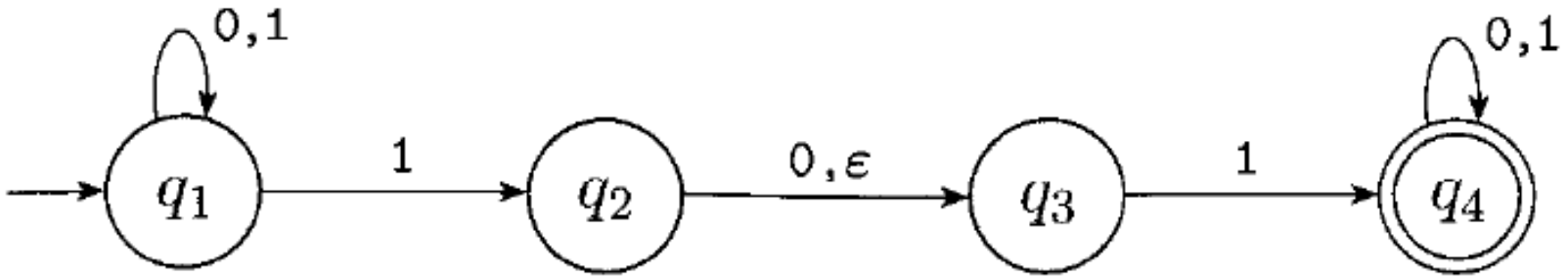
Nondeterminism

- This NFA takes input string and may proceed multiple states.
- From q_1 after getting input 1 the machine splits into two copies of itself and follows all the possibilities in parallel. If it gets subsequent choices, it spilt again. If the next input symbol does not appear on any of the arrows exiting the state occupied by a copy of the machine , the copy of the machine dies along with the branch of computation.

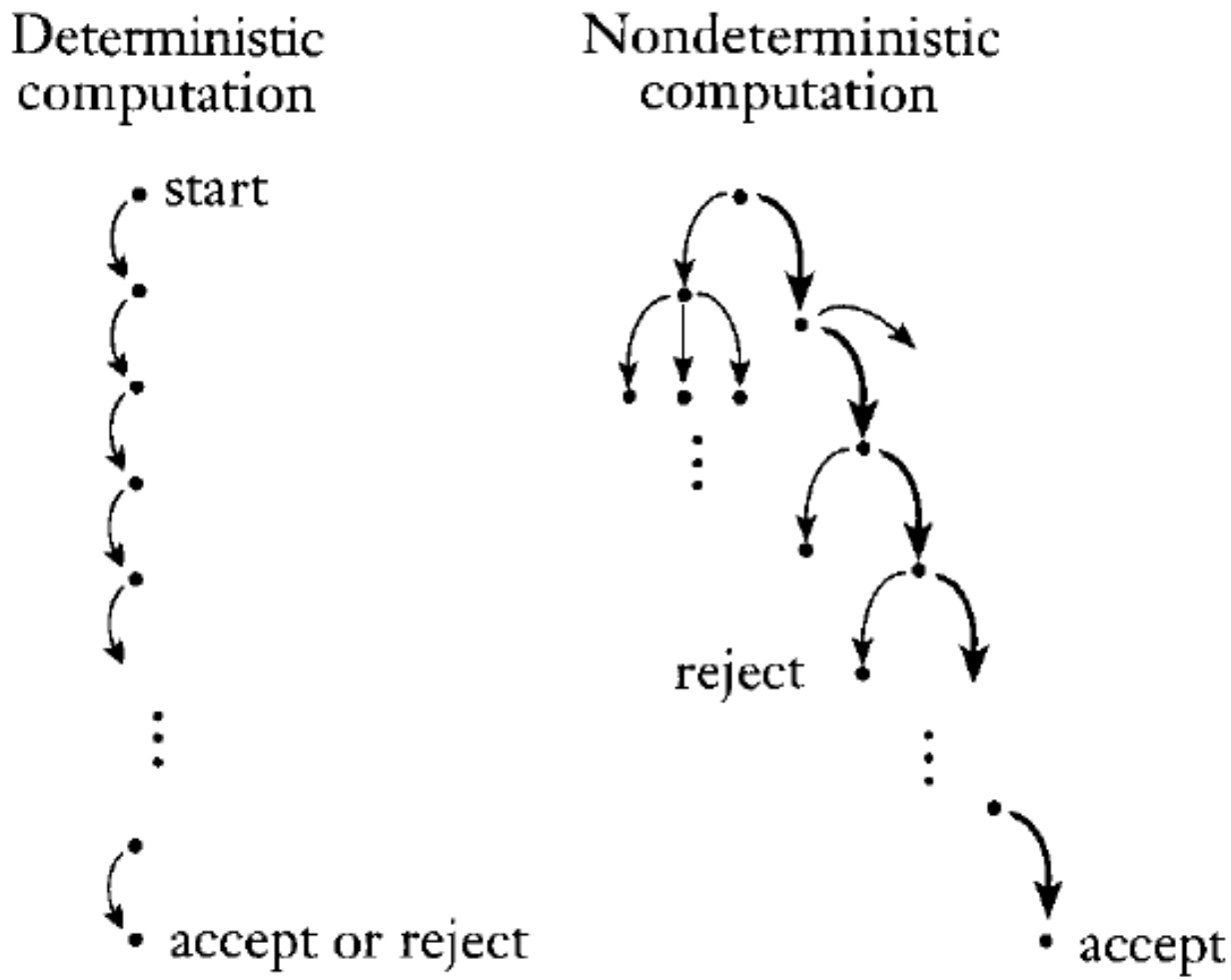


Nondeterminism

- If any of these copies of the machine is in accepted state at the end of the input the NFA accept the string.
- If ϵ is encountered , machine spits into multiple copies one following each of the exiting arrows with ϵ –label and one staying at the current state.
- Parallel computation



Nondeterminism



Formal Definition of NFA

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

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

- In NFA, transition function takes a state and an input symbol of the empty string and produces the set of possible states
- $P(Q) - collection\ of\ all\ subsets\ of\ Q$
- $\Sigma\ \cup \{\epsilon\} = \Sigma_\epsilon$
- $\delta{:}Q \times \Sigma_\epsilon \longrightarrow P(Q)$

Formal Definition of Computation of NFA

- Same as DFA
- Let M =($Q, \Sigma, \delta,$ $\mathrm{q}_{0,}$ F) be a finite automata and let $w = w_1, w_2, w_3 \ldots \ldots w_n$ be a string, w_k is a member of the alphabet Σ, then M accept w if a sequence of states $r_0, r_1, r_2 \ldots . r$ in Q exist with the following three conditions:
  - $r_0 = q_0$
  - $\delta(\mathrm{r_i},\ w_{i+1}) = r_{i+1}$ for $i = 0,1,2,3 \ldots . . n - 1$
  - $r_n \epsilon F$

  - Condition 1 states that machine start with the start state
  - Condition 2 states that the machine goes state to state according to the transition function
  - Condition 3 states that the machine accepts its input

- We can say that M recognizes A if $A = \{w | M\ accepts\ w\}$

Equivalence of NFA and DFA

- Every NFA can be turned into an equivalent DFA
- NFA is more powerful than DFA, so NFA recognize more language
- For a given language, describing with NFA is more easier than a DFA
- NFA and DFA are equivalent if they recognize same language
- NFA creates several branches
- If k is the number of states in an NFA, then it has 2^k number of subset of states
- DFA that simulates NFA may have 2^k number of states
- Need to figure out start and accepted state

Equivalence of NFA and DFA (without consider ε arrows)

- Subset construction of the states of NFA
- Starts from an NFA $N = (Q, \Sigma, \delta, \mathrm{q}_{0,} \mathrm{F})$
- Create an DFA $M = (Q', \Sigma, \delta', \{\mathrm{q'}_0\}, \mathrm{F'})$ such that $L(M) = L(N)$
- Input alphabets of the two automata are the same
- Start state of D is the set containing only the start state of N
- Other components of D are constructed as follows:
  - $Q' = P(Q)$
  - For $R \epsilon Q' and\ a \epsilon\ \Sigma$ let $\delta'(\mathrm{R,a}) = \{\mathrm{q}\}\ \{q \epsilon Q | q \epsilon \delta(r, a) for\ some\ r \epsilon R\}$

  R is a state of M and it is also a set of states of N. When M reads a symbol a in state R, it take each states in R and lastly take union of all states. It can be write as follows:

$$\delta'(\mathrm{R,a}) = \mathrm{U}_{r \epsilon R}\, \delta(r, a)$$

- $q'_0 = \{q_0\}$
- $F' = \{R \epsilon Q' | R\ contains\ any\ accepted\ state\ of\ N\}$

Equivalence of NFA and DFA

- Now we need to consider the ε arrows.
- To do so, we set up an extra bit of notation.
- For any state R of M, we define E(R) to be the collection of states that can be reached from members of R by going only along ε arrows, including the members of R themselves
- Formally, for $R \subseteq Q$ let $E(R) = \{q \mid q$ can be reached from R by traveling along 0 or more ε arrows$\}$
- Then we modify the transition function of M to place additional fingers on all states that can be reached by going along ε arrows after every step.
- Replacing $\delta(r,a)$ by $E(\delta(r,a))$ achieves this effect
- Thus $\delta_0(R,a) = \{q \in Q \mid q \in E(\delta(r,a))$ for some $r \in R\}$
- Additionally, we need to modify the start state of M to move the fingers initially to all possible states that can be reached from the start state of N along the ε arrows.

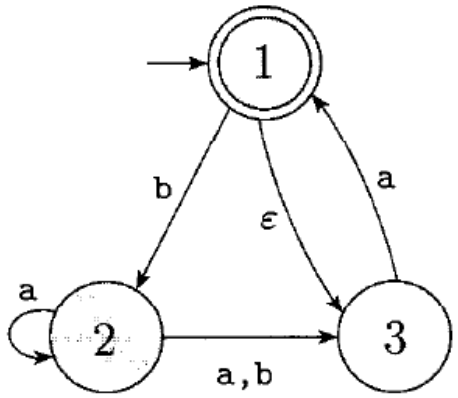
Equivalence of NFA and DFA

- Changing q'_0 to be $E(\{q_0\})$ achieves this effect
- We have now completed the construction of the DFA M that simulates the NFA N.
- The construction of M obviously works correctly.
- At every step in the computation of M on an input, it clearly enters a state that corresponds to the subset of states that N could be in at that point.
- Thus our proof is complete

Equivalence of NFA and DFA

- Transition Table:

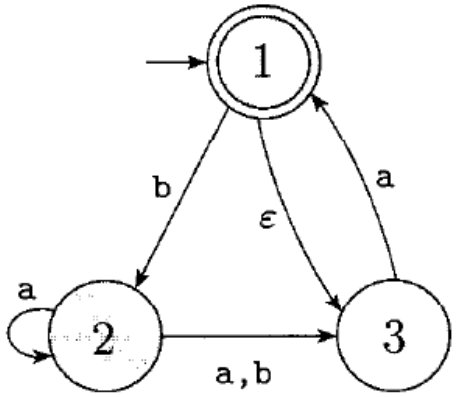
	a	b	ϵ
→1	Φ	2	3
2	{2,3}	3	Φ
3	1	Φ	Φ

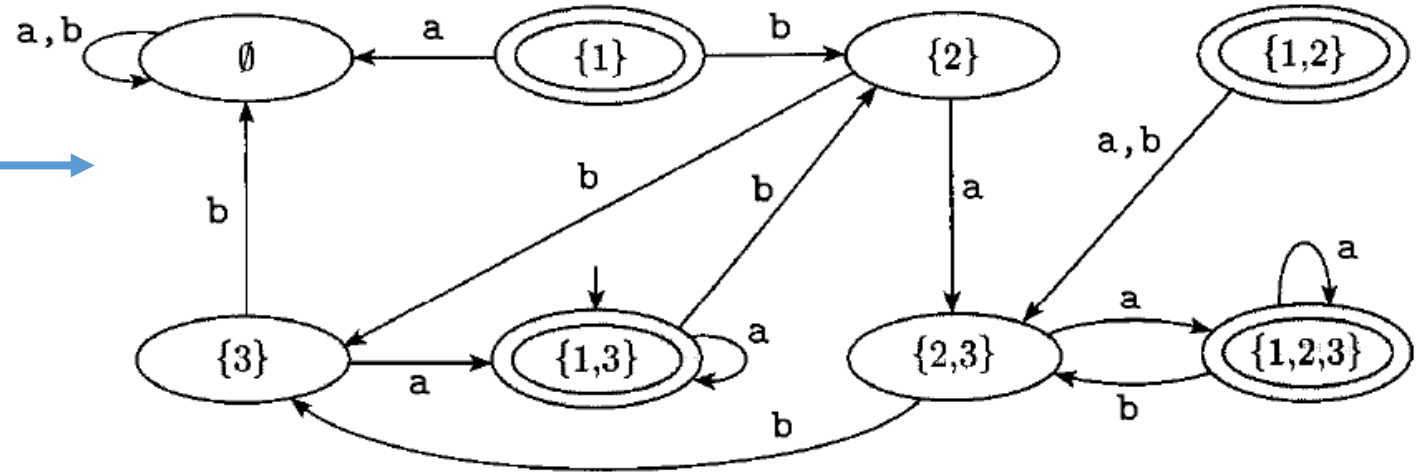


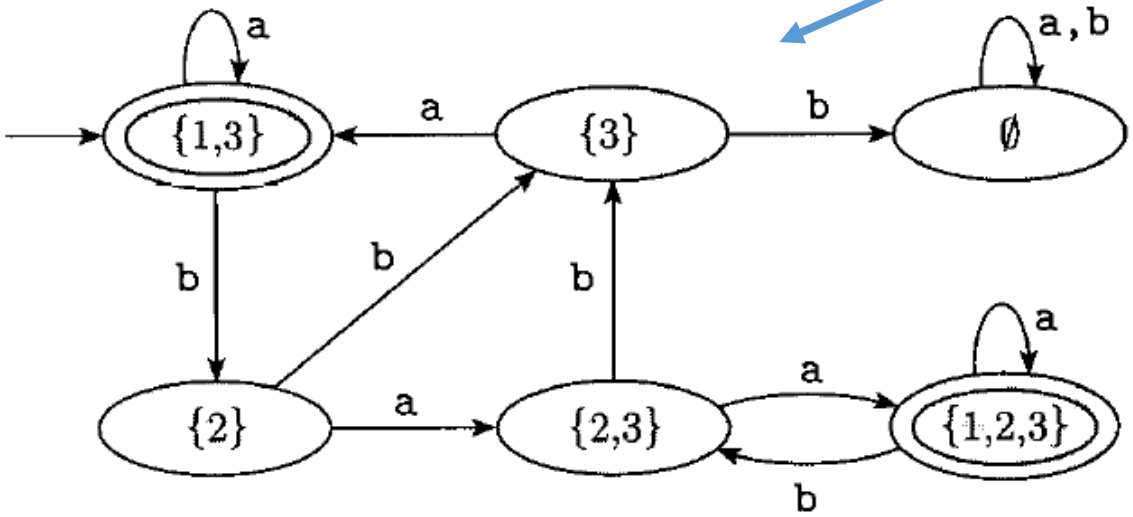
- Subset Construction: Φ, {1},{2},{3},{1,2},{1,3},{2,3},{1,2,3}
- Transition table:

	a	b
Φ	Φ	Φ
{1}	Φ	{2}
{2}	{2,3}	{3}
{3}	{1,3}	Φ
{1,2}	{2,3}	{2,3}
{1,3}	{1,3}	{2}
{2,3}	{1,2,3}	{3}
{1,2,3}	{1,2,3}	{2,3}

Equivalence of NFA and DFA

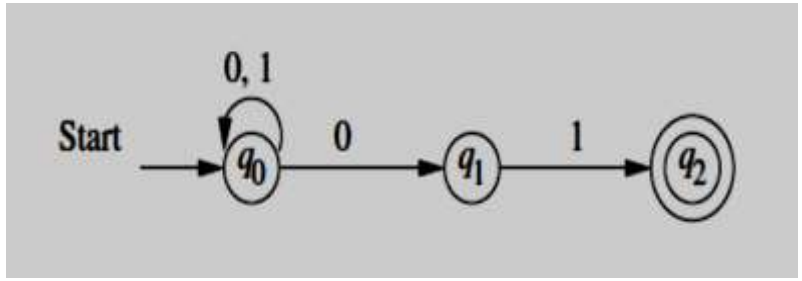




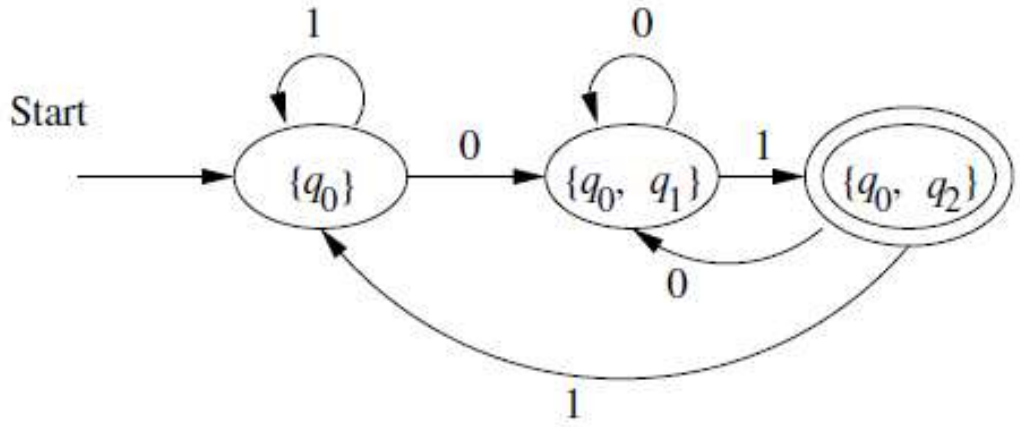


No arrows point at {1} and {1,2}, so they may be discarded without affecting the performance

Equivalence of NFA and DFA



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

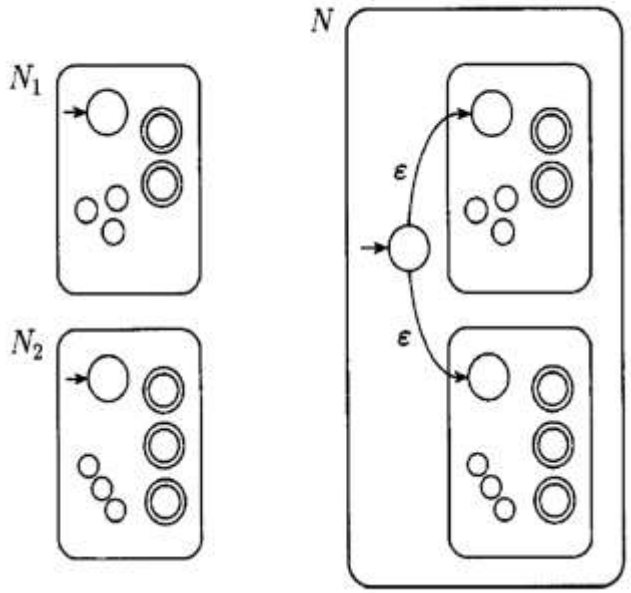


	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0,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

Closure Under Regular Operation

- Aim is to prove that the regular operation (union, concatenation and star) of regular languages are regular
- Using non determinism concept easier to prove
- Theorem: The class of regular language is closed under the union operation
- Sol: Let us consider two regular language A1 and A2 and we want to prove A1UA2 is also regular
- Let us take two NFA N1 and N2 for A1 and A2 respectively and combine them into a new NFA N.
- N must accept its input of either N1 or N2
- N has new start state that branches to the start state of old machines with epsilon arrows
- If one of them accepts the input N will accept it too

Closure Under Regular Operation



PROOF

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

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

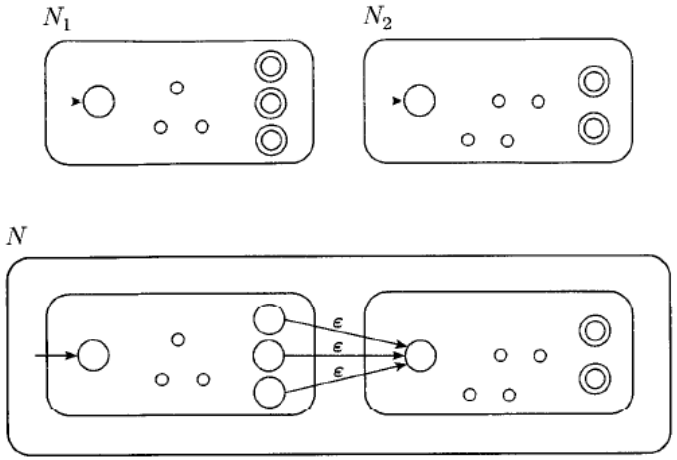
1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.
   The states of N are all the states of N_1 and N_2, with the addition of a new start state q_0.
2. The state q_0 is the start state of N.
3. The accept states $F = F_1 \cup F_2$.
   The accept states of N are all the accept states of N_1 and N_2. That way N accepts if either N_1 accepts or N_2 accepts.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

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

Closure Under Regular Operation

- Theorem: The class of regular language is closed under the concatenation operation
- Sol: Let us consider two regular language A1 and A2 and we want to prove A1oA2 is also regular
- Let us take two NFA N1 and N2 for A1 and A2 respectively and concatenate them into a new NFA N.
- N’s start state to be the start state of N1
- The accepted states of N1 have additional epsilon arrows that nondeterministically branches to N2’s start state
- Accepted state of N are the accepted states of N2
- Input can be split into two parts, first one is accepted byu N1 and second one is accepted by N2

Closure Under Regular Operation



PROOF

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

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

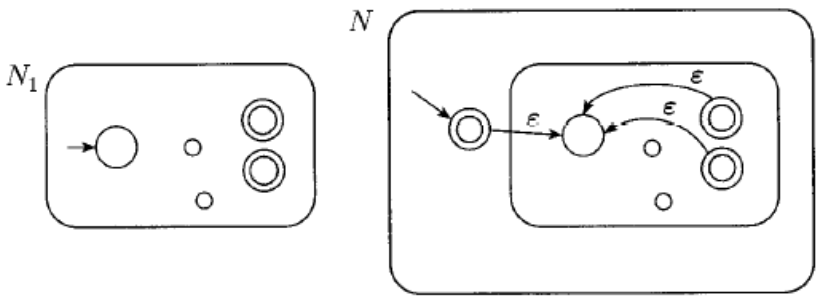
1. $Q = Q_1 \cup Q_2$.
   The states of N are all the states of N_1 and N_2.
2. The state q_1 is the same as the start state of N_1.
3. The accept states F_2 are the same as the accept states of N_2.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

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

Closure Under Regular Operation

- Theorem: The class of regular language is closed under the star operation
- Sol: Let us consider a regular language A1 and we want to prove A1* is also regular
- Let us take an NFA N1 for A1 and the resulting NFA N will accept its input whenever it can be broken into several pieces and N1 accept each pieces.
- N1 with additional epsilon arrows returning to the start state from the accepted states.
- Adding a new start state which is also an accepted state and has an epsilon arrow to the old start state.

Closure Under Regular Operation



PROOF Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1.
Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize A_1^*.

1. $Q = \{q_0\} \cup Q_1$.
   The states of N are the states of N_1 plus a new start state.
2. The state q_0 is the new start state.
3. $F = \{q_0\} \cup F_1$.
   The accept states are the old accept states plus the new start state.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

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

Reference Books

REFERENCE BOOKS
1. Introduction to the Theory of Computation, 3rd edition, 2012- Michael Sipser.
2. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2008 - J. E. Hopcroft, R. Motwani, and J. D. Ullman.
3. Elements of the Theory of Computation. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd edition, 1997- H. R. Lewis and C. H. Papadimitriou.

Welcome

Course Code: CSE 217

Course Name: Theory of Computation

Credit Hour: 3.00

Instructor: Maj Mahbub

Outline

- Regular Expression
- Formal Definition of Regular Expression

Regular Expression

- A Regular Expression (RE) is a sequence of characters that specifies search pattern in a text
- It is an expression that describes the languages
- (0U1)0*-> all strings starting with 0 or 1 and followed by any number of 0s
- Application: searching the string, describing patterns, Lexical analyzer generator
- Example:
  - (0U1)*-all possible strings of 0s and 1s
  - $\Sigma = \{0,1\}$, for (0U1) we can write Σ that describes that the languages consisting of all strings of length 1 over this alphabet
  - Σ*-all strings over the alphabet
  - Σ*1- language that contains all stings ended with 1

Regular Expression

In the following examples we assume that the alphabet Σ is $\{0,1\}$.

1. $0^*10^* = \{w \mid w \text{ has exactly a single } 1\}$.
2. $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}$.
3. $\Sigma^*001\Sigma^* = \{w \mid w \text{ contains the string } 001 \text{ as a substring}\}$.
4. $(\Sigma\Sigma)^* = \{w \mid w \text{ is a string of even length}\}$.[5]
5. $(\Sigma\Sigma\Sigma)^* = \{w \mid \text{the length of } w \text{ is a multiple of three}\}$.
6. $01 \cup 10 = \{01, 10\}$.
7. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w \text{ starts and ends with the same symbol}\}$.
8. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.
   The expression $0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$, so the concatenation operation adds either 0 or ε before every string in 1^*.
9. $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.
10. $1^*\emptyset = \emptyset$.
    Concatenating the empty set to any set yields the empty set.
11. $\emptyset^* = \{\varepsilon\}$.
    The star operation puts together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, giving only the empty string.

Regular Expression

- Precedence of calculation of RE
- Star then concatenation finally union unless parentheses changes

Formal Definition of Regular Expression

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

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

- R^+=RR* (one of more concatenation of strings from R)
- R*(0 or more concatenation of strings from R)

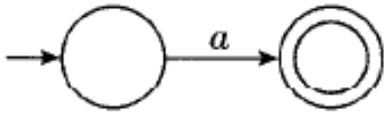
Regular Expression and Finite Automata Equivalence

- RE and Finite Automata are equivalent in their descriptive power
- Any RE can be converted to FA that recognize the language it describes or vice versa

Lemma

- If a language is described by a regular expression then it is regular

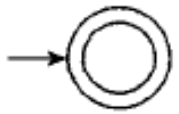
- Let us consider a RE R describing a language A. We need to convert R into a NFA recognizing R. If NFA recognize A then, A is regular
- Lets convert R into an NFA N. We consider six cases in the formal definition of RE
- 1. R=a for some a ϵ Σ then L(R)={a} and the following NFA recognize L(R)



Formally, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$, $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.

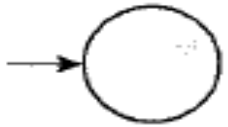
Lemma

- 2. R=ε then L(R)={ε} and the following NFA recognize L(R)



Formally, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$, where $\delta(r, b) = \emptyset$ for any r and b.

- 3. R=Φ then L(R)={Φ} the following NFA recognize L(R)



Formally, $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, where $\delta(r, b) = \emptyset$ for any r and b.

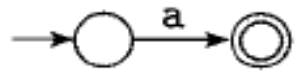
4. $R = R_1 \cup R_2$.

5. $R = R_1 \circ R_2$.

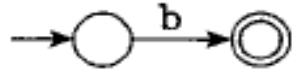
6. $R = R_1^*$.

RE Conversion to NFA

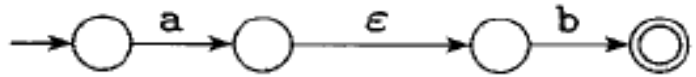
a



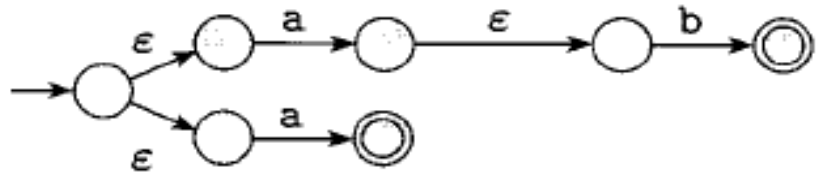
b



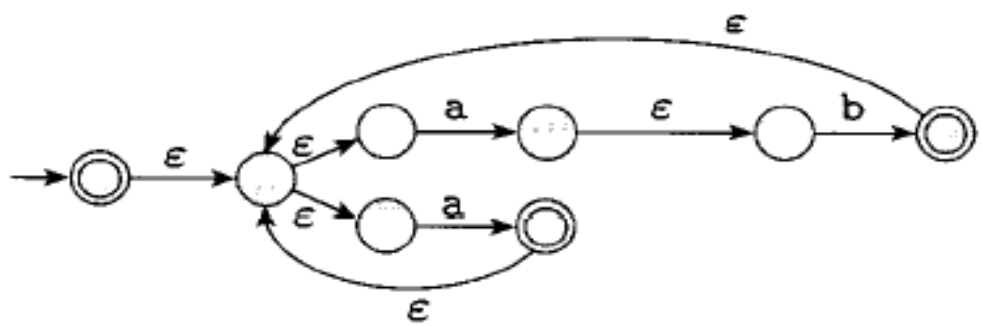
ab



ab ∪ a

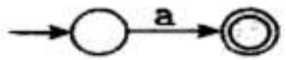


(ab ∪ a)*

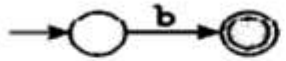


RE Conversion to NFA

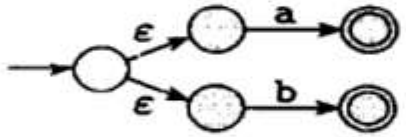
a



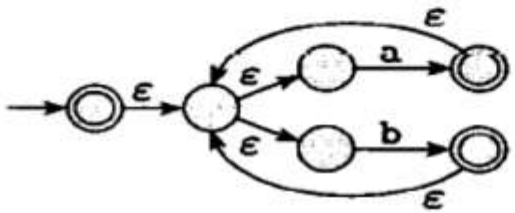
b



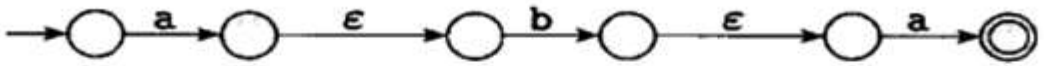
a∪b



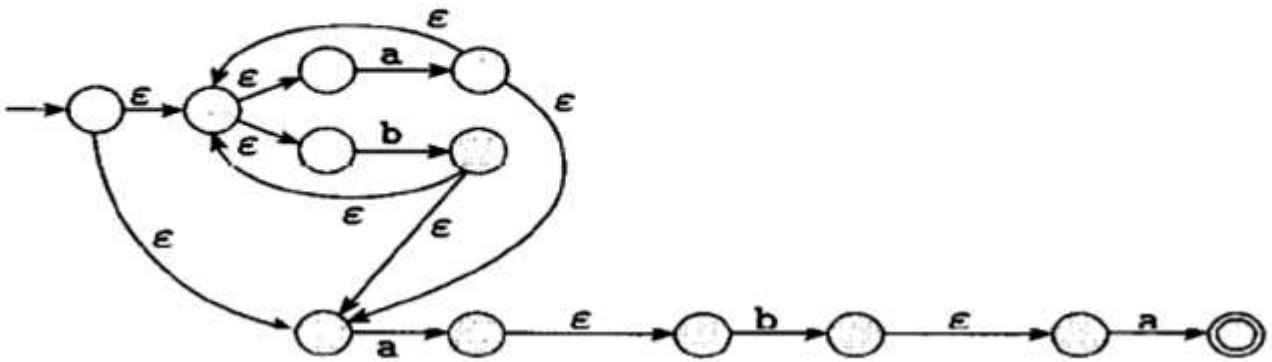
(a∪b)*



aba



(a∪b)*aba



Reference Books

REFERENCE BOOKS
1. Introduction to the Theory of Computation, 3rd edition, 2012- Michael Sipser.
2. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2008 - J. E. Hopcroft, R. Motwani, and J. D. Ullman.
3. Elements of the Theory of Computation. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd edition, 1997- H. R. Lewis and C. H. Papadimitriou.

Welcome

Course Code: CSE 217

Course Name: Theory of Computation

Credit Hour: 3.00

Instructor: Maj Mahbub

Outline

- Context Free Language
- Context Free Grammar (CFG)
- Formal Definition of CFG

Context-Free Grammars and Languages

$$S \rightarrow 0S1 \mid S0S$$

Context Free Language

- Language can be described by Finite automata and Regular Expression
- Context Free Grammar is another way of describing language (More powerful than FA and RE)
- Recursive structure
- What is grammar?
- Particular combination of words in a sentence is well formed or not. “ The cow can sing a song”-No grammatical errors
- The form of sentence is called syntax and the meaning is called semantics
- A language is called formal if it can be specified by a well defined set of syntax

Components of Grammar

- Collection of substitution rules called production
- Each rule appears in a line (Consisting a symbol and a string separated by arrow)
- Symbol is called variable
- Strings- variables and other symbols called terminals
- Variable often represented with capital letter and terminals often represented by lowercase letter and member of a alphabet
- One variable is designated as start variable

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

Generating Strings from a Grammar

- Grammar G1 generates the string 000#111.
- The sequence of substitutions to obtain a string is called a derivation.
- A derivation of string 000#111 in grammar G1 is
- $A \Rightarrow 0A1$
- $\Rightarrow 00A11$
- $\Rightarrow 000A111$
- $\Rightarrow 000B111$
- $\Rightarrow 000\#111$

$A \rightarrow 0A1$
$A \rightarrow B$
$B \rightarrow \#$

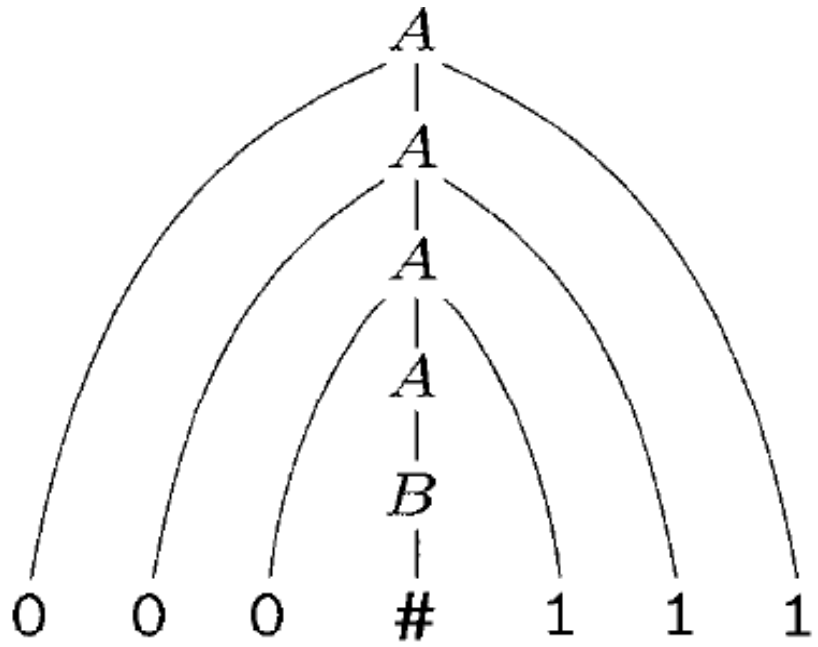
1. Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise.
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.
3. Repeat step 2 until no variables remain.

Generating Strings from a Grammar

- You may also represent the same information pictorially with a parse tree

  A ⇒ 0A1
  ⇒ 00A11
  ⇒ 000A111
  ⇒ 000B111
  ⇒ 000#111

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$



Parse tree-an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.

Generating Strings from a Grammar

- Language of the grammar L(G)={$0^n\#1^n \mid n \geq 0$}
- Any language that is generated from context free grammar is called context free language

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

Context Free Language

- Let consider of a grammar that describes English language

⟨SENTENCE⟩ → ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩
⟨NOUN-PHRASE⟩ → ⟨CMPLX-NOUN⟩ | ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩
⟨VERB-PHRASE⟩ → ⟨CMPLX-VERB⟩ | ⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩
⟨PREP-PHRASE⟩ → ⟨PREP⟩⟨CMPLX-NOUN⟩
⟨CMPLX-NOUN⟩ → ⟨ARTICLE⟩⟨NOUN⟩
⟨CMPLX-VERB⟩ → ⟨VERB⟩ | ⟨VERB⟩⟨NOUN-PHRASE⟩
⟨ARTICLE⟩ → `a` | `the`
⟨NOUN⟩ → `boy` | `girl` | `flower`
⟨VERB⟩ → `touches` | `likes` | `sees`
⟨PREP⟩ → `with`

- How many variable? How many terminals possible? How many rules?

Context Free Language

Derivation of string: **a boy sees**

⟨SENTENCE⟩ ⇒ ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩
⇒ ⟨CMPLX-NOUN⟩⟨VERB-PHRASE⟩
⇒ ⟨ARTICLE⟩⟨NOUN⟩⟨VERB-PHRASE⟩
⇒ a ⟨NOUN⟩⟨VERB-PHRASE⟩
⇒ a boy ⟨VERB-PHRASE⟩
⇒ a boy ⟨CMPLX-VERB⟩
⇒ a boy ⟨VERB⟩
⇒ a boy sees

Context Free Language

Another example of context free grammar of a palindrome

$$
\begin{array}{lrcl}
1. & P & \rightarrow & \epsilon \\
2. & P & \rightarrow & 0 \\
3. & P & \rightarrow & 1 \\
4. & P & \rightarrow & 0P0 \\
5. & P & \rightarrow & 1P1
\end{array}
$$

Formal Definition of CFG

A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

Formal Definition of CFG

If u, v, and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv yields uwv, written $uAv \Rightarrow uwv$.

Say that u derives v, written $u \overset{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$.

The language of the grammar is $\{w \in \Sigma* \mid S \Rightarrow w\}$.

CFG

- G4 = (V, Σ, R, <EXPR>).
- V is {<EXPR>, <TERM>, <FACTOR>} and Σ is {a, +, ×,(,)}.
- The rules are:

  <EXPR> → <EXPR> + <TERM> | <TERM>

  <TERM> → <TERM> × <FACTOR> | <FACTOR>

  <FACTOR> → (<EXPR>) | a

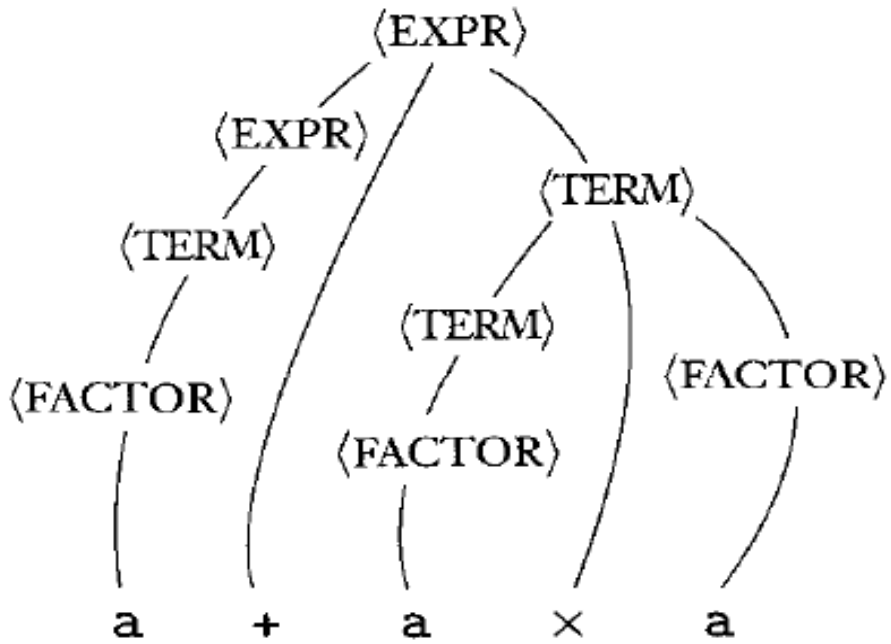
CFG

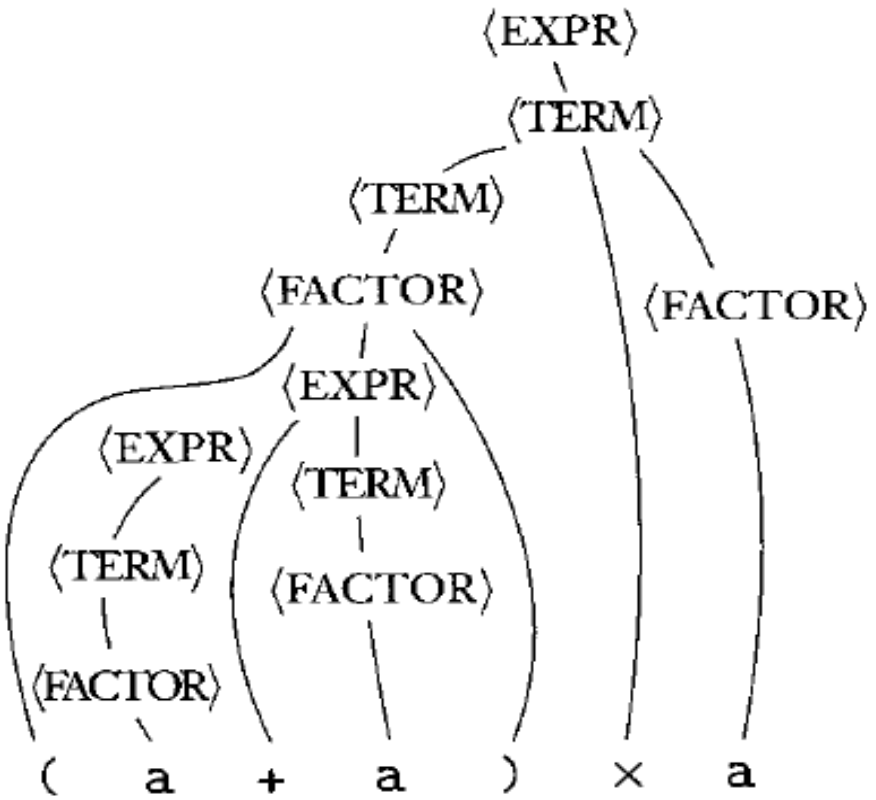
- Derive a+axa and (a+a)xa
  <expr>→<expr>+<term>
  <expr>→<expr>+<term>x<factor>
  <expr>→<term>+<term>xa
  <expr>→<factor>+<factor>xa
  <expr>→a+axa

- (a+a)xa ??

CFG

- Parse tree of a+axa and (a+a)xa





Designing CFG

- To get a grammar for the language $L = \{0^n1^n \mid n \geq 0\} \cup \{1^n0^n \mid n \geq 0\}$
- First construct the grammar $S1 \rightarrow 0S_11 \mid \epsilon$ for the language$\{0^n1^n \mid n \geq 0\}$.
- The grammar $S_2 \rightarrow 1S_20 \mid \epsilon$ for the language$\{1^n0^n \mid n \geq 0\}$
- Then add the rule $S \rightarrow S_1 \mid S_2$ to give the grammar
- $S \rightarrow S_1 \mid S_2$
- $S_1 \rightarrow 0S_11 \mid \epsilon$
- $S_2 \rightarrow 1S_20 \mid \epsilon$

Designing CFG

- Given language L = $\{0^n1^{2n} \mid n \geq 0\}$
- The grammar forces every 0 to match to 11
- The context-free grammar for L is
- $S \rightarrow 0S11 \mid \epsilon$

Designing CFG

- $L = \{0^n1^m \mid m,n \geq 0; 2n \leq m \leq 3n\}$
- The grammar forces every 0 to match to 11 or 111
- The context-free grammar for L is
- $S \rightarrow 0S11 \mid 0S111 \mid \epsilon$

Designing CFG

- $L = \{0^n1^m \mid m, n \geq 0; n \neq m\}$
- Two scenario:
- Let $L_1 = \{0^n1^m \mid m, n \geq 0, n > m\}$
- Let $L_2 = \{0^n1^m \mid m, n \geq 0, n < m\}$
- Then, if S_1 generates L_1, and S_2 generates L_2, our grammar will be,

$$S \rightarrow S_1 \mid S_2$$

Designing CFG

- L_1 is just the language of strings 0^n1^n with one or more extra 0's in front.

$$S_1 \rightarrow 0S_1 \mid 0S_1 1 \mid \epsilon$$

- L_2 is just the language of strings 0^n1^n with one or more extra 1's in the end.

$$S_2 \rightarrow S_2 1 \mid 0\, S_2\, 1 \mid \epsilon$$

- The context-free grammar for L is

$$S \rightarrow S_1 \mid S_2$$
$$S_1 \rightarrow 0S_1 \mid 0S_1 1 \mid \epsilon$$
$$S_2 \rightarrow S_2 1 \mid 0\, S_2\, 1 \mid \epsilon$$

Designing CFG

- L = {w| w$\epsilon\{a,b\}^*$,$n_a(w) = n_b(w)$}
- The grammar generates the basis strings of ϵ, ab and ba.
- If w is a string in this grammar, awb will belong to this grammar.
- awb will be generated from by using the rule S → aSb.

Designing CFG

- L = {w| w$\epsilon\{a,b\}^*$,$n_a(w) = n_b(w)$}
- If w is a string in this grammar, bwa will belong to this grammar.
- bwa will be generated from by using the rule S → bSa
- If w is a string in this grammar, ww will belong to this grammar
- w will be generated from by using the rule S → SS
- S → aSb | bSa | SS |ϵ

Designing CFG

- L = {w| wϵ{0,1} * and of even length}
- The grammar generates the basis strings of ϵ, 00, 01, 10 and 11.
- If w is a string in this grammar, 0w0 will belong to this grammar.
- 0w0 will be generated from by using the rule S → 0S0.
- Similarly, 0w1, 1w0, 1w1 belong to this grammar and the rules be like S → 0S1, S → 1S0, S → 1S1
- The CFG is :S → 0S0 | 0S1 | 1S0 |1S1|ϵ

Designing CFG

- $L = \{a^n b^m c^k \mid n,m,k \geq 0 \text{ and } n=m+k\}$
- Every b should match an a.
- Every c should match an a.

$S \rightarrow aSc \mid B$

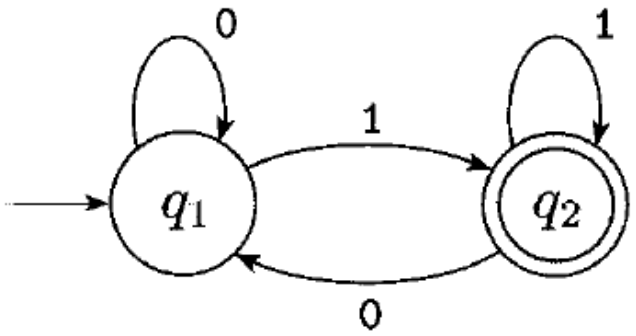
$B \rightarrow aBb \mid \epsilon$

Designing CFG

- Constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language
- You can convert any DFA into an equivalent CFG as follows.
- Make a variable R_i for each state q_i of the DFA.
- Add the rule $R_i \rightarrow aR_j$ to the CFG if $\delta(q_i , a) = q_j$ is a transition in the DFA.
- Add the rule $R_i \rightarrow \epsilon$ if q_i is an accept state of the DFA.
- Make R_0 the start variable of the grammar, where q_0 is the start state of the machine.

Designing CFG

- Constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language
- $R_1 \rightarrow 0R_1 | 1R_2$
- $R_2 \rightarrow 0R_1 | 1R_2 | \epsilon$



Designing CFG (Practice)

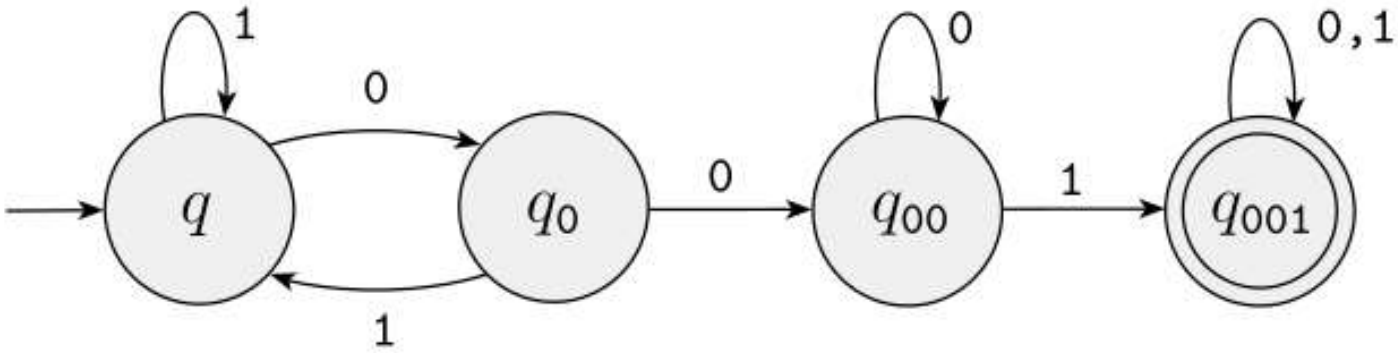


FIGURE 1.22
Accepts strings containing 001

Leftmost and Rightmost Derivation

- We want to restrict the number of choices we have in deriving a string.
- It is often useful to require that at each step we replace the leftmost variable by one of its production bodies.
- Such a derivation is called a leftmost derivation.
- We indicate that a derivation is leftmost by using the relations $\underset{lm}{\Rightarrow}$ and $\overset{*}{\underset{lm}{\Rightarrow}}$ for one or many steps, respectively.
- If the grammar G that is being used is not obvious, we can place the name G below the arrow in either of these symbols.

Leftmost and Rightmost Derivation

- Similarly, it is possible to require that at each step the rightmost variable is replaced by one of its bodies.
- If so, we call the derivation rightmost and use the symbols $\underset{rm}{\Rightarrow}$ and $\overset{*}{\underset{rm}{\Rightarrow}}$ to indicate one or many rightmost derivation steps, respectively.

- Again, the name of the grammar may appear below these symbols if it is not clear which grammar is being used.

Leftmost and Rightmost Derivation

A leftmost derivation.

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

$$
\begin{array}{lllll}
1. & E & \rightarrow & I \\
2. & E & \rightarrow & E + E \\
3. & E & \rightarrow & E * E \\
4. & E & \rightarrow & (E) \\
 & & & \\
5. & I & \rightarrow & a \\
6. & I & \rightarrow & b \\
7. & I & \rightarrow & Ia \\
8. & I & \rightarrow & Ib \\
9. & I & \rightarrow & I0 \\
10. & I & \rightarrow & I1
\end{array}
$$

Leftmost and Rightmost Derivation

- Thus, we can describe the same derivation by:

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

$$
\begin{array}{lllll}
1. & E & \rightarrow & I \\
2. & E & \rightarrow & E+E \\
3. & E & \rightarrow & E*E \\
4. & E & \rightarrow & (E) \\
 & & & \\
5. & I & \rightarrow & a \\
6. & I & \rightarrow & b \\
7. & I & \rightarrow & Ia \\
8. & I & \rightarrow & Ib \\
9. & I & \rightarrow & I0 \\
10. & I & \rightarrow & I1
\end{array}
$$

Leftmost and Rightmost Derivation

- Thus, we can describe the same derivation by:

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

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

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

Leftmost and Rightmost Derivation

- Rightmost derivation

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

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

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

- Thus the rightmost derivation can be expressed as: $E \overset{*}{\underset{lm}{\Rightarrow}} \quad a * (a + b00)$

Leftmost and Rightmost Derivation

- Any derivation has an equivalent leftmost and an equivalent rightmost derivation.
- That is, if w is a terminal string, and A is a variable, then
  - $A \overset{*}{\Rightarrow} w$ if and only if $A \underset{lm}{\overset{*}{\Rightarrow}} w$ and
  - $A \overset{*}{\Rightarrow} w$ if and only if $A \underset{rm}{\overset{*}{\Rightarrow}} w$

Language of a Grammar

- If G(V, T, P, S) is a CFG, the language of G, denoted L(G), is the set of terminal strings that have derivations from the start symbol

$$L(G) = \{w \; in \; T^{*} | S \overset{*}{\underset{G}{\Rightarrow}} \mathrm{w}\}$$

- If a language L is the language of some context-free grammar, then L is said to be a context-free language, or CFL.

Sentential Forms

- Derivations from the start symbol produce strings that have a special role.
- We call these “sentential forms.”

- That is, if G(V, T, P, S) is a CFG, then any string α in $(VUT)^*$ such that $S \overset{*}{\Rightarrow} \alpha$ is a sentential form.
- $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$ is a left-sentential form.
- $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha$ is a right-sentential form.

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

Sentential Forms

- For example, $E * (I + E)$ is a sentential form, since there is a derivation, $E \Longrightarrow E * E \Longrightarrow E * (E) \Longrightarrow E * (E + E) \Longrightarrow E * (I + E)$
- However this derivation is neither leftmost nor rightmost, since at the last step, the middle E is replaced.
- As an example of a left-sentential form, consider $a * E$,with the leftmost derivation, $E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E$

- Example of right sentential form:$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E)$

Parse Trees

- There is a tree representation for derivations that has proved extremely useful.
- This tree shows us clearly how the symbols of a terminal string are grouped into substrings.
- Each of the terminals belongs to the language of one of the variables of the grammar.

Parse Trees

- The tree, known as a “parse tree” when used in a compiler, is the data structure of choice to represent the source program.

- In a compiler, the tree structure of the source program facilitates the translation of the source program into executable code by allowing natural, recursive functions to perform this translation process.

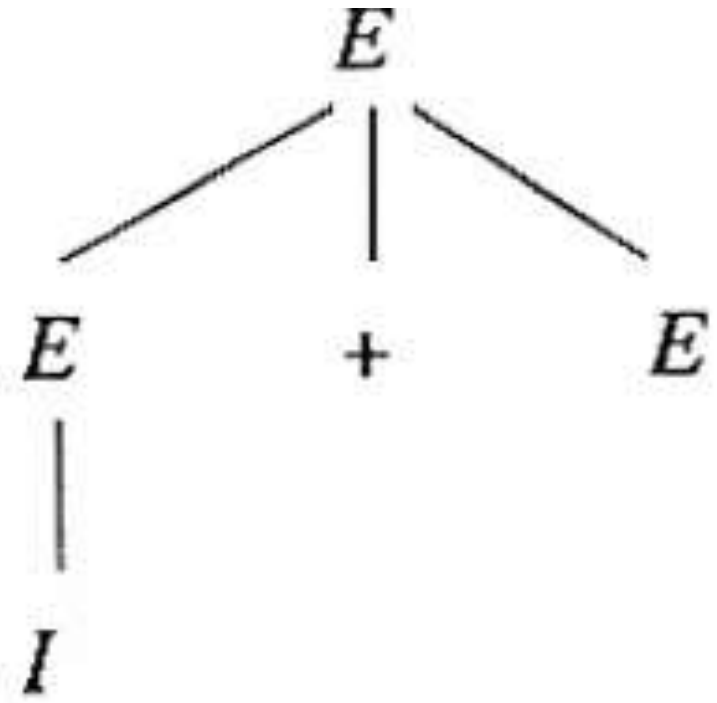
Parse Trees

- Certain grammars allow a terminal string to have more than one parse tree.
- That situation makes the grammar unsuitable for a programming language.
- The compiler could not tell the structure of certain source programs.
- And therefore could not with certainty deduce what the proper executable code for the program was.

Constructing Parse Trees

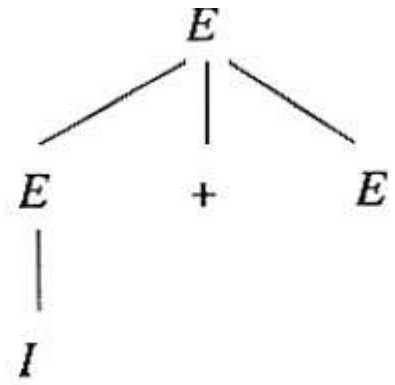
- Let us fix on a grammar G(V, T, P, S).
- The parse trees for G are trees with the following conditions:
  1. Each interior node is labeled by a variable in V.
  2. Each leaf is labeled by either a variable, a terminal, or ϵ
  3. However, if the leaf is labeled ϵ, then it must be the only child of its parent
  4. If an interior node is labeled A, and its children are labeled $X_1, X_2, \ldots X_k$ respectively, from the left, then $A \rightarrow X_1X_2 \ldots\ldots X_k$ is a production in P.
  5. Note that the only time one of the X's can be ϵ is if that is the label of the only child, and $A \rightarrow \epsilon$ is a production of G

Constructing Parse Trees



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

Constructing Parse Trees



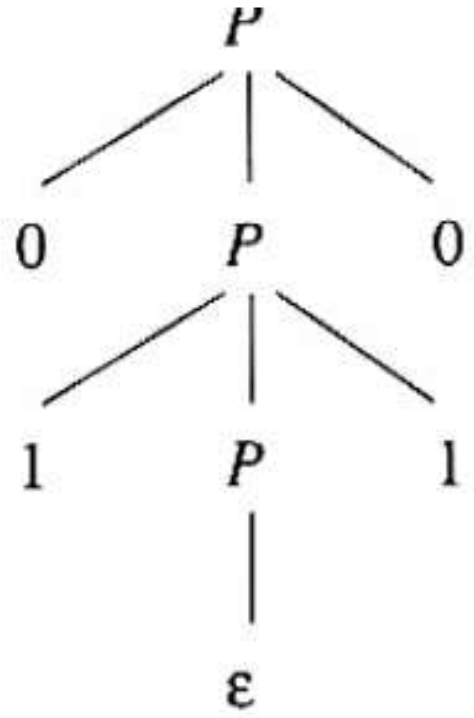
A parse tree showing the derivation of $I + E$ from E

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

A context-free grammar for simple expressions

- The production used at the root is E →E + E.
- At the leftmost child of the root, the production E → I is used.

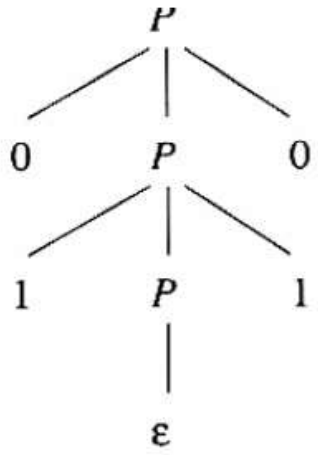
Constructing Parse Trees



A parse tree showing the derivation $P \overset{*}{\Rightarrow} 0110$

- Figure shows a parse tree for the palindromic grammar.

Constructing Parse Trees



A parse tree showing the derivation $P \stackrel{*}{\Rightarrow} 0110$

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

A context-free grammar for palindromes

- The production used at the root is P →0P0.
- At the middle child of the root it is P →1P1.
- Note that at the bottom is a use of the production P → ϵ
- That use, labeled ϵ is the only time that a node labeled ϵ can appear in a parse tree.

The Yield of a Parse Tree

- If we look at the leaves of any parse tree and concatenate them from the left, we get a string.
- This is called the yield of the tree.
- This is always a string that is derived from the root variable.

The Yield of a Parse Tree

- Of special importance are those parse trees such that:

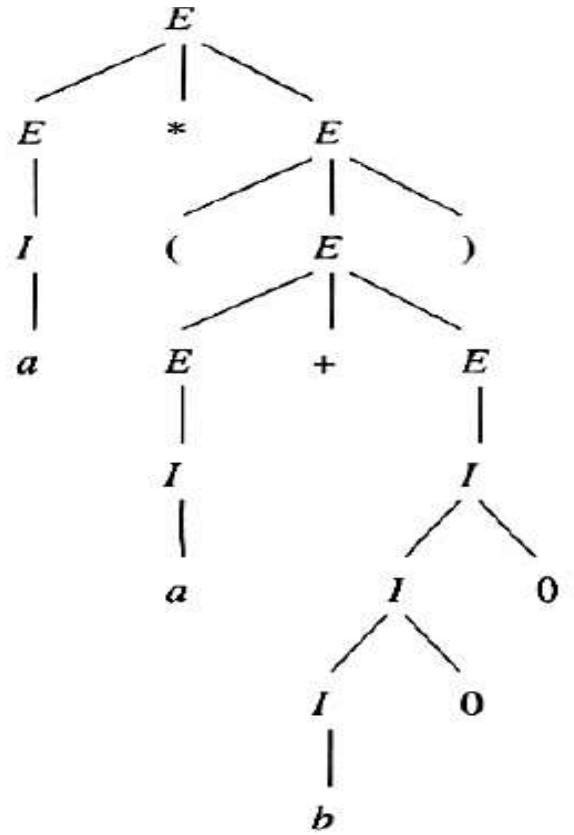
  1. The yield is a terminal string.
     That is, all leaves are labeled either with a terminal or with ϵ

  2. The root is labeled by the start symbol.

- These are the parse trees whose yields are strings in the language of the underlying grammar.

The Yield of a Parse Tree

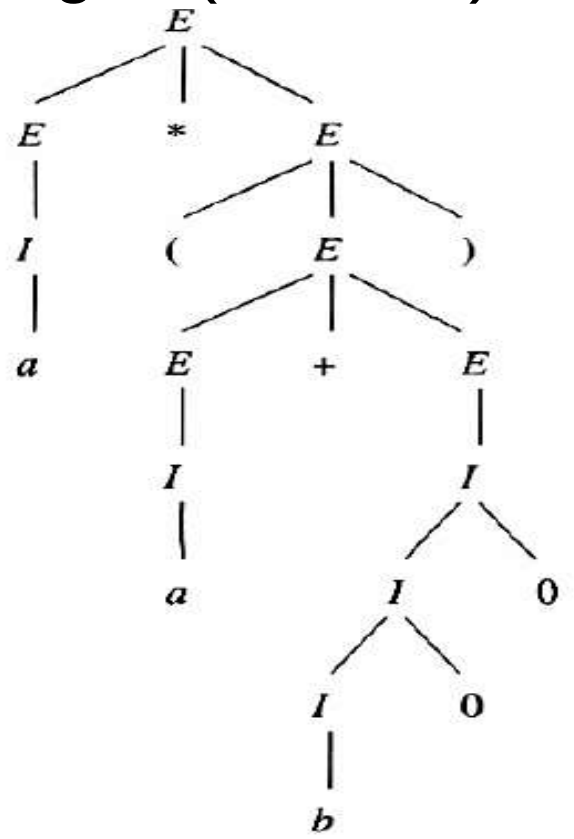
- The figure is an example of a tree with a terminal string as yield and the start symbol at the root.



Parse tree showing $a * (a + b00)$ is in the language of our expression grammar

The Yield of a Parse Tree

- It is based on the grammar for expressions.
- This tree's yield is the string a∗(a + b00).



Parse tree showing $a * (a + b00)$ is in the language of our expression grammar

Ambiguous Grammars

- Consider the sentential form E+ E∗E
- It has two derivations from E
  - $E \Longrightarrow E + E \Longrightarrow E + E * E$ .....(1)
  - $E \Longrightarrow E * E \Longrightarrow E + E * E$ ....(2)

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

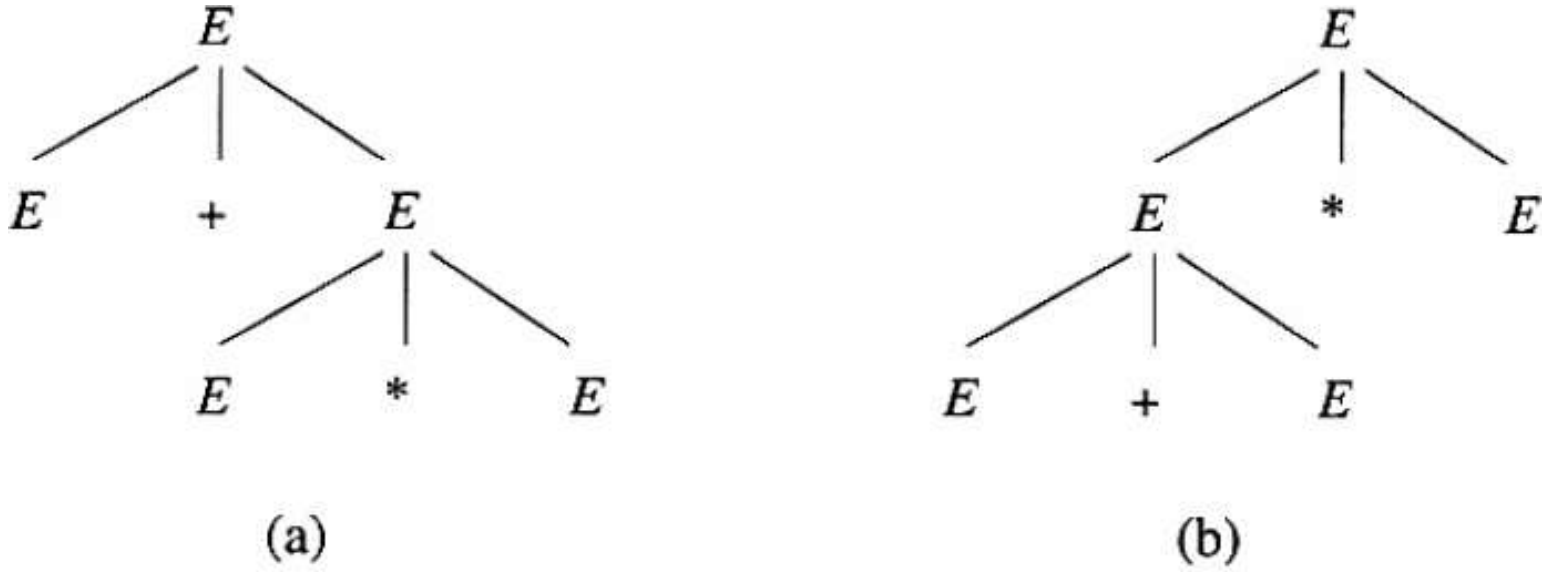
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

A context-free grammar for simple expressions

Notice that in derivation(1) the second E is replaced by $E * E$ while in derivation (2) the first E is replaced by $E + E$

Ambiguous Grammars

- $E \Longrightarrow E + E \Longrightarrow E + E * E$ .....(1)
- $E \Longrightarrow E * E \Longrightarrow E + E * E$ ....(2)



Two parse trees with the same yield

Ambiguous Grammars

- $E \Longrightarrow E + E \Longrightarrow E + E * E$ .....(1)
- $E \Longrightarrow E * E \Longrightarrow E + E * E$ ....(2)

$3 + 4 * 5 = 23$?.....(1)

$3 + 4 * 5 = 35$? .....(2)

- The difference between these two derivations is significant.
- Derivation (1) says that the second and third expressions are multiplied, and the result is added to the first expression.
- Derivation (2) adds the first two expressions and multiplies the result by the third.

Ambiguous Grammars

- $E \Longrightarrow E + E \Longrightarrow E + E * E$ .....(1)
- $E \Longrightarrow E * E \Longrightarrow E + E * E$ ....(2)

$3 + 4 * 5 = 23$?.....(1)

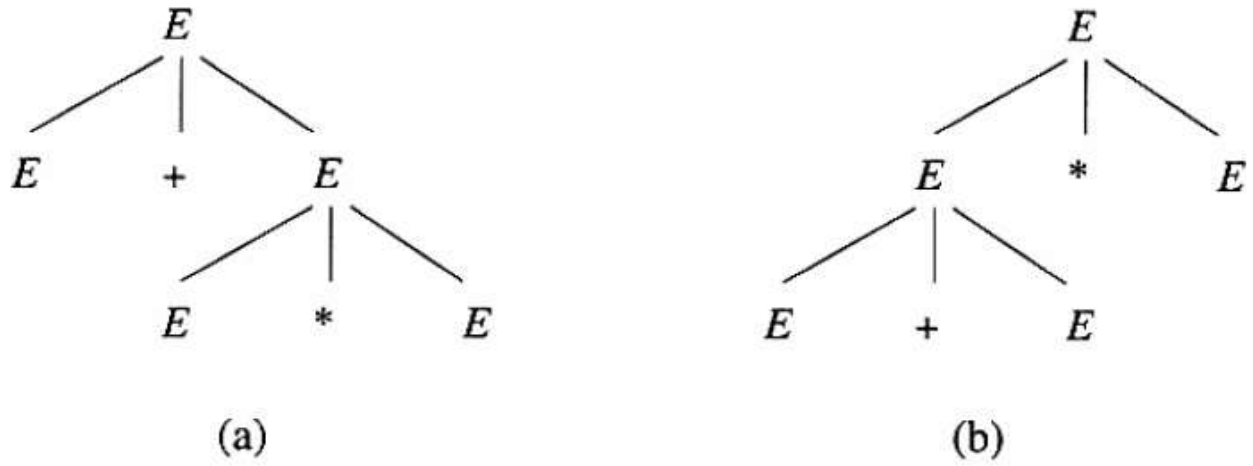
$3 + 4 * 5 = 35$? .....(2)

- In more concrete terms, the first derivation suggests that $1 + 2 * 3$ should be grouped $1 + (2 * 3) = 7$.
- The second derivation suggests the same expression should be grouped $(1 + 2) * 3 = 9$.
- Obviously, the first of these, and not the second, matches our notion of correct grouping of arithmetic expressions

Ambiguous Grammars

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

A context-free grammar for simple expressions



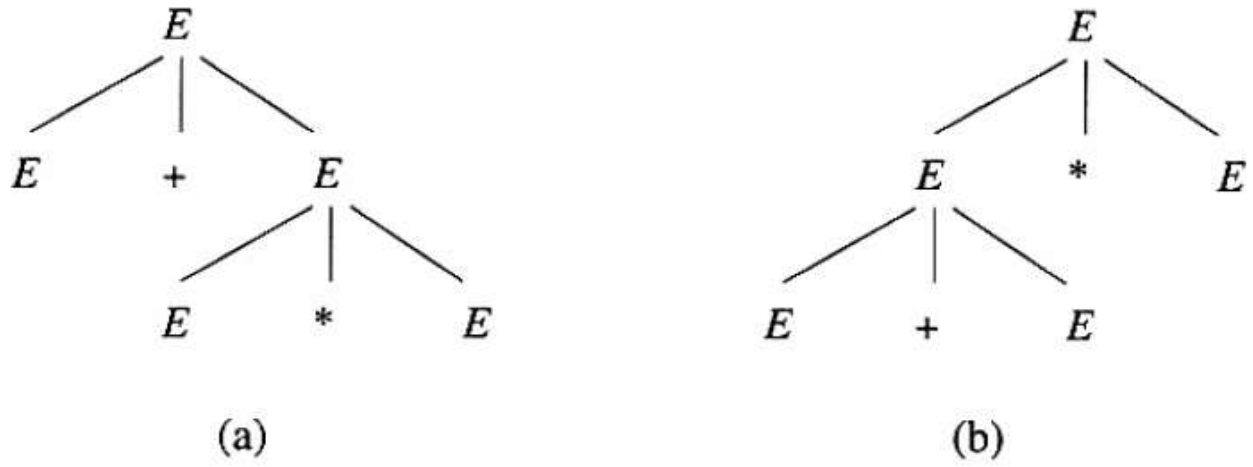
Two parse trees with the same yield

- The grammar of figure gives two different structures to any string of terminals that is derived by replacing the three expressions in E + E ∗ E by identifiers.
- We see that this grammar is not a good one for providing unique structure.

Ambiguous Grammars

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

A context-free grammar for simple expressions



Two parse trees with the same yield

- In particular, while it can give strings the correct grouping as arithmetic expressions, it also gives them incorrect groupings.
- To use this expression grammar in a compiler, we would have to modify it to provide only the correct groupings.

Ambiguous Grammars

- On the other hand, the mere existence of different derivations for a string (as opposed to different parse trees) does not imply a defect in the grammar.
- The following is an example.
- Using the same expression grammar, we find that the string a + b has many different derivations.

  1. $E \rightarrow E + E \rightarrow I + E \rightarrow a + E \rightarrow a + I \rightarrow a + b$
  2. $E \rightarrow E + E \rightarrow E + I \rightarrow I + I \rightarrow I + b \rightarrow a + b$

- However, there is no real difference between the structures provided by these derivations.
- They each say that a and b are identifiers, and that their values are to be added.

Ambiguous Grammars

- The two examples above suggest that it is not a multiplicity of derivations that cause ambiguity, but rather the existence of two or more parse trees.

- Thus, we say a CFG G(V, T, P, S) is ambiguous if there is at least one string w in T* for which we can find two different parse trees, each with root labeled S and yield w.

- If each string has at most one parse tree in the grammar, then the grammar is unambiguous.

Ambiguity

- Grammar, G5.

  <EXPR> →<EXPR> + <EXPR> |<EXPR> ∗ <EXPR> |(<EXPR>) |a

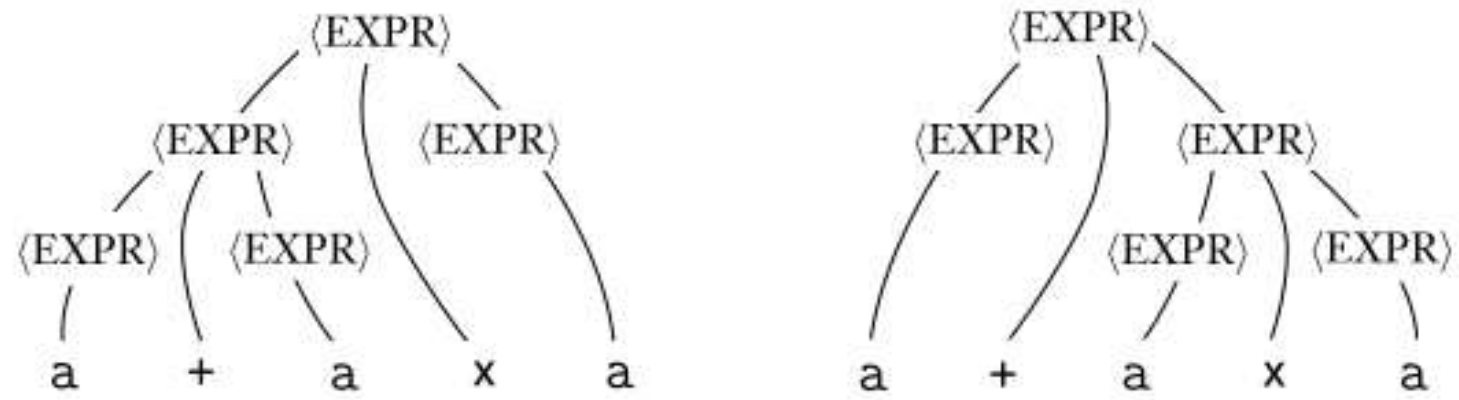


FIGURE 2.6

The two parse trees for the string a+a×a in grammar G_5

Ambiguity

DEFINITION 2.7

A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

Chomsky Normal Form

- Its always convenient to have CFG in simplified form
- Simplest and useful form of CFG
- A CFG is in Chomsky Normal Form (CNF) if every rule is of the form
  - A->BC
  - A->a

  where a is terminal, A,B,C are variables. B and C may not be start variable. We can permit S->ε where S is start variable.

Chomsky Normal Form

- The conversion to Chomsky Normal Form has four main steps:
- 1. Get rid of all ε productions.
- 2. Get rid of all productions where RHS is one variable.
- 3. Replace every production that is too long by shorter productions.
- 4. Move all terminals to productions where RHS is one terminal.

1) Eliminate ε Productions

- Determine the nullable variables (those that generate ε) (algorithm given earlier). Go through all productions, and for each, omit every possible subset of nullable variables. For example, if P → AxB with both A and B nullable, add productions P → xB | Ax | x. After this, delete all productions with empty RHS.

2) Eliminate Variable Unit Productions

- A unit production is where RHS has only one symbol. Consider production $A \rightarrow B$. Then for every production $B \rightarrow \alpha$, add the production $A \rightarrow \alpha$. Repeat until done (but don't re-create a unit production already deleted).

3) Replace Long Productions by Shorter Ones

- For example, if have production A → BCD, then replace it with A → BE and E → CD. (In theory this introduces many new variables, but one can re-use variables if careful.)

4) Move Terminals to Unit Productions

- For every terminal on the right of a non-unit production, add a substitute variable. For example, replace production A → bC with productions A → BC and B → b.

Example

- Consider the CFG:
  $S \rightarrow aXbX$
  $X \rightarrow aY \mid bY \mid \varepsilon$
  $Y \rightarrow X \mid c$
- The variable X is nullable and so therefore is Y . After elimination of ε, we obtain:
  $S \rightarrow aXbX \mid abX \mid aXb \mid ab$
  $X \rightarrow aY \mid bY \mid a \mid b$
  $Y \rightarrow X \mid c$

Example

- After elimination of the unit production Y → X, we obtain:

  S → aXbX | abX | aXb | ab

  X → aY | bY | a | b

  Y → aY | bY | a | b | c

Example

- Now, break up the RHSs of S; and replace a by A, b by B and c by C wherever not units:
- S → EF | AF | EB | AB
- X → AY | BY | a | b
- Y → AY | BY | a | b | c
- E → AX
- F → BX
- A → a
- B → b
- C → c

- S → aXbX | abX | aXb | ab
- X → aY | bY | a | b
- Y → aY | bY | a | b | c

Practice

- Convert the following CFG into Chomsky Normal Form:

  $S \rightarrow AbA$

  $A \rightarrow Aa \mid \varepsilon$

Practice

- Convert the following CFG into Chomsky Normal Form:

  $S \rightarrow aA|bB|b$

  $A \rightarrow Baa|ba$

  $B \rightarrow bAAb|ab$

Practice

- Convert the following CFG into Chomsky Normal Form:

  $S \rightarrow ASB$

  $A \rightarrow aAS|a|\varepsilon$

  $B \rightarrow SbS|A|bb$

Pushdown Automata

- New type of computational model called pushdown automata.
- These automata are like nondeterministic finite automata but have an extra component called a stack.
- The stack provides additional memory beyond the finite amount available in the control.
- The stack allows pushdown automata to recognize some non regular languages.
- Pushdown automata are equivalent in power to context-free grammars.

Pushdown Automata

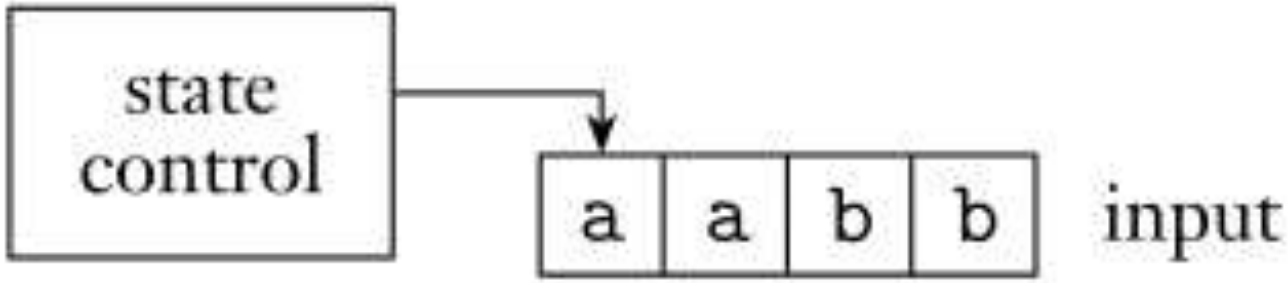


FIGURE 2.11
Schematic of a finite automaton

Pushdown Automata

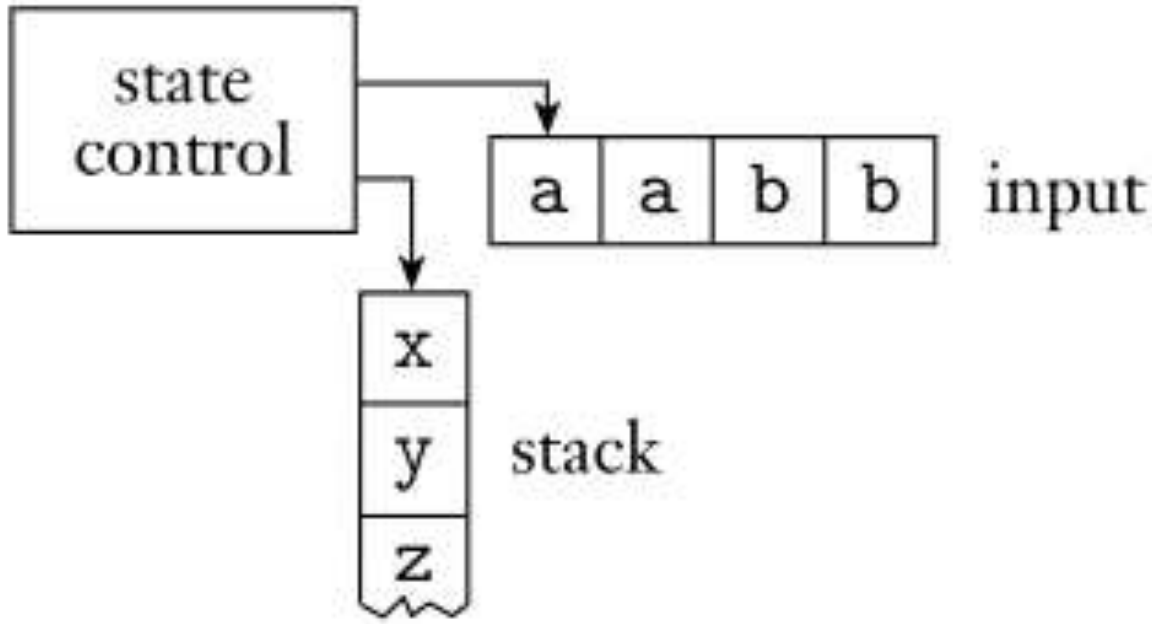


FIGURE 2.12
Schematic of a pushdown automaton

Pushdown Automata

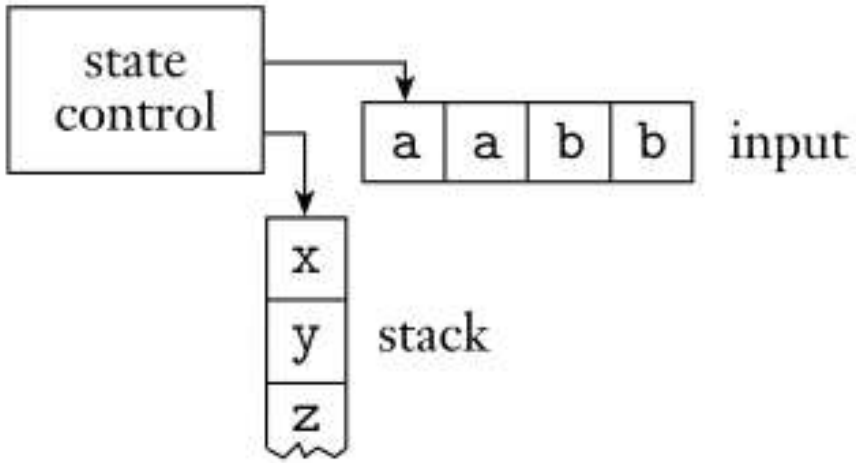


FIGURE 2.12
Schematic of a pushdown automaton

- A pushdown automaton (PDA) can write symbols on the stack and read them back later.
- Writing a symbol “pushes down” all the other symbols on the stack.

Pushdown Automata

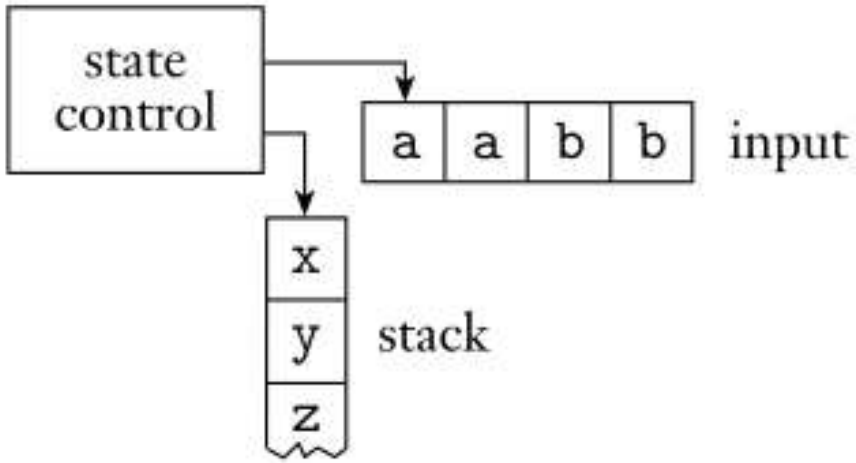


FIGURE 2.12
Schematic of a pushdown automaton

- At any time the symbol on the top of the stack can be read and removed.
- The remaining symbols then move back up.

Pushdown Automata

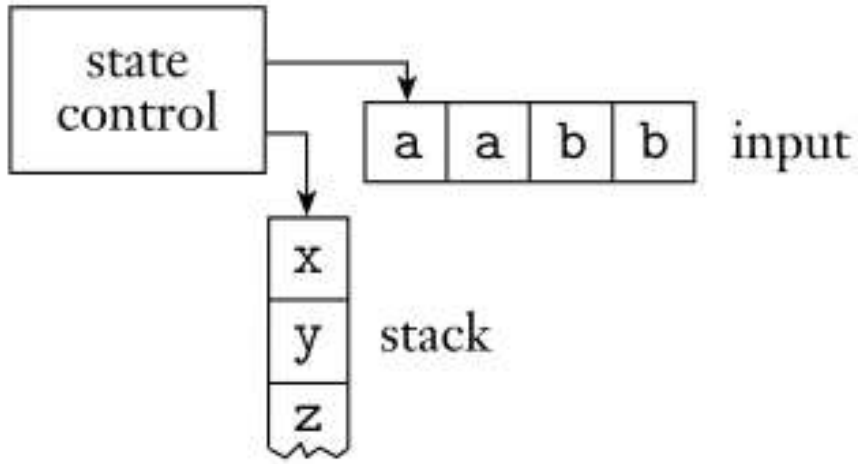


FIGURE 2.12
Schematic of a pushdown automaton

- Writing a symbol on the stack is often referred to as pushing the symbol, and removing a symbol is referred to as popping it.
- Note that all access to the stack, for both reading and writing, may be done only at the top.
- In other words a stack is a “last in, first out” storage device.

Pushdown Automata

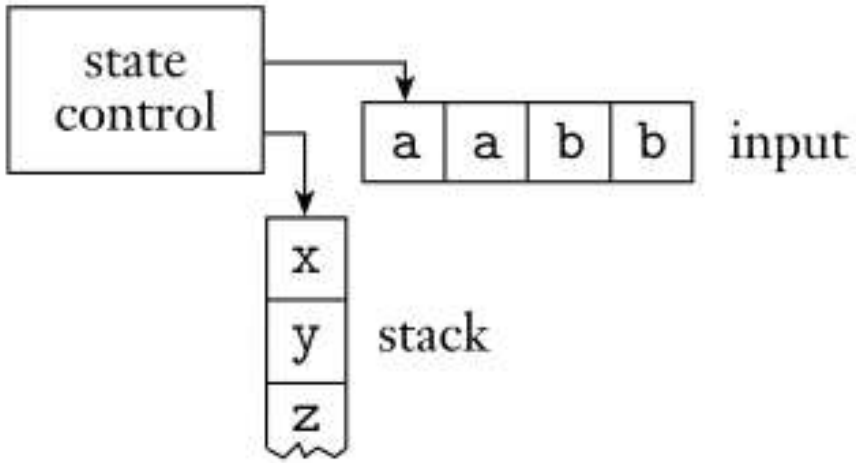


FIGURE 2.12
Schematic of a pushdown automaton

- If certain information is written on the stack and additional information is written afterward, the earlier information becomes inaccessible until the later information is removed.

Pushdown Automata

- $L = \{0^n 1^n \mid n \geq 0\}$
- A finite automaton is unable to recognize the language L because it cannot store very large numbers in its finite memory
- A PDA is able to recognize this language because it can use its stack to store the number of 0s it has seen.
- Thus the unlimited nature of a stack allows the PDA to store numbers of unbounded size.

Pushdown Automata

- $L = \{0^n 1^n \mid n \geq 0\}$
- The following informal description shows how the automaton for this language works.
- Read symbols from the input.
- As each 0 is read, push it onto the stack.
- As soon as 1s are seen, pop a 0 off the stack for each 1 read.
- If reading the input is finished exactly when the stack becomes empty of 0s, accept the input.
- If the stack becomes empty while 1s remain or if the 1s are finished while the stack still contains 0s or if any 0s appear in the input following 1s, reject the input.

Formal Definition Pushdown Automata

DEFINITION 2.13

A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

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

Formal Definition Pushdown Automata

- Similar to DFA except the stack
- Input alphabet Σ and stack alphabet Γ
- Transition function: $Qx\ \Sigma_\varepsilon\ x\ \Gamma_\varepsilon$, current state, next input symbol and top symbol of the stack determines next move
- The output of transition function return member of Q together with $\Gamma\varepsilon$ i.e $Q\ x\ \Gamma\varepsilon$

Formal Definition Pushdown Automata

- A pushdown automaton M = (Q, Σ, Γ, δ q_0, F) computes as follows.
  It accepts input w if w can be written as w = $w_1 w_2 \ldots\ldots w_m$
  where each $w_i \in \Sigma_\varepsilon$ and sequences of states $r_1, r_2 \ldots\ldots rm \in$ Q and strings $s_0, s_1, \ldots\ldots sm \in \Gamma^*$ exist that satisfy the following three conditions.

1. $r_0 = q_0$ and $s_0 = \varepsilon$. This condition signifies that M starts out properly, in the start state and with an empty stack.
2. For $i = 0, \ldots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol.
3. $r_m \in F$. This condition states that an accept state occurs at the input end.

Example Pushdown Automata

EXAMPLE 2.14

The following is the formal description of the PDA (page 112) that recognizes the language $\{0^n 1^n | \, n \geq 0\}$. Let M_1 be $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where

$Q = \{q_1, q_2, q_3, q_4\}$,

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

$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$, and

δ is given by the following table, wherein blank entries signify $\emptyset$.

Input:	0			1			ε		
Stack:	0	$	ε	0	$	ε	0	$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									

Example Pushdown Automata

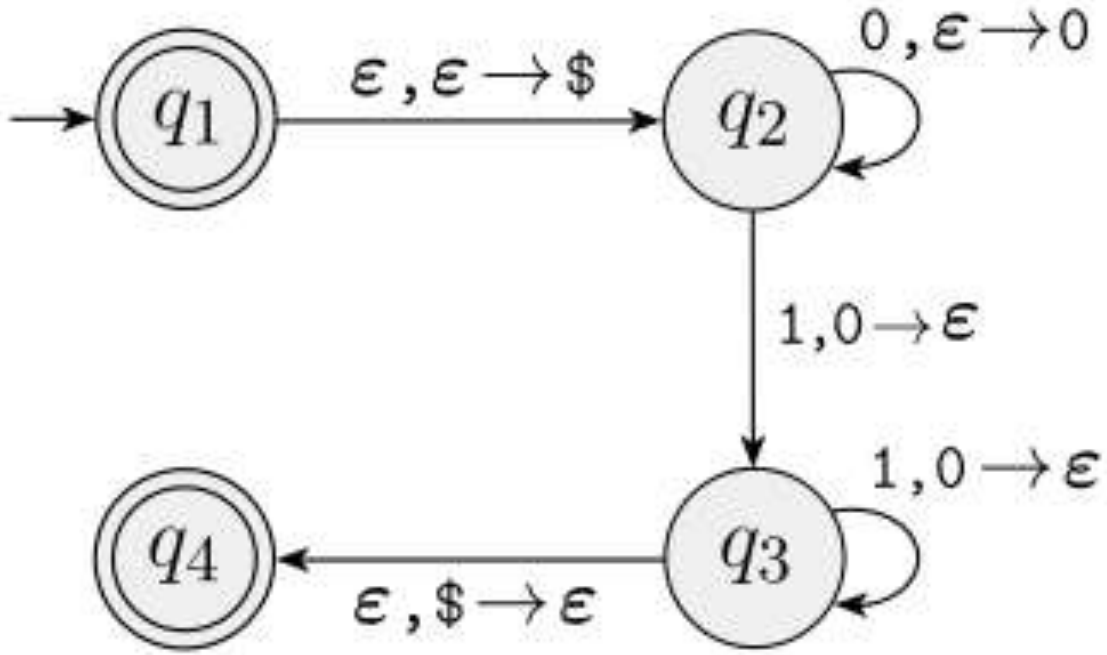


FIGURE 2.15
State diagram for the PDA M_1 that recognizes $\{0^n 1^n \mid n \geq 0\}$

Example Pushdown Automata

This example illustrates a pushdown automaton that recognizes the language $\{a^i b^j c^k \mid i = j; \text{ or } i = k\}$

- First reading and pushing a's, when a's are done then match them either with b's or c's
- Nondeterminism is handy here

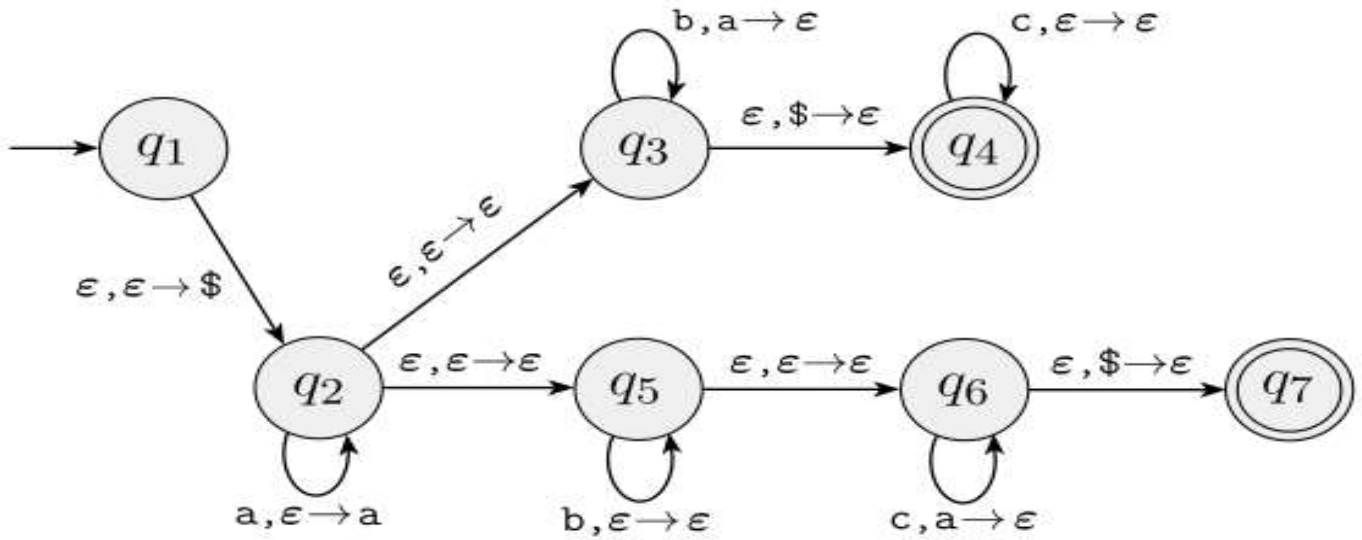


FIGURE 2.17
State diagram for PDA M_2 that recognizes
$\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$

Example Pushdown Automata

In this example we give a PDA M_3 recognizing the language $\{ww^R \mid w \epsilon \{0,1\}^*\}$.

- ww^R means w written backwards.

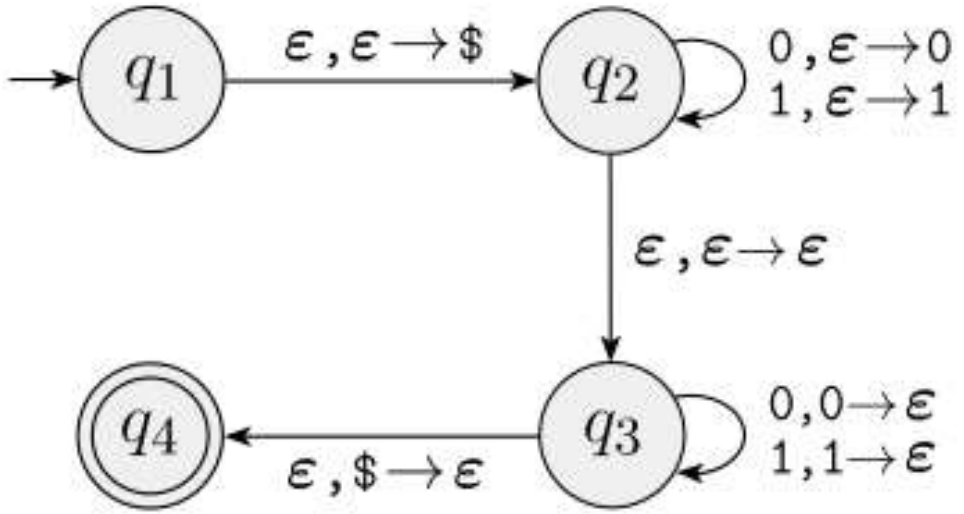


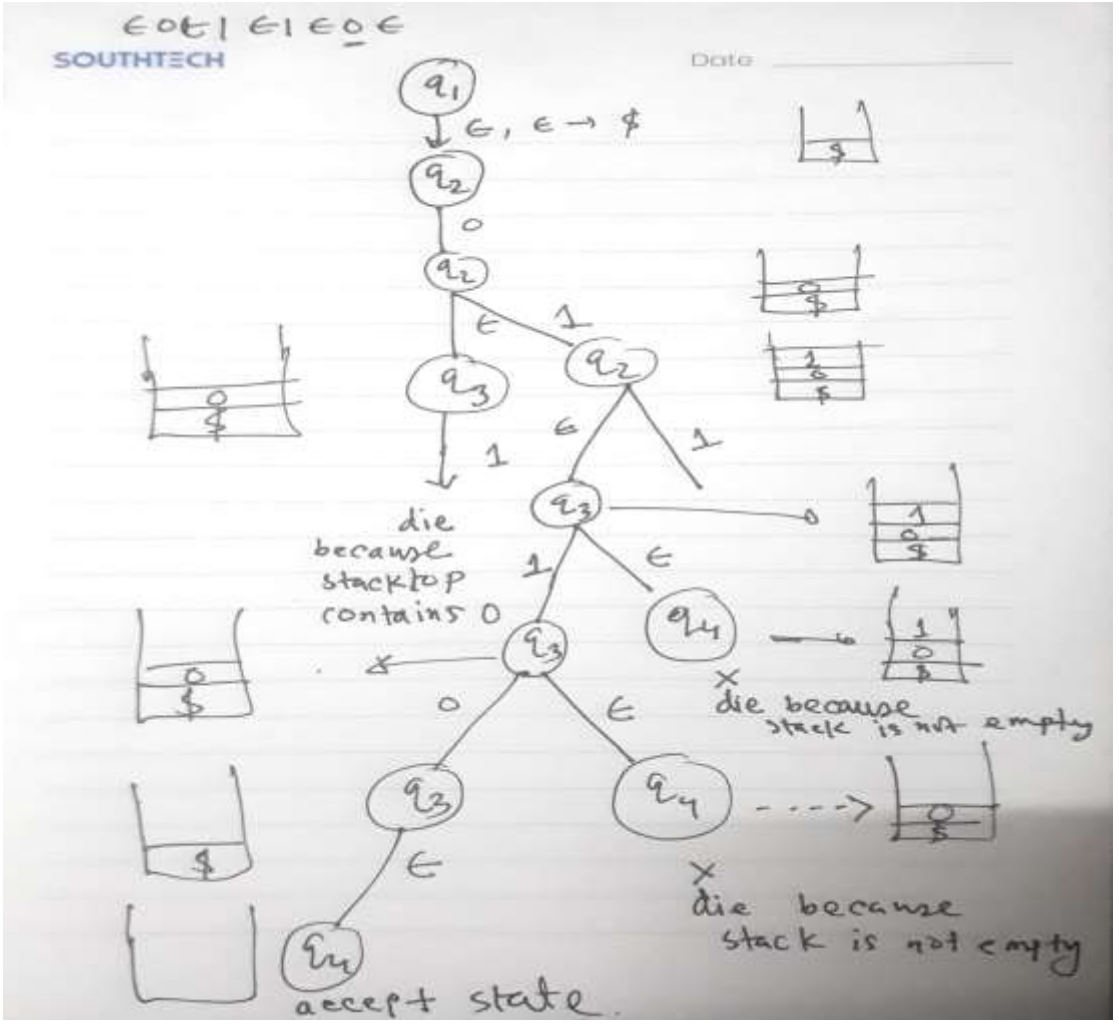
FIGURE 2.19

State diagram for the PDA M_3 that recognizes $\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$

Example Pushdown Automata

In this example we give a PDA M_3 recognizing the language $\{ww^R \mid w \epsilon \{0,1\}^*\}$.

- ww^R means w written backwards.



Reference Books

REFERENCE BOOKS
1. Introduction to the Theory of Computation, 3rd edition, 2012- Michael Sipser.
2. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2008 - J. E. Hopcroft, R. Motwani, and J. D. Ullman.
3. Elements of the Theory of Computation. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd edition, 1997- H. R. Lewis and C. H. Papadimitriou.

Chomsky Normal Form

We introduce Chomsky Normal Form, which is used to answer questions about context-free languages

Chomsky Normal Form

Chomsky Normal Form. *A grammar where every production is either of the form* $A \to BC$ *or* $A \to \mathtt{c}$ *(where* A*,* B*,* C *are arbitrary variables and* $\mathtt{c}$ *an arbitrary symbol).*

Example:

$$S \to AS \mid \mathtt{a}$$
$$A \to SA \mid \mathtt{b}$$

(If language contains ε, then we allow $S \to \varepsilon$ where S is start symbol, and forbid S on RHS.)

Why Chomsky Normal Form?

The key advantage is that in Chomsky Normal Form, every derivation of a string of n letters has exactly $2n - 1$ steps.

Thus: *one can determine if a string is in the language by exhaustive search of all derivations.*

Conversion

The conversion to Chomsky Normal Form has four main steps:

1. Get rid of all ε productions.
2. Get rid of all productions where RHS is one variable.
3. Replace every production that is too long by shorter productions.
4. Move all terminals to productions where RHS is one terminal.

1) Eliminate ε Productions

Determine the nullable variables (those that generate ε) (algorithm given earlier).

Go through all productions, and for each, omit every possible subset of nullable variables.

For example, if $P \to A\mathtt{x}B$ with both A and B nullable, add productions $P \to \mathtt{x}B \mid A\mathtt{x} \mid \mathtt{x}$.

After this, delete all productions with empty RHS.

2) Eliminate Variable Unit Productions

A unit production is where RHS has only one symbol.

Consider production $A \to B$. Then for every production $B \to \alpha$, add the production $A \to \alpha$. Repeat until done (but don't re-create a unit production already deleted).

3) Replace Long Productions by Shorter Ones

For example, if have production $A \to BCD$, then replace it with $A \to BE$ and $E \to CD$.

(In theory this introduces many new variables, but one can re-use variables if careful.)

4) Move Terminals to Unit Productions

For every terminal on the right of a non-unit production, add a substitute variable.

For example, replace production $A \to \texttt{b}C$ with productions $A \to BC$ and $B \to \texttt{b}$.

Example

Consider the CFG:

$S \to \mathtt{a}X\mathtt{b}X$

$X \to \mathtt{a}Y \mid \mathtt{b}Y \mid \varepsilon$

$Y \to X \mid \mathtt{c}$

The variable X is nullable; and so therefore is Y.
After elimination of ε, we obtain:

$S \to \mathtt{a}X\mathtt{b}X \mid \mathtt{ab}X \mid \mathtt{a}X\mathtt{b} \mid \mathtt{ab}$

$X \to \mathtt{a}Y \mid \mathtt{b}Y \mid \mathtt{a} \mid \mathtt{b}$

$Y \to X \mid \mathtt{c}$

Example: Step 2

After elimination of the unit production $Y \to X$, we obtain:

$S \to \mathtt{a}X\mathtt{b}X \mid \mathtt{ab}X \mid \mathtt{a}X\mathtt{b} \mid \mathtt{ab}$
$X \to \mathtt{a}Y \mid \mathtt{b}Y \mid \mathtt{a} \mid \mathtt{b}$
$Y \to \mathtt{a}Y \mid \mathtt{b}Y \mid \mathtt{a} \mid \mathtt{b} \mid \mathtt{c}$

Example: Steps 3 & 4

Now, break up the RHSs of S; and replace a by A, b by B and c by C wherever not units:

$S \to EF \mid AF \mid EB \mid AB$
$X \to AY \mid BY \mid \mathtt{a} \mid \mathtt{b}$
$Y \to AY \mid BY \mid \mathtt{a} \mid \mathtt{b} \mid \mathtt{c}$
$E \to AX$
$F \to BX$
$A \to \mathtt{a}$
$B \to \mathtt{b}$
$C \to \mathtt{c}$

Practice

Convert the following CFG into Chomsky Normal Form:

$$S \to A\mathtt{b}A$$
$$A \to A\mathtt{a} \mid \varepsilon$$

Solution to Practice

After the first step, one has:

$$S \to A\texttt{b}A \mid \texttt{b}A \mid A\texttt{b} \mid \texttt{b}$$
$$A \to A\texttt{a} \mid \texttt{a}$$

The second step does not apply. After the third step, one has:

$$S \to TA \mid \texttt{b}A \mid A\texttt{b} \mid \texttt{b}$$
$$A \to A\texttt{a} \mid \texttt{a}$$
$$T \to A\texttt{b}$$

Solution Continued

And finally, one has:

$$
\begin{aligned}
S &\to TA \mid BA \mid AB \mid \texttt{b} \\
A &\to AC \mid \texttt{a} \\
T &\to AB \\
B &\to \texttt{b} \\
C &\to \texttt{a}
\end{aligned}
$$

Summary

There are special forms for CFGs such as Chomsky Normal Form, where every production has the form $A \to BC$ or $A \to c$. The algorithm to convert to this form involves (1) determining all nullable variables and getting rid of all ε-productions, (2) getting rid of all variable unit productions, (3) breaking up long productions, and (4) moving terminals to unit productions.

Welcome

Course Code: CSE 217

Course Name: Theory of Computation

Credit Hour: 3.00

Instructor: Maj Mahbub

Outline

- Non-context-free languages
- The pumping lemma for context-free languages and proofs

Non-context-free languages

- Sometimes it is difficult to find a context free grammar for a language.
- For example, try to find a CFG for L = $\{a^n b^n c^n \mid n >= 0\}$

- If we suspect that L might not be context-free and we want to prove it.
- Find some property that context-free languages have to have and show that L does not have it
- Pumping Lemma for Context Free Languages

Pumping Lemma for Context Free Languages

- If A is Context Free Language - then there is a number p (pumping length) where, if s is any string in A (at least length p) then s may be divided into 5 pieces s = uvxyz satisfying these conditions:
  - For each $i \geq 0$, $uv^ixy^iz \in A$
  - $|vy| > 0$ and,
  - $|vxy| \leq p$

- When s being divided into uvxyz, condition 2 says either v or y is not be empty string
- Condition 3 states that the pieces v, x and y together have the length at most p

Pumping Lemma for Context Free Languages

Use the pumping lemma show that the language B = {$a^n b^n c^n$ | n >= 0} is not context free

- Let us assume that B is a CFL and let p is pumping length for B
- Select the string s=$a^p b^p c^p$, s is member of B with length at least p
- According to pumping lemma s can be pumped, if we can show that s cant be pumped then it will be proved that B is not context free.
- Let us divide s into five pieces uvxyz

Pumping Lemma for Context Free Languages

- Condition 2 of pumping lemma says that, v and y is non empty then we can consider one of two cases depending on whether substrings v and y contain more than one type of alphabet symbol

1. When both v and y contain more than one type of alphabet symbol, v does not contain both a's and b's or both b's or c's and same hold for y. In this case the string uv^2xy^2z can not contain equal number of a's, b's and c's. Thus s cant be member of B and we have got a contradiction

Pumping Lemma for Context Free Languages

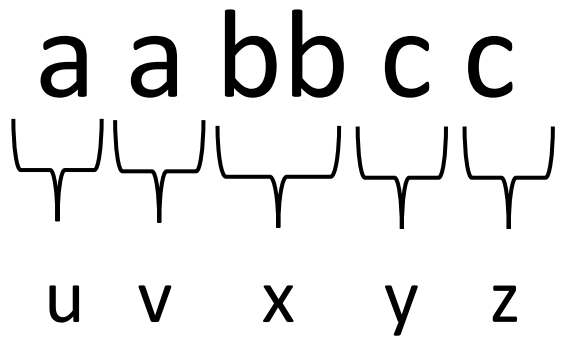
2. When either v or y contain more than one type of symbol uv^2xy^2z may contain equal number of three alphabet symbols but wont contain them in a correct order. Hence it cant be a member of B and a contradiction occur.

- Thus we have proved that B is not a CFL

Pumping Lemma for Context Free Languages

Let p=2, $s=a^2b^2c^2=aabbcc$

- Case-1: both v and y contain more than one type of alphabet

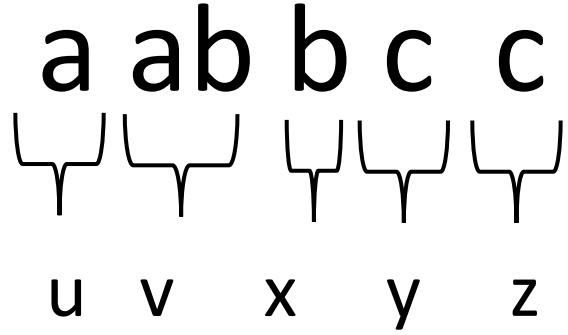


- Now for i=2, $uv^2xy^2z=aaabbccc=a^3b^2c^3 \notin B$

Pumping Lemma for Context Free Languages

Let $p=2$, $s=a^2b^2c^2=aabbcc$

- Case-2: either v or y contain more than one type of symbol



- Now for i=2, $uv^2xy^2z=aababbccc=a^3b^3c^3 \notin B$ and is not in correct order.

Reference Books

REFERENCE BOOKS
1. Introduction to the Theory of Computation, 3rd edition, 2012- Michael Sipser.
2. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2008 - J. E. Hopcroft, R. Motwani, and J. D. Ullman.
3. Elements of the Theory of Computation. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd edition, 1997- H. R. Lewis and C. H. Papadimitriou.

Welcome

Course Code: CSE 217

Course Name: Theory of Computation

Credit Hour: 3.00

Instructor: Maj Mahbub

Outline

- Turing Machines
- Formal Definition of Turing Machines
- Example of Turing Machines

Turing Machine

- We turn now to a much more powerful model, first proposed by Alan Turing in 1936, called the Turing machine.
- Similar to a finite automaton but with an unlimited and unrestricted memory, a Turing machine is a much more accurate model of a general purpose computer.
- A Turing machine can do everything that a real computer can do.
- Nonetheless, even a Turing machine cannot solve certain problems.
- In a very real sense, these problems are beyond the theoretical limits of computation.

Turing Machine

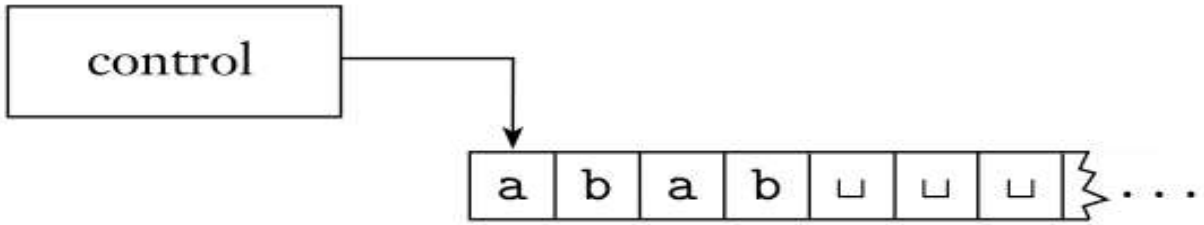


FIGURE 3.1
Schematic of a Turing machine

- The Turing machine model uses an infinite tape as its unlimited memory.
- It has a tape head that can read and write symbols and move around on the tape.
- Initially the tape contains only the input string and is blank everywhere else.

Turing Machine

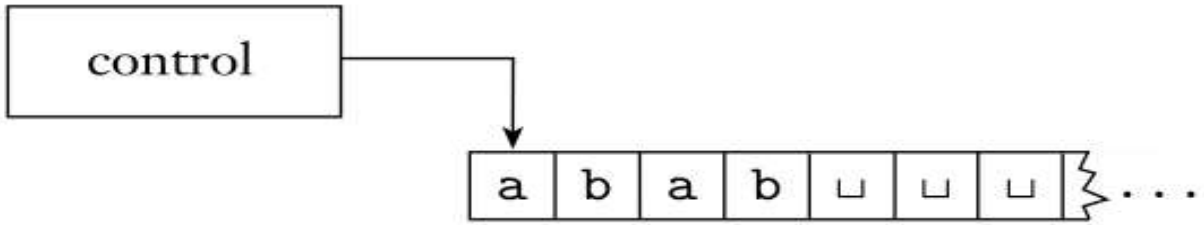


FIGURE 3.1
Schematic of a Turing machine

- If the machine needs to store information, it may write this information on the tape.
- To read the information that it has written, the machine can move its head back over it.
- The machine continues computing until it decides to produce an output.

Turing Machine

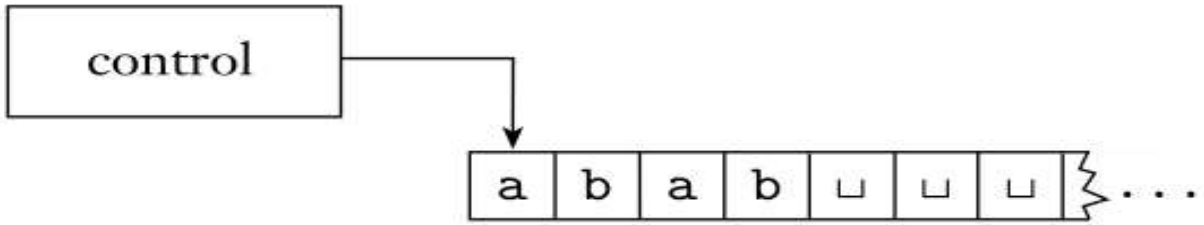


FIGURE 3.1
Schematic of a Turing machine

- The outputs accept and reject are obtained by entering designated accepting and rejecting states.
- If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting.

Difference between Turing Machine and Finite Automata

- The following list summarizes the differences between finite automata and Turing machines.

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

Turing Machine

- $\{w\#w \mid w \in \{0,1\}^*\}$

- Let's introduce a Turing machine M_1 for testing membership in the language B = $\{w\#w \mid w \in \{0,1\}^*\}$.
- We want M_1 to accept if its input is a member of B and to reject otherwise.

Turing Machine

- $\{w\#w \mid w \in \{0,1\}^*\}$

- To understand M_1 better, put yourself in its place by imagining that you are standing on a mile-long input consisting of millions of characters.
- Your goal is to determine whether the input is a member of B — that is, whether the input comprises two identical strings separated by a # symbol.

Turing Machine

- $\{w\#w \mid w \in \{0,1\}^*\}$

- The input is too long for you to remember it all, but you are allowed to move back and forth over the input and make marks on it.
- The obvious strategy is to zig-zag to the corresponding places on the two sides of the # and determine whether they match.
- Place marks on the tape to keep track of which places correspond.

Turing Machine

- $\{w\#w \mid w \in \{0,1\}^*\}$

- We design M_1 to work in that way.
- It makes multiple passes over the input string with the read - write head.
- On each pass it matches one of the characters on each side of the # symbol.
- To keep track of which symbols have been checked already, M_1 crosses off each symbol as it is examined.
- If it crosses off all the symbols, that means that everything matched successfully, and M_1 goes into an accept state.

Turing Machine

- $\{w\#w \mid w \in \{0,1\}^*\}$

- If it discovers a mismatch, it enters a reject state.

Turing Machine

- M_1 = On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol.
   - If they do not, or if no # is found, reject.
   - Cross off symbols as they are checked to keep track of which symbols correspond.
2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #.
   - If any symbols remain, reject ; otherwise, accept.

Turing Machine

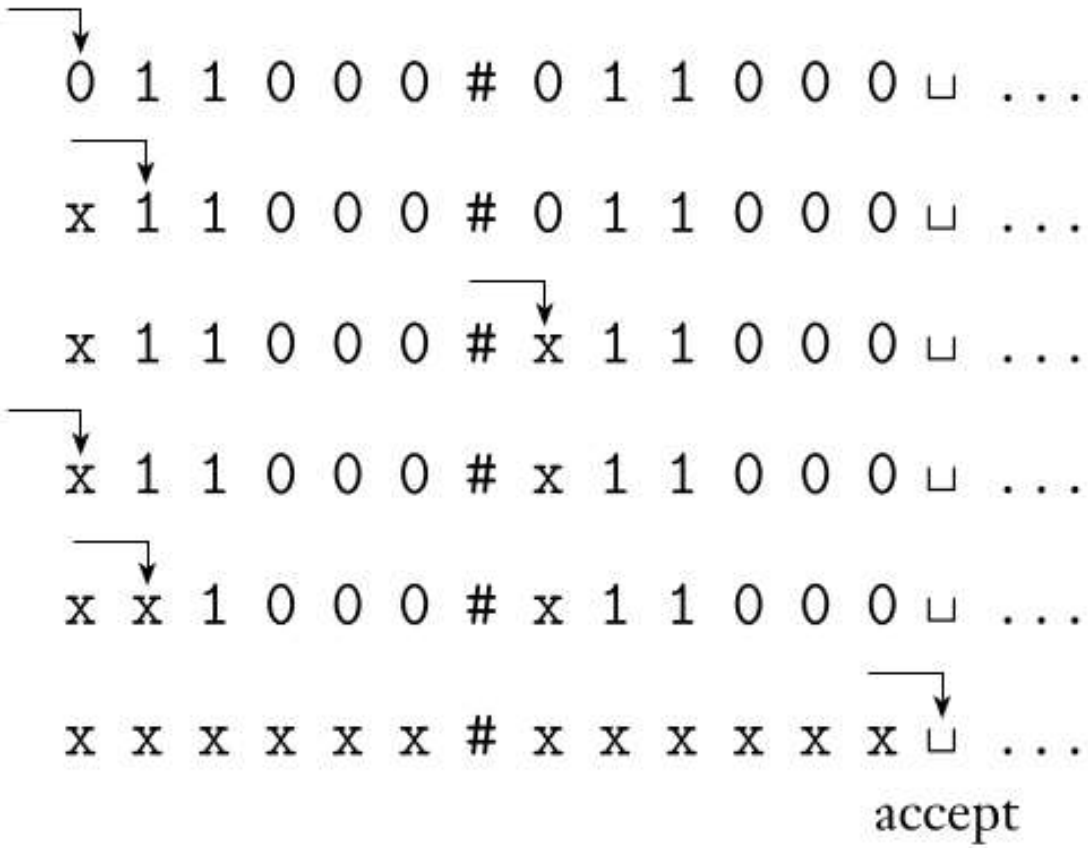
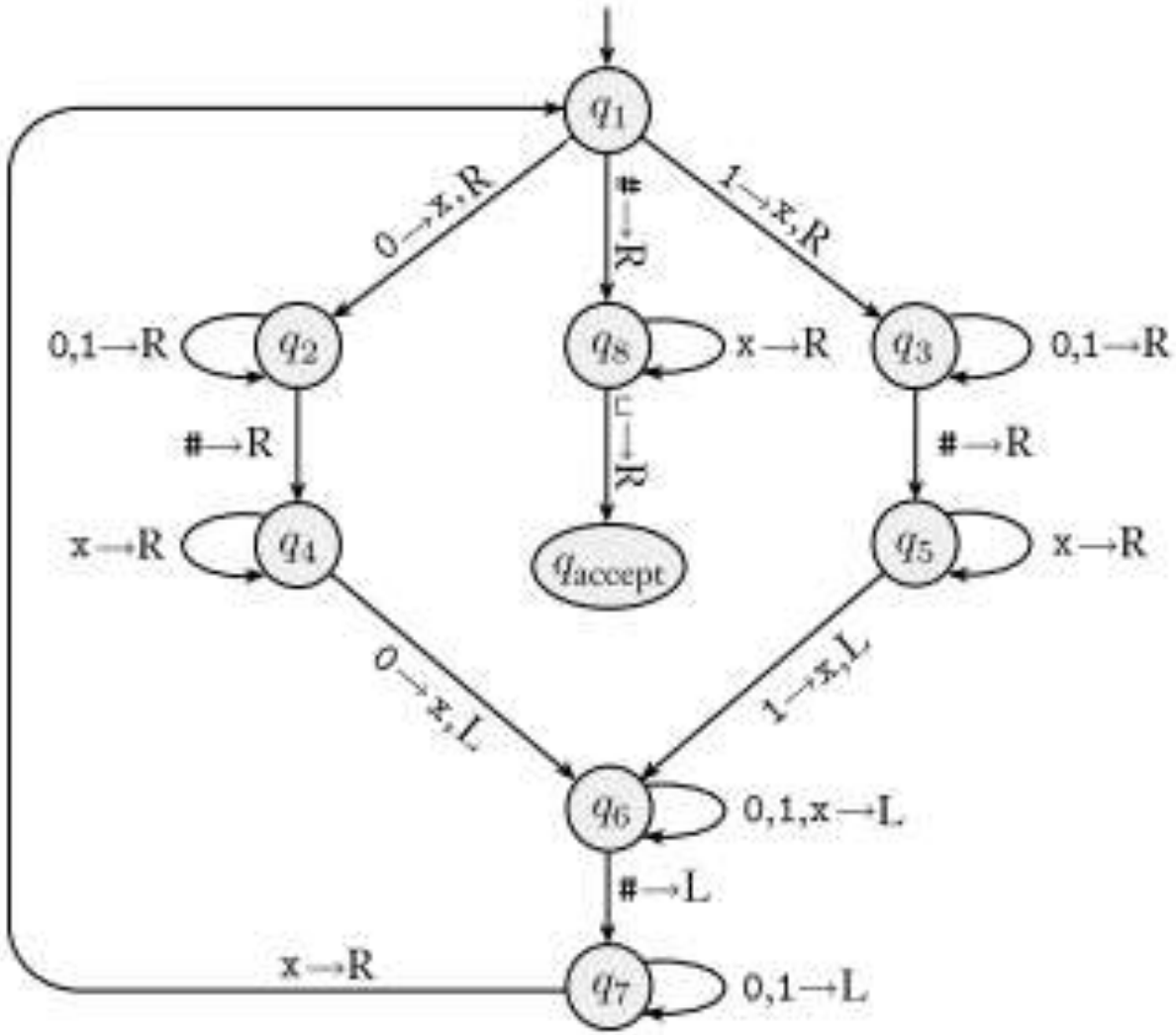


FIGURE 3.2
Snapshots of Turing machine M_1 computing on input 011000#011000

Turing Machine



Turing Machine

- Design a TM that decides A={0^{2^n} | n$\geq$0}

- M_2 = On input string w:
  1. Sweep left to right across the tape, crossing off every other 0.
  2. If in stage 1 the tape contained a single 0, accept.
  3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, reject.
  4. Return the head to the left-hand end of the tape.
  5. Go to stage 1.

Turing Machine

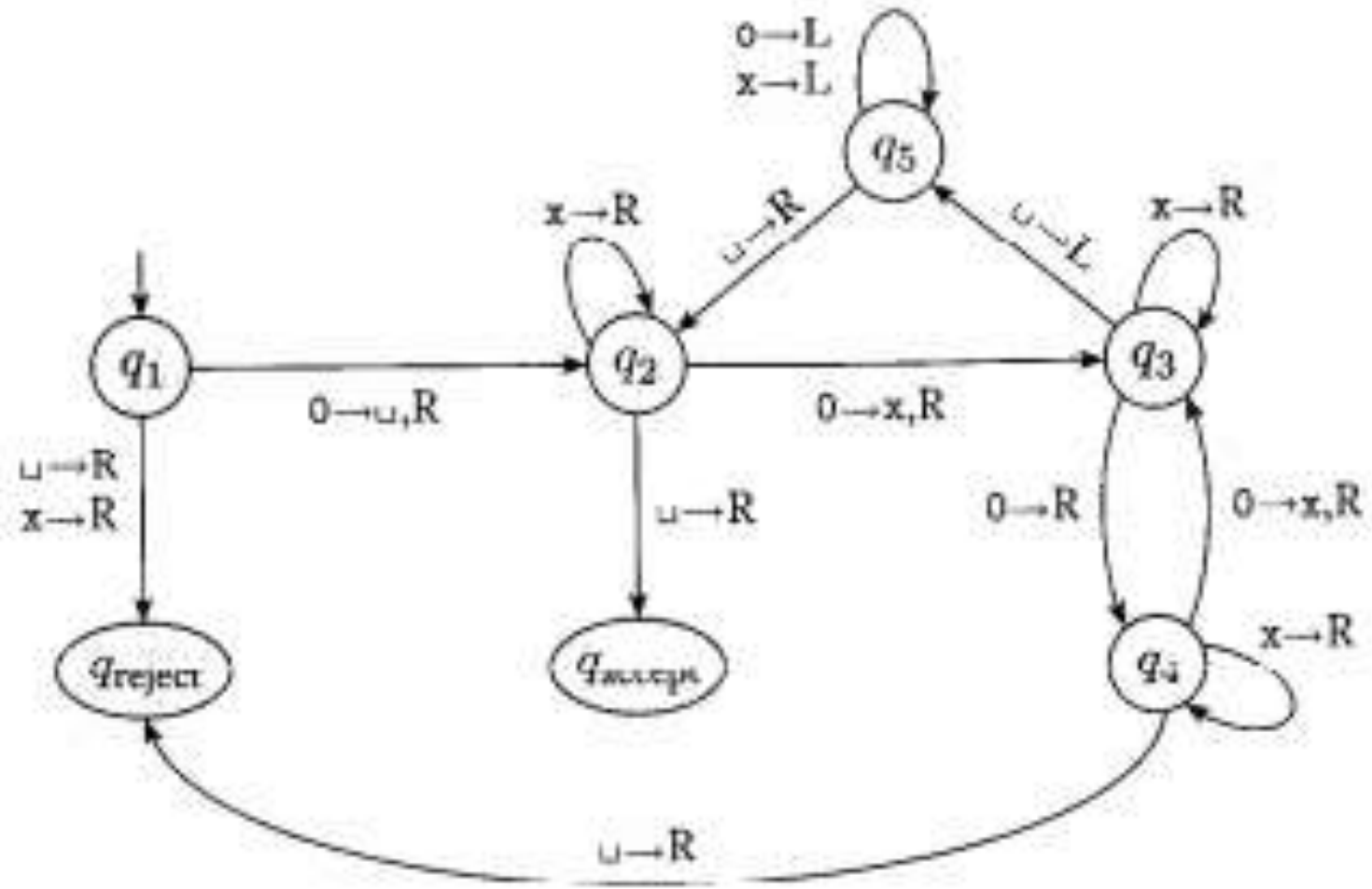
- Design a TM that decides A={0^{2^n} | n≥0}

- Each iteration of stage 1 cuts the number of 0s in half.
- As the machine sweeps across the tape in stage 1, it keeps track of whether the number of 0s seen is even or odd.
- If that number is odd and greater than 1, the original number of 0s in the input could not have been a power of 2.
- Therefore, the machine rejects in this instance.
- However, if the number of 0s seen is 1, the original number must have been a power of 2.
- So in this case, the machine accepts.

Turing Machine

$q_1 0000$
$\sqcup q_2 000$
$\sqcup x q_3 00$
$\sqcup x 0 q_4 0$
$\sqcup x 0 x q_3 \sqcup$
$\sqcup x 0 q_5 x \sqcup$
$\sqcup x q_5 0 x \sqcup$
$\sqcup q_5 x 0 x \sqcup$
$q_5 \sqcup x 0 x \sqcup$
$\sqcup q_2 x 0 x \sqcup$
$\sqcup x q_2 0 x \sqcup$
$\sqcup x x q_3 x \sqcup$
$\sqcup x x x q_3 \sqcup$
$\sqcup x x q_5 x \sqcup$
$\sqcup x q_5 x x \sqcup$
$\sqcup q_5 x x x \sqcup$
$q_5 \sqcup x x x \sqcup$
$\sqcup q_2 x x x \sqcup$
$\sqcup x q_2 x x \sqcup$
$\sqcup x x q_2 x \sqcup$
$\sqcup x x x q_2 \sqcup$
$\sqcup x x x \sqcup q_{\text{accept}}$

Turing Machine



Formal Definition of Turing Machine

DEFINITION 3.3

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

Formal Definition of Turing Machine

- The heart of the definition of a Turing machine is the transition function δ because it tells us how the machine gets from one step to the next.
- For a Turing machine, δ takes the form:

  $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L \times R\}$

Formal Definition of Turing Machine

- That is, when the machine is in a certain state *q* and the head is over a tape square containing a symbol *a*, and if $\delta(q, a) = (r , b,L)$, the machine writes the symbol *b* replacing the *a*, and goes to state *r* .
- The third component is either L or R and indicates whether the head moves to the left or right after writing.
- In this case, the L indicates a move to the left.

Reference Books

REFERENCE BOOKS
1. Introduction to the Theory of Computation, 3rd edition, 2012- Michael Sipser.
2. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2008 - J. E. Hopcroft, R. Motwani, and J. D. Ullman.
3. Elements of the Theory of Computation. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd edition, 1997- H. R. Lewis and C. H. Papadimitriou.

Welcome

Course Code: CSE 217

Course Name: Theory of Computation

Credit Hour: 3.00

Instructor: Maj Mahbub

Outline

- Decidability
- Decidable Languages
- Decidable problems concerning regular languages

Decidability

- The ability to determine whether a particular problem or language can be solved or recognized by a computer program or machine.
- The existence of an algorithm or procedure that can provide a definite answer (yes or no) for every instance of the problem.

Decidability

Formal Definition:
A language L is decidable if there exists a Turing machine (or any equivalent computational model) that, given any input string w, will halt and accept if w is a member of L, and halt and reject if w is not a member of L.

This means that for every possible input, the Turing machine will always produce a correct and unambiguous result, indicating whether the input belongs to the language or not.

Decidable Languages

- Language for which there exists an algorithmic procedure that can determine whether any given input string belongs to the language or not.
- In other words- a language that can be recognized by a Turing machine that always halts and produces the correct answer.
- **Formal Definition:**

A language L is decidable if there exists a Turing machine (or any equivalent computational model) that, given any input string w, will halt and accept if w is a member of L, and halt and reject if w is not a member of L.

This means that there is a well-defined algorithm or procedure that can be followed to make a definite determination for any input.

Decidable Problems Concerning Regular Languages

- Decidability in the context of regular languages provides theoretical foundations for designing algorithms and tools used in various areas, such as lexical analysis in programming languages, string matching, and pattern recognition.
- It also serves as a building block for understanding more complex languages and their associated computational models, such as context-free and recursively enumerable languages.

Decidable Problems Concerning Regular Languages

- The acceptance problem for DFAs of testing whether a particular finite automaton accepts a given string can be expressed as a language, A_{DFA}.
- This language contains the encodings of all DFAs together with strings that the DFAs accept.

Let A_{DFA} = {(B,w) B is a DFA that accepts input string w}.

- The problem of testing whether a DFA B accepts an input w is the same as the problem of testing whether (B. w) is a member of the language A_{DFA}.

Proof: A_{DFA} is a decidable language

- Proof idea is simple like we need to represent a TM M that decides A_{DFA}

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

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

PROOF We mention just a few implementation details of this proof. For those of you familiar with writing programs in any standard programming language, imagine how you would write a program to carry out the simulation.

First, let's examine the input $\langle B, w \rangle$. It is a representation of a DFA B together with a string w. One reasonable representation of B is simply a list of its five components, Q, Σ, δ, q_0, and F. When M receives its input, M first checks on whether it properly represents a DFA B and a string w. If not, M rejects.

Then M carries out the simulation in a direct way. It keeps track of B's current state and B's current position in the input w by writing this information down on its tape. Initially, B's current state is q_0 and B's current input position is the leftmost symbol of w. The states and position are updated according to the specified transition function δ. When M finishes processing the last symbol of w, M accepts the input if B is in an accepting state; M rejects the input if B is in a nonaccepting state.

Proof: A_{NFA} is a decidable language

THEOREM 4.2

A_{NFA} is a decidable language.

PROOF We present a TM N that decides A_{NFA}. We could design N to operate like M, simulating an NFA instead of a DFA. Instead, we'll do it differently to illustrate a new idea: have N use M as a subroutine. Because M is designed to work with DFAs, N first converts the NFA it receives as input to a DFA before passing it to M.

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

1. Convert NFA B to an equivalent DFA C using the procedure for this conversion given in Theorem 1.19.
2. Run TM M from Theorem 4.1 on input $\langle C, w \rangle$.
3. If M accepts, *accept*; otherwise *reject*."

Running TM M in stage 2 means incorporating M into the design of N as a subprocedure.

Proof: A_{NFA} is a decidable language

PROOF We present a TM N that decides A_{NFA}. We could design N to operate like M, simulating an NFA instead of a DFA. Instead, we'll do it differently to illustrate a new idea: have N use M as a subroutine. Because M is designed to work with DFAs, N first converts the NFA it receives as input to a DFA before passing it to M.

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

1. Convert NFA B to an equivalent DFA C, using the procedure for this conversion given in Theorem 1.39.
2. Run TM M from Theorem 4.1 on input $\langle C, w \rangle$.
3. If M accepts, *accept*; otherwise, *reject*."

Running TM M in stage 2 means incorporating M into the design of N as a subprocedure.

A_{REX} is a decidable language.

PROOF The following TM P decides A_{REX}.

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

1. Convert regular expression R to an equivalent NFA A by using the procedure for this conversion given in Theorem 1.54.
2. Run TM N on input $\langle A, w\rangle$.
3. If N accepts, *accept*; if N rejects, *reject*."

Theorems 4.1, 4.2, and 4.3 illustrate that, for decidability purposes, presenting the Turing machine with a DFA, NFA, or regular expression are all equivalent because the machine is able to convert one form of encoding to another.

Now we turn to a different kind of problem concerning finite automata: *emptiness testing* for the language of a finite automaton. In the preceding three theorems we had to determine whether a finite automaton accepts a particular string. In the next proof we must determine whether a finite automaton accepts any strings at all. Let

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

Reference Books

REFERENCE BOOKS
1. Introduction to the Theory of Computation, 3rd edition, 2012- Michael Sipser.
2. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2008 - J. E. Hopcroft, R. Motwani, and J. D. Ullman.
3. Elements of the Theory of Computation. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd edition, 1997- H. R. Lewis and C. H. Papadimitriou.